A Recursive Data Structure: Trees

Master MoSIG — Algorithms and Program Design

Florent Bouchez Tichadou — Gwenaël Delaval
{florent.bouchez-tichadou,gwenael.delaval}@imag.fr

September 18th, 2015

Objectives:

- Know basic recursive data-structures: lists, trees, binary trees;
- Being able to create trees and use them to store information;
- Being able to search information in trees;
- Know the complexities of classical algorithms using trees.

1 Recursive Data Structures

Contrary to static and linear data structures like arrays, recursive data structure are dynamic in essence and not restricted to one dimension. They are often used to store in memory objects that have dynamic components (i.e., creation and deletion or re-arrangement of components are common) and/or whose components have a non-linear hierarchy (e.g., nodes in a graph).

1.1 Lists

Lists are the most simple of recursive data structures. Their arrangement is linear, but compared to arrays, it is much more easier to insert elements in the middle of a list. We will not go in details over lists as this is a notion that should already be clear for everyone. If not, it is an absolute necessity to get up-to-date with this notion, by reading about it in a book or for instance `https://en.wikipedia.org/wiki/List_(abstract_data_type)`. We will just do some quick cost comparison.

	insertion*	search	sorted search	deletion†
list	$O(1)$	$O(n)$	$O(n)$	$O(1)$
array	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$

* Assuming insertion at a random position.
† Not including the cost of searching for the element to delete.

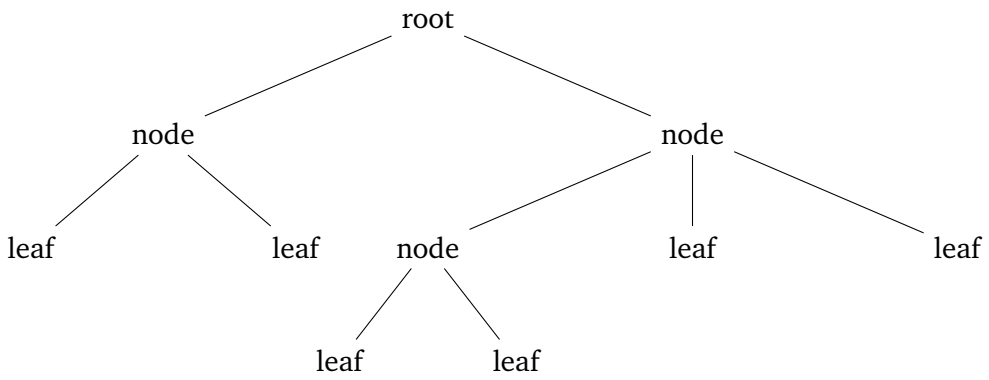
It should be clear with this example that choosing the right data structure for a problem heavily depends on the usage that will be made of the data. In particular, the complexity of an algorithm depends on what data structure it uses internally.

1.2 Trees

Trees are the second most common type of recursive data structures. The rest of these notes will cover various types of trees, as well as various kinds of operations on trees.

Trees are very similar to lists, the only difference being that an element in a tree, i.e., a *node*, may have more than one successor. There is usually a fixed maximum number of those successors—for instance, two in the case of *binary trees*—but this is not an obligation. Unless the tree is empty, there is a particular node, called the *root* of the tree, which is the entrance point to the structure. In a tree, the successors of a node are called its *children*, and a node with no successor is a *leaf*.

Sometimes, leaves also contain actual information in the data structures (for instance a number), sometimes, it is just their presence that is meaningful, marking the end of a branch of the tree.



1.3 Binary Trees

We will mainly cover binary trees in the rest of this document, i.e., trees in which nodes have at most two children, also called the *left* and *right* sons. Note that those are also the "roots" of the left and right subtree.

In the rest of the document, we will always present the algorithmic way (on the left side) along the actual implementation in the C language (on the right side). Note that it is possible to use the `calloc` function instead of the `malloc` but we preferred to explicit the setting to null pointers here.

1.3.1 Definition

```
1 Tree :=
2 | <empty>
3 | node of (value, tree, tree)
```

```
struct node {
  int value;
  struct node *left;
  struct node *right;
};
typedef struct node* tree;
```

1.3.2 Creation

```
1 root ← new node;
2 root.value ← 42;
```

```
struct node* new_node (int val) {
  struct node* n;
  n = malloc (sizeof (struct node));
  n->value = val;
  n->left = NULL;
  n->right = NULL;
  return n;
}

tree root = new_node (42);
```

1.3.3 Adding a node

We suppose here that we add a leaf—i.e., we are not inserting a node in the middle of the tree—as the left son of a node n.

```
1 x ← new node;
2 x.value ← 18;
3 n.left ← x;
```

```
struct node x = new_node (18);
assert (n != NULL && n->left == NULL);
n->left = x;
```

Complexity: $O(1)$

1.3.4 Deletion

```
1 delete x;
```

```
void delete_node (struct node *x) {
  assert (x->left != NULL && x->right != NULL
      );
  free (x);
}
```

Complexity: $O(1)$

1.3.5 Removing a node

Before deleting a node, care must be taken that we do not lose part of the tree. We suppose we know the father f of the node to remove n. We are removing a node that is not necessarily a leaf. If not, we are looking in the tree for a leaf to take the place of the node we want to remove, so that the left and right sons are still attached to the tree. In our case, we do not have any particular constraint so we can chose whichever leaf we want and we arbitrarily chose to take the left-most leaf of the subtree of n. Be aware that if your tree has a particular property that you need to maintain, you need to chose wisely the node with which you will replace n (see for instance the section on binary trees).

```
1  if n is a leaf then
2      if n is the left son of f then
3          f.left ← <empty>;
4      else
5          f.right ← <empty>;
6      delete n;
7  else
8      x ← n;
9      while x is not a leaf do
10         y ← x;      /* keep father of x */
11         if x.left ≠ <empty> then
12             x ← x.left;
13         else
14             x ← x.right;
       /* now exchange x and n          */
15     x.left ← n.left;
16     x.right ← n.right;
17     if n is the left son of f then
18         f.left ← x;
19     else
20         f.right ← x;
21     delete n;
       /* now x is not a child of y     */
22     if x is the left son of y then
23         y.left ← <empty>;
24     else
25         y.right ← <empty>;
```

Complexity: $O(\text{height})$

```
bool is_leaf (struct node *n) {
  return (n && !n->left && !n->right);
}

void remove_son (struct node *n, struct node *f) {
  assert (n && f);
  if (f->left == n) {
    f->left = NULL;
  } else {
    assert (f->right == n);
    f->right = NULL;
  }
}

void remove_node (struct node *n, struct node *f)
    {
  struct node *x, *y;

  if ( is_leaf (n)) {
    remove_son (n, f);
    delete_node (n);
    return ;
  }

  x = n; y = n;
  while (! is_leaf (x)) {
    y = x;
    if (x->left)
      x = x->left;
    else
      x = x->right;
  }

  x->left = n->left;
  x->right = n->right;
  n->left = NULL;
  n->right = NULL;

  if (f->left == n)
    f->left = x;
  else
    f->right = x;

  delete (n);
  remove_son (x, y);
}
```

1.4 Properties of Binary Trees

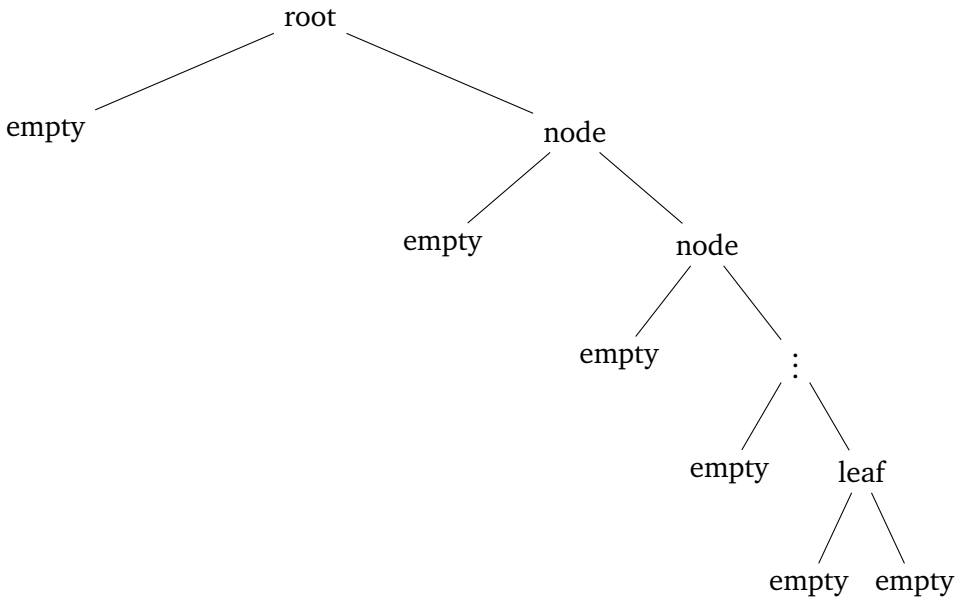
The *height* of a tree is defined as the number of nodes in the longest path from the root to a leaf. By definition, an empty tree has height zero, and a tree with a single node has height one. We can compute the height recursively as follows:

```
1 function height(n)
2 | n = <empty>→ 0
3 | otherwise → 1 + max (height (n.left),
  height (n.right))
```

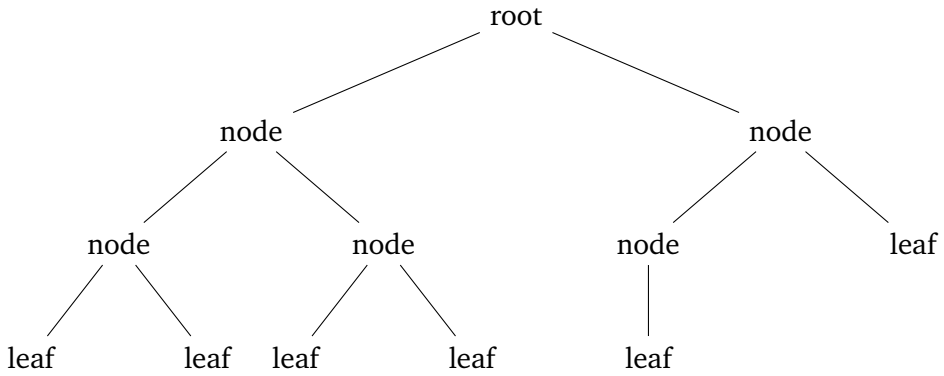
Complexity: $O(n)$

```
int height (struct node *n) {
  int l,r;
  if (is_leaf (n))
    return 0;
  l = height (n->left);
  r = height (n->right);
  return 1 + (l > r ? l : r);
}
```

Let us consider a binary tree containing n nodes. If this tree has no additional property, then its height is in $O(n)$ as in the worst case, we can have a tree with height exactly n: every node only has one child.



On the contrary, on a complete binary tree, every level is full—except for the last one. This means that if h is the height of the tree, every path from the root to a leaf has length $h+1$ or h.



Let us compute h depending of n. Supposing the root is at level 0, there are exactly $h-1$ levels in the tree, all full except for level $h-1$ which may not be full. Let us call $level(l)$ the number of nodes in level l. Since every node in levels 0 to $h-2$ have two children, $level(l) = 2 \times level(l-1)$ for $l \in \{1, h-2\}$. We have $level(0) = 1$ (the root), so we conclude that $level(l) = 2^l$ for $l \in \{0, h-2\}$, and we have $1 \leqslant level(h-1) \leqslant 2^{h-1}$.

We also have:

$$\begin{aligned} n &= \sum_{l=0}^{h-1} level(l) \\ &= \sum_{l=0}^{h-2} 2^l + level(h-1) \\ &= 2^{h-1} - 1 + level(h-1) \end{aligned}$$

Hence,

$$\begin{aligned} 2^{h-1} - 1 + 1 \leqslant \quad n \quad &\leqslant 2^{h-1} - 1 + 2^{h-1} \\ 2^{h-1} \leqslant \quad n \quad &\leqslant 2^h - 1 \end{aligned}$$

So we have $h - 1 \leqslant \log n$ and $h \geqslant \log(n+1)$.
So $\log(n+1) \leqslant h \leqslant \log n + 1$.

So, for a complete binary tree, its height is exactly $\log n$. The important thing to remember is that the height a binary tree that is balanced in is $O(\log n)$. It is also possible to prove that on average, binary trees are usually quite balanced, i.e., a random binary tree of n nodes has height in $O(n)$.

2 Walking The Trees

In this part, we will see how to perform visits on trees.

2.1 Whole Tree Visits—Traversals

The act of visiting systematically a tree is called a *traversal*. There are numerous reasons to want to do so, if only just for the sake of printing the tree. We will describe the traversals only on binary trees, as the extension to general trees is straightforward. The two most common traversals differ in the order in which they visit the nodes (see Figure 1):

Depth-first : we go down first, visiting the whole subtree of thei child of a node before visiting the subtree of the other child;

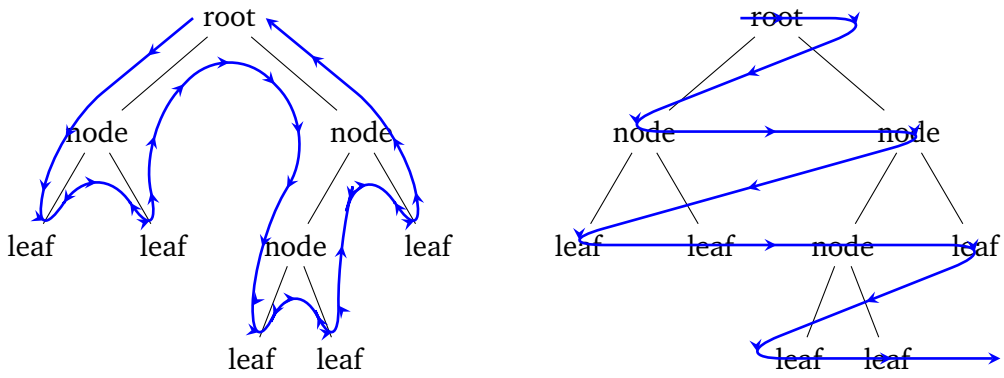
Breadth-first : we visit the nodes by level: first the root, then its children, then their children, etc.

2.1.1 Depth-first traversals

There are three versions of depth-first traversals, since we visit each node actually three times: when we arrive from father; when we return from the left son; and when we return from the right son. If there is an action to perform on the node (for instance, print its value), then we have three choices:

Pre-order: we perform the action when we enter the node for the first time, i.e., *before* visiting the children;

In-order: we perform the action when we enter the node for the second time, i.e., *between* visiting the left and right children;



(a) Depth-first (b) Breadth-first

Figure 1: Order of visiting depending on the traversal.

Post-order: we perform the action when we enter the node for the third and last time, i.e., *after* visiting both children.

It is also possible to do a mix of these order, performing actions before, between, and after traversing the children. In the algorithm below, we use this to do a pretty-printing of the tree that shows the structure using parentheses.

Recursive algorithm The easiest way to perform a depth-first traversal on a tree is to use recursion, as the structure is itself recursive.

```
1  function visit (n) =
2  begin
3  |  print "(";        /* Pre-order action */
4  |  if n has a left son then
5  |  |  visit (n.left);
6  |  print n.value;    /* In-order action */
7  |  if n has a right son then
8  |  |  visit (n.right);
9  |  print ")";        /* Post-order action */
10 in visit (root);
```

```
void visit_node_depth_first (struct node *n) {
  if (n == NULL) return;
  printf ("(");
  visit_node (n->left);
  printf ("%d", n->value);
  visit_node (n->right);
  printf (")");
}
```

Complexity: $O(n)$

Iterative algorithm However, for performance issue, it is sometimes required to do a pure iterative depth-first traversal of a binary tree, as there is no function call overhead in this case. We will however need another data structure to retain the information that was in the calling stack: we will be using a stack (LIFO) to "remember" what node to visit next.

In that case, is is easier to do a pre-order traversal, or we need to also keep in memory how many times we visited a node. We will just present the pre-order iterative traversal here.

Note: the implementation of the "stack" structure is not given here. It is easy to define you own structure using for instance arrays (if you know the maximum depth, or take care of expanding them if required) or linked lists.

```
1  stack ← ∅;
2  n ← root;
3  while n ≠< empty > or stack ≠ ∅ do
4  |  while n =< empty > and stack ≠ ∅
   |  do
5  |  |  n ← pop stack;
6  |  if n =< empty > then
7  |  |  return
8  |  print n.value;  /* Pre-order action */
9  |  push n.right on stack;
10 |  n ← n.left;
```

Complexity: $O(n)$

```
void visit_node_depth_first_iterative (struct
     node *n) {
  stack *stack;
  stack_init (stack);

  stack_push (stack, n);

  while (n != NULL || !stack_empty (stack)) {
    while (n == NULL && !stack_empty (stack)) {
      n = stack_pop (stack);
    }
    if (n == NULL) return;

    printf ("%d ", n->value);
    stack_push (stack, n->right);
    n = n->left;
  }
}
```

2.1.2 Breadth-first traversals

Breadth-first traversals cannot use the recursive property of trees as we do not stay in the same subtree during part of the traversal. As such, like the iterative version of depth-first traversals, we require another data structure to hold the information of which nodes we will need to visit in the future. However, in this case, the nodes that we insert last will also need to be visited last, so we will use a queue (FIFO) instead of a stack.

For breadth-first traversals, there is no notion of pre/in/post-order, as each node is only visited once.

```
1  queue ← {root};
2  while queue ≠ ∅ do
3  |  n ← pop queue;
4  |  print n.value;
5  |  if n has a left son then
6  |  |  append n.left to queue;
7  |  if n has a right son then
8  |  |  append n.right to queue;
```

Complexity: $O(n)$

```
void visit_node_breadth_first (struct node *n) {
  queue *queue;
  queue_init (queue);

  queue_append (queue, n);
  n = NULL;
  while (!queue_empty (queue)) {
    while (n == NULL && !queue_empty (queue))
          {
      n = queue_pop (queue);
    }
    if (n == NULL) return;

    printf ("%d ", n->value);
    queue_append (queue, n->left);
    queue_append (queue, n->right);
  }
}
```

2.1.3 Finding one's way

An interesting property of trees is that there is always a unique path linking any two nodes; Indeed, all nodes are connected and there is no cycle. We present here an algorithm to find this path.

The idea to find the path between nodes a and b is that we need to find the lowest common ancestor, called $x = \text{lca}(a, b)$, then the path is simply going up from a until we arrive at x,

then down to b. Of course, there is always only one option to go up but there are multiple possibilities to go down so we need to remember the way from b to x.

In order to find x, we will go up from a and b simultaneously, marking nodes along the way, thus creating two "paths" going up. Whenever we find a node already marked, it means the paths intersect, hence we have found x.

Up to now, we never had a use for "going up" the tree. In order to maintain an acceptable complexity of $O(height)$ and not $O(n)$, we augment the node structure by adding a new pointer, `father`, that points to the father of the node, or `NULL` for the root.

We will also need a way of marking which nodes we have already seen, so we also add a boolean `marked` to the node structure. We assume that when we enter the algorithms, all nodes are unmarked, and we must make sure that it is true when we algorithm finishes (to prepare for the next path search).

```
1  path ← ∅;
2  stack ← ∅;
3  v_a ← a; v_b ← b;
4  x ←< empty >;
5  while x =< empty > do
6  |  if v_a = v_b then
7  |  |  x ← v_a;
8  |  else if v_a is marked then
9  |  |  x ← v_a;
10 |  else if v_b is marked then
11 |  |  x ← v_b;
12 |  else
13 |  |  mark v_a; v_a ← v_a.father;
14 |  |  push v_b to stack;
15 |  |  mark v_b; v_b ← v_b.father;
   /* First, unmark all nodes from b that go
      above x (there may be none).          */
16 while v_b ≠ x do
17 |  unmark v_b;
18 |  v_b ← pop stack;
   /* Now, add all nodes from a up to x and
      unmark them.                          */
19 v_a ← a ;
20 while v_a ≠ x do
21 |  append v_a to path;
22 |  unmark v_a; v_a ← v_a.father;
23 append x to path;
   /* Continue cleaning marks above x.     */
24 while v_a is marked do
25 |  unmark v_a; v_a ← v_a.father;
   /* Then add what remains on the stack to
      the path.                             */
26 while stack ≠ ∅ do
27 |  v_b ← pop stack;
28 |  unmark v_b;
29 |  append v_b to path;
```

Complexity: $O(height)$

```
queue *find_path(struct node *a, struct node *b)
    {
  queue *path; stack *stack;
  queue_init (path);  stack_init  (stack);

  vA = a; vB = b;
  x = NULL;

  while (x == NULL) {
    if  (vA == vB) x = vA;
    else  if  (vA->marked) x = vA;
    else  if  (vB->marked) y = vB;
    else  {
      vA->marked = true; vA = vA->father;
      stack_push (stack,  vB);
      vB->marked = true; vB = vB->father;
    }
  }

  while (vB != x) {
    vB->marked = false;
    vB = stack_pop (stack);
  }

  vA = a;
  while (vA != x) {
    queue_append (path, vA);
    vA->marked = false;
    vA = vA->father;
  }

  queue_append (path, x);

  while (vA->marked) {
    vA->marked = false;
    vA = vA->father;
  }
  while (! stack_empty (stack)) {
    vB = stack_pop (stack);
    vB->marked = false;
    queue_append (path, vB);
  }
}
```

3 Exercises

Exercise 1 (Tree merge)

We want to merge two trees T_1 and T_2, and we suppose there is no particular property to maintain on those trees. The easiest way to do that is to make the root of say T_2 a child of a leaf of T_1.

Question 1.1 Write an algorithm that implements this idea. What is its complexity?

Question 1.2

Explain why this idea is not very good.

We want a solution so that the height of the merged tree is not greater than one plus the maximum height between T_1 and T_2.

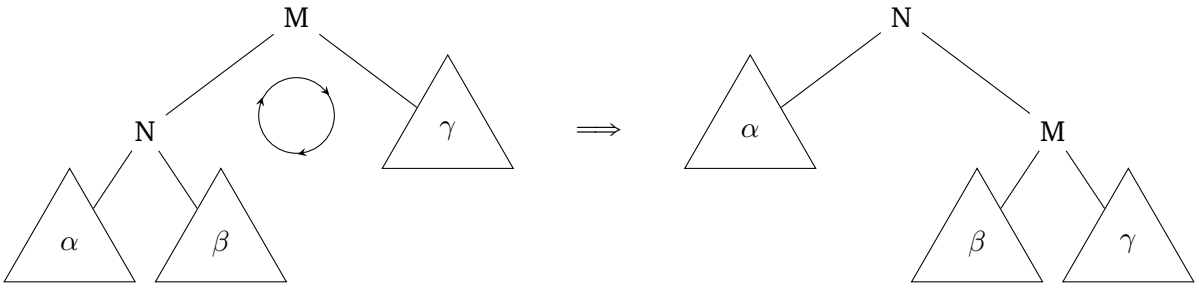
Question 1.3 Give a condition on the roots of T_1 and T_2 under which this is very easy to perform. What is its complexity?

Question 1.4 Suppose the roots of T_1 and T_2 do not satisfy the condition. Find a solution that keeps the structures of T_1 and T_2 nearly intact. What is its complexity?

Hint: have a look at the node deletion algorithm.

Exercise 2 (Rotations)

In this exercise, we will study transformations that changes the height of trees. The following drawing shows the effect of applying the transformation called `rotate_right` to the tree on the left, which produces the tree on the right (α, β and γ represent subtrees).

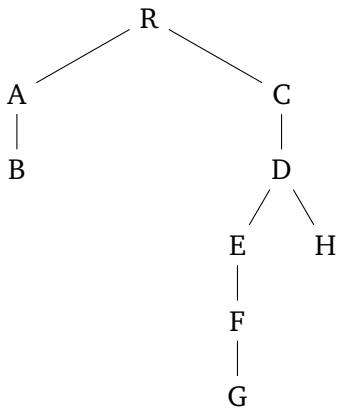


Question 2.1 Explain with a drawing the effect of the transformation `rotate_left`.

Question 2.2 Write the algorithm that implements `rotate_right`.

Question 2.3 Explain how rotation influences the height of a tree; Use the levels of leaves in α, β and γ for that.

Question 2.4 Use rotations (left and right) to balance the following tree as best as possible (i.e., make the height as small as possible).



Exercise 3 (Binary Search Trees (BST))

In a binary search tree, every node n contains a value v. Moreover, every node on the left subtree of n has a value $v' < v$, and every node on the right subtree of n has a value $v' > v$ (we suppose all values are different).

Question 3.1 Write the algorithm that searches for a particular value v in a BST. What is its complexity?

Question 3.2 Compare the complexity with that of searching for a value in a sorted array (using dichotomy). Is there any advantage in using a BST?

Question 3.3 Explain how the addition of a new node in a BST differs from the generic algorithm presented in Section 1.3.3.

Question 3.4 Explain how the deletion of an existing node in a BST differs from the generic algorithm presented in Section 1.3.4. Write the deletion algorithm in a BST.

Question 3.5 Consider the rotation transformations introduced in the previous exercise. Comment their uses in the case of BST, in particular with regard to the main property of BSTs (ordering of values).