Data Structures & Algorithms

Week 10

Trees

Lecturer: Dr. Msagha J Mbogholi, PhD

Flashback from Lesson 9

- The element (item) at the beginning of the list is at it's head while the last element is at the tail. Items in a list retain position relative to each other over time, and additions and deletions affect predecessor/successor relationships only at the point of modification.
- Operations on lists can be categorized as index based operations, content based operations, and positional list ADT.
- Indices are not a good abstraction for describing a local position in some applications, because the index of an entry changes over time due to insertions or deletions that happen earlier in the sequence.
- An iterator is an object that can be iterated upon, meaning that you can traverse through all the values. Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods __iter__() and __next__().
- Various implementations of the different types of linked lists (single linked, double linked, and circular linked) can be done using Python programming language.

Content

- Terms and definitions
- Tree algorithms
- Search trees

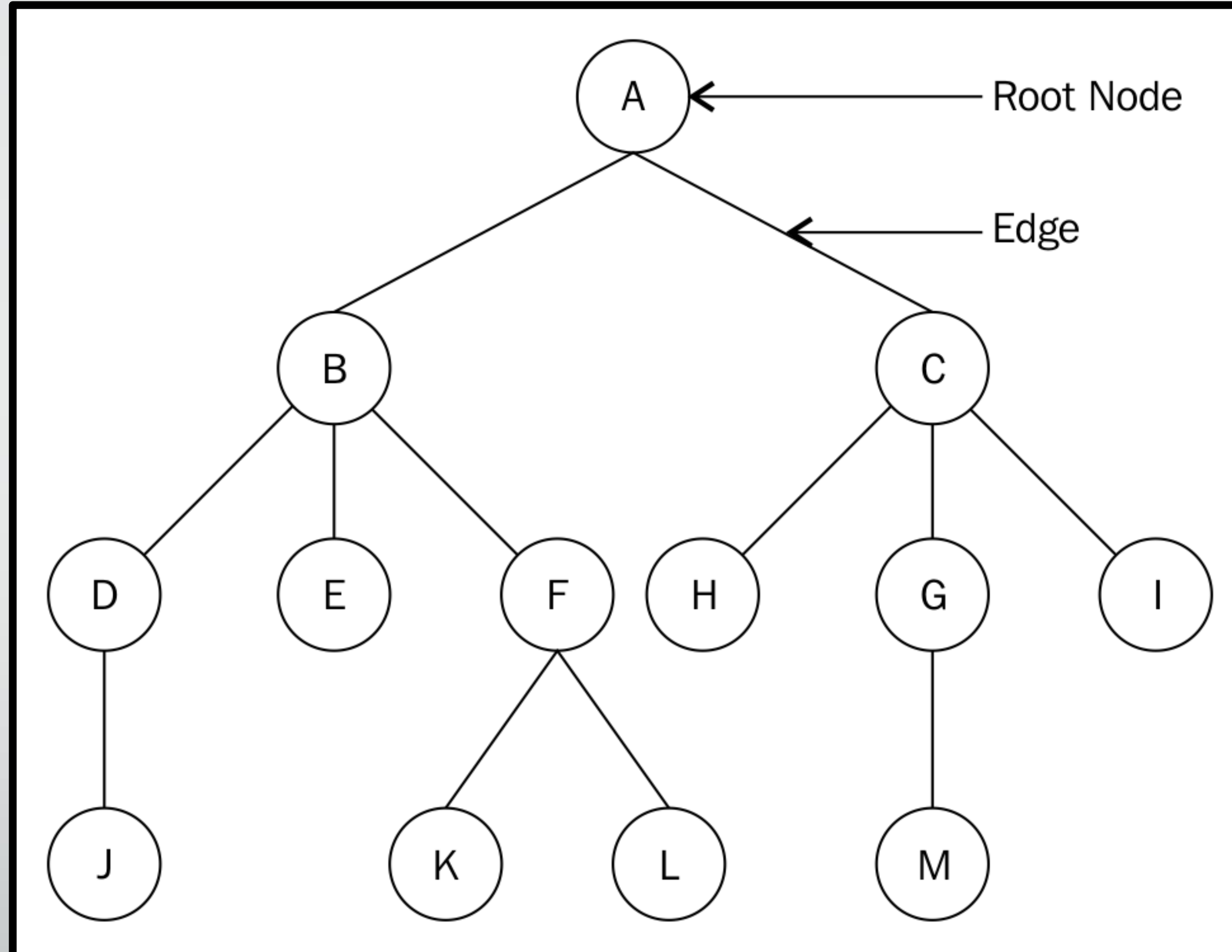
Overview

1.1 Introduction

- Can you imagine seeing a tree upside down?
- I have seen this in many fantasy movies, where they even show how the world looks like when turned upside down; talk of human imagination.
- Trees were introduced in lesson 2; we scratched the surface then by introducing the tree as a non-linear hierarchical structure.
- We can look at a tree as a structure in computing similar to the trees we see in nature; the difference is that we view the tree upside down (with the root at the top, and the branches below it going in to the ground) in order to understand their use in this domain.
- In this first part we introduce some terms and definitions that are necessary for us to understand how trees work, and how they are implemented.

Figure 1

Tree structure and terminology



(From Baka, 2017)

1.2 Terminology

- Figure 1 shows the structure of a tree. Arising therefrom we can define a few terminologies associated with trees (Baka, 2017):
- **Node**: Each circled alphabet represents a node. A node is any structure that holds data.
- **Root node**: The root node is the only node from which all other nodes come. A tree with an undistinguishable root node cannot be considered as a tree. The root node in our tree is the node A.
- **Sub-tree**: A sub-tree of a tree is a tree with its nodes being a descendant of some other tree. Nodes F, K, and L form a sub-tree of the original tree consisting of all the nodes.
- **Degree**: The number of sub-trees of a given node. A tree consisting of only one node has a degree of 0. This one tree node is also considered as a tree by all standards. The degree of node A is 2.

1.2 Terminology

- **Leaf node**: This is a node with a degree of 0. Nodes J, E, K, L, H, M, and I are all leaf nodes.
- **Edge**: The connection between two nodes. An edge can sometimes connect a node to itself, making the edge appear as a loop.
- **Parent**: A node in the tree with other connecting nodes is the parent of those nodes. Node B is the parent of nodes D, E, and F.
- **Child**: This is a node connected to its parent. Nodes B and C are children of node A, the parent and root node.
- **Sibling**: All nodes with the same parent are siblings. This makes the nodes B and C siblings.
- **Level**: The level of a node is the number of connections from the root node. The root node is at level 0. Nodes B and C are at level 1.

1.2 Terminology

- **Height of a tree**: This is the number of levels in a tree. Our tree has a height of 4.
- **Depth**: The depth of a node is the number of edges from the root of the tree to that node. The depth of node H is 2.

1.3 Nodes

- Just like the linked lists discussed in earlier chapters, trees are built up of nodes. But the nodes that make up a tree need to contain data about the parent-child relationship that we mentioned earlier.
- When you think about it this makes sense; every node needs to know who are its parents (if any) and who are its children (if any). In the linked lists the nodes kept track of preceding and after nodes using the pointers (which are similar to the edges in our trees).
- Just as we observed in figure 1 every node has at most 2 edges; this implies that for every node there is a left branch and right branch at most. We describe this as a binary relationship, and the tree is a binary tree.
- Let us construct the Python code to build a binary node tree class:

1.3 Nodes

- class Node:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.right_child = None
        self.left_child = None
```

- Just like the other data structures described earlier, a node is a container for data and holds references to other nodes; the difference being that since this is a non-linear structure (a hierarchical one), these references are to the left and the right children.

1.3 Nodes

- Let us use the class definition to instantiate a few nodes:
- n1 = Node("root node")
- n2 = Node("left child node")
- n3 = Node("right child node")
- n4 = Node("left grandchild node")
- The nodes have been created and labelled appropriately; for example n1 is labelled as a root node, n2 as a left child node, n3 as a right child node and n4 as a left grandchild node. However, the nodes are created but the relationship to each other is yet to be defined.

1.3 Nodes

- In the next step we wish to relate the nodes to each other; we use code to show where the root is and how other nodes branch out from it. As we created the nodes in our previous slide we now only need to relate them. Note that we could have created the nodes without labels, for example, n1 = node(), meaning it is a node without data, and from the constructor definition this is still correct.
- We define the relationships as follows:
- n1.left_child = n2 # the left child of n1 is n2
- n1.right_child = n3 # the right child of n1 is n2
- n2.left_child = n4 # the left child of n2 is n4, meaning n4 is a grandchild of n1
- By making this definition we have created the edges that connect all the nodes.

1.4 Node traversal

- The only thing left is to describe how to traverse the tree to reach the different nodes. Suppose we wish to traverse the left side of the tree, we shall have to start from the root (n1) and traverse downwards maintaining the left till we reach the end. In order to do this we use the while statement for printing out the nodes till there are no nodes left:

```
current = n1
# we can choose any other node to be current and start traversing from there
while current:
    print(current.data)
    current = current.left_child
```

Part 2

Tree algorithms

2.1 Binary trees

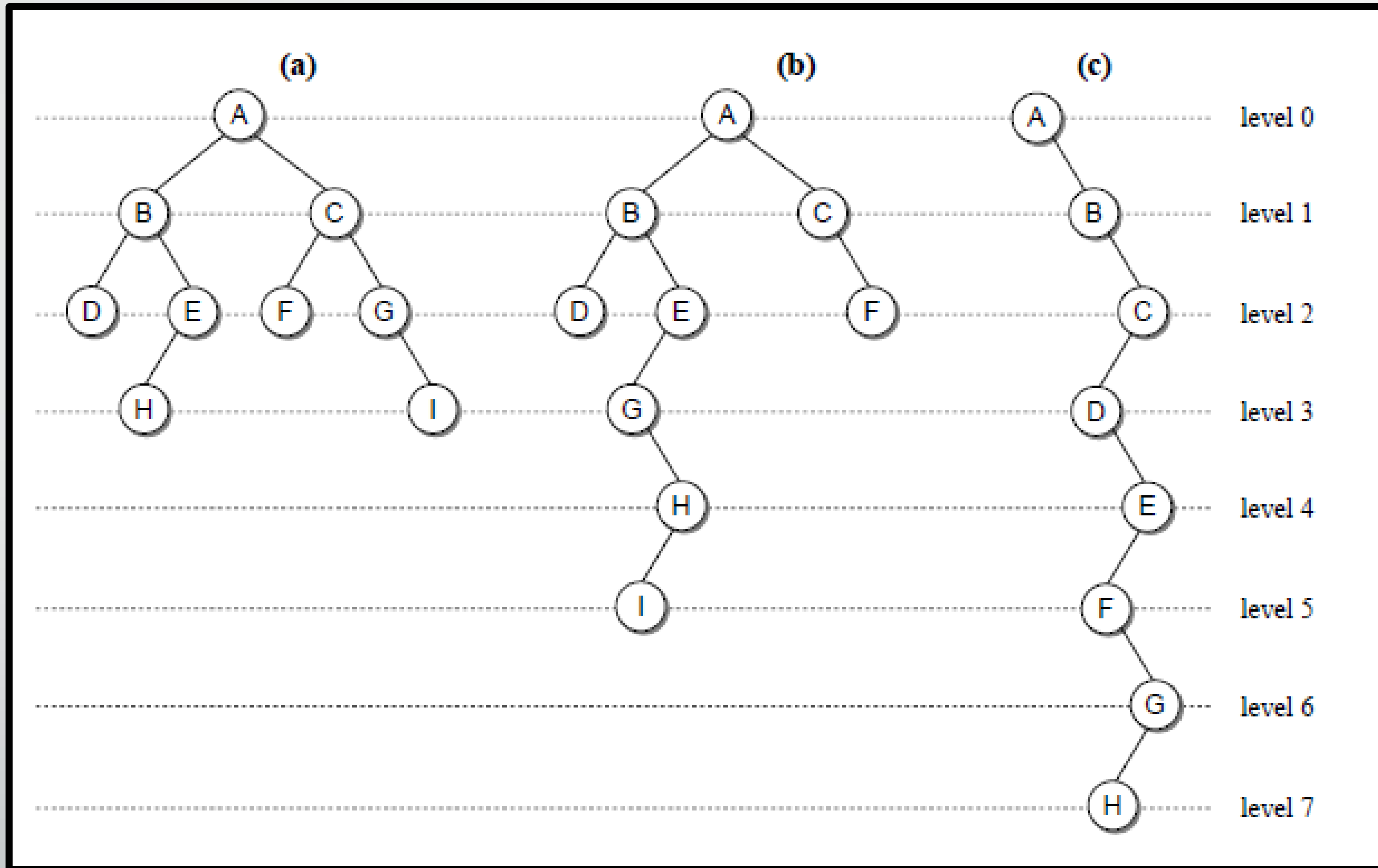
- Trees can come in many different shapes, and they can vary in the number of children allowed per node (depending on the rules) or in the way they organize data values within the nodes.
- In part 1 of this lesson we examined trees, which have at most two edges (left and right). We call such a tree a binary tree. A binary tree is defined as follows (Goodrich et al., 2018):
- A ***binary tree*** is an ordered tree with the following properties:
- 1. Every node has at most two children.
- 2. Each child node is labeled as being either a ***left child*** or a ***right child***.
- 3. A left child precedes a right child in the order of children of a node.

2.1.1 Properties

- The shapes of binary trees vary depending on the number of nodes and how the nodes are linked (as long as the rules defining a binary tree are maintained). Figure 2 illustrates three different shapes of a binary tree consisting of nine nodes. There are a number of properties and characteristics associated with binary trees, all of which depend on the organization of the nodes within the tree.
- We use this figure to describe the properties of binary trees. The properties we are interested in are:
  - Tree size
  - Tree structure

Figure 2

Binary tree shapes



(From Necaise, 2011)

2.1.1 Properties

- Tree size:
- As described in part 1 the nodes in a binary tree are organized into levels with the root node at level 0, its children at level 1, the children of level one nodes are at level 2, and so on.
- Moreover, each level corresponds to a generation: thus B, and C in figure 2(a) are of the same generation, as are D, E, and F. The root node (A) is always at level 0.
- We defined the depth of a node is its distance from the root, with distance being the number of levels that separate the two. A node's depth corresponds to the level it occupies; further the depth is also the number of edges to the node from the root as defined in part 1. For example the node G in figure 2 is at different depths in (a), (b) and (c) respectively. In tree (a) G has a depth of 2, in tree (b) it has a depth of 3, and in (c) its depth is 6. You see the connection between depth and level now?

2.1.1 Properties

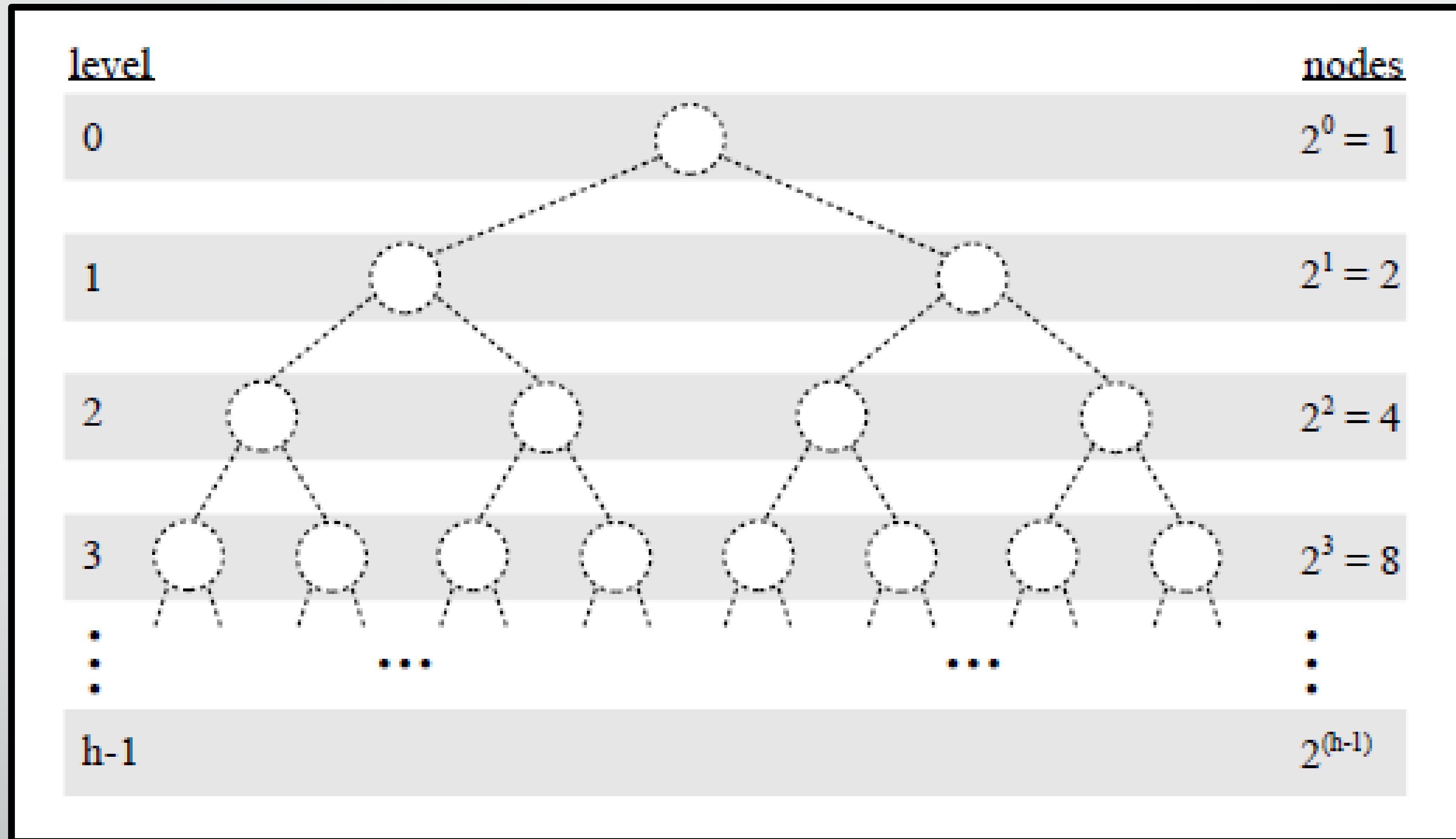
- We defined the height (of a binary tree) is the number of levels in the tree. Figure 2(a) has a height of 4; how about 2(b) and 2(c) ?
- Lastly the size of the tree is the total number of nodes that make up the tree. For example the size of the tree in figure 2(a) is 9; how about the other trees? We notice that the tree in figure 2(c) has only node per branch? This is a classical example of the maximum number height that a tree can have based on its size. That is to say that for a tree with n nodes the maximum height will also be n.
- To be clear, we talk of the height of the tree, while we talk of the depth of a node.
- We have seen the maximum height of the tree; how about the minimum height of a tree with n nodes?

2.1.1 Properties

- To determine this, we need to consider the maximum number of nodes at each level since the nodes will have to be organized with each level at full capacity.
- We know that in a binary tree the maximum number of child nodes each parent node will have is 2. We can visualize this and show it in figure 3. We further notice that since this is a binary increase we expect that at most each successive level in the tree doubles the number of nodes contained on the previous level. This corresponds to a given tree level p having a capacity for 2^p nodes.
- If we sum the size of each level, when all of the levels are filled to capacity, except possibly the last one, we find that the minimum height of a binary tree of size n is $[\log_2 n] + 1$.

Figure 3

Possible slots for placement of nodes in binary tree



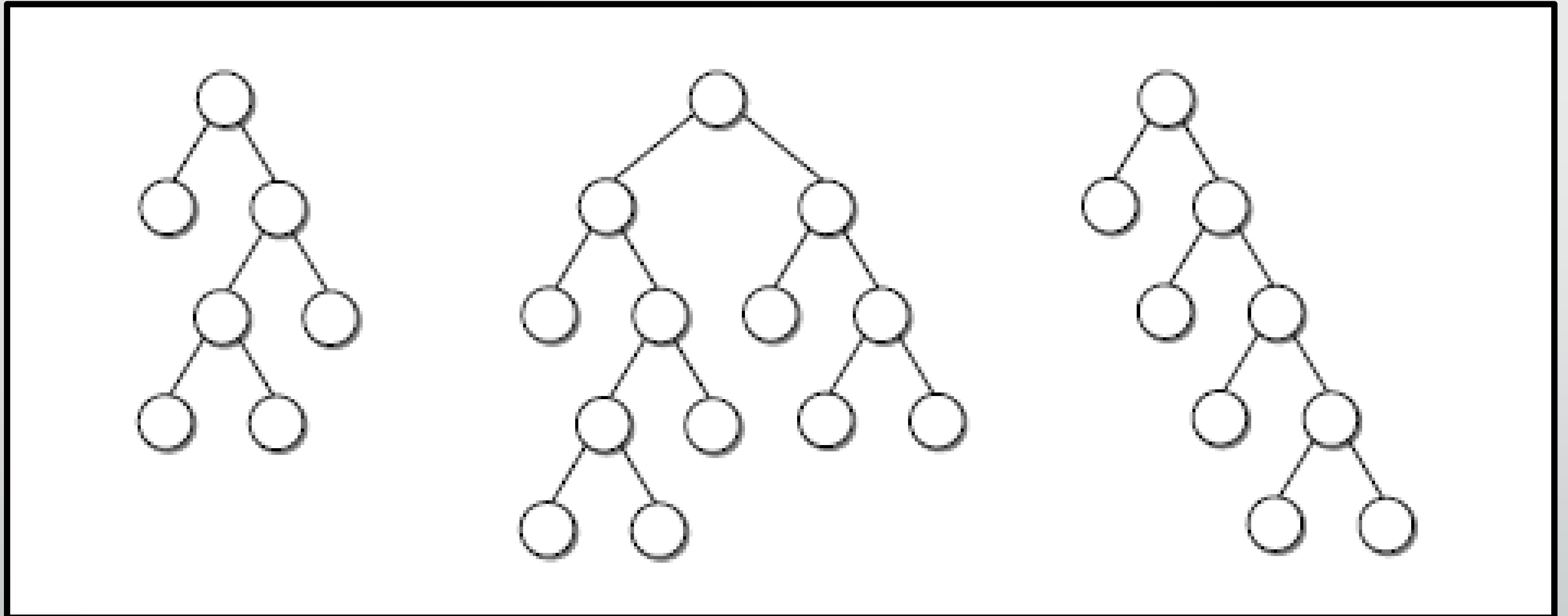
(From Necaise, 2011)

2.1.1 Properties

- Structure:
- The height of the tree will be important in analyzing the time-complexities of various algorithms applied to binary trees. The structural properties of binary trees can also play a role in the efficiency of an algorithm. In fact, some algorithms require specific tree structures. (Necaise, 2011)
- A binary tree can be defined as being a full binary tree or a perfect binary tree. Necaise (2011) provides the definition of the two: "
- A full binary tree is a binary tree in which each interior node contains two children. Full trees come in many different shapes, as illustrated in Figure (4).

Figure 4

Full binary tree



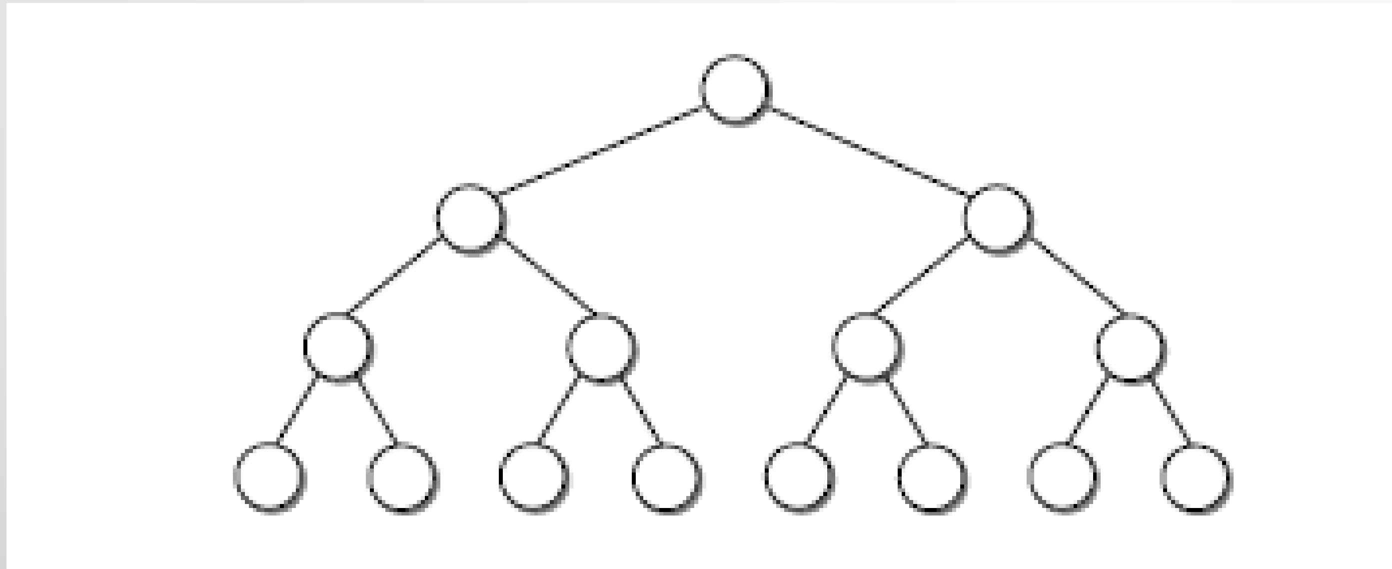
(From Necaise, 2011)

2.1.1 Properties

- A perfect binary tree is a full binary tree in which all leaf nodes are at the same level. The perfect tree has all possible node slots filled from top to bottom with no gaps, as illustrated in Figure (5)."
- Necaise (2011) further adds that "A binary tree of height h is a complete binary tree if it is a perfect binary tree down to height (h – 1) and the nodes on the lowest level fill the available slots from left to right leaving no gaps. Consider the two complete binary trees in Figure (6). If any of the three leaf nodes labeled A, B, or C in the left tree were missing, that tree would not be complete. Likewise, if either leaf node labeled X or Y in the right tree were missing, it would not be complete.
- Also we define a balanced tree as one where "the heights of any node's left and right subtrees differ by at most one." (Ghosh, 2023)

Figure 5

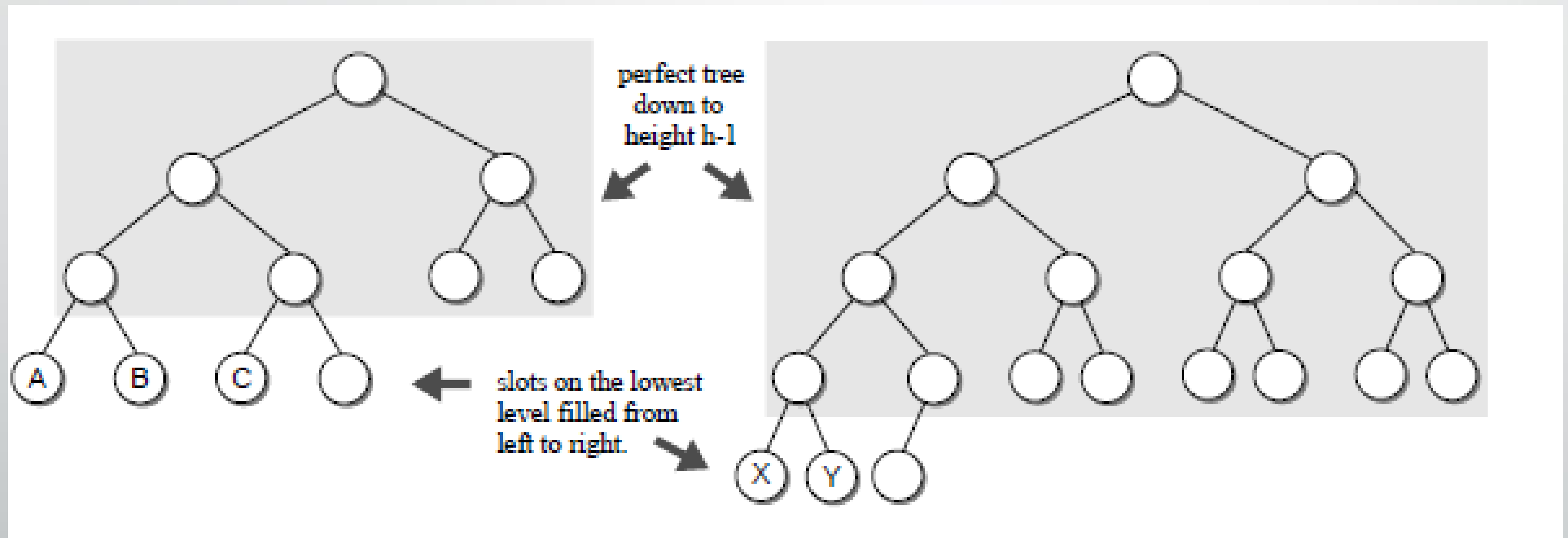
Perfect binary tree



(From Necaise, 2011)

Figure 6

Complete binary trees



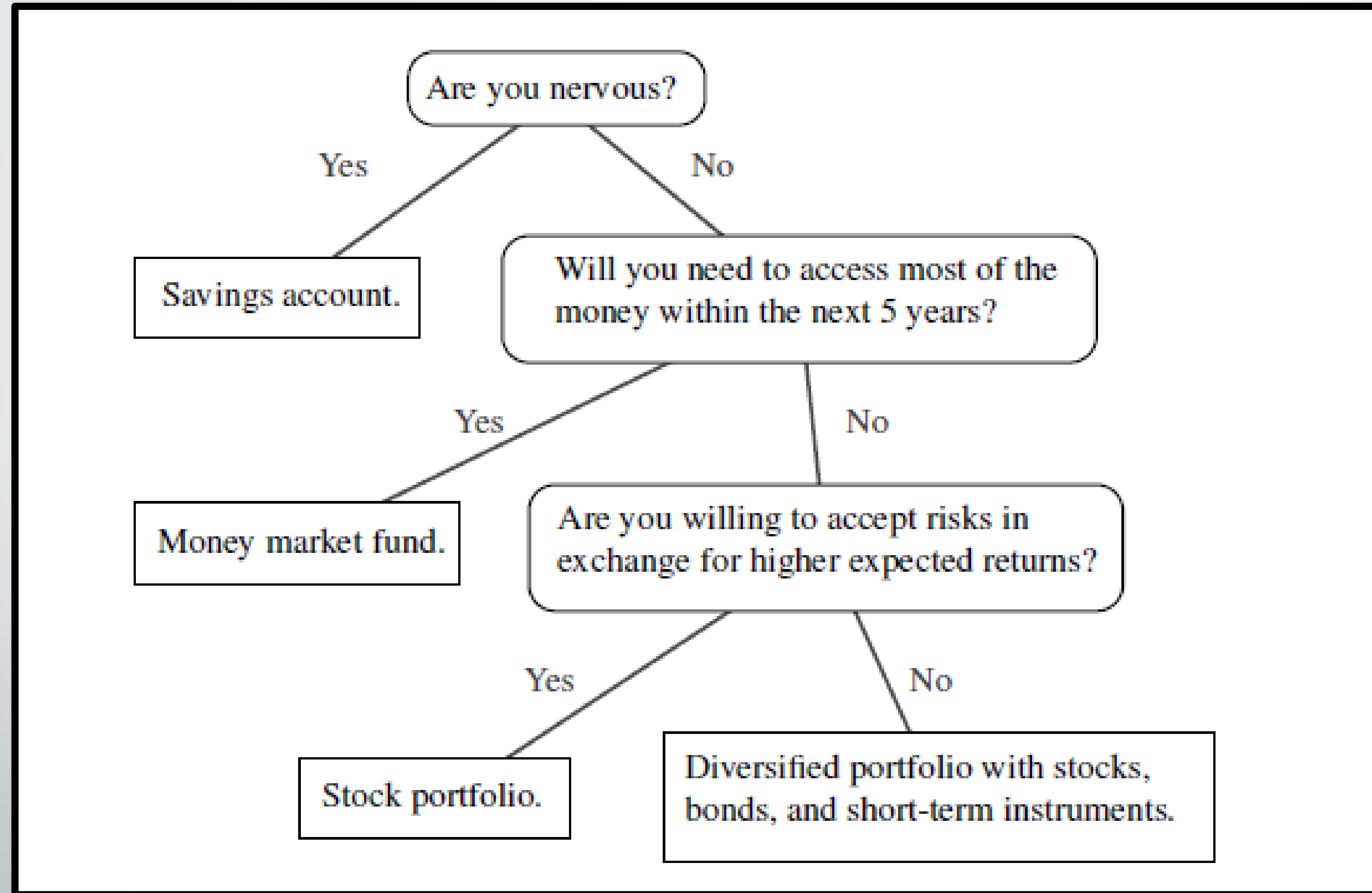
(From Necaise, 2011)

2.1.2 Binary tree as decision tree

- Goodrich et al. (2018) provide a nice example of a binary tree for giving investment advice; this is shown in figure 7.
- They explain it as follows: “An important class of binary trees arises in contexts where we wish to represent a number of different outcomes that can result from answering a series of yes-or-no questions. Each internal node is associated with a question. Starting at the root, we go to the left or right child of the current node, depending on whether the answer to the question is “Yes” or “No.” With each decision, we follow an edge from a parent to a child, eventually tracing a path in the tree from the root to a leaf. Such binary trees are known as **decision trees**, because a leaf position p in such a tree represents a decision of what to do if the questions associated with p’s ancestors are answered in a way that leads to p. A decision tree is a proper binary tree.”

Figure 7

Decision tree for an investor



(From Goodrich et al., 2018)

2.2 Algorithms

- Trees use different algorithms for traversal purposes. In this section we examine three common algorithms used for traversal:
- Preorder and postorder traversal
- Breadth-first traversal
- Inorder traversal
- In the last section of this part we introduce and discuss heaps.

2.2.1 Preorder and postorder traversal

- Goodrich et al., 2018) state the following regarding preorder traversal: “In a ***preorder traversal*** of a tree *T*, the root of *T* is visited first and then the subtrees rooted at its children are traversed recursively. If the tree is ordered, then the subtrees are traversed according to the order of the children. The algorithm is written as follows:

  Algorithm preorder(T, p):

  perform the “visit” action for position p

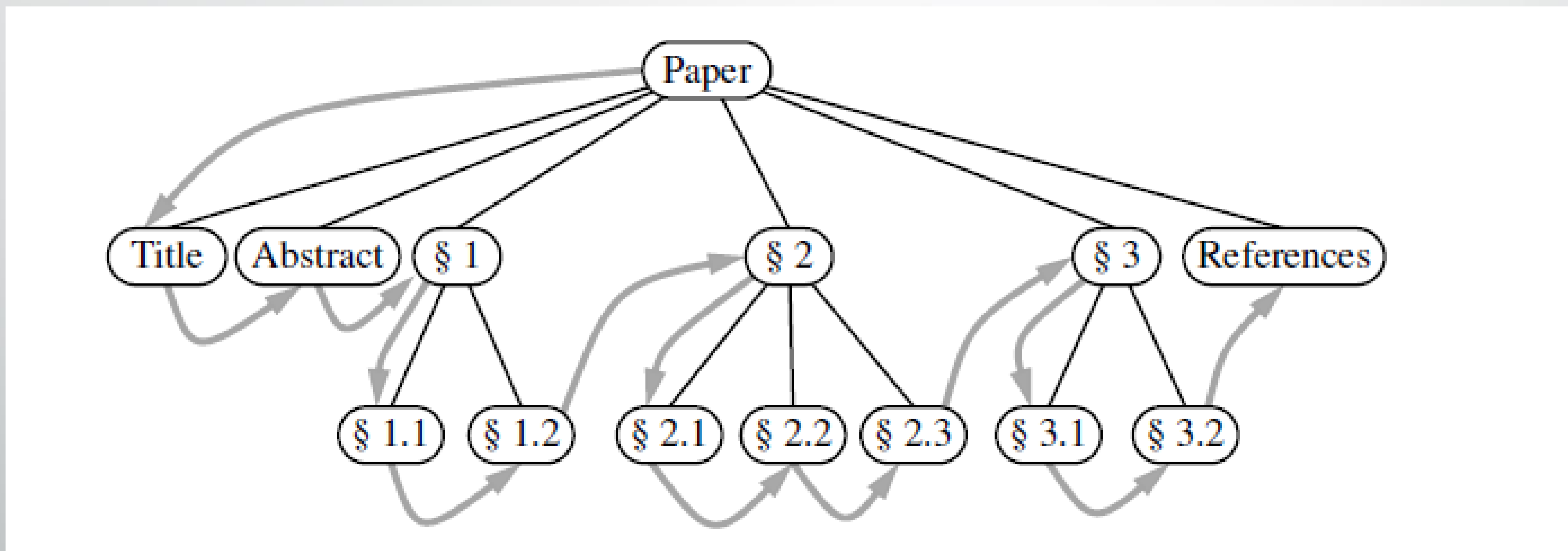
  for each child c in T.children(p) **do**

  preorder(T, c) {recursively traverse the subtree rooted at c} “

- Figure 8 demonstrates this traversal. In preorder traversal the children of each position are ordered from left to right.

Figure 8

Preorder traversal



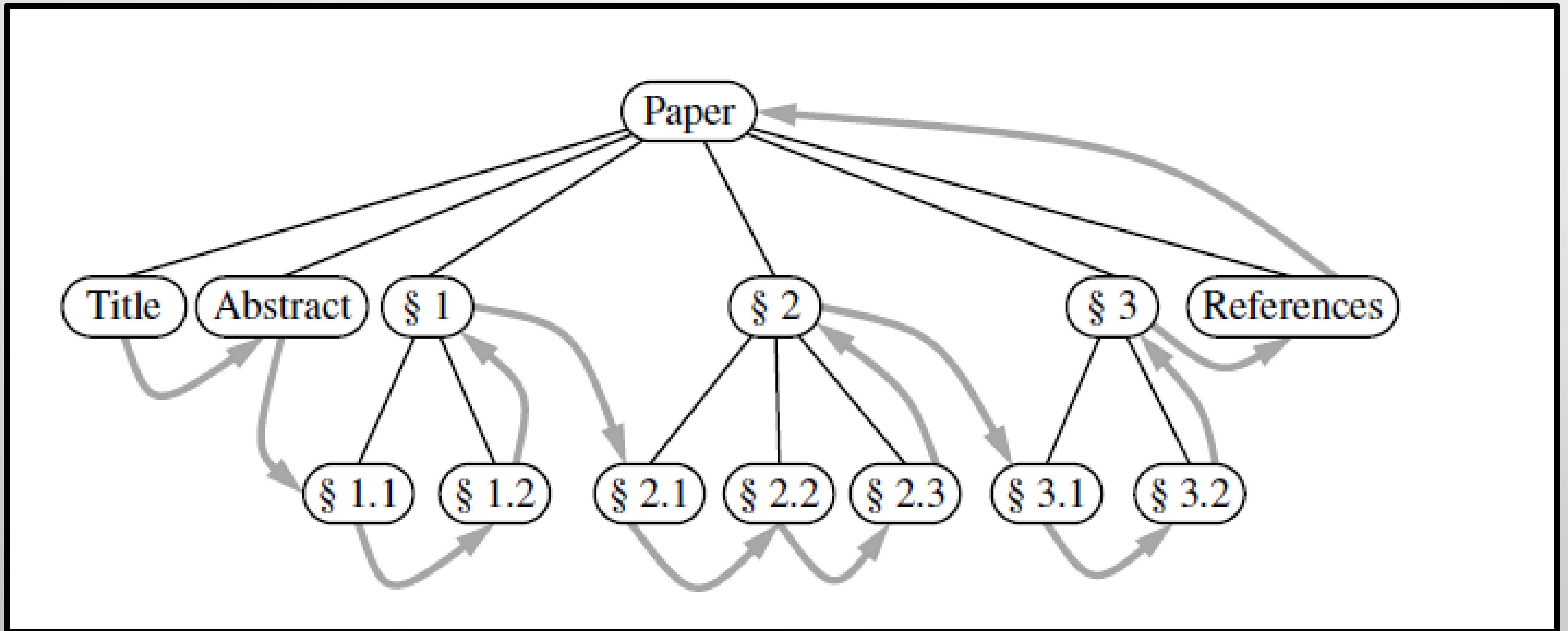
(From Goodrich et al., 2018)

2.2.1 Preorder and postorder traversal

- Goodrich et al., 2018) state the following regarding postorder traversal: "
- In some sense, this algorithm can be viewed as the opposite of the preorder traversal, because it recursively traverses the subtrees rooted at the children of the root first, and then visits the root (hence, the name "postorder"). The algorithm is as follows:
- **Algorithm** postorder(T, p):
  - **for** each child c in T.children(p) **do**
    - postorder(T, c) {recursively traverse the subtree rooted at c}
  - perform the "visit" action for position p "
- Figure 9 demonstrates the working of this algorithm

Figure 9

Postorder traversal



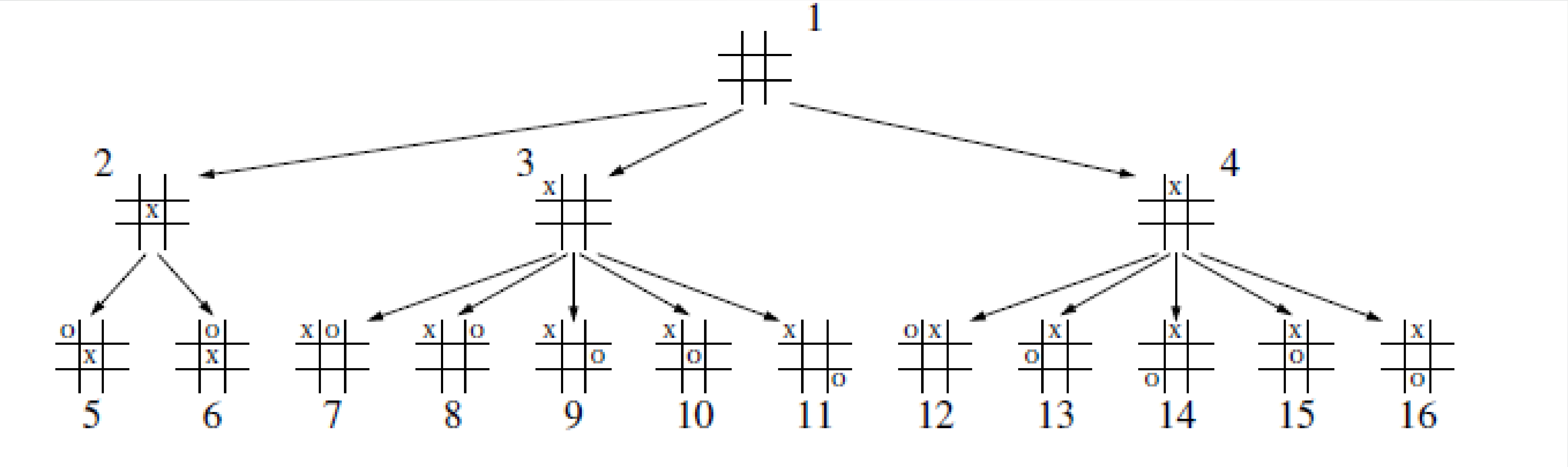
(From Goodrich et al., 2018)

2.2.2 Breadth-first traversal

- As the name implies the breadth-first algorithm works by traversing across the breadth (width) of the tree level by level, always starting from the left as you move to the right.
- It is widely used in computer games, and Goodrich et al. (2018) demonstrate this using an example of a common game tic-tac-toe. I am guessing you have played it but if not the target is to fill three diagonal or adjacent rows/columns with an X or an O. One player starts (choosing either X or O) and then the other follows; each gets a turn at each round..
- Figure 10 demonstrates the use of the breadth-first traversal algorithm using the tic-tac-toe game. Thereafter we examine the algorithm itself.

Figure 10

Breadth-first traversal



(From Goodrich et al., 2018)

2.2.2 Breadth-first traversal

- The algorithm for breadth-first traversal is as follows (Goodrich et al., 2018): "We use a queue to produce a FIFO (i.e., first-in first-out) semantics for the order in which we visit nodes. The overall running time is $O(n)$, due to the n calls to enqueue and n calls to dequeue:

Algorithm breadthfirst(T):

    Initialize queue Q to contain T.root( )

    while Q not empty **do**

        p = Q.dequeue( ) {p is the oldest entry in the queue}

        perform the "visit" action for position p

        for each child c in T.children(p) **do**

            Q.enqueue(c) {add p's children to the end of the queue for later visits}

2.2.3 Inorder traversal

- This algorithm was specifically designed for binary trees; however, binary trees can also use the previous traversal algorithms.
- The idea is to traverse in order from left to right of the tree. That is to say, “for every position *p*, the inorder traversal visits *p* after all the positions in the left subtree of *p* and before all the positions in the right subtree of *p*.” (Goodrich et al., 2018). This is demonstrated in figure 11 for evaluating an arithmetical expression $3+1\times3/9-5+2$. when using this form of traversal for expressions we refer to it as an **expression tree**. The algorithm as shared by Goodrich et al. (2018) is as follows:

Algorithm inorder(p):

  if p has a left child lc **then**

    inorder(lc) {recursively traverse the left subtree of p}

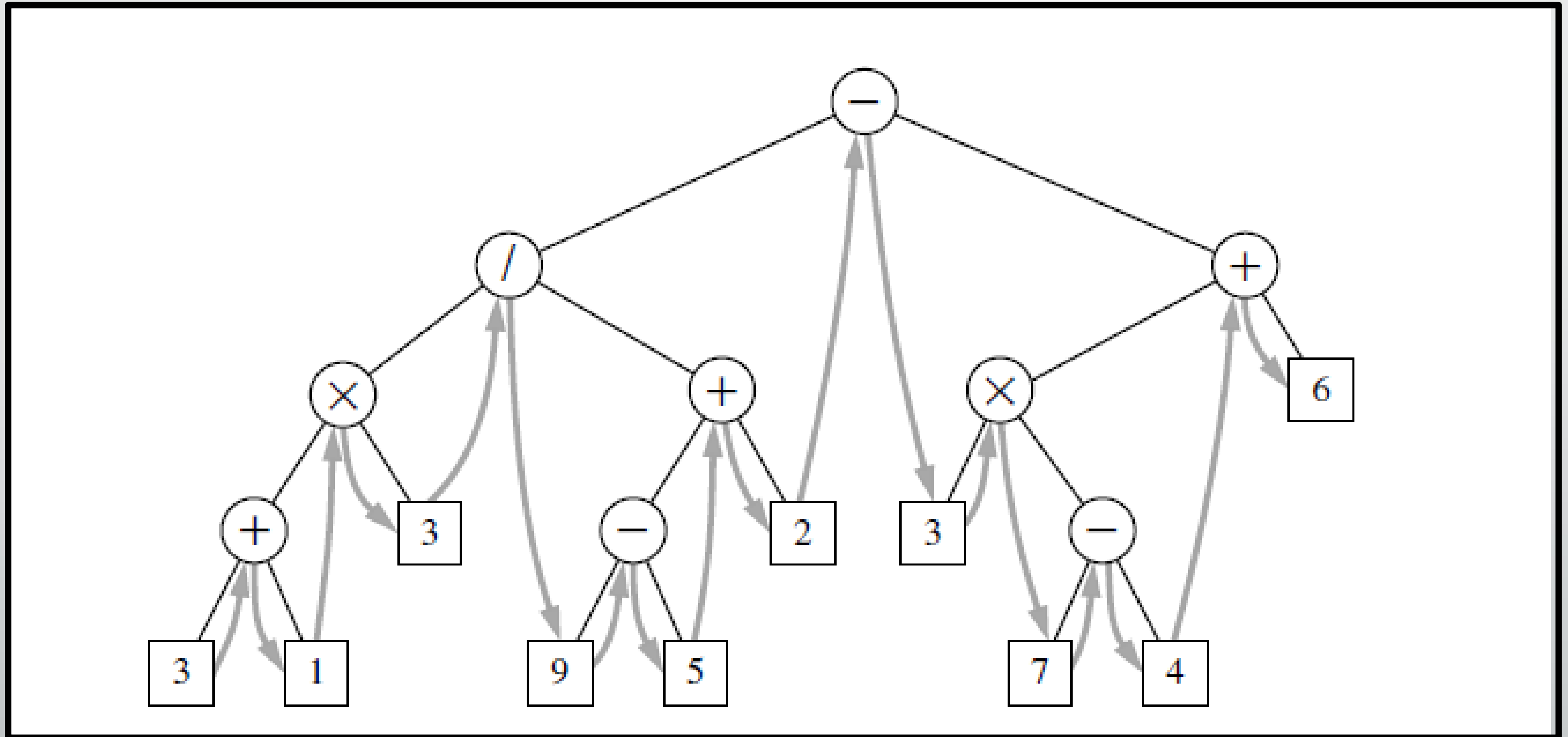
    perform the “visit” action for position p

  if p has a right child rc **then**

    inorder(rc) {recursively traverse the right subtree of p}

Figure 11

Inorder traversal of mathematical expression



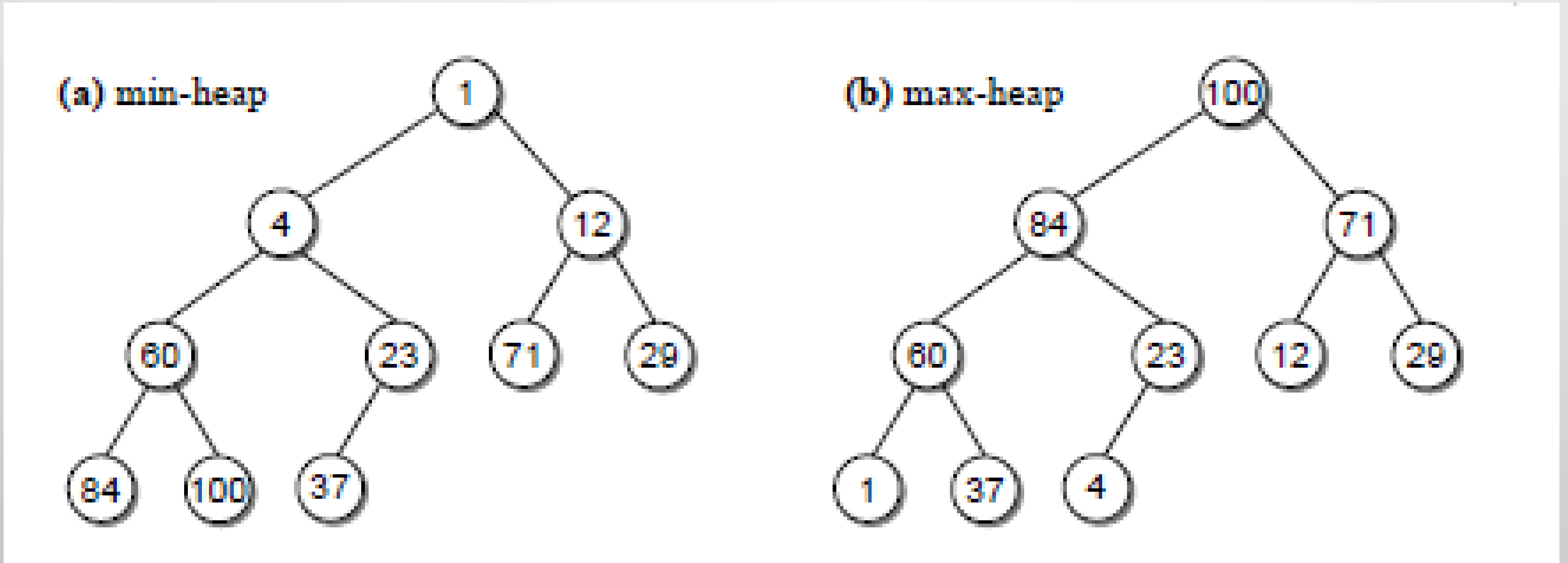
(From Goodrich et al., 2018)

2.3 Heap

- Definition: The heap is a specialized structure with limited operations. We can insert a new value into a heap or extract and remove the root node's value from the heap. (Necaise, 2011)
- Further, “there are two variants of the heap structure. A max-heap has the property, known as the heap order property, that for each non-leaf node V , the value in V is greater than the value of its two children. The largest value in a max-heap will always be stored in the root while the smallest values will be stored in the leaf nodes. The min-heap has the opposite property. For each non-leaf node V , the value in V is smaller than the value of its two children.” (Necaise, 2011)
- Figure 12 shows an example of both max heap and min heap. We then explain how to construct both heaps using an example.

Figure 12

Max and Min heap



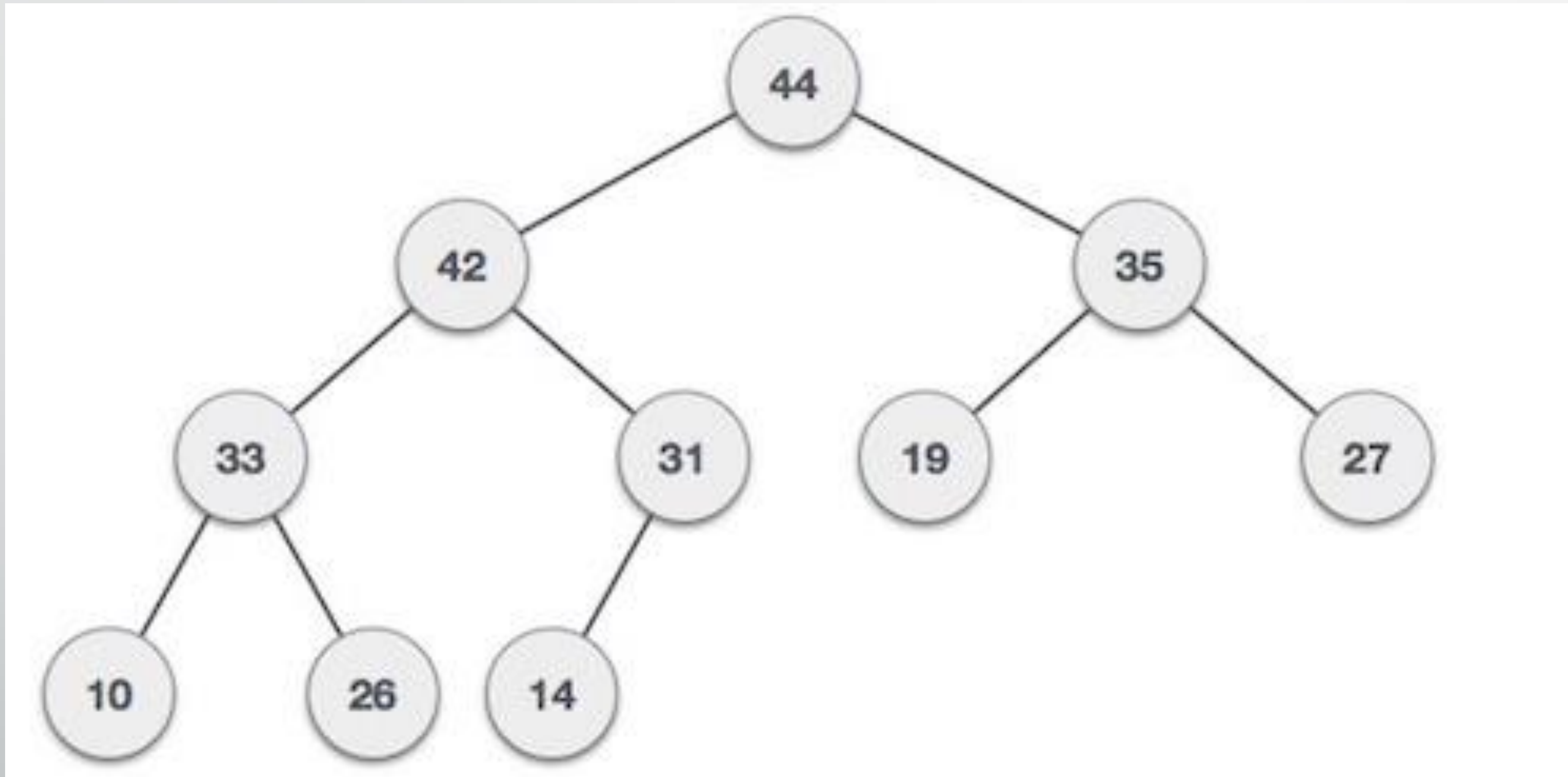
(From Goodrich et al., 2018)

2.3.1 Max Heap

- Let us use an example from (*Heap Data Structures - Tutorialspoint*, n.d.). Once completed then we shall examine how figure 12 was arrived at.
- Let us consider input as 35 33 42 10 14 19 27 44 26 31.
- In max heap the value of the root node is greater than or equal to either of its children.
- The algorithm for max heap is as follows:
  - Step 1 – Create a new node at the end of heap.
  - Step 2 – Assign new value to the node.
  - Step 3 – Compare the value of this child node with its parent.
  - Step 4 – If value of parent is less than child, then swap them.
  - Step 5 – Repeat step 3 & 4 until Heap property holds.
- Using the algorithm yields the tree as shown in figure 13. Figure 14 shows the animation of the same input to produce the final max heap.

Figure 13

Max heap of input



(From Heap Data Structures - Tutorialspoint, n.d.)

Figure 14

Max heap of input animation

Input 35 33 42 10 14 19 27 44 26 31

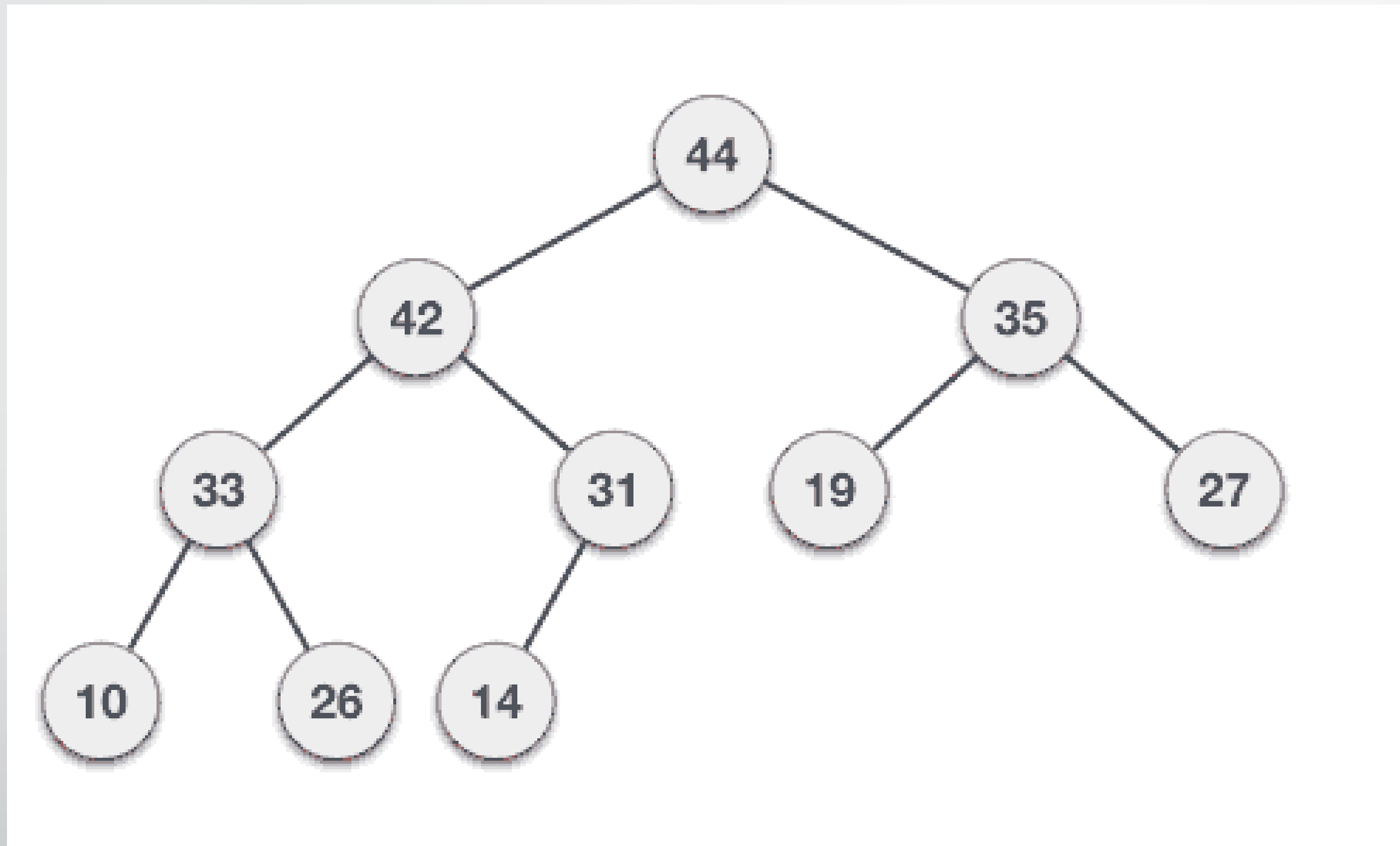
(From Heap Data Structures - Tutorialspoint, n.d.)

2.3.1 Max Heap

- We have seen how to add to Max heap. How about if we wish to delete?
- The algorithm is as follows (*Heap Data Structures - Tutorialspoint*, n.d.):" Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value:
- **Step 1** – Remove root node.
- **Step 2** – Move the last element of last level to root.
- **Step 3** – Compare the value of this child node with its parent.
- **Step 4** – If value of parent is less than child, then swap them.
- **Step 5** – Repeat step 3 & 4 until Heap property holds.
- Figure 15 shows an animation of the deletion process.

Figure 14

Max heap deletion of input animation

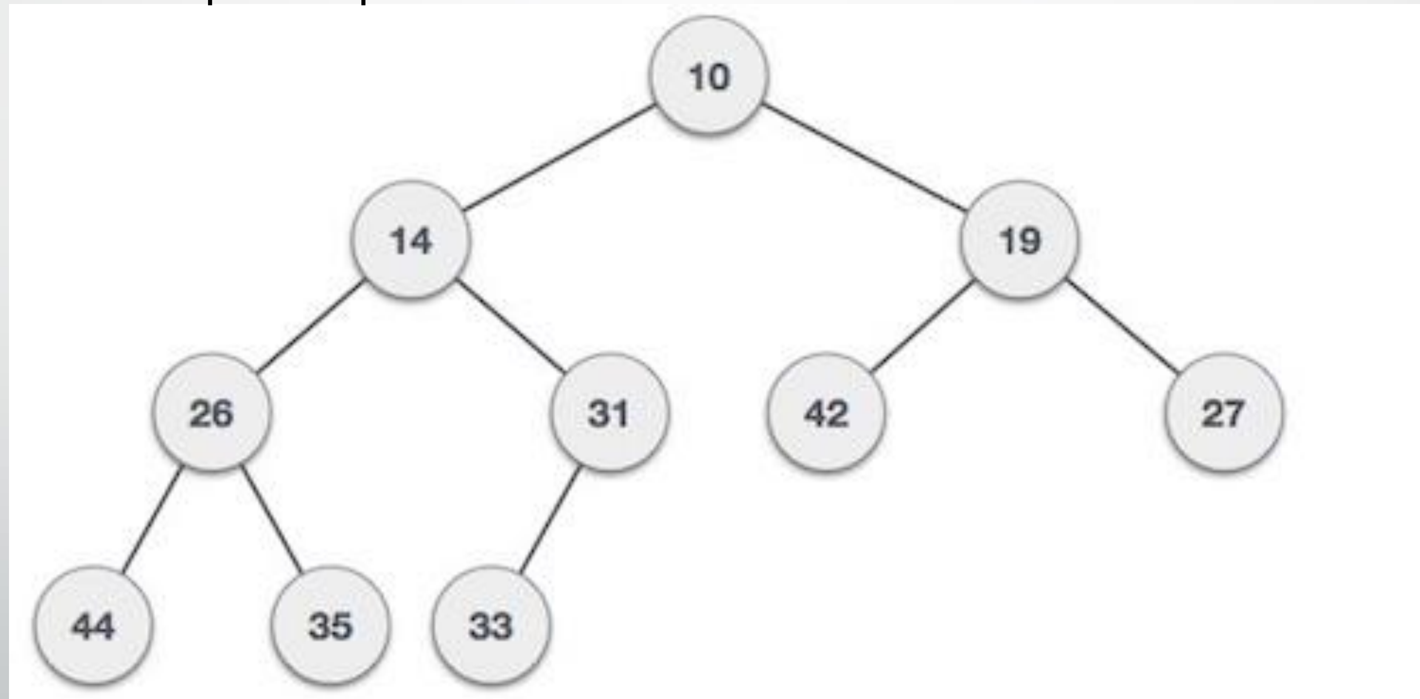


(From Heap Data Structures - Tutorialspoint, n.d.)

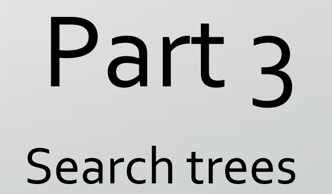
2.3.2 Min Heap

- As described earlier, Min heap is where the value of the root node is less than or equal to either of its children. Figure 15 shows the min heap for the input we used for the max heap.

Figure 15
Min heap for input



(From Heap Data Structures - Tutorialspoint, n.d.)



Part 3

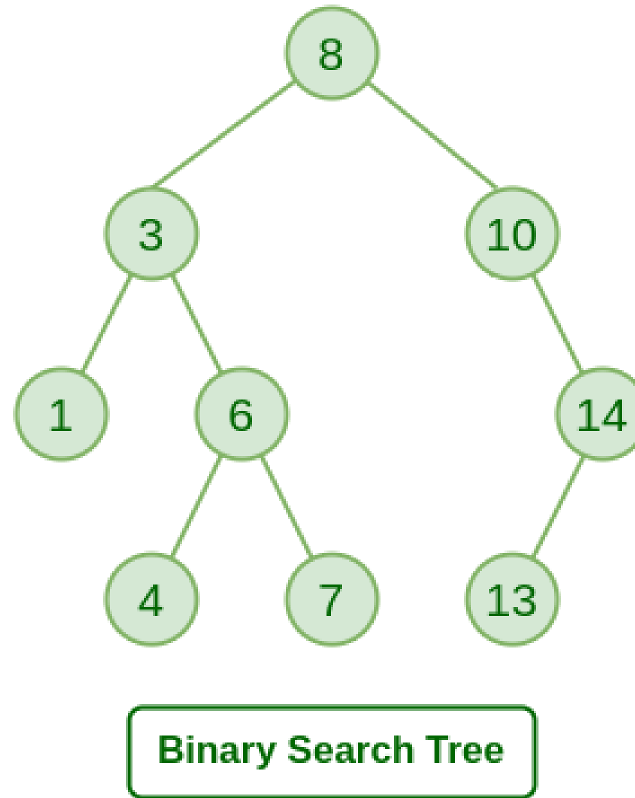
Search trees

3.1 Binary search tree (BST)

- **A Binary Search Tree** is a node-based binary tree data structure which has the following properties (*Binary Search Tree - GeeksforGeeks*, 2015) :
- 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.
- Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as (*Data Structure - Binary Search Tree - Tutorialspoint*, n.d.) –
- left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)

Figure 16

Binary search tree (BST)



(From *Binary Search Tree - GeeksforGeeks*, 2015)

3.1 Binary search tree (BST)

- Figure 16 shows an example of a BST. We note that all values on the left side are lower than the value of the root node; conversely, all values on the right side are higher than the value of the root node. This also holds true for every sub-tree in the tree.
- The five operations performed by the BST are:
- Insertion
- Searching
- Deletion
- Traversal
- Conversion from normal BST to balanced BST.

3.1.1 Searching

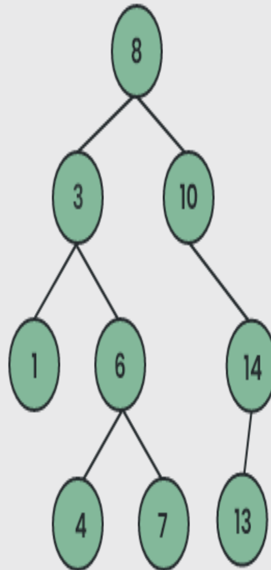
- Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node (*Data Structure - Binary Search Tree - Tutorialspoint*, n.d.) –
- 1. Check whether the tree is empty or not
- 2. If the tree is empty, search is not possible
- 3. Otherwise, first search the root of the tree.
- 4. If the key does not match with the value in the root, search its subtrees.
- 5. If the value of the key is less than the root value, search the left subtree
- 6. If the value of the key is greater than the root value, search the right subtree.
- 7. If the key is not found in the tree, return unsuccessful search.

Let us consider a simple example in figure 17

Figure 17

BSTtree search operation example

(a)

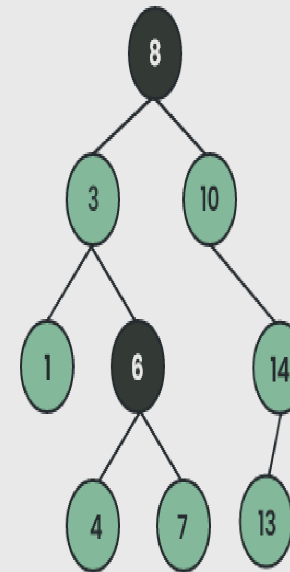


Consider The Following BST

Key = 6

Searching In BST

(b)



Compare Key With Root, i.e 8 as 6<8, search in left subtree of 8

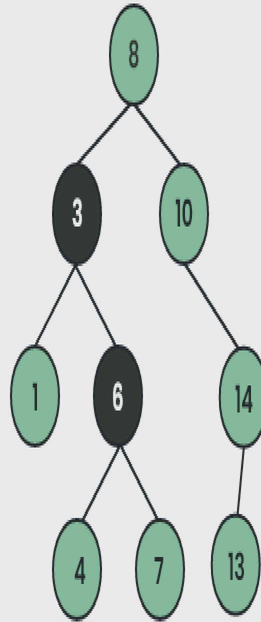
Searching In BST

(Adapted from *Binary Search Tree | Set 1 (Search and Insertion)*, 2014)

Figure 17

BSTtree search operation example

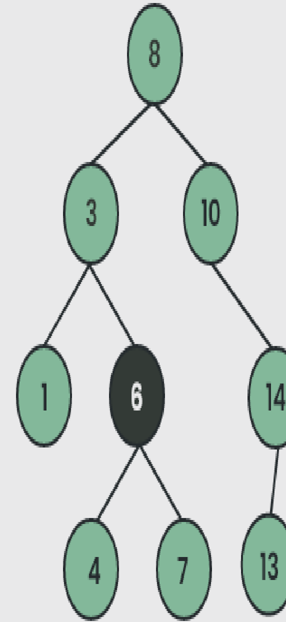
(c)



As Key (6) Is Greater Than 3,
Search In The Right Subtree Of 3

Searching In BST

(d)



As 6 Is Equal To Key (6), So We Have
Found The Key

Searching In BST

(Adapted from *Binary Search Tree | Set 1 (Search and Insertion)*, 2014)

3.2 AVL tree

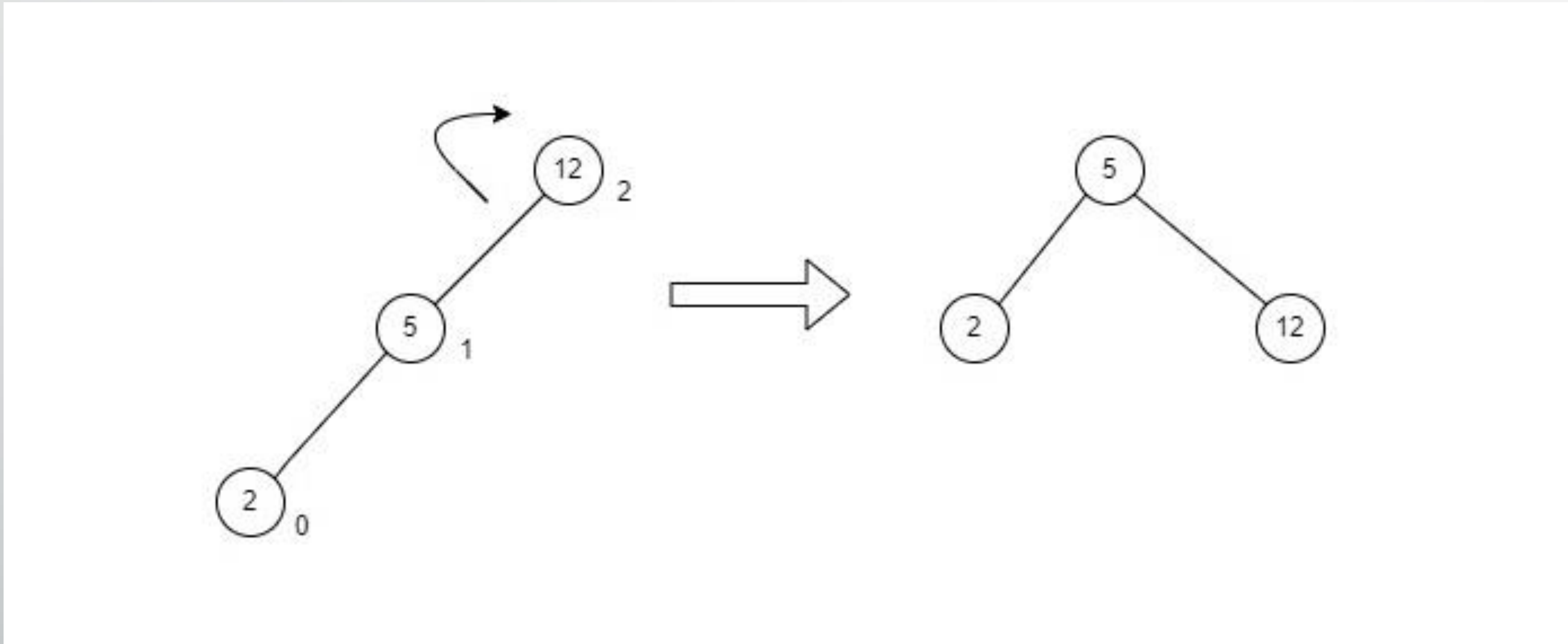
- In AVL trees, the difference between the heights of left and right subtrees, known as the **Balance Factor**, must be at most one. Once the difference exceeds one, the tree automatically executes the balancing algorithm until the difference becomes one again. (*Data Structure and Algorithms - AVL Trees - Tutorialspoint*, n.d.).
- BALANCE FACTOR = HEIGHT(LEFT SUBTREE) – HEIGHT(RIGHT SUBTREE)
- There are usually four cases of rotation in the balancing algorithm of AVL trees: LL, RR, LR, RL.
- Let us briefly describe each of these as shared by (*Data Structure and Algorithms - AVL Trees - Tutorialspoint*, n.d.)

3.2 AVL tree

- LL Rotations : LL rotation is performed when the node is inserted into the right subtree leading to an unbalanced tree. This is a single left rotation to make the tree balanced again. The node where the unbalance occurs becomes the left child and the newly added node becomes the right child with the middle node as the parent node. (figure 18)
- RR Rotations: RR rotation is performed when the node is inserted into the left subtree leading to an unbalanced tree. This is a single right rotation to make the tree balanced again. The node where the unbalance occurs becomes the right child and the newly added node becomes the left child with the middle node as the parent node. (figure 19)

Figure 18

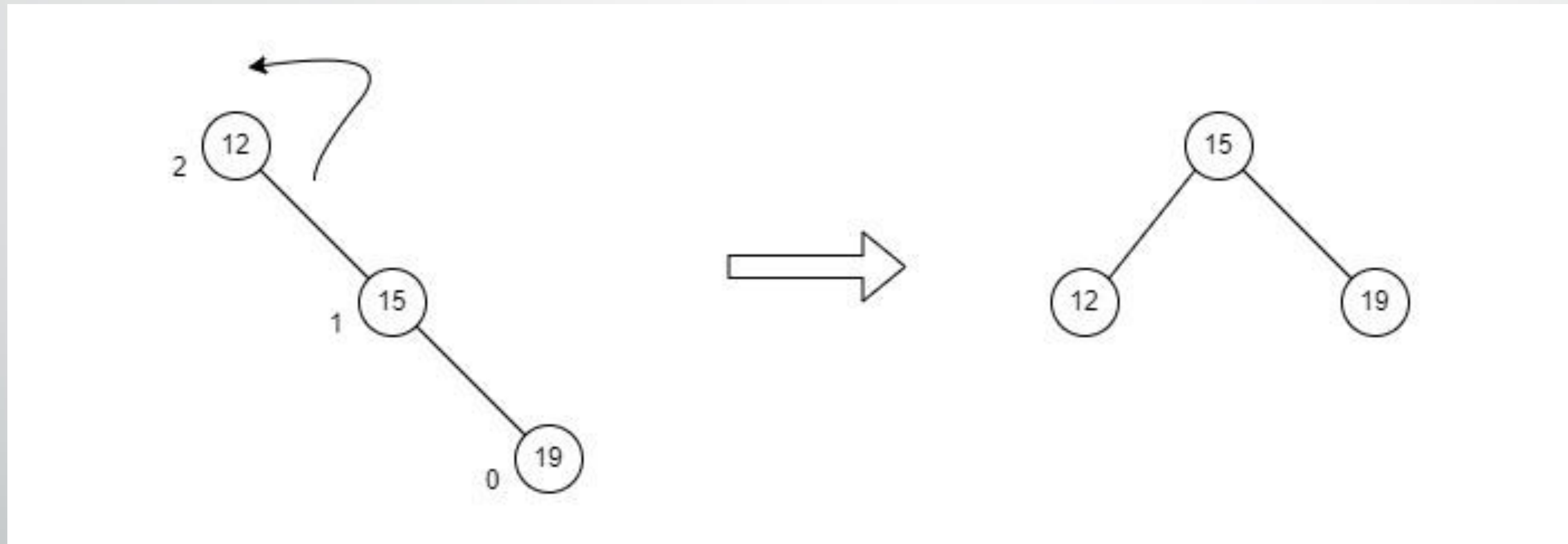
LL rotation



(From (*Data Structure and Algorithms - AVL Trees - Tutorialspoint*, n.d.)

Figure 19

RR rotation



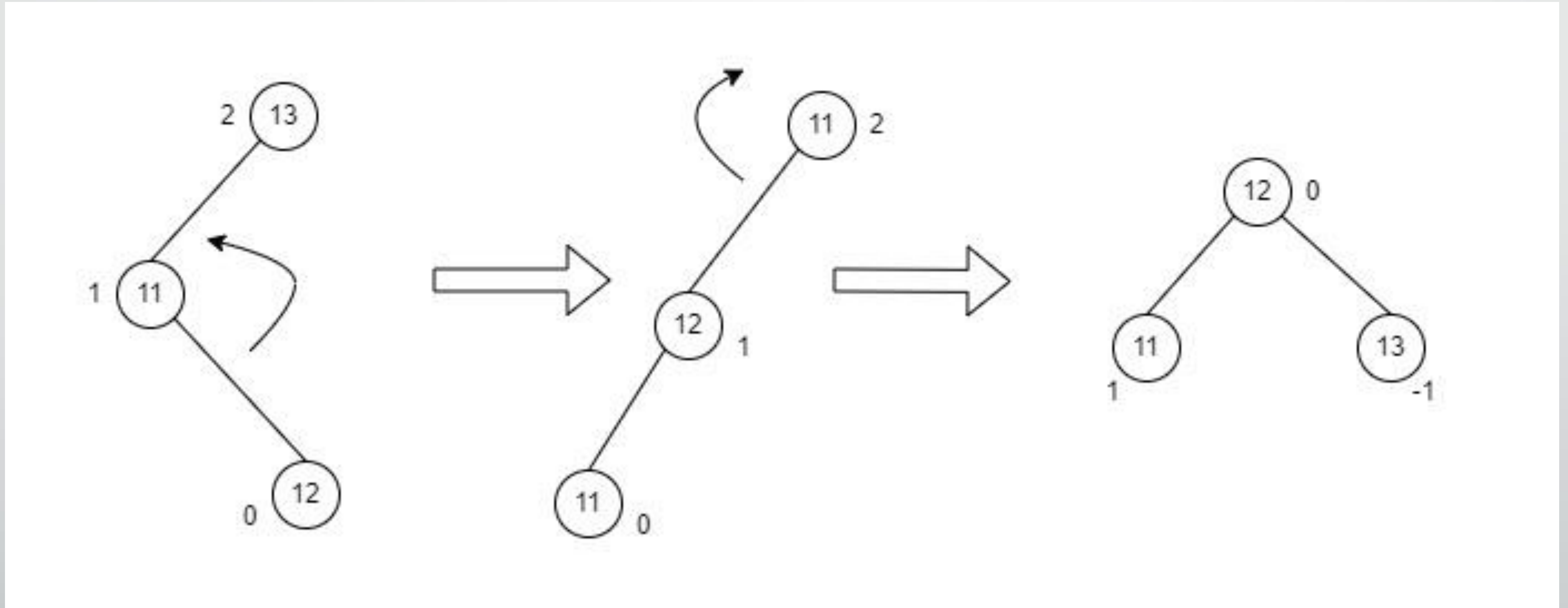
(From (*Data Structure and Algorithms - AVL Trees - Tutorialspoint*, n.d.)

3.2 AVL tree

- LR Rotations: LR rotation is the extended version of the previous single rotations, also called a double rotation. It is performed when a node is inserted into the right subtree of the left subtree. The LR rotation is a combination of the left rotation followed by the right rotation. (figure 20)
- RL Rotations: RL rotation is also the extended version of the previous single rotations, hence it is called a double rotation and it is performed if a node is inserted into the left subtree of the right subtree. The RL rotation is a combination of the right rotation followed by the left rotation (figure 21)
- Based on these rotations the basic operations of AVL trees namely insertion and deletion, can be performed. An example of insertion is demonstrated here.

Figure 20

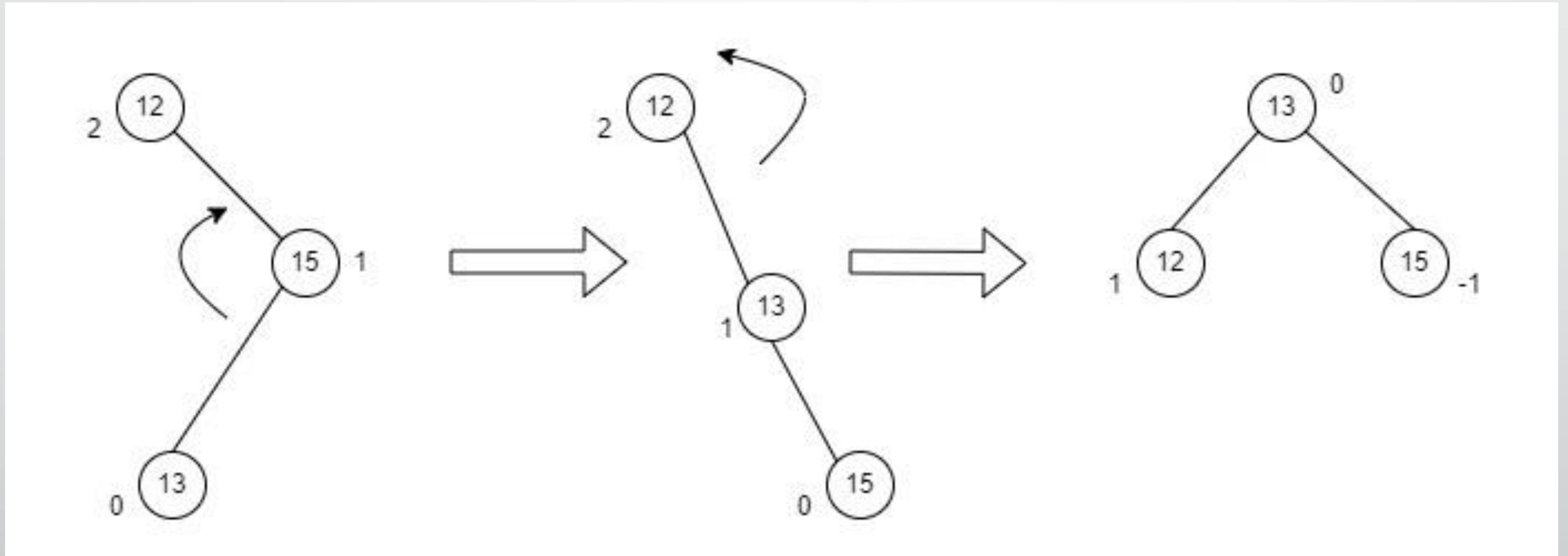
LR rotation



(From (*Data Structure and Algorithms - AVL Trees - Tutorialspoint*, n.d.)

Figure 21

RL rotation



(From (*Data Structure and Algorithms - AVL Trees - Tutorialspoint*, n.d.)

3.2.1 Insertion

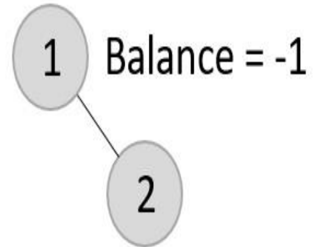
- As described by (*Data Structure and Algorithms - AVL Trees - Tutorialspoint*, n.d.) :
- The data is inserted into the AVL Tree by following the Binary Search Tree property of insertion, i.e. the left subtree must contain elements less than the root value and right subtree must contain all the greater elements. However, in AVL Trees, after the insertion of each element, the balance factor of the tree is checked; if it does not exceed 1, the tree is left as it is. But if the balance factor exceeds 1, a balancing algorithm is applied to readjust the tree such that balance factor becomes less than or equal to 1 again.
- Figure 22 demonstrates an example of arranging 7 integers into an AVL tree.

Figure 22

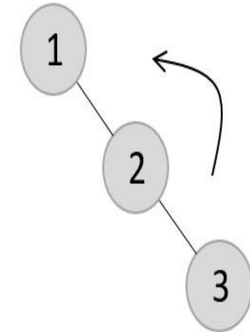
AVL tree of 7 integers

Step 1

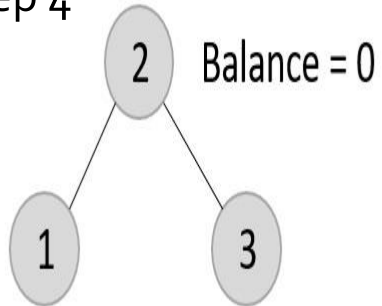
Step 2



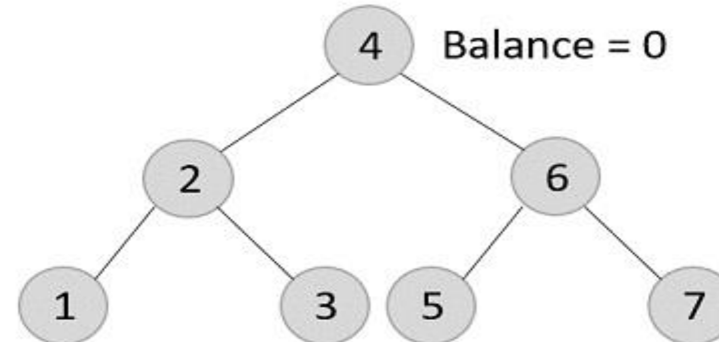
Step 3



Step 4



Step 5



(Adapted from *Data Structure and Algorithms - AVL Trees - Tutorialspoint*, n.d.)

3.3 Other trees

- Due to time and space considerations it is not possible to cover all the trees that there; further, as you may have noticed the details of implementations have been left out to save on space. However, it is hoped that with the algorithms and shared examples the learner will understand how to use these trees to solve problems in the computer science domain.
- Nonetheless the learner is encouraged to do some further reading on these trees:
  - 2-3 tree
  - Red black tree
  - B tree

Summary (1/2)

- The number of sub-trees of a given node is called its degree. A tree consisting of only one node has a degree of 0. This one tree node is also considered as a tree by all standards.
- A node with a degree of 0 is called a leaf node.
- A ***binary tree*** is an ordered tree with the following properties: Every node has at most two children, each child node is labeled as being either a ***left child*** or a ***right child,*** and a left child precedes a right child in the order of children of a node.
- A node's depth corresponds to the level it occupies; further the depth is also the number of edges to the node from the root.
- A full binary tree is a binary tree in which each interior node contains two children.

Summary (2/2)

- We define a balanced tree as one where “the heights of any node’s left and right subtrees differ by at most one.
- Three common algorithms used for traversal: preorder and postorder traversal, breadth-first traversal, and Inorder traversal.
- A max-heap has the property, known as the heap order property, that for each non-leaf node V , the value in V is greater than the value of its two children; the min-heap has the opposite property. For each non-leaf node V , the value in V is smaller than the value of its two children.
- Other trees include BST, AVL, red black, B, and 2-3.

References

- Baka, B. (2017). *Python Data Structures and Algorithms*. Packt Publishing Ltd.
- *Binary Search Tree - GeeksforGeeks*. (2015). GeeksforGeeks. https://www.geeksforgeeks.org/binary-search-tree-data-structure/
- *Binary Search Tree | Set 1 (Search and Insertion)*. (2014, January 30). GeeksforGeeks. https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/
- *Data Structure - Binary Search Tree - Tutorialspoint*. (n.d.). Www.tutorialspoint.com. Retrieved October 24, 2023, from https://www.tutorialspoint.com/data_structures_algorithms/binary_search_tree.htm
- *Data Structure and Algorithms - AVL Trees - Tutorialspoint*. (n.d.). Www.tutorialspoint.com. Retrieved October 24, 2023, from https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm

References

- Ghosh, B. (2023, September 28). *Difference Between Full, Complete, and Perfect Tree*. Baeldung.com. https://www.baeldung.com/cs/full-vs-complete-vs-perfect-tree#:~:text=Complete%20Binary%20Tree%3A%20All%20levels,differ%20by%20at%20most%20one
- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2018). *Data structures and algorithms in Python*. Wiley.
- *Heap Data Structures - Tutorialspoint*. (n.d.). Www.tutorialspoint.com. Retrieved October 24, 2023, from https://www.tutorialspoint.com/data_structures_algorithms/heap_data_structure.htm
- Necaise, R. D. (2011). *Data structures and algorithms using Python*. John Wiley And Sons.