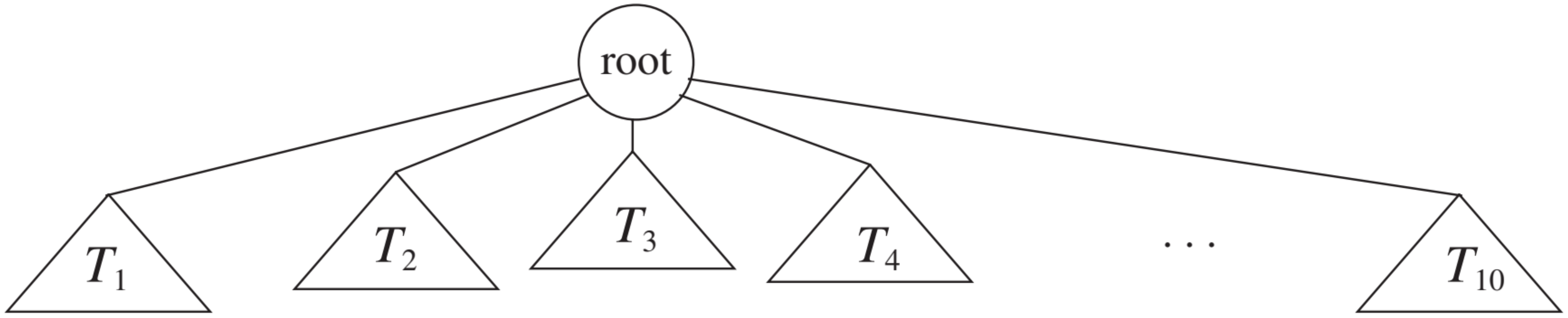
Algorithms and Complexity

Module 3

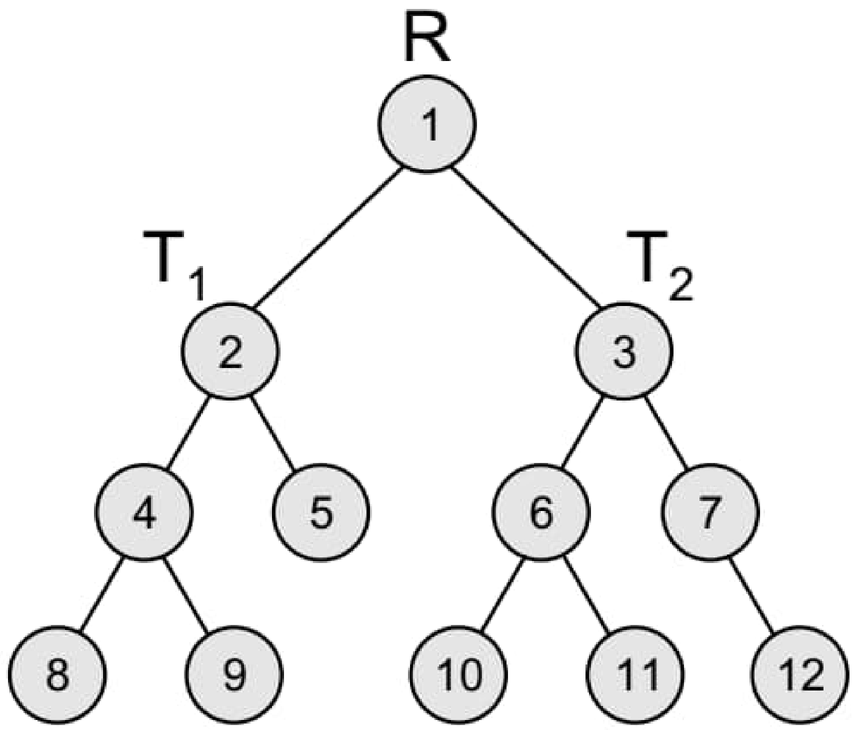
Trees

- A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical order
- One of the nodes is designated as the root node
- The remaining nodes can be partitioned into disjoint sets such that each set is a sub-tree of the root



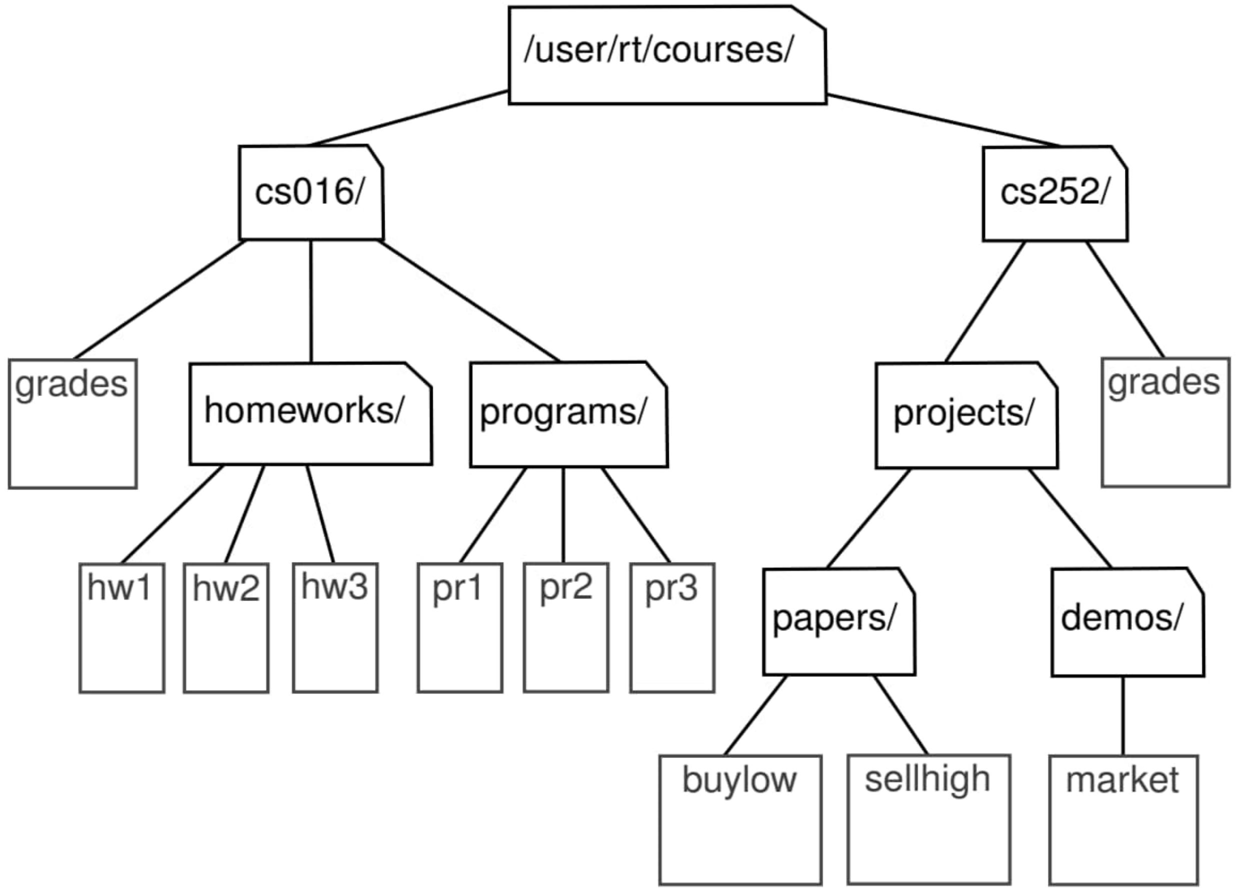
Trees

- The simplest form of a tree is a binary tree
- A binary tree consists of a root node and left and right sub-trees, where both sub-trees are also binary trees
- R is the root node and T1 and T2 are the left and right subtrees of R
- If T1 is non-empty, then T1 is said to be the left successor of R and if T2 is non-empty, it is called the right successor of R
- Node 2 is the left child and node 3 is the right child of the root node 1



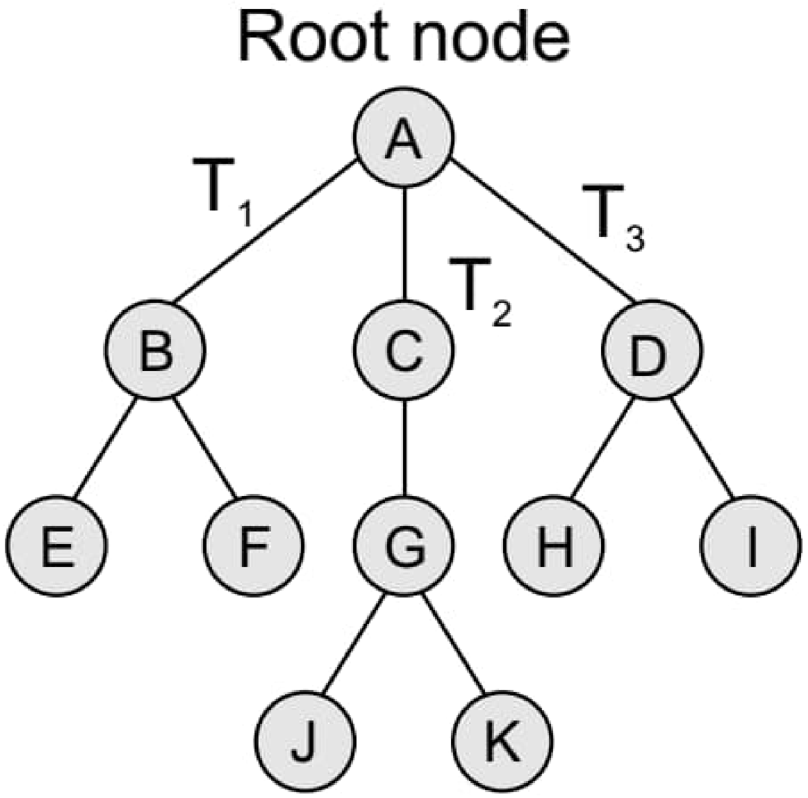
Trees

- Tree structures makes it possible to implement many algorithms much faster than when using linear data structures
- Trees also provide a natural organization for data and have become ubiquitous structures in file systems, graphical user interfaces, databases, Web sites, and other computer systems



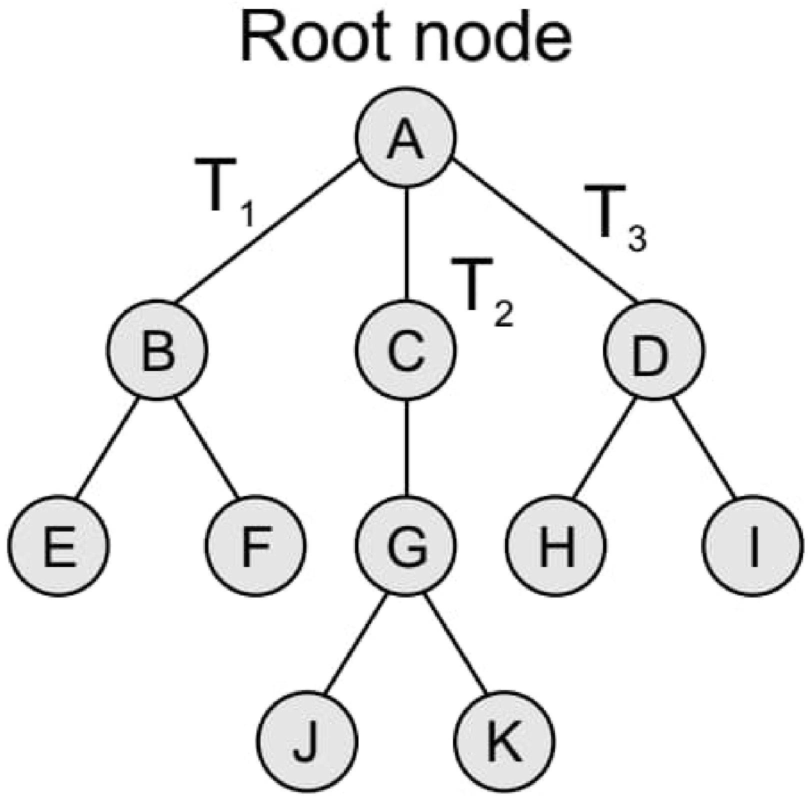
Trees

- Basic terminology:
  - Root node: the topmost node in the tree. If R = None, then it means the tree is empty
  - Sub-trees: if the root node R is not NULL, then the trees T1, T2, and T3 are called the sub-trees of R
  - Leaf node: a node that has no children (terminal node)
  - Edge: a pair of nodes (u,v) such that u is the parent of v, or vice versa
  - Path: a sequence of consecutive edges, e.g., the path from the root node A to node I is given as: A, D, and I



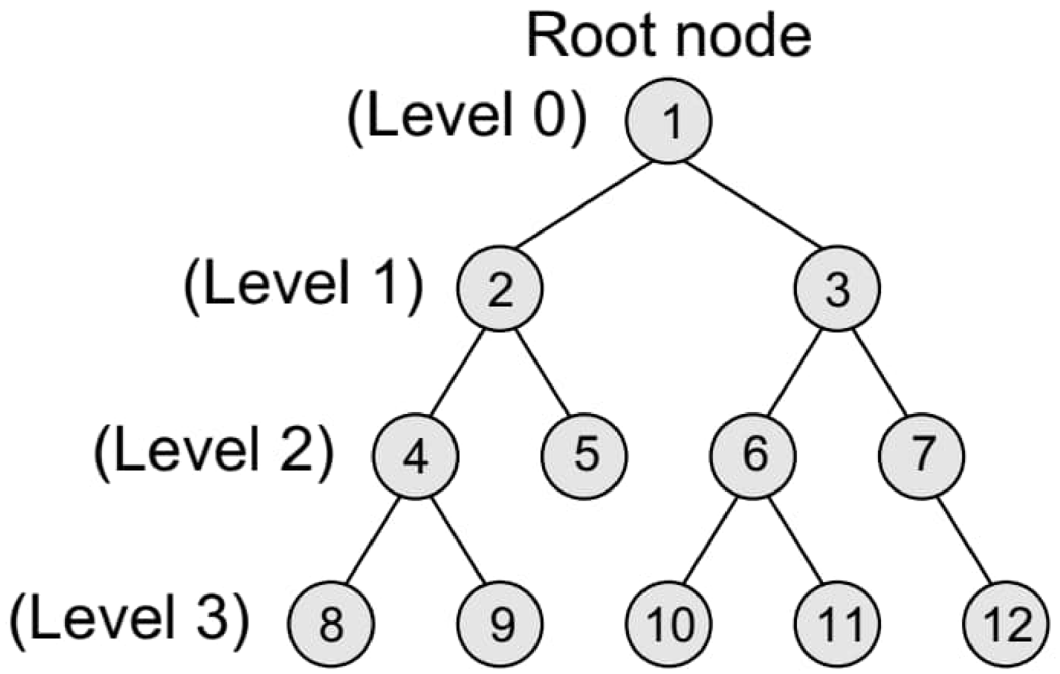
Trees

- Basic terminology:
  - Ancestor node: any predecessor node on the path from root to that node, e.g., nodes A, C, and G are the ancestors of node K
  - Descendant node: any successor node on any path from the node to a leaf node, e.g., nodes C, G, J, and K are the descendants of node A
  - Degree: the number of children that a node has. The degree of a leaf node is zero
  - In-degree: the number of edges arriving at that node



Trees

- Basic terminology:
  - Out-degree: the number of edges leaving that node
  - Level number: every node in the tree is assigned a level number in such a way that the root node is at level 0, children of the root node are at level number 1
  - Sibling: all nodes that are at the same level and share the same parent are called siblings
  - Height: the total number of nodes on the path from the root node to the deepest node in the tree

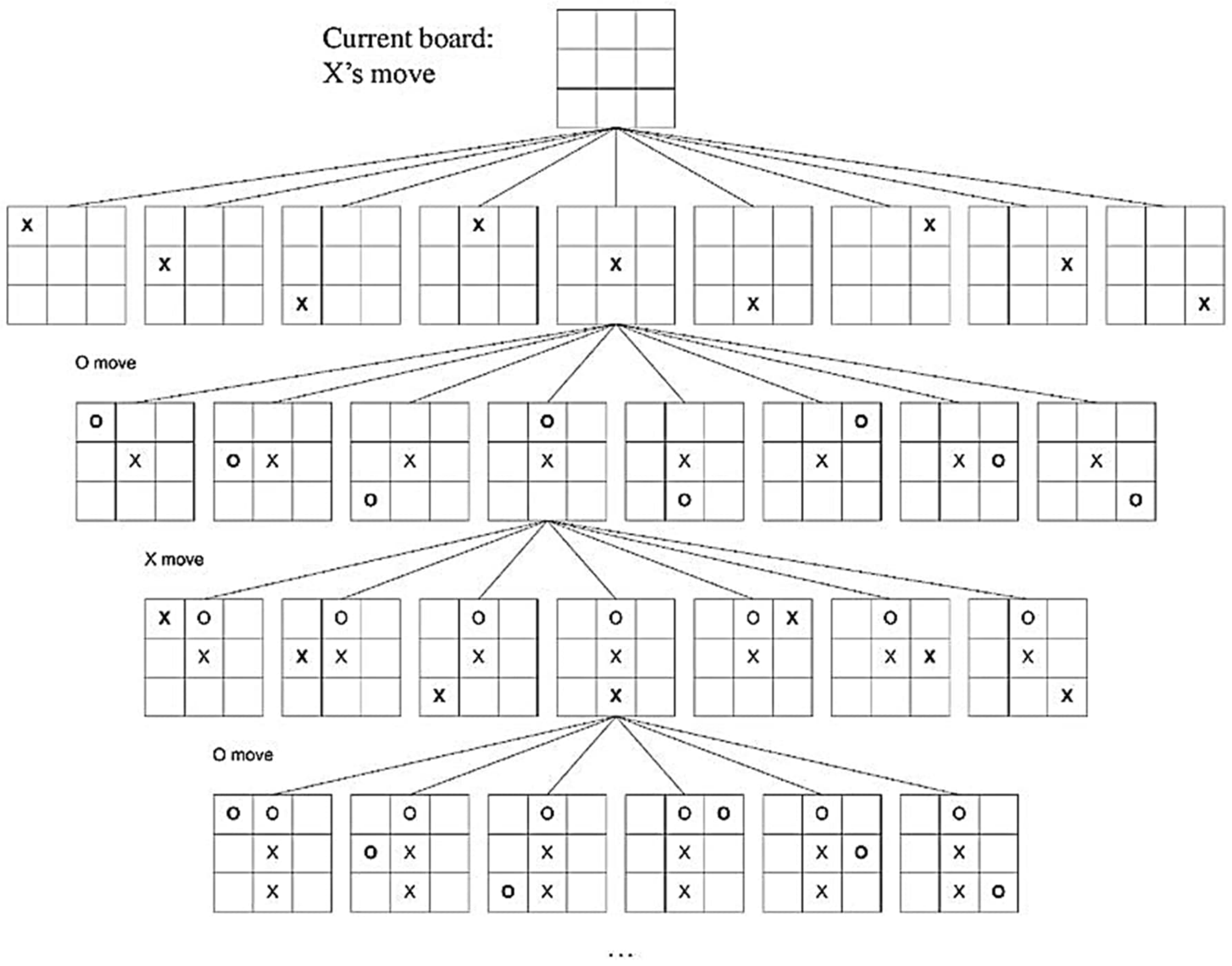


Trees

- Example: game trees
- All the possible moves by both players are written down onto a tree
- Root node represents the current status
- Internal nodes at odd levels correspond to positions in which the first player is to move
- Internal nodes at even levels correspond to positions in which the second player is to move
- Leaves correspond to positions at which the game has ended. One player or the other has won, or perhaps the game is drawn

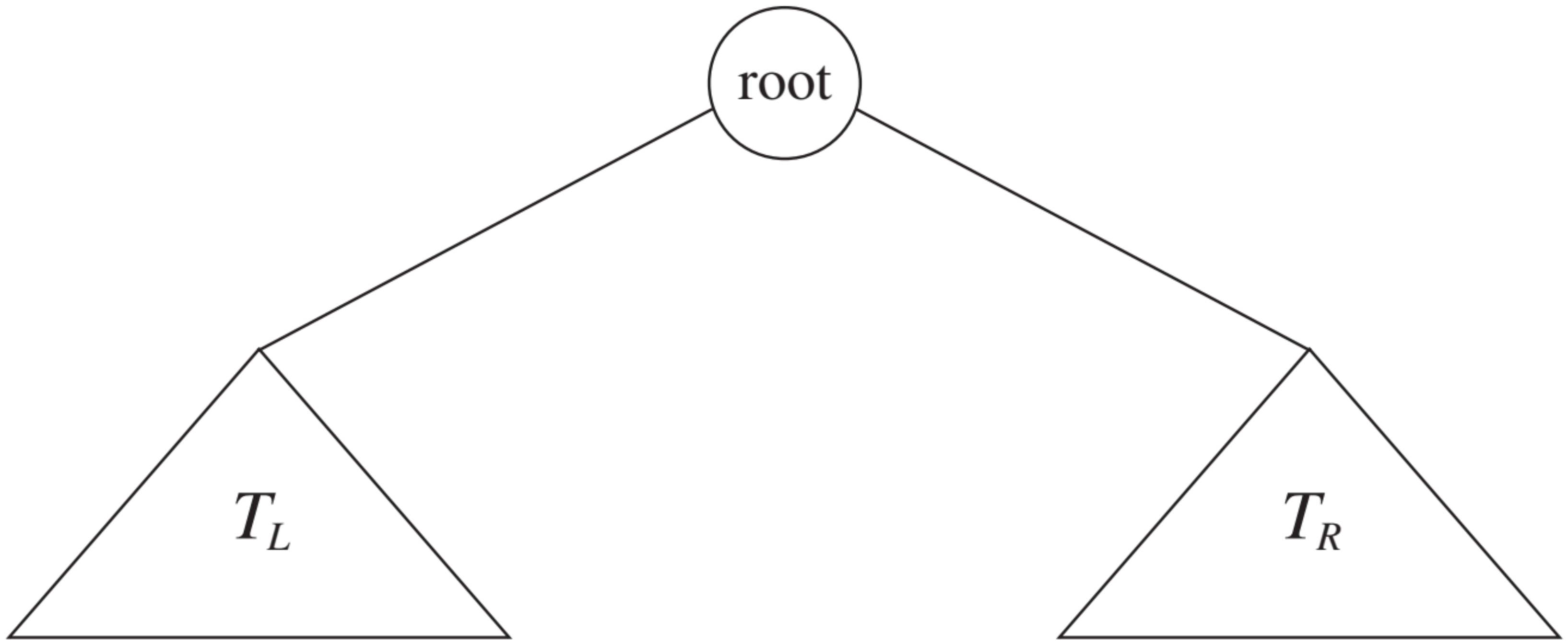
Trees

- Partial game tree for Tic-Tac-Toe



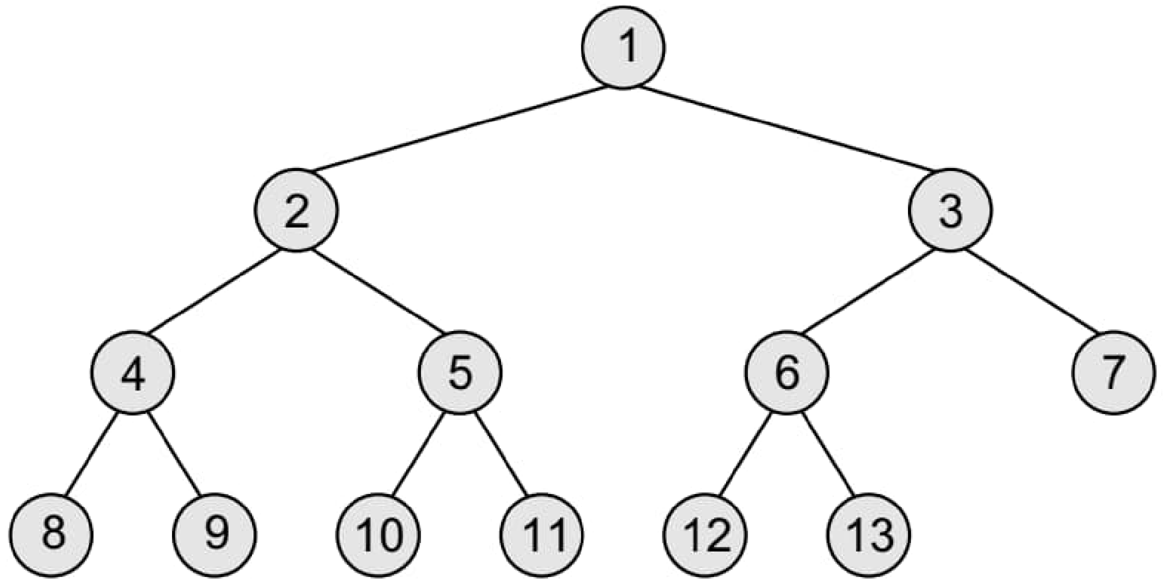
Binary Trees

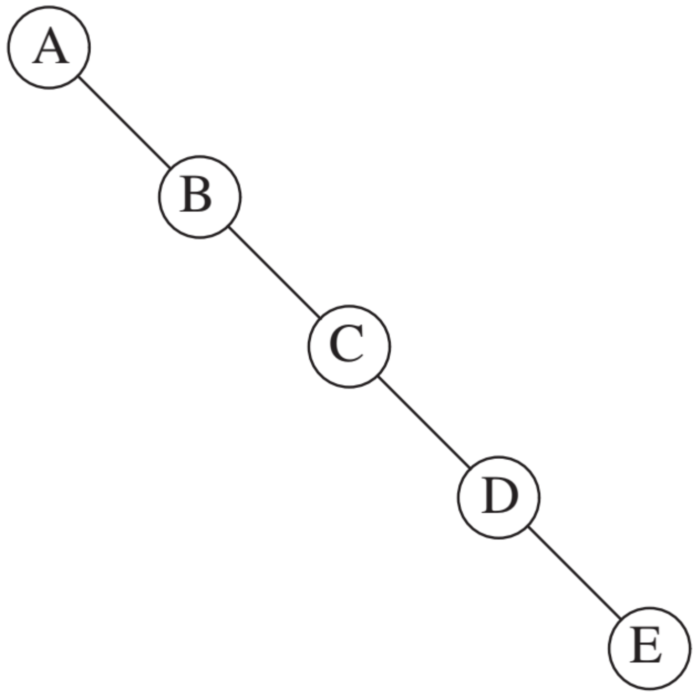
- A binary tree is a tree with the following properties:
  - Every node has at most two children.
  - Each child node is labeled as being either a left child or a right child.
  - A left child precedes a right child in the order of children of a node.



Binary Trees

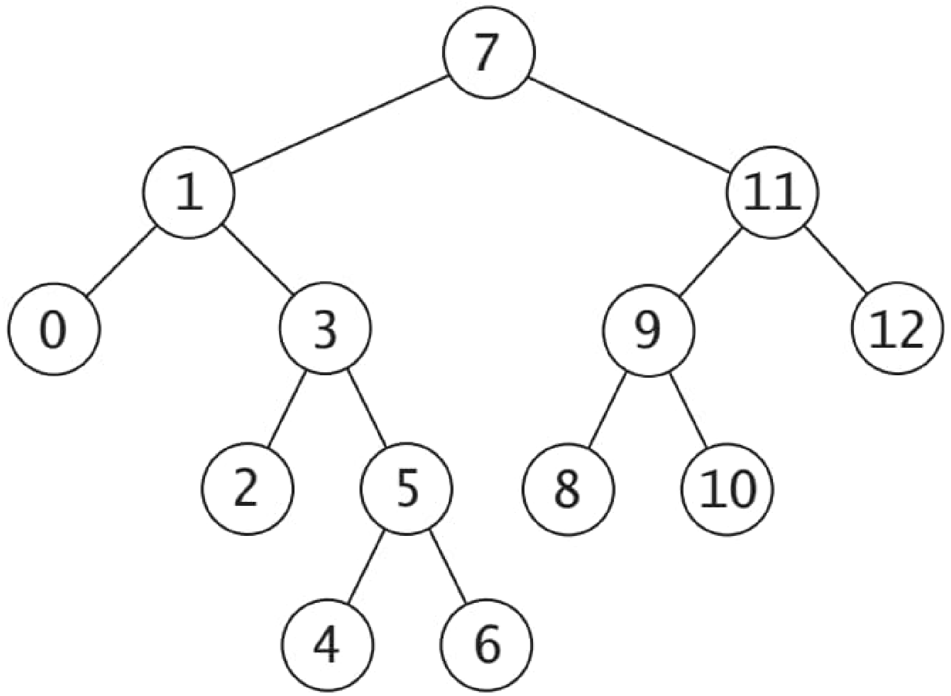
- A binary tree of height *h* has at least *h* nodes and at most $2^h - 1$ nodes
- This is because every level will have at least one node and can have at most 2 nodes
- The height of a binary tree with *n* nodes is at least $\log_2(n+1)$ and at most *n*

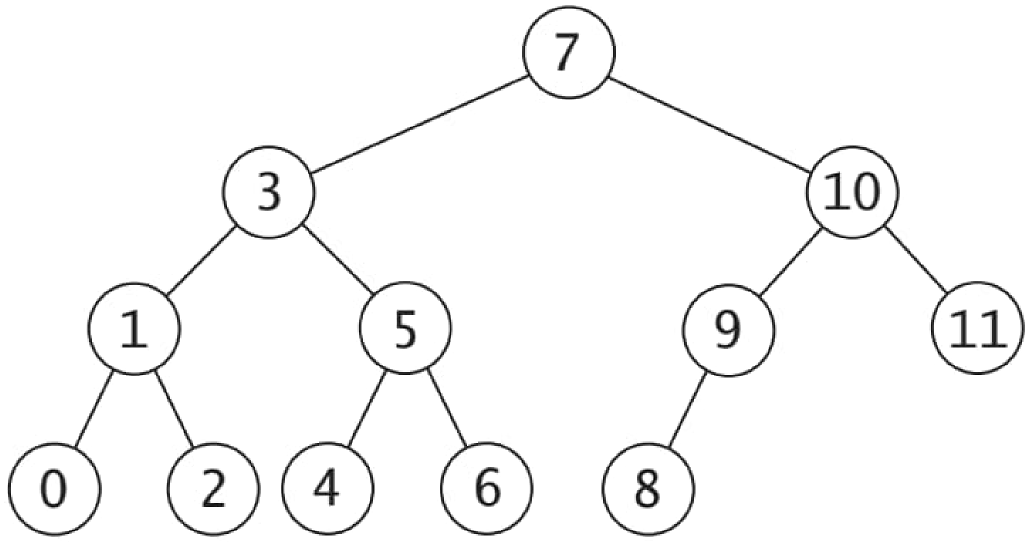




Binary Trees

- Full (proper) binary tree: a binary tree in which each node has 2 or 0 children
- Complete binary tree: a binary tree that satisfies two properties:
  - every level, except possibly the last, is completely filled
  - all nodes appear as far left as possible



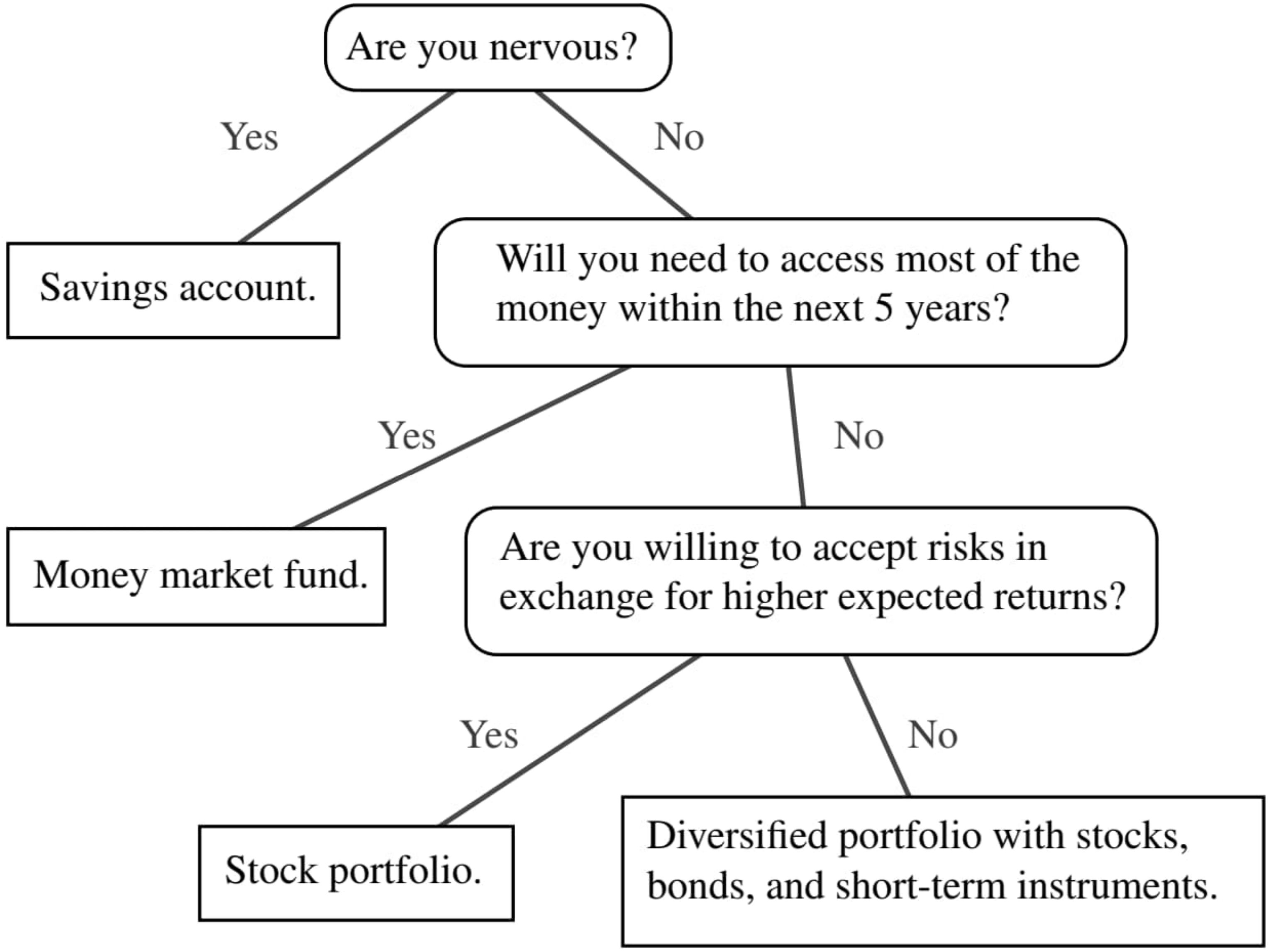


Binary Trees

- Example: decision trees
- 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.”
- 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*

Binary Trees

- A decision tree providing investment advice

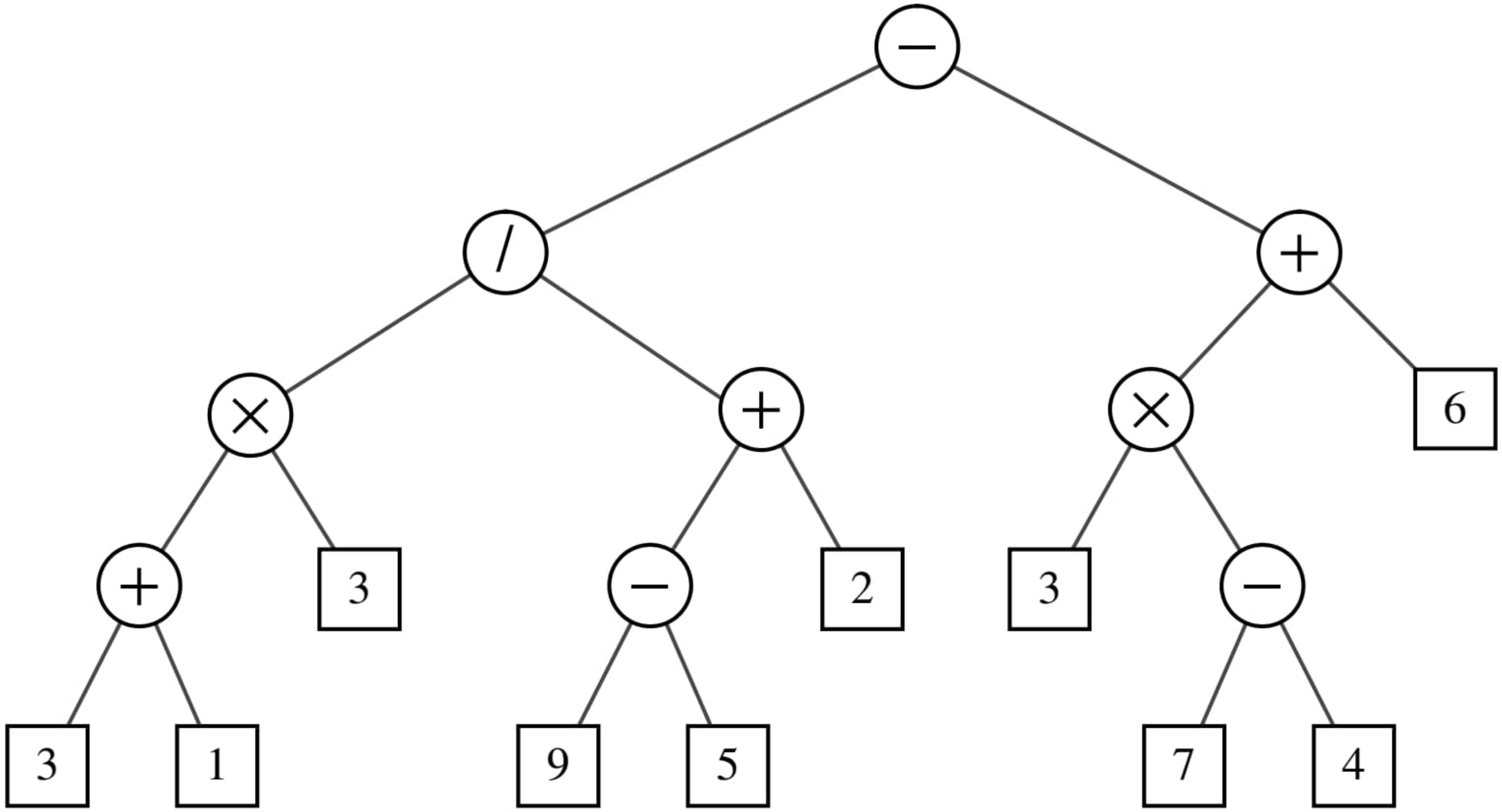

Are you nervous?
Yes
No
Savings account.
Will you need to access most of the money within the next 5 years?
Yes
No
Money market fund.
Are you willing to accept risks in exchange for higher expected returns?
Yes
No
Stock portfolio.
Diversified portfolio with stocks, bonds, and short-term instruments.

Binary Trees

- Example: arithmetic expression trees
- An arithmetic expression can be represented by a binary tree
- Leaves of the tree are associated with variables or constants, and internal nodes are associated with one of the operators +, −, ×, and /
- Each node in such a tree has a value associated with it:
  - for a leaf node, the value is that of its variable or constant
  - For an internal node, the value is defined by applying its operation to the values of its children
- An arithmetic expression tree is a full binary tree, since each operator +, −, ×, and / takes exactly two operands

Binary Trees

- Binary tree representing the arithmetic expression: ((((3 + 1) × 3)/((9 – 5) + 2)) – ((3 × (7 – 4)) + 6))

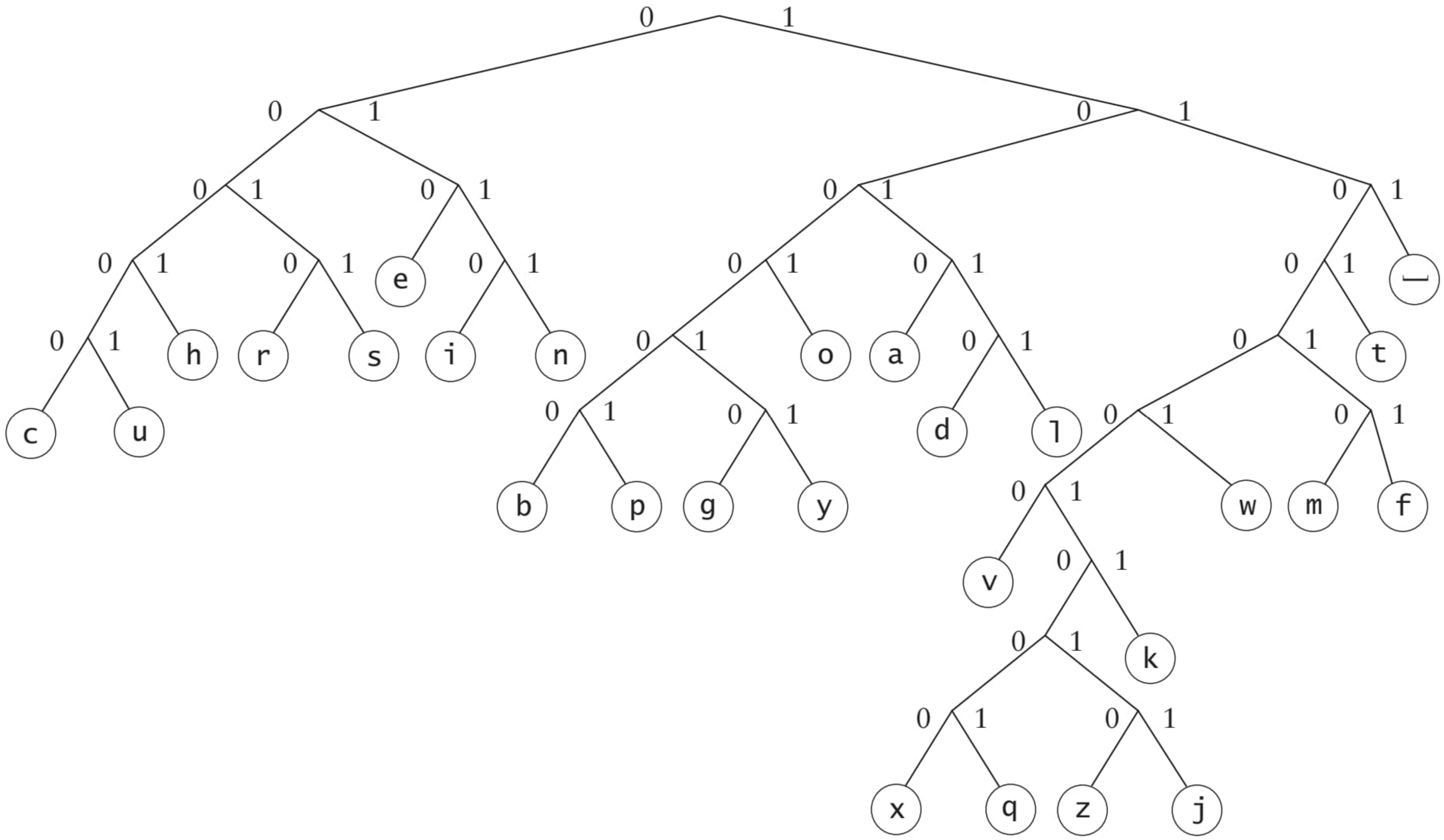


Binary Trees

- Example: Huffman encoding tree
- A Huffman code uses different numbers of bits to encode letters
- It uses fewer bits for the more common letters (for example, space, *e*, *a*, and *t*) and more bits for the less common letters (for example, *q*, *x*, and *z*)
- Many programs that compress files use Huffman encoding to generate smaller files to save disk space or to reduce the time spent sending the files over the Internet

Binary Trees

- The Huffman encoding tree for an alphabet consisting of the lowercase letters and the space character



Binary Trees

- All the characters are at leaf nodes
- To determine the code for a letter, you form a binary string by tracing the path from the root node to that letter
- Each time you go left, append a 0, and each time you go right, append a 1
- The two characters with a depth of 3 (space, *e*) are the most common and, therefore, have the shortest codes (111, 010)
- The next most common characters (*a*, *o*, *i*, and so forth) have a depth of 4

Binary Trees

- You could store the code for each letter in an array and encode each letter in a file by looking up its code in the array
- However, to decode a file of letters and spaces, you walk down the Huffman tree, starting at the root, until you reach a letter
- Once you have reached a letter, append that letter to the output text and go back to the root
- Example:

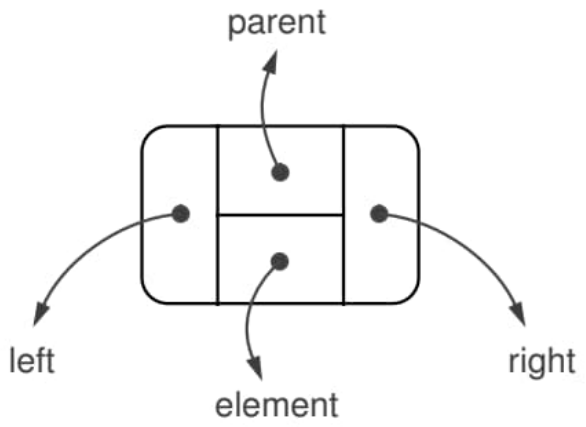
100000011001111010001010001111111010010010010

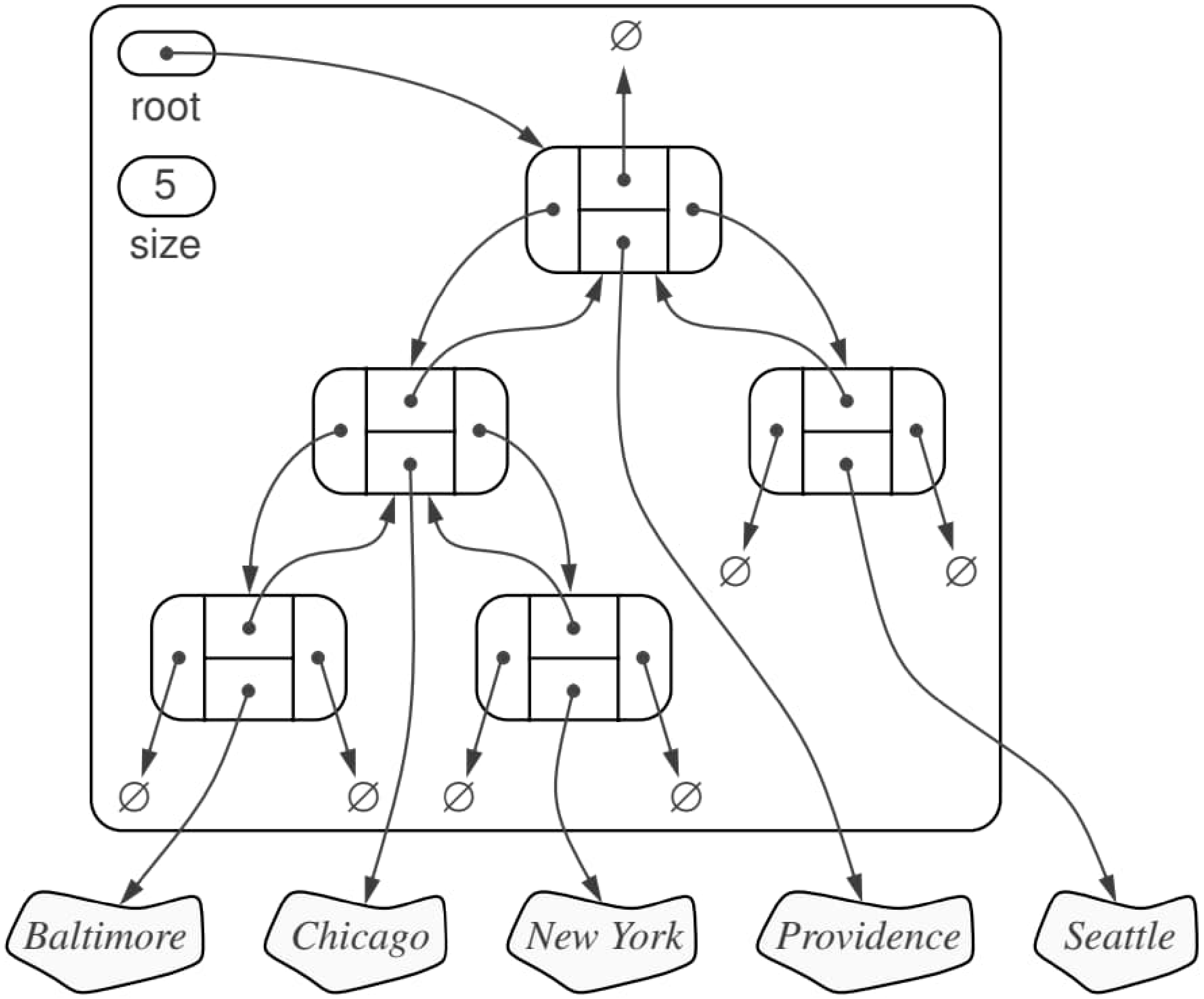
Binary Trees

- One way to implement a binary tree *T* is to use a linked structure, with a node that maintains references to the element stored at a position *p* and to the nodes associated with the children and parent of *p*
- If p is the root of *T*, then the parent field of *p* is None
- If *p* does not have a left/right child, the associated field is None
- The tree itself maintains an instance variable storing a reference to the root node, and a variable `size` that represents the overall number of nodes of *T*

Binary Trees

- A linked structure for representing a single node and a binary tree



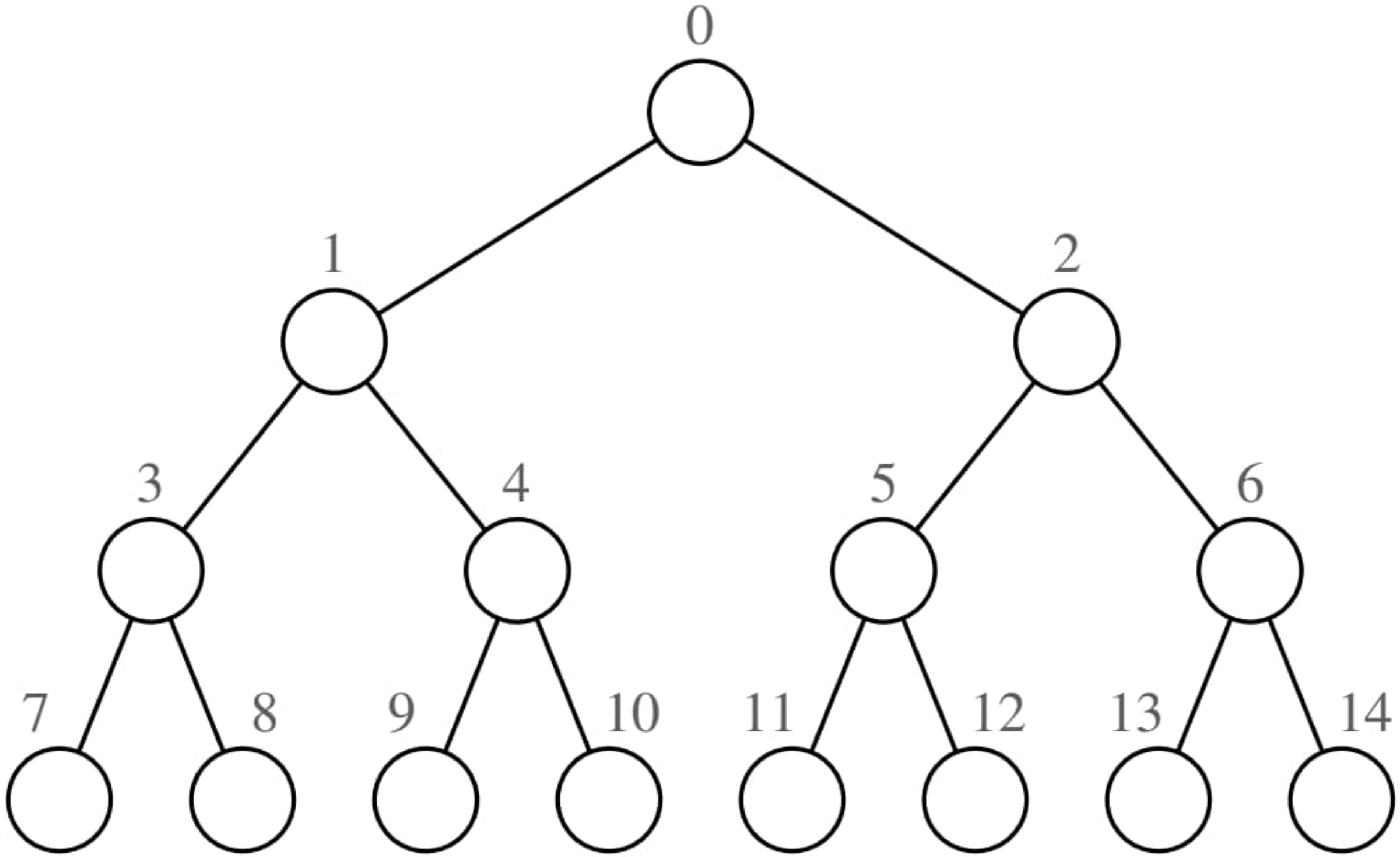


Binary Trees

- Alternatively, a binary tree *T* representation can be based on a way of numbering the nodes of *T*
- For every node *p* of *T*, let *f*(*p*) be the integer defined as follows:
  - If *p* is the root of *T*, then *f*(*p*) = 0
  - If *p* is the left child of node *q*, then *f*(*p*) = 2*f*(*q*) + 1
  - If *p* is the right child of node *q*, then f(*p*) = 2*f*(*q*) + 2
- The numbering function *f* is known as a level numbering of the positions in a binary tree *T*
- It numbers the positions on each level of *T* in increasing order from left to right

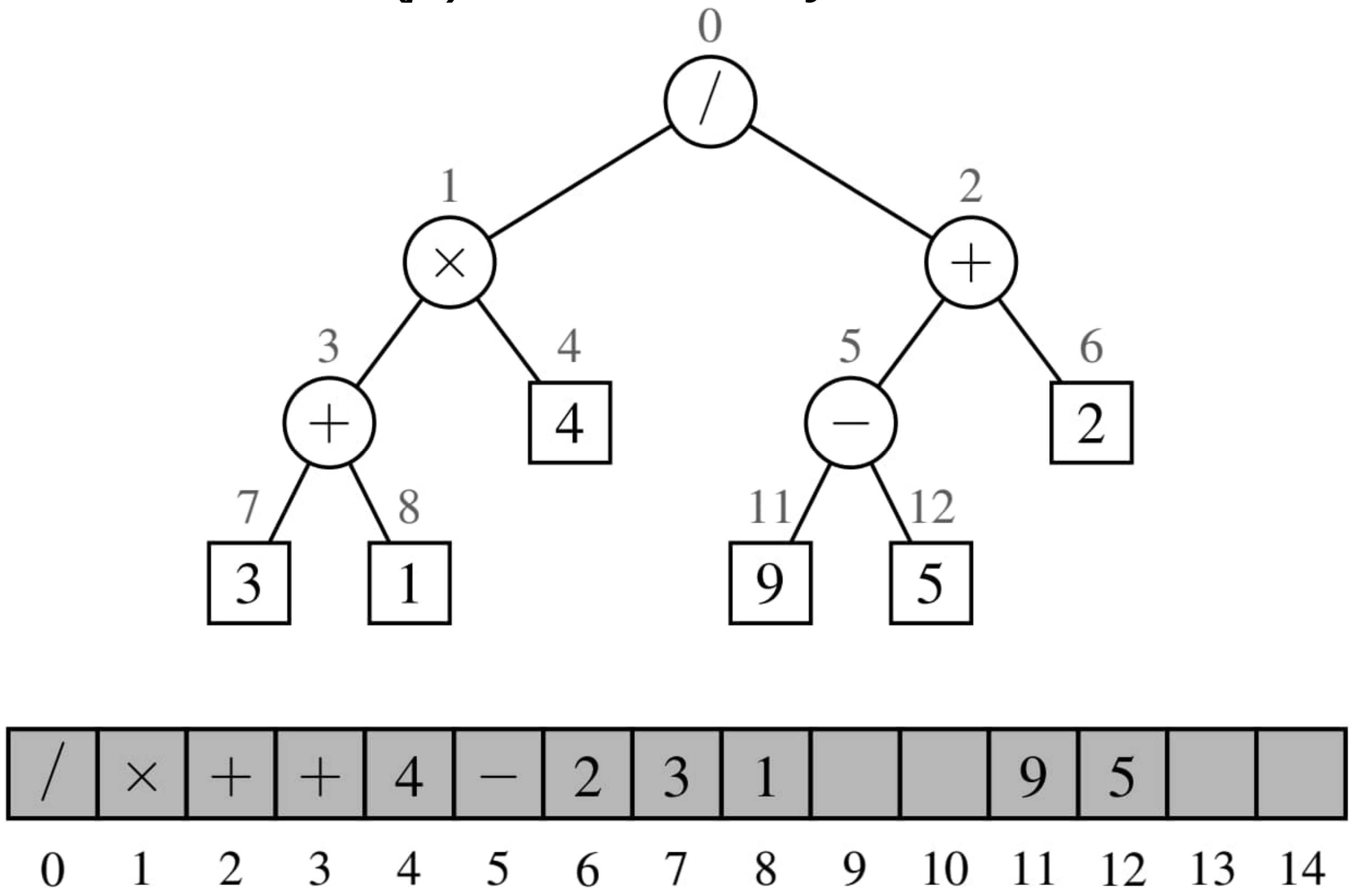
Binary Trees

- General scheme of binary tree level numbering



Binary Trees

- With the level numbering function, a binary tree *T* can be represented by means of an array-based structure *A* (such as a Python list), with the element at position p of *T* stored at index *f*(*p*) of the array

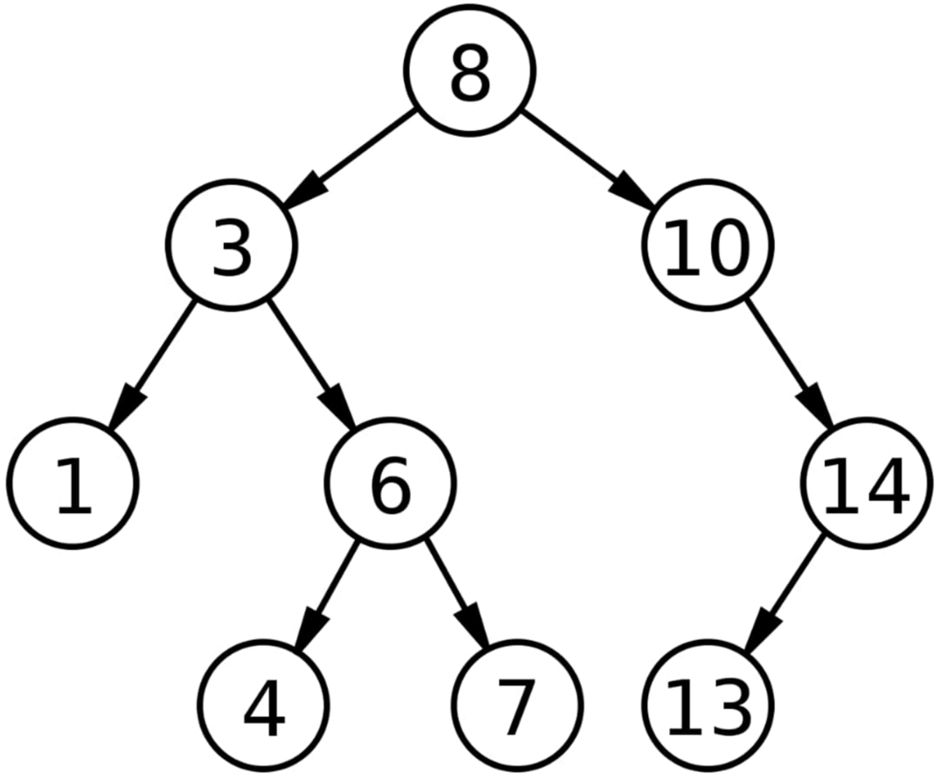


Binary Trees

- With an array-based representation of a binary tree, a node *p* can be represented by the single integer *f*(*p*)
- Methods such as `root`, `parent`, `left`, and `right` can be implemented using simple arithmetic operations on the number *f*(*p*)
- The space usage of an array-based representation depends greatly on the shape of the tree
- In the worst case, array length $N = 2^n - 1$, where *n* is the number of nodes of *T*
- Some update operations for trees cannot be efficiently supported with the array representation
- For example, deleting a node and promoting its child takes *O*(*n*) time because it is not just the child that moves locations within the array, but all descendants of that child

Binary Search Trees (BSTs)

- A set of nodes *T* is a binary search tree (BST) if either of the following is true:
  - *T* is empty
  - If *T* is not empty, its root node has two subtrees, T_L and T_R, such that T_L and T_R are binary search trees and the value in the root node of *T* is greater than all values in T_L and is less than all values in T_R



Binary Search Trees (BSTs)

- The order relations in a BST facilitate searching the tree
- A recursive algorithm for searching a BST:
  - 1. if the tree is empty return None (target is not found)
  - 2. else if the target matches the root node’s data, return the data stored at the root node
  - 3. else if the target is less than the root node’s data, return the result of searching the left subtree of the root
  - 4. else return the result of searching the right subtree of the root
- The first two cases are base cases

Binary Search Trees (BSTs)

- Just as with a binary search of an array, each probe into the BST has the potential of eliminating half the elements in the tree
- If the BST is relatively balanced (the depths of the leaves are approximately the same), searching a BST is an *O*(log *n*) process, just like a binary search of an ordered array
- A BST never has to be sorted, because its elements always satisfy the required order relations
- When new elements are inserted (or removed), the binary search tree property can be maintained

Binary Search Trees (BSTs)

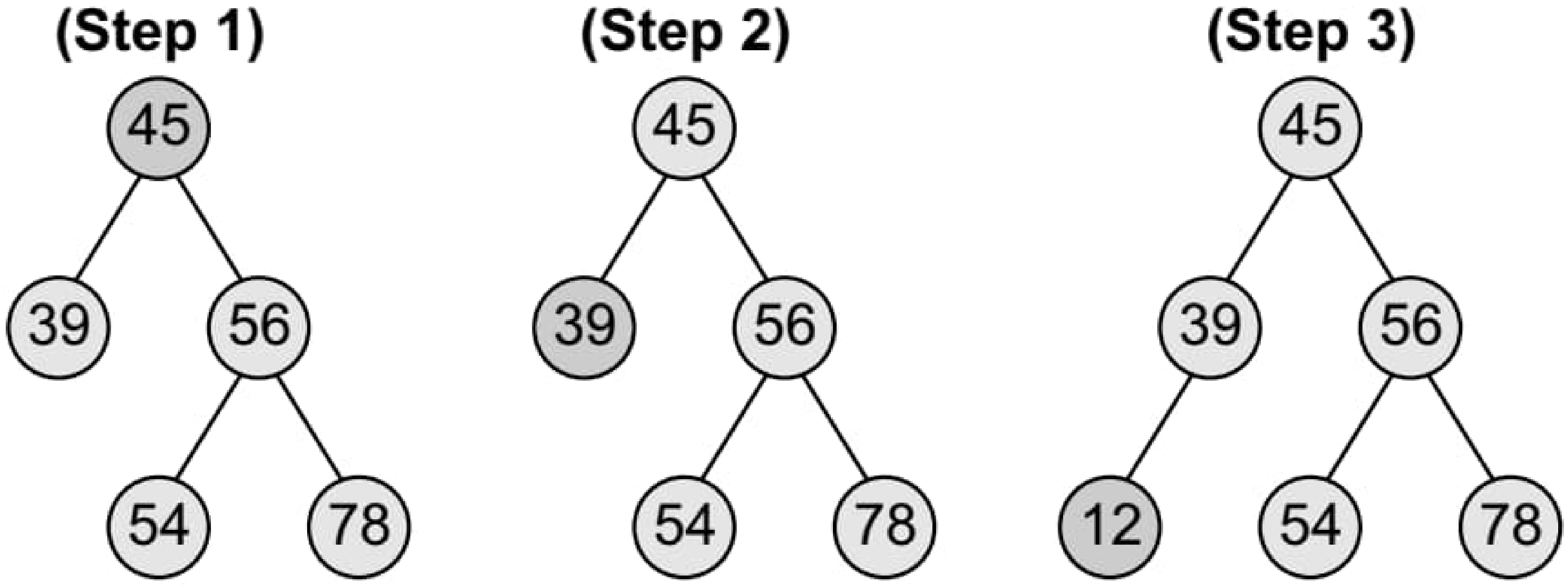
- If the BST is unbalanced, with one subtree much larger than the other, it can lead to slow search and insertion times. Unbalanced trees can also lead to higher memory usage.
- In the worst case, when each parent node has only one child, the BST becomes a linked list with a linear time complexity for searching and insertion
- Deleting a node in a BST can be complex, especially when the node has two children. In this case, the tree must be restructured to maintain the ordering property

Binary Search Trees (BSTs)

- Inserting a new node in a BST is similar to the search procedure
- First the correct position of the insertion must be found
- Then, the new node is added at that position
- The new node is added by following the rules of the BSTs
- If the new node's value is greater than that of the parent node, the new node is inserted in the right sub-tree, else it is inserted in the left sub-tree
- The insert function requires time proportional to the height of the tree in the worst case
- It takes $O(\log n)$ time to execute in the average case and $O(n)$ time in the worst case

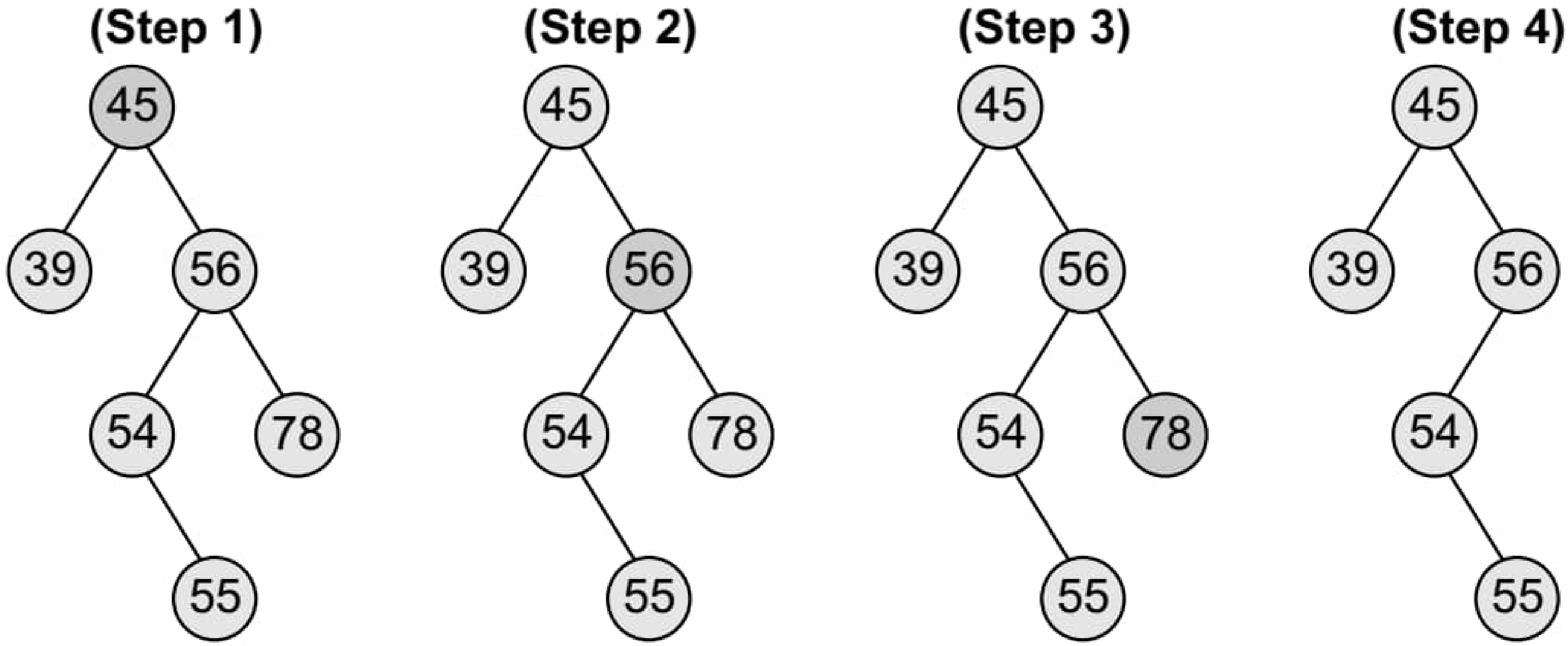
Binary Search Trees (BSTs)

- Inserting a node with value 12 in the given BST



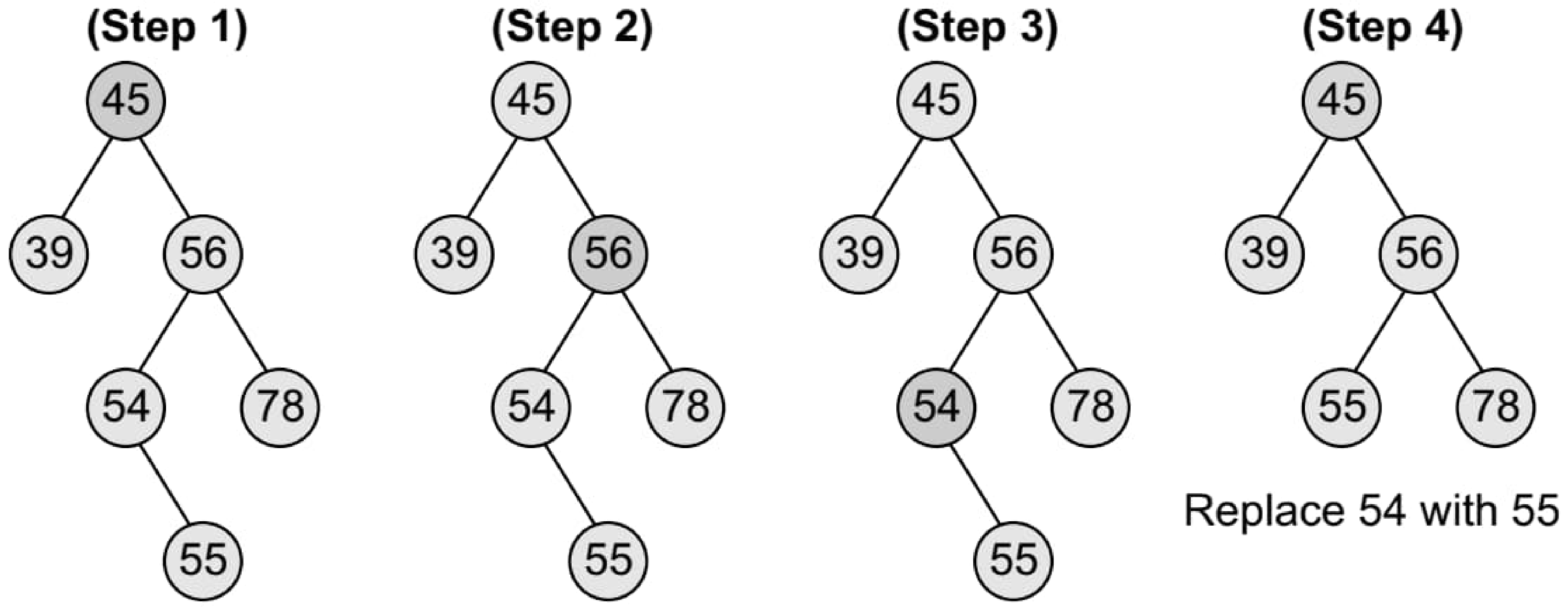
Binary Search Trees (BSTs)

- Deleting a node from a BST differs for nodes with zero, one, and two children
- A node with no children can be simply removed form the tree
- Deleting node 78 from the given BST:



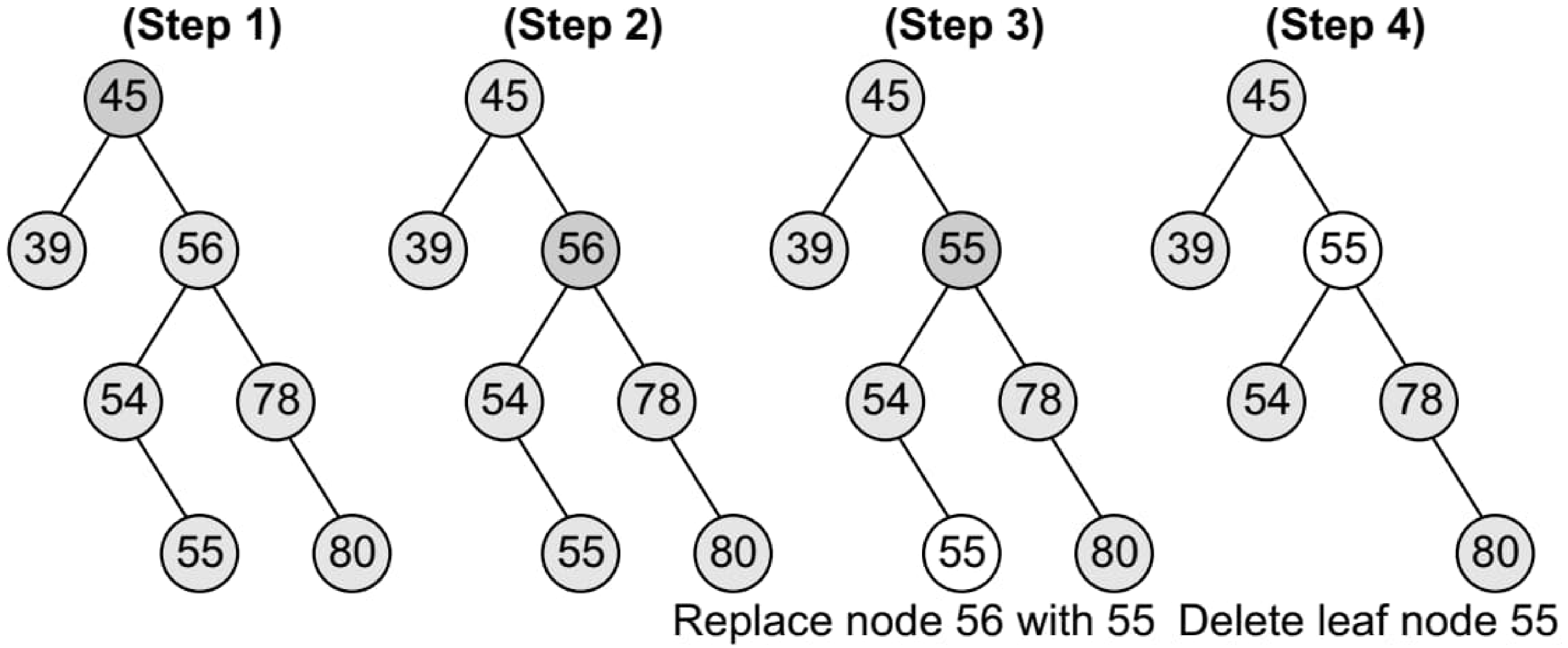
Binary Search Trees (BSTs)

- If the deleted node has one child, the node’s child is set as the child of the node’s parent
- If the node is the left/right child of its parent, the node’s child becomes the left/right child of the node’s parent
- Deleting node 54 from the given BST:



Binary Search Trees (BSTs)

- If the deleted node has two children, the node is replaced with its in-order predecessor (largest value in the left sub-tree) or in-order successor (smallest value in the right sub-tree)
- Then, the in-order predecessor or successor is deleted
- Deleting node 56 from the given BST:

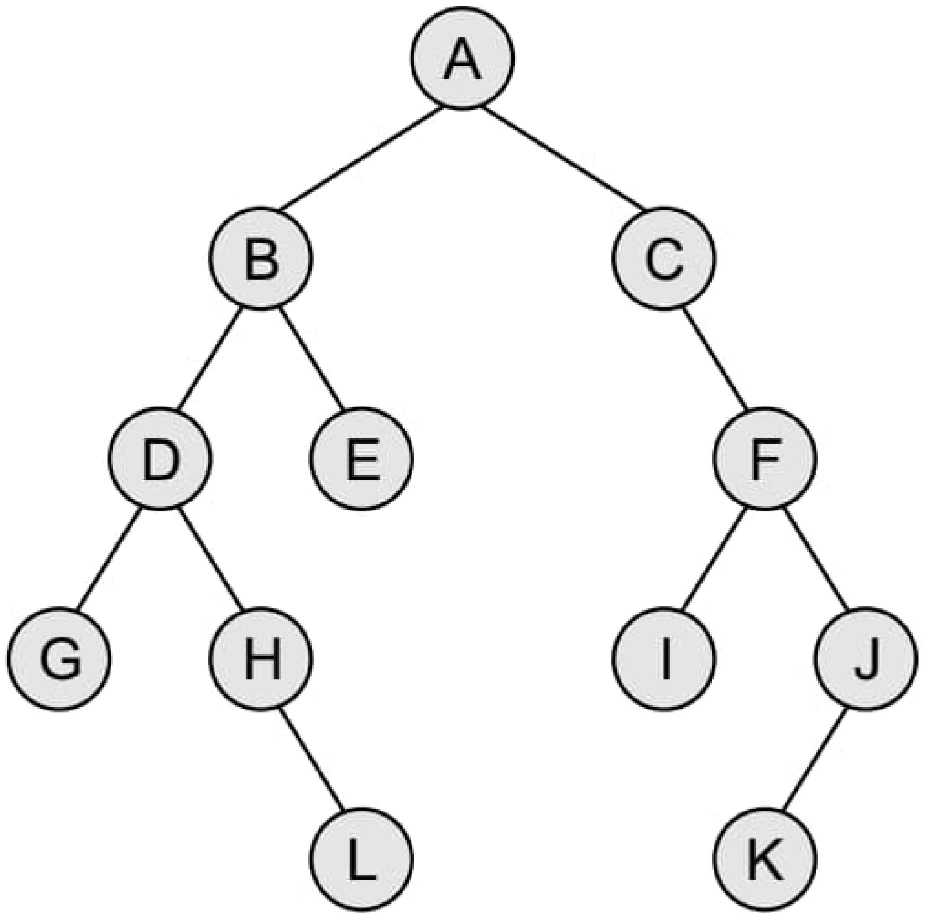


Tree Traversals

- Often, we want to iterate over and process nodes of a tree
- We can do this by walking through the tree in a prescribed order and visiting the nodes as they are encountered
- This process is called tree traversal
- The three most commonly used traversals are:
  - Pre-order: visit root node, traverse left subtree T_L, traverse right subtree T_R
  - In-order: traverse T_L, visit root node, traverse T_R
  - Post-order: traverse T_L, traverse T_R, visit root node
- The difference in the algorithms is whether the root is visited before the children are traversed (pre), in between traversing the left and right children (in), or after the children are traversed (post)

Tree Traversals

- Pre-order traversal is also called depth-first traversal
- In this algorithm, the left sub-tree is always traversed before the right sub-tree
- Pre-order algorithm is also known as the NLR traversal algorithm (Node-Left-Right)
- Recursive algorithm for pre-order traversal:
  – If the tree is empty Return
  – Visit the root
  – Pre-order traverse the left subtree
  – Pre-order traverse the right subtree

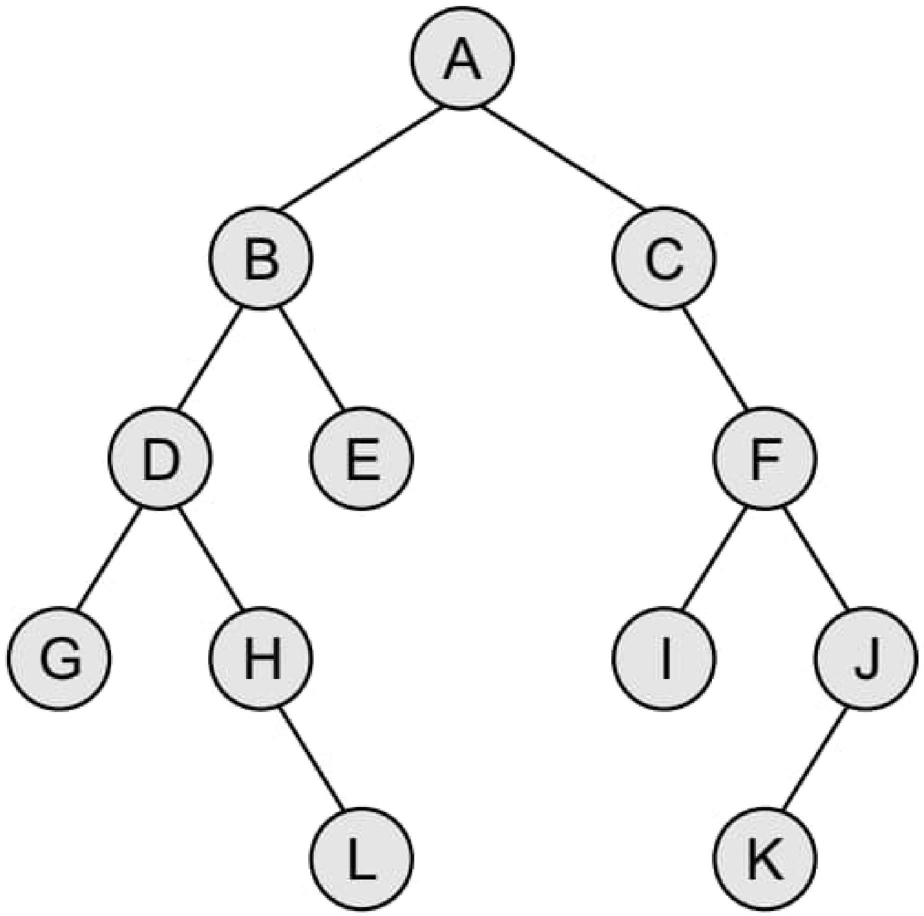


Pre-order traversal:

A, B, D, G, H, L, E, C, F, I, J, and K

Tree Traversals

- In-order traversal is also called symmetric traversal
- In this algorithm, the left subtree is always traversed before the root node and the right subtree
- In-order algorithm is also known as the LNR traversal algorithm (Left-Node-Right)
- Recursive algorithm for in-order traversal:
  - If the tree is empty Return
  - In-order traverse the left subtree
  - Visit the root
  - In-order traverse the right subtree

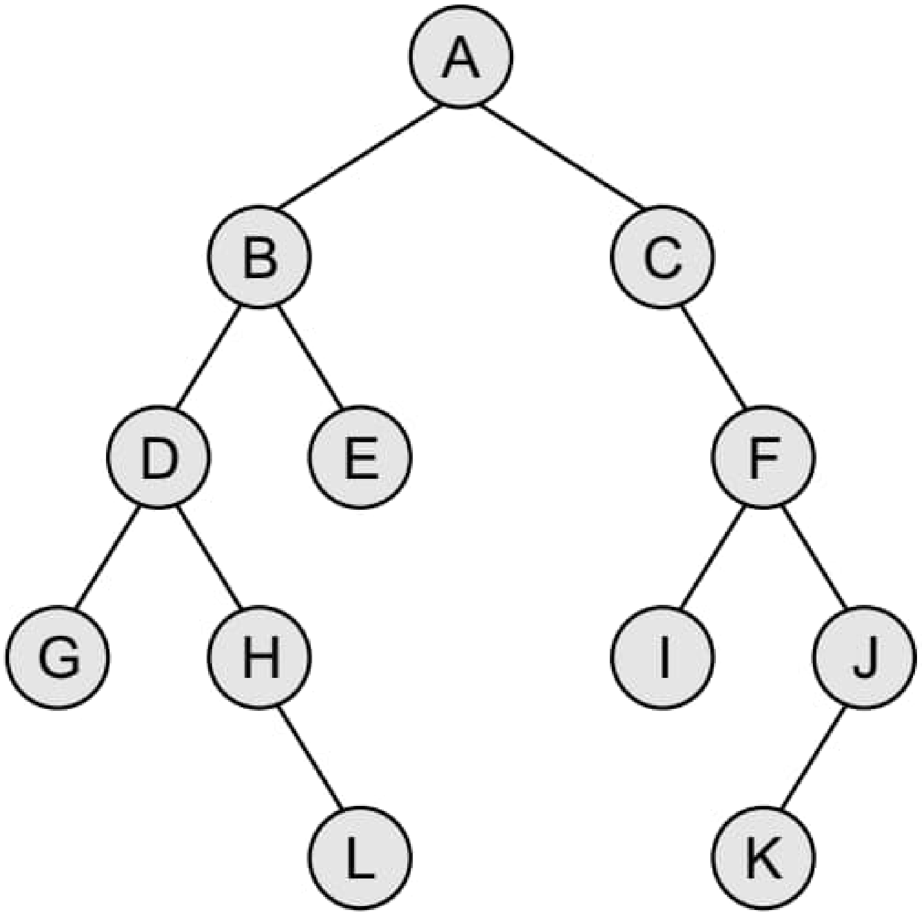


In-order traversal:

G, D, H, L, B, E, A, C, I, F, K, and J

Tree Traversals

- In the post-order traversal algorithm, the left subtree is always traversed before the right subtree and the root node
- Post-order algorithm is also known as the LRN traversal algorithm (Left-Right-Node)
- Recursive algorithm for in-order traversal:
  - If the tree is empty Return
  - Post-order traverse the left subtree
  - Post-order traverse the right subtree
  - Visit the root

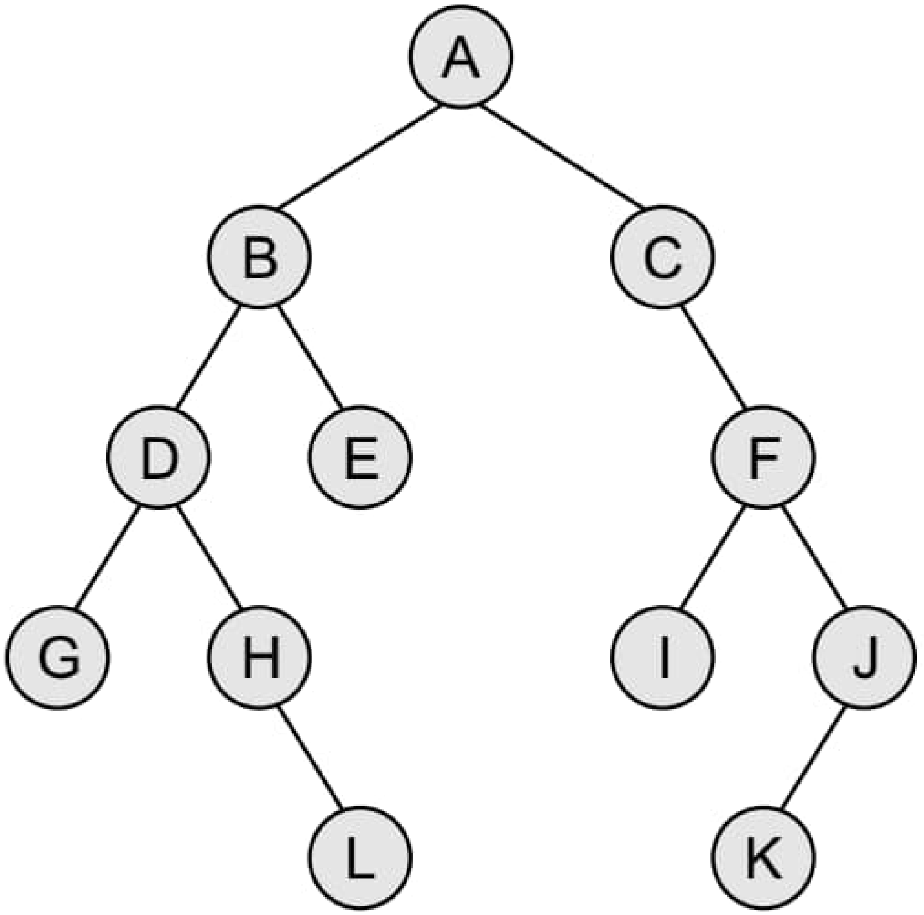


Post-order traversal:

G, L, H, D, E, B, I, K, J, F, C, and A

Tree Traversals

- Additionally, we can define the level-order traversal
- In this algorithm, all the nodes at a level are accessed before going to the next level
- This algorithm is also called the breadth-first traversal algorithm



Level-order traversal:

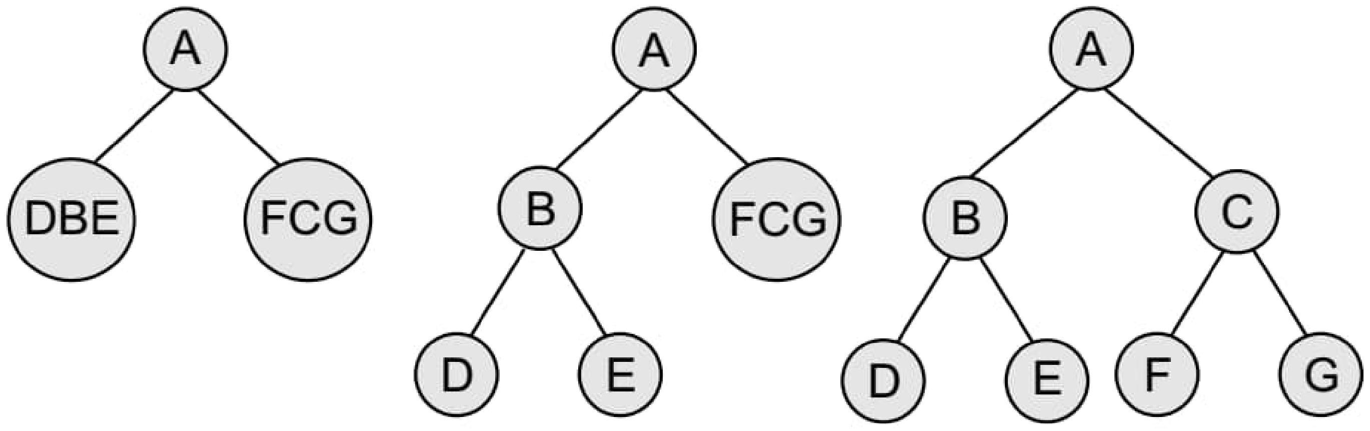
A, B, C, D, E, F, G, H, I, J, L, and K

Tree Traversals

- Having at least two traversal results, we can construct a binary tree
- The first traversal must be the in-order traversal and the second can be either pre-order or post-order traversal
- The in-order traversal result will be used to determine the left and the right child nodes, and the pre-order/post-order can be used to determine the root node

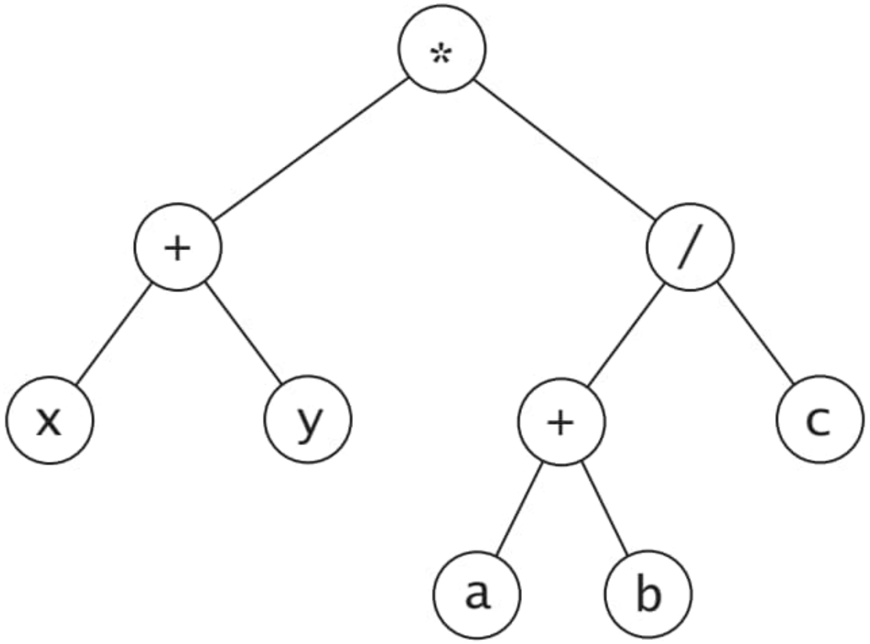
Tree Traversals

- Let’s consider the following traversal results:
  - In–order traversal: D B E A F C G
  - Pre–order traversal: A B D E C F G
- To construct the tree, follow the steps:
  - 1. Use the pre-order sequence to determine the root node of the tree. The first element would be the root node
  - 2. Elements on the left and right side of the root node in the in-order traversal sequence form the left and right sub-tree of the root node, respectively
  - 3. Recursively select each element from pre-order traversal sequence and create its left and right sub-trees from the in-order traversal sequence



Tree Traversals

- Traversals of expression trees give interesting results
- If we perform an in-order traversal of the expression tree, we get the infix expression: x + y * a + b / c
- The post-order traversal of this tree gives the postfix form of the expression: x y + a b + c / *
- The pre-order traversal visits the nodes in the sequence: * + x y / + a b c, which is the prefix form of the expression

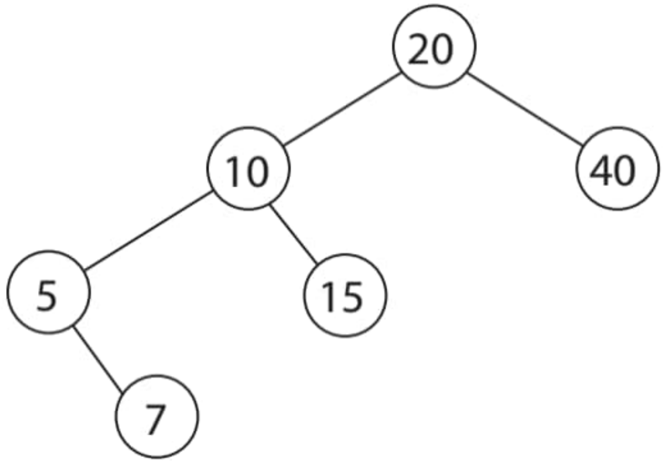


Balancing Binary Trees

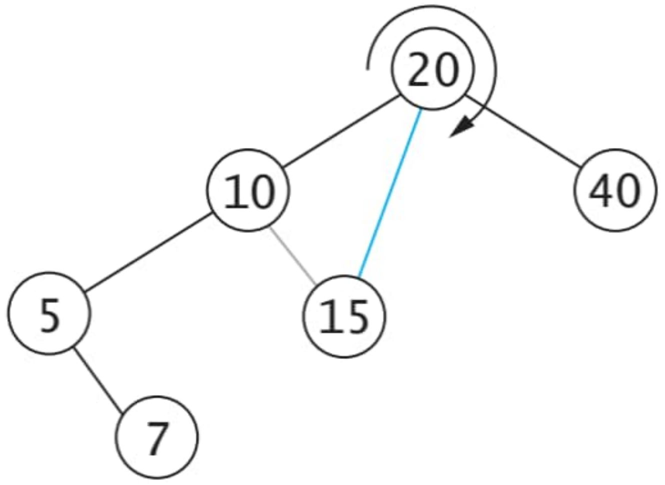
- In extremely unbalanced binary search trees, searches or insertions would be $O(n)$, not $O(\log n)$
- Tree rotation is an operation on a binary tree that changes the structure without interfering with the order of the elements
- A tree rotation moves one node up in the tree and one node down
- It is used to decrease height of a tree by moving smaller subtrees down and larger subtrees up

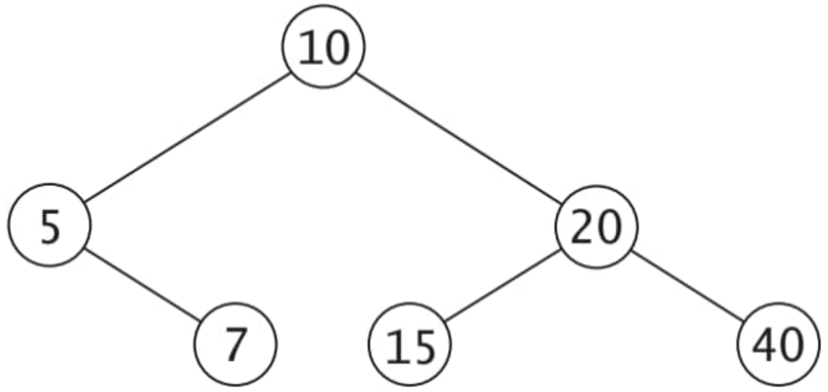
Balancing Binary Trees

- In this tree, the height of the left subtree of the root (20) is 3, and the height of the right subtree is 1:



- Balancing the tree through a right rotation:





Balancing Binary Trees

- Algorithm for rotation right:
  - Remember the value of `root.left` (`temp = root.left`)
  - Set `root.left` to the value of `temp.right`
  - Set `temp.right` to `root`
  - Set `root` to `temp`
- The algorithm for left rotation is symmetric to that for right rotation
- A single rotation modifies a constant number of parent-child relationships, so it can be implemented in *O*(1) time with a linked binary tree representation

Balancing Binary Trees

- Self balancing binary search trees:
  - AVL trees
  - Red-Black trees
  - Splay trees

AVL Trees

- Introduced by G.M. Adelson-Velsky and E.M. Landis in 1962
- AVL tree is also known as a height-balanced tree
- The structure of an AVL tree is the same as that of a binary search tree but every node has a balance factor associated with it
- The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree: $h_R – h_L$
- A binary search tree in which every node has a balance factor of –1, 0, or 1 is said to be height balanced
- A node with any other balance factor is considered to be unbalanced and requires rebalancing of the tree

Red-Black Trees

- Invented in 1972 by Rudolf Bayer
- AVL trees may require many restructure operations (rotations) to be performed after a deletion
- The red-black tree uses *O*(1) structural changes after an update in order to stay balanced
- Formally, a red-black tree is a binary search tree with nodes colored red and black in a way that satisfies the following properties:
  - The root is always black
  - A red node always has black children (None is considered to refer to a black node)
  - The number of black nodes in any path from the root to a leaf is the same

Splay Trees

- A splay tree consists of a binary tree with no additional fields
- In a splay tree, operations such as insertion, search, and deletion are combined with one basic operation called splaying
- Splaying the tree for a particular node rearranges the tree to place that node at the root
- A splay operation causes more frequently accessed elements to remain nearer to the root, thereby reducing the typical search times
- Splaying allows us to guarantee a logarithmic amortized running time for insertions, deletions, and searches

Binary Heaps

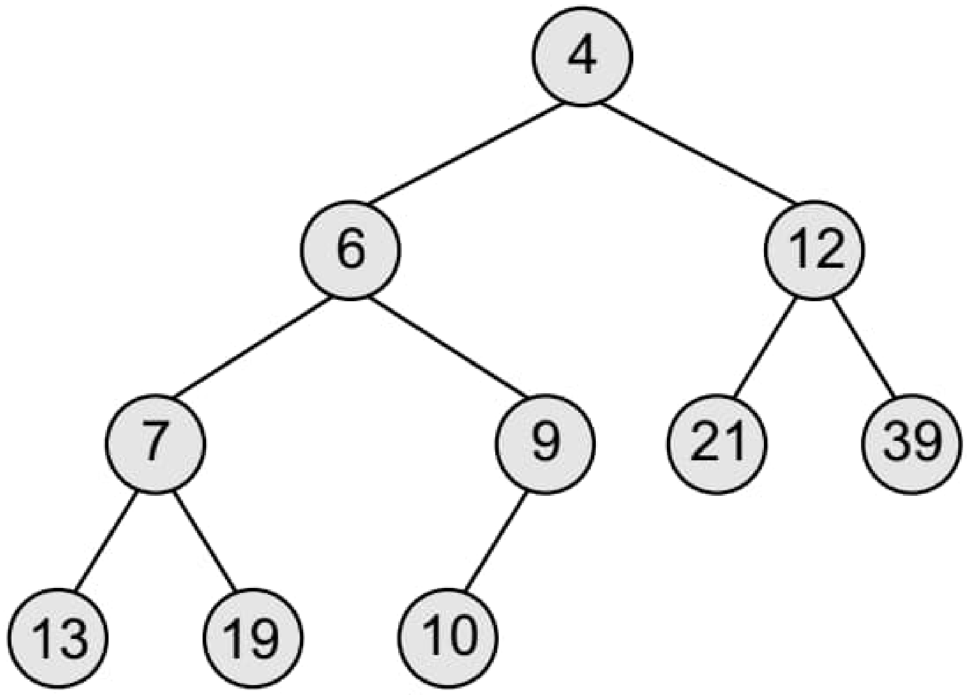
- In an unsorted list, insertions can be performed in *O*(1) time, but finding or removing an element with the minimum key requires an *O*(*n*)-time loop through the entire collection
- In contrast, if using a sorted list, the minimum element can be trivially found or removed in *O*(1) time, but adding a new element to the list may require *O*(*n*) time to restore the sorted order
- A binary heap data structure allows us to perform both insertions and removals in logarithmic time

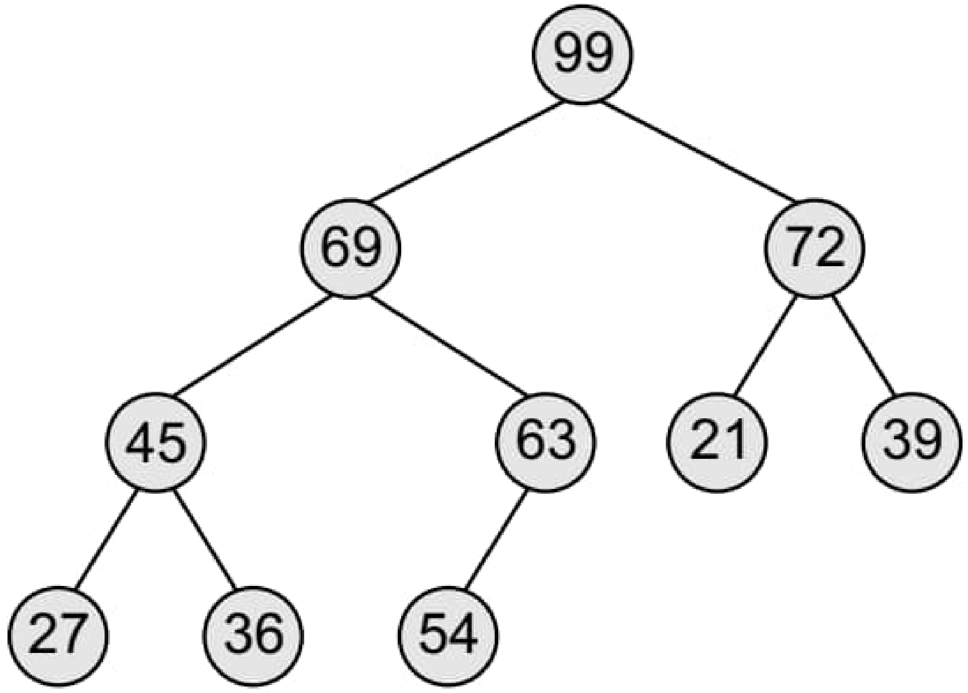
Binary Heaps

- A heap is a complete binary tree with the following properties:
  - The value in the root is greater (max heap) / smaller (min heap) than or equal to all items in the tree
  - Every subtree is a heap
- Complete binary tree: a binary tree that satisfies two properties:
  - every level, except possibly the last, is completely filled
  - all nodes appear as far left as possible
- As a consequence, a heap H storing n entries has height h = floor(log n)

Binary Heaps

- Min and max heaps:



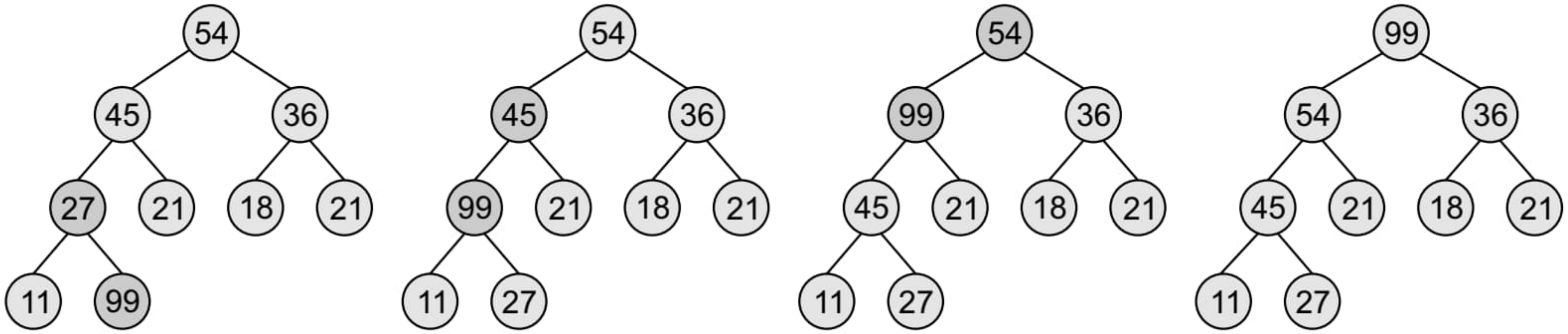


Binary Heaps

- Inserting a new value into the heap is done in the following steps:
  - 1. Insert the new item in the next position at the bottom of the heap *H* in such a way that *H* is still a complete binary tree but not necessarily a heap
  - 2. `while` new item is not at the root and new item is larger than its parent:
    - Swap the new item with its parent, moving the new item up the heap
- New items are added to the last row of a heap
- If a new item is smaller than or equal to its parent, nothing more need be done
- However, if the new item is larger than its parent, the new item and its parent are swapped

Binary Heaps

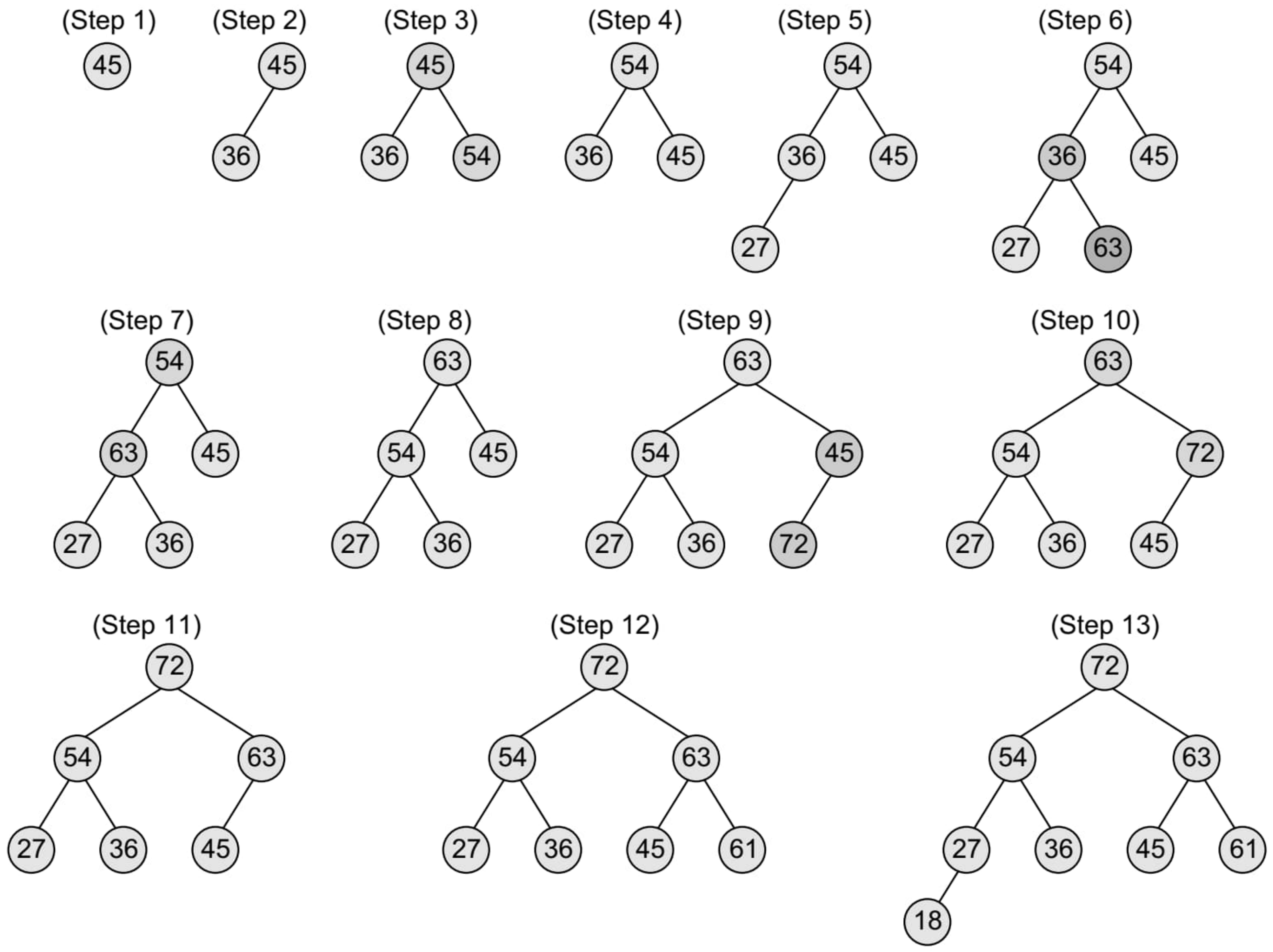
- Insertion of node 99 in the max heap:



- In the worst case, the number of swaps performed during the insertion process is equal to the height of *H*

Binary Heaps

- Building a max heap *H* from the set of numbers: 45, 36, 54, 27, 63, 72, 61, and 18:

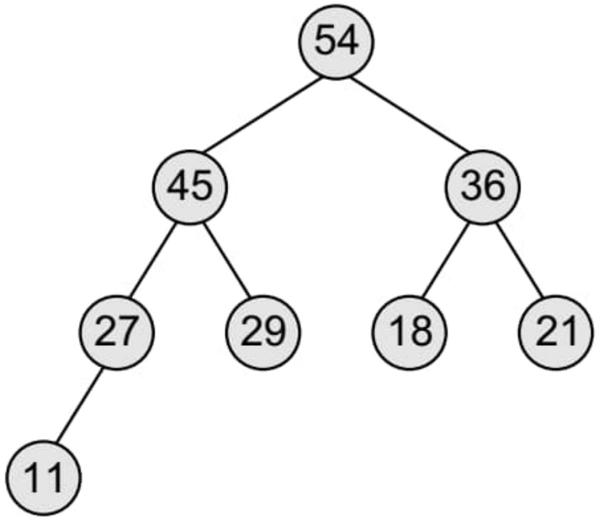


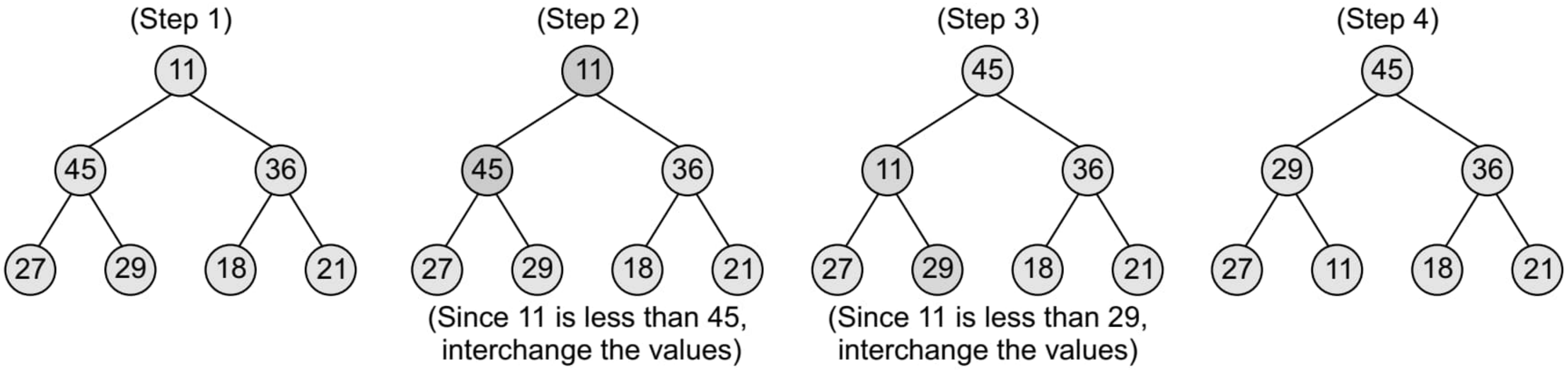
Binary Heaps

- Removal from a heap is always from the top
- The top item is first replaced with the last item in the heap (at the lower right-hand position), so that the heap remains a complete tree
- Then the new item at the top is moved down the heap until it is in its proper position
- Deleting an element from the heap is done in the following three steps:
  - 1. Remove the item in the root node by replacing it with the last item in the heap (LIH)
  - 2. while item LIH has children and item LIH is smaller than either of its children:
    - 3. Swap item LIH with its larger child, moving LIH down the heap

Binary Heaps

- Removal the root-node value from the max heap:





(Since 11 is less than 45, interchange the values)

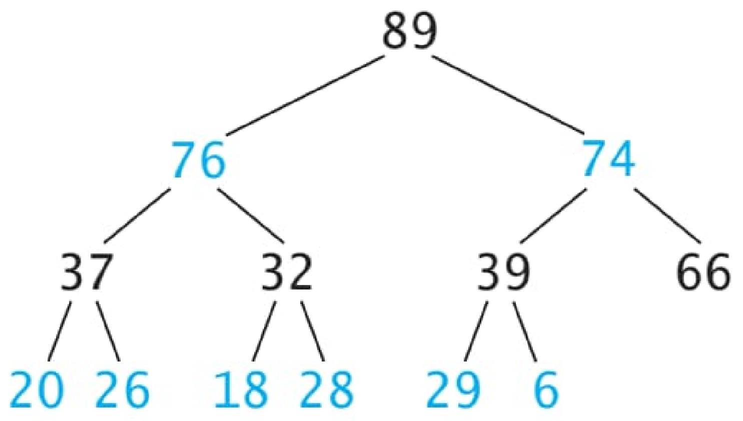
(Since 11 is less than 29, interchange the values)

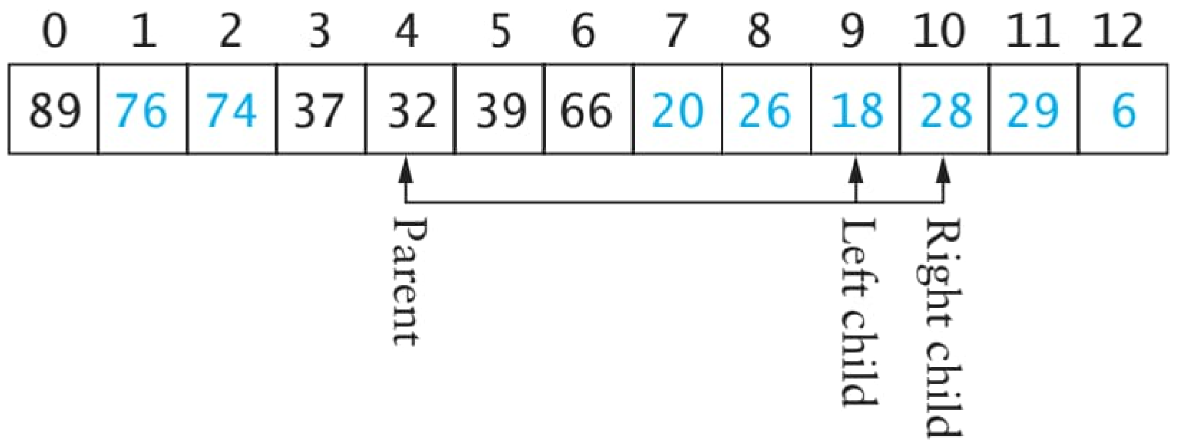
Binary Heaps

- Because a heap is a complete binary tree, it can be efficiently implemented using an array instead of a linked data structure
- The first element (index 0) stores the root data
- The next two elements (indices 1 and 2) store the two children of the root
- In general, for a node at position *p*, the left child is at 2*p* + 1 and the right child is at 2*p* + 2
- A node at position *c* can find its parent at (*c* – 1) // 2

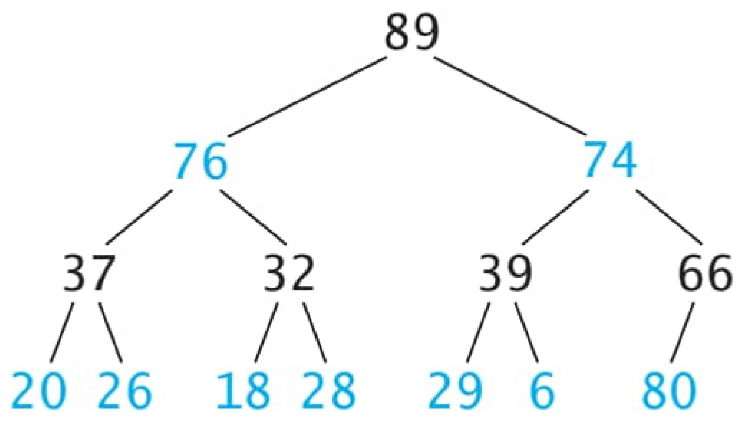
Binary Heaps

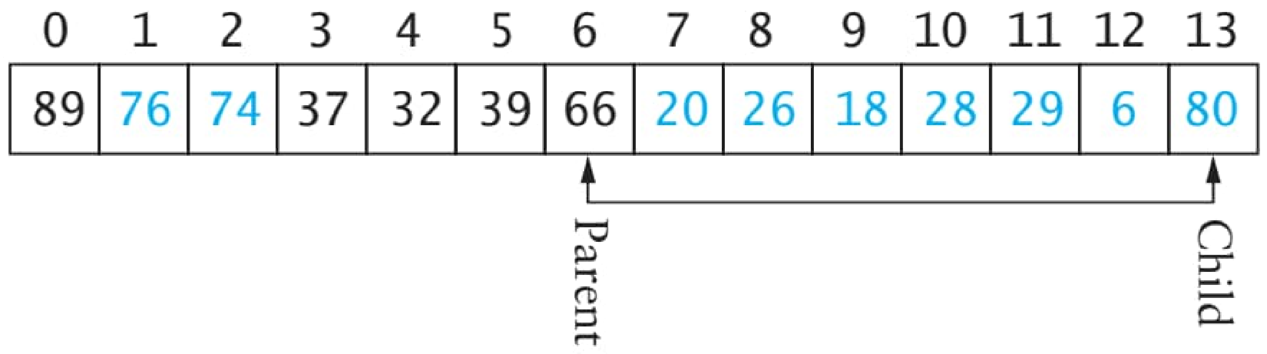
- Internal representation of the heap:





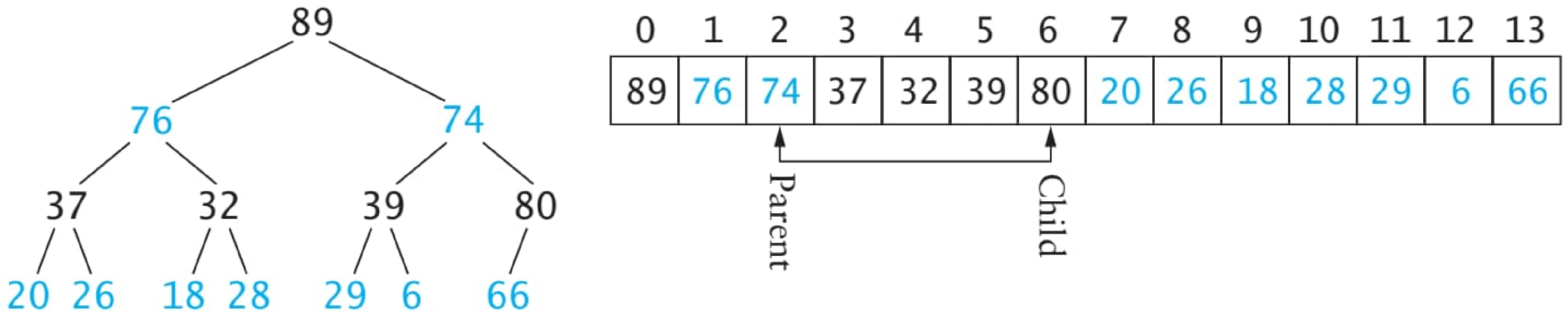
- Internal representation of the heap after insertion:



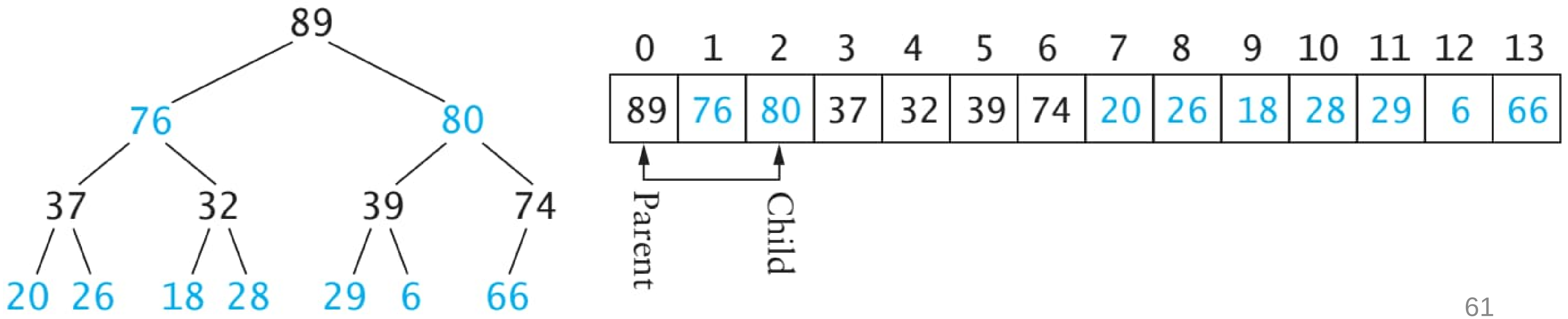


Binary Heaps

- Internal representation of the heap after the first swap:

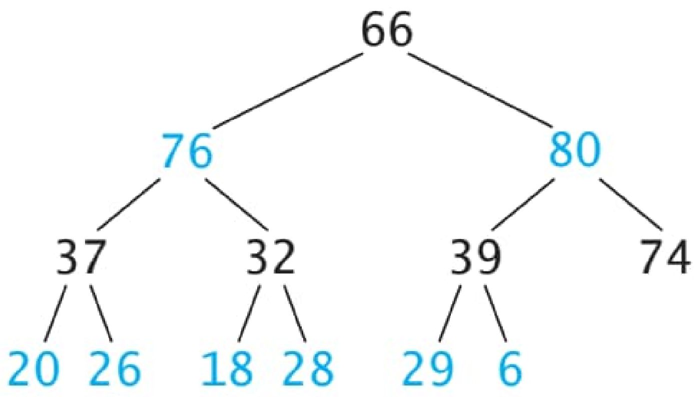


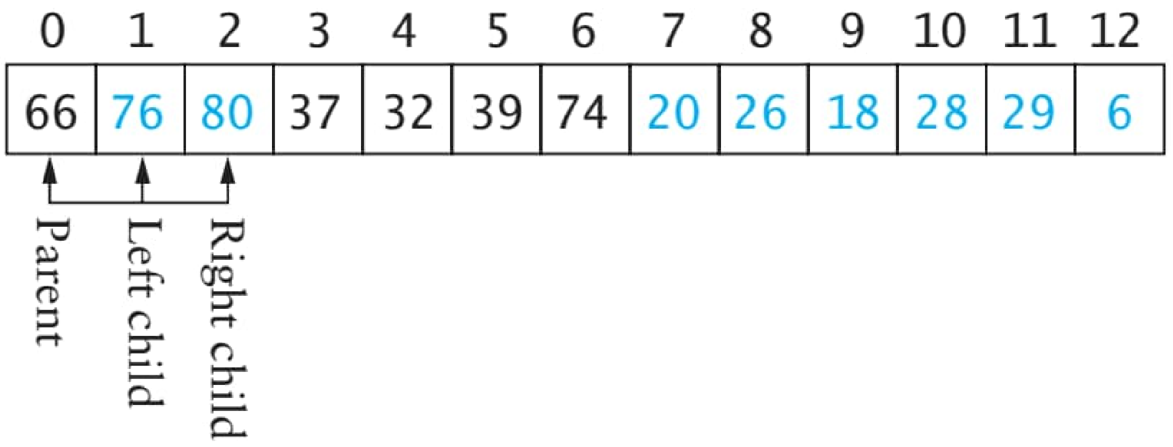
- Internal representation of the heap after the second swap:



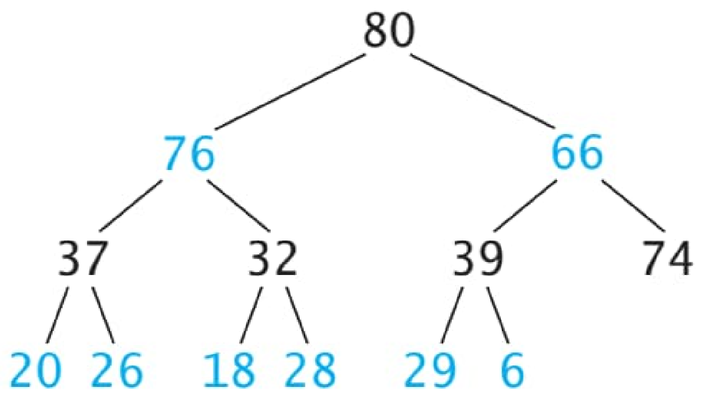
Binary Heaps

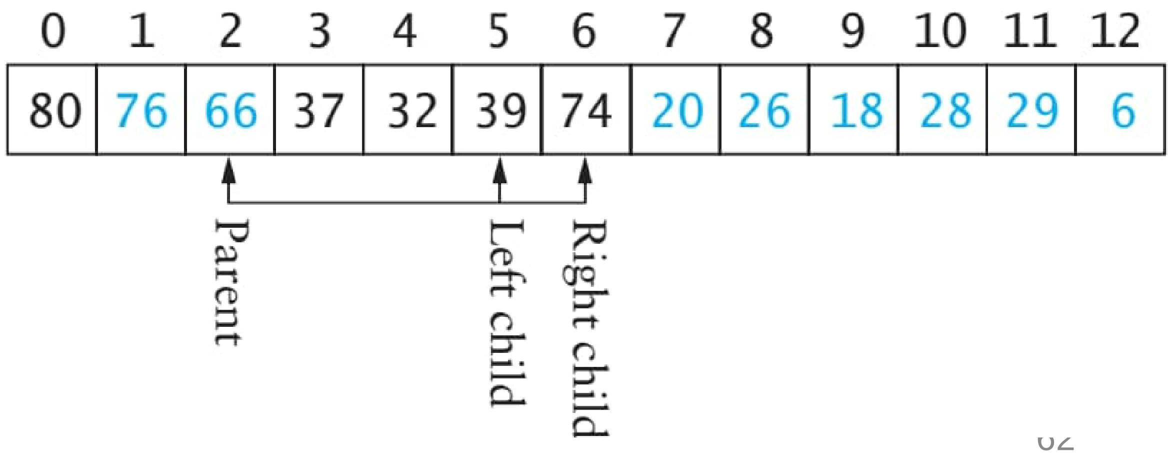
- Internal representation of the heap after 89 is removed:





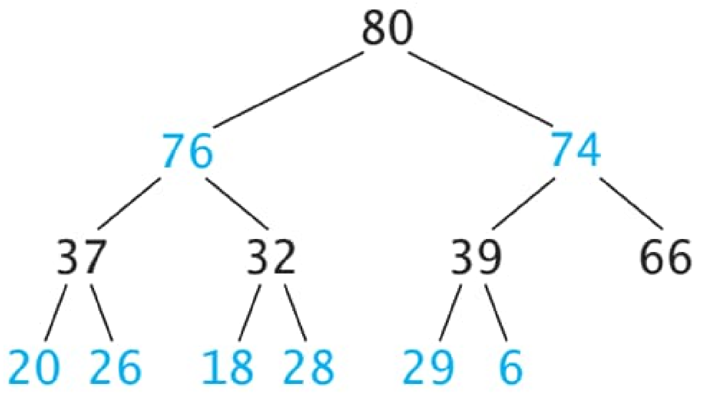
- Internal representation of the heap after 80 and 66 are swapped:

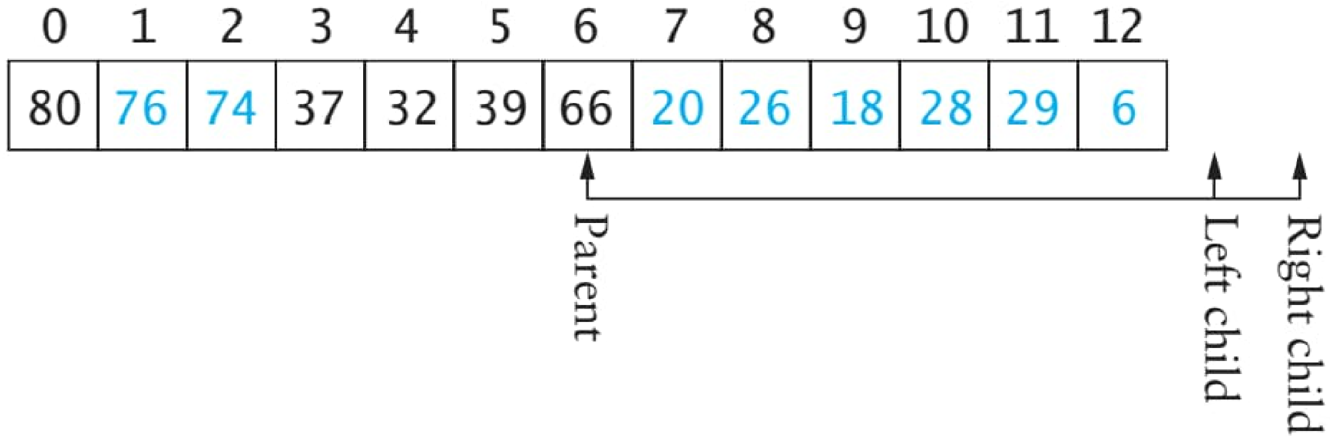




Binary Heaps

- Internal representation of the heap after 66 and 74 are swapped:





0	1	2	3	4	5	6	7	8	9	10	11	12
80	76	74	37	32	39	66	20	26	18	28	29	6

Binary Heaps

- The remove algorithm traces a path from the root to a leaf, and the insert algorithm traces a path from a leaf to the root
- This requires at most *h* steps, where *h* is the height of the tree
- The largest heap of height *h* is a full tree of height *h* with $2^h - 1$ nodes
- The smallest heap is a complete tree of height *h*, consisting of a full tree of height *h* – 1, with a single node as the left child of the leftmost child at height *h*. Thus, this tree has $2^{(h-1)}$ nodes
- Therefore, both insert and remove are *O*(log *n*), where *n* is the number of items in the heap

Priority Queues

- There are many applications in which a queue-like structure is used to manage objects, but for which the first-in, first-out (FIFO) policy does not suffice
- For example, an air-traffic control center decides which flight to clear for landing on the basis of factors such as each plane’s distance from the runway, time spent waiting in a holding pattern, or amount of remaining fuel
- In computer systems, some tasks are more important than others and should be executed first despite of later arrival
- A priority queue is a collection of prioritized elements that allows arbitrary element insertion, and allows the removal of the element that has first priority

Priority Queues

- We model a queue element and its priority as a key-value pair
- When an element is added to a priority queue, the user designates its priority by providing an associated key
- The element with the minimum key will be the next to be removed from the queue
- We define the priority queue ADT to support the following methods for a priority queue `P`:
  - `P.add(k, v)`: Insert an item with key k and value v
  - `P.min()`: Return a tuple, `(k, v)`, representing the key and value of an item in `P` with minimum key without removing the item; return `None` if the priority queue is empty
  - `P.remove_min()`: Remove an item with minimum key from `P`, and return a tuple, `(k,v)`, representing the key and value of the removed item; return `None` if the priority queue is empty
  - `P.is_empty()`: Return `True` if `P` does not contain any items
  - `len(P)`: Return the number of items in `P`

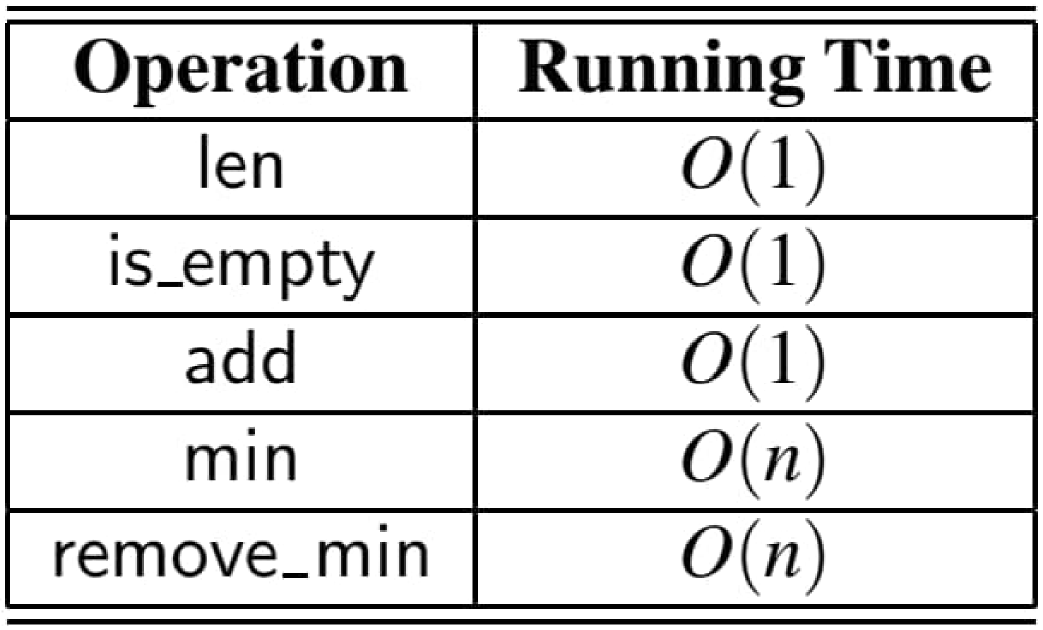
Priority Queues

- A series of operations and their effects on an initially empty priority queue P:

Operation	Return Value	Priority Queue
P.add(5,A)		{(5,A)}
P.add(9,C)		{(5,A), (9,C)}
P.add(3,B)		{(3,B), (5,A), (9,C)}
P.add(7,D)		{(3,B), (5,A), (7,D), (9,C)}
P.min( )	(3,B)	{(3,B), (5,A), (7,D), (9,C)}
P.remove_min( )	(3,B)	{(5,A), (7,D), (9,C)}
P.remove_min( )	(5,A)	{(7,D), (9,C)}
len(P)	2	{(7,D), (9,C)}
P.remove_min( )	(7,D)	{(9,C)}
P.remove_min( )	(9,C)	{ }
P.is_empty( )	True	{ }
P.remove_min( )	“error”	{ }

Priority Queues

- Priority queue can be implemented with an unsorted doubly linked list
- Due to the loop for finding the minimum key, both `min` and `remove_min` methods run in *O*(*n*) time, where *n* is the number of entries in the priority queue
- Worst-case running times of the methods of a priority queue of size *n*, realized by means of an unsorted, doubly linked list:



Operation	Running Time
len	$O(1)$
is_empty	$O(1)$
add	$O(1)$
min	$O(n)$
remove_min	$O(n)$

Priority Queues

- Alternatively, a priority queue can be implemented with a list of entries sorted by nondecreasing keys
- This ensures that the first element of the list is an entry with the smallest key
- Assuming that the list is implemented with a doubly linked list, operations `min` and `remove_min` take *O*(1) time
- However, the `add` method requires that we scan the list to find the appropriate position to insert the new item
- Therefore, the `add` method takes *O*(*n*) worst-case time, where *n* is the number of entries in the priority queue

Priority Queues

- Worst-case running times of the methods of a priority queue of size *n*, realized by means of an unsorted or sorted list. The space requirement is *O*(*n*)

Operation	**Unsorted List**	**Sorted List**
len	$O(1)$	$O(1)$
is_empty	$O(1)$	$O(1)$
add	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
remove_min	$O(n)$	$O(1)$

Priority Queues

- The item with the smallest key is always removed first from a priority queue, just as it is for a min heap
- Because insertion into and removal from a heap is *O*(*log n*), a heap can be the basis for an efficient implementation of a priority queue
- In a heap-based priority queue, `add` and `remove_min` operations take *O*(*log n*) time, and all remaining operations *O*(1) time
- The fundamental way the heap achieves this improvement is to use the structure of a binary tree to find a compromise between elements being entirely unsorted and perfectly sorted