



Trees

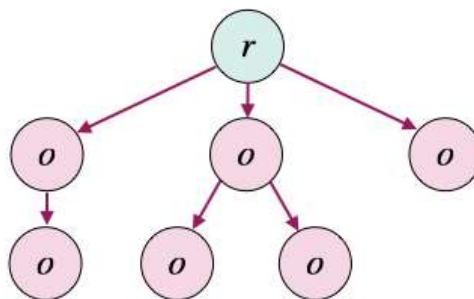
By *Afshine Amidi* and *Shervine Amidi*

General concepts

Definition

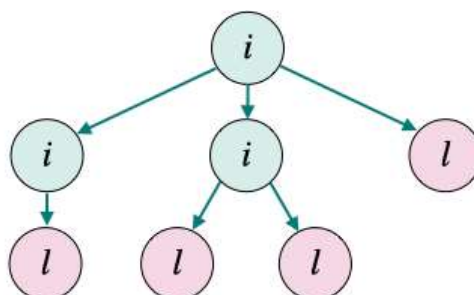
A tree is a DAG with the following properties:

- *Incoming edge*:
 - There is exactly one node that has no incoming edge, and that node is called the root.
 - Each of the other nodes has exactly one incoming edge.
- *Outgoing edge*: A node cannot have an outgoing edge that points to itself.

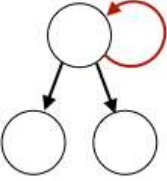
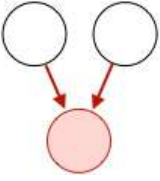
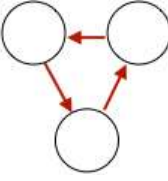
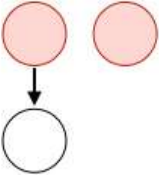


There are two types of nodes in a tree:

- *Internal node*: Node that has one or more children.
- *Leaf*: Node that does not have any children.



Here are some examples of graphs that are not trees, along with the associated reason:

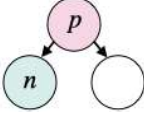
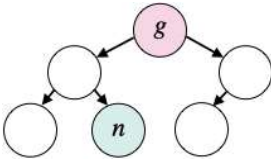
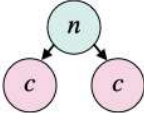
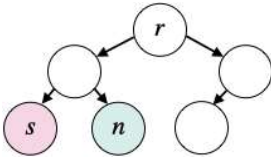
Self-loop	Multiple parents	Cycle	Multiple roots
			

REMARK

By construction, a tree with N nodes has $N - 1$ edges.

Notations

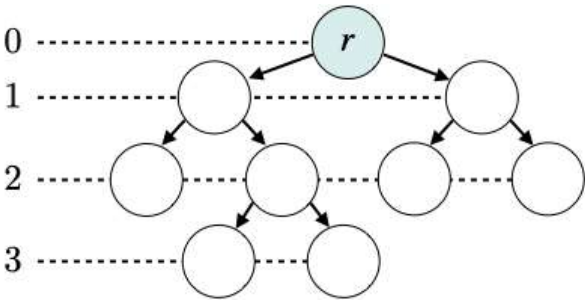
Names given to nodes follow a family-like vocabulary. The most common ones are presented in the table below:

Concept	Description	Illustration
Parent	<p>The parent p of a node n is the predecessor of that node.</p> <p>n has at most one parent.</p>	
Grandparent	<p>The grandparent g of a node n is the predecessor of the predecessor of that node.</p> <p>n has at most one grandparent.</p>	
Child	<p>A child c of a node n is a successor of that node.</p> <p>n can have 0, 1 or multiple children.</p>	
Sibling	<p>A sibling s of a node n is a different node with the same parent as n.</p> <p>n can have 0, 1 or multiple siblings.</p>	

Concept	Description	Illustration
Uncle	<p>An uncle u of a node n is a child of n's grandparent that is not its parent.</p> <p>n can have 0, 1 or multiple uncles.</p>	

Depth

The depth of a given node n , noted $\text{depth}(n)$, is the number of edges from the root r of the tree to node n .

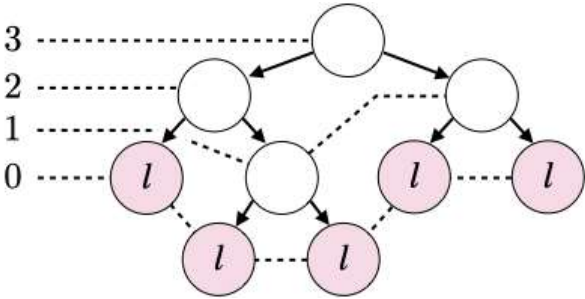


REMARK

The root node has a depth of 0.

Height

The height of a given node n , noted $\text{height}(n)$, is the number of edges of the longest path from node n to the deepest leaf.



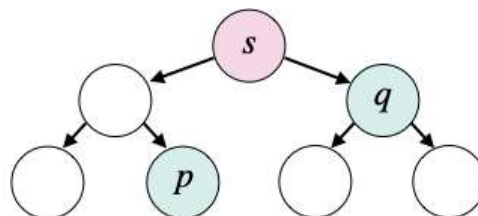
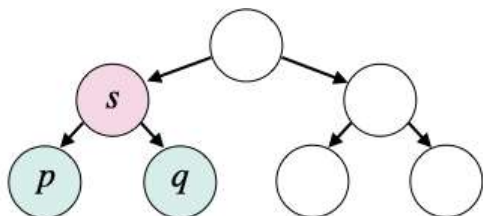
The height of a tree is the height of its root node.

REMARK

Leaf nodes have a height of 0.

Lowest Common Ancestor

In a given tree, the lowest common ancestor of two nodes p and q , noted $\text{LCA}(p, q)$, is defined as being the deepest node that has both p and q as descendants. The examples below use $s \triangleq \text{LCA}(p, q)$.



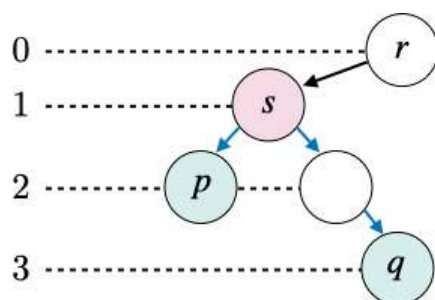
! REMARK

For the purposes of this definition, we may consider a node to be a descendant of itself.

Node distance

The distance $d(p, q)$ between two nodes p and q is the minimum number of edges between these two nodes. If we note $s \triangleq \text{LCA}(p, q)$, we have the following formula:

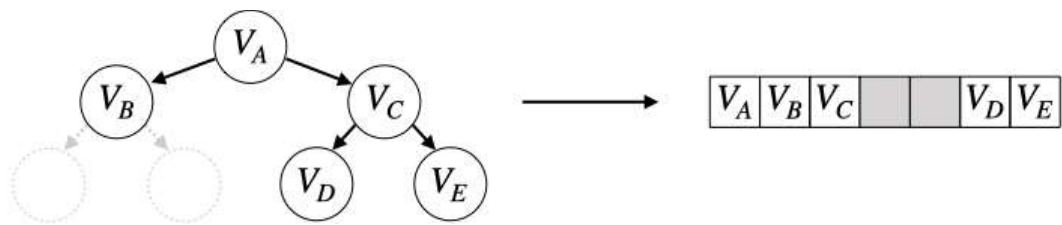
$$d(p, q) = \text{depth}(p) + \text{depth}(q) - 2 \times \text{depth}(s)$$



For instance, nodes p and q have a distance of 3 in the tree above.

Serialization

Tree serialization is a method to encode a tree into a generic format (e.g. string) without losing information.



REMARK

There can be more than one way to serialize a tree. The figure above illustrates a level-by-level approach in the case of binary trees.

Binary trees

Definition

A binary tree is a tree where each node has at most 2 children. The table below sums up possible properties of binary trees:

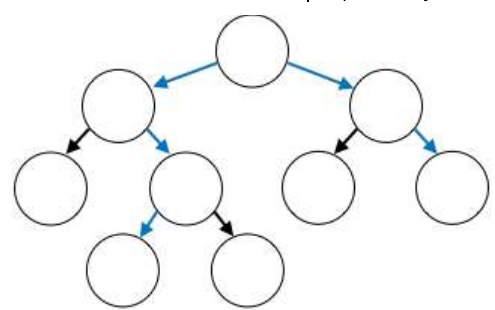
Type	Description	Illustration
Full	Every node has either 0 or 2 children.	
Complete	All levels are completely filled, except possibly the last one where all nodes are to the left.	
Perfect	All internal nodes have 2 children and all leaves are at the same level.	

REMARK

A perfect tree has exactly $2^{h+1} - 1$ nodes with h the height of the tree.

Diameter

The diameter of a binary tree is the longest distance between any of its two nodes.



For example, the binary tree above has a diameter of 5.

Main tree traversals

The table below summarizes the 3 main ways to recursively traverse a binary tree:

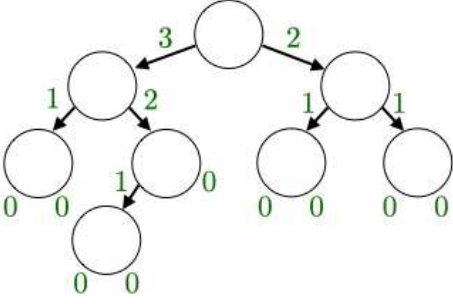
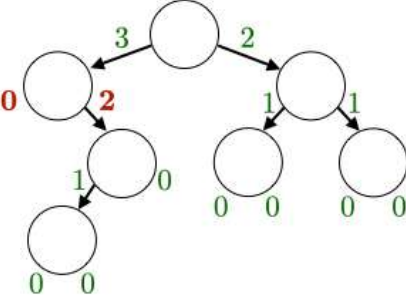
Traversal	Algorithm	Illustration	Example
Pre-order	<div>1. Visit the root</div> <div>2. Visit the left subtree</div> <div>3. Visit the right subtree</div>		
In-order	<div>1. Visit the left subtree</div> <div>2. Visit the root</div> <div>3. Visit the right subtree</div>		
Post-order	<div>1. Visit the left subtree</div> <div>2. Visit the right subtree</div> <div>3. Visit the root</div>		

REMARK

Pre-order, in-order and post-order traversals are all variations of DFS.

Balanced tree

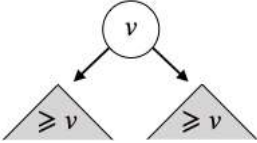
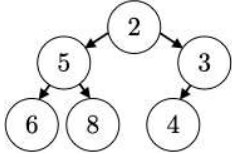
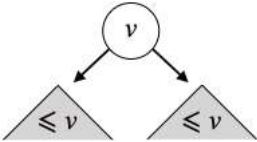
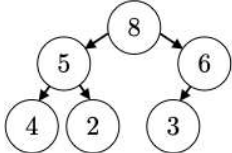
A binary tree is said to be balanced if the difference in height of each node restricted to its left and right subtrees is at most 1.

Balanced	Not balanced
	

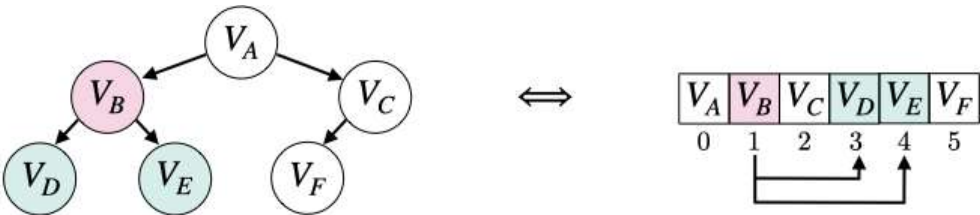
Heaps

Definition

A heap is a complete binary tree with an additional property that makes it either a min-heap or a max-heap:

Type	Description	Illustration	Example
Min-heap	The value of each node is lower than its children's, if any. By definition, the root has the lowest value.		
Max-heap	The value of each node is higher than its children's, if any. By definition, the root has the highest value.		

Since a heap is a complete binary tree, it is convenient to represent it with an array of size n , where n is the number of nodes.



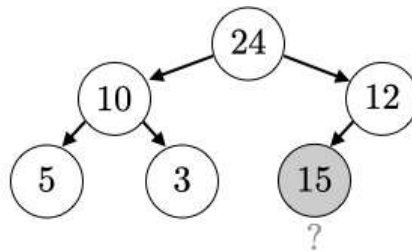
For a given node of index $i \in \llbracket 0, n - 1 \rrbracket$,

- its parent has an index of $\left\lfloor \frac{i-1}{2} \right\rfloor$
- its left child has an index of $2i + 1$ and its right child has an index of $2i + 2$

The following parts will use max-heaps for consistency. However, one could also use min-heaps to obtain the same results.

Heapify up

Let's suppose that the last child is potentially not fulfilling the properties of the max-heap. The heapify up operation, also called *bubble up*, aims at finding the correct place for this node.



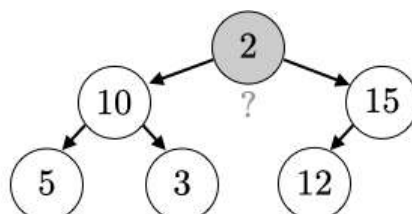
This operation is done in $\mathcal{O}(\log(n))$ time:

- *Update step:* While the node's parent has a lower value than the node's, swap the two.
- *Final step:* We now have a valid max-heap.



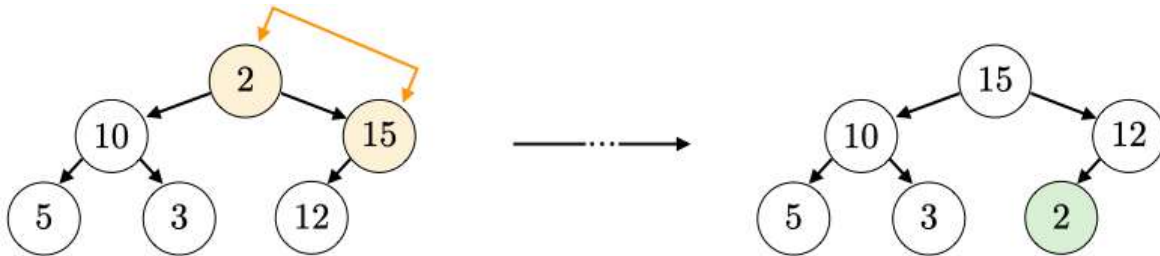
Heapify down

Let's suppose that the root is potentially not fulfilling the properties of the max-heap. The heapify down operation, also called **bubble down*, aims at finding the correct place for this node.



This operation is done in $\mathcal{O}(\log(n))$ time:

- *Update step:* While the highest-value child of the node has a higher value than the node's, swap the two.
- *Final step:* We now have a valid max-heap.



Operations

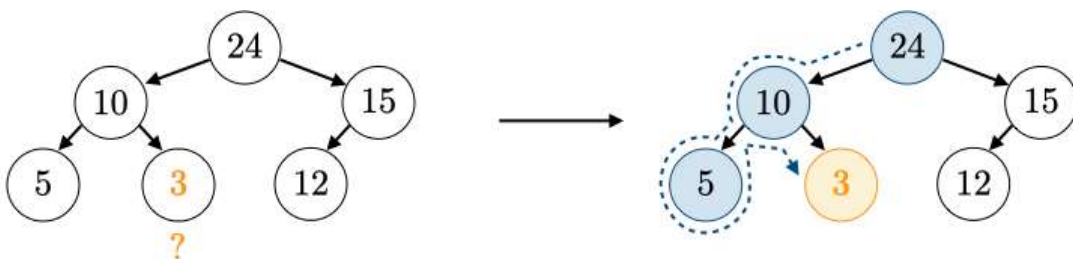
The main operations that can be performed on a max-heap are explained below:

Search We distinguish two cases:

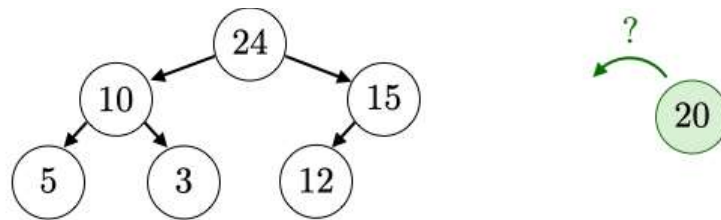
- *Maximum value:* Look at the value corresponding to the root of the heap. It takes $\mathcal{O}(1)$ time.



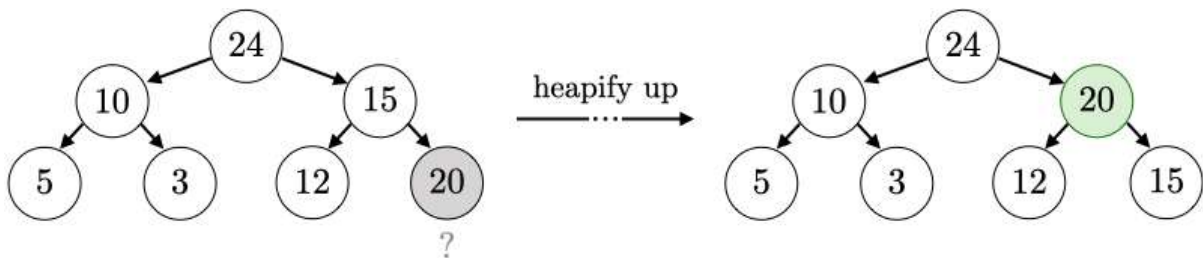
- *Any other value:* Traverse the tree, given that we have no information as to where each node is. It takes $\mathcal{O}(n)$ time.



Insertion It takes $\mathcal{O}(\log(n))$ time.

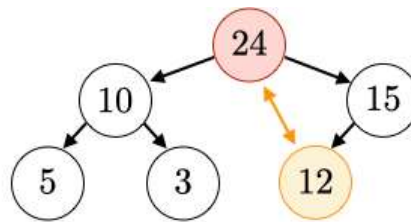


- *Placeholder step*: Add new node as the last child.
- *Heapify up step*: Heapify up the child to its final position.

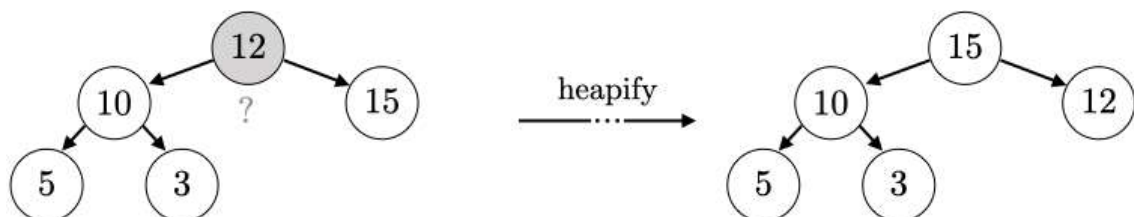


Deletion It takes $\mathcal{O}(\log(n))$ time.

- *Swap step*: Swap node with last child and remove new last child.

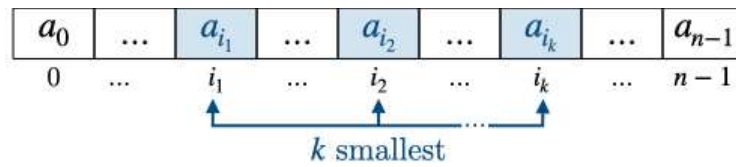


- *Heapify step*: Move the newly-placed node to its final position depending on the situation:
 - *Node's value is higher than parent's*: Heapify up to its final position.
 - *Node's value is lower than highest of its children*: Heapify down to its final position.
 - *Node's value is lower than parent's and higher than highest of its children*: There is nothing to do.



k smallest elements

Given array $A = [a_0, \dots, a_{n-1}]$, the goal is to find the k smallest elements, with $k \leq n$.



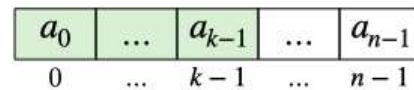
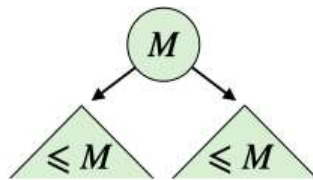
Trick Use a max-heap of size k .

Algorithm The solution is found in $\mathcal{O}(n \log(k))$ time and $\mathcal{O}(k)$ space:

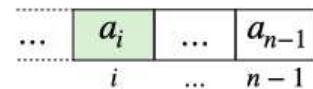
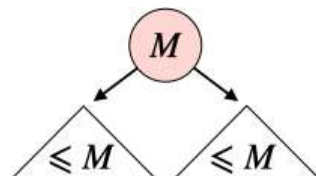
- *Initialization:*

- Set an empty max-heap that will keep track of the k smallest elements.
- Add the first k elements into the max-heap.

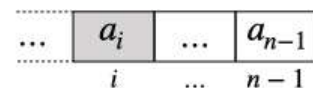
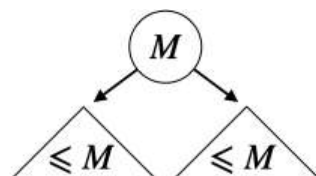
At any given point, we note M the value of the root of the max-heap, i.e. its maximum value.



- *Update step:* We need to see whether any of the remaining elements $a_{i \in [k, n-1]}$ is potentially part of the k smallest elements:
 - If $a_i < M$, pop the max-heap and insert a_i in $\mathcal{O}(\log(k))$ time.



- If $a_i \geq M$, it means that the element is greater than any of the current k smallest elements. We do not do anything. The check is done in $\mathcal{O}(1)$ time.



- *Final step:* The max-heap contains the k smallest elements.

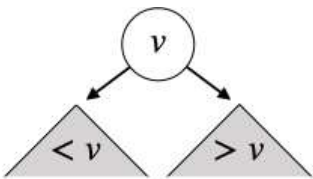
REMARK

Binary search trees

Definition

A binary search tree (BST) is a binary tree where each node has the following properties:

- Its value is greater than any node values in its left subtree
- Its value is less than any node values in its right subtree



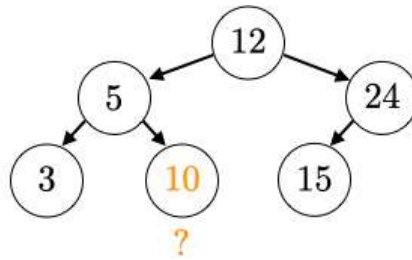
We note h the height of the BST, where h can go anywhere between:

	Best case	Worst case
Description	Tree is balanced	Every node has at most 1 child
Height	$h \approx \log(n)$	$h = n$
Illustration	<pre>graph TD; 12((12)) --> 5((5)); 12 --> 24((24)); 5 --> 3((3)); 5 --> 10((10)); 24 --> 15((15));</pre>	<pre>graph TD; 3((3)) --> 5((5)); 5 --> 10((10)); 10 --> 12((12)); 12 --> 15((15)); 15 --> 24((24));</pre>

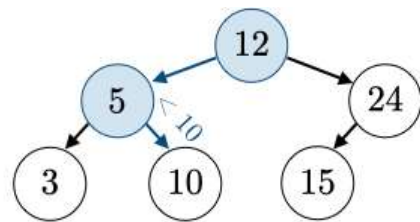
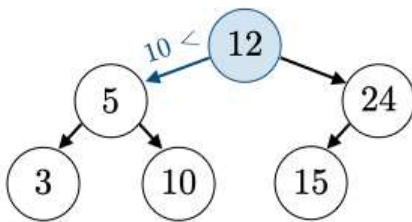
Operations

The main operations that can be performed on a BST each have a time complexity of $\mathcal{O}(h)$ and are explained below:

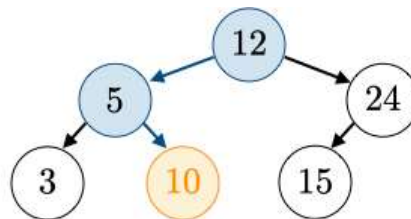
Search Starting from the root, compare the node value with the value v we want to search for.



- *Node value different than v :*
 - If it is higher than v , go to its left child.
 - If it is lower than v , go to its right child.

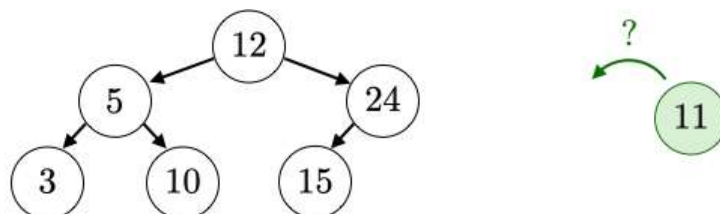


- *Node value equal to v :* We found the target node.



Continue this process recursively until either finding the target node or hitting the end of the tree, in which case v is not present in the BST.

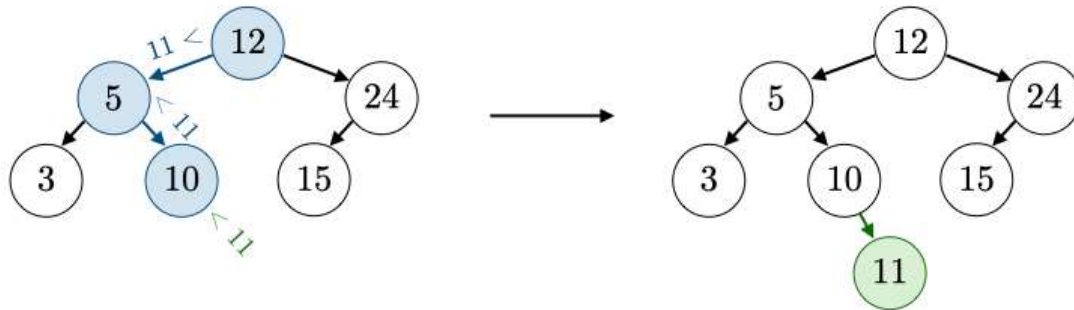
Insertion Let's suppose we want to insert a node of value v in the BST.



In order to do that, we check if there is already a node with the same value in $\mathcal{O}(h)$ time. At the end of the search, there are two possible situations:

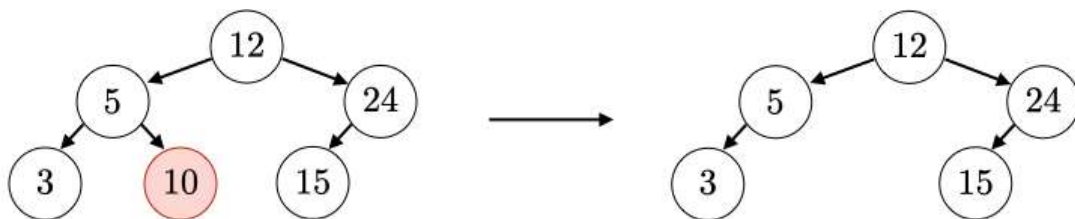
- *Node is found:* There is nothing else to do, since the element that we want to insert is already in the tree.

- *Node is not found*: By construction, the last node seen during the search is a leaf.
 - If v is higher than the value of the last node, add it as its right child.
 - If v is lower than the value of the last node, add it as its left child.

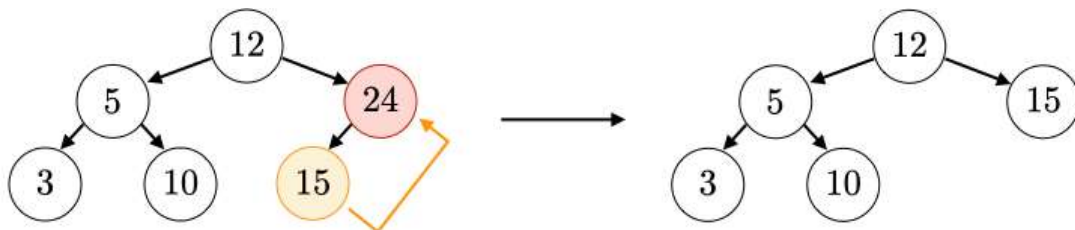


Deletion The process depends on the number of children of the node to be deleted. We have the following situations:

- *0 child*: Delete the node.



- *1 child*: Replace the node with its child.



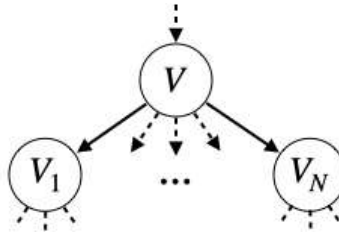
- *2 children*: Replace the node with the max of its left subtree, which is also equal to the in-order predecessor.



N-ary trees

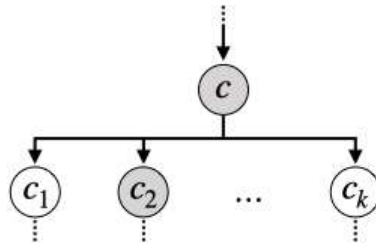
Definition

An N -ary tree is a tree where each node has at most N children.



Trie

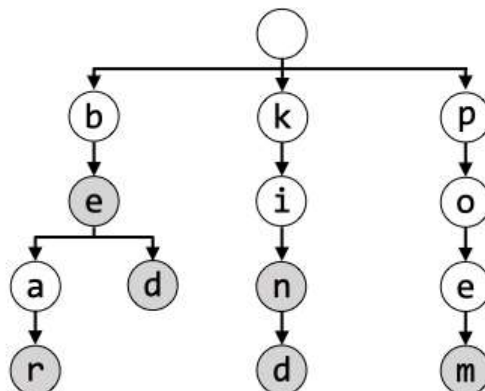
A trie, also called prefix tree, is an N -ary tree that allows for efficient storing and fast retrieval of words. The path from the root node to another given node forms a word that is deduced by concatenating the characters along that path.



Each node n is defined by the following quantities:

- A character c .
- A hash table $h = \{(c_1, n_1), \dots, (c_k, n_k)\}$ gathering the children of that node, where c_i is the child's character and n_i its associated node.
- A boolean that indicates whether the word formed by the path leading to that node is in the dictionary ● or not ○.

By convention, the root is a node for which c is an empty string.

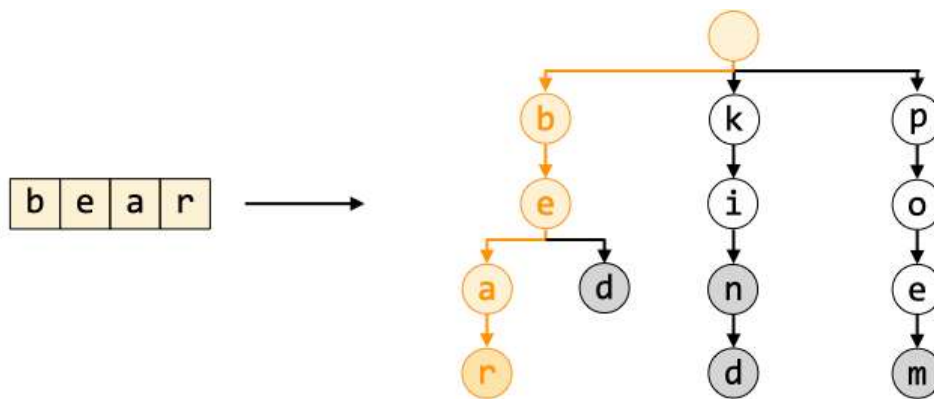


Note that even though all characters of a word are present in the correct order, it does not necessarily mean that the word itself is present in the trie. In order to ensure that the word is indeed in the trie, an important additional condition is for the boolean of the last character to be set to ●.

The operations that can be done with a trie are described below:

Search We would like to know whether a word w is in the trie.

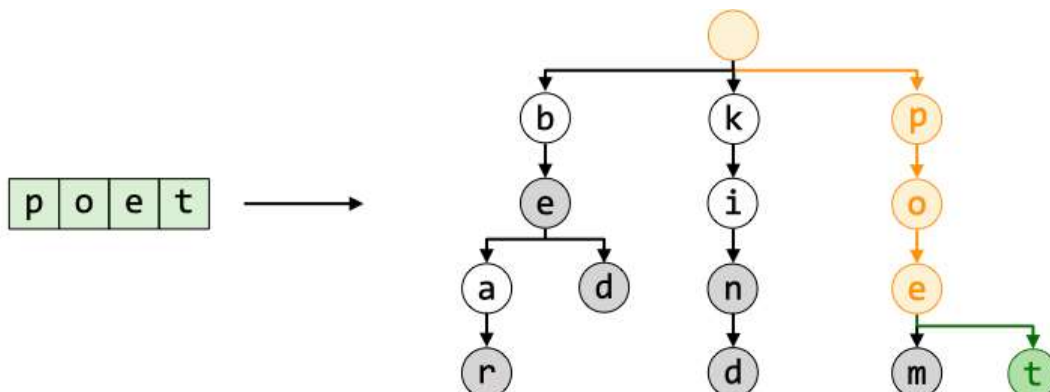
- *Character step*: Starting from the root, traverse the trie by checking one character of w at a time. If any character is missing, it means that the word is not present.



- *Word step*: Check whether the boolean of the node corresponding to the last character is set to ●. If it is not, it means that the word is not present.

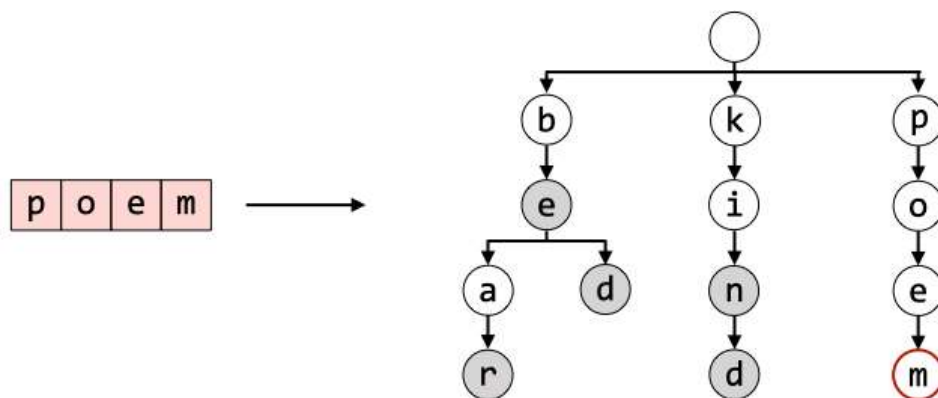
Insertion We would like to insert a word w in the trie.

- *Character step*: Starting from the root, traverse the trie one character of w at a time. Starting from the first missing character (if applicable), add corresponding nodes until completing the word.
- *Word step*: Set the boolean of the last character to ●.



Deletion We would like to remove a word w from the trie.

- *Character step*: Traverse the trie one character of w at a time.
- *Word step*: Set the boolean of the last character to \bigcirc .



REMARK

Use cases of a trie include searching for a word starting with a given prefix.

WANT MORE CONTENT LIKE THIS?

[Subscribe here](#) to be notified of new Super Study Guide releases!