Planar Graph (7A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice and Octave.

Planar Graph

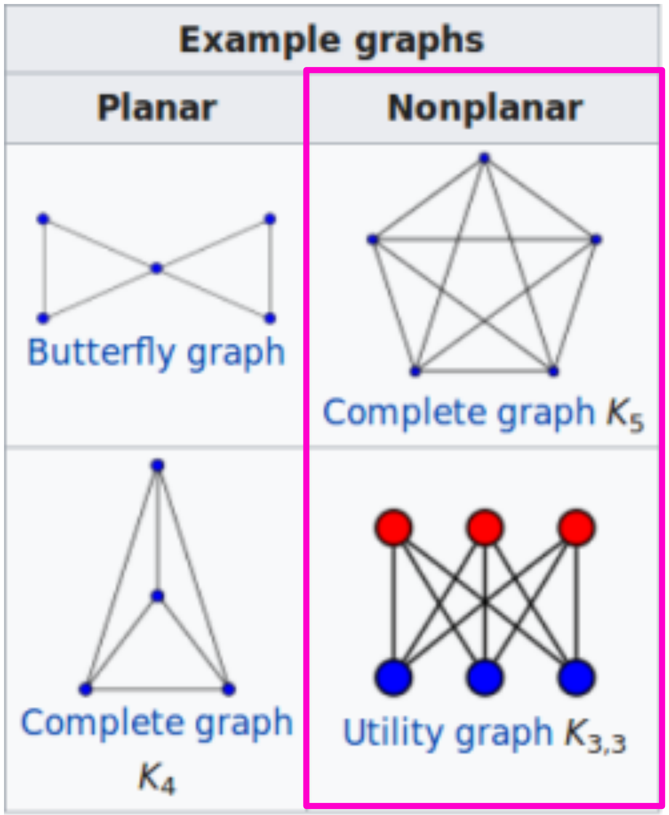
a planar graph is a graph that can be embedded in the plane, i.e., it can be drawn on the plane in such a way that its edges intersect only at their endpoints.

it can be drawn in such a way that no edges cross each other. Such a drawing is called a **plane graph** or **planar embedding** of the graph. (**planar representation**)

A **plane graph** can be defined as a planar graph with a mapping from every node to a point on a plane, and from every edge to a plane curve on that plane,
such that the extreme points of each curve are the points mapped from its end nodes, and all curves are disjoint except on their extreme points.

https://en.wikipedia.org/wiki/Planar_graph

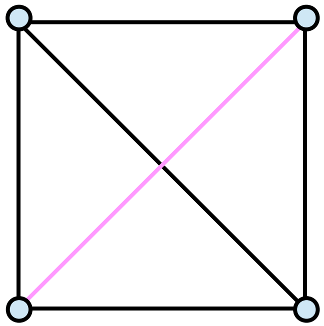
Planar Graph Examples

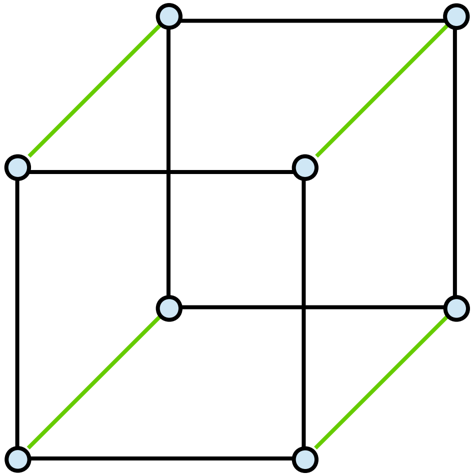


https://en.wikipedia.org/wiki/Planar_graph

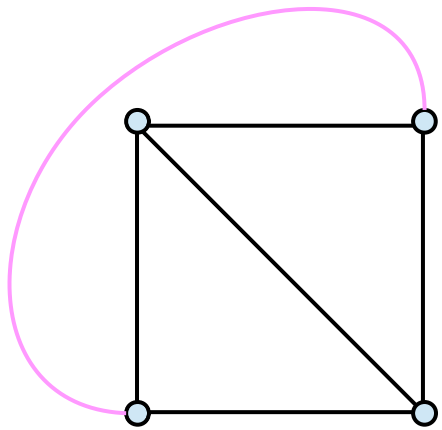
Planar Representation

K_4



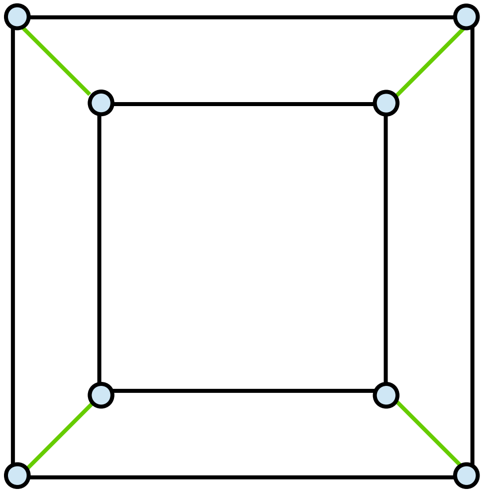


Q_3



No crossing

K_4 Planar

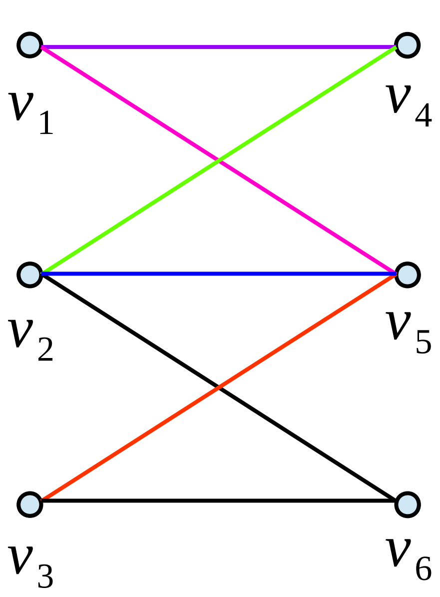


No crossing

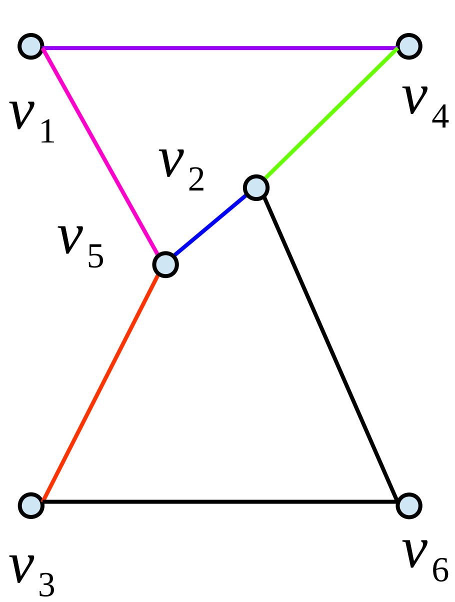
Q_3 Planar

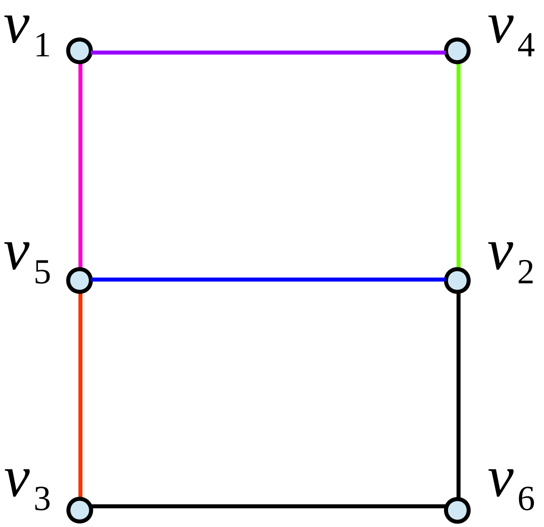
Discrete Mathematics, Rosen

A planar bipartite graph

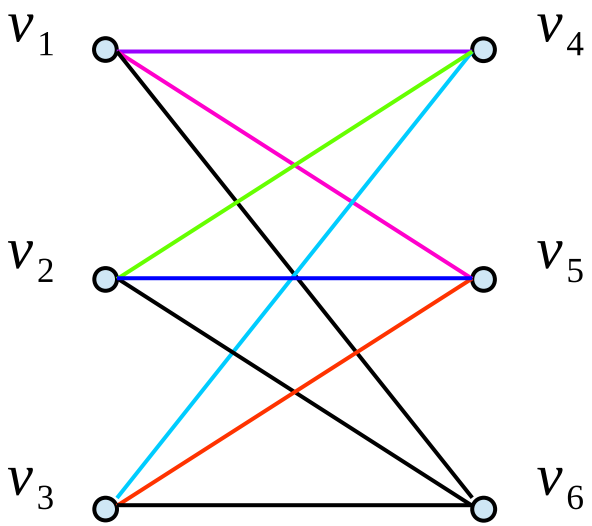


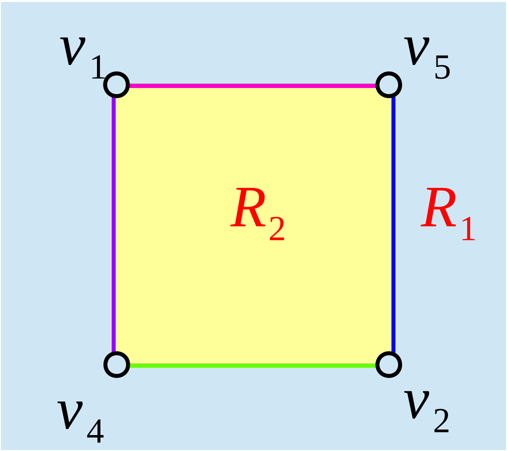
Bipartite graph
but <u>not</u> complete
bipartite graph
$K_{3,3}$



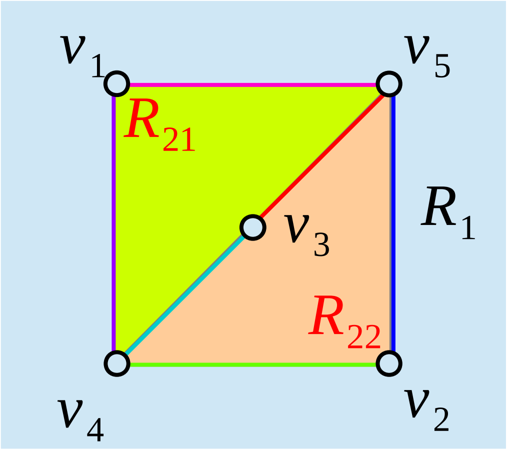


Planar Graph





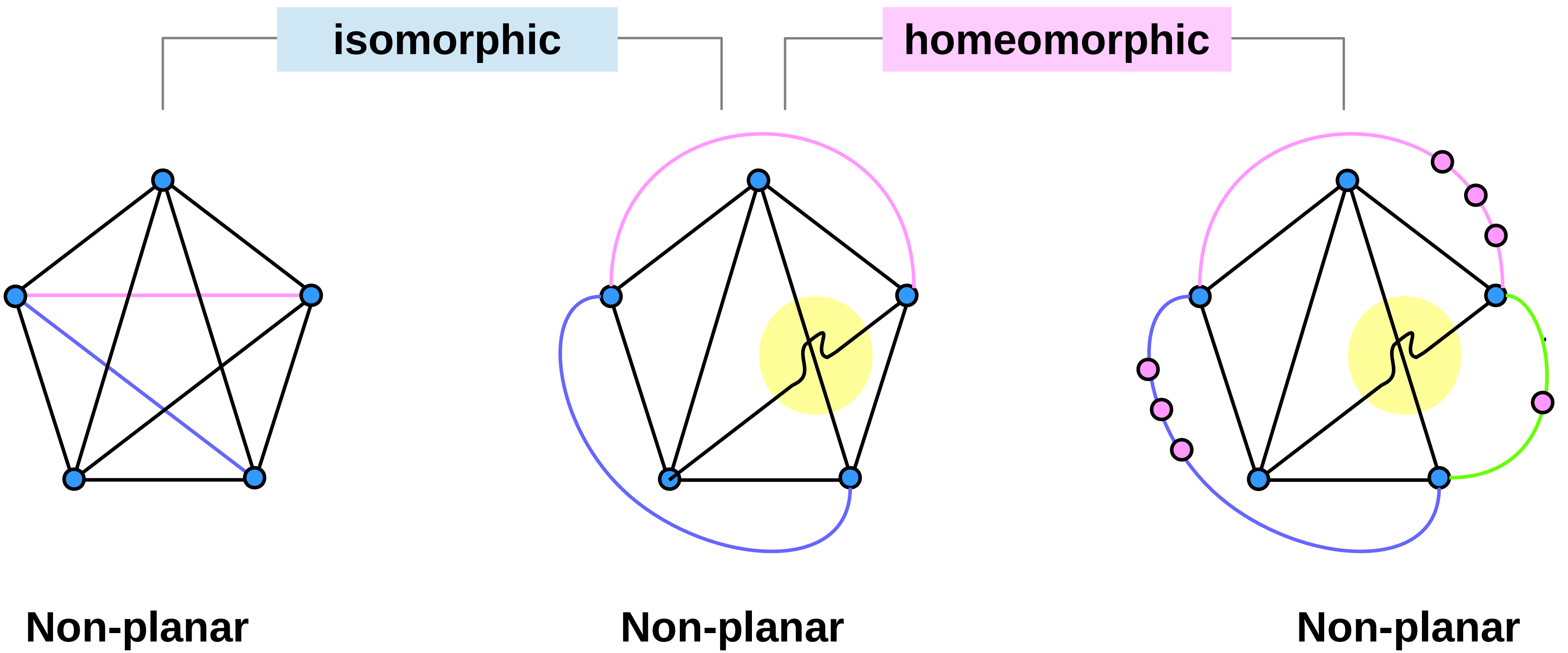
no where v_6



Non-planar

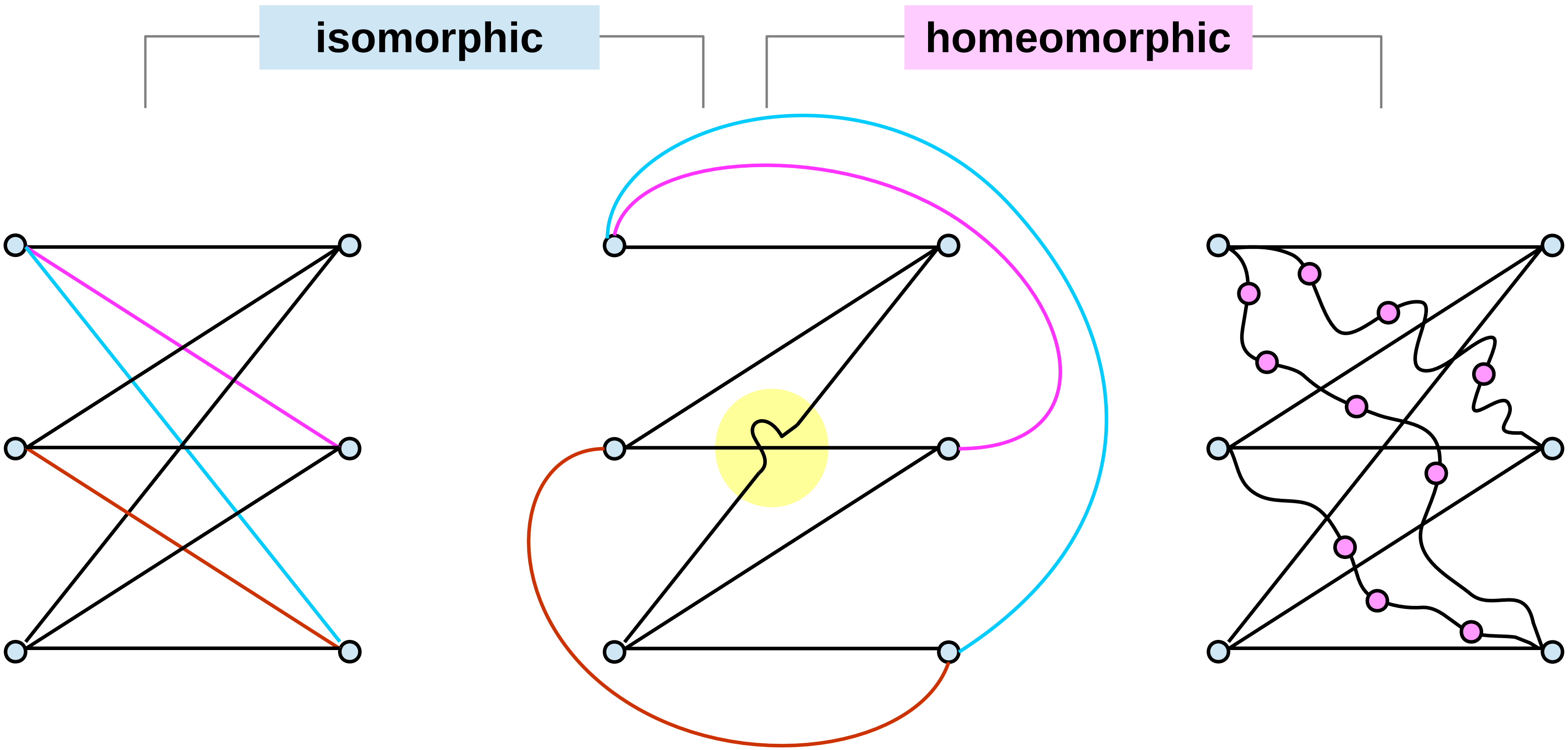
Discrete Mathematics, Rosen

Non-planar graph examples – K_5



All these graphs are similar
in determining whether
they are planar or not

Non-planar graph examples – $K_{3,3}$

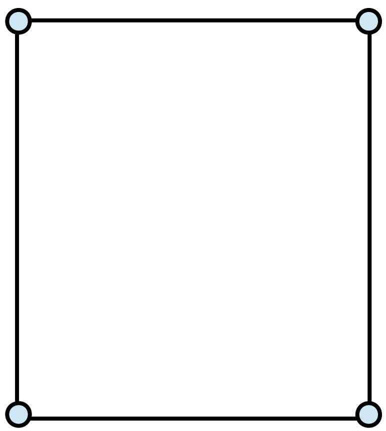


Non-planar Non-planar Non-planar

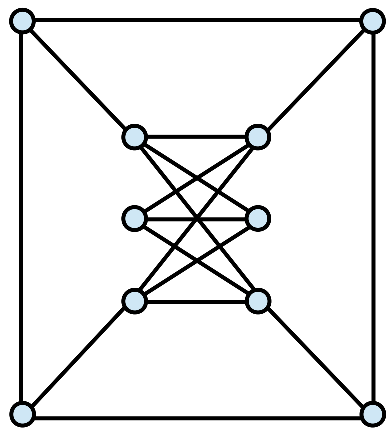
All these graphs are <u>similar</u> in determining whether they are planar or not

Non-planar graph examples – embedding $K_{3,3}$

Planar



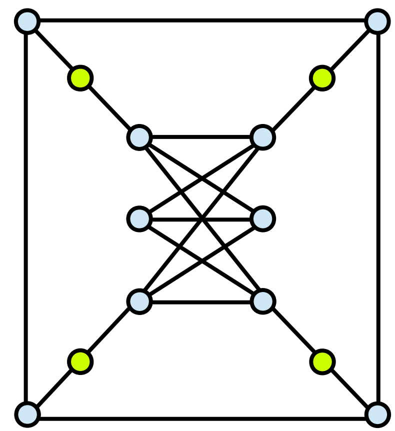
Non-planar



contains $K_{3,3}$

non-planar subgraph

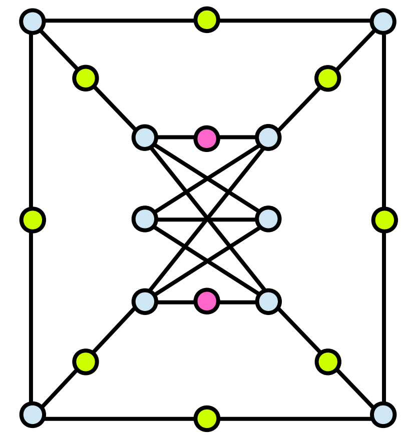
Non-planar



contains $K_{3,3}$

non-planar subgraph

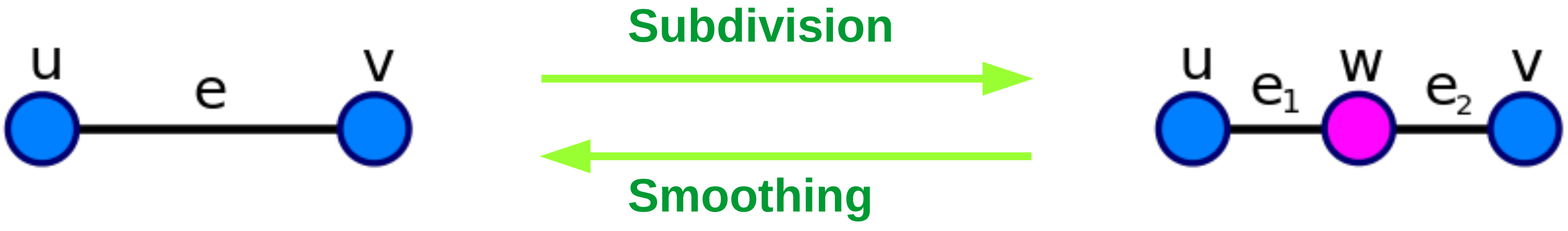
Non-planar



contains a <u>subdivision</u> of $K_{3,3}$

non-planar subgraph

Subdivision and Smoothing



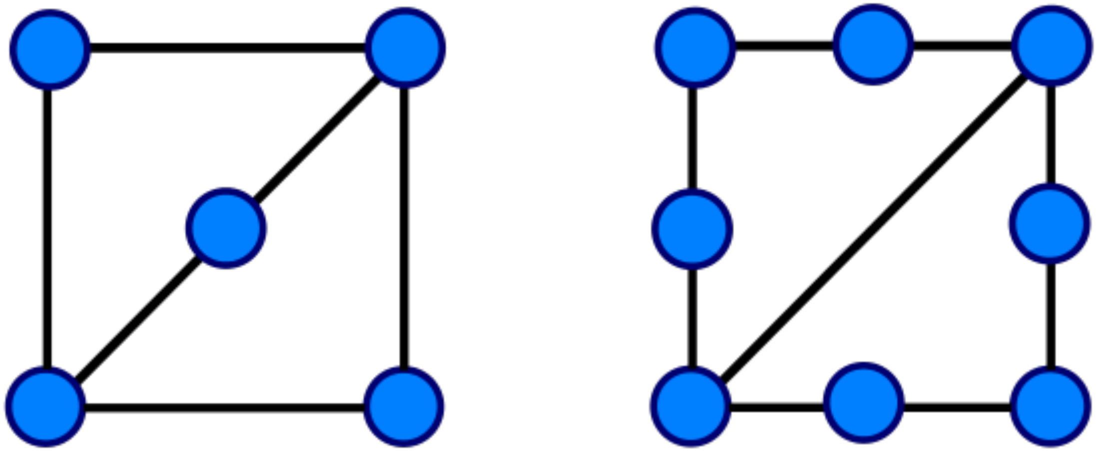
https://en.wikipedia.org/wiki/Planar_graph

Homeomorphism

two graphs G_1 and G_2 are **homeomorphic**
if there is a graph **isomorphism**
from some **subdivision** of G_1
to some **subdivision** of G_2

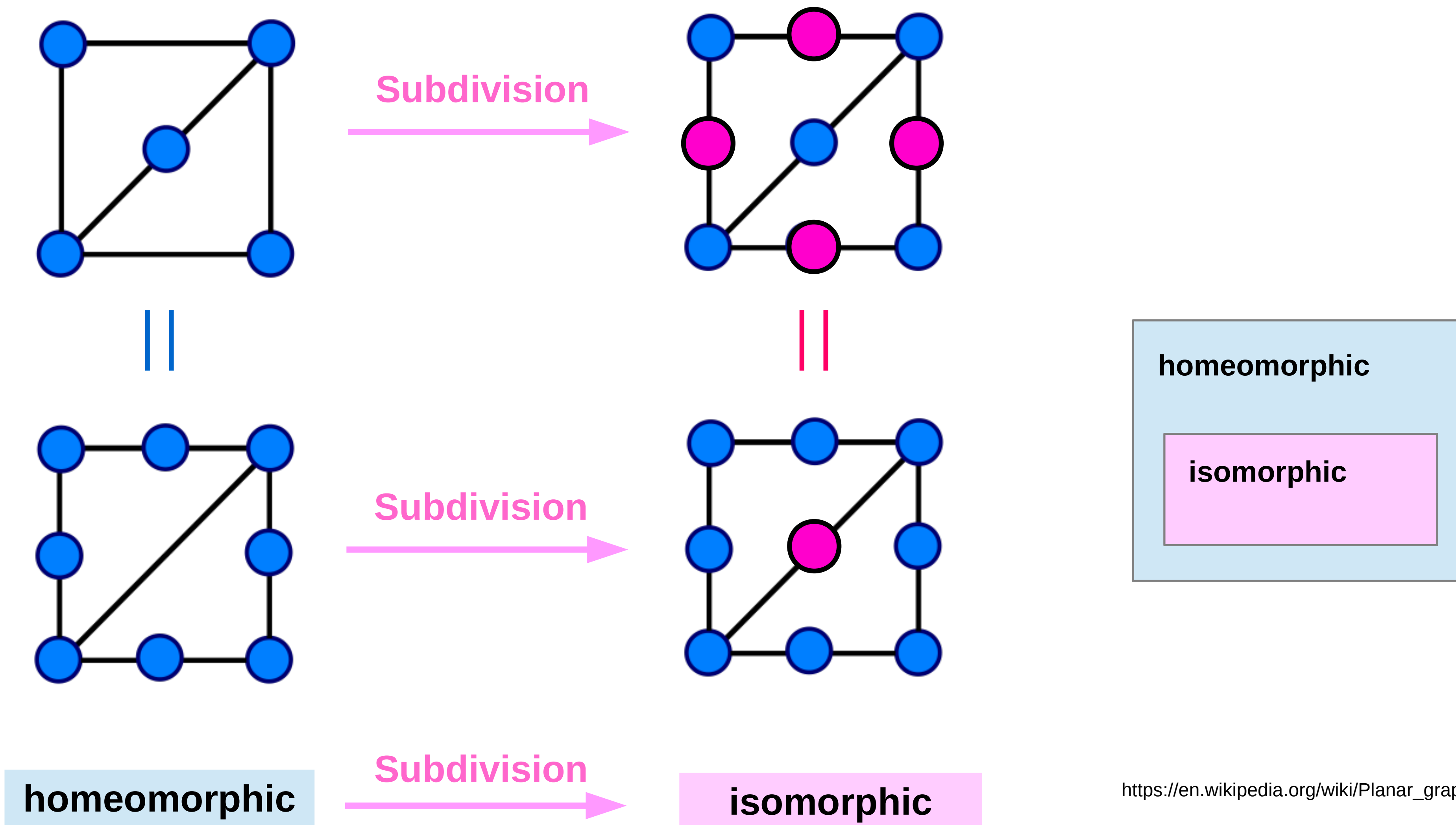
homeo (identity, sameness)

iso (equal)



https://en.wikipedia.org/wiki/Planar_graph

Homeomorphism Examples



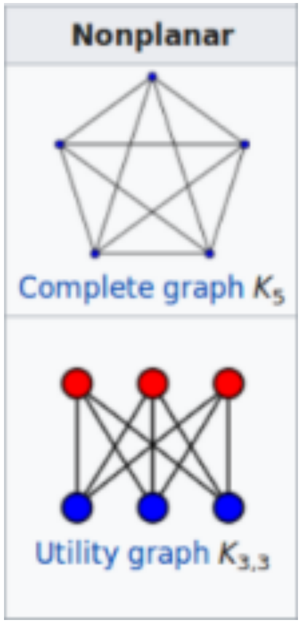
https://en.wikipedia.org/wiki/Planar_graph

subdividing a graph preserves **planarity**.

Kuratowski's theorem states that

> a finite graph is **planar** if and only if
> it contains **no** subgraph **homeomorphic**
> to $\mathbf{K_5}$ (complete graph on five vertices) or
> $\mathbf{K_{3,3}}$ (complete bipartite graph on six vertices,
> three of which connect to each of the other three).

In fact, a graph **homeomorphic** to $\mathbf{K_5}$ or $\mathbf{K_{3,3}}$
is called a **Kuratowski subgraph**.

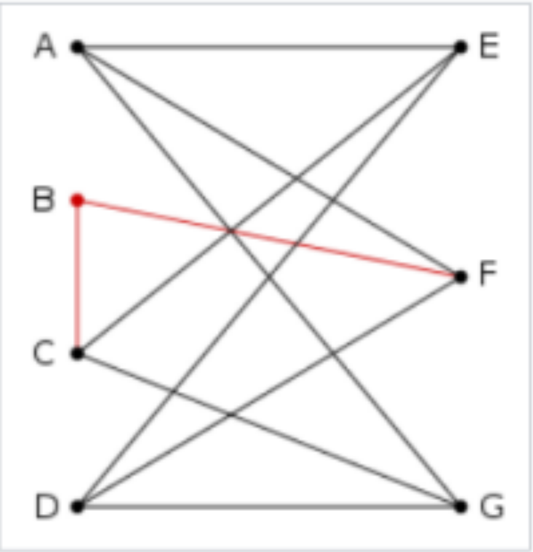


https://en.wikipedia.org/wiki/Planar_graph

Kuratowski's Theorem

A finite graph is **planar** if and only if
it does not contain a **subgraph**
that is a **subdivision** of the complete graph $\mathbf{K_5}$ or
the complete bipartite graph $\mathbf{K_{3,3}}$ (utility graph).

A **subdivision** of a graph results
from inserting vertices into edges
(changing an edge •——• to •—•—•)
zero or more times.

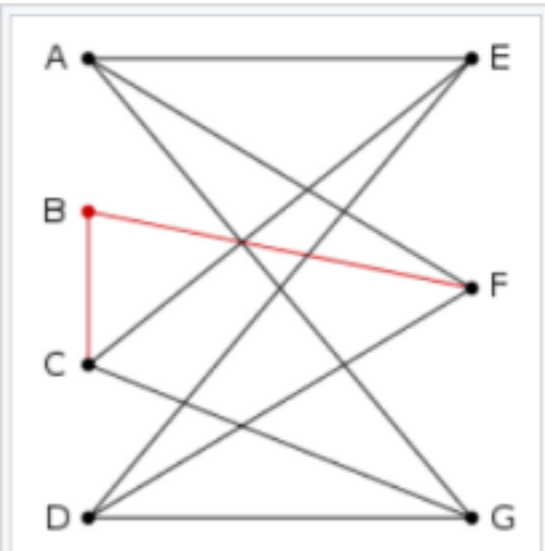


An example of a graph with no K_5 or $K_{3,3}$ subgraph. However, it contains a subdivision of $K_{3,3}$ and is therefore non-planar.

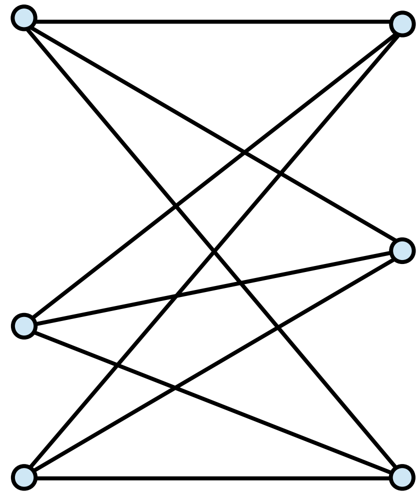
https://en.wikipedia.org/wiki/Planar_graph

Kuratowski's Theorem

homeomorphic



An example of a graph with no K_5 or $K_{3,3}$ subgraph. However, it contains a subdivision of $K_{3,3}$ and is therefore non-planar.



planar

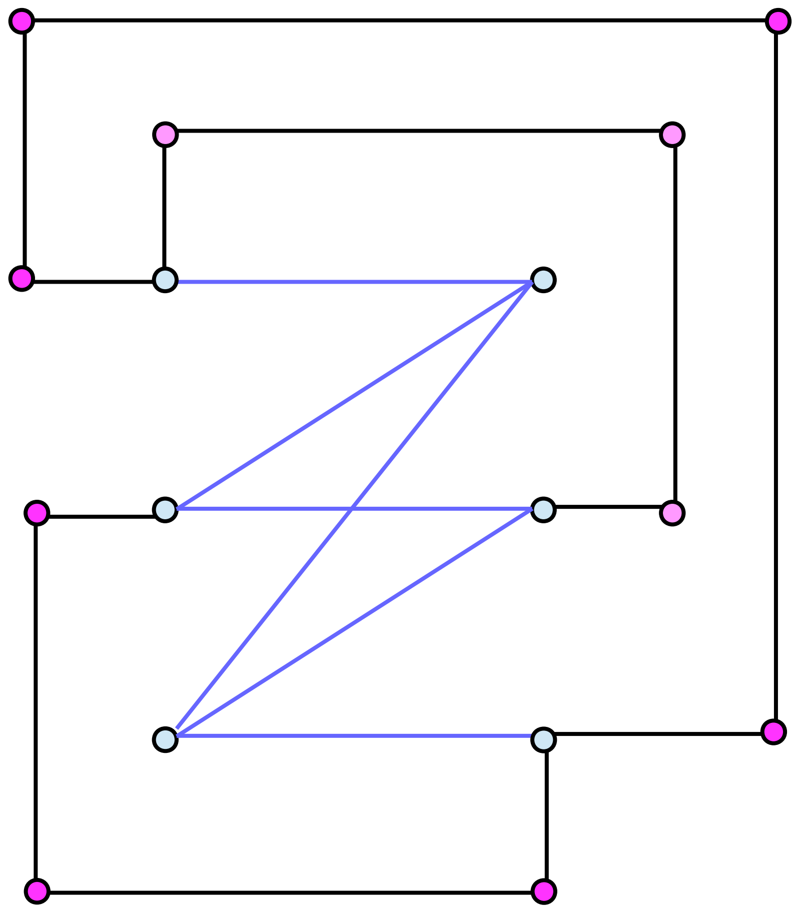
no subgraph **homeomorphic** to $\mathbf{K_5}$ or $\mathbf{K_{3,3}}$

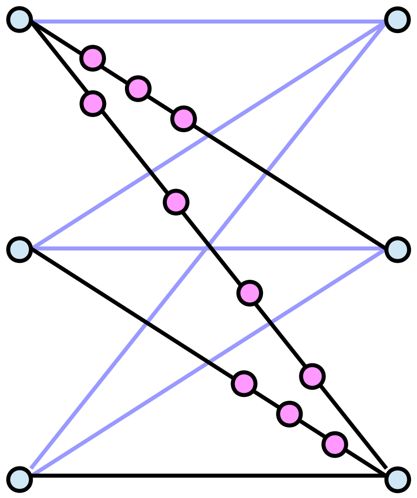
Non-planar

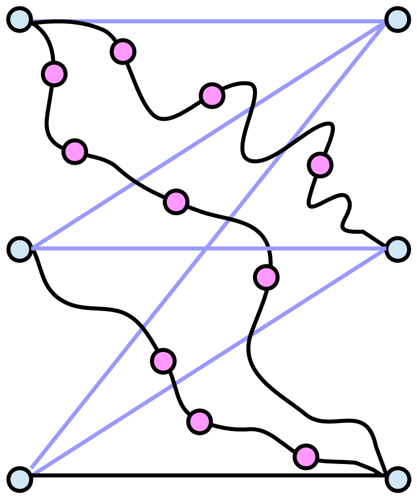
subgraph **homeomorphic** to $\mathbf{K_5}$ or $\mathbf{K_{3,3}}$

https://en.wikipedia.org/wiki/Planar_graph

Homeomorphic to $K_{3,3}$







References

[1] http://en.wikipedia.org/
[2]

Tree Overview (1A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice and Octave.

Tree

a tree is an **undirected** graph in which
any two **vertices** are **connected**
by exactly **one path**.

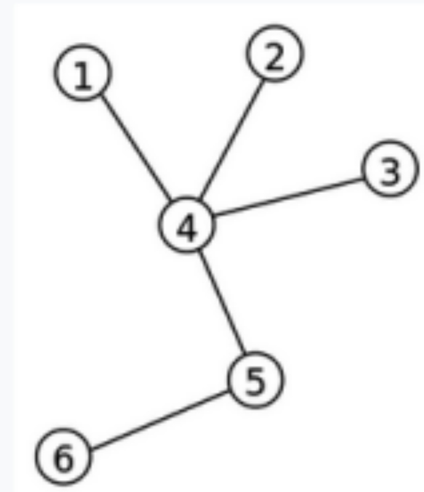
any **acyclic connected** graph is a **tree**.

A **forest** is a disjoint union of trees.

connect

acyclic

Trees



A labeled tree with 6 vertices and 5 edges.

Vertices	v
Edges	$v - 1$
Chromatic number	2 if $v > 1$

https://en.wikipedia.org/wiki/Tree_(graph_theory)

Tree Condition (1)

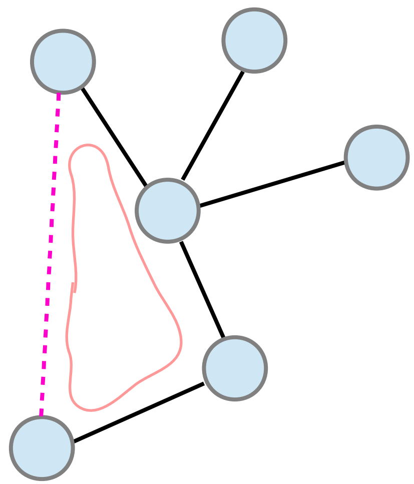
A **tree** is an **undirected** graph G
that satisfies any of the following equivalent conditions:

G is **connected** and has no **cycles**.
G is **acyclic**, and a **simple cycle** is formed if any **edge** is added to G.
G is **connected**, but is not **connected** if any single **edge** is removed from G.
G is **connected** and the 3-vertex complete graph $\mathbf{K_3}$ is not a **minor** of G.
Any **two vertices** in G can be **connected** by a unique **simple path**.

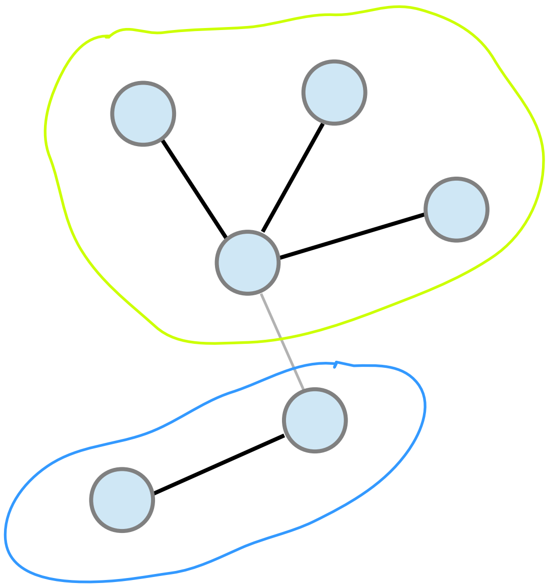
https://en.wikipedia.org/wiki/Tree_(graph_theory)

Tree Condition (2)

G is <u>**acyclic**</u>, and a **simple cycle** is formed if any **edge** is <u>added</u> to G.



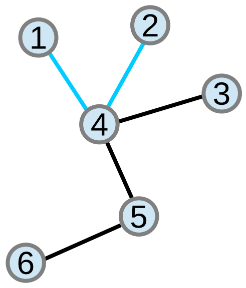
G is <u>**connected**</u>, but is <u>not</u> **connected** if any single **edge** is <u>removed</u> from G.



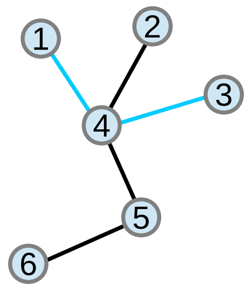
https://en.wikipedia.org/wiki/Tree_(graph_theory)

Tree Condition (3)

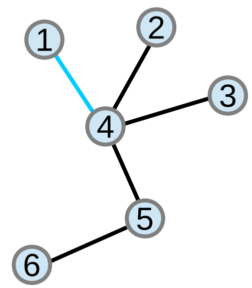
$p_{1,2}$



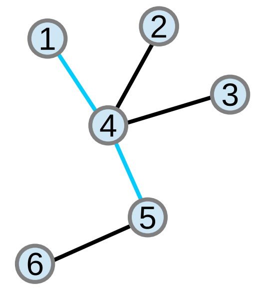
$p_{1,3}$



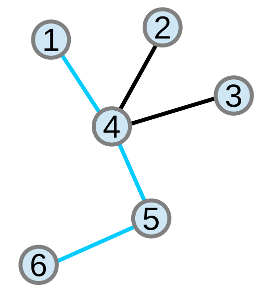
$p_{1,4}$



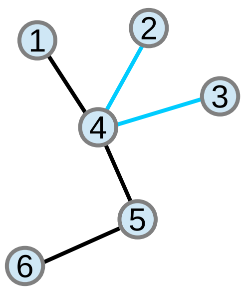
$p_{1,5}$



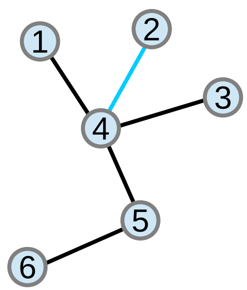
$p_{1,6}$



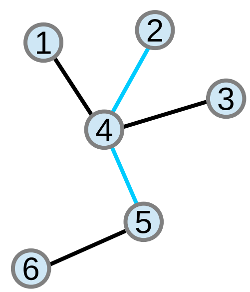
$p_{2,3}$



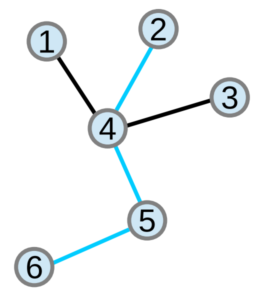
$p_{2,4}$



$p_{2,5}$



$p_{2,6}$



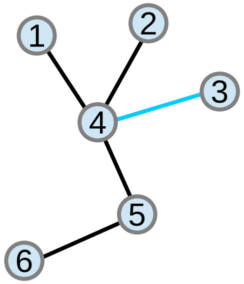
Any **two vertices** in G can be **connected** by a unique **simple path**.

https://en.wikipedia.org/wiki/Tree_(graph_theory)

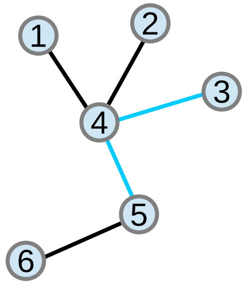
Young Won Lim
6/8/18

Tree Condition (4)

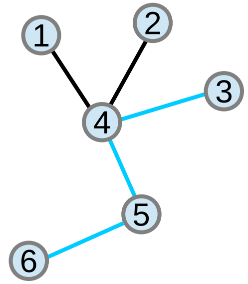
$p_{3,4}$



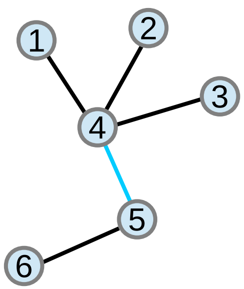
$p_{3,5}$



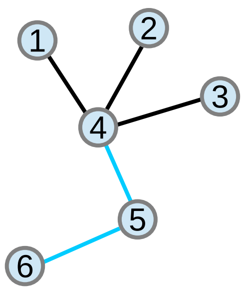
$p_{3,6}$



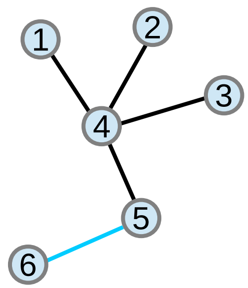
$p_{4,5}$



$p_{4,6}$



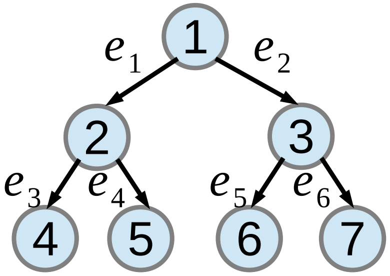
$p_{5,6}$



Any **two vertices** in G can be **connected** by a <u>unique</u> **simple path**.

https://en.wikipedia.org/wiki/Tree_(graph_theory)

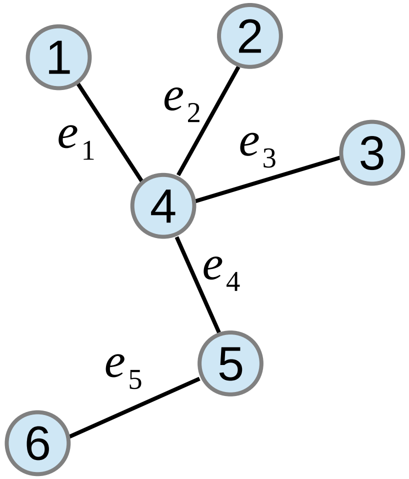
Tree Condition (5)

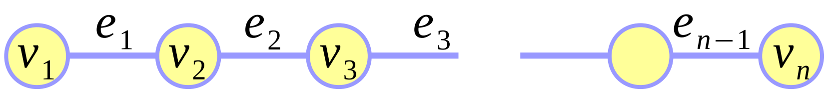


If G has finitely many **vertices**,
say **n vertices**, then the above statements
are also equivalent to any of the following conditions:

G is **connected** and has **n − 1 edges**.

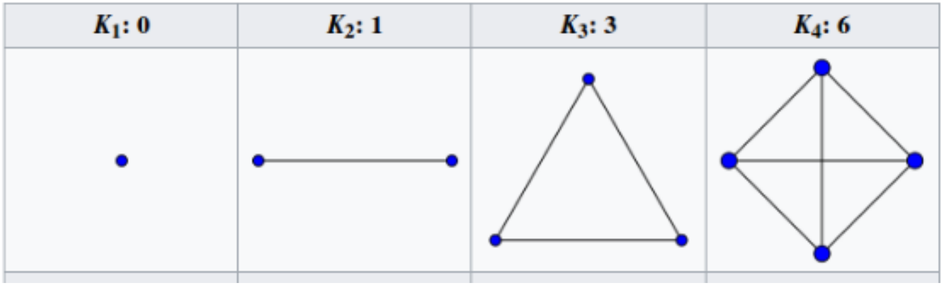
G has **no simple cycles** and has **n − 1 edges**.



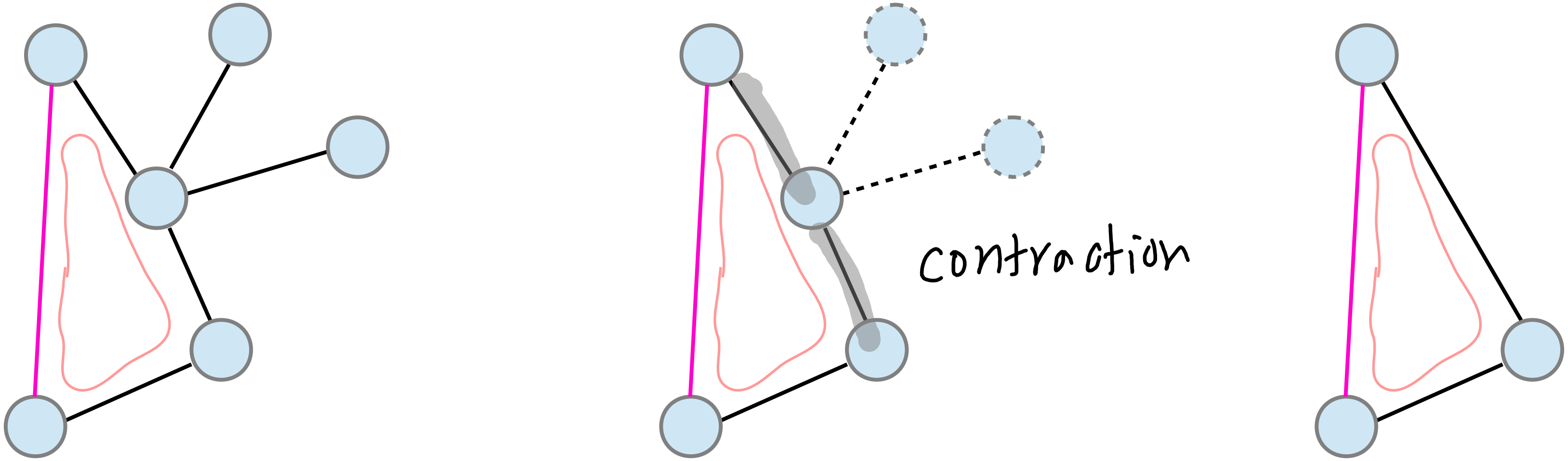


https://en.wikipedia.org/wiki/Tree_(graph_theory)

Tree Condition (6)



G is **connected** and the 3-vertex complete graph $\mathbf{K_3}$ is not a **minor** of G.



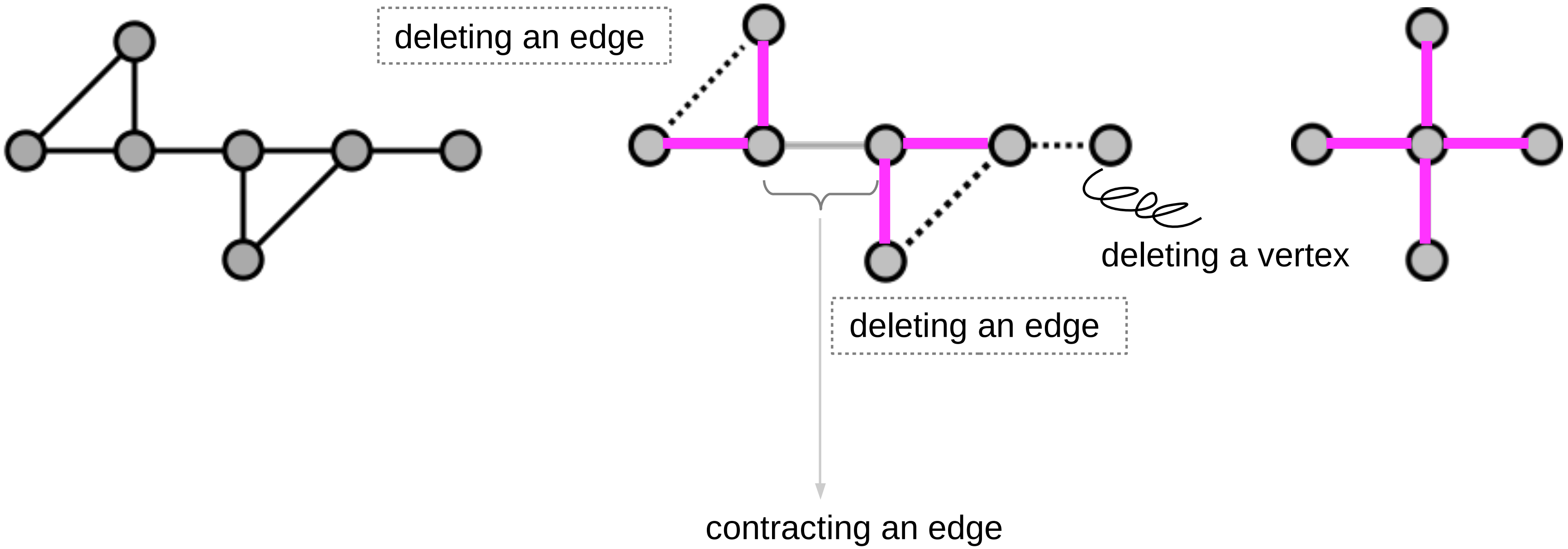
deleting edges
deleting vertices

contracting edges

https://en.wikipedia.org/wiki/Tree_(graph_theory)

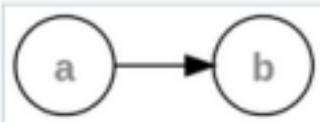
Graph Minor

In graph theory, an undirected graph H
is called a minor of the graph G
if H can be formed from G
by **deleting edges** and **vertices** and
by **contracting edges**.

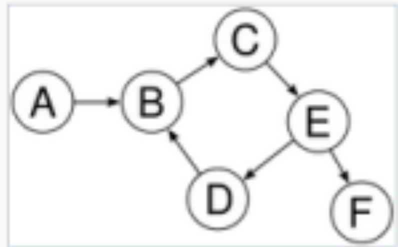


https://en.wikipedia.org/wiki/Graph_minor

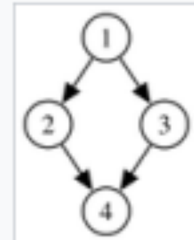
Tree Examples



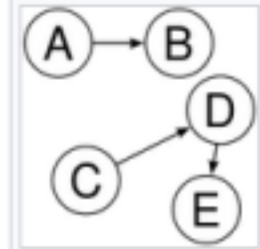
Each linear list is trivially a tree



Not a tree: cycle B→C→E→D→B. B has more than one parent (inbound edge).



Not a tree: undirected cycle 1-2-4-3. 4 has more than one parent (inbound edge).

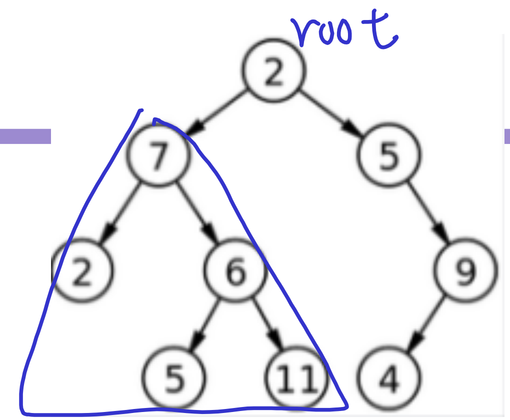


Not a tree: two non-connected parts, A→B and C→D→E. There is more than one root.

Not a tree: cycle A→A. A is the root but it also has a parent.

https://en.wikipedia.org/wiki/Tree_(data_structure)

Terminology used in trees (1)



Root

The top node in a tree.

Child 7 < 2 6

A node directly connected to another node when moving away from the Root.

Parent

The converse notion of a child. 2 6 > 7

Siblings

A group of nodes with the same parent. 2 6

Descendant

A node reachable by repeated proceeding from parent to child. 2, 6, 5, 11

Ancestor

A node reachable by repeated proceeding from child to parent. 11 → 6 7 2

https://en.wikipedia.org/wiki/Tree_(data_structure)

Terminology used in trees (2)

Leaf (less commonly called **External node**)

A node with no children.

Branch (**Internal node**)

A node with at least one child.

Degree

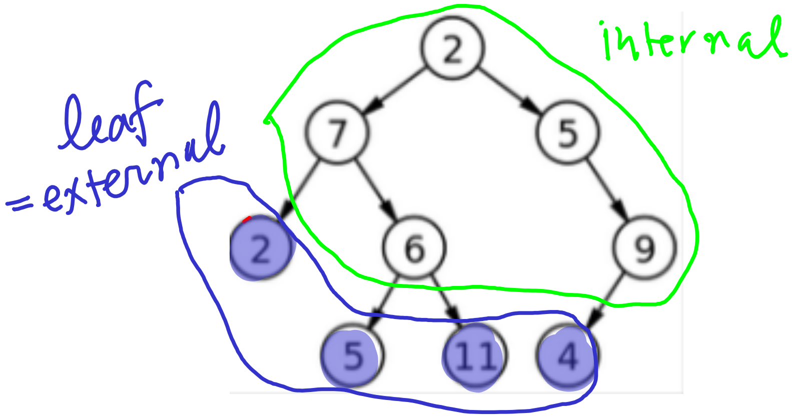
The number of subtrees of a node.

Edge

The connection between one node and another.

Path

A sequence of nodes and edges connecting a node with a descendant.



https://en.wikipedia.org/wiki/Tree_(data_structure)

Terminology used in trees (3)

Level

The level of a node is defined
by 1 + (the number of connections between the node and the root).

Height of node — **Depth**

The height of a node is the number of edges
on the longest path between that node and a leaf.

Height of tree — **Depth**

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

Depth — **Height**

The depth of a node is the number of edges
from the tree's root node to the node.

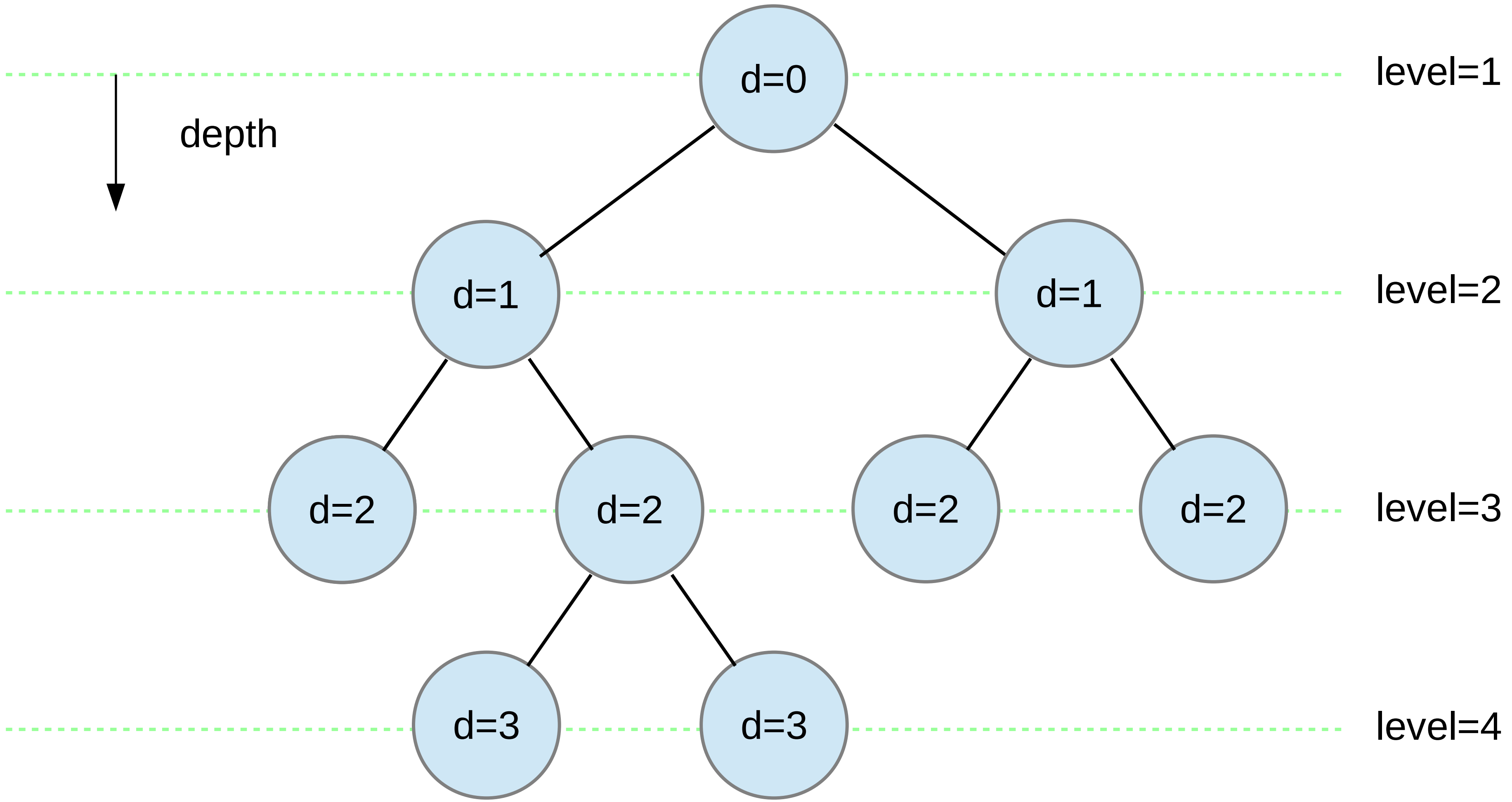
Forest

A forest is a set of n ≥ 0 disjoint trees.

Some literatures have the reversed definitions of height and depth

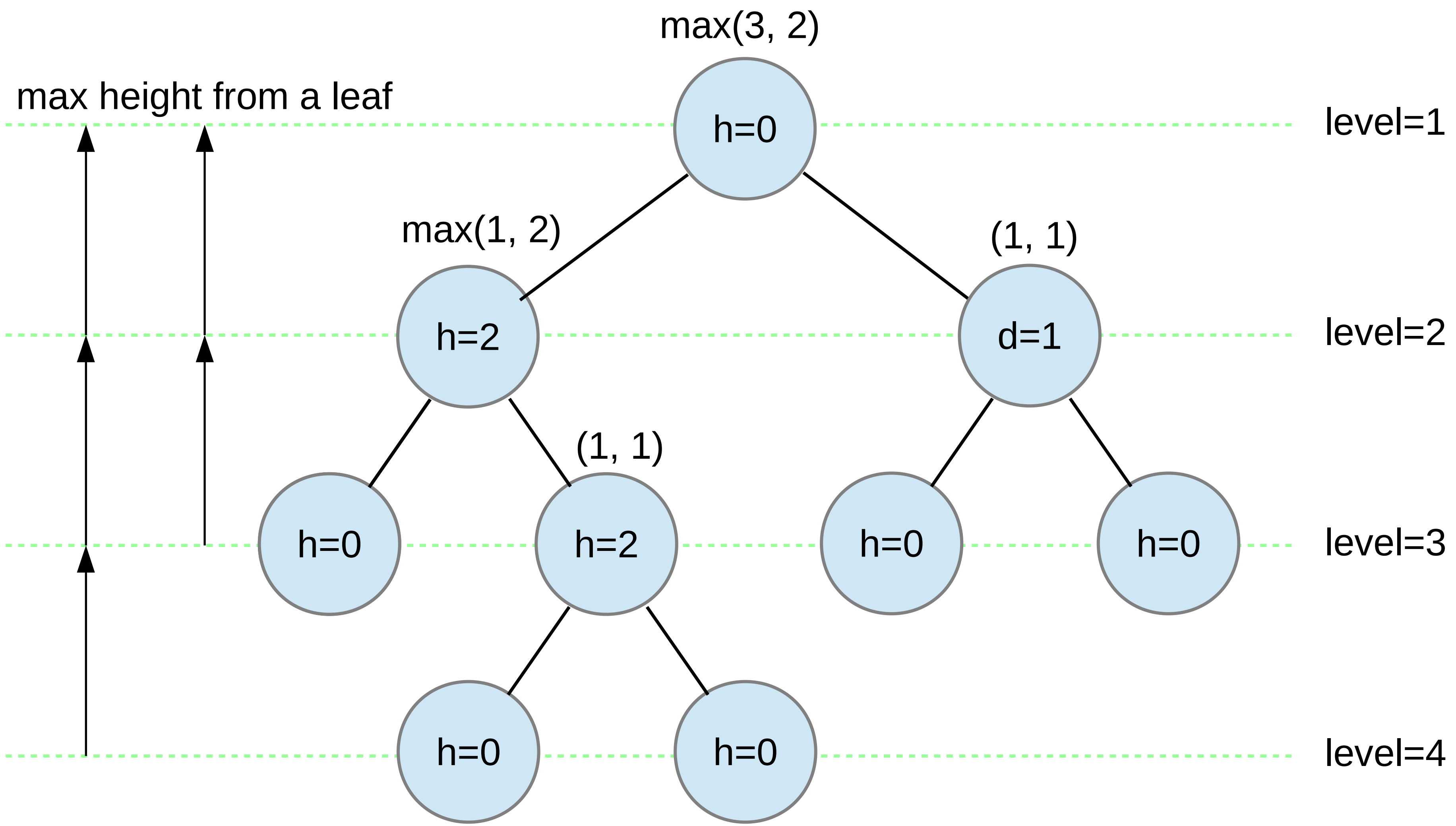
https://en.wikipedia.org/wiki/Tree_(data_structure)

Depth



https://en.wikipedia.org/wiki/Tree_(data_structure)

Height



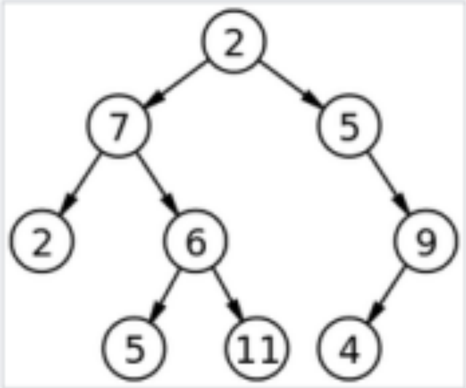
https://en.wikipedia.org/wiki/Tree_(data_structure)

Binary Tree

a **binary tree** is a tree data structure in which each **node** has <u>at most</u> <u>two</u> **children**, (the **left child**, the **right child**)

A **recursive definition** using just set theory notions is that a (non-empty) binary tree is a tuple **(L, S, R)**, where **L** and **R** are **binary trees** or the **empty set** and **S** is a **singleton set**.

Some authors allow the binary tree to be the empty set as well.



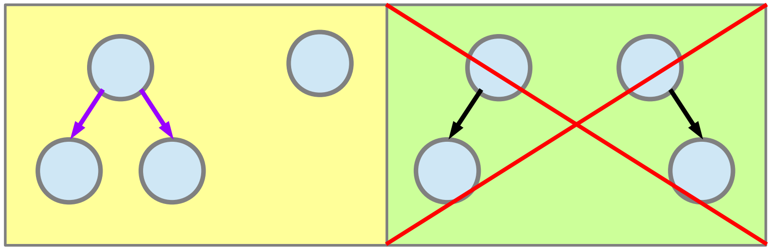
A labeled binary tree of size 9 and height 3, with a root node whose value is 2. The above tree is unbalanced and not sorted.

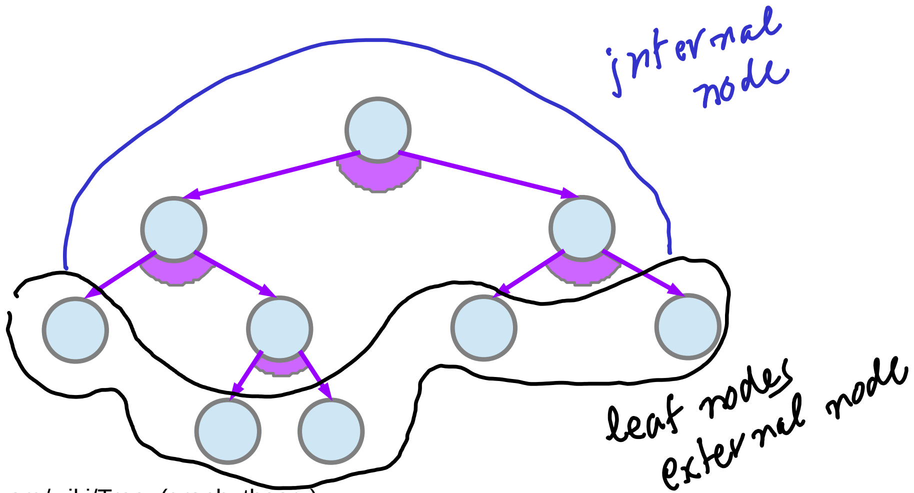
https://en.wikipedia.org/wiki/Binary_tree

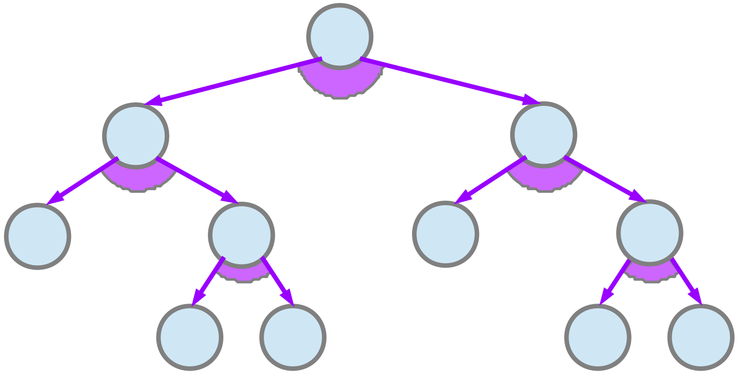
Full Binary Tree

A **rooted binary tree** has a **root node** and every **node** has at most two **children**.

A **full binary tree** is (**proper, plane binary tree**) a **tree** in which every **node** has either **0** or **2 children**.







https://en.wikipedia.org/wiki/Tree_(graph_theory)

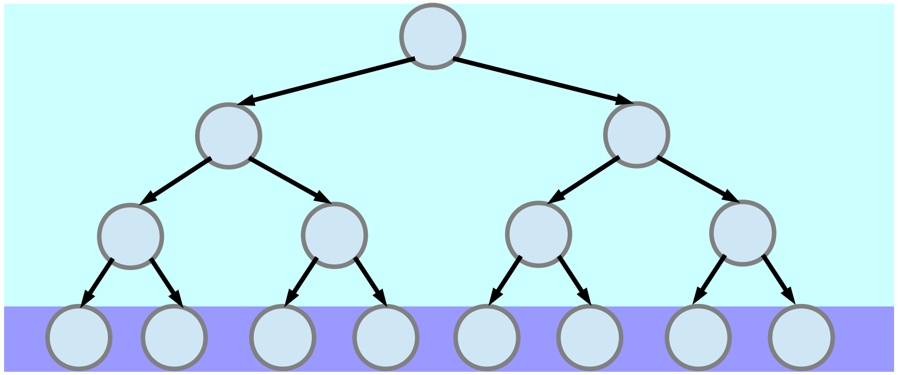
Perfect Binary Trees

A **perfect** **binary tree** is a binary tree in which
all **interior nodes** have two **children** and
all **leaves** have the same **depth** or same **level**.

also called a **complete binary tree**

two **children**

the same **depth** (**level**).

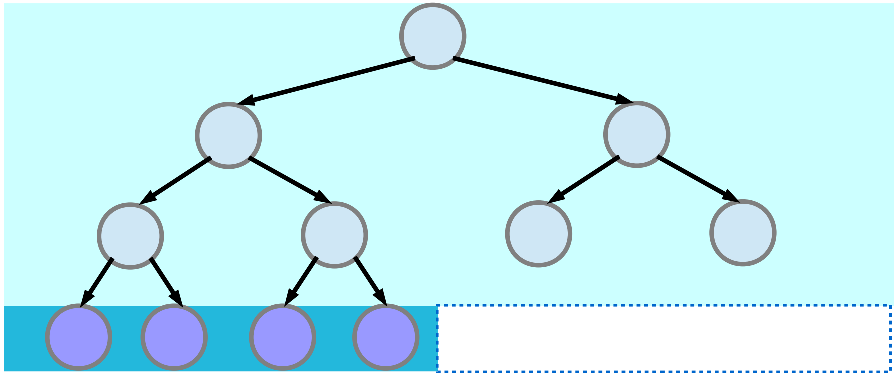


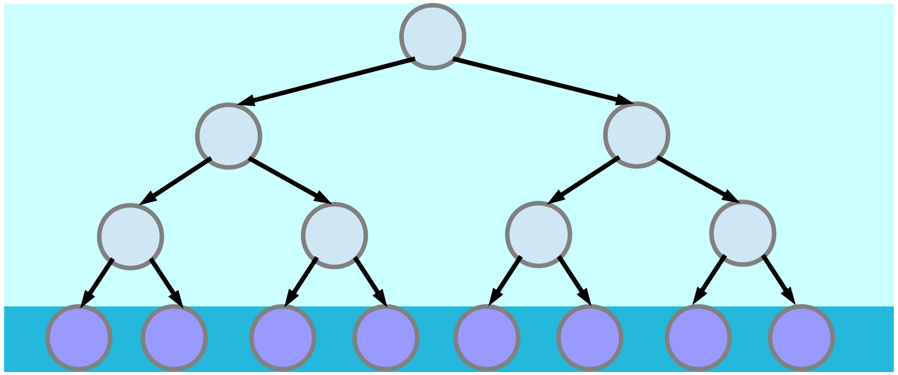
https://en.wikipedia.org/wiki/Tree_(graph_theory)

Complete Binary Trees

In a **complete binary tree**
every level, except possibly the last,
is completely filled,
and all nodes in the last level are
as far left as possible.

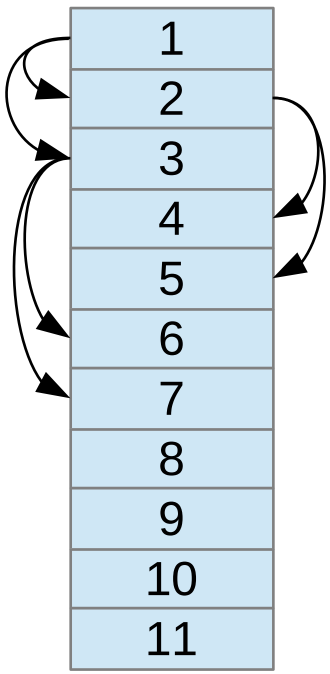
An alternative definition is a **perfect tree**
whose rightmost leaves (perhaps all)
have been removed.





https://en.wikipedia.org/wiki/Tree_(graph_theory)

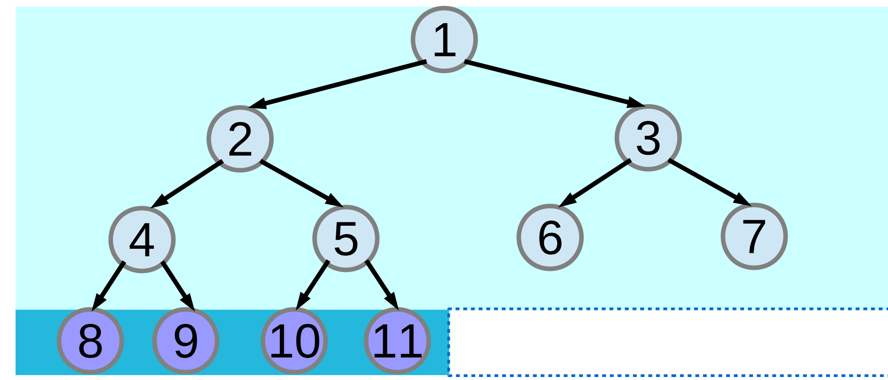
Complete Binary Trees and Linear Arrays

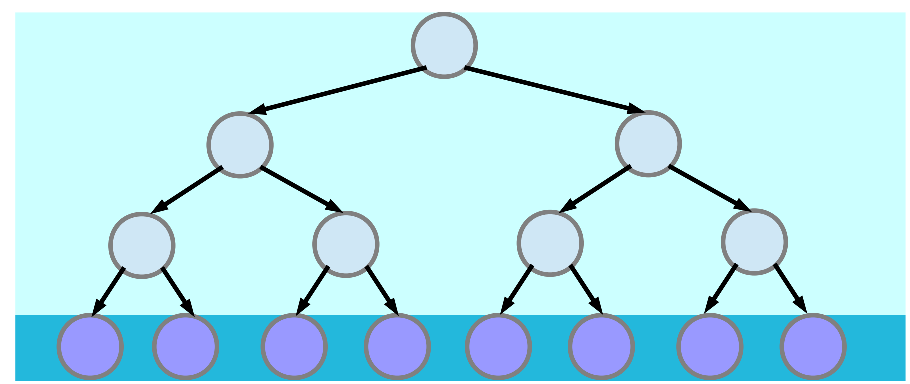


$2 \cdot i$ Left child

$2 \cdot i + 1$ Right child

A complete binary tree can be efficiently represented using an array.



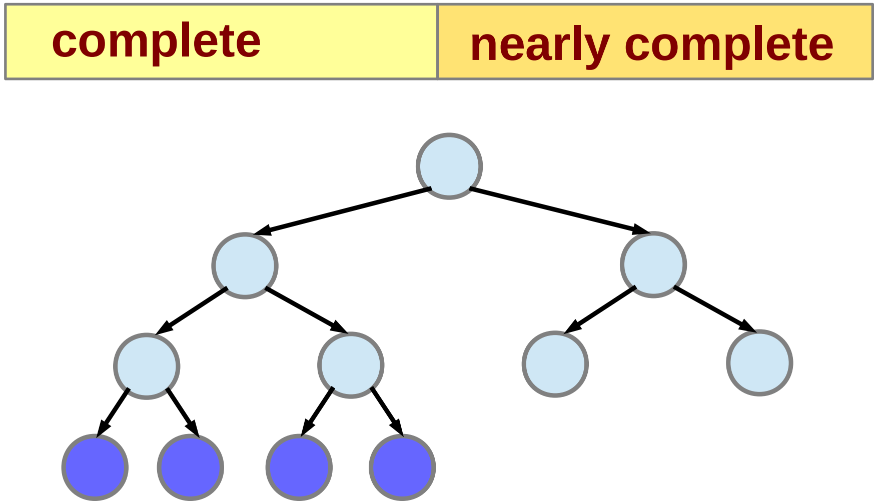


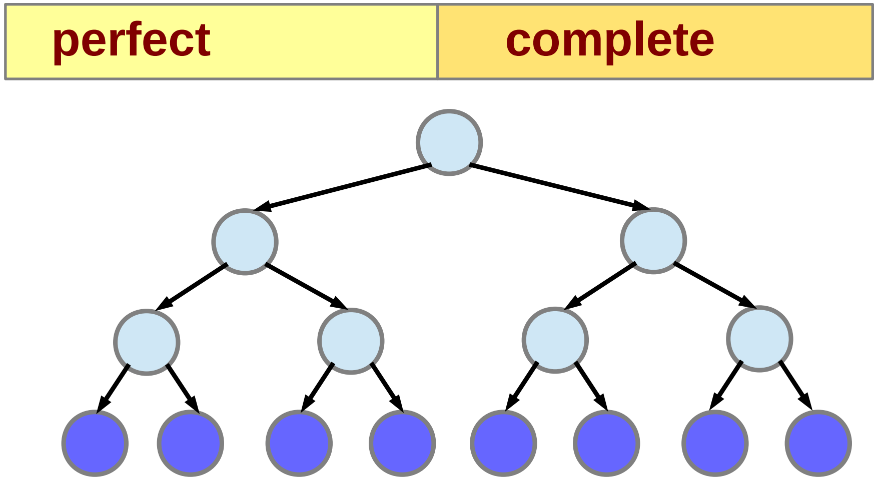
contiguous
no blanks
→ complete

https://en.wikipedia.org/wiki/Tree_(graph_theory)

Different use of compute binary trees

Some authors use the term **complete** to refer instead to a **perfect** binary tree as defined above,
in which case they call this type of tree an **almost complete binary tree** or **nearly complete binary tree**.

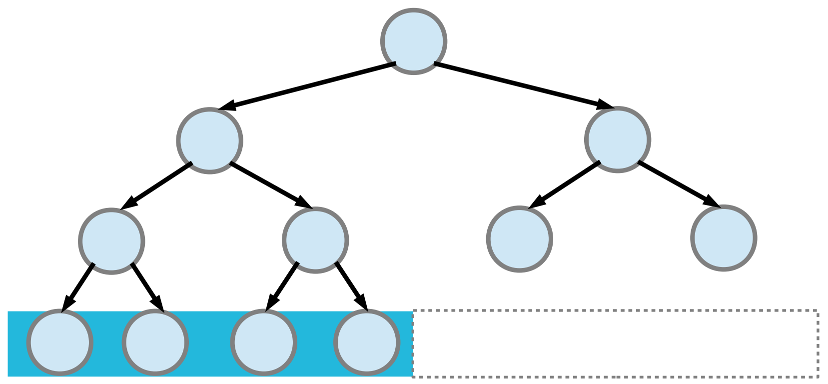




https://en.wikipedia.org/wiki/Tree_(graph_theory)

Properties of Binary Trees (1)

A **complete** **binary tree**
can have between **1** and $\mathbf{2^{m-1}}$ nodes
at the <u>last</u> <u>level</u> **m**.



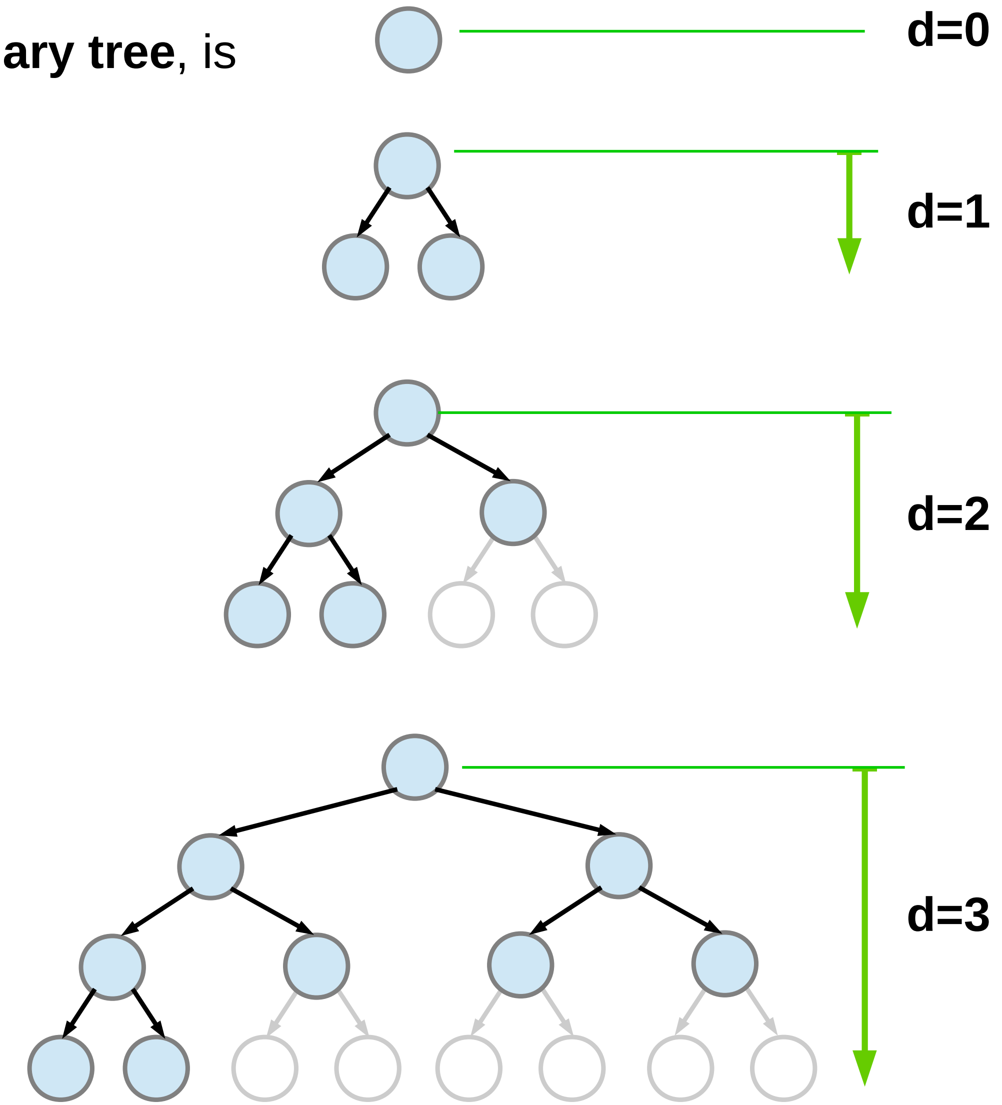
https://en.wikipedia.org/wiki/Tree_(graph_theory)

Young Won Lim
6/8/18

Properties of Binary Trees (2)

The **number** of **nodes n** in a **full binary tree**, is at least $\mathbf{n = 2^d + 1}$ and at most $\mathbf{n = 2^{d+1} - 1}$, where **d** is the **detph** of the tree.

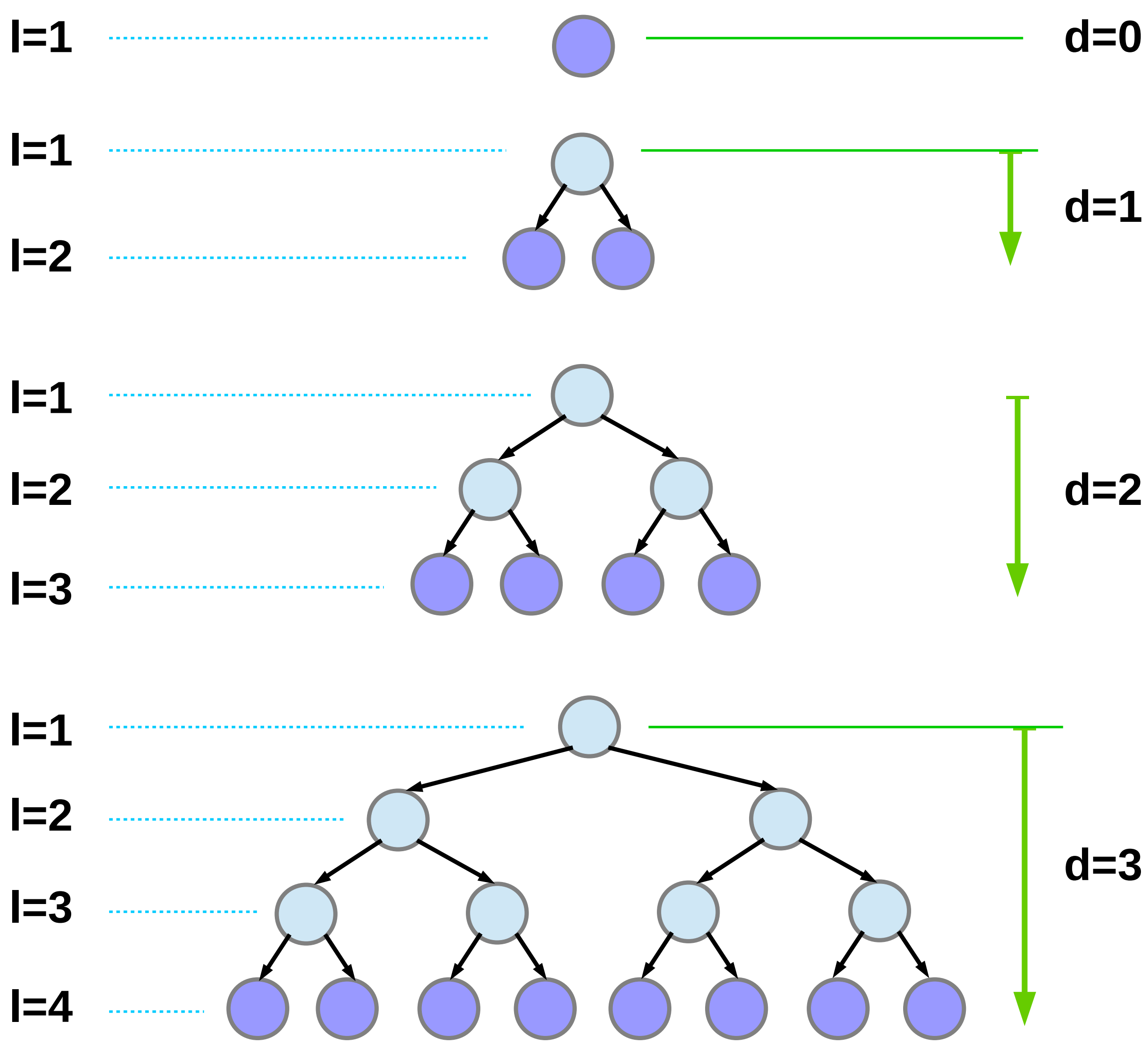
A tree consisting of only a **root node** has a **depth** of **0**.



https://en.wikipedia.org/wiki/Tree_(graph_theory)

Properties of Binary Trees (3)

$2^{d+1}-1$	2^d	2^0
$2^{d+1}-1$	2^d-1	2^1-1
	2^d	2^1
$2^{d+1}-1$	2^d-1	2^2-1
	2^d	2^2
$2^{d+1}-1$	2^d-1	2^3-1
	2^d	2^3

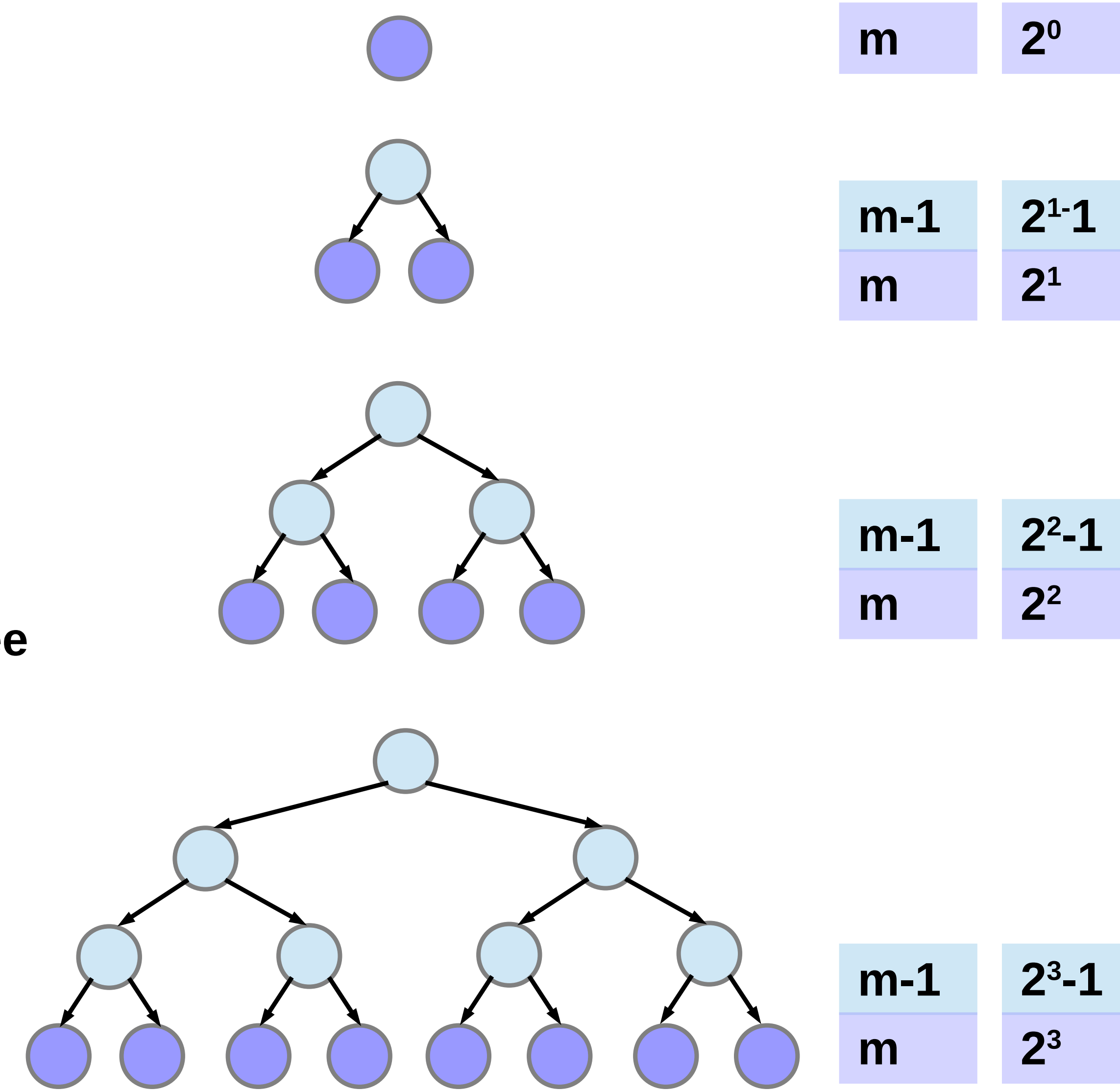


https://en.wikipedia.org/wiki/Tree_(graph_theory)

Properties of Binary Trees (4)

The number of **leaf nodes** is **m**
in a **perfect binary tree**,
is **m=(n+1)/2**

because the number of **non-leaf** (**internal**) **nodes** is **m–1**



This means that a **perfect binary tree** with **m leaves** has
n = 2m–1 nodes.

https://en.wikipedia.org/wiki/Tree_(graph_theory)

References

[1] http://en.wikipedia.org/
[2]

Tree Traversal (1A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice and Octave.

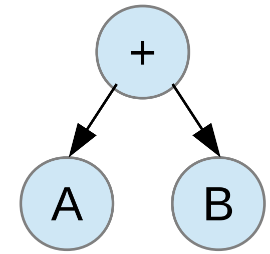
Infix, Prefix, Postfix Notations

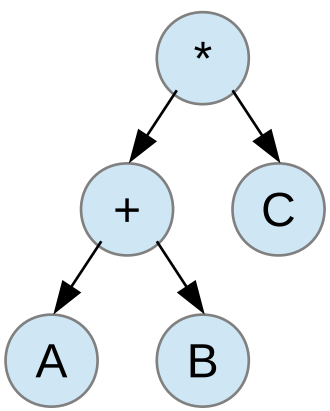
Infix Notation	Prefix Notation	Postfix Notation
A + B	+ A B	A B +
(A + B) * C	* + A B C	A B + C *
A * (B + C)	* A + B C	A B C + *
A / B + C / D	+ / A B / C D	A B / C D / +
((A + B) * C) – D	– * + A B C D	A B + C * D –

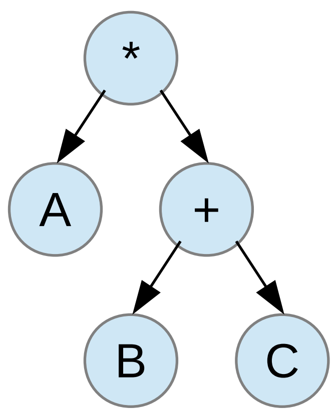
https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

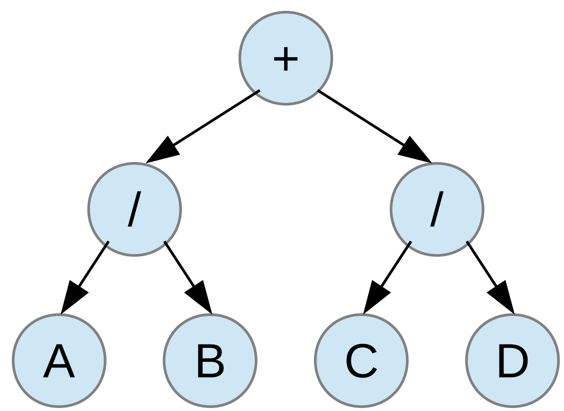
Infix, Prefix, Postfix Notations and Binary Trees

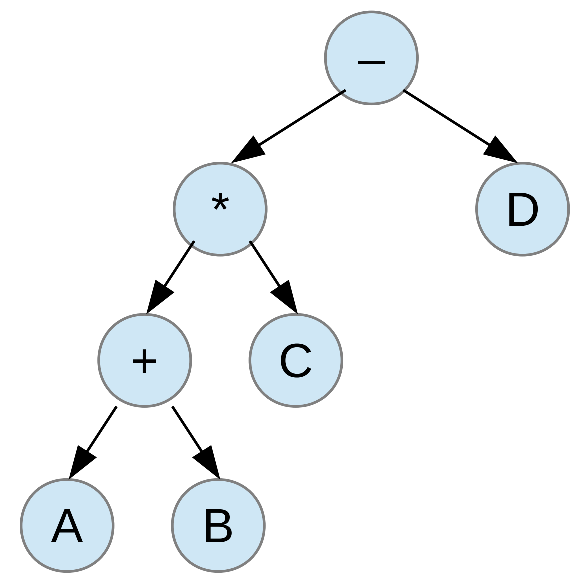
Infix Notation	Prefix Notation	Postfix Notation
A + B	+ A B	A B +
(A + B) * C	* + A B C	A B + C *
A * (B + C)	* A + B C	A B C + *
A / B + C / D	+ / A B / C D	A B / C D / +
((A + B) * C) – D	– * + A B C D	A B + C * D –









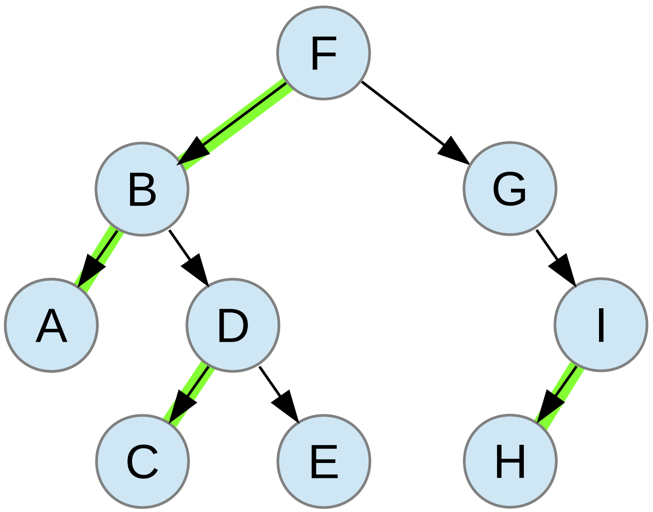


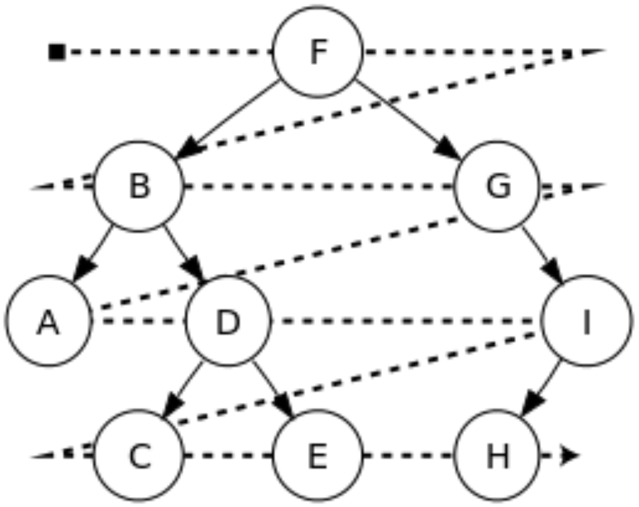
https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.htm

Tree Traversal

Depth First Search
- Pre-Order
- In-order
- Post-Order

Breadth First Search

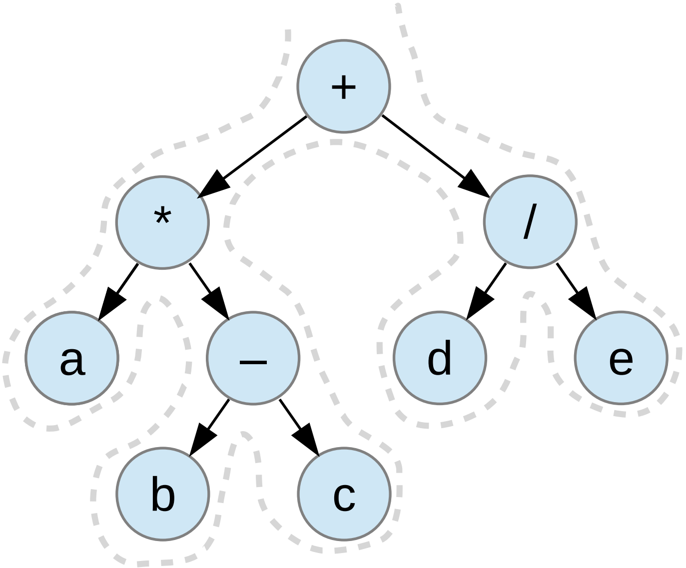




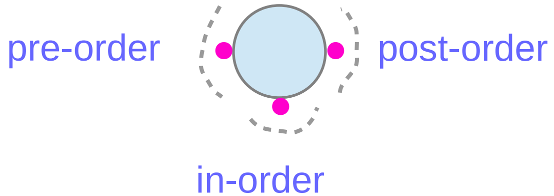
https://en.wikipedia.org/wiki/Morphism

Depth First Search on Binary Trees

Depth First Search

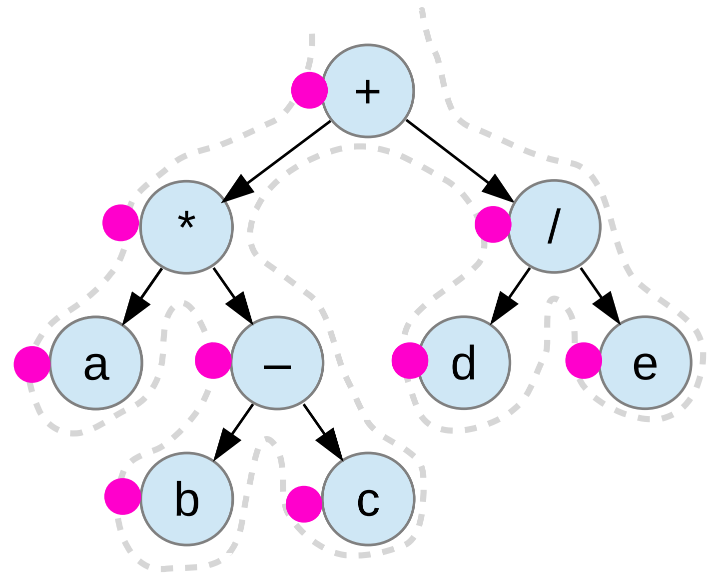


Three Variations
Pre-Order, In-Order, Post-Order



https://en.wikipedia.org/wiki/Tree_traversal

Pre-Order Binary Tree Traversals

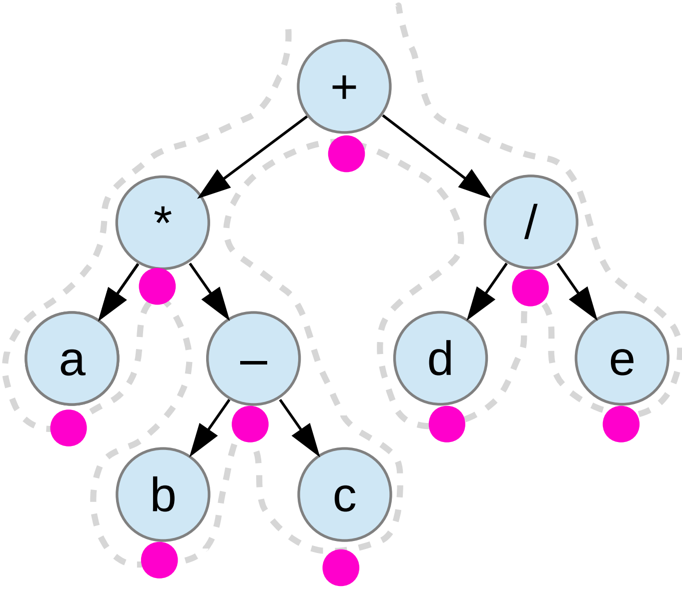


(a*(b-c))+(d/e)

a * b – c + d / e	Infix notation
+ * a – b c / d e	Prefix notation
a b c – * d e / +	Postfix notation

https://en.wikipedia.org/wiki/Tree_traversal

In-Order Binary Tree Traversals



(a*(b-c))+(d/e)

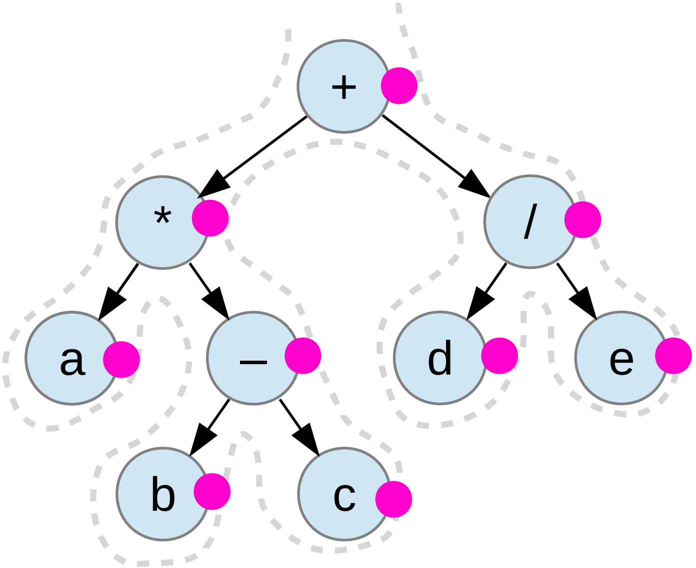
a * b – c + d / e   Infix notation

+ * a – b c / d e   Prefix notation

a b c – * d e / +   Postfix notation

https://en.wikipedia.org/wiki/Tree_traversal

Post-Order Binary Tree Traversals



(a*(b-c))+(d/e)

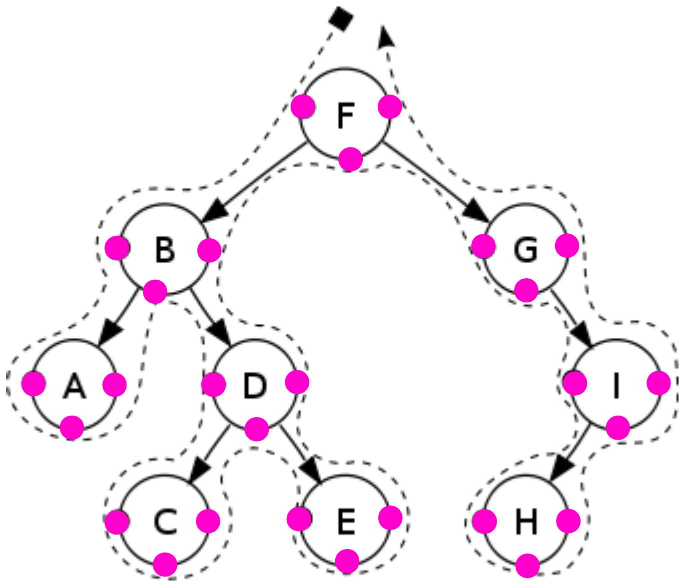
a * b – c + d / e	Infix notation
+ * a – b c / d e	Prefix notation
a b c – * d e / +	Postfix notation

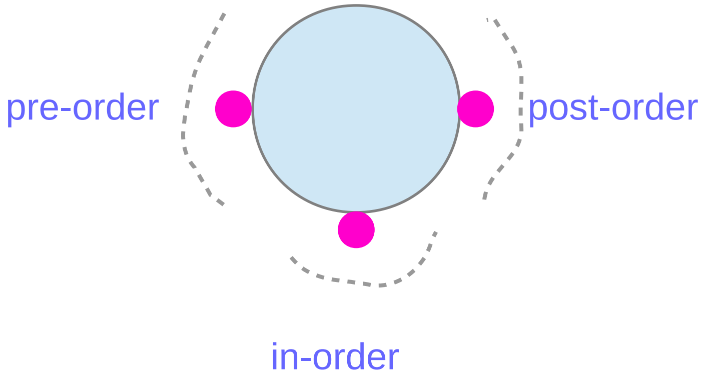
https://en.wikipedia.org/wiki/Tree_traversal

Binary Tree Traversal

Depth First Search
- Pre-Order
- In-order
- Post-Order

Breadth First Search





https://en.wikipedia.org/wiki/Tree_traversal

Pre-Order Traversal on Binary Trees

pre-order function

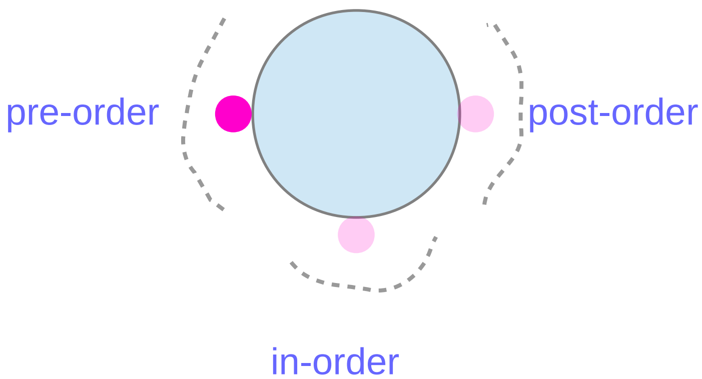
Check if the current node is empty / null.

Display the data part of the root (or current node).

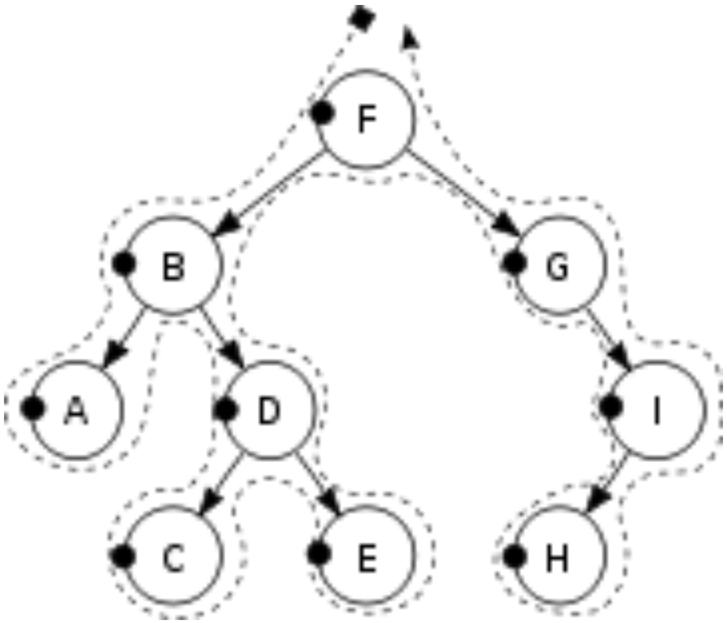
Traverse the **left** subtree by recursively calling the **pre-order** function.

Traverse the **right** subtree by recursively calling the **pre-order** function.

FBADCEGIH



https://en.wikipedia.org/wiki/Tree_traversal



In-Order Traversal on Binary Trees

in-order function

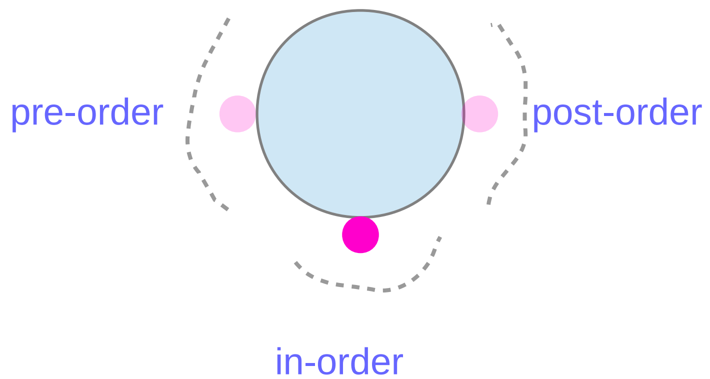
Check if the current node is empty / null.

Traverse the left subtree by recursively calling the **in-order** function.

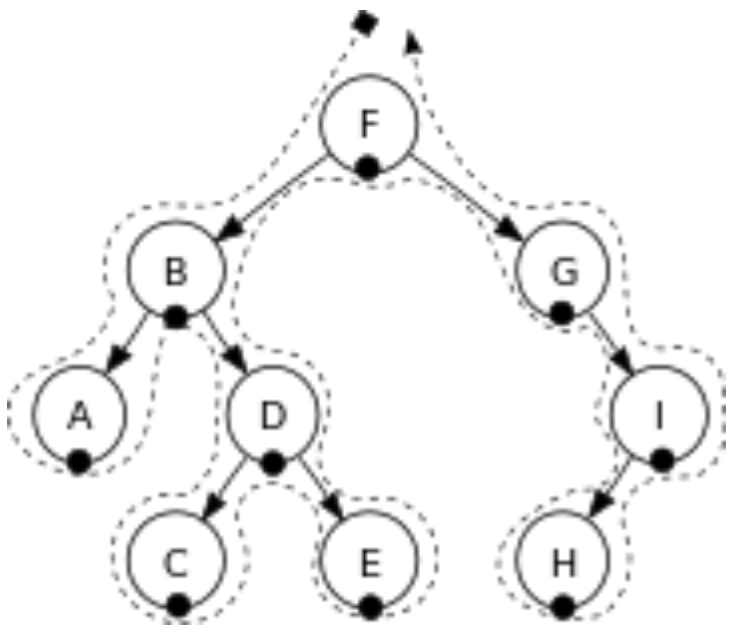
Display the data part of the root (or current node).

Traverse the right subtree by recursively calling the **in-order** function.

ABCDEFGHI



https://en.wikipedia.org/wiki/Tree_traversal



Post-Order Traversal on Binary Trees

post-order function

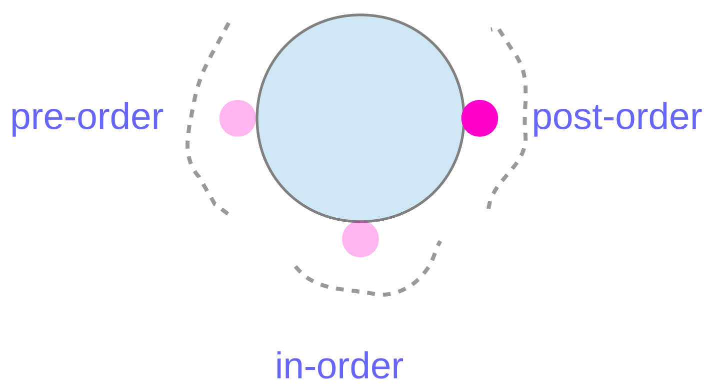
Check if the current node is empty / null.

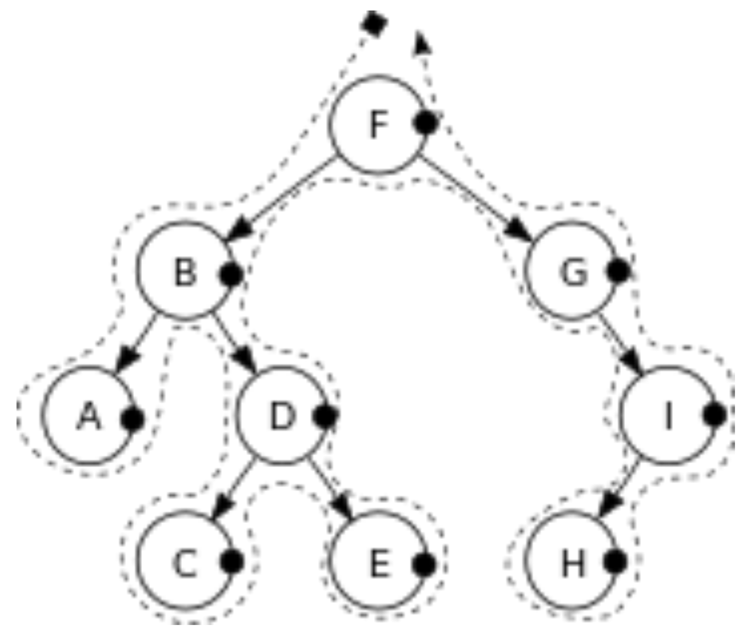
Traverse the left subtree by recursively calling the **post-order** function.

Traverse the right subtree by recursively calling the **post-order** function.

<u>**Display**</u> the data part of the root (or current node).

ACEDBHIGH





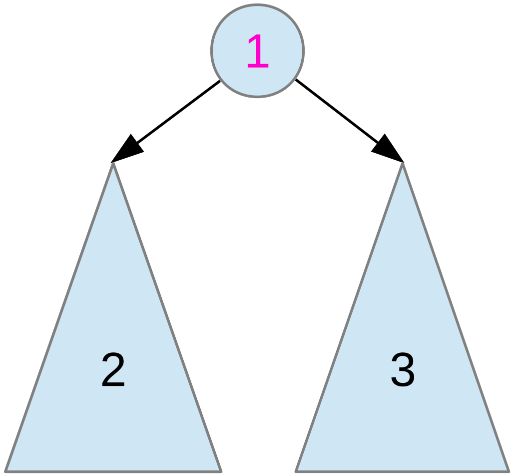
https://en.wikipedia.org/wiki/Tree_traversal

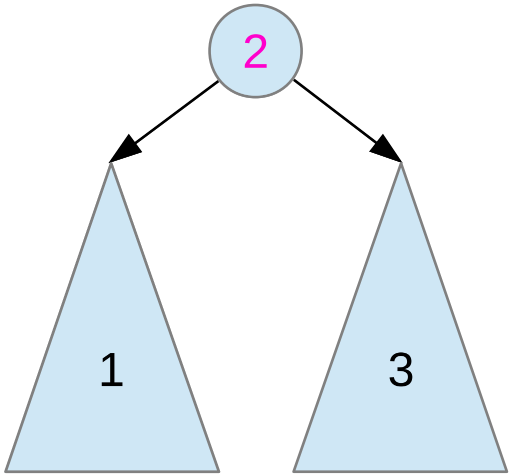
Recursive Algorithms

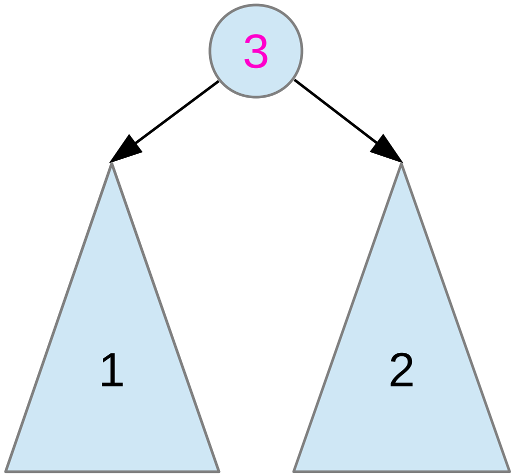
```
preorder(node)
  if (node = null)
    return
  visit(node)
  preorder(node.left)
  preorder(node.right)
```

```
inorder(node)
  if (node = null)
    return
  inorder(node.left)
  visit(node)
  inorder(node.right)
```

```
postorder(node)
  if (node = null)
    return
  postorder(node.left)
  postorder(node.right)
  visit(node)
```



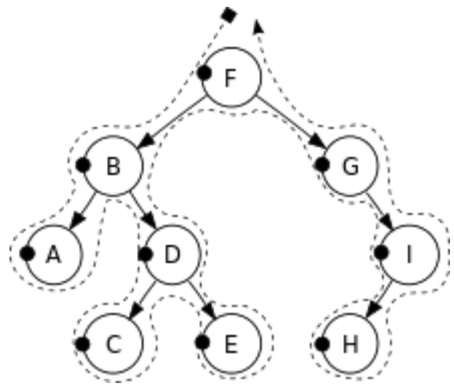


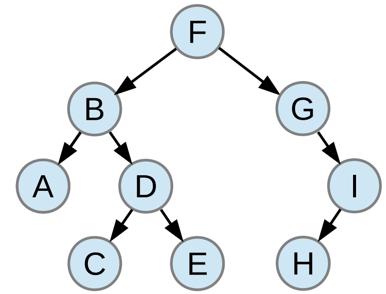


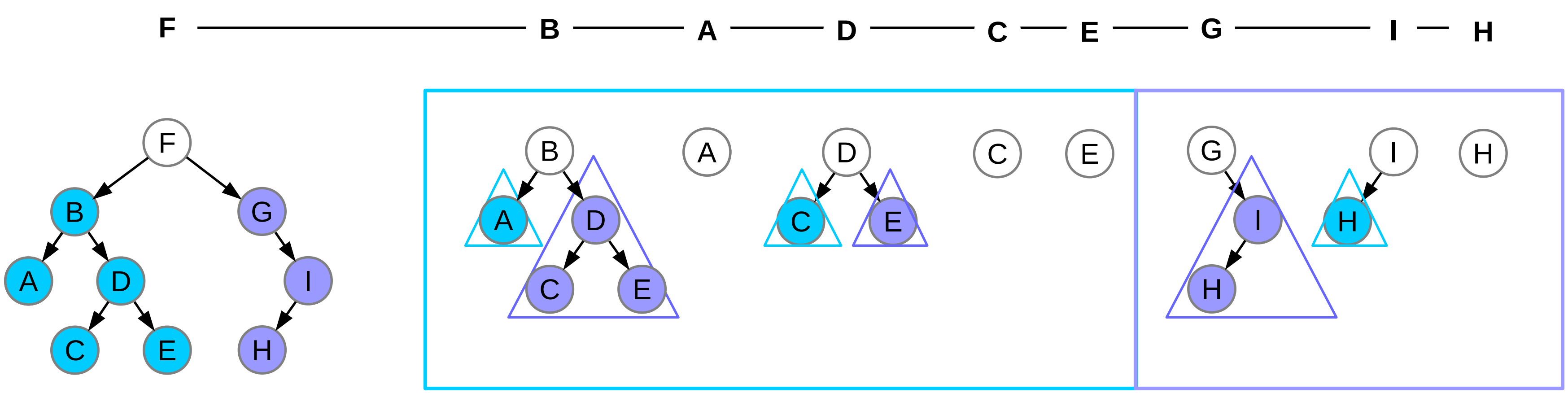
https://en.wikipedia.org/wiki/Tree_traversal

Pre-Order recursive algorithm

preorder(node)
  if (node = null)
    return
  visit(node)
  preorder(node.**left**)
  preorder(node.**right**)







https://en.wikipedia.org/wiki/Tree_traversal

Iterative Algorithms

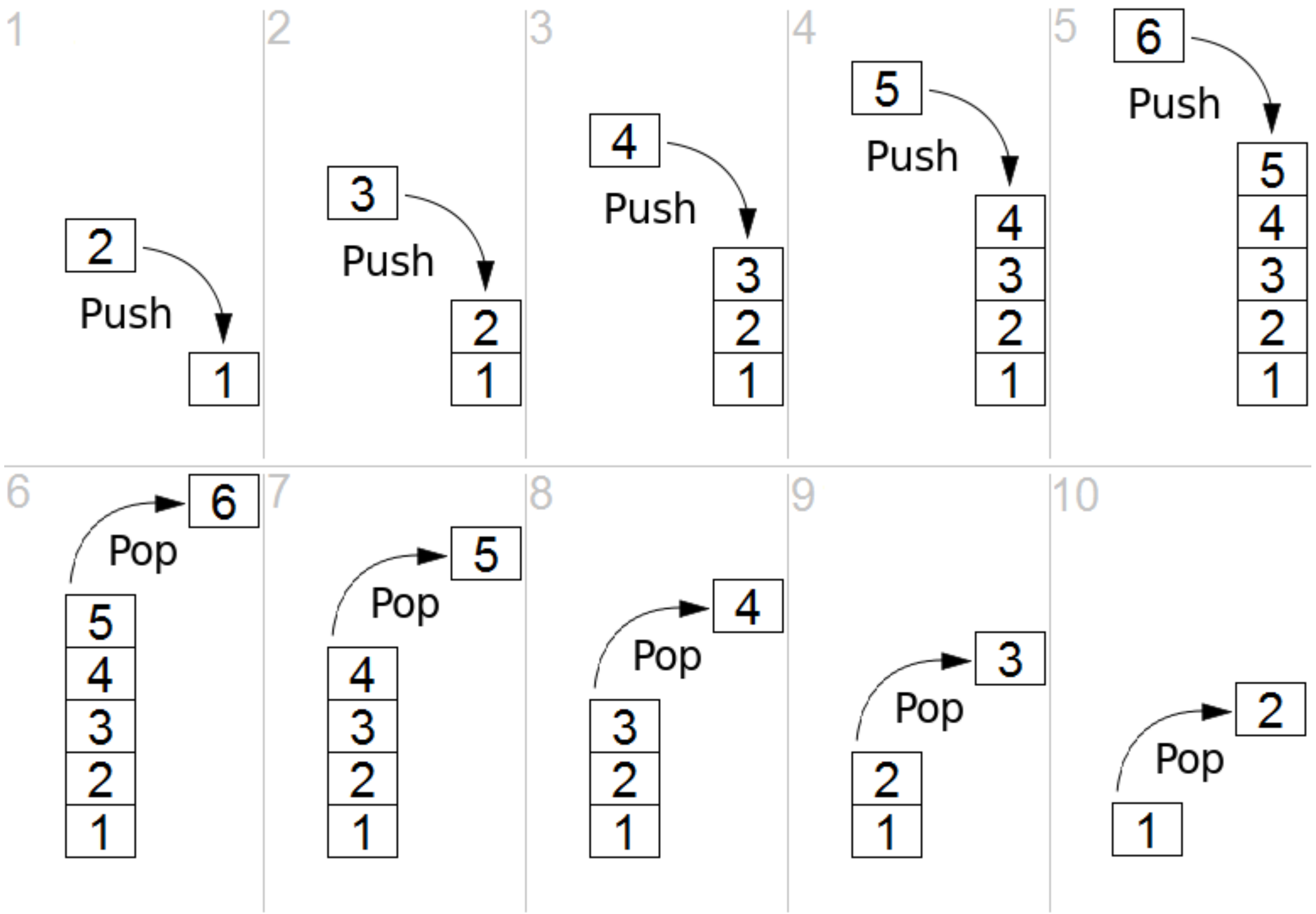
```
iterativePreorder(node)
  if (node = null)
    return
  s ← empty stack
  s.push(node)

  while (not s.isEmpty())
    node ← s.pop()
    visit(node)
    // right child is pushed first
    // so that left is processed first
    if (node.right ≠ null)
      s.push(node.right)
    if (node.left ≠ null)
      s.push(node.left)
```

https://en.wikipedia.org/wiki/Tree_traversal

```
iterativeInorder(node)
  s ← empty stack

  while (not s.isEmpty() or
           node ≠ null)
    if (node ≠ null)
      s.push(node)
      node ← node.left
    else
      node ← s.pop()
      visit(node)
      node ← node.right
```

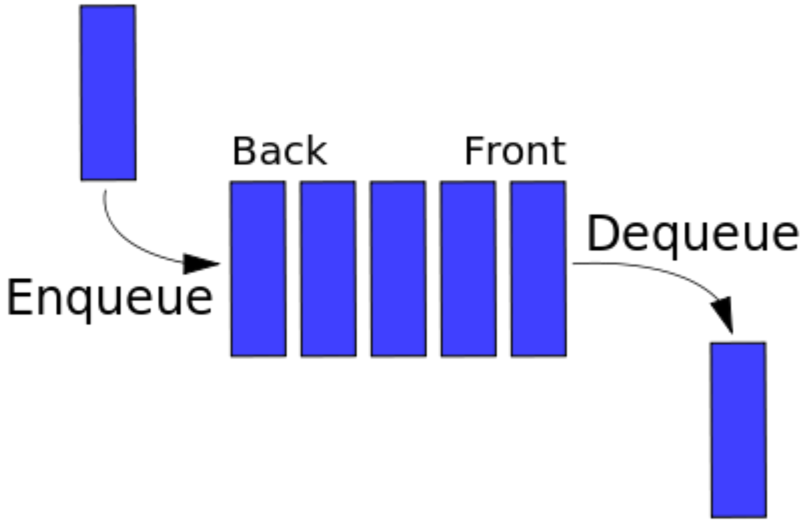
```
iterativePostorder(node)
  s ← empty stack
  lastNodeVisited ← null

  while (not s.isEmpty() or node ≠ null)
    if (node ≠ null)
      s.push(node)
      node ← node.left
    else
      peekNode ← s.peek()
      // if right child exists and traversing
      // node from left child, then move right
      if (peekNode.right ≠ null and
         lastNodeVisited ≠ peekNode.right)
        node ← peekNode.right
      else
        visit(peekNode)
        lastNodeVisited ← s.pop()
```

Stack



https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

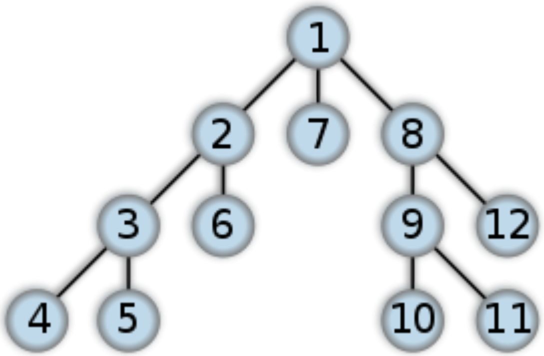
Queue



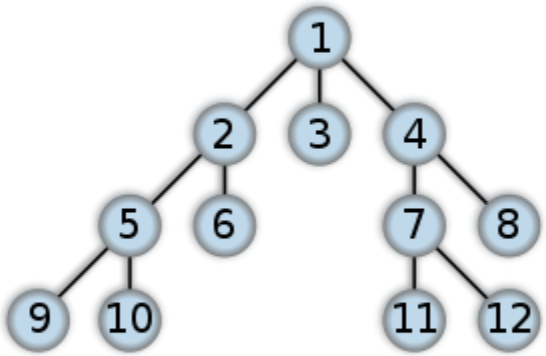
https://en.wikipedia.org/wiki/Queue_(abstract_data_type)#/media/File:Data_Queue.sv
g

Search Algorithms

DFS (Depth First Search)



BFS (Breadth First Search)



https://en.wikipedia.org/wiki/Breadth-first_search, /Depth-first_search

DFS Algorithm

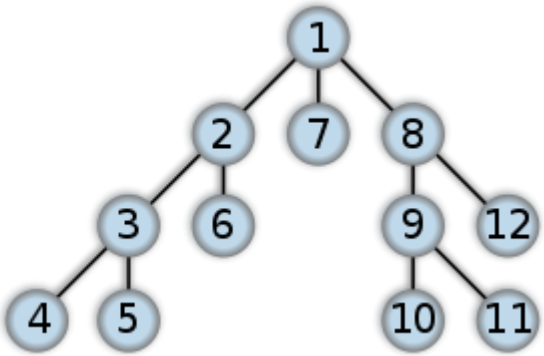
A recursive implementation of DFS:

```
procedure DFS(G,v):
    label v as discovered
    for all edges from v to w in G.adjacentEdges(v) do
        if vertex w is not labeled as discovered then
            recursively call DFS(G,w)
```

A non-recuUrsive implementation of DFS:

```
procedure DFS-iterative(G,v):
    let S be a stack
    S.push(v)
    while S is not empty
        v = S.pop()
        if v is not labeled as discovered:
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                S.push(w)
```

DFS (Depth First Search)

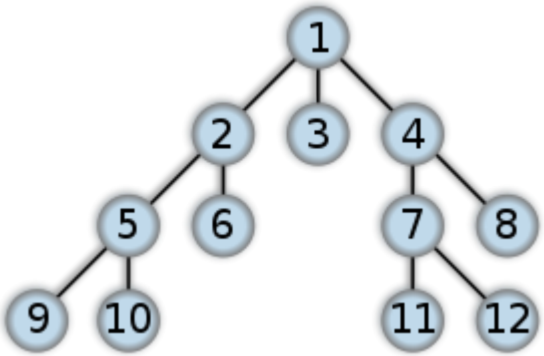


https://en.wikipedia.org/wiki/Breadth-first_search, /Depth-first_search

BFS Algorithm

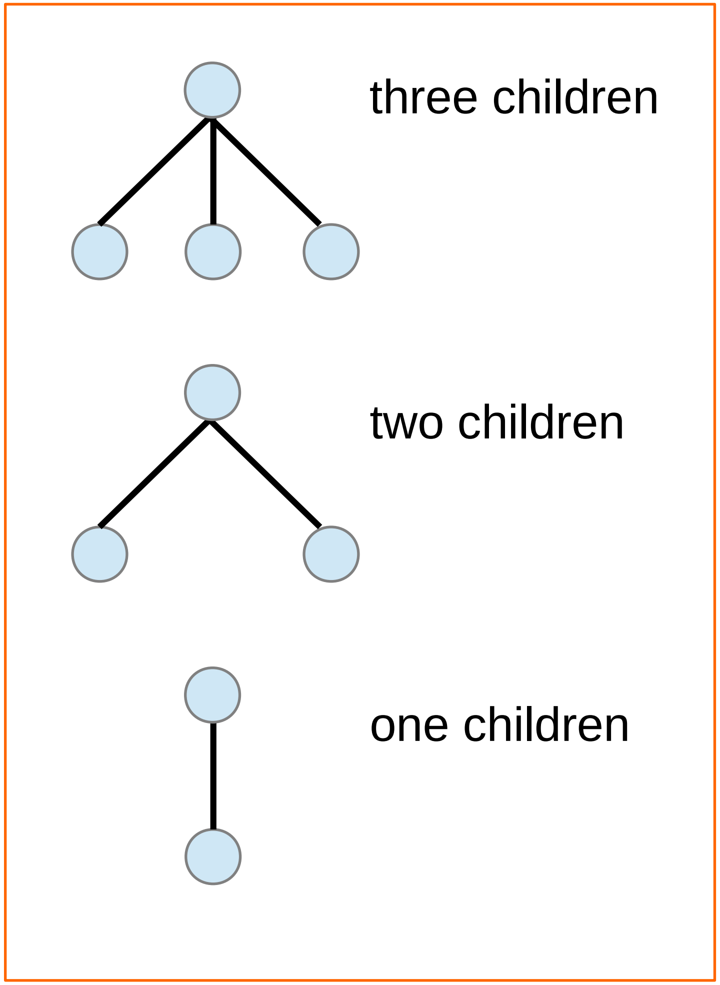
```
Breadth-First-Search(Graph, root):

    create empty set S
    create empty queue Q

    add root to S
    Q.enqueue(root)

    while Q is not empty:
        current = Q.dequeue()
        if current is the goal:
            return current
        for each node n that is adjacent to current:
            if n is not in S:
                add n to S
                n.parent = current
                Q.enqueue(n)
```

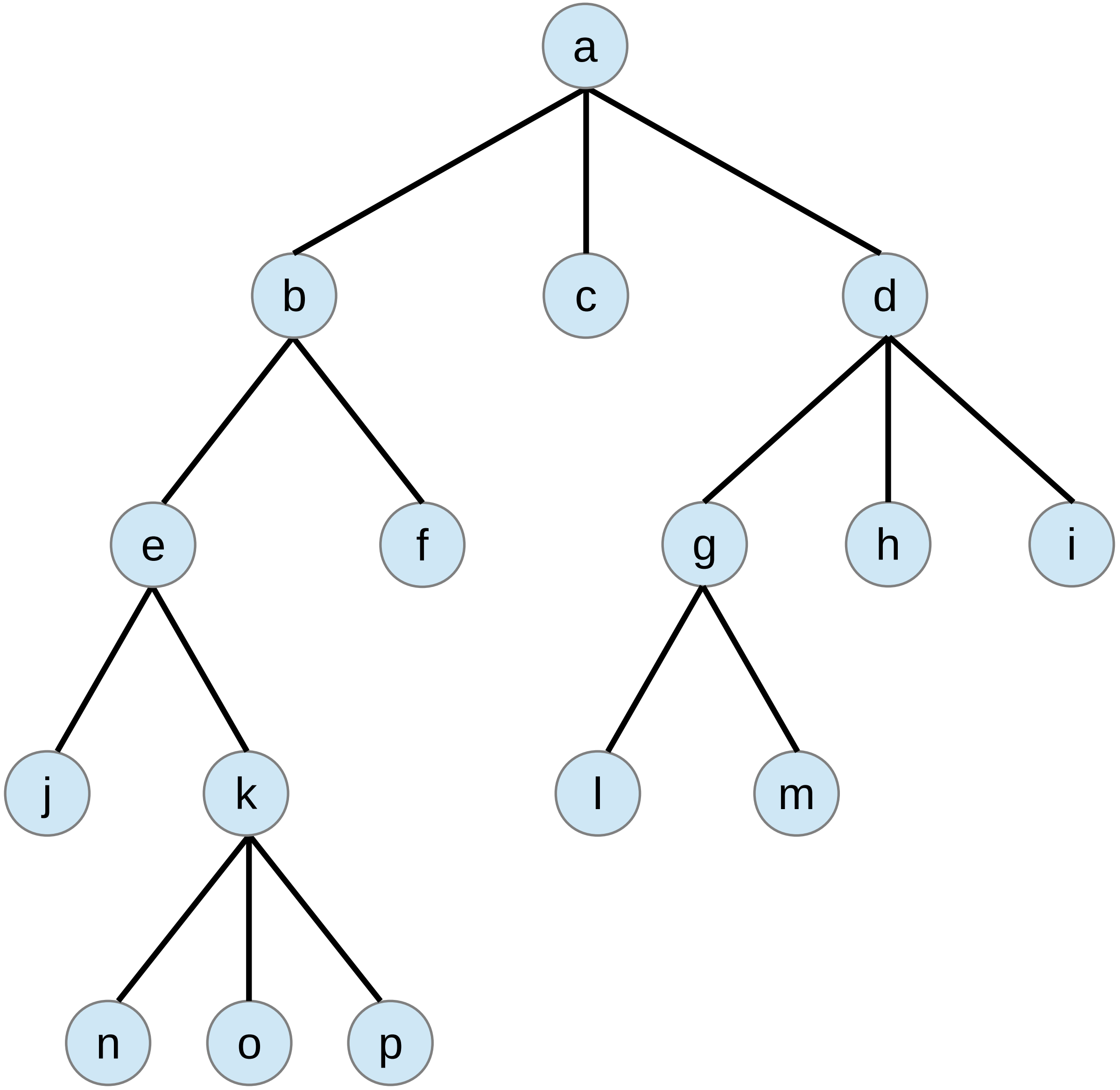
BFS (Breadth First Search)



https://en.wikipedia.org/wiki/Breadth-first_search, /Depth-first_search

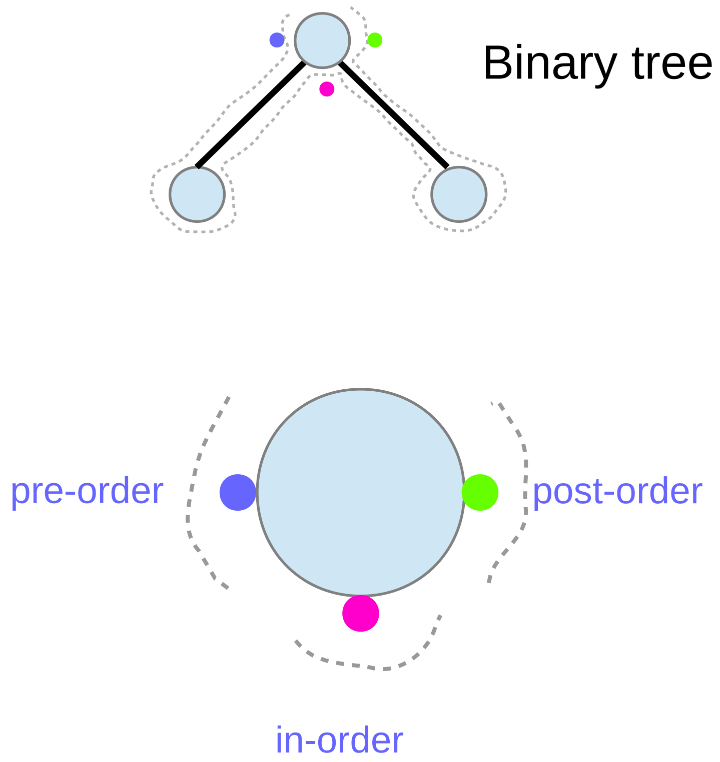
Ternary Tree

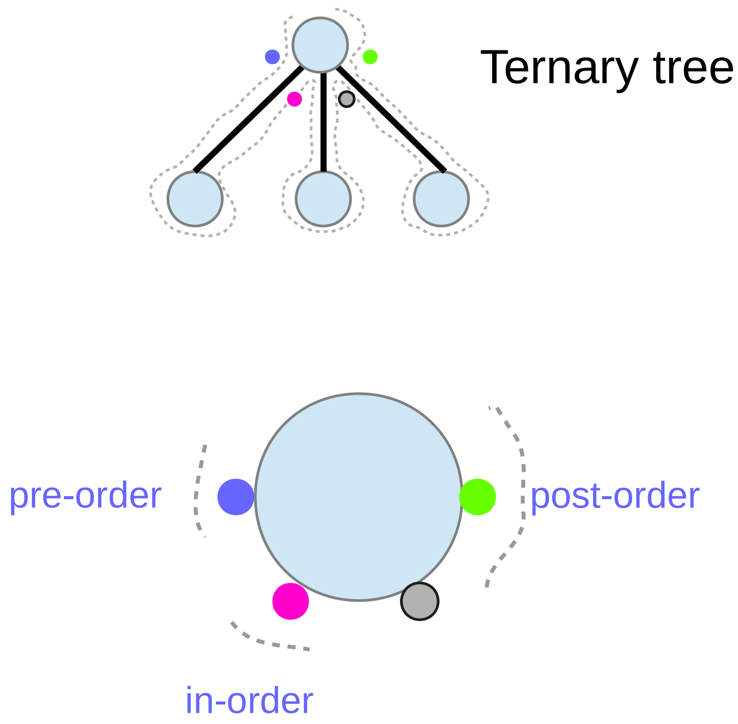




Rosen

Ternary Tree Traversal

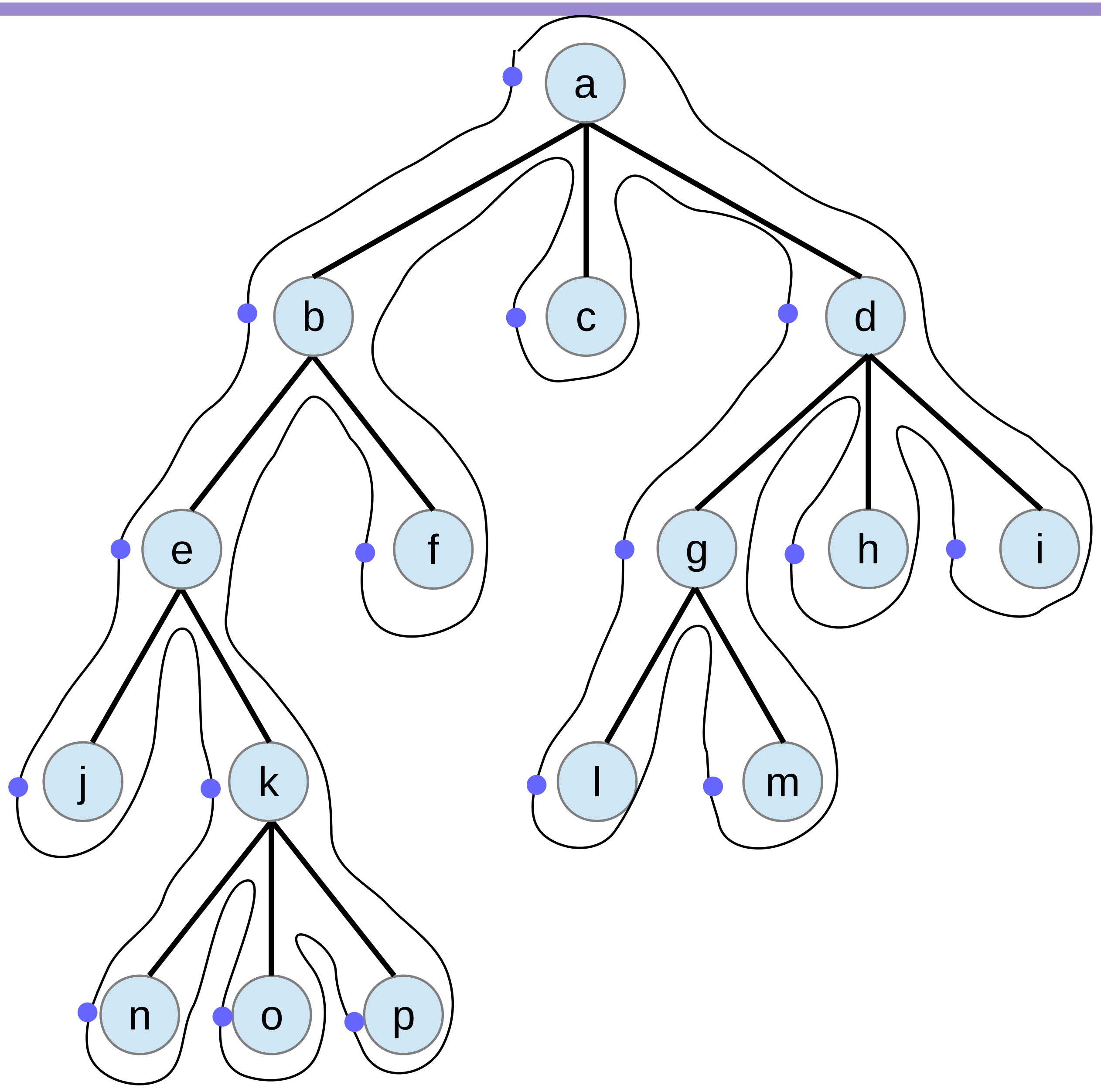




Rosen

Pre-Order Traversal on Ternary Trees

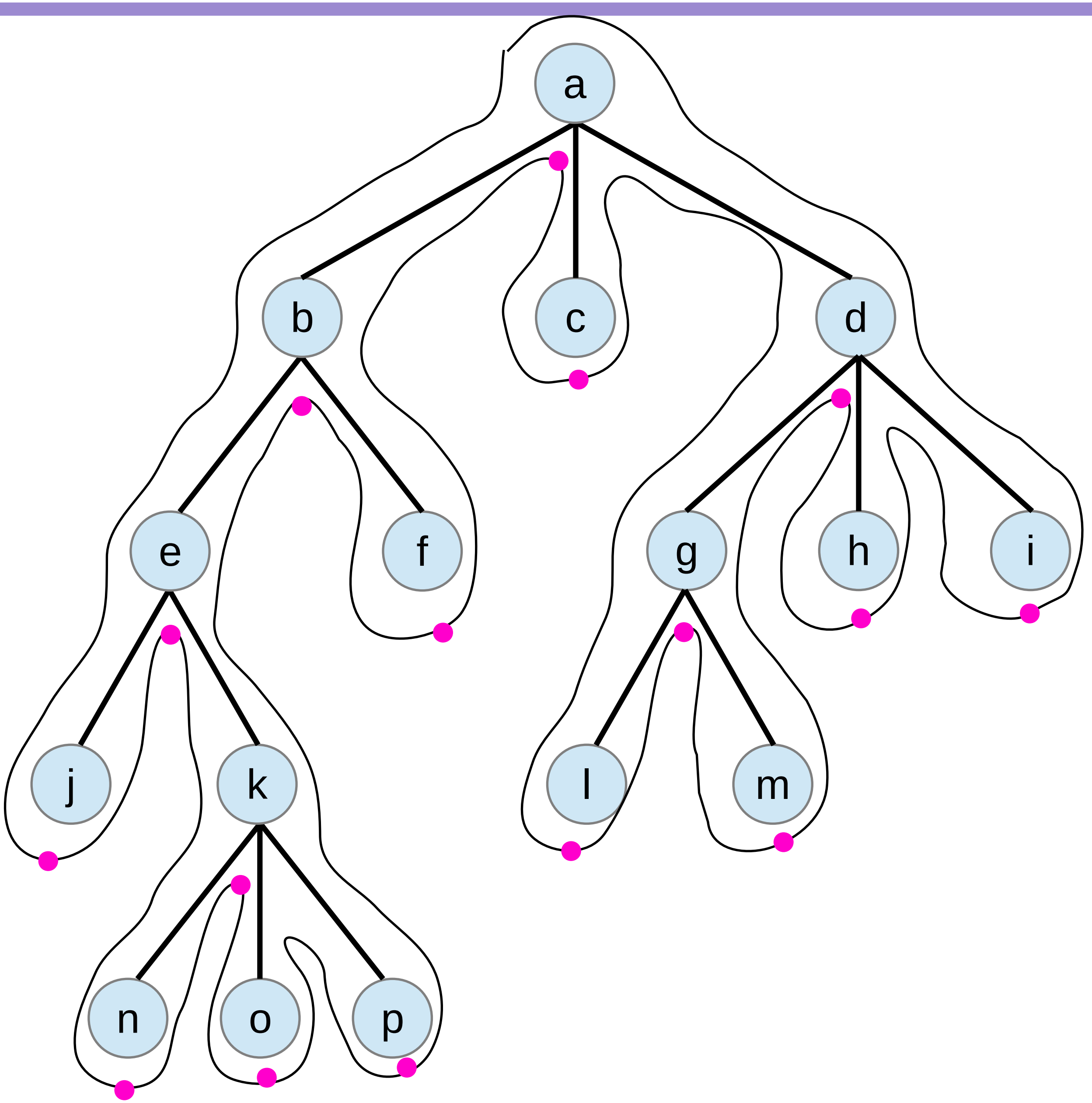
a-b-e-j-k-n-o-p-f-c-d-g-l-m-h-i



Rosen

In-Order Traversal on Ternary Trees

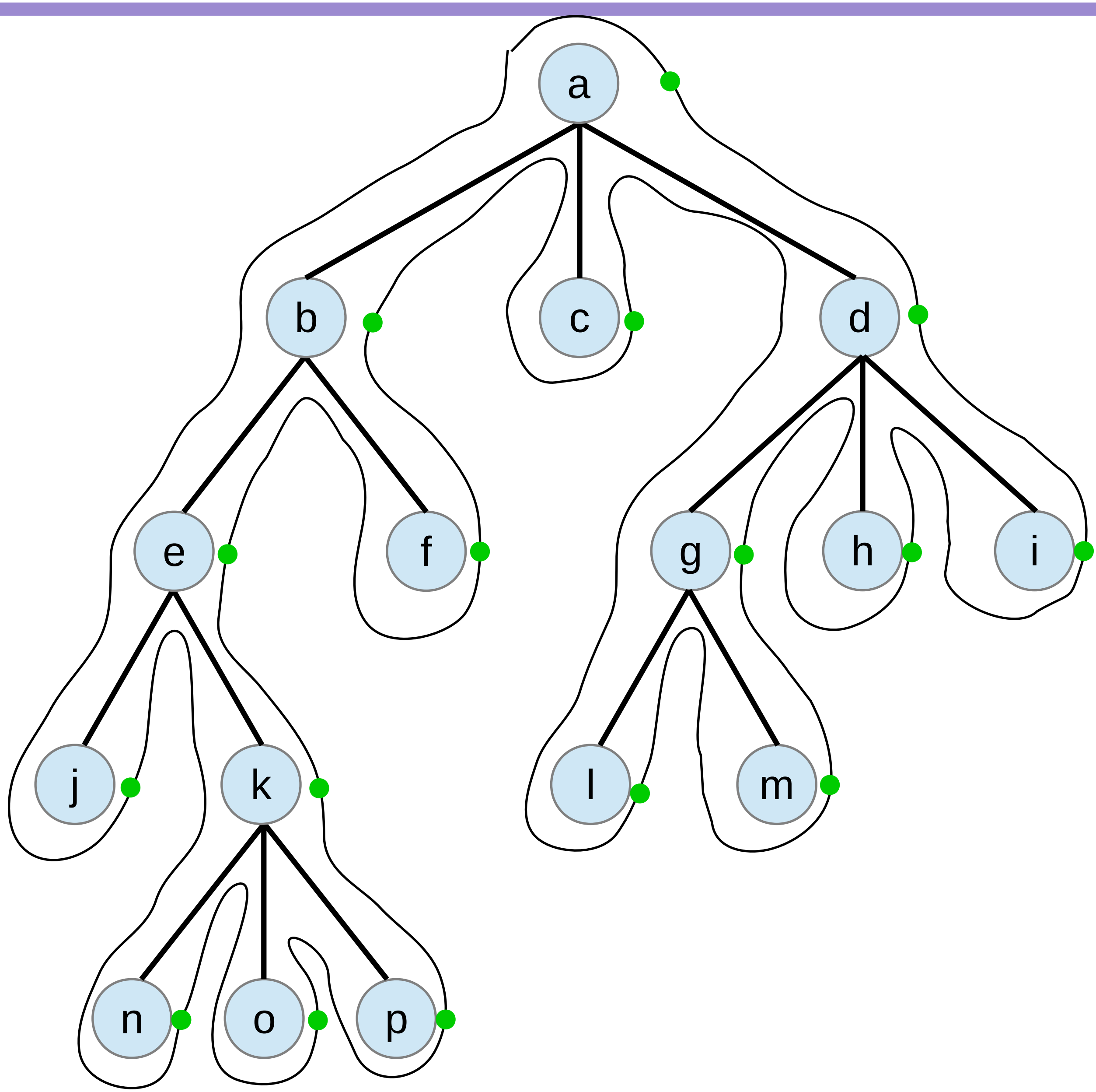
j-e-n-k-o-p-b-f-a-c-l-g-m-d-h-i



Rosen

Post-Order Traversal on Ternary Trees

j-n-o-p-k-e-f-b-c-l-m-g-h-i-d-a



Rosen

Ternary

Ternary

Etymology
Late Latin ternarius (“consisting of three things”), from terni (“three each”).
Adjective

ternary (not comparable)

- Made up of three things; treble, triadic, triple, triplex
- Arranged in groups of three
- (mathematics) To the base three [quotations ▼]
- (mathematics) Having three variables

https://en.wiktionary.org/wiki/ternary

The sequence continues with **quaternary**, **quinary**, **senary**, **septenary**, **octonary**, **nonary**, and **denary**, although most of these terms are rarely used. There's no word relating to the number eleven but there is one that relates to the number twelve: **duodenary**.

https://en.oxforddictionaries.com/explore/what-comes-after-primary-secondary-tertiary

References

[1] http://en.wikipedia.org/
[2]

Binary Search Tree (3A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice and Octave.

Binary Search Tree (1)

Binary search trees (BST),
ordered binary trees
sorted binary trees

are a particular type of **container**:
data structures that store "items"
(such as numbers, names etc.) in memory.

They allow fast **lookup**, **addition** and **removal** of items
can be used to implement either dynamic sets of items
lookup tables that allow finding an item by its **key**
(e.g., finding the phone number of a person by name).

https://en.wikipedia.org/wiki/Binary_search_tree

Binary Search Tree (2)

keep their **keys** in sorted order
lookup operations can use
the principle of **binary search**

allowing to skip searching half of the tree
each operation (**lookup**, **insertion** or **deletion**)
takes time proportional to **log n**

much better than the **linear time**
but slower than the corresponding operations
on **hash tables**.

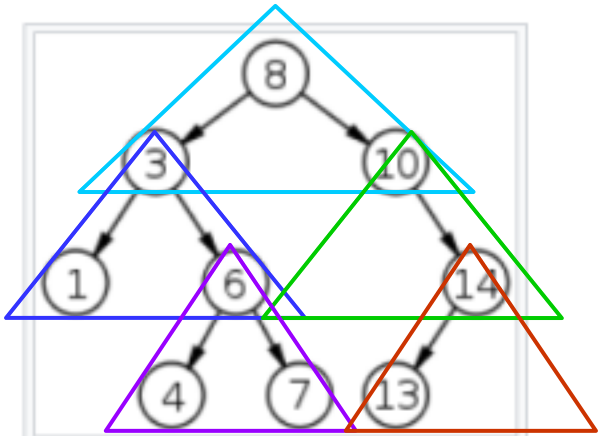
https://en.wikipedia.org/wiki/Binary_search_tree

Binary Search Tree (3)

when **looking** for a **key** in a tree
or **looking** for a **place** to insert a <u>new</u> <u>key</u>,
they <u>traverse</u> the tree from root to leaf,
making <u>comparisons</u> to keys stored in the nodes
<u>deciding</u> to continue in the **left** or **right subtrees**,
on the basis of the <u>comparison</u>**.**

https://en.wikipedia.org/wiki/Binary_search_tree

Node, Left Child, Right Child



A binary search tree of size 9 and depth 3, with 8 at the root. The leaves are not drawn.

3 < 8 < 10

1 < 3 < 6

10 < 14

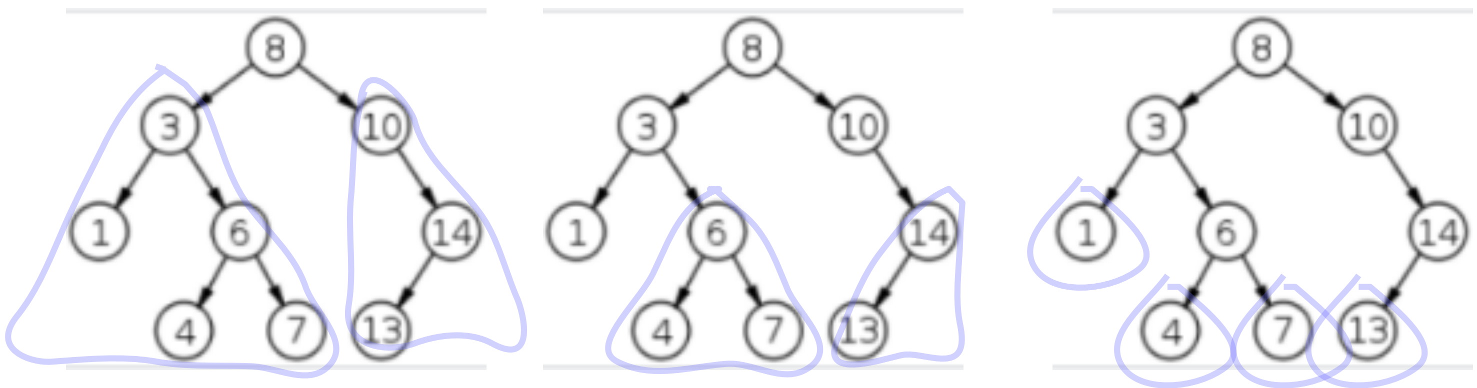
4 < 6 < 7

13 < 14

1, 3, 4, 6, 7, 8, 10, 13, 14

https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

Subtrees

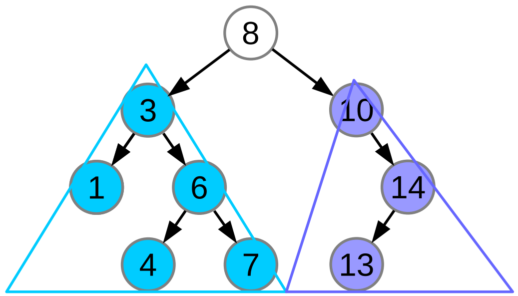


1, 3, 4, 6, 7, 8, 10, 13, 14

https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

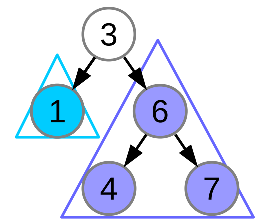
Node, Left Subtree, Right Subtree

$\boxed{1,3,4,6,7} < 8 < \boxed{10,13,14}$

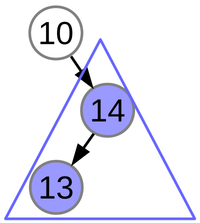


1, 3, 4, 6, 7, 8, 10, 13, 14

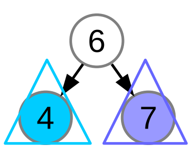
$\boxed{1} < 3 < \boxed{4,6,7}$



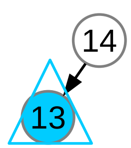
$10 < \boxed{13,14}$



$\boxed{4} < 6 < \boxed{7}$

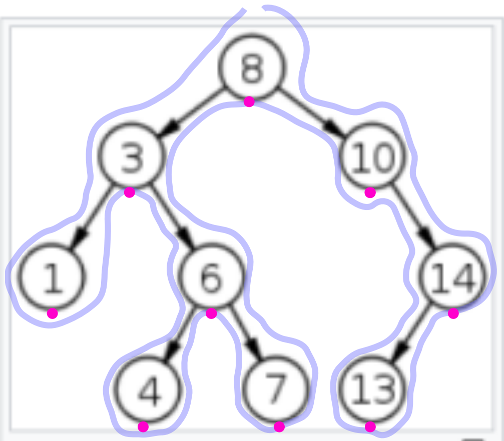


$\boxed{13} < 14$



https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

In-Order Traversal



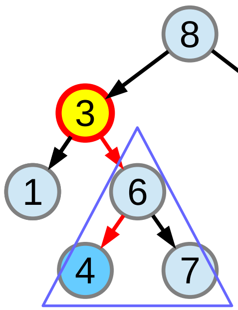
A binary search tree of size 9 and depth 3, with 8 at the root. The leaves are not drawn.

1, 3, 4, 6, 7, 8, 10, 13, 14

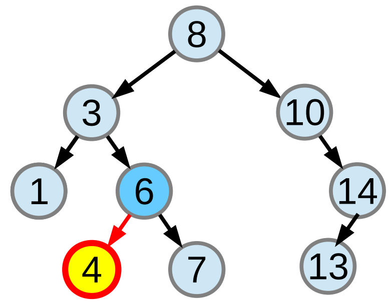
https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

Successor Examples (1)

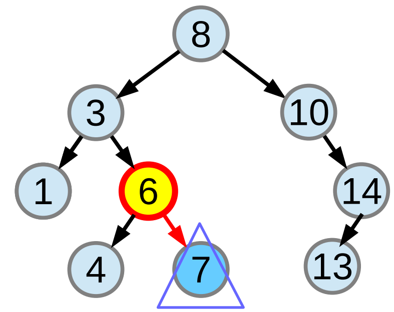
3-4



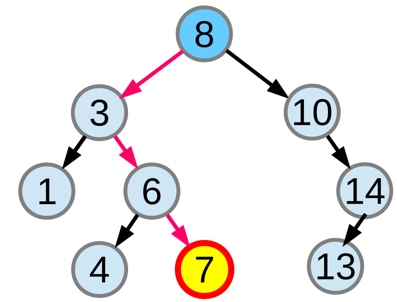
4-6



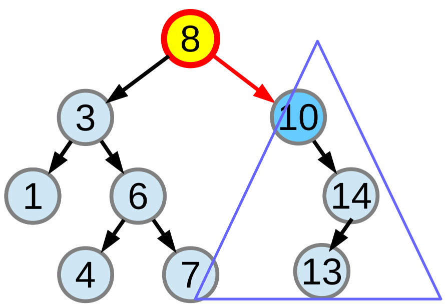
6-7



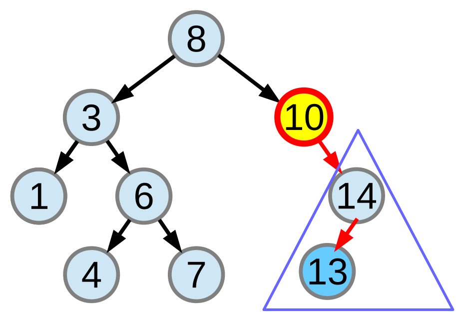
7-8



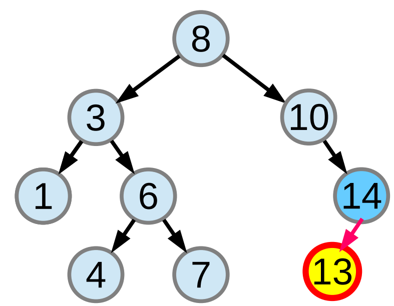
8-10



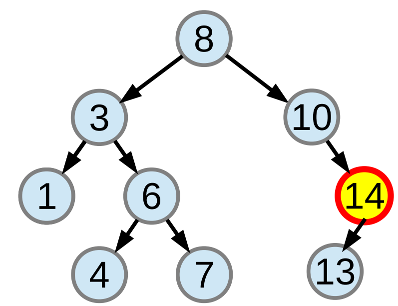
10-13



13-14

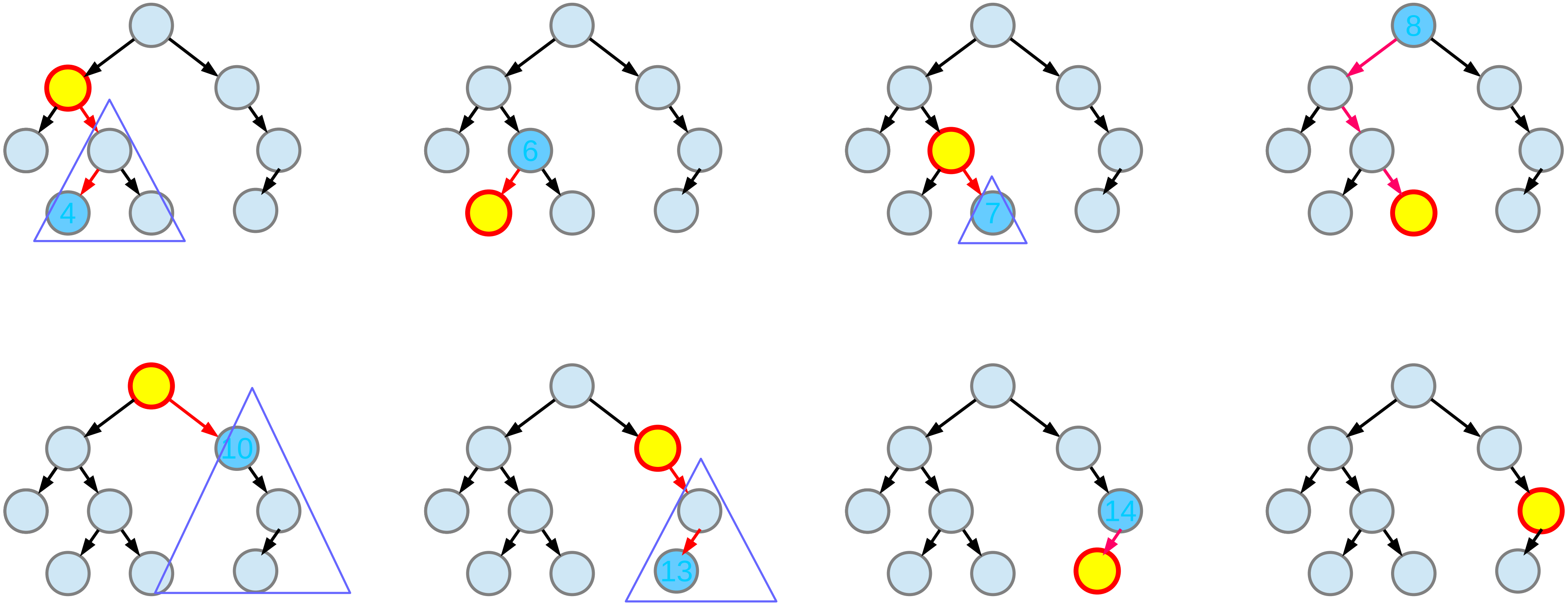


14-



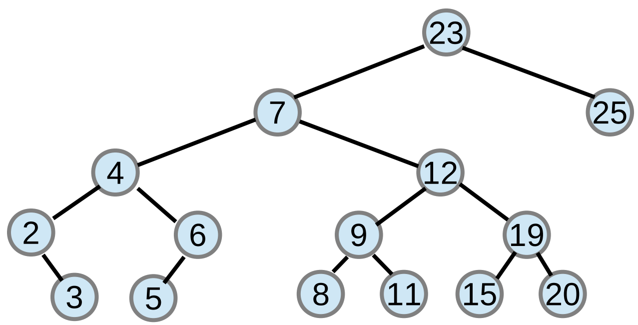
https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

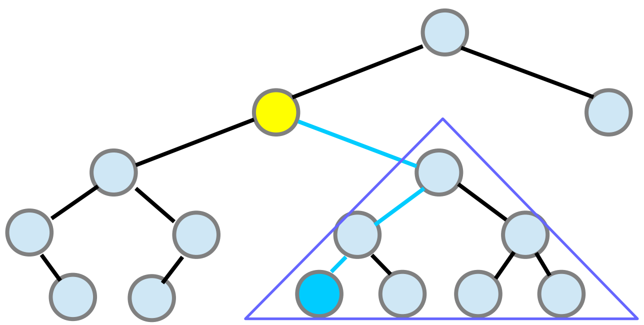
Successor Examples (2)

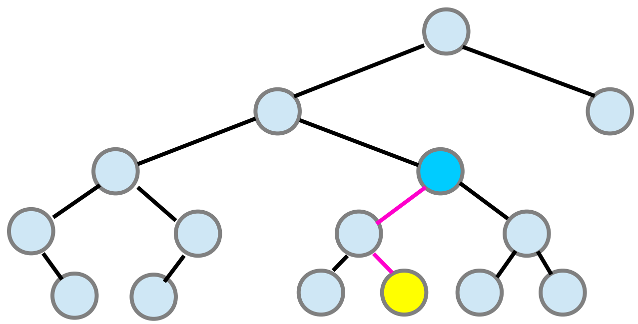


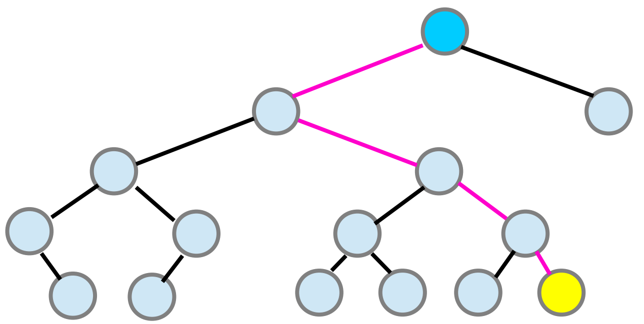
https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

Successor Examples (3)



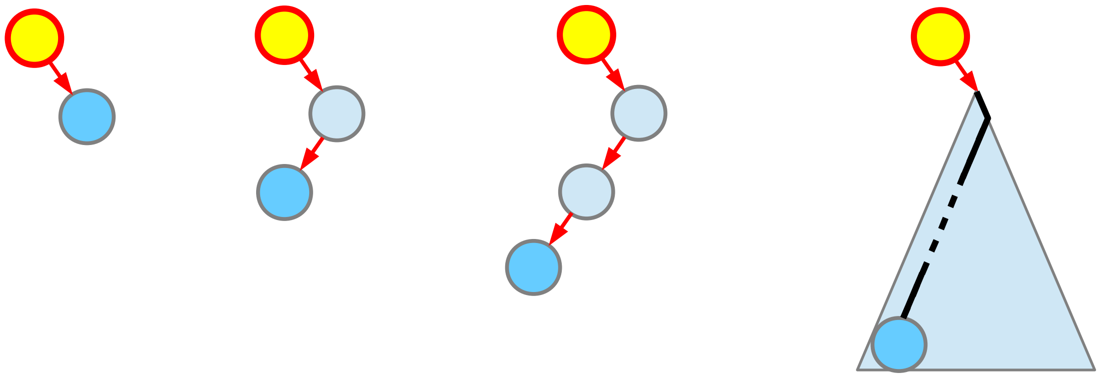




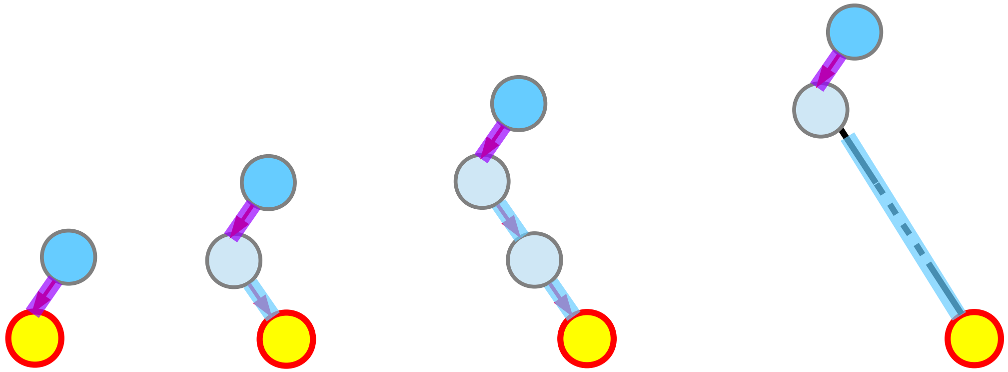


https://www.cs.rochester.edu/~gildea/csc282/slides/C12-bst.pdf

Successor Cases



If the right child exists, then the minimum in the **right** subtree
– the **leftmost** node

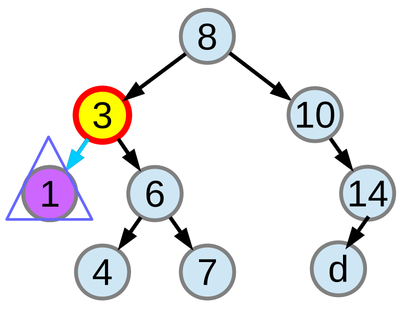


the **parent** of the farthest node that can be reached by following only **right** edges **backward**.

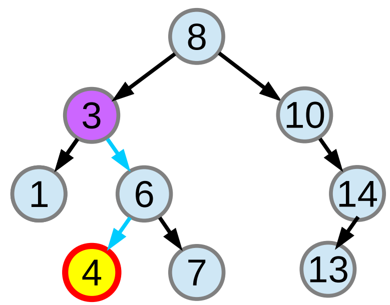
https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

Predecessor Examples (1)

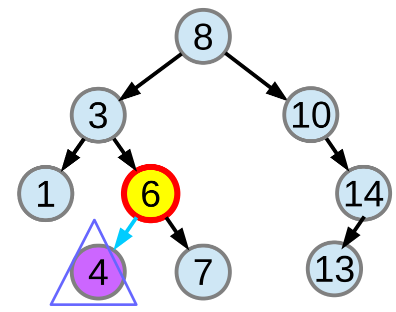
1-3



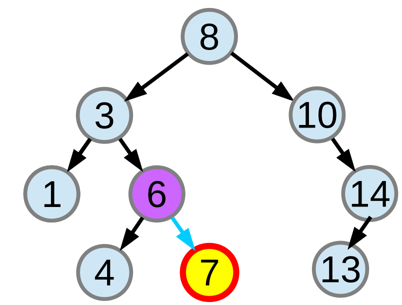
3-4



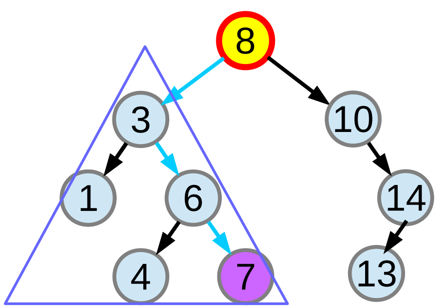
4-6



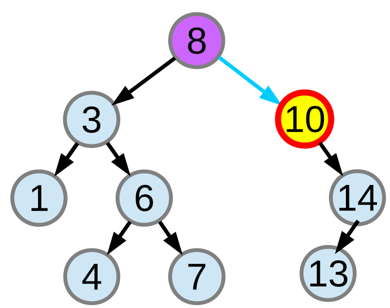
6-7



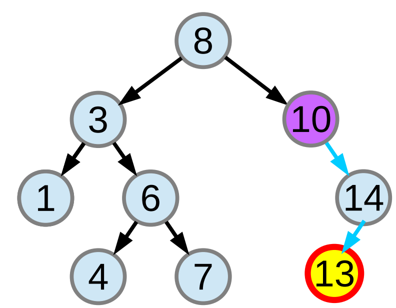
7-8



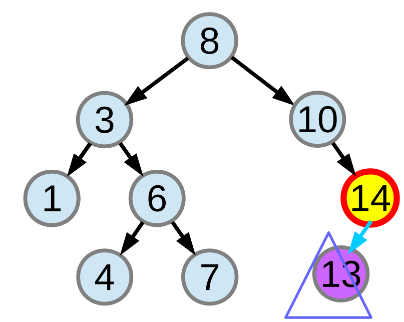
8-10



10-13

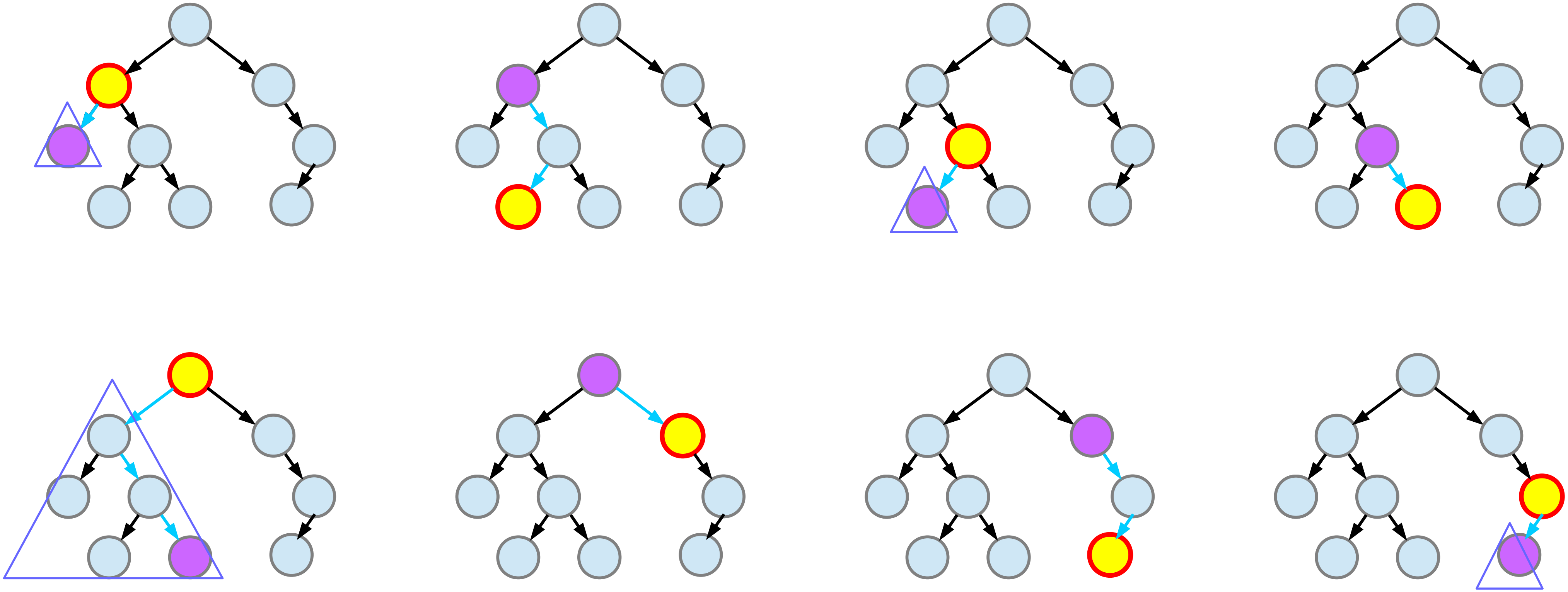


13-14



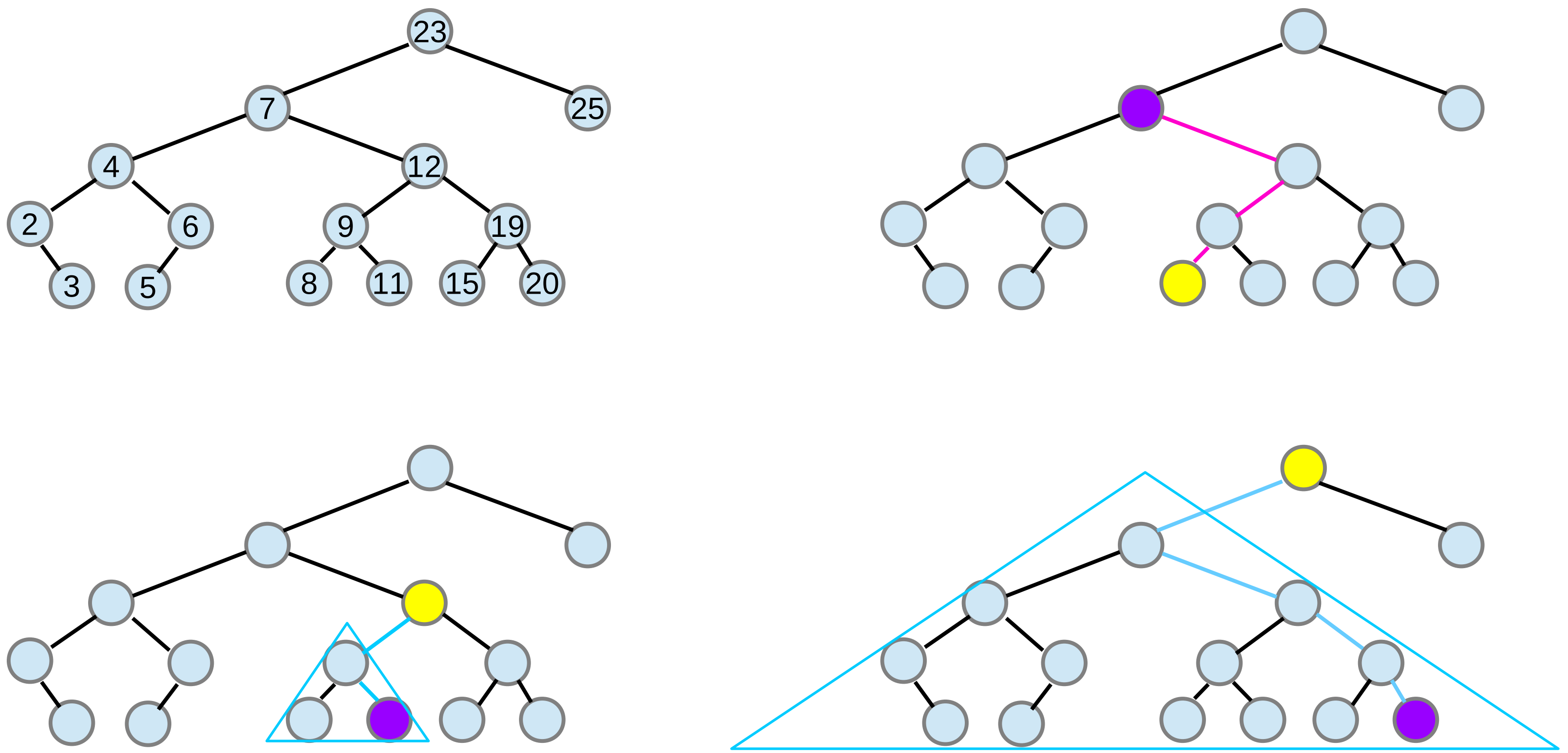
https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

Predecessor Examples (2)



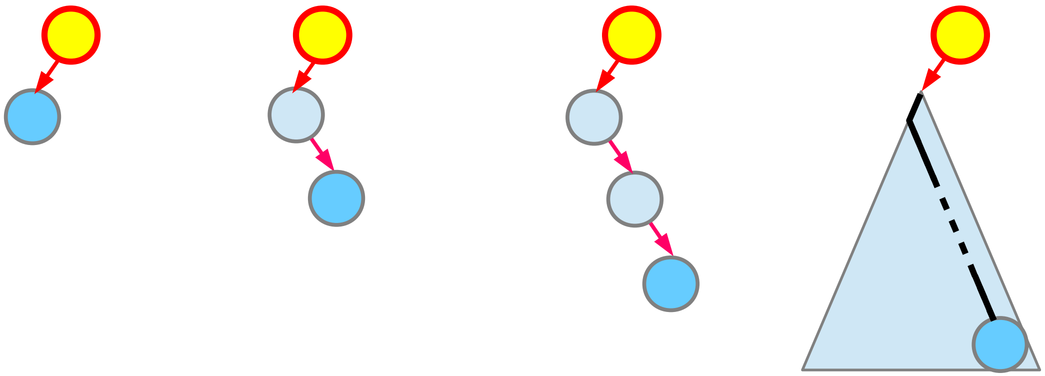
https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

Predecessor Examples (3)

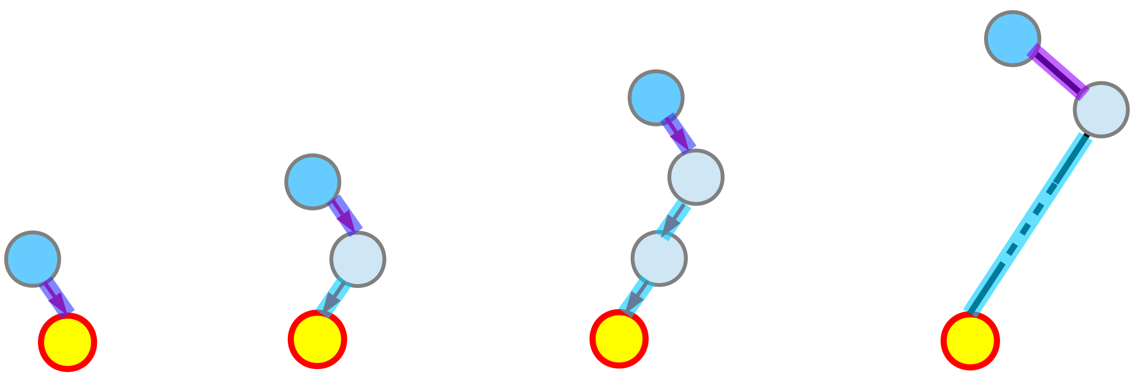


https://www.cs.rochester.edu/~gildea/csc282/slides/C12-bst.pdf

Predecessor Cases



If the left child exists, then the maximum
in the **left** subtree
– the **rightmost** node

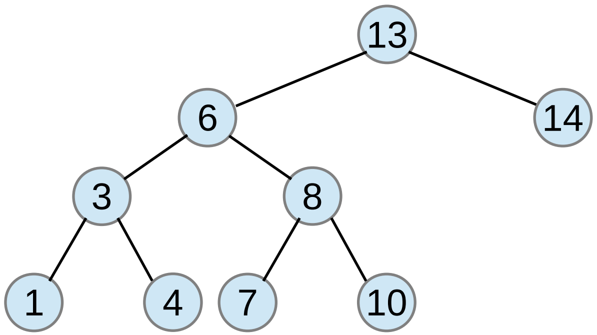


the **parent** of the farthest node that can be reached by following only **left** edges **backward**.

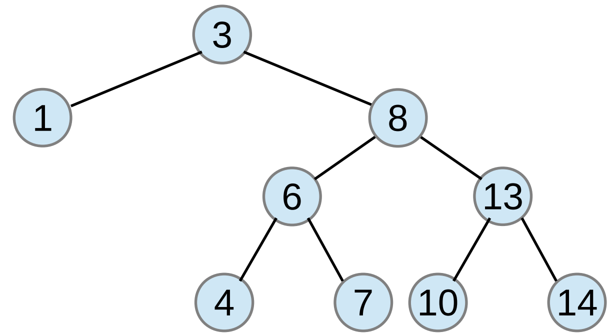
https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

Different BST's with the same data

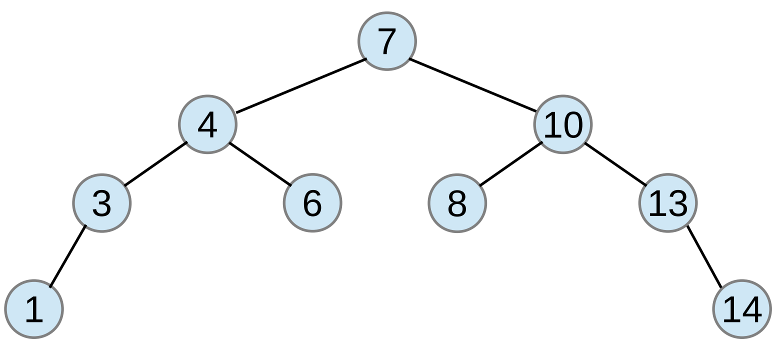
1, 3, 4, 6, 7, 8, 10, 13, 14



1, 3, 4, 6, 7, 8, 10, 13, 14

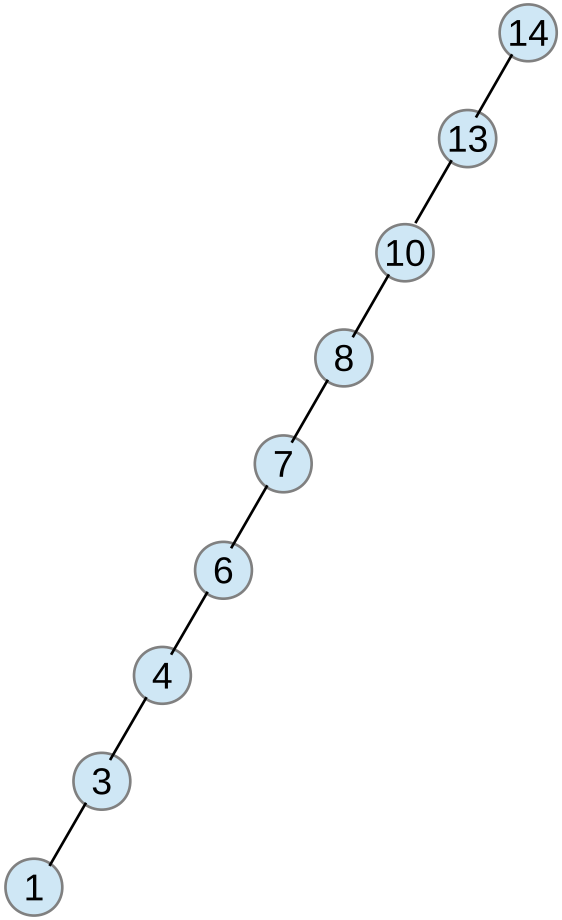


1, 3, 4, 6, 7, 8, 10, 13, 14

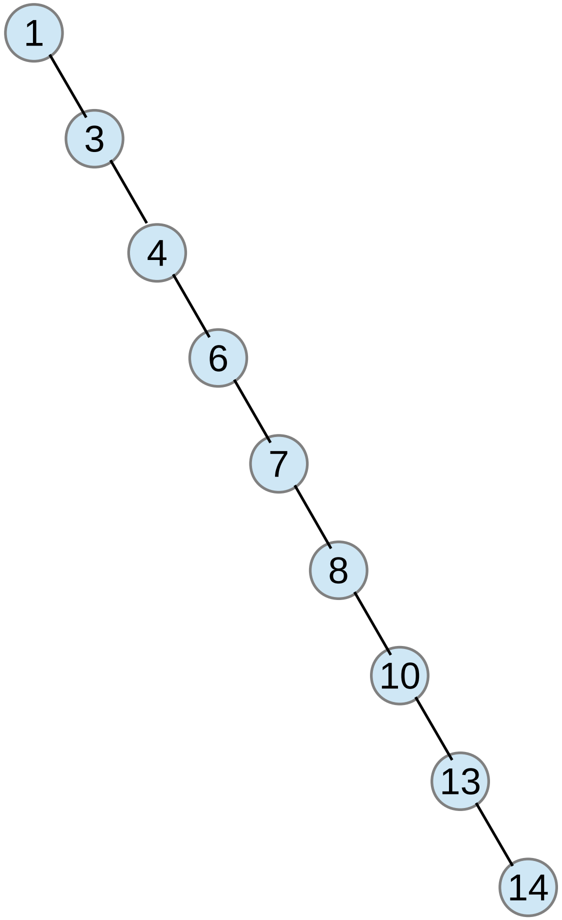


Unbalanced BSTs

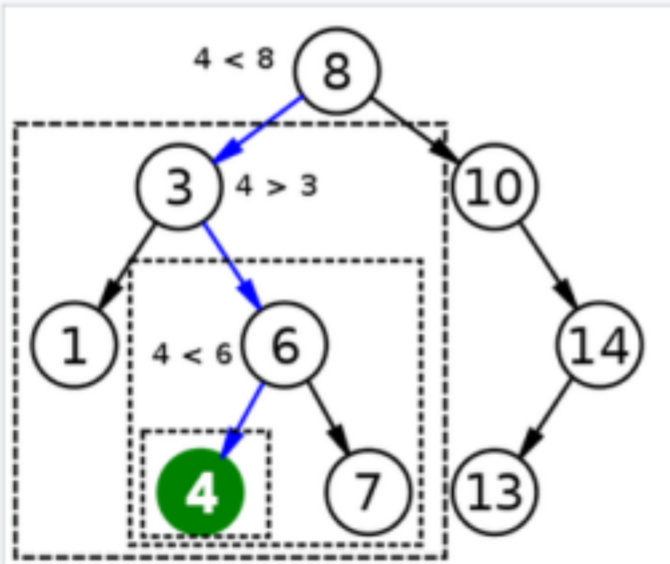
1, 3, 4, 6, 7, 8, 10, 13, 14



1, 3, 4, 6, 7, 8, 10, 13, 14



Binary Search on a Binary Search Tree



Binary search trees are searched using an algorithm similar to binary search.

$\boxed{1,\ 3,\ 4,\ 6,\ 7,}\ 8,\ 10,\ 13,\ 14$

$1,\ 3,\ \boxed{4,\ 6,\ 7,}\ 8,\ 10,\ 13,\ 14$

$1,\ 3,\ \boxed{4,}\ 6,\ 7,\ 8,\ 10,\ 13,\ 14$

https://en.wikipedia.org/wiki/Binary_search_algorithm

Insertion

Insertion begins as a **search** would begin;
if the key is not equal to that of the **root**,
we search the **left** or **right** subtrees as before.

at an **leaf node**, **add** the new key-value pair
as its **right** or **left child**,
depending on the node's **key**.

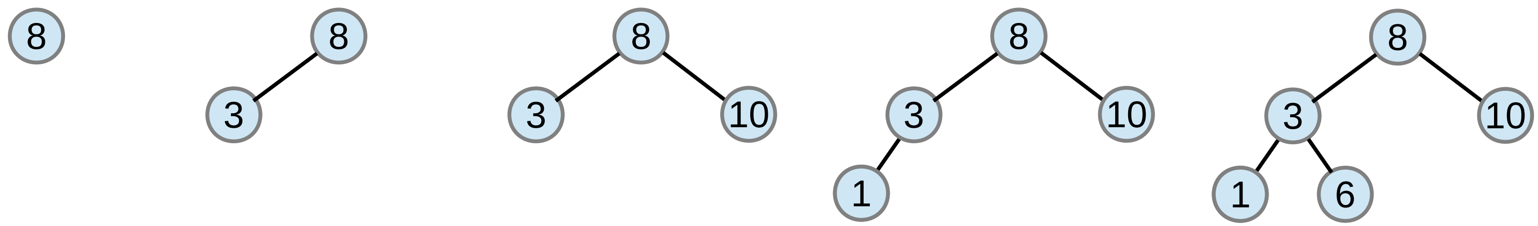
first <u>examine</u> the **root**
and <u>recursively</u> <u>insert</u> the new node
to the **left** subtree if <u>its</u> key is <u>less</u> than that of the **root**,
or the **right** subtree if its key is <u>greater</u> than or equal to the **root**.

https://en.wikipedia.org/wiki/Binary_search_tree

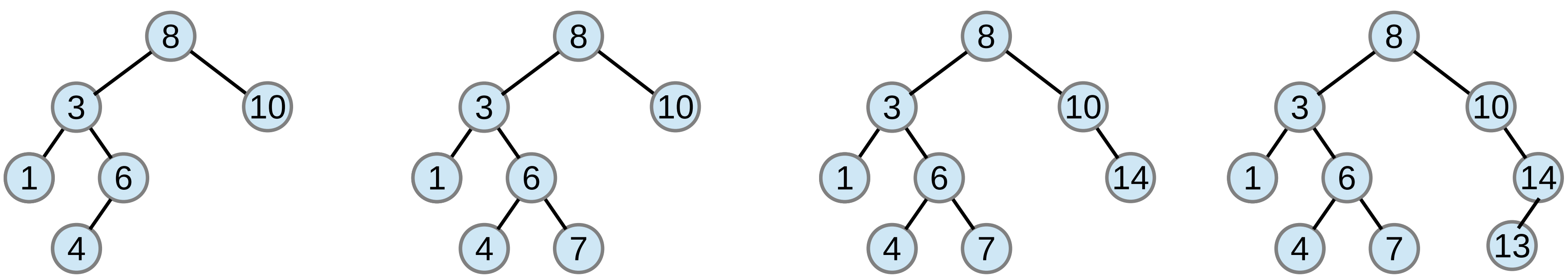
Insertion Example (1)

Insert(8→3→10→1→6→4→7→14→13)

insert(8) **insert**(3) **insert**(10) **insert**(1) **insert**(6)



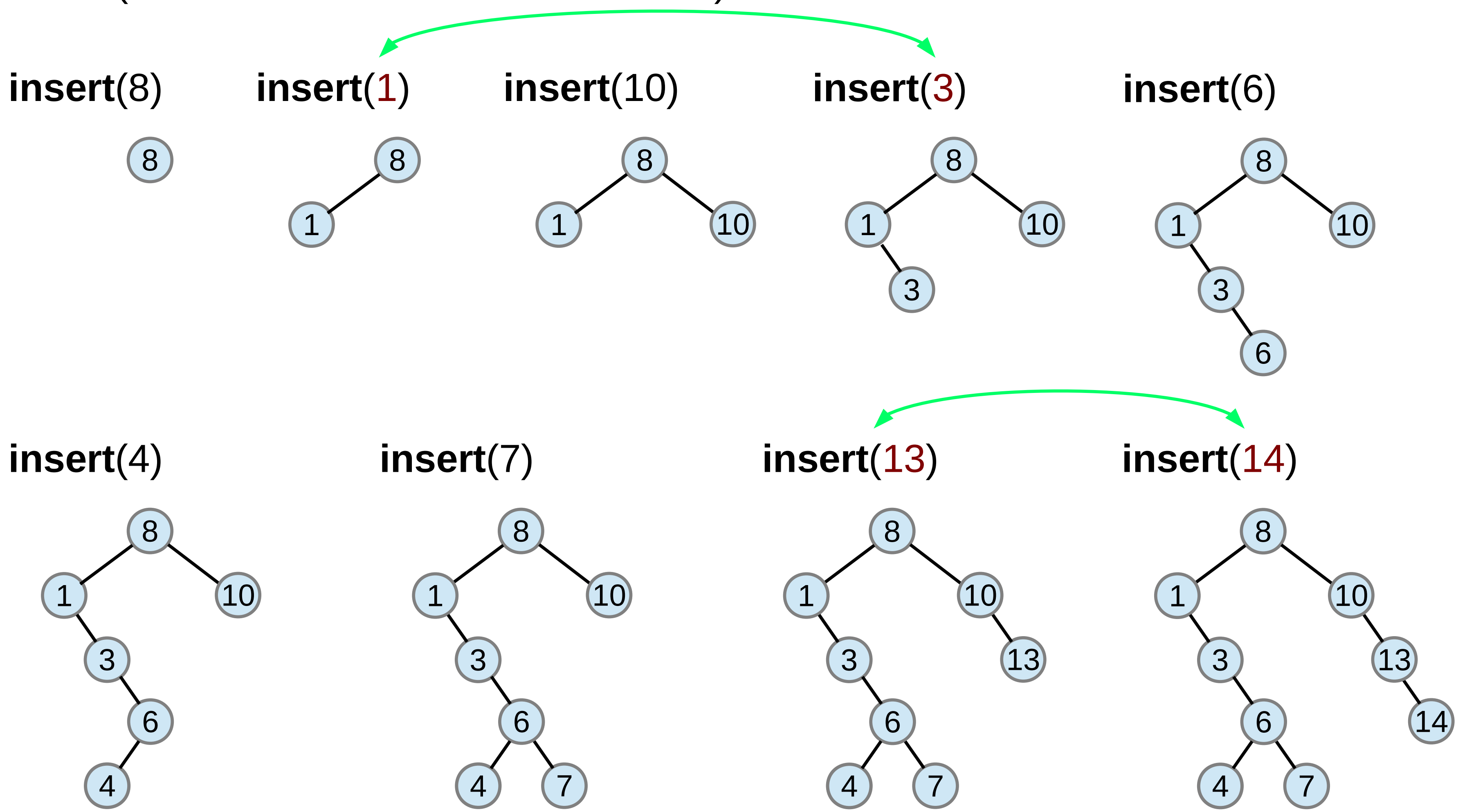
insert(4) **insert**(7) **insert**(14) **insert**(13)



Insertion Example (2)

Insert(8 → 3 → 10 → 1 → 6 → 4 → 7 → 14 → 13)

Insert(8 → 1 → 10 → 3 → 6 → 4 → 7 → 13 → 14)



Deletion

1. Deleting a **node** with no **children**:
   simply remove the node from the tree.

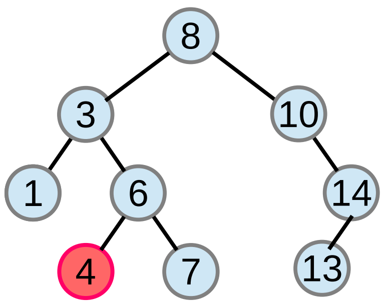
2. Deleting a **node** with one **child**:
   remove the node and replace it with its child.

3. Deleting a **node** with two **children**:
   call the **node** to be deleted D.
   Do not delete D.
   Instead, choose either its in-order **predecessor node** 3(a)
   or its in-order **successor node** as replacement node E. 3(b)
   Copy the user values of E to D
   If E does not have a **child**
      simply remove E from its previous parent G.
   If E has a **child**, say F, it is a right child.
      Replace E with F at E's parent.

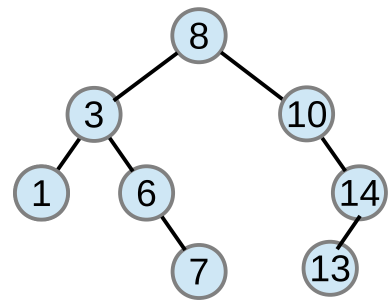
https://en.wikipedia.org/wiki/Binary_search_tree

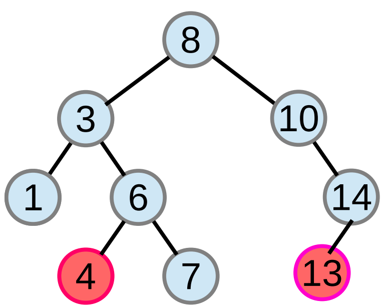
Deletion – Case 1

1. Deleting a **node** with no **children**:
   simply remove the node from the tree.

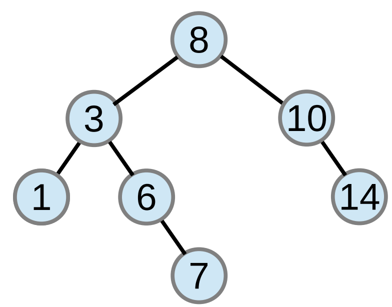


delete(4)





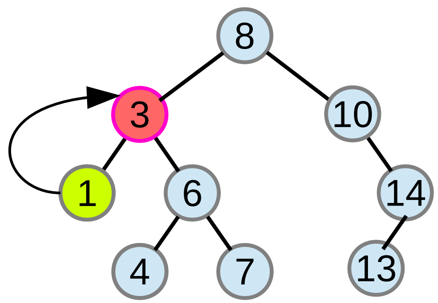
delete(4)
delete(13)



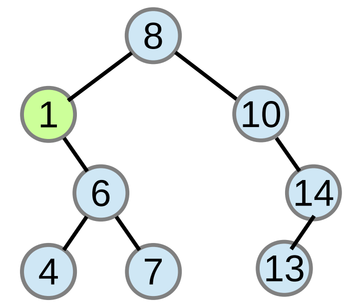
https://en.wikipedia.org/wiki/Binary_search_tree

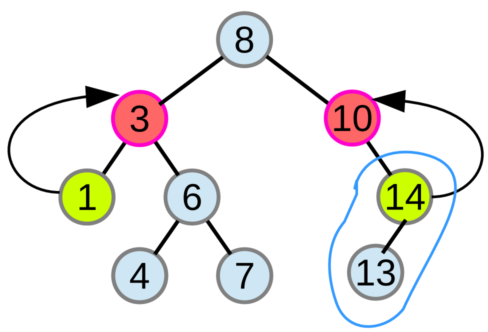
Deletion – Case 2

2. <u>Deleting</u> a **node** with <u>one</u> **child**:
remove the node and replace it with its child.

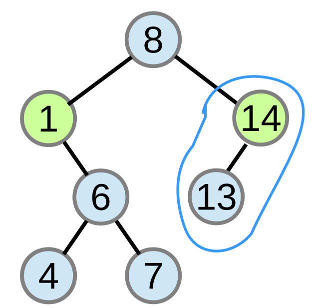


delete(3)





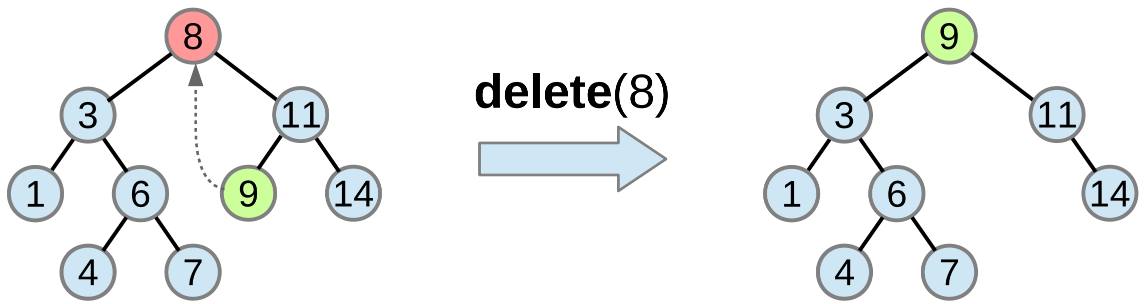
delete(3)
delete(10)



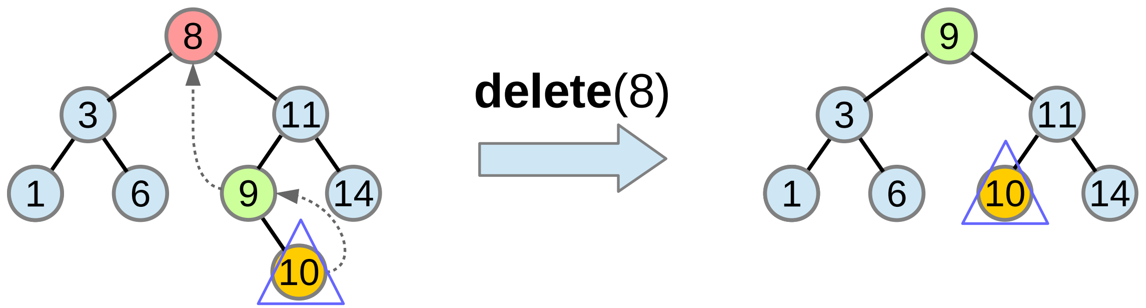
https://en.wikipedia.org/wiki/Binary_search_tree

Deletion – Case 3 : using a successor

3. Deleting a **node** with two **children**:
   call the **node** to be deleted **D**.
   its in-order **successor node** as **E**. Copy **E** to **D**



Leftmost
E has no **child**
simply remove **E**
from its parent **G**.

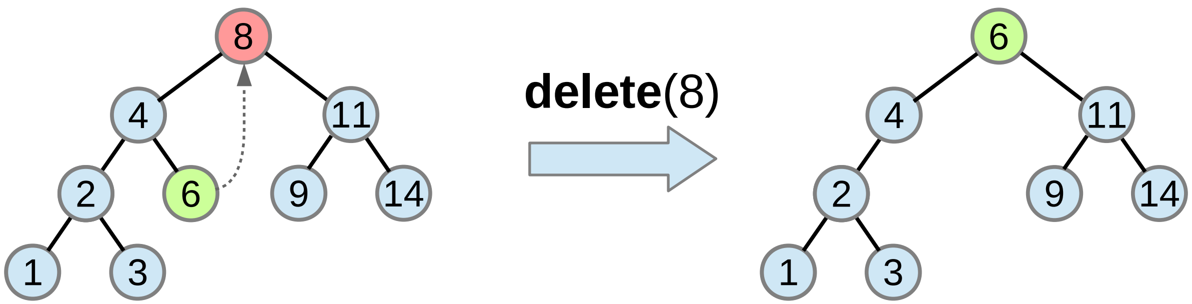


Leftmost
E has a child **F**
it is a **right** child
replace **E** with **F**
at **E**'s parent.

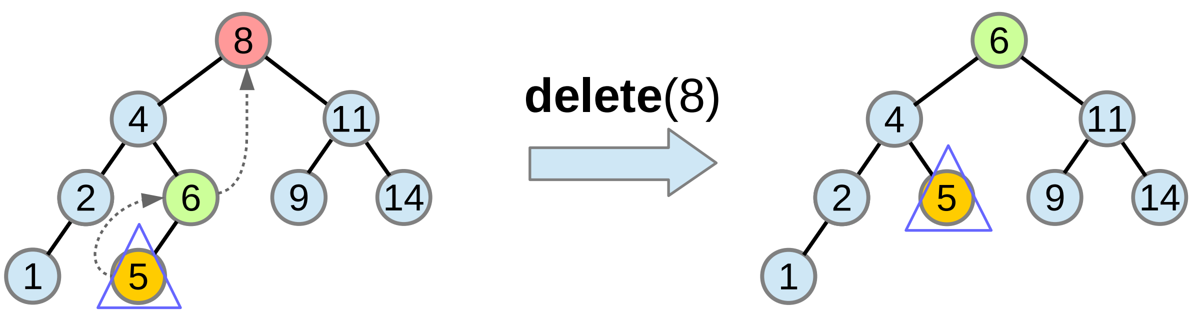
https://en.wikipedia.org/wiki/Binary_search_tree

Deletion – Case 3 : using a predecessor

3. Deleting a **node** with two **children**:
call the **node** to be deleted **D**.
its in-order **predecessor node** as **E**. Copy **E** to **D**



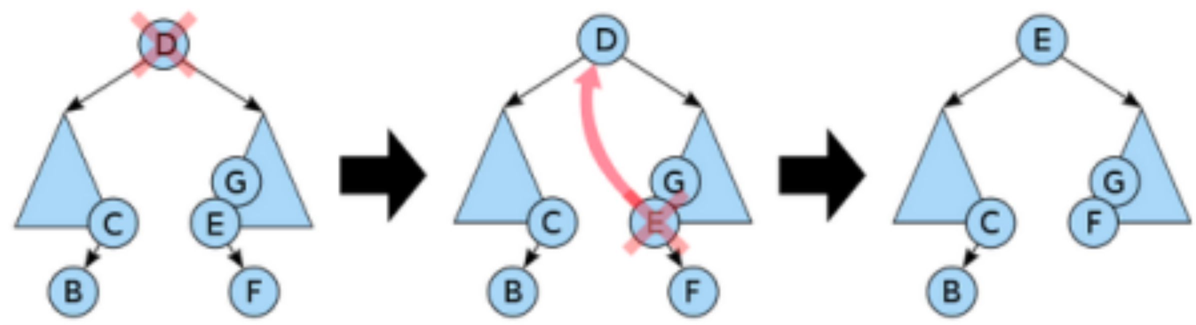
Rightmost
E has no **child**
simply remove **E**
from its parent **G**.



Rightmost
E has a child **F**
it is a **left** child
replace **E** with **F**
at **E**'s parent.

https://en.wikipedia.org/wiki/Binary_search_tree

Deletion



Deleting a **node** with **two children** from a binary search tree.
First the **leftmost** node in the **right** subtree,
the in-order **successor E**, is identified.
Its value is **copied** into the **node D** being deleted.
The in-order successor can then be easily deleted
because it has <u>at most</u> **one child**.
The same method works symmetrically
using the in-order **predecessor** C.

https://en.wikipedia.org/wiki/Binary_search_tree

References

[1] http://en.wikipedia.org/
[2]

Finite State Machine (1A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

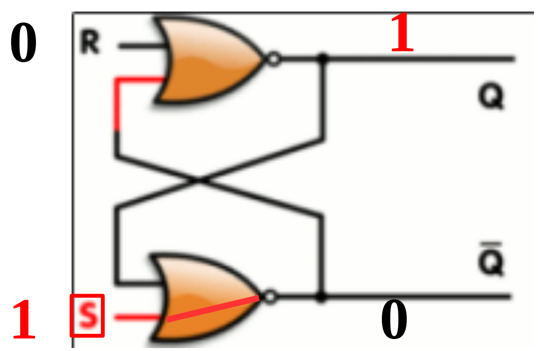
This document was produced by using LibreOffice and Octave.

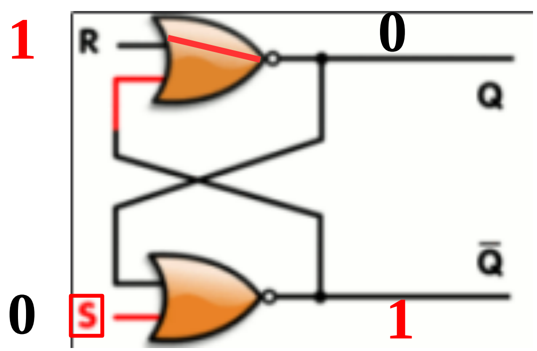
FSM and Digital Logic Circuits

- Latch
- D FlipFlop
- Registers
- Timing
- Mealy machine
- Moore machine
- Traffic Lights Examples

https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

NOR-based SR Latch – SET / RESET





SET	S=**1** R=0

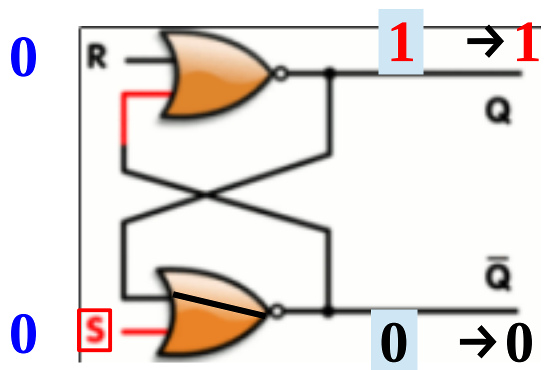
Q=1
$\overline{Q}$=0

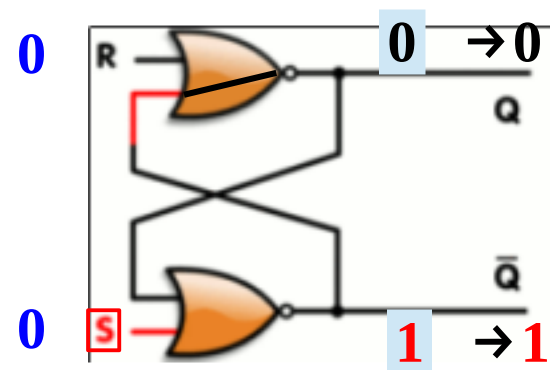
RESET	S=0 R=**1**

Q=0
$\overline{Q}$=1

https://en.wikipedia.org/wiki/Flip-flop_(electronics)

NOR-based SR Latch – HOLD





HOLD	S=0 R=0

Q=old Q

$\overline{Q}$=old $\overline{Q}$

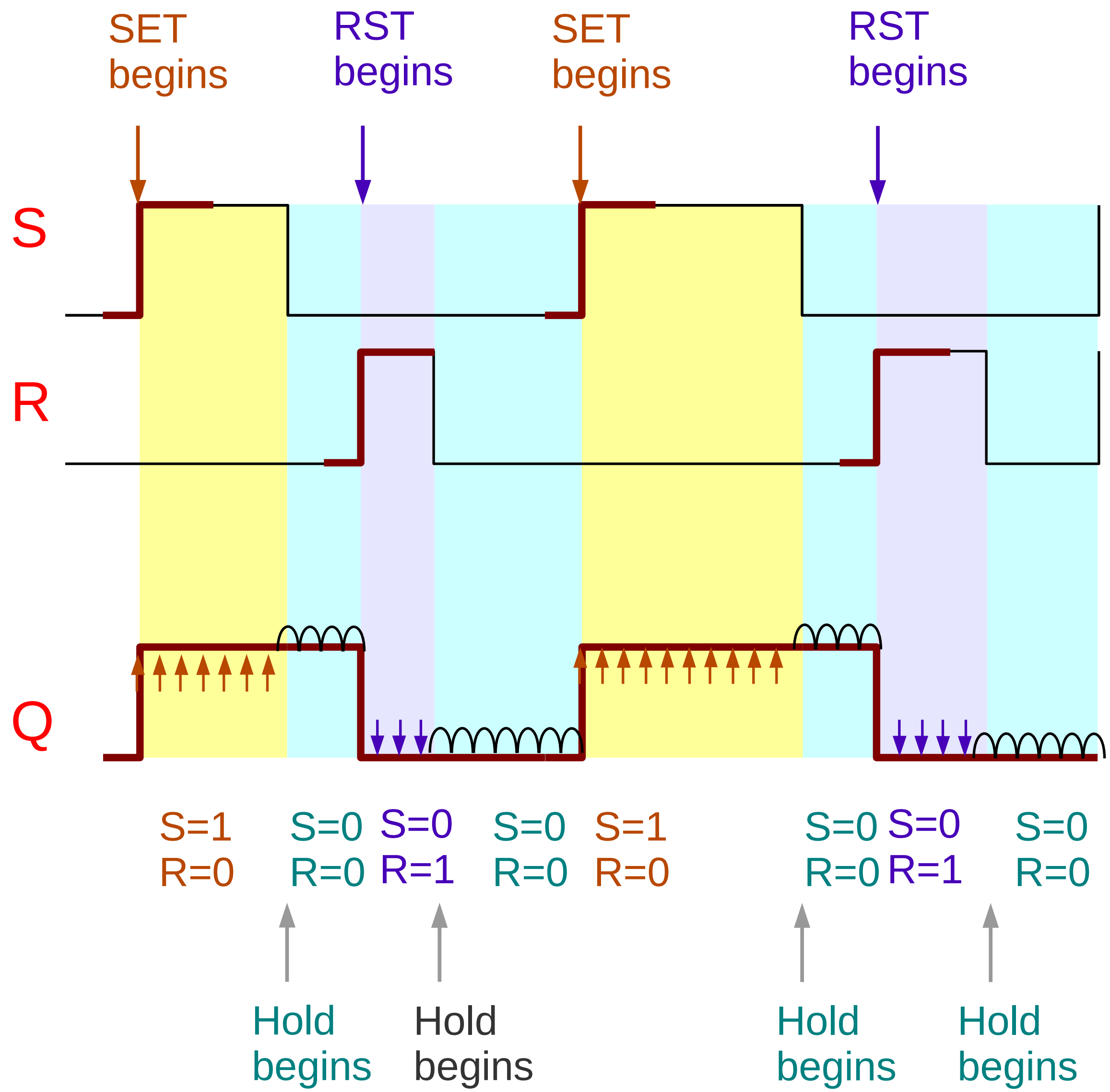
HOLD	S=0 R=0

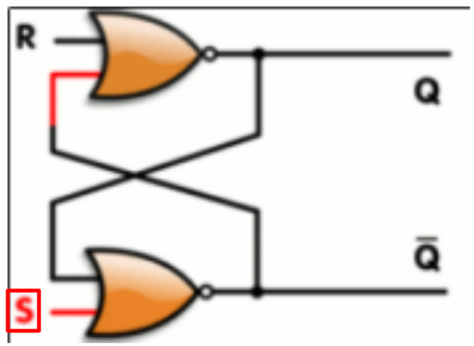
Q=old Q

$\overline{Q}$=old $\overline{Q}$

https://en.wikipedia.org/wiki/Flip-flop_(electronics)

NOR-based SR Latch

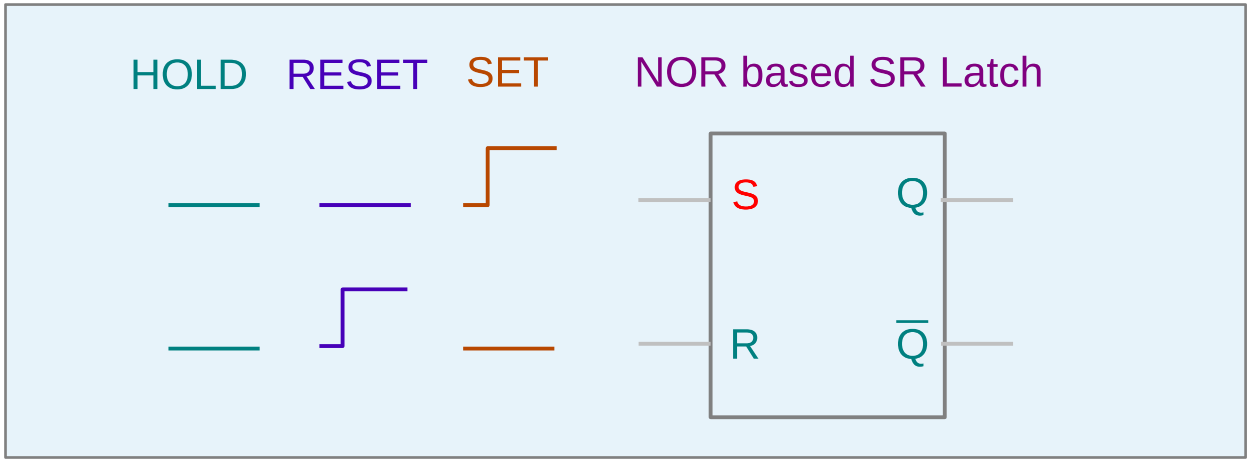


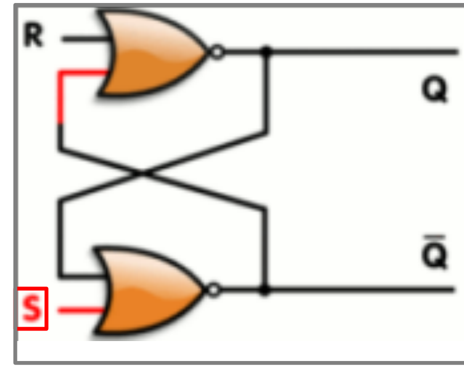


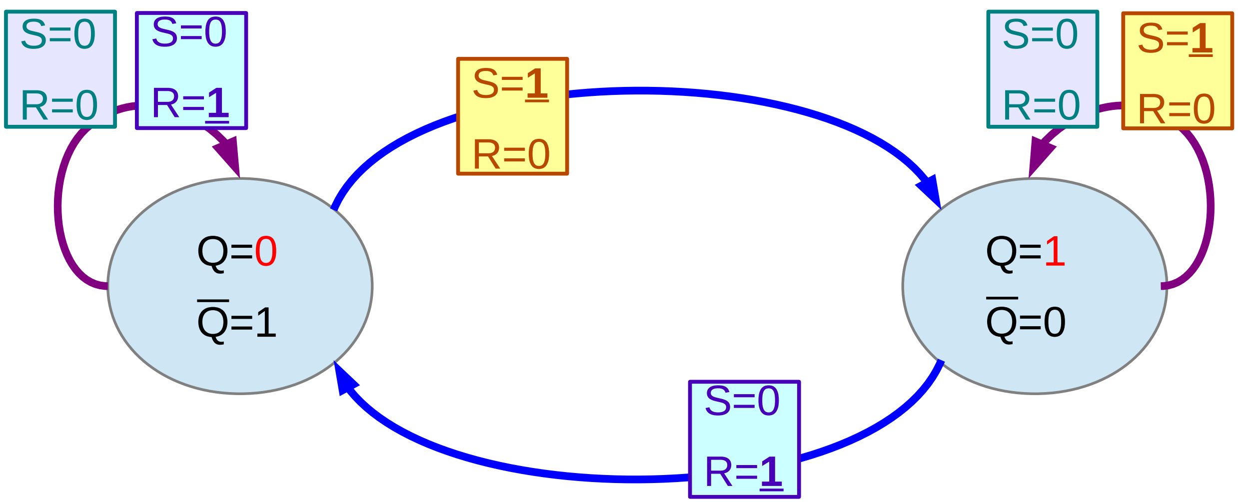
SET	S=**1** R=0	Q=1 $\overline{Q}$=0
RESET	S=0 R=**1**	Q=0 $\overline{Q}$=1
HOLD	S=0 R=0	Q=old Q $\overline{Q}$=old $\overline{Q}$

https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

NOR-based SR Latch States



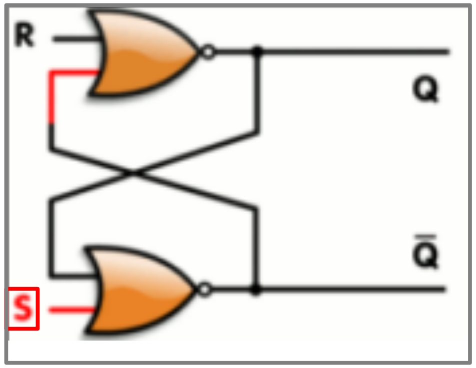


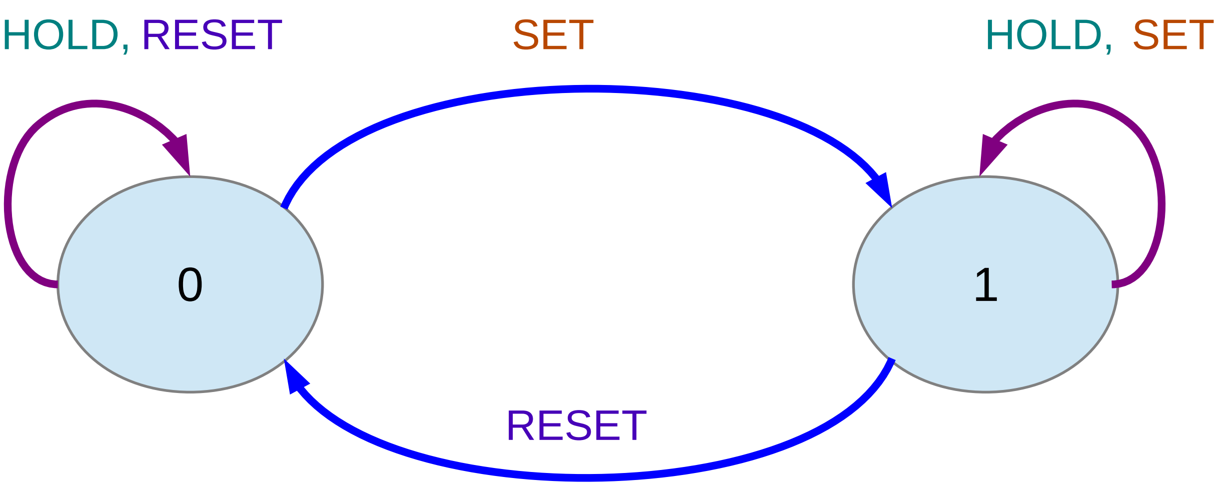


SET	S=**1** R=0	Q=1 $\overline{Q}$=0
RESET	S=0 R=**1**	Q=0 $\overline{Q}$=1
HOLD	S=0 R=0	Q=old Q $\overline{Q}$=old $\overline{Q}$

https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

SR Latch States





SET	S=**1** R=0	Q=1 $\overline{Q}$=0
RESET	S=0 R=**1**	Q=0 $\overline{Q}$=1
HOLD	S=0 R=0	Q=old Q $\overline{Q}$=old $\overline{Q}$

https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

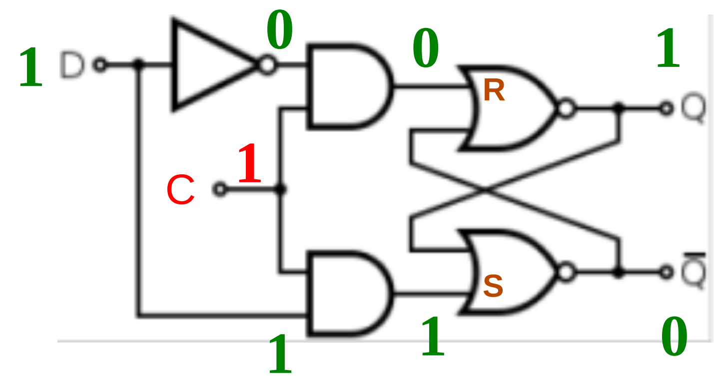
NOR-based D Latch – SET / RESET

https://en.wikipedia.org/wiki/Flip-flop_(electronics)

D=**1**
C=1

SET
S=**1**
R=0

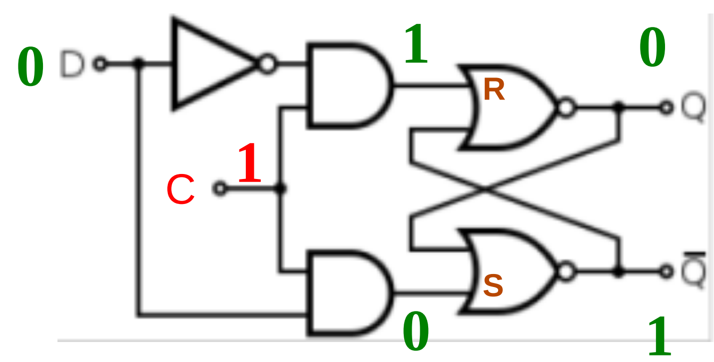
Q=1
$\overline{Q}$=0



D=**0**
C=1

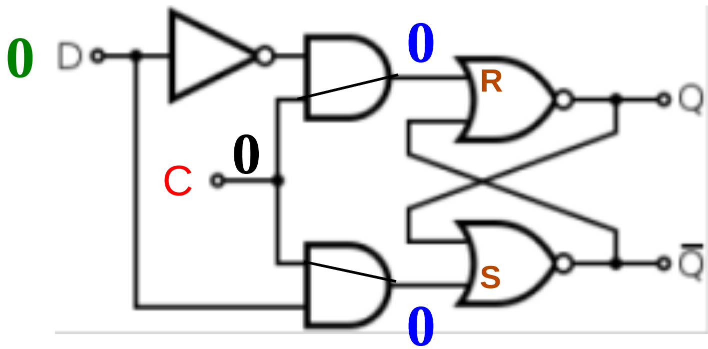
RESET
S=0
R=**1**

Q=0
$\overline{Q}$=1



NOR-based D Latch – HOLD

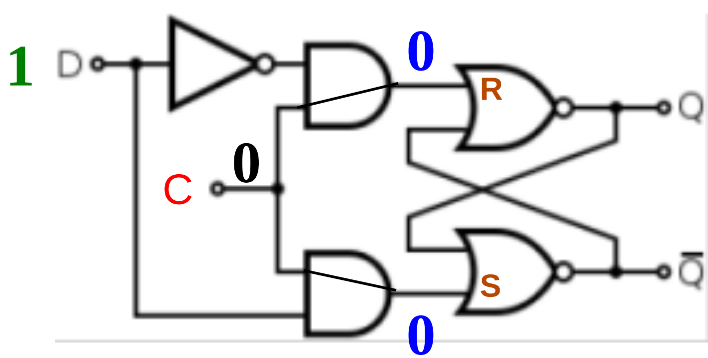
https://en.wikipedia.org/wiki/Flip-flop_(electronics)



D=**X**
C=0

HOLD
S=0
R=0

Q=old Q
$\overline{Q}$=old $\overline{Q}$

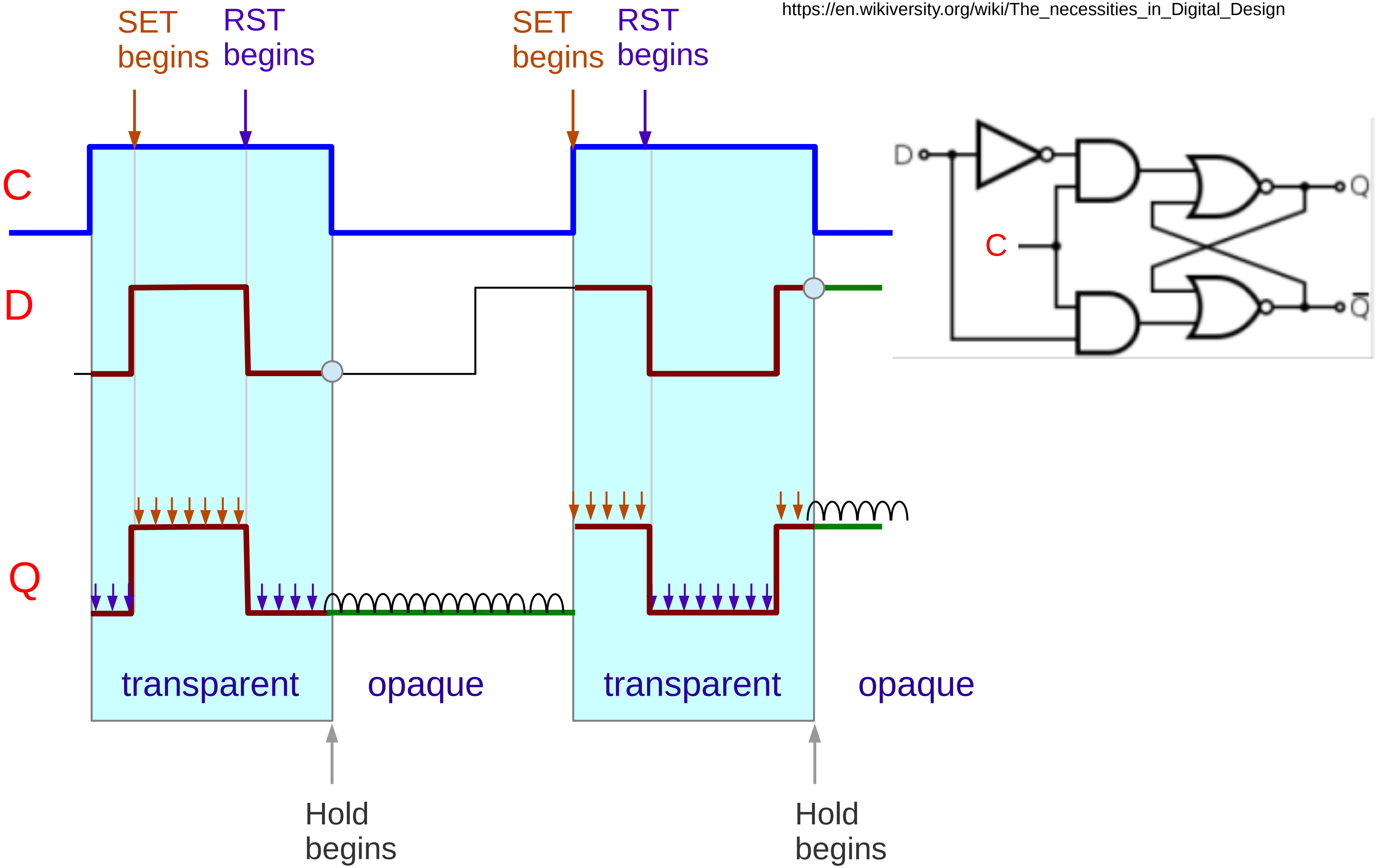


D=**X**
C=0

HOLD
S=0
R=0

Q=old Q
$\overline{Q}$=old $\overline{Q}$

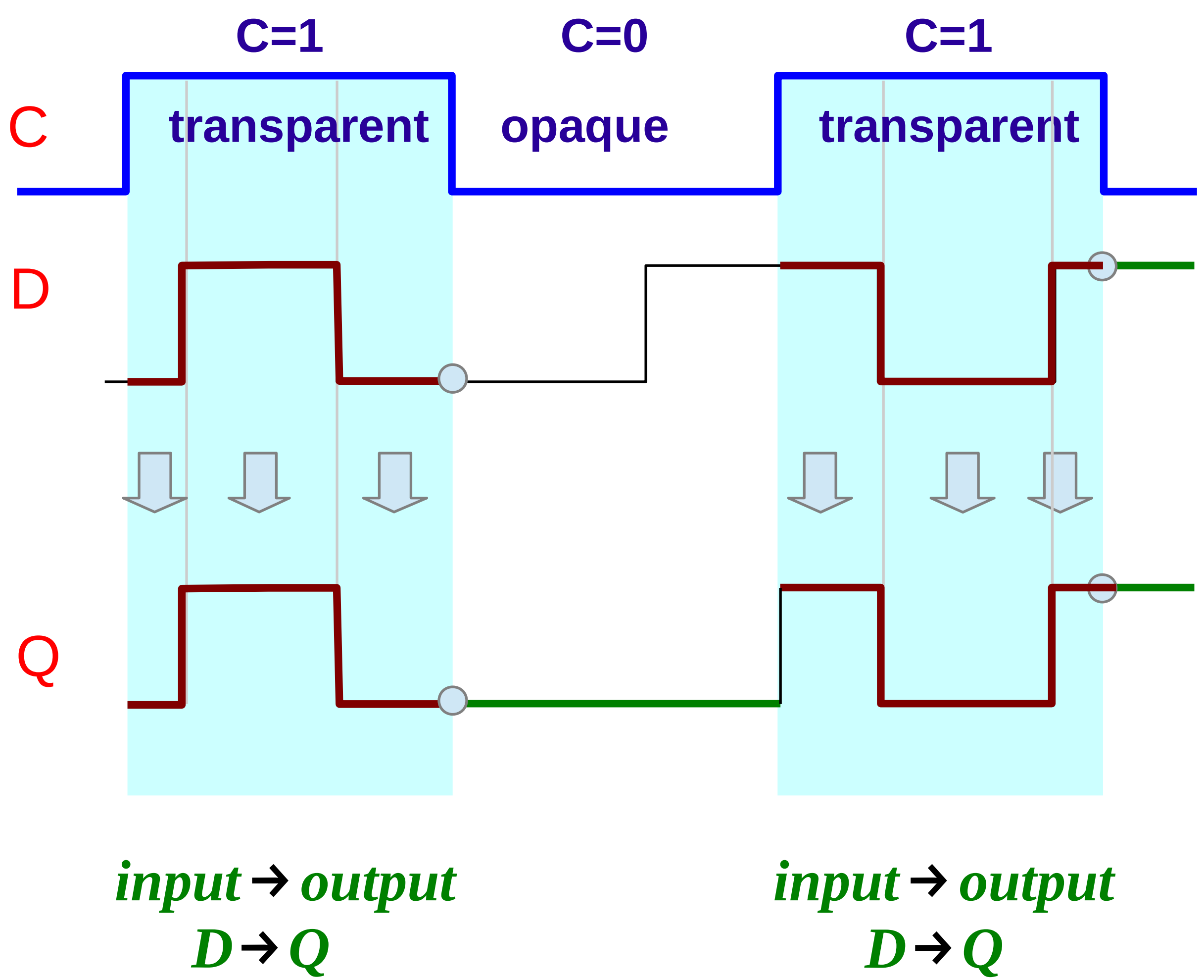
NOR-based D Latch – Set / Reset / Hold

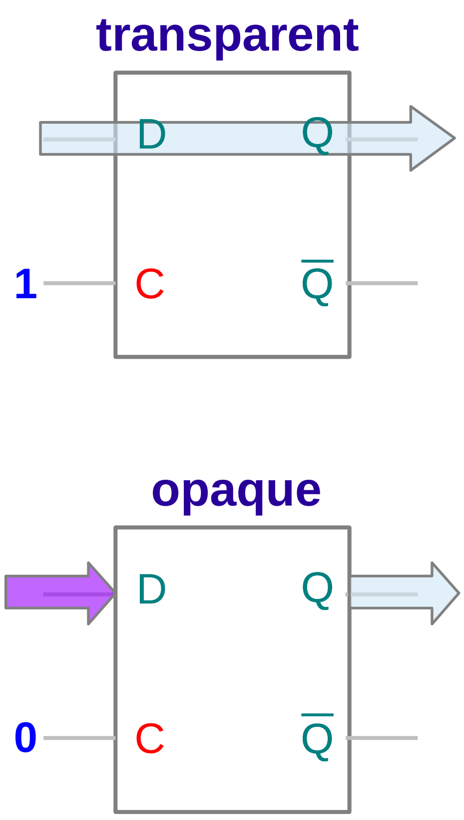


https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

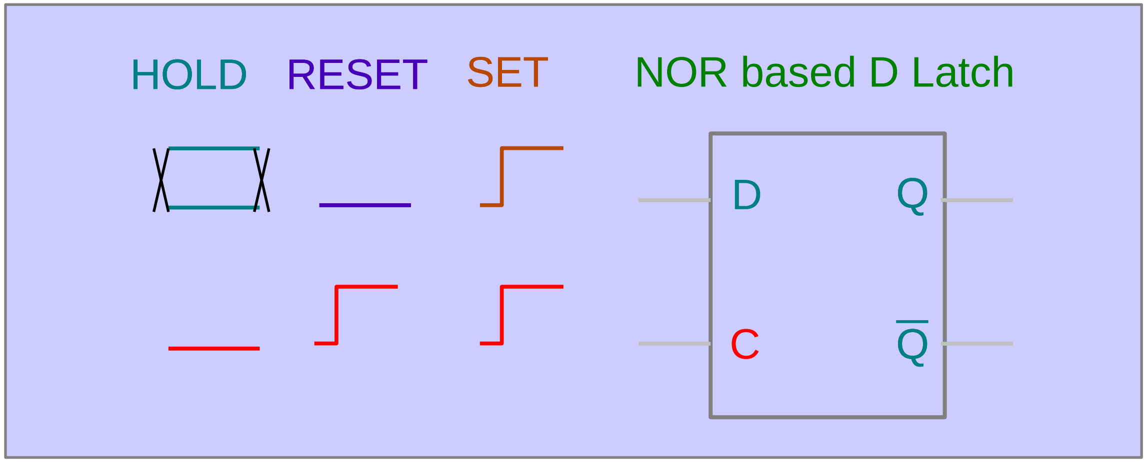
NOR-based D Latch – transparent / opaque

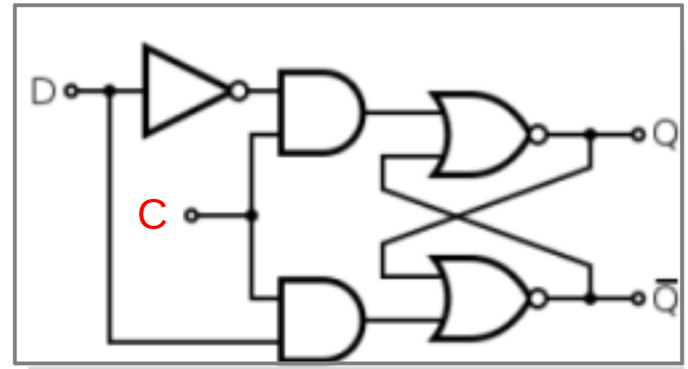
https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

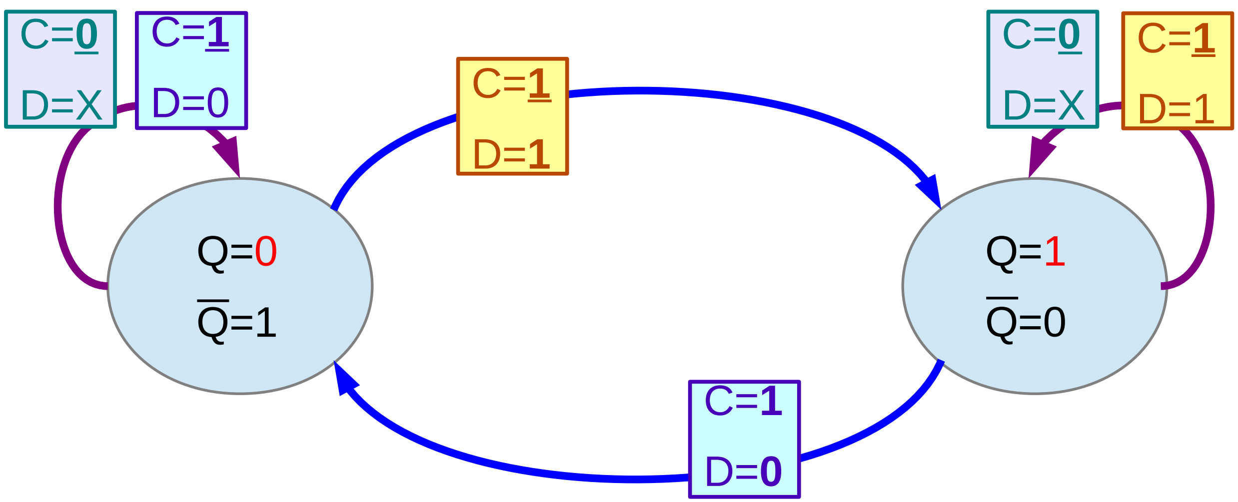




NOR-based D Latch States

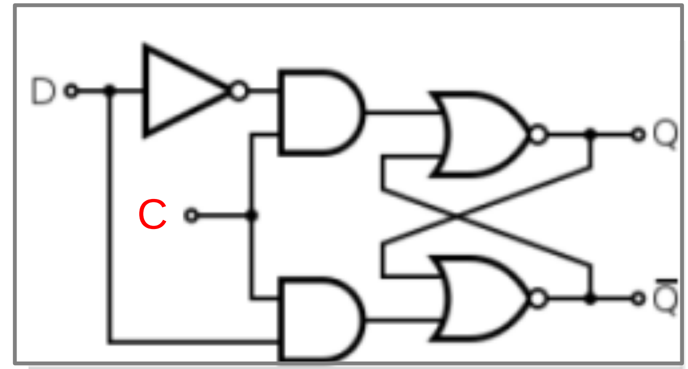


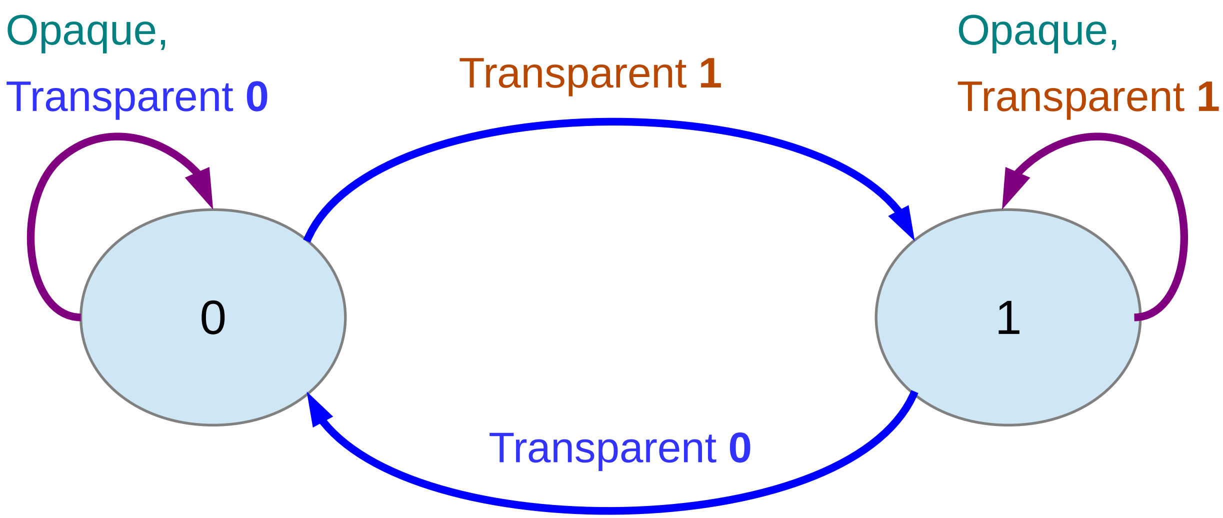




https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

D Latch States

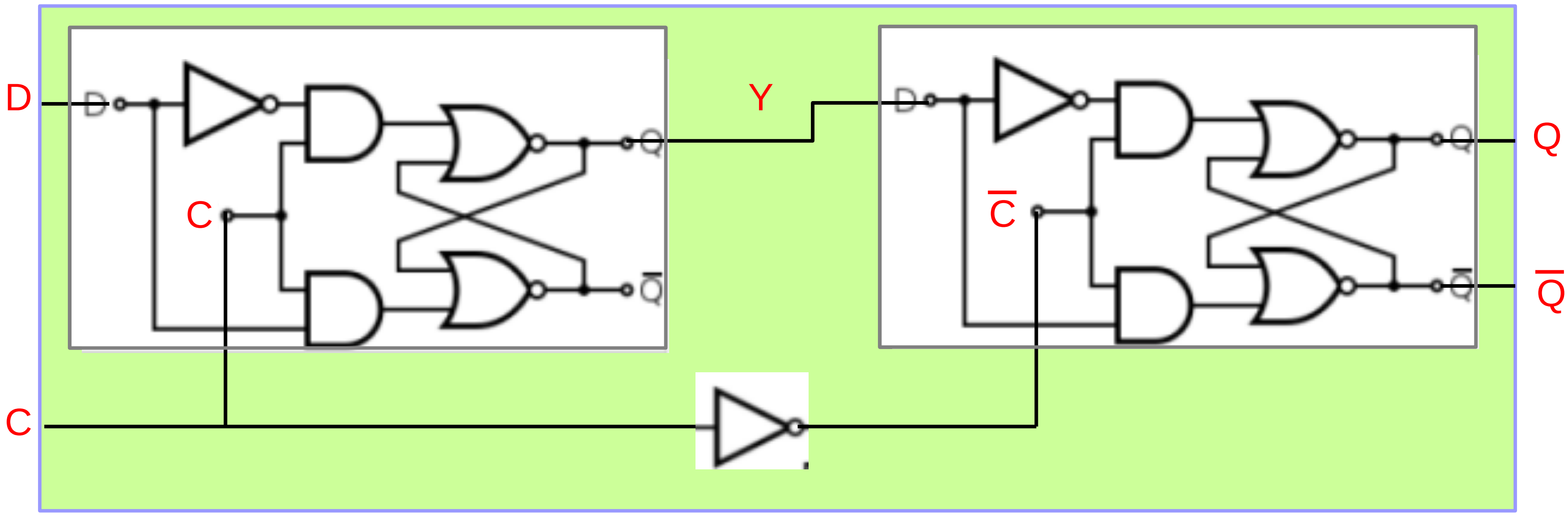


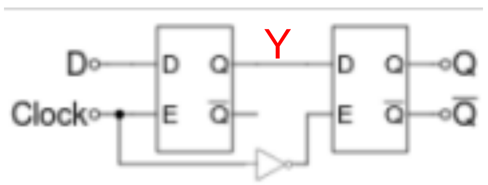


Trans 1	C=**1** D=**1**	Q=1 $\overline{Q}$=0
Trans 0	C=**1** D=0	Q=0 $\overline{Q}$=1
Opaque	C=**0** D=X	Q=old Q $\overline{Q}$=old $\overline{Q}$

https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

Master-Slave FlipFlops

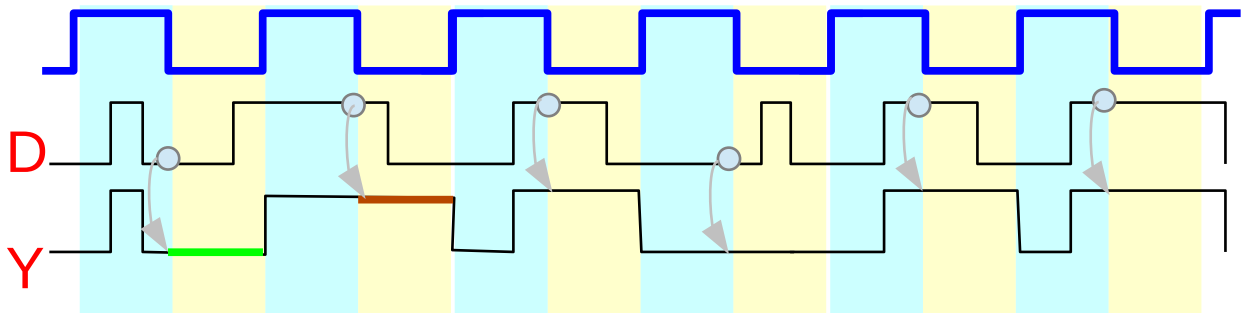


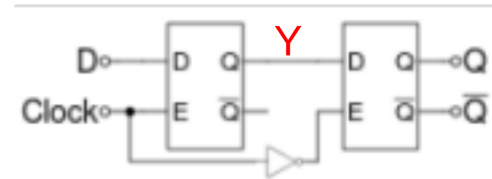


https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

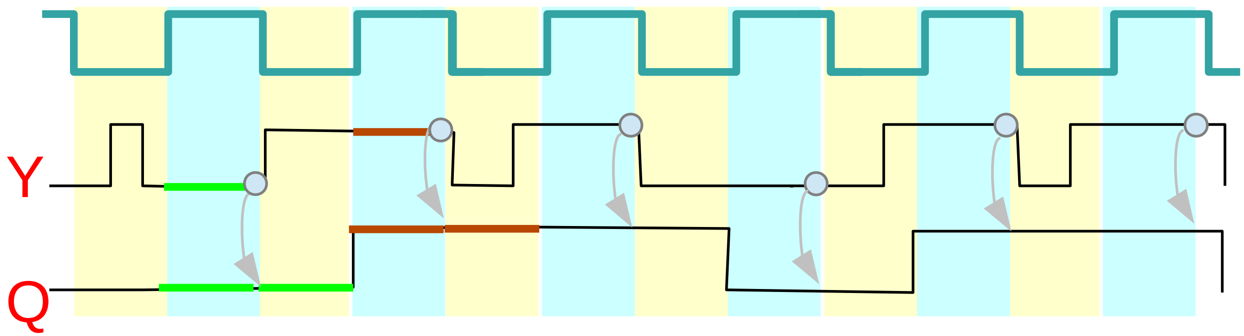
Master-Slave D FlipFlop

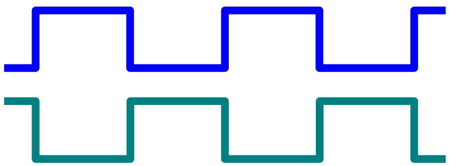
Master D Latch



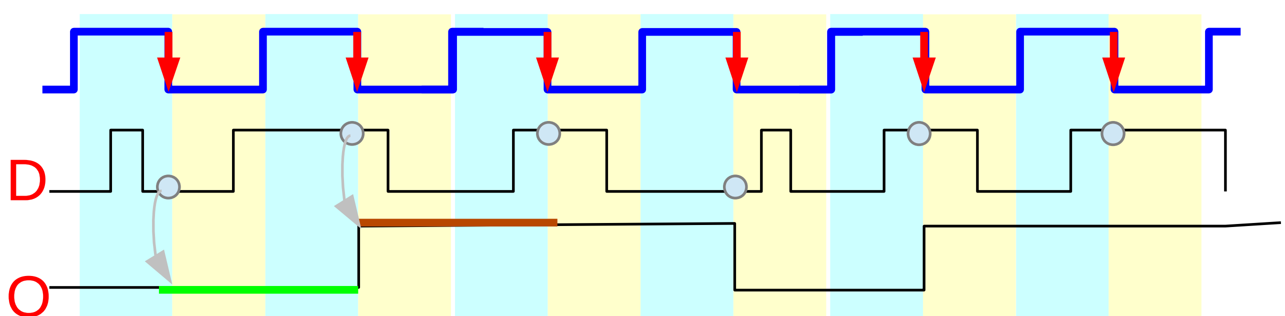


Slave D Latch

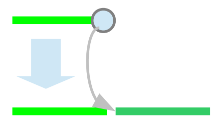




Master-Slave D F/F



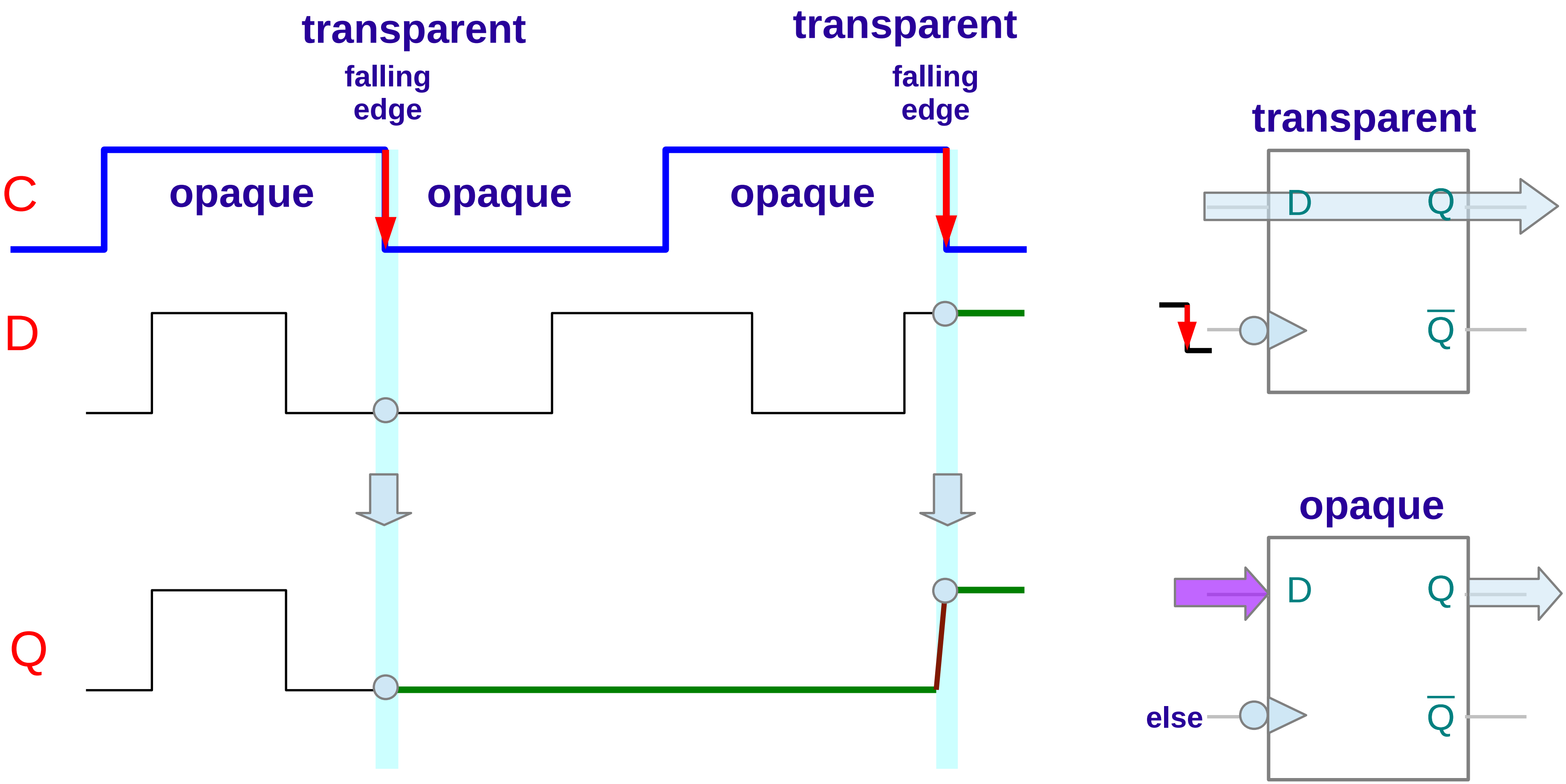
https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design



the hold output of the master is transparently reaches the output of the slave

this value is held for another half period

Master Slave D FlipFlop – transparent / opaque



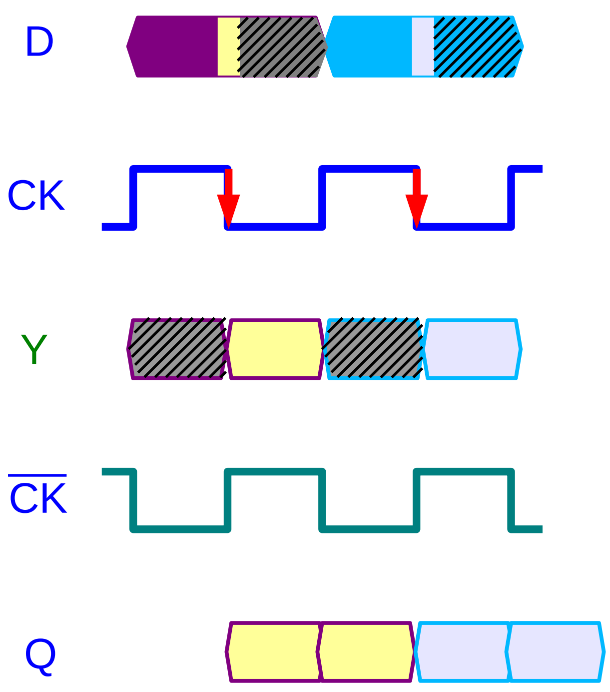
input → output
D → Q

input → output
D → Q

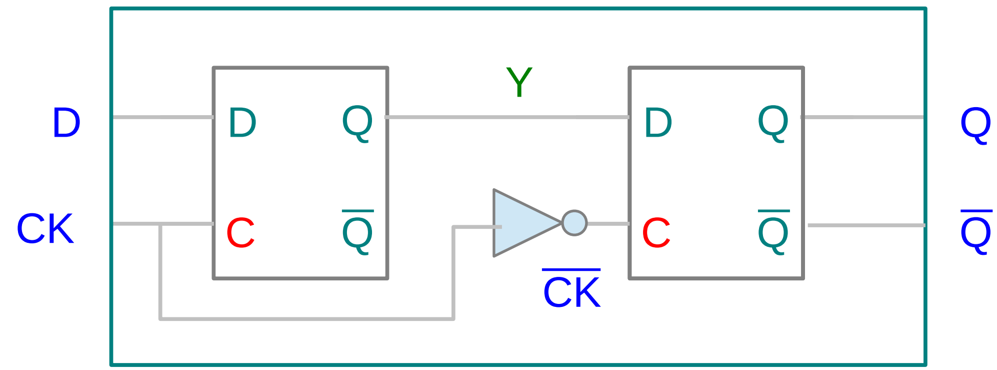
https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

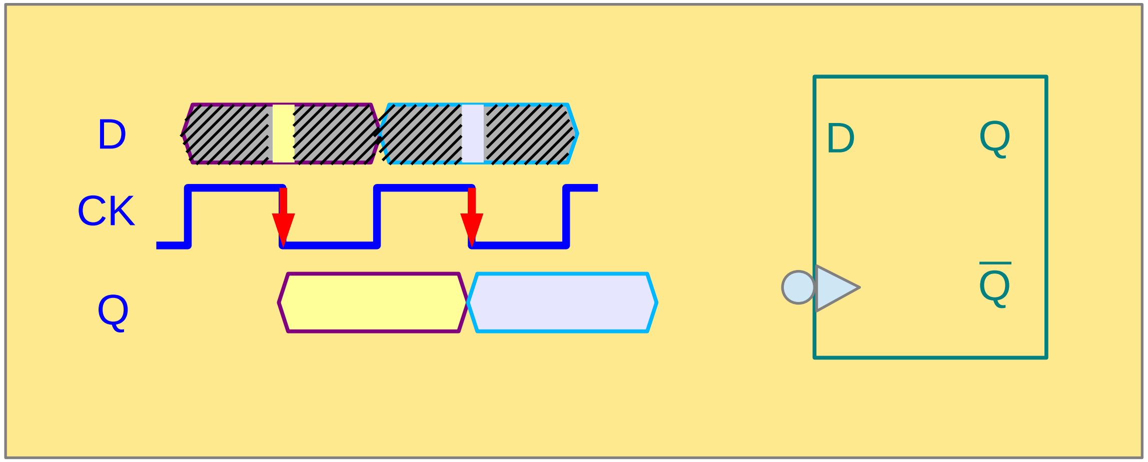
Master-Slave D FlipFlop – Falling Edge

Master D Latch



Slave D Latch



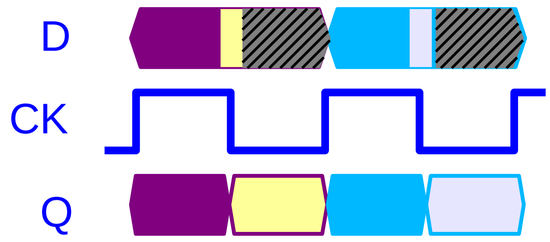


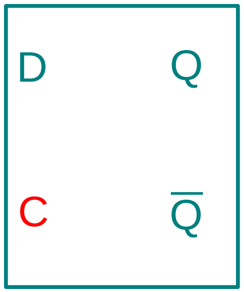
https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

D Latch & D FlipFlop

Level Sensitive D Latch

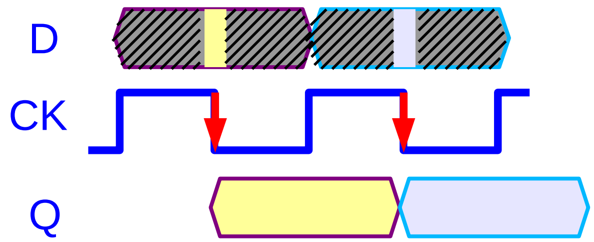
CK=1 transparent
CK=0 opaque

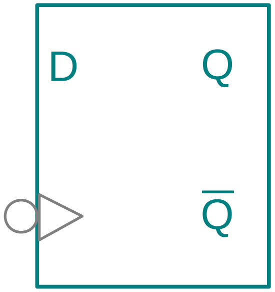




Edge Sensitive D FlipFlop

CK=1→0 transparent
else opaque

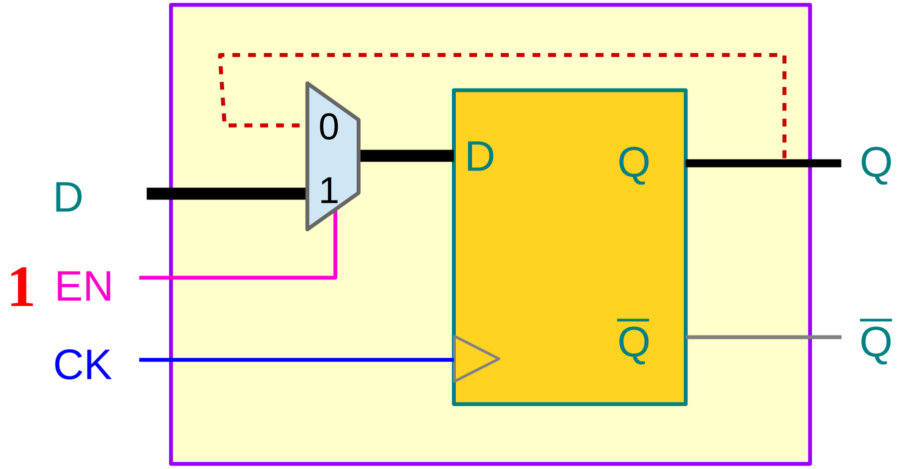




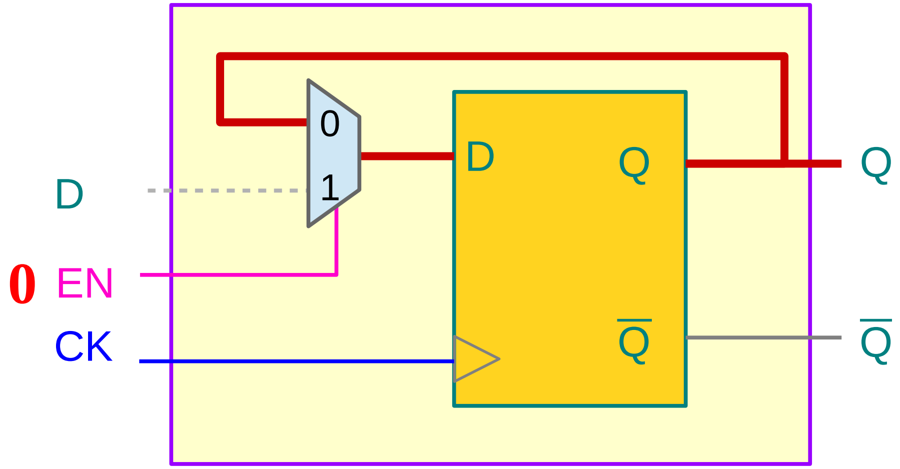
https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

D FlipFlop with Enable (1)

EN=1 Regular D Flip Flop
Sampling **D** input
@ **posedge** of CK

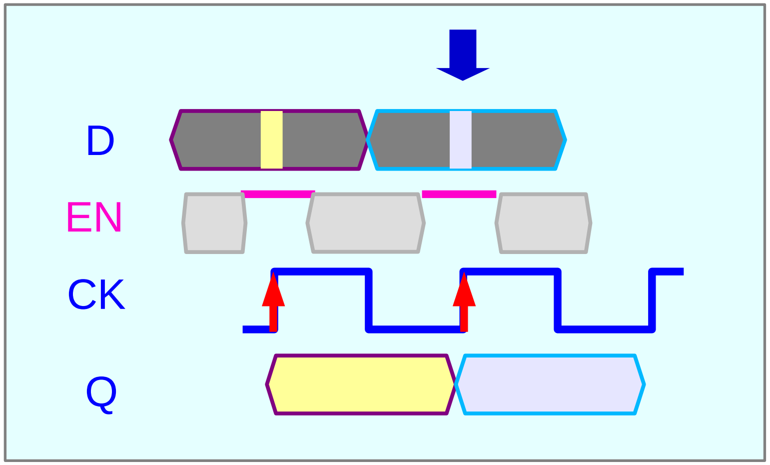


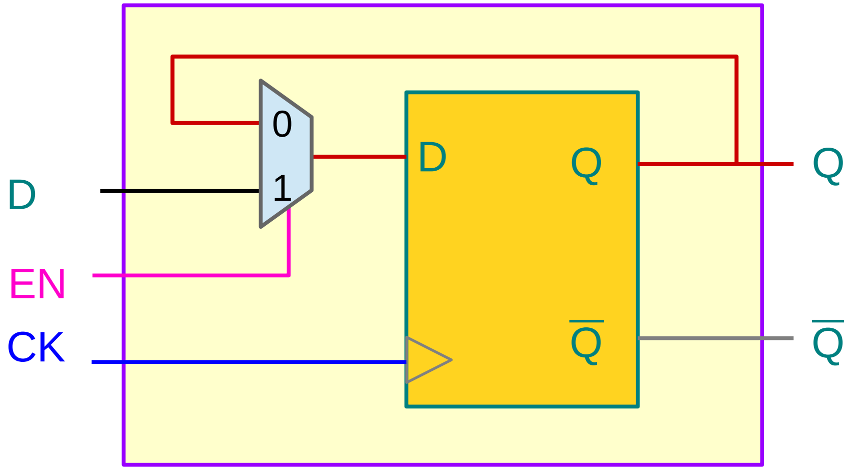
EN=0 Holding D Flip Flop
Sampling **Q** output
@ **posedge** of CK

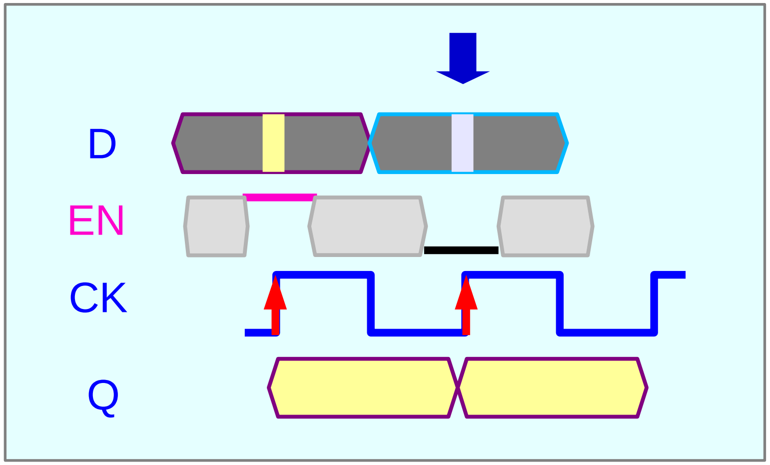


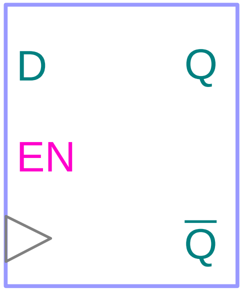
https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

D FlipFlop with Enable (2)



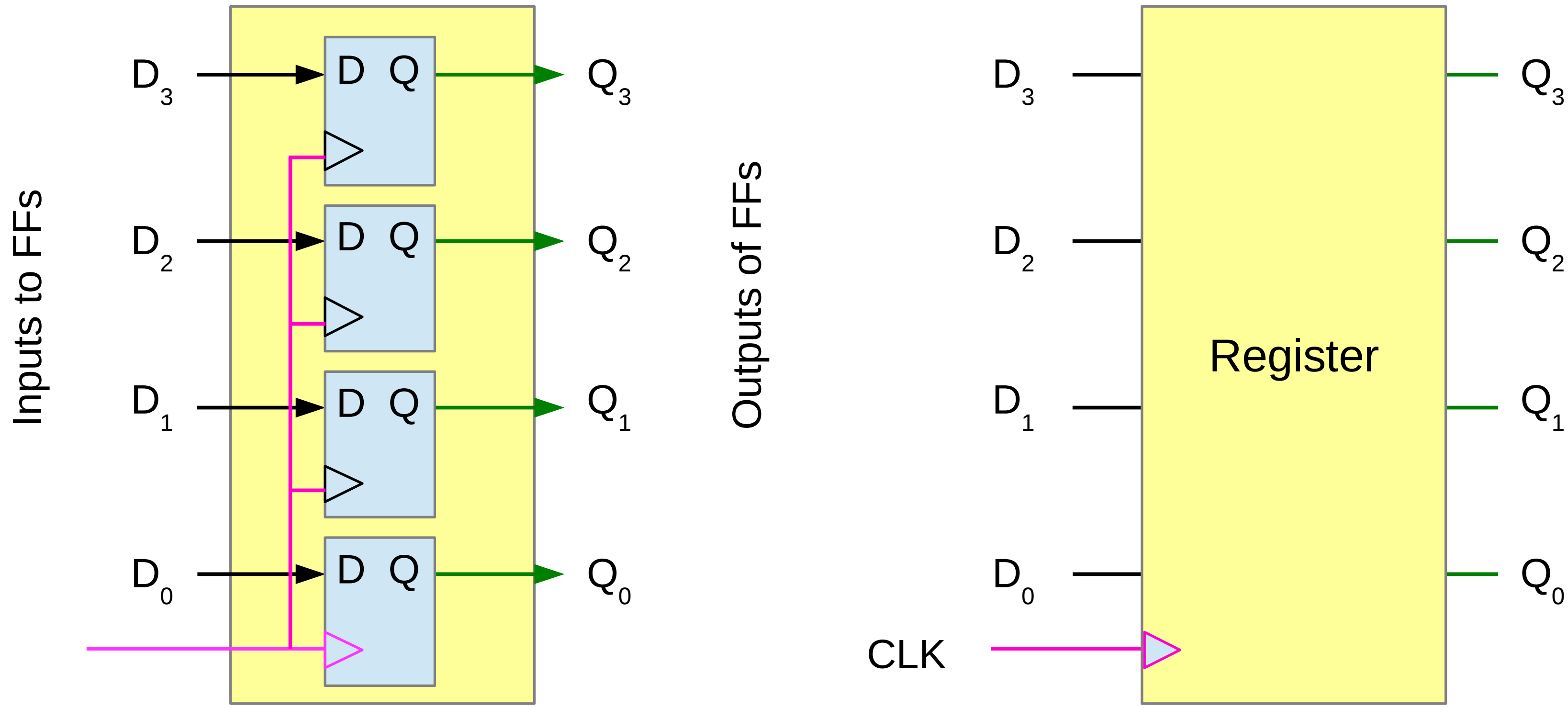






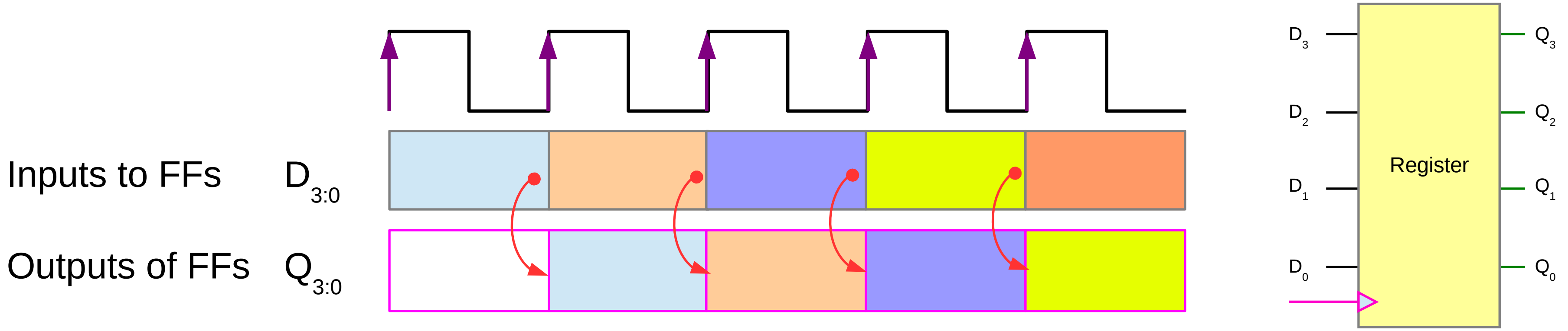
https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

Registers



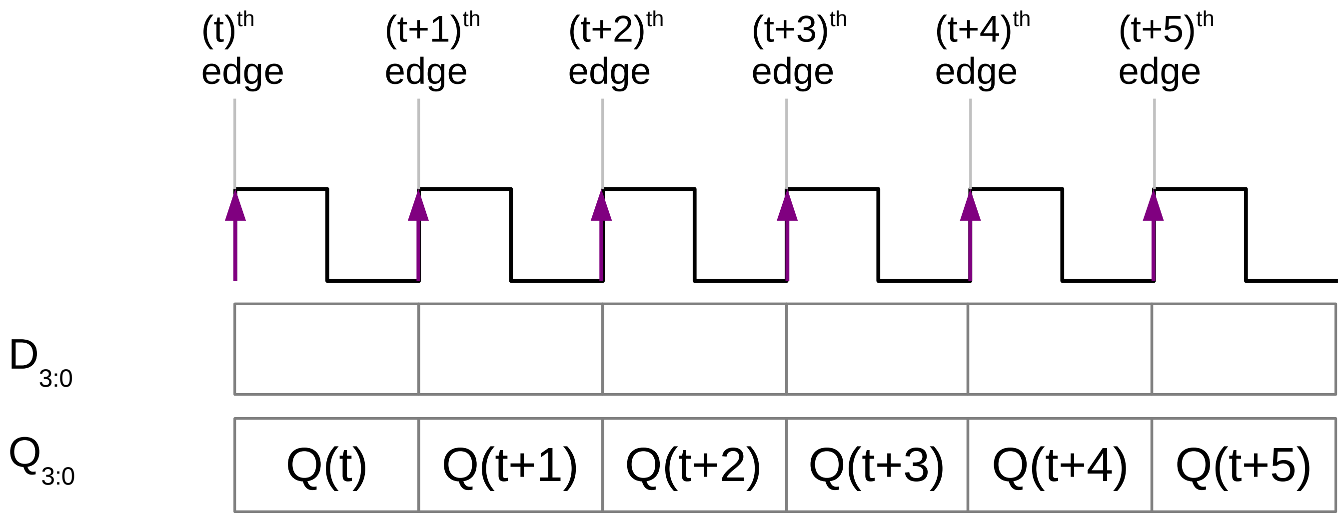
https://en.wikiversity.org/wiki/The_necessities_in_Computer_Design

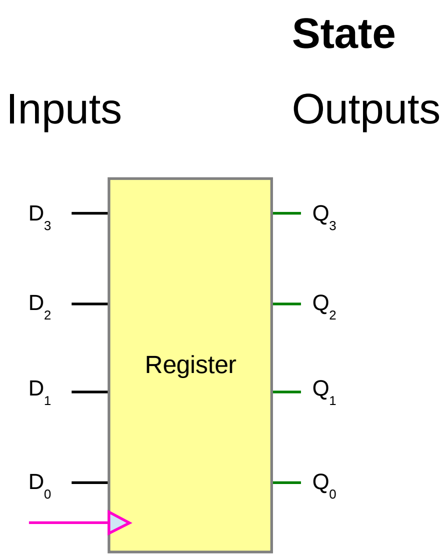
FF Timing (Ideal)



https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

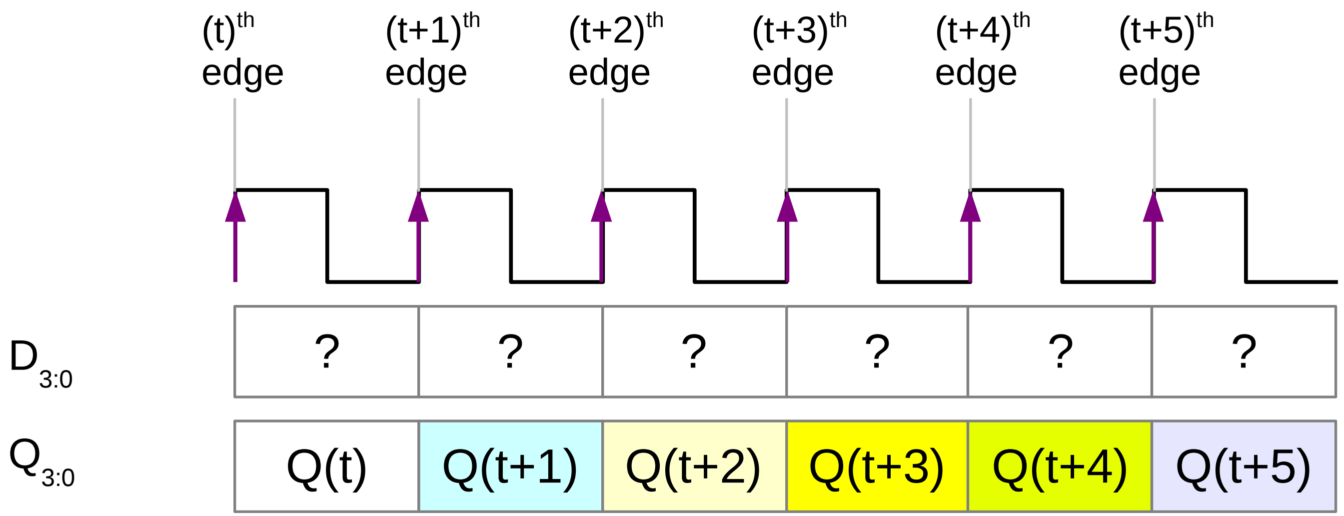
States

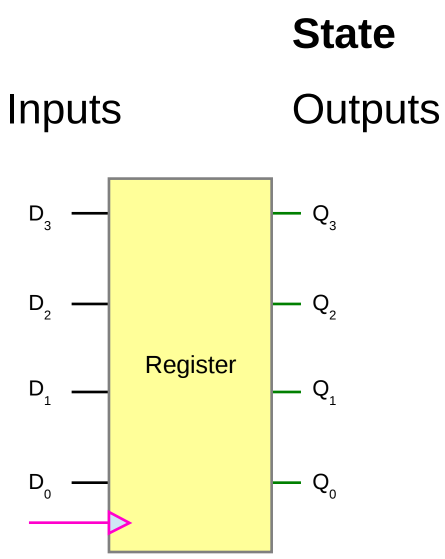




https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

Sequence of States



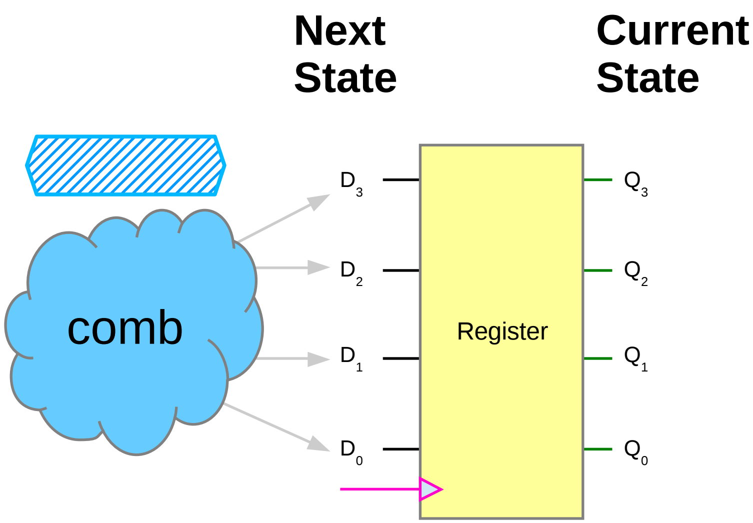


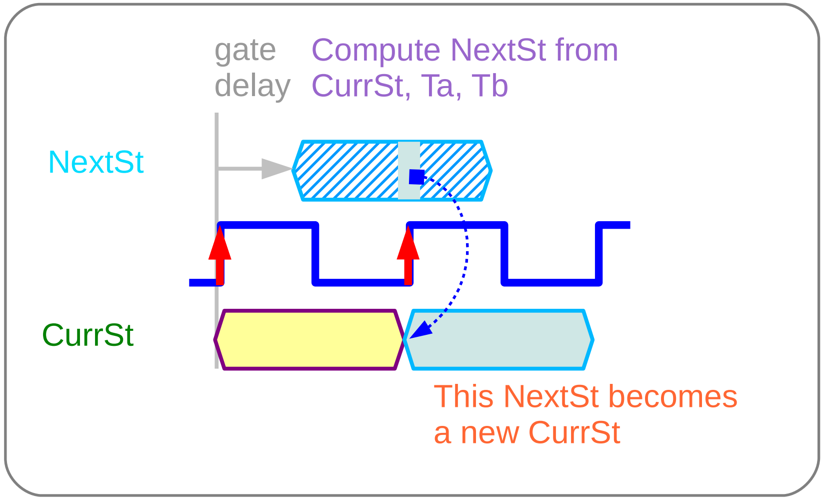
Find inputs to FFs

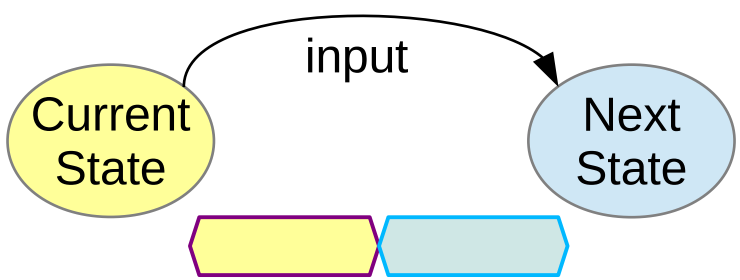
which will make outputs
in this sequence

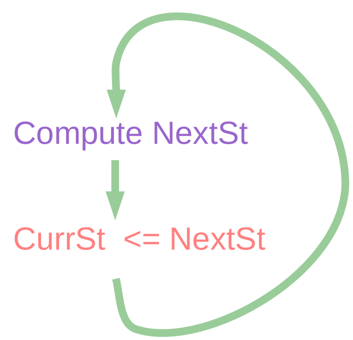
https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

How to change current state



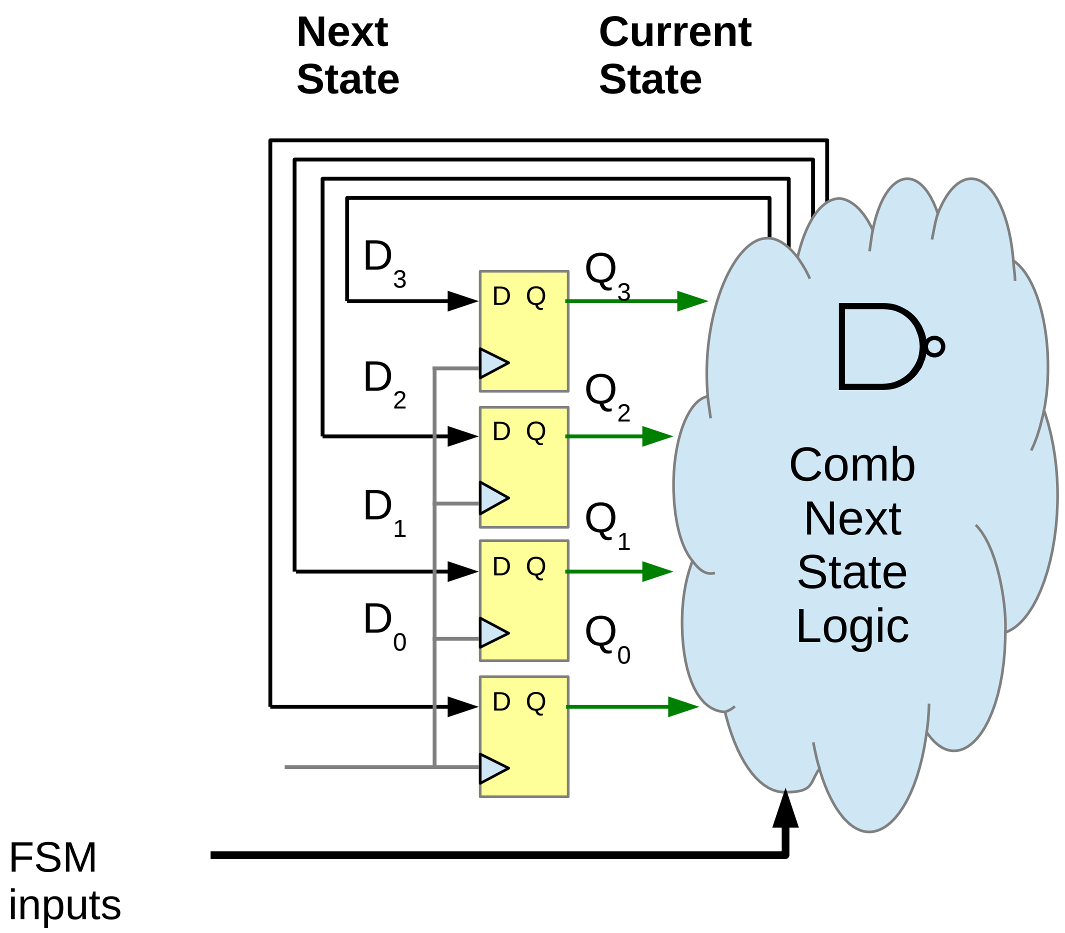






https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

Finding FF Inputs



During the t^{th} clock edge period,

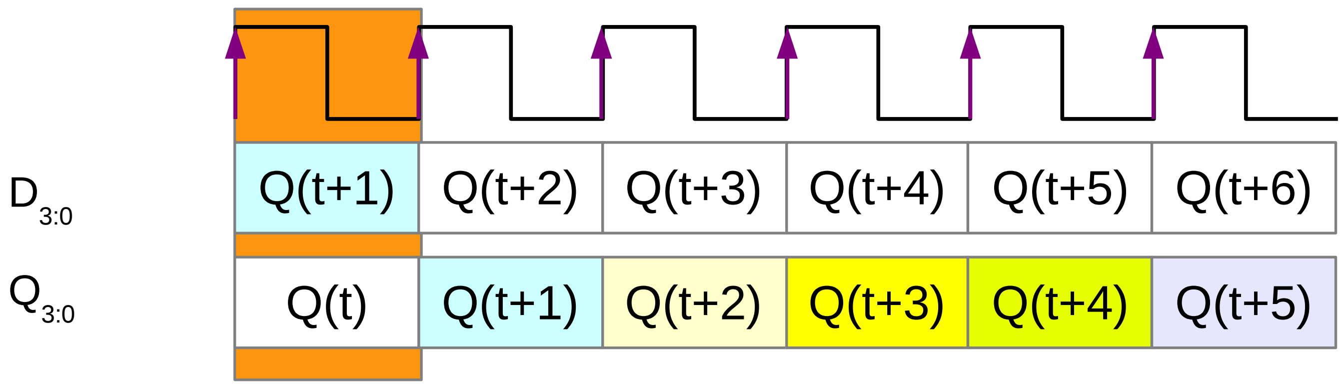
Compute the next state Q(t+1) using the current state Q(t) and other external inputs

Place it to FF inputs

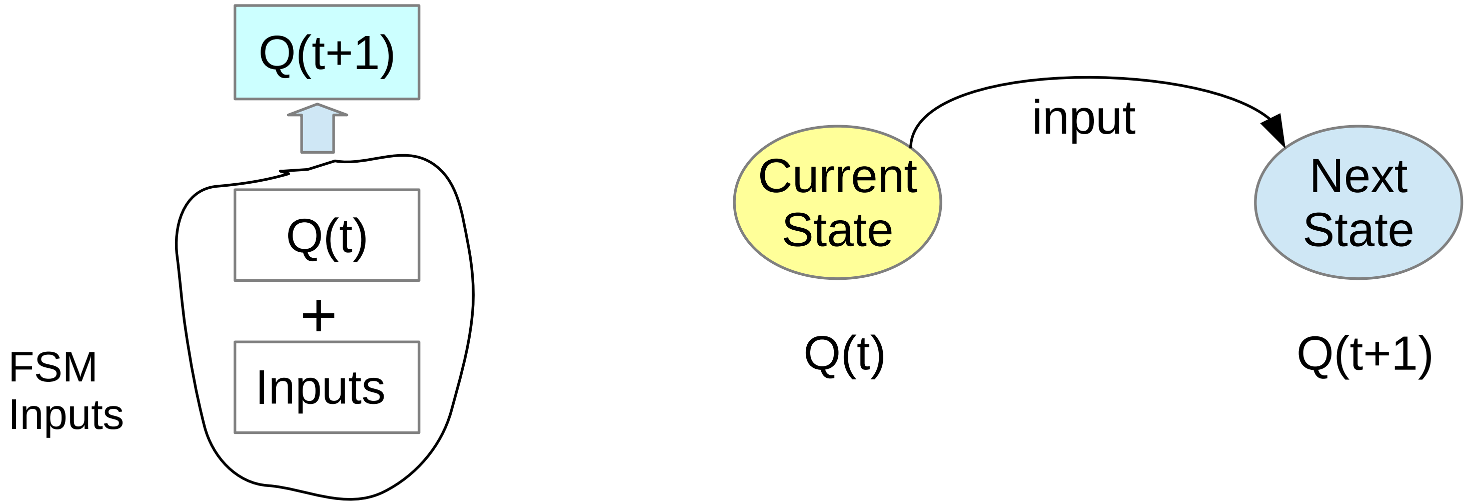
After the next clock edge, $(t+1)^{th}$, the computed next state Q(t+1) becomes the current state

https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

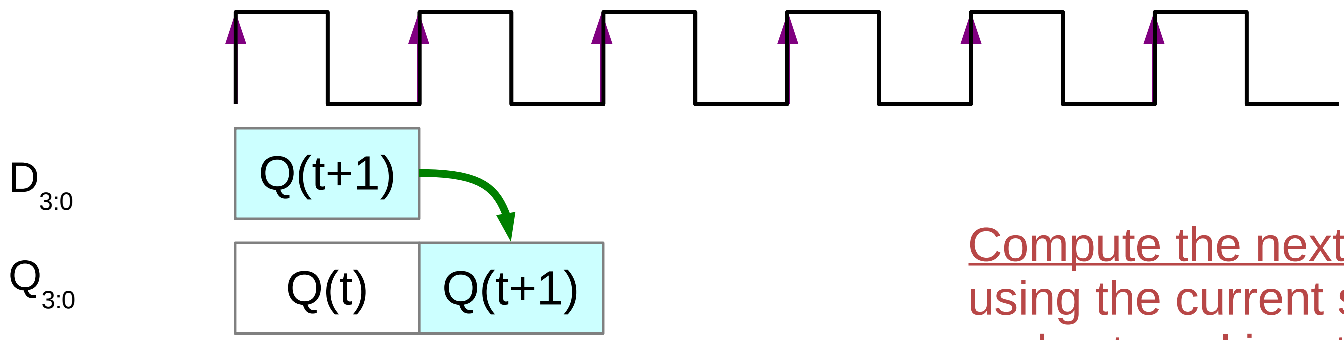
Method of Finding FF Inputs



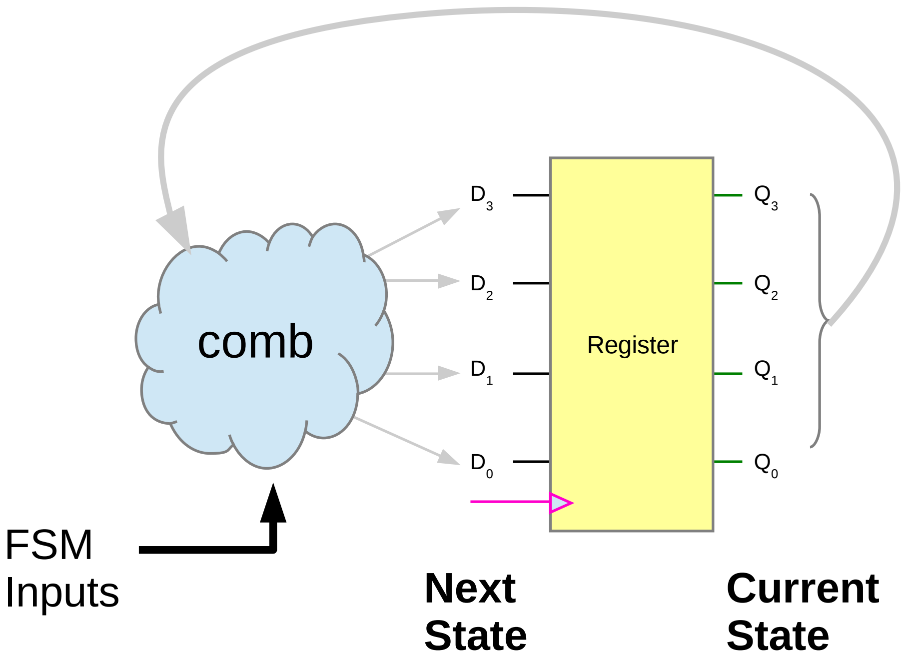
Find the boolean functions
D3, D2, D1, D0
in terms of Q3, Q2, Q1, Q0,
and external FSM inputs
for all possible cases.

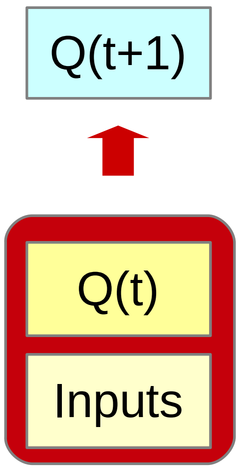


State Transition



<u>Compute the next state</u>
using the current state
and external inputs
in the current clock cycle





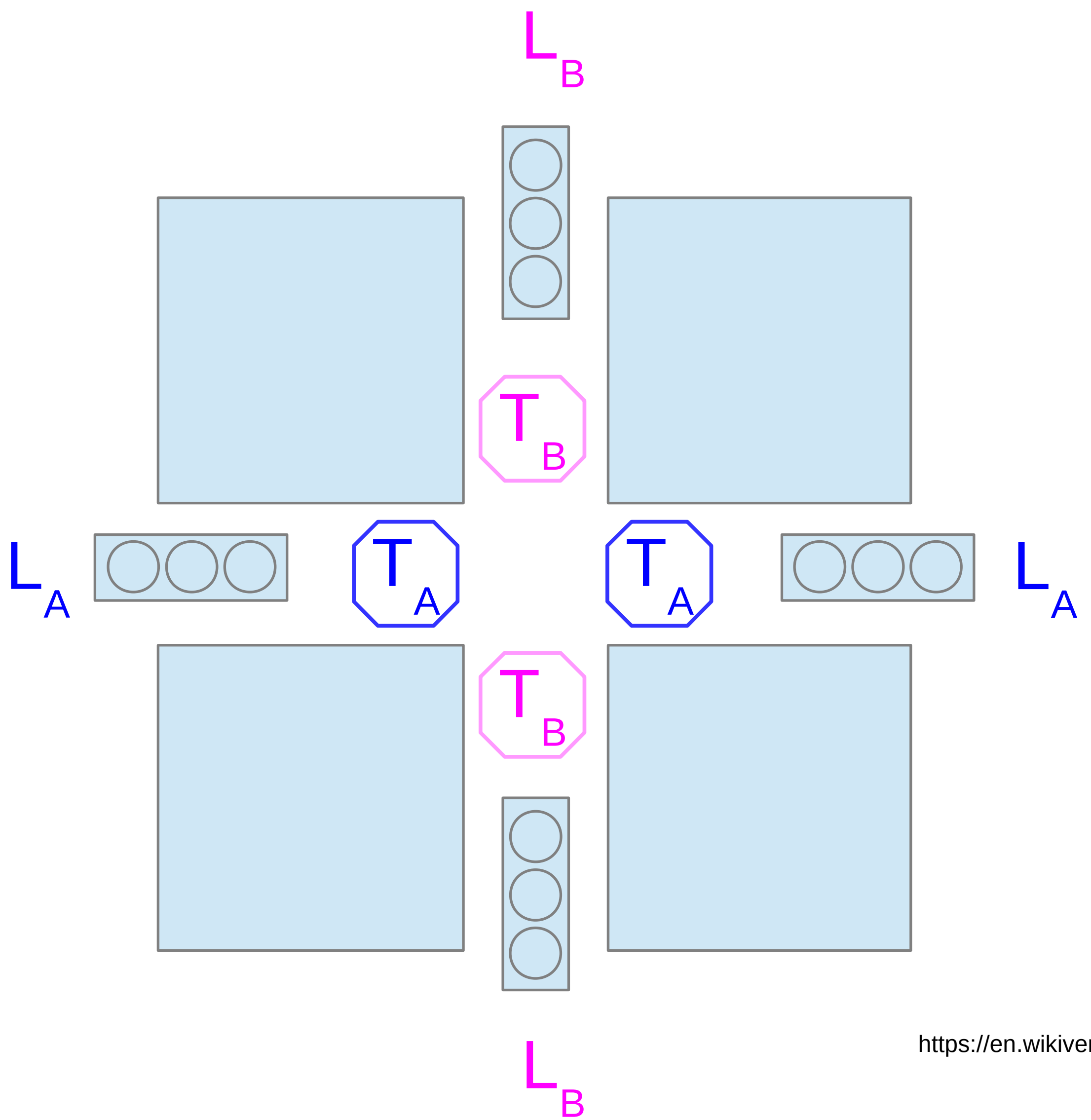
https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

After the next clock edge,
the computed next state (FF Inputs)
becomes the current state (FF Outputs)

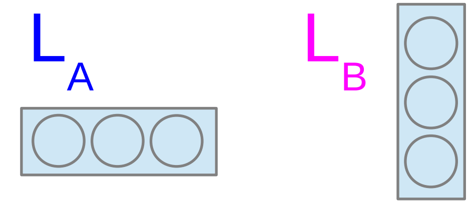
Traffic Lights Example

https://en.wikiversity.org/wiki/The_necessities_in_Computer_Design

FSM Inputs and Outputs



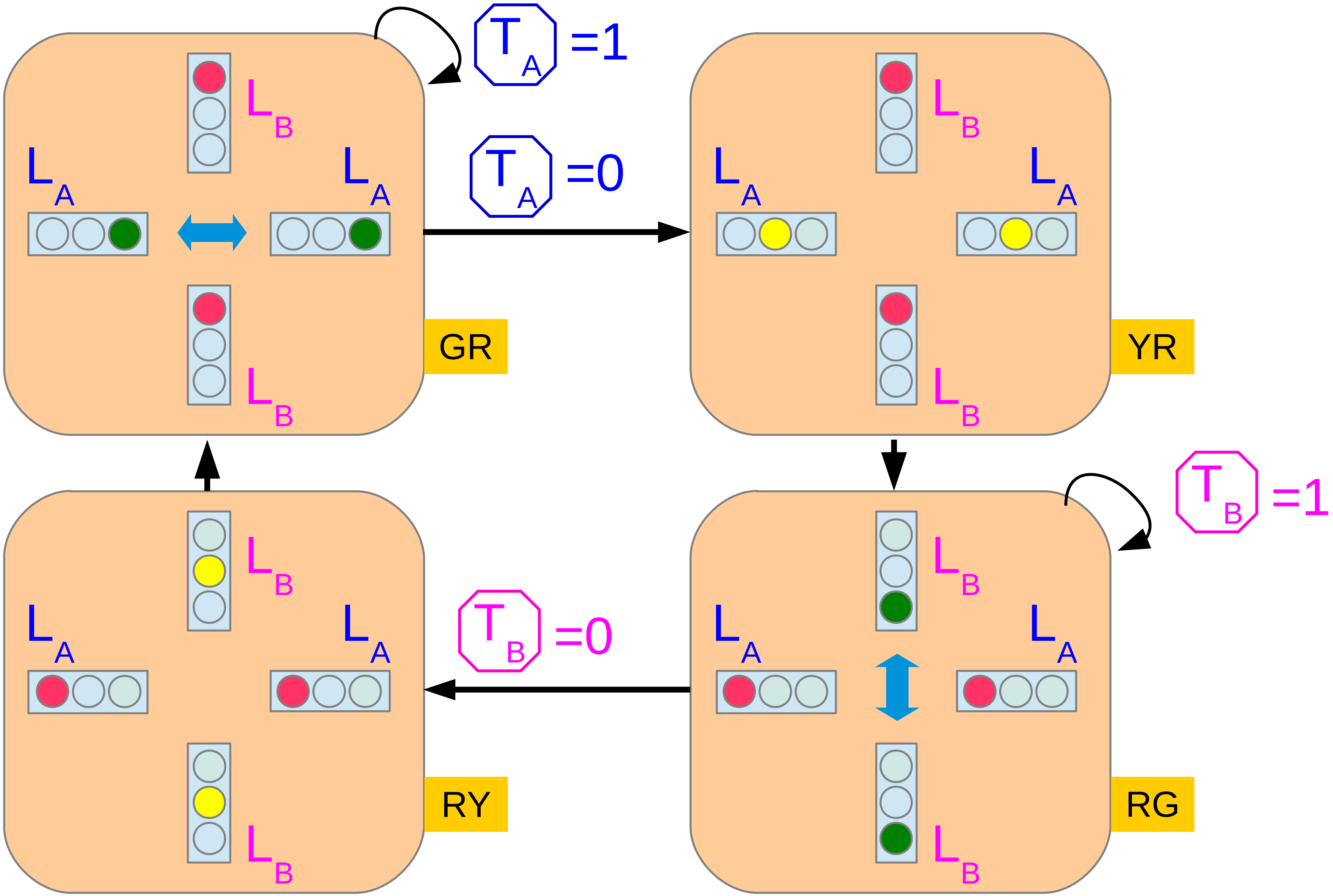
Traffic Lights - **Outputs**



Sensor - **Inputs**

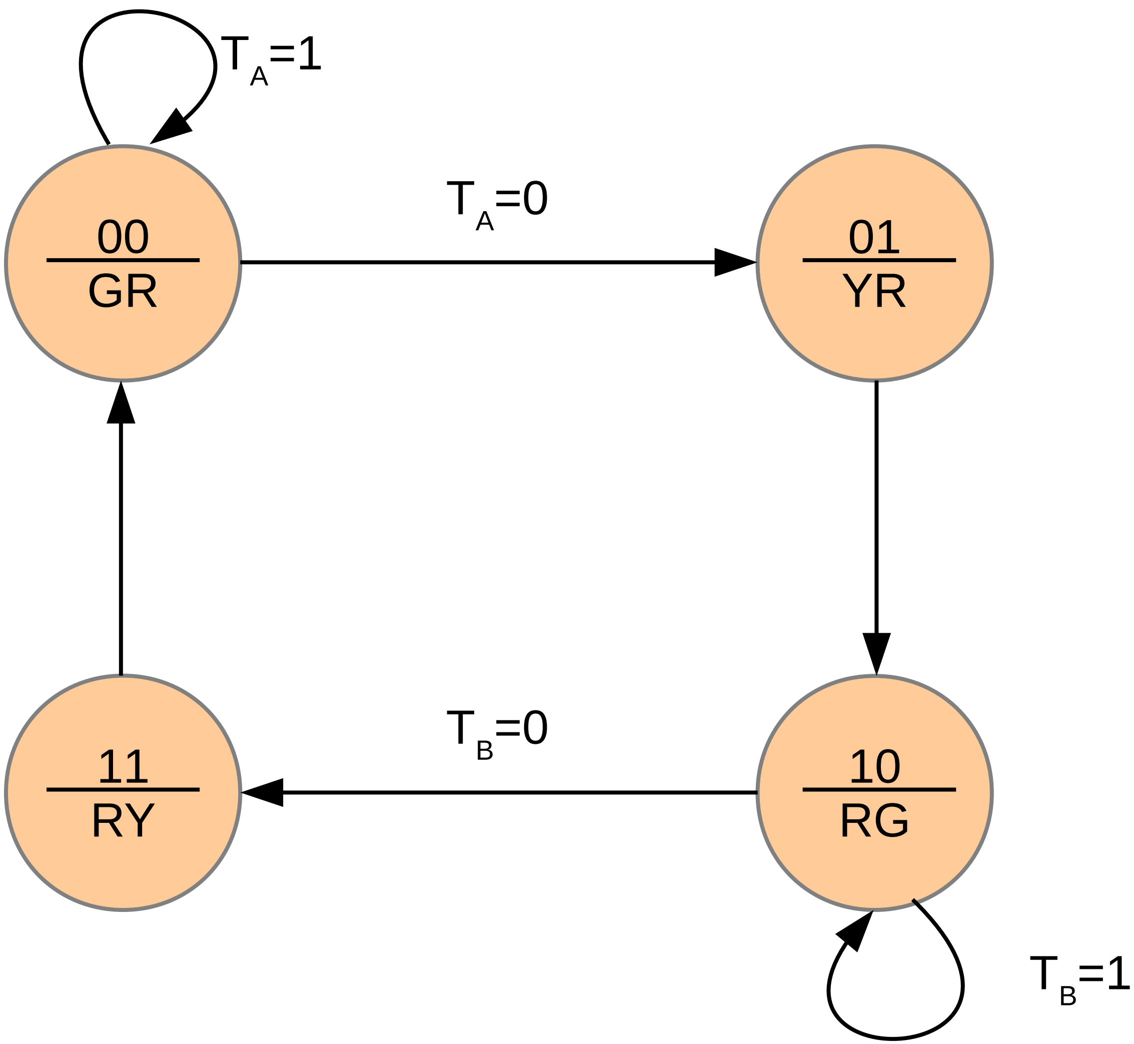
https://en.wikiversity.org/wiki/The_necessities_in_Computer_Design

Four States



https://en.wikiversity.org/wiki/The_necessities_in_Computer_Design

State Transition Diagrams and Tables



S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

S_1	S_2	L_{A1}	L_{A0}	L_{B1}	L_{B0}		
0	0	0	0	1	0	G	R
0	1	0	1	1	0	Y	R
1	0	1	0	0	0	R	G
1	1	1	0	0	1	R	Y

G:00
Y:01
R:10

Next State Functions S_1' and S_2'

S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

$$S'_1 = S_1 + S_0$$

$$S'_0 = \overline{S_1}\,\overline{S_0}\,\overline{T_A} + S_1\overline{S_0}\,\overline{T_B}$$

	S_1	S_0	T_A	T_B	S'_1
	0	0	0	X	0
	0	0	1	X	0
$\overline{S_1}S_0$	0	1	X	X	1
$S_1\overline{S_0}\,\overline{T_B}$	1	0	X	0	1
$S_1\overline{S_0}T_B$	1	0	X	1	1
	1	1	X	X	0

$$S'_1 = \overline{S_1}S_0 + S_1\overline{S_0}$$
$$= S_1 \oplus S_0$$

	S_1	S_0	T_A	T_B	S'_0
$\overline{S_1}\,\overline{S_0}\,\overline{T_A}$	0	0	0	X	1
	0	0	1	X	0
	0	1	X	X	0
$S_1\overline{S_0}\,\overline{T_B}$	1	0	X	0	1
	1	0	X	1	0
	1	1	X	X	0

$$S'_0 = \overline{S_1}\,\overline{S_0}\,\overline{T_A} + S_1\overline{S_0}\,\overline{T_B}$$

https://en.wikiversity.org/wiki/The_necessities_in_Computer_Design

Output Functions : L_{A1}, L_{A0}, L_{B0}, L_{B1}

S_1	S_2	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

00
01
10

$L_{A1} = S_1$
$L_{A0} = \overline{S_1} S_0$
$L_{B1} = \overline{S_1}$
$L_{B0} = S_1 S_0$

S_1	S_2	L_{A1}
0	0	0
0	1	0
1	0	**1**
1	1	**1**

$L_{A1} = S_1$

S_1	S_2	L_{A0}
0	0	0
0	1	**1**
1	0	0
1	1	0

$L_{A0} = \overline{S_1} S_0$

S_1	S_2	L_{B1}
0	0	**1**
0	1	**1**
1	0	0
1	1	0

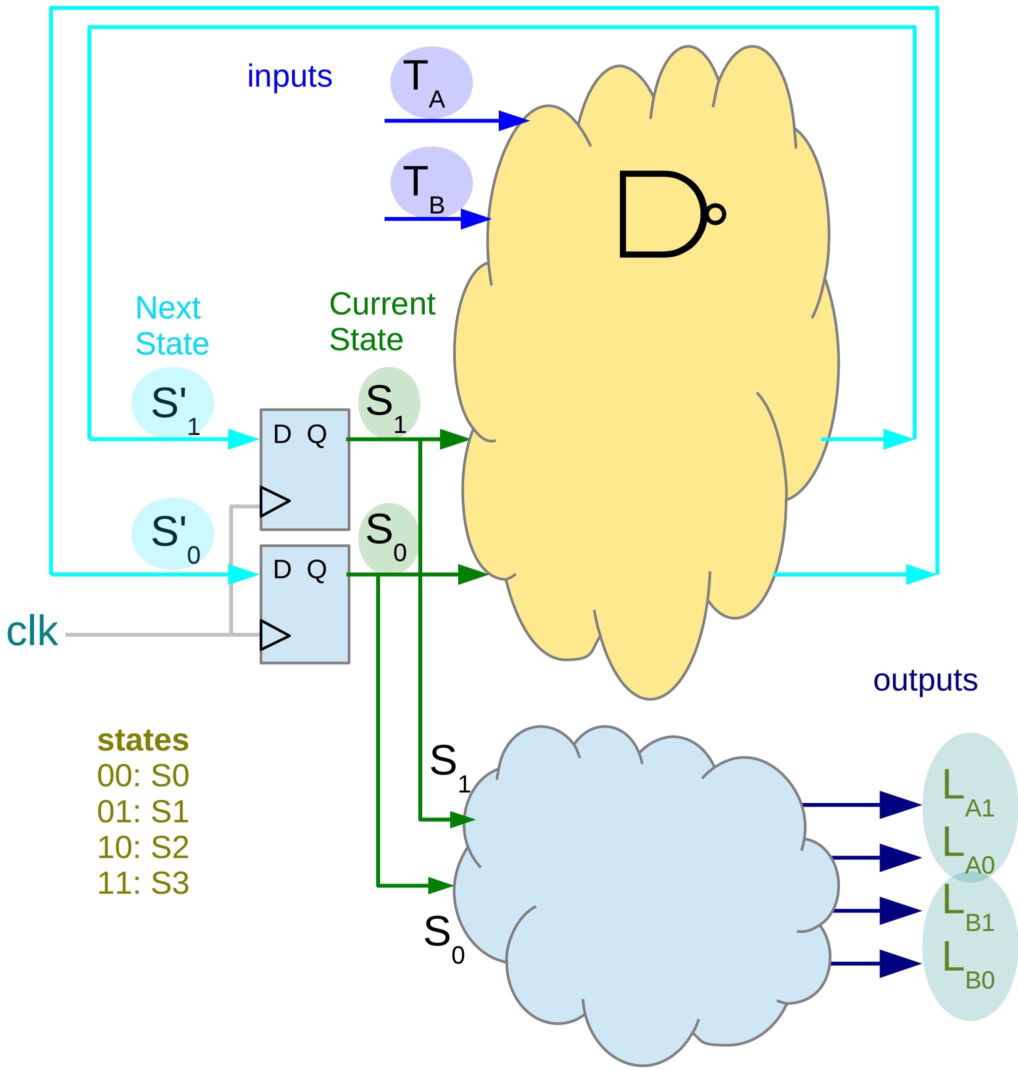
$L_{B1} = \overline{S_1}$

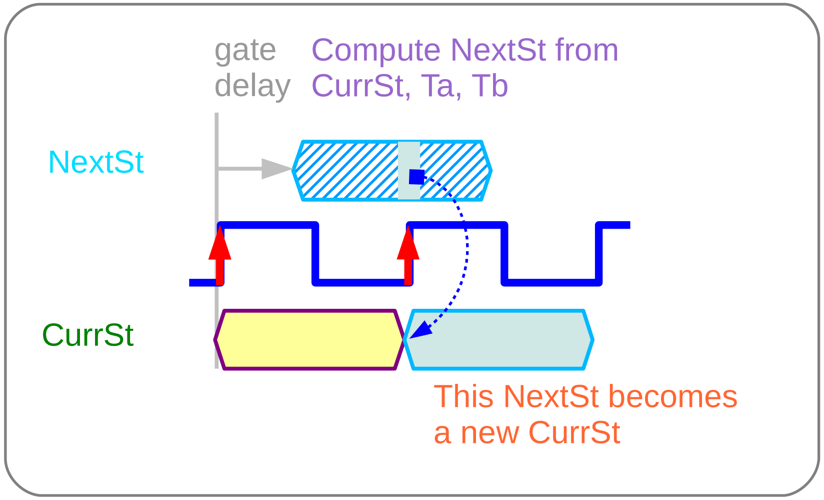
S_1	S_2	L_{B0}
0	0	0
0	1	0
1	0	0
1	1	**1**

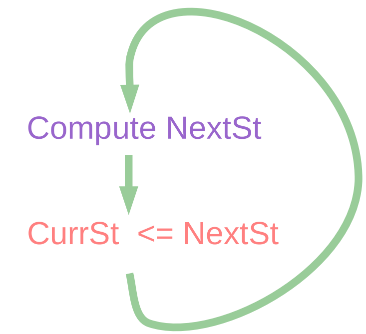
$L_{B0} = S_1 S_0$

https://en.wikiversity.org/wiki/The_necessities_in_Computer_Design

Moore FSM



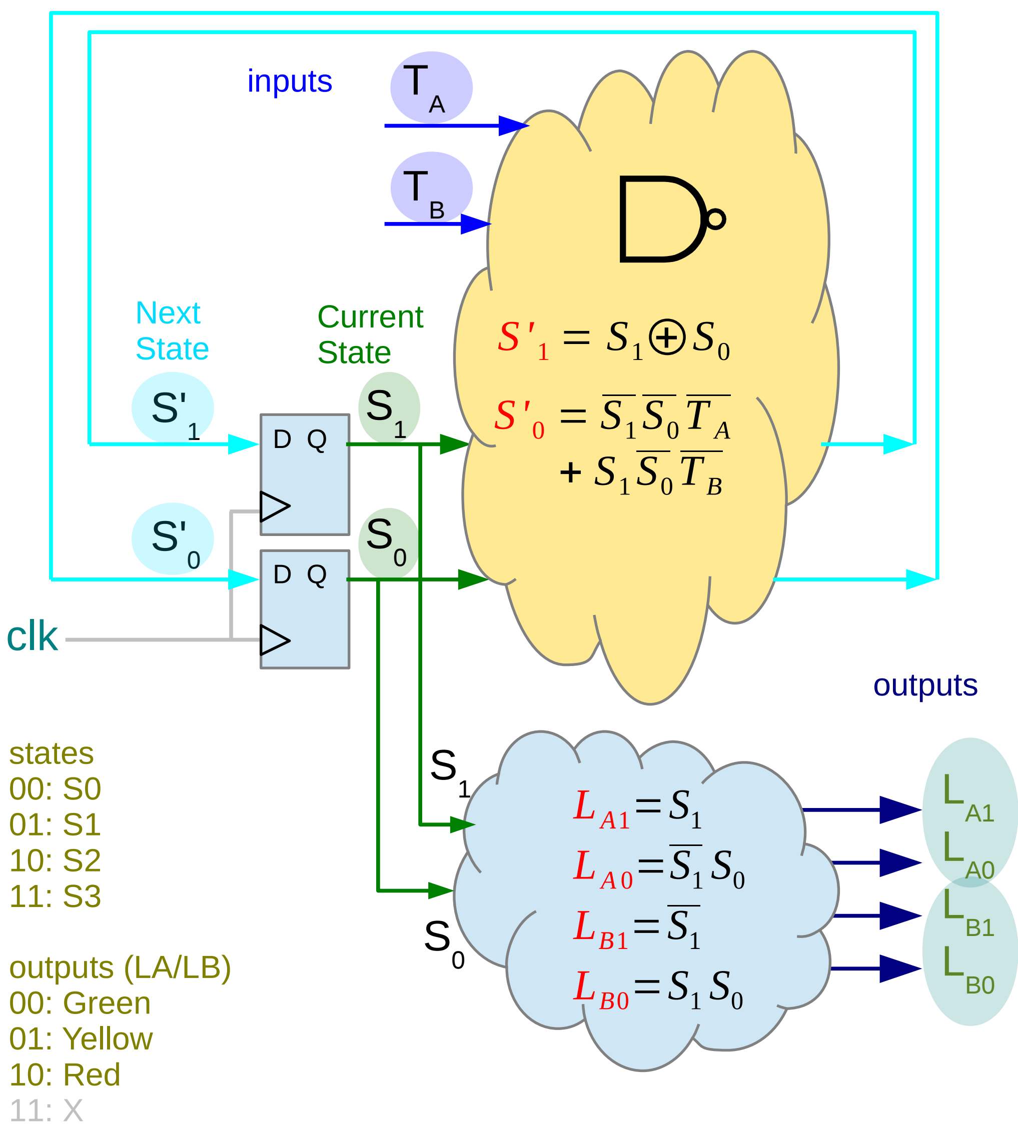




outputs (LA/LB)
00: Green
01: Yellow
10: Red
11: X

https://en.wikiversity.org/wiki/The_necessities_in_Computer_Design

Moore FSM Implementation



Inputs T_A T_B

Current State S_1 S_0

Next States

$$S'_1 = S_1 \oplus S_0$$

$$S'_0 = \overline{S_1}\,\overline{S_0}\,\overline{T_A} + S_1\,\overline{S_0}\,\overline{T_B}$$

Current State S_1 S_0

Outputs

$L_{A1} = S_1$ $\quad L_{B1} = \overline{S_1}$

$L_{A0} = \overline{S_1}\,S_0$ $\quad L_{B0} = S_1\,S_0$

https://en.wikiversity.org/wiki/The_necessities_in_Computer_Design

Next State Functions S_1' and S_2'

S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

$$S'_1 = S_1 + S_0$$

$$S'_0 = \overline{S_1}\,\overline{S_0}\,\overline{T_A} + S_1\,\overline{S_0}\,\overline{T_B}$$

https://en.wikiversity.org/wiki/The_necessities_in_Computer_Design

Current State $(S_1 S_0)$

FSM Inputs $(T_A T_B)$

Next State $(S_1 S_0)$

$$\{00, 01, 10, 11\} \times \{00, 01, 10, 11\} \rightarrow \{00, 01, 10, 11\}$$

Cartesian Product

S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	0	0	1
0	0	0	1	0	1
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	1	0
0	1	0	1	1	0
0	1	1	0	1	0
0	1	1	1	1	0
1	0	0	0	1	1
1	0	0	1	1	0
1	0	1	0	1	1
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	0
1	1	1	1	0	0

Current State $(S_1 S_0)$

FSM Inputs $(T_A T_B)$

Next State $(S_1 S_0)$

$$\{00, 01, 10, 11\} \times \{00, 01, 10, 11\} \rightarrow \{00, 01, 10, 11\}$$

Output Functions : L_{A1}, L_{A0}, L_{B1}, L_{B0}

S_1	S_2	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

G : 00
Y : 01
R : 10

$$L_{A1} = S_1$$
$$L_{A0} = \overline{S_1} S_0$$
$$L_{B1} = \overline{S_1}$$
$$L_{B0} = S_1 S_0$$

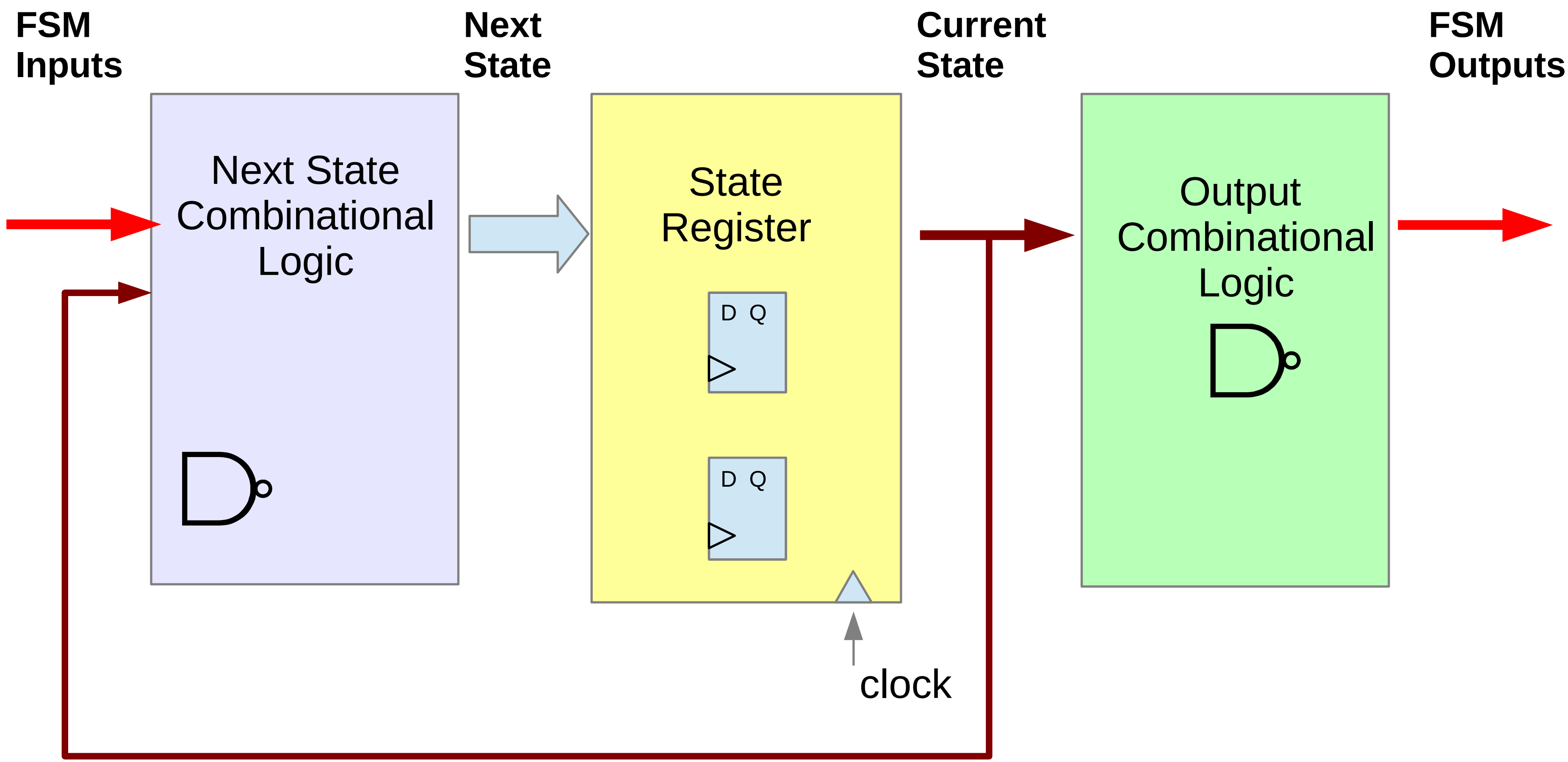
https://en.wikiversity.org/wiki/The_necessities_in_Computer_Design

Current State $(S_1 S_0)$

FSM Output $(L_{A1,} L_{A0,} L_{B1,} L_{B0})$

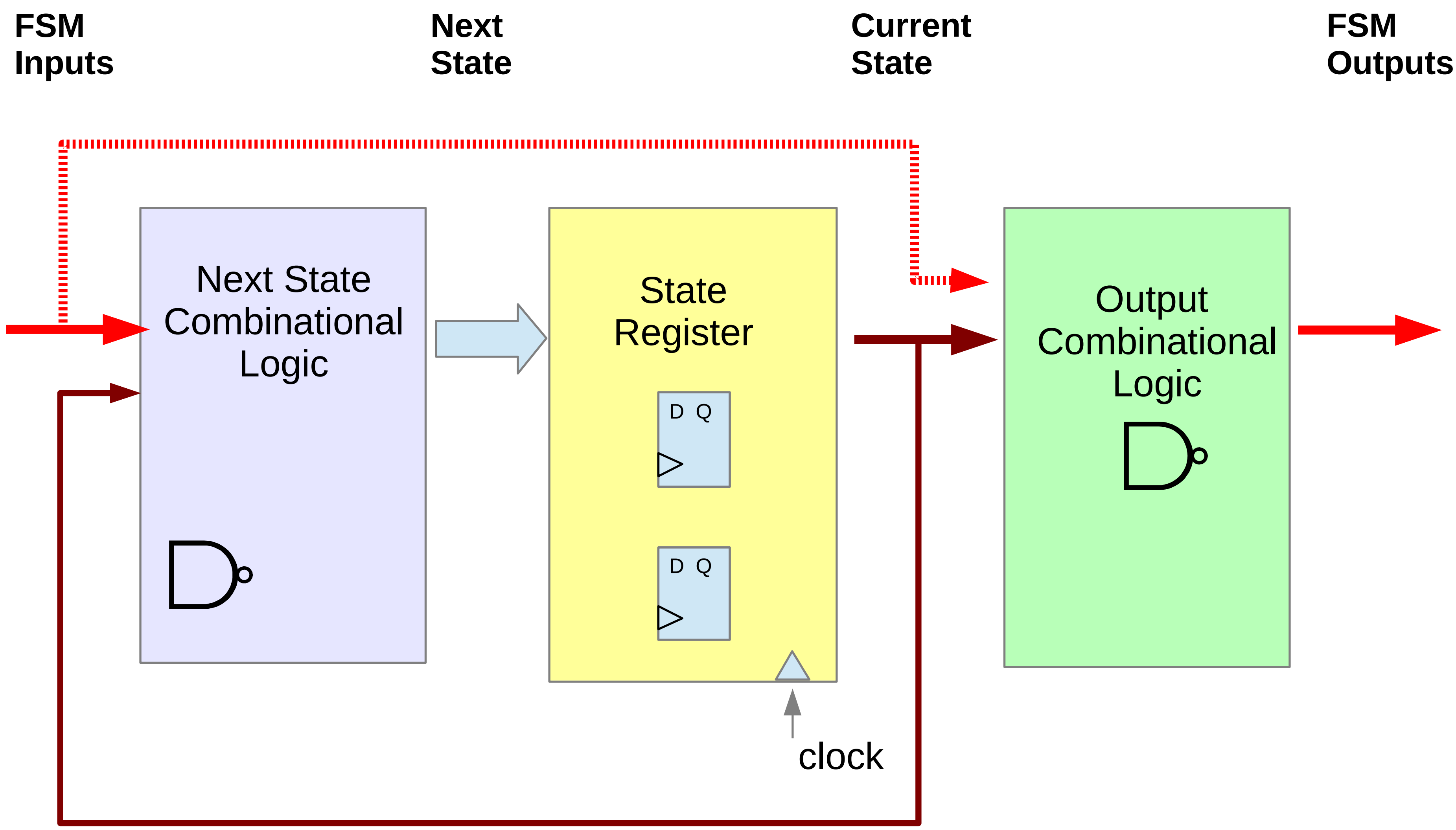
$\{00, 01, 10, 11\}$ → $\{0010, 0110, 1000, 1001\}$

Moore FSM



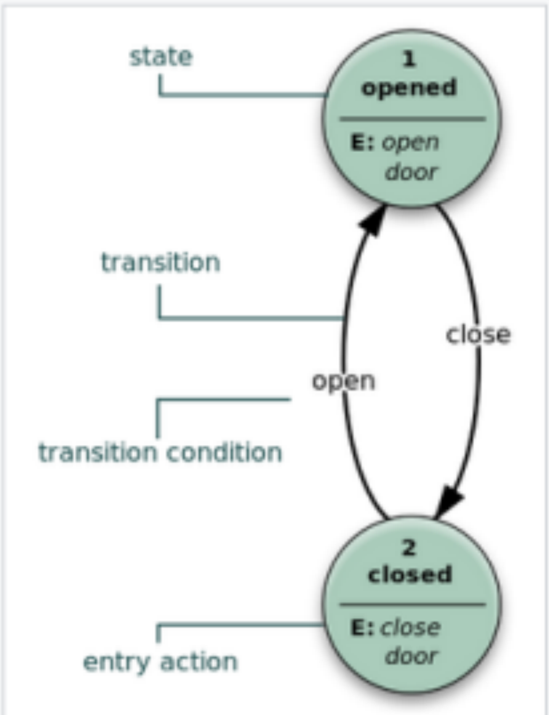
https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

Mealy FSM

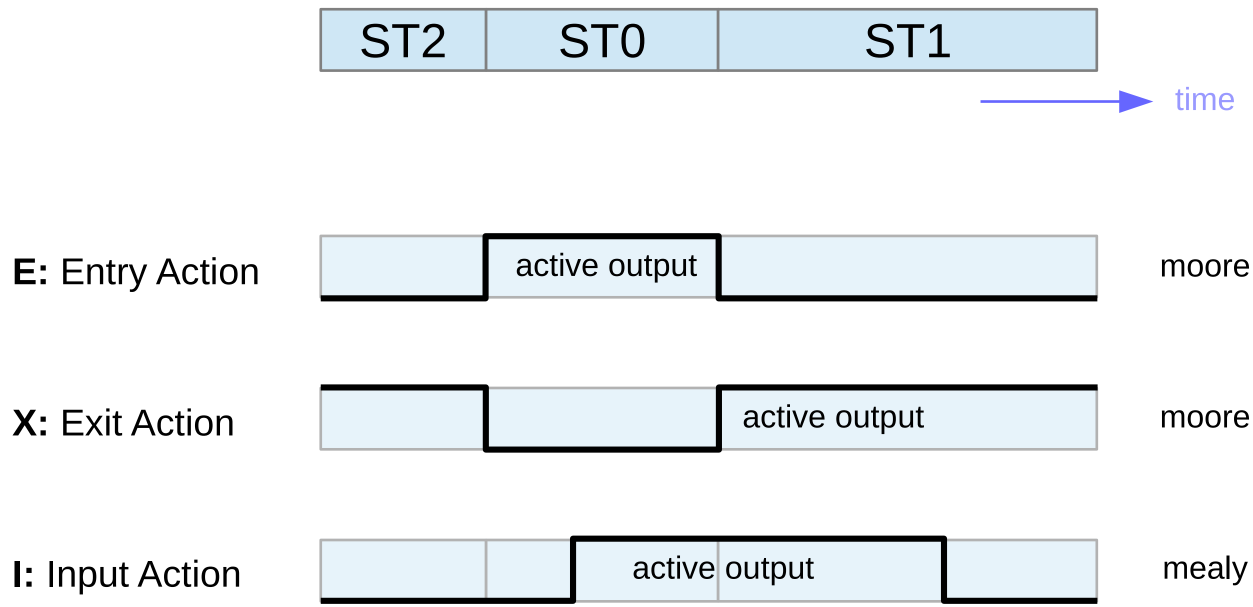


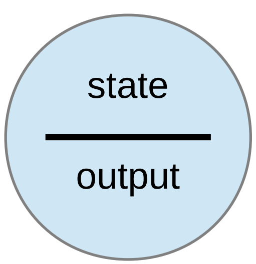
https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

State Diagram



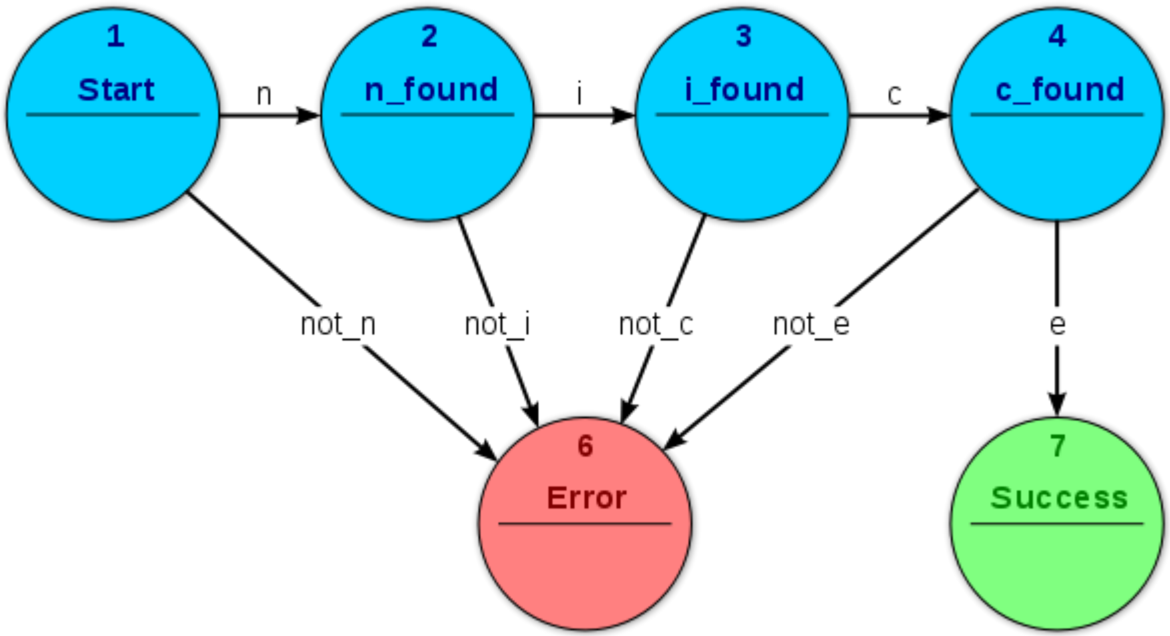
A state diagram for a door that can only be opened and closed





https://en.wikipedia.org/wiki/Finite-state_machine

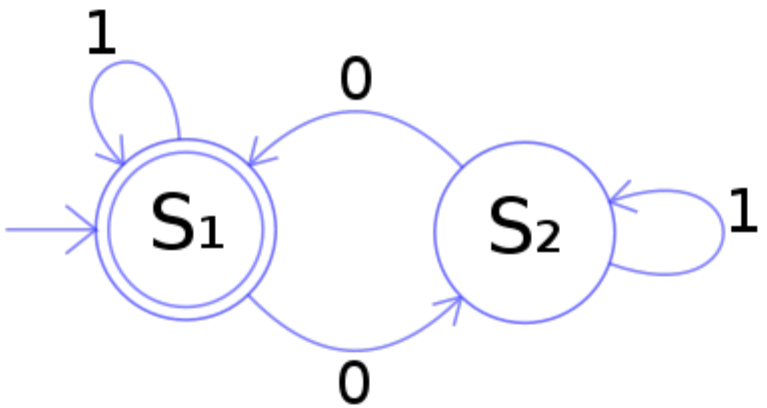
Acceptors



Acceptor FSM: parsing the string "nice"

https://en.wikipedia.org/wiki/Finite-state_machine

Recognizers



Representation of a finite-state machine;
determines whether a binary number has
an **even** number of **0s**,
where $\mathbf{S_1}$ is an **accepting state**.

https://en.wikipedia.org/wiki/Finite-state_machine

Classifiers

A **classifier** is a generalization of
a finite state machine that,
similar to an acceptor,
produces a single output on termination
but has more than two **terminal states**

https://en.wikipedia.org/wiki/Finite-state_machine

Transducers

Transducers generate **output** based on a given **input** and/or a **state** using actions.

They are used for control applications and in the field of computational linguistics.

https://en.wikipedia.org/wiki/Finite-state_machine

Acceptors, Recognizers, Transducers

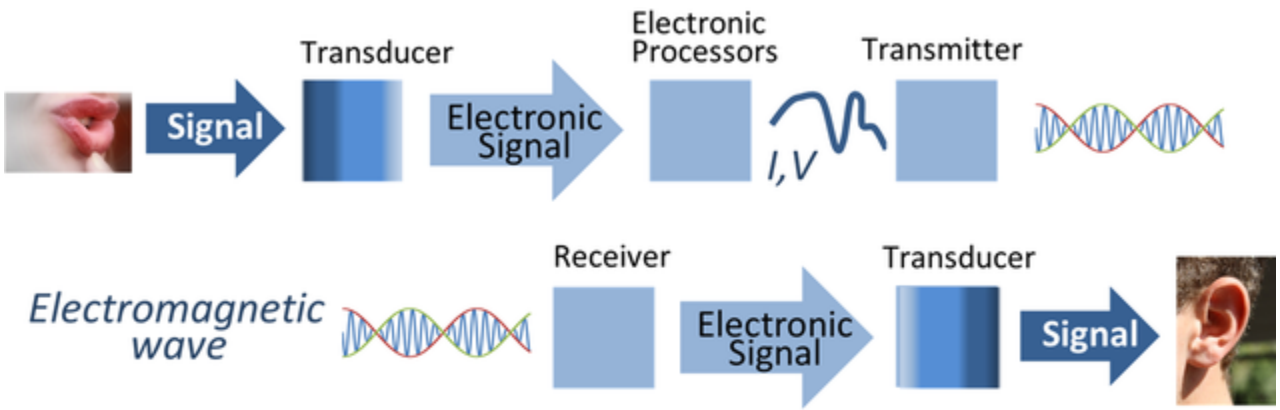
acceptors: either accept the input or not

recognizers: either recognize the input

transducers: generate output from given input

https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html

General Transducers



Transducers are used in electronic communications systems to convert signals of various physical forms to electronic signals, and vice versa. In this example, the first transducer could be a **microphone**, and the second transducer could be a **speaker**.

https://en.wikipedia.org/wiki/Transducer

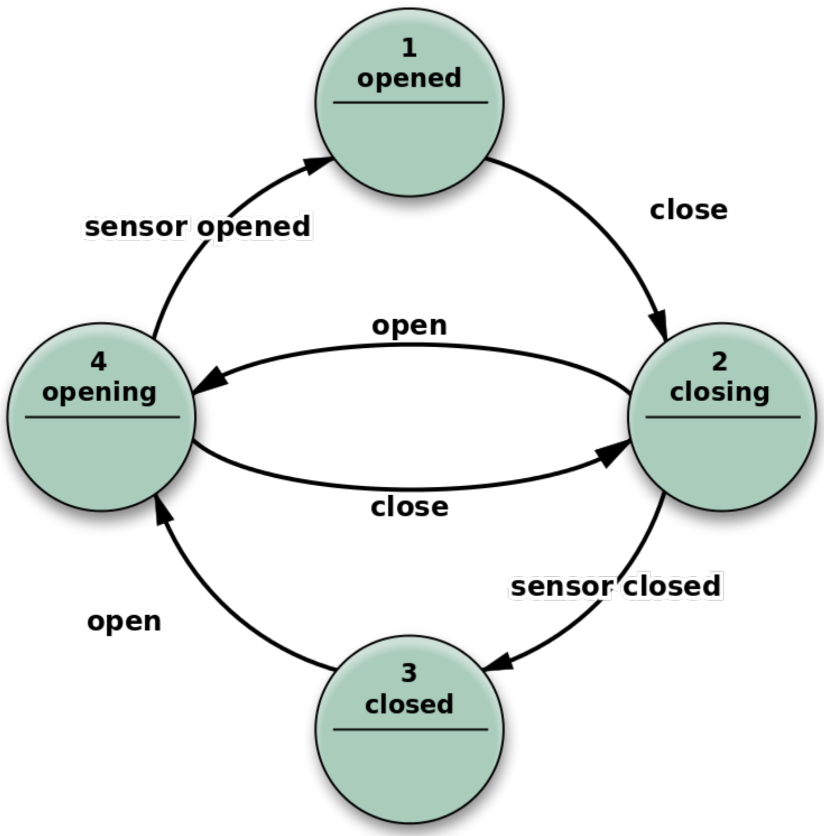


Fig. 6 Transducer FSM: Moore model example

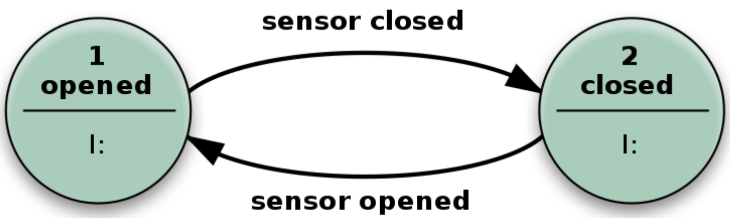


Fig. 7 Transducer FSM: Mealy model example

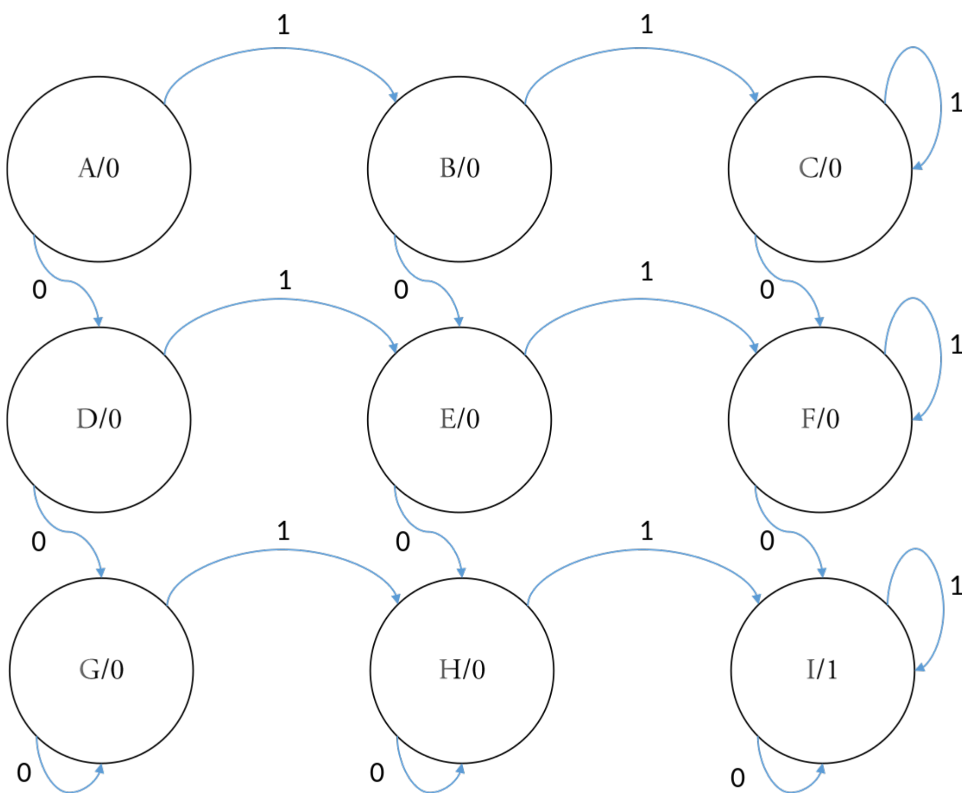
There are two **input actions** (I:):

"start motor to close the door
if command_close arrives"

"start motor in the other direction to open the door
if command_open arrives".

https://en.wikipedia.org/wiki/Finite-state_machine

Moore machine example

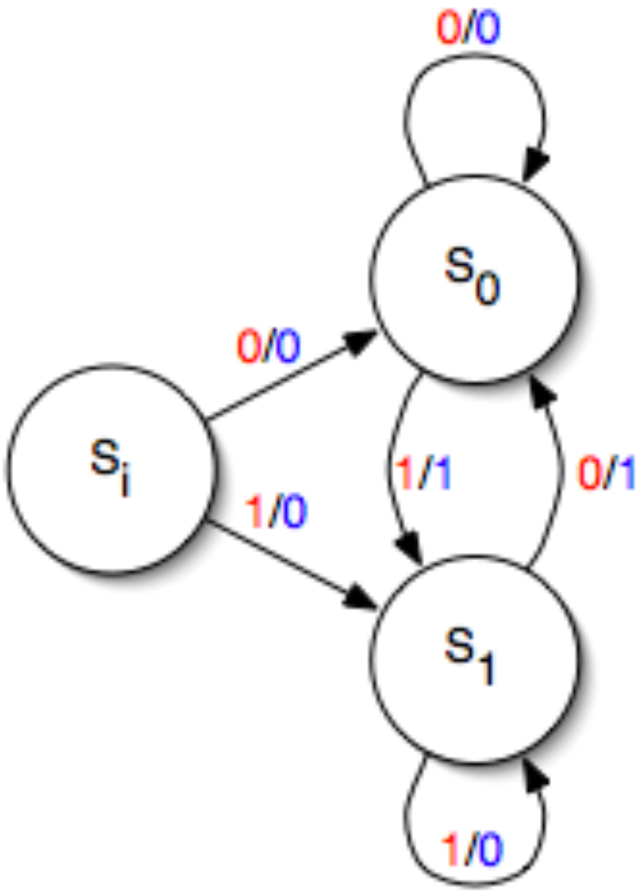


Current state	Input	Next state	Output
A	0	D	0
	1	B	
B	0	E	0
	1	C	
C	0	F	0
	1	C	
D	0	G	0
	1	E	
E	0	H	0
	1	F	
F	0	I	0
	1	F	
G	0	G	0
	1	H	
H	0	H	0
	1	I	
I	0	I	1
	1	I	

output does <u>not</u> <u>depend</u> on **inputs**

https://en.wikipedia.org/wiki/Moore_machine

Mealy machine



input / output

output does <u>depend</u> on **inputs**

https://en.wikipedia.org/wiki/Mealy_machine

Mathematical Model – transducers (1)

A **finite-state transducer** is a sextuple (**Σ**, **Γ**, **S**, **s_0**, **δ**, **ω**), where:

- **Σ** is the **input** alphabet (a finite non-empty set of symbols).
- **Γ** is the **output** alphabet (a finite, non-empty set of symbols).
- **S** is a finite, non-empty set of **states**.
- **s_0** is the **initial** state, an element of S.
- **δ** is the **state-transition function**: **δ : S × Σ → S**
- **ω** is the **output function**.
  - **Moore** machine : **ω : S → Γ**
  - **Mealy** machine : **ω : S × Σ → Γ**

$(\Sigma, \Gamma, S, s_0, \delta, \omega)$

(I, O, S, f, g, σ)

https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

Mathematical Model – transducers (2)

If the **output** function is a function of a **state** and **input** alphabet ($\boldsymbol{\omega : S \times \Sigma \rightarrow \Gamma}$) that definition corresponds to the **Mealy model**, and can be modelled as a **Mealy machine**.

If the **output** function depends only on a **state** ($\boldsymbol{\omega : S \rightarrow \Gamma}$) that definition corresponds to the **Moore model**, and can be modelled as a **Moore machine**.

A finite-state machine with <u>no output function</u> at all is known as a **semiautomaton** or **transition** system.

https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

Mathematical Models – acceptors

A **deterministic finite state machine** or **acceptor** deterministic finite state machine is a quintuple ($\boldsymbol{\Sigma}$, **S**, $\mathbf{s_0}$, $\boldsymbol{\delta}$, **F**), where:

output set {0, 1}

- $\boldsymbol{\Sigma}$ is the **input** alphabet (a finite, non-empty set of symbols).
- **S** is a finite, non-empty set of **states**.
- $\mathbf{s_0}$ is an **initial** state, an element of S.
- $\boldsymbol{\delta}$ is the **state-transition function**: $\boldsymbol{\delta : S \times \Sigma \rightarrow S}$
- **F** is the set of **final states**, a (possibly empty) subset of S.

output function $\boldsymbol{\omega}$

A set of accepted states

https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

Finite State Tranducers and Acceptors

limited finite state machine (Σ, S, δ)

finite state acceptor (Σ, S, s_0, δ, F)

finite-state transducer (Σ, Γ, S, s_0, δ, ω)

Finite State Automaton (FSA)

Finite State Machine (FSM)

Σ is the input alphabet (a finite non-empty set of symbols).
S is a finite, non-empty set of states.
δ is the state-transition function: $\delta : S \times \Sigma \rightarrow S$
s_0 is the initial state, an element of S.
F is the set of final states, a (possibly empty) subset of S.
Γ is the output alphabet (a finite, non-empty set of symbols).
ω is the output function.

References

[1] http://en.wikipedia.org/
[2]

Automata Theory (2A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice and Octave.

Automata

The word **automata** (the plural of **automaton**)

comes from the Greek word **αὐτόματα**,

which means "**self-acting**".

https://en.wikipedia.org/wiki/Automata_theory

Automata Theory

Automata theory is the study of
abstract machines and **automata**,
as well as the computational problems
that can be solved using them.

It is a theory in theoretical computer science
and discrete mathematics.

https://en.wikipedia.org/wiki/Automata_theory

Automata Informal description (1) – Inputs

An **automaton** runs
when it is given some sequence of **inputs**
in discrete (individual) time steps or steps.

word 1000100

An automaton processes one input
picked from a set of **symbols** or **letters**,
which is called an **alphabet**.

alphabet {0,1}

The symbols received by the automaton
as **input** at any step are
a finite sequence of **symbols** called **words**.

https://en.wikipedia.org/wiki/Automata_theory

Automata informal description (2) – States

An automaton has a finite set of **states**.

At each moment during a run of the automaton,
the automaton is in one of its **states**.

When the automaton receives new **input**
it moves to another **state** (or **transitions**)
based on a **function** that takes
the **current state** and **input symbol** as parameters.

This function is called the **transition function**.

https://en.wikipedia.org/wiki/Automata_theory

Automata informal description (3) – Stop

The **automaton**
reads the symbols of the **input word**
one after another and
transitions from **state** to **state**
according to the **transition function**
until the **word** is read completely.

Once the input **word** has been read,
the automaton is said to have stopped.

The state at which the automaton **stops**
is called the **final state**.

word	1000100

https://en.wikipedia.org/wiki/Automata_theory

Automata informal description (4) – Accept / Reject

Depending on the **final state**,
it's said that the automaton
either **accepts** or **rejects** an **input word**.

word 1000100

There is a **subset** of **states** of the automaton,
which is defined as the set of **accepting states**.

If the **final state** is an **accepting state**,
then the automaton **accepts** the **word**.

Otherwise, the **word** is **rejected**.

https://en.wikipedia.org/wiki/Automata_theory

Automata informal description (5) – Language

The set of **all the words accepted**
by an automaton is called the
"**language** of that automaton".

Any **subset** of the **language** of an automaton is
a language **recognized** by that automaton.

https://en.wikipedia.org/wiki/Automata_theory

Automata informal description (6) – Decision on inputs

an **automaton** is a mathematical object
that takes a word as **input**
and **decides** whether to **accept** it or **reject** it.

Since all computational problems are
reducible into the **accept/reject question** on **inputs**,
(all problem instances can be represented
in a finite length of symbols),
automata theory plays a crucial role
in computational theory.

https://en.wikipedia.org/wiki/Automata_theory

Automata Applications

Automata theory is closely related to **formal language** theory.

An automaton is a **finite representation** of a **formal language**
that may be an **infinite set**.

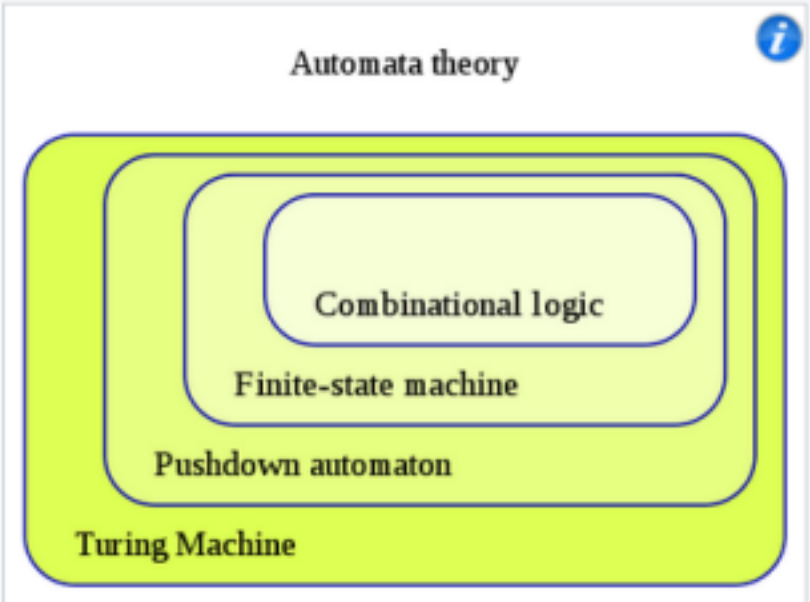
Automata are often classified by the **class** of **formal languages**
they can **recognize**, typically illustrated by the **Chomsky hierarchy**,
which describes the relations between various **languages** and kinds of
formalized **logic**.

Automata play a major role in
- theory of computation,
- compiler construction,
- artificial intelligence,
- parsing and
- formal verification.

https://en.wikipedia.org/wiki/Automata_theory

Class of Automata

- Combinational Logic
- Finite State Automaton (FSA)
- Pushdown Automaton (PDA)
- Turing Machine



Classes of automata
(Clicking on each layer will take you to an article on that subject)

https://en.wikipedia.org/wiki/Automata_theory

Class of Automata

Finite State Automaton (FSA)	Regular Language
Pushdown Automaton (PDA)	Context-Free Language
Turing Machine	Recursively Enumerable Language
Automaton	Formal Languages

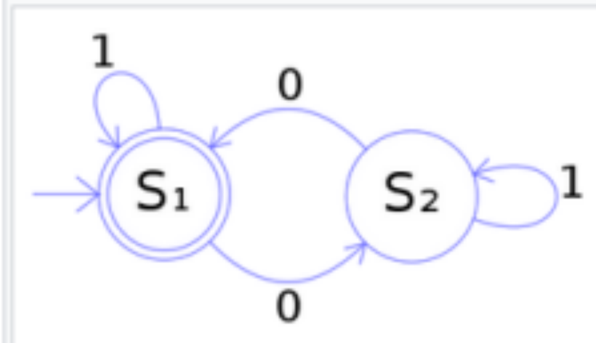
https://en.wikipedia.org/wiki/Automata_theory

Finite State Automaton

The figure at right illustrates a **finite-state machine**, which belongs to a well-known type of **automaton**.

This automaton consists of **states** (represented in the figure by circles) and **transitions** (represented by arrows).

As the automaton sees a **symbol** of **input**, it makes a **transition** (or jump) to another **state**, according to its **transition function**, which takes the **current state** and the recent **symbol** as its **inputs**.



The study of the mathematical properties of such automata is automata theory. The picture is a visualization of an automaton that recognizes strings containing an even number of *0*s. The automaton starts in state *S1*, and transitions to the non-accepting state *S2* upon reading the symbol *0*. Reading another *0* causes the automaton to transition back to the accepting state *S1*. In both states the symbol *1* is ignored by making a transition to the current state.

https://en.wikipedia.org/wiki/Automata_theory

Pushdown Automaton (1)

a type of automaton that employs a **stack**.

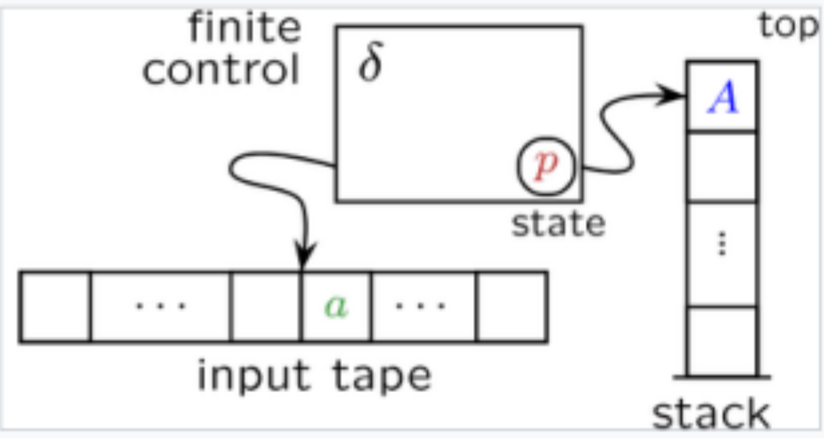
The term "pushdown" refers to the fact that
the stack can be regarded as being "pushed down"
like a tray dispenser at a cafeteria,
since the operations never work on elements
other than the **top element**.

A **stack automaton**, by contrast, does allow
access to and operations on deeper elements.

https://en.wikipedia.org/wiki/Automata_theory

Pushdown Automaton (2)

a pushdown automaton (PDA) is
a type of automaton that employs a stack



A diagram of a pushdown automaton

https://en.wikipedia.org/wiki/Pushdown_automaton

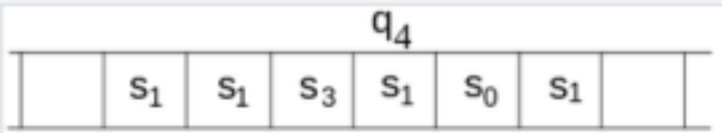
Turing Machine (1)

A **Turing machine** is
a mathematical **model** of computation
that defines an **abstract machine**,
which manipulates **symbols** on a strip of **tape**
according to a **table** of **rules**.

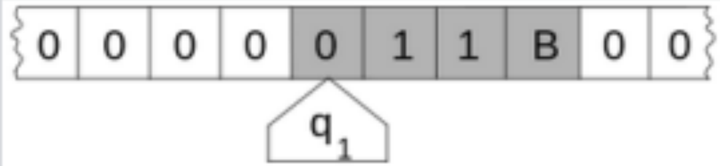
Despite the model's simplicity,
given any computer **algorithm**,
a **Turing machine** capable of **simulating**
that algorithm's logic can be constructed.

https://en.wikipedia.org/wiki/Automata_theory

Turing Machine (2)



The head is always over a particular square of the tape; only a finite stretch of squares is shown. The instruction to be performed (q_4) is shown over the scanned square. (Drawing after Kleene (1952) p. 375.)



Here, the internal state (q_1) is shown inside the head, and the illustration describes the tape as being infinite and pre-filled with "0", the symbol serving as blank. The system's full state (its *complete configuration*) consists of the internal state, any non-blank symbols on the tape (in this illustration "11B"), and the position of the head relative to those symbols including blanks, i.e. "011B". (Drawing after Minsky (1967) p. 121.)

https://en.wikipedia.org/wiki/Turing_machine

1. Definition of Finite State Automata

A deterministic finite automaton is represented formally by a 5-tuple <**Q**, **Σ**, **δ**, q_0, F>, where:

- **Q** is a finite set of **states**.
- **Σ** is a finite set of **symbols**, called the **alphabet** of the automaton.
- **δ** is the **transition function**, that is, **δ: Q × Σ → Q**.
- q_0 is the **start state**, that is, the state of the automaton before any input has been processed, where $q_0 \in$ **Q**.
- F is a set of **states** of **Q** (i.e. F⊆**Q**) called **accept states**.

https://en.wikipedia.org/wiki/Automata_theory

2. Deterministic Pushdown Automaton

A **PDA** is formally defined as a 7-tuple:

$M = (\mathbf{Q}, \mathbf{\Sigma}, \Gamma, \boldsymbol{\delta}, \mathbf{q_0}, Z, F)$ where

- **Q** is a finite set of **states**
- **Σ** is a finite set which is called the **input alphabet**
- **Γ** is a finite set which is called the **stack alphabet**
- **δ** is a finite subset of $\mathbf{Q}\times(\mathbf{\Sigma} \cup \{\varepsilon\})\times\mathbf{\Gamma}\times\mathbf{Q}\times\mathbf{\Gamma}^*$, the **transition relation**.
- $\mathbf{q_0} \in \mathbf{Q}$ is the **start state**
- $Z \in \mathbf{\Gamma}$ is the **initial stack symbol**
- $F \subseteq \mathbf{Q}$ is the set of **accepting states**

https://en.wikipedia.org/wiki/Pushdown_automaton

Turing machine as a 7-tuple M = (**Q**, **Γ**, **b**, **Σ**, **δ**, $\mathbf{q_0}$, **F**) where \ set minus

- **Q** is a finite, non-empty set of **states**;
- **Γ** is a finite, non-empty set of **tape alphabet symbols**;
- **b** ∈ **Γ** is the **blank symbol**
- **Σ** ⊆ **Γ** \ { b } is the set of **input symbols** in the initial tape contents;
- $\mathbf{q_0}$ ∈ **Q** is the **initial state**;
- **F** ⊆ **Q** is the set of **final states** or **accepting states**.
- **δ** : (**Q** \ **F**) × **Γ** → **Q** × **Γ** × { L , R } is **transition function**, where **L** is **left shift**, **R** is **right shift**.

The initial tape contents is said to be accepted by M if it eventually halts in a state from F .

https://en.wikipedia.org/wiki/Turing_machine

FSA, PDA, Turing Machine

Deterministic Finite State Automaton	($\mathbf{Q}$, $\mathbf{\Sigma}$, $\boldsymbol{\delta}$, $\mathbf{q_0}$, $\mathbf{F}$)
Deterministic Pushdown Automaton	($\mathbf{Q}$, $\mathbf{\Sigma}$, $\mathbf{\Gamma}$, $\boldsymbol{\delta}$, $\mathbf{q_0}$, $\mathbf{Z}$, $\mathbf{F}$)
Turing machine	($\mathbf{Q}$, $\mathbf{\Gamma}$, $\mathbf{b}$, $\mathbf{\Sigma}$, $\boldsymbol{\delta}$, $\mathbf{q_0}$, $\mathbf{F}$)

$\mathbf{\Sigma}$ is the input alphabet (a finite non-empty set of symbols).
$\mathbf{Q}$ is a finite, non-empty set of states.
$\boldsymbol{\delta}$ is the state-transition function: $\boldsymbol{\delta} : \mathbf{S} \times \mathbf{\Sigma} \rightarrow \mathbf{S}$
$\mathbf{s_0}$ is the initial state, an element of S.
$\mathbf{F}$ is the set of final states, a (possibly empty) subset of S.
$\mathbf{\Gamma}$ is a finite set which is called the **stack alphabet**
$\mathbf{Z} \in \mathbf{\Gamma}$ is the **initial stack symbol**
$\mathbf{\Gamma}$ is a finite, non-empty set of **tape alphabet symbols**;
$\mathbf{b} \in \mathbf{\Gamma}$ is the **blank symbol**

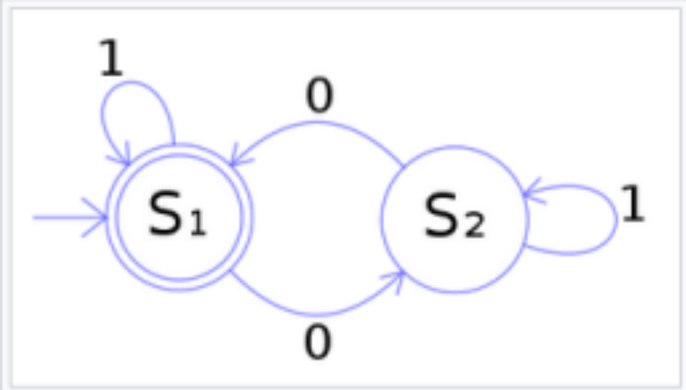
Deterministic Finite State Automaton (FSA)

Deterministic Finite Automaton Example (1)

The following example is of a DFA M, with a binary alphabet, which requires that the input contains an even number of 0s.

M = (Q, Σ, δ, q0, F) where
- Q = {S1, S2},
- Σ = {0, 1},
- q0 = S1,
- F = {S1}, and
- δ is defined by the following state transition table:

	0	1
S_1	S_2	S_1
S_2	S_1	S_2



The state diagram for *M*

(Σ, S, s_0, δ, F)

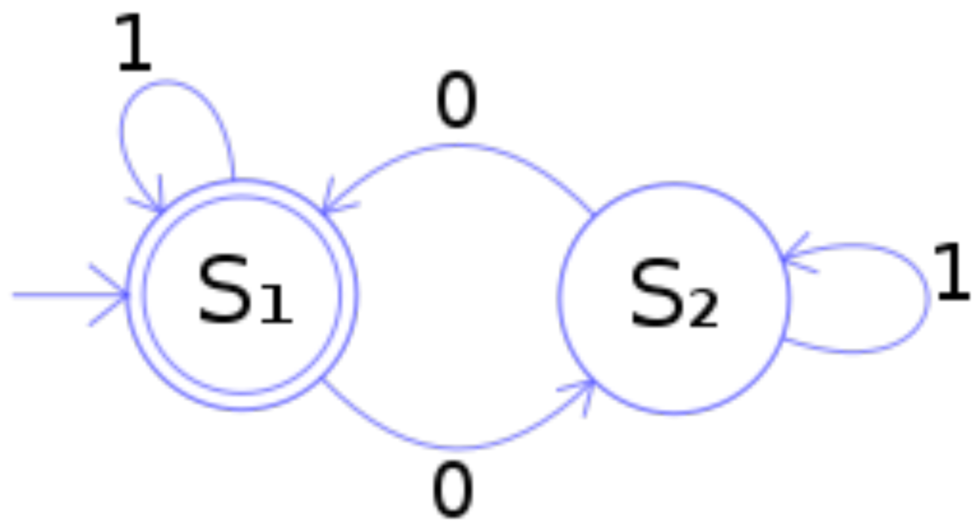
(Q, Σ, δ, q_0, F)

https://en.wikipedia.org/wiki/Deterministic_finite_automaton

Deterministic Finite Automaton Example (2)

State Transition Table

Input / State	1	0
S_1	S_1	S_2
S_2	S_2	S_1



$$\{S_{1,} S_2\} \quad \times \quad 0,1 \quad \rightarrow \quad \{S_{1,} S_2\}$$

https://en.wikipedia.org/wiki/State_transition_table

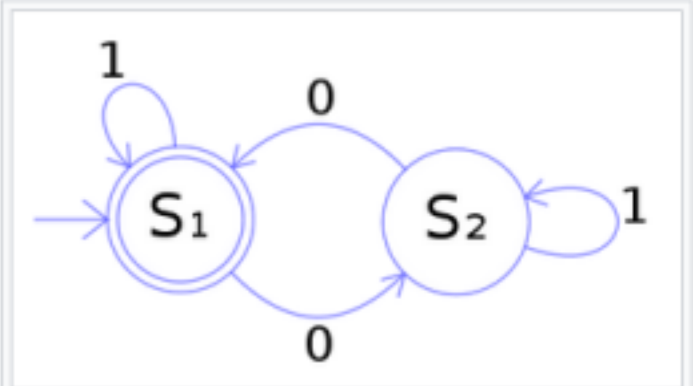
Deterministic Finite Automaton Example (3)

The **state S1** represents that there has been an even number of 0s in the input so far, while **S2** signifies an odd number.

A **1** in the input does not change the state of the automaton.

When the input ends, the state will show whether the input contained an even number of **0**s or not.

If the input did contain an even number of **0**s, M will finish in **state S1**, an accepting state, so the input string will be accepted.



The state diagram for *M*

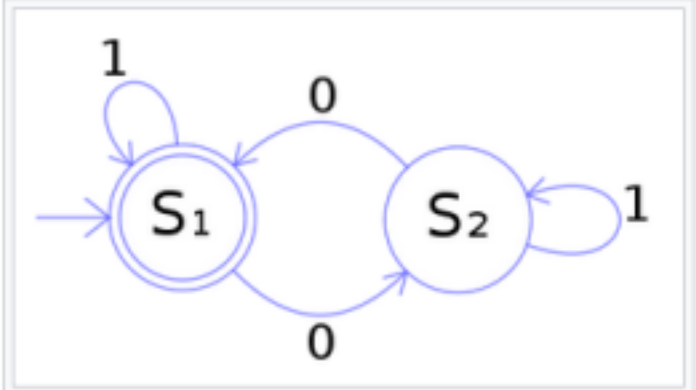
https://en.wikipedia.org/wiki/Deterministic_finite_automaton

Deterministic Finite Automaton Example (4)

The language recognized by M is
the regular language given
by the regular expression
((1*) 0 (1*) 0 (1*))*,

where "*" is the Kleene star,
e.g., **1*** denotes any number
(possibly zero) of consecutive **ones**.

zero or more



The state diagram for *M*

https://en.wikipedia.org/wiki/Deterministic_finite_automaton

References

[1] http://en.wikipedia.org/
[2]

Formal Language (3A)

- Regular Language

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice and Octave.

Formal Language

a **formal language** is
a set of **strings** of **symbols**
together with a set of **rules**
that are specific to it.

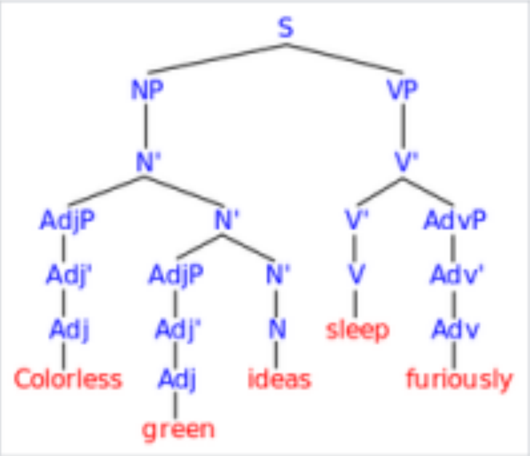
https://en.wikipedia.org/wiki/Formal_language

Alphabet and Words

The **alphabet** of a formal language is
the set of **symbols**, **letters**, or **tokens**
from which the **strings** of the language may be formed.

The **strings** formed from this alphabet
are called **words**

the **words** that belong to a particular formal language
are sometimes called **well-formed words**
or **well-formed formulas**.



Structure of a syntactically well-formed, although nonsensical, English sentence (historical example from Chomsky 1957).

https://en.wikipedia.org/wiki/Formal_language

Formal Language

A **formal language** (**formation rule**)
is often defined by means of
a **formal grammar**
  such as a **regular grammar** or
  context-free grammar,

https://en.wikipedia.org/wiki/Formal_language

Formal Language and Natural Language

The field of **formal language** theory studies
primarily the purely **syntactical aspects** of such languages—
that is, their internal **structural patterns**.

Formal language theory sprang out of linguistics,
as a way of understanding the **syntactic regularities**
of **natural languages**.

formalized versions of subsets of **natural languages**
in which the words of the language represent **concepts**
that are associated with particular **meanings** or **semantics**.

https://en.wikipedia.org/wiki/Formal_language

Formal Language and Programming Languages

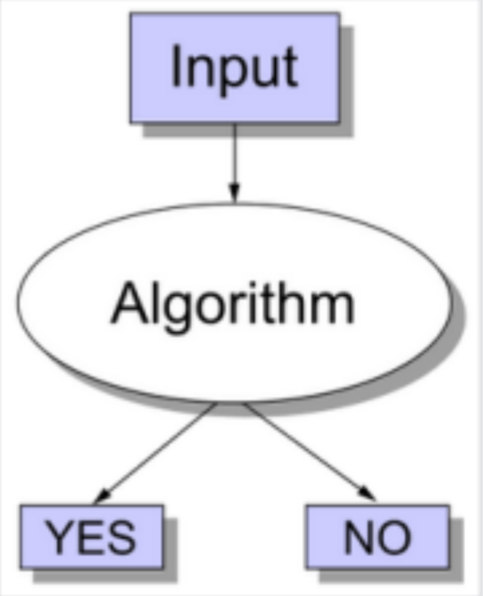
In computer science, formal languages are used among others as the basis for defining the **grammar** of **programming languages**

https://en.wikipedia.org/wiki/Formal_language

Formal Language and Complexity Theory

In computational **complexity theory**,
decision problems are typically defined
as formal languages, and

complexity classes are defined
as the sets of the formal languages
that can be parsed by machines
with limited computational power.



A *decision problem* has only two possible outputs (*yes* or *no*) on any input.

These inputs can be natural numbers,
but may also be values of some other kind,
such as strings over the binary alphabet {0,1}
or over some other finite set of symbols.
The subset of strings for which the problem
returns "yes" is a formal language,
and often decision problems are defined
in this way as formal languages.

https://en.wikipedia.org/wiki/Formal_language
https://en.wikipedia.org/wiki/Decision_problem

Formal Language and Logic / Mathematics

In **logic** and the foundations of **mathematics**,
formal languages are used to represent
the **syntax** of **axiomatic systems**,
and **mathematical formalism** is
the philosophy that all of mathematics
can be reduced to the **syntactic manipulation**
of formal languages in this way.

https://en.wikipedia.org/wiki/Formal_language

Alphabet

An **alphabet** can be any set
think a **character set** such as ASCII.
the elements of an alphabet are called its **letters**.
    an **infinite** number of elements
    a **finite** number of elements

https://en.wikipedia.org/wiki/Formal_language

Words over an alphabet

A **word** over an **alphabet** can be
any finite sequence (i.e., string) of **letters**.

The set of all words over an **alphabet Σ**
is usually denoted by **Σ*** (using the **Kleene star**).

The **length** of a word is the number of letters
only one word of **length 0**, the **empty word** (e / ε / λ or even Λ)
By **concatenation** one can combine two words to form a new word

in logic, the **alphabet** is also known as the **vocabulary**
and **words** are known as **formulas** or **sentences**;

the **letter**/**word** metaphor	: formal language
a **word**/**sentence** metaphor	: logic

https://en.wikipedia.org/wiki/Formal_language

Kleene star

Given a set V define

$V_0 = \{\varepsilon\}$ (the language consisting only of the empty string),
$V_1 = V$

and define recursively the set

$V_{i+1} = \{ wv : w \in V_i \text{ and } v \in V \}$ for each i>0.

$$V^* = \bigcup_{i \in \mathbb{N}} V_i = \{\varepsilon\} \cup V \cup V_2 \cup V_3 \cup V_4 \cup \ldots.$$

*: zero or more

$$V^+ = \bigcup_{i \in \mathbb{N} \setminus \{0\}} V_i = V_1 \cup V_2 \cup V_3 \cup \ldots.$$

+: one or more

$\mathbb{N}^0 = \mathbb{N}_0 = \{0, 1, 2, \ldots\}$
$\mathbb{N}^* = \mathbb{N}^+ = \mathbb{N}_1 = \mathbb{N}_{>0} = \{1, 2, \ldots\}.$

https://en.wikipedia.org/wiki/Kleene_star

$\mathbb{N} = \{0, 1, 2, \ldots\}.$
$\mathbb{Z}^+ = \{1, 2, \ldots\}.$

Kleene star examples (1)

{"ab","c"}* = { ε, "ab", "c", "abab", "abc", "cab", "cc", "ababab", "ababc", "abcab", "abcc", "cabab", "cabc", "ccab", "ccc", ...}.

{"a", "b", "c"}+ = { "a", "b", "c", "aa", "ab", "ac", "ba", "bb", "bc", "ca", "cb", "cc", "aaa", "aab", ...}.

{"a", "b", "c"}* = { ε, "a", "b", "c", "aa", "ab", "ac", "ba", "bb", "bc", "ca", "cb", "cc", "aaa", "aab", ...}.

∅* = {ε}.

∅+ = ∅

∅* = { } = ∅,

https://en.wikipedia.org/wiki/Kleene_star

Kleene star examples (2)

{ab, c}* =
{ ε,
ab, c,
abab, abc, cab, cc,
ababab, ababc, abcab, abcc, cabab, cabc, ccab, ccc, … }

{a, b, c}+ =
{ a, b, c,
aa, ab, ac, ba, bb, bc, ca, cb, cc,
aaa, aab, aac, aba, abb, abc, aca, acb, acc, baa, bab, bac, bba, bbb, bbc, bca, bcb, bcc, … }

{a, b, c}*
{ ε
a, b, c,
aa, ab, ac, ba, bb, bc, ca, cb, cc,
aaa, aab, aac, aba, abb, abc, aca, acb, acc, baa, bab, bac, bba, bbb, bbc, bca, bcb, bcc, … }

https://en.wikipedia.org/wiki/Kleene_star

Kleene star examples (3)

regular expression
((1*) 0 (1*) 0 (1*))*,

(1*) = {ε, 1, 11, 111, …}

$$\begin{Bmatrix} e \\ 1 \\ 11 \\ 111 \\ \vdots \end{Bmatrix} 0 \begin{Bmatrix} e \\ 1 \\ 11 \\ 111 \\ \vdots \end{Bmatrix} 0 \begin{Bmatrix} e \\ 1 \\ 11 \\ 111 \\ \vdots \end{Bmatrix}$$

$$\begin{matrix} 00\{e,1,11,111,\cdots\} \\ 010\{e,1,11,111,\cdots\} \\ 0110\{e,1,11,111,\cdots\} \\ 01110\{e,1,11,111,\cdots\} \\ \vdots \end{matrix} \quad \begin{matrix} 100\{e,1,11,111,\cdots\} \\ 1010\{e,1,11,111,\cdots\} \\ 10110\{e,1,11,111,\cdots\} \\ 101110\{e,1,11,111,\cdots\} \\ \vdots \end{matrix} \quad \begin{matrix} 1100\{e,1,11,111,\cdots\} \\ 11010\{e,1,11,111,\cdots\} \\ 110110\{e,1,11,111,\cdots\} \\ 1101110\{e,1,11,111,\cdots\} \\ \vdots \end{matrix} \quad \begin{matrix} 11100\{e,1,11,111,\cdots\} \\ 111010\{e,1,11,111,\cdots\} \\ 1110110\{e,1,11,111,\cdots\} \\ 11101110\{e,1,11,111,\cdots\} \\ \vdots \end{matrix}$$

https://en.wikipedia.org/wiki/Kleene_star

Formal Language Definition

A **formal language L** over an **alphabet Σ** is
a **subset of Σ***, that is, a set of **words** over that alphabet.

Sometimes the sets of **words** are grouped into **expressions**,
whereas **rules** and **constraints** may be formulated
for the creation of '**well-formed expressions**'.

https://en.wikipedia.org/wiki/Formal_language

Formal Language Examples (1)

The following **rules** describe
a **formal language L**
over the **alphabet Σ** = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, = }:

- every nonempty string is in **L**
  - that does not contain "**+**" or "**=**"
  - does not start with "**0**"
- the string "**0**" is in **L**.
- a string containing "**=**" is in **L**
  - if and only if there is exactly one "**=**",
  - and it separates two valid strings of **L**.
- a string containing "**+**" but not "=" is in **L**
  - if and only if every "**+**" in the string separates two valid strings of **L**.
- no string is in **L** other than those implied by the previous rules.

https://en.wikipedia.org/wiki/Formal_language

Formal Language Examples (2)

Under these rules,
the string "**23+4=555**" is in L,
but the string "**=234=+**" is not.

This formal language expresses

- **natural numbers**,
- well-formed **additions**,
- and well-formed **addition equalities**,

but it expresses only what they look like (their **syntax**),
not what they mean (**semantics**).

for instance, nowhere in these rules is
there any indication that "0" means the number zero,
or that "+" means addition.

https://en.wikipedia.org/wiki/Formal_language

Formal Language Examples (3)

- $L = \Sigma^*$, the set of all **words** over Σ;
- $L = \{\text{"a"}\}^* = \{\text{"a"}^n\}$, where n ranges over the natural numbers

  and "a"^n means "a" repeated n times

  (this is the set of words consisting only of the symbol "a");
- the set of **syntactically correct** programs in a given programming language (the syntax of which is usually defined by a **context-free grammar**);
- the set of inputs upon which a certain **Turing machine** halts; or
- the set of **maximal strings** of alphanumeric ASCII characters on this line, i.e., the set {"the", "set", "of", "maximal", "strings", "alphanumeric", "ASCII", "characters", "on", "this", "line", "i", "e"}.

https://en.wikipedia.org/wiki/Formal_language

Formal Language Examples (4)

For instance, a **language** can be given as

- those strings generated by some **formal grammar**;
- those strings described or matched by a particular **regular expression**;
- those strings accepted by some automaton,

  such as a **Turing machine** or **finite state automaton**;
- those **strings** for which some **decision procedure** produces the answer YES.

  (an algorithm that asks a sequence of related YES/NO questions)

https://en.wikipedia.org/wiki/Formal_language

Formal Grammar Example

the alphabet consists of **a** and **b**,
the start symbol is **S**,
the **production rules**:

1. **S** → **aSb**
2. **S** → **ba**

then we start with **S**, and can choose a rule to apply to it.
Application of rule 1, the string **aSb**.
Another application of rule 1, the string **aaSbb**.
Application of rule 2, the string **aababb**

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aababb$$

The **language** of the grammar is then the infinite set

$$\{a^n bab^n \mid n \geq 0\} = \{ba, abab, aababb, aaababbb, \ldots\}$$

https://en.wikipedia.org/wiki/Formal_language

Syntax of Formal Grammars

a **grammar** G consists of the following components:

- A finite set **N** of **nonterminal symbols**,
  that is disjoint with the strings formed from G.
- A finite set **Σ** of **terminal symbols**
  that is disjoint from **N**.
- A finite set **P** of **production rules**,
  $$(\Sigma \cup N)^* N (\Sigma \cup N)^* \to (\Sigma \cup N)^*$$
- A distinguished symbol **S** ∈ **N** that is the **start symbol**,
  also called the **sentence symbol**.

A grammar is formally defined as the tuple (**N** ,**Σ**, **P**, **S**)

often called a rewriting system
or a phrase structure grammar

https://en.wikipedia.org/wiki/Formal_language

Terminal and Non-terminal Symbols

Terminal symbols are the **elementary symbols** of the language defined by a formal grammar.

Nonterminal symbols (or **syntactic variables**) are replaced by groups of **terminal symbols** according to the **production rules**.

A formal grammar includes a **start symbol**, a designated member of the set of **nonterminals** from which all the strings in the language may be derived by successive applications of the **production rules**.

In fact, the language defined by a grammar is precisely the set of **terminal strings** that can be so derived.

https://en.wikipedia.org/wiki/Terminal_and_nonterminal_symbols

Production Rules

$$(\Sigma \cup N)^* N (\Sigma \cup N)^* \to (\Sigma \cup N)^*$$

Head → Body

- each **production rule** maps from one string of symbols to another
- the first string (the "head") contains
  - an arbitrary number of symbols
  - provided at least one of them is a **nonterminal**. ***N***
- If the second string (the "body") consists solely of the **empty string**
  - i.e., that it contains no symbols at all
  - it may be denoted with a special notation (Λ , e or ϵ)

https://en.wikipedia.org/wiki/Formal_language

Grammar Examples (1)

Consider the grammar **G**
where **N = { S , B }**, **Σ = { a , b , c }**, **S** is the start symbol,
and **P** consists of the following production rules:

1. **S** → **aBSc**
2. **S** → **abc**
3. **Ba** → **aB**
4. **Bb** → **bb**

This grammar defines the language **L(G)** $= \{a^n b^n c^n \mid n \geq 1\}$
where a^n denotes a string of **n** consecutive **a**'s.

Thus, the language is the set of strings that consist of 1 or more **a**'s,
followed by the same number of **b**'s, followed by the same number of **c**'s.

https://en.wikipedia.org/wiki/Formal_language

Grammar Examples (2)

$$\boldsymbol{S} \underset{2}{\Rightarrow} \boldsymbol{abc}$$

$$\begin{aligned} \boldsymbol{S} &\underset{1}{\Rightarrow} \boldsymbol{aBSc} \\ &\underset{2}{\Rightarrow} aB\boldsymbol{abc}c \\ &\underset{3}{\Rightarrow} a\boldsymbol{aB}bcc \\ &\underset{4}{\Rightarrow} aa\boldsymbol{bb}cc \end{aligned}$$

$$\begin{aligned} \boldsymbol{S} \underset{1}{\Rightarrow} \boldsymbol{aBSc} &\underset{1}{\Rightarrow} aB\boldsymbol{aBSc}c \\ &\underset{2}{\Rightarrow} aBaB\boldsymbol{abc}cc \\ &\underset{3}{\Rightarrow} a\boldsymbol{aB}Babccc \underset{3}{\Rightarrow} aaB\boldsymbol{aB}bccc \underset{3}{\Rightarrow} aa\boldsymbol{aB}Bbccc \\ &\underset{4}{\Rightarrow} aaaB\boldsymbol{bb}ccc \underset{4}{\Rightarrow} aaa\boldsymbol{bb}bccc \end{aligned}$$

1. **S → aBSc**
2. **S → abc**
3. **Ba → aB**
4. **Bb → bb**

https://en.wikipedia.org/wiki/Formal_language

Context Free Grammars

Context-free grammars are those grammars
in which the left-hand side of each **production rule**
consists of only a single **nonterminal symbol**.

This restriction is non-trivial;
not all languages can be generated
by context-free grammars.

Those that can are called **context-free languages**.

S → aSb
S → ba

https://en.wikipedia.org/wiki/Terminal_and_nonterminal_symbols

Context Free Grammar Examples

The language **L(G) = $\{a^n b^n c^n \mid n \geq 1\}$** is <u>not</u> a **context-free language**

the grammar **G**

where **N = { S , B }**, **Σ = { a , b , c }**, **S** is the start symbol,

and **P** consists of the following production rules:

1. **S** → **aBSc**
2. **S** → **abc**
3. **Ba** → **aB**
4. **Bb** → **bb**

The language **$\{a^n b^n \mid n \geq 1\}$** is **context-free**

(at least 1 **a** followed by the same number of **b**)

the grammar G2 with **N = { S }**, **Σ = { a , b }**, **S** the start symbol,

and **P** the following production rules:

1. **S** → **a S b**
2. **S** → **a b**

https://en.wikipedia.org/wiki/Formal_language

Regular Expression Examples

.at	matches any three-character string ending with "at", including "hat", "cat", and "bat".
[hc]at	matches "hat" and "cat".
[^b]at	matches all strings matched by .at except "bat".
[^hc]at	matches all strings matched by .at other than "hat" and "cat".
^[hc]at	matches "hat" and "cat", but only at the beginning of the string or line.
[hc]at$	matches "hat" and "cat", but only at the end of the string or line.
\[.\]	matches any single character surrounded by "[" and "]" since the brackets are escaped, for example: "[a]" and "[b]".
s.*	matches s followed by zero or more characters, for example: "s" and "saw" and "seed".

[hc]?at	matches "at", "hat", and "cat".
[hc]*at	matches "at", "hat", "cat", "hhat", "chat", "hcat", "cchchat", ...
[hc]+at	matches "hat", "cat", "hhat", "chat", "hcat", "cchchat",..., but not "at".
cat|dog	matches "cat" or "dog".

https://en.wikipedia.org/wiki/Regular_expression

Chomsky's four types of grammars

Grammar	Languages	Automaton	Production rules (constraints)*
Type-0	Recursively enumerable	Turing machine	$\alpha \rightarrow \beta$ (no restrictions)
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow \gamma$
Type-3	Regular	Finite state automaton	$A \rightarrow \mathrm{a}$ and $A \rightarrow \mathrm{a}B$

* Meaning of symbols:

a = terminal
α = string of terminals, non-terminals, or empty
β = string of terminals, non-terminals, or empty
γ = string of terminals, non-terminals, never empty
A = non-terminal
B = non-terminal

https://en.wikipedia.org/wiki/Chomsky_hierarchy

Type-0 grammars

Unrestricted grammar

Type-0 grammars include all **formal grammars**.

They generate exactly all languages
that can be recognized by a **Turing machine**.

These languages are also known
as the **recursively enumerable** or
Turing-recognizable languages.

Note that this is different from the **recursive languages**,
which can be decided by an **always-halting Turing machine**.

https://en.wikipedia.org/wiki/Regular_expression

Type-0 grammars

Context-sensitive grammar

Type-1 grammars generate the **context-sensitive languages**.

These grammars have rules of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$
with A a **nonterminal** and α, β, and γ strings of **terminals**
and/or **nonterminals**.

The strings α and β may be <u>empty</u>, but γ must be <u>nonempty</u>.

The rule $S \rightarrow \epsilon$ is allowed
if S does not appear on the right side of any rule.

The languages described by these grammars are exactly
all languages that can be <u>recognized</u> by a **linear bounded automaton**
(a **nondeterministic** Turing machine
whose tape is bounded by a constant times the length of the input.)

https://en.wikipedia.org/wiki/Regular_expression

Type-2 grammars

Context-free grammar

Type-2 grammars generate the context-free languages.

These are defined by rules of the form $A \rightarrow \gamma$
with A being a nonterminal and γ being a string of terminals and/or nonterminals.

These languages are exactly all languages
that can be recognized by a **non-deterministic pushdown automaton.**

Context-free languages—or rather its subset of deterministic context-free language—are the theoretical basis for the phrase structure of most **programming languages**, though their syntax also includes context-sensitive name resolution due to declarations and scope.

Often a subset of grammars is used to make parsing easier,
such as by an LL parser.

https://en.wikipedia.org/wiki/Regular_expression

Type-3 grammars (1)

Regular grammar

Type-3 grammars generate the regular languages.

restricts its rules to a single **nonterminal** on the **left**-hand side

a **right**-hand side consisting of a single **terminal**,
possibly followed by a **single nonterminal** (**right regular**).

the **right**-hand side consisting of a single **terminal**,
possibly preceded by a single **nonterminal** (**left regular**).

https://en.wikipedia.org/wiki/Regular_expression

Type-3 grammars (1)

Right regular and left regular generate the same languages.

However, if left-regular rules and right-regular rules are combined,
the language need no longer be regular.

The rule $S \rightarrow \epsilon$ is also allowed here
if S does not appear on the right side of any rule.

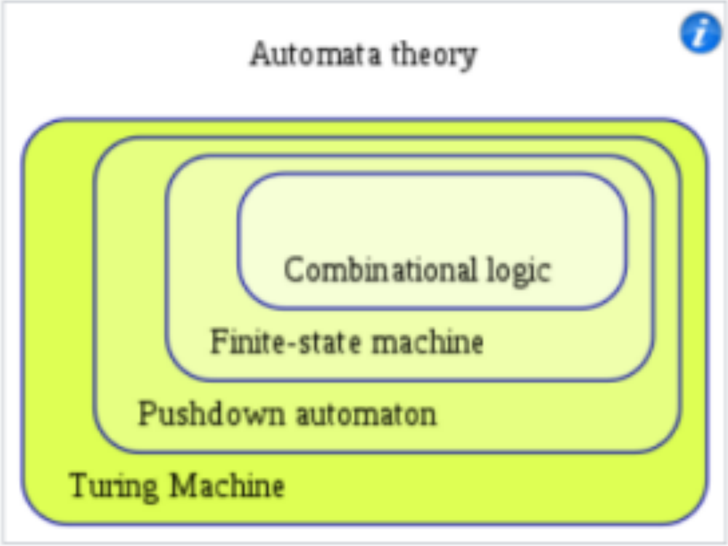
These languages are exactly all languages
that can be decided by a **finite state automaton**.

Additionally, this family of formal languages
can be obtained by **regular expressions**.

Regular languages are commonly used
to define search patterns and
the lexical structure of programming languages.

https://en.wikipedia.org/wiki/Regular_expression

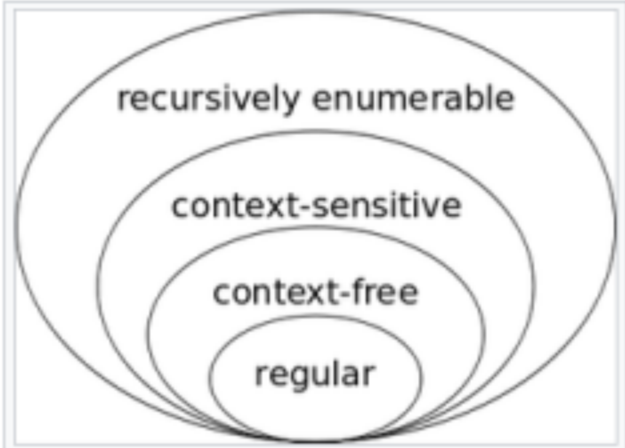
Class of Automata



Classes of automata
(Clicking on each layer will take you to an article on that subject)

https://en.wikipedia.org/wiki/Automata_theory

Chomsky Hierarchy



Regular language in classes of Chomsky hierarchy.

https://en.wikipedia.org/wiki/Regular_language

Class of Automata

Finite State Machine (FSM)	Regular Language
Pushdown Automaton (PDA)	Context-Free Language
Turing Machine	Recursively Enumerable Language

https://en.wikipedia.org/wiki/Automata_theory

Regular Language

a **regular language** (a **rational language**) is
a formal language that can be expressed
using a **regular expression**,
in the strict sense

Alternatively, a regular language can be defined
as a language recognized by a **finite automaton**.

The equivalence of **regular expressions** and
finite automata is known as **Kleene's theorem**.

Regular languages are very useful
in input parsing and programming language design.

https://en.wikipedia.org/wiki/Regular_language

Regular Language – Formal Definition

The collection of **regular languages** over an **alphabet** Σ
is defined recursively as follows:

The **empty language** Ø, and the **empty string language** {ε}
are **regular languages**.

For each a ∈ Σ (a belongs to Σ),
the **singleton language** {a} is a **regular language**.

If A and B are **regular languages**,
then A ∪ B (**union**), A • B (**concatenation**),
and A* (**Kleene star**) are **regular languages**.

No other languages over Σ are **regular**.

See regular expression for its syntax and semantics.
Note that the above cases are in effect the defining
rules of regular expression.

https://en.wikipedia.org/wiki/Regular_language

Equivalent Formalism

1. it is the language of a **regular expression** (by the above definition)
2. it is the language accepted by a **nondeterministic finite automaton** (**NFA**)
3. it is the language accepted by a **deterministic finite automaton** (**DFA**)
4. it can be generated by a **regular grammar**
5. it is the language accepted by an **alternating finite automaton**
6. it can be generated by a **prefix grammar**
7. it can be accepted by a read-only **Turing machine**

https://en.wikipedia.org/wiki/Regular_language

Regular Language Example

All **finite** languages are **regular**;

in particular the **empty** string language {ε} = Ø* is **regular**.

Other typical examples include the language
consisting of **all strings** over the **alphabet** {a, b}
which contain an even number of a's, or
the language consisting of all strings of the form:
several as followed by several b's.

A simple example of a language
that is **not regular** is the set of strings $\{ a^n b^n \mid n \geq 0 \}$.

Intuitively, it <u>cannot</u> be recognized with a **finite automaton**,
since a **finite automaton** has **finite memory**
and it cannot remember the exact number of a's.

https://en.wikipedia.org/wiki/Regular_language

References

[1] http://en.wikipedia.org/
[2]