Eulerian Cycle (2A)

Young Won Lim

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

This document was produced by using LibreOffice and Octave.

Path and Trail

A **path** is a **trail** in which all **vertices** are distinct.
(except possibly the first and last)

A **trail** is a **walk** in which all **edges** are distinct.

	Vertices	**Edges**	
Walk	may repeat	may repeat	(Closed/Open)
Trail	may repeat	cannot repeat	(Open)
Path	cannot repeat	cannot repeat	(Open)
Circuit	may repeat	cannot repeat	(Closed)
Cycle	cannot repeat	cannot repeat	(Closed)

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

Simple Paths and Cycles

Most literatures require that all of the **edges** and **vertices** of a **path** be distinct from one another.

But, some do not require this and instead use the term **simple path** to refer to a **path** which contains no repeated **vertices**.

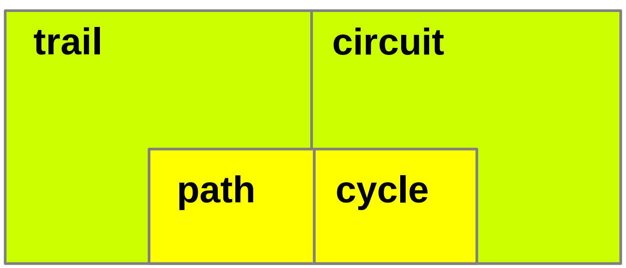
A **simple cycle** may be defined as a **closed walk** with no repetitions of **vertices** and **edges** allowed, other than the repetition of the **starting** and **ending vertex**

There is considerable variation of terminology!!!
Make sure which set of definitions are used...

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

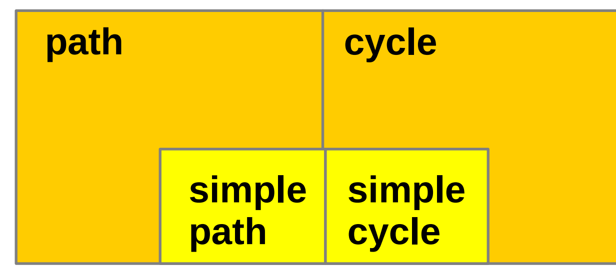
Simple Paths and Cycles

Most literatures



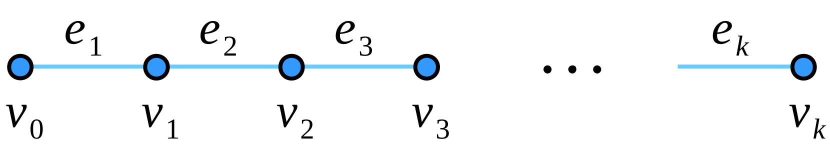
narrow sense path & cycle

some other literatures



wide sense path & cycle

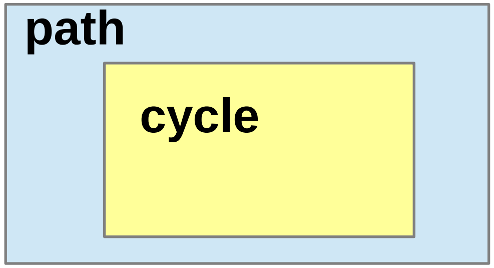
Paths and Cycles



One of a kind

path $v_{0,}\ e_{1,}\ v_{1,}\ e_{2,}\ \cdots,\ e_k,\ v_k$

cycle $v_{0,}\ e_{1,}\ v_{1,}\ e_{2,}\ \cdots,\ e_k,\ v_k \quad (v_0 = v_k)$



path $v_{0,}\ e_{1,}\ v_{1,}\ e_{2,}\ \cdots,\ e_k,\ v_k \quad (v_0 \neq v_k)$

cycle $v_{0,}\ e_{1,}\ v_{1,}\ e_{2,}\ \cdots,\ e_k,\ v_k \quad (v_0 = v_k)$

Two different kinds

Euler Cycle

Some people reserve the terms **path** and **cycle** to mean non-self-intersecting path and cycle. — no repeating vertices

A (potentially) self-intersecting path is known as a **trail** or an **open walk**; — repeating vertices

and a (potentially) self-intersecting cycle, a **circuit** or a **closed walk**. — repeating vertices

This ambiguity can be avoided by using the terms **Eulerian trail** and **Eulerian circuit** when self-intersection is allowed — repeating vertices

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

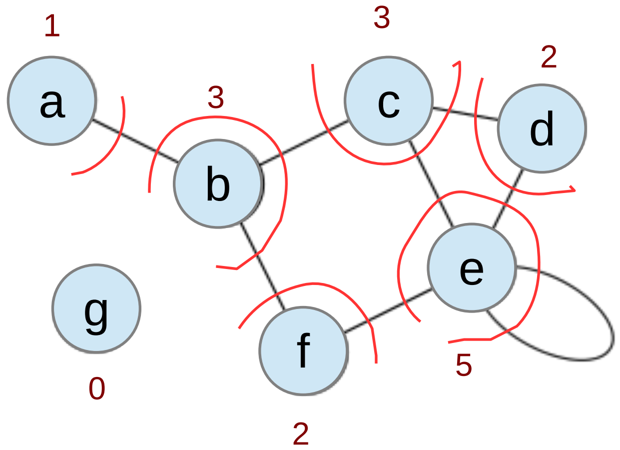
Degree of a vertex

the **degree** (or **valency**) of a vertex is
the number of edges incident to the vertex,
with loops counted twice.

The degree of a vertex v is denoted deg(v)
the maximum degree of a graph G, denoted by Δ(G)
the minimum degree of a graph, denoted by δ(G)

$\Delta(G) = 5$
$\delta(G) = 0$

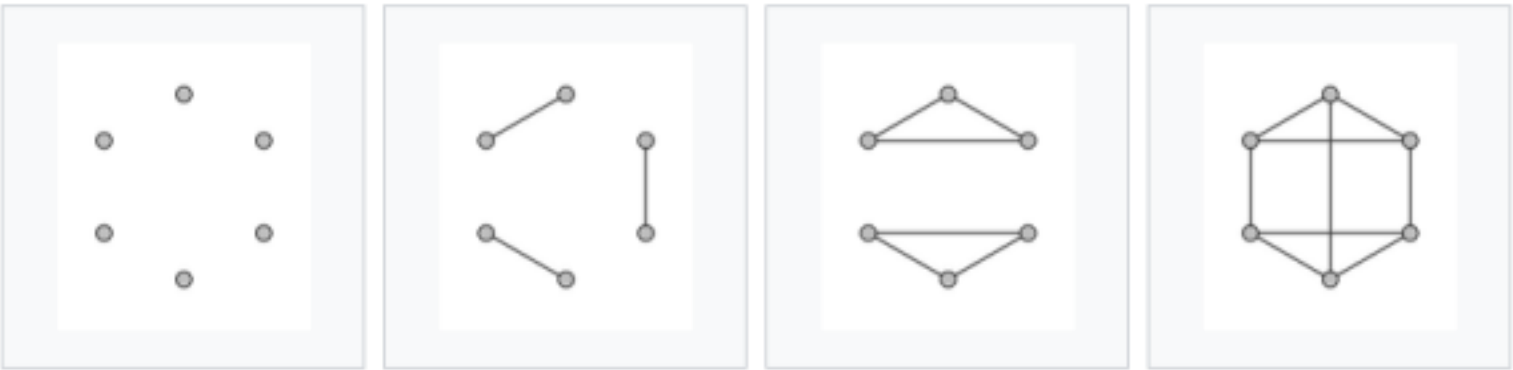
In a **regular** graph, all degrees are the same



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

Regular Graphs

a **regular graph** is a graph where each vertex has the same number of neighbors; i.e. every vertex has the same degree or valency.



0-regular graph 1-regular graph 2-regular graph 3-regular graph

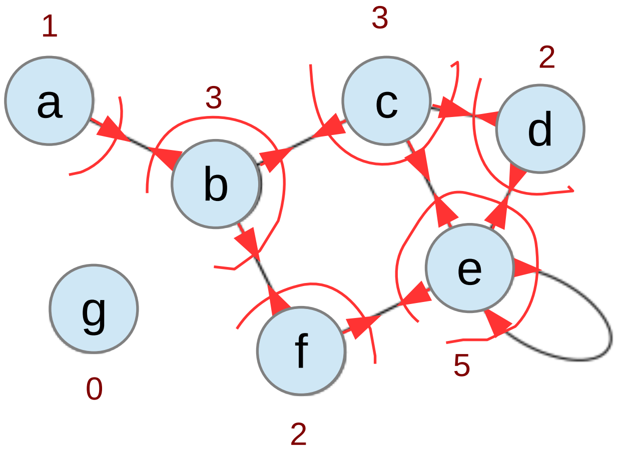
https://en.wikipedia.org/wiki/Regular_graph

Handshake Lemma

The degree sum formula states that,
given a graph G = (V , E)

$$\sum_{v \in V} \deg(v) = 2|E| \, .$$

This statement (as well as the degree sum formula) is known as the **handshaking lemma**.



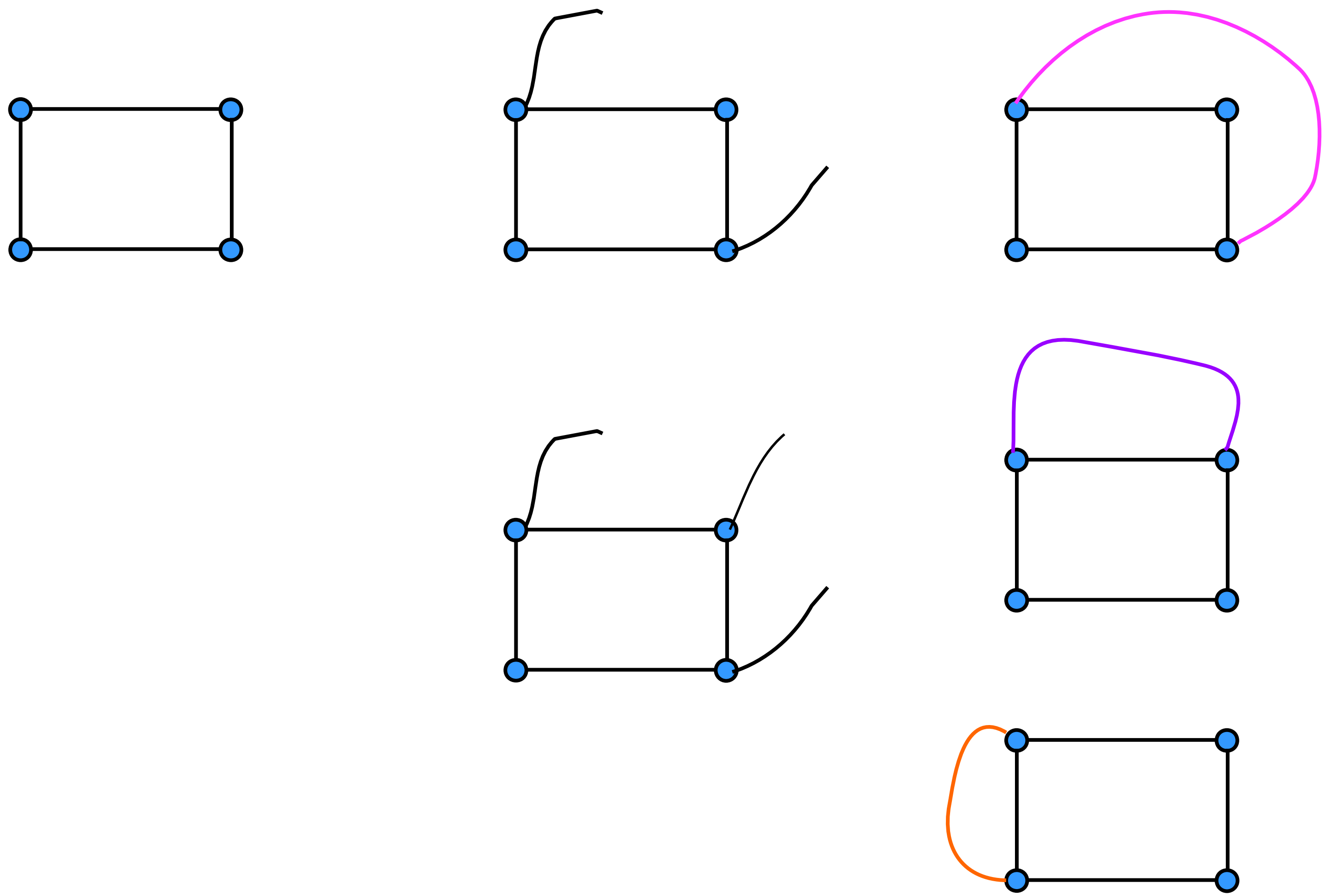
$$
\begin{aligned}
deg(a) &= 1 \\
deg(b) &= 3 \\
deg(c) &= 3 \\
deg(d) &= 2 \\
deg(e) &= 5 \\
deg(f) &= 2 \\
deg(g) &= 0 \\
\hline
& 16
\end{aligned}
$$

$$|E| = 8$$

$$2|E| = 16$$

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

Adding odd vertex



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

The number of odd vertices

Even vertices : $\{x_1, x_2, \cdots, x_m\}$

$$S = \underline{deg(x_1)} + \underline{deg(x_2)} + \cdots + \underline{deg(x_m)}$$

$$deg(x_i) : even$$

$$S = \underline{even} + \underline{even} + \cdots + \underline{even}$$

Odd vertices : $\{y_1, y_2, \cdots, y_n\}$

$$T = \underline{deg(y_1)} + \underline{deg(y_2)} + \cdots + \underline{deg(y_n)}$$

$$deg(y_i) : odd$$

$$T = \underline{odd} + \underline{odd} + \cdots + \underline{odd}$$

$S : even$

$S+T : even$

$\rightarrow$ $T : even = \sum n \text{ odd numbers}$

$\Rightarrow$ $n : even$

in any graph, the number of vertices with odd degree is even.

# of **odd** vertices	Eulerian **Path**	Eulerian **Cycle**
0	No	Yes
2	Yes	No
4,6,8, …	No	No
1,3,5,7, …	No such graph	No such graph

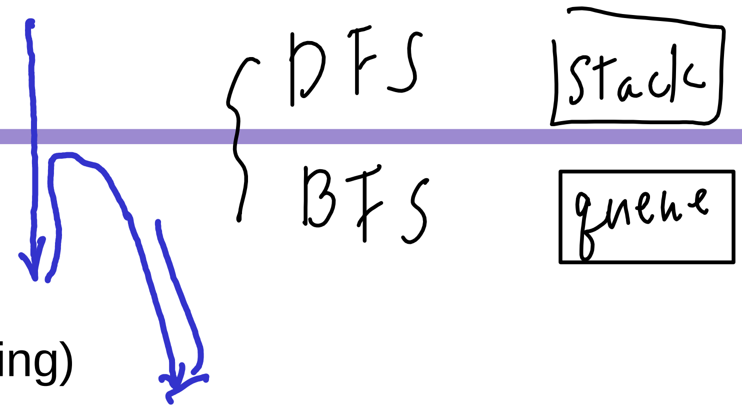
References

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

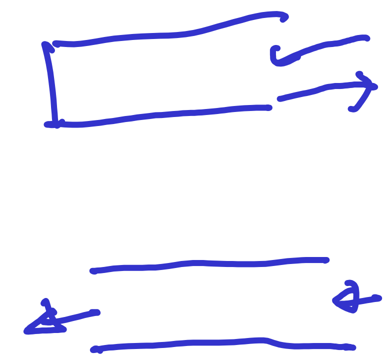
Graph Search (6A)

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

This document was produced by using LibreOffice and Octave.

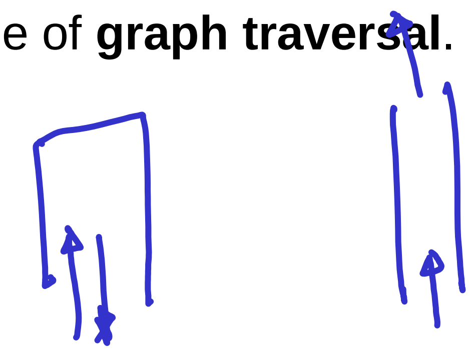


graph traversal (**graph search**) refers to the process of visiting (checking and/or updating) each **vertex** in a graph.



Such traversals are classified by the order in which the vertices are visited.

Tree traversal is a special case of **graph traversal**.



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

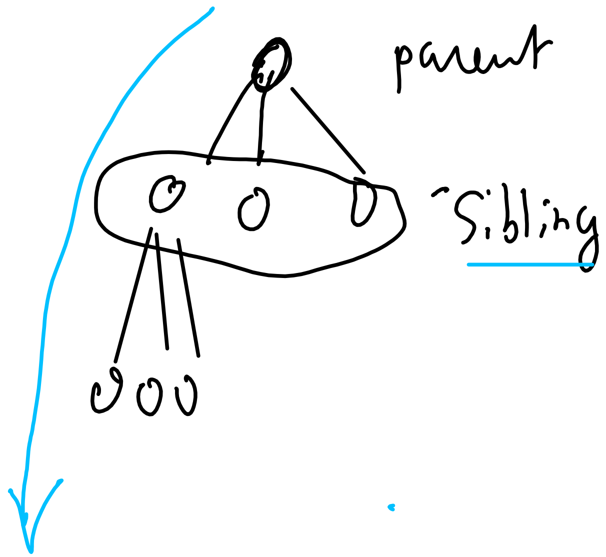
DFS

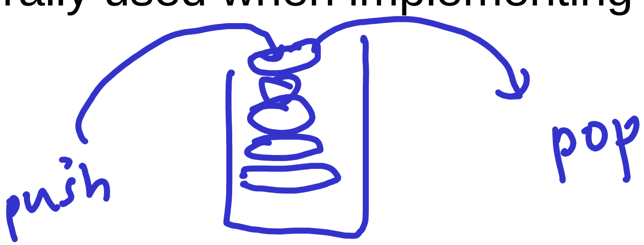
A depth-first search (**DFS**)
is an algorithm for traversing a finite graph.

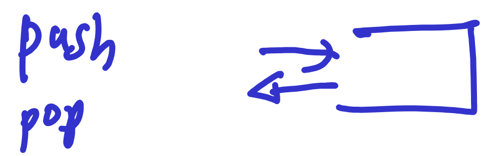
DFS visits the **child vertices**
before visiting the **sibling vertices**;

that is, it traverses the **depth** of any particular path
before exploring its **breadth**.

A **stack** (often the program's call stack via recursion) is
generally used when implementing the algorithm.







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

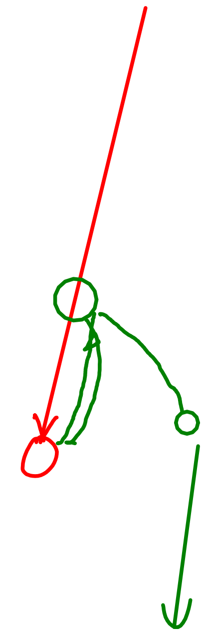
DFS Backtrack

The algorithm begins with a chosen "**root**" vertex;

it then iteratively transitions from the **current** vertex to an **adjacent**, **unvisited** vertex, until it can no longer find an **unexplored vertex** to transition to from its current location.

The algorithm then **backtracks** along previously **visited vertices**, until it finds a vertex connected to yet more uncharted territory.

It will then proceed down the **new path** as it had before, **backtracking** as it encounters **dead-ends**, and ending only when the algorithm has backtracked past the original "root" vertex from the very first step.



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

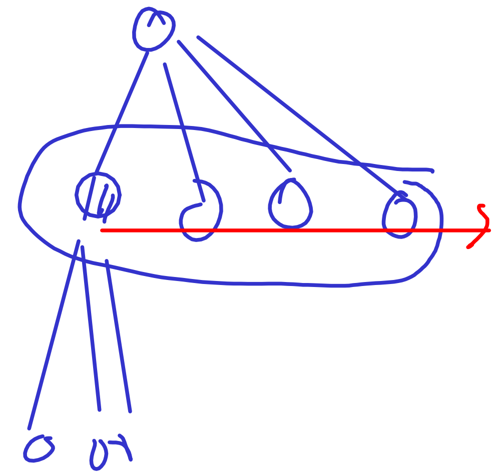
BFS

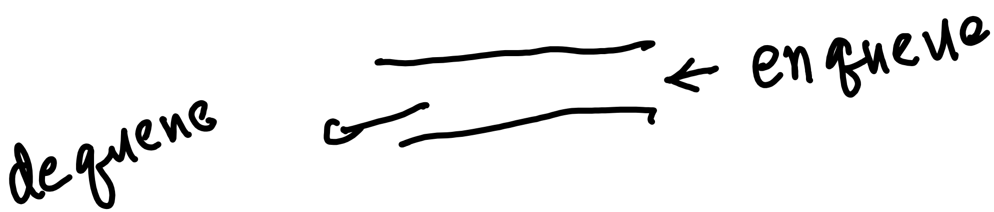
A breadth-first search (**BFS**) is
another technique for traversing a finite graph.

BFS visits the **neighbor** vertices
before visiting the **child** vertices

a **queue** is used in the search process

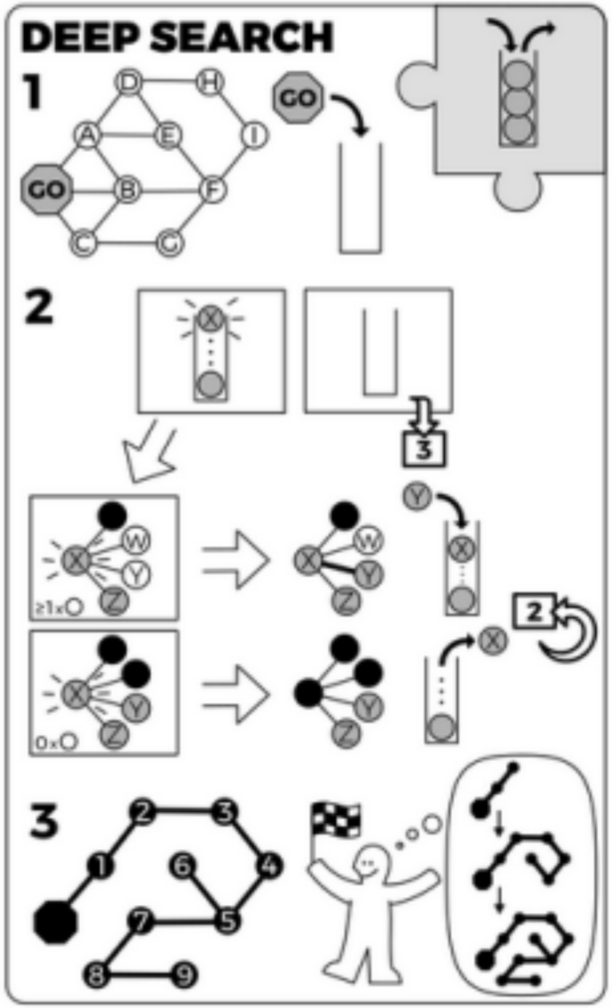
This algorithm is often used to find
the **shortest path** from one vertex to another.



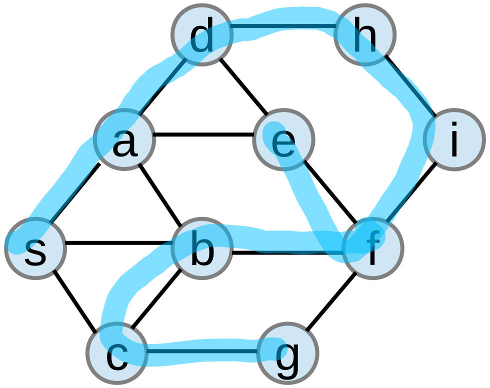


https://en.wikipedia.org/wiki/Graph_traversal

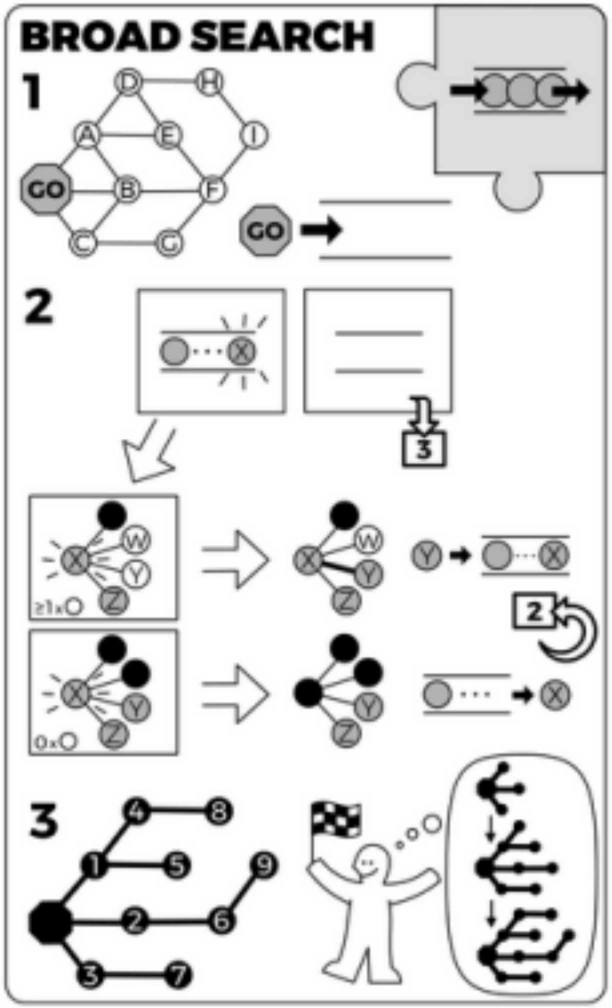
Depth First Search Example

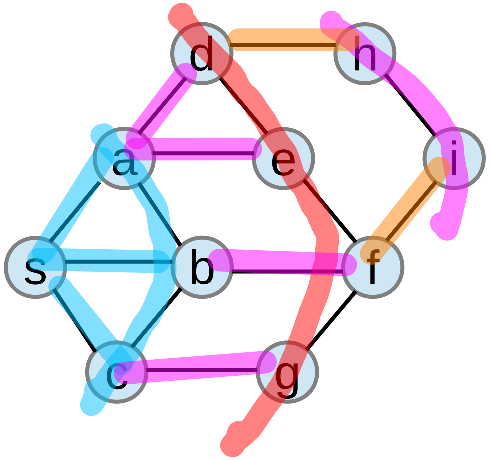


https://en.wikipedia.org/wiki/Graph_traversal



Breadth First Search Example

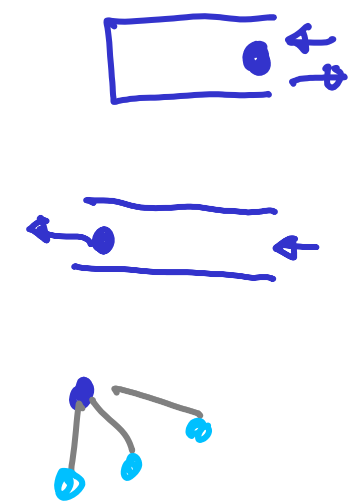




https://en.wikipedia.org/wiki/Graph_traversal

General Graph Search Algorithm – 1

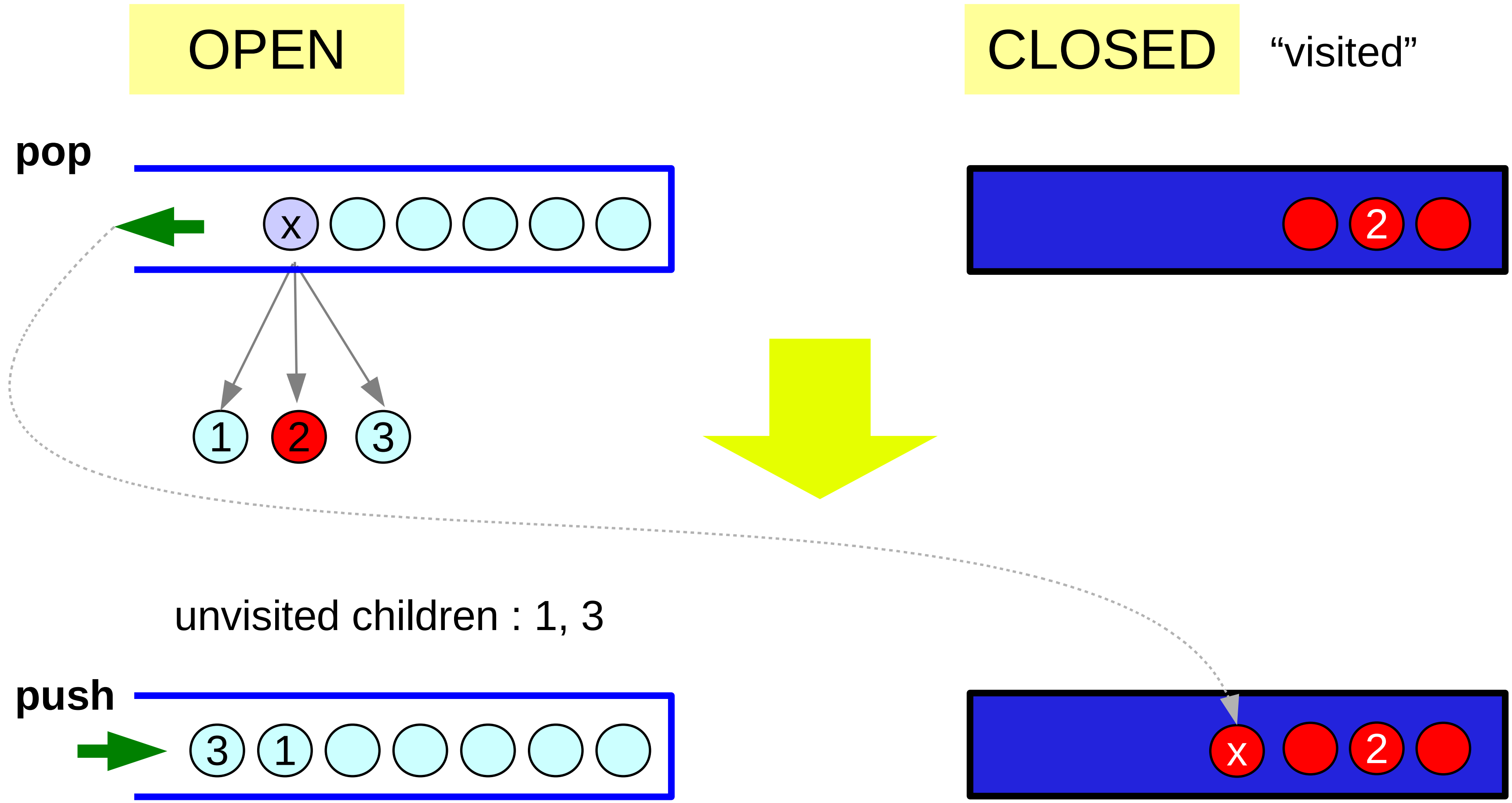
```
Search(Start, isGoal, Criteria)
    insert(Start, Open);
    repeat
        if (empty(Open)) then return fail;
        select node from Open using Criteria;
        mark node as visited;
        if (isGoal(node)) then return node;
        for each child of node do
            if (child not already visited)
                then insert(child, Open);
```



https://courses.cs.washington.edu/courses/cse326/08wi/a/lectures/lecture13.pdf

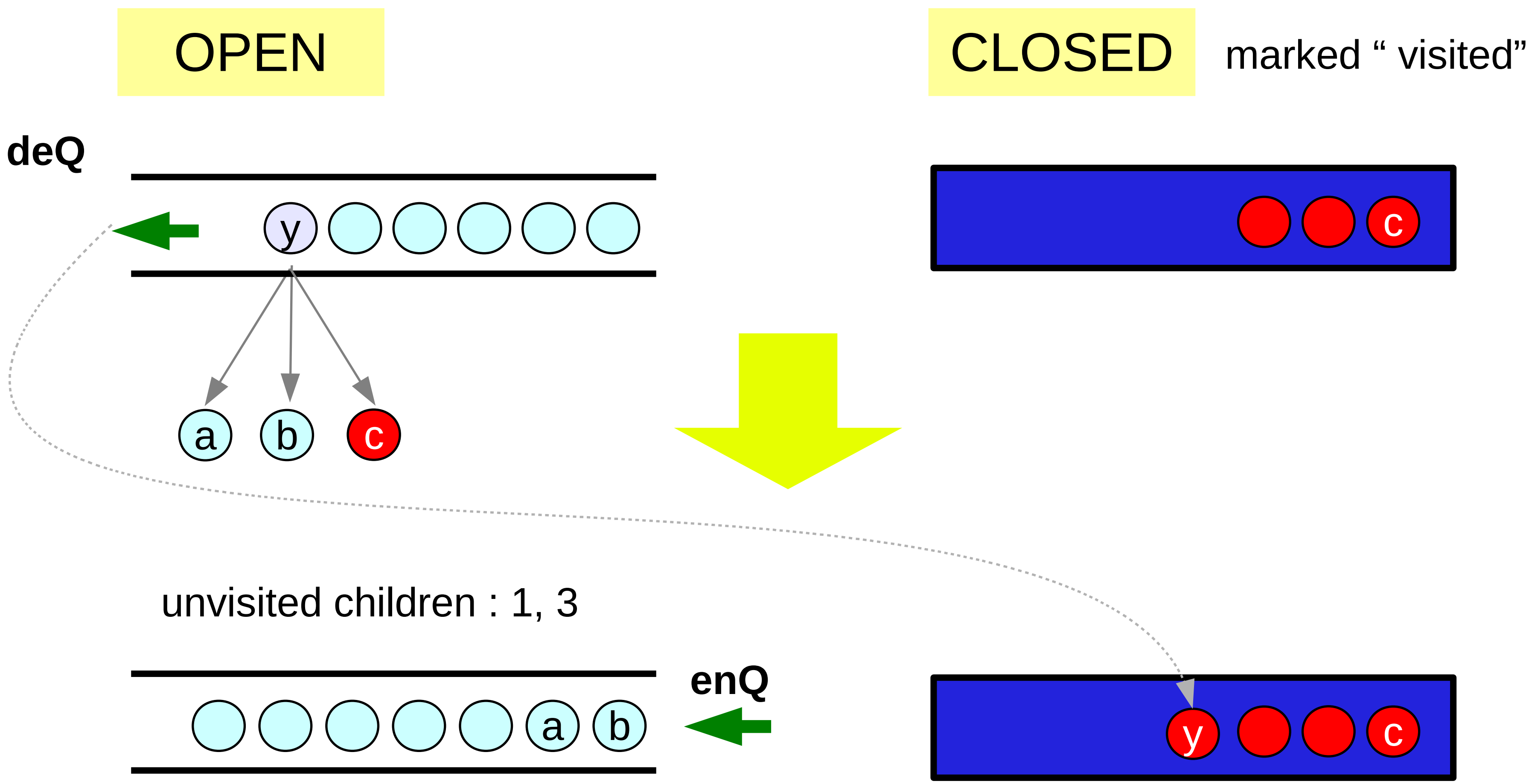
DFS

https://en.wikiversity.org/wiki/Artificial_intelligence/Lecture_aid



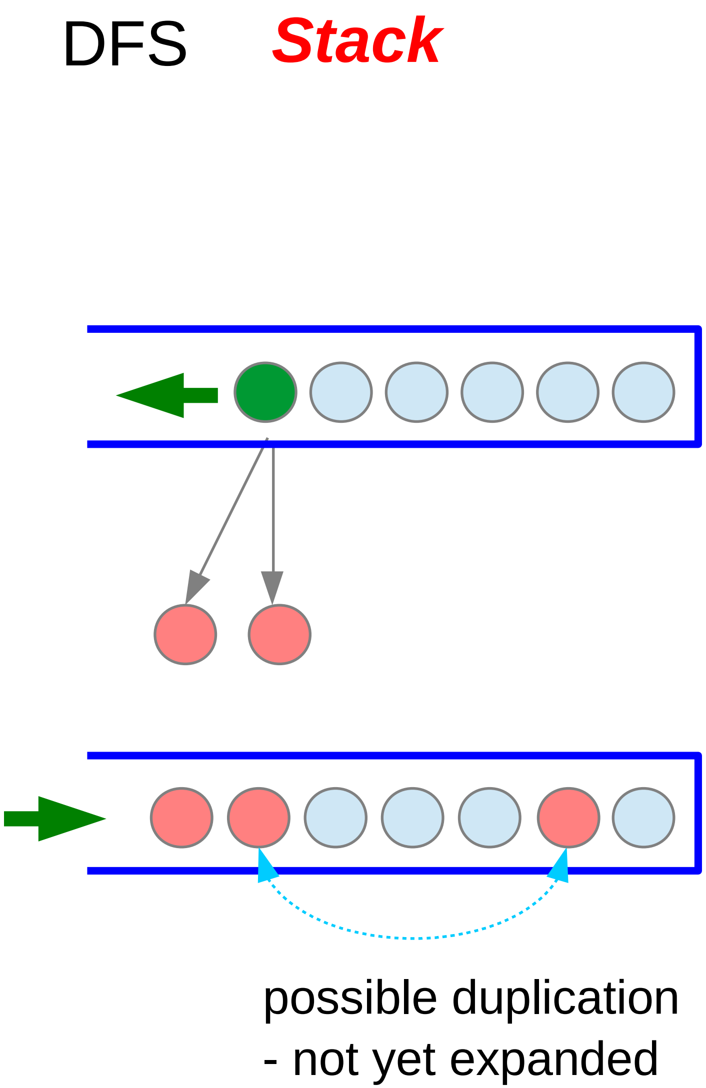
BFS

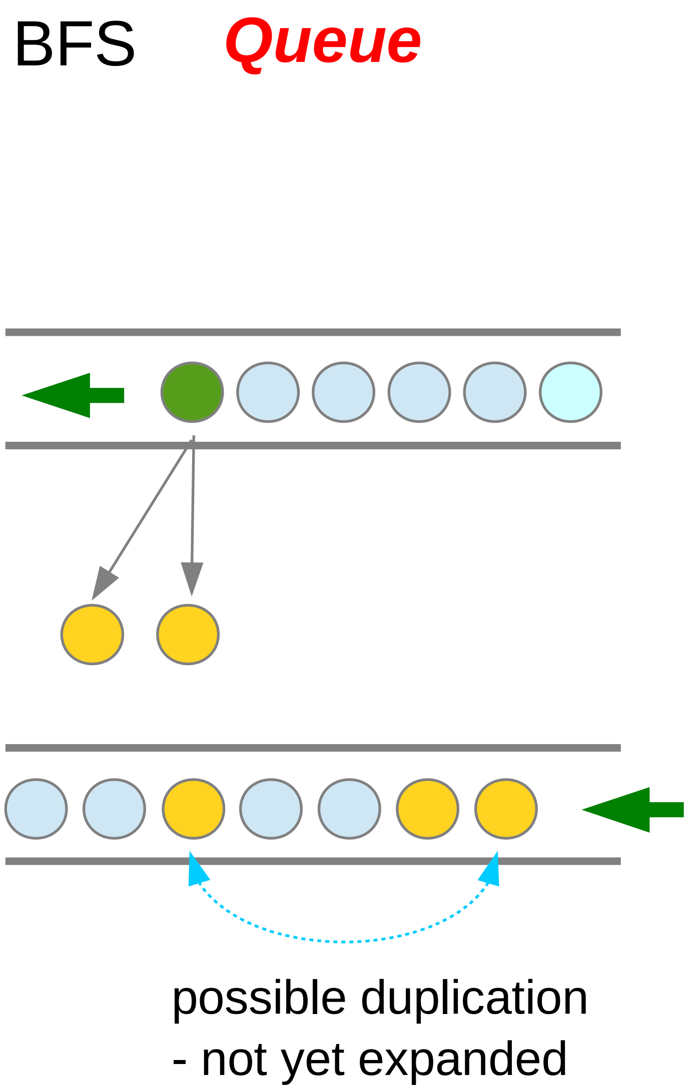
https://en.wikiversity.org/wiki/Artificial_intelligence/Lecture_aid



Possible duplication

https://en.wikiversity.org/wiki/Artificial_intelligence/Lecture_aid



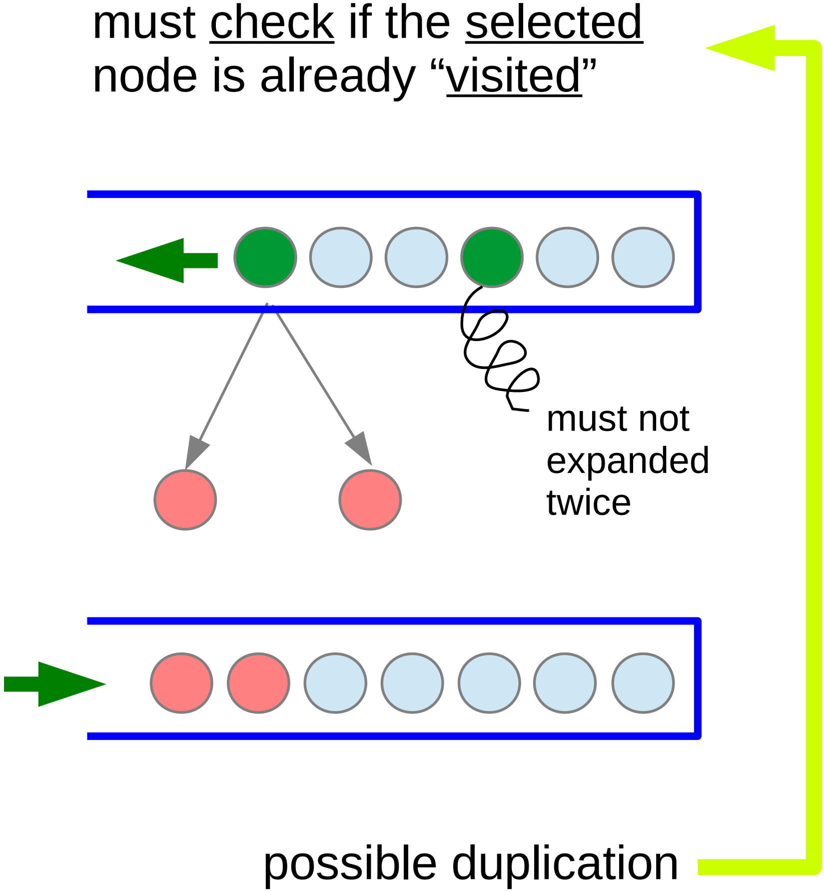


Must check before expansion

https://en.wikiversity.org/wiki/Artificial_intelligence/Lecture_aid

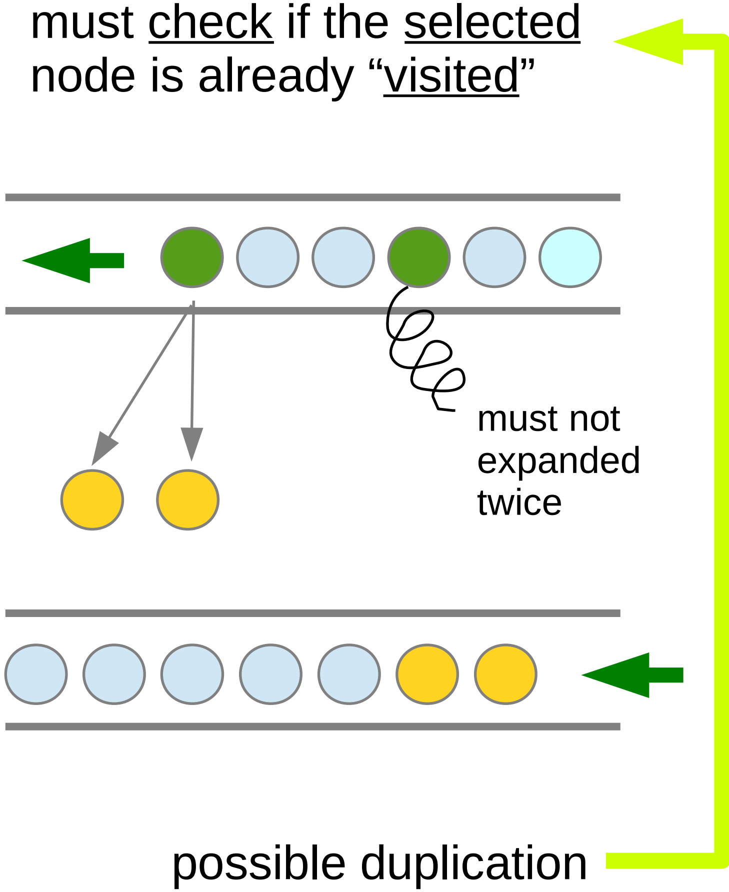
DFS ***Stack***

must check if the selected node is already "visited"



BFS ***Queue***

must check if the selected node is already "visited"



General Graph Search Algorithm – 1

```
Search(Start, isGoal, Criteria)
    insert(Start, Open);
    repeat
        if (empty(Open)) then return fail;
        select node from Open using Criteria;
        mark node as visited;
        if (isGoal(node)) then return node;
        for each child of node do
            if (child not already visited)
                then insert(child, Open);
```

Remedy 1:
check if visited when selecting

Remedy 2:
check redundant nodes

https://courses.cs.washington.edu/courses/cse326/08wi/a/lectures/lecture13.pdf

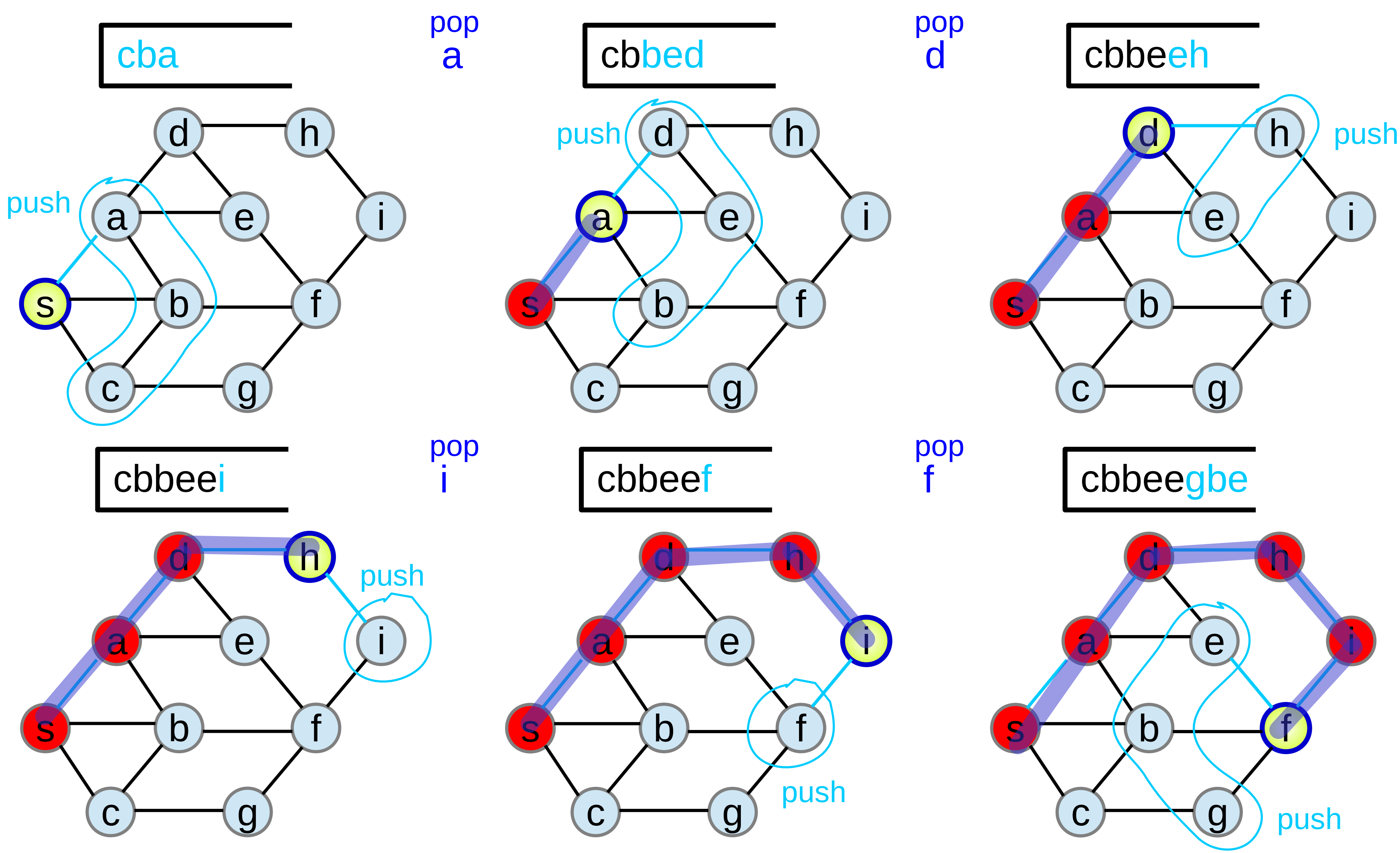
DFS-1 (Depth First Search)

Open list – use a stack
Select with Criteria – **pop**

```
DFS(Start, isGoal)
    push(Start, Open);                         // push
    repeat
        if (empty(Open)) then return fail;
        node := pop(Open);                     // pop
        mark node as visited;
        if (isGoal(node)) then return node;
        for each child of node do
            if (child not already visited) then
                push(child, Open);             // push
```

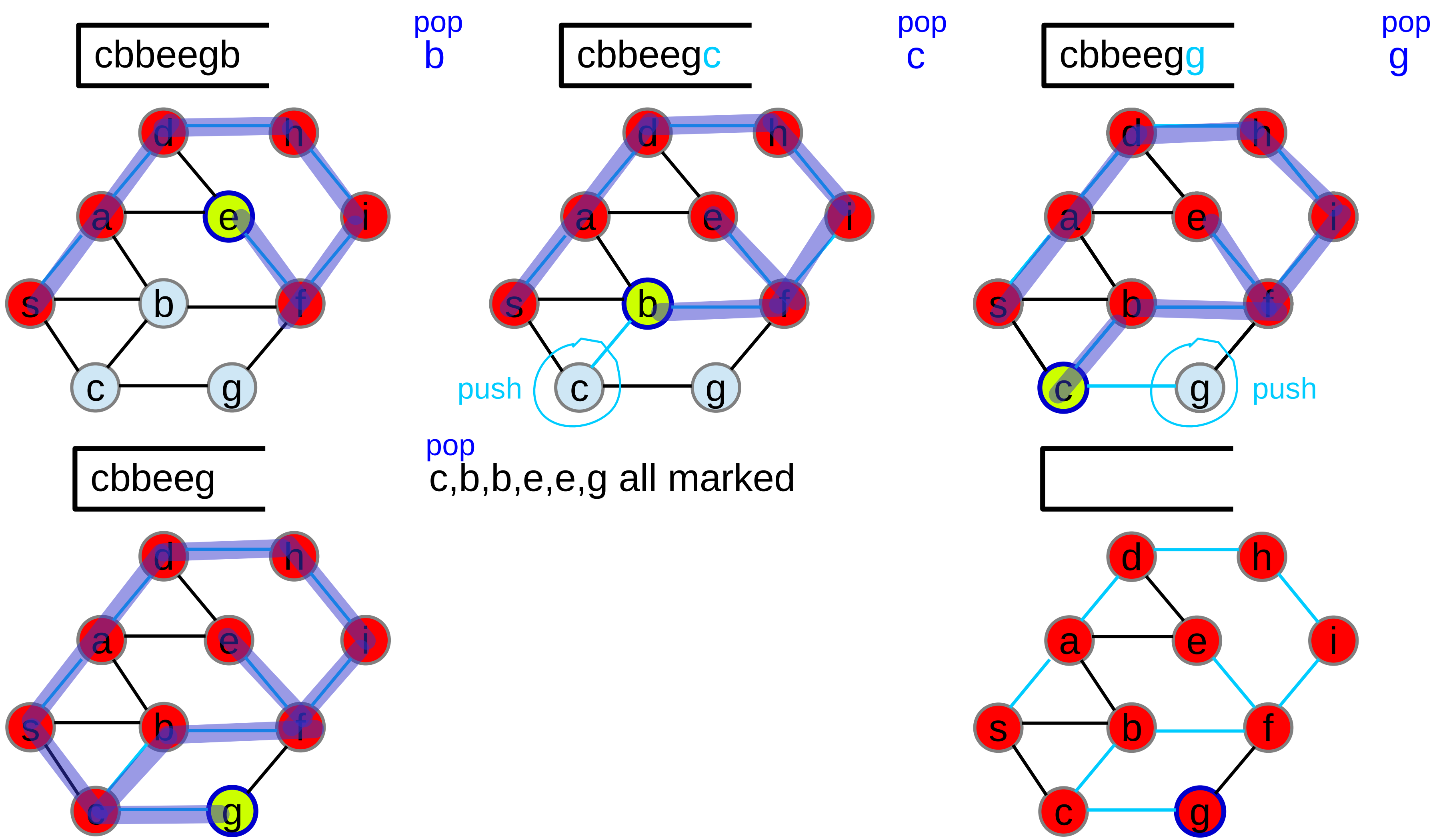
https://courses.cs.washington.edu/courses/cse326/08wi/a/lectures/lecture13.pdf

DFS-1 Example (1)



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

DFS-1 Example (2)



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

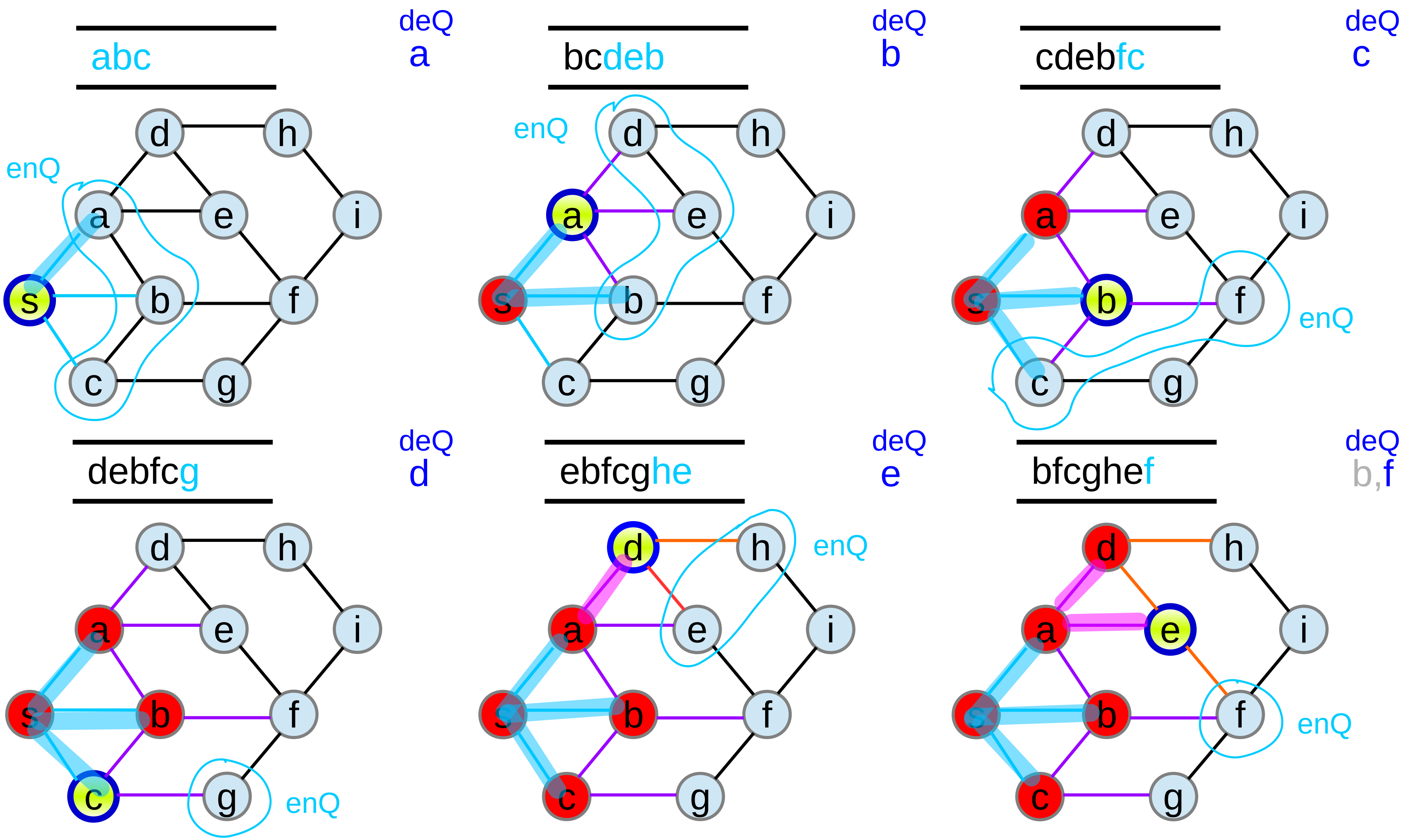
BFS-1 (Breadth First Search)

Open list – use a FIFO
Select with Criteria – **dequeue**

```
BFS(Start, isGoal)
    enqueue(Start, Open);                          // enqueue
    repeat
        if (empty(Open)) then return fail;
        node := dequeue(Open);                     // dequeue
        mark node as visited;
        if (isGoal(node)) then return node;
        for each child of node do
            if (child not already visited) then
                enqueue(child, Open);              // enqueue
```

https://courses.cs.washington.edu/courses/cse326/08wi/a/lectures/lecture13.pdf

BFS-1 Example (1)

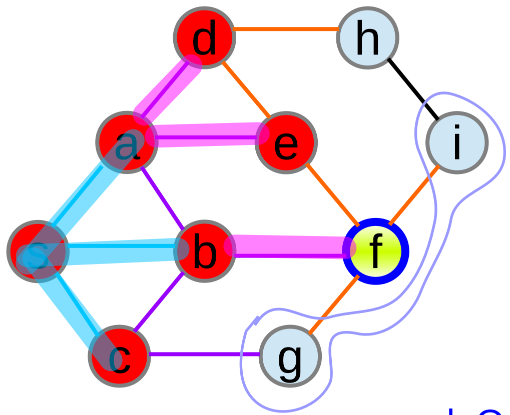


https://en.wikipedia.org/wiki/Graph_traversal

BFS-1 Example (2)

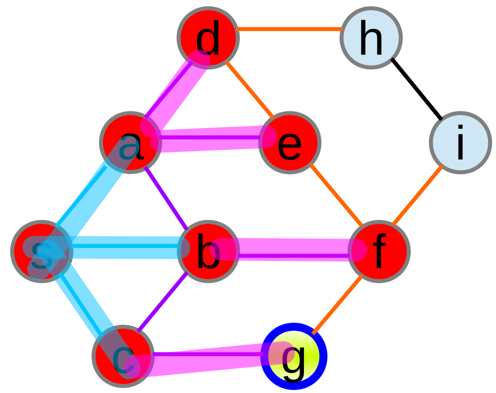
cghefig

deQ
c,g



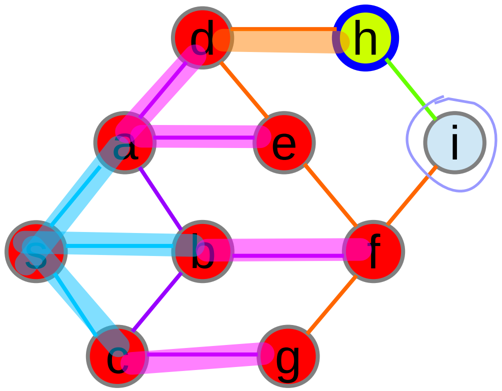
hefig

deQ
h



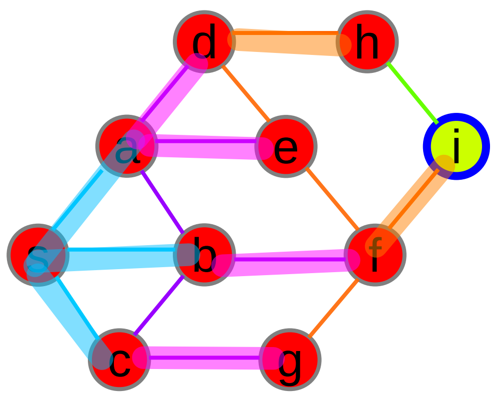
efigi

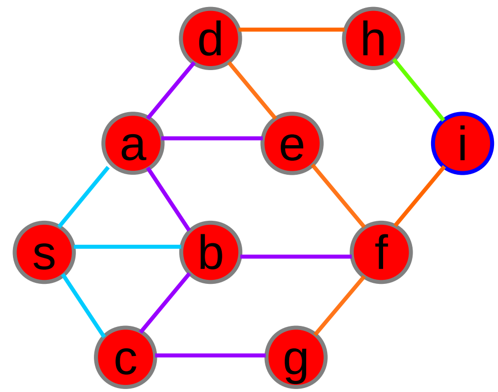
deQ
e,f,i



gi

deQ
g,i





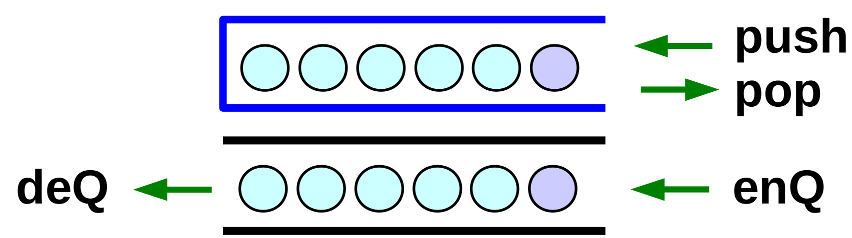
https://en.wikipedia.org/wiki/Graph_traversal

General Graph Search Algorithm – 2

```
Initialize as follows:
    unmark all nodes in N;
    mark node s;
    pred(s) = 0;   {that is, it has no predecessor}
    LIST = {s}
while LIST ≠ ø do
    select a node i in LIST;
    if node j is incident to an admissible arc (i,j) then
        mark node j;
        pred(j) := i;
        add node j to the end of LIST;
    else
        delete node i from LIST
```

DFS : select the last node i in LIST;

BFS : select the first node i in LIST;



https://ocw.mit.edu/courses/sloan-school-of-management/15-082j-network-optimization-fall-2010/lecture-notes/MIT15_082JF10_lec03.pdf

Admissible arc

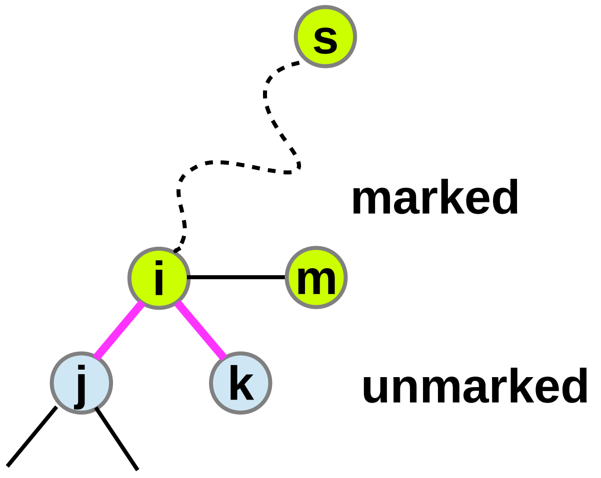
pred(**j**) is a node that **precedes j** on some path from s;

A node is either **marked** or **unmarked**.

Initially only node s is **marked**.

If a node is **marked**, it is **reachable** from node s.

An arc (**i**,**j**) ∈ A is **admissible**
if node **i** is marked and **j** is not.



https://ocw.mit.edu/courses/sloan-school-of-management/15-082j-network-optimization-fall-2010/lecture-notes/MIT15_082JF10_lec03.pdf

LIST

Before a node is added into LIST,
the node is **marked**

LIST contains only the **marked** nodes

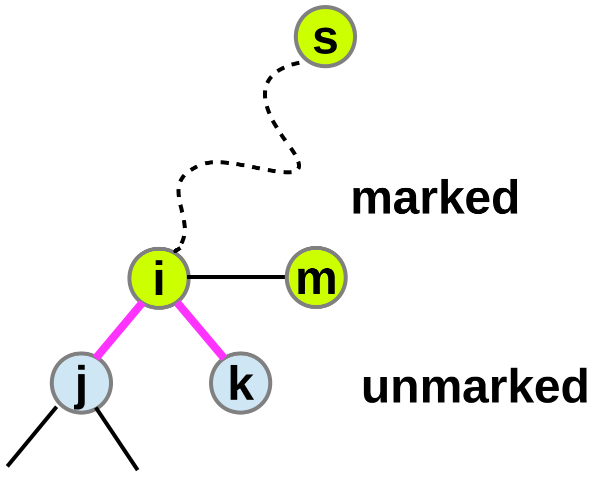
thus, the selected node **i** is **marked** already

The node **j** incident to the **admissible** arc(**i**,**j**)
must be **unmarked**

This node **j** is **marked** and added into LIST

In this way, LIST contains
only **marked** and **non-repeating** nodes

Check before inserting



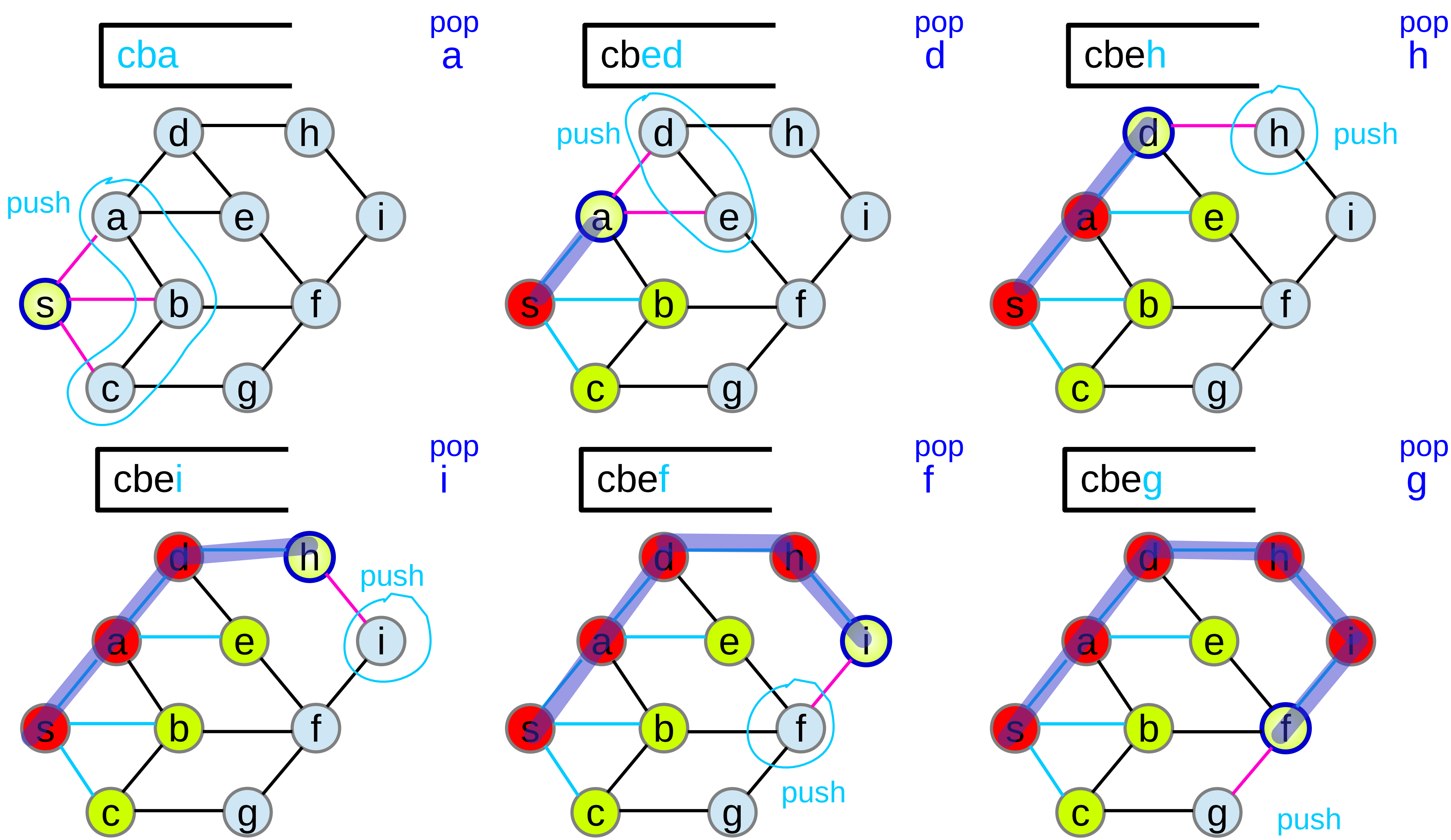
https://ocw.mit.edu/courses/sloan-school-of-management/15-082j-network-optimization-fall-2010/lecture-notes/MIT15_082JF10_lec03.pdf

DFS-2

```
Initialize as follows:
    unmark all nodes in N;
    mark node s;
    pred(s) = 0;   {that is, it has no predecessor}
    push s onto LIST
while LIST ≠ ø do
    pop a node i from LIST;
    if node j is incident to an admissible arc (i,j) then
        mark node j;
        pred(j) := i;
        push(node j) onto LIST;
    else
        delete node i from LIST
```

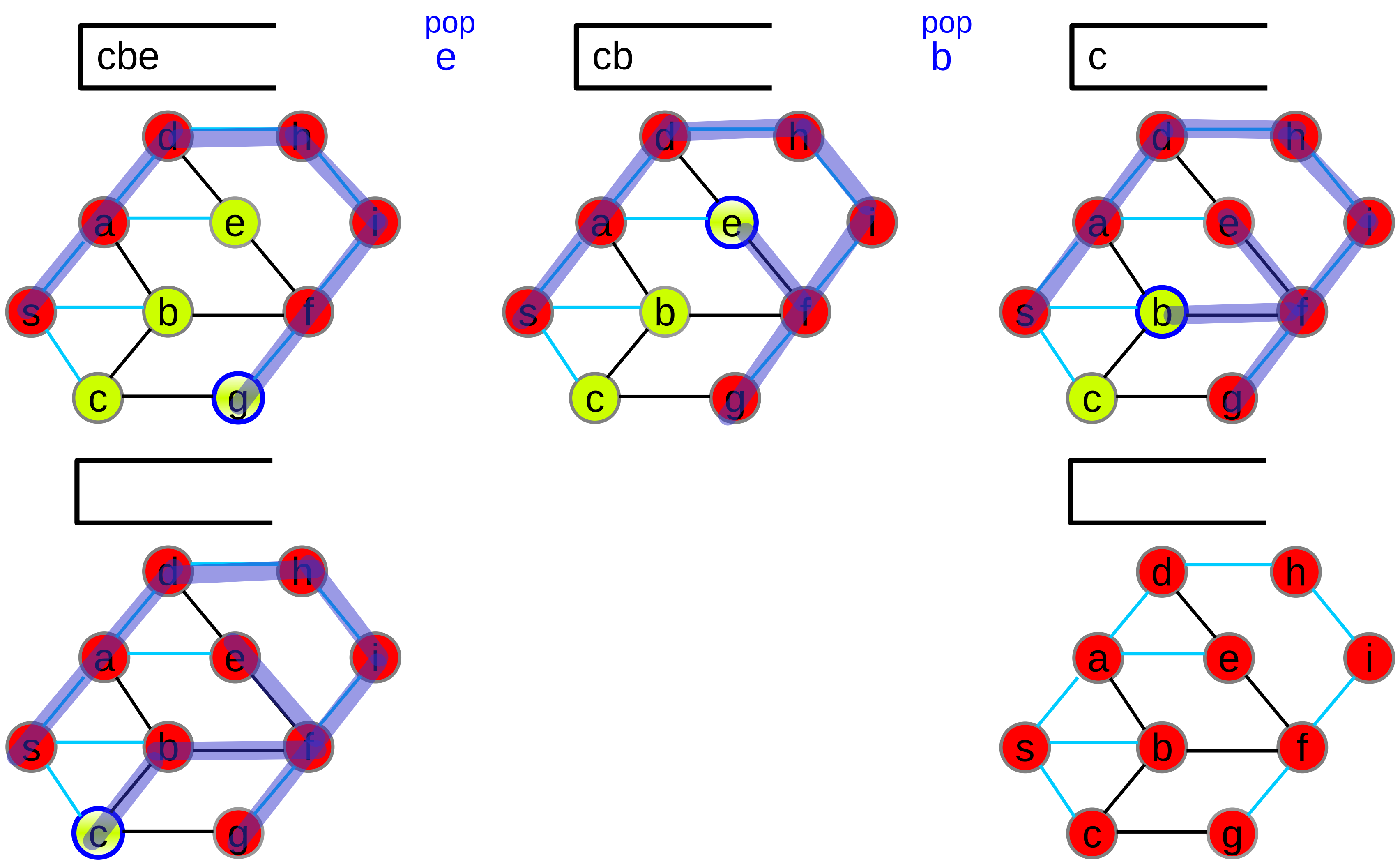
https://ocw.mit.edu/courses/sloan-school-of-management/15-082j-network-optimization-fall-2010/lecture-notes/MIT15_082JF10_lec03.pdf

DFS-2 Example (1)



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

DFS-2 Example (2)



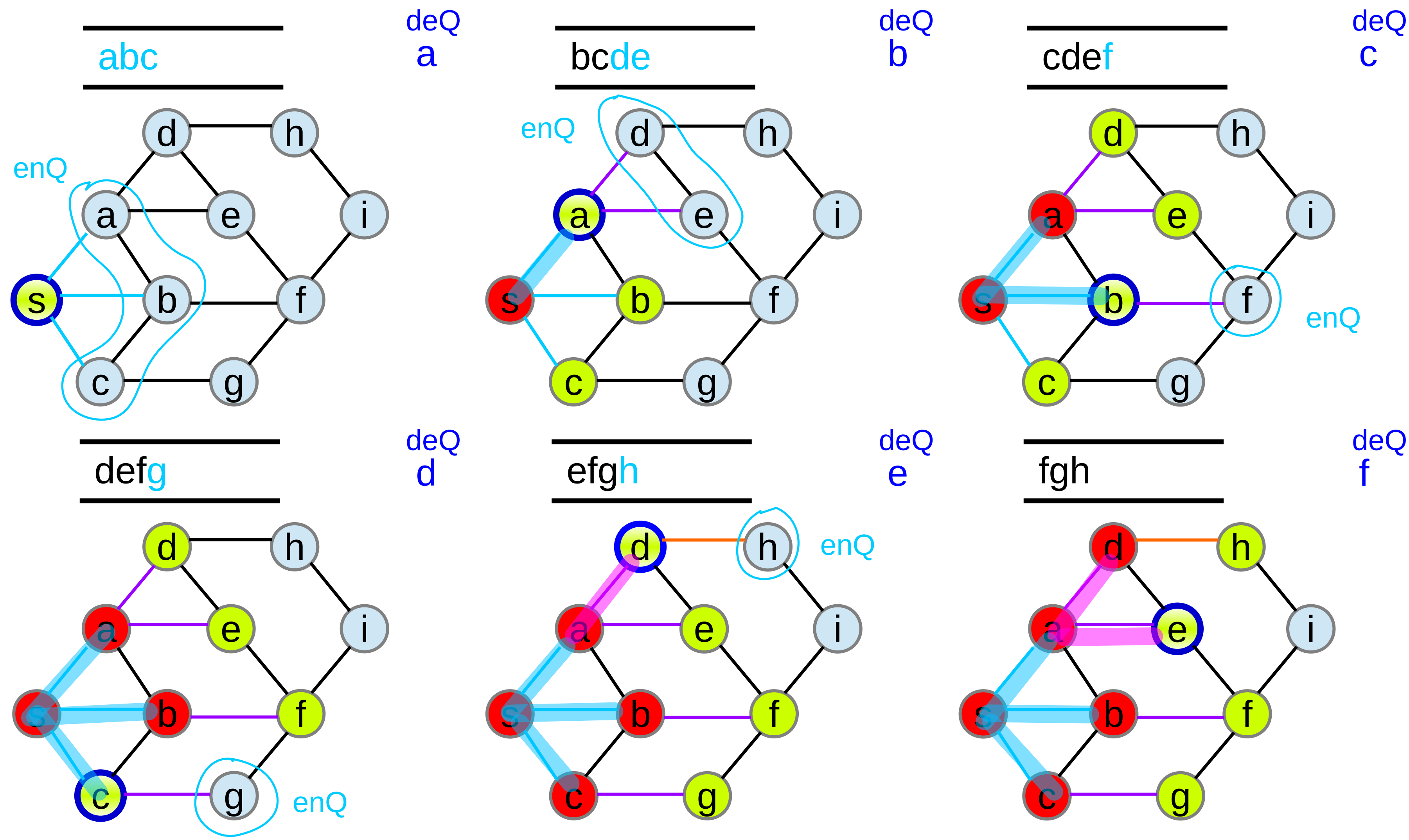
https://en.wikipedia.org/wiki/Graph_traversal

BFS-2

```
Initialize as follows:
      unmark all nodes in N;
      mark node s;
      pred(s) = 0;   {that is, it has no predecessor}
      enqueue s onto LIST
while LIST ≠ ø do
      dequeue node i from LIST;
      if node j is incident to an admissible arc (i,j) then
            mark node j;
            pred(j) := i;
            enqueue node j onto LIST;
      else
            delete node i from LIST
```

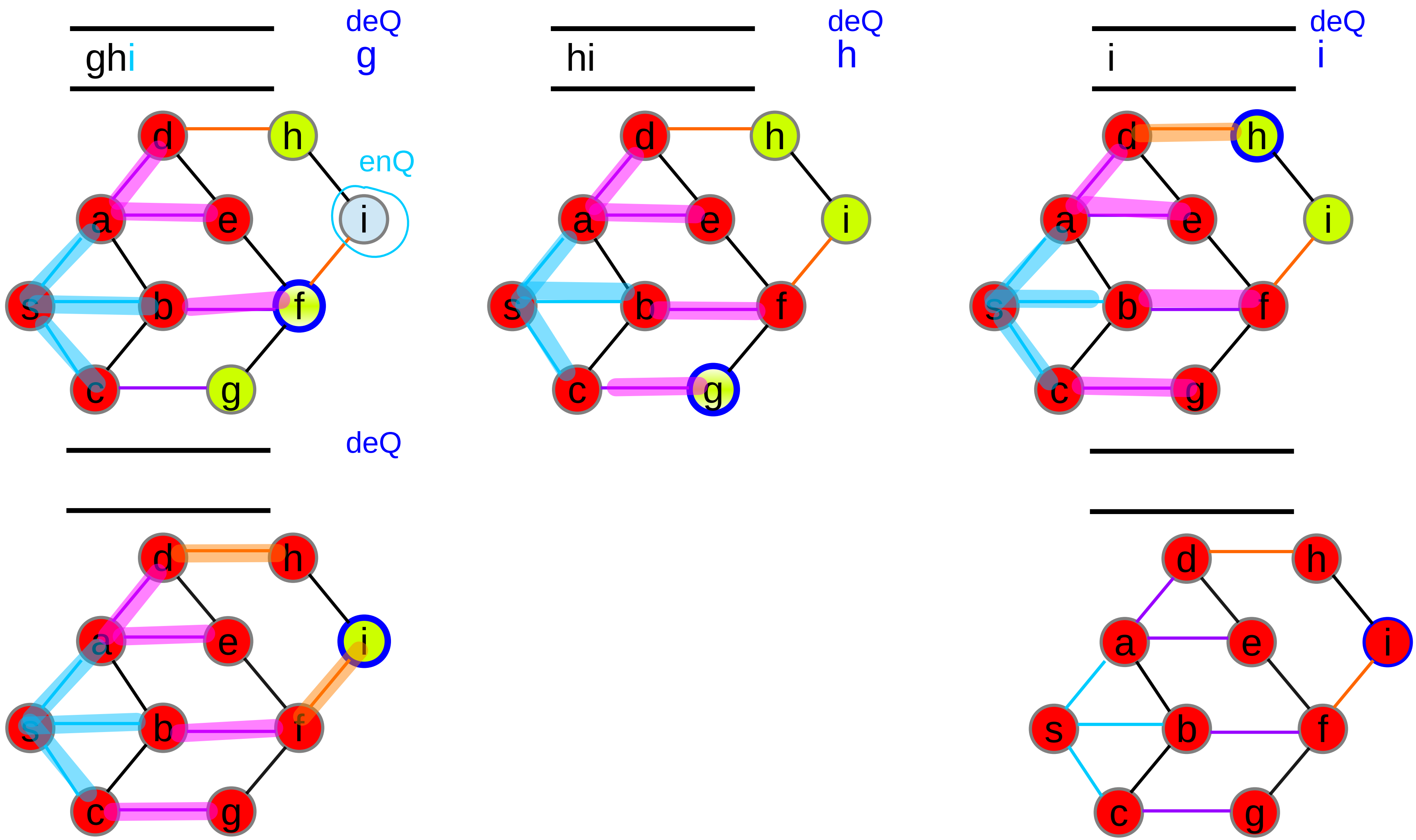
https://ocw.mit.edu/courses/sloan-school-of-management/15-082j-network-optimization-fall-2010/lecture-notes/MIT15_082JF10_lec03.pdf

BFS-2 Example (1)



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

BFS-2 Example (2)



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

DFS Pseudocode

```
procedure DFS(G, v):
    label v as explored
    for all edges e in G.incidentEdges(v) do
        if edge e is unexplored then
            w ← G.adjacentVertex(v, e)
            if vertex w is unexplored then
                label e as a discovered edge
                recursively call DFS(G, w)
            else
                label e as a back edge
```

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

BFS Pseudocode

```
procedure BFS(G, v):
    create a queue Q
    enqueue v onto Q
    mark v
    while Q is not empty:
        t ← Q.dequeue()
        if t is what we are looking for:
            return t
        for all edges e in G.adjacentEdges(t) do
            o ← G.adjacentVertex(t, e)
            if o is not marked:
                mark o
                enqueue o onto Q
    return null
```

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

References

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

Planar Graph (7A)

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

This document was produced by using LibreOffice and Octave.

Planar Graph

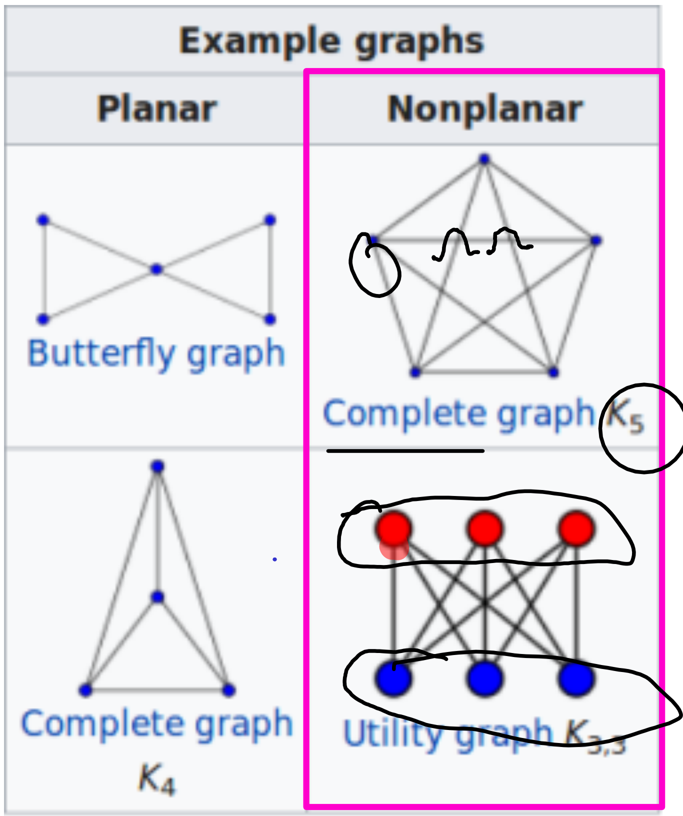
a planar graph is a graph that can be embedded in the plane, i.e., it can be drawn on the plane in such a way that its edges intersect only at their endpoints.

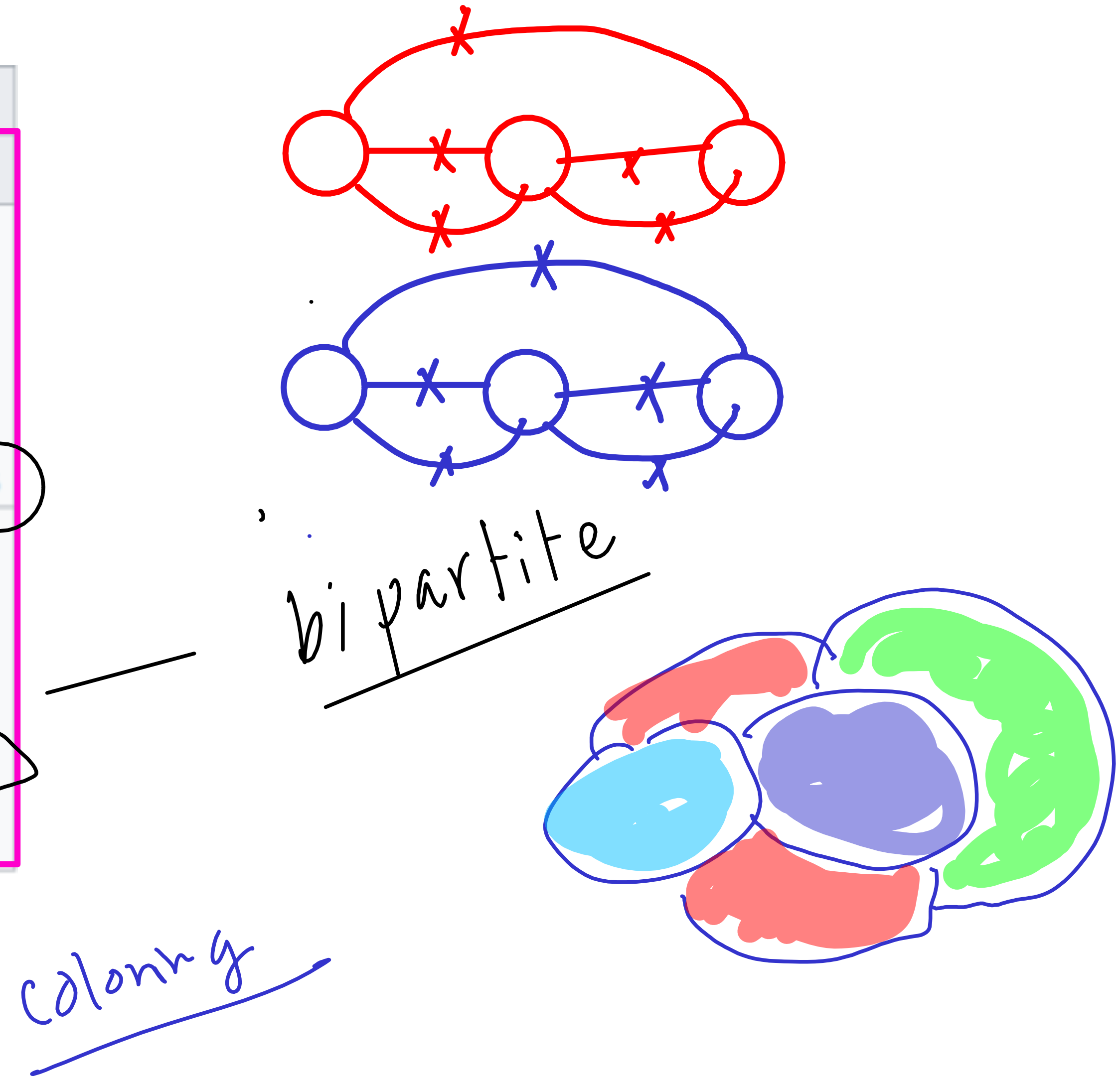
it can be drawn in such a way that no edges cross each other. Such a drawing is called a **plane graph** or **planar embedding** of the graph. (**planar representation**)

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

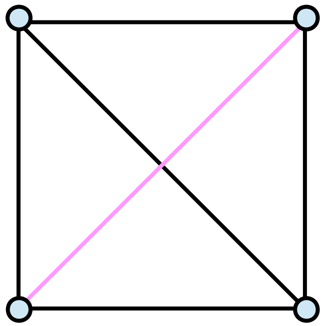


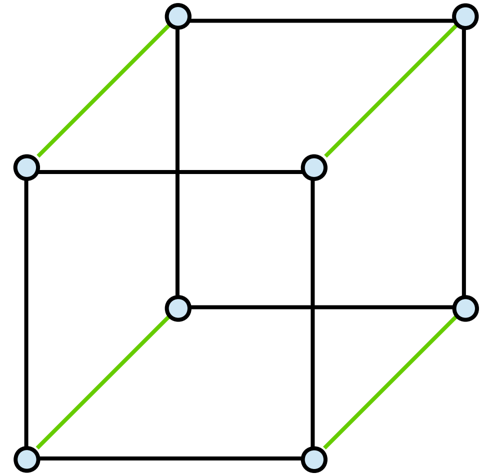


https://en.wikipedia.org/wiki/Planar_graph

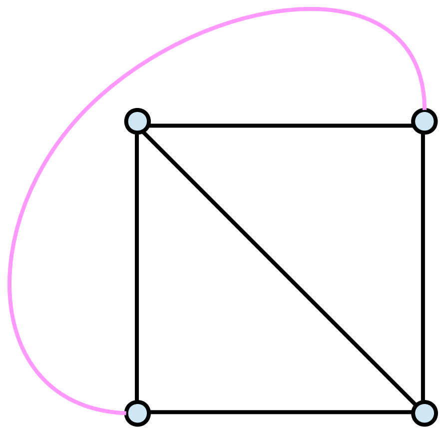
Planar Representation

K_4



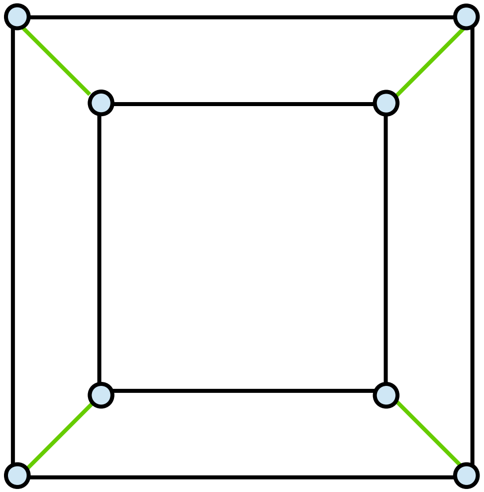


Q_3



No crossing

K_4 Planar

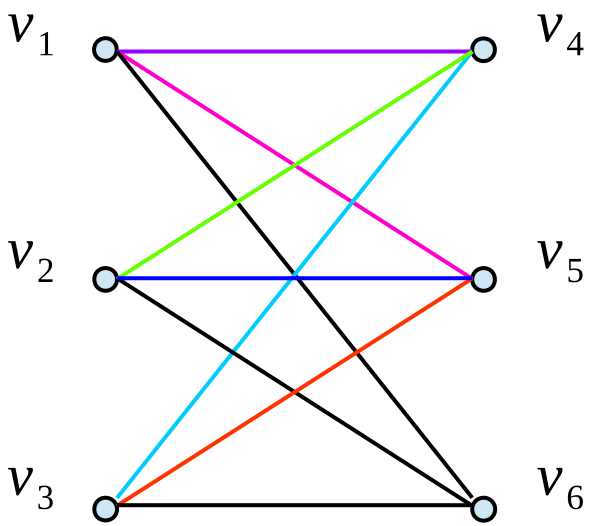


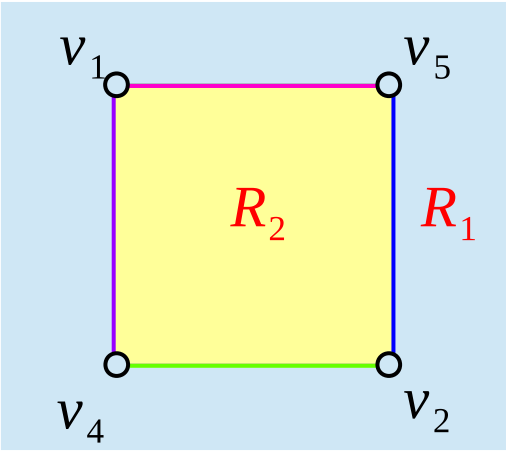
No crossing

Q_3 Planar

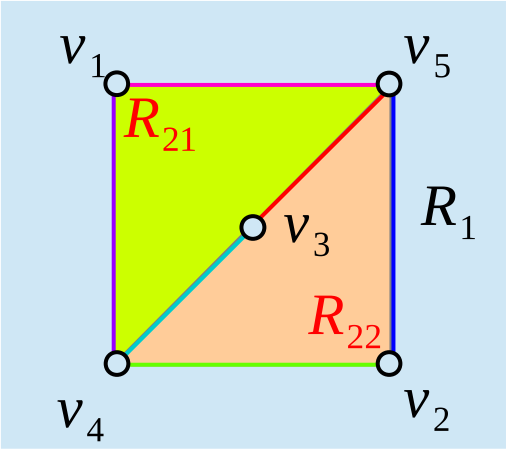
Discrete Mathematics, Rosen

Non-planar Graph $K_{3,3}$





no where v_6

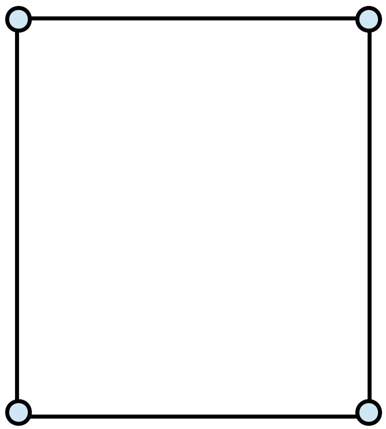


Non-planar

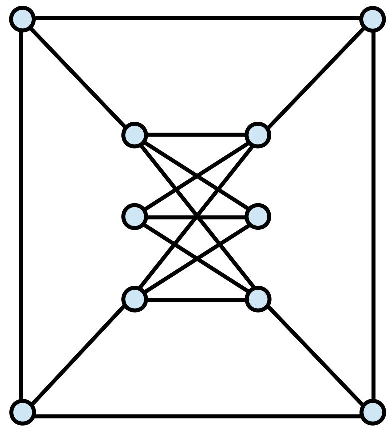
Discrete Mathematics, Rosen

Non-planar graph examples

Planar



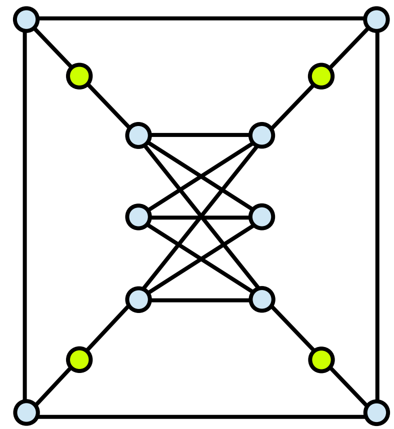
Non-planar



contains $K_{3,3}$

non-planar subgraph

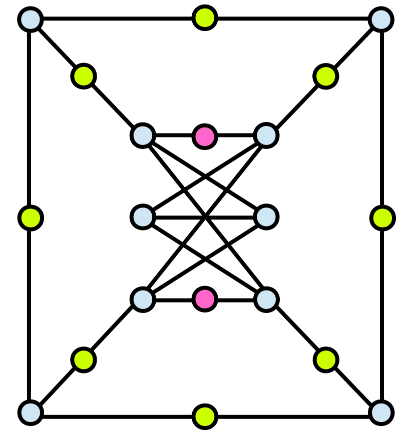
Non-planar



contains $K_{3,3}$

non-planar subgraph

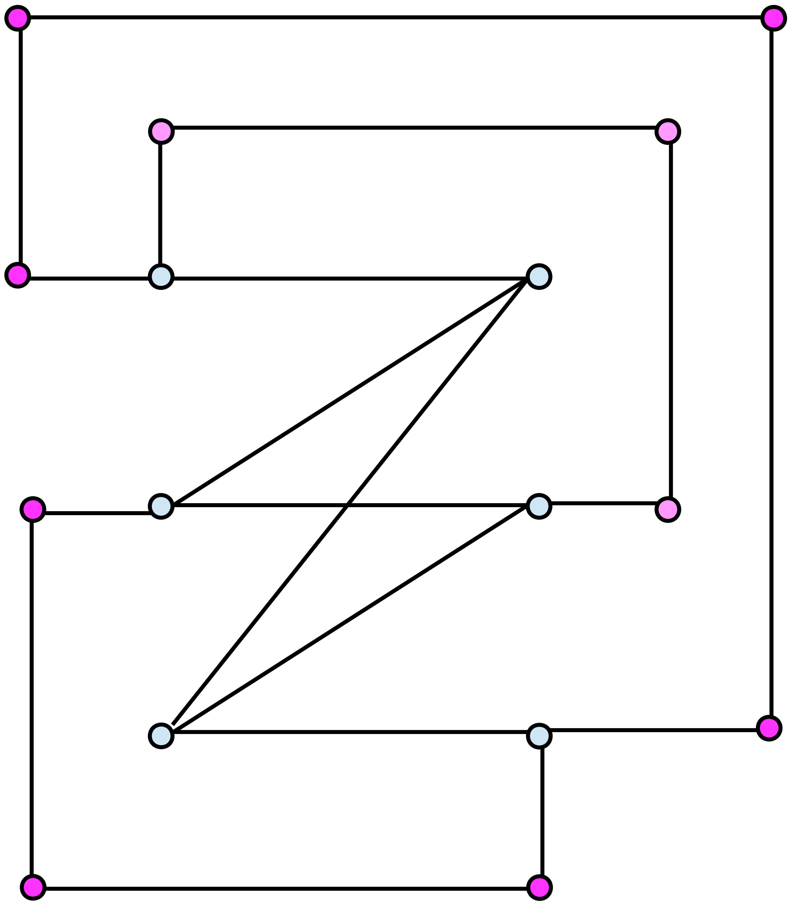
Non-planar

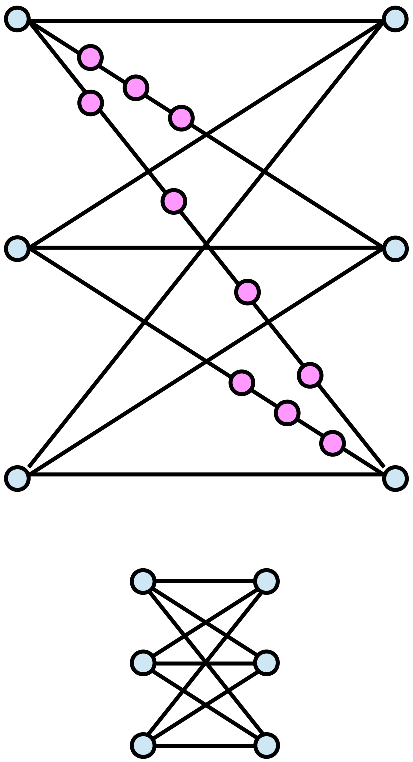


contains a <u>subdivision</u> of $K_{3,3}$

non-planar subgraph

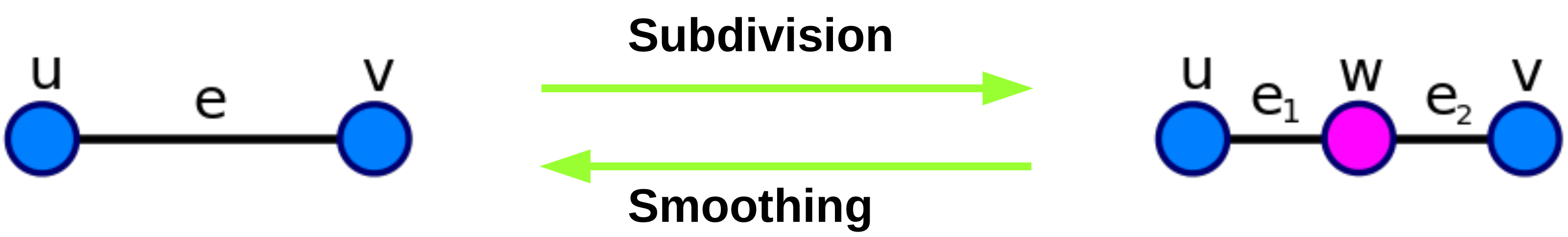
Homeomorphic





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

Subdivision and Smoothing



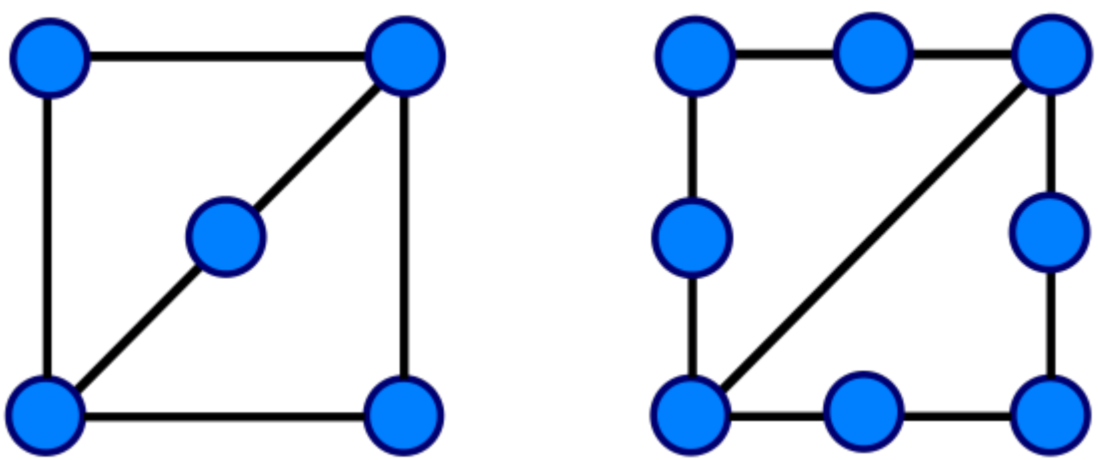
https://en.wikipedia.org/wiki/Planar_graph

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

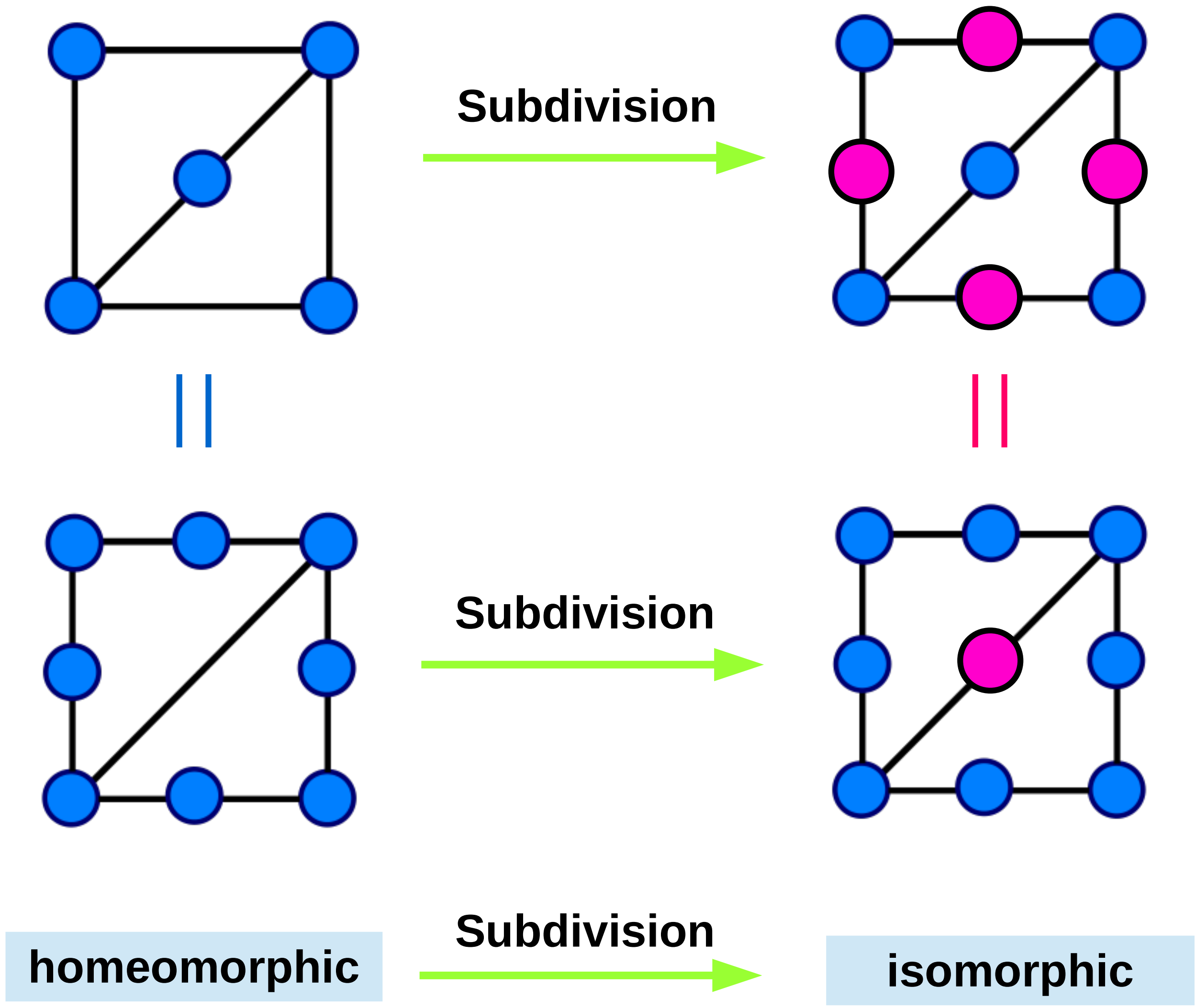
homeo (identity, sameness)

iso (equal)



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

Homeomorphism Examples



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

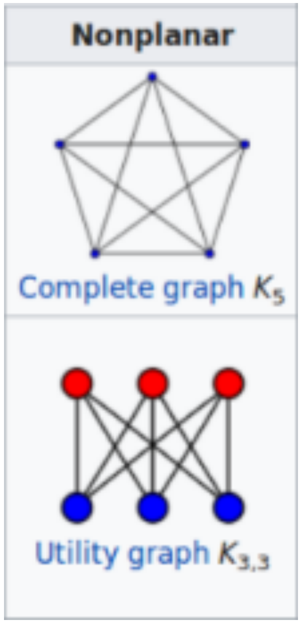
Embedding on a surface

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 **K_5** (complete graph on five vertices) or
> **$K_{3,3}$** (complete bipartite graph on six vertices,
> three of which connect to each other of the other three).

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

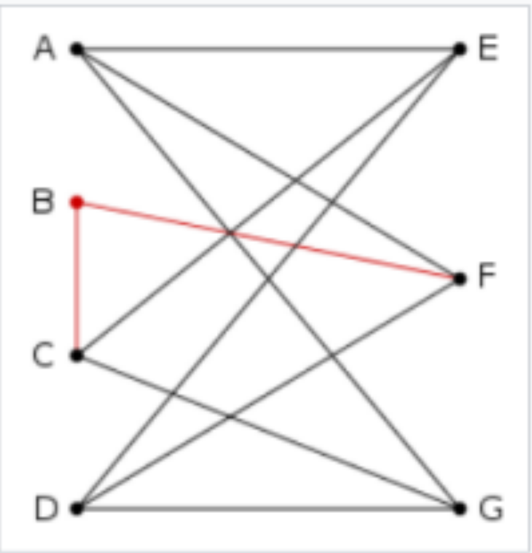


https://en.wikipedia.org/wiki/Planar_graph

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).

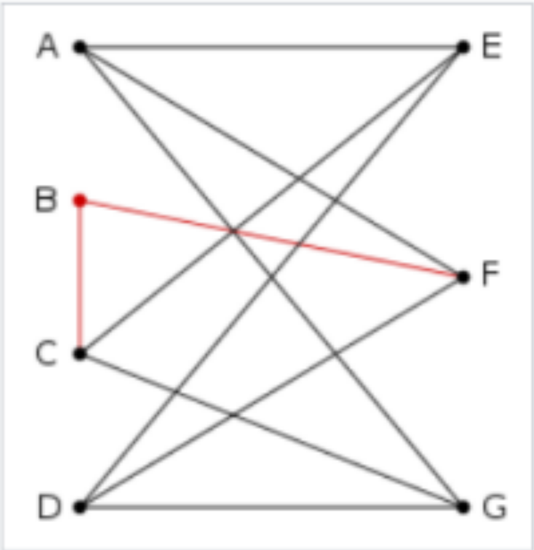
A subdivision of a graph results
from inserting vertices into edges
(changing an edge •——• to •—•—•)
zero or more times.



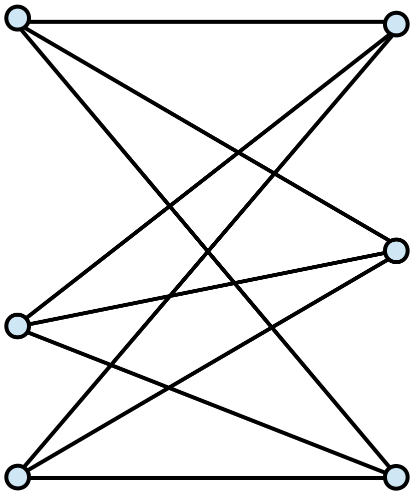
An example of a graph with no K_5 or $K_{3,3}$ subgraph. However, it contains a subdivision of $K_{3,3}$ and is therefore non-planar.

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

Kuratowski's Theorem

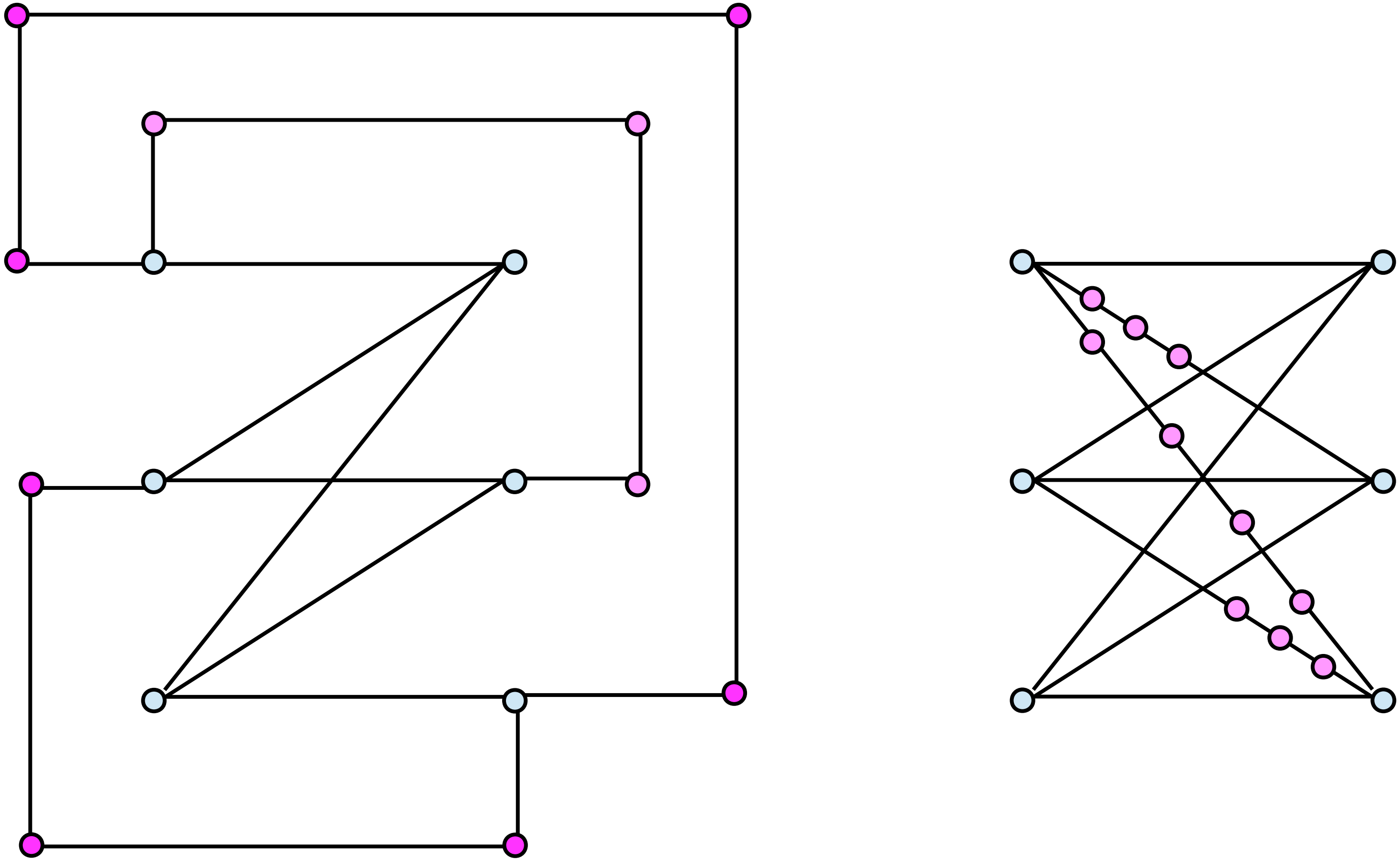


An example of a graph with no K_5 or $K_{3,3}$ subgraph. However, it contains a subdivision of $K_{3,3}$ and is therefore non-planar.



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

A subdivision of $K_{3,3}$



Euler's Formula

Euler's formula states that if a **finite**, **connected**, **planar graph** is drawn in the plane without any edge intersections, and **v** is the number of **vertices**, **e** is the number of **edges** and **f** is the number of **faces** (regions bounded by edges, including the outer, infinitely large region), then

$$\mathbf{v - e + f = 2}$$

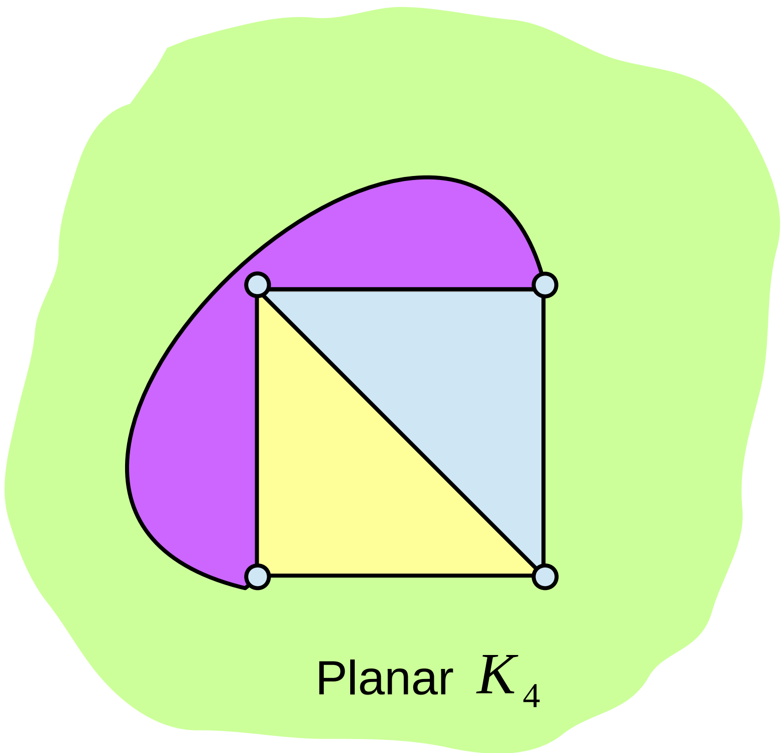
https://en.wikipedia.org/wiki/Planar_graph

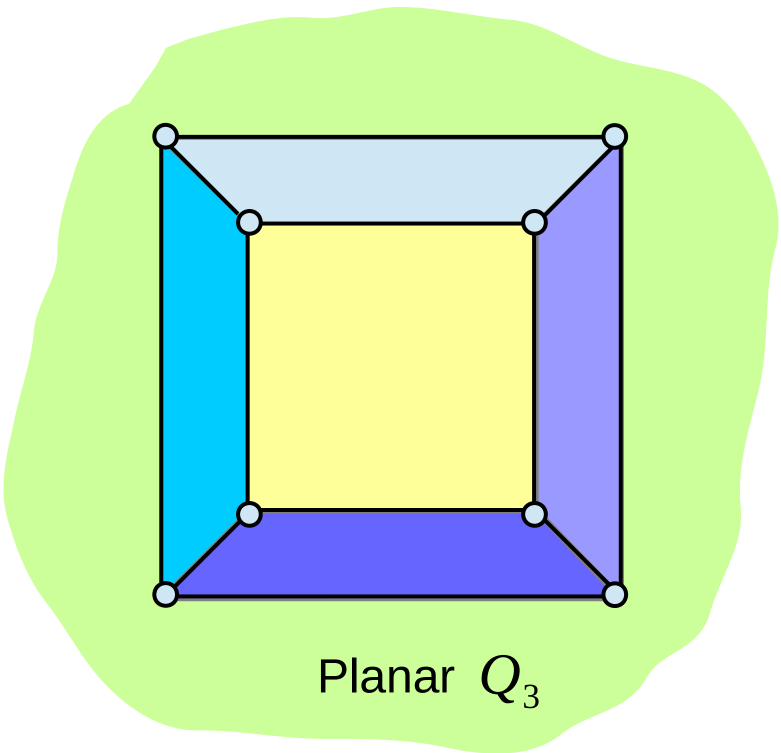
Euler's Formula Examples

$v = 4$
$e = 6$
$f = 4$

$v - e + f = 2$

$v = 8$
$e = 12$
$f = 6$





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

Corollary 1

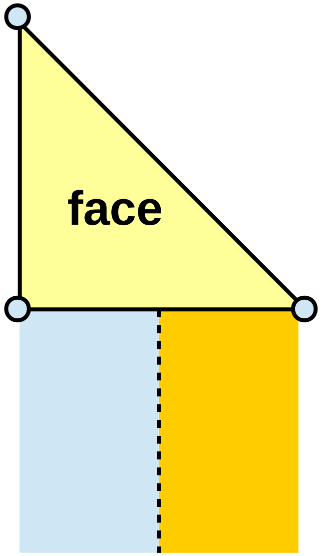
In a **finite**, **connected**, **simple**, **planar graph**,

any **face** (except possibly the outer one) is bounded by at least three **edges** and

every **edge** touches at most two **faces**;

using Euler's formula, one can then show that these graphs are **sparse** in the sense that if $v \geq 3$:

$$e \leq 3v - 6$$

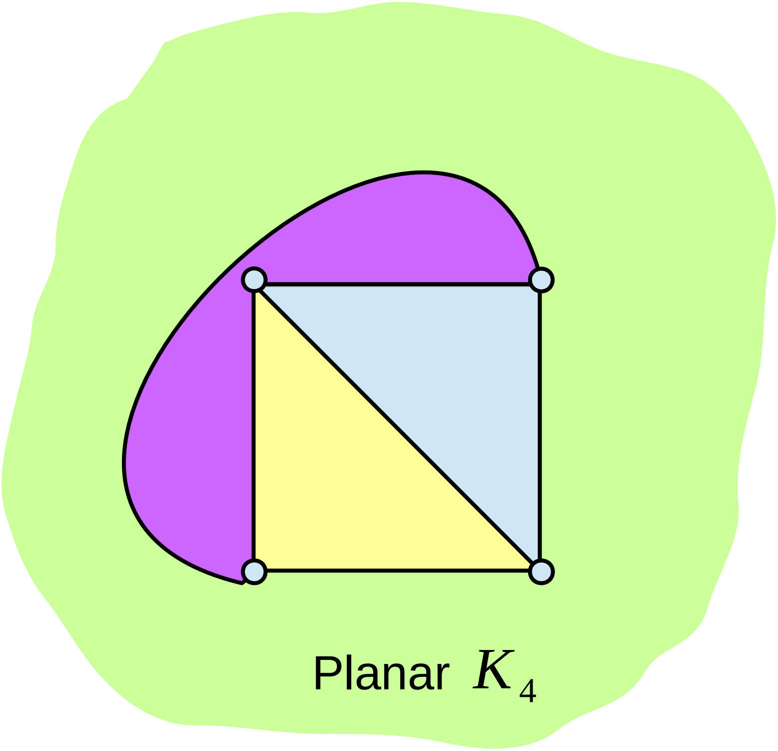


https://en.wikipedia.org/wiki/Planar_graph

Corollary 1 Examples

v = 4
e = 6
f = 4

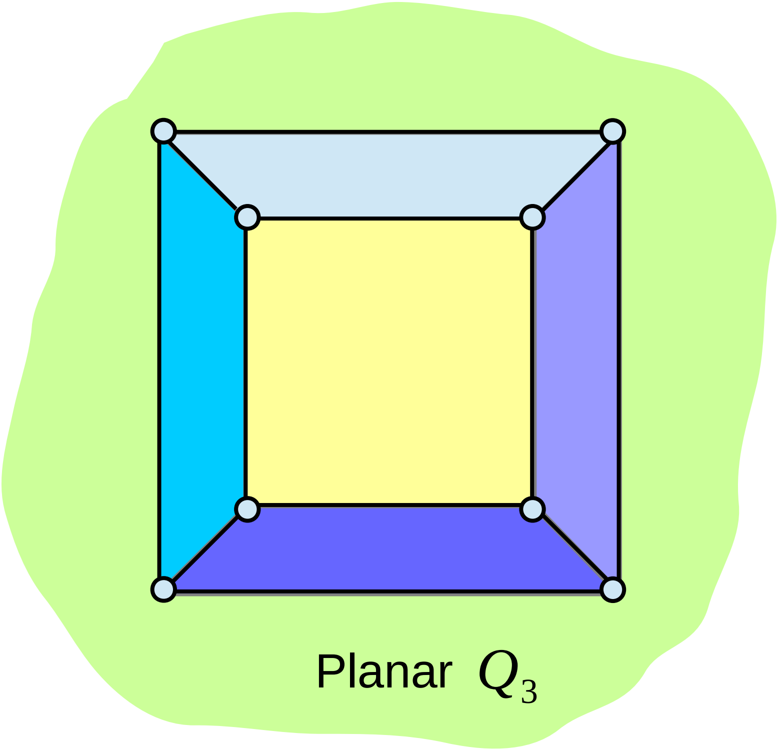
e ≤ 3 v − 6
6 ≤ 3·4 − 6



Planar K_4

v = 8
e = 12
f = 6

e ≤ 3 v − 6
12 ≤ 3·8 − 6



Planar Q_3

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

Euler’s Formula : Corollary 2

In a **finite**, **connected**, **simple**, **planar graph**,

Every vertex has a **degree** not exceeding **5**.

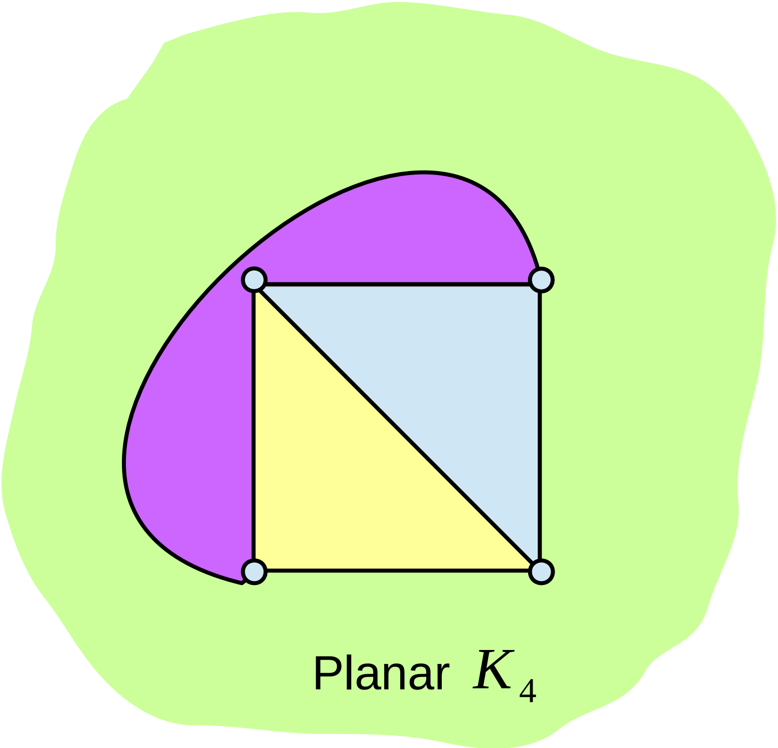
deg(v) ≤ 5

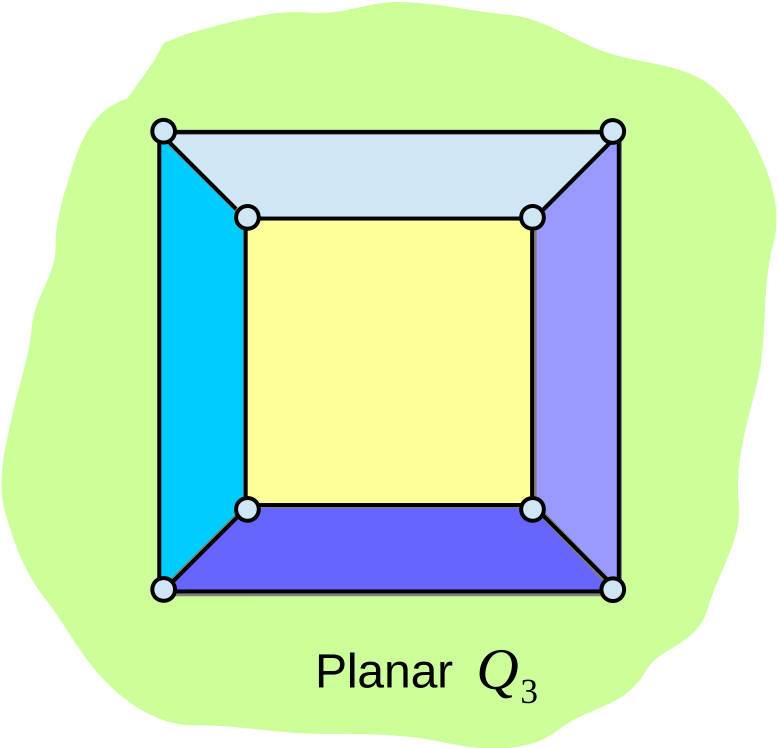
https://en.wikipedia.org/wiki/Planar_graph

Corollary 2 Examples

degree: 3 **deg(v) ≤ 5**

degree: 3 **deg(v) ≤ 5**





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

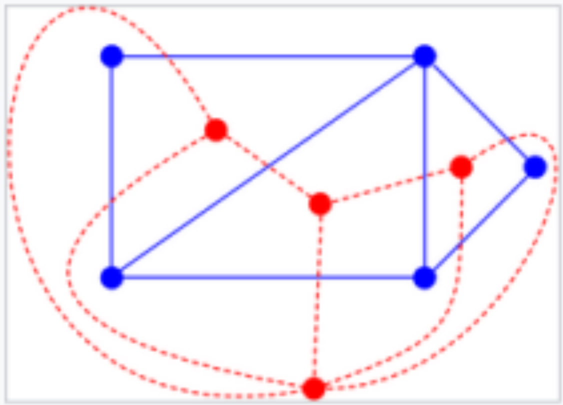
Dual Graph

the dual graph of a plane graph G is a graph that has a **vertex** for each **face** of G.

The dual graph has an **edge** whenever two **faces** of G are separated from each other by an **edge**,

and a **self-loop** when the same **face** appears on both sides of an **edge**.

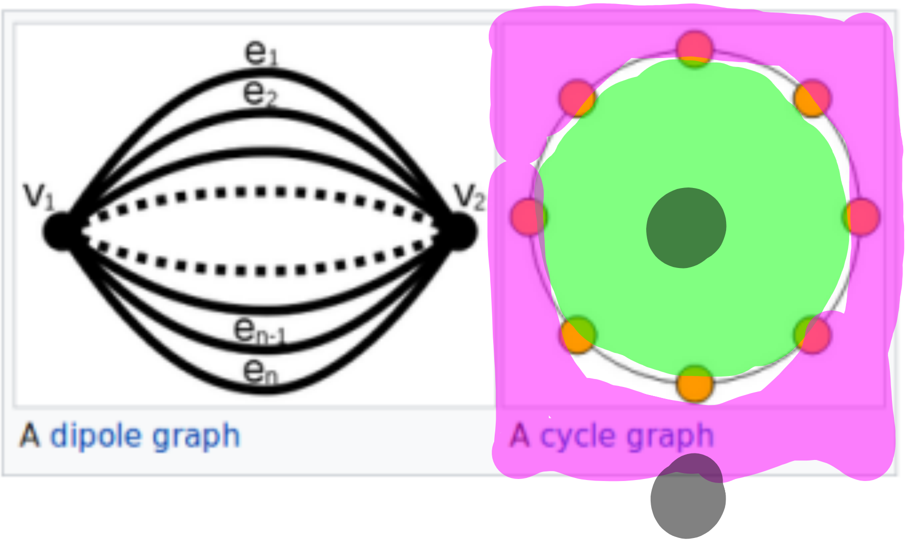
each **edge e** of G has a corresponding **dual edge**, whose endpoints are the **dual vertices** corresponding to the **faces** on either side of **e**.



The red graph is the dual graph of the blue graph, and *vice versa*.

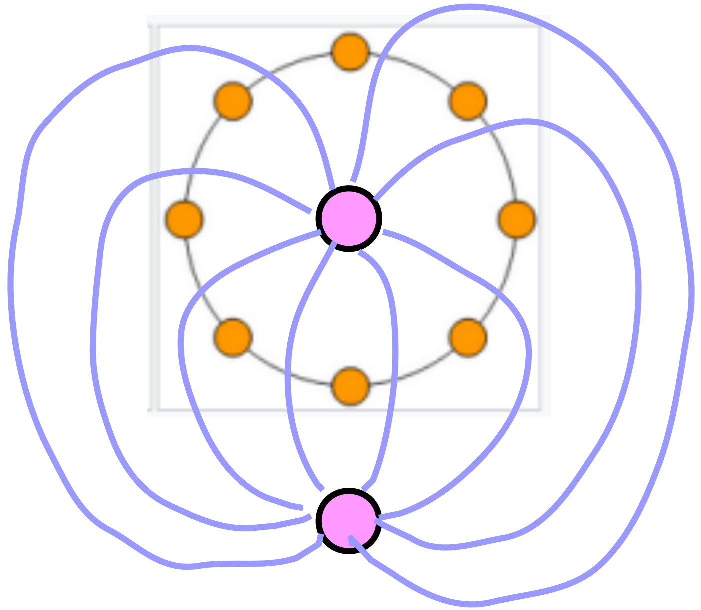
https://en.wikipedia.org/wiki/Dual_graph

Dipoles and Cycles



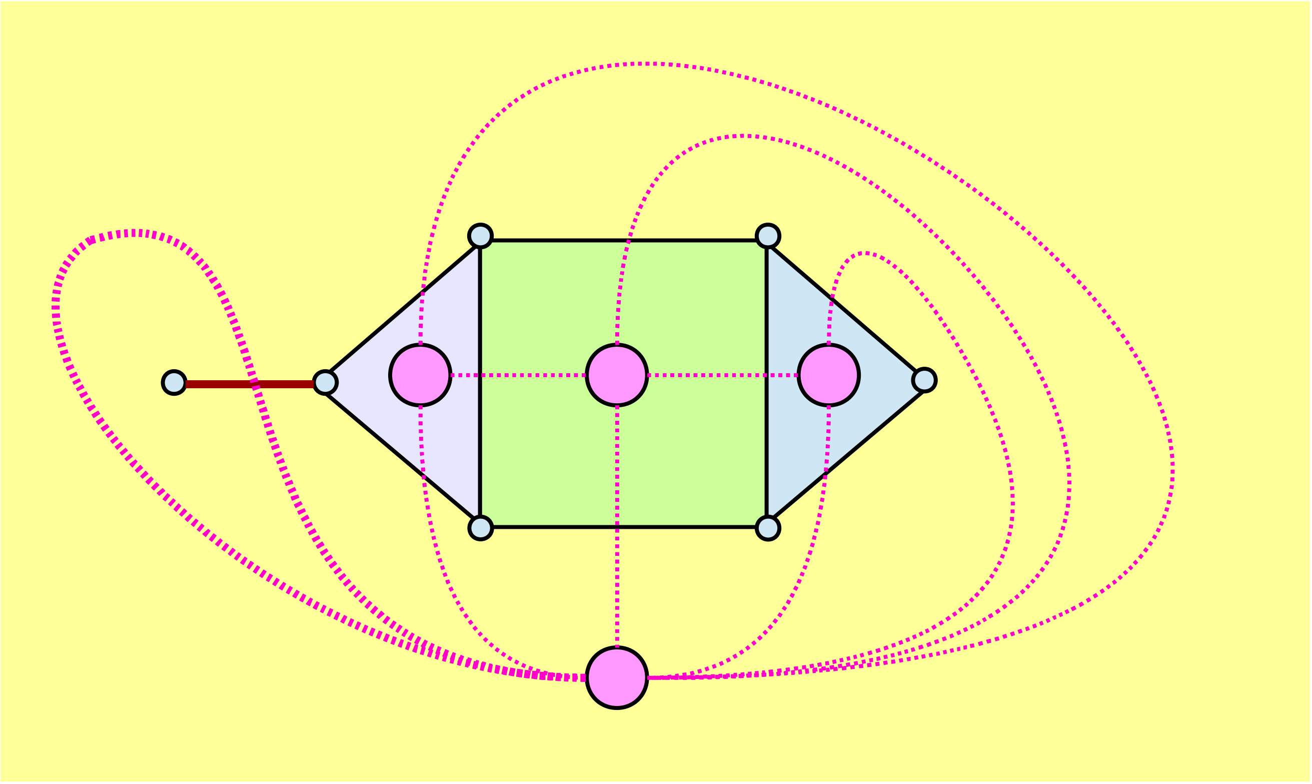
A dipole graph

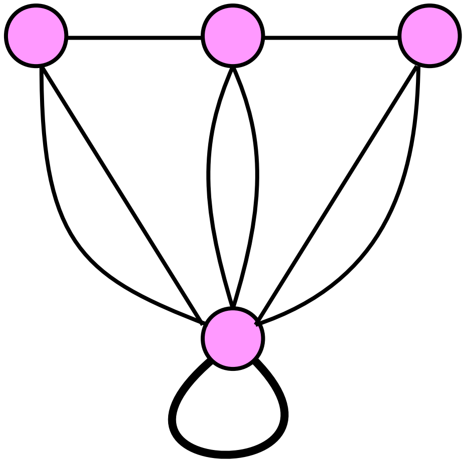
A cycle graph



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

Self-loop in a dual graph





a **self-loop** when the <u>same</u> **face** appears on <u>both</u> <u>sides</u> of an **edge**.

https://www.math.hmc.edu/~kindred/cuc-only/math104/lectures/lect17-slides-handout.pdf

Correspondence between G and G*

Vertices of G*	Faces of G
Edges of G*	Edges of G
Multigraph	Dual of a plane graph
Loops of G*	Cut edge of G
Multiple edges of G*	distinct faces of G with multiple common boundary edges

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

Cut

a **cut** is a **partition** of the **vertices** of a graph
into two disjoint **subsets**.

Any **cut** determines a **cut-set**
the **set** of **edges** that have one endpoint
in each subset of the partition.

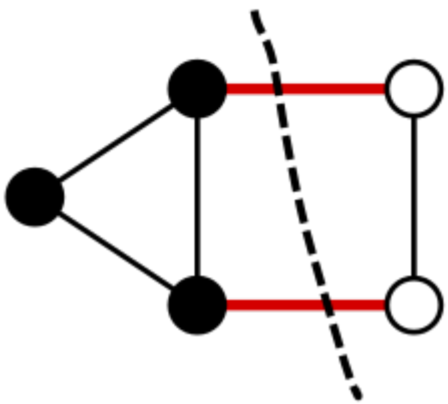
These edges are said to **cross** the cut.

In a connected graph, each **cut-set** determines a unique **cut**,
and in some cases cuts are identified with their **cut**-**sets** rather than
with their **vertex** partitions.

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

Minimum Cut

A cut is minimum if the size or weight of the cut
is not larger than the size of any other cut.

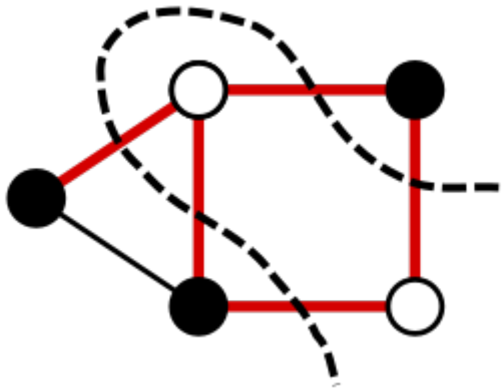


the size of this cut is 2,
and there is no cut of size 1
because the graph is bridgeless.

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

Maximum Cut

A cut is maximum if the size of the cut
is not smaller than the size of any other cut.



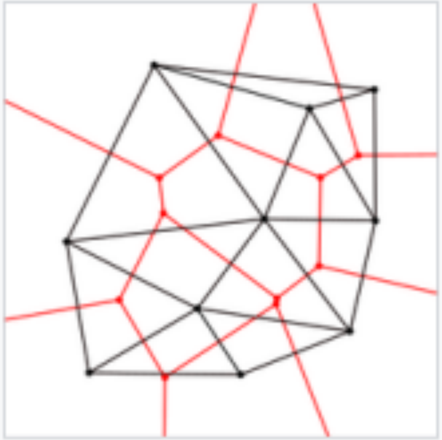
the size of the cut is equal to 5,
and there is no cut of size 6,
or |E| (the number of edges),
because the graph is not bipartite
(there is an odd cycle).

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

Infinite Graphs and Tessellations

The concept of duality applies as well
to **infinite graphs** embedded in the plane
as it does to **finite graphs**.

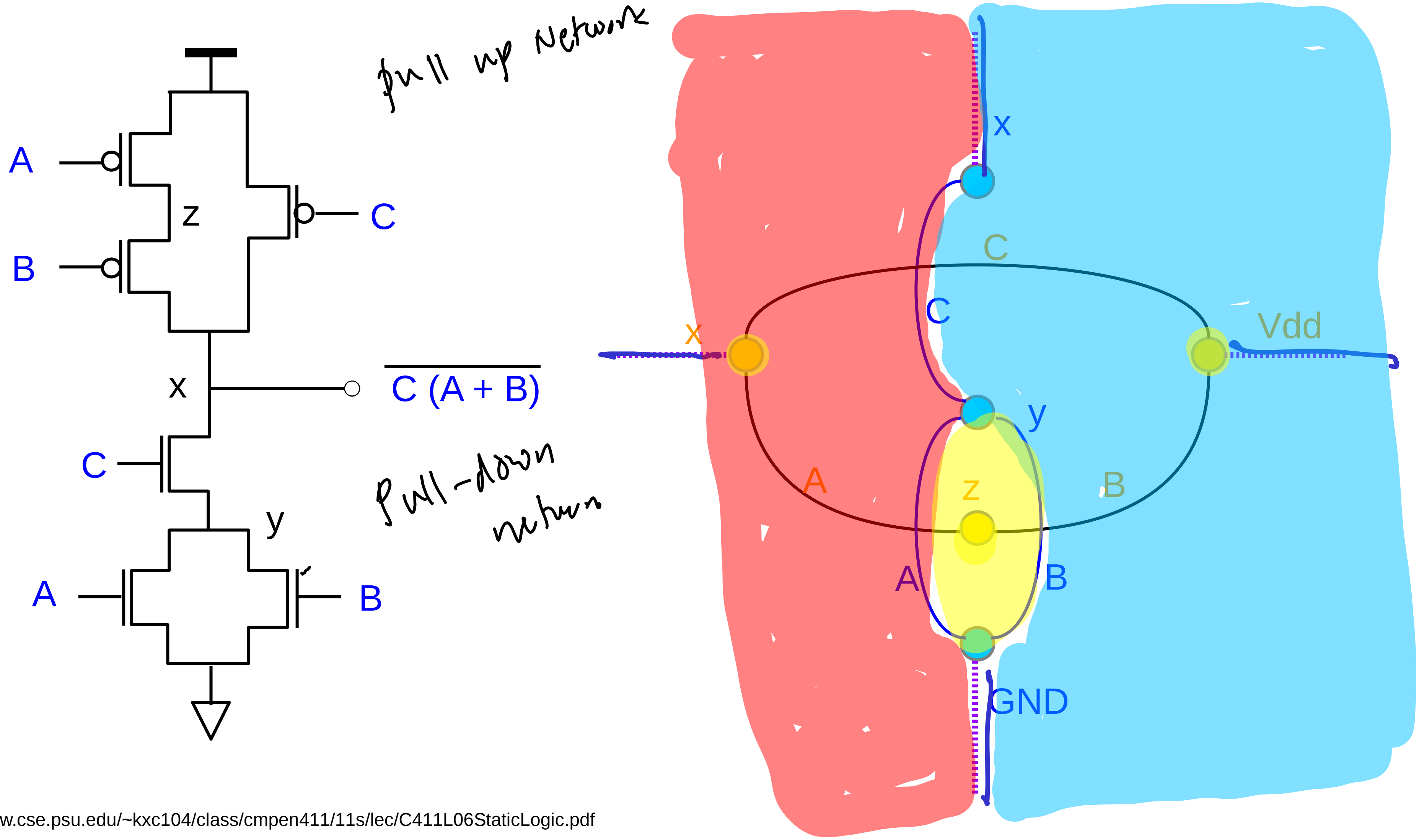
When all faces are bounded regions
surrounded by a cycle of the graph,
an **infinite planar** graph embedding
can also be viewed as a **tessellation** of the plane,
a covering of the plane by closed disks
(the **tiles** of the **tessellation**) whose interiors
(the **faces** of the **embedding**) are disjoint open disks.



A Voronoi diagram (red) and Delaunay triangulation (black) of a finite point set (the black points)

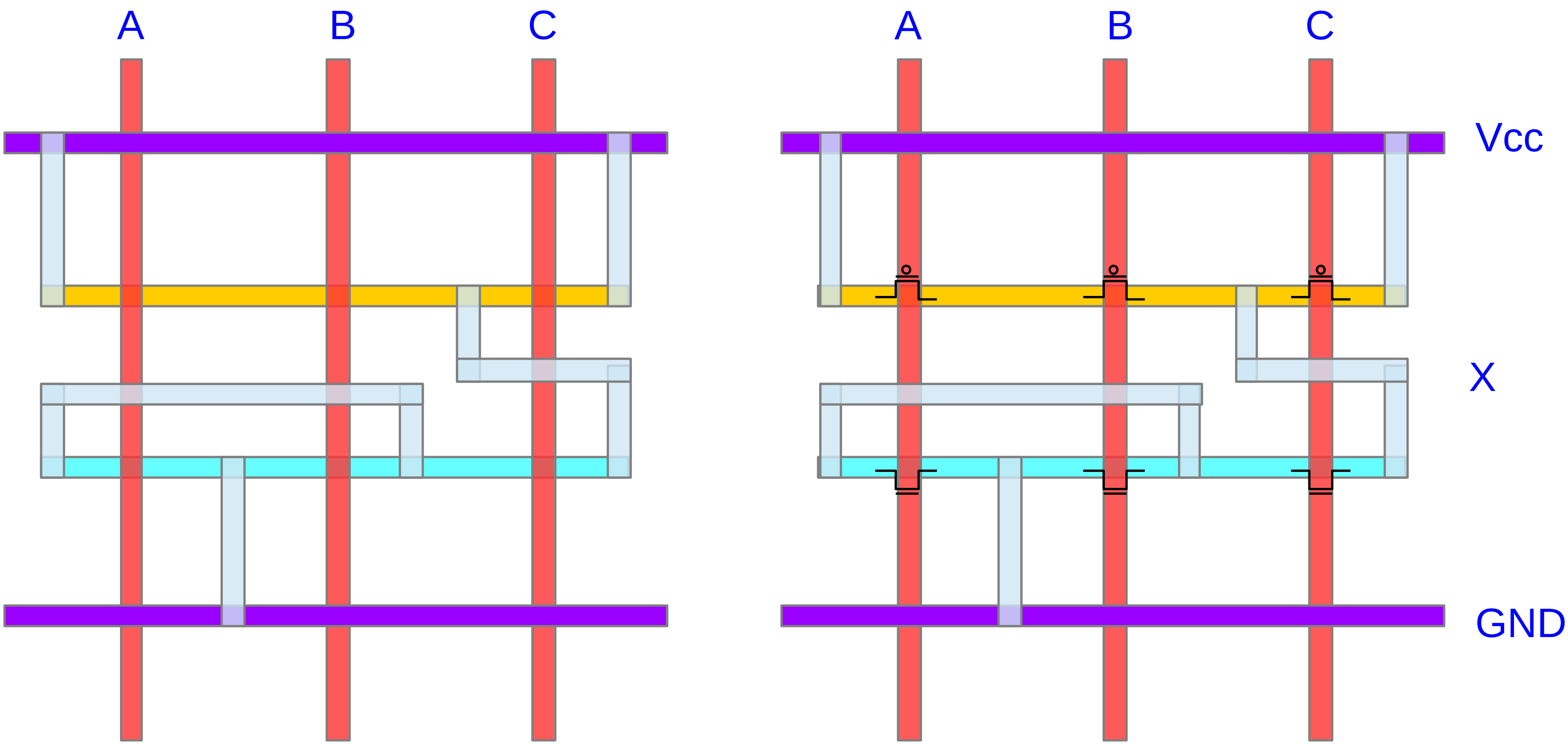
https://en.wikipedia.org/wiki/Dual_graph

Dual Logic Graph



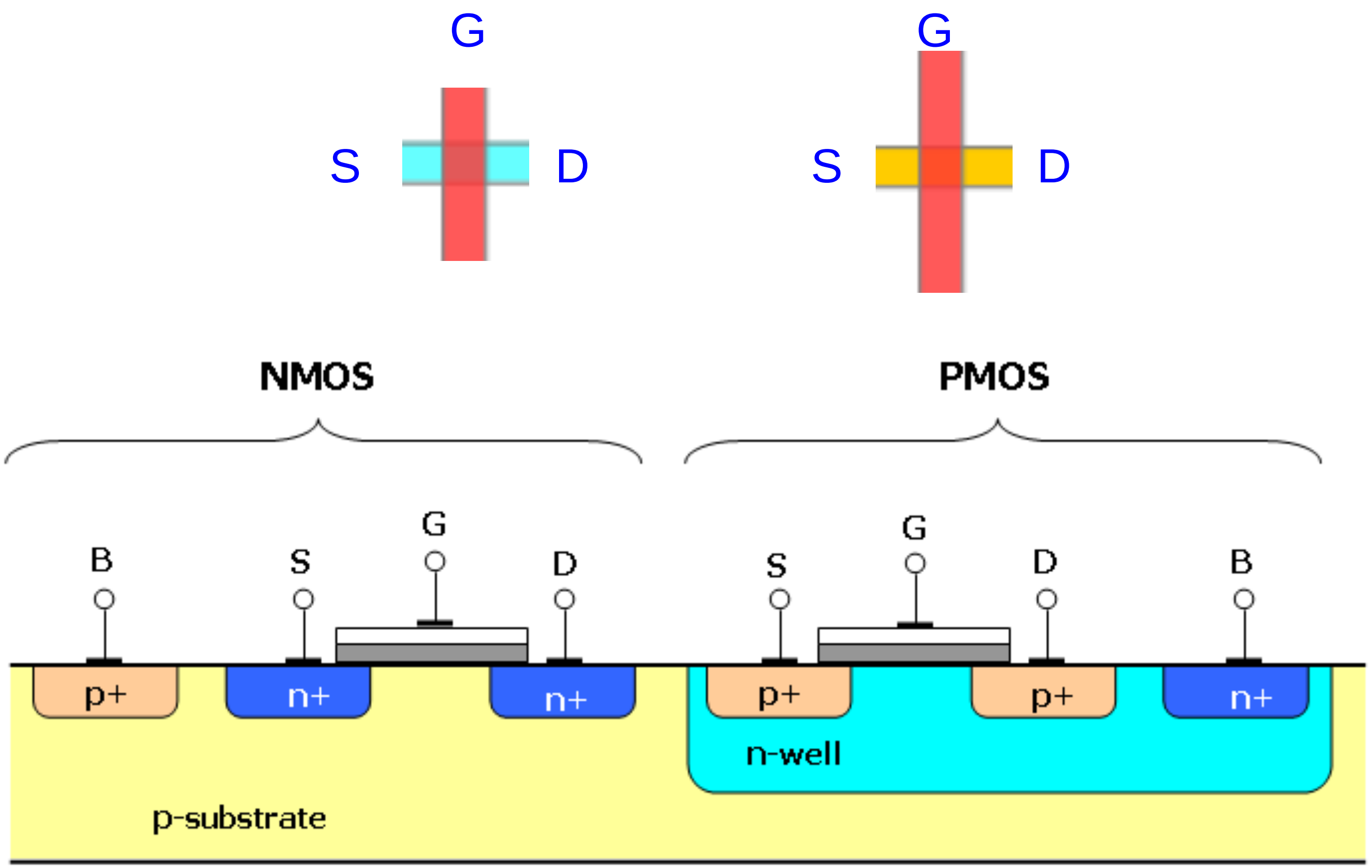
http://www.cse.psu.edu/~kxc104/class/cmpen411/11s/lec/C411L06StaticLogic.pdf

Stick Layout



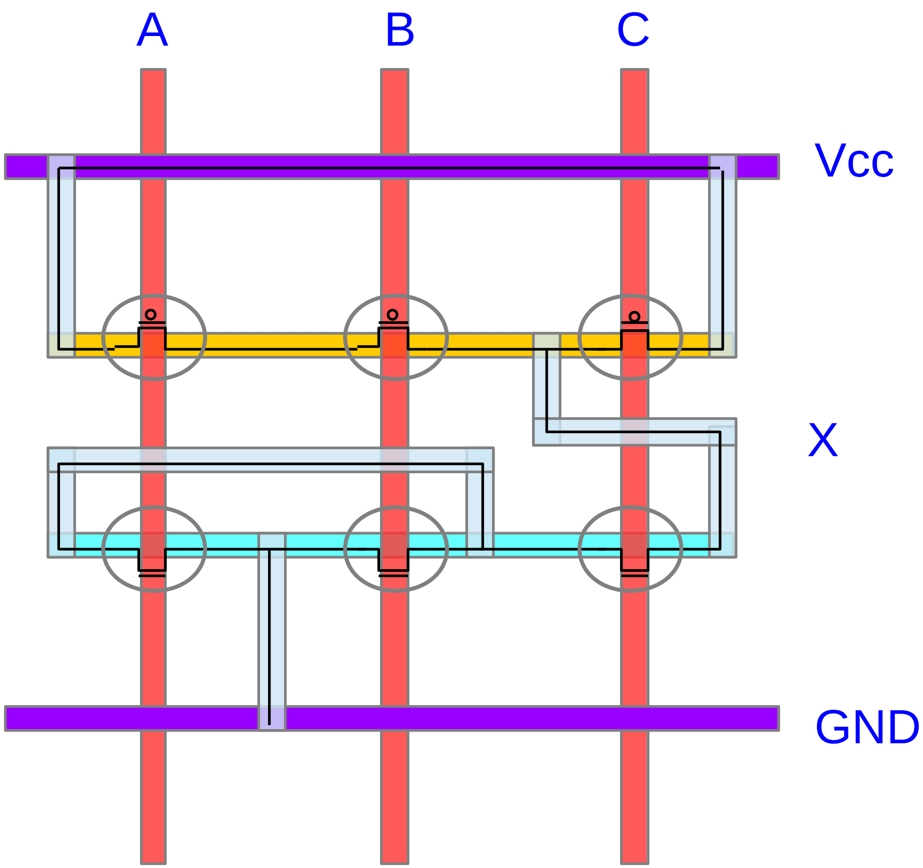
http://www.cse.psu.edu/~kxc104/class/cmpen411/11s/lec/C411L06StaticLogic.pdf

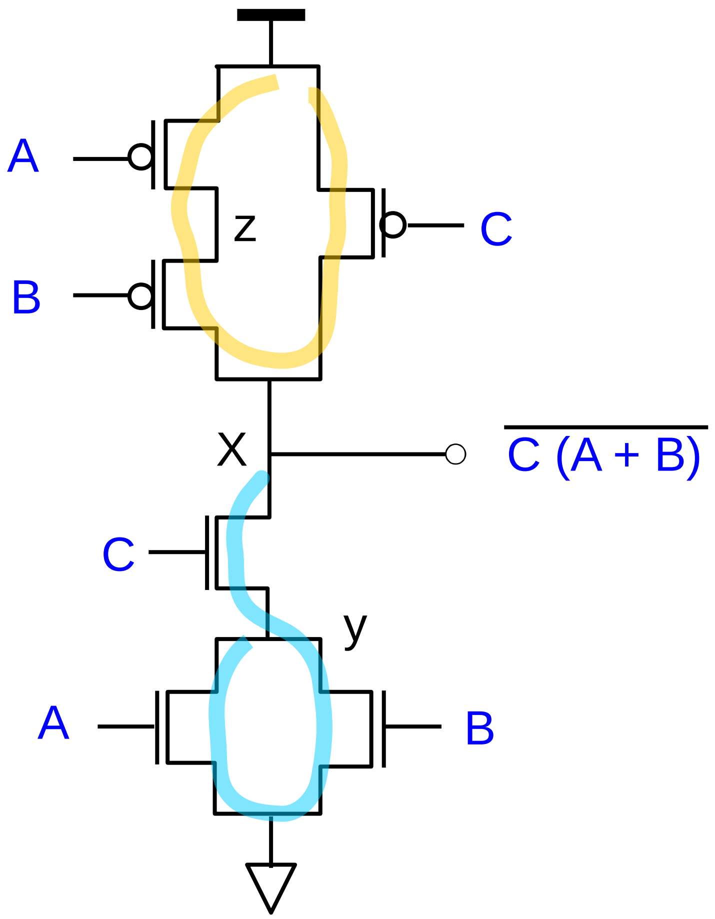
CMOS Transistors and Stick Layout



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

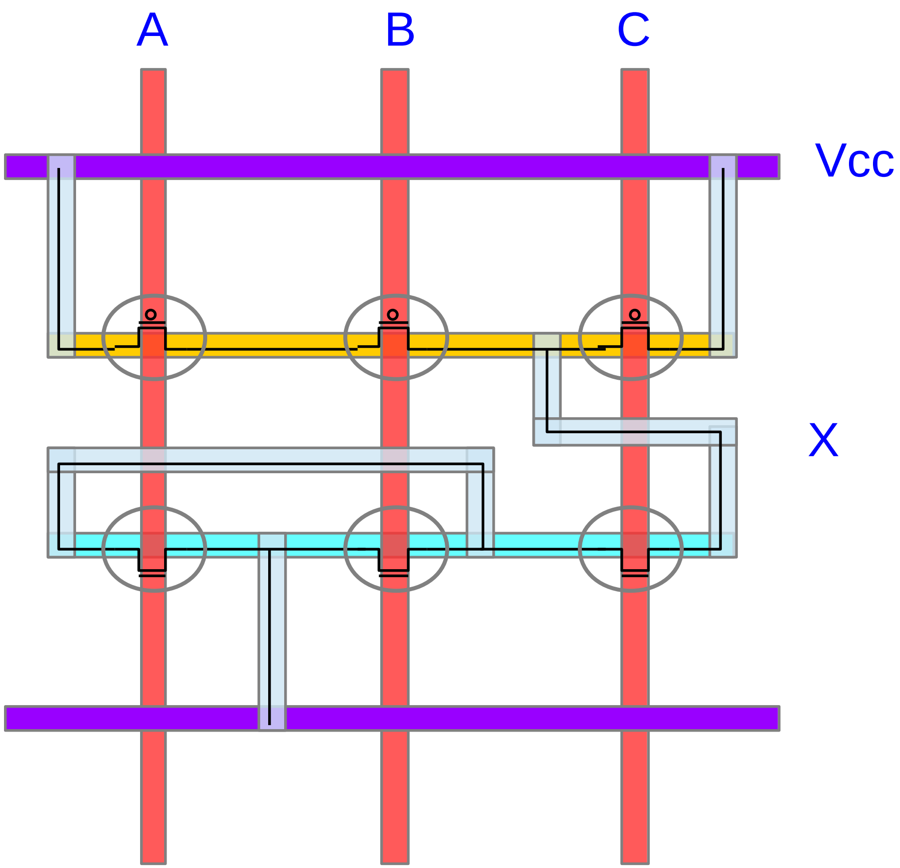
Single-Strip Stick Graph and Logic Graph



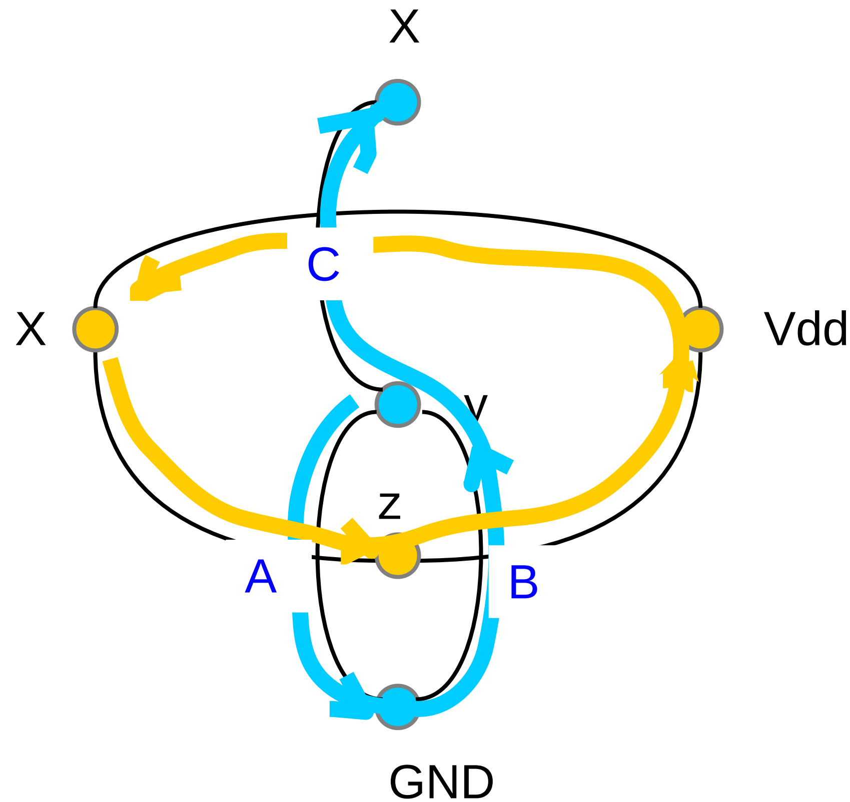


http://www.cse.psu.edu/~kxc104/class/cmpen411/11s/lec/C411L06StaticLogic.pdf

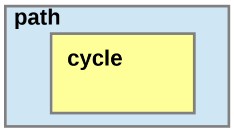
Stick Graph and Logic Diagram



uninterrupted diffusion strip

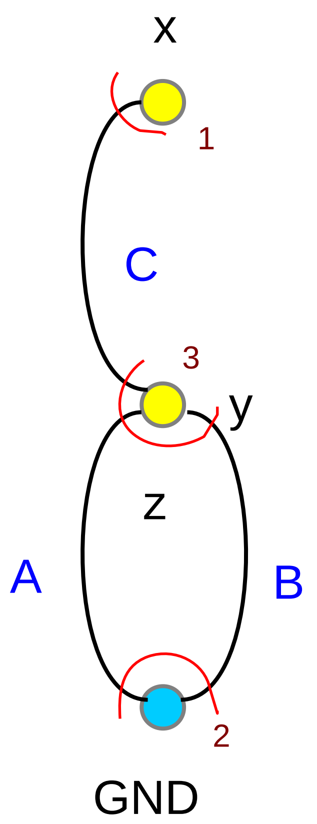


consistent Euler **paths** (PUN & PDN)

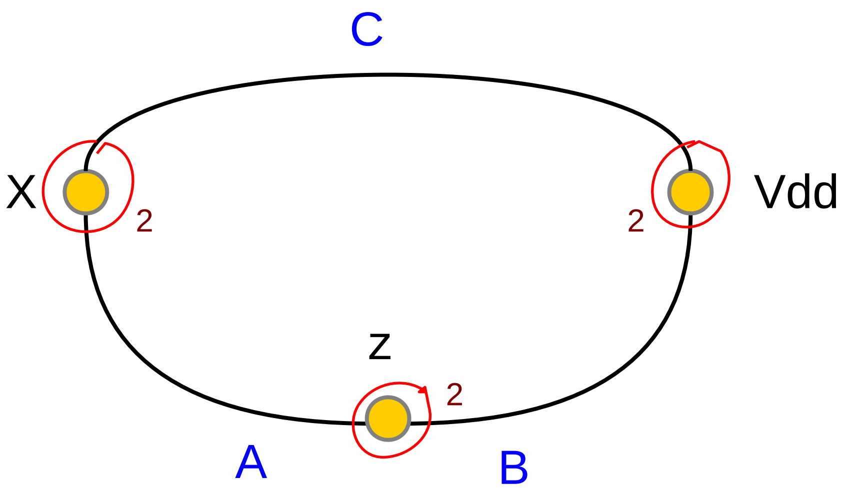


http://www.cse.psu.edu/~kxc104/class/cmpen411/11s/lec/C411L06StaticLogic.pdf

Stick Graph and Logic Diagram



Eulerian Trail



Eulerian Circuit

http://www.cse.psu.edu/~kxc104/class/cmpen411/11s/lec/C411L06StaticLogic.pdf

References

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

Graph Coloring (9A)

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

This document was produced by using LibreOffice and Octave.

Graph Coloring

planar graph

graph coloring is a special case of graph labeling;

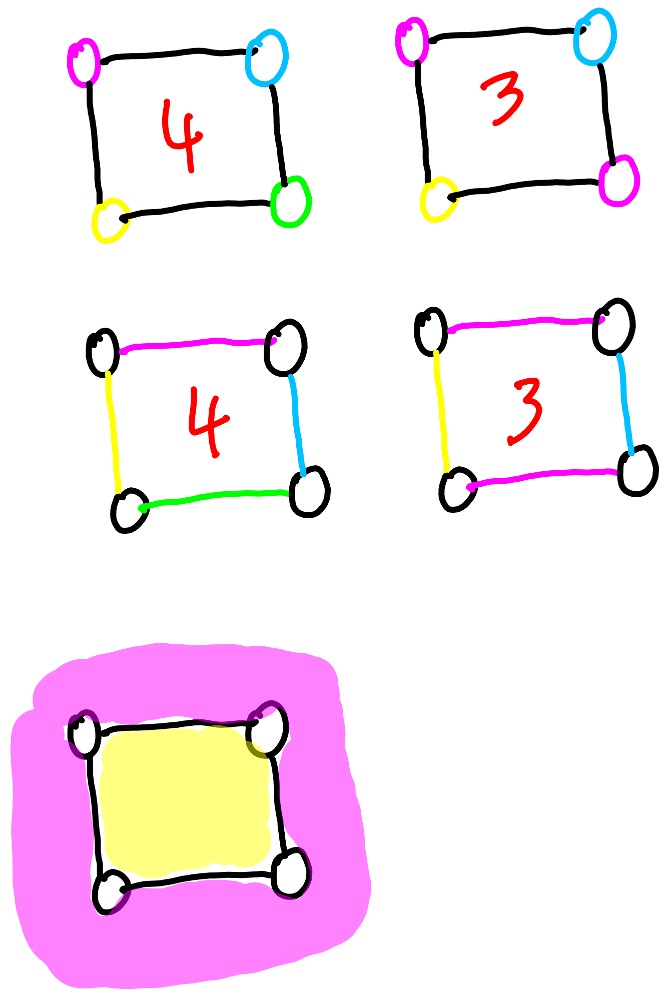
it is an assignment of labels (colors)
to elements of a graph subject to certain constraints.

V

a **vertex coloring**
is a way of coloring the vertices of a graph
such that no two adjacent vertices share the same color

an **edge coloring**
assigns a color to each edge so that no two adjacent
edges share the same color

a **face coloring** of a planar graph
assigns a color to each face or region so that no two
faces that share a boundary have the same color.



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

Graph Coloring Relations

an **edge coloring** of a graph
is just a **vertex coloring** of its **line graph**,

a **face coloring** of a plane graph
is just a **vertex coloring** of its **dual graph**.

However, non-vertex coloring problems
are often stated and studied as is.

a graph coloring means almost always a **vertex coloring**.

Since a vertex with a loop could never be properly colored, a **loopless** graph is generally assumed.

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

k-coloring and chromatic number

k-coloring
a coloring using at most **k colors**

chromatic number, **χ(G)**
the smallest number of colors
needed to color a graph **G**

A graph that can be assigned a (proper) **k-coloring** is
k-colorable

A graph whose **chromatic number** is exactly **k** is
k-chromatic

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

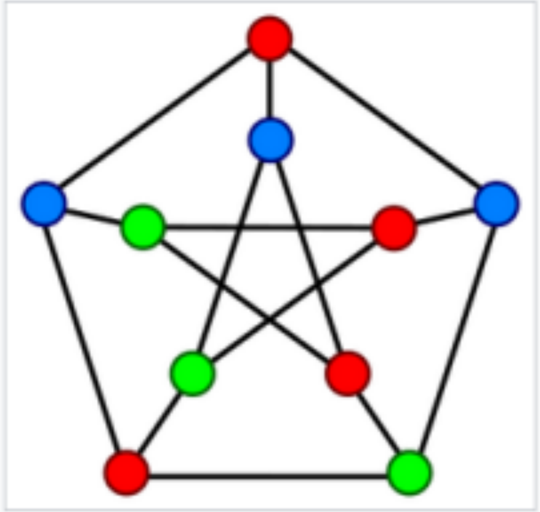
Color Class

A subset of vertices assigned to the same color is called a **color class**,

every such class forms an independent set.

a **k-coloring** is the same
as a **partition** of the vertex set
into **k** independent sets,

the terms **k-partite** and **k-colorable**
have the same meaning.



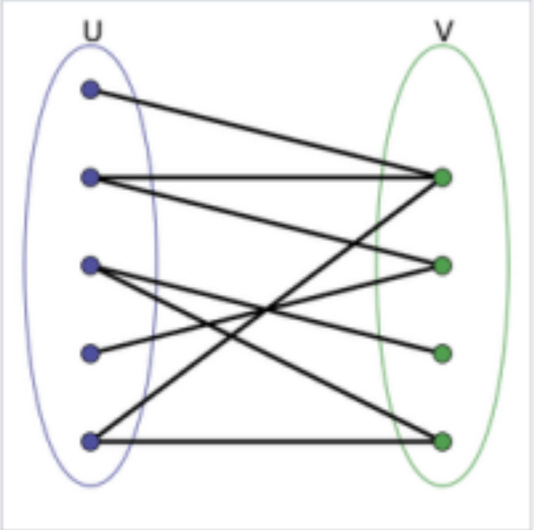
A proper vertex coloring of the Petersen graph with 3 colors, the minimum number possible.

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

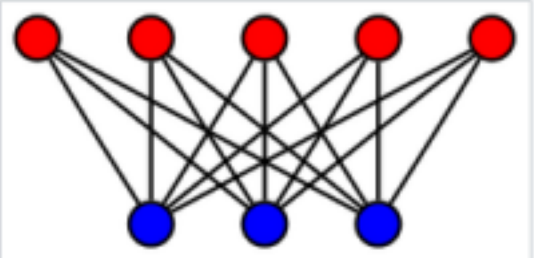
a bipartite graph (or bigraph) is a graph
whose vertices can be divided
into two disjoint and independent sets U and V
such that every edge connects
a vertex in U to one in V.

Vertex sets U and V are usually called
the parts of the graph.

Equivalently, a bipartite graph is
a graph that does not contain any **odd-length cycles**.



Example of a bipartite graph without cycles



A complete bipartite graph with m = 5 and n = 3

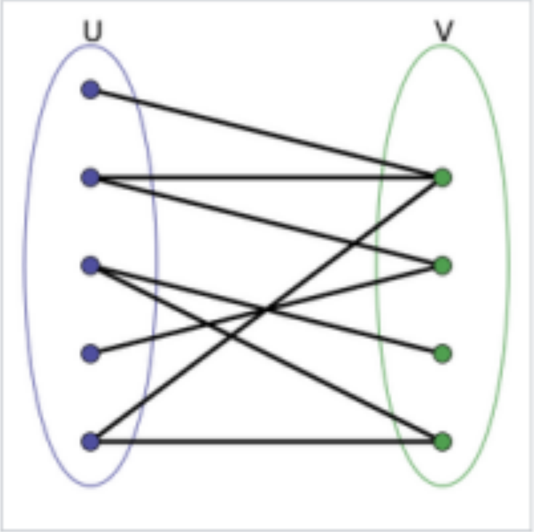
https://en.wikipedia.org/wiki/Bipartite_graph

Bipartite Graph : 2-colorable

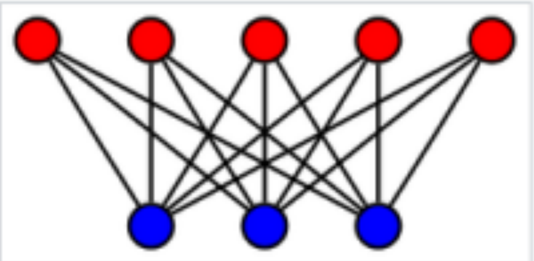
The two sets U and V may be thought of as a coloring of the graph with **two colors**:

if one colors all nodes in U blue,
and all nodes in V green,
each edge has endpoints of differing colors,
as is required in the graph coloring problem.

In contrast, such a coloring is impossible
in the case of a non-bipartite graph,
such as a triangle: 3 colors



Example of a bipartite graph without cycles


A complete bipartite graph with m = 5 and n = 3

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

Bipartite Graph : degree sequence

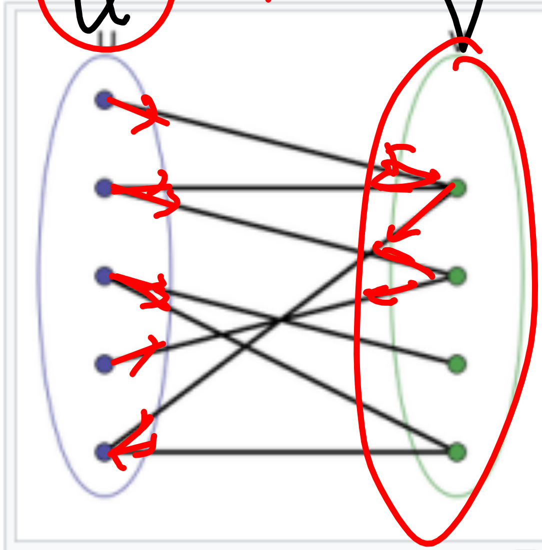
The degree sum formula
for a bipartite graph states that

$$\sum_{v \in V} \deg(v) = \sum_{u \in U} \deg(u) = |E| .$$

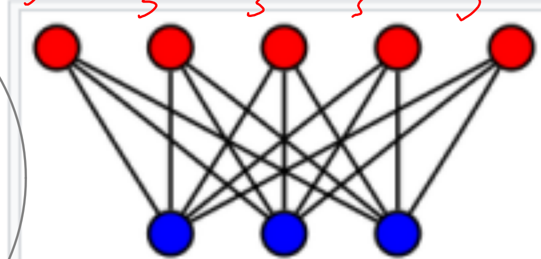
The degree sequence of a bipartite graph is the pair of lists each containing the degrees of the two parts U and V.

For example, the complete bipartite graph $K_{3,5}$ has degree sequence (5,5,5), (3,3,3,3,3)

$K_{5,3}$ has degree sequence (3,3,3,3,3), (5,5,5)



Example of a bipartite graph without cycles



A complete bipartite graph with m = 5 and n = 3

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

References

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

Tree Traversal (1A)

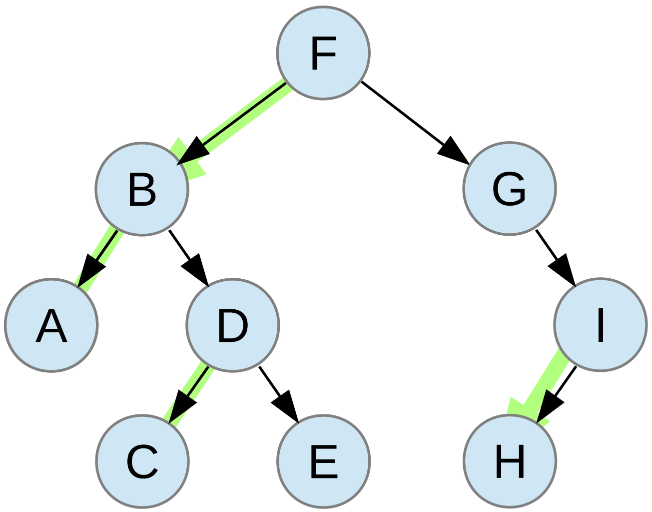
Please send corrections (or suggestions) to youngwlim@hotmail.com.

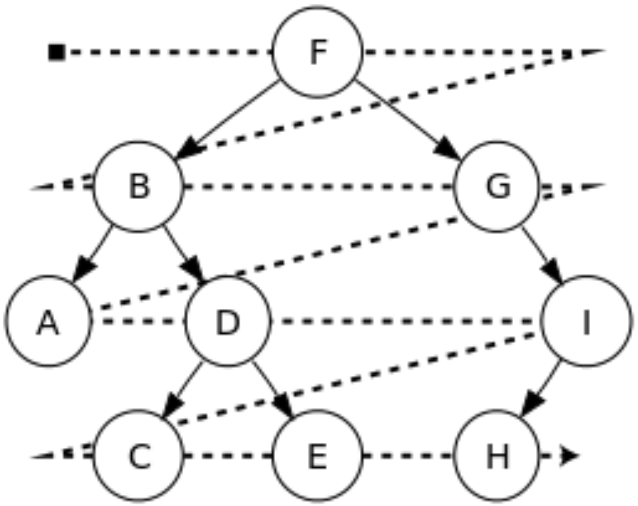
This document was produced by using LibreOffice and Octave.

Tree Traversal

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

Breadth First Search

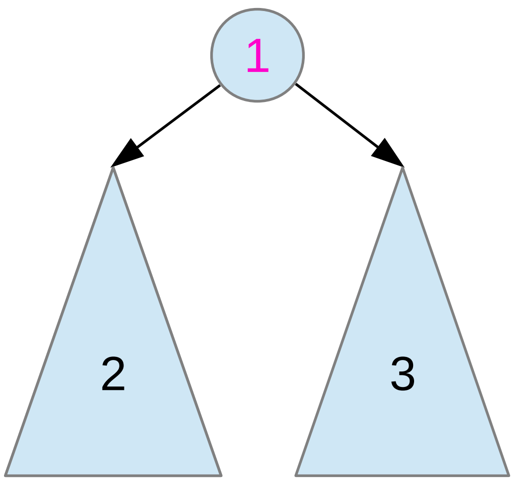




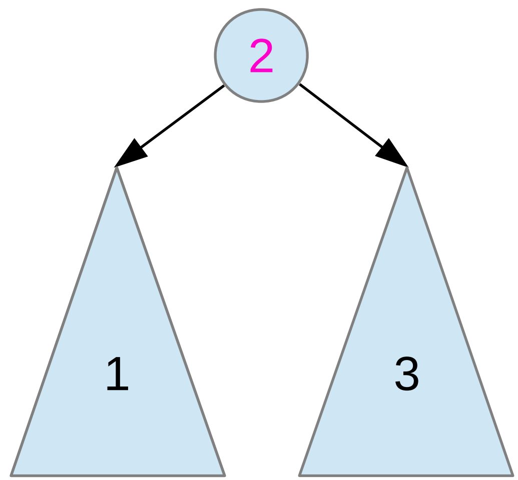
https://en.wikipedia.org/wiki/Morphism

Recursive Algorithms

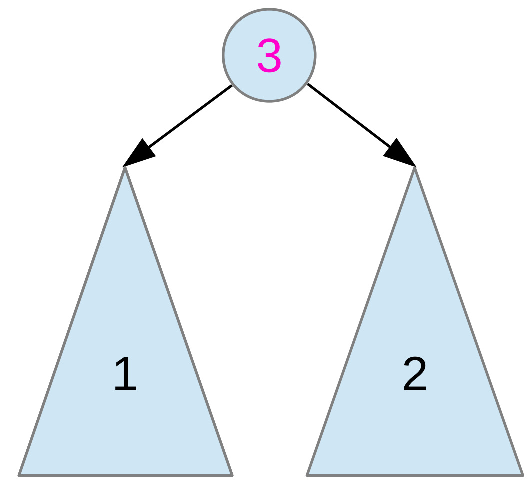
```
preorder(node)
  if (node = null)
    return
  visit(node)
  preorder(node.left)
  preorder(node.right)
```




```
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

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()
```

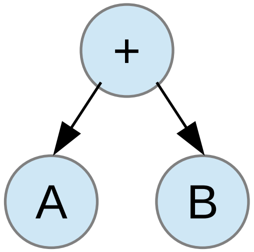
Infix, Prefix, Postfix Notations

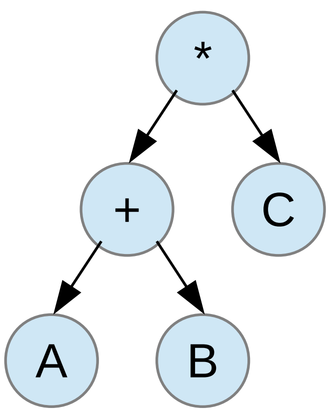
Infix Notation	Prefix Notation	Postfix Notation
A + B	+ A B	A B +
(A + B) * C	* + A B C	A B + C *
A * (B + C)	* A + B C	A B C + *
A / B + C / D	+ / A B / C D	A B / C D / +
((A + B) * C) – D	– * + A B C D	A B + C * D –

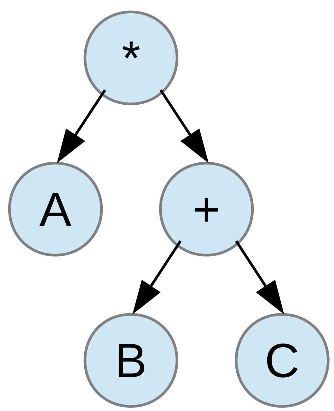
https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

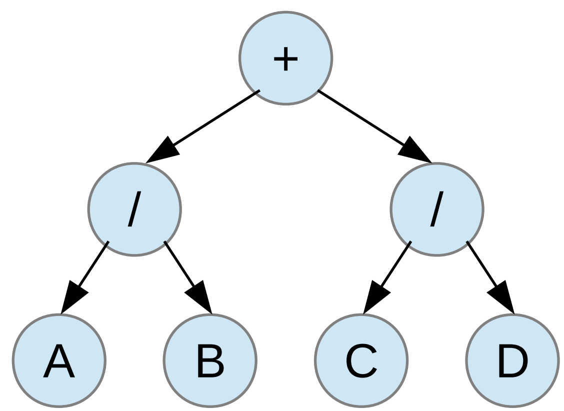
Infix, Prefix, Postfix Notations and Binary Trees

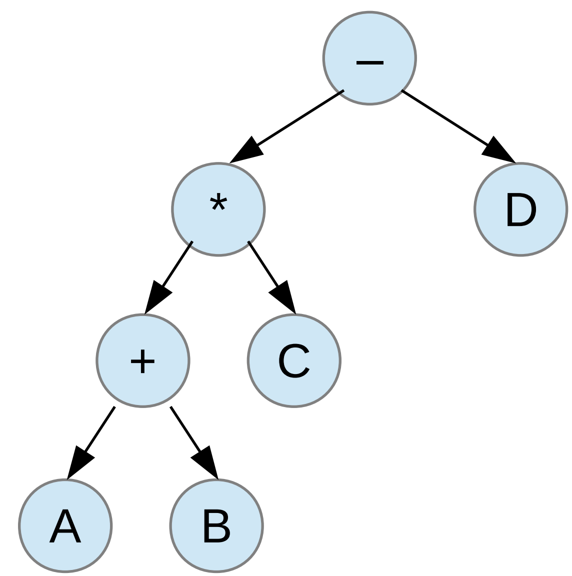
Infix Notation	Prefix Notation	Postfix Notation
A + B	+ A B	A B +
(A + B) * C	* + A B C	A B + C *
A * (B + C)	* A + B C	A B C + *
A / B + C / D	+ / A B / C D	A B / C D / +
((A + B) * C) – D	– * + A B C D	A B + C * D –







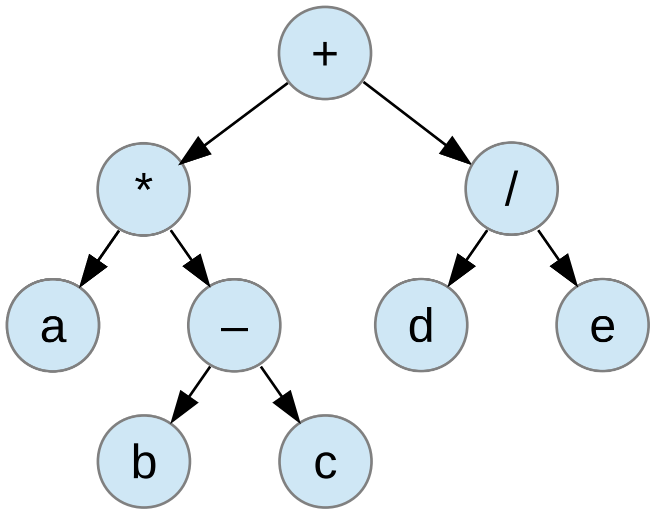


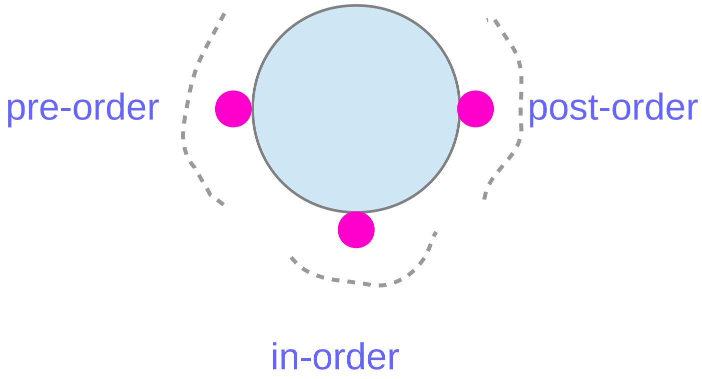


In-Order, Pre-Order, Post-Order Binary Tree Traversals

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

Breadth First Search



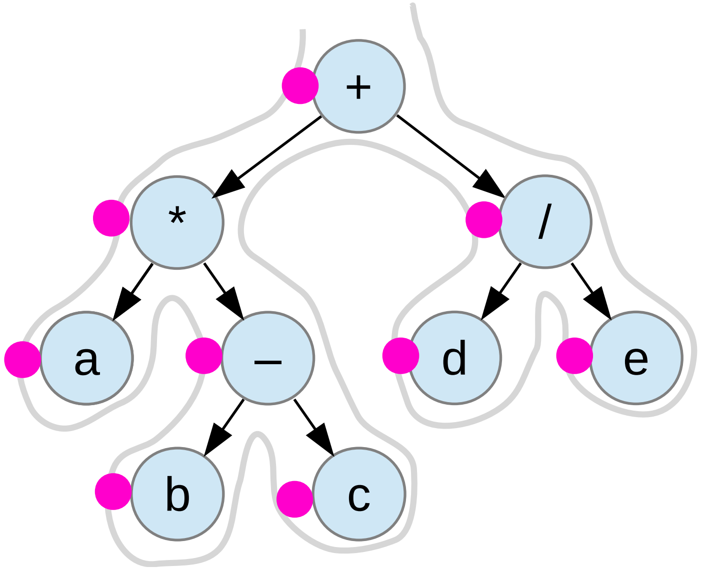


(a*(b-c))+(d/e)

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/Morphism

Pre-Order Binary Tree Traversals

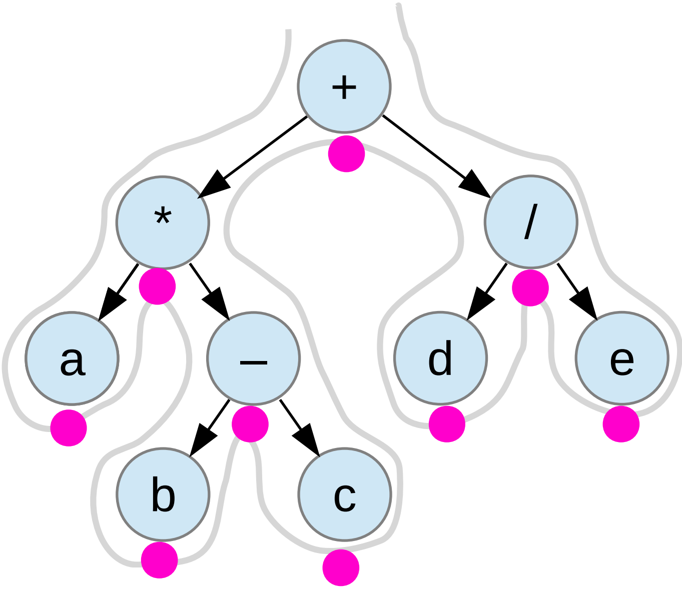


(a*(b-c))+(d/e)

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/Morphism

In-Order Binary Tree Traversals

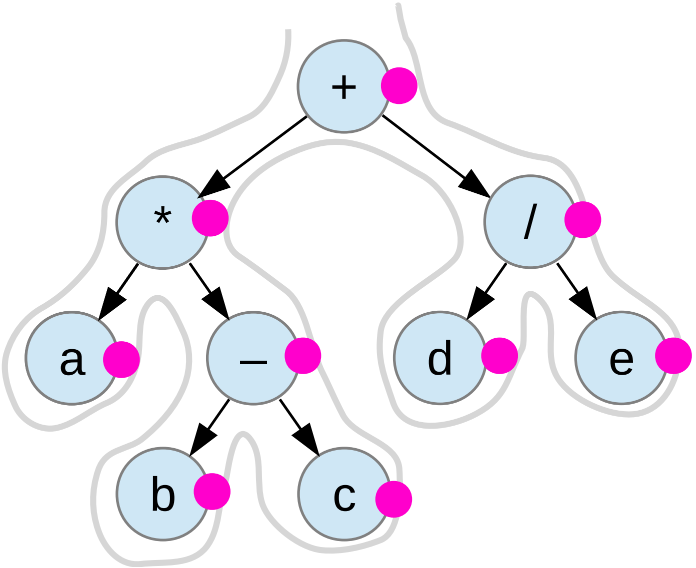


(a*(b-c))+(d/e)

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/Morphism

Post-Order Binary Tree Traversals



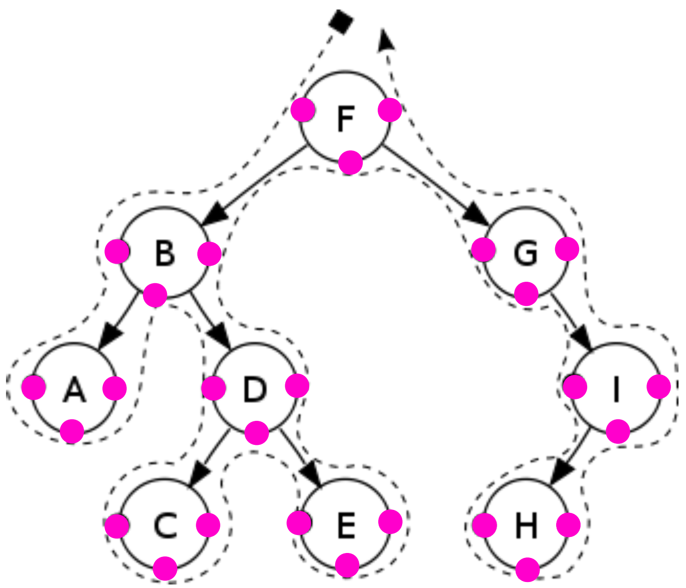
(a*(b-c))+(d/e)

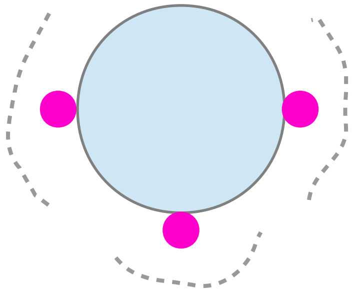
a * b – c + d / e	Infix notation
+ * a – b c / d e	Prefix notation
a b c – * d e / +	Postfix notation

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

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

Breadth First Search



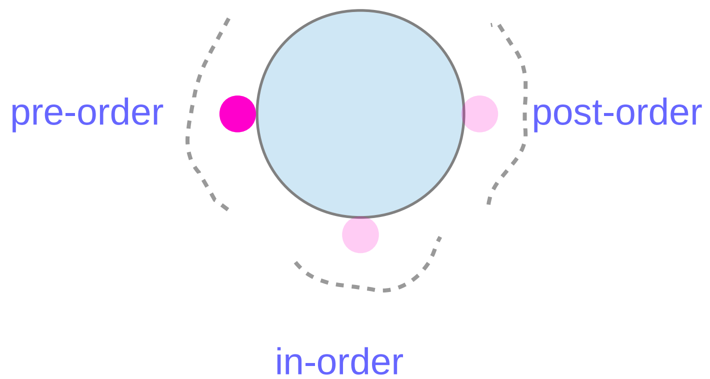


https://en.wikipedia.org/wiki/Morphism

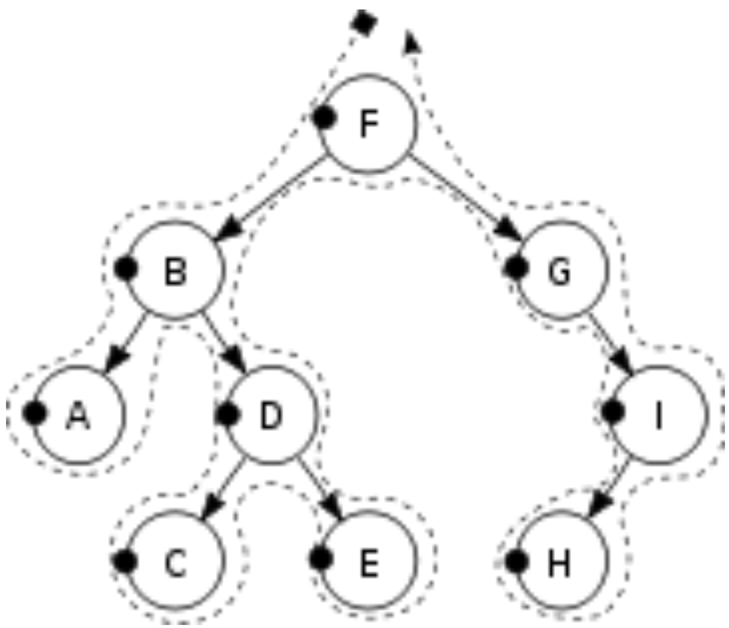
Pre-Order

pre-order function
Check if the current node is empty / null.
<u>Display</u> 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



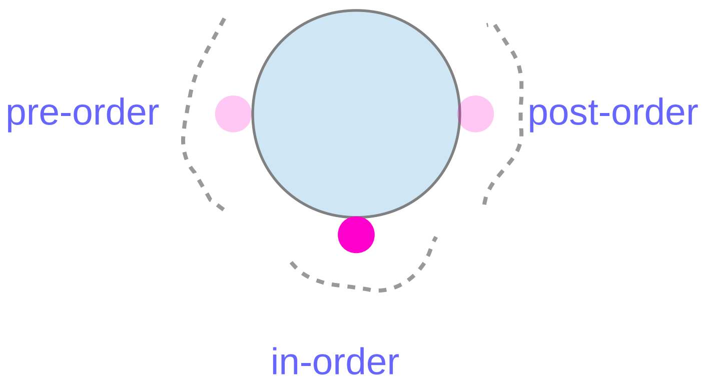
https://en.wikipedia.org/wiki/Morphism



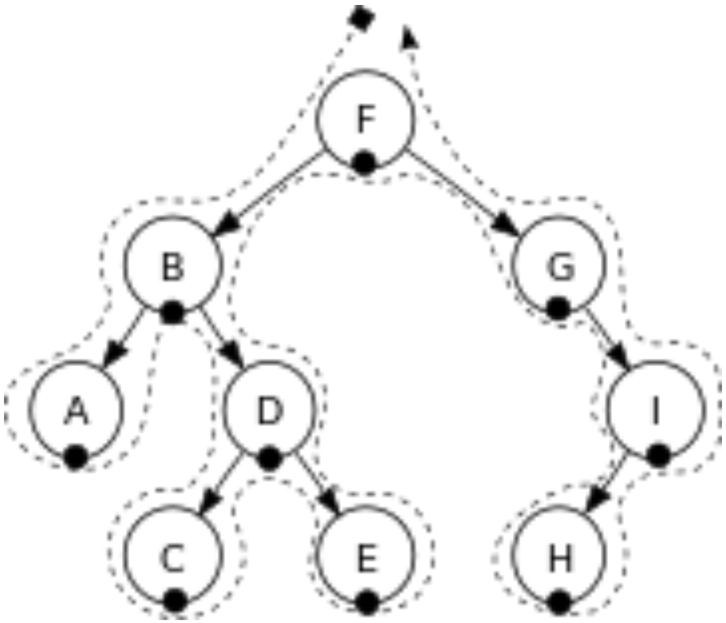
In-Order

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



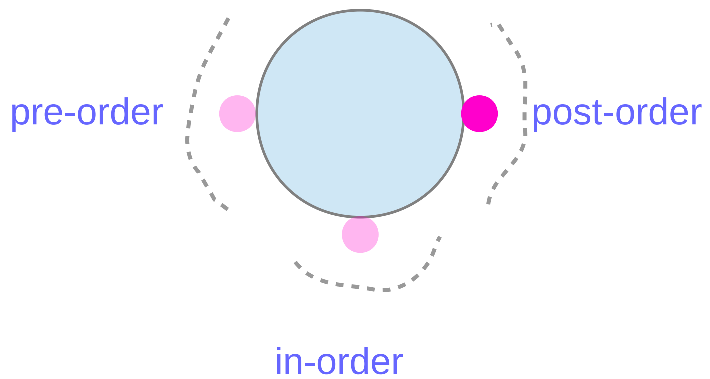
https://en.wikipedia.org/wiki/Morphism



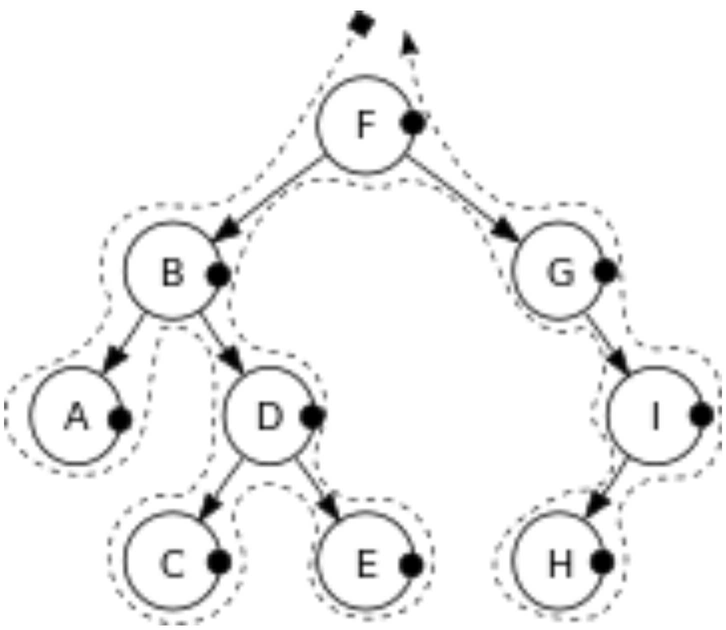
Post-Order

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

ACEDBHIGH



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

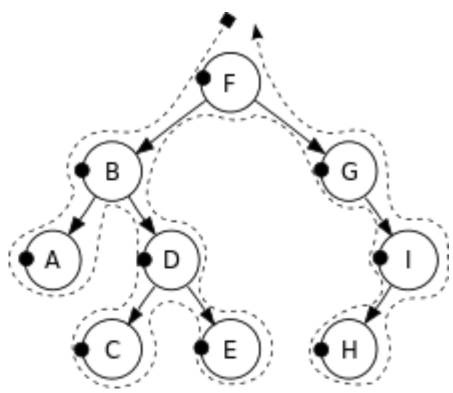


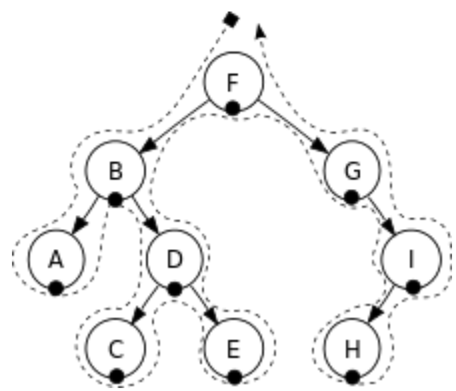
Recursive Algorithms

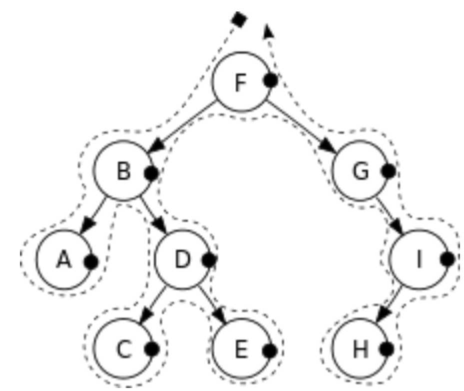
```
preorder(node)
  if (node = null)
    return
  visit(node)
  preorder(node.left)
  preorder(node.right)
```

```
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)
```



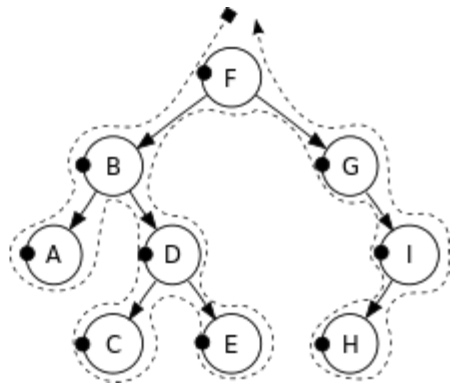


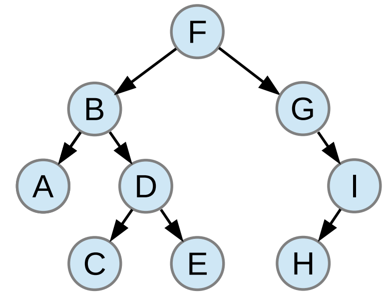


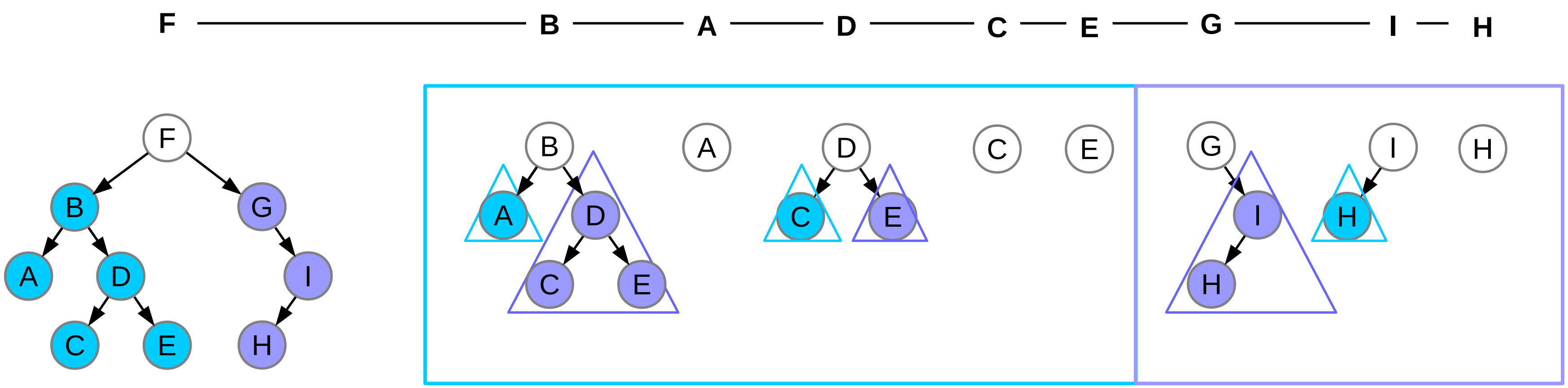
https://en.wikipedia.org/wiki/Tree_traversal

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

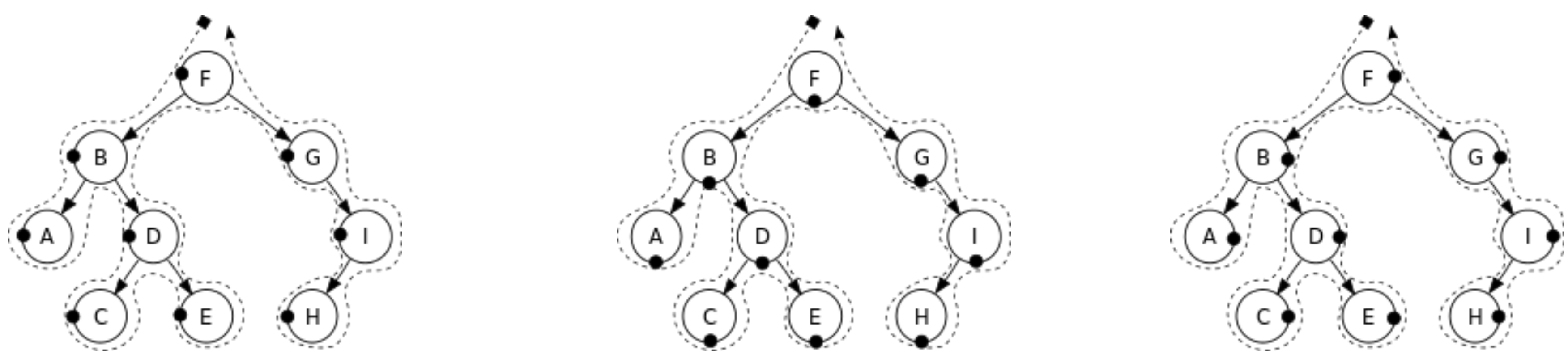
Iterative Algorithms

```
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)
```

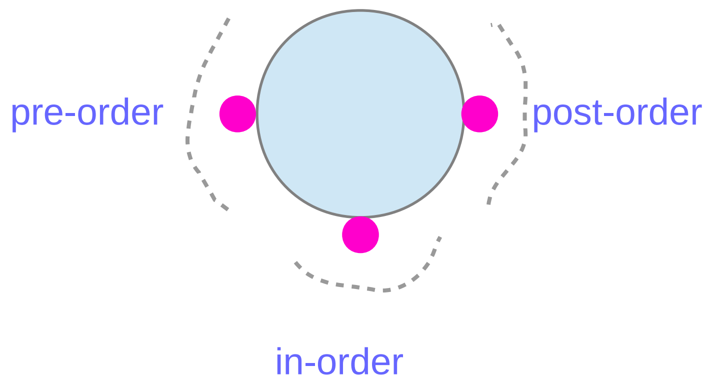
https://en.wikipedia.org/wiki/Tree_traversal

```
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()
```

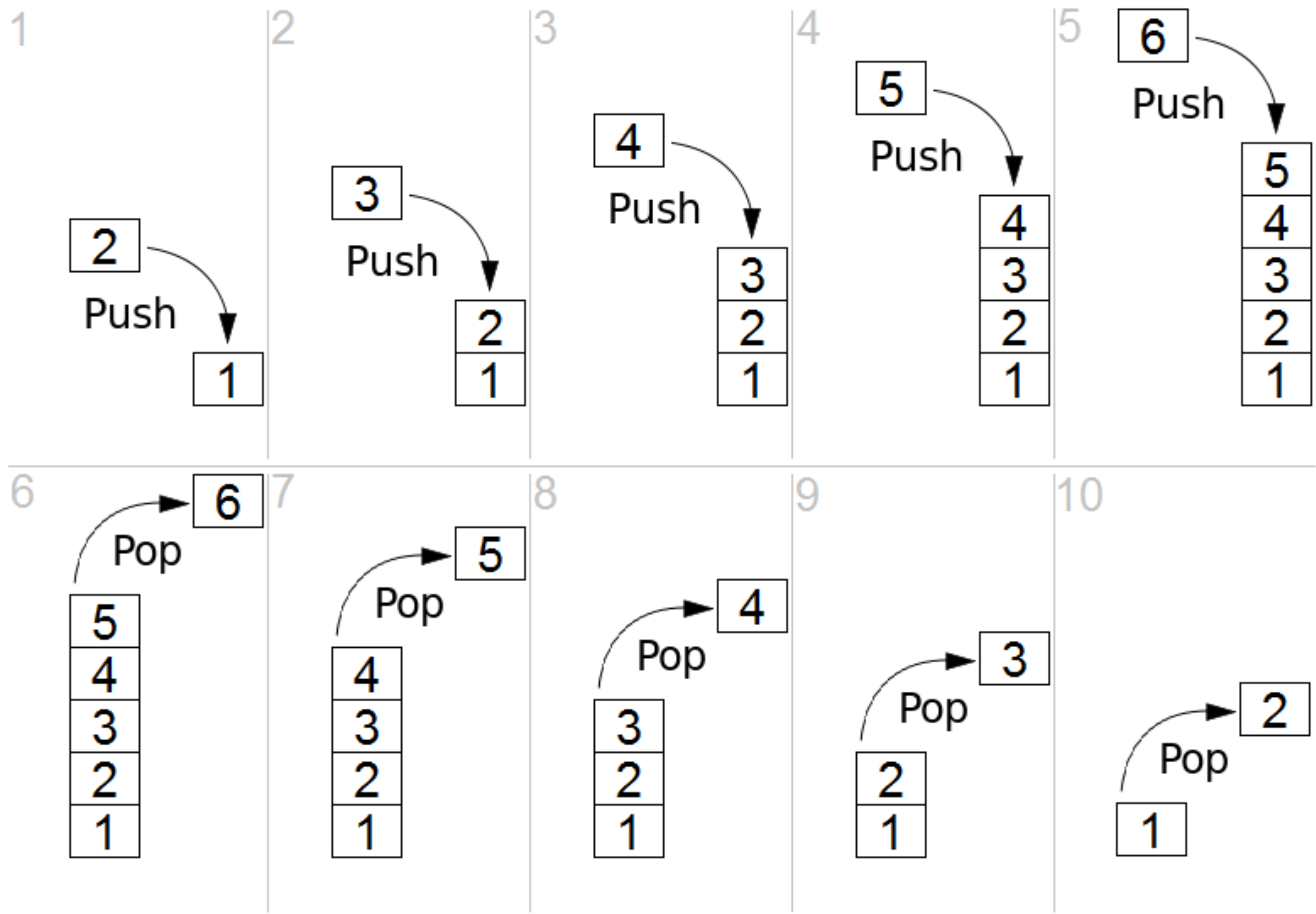


Tree Traversal



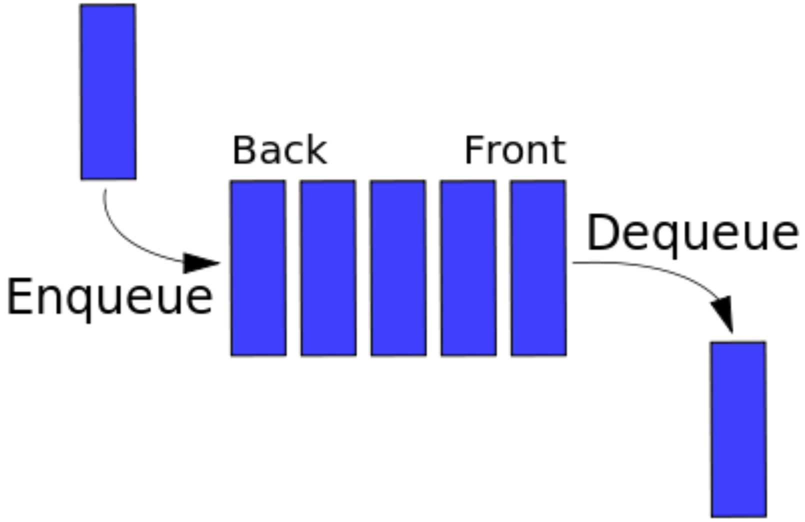
https://en.wikipedia.org/wiki/Morphism

Stack



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

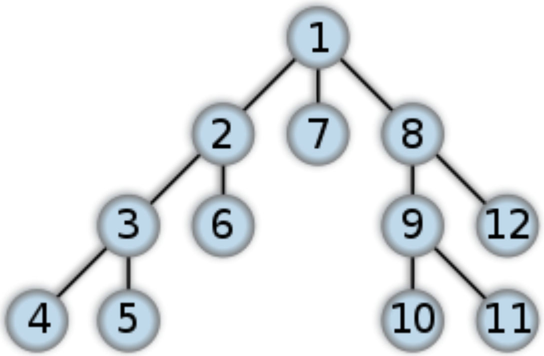
Queue



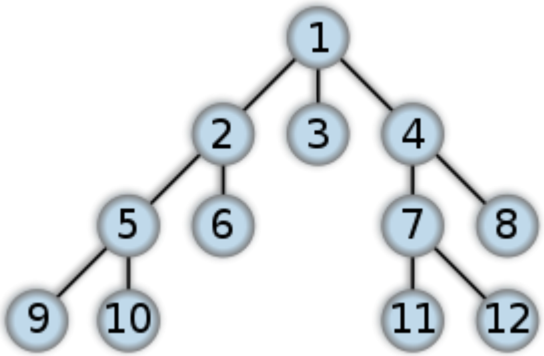
https://en.wikipedia.org/wiki/Queue_(abstract_data_type)#/media/File:Data_Queue.sv
g

Search Algorithms

DFS (Depth First Search)



BFS (Breadth First Search)



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

DFS Algorithm

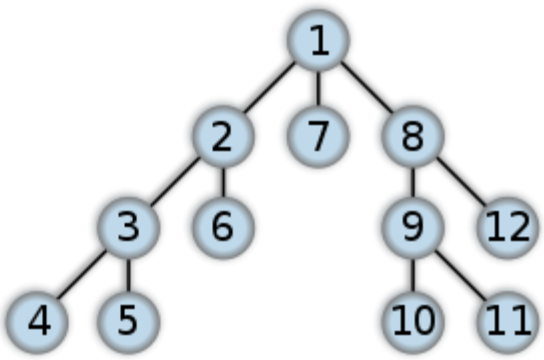
A recursive implementation of DFS:

```
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)
```

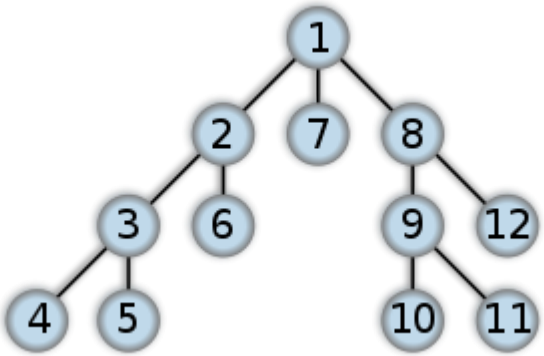
DFS (Depth First Search)



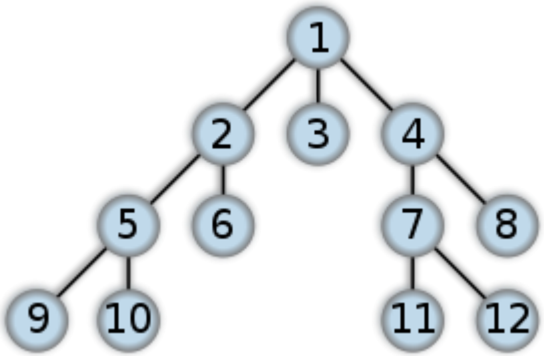
https://en.wikipedia.org/wiki/Breadth-first_search, /Depth-first_search

Search Algorithms

DFS (Depth First Search)



BFS (Breadth First Search)

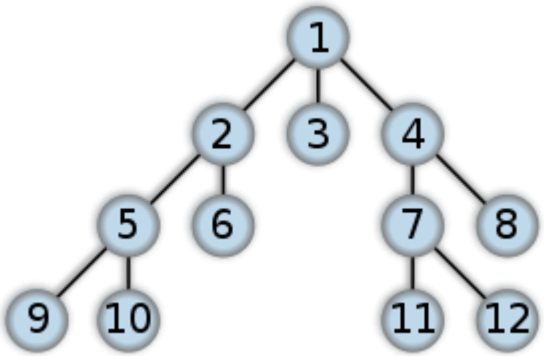


https://en.wikipedia.org/wiki/Breadth-first_search, /Depth-first_search

BFS Algorithm

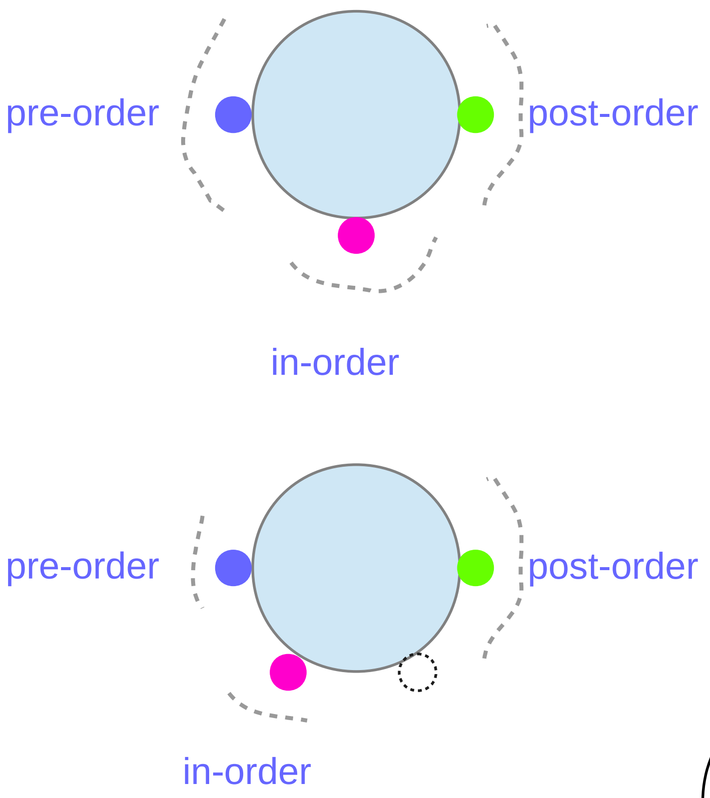
```
Breadth-First-Search(Graph, root):

    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)
```

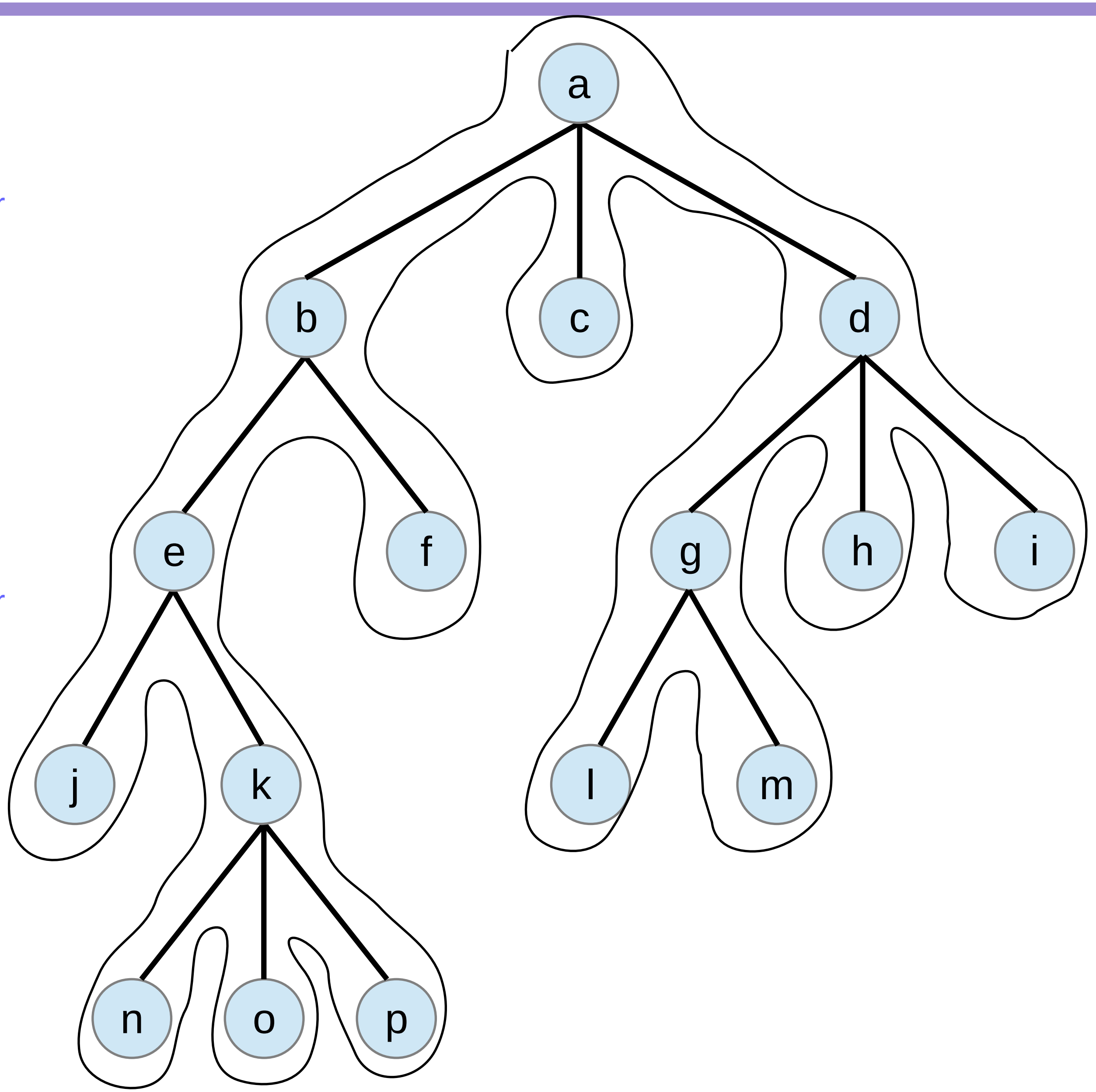
BFS (Breadth First Search)



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

In-Order

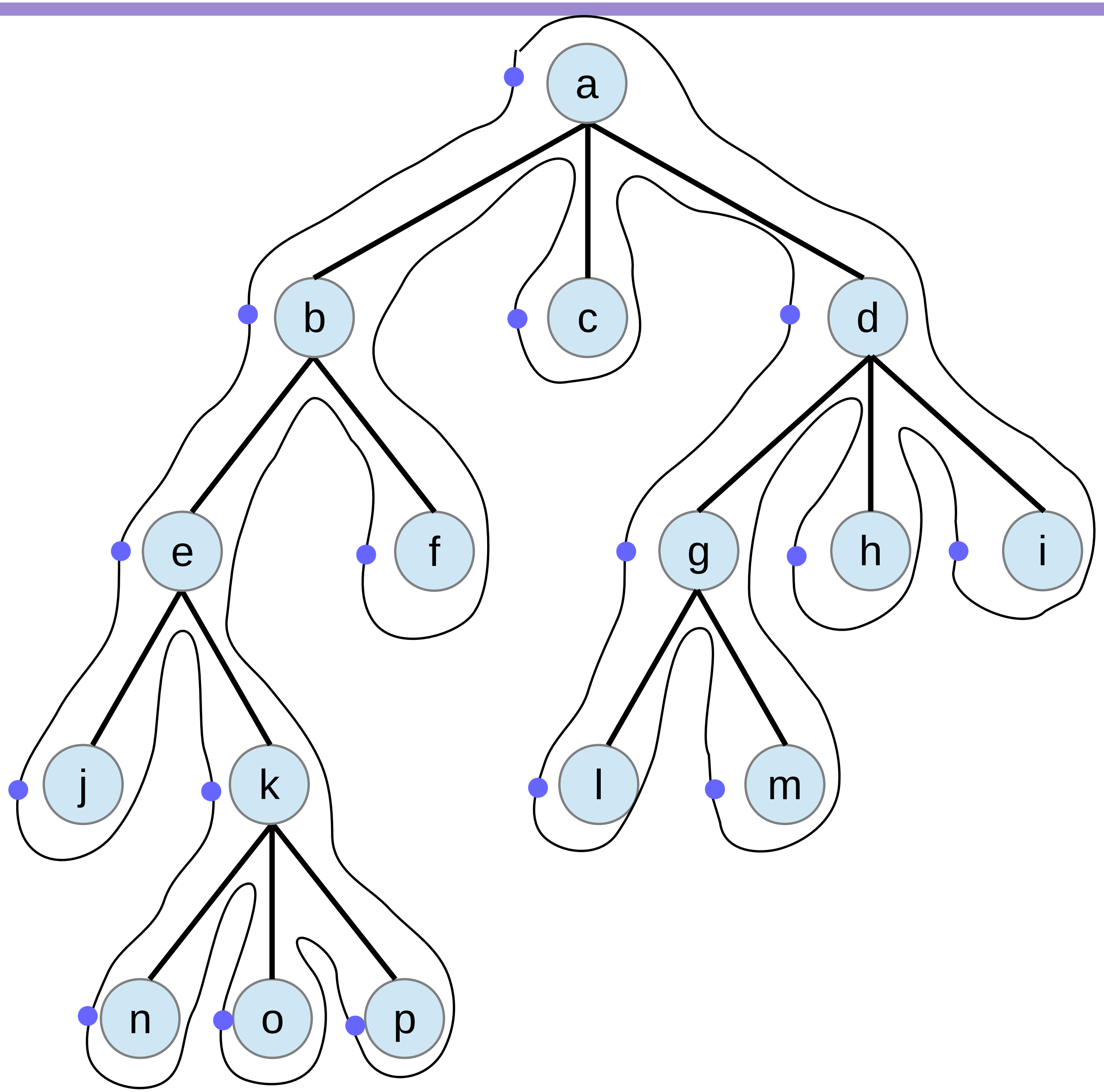




Rosen

Ternary Tree

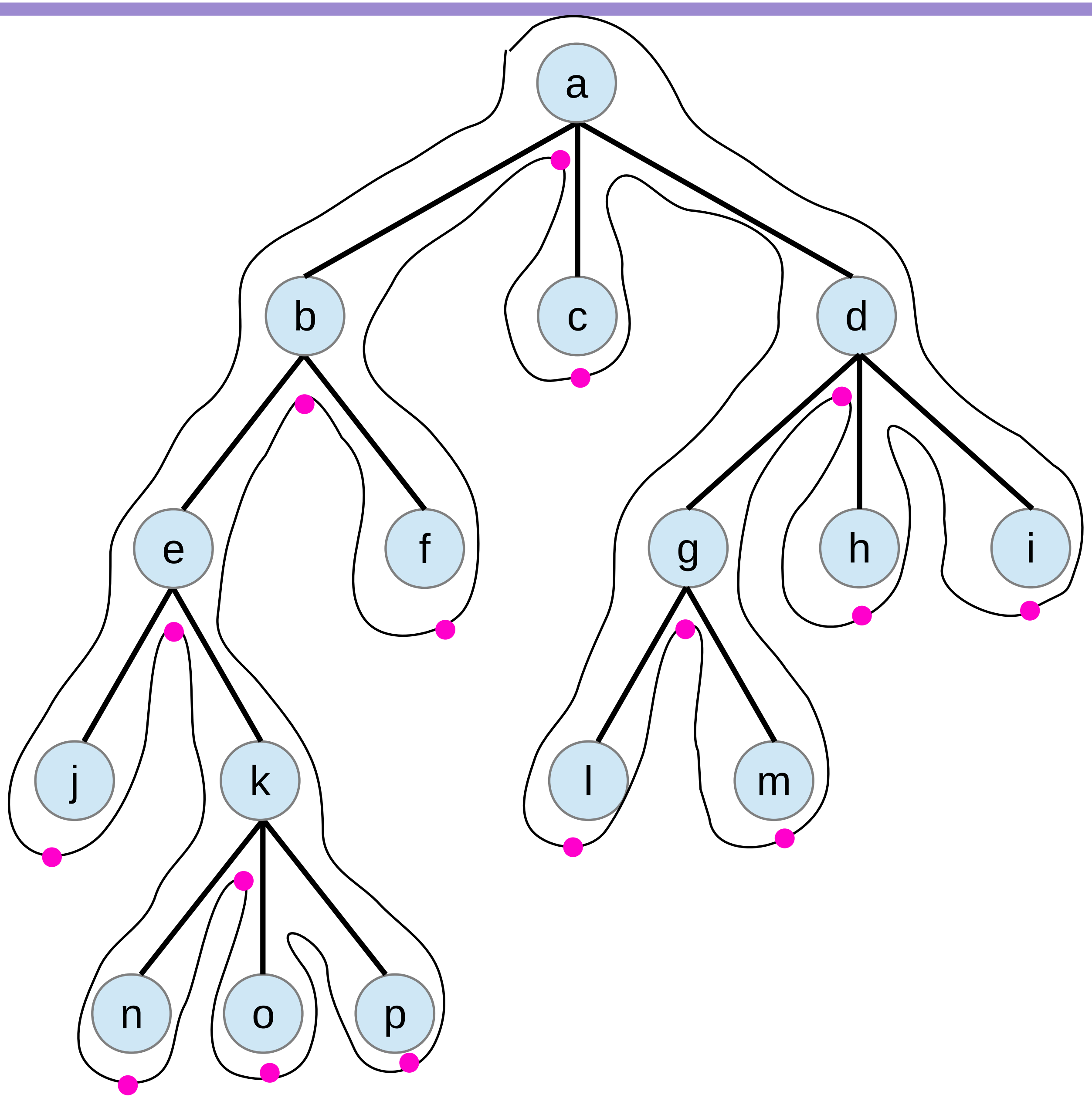
a-b-e-j-k-n-o-p-f-c-d-g-l-m-h-i



Rosen

In-Order

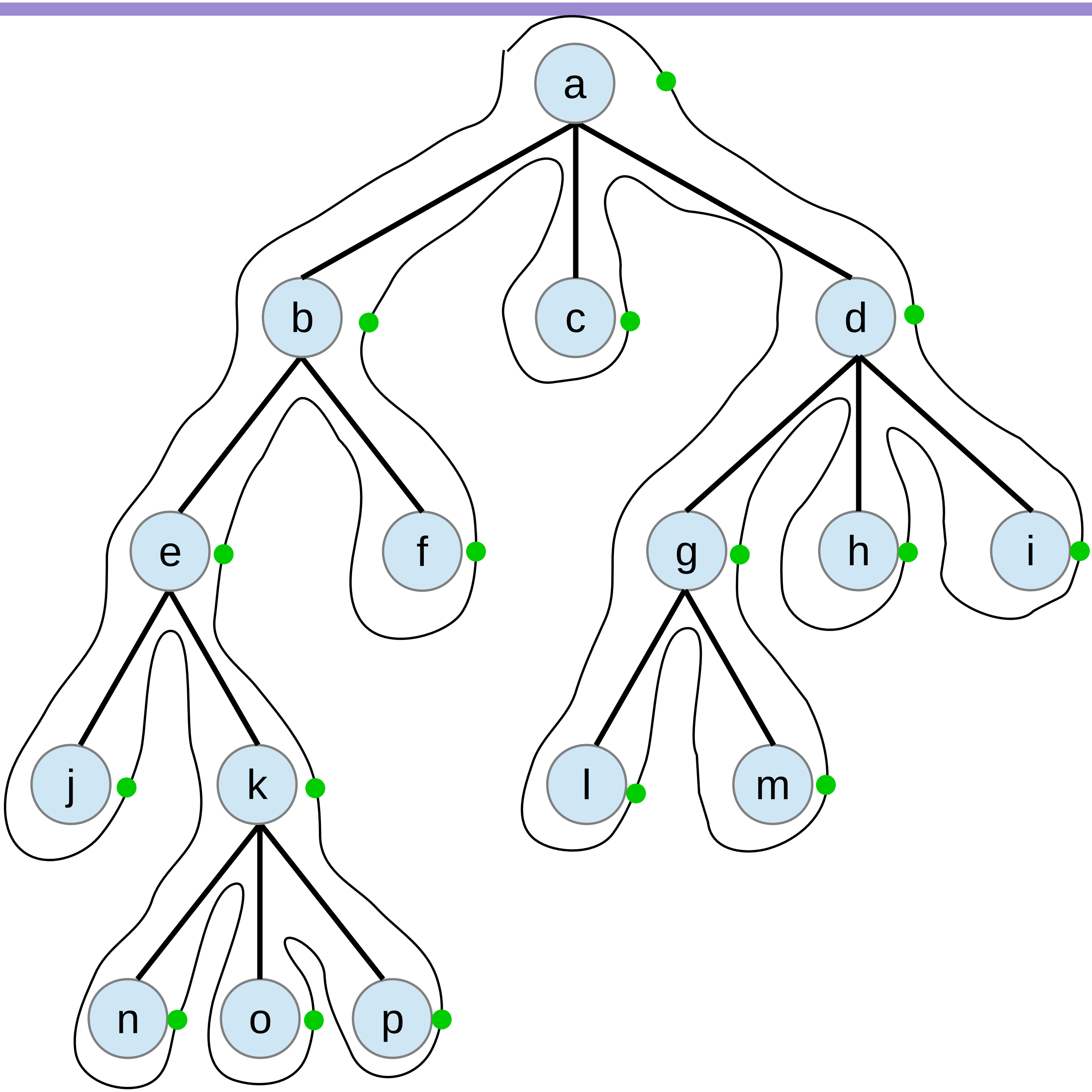
j-e-n-k-o-p-b-f-a-c-l-g-m-d-h-i



Rosen

Post-Order

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



Rosen

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

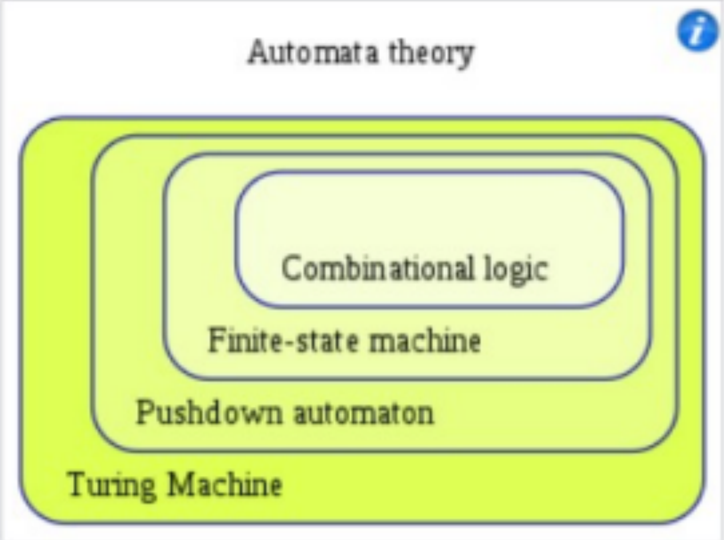
[1] http://en.wikipedia.org/
[2]

Formal Language (1A)

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

This document was produced by using LibreOffice and Octave.

Class of Automata



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

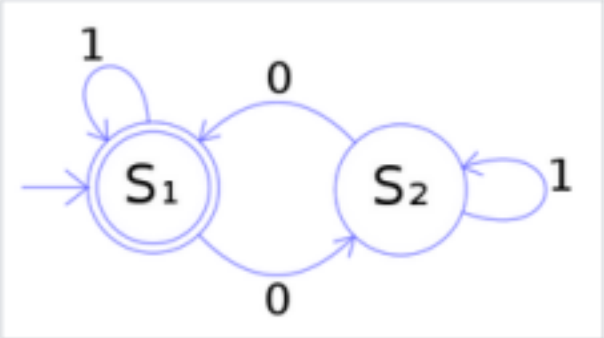
https://en.wikipedia.org/wiki/Automata_theory

Finite State Machine

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).

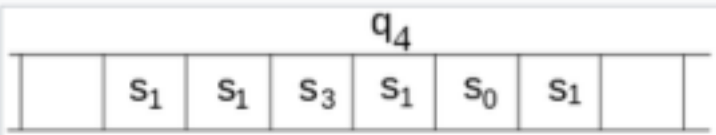
As the automaton sees a **symbol** of **input**, it makes a **transition** (or jump) to another **state**, according to its **transition function**, which takes the **current state** and the recent **symbol** as its **inputs**.



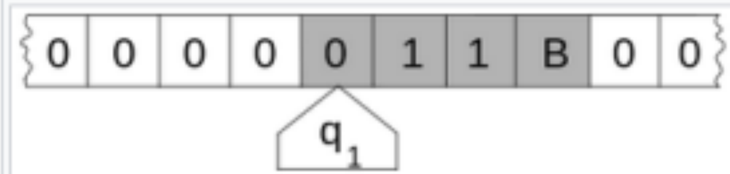
The study of the mathematical properties of such automata is automata theory. The picture is a visualization of an automaton that recognizes strings containing an even number of *0*s. The automaton starts in state *S1*, and transitions to the non-accepting state *S2* upon reading the symbol *0*. Reading 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

Turing Machine



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.)

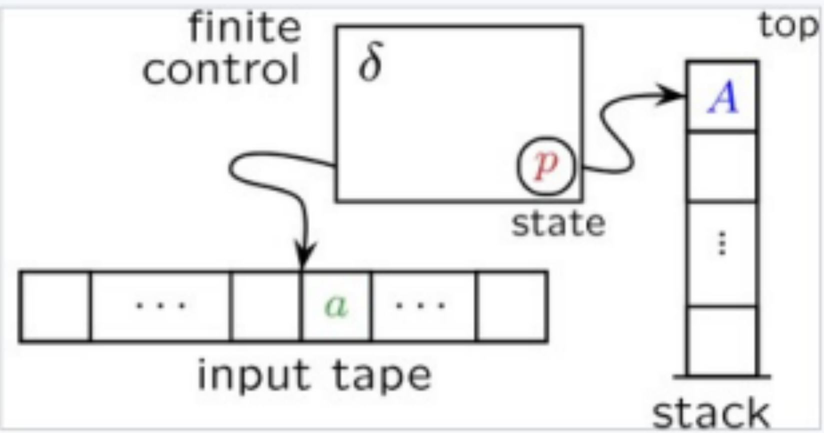


Here, the internal state (q_1) is shown inside the head, and the illustration describes the tape as being infinite and pre-filled with "0", the symbol serving as blank. The system's full state (its *complete configuration*) consists of the internal state, any non-blank symbols on the tape (in this illustration "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

Pushdown Automaton

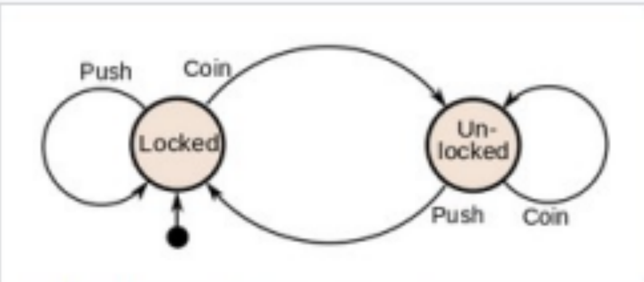
a pushdown automaton (PDA) is
a type of automaton that employs a stack



A diagram of a pushdown automaton

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

Finite State Machine



State diagram for a turnstile

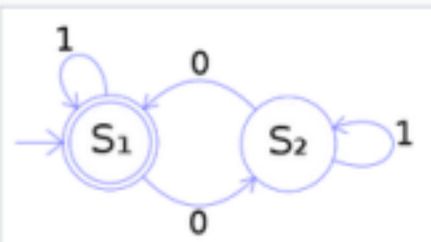


Fig. 5: Representation of a finite-state machine; this example shows one that determines whether a binary number has an even number of 0s, where S_1 is an **accepting state**.

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

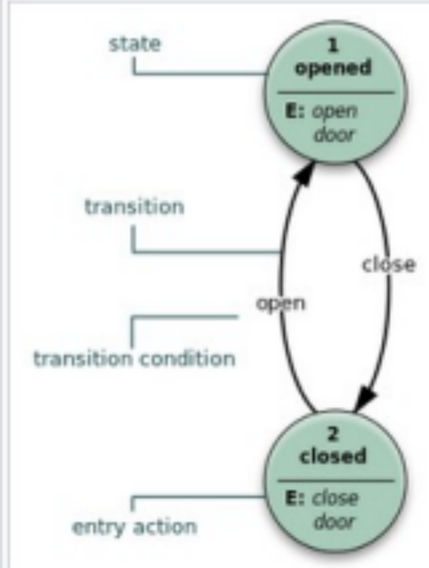


Fig. 3 Example of a simple finite state machine

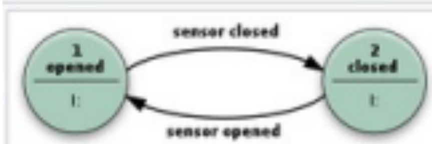


Fig. 7 Transducer FSM: Mealy model example

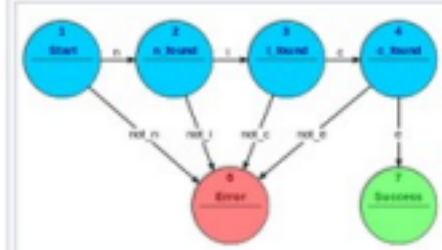


Fig. 4 Acceptor FSM: parsing the string "nice"

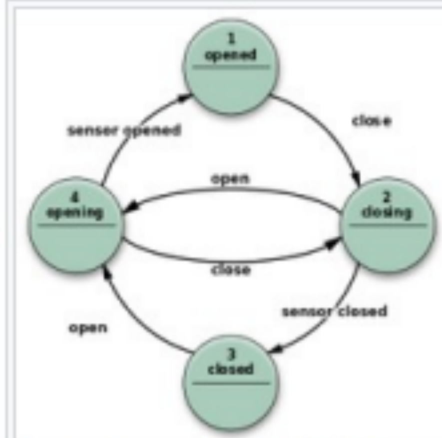


Fig. 6 Transducer FSM: Moore model example

References

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

Finite State Machine (3A)

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

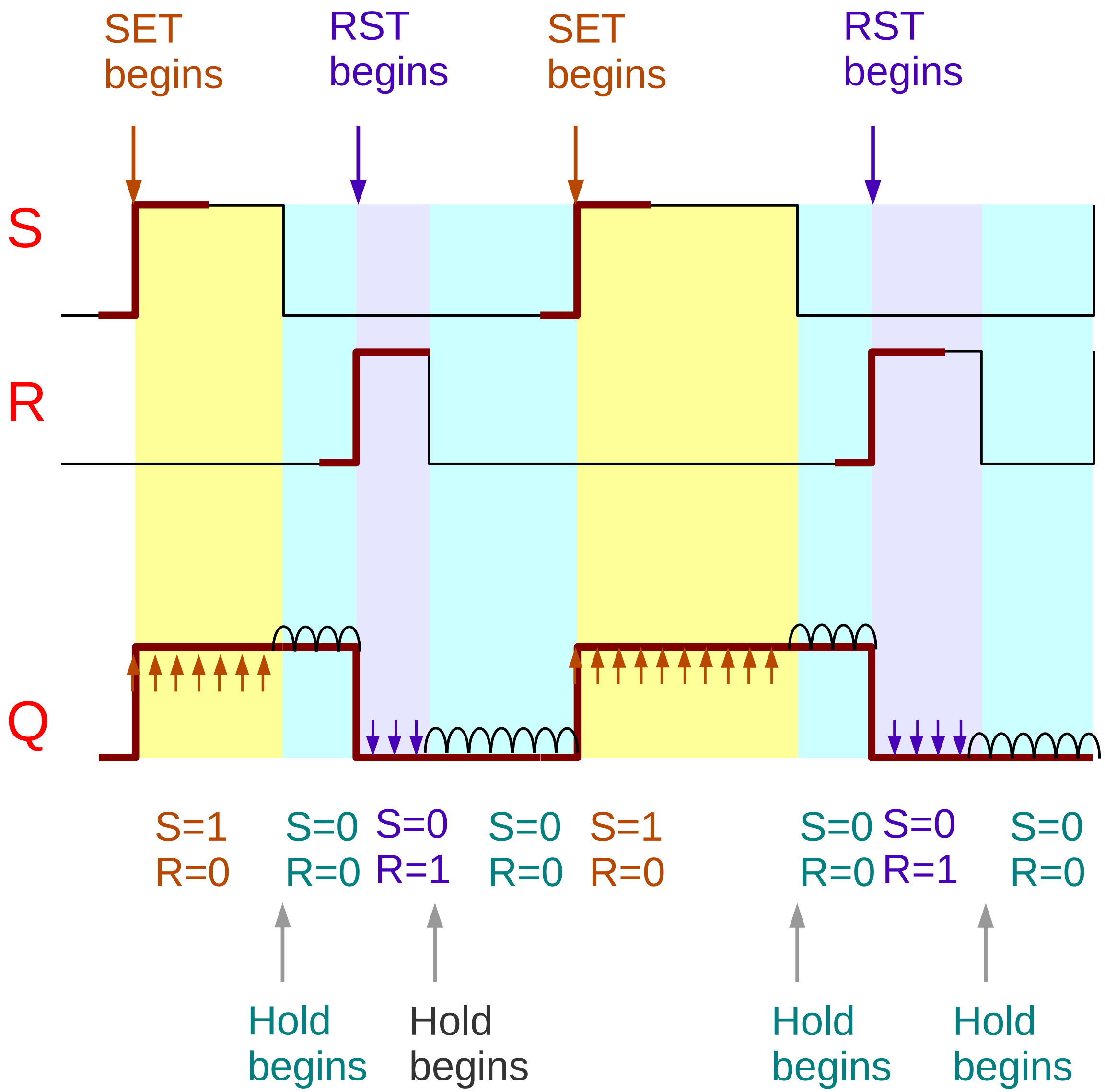
This document was produced by using LibreOffice and Octave.

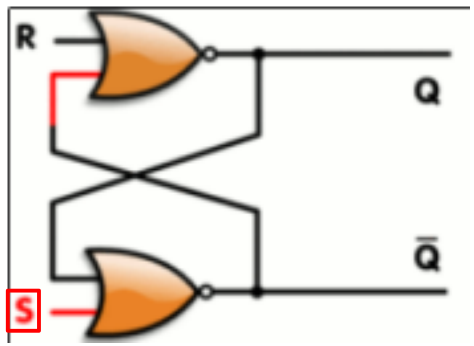
Formal Language

a

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

NOR-based SR Latch

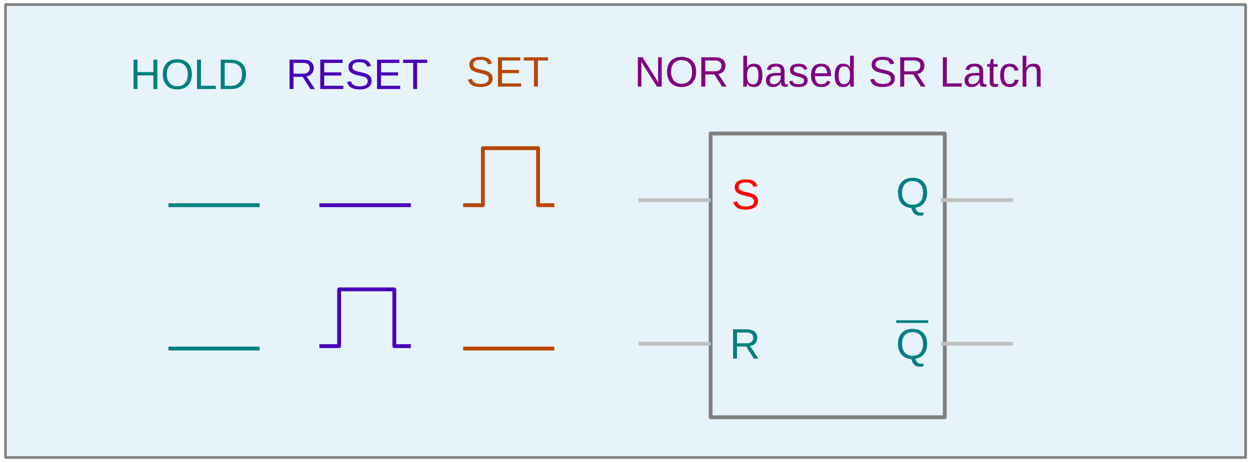


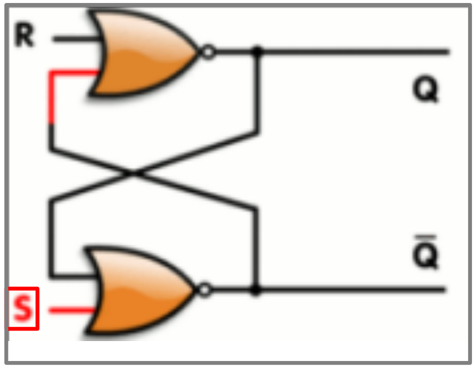


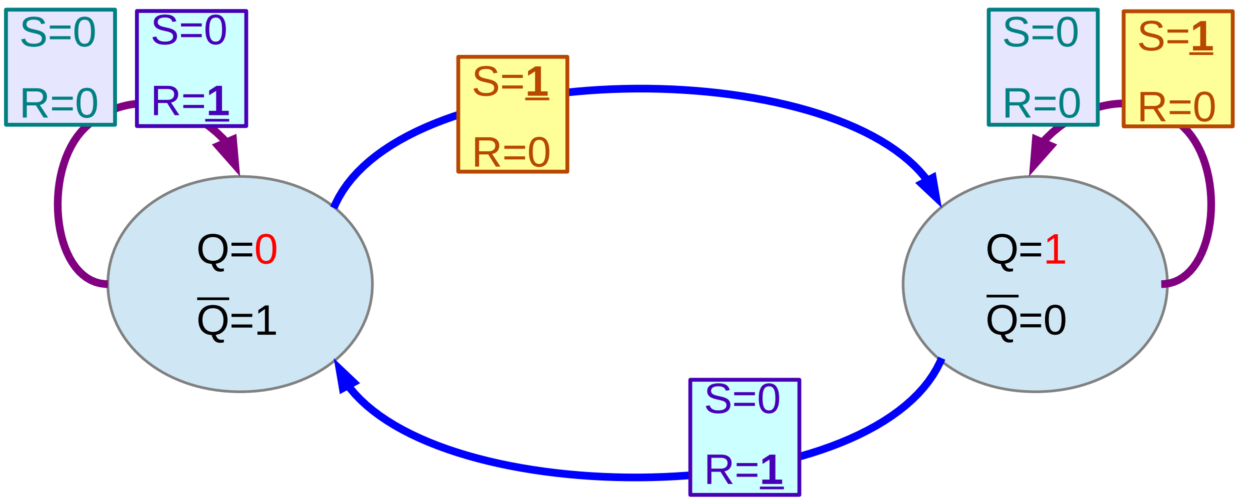
SET	S=**1** R=0	Q=1 $\overline{Q}$=0
RESET	S=0 R=**1**	Q=0 $\overline{Q}$=1
HOLD	S=0 R=0	Q=old Q $\overline{Q}$=old $\overline{Q}$

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

NOR-based SR Latch States



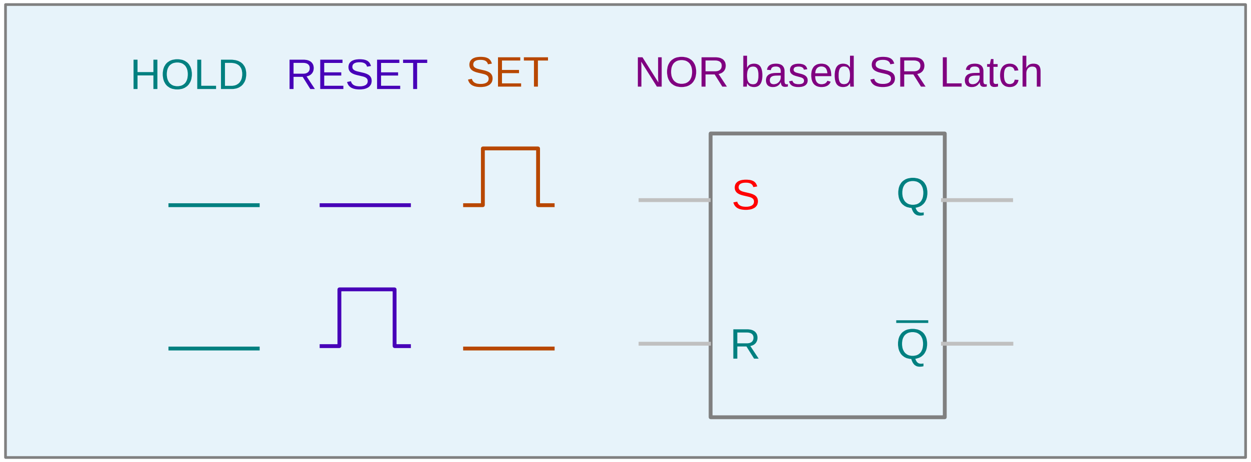


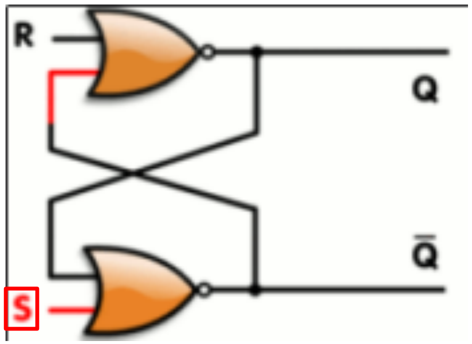


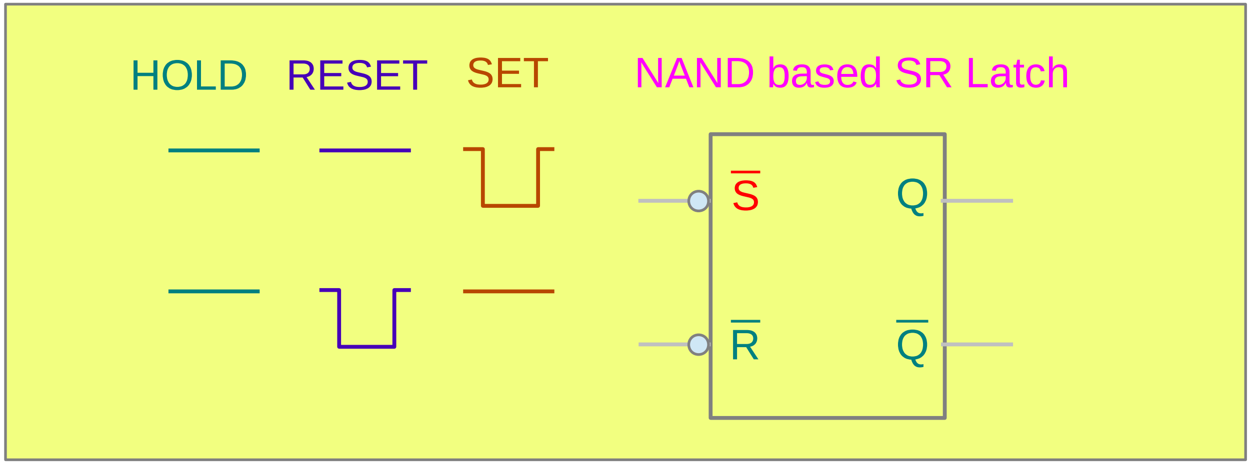
SET	S=1 R=0	Q=1 $\overline{Q}$=0
RESET	S=0 R=1	Q=0 $\overline{Q}$=1
HOLD	S=0 R=0	Q=old Q $\overline{Q}$=old $\overline{Q}$

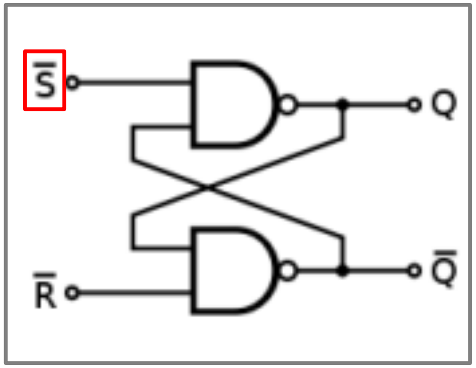
https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

SR Latch Symbols



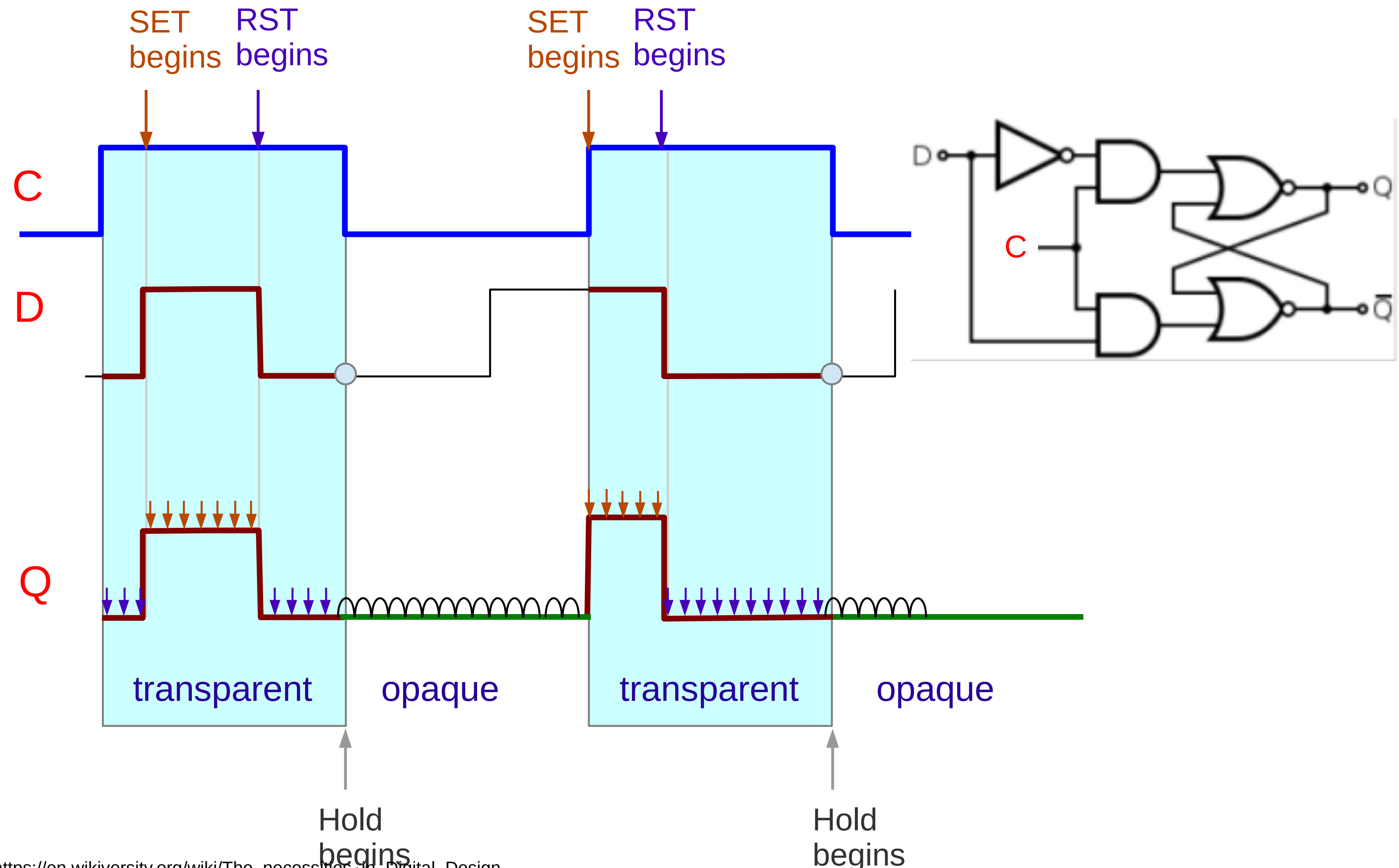






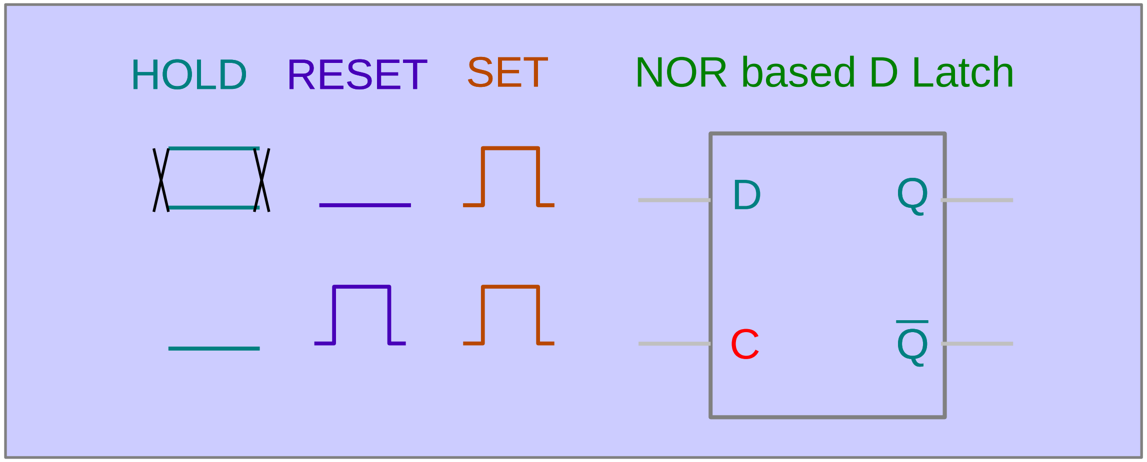
https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

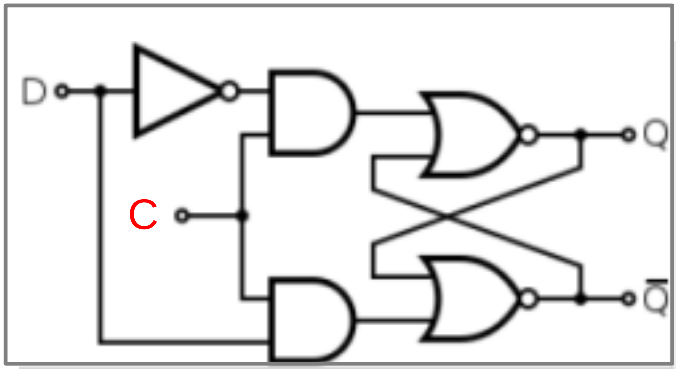
NOR-based D Latch

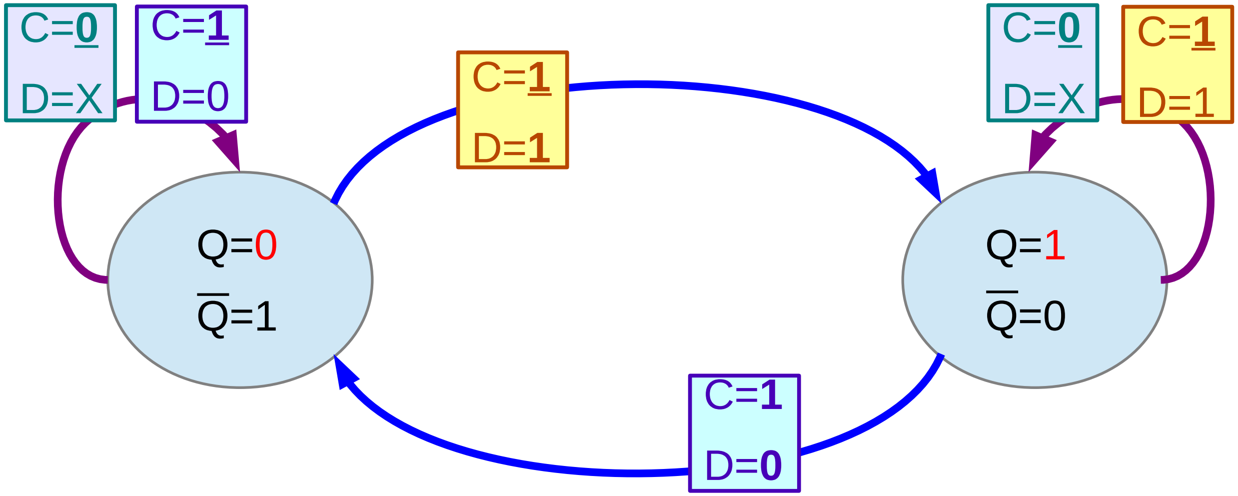


https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

NOR-based D Latch

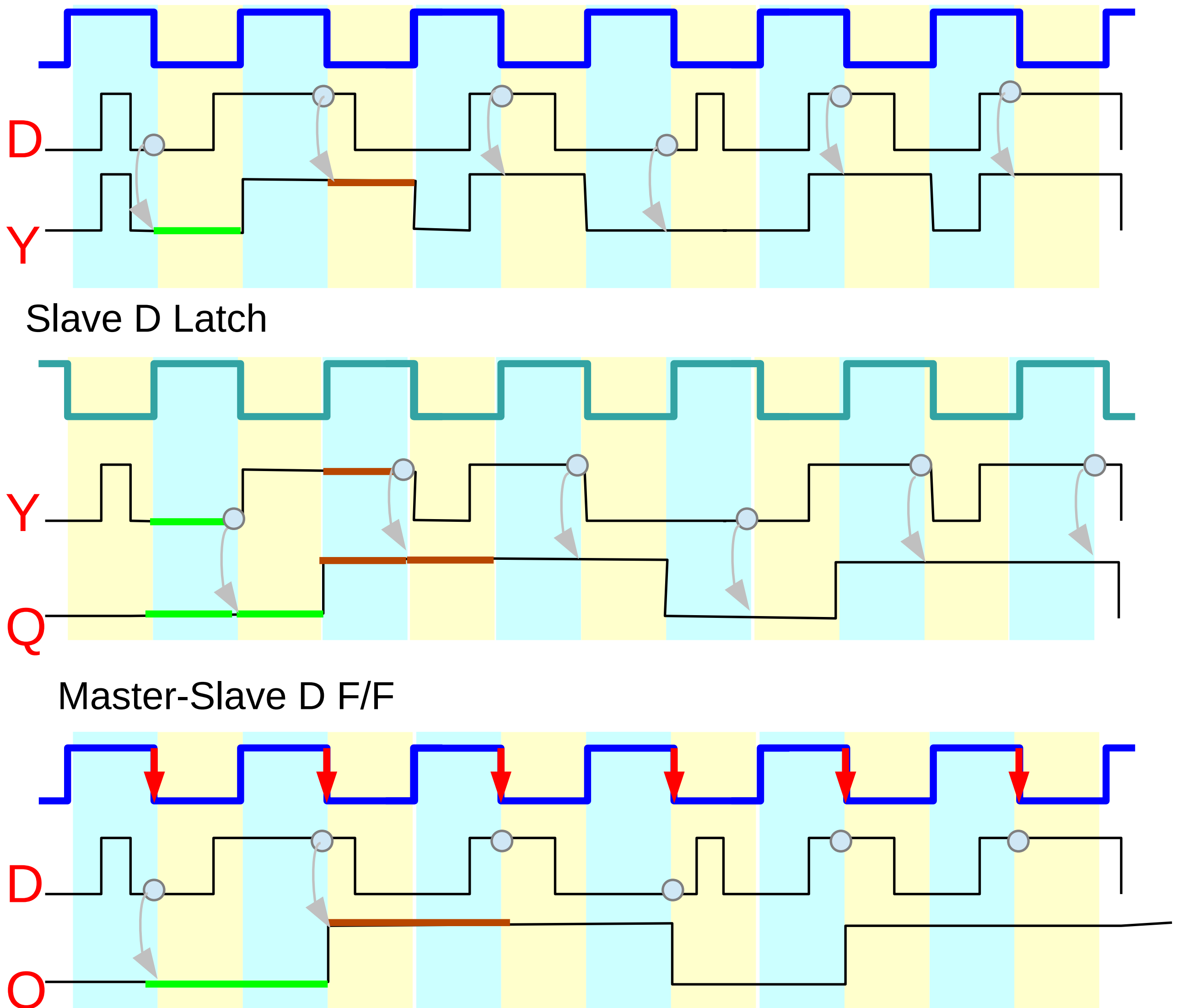




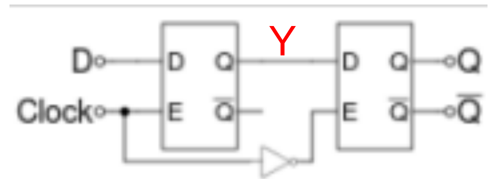


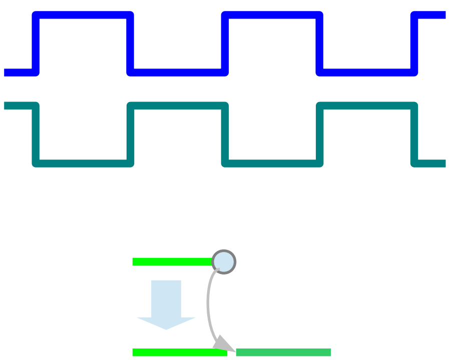
https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

Master-Slave D FlipFlop



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



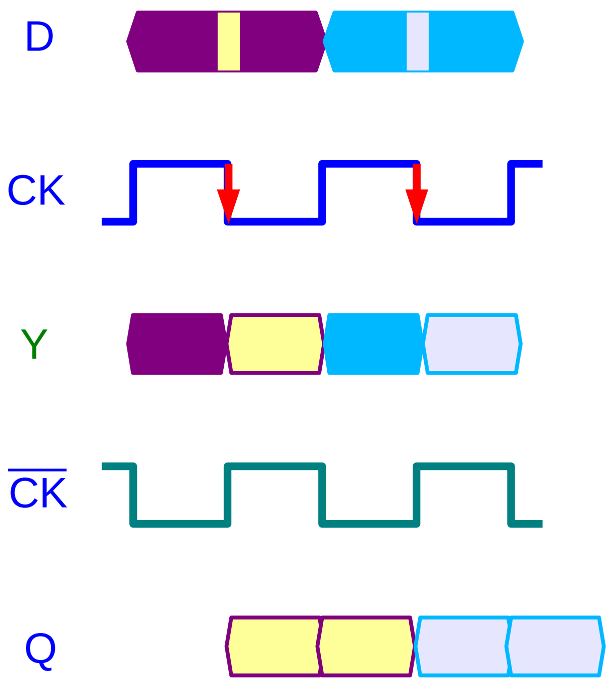


the hold output of the master is transparently reaches the output of the slave

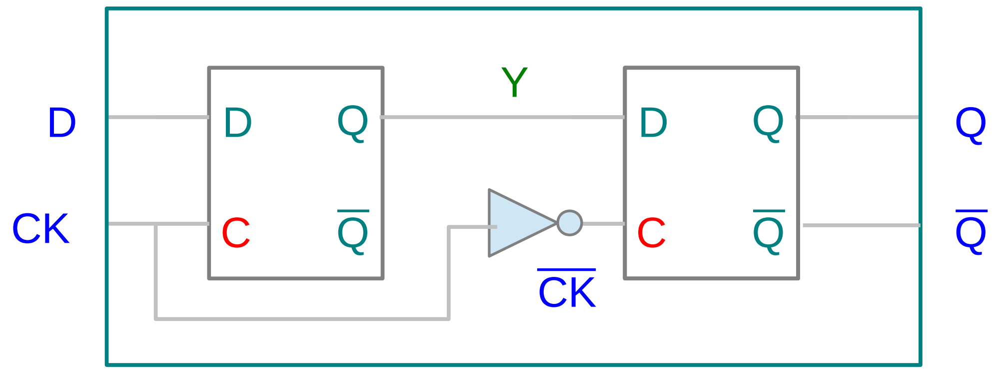
this value is held for another half period

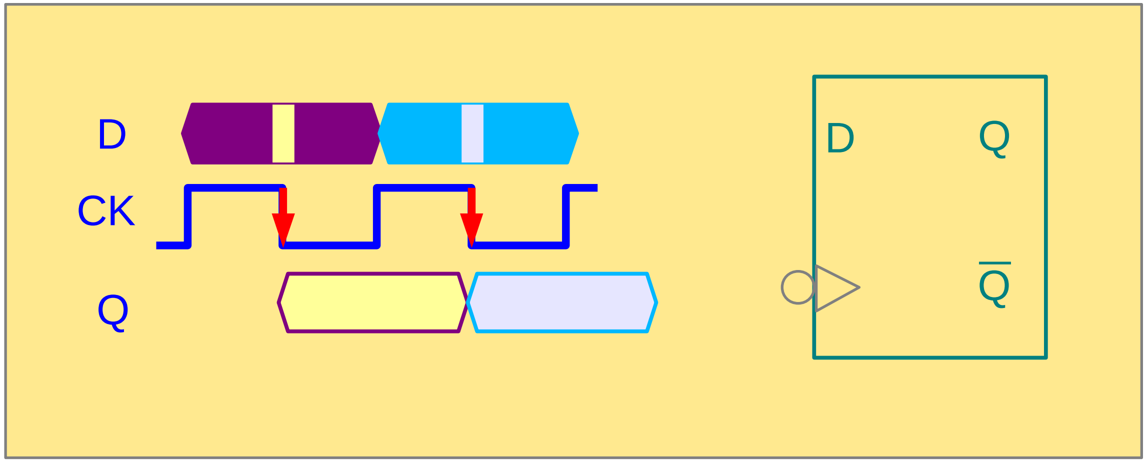
Master-Slave D FlipFlop – Falling Edge

Master D Latch



Slave D Latch

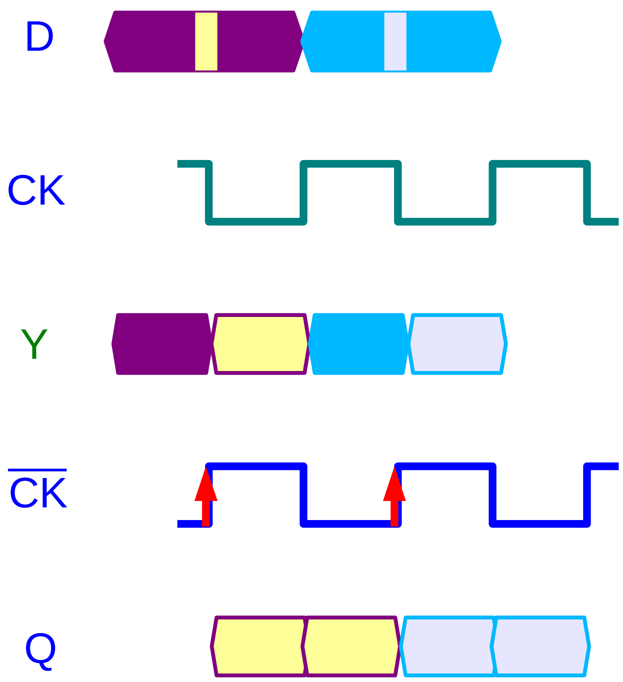




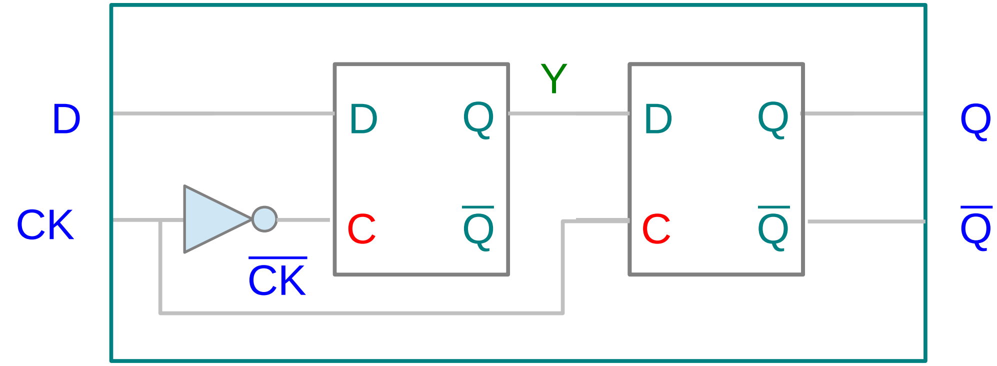
https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

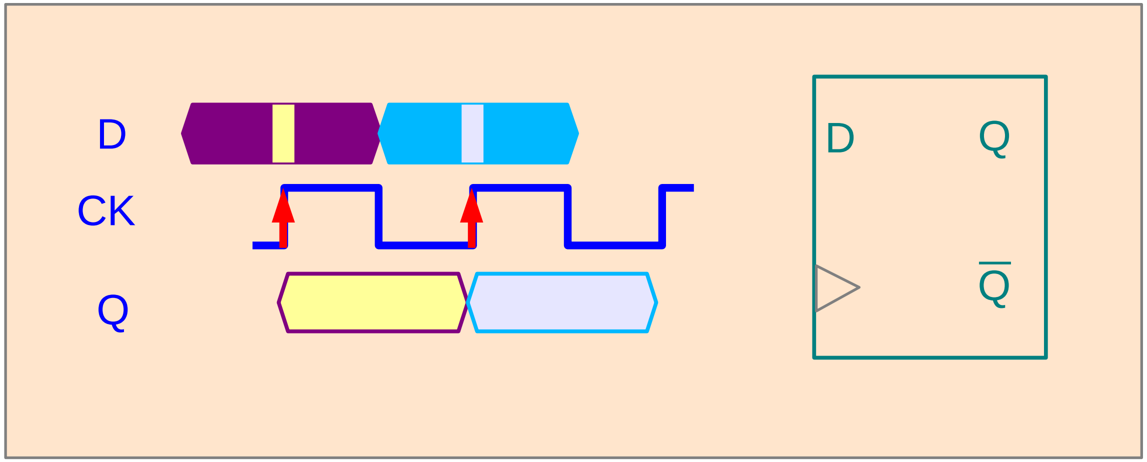
Master-Slave D FlipFlop – Rising Edge

Master D Latch



Slave D Latch



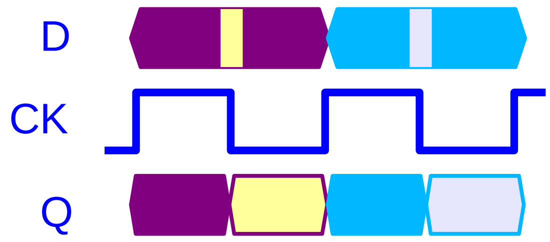


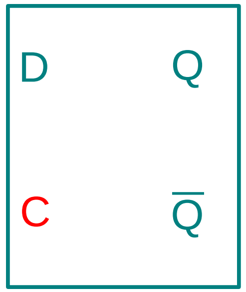
https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

D Latch & D FlipFlop

Level Sensitive D Latch

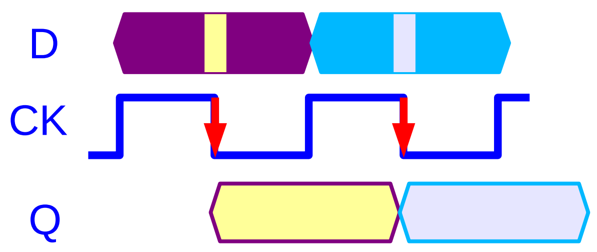
CK=1 transparent
CK=0 opaque

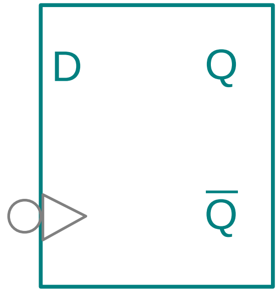




Edge Sensitive D FlipFlop

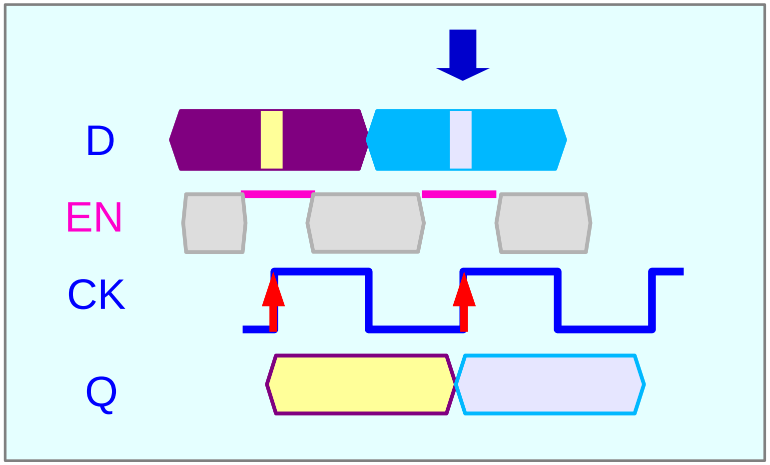
CK=1→0 transparent
else opaque

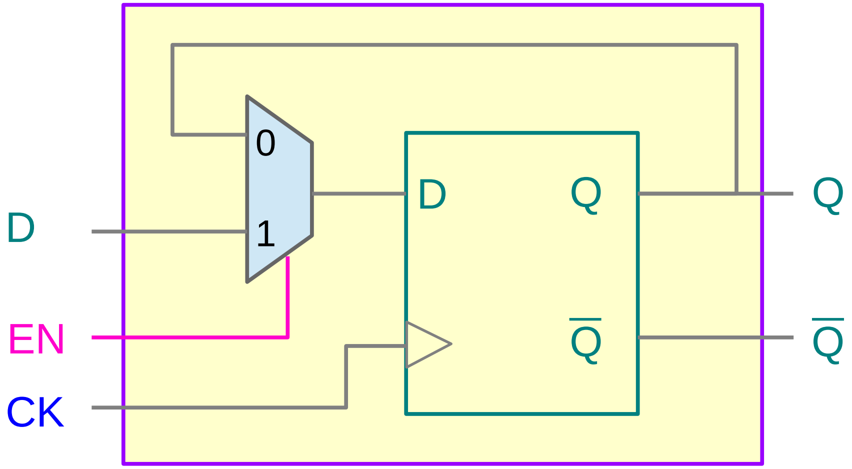


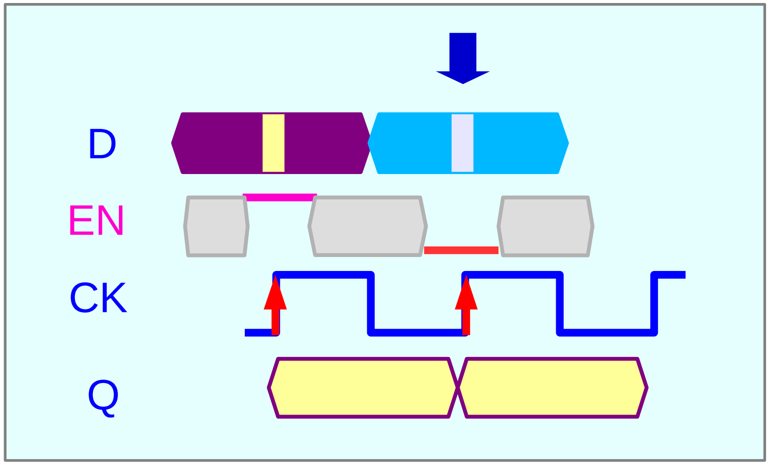


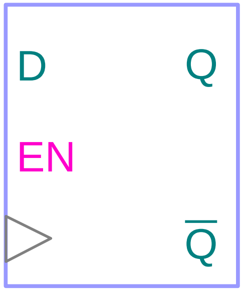
https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

D FlipFlop with Enable



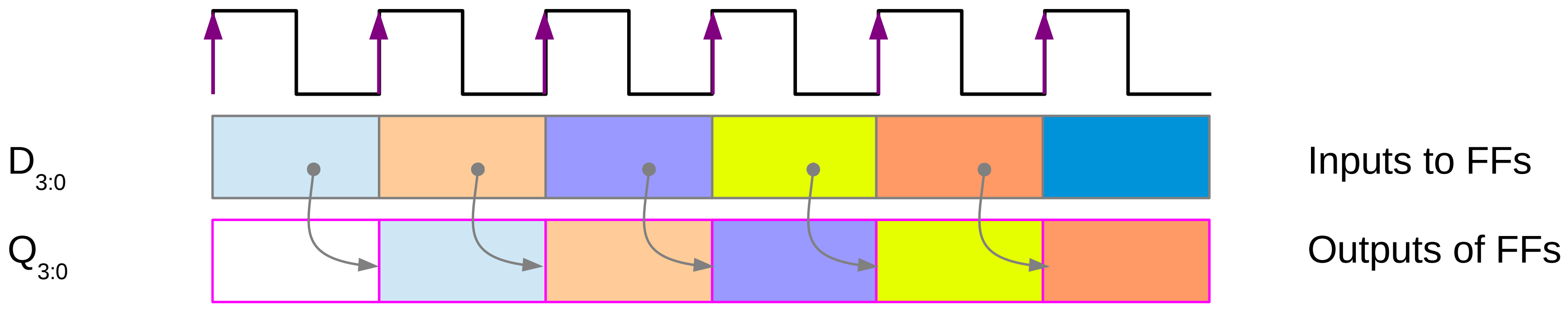


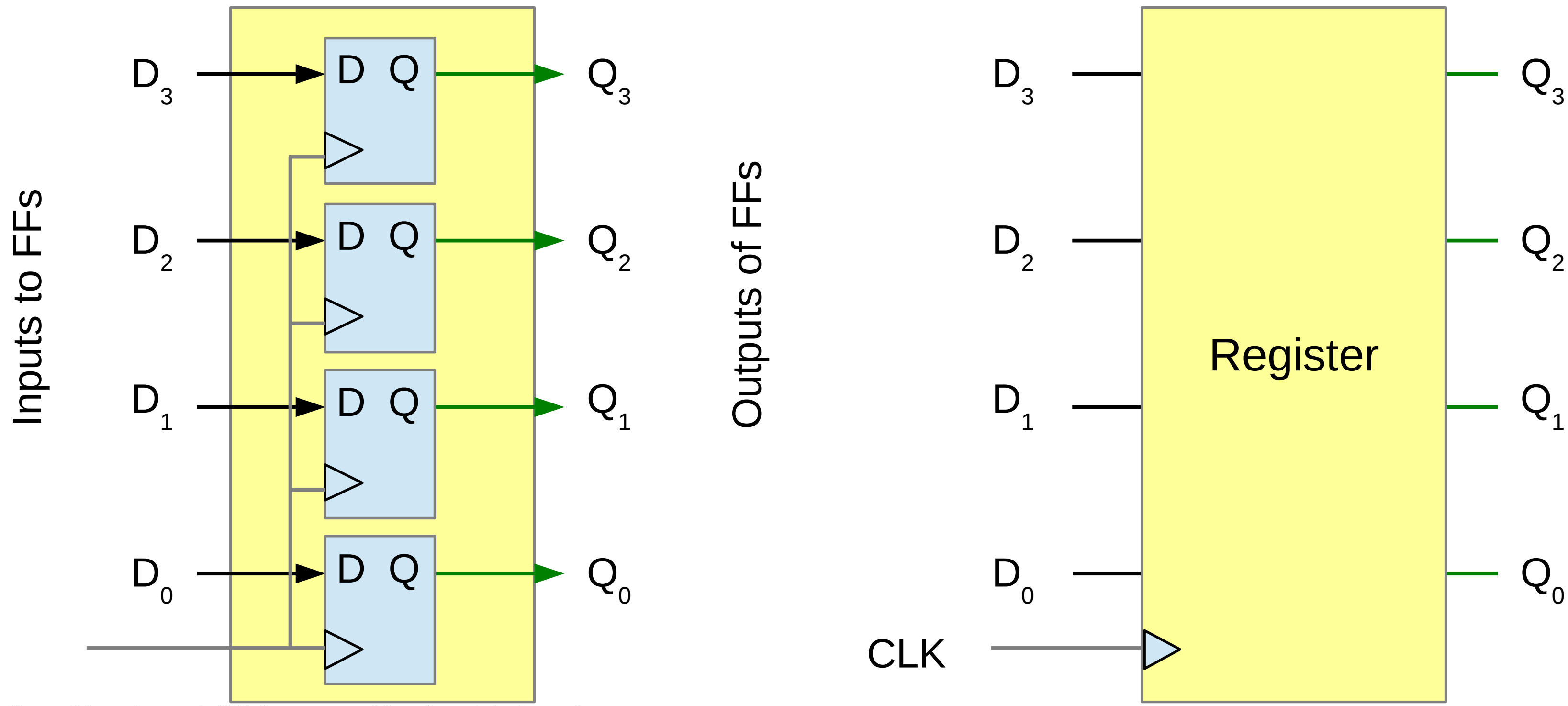




https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

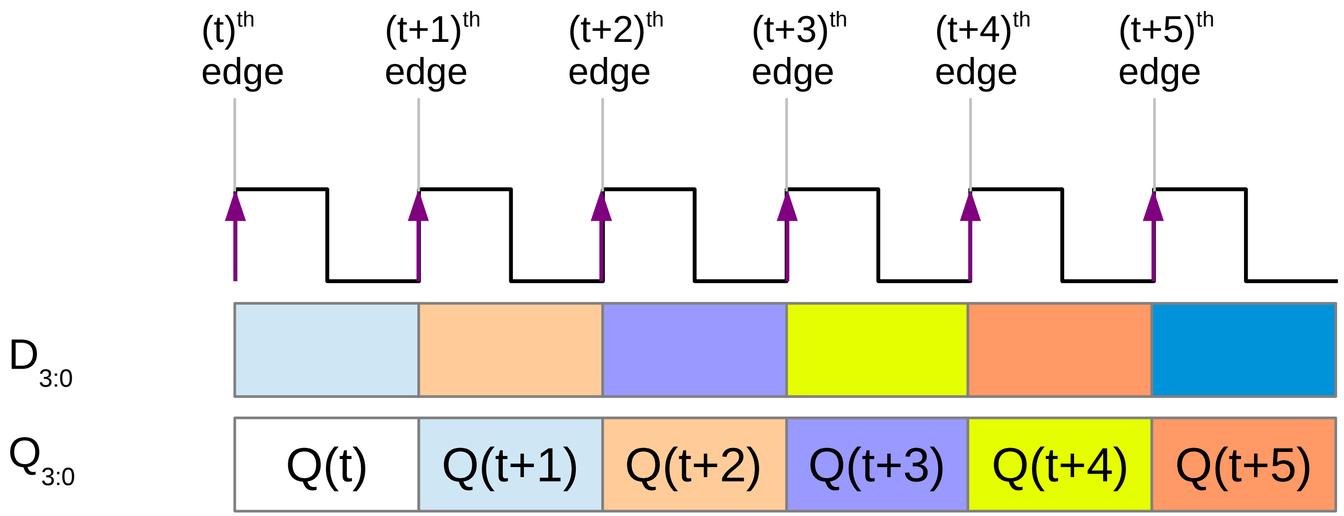
FF Timing (Ideal)





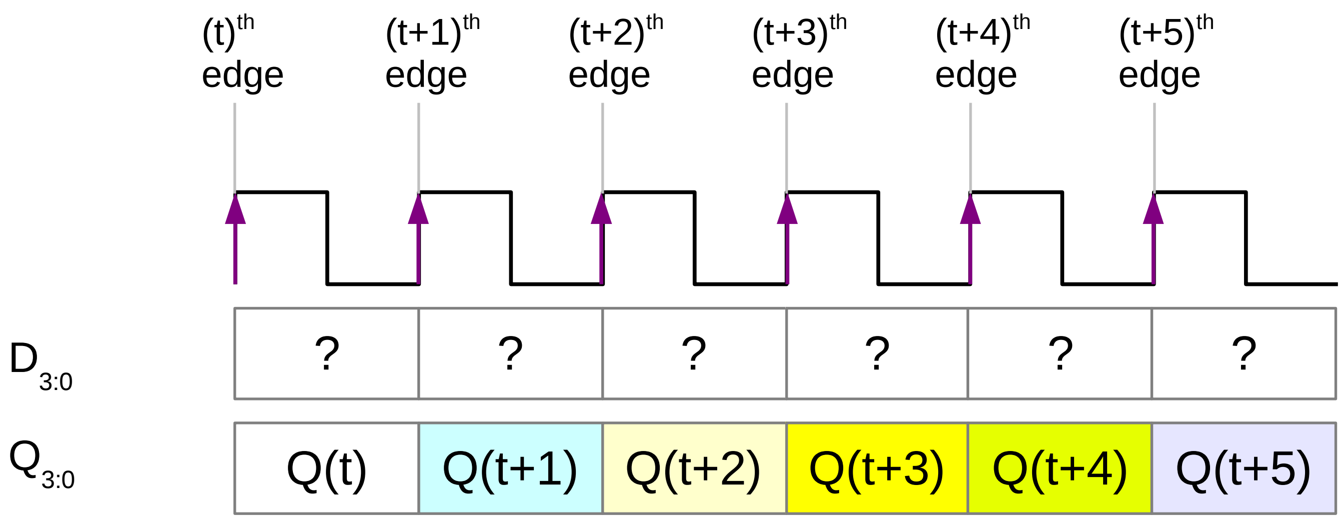
https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

Sequence of States



Inputs to FFs

Outputs of FFs

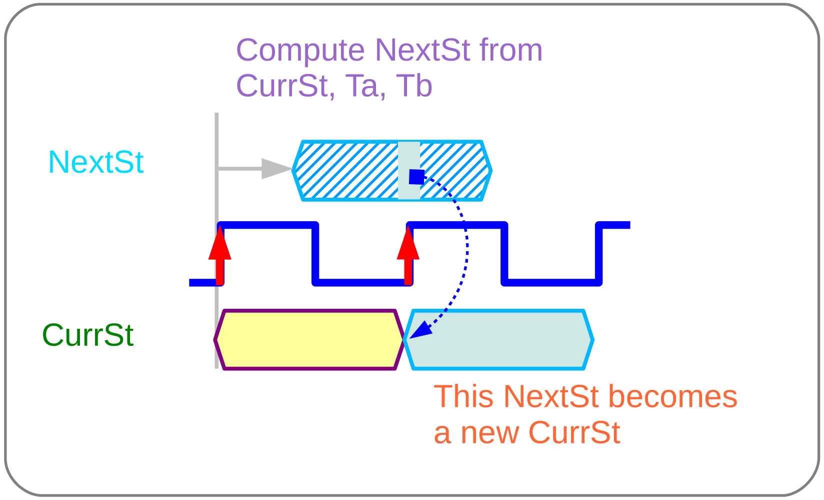


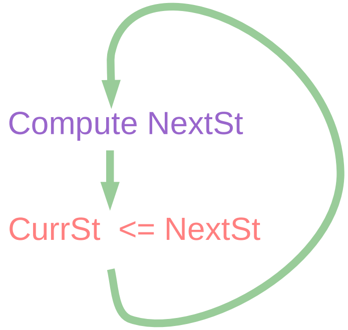
Find inputs to FFs

which will make outputs in this sequence

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

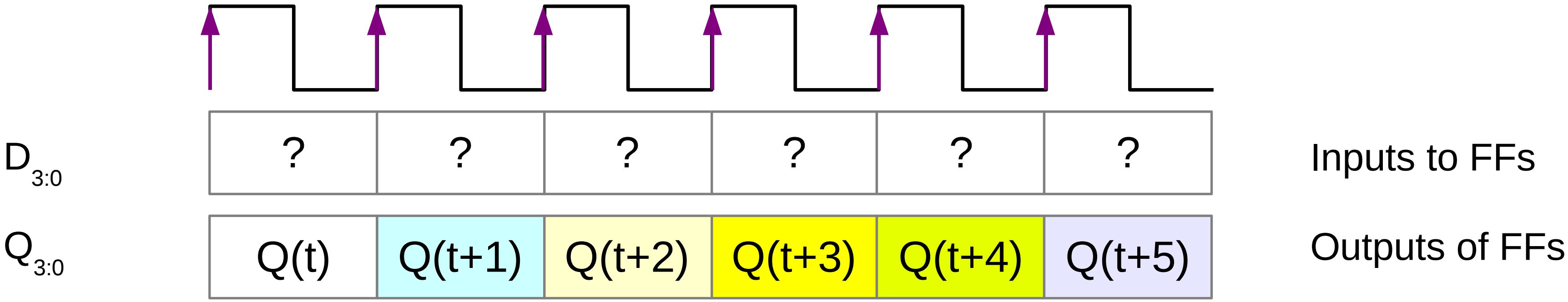
When NextSt becomes CurrSt

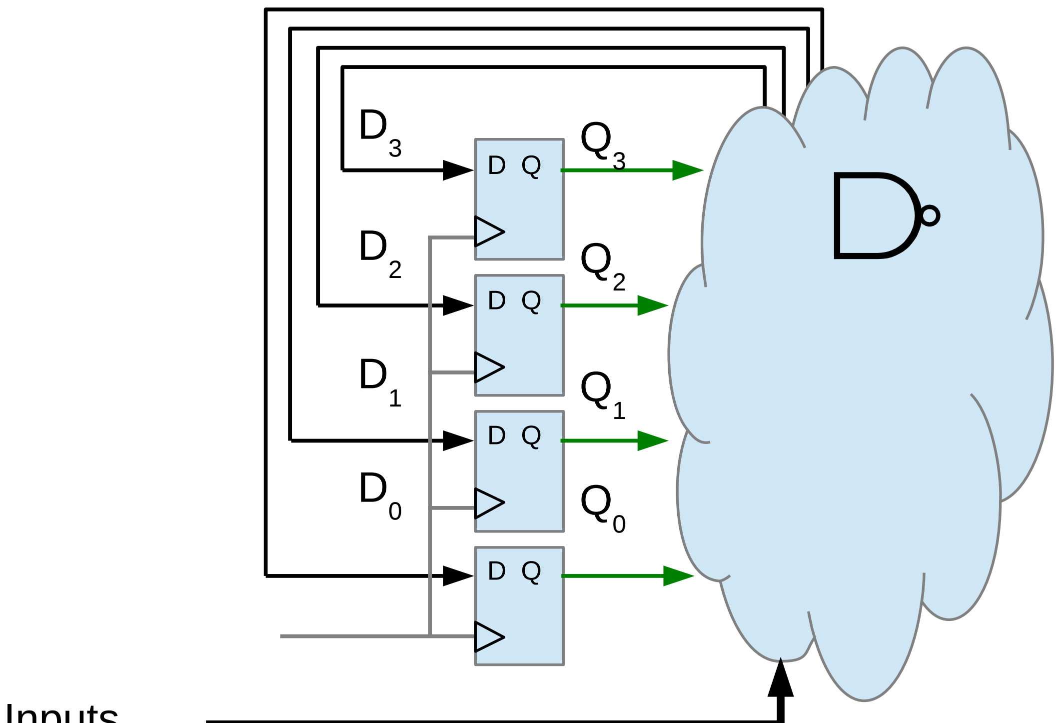




https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

Finding FF Inputs





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

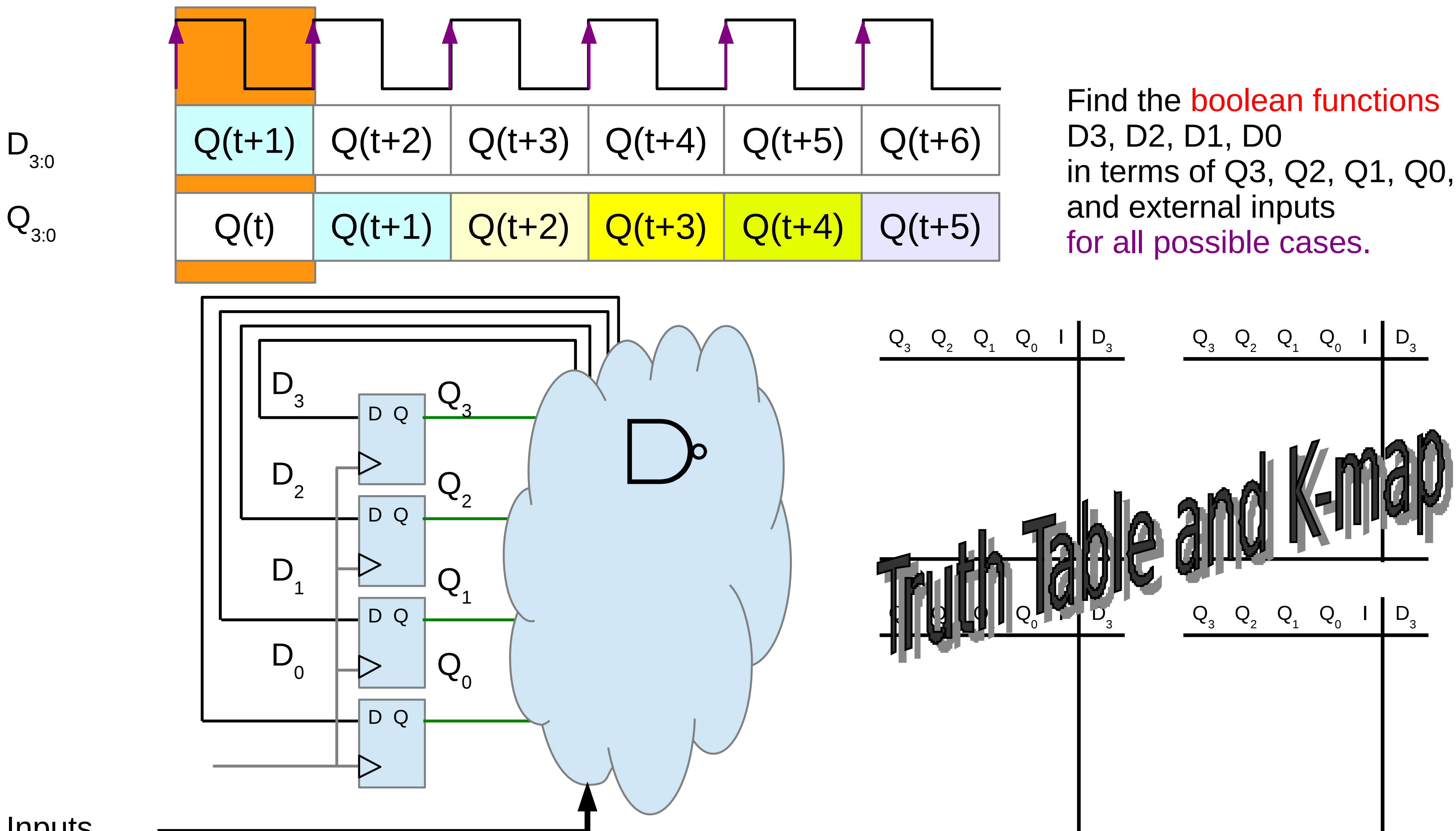
During the t^{th} clock edge period,

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

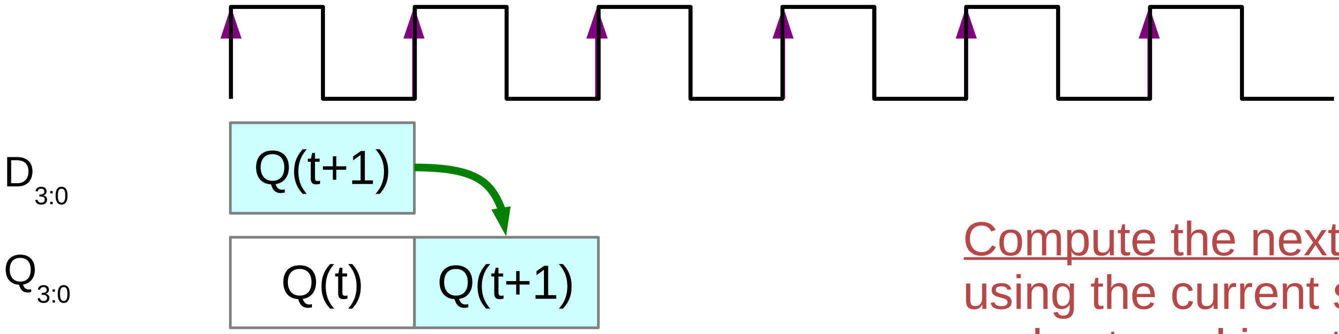
Method of Finding FF Inputs



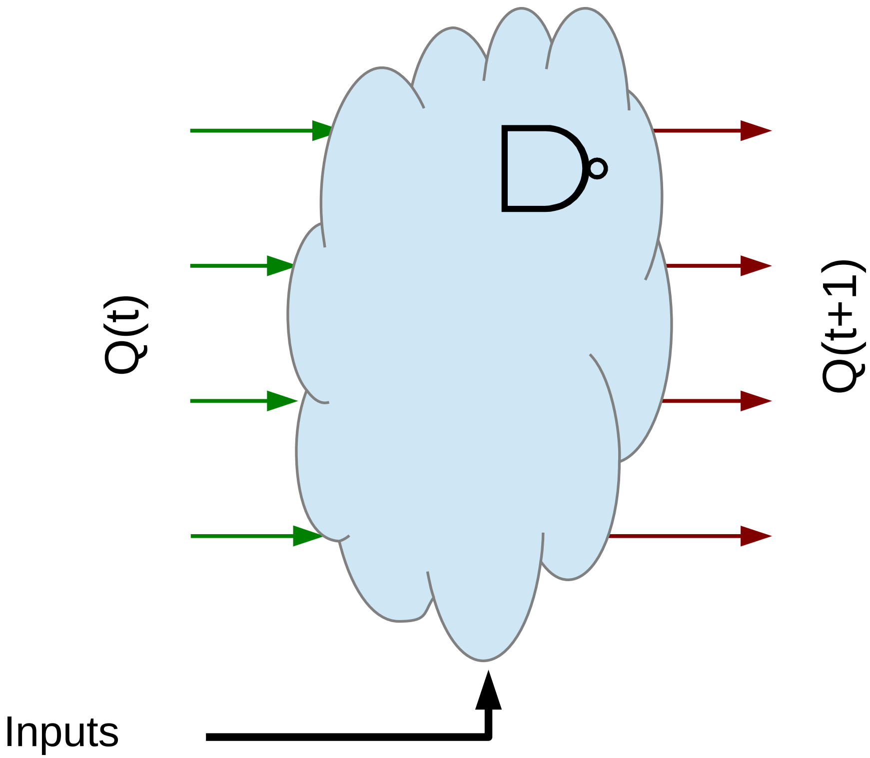
Find the boolean functions
D3, D2, D1, D0
in terms of Q3, Q2, Q1, Q0,
and external inputs
for all possible cases.

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

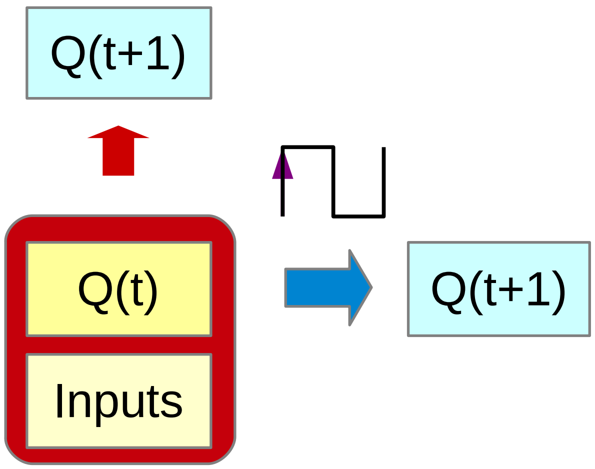
State Transition



Compute the next state
using the current state
and external inputs
in the current clock cycle

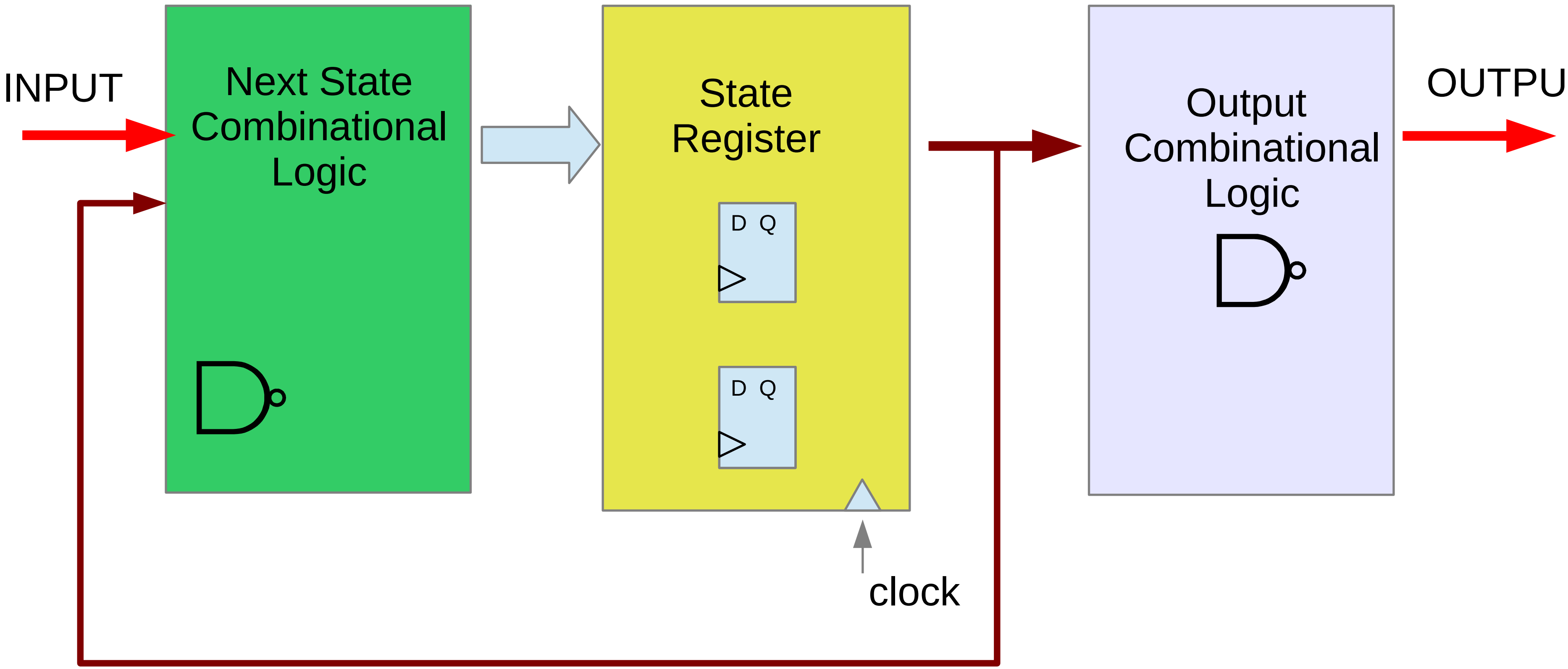


https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design



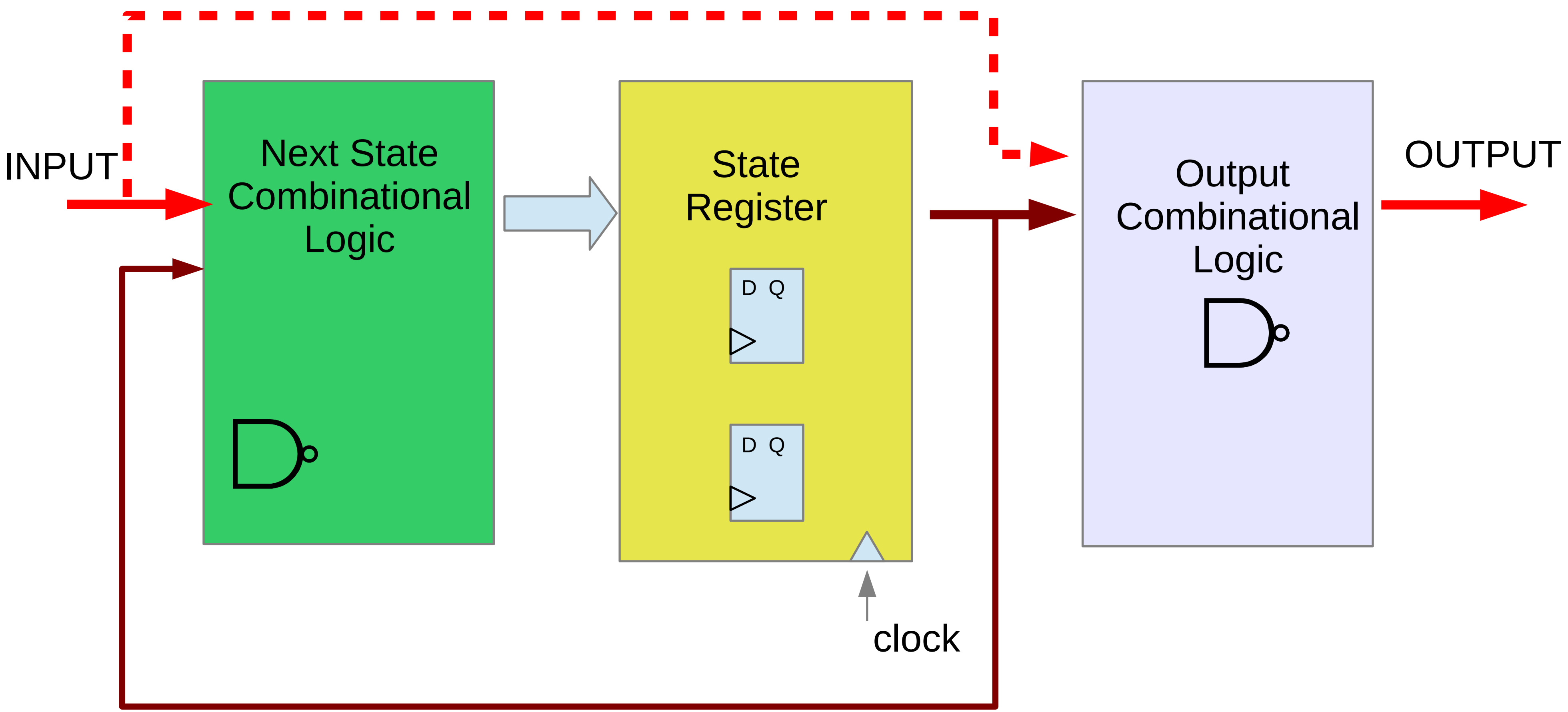
After the next clock edge,
the computed next state (FF Inputs)
becomes the current state (FF Outputs)

Moore FSM



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

Mealy Machine

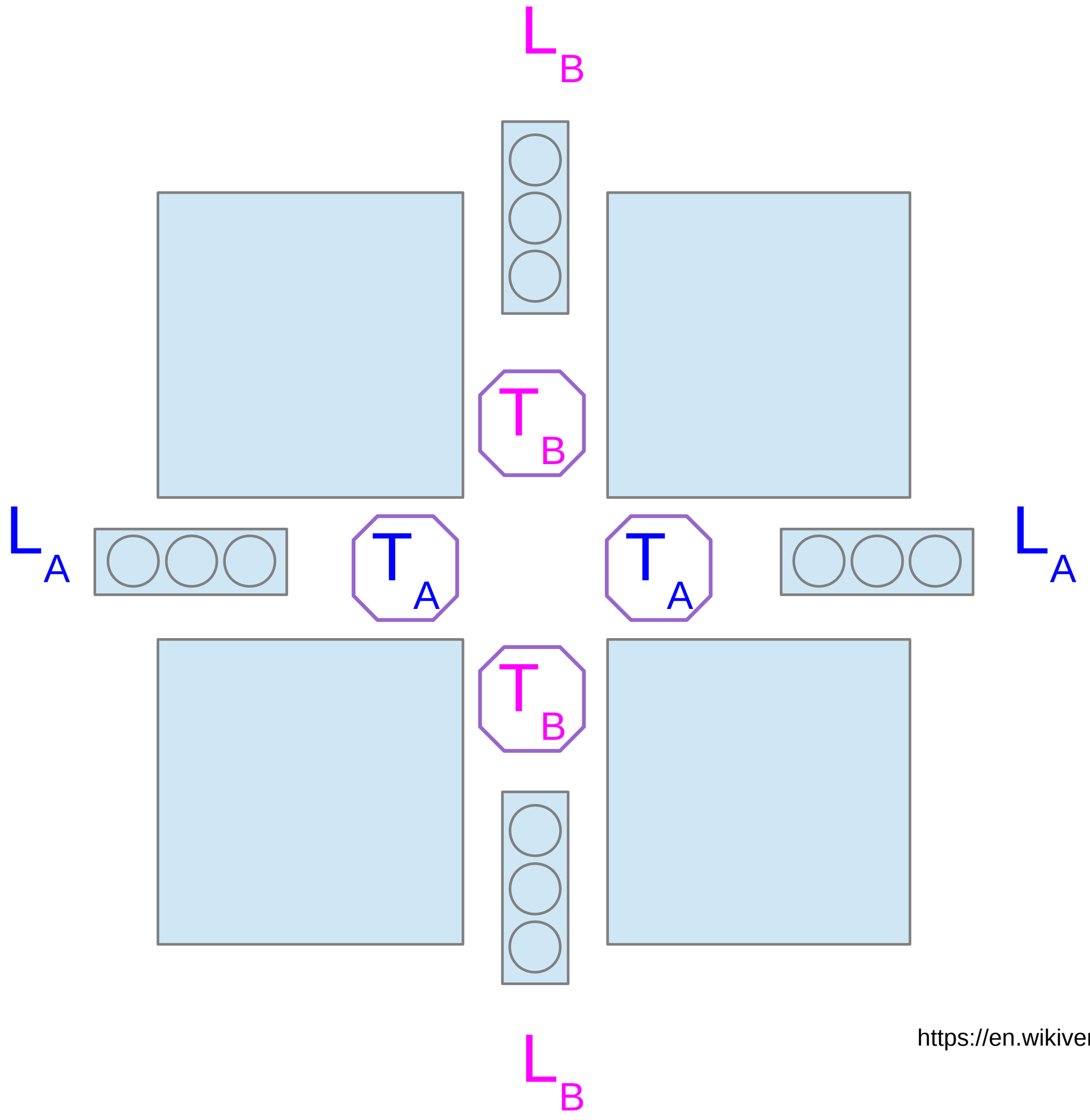


https://en.wikiversity.org/wiki/The_necessities_in_Digital_Design

Latches and FF's

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

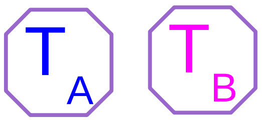
FSM Inputs and Outputs



Traffic Lights - Outputs

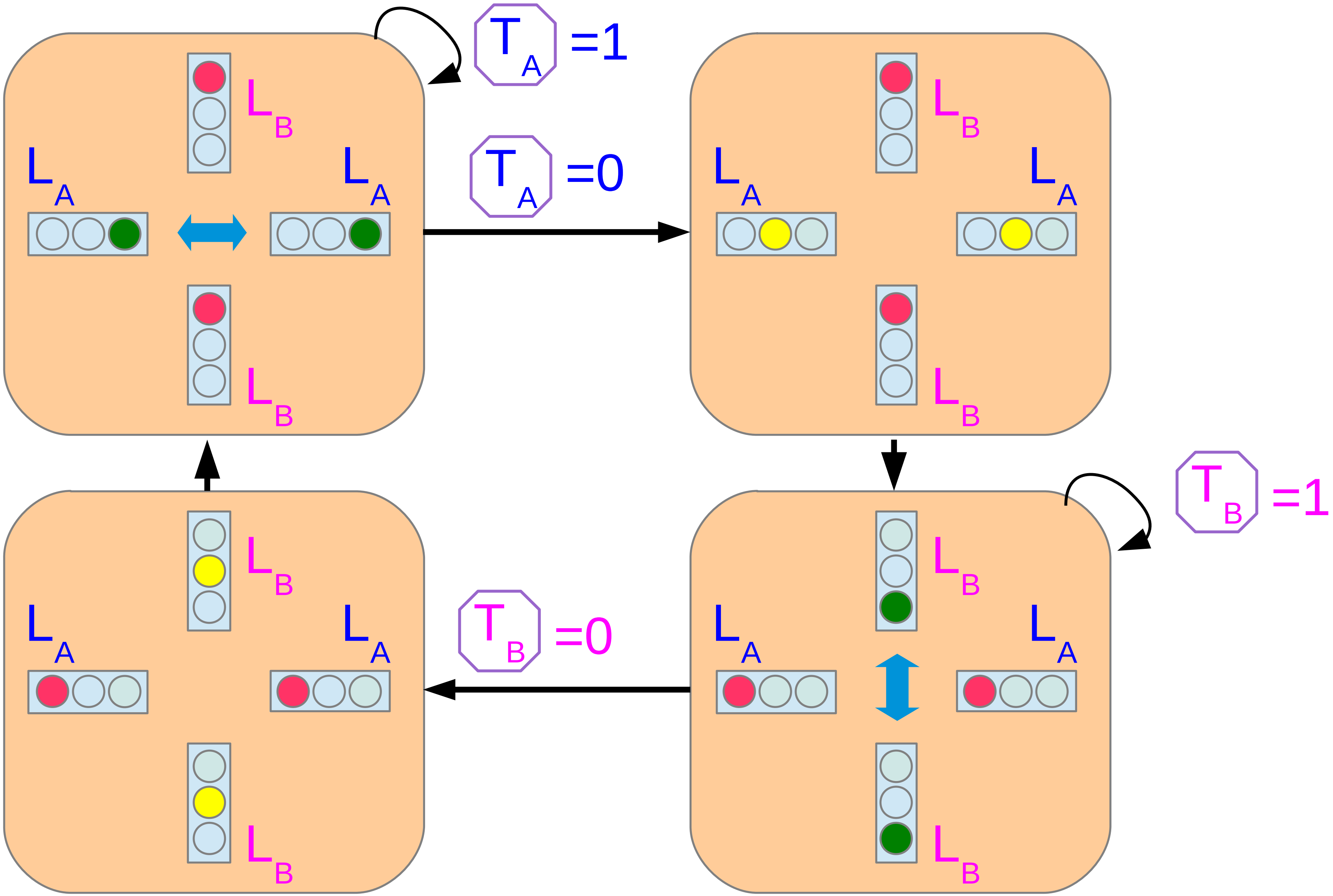
L_A L_B

Sensor - Inputs



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

States



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

Moore FSM State Transition Table

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
	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

States

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

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

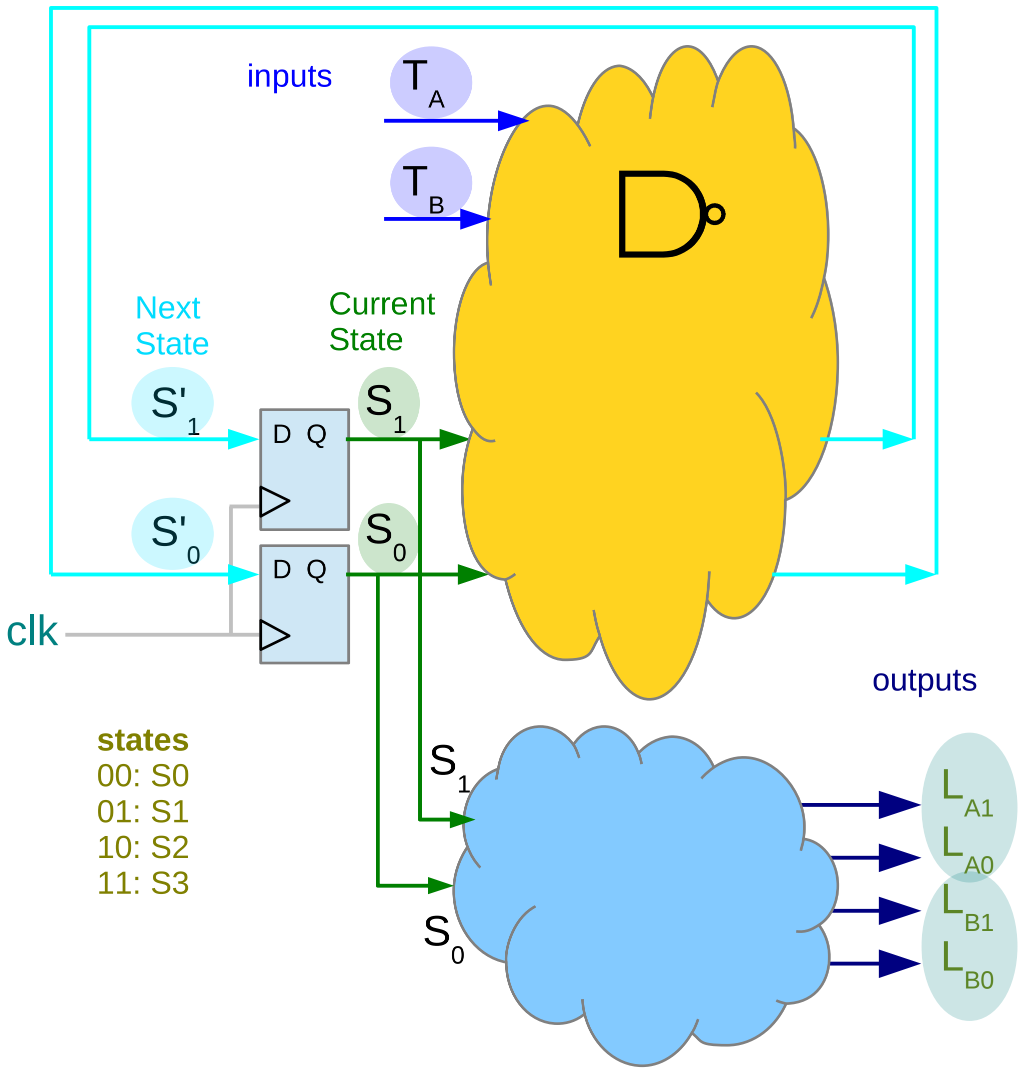
$L_{B1} = \overline{S_1}$

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

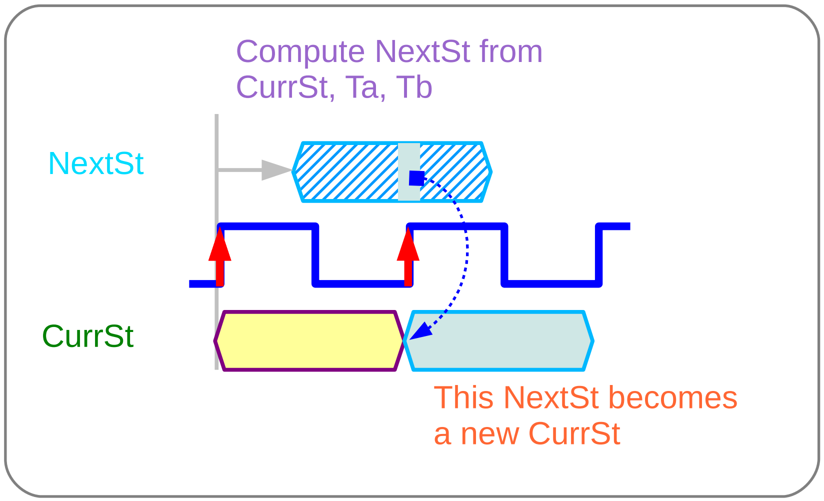
$L_{A0} = S_1 S_0$

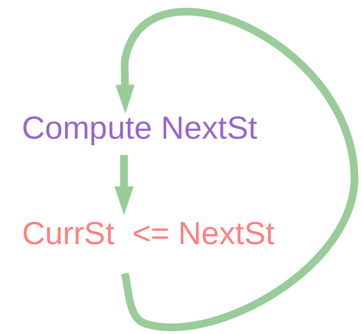
https://en.wikiversity.org/wiki/The_necessities_in_Computer_Design

Moore FSM (1)



states
00: S0
01: S1
10: S2
11: S3

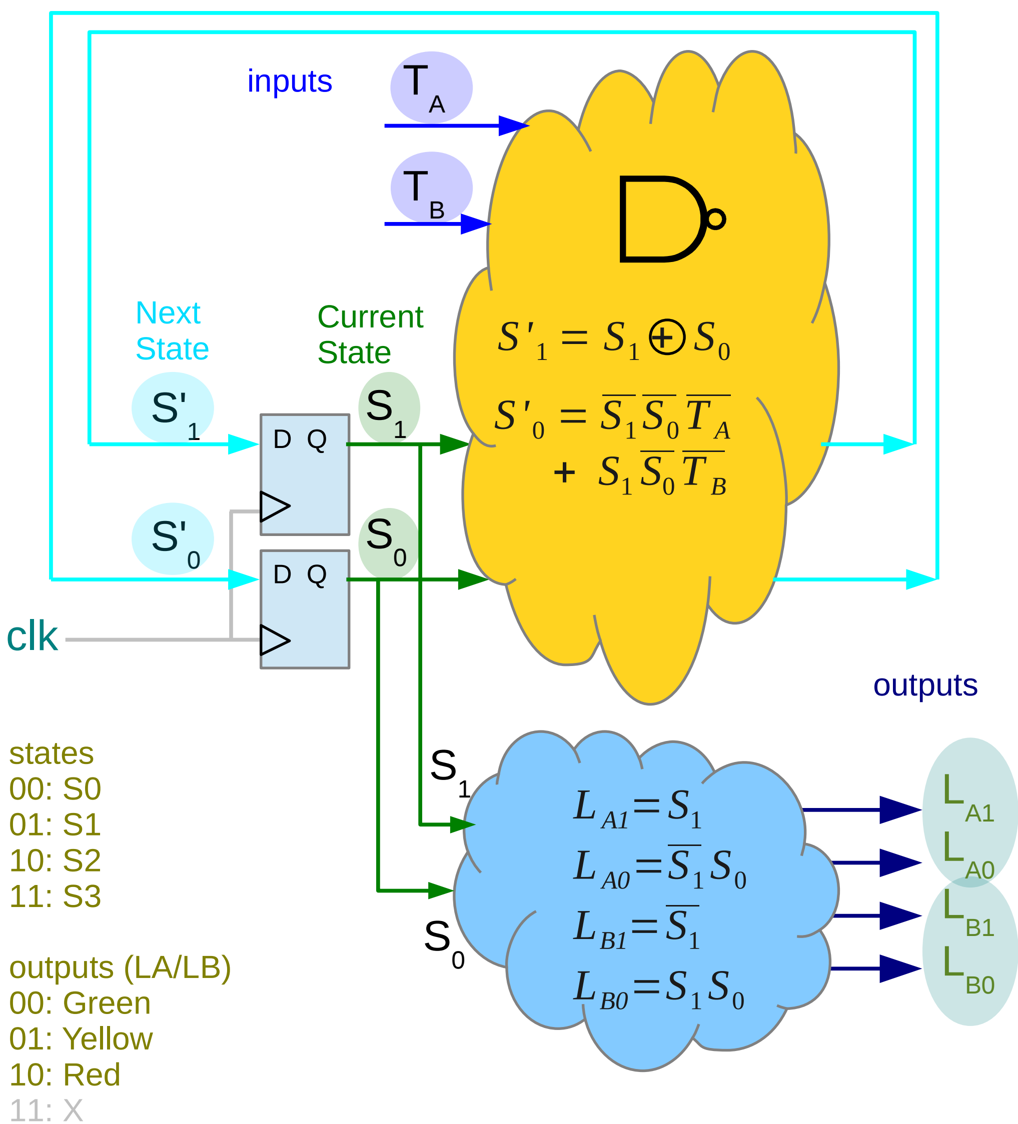




outputs (LA/LB)
00: Green
01: Yellow
10: Red
11: X

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

Moore FSM



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

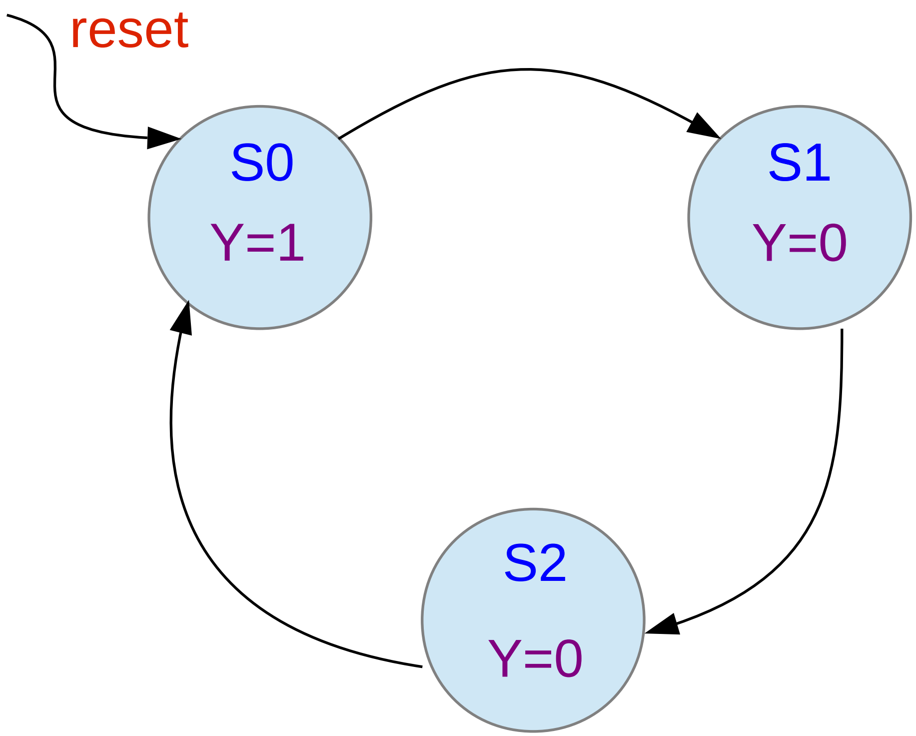
Outputs

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

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

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

Divide By N Counter FSM



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

Input: none

Output: Y=1 every 3 cycles

State Transition Table

Curr St	Next St
S0	S1
S1	S2
S2	S0

Output Table

Curr St	Output
S0	1
S1	0
S2	0

Encoding States

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

State Transition Table

Curr St	Next St
S0	S1
S1	S2
S2	S0

Output Table

Curr St	Output
S0	1
S1	0
S2	0

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

S_1	S_0	Y
0	0	1
0	1	0
1	0	0

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

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

$$Y = \overline{S_1}\,\overline{S_0}$$

State Transition Table

Curr St	Next St
S0	S1
S1	S2
S2	S0

Output Table

Curr St	Output
S0	1
S1	0
S2	0

S_2	S_1	S_0	S'_2	S'_1	S'_0
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	0	0	1

S_2	S_1	S_0	Y
0	0	1	1
0	1	0	0
1	0	0	0

$$S'_2 = \overline{S_2} S_1 \overline{S_0} \quad \Rightarrow S_1$$

$$S'_1 = \overline{S_2}\,\overline{S_1} S_0 \quad \Rightarrow S_0$$

$$S'_0 = S_2 \overline{S_1}\,\overline{S_0} \quad \Rightarrow S_2$$

$$Y = \overline{S_2}\,\overline{S_1} S_0 \quad \Rightarrow S_0$$

References

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