Lecture 5: Queue, Stack, Breadth / Depth First Search

BE5B33ALG — Algorithms

Ing. Robert Pěnička, Ph.D.
doc. RNDr. Daniel Průša, Ph.D.

Faculty of Electrical Engineering
Czech Technical University in Prague

Queue

- Operations Enqueue, Dequeue, Front, Empty....
- Cyclic queue implementation

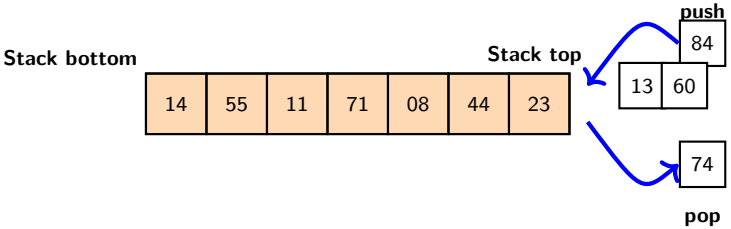
Graphs

- Breadth-first search (BFS) in a tree
- Depth-first search (DFS) in a graph
- Breadth-first search (BFS) in a graph

Search pruning

Stack

- Elements are stored at the stack top before they are processed.



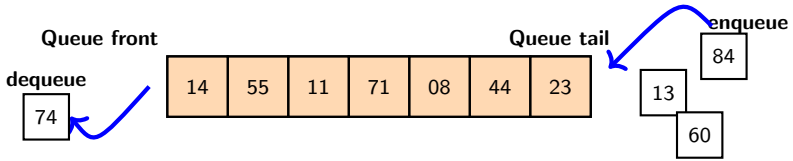
- Elements are removed from the stack top and then they are processed.
- Last In First Out (LIFO) principle.

Possible operations

- **Push** — Put at the top
- **Pop** — Remove from the top
- **Top** — Read the top
- **Empty** — Is the stack empty?

Queue

- Elements are stored at the queue tail before they are processed.



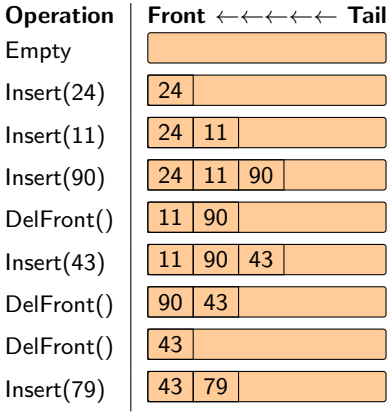
- Elements are removed from the queue front and then they are processed.
- First In First Out (FIFO) principle.

Possible operations

- **Enqueue / InsertLast / Push** — Insert at the tail
- **Dequeue / DelFront / Pop** — Remove from the front
- **Front / Peek** — Read the front element
- **Empty** — Is the queue empty?

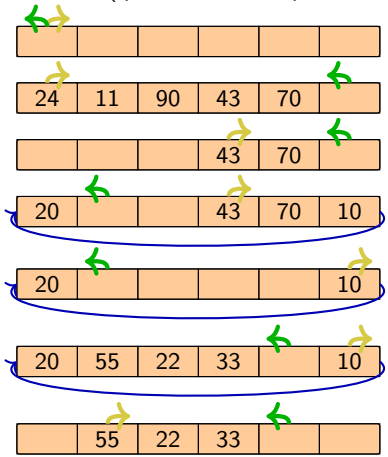
Queue – example life cycle

- **Easy example of a queue life cycle.**



Cyclic queue implementation in an array

Operation	**Queue state** (queue front ↷, queue tail ↶)
An empty queue in a fixed length array	
Insert 24, 11, 90, 43, 70	
DelFront, DelFront, DelFront	
Insert 10, 20	
DelFront, DelFront	
Insert 55, 22, 33	
DelFront, DelFront	



Cyclic queue implementation in an array

- Tail index points to the first free position behind the last queue element.
- Front index points to the first position occupied by a queue element.
- When both indices point to the same position the queue is empty.

```
class Queue:
    def __init__(self, sizeOfQ):
        self.size = sizeOfQ
        self.q = [None] * sizeOfQ
        self.front = 0
        self.tail = 0

    def isEmpty(self):
        return (self.tail == self.front)

    def Enqueue(self, node):
        if self.tail+1 == self.front or \
           self.tail - self.front == self.size-1:
            pass       # implement overflow fix here
        self.q[self.tail] = node
        self.tail = (self.tail + 1) % self.size
```

Continue...

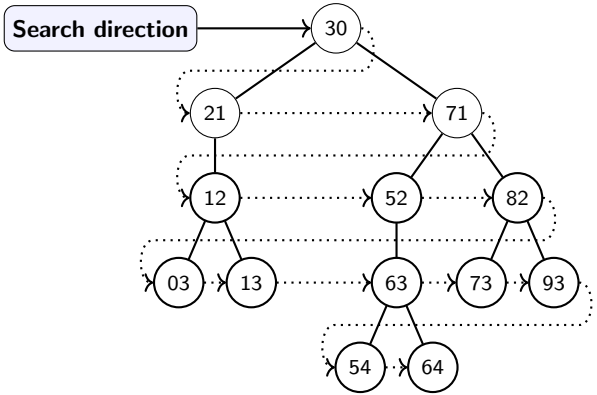
Cyclic queue implementation in an array

- Tail index points to the first free position behind the last queue element.
- Front index points to the first position occupied by a queue element.
- When both indices point to the same position the queue is empty.

... continued

```
def Dequeue(self):
    node = self.q[self.front]
    self.front = (self.front + 1) % self.size
    return node

def pop(self):
    return self.Dequeue()

def push(self, node):
    self.Enqueue(node)
```

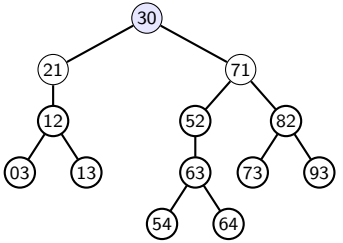
Breadth-first search (BFS) in a tree



Order of visited nodes: 30, 21, 71, 12, 52, 82, 03, 13, 63, 73, 93, 54, 64

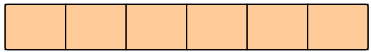
Neither the tree structure nor recursion support this approach directly.

Breadth-first search (BFS) in a tree — Initialization

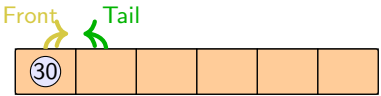


Output

- Create an empty queue.



- Enqueue the tree root.

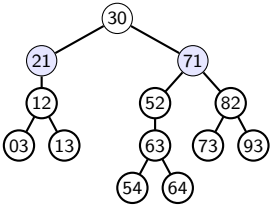


Main loop

While the queue is not empty do:

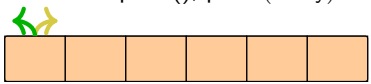
- Remove the first element from the queue and process it.
- Enqueue the children of removed element.

Breadth-first search (BFS) in a tree — iteration

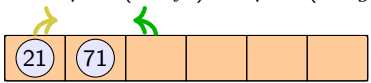


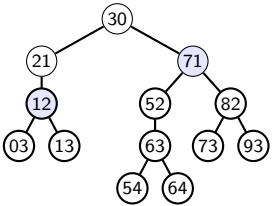
Output: 30

1. $x =$ Dequeue(), print $(x.\text{key})$.



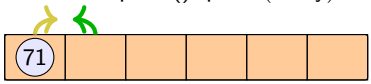
2. Enqueue$(x.left)$, Enqueue$(x.right)^{*)}$.



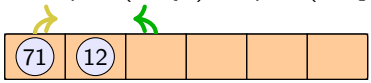


Output: 30 21

1. $x =$ Dequeue(), print $(x.\text{key})$.

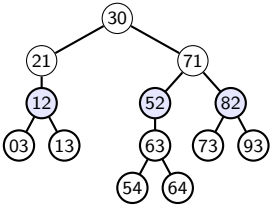


2. Enqueue$(x.left)$, Enqueue$(x.right)^{*)}$.



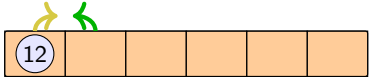
*) If the child exists.

Breadth-first search (BFS) in a tree — iteration

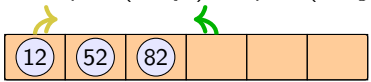


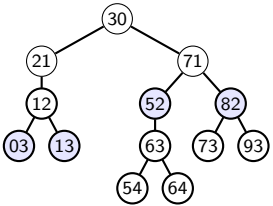
Output: 30 21 71

1. $x =$ Dequeue(), print $(x.\text{key})$.



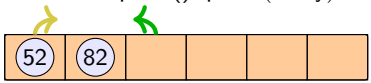
2. Enqueue$(x.left)$, Enqueue$(x.right)^{*)}$.



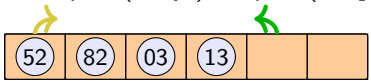


Output: 30 21 71 12

1. $x =$ Dequeue(), print $(x.\text{key})$.

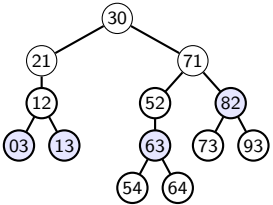


2. Enqueue$(x.left)$, Enqueue$(x.right)^{*)}$.



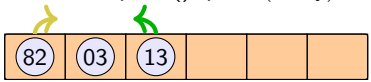
*) If the child exists.

Breadth-first search (BFS) in a tree — iteration

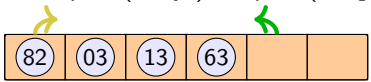


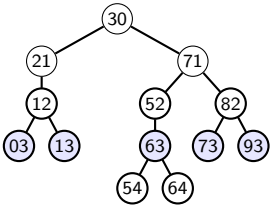
Output: 30 21 71 12 52

1. $x =$ Dequeue(), print $(x.\text{key})$.



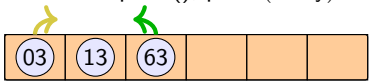
2. Enqueue$(x.left)$, Enqueue$(x.right)^{*)}$.



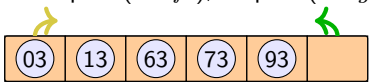


Output: 30 21 71 12 52 82

1. $x =$ Dequeue(), print $(x.\text{key})$.

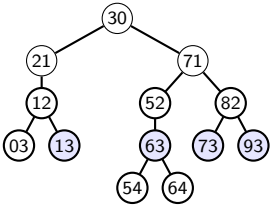


2. Enqueue$(x.left)$, Enqueue$(x.right)^{*)}$.



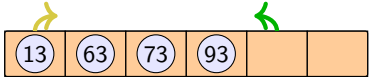
*) If the child exists.

Breadth-first search (BFS) in a tree — iteration

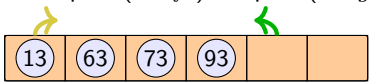


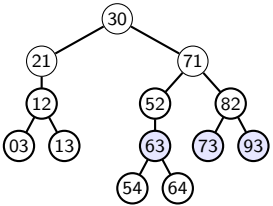
Output: 30 21 71 12 52 82 03

1. $x =$ Dequeue(), print $(x.\text{key})$.



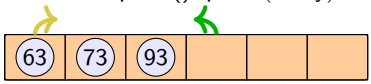
2. Enqueue$(x.left)$, Enqueue$(x.right)^{*)}$.



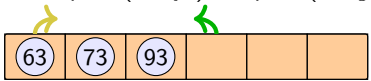


Output: 30 21 71 12 52 82 03 13

1. $x =$ Dequeue(), print $(x.\text{key})$.

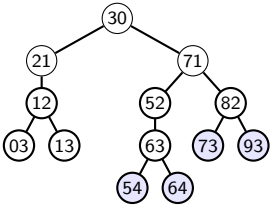


2. Enqueue$(x.left)$, Enqueue$(x.right)^{*)}$.



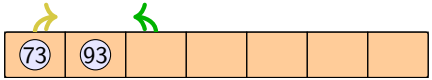
*) If the child exists.

Breadth-first search (BFS) in a tree — iteration

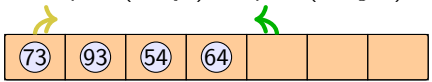


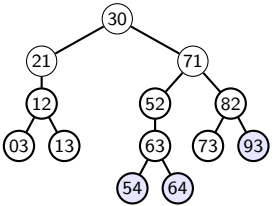
Output: 30 21 71 12 52 82 03 13 63

1. $x =$ Dequeue(), print $(x.\text{key})$.



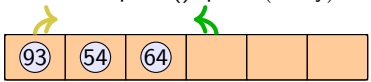
2. Enqueue$(x.left)$, Enqueue$(x.right)^{*)}$.



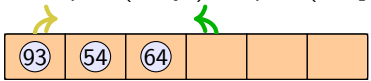


Output: 30 21 71 12 52 82 03 13 63 73

1. $x =$ Dequeue(), print $(x.\text{key})$.

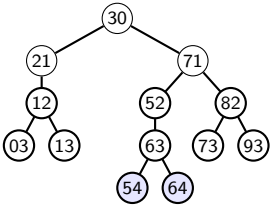


2. Enqueue$(x.left)$, Enqueue$(x.right)^{*)}$.



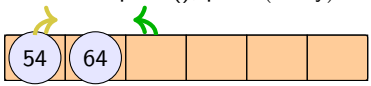
$^{*)}$ If the child exists.

Breadth-first search (BFS) in a tree — iteration

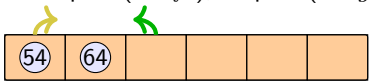


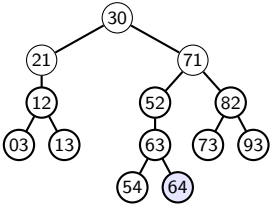
Output: 30 21 71 12 52 82 03 13 63 73 93

1. x = Dequeue(), print (x.key).



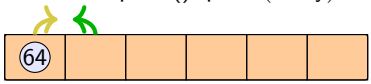
2. Enqueue($x.left$), Enqueue($x.right$)$^{*)}$.



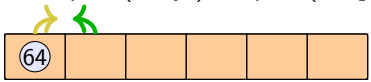


Output: 30 21 71 12 52 82 03 13 63 73 93 54

1. x = Dequeue(), print (x.key).

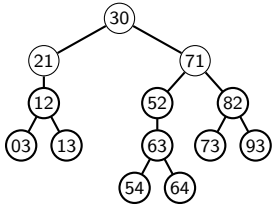


2. Enqueue($x.left$), Enqueue($x.right$)$^{*)}$.



$^{*)}$ If the child exists.

Breadth-first search (BFS) in a tree — completion



Output: 30 21 71 12 52 82 03 13 63 73 93 54 64

1. $x =$ Dequeue(), print $(x.\text{key})$.

2. Enqueue$(x.left)$, Enqueue$(x.right)^{*)}$.

The queue is empty, BFS is complete.

An nonempty **queue** always contains exactly:

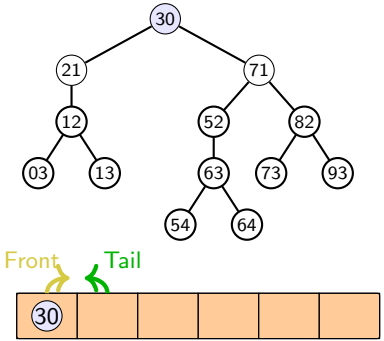
- some (or all) nodes of one level and
- all children of those nodes of this level which have already left the queue.

Sometimes the queue contains just nodes of one level.

Breadth-first search (BFS) in a tree

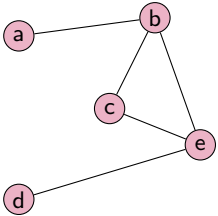
- BFS in a tree graph algorithm.

```
def binaryTreeBFS(node):
    if node == None:
        return
    q = Queue(100)                  # init
    q.Enqueue(node)                 # root into queue
    while (not q.isEmpty()):
        node = q.Dequeue()
        print(node.key, end=' ')    # process node
        if node.left != None:
            q.Enqueue(node.left)
        if node.right != None:
            q.Enqueue(node.right)
```



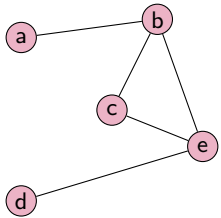
Graphs recap

- Graph is an ordered pair of set of vertices (nodes) V and set of pairs of vertices E.
- Each pair in E is an edge.
- Graph is $G = (V, E)$
- Example:
  - $V = \{a, b, c, d, e\}$
  - $E = \{\{a, b\}, \{b, e\}, \{b, c\}, \{c, e\}, \{e, d\}\}$



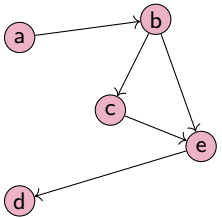
Undirected graph

- An edge is an **unordered** pair of vertices.
- $E = \{\{a, b\}, \{b, e\}, \{b, c\}, \{c, e\}, \{e, d\}\}$



Directed graph

- An edge is an **ordered** pair of vertices.
- $E = \{(a, b), (b, e), (b, c), (c, e), (e, d)\}$



Graph – adjacency matrix

- Let $G = (V, E)$ be a graph with n vertices.
- Denote vertices $v_1, \ldots, v_n$ (in an arbitrary order).
- Adjacency matrix of G is a matrix of order n

$$A_G = (a_{i,j})_{i,j=1}^{n}$$

defined by the relation

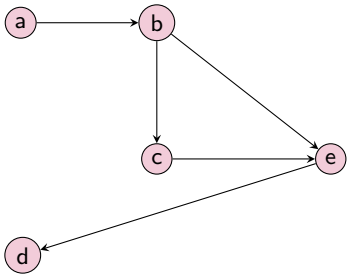
$$a_{i,j} = \begin{cases} 1 & \text{for } \{v_i, v_j\} \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Directed graph example

	a	**b**	**c**	**d**	**e**
a	0	1	0	0	0
b	0	0	1	0	1
c	0	0	0	0	1
d	0	0	0	0	0
e	0	0	0	1	0

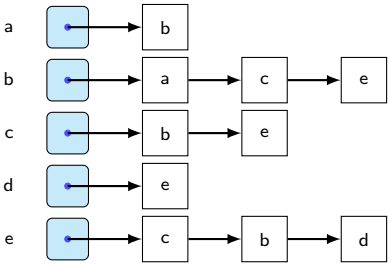
Directed graph example

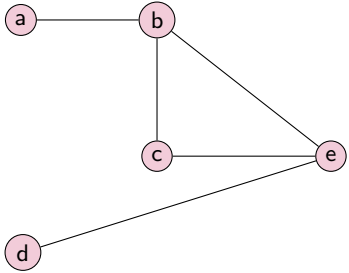
	a	b	c	d	e
a	0	1	0	0	0
b	0	0	1	0	1
c	0	0	0	0	1
d	0	0	0	0	0
e	0	0	0	1	0



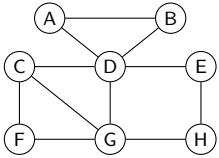
Graph – list of neighbours

- Let $G = (V, E)$ be an (un)directed graph with n vertices.
- Denote vertices $v_1, \ldots, v_n$ (in an arbitrary order).
- **List of neighbours** of G is an array $\mathcal{P}$ of size n of pointers.
  - $\mathcal{P}[i]$ points to the list of all vertices which are adjacent to v_i.

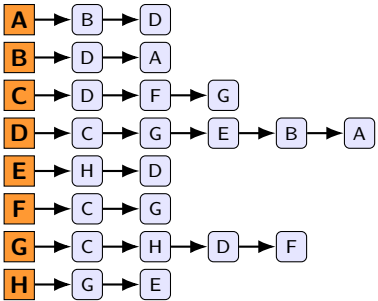




Graph most usual representations



Linked list representation

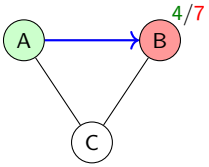


Adjacency matrix

	A	B	C	D	E	F	G	H
A	0	1	0	1	0	0	0	0
B	1	0	0	1	0	0	0	0
C	0	0	0	1	0	1	1	0
D	1	1	1	0	1	0	1	0
E	0	0	0	1	0	0	0	1
F	0	0	1	0	0	0	1	0
G	0	0	1	1	0	1	0	1
H	0	0	0	0	1	0	1	0

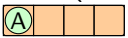
Depth-first search (DFS) in a graph — notation for example that follows

- **Node colors:**
  - **White node: fresh node** not visited yet.
  - **Green node:** currently **open node** (discovered, but not finished/closed).
  - **Red node: closed node** (completely explored/finished).
- **Numbers on nodes:** X/Y
  - X = iteration when first visited / added to the stack.
  - Y = finish iteration (when removed from the stack).
- **Blue arrow:** the DFS is currently expanded along this edge.
- **Stack:** a node is *pushed* onto the stack when discovered (green).
- **Output:** a node is *printed* when it is first entered.

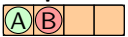


Example DFS state

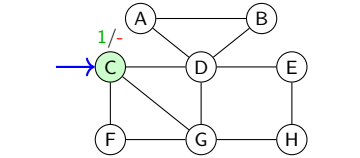
Stack (active nodes):

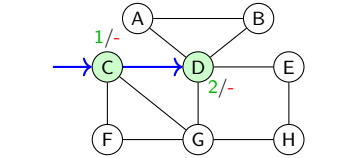


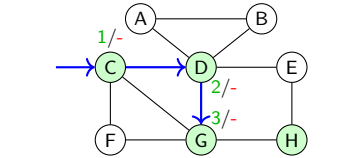
Output:

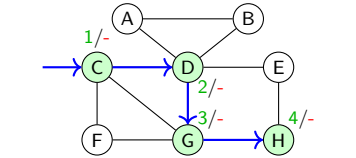


Depth-first search (DFS) in a graph

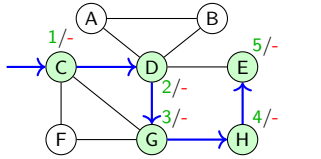


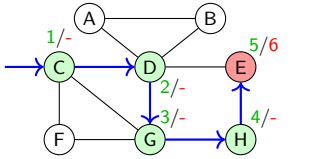


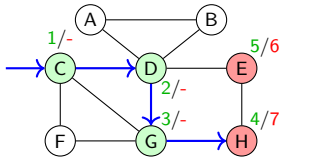


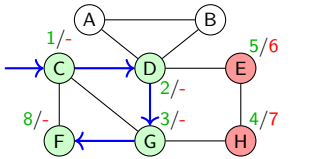


Depth-first search (DFS) in a graph

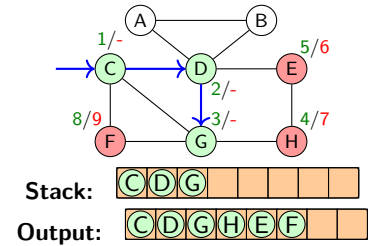


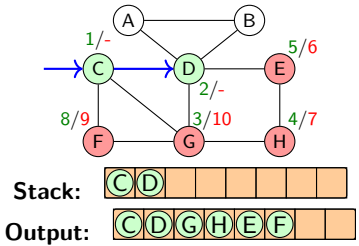


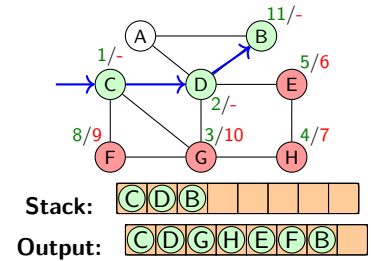


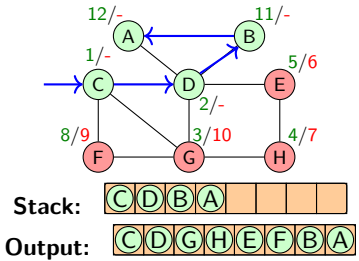


Depth-first search (DFS) in a graph

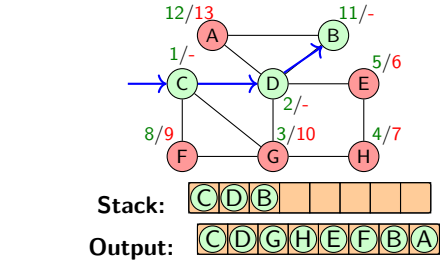


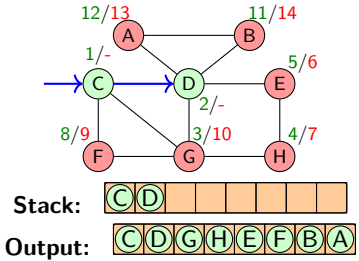


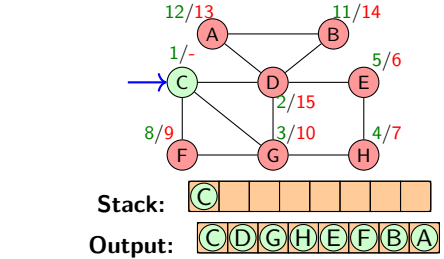


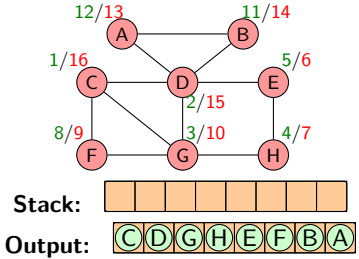


Depth-first search (DFS) in a graph









Depth-first search (DFS) in a graph — Life cycle

Life cycle of a node during the DFS: Fresh – Open – Closed

Fresh nodes

- Fresh nodes are those nodes which have not been visited yet.
- Before the search starts, all nodes are fresh.
- A fresh node becomes *open* when it is visited for the first time.
- The set of fresh nodes shrinks or remains the same during the search.

Open nodes

- Open nodes are those nodes which have already been visited but are not *closed* yet.
- The set of open nodes may grow and shrink during the search.

Closed nodes

- Closed nodes are those nodes which will not be visited any more.
- When each neighbour of a current node in the search is either open or closed, the current node becomes closed.
- The set of closed nodes only grows during the search.
- When the search terminates, all nodes are closed.

Implementation remark

- **Fresh:** A fresh node is assigned no time (neither open nor closed).
- **Open:** An open node is assigned open time and no close time.
- **Closed:** A closed node is assigned both open and close times.

In some implementations, it is not necessary to produce the open and close times.

However, **it is always necessary to register explicitly the state of each node – fresh/closed.** Open nodes are then those which were not closed yet and are still on the stack.

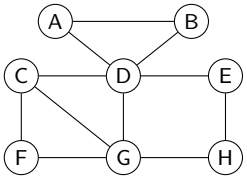
In the recursive variant of DFS, each recursive call corresponds to a single node processing, including all visits to this node. The node becomes open when the node is the actual parameter of the current recursive function call. The node becomes closed when the same call terminates.

The neighbours of the node are checked one by one in the body of the function, and the fresh ones become the parameters of the recursive calls. Therefore, **it is enough to register only one-bit information in each node: Fresh or not fresh**.

Depth-first search (DFS) in a graph

Stack contents

C
C D
C D G
C D G H
C D G H E
C D G H
C D G
C D G F
C D G
C D
C D B
C D B A
C D B
C D
C
–



Printing the node when it becomes open:

$$C\ D\ G\ H\ E\ F\ B\ A$$

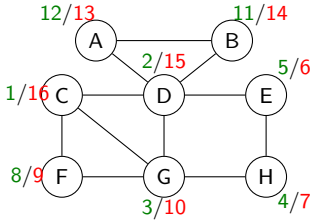
Printing the node when it becomes closed:

$$E\ H\ F\ G\ A\ B\ D\ C$$

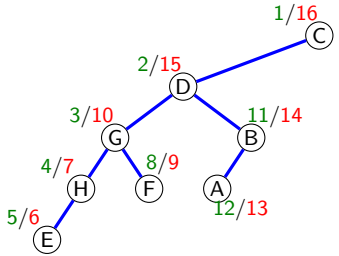
Processing a node when it becomes closed is used in algorithms of:

- bridges and cut-vertices detection in undirected graphs
- strongly connected components detection in directed graphs

Depth-first search (DFS) in a graph



DFS-tree with open and close times of the nodes



Properties:

- For subtree rooted at X it holds for each node $Y \neq X$:
  $\text{OpenTime}(X) < \text{OpenTime}(Y) < \text{CloseTime}(Y) < \text{CloseTime}(X)$
- If Y is not in subtree of X: $\text{CloseTime}(X) < \text{OpenTime}(Y)$ or
  $\text{CloseTime}(Y) < \text{OpenTime}(X)$
- Number of nodes in subtree of X is always: $\big(\text{CloseTime}(X) + 1 - \text{OpenTime}(X)\big)/2$

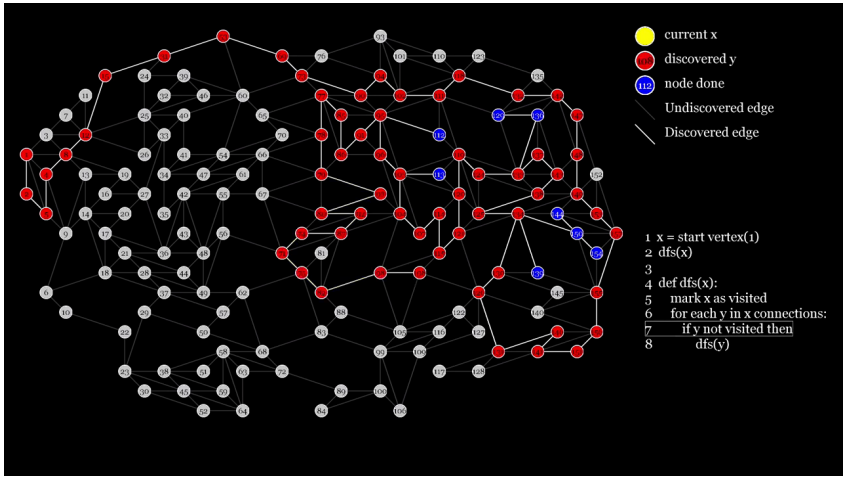
Depth-first search (DFS) in a graph – iteratively


```
def DFS(graph):
    visited = [False] * graph.size
    stack = Stack()
    stack.push(graph.nodes[0])      # start search in node 0
    visited[0] = True

    while not stack.isEmpty():
        node = stack.pop()
        print(node.id, end=" ")     # process the node
        for neigh in node.neighbours:
            if not visited[neigh.id]:
                stack.push(neigh)
                visited[neigh.id] = True
```

Depth-first search (DFS) in a graph – recursively


```
def DFSrec(node, visited):
    visited[node.id] = True
    print(node.id, end=" ")      # process the node
    for neigh in node.neighbours:
        if visited[neigh.id] == False:
            DFSrec(neigh, visited)

def DFSrecRun(graph):
    visited = [False] * graph.size
    DFSrec(graph.nodes[0], visited)
```

DFS animation



https://www.youtube.com/watch?v=NUgMa5coCoE

Breadth-first search (BFS) in a graph

Life cycle of a node during BFS
is conceptually identical to the node lifecycle during DFS.

Fresh nodes

- Fresh nodes are those nodes which have not been visited yet.
- Before the search starts, all nodes are fresh.
- A fresh node becomes **open** when it is visited for the first time.
- The set of fresh nodes shrinks or remains the same during the search.

Open nodes

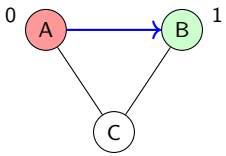
- Open nodes are those nodes which have been already visited but were not **closed** yet.
- The set of open nodes may grow and shrink during the search.

Closed nodes

- Closed nodes are those nodes which will not be visited any more.
- When each neighbour of a current node in the search is either open or closed, the current node becomes closed.
- The set of closed nodes only grows during the search.
- When the search terminates, all nodes are closed.

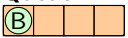
Breadth-first search (BFS) in a graph — notation for example that follows

- **Node colors:**
  - **White node: fresh node** not visited yet.
  - **Green node:** currently **open node** (discovered, in the queue).
  - **Red node: closed node** (fully processed, removed from queue).
- **Numbers on nodes:** k
  - k = distance from the BFS start node.
- **Blue arrow:** BFS is currently exploring an edge from the active node.
- **Queue:** a node is *enqueued* when discovered (green). The queue evolves as BFS processes nodes in FIFO order.
- **Output:** a node is *printed* when it is processed (dequeued/closed).

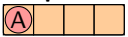


Example BFS state

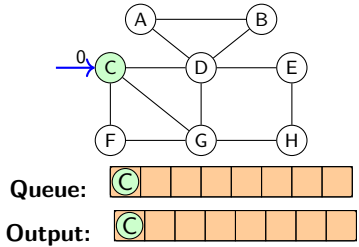
Queue:

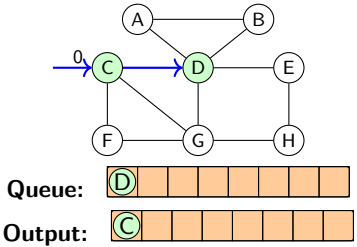


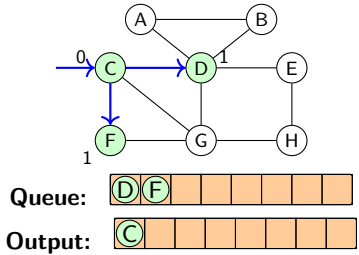
Output:

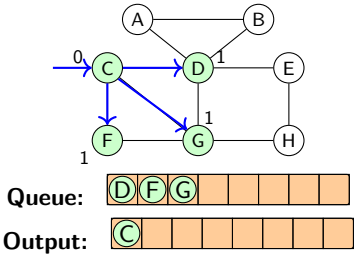


Breadth-first search (BFS) in a graph

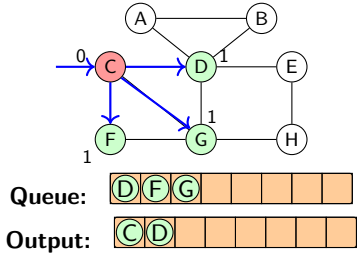


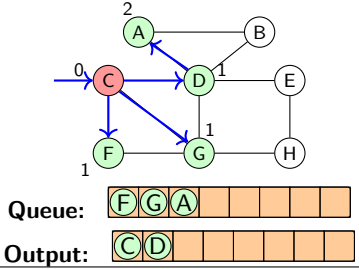


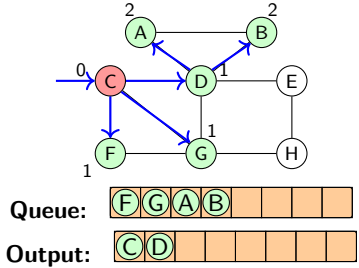


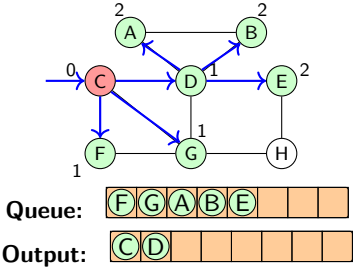


Breadth-first search (BFS) in a graph

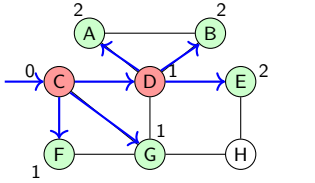






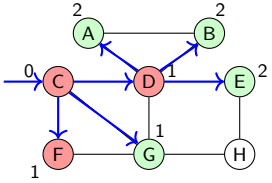


Breadth-first search (BFS) in a graph



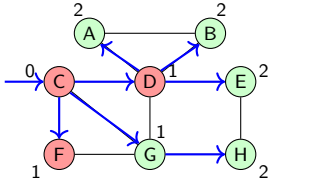
Queue: F G A B E

Output: C D F



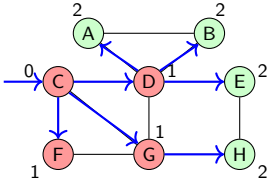
Queue: G A B E

Output: C D F G



Queue: A B E H

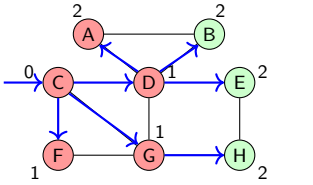
Output: C D F G

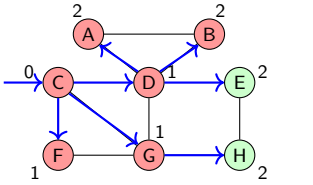


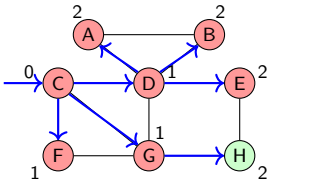
Queue: A B E H

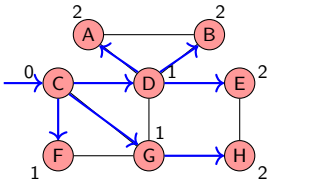
Output: C D F G A

Breadth-first search (BFS) in a graph

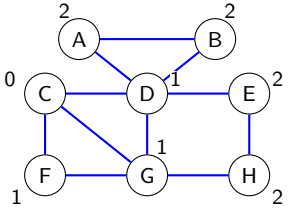






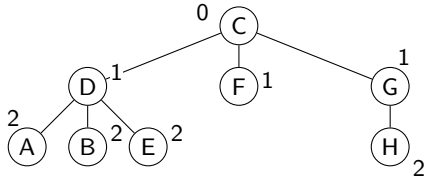


Breadth-first search (BFS) in a graph



The open and close times are not essential in BFS.

BFS-tree with distances from the start node (root)



The node depth in the BFS tree is equal to its distance from the start node in BFS.

Applications of BFS

- Testing graph connectivity
- Testing existence of a cycle in a graph
- Testing if a graph is bipartite
- Computing distance(s) from a given node to one or all other nodes

Breadth-first search (BFS) in a graph — Implementation remark

- **Fresh:** A fresh node is assigned no distance from the start node.
- **Open:** An open node is assigned a distance from the start node and it is in the queue.
- **Closed:** A closed node is assigned a distance from the start node and it is not in the queue.

It is not necessary to register explicitly fresh/open/closed state of the nodes. The contents of the queue and the distance (assigned / not assigned) define unambiguously the node state.

BFS is an iterative process — a recursive variant is not used. (A recursive implementation would be more artificial and less clear.)

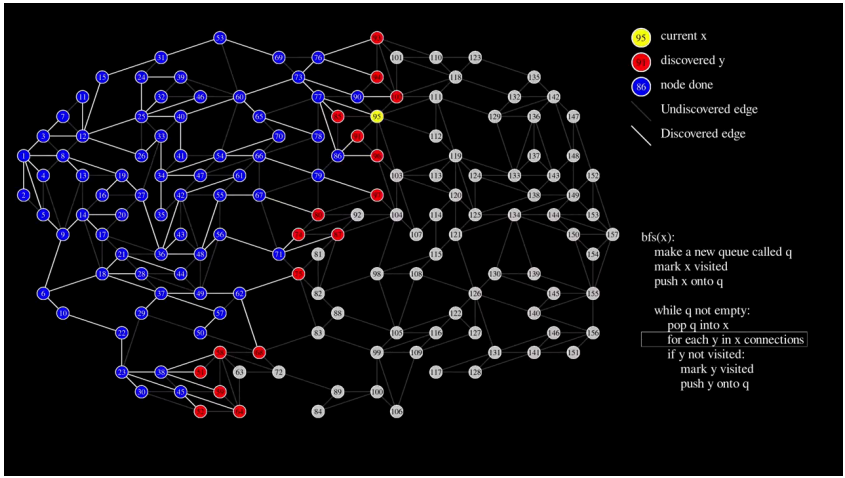
Breadth-first search (BFS) in a graph


```
def BFS(graph):
    visited = [False] * graph.size
    queue = Queue(200)
    queue.Enqueue(graph.nodes[0])
    visited[0] = True
    while not queue.isEmpty():
        node = queue.Dequeue()
        print(node.id, end=" ")    # process node
        for neigh in node.neighbours:
            if not visited[neigh.id]:
                queue.Enqueue(neigh)
                visited[neigh.id] = True
```

Node distances by BFS in a graph


```
def BFSdist(graph):
    visited = [False] * graph.size
    dist = [9999999999999] * graph.size    # infinity == 99...9
    queue = Queue(graph.size)

    queue.Enqueue(graph.nodes[0])          # start in node 0
    visited[0] = True
    dist[0] = 0

    while not queue.isEmpty():
        node = queue.Dequeue()
        print(node.id, end=" ")            # process node
        for neigh in node.neighbours:
            if not visited[neigh.id]:
                queue.Enqueue(neigh)
                visited[neigh.id] = True
                dist[neigh.id] = dist[node.id] + 1

    print(dist)   # process the distances or return, etc.
```

BFS animation



https://www.youtube.com/watch?v=x-VTfcmrLEQ

Asymptotic complexity

Each single operation on the queue/stack and each single operation on additional data structures and nodes/edges is of constant time (and memory) complexity.

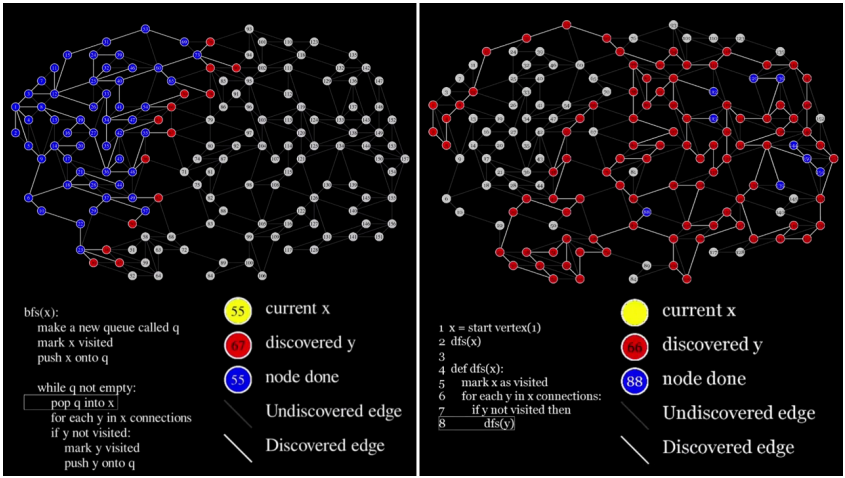
Each node enters the queue/stack only once and it leaves the queue/stack only once. The state of the node (fresh/open/closed) is tested more times. The number of these tests is equal to the degree of the node (the search tries to access the node from its neighbours).

The sum of all node degrees is equal to twice the number of edges, in any graph.

In total:

$$\Theta(|V| + |E|)$$

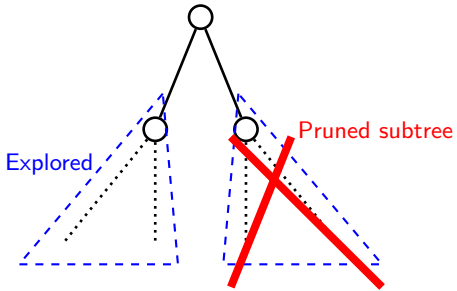
DFS and BFS comparison animation



https://www.youtube.com/watch?v=VsEla1bVVro

Search pruning

- Search speedup
- Pruning (skipping) of unpromising possibilities
- When the analysis of the current state reveals that
  - it is an unpromising state
  - surely it does not lead to the solution
- We **"cut off"** (prune) the whole subtree of states of which the current state is the root



Pruning example – magic square

- **Magic square of order $\mathcal{N}$**
  - square matrix of order $\mathcal{N}$
  - contains exactly once each value from 1 to $\mathcal{N}^2$
  - sum of all rows and all columns is the same
- Example

2	9	4
7	5	3
6	1	8

- Brute force approach: Generate all possible permutations of positions of numbers from 1 to $\mathcal{N}^2$
- Pruning: Whenever the sum of the row or column is not correct:
  - sum of all values in the square is $\frac{1}{2}\mathcal{N}^2(\mathcal{N}^2+1)$
  - sum of all values in a row or column is $\frac{1}{2}\mathcal{N}(\mathcal{N}^2+1)$

Search pruning heuristics

Heuristic is a hint which tells us which order of actions is *likely* to produce quickly the solution.

- The effectivity of the solution is *not guaranteed*.
- Heuristics can be used to assess the order of vertices/edges/paths in which they are processed during the search in large graphs.

Example: Knight tour on an $\mathbb{N} \times \mathbb{N}$ chessboard (visit all fields).

- **Problem of:** finding a sequence of moves of a knight on a chessboard such that the knight visits every square exactly once.
- **Good heuristic:** Explore first those fields from which there are fewest possibilities of continuing the tour in different directions.
- **Speedup** on the 8×8 chessboard: Almost 100 000 times.

Homework Assignment — Drawing Binary Tree (PY)

Goal: Draw a binary tree in a **square grid** so that:

- Each node is a **circle** centered at a grid intersection point.
- Neighbouring nodes are connected by a straight line segment.
- The radius of each circle is less than 0.5.

Geometric placement rules:

- Nodes at the same **depth** share the same y coordinate.
- Deeper nodes have **smaller** y **values**.
- Nodes in the **left subtree** of a node have smaller x coordinates.
- Nodes in the **right subtree** have larger x coordinates.
- The drawing should **minimize the area** of the bounding rectangle.

Example

Examples of tree drawings in a grid:

- (a) 3×4, (b) 4×6, (c) 4×10 area rectangles.

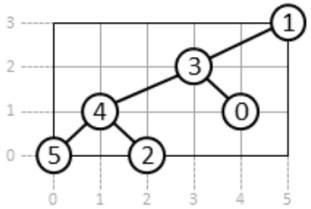
Homework Assignment — Drawing Binary Tree

Goal: Draw a binary tree in a **square grid** such that:

- Each node is a **circle** centered at a grid intersection.
- Neighbouring nodes are connected by a straight line segment.
- The radius of each circle is less than 0.5.

Geometric placement rules:

- Nodes at the same **depth** share the same y coordinate.
- Deeper nodes have **smaller** y **values**.
- Left subtree $\Rightarrow$ smaller x coordinate.
- Right subtree $\Rightarrow$ larger x coordinate.
- The **goal** is to **minimize** total rectangular area.



Homework Assignment — Drawing Binary Tree

Input format:

- First line: three integers `N R L`, where N is number of nodes ($0 \leq i < N$), R is label of the root node, and L is number of inspected nodes.
- Next $N-1$ lines: edges (u, v) of the tree.
- The edge to the **left child** appears before the edge to the **right child**.
- Last L lines: labels of inspected nodes.

Output format:

- L lines: for each inspected node u_i, output coordinates `x y`.

Task: Compute coordinates (x, y) for each inspected node so that all rules are met and the drawing area is minimal.

Hints:

- Remember the **in-order traversal** of a binary tree and its nice properties.
- Remember how to calculate depth of a tree node