

ALG 04

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

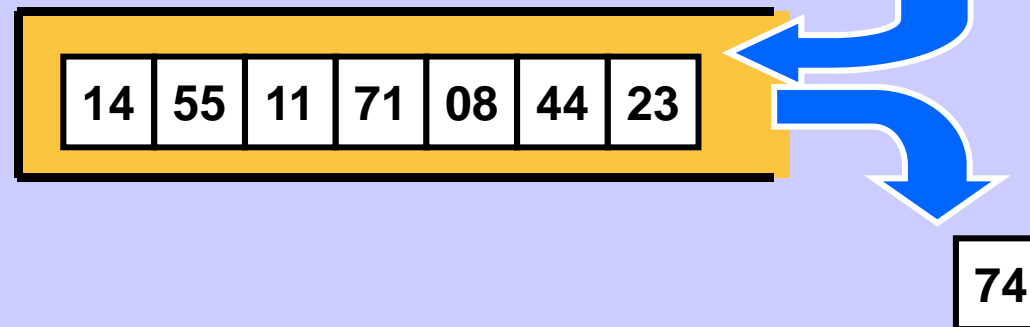
Search pruning

Stack

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

Stack bottom

Stack top



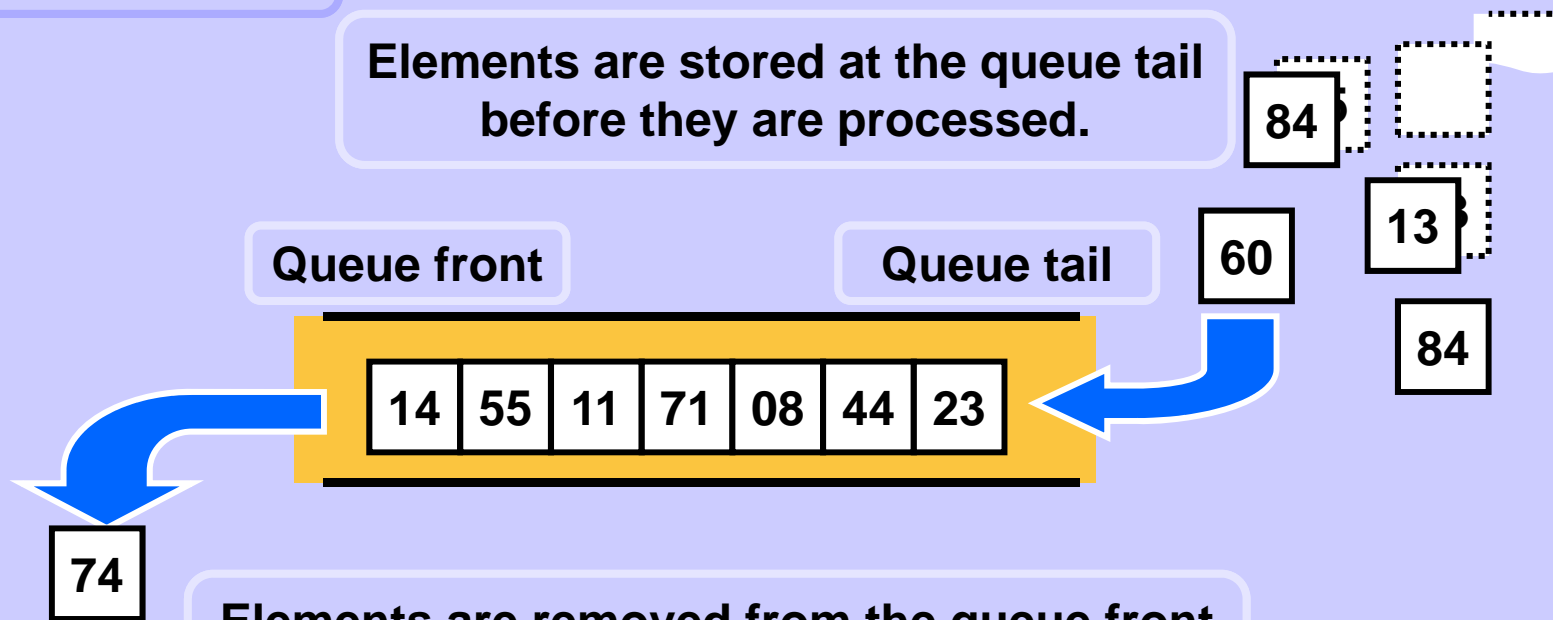
Elements are removed from the stack top and then they are processed.

Operation names

Put at the top	Push
Remove from the top	Pop
Read the top	Top
Is the stack empty?	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.

Operation names

Insert at the tail

Enqueue / InsertLast / Push ...

Remove from the front

Dequeue / delFront / Pop ...

Read the front elem

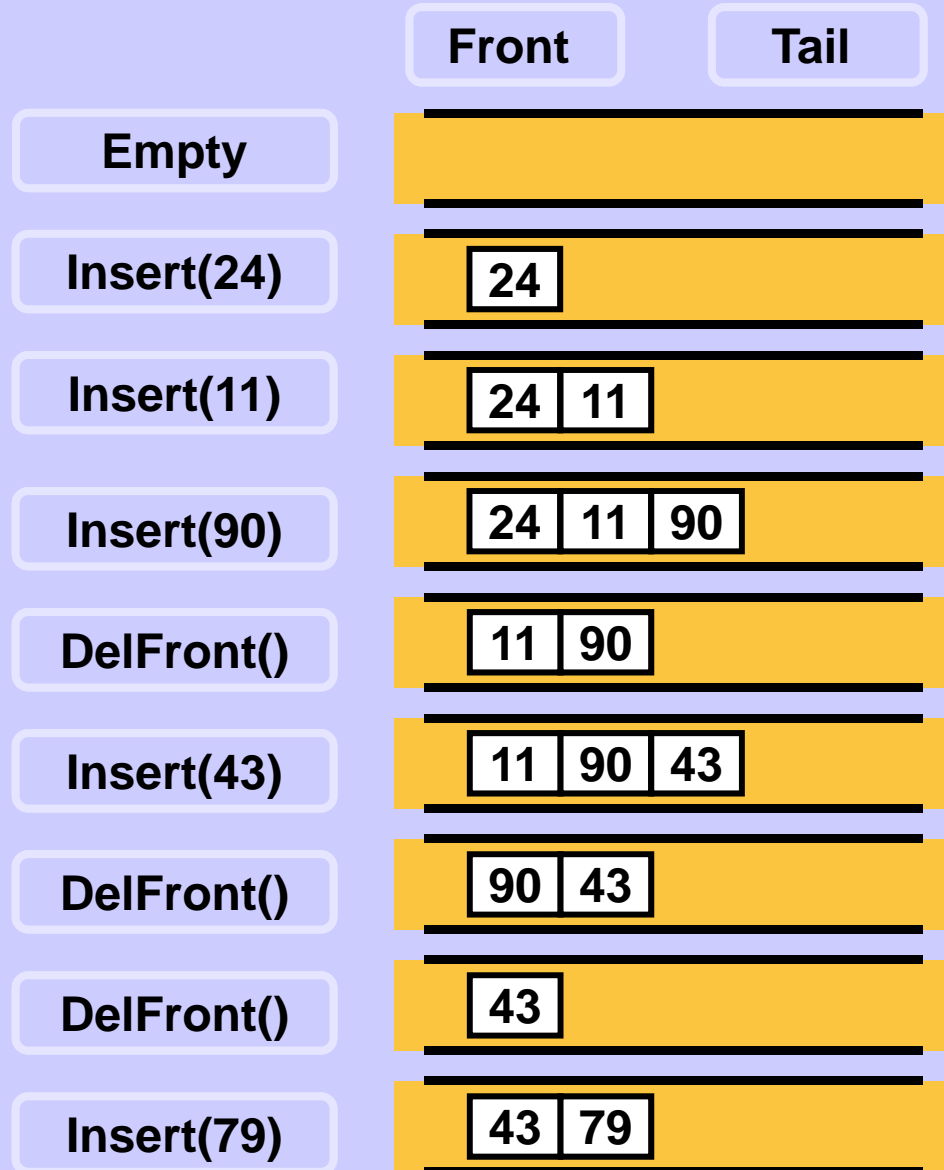
Front / Peek ...

Is the queue empty?

Empty

Queue

Easy example
of a queue
life cycle.



Cyclic queue implementation in an array

An empty queue in a fixed length array

Insert 24, 11, 90, 43, 70.

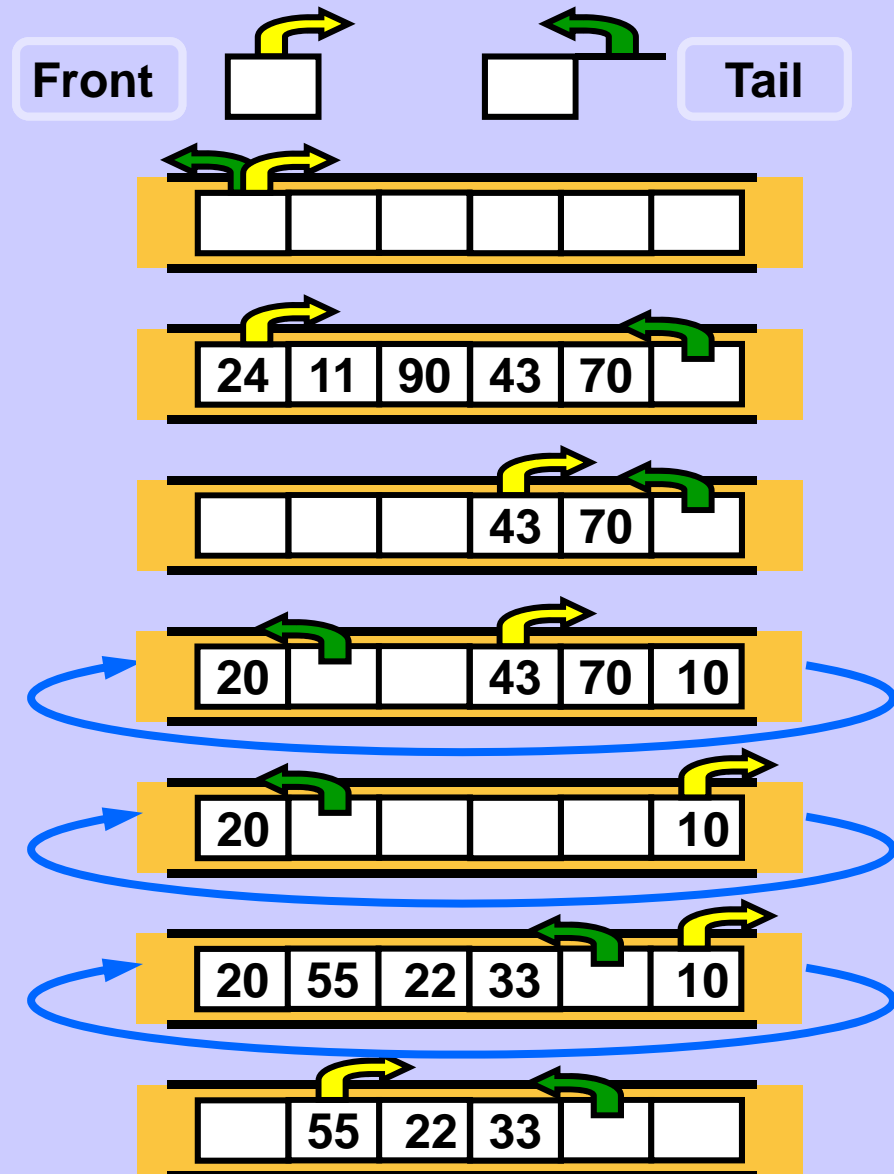
DelFront, DelFront, DelFront .

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

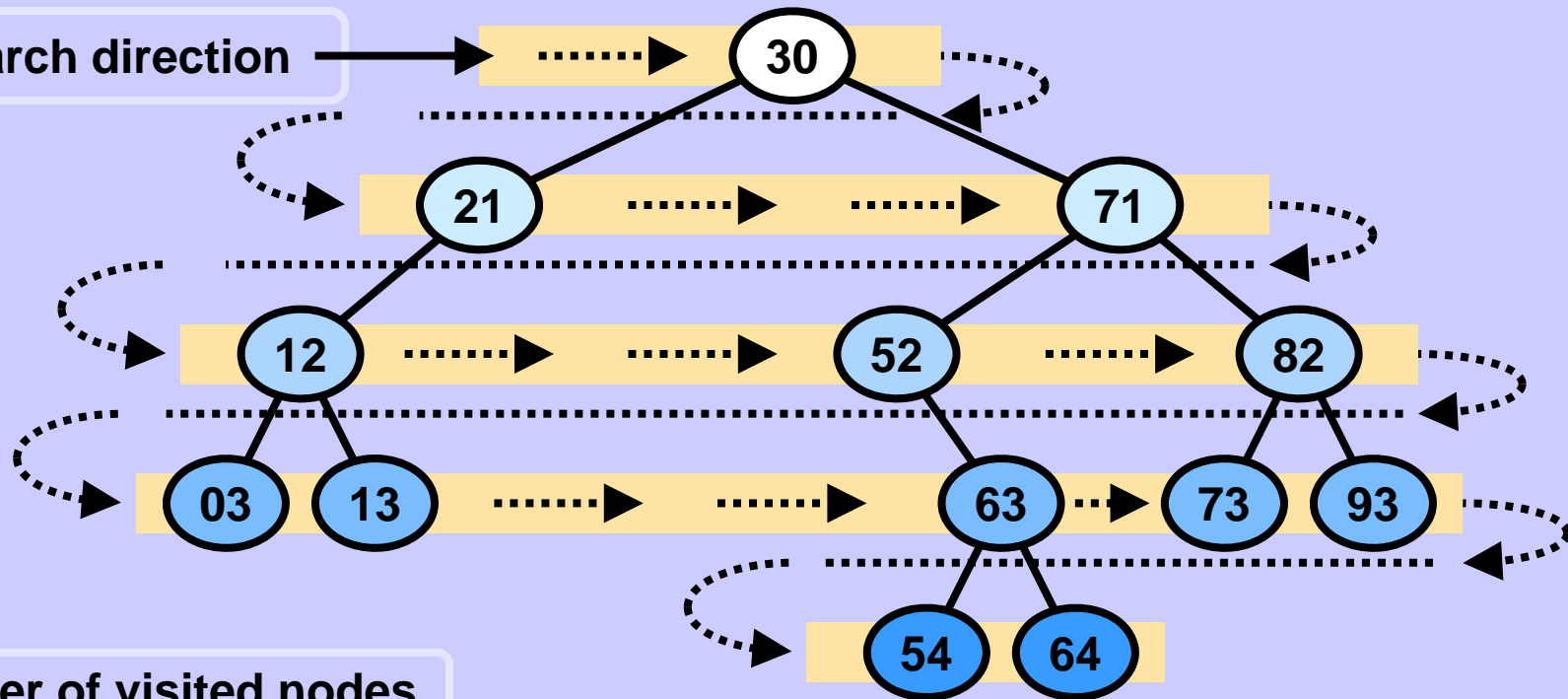
Tail index points to the first free position behind the last queue element.
Front index points to the first position occupied by a queue element.
When both indices point to the same position the queue is empty.

... 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

Search direction



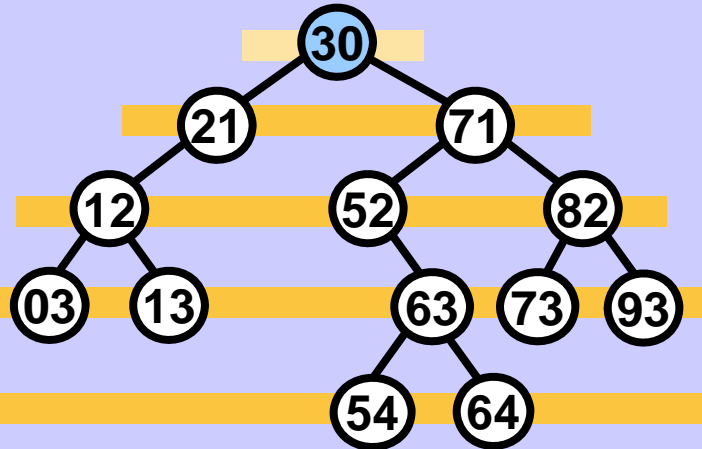
Order of visited nodes

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

Neither the tree structure nor the recursion support this approach directly.

Breadth-first search (BFS) in a tree

Initialization



Output

Create an empty queue.



Enqueue the tree root.



Front

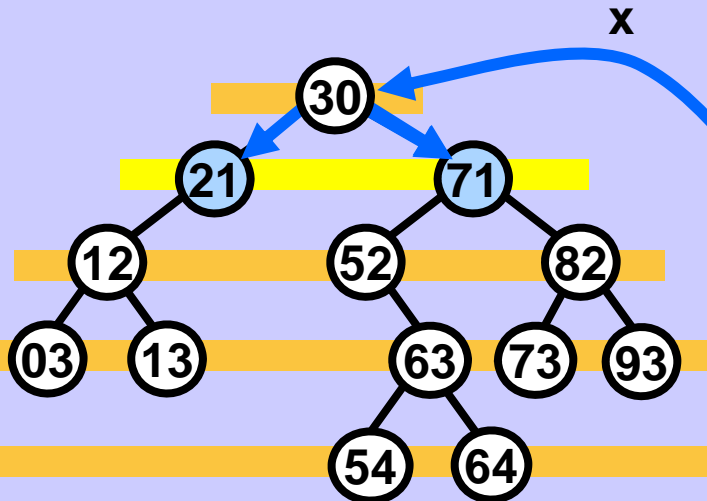
Tail

Main loop

While the queue is not empty do:

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

Breadth-first search (BFS) in a tree

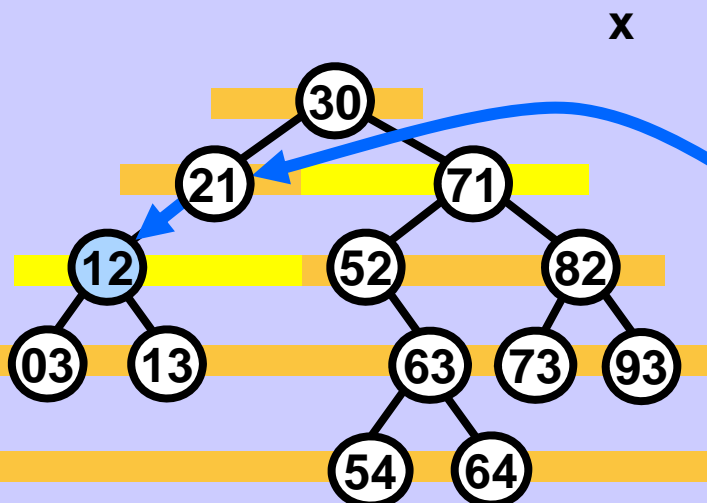
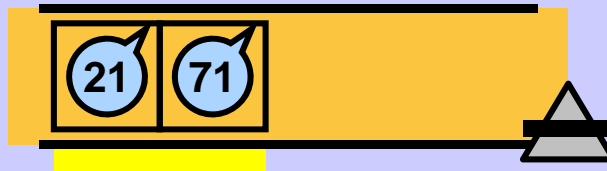


Output 30

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



2. `Enqueue(x.left), Enqueue(x.right). *`

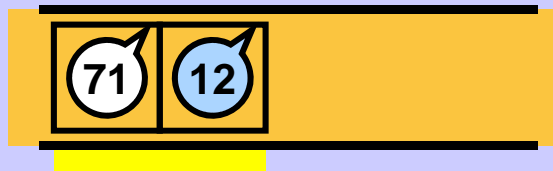


Output 30 21

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

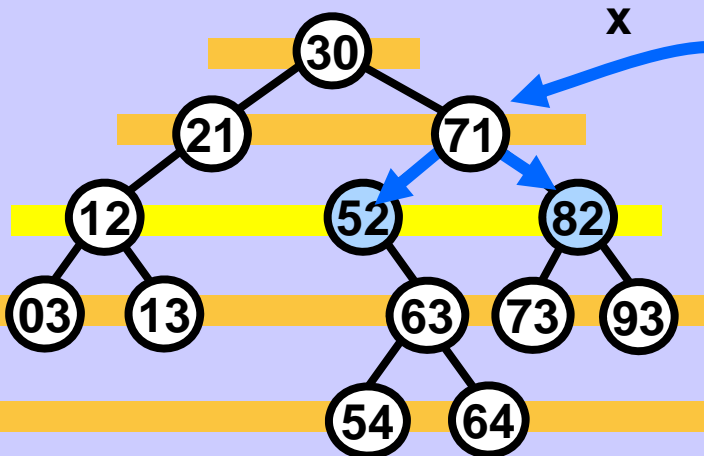


2. `Enqueue(x.left), Enqueue(x.right). *`



*) if exists

Breadth-first search (BFS) in a tree

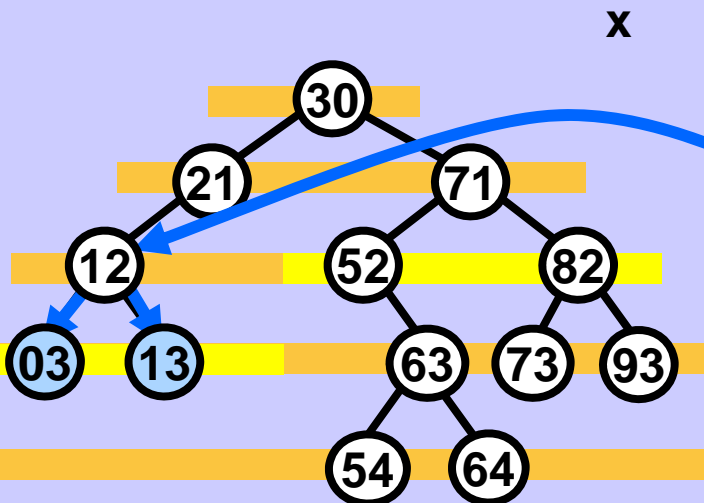
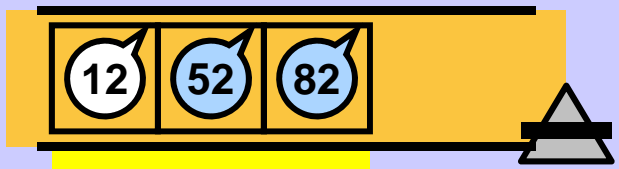


Output 30 21 71

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



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



Output 30 21 71 12

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

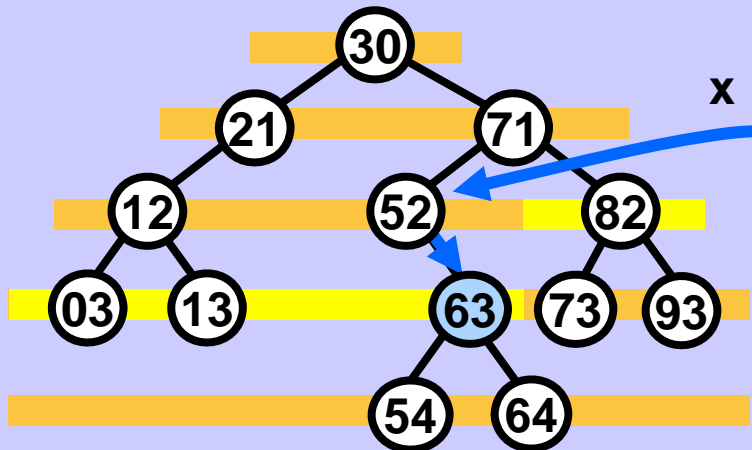


2. $\text{Enqueue}(x.\text{left}), \text{Enqueue}(x.\text{right}). *$



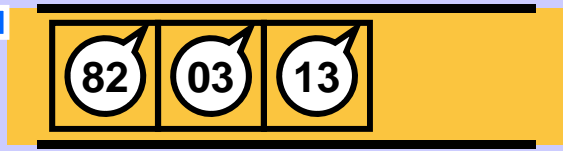
*) if exists

Breadth-first search (BFS) in a tree

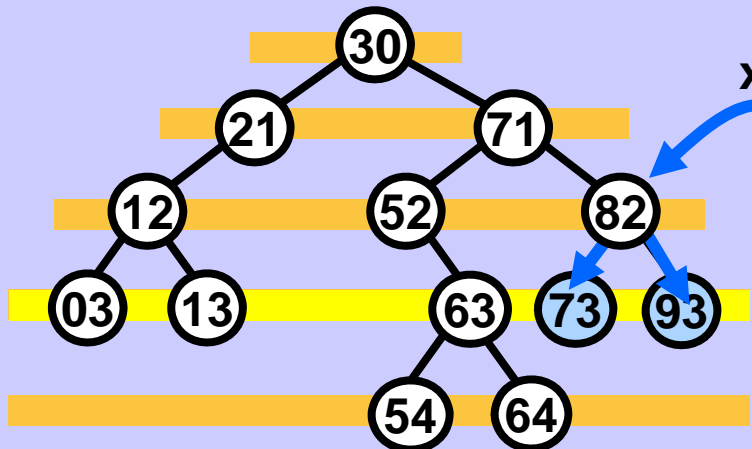


Output 30 21 71 12 52

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

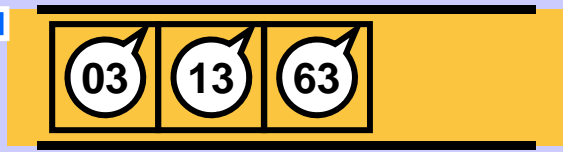


2. $\text{Enqueue}(x.\text{left}), \text{Enqueue}(x.\text{right}). *$

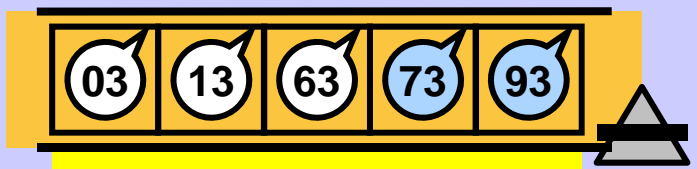


Output 30 21 71 12 52 82

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

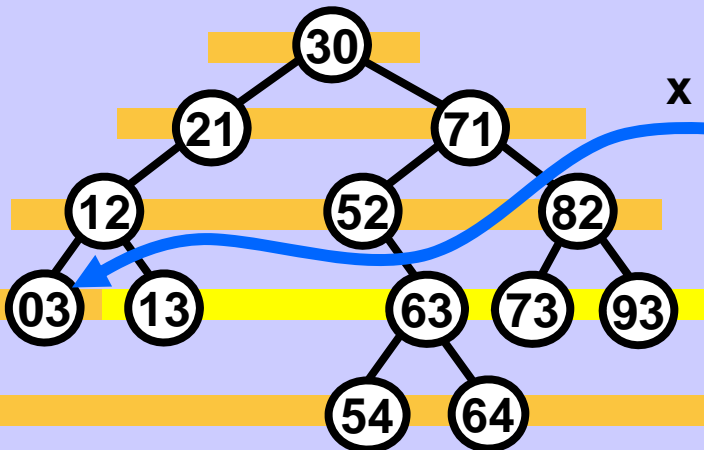


2. $\text{Enqueue}(x.\text{left}), \text{Enqueue}(x.\text{right}). *$



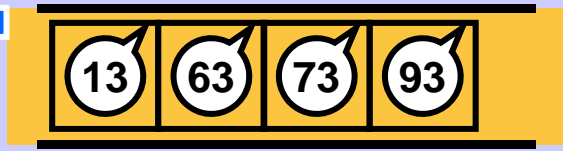
*) if exists

Breadth-first search (BFS) in a tree

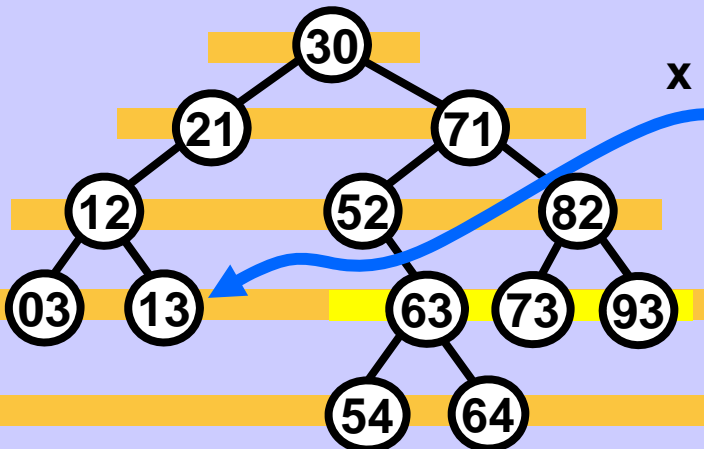


Output 30 21 71 12 52 82 03

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

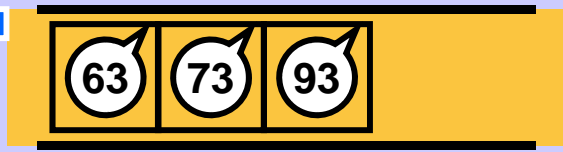


2. $\text{Enqueue}(x.\text{left}), \text{Enqueue}(x.\text{right}). *$



Output 30 21 71 12 52 82 03 13

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

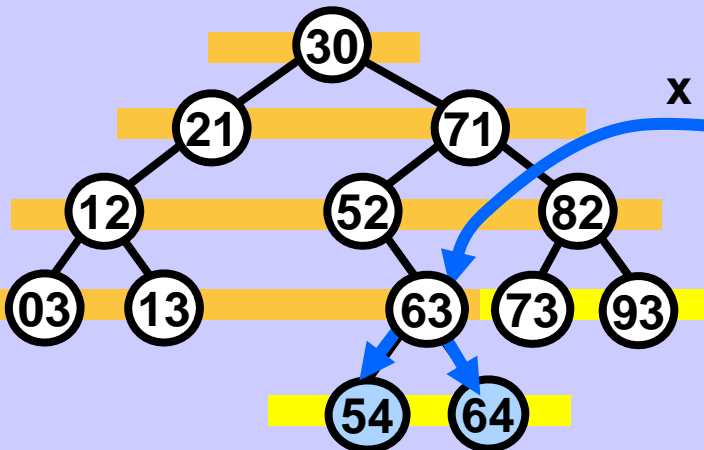


2. $\text{Enqueue}(x.\text{left}), \text{Enqueue}(x.\text{right}). *$



*) if exists

Breadth-first search (BFS) in a tree



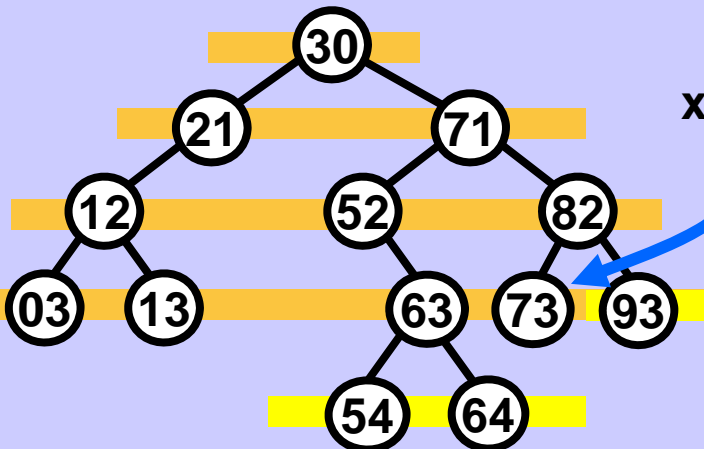
1. $x = \text{Dequeue}(), \text{print}(x.\text{key}).$



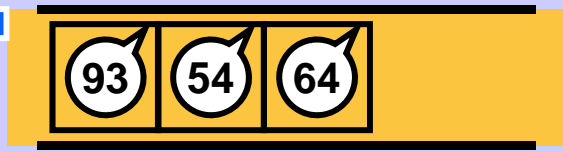
2. $\text{Enqueue}(x.\text{left}), \text{Enqueue}(x.\text{right}). *$



Output 30 21 71 12 52 82 03 13 63



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



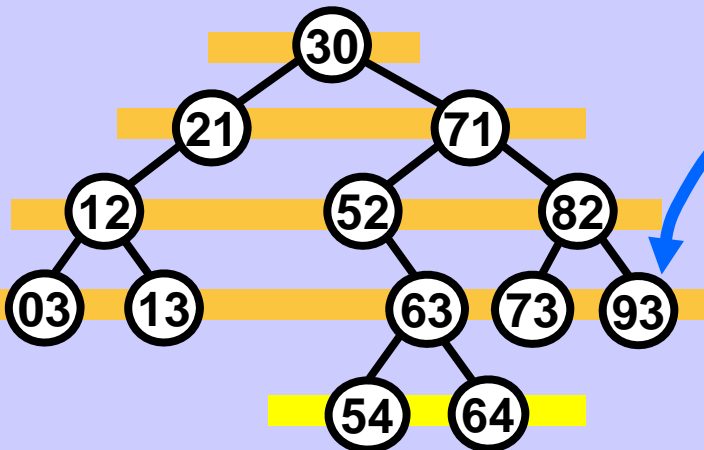
2. $\text{Enqueue}(x.\text{left}), \text{Enqueue}(x.\text{right}). *$



Output 30 21 71 12 52 82 03 13 63 73

*) if exists

Breadth-first search (BFS) in a tree



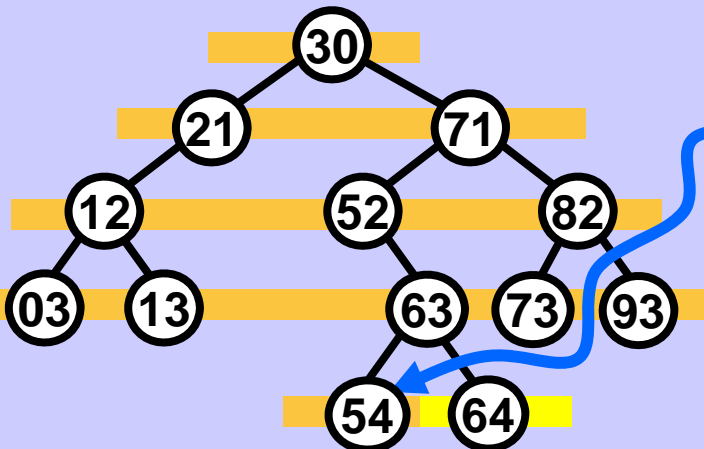
1. `x = Dequeue(), print (x.key).`



2. `Enqueue(x.left), Enqueue(x.right). *`



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



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



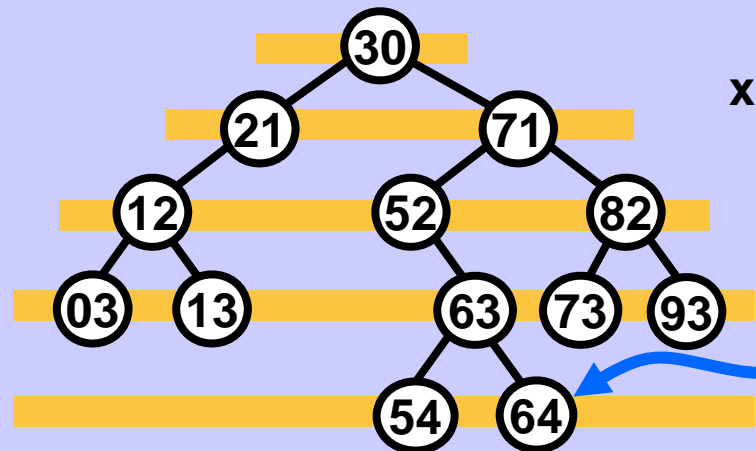
2. `Enqueue(x.left), Enqueue(x.right). *`



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

*) if exists

Breadth-first search (BFS) in a tree



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

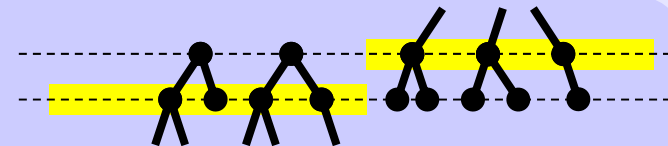
2. `Enqueue(x.left), Enqueue(x.right). *`

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

*) if exists.

The queue is empty,
BFS is complete.

An unempty **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. See above:



Breadth-first search (BFS) in a tree

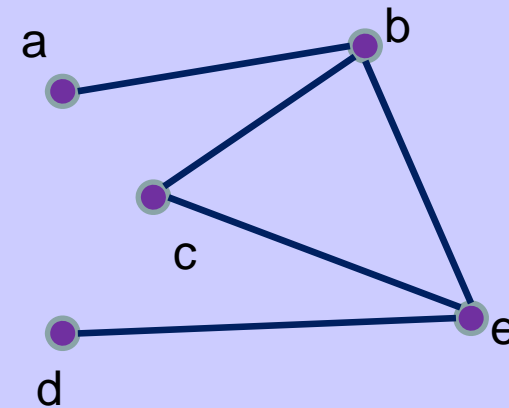
```
def binaryTreeBFS(node):  
    if node == None: return  
    q = Queue(100)                # init  
    q.Enqueue(node)               # root into queue  
    while (not q.isEmpty()):  
        node = q.Dequeue()  
        print(node.key, end = ' ') # process node  
        if node.left != None: q.Enqueue(node.left)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

Graphs

- Graph is an ordered pair of
- set of vertices (nodes) \mathcal{V} and set of pairs of vertices \mathcal{E} .

Each pair is an edge.

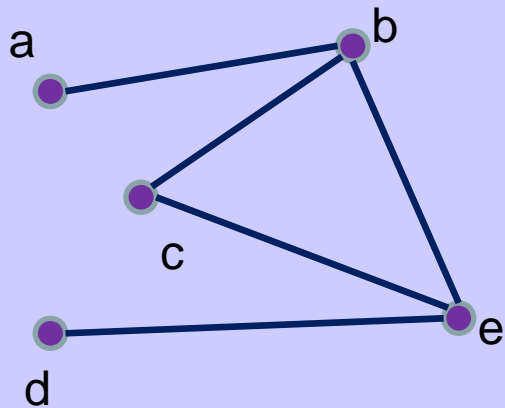
- $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
- Example:
 - $\mathcal{V} = \{a, b, c, d, e\}$
 - $\mathcal{E} = \{\{a,b\}, \{b,e\}, \{b,c\}, \{c,e\}, \{e,d\}\}$



Graphs - directed/undirected

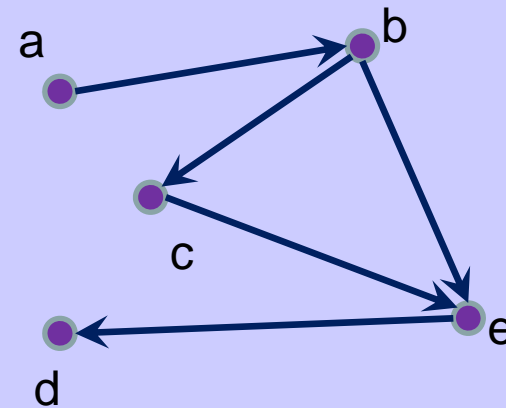
• Undirected graph

- 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 $G = (\mathcal{V}, \mathcal{E})$ be graph with n vertices
- Denote vertices v_1, \dots, v_n (in an arbitrary order)
- Adjacency matrix of G is a matrix of order n

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

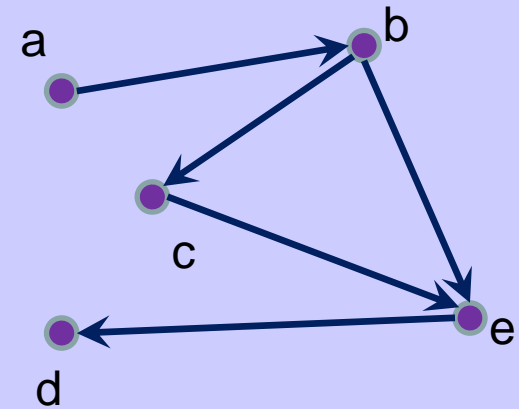
defined by the relation

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

Graph – adjacency matrix

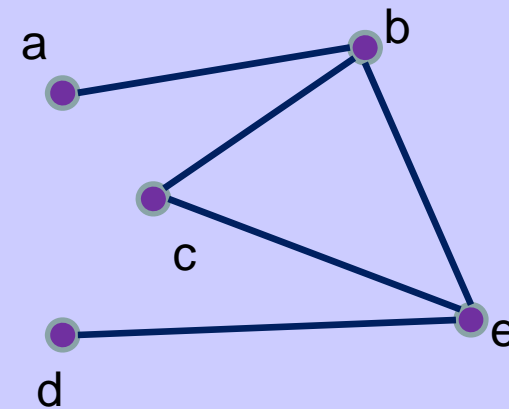
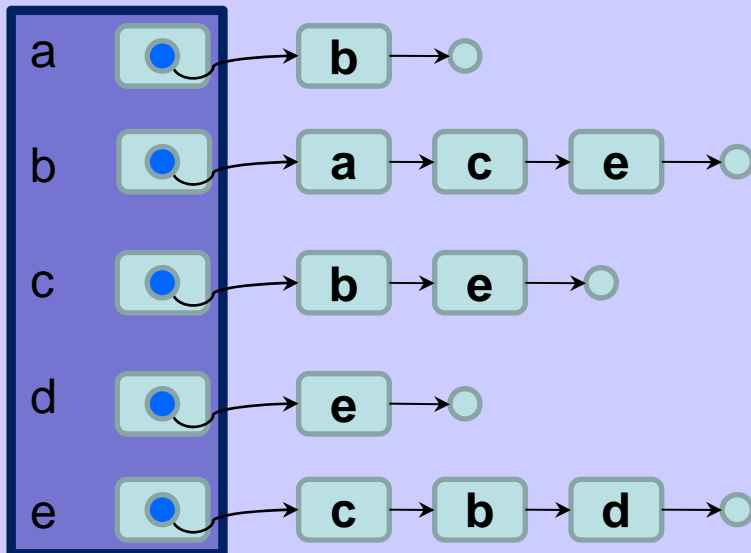
- Directed graph example

	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

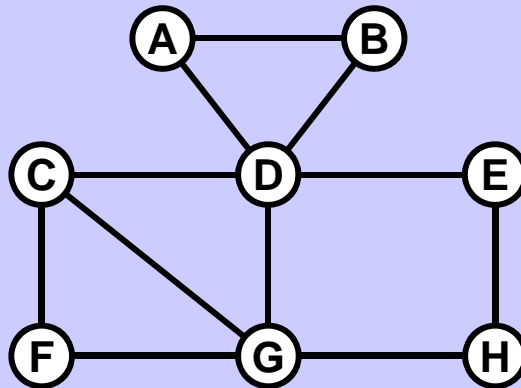


Graph – list of neighbours

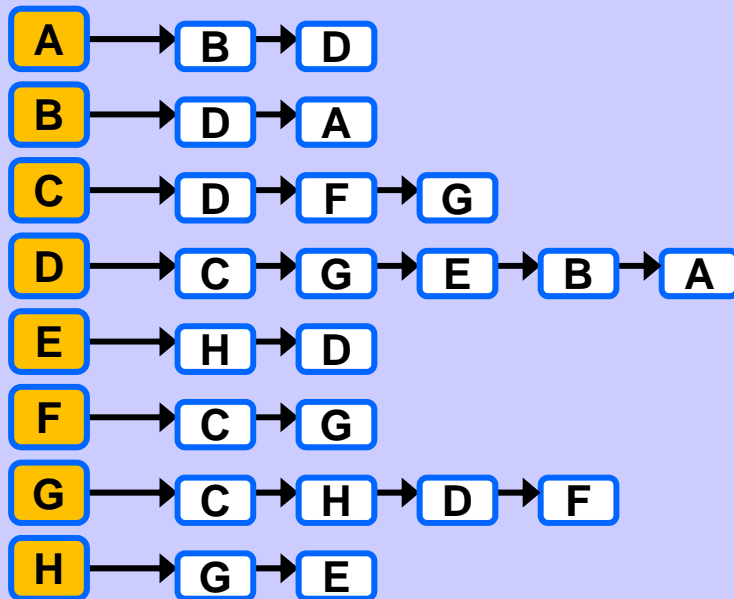
- Let $G = (\mathcal{V}, E)$ be an (un)directed graph with n vertices.
- Denote vertices v_1, \dots, v_n (in an arbitrary order).
- List of neighbours of G is an array \mathcal{P} of size n of pointers.
 - $\mathcal{P}[i]$ points to the list of all vertices which are adjacent to v_i .



Depth-first search (DFS) in a graph



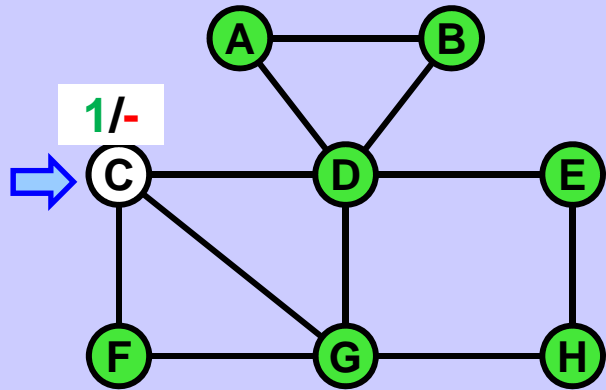
Linked list representation



Adjacency matrix

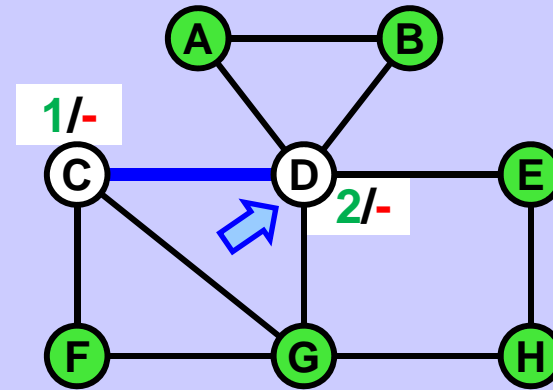
	A	B	C	D	E	F	G	H
A	0	1	0	1	0	0	0	0
B	1	0	0	1	0	0	0	0
C	0	0	0	1	0	1	1	0
D	1	1	1	0	1	0	1	0
E	0	0	0	1	0	0	0	1
F	0	0	1	0	0	0	1	0
G	0	0	1	1	0	1	0	1
H	0	0	0	0	1	0	1	0

Depth-first search (DFS) in a graph



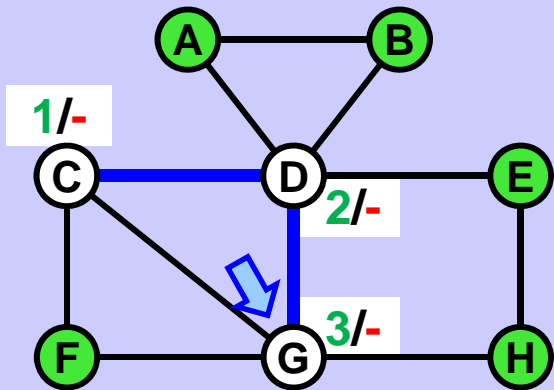
Stack C

Output C



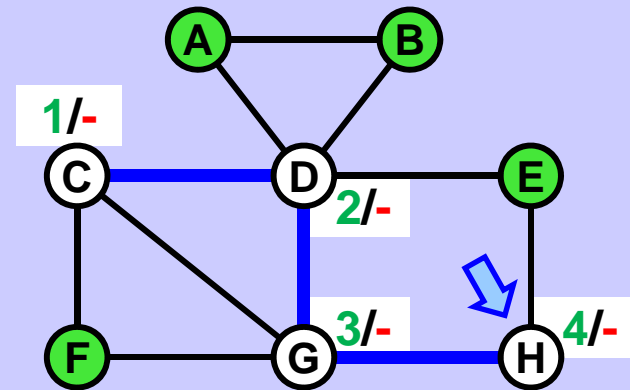
Stack C D

Output C D



Stack C D G

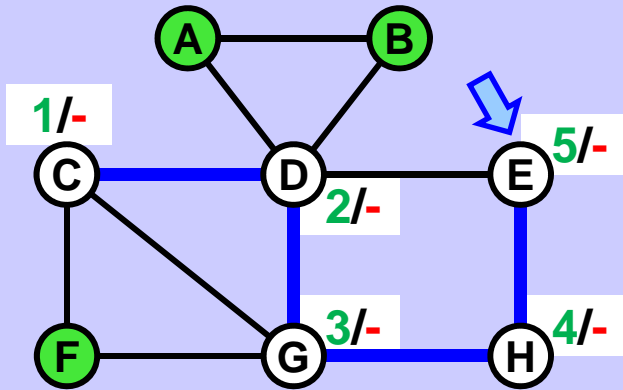
Output C D G



Stack C D G H

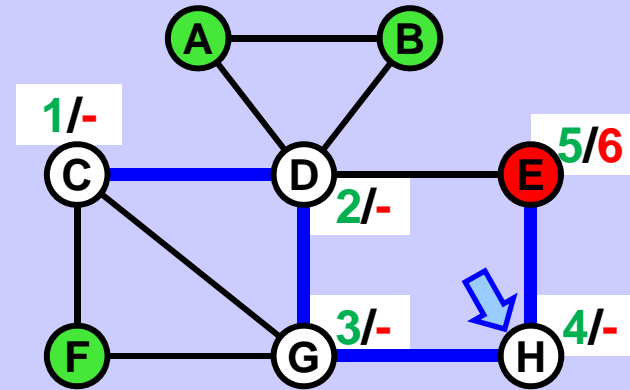
Output C D G H

Depth-first search (DFS) in a graph



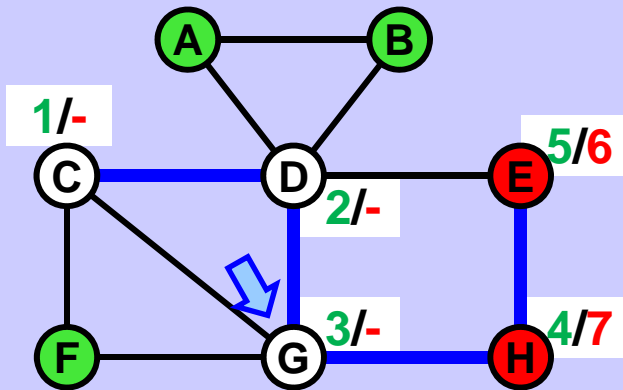
Stack: C D G H E

Output: C D G H E



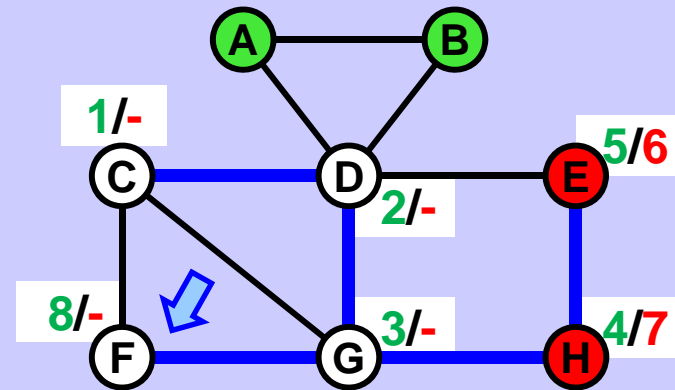
Stack: C D G H

Output: C D G H E



Stack: C D G

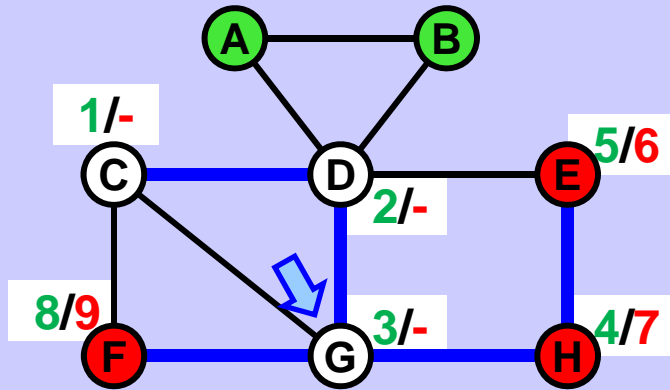
Output: C D G H E



Stack: C D G F

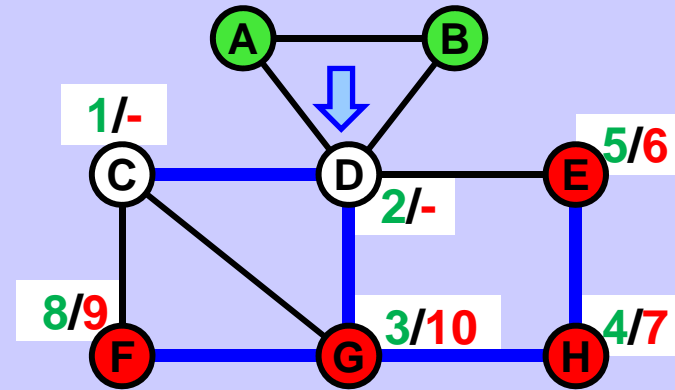
Output: C D G H E F

Depth-first search (DFS) in a graph



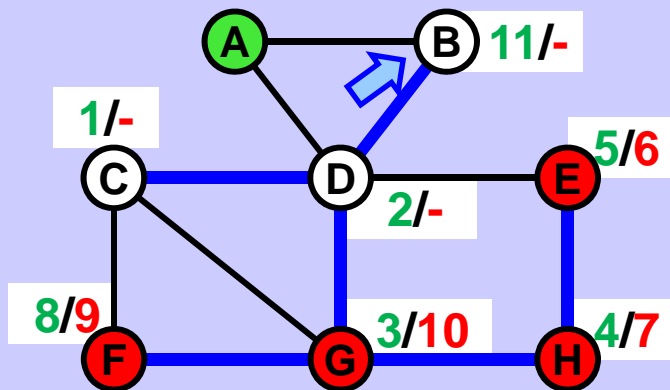
Stack C D G

Output C D G H E F



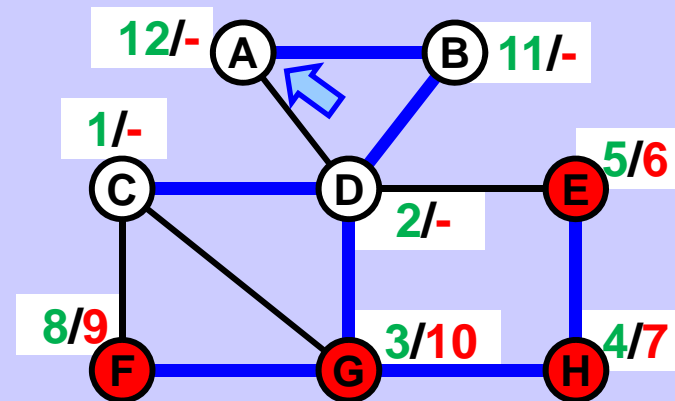
Stack C D

Output C D G H E F



Stack C D B

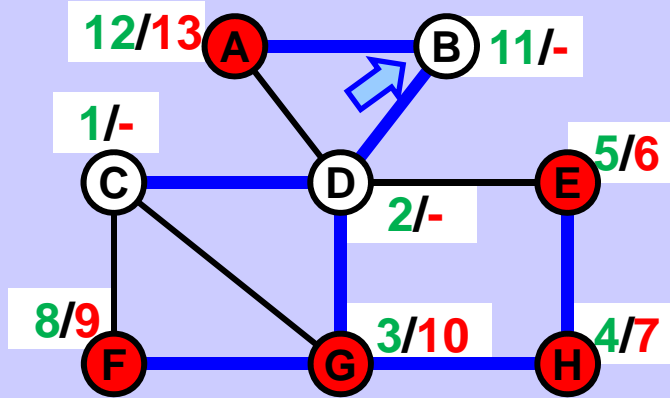
Output C D G H E F B



Stack C D B A

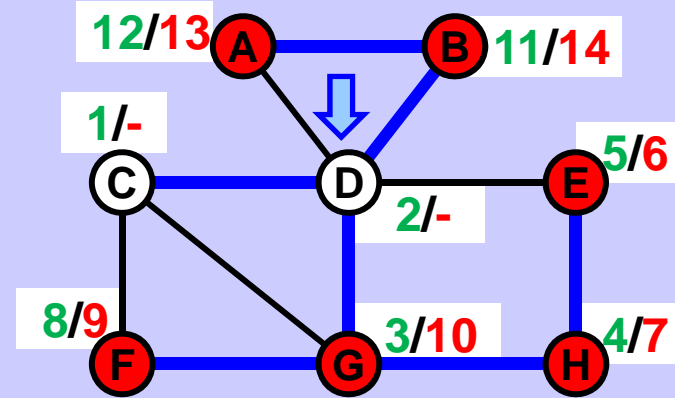
Output C D G H E F B A

Depth-first search (DFS) in a graph



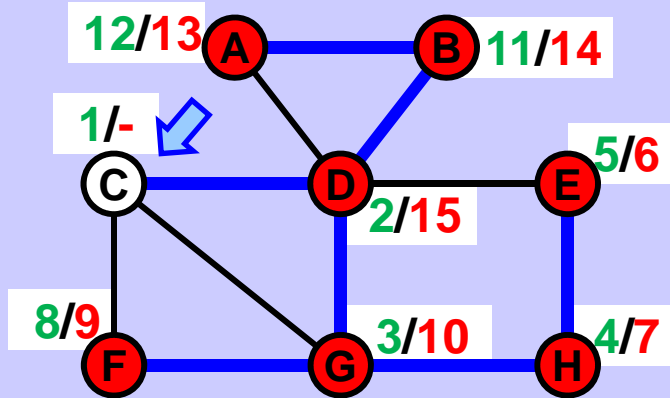
Stack C D B

Output C D G H E F B A



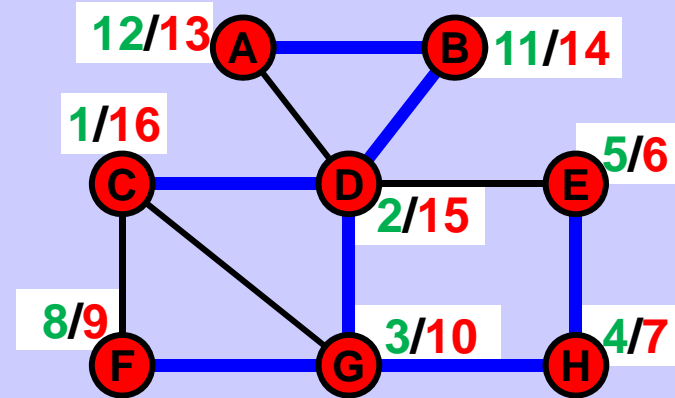
Stack C D

Output C D G H E F B A



Stack C

Output C D G H E F B A



Stack

Output C D G H E F B A

Depth-first search (DFS) in a graph

Life cycle of a node during the DFS Fresh - open - closed

Fresh

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 never gets bigger during the search.

Open

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

Closed nodes are those nodes which will not be visited any more.
When all neighbours of a node are either open or closed the node becomes closed.
The set of closed nodes never gets smaller during the search.
When the search terminates all nodes are closed.

Depth-first search (DFS) in a graph

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.

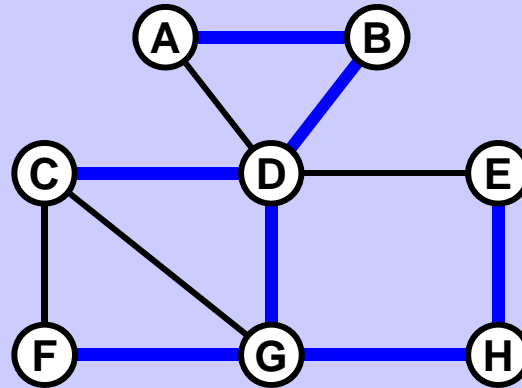
However, in an iterative DFS with a stack, it is necessary to register the state of each node -- fresh/open/closed.

In a recursive variant, the explicit information fresh/open/closed might be discarded. 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 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 the node becomes open results in the sequence

C D G H E F B A

Printing the node when the node becomes closed results in the sequence

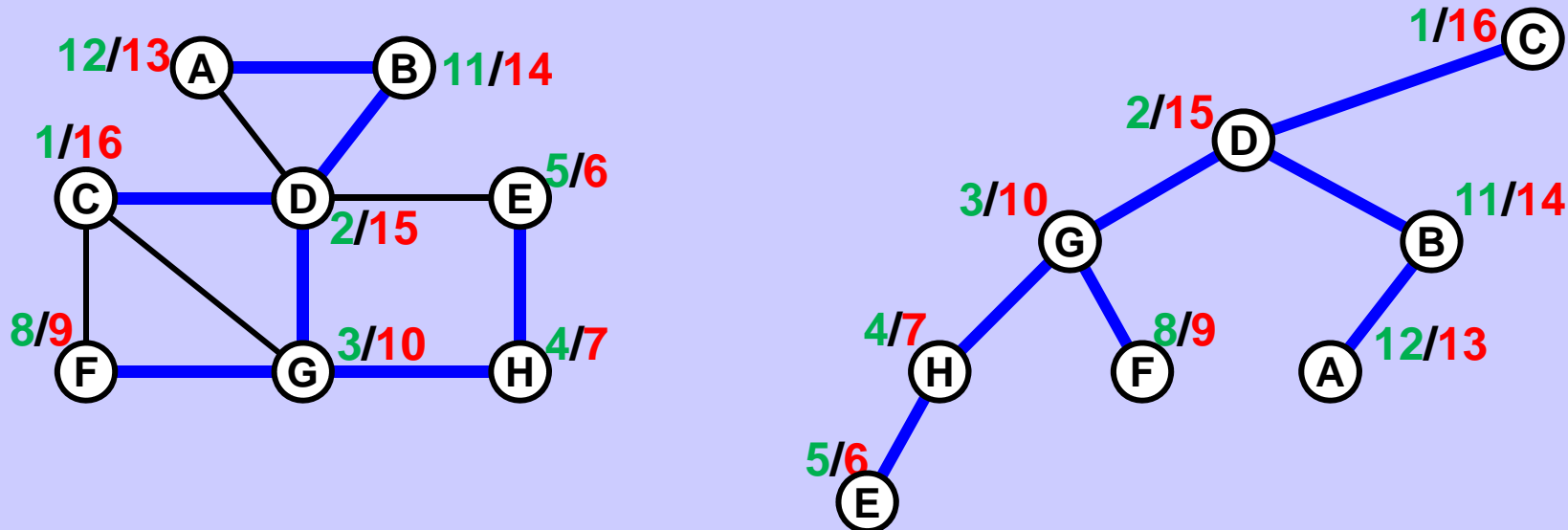
E H F G A B D C

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

- bridges and cutvertices 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



Note, that in a subtree with a root X it holds for each node $Y \neq X$:

$\text{Open_time}(X) < \text{Open_time}(Y) < \text{Close_time}(Y) < \text{Close_time}(X)$.

On the other hand, when Y is not a part of the subtree rooted in X then

$\text{Close_time}(X) < \text{Open_time}(Y)$ nebo $\text{Close_time}(Y) < \text{Open_time}(X)$

The number of nodes in the subtree rooted in X is always

$(\text{Close_time}(X) + 1 - \text{Open_time}(X)) / 2$.

Depth-first search (DFS) in a graph -- recursively C++

```
void DFS_rec_full( int start ) {
    vector<int> openT( N, 0 );
    vector<int> closeT( N, 0 );
    vector<int> pred( N, -1 ); // -1 == no predecessor
    int time = 0;
    DFS_rec_full( start, time, openT, closeT, pred );

    for( int n = 0; n < N; n++ )
        cout << "node " << n << " open/close "
             << openT[n] << "/" << closeT[n]
             << " pred " << pred[n] << endl;
}
```


Depth-first search (DFS) in a graph -- recursively C++

```
void DFS_rec_full( int currNode, int & time,
                  vector<int> & openT,
                  vector<int> & closeT,
                  vector<int> & pred ){

    int neigh;
    openT[currNode] = ++time;
    for( int j = 0; j < edge[currNode].size(); j++ ){
        neigh = edge[currNode][j];
        if( openT[neigh] == 0 ) {
            pred[neigh] = currNode;
            DFS_rec_full( neigh, time, openT, closeT, pred );
            cout << currNode << " --> " << neigh << endl;
        }
    }
    closeT[currNode] = ++time;
}
```

Depth-first search (DFS) in a graph -- recursively, basic variant C++

```
void DFS_rec_plain( int currNode, vector<bool> & fresh ){
    int neigh;
    fresh[currNode] = false;
    for( int j = 0; j < edge[currNode].size(); j++ ){
        neigh = edge[currNode][j];
        if( fresh[neigh] ) {
            DFS_rec_plain( neigh, fresh );
            cout << currNode << " --> " << neigh << endl;
        }
    }
}

void DFS_rec_plain( int start) {
    vector<bool> fresh( N, true );
    DFS_rec_plain(start, fresh);
}
```

Breadth-first search (BFS) in a graph

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

Fresh

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 never gets bigger during the search.

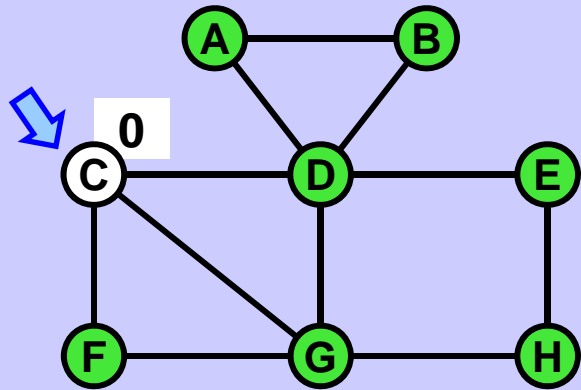
Open

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

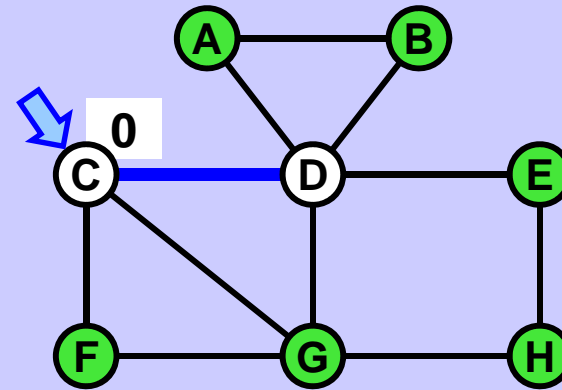
Closed nodes are those nodes which will not be visited any more. When all neighbours of a node are either open or closed the node becomes closed. The set of closed nodes never gets smaller during the search. When the search terminates all nodes are closed.

Breadth-first search (BFS) in a graph



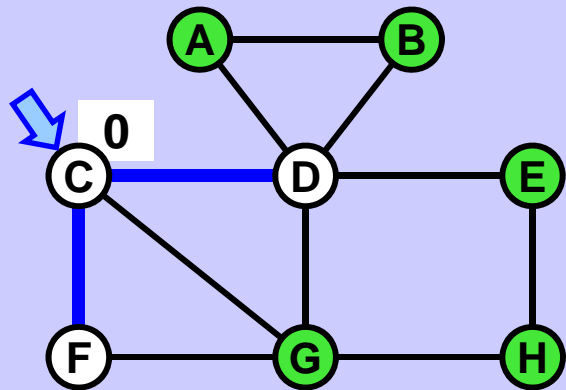
Queue C

Output C



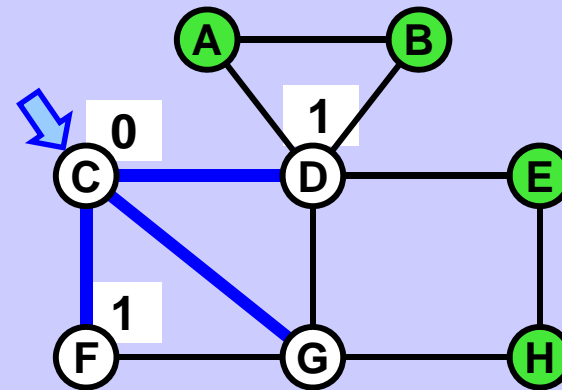
Queue D

Output C



Queue D F

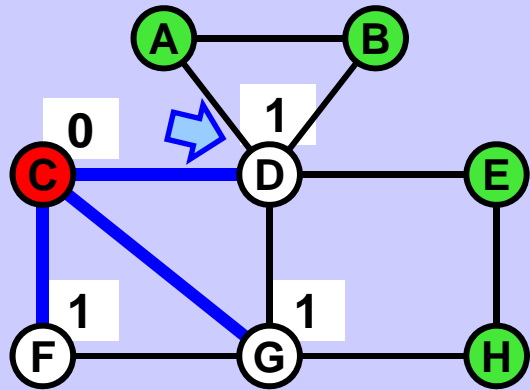
Output C



Queue D F G

Output C

Breadth-first search (BFS) in a graph

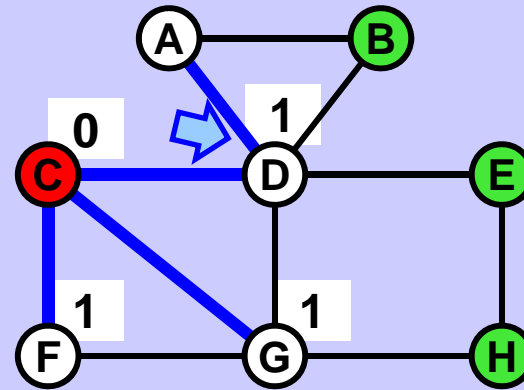


Queue

D F G

Output

C D

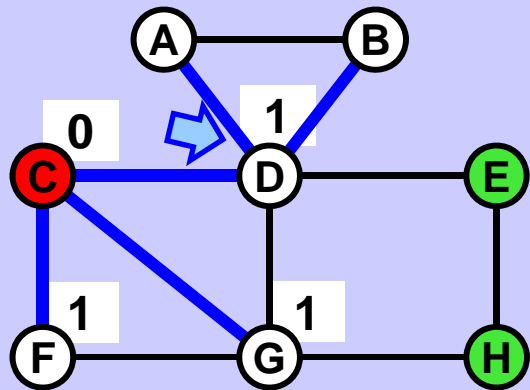


Queue

F G A

Output

C D

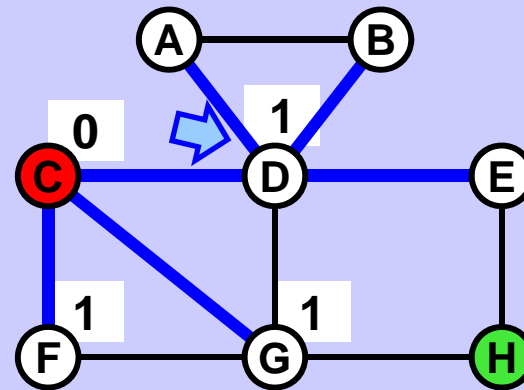


Queue

F G A B

Output

C D



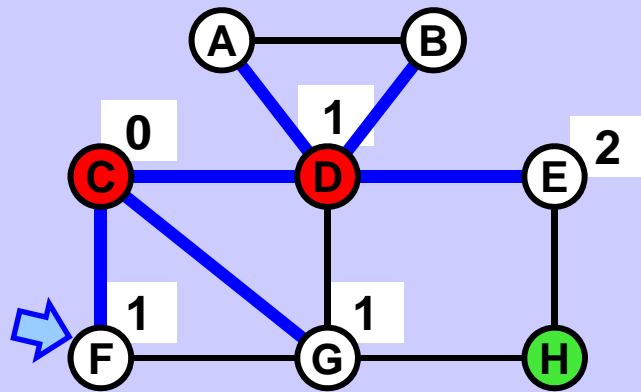
Queue

F G A B E

Output

C D

Breadth-first search (BFS) in a graph

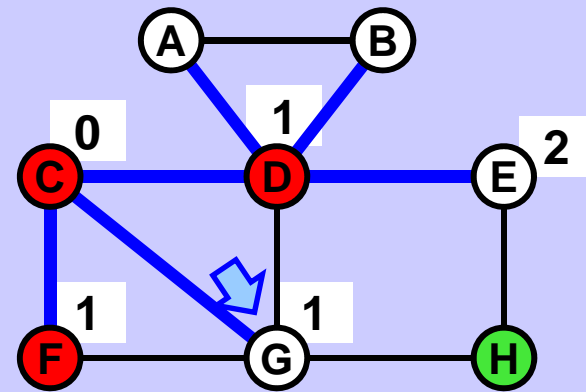


Queue

F G A B E

Output

C D F

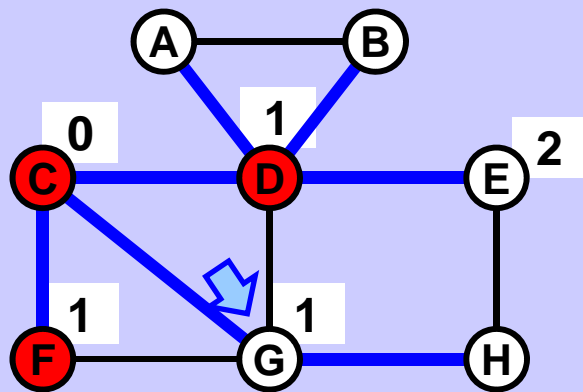


Queue

G A B E

Output

C D F G

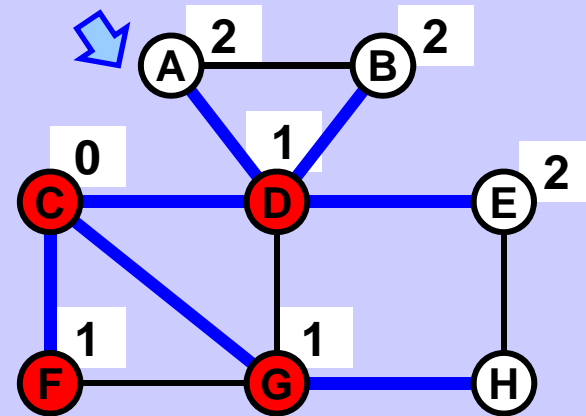


Queue

A B E H

Output

C D F G



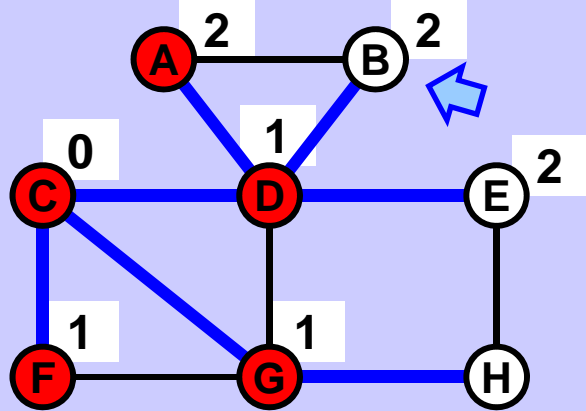
Queue

A B E H

Output

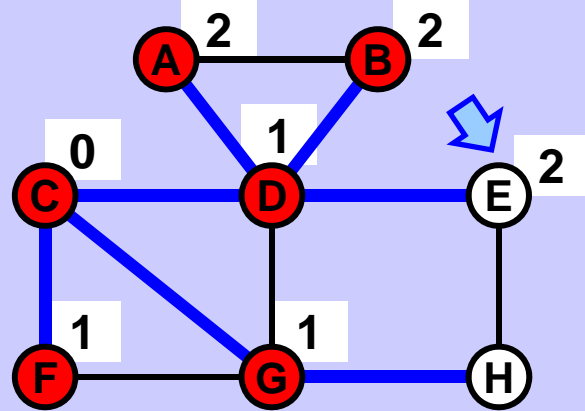
C D F G A

Breadth-first search (BFS) in a graph



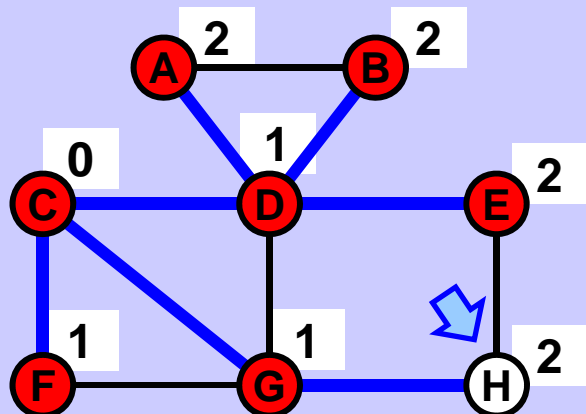
Queue B E H

Output C D F G A B



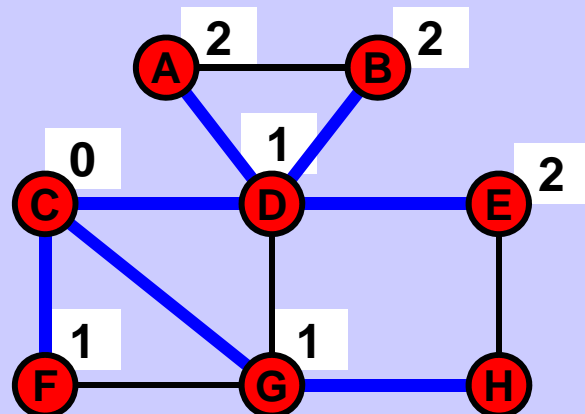
Queue E H

Output C D F G A B E



Queue H

Output C D F G A B E H

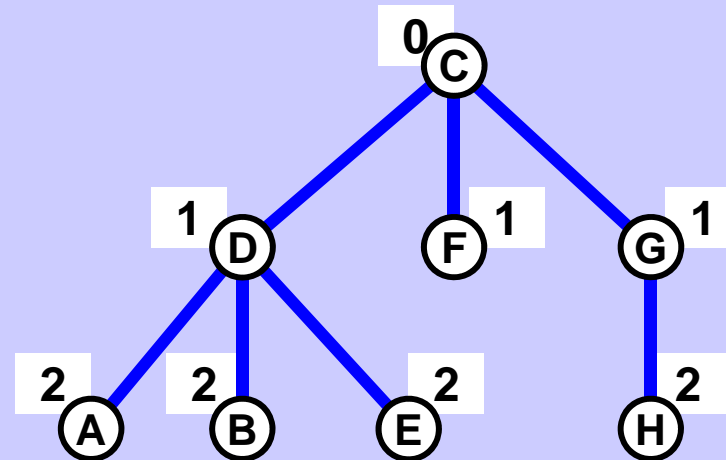
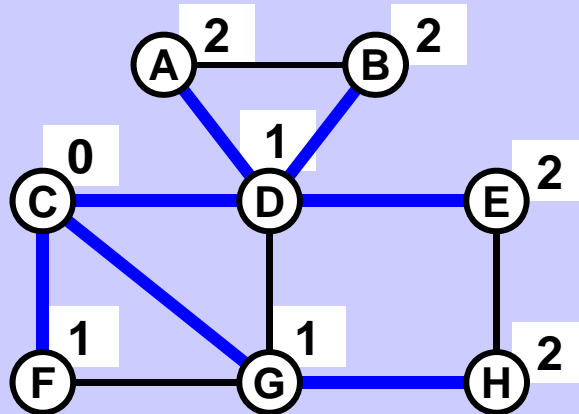


Queue

Output C D F G A B E H

Breadth-first search (BFS) in a graph

BFS-tree
with distances to the start node (root)
of all nodes



The open and close times are not essential in BFS.

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

BFS algorithm is exploited in e.g.:

Testing of graph connectivity, testing existence of a cycle in a graph, testing if a graph is bipartite, etc.

Typically BFS is used to compute distance(s) from a given node to either one other node or to 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

Pseudocode

```
void graphBreadthFirstSearch( Node startNode ) {  
    Set visited = new Set();           // visited == not fresh  
    Queue q = new Queue();  
    q.Enqueue( startNode );  
    visited.add( startNode );  
    while( !q.Empty() ) {  
        node = q.Dequeue();  
        print( node.key );           // process node  
        forall x in node.Neighbors()  
            if( x not in visited ){  
                visited.add( x );  
                q.Enqueue( x );  
            }  
    }  
}
```

Breadth-first search (BFS) in a graph C++

```
void BFS( int start ){
    queue<int> q;
    vector<int> dist( N, -1 ); // -1 ... fresh node
    vector<int> pred( N, -1 ); // -1 ... no predecessor (yet)

    q.push( start );
    dist[start] = 0;

    int currn, neigh;
    while( !q.empty() ){
        currn = q.front(); q.pop();
        cout << "node " << currn << endl; // process the node
        for( int j = 0; j < edge[currn].size(); j++ ){
            neigh = edge[currn][j];
            if( dist[neigh] == -1 ){ // neigh is fresh
                q.push( neigh );
                dist[neigh] = dist[currn] + 1;
                pred[neigh] = currn;
                cout << "BFS edge: " << currn << "->" << neigh << endl;
            }
        }
    }
}
```

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

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 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

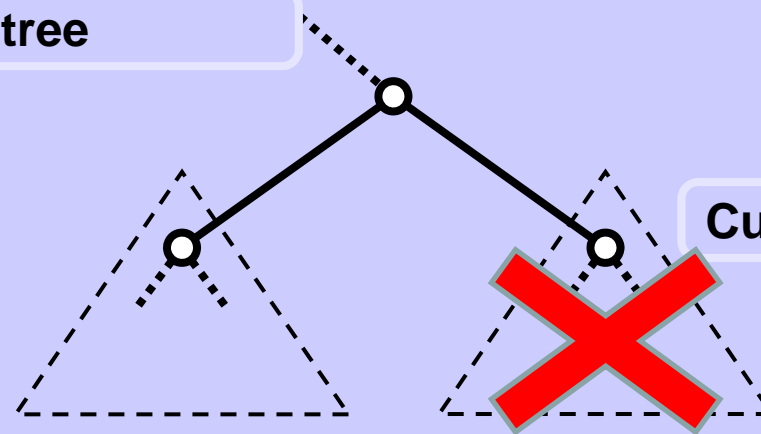
In total

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

Search pruning

- Search speedup
- Pruning (skipping) of unpromising possibilities
- When the analyse 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

Search tree



Current state

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 $\mathcal{N} \times \mathcal{N}$ chessboard (visit all fields).
- 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**.