ALG 04

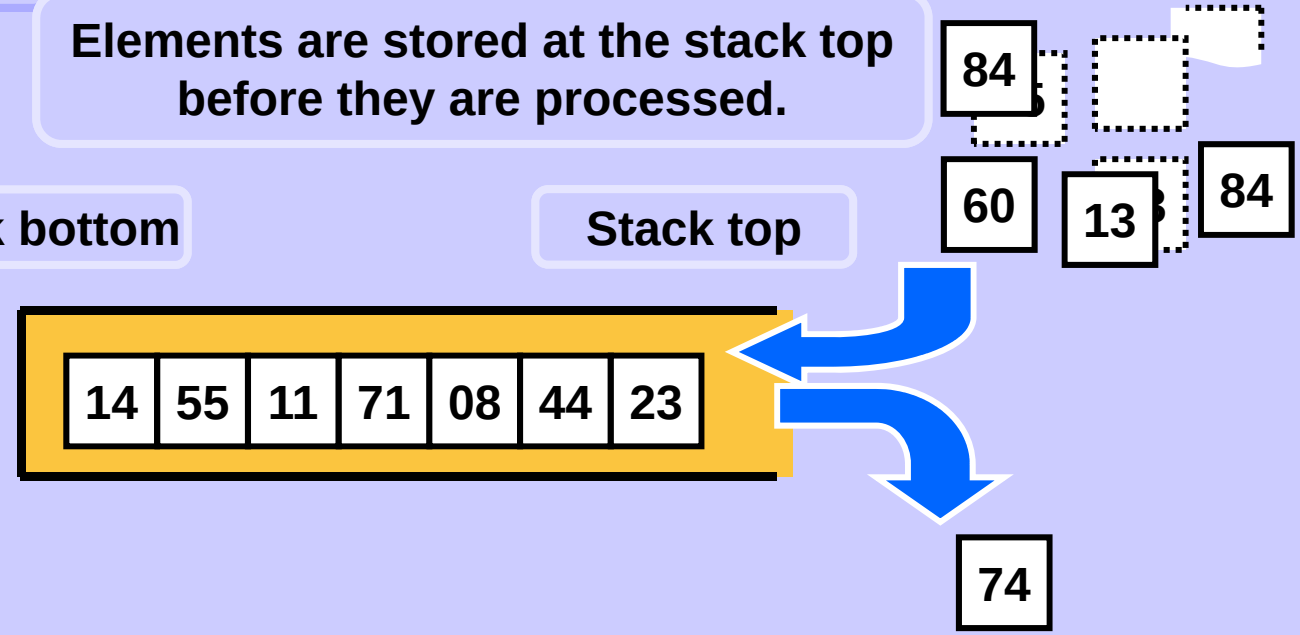
Queue
- Operations Enqueue, Dequeue, Front, Empty....
- Cyklic queue implementation

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.



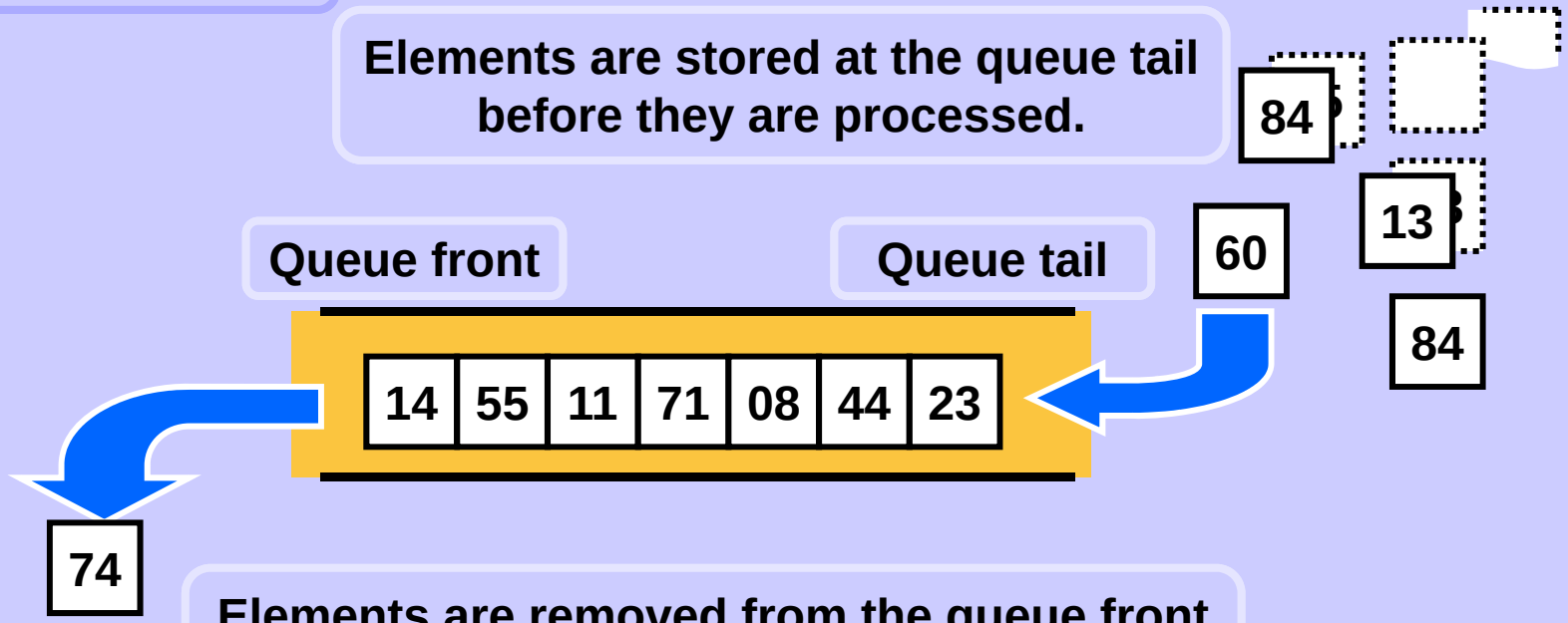
Elements are removed from the stack top and then they are processed.

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.



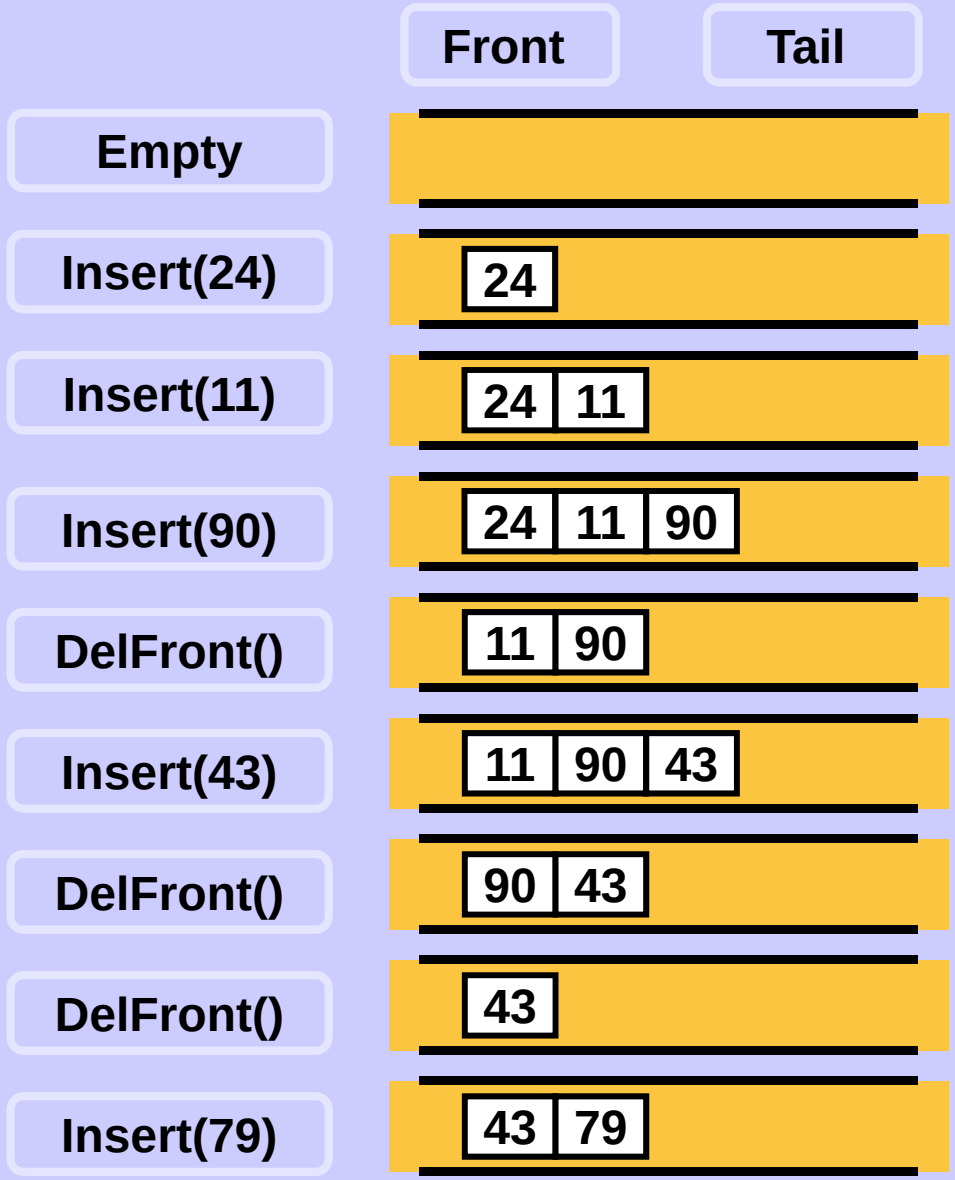
Elements are removed from the queue front and then they are processed.

Operation names

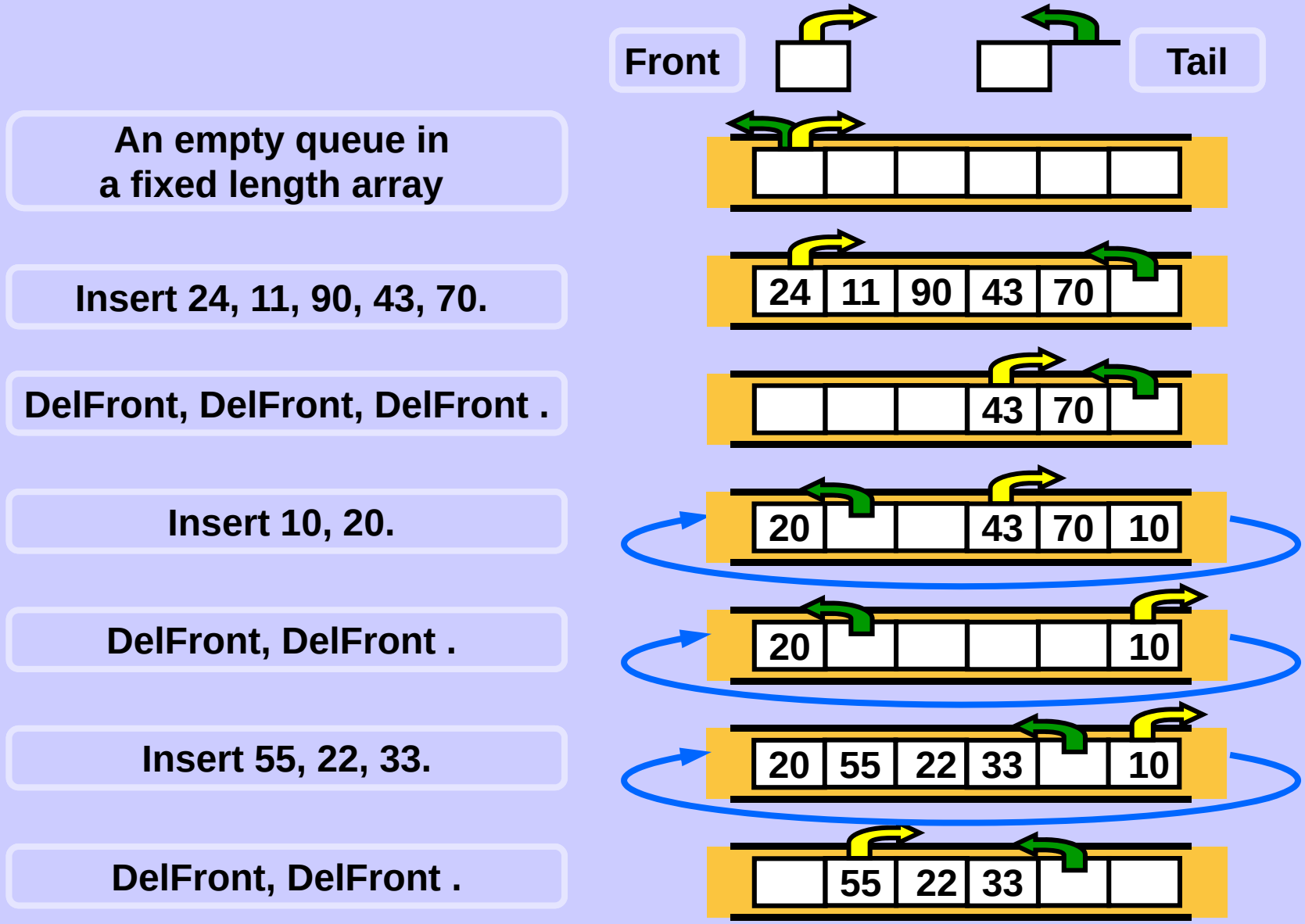
Insert at the tail	Enqueue / InsertLast / Push ...
Remove from the front	Dequeue / delFront / Pop ...
Read the front elem	Front / Peek ...
Is the queue empty?	Empty

Queue

Easy example of a queue life cycle.



Cyclic queue implementation in an array



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 {
  Node q [];
  int size;
  int front;
  int tail;
           //constructor:
 Queue(int qsize) {
   size = qsize;
   q = new Node[size];
   front = 0;
   tail = 0;
   }

 boolean Empty() {
   return (tail==front);
 }
```

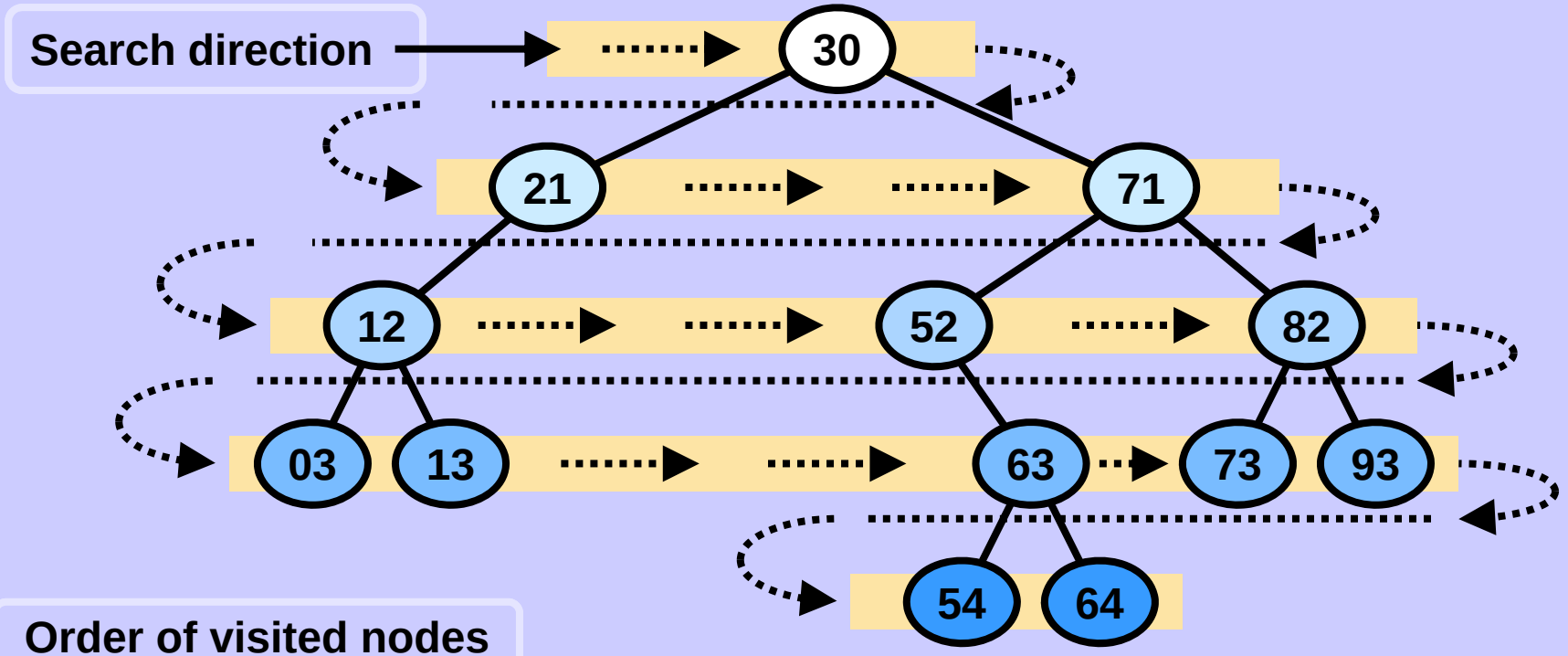
```
void Enqueue(Node node) {
  if ((tail+1 == front) ||
      (tail-front == size-1))
      ... // queue full, fix it

  q[tail++] = node;
  if (tail==size) tail=0;
}

Node Dequeue() {
  Node n = q[front++];
  if (front==size) front=0;
  return n;
}

} // end of Queue
```

Breadth-first search (BFS) of a tree



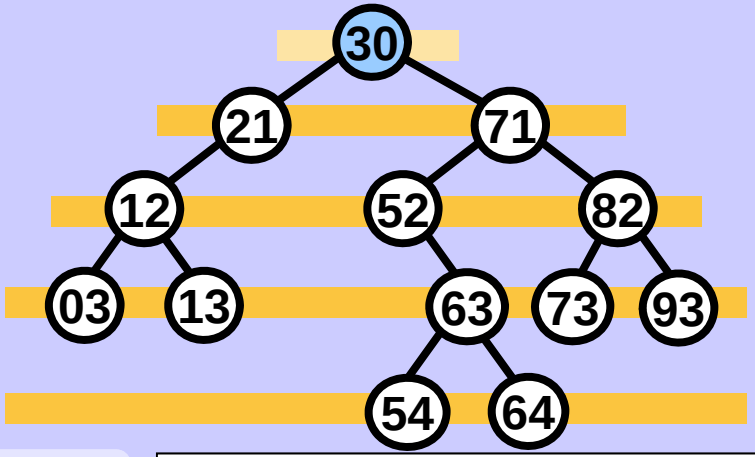
Order of visited nodes

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

Nor the tree structure nor the recursion support this approach directly.

Breadth-first search (BFS) of a tree

Initialization



Output

Create an empty queue.

Enqueue the tree root.

Front

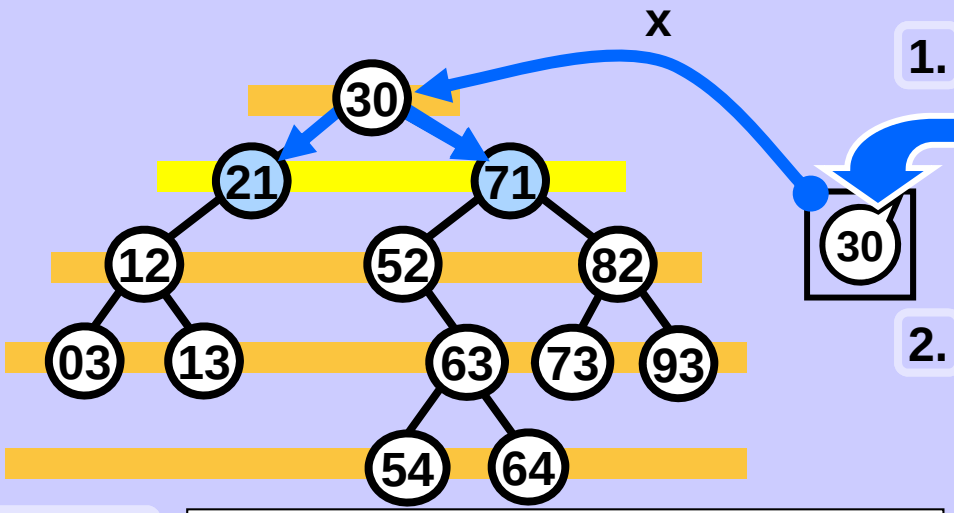
Tail

Main loop

While the queue is not empty do:

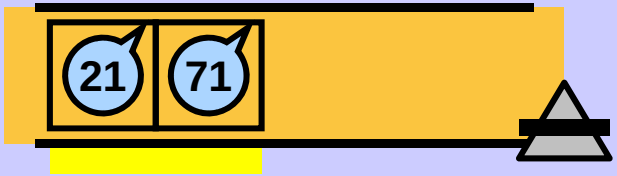
1. Remove the first element from the queue and process it.
2. Enqueue the children of removed element.

Breadth-first search (BFS) of a tree

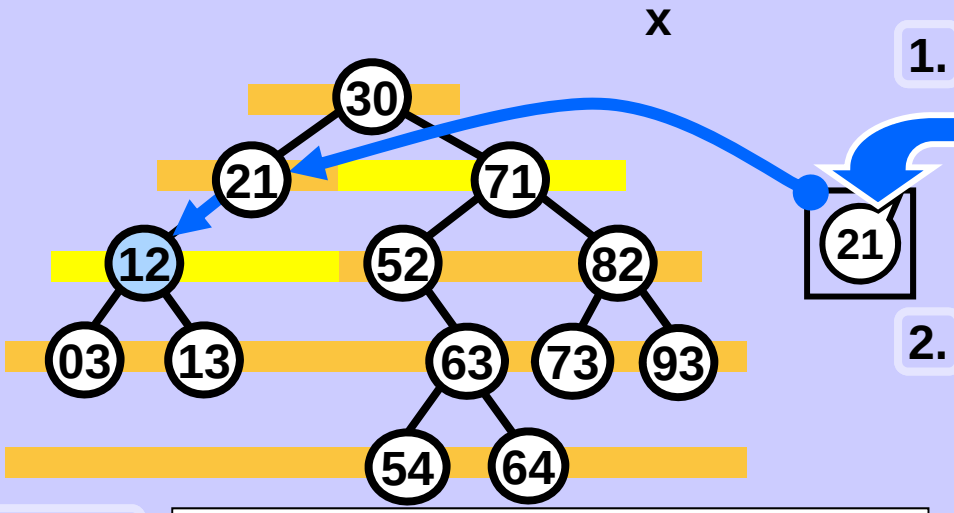


1. x = Dequeue(), print (x.key).

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



Output: 30



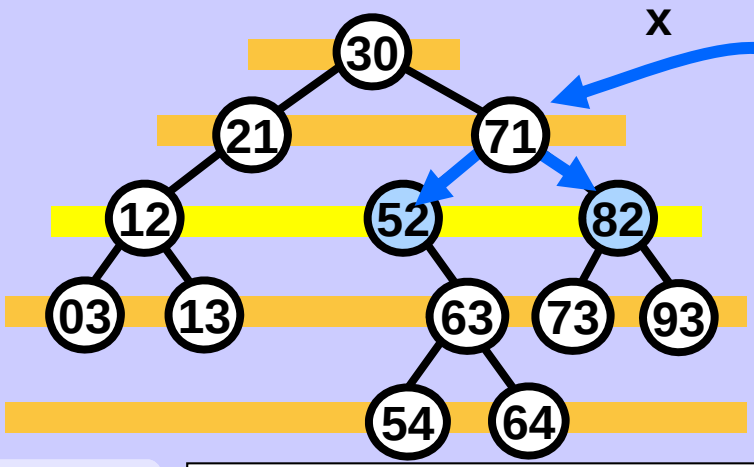
1. x = Dequeue(), print (x.key).

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

Output: 30 21

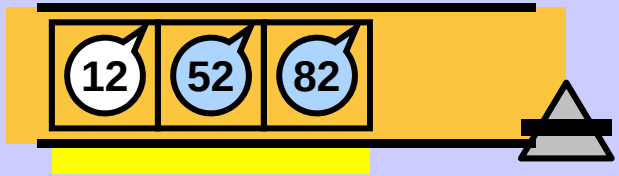
*) if exists

Breadth-first search (BFS) of a tree

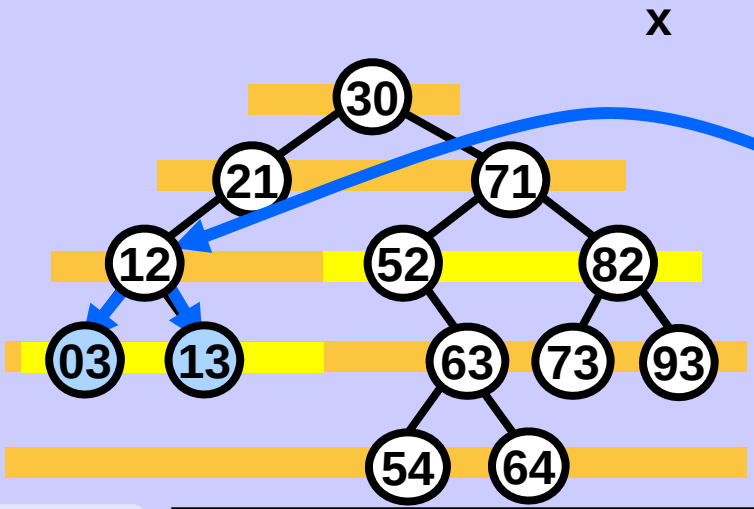


1. x = Dequeue(), print (x.key).

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

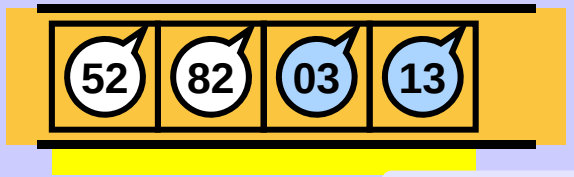


Output: 30 21 71



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

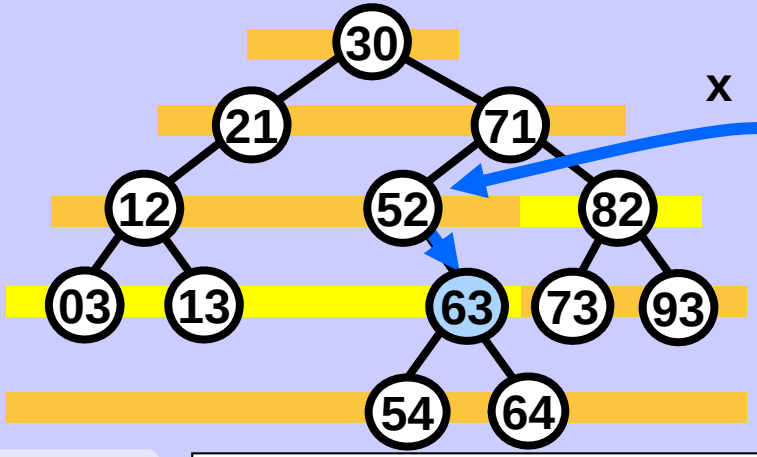
2. Enqeue(x.left), Enqueue(x.right). *)



Output: 30 21 71 12

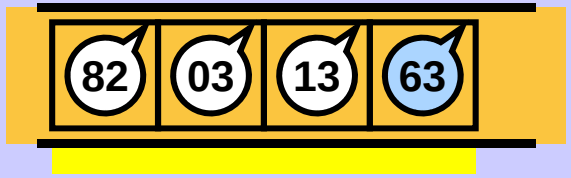
*) if exists

Breadth-first search (BFS) of a tree

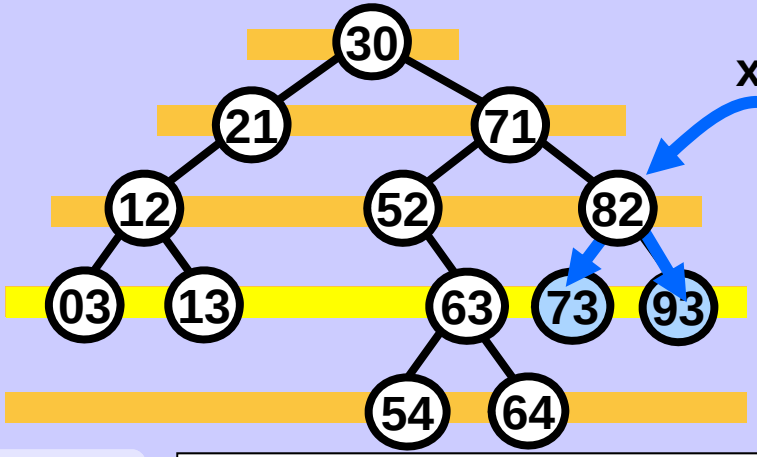


1. x = Dequeue(), print (x.key).

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

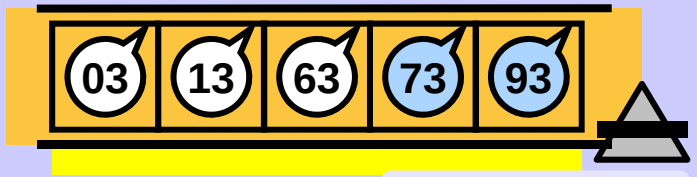


Output: 30 21 71 12 52



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

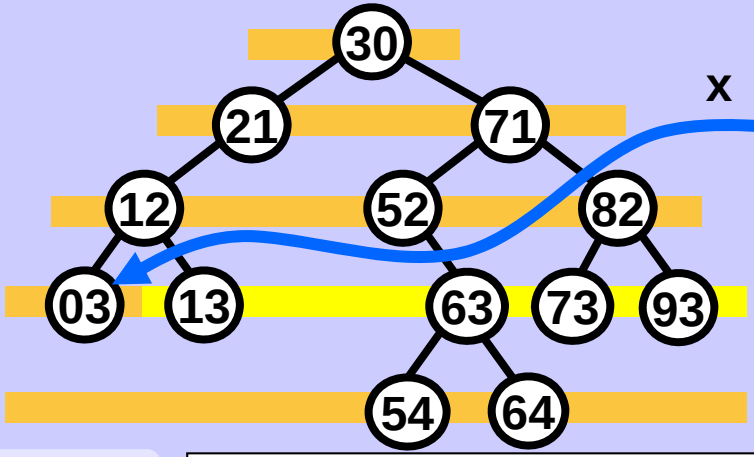
2. Enqeue(x.left), Enqueue(x.right). *)



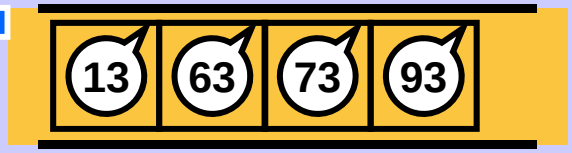
Output: 30 21 71 12 52 82

*) if exists

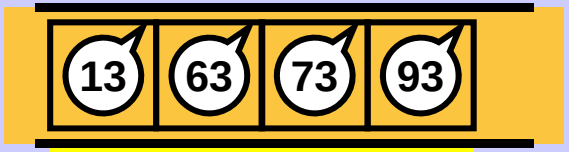
Breadth-first search (BFS) of a tree



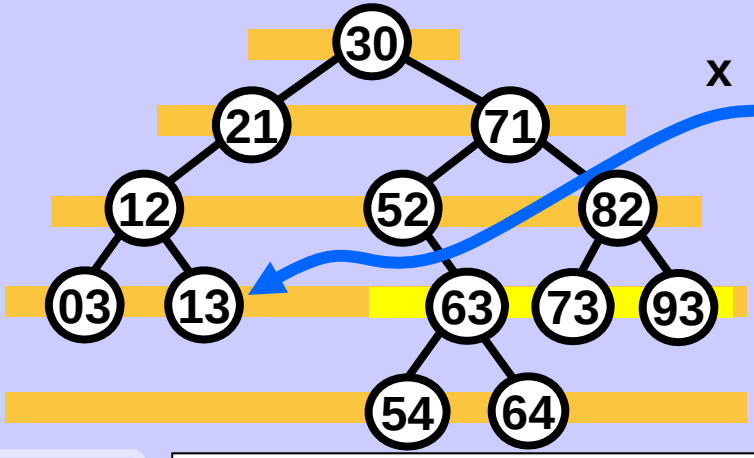
1. x = Dequeue(), print (x.key).



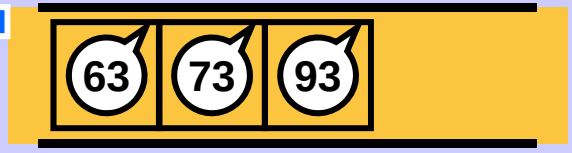
2. Enqeue(x.left), Enqueue(x.right). *)



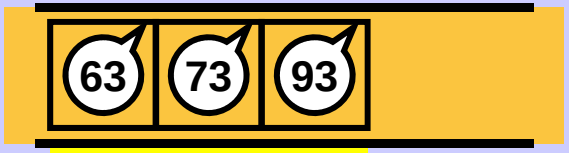
Output: 30 21 71 12 52 82 03



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



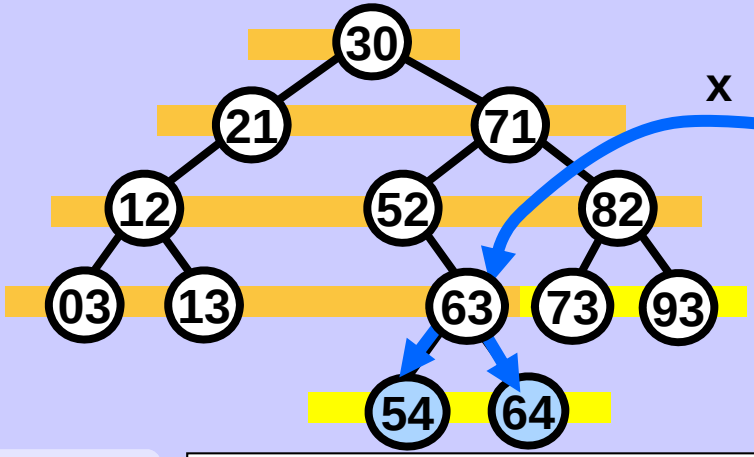
2. Enqeue(x.left), Enqueue(x.right). *)



Output: 30 21 71 12 52 82 03 13

*) if exists

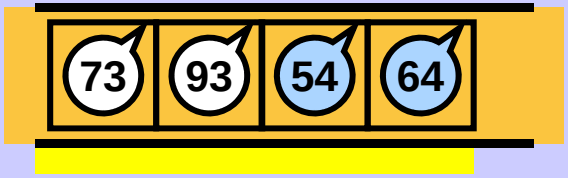
Breadth-first search (BFS) of a tree



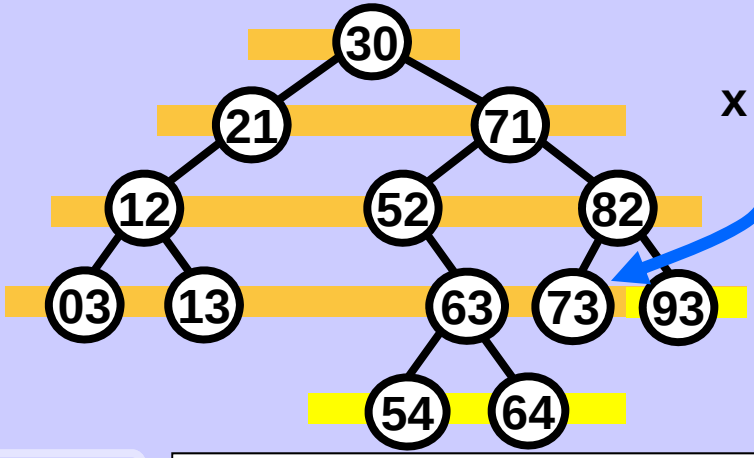
1. x = Dequeue(), print (x.key).

x → 63

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

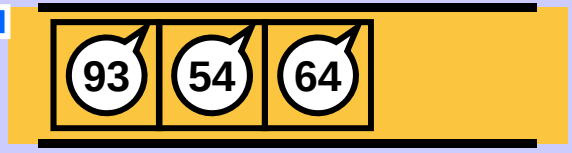


Output: 30 21 71 12 52 82 03 13 63

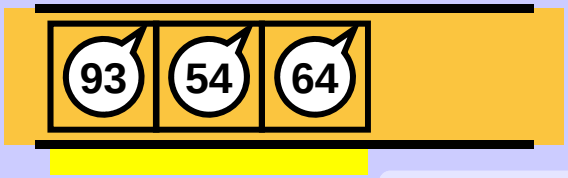


1. x = Dequeue(), print (x.key).

x → 73



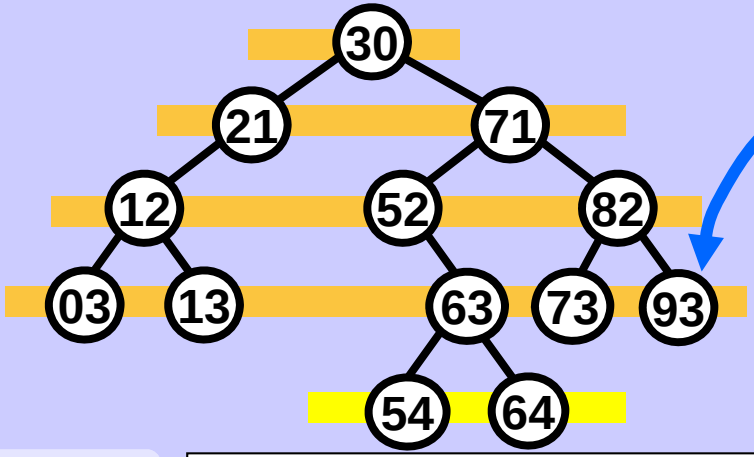
2. Enqeue(x.left), Enqueue(x.right). *)



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

*) if exists

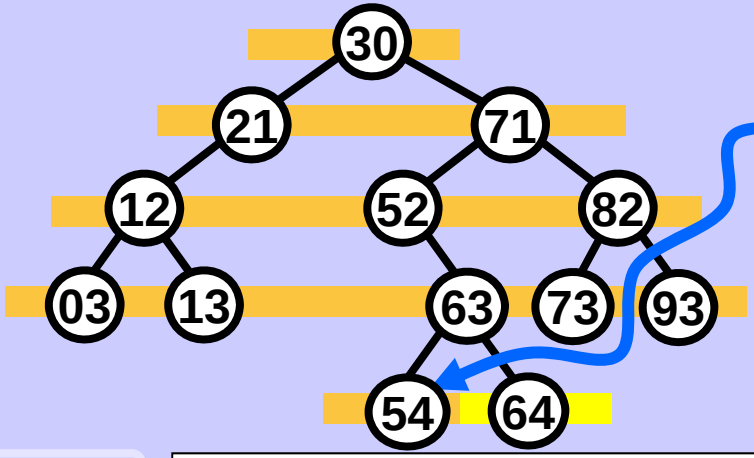
Breadth-first search (BFS) of a tree



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

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

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



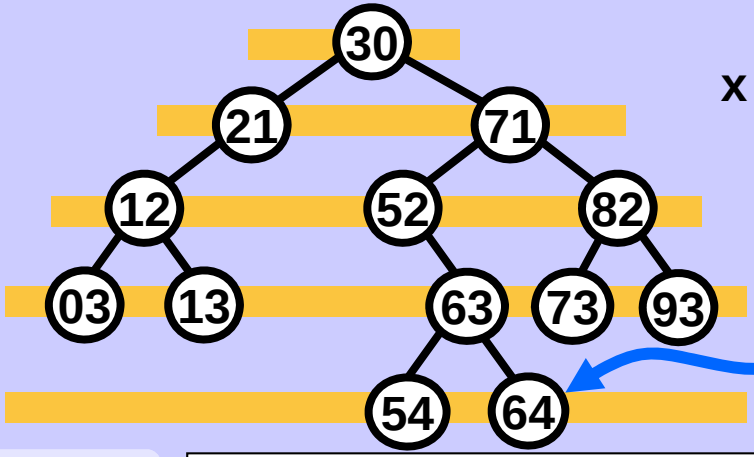
1. x = Dequeue(), print (x.key).

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

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

*) if exists

Breadth-first search (BFS) of a tree



x

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

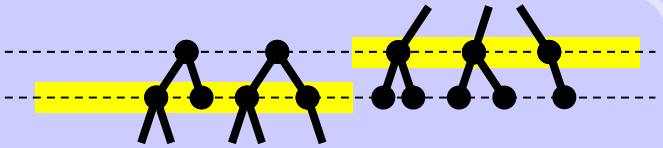
2. Enqeue(x.left), Enqueue(x.right). *)

*) if exists.

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

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) of a tree

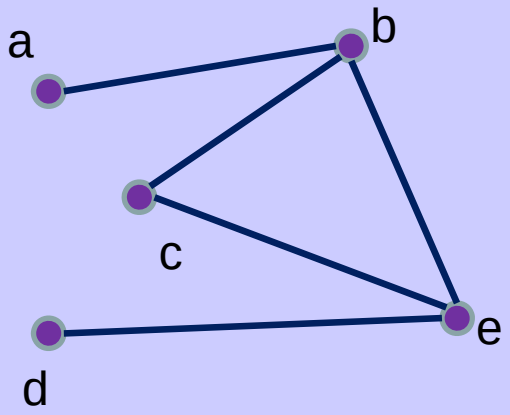
```
void binaryTreeBFS (Node node) {
    if (node == null) return;
    Queue q = new Queue();          // init
    q.Enqueue(node);                // root into queue
    while (!q.Empty()) {
        node = q.Dequeue();
        print(node.key);            // process node
        if (node.left  != null) q.Enqueue(node.left);
        if (node.right != null) q.Enqueue(node.right);
    }
}
```

Graphs

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

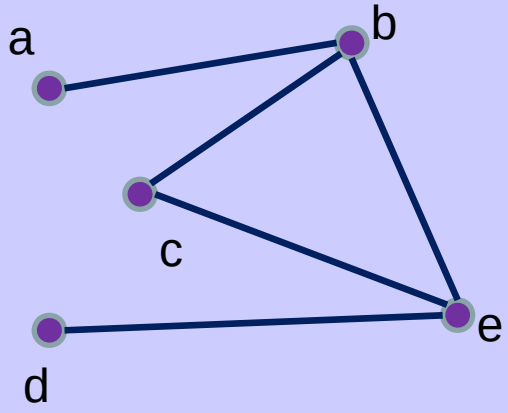
  set of pairs of vertices $\mathcal{E}$.

  Each pair is an edge.
- $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
- Example:
  - V = {a, b, c, d, e}
  - E = {{a,b},{b,e},{b,c},
    {c,e},{e,d}}

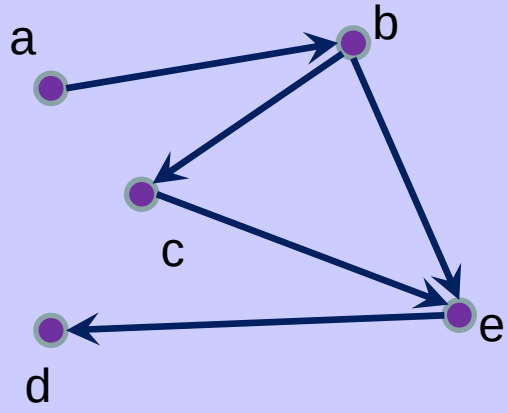


Graphs - directed/undirected

- **Undirected graph**
  - **An edge is an <u>unordered</u> pair of vertices.**
- **E = {{a,b},{b,e},{b,c}, {c,e},{e,d}}**



- **Directed graph**
  - **An edge is an <u>ordered</u> pair of vertices.**
- **E = {{a,b},{b,e},{b,c}, {c,e},{e,d}}**



Graph – adjacency matrix

- Let $G = (V, E)$ be 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

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

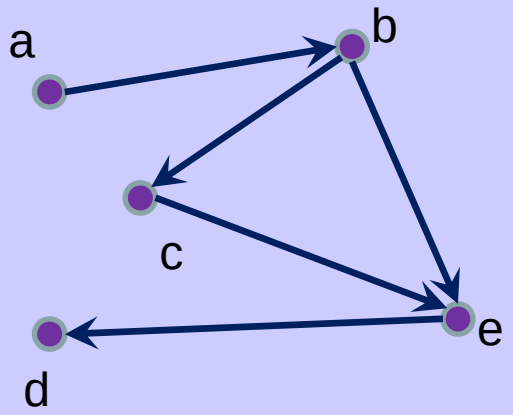
defined by the relation

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

Graph – adjacency matrix

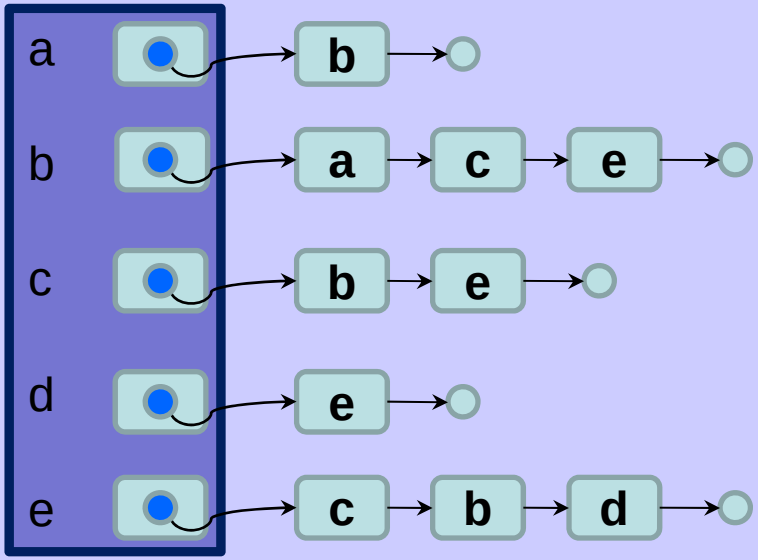
- **Directed graph example**

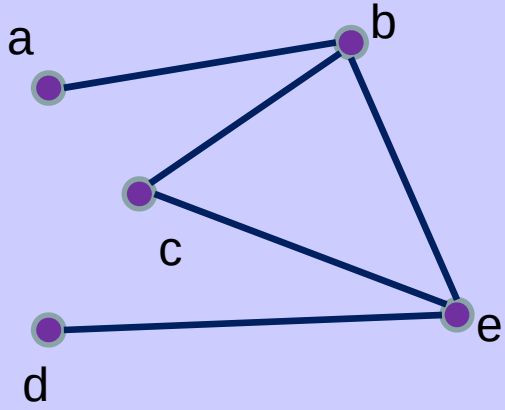
	a	b	c	d	e
a	0	1	0	0	0
b	0	0	1	0	1
c	0	0	0	0	1
d	0	0	0	0	0
e	0	0	0	1	0



Graph – list of neighbours

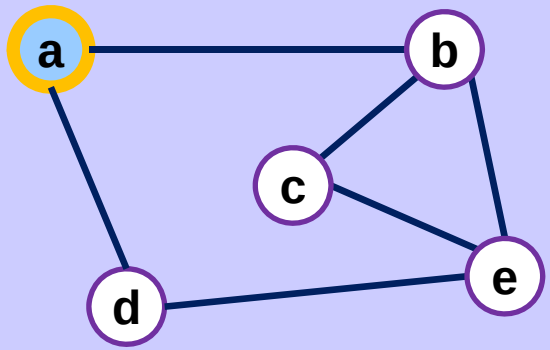
- Let $G = (V, E)$ be an (un)directed graph with n vertices.
- Denote vertices $v_1, \ldots, v_n$ (in an arbitrary order).
- List of neighbours of G is an array P of size n of pointers.
  - $P[i]$ points to the list of all vertices which are adjacent to v_i.





Depth-first search in a graph

Initialisation



Create empty stack

Push the start vertex to the stack

Output

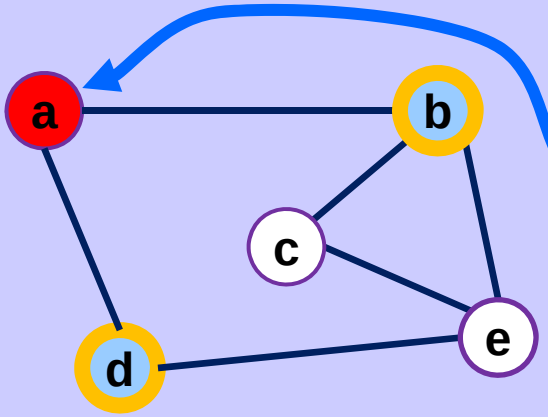
Vertex

Main loop

While the stack is not empty do:

1. Remove the top element from the stack and process it.
2. Push the unvisited children of the removed element to the stack.

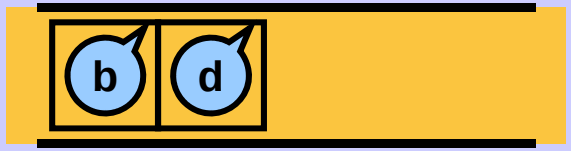
Depth-first search in a graph



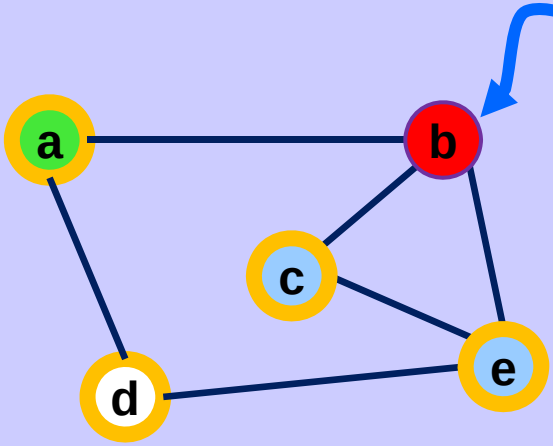
1. node = Pop(), print (node.key).

node

2. Push(node.Neighbors()).



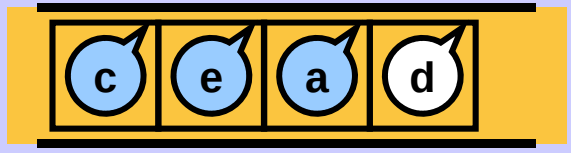
Output: a



1. node = Pop(), print(node.key).

node

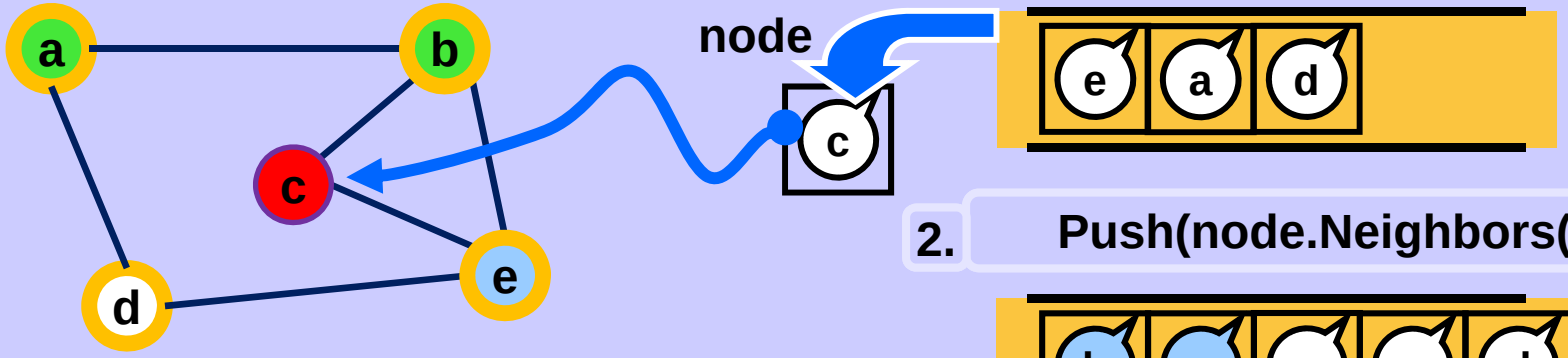
2. Push(node.Neighbors()).



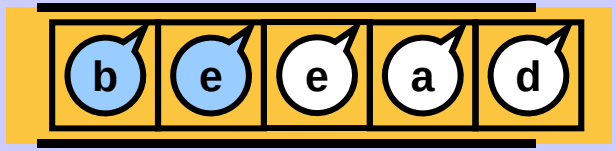
Output: a b

Depth-first search in a graph

1. node = Pop(), print(node.key).

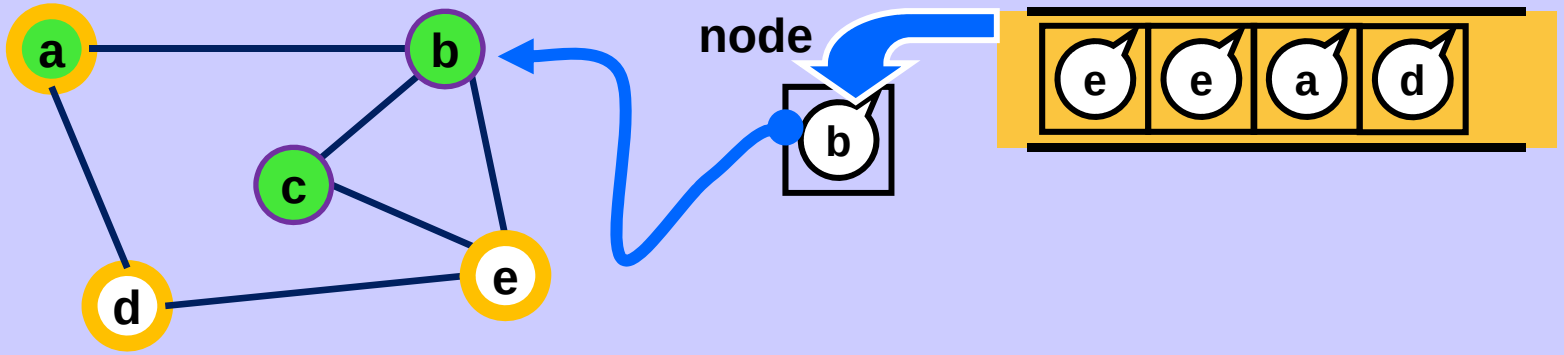


2. Push(node.Neighbors()).



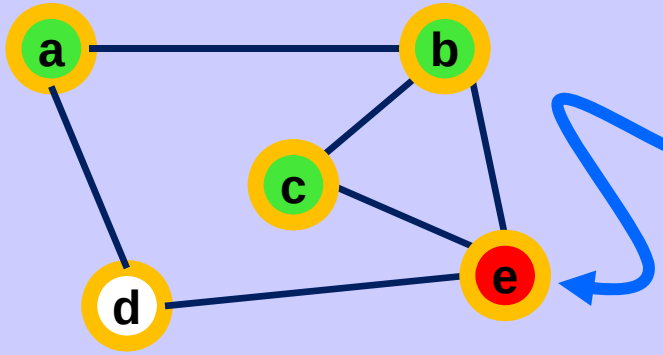
Output: a b c

1. node = Pop().



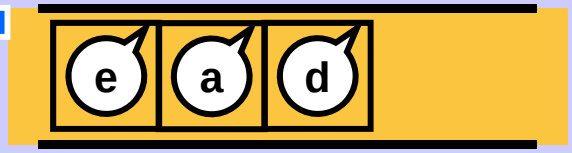
Output: a b c

Depth-first search in a graph

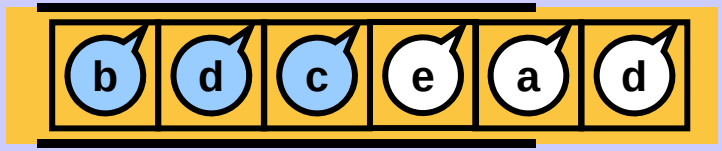


1. node = Pop(), print(node.key).

node

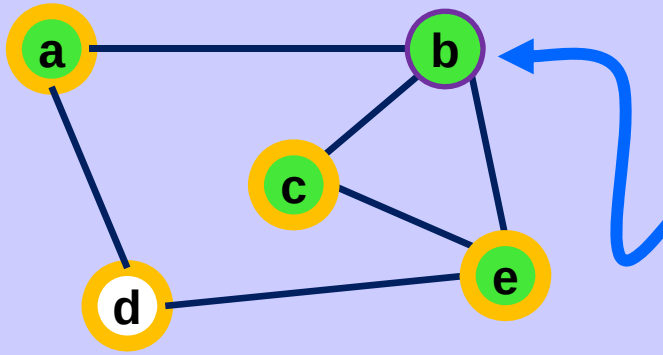


2. Push(node.Neighbors()).

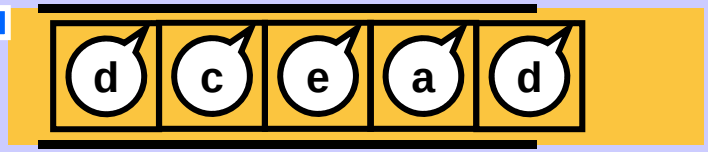


Output: a b c e

1. node = Pop().

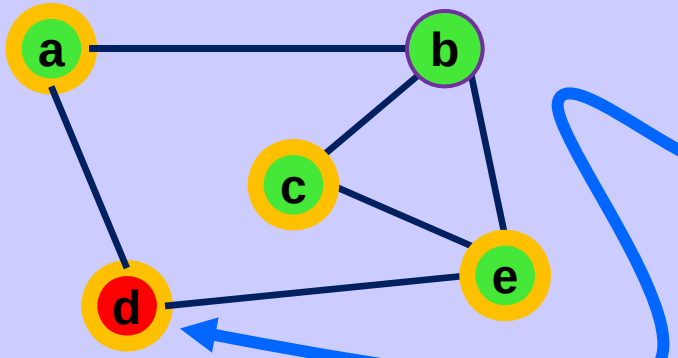


node



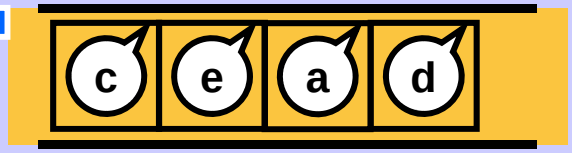
Output: a b c e

Depth-first search in a graph

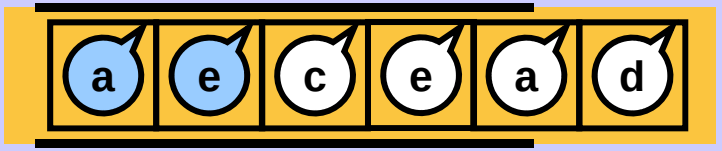


1. node = Pop(), print(node.key).

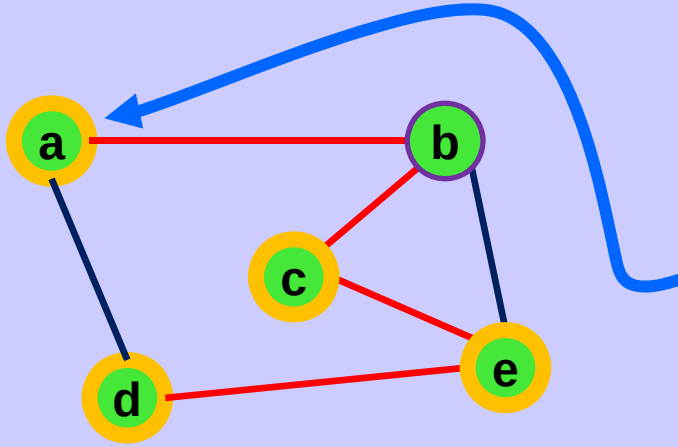
node



2. Push(node.Neighbors()).

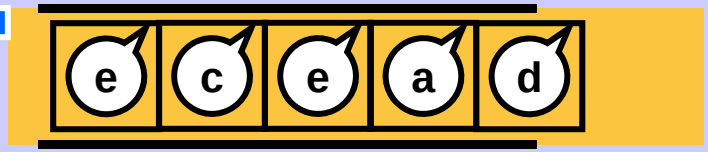


Output: a b c e d



1. node = Pop().

node



…. Pop()…. Pop()
FINISHED

Output: a b c e d

Depth-first search in a graph

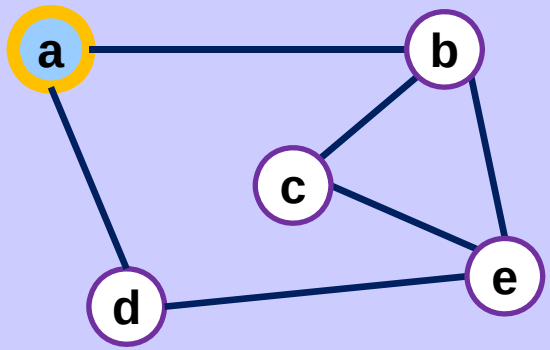
```
void graphDepth (Node startNode) {
    Set   visited = new Set();
    Stack q = new Stack ();          // init
    q.Push(startNode);               // startNode into queue
    while (!q.Empty()) {
        node = q.Pop();
        if (node not in visited) {
            visited.add(node);
            print(node.key);         // process node
            forall x in node.Neighbors()
              q.Push(x);
        }
    }
```

Depth-first search in a graph -- speedup

```
void graphDepth2 (Node startNode) {
    Set   touched = new Set();
    Stack q = new Stack();          // init
    q.Push(startNode);              // startNode into queue
    touched.add(startNode);
    while (!q.Empty()) {
        node = q.Pop();
        print(node.key);       // process node
        forall x in node.Neighbors()
            if (x not in touched) {
                q.Push(x);
                touched.add(node);
            }
    }
```

Breadth-first search in a graph

Initialisation



Create empty queue

Enqueue the start vertex

Output

Front

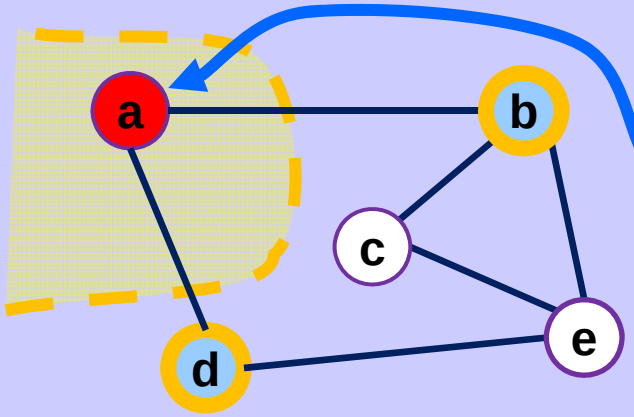
Tail

Main loop

While the queue is not empty do:

1. Remove the front element from the queue and process it.
2. Enqueue the ***unvisited*** children of the removed element.

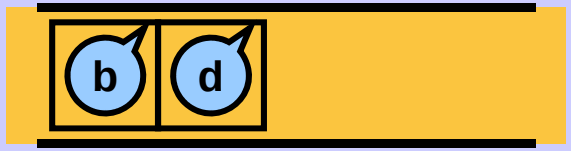
Breadth-first search in a graph



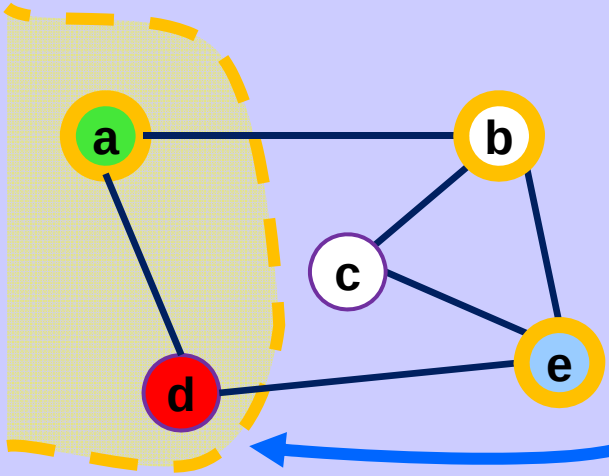
1. node = Dequeue(), print (node.key).

vertex

2. Enqueue(node.Neighbors()).

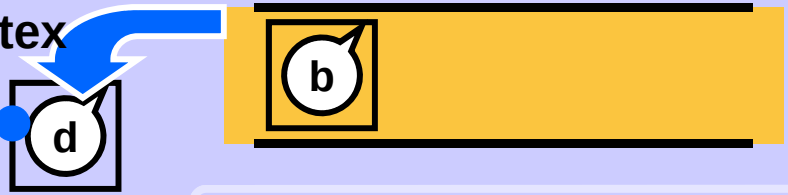


Output: a

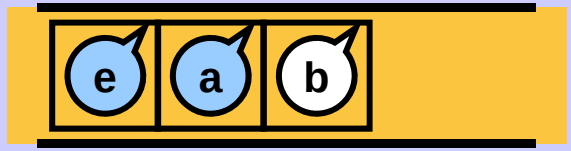


1. node = Dequeue(), print (node.key).

vertex

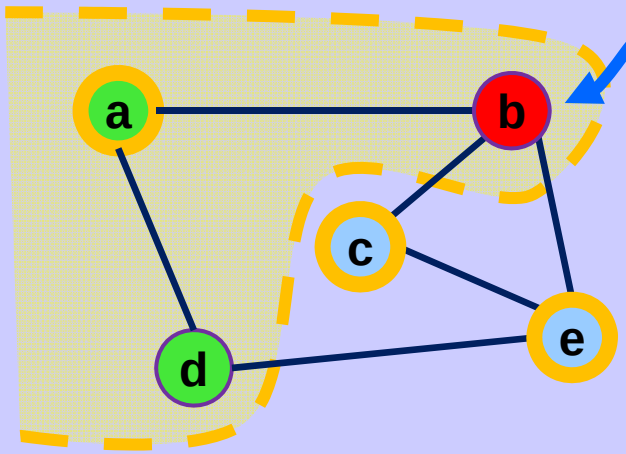


2. Enqueue(node.Neighbors()).



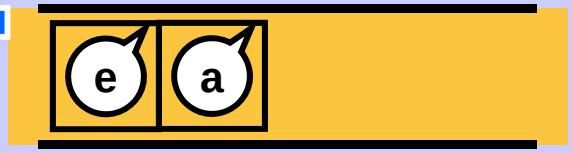
Output: a d

Breadth-first search in a graph

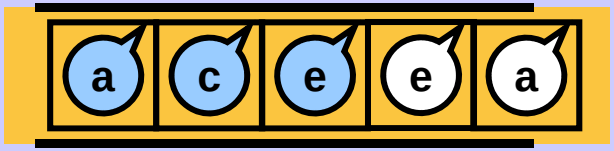


1. node = Dequeue(), print (node.key).

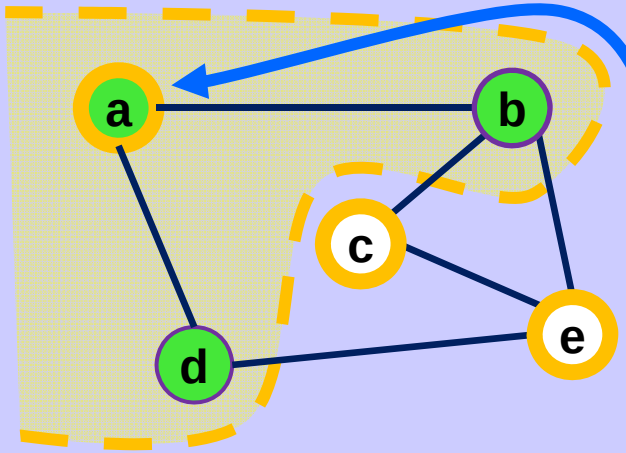
vertex



2. Enqueue(node.Neighbors()).

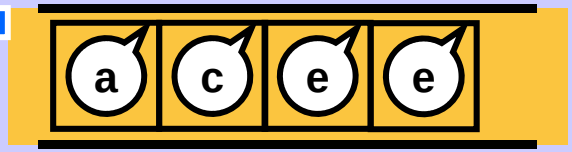


Output: a d b



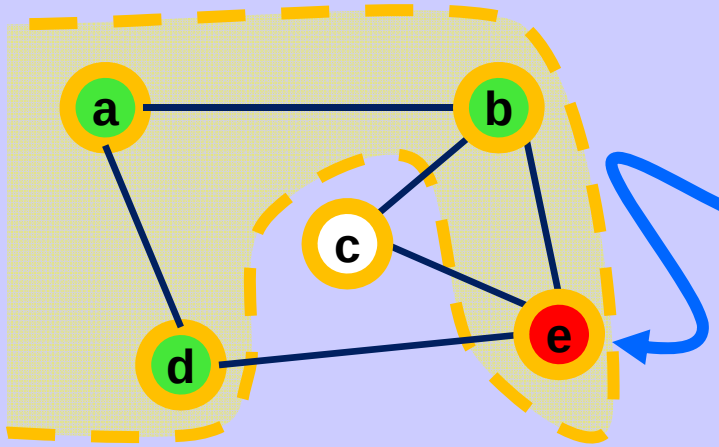
1. node = Dequeue().

vertex



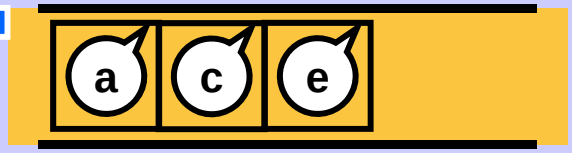
Output: a d b

Breadth-first search in a graph

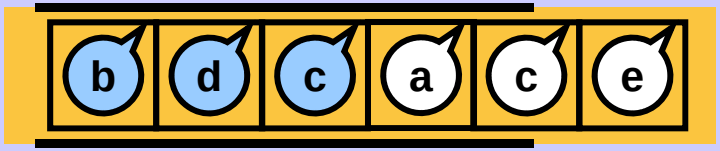


1. node = Dequeue(), print (node.key).

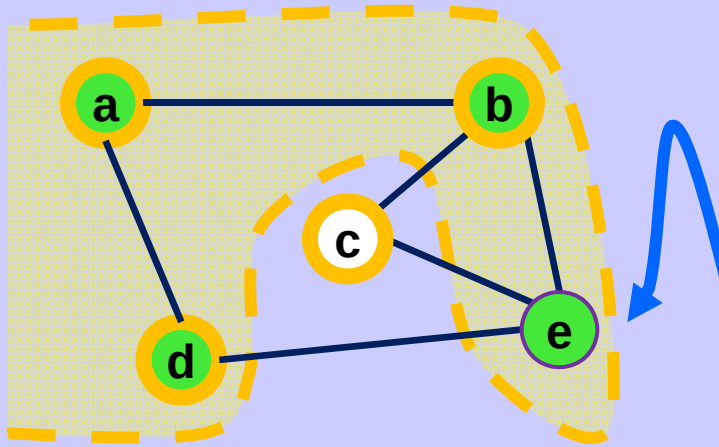
vertex



2. Enqueue(node.Neighbors()).

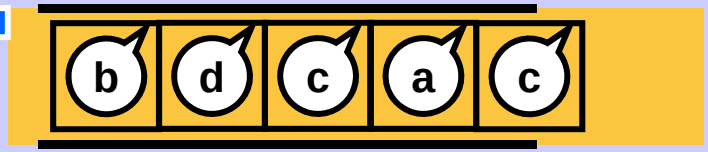


Output: a d b e



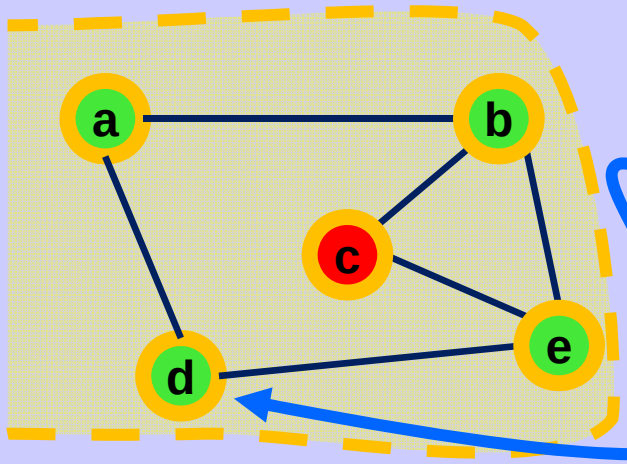
1. node = Dequeue().

vertex



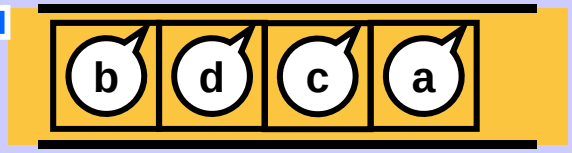
Output: a d b e

Breadth-first search in a graph

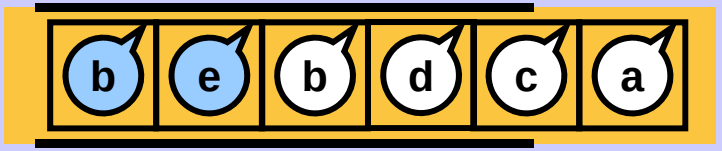


1. node = Dequeue(), print (node.key).

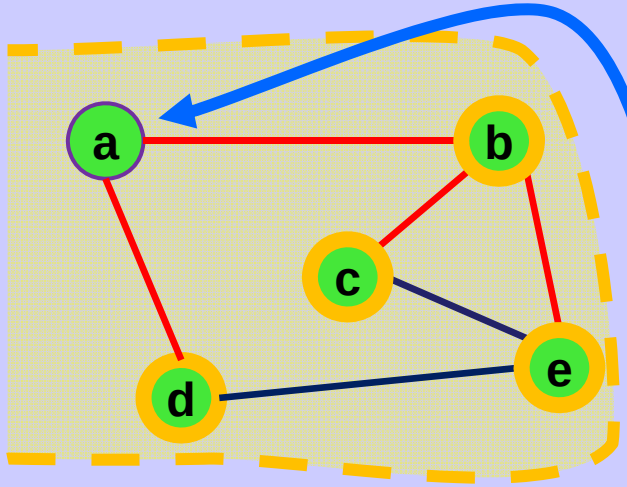
vertex



2. Enqueue(node.Neighbors()).

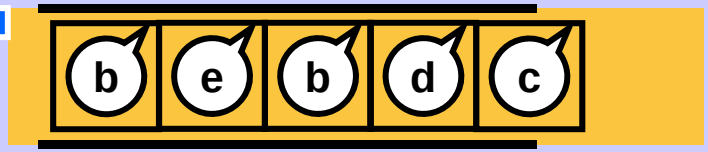


Output: a d b e c



1. node = Dequeue().

vertex



…. Dequeue()…. Dequeue()
FINISHED

Output: a d b e c

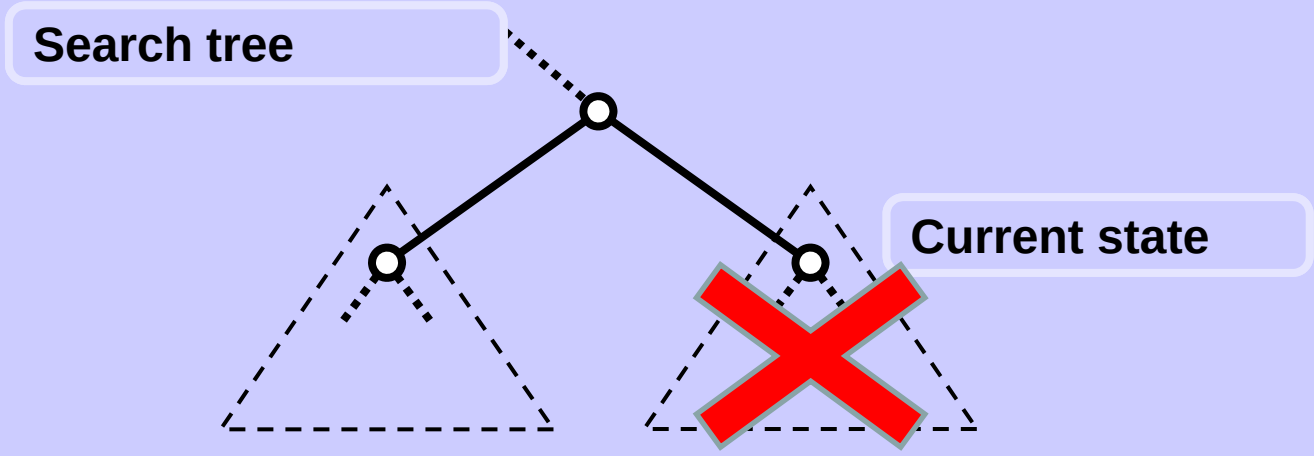
Breadth-first search in a graph

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

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



Pruning example – magic square

- Magic square of order N
  - square matrix of order N
  - contains exactly once each value from 1 to 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 N^2
- Pruning: Whenever the sum of the row or column is not correct:
  - sum of all values in the square is $\frac{1}{2} N^2 (N^2+1)$
  - sum of all values in a row or column is $\frac{1}{2} N (N^2+1)$

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 asses the order of vertices/edges/paths in which they are processed during the search in large graphs.**

- **Example: Knight tour on an $N \times N$ 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.**