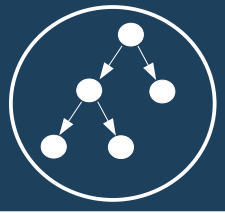
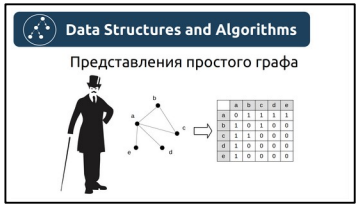


# Data Structures and Algorithms

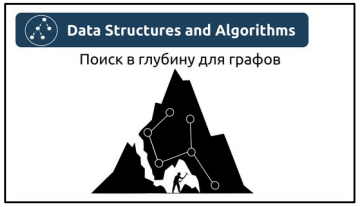
## Поиск в ширину для графов

# Список лекций необходимых для занятия

Перед просмотром этого занятия нужно просмотреть следующие лекции.




Представление простого графа

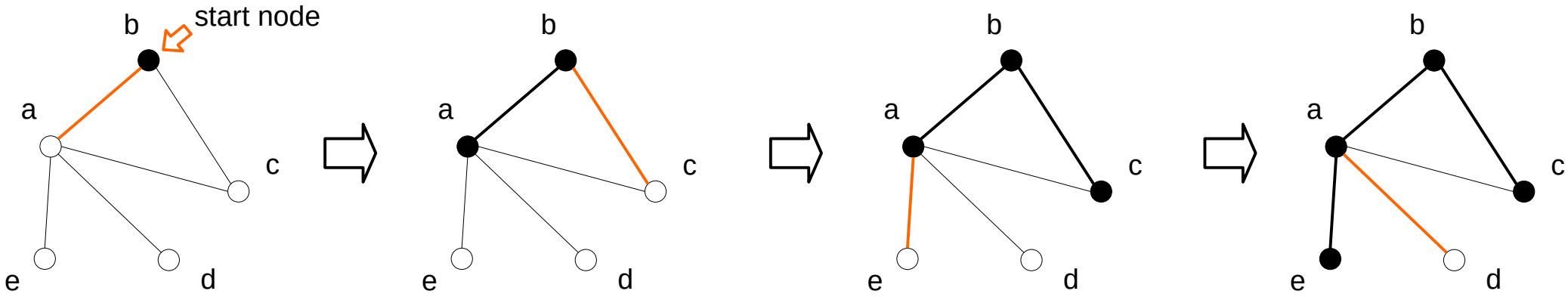


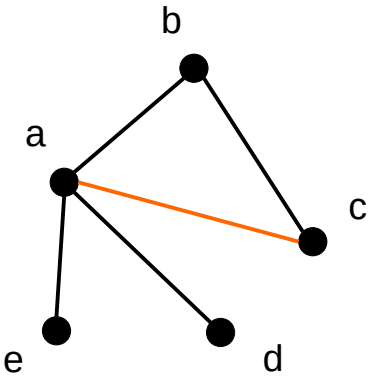

Поиск в глубину для графов

# Поиск в ширину

Поиск в ширину (breadth-first search, BFS) — один из методов обхода графа. Особенностью является поочередный обход ближайших вершин к стартовой. В отличии от поиска в глубину, сначала обрабатываются все вершины смежные данной, а только потом происходит переход. Для решения подобной задачи используется тот же подход (цветовая маркировка) как и для поиска в глубину.

# Графическое пояснение алгоритма

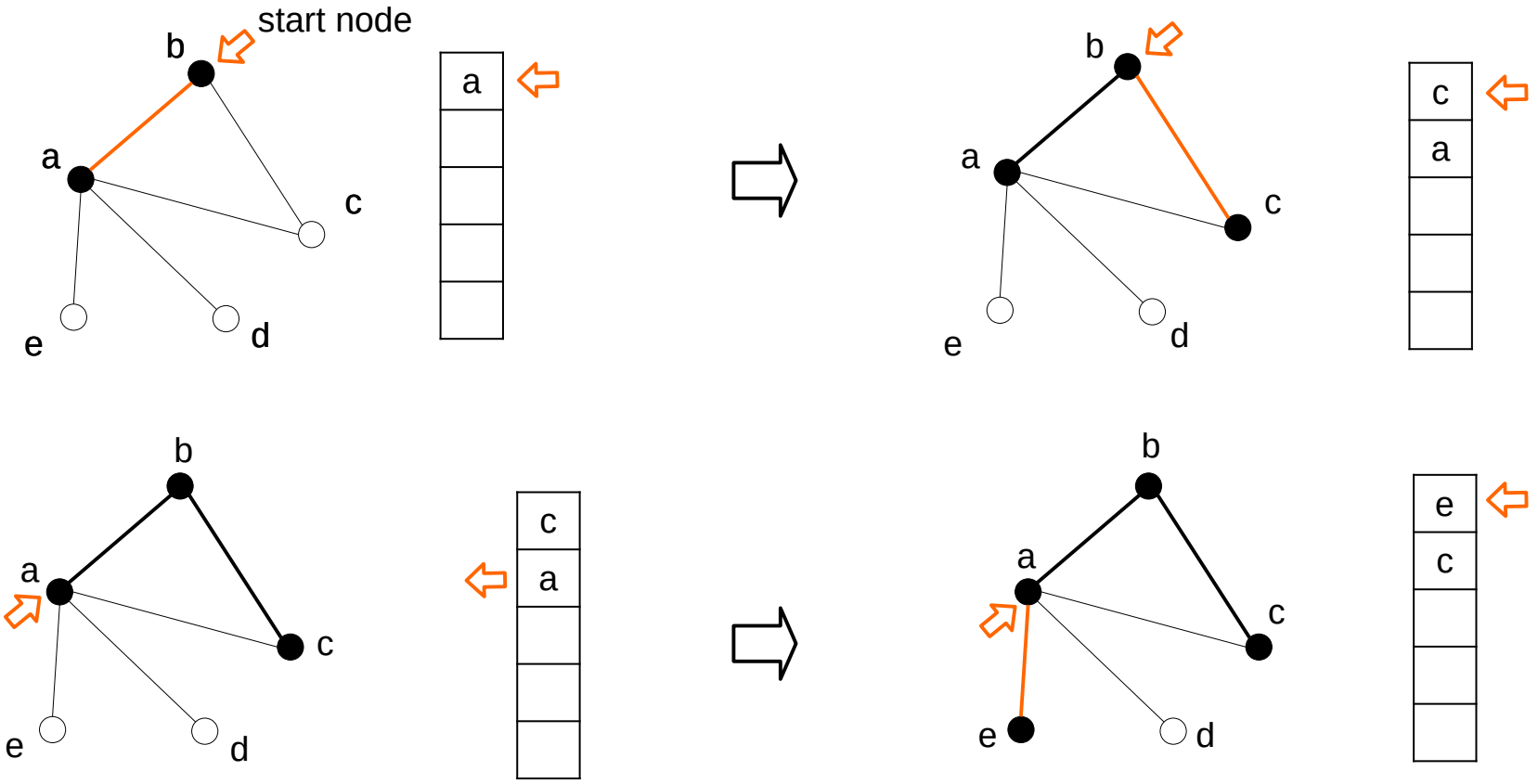


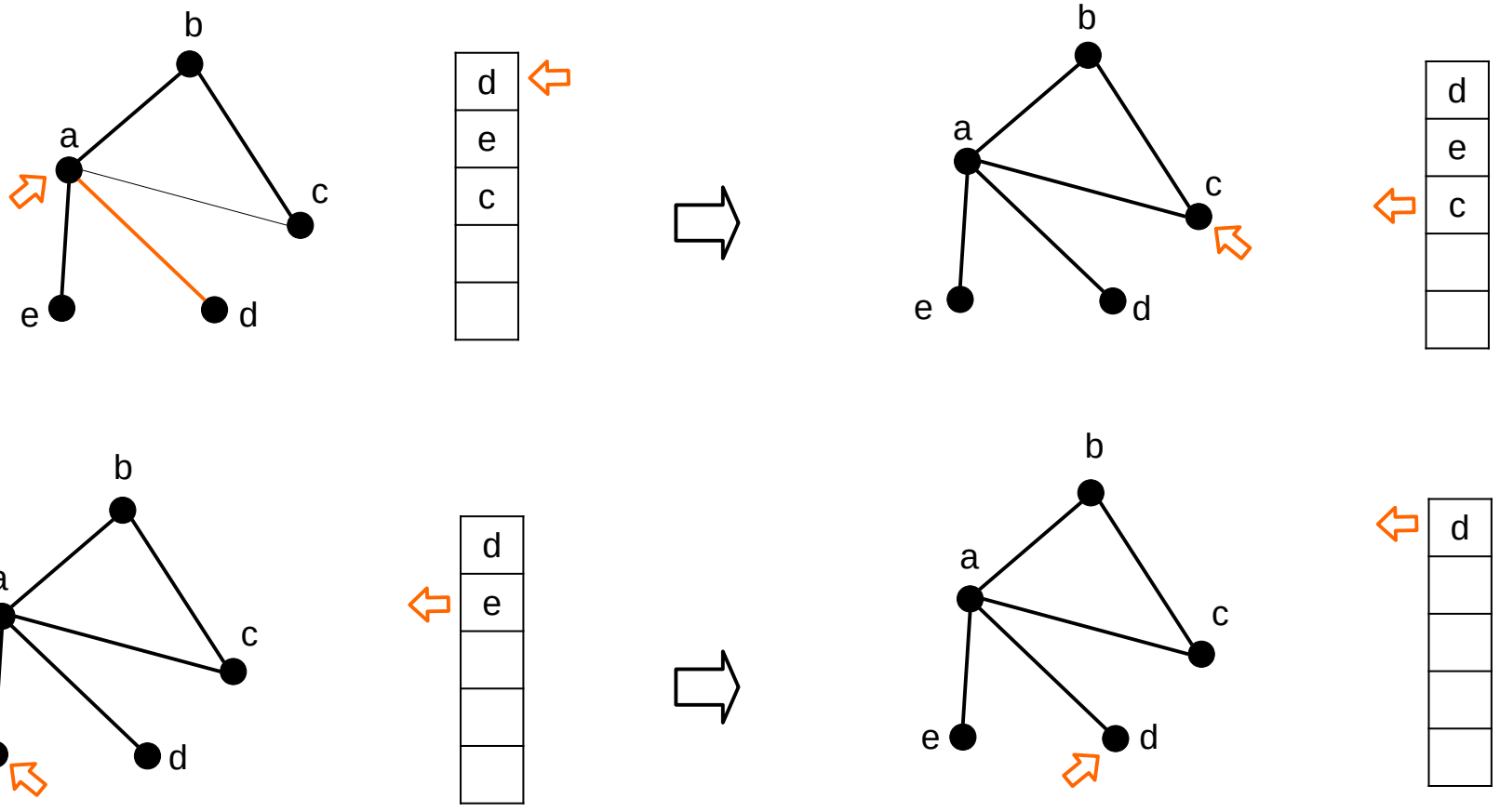
# Наиболее простая реализация

В отличии от поиска в глубину, поиск в ширину реализуется обычным циклическим алгоритмом без применения рекурсии. Для этого используют очередь в которую добавляют все вершины смежные данной и имеющие белый цвет, после этого начинают извлекать и обрабатывать вершины из этой очереди. Алгоритм заканчивается когда очередь становиться пустой.

# Графическое объяснение реализации



# Графическое объяснение реализации



# Проверка графа на связность

Поиск в ширину также можно использовать для проверки графа на связность. Достаточно запустить поиск в ширину (указывая в качестве стартовой любую вершину) и после него проверить все вершины на цвет. Если все вершины поменяли цвет, то граф связный, если остались «белого цвета» то граф не связный.

# Реализация на Python

# Описание структуры вершины и графа

```
class Graph:
    class Node:
        def __init__(self, node_id, node_data=None):
            self.node_id = node_id
            self.node_color = 0  # 0 - white, 1 - black
            self.node_data = node_data
            self.adjacent_nodes = set()

        def __str__(self):
            result = ""
            result += "node [id = "+str(self.node_id)+", data =  " + \
                str(self.node_data)+", color = "+str(self.node_color)+"]"
            return result

    def __init__(self):
        self.nodes = {}
```

# Метод поиска вершины по индификатору

```
def find_node_by_id(self, node_id):
    return self.nodes.get(node_id)
```

# Методы добавления и удаления вершины

```
def add_node(self, node_id, node_data):
    if self.find_node_by_id(node_id) is not None:
        raise Exception("node already exists")
    new_node = self.Node(node_id, node_data)
    self.nodes[node_id] = new_node

def del_node(self, node_id):
    delete_node = self.find_node_by_id(node_id)
    if delete_node is None:
        raise Exception("node does not exists")
    for nodes in delete_node.adjacent_nodes:
        nodes.adjacent_nodes.remove(delete_node)
    del (self.nodes[node_id])
```

# Методы добавления и удаления ребра

```
def add_edge(self, node_id_from, node_id_to):
    node_from = self.find_node_by_id(node_id_from)
    node_to = self.find_node_by_id(node_id_to)
    if node_from is None or node_to is None:
        raise Exception("node does not exist")
    node_from.adjacent_nodes.add(node_to)
    node_to.adjacent_nodes.add(node_from)

def del_edge(self, node_id_from, node_id_to):
    node_from = self.find_node_by_id(node_id_from)
    node_to = self.find_node_by_id(node_id_to)
    if node_from is None or node_to is None:
        raise Exception("node does not exist")
    node_from.adjacent_nodes.remove(node_to)
    node_to.adjacent_nodes.remove(node_from)
```

# Метод проверки смежности вершин

```
def is_adjacent_nodes(self, node_id_from, node_id_to):
    node_from = self.find_node_by_id(node_id_from)
    node_to = self.find_node_by_id(node_id_to)
    if node_from is None or node_to is None:
        raise Exception("node does not exist")
    return node_from in node_to.adjacent_nodes and node_to in node_from.adjacent_nodes
```

# Метод поиска вершин смежных данной

```
def get_adjacent_nodes(self, node_id):
    node = self.find_node_by_id(node_id)
    if node is None:
        raise Exception("node does not exists")
    return node.adjacent_nodes
```

# Метод поиска в ширину

```
def bfs(self, start_node_id):
    if type(start_node_id) is not self.Node:
        node = self.find_node_by_id(start_node_id)
        if node is None:
            raise Exception("node does not exists")
    else:
        node = start_node_id
    node_queue = []
    node_queue.insert(0, node)
    while len(node_queue) > 0:
        start_node = node_queue.pop()
        if start_node.node_color == 0:
            for node in start_node.adjacent_nodes:
                if node.node_color == 0:
                    node_queue.insert(0, node)
            start_node.node_color = 1
```

# Метод проверки графа на связность

```
def paint_nodes_to_white(self):
    for node_id in self.nodes:
        node = self.nodes.get(node_id)
        node.node_color = 0

def is_connected_graph(self):
    result = True
    self.paint_nodes_to_white()
    for node_id in self.nodes:
        self.dfs(node_id)
        break
    for node_id in self.nodes:
        if self.nodes.get(node_id).node_color == 0:
            result = False
            break
    self.paint_nodes_to_white()
    return result
```

# Реализация на Java

# Описание представления графа

```
private class Node {
        final String id;
        Object data;
        List<Node> adjacentNodes = new ArrayList<>();
        int color = 0;// 0 - white, 1- black

        public Node(String id) {
            super();
            this.id = id;
        }
    }

    private final Map<String, Node> nodes = new HashMap<>();
```

# Методы добавления вершины

```
public void addNode(String id, Object data) {
    if (nodes.get(id) != null) {
        throw new IllegalArgumentException("Node with this ID already exists");
    }
    Node newNode = new Node(id);
    newNode.data = data;
    nodes.put(id, newNode);
}

public void addNode(String id) {
    addNode(id, null);
}
```

# Метод добавления ребра

```
public void addEdge(String idFrom, String idTo) {
    Node nodeFrom = nodes.get(idFrom);
    Node nodeTo = nodes.get(idTo);
    if (nodeFrom == null || nodeTo == null) {
        throw new IllegalArgumentException("Node with this id does not exist");
    }
    if (nodeFrom == nodeTo) {
        throw new IllegalArgumentException("Loop edge");
    }
    nodeFrom.adjacentNodes.add(nodeTo);
    nodeTo.adjacentNodes.add(nodeFrom);
}
```

# Метод удаления вершины

```
public void removeNode(String id) {
    Node removeNode = nodes.get(id);
    if (removeNode == null) {
        return;
    }
    for (Node node : removeNode.adjacentNodes) {
        node.adjacentNodes.remove(removeNode);
    }
    nodes.remove(id);
}
```

# Метод удаления ребра

```
public void removeEdge(String idFrom, String idTo) {
    Node nodeFrom = nodes.get(idFrom);
    Node nodeTo = nodes.get(idTo);
    if (nodeFrom == null || nodeTo == null) {
        throw new IllegalArgumentException("Node with this id does not exist");
    }
    nodeFrom.adjacentNodes.remove(nodeTo);
    nodeTo.adjacentNodes.remove(nodeFrom);
}
```

# Метод проверки смежности вершин

```
public boolean isAdjacent(String idFrom, String idTo) {
     Node nodeFrom = nodes.get(idFrom);
     Node nodeTo = nodes.get(idTo);
     if (nodeFrom == null || nodeTo == null) {
          return false;
     }
     return nodeFrom.adjacentNodes.contains(nodeTo);
}
```

# Метод получения списка смежных вершин

```
public String[] getAdjacentNodesId(String id) {
    Node node = nodes.get(id);
    if (node == null) {
        return null;
    }
    String[] ids = new String[node.adjacentNodes.size()];
    for (int i = 0; i < ids.length; i++) {
        ids[i] = node.adjacentNodes.get(i).id;
    }
    return ids;
}
```

# Метод получения данных и установки данных для вершины

```
public Object getNodeDataById(String id) {
    Node node = nodes.get(id);
    if (node == null) {
        return null;
    }
    return node.data;
}

public void setNodeDataById(String id, Object data) {
    Node node = nodes.get(id);
    if (node == null) {
        return;
    }
    node.data = data;
}
```

# Методы реализующие поиск в ширину

```
public void bfs(Node startNode) {
     Deque<Node> nodeDeq = new ArrayDeque<>();
     nodeDeq.push(startNode);
     for (; nodeDeq.size() > 0;) {
          Node currentNode = nodeDeq.poll();
          for (Node node : currentNode.adjacentNodes) {
               if (node.color == 0) {
                    nodeDeq.push(node);
               }
          }
          currentNode.color = 1;
     }
}

public void bfs(String startNodeId) {
     Node node = nodes.get(startNodeId);
     if (node == null) {
          return;
     }
     bfs(node);
}
```

# Методы для проверки графа на связность

```
public boolean isConnectedGraph() {
     boolean result = true;
     repaintNodesToWhiteColor();
     for (String nodeId : nodes.keySet()) {
          bfs(nodes.get(nodeId));
          break;
     }
     for (String nodeId : nodes.keySet()) {
          if (nodes.get(nodeId).color == 0) {
               result = false;
               break;
          }
     }
     repaintNodesToWhiteColor();
     return result;
}

public void repaintNodesToWhiteColor() {
     for (String nodeId : nodes.keySet()) {
          nodes.get(nodeId).color = 0;
     }
}
```

# Реализация на Fortran

# Описание вершины и графа

```
type Node
    character(len=10)::node_id
    integer::node_data
    integer::node_color !0 - white, 1 - black
    logical::is_present

end type Node

type Graph
    type(Node), allocatable::nodes(:)
    integer, allocatable::adjacency_matrix(:,:)
    integer::matrix_size
    contains

    procedure,pass::init
    procedure,pass::add_node
    procedure,pass::add_edge
    procedure, pass::find_node_index_by_id
    procedure, pass::remove_node
    procedure, pass::remove_edge
    procedure, pass::print_graph
    procedure, pass::is_node_adjacency
    procedure, pass::get_adjacency_nodes_index
    procedure, pass::bfs
    procedure, pass::repaint_node_to_white_color
    procedure, pass::is_connected_graph
end type Graph
```

# Процедура инициализации и поиска вершины по id

```
subroutine init(this)
    class(Graph)::this
    this%matrix_size = 100
    allocate(this%nodes(this%matrix_size))
    allocate(this%adjacency_matrix(this%matrix_size,this%matrix_size))
    this%nodes%is_present = .false.
    this%nodes%node_color = 0
    this%adjacency_matrix = 0
end subroutine init

subroutine find_node_index_by_id(this, node_id, node_index)
    class(Graph)::this
    character(len=*), intent(in)::node_id
    integer, intent(inout)::node_index
    integer::i
    node_index = -1
    do i = 1, size(this%nodes)
        if (this%nodes(i)%is_present .and. this%nodes(i)%node_id == node_id) then
            node_index = i
            exit
        end if
    end do
end subroutine find_node_index_by_id
```

# Процедура добавления вершины

```
subroutine add_node(this, node_id, node_data, op_result)
    class(Graph)::this
    character(len=*), intent(in)::node_id
    integer, intent(in)::node_data
    logical, intent(inout)::op_result
    integer::i
    op_result = .false.
    call this%find_node_index_by_id(node_id,i)
    if(i /= -1) then
        return
    end if
    do i = 1, size(this%nodes)
        if (.not.this%nodes(i)%is_present) then
            this%nodes(i)%is_present = .true.
            this%nodes(i)%node_id = node_id
            this%nodes(i)%node_data = node_data
            op_result = .true.
            exit
        end if
    end do
end subroutine add_node
```

# Процедура добавления ребра

```
subroutine add_edge(this, node_id_from, node_id_to, op_result)
    class(Graph)::this
    character(len=*),intent(in)::node_id_from, node_id_to
    logical, intent(inout)::op_result
    integer::i, j
    i = -1
    j = -1
    op_result = .false.
    call this%find_node_index_by_id(node_id_from, i)
    call this%find_node_index_by_id(node_id_to, j)
    if(i == -1 .or. j == -1) then
        return
    end if
    this%adjacency_matrix(i,j) = 1
    this%adjacency_matrix(j,i) = 1
    op_result = .true.
end subroutine add_edge
```

# Процедура удаления вершины

```
subroutine remove_node(this, node_id, op_result)
    class(Graph)::this
    character(len=*),intent(in)::node_id
    logical, intent(inout)::op_result
    integer::i
    i = -1
    op_result = .false.
    call this%find_node_index_by_id(node_id, i)
    if (i == -1) then
        return
    end if
    this%nodes(i)%is_present = .false.
    this%adjacency_matrix(i,:) = 0
    this%adjacency_matrix(:,i) = 0
    op_result = .true.
end subroutine remove_node
```

# Процедура удаления ребра

```
subroutine remove_edge(this, node_id_from, node_id_to, op_result)
    class(Graph)::this
    character(len=*), intent(in)::node_id_from, node_id_to
    logical, intent(inout)::op_result
    integer::i, j
    i = -1
    j = -1
    op_result = .false.
    call this%find_node_index_by_id(node_id_from, i)
    call this%find_node_index_by_id(node_id_to, j)
    if(i == -1 .or. j == -1) then
        return
    end if
    this%adjacency_matrix(i,j) = 0
    this%adjacency_matrix(j,i) = 0
    op_result = .true.
end subroutine remove_edge
```

# Процедура проверки вершин на смежность

```
subroutine is_node_adjacency(this, node_id_from, node_id_to, op_result)
    class(Graph)::this
    character(len=*), intent(in)::node_id_from, node_id_to
    logical, intent(inout)::op_result
    integer::i, j
    i = -1
    j = -1
    op_result = .false.
    call this%find_node_index_by_id(node_id_from, i)
    call this%find_node_index_by_id(node_id_to, j)
    if(i == -1 .or. j == -1) then
        return
    end if
    if (this%adjacency_matrix(i,j)/=0) then
        op_result = .true.
    end if
end subroutine is_node_adjacency
```

# Процедура поиска в ширину

```
subroutine bfs(this, node_id)
    class(Graph)::this
    character(len=*), intent(in)::node_id
    integer::i, j
    integer,allocatable::adjacency_nodes_index(:)
    logical::op_result
    Type(Array_Queue)::node_deq

    op_result = .false.
    call this%find_node_index_by_id(node_id, i)
    if(i == -1) then
        return
    end if

    call node_deq%init_array_queue()

    call node_deq%enqueue_array_queue(i)
    do
        if (node_deq%queue_size == 0) then
            exit
        end if
        call node_deq%dequeue_array_queue(i, op_result)
        call this%get_adjacency_nodes_index(this%nodes(i)%node_id,adjacency_nodes_index, op_result)
        if(op_result) then
            do j = 1, size(adjacency_nodes_index)
                if (this%nodes(adjacency_nodes_index(j))%node_color == 0) then
                    call node_deq%enqueue_array_queue(adjacency_nodes_index(j))
                end if
            end do
            deallocate(adjacency_nodes_index)
        end if
        this%nodes(i)%node_color = 1
    end do
    call node_deq%clear_array_queue()
    call node_deq%destroy_array_queue()
end subroutine bfs
```

# Процедура для проверки связности графа

```
subroutine repaint_node_to_white_color(this)
    class(Graph)::this
    integer::i
    do i = 1, size(this%nodes)
        if (this%nodes(i)%is_present) then
            this%nodes(i)%node_color = 0
        end if
    end do
end subroutine repaint_node_to_white_color

subroutine is_connected_graph(this, op_result)
    class(Graph)::this
    logical, intent(inout):: op_result
    integer::i
    op_result = .true.
    call this%repaint_node_to_white_color()
    do i = 1, size(this%nodes)
        if (this%nodes(i)%is_present) then
            call this%bfs(this%nodes(i)%node_id)
            exit
        end if
    end do

    do i = 1, size(this%nodes)
        if (this%nodes(i)%is_present .and. this%nodes(i)%node_color == 0) then
            op_result = .false.
            exit
        end if
    end do
    call this%repaint_node_to_white_color()
end subroutine is_connected_graph
```

# Список литературы

1) Джеймс А. Андерсон «Дискретная математика и комбинаторика». Издательский дом «Вильямс», 2004.
2) Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн // Алгоритмы: построение и анализ 3-е издание. — М.: «Вильямс», 2013. — С. 1328. ISBN 978-5-8459-1794-2
3) Роберт Седжвик, Кевин Уэйн «Алгоритмы на java 4-е издание» Пер. с англ. - М. : ООО "И.Д. Вильямс", 2013. ISBN 978-5-8459-1781-2.