

# Problem 2-1

Let  $\mathcal{E}$  be the set of all elements of type  $\mathcal{E}$ , and  $\text{null}$  be an distinguished element to represent `null`. Write formal abstract specifications of the interfaces below with respect to following abstract values:

- A graph is a pair  $(\mathcal{V}, \mathcal{E})$ , where  $\mathcal{V}$  is a set and  $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ .
- A tree is a triple  $(\mathcal{V}, \mathcal{E}, r)$ , where  $(\mathcal{V}, \mathcal{E})$  is a graph and  $r \in \mathcal{V}$  denotes the root.

Other data types, such as `boolean`, `int`, `Set<N>`, etc. have conventional abstract values, e.g., Boolean values, integers, and subsets of  $\mathcal{E}$ , etc.

## Graph<N>

Let  $\mathcal{G}$  be an abstract value of the current graph object.

### Class invariant

#### containsVertex

```
boolean containsVertex(N vertex);
```

- requires: vertex is in  $\mathcal{V}$  and not `null`
- ensures:
  - returns true if vertex is in  $\mathcal{V}$ ; and
  - returns false, otherwise.

#### containsEdge

```
boolean containsEdge(N source, N target);
```

- requires:
  - source is in  $\mathcal{V}$  and not `null`
  - target is in  $\mathcal{V}$  and not `null`
- ensures:
  - returns true if  $(source, target) \in \mathcal{E}$ ; and

- returns false, otherwise.

## getNeighborhood

```
Set<N> getNeighborhood(N vertex);
```

- requires:
  - vertex is in      and not
- ensures:
  - return the set of adjacent vertices of vertex

## Tree<N>

Let                                      be an abstract value of the current graph object.

### Class invariant

## getDepth

```
int getDepth(N vertex);
```

- requires:
  - vertex is in      and not      .
- ensures:
  - returns 0 if vertex is getRoot(); and
  - returns getDepth(getParent(vertex)) + 1, otherwise.

## getHeight

```
int getHeight();
```

- requires: true
- ensures: returns the height of

## getChildren

```
Set<N> getChildren(N vertex);
```

- vertex is in  $V_1$  and not  $V_2$ .

## getParent

```
Optional<N> getParent(N vertex);
```

- requires: vertex is in  $V$  and not  $u$ .

# MutableGraph<N>

Let  $\alpha$  be an abstract value of the current graph object,  
and  $\alpha'$  be an abstract value of the graph object *modified by* the method call.

## Class invariant

## addVertex

```
boolean addVertex(N vertex);
```

- requires: vertex is in `graph` and not `removed`
- ensures:
  - `graph` is a graph; `removed` is a set of vertices
  - `graph` is a graph (the edges are not modified)
  - If `graph` satisfies the class invariant, `removed` also satisfies the class invariant; and
  - returns true if and only if `graph` is a graph.

## removeVertex

```
boolean removeVertex(N vertex);
```

- requires: vertex is in  $V$  and not in  $E$

- ensures:
  - 
  - 
  - If `!isEdge(source, target)` satisfies the class invariant, `addEdge(source, target)` also satisfies the class invariant; and
  - returns true if and only if `addEdge(source, target)` .

## addEdge

```
boolean addEdge(N source, N target);
```

- requires:
  - source is in `vertices` and not in `edges`
  - target is in `vertices` and not in `edges`
- ensures:
  - `addEdge(source, target)` returns true
  - 
  - If `!isEdge(source, target)` satisfies the class invariant, `addEdge(source, target)` also satisfies the class invariant; and
  - returns true if and only if `addEdge(source, target)` .

## removeEdge

```
boolean removeEdge(N source, N target);
```

- requires:
  - source is in `vertices` and not in `edges`
  - target is in `vertices` and not in `edges`
- ensures:
  - `removeEdge(source, target)` returns true (the vertices are not modified)
  - 
  - If `isEdge(source, target)` satisfies the class invariant, `removeEdge(source, target)` also satisfies the class invariant; and
  - returns true if and only if `removeEdge(source, target)` .

## MutableTree<N>

Let `tree` be an abstract value of the current tree object,  
and `newTree` be an abstract value of the tree object *modified by* the method call.

### Class invariant

## addVertex

```
boolean addVertex(N vertex);
```

- requires: vertex is in `V` and not `removed`
- ensures:
  - `vertex` is in `V` ;
  - `edges` (the edges are not modified)
  - `addVertex` will not satisfies the class invariant if `vertex` is already in `V`
  - returns true if and only if `vertex` is added to `V` .

## removeVertex

```
boolean removeVertex(N vertex);
```

- requires: vertex is in `V` and not `removed`
- ensures:
  - throws `IllegalArgumentException` , if `vertex` is not in `V`
  - otherwise
    - Let `E` be the set of edges of `G` that have `vertex` as source or target.
    - `removeVertex` removes `vertex` from `V` and removes all edges in `E` from `E`.
    - If `G` satisfies the class invariant, `G` also satisfies the class invariant; and
    - returns true if and only if `vertex` is removed from `V` .

## addEdge

```
boolean addEdge(N source, N target);
```

- requires:
  - source is in `V` and not `removed`
  - target is in `V` and not `removed`
- ensures:
  - If `G` satisfies the class invariant, `G` also satisfies the class invariant; and
  - if `source` and `target` are not connected by an edge in `G` ,
    - `addEdge` adds an edge from `source` to `target` to `E`.
    - returns true
  - otherwise

- return false

## removeEdge

```
boolean removeEdge(N source, N target);
```

- requires:
  - source is in `graph` and not in `removed`
  - target is in `graph` and not in `removed`
- ensures:
  - Let `graph` be the graph before the call
  - Let `graph'` be the graph after the call
  - Let `removed` be the set of removed vertices
  - `graph'` is the same as `graph` except that the edge between `source` and `target` has been removed
  - If `graph` satisfies the class invariant, `graph'` also satisfies the class invariant; and
  - returns true if and only if the edge between `source` and `target` was present.

# Problem 2-2

Identify whether the abstract interfaces satisfy the Liskov substitution principle.

For each question, explain your reasoning *using the abstract specifications that you have defined in Problem 1*.

## Tree<N> and Graph<N>

- Is `Tree<N>` a subtype of `Graph<N>` ?
  - Answer : Yes
  - Reasons
    - Since performing the methods of `Graph<N>` on `Tree<N>` works same; and
    - class invariants of `Tree<N>` is stronger than `Graph<N>` 's.

## MutableGraph<N> and Graph<N>

- Is `MutableGraph<N>` a subtype of `Graph<N>` ?
  - Answer : Yes
  - Reason
    - After executing `removeVertex` from `MutableGraph<N>` does not guarantee that the result of executing `containsVertex` on both `Graph<N>` and `MutableGraph<N>` are the same.

same.

### **MutableTree<N> and Tree<N>**

- Is `MutableTree<N>` a subtype of `Tree<N>` ?
  - Answer : No
  - Reason
    - `int getDepth(@NotNull N vertex)` : If a vertex added to `MutableTree<N>` by `boolean addVertex(@NotNull N vertex)` with vertex not connected to any vertex, calling `getDepth(vertex)` cannot perform well since the vertex added by `addVertex` does not have path from root to given vertex.

### **MutableTree<N> and MutableGraph<N>**

- Is `MutableTree<N>` a subtype of `MutableGraph<N>` ?
  - Answer : No
  - Reason
    - `boolean addEdge(@NotNull N source, @NotNull N target)` : If a edge between a vertex in      and another vertex not in      is added to `MutableTree<N>` by `addEdge` , `MutableTree<N>` returns false but `MutableGraph<N>` returns true .