Problem 2-1

Let be the set of all elements of type , and 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 , where and .
- A tree is a triple , where is a graph and 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 , etc.

`Graph<N>`

Let be an abstract value of the current graph object.

Class invariant

containsVertex

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

- requires: vertex is in and not
- ensures:
  - returns true if vertex is in ; and
  - returns false, otherwise.

containsEdge

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

- requires:
  - `source` is in and not
  - `target` is in and not
- ensures:
  - returns true if is in ; 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);
```

- requires:
  - vertex is in and not .
- ensures:
  - returns the set of children of vertex in the tree; and
  - returns an empty set if given vertex does not have any children.

getParent

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

- requires: vertex is in and not .
- ensures:
  - returns the parent of vertex; and
  - returns `Optional.empty()` if given vertex does not have parent.

`MutableGraph<N>`

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

Class invariant

addVertex

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

- requires: vertex is in and not
- ensures:
  - ;
  - (the edges are not modified)
  - If satisfies the class invariant, also satisfies the class invariant; and
  - returns true if and only if .

removeVertex

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

- requires: vertex is in and not

- ensures:
  -
  -
  - If satisfies the class invariant, also satisfies the class invariant; and
  - returns true if and only if .

addEdge

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

- requires:
  - source is in and not
  - target is in and not
- ensures:
  -
  -
  - If satisfies the class invariant, also satisfies the class invariant; and
  - returns true if and only if .

removeEdge

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

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

`MutableTree<N>`

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

Class invariant

addVertex

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

- requires: vertex is in and not
- ensures:
  - ;
  - (the edges are not modified)
  - will not satisfies the class invariant if
  - returns true if and only if .

removeVertex

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

- requires: vertex is in and not
- ensures:
  - throws `IllegalArgumentException` , if
  - otherwise
    - Let
    -
    -
    - If satisfies the class invariant, also satisfies the class invariant; and
    - returns true if and only if .

addEdge

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

- requires:
  - source is in and not
  - target is in and not
- ensures:
  - If satisfies the class invariant, also satisfies the class invariant; and
  - if
    -
    -
    - returns true
  - otherwise

        - return false

removeEdge

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

- requires:
    - source is in and not
    - target is in and not
- ensures:
    - Let
    - Let
    - Let
    -
    -
    - If satisfies the class invariant, also satisfies the class invariant; and
    - returns true if and only if .

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 gaurantee that the result of executing `containsVertex` on both `Graph<N>` and `MutableGraph<N>` are

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` .