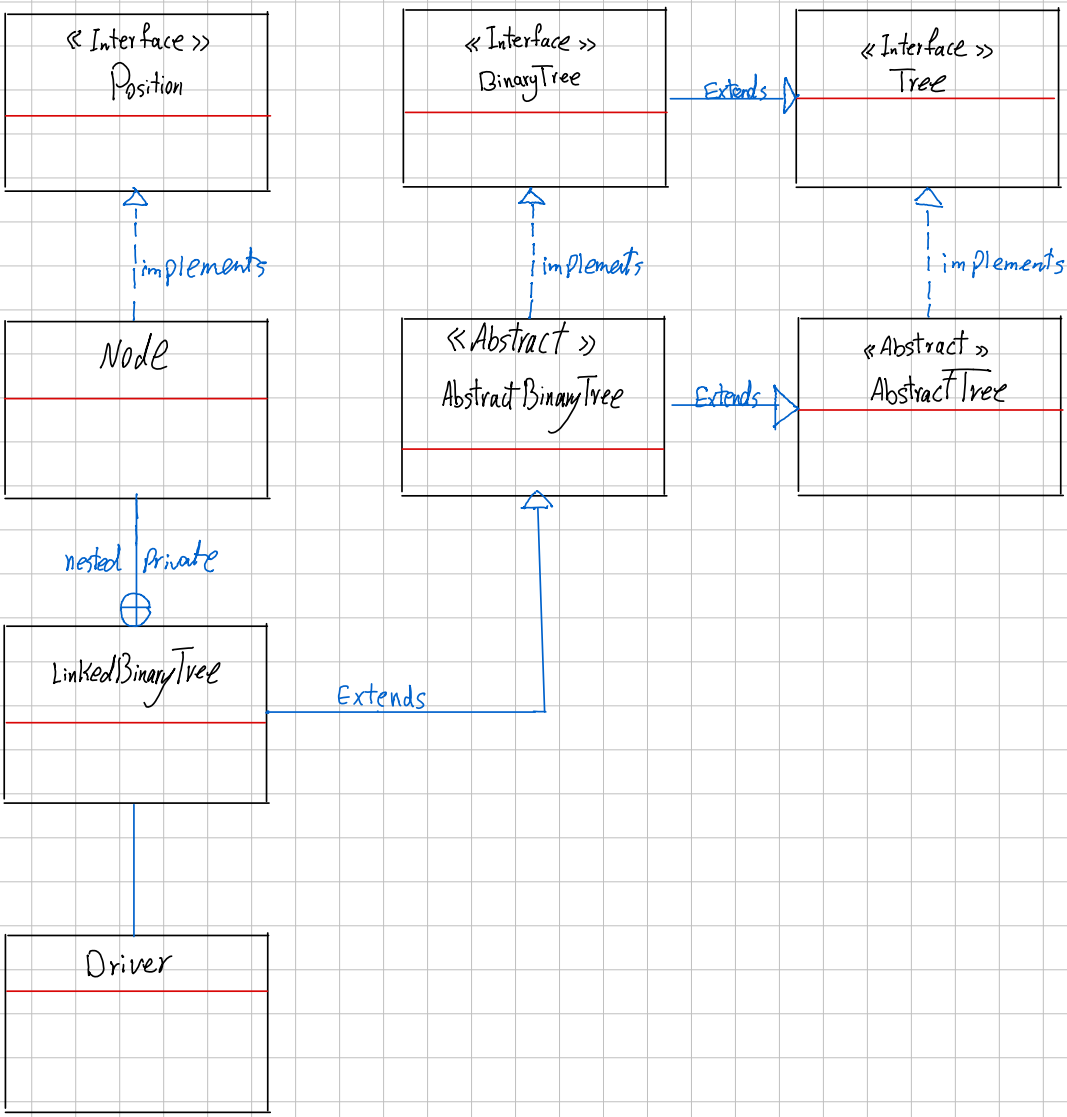


we are given
- Position
- Tree
- BinaryTree

+

use notes for
- AbstractBinaryTree
- AbstractTree

Position Interface

└ wrapper

└ why we need it?

let's look at how it leverages OO Concepts

- Abstraction → hide Node (what is Node)
- Encapsulation → user interact with positions without knowing Node impl (How things done)
- Polymorphism → node "implements" position, so every implementation of Node can use inheritence of Position class
- Modularity → makes it easier to change or modification (Position: handles Contracts, Node: handles specifications)
- Reusability → Can be reused across various Data Structures

Tree interface

- root()
- parent()
- children()
- numChildren()
- isInternal()
- isExternal()
- isRoot()
- isEmpty()
- size()

* implemented by AbstractTree based on the definition of Tree (* in notes)

why the others doesn't have common implementation? They depends on what structure we ended up using as node (which implements position)

what BinaryTree (a type of tree) added to methods?

- left()
- right()
- siblings()

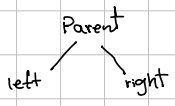
left() and right() again we still don't know how these will work

but for this guy we know the big pic here,

there is something called parent() * from Tree interface that somehow! give us the parent position

+ two method that give us left() and right() position

+ this is "Binary" tree so the structure is



Abstraction OO principle
we know what it does
but we don't know how
(Encapsulation)

by knowing all this we could write Common implementation for

- Sibling() from BinaryTree
- numchildren() } from Tree (that used in BinaryTree)
- children() }

TODO: Complete ...

Q2, Build a LinkedBinaryTree with real names.

Q3, add allDescendants(), recursively show the Children of specified Position.
add pathTO(), Print the path (Node's Elements) from root to the Specified Position.

Q4, Test depth(), height(), size() Amethod defined in LinkedBinaryTree.

Q5, add methods to calculate height and depth based on their definition!

Q1, now we know will use LinkedList ADT

So let's write class LinkedBinaryTree

let's see How we define Node which wrapped with Position (implements Position)

So our node has 4 fields

Private Node<E> Parent
left
right
E element

Encapsulation

now we can write Constructor, Setters and getters

why Node class should be "protected"? I don't know 😅
I guess for subclasses of LinkedBinaryTree. when protected they also could use it. but when Private only is accessible inside the LinkedBinaryTree class.

Q2, what are the fields in LinkedBinaryTree?

- Size: to keep track of size
- root: to keep track of the starting Point of tree

* we have 2 approaches
1, wite defining fields set the to a default values

```
size =0
root = null
Public LinkedBinaryTree() {}
```

2, just define them and later in constructor set the default value

Q3, There Some methods that

return { root of tree
size of tree

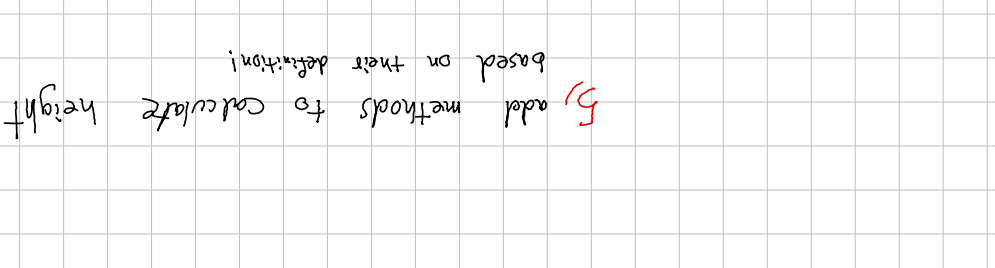
Parent { left of an Specified Position
right

Q4, and Some that

add root to Empty tree
~~size~~ update automatically at an event of adding/removing

~~Parent~~ → doesn't make any sense!

left to a valid Position if it doesn't already have One!

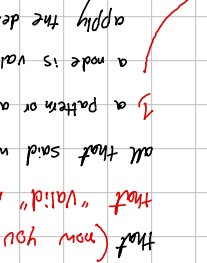


* after this no node in tree will point at the removed node but the use still hold the reference of that node! So we should define how a defunc node looks like and turn the removed node like that So if user try to Pass that to us we could detect that (how you realize what that "Valid" means!)
all that said we need 2 things
1, a pattern or a way of saying a node is valid or not. and apply the defined pattern at the of remove method.
2, a method that "validate" a node to avoid user Passing invalide or removed node

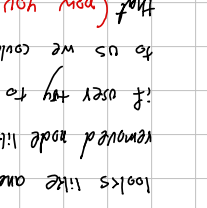
Firstly Ide instance of node
* we're using Position
and Position could be used to wrong other kind of stuff!

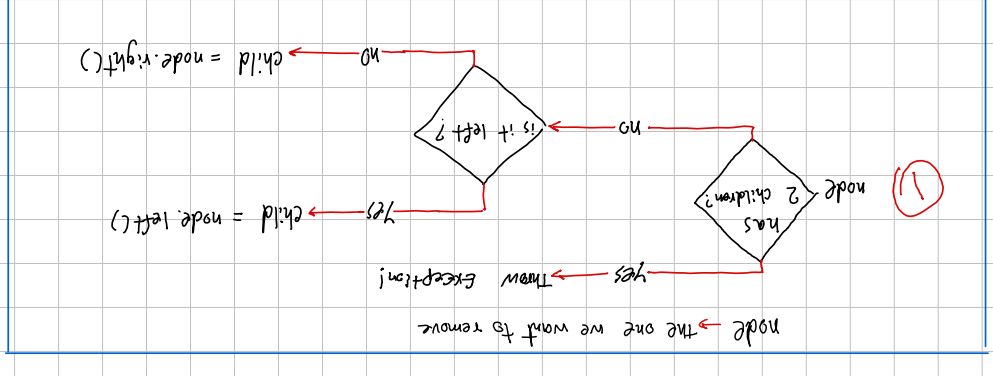
Second: here in notes, teacher defines defunc node as a node that it is its Parent!!

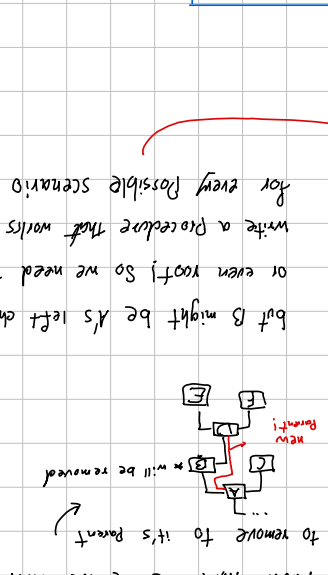
what should we do at the end of removing! to make a node invalid?
node.setParent(node);



apply the First and Second check in every method that get a Position from the user since it is duplicating the same code, wrap it in a method and call that instead.







but B might be A's left child or even root! So we need to write a Procedure that works for every Possible Scenario