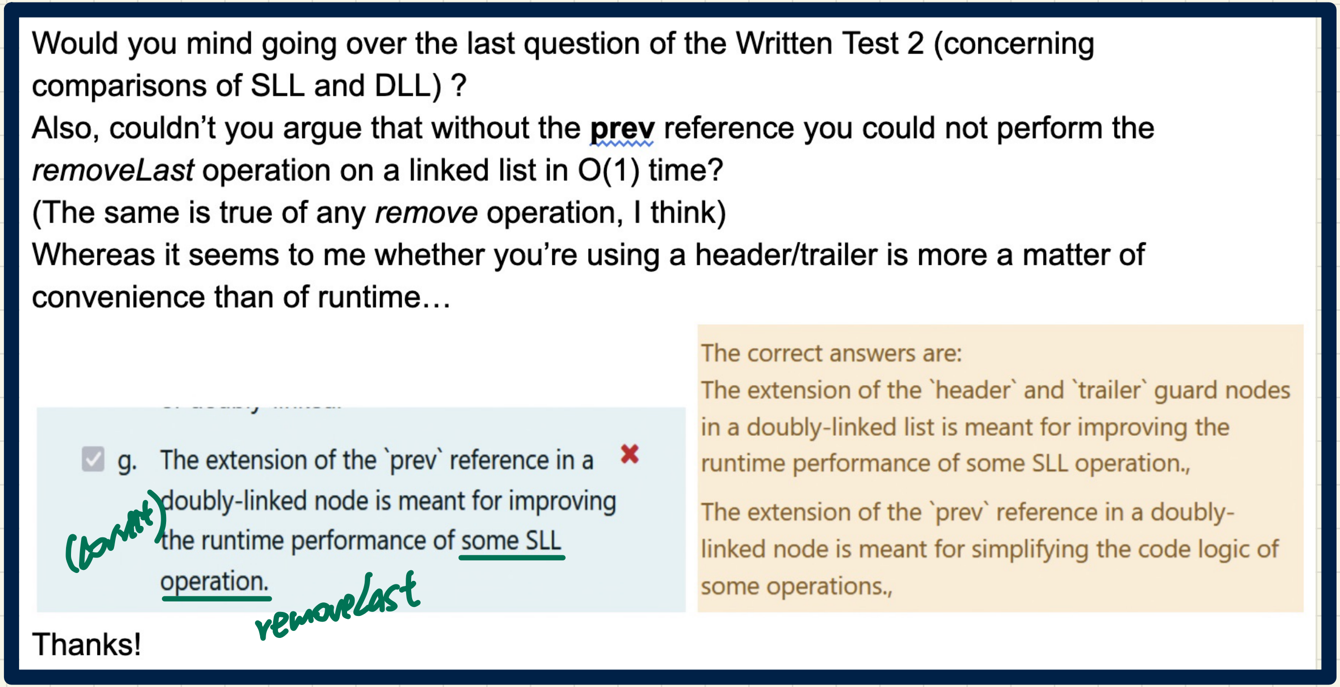
EECS2011 Fundamentals of Data Structures (Winter 2022)

Q&A - Lectures 10

Wednesday, March 30

Announcements

- Lecture W11 to released
    + ✓ Balanced Binary Search Trees
    + Tree Rotations
- Assignment 3 released
- Programming Test 2 coming soon (guide released)
- ProgTest1 grades still being processed...



Would you mind going over the last question of the Written Test 2 (concerning comparisons of SLL and DLL) ?

Also, couldn't you argue that without the **prev** reference you could not perform the *removeLast* operation on a linked list in O(1) time?

(The same is true of any *remove* operation, I think)

Whereas it seems to me whether you're using a header/trailer is more a matter of convenience than of runtime…

of doubly-linked.

☑ g. The extension of the `prev` reference in a doubly-linked node is meant for improving the runtime performance of some SLL operation. ✗

(const)

removeLast

The correct answers are:

The extension of the `header` and `trailer` guard nodes in a doubly-linked list is meant for improving the runtime performance of some SLL operation.,

The extension of the `prev` reference in a doubly-linked node is meant for simplifying the code logic of some operations.,

Thanks!

Hi professor.
I was not able to completely understand the question or to derive an argument for the exercise in slide 19 (~min 10th in the video).
Can you explain it or discuss the answer please? Thanks

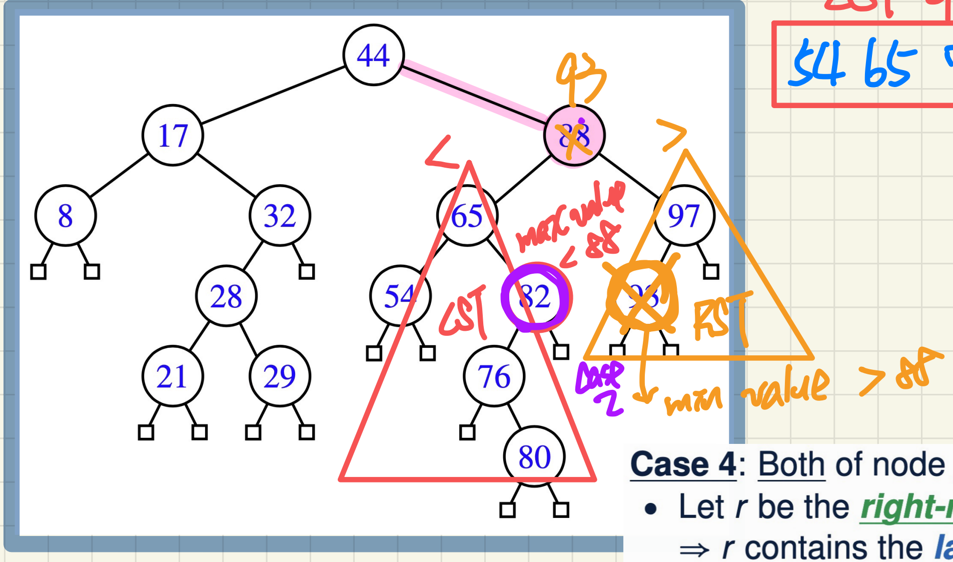
Case 4.2: Delete Entry with Key 88

LST of 88

54 65 76 80 82 ~~88~~ 93 97

RST of 88

<



Case 4: Both of node *p*'s children are *internal*.

- Let *r* be the ***right-most internal node*** *p*'s ***LST***.

  ⇒ *r* contains the ***largest key s.t. key(r) < key(p)***.

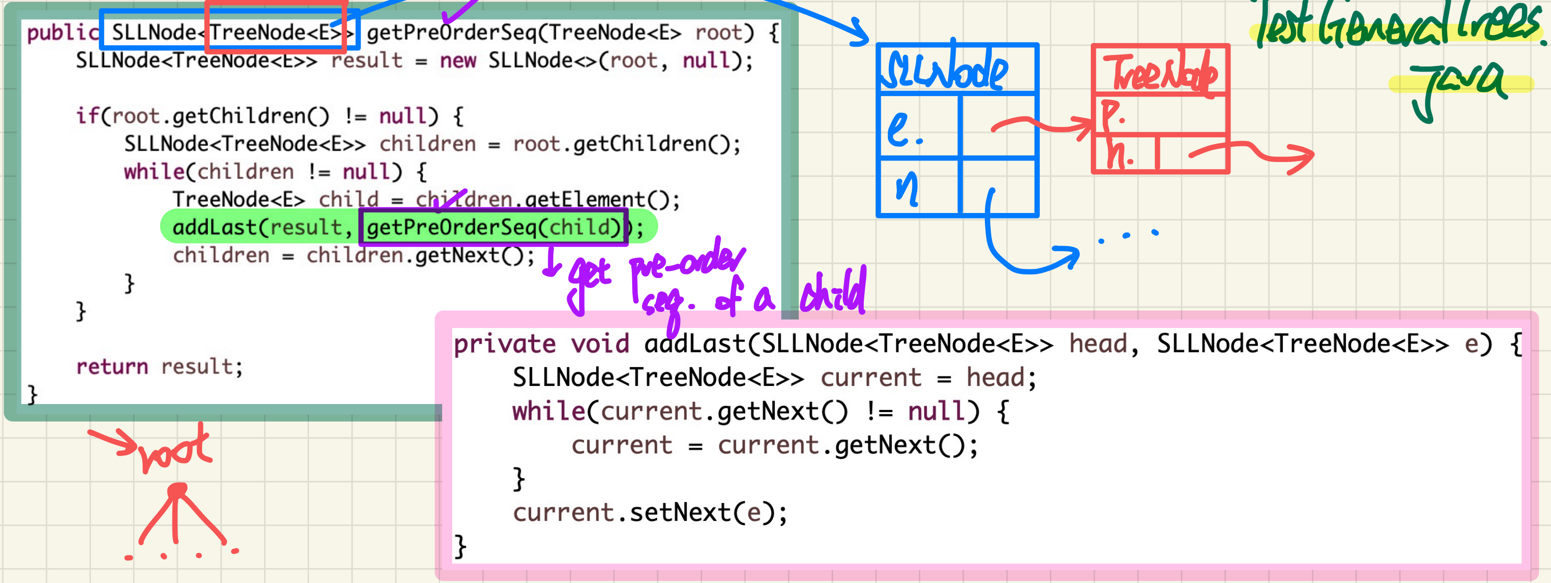
✓**Exercise**: Can *r* contain the ***smallest key s.t. key(r) > key(p)***?

In-Order Traversal:

If there's time can u please demonstrate in Q/A session how to implement this method using arrays or SLL?

I am having problems understanding using them with recursion

EECS4302



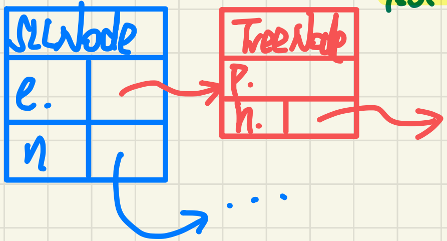
elem. of each node

```java
public SLLNode<TreeNode<E>> getPreOrderSeq(TreeNode<E> root) {
    SLLNode<TreeNode<E>> result = new SLLNode<>(root, null);

    if(root.getChildren() != null) {
        SLLNode<TreeNode<E>> children = root.getChildren();
        while(children != null) {
            TreeNode<E> child = children.getElement();
            addLast(result, getPreOrderSeq(child));
            children = children.getNext();
        }
    }

    return result;
}
```

get pre-order seq. of a child

root

```java
private void addLast(SLLNode<TreeNode<E>> head, SLLNode<TreeNode<E>> e) {
    SLLNode<TreeNode<E>> current = head;
    while(current.getNext() != null) {
        current = current.getNext();
    }
    current.setNext(e);
}
```

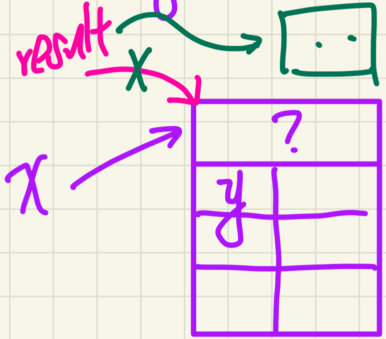


TestGeneralTrees.java

Call by value

```
void m ( Object result , ... ) {
      *                             **
    ① result = ... ;              ② result . setElement ( ... );

}
```

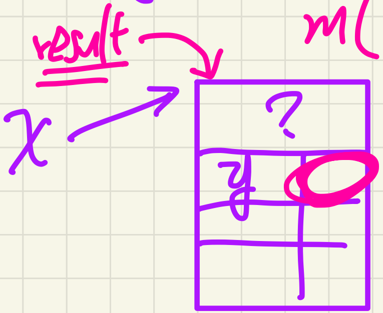
* Object x = ... ;



m (x , ...) ;

** Object x = ... ;

m (x , ---)

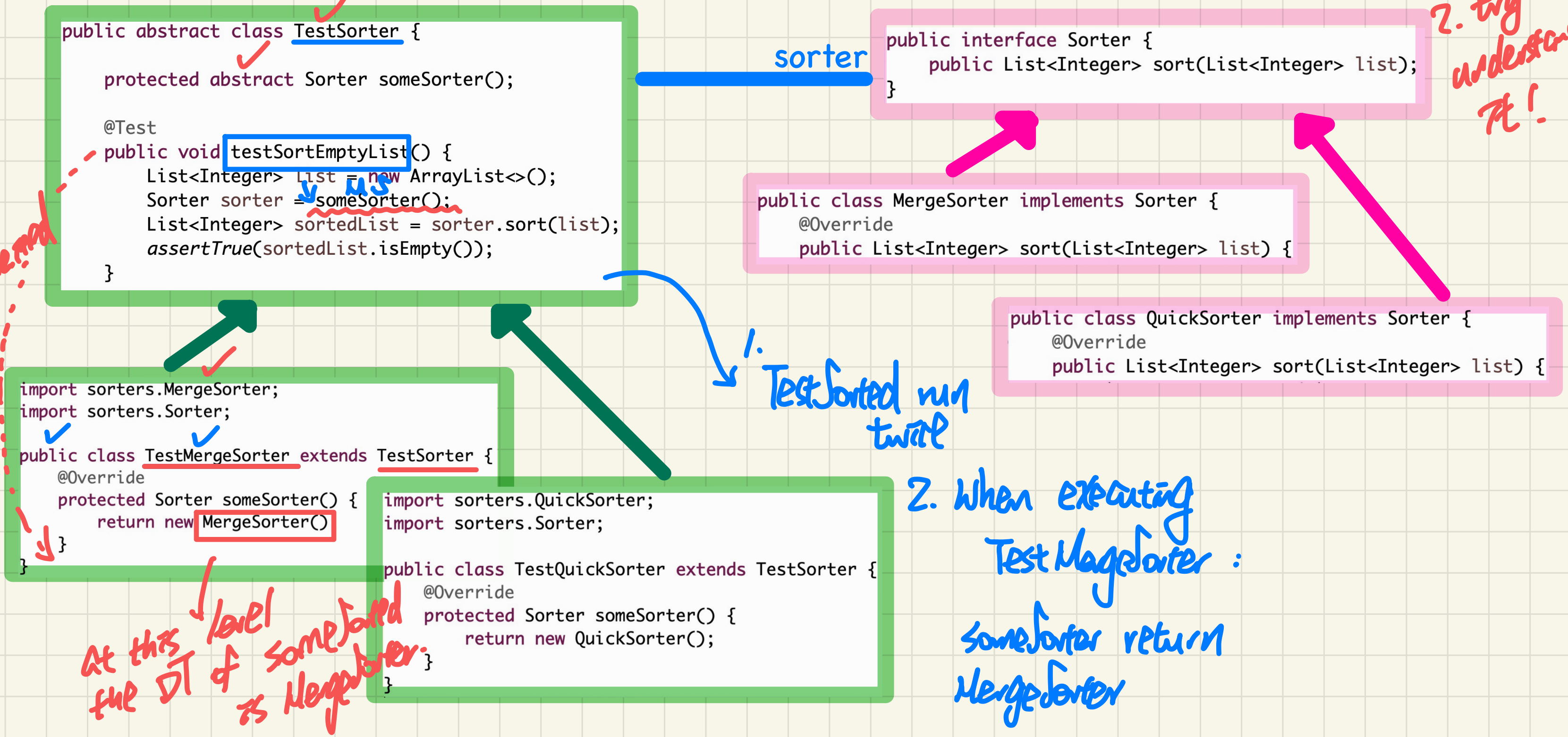


Professor, I am a little confused about how all those different classes interact with each other for junit_testing.
For example, how were you able to return MergeSorter() class?
Would you mind going over them quickly please?

factory method design pattern

↳ 1. not covered in EECS

2. try understanding it!



```java
public abstract class TestSorter {

    protected abstract Sorter someSorter();

    @Test
    public void testSortEmptyList() {
        List<Integer> list = new ArrayList<>();
        Sorter sorter = someSorter();
        List<Integer> sortedList = sorter.sort(list);
        assertTrue(sortedList.isEmpty());
    }
```

```java
public interface Sorter {
    public List<Integer> sort(List<Integer> list);
}
```

```java
public class MergeSorter implements Sorter {
    @Override
    public List<Integer> sort(List<Integer> list) {
```

```java
public class QuickSorter implements Sorter {
    @Override
    public List<Integer> sort(List<Integer> list) {
```

```java
import sorters.MergeSorter;
import sorters.Sorter;

public class TestMergeSorter extends TestSorter {
    @Override
    protected Sorter someSorter() {
        return new MergeSorter()
    }
}
```

```java
import sorters.QuickSorter;
import sorters.Sorter;

public class TestQuickSorter extends TestSorter {
    @Override
    protected Sorter someSorter() {
        return new QuickSorter();
    }
}
```

1. TestSorter run twice

2. When executing TestMergeSorter: someSorter return MergeSorter

At this level the DT of someSorter is MergeSorter.

ProgTest 2

- will be given 1~2 methods to implement
- 1 q. will be close to the level of difficulty of examples.
- 1 q. will be about recursion on tree
  ↳ may have to use SLLNode.

1. leetcode
2. A2 solution