Exercise: Mixing the "Best" from both Sorts?

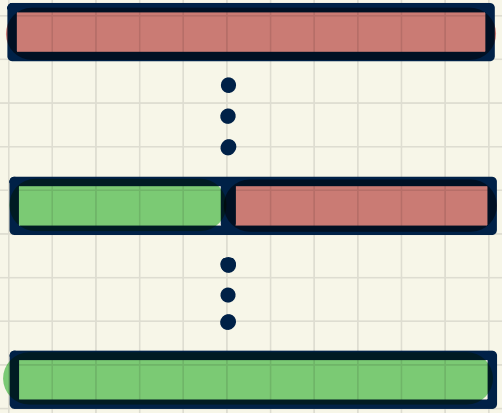
Recall:

- In insertion sort, costs of insertions are increasing.
- In selection sort, costs of selections are decreasing.

Idea:

- Perform insertion sort until half of the input is sorted.
- Perform selection sort to finish sorting the remaining half.

Q: Will this "new" algorithm perform better than $O(n^2)$?

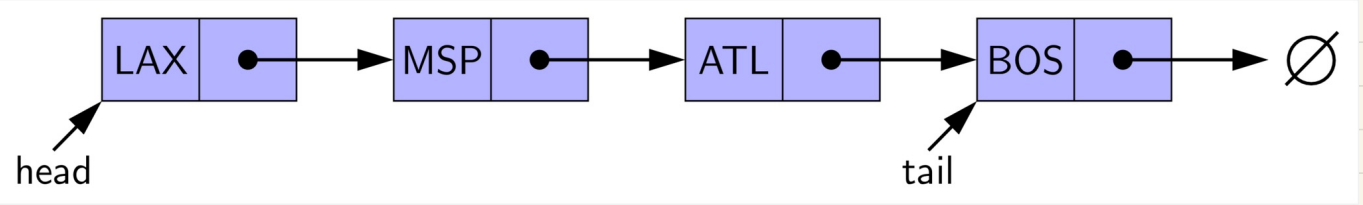


Singly-Linked Lists (SLL): Visual Introduction

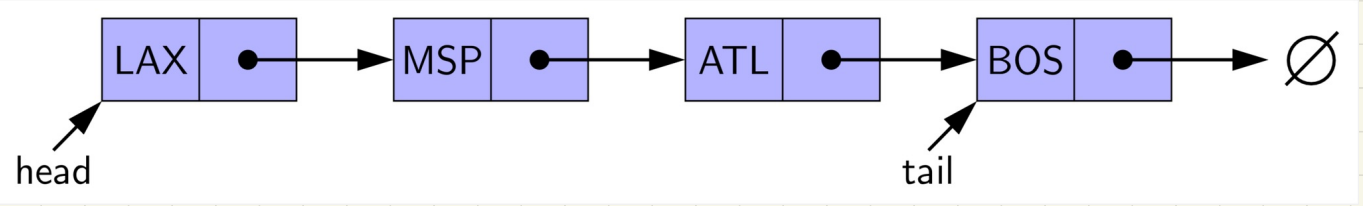
- A chain of connected nodes (via aliasing)
- Each node contains:
  - + reference to a data object
  - + reference to the next node
- Head vs. Tail
- The chain may grow or shrink dynamically.
- Accessing a position in a linear collection:
  - + Array uses absolute indexing: O(1)
  - + SLL uses relative positioning: O(n)

A SLL Grows or Shrinks Dynamically

e.g., **Inserting** TOR/VAN/MON to the **beginning**/**middle**/**end**.



e.g., **Removing** LAX/ATL/BOS from the **beginning**/**middle**/**end**.



Implementing SLL in Java: SinglyLinkedList vs. Node

```
public class SinglyLinkedList {
   private Node head = null;
   public void setHead(Node n) { head = n; }
   public int getSize() { ... }
   public Node getTail() { ... }
   public void addFirst(String e) { ... }
   public Node getNodeAt(int i) { ... }
   public void addAt(int i, String e) { ... }
   public void removeLast() { ... }
}
```

Runtime

```
public class Node {
   private String element;
   private Node next;
   public Node(String e, Node n) { element = e; next = n; }
   public String getElement() { return element; }
   public void setElement(String e) { element = e; }
   public Node getNext() { return next; }
   public void setNext(Node n) { next = n; }
}
```

SLL: Constructing a Chain of Nodes

```
public class Node {
   private String element;
   private Node next;
   public Node(String e, Node n) { element = e; next = n; }
   public String getElement() { return element; }
   public void setElement(String e) { element = e; }
   public Node getNext() { return next; }
   public void setNext(Node n) { next = n; }
}
```

Approach 1

```
Node tom = new Node("Tom", null);
Node mark = new Node("Mark", tom);
Node alan = new Node("Alan", mark);
```

SLL: Constructing a Chain of Nodes

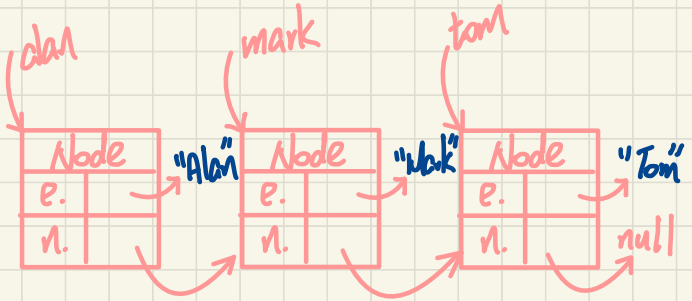
```
public class Node {
   private String element;
   private Node next;
   public Node(String e, Node n) { element = e; next = n; }
   public String getElement() { return element; }
   public void setElement(String e) { element = e; }
   public Node getNext() { return next; }
   public void setNext(Node n) { next = n; }
}
```

Approach 2

```
Node alan = new Node("Alan", null);
Node mark = new Node("Mark", null);
Node tom = new Node("Tom", null);
alan.setNext(mark);
mark.setNext(tom);
```

SLL: Setting a List's Head to a Chain of Nodes

```
public class SinglyLinkedList {
   private Node head = null;
   public void setHead(Node n) { head = n; }
   public int getSize() { ... }
   public Node getTail() { ... }
   public void addFirst(String e) { ... }
   public Node getNodeAt(int i) { ... }
   public void addAt(int i, String e) { ... }
   public void removeLast() { ... }
}
```

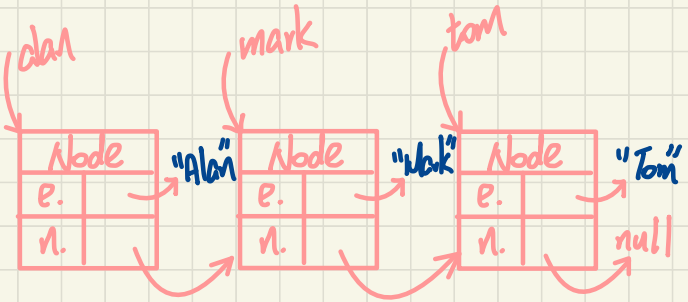


Approach 1

```
Node tom = new Node("Tom", null);
Node mark = new Node("Mark", tom);
Node alan = new Node("Alan", mark);
SinglyLinkedList list = new SinglyLinkedList();
list.setHead(alan);
```

SLL: Setting a List's Head to a Chain of Nodes

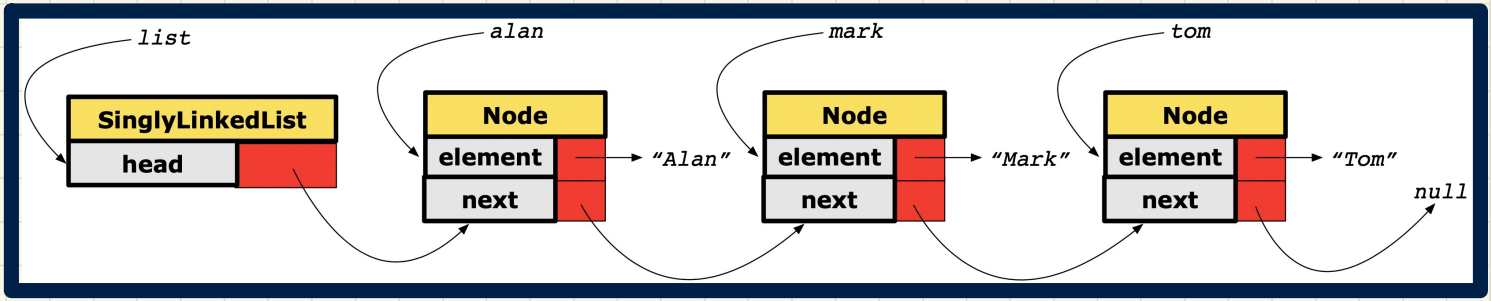
```
public class SinglyLinkedList {
   private Node head = null;
   public void setHead(Node n) { head = n; }
   public int getSize() { ... }
   public Node getTail() { ... }
   public void addFirst(String e) { ... }
   public Node getNodeAt(int i) { ... }
   public void addAt(int i, String e) { ... }
   public void removeLast() { ... }
}
```



Approach 2

```
Node alan = new Node("Alan", null);
Node mark = new Node("Mark", null);
Node tom = new Node("Tom", null);
alan.setNext(mark);
mark.setNext(tom);
SinglyLinkedList list = new SinglyLinkedList();
list.setHead(alan);
```

SLL Operation: Counting the Number of Nodes

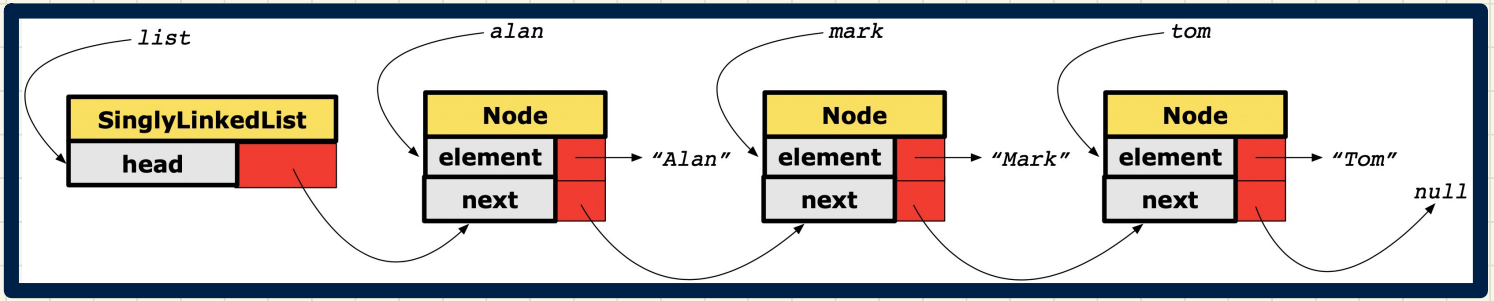



```
1  int getSize() {
2     int size = 0;
3     Node current = head;
4     while (current != null) {
5        current = current.getNext();
6        size ++;
7     }
8     return size;
9  }
```

Trace: list.getSize()

current	current != null	End of Iteration	size

SLL Operation: Finding the Tail of the List




```
1  Node getTail() {
2     Node current = head;
3     Node tail = null;
4     while (current != null) {
5        tail = current;
6        current = current.getNext();
7     }
8     return tail;
9  }
```

Trace: list.getTail()

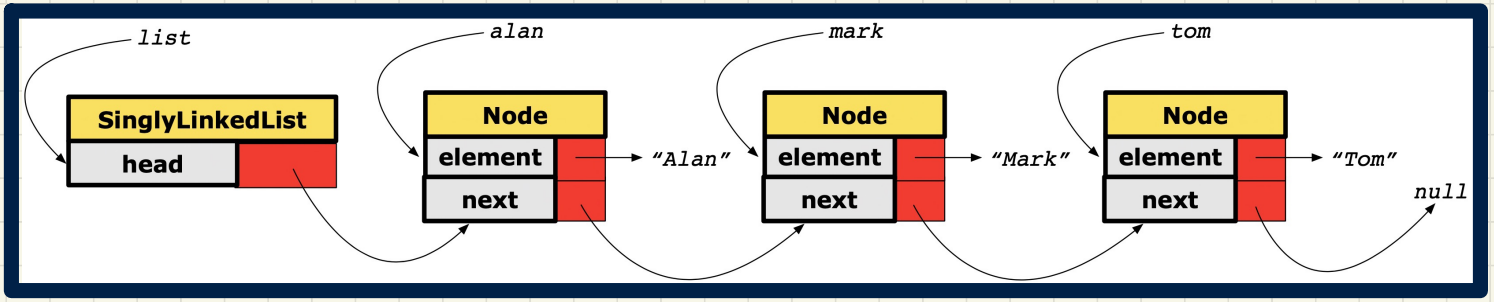
current	current != null	End of Iteration	tail

SLL Operation: Inserting to the Front of the List

```java
@Test
public void testSLL_02() {
    SinglyLinkedList list = new SinglyLinkedList();
    assertTrue(list.getSize() == 0);
    assertTrue(list.getFirst() == null);

    list.addFirst("Tom");
    list.addFirst("Mark");
    list.addFirst("Alan");
    assertTrue(list.getSize() == 3);
    assertEquals("Alan", list.getFirst().getElement());
    assertEquals("Mark", list.getFirst().getNext().getElement());
    assertEquals("Tom", list.getFirst().getNext().getNext().getElement());
}
```

```java
1  void addFirst (String e) {
2    head = new Node(e, head);
3    if (size == 0) {
4      tail = head;
5    }
6    size ++;
7  }
```

SLL Operation: Accessing the Middle of the List



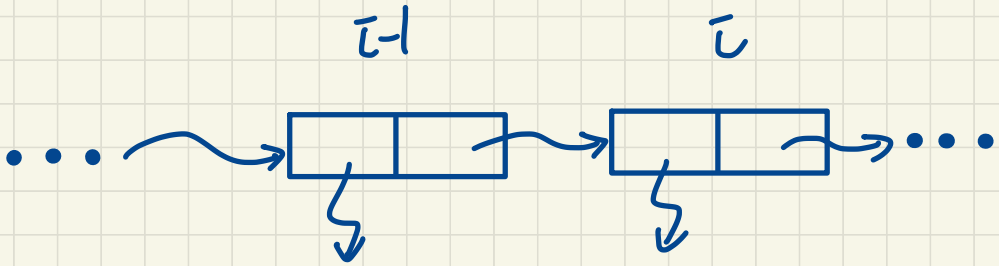

```
1   Node getNodeAt (int i) {
2     if (i < 0 || i >= size) { /* error
3     else {
4       int index = 0;
5       Node current = head;
6       while (index < i) { /* exit when
7         index ++;
8         current = current.getNext();
9       }
10      return current;
11    }
12  }
```

Trace: list.getNodeAt(2)

current	index	index < 2	Start of Iteration

Idea of Inserting a Node at index i

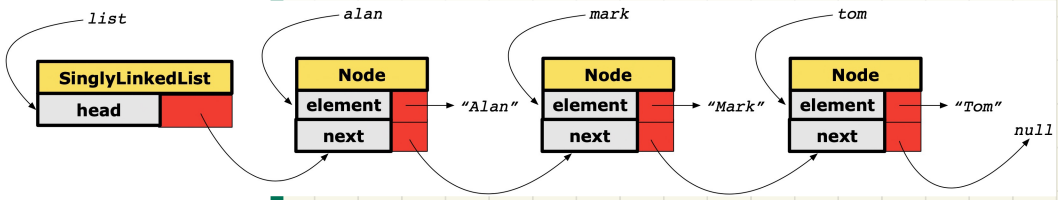
Case: addAt(i, e), where i > 0



e → "..."

SLL Operation: Inserting to the Middle of the List

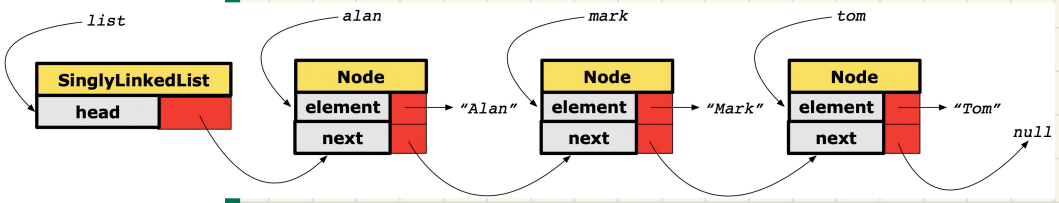
```java
@Test
public void testSLL_addAt() {
    SinglyLinkedList list = new SinglyLinkedList();
    assertTrue(list.getSize() == 0);
    assertTrue(list.getFirst() == null);

    list.addFirst("Tom");
    list.addFirst("Mark");
    list.addFirst("Alan");
    assertTrue(list.getSize() == 3);

    list.addAt(0, "Suyeon");
    list.addAt(2, "Yuna");
    assertTrue(list.getSize() == 5);
    list.addAt(list.getSize(), "Heeyeon");
    assertTrue(list.getSize() == 6);
    assertEquals("Suyeon", list.getNodeAt(0).getElement());
    assertEquals("Alan", list.getNodeAt(1).getElement());
    assertEquals("Yuna", list.getNodeAt(2).getElement());
    assertEquals("Mark", list.getNodeAt(3).getElement());
    assertEquals("Tom", list.getNodeAt(4).getElement());
    assertEquals("Heeyeon", list.getNodeAt(5).getElement());
}
```




```java
1  void addAt (int i, String e) {
2    if (i < 0 || i > size) {
3      throw new IllegalArgumentException("Invalid Index.");
4    }
5    else {
6      if (i == 0) {
7        addFirst(e);
8      }
9      else {
10       Node nodeBefore = getNodeAt(i - 1);
11       Node newNode = new Node(e, nodeBefore.getNext());
12       nodeBefore.setNext(newNode);
13       size ++;
14     }
15   }
16 }
```

SLL Operation: Removing the End of the List

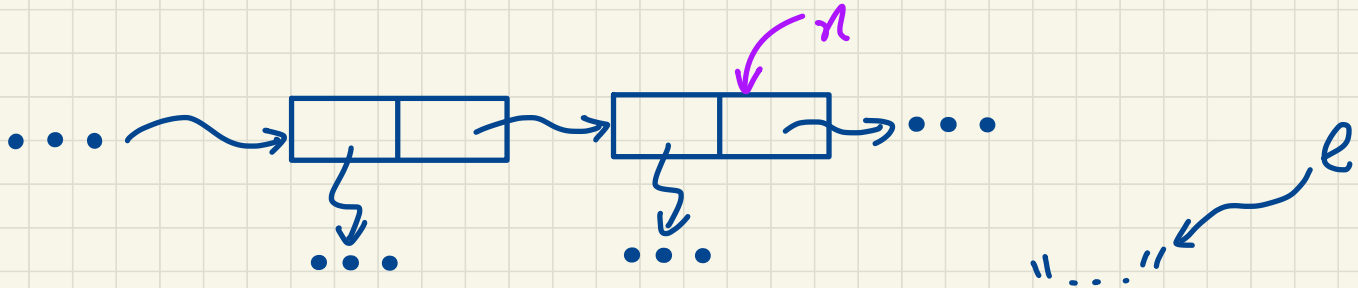
```java
@Test
public void testSLL_removeLast() {
    SinglyLinkedList list = new SinglyLinkedList();
    assertTrue(list.getSize() == 0);
    assertTrue(list.getFirst() == null);

    list.addFirst("Tom");
    list.addFirst("Mark");
    list.addFirst("Alan");
    assertTrue(list.getSize() == 3);

    list.removeLast();
    assertTrue(list.getSize() == 2);
    assertEquals("Alan", list.getNodeAt(0).getElement());
    assertEquals("Mark", list.getNodeAt(1).getElement());

    list.removeLast();
    assertTrue(list.getSize() == 1);
    assertEquals("Alan", list.getNodeAt(0).getElement());

    list.removeLast();
    assertTrue(list.getSize() == 0);
    assertNull(list.getFirst());
}
```




```java
1  void removeLast () {
2    if (size == 0) {
3      throw new IllegalArgumentException("Empty List.");
4    }
5    else if (size == 1) {
6      removeFirst();
7    }
8    else {
9      Node secondLastNode = getNodeAt(size - 2);
10     secondLastNode.setNext(null);
11     tail = secondLastNode;
12     size --;
13   }
14 }
```

Exercises: insertAfter vs. insertBefore

Case: insertAfter(Node n, String e)



Case: insertBefore(Node n, String e)