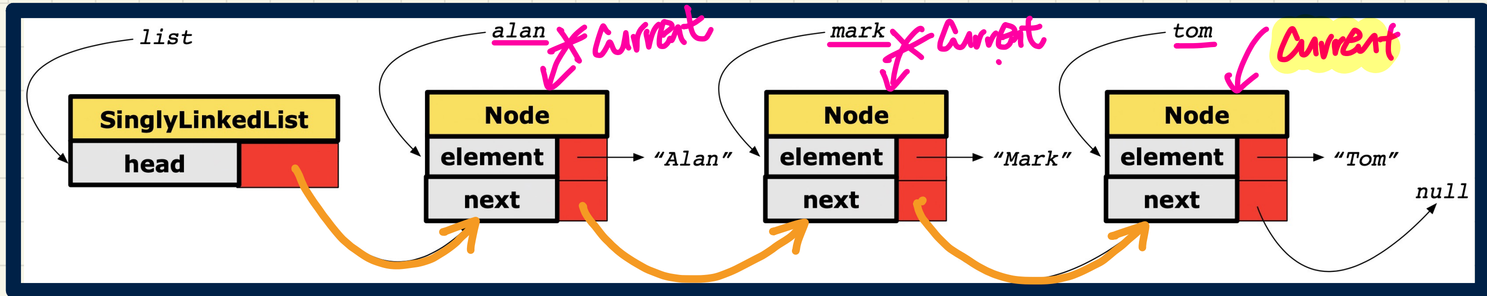
Lecture 2

Part E (continued)

Singly-Linked Lists -
Java Implementation: String Lists
Operations on a List

Exercises on Non-Empty list: lst.getNodeAt(-1) vs. lst.getNodeAt(3)

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  }
```

O(1)

Exit: index >= 2

O(1)

O(1)

O(1)

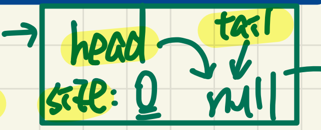
Max # of iterations: Exit: index == i → max: size − 1 (n)

Trace: list.getNodeAt(2)

current	index	index < 2	Start of Iteration
alan	0	T	1
mark	1	T	2
tom	2	F	
		O(n)	

No valid indices

1. Empty list
2. Non-Empty list

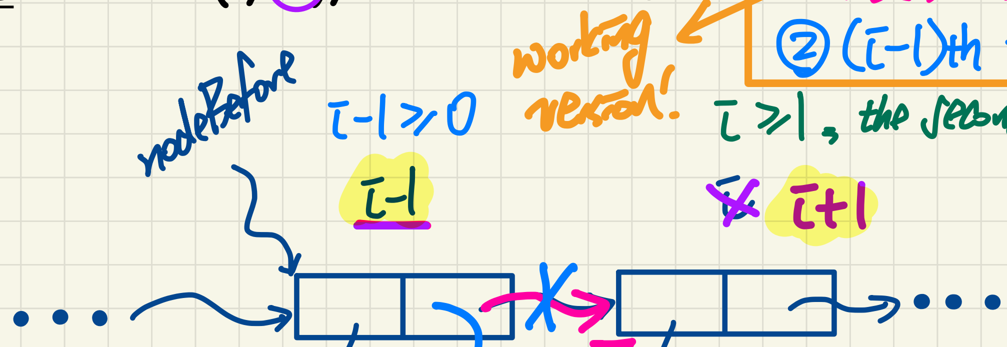


→ lst.getNodeAt(0); 0<0 || 0 >= 0 → T

Idea of Inserting a Node at index i

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

① The newNode's next is set to (i-1)th node's next.
② (i-1)th node's nex set to newNode.



i ≥ 1, the second node onwards.

seq. of code does not work!

① The (i-1)th node's next is set to the newNode
② The newNode's next is set to the old ith node.

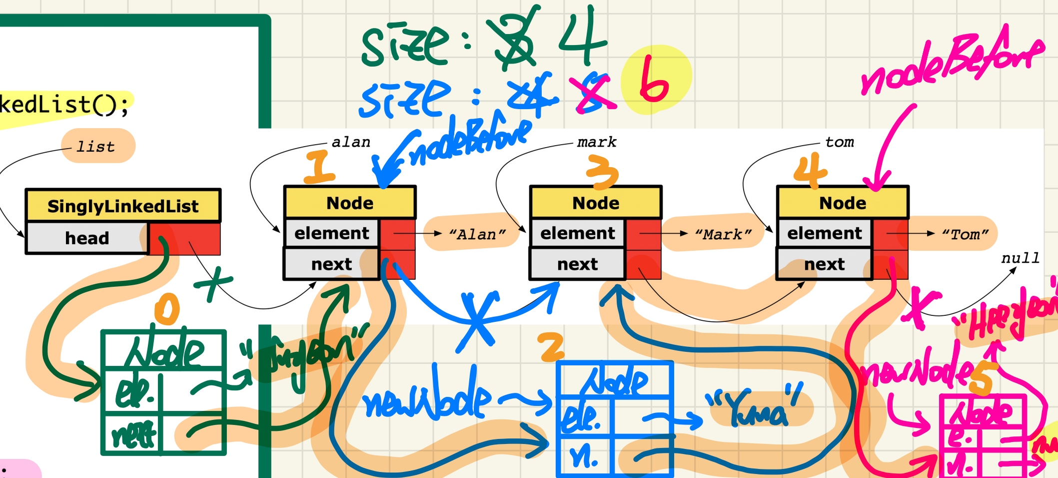
e → "..."

Q. Do we need the reference to the (i-1)th node?

YES!

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());
}
```



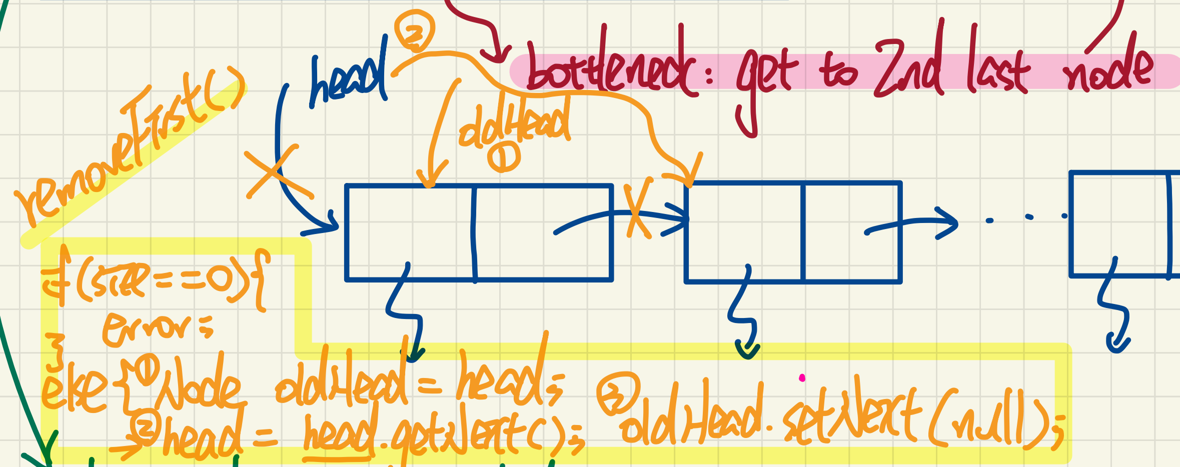

```java
1  void addAt (int i, String e) {
2    if (i < 0 || i > size) {
3      throw new IllegalArgumentException("Invalid Index.");
4    }
5    else {
6      if (i == 0) {
7        addFirst(e);
8      }
9      else {
10       Node nodeBefore = getNodeAt(i - 1);
11       Node newNode = new Node(e, nodeBefore.getNext());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

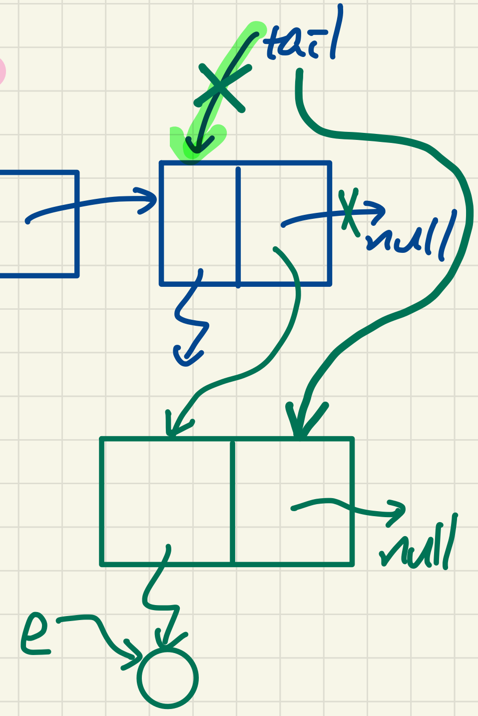
O(1)? O(n)

O(n) — dominated by getNodeAt

void addLast(String e) → O(1)

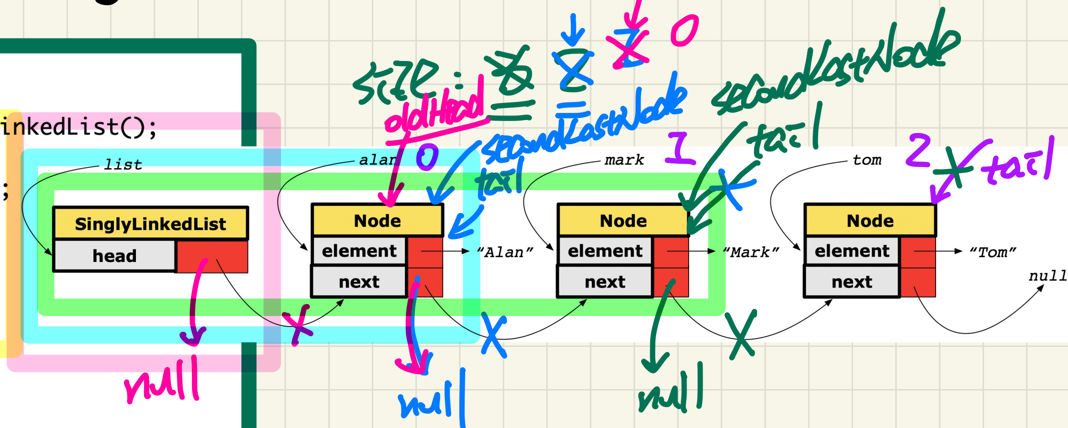
void removeLast() → O(n)



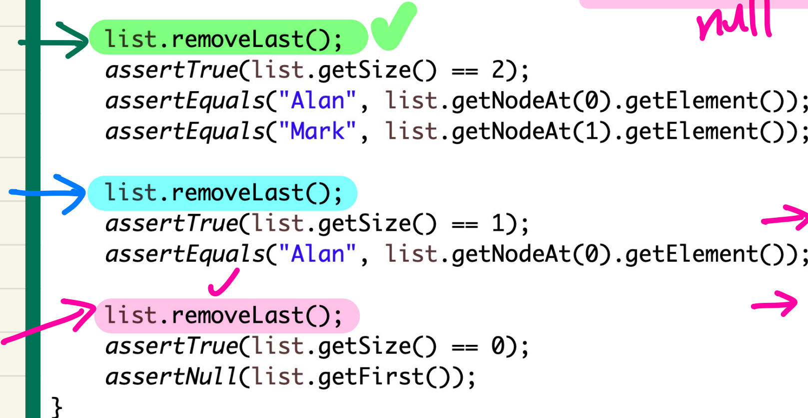



```
tail.setNext(new Node(e, null));
tail = tail.getNext();
size++;
```

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);
```



```java
    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  }
```

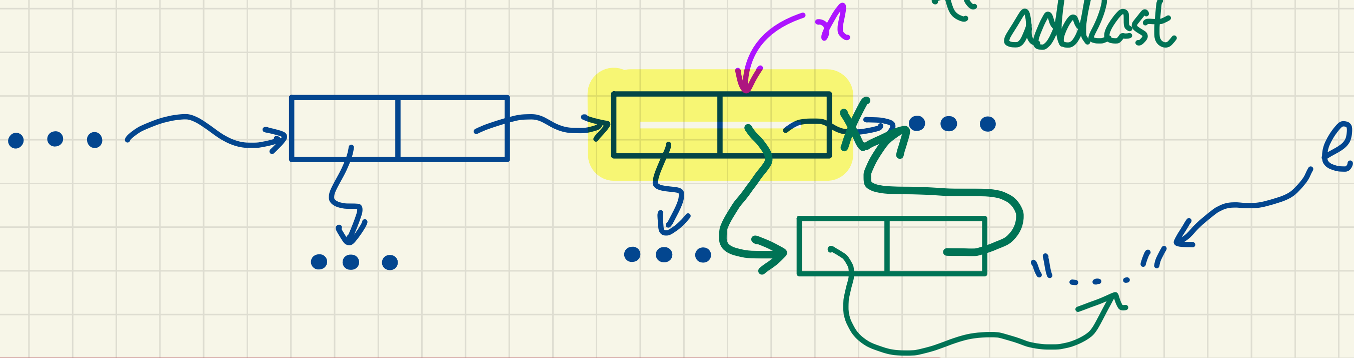
O(n) - dominating term

Exercises: insertAfter vs. insertBefore

reference node

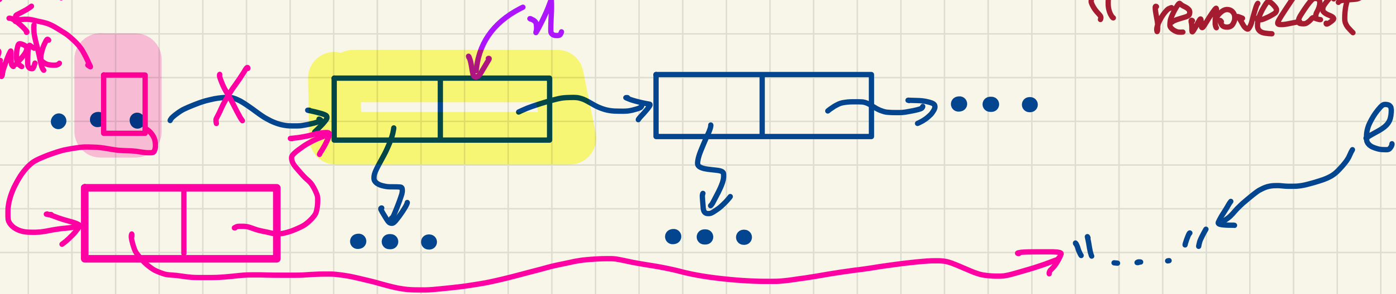
Case: insertAfter(Node n, String e)

O(1) ≈ addLast



Case: insertBefore(Node n, String e)

O(n) ≫ removeLast



Lecture 2

Part F

Singly-Linked Lists -
Comparing Arrays and Singly-Linked Lists

Running Time: Arrays vs. Singly-Linked Lists

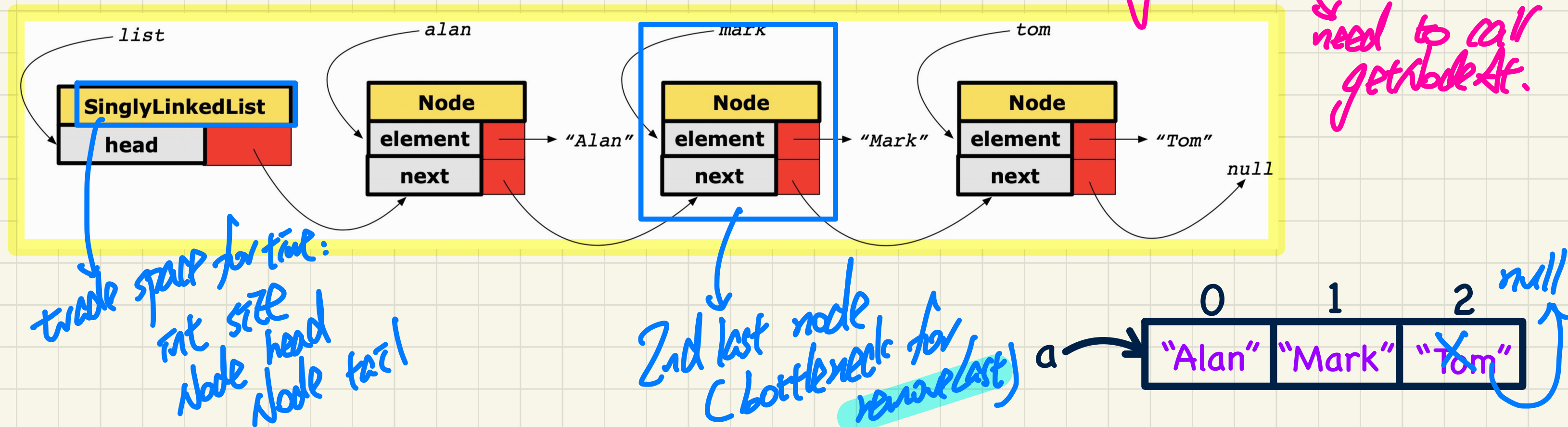
Operation \ Data Structure		Array	Singly-Linked List
get size			O(1)
get first/last element			O(1)
get element at index i		O(1)	O(n)
remove last element		O(1)	O(n)
add/remove first element, add last element		O(n)	O(1)
add/remove i^{th} element	given reference to $(i-1)^{th}$ element	O(n)	O(1)
	not given	O(n)	O(n)

no resizing

resizing

no need to call getNodeAt

need to call getNodeAt.



Lecture 2

Part G

Singly-Linked Lists -
Implementing Generic Lists in Java

Non-Generic Classes: Node vs. SinglyLinkedList

→ as an implementor, you must commit to a type for node elements

```
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; }
}
```

Everything else is tied to this String type accordingly.

```
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() { ... }
}
```

```
Node n1 = new Node("Alan", null);
Node n2 = new Node("Mark", n1);
Node n3 = new Node(23, null);
SLL list = new SLL();
list.setHead(n2);
list.addAt(0, "Tom");
list.addAt(1, 23);
Node n4 = list.getNodeAt(1);
String e = n4.getElement();
```

`Node(23, null)` ✗ → inconsistent with the committed String type.

`list.addAt(1, 23)` ✗ → not compatible

Generic Classes: Node and SinglyLinkedList

As an implementer, we leave the choice of element type to whoever uses these classes for declarations.

declaration of a generic param.

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

```java
public class SinglyLinkedList<E> {
  private Node<E> head;
  private Node<E> tail;
  private int size;
  public void setHead(Node<E> n) { head = n; }
  public void addFirst(E e) { ... }
  Node<E> getNodeAt (int i) { ... }
  void addAt (int i, E e) { ... }
}
```

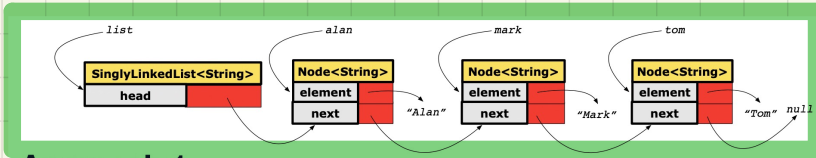
```java
Node<String> n1 = new Node<>("Alan", null);
Node<String> n2 = new Node<>("Mark", n1);
Node<Integer> n3 = new Node<>(23, null);
Node<Integer> n4 = new Node<>(46, n3);
Node<Integer> n5 = new Node<>("Tom", null);
Node<Integer> n6 = new Node<>(46, n2);

SLL<String> list1 = new SLL<>();
list1.setHead(n2);
list1.addAt(0, "Tom");
Node<String> n7 = list1.getNodeAt(1);
String e1 = n7.getElement();

SLL<Integer> list2 = new SLL<>();
list2.setHead(n4);
list2.addAt(0, 68);
Node<Integer> n8 = list2.getNodeAt(1);
Integer e2 = n8.getElement();
```

expecting Node<Integer> but getting Node<String>

Node< E > → generic type parameter

list2.setHead(n1) ✗ — expecting Node<I> but getting Node<S>

List Constructions

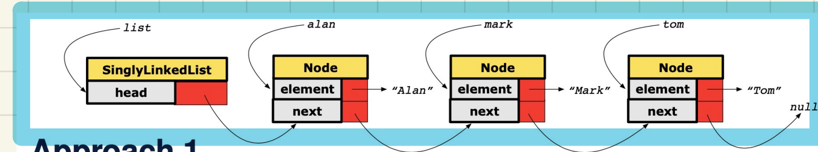


Approach 1

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

Approach 2

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



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);
```

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);
```

Generic List

Non-Generic List

```
Node<String> alan = new Node<String>(...);
                    new Node<>(...);
```

List Methods

class SLL<E SP>{

SLL<String>

SLL<Person>

Generic List

```
void addFirst (E P e) {
  head = new Node<E P>(e, head);
  if (size == 0) { tail = head; }
  size ++;
}
```

```
Node<E P> getNodeAt (int i) {
  if (i < 0 || i >= size) {
    throw new IllegalArgumentExcept
  else {
    int index = 0;
    Node<E P> current = head;
    while (index < i) {
      index ++;
      current = current.getNext();
    }
    return current;
  }
}
```

}

Non-Generic List

```
void addFirst (String e) {
  head = new Node(e, head);
  if (size == 0) {
    tail = head;
  }
  size ++;
}
```

```
Node getNodeAt (int i) {
  if (i < 0 || i >= size) { /* error
  else {
    int index = 0;
    Node current = head;
    while (index < i) { /* exit when
      index ++;
      current = current.getNext();
    }
    return current;
  }
}
```