Lecture 2

Part H

Doubly-Linked Lists - Intuitive Introduction

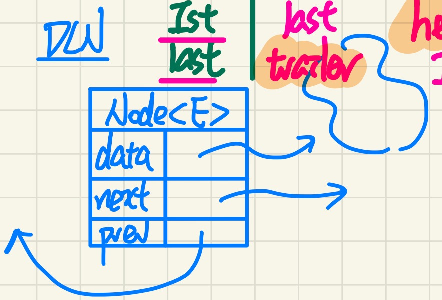
Doubly-Linked Lists (DLL): Visual Introduction

	next	prev
header	1st	null
trailer	null	last
1st	last	header
last	trailer	1st

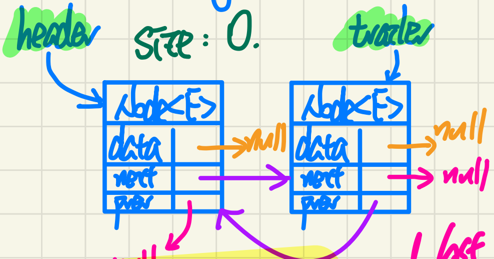
- A chain of bi-directionally connected nodes
- Each node contains:
  - + reference to a data object
  - + reference to the next node
  - + reference to the ✓ previous node   1. prev. ref
- A DLL is also a SLL: not vice versa   2. header vs. trailer
  - + many methods implemented the same way
  - + some method implemented more efficiently → removeLast → 2nd last node: trailer.prev.prev O(1)
- ✓ Accessing a node in a list:
  - + Relative positioning: O(n)
  - + Absolute indexing: O(1)
- The chain may grow or shrink dynamically.
- Dedicated Header vs. Trailer Nodes

(no head refeference and no tail reference)

∵ next ref. is available



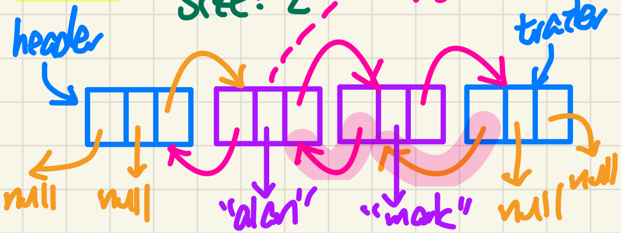
Case 1 : Empty DLL



Case 2: Non-Empty List

Size: 2

2nd last node



Lecture 2

Part I

Doubly-Linked Lists -
Java Implementation: Generic Lists
Initializing a List

Generic DLL in Java: DoublyLinkedList vs. Node

String

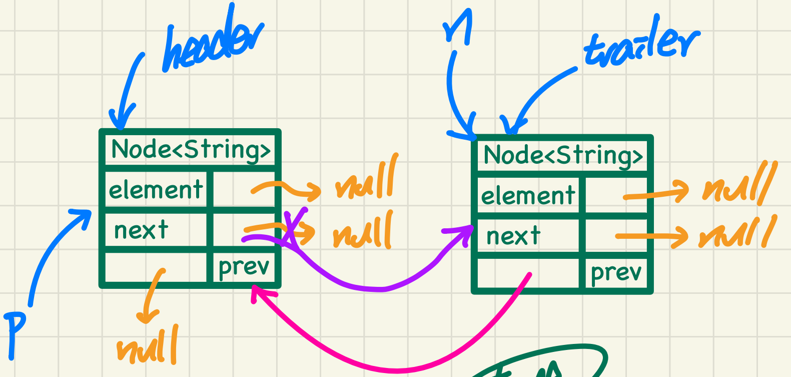
```
public class DoublyLinkedList<E> {
  private int size = 0;
  public void addFirst(E e) { ... }
  public void removeLast() { ... }
  public void addAt(int i, E e) { ... }
  private Node<E> header;
  private Node<E> trailer;
  public DoublyLinkedList() {
    header = new Node<>(null, null, null);
    trailer = new Node<>(null, header, null);
    header.setNext(trailer);
  }
}
```

```
public class Node<E> {
  private E element;
  private Node<E> next;
  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; }
  private Node<E> prev;
  public Node<E> getPrev() { return prev; }
  public void setPrev(Node<E> p) { prev = p; }
  public Node(E e, Node<E> p, Node<E> n) {
    element = e;
    prev = p;
    next = n;
  }
}
```

call by value

null

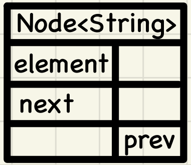
```
@Test
public void test_String_DLL_Empty_List() {
    DoublyLinkedList<String> list = new DoublyLinkedList<>();
    assertTrue(list.getSize() == 0);
    assertTrue(list.getFirst() == null);
    assertTrue(list.getLast() == null);
}
```



true

header.getNext() == trailer

trailer.getPrev() == header

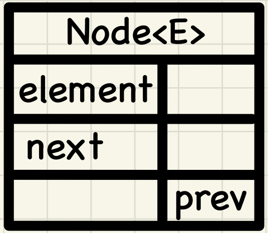


Lecture 2

Part J

Doubly-Linked Lists -
Java Implementation: Generic Lists
Operations on a List

Generic DLL in Java: Inserting between Nodes



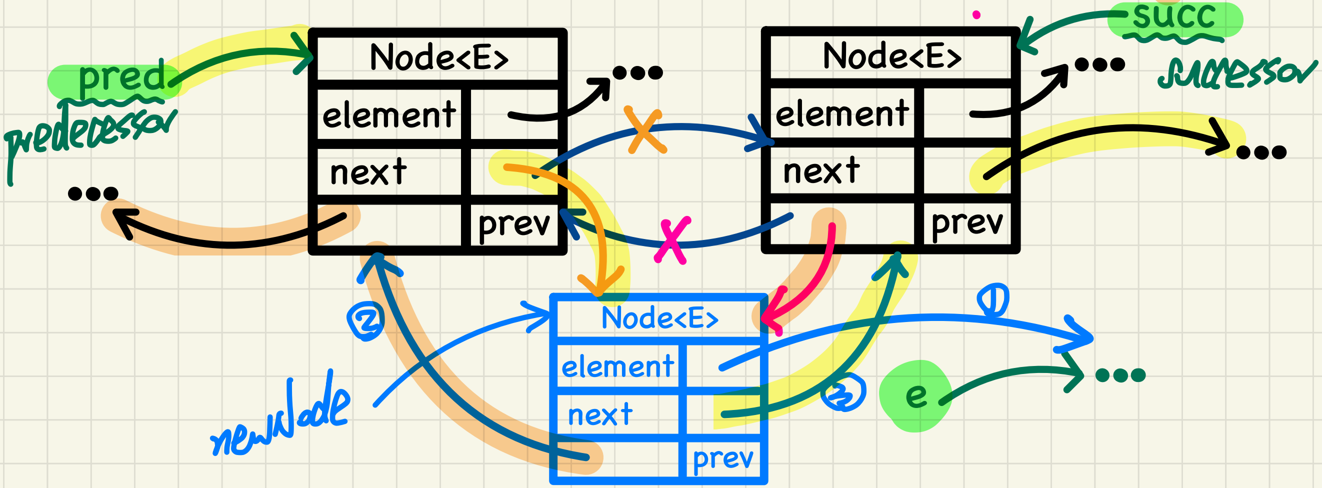
```
void addBetween(E e, Node<E> pred, Node<E> succ) {
  ✓Node<E> newNode = new Node<>(e, pred, succ);
  ✓pred.setNext(newNode);
  ✓succ.setPrev(newNode);
  size ++;
}
```

① ② ③

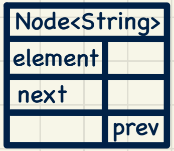
prev next

RT: O(1)

Assumption: pred and succ are directly connected.



Generic DLL in Java: Inserting to the Front/End

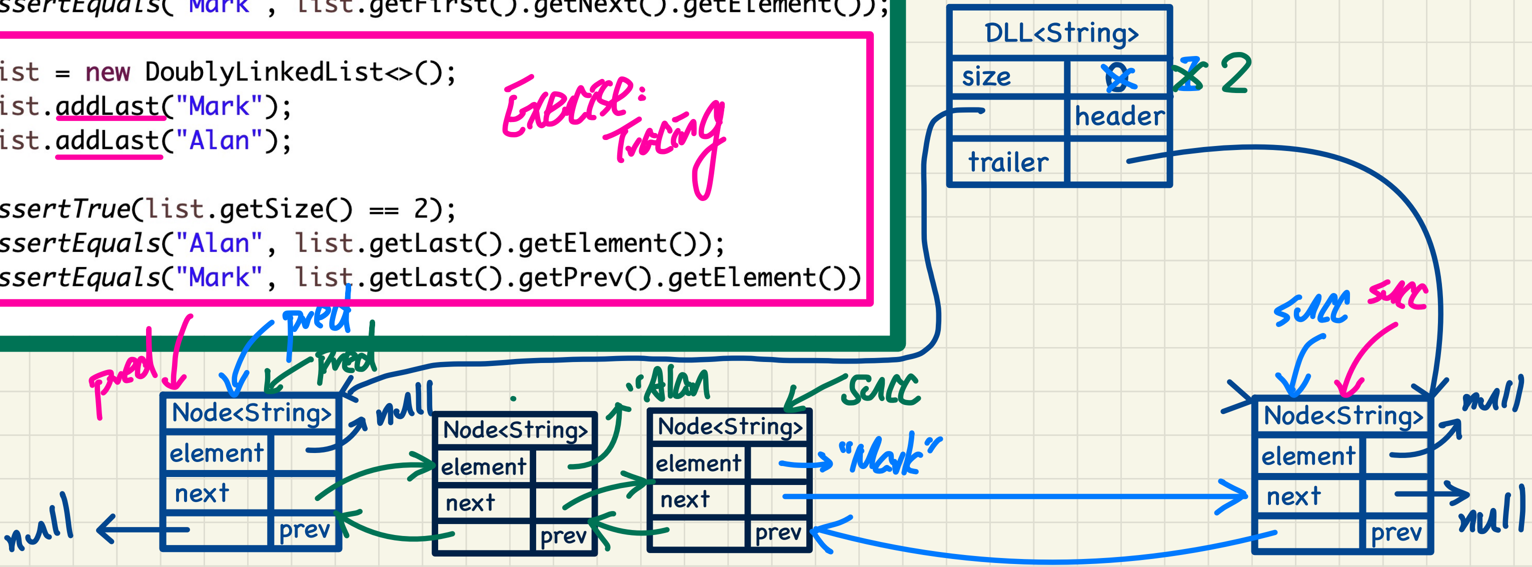


```
@Test
public void test_String_DLL_Insert_Front_End() {
    DoublyLinkedList<String> list = new DoublyLinkedList<>();
 ✓  list.addFirst("Mark");
    list.addFirst("Alan");

    assertTrue(list.getSize() == 2);
    assertEquals("Alan", list.getFirst().getElement());
    assertEquals("Mark", list.getFirst().getNext().getElement());

    list = new DoublyLinkedList<>();
    list.addLast("Mark");
    list.addLast("Alan");

    assertTrue(list.getSize() == 2);
    assertEquals("Alan", list.getLast().getElement());
    assertEquals("Mark", list.getLast().getPrev().getElement())
}
```

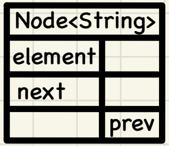
Exercise: Tracing

```
void addFirst(E e) {      pred          succ
   addBetween(e, header, header.getNext())
}
```

```
void addLast(E e) {           pred                succ
   addBetween(e, trailer.getPrev(), trailer)
}
```



Generic DLL in Java: Inserting to the Middle



```java
@Test
public void test_String_DLL_addAt() {
    DoublyLinkedList<String> list = new DoublyLinkedList<>();
    list.addAt(0, "Alan");
    list.addAt(1, "Tom");
    list.addAt(1, "Mark");

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

Exercise: Tracing.

0

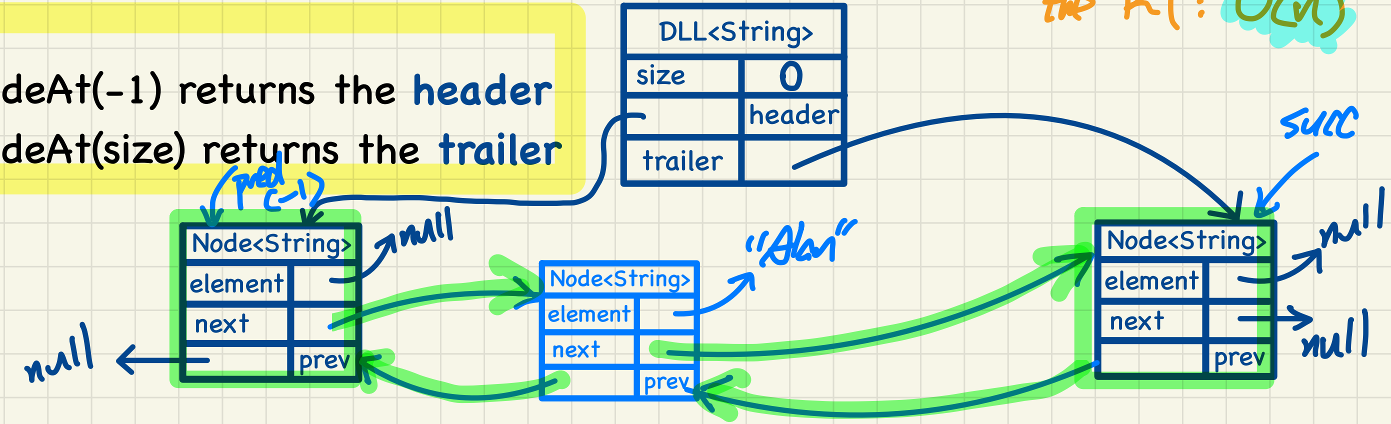
```java
addAt(int i, E e) {
  if (i < 0 || i > size) {
    throw new IllegalArgumentException
  else {
    Node<E> pred = getNodeAt(i - 1);
    Node<E> succ = pred.getNext();
    addBetween(e, pred, succ);
  }
}
```

0-1

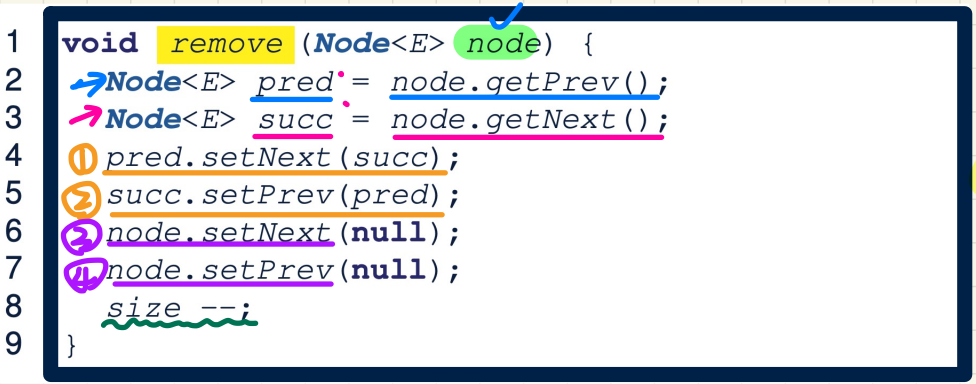
still dominates the RT: O(n)

Notes.

+ getNodeAt(-1) returns the **header**
+ getNodeAt(size) returns the **trailer**



Generic DLL in Java: Removing a Node

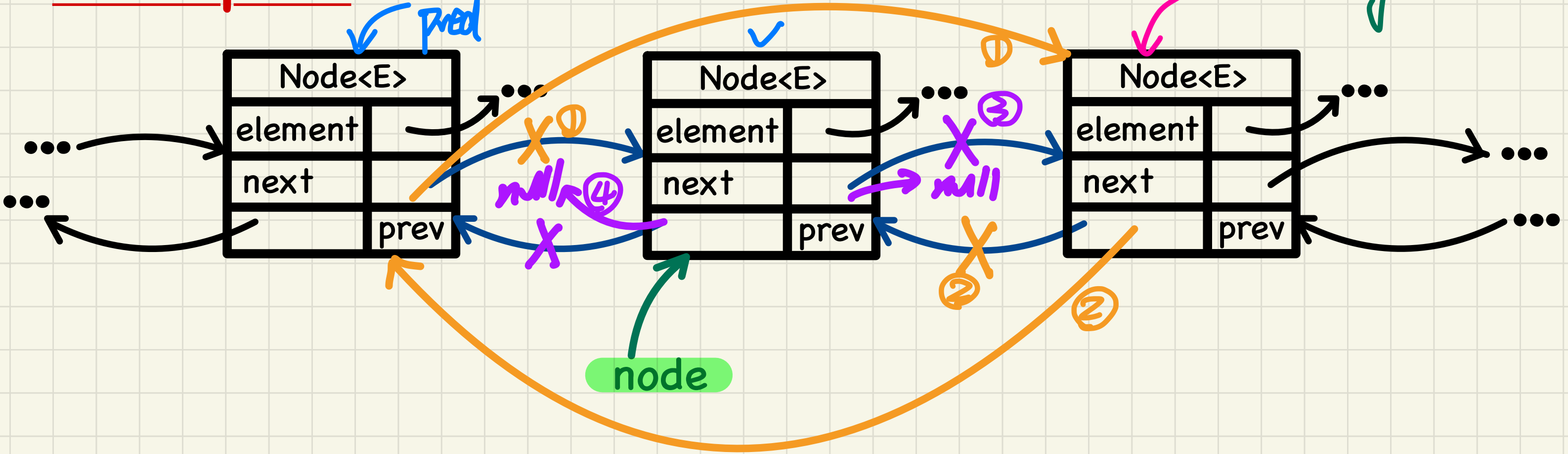


```
void remove (Node<E> node) {
  Node<E> pred = node.getPrev();
  Node<E> succ = node.getNext();
① pred.setNext(succ);
② succ.setPrev(pred);
③ node.setNext(null);
④ node.setPrev(null);
  size --;
}
```

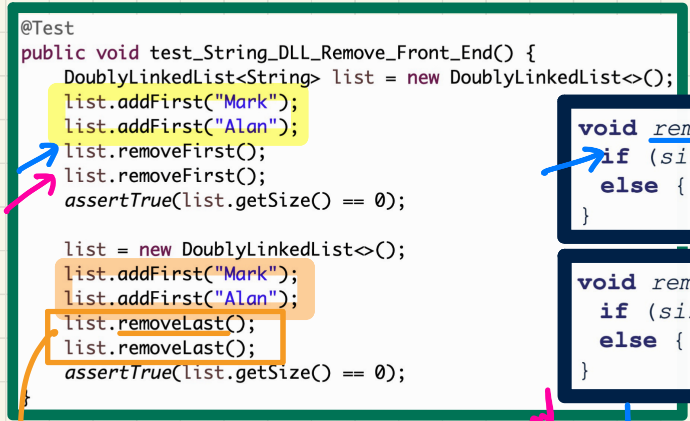
RT: O(1)

efficient solely because the ref. of the node to remove is given.

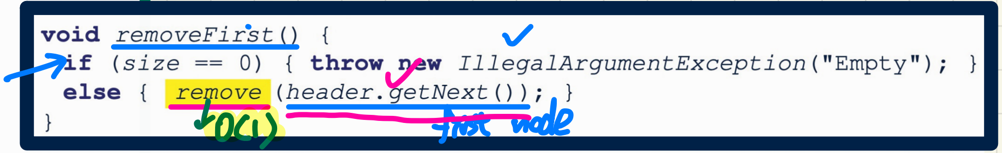
Assumption: node exists in some DLL.



Generic DLL in Java: Removing from the Front/End



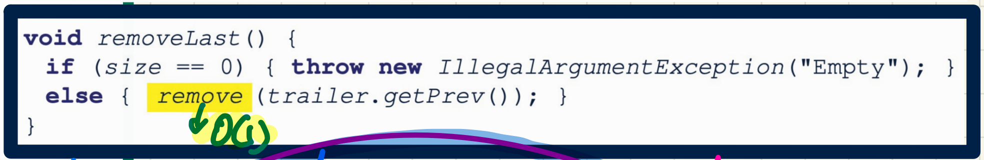
```java
@Test
public void test_String_DLL_Remove_Front_End() {
    DoublyLinkedList<String> list = new DoublyLinkedList<>();
    list.addFirst("Mark");
    list.addFirst("Alan");
    list.removeFirst();
    list.removeFirst();
    assertTrue(list.getSize() == 0);

    list = new DoublyLinkedList<>();
    list.addFirst("Mark");
    list.addFirst("Alan");
    list.removeLast();
    list.removeLast();
    assertTrue(list.getSize() == 0);
}
```



```java
void removeFirst() {
  if (size == 0) { throw new IllegalArgumentException("Empty"); }
  else { remove(header.getNext()); }
}
```

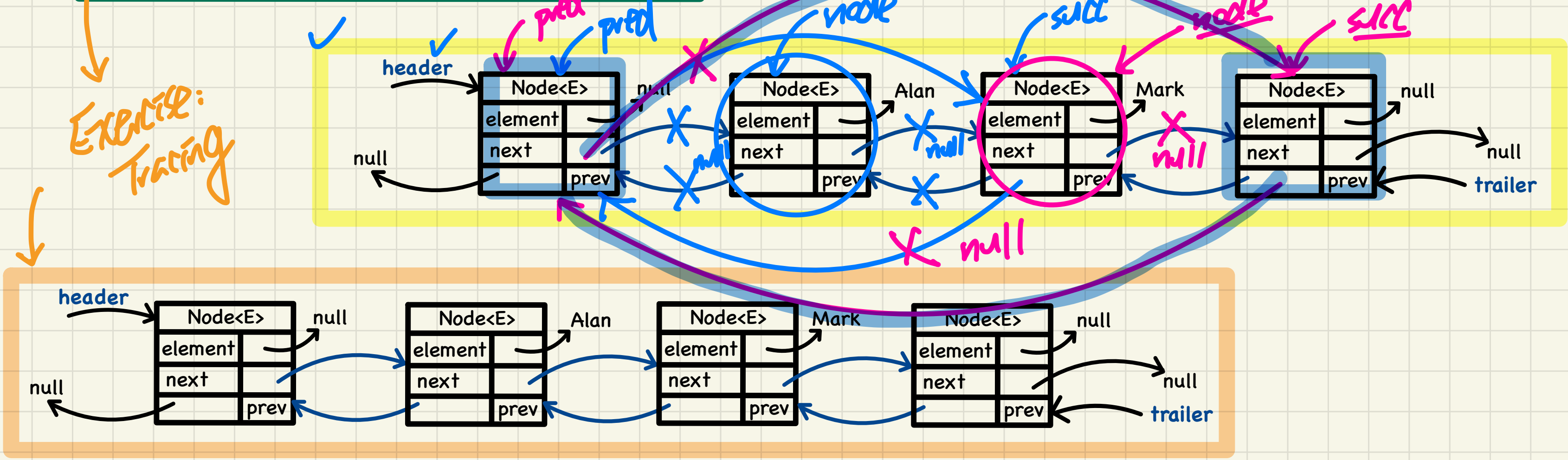
O(1)

first node

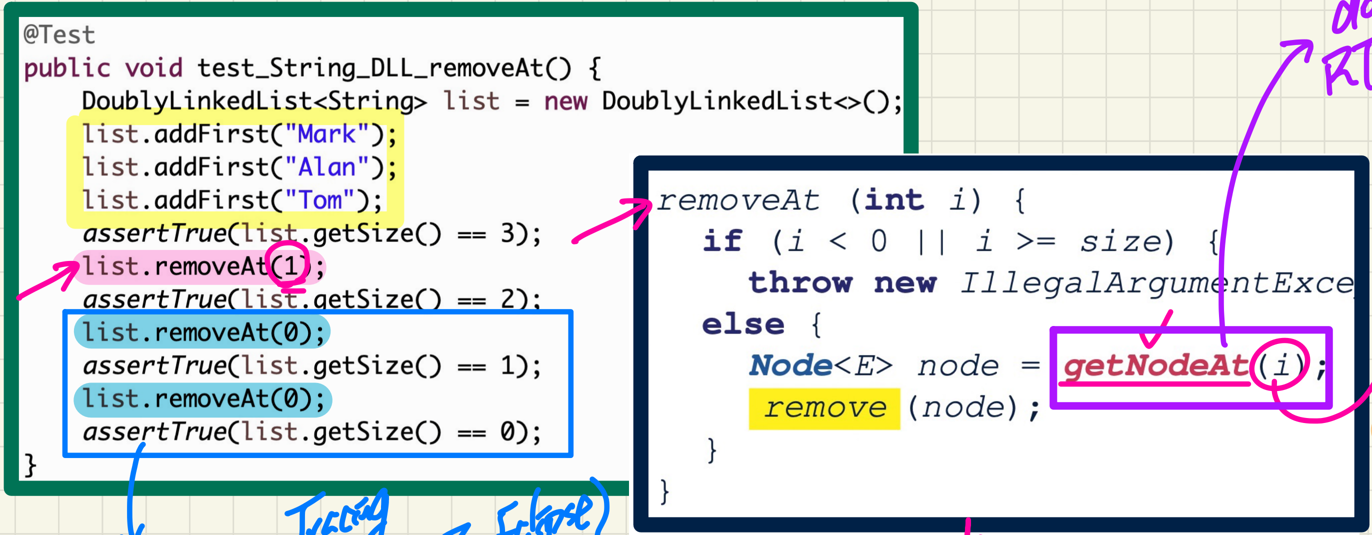


```java
void removeLast() {
  if (size == 0) { throw new IllegalArgumentException("Empty"); }
  else { remove(trailer.getPrev()); }
}
```

O(1)



Generic DLL in Java: Removing from the Middle



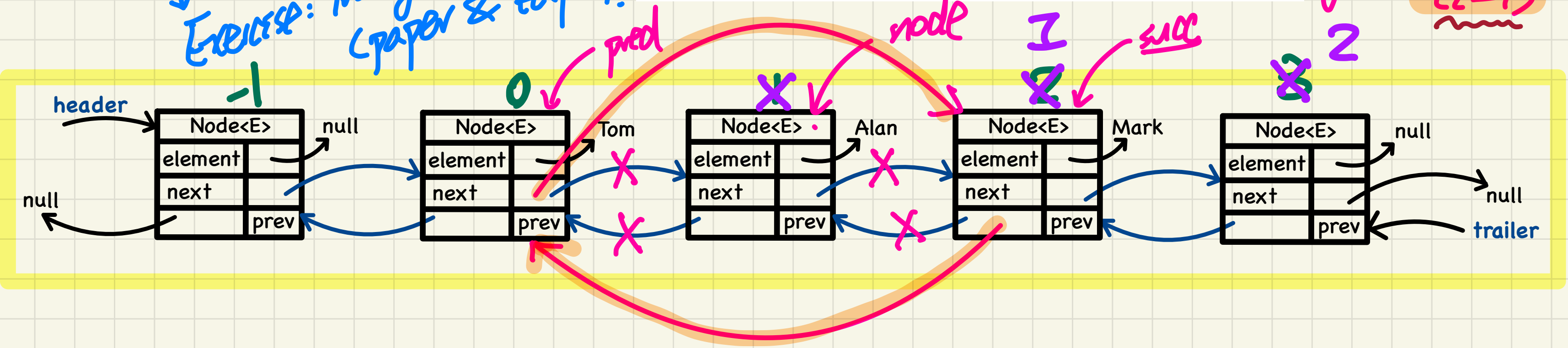
```
@Test
public void test_String_DLL_removeAt() {
    DoublyLinkedList<String> list = new DoublyLinkedList<>();
    list.addFirst("Mark");
    list.addFirst("Alan");
    list.addFirst("Tom");
    assertTrue(list.getSize() == 3);
    list.removeAt(1);
    assertTrue(list.getSize() == 2);
    list.removeAt(0);
    assertTrue(list.getSize() == 1);
    list.removeAt(0);
    assertTrue(list.getSize() == 0);
}
```

```
removeAt (int i) {
  if (i < 0 || i >= size) {
    throw new IllegalArgumentExce
  else {
    Node<E> node = getNodeAt(i);
    remove(node);
  }
}
```

dominates
RT: O(n)

Contrast
SLL removeAt:
getNodeAt
(i-1)

Exercise: Tracing (paper & Eclipse).



Lecture 2

Part K

Doubly-Linked Lists -
Comparing Arrays, SLL, and DLL

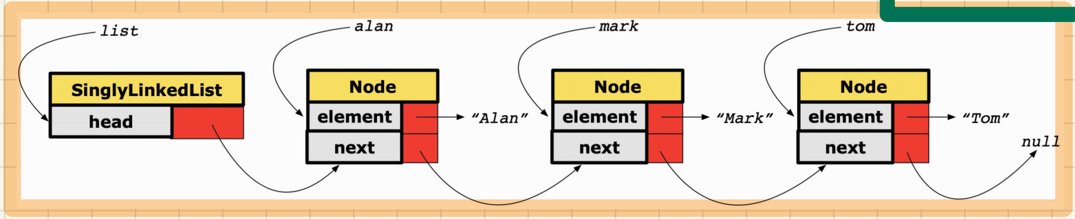
Running Time: Arrays vs. SLL vs. DLL

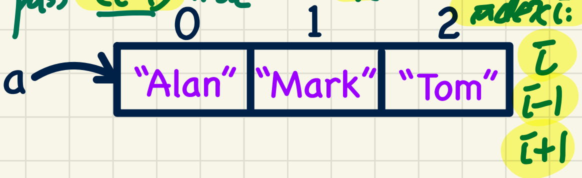
see discussion at the end of SLL.

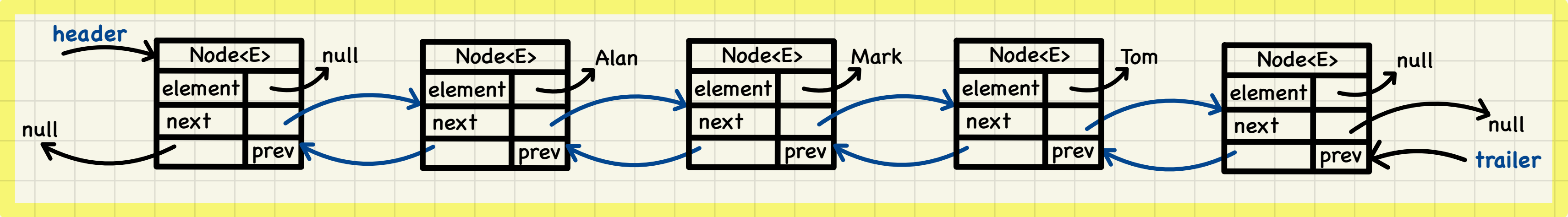
OPERATION \ DATA STRUCTURE		ARRAY	SINGLY-LINKED LIST	DOUBLY-LINKED LIST
size		O(1)	O(1)	O(1)
first/last element				
element at index i		O(1)	O(n)	O(n)
remove last element		O(n)		O(1)
add/remove first element, add last element			O(1) remove at index	
add/remove i^{th} element	given reference to $(i-1)^{th}$ element			
	not given		i : O(n)	

pass (i-1) node.

remove at index i:







Lecture 3

Part A

Modularity, Abstract Data Types (ADTs) - Definition & Terminology

Supplier vs. Client in OOP

supplier class (callee)

client class (caller)

```
class Microwave {
  private boolean on;
  private boolean locked;
  void power() {on = true;}
  void lock() {locked = true;}
  void heat(Object stuff) {
    /* Assume: on && locked */
    /* stuff not explosive. */
  } }
```

```
class MicrowaveUser {
  public static void main(...) {
    Microwave m = new Microwave();
    Object obj = ??? ;
    m.power(); m.lock();]
    m.heat(obj);
} }
```

client fulfilling obligation

contractual relation in effect

given client fulfilling their obligations, supplier must fulfill their obligation.

supplier method/service used in the context of the client class.

pre-state (pre-execution of supplier method)

m.heat(obj);

post-state (post-execution of supplier method)

supplier's obligation must be fulfilled.

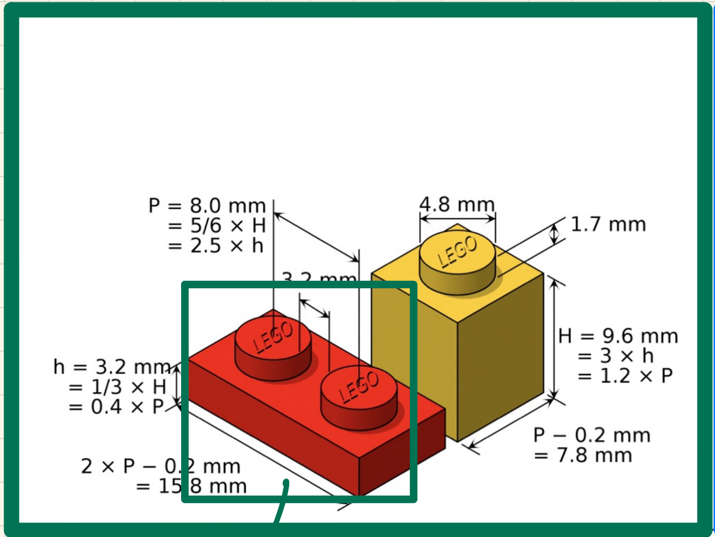
postcondition (conditions for supplier to satisfy).

client's obligation must be fulfilled

✓ precondition (conditions for client to satisfy in order to use the supplier's method)

in Java: exceptions.

Modularity: Childhood Activities

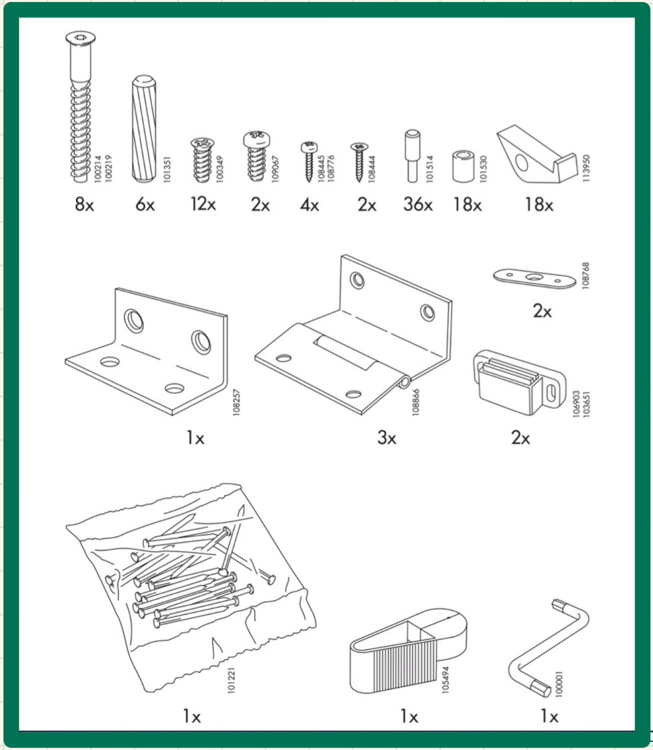


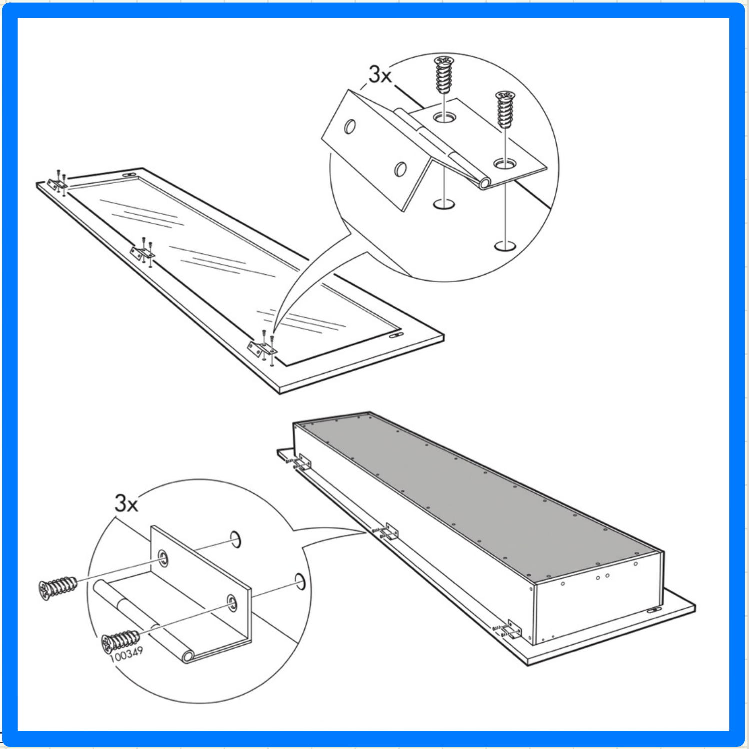
Interface specification (of a module)

Architecture (assembly)

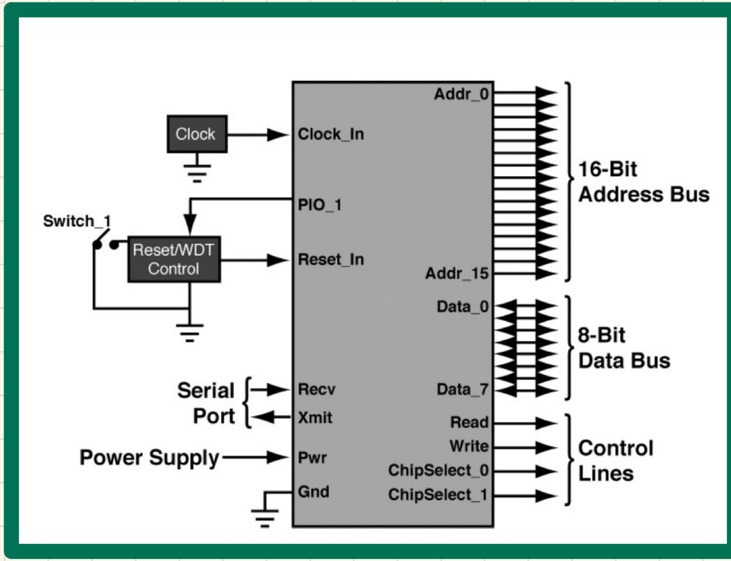
↳ of building blocks

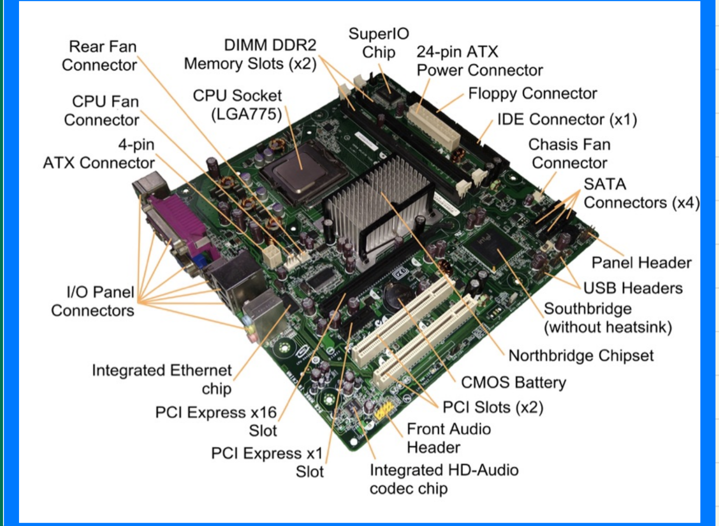
Modularity: Daily Constructions



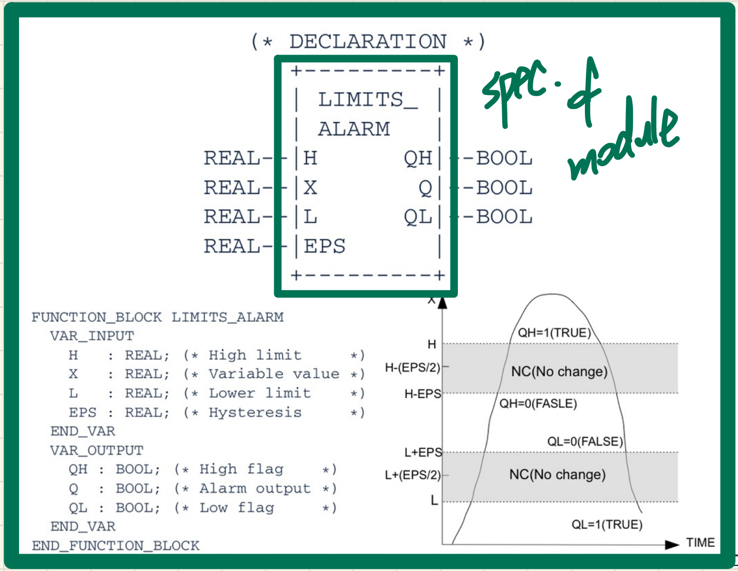


Modularity: Computer Architectures





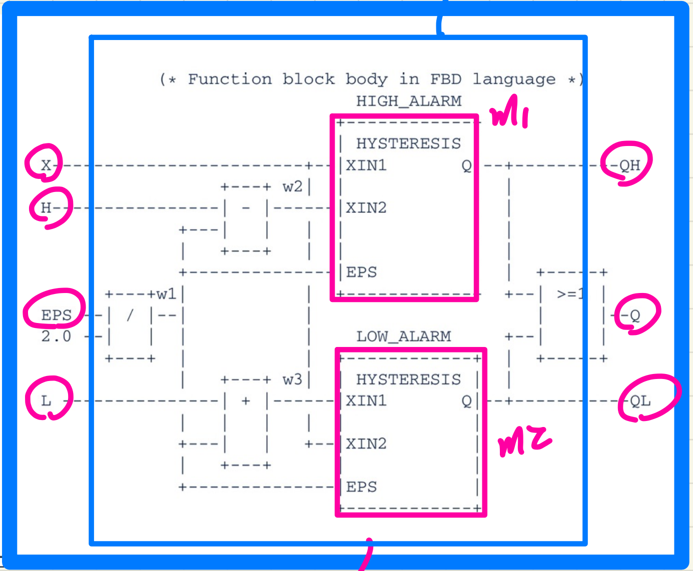
Modularity: System Developments



```
                 (* DECLARATION *)
                   +----------+
                   | LIMITS_  |
                   | ALARM    |
           REAL--|H        QH|--BOOL
           REAL--|X         Q|--BOOL
           REAL--|L        QL|--BOOL
           REAL--|EPS        |
                   +----------+
```

```
FUNCTION_BLOCK LIMITS_ALARM
  VAR_INPUT
    H   : REAL; (* High limit     *)
    X   : REAL; (* Variable value *)
    L   : REAL; (* Lower limit    *)
    EPS : REAL; (* Hysteresis     *)
  END_VAR
  VAR_OUTPUT
    QH : BOOL; (* High flag    *)
    Q  : BOOL; (* Alarm output *)
    QL : BOOL; (* Low flag     *)
  END_VAR
END_FUNCTION_BLOCK
```

spec. of module



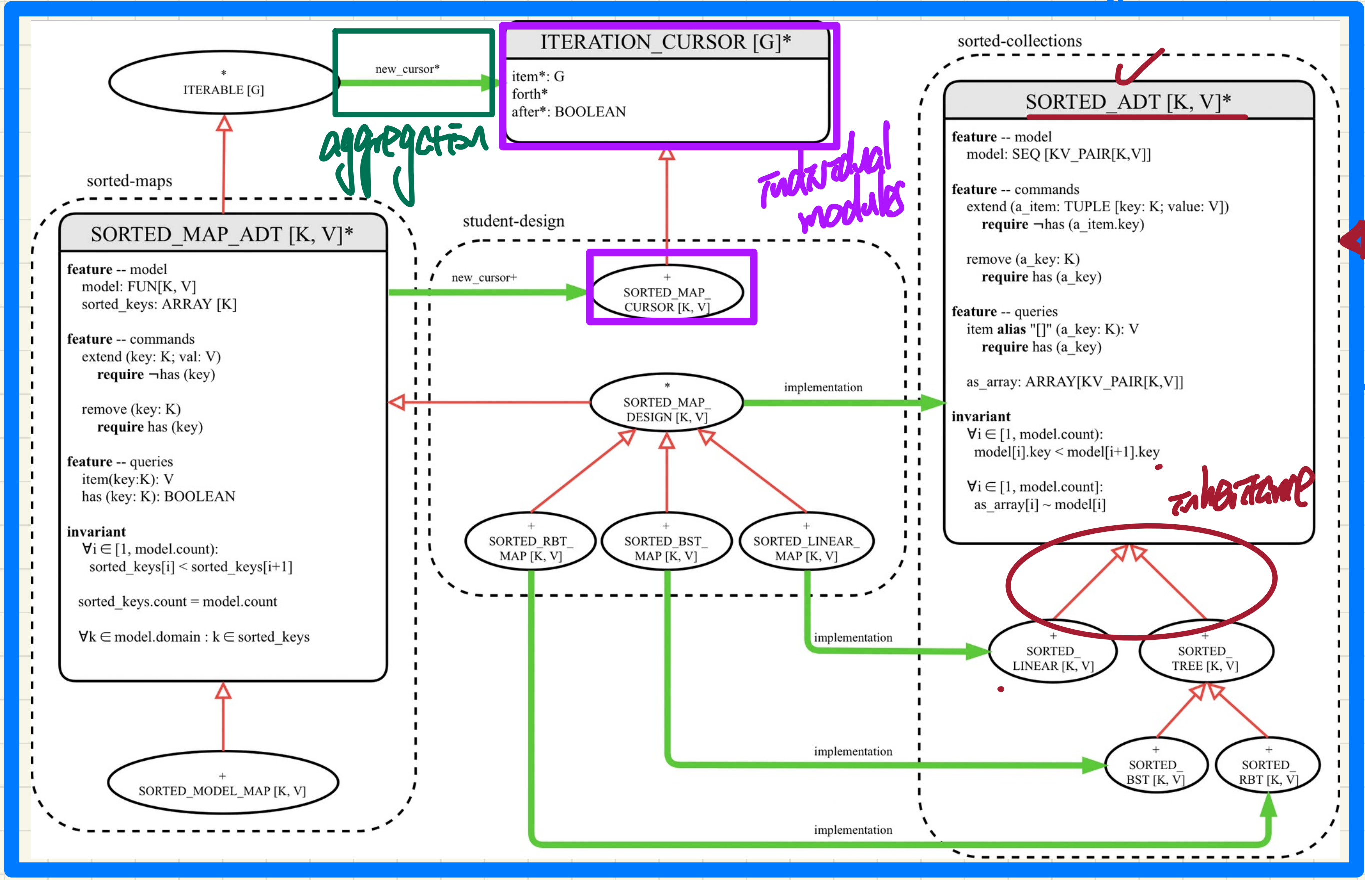
```
              (* Function block body in FBD language *)
                               HIGH_ALARM
                         +----------------+
                         | HYSTERESIS     |
X------------------+-----|XIN1          Q |--+------------QH
             +----+ w2|  |                |  |
H------------|  - |------|XIN2            |  |
          +--|    |   |  |                |  |
          |  +----+   |  |                |  |
          +--------------|EPS             |  |   +-----+
   +----+w1|          |  +----------------+  +--| >=1 |
EPS--| / |--|         |                         |     |--Q
2.0 --|   | |         |      LOW_ALARM       +--|     |
   +----+  |          |  +----------------+  |  +-----+
           |  +----+ w3| | HYSTERESIS     |  |
L------------|  + |------|XIN1          Q |--+------------QL
           |  |    |   | |                |
           +--|    |   +-|XIN2            |
           |  +----+     |                |
           +-------------|EPS             |
                         +----------------+
```

a bigger module

assembly as a Composition of well-specified modules

Modularity: Software Design

In OOP, assemble classes via:
1. aggregations 2. compositions
3. inheritance



```
ITERATION_CURSOR [G]*
item*: G
forth*
after*: BOOLEAN
```

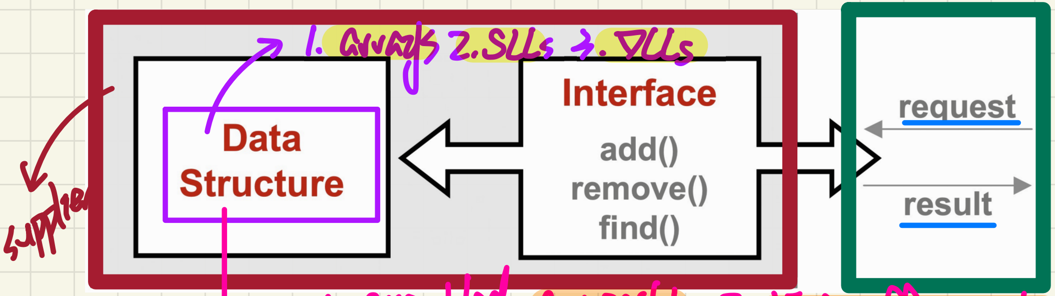
```
SORTED_MAP_ADT [K, V]*
feature -- model
   model: FUN[K, V]
   sorted_keys: ARRAY [K]

feature -- commands
   extend (key: K; val: V)
      require ¬has (key)

   remove (key: K)
      require has (key)

feature -- queries
   item(key:K): V
   has (key: K): BOOLEAN

invariant
   ∀i ∈ [1, model.count):
     sorted_keys[i] < sorted_keys[i+1]

   sorted_keys.count = model.count

   ∀k ∈ model.domain : k ∈ sorted_keys
```

```
SORTED_ADT [K, V]*
feature -- model
   model: SEQ [KV_PAIR[K,V]]

feature -- commands
   extend (a_item: TUPLE [key: K; value: V])
      require ¬has (a_item.key)

   remove (a_key: K)
      require has (a_key)

feature -- queries
   item alias "[]" (a_key: K): V
      require has (a_key)

   as_array: ARRAY[KV_PAIR[K,V]]

invariant
   ∀i ∈ [1, model.count):
     model[i].key < model[i+1].key

   ∀i ∈ [1, model.count]:
     as_array[i] ~ model[i]
```

New Module

architectural diagram.

Abstract Data Types (ADTs)



client (relies on the public interface of ADT)

1. input types
2. output types
3. what to be expected on IO relation.

1. assembled correctly 2. time efficiency

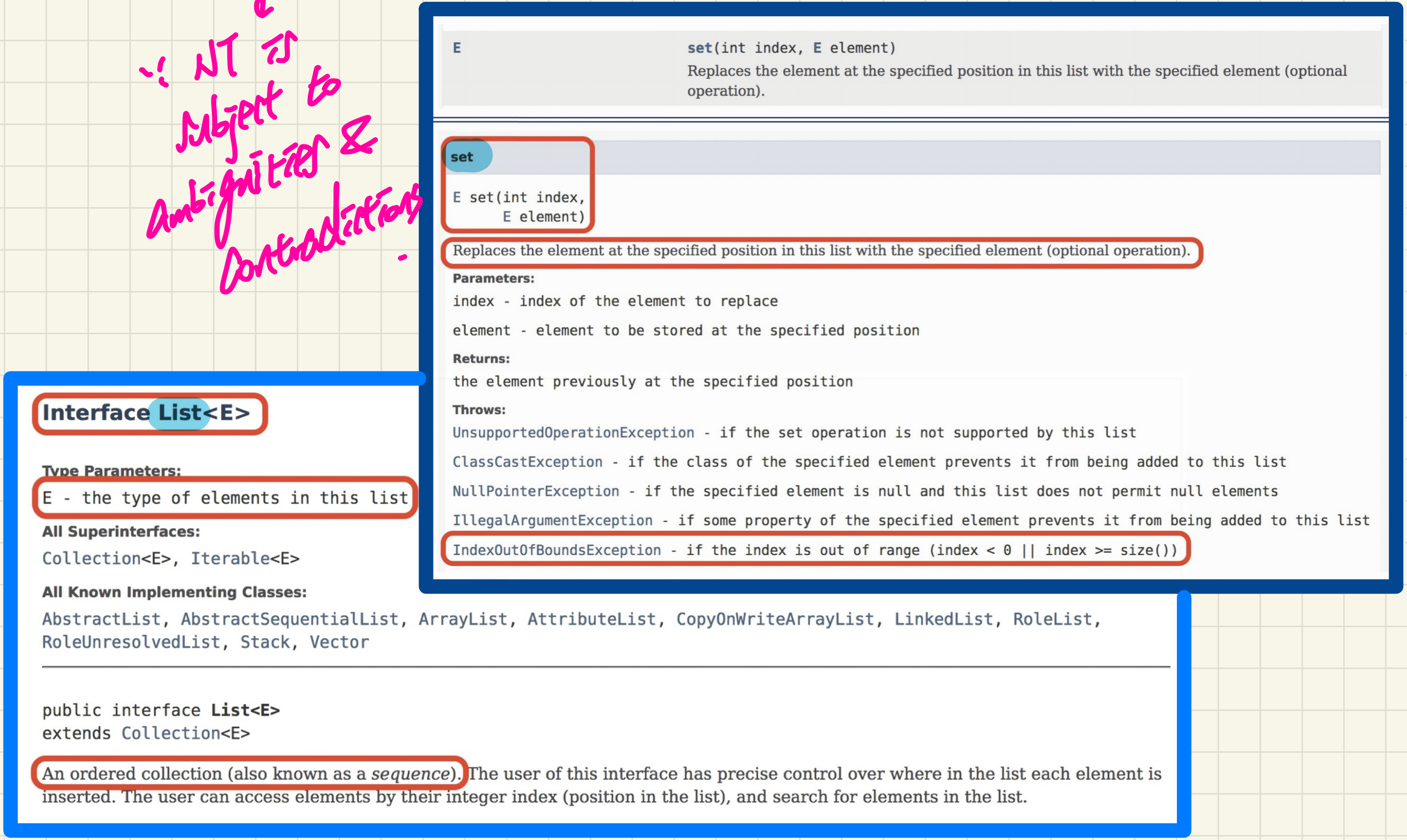
```
class Microwave {
  private boolean on;
  private boolean locked;
  void power() {on = true;}
  void lock() {locked = true;}
  void heat(Object stuff) {
    /* Assume: on && locked */
    /* stuff not explosive. */
} }
```

```
class MicrowaveUser {
  public static void main(...) {
    Microwave m = new Microwave();
    Object obj = ???;
    m.power(); m.lock();]
    m.heat(obj);
} }
```

	benefits	*obligations*
CLIENT	obtain a service	follow instructions
SUPPLIER	assume instructions followed	provide a service

Java API ≈ Abstract Data Types

∵ NL is subject to Ambiguities & Contradictions.



Interface List<E>

Type Parameters:

E - the type of elements in this list

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Implementing Classes:

AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector

```
public interface List<E>
extends Collection<E>
```

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element (optional operation).

set

```
E set(int index,
      E element)
```

Replaces the element at the specified position in this list with the specified element (optional operation).

Parameters:

index - index of the element to replace

element - element to be stored at the specified position

Returns:

the element previously at the specified position

Throws:

UnsupportedOperationException - if the set operation is not supported by this list

ClassCastException - if the class of the specified element prevents it from being added to this list

NullPointerException - if the specified element is null and this list does not permit null elements

IllegalArgumentException - if some property of the specified element prevents it from being added to this list

IndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())