Basic Data Structures: Arrays vs. Linked-Lists

EECS2011 N & Z:
Fundamentals of Data Structures
Winter 2022

CHEN-WEI WANG

Background Study: Generics in Java

- It is assumed that, in EECS2030, you learned about the basics of Java ***generics***:
  - General collection (e.g., `Object[]`) vs. Generic collection (e.g., `E[]`)
  - How using generics minimizes ***casts*** and ***`instanceof` checks***
  - How to implement and use generic classes
- If needed, review the above assumed basics from the relevant parts of EECS2030 (`https://www.eecs.yorku.ca/~jackie/teaching/lectures/index.html#EECS2030_F21`):
  - Parts A1 – A3, Lecture 7, Week 10
  - Parts B – C, Lecture 7, Week 11

  Tips.
  - Skim the ***slides***: watch lecture videos if needing explanations.
  - Ask questions related to the assumed basics of ***generics***!
- Assuming that know the basics of Java ***generics***, we will implement and use ***generic SLL*** and ***DLL***.

Learning Outcomes of this Lecture

This module is designed to help you learn about:

- ***basic data structures***: ***Arrays*** vs. ***Linked Lists***
- Two ***Sorting*** Algorithms: Selection Sort vs. Insertion Sort
- ***Linked Lists***: **Singly**-Linked vs. **Doubly**-Linked
- ***Running Time***: Array vs. Linked-List Operations
- Java ***Implementations***: `String` Lists vs. ***Generic*** Lists

Basic Data Structure: Arrays

- An array is a sequence of <u>indexed</u> elements.
- ***Size*** of an array is **<u>fixed</u>** at the time of its construction.
  - e.g., `int[] numbers = new int[10];`
  - **<u>Heads-Up</u>**. Two ***resizing*** strategies: ***increments*** vs. ***doubling***.
- Supported operations on an array:
  - ***Accessing***: e.g., `int max = a[0];`
    Time Complexity: *O(1)* [constant-time op.]
  - ***Updating***: e.g., `a[i] = a[i + 1];`
    Time Complexity: *O(1)* [constant-time op.]
  - ***Inserting/Removing***:

```
String[] insertAt(String[] a, int n, String e, int i)
  String[] result = new String[n + 1];
  for(int j = 0; j <= i - 1; j ++){ result[j] = a[j]; }
  result[i] = e;
  for(int j = i + 1; j <= n; j ++){ result[j] = a[j-1]; }
  return result;
```

Time Complexity: *O(n)* [linear-time op.]

Array Case Study: Comparing Two Sorting Strategies

- ***The Sorting Problem***:

  Input: An array a of n numbers $\langle a_1,\ a_2,\ \ldots,\ a_n \rangle$ (e.g., $\langle 3, 4, 1, 3, 2 \rangle$)

  Output: A permutation/reordering $\langle a'_1,\ a'_2,\ \ldots,\ a'_n \rangle$ of the input sequence s.t. elements are arranged in a ***non-descending*** order (e.g., $\langle 1, 2, 3, 3, 4 \rangle$): $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

  Remark. Variants of the ***sorting problem*** may require different ***orderings***:
  - non-descending
  - ascending/increasing
  - non-ascending
  - descending/decreasing
- Two *alternative* ***implementation strategies*** for solving this problem
- At the end, choose one based on their ***time complexities***.

Sorting: Strategy 1 – Selection Sort

- Maintain a (initially empty) ***sorted portion*** of array *a*.
- From left to right in array *a*, select and insert the ***minimum*** element to the ***end*** of this sorted portion, so it remains sorted.

```
1 void selectionSort(int[] a, int n)
2   for (int i = 0; i <= (n - 2); i ++)
3     int minIndex = i;
4     for (int j = i; j <= (n - 1); j ++)
5       if (a[j] < a[minIndex]) { minIndex = j; }
6     int temp = a[i];
7     a[i] = a[minIndex];
8     a[minIndex] = temp;
```

- How many times does the body of **for-loop** (L4) run? [(n - 1)]
- Running time? [$O(n^2)$]

$$\underbrace{n}_{\texttt{find \{a[0], ..., a[n-1]\}}} + \underbrace{(n-1)}_{\texttt{find \{a[1], ..., a[n-1]\}}} + \cdots + \underbrace{2}_{\texttt{find \{a[n - 2], a[a[n - 1]]\}}}$$

- So ***selection sort*** is a ***quadratic-time algorithm***.

Sorting: Strategy 2 – Insertion Sort

- Maintain a (initially empty) ***sorted portion*** of array *a*.
- From left to right in array *a*, insert ***one element*** at a time into the ***"correct" spot*** in this sorted portion, so it remains sorted.

```
1 void insertionSort(int[] a, int n)
2   for (int i = 1; i < n; i ++)
3     int current = a[i];
4     int j = i;
5     while (j > 0 && a[j - 1] > current)
6       a[j] = a[j - 1];
7       j --;
8     a[j] = current;
```

- **while-loop** (L5) exits when? [`j <= 0` or `a[j - 1] <= current`]
- Running time? [$O(n^2)$]

$$O(\underbrace{1}_{\texttt{insert into \{a[0]\}}} + \underbrace{2}_{\texttt{insert into \{a[0], a[1]\}}} + \cdots + \underbrace{(n-1)}_{\texttt{insert into \{a[0], ..., a[n-2]\}}})$$

- So ***insertion sort*** is a ***quadratic-time algorithm***.

Sorting: Alternative Implementations?

- In the Java implementations of ***selection sort*** and ***insertion sort***, we maintain the ***"sorted portion"*** from the ***left*** end.
  - For ***selection sort***, we select the ***minimum*** element from the ***"unsorted portion"*** and insert it to the ***end*** of the ***"sorted portion"***.
  - For ***insertion sort***, we choose the ***left-most*** element from the ***"unsorted portion"*** and insert it at the ***"correct spot"*** in the ***"sorted portion"***.
- **Exercise:** Modify the Java implementations, so that the ***"sorted portion"*** is:
  - arranged in a ***non-ascending*** order (e.g., $\langle 5, 4, 3, 2, 1 \rangle$); and
  - maintained and grown from the ***right*** end instead.

Comparing Insertion & Selection Sorts

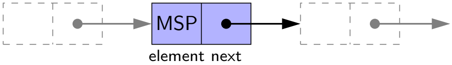
- *Asymptotically*, running times of ***selection sort*** and ***insertion sort*** are both $O(n^2)$.
- We will later see that there exist better algorithms that can perform better than quadratic: $O(n \cdot logn)$.

Basic Data Structure: Singly-Linked Lists

- We know that ***arrays*** perform:
  - ***well*** in indexing
  - ***badly*** in inserting and deleting
- We now introduce an alternative data structure to arrays.
- A ***linked list*** is a series of **connected** ***nodes***, forming a ***linear sequence***.

  Remark. At ***runtime***, node ***connections*** are through ***reference aliasing***.
- Each ***node*** in a *singly-linked list (SLL)* stores:
  - ***reference*** to a ***data object***; and
  - ***reference*** to the ***next node*** in the list.

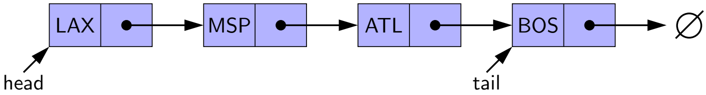
  Contrast. *relative* positioning of LL vs. **absolute** indexing of arrays



- The ***last node*** in a singly-linked list is different from others. How so? Its reference to the ***next node*** is simply `null`.

Singly-Linked List: How to Keep Track?

- Due to its ***"chained" structure***, a SLL, when first being created, does **not** need to be specified with a **fixed length**.
- We can use a SLL to ***dynamically*** store and manipulate as many elements as we desire **without** the need to ***resize*** by:
  - e.g., ***creating*** a new node and setting the relevant ***references***.
  - e.g., ***inserting*** some node to the ***beginning***/***middle***/***end*** of a SLL
  - e.g., ***deleting*** some node from the ***beginning***/***middle***/***end*** of a SLL
- **Contrary to arrays**, we do **not** keep track of all nodes in a SLL **directly** by indexing the ***nodes***.
- Instead, we only store a ***reference*** to the ***head*** (i.e., ***first node***), and find other parts of the list ***indirectly***.



- **Exercise**: Given the ***head*** reference of a SLL, describe how we may:
  - Count the number of nodes currently in the list. [Running Time?]
  - Find the reference to its ***tail*** (i.e., ***last node***) [Running Time?]

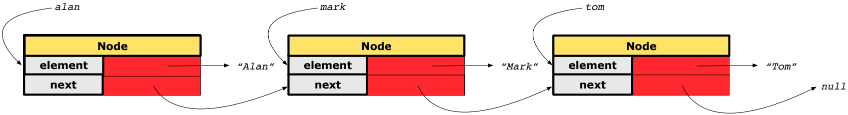
Singly-Linked List: Java Implementation

We first implement a ***SLL*** storing strings only.

```
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; }
}
```

```
public class SinglyLinkedList {
  private Node head;
  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() { ... }
}
```

Singly-Linked List: Constructing a Chain of Nodes



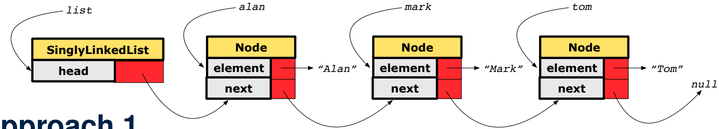
Approach 1

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

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);
```

Singly-Linked List: Setting a List's Head



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);
```

Singly-Linked List: Counting # of Nodes (1)

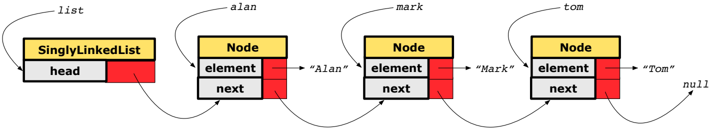
Problem: Return the number of nodes currently stored in a SLL.

- **Hint**. Only the ***last node*** has a ***null next*** reference.
- Assume we are in the context of class `SinglyLinkedList`.

```
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 }
```

- When does the while-loop (L4) exit? [`current == null`]
- RT of `getSize`: *O(n)* [linear-time op.]
- **Contrast**: RT of `a.length`: *O(1)* [constant-time op.]

Singly-Linked List: Counting # of Nodes (2)




```
int getSize() {
  int size = 0;
  Node current = head;
  while (current != null) { /* exit when current == null */
    current = current.getNext();
    size ++;
  }
  return size;
}
```

Let's now consider `list.getSize()`:

`current`	`current != null`	End of Iteration	`size`
alan	*true*	1	1
mark	*true*	2	2
tom	*true*	3	*3*
`null`	*false*	–	–

Singly-Linked List: Finding the Tail (1)

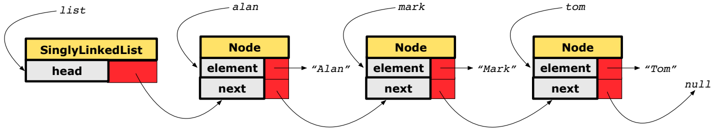
Problem: Retrieved the tail (i.e., last node) in a SLL.

- **Hint**. Only the ***last node*** has a ***null next*** reference.
- Assume we are in the context of class `SinglyLinkedList`.

```
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 }
```

- When does the while-loop (L4) exit? [`current == null`]
- RT of `getTail`: *O(n)* [linear-time op.]
- **Contrast**: RT of `a[a.length - 1]`: *O(1)* [constant-time op.]

Singly-Linked List: Finding the Tail (2)




```
Node getTail() {
  Node current = head;
  Node tail = null;
  while (current != null) { /* exit when current == null */
    tail = current;
    current = current.getNext();
  }
  return tail;
}
```

Let's now consider `list.getTail()`:

`current`	`current != null`	End of Iteration	`tail`
alan	*true*	1	alan
mark	*true*	2	mark
tom	*true*	3	*tom*
`null`	*false*	–	–

Singly-Linked List: Can We Do Better?

- In practice, we may frequently need to:
  - Access the ***tail*** of a list. [e.g., customers joining a service queue]
  - Inquire the ***size*** of a list. [e.g., the service queue full?]

  Both operations cost $O(n)$ to run (with only ***head*** available).
- We may improve the ***RT*** of these two operations.

  Principle. Trade ***space*** for ***time***.

  - Declare a new attribute ***tail*** pointing to the end of the list.
  - Declare a new attribute ***size*** denoting the number of stored nodes.
  - ***RT*** of these operations, accessing attribute values, are $O(1)$!
- Why not declare attributes to store references of ***all nodes*** between ***head*** and ***tail*** (e.g., `secondNode`, `thirdNode`)?
  - No – at the ***time of declarations***, we simply do **not** know how many nodes there will be at ***runtime***.

Singly-Linked List: Inserting to the Front (1)

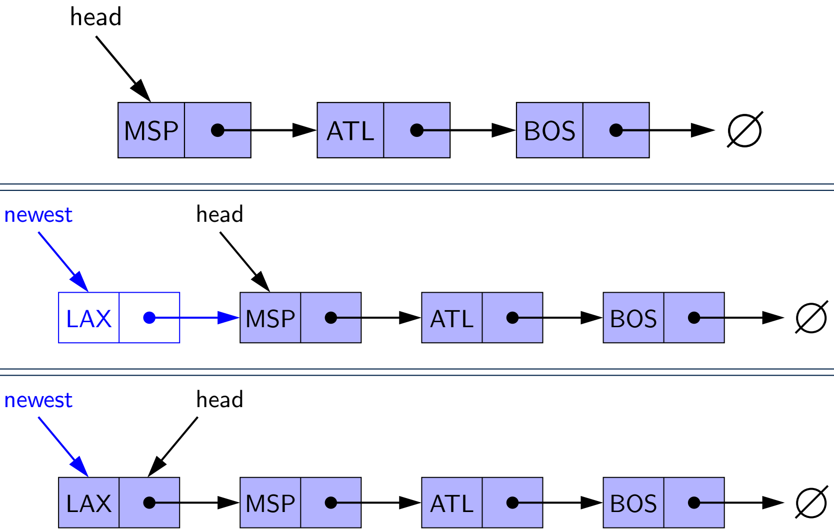
Problem: Insert a new string *e* to the front of the list.

- **Hint**. The list's new head should store *e* and point to the old head.
- Assume we are in the context of class `SinglyLinkedList`.

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

- Remember that RT of accessing *head* or *tail* is $O(1)$
- RT of `addFirst` is *O(1)* [constant-time op.]
- **Contrast**: Inserting into an array costs *O(n)* [linear-time op.]

Singly-Linked List: Inserting to the Front (2)



Exercise

See `ExampleStringLinkedLists.zip`.

Compare and contrast two alternative ways to constructing a SLL: `testSLL_01` vs. `testSLL_02`.

Exercise

- Complete the Java ***implementations***, ***tests***, and ***running time analysis*** for:
  - `void removeFirst()`
  - `void addLast(String e)`
- **Question:** The `removeLast()` method may not be completed in the same way as is `void addLast(String e)`. Why?

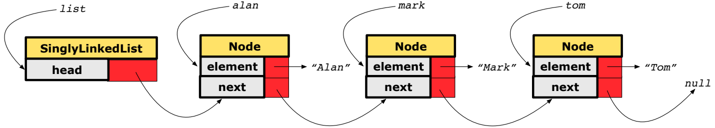
Singly-Linked List: Accessing the Middle (1)

Problem: Return the node at index *i* in the list.

- **Hint**. $0 \leq i <$ `list.getSize()`
- Assume we are in the context of class `SinglyLinkedList`.

```
Node getNodeAt (int i) {
  if (i < 0 || i >= size) {
    throw new IllegalArgumentException("Invalid Index");
  }
  else {
    int index = 0;
    Node current = head;
    while (index < i) { /* exit when index == i */
      index ++;
      /* current is set to node at index i
       * last iteration: index incremented from i - 1 to i
       */
      current = current.getNext();
    }
    return current;
  }
}
```

Singly-Linked List: Accessing the Middle (2)




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

Let's now consider `list.getNodeAt(2)`:

current	index	index < 2	Beginning of Iteration
alan	0	*true*	1
mark	1	*true*	2
tom	2	*false*	–

Singly-Linked List: Accessing the Middle (3)

- What is the ***worst case*** of the index `i` for `getNodeAt(i)`?
  - Worst case: `list.getNodeAt(list.size - 1)`
  - RT of `getNodeAt` is *O(n)* [linear-time op.]
- **Contrast**: Accessing an array element costs *O(1)* [constant-time op.]

Singly-Linked List: Inserting to the Middle (1)

Problem: Insert a new element at index *i* in the list.

- **Hint 1**. $0 \leq i \leq$ `list.getSize()`
- **Hint 2**. Use `getNodeAt(?)` as a helper method.

```
void addAt (int i, String e) {
  if (i < 0 || i > size) {
    throw new IllegalArgumentException("Invalid Index.");
  }
  else {
    if (i == 0) {
      addFirst(e);
    }
    else {
      Node nodeBefore = getNodeAt(i - 1);
      Node newNode = new Node(e, nodeBefore.getNext());
      nodeBefore.setNext(newNode);
      size ++;
    }
  }
}
```

Example. See `testSLL_addAt` in `ExampleStringLinkedLists.zip`.

Singly-Linked List: Inserting to the Middle (2)

- A call to `addAt(i, e)` may end up executing:
  - Line 3 (throw exception) [$O(1)$]
  - Line 7 (addFirst) [$O(1)$]
  - Lines 10 (getNodeAt) [$O(n)$]
  - Lines 11 – 13 (setting references) [$O(1)$]
- What is the ***worst case*** of the index `i` for `addAt(i, e)`?

  A. `list.addAt(list.getSize(), e)`

  which requires list.getNodeAt(list.getSize() - 1)
- RT of `addAt` is $O(n)$ [linear-time op.]
- **Contrast**: Inserting into an array costs $O(n)$ [linear-time op.]

  For arrays, when given the *index* to an element, the RT of inserting an element is always $O(n)$!

Singly-Linked List: Removing from the End

Problem: Remove the last node (i.e., tail) of the list.

Hint. Using ***tail*** sufficient? Use `getNodeAt(?)` as a helper?

- Assume we are in the context of class `SinglyLinkedList`.

```
void removeLast () {
  if (size == 0) {
    throw new IllegalArgumentException("Empty List.");
  }
  else if (size == 1) {
    removeFirst();
  }
  else {
    Node secondLastNode = getNodeAt(size - 2);
    secondLastNode.setNext(null);
    tail = secondLastNode;
    size --;
  }
}
```

Running time? $O(n)$

Singly-Linked List: Exercises

Consider the following two linked-list operations, where a *reference node* is given as an input parameter:

- `void insertAfter(Node n, String e)`
  - Steps?
    - *Create a new node* `nn`*.*
    - *Set* `nn`*'s next to* `n`*'s next.*
    - *Set* `n`*'s next to* `nn`*.*
  - Running time? [*O(1)*]
- `void insertBefore(Node n, String e)`
  - Steps?
    - *Iterate from the* `head`*, until* `current.next == n`*.*
    - *Create a new node* `nn`*.*
    - *Set* `nn`*'s next to* `current`*'s next (which is* `n`*).*
    - *Set* `current`*'s next to* `nn`*.*
  - Running time? [*O(n)*]

Exercise

- Complete the Java ***implementation***, ***tests***, and ***running time analysis*** for `void removeAt(int i)`.

Arrays vs. Singly-Linked Lists

Operation \ Data Structure		Array	Singly-Linked List
get size		O(1)	O(1)
get first/last element		O(1)	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)

Background Study: Generics in Java

- It is assumed that, in EECS2030, you learned about the basics of Java ***generics***:
  - General collection (e.g., `Object[]`) vs. Generic collection (e.g., `E[]`)
  - How using generics minimizes ***casts*** and ***`instanceof` checks***
  - How to <u>implement</u> and <u>use</u> generic classes
- If needed, review the above assumed basics from the relevant parts of EECS2030 (`https://www.eecs.yorku.ca/~jackie/teaching/lectures/index.html#EECS2030_F21`):
  - Parts A1 – A3, Lecture 7, Week 10
  - Parts B – C, Lecture 7, Week 11

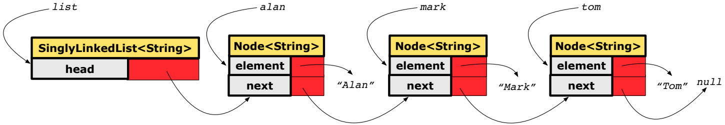
  Tips.
  - Skim the ***slides***: watch lecture videos if needing explanations.
  - Ask questions related to the assumed basics of ***generics***!
- Assuming that know the basics of Java ***generics***, we will <u>implement</u> and <u>use</u> ***generic SLL*** and ***DLL***.

Generic Classes: Singly-Linked List (1)

```
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; }
}
```

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

Generic Classes: Singly-Linked List (2)



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);
```

Generic Classes: Singly-Linked List (3)

Assume we are in the context of class `SinglyLinkedList`.

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

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

Singly-Linked Lists: Handling Edge Cases

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

```
void removeFirst () {
  if (size == 0) { /* error */ }
  else if (size == 1) {
    head = null; tail = null; size --; }
  else {
    Node<E> oldHead = head;
    head = oldHead.getNext();
    oldHead.setNext(null); size --;
  } }
```

- We have to ***explicitly*** deal with special cases where the ***current list*** or ***resulting list*** is empty.
- We can actually resolve this issue via a ***small extension***!

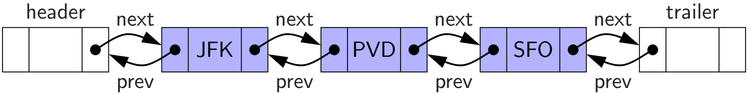
Basic Data Structure: Doubly-Linked Lists (1)

- We know that *singly-linked* lists perform:
  - **WELL**: [*O(1)*]
    - inserting to the front/end [***head***/***tail***]
    - removing from the front [***head***]
    - inserting/deleting the middle [given ref. to previous node]
  - **POORLY**: [*O(n)*]
    - accessing the middle [`getNodeAt(i)`]
    - removing from the end [`getNodeAt(list.getSize() - 2)`]
- We may again improve the performance by

  trading space for time

  just like how attributes ***size*** and ***tail*** were introduced.

Basic Data Structure: Doubly-Linked Lists (2)



- Each ***node*** in a *doubly-linked list (DLL)* stores:
  - A ***reference*** to an element of the sequence
  - A ***reference*** to the next node in the list
  - A ***reference*** to the *previous node* in the list [SYMMETRY]
- Each ***DLL*** stores:
  - A ***reference*** to a dedicated *header node* in the list
  - A ***reference*** to a dedicated *trailer node* in the list

  Remark. Unlike SLL, ***DLL*** does not store refs. to ***head*** and ***tail***.
- These two special nodes are called ***sentinels*** or ***guards***:
  - They do **not** store data, but store node references:
    - The ***header node*** stores the **next** reference only
    - The ***trailer node*** stores **previous** reference only
  - They **always** exist, even in the case of empty lists.

Generic Doubly-Linked Lists in Java (1)

```
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;
  }
}
```

Generic Doubly-Linked Lists in Java (2)

```java
1  public class DoublyLinkedList<E> {
2    private int size = 0;
3    public void addFirst(E e) { ... }
4    public void removeLast() { ... }
5    public void addAt(int i, E e) { ... }
6    private Node<E> header;
7    private Node<E> trailer;
8    public DoublyLinkedList() {
9      header = new Node<>(null, null, null);
10     trailer = new Node<>(null, header, null);
11     header.setNext(trailer);
12   }
13 }
```

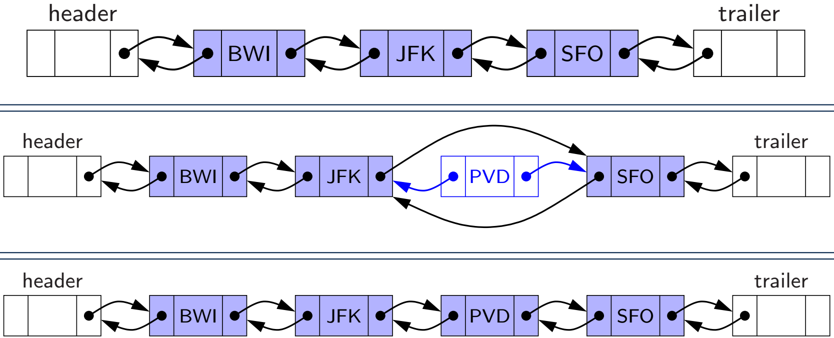
Lines 8 to 10 are equivalent to:

```java
header = new Node<>(null, null, null);
trailer = new Node<>(null, null, null);
header.setNext(trailer);
trailer.setPrev(header);
```

Header, Trailer, and `prev` Reference

- The ***`prev`* reference** helps ***improve the performance*** of `removeLast()`.
  - ∵ The ***second last node*** can be accessed in *constant time*.
    [`trailer.getPrev().getPrev()`]
- The two ***sentinel/guard*** nodes (***header*** and ***trailer***) do **not** help improve the performance.
  - Instead, they help *simplify the logic* of your code.
  - Each insertion/deletion can be treated
    - *Uniformly* : a node is always inserted/deleted in-between two nodes
    - Without worrying about re-setting the ***head*** and ***tail*** of list

Doubly-Linked List: Insertions



Doubly-Linked List: Inserting to Front/End

```
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 ++;
}
```

Running Time? $O(1)$

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

Running Time? $O(1)$

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

Running Time? $O(1)$

Doubly-Linked List: Inserting to Middle

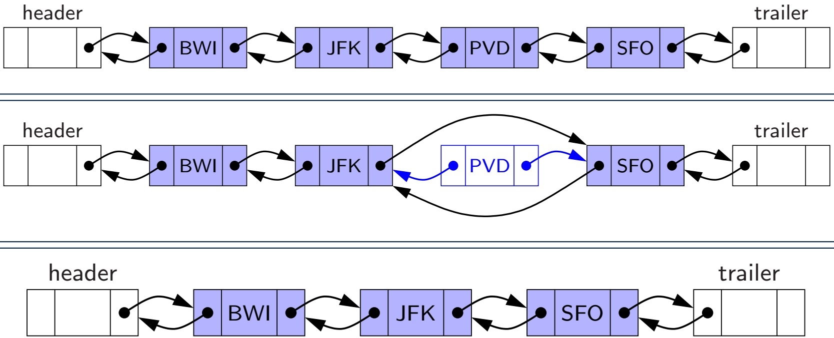
```
void addBetween(E e, Node<E> pred, Node<E> succ) {
  Node<E> newNode = new Node<>(e, pred, succ);
  pred.setNext(newNode);
  succ.setPrev(newNode);
  size ++;
}
```

Running Time? $O(1)$

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

Running Time? Still $O(n)$!!!

Doubly-Linked List: Removals



Doubly-Linked List: Removing from Front/End

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

Running Time? $O(1)$

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

Running Time? $O(1)$

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

Running Time? Now $O(1)$!!!

Doubly-Linked List: Removing from Middle

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

Running Time? $O(1)$

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

Running Time? Still $O(n)$!!!

Reference Node: To be Given or Not to be Given

Exercise 1: Compare the steps and running times of:

- *Not given* a reference node:
  - `addNodeAt(int i, E e)` [$O(n)$]
- *Given* a reference node:
  - `addNodeBefore(Node<E> n, E e)` [SLL: $O(n)$; DLL: $O(1)$]
  - `addNodeAfter(Node<E> n, E e)` [$O(1)$]

Exercise 2: Compare the steps and running times of:

- *Not given* a reference node:
  - `removeNodeAt(int i)` [$O(n)$]
- *Given* a reference node:
  - `removeNodeBefore(Node<E> n)` [SLL: $O(n)$; DLL: $O(1)$]
  - `removeNodeAfter(Node<E> n)` [$O(1)$]
  - `removNode(Node<E> n)` [SLL: $O(n)$; DLL: $O(1)$]

Arrays vs. (Singly- and Doubly-Linked) Lists

<table>
<tr><th colspan="2">DATA STRUCTURE / OPERATION</th><th>ARRAY</th><th>SINGLY-LINKED LIST</th><th>DOUBLY-LINKED LIST</th></tr>
<tr><td colspan="2">size</td><td colspan="3" rowspan="2">O(1)</td></tr>
<tr><td colspan="2">first/last element</td></tr>
<tr><td colspan="2">element at index i</td><td>O(1)</td><td rowspan="2">O(n)</td><td>O(n)</td></tr>
<tr><td colspan="2">remove last element</td><td rowspan="4">O(n)</td><td rowspan="3">O(1)</td></tr>
<tr><td colspan="2">add/remove first element, add last element</td><td rowspan="2">O(1)</td></tr>
<tr><td rowspan="2">add/remove i^{th} element</td><td>given reference to $(i-1)^{th}$ element</td></tr>
<tr><td>not given</td><td colspan="2">O(n)</td></tr>
</table>

Beyond this lecture . . .

- In Eclipse, ***implement*** and ***test*** the assigned methods in `SinglyLinkedList` class and `DoublyLinkedList` class.
- Modify the ***insertion sort*** and ***selection sort*** implementations using a SLL or DLL.

Index (1)

Index (2)

Index (3)

Index (4)

Index (5)