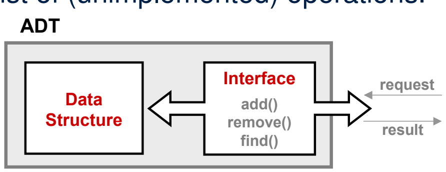
ADTs, Arrays, and Linked-Lists

EECS2030: Advanced Object Oriented Programming
Fall 2017

CHEN-WEI WANG

Abstract Data Types (ADTs)

- Given a problem, you are required to filter out *irrelevant* details.
- The result is an *abstract data type (ADT)*, whose *interface* consists of a list of (unimplemented) operations.



- *Supplier*'s *Obligations*:
  - Implement all operations
  - Choose the "right" data structure (DS)
- *Client*'s *Benefits*:
  - Correct output
  - Efficient performance
- The internal details of an *implemented ADT* should be **hidden**.

Standard ADTs

- *Standard* ADTs are *reusable components* that have been adopted in solving many real-world problems.

  e.g., Stacks, Queues, Lists, Tables, Trees, Graphs
- You will be required to:
  - *Implement* standard ADTs
  - *Design* algorithms that make use of standard ADTs
- For each standard ADT, you are required to know:
  - The list of supported operations (i.e., *interface*)
  - Time (and sometimes space) *complexity* of each operation
- In this lecture, we learn about two *basic data structures*:
  - arrays
  - linked lists

Basic Data Structure: Arrays

- An array is a sequence of indexed elements.
- *Size* of an array is **fixed** at the time of its construction.
- Supported *operations* on an array:
  - *Accessing*: e.g., `int max = a[0];`

    Time Complexity: *O(1)* [constant operation]
  - *Updating*: e.g., `a[i] = a[i + 1];`

    Time Complexity: *O(1)* [constant operation]
  - *Inserting/Removing*:

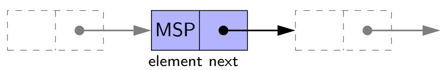
```
insertAt(String[] a, int n, String e, int i)
  String[] result = new String[n + 1];
  for(int j = 0; j < i; j ++){ result[i] = a[i]; }
  result[i] = e;
  for(int j = i + 1; j < n; j ++){ result[j] = a[j - 1]; }
  return result;
```

Time Complexity: *O(n)* [linear operation]

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* that collectively form a *linear sequence*.
- Each node in a *singly-linked* list has:
  - A *reference* to an *element of the sequence*
  - A *reference* to the *next node* in the list

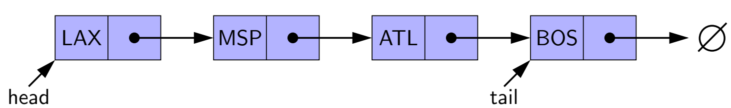
  Contrast this *relative* positioning with the *absolute* indexing of arrays.



- The *last element* 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, we can use a singly-linked list to *dynamically* store as many elements as we desire.
  - By creating a *new node* and setting the relevant *references*.
  - e.g., inserting an element to the beginning/middle/end of a list
  - e.g., deleting an element from the list requires a similar procedure
- *Contrary to the case of arrays*, we simply *cannot* keep track of all nodes in a lined list *directly* by indexing the *next* references.
- Instead, we only store a reference to the *head* (i.e., *first node*), and find other parts of the list *indirectly*.



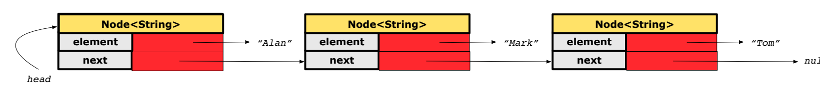
- **Exercise**: Given the *head* reference of a singly-linked list:
  - Count the number of nodes currently in the list [Running Time?]
  - Find the reference to its *tail* (i.e., last element) [Running Time?]

Singly-Linked List: Java Implementation

```
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 = null;
  public void addFirst(String e) { ... }
  public void removeLast() { ... }
  public void addAt(int i, String e) { ... }
}
```


Singly-Linked List: A Running Example



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);
```

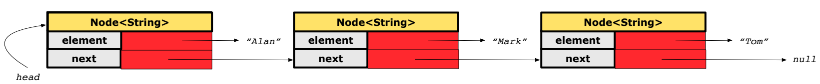
Singly-Linked List: Counting # of Nodes (1)

- 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      /* exit when current == null */
6      current = current.getNext();
7      size ++;
8    }
9    return size;
10 }
```

- When does the *while loop* (Line 4) terminate? `current` is null
- Only the *last node* has a null *next* reference.
- RT of `getSize` *O(n)* [linear operation]
- **Contrast**: RT of `a.length` is *O(1)* [constant]

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;
}
```

`current`	`current != null`	Beginning of Iteration	`size`
Alan	*true*	1	1
Mark	*true*	2	2
Tom	*true*	3	*3*
`null`	*false*	–	–

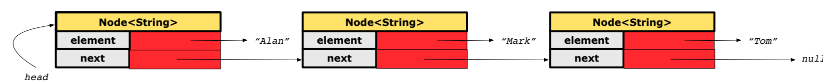
Singly-Linked List: Finding the Tail (1)

- 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      /* exit when current == null */
6      tail = current;
7      current = current.getNext();
8    }
9    return tail;
10 }
```

- When does the *while loop* (Line 4) terminate? `current` is null
- Only the *last node* has a null *next* reference.
- RT of `getTail` is *O(n)* [linear operation]
- **Contrast**: RT of `a[a.length - 1]` is *O(1)* [constant]

Singly-Linked List: Finding the Tail (2)



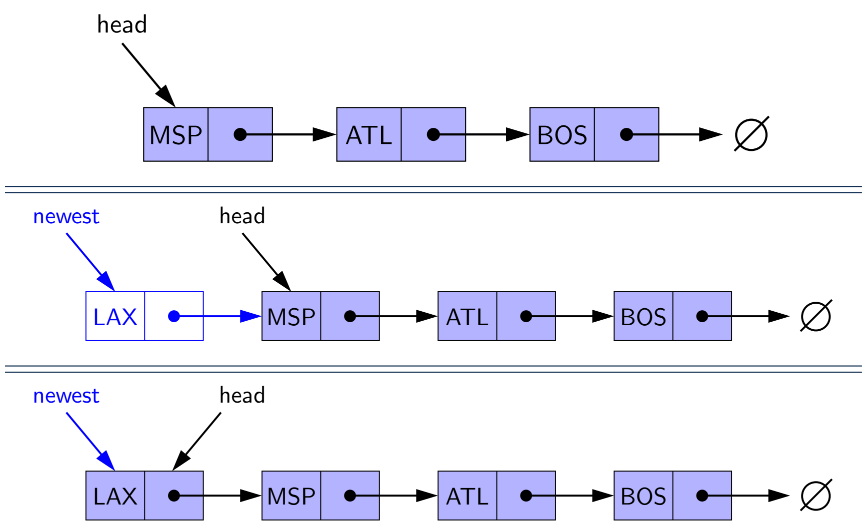

```
Node getTail() {
  Node current = head;
  Node tail = null;
  while (current != null) { /* exit when current == null */
    tail = current;
    current = current.getNext();
  }
  return tail;
}
```

`current`	`current != null`	Beginning of Iteration	`tail`
Alan	*true*	1	Alan
Mark	*true*	2	Mark
Tom	*true*	3	*Tom*
`null`	*false*	–	–

Singly-Linked List: Can We Do Better?

- It is frequently needed to
  - access the *tail* of list [e.g., a new customer joins service queue]
  - query about its *size* [e.g., is the service queue full?]
- How can we improve the *running time* of these two operations?
- We may trade *space* for *time*.
- In addition to *head*, we also declare:
  - A variable *tail* that points to the end of the list
  - A variable *size* that keeps tracks of the number of nodes in list
  - Running time of these operations are both $O(1)$!
- Nonetheless, we cannot declare variables to store references to *nodes in-between* the head and tail. Why?
  - 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)



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

- 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 operation]
- **Contrast**: RT of inserting into an array is *O(n)* [linear]

Your Homework

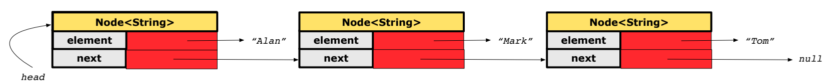
- Complete the Java *implementations* and *running time analysis* for `removeFirst()`, `addLast(E e)`.
- **Question:** *The `removeLast()` method may not be completed in the same way as is `addLast(String e)`. Why?*

Singly-Linked List: Accessing the Middle (1)

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

```
Node getNodeAt (int i) {
  if (i < 0 || i >= size) {
    throw 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) { /* print 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 operation]
- **Contrast**: RT of accessing an array element is *O(1)* [constant]

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

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

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

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)`?
- Worst case: `list.addAt(list.getSize() - 1, e)`
- RT of `addAt` is *O(n)* [linear operation]
- **Contrast**: RT of inserting into an array is *O(n)* [linear]
- On the other hand, 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

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

```
void removeLast () {
  if (size == 0) {
    System.err.println("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)*]

Your Homework

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

Arrays vs. Singly-Linked Lists

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

Index (1)

Index (2)