Data Structures and Algorithms

Lecture 2: Linked Lists

Department of Computer Science & Technology
United International College

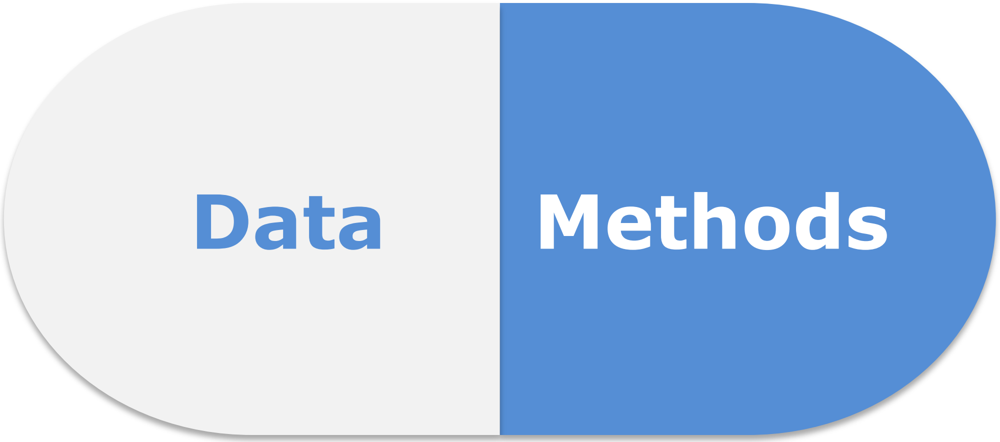
Outline

- Abstract Data Type (ADT)
- List ADT
- Linked lists

Abstract Data Type

- Data type = Data + Operation
  - Example 1: integer
    - Data: a whole number
    - Operations: +, -, x, /, …
  - Example 2: string
    - Data: an array of characters
    - Operations: strlen, strcpy, strcat, strcmp, …
- Can this be generalized?
  - Abstract Data Type (ADT)
  - Encapsulation

Encapsulation - What



Data

Methods

- Users of Data
  - do not touch data directly
  - operates on data by calling the methods
  - do not know how the methods are implemented

Encapsulation - Why

- Modular: one module for one ADT
  - Implementation of the ADT is separate from its use
  - Allows parallel development
  - Easier to debug
- Code for the ADT can be reused in different applications
- Information hiding
  - Protect data from unwanted operations
  - implementation details can be changed without affecting user programs
- Allow rapid prototyping
  - Prototype with simple ADT implementations, then tune them later when necessary

Encapsulation - How

- In OOP Languages:
  - ADT: Class
  - Data: member variables
  - Methods: member functions
- In C:
  - Data: variables (usually of a struct data type)
  - Methods: functions
  - *Information hiding is not supported in C*

The List ADT - Data

- A sequence of zero or more elements

  $A_1, A_2, A_3, \ldots A_N$

  - N: length of the list
  - A_1: first element
  - A_N: last element
  - A_i: element at position i
  - If N=0, then it is an empty list
- Linearly ordered
  - A_i precedes A_{i+1}
  - A_i follows A_{i-1}
- The elements can be of any data type but we use double for this discussion

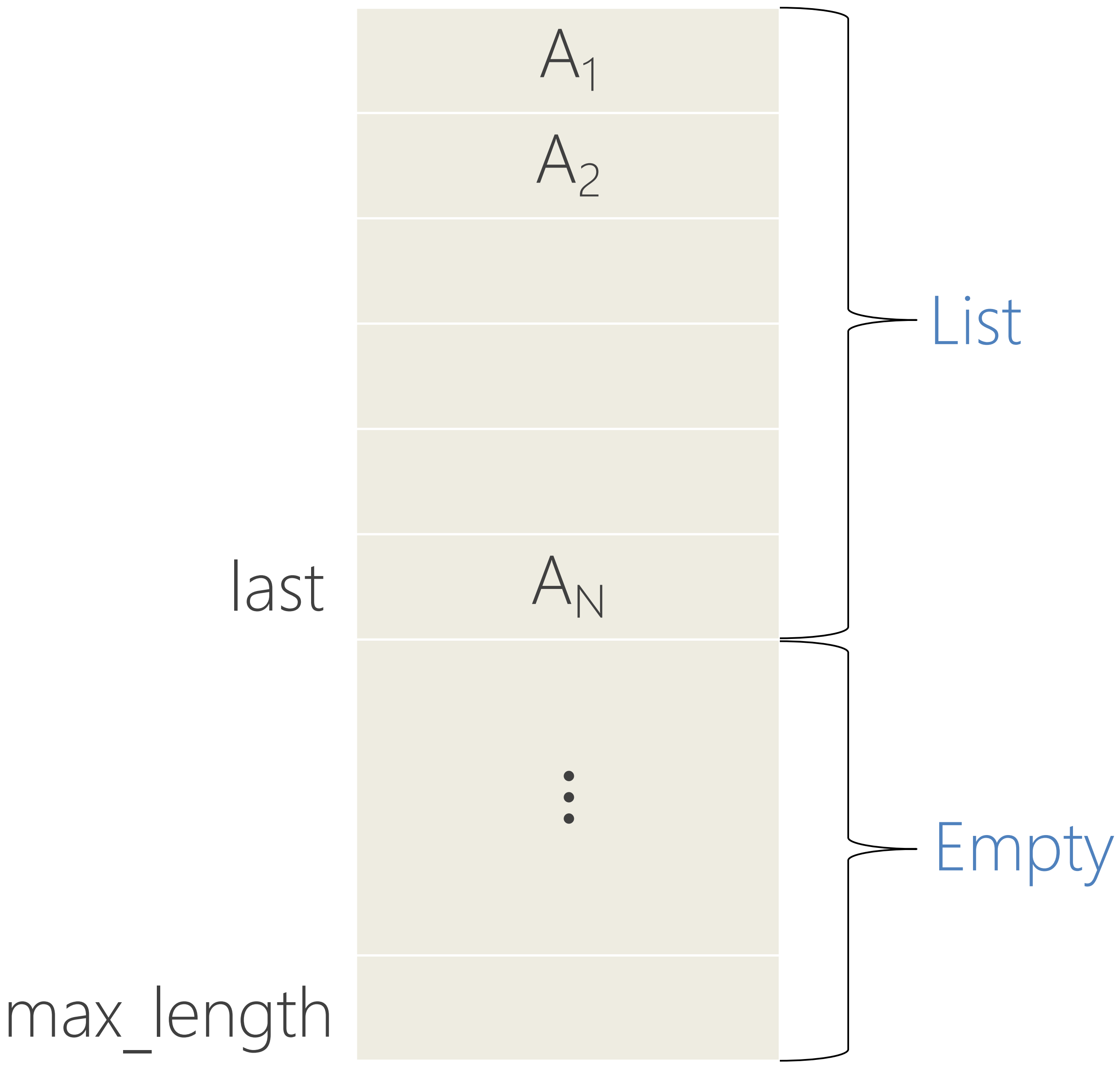
The List ADT - Operations

- makeEmpty: create an empty list
- insert: insert an object into a list
  - insert(x,3) → 34, 12, 52, x, 16, 12
- remove: delete an element from the list
  - remove(52) → 34, 12, x, 16, 12
- find: locate the position of an object in a list
  - list: 34, 12, 52, 16, 12
  - find(52) → 3
- findKth: retrieve the element at a certain position
- printList: print the list

Implementation of an ADT

- Define data using data types
- Define operation using functions
- Two standard implementations for the list ADT
  - Array-based
  - Linked list

Array Implementation

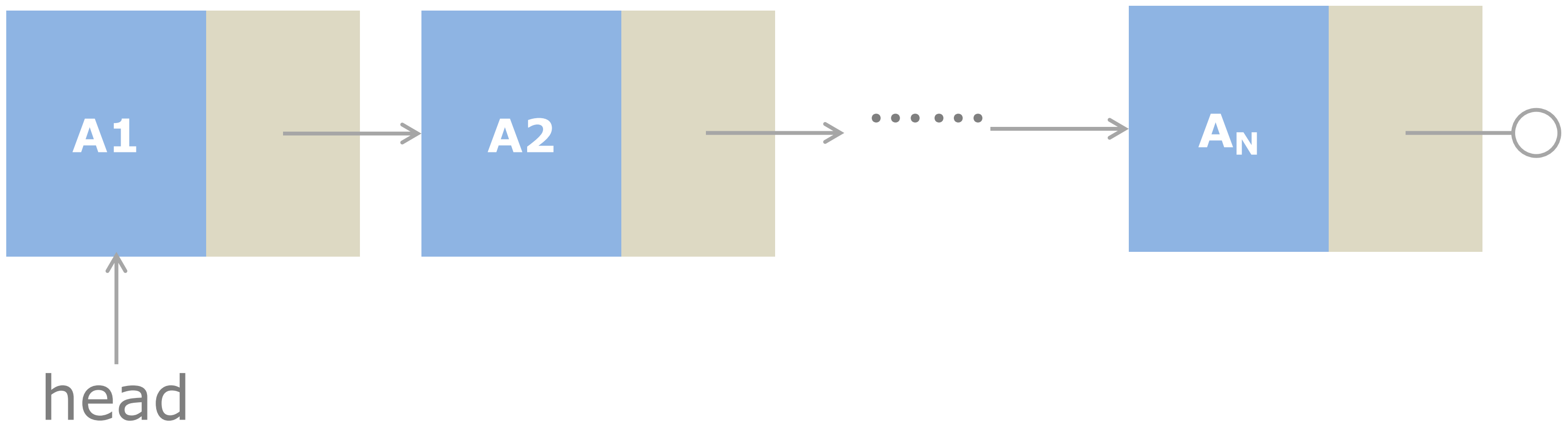


Discussion

- Are these operations suitable for the array implementation?
  - insert(x, n)
  - remove(x)
  - find(x)
  - findKth(k)
- Any additional pros and cons?
  - Space?

Pointer Implementation (Linked List)

- Ensure that the list is not stored contiguously
  - A node stores one element
  - The address of a node is stored in its previous node
  - The address of the first node must be stored



Discussion

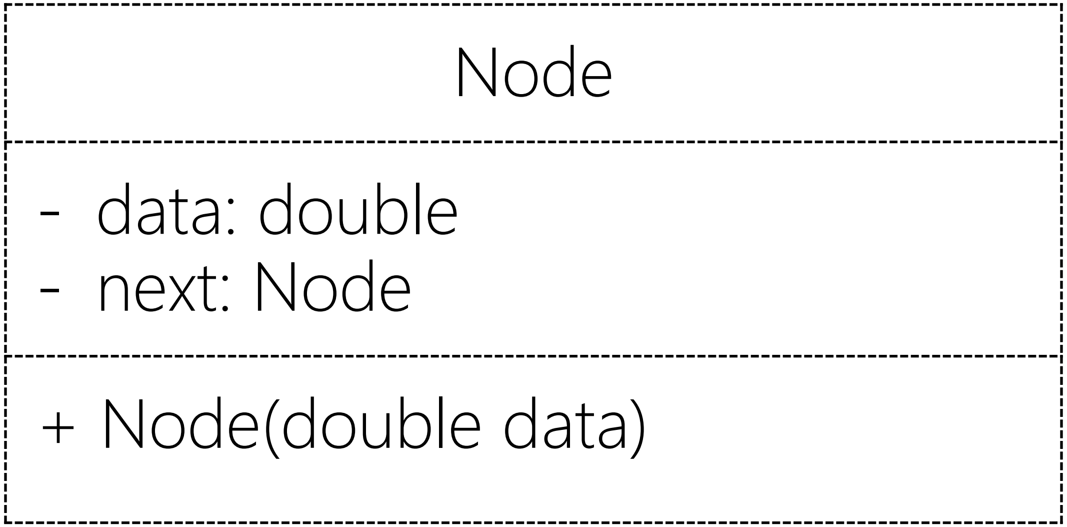
- Are these operations suitable for the linked list?
  - insert(x, n)
  - remove(x)
  - find(x)
  - findKth(k)
- Any additional pros and cons?
  - Space?

A Complete list of Comparison

Topic		Array	Linked List
Efficiency	insert		
	remove		
	find		
	findKth		
space			

Linked List Implementation

Class: Node



Node

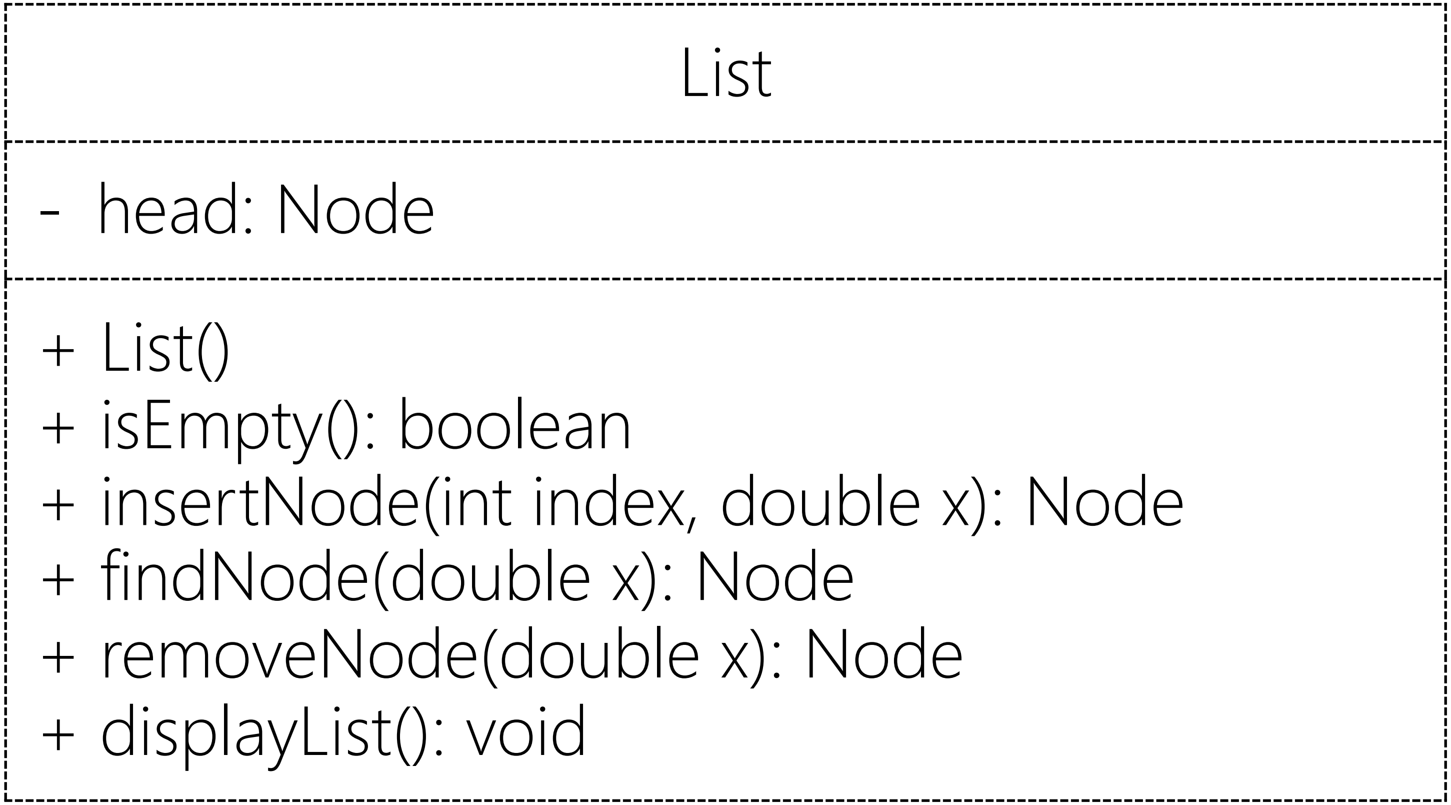
- data: double
- next: Node

+ Node(double data)

Setters and getters are not listed.

Linked List Implementation

Class: List



List

- head: Node

+ List()
+ isEmpty(): boolean
+ insertNode(int index, double x): Node
+ findNode(double x): Node
+ removeNode(double x): Node
+ displayList(): void

Setters and getters are not listed.

Methods

- public List()
  - creates an empty list
- public boolean isEmpty()
  - returns *true* if the list is empty and *false* otherwise
- public Node insertNode(int index, double x)
  - insert a new node after position *index*
  - position of nodes starts from 1
  - insert a new node as the head if *index=0*
  - returns the new node if insertion is successful and *null* otherwise

Methods

- public Node findNode(double x)
  - returns the first node whose *data=x*
  - returns *null* if no such node exists
- public Node removeNode(double x)
  - removes from the list the first node whose *data=x*
  - returns the removed node
  - returns *null* if no such node exists
- public void displayList()
  - prints all the nodes in the list

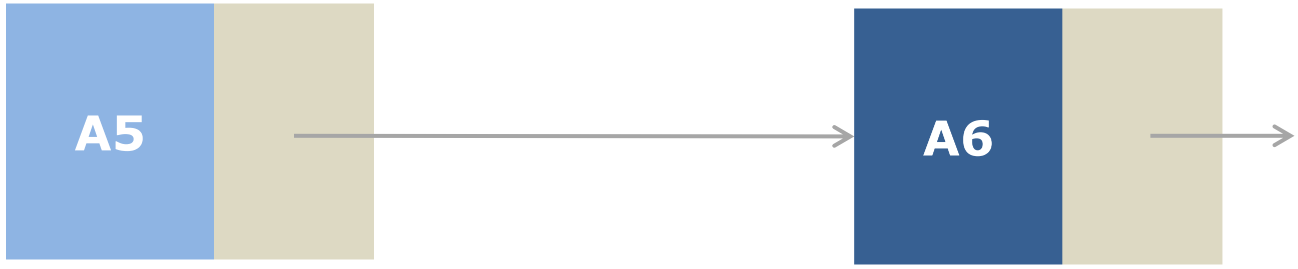
Insert

- public Node insertNode(int index, double x)
  - insert a new node after position *index*
  - position of nodes starts from 1
  - insert a new node as the head if *index=0*
  - returns the new node if insertion is successful and *null* otherwise

Insert

1. Locate the element at position *index*
2. Create a new Node object
3. Point the new node to its successor
4. Point the new node's predecessor to the new node

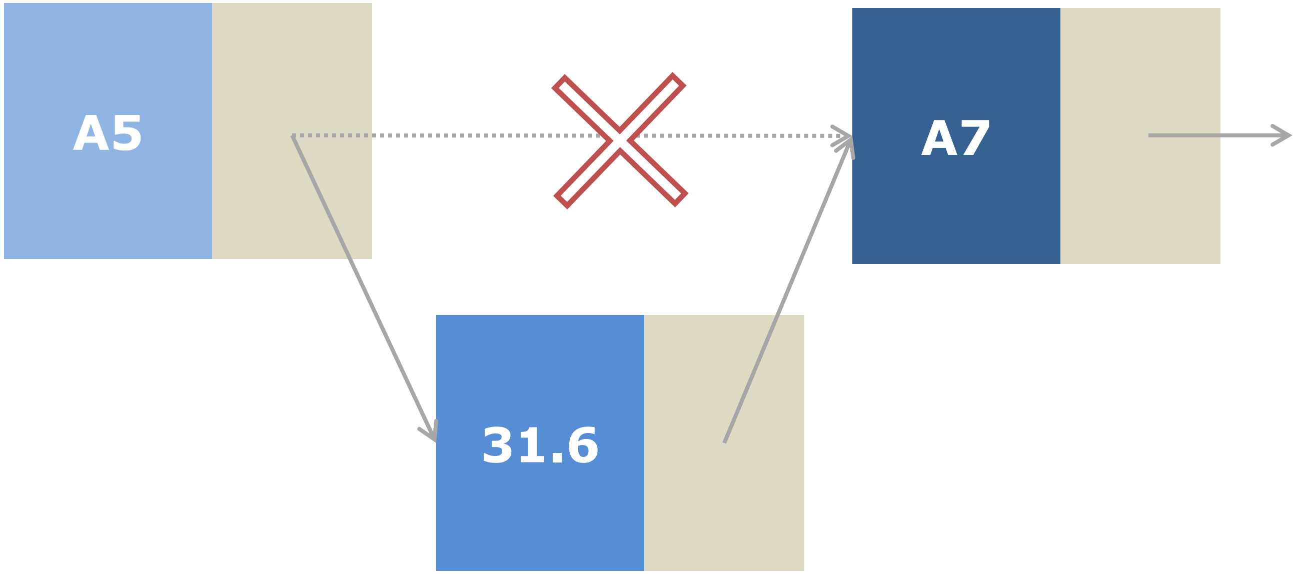
insertNode(5, 31.6)



Insert

1. Locate the element at position *index*
2. Allocate memory for the new node
3. Point the new node to its successor
4. Point the new node's predecessor to the new node

insertNode(&head, 5, 31.6)



Insert

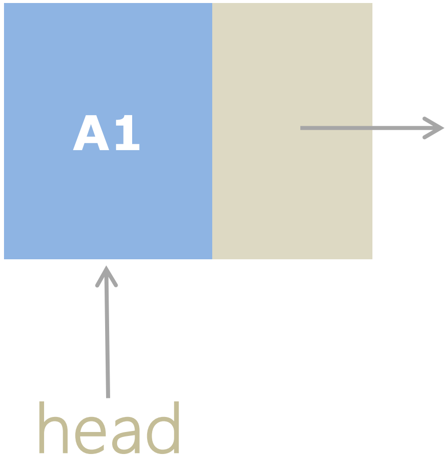
- Possible cases of `insertNode`
  1. Insert into an empty list
  2. Insert in front
  3. Insert at back
  4. Insert in middle
- But, in fact, only need to handle two cases
  - Insert as the first node (Case 1 and Case 2)
  - Insert in the middle or at the end of the list (Case 3 and Case 4)

Two Cases for Insert

- Insert as the first node
  - handles the *next* of one node
  - updates the *head* of the list
- Insert in the middle or at the end
  - handles the *next* of two nodes

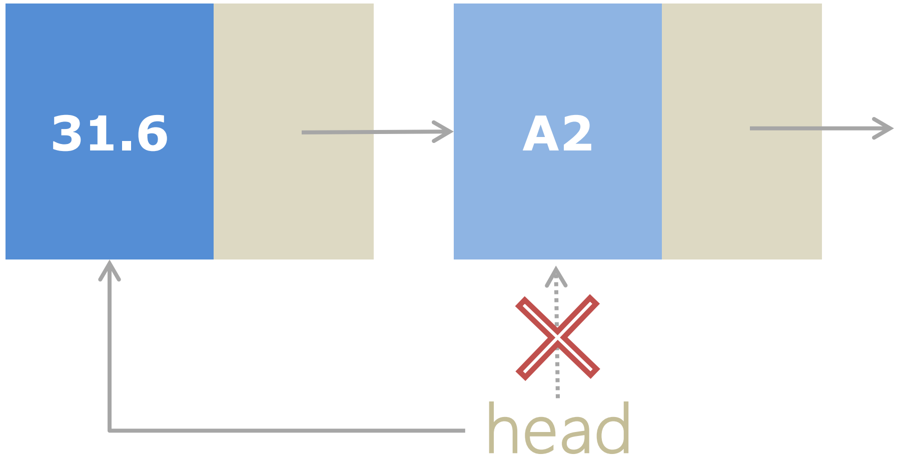
Insert as the First Node

insertNode(0, 31.6)



Insert as the First Node

insertNode(0, 31.6)



Code for Insert

```java
public Node insertNode(int index, double x) {
        if(index < 0)
                return null;
        int currIndex = 1;
        Node currNode = this.head;
        while(currNode != null && index > currIndex) {
                currNode = currNode.getNext();
                currIndex ++;
        }
        if(index > 0 && currNode == null)
                return null;

        Node newNode = new Node(x);
        if(index == 0) {
                newNode.setNext(this.head);
                this.head = newNode;
        }
        else {
                newNode.setNext(currNode.getNext());
                currNode.setNext(newNode);
        }
        return newNode;
}
```

```java
public Node insertNode(int index, double x) {
        if(index < 0)
                return null;
        int currIndex = 1;
        Node currNode = this.head;
        while(currNode != null && index > currIndex) {
                currNode = currNode.getNext();
                currIndex ++;
        }
        if(index > 0 && currNode == null)
                return null;

        Node newNode = new Node(x);
        if(index == 0) {
                newNode.setNext(this.head);
                this.head = newNode;
        }
        else {
                newNode.setNext(currNode.getNext());
                currNode.setNext(newNode);
        }
        return newNode;
}
```

Try to locate the node at position *index*. If it does not exist, return *null*.

```
public Node insertNode(int index, double x) {
        if(index < 0)
                return null;
        int currIndex = 1;
        Node currNode = this.head;
        while(currNode != null && index > currIndex) {
                currNode = currNode.getNext();
                currIndex ++;
        }
        if(index > 0 && currNode == null)
                return null;

        Node newNode = new Node(x);
        if(index == 0) {
                newNode.setNext(this.head);
                this.head = newNode;
        }
        else {
                newNode.setNext(currNode.getNext());
                currNode.setNext(newNode);
        }
        return newNode;
}
```

Create a new Node.

```java
public Node insertNode(int index, double x) {
        if(index < 0)
                return null;
        int currIndex = 1;
        Node currNode = this.head;
        while(currNode != null && index > currIndex) {
                currNode = currNode.getNext();
                currIndex ++;
        }
        if(index > 0 && currNode == null)
                return null;

        Node newNode = new Node(x);
        if(index == 0) {
                newNode.setNext(this.head);
                this.head = newNode;
        }
        else {
                newNode.setNext(currNode.getNext());
                currNode.setNext(newNode);
        }
        return newNode;
}
```

Insert as the new head.

```
public Node insertNode(int index, double x) {
        if(index < 0)
                return null;
        int currIndex = 1;
        Node currNode = this.head;
        while(currNode != null && index > currIndex) {
                currNode = currNode.getNext();
                currIndex ++;
        }
        if(index > 0 && currNode == null)
                return null;

        Node newNode = new Node(x);
        if(index == 0) {
                newNode.setNext(this.head);
                this.head = newNode;
        }
        else {
                newNode.setNext(currNode.getNext());
                currNode.setNext(newNode);
        }
        return newNode;
}
```

Insert after currNode.

Find

- Node findNode(double x)
  - returns the first node whose *data=x*
  - returns *null* if no such node exists
- Steps
  1. Search for a node with the value equal to `x` in the list.
  2. If such a node is found, return it. Otherwise, return *null*.

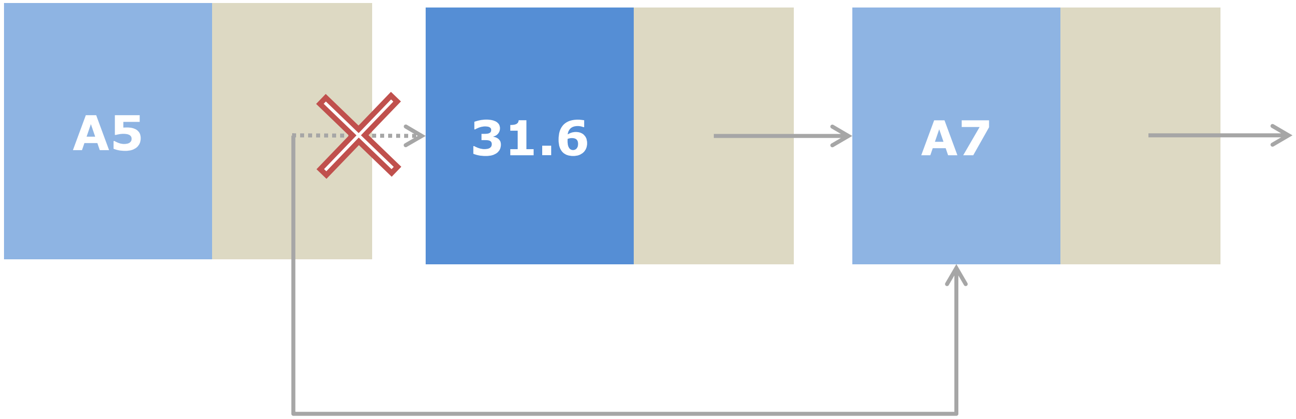
Remove

- Node removeNode(double x)
  - removes a node from the list whose *data=x*
  - returns the removed node
  - returns *null* if no such node exists
- Steps
  1. Find the desirable node (similar to *FindNode*)
  2. In addition, record the node's predecessor
  3. Set the *next* pointers
  4. Return the removed node

Remove

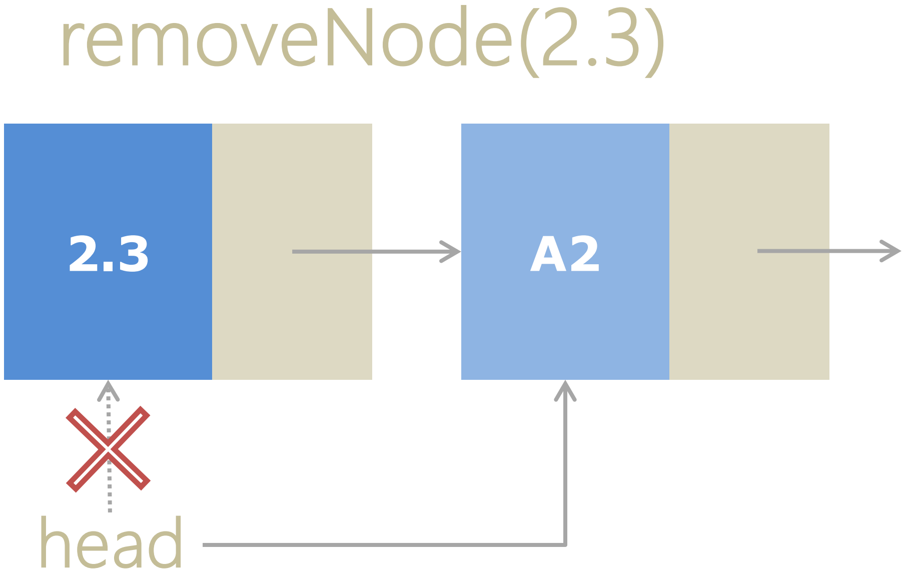
- Deleting a middle or an end node

removeNode(31.6)



Remove

- Removing the head



Task

- Given *Node.java*, complete *List.java* with
  - all the complete functions defined
  - a main function has been given for the class which tests 5 functions:
    - isEmpty
    - insertNode
    - findNode
    - removeNode
    - displayList
- Submit *List.java* to iSpace.