Chapter 6

Data Structures: Linked Lists

CSC 113
King Saud University
College of Computer and Information Sciences
Department of Computer Science

Dr. S. HAMMAMI

1. Objectives

After you have read and studied this chapter, you should be able to:

- Understand the concept of a dynamic data structure.
- Be able to create and use dynamic data structures such as linked lists.
- Understand the stack and queue ADTs.
- Various important applications of linked data structures.
- Know how to use inheritance to define extensible data structures.
- Create reusable data structures with classes, inheritance and composition.

2. Self-Referential Classes: Definition

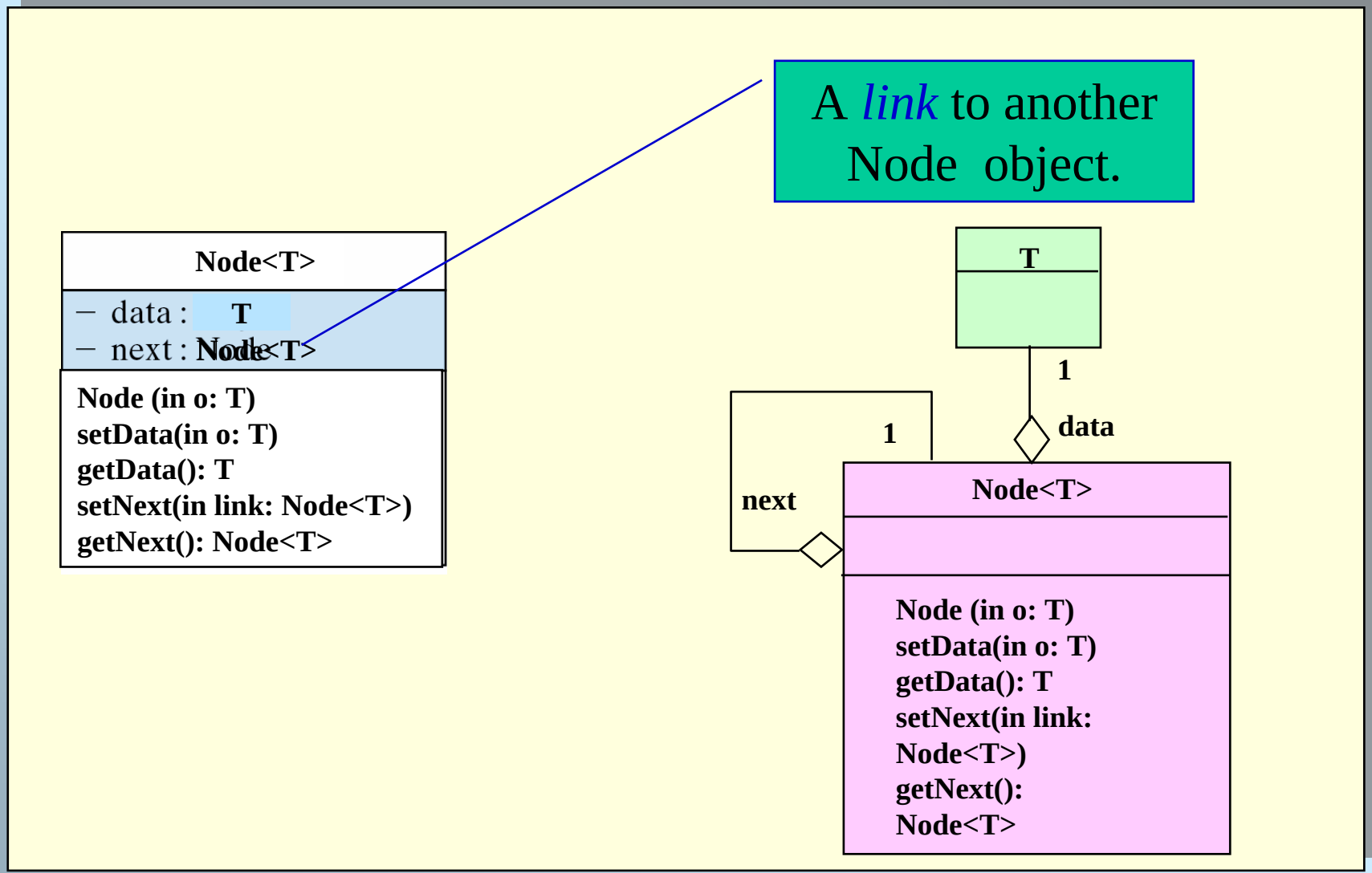
- **Self-referential class**

  Contains an instance variable that refers to another object of the same class type

  That instance variable is called a link

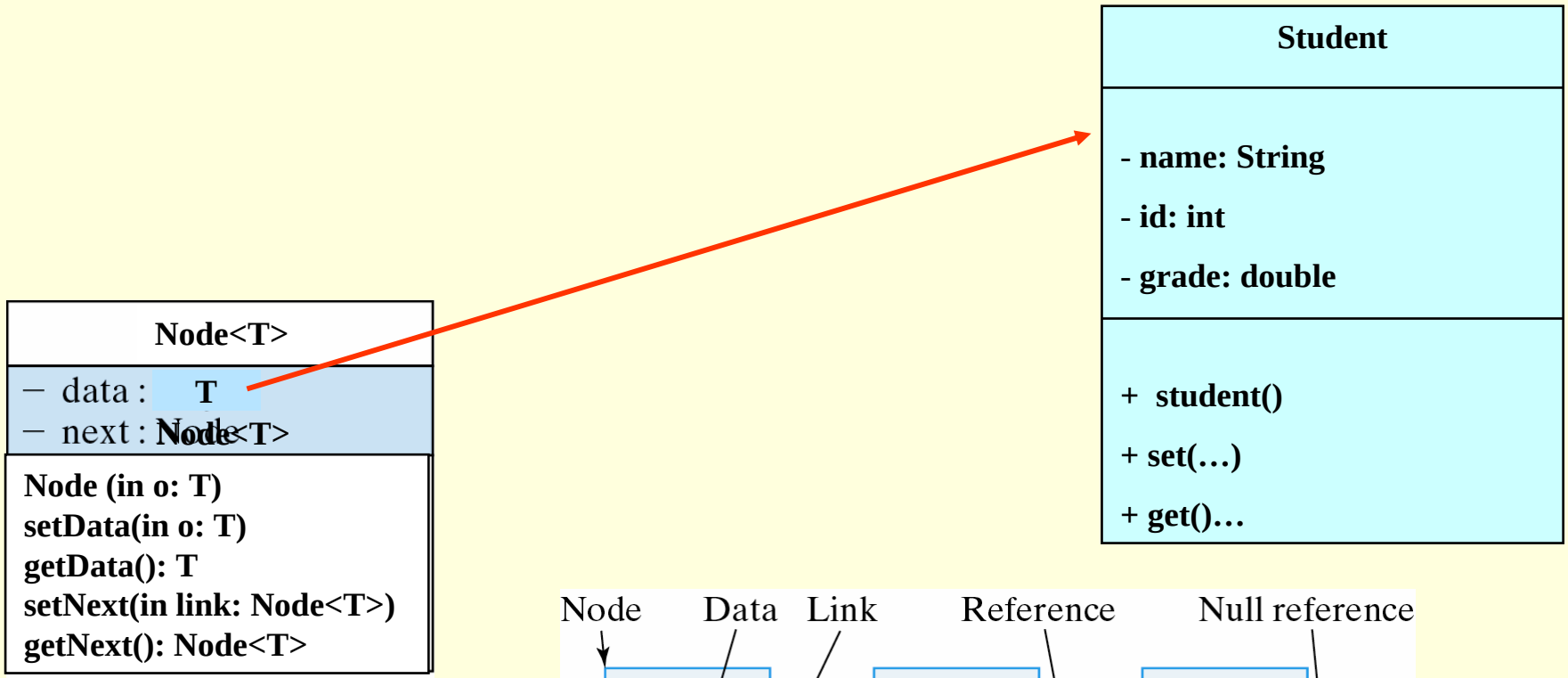
  A `null` reference indicates that the link does not refer to another object

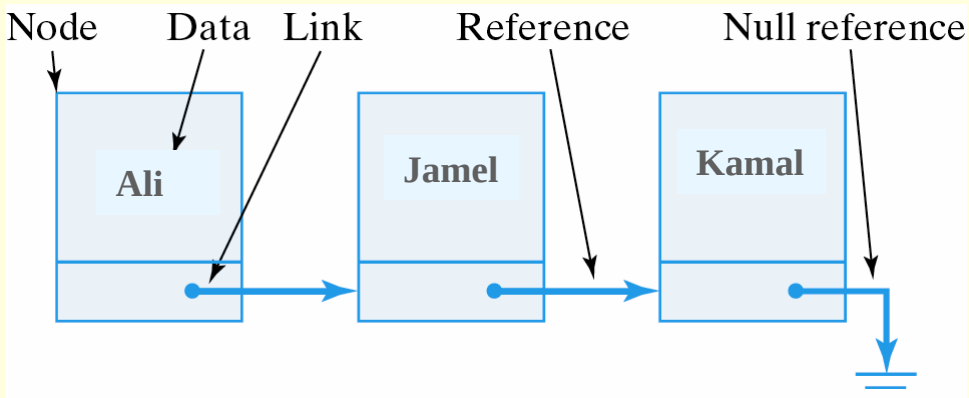
2. Self-Referential Classes (cont)



Basic Node: The Generic Node Class

- **A node in a linked list contains data elements and link elements.**





Generic Node Class: Implementation

```
public class Node<T> {
    private T data;  // Stores any kind of data
    private Node<T> next;
    public Node<T>(T obj) {
        data = obj;
        next = null;
    }
    public void setNext( Node<T> nextPtr ) {
        next = nextPtr;
    }
    public Node<T> getNext() {
        return next;
    }
    public void setData(T obj) {
        data = obj;
    }
    public T getData() {
        return data;
    }
    public String toString() {
        return data.toString();
    }
}
```

Connecting two nodes

The statements

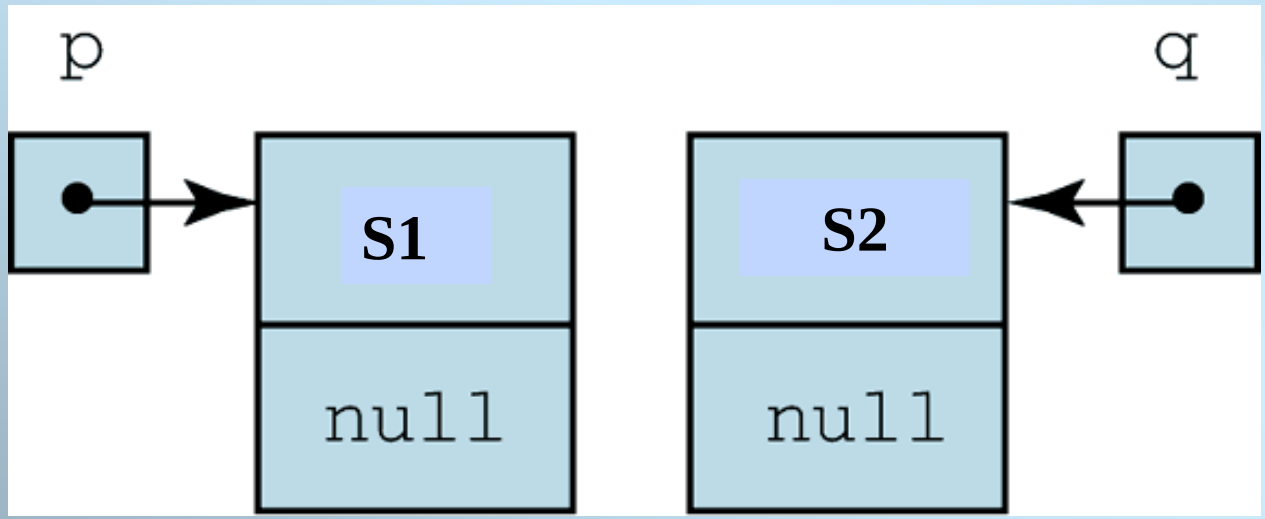
Student S1 = new Student("Ali", 42700000, 3.5)

Student S2 = new Student("Ahmed", 4271000, 3.9)

```
Node <Student> p = new Node <Student>(S1);
Node <Student> q = new Node(S2);
```

allocate storage for two objects of type **Node** referenced by **p** and **q**. The node referenced by **p** stores the object **S1**, and the node referenced by **q** stores the object **S2**. The **next** fields of both nodes are **null**.

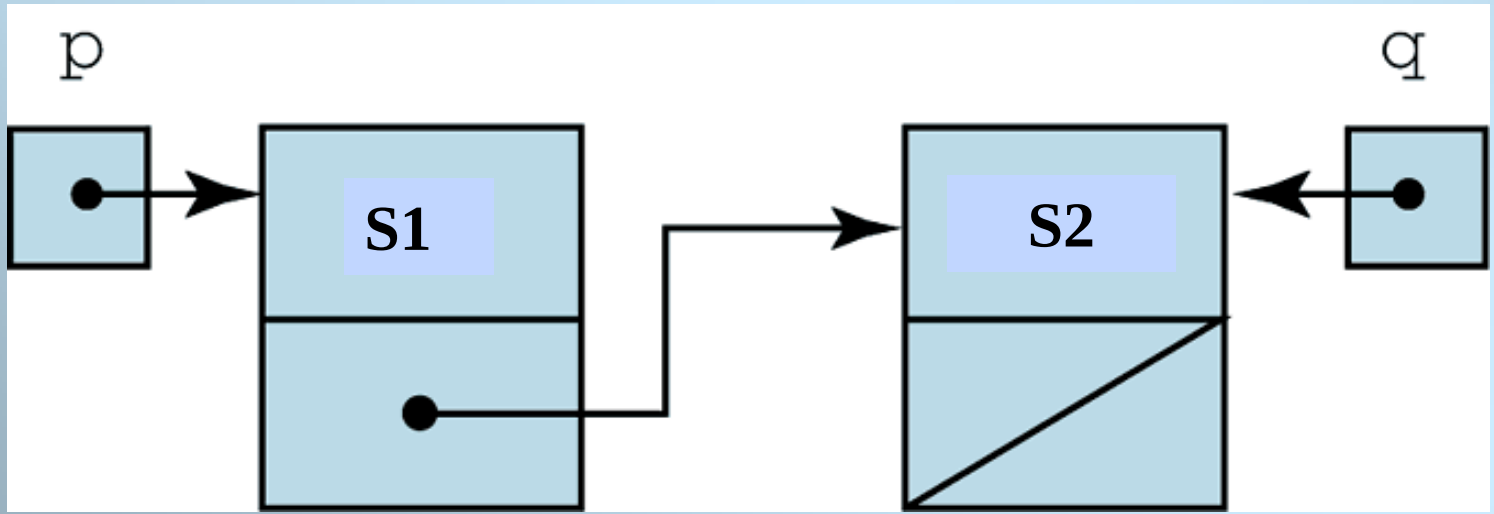
Nodes referenced by p and q



Connecting two nodes (Cont)

The statement
p.setNext (q);
stores the address of node **q** in the link field of node **p**, thereby connecting node **p** to node **q**, and forming a linked list with 2 nodes. The diagonal line in the **next** field of the second list node indicates the value **null**.

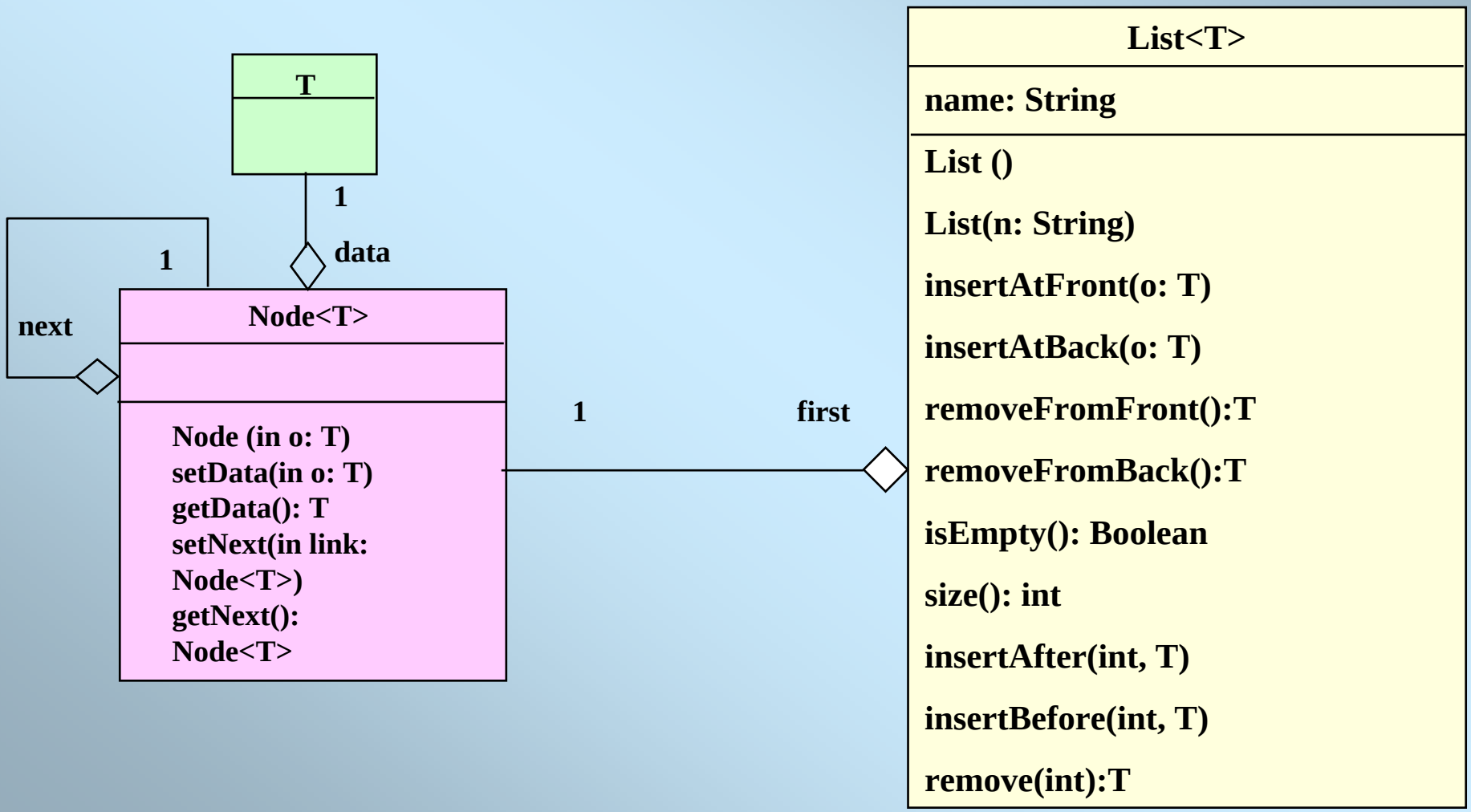
Linked list with two nodes



Linked Lists: Definition

- Linked list
  - Linear collection of nodes
    - A *linked list* is based on the concept of a **self-referential object** -- an object that refers to an object of the same class.
  - A program typically accesses a linked list via a reference to the first node in the list
    - A program accesses each subsequent node via the link reference stored in the previous node
  - Are dynamic
    - The length of a list can increase or decrease as necessary
    - Become full only when the system has insufficient memory to satisfy dynamic storage allocation requests

Generic Linked Lists



Implementation of the Generic Linked Lists

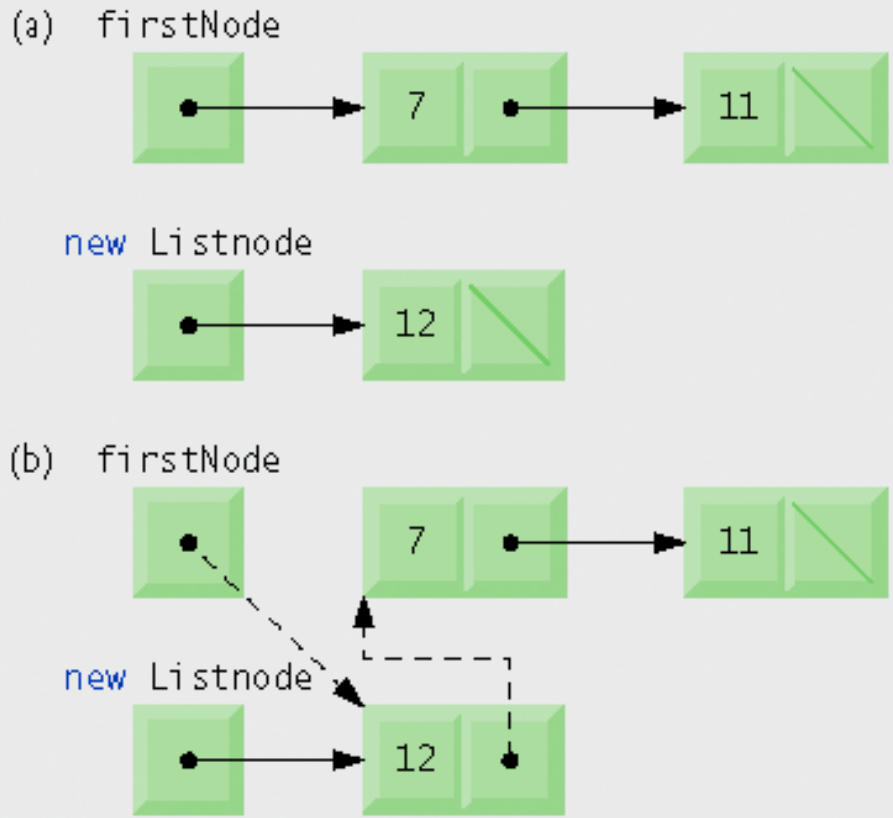
```
public class List<T>
{    private Node<T> first;
     private String name;

     public List()
    {  name=" ";
       first=null;
     }

     public List(String listName )
    {   name = listName;
        first = null;
    }
……
……
}
```

Linked List: insertAtFront

- **Method insertAtFront's steps**
  - Call isEmpty to determine whether the list is empty
  - If the list is empty, assign firstNode and lastNode to the new ListNode that was initialized with insertItem
    - The ListNode constructor call sets data to refer to the insertItem passed as an argument and sets reference nextNode to null
  - If the list is not empty, set firstNode to a new ListNode object and initialize that object with insertItem and firstNode
    - The ListNode constructor call sets data to refer to the insertItem passed as an argument and sets reference nextNode to the ListNode passed as argument, which previously was the first node

Linked List: insertAtFront (Cont)

Graphical representation of operation `insertAtFront`



Linked List: insertAtFront (Cont)

Code of `insertAtFront`

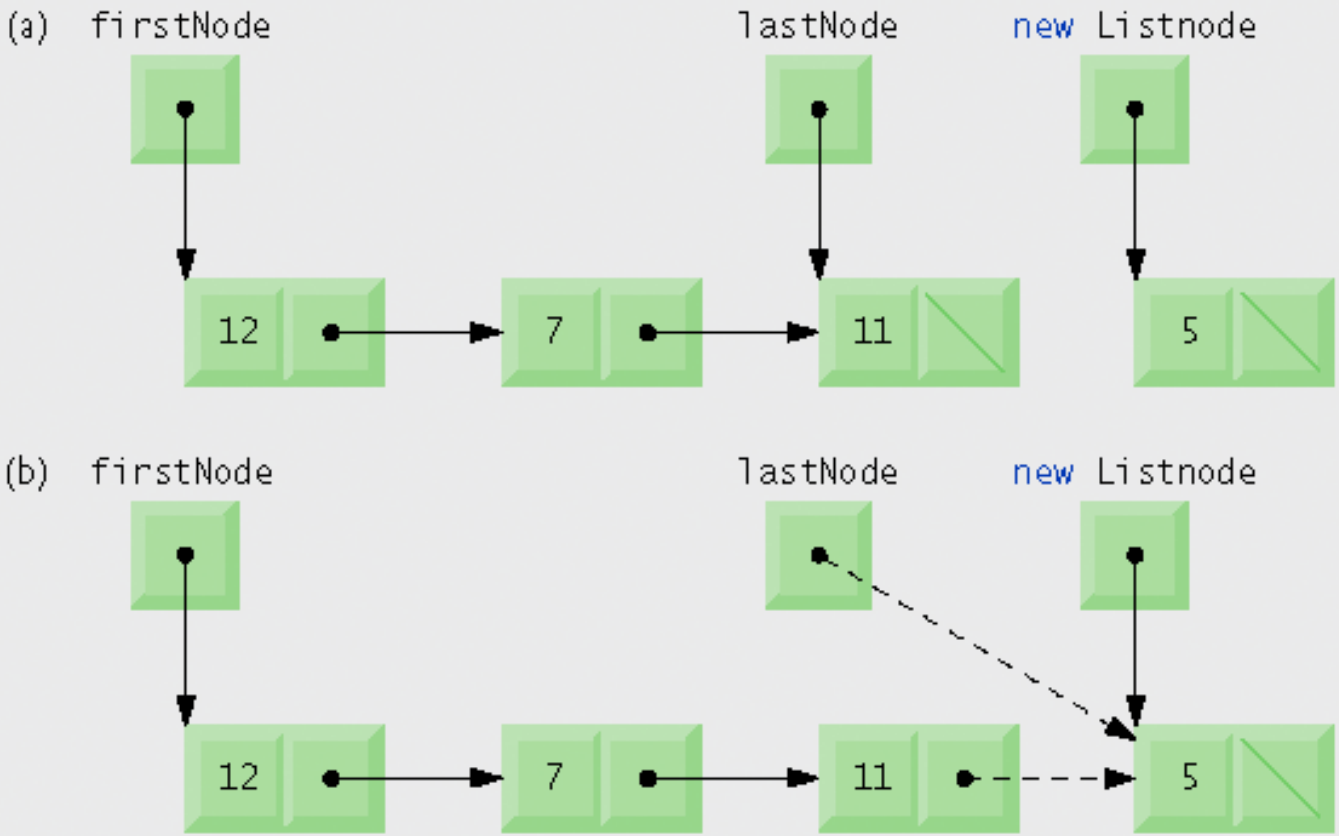
```
public void insertAtFront(T obj) {
   Node N =  new Node(obj);
   N.setNext(first);
   first = N;
} // insertAtFront()
```

Linked List: insertAtBack

- **Method insertAtBack's steps**
  - Call isEmpty to determine whether the list is empty
  - If the list is empty, assign firstNode and lastNode to the new ListNode that was initialized with insertItem
    - The ListNode constructor call sets data to refer to the insertItem passed as an argument and sets reference nextNode to null
  - If the list is not empty, assign to lastNode and lastNode.nextNode the reference to the new ListNode that was initialized with insertItem
    - The ListNode constructor sets data to refer to the insertItem passed as an argument and sets reference nextNode to null

Linked List: insertAtBack (Cont)

Graphical representation of operation insertAtBack.



Linked List: insertAtBack (Cont)

Solution 1

```
public void insertAtBack(T obj) {
    if (isEmpty())
        first = new Node(obj);
    else {
        Node current = first;                // Start at head of list
        while (current.getNext() != null)   // Find the end of the list
            current = current.getNext();
        current.setNext(new Node(obj));    // Insert the newObj
    }
} // insertAtRear
```

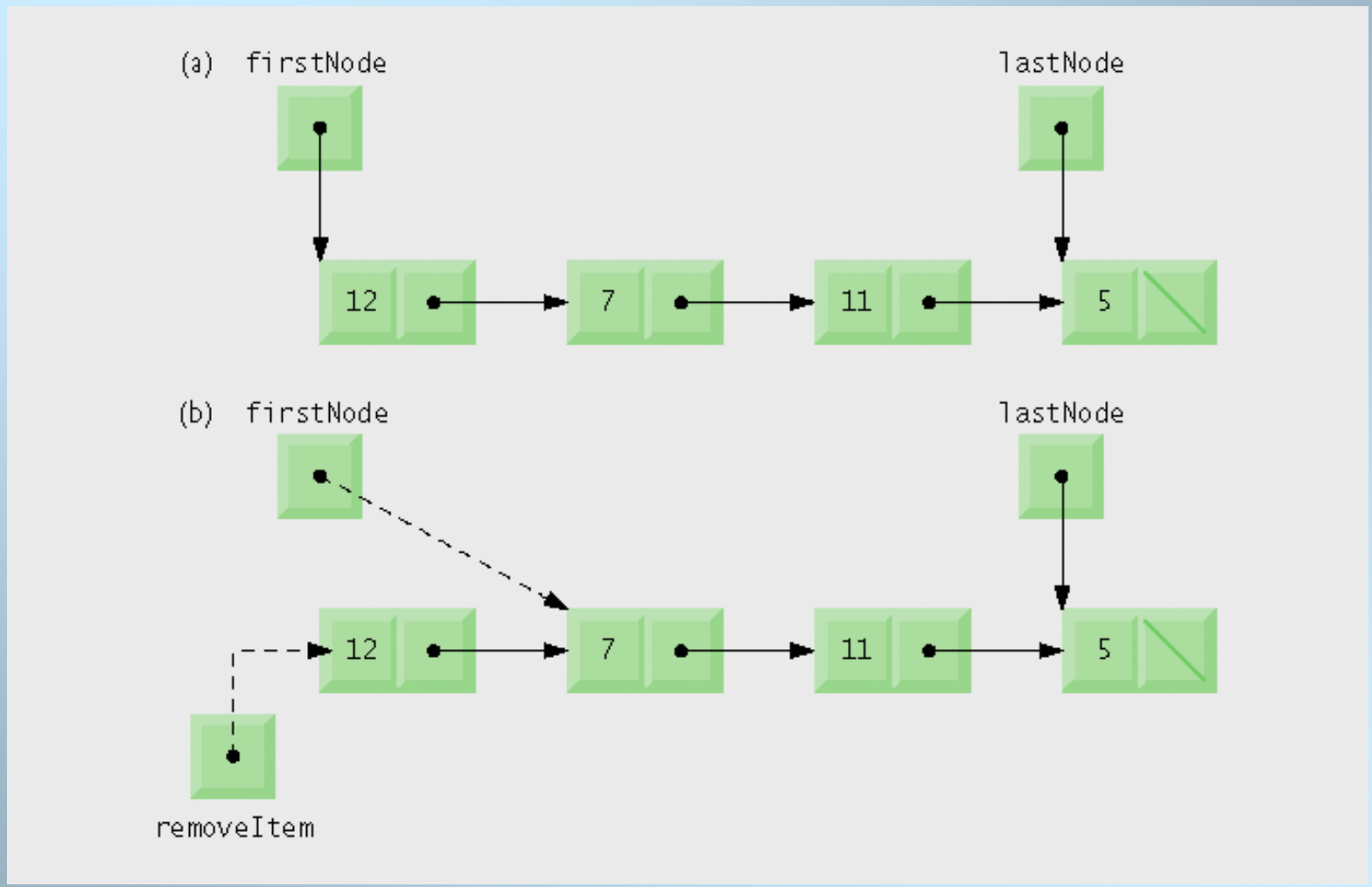
Solution 2

```
public void insertAtBack(T obj) {
    Node N =new Node(obj);
    if (isEmpty())
        first = N;
    else {
        Node current = first;                // Start at head of list
        while (current.getNext() != null)   // Find the end of the list
            current = current.getNext();
        current.setNext(N));    // Insert the newObj
    }
} // insertAtRear
```

Linked List: removeFromFront

- **Method removeFromFront's steps**
  - Throw an EmptyListException if the list is empty
  - Assign firstNode.data to reference removedItem
  - If firstNode and lastNode refer to the same object, set firstNode and lastNode to null
  - If the list has more than one node, assign the value of firstNode.nextNode to firstNode
  - Return the removedItem reference

Linked List: removeFromFront



Graphical representation of operation removeFromFront.

Linked List: removeFromFront

Code of removeFromFront

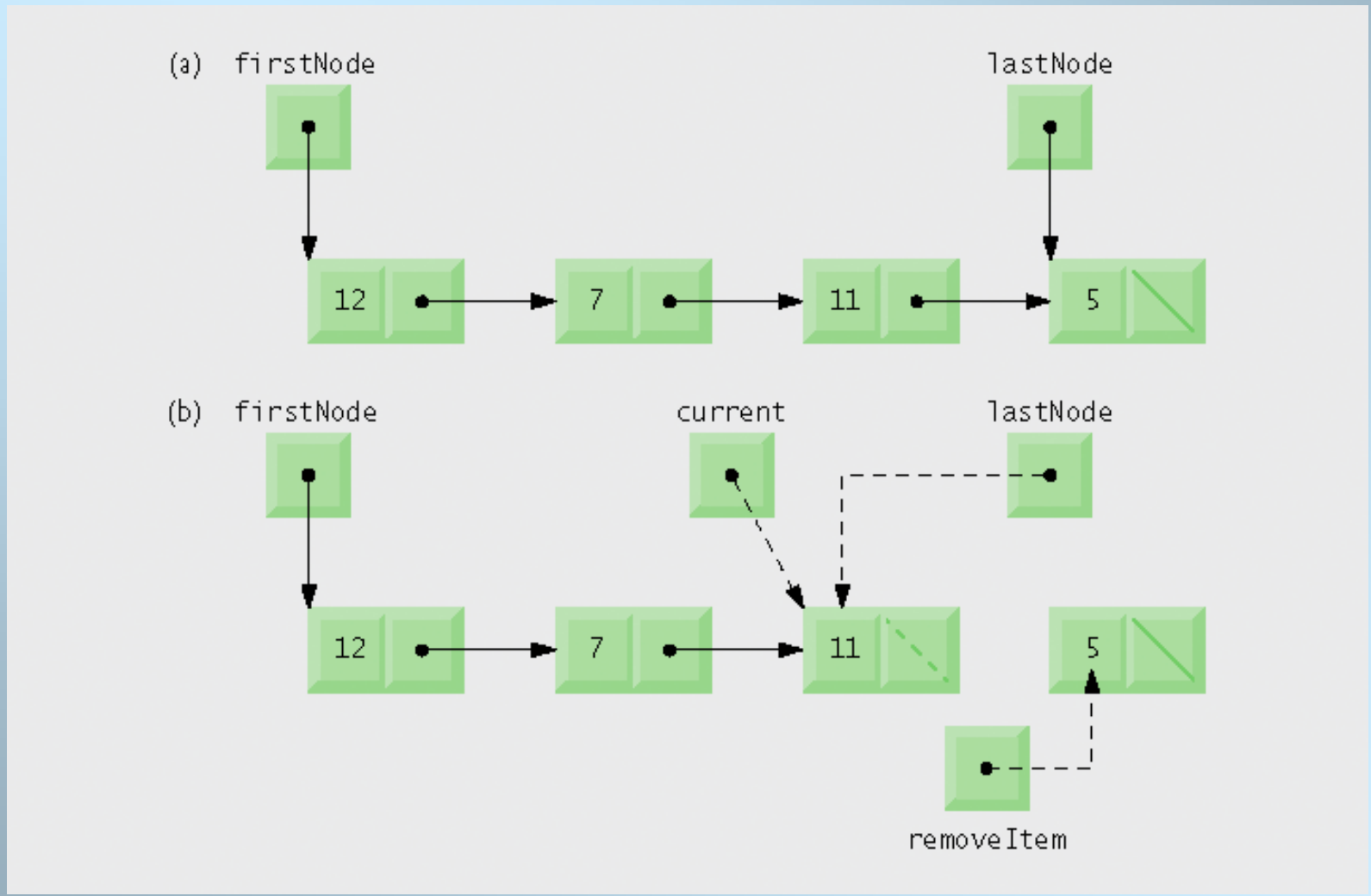
```
public T removeFromFrontt() {
    if (isEmpty())
        return null;
    Node N = first; // T d = first.getdata();
    first = first.getNext();
    return N.getData(); // return d;
}
```

Linked List: removeFromBack

- **Method removeFromBack's steps**
  - Throws an EmptyListException if the list is empty
  - Assign lastNode.data to removedItem
  - If the firstNode and lastNode refer to the same object, set firstNode and lastNode to null
  - If the list has more than one node, create the ListNode reference current and assign it firstNode
  - "Walk the list" with current until it references the node before the last node
    - The while loop assigns current.nextNode to current as long as current.nextNode is not lastNode

Linked List: removeFromBack

Graphical representation of operation removeFromBack.

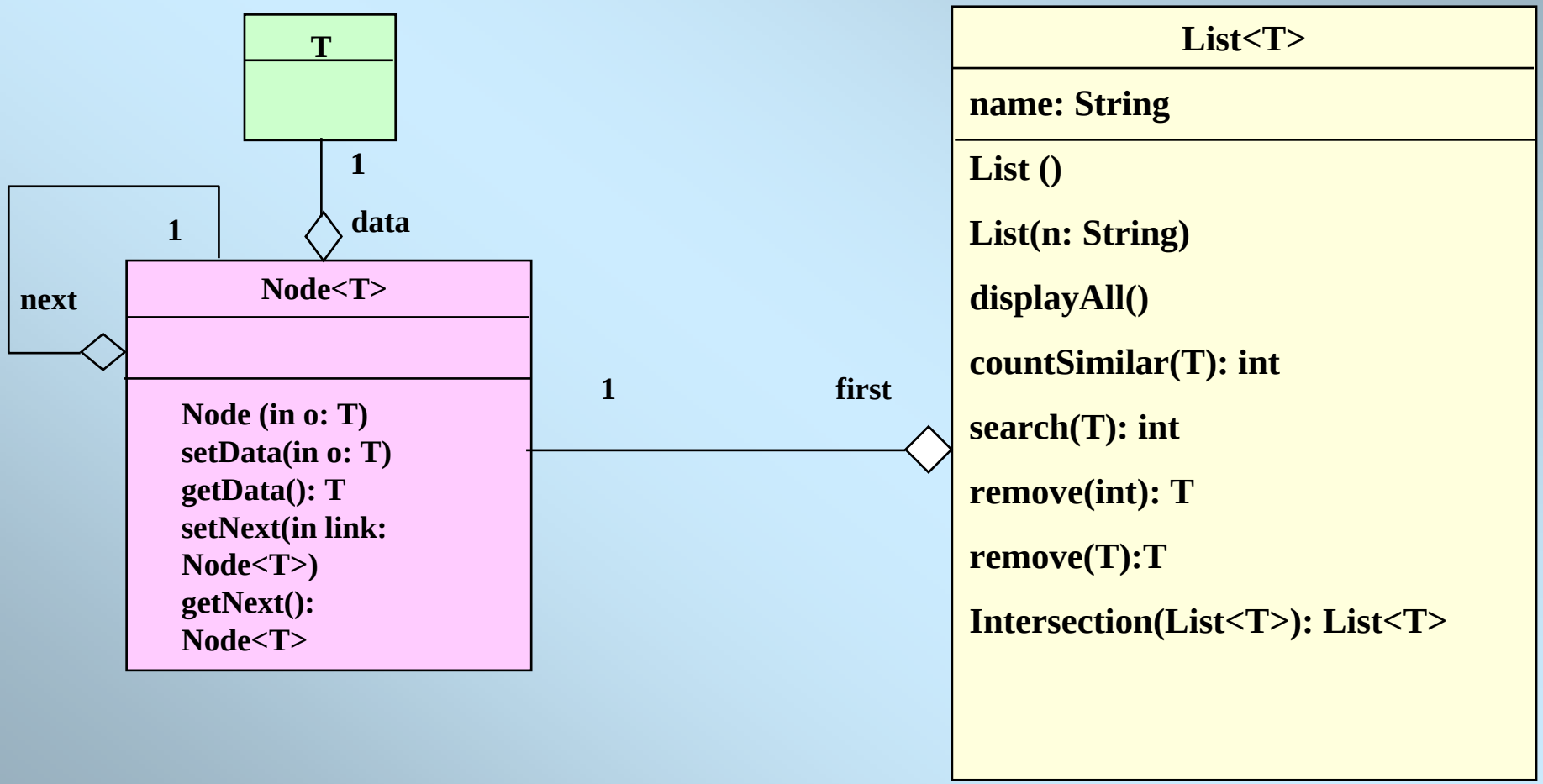


Linked List: removeFromBack

Code of removeFromBack

```
public T removeFromBack() {
    if (isEmpty())  // Empty list
        return null;

    Node current = first;
    if (current.getNext() == null) { // Singleton list
        first = null;
        return current.getData();
    }

    Node previous = null;            // All other cases
    while (current.getNext() != null) {
        previous = current;
        current = current.getNext();
    }
    previous.setNext(null);
    return current.getData();
} // removeLast()
```

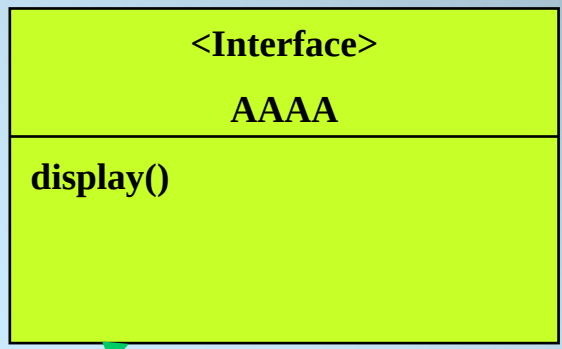
Generic Linked Lists and Interface



Generic Linked Lists and Interface

Code of displayAll()

```
public void displayAll() {
    if (!(isEmpty()))
    {
        Node current = first;
        while (current!= null)
        {
           current.getData().display();
           current=current.getNext();
        }
     }
}
```

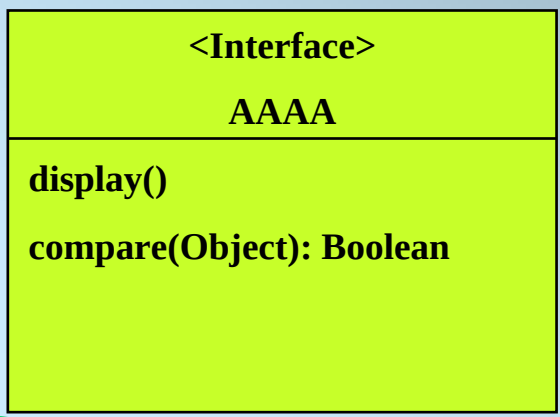


This method must be implemented by each substitute class of the generic type.
We add an Interface to our package containing this method "display" and each substitution's class of the generic type must implement the Interface.

Generic Linked Lists and Interface

Code of countSimilar(T):int

```
public int count(T t) {
    Node current = first;
    int x=0;
    while (current!= null)
    {
       if (current.getData().compare(t))
            x++;
       current=current.getNext();
     }
    return x;
}
```

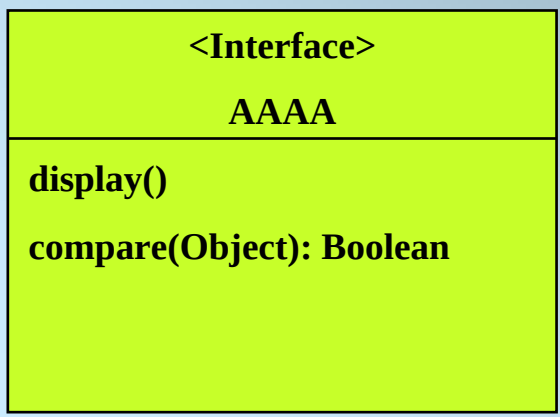


This method must be implemented by each substitute class of the generic type.
We add an Interface to our package containing this method "compare" and each substitution's class of the generic type must implement the Interface.

Generic Linked Lists and Interface

Code of search(T):int

```
public int search(T t) {
    Node current = first;
    int x=0;
    while (current!= null)
    {
       x++;
       if (current.getData().compare(t))
         return x;
       current=current.getNext();
     }
    return -1;
}
```



This method must be implemented by each substitute class of the generic type.
We add an Interface to our package containing this method "compare" and each substitution's class of the generic type must implement the Interface.

Generic Linked Lists and Interface

Code of remove(int):T

```
public T remove (int x) {
    if ((isEmpty()) || (x<0) || (x>size())
        return null;
    if (x==1) return removeFromFront();
    if (x==size()) return removeFromBack();
    Node current = first;
    for(int i=1, i<x-1 ; i++)
       current = current.getNext();
    T d=current.getNext().getdata();
    current.setNext(current.getNext().getNext());
    return d;
}
```

Generic Linked Lists and Interface

Code of remove(T):T

```
public int remove(T t) {
    int x = search(t);
    return remove(x);
}
```