COP-3530
Data Structures

Instructor: Dr. Antonio Hernandez

Text: Data Structures and Algorithm Analysis in Java, 3rd Edition

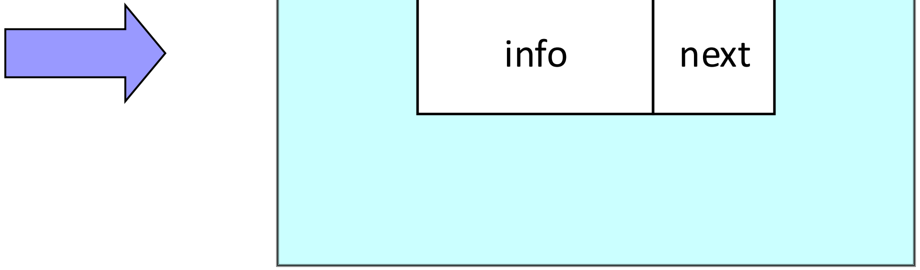
Abstract Data Types

13. Linked lists

Linked Lists

Definition: a list of items, called **nodes**, where each node has two fields: one containing the information (*info)*, and one that is a reference to the next node in the list (*next* or *link)*.

Structure of a node



Linked Lists

In linked lists:

1) A special variable that stores a reference to the first node is used to provide access to the list nodes. We typically call this variable the *first* or the *header* of the linked list.
2) Sometimes, it is a good idea not to have the *first* pointing to the first node, but rather to a *dummy* or *sentinel* node that in turn points to the first node.

(The rationale for this is that the implementation of some operations will be easier -some special cases can be avoided)

Linked Lists

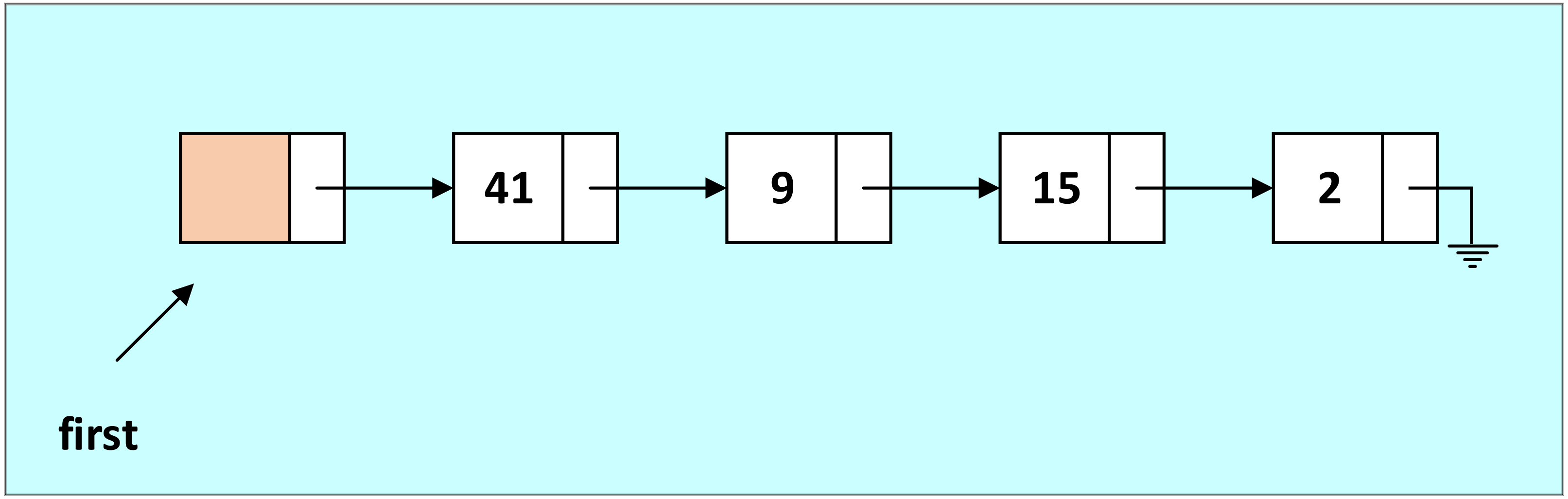
<u>In linked lists</u> (cont.):

3) It is convenient to represent a linked list as a box with two sections, where

- the first section is used for the element it holds
- the second section is used for the reference to the next node.
- an ***arrow*** is used to indicate the next node for a given node.
- a symbol that is similar to the ground symbol used in electronic diagrams is used to symbolize the value ***null***.

Linked Lists

Example of linked list with *first* pointing to a dummy node:



☼ A value of *null* is needed in the *next* field of the last node to indicate that no other elements follow.

Linked List: implementing the Node class

```
//Node class

public class Node
{
       private int info;
       private Node next;

       public Node()
       {
              info = 0;
              next = null;
       }

       public void setInfo(int i)
       {
              info = i;
       }
```

Linked List: the Node class (cont.)

```
       public void setNext(Node L)
       {
             next = L;
       }

       public int getInfo()
       {
             return info;
       }

       public Node getNext()
       {
             return next;
       }
}
```

A basic **LinkedList** class

```
//Linked list class

public class LinkedList
{
	private Node first;

	public LinkedList(){...}

	public boolean isEmpty(){...}

	public void display(){...}

	public boolean search(int x){...}

	public void insert(int i){...}

	public void remove(int x){...}
}
```

Linked List: implementation

The constructor simply creates an empty list:

```
public LinkedList()
{
        first = new Node();
}
```

Testing to determine if a list is empty is done by finding the value of **first.next**:

```
public boolean isEmpty()
{
        return (first.getNext() == null);
}
```

Linked Lists: traversing

To traverse a linked list (for example, to display its elements):

1) get the reference to the first node (**`current`**)
2) while the value of *current* is not *null*
    - process the node referenced by *current*
    - make *current* point to the next node by assigning *current.next* to *current*

Note: the pattern used to traverse a linked list can be used in many different operations

Linked List: implementation

To *display* the elements in a linked list:

```
public void display()
{
       //get the reference to first
       Node current = first.getNext();

       while (current != null)
       {
              //process the current item
              System.out.print(current.getInfo() + " ");

              //advance current
              current = current.getNext();
       }

       System.out.println();
}
```

Linked List: implementation

The logic behind *search*: traverse the list while comparing each element in the list with *x* :

```
public boolean search(int x)
{
        Node current = first.getNext();

        while(current != null)
        {
                if (current.getInfo() == x) return true;
                current = current.getNext();
        }

        return false;
}
```

Linked List: implementation

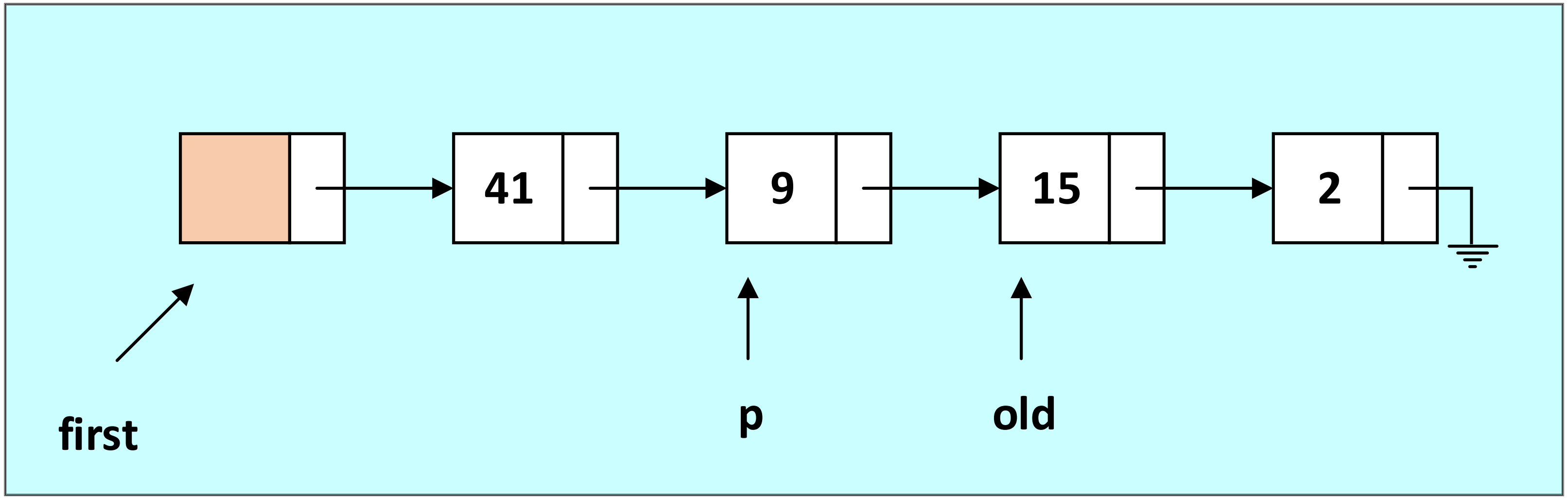
The method *insert* inserts a new node before the node referenced by *first*. :

```
public void insert(int x)
{
        Node newListNode = new Node();

        newListNode.setInfo(x);
        newListNode.setNext(first.getNext());

        first.setNext(newListNode);
}
```

Note: If we need to insert a value before a node *n* that is not the first in the list, a similar approach is followed.

Linked List: implementation

remove an element: find a reference (*p*) to the node before the one to be deleted (*old*)

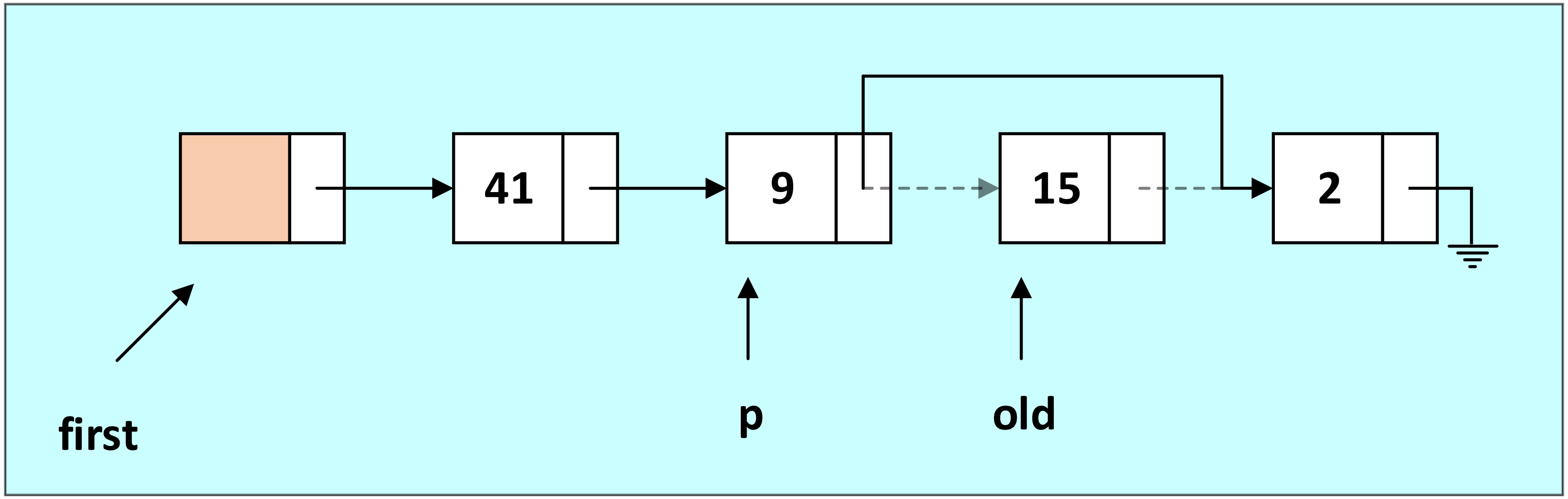


remove(15)

Linked List: implementation

remove an element:

```
p.setNext(old.getNext());
```



remove(15)

Linked List: implementation

remove an element:

```
public void remove(int x)
{
        Node old = first.getNext(),
                p = first;

        //Finding the reference to the node before the
        //one to be deleted
        boolean found = false;
        while (old != null && !found)
        {
                if (old.getInfo() == x) found = true;
                else
                {
                        p = old;
                        old = p.getNext();
                }
        }
```

Linked List: implementation

remove an element (cont.):

```
              //if x is in the list remove it.
              if (found) p.setNext(old.getNext());

       }
```

Linked List: implementation

Testing the class LinkedList:

```
public class Main
{
       public static void main(String args[])
       {
              LinkedList intList = new LinkedList();

              System.out.print("List of numbers before
                                 list creation: ");
              for (int i =0; i < 10; i++)
              {
                     int info = (int)(Math.random()*10);
                     System.out.print(info + " ");

                     intList.insert(info);
              }
```

Linked List: implementation

Testing the class LinkedList (cont.):

```
              System.out.print("\nList of numbers after
                                 creation:  ");

              intList.display();
       }
}
```

PRACTICE

Program 13_01:

Create a project with the code given. Write a Prog13_01 class to test it.

PRACTICE

Program 13_02:

Implement the methods

- public void insert (int x, int loc);
- public void removeItemAt (int loc);

Assume that list item locations are as in an array: 0, 1, 2, … . Check that **loc** is a valid value.

public void insert (int x, int loc)

```
public void insert(int x, int loc) {

       if (loc >= length())
                   System.out.println("Incorrect location!");
       else
       {
           Node current = first;
           for(int i=0; i<loc; i++)
           {
               current = current.getNext();
           }

           Node p = new Node();
           p.setInfo(x);
           p.setNext(current.getNext());
           current.setNext(p);
       }
   }
```

public void removeItemAt (int loc)

```
public void removeItemAt(int loc) {

       if (loc >= length())
                   System.out.println("Incorrect location!");
       else
       {
           Node current = first;
           for(int i=0; i<loc; i++)
           {
               current = current.getNext();
           }

           current.setNext(current.getNext().getNext());
       }
   }
```

Abstract Data Types

14. Variations of linked lists

Variations

Variations of linked lists:

- Ordered (or sorted) linked lists
- Doubly linked list
- Lists with header/trailer nodes
- Circular linked lists

Variations: ordered linked list

- In an *ordered linked list* the elements are sorted
- Because the list is ordered, we need to modify the algorithms (from how they were implemented for the regular linked list) for the *search* and *insert* operations (*remove* operation remains basically the same)

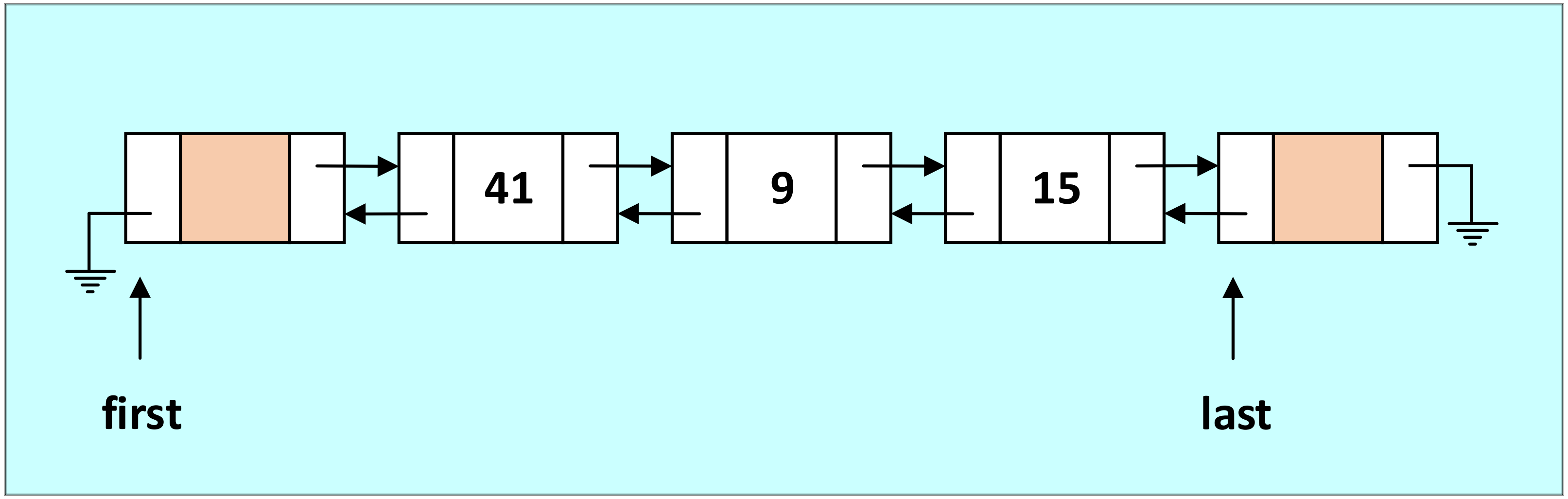
Variations: doubly linked list

- A doubly linked list is a linked list in which every node has a *next pointer* and a *back pointer*

- Every node (except the last node) contains the address of the next node, and every node (except the first node) contains the address of the previous node.

- A doubly linked list can be traversed in either direction

Variations: doubly linked list



Variations: *Header* and *Trailer* Nodes

- You can *set a header node* at the beginning of the list

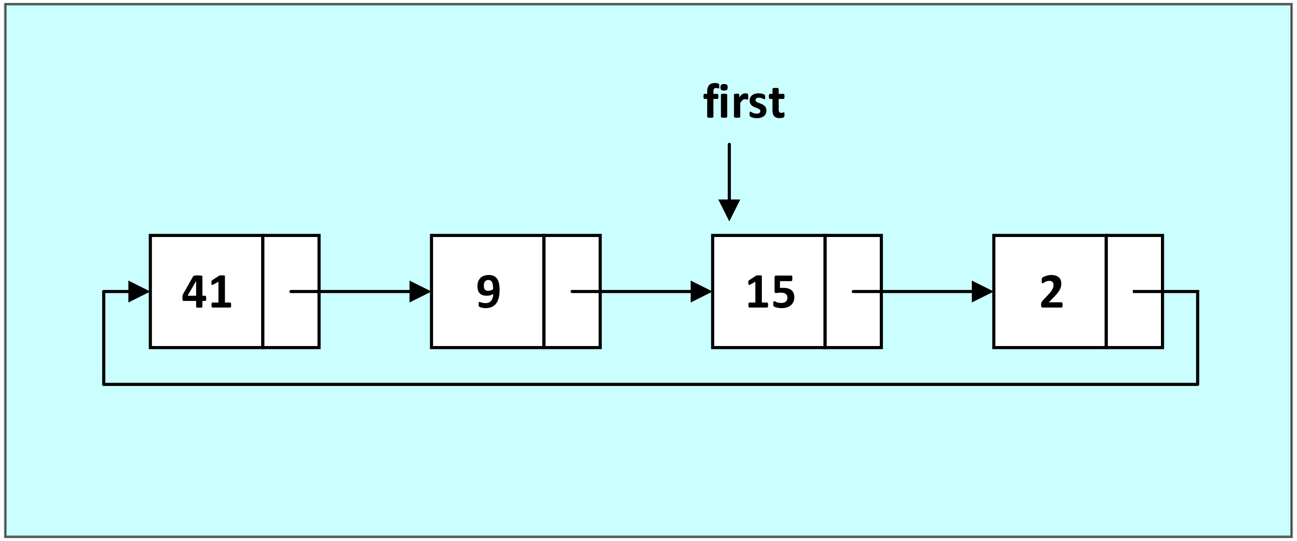
- You can *set a trailer node* at the end of the list.

- These two nodes, header and trailer, serve merely to simplify the insertion and deletion algorithms and are not part of the actual list.

Variations: *circular* linked list

- A linked list in which the last node points to the first node is called a *circular linked list*
- In a circular linked list with more than one node, it is convenient to make the pointer **first** point to the node before the last.

Variations: *circular* linked list



PRACTICE

Program 14_01:

Modify the class **LinkedList** in Exercise 13_01 to make it a doubly linked list:

- Name your class **DoublyLinkedList** and create a tester class **Main.**

- Use dummy header and trailer nodes.

- Add a **printInReverse** method.

- Add an **append** method to add an item at the end of the list.