BACS2063 Data Structures and Algorithms

Linked Implementations of Collection ADTs

Chapter 5

Chapter 5 Notes
(Updated for 2023)

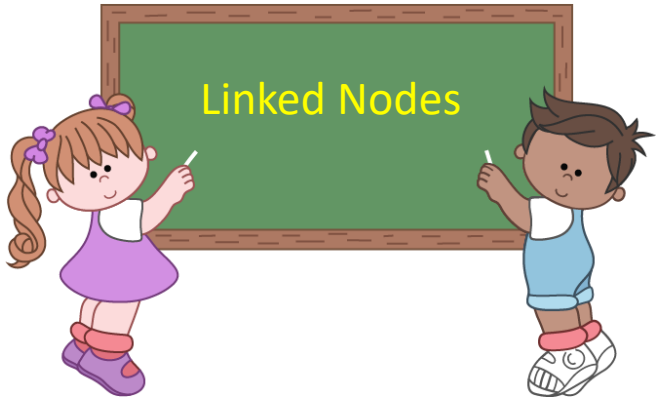
- Analogy of desks removed. All other slides remained.
- Some slides are rearranged (location moved).
- Additional slides are added to highlight important concepts.
- Additional slides for code (e.g. Linked Stack and Linked Queue).
- Past year questions added at the end of the chapter.

Learning Outcomes

At the end of this chapter, you should be able to

- Describe what are linked structures.
- **Implement** the ADT list, stack and queue using linked implementation.
- Evaluate the advantages and disadvantages of a linked implementation of linear collections.
- Describe and implement variations of linked structures.

Chapter 5 Summary



Topics covered:

- What is a node?
- How to write the code for a Node class?
- Basics about node activities
  - Traversing a chain of nodes
  - Adding a node
  - Removing a node
- Linked List
- Linked Stack
- Linked Queue
- Variations: Tail Reference, Circular chain, doubly-linked

Why do we need another way of storing data in a program?

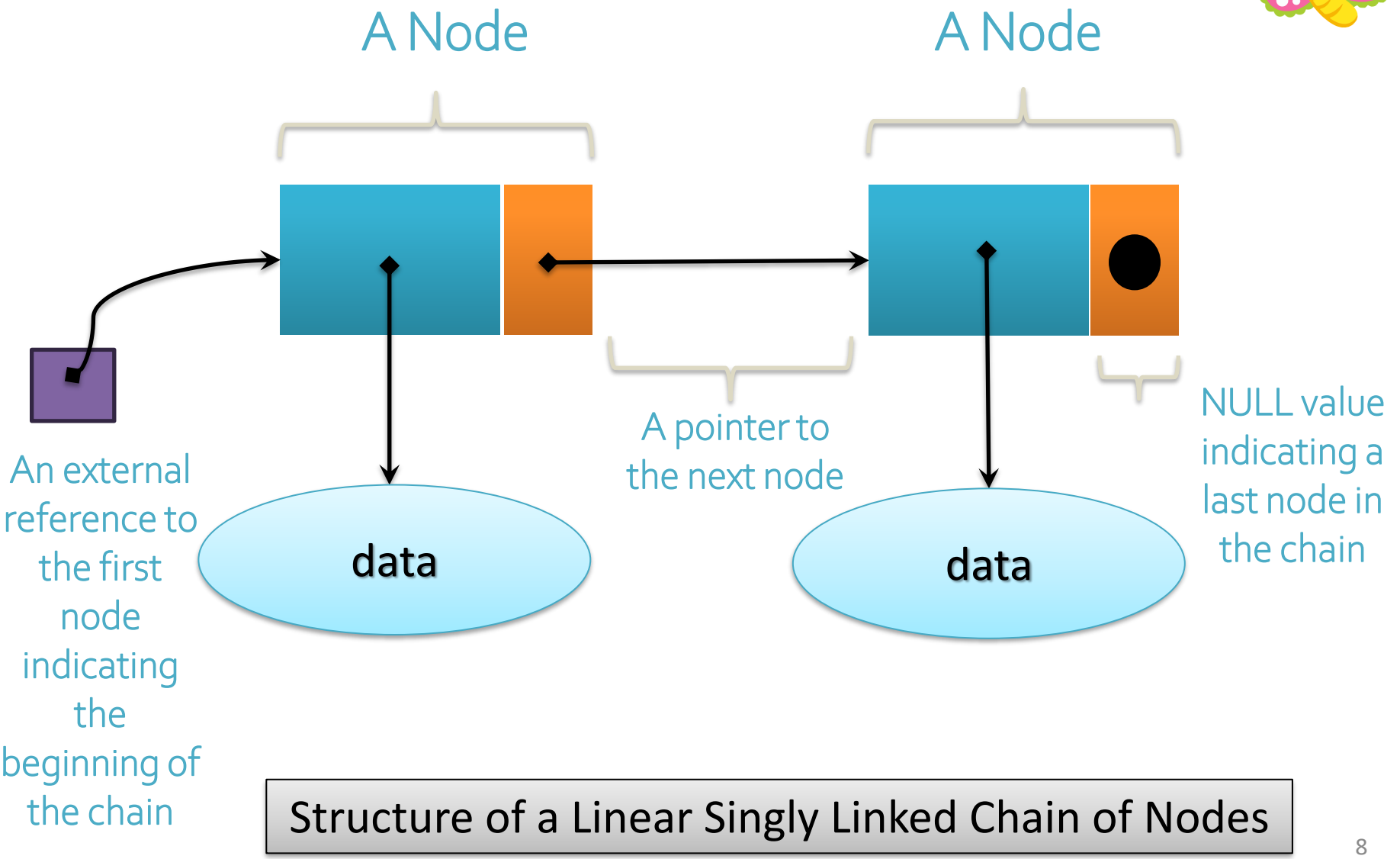
Limitations of using arrays

1. Overhead due to shifting during ***add*** and ***remove*** operation.
2. Inflexibility due to fixed array size
   - If array size insufficient - to "expand" array dynamically, a new array has to be allocated and the contents of the old array copied to the new array.
   - If array size too big - space wastage as only a small part of the array is utilized.

Is it possible to have a **flexible** data structure?

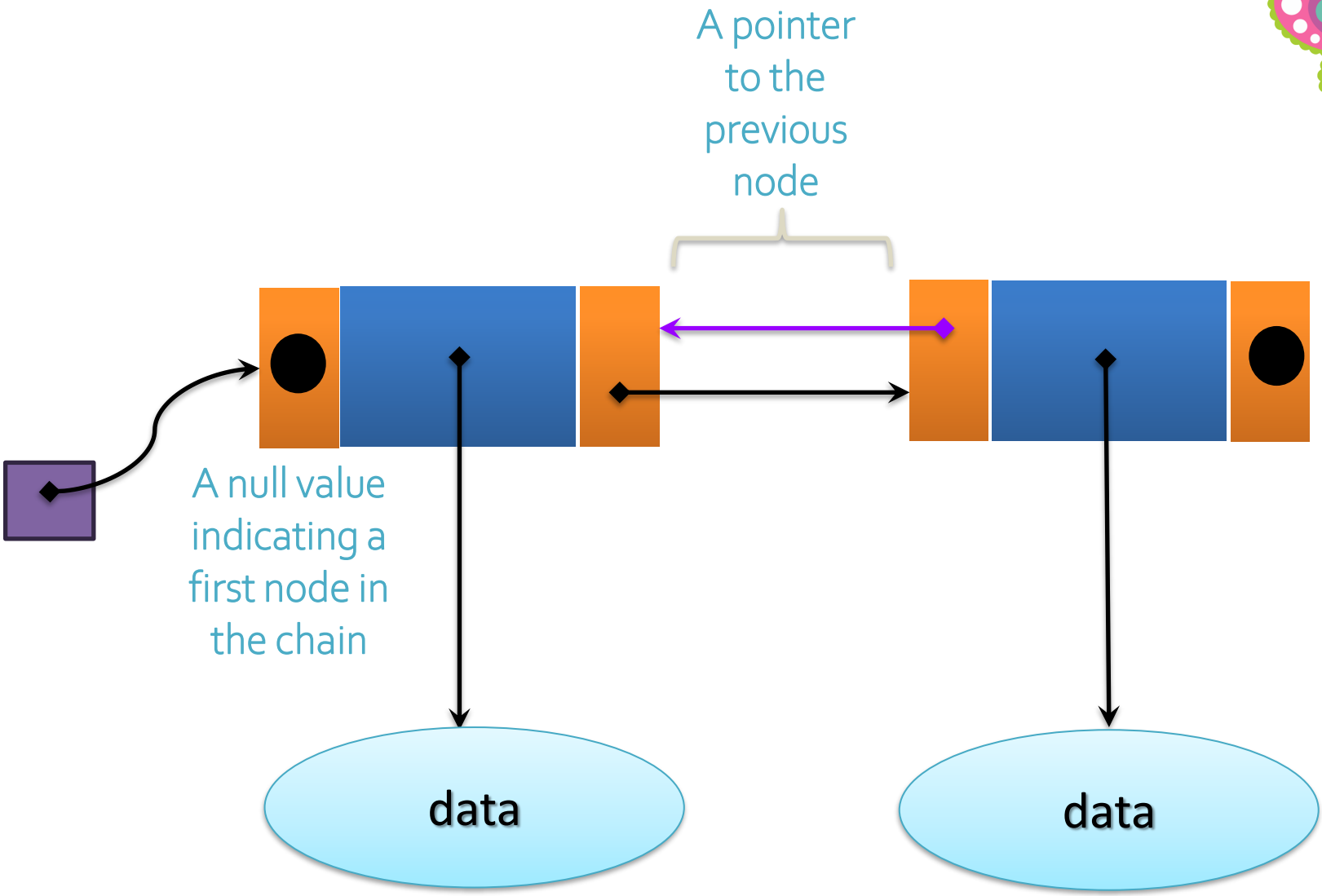
- *Allocate* space (create the object) for each entry at the point when we actually add the entry.
- *Deallocate* space when an entry is removed.
- For add and remove - instead of shifting, just adjust links between the objects.
  - The object would need 2 parts: one for the data and the other for the link to the next object.

What is a node?



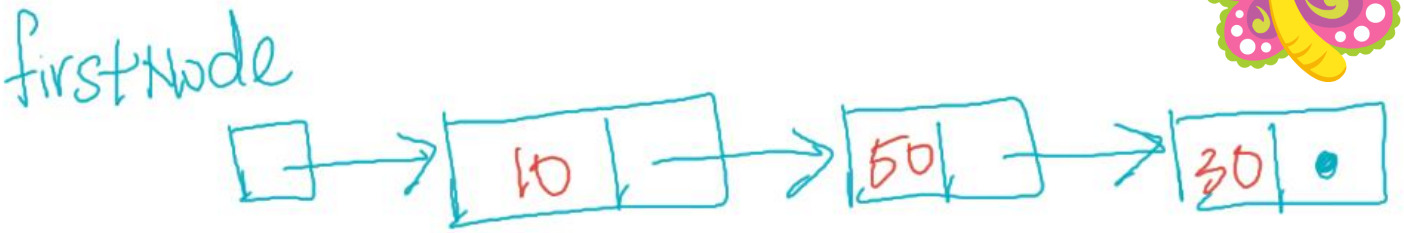
Structure of a Linear Singly Linked Chain of Nodes

What is a node?

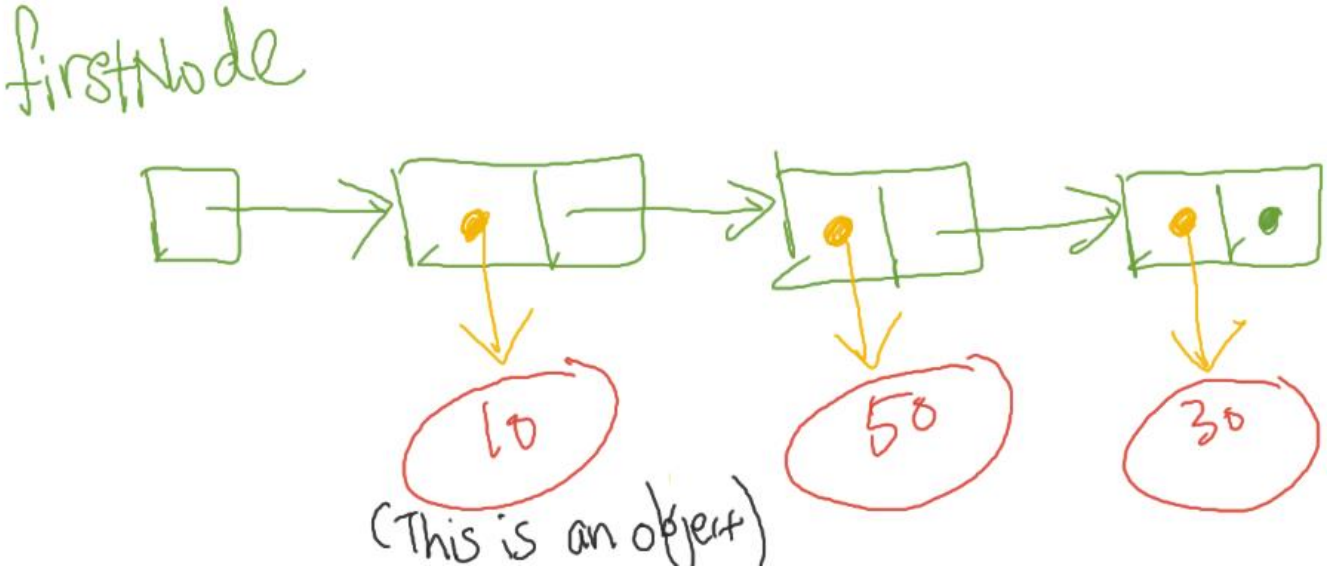


Structure of a Linear Doubly Linked Chain of Nodes

Reminder: WHAT I DRAW



HOW IT ACTUALLY IS



Value Types and Reference Types

- Types are categorized as either Value Types or Reference Types, each with distinct memory allocation behaviors.

- Value Types encapsulate and store data directly within their own memory allocation, while Reference Types contain a pointer that refers to another memory location where the actual data is stored.

- It is important to note that Reference Type variables are stored in the *heap*, a region of memory used for dynamic memory allocation, while Value Type variables are stored in the *stack*, a region of memory used for local variable storage and function call operations.

Examples of Value Type and Reference Type

- Value Type

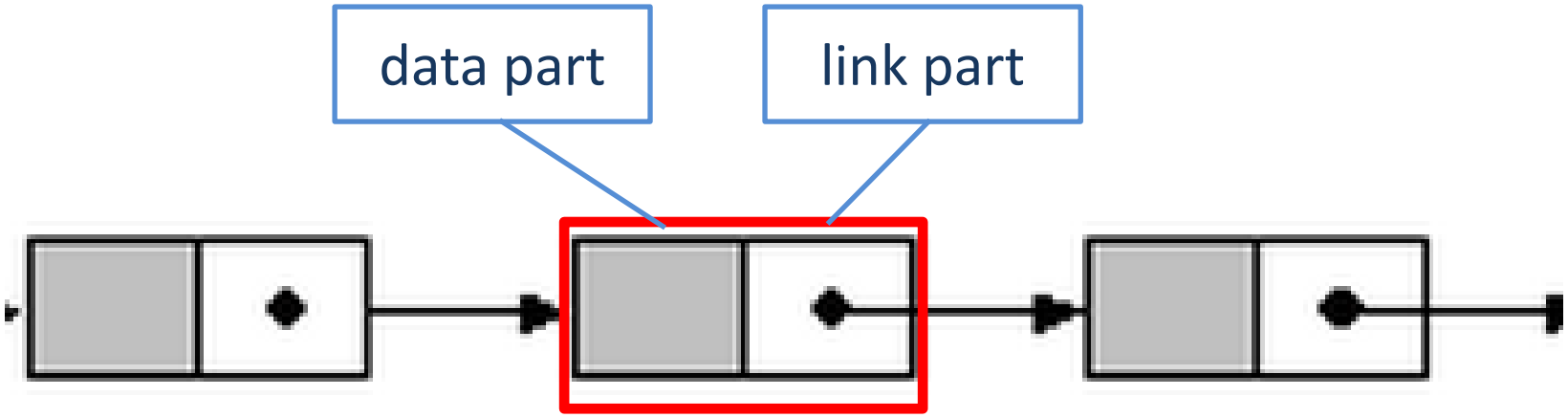
```
int num1 = 55;
```

- Reference Type

```
Customer cust1 = new
    Customer("C111", "Amy")
```

Nodes

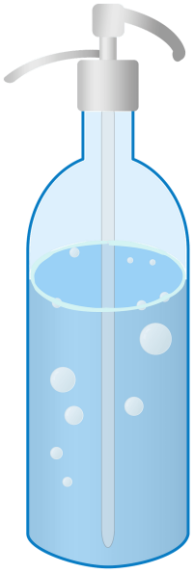
- Nodes are objects that are linked together to form a *data structure**
- A node comprises of two parts:
  - A data part and
  - A link part (contains the address of the next node)



a node

*What is a Data Structure?

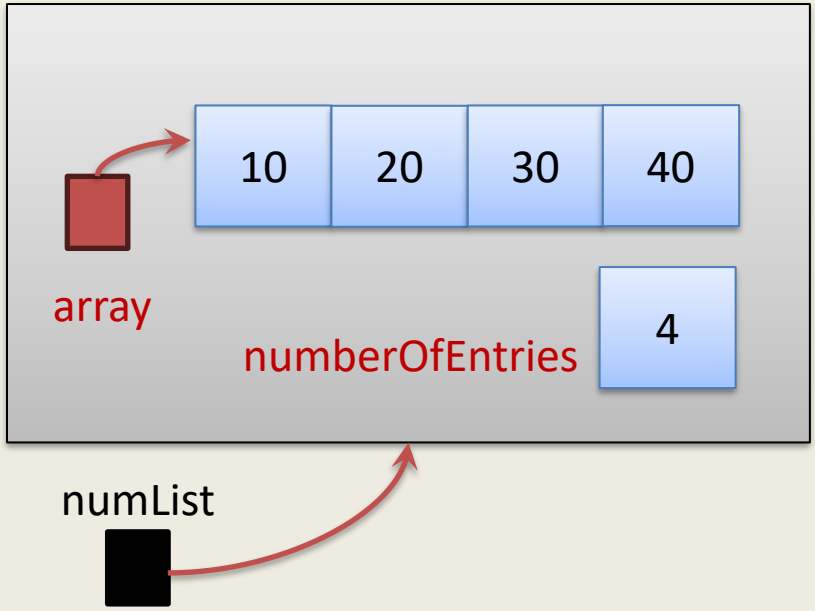
- Metaphorically speaking, a data **structure** is the bottle and the **data** is the content of the bottle.
- So, a data structure can be imagined as the **shape and structure of the bottle holding the content**.



Array List vs Linked List

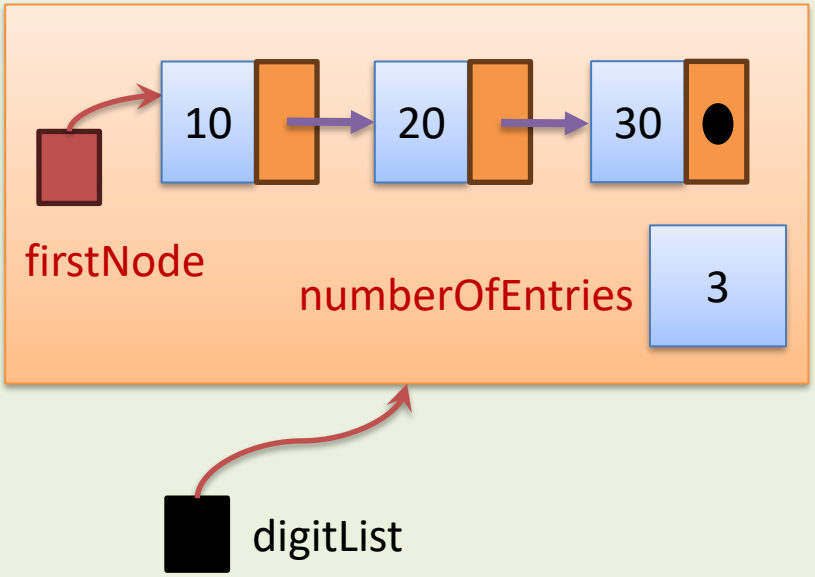
Array implementation:

ArrayList<Integer> numList =
new ArrayList<>();



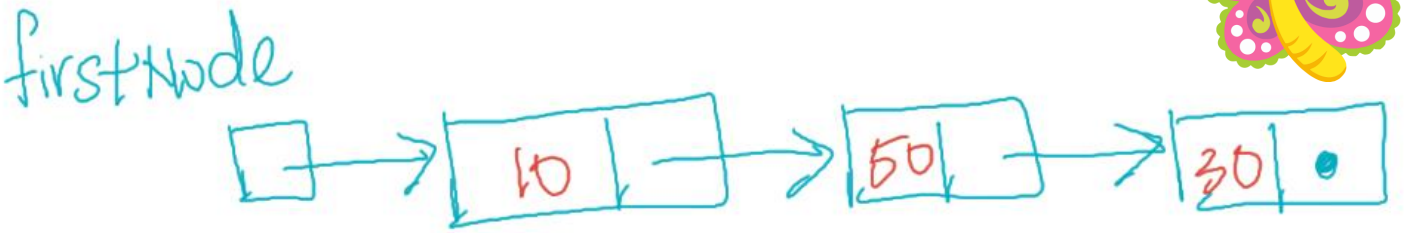
Linked Nodes Implementation

LinkedList<Integer> digitList = new LinkedList<>();

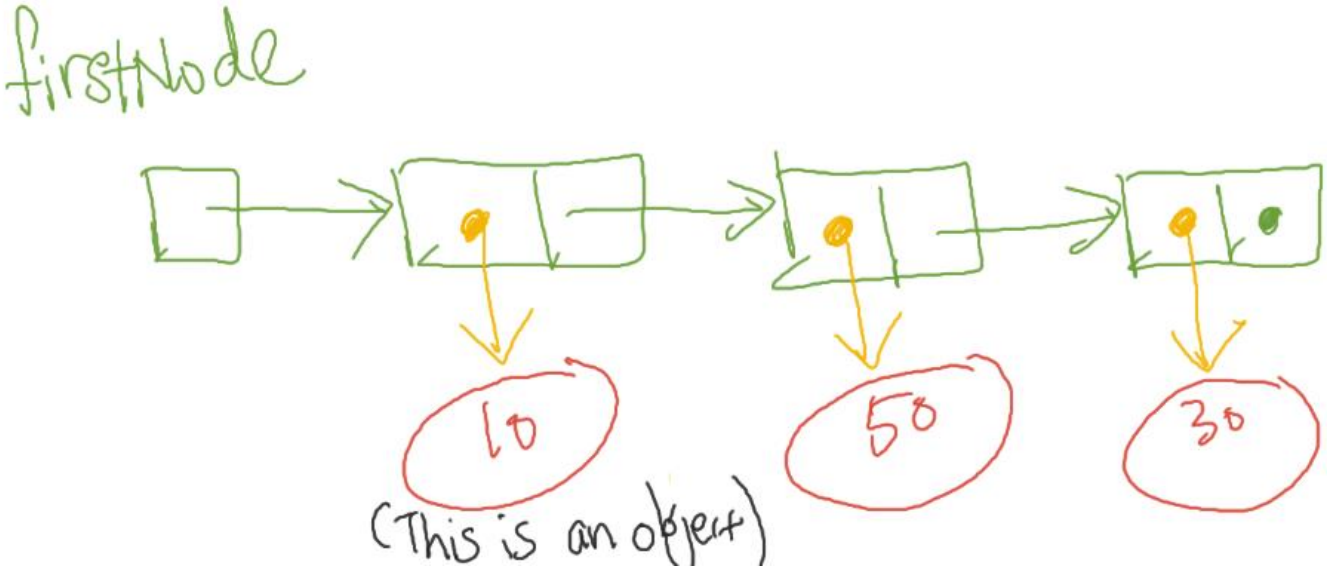


REMINDER!!!

Reminder: WHAT I DRAW



HOW IT ACTUALLY IS



The Class **Node**

- Two data fields:
  - **`data`**: A reference to an entry in the list
  - **`next`**: A reference to another node
- Refer to `Chapter5\samplecode\`**`Node.java`**

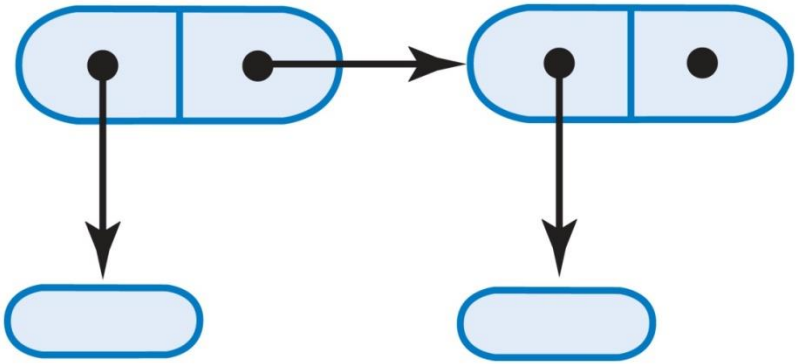


Fig. 12 Two linked nodes that each reference object data

The Node Class in Java

```
private class Node {

  private T data;
  private Node next;

  private Node(T data) {
    this.data = data;
    this.next = null;
  }

  private Node(T data, Node next) {
    this.data = data;
    this.next = next;
  }
}
```

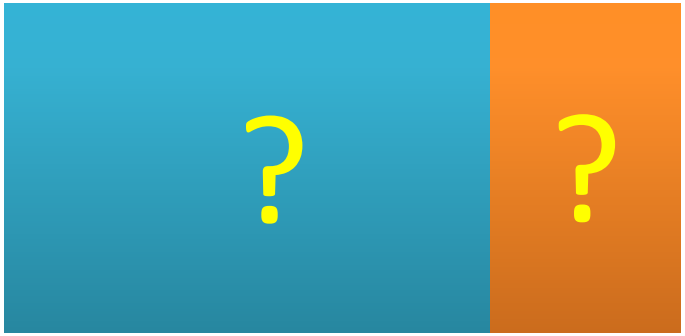
The class Node as an inner class

- ***Node*** is an implementation detail of the ADT list that should be *hidden* from the list's client.
- Therefore, we will define **Node** as an *inner class i.e.*,
  - as a **private** class within the class that implements the list.
  - the data fields of an inner class are accessible directly by the enclosing class without the need for accessor and mutator methods.

The class Node as an inner class

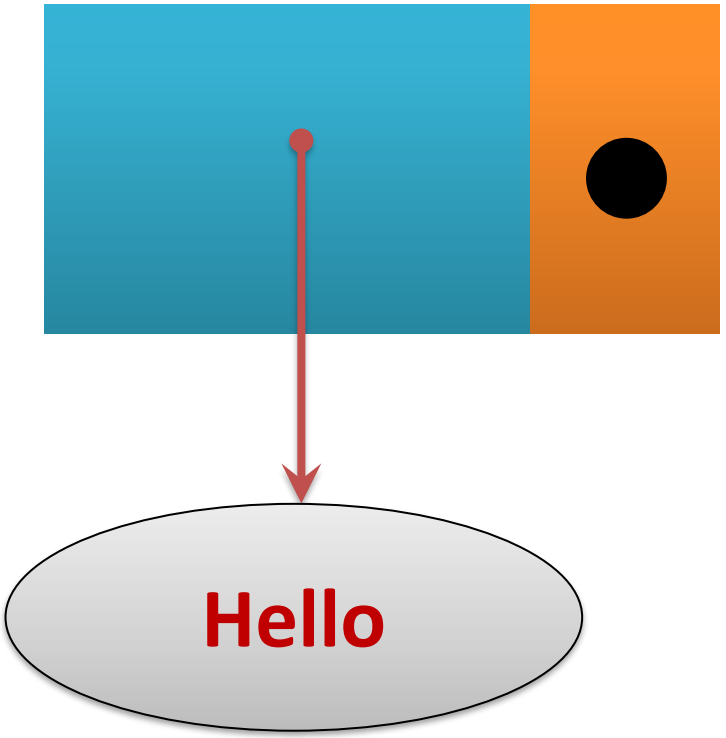
```
public class LinkedList<T> implements ListInterface<T> {
   private Node firstNode;    // reference to first node
   private int numberOfEntries;  // number of entries in list
   .    .       .

   private class Node {
      private T    data;     // entry in list
      private Node next;     // link to next node

      private Node(T data) {
        this.data = data;
        next = null;
      }

      private Node(T data, Node node) {
        this.data = data;
        this.next = next;
      }
   }


}
```

Review:
What are the contents of a single node?



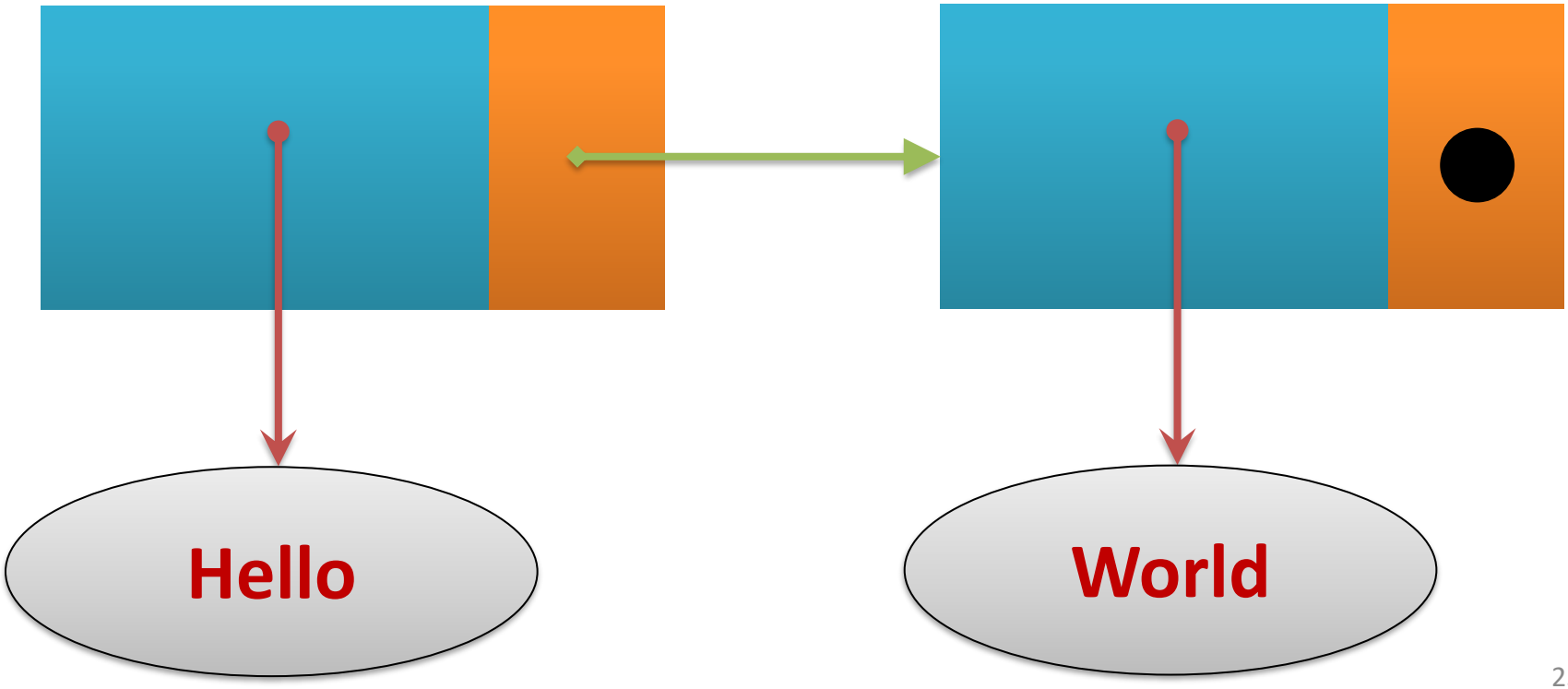
Review:
What are the contents of a single node?
(Example Page 1 of 2)

Node newNode = new Node("Hello");



Review:
What are the contents of a single node?
(Example Page 2 of 2)

```
Node newNode2 = new Node("World");
newNode.next = newNode2;
```



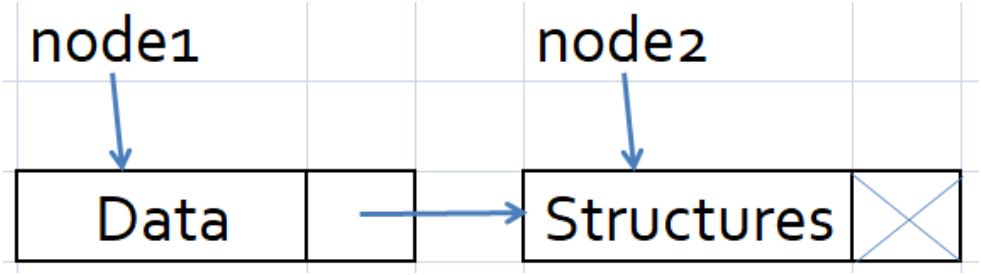
Allocating Memory

- When you use the **`new`** operator, you *create/instantiate* an object.
- At that time, the JRE *allocates/assigns memory* to the object.
- When you create a node for a linked list, we say that you have *allocated the node*.

```
Node newNode = new Node(newEntry);
```

Exercise 5.1

- Write a driver program that builds a linked list of strings containing two nodes as follows:



- Add statements to the driver program so that 2 additional nodes are added to the front of the linked list such that the following output will be displayed using a **`for`** loop:

```
I love Data Structures
```

Exercise 5.1 Answer

```
package samplecode;

public class TestNodes{
   public static void main(String[] args) {
      Node<String> node1 = new Node("Data");
      Node<String> node2 = new Node("Structures");

      node1.setNext(node2); //let node1 point to node2

      Node<String> tempNode = node1; //declare a temporary node which initially points to first node

      while(tempNode != null){
         System.out.print(tempNode.getData() + "  ");
         tempNode = tempNode.getNext();
      }
      System.out.println("\n");

      Node<String> node3 = new Node("I");
      Node<String> node4 = new Node("Love");

      node3.setNext(node4);
      node4.setNext(node1);

      for(tempNode=node3; tempNode != null; tempNode=tempNode.getNext()){
         System.out.print(tempNode.getData() + " ");
      }
      System.out.println("\n");
   }
}
```

Traversing a Linked list

- *Traverse* = move / travel across
- To traverse a linked list
  - Means to traverse the chain from the 1st node to the desired node.
  - Use a temporary reference variable (*e.g.*, `currentNode`) to reference the nodes one at a time:
    - **Initially**, set `currentNode` to `firstNode` so that it references the first node in the chain.
    - To move to the next node, we use the statement

      ```
      currentNode = currentNode.next;
      ```

      until we locate the desired node.

Basics that you need to know

- Adding a node
  - to an empty chain
  - to the front of a chain
  - to the middle of a chain
  - to the back of a chain

- Removing a node
  - the only node
  - the front node
  - a middle node
  - the last node

Linked List Summary

- Can be implemented with only a head reference (as given in the sample code), or both head and tail references.
- Items can be **added/removed** anywhere in the list.
- Traversing the chain is needed based on situation (e.g. inserting in the middle of the chain or during a search operation).
- **Add** to empty & non-empty list:
  - Add a node in the front
  - Add a node in the middle
  - Add a node at the back
- **Remove** from empty & non-empty list:
  - Removing the last node
  - Removing a node from the front
  - Removing a node in the middle
  - Removing a node from the back

Linked Stack Summary

- Implemented with only a head/top reference.
- Items can be added (pushed) and removed (popped) from only the top of the stack.
- Traversal of the chain is rarely needed unless search function is implemented.
- First node represents the top of the stack.
- **Add** to empty & non-empty stack:
  - Add a node *in the front*
- **Remove** from empty & non-empty stack:
  - Removing the last node
  - Removing a node from the front

Linked Queue Summary

- Implemented with both head and tail references.
- Items can be added (enqueue) to the back and removed (dequeue) from the front of the queue.
- Traversal of the chain is rarely needed unless search function is implemented.

- **Add** to empty & non-empty queue:
  - Add a node at the back (tail reference affected).
- **Remove** from empty & non-empty queue:
  - Removing the only node (head and tail references affected)
  - Remove a node from the front of the queue (head reference affected)

What is the goal of learning Data Structures?

- To be able to create a structure to hold the data and give efficient access and storage to the type of data being stored and also provide dynamic expansion on the size if needed.

- Efficiency can be defined as followed:
  - Quick to retrieve.
  - Efficient when performing insertion, modification and deletion.
  - Most efficient management of storage space required.

- Different systems have different needs. Some system should have more efficient retrieval compared to insertion or deletion.

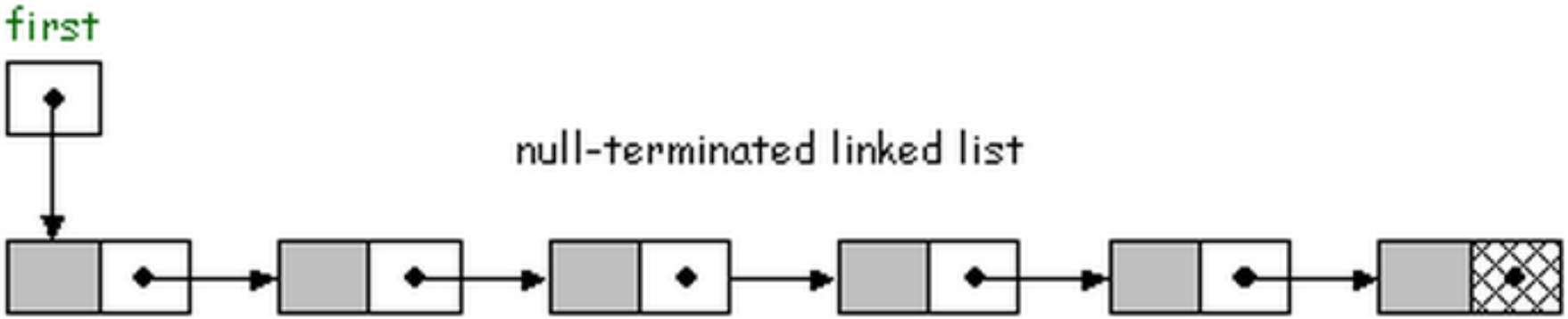
Linked Implementation of List ADT

Recall: Linked List Summary

- Items can be **added/removed** anywhere in the list.
- Traversing the chain is needed based on situation (e.g. inserting in the middle of the chain).
- **Add** to empty & non-empty list:
  - Add a node in the front
  - Add a node in the middle
  - Add a node at the back
- **Remove** from empty & non-empty list:
  - Removing the last node
  - Removing a node from the front
  - Removing a node in the middle
  - Removing a node from the back

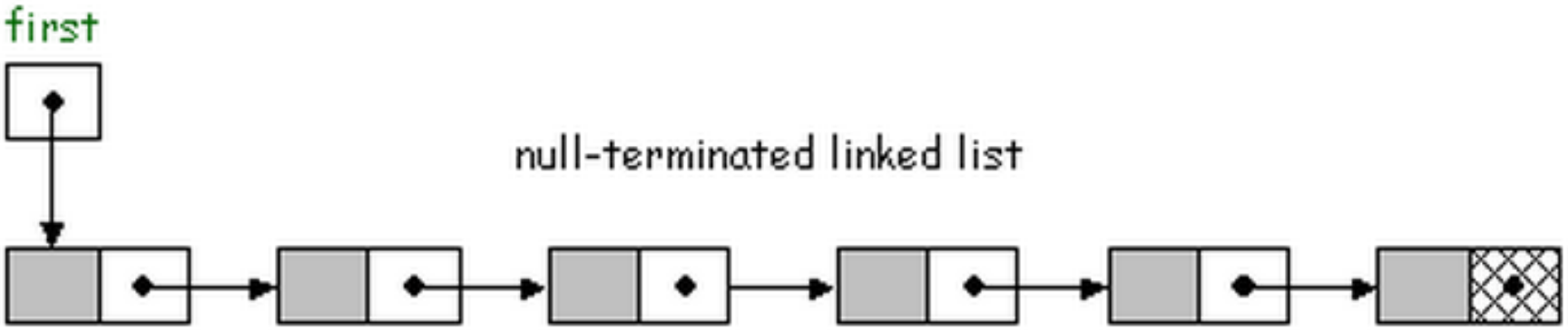
Linked Lists

- A **linked list** is a collection of nodes.
- Each node **points to the next** node in the list.
- The basic (default) linked list is a *linear* linked list.



3 Rules for a Linear Linked List

1. There must be an *external reference (pointer)* to the **first node** in order to access a linked list.
2. The last node's **next** field must be **`null`**.
3. Other than the last node, each node's **next** field must contain the address of the next node.



A Linked Implementation of the ADT List

- Use a chain of nodes to represent the list's entries
- Need a *head reference* to store the reference to the first node
- Sample code in `Chapter5\adt` folder:
  - **`LinkedList.java`**
    - The variable **`firstNode`** contains a reference to the 1st node in the chain,
    - The 1st node contains a reference to the 2nd node,
    - The 2nd node contains a reference to the 3rd node, …
    - …and so on.
  - **`ListInterface.java`** (same as Chapter 4's)

CheatSheet

- In pointer assignment, the outcome is that the **LHS** variable/pointer will point to the same thing as the **RHS** pointer.
- The following statement updates `currentNode` to point to the next node in the linked list

```
currentNode = currentNode.next
```

Adding at Beginning of the List

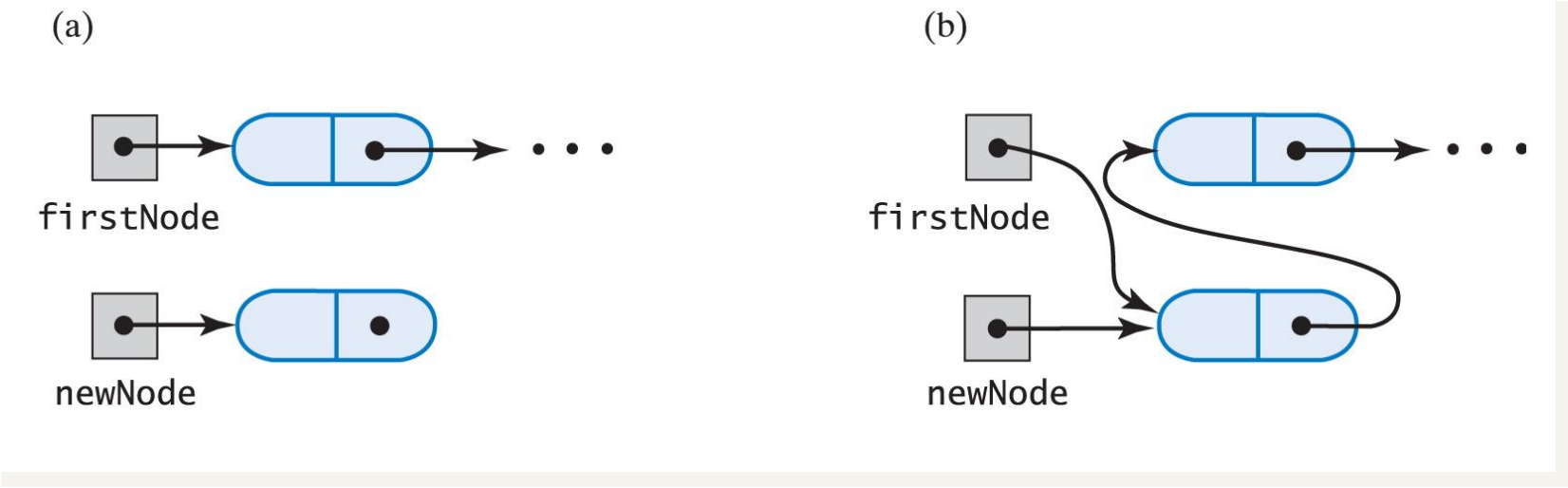


Fig. 15 A chain of nodes (a) prior to adding a node at the beginning; (b) after adding a node at the beginning.

Diagram:
Adding to a List (Empty List)

Diagram:
Adding to a List (First Node)

Adding to the End of the List

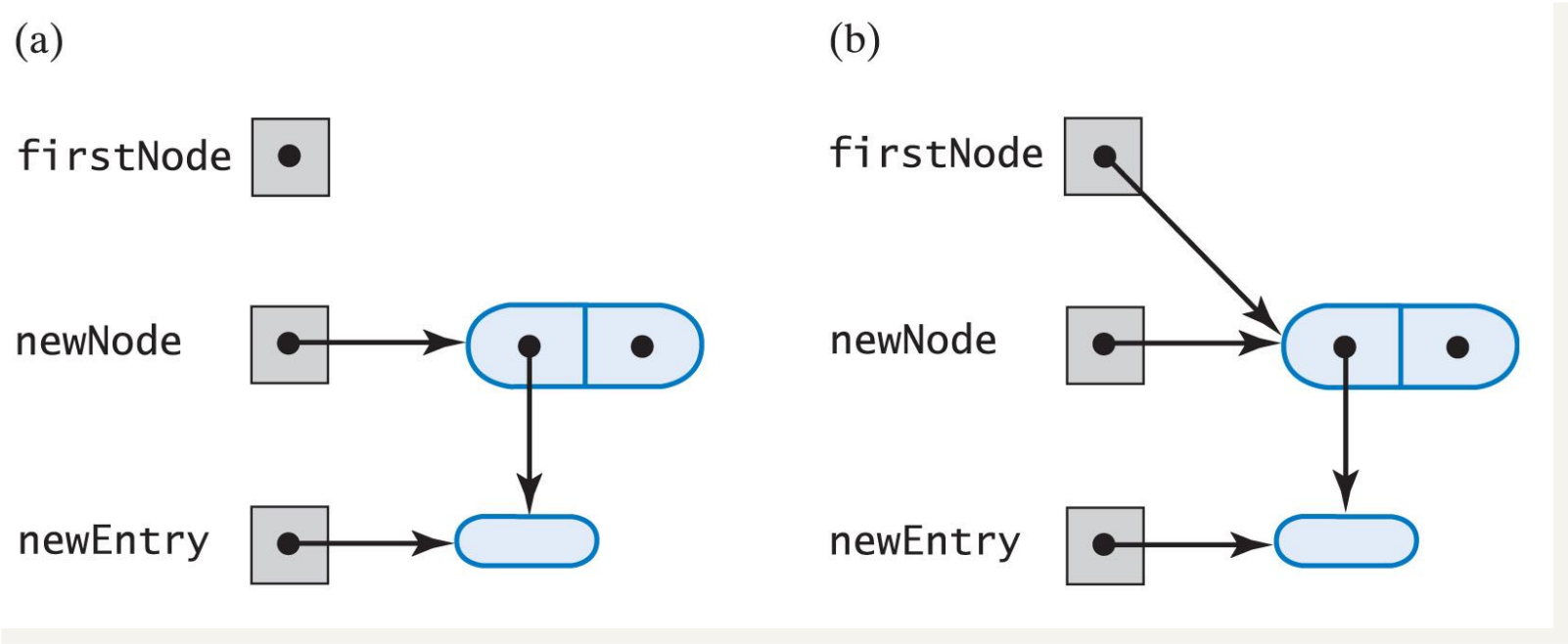


Fig. 13 (a) An empty list and a new node;
(b) after adding a new node to a list that was empty

Adding to the End of the List

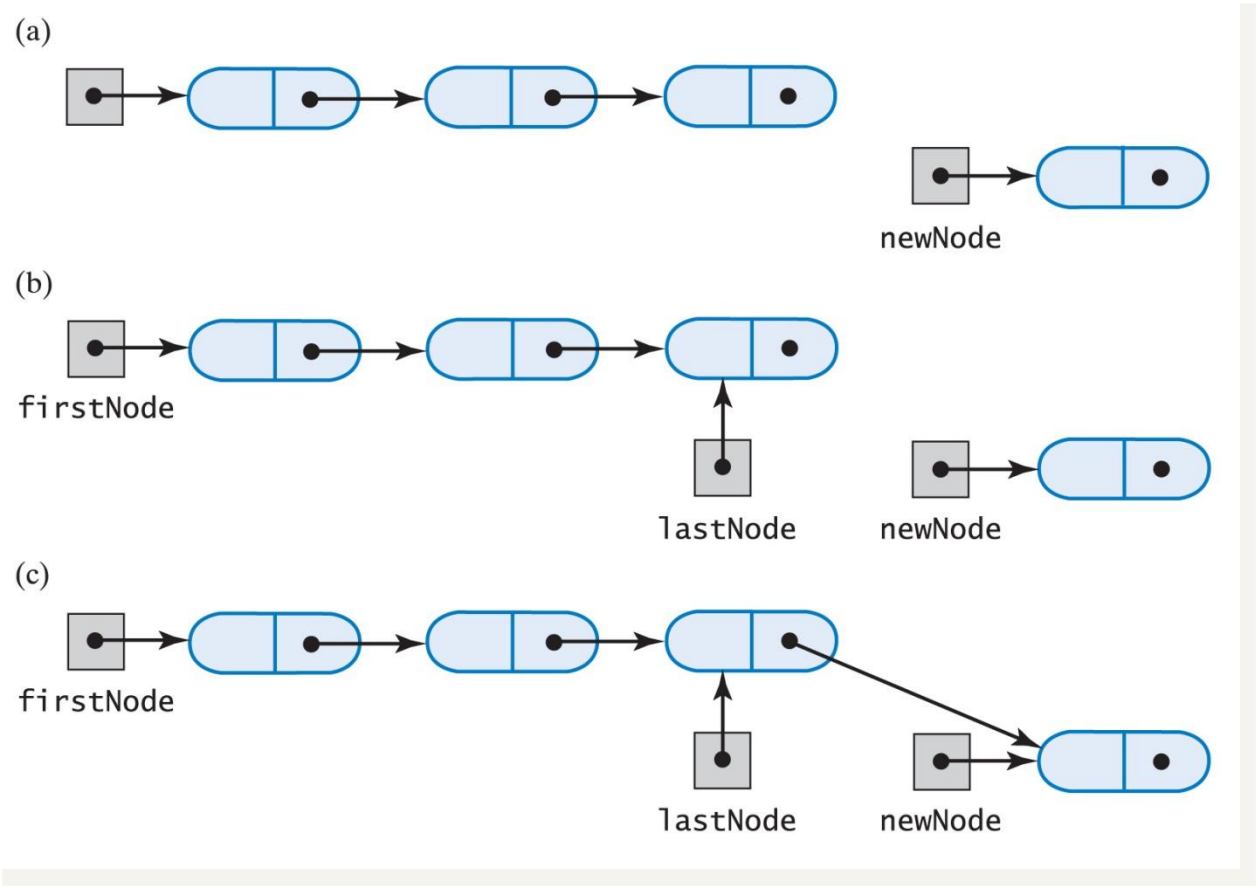


Fig. 14 A chain of nodes (a) just prior to adding a node at the end; (b) just after adding a node at the end.

Diagram:
Adding to a List (Last Node)

LinkedList add method

```
@Override
public boolean add(T newEntry) {
  Node newNode = new Node(newEntry);

  if (isEmpty()) {
    firstNode = newNode;
  } else {
    Node currentNode = firstNode;
    while (currentNode.next != null) {
      currentNode = currentNode.next;
    }
    currentNode.next = newNode;
  }

  numberOfEntries++;
  return true;
}
```

Adding Within (in "middle of ") the List

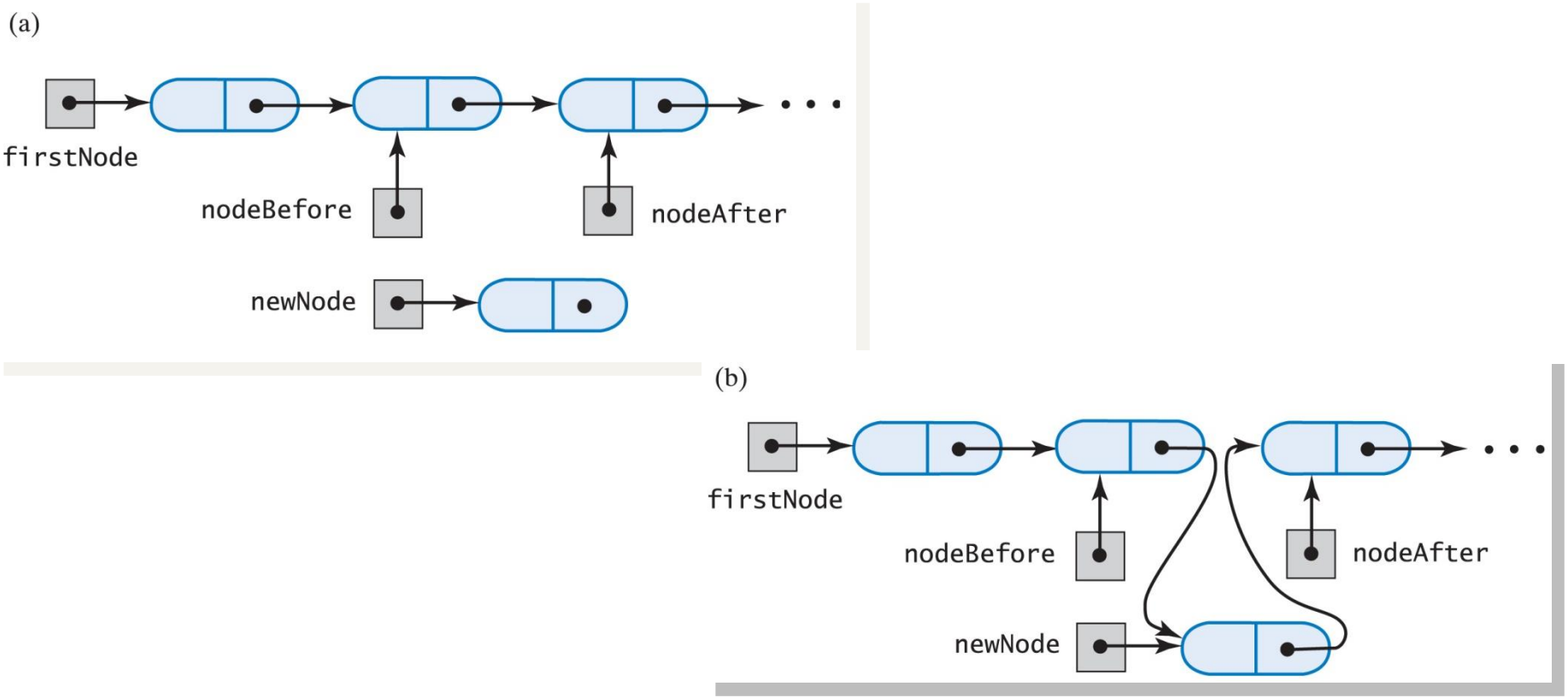


Fig. 16 A chain of nodes (a) prior to adding node between adjacent nodes; (b) after adding node between adjacent nodes

Diagram:
Adding to a List (Middle Node)

```
@Override
public boolean add(int newPosition, T newEntry) { // Ou
  boolean isSuccessful = true;

  if ((newPosition >= 1) && (newPosition <= numberOfEntries + 1))
    Node newNode = new Node(newEntry);

    if (isEmpty() || (newPosition == 1)) { // case 1: add to beginning
      newNode.next = firstNode;
      firstNode = newNode;
    } else {
      Node nodeBefore = firstNode;
      for (int i = 1; i < newPosition - 1; ++i) {
        nodeBefore = nodeBefore.next;
      }

      newNode.next = nodeBefore.next;      // make new node point t
      nodeBefore.next = newNode;                    // make the n
    }

    numberOfEntries++;
  } else {
    isSuccessful = false;
  }

  return isSuccessful;
}
```

add to the beginning

add to the middle or end

Traverse to find the node before the target node

```
} else {
  Node nodeBefore = firstNode;
  for (int i = 1; i < newPosition - 1; ++i) {
    nodeBefore = nodeBefore.next;
  }

  newNode.next = nodeBefore.next;      //
  nodeBefore.next = newNode;
```

Removing an Item from a Linked Chain

- Possible cases
  1. Removing from an empty list.
  2. Remove from beginning (first node) of the chain
  3. Remove from middle of the chain
  4. Remove from the end (last node) of the chain

`remove()` - removing first node

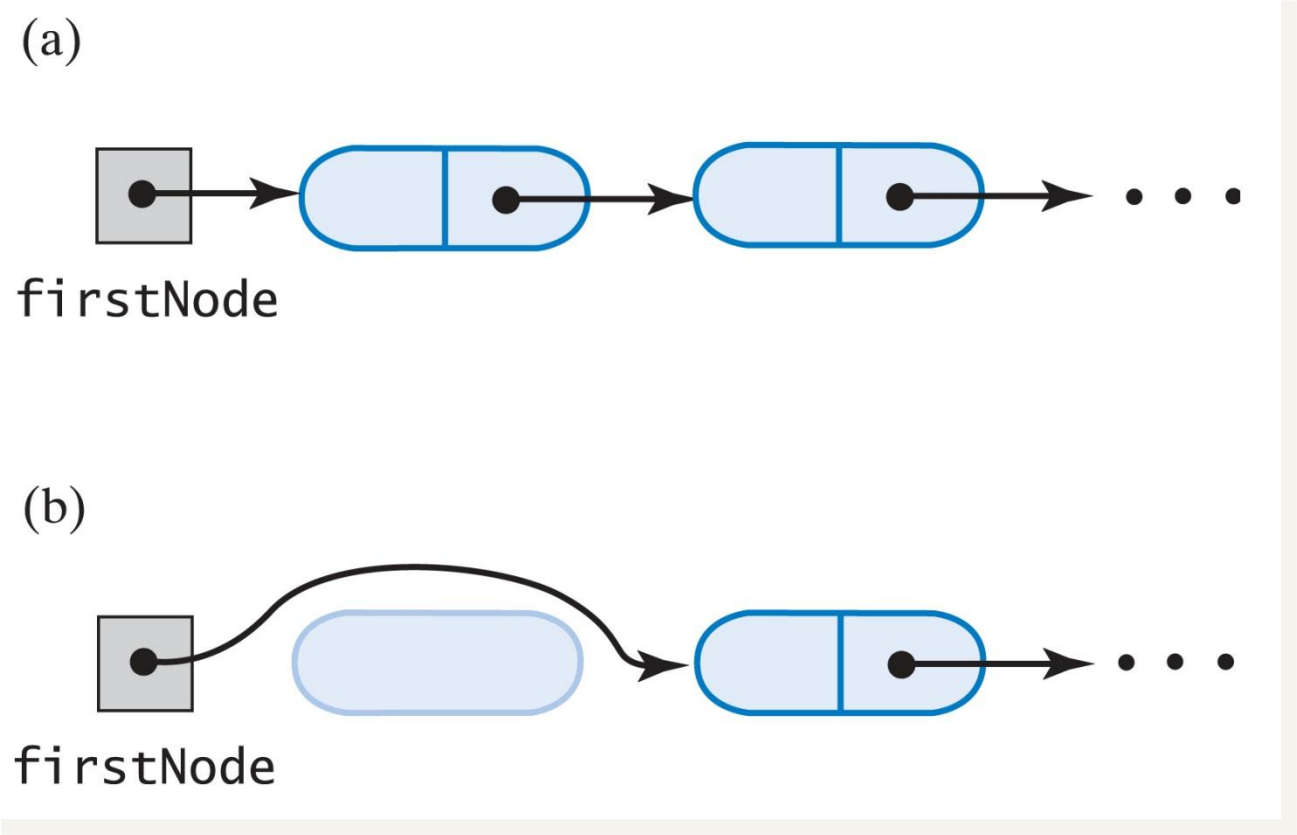
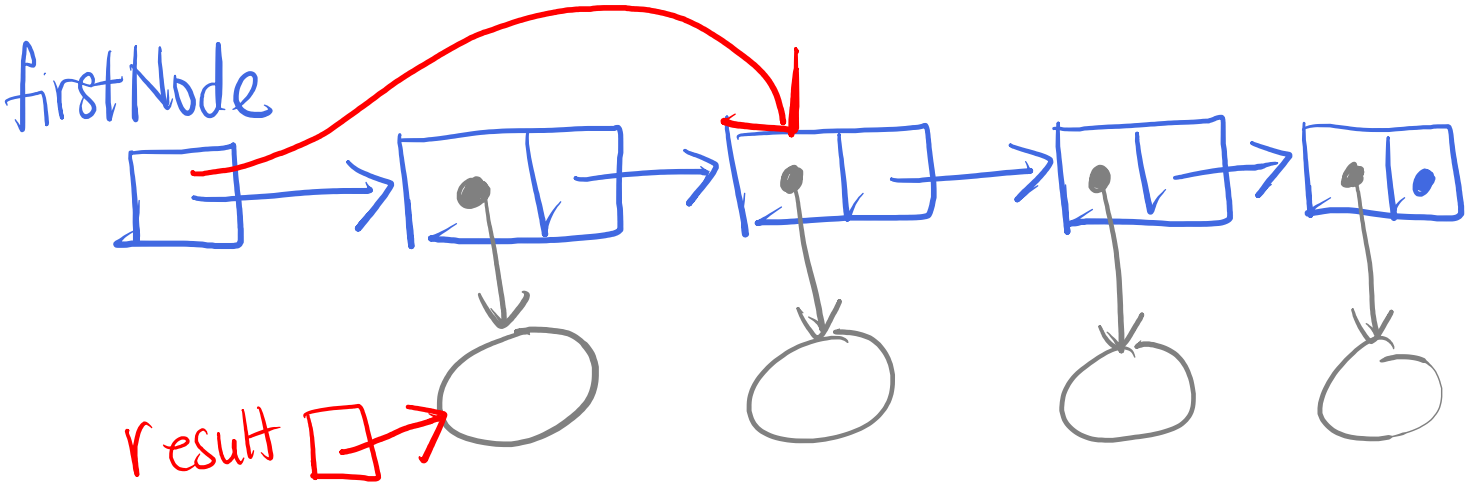


Fig. 22 A chain of nodes (a) prior to removing first node; (b) after removing the first node

Example of removing an item (front/first node)

```
if (givenPosition == 1) {
  result = firstNode.data;
  firstNode = firstNode.next;
```



`remove()` - removing interior node

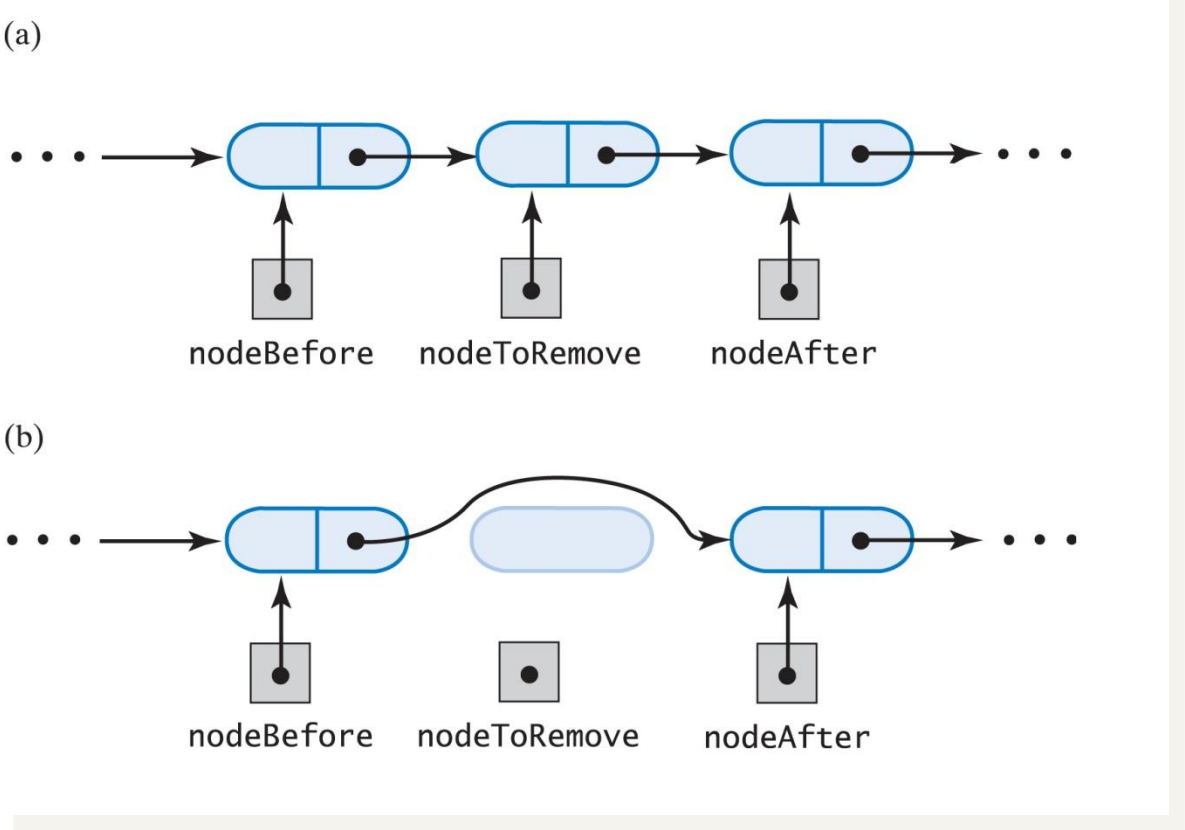


Fig. 23 A chain of nodes (a) prior to removing interior node; (b) after removing interior node

Example of removing an item (middle node)

```
Node nodeBefore = firstNode;
for (int i = 1; i < givenPosition - 1; ++i) {
  nodeBefore = nodeBefore.next;
```

```
result = nodeBefore.next.data;  // save entr
nodeBefore.next = nodeBefore.next.next;
```

Example of removing an item (last node)

```
Node nodeBefore = firstNode;
for (int i = 1; i < givenPosition - 1; ++i) {
  nodeBefore = nodeBefore.next;
```

```
result = nodeBefore.next.data;  // save entr
nodeBefore.next = nodeBefore.next.next;
```

LinkedList remove method

```
@Override
public T remove(int givenPosition) {
  T result = null;                    // return value

  if ((givenPosition >= 1) && (givenPosition <= numberOfEntries)) {
    if (givenPosition == 1) {         // case 1: remove first entry
      result = firstNode.data;        // save entry to be removed
      firstNode = firstNode.next;
    } else {                          // case 2: givenPosition > 1
      Node nodeBefore = firstNode;
      for (int i = 1; i < givenPosition - 1; ++i) {
        nodeBefore = nodeBefore.next;                    // advance nodeB
      }
      result = nodeBefore.next.data;  // save entry to be removed
      nodeBefore.next = nodeBefore.next.next;            // make node bef
    }

    numberOfEntries--;
  }

  return result; // return removed entry, or null if operation fails
}
```

find node before target node

Deallocating Memory

- After the method **`remove`** removes a node from a linked list, you have no way to reference the removed node.
- The JRE implements automatic garbage collection, i.e.
  - It automatically deallocates and recycles the memory associated with the node that has **no references** to it
  - No explicit program statement is necessary

Sample Code

- Chapter5\adt\
  - ListInterface.java
  - **LinkedList.java**
- Chapter5\entity\
  - Runner.java
- Chapter5\client\
  - Registration.java

Efficiency of Implementations of ADT List

- For array-based implementation - **ArrayList**
  - Add to end of list **O(1)**
  - Add to list at given position **O(n)** → Shifting required
  - Retrieving an entry **O(1)**
- For linked implementation - **LinkedList**
  - Add to end of list **O(n)**
  - Add to list at given position **O(n)**
  - Retrieving an entry **O(n)**

Comparing Implementations

Operation	Fixed-Size Array	Linked
`add(new Entry)`	O(1)	O(n)
`add(newPosition, newEntry)`	O(n) to O(1)	O(1) to O(n)
`remove(givenPosition)`	O(n) to O(1)	O(1) to O(n)
`replace(givenPosition, newEntry)`	O(1)	O(1) to O(n)
`getEntry(givenPosition)`	O(1)	O(1) to O(n)
`contains(anEntry)`	O(1) to O(n)	O(1) to O(n)
`display`	O(n)	O(n)
`clear(), getLength(), isEmpty(), isFull()`	O(1)	O(1)

Fig. 9.13: The time efficiencies of the ADT list operations for two implementations, expressed in Big Oh notation

Note: Assuming a fixed-size array, i.e. that the `doubleArray()` method is never invoked.

Tail References

- Consider a set of data where we repeatedly add data to the end of the list
- Each time the **add** method must traverse the whole list
  - This is inefficient
- **Solution**: maintain a pointer that always keeps track of the end of the chain
  - The tail reference

Tail References

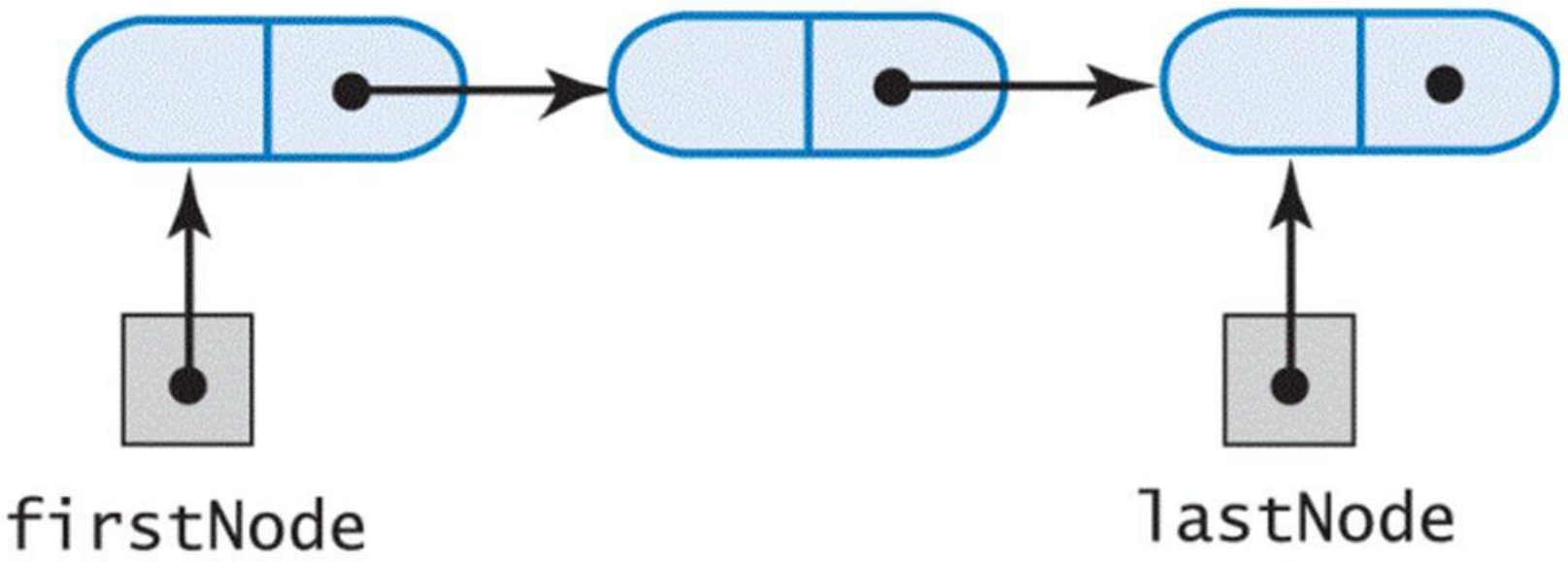


Fig. 24 A linked chain with a head and tail reference.

Tail References

- Private data fields – need to add the external reference to the last node:

```
private Node firstNode;
private Node lastNode;
private int  numberOfEntries;
```

Exercise 5.2

Examine the implementation of the class `LinkedList` given in the `adt` folder. Which methods would require a new implementation if you used both a head reference and a tail reference?

Tail References

- Must change the `clear` method
  - Constructor calls it

```
public final void clear() {
  firstNode = null;
  lastNode = null;
  numberOfEntries = 0;
}
```

Tail References

- When adding to an empty list
  - Both head and tail references must point to the new solitary node
- When adding to a non empty list
  - No more need to traverse to the end of the list
  - `lastNode` points to it
  - Adjust `lastNode.next` in new node and `lastNode`

Tail References

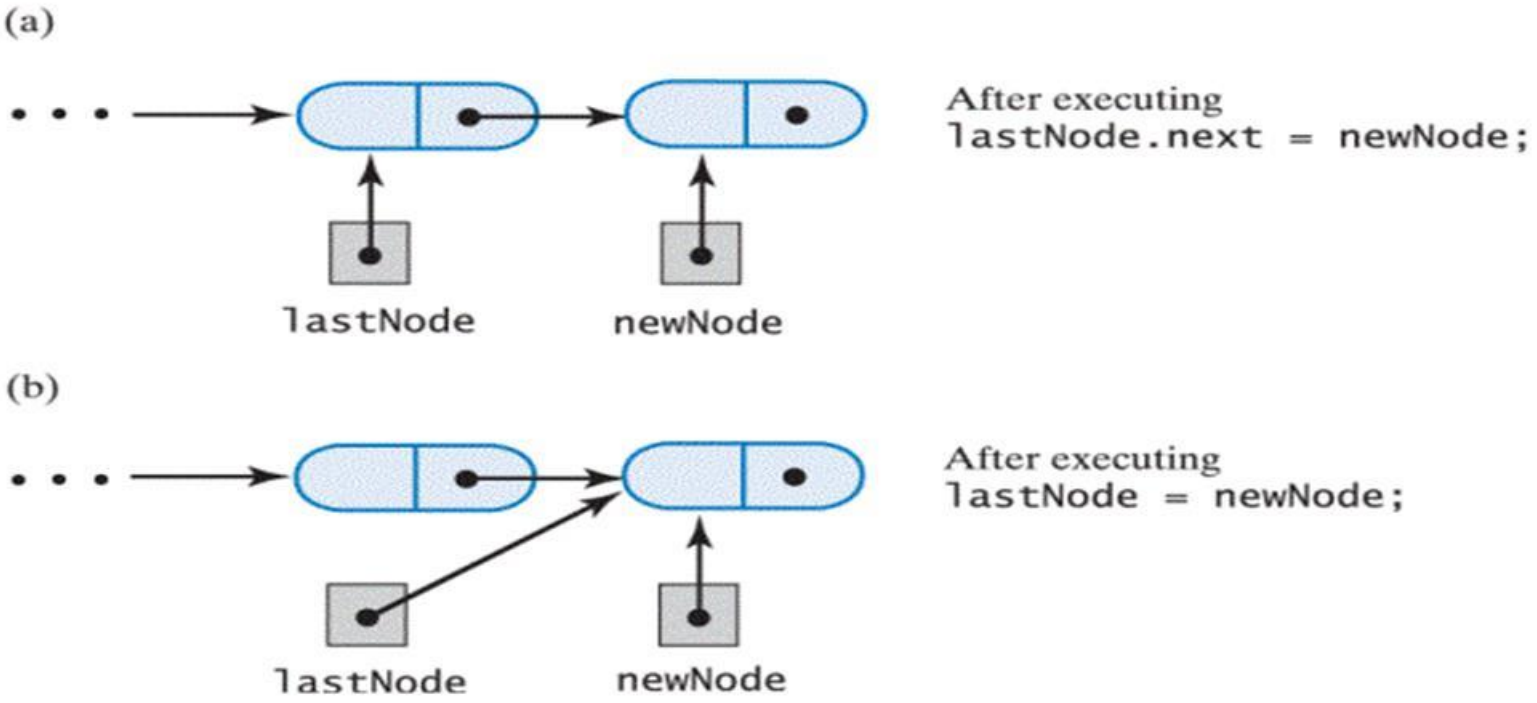


Fig. 25 Adding a node to the end of a nonempty chain that has a tail reference

Tail References

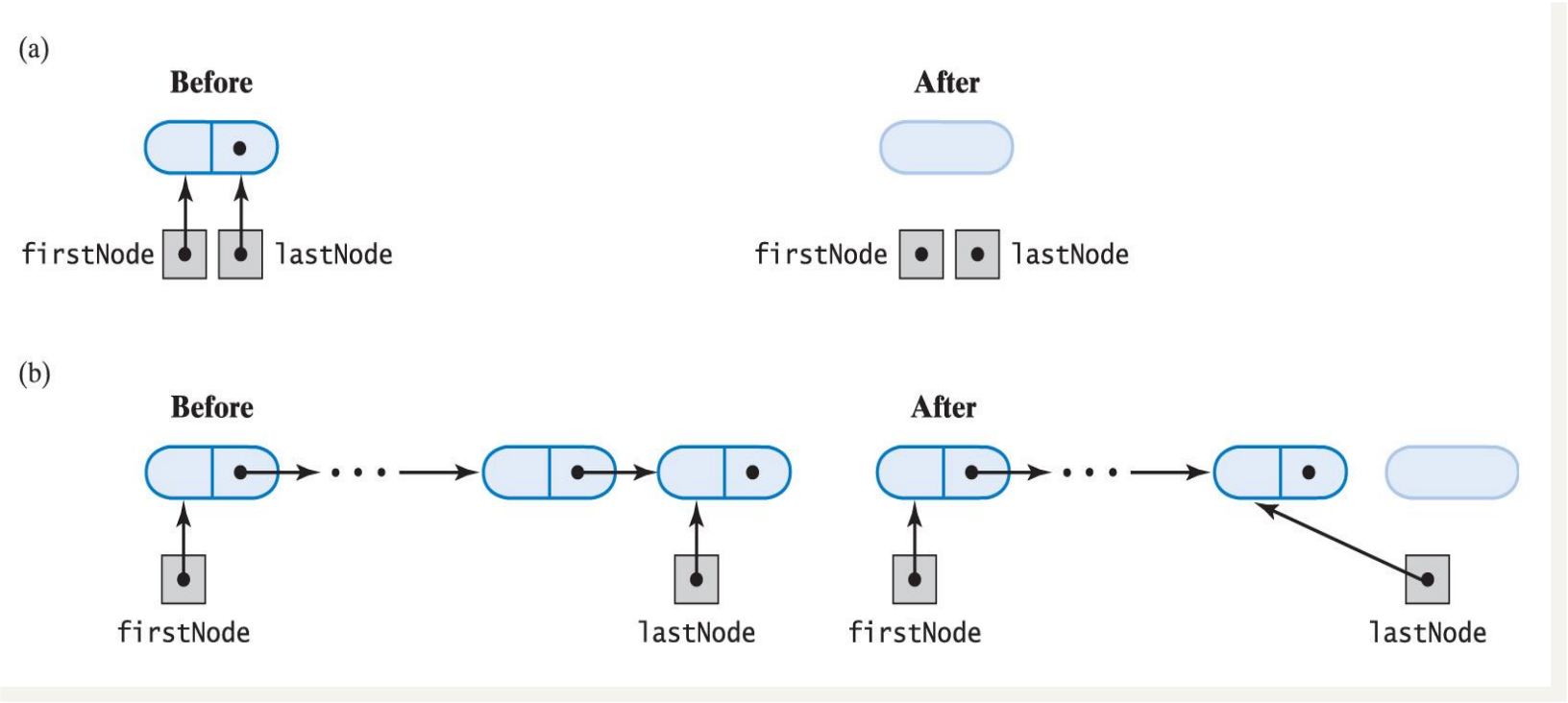
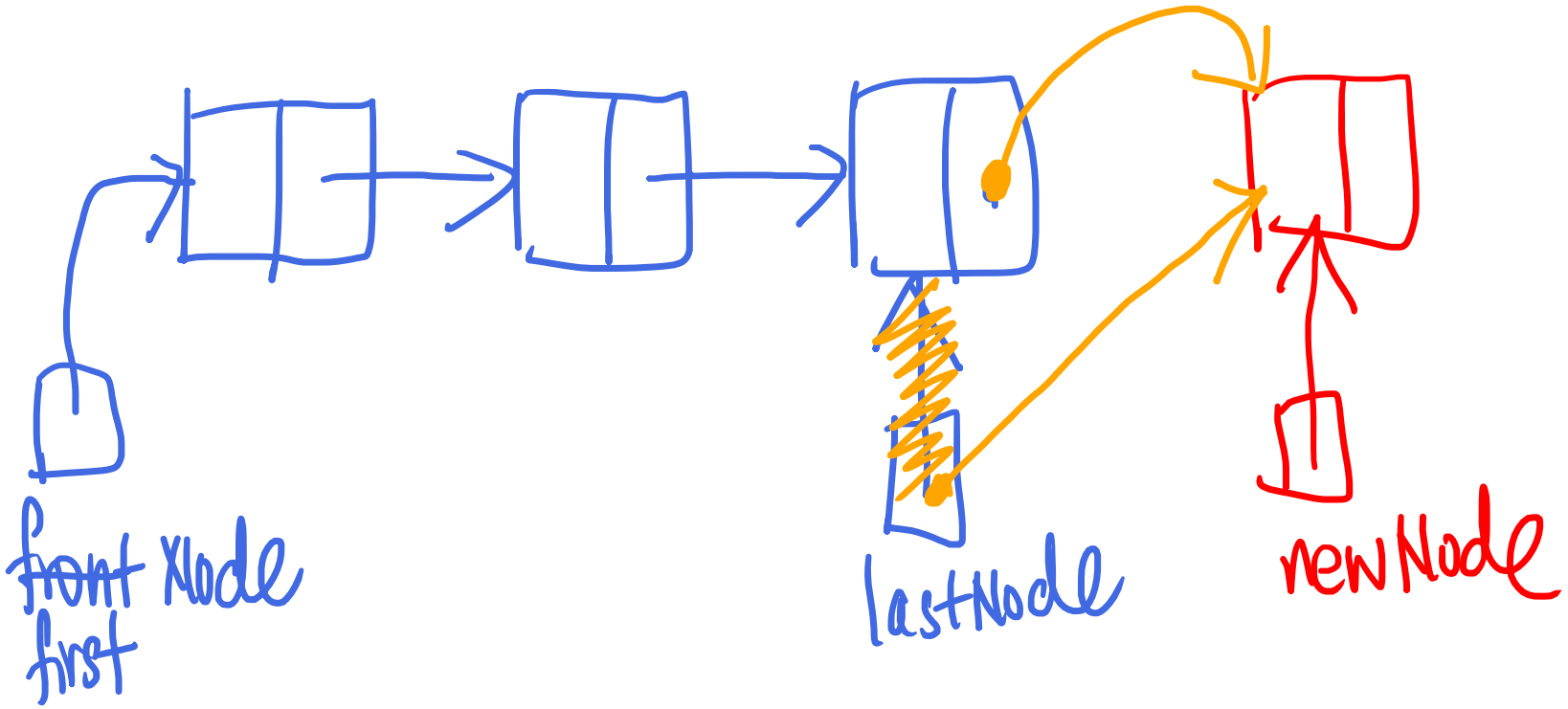


Fig. 26 Removing a node from a chain that has both head and tail references when the chain contains (a) one node; (b) more than one node

Adding to a non-empty list with Tail Reference

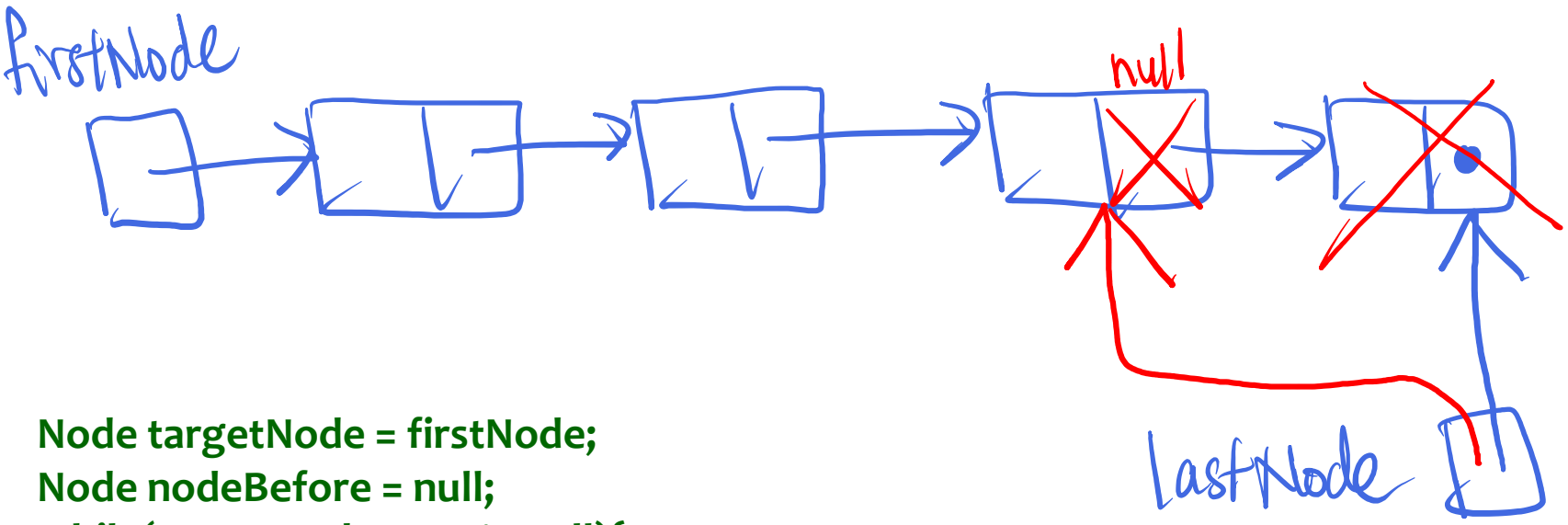



```
lastNode.next = newNode;
lastNode = newNode;
```

Adding to a non-empty list with Tail Reference

lastNode.next = newNode;
lastNode = newNode;

Removing the last node from a linked list with tail reference




```
Node targetNode = firstNode;
Node nodeBefore = null;
while(targetNode.next != null){
        nodeBefore = targetNode;
        targetNode = targetNode.next;
}
nodeBefore.next = null;
lastNode = nodeBefore;
```

Removing the last node from a linked list with tail reference

```
Node targetNode = firstNode;
Node nodeBefore = null;
while(targetNode.next != null){
        nodeBefore = targetNode;
        targetNode = targetNode.next;
}
nodeBefore.next = null;
lastNode = nodeBefore;
```

Pros and Cons of a Chain for an ADT List

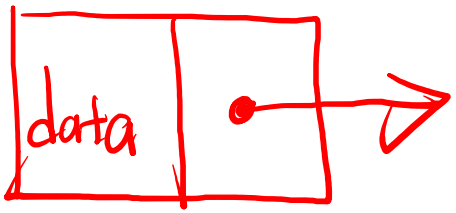
- The chain (list) can grow as large as necessary
- Can add and remove nodes without shifting existing entries

But …

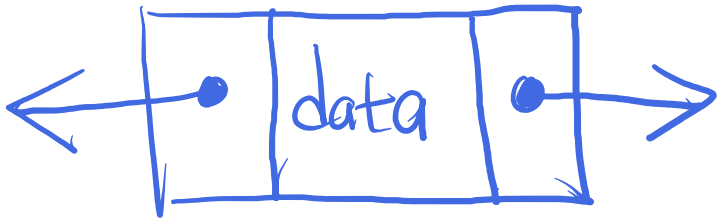
- Must traverse a chain to determine where to make *addition/deletion*
- *Retrieving* an entry requires traversal
  - As opposed to direct access in an array
- Requires more memory for links
  - But does not waste memory for oversized array

Other Variations of Linked Lists

- Linear or Circular
- Singly or Doubly
- With Dummy Node or Without

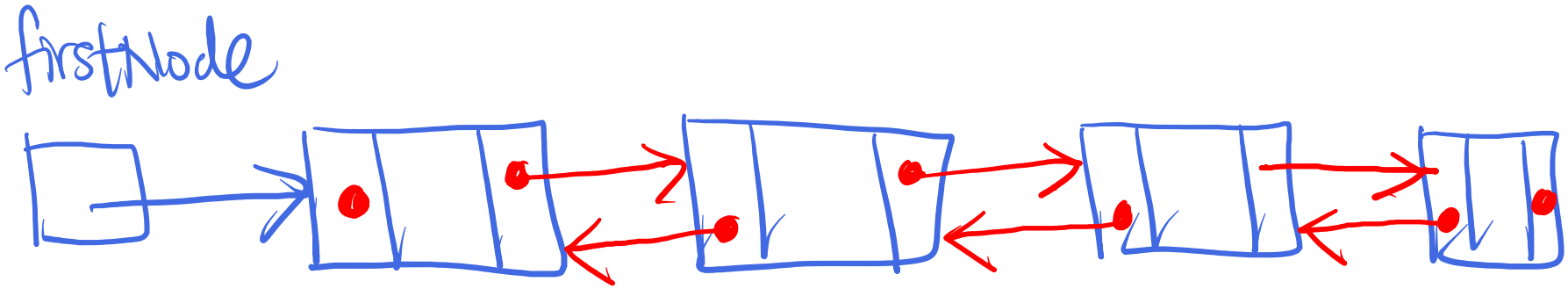


Singly

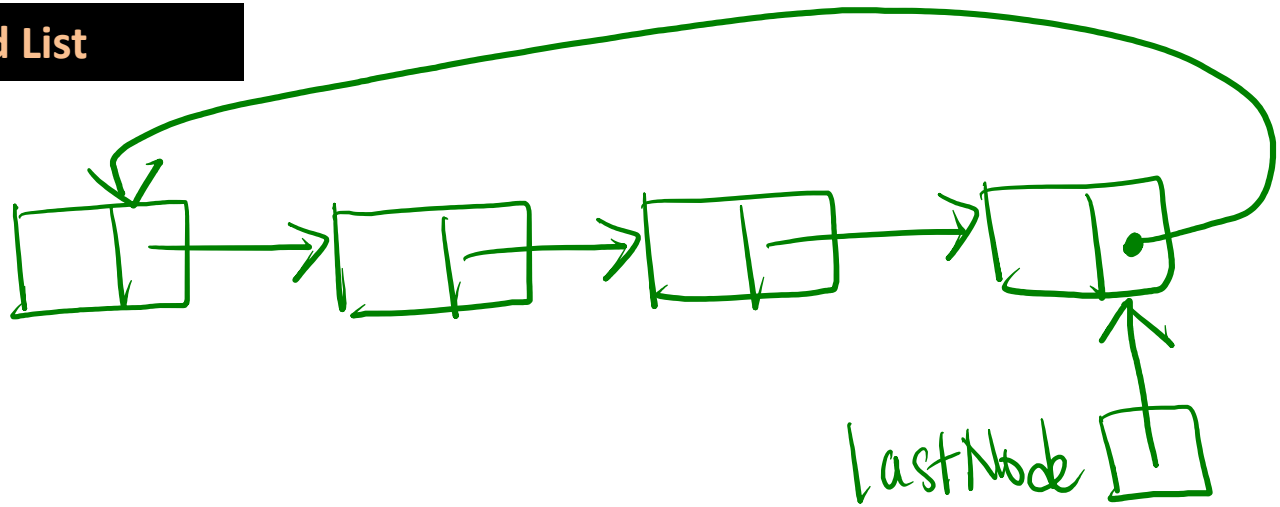


Doubly

Doubly Linked List



Circular Linked List



Java Class Library: The Class `LinkedList`

- The standard java package **`java.util`** contains the class **`LinkedList`**
- This class implements the interface **`List`**
- Contains additional methods:
  - **`addFirst()`**
  - **`addLast()`**
  - **`removeFirst()`**
  - **`removeLast()`**
  - **`getFirst()`**
  - **`getLast()`**
- Refer to Appendix 5.1 for diagrams illustrating relationships between Java Collection Framework interfaces and classes

Linked Implementation of Stack ADT

Recall: Linked Stack Summary

- Items can be added (pushed) and removed (popped) from only the top of the stack.
- Traversal of the chain is rarely needed, unless during a search operation implemented.
- First node represents the top of the stack.
- **Add** to empty & non-empty stack:
  - Add a node *in the front*
- **Remove** from empty & non-empty stack:
  - Removing the last node
  - Remove a node from the front

A Linked Implementation

- When using a chain of linked nodes to implement a stack
  - The **first** node should reference the stack's **top**

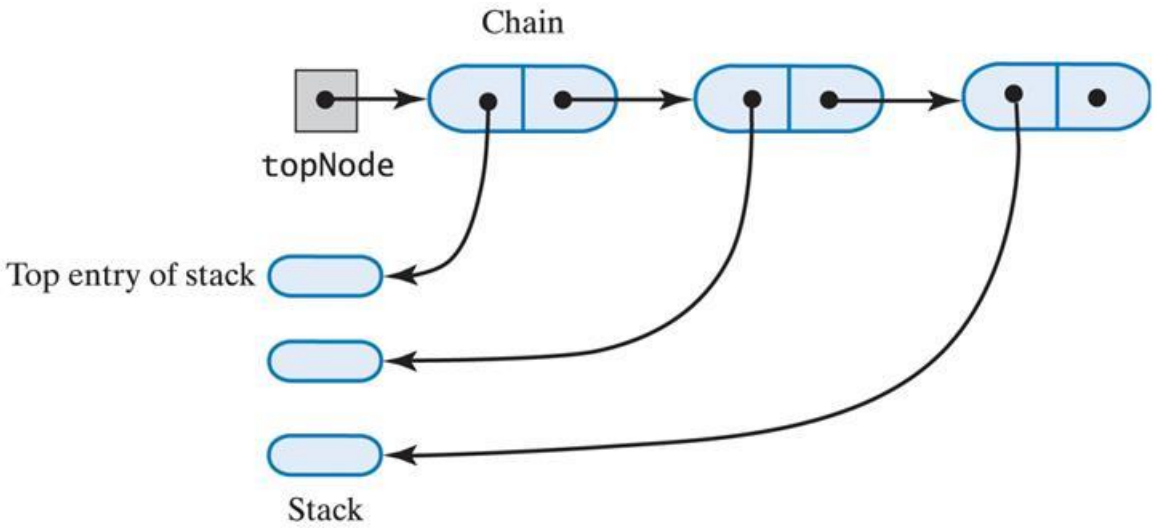


Fig. 27 A chain of linked nodes that implements a stack.

Private variables needed for a Linked Stack

```
public class LinkedStack<T> implements StackInterface<T> {

  private Node topNode; // references the first node in the chain

  public LinkedStack() {
    topNode = null;
  }
```

`push()` method

- Create a new node
- Make the new node point to the current top node
- Make the head reference **`topNode`** point to the new node

Pushing an entry

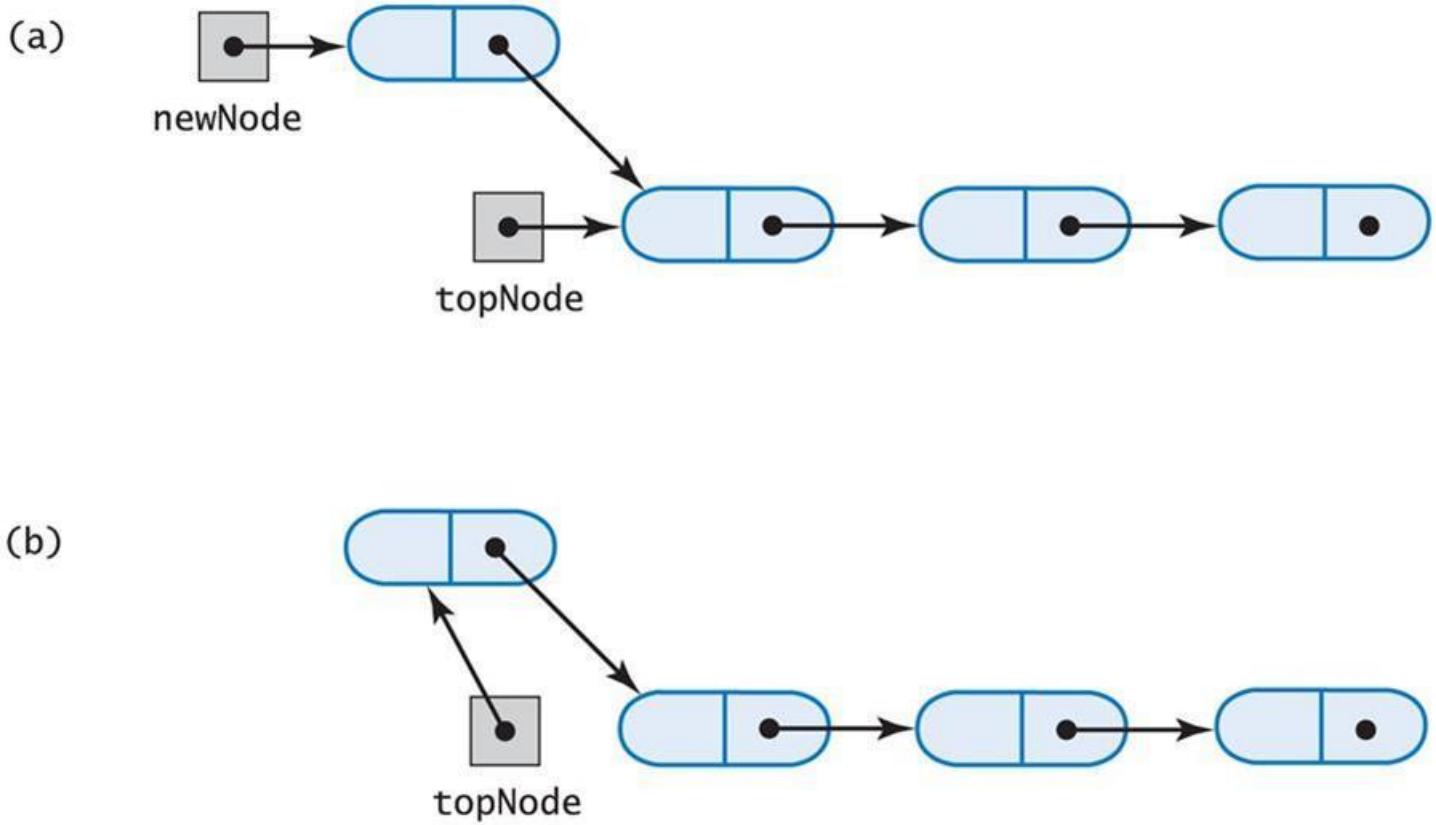


Fig. 28 (a) A new node that references the top of the stack; (b) the new node is now at the top of the stack.

Diagram:
Pushing an item to a Linked Stack

Java code for the push operation

```
public void push(T newEntry) {
  Node newNode = new Node(newEntry, topNode);
  topNode = newNode;
}
```

`pop()` method

If the stack is not empty

- Assign current top node to a reference to be returned
- Make the head reference **`topNode`** point to the next node

Popping an entry (1/2)

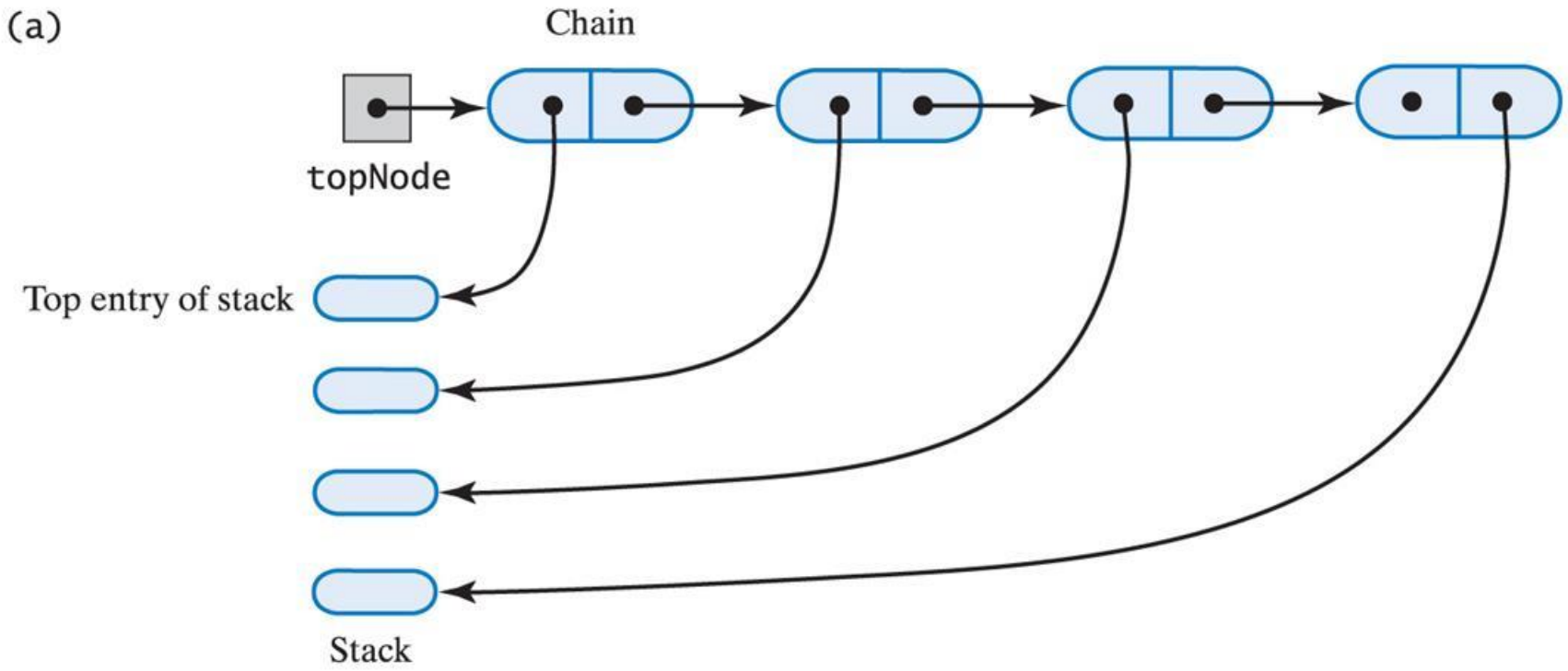


Fig. 29 The stack (a) before the first node in the chain is deleted

Popping an entry (2/2)

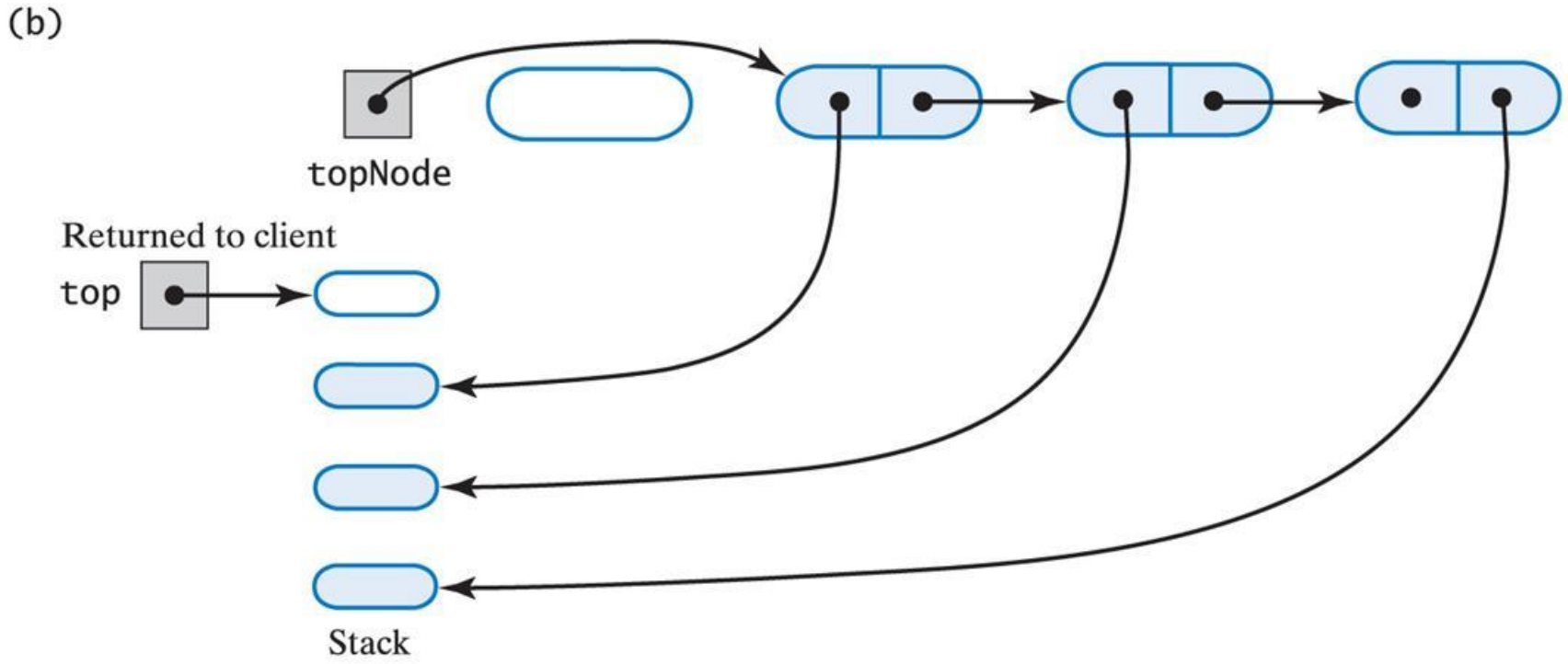


Fig. 30 The stack (b) after the first node in the chain is deleted

Diagram:
Popping an Item from a Linked Stack

Java code for pop() operation

```
public T pop() {
  T top = peek();

  if (topNode != null) {
    topNode = topNode.next;
  }

  return top;
}
```

Linked Implementation of Queue ADT

Linked Queue Summary

- Items can be added (enqueue) to the back and removed (dequeue) from the front of the queue.
- Traversal of the chain is rarely needed, unless if a search function is implemented.
- 2 external references – front and back.
- **Add** to empty & non-empty queue:
  - Add a node at the back (tail reference affected).
- **Remove** from empty & non-empty queue:
  - Removing the last node (tail reference affected)
  - Remove a node from the front of the queue (head reference affected)

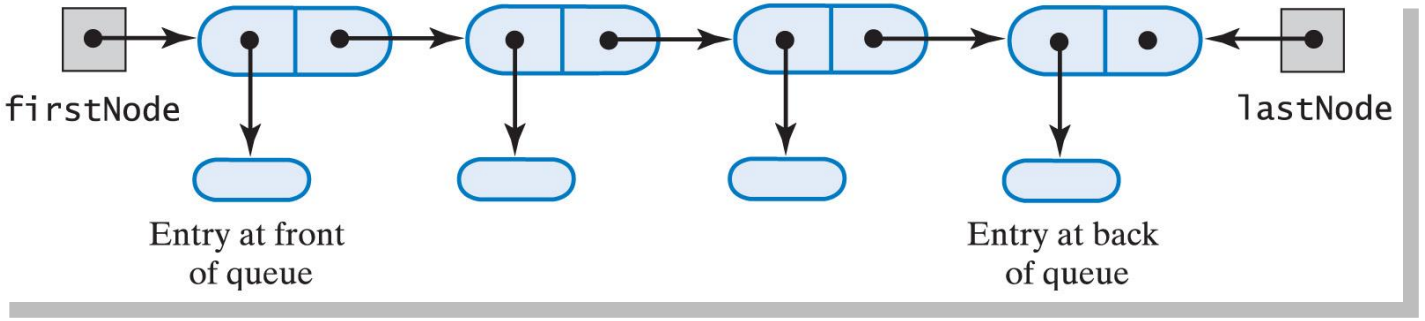
A Linked Implementation of a Queue

- Use a chain of linked nodes for the queue
  - Two ends of the queue are at opposite ends of chain
  - Accessing last node is inefficient
    - Thus, keep a reference to the tail of the chain
- In summary,
  - Place front of queue at beginning of chain
  - Place back of queue at end of chain
  - With references to both

Private variables needed for a Linked Queue implementation

```
public class LinkedQueue<T> implements QueueInterface<T> {

  private Node firstNode; // references node at front of queue
  private Node lastNode;  // references node at back of queue

  public LinkedQueue() {
    firstNode = null;
    lastNode = null;
  }
```

A. Linear Linked Implementation of Queue ADT



Adding to an empty queue

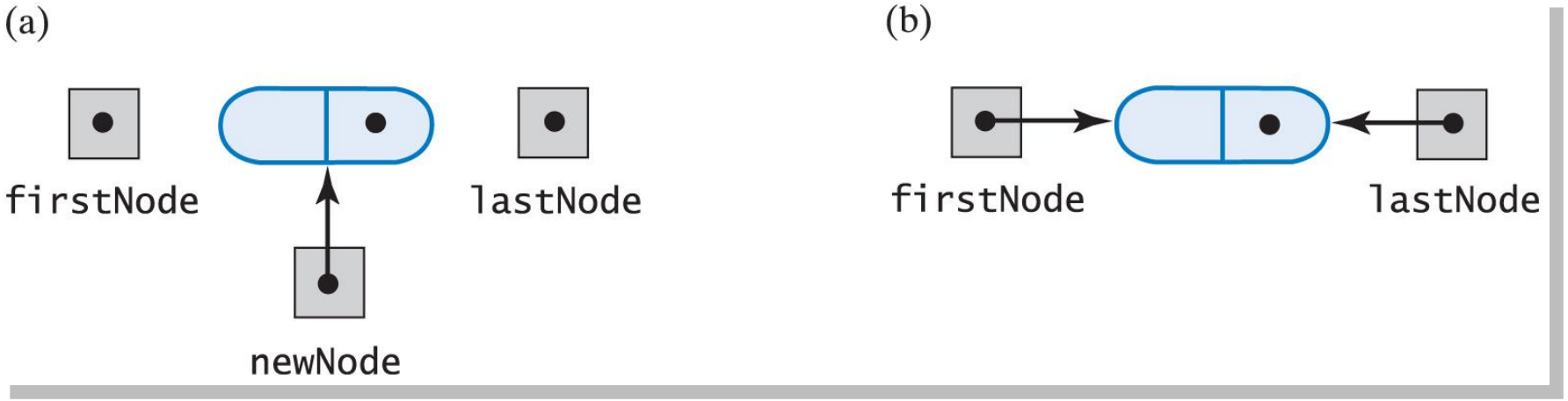


Fig. 32-2(a) Before adding a new node to an empty chain; (b) after adding to it.

Adding to a non-empty queue

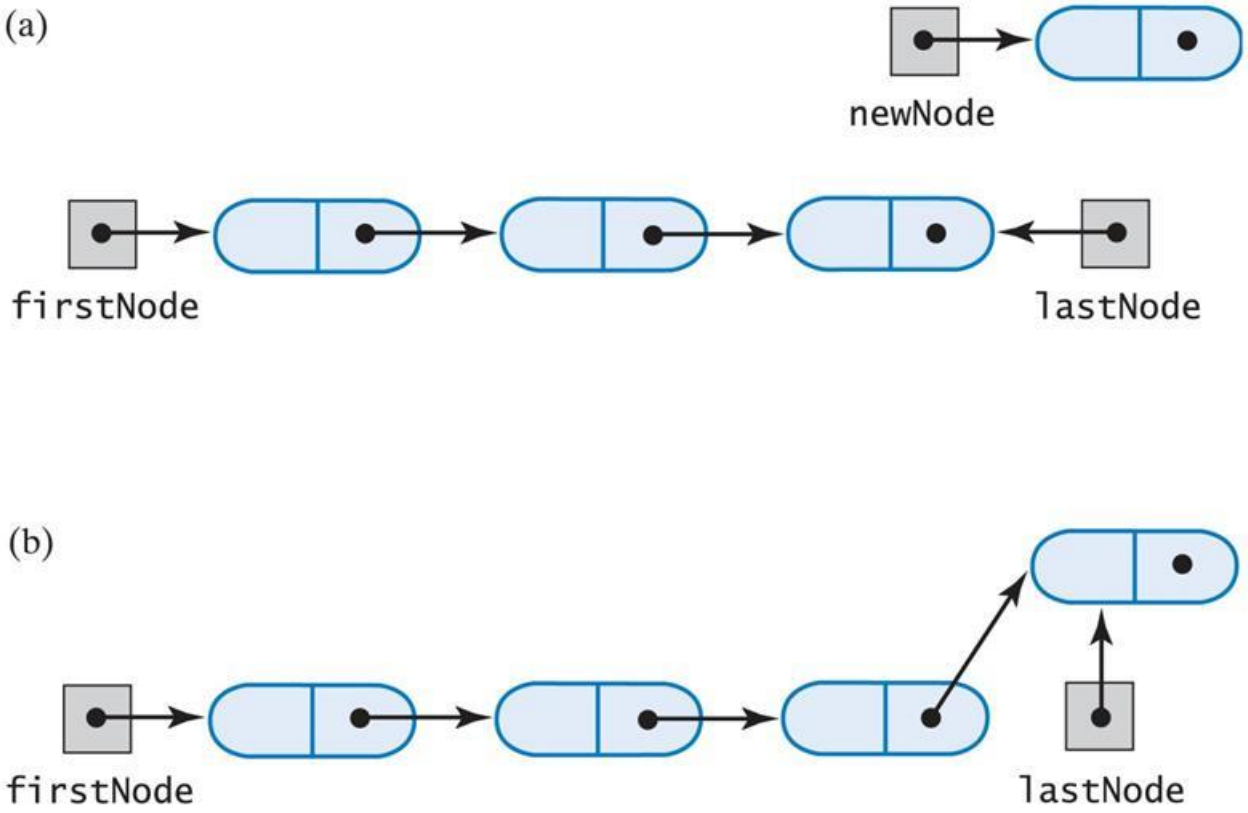


Fig. 32-3(a) Before adding a new node to the end of a chain; (b) after adding it.

`enqueue()` method

1. Create a new node
2. If the queue was empty
   - Make the head reference `firstNode` point to the new node
3. Else [the queue was not empty]
   - Make the current last node point to the new node
4. Make the tail reference `lastNode` point to the new node

Diagram:
Enqueue Operation

Java code for enqueue operation

```
public void enqueue(T newEntry) {
  Node newNode = new Node(newEntry, null);

  if (isEmpty()) {
    firstNode = newNode;
  } else {
    lastNode.next = newNode;
  }

  lastNode = newNode;
}
```

`dequeue()` method

If the queue is not empty

- Assign current first node to a reference to be returned
- Update the head reference `firstNode` to point to the next node in the queue

```
firstNode = firstNode.next;
```

- If the queue is now empty
  - Update the tail reference `lastNode` to null

Removing an entry

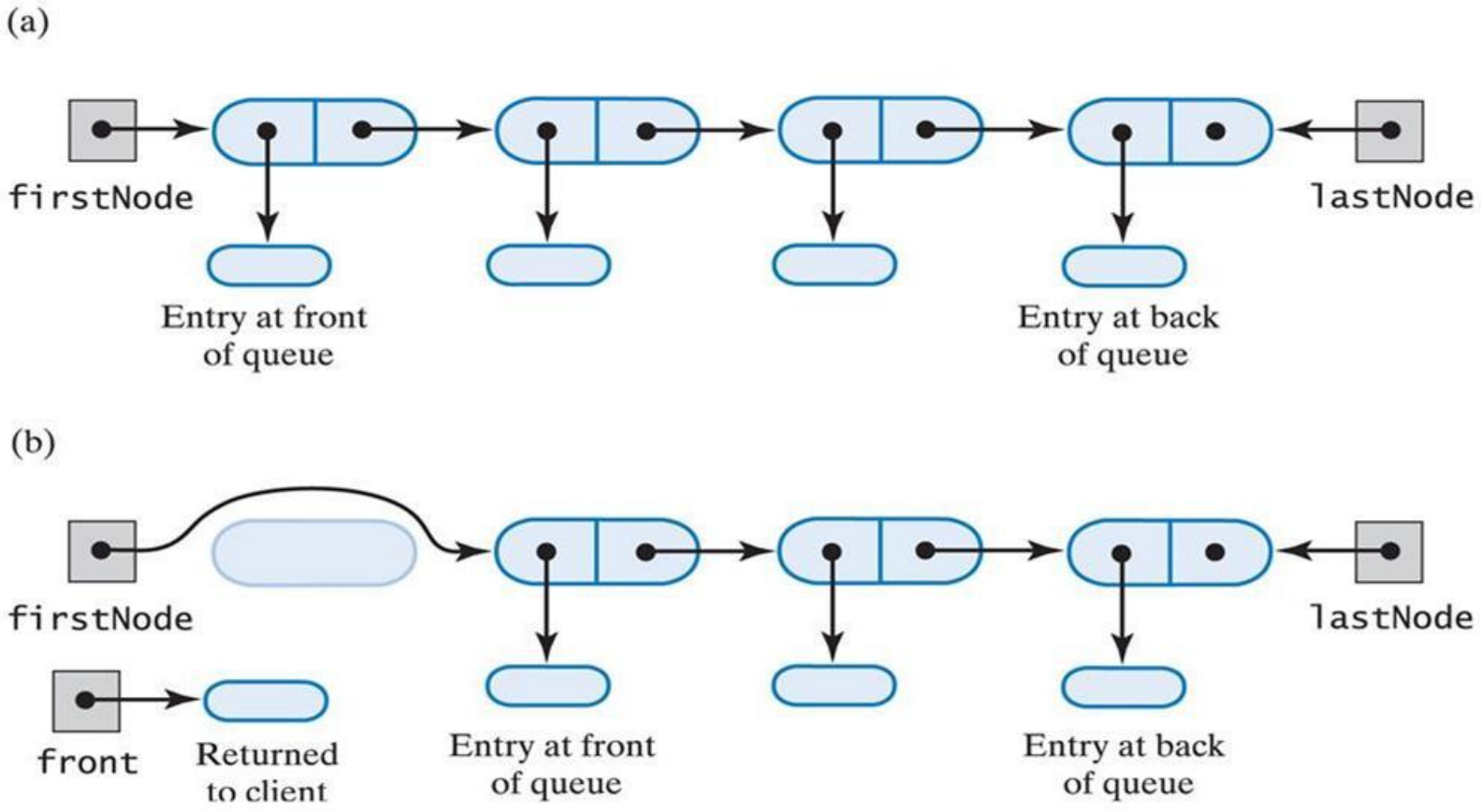


Fig. 32-4(a) A queue of more than one entry; (b) after removing the queue's front.

Removing the only entry from the queue

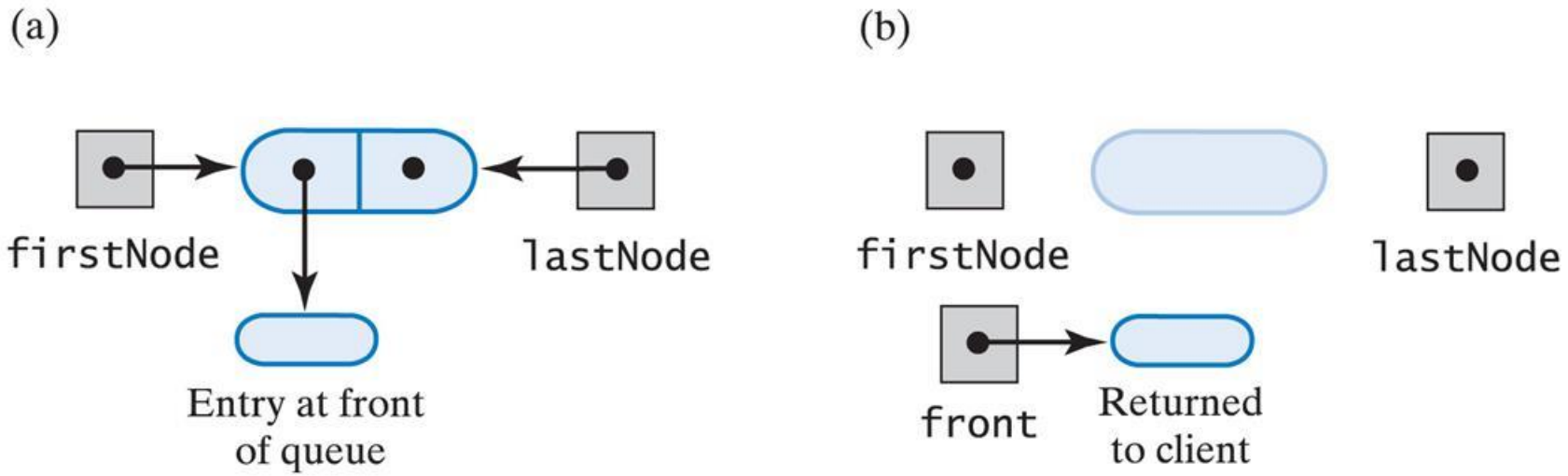


Fig. 32-5 (a) A queue of one entry; (b) after removing the queue's front.

Diagram:
Dequeue Operation

Java Code for dequeue operation

```
public T dequeue() {
  T front = null;

  if (!isEmpty()) {
    front = firstNode.data;
    firstNode = firstNode.next;

    if (firstNode == null) {
      lastNode = null;
    }
  }

  return front;
} // end dequeue
```

A Linked Implementation of a Queue

Refer to `Chapter5\adt\`**`LinkedQueue.java`**

Note:

- methods
  - **`enqueue`**
  - **`getFront`**
  - **`dequeue`**
  - **`isEmpty`**
  - **`clear`**
- **`private class Node`**

Sample Code

- Chapter5\adt\
  - QueueInterface.java
  - **LinkedQueue.java**
- Chapter5\entity\
  - Customer.java
- Chapter5\client\
  - SimulationDriver.java
  - WaitLine.java

Iterators in Linked Implementation (1)

- The iterator class is defined as an *inner class* of the collection ADT
  - Thus, it has direct access to the ADT's data fields.

```
public class LinkedQueue<T> implements QueueInterface<T> {
  private Node firstNode;
  private Node lastNode;
 . . .
 private class LinkedQueueIterator implements Iterator<T> {
     private Node currentNode;
     public LinkedQueueIterator() {
       currentNode = firstNode;
     }
     public boolean hasNext() {
       return currentNode != null;
     }
     public T next() {
       if (hasNext()) {
         T returnData = currentNode.data;
         currentNode = currentNode.next;
         return returnData;
       } else {
         return null;
       }
     }
   }
```

Iterators in Linked Implementation (2)

- Then, provide a public method that returns an iterator to the collection:

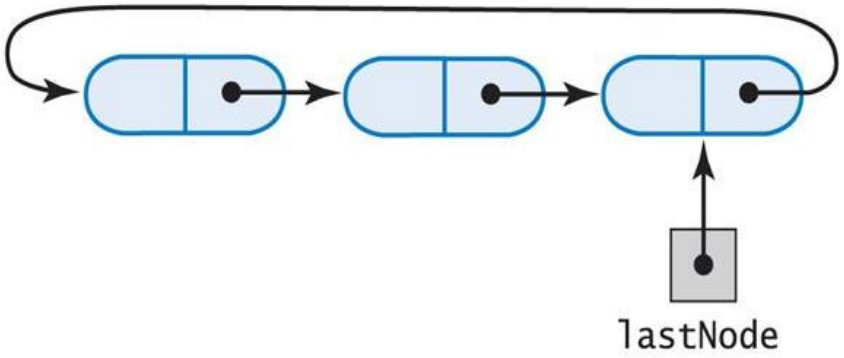
```
public class LinkedQueue<T> implements QueueInterface<T> {
   . . .
   public Iterator<T> getIterator() {
     return new ArrayQueueIterator();
   }
   . . .
}
```

Why is there a need for an interator ?

- Can you **retrieve** an element in the **middle of a queue or a stack**?

- Can you **remove** an element int the **middle of a queue or a stack**?

B. Circular Linked Implementation of Queue ADT



Circular Linked Implementation of Queue (1)

Last node references first node

- Now we have a single reference to last node
- And still locate first node quickly

No node contains a null

When a class uses circular linked chain for queue

- Only one data field in the class: the reference to the chain's last node

```
public class CircularLinkedQueue<T> implements QueueInterface<T> {

  private Node lastNode;  // references node at back of queue

  public CircularLinkedQueue() {
    lastNode = null;
  }
```

Circular Linked Implementation of Queue (2)

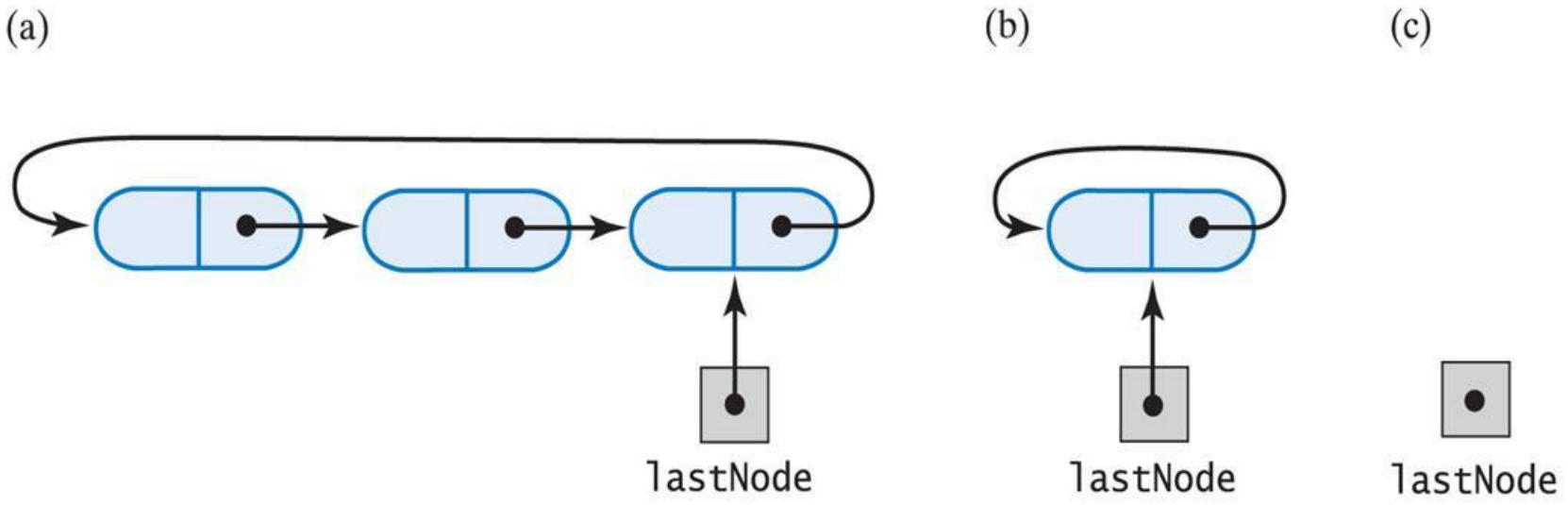


Fig. 33 A circular linked chain with an external reference to its last node that (a) has more than one node; (b) has one node; (c) is empty.

C. Doubly Linked Implementation of Queue ADT

Doubly Linked Implementation of Queue (1)

Singly Linear Linked Implementation

- Chain with head reference enables reference of first and then the rest of the nodes
- Tail reference allows reference of last node but not next-to-last

Solution:

- We need nodes that can reference both
  - Previous node
  - Next node
- Thus the doubly linked chain

Doubly Linked Implementation of Queue (2)

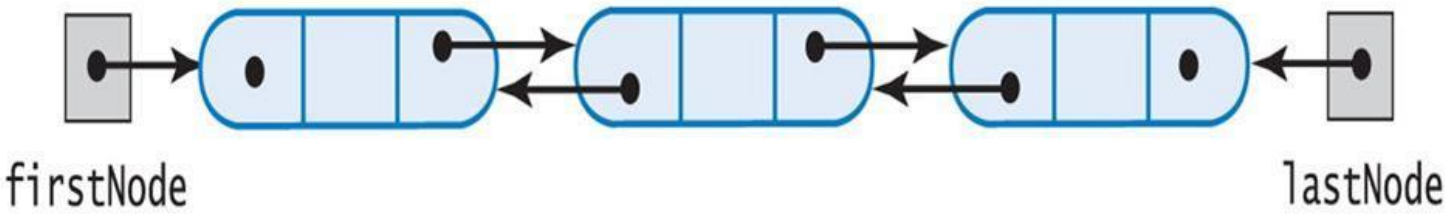
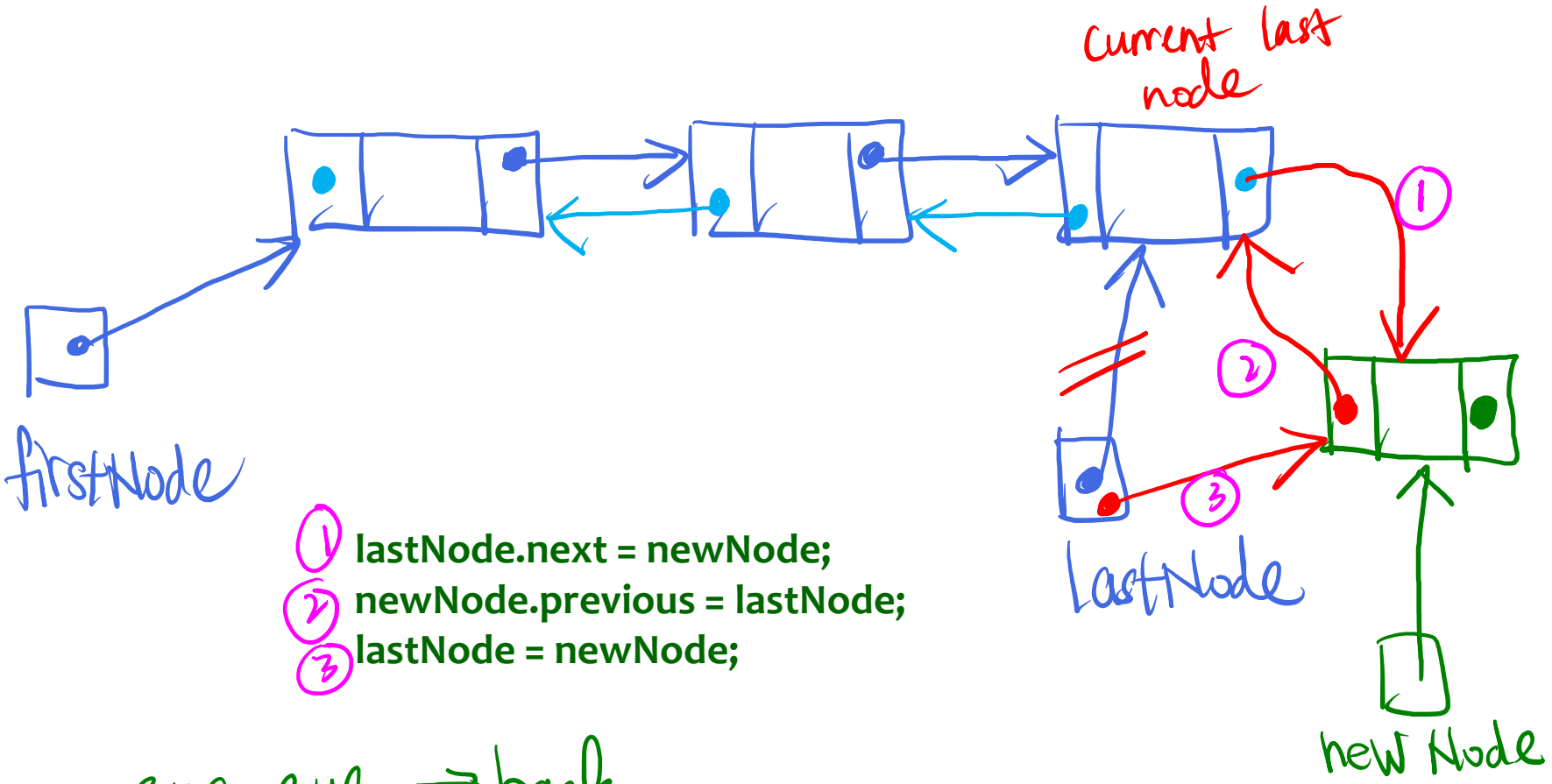


Fig. 34 A doubly linked chain with head and tail references

Creating a doubly linked node

```
private class Node {

  private T data;
  private Node next;
  private Node previous;

  private Node(T data) {
    this.data = data;
    this.next = null;
    this.previous = null;
  }

  private Node(T data, Node next, Node previous) {
    this.data = data;
    this.next = next;
    this.previous = previous;
  }
}
```

Adding a node at the back of a doubly linked queue



enqueue → back

Doubly Linked Implementation of Queue (4)

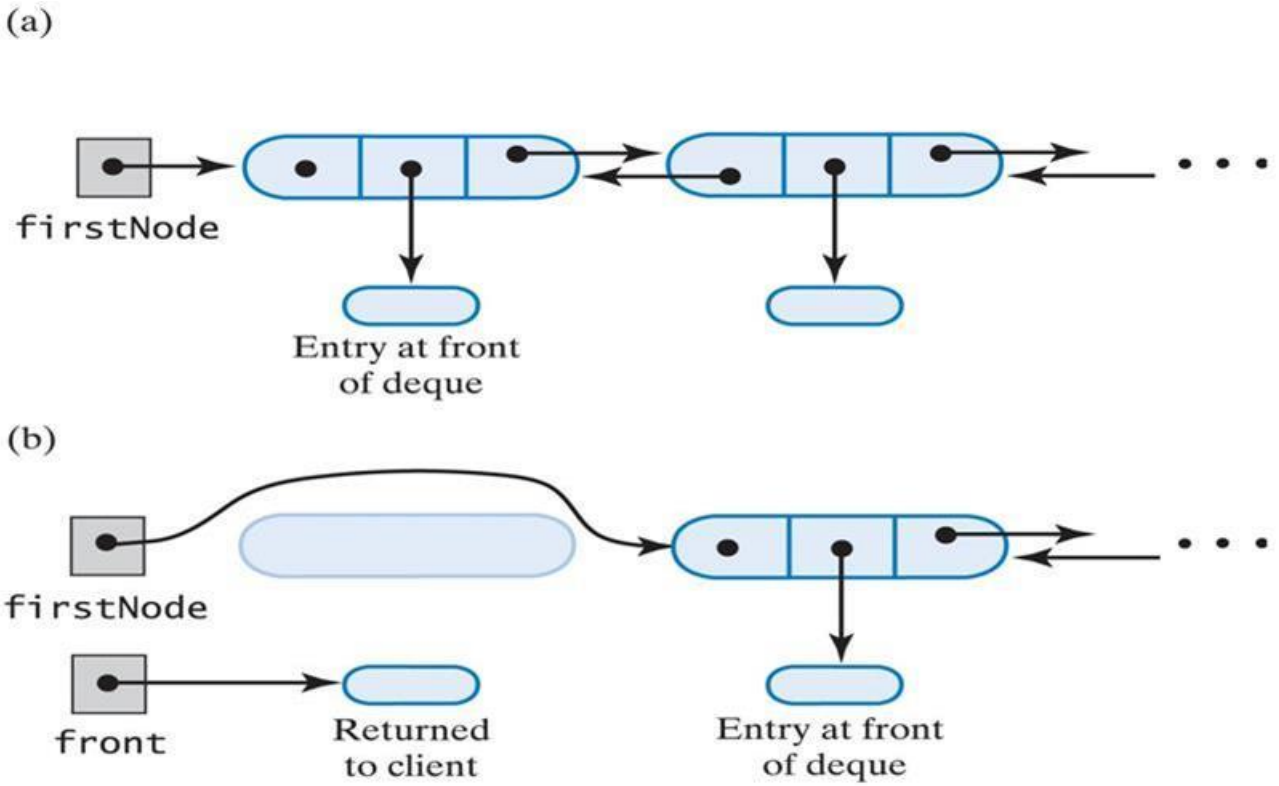
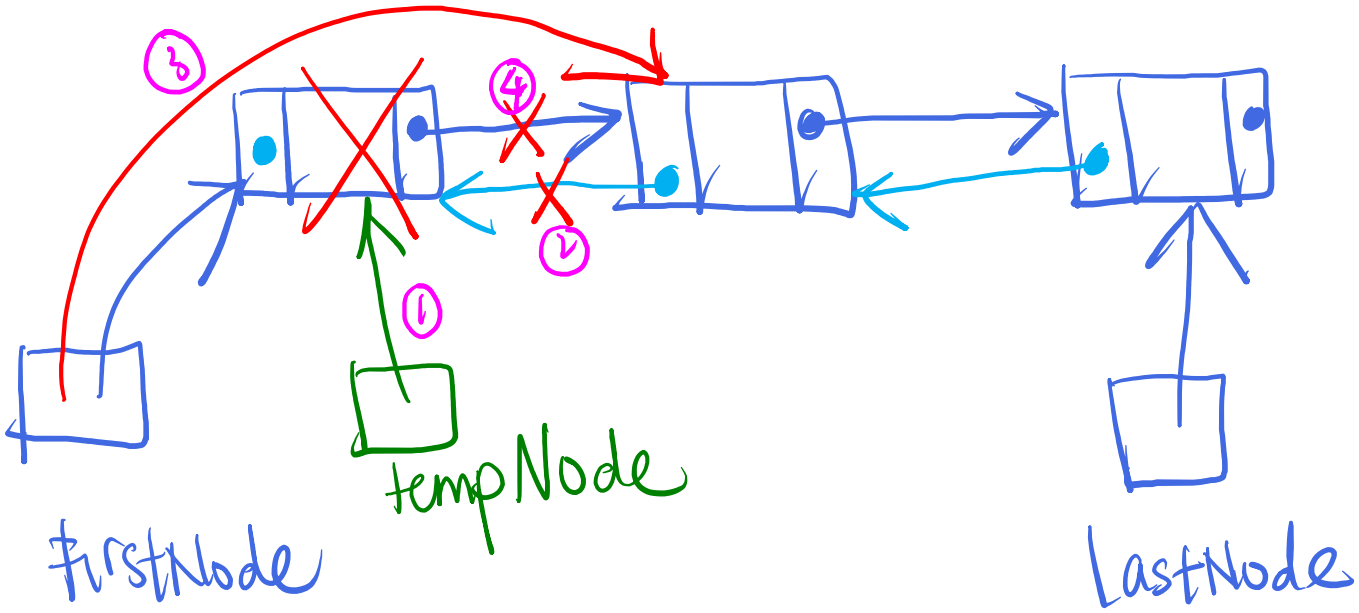


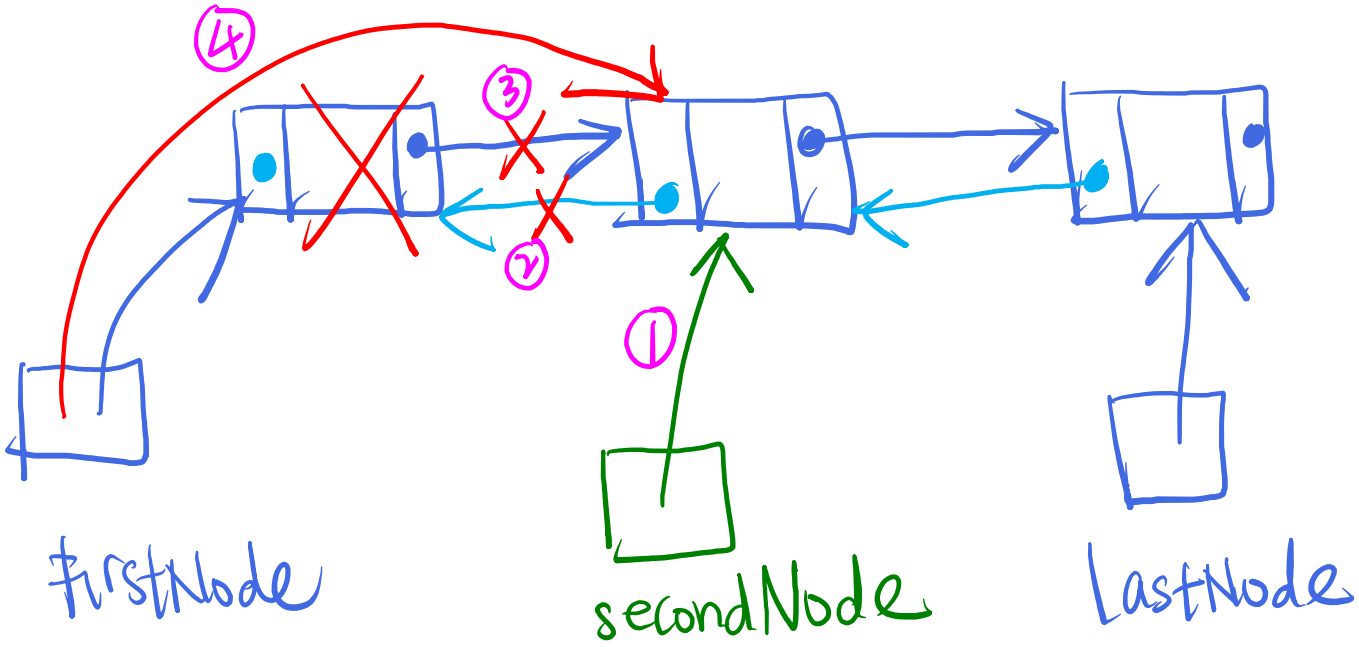
Fig. 36 : (a) a queue containing at least two entries; (b) after removing first node and obtaining reference to the queue's first entry.

Removing a node from the front of a doubly linked queue (method 1)



1. tempNode = firstNode;
2. firstNode.next.previous = null;
3. firstNode = firstNode.next;
4. tempNode.next = null; // this must be the last, cannot be before // step 3

Removing a node from the front of a doubly linked queue (method 2)



1. secondNode = firstNode.next;
2. secondNode.previous = null;
3. firstNode.next = null;
4. firstNode = secondNode;

Let's review

- What is a node?
- What is a linked list?
- What does a program need in order to access a linked list?
- What should the head reference contain if the linked list is empty?
- What should the **`next`** field of each node store?
- What should the **`next`** field of the last node store?
- What does it mean to traverse a linked list?
- How can you find the last node in a linked list?

CHECK YOUR UNDERSTANDING

Let us look at Past Year Questions !!!

PYQ (June 2023 Q3c) – Linked Stack

Question 3 (Continued)

c) Given a LinkedStack implementation in Figure 6, construct a **pop method** that allows an entry to be removed from the Stack.

```
public class LinkedStack<T> implements StackInterface<T> {

  private Node topNode;

  public LinkedStack() {
    topNode = null;
  }

  public void push(T newEntry) {
    Node newNode = new Node(newEntry, topNode);
    topNode = newNode;
  }

  public T peek() {
    T top = null;

    if (topNode != null) {
      top = topNode.data;
    }

    return top;
  }

  public T pop() {

   ...

  } //Linked Stack
```

Figure 6: LinkedStack implementation

(5 marks)

PYQ (Jan 2023 Q3c) – Linked Queue

Question 3 (Continued)

c) Given an array implementation of dequeue method in Figure 5, construct a *Linked implementation* of dequeue method.

```
public T dequeue() {
   T front = null;
   if (!isEmpty()) {
     front = array[frontIndex];
     for (int i = frontIndex; i < backIndex; ++i) {
       array[i] = array[i + 1];
     }
     backIndex--;
   }
   return front;
 }
```

Figure 5: Array implementation of dequeue

(9 marks)

PYQ (May 2022 Q3b) – Linked List

b) Construct an **add(int newPosition, T newEntry)** method to insert an element into List using **Linked** implementation. Assume that the Node class has been defined. (10 marks)

Review of learning outcomes

You should now be able to

- Describe what are linked structures.
- Implement the ADT list, stack and queue using linked implementation.
- Evaluate the advantages and disadvantages of a linked implementation of linear collections.
- Describe and implement variations of linked structures.

To Do

- Review the slides and source code for this chapter.
- Read up the relevant portions of the recommended text.
- Compete the remaining practical questions for this chapter.

References

1. Carrano, F. M., 2019, Data Structures and Abstractions with Java, 5th edn, Pearson

2. Liang, Y.D., 2018. Introduction to Java Programming and Data Structures.11th ed. United Kingdom: Pearson