BACS2063 Data Structures and Algorithms

Linked Implementations of Collection ADTs

Chapter 5

Limitations of using arrays

- Overhead due to shifting during add and remove operation.
- 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.

Learning Outcomes

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

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

Analogy for Linked Data

- Consider the analogy of desks in a classroom
  - Placed in classroom as needed
  - Each desk has a unique id, the “address”
  - The desks are linked by keeping the address of another chair
  - We have a chain of chairs

Analogy for Linked Data

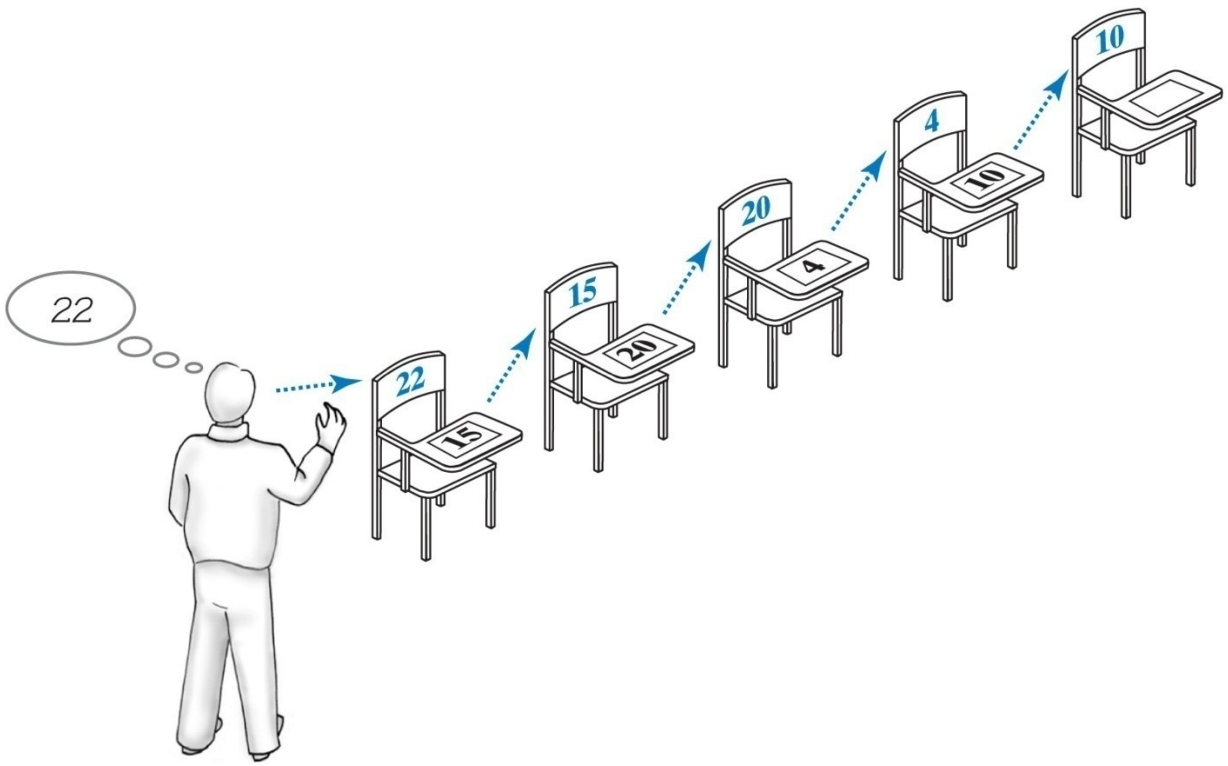


Fig. 1 A chain of 5 desks.

Forming a Chain by Adding to Beginning

- First desk placed in room
  - Blank desk top, no links
  - Address of the first chair given to teacher

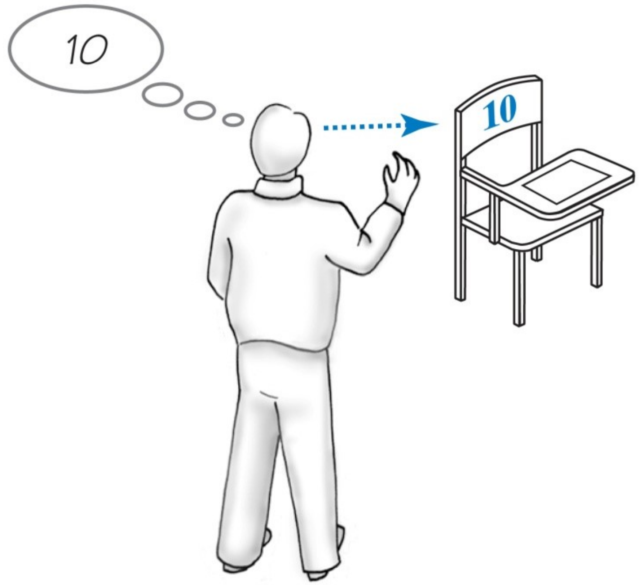


Fig. 2 One desk in the room.

Forming a Chain by Adding to Beginning

- Second student arrives, takes a desk
  - Address of first desk placed on new desk
  - Instructor "remembers" address of new desk

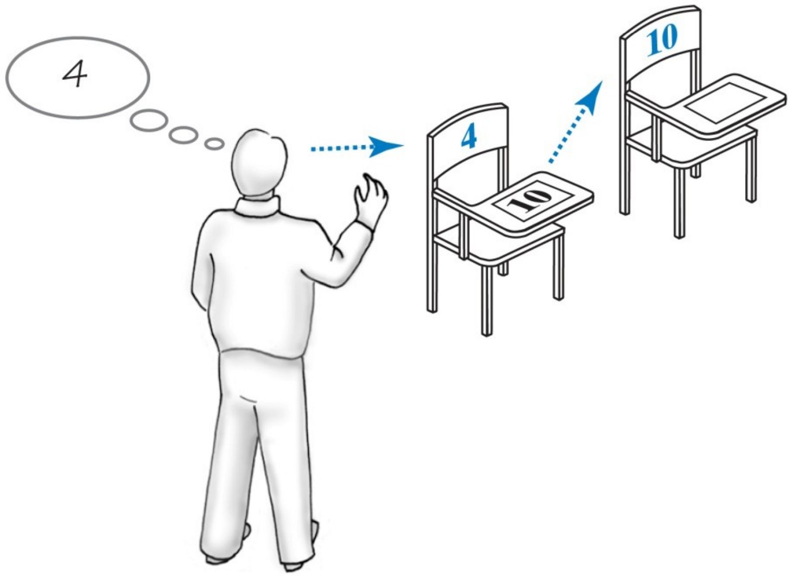


Fig. 3 Two linked desks

Forming a Chain by Adding to Beginning

- Third desk arrives
  - New desk gets address of second desk
  - Instructor remembers address of new desk

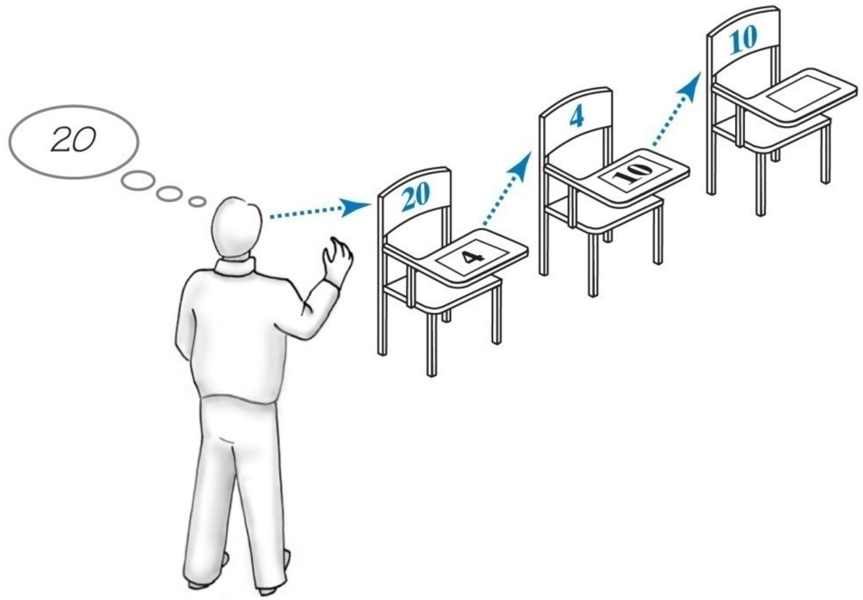


Fig. 4 Three linked desks, newest desk first.

Forming a Chain by Adding to End

- This time the first student is always at the beginning of the chain
  - Instructor only remembers first address
- The address of a new desk is placed on the previous desk (at end of chain)
  - End of chain found by following links
  - Newest desk does not have a pointer address to any other desk

Forming a Chain by Adding to End

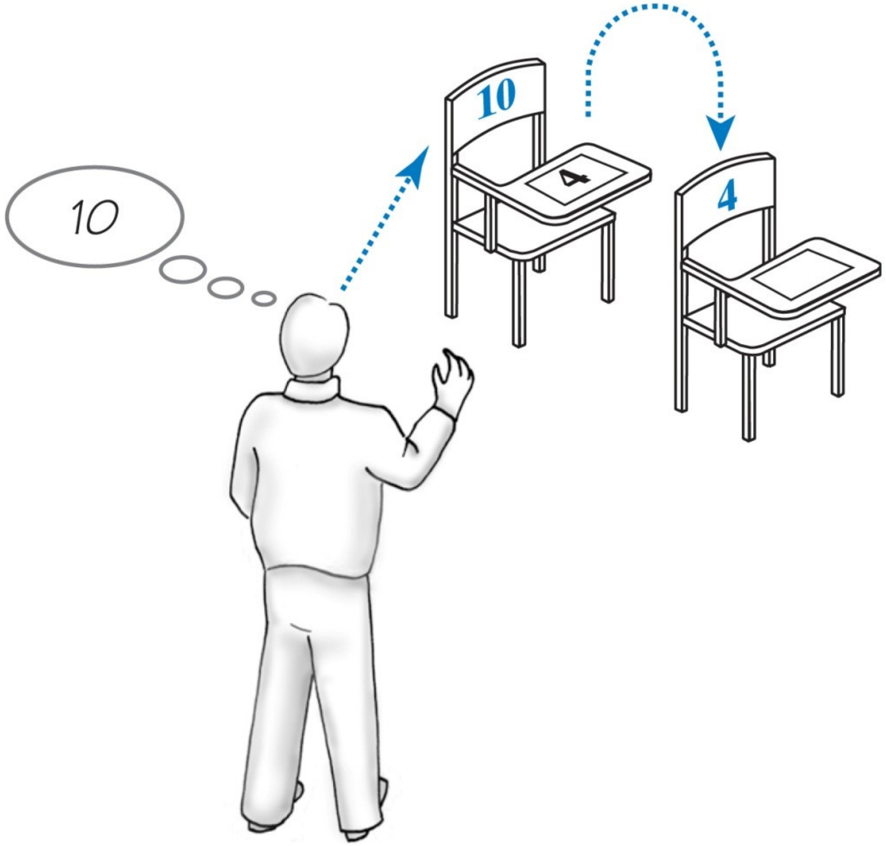


Fig. 5 Two linked desks, newest desk last.

Forming a Chain by Adding to End

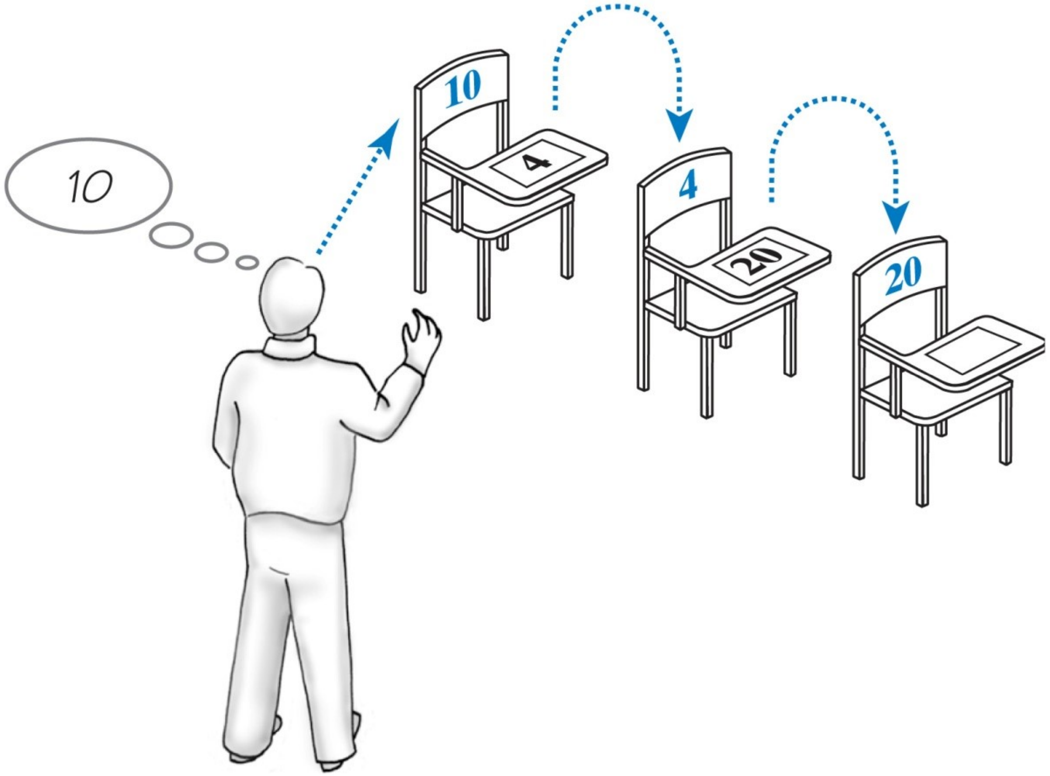


Fig. 6 Three linked desks, newest desk last.

Forming a Chain by Adding to End

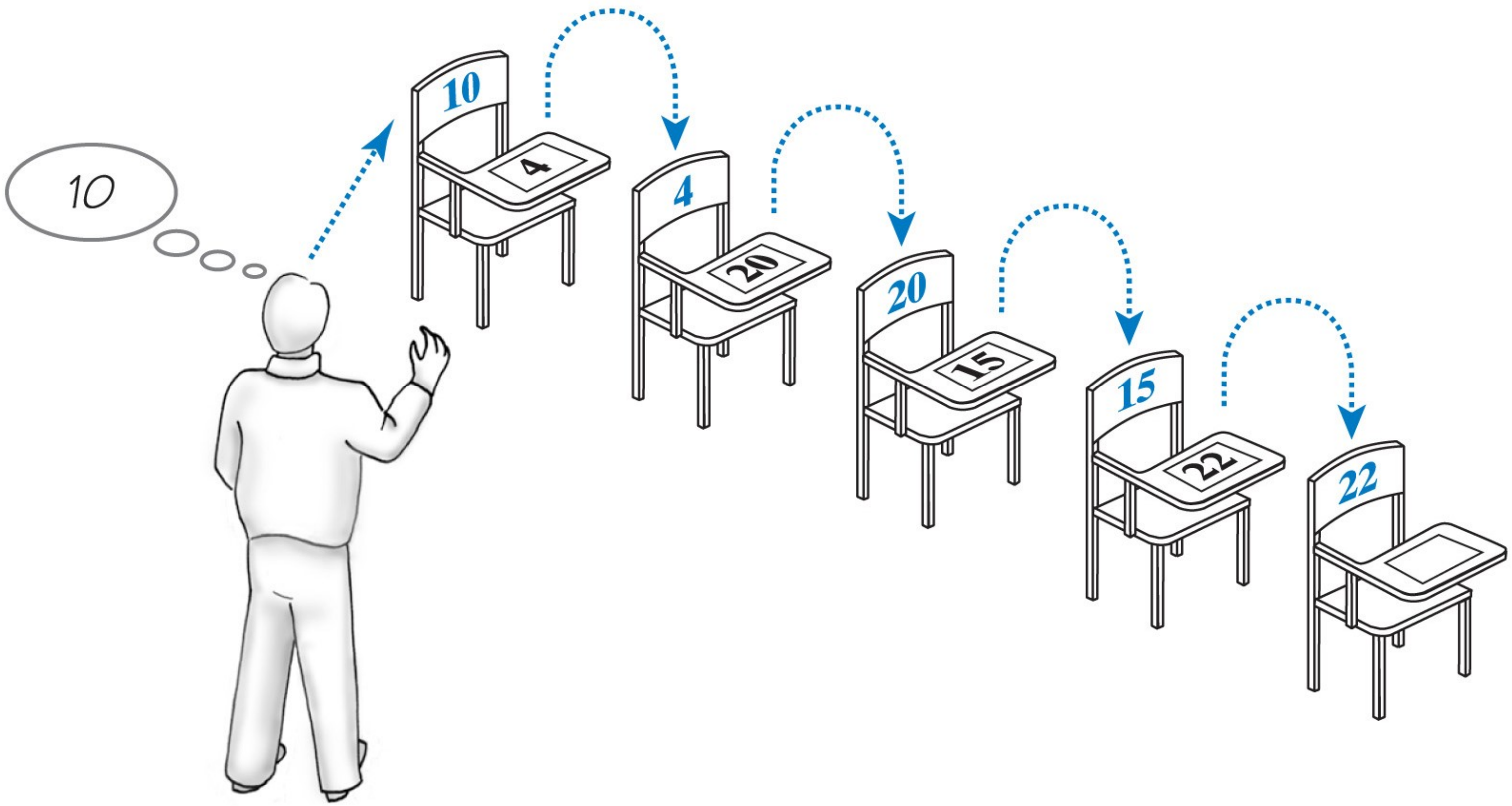


Fig. 7 Five linked desks, newest desk last.

Forming Chain by Adding at Various Positions

- New entries may be placed somewhere in the chain, not necessarily at the end
- Possibilities for placement of a new desk
  – Before all current desks
  – Between two existing desks
  – After all current desks

Forming Chain by Adding at Various Positions

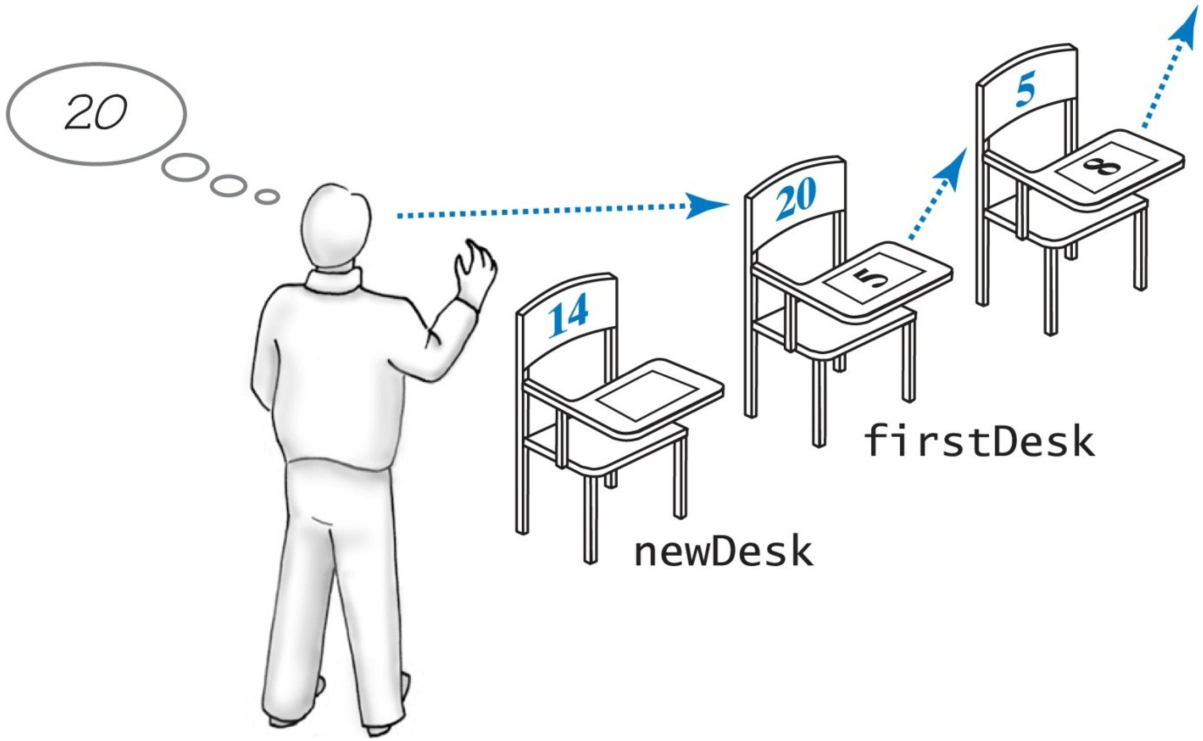


Fig. 8 Chain of desks prior to adding a new desk to beginning of the chain

Forming Chain by Adding at Various Positions

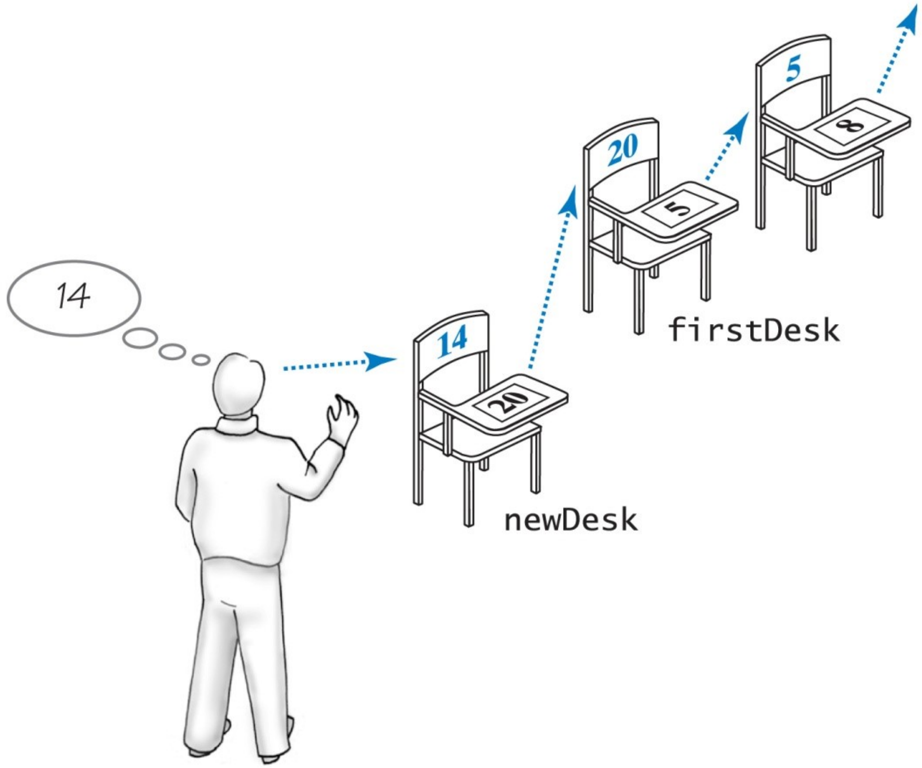


Fig. 9 Addition of a new desk to beginning of a chain of desks

Forming Chain by Adding at Various Positions

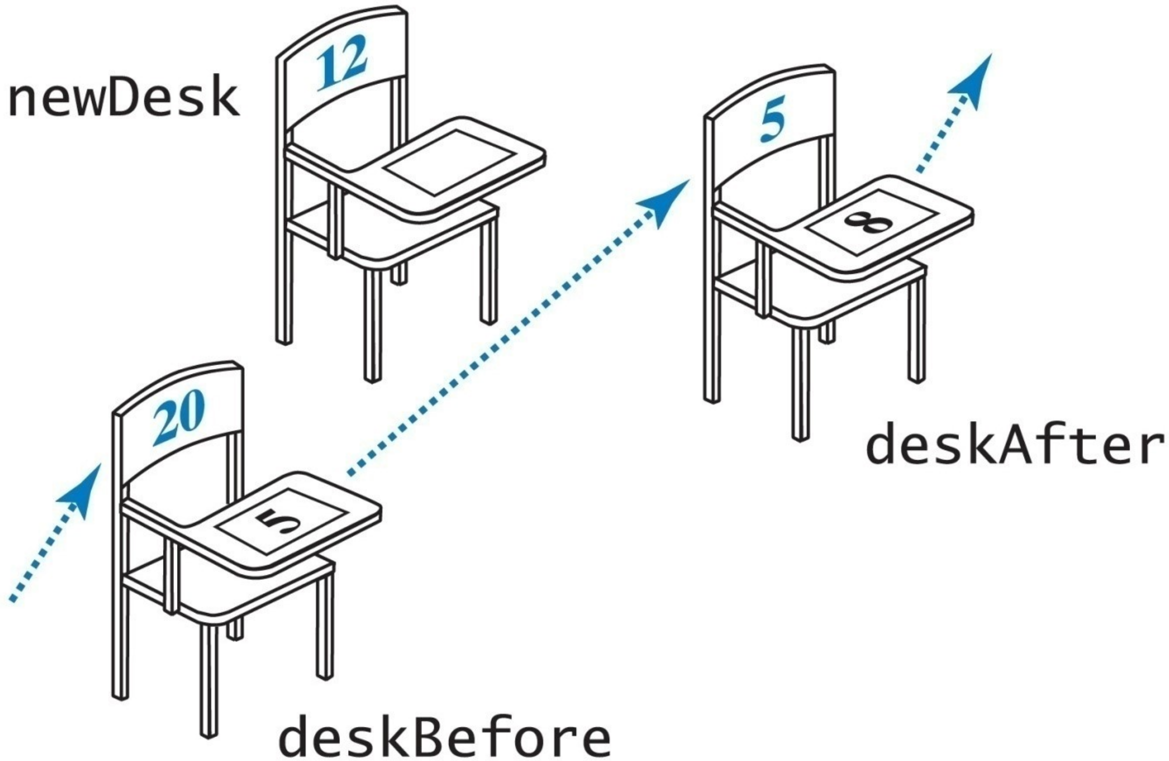


Fig. 10 Two consecutive desks within a chain prior to adding new desk between

Forming Chain by Adding at Various Positions

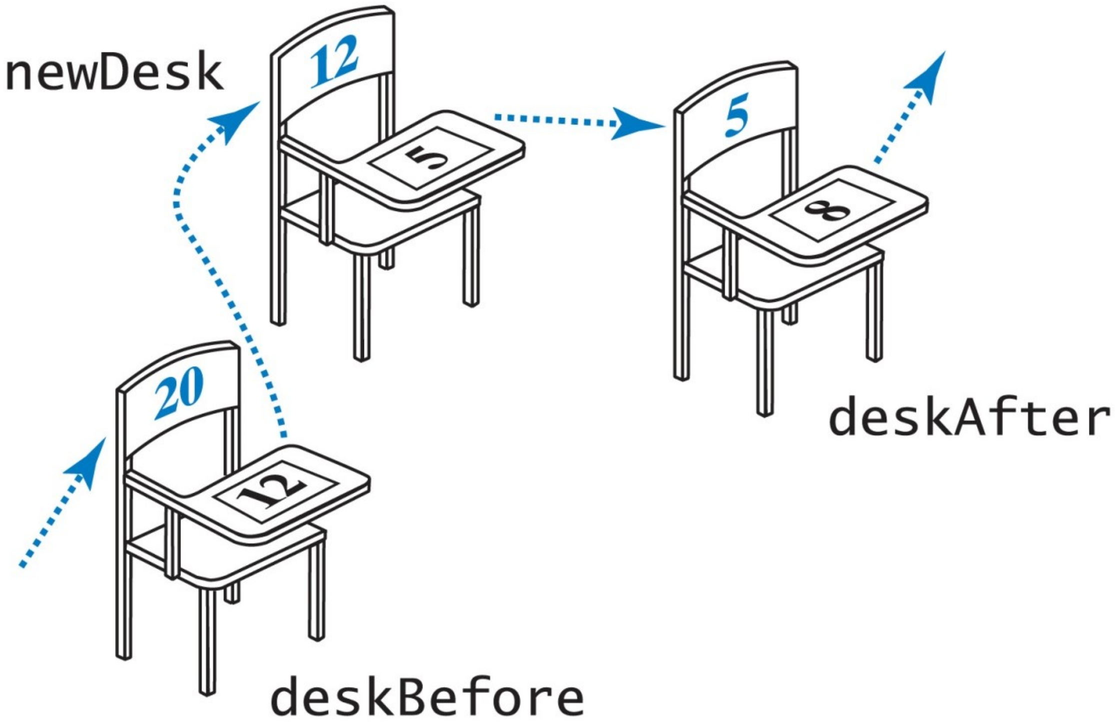
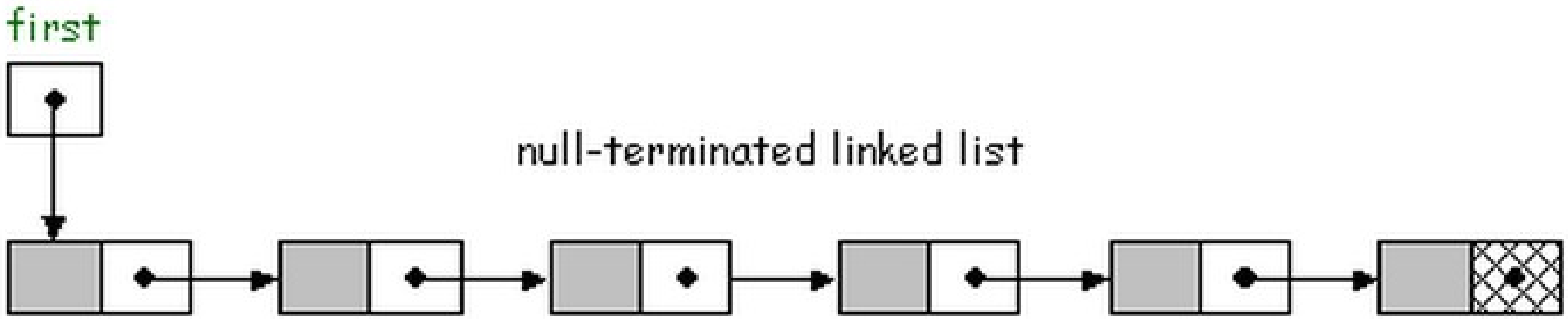


Fig. 11 Addition of a new desk between two other desks.

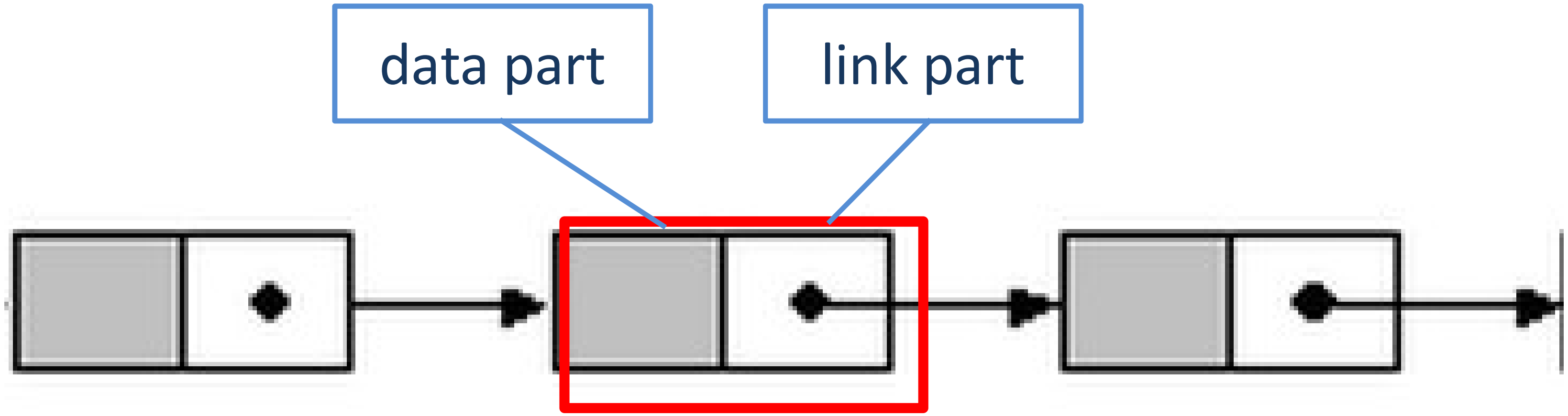
Linked Lists

- 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.



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

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`**

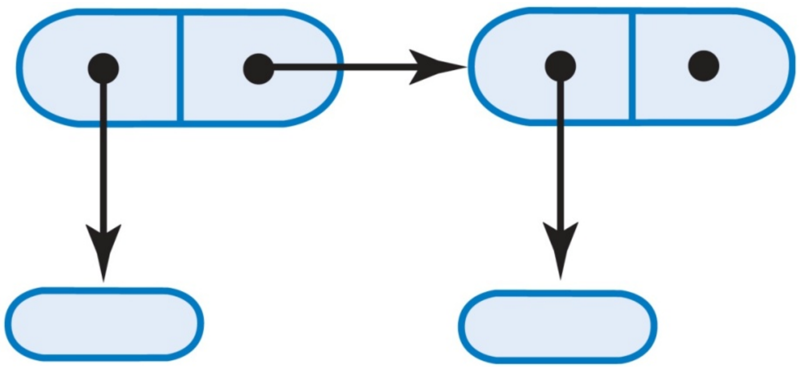
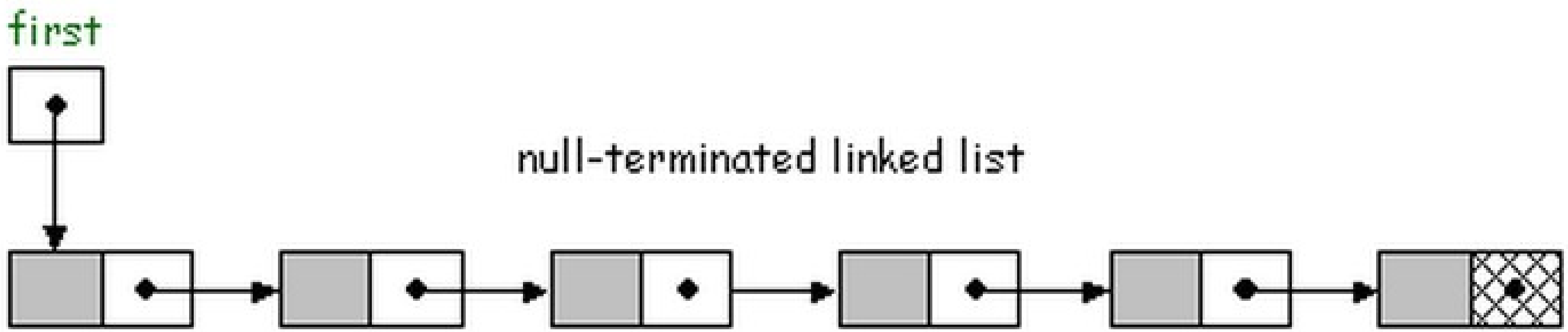


Fig. 12 Two linked nodes that each reference object data

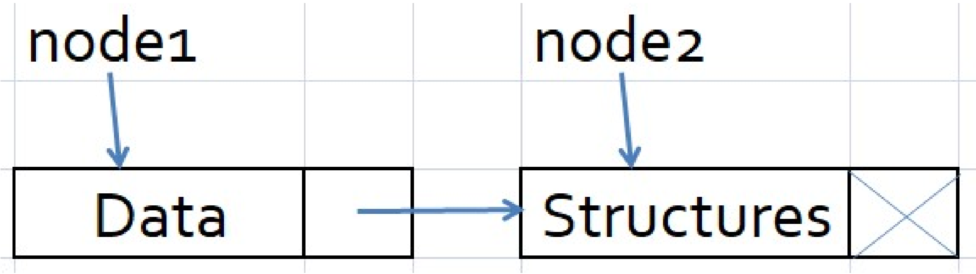
3 Rules for a Linear Linked List

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.



Exercise 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 2

a. Add a new method in the **`Node`** class as follows:

Note: Instead of using the set and get methods, directly access the data fields of the node object.

- ❑ build a linked list with the same 4 nodes as Exercise 1.
- ❑ Include a `for` loop to traverse and display the contents of the linked list.

b. Write a driver program to test the method you have written.

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?

Linked Implementation of List ADT

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 <u>inner class</u>

```
public class LList<T> implements ListInterface<T> {
  private Node firstNode; // reference to first node
  private int  length;    // 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;
    }
  }
}
```

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:
  - **`LList.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)

Methods of `LList`

- Methods **add**
- Method **display**
- Method **isEmpty**
- Method **remove**
- Method **replace**
- Method **getEntry**
- Method **contains**
- Method **isFull**
  - Always returns **`false`** in this context

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.

Adding to the End of the List

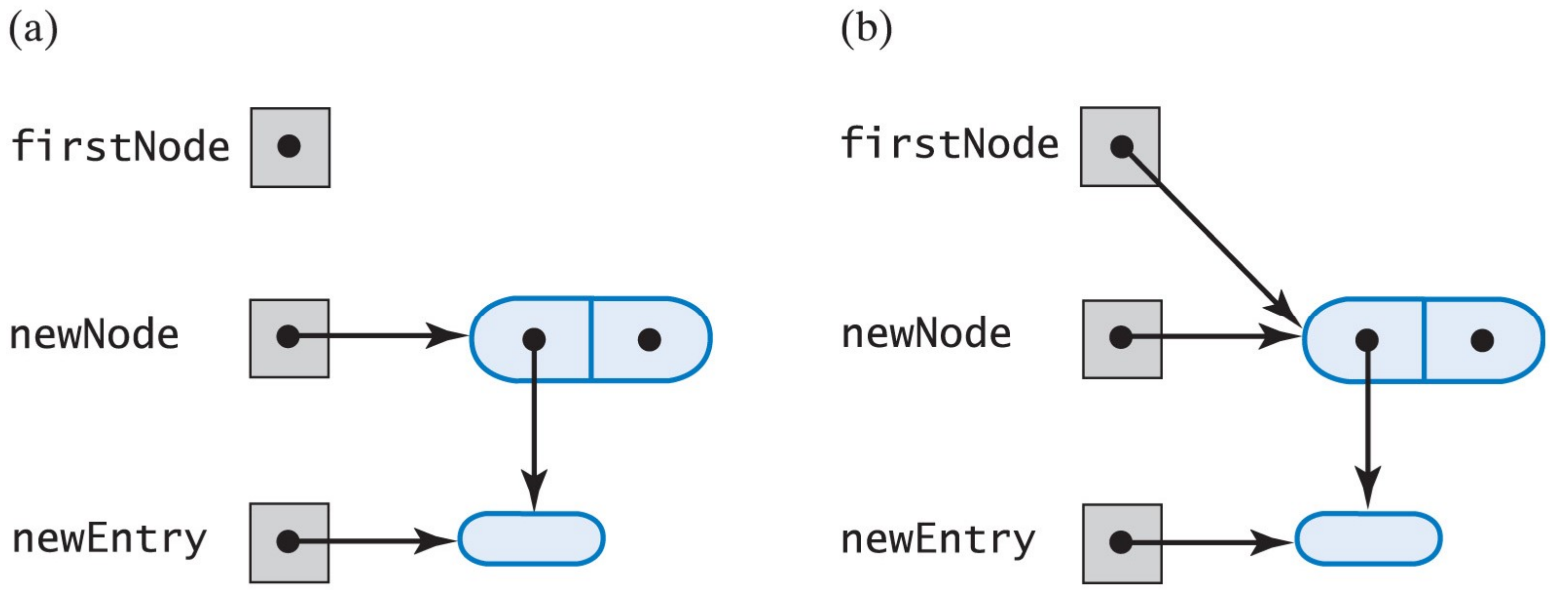


Fig. 13 (a) An empty list and a new node;
(b) after adding a new node to a list that was empty

Adding to the End of the List

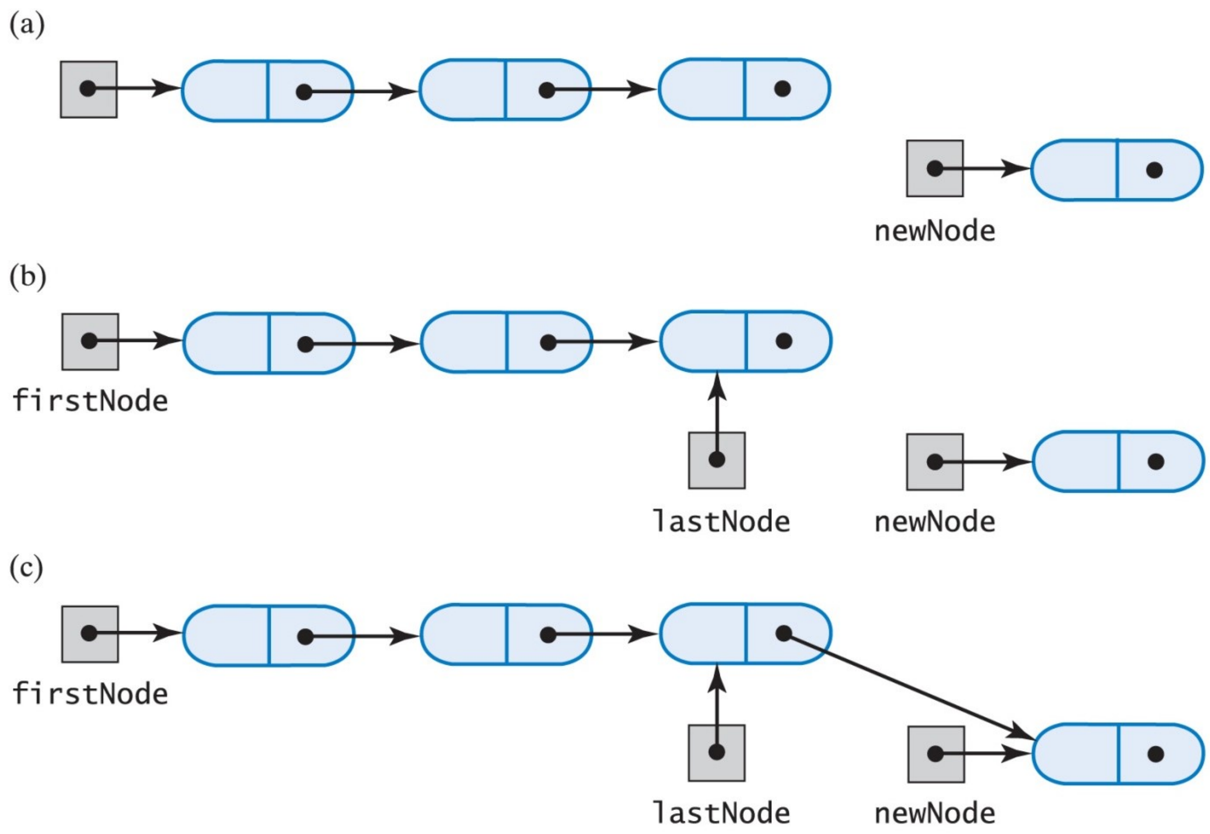


Fig. 14 A chain of nodes (a) just prior to adding a node at the end; (b) just after adding a node at the end.

Adding at Beginning of the List

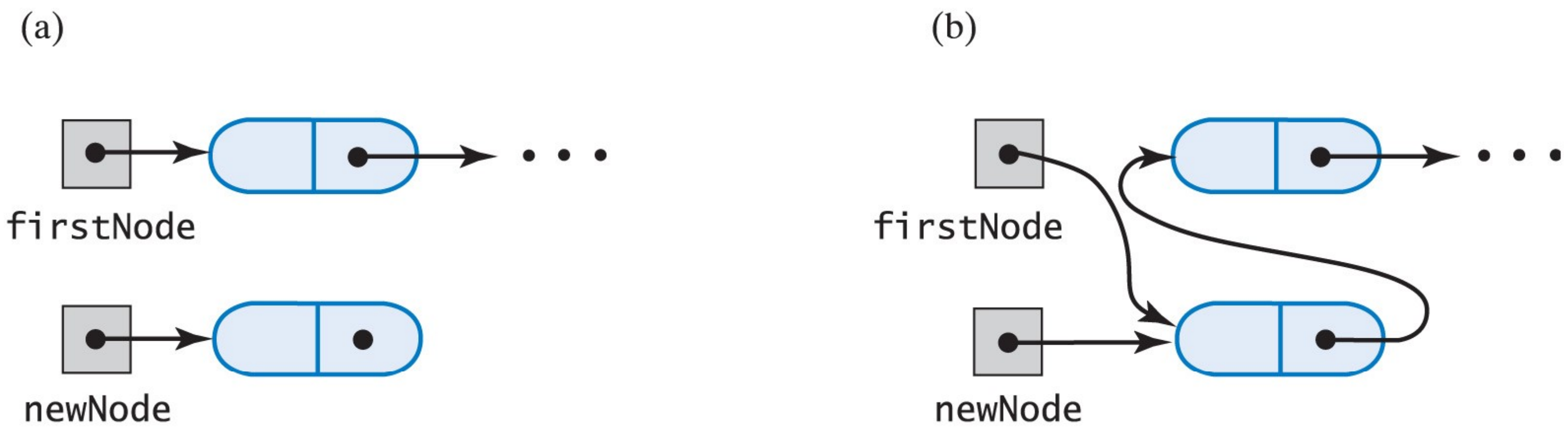


Fig. 15 A chain of nodes (a) prior to adding a node at the beginning; (b) after adding a node at the beginning.

Adding Within (in “middle of ”) the List

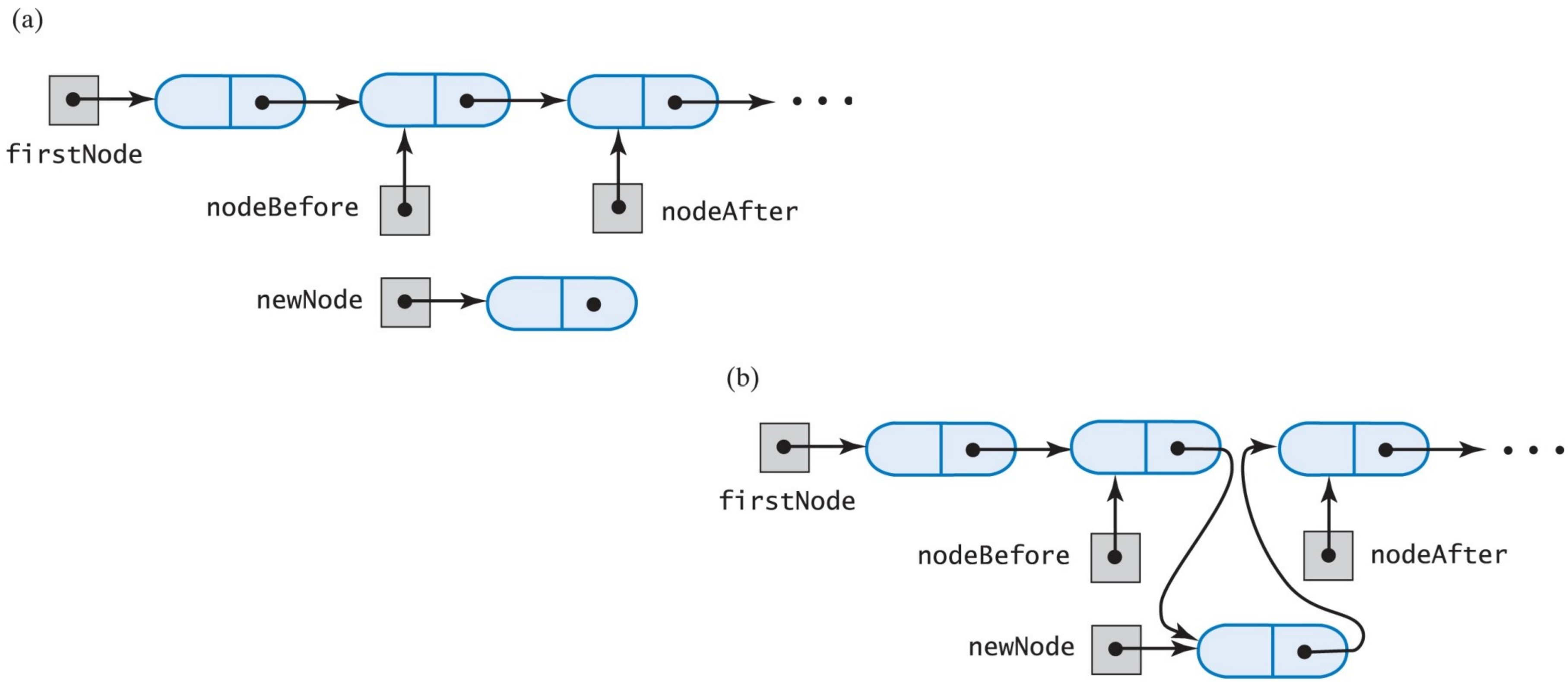


Fig. 16 A chain of nodes (a) prior to adding node between adjacent nodes; (b) after adding node between adjacent nodes

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*.

Removing an Item from a Linked Chain

- Possible cases
  1. Remove from beginning (first node) of the chain
  2. Remove from middle of the chain
  3. Remove from the end (last node) of the chain
- Analogy of chain of desks illustrates in following slides

Removing 1st node - an analogy

Case 1

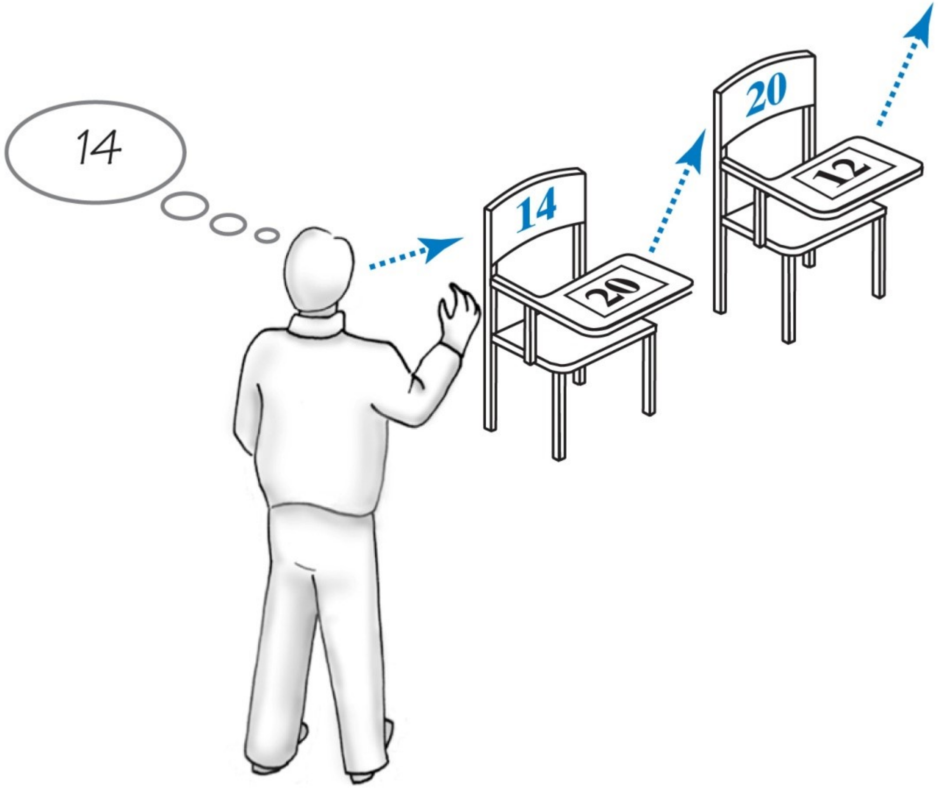


Fig. 17 A chain of desks just prior to removing its first desk

Removing 1st node - an analogy

Case 1

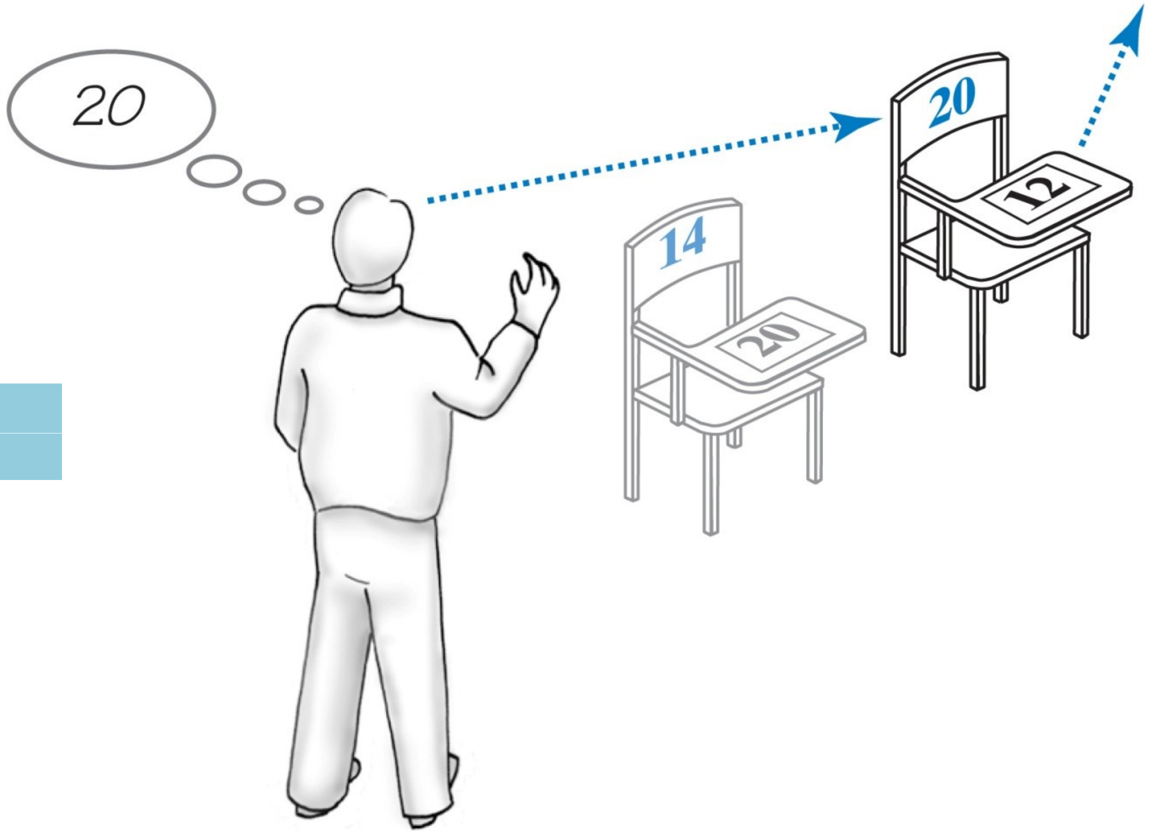


Fig. 18 A chain of desks just after removing its first desk.

Removing a middle node - an analogy

Case 2

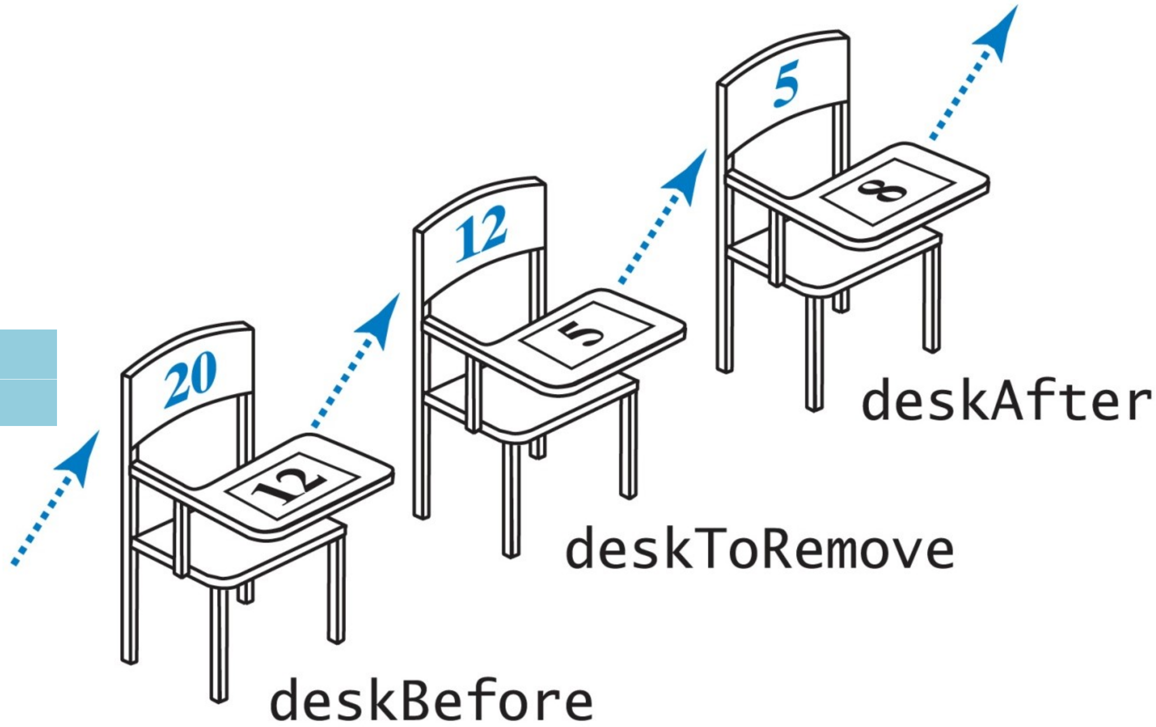


Fig.19 A chain of desks just prior to removing a desk between two other desks

Removing a middle node - an analogy

Case 2

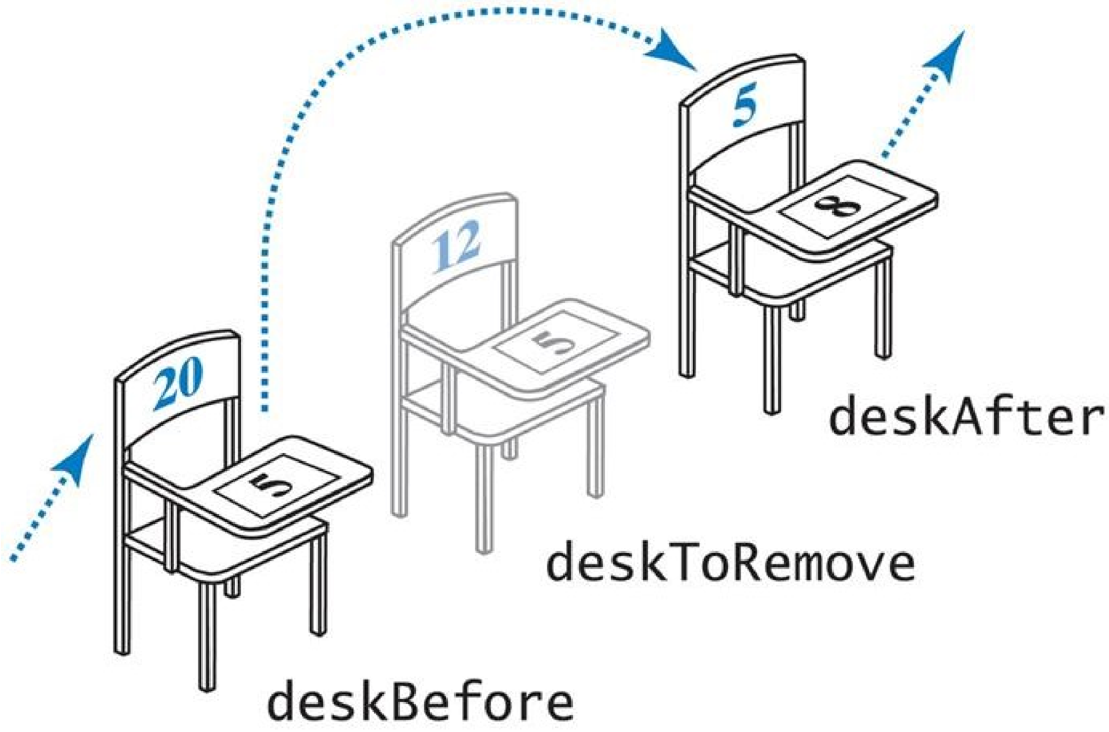


Fig. 20 A chain of desks just after removing a desk between two other desks

Removing the last node - an analogy

Case 3

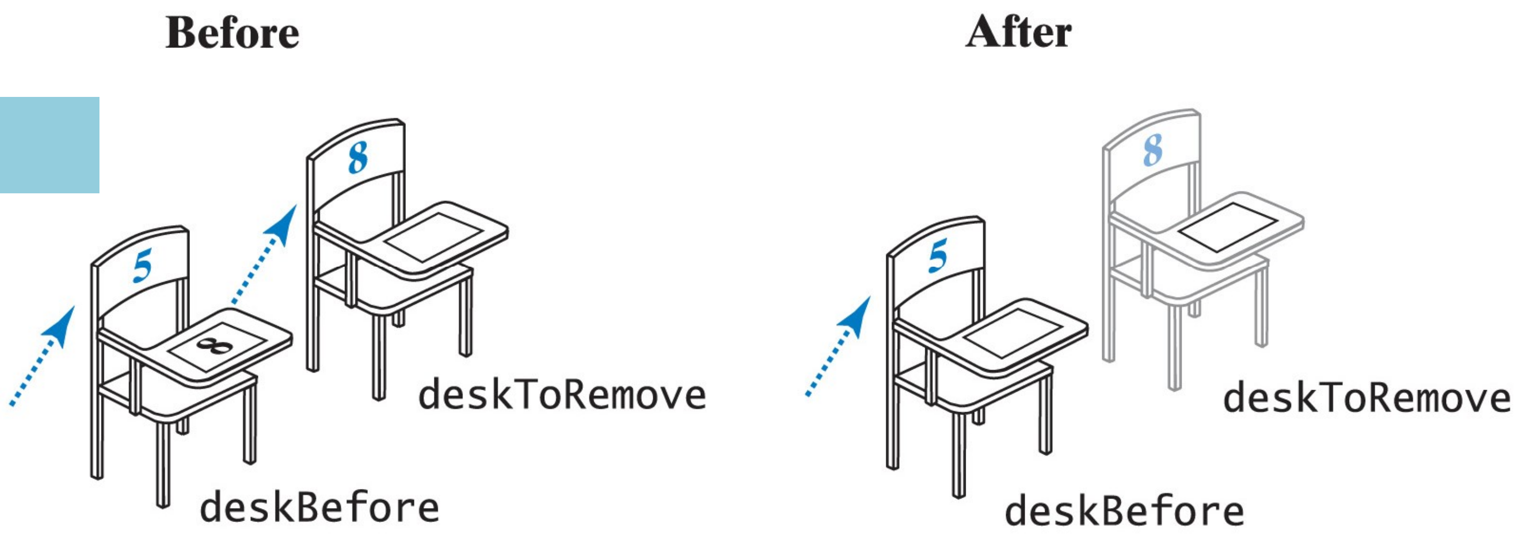


Fig. 21 Before and after removing the last desk from a chain

`remove()` - removing first node

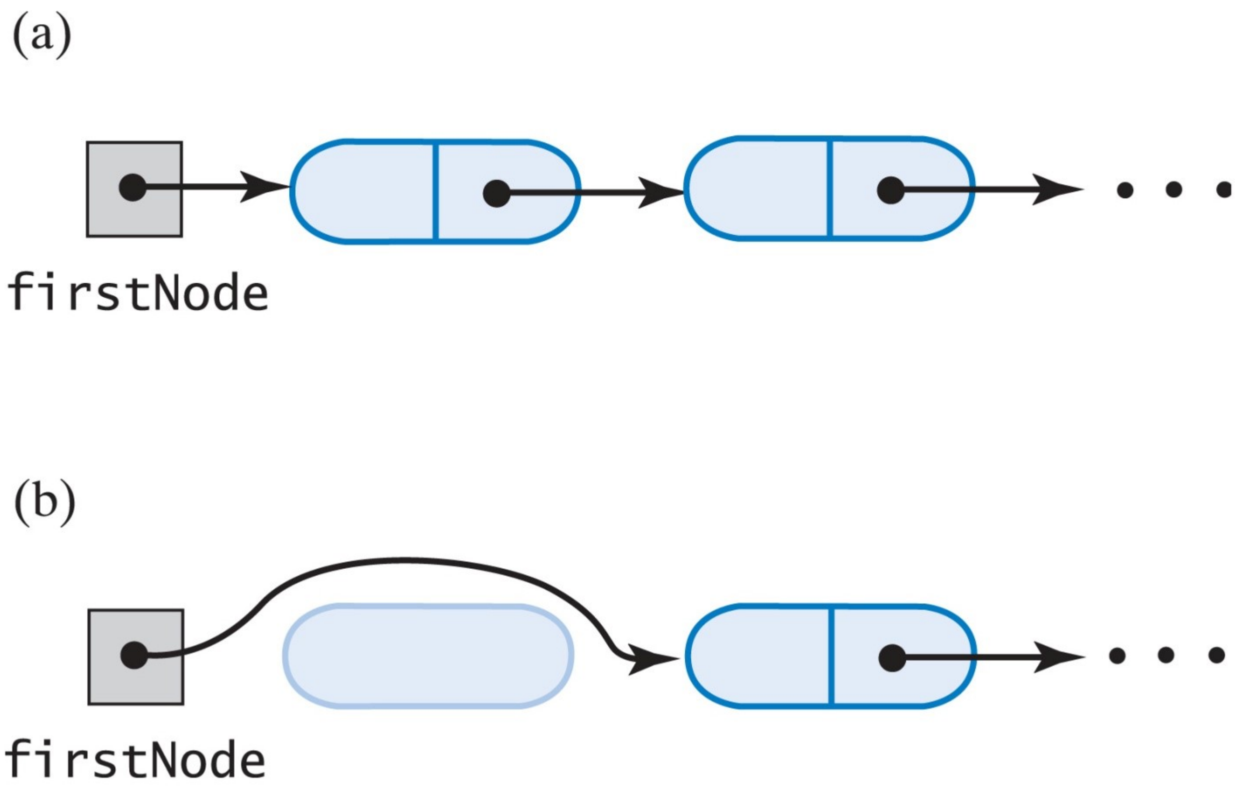


Fig. 22 A chain of nodes (a) prior to removing first node;
(b) after removing the first node

`remove()` - removing interior node

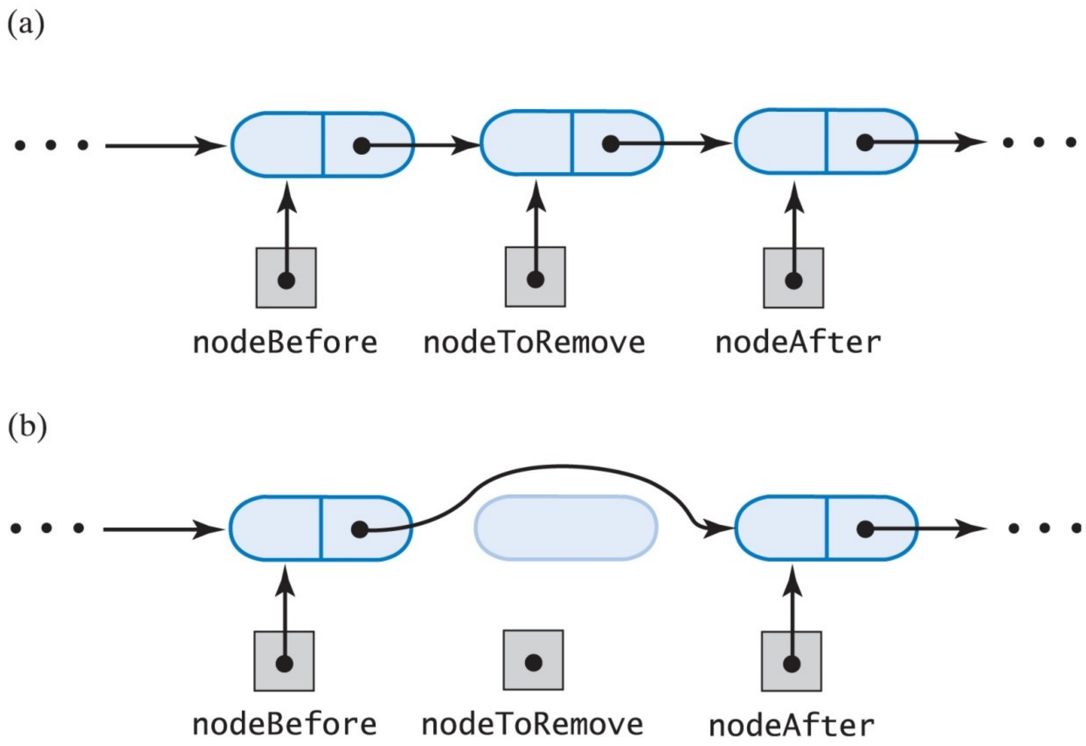


Fig. 23 A chain of nodes (a) prior to removing interior node; (b) after removing interior node

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`**
  - **`LList.java`**
- **`Chapter5\entity\`**
  - **`Runner.java`**
- **`Chapter5\client\`**
  - **`Registration.java`**

Efficiency of Implementations of ADT List

- For array-based implementation - **ArrList**
  - Add to end of list **O(1)**
  - Add to list at given position **O(n)**
- For linked implementation - **LList**
  - 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

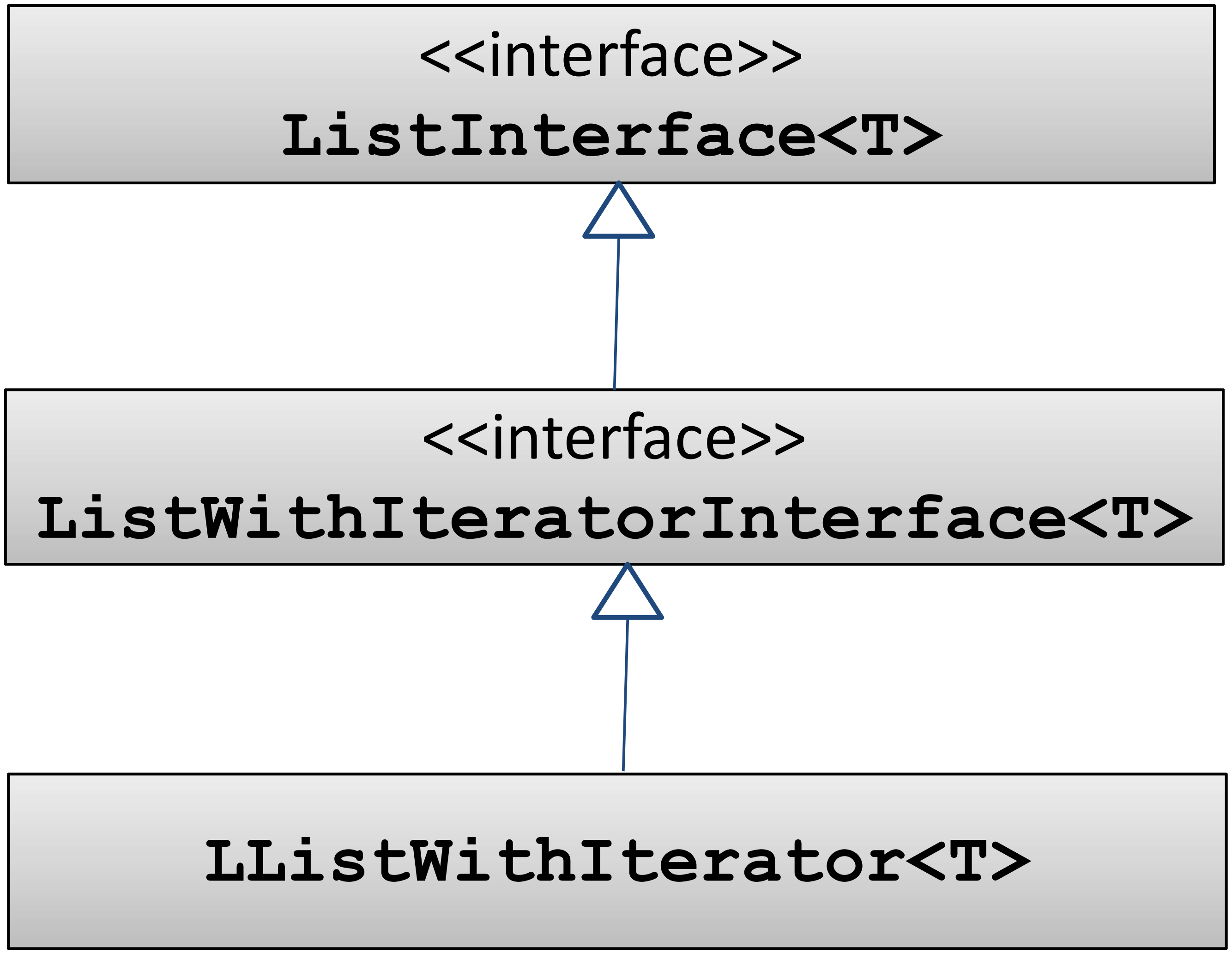
Note: Assuming a fixed-size array, i.e. that the `doubleArray()` method is never invoked.

Linked List with Iterator Implementation

`Chapter5\adt\`

- **`ListWithIteratorInterface.java`**:
  - An interface that extends the interface **`ListInterface`**.
  - Contains the abstract method **`getIterator()`** which returns an iterator to the list
- **`LListWithIterator.java`**
  - A class that implements the interface **`ListWithIteratorInterface`**

UML Class Diagram for Example



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

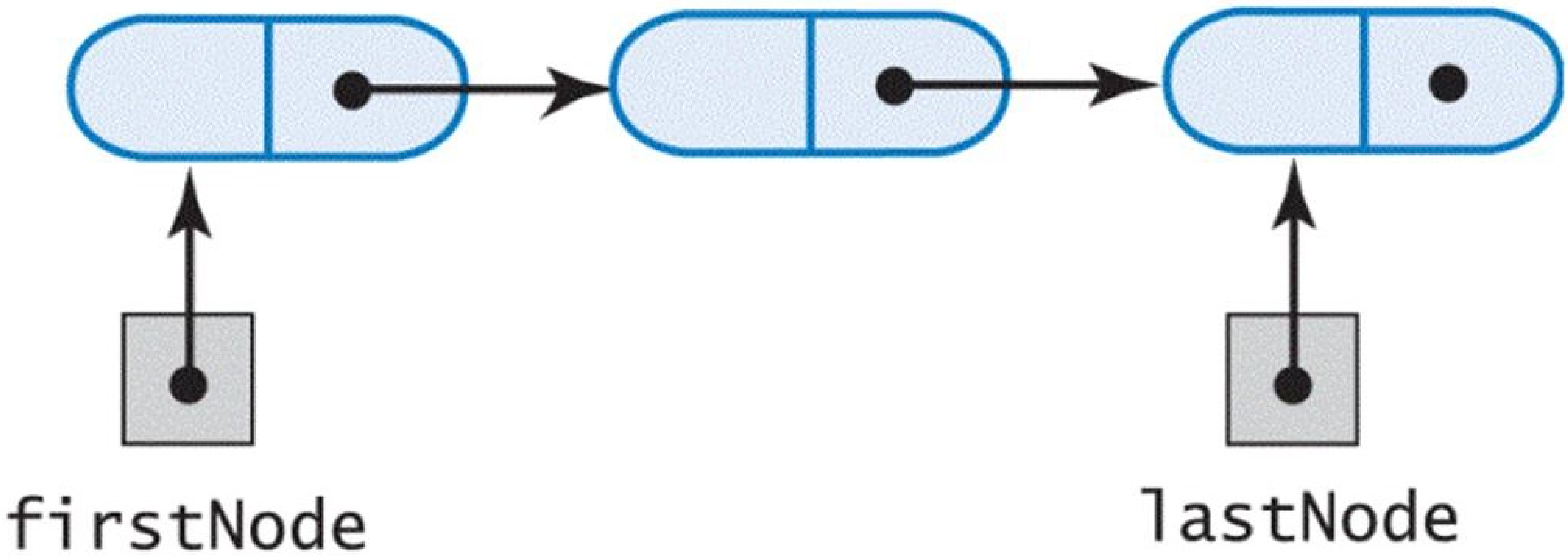


Fig. 24 A linked chain with a head and tail reference.

Tail References

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

```
private Node firstNode;
private Node lastNode;
private int  length;
```

Exercise 3

Examine the implementation of the class **`LList`** 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;
   length = 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

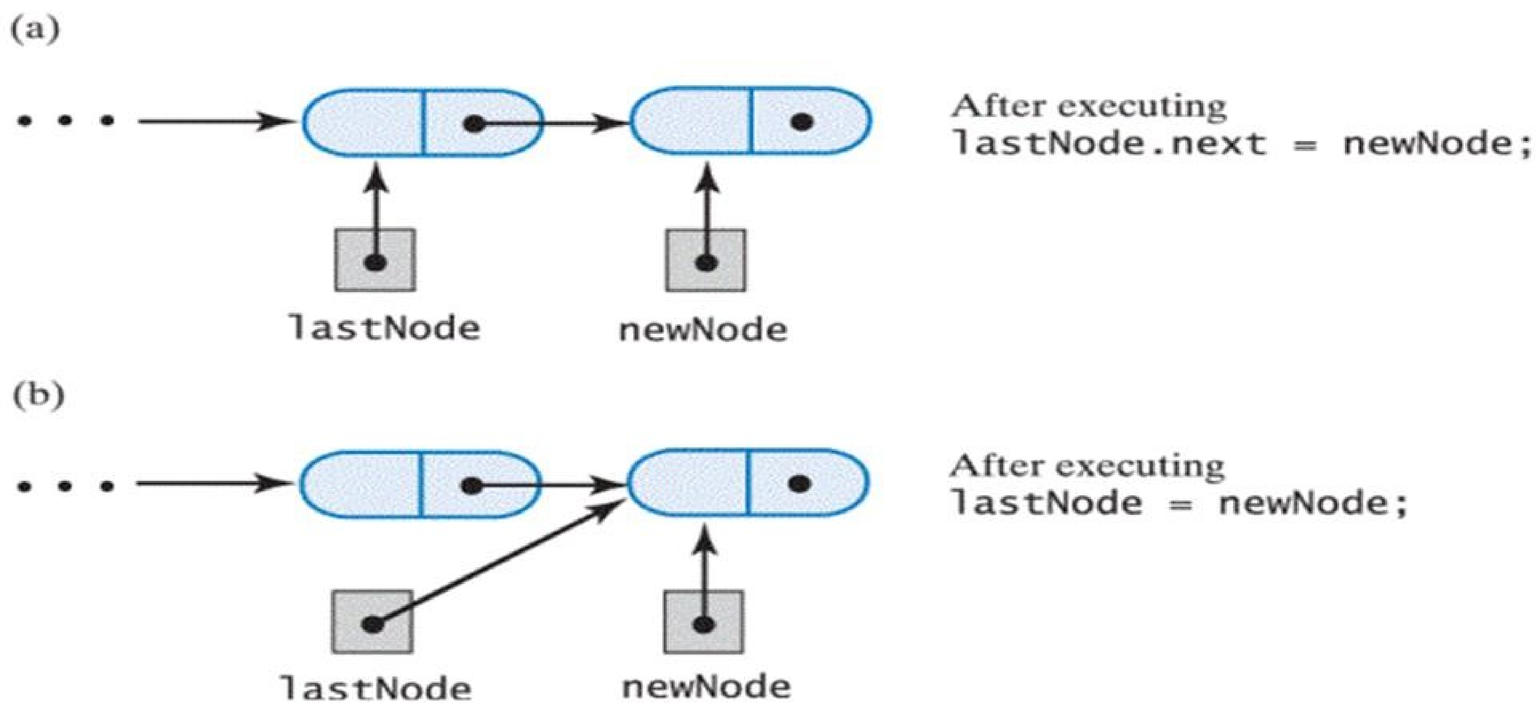


Fig. 25 Adding a node to the end of a nonempty chain that has a tail reference

Tail References

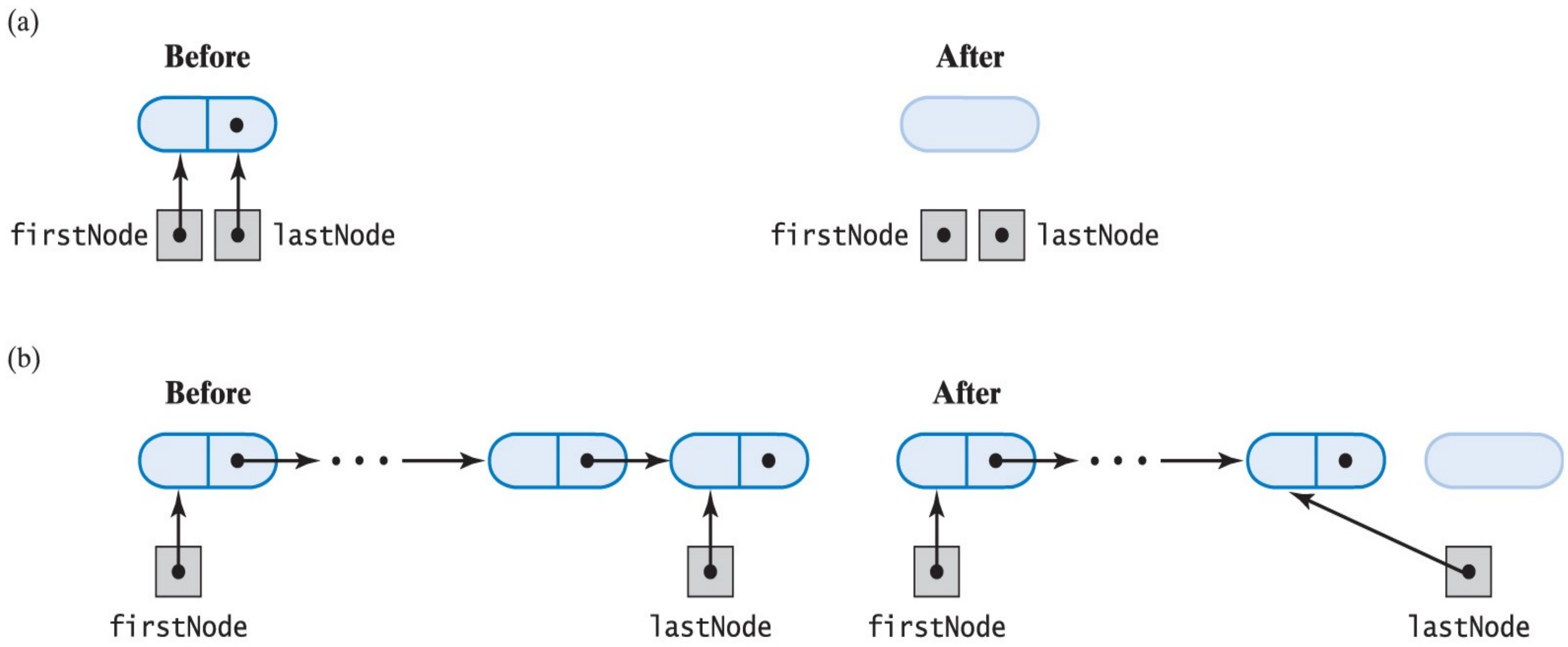


Fig. 26 Removing a node from a chain that has both head and tail references when the chain contains (a) one node; (b) more than one node

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

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

Exercise 4

Compare & contrast *singly* linked list with *doubly* linked list in terms of:

- The overall linked list structure
- The declaration of the class Node
- Advantages and disadvantages of each type of linked list

Linked Implementation of Stack ADT

A Linked Implementation

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

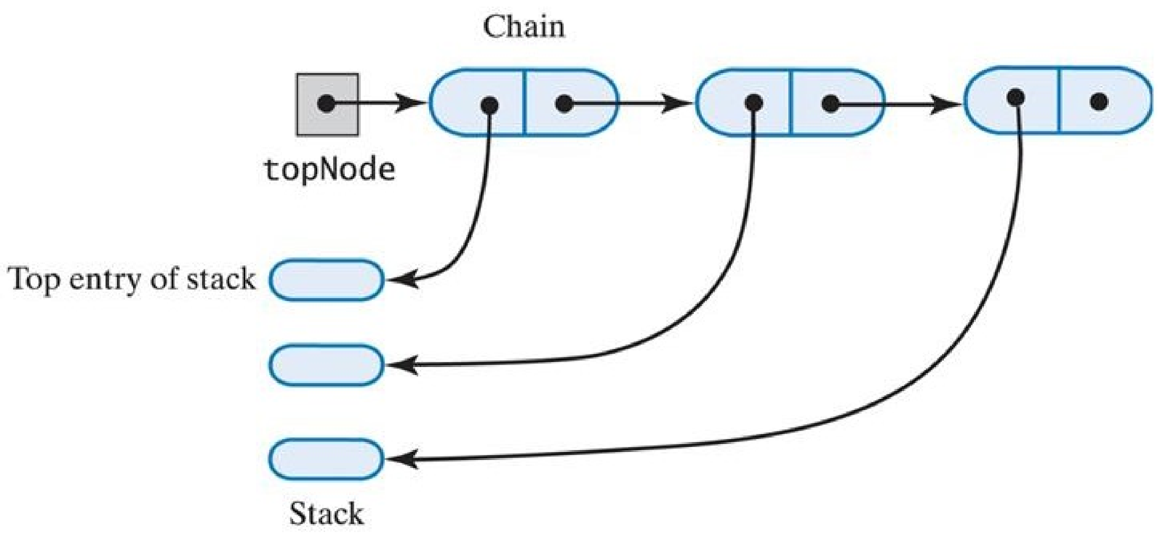


Fig. 27 A chain of linked nodes that implements a stack.

Pushing an entry

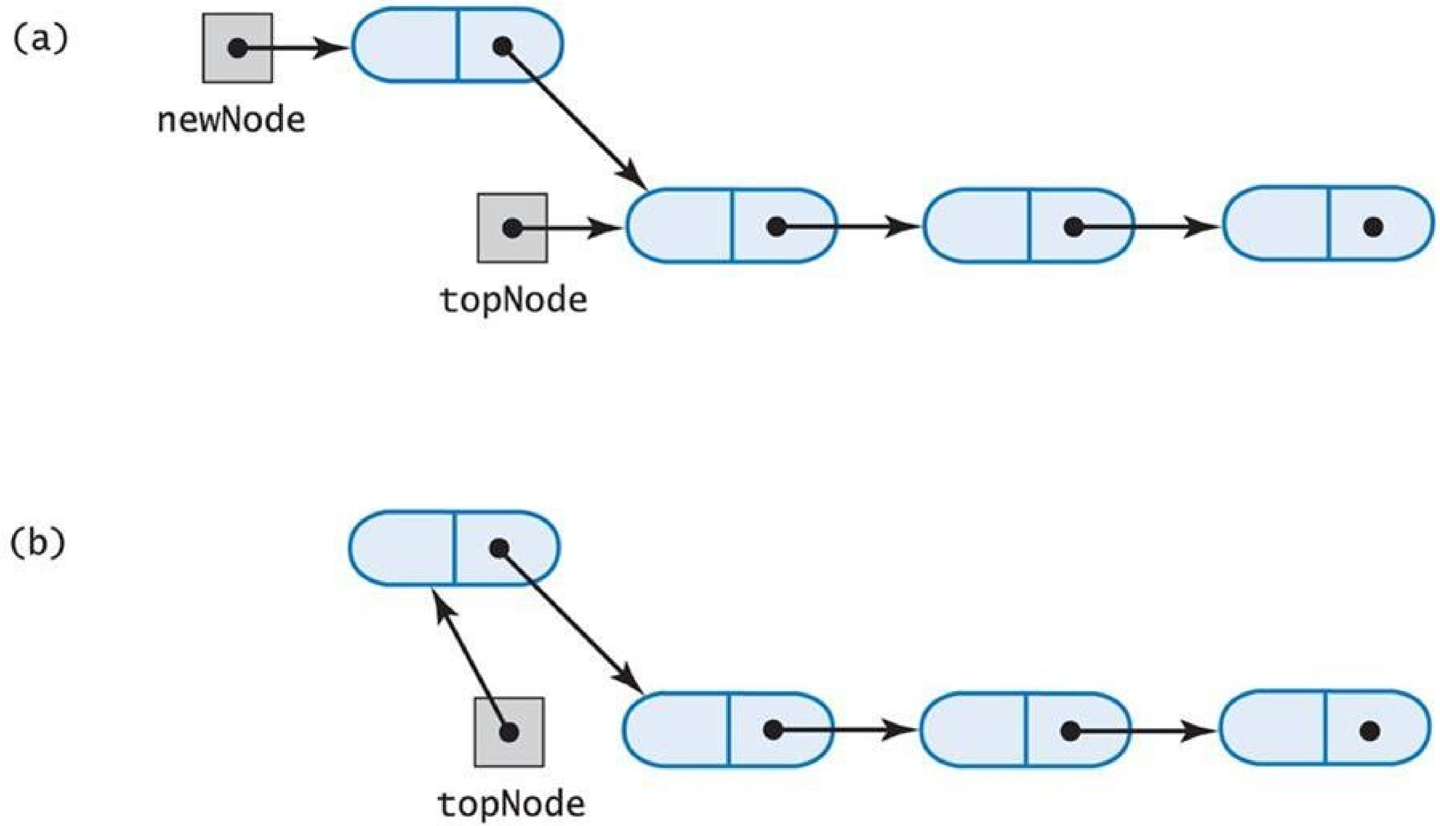


Fig. 28 (a) A new node that references the top of the stack; (b) the new node is now at the top of the stack.

`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

Popping an entry (1/2)

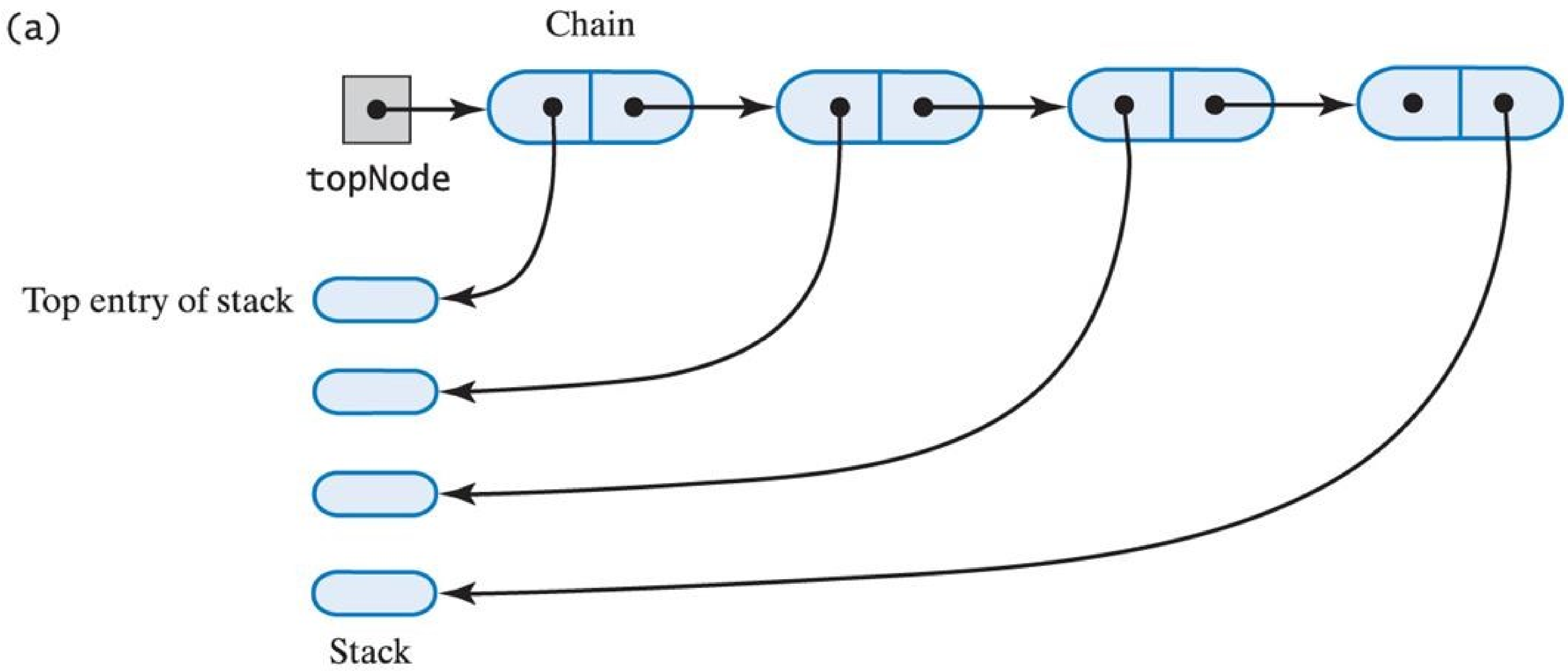


Fig. 29 The stack (a) before the first node in the chain is deleted

Popping an entry (2/2)

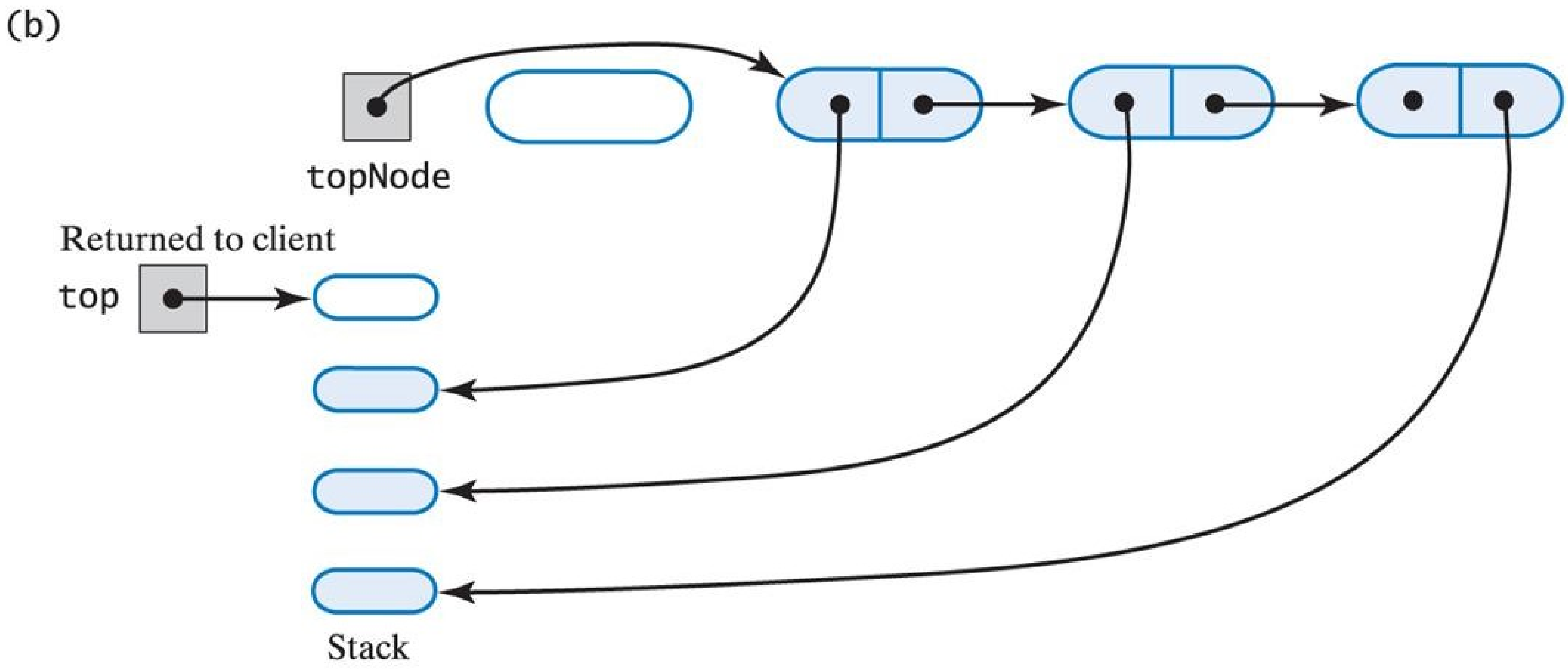


Fig. 30 The stack (b) after the first node in the chain is deleted

`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

Linked Implementation of Queue ADT

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

A Linked Implementation of a Queue

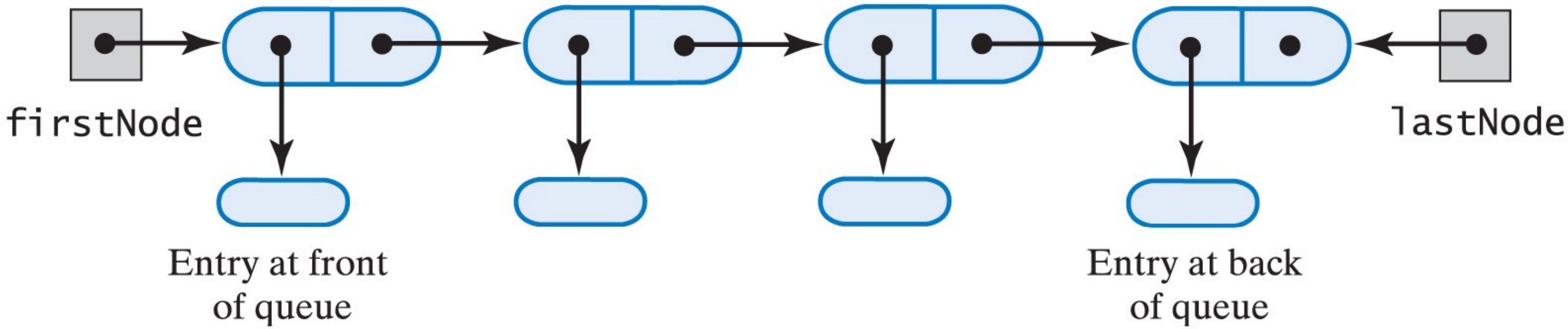


Fig. 31-1 A chain of linked nodes that implements a queue.

Adding to an empty queue

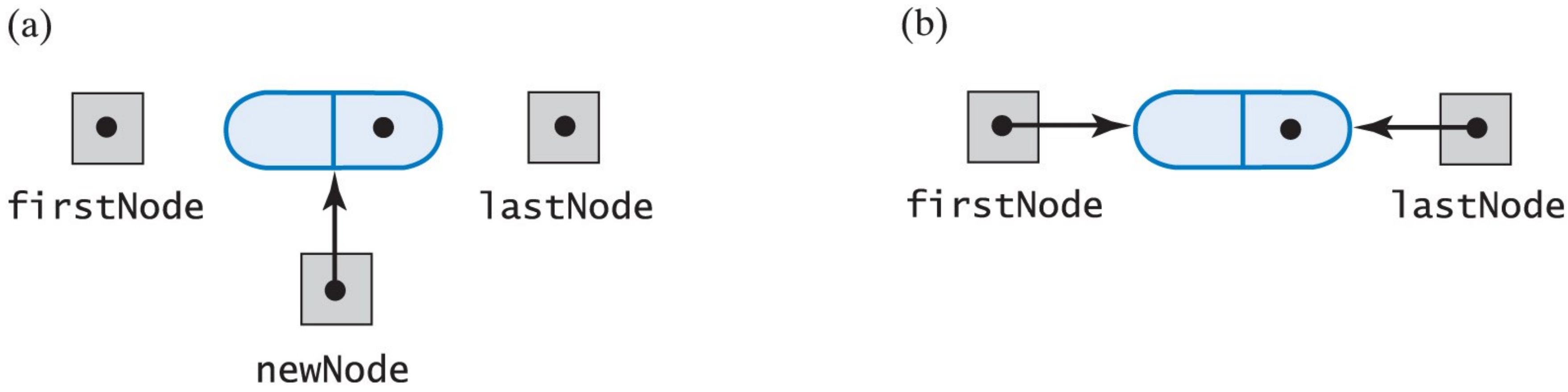


Fig. 32-2(a) Before adding a new node to an empty chain; (b) after adding to it.

Adding to a non-empty queue

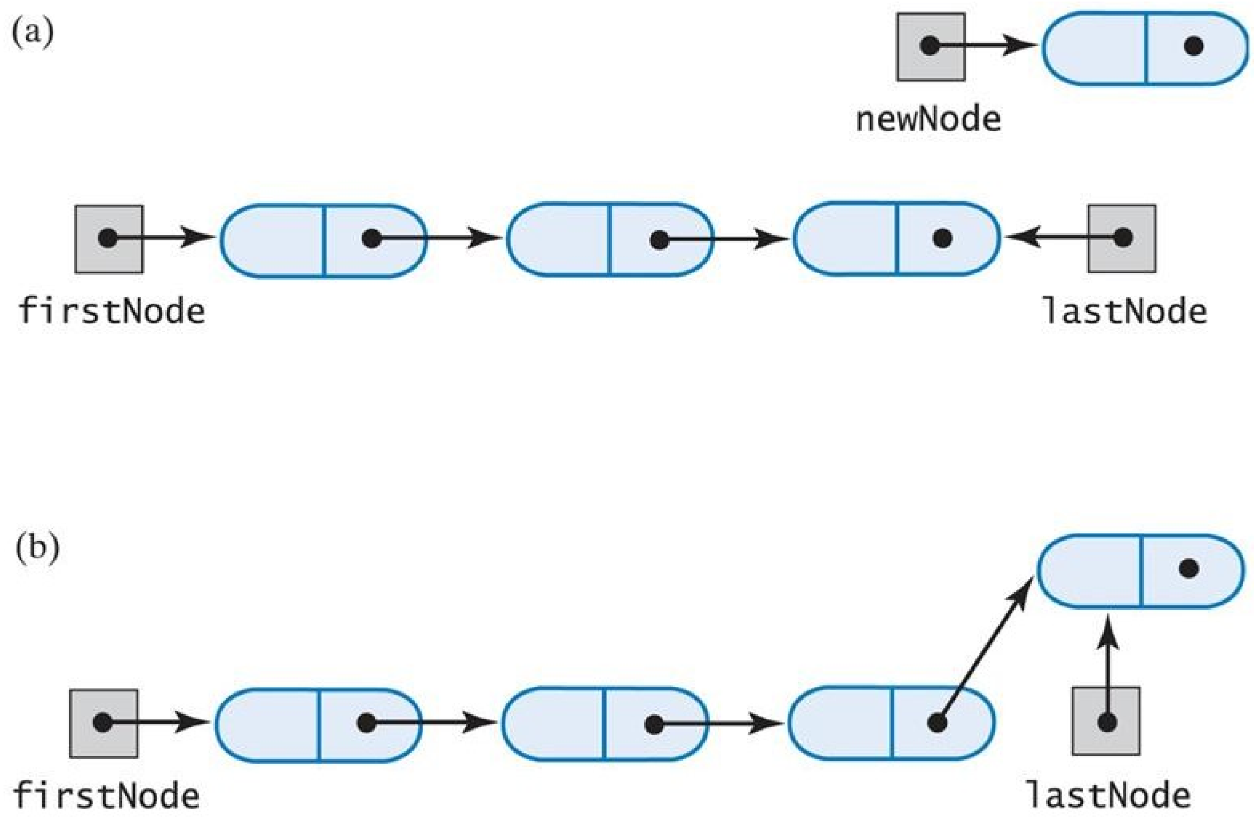


Fig. 32-3(a) Before adding a new node to the end of a chain; (b) after adding it.

`enqueue()` method

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

Removing an entry

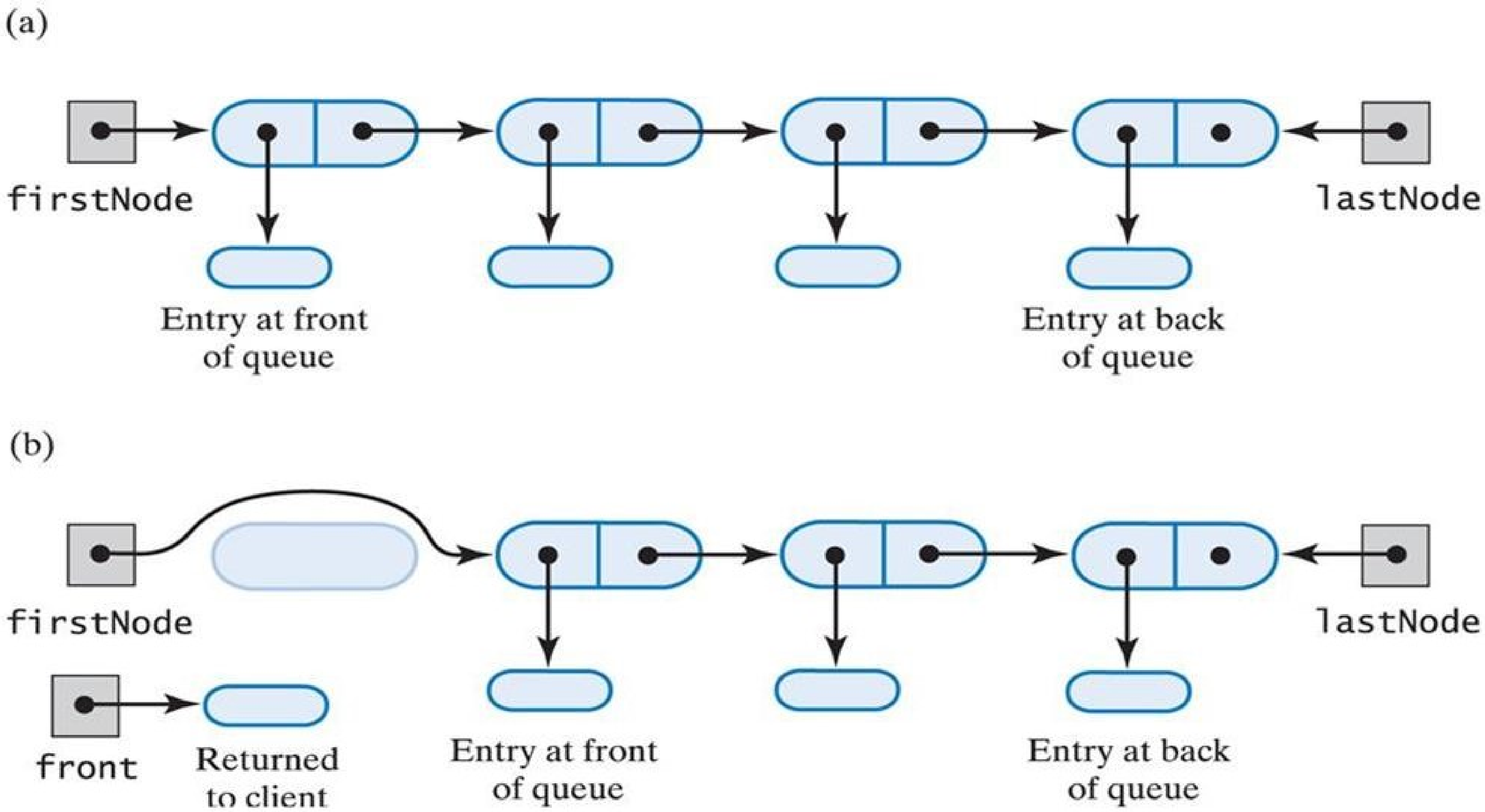


Fig. 32-4(a) A queue of more than one entry; (b) after removing the queue's front.

Removing the only entry from the queue

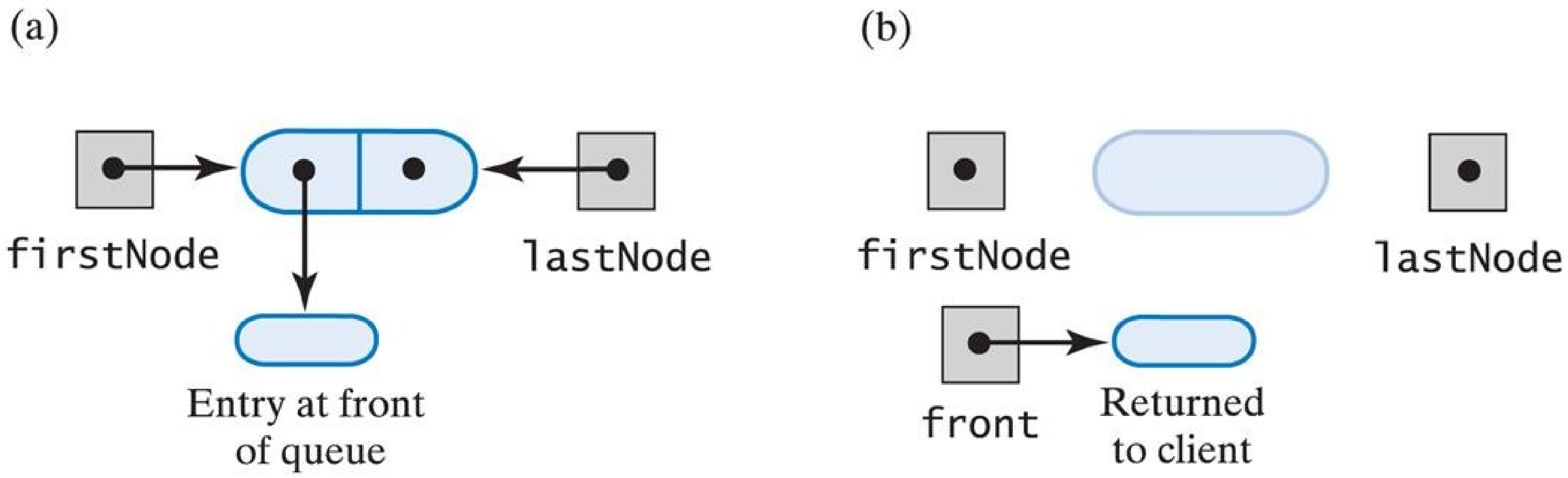


Fig. 32-5 (a) A queue of one entry; (b) after removing the queue's front.

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

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`**

Circular Linked Implementation of a Queue

- 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

Circular Linked Implementation of a Queue

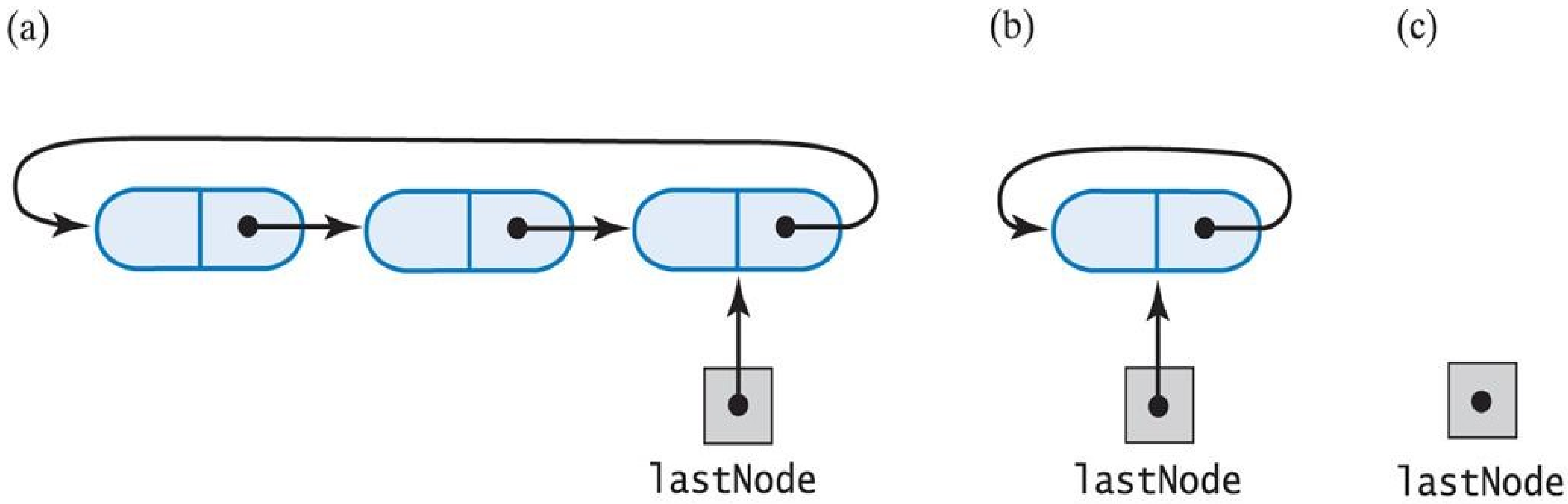


Fig. 33 A circular linked chain with an external reference to its last node that (a) has more than one node; (b) has one node; (c) is empty.

A Doubly Linked Implementation of a Queue

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

A Doubly Linked Implementation of a Queue

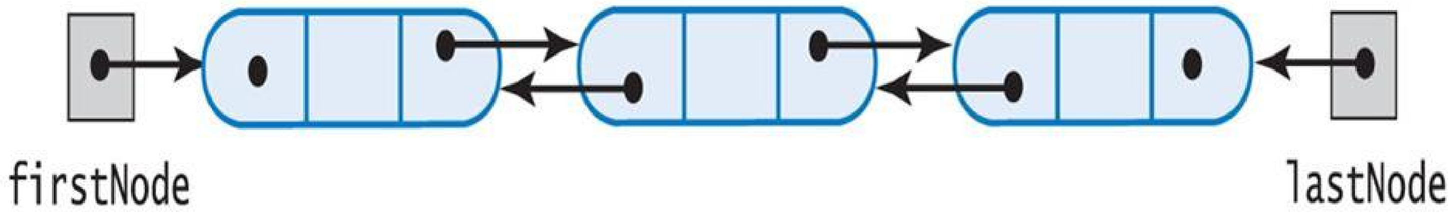


Fig. 34 A doubly linked chain with head and tail references

A Doubly Linked Implementation of a Queue

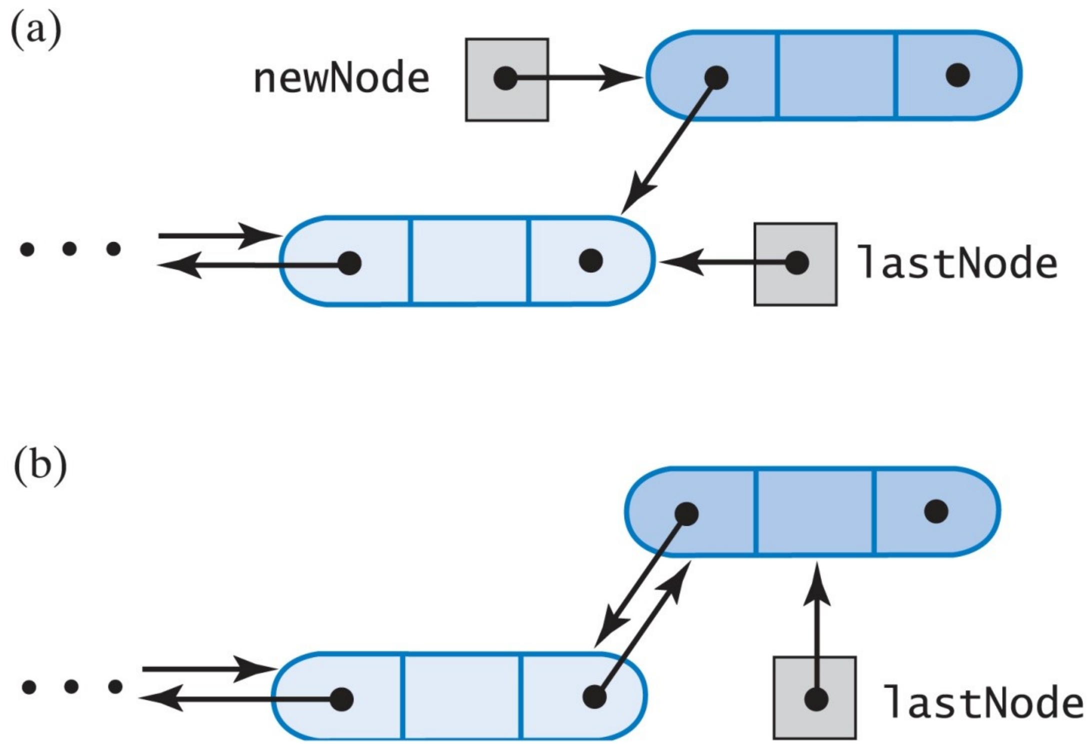


Fig. 35 Adding to the back of a non empty queue:
(a) after the new node is allocated;
(b) after the addition is complete.

A Doubly Linked Implementation of a Queue

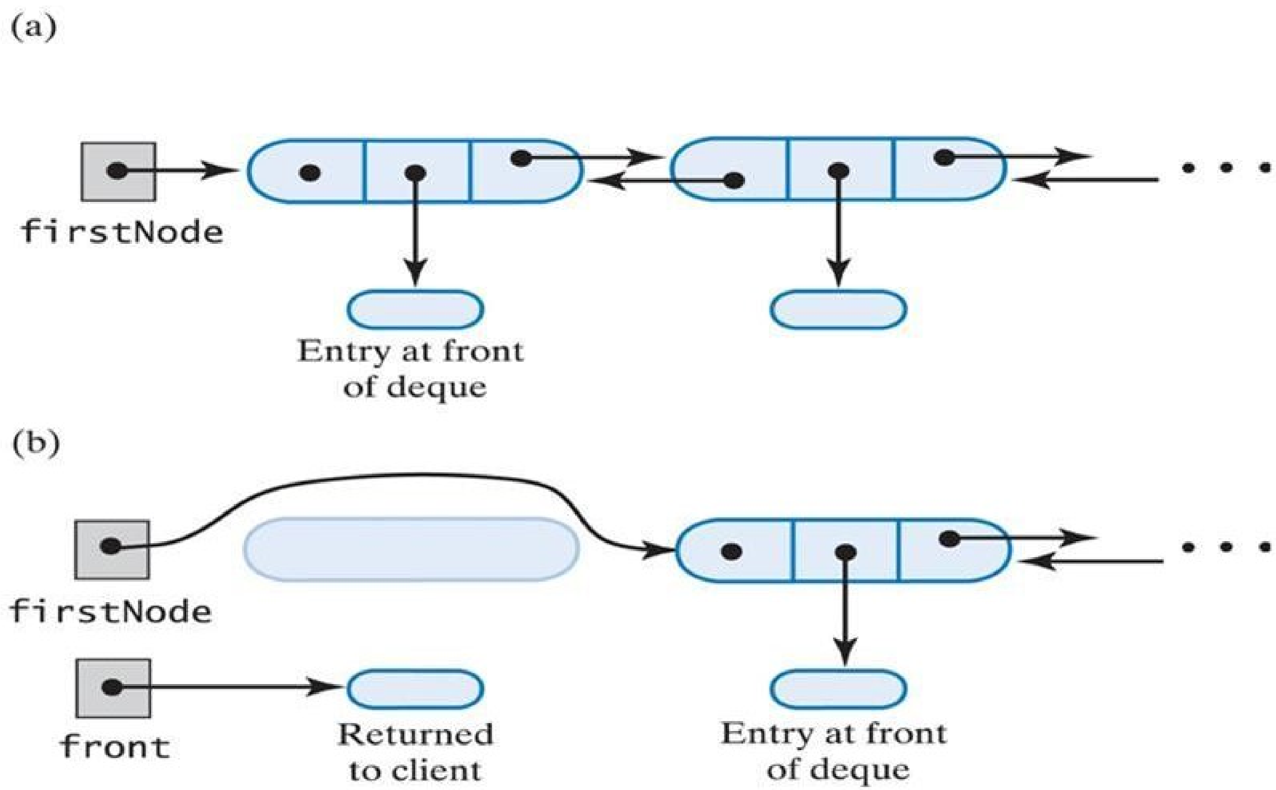


Fig. 36 : (a) a queue containing at least two entries; (b) after removing first node and obtaining reference to the queue's first entry.

Exercise 5

List the steps to remove a node from a doubly linked chain with *both a head reference and tail reference* when the node is

a) at the front of the chain; b) at the end of the chain

Review of learning outcomes

You should now be able to

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

To Do

- Review the slides and source code for this chapter.
- Read up the relevant portions of the recommended text.
- Cthe 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