بنام خداوند جان و خرد

دانشگاه فردوسی مشهد

Data Structures and Algorithms

Soheila Ashkezari-T.

@SAshkezari

Lecture 2: Intro to ADTs

List Case Study

Array

Linked List

Questions

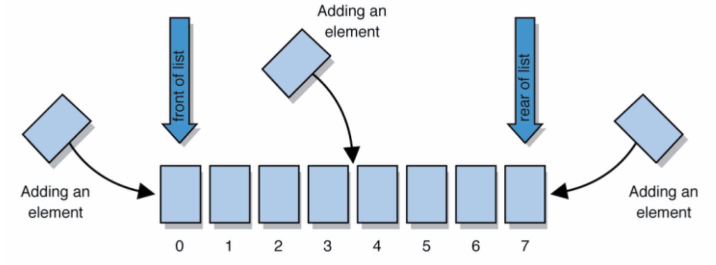
- Consider situations where we want to sore a sequence of objects:
  - high score entries for a video game,
  - patients in a hospital,
  - the names of players on a football team
  - etc.

Case Study: List Implementations

List ADT

state
- Set of ordered items
- Count of items

behavior
- get(index) return item at index
- set(item, index) replace item at index
- append(item) add item to end of list
- insert(item, index) add item at index
- delete(index) delete item at index
- size() count of items

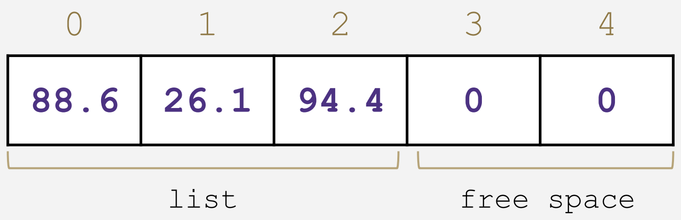


ArrayList
uses an Array as underlying storage

ArrayList<E>

state
```
data[]
size
```

behavior
```
get return data[index]
set data[index] = value
append data[size] =
value, if out of space
grow data
insert shift values to
make hole at index,
data[index] = value, if
out of space grow data
delete shift following
values forward
size return size
```

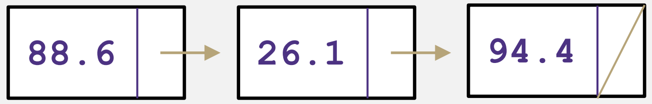


LinkedList
uses nodes as underlying storage

LinkedList<E>

state
```
Node front
size
```

behavior
```
get loop until index,
return node's value
set loop until index,
update node's value
append create new node,
update next of last node
insert create new node,
loop until index, update
next fields
delete loop until index,
skip node
size return size
```

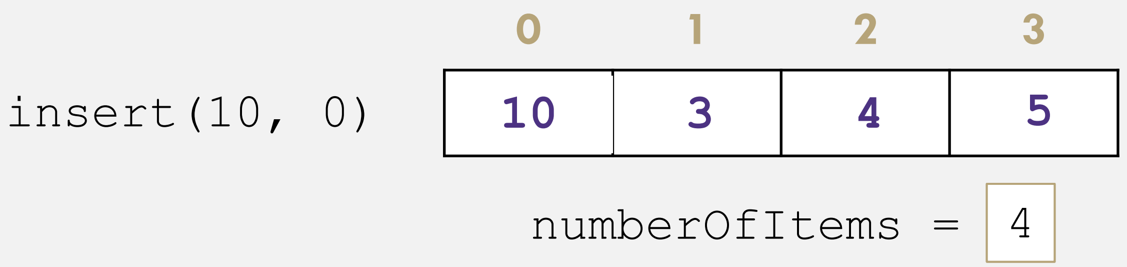


Snapshot

Implementing Insert

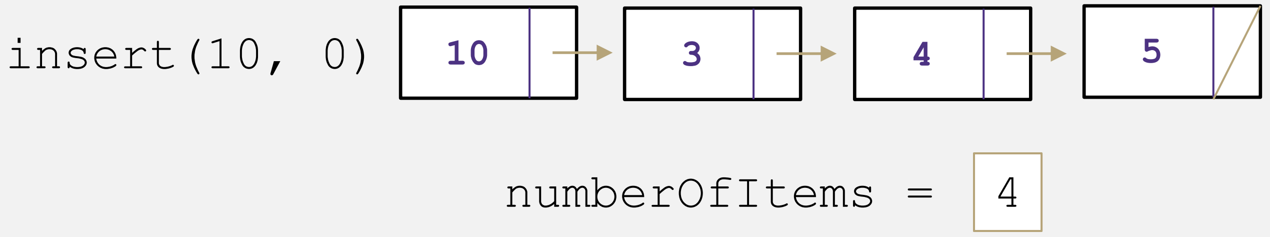
ArrayList<E>

`insert(element, index)` with shifting



LinkedList<E>

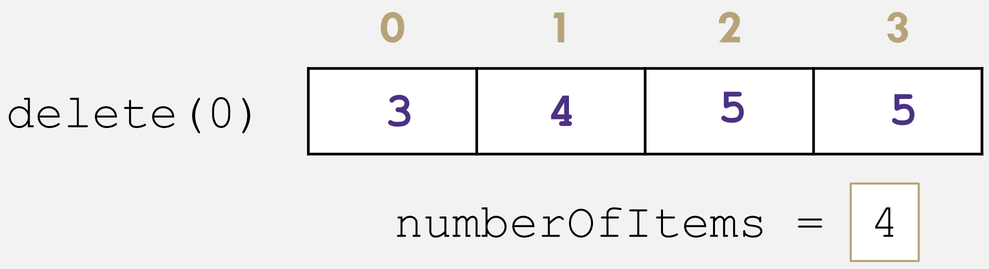
`insert(element, index)` with shifting



Implementing Delete

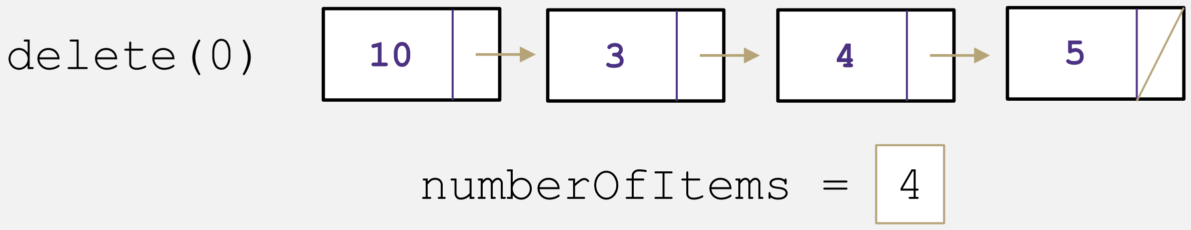
ArrayList<E>

`delete(index)` with shifting



LinkedList<E>

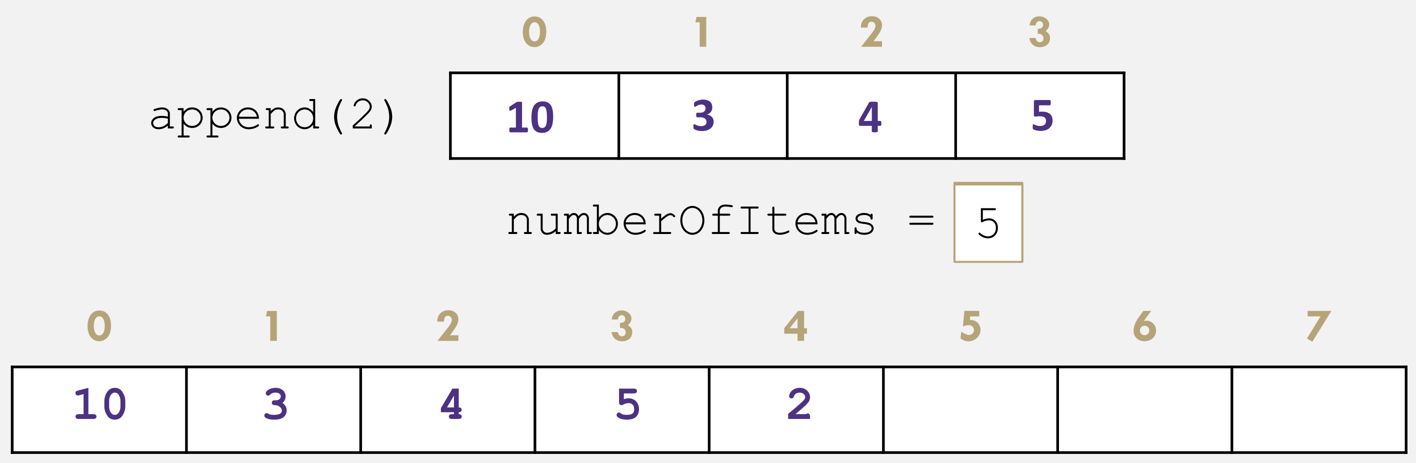
`delete(index)` with shifting



Implementing Append

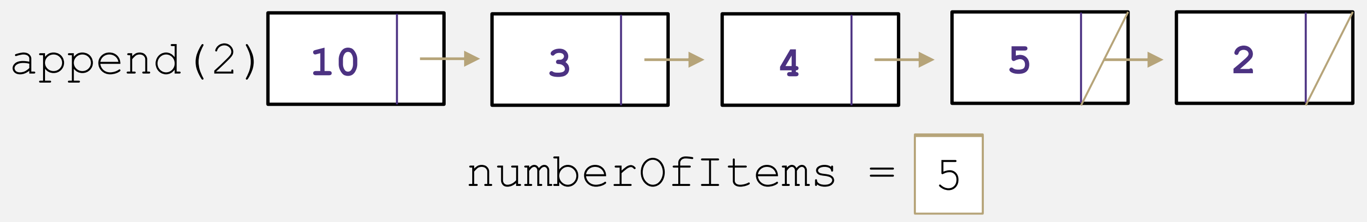
ArrayList<E>

`append(element)` with growth



LinkedList<E>

`append(element)` with growth

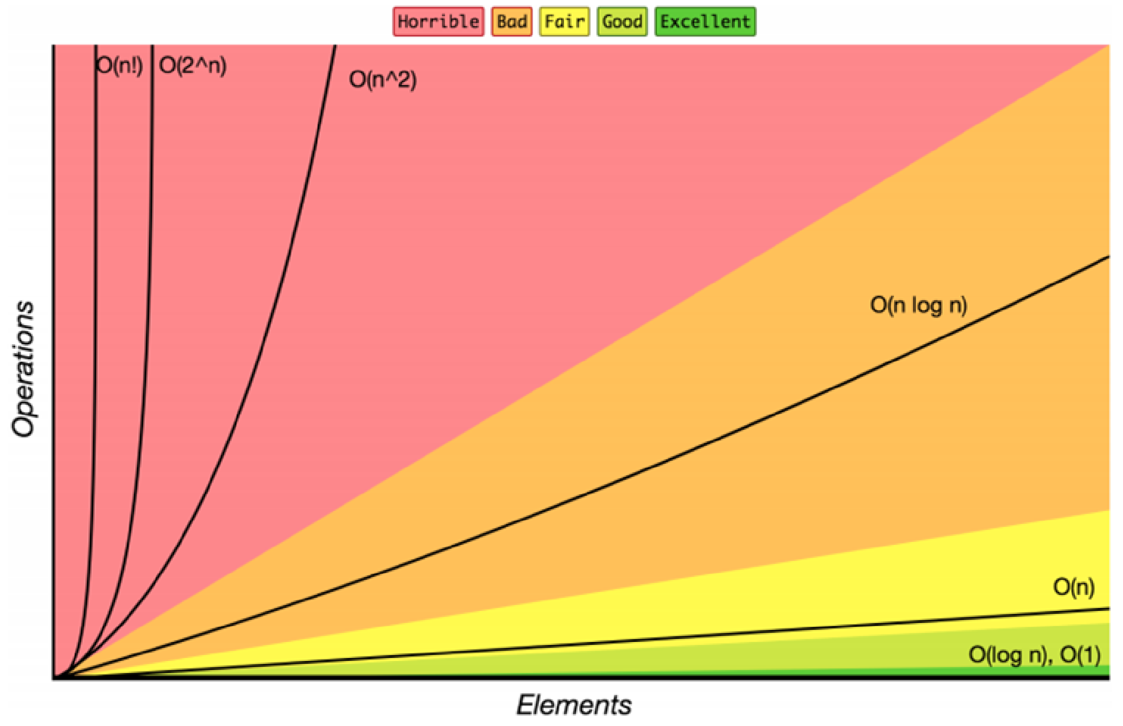


Review: Complexity Class

Note: You don't have to understand all of this right now – we'll dive into it soon.

complexity class: A category of algorithm efficiency based on the algorithm's relationship to the input size N.

Complexity Class	Big-O	Runtime if you double N	Example Algorithm
constant	O(1)	unchanged	Accessing an index of an array
logarithmic	$O(\log_2 N)$	increases slightly	Binary search
linear	O(N)	doubles	Looping over an array
log-linear	$O(N \log_2 N)$	slightly more than doubles	Merge sort algorithm
quadratic	$O(N^2)$	quadruples	Nested loops!
...	...	...	...
exponential	$O(2^N)$	multiplies drastically	Fibonacci with recursion



List ADT tradeoffs

Last time: we used “slow” and “fast” to describe running times.

Let’s be a little more precise.

Recall these basic Big-O ideas : Suppose our list has N elements

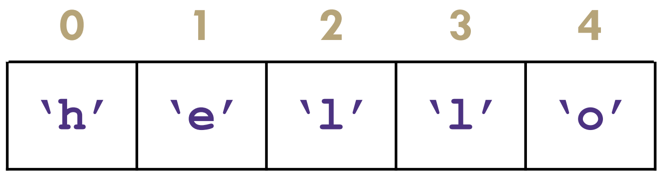
- If a method takes a constant number of steps (like 23 or 5) its running time is O(1)
- If a method takes a linear number of steps (like 4N+3) its running time is O(N)

For ArrayLists and LinkedLists, what is the O() for each of these operations?

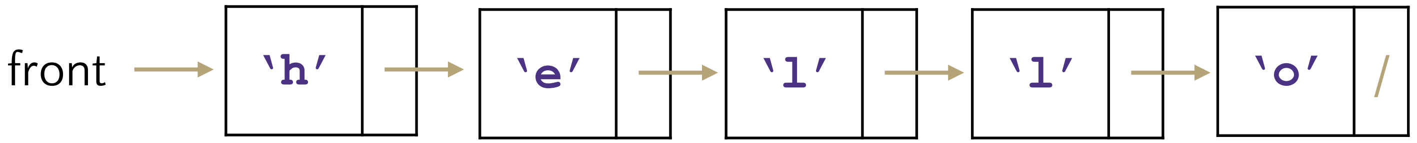
- Time needed to access Nth element
- Time needed to insert at end (what if the array is full?)

What are the memory tradeoffs for our two implementations?

`ArrayList<Character> myArr`



`LinkedList<Character> myLl`



List ADT tradeoffs

Time needed to access Nth element:

- ArrayList: O(1) constant time
- LinkedList: O(N) linear time

Time needed to insert at Nth element (if the array is full!)

- ArrayList: O(N) linear time
- LinkedList: O(N) linear time

Amount of space used overall/across all elements

- ArrayList: sometimes wasted space at end of array
- LinkedList: compact, one node for each entry

Amount of space used per element

- ArrayList: minimal, one element of array
- LinkedList: tiny bit extra, object with two fields

Slightly deeper…

Design Decisions

For every ADT there are lots of different ways to implement them

Based on your situation you should consider:

- Memory vs Speed
- Generic/Reusability vs Specific/Specialized
- One Function vs Another
- Robustness vs Performance

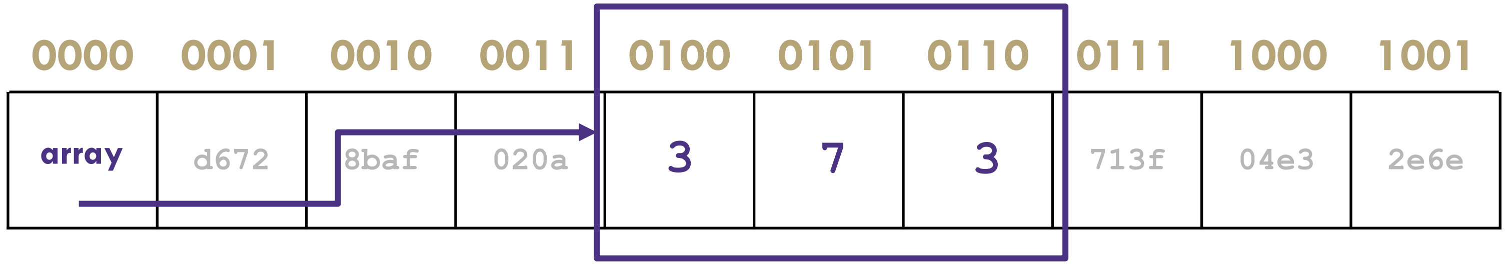
This class is all about implementing ADTs based on making the right design tradeoffs!

A common topic in interview questions!

A quick aside: Types of memory

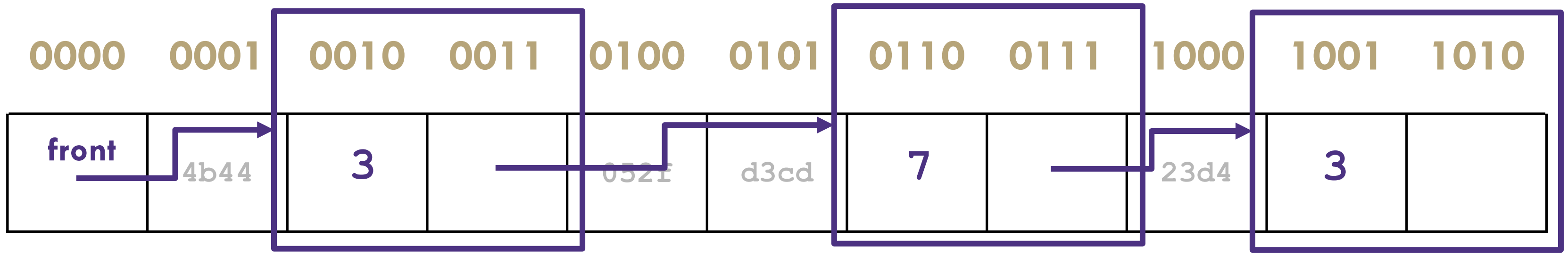
Arrays – **contiguous memory**: when the “new” keyword is used on an array the operating system sets aside a single, right-sized block of computer memory

```
int[] array = new int[3];
array[0] = 3;
array[1] = 7;
array[2] = 3;
```



Nodes– **non-contiguous memory**: when the “new” keyword is used on a single node the operating system sets aside enough space for that object at the next available memory location

```
Node front = new Node(3);
front.next = new Node(7);
front.next.next = new Node(3);
```



Design Decisions

Situation #1: Write a data structure that implements the List ADT that will be used to store a list of songs in a playlist.

Features to consider:

- add or remove songs from list
- change song order
- shuffle play

Why ArrayList?

- optimized element access makes shuffle more efficient
- accessing next element faster in contiguous memory

Why LinkedList?

- easier to reorder songs
- memory right sized for changes in size of playlist, shrinks if songs are removed

Design Decisions

Situation #2: Write a data structure that implements the List ADT that will be used to store the history of a bank customer's transactions.

Features to consider:
- adding a new transaction
- reviewing/retrieving transaction history

Why ArrayList?
- optimized element access makes reviewing based on order easier
- contiguous memory more efficient and less waste than usual array usage because no removals

Why LinkedList?
- if structured with front pointing to most recent transaction, addition of transactions constant time
- memory right sized for large variations in different account history size

Example: Storing Video game scores in Array

```
public class GameEntry {
  private String name;                    // name of the person earning this score
  private int score;                      // the score value
  /** Constructs a game entry with given parameters.. */
  public GameEntry(String n, int s) {
    name = n;
    score = s;
  }
  /** Returns the name field. */
  public String getName( ) { return name; }
  /** Returns the score field. */
  public int getScore( ) { return score; }
  /** Returns a string representation of this entry. */
  public String toString( ) {
    return "(" + name + ", " + score + ")";
  }
}
```

Code Fragment 3.1: Java code for a simple GameEntry class. Note that we include methods for returning the name and score for a game entry object, as well as a method for returning a string representation of this entry.

- Storing high score entries for a video game:
- Variables:
  - Person name
  - His score

Example: Storing Video game scores in Array in C++ !

```
class GameEntry {                              // a game score entry
public:
  GameEntry(const string& n="", int s=0);  // constructor
  string getName() const;                      // get player name
  int getScore() const;                        // get score
private:
  string name;                                 // player's name
  int score;                                   // player's score
};
```

Code Fragment 3.1: A C++ class representing a game entry.

```
GameEntry::GameEntry(const string& n, int s) // constructor
  : name(n), score(s) { }
                                         // accessors
string GameEntry::getName() const { return name; }
int GameEntry::getScore() const { return score; }
```

Code Fragment 3.2: GameEntry constructor and accessors.

Example: Storing Video game scores in Array

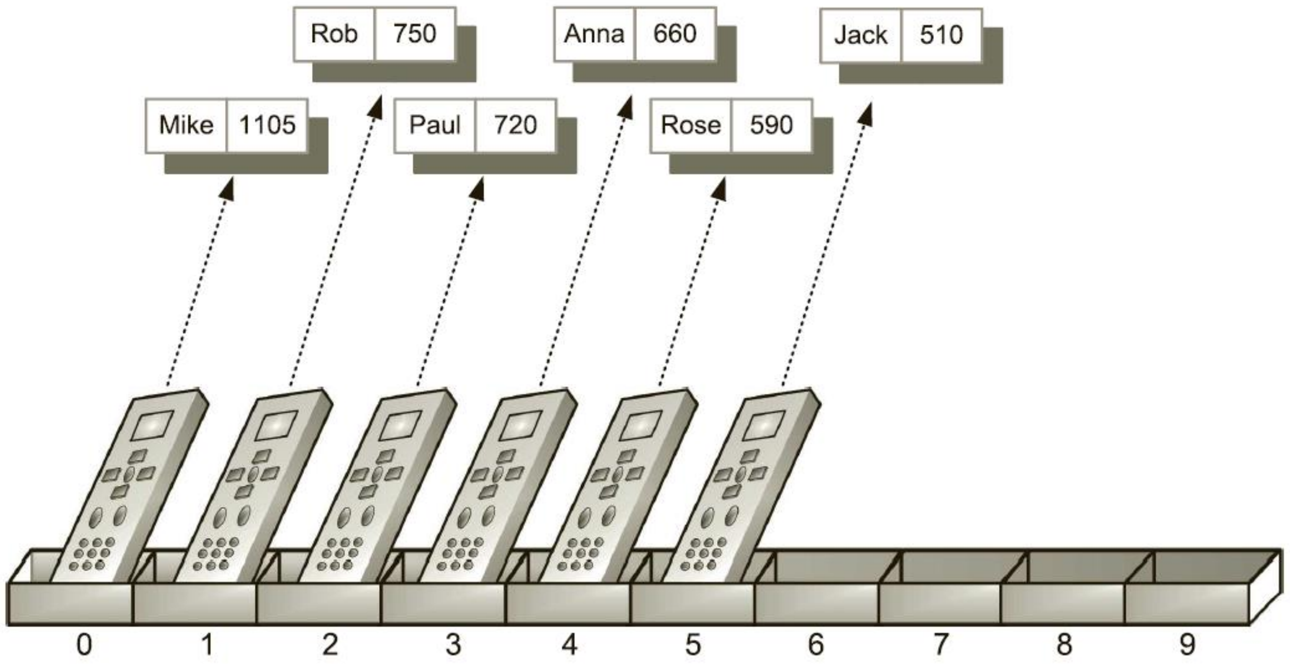


Figure 3.1: An illustration of an array of length ten storing references to six GameEntry objects in the cells with indices 0 to 5; the rest are **null** references.

Example: Storing Video game scores in Array

- Creating an array of instance from GameEntry class

```
/** Class for storing high scores in an array in nondecreasing order. */
public class Scoreboard {
  private int numEntries = 0;                    // number of actual entries
  private GameEntry[ ] board;                    // array of game entries (names & scores)
  /** Constructs an empty scoreboard with the given capacity for storing entries. */
  public Scoreboard(int capacity) {
    board = new GameEntry[capacity];
  }
  // more methods will go here
}
```

Code Fragment 3.2: The beginning of a Scoreboard class for maintaining a set of scores as GameEntry objects. (Completed in Code Fragments 3.3 and 3.4.)

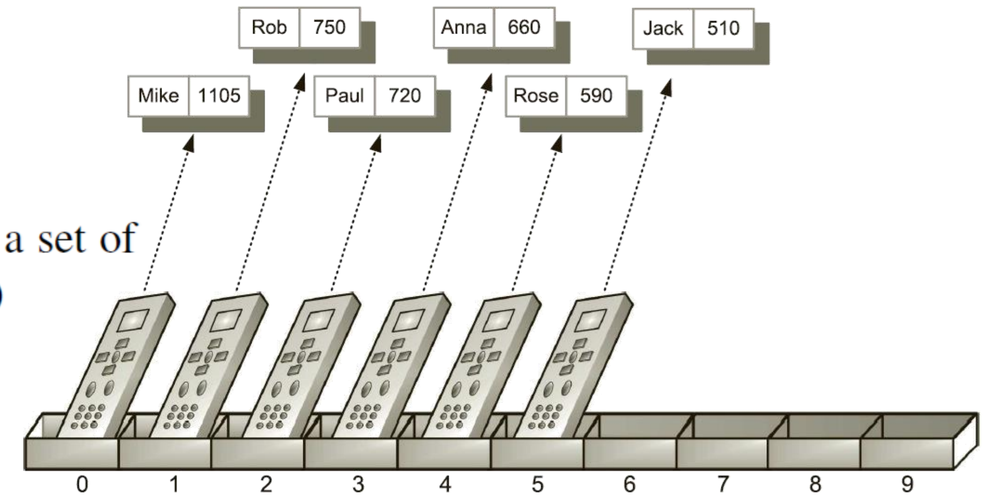


Figure 3.1: An illustration of an array of length ten storing references to six GameEntry objects in the cells with indices 0 to 5; the rest are **null** references.

Example: Storing Video game scores in Array

- Add to an Array

```
/** Attempt to add a new score to the collection (if it is high enough) */
public void add(GameEntry e) {
  int newScore = e.getScore( );
  // is the new entry e really a high score?
  if (numEntries < board.length || newScore > board[numEntries−1].getScore( )) {
    if (numEntries < board.length)               // no score drops from the board
      numEntries++;                              // so overall number increases
    // shift any lower scores rightward to make room for the new entry
    int j = numEntries − 1;
    while (j > 0 && board[j−1].getScore( ) < newScore) {
      board[j] = board[j−1];                     // shift entry from j-1 to j
      j−−;                                       // and decrement j
    }
    board[j] = e;                                // when done, add new entry
  }
}
```

Code Fragment 3.3: Java code for inserting a GameEntry object into a Scoreboard.

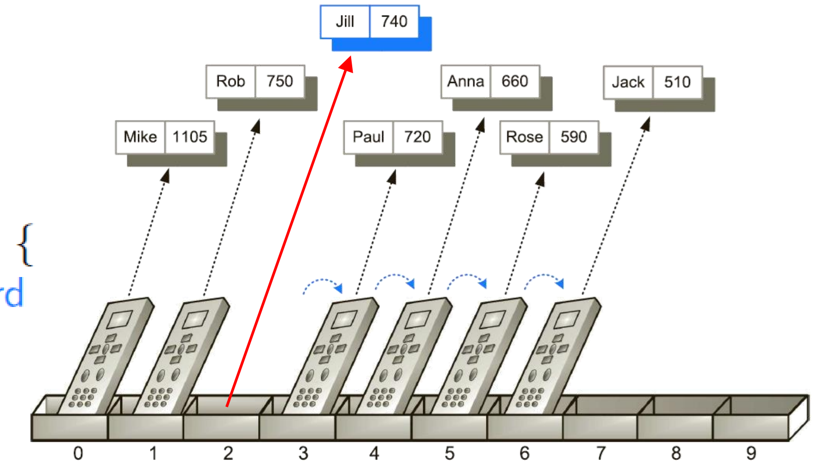


Figure 3.2: Preparing to add Jill's GameEntry object to the board array. In order to make room for the new reference, we have to shift any references to game entries with smaller scores than the new one to the right by one cell.

Example: Storing Video game scores in Array

- Remove from an array

```
/** Remove and return the high score at index i. */
public GameEntry remove(int i) throws IndexOutOfBoundsException {
  if (i < 0 || i >= numEntries)
    throw new IndexOutOfBoundsException("Invalid index: " + i);
  GameEntry temp = board[i];                    // save the object to be removed
  for (int j = i; j < numEntries − 1; j++)      // count up from i (not down)
    board[j] = board[j+1];                      // move one cell to the left
  board[numEntries −1 ] = null;                 // null out the old last score
  numEntries−−;
  return temp;                                  // return the removed object
}
```

Code Fragment 3.4: Java code for performing the Scoreboard.remove operation.

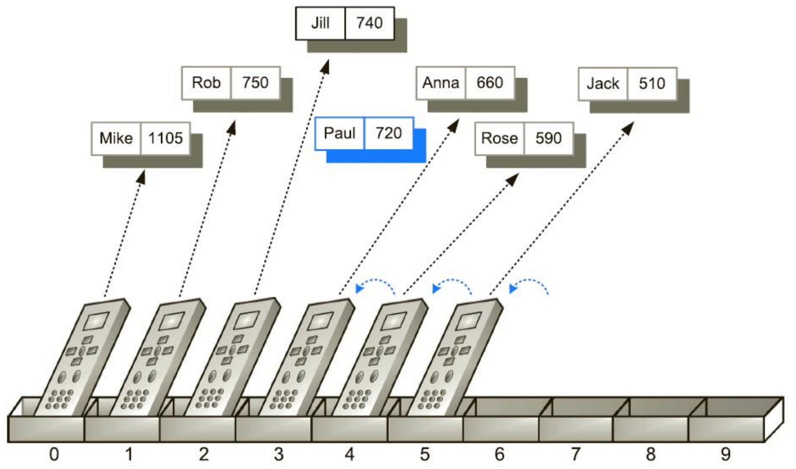


Figure 3.4: An illustration of the removal of Paul's score from index 3 of an array storing references to GameEntry objects.

Example: Storing Video game scores in Array

- Sorting an Array (warm-up)
- Insertion Sort
  - The algorithm proceeds by considering one element at a time, placing the element in the correct order relative to those before it. We start with the first element in the array, which is trivially sorted by itself. When considering the next element in the array, if it is smaller than the first, we swap them. Next we consider the third element in the array, swapping it leftward until it is in its proper order relative to the first two elements. We continue in this manner with the fourth element, the fifth, and so on, until the whole array is sorted.

Sorting an Array

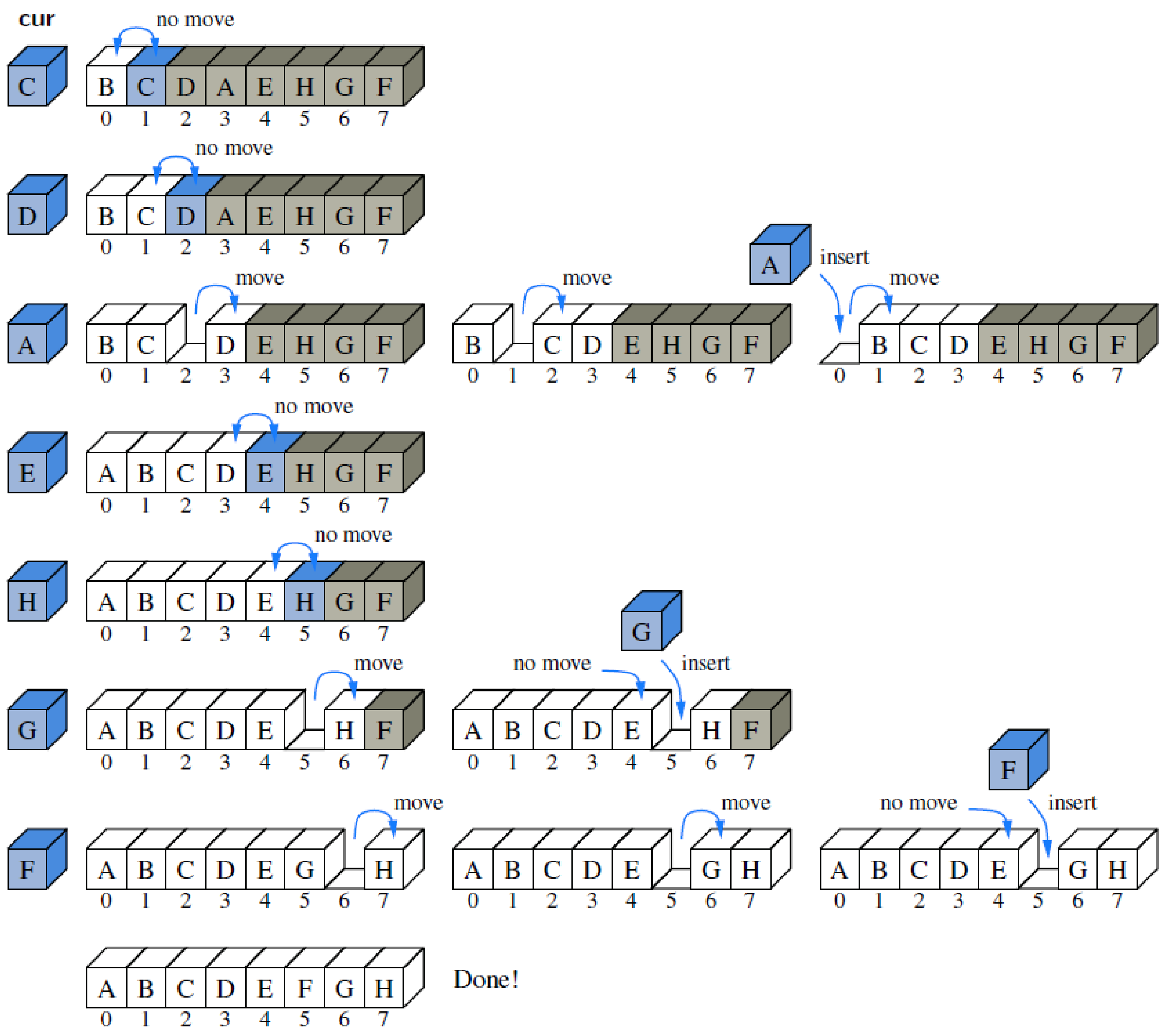
- Insertion Sort

```
/** Insertion-sort of an array of characters into nondecreasing order */
public static void insertionSort(char[ ] data) {
  int n = data.length;
  for (int k = 1; k < n; k++) {                   // begin with second character
    char cur = data[k];                           // time to insert cur=data[k]
    int j = k;                                    // find correct index j for cur
    while (j > 0 && data[j−1] > cur) {            // thus, data[j-1] must go after cur
      data[j] = data[j−1];                        // slide data[j-1] rightward
      j−−;                                        // and consider previous j for cur
    }
    data[j] = cur;                                // this is the proper place for cur
  }
}
```

Code Fragment 3.6: Java code for performing insertion-sort on a character array.

Sorting an Array

- **Figure 3.5:** Execution of the insertion-sort algorithm on an array of eight characters.
- Each row corresponds to an iteration of the outer loop, and each copy of the sequence in a row corresponds to an iteration of the inner loop. The current element that is being inserted is highlighted in the array, and shown as the cur value.



Singly linked list

- Arrays are great for storing things in a certain order, but they have drawback:
  - The capacity of the array must be fixed when it is created,
  - insertions and deletions at interior positions of an array can be time consuming if many elements must be shifted.
- ***linked list*** provides an alternative to an array-based structure. A linked list, in its simplest form,
  - is a collection of ***nodes*** that collectively form a linear sequence. In a ***singly linked list***, each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list.

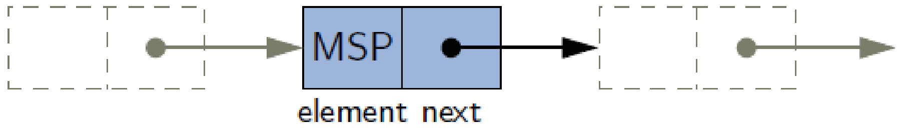


Figure 3.10: Example of a node instance that forms part of a singly linked list. The node's element field refers to an object that is an element of the sequence (the airport code MSP, in this example), while the next field refers to the subsequent node of the linked list (or null if there is no further node).

Singly linked list

- the linked list instance must keep a reference to the first node of the list, known as the ***head***. Without an explicit reference to the head, there would be no way to locate that node (or indirectly, any others).
- The last node of the list is known as the ***tail***. The tail of a list can be found by ***traversing*** the linked list—starting at the head and moving from one node to another by following each node's next reference. We can identify the tail as the node having null as its next reference. This process is also known as **link hopping** or **pointer hopping**.

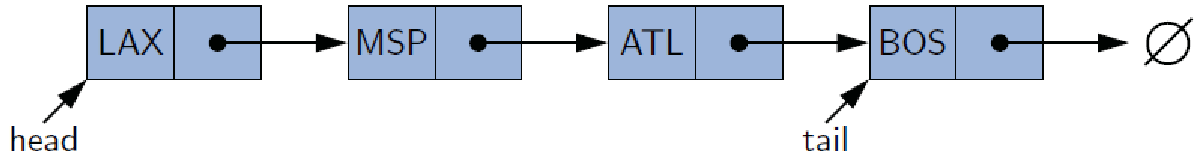


Figure 3.11: Example of a singly linked list whose elements are strings indicating airport codes. The list instance maintains a member named head that refers to the first node of the list, and another member named tail that refers to the last node of the list. The **null** value is denoted as Ø.

Inserting an Element at the Head of a Singly Linked List

- Inserting a new element at the beginning of a singly linked list.
    - Note that we set the next pointer of the new node ***before*** we reassign variable head to it. If the list were initially empty (i.e., head is null), then a natural consequence is that the new node has its next reference set to null.

```
Algorithm addFirst($e$):
    newest = Node($e$)      {create new node instance storing reference to element $e$}
    newest.next = head      {set new node's next to reference the old head node}
    head = newest           {set variable head to reference the new node}
    size = size + 1         {increment the node count}
```

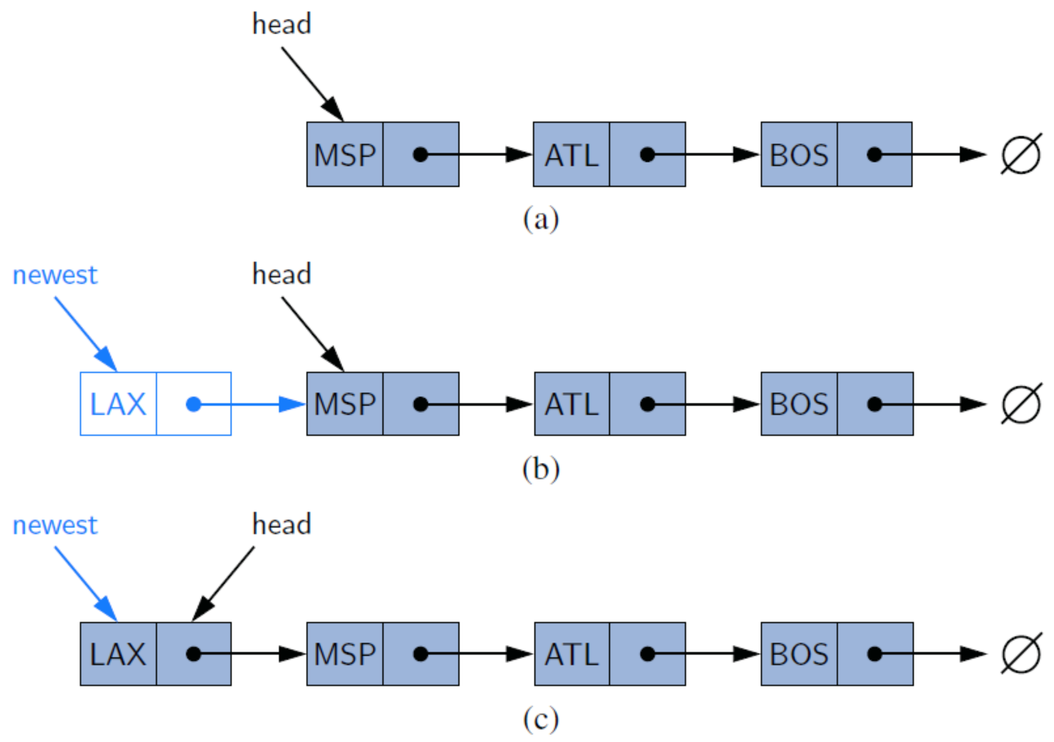


Figure 3.12: Insertion of an element at the head of a singly linked list: (a) before the insertion; (b) after a new node is created and linked to the existing head; (c) after reassignment of the head reference to the newest node.

Inserting an Element at the Tail of a Singly Linked List

- Inserting a new node at the end of a singly linked list.
    - Note that we set the next pointer for the old tail node ***before*** we make variable tail point to the new node. This code would need to be adjusted for inserting onto an empty list, since there would not be an existing tail node.

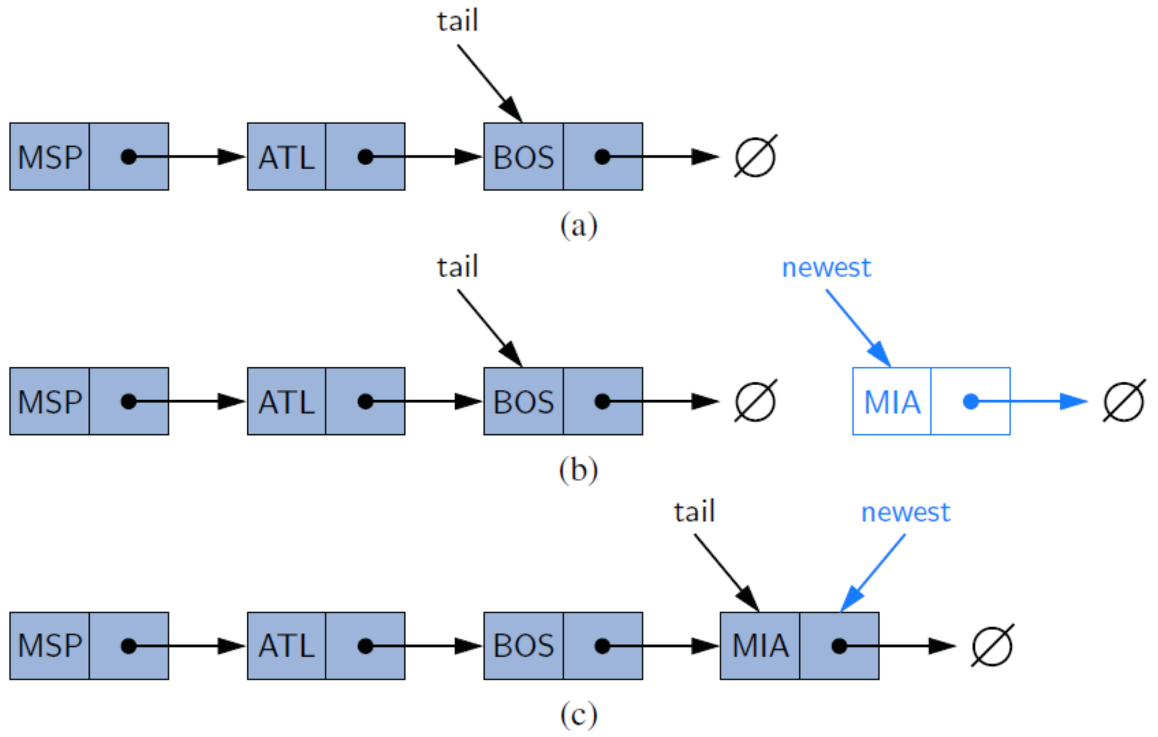


Figure 3.13: Insertion at the tail of a singly linked list: (a) before the insertion; (b) after creation of a new node; (c) after reassignment of the tail reference. Note that we must set the next link of the tail node in (b) before we assign the tail variable to point to the new node in (c).

```
Algorithm addLast($e$):
    newest = Node($e$)    {create new node instance storing reference to element $e$}
    newest.next = null    {set new node's next to reference the null object}
    tail.next = newest    {make old tail node point to new node}
    tail = newest         {set variable tail to reference the new node}
    size = size + 1       {increment the node count}
```

Removing an Element from a Singly Linked List

- Removing the node at the beginning of a singly linked list.

```
Algorithm removeFirst( ):
    if head == null then
        the list is empty.
    head = head.next                {make head point to next node (or null)}
    size = size − 1                 {decrement the node count}
```

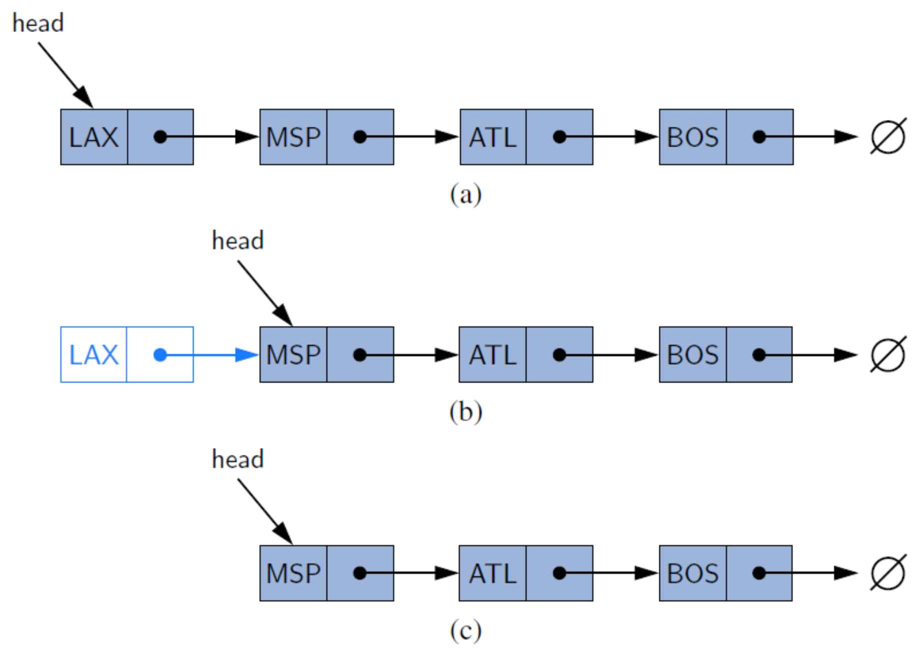


Figure 3.14: Removal of an element at the head of a singly linked list: (a) before the removal; (b) after "linking out" the old head; (c) final configuration.

Removing an Element from a Singly Linked List

- HOW SHOULD WE DO IT ?

- Unfortunately, we cannot easily delete the last node of a singly linked list. Even if we maintain a tail reference directly to the last node of the list, we must be able to access the node ***before*** the last node in order to remove the last node.
- But we cannot reach the node before the tail by following next links from the tail. The only way to access this node is to start from the head of the list and search all the way through the list. But such a sequence of link-hopping operations could take a long time. If we want to support such an operation efficiently, we will need to make our list ***doubly linked*** (will be discussed later).

Implementing a Singly Linked List Class

- Methods:

size( ):	Returns the number of elements in the list.
isEmpty( ):	Returns **true** if the list is empty, and **false** otherwise.
first( ):	Returns (but does not remove) the first element in the list.
last( ):	Returns (but does not remove) the last element in the list.
addFirst(*e*):	Adds a new element to the front of the list.
addLast(*e*):	Adds a new element to the end of the list.
removeFirst( ):	Removes and returns the first element of the list.

If first( ), last( ), or removeFirst( ) are called on a list that is empty, we will simply return a **null** reference and leave the list unchanged.

Implementing a Singly Linked List Class

```
public class SinglyLinkedList<E> {
  //---------------- nested Node class ----------------
  private static class Node<E> {
    private E element;                // reference to the element stored at this node
    private Node<E> next;             // reference to the subsequent node in the list
    public Node(E e, Node<E> n) {
      element = e;
      next = n;
    }
    public E getElement( ) { return element; }
    public Node<E> getNext( ) { return next; }
    public void setNext(Node<E> n) { next = n; }
  } //----------- end of nested Node class -----------
  ... rest of SinglyLinkedList class will follow ...
```

Implementing a Singly Linked List Class

```
public class SinglyLinkedList<E> {
...
  (nested Node class goes here)

  // instance variables of the SinglyLinkedList
  private Node<E> head = null;          // head node of the list (or null if empty)
  private Node<E> tail = null;          // last node of the list (or null if empty)
  private int size = 0;                 // number of nodes in the list
  public SinglyLinkedList( ) { }        // constructs an initially empty list
  // access methods
  public int size( ) { return size; }
  public boolean isEmpty( ) { return size == 0; }
  public E first( ) {              // returns (but does not remove) the first element
    if (isEmpty( )) return null;
    return head.getElement( );
  }
  public E last( ) {               // returns (but does not remove) the last element
    if (isEmpty( )) return null;
    return tail.getElement( );
  }
  // update methods
  public void addFirst(E e) {           // adds element e to the front of the list
    head = new Node<>(e, head);         // create and link a new node
    if (size == 0)
      tail = head;                      // special case: new node becomes tail also
    size++;
  }
```

Implementing a Singly Linked List Class

```
  public void addLast(E e) {                 // adds element e to the end of the list
    Node<E> newest = new Node<>(e, null);    // node will eventually be the tail
    if (isEmpty())
      head = newest;                         // special case: previously empty list
    else
      tail.setNext(newest);                  // new node after existing tail
    tail = newest;                           // new node becomes the tail
    size++;
  }
  public E removeFirst() {                   // removes and returns the first element
    if (isEmpty()) return null;              // nothing to remove
    E answer = head.getElement();
    head = head.getNext();                   // will become null if list had only one node
    size--;
    if (size == 0)
      tail = null;                           // special case as list is now empty
    return answer;
  }
}
```

Circularly Linked Lists

- Linked lists are traditionally viewed as storing a sequence of items in a linear order, from first to last. However, there are many applications in which data can be more naturally viewed as having a ***cyclic order***, with well-defined neighboring relationships, but no fixed beginning or end.
    - For example, many multiplayer games are turn-based, with player A taking a turn, then player B, then player C, and so on, but eventually back to player A again, and player B again, with the pattern repeating.
    - As another example, city buses and subways often run on a continuous loop, making stops in a scheduled order, but with no designated first or last stop per se.

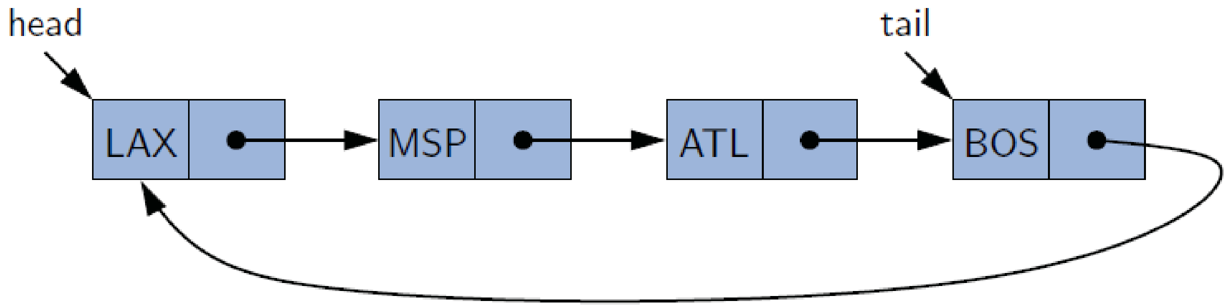


Figure 3.16: Example of a singly linked list with circular structure.

Doubly Linked Lists

- To provide greater symmetry, we define a linked list in which each node keeps an explicit reference to the node before it and a reference to the node after it. Such a structure is known as a ***doubly linked list***. These lists allow a greater variety of *O*(1)-time update operations, including insertions and deletions at arbitrary positions within the list.

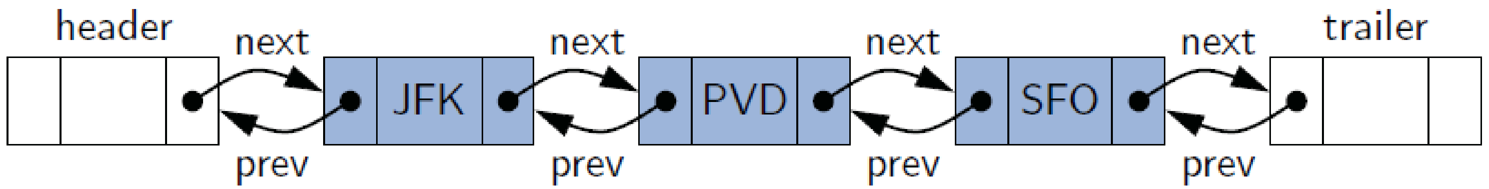


Figure 3.19: A doubly linked list representing the sequence { JFK, PVD, SFO }, using sentinels header and trailer to demarcate the ends of the list.

Inserting and Deleting with a Doubly Linked List

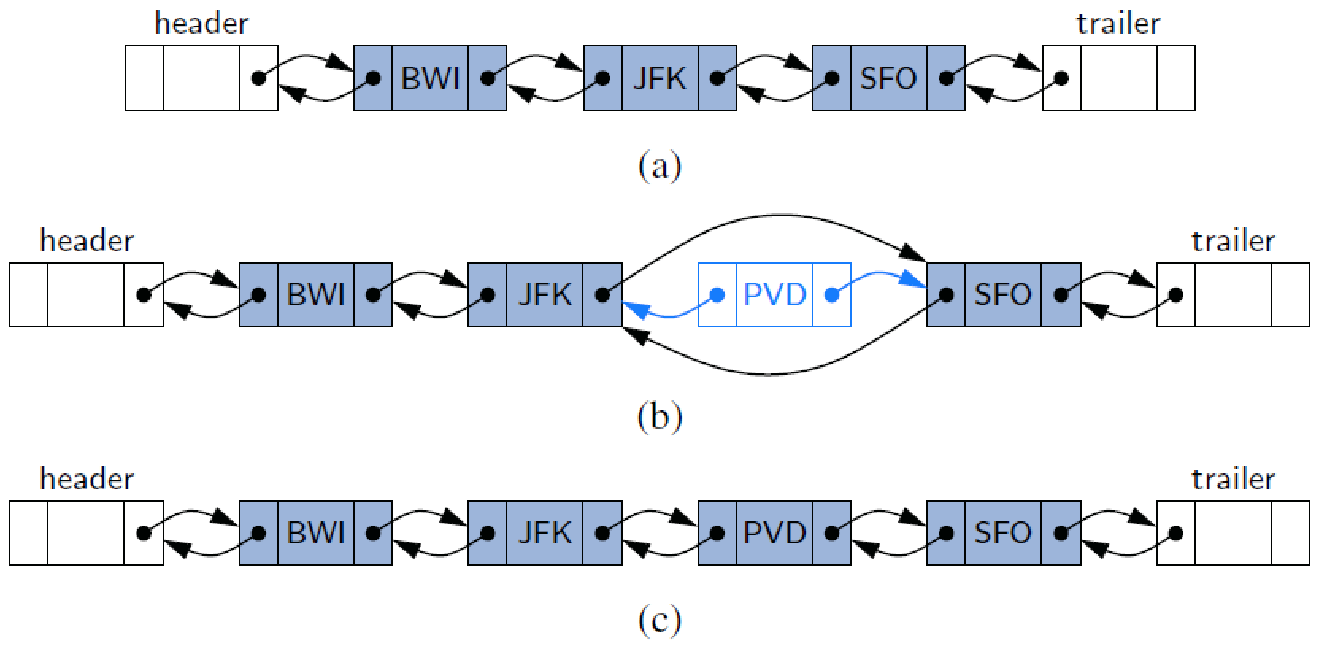


Figure 3.20: Adding an element to a doubly linked list with header and trailer sentinels: (a) before the operation; (b) after creating the new node; (c) after linking the neighbors to the new node.

Inserting and Deleting with a Doubly Linked List

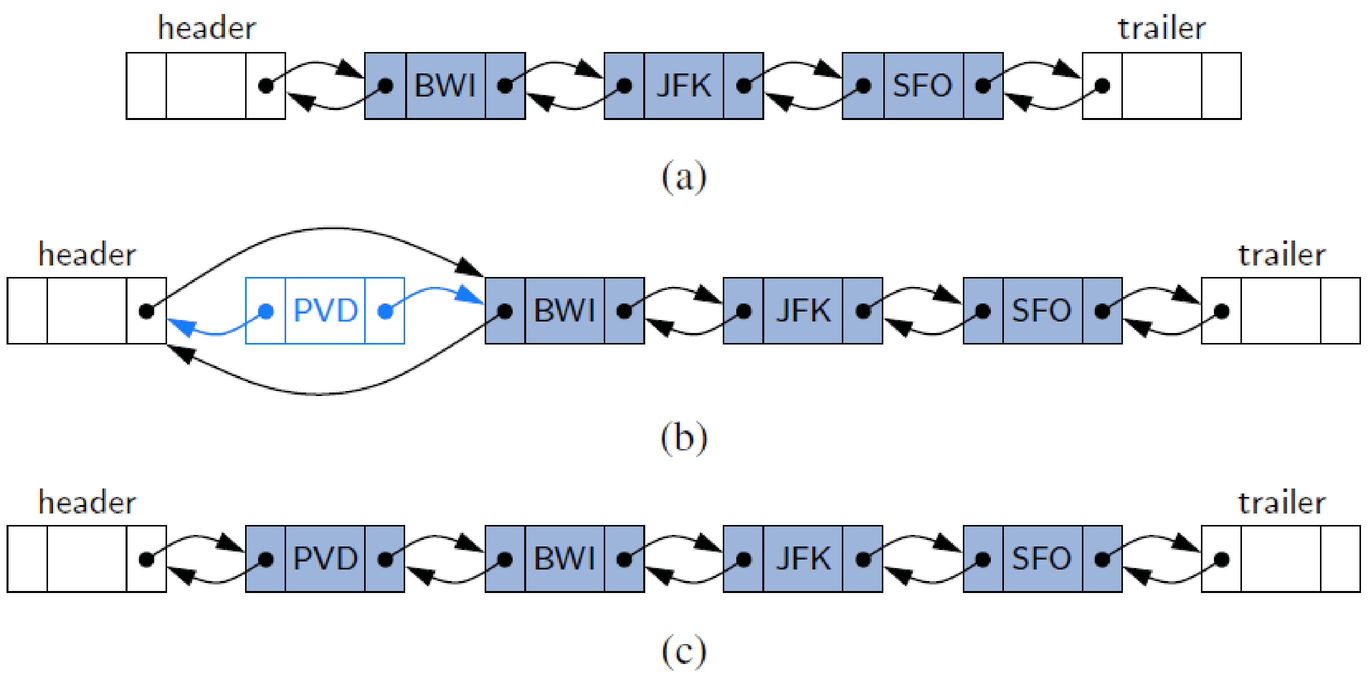


Figure 3.21: Adding an element to the front of a sequence represented by a doubly linked list with header and trailer sentinels: (a) before the operation; (b) after creating the new node; (c) after linking the neighbors to the new node.

Inserting and Deleting with a Doubly Linked List

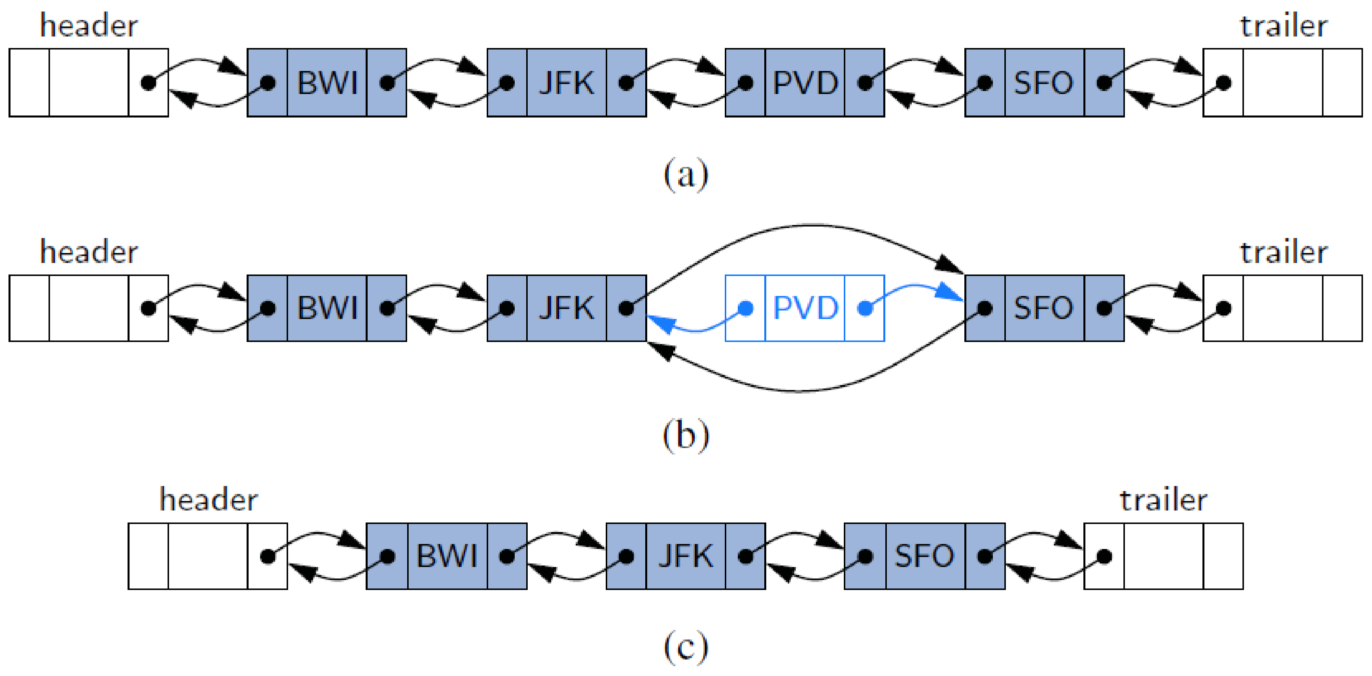


Figure 3.22: Removing the element PVD from a doubly linked list: (a) before the removal; (b) after linking out the old node; (c) after the removal (and garbage collection).

- Read section 3.4.1 for more details of implementing

List Case Study

Questions?