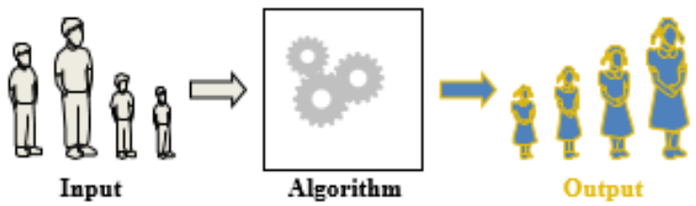
Data Structures & Algorithms

Adil M. Khan
Professor of Computer Science
Innopolis University

Recap -1

Algorithm



Analyzing Algorithms

- How do we analyze algorithms?

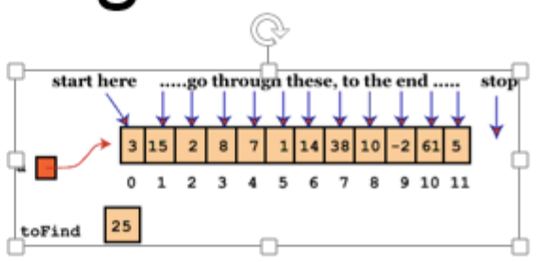
  Complexity Analysis: predicting the resources that an algorithm requires!

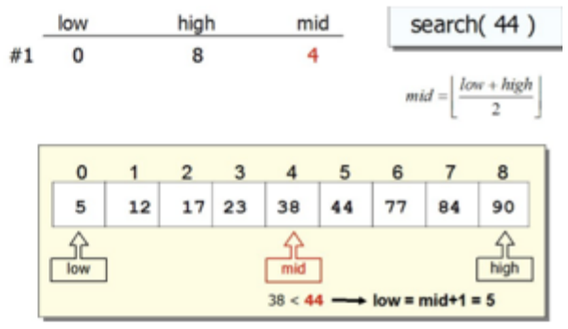
- ***Time Complexity:*** amount of time that an algorithm takes to run to completion

- ***Space Complexity:*** amount of memory that an algorithm needs to run to completion

Why Analyze Algorithms?

- Allows us to:
  - Compare the merits of two alternative approaches to a problem we need to solve
  - Determine whether a proposed solution will meet required resource constraints before we invest money and time coding





How to Measure Time Complexity?

- Analyze running time for the
  - ***best case:*** usually useless
  - ***average case:*** very difficult to determine
  - ***worst case:*** a safer choice

Recap -2

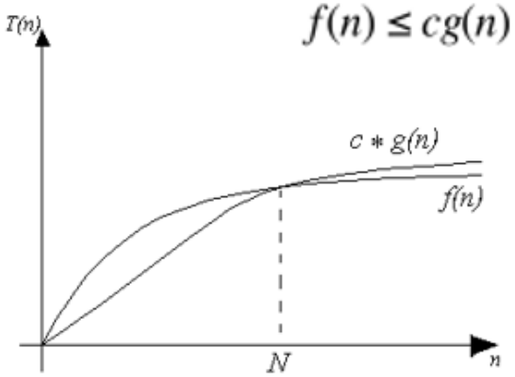
Measuring Time Complexity

- To measure the time complexity
  - We count the total number of primitive operations for an algorithm as a function of the input size ***T(n)***
  - Analyze growth rate of ***T(n)***

Big-Oh Notation

- In other words, ***f(n)*** is bounded above by a constant times of ***g(n)***

$f(n) \leq cg(n)$ *for all* $n \geq k$



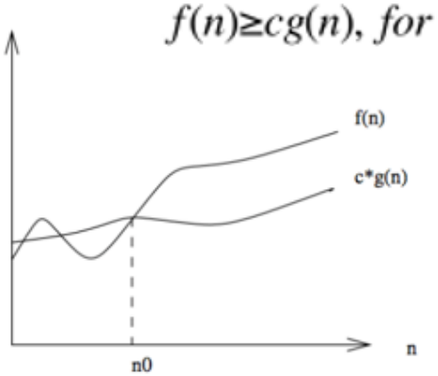
Here, ***N*** is representing ***k***

Big-Omega

- In other words, ***f(n)*** is bounded below by a constant times of ***g(n)***

$f(n) \geq cg(n)$, *for all* $n \geq k$

Ω



Here, ***n0*** is representing ***k***

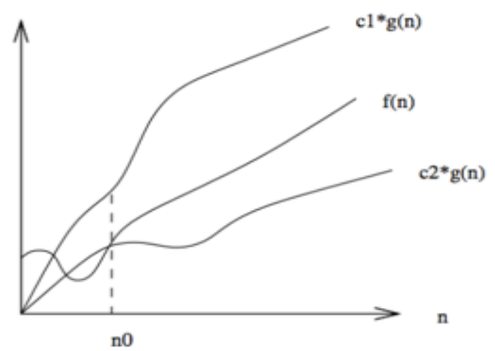
- Lower bounds are useful because they say that an algorithm requires **at least** so much time

Big-Theta

- In other words, ***f(n)*** is bounded above by **c1** times of ***g(n)*** and below by **c2** times of ***g(n)***

$f(n) \leq c_1 g(n)$, *and*, $f(n) \geq c_2 g(n)$ *for all* $n \geq k$

Θ



Here, ***n0*** is representing ***k***

Recap -3 (Correction)

```
1    procedure fibonacci {print nth term}
2       read(n)
3       if n<0
4          then print(error)
5          else if n=0
6             then print(0)
7             else if n=1
8                then print(1)
9             else
10               fnm2 := 0;
11               fnm1 := 1;
12               FOR i := 2 to n DO
13                  fn := fnm1 + fnm2;
14                   fnm2 := fnm1;
15                   fnm1 := fn
16                end
17                print(fn);
```

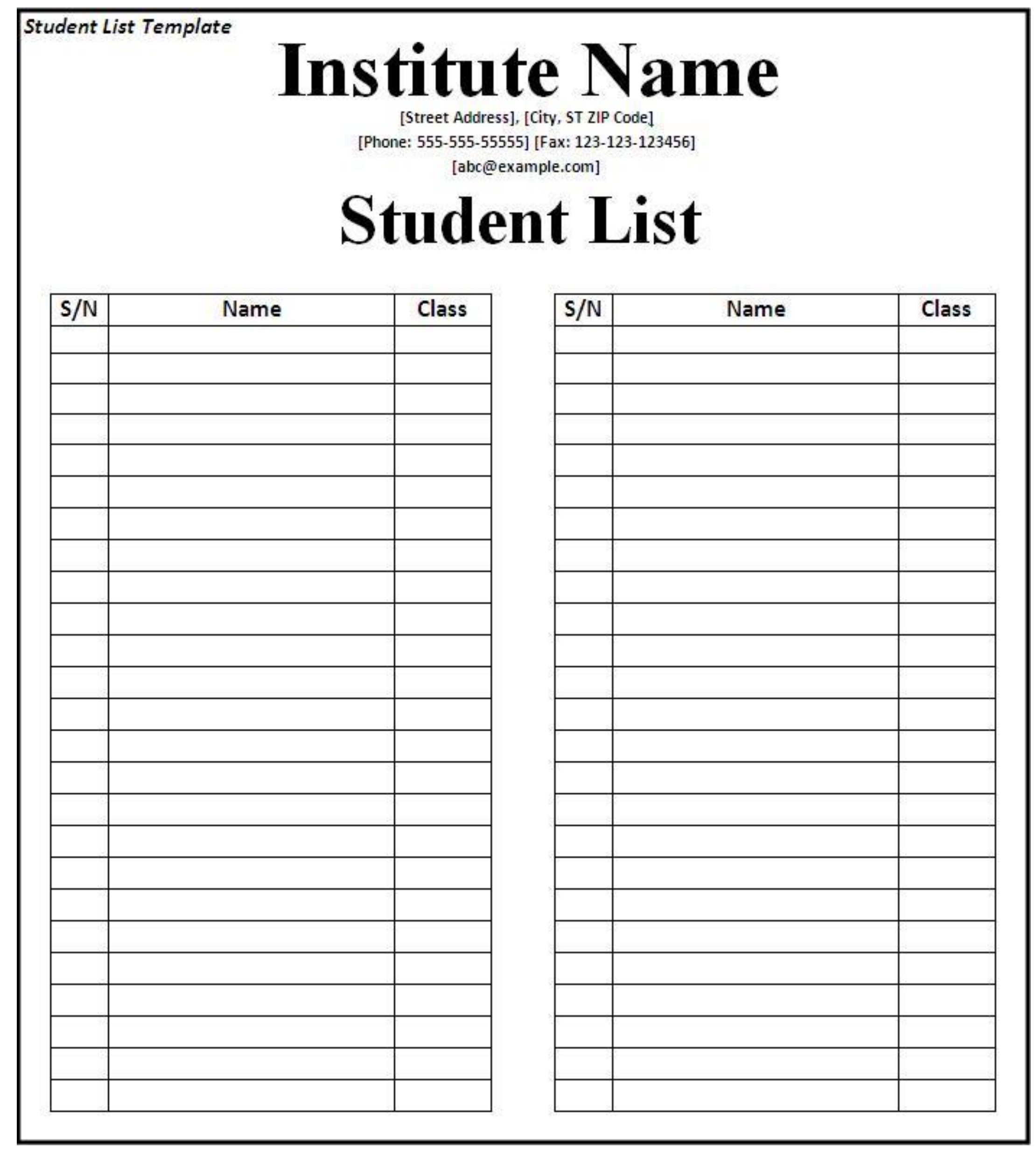
step	n>1
1	1
2	1
3	1
4	0
5	1
6	0
7	1
8	0
9	1
10	1
11	1
12	n
13	n-1
14	n-1
15	n-1
16	n-1
17	1

$T(n) = 5n + 5$

T(n) is ~~O(n)~~

Lists

Keeping Track of Items



Student List Template

Institute Name

[Street Address], [City, ST ZIP Code]
[Phone: 555-555-55555] [Fax: 123-123-123456]
[abc@example.com]

Student List

S/N	Name	Class	S/N	Name	Class

Basic Operations

- Insert an item
- Check to see if a particular item is present
- Delete an item

List as a Data Structure

- Attendance monitoring system (List of attendees)
- Computer Games (List of top scores)
- Online shopping (List of selected items)

List Basics

- Completely unrestricted sequence of zero or more items
  - add, update, and remove items at any position
  - lookup an item by position
  - find the index at which a given value appears

Basics

Some of The Important List Operations	
add(x)	add an item at the end
addFirst(x)	add an item at the beginning
addLast(x)	add an item at the end
add(i, x)	add an item at position i
remove(x)	remove the item from the list
remove(i)	remove the item at position I
size()	return the number of items in the list
isEmpty()	return whether the list has no items
contains(x)	return whether x is in the list
…	
.	…

Give some more operations as an exercise!

List ADT

```
1  /** A simplified version of the java.util.List interface. */
2  public interface List<E> {
3    /** Returns the number of elements in this list. */
4    int size( );
5
6    /** Returns whether the list is empty. */
7    boolean isEmpty( );
8
9    /** Returns (but does not remove) the element at index i. */
10   E get(int i) throws IndexOutOfBoundsException;
11
12   /** Replaces the element at index i with e, and returns the replaced element. */
13   E set(int i, E e) throws IndexOutOfBoundsException;
14
15   /** Inserts element e to be at index i, shifting all subsequent elements later. */
16   void add(int i, E e) throws IndexOutOfBoundsException;
17
18   /** Removes/returns the element at index i, shifting subsequent elements earlier. */
19   E remove(int i) throws IndexOutOfBoundsException;
20 }
```

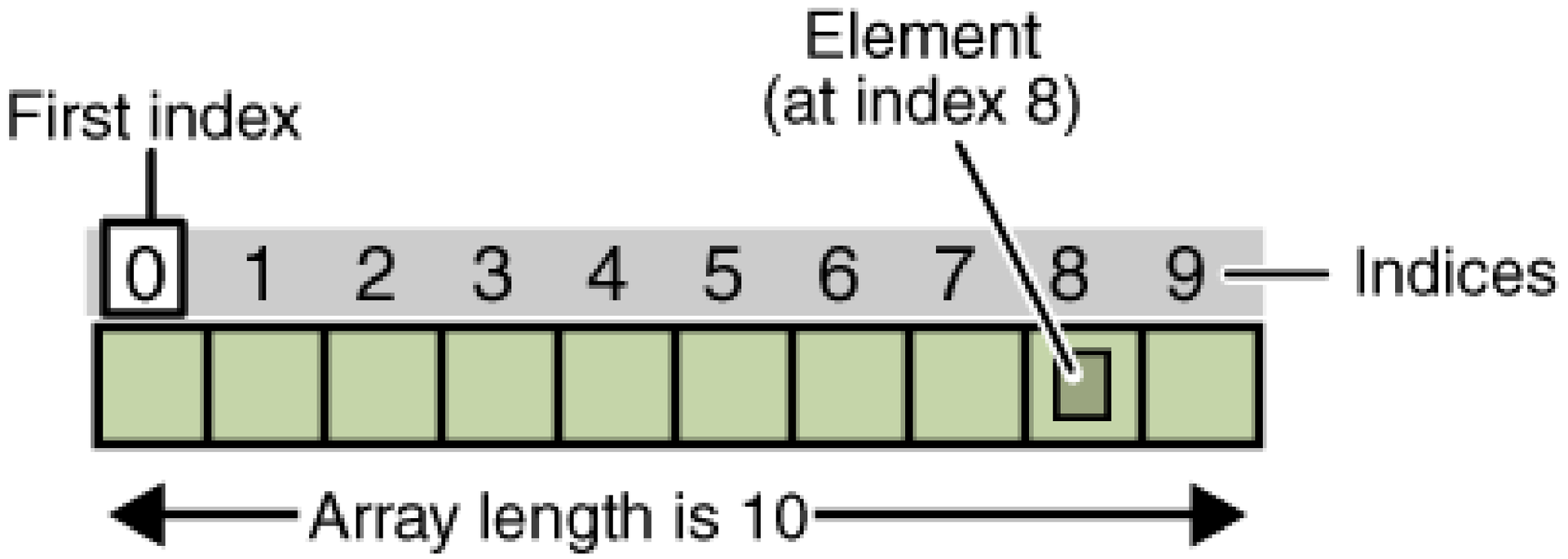
Code Fragment 7.1: A simple version of the List interface.

The key idea is that we have not specified how the list is to be implemented

Implementation

- **Two main representations**
  - **Array-based**
    - uses a static data structure
    - reasonable if we know in advance the maximum number of the items in the list
  - **Linked-list based**
    - uses dynamic data structure
    - best if don’t know in advance the number of elements in the list (or if it varies greatly)

Array


Element
(at index 8)
First index
0
1
2
3
4
5
6
7
8
9
Indices
Array length is 10

Java Arrays Basics

```
int [ ] intArray; // defines a reference to an array
intArray = new int[100]; // creates the array, and
                         // sets intArray to refer to it
```

Java Arrays Basics

```
int [ ] intArray = new int[100]; // equivalent single-
                                 // statement approach
```

```
int arrayLength = intArray.length; // find array length
```

Java Arrays Basics

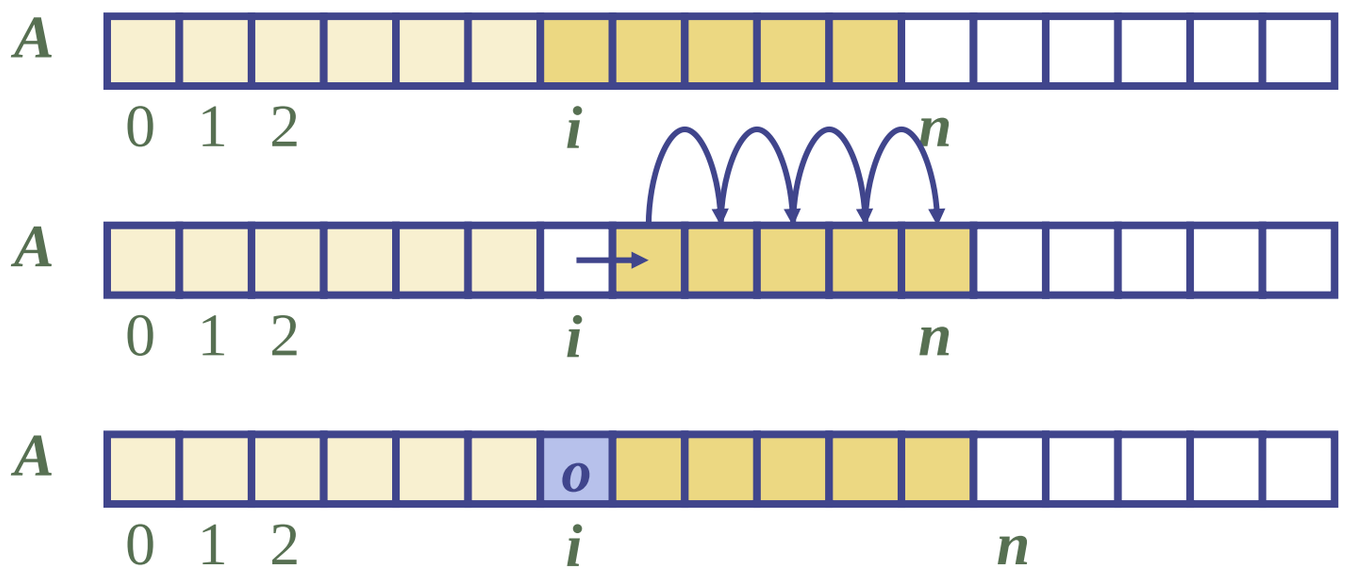
```
int temp = intArray[5]; // get contents of the sixth
                        // element of the array

intArray[6] = 46; // insert 46 into the seventh cell
```

Array-based Lists

- An obvious choice for implementing the list ADT is to use an array, A,
  - where **A[i]** stores (a reference to) the element with index i.
- With a representation based on an array A, the get(i) and set(i, e) methods are easy to implement by accessing A[i] (assuming i is a legitimate index).

Array-based Lists

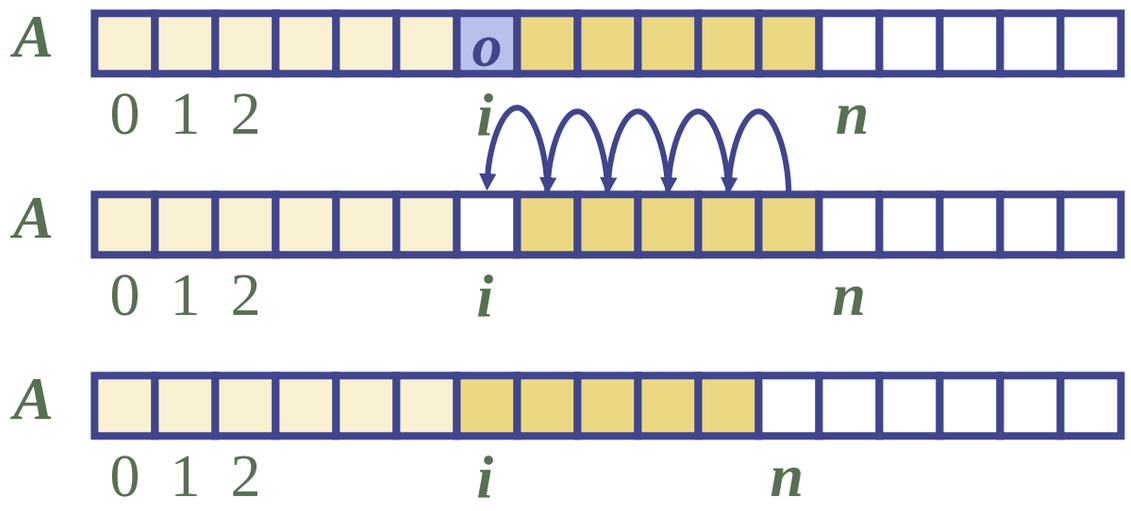
- Insertion at index i — add(i, o)
  - shifting forward the $n - i$ items — worst case $i = 0$ [takes ***O(n)*** time]



What about insertion at the end? — add(o)
How can we implement add(o) using add(i, o)?

Array-based Lists

- Removal at index i — remove(i)
  - shifting backwards the $n - i - 1$ items — worst case $i = 0$ [takes ***O(n)*** time]



Array-based Lists

```
1   public class ArrayList<E> implements List<E> {
2     // instance variables
3     public static final int CAPACITY=16;      // default array capacity
4     private E[ ] data;                        // generic array used for storage
5     private int size = 0;                     // current number of elements
6     // constructors
7     public ArrayList( ) { this(CAPACITY); }   // constructs list with default capacity
8     public ArrayList(int capacity) {          // constructs list with given capacity
9       data = (E[ ]) new Object[capacity];     // safe cast; compiler may give warning
10    }
```

Array-based Lists

```
11    // public methods
12    /** Returns the number of elements in the array list. */
13    public int size( ) { return size; }
14    /** Returns whether the array list is empty. */
15    public boolean isEmpty( ) { return size == 0; }
16    /** Returns (but does not remove) the element at index i. */
17    public E get(int i) throws IndexOutOfBoundsException {
18      checkIndex(i, size);
19      return data[i];
20    }
21    /** Replaces the element at index i with e, and returns the replaced element. */
22    public E set(int i, E e) throws IndexOutOfBoundsException {
23      checkIndex(i, size);
24      E temp = data[i];
25      data[i] = e;
26      return temp;
27    }
```

Array-based Lists

```
28    /** Inserts element e to be at index i, shifting all subsequent elements later. */
29    public void add(int i, E e) throws IndexOutOfBoundsException,
30                                                    IllegalStateException {
31      checkIndex(i, size + 1);
32      if (size == data.length)                      // not enough capacity
33        throw new IllegalStateException("Array is full");
34      for (int k=size-1; k >= i; k--)               // start by shifting rightmost
35        data[k+1] = data[k];
36      data[i] = e;                                  // ready to place the new element
37      size++;
38    }
39    /** Removes/returns the element at index i, shifting subsequent elements earlier. */
40    public E remove(int i) throws IndexOutOfBoundsException {
41      checkIndex(i, size);
42      E temp = data[i];
43      for (int k=i; k < size-1; k++)                // shift elements to fill hole
44        data[k] = data[k+1];
45      data[size-1] = null;                          // help garbage collection
46      size--;
47      return temp;
48    }
```

Array-based Lists

```
49    // utility method
50    /** Checks whether the given index is in the range [0, n−1]. */
51    protected void checkIndex(int i, int n) throws IndexOutOfBoundsException {
52      if (i < 0 || i >= n)
53        throw new IndexOutOfBoundsException("Illegal index: " + i);
54    }
55  }
```

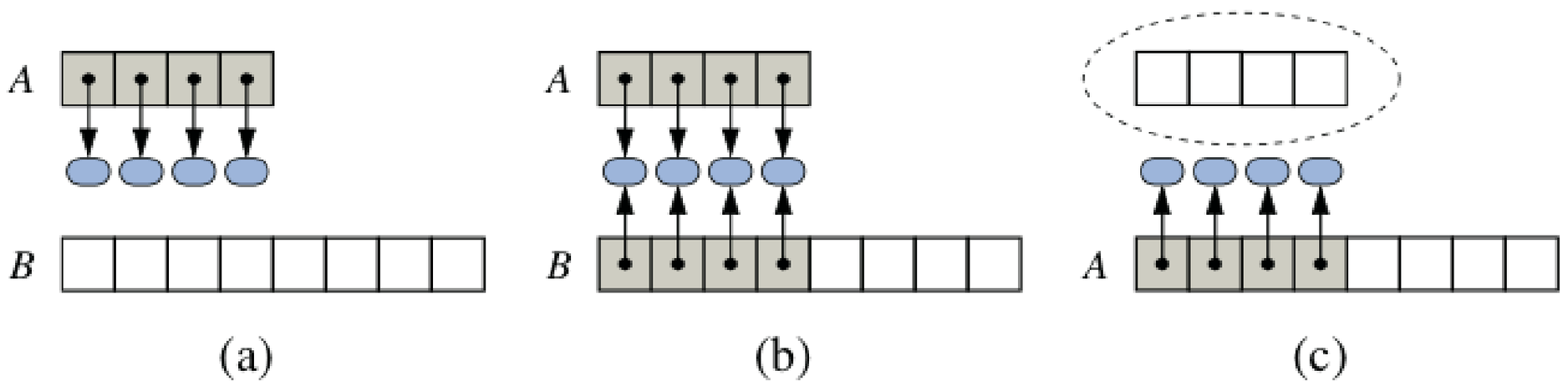
Array-based Lists

Method	Running Time
size( )	$O(1)$
isEmpty( )	$O(1)$
get(i)	$O(1)$
set(i, e)	$O(1)$
add(i, e)	$O(n)$
remove(i)	$O(n)$

Table 7.1: Performance of an array list with n elements realized by a fixed-capacity array.

Dynamic Arrays-based Lists

- When the array is full, we replace the array with a larger one



An illustration of "growing" a dynamic array: (a) create new array *B*; (b) store elements of *A* in *B*; (c) reassign reference *A* to the new array.

Dynamic Array-based Lists

```
/** Resizes internal array to have given capacity >= size. */
protected void resize(int capacity) {
  E[ ] temp = (E[ ]) new Object[capacity]; // safe cast; compiler may give warning
  for (int k=0; k < size; k++)
    temp[k] = data[k];
  data = temp;                           // start using the new array
}
```

A concrete implementation of a resize method, which should be included as a protected method within the original ArrayList class. The instance variable **data** corresponds to array *A* in the discussion (previous slide), and local variable **temp** corresponds to array *B*.

Dynamic Arrays-based Lists

- How large should the new array be?
  - ***Doubling strategy:*** double the size

```
28    /** Inserts element e to be at index i, shifting all subsequent elements later. */
29    public void add(int i, E e) throws IndexOutOfBoundsException {
30      checkIndex(i, size + 1);
31      if (size == data.length)                    // not enough capacity
32        resize(2 * data.length);                  // so double the current capacity
...     // rest of method unchanged...
```

Code Fragment 7.5: A revision to the ArrayList.add method, originally from Code Fragment 7.3, which calls the resize method of Code Fragment 7.4 when more capacity is needed.

Dynamic Arrays-based Lists

- Doubling Strategy

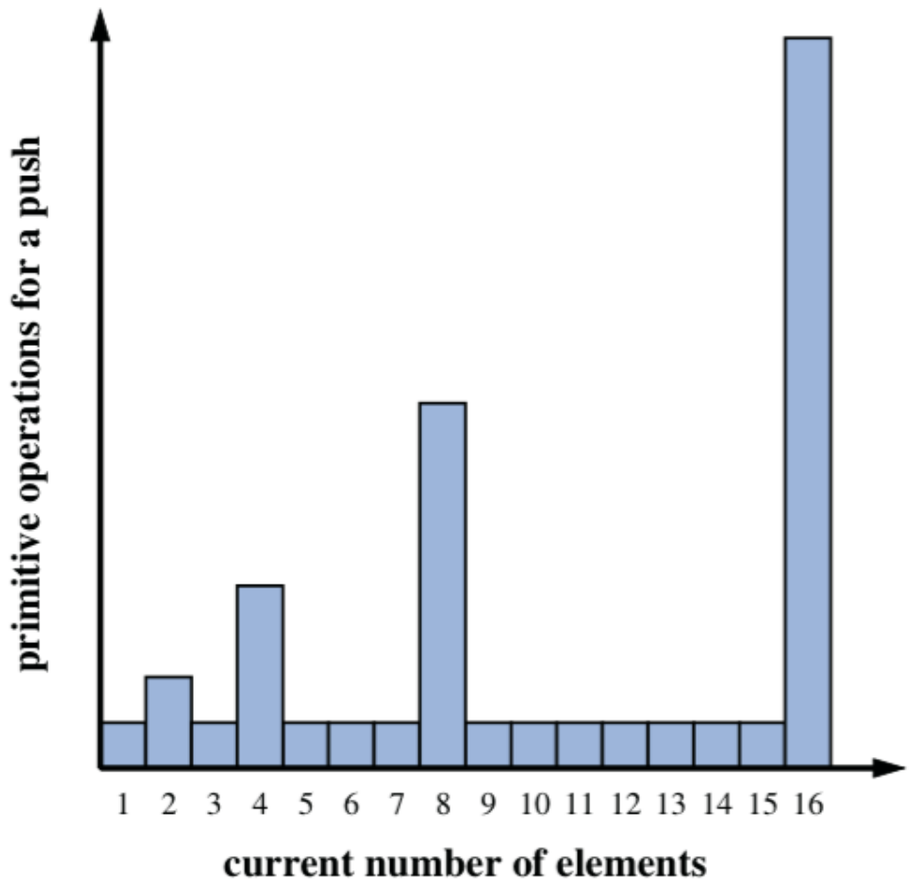


Figure 7.4: Running times of a series of push operations on a dynamic array.

Array vs. Array-based Lists

```
int [ ] intArray = new int[100];
```

```
ArrayList<Integer> arrayList = new ArrayList<Integer>;
```

What do you think is the difference between these two?

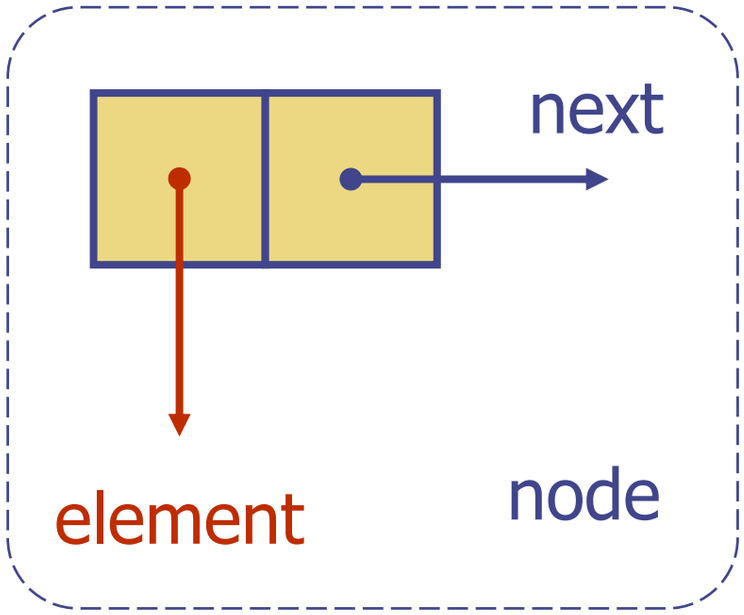
Implementation

- Two main representations
  - Array-based
    - uses a static data structure
    - reasonable if we know in advance the maximum number of the items in the list
  - **Linked-list based**
    - **uses dynamic data structure**
    - **best if don’t know in advance the number of elements in the list (or if it varies greatly)**

Singly Linked List

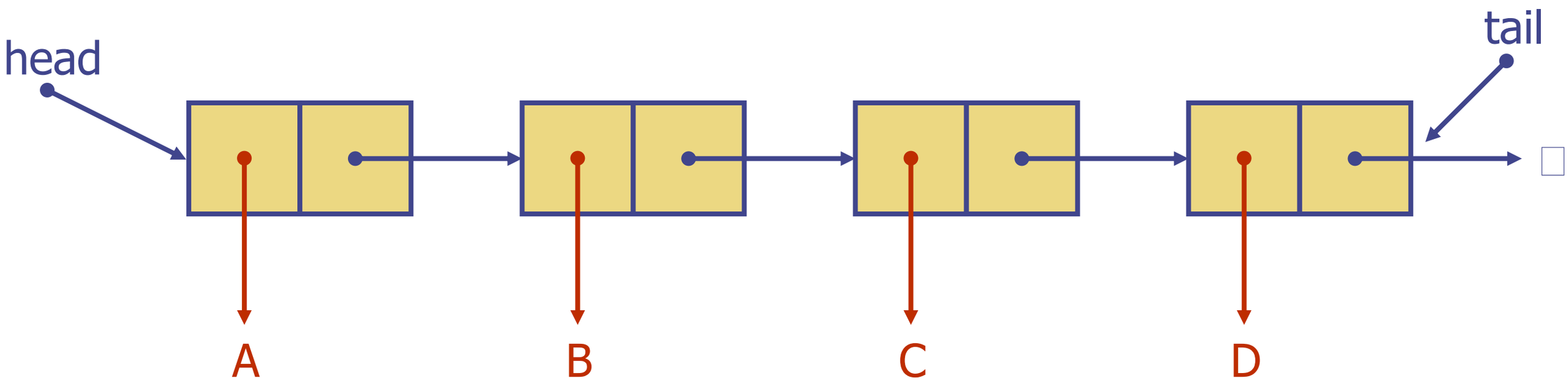
Singly Linked List

A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer



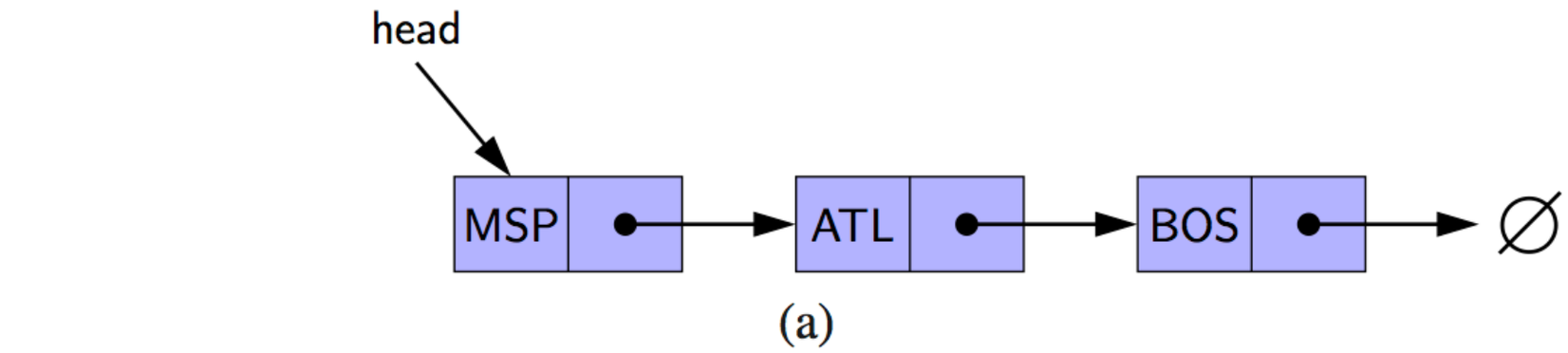
Each node stores

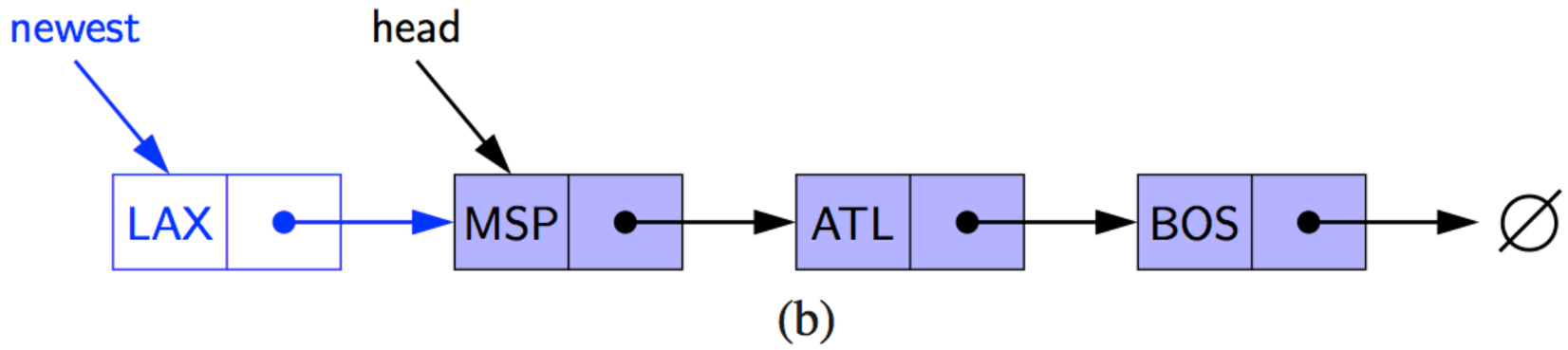
- element
- link to the next node

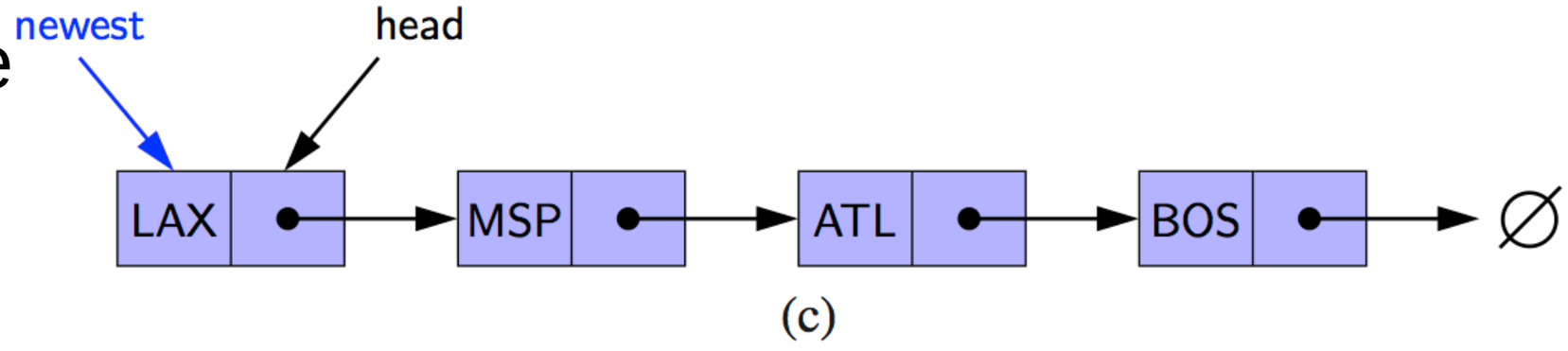


Addition to the Front

- Allocate new node
- Insert new element
- Have new node point to old head
- Update head to point to new node



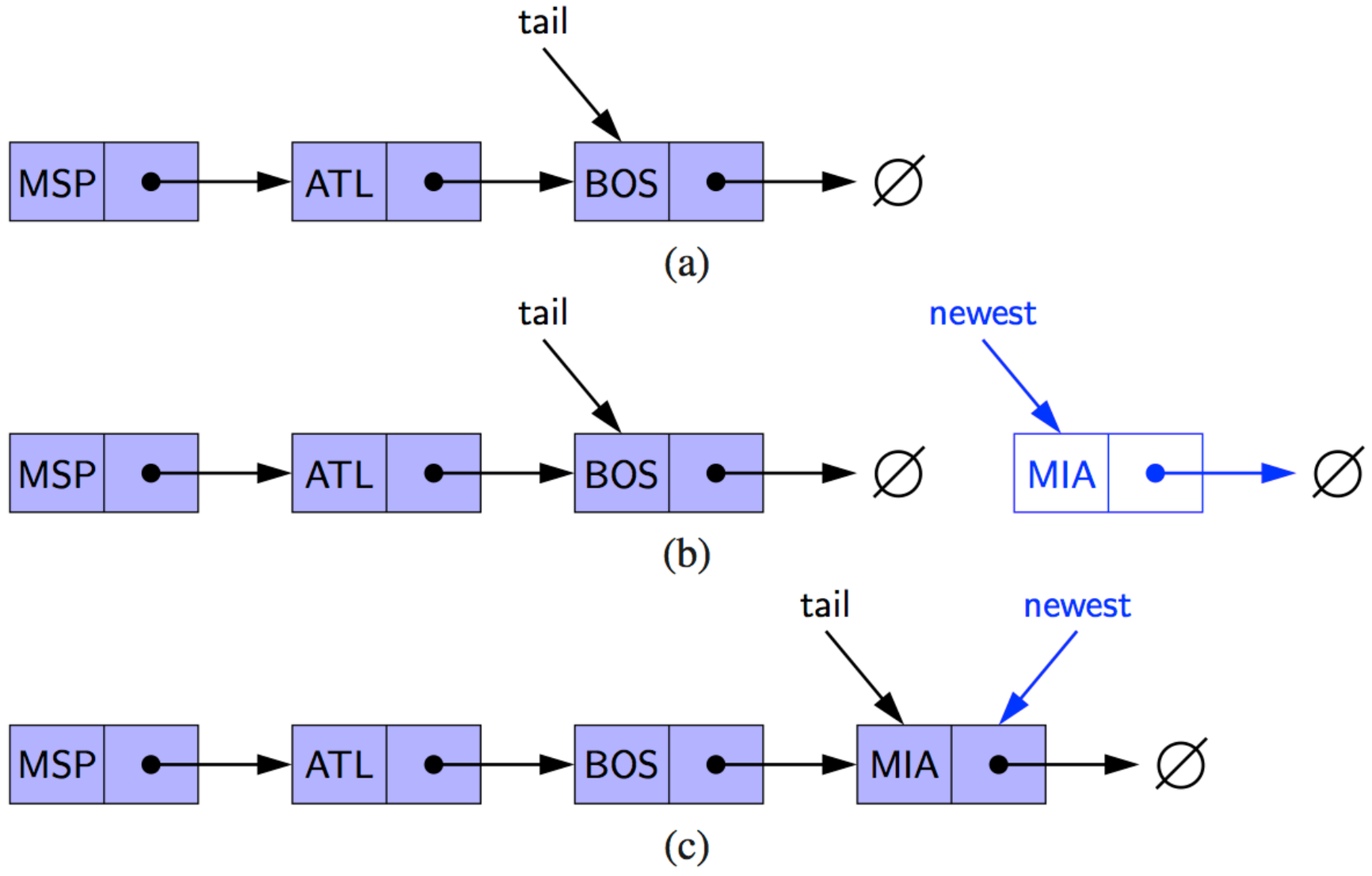




Time Complexity?

Addition to the Back

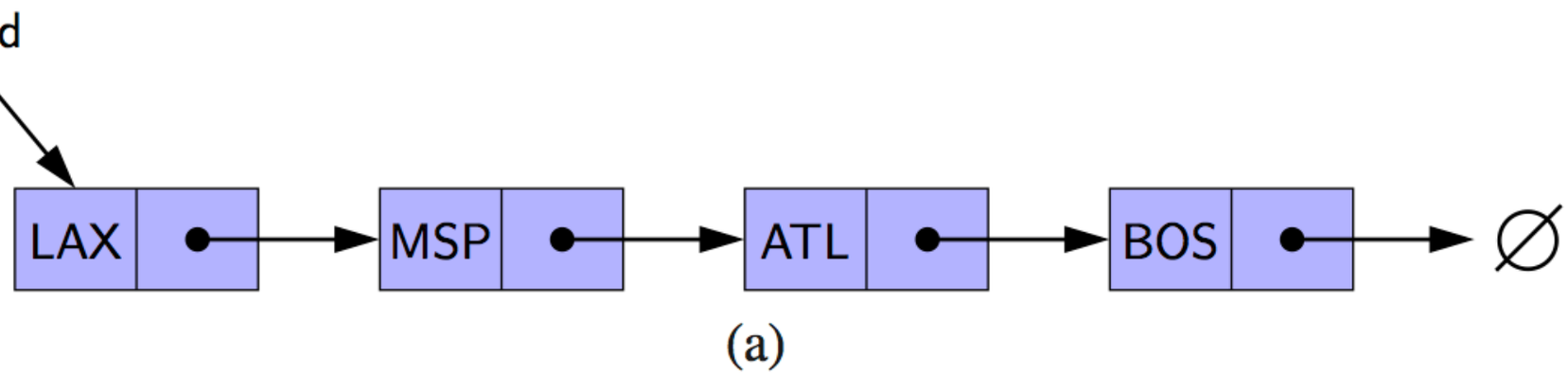
- Allocate a new node
- Insert new element
- Have new node point to null
- Have old last node point to new node
- Update tail to point to new node



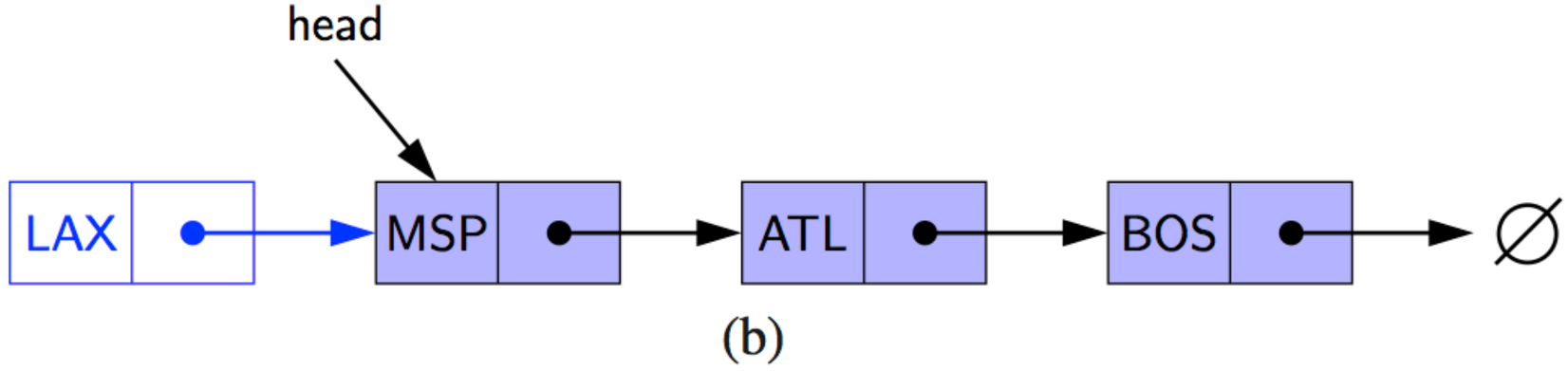
Time Complexity?

Removal from the Front

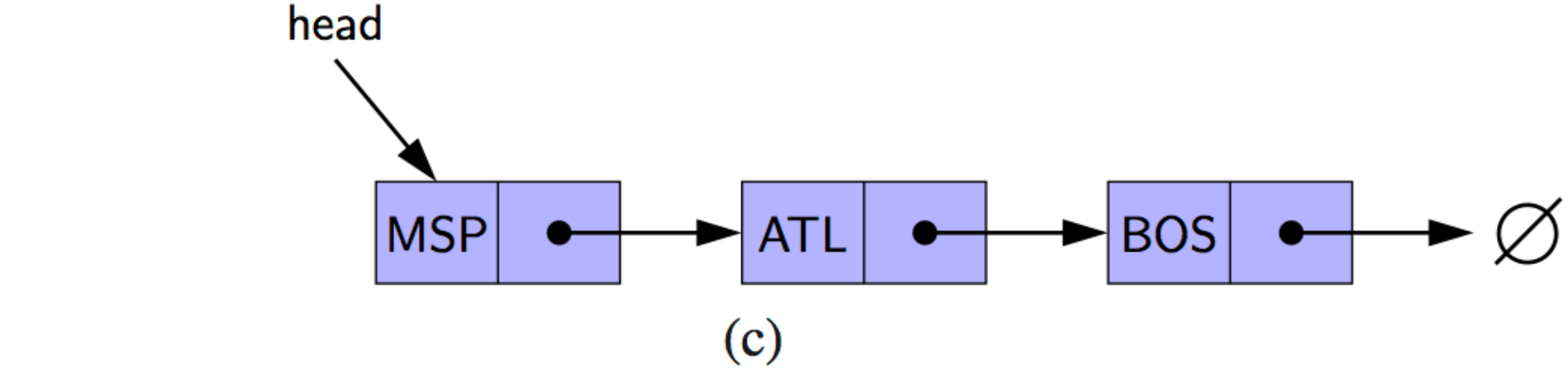
- Update head to point to next node in the list
- Allow garbage collector to reclaim the former first node



(a)



(b)

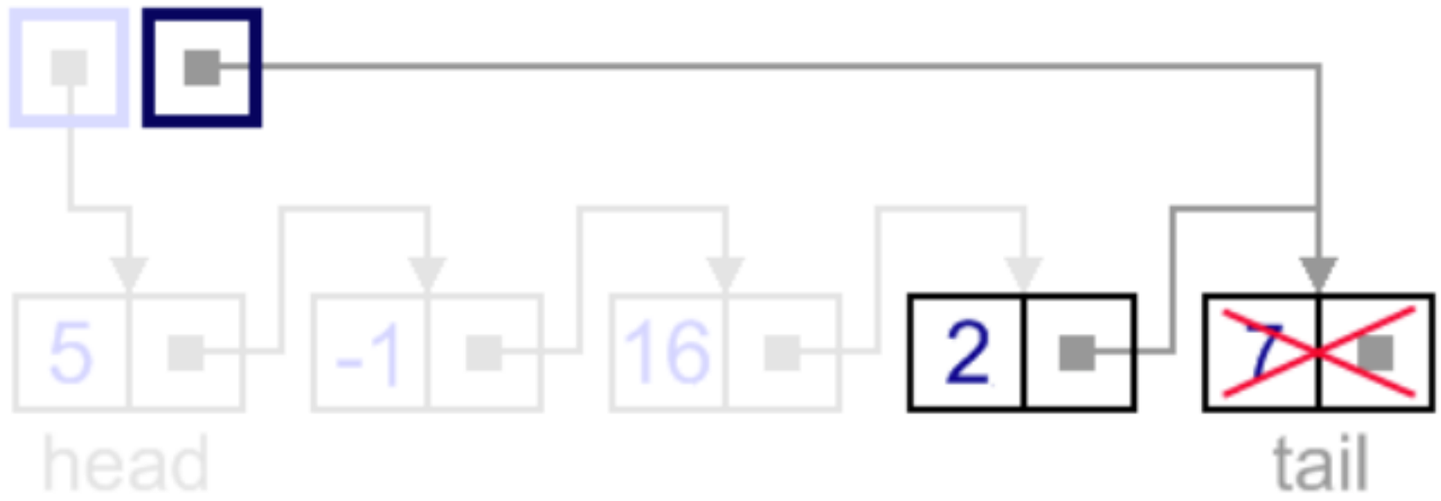


(c)

Time Complexity?

Removal from the Back

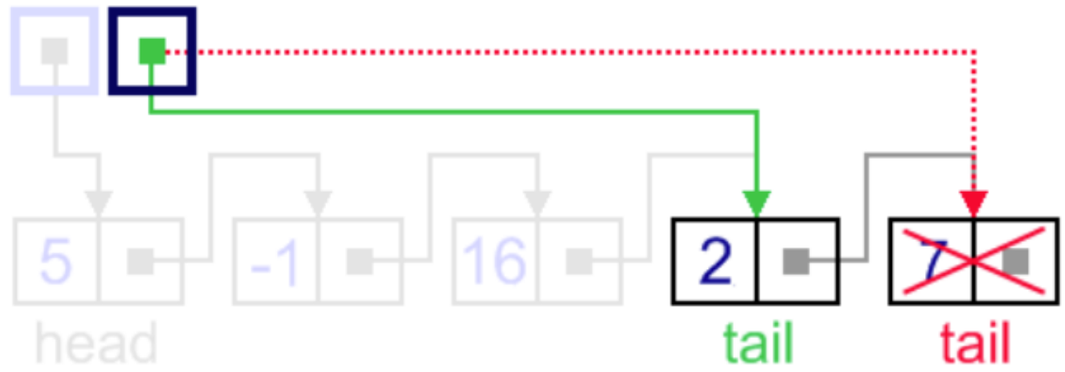
- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node
  - you must find the node before the tail iteratively



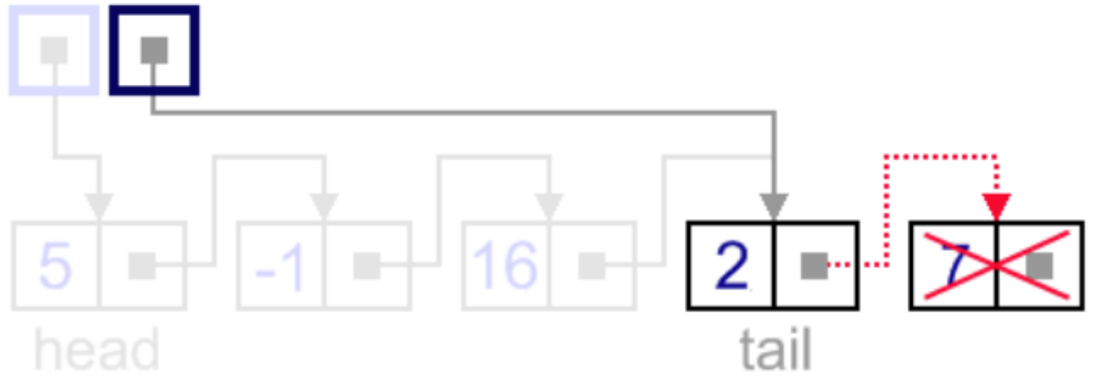
Time Complexity?

Removal from the Back

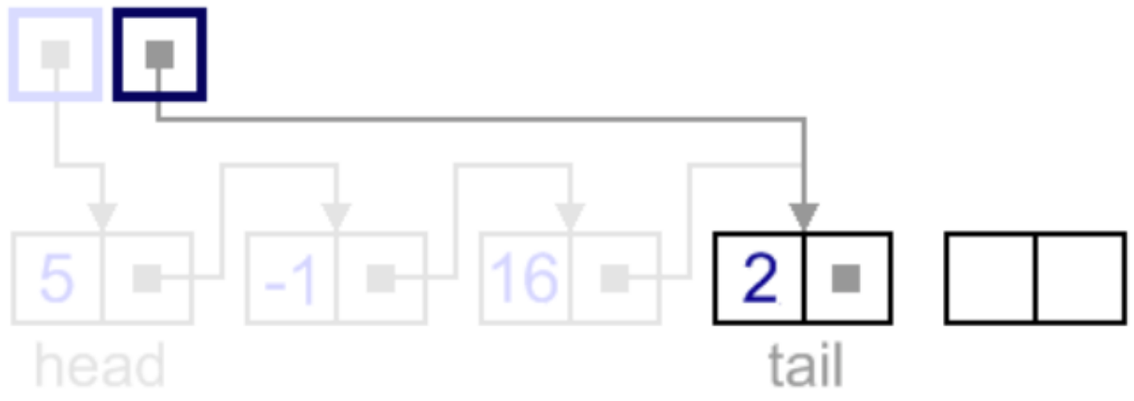
1



2

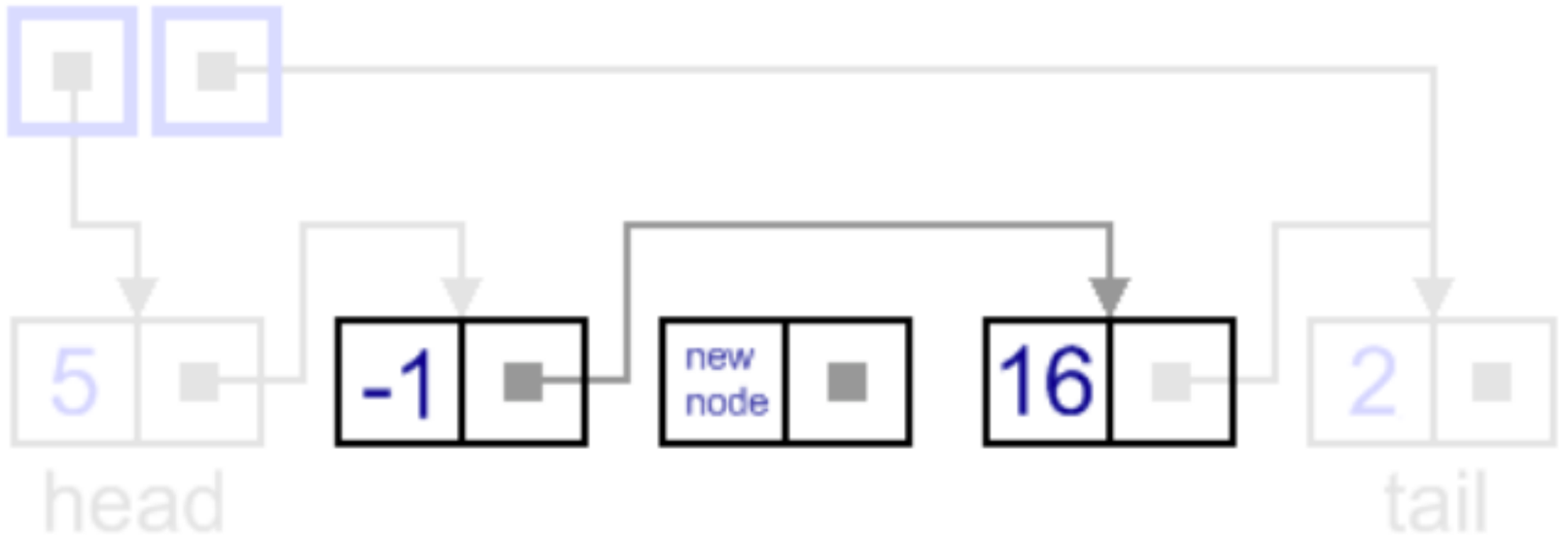


3



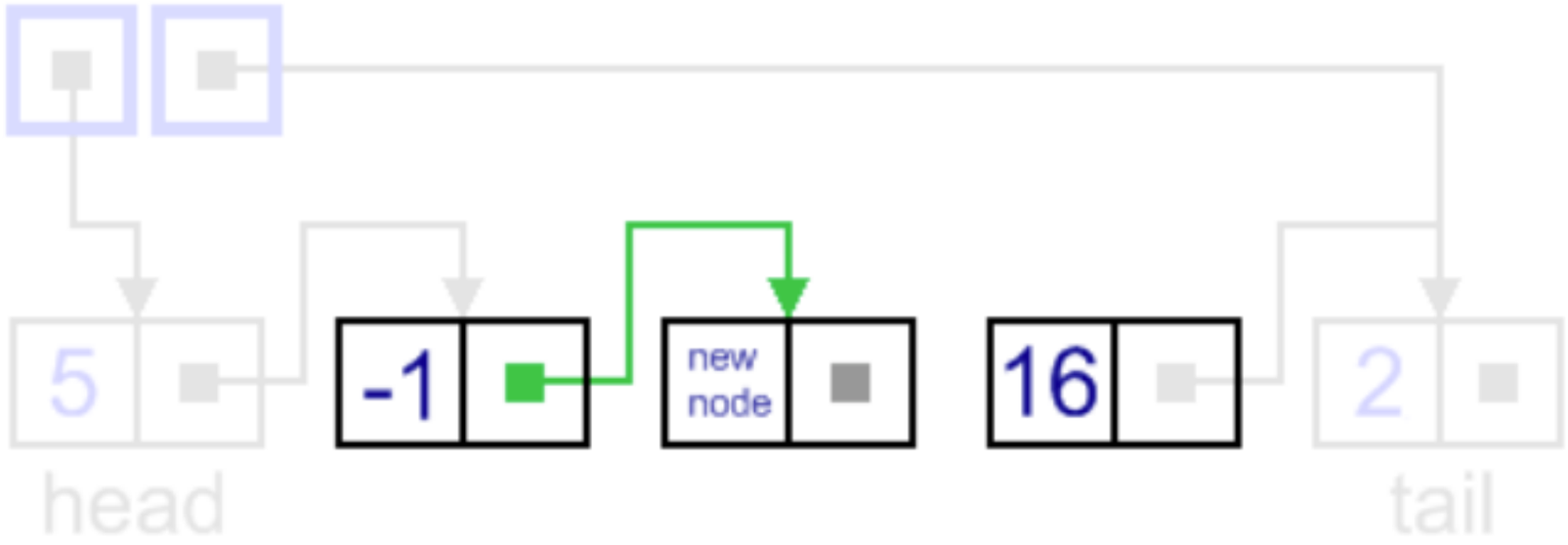
Addition between Nodes

- Inserted between two nodes

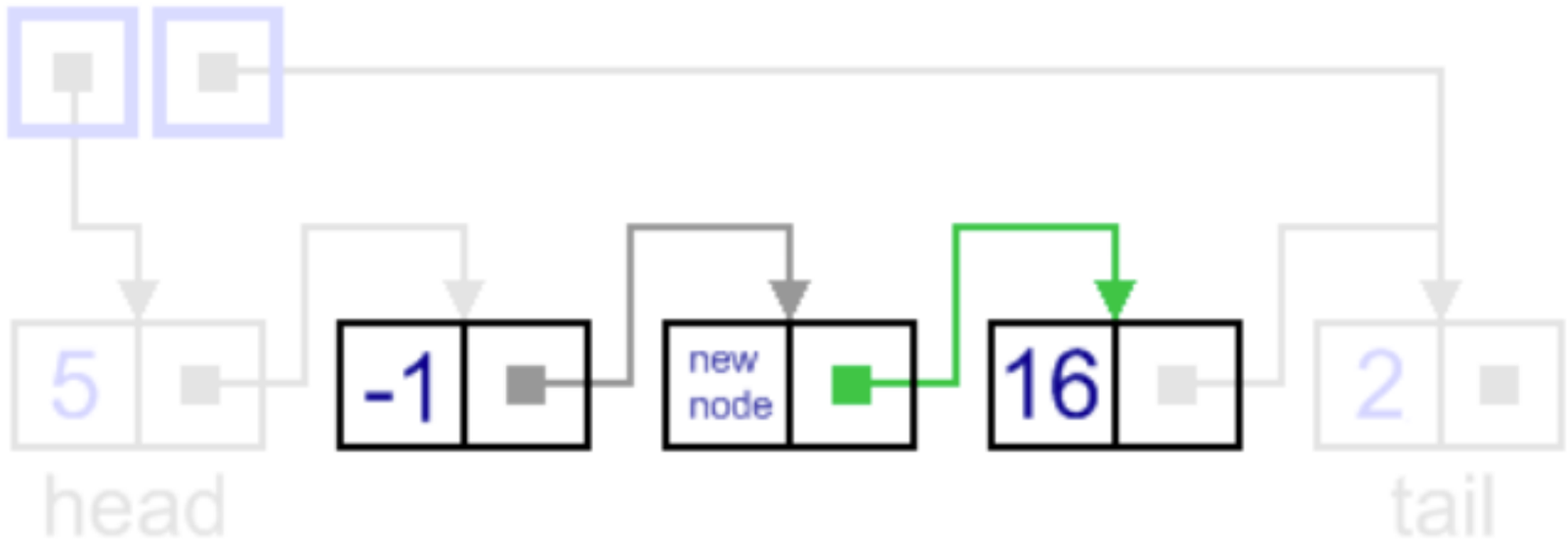


Addition between Nodes

1



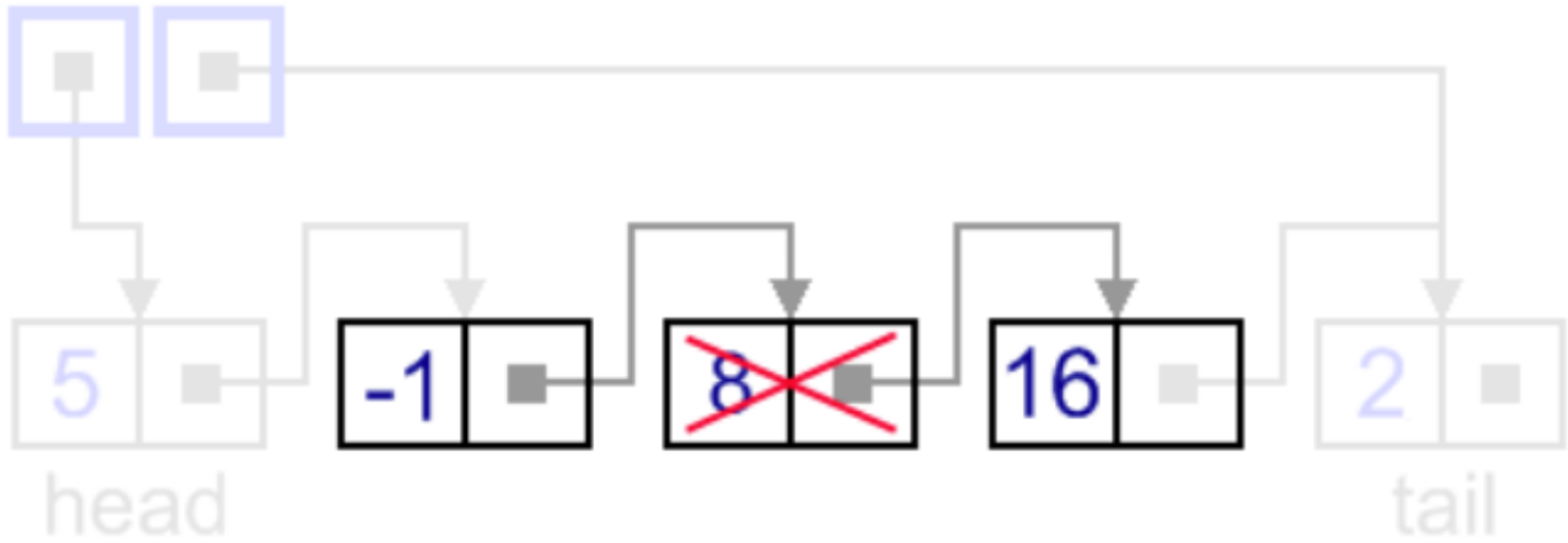
2



Time Complexity?

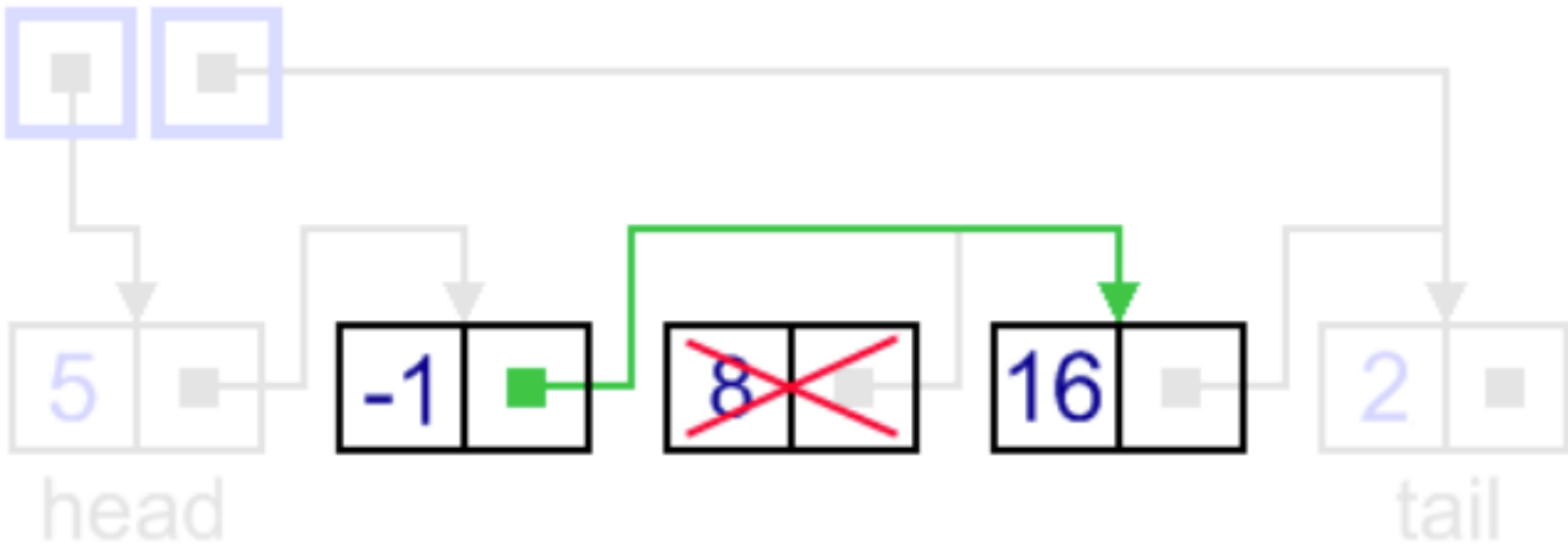
Removal between Nodes

- Node located between two nodes

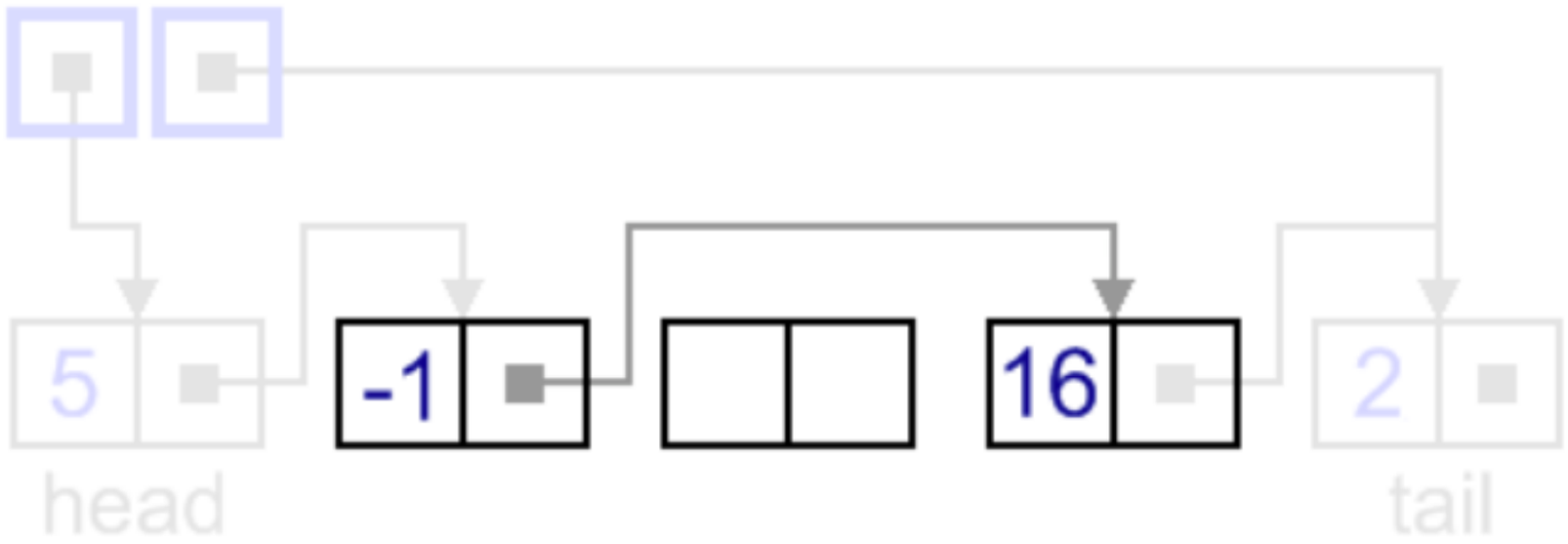


Removal between Nodes

1



2



Time Complexity?

Singly Linked List Implementation

- Next few slides will present a complete implementation of a SinglyLinkedList supporting the following 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.

Singly Linked List

```
1   public class SinglyLinkedList<E> {
2     //---------------- nested Node class ----------------
3     private static class Node<E> {
4       private E element;                  // reference to the element stored at this node
5       private Node<E> next;               // reference to the subsequent node in the list
6       public Node(E e, Node<E> n) {
7         element = e;
8         next = n;
9       }
10      public E getElement( ) { return element; }
11      public Node<E> getNext( ) { return next; }
12      public void setNext(Node<E> n) { next = n; }
13    } //---------- end of nested Node class ----------
```

Singly Linked List

```
14    // instance variables of the SinglyLinkedList
15    private Node<E> head = null;          // head node of the list (or null if empty)
16    private Node<E> tail = null;          // last node of the list (or null if empty)
17    private int size = 0;                 // number of nodes in the list
18    public SinglyLinkedList( ) { }        // constructs an initially empty list
19    // access methods
20    public int size( ) { return size; }
21    public boolean isEmpty( ) { return size == 0; }
22    public E first( ) {                   // returns (but does not remove) the first element
23      if (isEmpty( )) return null;
24      return head.getElement( );
25    }
26    public E last( ) {                    // returns (but does not remove) the last element
27      if (isEmpty( )) return null;
28      return tail.getElement( );
29    }
```

Singly Linked List

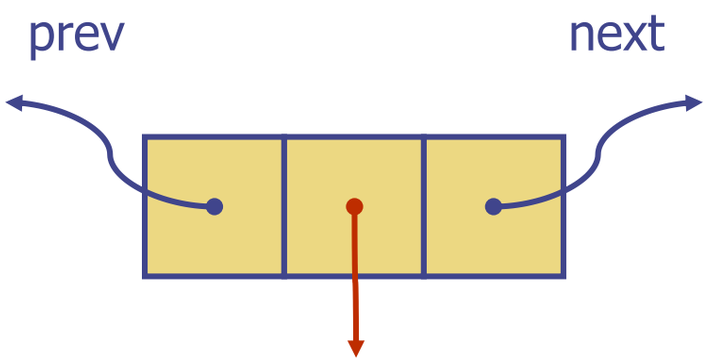
```
30    // update methods
31    public void addFirst(E e) {                  // adds element e to the front of the list
32      head = new Node<>(e, head);                // create and link a new node
33      if (size == 0)
34        tail = head;                             // special case: new node becomes tail also
35      size++;
36    }
37    public void addLast(E e) {                   // adds element e to the end of the list
38      Node<E> newest = new Node<>(e, null);      // node will eventually be the tail
39      if (isEmpty())
40        head = newest;                           // special case: previously empty list
41      else
42        tail.setNext(newest);                    // new node after existing tail
43      tail = newest;                             // new node becomes the tail
44      size++;
45    }
```

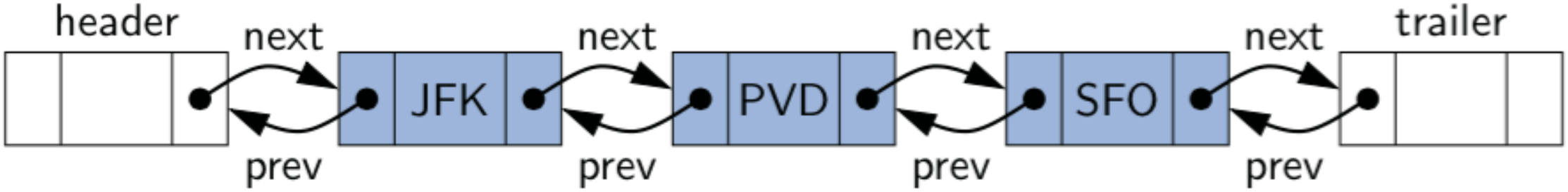
Singly Linked List

```
46    public E removeFirst( ) {                    // removes and returns the first element
47      if (isEmpty( )) return null;               // nothing to remove
48      E answer = head.getElement( );
49      head = head.getNext( );                    // will become null if list had only one node
50      size−−;
51      if (size == 0)
52        tail = null;                             // special case as list is now empty
53      return answer;
54    }
55  }
```

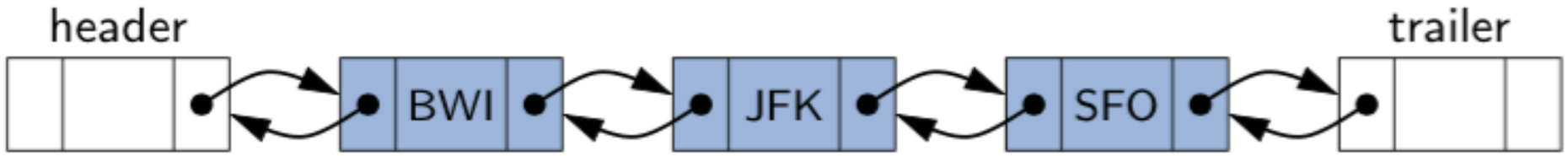
Doubly Linked List

Doubly Linked List

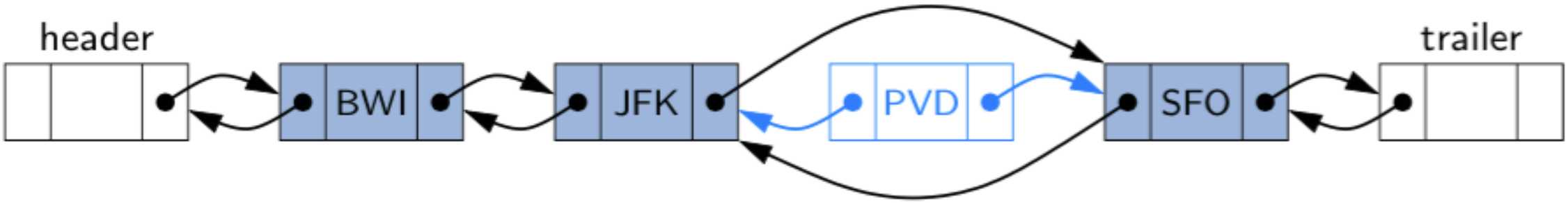




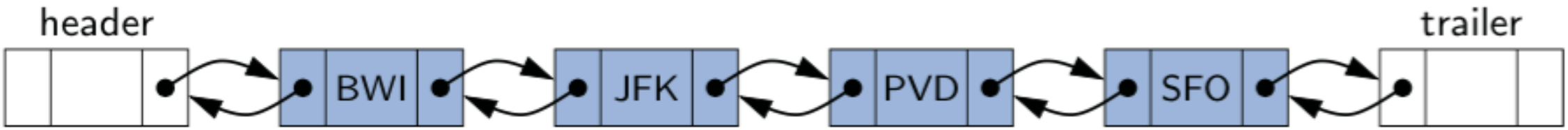
Addition between Nodes



(a)



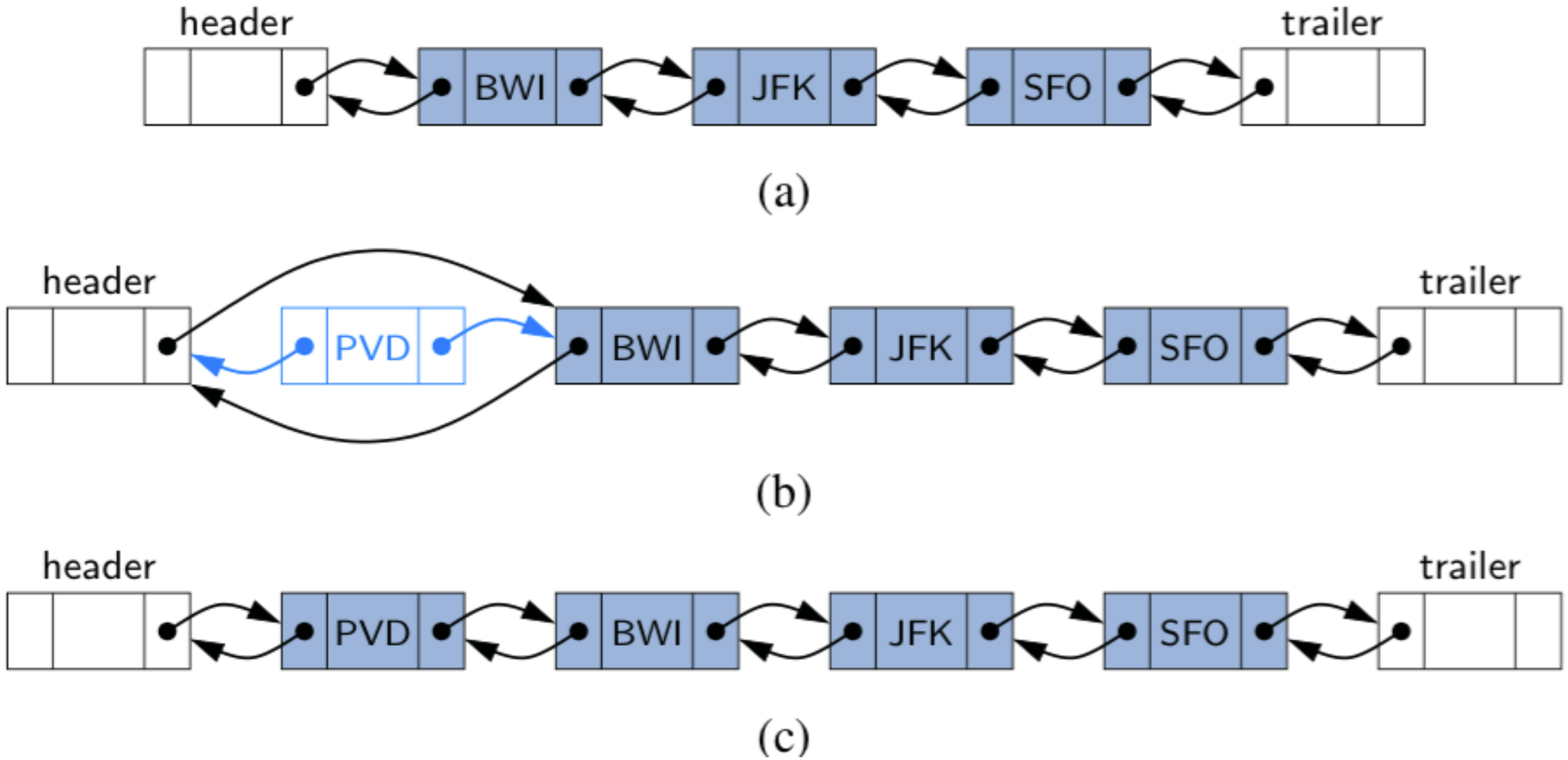
(b)



(c)

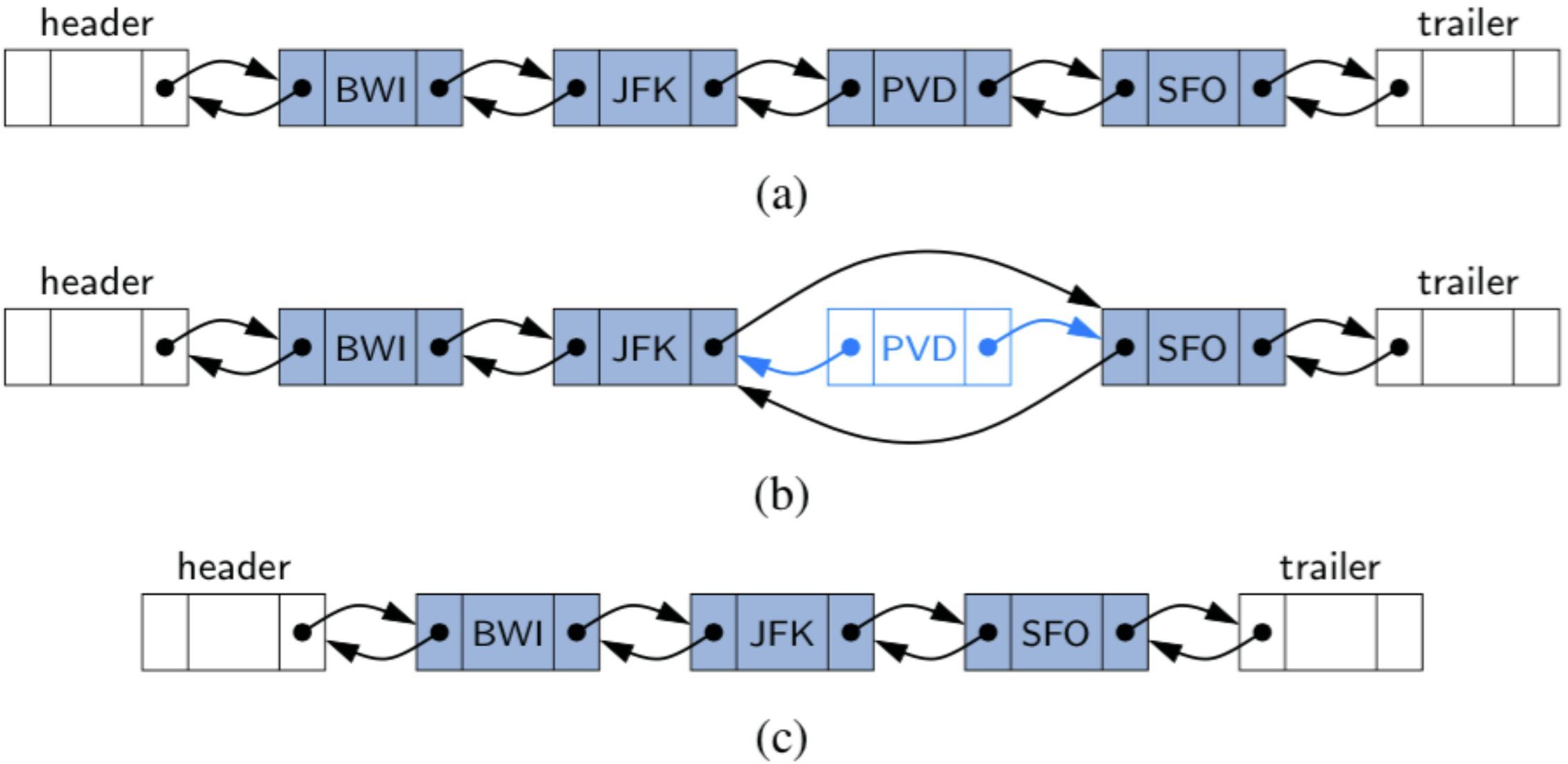
Adding an element to a doubly linked list: (a) before the operation; (b) after creating the new node; (c) after linking the neighbors to the new node.

Addition to the Front



Adding an element to the front of a doubly linked list: (a) before the operation; (b) after creating the new node; (c) after linking the neighbors to the new node.

Removal between Nodes



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

Double Linked List Implementation

- Next few slides will present a complete implementation of a DoublyLinkedList supporting the following 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.

removeLast( ): Removes and returns the last element of the list.

Double Linked List

```
1   /** A basic doubly linked list implementation. */
2   public class DoublyLinkedList<E> {
3     //---------------- nested Node class ----------------
4     private static class Node<E> {
5       private E element;                  // reference to the element stored at this node
6       private Node<E> prev;               // reference to the previous node in the list
7       private Node<E> next;               // reference to the subsequent node in the list
8       public Node(E e, Node<E> p, Node<E> n) {
9         element = e;
10        prev = p;
11        next = n;
12      }
13      public E getElement( ) { return element; }
14      public Node<E> getPrev( ) { return prev; }
15      public Node<E> getNext( ) { return next; }
16      public void setPrev(Node<E> p) { prev = p; }
17      public void setNext(Node<E> n) { next = n; }
18    } //---------- end of nested Node class ----------
19
```

Double Linked List

```
20    // instance variables of the DoublyLinkedList
21    private Node<E> header;                    // header sentinel
22    private Node<E> trailer;                   // trailer sentinel
23    private int size = 0;                      // number of elements in the list
24    /** Constructs a new empty list. */
25    public DoublyLinkedList( ) {
26      header = new Node<>(null, null, null);   // create header
27      trailer = new Node<>(null, header, null); // trailer is preceded by header
28      header.setNext(trailer);                  // header is followed by trailer
29    }
30    /** Returns the number of elements in the linked list. */
31    public int size( ) { return size; }
32    /** Tests whether the linked list is empty. */
33    public boolean isEmpty( ) { return size == 0; }
34    /** Returns (but does not remove) the first element of the list. */
35    public E first( ) {
36      if (isEmpty( )) return null;
37      return header.getNext( ).getElement( );   // first element is beyond header
38    }
```

Double Linked List

```
39      /** Returns (but does not remove) the last element of the list. */
40      public E last( ) {
41        if (isEmpty( )) return null;
42        return trailer.getPrev( ).getElement( );          // last element is before trailer
43      }
```

Double Linked List

```
44    // public update methods
45    /** Adds element e to the front of the list. */
46    public void addFirst(E e) {
47      addBetween(e, header, header.getNext( ));          // place just after the header
48    }
49    /** Adds element e to the end of the list. */
50    public void addLast(E e) {
51      addBetween(e, trailer.getPrev( ), trailer);        // place just before the trailer
52    }
53    /** Removes and returns the first element of the list. */
54    public E removeFirst( ) {
55      if (isEmpty( )) return null;                       // nothing to remove
56      return remove(header.getNext( ));                  // first element is beyond header
57    }
58    /** Removes and returns the last element of the list. */
59    public E removeLast( ) {
60      if (isEmpty( )) return null;                       // nothing to remove
61      return remove(trailer.getPrev( ));                 // last element is before trailer
62    }
```

Double Linked List

```
63
64      // private update methods
65      /** Adds element e to the linked list in between the given nodes. */
66      private void addBetween(E e, Node<E> predecessor, Node<E> successor) {
67        // create and link a new node
68        Node<E> newest = new Node<>(e, predecessor, successor);
69        predecessor.setNext(newest);
70        successor.setPrev(newest);
71        size++;
72      }
73      /** Removes the given node from the list and returns its element. */
74      private E remove(Node<E> node) {
75        Node<E> predecessor = node.getPrev( );
76        Node<E> successor = node.getNext( );
77        predecessor.setNext(successor);
78        successor.setPrev(predecessor);
79        size--;
80        return node.getElement( );
81      }
82    } //---------- end of DoublyLinkedList class ----------
```

Look at remove(e) method, and tell me what can go wrong?