Data Structures & Algorithms

Alexandr Klimchik
Head of Intelligent Robotic Systems Laboratory
Innopolis University

Based on the material from Adil M. Khan

Elementary Data Structures

Basic Classification

- Contiguously Allocated Structures
- Linked Data Structures

List ADT

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

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
…	
.	…

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

Generics

- We should be able to use collections for any type of data
- A specific Java mechanism known as Generics, also known as parameterized types, enables this capability
- The notation <Item> after the class name

Array List without Generics

```
public class ArrayList implements List {

    ….

    datatype [ ] data;

    ….

}
```

Array List with Generics

```
public class ArrayList<E> implements List<E> {

    ….
    E [ ] data;

    ….

}
```

Array List with Generics

```
ArrayList<String> stringList =
                    new ArrayList<String>();

stringList.add(“Kazan”);
```

```
ArrayList<Date> dateList =
                    new ArrayList<Date>();

dateList.add(new Date(12,31,1999));
```

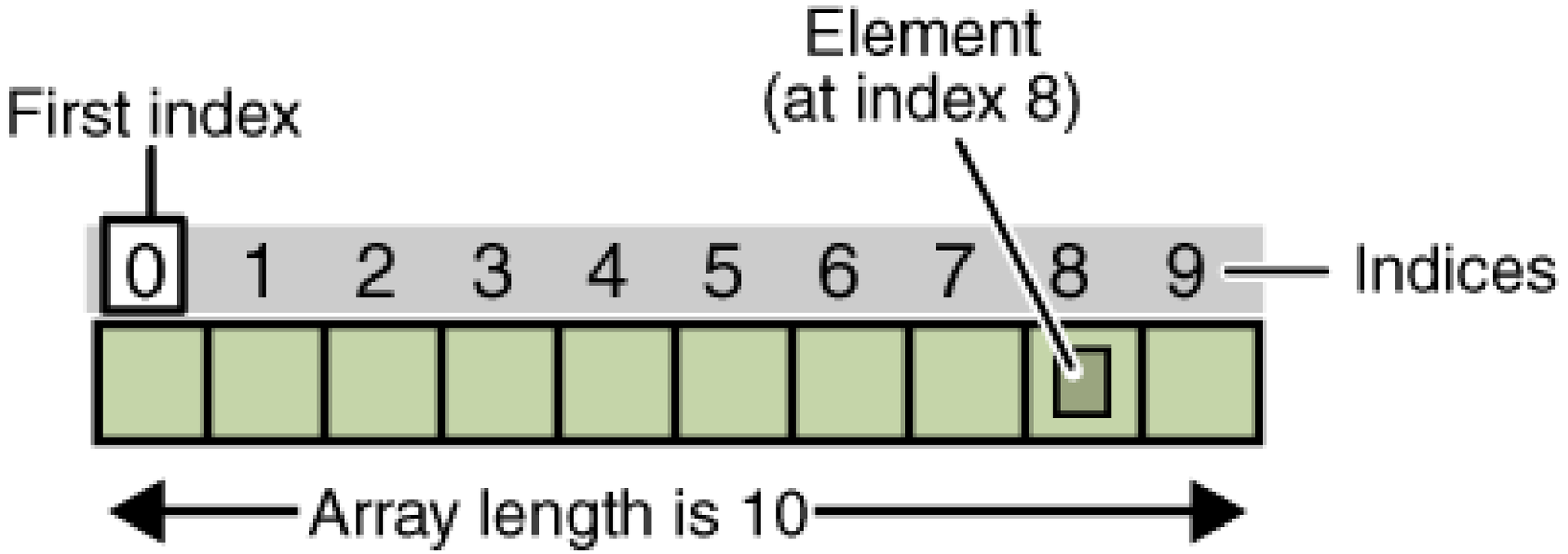
Implementation

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

Array-based Lists

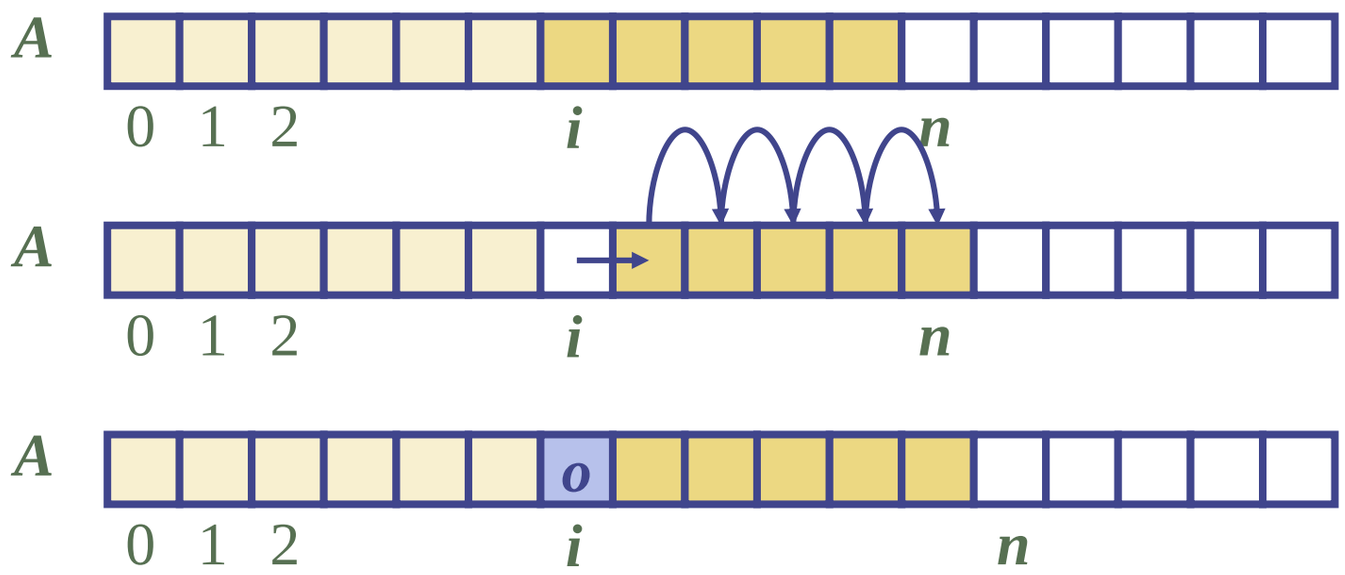
- An obvious choice for implementing the list ADT is to use an array, A,
  - where **A[i]** stores (a reference to) the element with index i.
- With a representation based on an array A, the get(i) and set(i, e) methods are easy to implement by accessing A[i] (assuming i is a legitimate index).

Array


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

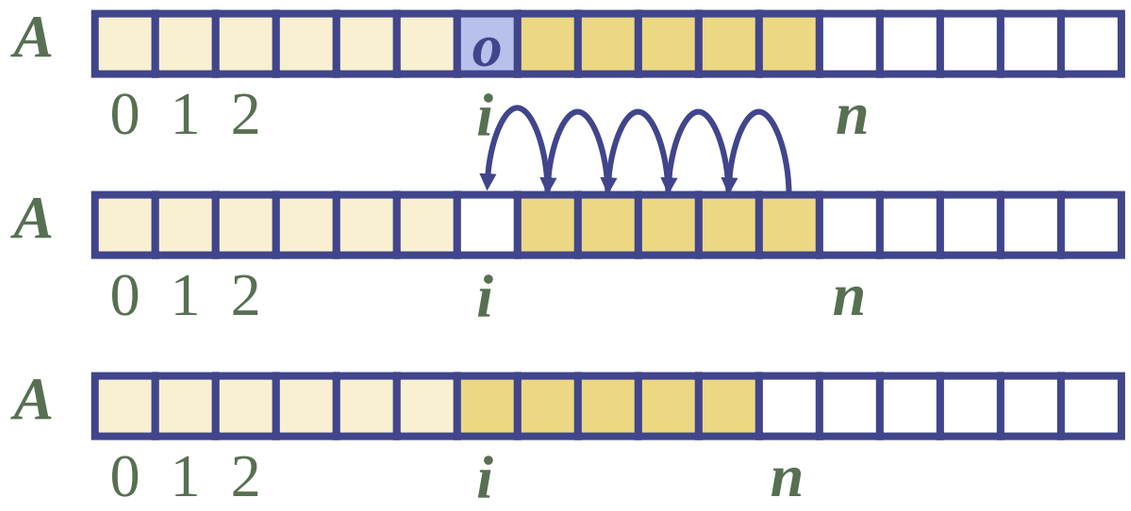
Array-based Lists

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



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  }
```

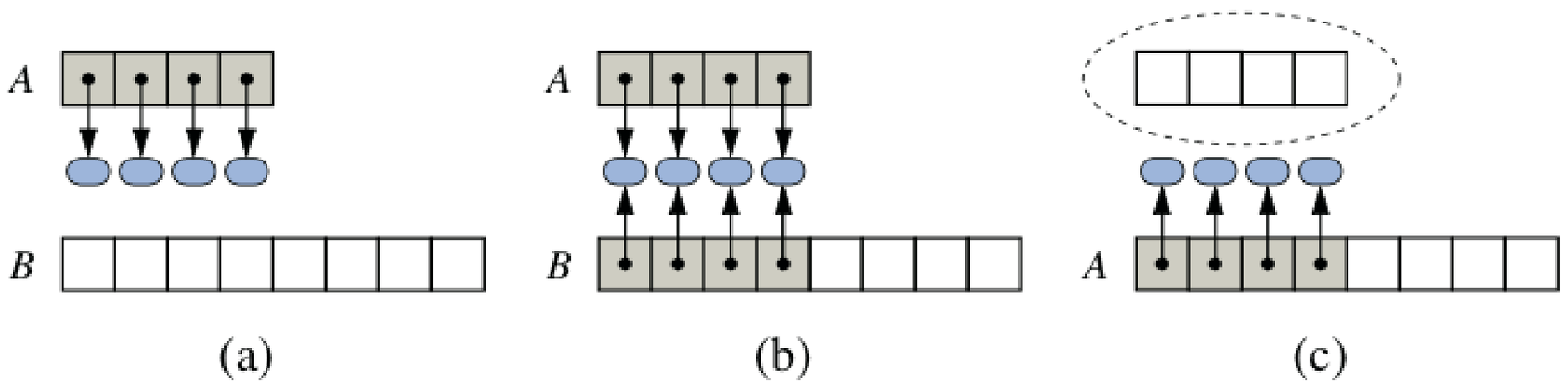
Array-based Lists

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

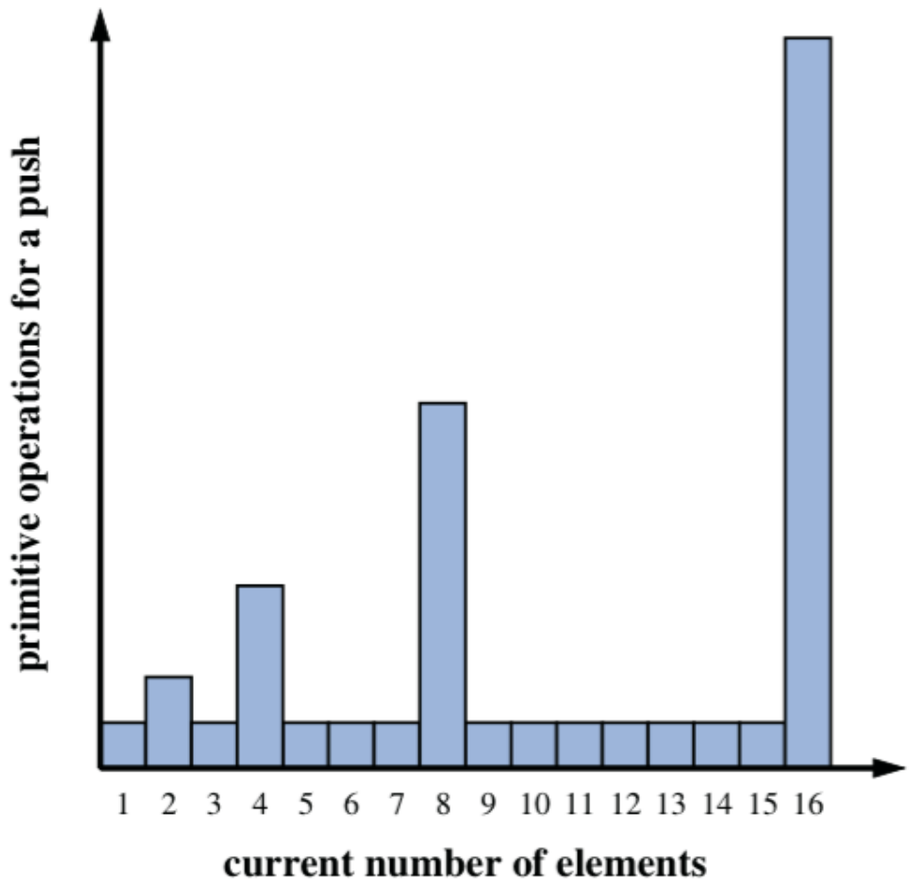


Figure 7.4: Running times of a series of push operations on a dynamic array.

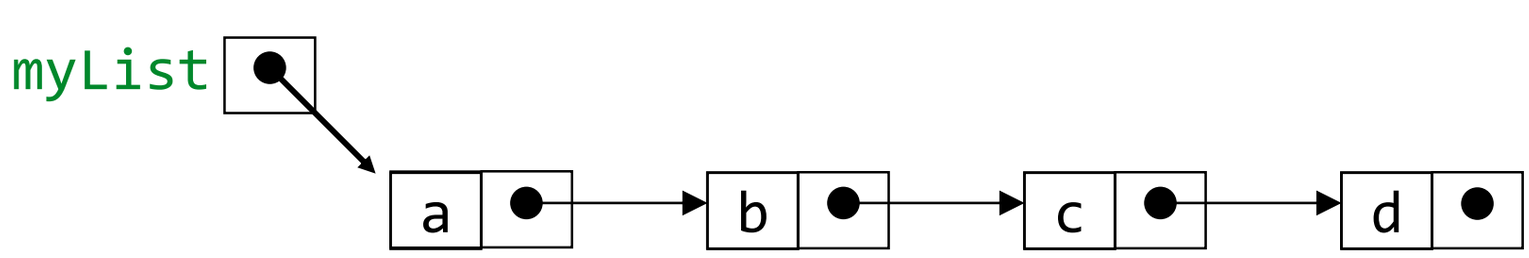
Implementation

- 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

Anatomy of a linked list

- A linked list consists of:
  - A sequence of nodes



Each node contains a value
and a link (pointer or reference) to some other node
The last node contains a null link
The list may (or may not) have a header

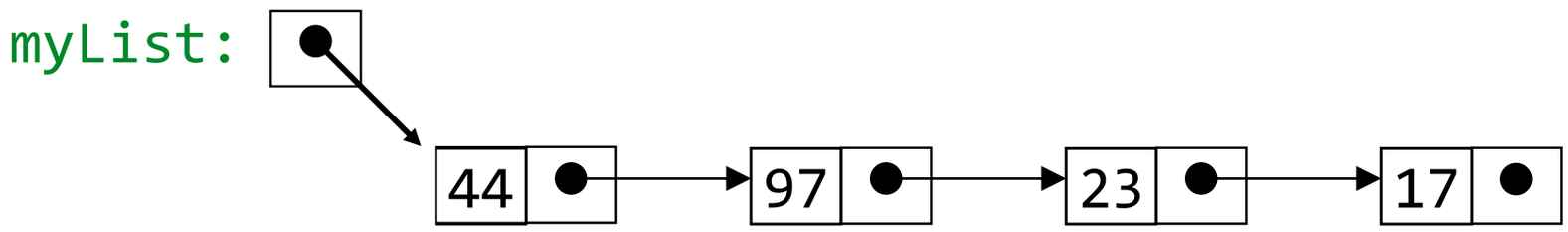
Terminology

- A node's successor is the next node in the sequence
  - The last node has no successor
- A node's predecessor is the previous node in the sequence
  - The first node has no predecessor
- A list's length is the number of elements in it
  - A list may be empty (contain no elements)

Creating references

- The keyword new creates a new object, but also returns a *reference* to that object
- For example, Person p = new Person("John")
  - new Person("John") creates the object and returns a reference to it
  - We can assign this reference to p, or use it in other ways

Creating links in Java



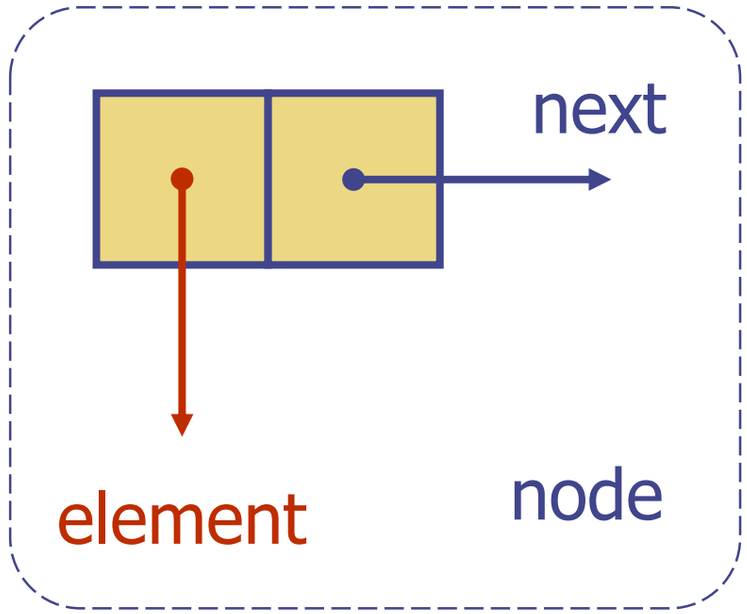

```
class Node {
     int value;
     Node next;

   Node (int v, Node n) { // constructor
        value = v;
        next = n;
   }
}

Node temp = new Node(17, null);
temp = new Node(23, temp);
temp = new Node(97, temp);
Node myList = new Node(44, temp);
```

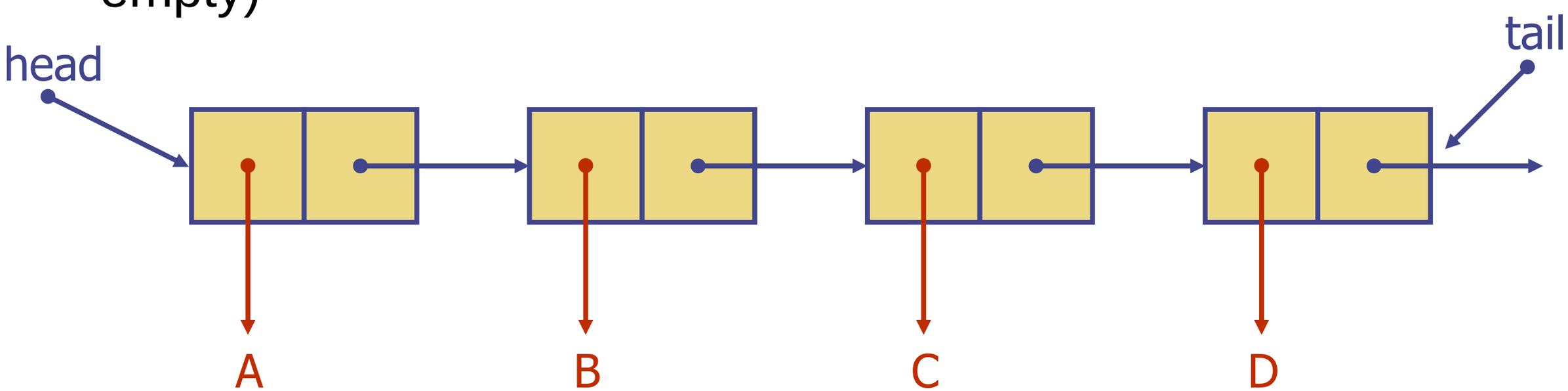
Singly Linked List

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



Each node contains a value and a link to its successor (the last node has no successor)

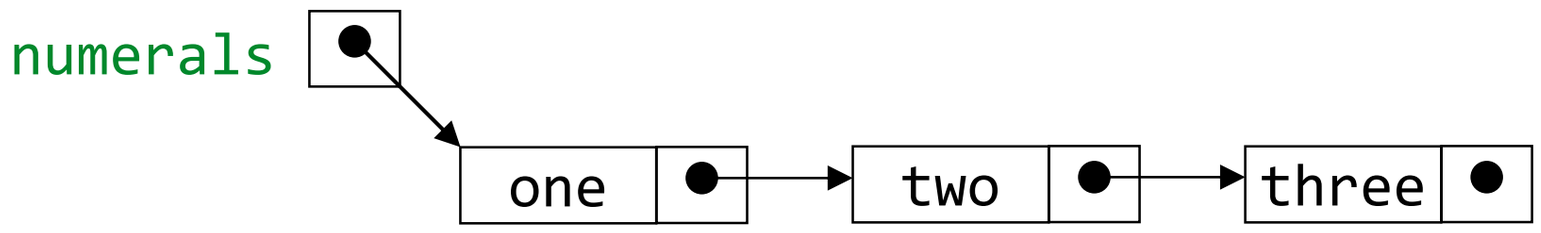
The header points to the first node in the list (or contains the null link if the list is empty)



Creating a simple list

- To create the list ("one", "two", "three"):
- Node numerals = new Node();
- numerals =

```
    new Node("one",
        new Node("two",
new Node("three", null)));
```



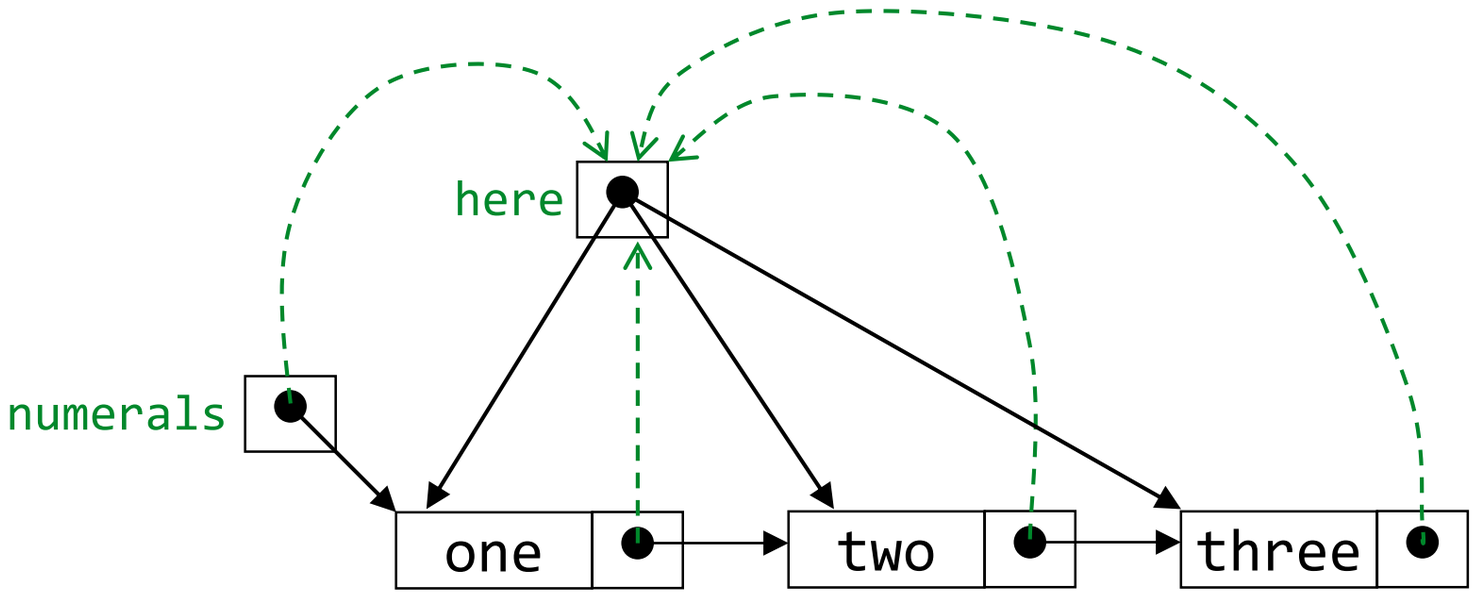
Traversing a SLL

- The following method traverses a list (and prints its elements):

```
public void printFirstToLast(Node here) {
  while (here != null) {
     System.out.print(here.value + "  ");
      here = here.next;
    }
}
```

- You would write this as an instance method of the `Node` class

Traversing a SLL



Inserting a node into a SLL

- There are many ways you might want to insert a new node into a list:
  - As the new first element
  - As the new last element
  - Before a given node (specified by a *reference*)
  - After a given node
  - Before a given value
  - After a given value

- All are possible, but differ in difficulty

Deleting a node from a SLL

- In order to delete a node from a SLL, you have to change the link in its *predecessor*
- This is slightly tricky, because you can't follow a pointer backwards
- Deleting the first node in a list is a special case, because the node's predecessor is the list header

Inserting as a new first element

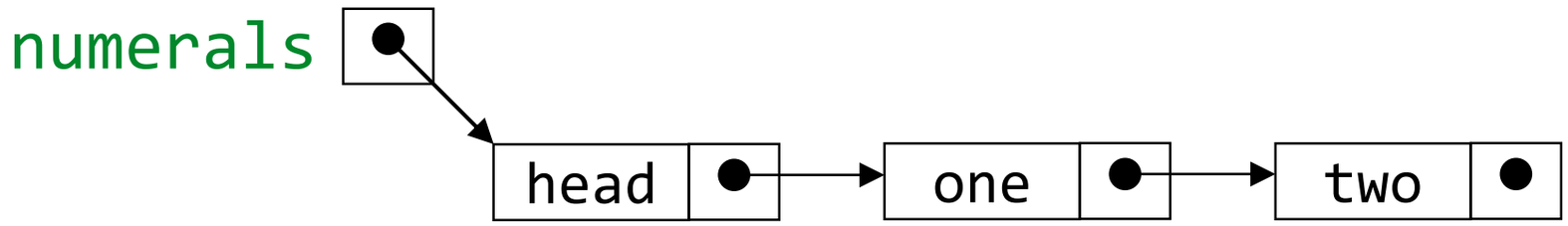
- This is probably the easiest method to implement
- In class `Node`:

```
Node insertAtFront(Node oldFront, Object value) {
    Node newNode = new Node(value, oldFront);
    return newNode;
}
```

- Use this as: `myList = insertAtFront(myList, value);`
- Why can’t we just make this an instance method of `Node`?

Using a header node

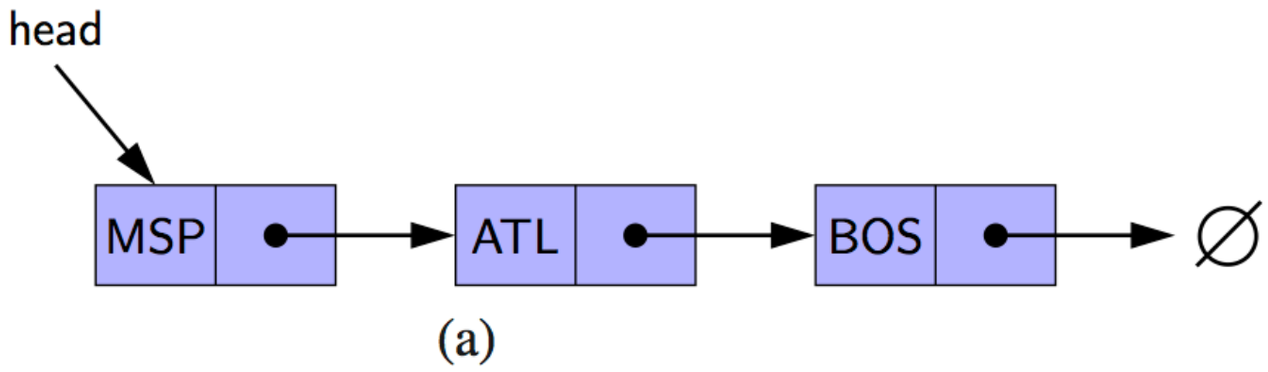
- A header node is just an initial node that exists at the front of every list, even when the list is empty
- The purpose is to keep the list from being `null`, and to point at the first element



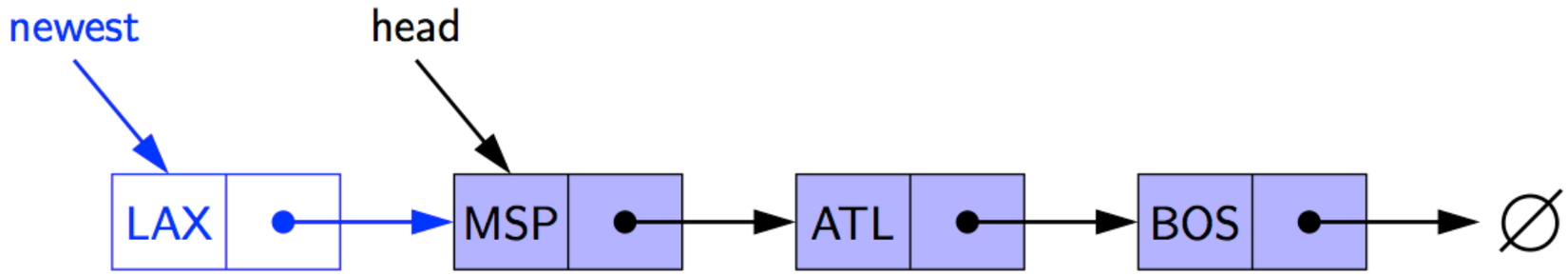

```
void insertAtFront(Object value) {
    Node front = new Node(value, this);
    this.next = front;
}
```

Addition to the Front

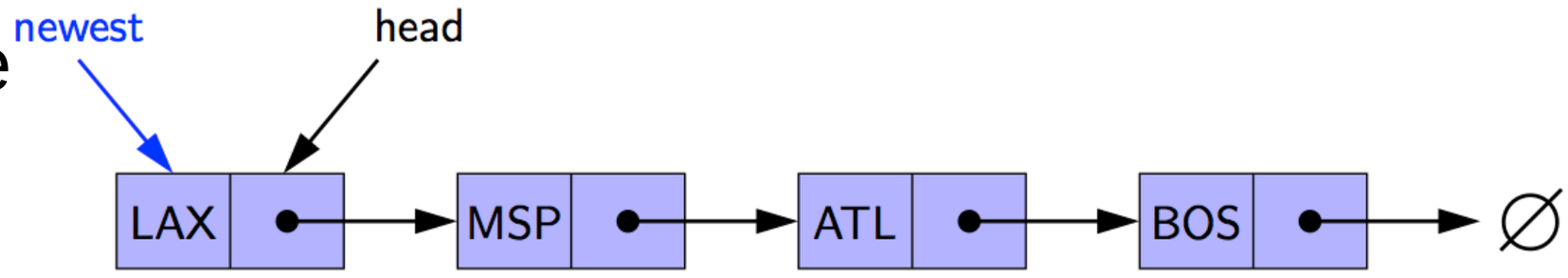
- Allocate new node
- Insert new element
- Have new node point to old head
- Update head to point to new node



(a)



(b)

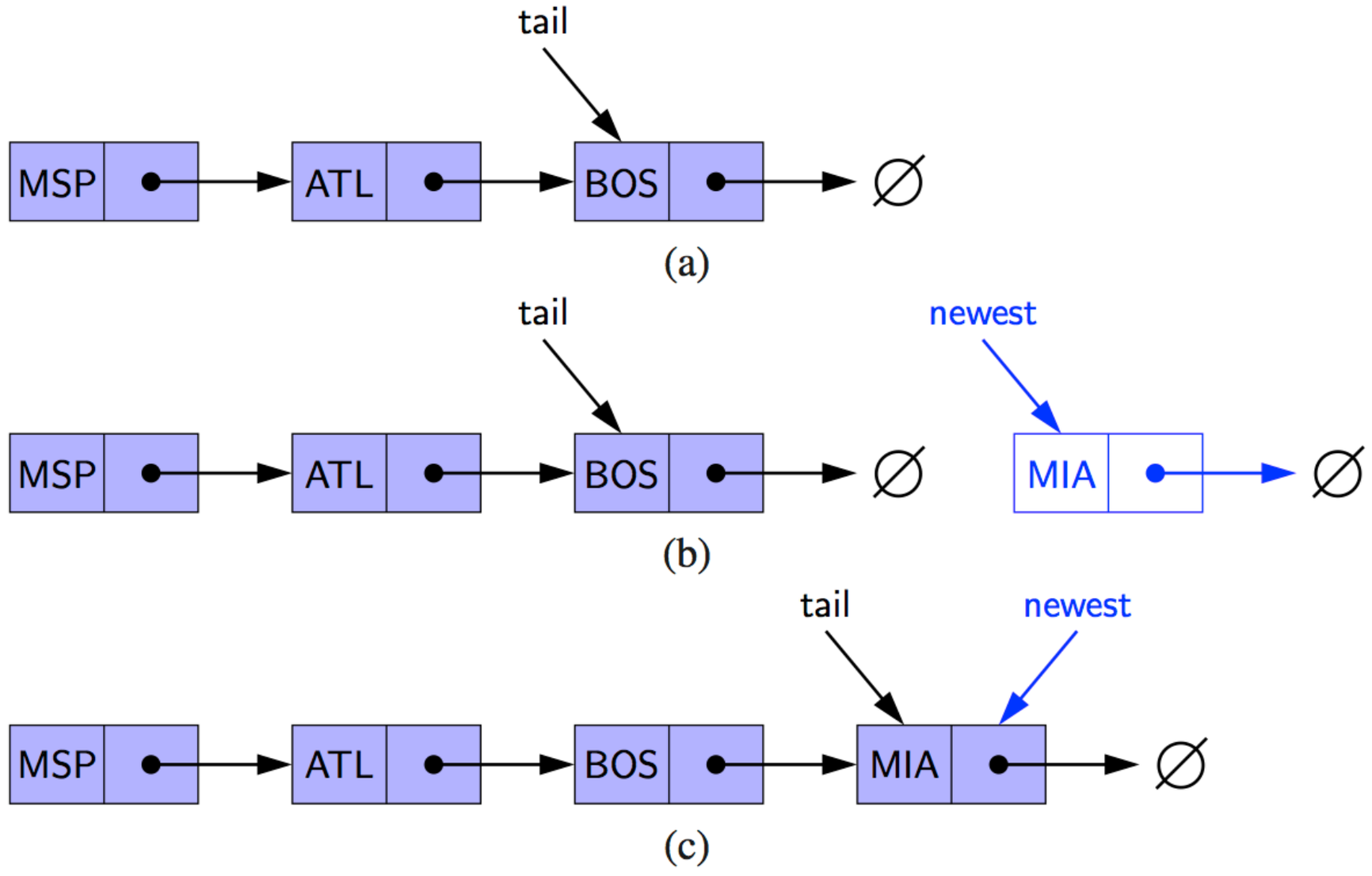


(c)

Time Complexity?

Addition to the Back

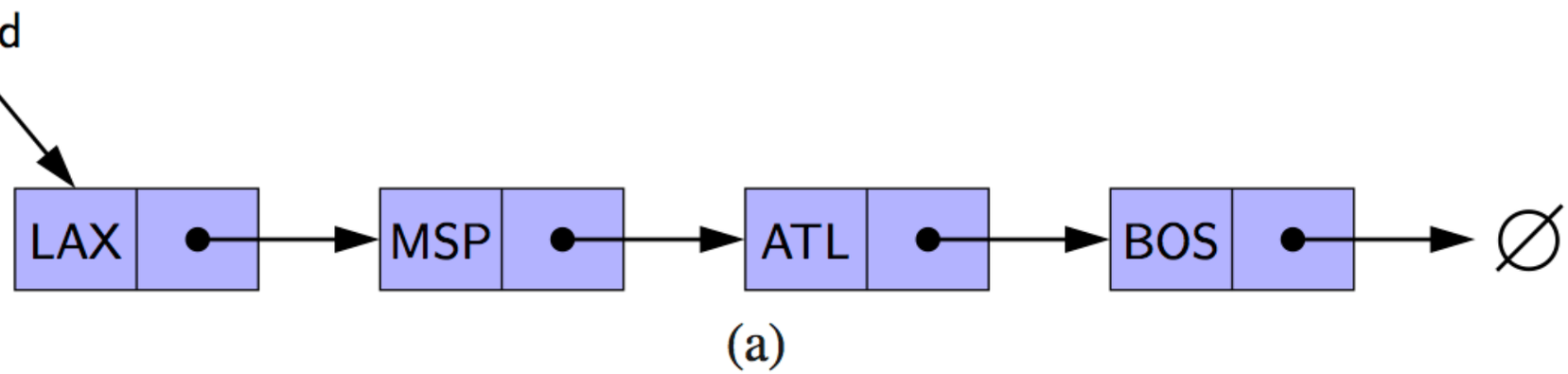
- Allocate a new node
- Insert new element
- Have new node point to null
- Have old last node point to new node
- Update tail to point to new node



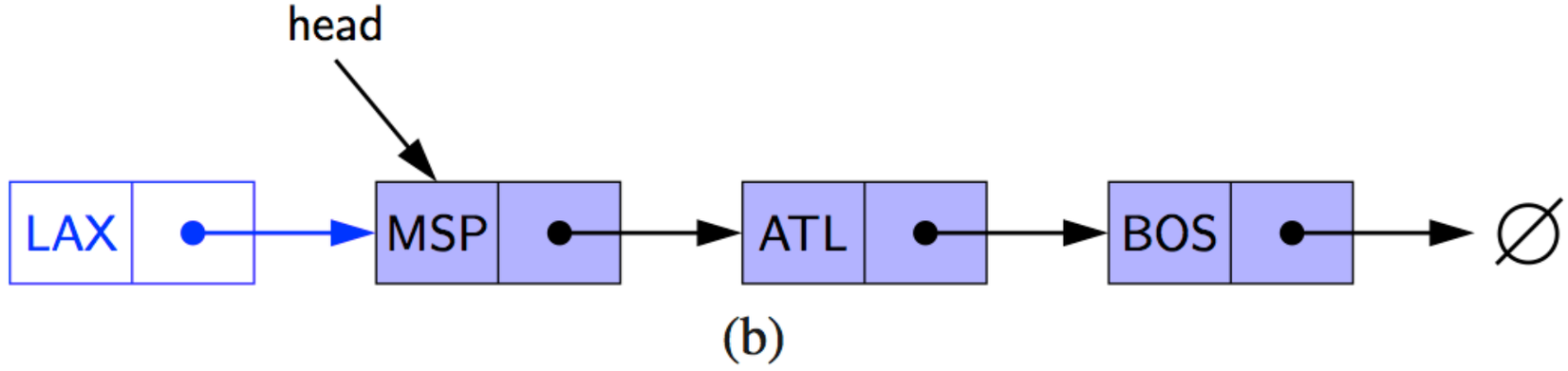
Time Complexity?

Removal from the Front

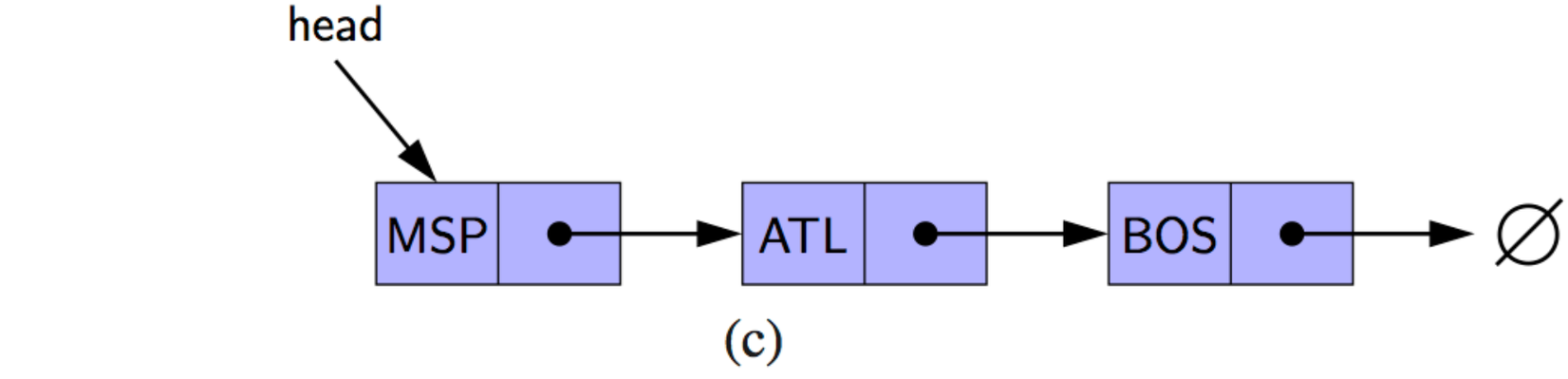
- Update head to point to next node in the list
- Allow garbage collector to reclaim the former first node



(a)



(b)

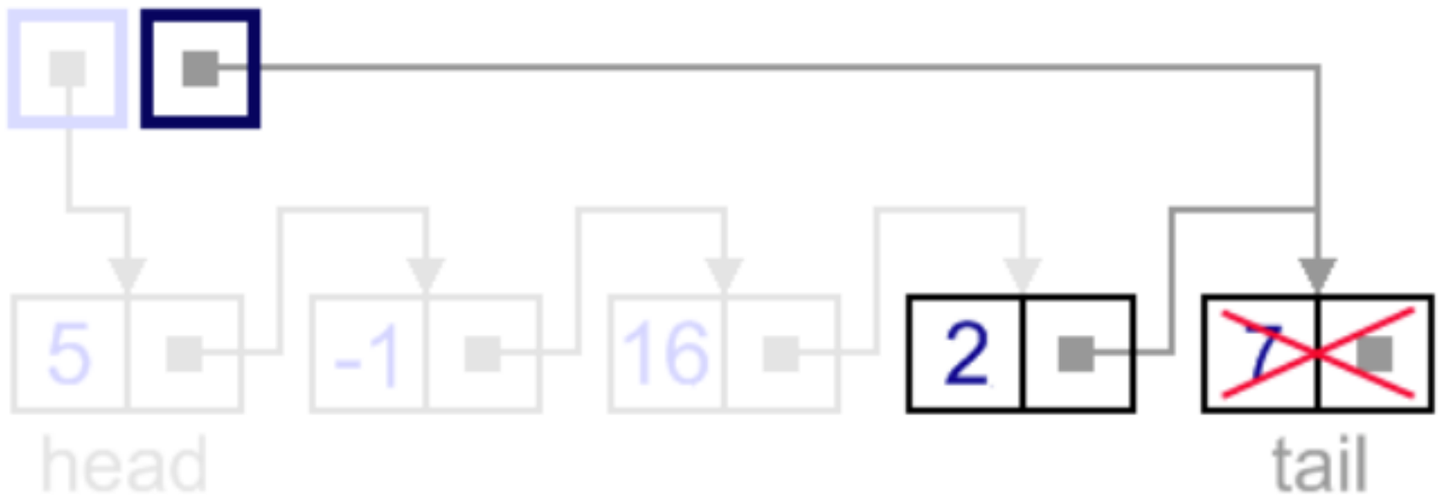


(c)

Time Complexity?

Removal from the Back

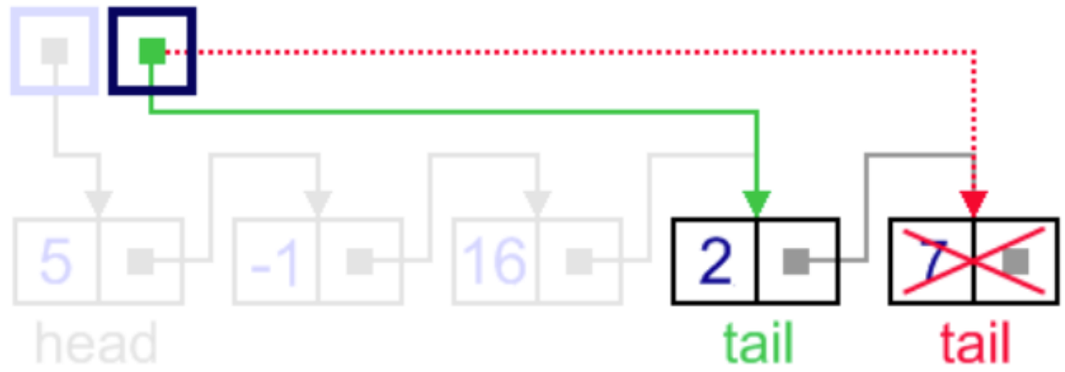
- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node
  - you must find the node before the tail iteratively



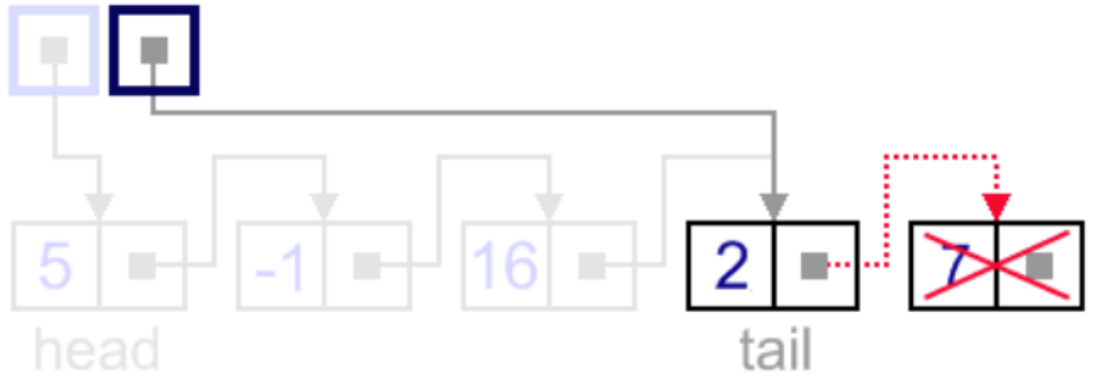
Time Complexity?

Removal from the Back

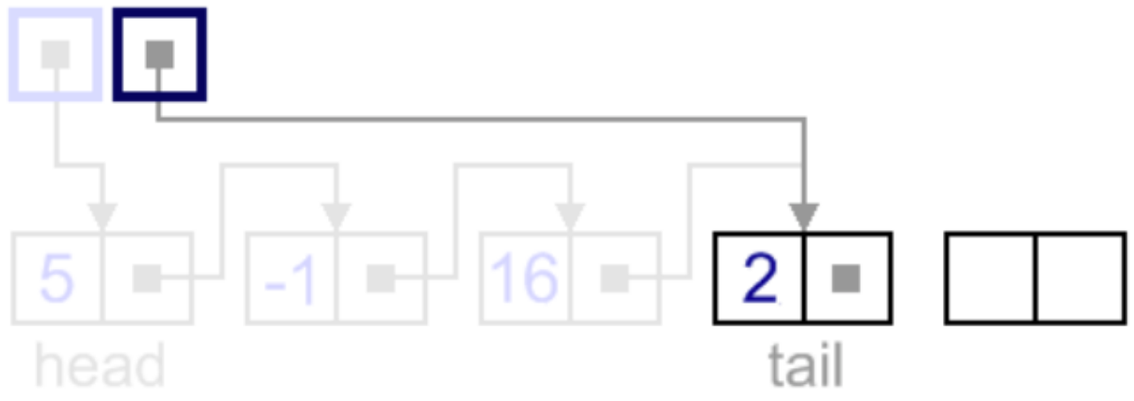
1



2

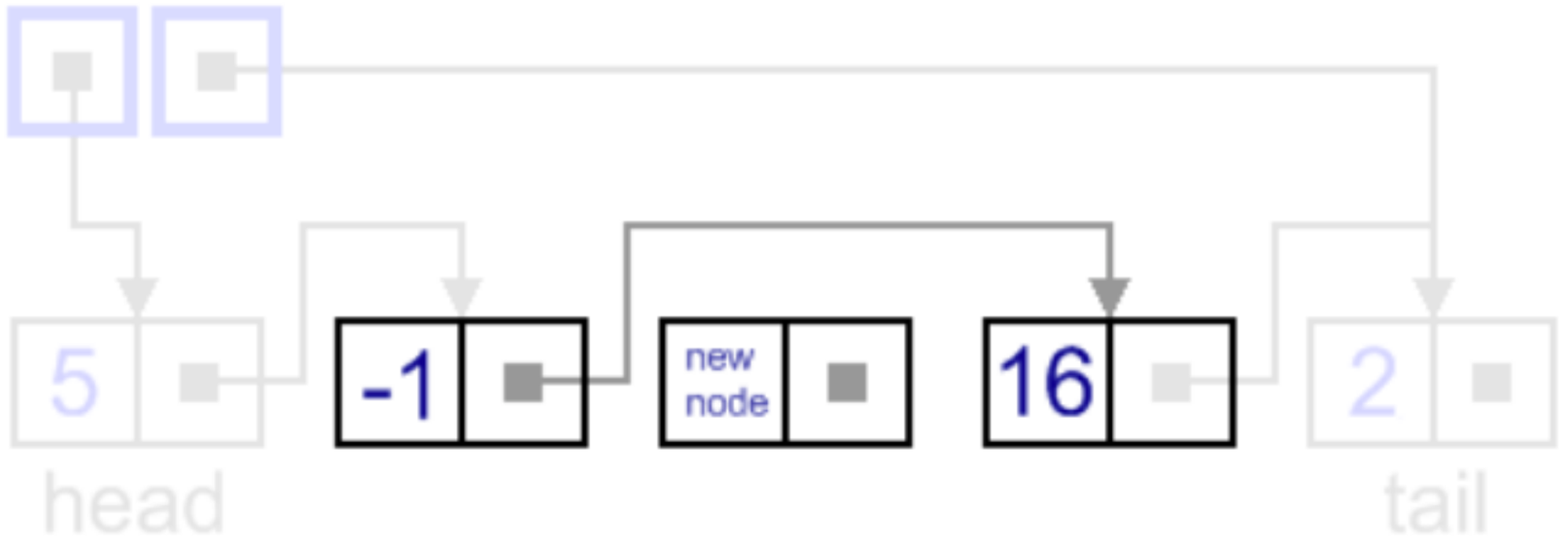


3



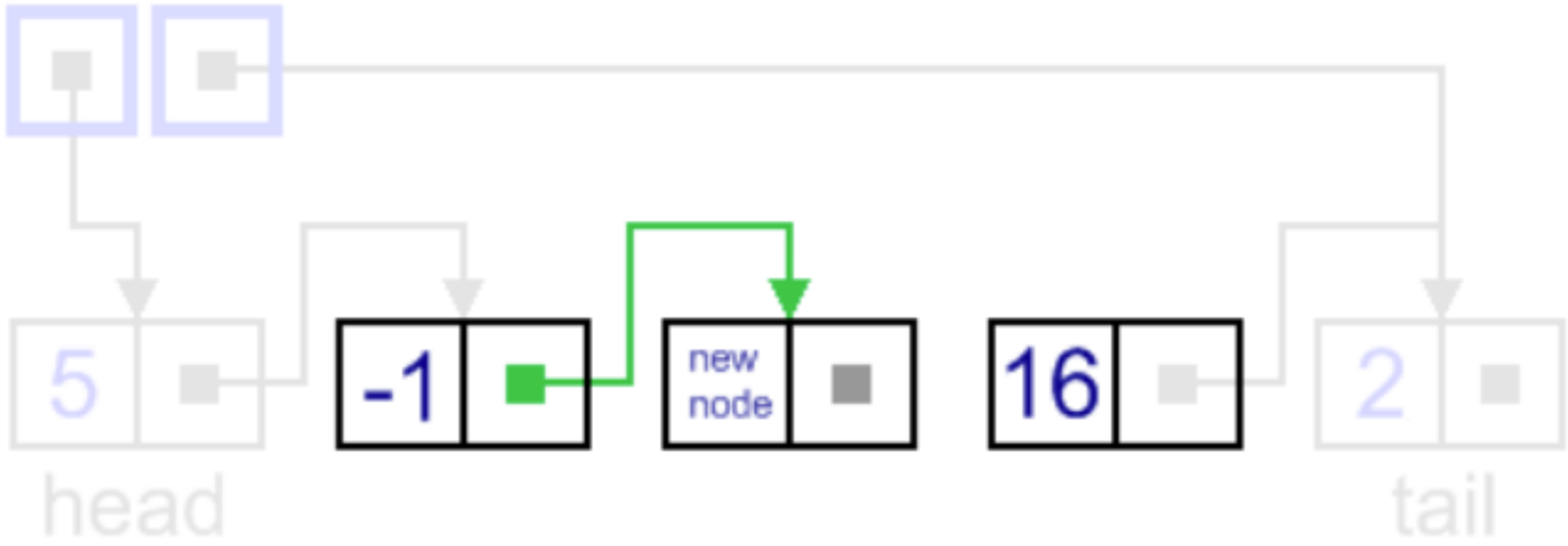
Addition between Nodes

- Inserted between two nodes

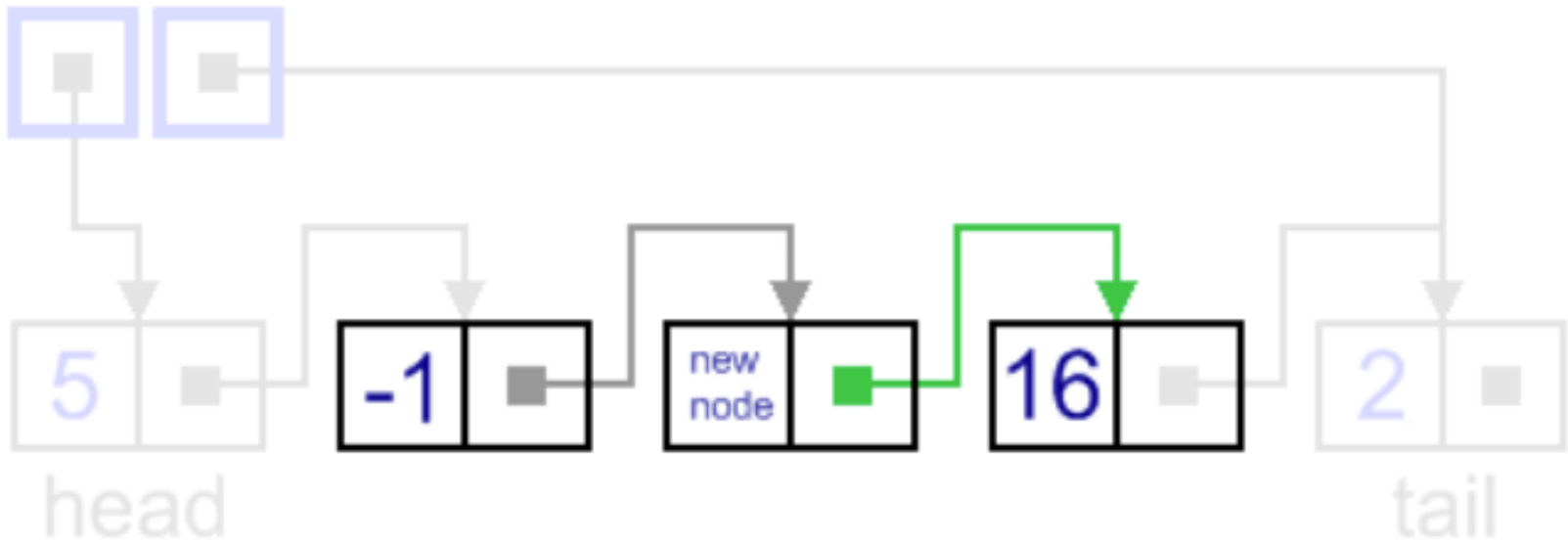


Addition between Nodes

1



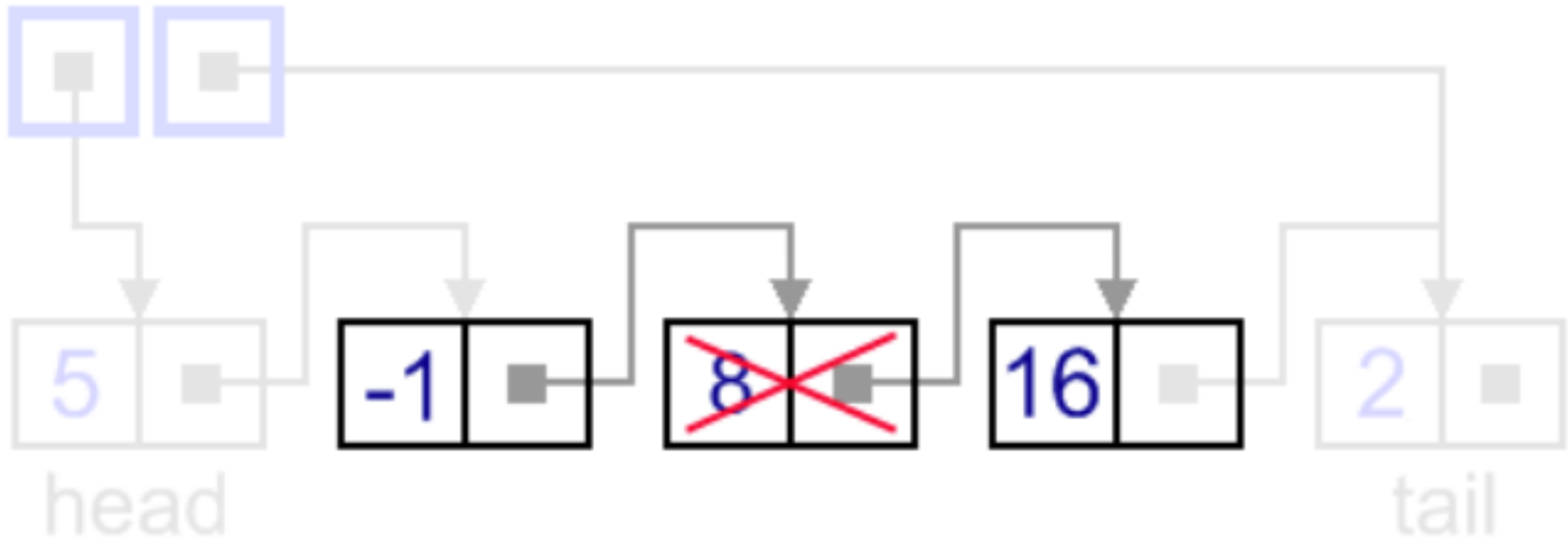
2



Time Complexity, assuming we have the pointers to the left and right nodes?

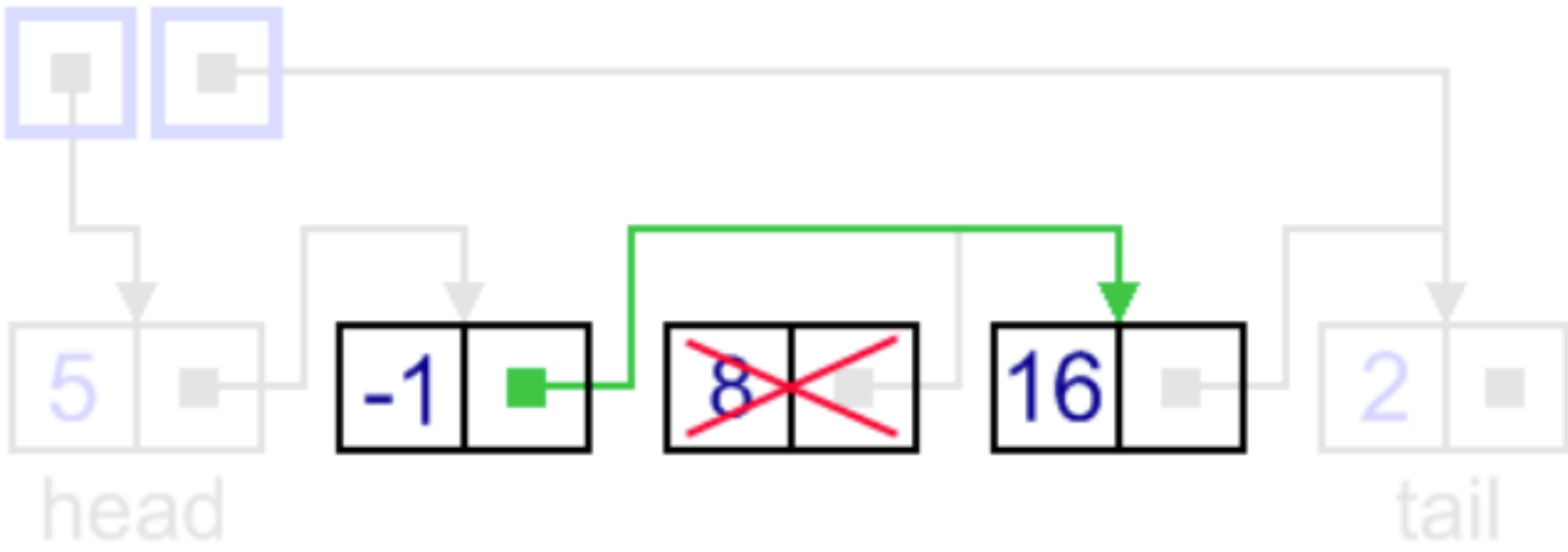
Removal between Nodes

- Node located between two nodes

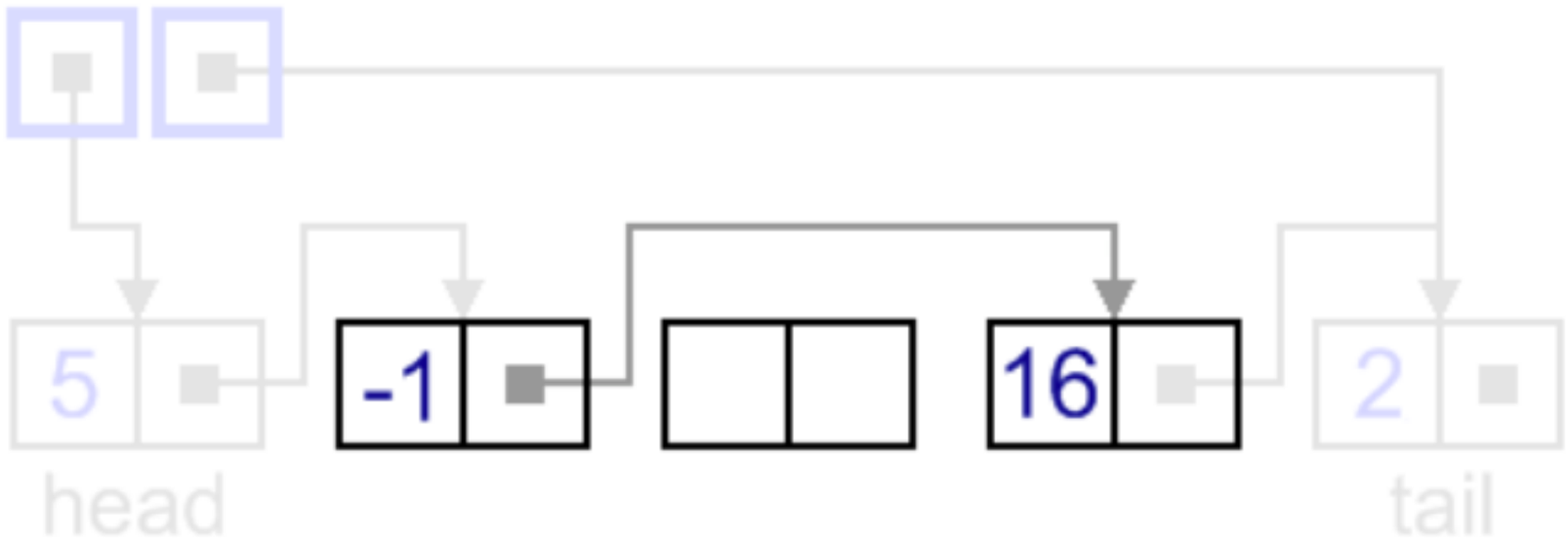


Removal between Nodes

1



2

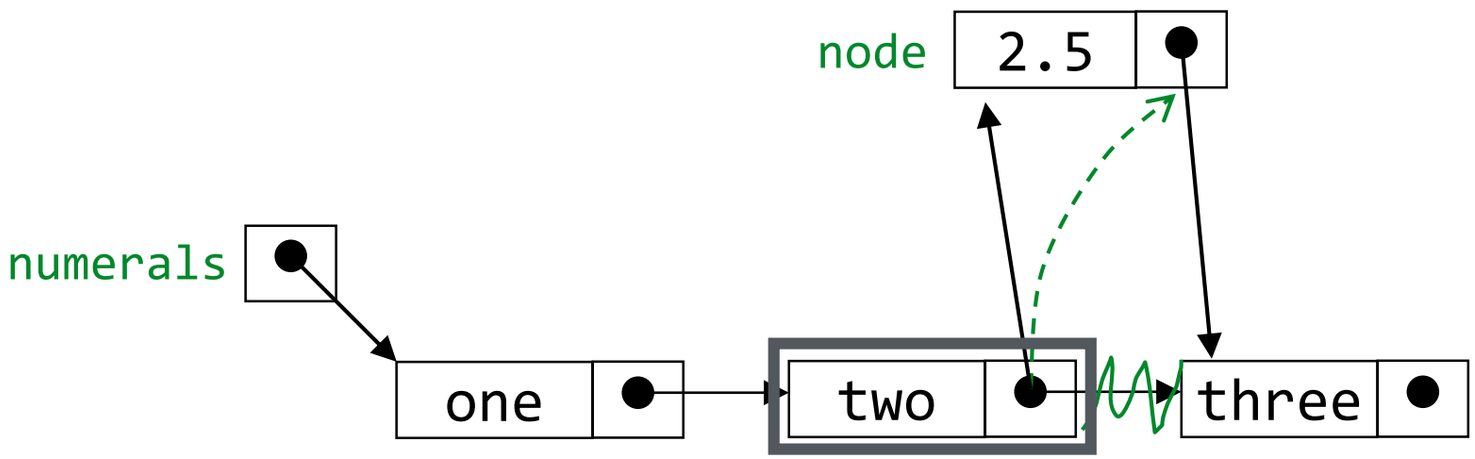


Time Complexity, assuming that we have the pointers to the left and right nodes?

Inserting a node after a given value

```
void insertAfter(Object target, Object value) {
    for (Node here = this; here != null; here = here.next) {
        if (here.value.equals(target)) {
            Node node = new Node(value, here.next);
            here.next = node;
            return;
        }
    }
    // Couldn't insert--do something reasonable here!
}
```

Inserting after



Find the node you want to insert after

First, copy the link from the node that's already in the list

Then, change the link in the node that's already in the list

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

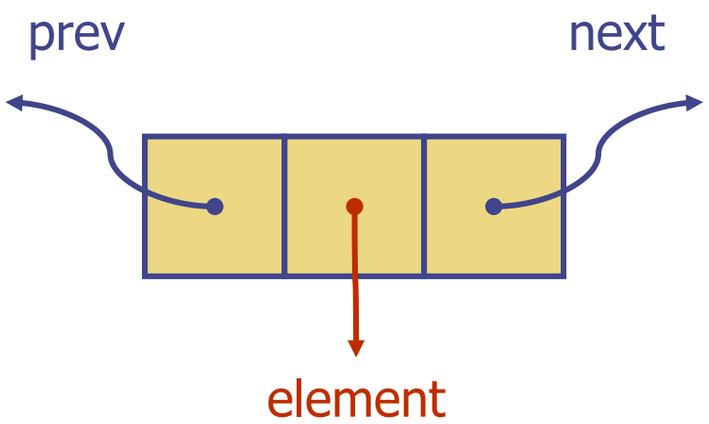
```
30    // update methods
31    public void addFirst(E e) {                  // adds element e to the front of the list
32      head = new Node<>(e, head);                // create and link a new node
33      if (size == 0)
34        tail = head;                             // special case: new node 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 (DDL)

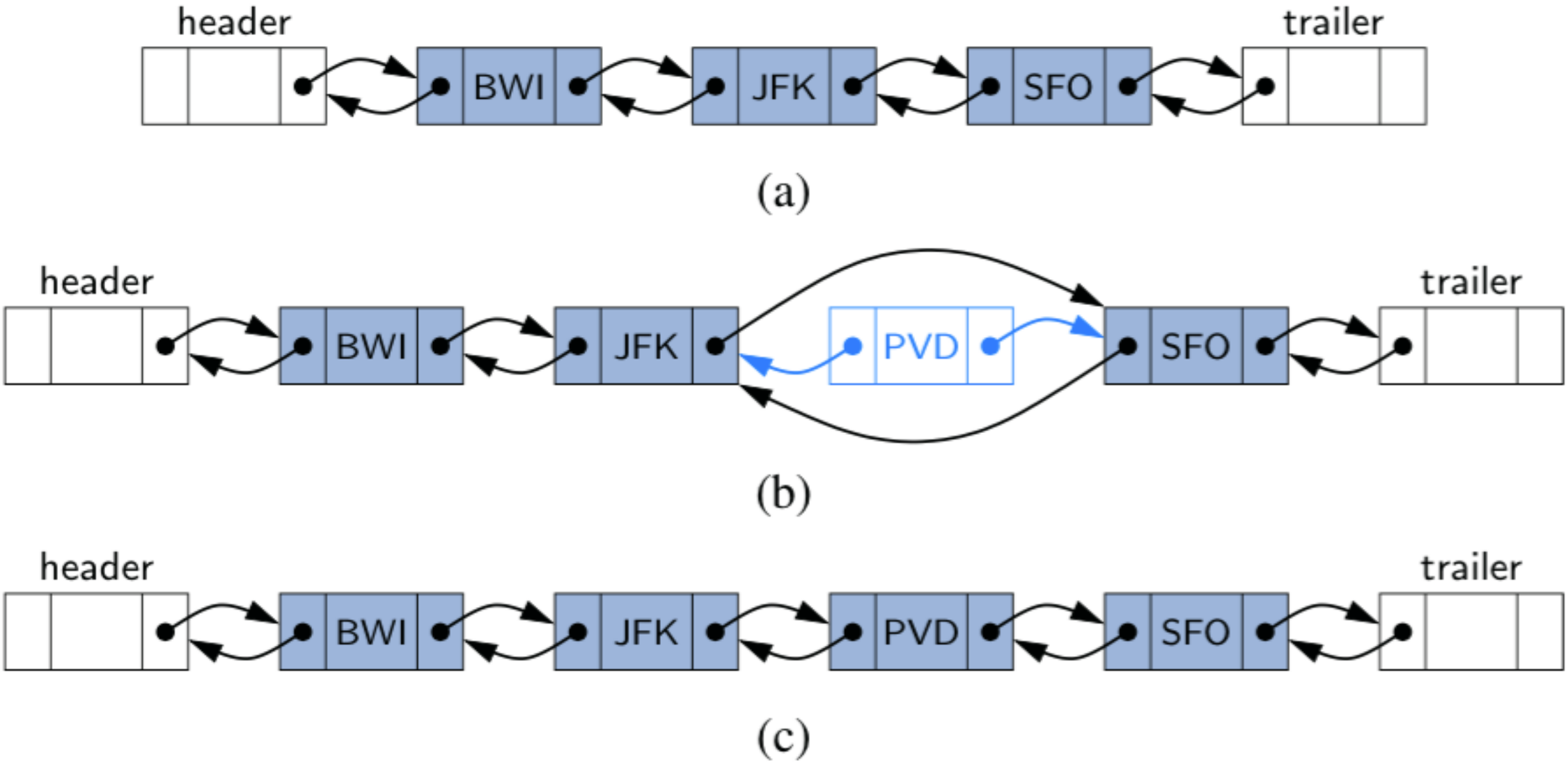


- Each node contains a value, a link to its successor (if any), *and* a link to its predecessor (if any)
- The header points to the first node in the list *and* to the last node in the list (or contains null links if the list is empty)

DLLs compared to SLLs

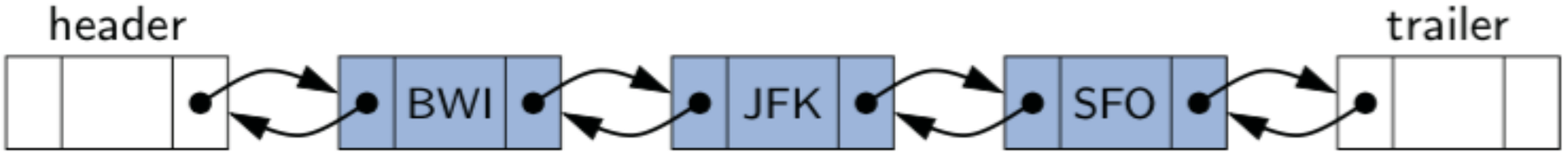
- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier
- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

Addition between Nodes

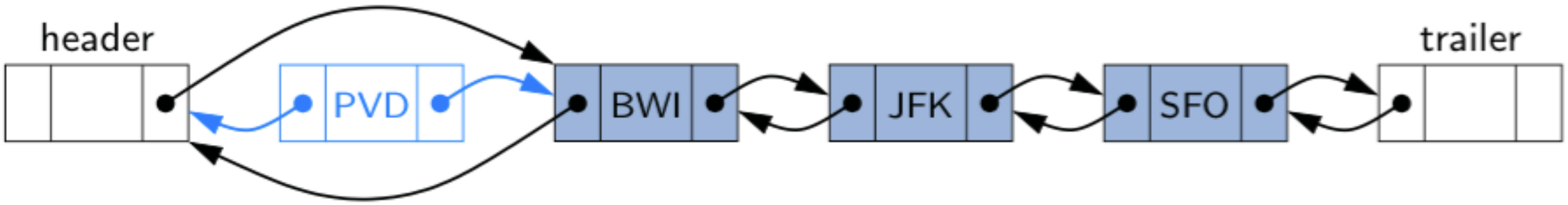


Adding an element to a doubly linked list: (a) before the operation; (b) after creating the new node; (c) after linking the neighbors to the new node.

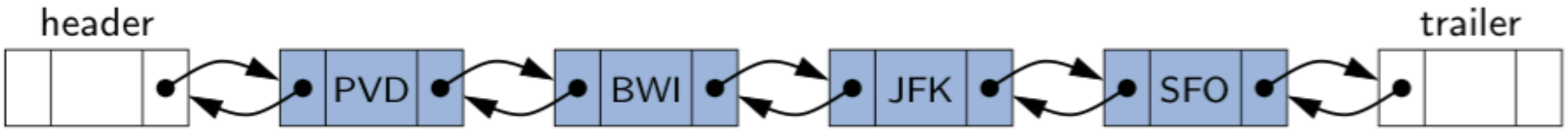
Addition to the Front



(a)



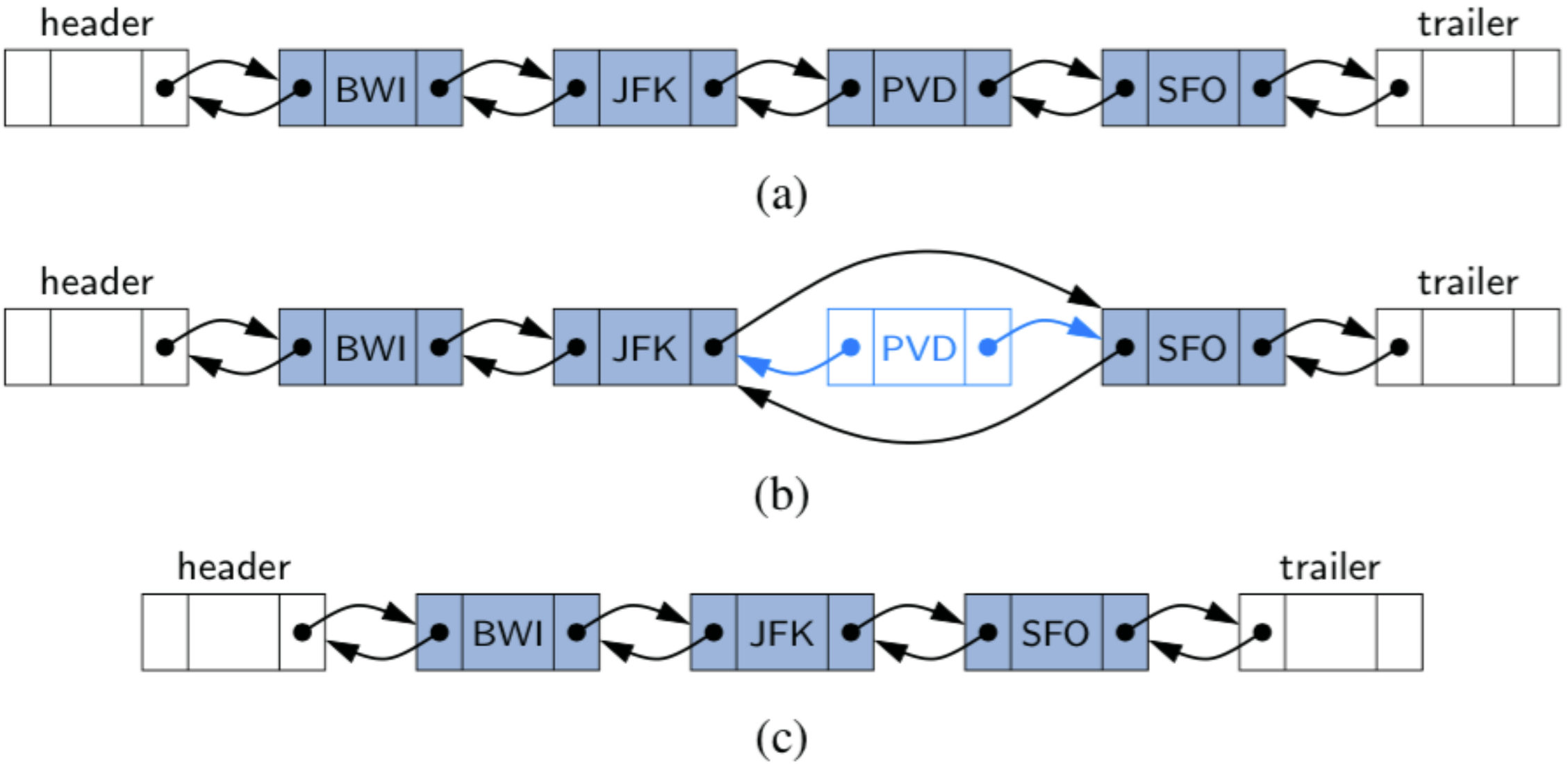
(b)



(c)

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      }
```

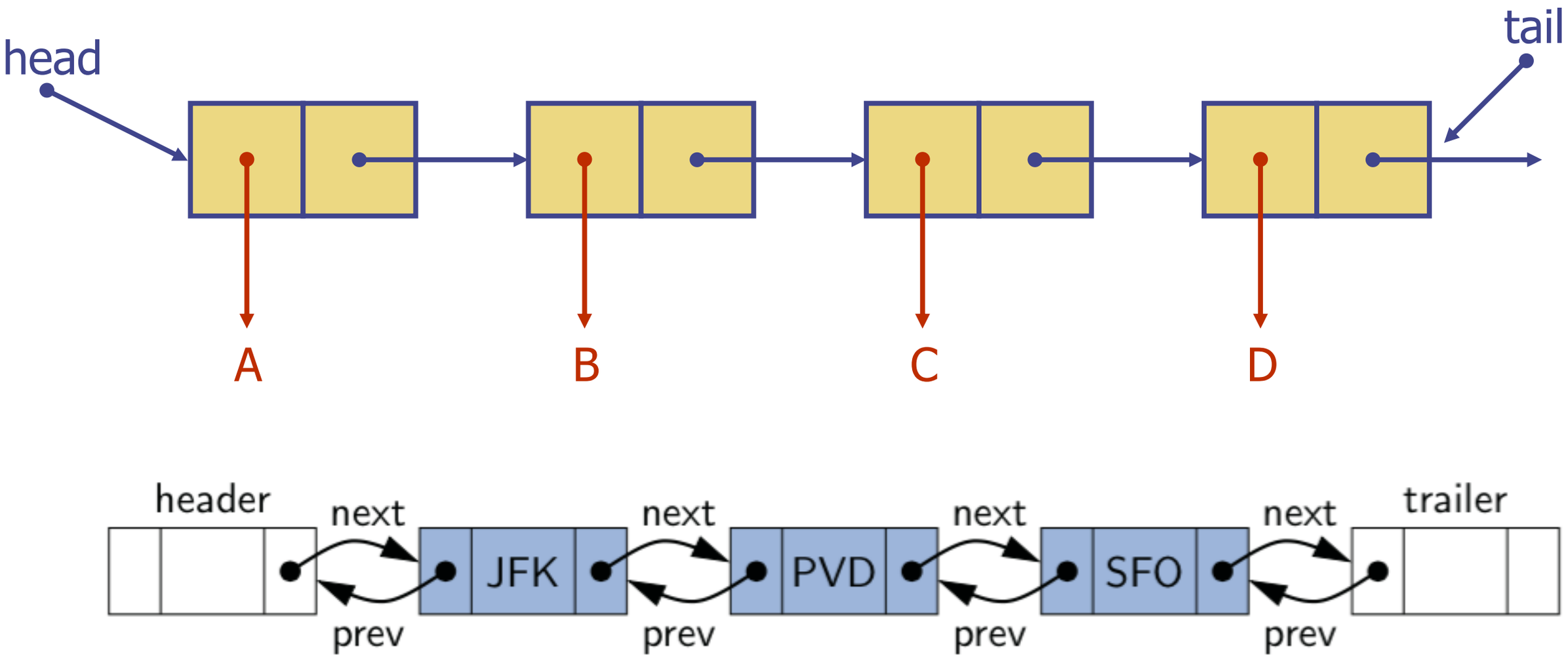
Double Linked List

```
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 ----------
```

Head & Tail of SLL vs. Header & Trailer of DLL



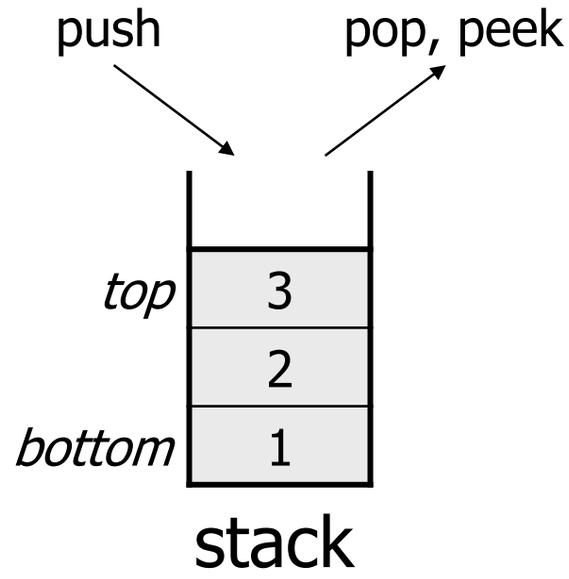
Analyze the difference! I will ask about this in the next lecture!

Stacks & Queues

Stacks

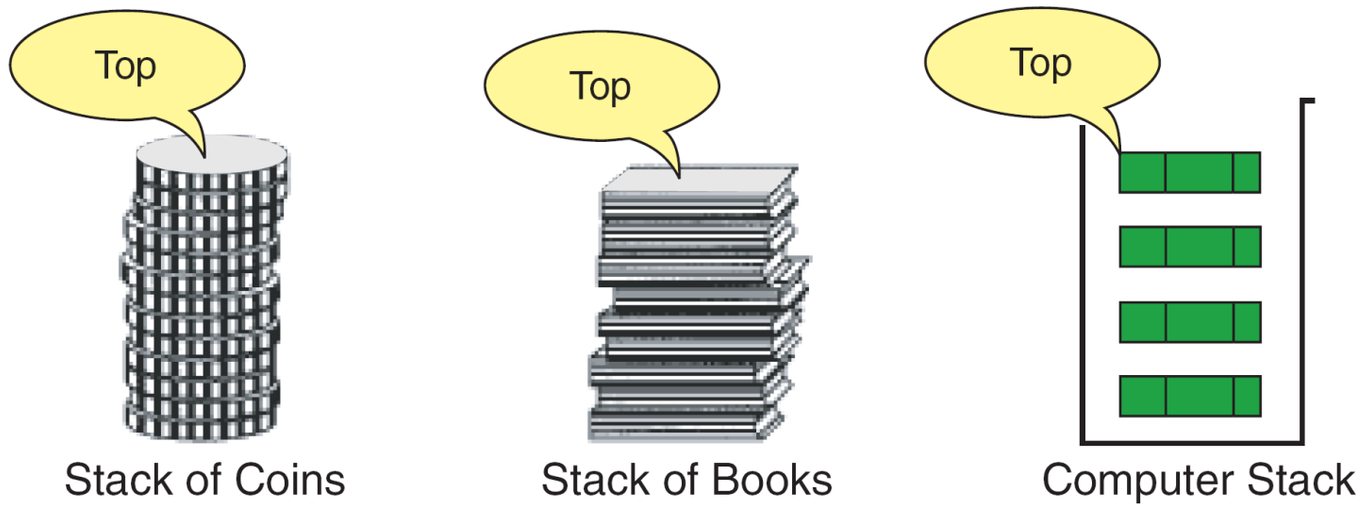
Stacks

- A special kind of list
  - Addition and Removal takes place only at one end, called the top
  - the last element added, is always the first one to be deleted
- So, stack is a LIFO sequence (Last-In, Firs-Out)
- Basic stack operations:
  - **push**: Add an element to the top.
  - **pop**: Remove the top element.
  - **peek**: Examine the top element.



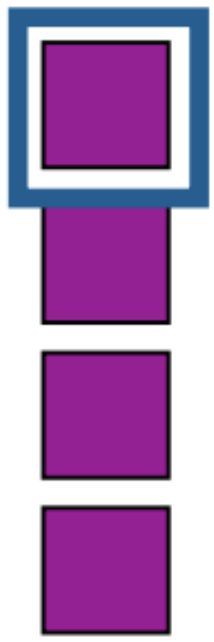
Stacks

- Mail in a pile on your desk
- Hyperlinks in your browser
- Method calls



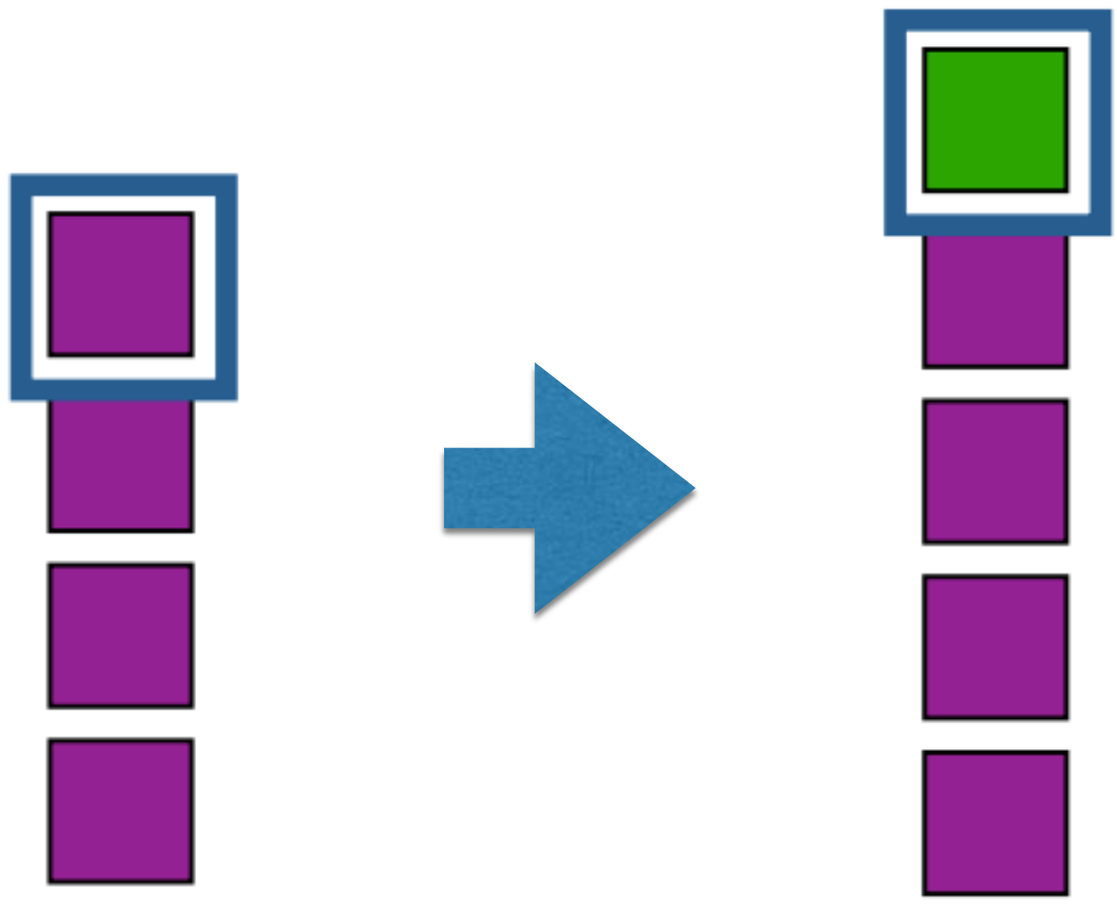
Stack Operations

- size: returns the number of items in the stack
- isEmpty: returns whether stacks has no items
- top: returns the item at the top (without removing it)



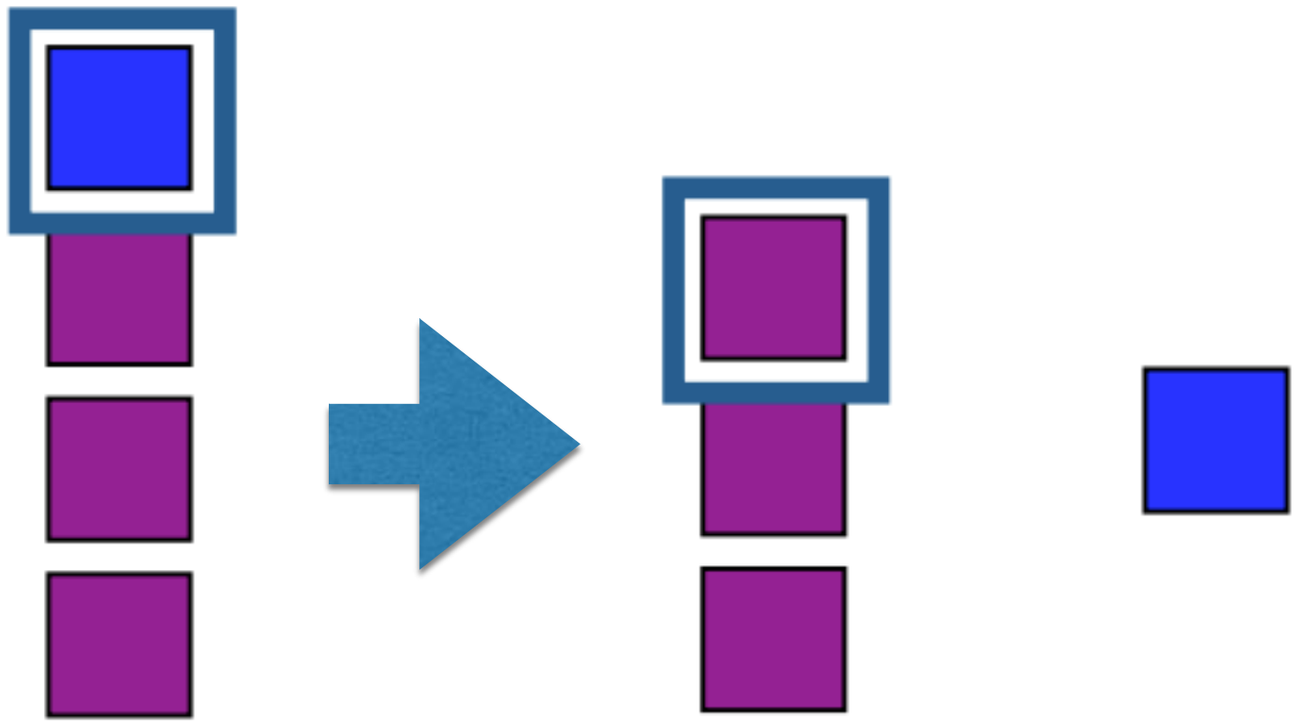
Stack Operatios

- push(x): insert an item x at the top of the stack



Stack Operations

- pop: remove the item at the top



Stack Implementation

- Dedicated Implementation, OR
- All the operations can be directly implemented using the LIST ADT (as a wrapper around a built-in list object)

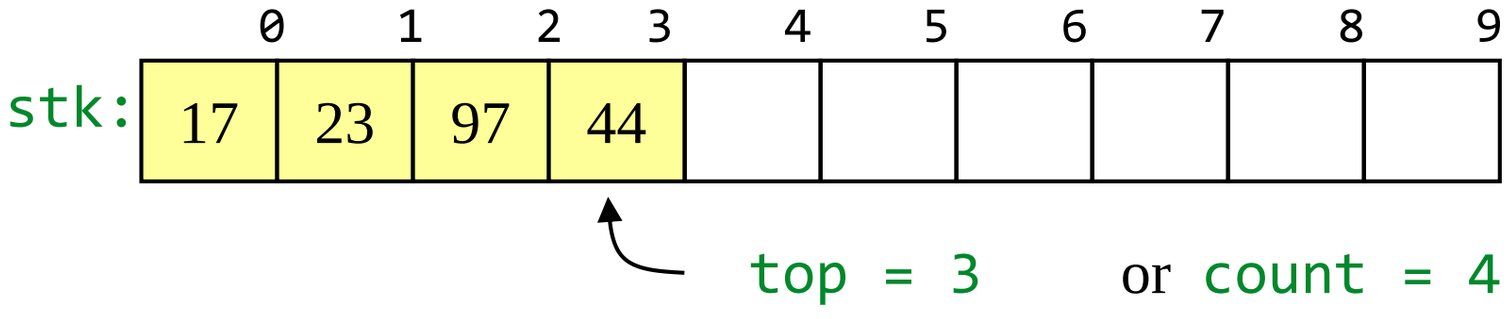
Stack ADT

```
public interface Stack<E> {
 int size();
 boolean isEmpty();
 E top();
 void push(E element);
 E pop();
}
```

Array implementation of stacks

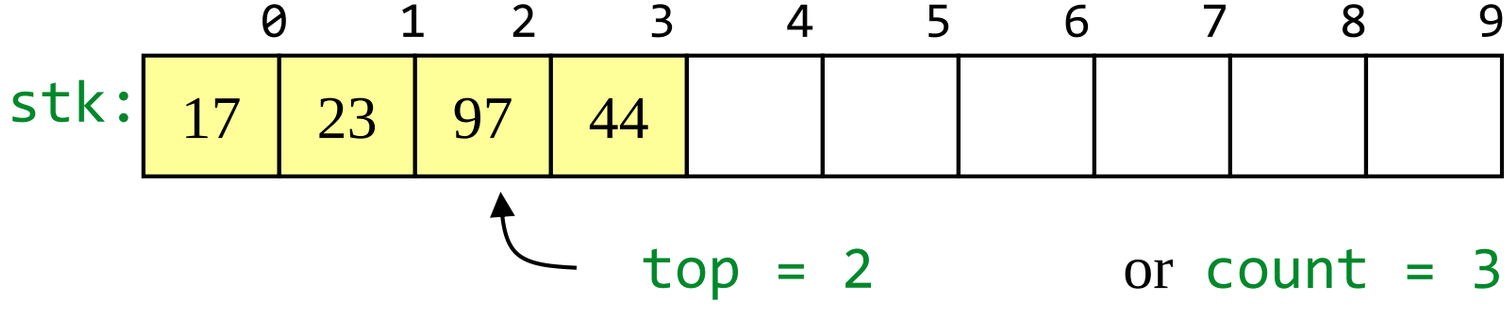
- To implement a stack, items are inserted and removed at the same end (called the top)
- Efficient array implementation requires that the top of the stack be towards the center of the array, not fixed at one end
- To use an array to implement a stack, you need both the array itself and an integer
  - The integer tells you either:
    - Which location is currently the top of the stack, or
    - How many elements are in the stack

Pushing and popping



- If the bottom of the stack is at location `0`, then an empty stack is represented by `top = -1` or `count = 0`
- To add (push) an element, either:
  - Increment `top` and store the element in `stk[top]`, or
  - Store the element in `stk[count]` and increment `count`
- To remove (pop) an element, either:
  - Get the element from `stk[top]` and decrement `top`, or
  - Decrement `count` and get the element in `stk[count]`

After popping



- When you pop an element, do you just leave the "deleted" element sitting in the array?
- The surprising answer is, *"it depends"*
  - If this is an array of primitives, *or* if you are programming in C or C++, *then* doing anything more is just a waste of time
  - If you are programming in Java, and the array contains objects, you should set the "deleted" array element to `null`
  - Why? To allow it to be garbage collected!

Error checking

- There are two stack errors that can occur:
  - Underflow: trying to pop (or peek at) an empty stack
  - Overflow: trying to push onto an already full stack
- For underflow, you should throw an exception
  - If you don't catch it yourself, Java will throw an `ArrayIndexOutOfBounds` exception
  - You could create your own, more informative exception
- For overflow, you could do the same things
  - Or, you could check for the problem, and copy everything into a new, larger array

ArrayStack

```
public class ArrayStack<E> implements Stack<E> {
  /** Default array capacity. */
  public static final int CAPACITY=1000;

  /** Generic array used for storage of stack elements. */
  private E[] data;

/** Index of the top element of the stack in the array. */
private int t = -1;
```

ArrayStack

```
/** Constructs an empty stack using the default array capacity. */
public ArrayStack() { this(CAPACITY); }

/**
 * Constructs and empty stack with the given array capacity.
 * @param capacity length of the underlying array
 */
@SuppressWarnings({"unchecked"})
public ArrayStack(int capacity) {
  data = (E[]) new Object[capacity];
}
```

ArrayStack

```
/**
  * Returns the number of elements in the stack.
  * @return number of elements in the stack
  */
 @Override
public int size() { return (t + 1); }

/**
 * Tests whether the stack is empty.
 * @return true if the stack is empty, false otherwise
 */
@Override
public boolean isEmpty() { return (t == -1); }
```

ArrayStack

```
/**
 * Inserts an element at the top of the stack.
 * @param e   the element to be inserted
 * @throws IllegalStateException if the array storing the elements is full
 */
@Override
public void push(E e) throws IllegalStateException {
  if (size() == data.length) throw new IllegalStateException("Stack is full");
  data[++t] = e; // increment t before storing new item
}
```

ArrayStack

```
 /**
    * Returns, but does not remove, the element at the top of the
stack.
    * @return top element in the stack (or null if empty)
    */
   @Override
   public E top() {
     if (isEmpty()) return null;
     return data[t];
   }
```

ArrayStack

```
/**
 * Removes and returns the top element from the stack.
 * @return element removed (or null if empty)
 */
@Override
public E pop() {
  if (isEmpty()) return null;
  E answer = data[t];
  data[t] = null;                    // dereference to help garbage collection
  t--;
  return answer;
}
```

ArrayStack

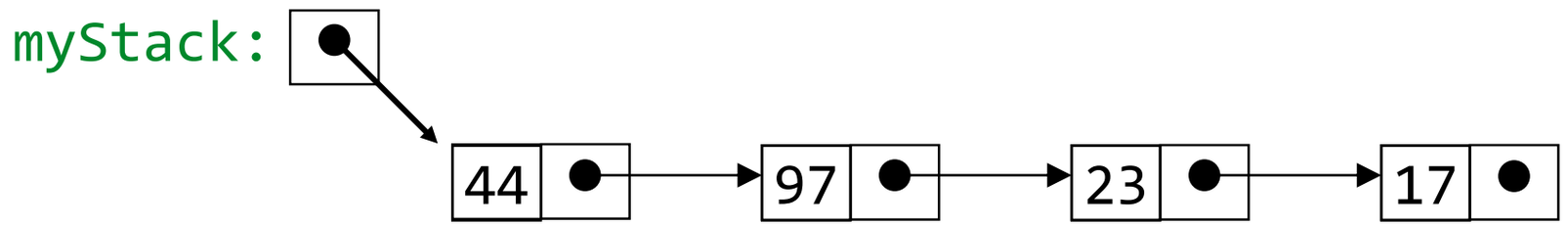
```
/** Demonstrates sample usage of a stack. */
  public static void main(String[] args) {
    Stack<Integer> S = new ArrayStack<>();  // contents: ()
    S.push(5);                              // contents: (5)
    S.push(3);                              // contents: (5, 3)
    System.out.println(S.size());           // contents: (5, 3)     outputs 2
    System.out.println(S.pop());            // contents: (5)        outputs 3
    System.out.println(S.isEmpty());        // contents: (5)        outputs false
    System.out.println(S.pop());            // contents: ()         outputs 5
    System.out.println(S.isEmpty());        // contents: ()         outputs true
    System.out.println(S.pop());            // contents: ()         outputs null
    S.push(7);                              // contents: (7)
    S.push(9);                              // contents: (7, 9)
    System.out.println(S.top());            // contents: (7, 9)     outputs 9
    S.push(4);                              // contents: (7, 9, 4)
    System.out.println(S.size());           // contents: (7, 9, 4)  outputs 3
    System.out.println(S.pop());            // contents: (7, 9)     outputs 4
    S.push(6);                              // contents: (7, 9, 6)
    S.push(8);                              // contents: (7, 9, 6, 8)
    System.out.println(S.pop());            // contents: (7, 9, 6)  outputs 8
  }
}
```

ArrayStack

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

Linked-list implementation of stacks

- Since all the action happens at the top of a stack, a singly-linked list (SLL) is a fine way to implement it
- The header of the list points to the top of the stack



- Pushing is inserting an element at the front of the list
- Popping is removing an element from the front of the list

Linked-list implementation details

- With a linked-list representation, overflow will not happen (unless you exhaust memory, which is another kind of problem)
- Underflow can happen, and should be handled the same way as for an array implementation
- When a node is popped from a list, and the node references an object, the reference (the pointer in the node) does *not* need to be set to `null`
  - Unlike an array implementation, it really *is* removed--you can no longer get to it from the linked list
  - Hence, garbage collection can occur as appropriate

LinkedStack

```
// Realization of a stack as an adaptation of a SinglyLinkedList.
public class LinkedStack<E> implements Stack<E> {

  /** The primary storage for elements of the stack */
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();

  /** Constructs an initially empty stack. */
  public LinkedStack() { }
```

LinkedStack

```
/**
   * Returns the number of elements in the stack.
   * @return number of elements in the stack
   */
  @Override
public int size() { return list.size(); }

/**
 * Tests whether the stack is empty.
 * @return true if the stack is empty, false otherwise
 */
@Override
public boolean isEmpty() { return list.isEmpty(); }
```

LinkedStack

```
/**
   * Inserts an element at the top of the stack.
   * @param element   the element to be inserted
   */
  @Override
public void push(E element) { list.addFirst(element); }

/**
 * Returns, but does not remove, the element at the top of the stack.
 * @return top element in the stack (or null if empty)
 */
@Override
public E top() { return list.first(); }
```

LinkedStack

```
/**
   * Removes and returns the top element from the stack.
   * @return element removed (or null if empty)
   */
  @Override
  public E pop() { return list.removeFirst(); }

  /** Produces a string representation of the contents of the stack.
   *  (ordered from top to bottom)
   *
   * This exists for debugging purposes only.
   *
   * @return textual representation of the stack
   */
  public String toString() {
    return list.toString();
  }
}
```

LinkedStack

Stack Method	*Singly Linked List Method*
size( )	list.size( )
isEmpty( )	list.isEmpty( )
push(e)	list.addFirst(e)
pop( )	list.removeFirst( )
top( )	list.first( )

Method	**Running Time**
size	$O(1)$
isEmpty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

Stack limitations/idioms

- You cannot loop over a stack in the usual way.

```
Stack<Integer> s = new Stack<Integer>();
...
for (int i = 0; i < s.size(); i++) {
    do something with s.get(i);
}
```

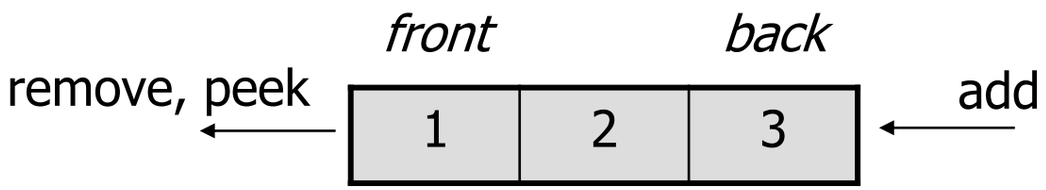
- Instead, you pull elements out of the stack one at a time.
  - common idiom: Pop each element until the stack is empty.

```
// process (and destroy) an entire stack
while (!s.isEmpty()) {
    do something with s.pop();
}
```

Queues

Queues

- Another special kind of list
  - Additions are made at one end, called the tail
  - Removals take place at the other end, called the head
  - the last element added, is always the last one to be deleted
- So, queue is a FIFO sequence
- Basic queue operations:
  - **add** (enqueue): Add an element to the back.
  - **remove** (dequeue): Remove the front element.
  - **peek**: Examine the front element.



queue

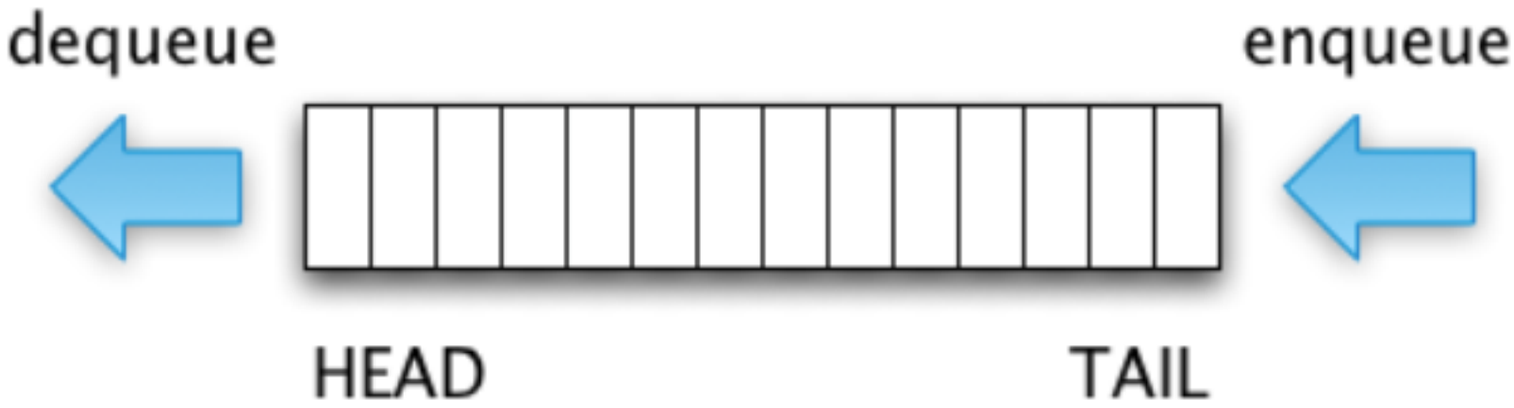
Queues

- Policy of doing tasks in the order they arrive
- Print jobs sent to the printer
- People waiting in line at a theater
- Cars waiting in line at a toll booth
- Tasks waiting to be serviced by an application on your computer

Queues

- A typical reason to use a queue in an application is to save the item in a collection while at the same time *preserving their relative order*

Queues

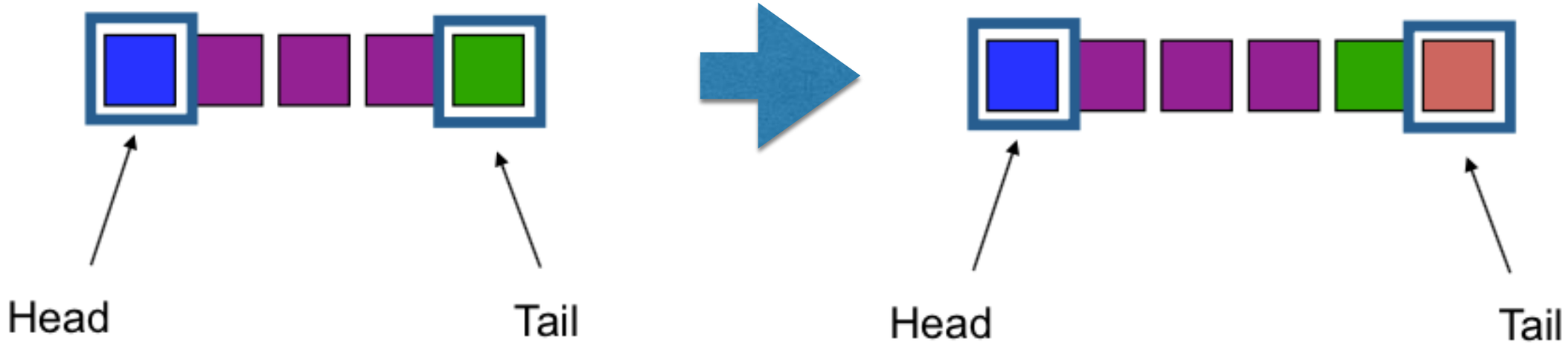


Queue Operations

- size: returns the number of items in the stack
- isEmpty: returns whether stacks has no items
- first: returns the item at the head (without removing it)

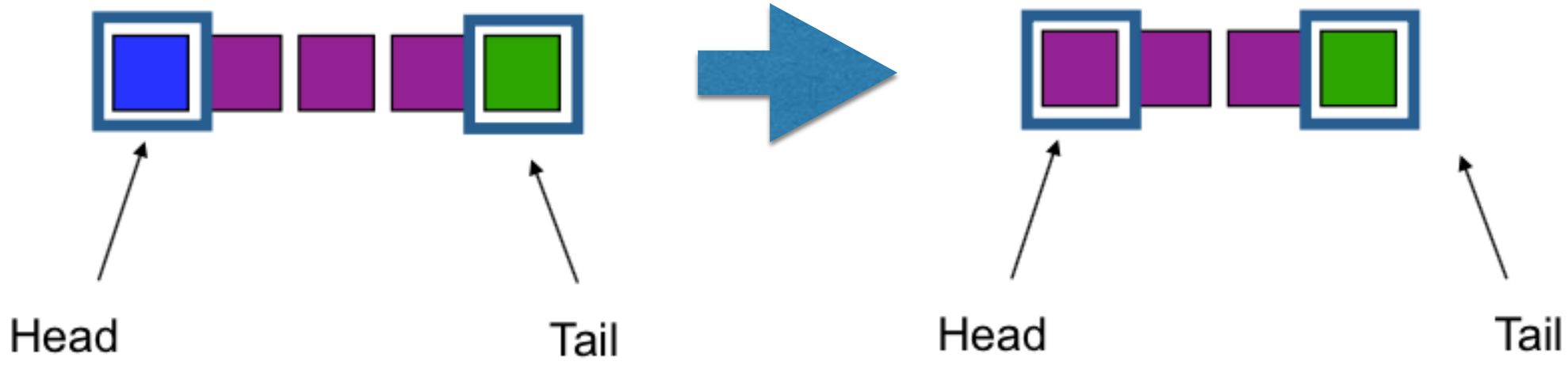
Queue Operatios

- enqueue(x): insert an item x at the tail



Queue Operatios

- dequeue: remove the element from the head



Queue Implementation

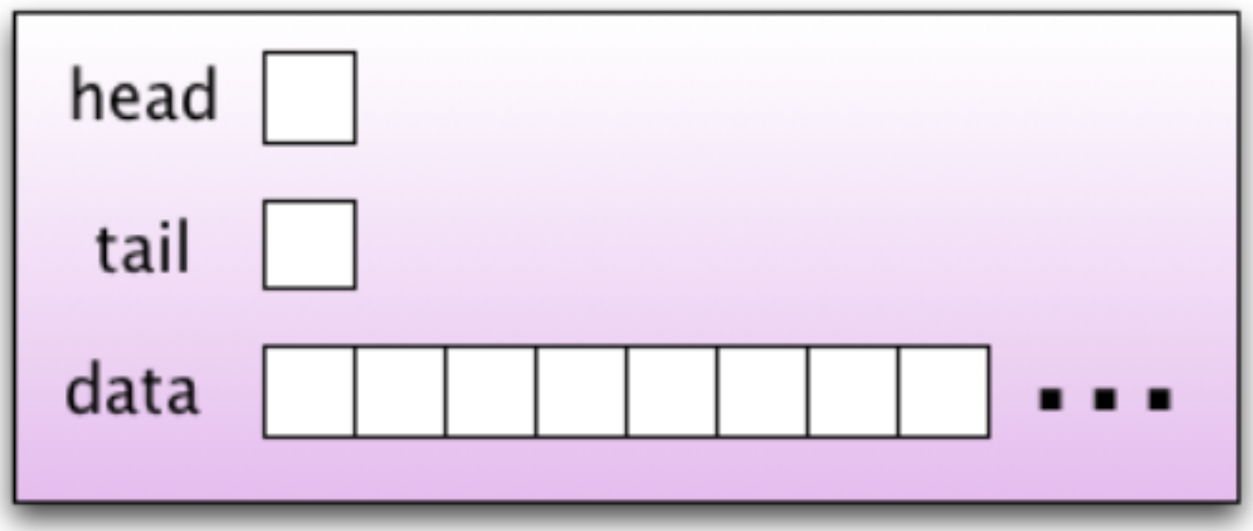
- Dedicated implementation, OR
- All the operations can be directly implemented using the LIST ADT (as a wrapper around a built-in list object)

Queue ADT

```
public interface Queue<E>
{
 int size();
 boolean isEmpty();
 E first();
 void enqueue(E e);
 E dequeue();
}
```

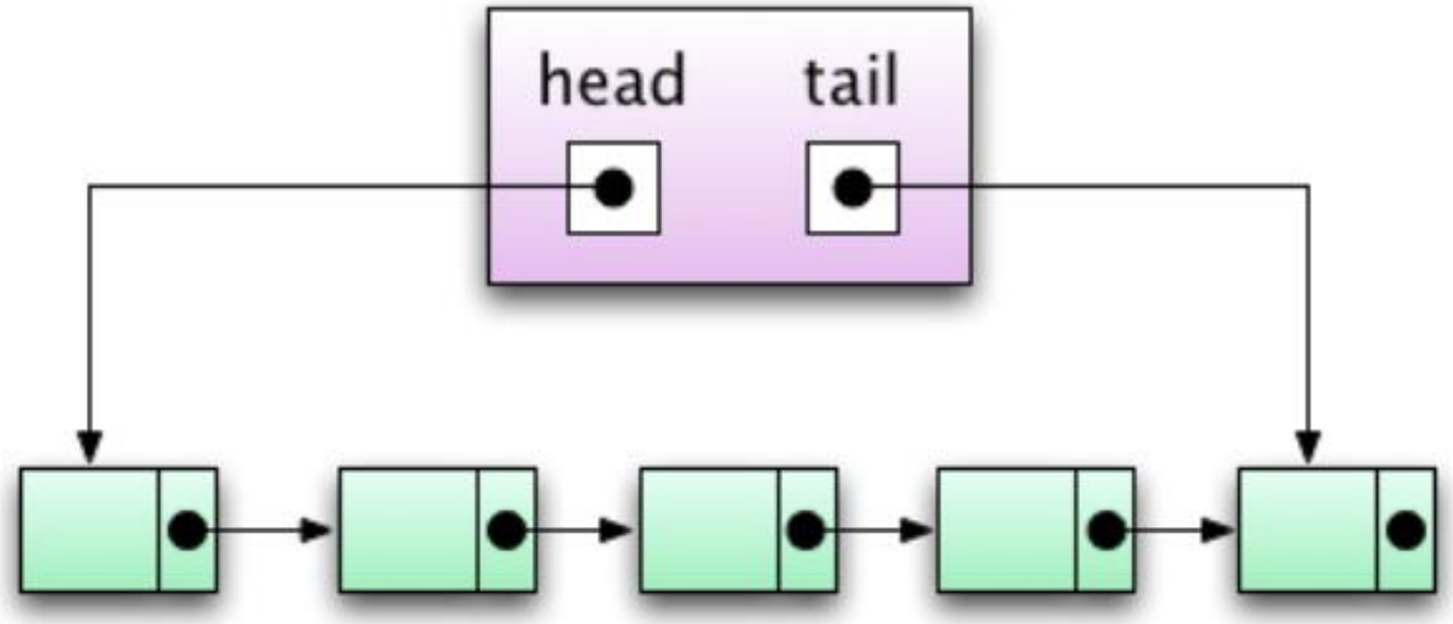
Implementation

1. Array-based



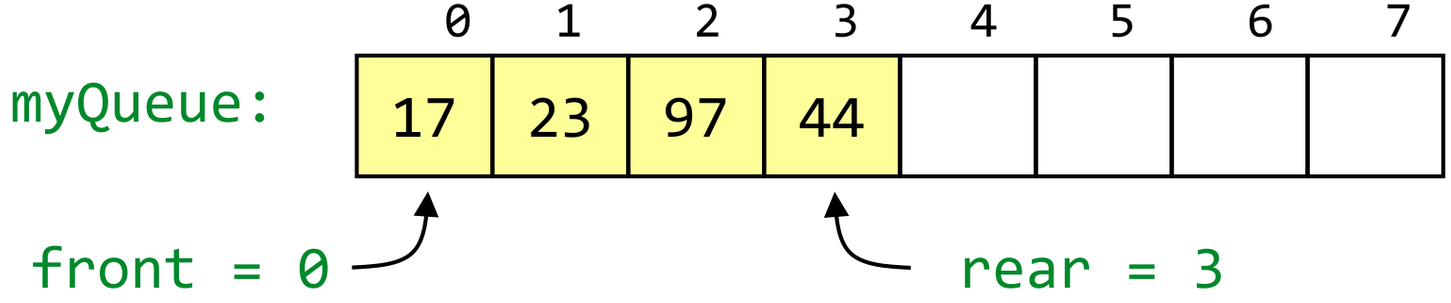
Implementation

2. Linked Implementation



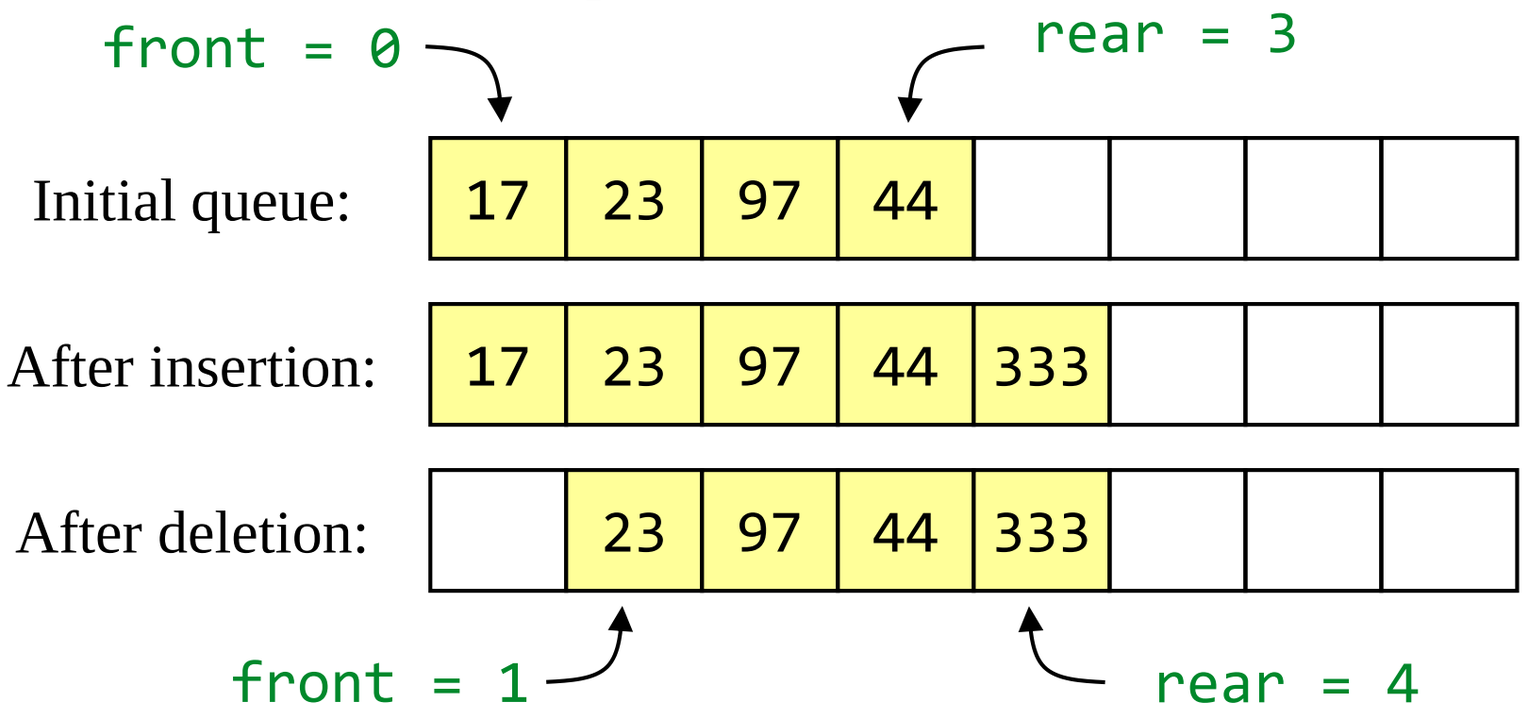
Array implementation of queues

- A queue is a first in, first out (FIFO) data structure
- This is accomplished by inserting at one end (the rear) and deleting from the other (the front)



- **To insert:** put new element in location 4, and set rear to 4
- **To delete:** take element from location 0, and set front to 1

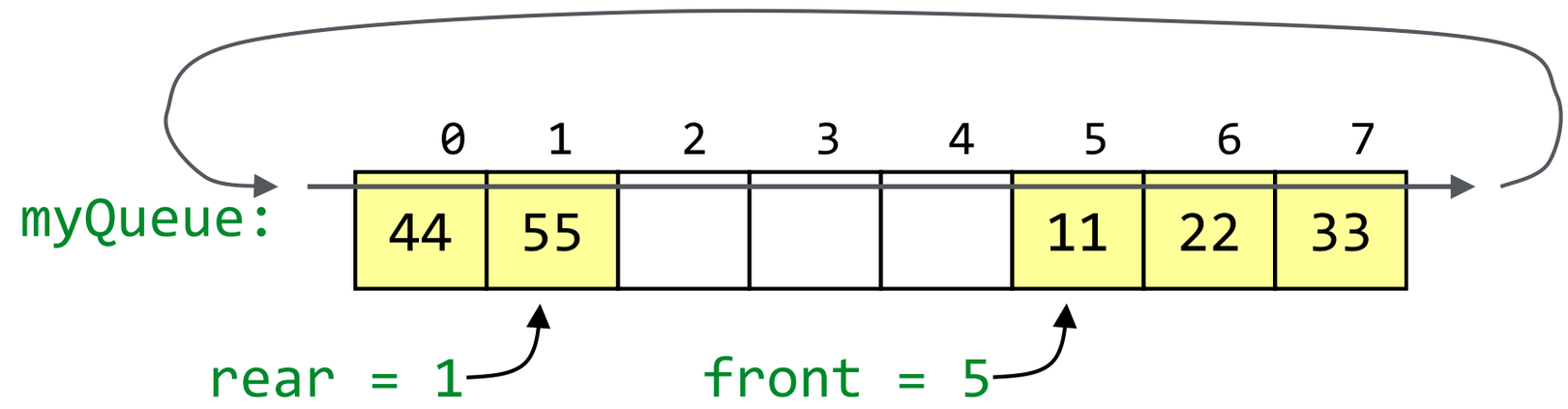
Array implementation of queues



- Notice how the array contents "crawl" to the right as elements are inserted and deleted
- This will be a problem after a while!

Circular arrays

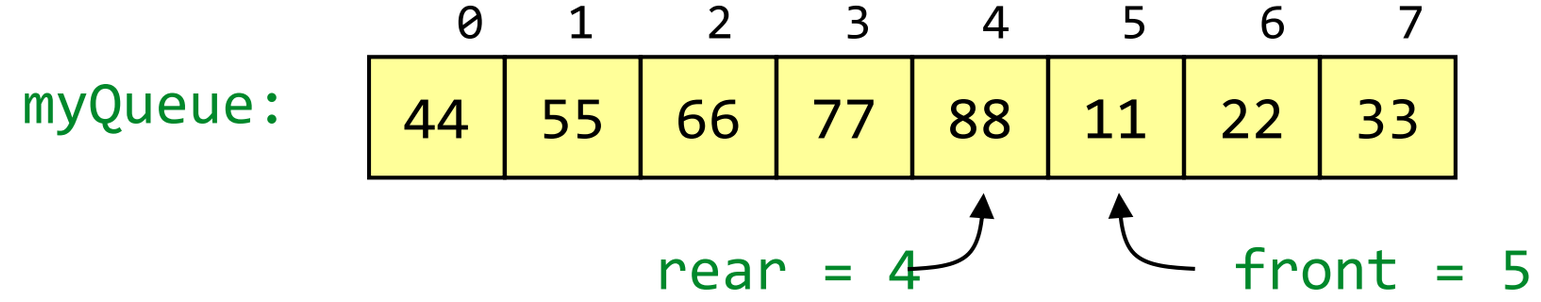
- We can treat the array holding the queue elements as circular (joined at the ends)



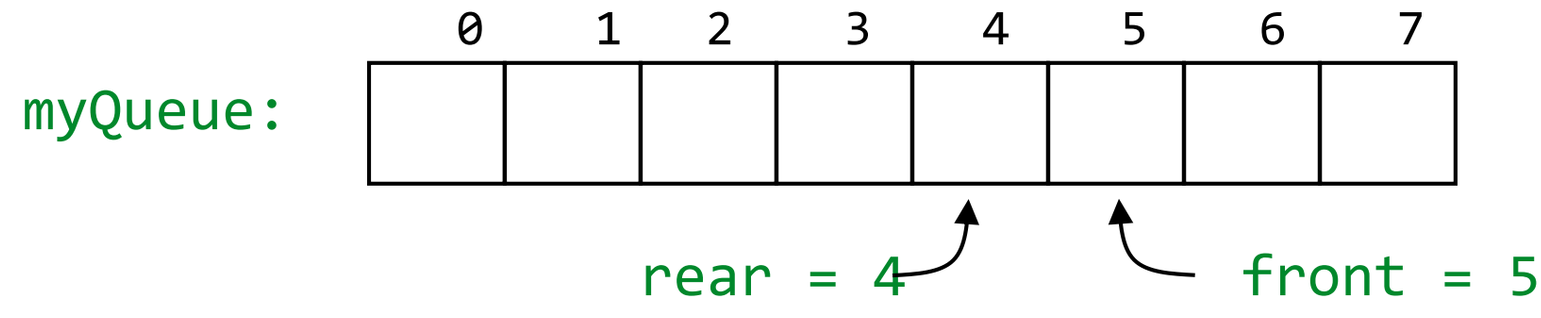
- Elements were added to this queue in the order `11`, `22`, `33`, `44`, `55`, and will be removed in the same order
- Use: `front = (front + 1) % myQueue.length;`
  and: `rear = (rear + 1) % myQueue.length;`

Full and empty queues

- If the queue were to become completely full, it would look like this:



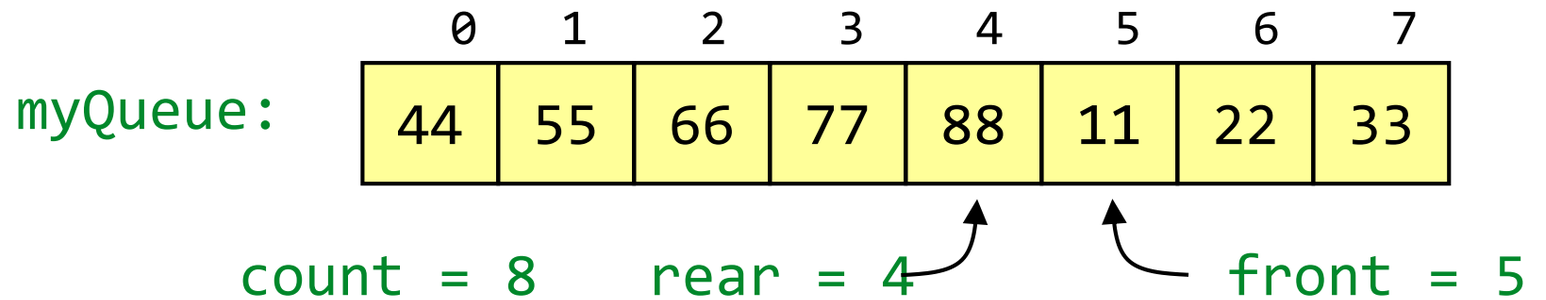
- If we were then to remove all eight elements, making the queue completely empty, it would look like this:



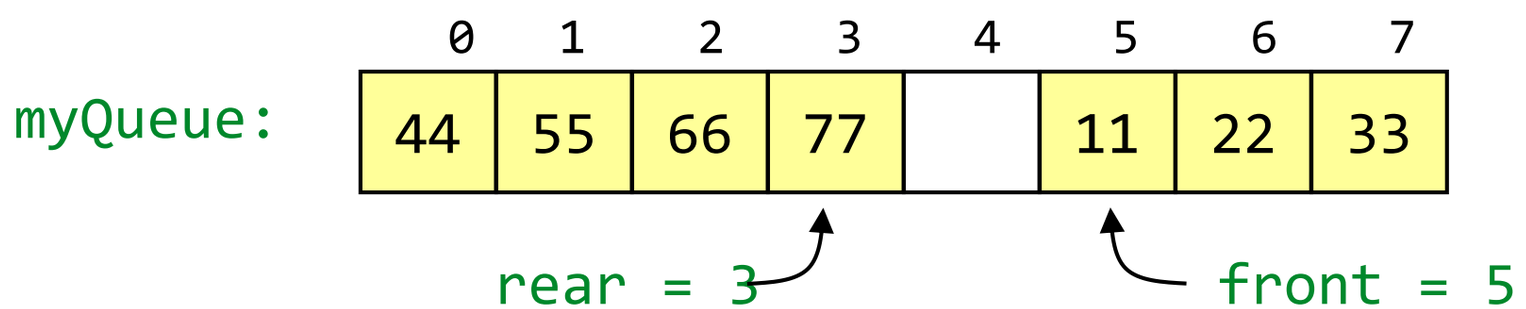
This is a problem!

Full and empty queues: solutions

- **Solution #1:** Keep an additional variable



- **Solution #2:** (Slightly more efficient) Keep a gap between elements: consider the queue full when it has `n-1` elements



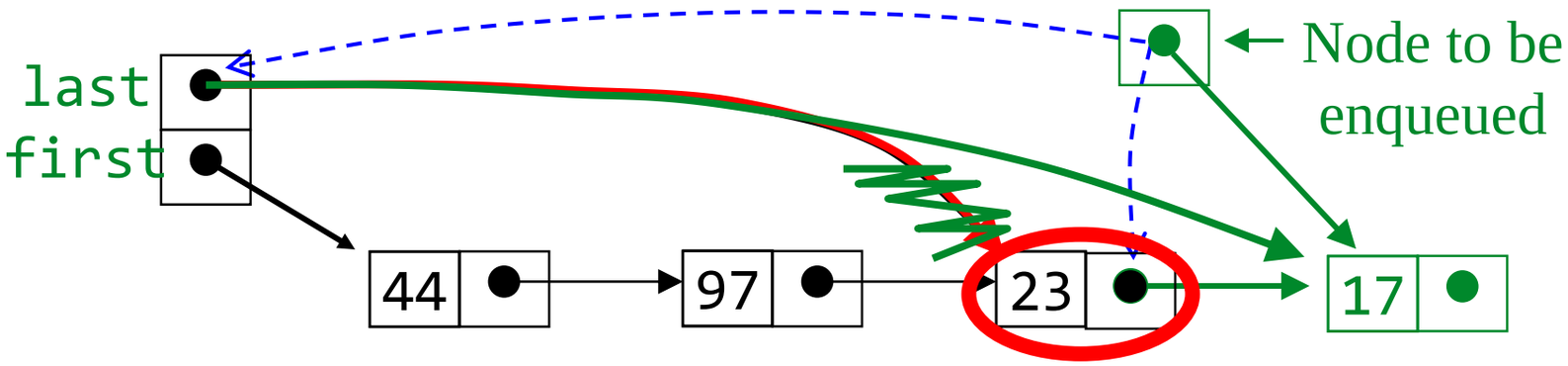
Linked-list implementation of queues

- In a queue, insertions occur at one end, deletions at the other end
- Operations at the front of a singly-linked list (SLL) are O(1), but at the other end they are O(n)
  - Because you have to find the last element each time
- BUT: there is a simple way to use a singly-linked list to implement both insertions and deletions in O(1) time
  - You always need a pointer to the first thing in the list
  - You can keep an additional pointer to the *last* thing in the list

SLL implementation of queues

- In an SLL you can easily find the successor of a node, but not its predecessor
  - Remember, pointers (references) are one-way
- If you know where the *last* node in a list is, it's hard to remove that node, but it's easy to add a node after it
- Hence,
  - Use the *first* element in an SLL as the *front* of the queue
  - Use the *last* element in an SLL as the *rear* of the queue
  - Keep pointers to *both* the front and the rear of the SLL

Enqueueing a node



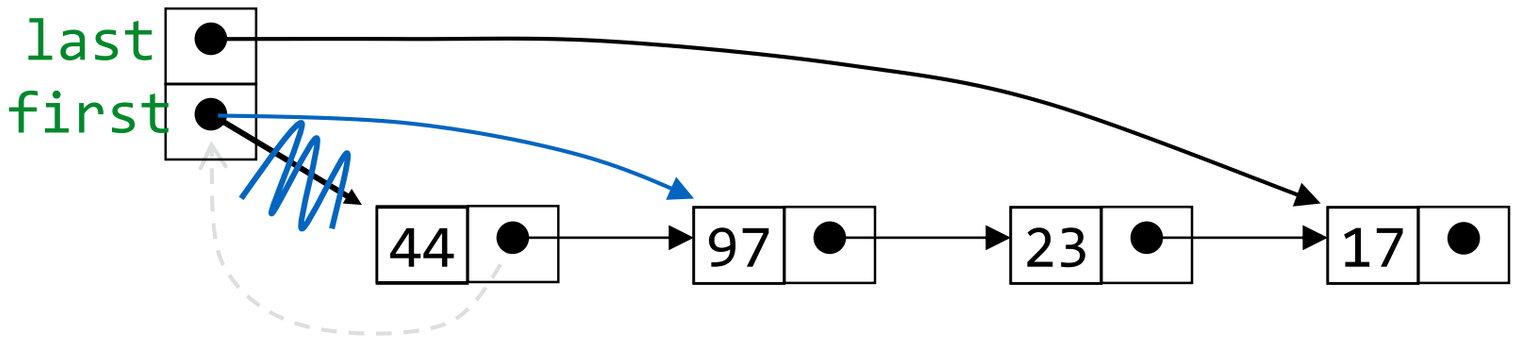
To enqueue (add) a node:

Find the current last node

Change it to point to the new last node

Change the `last` pointer in the list header

Dequeueing a node



- To dequeue (remove) a node:
  - Copy the pointer from the first node into the header

Queue implementation details

- With an array implementation:
  - you can have both overflow and underflow
  - you should set deleted elements to `null`

- With a linked-list implementation:
  - you can have underflow
  - overflow is a global out-of-memory condition
  - there is no reason to set deleted elements to `null`

ArrayQueue

```
public class ArrayQueue<E> implements Queue<E> {
  // instance variables
  /** Default array capacity. */
  public static final int CAPACITY = 1000;

  /** Generic array used for storage of queue elements. */
  private E[] data;

  /** Index of the top element of the queue in the array. */
  private int f = 0;
```

ArrayQueue

```
/** Current number of elements in the queue. */
private int sz = 0;

// constructors
/** Constructs an empty queue using the default array capacity. */
public ArrayQueue() {this(CAPACITY);}

/**
 * Constructs and empty queue with the given array capacity.
 * @param capacity length of the underlying array
 */
@SuppressWarnings({"unchecked"})
public ArrayQueue(int capacity) {
  data = (E[]) new Object[capacity];
}
```

ArrayQueue

```
// methods
  /**
   * Returns the number of elements in the queue.
   * @return number of elements in the queue
   */
  @Override
  public int size() { return sz; }

  /** Tests whether the queue is empty. */
  @Override
  public boolean isEmpty() { return (sz == 0); }
```

ArrayQueue

```
/**
 * Inserts an element at the rear of the queue.
 * This method runs in O(1) time.
 * @param e   new element to be inserted
 * @throws IllegalStateException if the array storing the elements is full
 */
@Override
public void enqueue(E e) throws IllegalStateException {
  if (sz == data.length) throw new IllegalStateException("Queue is full");
  int avail = (f + sz) % data.length;   // use modular arithmetic
  data[avail] = e;
  sz++;
}
```

ArrayQueue

```
/**
  * Returns, but does not remove, the first element of the queue.
  * @return the first element of the queue (or null if empty)
  */
 @Override
 public E first() {
   if (isEmpty()) return null;
   return data[f];
 }
```

ArrayQueue

```
/**
 * Removes and returns the first element of the queue.
 * @return element removed (or null if empty)
 */
@Override
public E dequeue() {
  if (isEmpty()) return null;
  E answer = data[f];
  data[f] = null;                    // dereference to help garbage collection
  f = (f + 1) % data.length;
  sz--;
  return answer;
}
```

ArrayQueue

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
first	$O(1)$
enqueue	$O(1)$
dequeue	$O(1)$

LinkedQueue

```
//Realization of a FIFO queue as an adaptation of a SinglyLinkedList.
public class LinkedQueue<E> implements Queue<E> {

  /** The primary storage for elements of the queue */
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();

  /** Constructs an initially empty queue. */
  public LinkedQueue() { }
```

LinkedQueue

```
/**
   * Returns the number of elements in the queue.
   * @return number of elements in the queue
   */
  @Override
public int size() { return list.size(); }

/**
 * Tests whether the queue is empty.
 * @return true if the queue is empty, false otherwise
 */
@Override
public boolean isEmpty() { return list.isEmpty(); }
```

LinkedQueue

```
/**
   * Inserts an element at the rear of the queue.
   * @param element  the element to be inserted
   */
  @Override
public void enqueue(E element) { list.addLast(element); }

/**
 * Returns, but does not remove, the first element of the queue.
 * @return the first element of the queue (or null if empty)
 */
@Override
public E first() { return list.first(); }
```

LinkedQueue

```
  /**
   * Removes and returns the first element of the queue.
   * @return element removed (or null if empty)
   */
    @Override
    public E dequeue() { return list.removeFirst(); }

    /** Produces a string representation of the contents of the queue.
     *  (from front to back). This exists for debugging purposes only.
     */
    public String toString() {
      return list.toString();
    }
  }
```

LinkedQueue

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
first	$O(1)$
enqueue	$O(1)$
dequeue	$O(1)$