Data structures and Algorithms

Fundamentals of Data Structures

Course Objective Outline

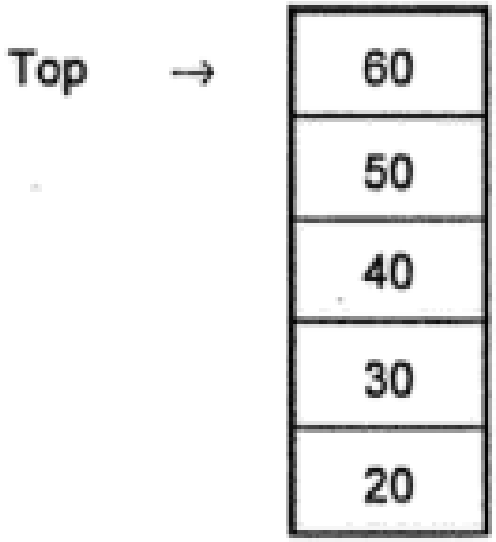
- Stacks
  - Introduction
  - What is a stack
  - ADT stack
  - Implementation of a stack
    - Arrays
    - Linked list
- Queues
  - Introduction
  - What is a queue
  - ADT queues
  - Implementation of a queue
    - Arrays
    - Linked list

Stack

- The data object stack is an ordered list but the access, insertions and deletions of elements are restricted by following rules.
  - For example, in a stack, the operations are carried out in such a way that the last element which is inserted will be the first element out.
  - Such order is called **Last In First Out or 'LIFO'**
  - The ordered list does not have specific ordering and the elements are inserted and deleted in any random order.
  - Thus stacks may be treated as special cases of ordered lists.

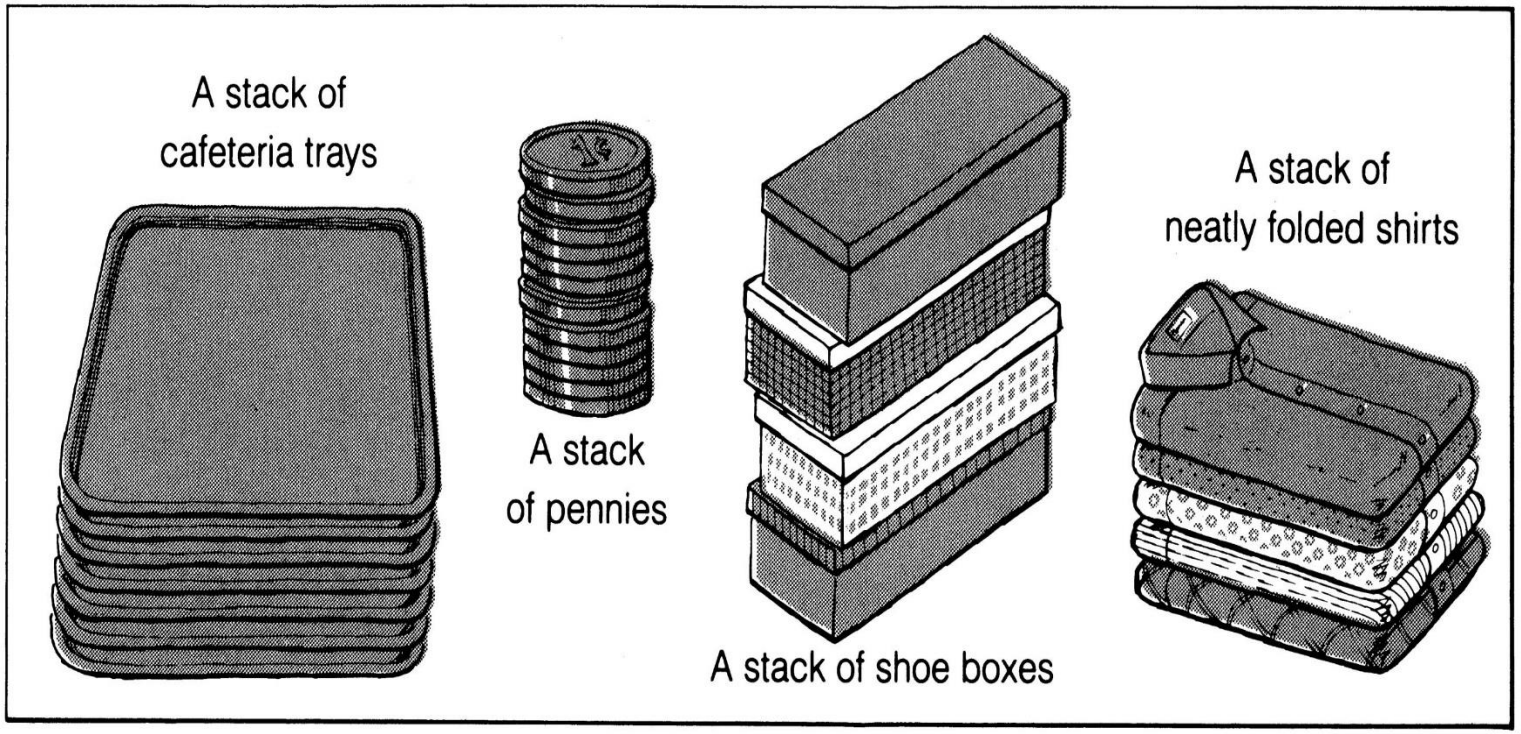
What is a stack?

- A stack is an ordered list in which all insertions and deletions are made at one end called the top.
- If we have to make a stack of elements 20, 30, 40, 50, 60 then 20 will be the bottommost element and 60 will be the topmost element in the stack.



What is a stack?

- Other examples of stacks can be as in the figure below



Primitive Operations on the Stack

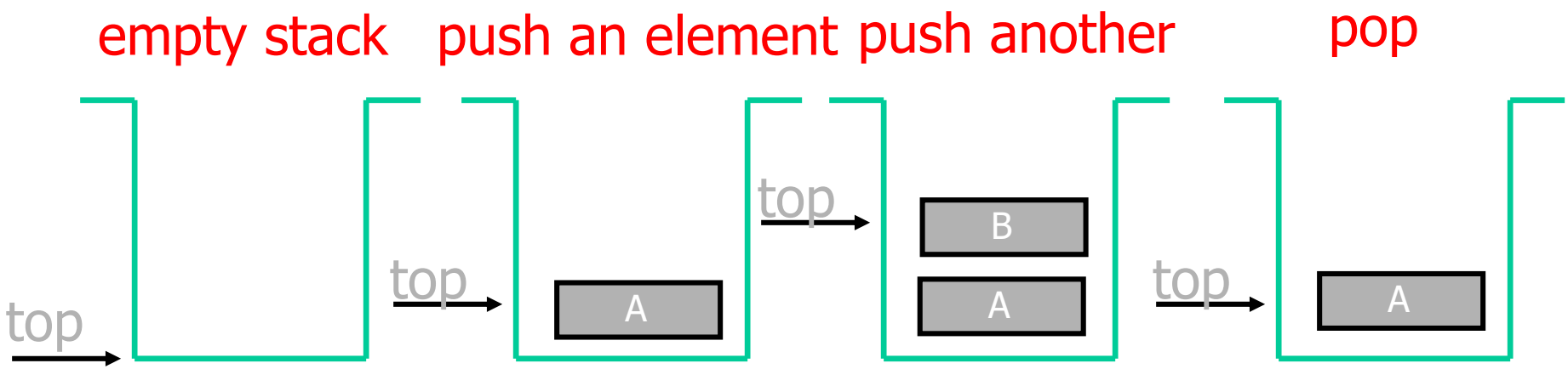
- We will now discuss the operations which can be carried out on the stack. The operations are as follows:
  - To create a stack
  - To insert an element on the stack
  - To delete an element from the stack
  - To check which element is at the top of the stack.
  - To check whether a stack is empty or not.

The Stack ADT

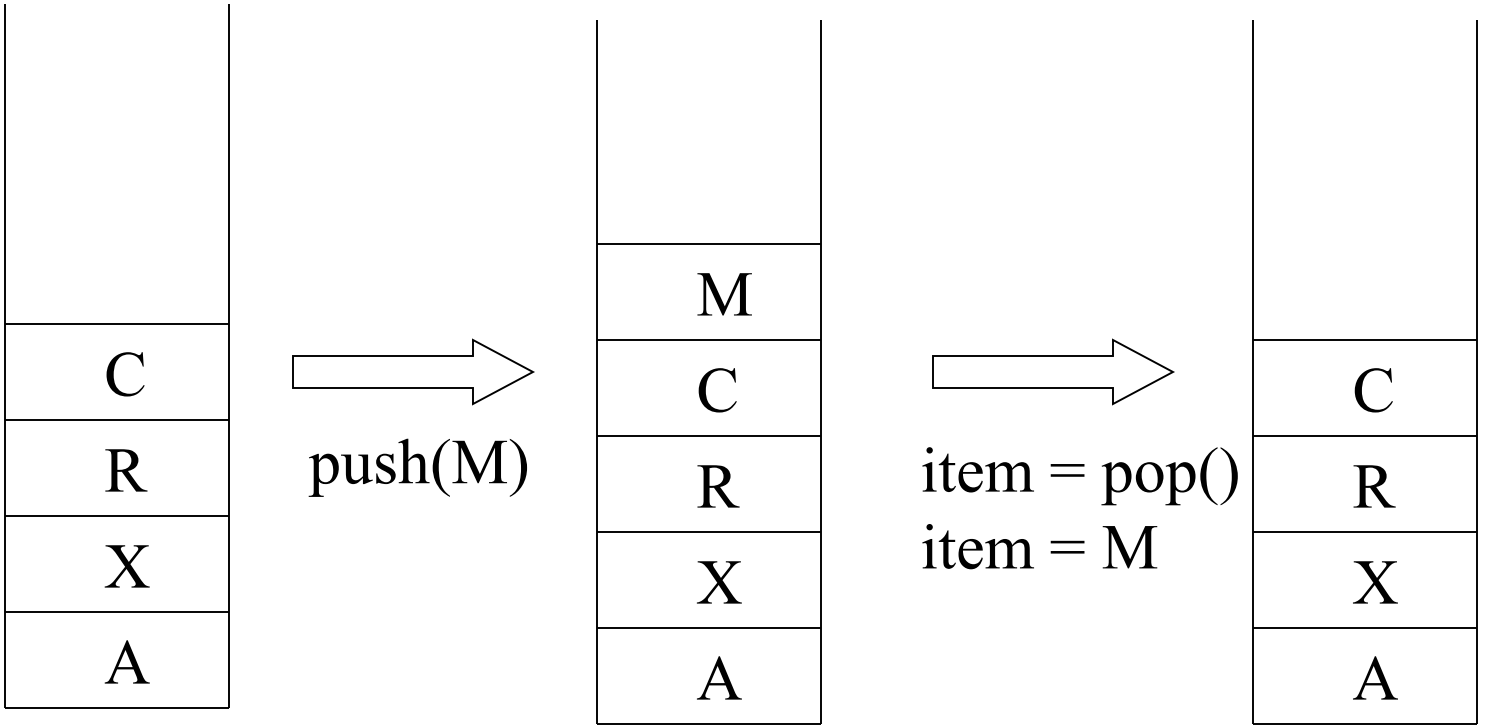
- The stack operations are given below.
  - **Stack()** - creates a new stack that is empty. It needs no parameters and returns an empty stack.
  - **push(item)** -adds a new item to the top of the stack. It needs the item and returns nothing.
  - **pop()** - removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.
  - **peek()** - returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.
  - **is_empty()** - tests to see whether the stack is empty. It needs no parameters and returns a Boolean value.
  - **size()** - returns the number of items on the stack. It needs no parameters and returns an integer.

Push and Pop

- Primary operations: Push and Pop
- Push
  - Add an element to the top of the stack
- Pop
  - Remove the element at the top of the stack



Push and Pop



Stack Applications

- Real life
  - Pile of books
  - Plate trays
- More applications related to computer science
  - Program execution stack (read more from your text)
  - Evaluating expressions
- Functional calls/recursion
- Undo in an editor
- Balanced parenthesis

Implementing a Stack

- At least two different ways to implement a stack
  - array
  - linked list
  - Also vectors can also be used to implement stack.
- Which method to use depends on the application
  - what advantages and disadvantages does each implementation have?

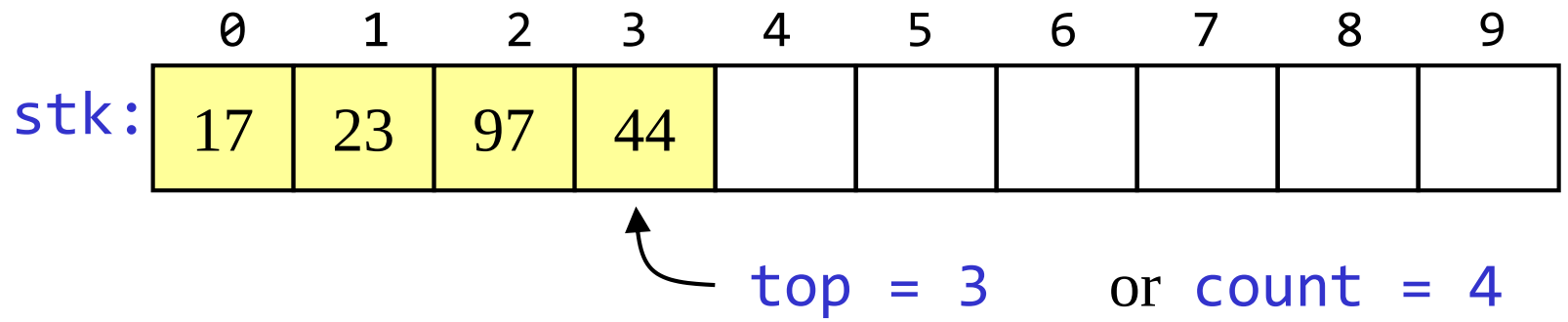
Implementing Stacks: Array

- Advantages
  - best performance
- Disadvantage
  - fixed size
- Basic implementation
  - initially empty array
  - field to record where the next data gets placed into
  - if array is full, push() returns false
  - otherwise adds it into the correct spot
  - if array is empty, pop() returns null
  - otherwise removes the next item in the stack

Array implementation of stacks

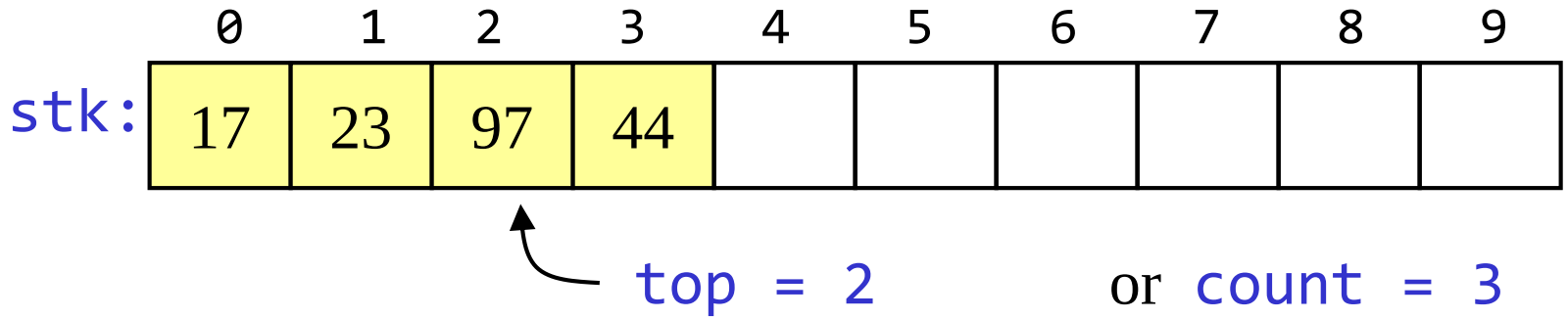
- To implement a stack, items are inserted and removed at the same end (called the top)
- Efficient array implementation requires that the top of the stack be towards the center of the array, not fixed at one end
- To use an array to implement a stack, you need both the array itself and an integer
  - 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]`

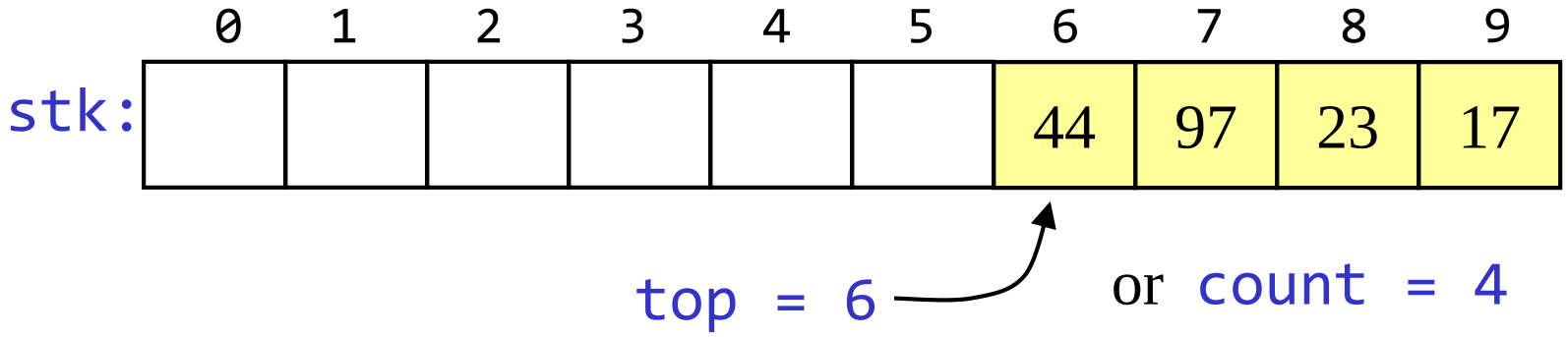
After popping



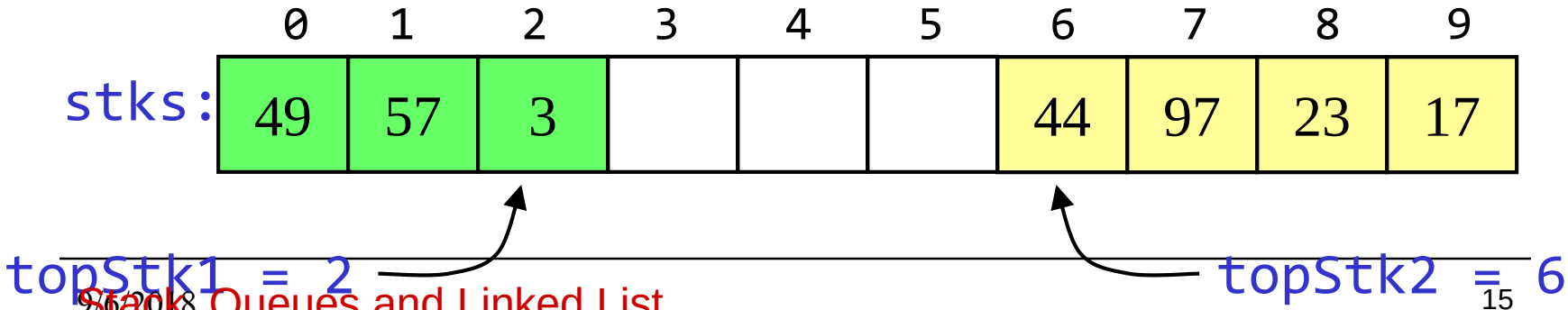
- When you pop an element, do you just leave the "deleted" element sitting in the array?
- The surprising answer is, "*it depends*"
  - If this is an array of primitives, *or* if you are programming in C or C++, *then* doing anything more is just a waste of time
  - If you are programming in Java, and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

Sharing space

- Of course, the bottom of the stack could be at the *other* end



- Sometimes this is done to allow two stacks to share the *same storage area*



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

Array Implementation Algorithm

- Push and Pop

```
int A[10]
   top ← −1 // empty stack
   Push(x)
     {
       top ← top + 1
       A[top] ← x
     }
    Pop()
     {
      top ← top − 1
     }
```

Array Implementation Algorithm

- Checking for empty stack

```
IsEmpty()
{
  if (top == −1)
      return true
  else
      retrun false
 }
```

- Checking for top element in a stack

```
Top()
 {
   return A[top]
 }
```

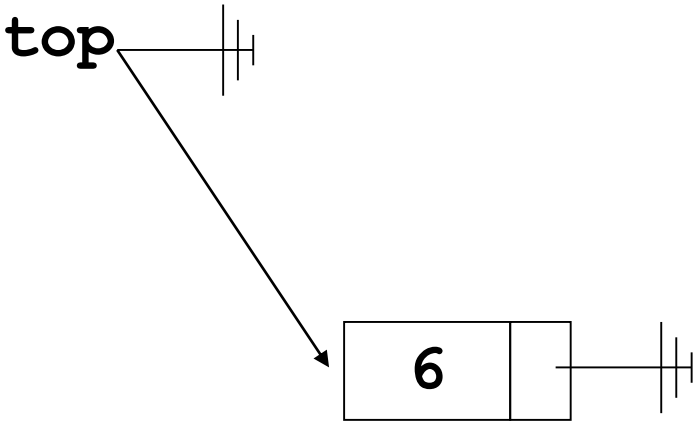
Implementing a Stack: Linked List

- Advantages:
  - always constant time to push or pop an element
  - can grow to an infinite size
- Disadvantages
  - the common case is the slowest of all the implementations
  - can grow to an infinite size
- Basic implementation
  - list is initially empty
  - *push()* method adds a new item to the head of the list
  - *pop()* method removes the head of the list

List Stack Example

Java Code

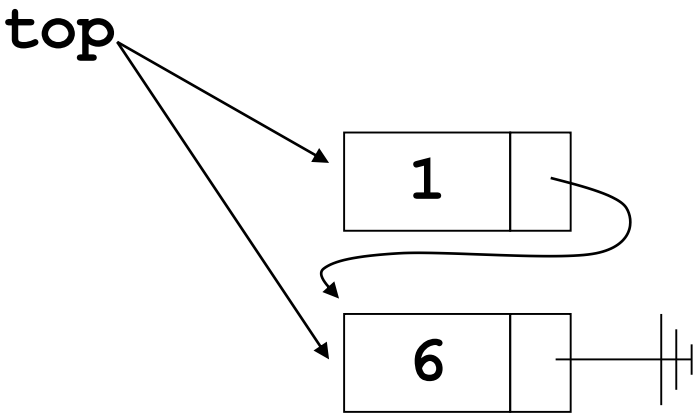
```
Stack st = new Stack();
st.push(6);
```



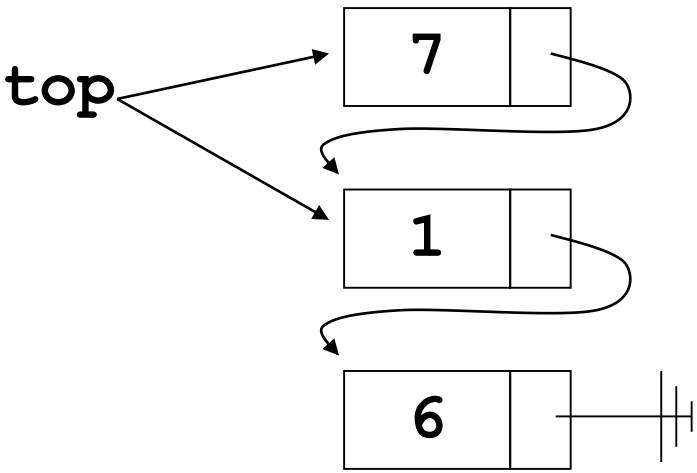
List Stack Example

Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
```



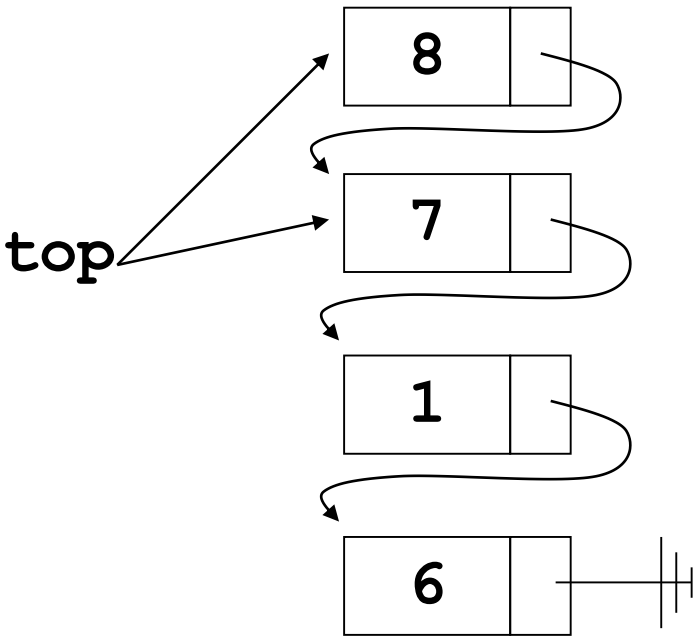
List Stack Example



Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
st.push(7);
```

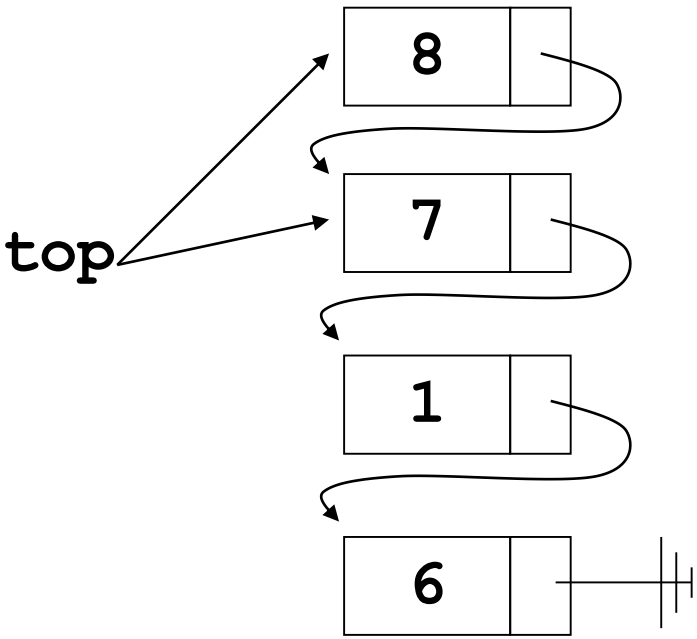
List Stack Example



Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
st.push(7);
st.push(8);
```

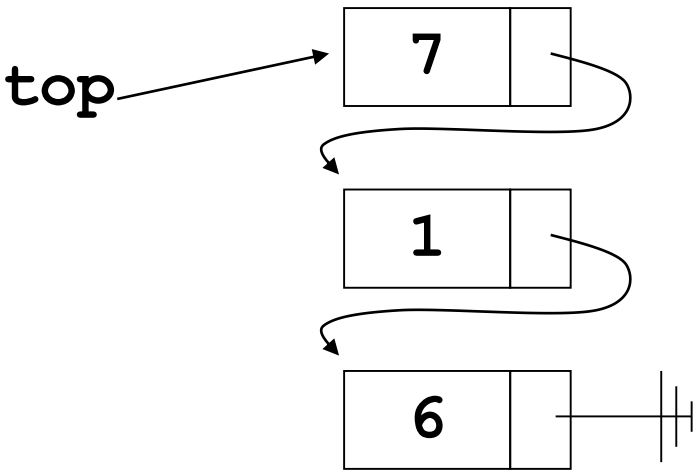
List Stack Example



Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
st.push(7);
st.push(8);
st.pop();
```

List Stack Example

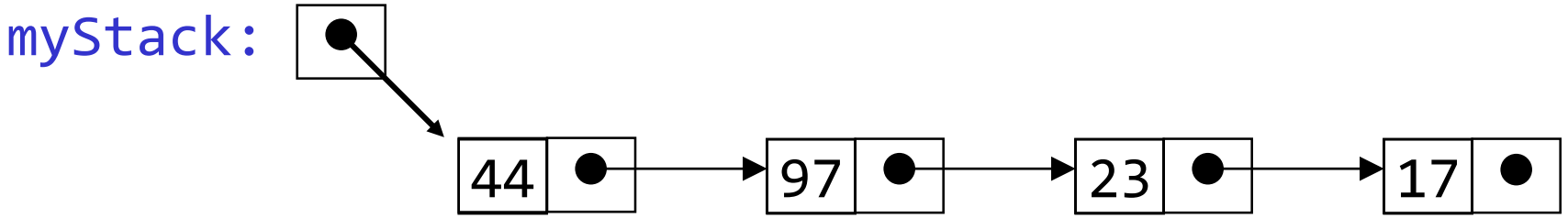


Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
st.push(7);
st.push(8);
st.pop();
```

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



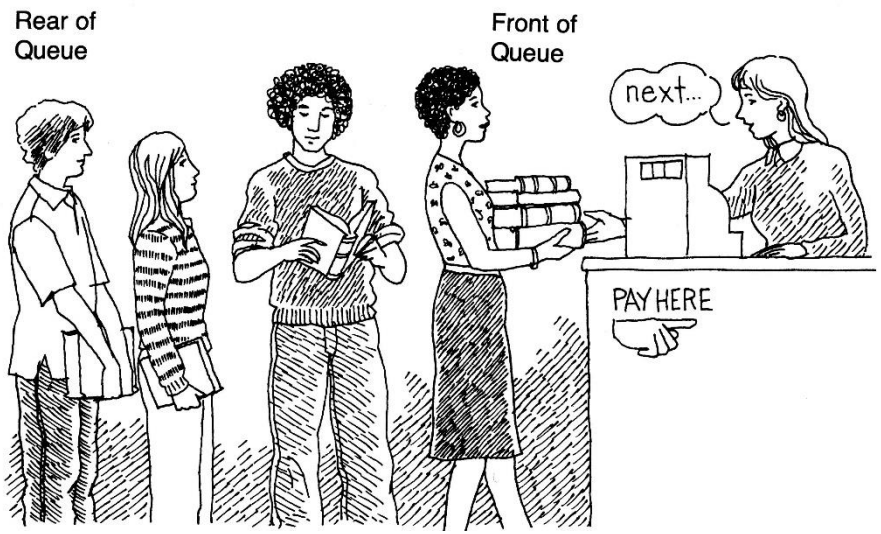
- Pushing is inserting an element at the front of the list
- Popping is removing an element from the front of the list

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

What is a queue?

- It is an ordered group of homogeneous items of elements.
- Queues have two ends:
  - Elements are added at one end called the rear .
  - Elements are removed from the other end called the front.
- The element added first is also removed first (**FIFO**: First In, First Out).



The Queue ADT

- The queue abstract data type is defined by the following structure and operations.
- A queue is structured as an ordered collection of items which are added at one end, called the "rear," and removed from the other end, called the "front."
- Queues maintain a FIFO ordering property.

The Queue ADT

- The queue operations are given below.
  - Queue() creates a new queue that is empty. It needs no parameters and returns an empty queue.
  - enqueue(item) adds a new item to the rear of the queue. It needs the item and returns nothing.
  - dequeue() removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.
  - is_empty() tests to see whether the queue is empty. It needs no parameters and returns a Boolean value.
  - size() returns the number of items in the queue. It needs no parameters and returns an integer.

Queues in computer science

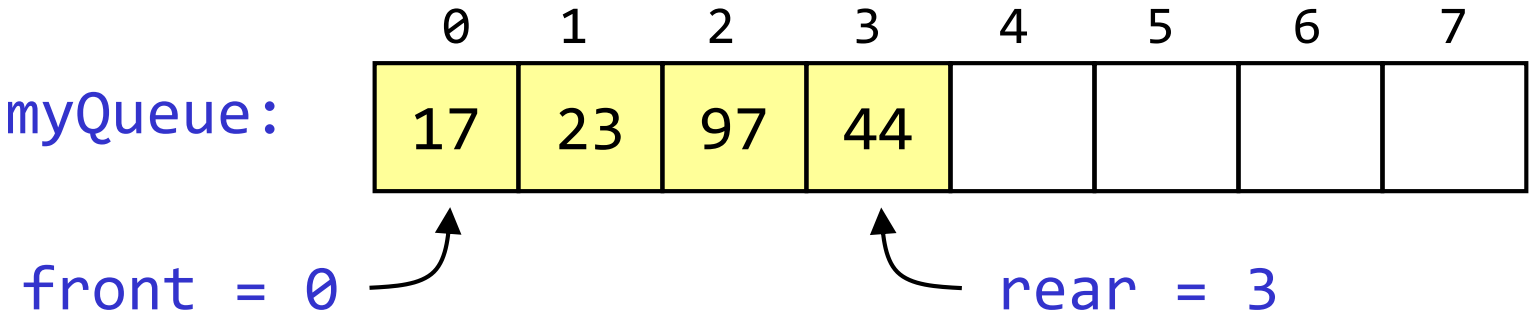
- Operating systems:
  - queue of print jobs to send to the printer
  - queue of programs / processes to be run
  - queue of network data packets to send
- Programming:
  - modeling a line of customers or clients
  - storing a queue of computations to be performed in order
- Real world examples:
  - people on an escalator or waiting in a line
  - cars at a gas station (or on an assembly line)

Applications of Queue

- Printing Job Management
- Packet Forwarding in Routers
- Message queue in Windows
- I/O buffer

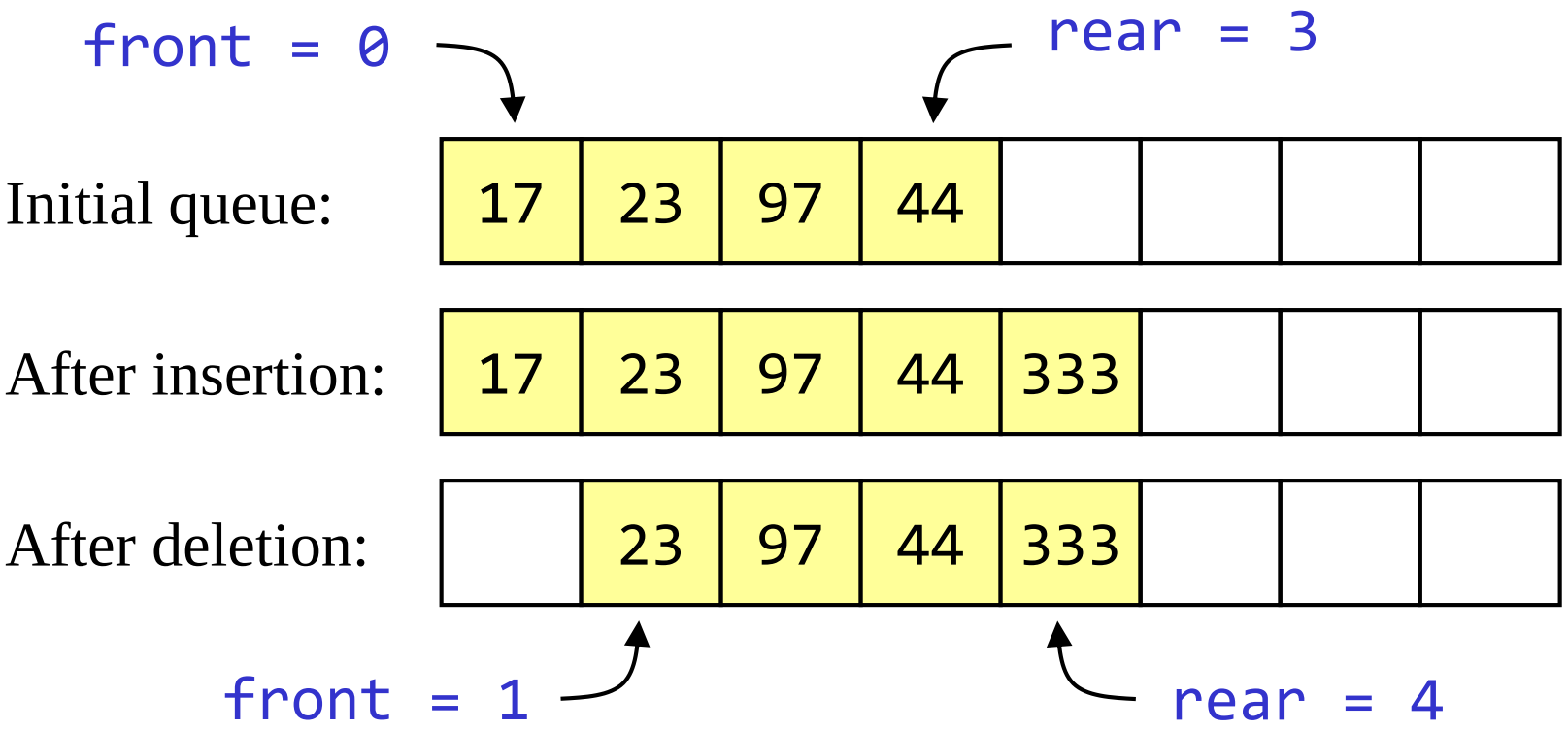
Array implementation of queues

- 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!

Array implementation of queues

```
Int A[10]
front ← −1
rear ← −1
IsEmpty()
{
  if front = −1 && rear == −1
            return true
   else
            return false
}
```

Array implementation of queues

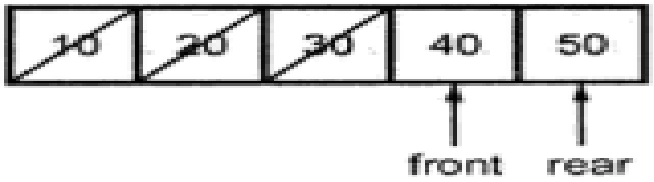
```
Enqueue(x)
 {
     if IsFull( )
        return
    else if IsEmpty( )
       {
          front ← rear ← 0
          A[rear] ← x
        }
       else
        {
          rear ← rear + 1
        }
             A[rear] ← x
      }
```

Array implementation of queues

```
Dequeue()
    {
    if IsEmpty( )
        return
    else if front == rear
            front ← rear ← −1
     else
        front ← front + 1
    }
```

Circular arrays

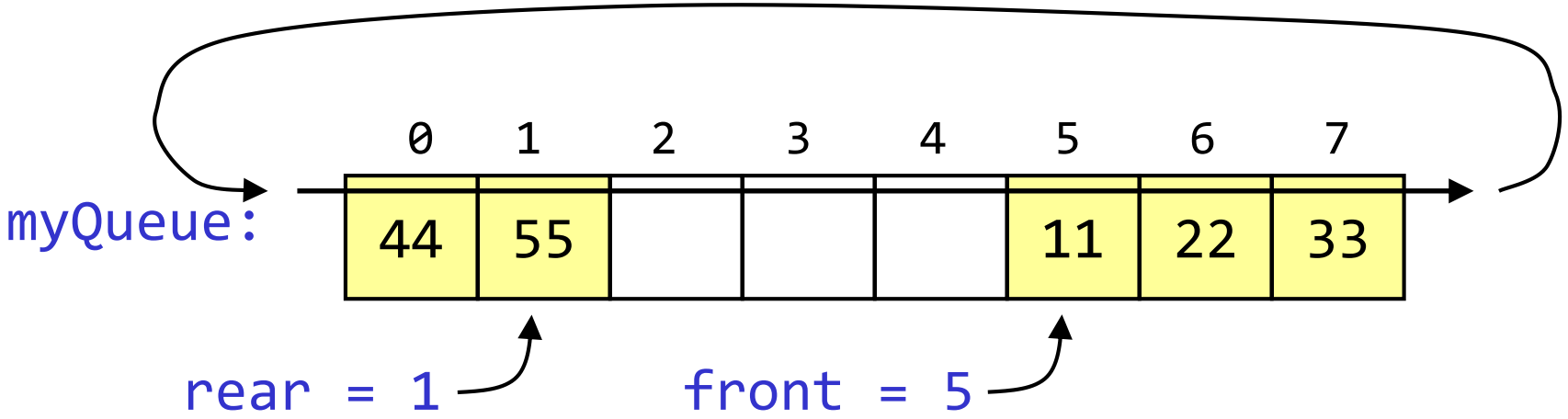
- As we have seen, in case of linear queue the elements get deleted logically as in this figure below



- We have deleted the elements 10, 20 and 30 means simply the front pointer is shifted ahead.
- Now if we try to insert any more element then it won't be possible as it is going to give 'queue full' message. Although there is a space of elements 10, 20 and 30 (these are deleted elements) we cannot utilize them because queue is nothing but a linear array.
- The concept of **circular queue**, has the main advantage of utilizing the space of the queue fully.

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;`

Array implementation of queues

```
Int A[10]
front ← −1
rear ← −1
IsEmpty()
{
  if front = −1 && rear == −1
            return true
   else
            return false
}
```

Array implementation of queues

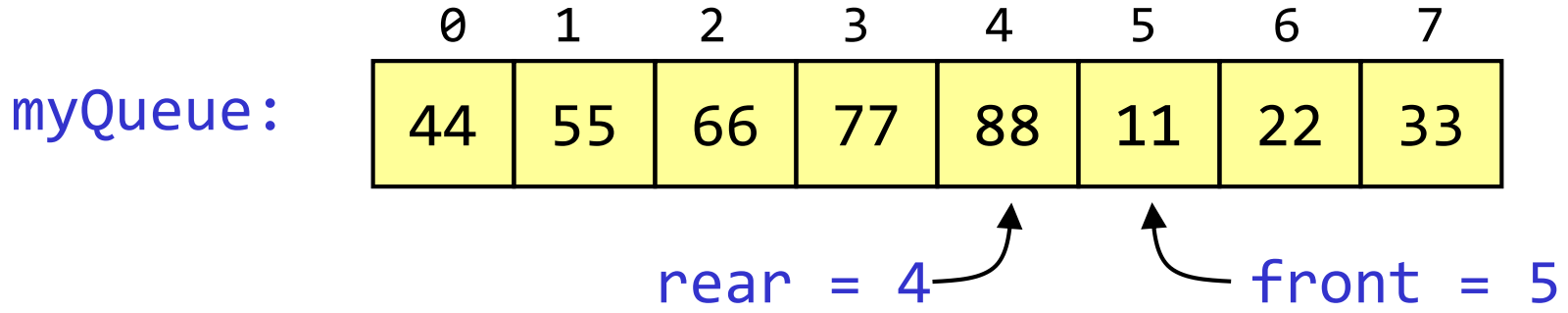
```
Enqueue(x)
 {
     if (rear + 1)% N == front
        return
    else if IsEmpty(   )
       {
          front ← rear ← 0
          A[rear] ← x
        }
       else
        {
          rear ← (rear + 1)% N
        }
             A[rear] ← x
      }
```

Array implementation of queues

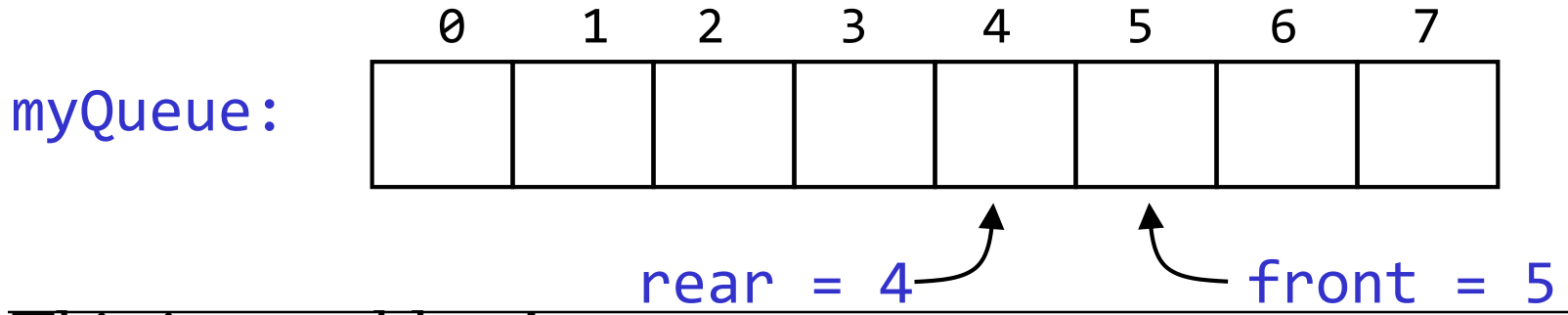
```
Dequeue()
    {
    if IsEmpty(   )
        return
    else if front == rear
            front ← rear ← −1
     else
         front ← (front + 1)% N
     }
front()
{
    return A[front]
}
```

Full and empty queues

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



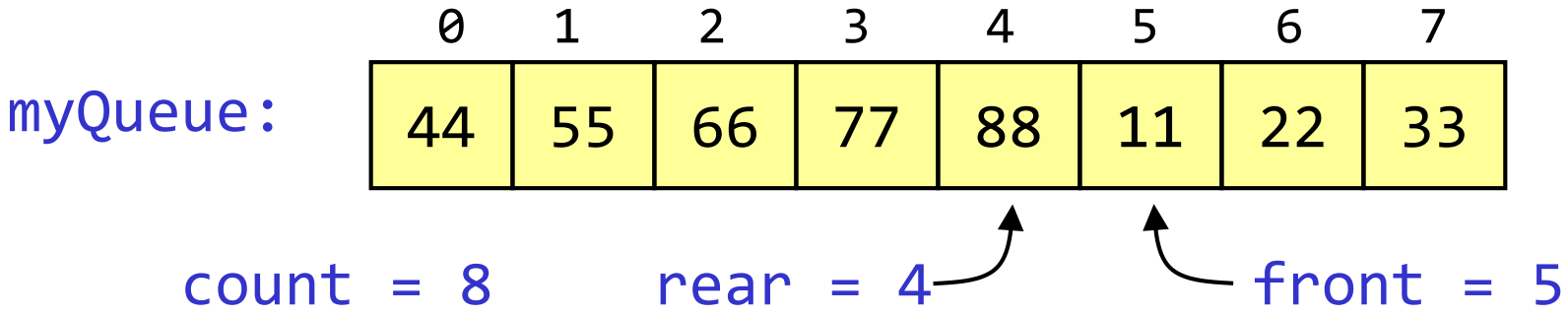
- If we were then to remove all eight elements, making the queue completely empty, it would look like this:



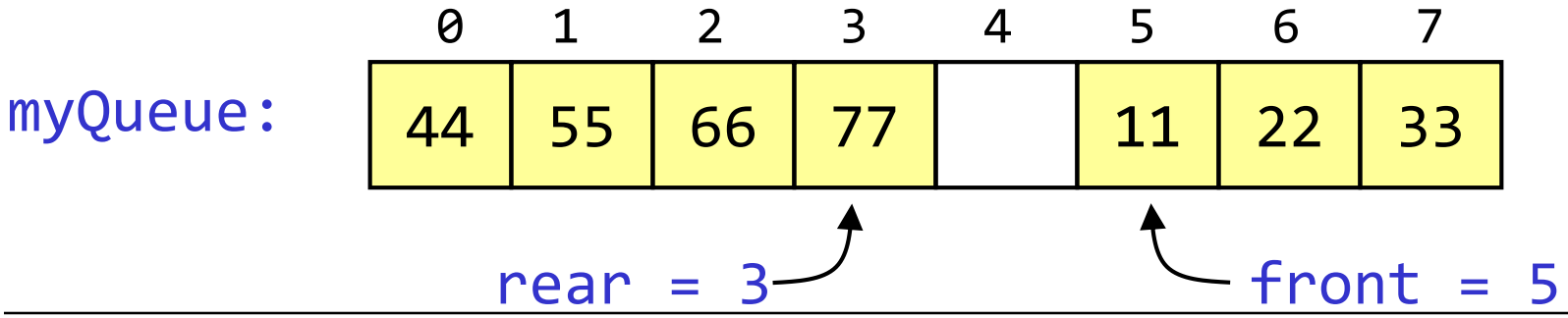
This is a problem!

Full and empty queues: solutions

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

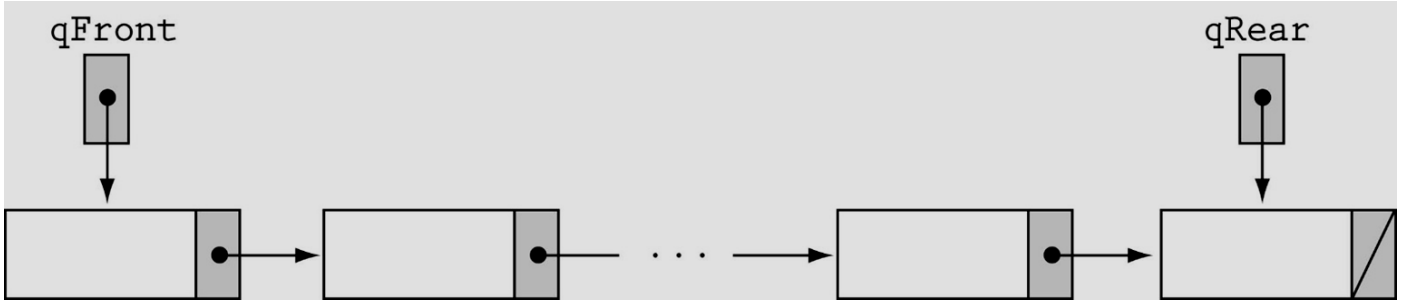


- **Solution #2:** (Slightly more efficient) Keep a gap between elements: consider the queue full when it has n-1 elements



Implementing queues using linked lists

- Allocate memory for each new element dynamically
- Link the queue elements together
- Use two pointers, *qFront* and *qRear*, to mark the front and rear of the queue



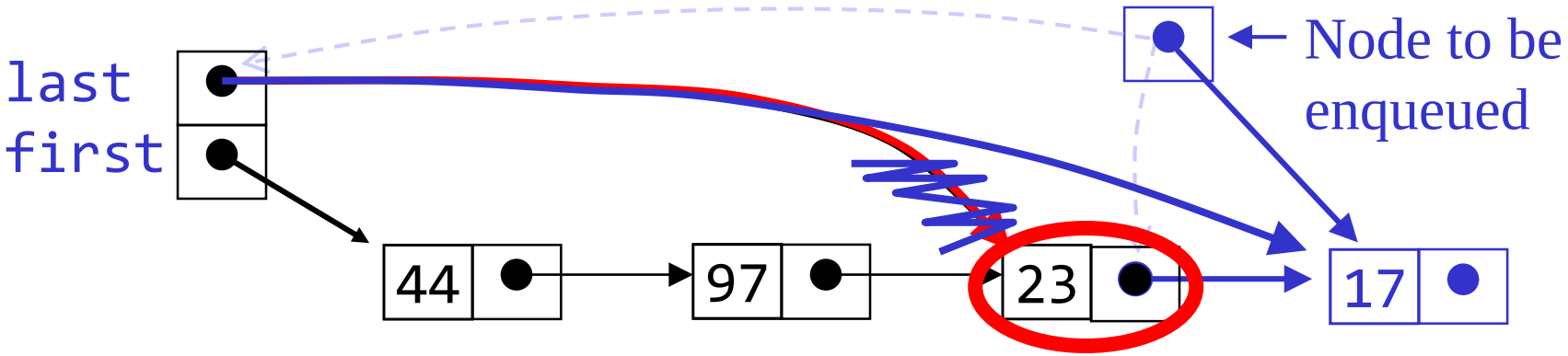
Linked-list implementation of queues

- 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

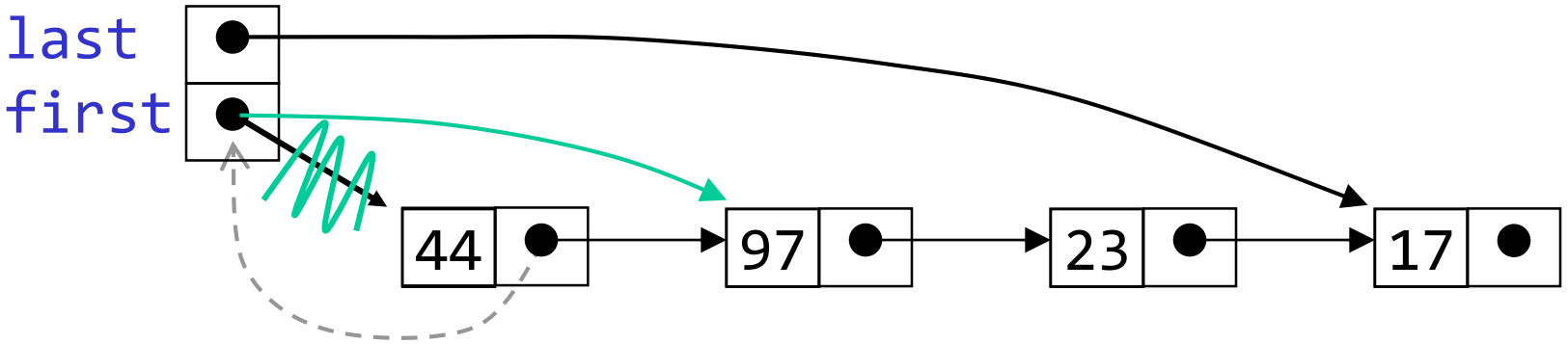
Enqueueing a node



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`

Deques

- A deque is a double-ended queue
- Insertions *and* deletions can occur at *either* end
- Implementation is similar to that for queues
- Deques are not heavily used
- You should know what a deque is, but we won't explore them much further

Lists

- The list is a powerful, yet simple, collection mechanism that provides the programmer with a wide variety of operations.
- However, not all programming languages include a list collection. In these cases, the notion of a list must be implemented by the programmer.
- A **list** is a collection of items where each item holds a relative position with respect to the others.
- Lists can be divided into **ordered** and **unordered** list

The Unordered List ADT

- The structure of an unordered list, as described above, is a collection of items where each item holds a relative position with respect to the others. Some possible unordered list operations are given below.
- **List()** creates a new list that is empty. It needs no parameters and returns an empty list.
- **add(item)** adds a new item to the list. It needs the item and returns nothing. Assume the item is not already in the list.
- **remove(item)** removes the item from the list. It needs the item and modifies the list. Assume the item is present in the list. search(item) searches for the item in the list. It needs the item and returns a Boolean value.
- **is_empty()** tests to see whether the list is empty. It needs no parameters and returns a Boolean value.
- **size()** returns the number of items in the list. It needs no parameters and returns an integer.

The Unordered List ADT

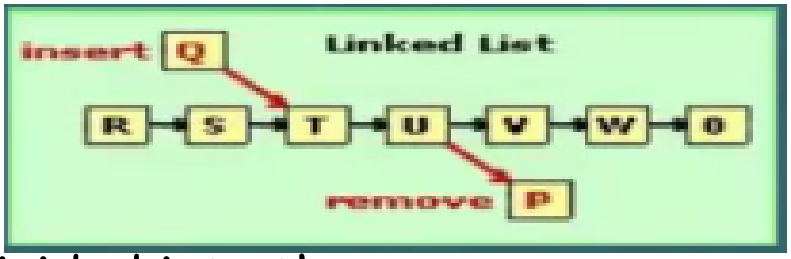
- **append(item)** adds a new item to the end of the list making it the last item in the collection. It needs the item and returns nothing. Assume the item is not already in the list.
- index(item) returns the position of item in the list. It needs the item and returns the index. Assume the item is in the list.
- **insert(pos,item)** adds a new item to the list at position pos. It needs the item and returns nothing. Assume the item is not already in the list and there are enough existing items to have position pos.
- **pop()** removes and returns the last item in the list. It needs nothing and returns an item. Assume the list has at least one item.
- **pop(pos)** removes and returns the item at position pos. It needs the position and returns the item. Assume the item is in the list.

The Ordered List ADT

- The structure of an ordered list is a collection of items where each item holds a relative position that is based upon some underlying characteristic of the item.
- The ordering is typically either ascending or descending and we assume that list items have a meaningful comparison operation that is already defined.
- Many of the ordered list operations are the same as those of the unordered list. Except this operation
  - OrderedList() creates a new ordered list that is empty. It needs no parameters and returns an empty list.

Implementing an Unordered List: Linked Lists

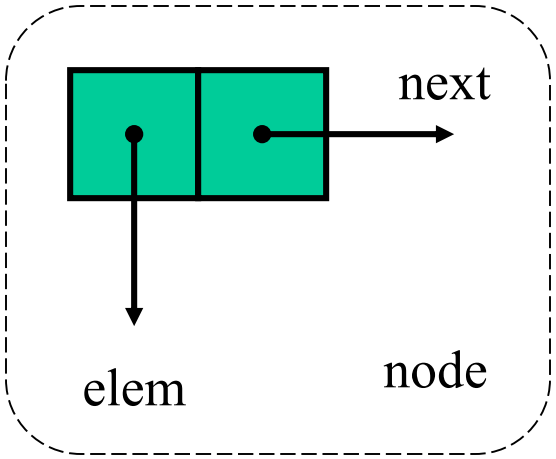
- In order to implement an unordered and ordered list , we will construct what is commonly known as a linked list.
- Recall that we need to be sure that we can maintain the relative positioning of the items.
- **Linked List** -consist of data items called nodes. Each data element contains two field; **information field** and a **link field** connected to another data items.

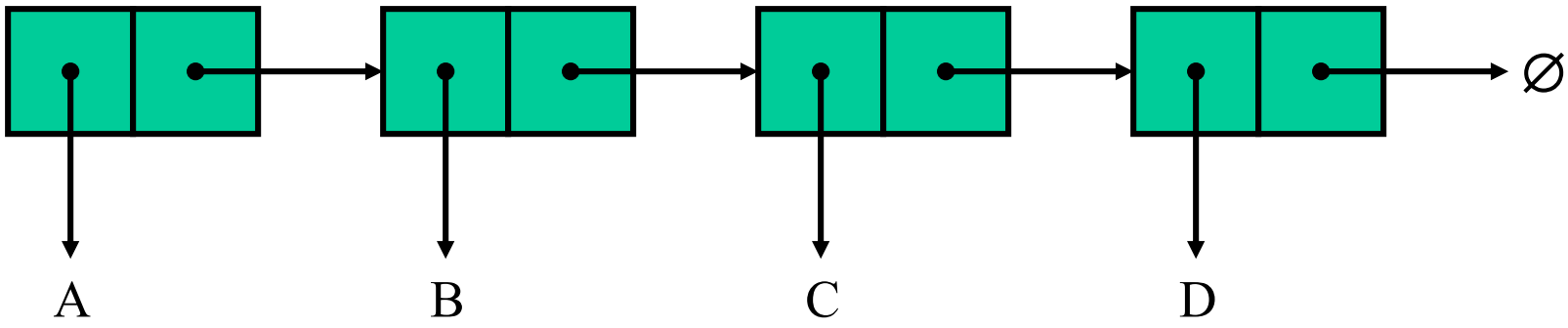


- Linked list is divided into three:
  - Singly linked list
  - Doubly linked list
  - Circularly linked list

Singly Linked Lists

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node





Recursive Node Class

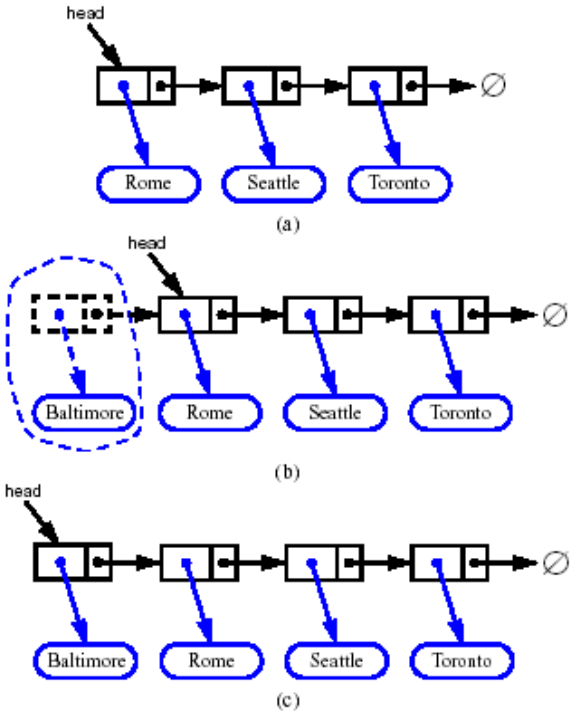
```
public class Node        {
   // Instance variables:
    private  Object element;
    private  Node next;
   /** Creates a node with null references to its element and next node. */
    public  Node()        {
       this(null, null);
  }
   /** Creates a node with the given element and next node. */
    public  Node(Object e, Node n) {
      element = e;
      next = n;
  }
   // Accessor methods:
    public  Object getElement() {
       return  element;
  }
    public  Node getNext() {
       return  next;
  }
   // Modifier methods:
    public void  setElement(Object newElem) {
      element = newElem;
  }
    public void  setNext(Node newNext) {
      next = newNext;
  }
}
```

Singly linked list

```
public class SLinkedList {
   protected Node head; // head node of the list
   /** Default constructor that creates an empty list
   */
   public SLinkedList() {
        head = null;
   }
   // ... update and search methods would go here ...
}
```

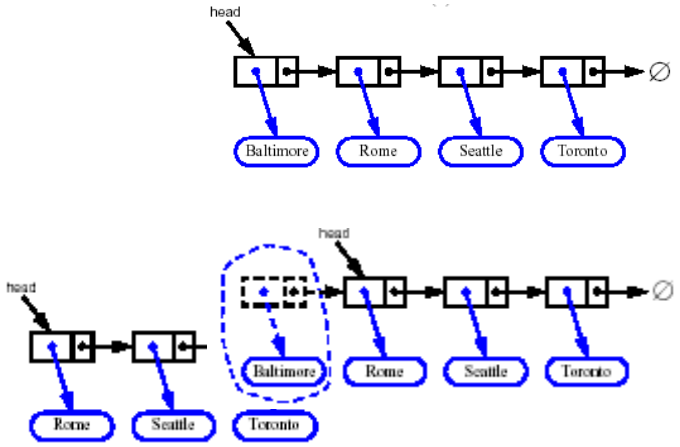
Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Make new node point to old head
4. Update head to point to new node



Removing at the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node

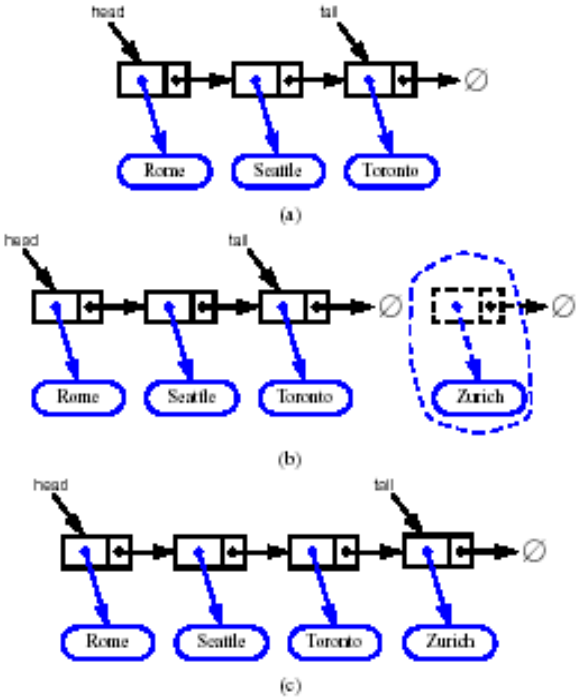


Singly linked list with 'tail' sentinel

```
public class SLinkedListWithTail {
    protected Node head; // head node of the list
    protected Node tail; // tail node of the list
    /** Default constructor that creates an empty list */
    public SLinkedListWithTail() {
        head = null;
        tail = null;
    }
    // ... update and search methods would go here ...
}
```

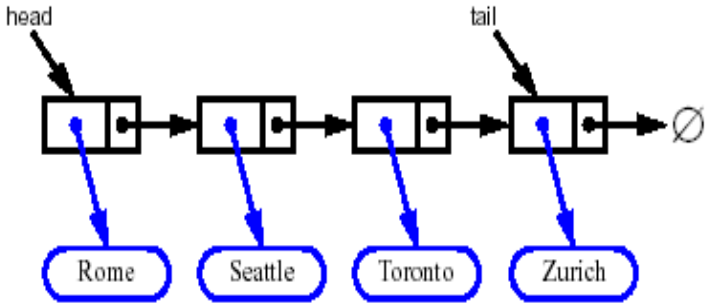
Inserting at the Tail

1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node



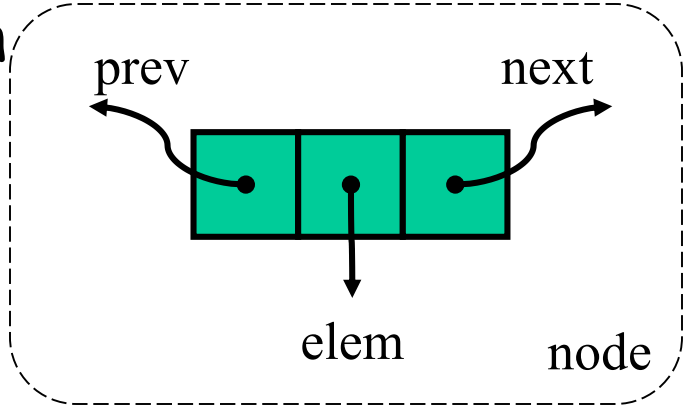
Removing at the Tail

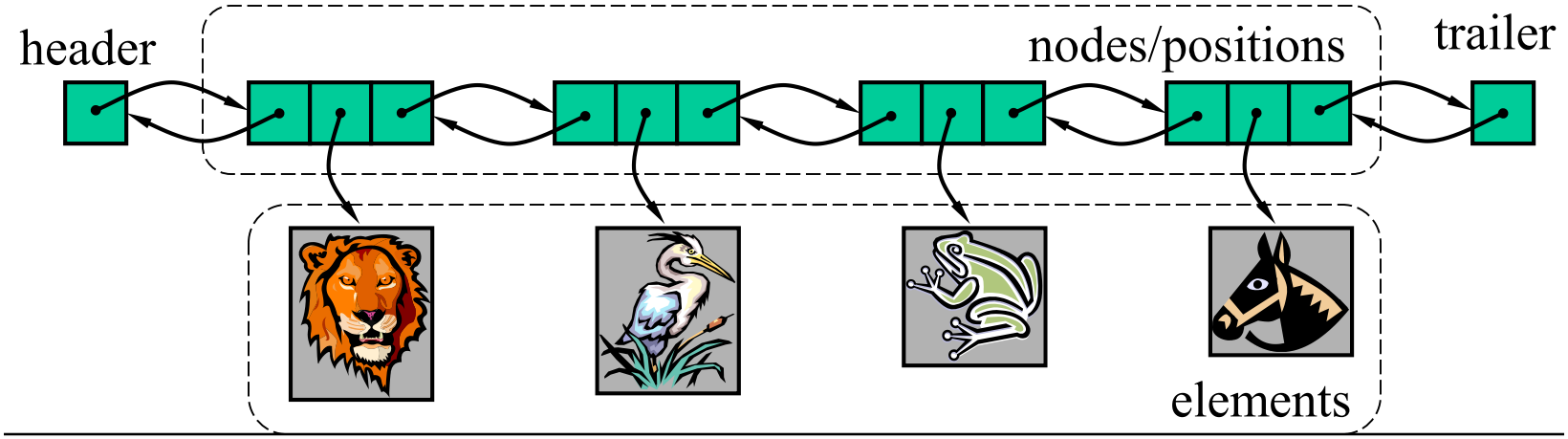
- Removing at the tail of a singly linked list cannot be efficient!
- There is no constant-time way to update the tail to point to the previous node



Doubly Linked List

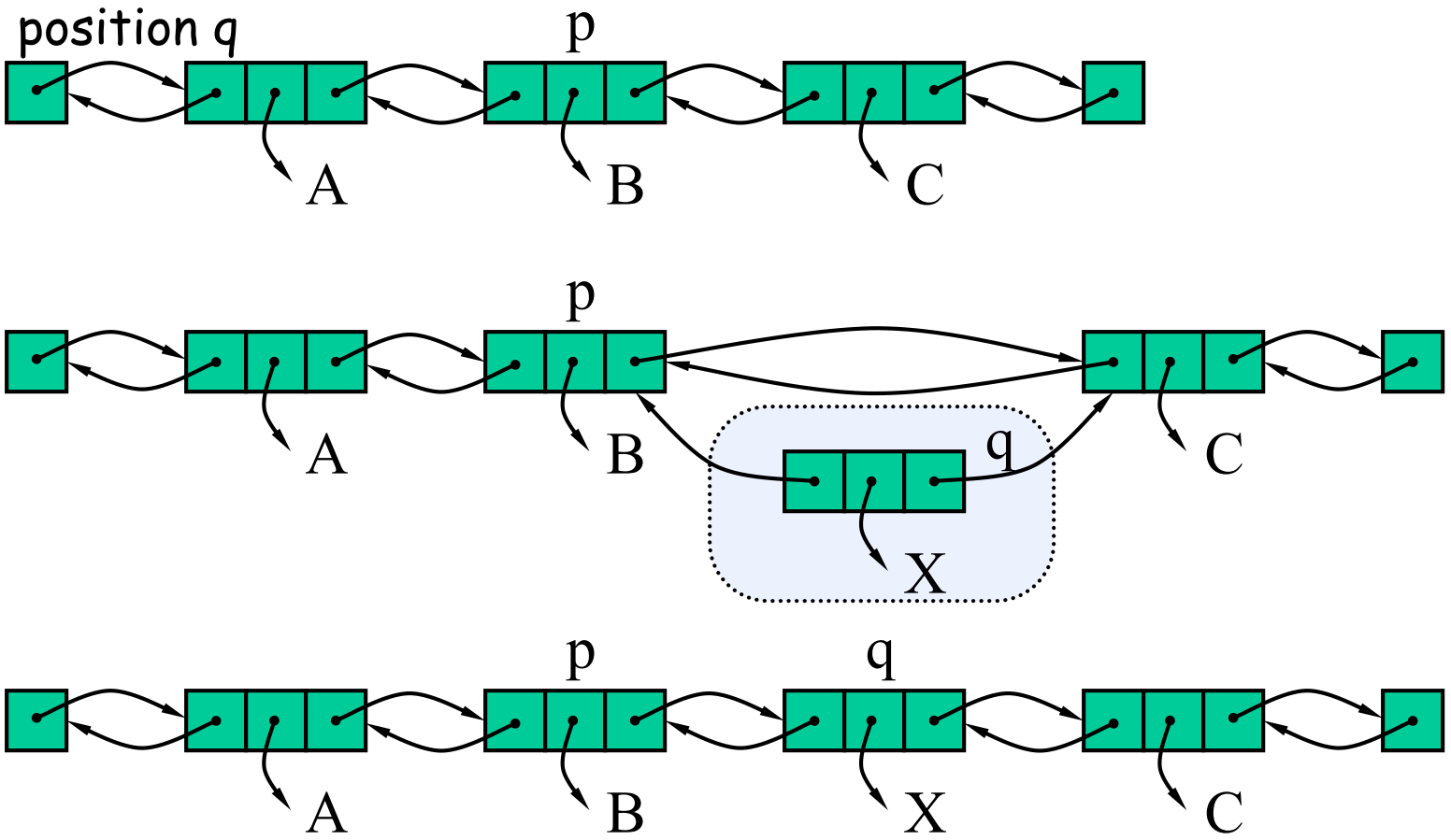
- A doubly linked list is often more convenient!
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





Insertion

- We visualize operation insertAfter(p, X), which returns position q

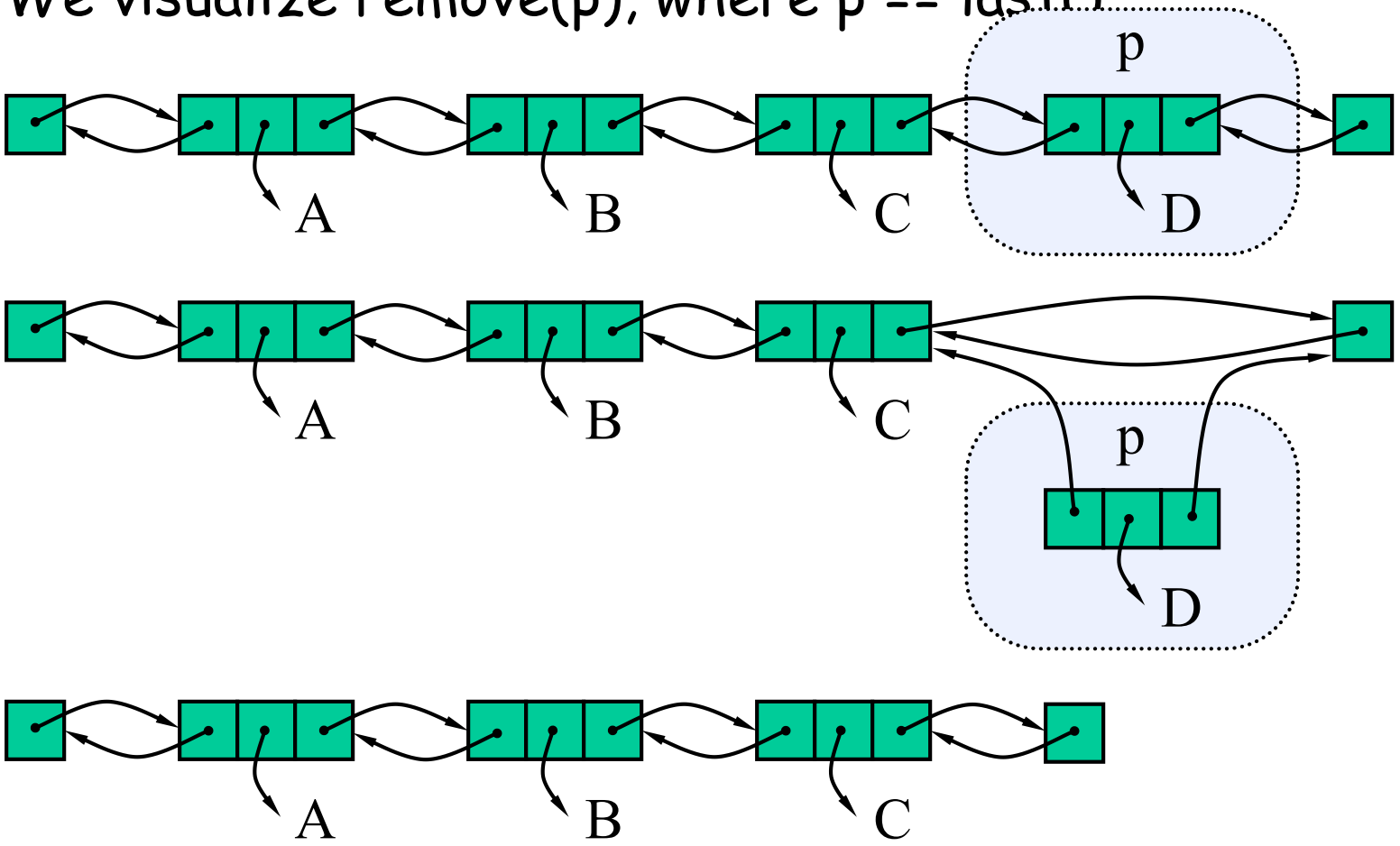


Insertion Algorithm

Algorithm insertAfter(*p*,*e*):
Create a new node *v*
v.setElement(*e*)
v.setPrev(*p*) {link *v* to its predecessor}
v.setNext(*p*.getNext()){link *v* to its successor}
(*p*.getNext()).setPrev(*v*) {link *p*'s old successor to *v*}
p.setNext(*v*) {link *p* to its new successor, *v*}
return *v* {the position for the element *e*}

Deletion

- We visualize remove(p), where p == last()



Deletion Algorithm

```
Algorithm remove(p):
    t = p.element      {a temporary variable to hold the
                        return value}
    (p.getPrev()).setNext(p.getNext())  {linking out p}
    (p.getNext()).setPrev(p.getPrev())
    p.setPrev(null)    {invalidating the position p}
    p.setNext(null)
    return t
```

Worst-cast running time

- In a doubly linked list
  - + insertion at head or tail is in $O(1)$
  - + deletion at either end is on $O(1)$
  - -- element access is still in $O(n)$

The End