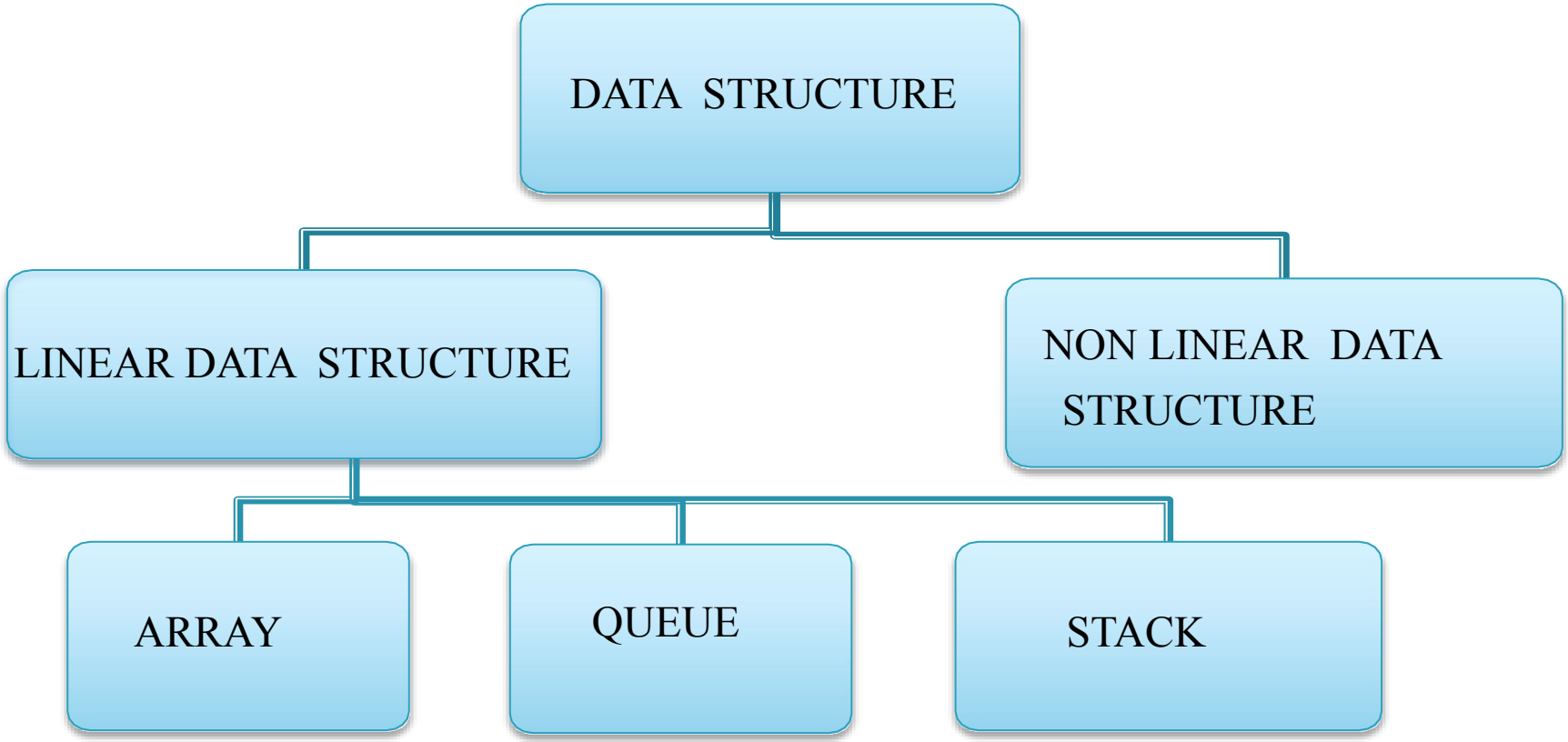
Stacks and Queues

Stacks and Queues



What is linear data structure

- In linear data structure, data is arranged in linear sequence.
- Data items can be traversed in a single run.
- In linear data structure elements are accessed or placed in contiguous (together in sequence) memory locations.

What is stack

- A stack is a last in, first out (LIFO) data structure
  - Items are removed from a stack in the reverse order from the way they were inserted
  - The last items we added (pushed) is the first item that gets pulled (popped) off.
  - A stack is a sequence of items that are accessible at only one end of the sequence.

Examples of stack

A stack of books

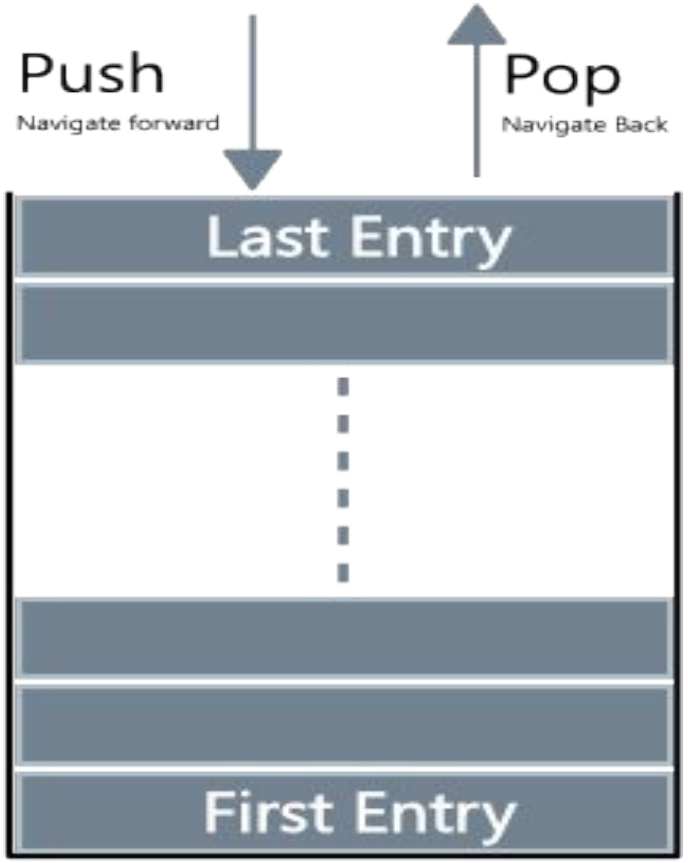
A pile of plates

A stack of dvd/cd

Fig: Realtime examples of Stack

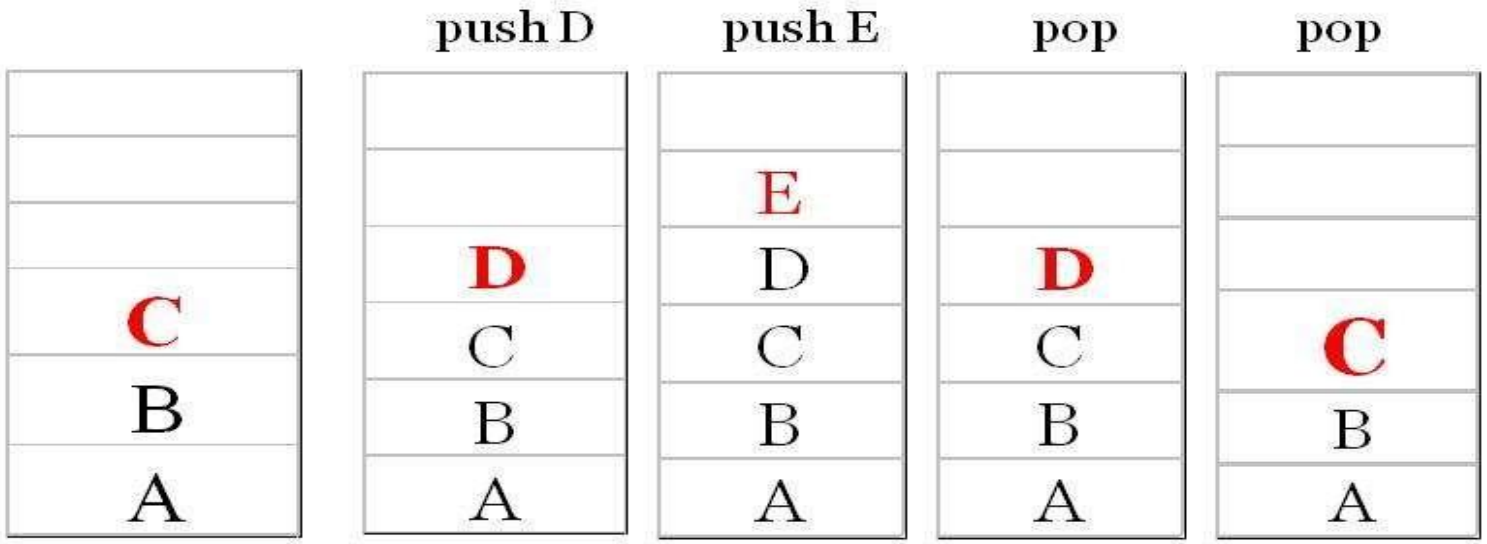
Operations that can be performed on Stack

- Push
- Pop



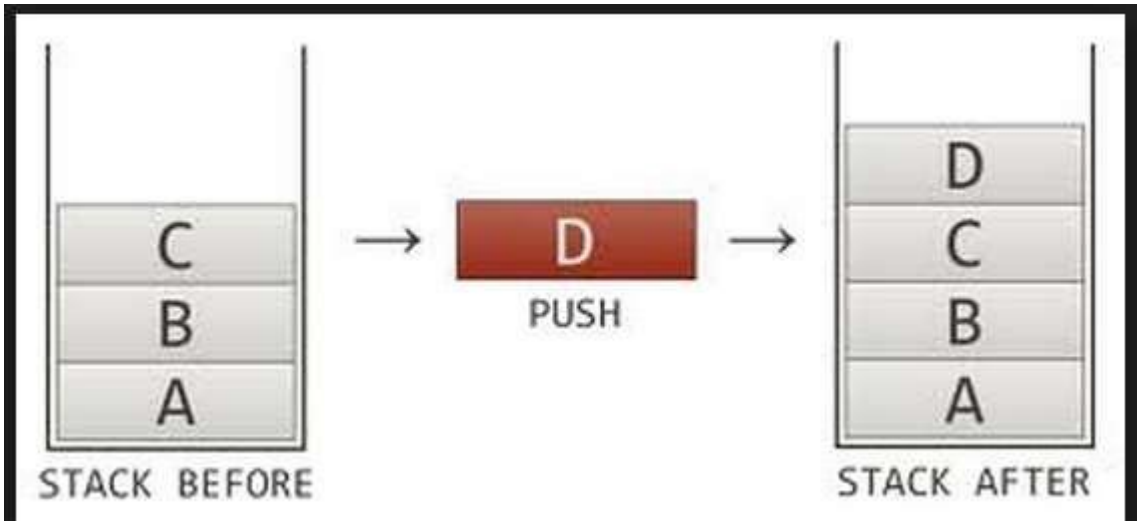
Operations that can be performed on Stack

- Push: It is used to insert items into the stack
- PoP: It is used to delete items from stack
- Top: It represents the current location of data in stack.



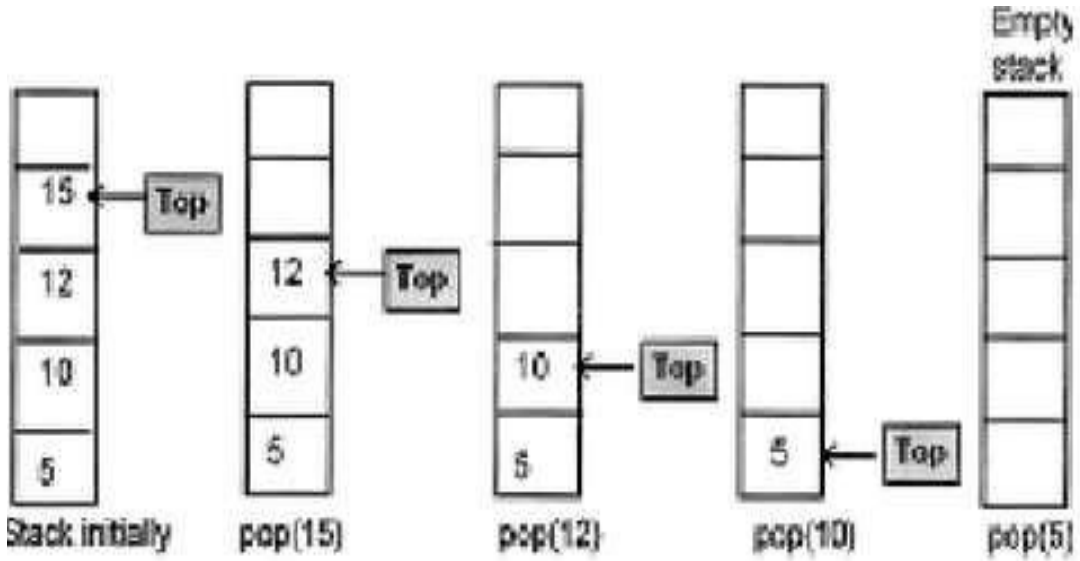
Algorithm of Insertion in Stack: (Push)

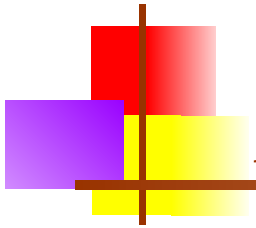
1. Insertion(a,top,item,max)
2. If top=max then
   print 'STACK OVERFLOW'
   exit
   else
3. top=top+1 end
4. a[top]=item
5. Exit



Algorithm of deletion in Stack: (PoP)

1. Deletion(a,top,item)
2. If top=0 then
   print 'STACK UNDERFLOW'
   exit
   else
3. item=a[top]
   end if
4. top=top-1
5. Exit



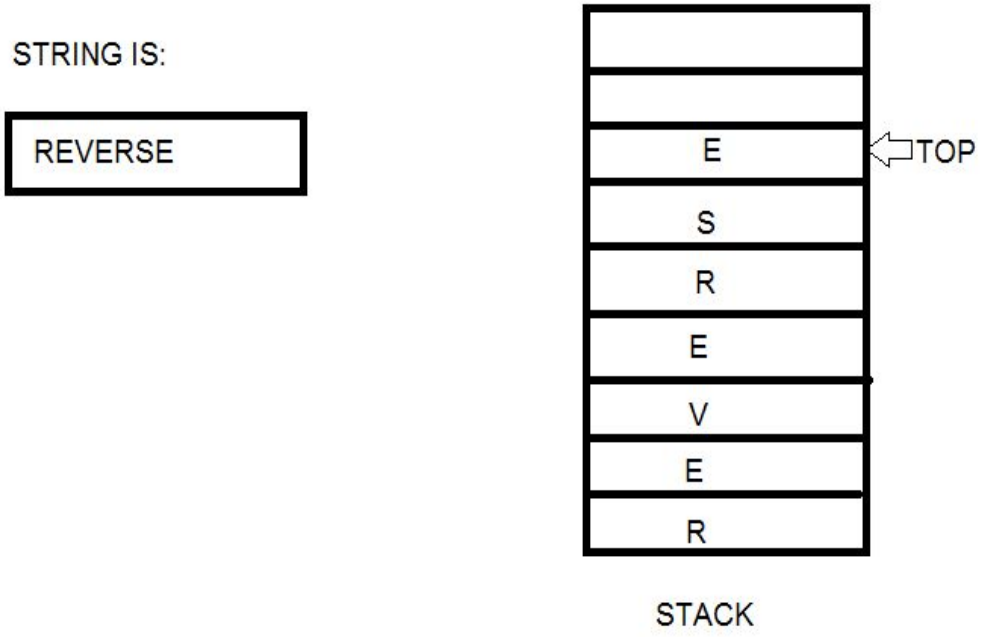


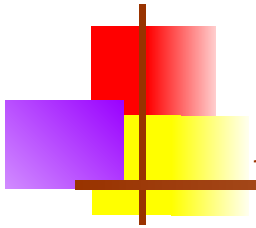
Applications of Stacks are:

- Reversing Strings:
  - A simple application of stack is reversing strings.
  - To reverse a string, the characters of strings are pushed onto the stack one by one as the string is read from left to right.
  - Once all the characters of string are pushed onto stack, they are popped one by one. Since the character last pushed in comes out first, subsequent pop operation results in the reversal of the string.

For example:

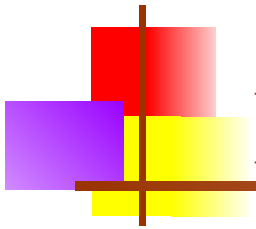
- To reverse the string ‘REVERSE’ the string is read from left to right and its characters are pushed. Like:





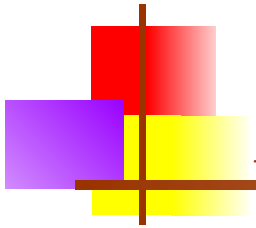
Applications of Stacks are:

- Checking the validity of an expression containing nested parenthesis:
  - Stacks are also used to check whether a given arithmetic expressions containing nested parenthesis is properly parenthesized.
  - The program for checking the validity of an expression verifies that for each left parenthesis braces or bracket, there is a corresponding closing symbol and symbols are appropriately nested.



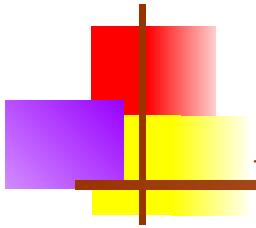
For example:

VALID INPUTS	INVALID INPUTS
{} ({[]}) {[]()} [{({}[](({ })}]	{(} ([(())]) {}[]) [{)}(]}]



Applications of Stacks are:

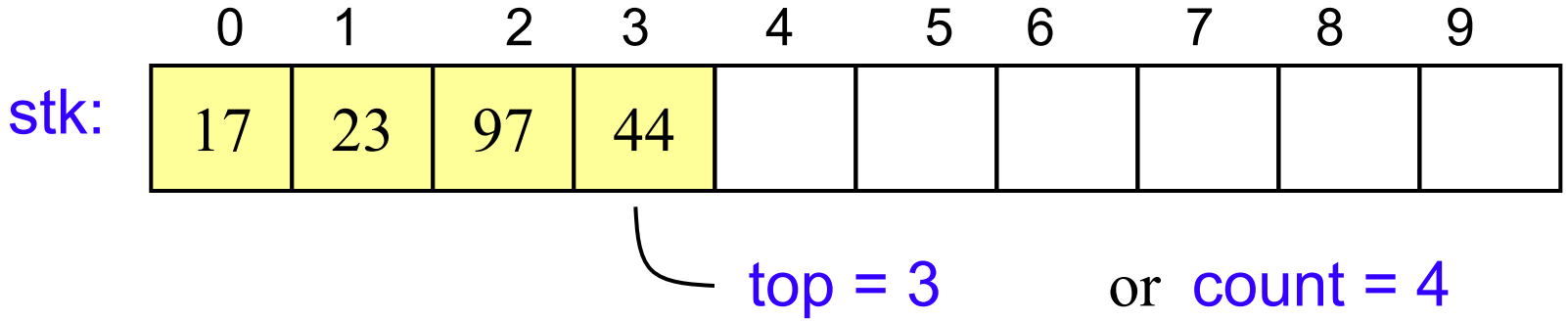
- Evaluating arithmetic expressions:
  - INFIX notation:
    - The general way of writing arithmetic expressions is known as infix notation. E.g, (a+b)
  - PREFIX notation:
    - e.g, +AB
  - POSTFIX notation:
    - e.g. AB+



Array implementation of stacks

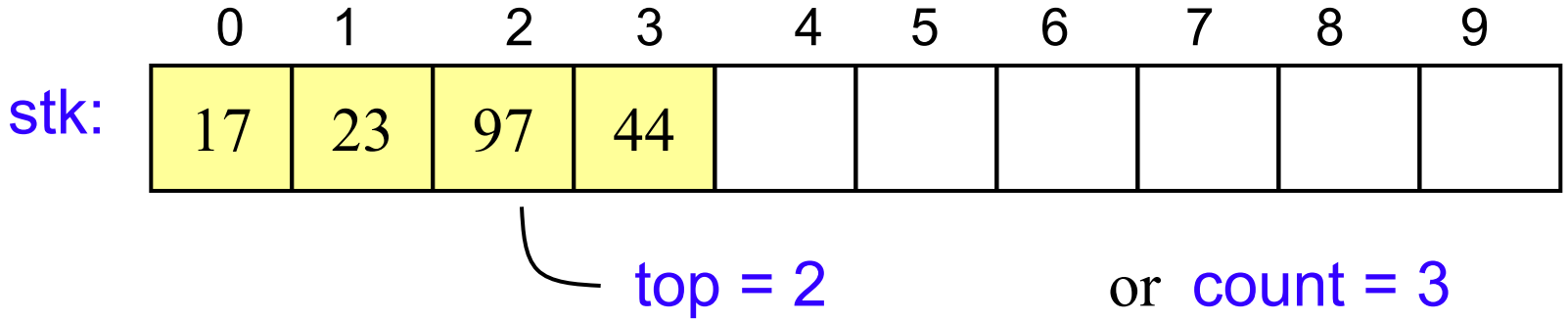
- To implement a stack, items are inserted and removed at the same end (called the top)
- Efficient array implementation requires that the top of the stack be towards the center of the array, not fixed at one end
- To use an array to implement a stack, you need both the array itself and an integer
 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

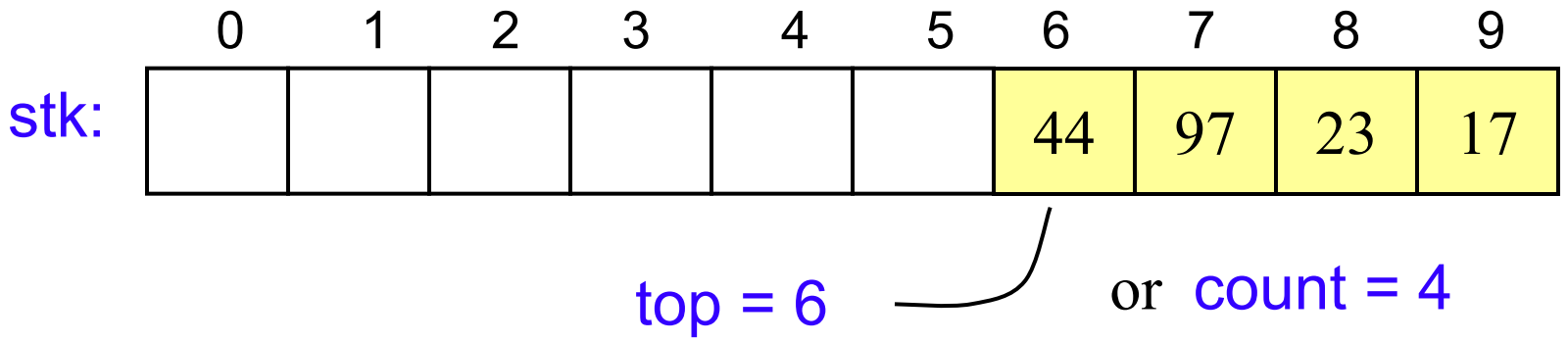
After popping



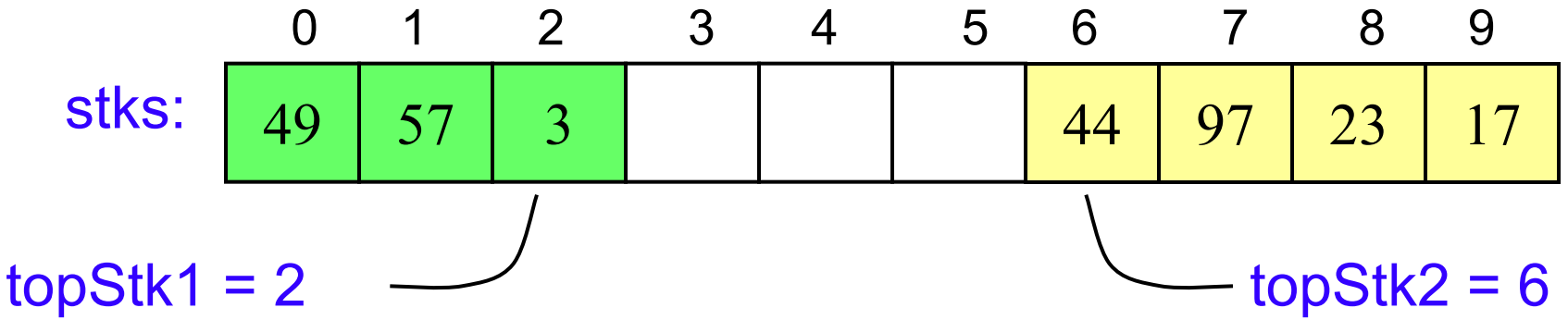
- When you pop an element, do you just leave the "deleted" element sitting in the array?
- The surprising answer is, *"it depends"*
  - If this is an array of primitives, *or* if you are programming in C or C++, *then* doing anything more is just a waste of time
  - If you are programming in Java, and 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

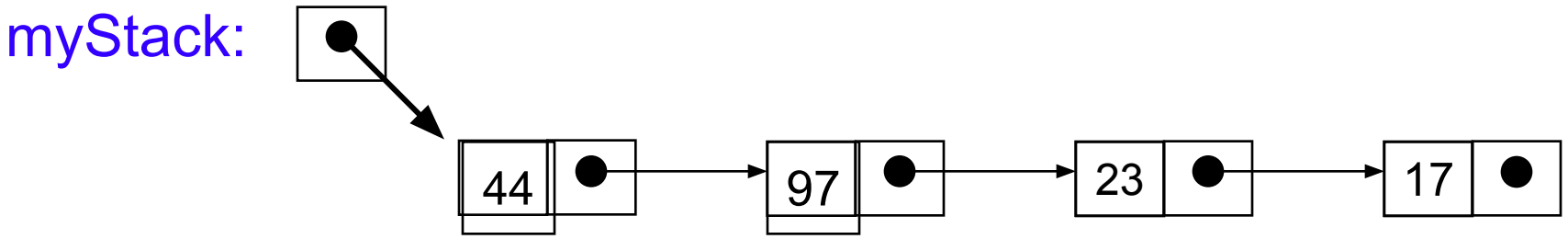


- Sometimes this is done to allow two stacks to share the *same storage area*

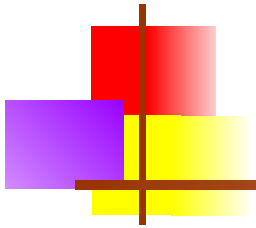


Linked-list implementation of stacks

- 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

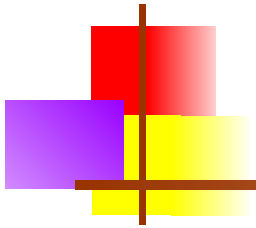


- Pushing is inserting an element at the front of the list
- Popping is removing an element from the front of the list



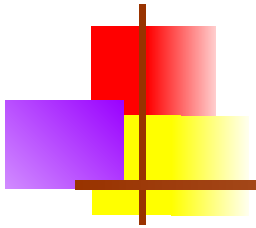
Linked-list implementation details

- 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



Queues

- Queue is an ADT data structure similar to stack except that the first item to be inserted is the first one to be removed.
- This mechanism is called First-in-First-Out (FIFO).
- Placing an item in a queue is called “insertion or enqueue”, which is done at the end of the queue called “rear”.
- Removing an item from the queue is called “deletion or dequeue”, which is done at the other end of the queue called “front”.
- Some of the applications are :printer queue, keystroke queue, etc.

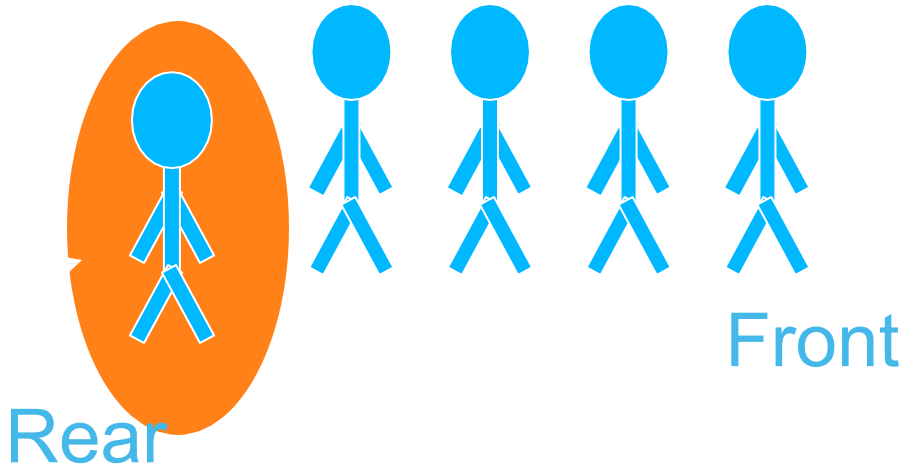


Operations on a Queue

- To insert an element in queue
- Delete an element from queue

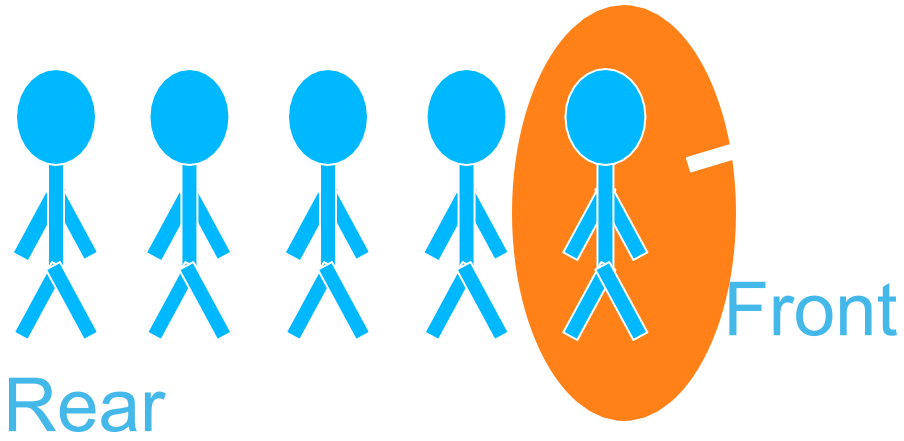
The Queue Operation

- Placing an item in a queue is called “insertion or enqueue”, which is done at the end of the queue called “rear”.



The Queue Operation

- Removing an item from a queue is called “deletion or dequeue”, which is done at the other end of the queue called “front”.



Algorithm Qinsert (Item)

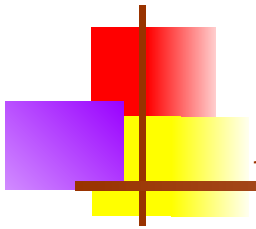
*1.*If(rear = maxsize-1)

print (“queue overflow”) and return

*2.*Else

rear = rear + 1 Queue

[rear] = item



Algorithm Qdelete ()

1. If (front =rear)

   print “queue empty” and return

2. Else

   Front = front + 1 item

   = queue [front];

   Return item

States of the Queue

1. Queue is empty

   FRONT=REAR

1. Queue is full

   REAR=N

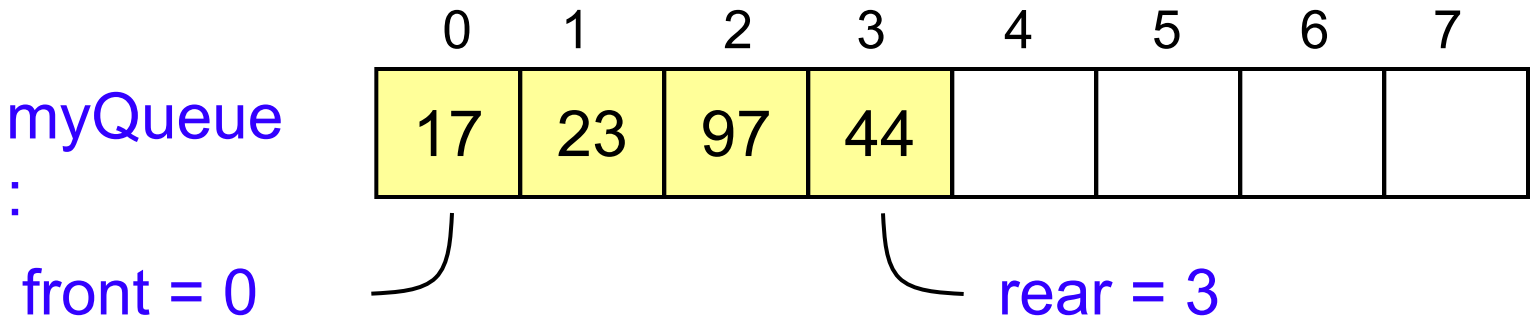
1. Queue contains element >=1

   FRONT<REAR

   NO. OF ELEMENT=REAR-FRONT+1

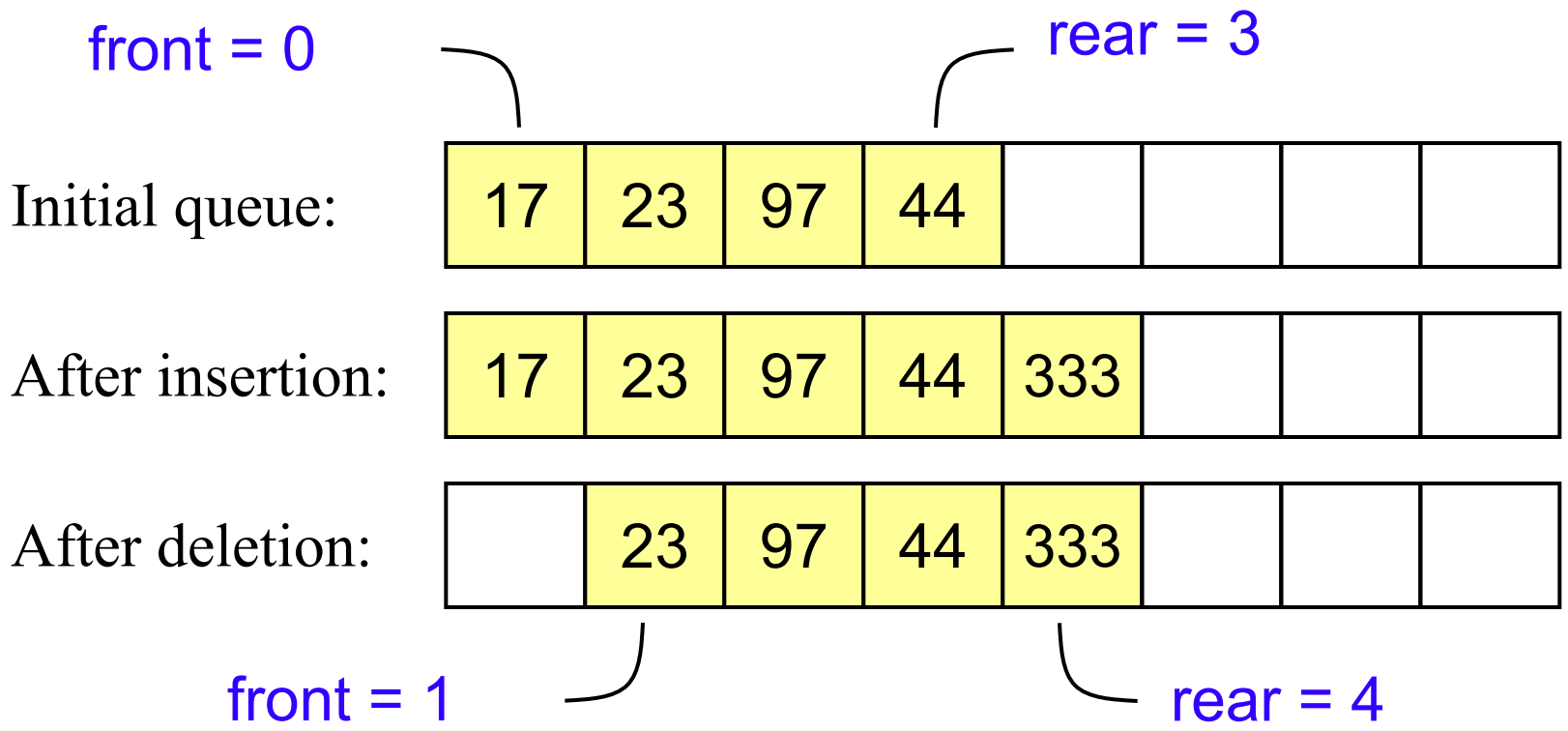
Array implementation of queues

- 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

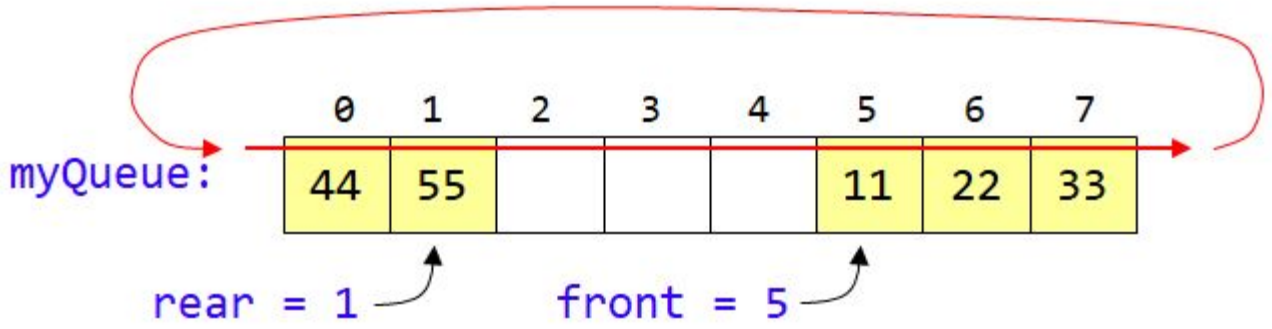
Array implementation of queues



- 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

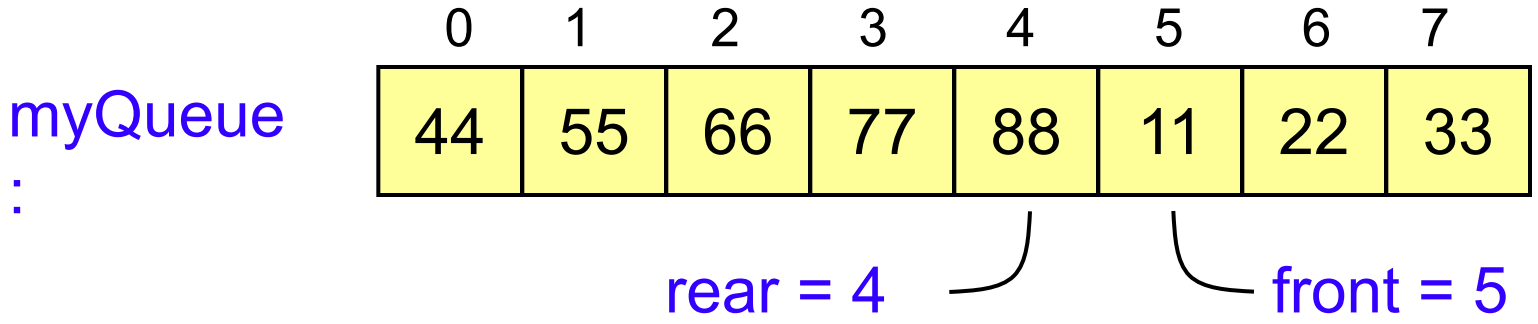
- We can treat the array holding the queue elements as circular (joined at the ends)



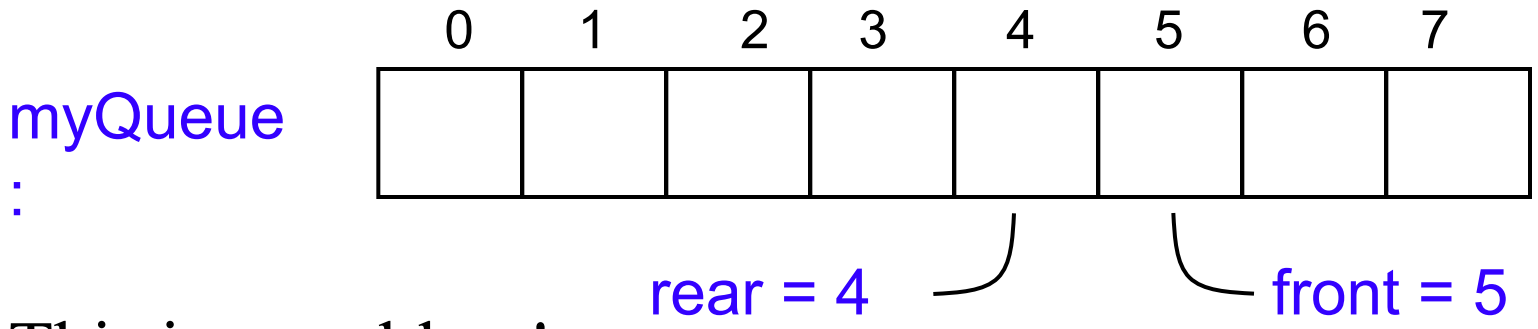
- Elements were added to this queue in the order 11, 22, 33, 44, 55, and will be removed in the same order
- Use: front = (front + 1) % myQueue.length;
  and: rear = (rear + 1) % myQueue.length;

Full and empty queues

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



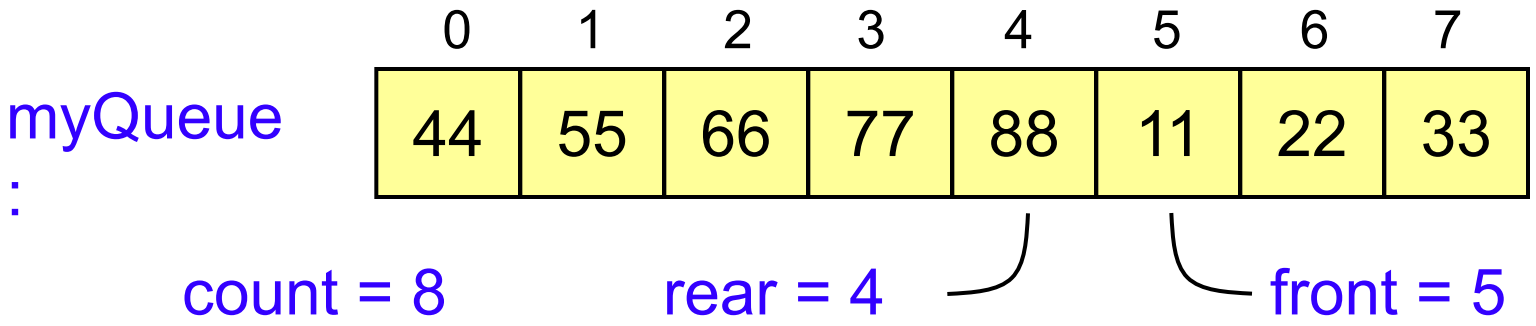
- If we were then to remove all eight elements, making the queue completely empty, it would look like this:



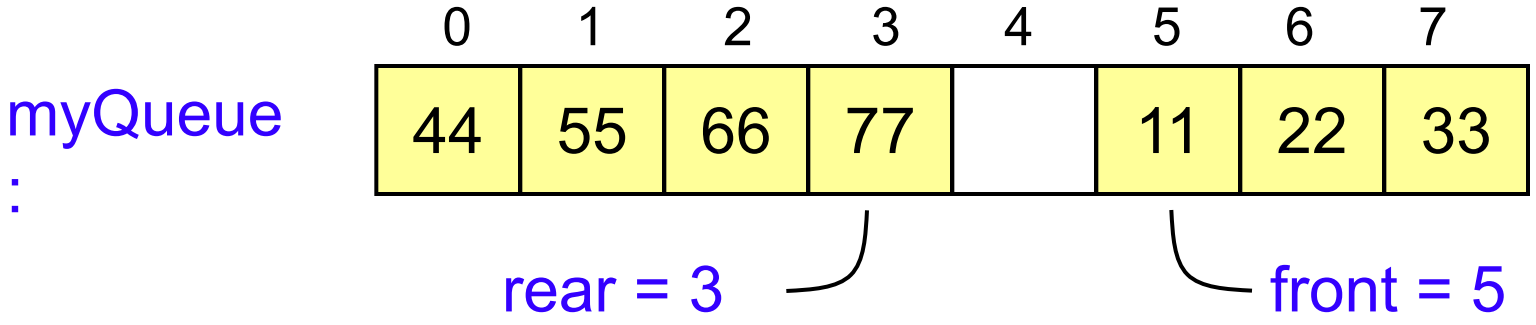
This is a problem!

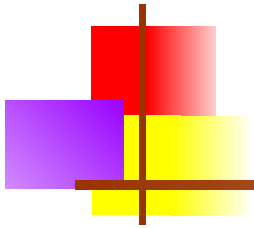
Full and empty queues: solutions

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



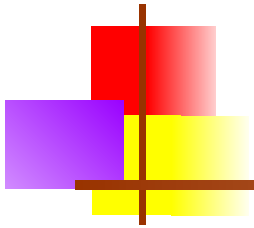
- **Solution #2:** (Slightly more efficient) Keep a gap between elements: consider the queue full when it has n-1 elements





Linked-list implementation of queues

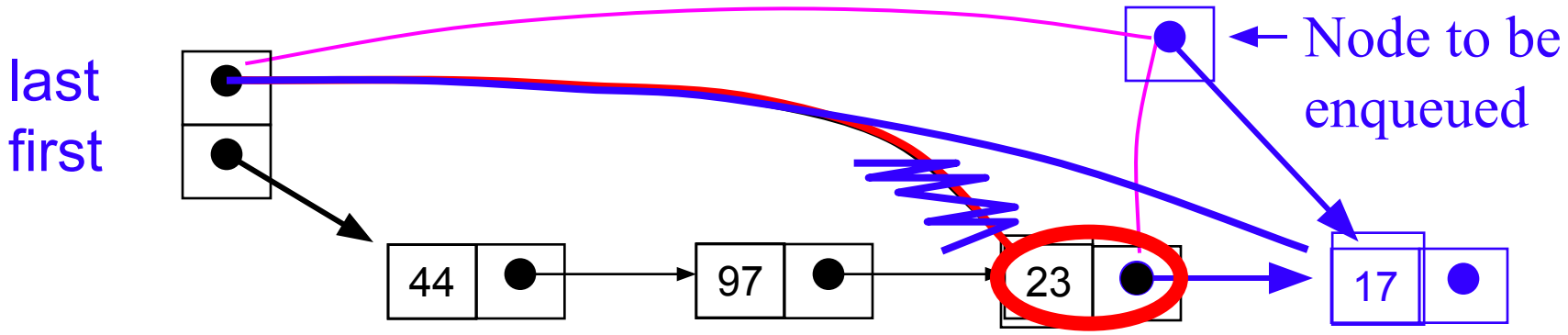
- In a queue, insertions occur at one end, deletions at the other end
- Operations at the front of a singly-linked list (SLL) are O(1), but at the other end they are O(n)
  - Because you have to find the last element each time
- BUT: there is a simple way to use a singly-linked list to implement both 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



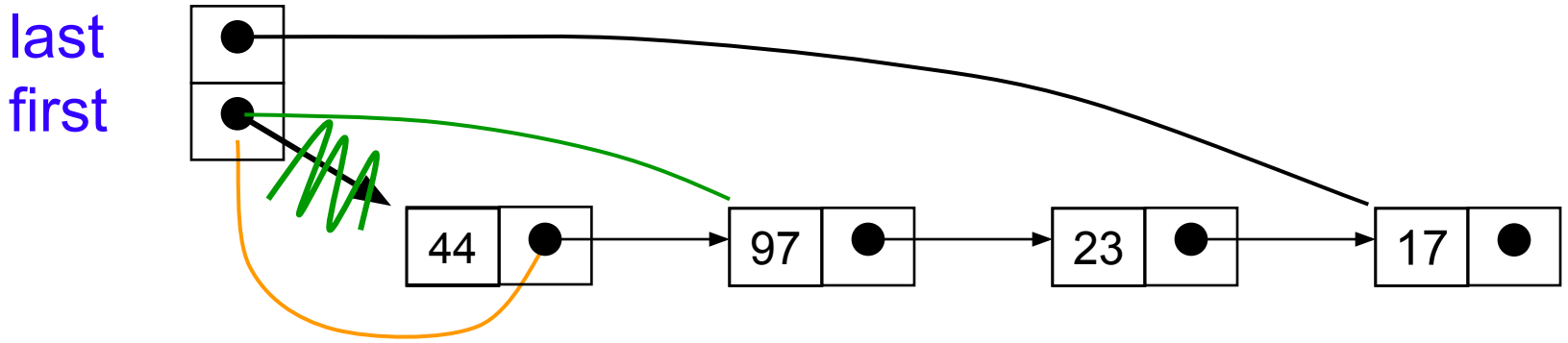
To enqueue (add) a node:

Find the current last node

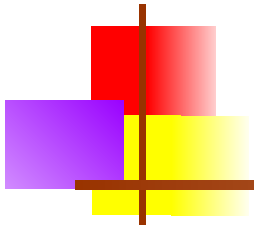
Change it to point to the new last node

Change the last pointer in the list header

Dequeueing a node

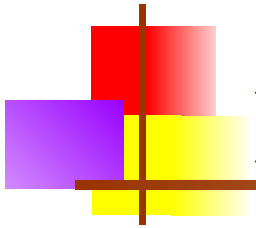


- To dequeue (remove) a node:
  - Copy the pointer from the first node into the header



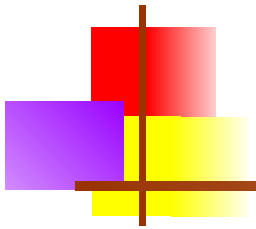
Queue implementation details

- 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



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



The End