CS 10: Problem solving via Object Oriented Programming

Winter 2017

Tim Pierson

260 (255) Sudikoff

Day 12 – Keeping order

Agenda

1. Stacks
2. Queues

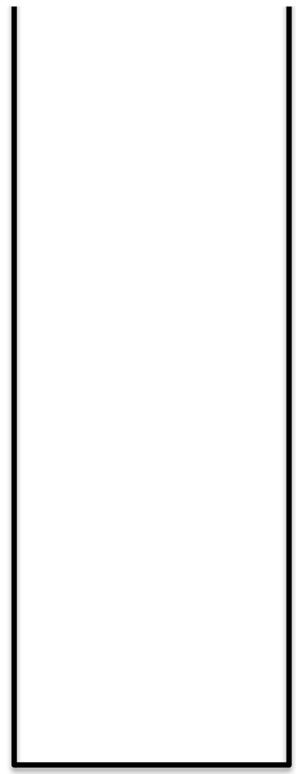
Stacks are a Last In, First Out (LIFO) data structure

Stack overview

- Think of stack of dinner plates (or Pez dispenser)
- Add item to the top, others move down
- To remove, take top item (last one inserted)
- Commonly used in CS – function calls, paren matching, …
- **Operations**
  - `push` – add item to top of stack
  - `pop` – remove top item and return it
  - `peek` – return top item, but don't remove it
  - `isEmpty` – true if stack empty

Stack in action

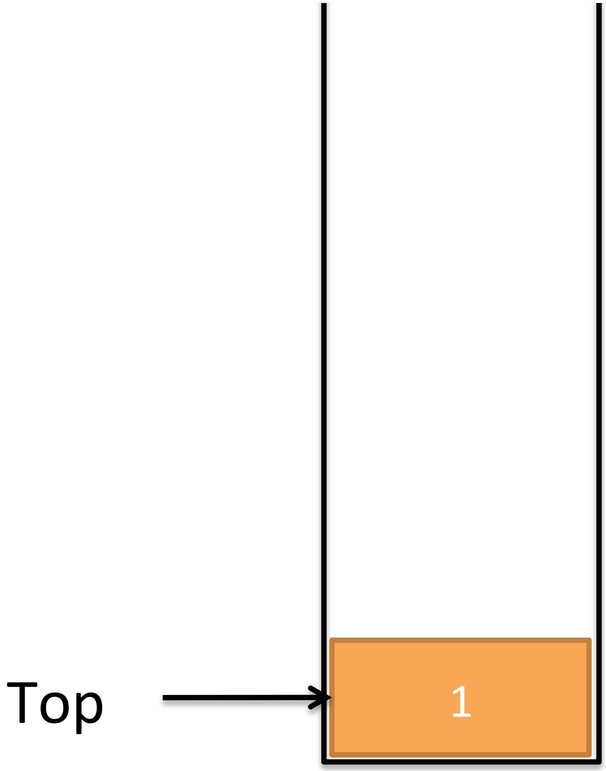
Initially empty



Stack

Stack in action

Push 1



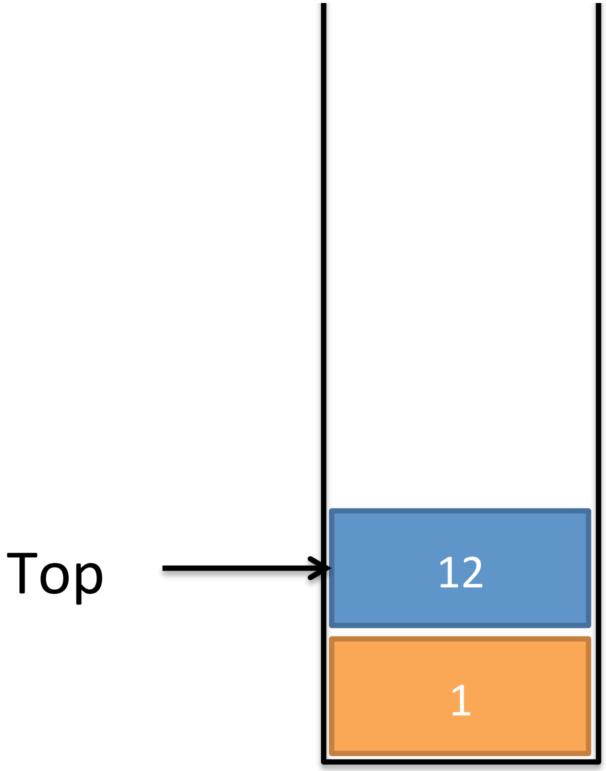
Stack

Operations

Push 1

Stack in action

Push 12



Stack

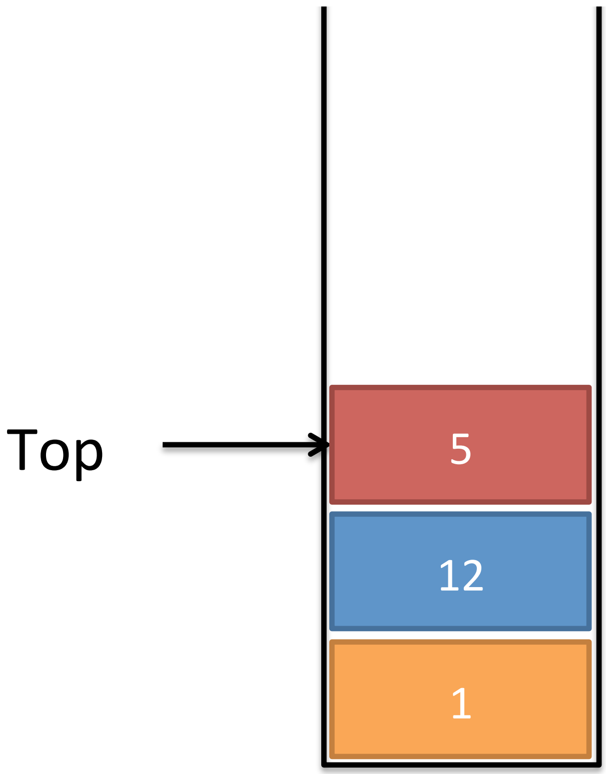
Operations

Push 1

Push 12

Stack in action

Push 5

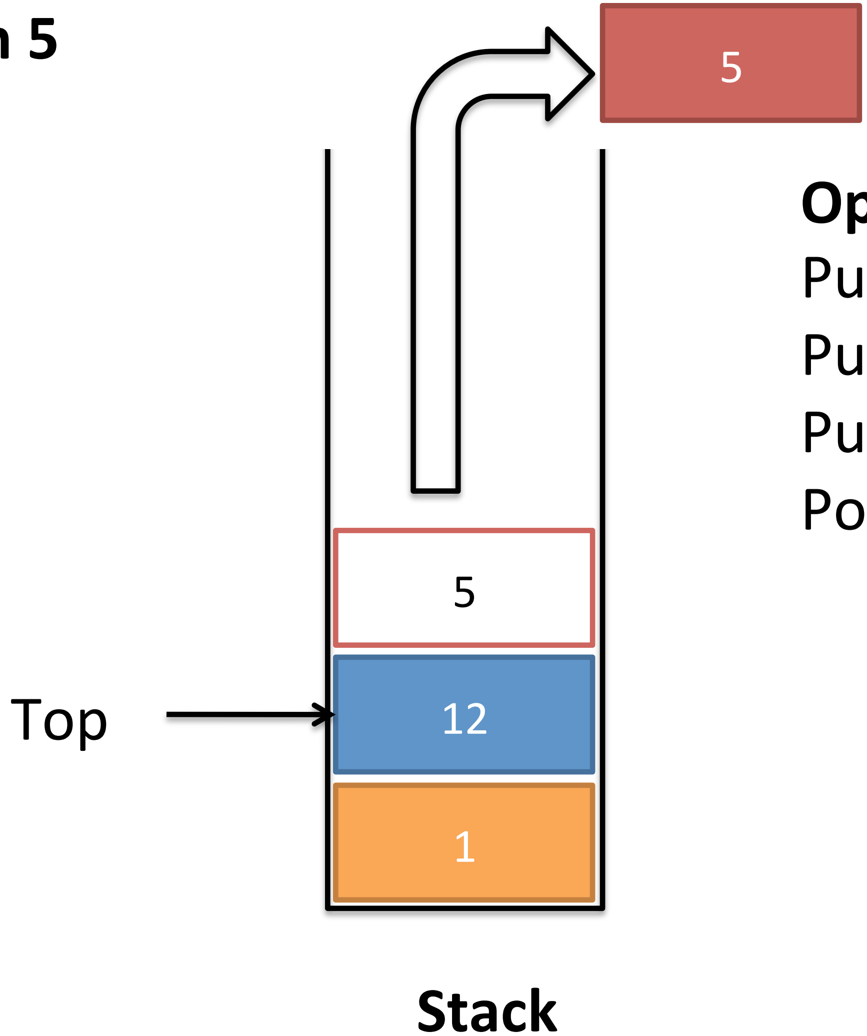


Stack

Operations
Push 1
Push 12
Push 5

Stack in action

Pop – return 5



Operations

Push 1

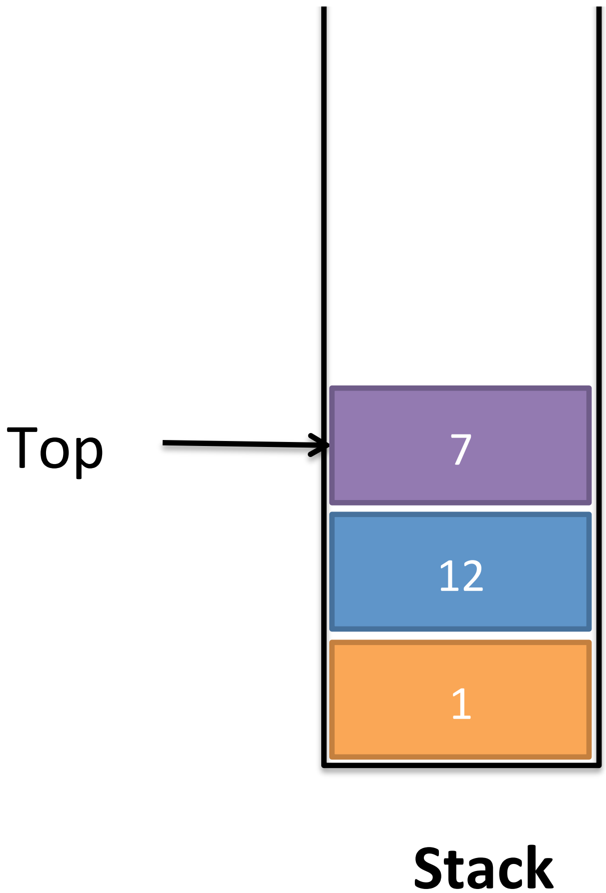
Push 12

Push 5

Pop – return 5

Stack in action

Push 7



Operations

Push 1

Push 12

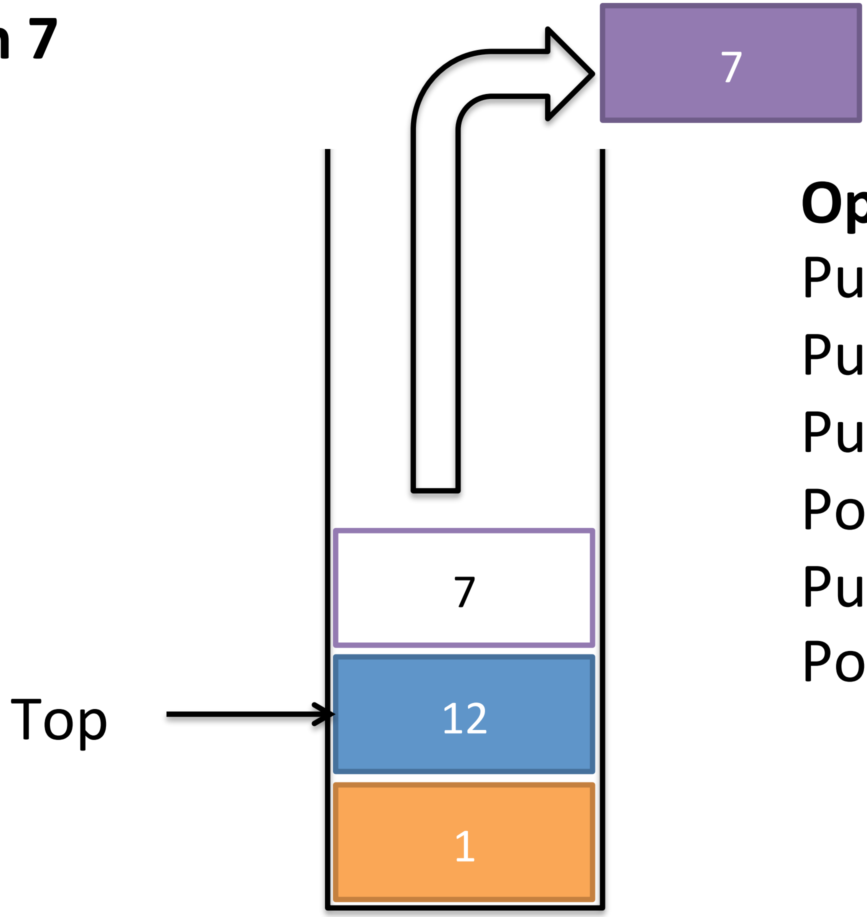
Push 5

Pop – return 5

Push 7

Stack in action

Pop – return 7

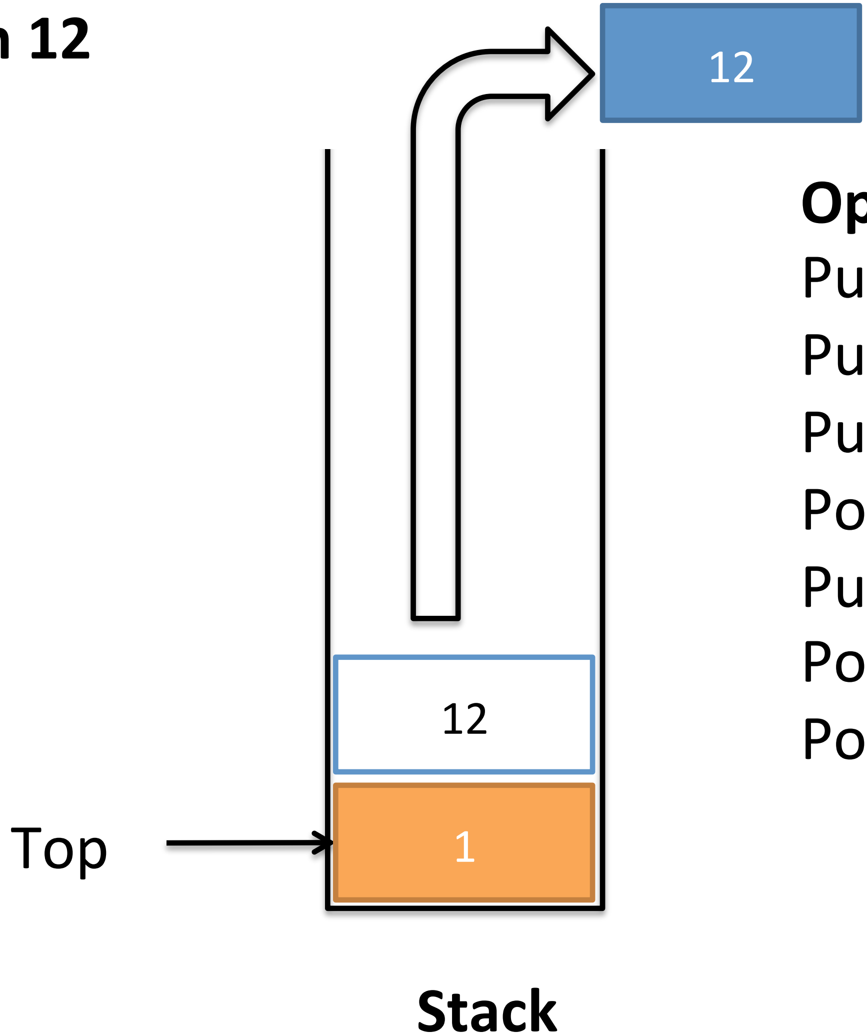


Stack

Operations
Push 1
Push 12
Push 5
Pop – return 5
Push 7
Pop – return 7

Stack in action

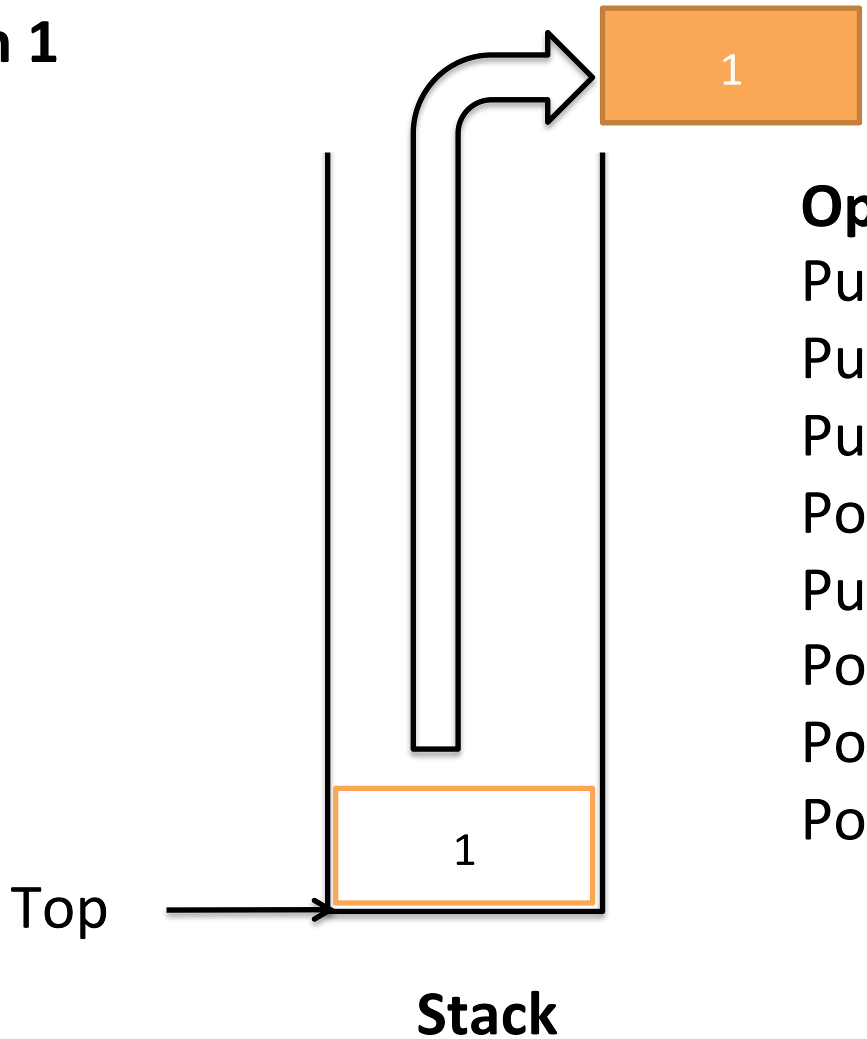
Pop – return 12



Operations
Push 1
Push 12
Push 5
Pop – return 5
Push 7
Pop – return 7
Pop – return 12

Stack in action

Pop – return 1

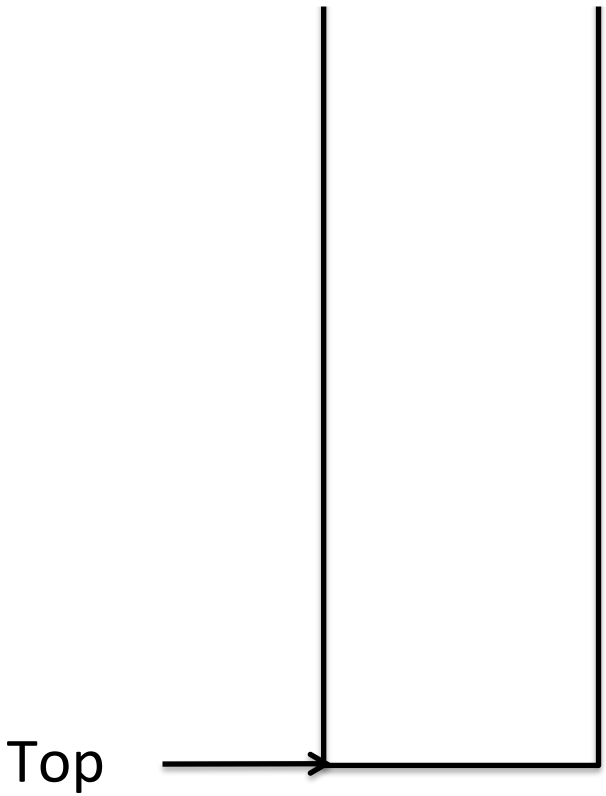


Operations

Push 1
Push 12
Push 5
Pop – return 5
Push 7
Pop – return 7
Pop – return 12
Pop – return 1

Stack in action

Pop – throw exception



Stack

Operations
Push 1
Push 12
Push 5
Pop – return 5
Push 7
Pop – return 7
Pop – return 12
Pop – return 1
Pop – throw exception

SimpleStack.java is an interface defining Stack operations

SimpleStack.java

- Interface that mandates Stack operations

We can use the simple stack to easily match parens in a string

MatchParens.java

- Define what constitutes an open and a close paren
- *check(String s)*
  - Creates new stack of integers called *opened*
  - Loop over all characters in *s*
    - If *s* in open parens, push *idx* (paren type) on to stack
    - If *s* in close parens,
      - If *opened* empty, then string is invalid because we have a close with an open
      - Else if *pop()* ≠*idx* then didn't match
  - If stack empty after all chars checked, then all parens match

We can implement a Stack using an array

Stack array implementation

- Create array and set *top = -1*
- To push(T elmt), add 1 to *top* and *stack[top] = elmt*
- To peek() return *stack[top],* if *top >=0*
- To pop(), do *peek()* and set *top -= 1*
- Implementation is O(1) for all operations, never need to move items
- Might run out of space using an array
- Can use ArrayList and not run out of space
    - To *push()*, add on to end
    - To *pop()*, remove from end

An ArrayList implementation makes sure the Stack does not run out of space

ArrayListStack.java

- Implements SimpleStack
- ArrayList keeps track of top for us with size!
- *main()*
  - Set breakpoints and run
- Adds are O(1), but might need to resize the array, so amortized O(1), some times takes longer

A Singly Linked List also works well for a Stack, using top as head of list

SLLStack.java

- Implements SimpleStack
- Top of stack is head
- Implementation is straight forward
  - *push()* adds to front
  - *pop()* removes from front
- All operations O(1)
- Never have empty space list an array implementation

Agenda

1. Stacks
2. Queues

Queues are a First In, First Out (FIFO) data structure

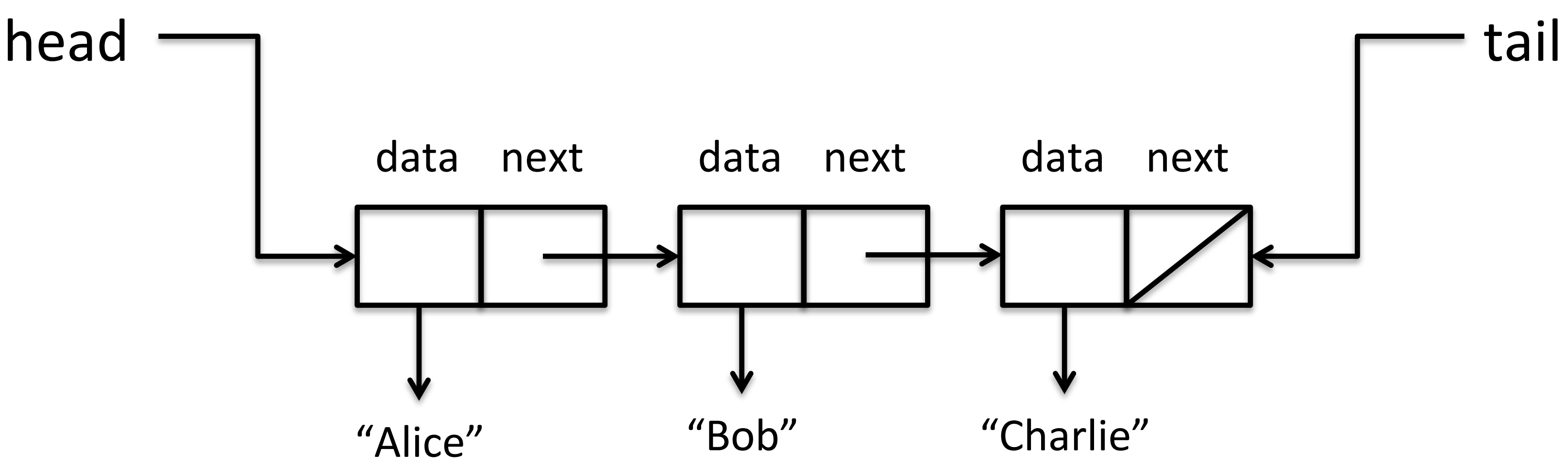
Queue overview

- Think of line at a store, join in back, leave from front
- Used in simulations, queuing print jobs, running jobs, could have used it for PS-1 to visit neighbor pixels
- **Operations**
  - `enqueue` – add item at rear of queue
  - `dequeque` – remove and return first item in queue
  - `front` – return first item, but don't remove it
  - `isEmpty` – true if queue empty
- Java uses different names (first ones throw errors, seconds ones return false if unable to complete)
  - `enqueue` == `add`*()* and `offer`*()*
  - `dequeue` == `remove`*() and* `poll`*()*
  - `front` == `element`*()* and `peek`*()*

Queues can be implemented with Singly Linked List using head and tail pointers

Queue implementation

- Easy to remove from head
- Use tail to add to back of queue
    - Set new element next to null
    - Set previous tail next to new element
- All operations O(1)



All operations on a Singly Linked List implementation are O(1)

SLLQueue.java

- Implements SimpleQueue, tracks *head* and *tail*
- *enqueue(T item)*
  - If *isEmpty()*, set *head* and *tail* to *item*
  - Else *tail.next = new Element, tail=tail.next*
- *dequeue()*
  - Check for *isEmpty()*
  - Save *head.data* in temporary *item*
  - *head = head.next*
  - return *item*
- *front()*
  - Return *head.data*
- Run

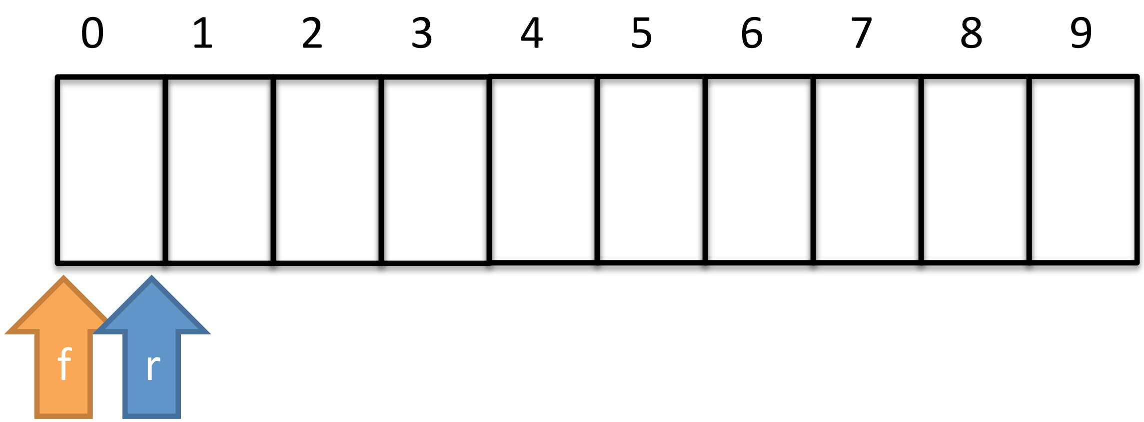
Arrays are seemingly unpromising as a Queue data structure, but it can work well

Array implementation

- Could `enqueue` at back, `dequeue` from front
  - `enqueue` is fast, just add item to end O(1)
  - `dequeue` must move all elements left one space O(n)
- Could `enqueue` at front and `dequeue` from back
  - `enqueue` must move all elements right one space O(n)
  - `dequeue` is fast, just take last item O(1)
- Could track `front` and `rear` indexes (circular array)
  - `enqueue` at `r`, then increment `r`
  - `dequeue` at `f`, then increment `f`
  - If `f` or `r > m-1`, wrap around to empty spaces at front
  - Full or empty when `f==r` (use empty space or track count of items)
  - `enqueue` and `dequeue` O(1)

Array implementing a Queue using index for front and rear

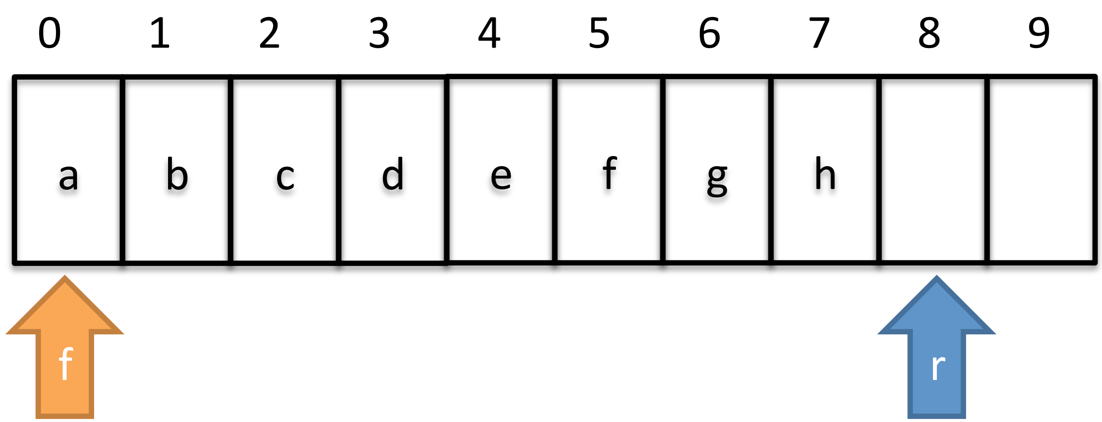
Array implementing Queue



Empty `f==r, size = 0`

Array implementing a Queue using index for front and rear

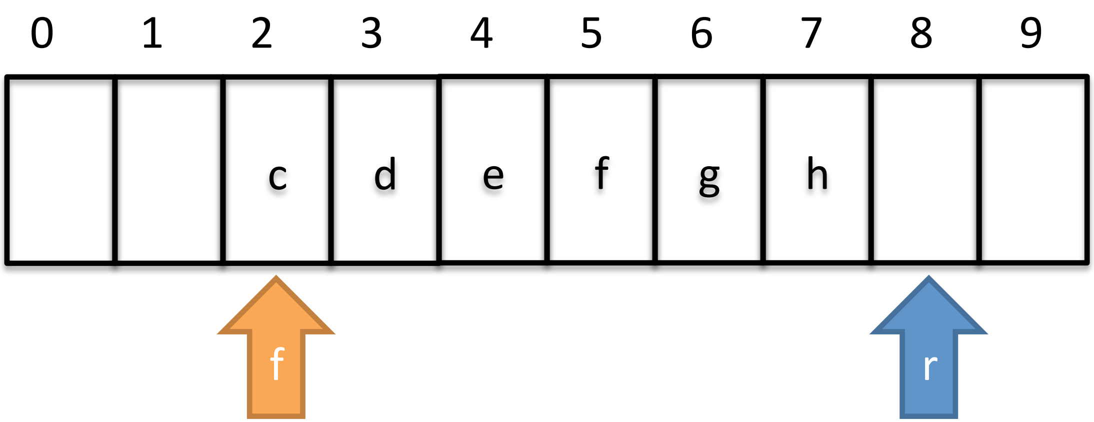
Array implementing Queue




```
enqueue(a)
…
enqueue(h)
```

Array implementing a Queue using index for front and rear

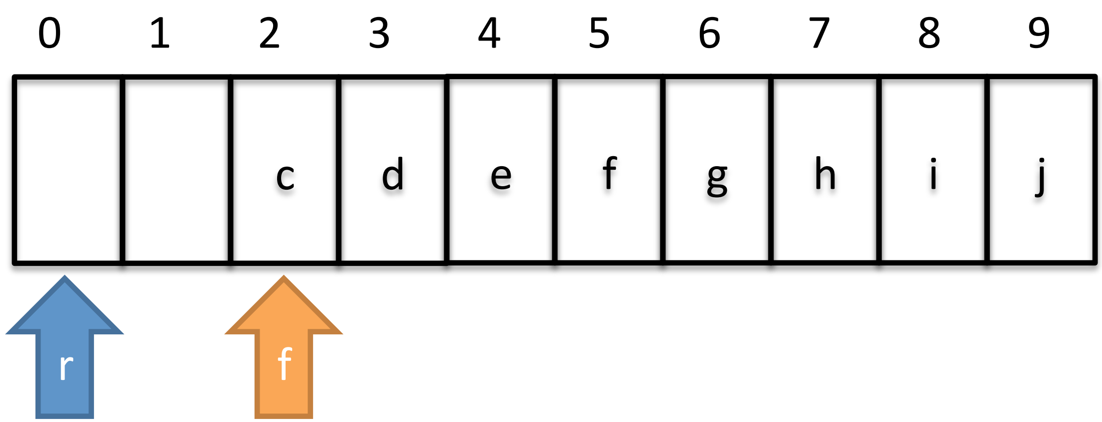
Array implementing Queue




```
dequeue()
dequeue()
```

Array implementing a Queue using index for front and rear

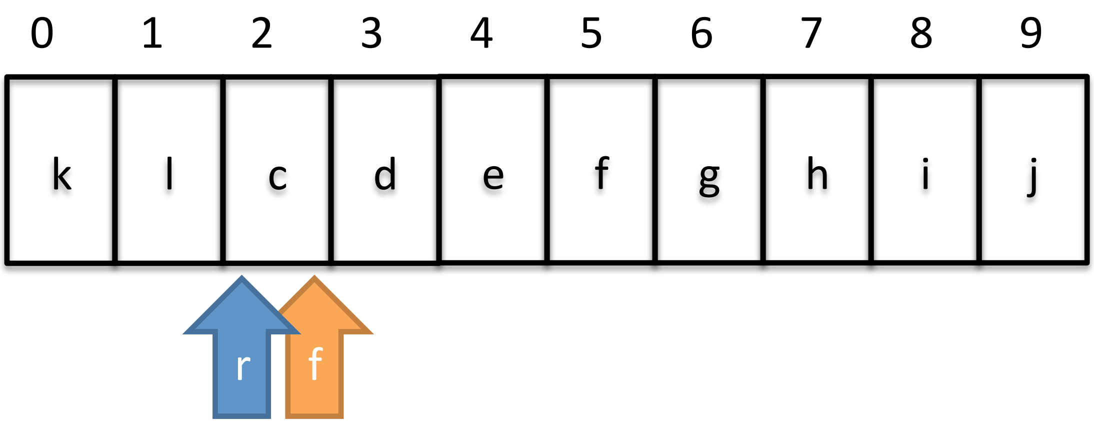
Array implementing Queue



`enqueue(i)`
`enqueue(j)`
Wrap around `r` to front empty spaces due to prior `dequeue` operations

Array implementing a Queue using index for front and rear

Array implementing Queue



`Enqueue(k)`
`Enqueue(l)`
Queue is full `f == r` and `size != 0`
How would extending array size work?
Start with `f` and copy to end of array (2-9), then copy from 0 to `r-1` (0 and 1)