Computer Science 112
Data Structures

Lecture 08:
Stacks
Queues

Review: ArrayLists

- **Partially full arrays**
  - **Make array "big enough"**
  - **keep a variable numInUse: number of elements actually in use**
  - **if numInUse == array.size, we are out of room**
  - **allocate a longer array and copy shorter => longer**
- **operations on ArrayLists**
  - **constructor, add, add at index, get, set, remove**
  - **see DriveAL.java on Sakai**

Amortized big-O

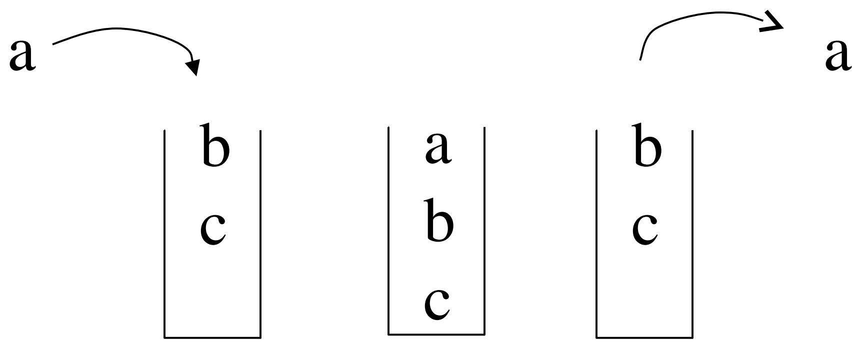
- Increasing numInUse when space is available is very cheap
- Increasing capacity costs more as n gets larger
  - but we do it increasingly rarely
- Average total work to increase size step by step to n is O(n)

Review: Stacks

- Motivation: Last In First Out
- Metaphor: stack of trays in cafeteria
- Operations
- Example use: match parens
- Implementations:
  - ArrayList
  - Array
  - Linked List
- Big-Os

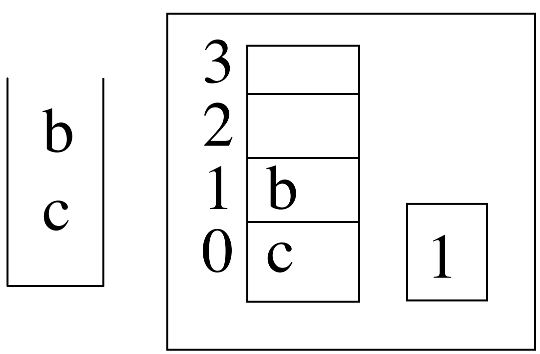
Stacks

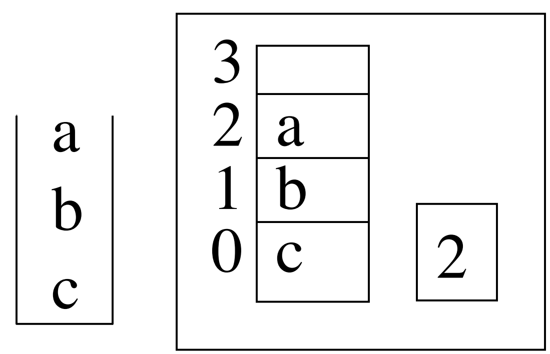
- **Last in first out:**



New: Implementing Stacks

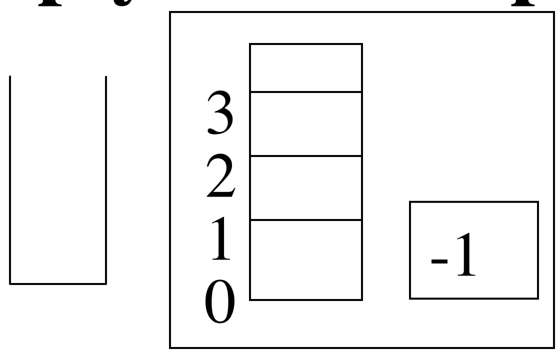
- Stacks can be implemented using
  - Arrays / ArrayLists
  - Linked lists
- Arrays:
  - Array holds data, also need int "top of stack"



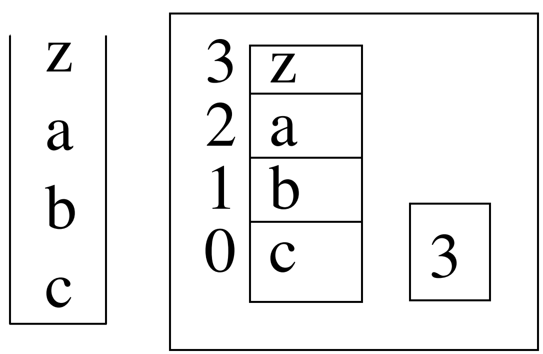


Implementing Stacks

- **Empty stack: top == -1**



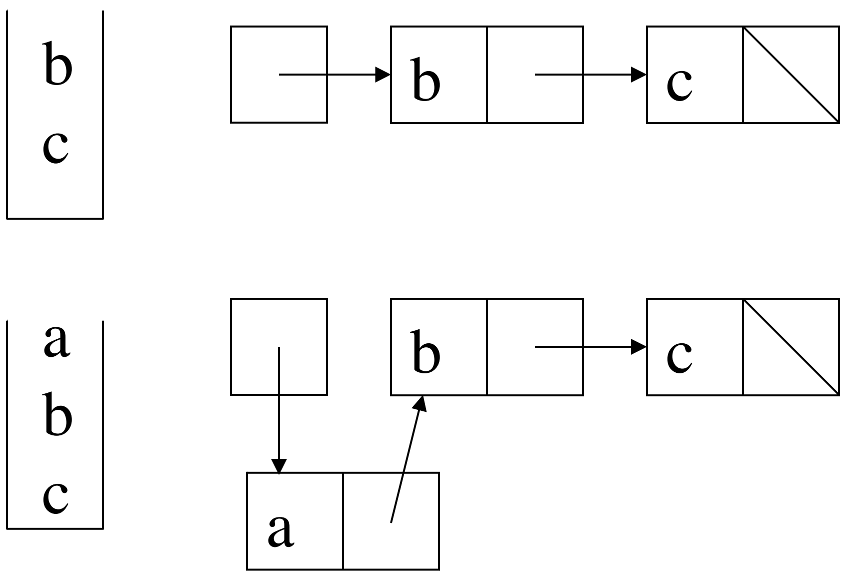
- **Full stack: top == array size -1**



See ALStack.java

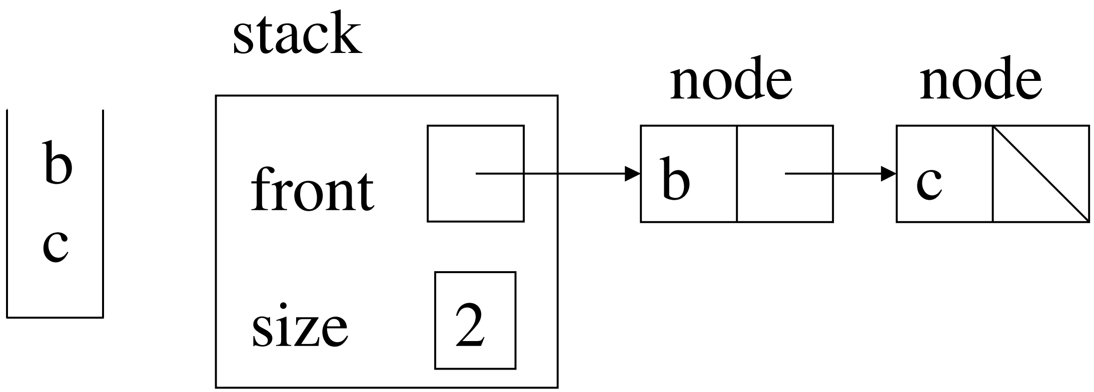
Stacks as linked lists

- **Front of list is top of stack**



Stacks as linked lists

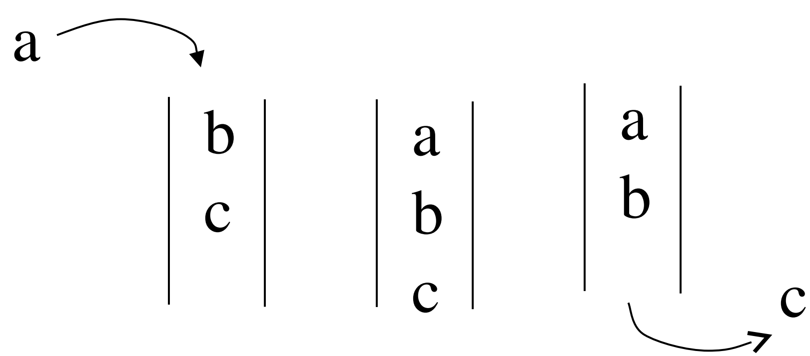
- **Only operation that is not fast is size**
- **So we also have a field to keep track of size**



See Stack.java

Queues

- **First in first out: Queue**



Operations

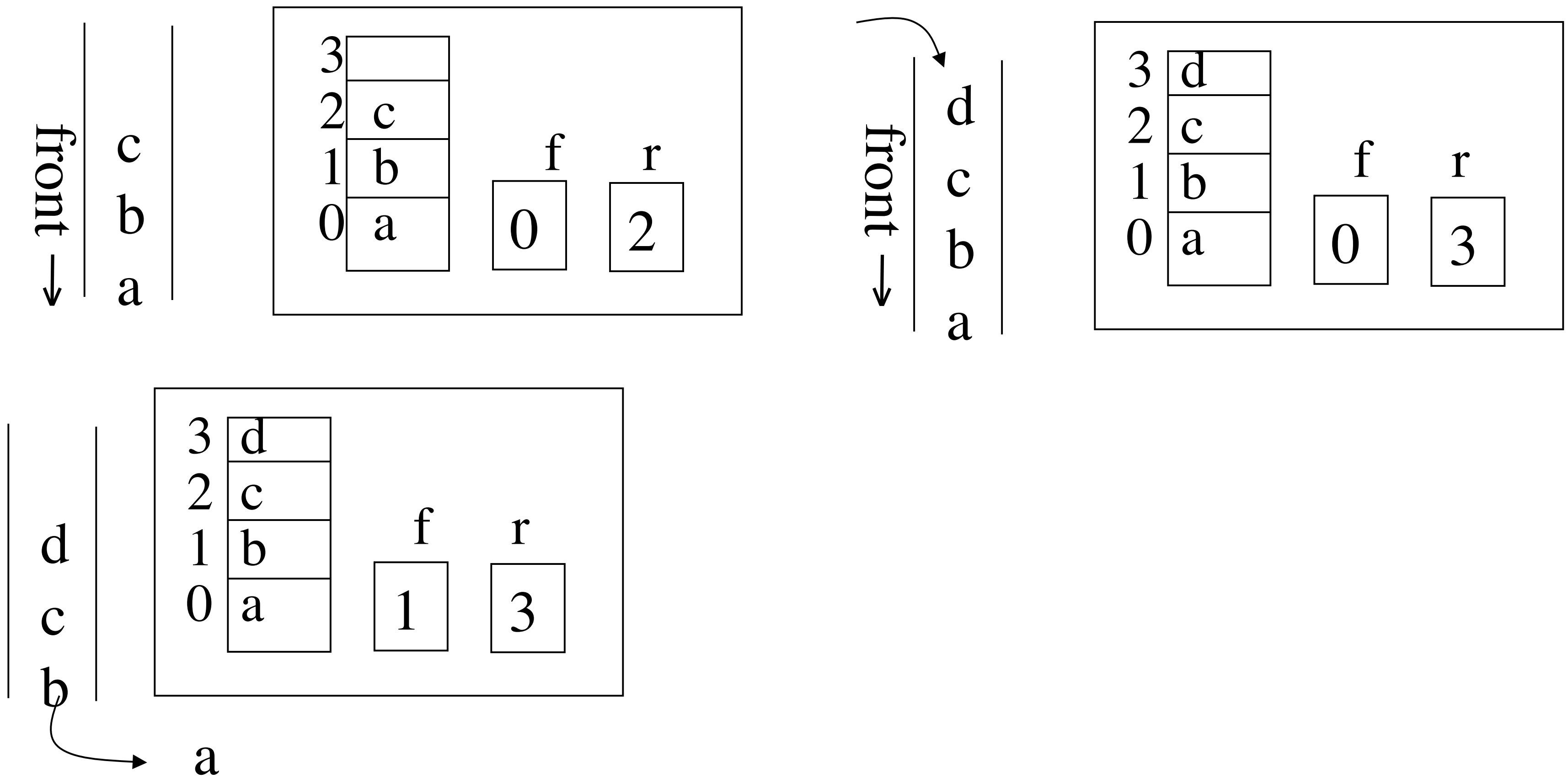
- Queue<T>
  - public void enqueue(T data)
  - public T dequeue( )
  - public boolean isEmpty( )
  - public int size( )
  - public void clear( )

Implementing Queues

- Queues need to be accessed at both ends, so implementations are a bit messier
  - Arrays: need two ints to keep track of both front and back
  - linked lists: use circular lists or have two pointers

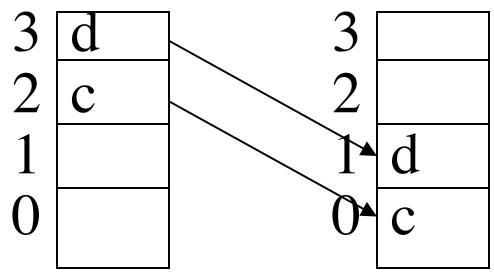
Queues as arrays

- **Keep track of both front & rear**



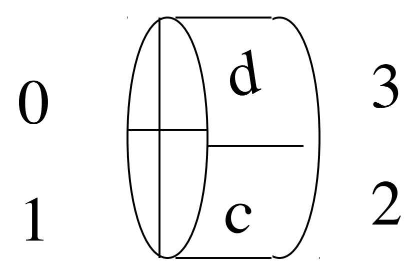
Queues as arrays

- **Problem: how to reuse space emptied by dequeue?**
  - **Could move data down: O(n)**



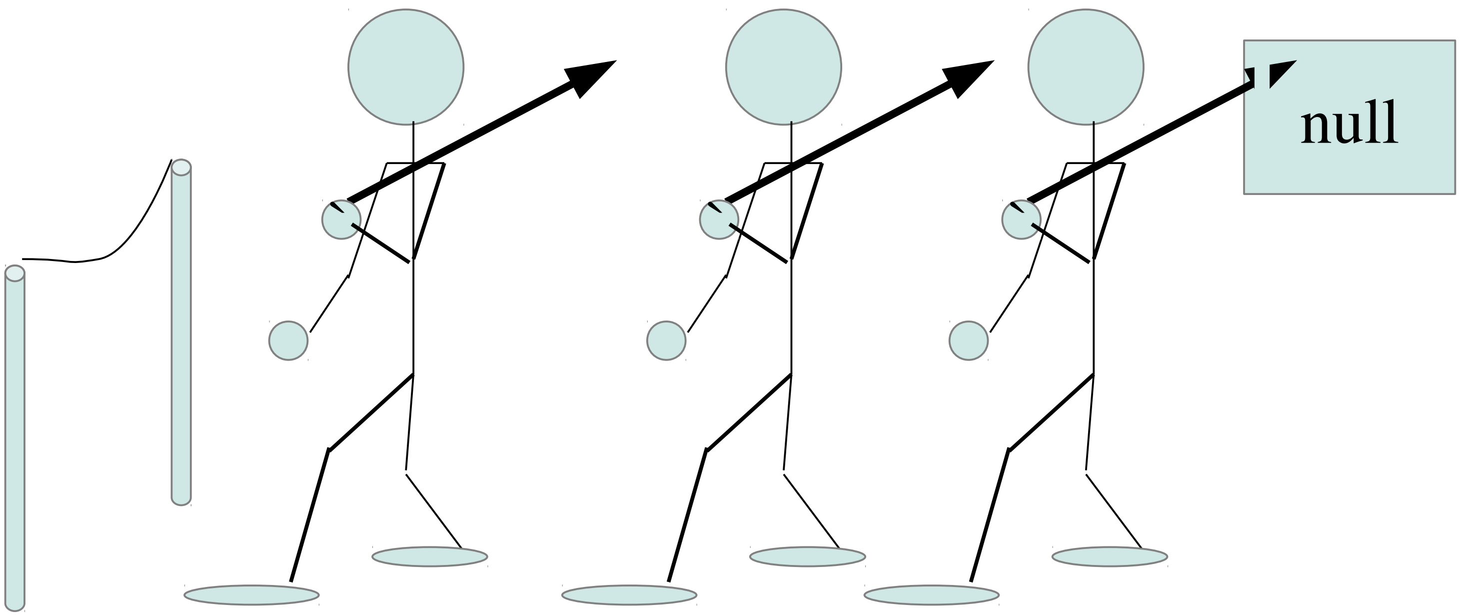
  - **Treat array as circular**

    front = (front + 1) % size



Queues as linked lists

- **Problem: Need to access both ends**
- **Solution: Linked list with head/tail pointer or circular linked list**
- **Which end of the list should be the front of the queue?**
  - **enqueue is O(1) time whether at head or tail**
  - **dequeue is O(1) at head but O(n) at tail (Why?)**
  - **so more efficient when front is head**


null

Sequential Search

- **Is target value in this linked list / array?**

=> Is target the first element? No

Is target the second element? No

…

Is target the 35th element? Yes!

Cost

- **Operation to count:**
  - **test if target equals element**
- **Best case?**
- **Worst case?**
- **Average case … ????**

Input Cases

- **We group the inputs into *cases*.**
- **A case is either**
  - **A specific input**

    target is 5, array is {4, 6}
  - **Or a group of inputs, all with the same cost**

    target is not in the array

Average Cost

- **when we say "average cost" we mean a <u>probability-weighted</u> average.**
- **If all input cases are equally likely, average cost is**

$$\frac{\sum_i C(i)}{N}$$

- **C(i) is the cost of input case i**
- **N is number of different input cases**

Average Cost
Equal Probabilities

- **Suppose 3 possible input cases, with equal probabilities:**

Input Case i	Cost C(i)
Target in element 0	1
Target in element 1	2
Target in element 2	3

Average Cost = Total Cost / N
= (1+2+3) / 3 = 2

Average Cost Different Probabilities

- **If different input cases have different probabilities, average cost is**

$$\sum_i \left(C(i) * P(i) \right)$$

- **C(i) is the cost of input case i**
- **P(i) is the probability of input case i**

Average Cost
Different Probabilities

- **Suppose 3 possible input cases:**

i	Input Case i	Cost C(i)	Probability P(i)
1	target not in array	2	1/2
2	target in element 0	1	1/4
3	target in element 1	2	1/4

Average Cost = C(1)P(1) + C(2)P(2) + C(3)P(3)

= 2 * 1/2 + 1 * 1/4 + 2 * 1/4 = 1.75

Average Cost of Sequential Search

- **Target is in the array, equal probability at each position, length n**

  Average cost = (1 + 2 + … + n) / n

  = (n*(n+1)/2) / n

  = (n+1) / 2