CS112: Data Structures

Lecture 03

Stacks and Queues

Review: Generic Lists

- **Problem: suppose you want to have a list of Strings and a list of ints and …**
- **Class declarations and methods are almost identical**
- **Solution in java 1.5: "generics"**
  - **Class & method definitions parameterized by type**

Generic List

Type parameter, eg String

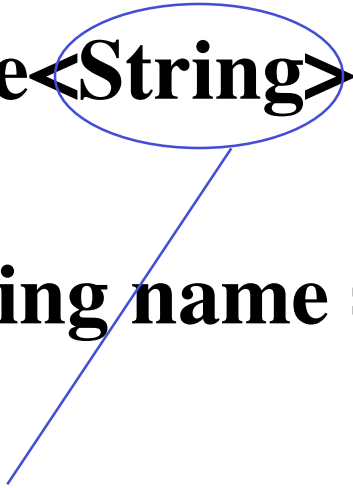
```
public class Node<E> {
  private E data;
  private Node<E> next;

  public Node<E>(E dat, Node<E> nxt){
  data = dat;
  next = nxt;
  }
 public E getHead(Node<E> head){
    E headData = head.data;
  return headData;} …
```

Declare an instance variable of type E, e.g. String

Generic List

In some method:

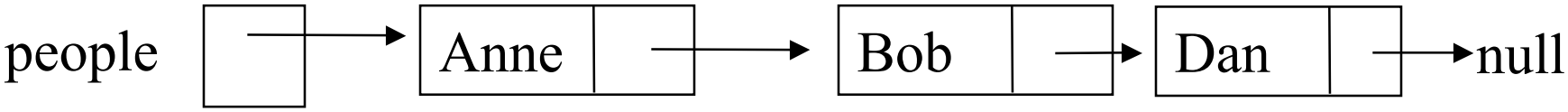
```
Node<String> n1 = new Node<String>;
   …
   String name = n1.getHead( );
```



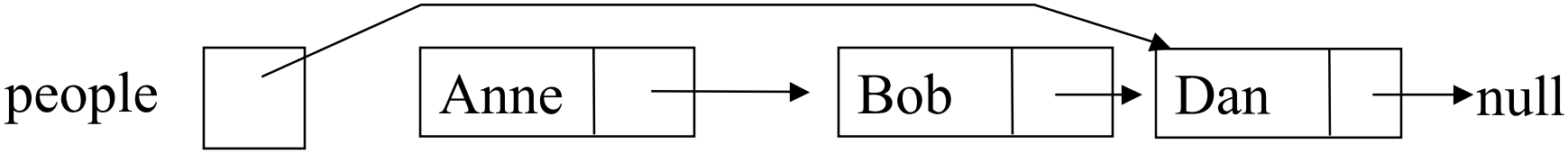
Actual argument corresponding to type parameter E

Circular Lists

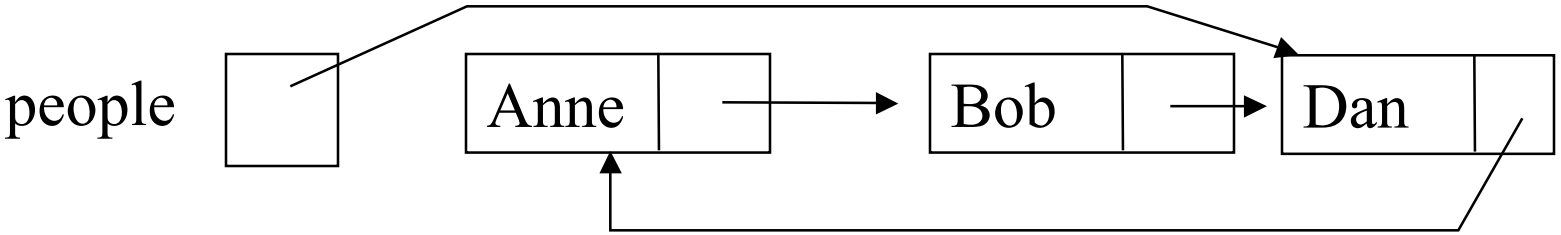
- **Problem: Cost to Access Tail**



- **Solution: point to tail, not head**

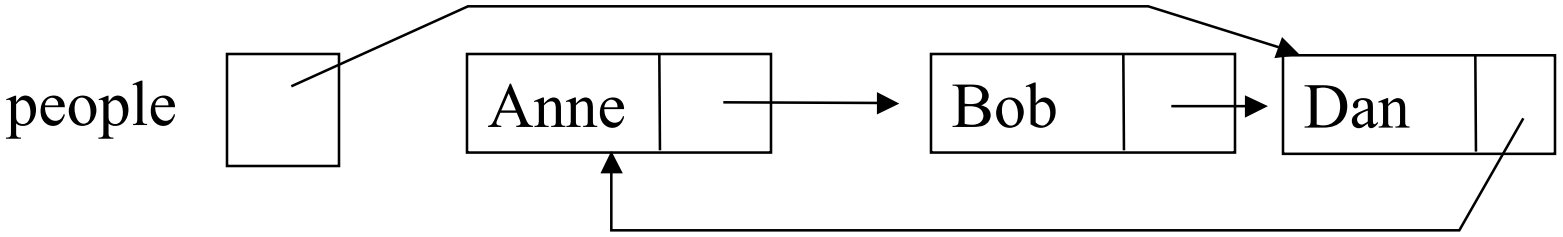


- **Problem: access to head. Solution:**



Circular List

- **First node is people.next**
- **Variable place points at last node when place == people**



Insert at Head

```
if(people == null){
    people =  new Node(newName, null);
    people.next = people;
   } else {
    Node newNode= new Node(newName,
                                        people.next);
    people.next = newNode;
   }
```

Delete Head

- **For circular lists**

```
if (people == null){
} else if (people == people.next){
     people = null;
} else {
     people.next = people.next.next;
}
```

Other CLL Methods

- **See resources => Java examples => fancy lists**

DLL Methods

- **See resources => Java examples => fancy lists**

Dummy Headers

- **Problem: delete head is different that delete elsewhere in list**
  - **Change pointer to list as a whole vs change the next field of some node**
- **Solution: Keep an extra "dummy" node at the head of the list**

Iterators

- **Abstract data type: a container**
  - **E.g. array or linked list**
  - **Can do mostly the same things with them, main difference is cost**
  - **Problem: one of the things I want to do is go through the data items one by one**

Processing Data Items

- **Normal way to handle same-process-different-structure problem is with a method that is defined appropriately for each class**
  - **Same name and abstract behavior**
  - **Different code**
- **But what would be the interface?**
  - **What changes from call to call? Pieces of progam** ☹

Solution

- **Instead of a method to do whole loop, have methods you can use to build the loop**
  - **hasNext**
  - **getNext**
- **State: an object**
  - **Represents a particular instance of iteration**
  - **Initialized by new**

Abstract List Traversal

- **while (list.hasNext()) {**
  print(list.getNext().data);
  }

- **list could be an Array:**
  hasNext() { return (i != list.length) }
  getNext() { i++; return list[i]; }

Abstract List Traversal

- **while (list.hasNext()) {**
  print(list.getNext().data);
  }

- **list could be a LinkedList:**
  hasNext() { return (curr != null) }
  getNext() { curr = curr.next;
  return curr; }

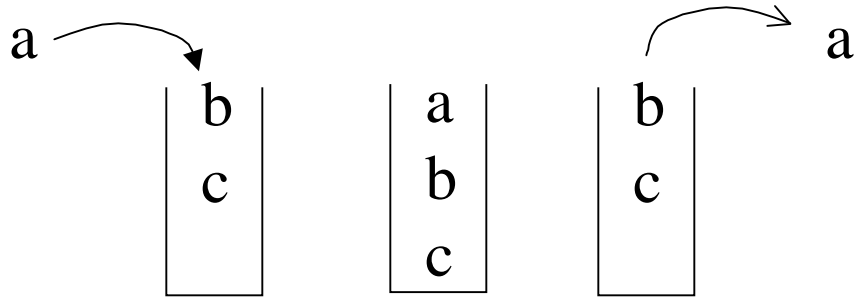
Iterators

- **See StringList.java and StringListIterator.java**

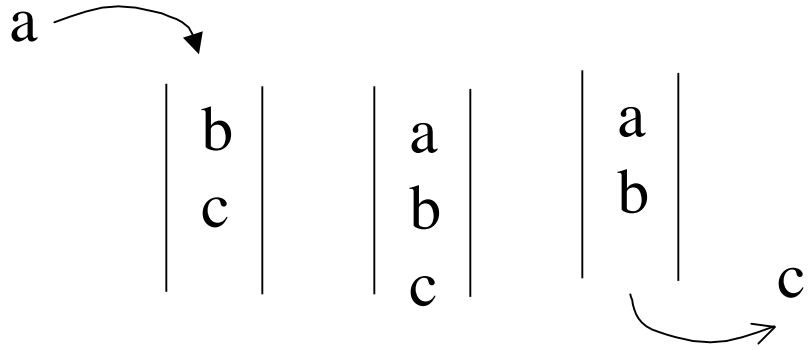
Stacks & Queues

Stacks & Queues

- **Last in first out: Stack**



- **First in first out: Queue**



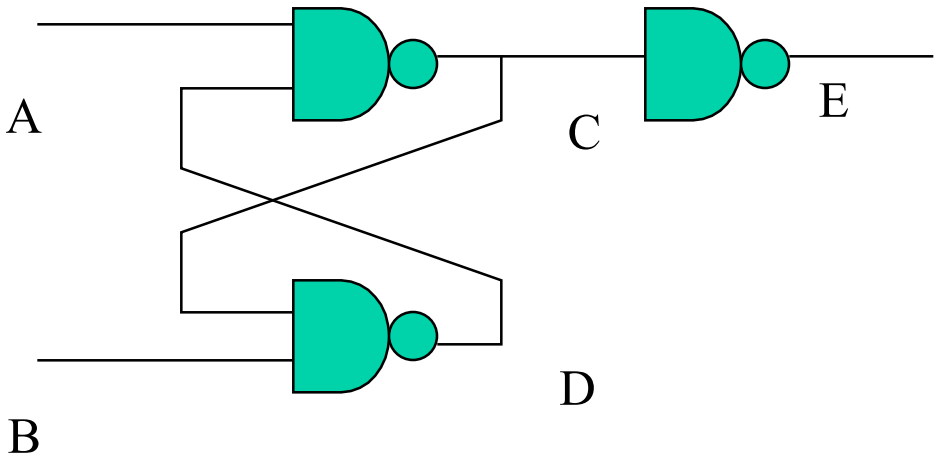
Operations

- **Queue**
  - **enqueue, dequeue**
  - **isEmpty, size**
  - **clear, remove, removeAll**
  - **first, next (Enumerator would be better.)**
- **Stack**
  - **same but enqueue, dequeue called push, pop**

Uses of Queues

- **Printer queue**
- **Simulation of real world queues**
  - **Queue in simulator models line of students.**
- **More generally, waiting lists when processing one item creates two more items to process**
  - **E.g. simulator**
  - **E.g. family tree**

Circuit Simulator



Initial state:

A,B,E = 0

C,D = 1

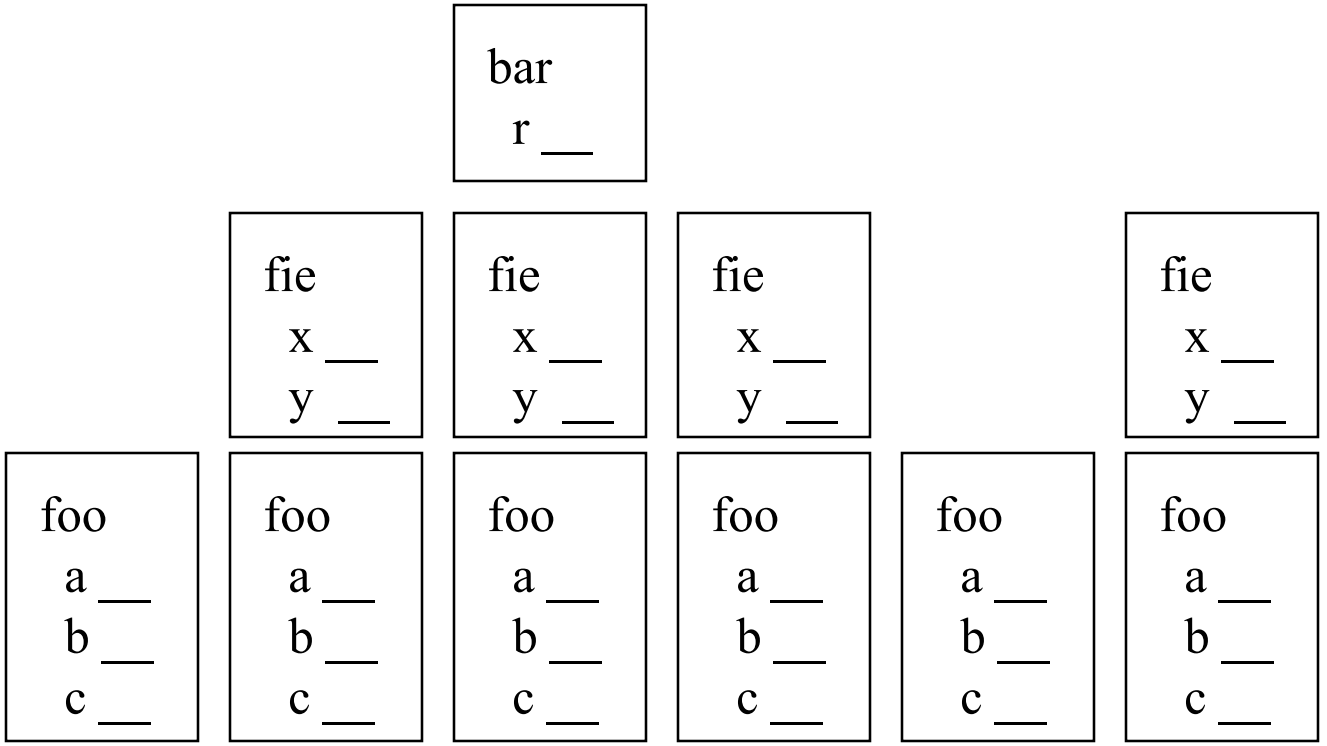
Change at A => change at C	C
Change at C => changes at E, D	E D
Change at E => no effect	D
Change at D => change at C	C

Invocation Record

- **Each procedure / method call needs to record values of**
  - **Parameters**
  - **Local variables**
  - **Other things**
- **When a procedure starts, space allocated for "invocation record" to store these things.**
- **Invocation record is kept until invocation exits**
- **Behavior is LIFO**

Stack of Invocation Records

```
Proc foo(int a)
  … int b, c;
  … fie(b);
  … fie(c );
Proc fie(int x)
  …int y;
  …bar(y);
Proc bar(int r)
  …
```

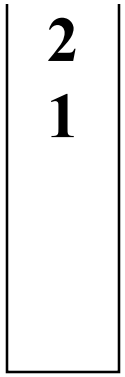


Uses of stacks

- **Postfix (RPN) calculator**
  - **Permits any expression to be evaluated**
  - **Does not require parentheses**

((1 + 2) * (3 + 4)) / 7

1 2 + 3 4 + * 7 /

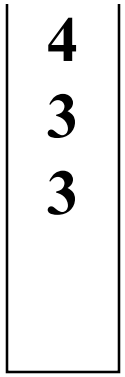


Uses of stacks

- **Postfix (RPN) calculator**
  - **Permits any expression to be evaluated**
  - **Does not require parentheses**

((1 + 2) * (3 + 4)) / 7

1 2 + 3 4 + * 7 /



Uses of stacks

- **Postfix (RPN) calculator**
  - **Permits any expression to be evaluated**
  - **Does not require parentheses**

((1 + 2) * (3 + 4)) / 7

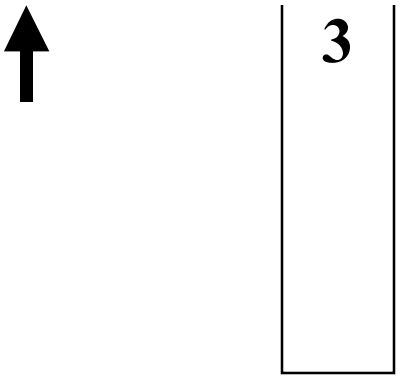
1 2 + 3 4 + * 7 /

Uses of stacks

- **Postfix (RPN) calculator**
  - **Permits any expression to be evaluated**
  - **Does not require parentheses**

((1 + 2) * (3 + 4)) / 7

1 2 + 3 4 + * 7 /

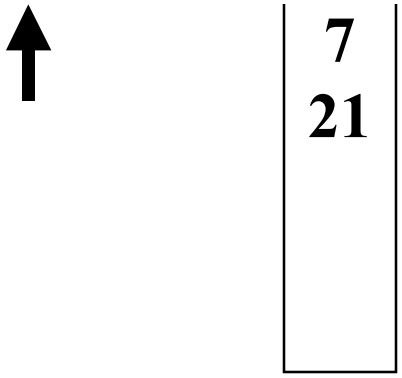


Uses of stacks

- **Postfix (RPN) calculator**
  - **Permits any expression to be evaluated**
  - **Does not require parentheses**

((1 + 2) * (3 + 4)) / 7

1 2 + 3 4 + * 7 /



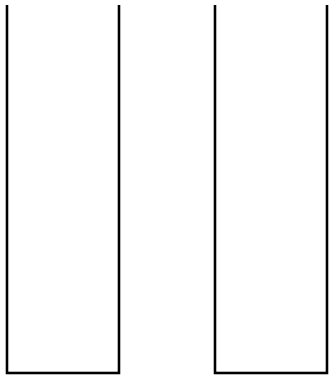
Uses of stacks

- **Interpret infix-with-precedence**
  - **Each operator has a numeric precedence**
    + 10, * 20, > 5, < 5
  - **Two stacks: operators, operands**
  - **scan expression:**
    - **operand: push**
    - **operator:**
      - **if operator stack empty or**
        precedence of op in string > prcedence of top of stack, push
      - **else: pop operator , pop operands, push result**
- **e.g. 1 + 2 * 3 > 4 * 5**

Uses of stacks

- operand: push
- operator:
  - if operator stack empty or
    precedence > top of stack, push
  - else: pop operator & do

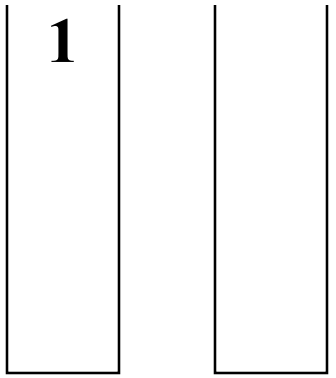
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

- operand: push
- operator:
  - if operator stack empty or
    precedence > top of stack, push
  - else: pop operator & do

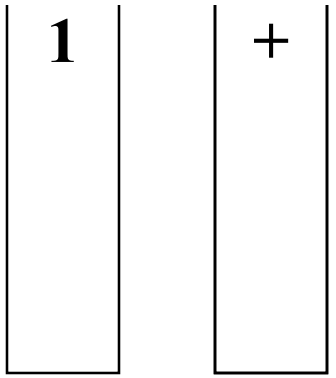
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

- operand: push
- operator:
  - if operator stack empty or
    precedence > top of stack, push
  - else: pop operator & do

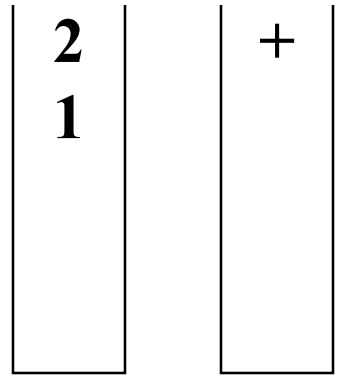
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

- operand: push
- operator:
  - if operator stack empty or
    precedence > top of stack, push
  - else: pop operator & do

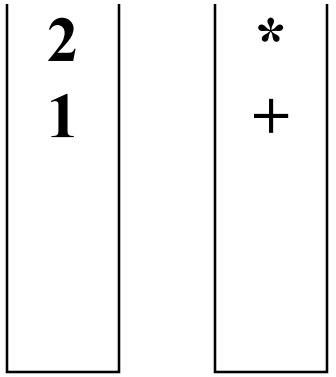
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

- operand: push
- operator:
  - if operator stack empty or
    precedence > top of stack, push
  - else: pop operator & do

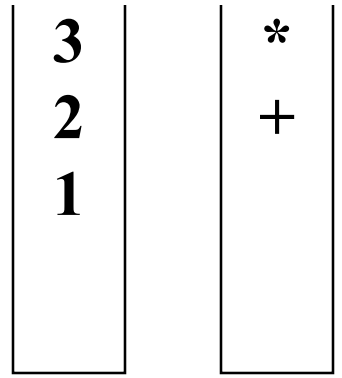
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

- operand: push
- operator:
  - if operator stack empty or
    precedence > top of stack, push
  - else: pop operator & do

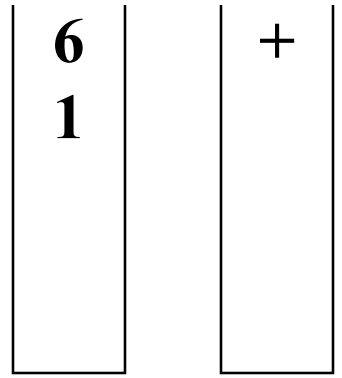
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

- operand: push
- operator:
  - if operator stack empty or
    precedence > top of stack, push
  - else: pop operator & do

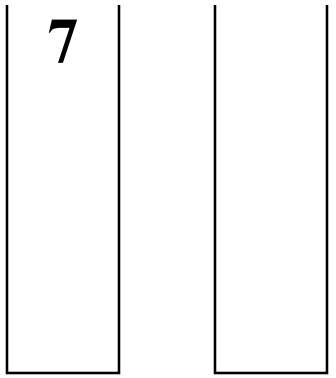
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

- operand: push
- operator:
  - if operator stack empty or
    precedence > top of stack, push
  - else: pop operator & do

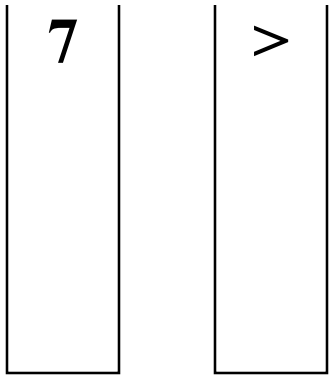
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

- operand: push
- operator:
  - if operator stack empty or
    precedence > top of stack, push
  - else: pop operator & do

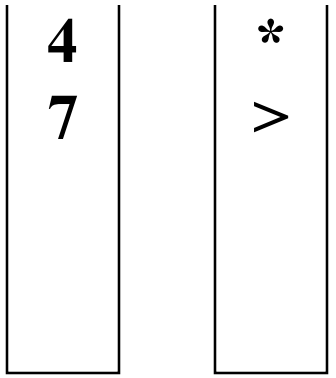
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

- operand: push
- operator:
  - if operator stack empty or
    precedence > top of stack, push
  - else: pop operator & do

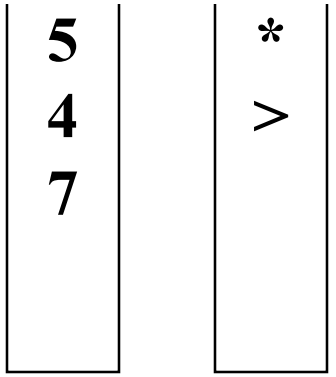
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

- operand: push
- operator:
  - if operator stack empty or
    precedence > top of stack, push
  - else: pop operator & do

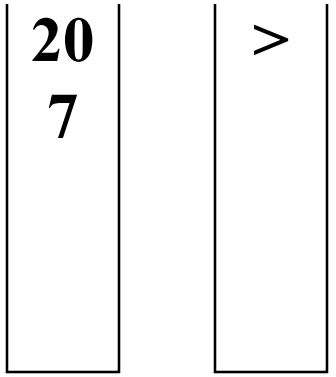
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

- operand: push
- operator:
  - if operator stack empty or
    precedence > top of stack, push
  - else: pop operator & do

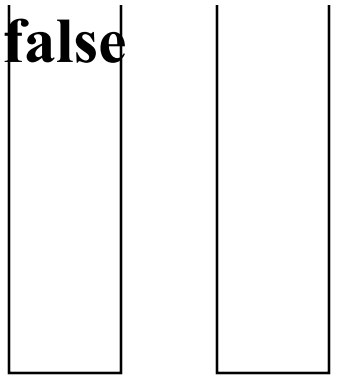
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

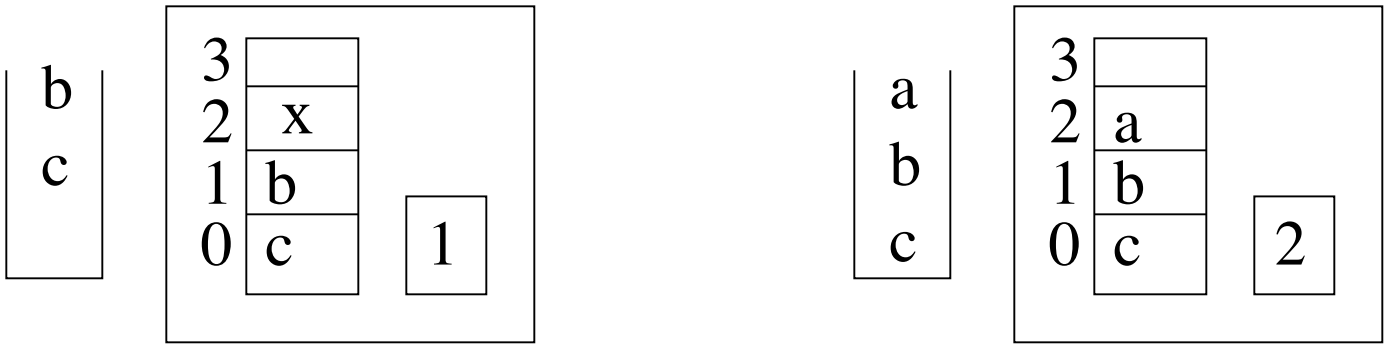
- operand: push
- operator:
  - if operator stack empty or
    precedence > top of stack, push
  - else: pop operator & do

- e.g. 1 + 2 * 3 > 4 * 5



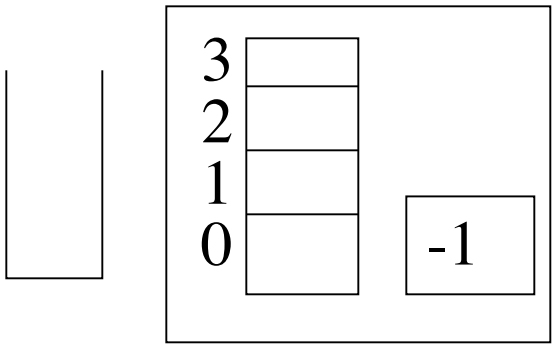
Implementing Stacks

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

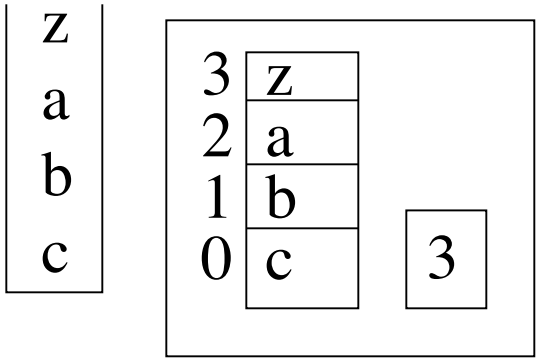


Implementing Stacks

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

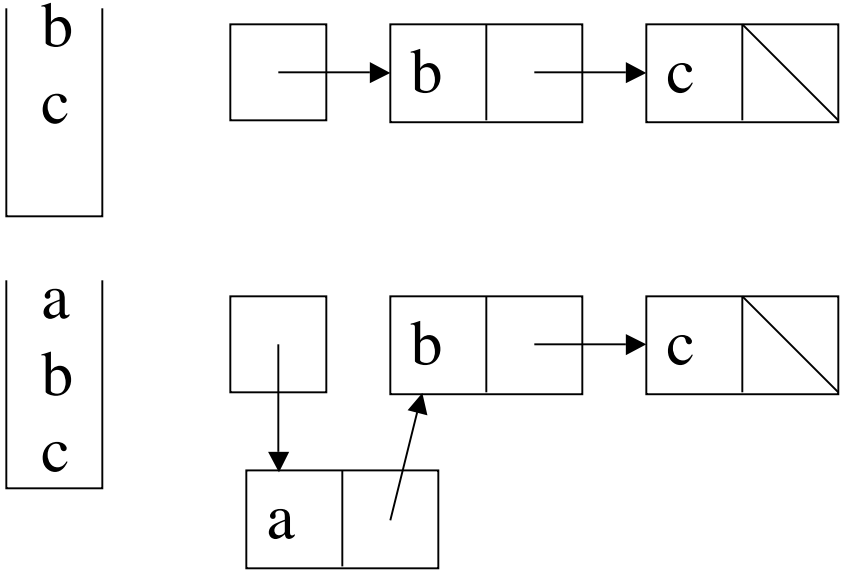


- **Full stack: top == array size -1**



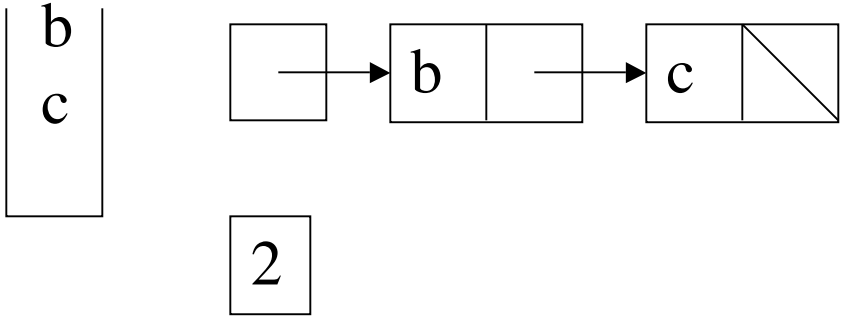
Stacks as linked lists

- **Linked lists are easy to manipulate at head -> natural representation for stacks**



Stacks as linked lists

- **Only operation that is not fast is size**
- **So also have an int**

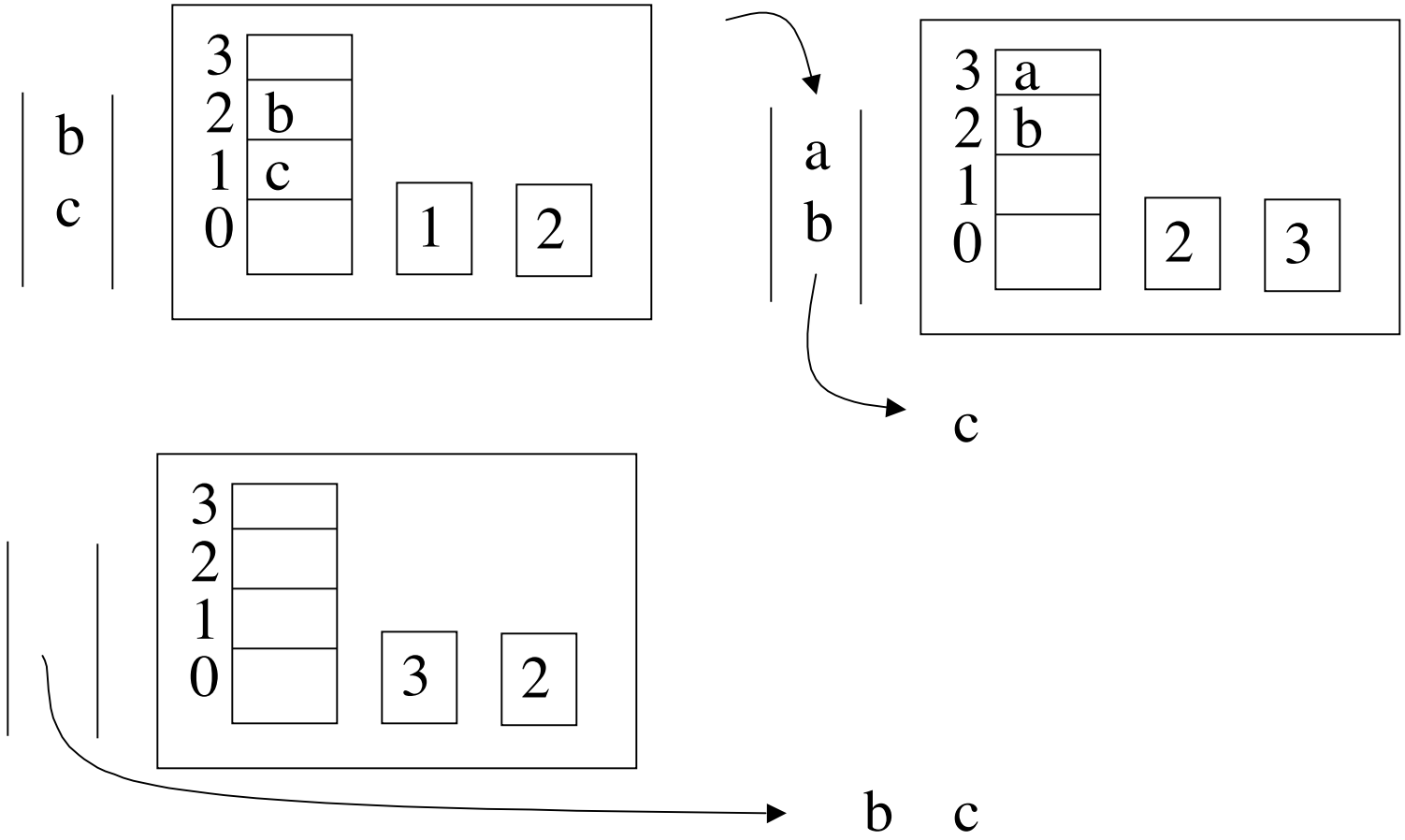


Implementing Queues

- **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**

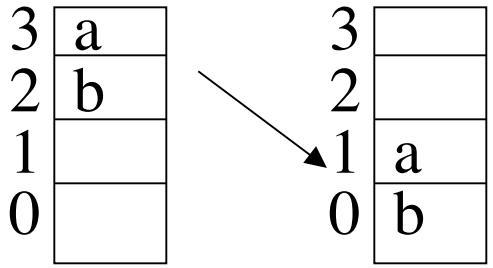
Queues as arrays

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



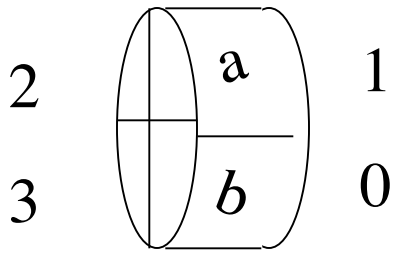
Queues as arrays

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



  - **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**
- **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? Need pointer to second to last.)**
  - **so more efficient when front is head**