ENSF 593/594
Data Structures – Stacks and Queues

Mohammad Moshirpour

Outline

- Stack
  - Push
  - Pop
- Queue
  - Enqueue
  - Dequeue
- Priority Queues

Goal

- In this lecture we will be introduced to important data structures called stacks and queues. We will study the implementation of these data structures using arrays and linked lists.

Stacks

- Are last in, first out (LIFO) linear data structures
- Can only be accessed at the “top” using push and pop operations
- A place where additions and deletions are made at the same location in the data structure (on one side).

Stacks (cont'd)

- *Push*: store an element on the top of the stack
  - If the stack has a maximum size, you cannot push when the stack is full
- *Pop*: remove and return the element on the top of the stack
  - You cannot pop an empty stack

Stacks (cont'd)

- May implement additional operations to:
  - Return the top element without popping
  - Clear the entire stack
  - Check if the stack is full
  - Check if the stack is empty

Stacks (cont'd)

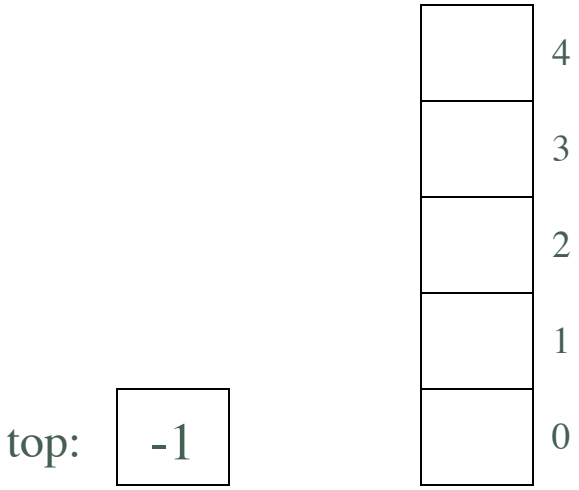
- Stacks may be implemented using arrays
  - Must use a variable to point to the top
    - Will initially be set to -1, to indicate an empty stack
    - E.g. `int top = -1;`
  - Will have a maximum size
    - Unless a resizable array (vector) is used
  - To push, increment *top*, and store the element at that position in the array
    - E.g. `array[++top] = elementValue;`

Stacks (cont'd)

- To pop
  - Copy the top element in the array to a temporary variable
  - Decrement *top*
  - Return the value in the temporary variable
  - E.g. `temp = array[top--];` `return temp;`

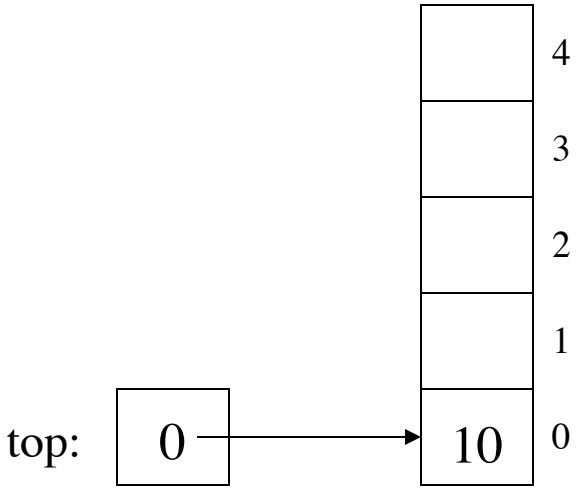
Stacks (cont'd)

- E.g.
  - Empty stack



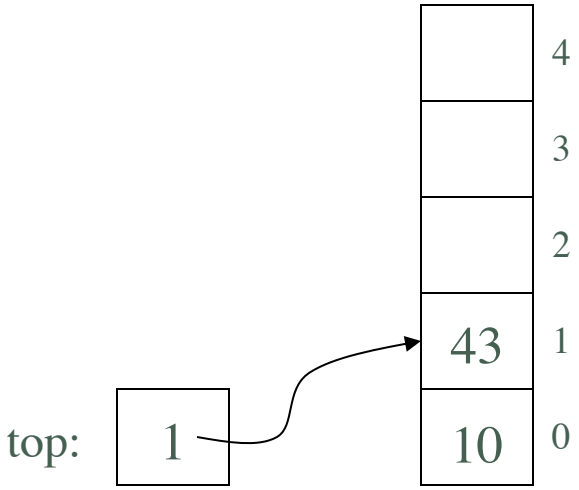
Stacks (cont'd)

- Push 10



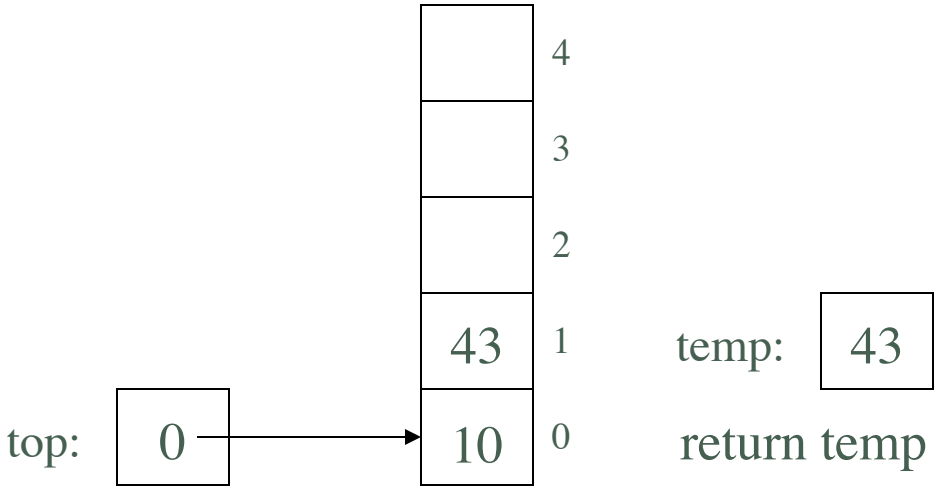
Stacks (cont'd)

- Push 43



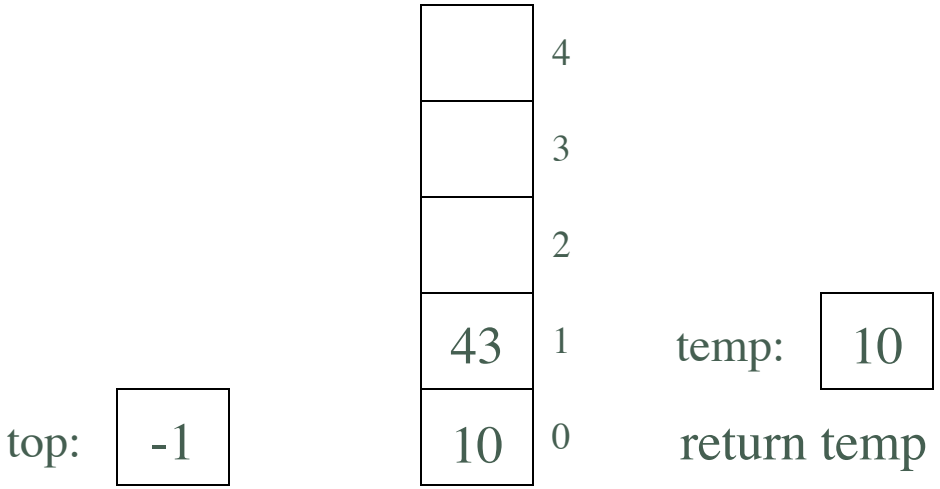
Stacks (cont'd)

- Pop



Stacks (cont'd)

- Pop



Stacks (cont'd)

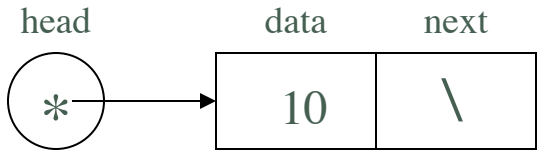
- Stacks may also be implemented using linked lists
  - Unlike arrays, have no maximum size
  - To push, insert the element at the head of the list
  - To pop, copy the element at the head of the list, delete the node, then return the element

Stacks (cont'd)

- E.g.
  - Empty stack

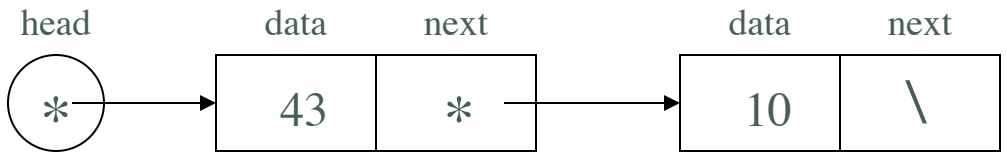
Stacks (cont'd)

- Push 10



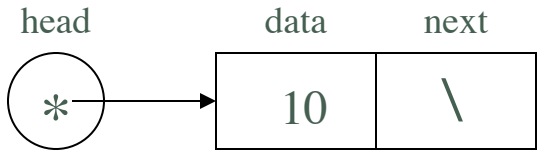
Stacks (cont'd)

- Push 43



Stacks (cont'd)

- Pop

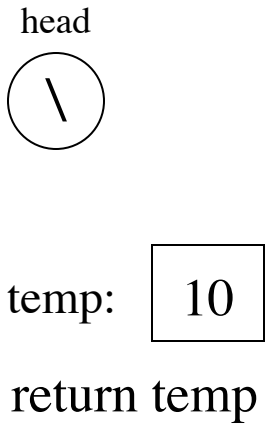


temp: 43

return temp

Stacks (cont'd)

- Pop



Stacks (cont'd)

- Push and pop are constant time operations
  - i.e. *O*(1)
- Java provides a generic implementation with the class java.util.Stack
  - Extends Vector, so not a true stack
    - i.e. Allows access to elements not at the top

Queues

- Are analogous to lineups at store checkouts
- Are linear data structures that follow a “first in, first out” policy
  - i.e. are FIFO queues
  - Elements can only be accessed at the head and tail of the list

Queues (cont'd)

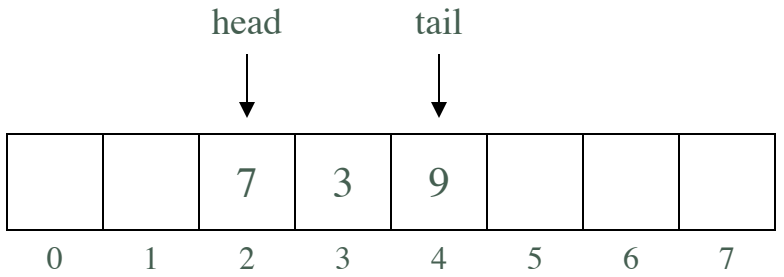
- Have two basic operations:
  - *Enqueue*: add an element to the end of the list
    - If the queue has a maximum size, you cannot enqueue to a full queue
  - *Dequeue*: delete and return the element at the beginning of the list
    - You cannot dequeue from an empty queue

Queues (cont'd)

- May implement additional operations to:
  - Return the first element without dequeuing
  - Clear the entire queue
  - Check if the queue is full
  - Check if the queue is empty

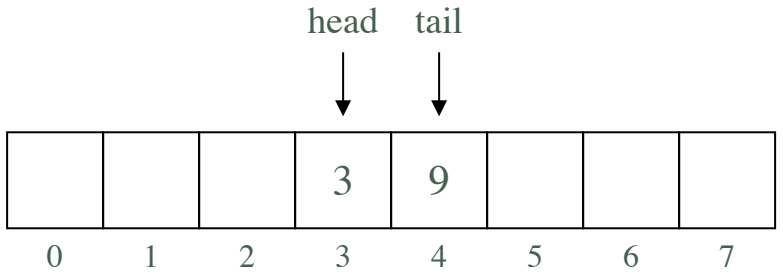
Queues (cont'd)

- May be implemented using an array
  - Use two variables to point to the beginning and end of the list
    - The "head" index is incremented after dequeuing, the "tail" index when enqueuing
    - E.g.

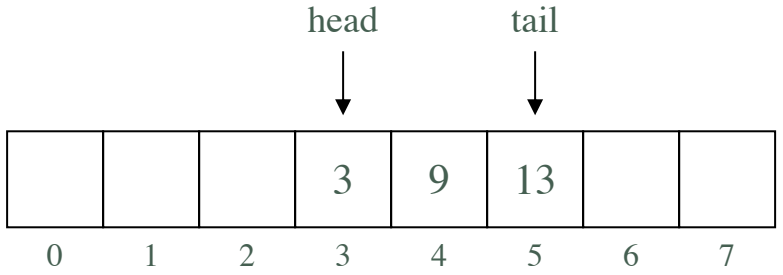


Queues (cont'd)

Dequeue

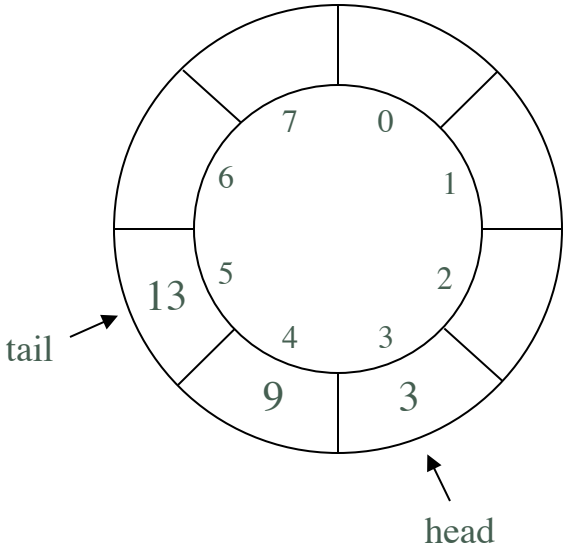


Enqueue 13



Queues (cont'd)

- Since the indices will eventually run off the end, the array is "wrapped around" to form a *circular array* (*ring buffer*)
  - E.g.



Queues (cont'd)

- Modulus arithmetic must be used when incrementing the indices
  - i.e. Keep them in the range 0 to N–1, where N is the size of the array
- Head and tail are set to -1 to indicate an empty queue

Queues (cont'd)

- To enqueue:
  - If the queue is empty
    - Set head and tail to 0
  - Else
    - Increment tail mod N
  - Set array[tail] to element value

Queues (cont'd)

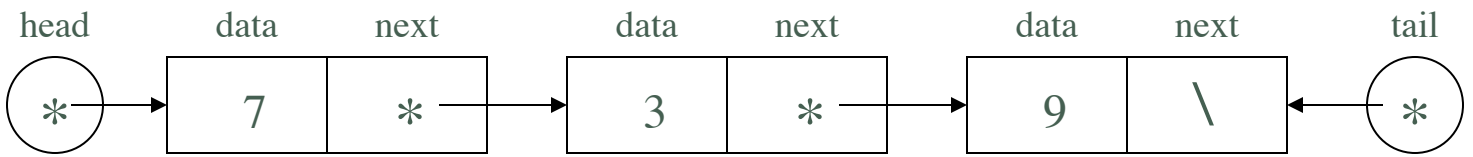
- To dequeue:
  - Store array[head] in a temporary variable
  - If only one element in the queue (head == tail)
    - Set head and tail to -1 (indicates empty queue)
  - Else
    - Increment head mod N
  - Return the value in the temporary variable

Queues (cont'd)

- Queues may also be implemented using a singly linked list, with head and tail pointers
  - Unlike arrays, have no maximum size
  - To enqueue, insert the element at the tail of the list
  - To dequeue, copy the element at the head of the list, delete the node, then return the element

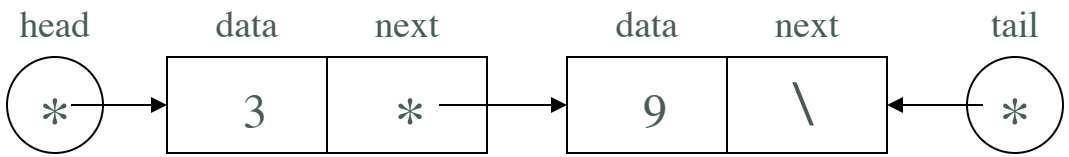
Queues (cont'd)

- E.g.
  - Original list



Queues (cont'd)

- Dequeue

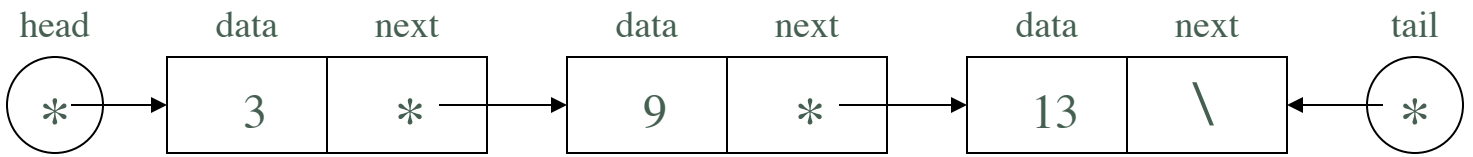


temp: 7

return temp

Queues (cont'd)

- Enqueue 13



Queues (cont'd)

- Enqueue and dequeue are constant time operations
  - i.e. $O(1)$

Priority Queues

- Are linear data structures that store *prioritized elements*
- Each element has an associated *priority*
  - Usually a numeric value, where the smallest value means the highest priority
  - Stored as a key in the node for an element
- When dequeuing, one always removes the element with the highest priority (lowest key) from the list

Priority Queues (cont'd)

- May be implemented using an unsorted linked list
  - New elements are always added to the tail
    - i.e. Do the standard enqueue operation
    - Is $O(1)$
  - To dequeue the highest priority element, one must search the entire list for the lowest key
    - Is $O(n)$ in the best and worst cases

Priority Queues (cont'd)

- May be implemented using a sorted linked list
  - New elements are inserted into the list in their proper position using the key
    - Is $O(n)$ in the worst case
  - To dequeue the highest priority element, simply remove the first element
    - Is $O(1)$

Priority Queues (cont'd)

- Other possible implementations:
  - Use a separate (linked) list for each priority group (level)
    - Or references to the beginning and ends of sublists within a larger list
  - Use a type of binary tree called a *heap*
    - Covered later in the course

Summary

- Stack is a last in first out (LIFO) data structure
- Two important operations on stack are push and pop
- Stacks may be implemented as arrays or linked list
- Push and pop operation is O(1)

Summary (Cont'd)

- Queue is a first in, first out (FIFO) data structure
- Two important operations on queue are Enqueue and Dequeue
- Queues may be implemented as arrays or linked list
- Enqueue and Dequeue operation is O(1)
- A linear data structures that store prioritized elements (priority queue) that can be implemented as sorted, unsorted linked list or a set of lists

Review Questions

- What is a stack?
- What are the two data structures that can be used to implement stack?
- Explain the algorithm for pushing and popping from a stack implemented by an array.
- Explain the algorithm for pushing and popping from a stack implemented by a linked list.

Review Questions (Cont'd)

- What is a queue?
- What are the two data structures that can be used to implement queue?
- Explain the algorithm for enqueuing and dequeuing from a queue implemented by an array.
- Explain the algorithm for enqueuing and dequeuing from a queue implemented by a linked list.

Any questions?