Stacks and Queues

CS 580U Fall 2017

ADTs describe behavior

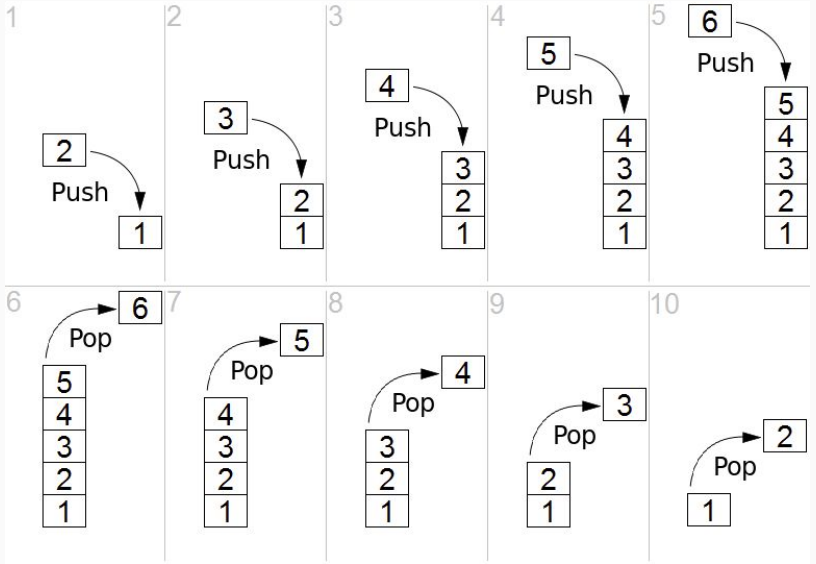
- Data Structure’s describe behavior, not implementation
- This means that a specific Data Structure should have a conventional interface, but implementation can vary greatly

Stacks

- A Sequential Collection
  - Sequential
    - *Order of element insertion matters*
  - Collection
    - *Used to group items of the same type*
- Purpose is to restricts how the data is accessed, not how the data is stored
  - Provides access control through encapsulation

LIFO

- Example: Stack of Books
- Last in First Out
  - The order items are inserted and removed
- Stacks limit the user to removing elements in reverse order of their arrival



Examples of Stack

- Stack of Books
- Undo in a word processor
- The function stack

Push / Pop

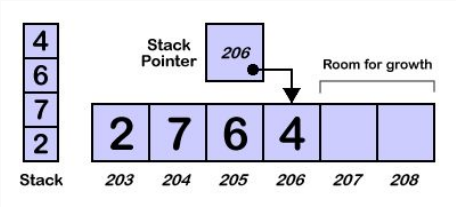
- Push
  - Inserting onto a stack is called a push
  - You can imagine pushing everything in the stack down
- Pop
  - Removing from the stack is called pop
  - Every element in the stack pops up whenever you remove an element
  - To access an element, you must take every element before it off the stack

Stack Public Interface

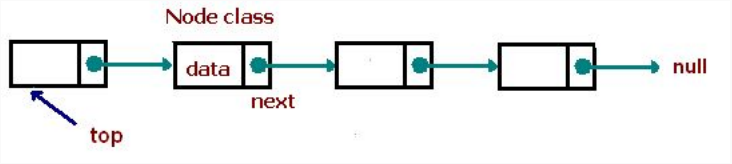
```
struct Stack { // Stack struct ADT

        int length;  // Return the number of elements in the stack
        Collection items;

}

void clear(Stack *); // Reinitialize the stack.
int push(Stack *, Data d);  // Push "d" onto the top of the stack
Data pop(Stack *);  // Remove and return the element at the top of the stack
Data peek(Stack *);  // Return a copy of the top element
```

Two Implementations

- Array Based
  - top - an index pointing to the ‘top’ of the list
    - *the top of the stack is the end of the list*
  - the initialization size
  - pros
    - *easier memory management*
  - cons
    - *requires a static size*



- Linked Based
  - Can use your existing linked list as the underlying data structure
  - pros
    - *internal memory is dynamic*
  - cons
    - *more complex memory management*

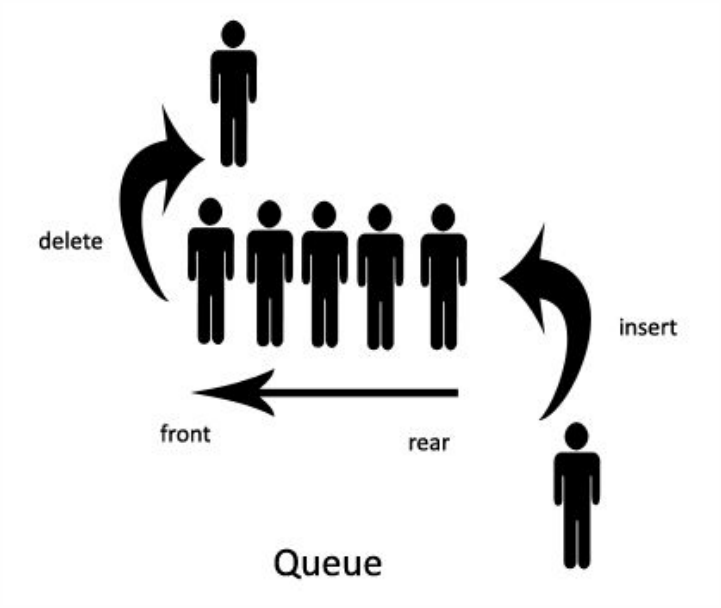


Queues

- Another Collection Data Structure
- Purpose is also to restrict how the data is accessed
  - Provides access control through encapsulation
- A Sequential data structure
  - Order of element insertion matters

FIFO

- First In First Out
- queues remove elements in the order of their arrival
- Example
  - Grocery Store Lines
  - The input/output buffer



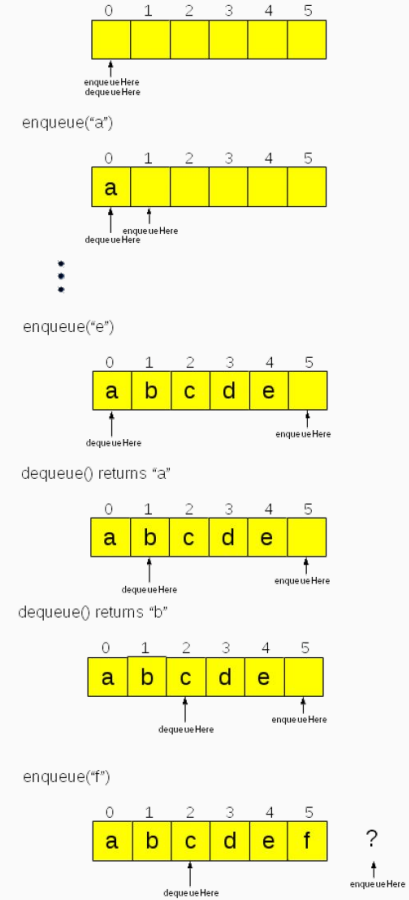
Dequeue / Enqueue

- Enqueue
  - Inserting onto a queue is called a enqueue
  - You can imagine the data getting in line
- Dequeue
  - Removing from the stack is called dequeue
  - Every element in the stack moves up whenever you remove an element
  - To access an element, you must take every element before it off

Queue Public Interface

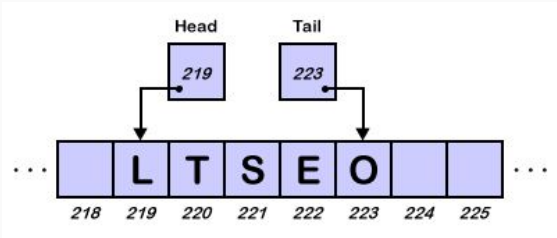
```
struct Queue { // Stack class ADT
        int length;  // Return the number of elements in the stack
}
void clear(Queue *); // Reinitialize the stack.
int enqueue(Queue *, Data d);  // Push "d" onto the top of the stack
Data dequeue(Queue *);  // Remove and return the element at the top of the stack
Data peek(Queue *);  // Return a copy of the top element
```

Array Based Queue Implementation

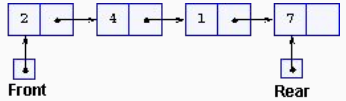


Two Implementations

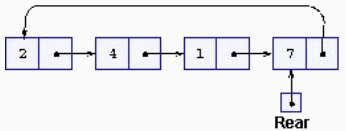
- Array Based
  - top and bottom - an index pointing to the ends of the list
  - need initial size
    - *the initialization size*
  - pros
    - *easier memory management*
  - cons
    - *requires a static size*



- Linked Based
  - Can use your existing linked list as the underlying data structure
  - pros
    - *internal memory is dynamic*
  - cons
    - *more complex memory management*



Linked List



Classwork: Stacks and Queues