Stacks and Queues

Warmup question:

Implement this constructor for a linked list:

```
SinglyLinkedList(ArrayList<t> arr)
```

New ADTs

- Today we'll cover 2 new ADTs
- They're sort of similar to the list ADT, but much simpler
- They only allow access to elements at the end(s) of the collection, and do not support iteration or access anywhere else
- They might seem extremely limited but LOTS of applications need just these operations and not all the features of a list!

Stack ADT

```
public interface Stack<t> {
    T peek(); //look at top element
    T pop(); //remove and return top element
    void push(T t); //add element to top
}
```

- Elements are added/removed from the same “end”
- Also known as “last in first out” (LIFO or FILO)
- Fewer supported operations than list... why bother?

Why simple ADTs?

- For many applications (like tracking data associated with method calls) we only need push/pop/peek operations
- Keeping the set of operations small allows lots of flexibility in implementation choice
- Basically we're designing based on usage requirements, not “what we can do with a particular implementation strategy”

Stack implementations

- We need to store many elements, and the order they're added matters
- So any sort of “List” like data structure will potentially work
- We'll look at Array (really ArrayList) based and Linked List implementations

ArrayList implementation

- We only need to add/remove elements from one end of the stack (the “top”)
- Which end should we use in an arraylist implementation?
- Adding/removing from the back of an arraylist is fast (O(1)), basically just `size--` or `size++`
- Adding/removing from the front is slow (O(N)), requires shifting all elements left/right

ArrayList Stack

```java
class ALStack<t> {
    private ArrayList<t> data;

    void push(T t){ data.add(t); }
    T peek(){ return data.get(data.size() - 1); }
    T pop(){
        T ret = peek();
        data.remove(data.size() -1);
        return ret;
    }
}
```

Linked List Stack

- Let's assume we're using a singly linked list stack
- We could add elements at the head or the tail... which should we pick?
- Adding to either end is O(1) if we track the head/tail
- Removing from tail from a SLL is O(N) because we can't go backward to the node before the tail
- Removing from head is O(1)
- Let's make the head of our linked list the top of the stack

Comparison to ArrayList version

- For both implementations, all operations are O(1) (sort of)
- The arraylist version will sometimes be O(N) when we need to double the array and copy elements over, but on average will be O(1)
- The LinkedList version will have all pushes take about the same amount of time
- But each LinkedList push requires a heap allocation, while most ArrayList pushes only require incrementing the size integer
- The LinkedList version will take more memory since each node stores an extra reference

Fun stack trick: Bracket matching

- We can check if the parens, brackets, and braces of a string are matched using a stack

```
stack = empty stack
for each character:
    if it's one of "([{", push it on the stack
    if it's one of ")]}":
        if it matches the top of the stack, pop the stack
        else: it's a mismatched bracket
    ignore any other characters
if the stack isn't empty, we have a mismatched bracket
```

Fun stack trick: “postfix expression evaluation” aka “reverse notation”

- The math “grammar” you learned in school is called “infix notation” where we put operators between arguments like 3 + 4
- Infix notation is ambiguous (3 + 4 * 5) unless we specify rules for grouping operations
- Postfix notation puts operators after the arguments and doesn't require parentheses or order of operation rules
- `3 4 + 5 *` is equivalent to the infix expression `(3 + 4) * 5`

Postfix implmentation with a stack

```
stack = empty stack
for each token in the expression
    if it's an argument: push it on the stack
    if it's an operator:
        right = stack.pop
        left = stack.pop
        result = evaluate left operator right
        push(result)
top of (only thing in) stack is the value of the expression
```

Exercise

Evaluate `1 2 3 2 ^ * +`

Queue ADT

```
public interface Queue {
    void enqueue(T t);
    T dequeue();
    T peek();
}
```

- A Queue is what british people call a “line”
- Items are added at one end and removed from the other
- Also known as “first in first out” FIFO

Queue Implementations

- Like a stack, we have a collection where the order that elements are added matters, so we need a list based data structure
- The linked list implementation is a bit simpler so we'll start there

Linked List Queue

- If we use a singly linked list, we want to add to an end that's fast to add to, and remove from an end that's fast to remove from
- Adding to the end is O(1) if we store a tail pointer
- Removing from head is O(1)
- So a linked list queue enqueues at tail, dequeues from head
- If we have a double linked list, adding/removing from either end is equally efficient

Array Based Queue

- There's no easy solution here like there was for a stack
- We either have to add to or remove from the front which is O(N)
- To be efficient, we'll have to track which elements of the array are in use ourselves instead of relying on the ArrayList itself to do all the work
- Basically we need to track the front and back of the queue
- I find it simplest to store the front index and the queue size, but you can store front and back indices too
- We can “wrap around” when the back gets to the end of the array, so this is known as a “circular array queue”

Array Based Queue pseudocode

```
T[] data = new T[whatever];
int front = 0, size = 0;

void enqueue(T t){
    //what if we run out of space?
    data[(front + size)%data.length] = t;
    size++;
}
T dequeue(){
    ret = data[front];
    size--
    front = (front + 1) % arr.size
}
```

What do we do when the array fills up?

- For us: probably double it and copy into the new array
- For some applications: let the code calling enqueue that the queue is full and make it try again later

Array vs Linked List

- Array version is probably faster because it only has to allocate new arrays rarely
- Linked list operations take consistent time, array will have occasional very slow enqueue operations

Summary

- Stacks and Queues are ADTs that can be implemented with either linked lists or arrays
- All operations are O(1) for both implementations
- There's a related ADT called a “dequeue” which stand for “double ended queue” which lets you add or remove from either end, which is sometimes useful (addFront, removeFront, addBack, removeBack). How would we implement a dequeue efficiently?