

Introduction to Linked Lists, Stacks and Queues

Mahdi Ebrahimi

Data structures

Ways to organize information to enable
efficient computation over that information

A data structure supports certain *operations*, each with a:

- Meaning: what does the operation do/return
- Performance: how efficient is the operation

Examples:

- **List** with operations **insert** and **delete**
- **Stack** with operations **push** and **pop**

Trade-offs

A data structure strives to provide many useful, efficient operations

But there are unavoidable trade-offs:

- Time vs. space
- One operation more efficient if another less efficient
- Generality vs. simplicity vs. performance

We ask ourselves questions like:

- Does this support the operations I need efficiently?
- Will it be easy to use (and reuse), implement, and debug?
- What assumptions am I making about how my software will be used? (E.g., more lookups or more inserts?)

Terminology

- **Abstract Data Type (ADT)**
 - Mathematical description of a “thing” with set of operations
- **Algorithm**
 - A high level, language-independent description of a step-by-step process
- **Data structure**
 - A specific organization of data and family of algorithms for implementing an ADT
- **Implementation** of a data structure
 - A specific implementation in a specific language

Example: Stacks

- The **Stack** ADT supports operations:
 - **isEmpty**: have there been same number of pops as pushes
 - **push**: takes an item
 - **pop**: raises an error if empty, else returns most-recently pushed item not yet returned by a pop
 - ... (possibly more operations)
- A Stack data structure could use a linked-list or an array or something else, and associated algorithms for the operations
- One implementation is in the library `java.util.Stack`

Linked Lists

- Dynamic data structures
- Singly linked lists
- Generic linked lists
- Doubly linked lists

Array and Linked List

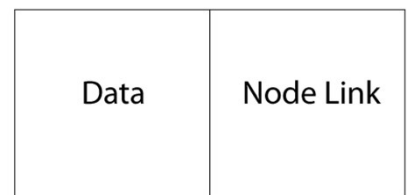
- An **array** is a *random access* data structure, where each element can be accessed directly and in constant time. A typical illustration of random access is a book - each page of the book can be open independently of others. Random access is critical to many algorithms, for example binary search.
- A **linked list** is a *sequential access* data structure, where each element can be accessed only in particular order. A typical illustration of sequential access is a roll of paper or tape - all prior material must be unrolled in order to get to data you want.

Linked List Data Structures

The core concept in a linked data structures is the node.

- A data structure is a collection of nodes linked in a particular way. A
- node holds a data item and links to other nodes.
- The nature of the links determines the kind of data structure, e.g., singly linked list, doubly linked list, binary tree, etc.

Here is a depiction of a node for a singly linked list and code that implements such a node (public members for brevity)



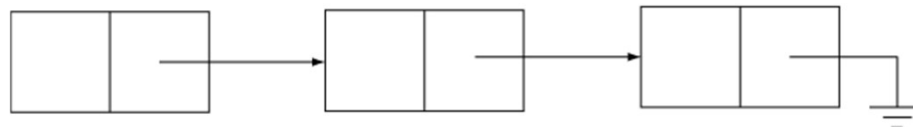
```
private class Node {  
    public Object data;  
    public Node next;  
  
    public Node(Object data, Node next) {  
        this.data = data;  
        this.next = next;  
    }  
}
```


Singly Linked Lists

A singly linked list

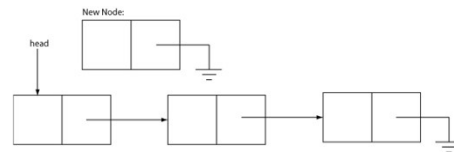
- Contains a pointer to a “head” node (`null` if empty).
- The head node’s `next` points to the second node, the second node’s `next` points to the third node, and so on.
- The `next` reference of the last node is `null`

Here’s a depiction of the nodes in a singly linked list with three elements:

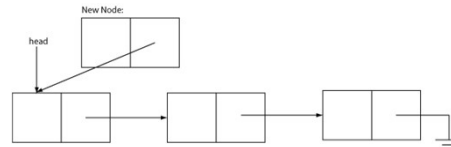


Adding Elements to a Singly Linked List

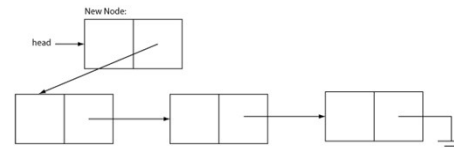
1. Create a new Node.



2. Set the next reference of the new Node to the current head.



3. Set the head reference to the new Node



Finding an Item in a Linked List

An algorithm for finding an item in a linked list:

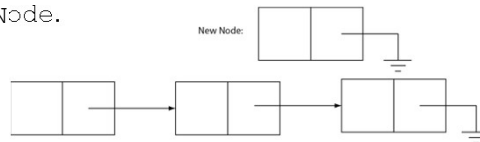
```
foundNode: Node := null
currentNode: Node := LinkedList.head
while currentNode != null && foundNode = null
    if currentNode.data = queryItem
        foundNode := currentNode
    currentNode := currentNode.next
```

The postcondition of this algorithm is that `foundNode` will be:

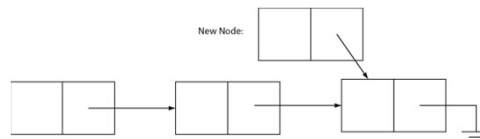
- The node containing the query item, or
- `null` if the query item is not in the list.

Inserting Elements into a Linked List

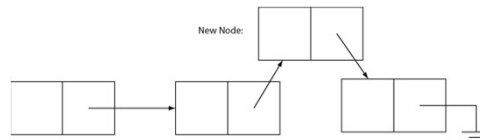
1. Find the existing Node to insert new element after.
2. Create a new Node.



3. Set the next reference of the new Node to the next reference of the existing node.



4. Set the next reference of the existing node to the new Node.



Computing the Length of a Linked List

An algorithm for computing the length a linked list:

```
length: int := 0
currentNode: Node := LinkedList.head
while currentNode != null
  length := length + 1
  currentNode := currentNode.next
```

The postcondition of this algorithm is that `length` will be equal to the number of nodes in the list.

Generic Linked Lists

To make our `LinkedList` generic we only need to add a type parameter to the declaration:

```
public class GenericLinkedList<E> { ...
```

and replace `Object` with `E` in the body of the class.

Doubly Linked Lists

A doubly linked list simply adds a `previous` reference to the `Node` class so that the list can be traversed forwards or backwards.

```
private class Node<E> {  
    E data;  
    Node<E> next;  
    Node<E> previous;  
  
    public Node(E data, Node<E> next, Node<E> previous) {  
        this.data = data;  
        this.next = next;  
        this.previous = previous;  
    }  
}
```

Doubly linked lists work the same, except that the algorithms for inserting and removing items requires a bit more link fiddling (have to set previous links as well).

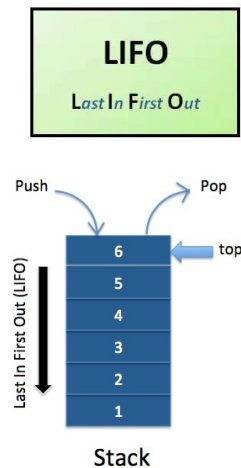
Running times of List operations

TABLE 24.1 Time Complexities for Methods in `MyArrayList` and `MyLinkedList`

Methods	<i>MyArrayList/ArrayList</i>	<i>MyLinkedList/LinkedList</i>
<code>add(e: E)</code>	$O(1)$	$O(1)$
<code>add(index: int, e: E)</code>	$O(n)$	$O(n)$
<code>clear()</code>	$O(1)$	$O(1)$
<code>contains(e: E)</code>	$O(n)$	$O(n)$
<code>get(index: int)</code>	$O(1)$	$O(n)$
<code>indexOf(e: E)</code>	$O(n)$	$O(n)$
<code>isEmpty()</code>	$O(1)$	$O(1)$
<code>lastIndexOf(e: E)</code>	$O(n)$	$O(n)$
<code>remove(e: E)</code>	$O(n)$	$O(n)$
<code>size()</code>	$O(1)$	$O(1)$
<code>remove(index: int)</code>	$O(n)$	$O(n)$
<code>set(index: int, e: E)</code>	$O(n)$	$O(n)$
<code>addFirst(e: E)</code>	$O(n)$	$O(1)$
<code>removeFirst()</code>	$O(n)$	$O(1)$

Stacks

- A stack is a container of objects that are inserted and removed according to the **last-in first-out (LIFO)** principle.
- In the pushdown stacks only two operations are allowed: **push** the item into the stack, and **pop** the item out of the stack.
- A stack is a limited access data structure - elements can be added and removed from the stack only at the **top**. push adds an item to the top of the stack, pop removes the item from the top.
- A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.
- A stack is a recursive data structure. Here is a structural definition of a Stack:
 - a stack is either empty or
 - it consists of a top and the rest which is a stack;

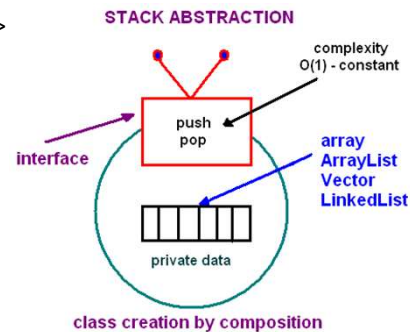


Stacks Implementation

The structure for a stack could be an array, a vector, an ArrayList, a linked list, or any other collection.








Regardless of the type of the underlying data structure, a Stack must implement the same functionality. This is achieved by providing a unique interface:

```
public interface StackInterface <AnyType>
{
    public void push(AnyType e);
    public AnyType pop();
    public AnyType peek();
    public boolean isEmpty();
}
```



Stacks Implementation

- **push():** Adds an item into (top of) the stack
- **pop():** Removes the item from the top of the stack.
- **peek():** Retrieves the first element of the Stack or the element present at the top of the *Stack*.
- **empty():** Check if a stack is empty or not.
- **search(), min(), ...**

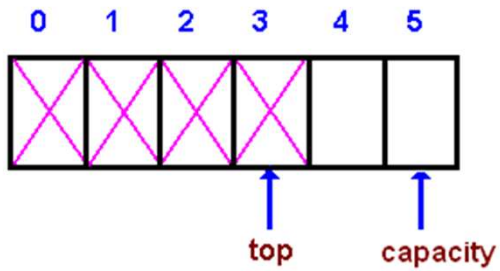
 Stack<E> java.util
 <u>serialVersionUID: long</u>
 Stack()  push(E):E  pop():E  peek():E  empty():boolean

Stacks Implementation - Array-based

In an array-based implementation we maintain the following fields:
an array A of a default size (≥ 1), the variable `top` that refers to the top element in the stack
and the `capacity` that refers to the array size.

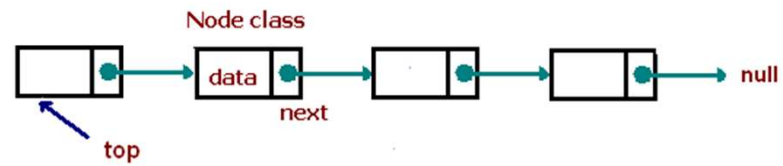
The variable `top` changes from -1 to `capacity - 1`.

We say that a stack is empty when `top = -1`, and the stack is full when `top = capacity-1`.



Stacks Implementation - Linked List-based

Linked List-based implementation provides the best (from the efficiency point of view) dynamic stack implementation.



Stacks Applications

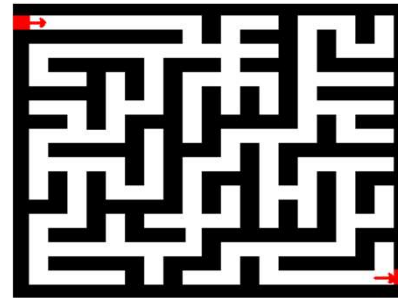
- ✓ The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
- ✓ Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.



Stacks Applications

Backtracking. This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit?

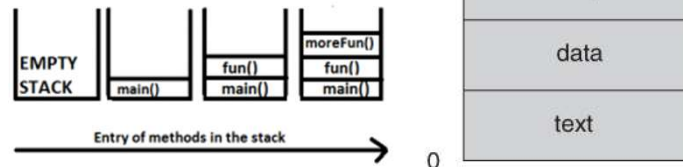
Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.



Stacks Applications

Language processing:

- Space for parameters and local variables is created internally using a stack.
- Compiler's syntax check for matching braces is implemented by using stack.

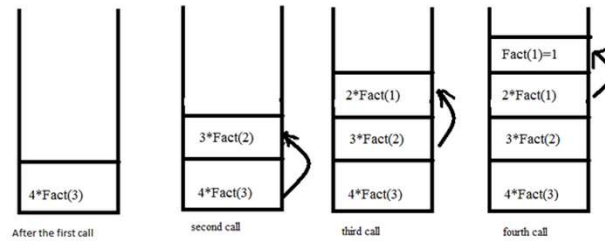


Stacks Applications

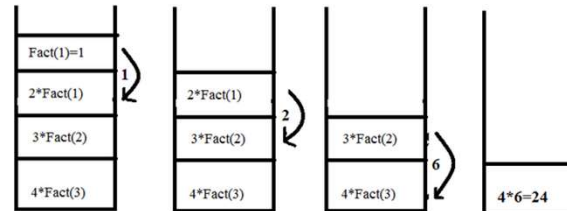
Language processing:

- Support for recursion

When function call happens previous variables gets stored in stack

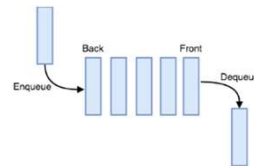


Returning values from base case to caller function



Queues

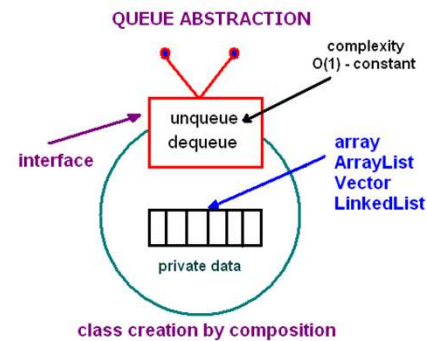
- A queue is a container of objects (a linear collection) that are inserted and removed according to the first-in first-out (FIFO) principle. An excellent example of a queue is a line of students in the food court of the CSUN. New additions to a line made to the back of the queue, while removal (or serving) happens in the front. In the queue only two operations are allowed **enqueue** and **dequeue**.
- Enqueue means to insert an item into the back of the queue, dequeue means removing the front item. The picture demonstrates the FIFO access.
- The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.



Queues Implementation

The structure for a queue could be an array, a Vector, an ArrayList, a LinkedList, or any other collection.
Regardless of the type of the underlying data structure, a queue must implement the same functionality. This is achieved by providing a unique interface.

```
interface QueueInterface <AnyType>
{
    public boolean isEmpty();
    public AnyType getFront();
    public AnyType dequeue();
    public void enqueue(AnyType e);
    public void clear();
}
```

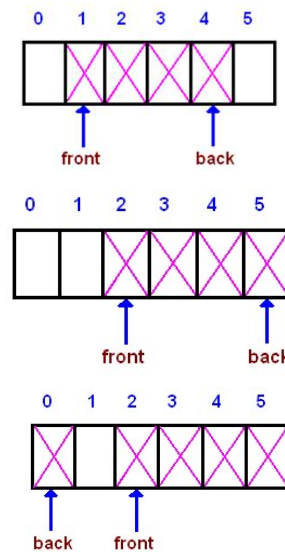


Circular Queues

Given an array A of a default size (≥ 1) with two references *back* and *front*, originally set to -1 and 0 respectively. Each time we insert (enqueue) a new item, we increase the back index; when we remove (dequeue) an item - we increase the front index.

As you see from the picture, the queue logically moves in the array from left to right. After several moves back reaches the end, leaving no space for adding new elements.

However, there is a free space before the front index. We shall use that space for enqueueing new items, i.e. the next entry will be stored at index 0, then 1, until *front*. Such a model is called a **wrap around queue** or a **circular queue**.

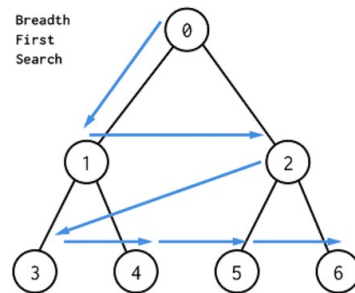


Stacks/Queues Applications – search tree

Breadth-First Search with a Queue

In breadth-first search we explore all the nearest possibilities by finding all possible successors and enqueue them to a queue.

1. Create a **queue**
2. Create a new choice point
3. Enqueue the choice point onto the queue
4. while (not found and queue is not empty)
 1. Dequeue the queue
 2. Find all possible choices after the last one tried
 3. Enqueue these choices onto the queue
5. Return



Depth-First Search with a Stack

In depth-first search we go down a path until we get to a dead end; then we *backtrack* or back up (by popping a stack) to get an alternative path.

1. Create a **stack**
2. Create a new choice point
3. Push the choice point onto the stack
4. while (not found and stack is not empty)
 1. Pop the stack
 2. Find all possible choices after the last one tried
 3. Push these choices onto the stack
5. Return

