# CS 5010: Programming Design Paradigms
## Fall 2017

## Lecture 7: Data Structures and Algorithms

Acknowledgement: lecture notes inspired by course material prepared by UW faculty members Z. Fung and H. Perkins.

Tamara Bonaci
t.bonaci@northeastern.edu

# Administrivia

- No code walk this week (grades for your assignments coming soon)

- Assignment 4 due on Monday, October 23by 6pm
- Code walkthroughs on Tuesday, October 24 from 10am-9pm, in 401 Terry Ave N:
  - 10am – 5:30pm in Camano classroom
  - 6pm – 10pm in Cypress classroom

# Agenda – Algorithms and Data Structures 1

- Data collections
- List ADT
  - Doubly-linked List
  - Algorithm: Recursion
- Stack ADT
  - Some implementation details
  - Algorithm: backtracking
- Queue ADT
  - Mixing stacks and queues
  - Circular queues
- Trees
  - Binary trees
- Tree Traversals
- Some comments and hints about Assignment 4

Algorithms and Data Structures 1

# DATA COLLECTIONS

# Data Collections?

Collection of chewed gums

Collection of pens

Collection of cassette tapes

Collection of old radios

# What is a data collection?

Shoes collection

Star wars collection

Cars collection

[Pictures credit: http://www.smosh.com/smosh-pit/articles/19-epic-collections-strange-things ]

# Data Collections?

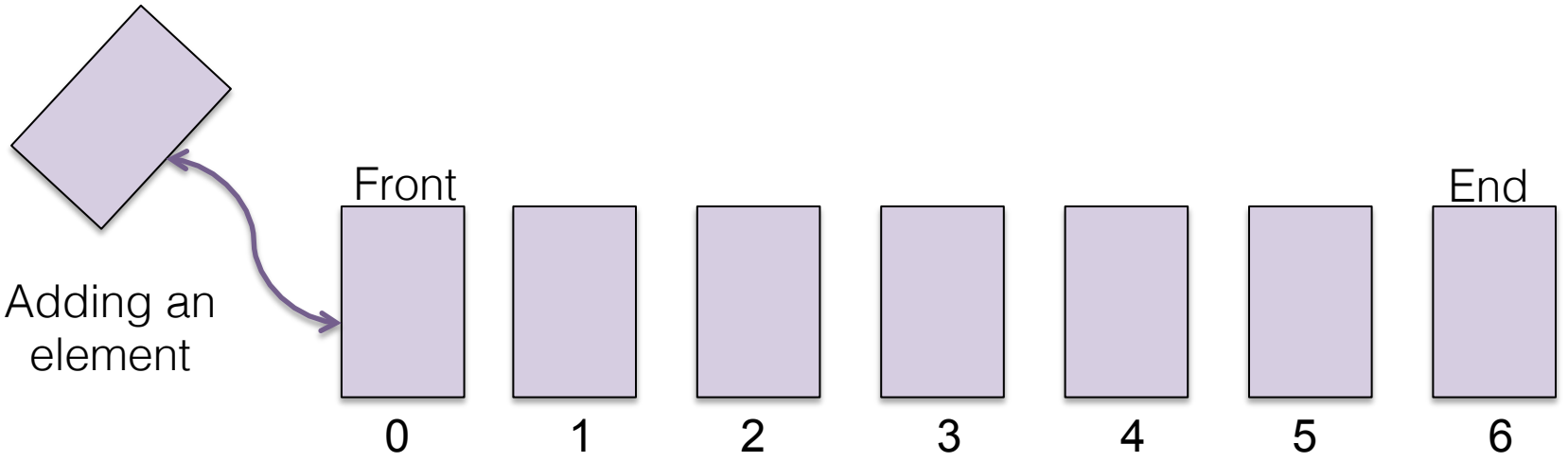
- Data collection - an object used to store data (think *data structures*)
  - Stored objects called elements
  - Some typically operations:
    - `add()`
    - `remove()`
    - `clear()`
    - `size()`
    - `contains()`
- Some examples: ArrayList, LinkedList, Stack, Queue, Maps, Sets, Trees

**Why do we need different data collections?**

# Example: ArrayList vs. LinkedList

- List - a collection of elements with 0-based **indexes**
  - Elements can be added to the front, back, or in the middle
  - Can be implemented as an ArrayList or as a LinkedList

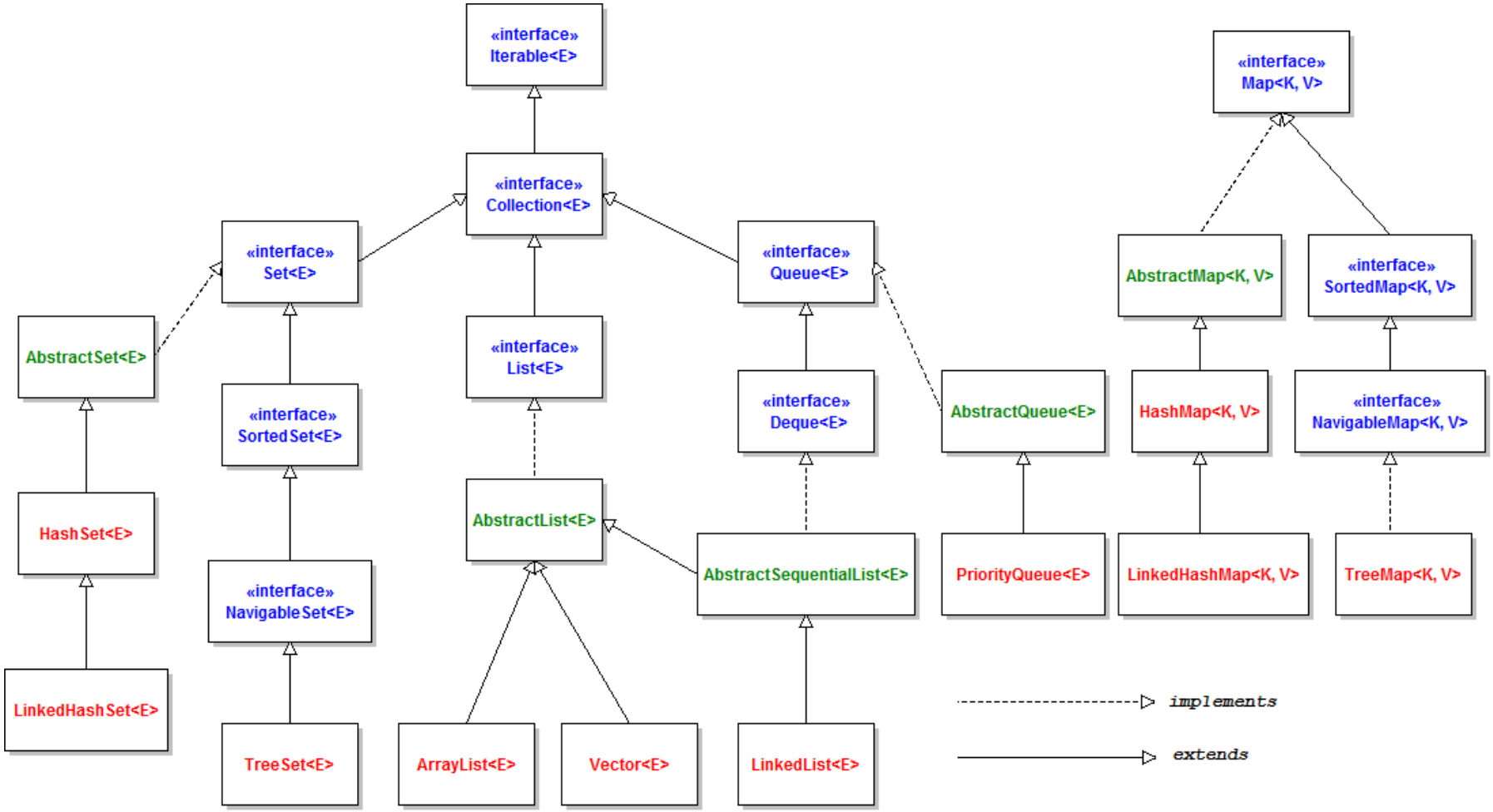
  - What is the complexity of adding an element to the front of an:
    - ArrayList?
    - LinkedList?



Algorithms and Data Structures 1

# JAVA COLLECTIONS

# Java Collections API




[Pictures credit: http://www.codejava.net]

# Java Collections API

- In Java, common data structures implemented in Java Collections API
  - Part of the `java.util` package

- Interface `Collection<E>`:
  - Extended by four interfaces:
    - `List<E>`
    - `Set<E>`
    - `Queue<E>`
    - `Map<K,V>`
  - Extends interface `Iterable<T>`

# Interfaces `Iterable<T>` and `Iterator<E>`

- Implementing interface `Iterable<T>` allows an object to be traversed using the `for each` loop
- Every object that implements Iterable<T> must provide a method `Iterator iterator()`

- Interface `Iterator` – an iterator over a collection

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove():
}
```

- Iterator `remove()` method – removes the last item returned by method `next()`

# Direct Use of an `Iterator<E>`

- Interface `Iterator` – an iterator over a collection

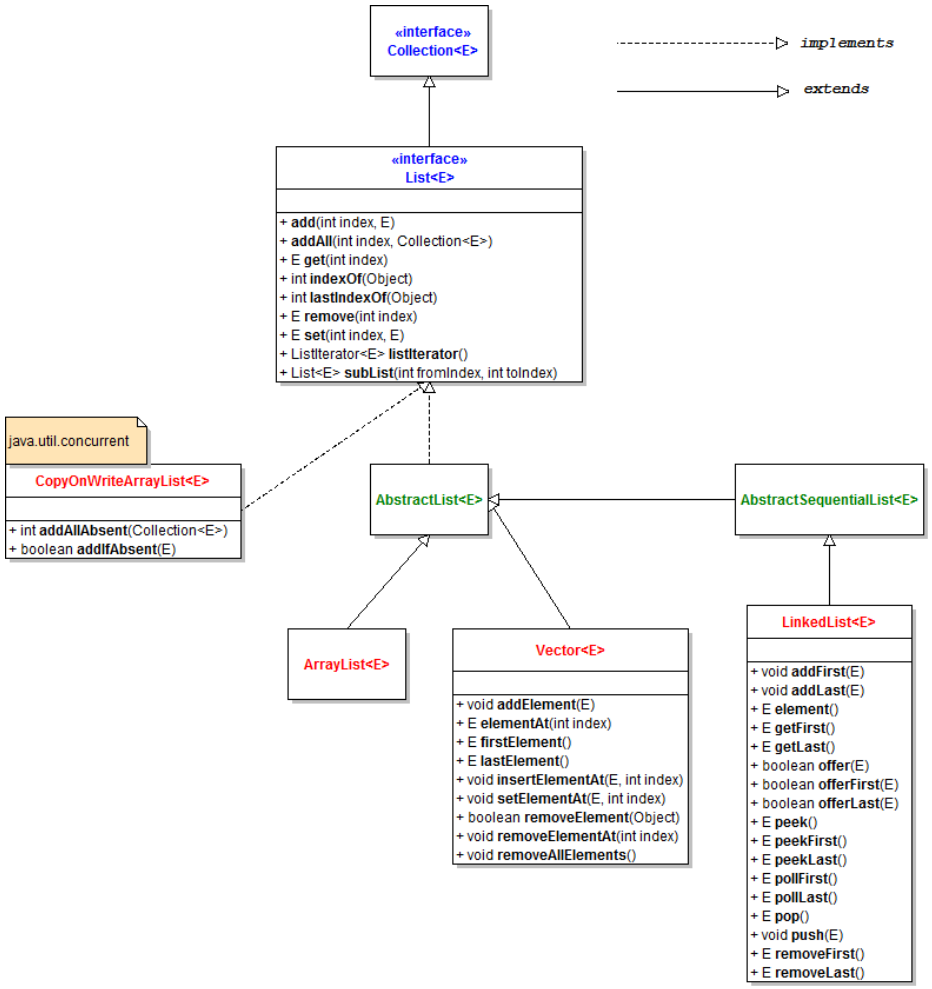
```
public interface Iterator<E> {
   boolean hasNext();
   E next();
   void remove():
}
```

- Careful when an iterator is used directly (not via a for each loop)→ if you make any structural changes to a collection being iterated (`add, remove, clear`), the iterator is no longer valid (`ConcurrentModificationException thrown`)

Algorithms and Data Structures 1

# LIST ADT

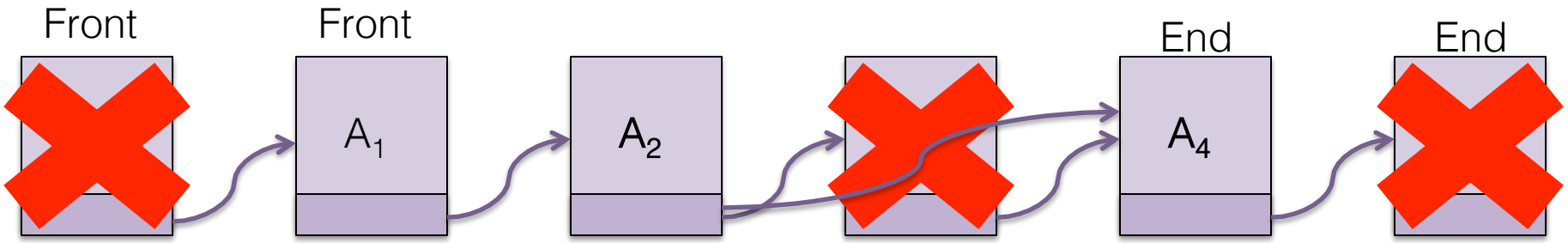
# Java List API




[Pictures credit: http://www.codejava.net]

- `List<E>` - the base interface
- Abstract subclasses:
  - `AbstractList<E>`
  - `AbstractSequentialList<E>`
- Concrete classes:
  - `ArrayList<E>`
  - `LinkedList<E>`
  - `Vector<E>` (legacy collection)
  - `CopyOnWriteArrayList<E>` (class under java.util.concurrent package)
- Main methods:
  - `E get(int index);`
  - `E set(int index, E newValue);`
  - `Void add(int index, E x);`
  - `Void remove(int index);`
  - `ListIterator<E> listIterator();`

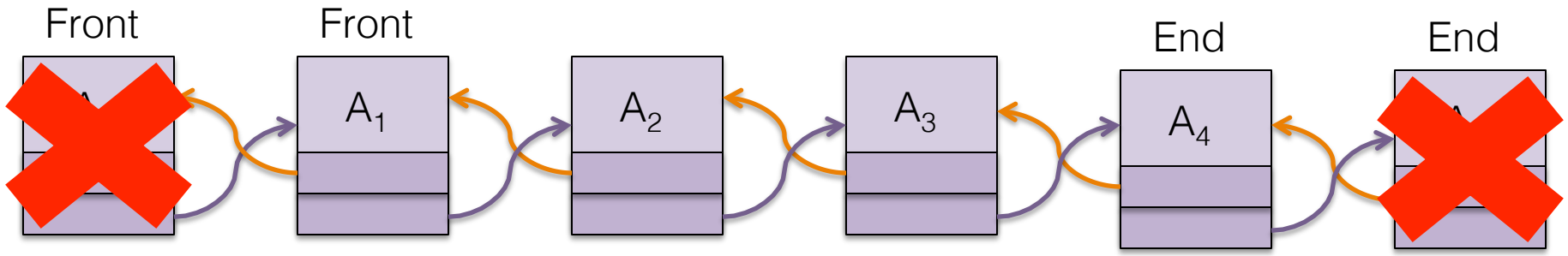
# Example: Removing Elements from a LinkedList




- Remove the first element of the list
- Remove element $A_3$ from the list
- Remove the last element in the list

- What's the tricky part about removing elements $A_3$ and $A_5$?
- Having to find their predecessors (elements $A_2$ and $A_4$) and updating their link to last node
- Idea: every node maintains the link to its previous and its next node → doubly linked list

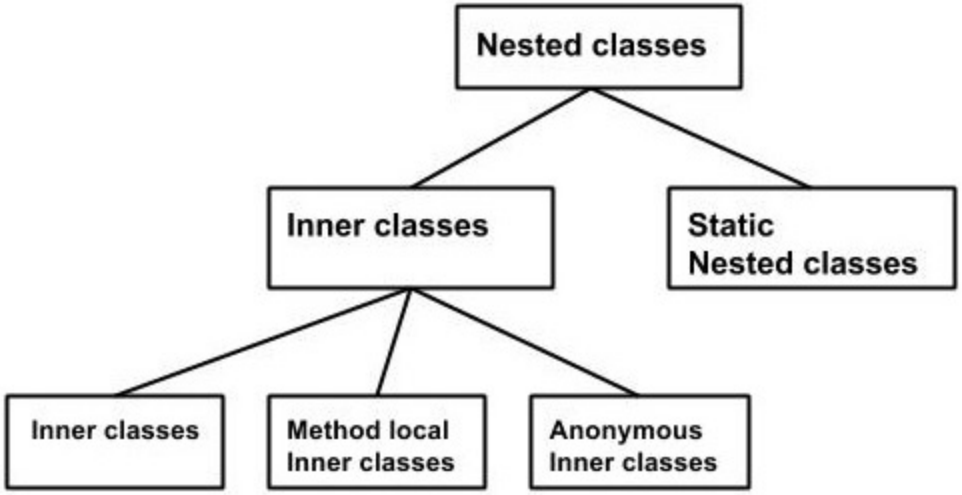
# Doubly LinkedList




- Removing the first element of the list
- Removing the last element of the list

# Implementation of a Doubly LinkedList

- Review – inner and nested classes:



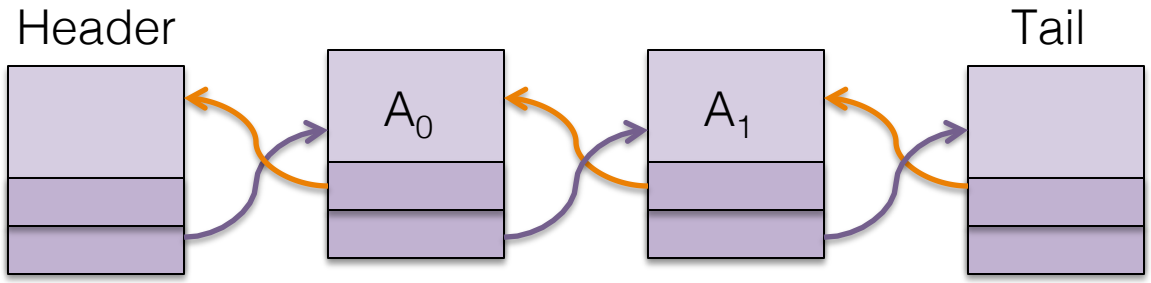

[Pictures credit: https://www.tutorialspoint.com/java/images/inner_classes.jpg]

- Java allows us to define a class *within* another class – a nested class
  - Static nested classes
  - Non-static nested classes or inner classes – have access to other members of the enclosing class, even if they are declared private

# Implementation of a Doubly LinkedList

- Doubly linked list – need to provide and maintain links to both ends of the list
- Implemented classes:
  - Class `MyLinkedList`
  - Class `Node` – private nested class, contains the data and links to previous and next nodes
  - Class `LinkedListIterator` – private inner class, implementing the interface `Iterator`

- Sentinel nodes:
  - Header and tail nodes, used to logically represent the beginning and the end markers



# Implementation of a Doubly LinkedList: MyLinkedList

```
public class MyLinkedList<E> implements Iterable<E>{
private int listSize;
private int modCount = 0;
private Node<E> beginMarker;
private Node<E> endMarker;

public MyLinkedList(){
    doClear();
}
public int size(){
    return listSize;
}
public boolean isEmpty(){
    return size()==0;
}
public boolean add(E x){
    add(size(), x);
    return true;
}
public void add(int index, E x){
    addBefore(getNode(index, 0, size()), x);
}
```

# Implementation of a Doubly LinkedList: MyLinkedList

```
public E get(int index){
    return getNode(index).data;
}
public E set(int index, E newVal){
    Node<E> p = getNode(index);
    E oldVal = p.data;
    p.data = newVal;
    return oldVal;
}
public E remove(int index){
    return remove(getNode(index));
}
public java.util.Iterator<E> iterator(){
    return new LinkedListIterator();
}
```

# Implementation of a Doubly LinkedList: `Node`

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

    public Node(E d, Node<E> p, Node<E> n){
        data = d;
        prev = p;
        next = n;
    }
}
```

# Implementation of a Doubly LinkedList: `clear()`

```
public void clear(){
    doClear();
}

Private void doClear(){
    listSize = 0;
    modCount ++;

    beginMarker = new Node<E>(null, null, null);
    endMarker = new Node<E>(null, beginMarker, null);
    beginMarker.next = endMarker;
}
```

# Implementation of a Doubly LinkedList: `addBefore()`

```
/**
*Adds an item to this collection, at specified position p.
* Items at or after that position are slid one position higher.
* @param p Node to add before.
* @param x any object.
*/
private void addBefore(Node<E> p, AnyType x){
    Node<E> newNode = new Node<>(x, p.prev, p);
    p.prev = newNode;
    listSize++;
    modCount++;
}
```

# Implementation of a Doubly LinkedList: `remove()`

```
/**
* Removes the object contained in Node p.
* @param p the Node containing the object.
* @return the item that was removed from the collection.
*/
private E remove( Node<E> p ) {
    p.next.prev = p.prev;
    p.prev.next = p.next;
    listSize--;
    modCount++;
    return p.data;

}
```

# Implementation of a Doubly LinkedList: `getNode()`

```
/**
* Gets the Node at position index, which must be in range 0
to size()-1.
* @param index index to search at.
* @return internal node corresponding to index
* @throws IndexOutOfBoundsException if index is not between
0 and size()-1, inclusive.
*/

private Node<E> getNode(int index){
    return getNode(index, 0, size()-1);
}
```

# Implementation of a Doubly LinkedList: `getNode()`

```
/**
* Gets the Node at position index, which must be in range
from lower to upper.
* @param index index to search at.
* @param lower lowest valid index.
* @param upper highest valid index.
* @return internal node corresponding to index
* @throws IndexOutOfBoundsException if index is not between
lower and upper, inclusive.
*/

private Node<E> getNode(int index, int lower, int upper){
    Node<E> p;

    if(index < lower || index > upper)
        throw new IndexOutOfBoundsException();
}
```

# Implementation of a Doubly LinkedList: `getNode()`

```
if(index < size()/2){
    p = beginMarker.next;
    for(int i = 0; i < index; i++)
        p = p.next;
}else{
    p = endMarker;
    for(int i = 0; i > index; i--)
        p = p.prev;
}
return p;
}
```

# Implementation of a Doubly LinkedList: `LinkedListIterator()`

```
private class LinkedListIterator implements java.util.Iterator<E>{
    private Node<E> current = beginMarker.next;
    private int expectedModCount = modCount;
    private boolean okToRemove = false;

    public boolean hasNext(){
        return current != endMarker;
    }
    public AnyType next(){
        if(modCount != expectedModCount)
            throw new java.util.ConcurrentModificationException();
        if(!hasNext())
            throw new java.util.NoSuchElementException();

        E nextItem = current.data;
        current = current.next;
        okToRemove = true;
        return nextItem;
    }
```

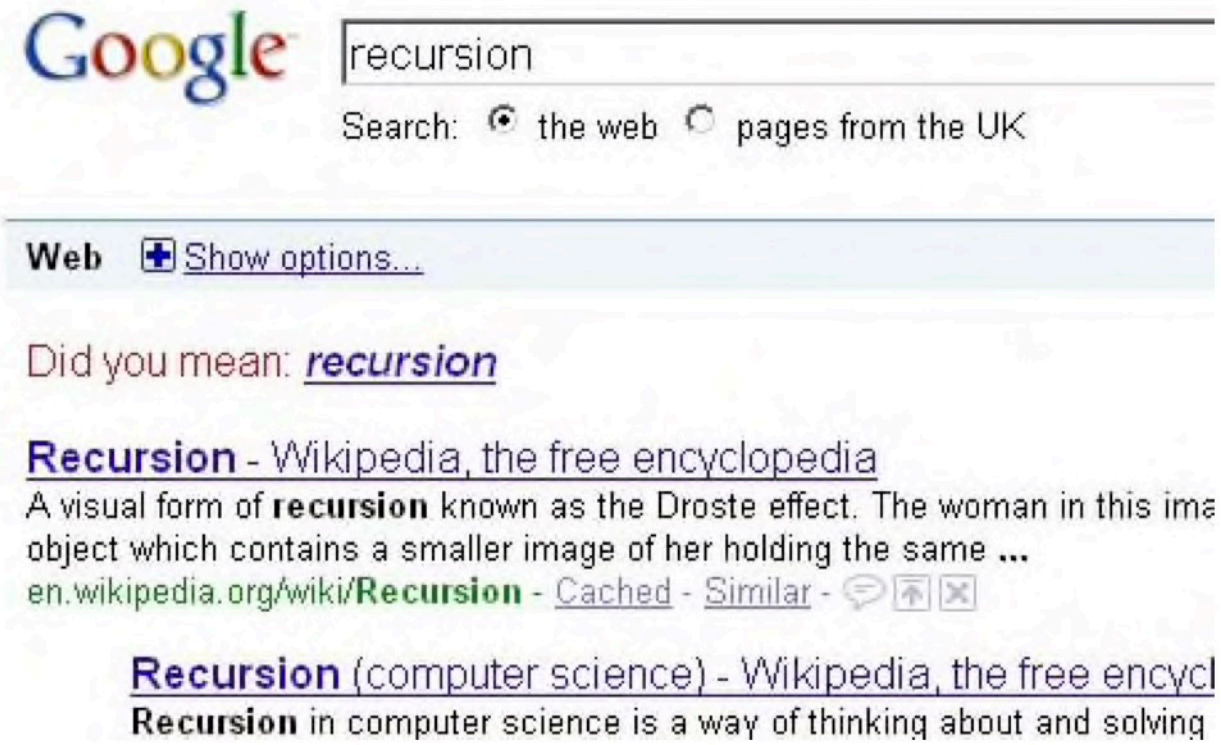
# Implementation of a Doubly LinkedList: `LinkedListIterator()`

```
private class LinkedListIterator implements java.util.Iterator<E>{

    public void remove(){
        if(modCount != expectedModCount)
            throw new java.util.ConcurrentModificationException();
        if(!okToRemove())
            throw new IllegalStateException();

        MyLinkedList.false.remove(current.prev);
        expectedModCount++;
        okToRemove = false;
    }
}
```

# Recursion




[Pictures credit: http://www.telegraph.co.uk/technology/google/6201814/Google-easter-eggs-15-best-hidden-jokes.html]

# Recursion

- Recursion – an operation defined in terms of itself
  - Solving a problem recursively means solving smaller occurrences of the same problem
- Recursive programming – an object consist of methods that call themselves to solve some problem
- Can you think of some examples of recursions and recursive programs?

# Recursive Algorithm

- Every recursive algorithm consists of:
  - Base case – at least one simple occurrence of the problem that can be answered directly
  - Recursive case - more complex occurrence that cannot be directly answered, but can be described in terms of smaller occurrences of the same problem

- A crucial part of recursive programming is identifying these cases
- What were base cases for our Fibonacci problem in Assignment 1?
- Why are we now talking about recursion???

# Recursive Data Structures

- Recursive data structure - a data structure partially composed of smaller or simpler instances of the same data structure

- Is linked list a recursive data structure?
- Let's see - a linked list is either
  - Null (base case)
  - A node whose next field references a list

# Example: Printing Values in a Linked List

```
public static void printList (Node nodeToPrint){
   if(nodeToPrint == null)
      return;
   else{
   System.out.println(nodeToPrint.data);
   print(nodeToPrint.next);
   }
}
```

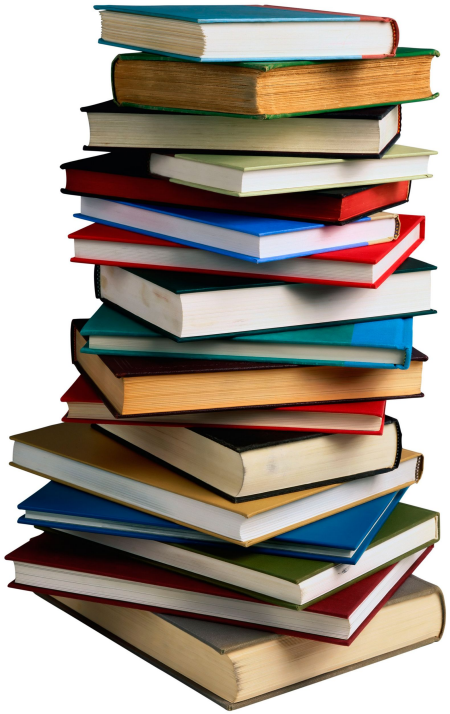
- Similar problems: print in the reverse order, sum elements of the list,...
- Careful – this is a tail recursion
  - Tail recursions are not bad, but they do consume the stack

Algorithms and Data Structures 1

# STACK ADT

# Stacks

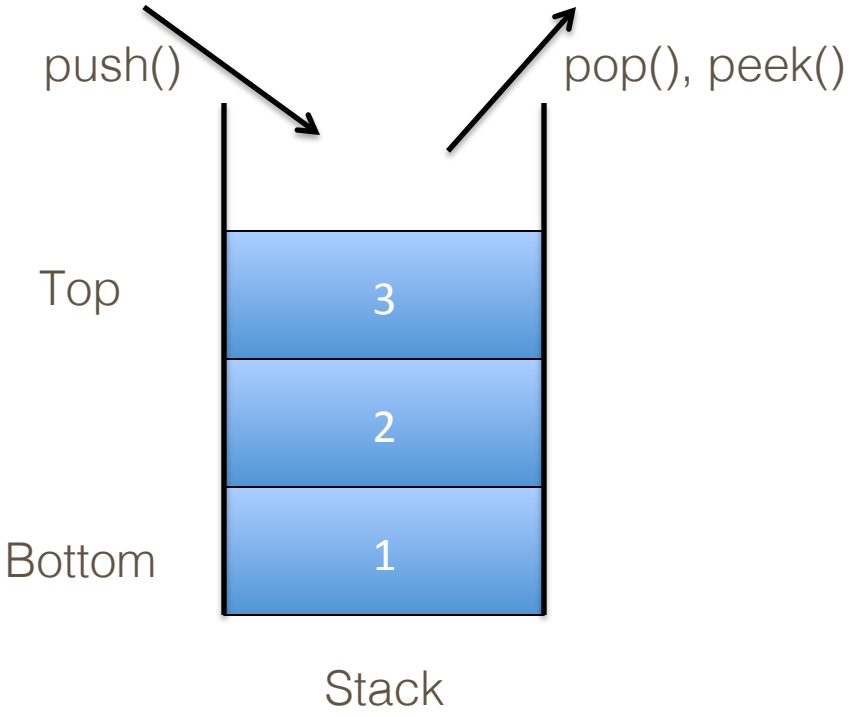
- Some of my favorite stacks:



[Pictures credit: https://rukminim1.flixcart.com/image/1408/1408/stacking-toy, http://battellemedia.com/wp-content/uploads/2014/08/National-Pancake-Day-at-IHOP.jpg, http://all4desktop.com/data_images/original/4245681-book.jpg]

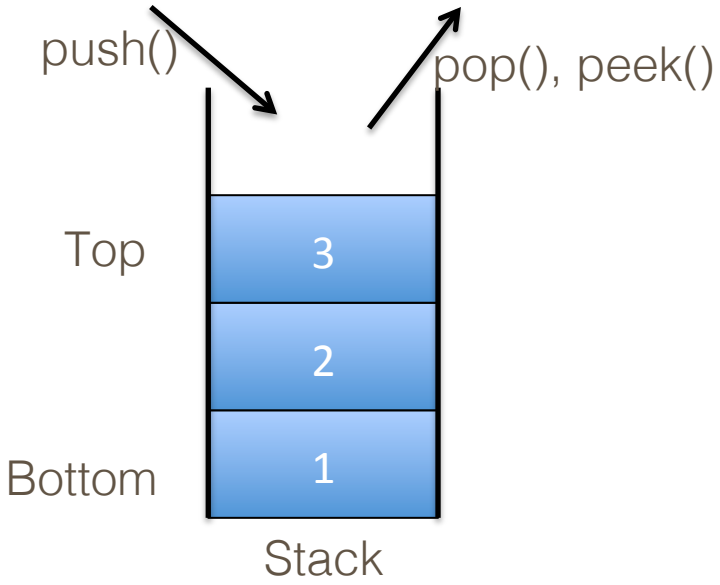
# What is a Stack?

- Stack – a data collection that retrieves elements in the LIFO order (last-in-first-out)




- Is there another way to think about a stack?

# What is a Stack?

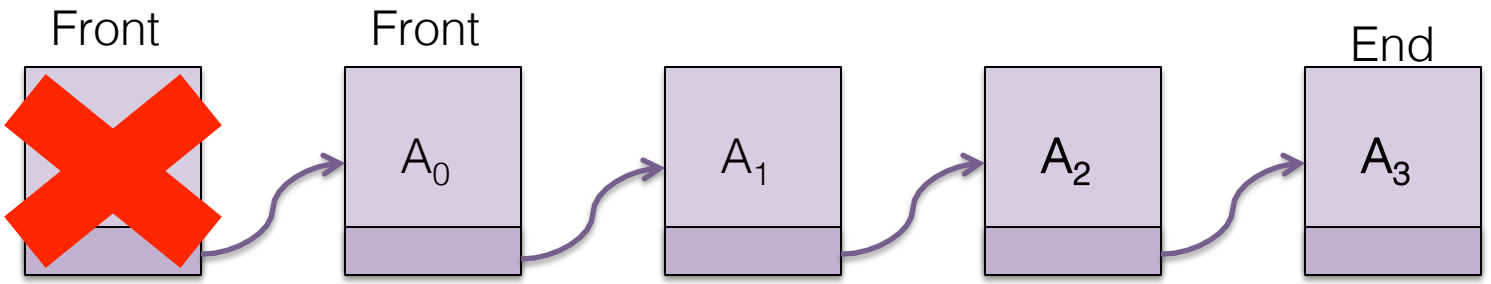



- Is there another way to think about a stack?
- Stack – a constrained data collection where clients are limited to use only limited optimized methods (`pop, push, peek`)
- Stack – a list with restriction that insertions and deletions can be performed in only one position, the end of the list, called the top

# Implementations of a Stack

- Stack – a list with restriction that insertions and deletions can be performed in only one position, the end of the list, called the top

- Since a stack is a list, any list implementation will do:
  - Example: linked list implementation
    - `push()`
    - `peek()/top()`
    - `pop()`



# Java Class `Stack`

| | |
|---|---|
| `Stack <E> ()` | Object constructor – constructs a new stack with elements of type E |
| `push(value)` | Places given value on top of the stack |
| `pop()` | Removes top value from the stack, and returns it. Throws `EmptyStackException` if the stack is empty. |
| `peek()` | Returns top value from the stack without removing it. Throws EmptyStackException if the stack is empty. |
| `size()` | Returns the number of elements on the stack. |
| `isEmpty()` | Returns true if the stack is empty. |

- Example:

```
Stack<String> s = new Stack<String>();
s.push("Hello");
s.push("PDP");
S.push("Fall 2017"); //bottom ["Hello", "PDP", "Fall 2017"] top
System.out.println(s.pop()); //Fall 2017
```

# Applications of a Stack

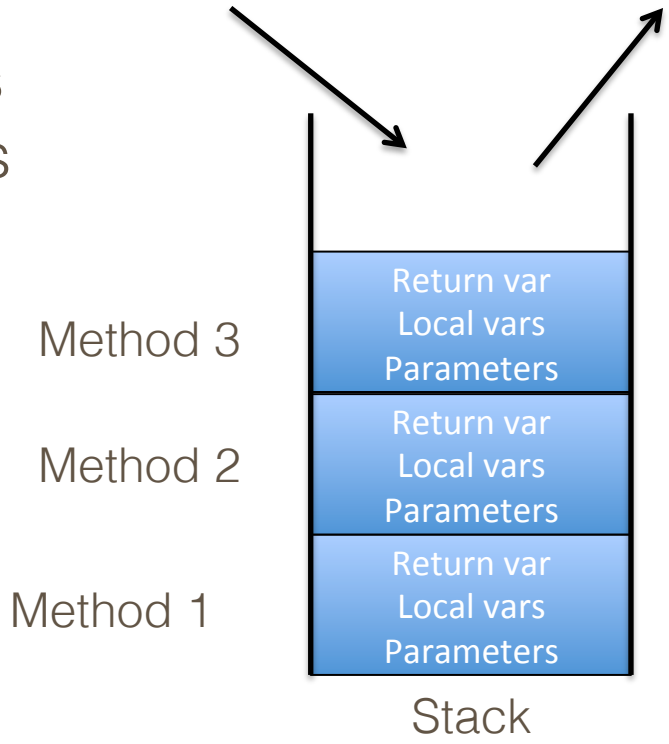
- Programming languages and compilers:
  - Method calls *(call=push, return=pop)*
  - Compilers (parsers)

- Matching up related pairs of things:
  - Find out whether a string is a palindrome
  - Examine a file to see if its braces { } match
  - Convert "infix" expressions to pre/postfix

- Sophisticated algorithms:
  - Searching through a maze with "backtracking"

# Example: Methods Call

- In some system whenever there is a method call, some information about the current state of the system needs to be stored before the control is transferred to a new method:
  - Parameters
  - Local variables
  - Return address



# Example: Postfix Expressions

- Suppose you are using a calculator to compute the total cost of your groceries
  - Add the costs of individual items
  - Multiply by 1.1 to account for local sales tax

- The natural way to do this with a calculator:

  5.5 + 4.5 + 7 + 8*1.1

- What is the expected result?
  - 27.5 (expected value)
  - 25.8

- That depends on how "smart" is your calculator!

# Example: Postfix Expressions

- The natural way to do this with a calculator:

  5.5 + 4.5 + 7 + 8*1.1

- What if we represent the given expression in the postfix or Reverse Polish notation:

  5.5 1.1 * 4.5 1.1*+ 7 1.1*+ 8 1.1*+

- The easiest way to implement this is with a stack:

  (5.5 * 1.1) + (4.5 * 1.1) + (7*1.1) + (8*1.1)

# Backtracking

- Backtracking – an algorithmic approach that finds solution(s) by trying partial solutions, and abandoning them as soon as they are no longer suitable

- A "brute force" algorithmic method
  - All possible solutions paths explored
  - Often implemented recursively

- Some applications:
  - Finding all permutations of a set of values
  - Parsers
  - Different games: sudoku, anagrams, crosswords, word jumbles, 8 queens

# Backtracking Strategies

- When solving a backtracking problem, we often ask:
  - What are the "choices" in this problem?
    - What is the "base case"?
    - How do I know when I'm out of choices?
  - How do I "make" a choice?
    - Do I need to create additional variables to remember my choices?
    - Do I need to modify the values of existing variables?
  - How do I explore the rest of the choices?
    - Do I need to remove the made choice from the list of choices?
  - Once I'm done exploring, what should I do?
  - How do I "un-make" a choice?

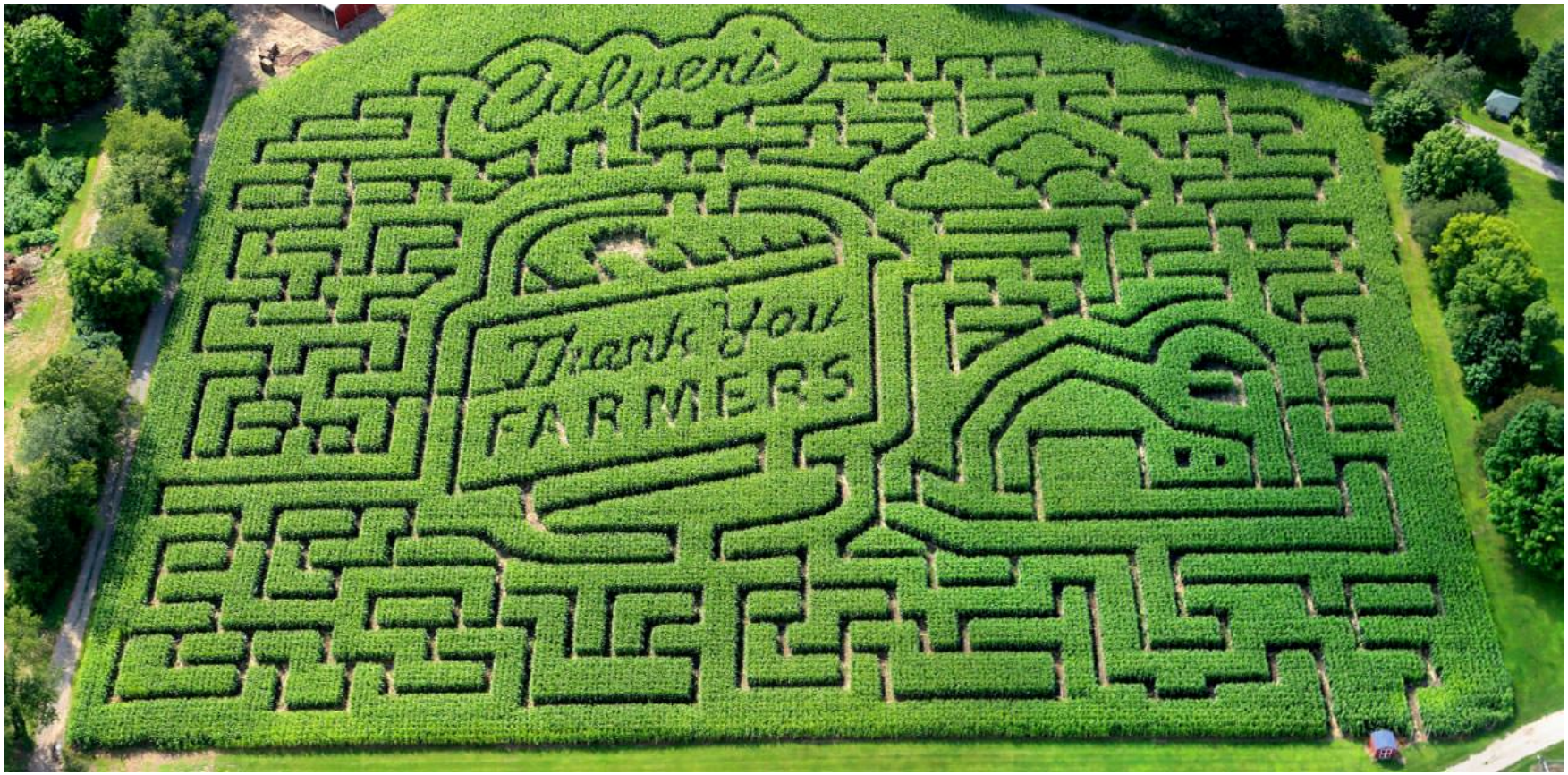
# Backtracking Algorithm

*A general pseudo-code algorithm for backtracking problems:*

```
Explore(choices):
    If there are no more choices to make:
        stop
    Else:
        Choose C.
        Explore the remaining choices.
        Un-choose C, if necessary. (backtrack!)
```

- Why are we talking about backtracking now?
  - Many backtracking problems can be implemented using a stack!

# Example: Maze




[Pictures credit: https://d1f28u9l1tudce.cloudfront.net/inside-culvers/tyf-story-corn-maze.jpg]

[Example credit: Zorah Fung, University of Washington]

# Example: Class `Maze`

- Let's assume that we have class `Maze` with the following methods:

| Method/Constructor | Description |
|---|---|
| `public Maze(String text)` | construct a given maze |
| `public int getHeight(), getWidth()` | get maze dimensions |
| `public boolean isExplored(int r, int c)`<br>`public void setExplored(int r, int c)` | get/set whether you have visited a location |
| `public void isWall(int r, int c)` | whether given location is blocked by a wall |
| `public void mark(int r, int c)`<br>`public void isMarked(int r, int c)` | whether given location is marked in a path |
| `public String toString()` | text display of maze |

[Example credit: Zorah Fung, University of Washington]

# Example: Solving the Maze

- Write a method `solveMaze` that accepts as parameters:
  - A maze
  - A starting row/column

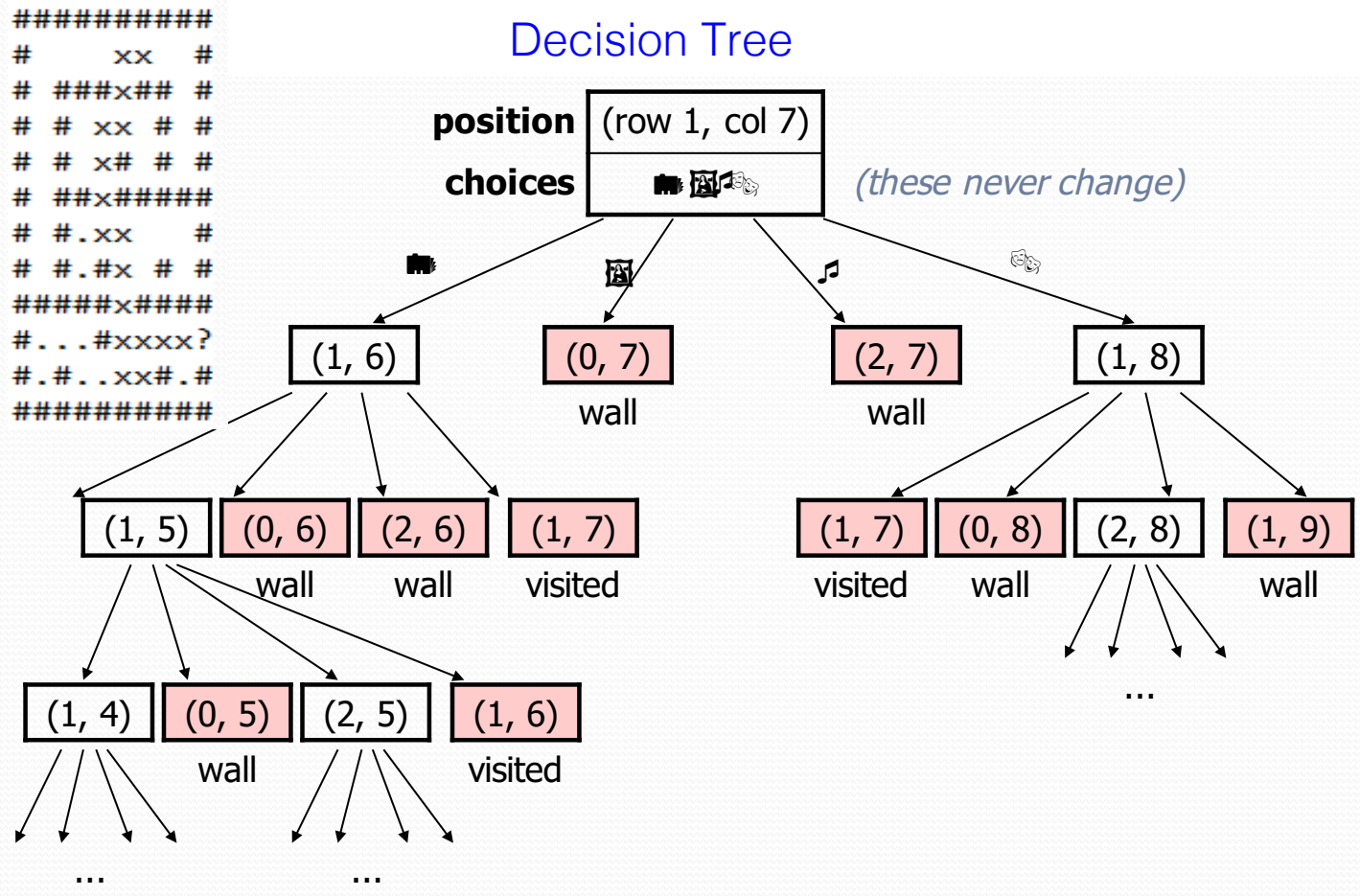
  and tries to find a path in out of the maze starting from that position

- If you find a solution:
  - Your code should **stop** exploring
  - You should **mark** the path out of the maze on your way back out of the recursion, using backtracking
  - (As you explore the maze, squares you set as 'explored' will be printed with a dot, and squares you 'mark' will display an X)

```
##########
#    xx  #
# ###x## #
# # xx # #
# # x# # #
# ##x#####
# #.xx    #
# #.#x # #
#####x####
#...#xxxx?
#.#..xx#.#
##########
```

[Example credit: Zorah Fung, University of Washington]

# Example: Solving the Maze




[Picture credit: Zorah Fung]

What are our choices in this problem?

Algorithms and Data Structures 1

# QUEUE ADT

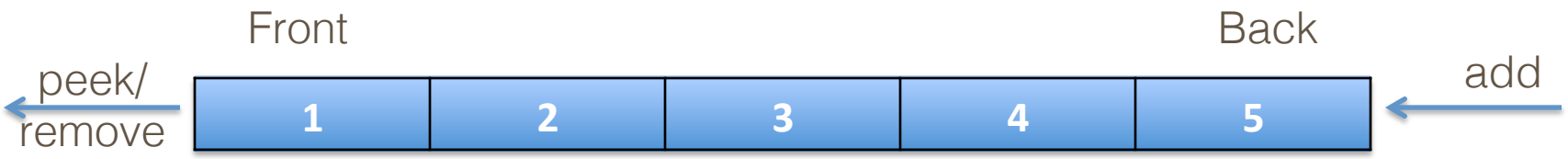
# My Least Favorite Queues

[Pictures credit: http://airport.blog.ajc.com, https://s1.cdn.autoevolution.com/images/news/the-longest-traffic-jam-in-history-12-days-62-mile-long-47237-7.jpg]

# What is a Queue?

- Queue – a data collection that retrieves elements in the FIFO order (first in, first out)
  - Elements are stored in order of insertion, but don't have indexes
    - Client can only:
      - Add to the end of the queue,
      - Examine/remove the front of the queue

- Basic queue operations:
  - **Add** (`enqueue`) - add an element to the back of the queue
  - **Peek** - examine the front element
  - **Remove** (`dequeue`) - remove the front element



# Implementations of Queues

- Like stack, queue can be seen as a list with restriction, and can be implemented as a list:

- Example: ArrayList implementation
  - Initially, queue only has elements 1 and 2
  - Add another element, 3, to the queue
  - Add another element, 4, to the queue
  - Add another element, 5, to the queue
  - Remove an element from the queue
  - Remove an element from the queue

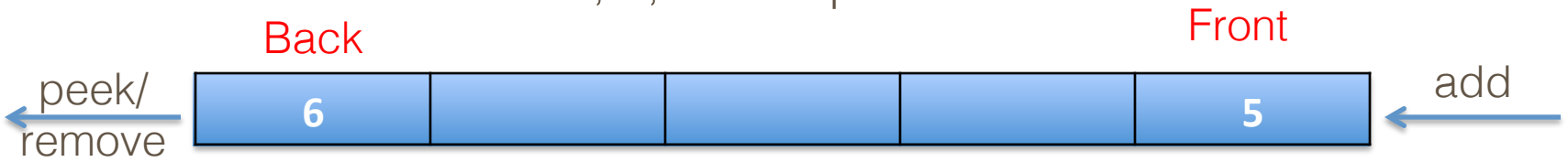

What happens when we remove two more elements from the queue?

# Implementations of Queues

- What happens when we remove two more elements from the queue?


- Approach – circular array implementation - whenever front or back get to the end of the array, allow them to wrap around to the beginning

- Example:
  - Add another element, 6, to the queue

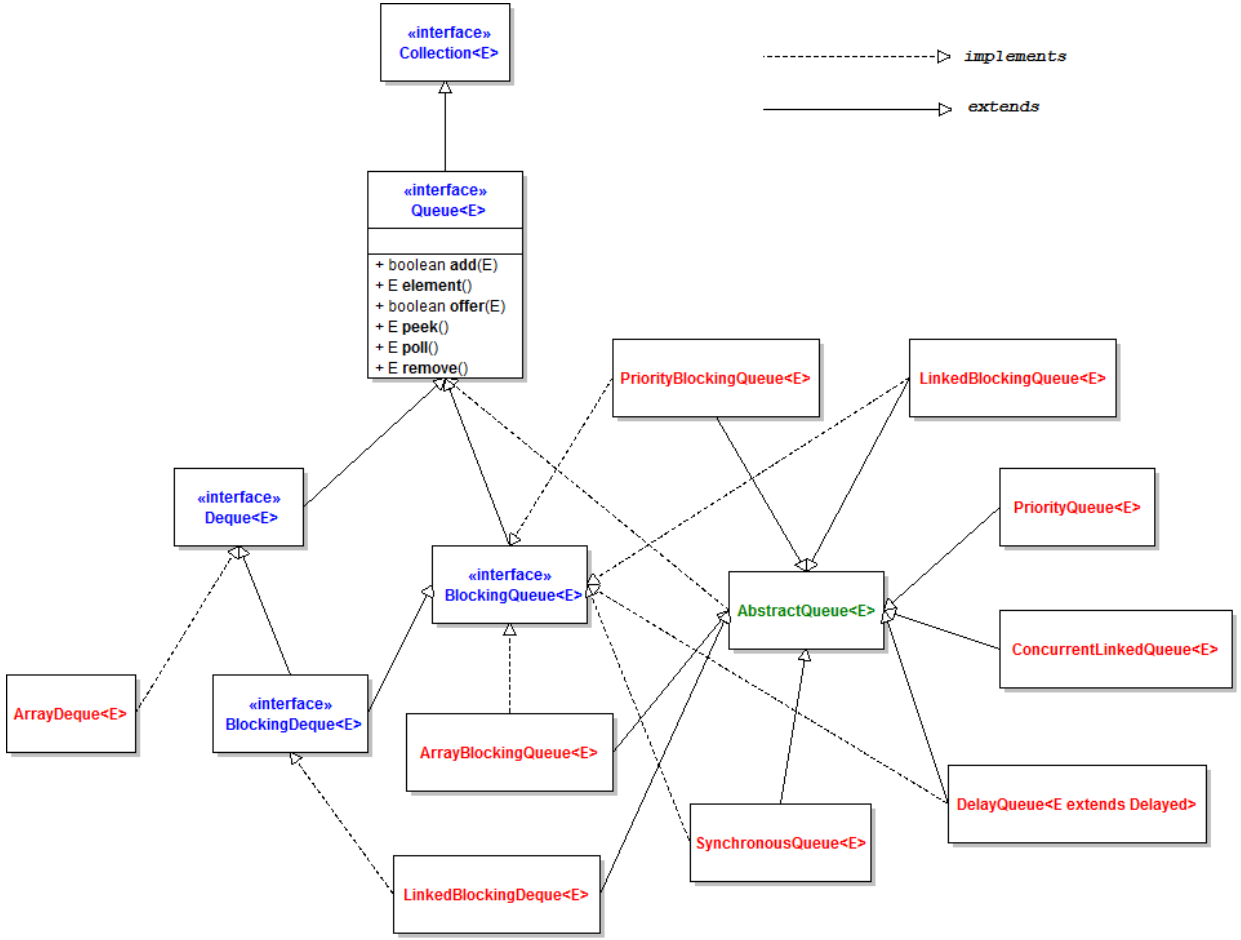



  - What is the front and what is the back of the queue now?

# Applications of Queues

- Operating systems:
  - Queue of print jobs to send to the printer
  - Queue of programs / processes to be run
  - Queue of network data packets to send
- Programming:
  - Modeling a line of customers or clients
  - Storing a queue of computations to be performed in order
- Real world examples:
  - People waiting in some line
  - ???

# Class Diagram of the Queue API




[Pictures credit: http://www.codejava.net/java-core/collections/class-diagram-of-queue-api]

# Java Interface `Queue`

| | |
|---|---|
| `add(`**value**`)` | places given value at back of queue |
| `remove()` | removes value from front of queue and returns it; throws a `NoSuchElementException` if queue is empty |
| `peek()` | returns front value from queue without removing it; r eturns `null` if queue is empty |
| `size()` | returns number of elements in queue |
| `isEmpty()` | returns `true` if queue has no elements |

- Example:

```
Queue<Integer> myQueue = new LinkedList<Integer>();
myQueue.add(10);
myQueue.add(18);
myQueue.add(2017); // front [10, 18, 2017] back
System.out.println(myQueue.remove()); // 10
```

# Mixing Queues and Stacks

- We often mix stacks and queues to achieve certain effects
- Example: Reverse the order of the elements of a queue

```
Queue<Integer> q = new LinkedList<Integer>();
q.add(1);
q.add(2);
q.add(3); // [1, 2, 3]
Stack<Integer> s = new Stack<Integer>();
while (!q.isEmpty()) {
   s.push(q.remove()); } // Q -> S
while (!s.isEmpty()) {
   q.add(s.pop()); } // S -> Q
System.out.println(q); // [3, 2, 1]
```

Algorithms and Data Structures 1

# TREES

# Trees

[Pictures credit: https://static1.squarespace.com/]

# Trees

- Tree - a directed, acyclic structure of linked nodes
  - Directed - one-way links between nodes
  - Acyclic - no path wraps back around to the same node twice

- Can be defined recursively:
  - A tree is either:
    - Empty(null), or
    - A root node that contains:
      - Data
      - A left subtree
      - A right subtree

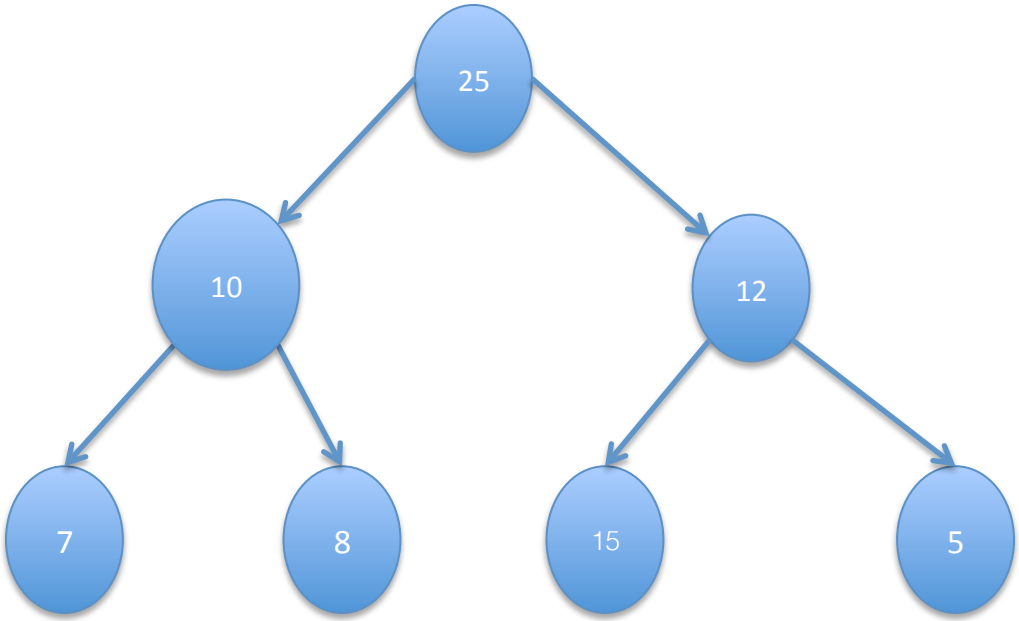
      (The left and/or right subtree could be empty)

# Trees Terminology

- Node - an object containing a data value and left/right children
- Root - topmost node of a tree
- Subtree – a smaller tree of nodes on the left or right of the current node
- Parent - a node above the left and right subtrees, that both subtrees are connected to
- Child - a root of each subtree
- Sibling - a node with a common parent
- Leaf - a node that has no children
- Branch - any internal node; neither the root nor a leaf
- Level or depth - length of the path from a root to a given node
- Height - length of the longest path from the root to any node

# Trees Terminology Example

- **Nodes:** {25, 10 ,12, 7, 8, 15, 5}
- **Root:** 25
- **Subtrees:** {10, 7, 8} and {12, 15, 5}
- **Parents:** 10 → {7, 8}, 12 → {15, 5}, 25→ {10, 12}
- **Children:** {10, 12, 7, 8, 15, 5}
- **Siblings:** {7, 8}, {15, 5} and {10, 12}
- **Leaves:** {7, 8, 15, 5}
- **Height:** 3



# Implementation of Trees

- Idea: each node in the tree keeps track of:
  - Its data
  - Links to all of its children

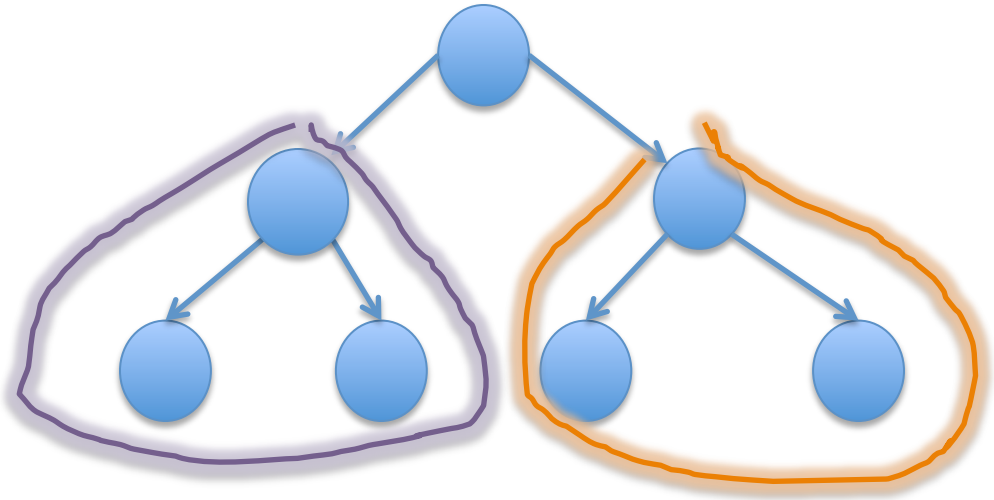
- Problem: The number of children per node is not known in advance, and may vary significantly

- Approach: Keep a children of each node in a linked list of tree nodes

```
class TreeNode {
    Object element;
    TreeNode firstChild;
    TreeNode nextSibling;
    }
```

# Binary Trees

- Binary tree – a tree in which no node can have more than two children



# Binary Tree Implementation

- A basic `BinaryNode` object stores:
  - Data,
  - Link to the left child
  - Link to the right child
- Multiple nodes can be linked together into a larger tree

```
class BinaryNode{
//Friendly data; accessible by other package routines
   Object element;
   BinaryNode left;
   BinaryNode right;
}
```

# Example: Class `StrigTreeNode`

```
StringTreeNode class
// An StringTreeNode object is one node in a binary tree of String
    public class StringTreeNode{
    public String data; // data stored at this node
    Public StringTreeNode left; // reference to left subtree
    Public StringTreeNode right; // reference to right subtree

    // Constructs a leaf node with the given data
    Public StringTreeNode(String data){
     this(data, null, null);
    }
    // Constructs a branch node with the given data and links
    Public StringTreeNode(String data, StringTreeNode left, StringTreeNode right){
     this.data = data;
     this.left = left;
     this.right = right;
    }
}
```

# Example: Class `StringTree`

```
// An StringTree object represents an entire binary tree of
String.
public class StringTree{
    private StringTreeNode root;
    //some methods
}
```

- Observations:
  - We can only talk to the `StringTree`, not to the node objects inside the tree
  - Methods of the `StringTree` create and manipulate the nodes, their data and links between them

Algorithms and Data Structures 1

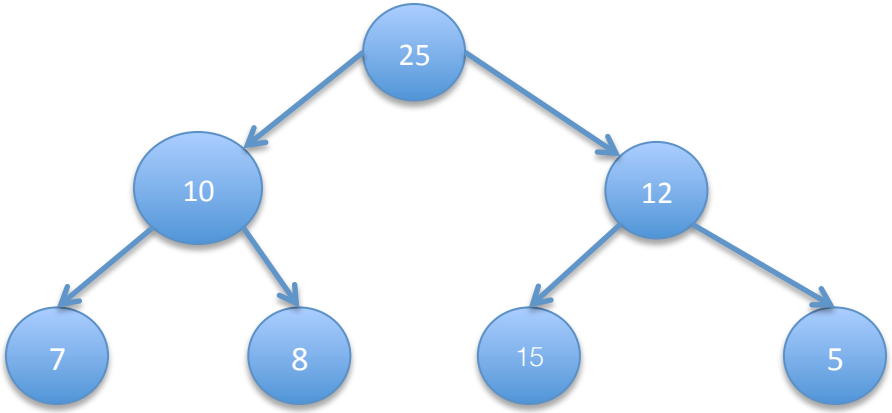
# TREE TRAVERSALS

# Tree Traversals

- Tree traversal - an examination of the elements of a tree
  - Used in many tree algorithms and methods
- Common orderings for traversals:
  - Pre-order – process root node, then its left/right subtrees
  - In-order – process left subtree, then root node, then right subtree
  - Post-order – process left/right subtrees, then root node

# Tree Traversals Example

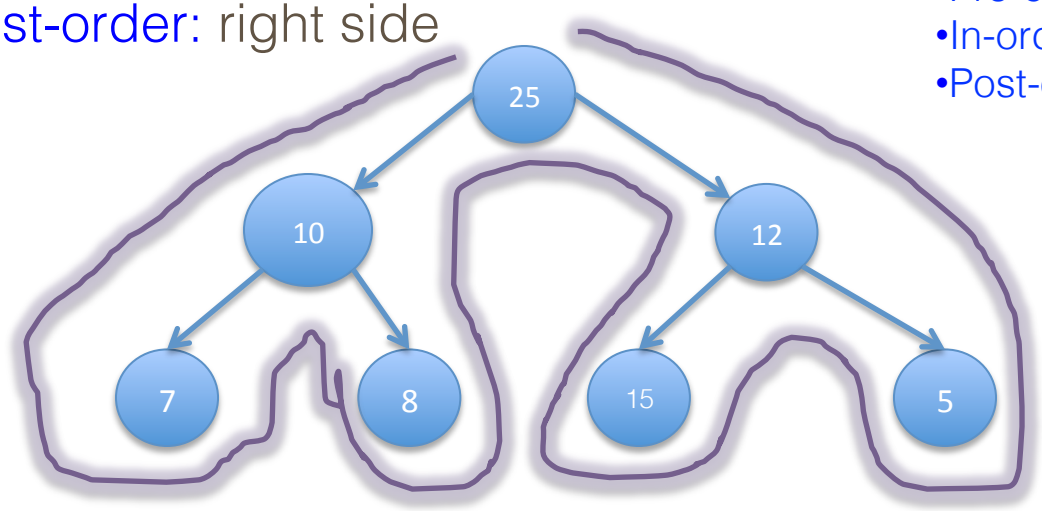
- Common orderings for traversals:
  - Pre-order – process root node, then its left/right subtrees
  - In-order – process left subtree, then root node, then right subtree
  - Post-order – process left/right subtrees, then root node




- Pre-order: `25 10 7 8 12 15 5`
- In-order: `7 10 8 25 15 12 5`
- Post-order: `7 8 10 15 5 12 25`

# Tree Traversals Trick

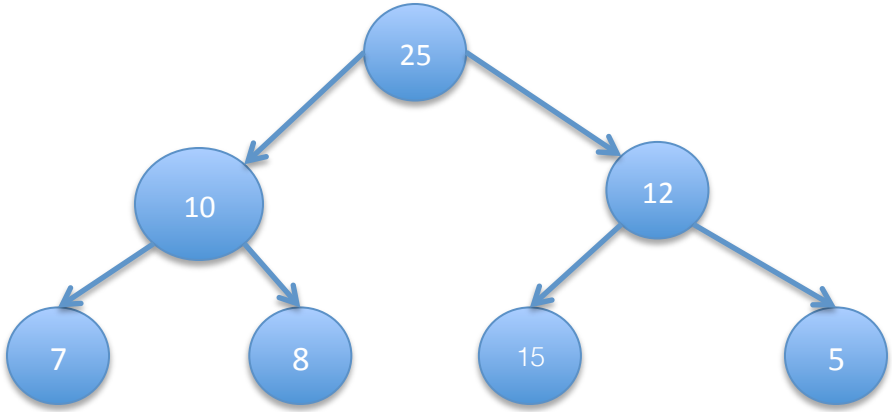
- To quickly generate a traversal, trace a path around the tree
- As you pass a node on the proper side, process it:
  - Pre-order: left side
  - In-order: bottom
  - Post-order: right side




- Pre-order: 25 10 7 8 12 15 5
- In-order: 7 10 8 25 15 12 5
- Post-order: 7 8 10 15 5 12 25

# Example: Printing a Tree

- Assume we have some class `IntTree`
- Add a method `print` to the `IntTree` class that prints the elements of the tree, such that
  - Elements of a tree are separated by spaces
  - A node's left and right subtree should be printed before it
- Example: `tree.print(); //7 8 10 15 5 12 25`



# Example: Printing a Tree

```
// An IntTree object represents an entire binary tree of ints
public class IntTree{
    private IntTreeNode overallRoot; // null for an empty tree ...
    public void print(){
    print(overallRoot);
    System.out.println(); // end the line of output
    }
    private void print(IntTreeNode root){
    // (base case is implicitly to do nothing on null)
        if (root != null){
        // recursive case: print left, right, center
            print(overallRoot.left);
            print(overallRoot.right);
            System.out.print(overallRoot.data + " ");
        }
    }
}
```

# Template for Tree Methods

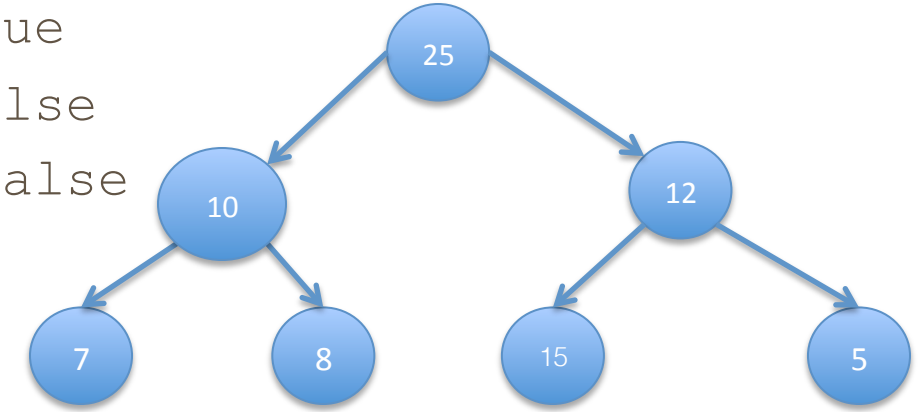
- Tree methods are often implemented recursively with a public/private pair
  - The private version accepts the root node to process

```
public class IntTree {
    private IntTreeNode overallRoot;
    ...
    public type name(parameters) {
        name(overallRoot, parameters);
    }
    private type name(IntTreeNode root, parameters) {
        ...
    }
}
```

# Example: `contains()`

- Add a method `contains` to the `IntTree` class that searches the tree for a given integer, returning `true` if it is found.

- Example: If an `IntTree` variable `tree` referred to the tree below, the following calls would have these results:
  - `tree.contains(25)` → `true`
  - `tree.contains(12)` → `true`
  - `tree.contains(4)` → `false`
  - `tree.contains(77)` → `false`



# Example: `contains()`

```
// Returns whether this tree contains the given integer
public boolean contains(int value){
    return contains(overallRoot, value);
}
private boolean contains(IntTreeNode node, int value){
    if (node == null){
        return false; // base case: not found here
    }else if (node.data == value){
        return true; // base case: found here
    }else{
        // recursive case: search left/right subtrees
        return contains(node.left, value) || contains(node.right,
                value);
     }
}
```

Algorithms and Data Structures 1

# ASSIGNMENT 4 – COMMENTS AND HINTS

# Assignment 4 – Background

- Halloween Trick-or-treating:
  - Kids walk from a house to a house, asking for a trick or a treat
  - Different houses give out different kinds of treats (candy)
  - Kids have very particular preferences what kinds of candy would they like to get
  - Kids also have historical knowledge about the type of candy given out in different households

# Assignment 4

- Your Assignment:
  - Given:
    - Historical knowledge about the neighborhood
    - A list of desired candy for every child
  - Find a way to traverse a neighborhood to get all the candy from the list

# Assumptions and Simplifications

- Only one neighborhood, with four kinds of houses:
  - Mansions
  - Detached homes
  - Duplexes
  - Townhomes
- Houses give out four different sizes of candy:
  - Super size
  - King size
  - Regular size (default)
  - Fun size

# Assumptions and Simplifications

- Houses give out ten different kinds of candy:
  - Twix
  - Snickers
  - Mars
  - Kit Kat
  - Whoopers
  - Milky Way
  - Toblerone
  - Crunch
  - Baby Ruth
  - Almond Joy
- And not other kind and size of candy

# Assumptions and Simplifications

- Desired list of candy:
  - Case-insensitive
  - Ordered
- Historical knowledge about households:
  - Specifics which household gives out which size and kind of candy

# How Do You Implement This?

- Classic slogans:
  - Do one thing well
  - Minimize coupling, maximize cohesion
  - Isolate operations/abstractions in modules
  - Hide implementation details
- Idea: abstract class Candy, inherited by a concrete candy class for every candy type

# Where do the Operations Go?

- Pure "object-oriented" style
  - Really, really, really smart candy objects
  - Each candy object knows how to perform every operation on itself

```
public class TwixCandy extends Candy{
  public TwixCandy(...);
  public typeCheck(...);
  public print();
  public prettyPrint(...);
  ...
}
```

# Critique

- This is nicely encapsulated – all details about a TwixCandy are hidden in that class
- But it is poor modularity
- What happens if we want to add some new operation on a candy?
  - Have to modify every concrete candy class ☹
- Worse: the details of any particular operation (e.g., type checking, print) are scattered across all candy classes

# Modularity Issues

- Smart objects make sense if the set of operations is relatively fixed, but we expect to need flexibility to add new kinds of objects
- Example: graphics system
  - Operations: draw, move, iconify, highlight
  - Objects: textbox, scrollbar, canvas, menu, dialog box, window, plus new objects defined as the system evolves

# Visitor Pattern

- Idea: package each operation (`print`, `pretty print`, `maximize candy`) in a separate visitor class
- Create exactly one instance of each visitor class
  - Sometimes called a "function object"
  - Contains all of the methods for that particular operation, one for each kind of an object
- Include a generic "accept visitor" method in every class
- To perform an operation, pass the appropriate "visitor object" around during a traversal

# Avoiding instanceof

- We'd like to avoid huge if-elseif nests in the visitor to discover the node types

```
void checkTypes(Candy c) {
    if (c instanceof TwixCandy) { ... }
    else if (c instanceof MarsCandy) { ... }
    else if (c instanceof KitKatCandy) { ... }
...
}
```

# Visitor Double Dispatch

- Idea: include a "**visit**" method for every type object in each Visitor

```
void visit(TwixCandy);
void visit(MarsCandy);
etc.
```

- Include an **accept(Visitor v)** method in every class
- When **Visitor v** is passed to an **object**, the **object's** accept method calls **v.visit(this)**
  - Selects correct Visitor method for this node
  - "Double dispatch"

# Visitor Interface

- Every separate Visitor implements this interface

```
interface Visitor {
    // overload visit for every object type
    public void visit(TwixCandy c);
    public void visit(MarsCandy c);
    public void visit(KitKatCandy c);
    …
}
```

# Accept Method in Each Class

- Every class overrides accept(Visitor)
- Example

```
public class TwixCandy extends Candy{
    ...
    // accept a visit from a Visitor object v
    public void accept(Visitor v) {
      v.visit(this);
    }
    ...
}
```

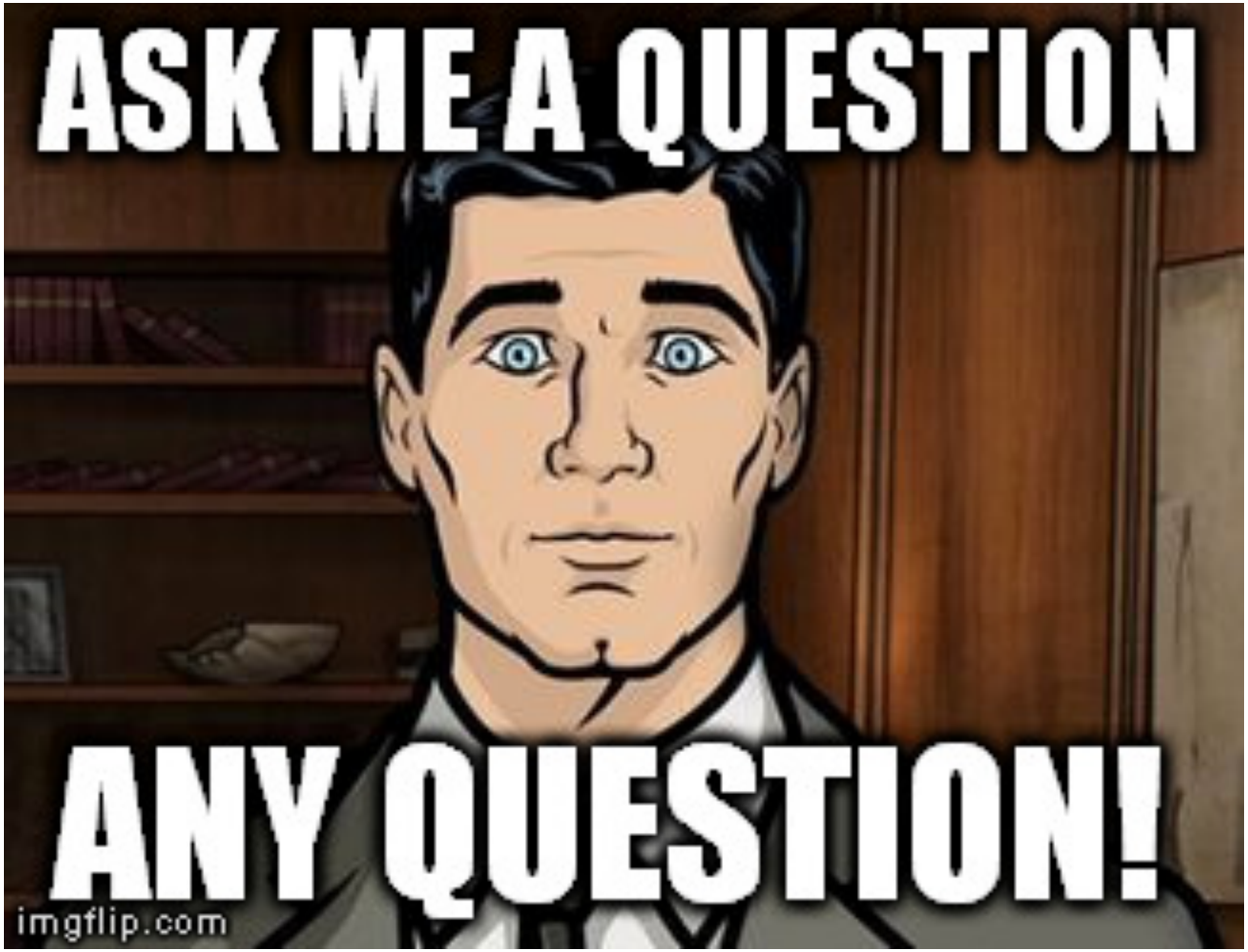
- Key points
  - Visitor object passed as a parameter to TwixCandy
  - TwixCandy calls visitor's visit method, which dispatches to visit(TwixCandy) automatically – i.e., the correct method in the visitor object for this kind of node

# Encapsulation

- A visitor object often needs to be able to access state in the objects
  - ∴ May need to expose more object's state than we might have done otherwise
    - i.e., lots of public fields in objects
  - – Overall a good tradeoff – better modularity

    (plus, the objects are relatively simple data objects anyway – not hiding much of anything)

# Your Questions




[Meme credit: imgflip.com]

# References and Reading Material


- Mark Allen Weiss, Data Structures and Algorithm Analysis in Java, chapters 1 through 4
- Oracle, java.util Class Collections, [Online] http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html
- Oracle, Java Tutorials Collections, [Online] https://docs.oracle.com/javase/tutorial/collections/
- Vogella, Java Collections – Tutorial, [Online] http://www.vogella.com/tutorials/JavaCollections/article.html
- Oracle, Java Tutorials, Nester Classes, [Online] https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html
- Princetion, Introduction to Programming in Java, Recursion, [Online] http://introcs.cs.princeton.edu/java/23recursion/
- Jeff Ericson, Backtracking, [Online] http://introcs.cs.princeton.edu/java/23recursion/
- Wikibooks, Algorithms/Backtracking, [Online], https://en.wikibooks.org/wiki/Algorithms/Backtracking