Northeastern University

CS 5010: Programming Design Paradigms

Fall 2022

Lecture 8: Data Structures and Algorithms 2

Acknowledgement: lecture notes inspired by course material prepared by UW faculty members Z. Fung and H. Perkins.
Additional credits to Tamara Bonaci

Brian Cross
b.cross@northeastern.edu

Administrivia

- Homework #3 done!
- Codewalk #3 was due last night
- No Lab next week ☺
  - Meet with your partner, go over the assignment.
- Homework 4
  - Comes out tomorrow
  - UML Design Draft due Monday Oct 31 @ 11:59pm
  - HW Due Monday, November 7th @ 11:59pm

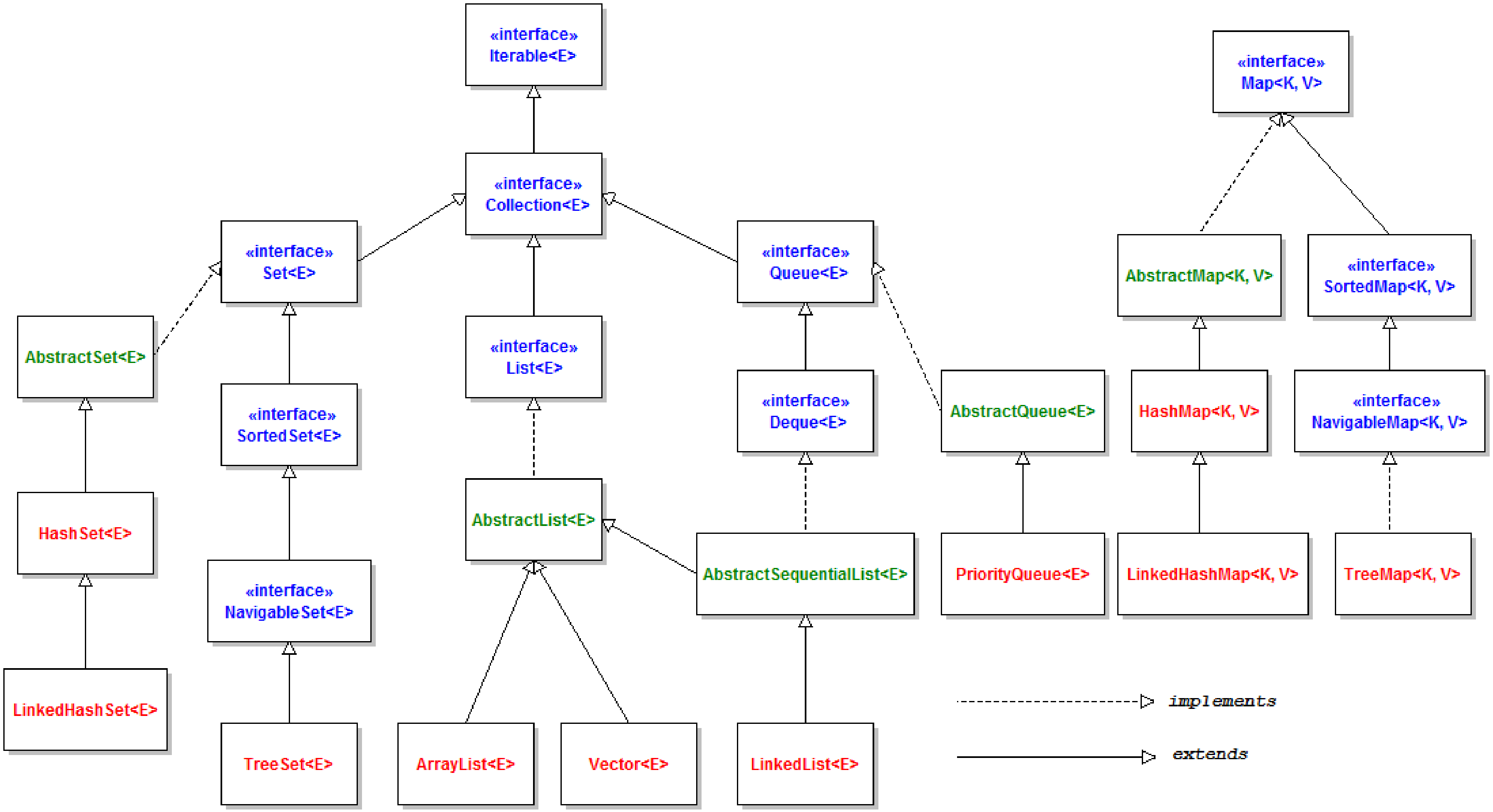
Administrivia

- Group projects
  - Homeworks #4 - #6
- New group repos created this week
  - Group_Id1_Id2
- Who still needs a partner?

Agenda – Algorithms and Data Structures 2

- Stack ADT
- Queue ADT
- Deque ADT
- Set ADT
- Map ADT
- Tree ADT
- Graph ADT

Java Collections API



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

Java Collections Framework

- Part of the `java.util` package

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

Interfaces Iterable<T>

- Super-interface for interface `Collection<T>`
- 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()`

Modifier and Type	Method and Description
default void	forEach(Consumer<? super T> action) Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
Iterator<T>	iterator() Returns an iterator over elements of type T.
default Spliterator<T>	spliterator() Creates a Spliterator over the elements described by this Iterable.

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

- 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()`

`Comparable` Template

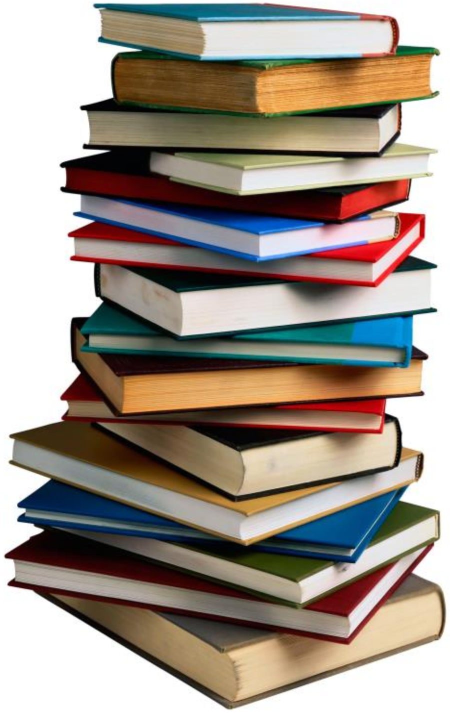
```
public class name implements Comparable<name> {

    ...

    public int compareTo(name other) {
        ...
    }
}
```

Algorithms and Data Structures 1

STACK ADT

Stacks

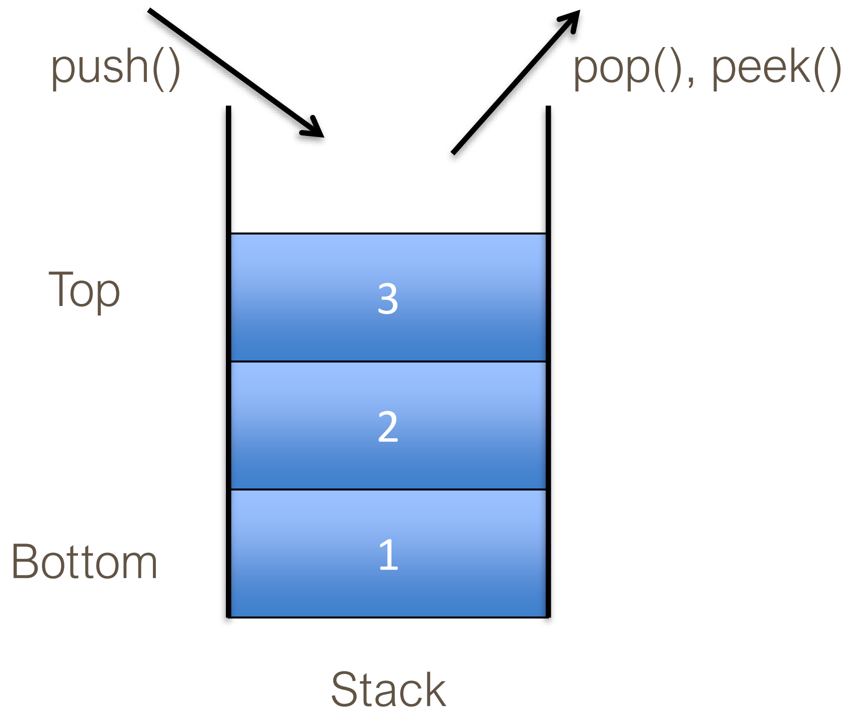
- Some popular stacks:



[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]

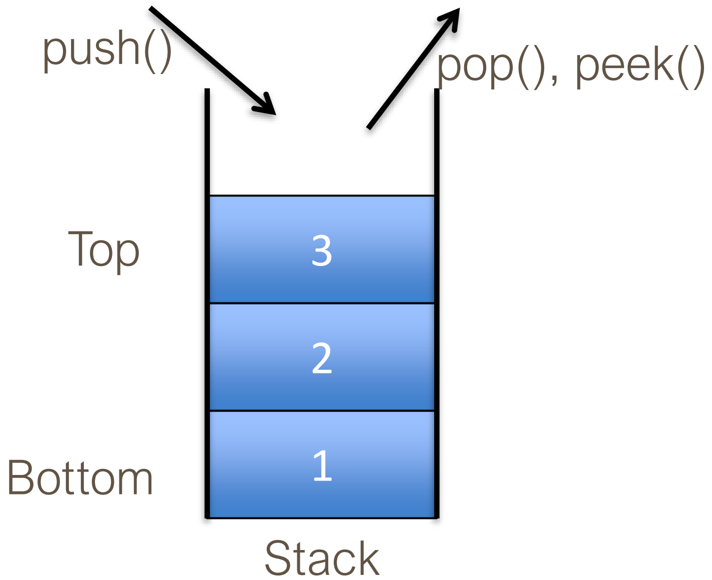
What is a Stack?

- 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?

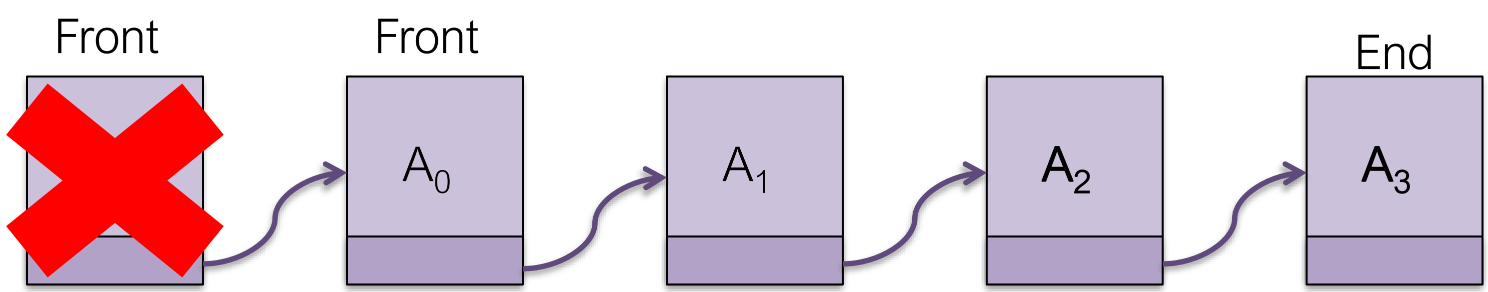


- Is there another way to think about a stack?
- Stack – a constrained data collection where clients are limited to use only limited optimized methods (`pop, push, peek`)
- Stack – a list with restriction that insertions and deletions can be performed in only one position, the end of the list, called 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 2022"); //bottom ["Hello", "PDP", "Fall 2022"] top
System.out.println(s.pop()); //Fall 2022
```

Applications of a Stack

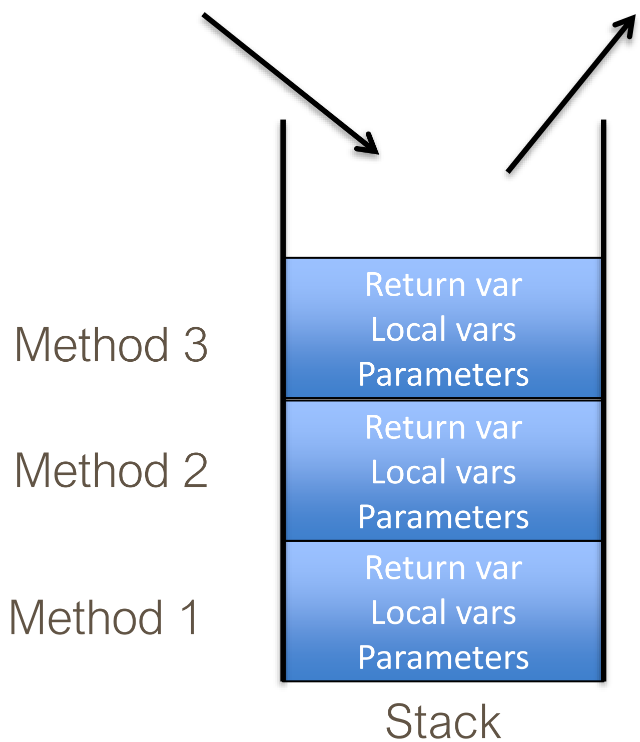
- Programming languages and compilers:
  - Method calls *(call=push, return=pop)*
  - Compilers (parsers)

- 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)

Example: Postfix Expressions

Postfix

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

Example

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

Stack
1.1
5.5

Example: Postfix Expressions

Postfix

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

Example

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

Stack
6.05

Example: Postfix Expressions

Postfix

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

Example

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

Stack
1.1
4.5
6.05

Example: Postfix Expressions

Postfix

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

Example

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

Stack
4.95
6.05

Example: Postfix Expressions

Postfix

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

Example

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

Stack
11

Example: Postfix Expressions

Postfix

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

Example

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

Stack
1.1
7
11

Example: Postfix Expressions

Postfix

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

Example

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

Stack
7.7
11

Example: Postfix Expressions

Postfix

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

Example

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

Stack
18.7

Example: Postfix Expressions

Postfix

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

Example

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

Stack
1.1
8
18.7

Example: Postfix Expressions

Postfix

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

Example

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

Stack
8.8
18.7

Example: Postfix Expressions

Postfix

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

Example

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

Stack
27.5

Algorithms and Data Structures 2

QUEUE ADT

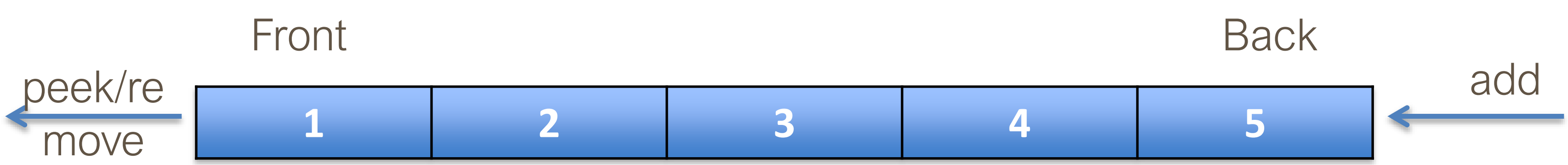
Less Than Popular Queues

[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

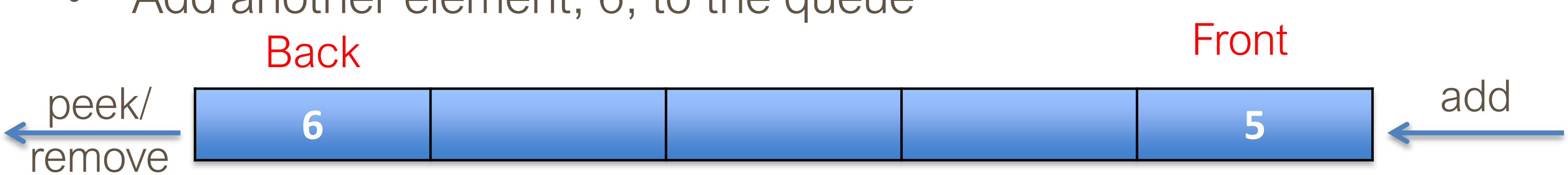
What happens when we remove two more elements from the queue?

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

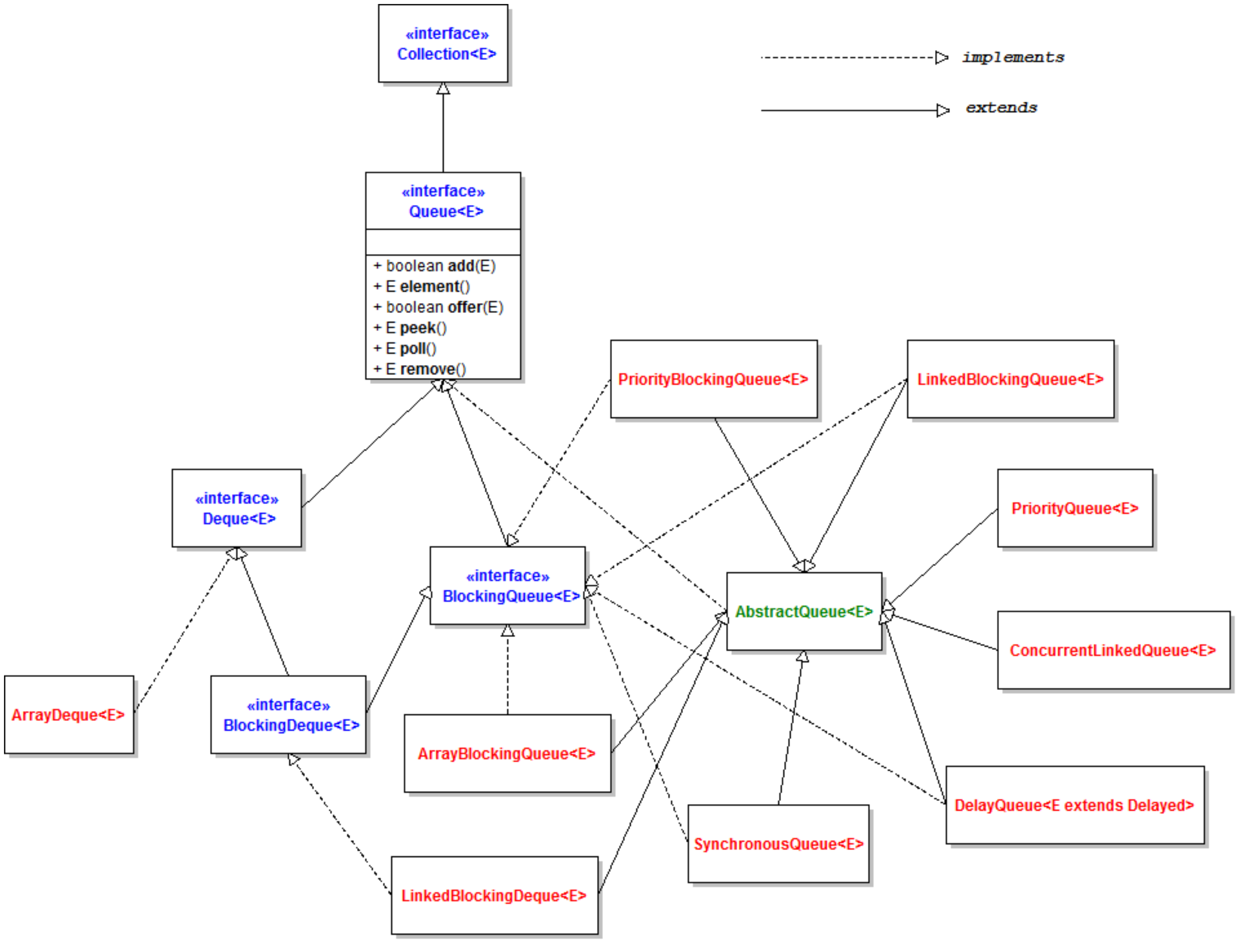


  - What is the front and what is the back of the queue now?

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(26);
myQueue.add(2022); // front [10, 26, 2022] 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 2

DEQUE ADT

Deque

[Pictures credit: http://www.java2novice.com/data-structures-in-java/queue/double-ended-queue/]

Deque

	First Element (Head)	
	Throws exception	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`
Remove	`removeFirst()`	`pollFirst()`
Examine	`getFirst()`	`peekFirst()`

	Last Element (Tail)	
	Throws exception	*Special value*
Insert	`addLast(e)`	`offerLast(e)`
Remove	`removeLast()`	`pollLast()`
Examine	`getLast()`	`peekLast()`

[Pictures credit:https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html

Deque

Queue Method	Equivalent Deque Method
add(e)	addLast(e)
offer(e)	offerLast(e)
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

[Pictures credit:https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html

Deque

Stack Method	Equivalent `Deque` Method
`push(e)`	`addFirst(e)`
`pop()`	`removeFirst()`
`peek()`	`peekFirst()`

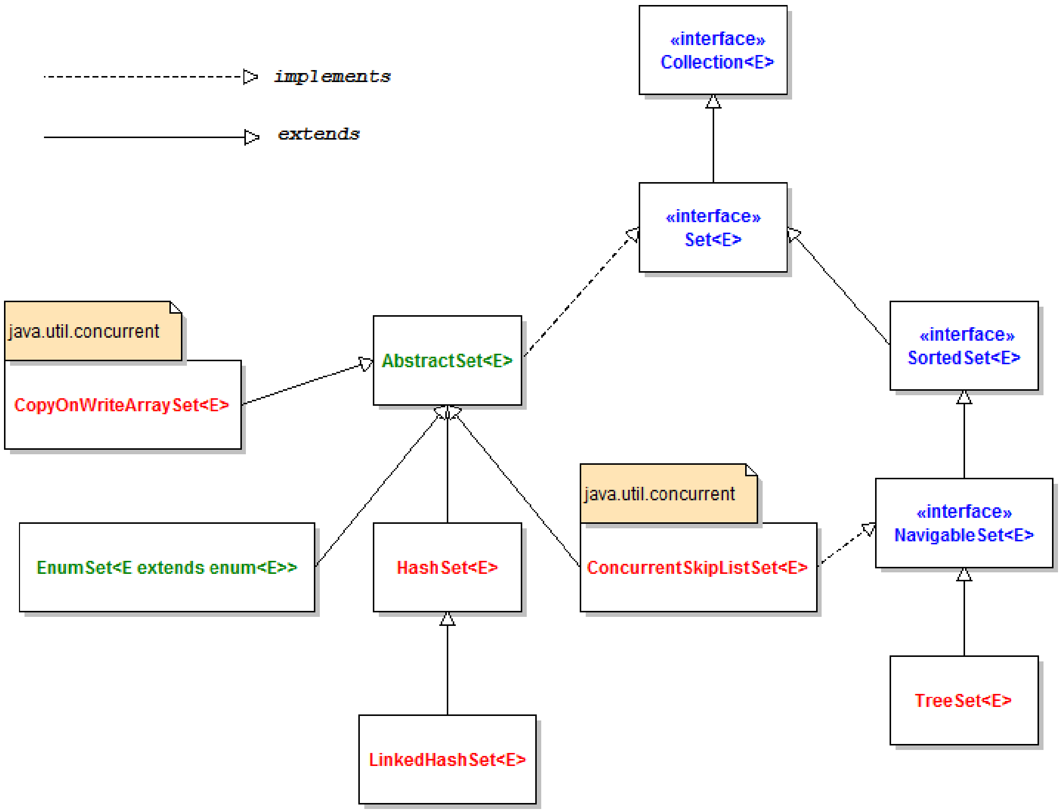
Algorithms and Data Structures 2

SET ADT

Sets

- Set - a collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  - `add,`
  - `remove,`
  - `search (contains)`
- We don't think of a set as having indexes; we just add things to the set in general and don't worry about order

Set API Class Diagram



[Pictures credit: http://www.codejava.net/images/articles/javacore/collections/Set%20API%20class%20diagram.png]

`Set` Implementations

- In Java, sets are represented by `Set` type in `java.util`
- Set is implemented by `HashSet` and `TreeSet` classes
  - `HashSet:` implemented using a "hash table" array
    - Very fast: O(1) for all operations
    - Elements are stored in unpredictable order
  - `TreeSet`: implemented using a binary search tree
    - Pretty fast: O(log N) for all operations
    - Elements are stored in sorted order
  - `LinkedHashSet:`
    - O(1) but stores in order of insertion, but slightly slower than `HashSet` because of extra info stored

`Set` Methods

- We can construct an empty set, or one based on a given collection

- Examples:

```
Set<Integer> set = new TreeSet<Integer>(); // empty

List<String> list = new ArrayList<String>();
...
Set<String> set2 = new HashSet<String>(list);
```

`Set` Methods

`add(value)`	adds the given value to the set
`contains(`**`value`**`)`	returns true if the given value is found in this set
`remove(`**`value`**`)`	removes the given value from the set
`clear()`	removes all elements of the set
`size()`	returns the number of elements in list
`isEmpty()`	returns true if the set's size is 0
`toString()`	returns a string such as "[3, 42, -7, 15]"

Algorithms and Data Structures 2

MAP ADT

Maps

- Write a program that stores, modifies and retrieves:
  - Assignment grades for every student in this College
  - Financial information for every client of some bank
  - Browsing history for every user of some search engine
  - Searches and transactions for every user of some online retailer
  - Activity and likes of every user of some online platform

- Question: What do these records have in common?
- The way we think about them → every data sample has a unique user → unique ID (key)
- What is the appropriate data collection for this data?
- Maps

Maps

- **Map** – a data collection that holds a set of unique *keys* and a collection of *values*, where each key is associated with one value
- Also known as:
  - Dictionary
  - Associative array
  - Hash
- Basic map operations:
  - `put(key, value)` - adds a mapping from a key to a value
  - `get(key)` - retrieves the value mapped to the key
  - `remove(key)` - removes the given key and its mapped value

`Map` Implementations

- In Java, maps are represented by `Map` type in `java.util`
- Map is implemented by the `HashMap` and `TreeMap` classes
  - `HashMap` - implemented using a "hash table"
    - Extremely fast: **O(1)**
    - Keys are stored in unpredictable order
  - `TreeMap` - implemented as a linked "binary tree" structure
    - Very fast: **O(log N)**
    - Keys are stored in sorted order
  - `LinkedHashMap` - O(1)
    - Keys are stored in order of insertion

`Map` Implementations

- Map requires 2 types of parameters:
  - One for keys
  - One for values

- Example:

```
// maps from String keys to Integer values
Map<String, Integer> votes = new HashMap<String, Integer>();
```

`Map` Methods

`put(key, value)`	adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one
`get(`**`key`**`)`	returns the value mapped to the given key (`null` if not found)
`containsKey(`**`key`**`)`	returns `true` if the map contains a mapping for the given key
`remove(`**`key`**`)`	removes any existing mapping for the given key
`clear()`	removes all key/value pairs from the map
`size()`	returns the number of key/value pairs in the map
`isEmpty()`	returns `true` if the map's size is 0
`toString()`	returns a string such as "{a=90, d=60, c=70}"

keySet and Values

- `keySet` method returns a `Set` of all keys in the map
  - It can loop over the keys in a `foreach` loop
  - It can get each key's associated value by calling `get` on the map

Example:

```
Map<String, Integer> ages = new TreeMap<String, Integer>();
ages.put("Marty", 19);
ages.put("Geneva", 2);
ages.put("Vicki", 57);

// ages.keyset() returns Set<String>
for (String name : ages.keySet()) {              // Geneva -> 2
    int age = ages.get(name);                     // Marty -> 19
    System.out.println(name + " -> " + age); // Vicki -> 57
}
```

Methods `keySet` and `Values`

- `values` method returns a collection of all values in the map
  - It can loop over the values in a `foreach` loop
  - No easy way to get from a value to its associated key(s)

`keySet()`	returns a set of all keys in the map
`values()`	returns a collection of all values in the map
`putAll(`**map**`)`	adds all key/value pairs from the given map to this map
`equals(`**map**`)`	returns `true` if given map has the same mappings as this one

Algorithms and Data Structures 2

HASHING AND HASH FUNCTIONS

Introduction to Hashing

- Problem: how much does a new phone cost?
  - e.g., Pixel??
  - e.g., iPhone??

- Ways to answer this question:
  - Look it up in an unsorted list of all phone prices: O(n)
  - Look it up in a sorted list of all phone prices: O(log n)
  - Ask your buddy if she remembers: O(1)

Introduction to Hashing

- Does this difference in time complexity matter?

Num Items	Simple Search $O(n)$	Binary Search $O(\log n)$	Buddy $O(1)$
100	10 sec	1 sec	Instant
1000	1.6 min	1 sec	Instant
10000	16.6 min	2 sec	Instant

It sure does!

How do we replicate our buddy with data structures???

Introduction to Hashing

- How do we replicate our buddy with data structures???

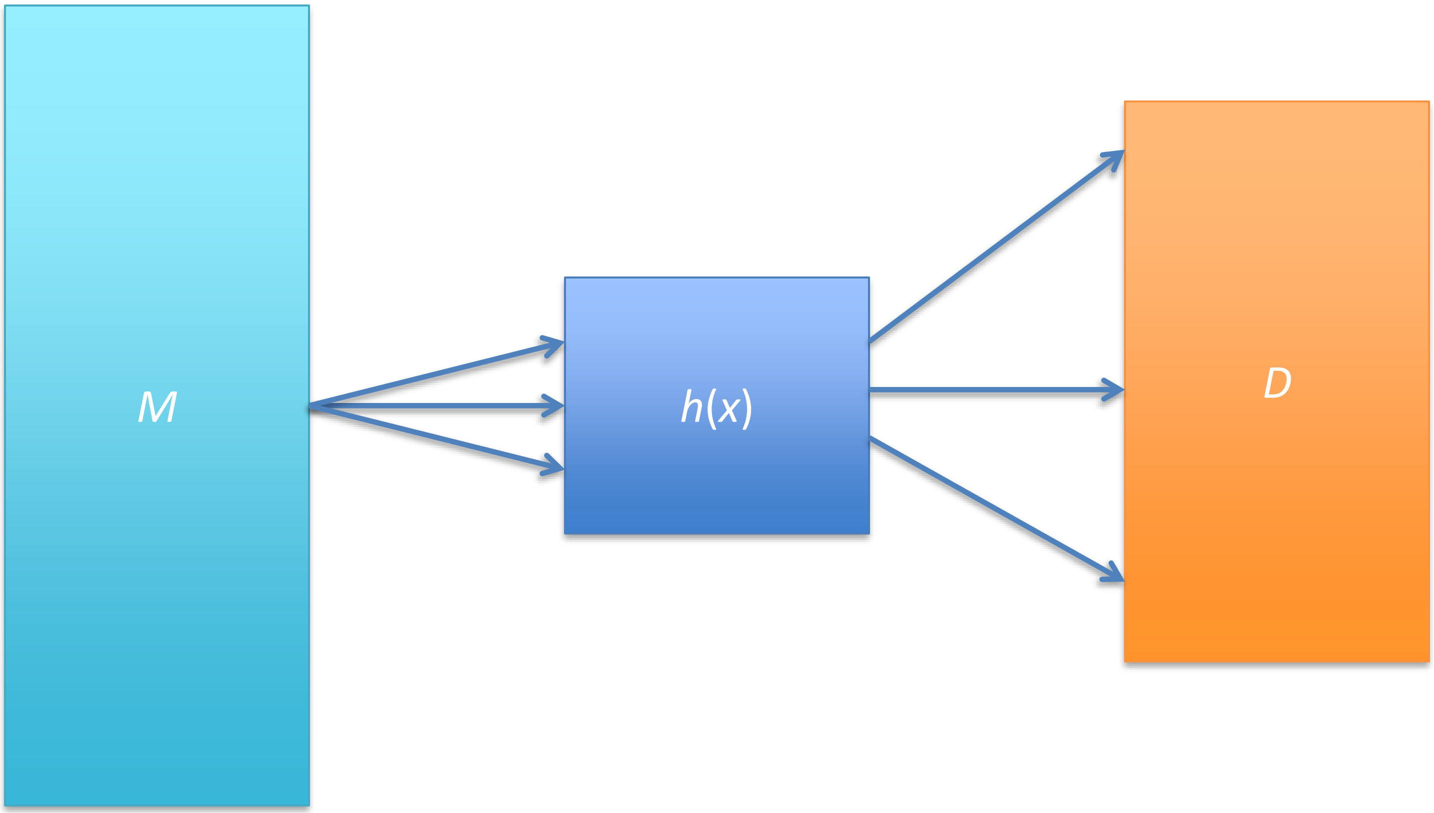
- Observation - each item is essentially 2 items:
    - Product name
    - Price
- If we sort by name, we can find a product in O(log n) time
    - How can we get down to O(1)?

Introduction to Hash Functions

- Use a hash function to transform a {string, number, struct, object, . . . } into a number
- Be consistent - every time you give it the same input, it returns the same value
  - Every time you give it “iPhone 13”, it returns ‘xxx’
- Make sure that the output is distinguishing
  - In the best case, it returns a different value for every distinct input given to it
  - Why: a function that always returns 1 is not helpful

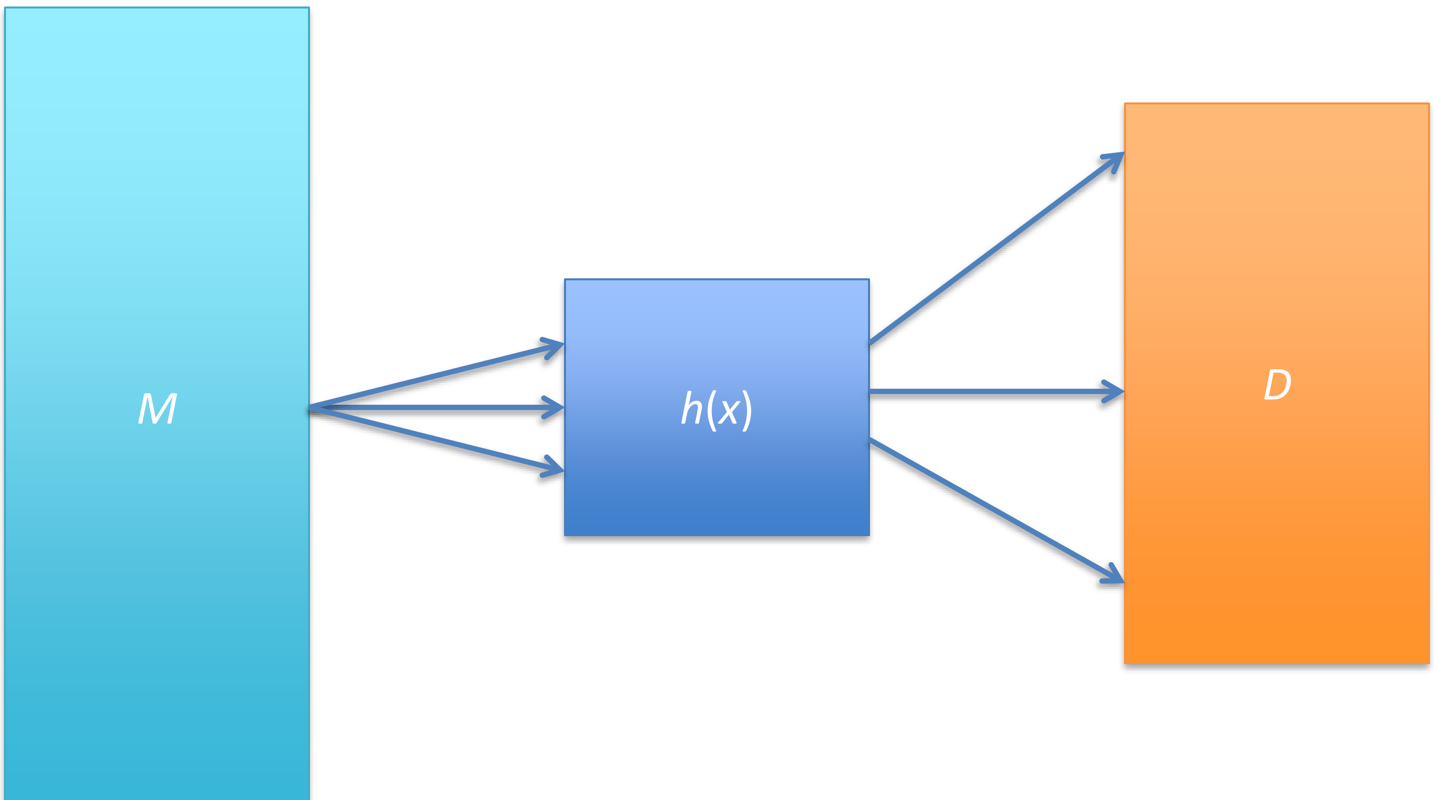
Introduction to Hash Functions

- Another possible approach
  pseudo-random mapping using a hash function *h*(*x*)



Introduction to Hash Functions

- Hash function
  maps the large space M into target space D



Introduction to Hash Functions

- Hash function
  maps the large space M into target space D

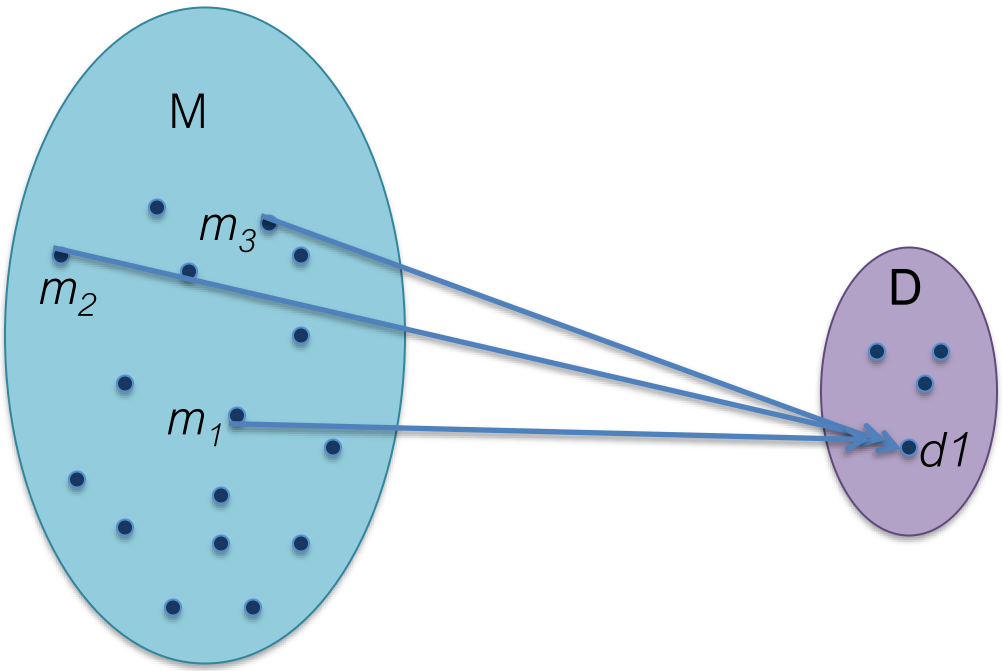
- Desired properties of hash functions:
  - Repeatability:
    - For every x in M, it should always be h(x) = h(x)
  - Equally distributed:
    - For some y, z in M, P(h(y)) = P(h(z))
  - Constant-time execution: h(x) = O(1)

Simple Hash Function

```
/**
* A hash method for String objects.
* @param key the String to hash.
* @param tableSize the size of the hash table.
• @return the hash value.
  */
  public static int hash(String key, int tableSize) {
      int hashVal = 0;

      for(int i=0; i<key.length(); i++)
         hashVal = 37 * hashVal + key.charAt(i);

      hashVal %= tableSize;
      if(hashVal < 0)
         hashVal += tableSize;
      return hashVal;
}
```

Problems with Hash Functions

- **Hash function** – can be thought of as a "lossy compression function"



- Do you see any problems here?
- Yes, collision!

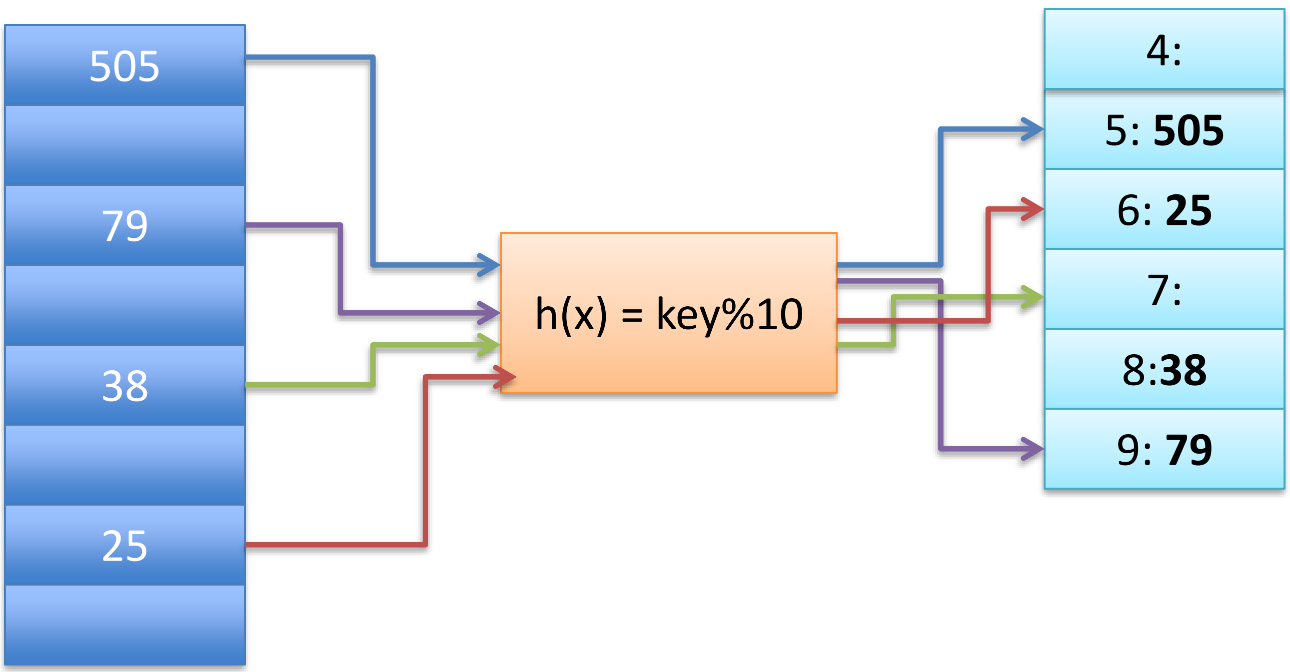
Resolving Collisions

- **Hash function** – can be thought of as a "lossy compression function"
- Problem – collision

- Possible approaches to resolve collisions:
  - Store data in the next available space
  - Store both in the same space
  - Try a different hash
  - Resize the array

Resolving Collisions – Linear Probing

- Linear probing – a simple approach
  - When a collision occurs, find the next available spot in the array



Problems with Linear Probing

- Linear probing – a simple approach
  - When a collision occurs, find the next available spot in the array
- Searching for some element *x*:
  - Go to position *h*(*x*), then cycle through all entries until you either find the element, or the blank space
- Adding an element *y*, that should go to the position taken by the colliding element:
  - Add element *y* to the next available spot - clustering

Problems with Linear Probing

- If a cluster becomes too large, negative impact on the hashing performance
- The chances of collision with the cluster increase
- The time it takes to find something in the cluster increases, and it **isn't O(1)**

Quadratic Probing

- Whereas linear probing increments indices by one each time, quadratic probing goes through the squares

- For example, linear probing would check index 3, then:
  - 3+1,
  - 3+2,
  - 3+3,
  - 3+4 etc.
- Quadratic probing would check index 3, then
  - 3+1,
  - 3+4
  - 3+9
  - 3+16 etc.

Problems with Quadratic Probing

- Example: Consider a hash function for ints, h(x) = x%7

- Insert, 3, 10,17,24,31,38

- What happens? Where does 31 go?
- 31%7=3
- 3+1%7 = 4
- 3+2%7 = 5
- 3+4%7 = 0
- 3+9%7 = 5
- 3+16%7 = 5

0: 17
1
2
3: 3
4:10
5: 24
6

Problems with Quadratic Probing

- Secondary clustering problem
  - Even when there is space available in the table, quadratic probing is not guaranteed to find an opening
  - In fact, half the array has to be empty to guarantee an opening
  - This approach reduces the O(n) problem of linear probing, but it introduces even larger memory constraints

Secondary Hashing

- If two keys collide in the hash table, then a secondary hash indicates the probing size
- Need to be careful, possible for infinite loops with a very empty array

Chaining

- Rather than probing for an open position, we could just save multiple objects in the same position
- Some data structure is necessary here
- Commonly a linked list, AVL tree or secondary hash table
- Resizing isn't **necessary**, but if you don't, you will get O(n) runtime

Hash Functions

- **In reality, good hash functions are difficult to produce**
- We want a hash that distributes our data evenly throughout the space
- Usually, our hash function returns some integer, which must then be moded to our table size
- **When discussing hash table efficiency, we call the proportion of stored data to table size the *load factor***

Java `HashTable` Class

- Implements a hash table, which maps keys to values

- Example: a HashTable of integers

```
Hashtable<String, Integer> number = new
                         HashTable<String, Integer>();
numbers.put(“one”, 1);
numbers.put(“two”, 2);
numbers.put(“three”, 3);

Integer n = numbers.get(“two”);
if(n != null) {
   System.out.println(“two = ” +n);
}
```

Algorithms and Data Structures 2

TREES

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

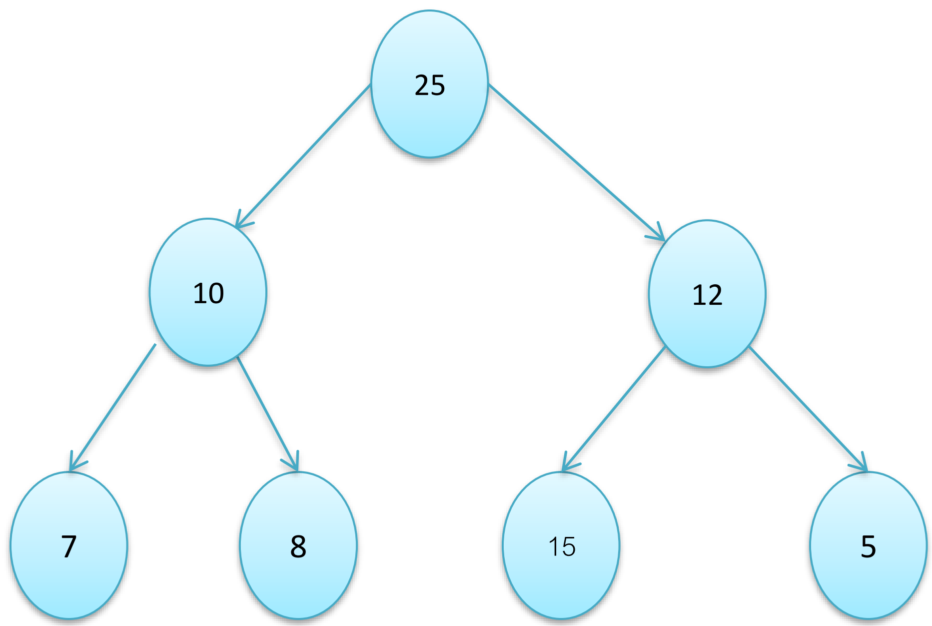
      (The left and/or right subtree could be empty)

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

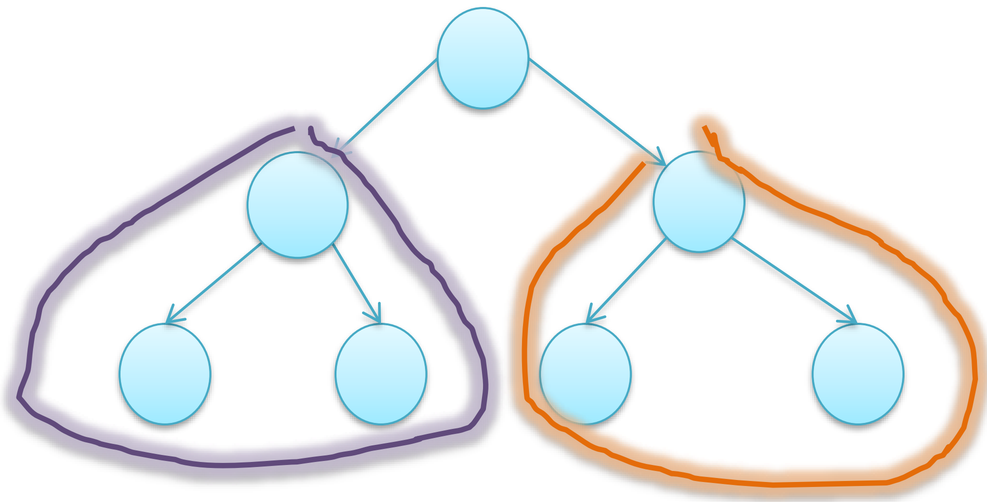
Trees Terminology Example

- **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



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{
   Object element;
   BinaryNode left;
   BinaryNode right;
}
```

Example: Class `StringTreeNode`

StringTreeNode class

```
// A 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;
    }
}
```

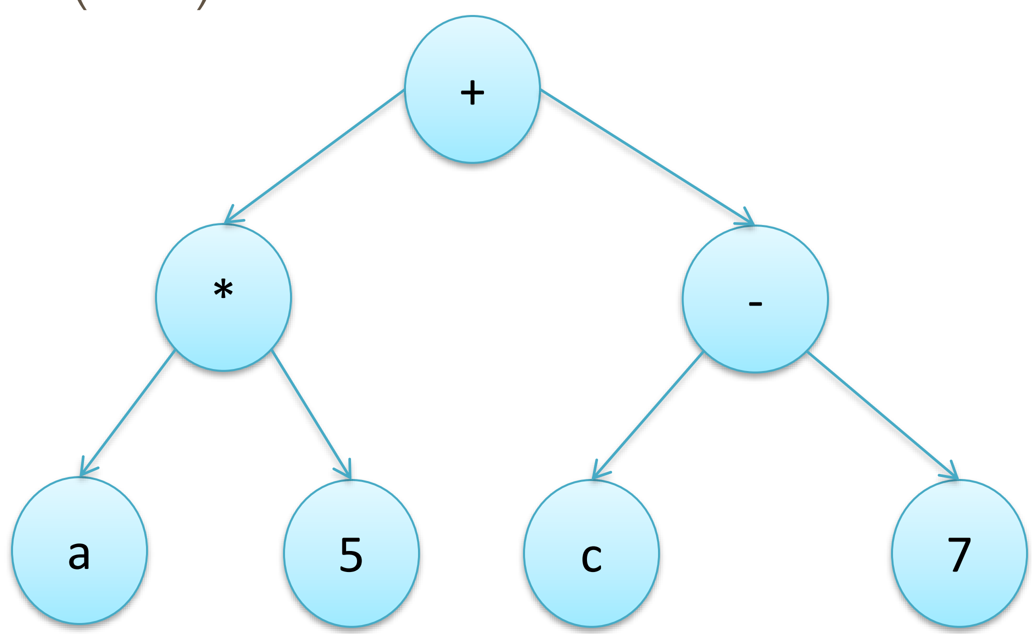
Example: Class `StringTree`

```
// A 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

Example: Expression Trees

- In an expression tree:
  - Leaves are operands (constants or variable names)
  - All other nodes are operators (unary or binary)
  - Example: (a * 5) + (c -7)

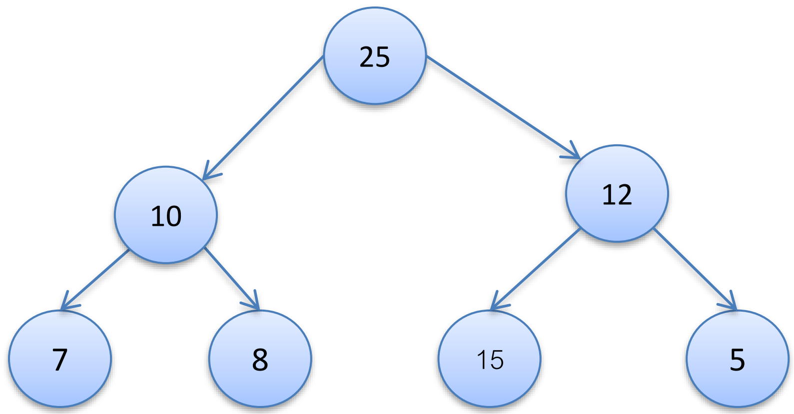


Algorithms and Data Structures 2

TREE TRAVERSALS

Searching an Element in a Tree

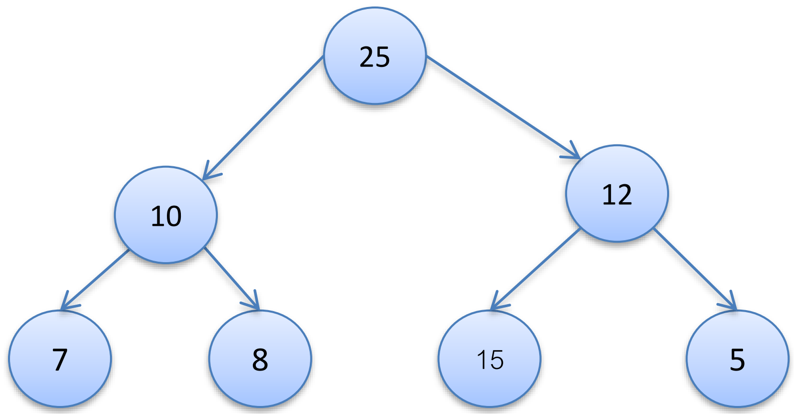
- Example: find element 15 in the given tree



- Possible approaches:
  - Depth-first search (DFS)
  - Breath-first search (BFS)

Breath-First Search

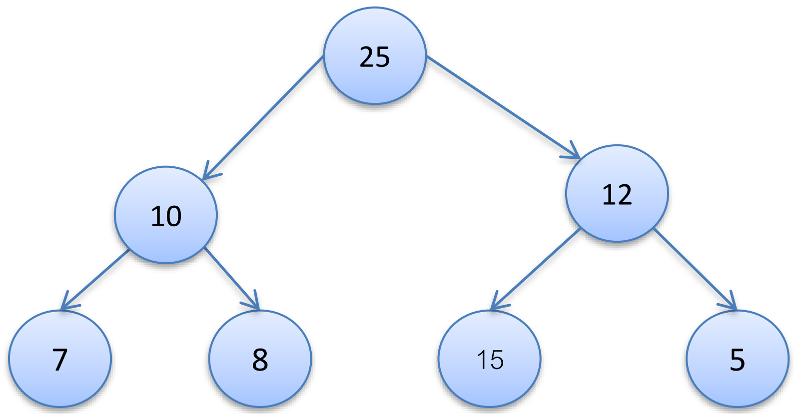
- Example: find element 15 in the given tree



- Traverse all of the nodes on the same level first, and then move on to the next (lower) level

Depth-First Search

- Example: find element 15 in the given tree



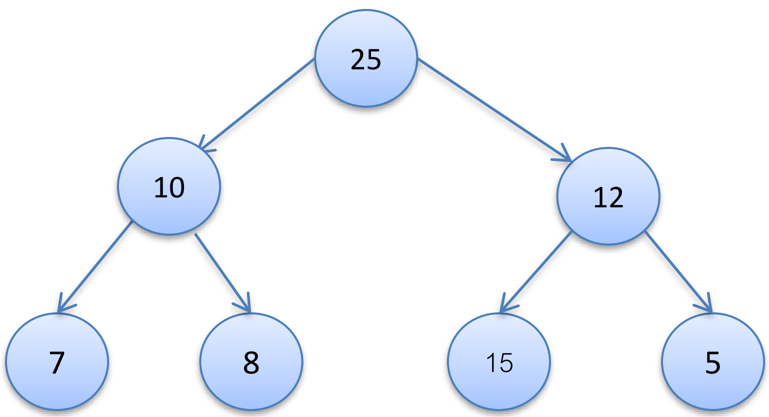
- Traverse one side of the tree all the way to the leaves, followed by the other side

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

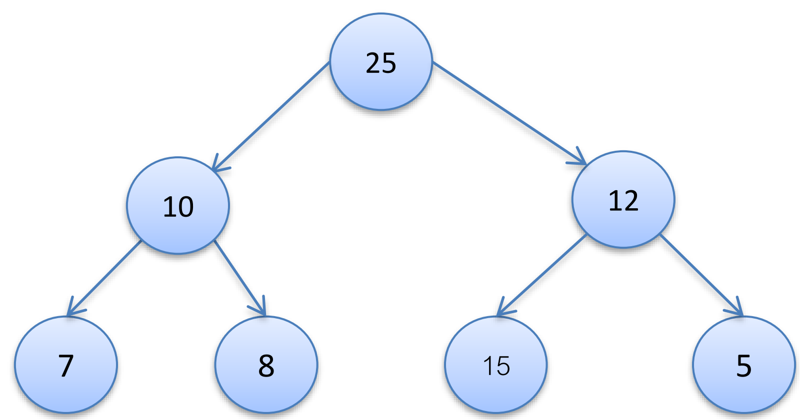
- Common orderings for traversals:
  - **Pre-order** – process root node, then its left/right subtrees
  - **In-order** – process left subtree, then root node, then right subtree
  - **Post-order** – process left/right subtrees, then root node



- 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`

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

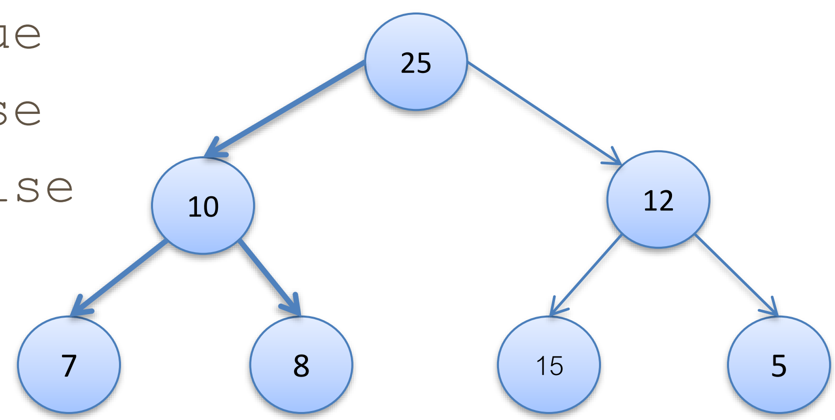
- Tree methods are often implemented recursively with a public/private pair
  - The private version accepts the root node to process

```
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`



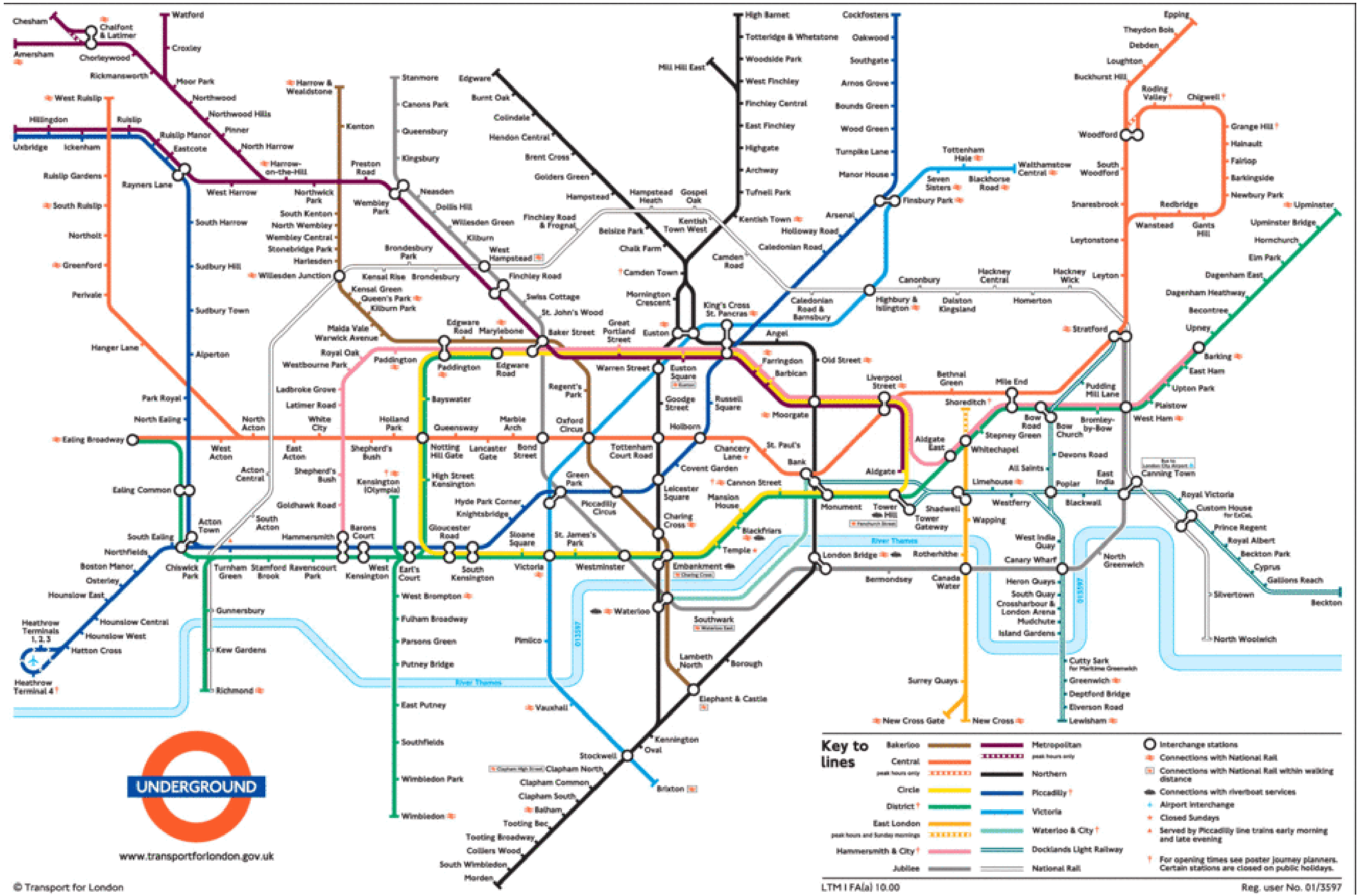
Example: contains()

```
// 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 2

GRAPHS

Graphs



[Picture credit: https://i308.wikispaces.com/file/view/tube_map.gif/58770170/952x628/tube_map.gif]

Graphs

- Graphs are a theoretical framework for understanding certain types of problems
- Some examples problems:
    - Telecommunication networks
    - Distributed systems
    - Information propagation
    - Traffic flow
    - Social networks
    - Propagation of contagious diseases
    - Path finding
    - Resource allocation

Graphs

- Every graph $G(V, E)$ consisting of two sets:
  - Set of vertices, V
  - Set of edges, E

- Every edge, e, is a pair of vertices (v, w)
  - Undirected graph – pair of vertices not ordered
    ((v, w) == (w, v))
  - Directed graph (digraph) – pair of vertices ordered
    ((v, w) != (w, v))

Paths and Cycles in a Graph

- Path - a set of edges connecting two vertices in a graph, where neither edges nor vertices repeat
  - Path **length** – the number of edges in the path

- Cycle - a path that starts and ends on the same vertex
  - Directed acyclic graph (DAG) - a directed graph that has no cycles

Walks, Trails and Circuits in a Graph

- Paths and cycles cannot have repeated vertices or edges

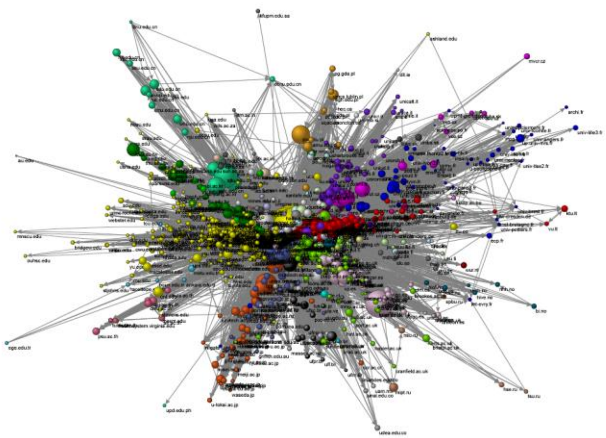
- Walk – a path that can repeat either vertices or edges
- Trail – a path that can repeat vertices, but not edges
- Circuit – a trail that starts and ends at the same vertex

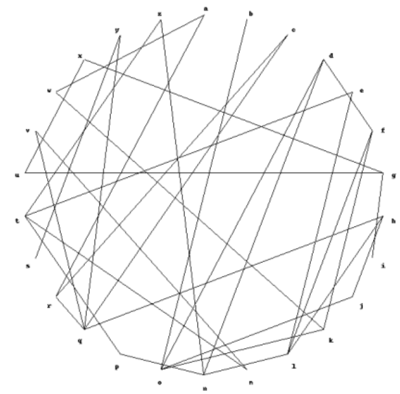
Connected and Complete Graphs

- Connected graph - an undirected graph that has a path from every vertex to every other vertex
- Strongly connected graph - a directed graph that has a path from every vertex to every other vertex
- Weakly connected graph - a directed graph that is not strongly connected, but its underlying undirected graph (without direction to the arcs) is connected
- **Complete graph** - a graph in which there is an edge between every pair of vertices

Graph Density

- We often make determinations about a graph's density
  - Dense graphs - very connected - $O(V^2)$ edges
  - Sparse graphs - less connected, and can be more clustered (each vertex is connected to some smaller number of vertices) – $O(V)$ edges





[Pictures credit: http://internetlab.cchs.csic.es/cv/11/world_map/image003.png, http://livetoad.org/Courses/Documents/132d/Notes/dfs_example.html]

Graph Representations

- **Adjacency Matrix -** a two-dimensional array A, where:
  - If some edge (u, v) exists, set $A[u, v]$ to 1 (true)
  - Else, set $A[u, v]$ to 0 (false)

- If an edge has a weight associated with it, then we can set $A[u, v]$ to the weight
- How to represent non-existing edges?
- Use either a very large or a very small weight as a sentinel to indicate nonexistent edges

- Appropriate representation if a graph is dense

Graph Representations

- For sparse graphs, we typically represent graphs using adjacency lists
- Adjacency List - for each vertex, we keep a list of all adjacent vertices, and possibly their weights
  - Can be implemented using any List data collection (ArrayList, LinkedList)

Graph Traversals

- Since graphs are abstractions similar to trees, we can also perform traversals
  - If a graph is connected, i.e. there is a path between all pairs of vertices, then a traversal can output all nodes if you do it cleverly

- Idea: DFS and BFS
  - Depth first search needs to check which nodes have been output or else it can get stuck in loops
  - In a connected graph, a BFS will print all nodes, but it will repeat if there are cycles and may not terminate

Graph Traversals

- **Why might we want to traverse a graph?**
  - To find all nodes *reachable* from *v* (in social networks, reachable nodes may represents people we're connected to)
  - To process nodes in the graph (example: print out the nodes' value)
  - To determine if an undirected graph is connected (idea: if a traversal goes through all vertices, a graph it is connected)

- **Basic traversal idea:**
  - Traverse through the nodes like a tree
  - Mark the nodes as visited to prevent cycles, and from processing the same node twice

Graph Traversals

- **Basic traversal idea:**
  - Traverse through the nodes like a tree
  - Mark the nodes as visited to prevent cycles, and from processing the same node twice
- Basic idea – pseudocode:

```
void traverseGraph(Node start) {
    Set pending = emptySet()
    pending.add(start)
    mark start as visited
    while(pending is not empty) {
        next = pending.remove()
        for each node u adjacent to next
        if (u is not marked visited) {
            mark u
            pending.add(u)
        }
    }
}
```

Questions?

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