CS 5010: Programming Design Paradigms
Fall 2017

Lecture 9: Data Structures and Algorithms III

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

Tamara Bonaci
t.bonaci@northeastern.edu

Administrivia

- Welcome to lecture 9
- Welcome to the group assignments part of the course
- Assignment 6 posted
  - Due on Monday, November 13 by 6pm
  - Code walks on Tuesday, November 13 (more info likely to follow)
- Your ASAP assignment – choose a team repo
  → (Do it today, during the breaks)

Agenda – Algorithms and Data Structures 3

- Some comments and hints about Assignments 5 and 6
- Hashing and Hash Functions in Java
  - HashTables
  - Collisions, probing and chaining
- Heaps in Java
- Sorting in Java
- Graphs
  - Representations
  - Topological sort
  - DAG
  - Traversals
  - Shortest path algorithms - Dijkstra
- Quick review of the main algorithm design techniques
  - Greedy approach
  - Divide-and-conquer
  - Recursion
  - Backtracking
- Some more trees
  - Red black trees
- Handling events in Java

Algorithms and Data Structures 3

ASSIGNMENTS 5 & 6 – COMMENTS AND HINTS

Assignment 5 – Your Task

- Simulate digital signature verification process for some fictional bank and its unique clients
- Simulator takes several input arguments:
  - Number of unique bank clients
  - Number of distinct transactions
  - Fraction of invalid messages
  - Output files

Assignment 6 – Background

- Social networks – important to continue growing:
  - Number of total users
  - Number of active monthly users
- Recommendation systems – an important component of maintaining a growth of a social network

Assignment 6 – Your Task

- Build a simple recommendation system for a Twitter-like network

- Network represented as a directed graph
  - Nodes – users
  - Directed edges (x, y) – user x follows user y

Assignment 6 – Your Task

- Data set – two files:
  - `nodes.csv` – node ID, date profile created, gender, age, city
  - `edges.csv` – source node (follower), destination node (followee)

Assignment 6 – Your Task

- Four ordered recommendation criteria:
  - Newbies mimic a friendliest friend
  - Friend of a friend is a friend
  - Always follow the influencer
  - When in doubt, branch out

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

Algorithms and Data Structures 3

HASHING AND HASH FUNCTIONS

Introduction to Hashing

- Suppose we have a large set of items *M*
- Set *M* contains the subset *D*, where *D* << *M* of items that we actually care about
- Example: *M* may be English dictionary, and *D* maybe a set of English words we use in everyday life
- Problem: How to store data such that we use only O(*D*) memory, while achieving fast O(1) access

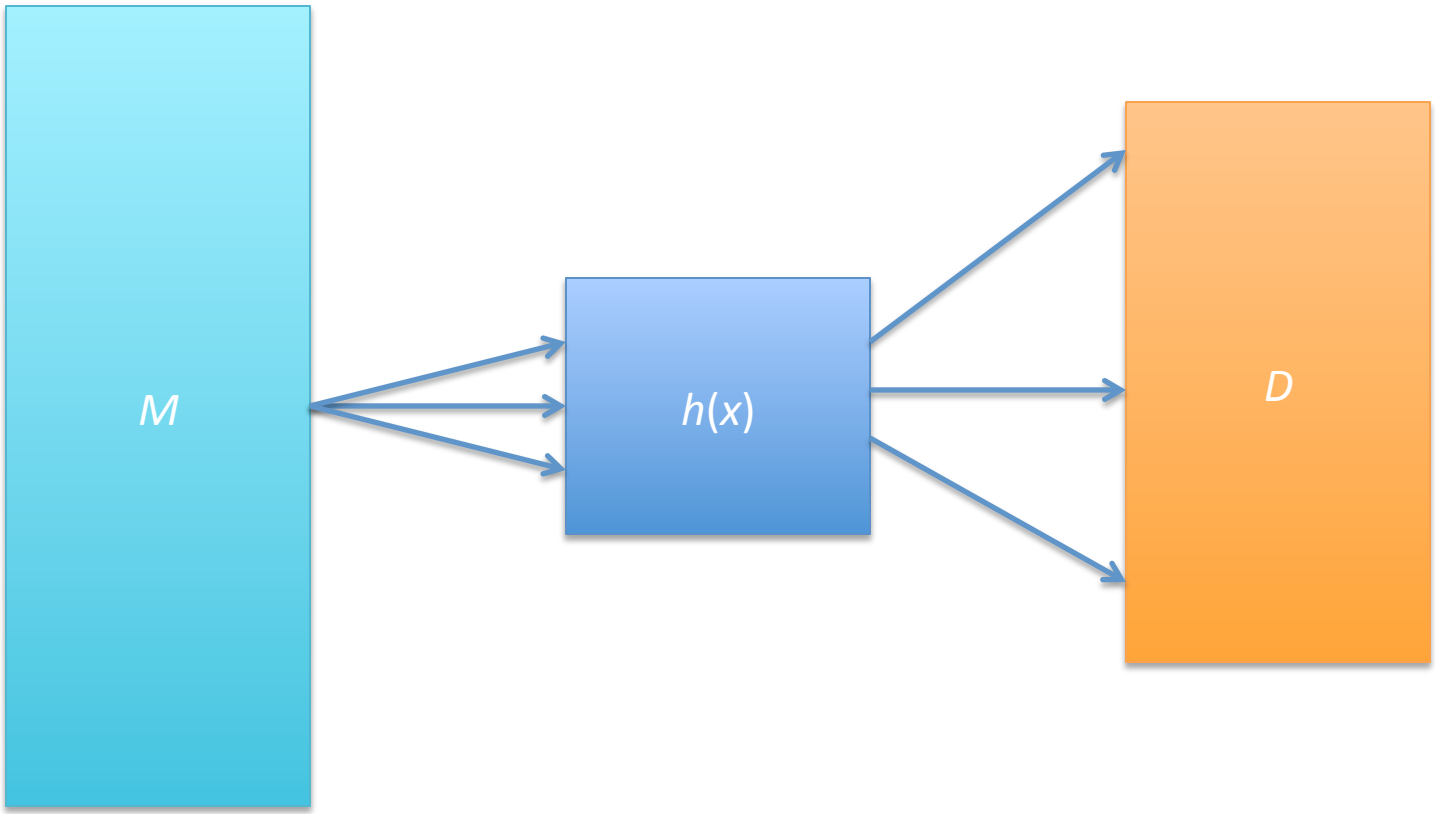
Introduction to Hashing

- Memory: A Hash Table
  - Consider an array of size c * *D*
  - Each index in the array corresponds to *some* element in *M* that we want to store
  - The data in *D* does not need any particular ordering

- Possible approaches:
  - Unsorted array – search takes O(*D*)
  - Sorted array – search still takes O(*D*)
  - Random mapping – search still takes O(*D*)

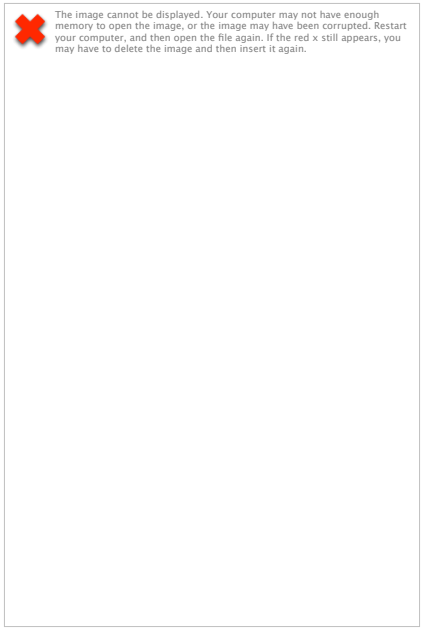
Introduction to Hashing

- 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*



[Pictures credit: https://upload.wikimedia.org/wikipedia/commons/3/37/African_Bush_Elephant.jpg https://aos.iacpublishinglabs.com/question/aq/1400px-788px/what-are-elephant-tusks-used-for_3c174bec-bd85-4fab-a617-7a1a33f14c62.jpg?domain=cx.aos.ask.com]

Introduction to Hash Functions

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

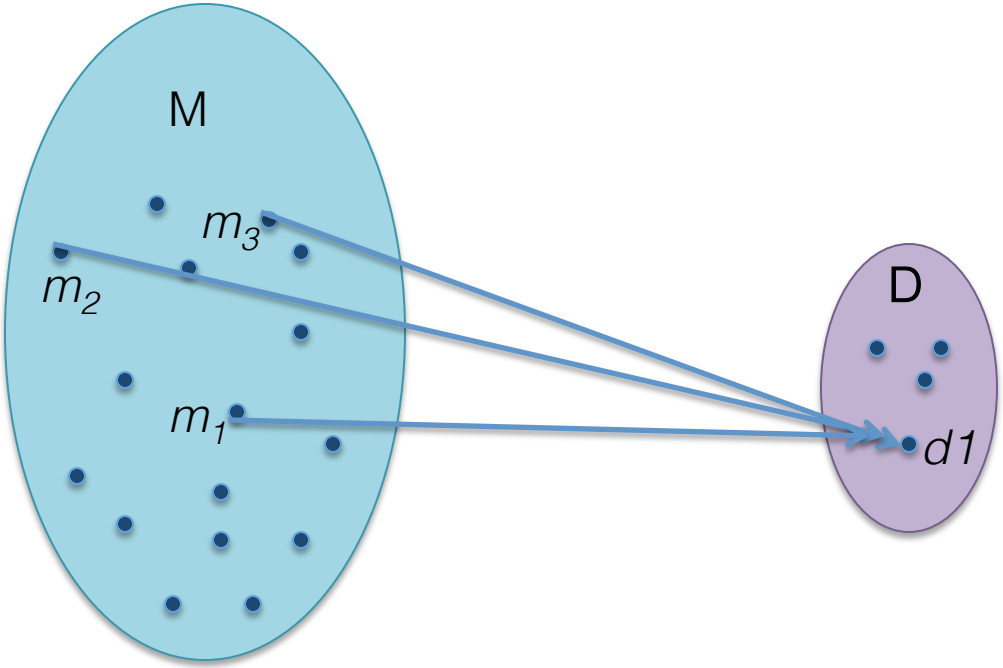
- Desired properties of hash functions:
  - Repeatability:
    - For every *x* in *D*, it should always be h(*x*) = h(*x*)
  - Equally distributed:
    - For some *y*, *z* in *D*, P(h(y)) = P(h(z))
  - Constant-time execution: h(*x*) = O(1)

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(inti=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", since |*M*| << |*D*|



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

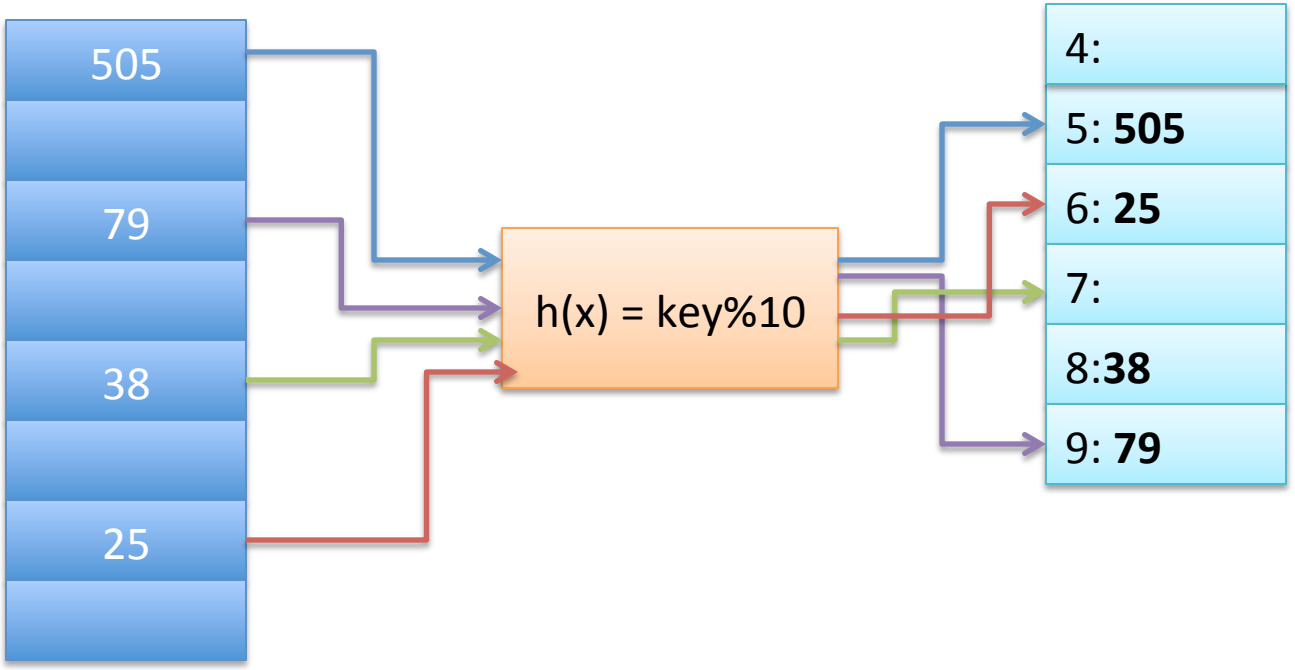
Resolving Collisions

- Hash function – can be thought of as a "lossy compression function", since $|M| << |D|$
- Problem – collision
  → a hash function maps multiple values into to same index

- 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

Collision Resolutions – Linear Probing

- Probing – general approach of resolving a collision by moving to another index
- Linear probing – when a collision occurs, find the next available index in the array



Problems with Linear Probing

- Linear probing – 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

- Possible negative consequences of large clusters:
  - The chances of collision increase
  - The time it takes to find something in the cluster increases, and it **is not O(1) anymore**

Quadratic Probing

- Linear probing updates colliding indices linearly (increment of one each time)
- Quadratic probing updates colliding indices as squares
- For example, linear probing would check index 5, then:
  - 5+1,
  - 5+2,
  - 5+3,
  - 5+4 etc.
- Quadratic probing would check index 5, then
  - 5+1,
  - 5+4,
  - 5+9,
  - 5+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+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:
  - Quadratic probing is not guaranteed to find an opening, even when there is space available
  - Moreover, 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

Double Hashing

- Double hashing – in a case of a collision, apply a second hash function, and probe at a distance determine by the second hash function

- Important:
  - The second hash function must never evaluate to zero
  - All indexes should be reachable

Chaining

- Rather than probing for an open position, we save multiple objects in the same position
- Some data structure is necessary here, and commonly we use:
  - A linked list,
  - AVL tree, or
  - A secondary hash table

- Problem: how much time to find an element?

Chaining

- Problem: how much time to find an element?

- Hash table load factor, λ - the ratio of the number of elements in the hash table and the table size
  → the average length of a list is λ

- Time needed to find an element:
  - Hash function evaluation – O(1)
  - Traversing the list – O(λ)
  - → Total time: O(1 + λ)

→ The table size is not really important, but the load factor is

Rehashing

- If the table gets too full, running time for the operations becomes too long

- Approach – rehashing:
  - Build another table that is about twice as big, with its associated new hash function
  - Computing the new hash value for each (non-deleted) element of the original table
  - Insert it in the new table

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 3

HEAPS AND PRIORITY QUEUES

Heaps and Priority Queues

[Picture credit: http://3.bp.blogspot.com/_TIZSdQwlCic/TQcpm7K8aSI/AAAAAAAAATE/wipxP2Y3FX0/s1600/n.JPG]

Prioritization Problems

- Emergency response: an emergency (911) dispatcher having to respond to several phone calls:
  - Cat stuck on the tree
  - A non-injury fender-bender
  - An ongoing robbery

- Load-balancing in a cell-phone network: providing an upload/download speed to:
  - Commercial customers
  - Private customers
  - Customers from other networks (e.g., roaming)
- What would be the runtime of solutions to these problems using the data structures we know (list, sorted list, map, set, BST, etc.)?

Some Inefficient Approaches

- List:
  - Store emergency cases/customers in a list
  - Remove min/max by searching (O(N))
  - Problem: expensive to search
- Sorted list:
  - Store in sorted list
  - Binary search it in O(log N) time
  - Problem: expensive to add/remove (O(N))
- Binary Search Tree:
  - Store in BST
  - Go right for max in O(log N)
  - Problem: tree becomes unbalanced

Priority Queue ADT

- Priority queue - a collection of ordered elements that provides fast access to the minimum (or maximum) element

- Priority queue operations:
  - `add` – adds in order (worst time complexity O(log N))
  - `peek` – returns minimum value (time complexity always O(1))
  - `remove` – returns/removes the minimum value (worst O(log N)
  - `isEmpty, clear, size, iterator` - (always O(1))

Java `PriorityQueue` Class

```
public class PriorityQueue<E> implements Queue<E>
```

- Example:

```
Queue<String> priorityQueue = new
PriorityQueue<String>();
priorityQueue.add(”Daniel");
priorityQueue.add(”Emily");
```

Java `PriorityQueue` Class

Method/constructor	Description	Runtime
`PriorityQueue<E>()`	constructs new empty queue	O(1)
`add(E value)`	adds value in sorted order	O(log N)
`clear()`	removes all elements	O(1)
`iterator()`	returns iterator over elements	O(1)
`peek()`	returns minimum element	O(1)
`remove()`	removes/returns min element	O(log N)
`size()`	number of elements in queue	O(1)

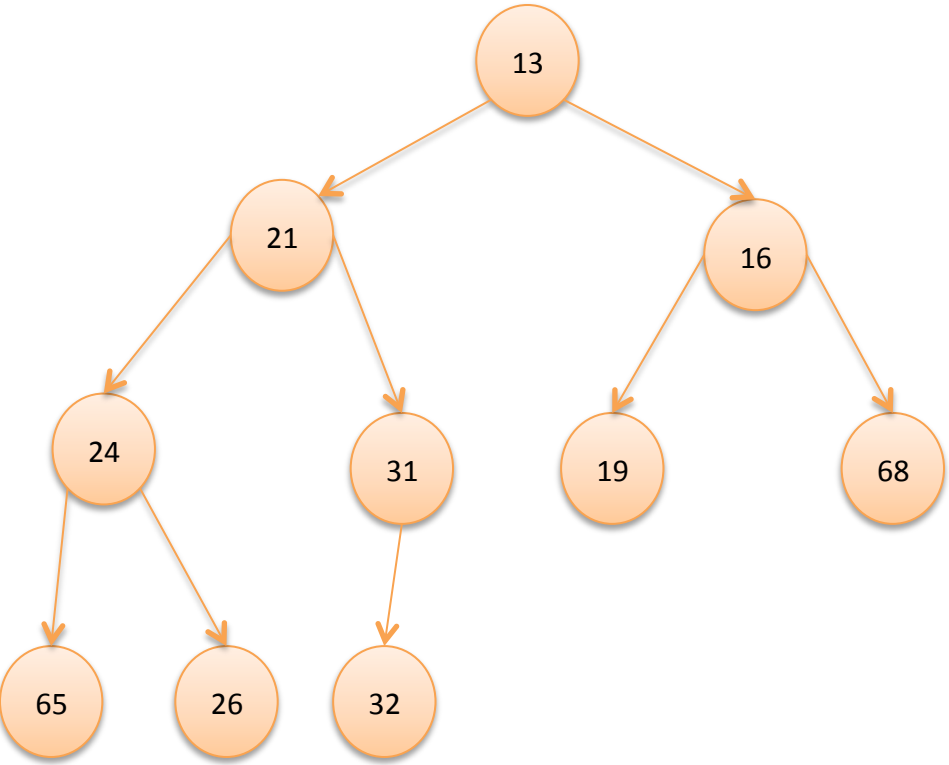
Inside a Priority Queue

- Priority queue usually implemented as a **heap**

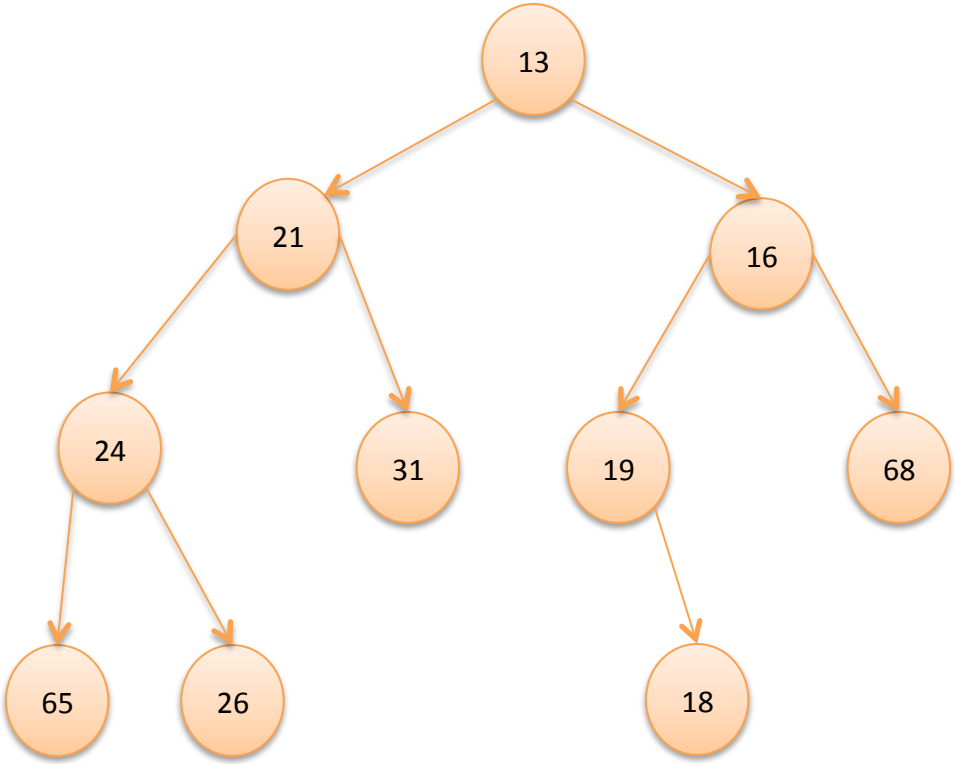
- Heap – a binary tree that satisfies:
  - The completeness property - completely filled, with the possible exception of the bottom level, which is filled from left to right
  - The heap order property:
    - Minimum element on the top
    - Tree is sorted top to bottom
    - Every child is guaranteed to have a lower priority than its ancestors

Priority Queue Example

- Are these heaps?



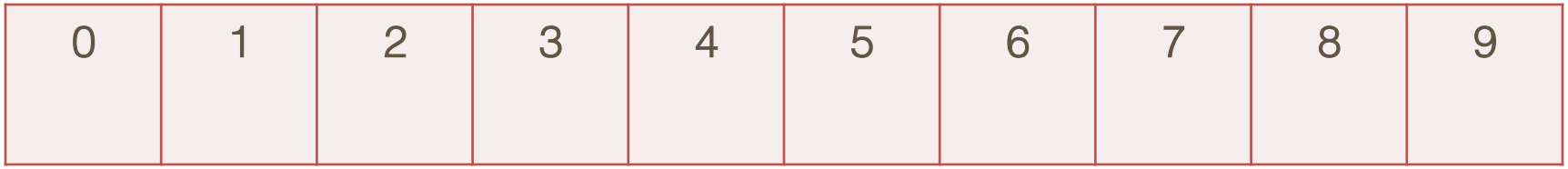
Yes!



No!

Heaps and Arrays

- Completeness + heap order – can be implemented using an array
  - Insert into array from left to right
  - For any parent at index i, children at 2*i+1 and 2*i+2



Heaps and Arrays

- Array property for heaps – 0 vs. 1 indexing

- 0 indexing:
  - Left child at 2*i+1
  - Right child at 2*i+2
  - Parent at (i-1)/2

- 1 indexing:
- Left child at 2*I
- Right child at 2*i +1
- Parent at i/2

Basic Heap Operations

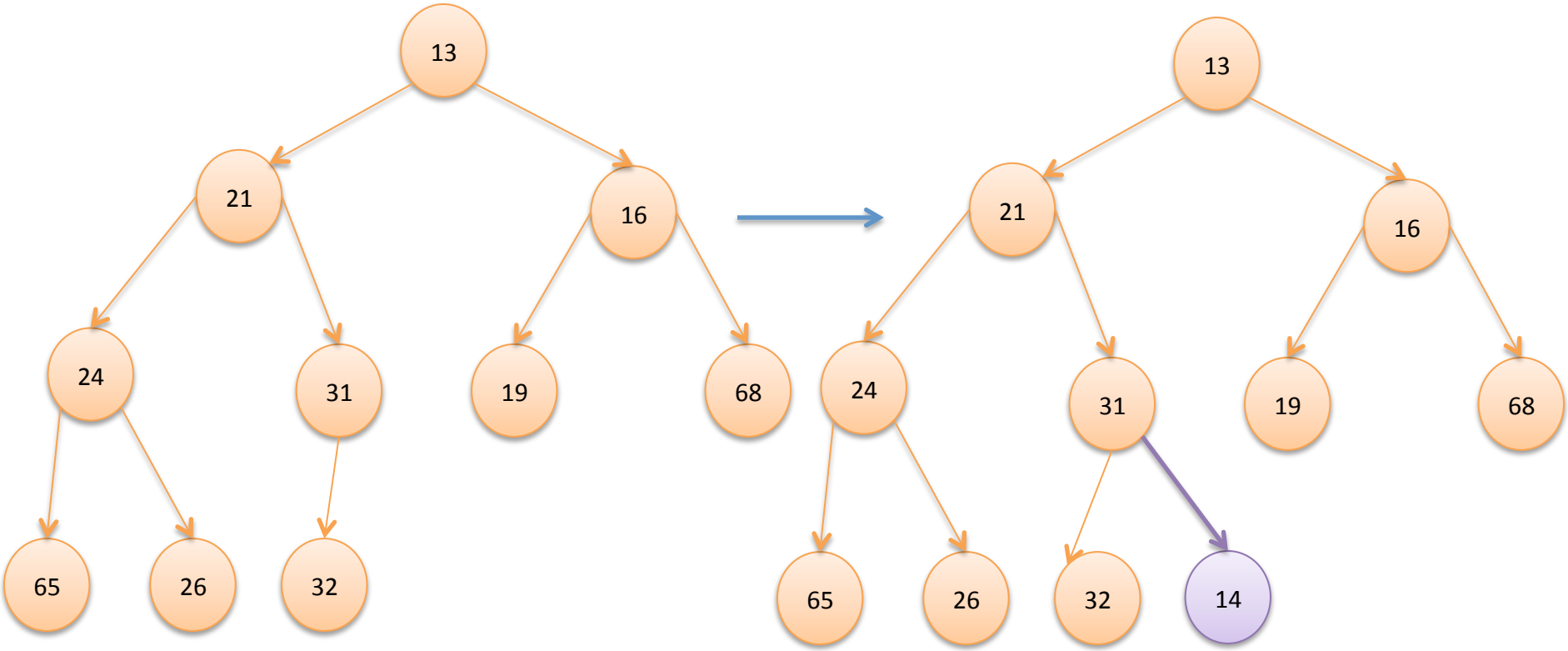
- `insert()` – add a data, priority pair into the heap
- `deleteMin()` – return and remove the item of smallest priority from the heap
- `changePriority()` – change the priority of a particular item in the heap

Method `insert()`

- To insert some element *X* into the heap, we use the percolate up approach:
  - Place the item at the next position to preserve completeness
  - Swap the item up the tree until it is larger than its parent

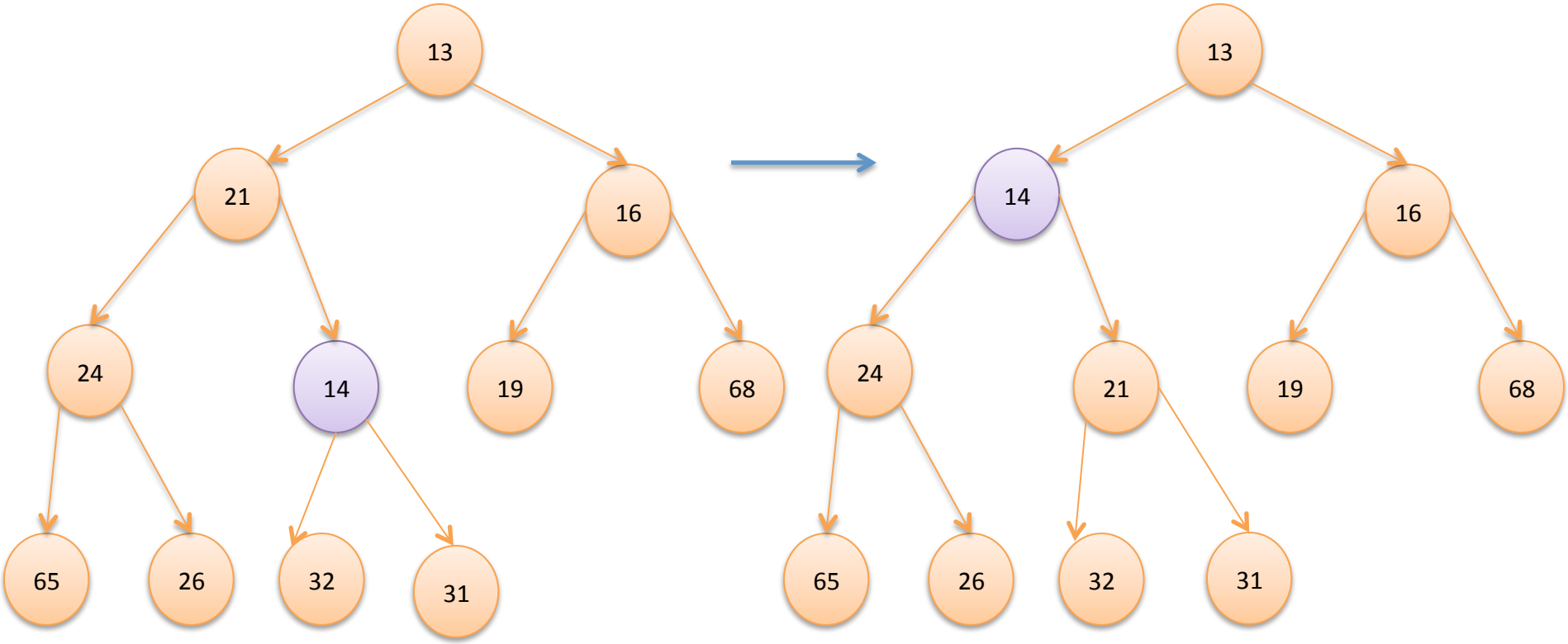
Method `insert()` Example

- Example: add 14



Method `insert()` Example

- Example: add 14



Method `insert()`

```
/**
* Insert into the priority queue, maintaining heap order.
* Duplicates are allowed.
* @param x the item to insert.
 */

public void insert( T x ) {
    if (currentSize == array.length - 1)
        enlargeArray(array.length * 2 + 1);

    // Percolate up
    int hole = ++currentSize;
    for(array[0] = x; x.compareTo(array[ hole / 2 ] ) < 0;
hole /= 2 )
        array[ hole ] = array[ hole / 2 ];
    array[ hole ] = x;
```

Method `deleteMin()`

- To remove the minimum element from the heap, we use the percolate down approach:
  - Remove the root of the tree (the minimum element)
  - Move the last object in the tree to the root
  - Swap the moved piece down while it is larger than its smallest child
  - Only swap with the smallest child

Method `deleteMin()`

```
/** * Remove the smallest item from the priority queue.
* @return the smallest item, or throw UnderflowException,
if empty.
*/
public T deleteMin( ) {
  if( isEmpty( ) )
  throw new UnderflowException( );

  T minItem = findMin( );
  array[1] = array[currentSize-- ];
  percolateDown( 1 );

return minItem;
}
```

Method `deleteMin()`

```
/** * Internal method to percolate down in the heap.
* @param hole the index at which the percolate begins.
*/
private void percolateDown(int hole)  {
    int child;
    T temp = array[hole];

    for(; hole* 2 <= currentSize; hole = child) {
        child = hole*2;
        if(child != currentSize && array[child +
1].compareTp(array[child]) < 0)
            child++;
        if(array[child].compareTo(tmp) < 0)
            aray[hole] = array[child];
        else
            break;
    }
    array[hole] = tmp;
}
```

Method `buildHeap()`

- A binary heap is sometimes constructed from an initial collection of *N* items

- Naïve approach: *N* successive inserts

- Every insert takes:
  - O(1) average-case time
  - O(log *N*) worst- case time

→ the total running time of this algorithm would be O(*N*) average, but O(*N* log *N*) worst-case

- Can we do better? Can we achieve O(*N*) worst-case?

Floyd's Algorithm

- Floyd's Algorithm:
  - Traverse the tree from bottom to top
    - (Reverse order in the array)
  - Start with the last node that has children
  - Percolate down each node as necessary

- Wait! Percolate down is O(log *N*)! This is an O(*N* log *N*) approach!

- Or is it?

Floyd's Algorithm

- Floyd's Algorithm:
  - Traverse the tree from bottom to top
    - (Reverse order in the array)
  - Start with the last node that has children
  - Percolate down each node as necessary

- It is O(*N* log *N*), because big O is an upper bound, but there is a tighter analysis possible!
- How far does each node travel (at worst)?
  - 1/2 of the nodes don't move:
  - These are leaves – Height = 0
  - 1/4 can move at most one
  - 1/8 can move at most two ...

Floyd's Algorithm

- It is O(*N* log *N*), because big O is an upper bound, but there is a tighter analysis possible!
- How far does each node travel (at worst)?
  - 1/2 of the nodes don't move:
  - These are leaves – Height = 0
  - 1/4 can move at most one
  - 1/8 can move at most two ...

$$\sum_{i=0}^{n} \frac{i}{2^{i+1}} = 2^{-n-1}\left(-n + 2^{n+1} - 2\right)$$

$$\sum_{i=0}^{\infty} \frac{1}{2^{i+1}} = 1$$

Floyd's Algorithm

- It is O(n log n), because big O is an upper bound, but there is a tighter analysis possible!
- How far does each node travel (at worst)?
  - 1/2 of the nodes don't move:
  - These are leaves – Height = 0
  - 1/4 can move at most one
  - 1/8 can move at most two ...

$$\sum_{i=0}^{\infty} \frac{1}{2^{i+1}} = 1$$

- What is the right summation? Our original summation plus 1
- This means that the number of swaps we perform in Floyd's method is 2 times the size... So Floyd's method is O(N)

Floyd's Algorithm

```
/**
* Construct the binary heap given an array of items.
* /
public BinaryHeap(T[] items ) {
    currentSize = items.length;
    array = (T[]) new Comparable[(currentSize + 2)*1.1];

    int i=1;
    for(T item : items)
        array[i++] = item;

    buildHeap();

}
```

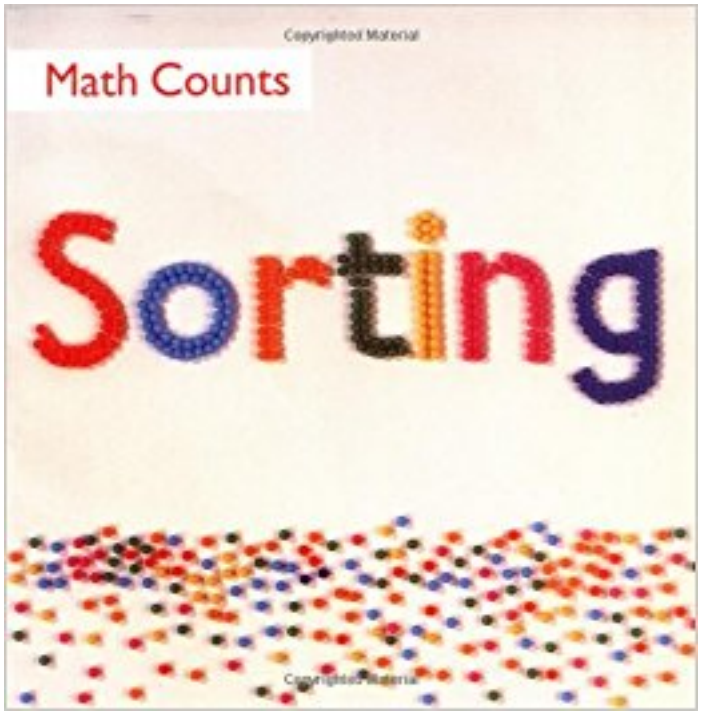
Floyd's Algorithm

```
/**
* Establish heap order property from an arbitrary
arrangement of items. Runs in linear time.
*/

private void buildHeap() {
    for( int i = currentSize / 2; i > 0; i-- )
        percolateDown(i);
}
```

Algorithms and Data Structures 3

SORTING

Sorting



[Picture credit: https://images-na.ssl-images-amazon.com/images/I/51G8A3-F%2BHL._SX258_BO1,204,203,200_.jpg]

Sorting

- Sorting – process of rearranging the values in a data collection into a specific order (usually into their "natural ordering")

- One of the fundamental problems in computer science – *can be solved in many ways*
- Many sorting algorithms, and they differ in their:
  - Time complexity
  - Memory complexity
  - Type of data they can process (fast)
  - Ability to utilize multiple computers / processors, ...
- Comparison-based sorting - order determined by comparing pairs of elements
  - Object of type `Comparable`, compared using method `compareTo()`

Some Definitions

- In-Place Sort – a sorting algorithm that requires only O(1) extra space to sort the array
  - Usually modifies input array
  - Can be useful: let's us minimize memory

- Stable Sort - a sorting algorithm where any equal items remain in the same relative order before and after the sort
  - Items that 'compare' the same might not be the exact duplicates
  - Might want to sort on some, but not all attributes of an item
  - Can be useful to sort on one attribute first, then another one

Stable Sort Example

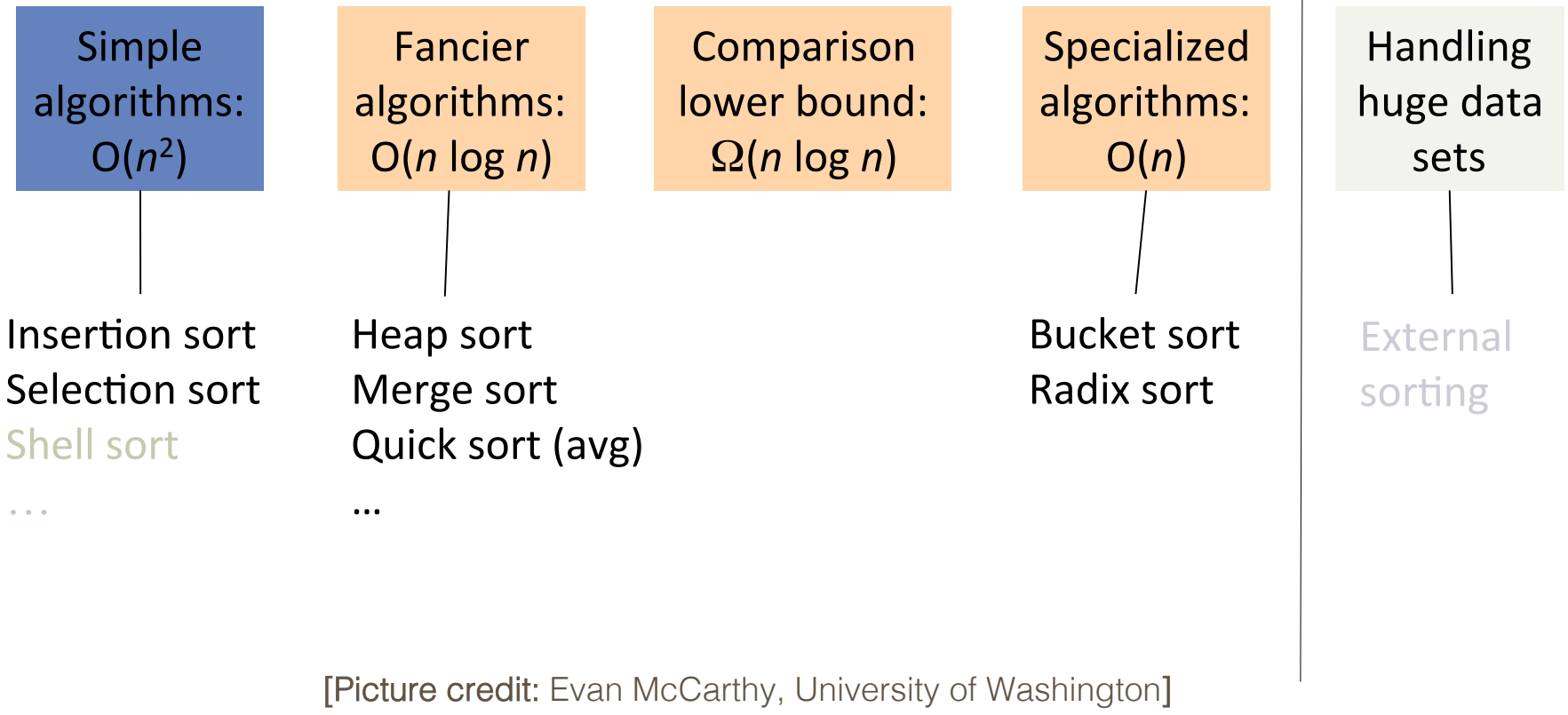
- Input:
  [(8, "fox"), (9, "dog"), (4, "wolf"), (8, "cow")]
- Compare function: compare pairs by number only
- Output (stable sort):
  [(4, "wolf"), (8, "fox"), (8, "cow"), (9, "dog")]
- Output (unstable sort):
  [(4, "wolf"), (8, "cow"), (8, "fox"), (9, "dog")]

Sorting Methods in Java

- Classes `Arrays` and `Collections` in `java.util` have a static method `sort` that sorts the elements of an array/list
- Example:

```
String[] words = {”PDP", ”Fall", ”Class", ”Students"};
Arrays.sort(words);
System.out.println(Arrays.toString(words));
// [Class, Fall, PDP, Students]
List<String> words2 = new ArrayList<String>(); for
(String word : words) {
words2.add(word); }
Collections.sort(words2);
System.out.println(words2);
// [Class, Fall, PDP, Students]
```

Sorting: Big Picture



[Picture credit: Evan McCarthy, University of Washington]

Some Sorting Algorithms

Algorithm	Description	Worst –case Time complexity
Bubble sort	Swap adjacent pairs that are out of order	$O(N^2)$
Selection sort	Look for the smallest element, move to front	$O(N^2)$
Insertion sort	Build an increasingly large sort front portion	$O(N^2)$
Merge sort	Recursively divide the array in half and sort it	$O(N \log N)$
Heap sort	Place the values into a sorted tree structure	$O(N \log N)$
Quick sort	Recursively partition array based on a middle value	$O(N^2)$
Bucket sort	Cluster elements into smaller groups, and sort them	$O(N^2)$
Radix sort	Sort integers by last digit, then 2nd to last, etc.	$O(N k)$

Selection Sort

- Selection sort - orders a list of values by repeatedly putting the smallest (or the largest) unplaced value into its final position

- The algorithm:
  - Look through the list to find the smallest value
  - Swap it so that it is at index 0
  - Look through the list to find the second-smallest value
  - Swap it so that it is at index 1
  - Repeat until all values are in their proper places

.

Selection Sort

```
// Rearranges the elements of a into sorted order
using the selection sort algorithm.
public static void selectionSort(int[] a) {
   for (int i = 0; i < a.length - 1; i++) {
      // find index of smallest remaining value
      int min = i;
      for (int j = i + 1; j < a.length; j++) {
         if (a[j] < a[min])
            min = j;
      }
   // swap smallest value its proper place, a[i]
   swap(a, i, min);
   }
}
```

Selection Sort

```
// Swaps a[i] with a[j].
public static void swap(int[] a, int i, int j) {
   if (i != j) {
      int temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

- Similar algorithms:
  - Bubble sort - make repeated passes, swapping adjacent values → *slower than selection sort (has to do more swaps)*
  - Insertion sort - shift each element into a sorted sub-array → *faster than selection sort (examines fewer values)*

Heap Sort

- Heap sort – sorts elements using priority queues
- The algorithm:
  - Build a binary heap of *N* elements
  - Perform *N* **`deleteMin`** operations
  - The elements leave the heap smallest first, in sorted order
- Time complexity:
  - BuildHeap() – O(*N*)
  - *N* deleteMin() operations – O(*N* log *N*)
- The main problem: doubled memory requirement (the algorithm uses an extra array)

Heap Sort

```
/**
 * Internal method for heapsort.
 * @param i the index of an item in the heap.
 * @return the index of the left child.
 */

private static int leftChild (int i) {
     return 2* i + 1;
}

/** * Internal method for heapsort that is used in deleteMax and buildHeap. * @param a an array
of Comparable items.
* @int i the position from which to percolate down.
* @int n the logical size of the binary heap.
*/

private static <Textends Comparable<? super T>> void percDown( AnyType[]a,inti,intn)
{
     int child;
     T tmp;
     for(tmp=a[i];leftChild(i)<n;i=child) {
          child = leftChild(i);
          if(child != n-1 && a[child].compareTo(a[child + 1]) < 0)
               child++;
          if( tmp.compareTo(a[child]) < 0 )
               a[i]=a[child];
          else
          break;
}
     a[i]=tmp;
}
```

Heap Sort

```
/**
* Standard heapsort.
* @param a an array of Comparable items.
*/

public static <T extends Comparable<? super T>> void
heapsort(T[] a) {
    for(inti=a.length/2 - 1; i >= 0; i--)
        /*buildHeap*/
        percDown( a, i, a.length );
    for( int i = a.length - 1; i > 0; i-- ) {
        swapReferences( a, 0, i );
        percDown( a, 0, i );
    }
}
```

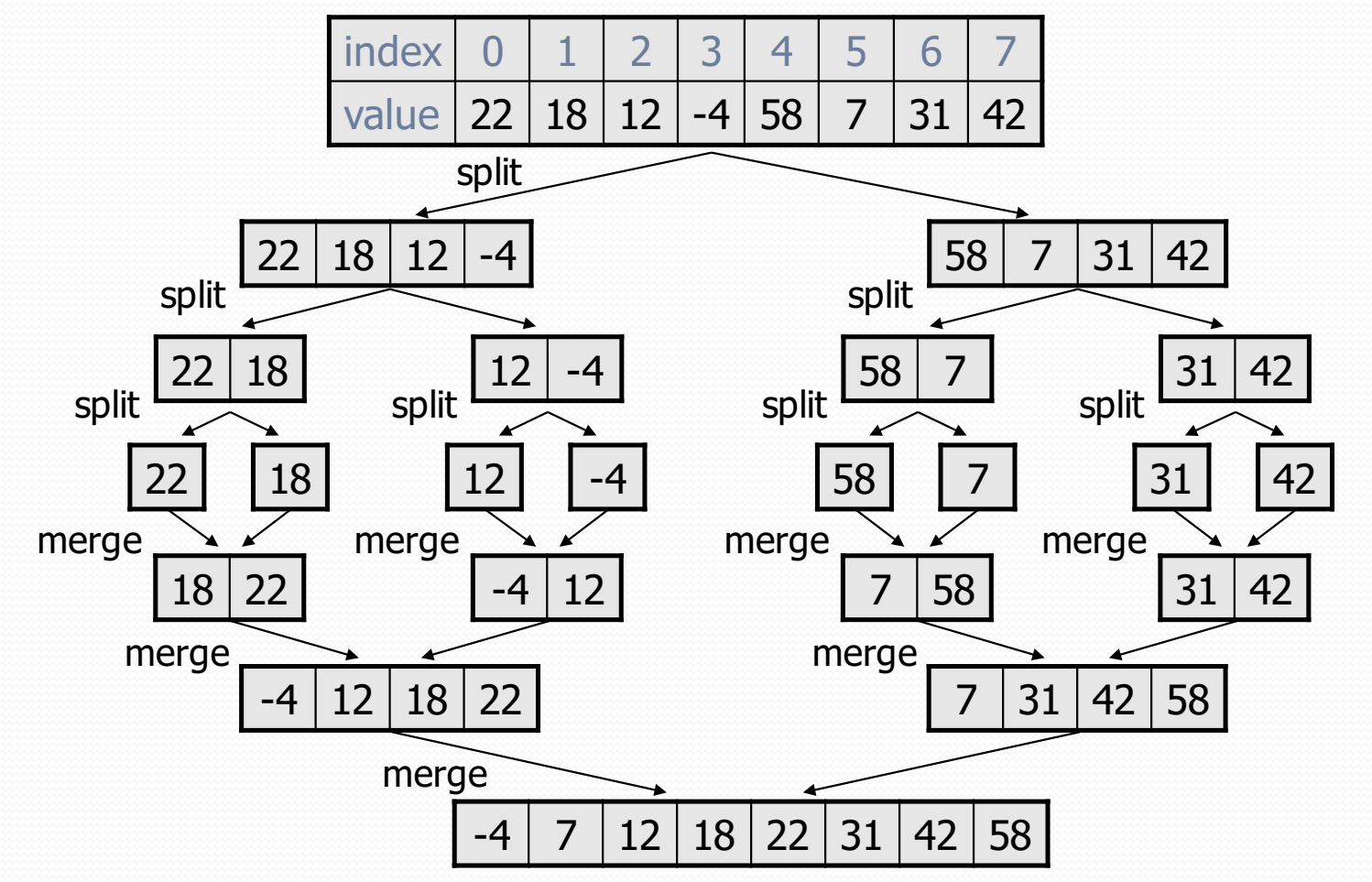
Merge Sort

- Merge sort - repeatedly divides the data in half, sorts each half, and combines the sorted halves into a sorted whole

- An example of a "divide and conquer" algorithm
  - Invented by John von Neumann in 1945

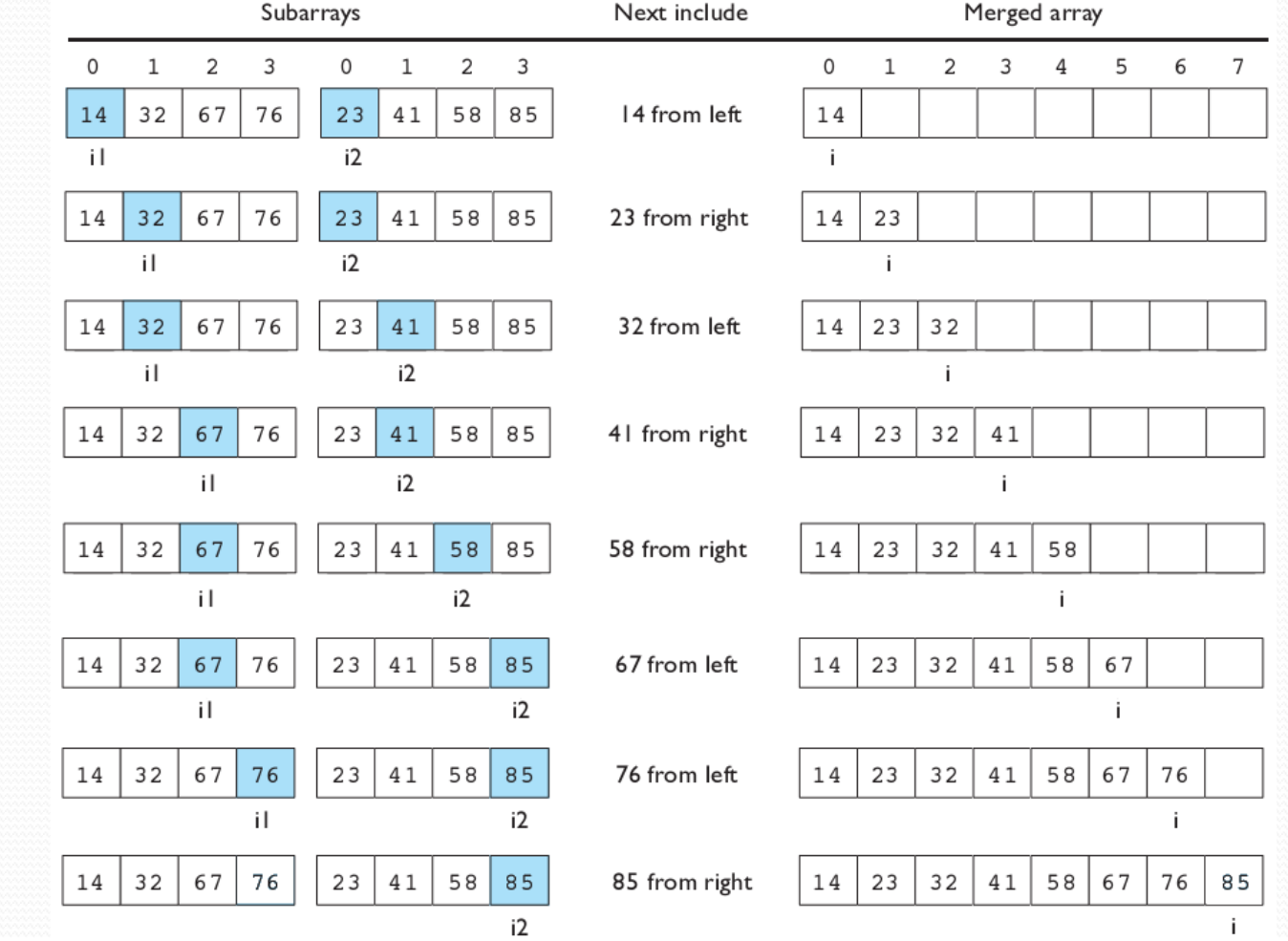
- The algorithm:
  - Divide the collection into two roughly equal halves
  - Sort the left half
  - Sort the right half
  - Merge the two sorted halves into one sorted list

Merge Sort Example



[Example credit: Zorah Fung, University of Washington]

Merge Sort Example



[Example credit: Zorah Fung, University of Washington]

Merge Sort – Code For Merging Halves

```
/**
* Merges the left/right elements into a sorted result.
* Precondition: left/right are sorted
*
*/
public static void merge(int [] result, int[] left,  int[] right) {
    int i1 = 0; // index into left array
    int i2 = 0; // index into right array
    for (int i = 0; i < result.length; i++) {
        if (i2 >= right.length || (i1 < left.length && left[i1] <=
right[i2]))  {
            result[i] = left[i1]; // take from left
            i1++;
        } else {
            result[i] = right[i2]; // take from right
            i2++;
        }
    }
}
```

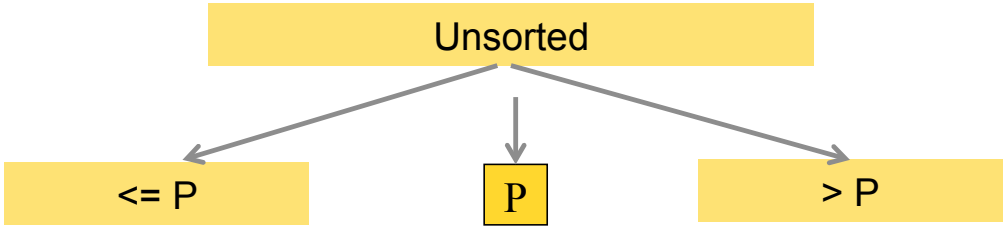
Merge Sort Code

```
/**
*/ Rearranges the elements of a into sorted order using the merge sort
* algorithm.
*/

public static void mergeSort(int[] a) {
    // split array into two halves
    int[] left = Arrays.copyOfRange(a, 0, a.length/2);
    int[] right = Arrays.copyOfRange(a, a.length/2, a.length);

    // sort the two halves
    mergeSort(left);
    mergeSort(right);

    // merge the sorted halves into a sorted whole
    merge(a, left, right);
}
```

Quick Sort

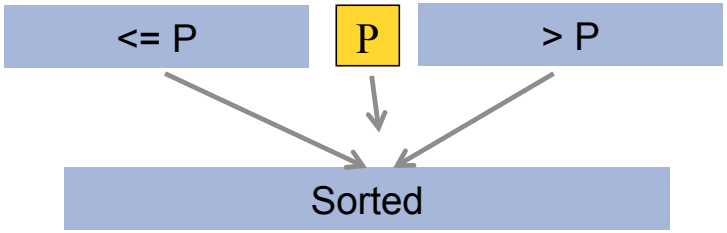
- Another divide-and-conquer algorithm:

Divide: Pick a pivot, partition into groups



Conquer: Return array when length ≤ 1

Combine: Combine sorted partitions and pivot



[Picture credit: Evan McCarthy, University of Washington]

Quick Sort

- Another divide-and-conquer algorithm

- Idea:
  - Pick a pivot item from the array
  - Put all items smaller than the pivot into one group
  - Put all items larger in the other group
  - Recursively sort groups
  - If groups has size 0 or 1, just return them unchanged

Quick Sort

- Another divide-and-conquer algorithm

- Pseudocode:

```
quicksort(input) {
   if (input.length < 2){
   return input;
   }else{
      pivot = getPivot(input);
      smallerHalf = sort(getSmaller(pivot,input));
      largerHalf = sort(getBigger(pivot,input));
      return smallerHalf + pivot + largerHalf;
   }
}
```

Quick Sort - Details

- How do we pick the pivot element?
  - Any choice is correct: data will end up sorted
  - But ideally we want the two partitions to be about equal in size
- How to implement partitioning that is…
  - In linear time
  - In place

Potential Pivot Rules

- While sorting **`arr`** from **`lo`** (inclusive) to **`hi`** (exclusive):

- Pick **`arr[lo]`** or **`arr[hi-1]`**
  - Fast, but worst-case occurs with mostly sorted input
- Pick random element in the range
  - Works as well as any technique, but (pseudo)random number generation can be slow
  - Still probably the most elegant approach
- Median of three values: , e.g., **`arr[lo], arr[hi-1], arr[(hi+lo)/2]`**
  - Common heuristic that tends to work well

Partitioning

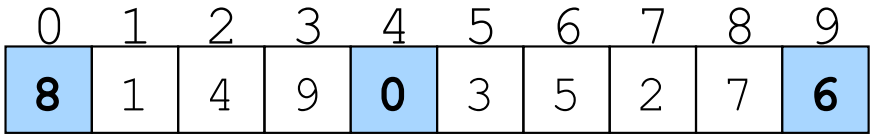
- Conceptually simple, but hardest part to code up correctly
- After picking pivot, we need to partition in linear time in place
- One possible approach (there are slightly fancier ones):
  1. Swap pivot with **`arr[lo]`**
  2. Use two counters **`i`** and **`j`**, starting at **`lo+1`** and **`hi-1`**
  3. **`while (i < j)`**
     `if (arr[j] > pivot)`
     `j-`
     `else if (arr[i] < pivot)`
     `i++`
     `else`
     `swap arr[i] with arr[j]`
  4. Swap pivot with **`arr[i] *`**

- *skip step 4 if pivot ends up being least element

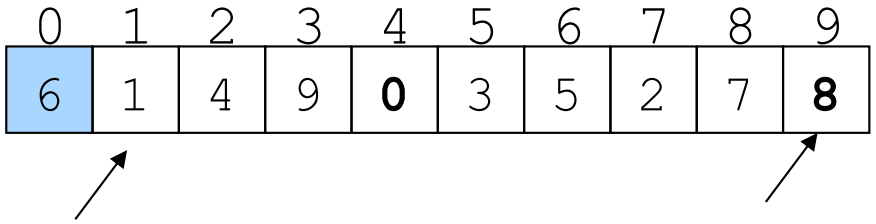
Quick Sort Example

Step one: pick pivot as median of 3

- **`lo`** = 0, **`hi`** = 10



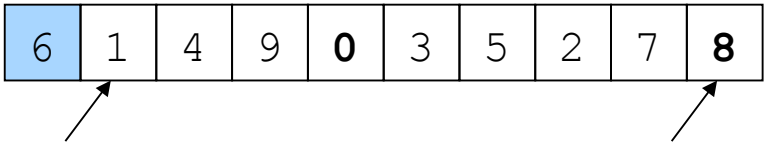
- Step two: move pivot to the `lo` position



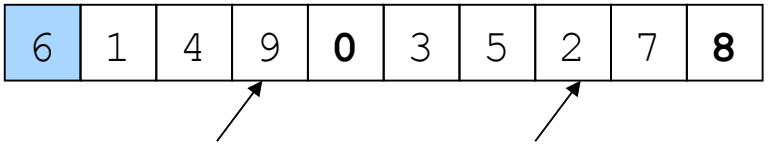
[Example credit: Evan McCarthy, University of Washington]

Quick Sort Example

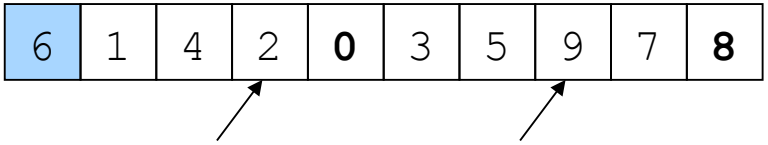
Now partition in place



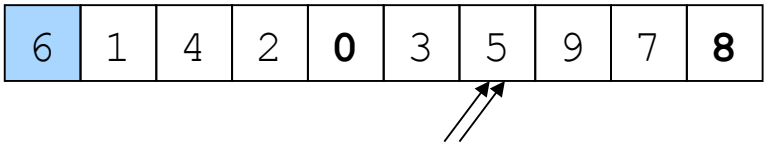
Move cursors



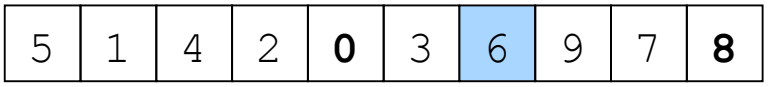
Swap



Move cursors



Move pivot



[Example credit: Evan McCarthy, University of Washington]

Quick Sort Implementation

```
public static void sort(List<Integer> items) {
     if(items.size() > 1) {
          List<Integer> smaller = new ArrayList<>();
          List<Integer> same    = new ArrayList<>();
          List<Integer> larger  = new ArrayList<>();

          Integer chosenItem = items.get(items.size()/2);
          for(Integer i : items)
          {
          if(i < chosenItem)
               smaller.add(i);
          else if(i > chosenItem)
               larger.add( i );
          else
               same.add( i );
          }
          sort( smaller ); // Recursive call!
          sort( larger ); // Recursive call!

          items.clear();
          items.addAll(smaller);
          items.addAll(same);
          itemas.addAll(larger);
     }
}
```

Non-Comparison Sorts

Idea:

- If we know something about the data, we don't strictly need to compare objects to each other
- If there are only a few possible values, and we know what they are, we can just sort by identifying the value
- If the data are strings and ints of finite length, then we can take advantage of their sorted order

- Two sorting techniques we use to this end:
  - Bucket sort
  - Radix sort
- If the data is sufficiently structured, we can get O(n) runtimes

Bucket Sort

- Idea: if all values to be sorted are known to be integers between 1 and *K* (or any small range):
  - Create an array of size *K*
  - Put each element in its proper bucket (a.k.a. bin)
  - *If* data is only integers, no need to store more than a *count* of how times that bucket has been used
- Output result via linear pass through array of buckets

Bucket Sort Example

- Example:
  - K = 7
  - Input = (7, 2, 2, 5, 5, 5, 1, 1, 1, 2, 2, 3, 3, 4, 6, 5, 2, 3, 5)
  - Output = (1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 5, 5, 6, 7)

Number	Count
1	3
2	5
3	3
4	1
5	5
6	1
7	1

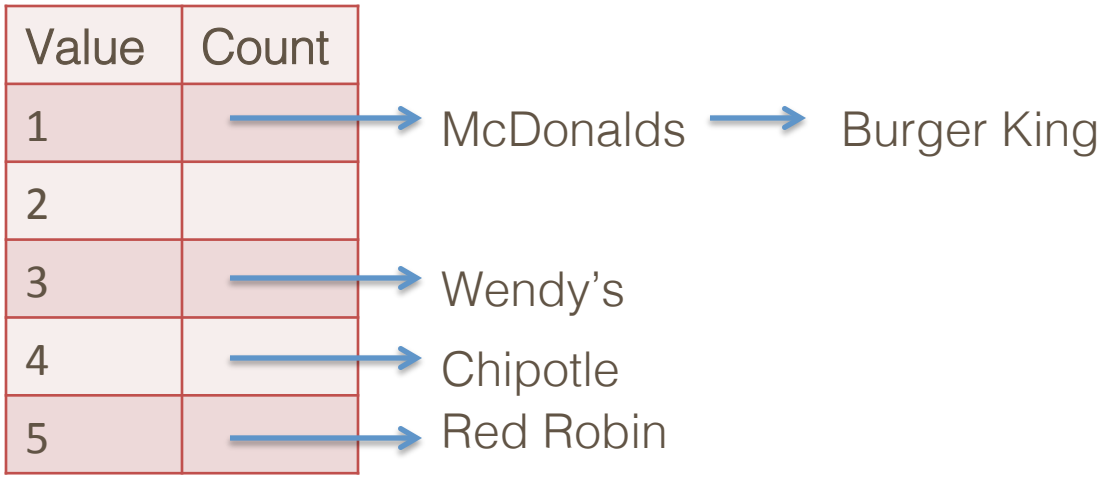
Bucket Sort Analysis

- Overall: $O(n+K)$
  - Linear in n, but also linear in K
- Good when K is smaller (or not much larger) than n
  - We don't spend time doing comparisons of duplicates
- Bad when K is much larger than n
  - Wasted space; wasted time during linear $O(K)$ pass

For data in addition to integer keys, use list at each bucket

Bucket Sort With Data

- Most real lists aren't just keys; we have data
- Each bucket is a list (say, linked list)
- To add to a bucket, insert in $O(1)$ (at beginning, or keep pointer to last element)
- Example: Fast foo restaurants, scale 1-5
- Input:
  - 1, McDonalds
  - 1, Burger King
  - 3, Wendy's
  - 4, Subway
  - 4, Chipotle
  - 5, Red Robin



- Easy to keep stable – McDonalds is still before Burger King

Radix Sort

- Radix = "the base of a number system"
  - Examples will use base 10 because we are used to that
  - In implementations use larger numbers
    - For example, for ASCII strings, might use 128 Idea:

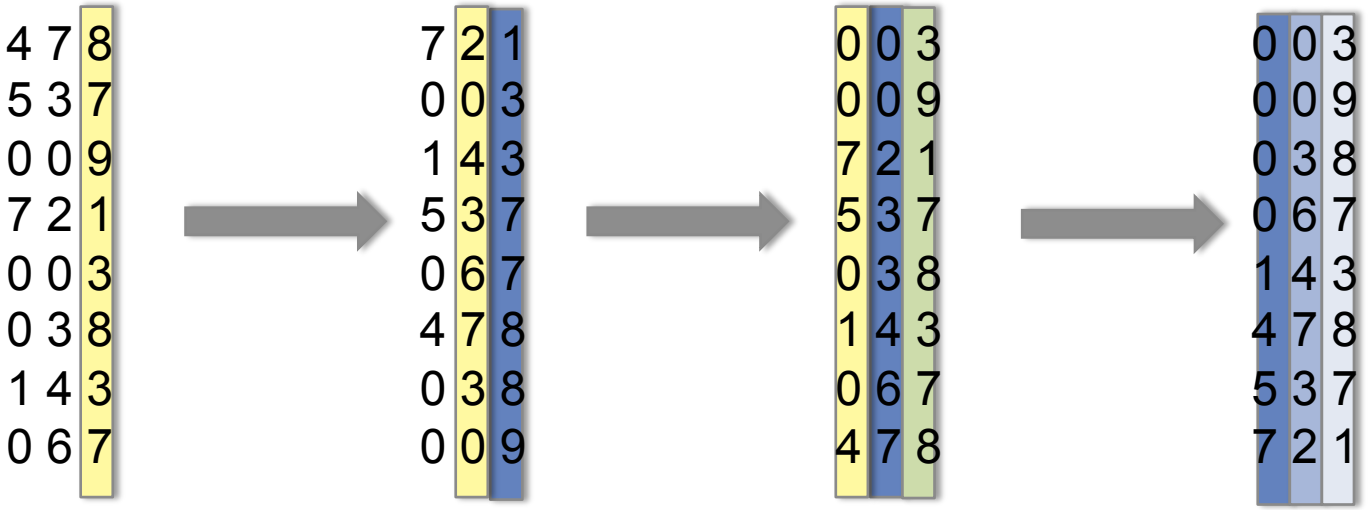
- Main idea: bucket sort on one digit at a time
  - Number of buckets = radix
  - Starting with *least* significant digit
  - Keeping sort *stable*
- Do one pass per digit
- Invariant: After *k* passes (digits), the last *k* digits are sorted

Radix Sort Example

Radix = 10

Input: 478, 537, 9, 721, 3, 38, 143, 67

3 passes (input is 3 digits at max), on each pass, stable sort the input highlighted in yellow



[Example credit: Evan McCarthy, University of Washington]

Radix Sort Analysis

- Input size: *n*
- Number of buckets = Radix: *B*
- Number of passes = "Digits": *P*

- Work per pass is 1 bucket sort: $O(B+n)$

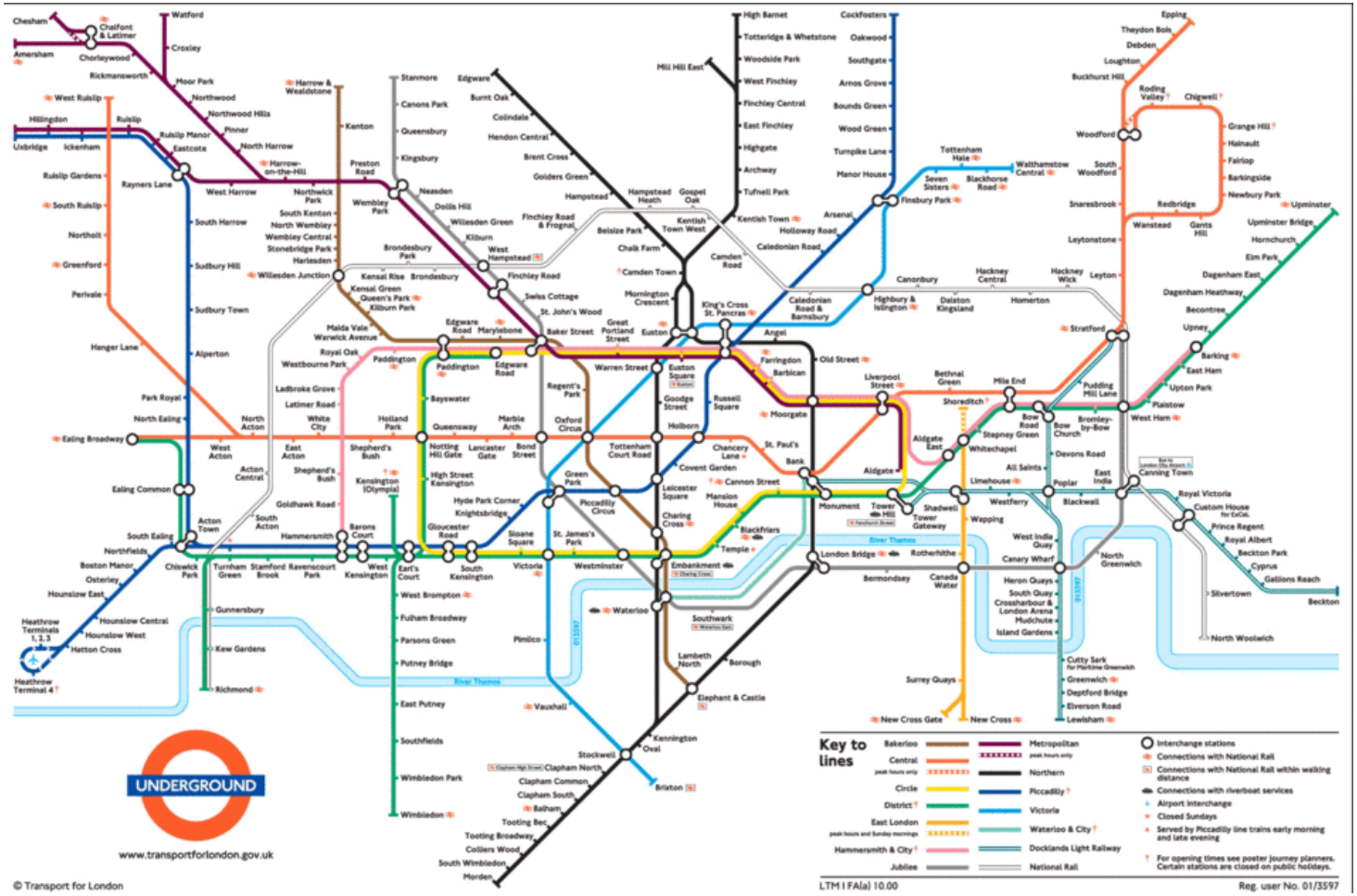
- Total work is $O(P(B+n))$

- Compared to comparison sorts, sometimes a win, but often not
- Example: Strings of English letters up to length 15
  - Run-time proportional to: $15*(52 + n)$
  - This is less than $n \log n$ only if $n>33{,}000$

Algorithms and Data Structures 3

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

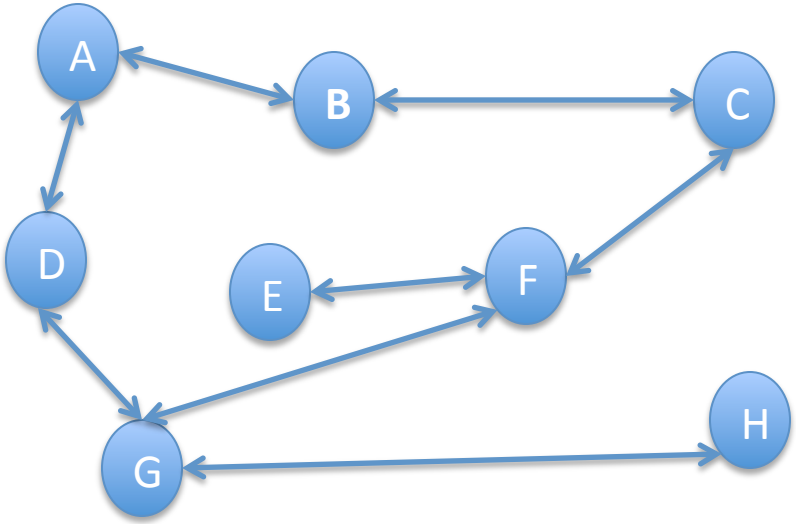
Walks, Trails and Circuits in a Graph

- 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

Example: Find a Path, a Walk, a Trail and a Circuit in a Graph

- Path - a set of edges connecting two vertices in a graph, where neither edges nor vertices repeat
- 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

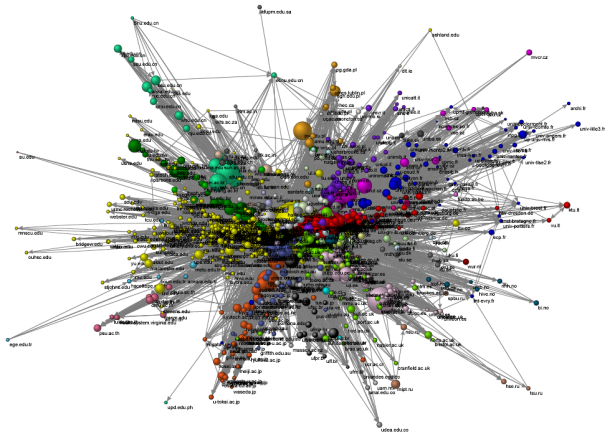


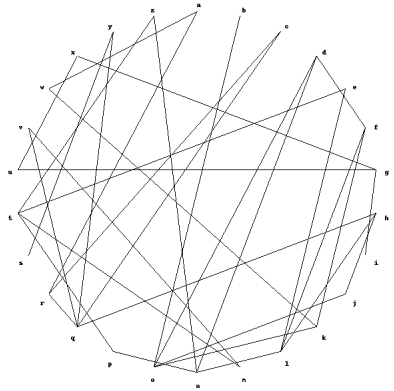
Connected and Complete Graphs

- Connected graph - an undirected graph that has a path from every vertex to every other vertex
- Strongly connected graph - a directed 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 each 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

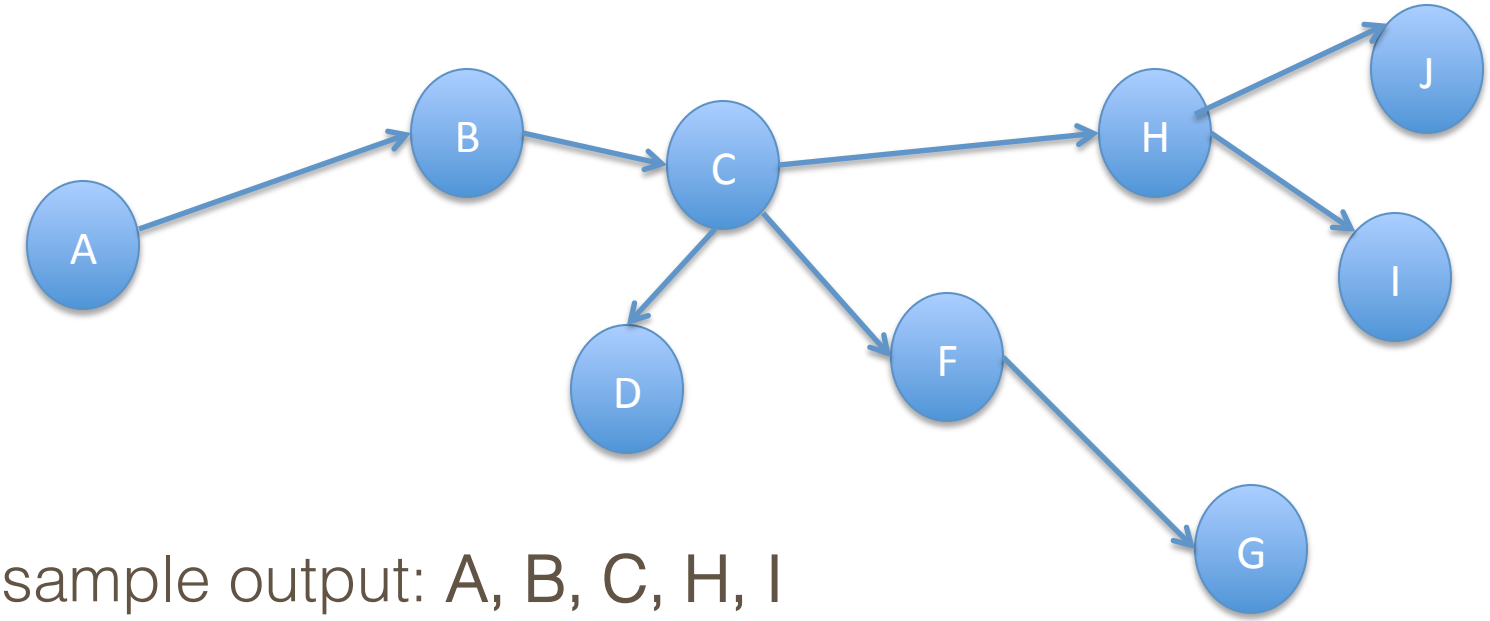
Graph Representations

- 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)

Topological Sort

- Problem: given a DAG G=(V,E), output all vertices in such an order that no vertex appears before another vertex that has an edge to it



- One sample output: A, B, C, H, I

Topological Sort

- Problem: given a DAG G=(V,E), output all vertices in such an order that no vertex appears before another vertex that has an edge to it

- Topological sort useful when dependency of execution matters

- A simple topological sort algorithm:
  1. Find any vertex with no incoming edges
  2. Print that vertex, and remove it from the graph, along with its edges
  3. Apply this same strategy to the rest of the graph

Topological Sort

- A simple topological sort algorithm:
  1. Find any vertex with no incoming edges
  2. Print that vertex, and remove it from the graph, along with its edges
  3. Apply this same strategy to the rest of the graph

- Indegree of a vertex *v* as the number of edges (*u*, *v*)

Topological Sort

```
void topsort() throws CycleFoundException {
    for(int counter = 0; counter < NUM_VERTICES; counter++) {
        Vertex v = findNewVertexOfIndegreeZero();
        if(v == null)
            throw new CycleFoundException();
        v.topNum = counter;
        for each Vertex w adjacent to v
            w.indegree--;
    }
}
```

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:

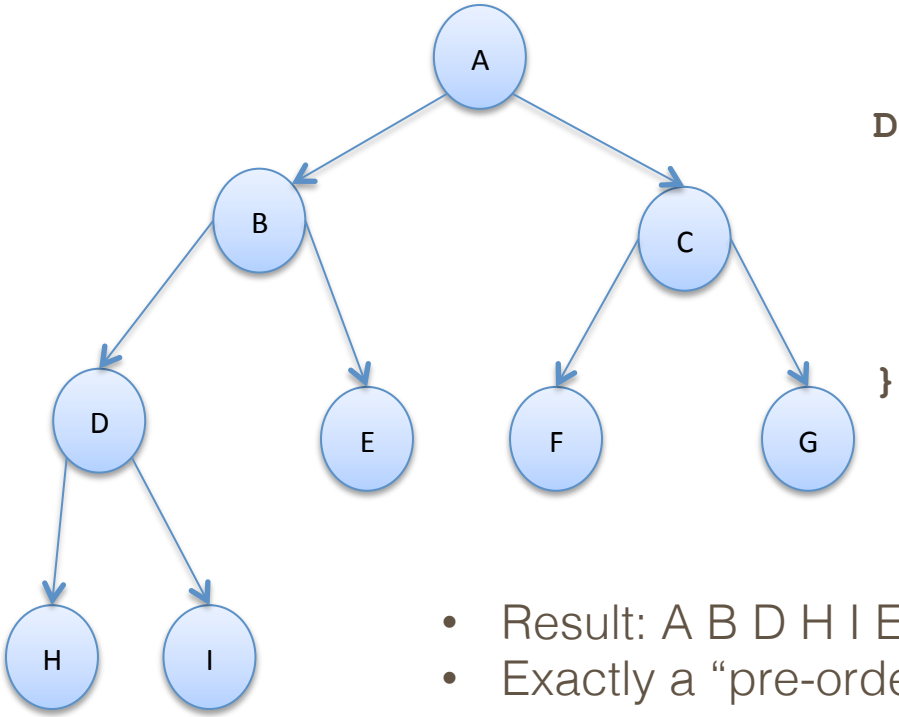
```
void traverseGraph(Node start) {
    Set pending = emptySet()
    pending.add(start)
    mark start as visited
    while(pending is not empty) {
        next = pending.remove()
        for each node u adjacent to next
        if (u is not marked visited) {
            mark u
            pending.add(u)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

Graph Traversals –Options

- Observations:
  - Assuming we can add and remove from our `pending` data structure in O(1) time, the entire traversal is O(|E|)
  - Our traversal order depends on what we use for our `pending` data structure
    - Stack:DFS
    - Queue:BFS

Example: DFS on Trees

- **Observation** - a tree is a graph, and makes DFS and BFS are easier to "see"

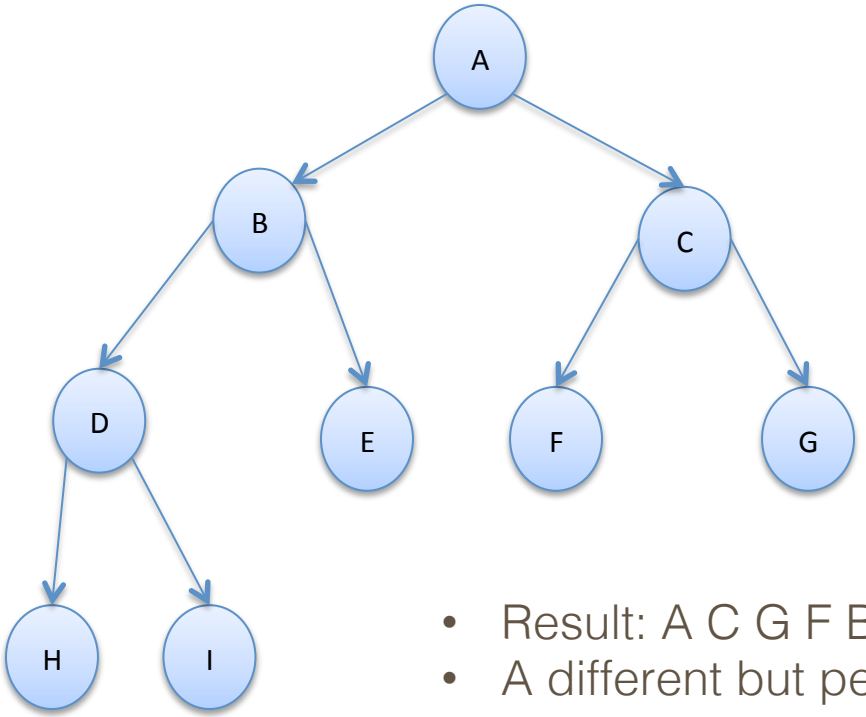



```
DFS(Node start) {
    mark and process start
    for each node u adjacent to start
        if u is not marked
            DFS(u)
}
```

- Result: A B D H I E C F G
- Exactly a "pre-order traversal" for trees
- The marking is because we support arbitrary graphs and we want to process each node exactly once

Example: DFS on Trees

- **Observation** - a tree is a graph, and makes DFS and BFS are easier to "see"




```
DFS2(Node start) {
    initialize stack s to hold start
    mark start as visited
    while(s is not empty) {
    next = s.pop() // and "process"
    for each node u adjacent to next
        if(u is not marked)
            mark u and push onto s}
}
```

- Result: A C G F B E D I H
- A different but perfectly fine depth traversal

DFS/BFS Comparison

- Breadth-first (BFS) always finds shortest length paths ("optimal solutions")
  - Better for "what is the shortest path from *x* to *y*"
- Depth-first (DFS) can use less space to find a path
  - If *longest path* in the graph is ***p*** and highest out-degree is ***d***, then DFS stack never has more than ***d**p*** elements
  - But a queue for BFS may hold *O*(|V|) node
- *Iterative deepening (IDFS)* - a third approach
  - Useful in Artificial Intelligence
  - Idea:
    - Try DFS but disallow recursion more than *K* levels deep
    - If that fails, increment *K* and start the entire search over
  - Like BFS, finds shortest paths
  - Like DFS, less space

Shortest Path Problems

- Shortest path problem for un-weighted graphs
  - Done – use BFS to find the minimum path length from node *v* to *u* in *O*(|E|+|V|)
  - More - we can find the minimum path length from *v* to *every node* (still *O*(|E|+|V|))

- Shortest path problem for weighted graphs
  - Given a weighted graph and node *v*, find the minimum-cost path from *v* to every node
  - BFS does not work → it only looks at path length, and the minimum cost path may not have the fewest edges (don't you hate when this happen with your airfare?)

Shortest Path Problems For Weighted Graphs

- Assumption:
  - Weights represent cost, so there are no negative weights
  - Problem is *ill-defined* if there are negative-cost *cycles*
  - Today's algorithm is *wrong* if *edges* can be negative
    - There exist other slower (but not terrible) algorithms for dealing with negative edges

Dijkstra's Algorithm

- Idea: somewhat similar to BFS, but adapted to handle weights
- Initially, start node has cost 0 and all other nodes have cost ∞
- At each step:
  - Pick closest unknown vertex *v*
  - Add it to the "cloud" of known vertices
  - Update distances for nodes with edges from *v*

Dijkstra's Algorithm

- For each node *v*, set:
  - *v*.cost = ∞ and
  - v.known = false
- Set source.cost = 0
- While there are unknown nodes in the graph
  - Select the unknown node *v* with lowest cost
  - Mark *v* as known
  - For each edge (*v*,*u*) with weight *w*,

```
   c1 = v.cost + w//cost of best path through v to u
   c2 = u.cost //cost of best path to u previously
known
`  if(c1 < c2){ //if the path through v is better
   u.cost = c1
   u.path = v // for computing actual paths
```

Dijkstra's Algorithm - Features

- When a vertex is marked known, the cost of the shortest path to that node is known
  - The path is also known by following back-pointers
- While a vertex is still not known, another shorter path to it might still be found
- Note: The "Order Added to Known Set" is not important
  - A detail about how the algorithm works (client doesn't care)
  - Not used by the algorithm (implementation doesn't care)
  - It is sorted by path-cost, resolving ties in some way
  - Helps give intuition of why the algorithm works

Algorithms and Data Structures 3

QUICK REVIEW OF THE ALGORITHM DESIGN TECHNIQUES

Greedy Algorithm

[Picture credit: http://www.arborinvestmentplanner.com/wp-content/uploads/2013/05/Greedy.jpg]

Greedy Algorithm

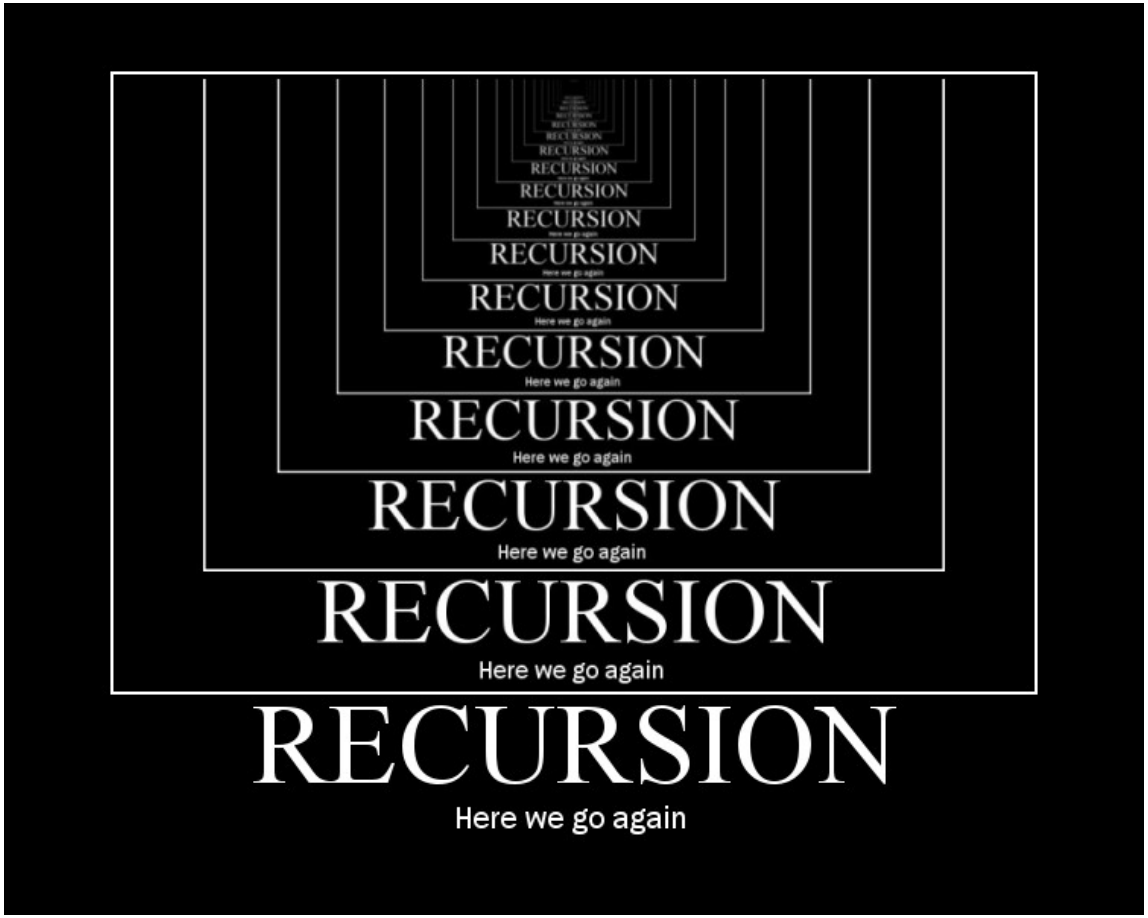
- Greedy algorithm - an algorithmic approach where we make a locally optimal choice at every stage of the problem, with the hope of finding a global optimum

- In general, greedy algorithms have five components:
  - A candidate set, from which a solution is created
  - A selection function, which chooses the best candidate
  - A feasibility function, that is used to determine if the candidate can be used to contribute to a solution
  - An objective function, which assigns a value to a solution
  - A solution function, which indicates when the complete solution has been discovered

Greedy Algorithm

- Some applications:
  - Some scheduling problems
  - Minimum spanning tree
  - Huffman coding
  - Network routing problems
  - Some decision tree problems
  - Many submodularity heuristics

Recursive Algorithm



[Picture credit: https://prateekvjoshi.files.wordpress.com/2013/10/part-1.jpg]

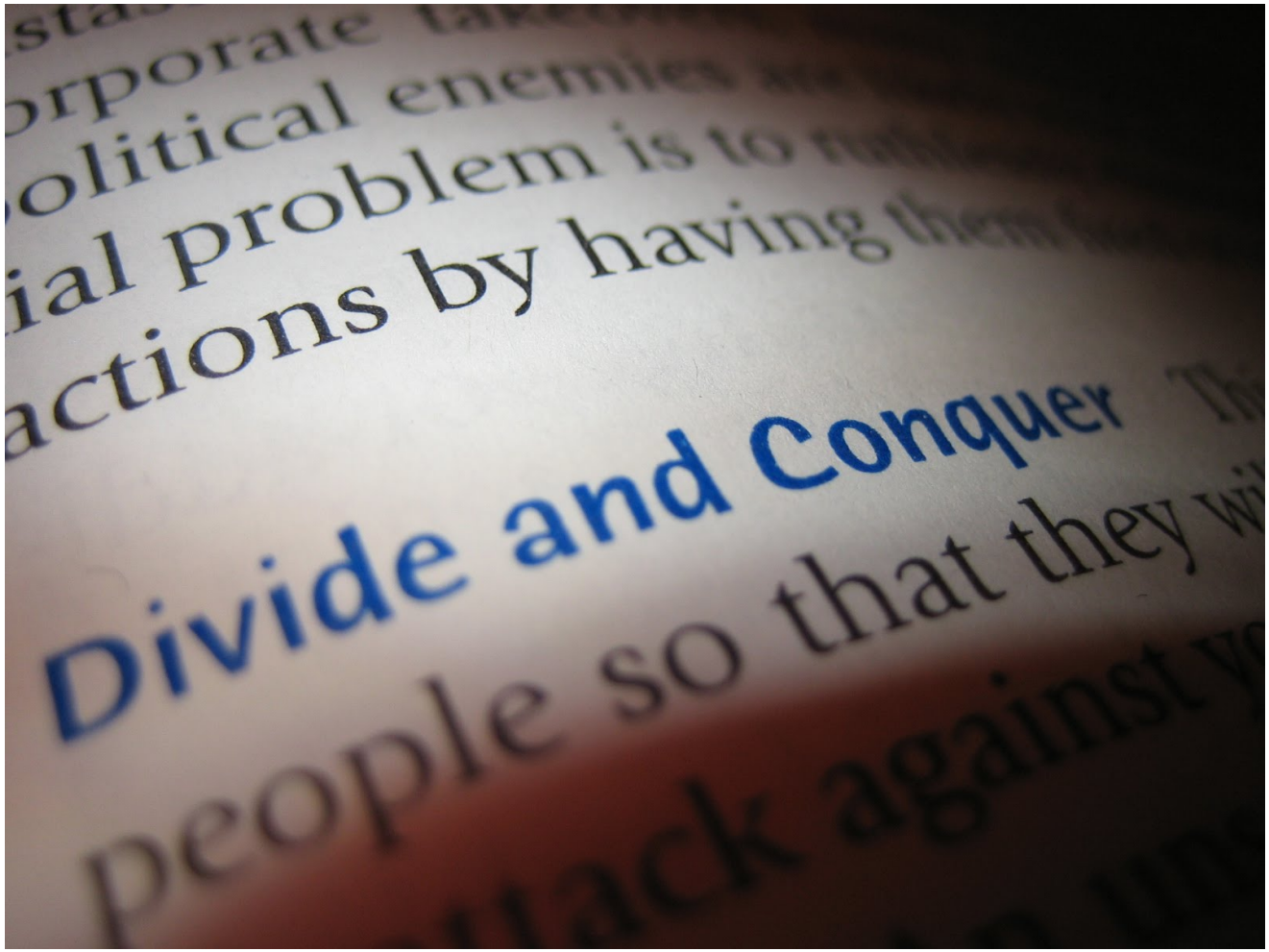
Recursive Algorithm

- 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

- 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

Divide and Conquer Algorithm



[Picture credit: https://i.ytimg.com/vi/yjCHpKBx3ak/maxresdefault.jpg]

Divide and Conquer Algorithm

- Divide-and-conquer - useful technique for solving many kinds of problems:
  - Tree traversals (DFS)
  - Sorting (merge sort, quick sort)

- Divide-and-conquer steps:
  1. Divide work up into smaller pieces (recursively)
  2. Conquer the individual pieces (as base cases)
  3. Combine the results together (recursively)

- Pseudocode:

```
divideAndConquer(input) {
    if (small enough) {
        CONQUER,solve,and return input
    }else{
        DIVIDE input into multiple pieces
        RECURSE on each  piece
        COMBINE and return results
        }
    }
```

Backtracking Algorithm

[Picture credit: https://upload.wikimedia.org/wikipedia/commons/thumb/8/8c/Sudoku_solved_by_bactracking.gif/220px-Sudoku_solved_by_bactracking.gif]

Backtracking Algorithm

- 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?

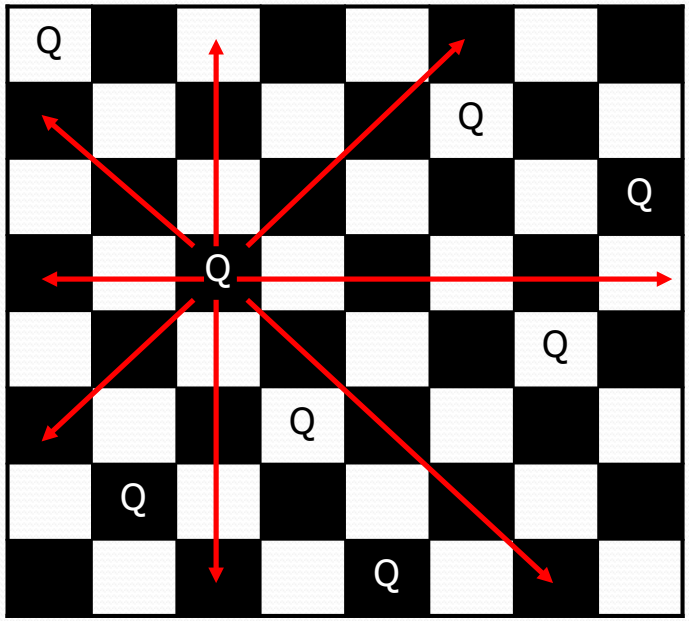
Backtracking Algorithm

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!)
```

Example Backtracking Algorithm: Eight Queens

- Consider the problem of trying to place 8 queens on a chess board such that no queen can attack another queen.
  - What are the "choices"?
  - How do we "make" or "un-make" a choice?
  - How do we know when to stop?



[Picture credit: Zorah Fung, University of Washington]

Example Backtracking Algorithm: Eight Queens

- Suppose we have a `Board` class with these methods:

Method/Constructor	Description
`public` **Board**`(int size)`	construct empty board
`public boolean` **isSafe**`(int row, int column)`	true if queen can be safely placed here
`public void` **place**`(int row, int column)`	place queen here
`public void` **remove**`(int row, int column)`	remove queen from here
`public String` **toString**`()`	text display of board

- Write a method `solveQueens` that accepts a `Board` as a parameter and tries to place 8 queens on it safely
- Your method should stop exploring if it finds a solution

Algorithms and Data Structures 3

SOME MORE TREES

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

AVL (Adelson-Velskii-Landis) Tree

- AVL tree – binary search tree with a balance condition (AVL condition):
  - The height of the left and right subtrees differ by at most 1

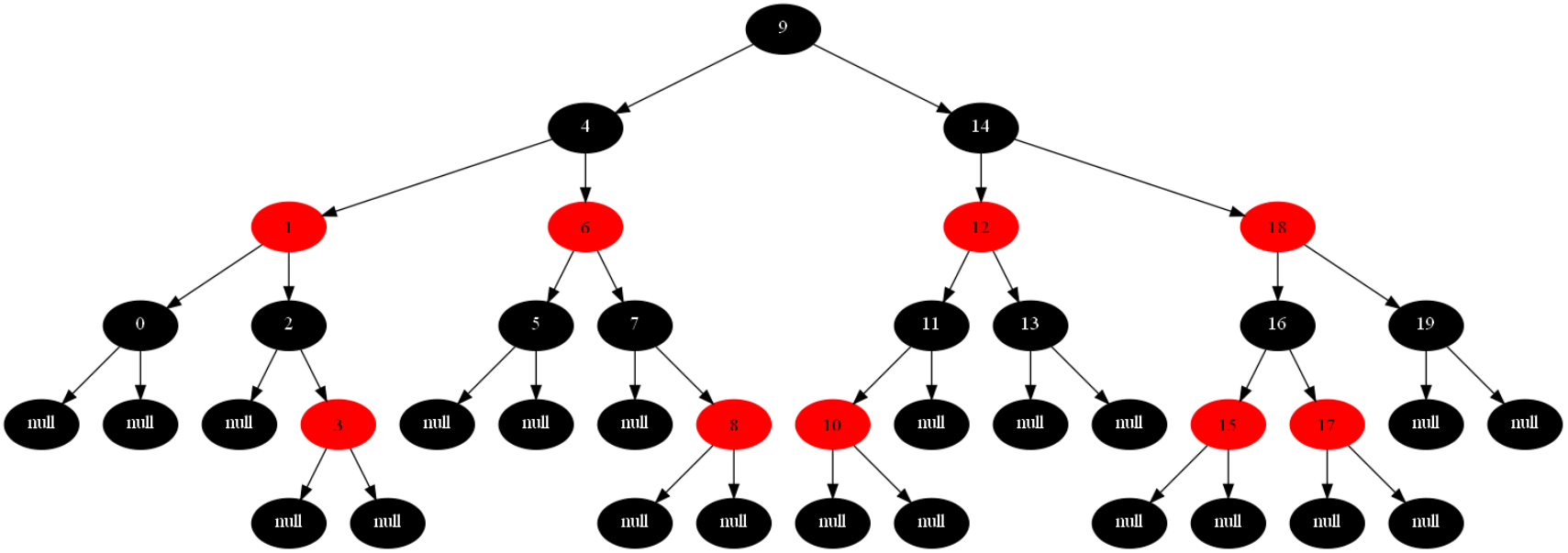
- The height of an empty tree defined to be -1
- All tree operations (exception insertion) can be performed in O(logN)

AVL Trees – Concluding Remarks

- If AVL rotation can enforce O(log n) height, what are the asymptotic runtimes for our functions?
  - insert(T x, AvlNode node) = O(log n) + balancing
  - find(T x) : O(height) = O(log n)

- How expensive is balancing?
- There are at most three nodes and four subtrees to move around → O(1)

Red-Black Trees



[Picture credit: http://www.programering.com/images/remote/ZnJvbT1jc2RuJnVybD1uQm5hdUFUTzNrVE0xRURNMWtUTzFRek40WUROM1V6TDkwVFFhZGplMGcxUXRWMGN2eEdUbEozWHhsVEx6RUdTdjhHZHZoR2N2UVhadTVpTnlFakxuMVdhaTV5WnRsMkx2b0RjMFJIYQ.jpg]

Red-Black Trees

- Red-black tree - a binary search tree which satisfies the *red-black property*:
  - Every node has an additional property, color, which can be either red or black.
  - The root of the tree is always black.
  - All leaves (NIL) are always black.
  - If a node is red, then both its children are black.
  - Every simple path from a node to a descendant leaf contains the same number of black nodes.

Comparison between AVL and Red-Black Trees

- The height of both AVL and Red-black trees is always O(log N), where N is the number of nodes
- AVL trees tend to be more balanced than Red-black trees, but they may cause more rotations during insertion and deletion
- → if an application involves many frequent insertions and deletions, Red-Black tree may be preferred

Inserting a Node into a Red-Black Tree

- To balance a Red-Black tree after insertion, we use two tools:
  - Recoloring
  - Rotation

- Pseudocode: let x be a newly inserted node
  1. Perform a standard BST insertion, and make the color of newly inserted node Red.
  2. If x is root, change its color to Black.
  3. If x is not root, and the color of x's parent is Red:
     - If x's uncle is Red (x's grand parent must have been Black):
       - Change the color of parent and uncle to Black.
       - Change the color of grandparent to Red.
       - Set x = x's grandparent, and repeat steps 2 and 3.

Inserting a Node into a Red-Black Tree

- Pseudocode: let x be a newly inserted node
  1. Perform a standard BST insertion, and make the color of newly inserted node Red.
  2. If x is root, change its color to Black.
  3. If x is not root, and the color of x's parent is Red:
     - If x's uncle is Black, then there are four possible configurations for x, x's parent p, and x's grandparent g:
       - Left left case (p is left child of g, and x is left child of p)
       - Left right case (p is left child of g, and x is right child of p)
       - Right right case
       - Right left case

- The cases are similar to the AVL tree

Algorithms and Data Structures 3

HANDLING EVENTS IN JAVA

Event-Driven Programming

- Many Java applications are *event-driven* (GUI) programs
  - Program initializes itself on startup then enters an *event loop*
- Abstractly:

```
do {
    e = getNextEvent();
    process event e;
} while (e != quit);
```

- Contrast with application- or algorithm-driven control, where a program expects input data in a particular order

Some Examples of GUI Events

- Typical *events* handled by a GUI program:
  - Keyboard: key press or release, sometimes with modifiers like shift/control/alt/etc.
  - Mouse: move/drag/click, button press, button release – also can have modifiers like shift/control/alt/etc.
  - Finger: tap or drag on a touchscreen
  - Other device inputs (e.g., joystick, drawing tablet)
  - Window: resize/minimize/restore/close
  - Network activity or file I/O: start, done, error
  - Timer interrupt (including animations)

Events in Java AWT/Swing

- Most of the GUI widgets can generate events
- Events handled using the observer design pattern:
  - Objects wishing to handle events register as observers with the objects that generate them
  - When an event happens, appropriate method in each observer is called
  - As expected, multiple observers can watch for, and be notified of an event generated by an object

Event Objects

- A Java GUI event is represented by an *event object*
  - Superclass is `AWTEvent`
  - Some subclasses:
    - `ActionEvent` – GUI-button press
    - `KeyEvent` – keyboard

      `MouseEvent` – mouse move/drag/click/button
- Event objects contain information about the event
  - UI object that triggered the event
  - Other information depending on event
    - Examples:
      - `ActionEvent` – text string from a button
      - `MouseEvent` – mouse coordinates

Event Listeners

- A *Event listeners* must implement the proper interface:
  - `KeyListener, ActionListener, MouseListener` (buttons)
  - `MouseMotionListener` (move/drag), ...
- Or extend the appropriate *abstract class* that provides empty implementations of the *interface* methods
- When an event occurs, the appropriate method specified in the interface is called:
  - `actionPerformed,`
  - `keyPressed,`
  - `mouseClicked,`
  - `mouseDragged, ...`
- An event object is passed as a parameter to the event listener method

References and Reading Material

- Mark Allen Weiss, Data Structures and Algorithm Analysis in Java, chapters 5, 6, 7, 9, 10, 12
  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/
- Geeks for Geeks, Sorting algorithms, [Online],
  http://www.geeksforgeeks.org/sorting-algorithms/
- Graphs in Computer Science, [Online]
  http://web.cecs.pdx.edu/~sheard/course/Cs163/Doc/Graphs.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
- Geeks for geeks, Red-Black Trees [Online],
  http://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/