Sorting

- Definition
- Built in Java
- Sorts
  * Bubble
  * Combsort
  * Selection
  * Insertion
  * Quicksort
  * Radix/Bucket Sort

Sorting: definition

- Sorting: process by which a collection of items is placed into order
  * Operation of arranging data
  * Order typically based on a data field called a key
- May be done to facilitate some other operation
  * Searching for elements
- Must handle all configurations
  * 8 elements: 8! = 40,320 possible arrangement

Sorting: description

- Algorithms consist of:
  - * Comparisons
  - * Swaps
  - * Assignments
- Selection of algorithm based on:
  - * Number of elements
    - Internal: All elements in memory
    - External: Not all fit
  - * Amount of data to be moved

Sorting: algorithm selection

- Evaluation criteria
  - * Best vs. worst vs. average case
- Is stable sort needed?
  - * Stable: elements with equals keys stay in same order

JDK approaches

- **Two approaches**
  - **Keep collection sorted all the time**
    - ***java.util.TreeSet*** **- we'll discuss with trees**
  - **Sort collection upon demand**
- **java.util.Collections class provides static utility methods, including sorting methods**
  - **simplest is *public static void sort(*** *List* ***list)***
  - **How does implementation know how to compare objects in less for greater than / less than?**
    - ***Object*** **class provides *equals*, but no method for determing greater than / less than.**

Comparable interface

- **in java.lang**
- **Single method: *public int compareTo(*** *Object* ***o)***
  - **returns int < 0 if current object < "o"**
  - **returns 0 if current object = "o"**
  - **returns int > 0 if current object > "o"**
- **Consider an employee class with a name and id number:**

```
public class Employee {
   private String name;
   private int id;
   public Employee(String name, int id) {
      this.name = name;
      this.id = id;
   }
//setters and getters omitted - setName, setId, getName,getId
```

```
public String toString( ) {
    StringBuffer sb = new StringBuffer(name);
    sb.append(" - ");
    sb.append(id);
    return sb.toString( );
}
```

- **To allow sorting, create a subclass which implements *Comparable***

```
public class ComparableEmployee extends Employee implements Comparable {
    public ComparableEmployee(String name,int id) {
        super(name,id);
    }
    //sorts by name
    public int compareTo(Object o) {
        Employee other = (Employee)o;
        return getName( ).compareTo(other.getName( ));
    }
```


Sorting example

```
public static void main(String args[]) {
        Vector emps = new Vector( );
        emps.add(new ComparableEmployee("Tim",1));
        ...
        emps.add(new ComparableEmployee("Aaron",7));
        Collections.sort(emps); //sorting here
        Iterator iter = emps.iterator( );
        while (iter.hasNext( )) {
            System.out.println(iter.next( ));
        }
```

- **What if we want to sort by id instead of name?**
    - **Can we use same ComparableEmployee class?**

1

Comparator interface

- **Specifies specific method of comparing objects**

  In java.util package

- **Implementors must implement *public int compare(*Object *o1,* Object *o2)***

  *** return <0, 0, or >0 if o1 is less than, equal to, or greater than o2**

  *** Separates specific comparision algorithm from object**

- **To sort by name**

```
public class EmployeeNameSort implements Comparator {
   public int compare(Object o1, Object o2) {
      Employee e1 = (Employee)o1;
      Employee e2 = (Employee)o2;
      return e1.getName( ).compareTo(e2.getName( ));  }  }
```


- **To sort by id**

```
public static class EmployeeIdSort implements Comparator {
   public int compare(Object o1, Object o2) {
      Employee e1 = (Employee)o1;
      Employee e2 = (Employee)o2;
      final int id1 = e1.getId( );
      final int id2 = e2.getId( );
      if (id1<id2)
         { return -1; }
      if (id1 > id2)
         { return 1; }
      return 0;  }   }
```

- **Sample usage**

```
....
Collections.sort(emps,new EmployeeNameSort( ));
...
Collections.sort(emps,new EmployeeIdSort( ));
```

  *** Note *sort* is overloaded on collections**

Bubble Sort

- Scan through list swapping pairs, repeat until done
  - * Start on left. Compare first pair. If out of order, swap them.
  - * Compare two and three next.
  - * Go until end of list
  - * Repeat (How far do we have to go?)
  - * 17 5 11 31 1 29
  - * What's the performance?

Comb-sort

- Like bubble except swap elements further apart
  - * Divide size of list by 1.3

    6 / 1.3 = 4.5 -> 4

    17 5 11 31 1 29
  - * On subsequent pass, reduce swap distance by 1.3

    4 / 1.3 = 3.07 -> 3

    3 / 1.3 = 2.3 -> 2

    2 /1.3 = 1.5 -> 1
  - * Ends as a bubble sort

Comb-sort evaluation

- What's the performance?
- Why is the factor 1.3?
  * I don't know either.
  * 1.3 seems to work the best.
  * Empirically, performance is O(n log n)
- Used in Eiffel standard library

Selection Sort

- Find smallest, then next smallest, etc.
  * Scan array for smallest element
  * Swap with first element
  * Scan array (2…n) for smallest element
  * Swap with second element
  * Repeat until you get to the end
- 17 5 11 31 1 29

Insertion sort

- Simple sort, conceptually
- Take second element
  * arrange in proper relation to first
- Take third element
  * arrange in proper position in relation to first to
- And so on…
- 17 5 11 31 1 29

Insertion sort performance

- Two loops
  * Once through the array
  * Once to place element in sorted subset
- Best case, already sorted
  * once through, length -1 comparisons, no swaps
  * O(N)
- Average case
  * length -1 comparisons by sorted_length/2 comparisons, have to swap 1/2 the time
  * $O(N^2)$

Insertion sort performance

- **Worst case: Reverse order, move everything**
- **Advantages**
  - *** Best performance is best possible**
  - *** Works well with partially sorted lists**
  - *** Stable sort**
  - *** Simple to understand and code (maybe)**
- **Disadvantages**
  - *** Poor performance in random order case**
  - *** $O(N^2)$ inappropriate for large arrays**

QuickSort

- **Most common sort used in libraries.**
  - *** C: *qsort***
  - *** C++: *std::sort***
- **Uses divide and conquer approach**
  - *** One pass divides set into two pieces**
    - **Everything in one pile smaller than the other**
  - *** It can be shown performance is O(N log N)**
  - *** Developed by C.A.R. Hoare**

Quicksort Algorithm

- Splits array into two parts
  - * *Not* equal halves
  - * Randomness makes it work
    - On average, the time partitions will be reasonable sizes
- Recursive algorithm
- Pick a "pivot" value to split the values
  - * Actual value doesn't matter
  - * Use the first element of the array
  - * Use two pointers, one at high end, one at low end

Quicksort algorithm

- Partitioning
  - * Starting at high end and going down, find first element that is less than the pivot
  - * Starting at low end and going up, find first element that is greater than or equal to the pivot
  - * Swap elements
  - * Repeat
  - * When pointers meet, the two partitions are separated
  - * Repeat on each half

Quicksort example

- 17 5 11 31 1 29 2 13 23

Quicksort analysis

- Worst case is reverse order sort
  - * Why?
  - * Worst case $O(N^2)$
  - * Recursion depth N, lots of stack
- Versions often coded to handle special cases gracefully
  - * Randomized quicksort
  - * Best average performance O(NlogN)
  - * Most widely used method for internal sorting of large files

Quicksort disadvantages

- **Not stable**
- **Worse case bad, takes lots of space**

Mergesort

- **Given two sorted lists, it's easy to create new, sorted list**
  - **5 11 17 31**
  - **1 2 13 23 29**

- **If mergesort recursively done on small portions of original data, it can then be applied to larger pieces.**
- **17 5 11 31 1 29 2 13 23**

- **Implementation used in *java.util.Collections.sort* methods.**

Heapsort

- **Heapsort is an in place sorting method that uses no recursion.**
- **<u>Binary tree:</u> Tree where each node has a maximum of two children.**
- **Heap is special kind of binary tree.**
- **<u>Almost full binary tree</u>: All leaves are on two levels with all bottom leaves as far left as possible.**
- **<u>Heap:</u> Almost full binary tree in which the value in each node is greater than or equal to the value of both of its children.**
- **Heapsort is a two-stage process:**

1. Make a heap of the elements to be sorted.

1.1 Add element to the lowest, left open position in the tree.

1.2 Swap element up until added element is greater than values of either of its children.

1.3 Repeat 1.1 and 1.2 until all elements in heap.

2. Convert the heap into a sorted list.

2.1 Swap the root (largest element) with element in last unsorted position.

2.2 Swap root element down into position in heap.

2.3 Repeat 2.1 and 2.2 until all elements are sorted.

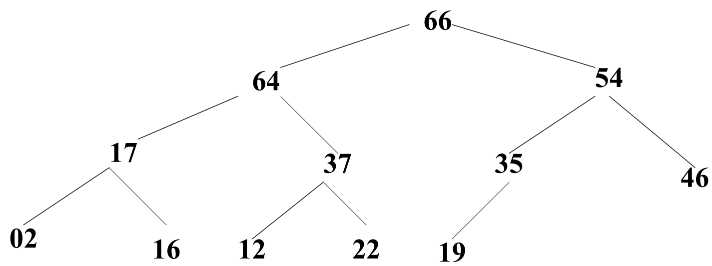
- **Elements stored in an array:**

  *** Root stored in first position.**

  *** Children of an element are stored in (2 * position) and (2 * position + 1) locations.**

Original list: 19 02 46 16 12 54 64 22 17 66 37 35

Completed heap:



Array: 66 64 54 17 37 35 46 02 16 12 22 19

- **Sorting the heap:**

1. Exchange root with last element in heap.

*** Since root is largest element, it will be in proper place.**

*** Heap array is now one element smaller.**

2. Swap root element down to proper heap position.

3. Repeat steps 1 and 2 until all elements sorted.

- **What does this look like?**

Heapsort analysis

- Advantages to Heapsort:
  * Heapsort has a better worst case performance than quick-sort.
  * Guaranteed to run in O(NlogN) time.
  * Natural implementation for a priority queue.
- Disadvantages:
  * Average case performance considerably worse than quick-sort.

Radix Sort

- Sorts elements into categories based on a given commonality
  * then sorts each category
  * e.g. sorting playing cards

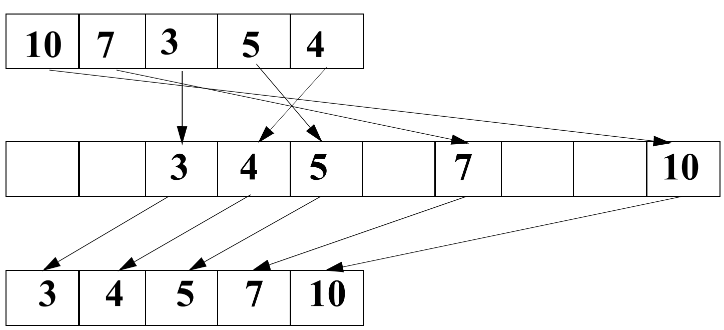
    sort first by suit (Hearts, Spades, Clubs, Diamonds)
- Sorting by names
  * Filing cabinet
  * Library

Bucket sort

- **Specialized radix sort**
  - *** Let A be an array of integers to be sorted**
  - *** Create B, a second array whose size is maximum value in A**
  - *** Put elements from A into B using value as an index into B**
  - *** Backcopy in order into A**

Heidi C. Ellis and Gerard C. Weatherby

Bucket Sort



- **Advantage: VERY fast**
- **Disadvantages**
  - *** Elements must be indices**
  - *** No duplicates**
  - *** Lots of space**