CSE 373

Sorting 4: Heap Sort, Bucket Sort, Radix Sort, Stooge Sort

reading: Weiss Ch. 7

slides created by Marty Stepp
http://www.cs.washington.edu/373/

Heap sort

- **heap sort**: Arranges elements by adding all of them to a priority queue, then calling remove-min to grab the elements in order.
  - For an input array *a*, create a priority queue *pq*.
  - For each element in *a*, add *a* to *pq*.
  - For each index *i* in *a*, until *pq* is empty, remove-min from *pq* and put it into the next slot *a*[*i*].

  - Takes advantage of the heap's ordering property.
  - Requires O(*N* log *N*) average runtime.
    - Faster with ascending or descending input (less bubbling needed).
    - Faster than shell sort but somewhat slower than merge sort, and harder to parallelize across multiple computers / processors, so generally considered worse than merge sort for general usage.

Heap sort code

```
// Sorts the contents of a using heap sort algorithm.
public static void heapSort(int[] a) {
    Queue<Integer> pq = new PriorityQueue<Integer>();
    for (int k : a) {
        pq.add(k);
    }

    // put the elements back into a, sorted
    int i = 0;
    while (!pq.isEmpty()) {
        a[i] = pq.remove();
        i++;
    }
}
```

- O(*N*) memory is used to store the auxiliary data structure.

Building a heap in-place

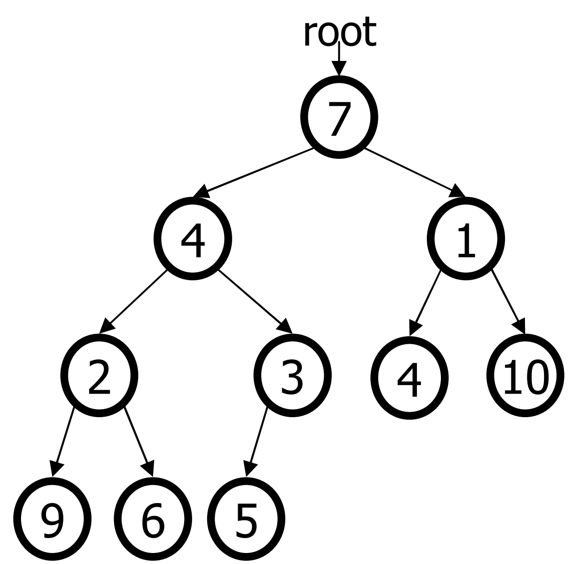
- Rather than copying *a* into a separate *pq*, we can just treat *a* as our *pq*'s internal heap array.
  - Requires us to arrange *a* into max-heap vertical order.
  - A clever way to build a heap from an unordered array is to run the equivalent of a "bubble down" on each non-leaf from right to left.

index	0	1	2	3	4	5	6	7	8	9
value	7	4	1	2	3	4	10	9	6	5

index	0	1	2	3	**4**	5	6	7	8	9
value	7	4	1	2	**5**	4	10	9	6	**3**

index	0	1	2	**3**	4	5	6	7	8	9
value	7	4	1	**9**	5	4	10	**2**	6	3

...



Heap sort code v2

```
// Sorts the contents of a using heap sort algorithm.
public static void heapSort(int[] a) {
    // turn a into a max-heap
    for (int i = a.length / 2; i >= 0; i--) {
        bubbleDown(a, i, a.length - 1);
    }
    for (int i = a.length - 1; i > 0; i--) {
        swap(a, 0, i);              // remove-max, move to end
        bubbleDown(a, 0, i - 1);
    }
}
```

- *variation:* Turn *a* itself into a max-heap, rather than copying it into an external priority queue.
  - On remove-max, move the max to the end of the array.
  - No extra memory used! More efficient.

Heap sort code v2, cont'd.

```
// Swaps a[hole] down with its larger child until in place.
private static void bubbleDown(int[] a, int hole, int max) {
    int temp = a[hole];
    while (hole * 2 <= max) {
        // pick larger child
        int child = hole * 2;
        if (child != max && a[child + 1] > a[child]) {
            child++;
        }
        if (a[child] > temp) {
            a[hole] = a[child];
        } else {
            break;
        }
        hole = child;
    }
    a[hole] = temp;
}
```

Bucket sort

- **bucket sort**: A sort on integers in a known range where elements are "sorted" by tallying counts of occurrences of each unique value.
  - For an input array *a* of integers from [0 .. *M*), create an array of counters *C* of size *M*.
  - For each element in *a*, if its value is *k*, increment *C*[*k*].
  - Use the counters in *C* to place *a*'s contents back in sorted order.

  - This sort can be even faster than quick sort!
    - It takes advantage of additional information about the data, namely, that it is a collection of integers in a specific range.
    - With some kinds of data, can be even more clever:
      - if all values in *a* are unique, *C* can be boolean[] instead of int[]
      - can use a *BitSet* to save memory (*C* will be smaller)

Bucket sort example

- input array *a*:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	7	4	1	2	5	4	10	9	1	4	7	8	9	8	9	4	4

- create array of tallies:

index	0	1	2	3	4	5	6	7	8	9	10
value	0	2	1	0	5	1	0	2	2	3	1

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	1	1	2	4	4	4	4	4	5	7	7	8	8	9	9	9	10

- use tallies to generate sorted contents of *a*

Bucket sort code

```
// Sorts the contents of a using bucket sort algorithm.
// Precondition: all elements in a are in range 0 .. 999999
public static void bucketSort(int[] a) {
    int[] counters = new int[1000000];
    for (int k : a) {
        counters[k]++;    // tally the counters
    }

    // put the counter information back into a
    int i = 0;
    for (int j = 0; j < counters.length; j++) {
        for (int k = 0; k < counters[j]; k++) {
            a[i] = j;
            i++;
        }
    }
}
```

Bucket sort code v2

```
// Sorts the contents of a using bucket sort algorithm.
// Works for any range of integers in a.
public static void bucketSort(int[] a) {
    int min = Integer.MAX_VALUE;   // find range of values
    int max = Integer.MIN_VALUE;   // stored in a
    for (int k : a) {
        max = Math.max(max, k);
        min = Math.min(min, k);
    }
    int[] counters = new int[max - min + 1];
    for (int k : a) {
        counters[k - min]++;
    }
    int i = 0;
    for (int j = 0; j < counters.length; j++) {
        for (int k = 0; k < counters[j]; k++) {
            a[i] = j + min;
            i++;
        }
    }
}
```

Bucket sort runtime

```
bucketSort(a, length=N):
  C = new int[M]                          // O(M)
  for (k : a) {C[k]++}                    // O(N)
  for (j : C) {                           // T(N/2)
    for (C[j]) times:  a[i] = j.          // O(M)
```

- $O(M + N)$
- ~ $O(N)$ time when $N >>> M$
- *linear runtime on sorting! w00t*

Radix sort

- **radix sort**: Makes several passes of bucket sort, one for each "digit" or significant part of each element.

  The algorithm:
  - Create an array of queues *C* of size 10.
  - For each digit #*i* from least to most significant:
    - For each element in *a*, if its digit #*i* has value is *k*, add it to queue *C*[*k*].
    - Use the queues in *C* to place *a*'s contents back in sorted order.

  - O(*k N*) for *N* elements with *k* digits each
  - lower memory usage than bucket sort

Radix sort example

- input array *a*:

index	0	1	2	3	4	5	6	7	8	9	10	11
value	714	128	206	34	722	8	142	533	646	29	240	373

- sort by last digit, then by tens digit, then by hundreds digit:

index	0	1	2	3	4	5	6	7	8	9	10	11
value	240	722	142	533	373	714	034	206	646	008	128	029

index	0	1	2	3	4	5	6	7	8	9	10	11
value	206	008	714	722	128	029	533	034	240	142	646	373

index	0	1	2	3	4	5	6	7	8	9	10	11
value	008	029	034	128	142	206	240	373	533	646	714	722

Radix sort, detailed

- input array *a*:

index	0	1	2	3	4	5	6	7	8	9	10	11
value	714	128	206	34	722	8	142	533	646	29	240	373

- create array of queues, ordered by last digit:

index	0	1	2	3	4	5	6	7	8	9
value	240		722, 142	533, 373	714, 34		206, 646		8, 128	29

index	0	1	2	3	4	5	6	7	8	9	10	11
value	240	722	142	533	373	714	34	206	646	8	128	29

- put elements back into *a*, sorted by last digit. ...

Radix sort code

```
// Arranges the elements in the array into ascending order
// using the "radix sort" algorithm, which sorts into ten
// queues by ones digit, then tens, ... until sorted.
@SuppressWarnings("unchecked")
public static void radixSort(int[] a) {
    // initialize array of 10 queues for digit value 0-9
    Queue<Integer>[] buckets =
                     (Queue<Integer>[]) new Queue[10];
    for (int i = 0; i < buckets.length; i++) {
        buckets[i] = new ArrayDeque<Integer>();
    }

    // copy to/from buckets repeatedly until sorted
    int digit = 1;
    while (toBuckets(a, digit, buckets)) {
        fromBuckets(a, buckets);
        digit++;
    }
}
```

Radix sort code, continued

```
// Organizes the integers in the array into the given array
// of queues based on their digit at the given place.
// For example, if digit == 2, uses the tens digit, so array
// {343, 219, 841, 295} would be put in queues #4, 1, 4, 9.
// Returns true if any elements have a non-zero digit value.
private static boolean toBuckets(int[] a, int digit,
                                 Queue<Integer>[] buckets) {
    boolean nonZero = false;
    for (int element : a) {
        int which = kthDigit(element, digit);
        buckets[which].add(element);
        if (which != 0) {
            nonZero = true;
        }
    }
    return nonZero;
}
```

Radix sort code, continued

```
// Moves the data in the given array of queues back into the
// given array, in ascending order by bucket.
private static void fromBuckets(int[] a,
                                Queue<Integer>[] buckets) {
    int i = 0;
    for (int j = 0; j < buckets.length; j++) {
        while (!buckets[j].isEmpty()) {
            a[i] = buckets[j].remove();
            i++;
        }
    }
}
// Returns the k'th least significant digit from the integer.
// For example, kthDigit(9814728, 3) returns 7.
private static final int kthDigit(int element, int k) {
    for (int i = 1; i <= k - 1; i++) {
        element = element / 10;
    }
    return element % 10;
}
```

Stooge sort

- **stooge sort**: A silly sorting algorithm with the following algorithm:

  stoogeSort(a, min, max):

  - if a[min] and a[max] are out of order: swap them.
  - stooge sort the first 2/3 of a.
  - stooge sort the last 2/3 of a.
  - stooge sort the first 2/3 of a, again.

- Surprisingly, it works!
- It is very inefficient. **O($N^{2.71}$)** on average, slower than bubble sort.
- Named for the Three Stooges, where Moe would repeatedly slap the other two stooges, much like stooge sort repeatedly sorts 2/3 of the array multiple times.

Stooge sort code

```
public static void stoogeSort(int[] a) {
    stoogeSort(a, 0, a.length - 1);
}

private static void stoogeSort(int[] a, int min, int max) {
    if (min < max) {
        if (a[min] > a[max]) {
            swap(a, min, max);
        }
        int oneThird = (max - min + 1) / 3;
        if (oneThird >= 1) {
            stoogeSort(a, min, max - oneThird);
            stoogeSort(a, min + oneThird, max);
            stoogeSort(a, min, max - oneThird);
        }
    }
}
```