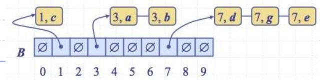
Lesson 8
PriorityQueueSort and RadixSort:
Discovering the Range of Natural Law



Wholeness of the Lesson

All the comparison-based sorting algorithms we have seen so far have at least one worst case for which running time is Ω(nlog n). Bucket Sort and its relatives, under suitable conditions, run in linear time in the worst case, but are not comparison-based algorithms. *Science of Consciousness*: Each level of existence has its own laws of nature. The laws of nature that operate at one level of existence may not apply to other levels of existence, e.g., macroscopic vs. subatomic, classical vs. quantum field level.

PQ-Sort and Radix-Sort 1

1

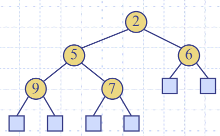
Overview

- Priority Queue ADT
- Sorting with a Priority Queue
- Bucket Sort
  - Each bucket contains elements with the same key
  - Disadvantage: requires too many buckets
- Radix Sort
  - By pairing with Bucket Sort, the number of buckets is reduced to a practical number
- Bucket Sort (Generic)
  - Number of buckets is reduced by allowing each bucket to contain keys within a range rather than one key per bucket
  - Disadvantage: Each bucket has to be sorted ==> buckets need to contain approximately the same number of elements

Priority Queues & Sorting 2

2

Priority Queues



Priority Queues & Sorting 3

3

Priority Queue ADT

- A priority queue stores a collection of sortable elements
- Main methods of the Priority Queue ADT (Java Interface)
  - insertItem(k, e) inserts the item (k, e)
  - removeMin() removes the item with smallest key and returns its associated element
- Additional methods
  - minKey() returns, but does not remove, the smallest key of an item
  - minValue() returns, but does not remove, the element of an item with smallest key
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market
  - Sorting

Priority Queues & Sorting 4

4

Comparator ADT

- A comparator encapsulates the action of comparing two objects according to a given total order relation
- A generic priority queue uses an auxiliary comparator
- The comparator is external to the keys being compared
- When the priority queue needs to compare two keys, it uses its comparator
- The Comparator Method returns a number as its return type indicating the relative size of its input arguments
  - compare(x, y)
    returns a negative number if x<y, zero if x=y, and a positive number if x>y

Priority Queues & Sorting 5

5

Sorting with a Priority Queue

- We can use a priority queue to sort a set of comparable elements
  - Insert the elements one by one with a series of insertElem(e) operations
  - Remove the elements in sorted order with a series of removeMin() operations
- The running time of this sorting method depends on the priority queue implementation. Why?
- Would we need to change *S.remove*(*S.first*()) for efficiency, if S is a Sequence? What about *S.remove*(*S.last*())?

Algorithm *PQ-Sort*(*S, C*)
  Input List *S*, comparator *C* for the elements of *S*
  Output Sequence *S* sorted in increasing order according to *C*

  PQ ← new priority queue using *C*
  while *S.size*() > 0 **do** O(n)
    e ← *S.remove* (S.first()) O(n)
    PQ.insertItem(*e, e*) O(n*?)
  while *P.size*() > 0 **do** O(n)
    e ← *PQ.removeMin*() O(n*?)
    S.insertLast(*e*) O(n)

Priority Queues & Sorting 6

6

Sequence-based Priority Queue

- Implementation with an unsorted list

  4 — 5 — 2 — 3 — 1

- Performance:
  - insertElem takes $O(1)$ time since we can insert the item at the end of the sequence
  - removeMin, minKey, and minElem take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

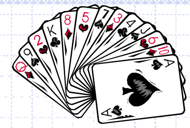
- Implementation with a sorted list

  1 — 2 — 3 — 4 — 5

- Performance:
  - insertElem takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin, minKey, and minElem take $O(1)$ time since the smallest key is at the beginning of the sequence (requires circular array)

7

Selection-Sort



- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence

  4 — 5 — 2 — 3 — 1

- Running time of Selection-sort:
  - Inserting the elements into the priority queue with n *insertElem* operations take $O(n)$ time
  - Removing the elements in sorted order from the priority queue with n *removeMin* operations take time proportional to

    $$n + \ldots + 2 + 1$$
- Selection-sort runs in $O(n^2)$ time

8

Recall SelectionSort

Algorithm *SelectionSort (arr)*
Input Array *arr*
Output elements in *arr* are in sorted order

```
for i ← 0 to arr.length - 2 do
    min ← arr[i]
    minIndex ← i
    for j ← i + 1 to arr.length - 1 do // select next smallest element
        if arr[j] < min then
            min ← arr[j]
            minIndex ← j
    if i != minIndex then
        swapElements(arr, i, minIndex) // place element in sorted location
```

Algorithm *swapElements(arr, i, j)*

```
temp ← arr[i]
arr[i] ← arr[j]
arr[j] ← temp
```


9

Insertion-Sort

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence

  1 — 2 — 3 — 4 — 5

- Running time of Insertion-sort:
  - Inserting the elements into the priority queue with n *insertElem* operations take time proportional to

    $$1 + 2 + \ldots + n$$
  - Removing the elements in sorted order from the priority queue with a series of n *removeMin* operations take $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

10

Recall InsertionSort

Algorithm *InsertionSort(arr)*
Input Array *arr*
Output elements in *arr* are in sorted order

```
for i ← 1 to arr.length - 1 do
    j ← i
    temp ← arr[i]
// this loop corresponds to inserting into the PQ
    while j > 0 ∧ temp < arr[j - 1] do
        arr[j] ← arr[j - 1]        // shift element to right
        j ← j – 1
    arr[j] ← temp
```


11

Analysis of Heap Operations

- insert(e)
- removeMin()
- minimum()
- Helpers
  - upheap
  - downHeap

Analysis of Heap-based Priority Queue Operations

- insertItem(k, e)
- removeMin()
- minKey()
- minValue()
- size()
- isEmpty()

12

Analysis of Sorting with a Heap-based Priority Queue

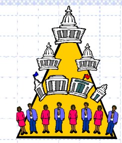
- What is the running time of this sorting method if the priority queue is implemented as a Heap?

```
Algorithm PQ-Sort(S, C)
  Input List S, comparator C for
      the elements of S
  Output Sequence S sorted in
      increasing order according to C

  PQ ← new priority queue using C
  while S.size() > 0 do
      e ← S.remove (S.first())
      PQ.insertItem(e, e)
  while PQ.size() > 0 do
      e ← PQ.removeMin()
      S.insertLast(e)
```


13

Analysis of Heap-Based Priority Queue



- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods insertItem and removeMin take $O(\log n)$ time
  - methods size, isEmpty, minKey, and minElement take $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

14

Main Point

1. A heap is a binary tree that stores *keys* at each internal node and maintains the *heap-order* property on each path from root to leaf and the tree is *balanced*. The Heap ADT is the most efficient way of implementing a Priority Queue. The difference between a PQ and a Heap is that the PQ stores key element items, rather than only keys.

   Science of Consciousness: Pure consciousness is the field of wholeness, perfectly orderly, and complete. Experience of pure consciousness is the basis for efficient, life-supporting action that benefits individual and society.

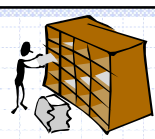
15

Comparison-Based Sorting Algorithms

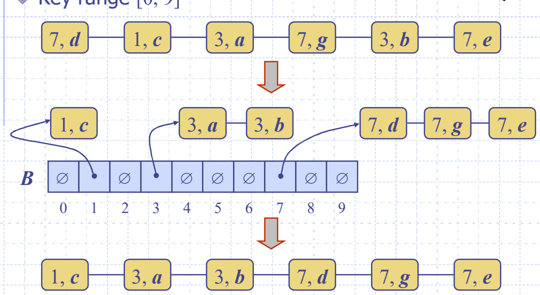
- All sorting algorithms discussed so far have used comparison as a central operation
- So far, all sorting algorithms discussed have a worst-case running time of Ω(nlog n)
- It can be proven that sorting by key comparison requires at least nlog n comparisons to sort n elements, i.e., Ω(nlog n)

16

Example

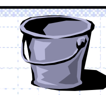


- Key range $[0, 9]$



17

Bucket-Sort



- Let S be a list containing n (key,element) items with keys in the range $[0, N-1]$
- Bucket-sort uses the keys as indices into an auxiliary array B of lists (buckets)
  - Phase 1: Empty list L by moving each item (k, o) into its bucket $B[k]$
  - Phase 2: For $i = 0, \ldots, N-1$, move the items of bucket $B[i]$ to the end of list L
- Analysis:
  - Phase 1 takes $O(n)$ time
  - Phase 2 takes $O(n + N)$ time

Bucket-sort takes $O(n + N)$ time

```
Algorithm bucketSort(L, N)
  Input List L of (key, element) items
      with keys in the range [0, N – 1]
  Output List L sorted by increasing
keys

  B ← array of N empty lists              N
  while ¬L.isEmpty() do                   n
      (k, o) ← L.remove(L.first())        n
      B[k].insertLast( (k, o) )           n
  for i ← 0 to N – 1 do                   N
      while ! B[i].isEmpty() do           N+n
          f ← B[i].first()                n
          (k, o) ← B[i].remove(f)         n
          L.insertLast( (k, o) )          n

  T(n) is O(N+n)
```


18

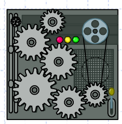
Expanding the Context

- Negative integers can be included if lower bound on these integers is known in advance. Similarly, integers in any range [a,b] can be sorted; worthwhile if b-a is O(n)
  Example: Sorting 100 integers that lies in the range from -100 to 100.
- If m is too big (i.e., not O(n)), BucketSort is not useful because scanning the bucket array is too costly

Main Point

2. BucketSort is an example of a sorting algorithm that runs in O(n). This is possible only because BucketSort does not rely primarily on key comparisons in order to perform sorting.
   Science of Consciousness: This phenomenon illustrates two points from SCI. First, to solve a problem, often the best approach is to bring a new element to the situation (in this case, an array of buckets); this is the Principle of the Second Element. The second point is that different laws of nature are applicable at different levels of creation. Deeper levels are governed by more comprehensive and unified laws of nature.

Radix-Sort



- What do we do if BucketSort isn't efficient because the range is too big?
  - Since it would run in O(n+N)
- Radix-sort is a refinement of BucketSort that uses multiple bucket sorts

What is the meaning of Radix?

- The base of a number system,
  - e.g., base 10 in decimal numbers or base 2 in binary numbers
  - aka radix 10 or radix 2

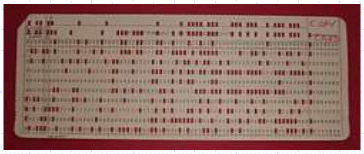
Radix Sort

- The algorithm used by card sorting machines (now found only in museums)
  - Cards were organized into 80 columns such that a hole could be punched in 12 possible slots per column
  - The sorter was mechanically "programmed" to examine a given column of each card and distribute the card into one of 12 bins
- What if we need to sort more than one column?
  - Radix sort solves the problem
  - Requires a stable sorter (defined below)


```
    /&-0123456789ABCDEFGHIJKLMNOPQR/STUVWXYZ
Y /  x           xxxxxxxxx
X |   x                   xxxxxxxxx
0 |    x                           xxxxxxxxx
1 |     x        x        x        x
2 |      x        x        x        x
3 |       x        x        x        x
4 |        x        x        x        x
5 |         x        x        x        x
6 |          x        x        x        x
7 |           x        x        x        x
8 |            x        x        x        x
9 |             x        x        x        x
  |
```

Punch Cards

From a Fortran program: Z(1) = Y + W(1)



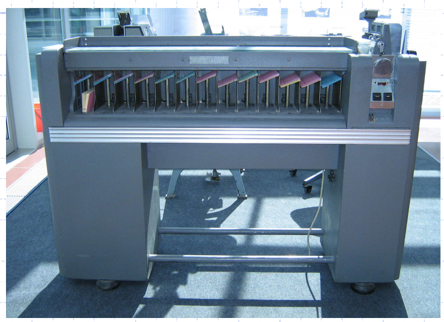
Binary punch card

25

25

26

26



27

27

Lexicographic Sort

28

28

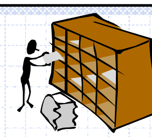
Stable Sorting

- **Stable** Sort Property
  - The relative order of any two items with the same key is preserved after the execution of the algorithm
- Not all sorting algorithms preserve this property

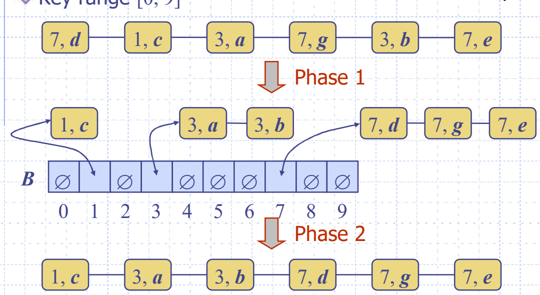
29

29

Example



- Key range [0, 9]



30

30

Lexicographic Order

- A d-tuple is a sequence of d keys $(k_1, k_2, \ldots, k_d)$, where key k_i is said to be the i-th dimension of the tuple
- Example:
  - The Cartesian coordinates of a point in space are a 3-tuple
- The lexicographic order of two d-tuples is recursively defined as follows

$$(x_1, x_2, \ldots, x_d) < (y_1, y_2, \ldots, y_d)$$
$$\Leftrightarrow$$
$$x_1 < y_1 \vee (x_1 = y_1 \wedge (x_2, \ldots, x_d) < (y_2, \ldots, y_d))$$

I.e., the tuples are compared by the first dimension, then by the second dimension, etc.

31

31

Lexicographic-Sort

- Let C_i be the comparator that compares two tuples by their i-th dimension
- Let *stableSort*(S, C) be a stable sorting algorithm that uses comparator C
- Lexicographic-sort sorts a sequence of d-tuples in lexicographic order by executing d times algorithm *stableSort*, one per dimension
- Lexicographic-sort runs in $O(dT(n))$ time, where $T(n)$ is the running time of *stableSort*

Algorithm *lexicographicSort*(S)
Input sequence S of d-tuples
Output sequence S sorted in lexicographic order

for $i \leftarrow d$ **downto** 1
  stableSort(S, C_i)

Example:

(7,4,6) (5,1,5) (2,4,6) (2,1,4) (3,2,4)

(2,1,4) (3,2,4) (5,1,5) (7,4,6) (2,4,6)

(2,1,4) (5,1,5) (3,2,4) (7,4,6) (2,4,6)

(2,1,4) (2,4,6) (3,2,4) (5,1,5) (7,4,6)

32

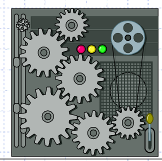
32

- This kind of ordering is sometimes used when records are keyed by multiple fields
- Question:
  - What is the meaning of "radix"?

33

33

Radix-Sort



- Radix-sort is a specialization of lexicographic-sort that uses bucket-sort as the stable sorting algorithm in each dimension
- Radix-sort is applicable to tuples where the keys in each dimension i are integers in the range $[0, N-1]$
- Radix-sort runs in time $O(d(n+N))$

Algorithm *radixSort*(S, N)
Input sequence S of d-tuples such that $(0, \ldots, 0) \le (x_1, \ldots, x_d)$ and $(x_1, \ldots, x_d) \le (N-1, \ldots, N-1)$ for each tuple $(x_1, \ldots, x_d)$ in S
Output sequence S sorted in lexicographic order

for $i \leftarrow d$ **downto** 1
  replace the key k of each item (k, x) of S with dimension x_i of x
  bucketSort(S, N)

34

34

Radix-Sort for Binary Numbers

- Consider a sequence of n b-bit integers
  $x = x_{b-1} \ldots x_1 x_0$
- We represent each element as a b-tuple of integers in the range $[0, 1]$ and apply radix-sort with $N = 2$
- This application of the radix-sort algorithm runs in $O(bn)$ time
- For example, we can sort a sequence of 32-bit integers in linear time

Algorithm *binaryRadixSort*(S)
Input sequence S of b-bit integers
Output sequence S sorted
  replace each element x of S with the item $(0, x)$
for $i \leftarrow 0$ **to** $b - 1$
  replace the key k of each item (k, x) of S with bit x_i of x
  bucketSort$(S, 2)$

35

35

Example

- Sorting a sequence of 4-bit integers

1001		0010		1001		1001		0001
0010		1110		1101		0001		0010
1101	→	1001	→	0001	→	0010	→	1001
0001		1101		0010		1101		1101
1110		0001		1110		1110		1110

36

36

Summary of Sorting Algorithms

Algorithm	Time	Notes (pros & cons)
selection-sort	$O(n^2)$	slow even on small inputs
insertion-sort	$O(n^2)$ or $O(n+k)$	excellent for small inputs, fast for 'almost' sorted
heap-sort	$O(n \log n)$	in-place, fewest comparisons of any sort
PQ-sort	$O(n \log n)$	not in-place, fast but needs supplemental data structure
bucket-sort radix-sort	$O(n+N)$ $O(d(n+N))$	if integer keys & keys known, faster than heap-sort

37

37

Main Point

3. A radix-sort does successive bucket sorts, one for each "digit" in the key beginning with the least significant digit going up to the most significant; it has linear running time.
Science of Consciousness: The nature of life is to grow and progress; Natural Law unfolds in perfectly orderly sequence that gives rise to the universe, all of manifest creation.

38

38

Standard Bucket Sort (another version)

- Three phases
  1. Distribution into buckets
  2. Sorting the buckets (the keys are not the same in the buckets)
  3. Combining the buckets

39

39

1. Distribution

- Each key is examined once
  - a particular field of bits is examined or
  - some work is done to determine in which bucket it belongs
    - e.g., the key is compared to at most k preset values
- The item is then inserted into the proper bucket
- The work done in the distribution phase must be $\Theta(n)$

40

40

2. Sorting the Buckets

- Most of the work is done here
- O(m log m) operations are done for each bucket
  - m is the bucket size

3. Combining the Buckets

- The sorted sequences are concatenated
- Takes $\Theta(n)$ time

41

41

Analysis of Standard Bucket Sort

- If the keys are evenly distributed among the buckets and there are *k* buckets
  - Then the size of the buckets m = n/k
  - Thus the work (key comparisons) done would be
    c m log m for each of the k buckets
  - That is, the total work would be
    k c (n/k) log (n/k) = c n log (n/k)
- If the number of buckets k = n/20, then the size of each bucket (n/k) is equal to 20, so the number of key comparisons would be
  c n log 20
- Thus bucket sort would be linear when the input comes from a uniform distribution
- Note also that the larger the bucket size, the larger the constant (log m)

42

42

Main Point

3. We can prove that the lower bound on sorting by key comparisons in the best and worst cases is Ω(n log n). However, we can do better, i.e. linear time, but only if we have knowledge of the structure and distribution of keys. This knowledge is the basis for organizing a more efficient algorithm for sorting, but can only be applied in rare circumstances.

   Science of Consciousness: Knowledge has organizing power. Deeper levels are governed by more comprehensive and unified laws of nature and have greater organizing power; pure knowledge has infinite organizing power.

43

43

Summary of Sorting Algorithms

Algorithm	Time	Notes (pros & cons)
insertion-sort		
Shell-sort		
heap-sort		
PQ-sort		
standard bucket-sort		

44

44

Summary of Sorting Algorithms

Algorithm	Time	Notes (pros & cons)
insertion-sort	$O(n^2)$ or $O(n+k)$	excellent for small inputs, fast for 'almost' sorted, simple, in-place
Shell-sort	$O(n^{3/2})$	fast even for fairly large inputs, simple, in-place
heap-sort	$O(n \log n)$	fast, as few comparisons as any sort, in-place
PQ-sort	$O(n \log n)$	not in-place, fast, excellent if input needs to be unchanged
Generic bucket-sort	$O(n \log(n/k))$	if keys can be distributed evenly into relatively small bucket sizes

45

45

Overview

- Priority Queue ADT
- Sorting with a Priority Queue
- Bucket Sort
  - Each bucket contains elements with the same key
  - Disadvantage: requires too many buckets
- Radix Sort
  - By pairing with Bucket Sort, the number of buckets is reduced to a practical number
- Bucket Sort (Generic)
  - Each bucket contains keys within a range to reduce the number of buckets required
  - Disadvantage: Each bucket has to be sorted ==> buckets need to contain approximately the same number of elements

Priority Queues & Sorting 46

46

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. Comparison-based sorting algorithms can achieve a worst-case running time of Θ(n log n), but can do no better.

2. Under certain conditions on the input, Bucket Sort and Radix Sort can sort in O(n) steps, even in the worst case. The nlog n lower bound does not apply because these algorithms are not comparison-based.

47

47

3. **Transcendental Consciousness** is the field of all possibilities and of pure orderliness. Contact with this field brings to light new possibilities and leads to spontaneous orderliness in all aspects of life.
4. **Impulses within Transcendental Consciousness**: The organizing power of pure knowledge is the lively expression of the Transcendent, giving rise to all expressions of intelligence.
5. **Wholeness moving within itself**: In Unity Consciousness, the organizing dynamics at the source of creation are appreciated as an expression of one's own Self.

48

48