Ch8: Linear-time sorts
Ch11: Hash tables

8 Oct 2013
CMPT231
Dr. Sean Ho
Trinity Western University

Outline for today

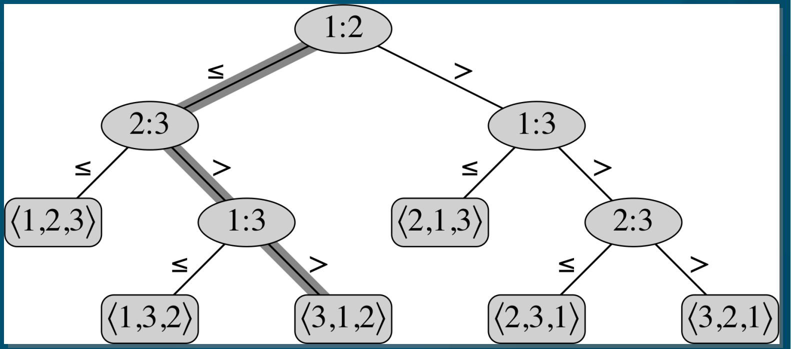
  - Proof why comparison sorts must be Ω(n lg n)
- Linear-time non-comparison sorts:
  - Counting sort
  - Radix sort, complexity
  - Bucket sort: proof w/ probabilistic analysis
- Hash tables:
  - Collision handling by chaining
  - Hash functions and universal hashing
  - Collision handling by open addressing

Summary of sorting algorithms

- Comparison sorts (ch2, 6, 7)
  - Insertion sort: $\Theta(n^2)$, easy to program, slow
  - Merge sort: $\Theta(n \lg(n))$, out-of-place sorting, slow due to lots of copying / memory operations
  - Heap sort: $\Theta(n \lg(n))$, in-place, uses max-heap
  - Quick sort: $\Theta(n^2)$ worst-case, $\Theta(n \lg(n))$ average, in-place, fast (small) constant factors
- Linear-time non-comparison sorts (ch8):
  - Counting sort: k distinct values: $\Theta(n+k)$
  - Radix sort: d digits w/k values: $\Theta(d(n+k))$
  - Bucket sort: for uniform distrib. of values: $\Theta(n)$

Comparison sorts are $\Omega(n \lg n)$

- Decision tree model of computation:
  - Leaves are possible outputs
    - i.e., permutations of the input
  - Nodes are decision points
    - when comparisons are made
  - Path through tree is one run on an input
- # leaves = # permutations = $n!$
- # comparisons = # nodes along path
  - = depth of tree
  - = $\Omega(\lg(\# \text{ leaves})) = \Omega(\lg n!)$
  - = $\Omega(n \lg n)$ (by Stirling, Eq3.19)



Outline for today

  - Proof why comparison sorts must be Ω(n lg n)
- Linear-time non-comparison sorts:
  - Counting sort
  - Radix sort, complexity
  - Bucket sort: proof w/ probabilistic analysis
- Hash tables:
  - Collision handling by chaining
  - Hash functions and universal hashing
  - Collision handling by open addressing

Linear-time sorts

- Linear-time sorts use assumptions on input data
  - e.g., range of possible values is limited+known
  - e.g., distribution of values is known
- In practise, Θ(n) and Θ(n lg n) are very similar
    - e.g., up to $n=10^6$: lg n < 21, a smallish factor
  - A fast n lg n sort (like quicksort) may have smaller constants than a linear-time sort
- Hybrid algorithms: e.g., *(7.4-5)*
  - Pass 1 w/quicksort, stop when length < c
  - Pass 2 w/insertion sort on "nearly sorted" data
- Recursion (function call) is expensive

Counting sort

- Assume: values are integers in {0, ..., k}
  - countingSort(A, n, k):

```
countingSort(A, n, k):
    let B[1 .. n] be new (output) array
    let C[0 .. k] be temp array, initialised to 0
census  for j in 1 .. n:
            C[ A[ j ] ] ++
        for i in 1 .. k:
            C[ i ] += C[ i-1 ]
move    for j in n .. 1:
            B[ C[ A[ j ] ] ] = A[ j ]
            C[ A[ j ] ] --
    return B
```

A:	2	5	3	0	2'	3'	0'	3''
C:	2	0	2	3	0	1		
C:	2	2	4	7	7	8		
B:	0	0'	2	2'	3	3'	3''	5

- Stable: preserves order of duplicate keys
- Complexity: Θ(n+k) *(watch out if k gets big!)*

Radix sort

  - (How IBM made its fortune! punch cards ~1900)
- Sort one digit at a time, least-significant first
- Assume: values have max #digits d
  - radixSort(A, n, d):
    - for i in 1 .. d:
      - stableSort(A on digit i)
  - stableSort() can be, e.g., counting sort
  - (why is stability important?)
  - (why start from least-significant digit?)

3	7	4	5
2	9	1	3
1	0	1	6
2	0	1	6
	9	1	3

Radix sort: complexity

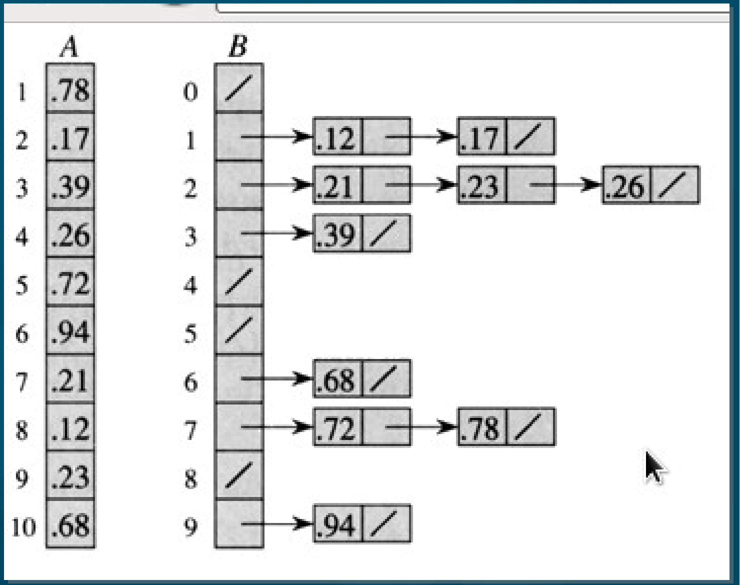
- Using counting sort, we have d loops of $\Theta(n+k)$:
  - ⇒ complexity of radix sort is $\Theta(d(n+k))$
    - n items of d digits,
      where each digit can take k values (e.g., $k=10$)
- Given b-bit items, find r to get optimal r-bit digits:
    - $d = b/r$ and $k = 2^r - 1$
      - e.g., 32-bit ints, 8-bit digits ⇒ $b=32$, $r=8$, $d=4$, $k=255$
    - Complexity is $\Theta(d(n+k)) = \Theta((b/r) (n + 2^r))$
  - Balance the b/r with the $n + 2^r$
    - e.g., by choosing $r = \lg n$:
    - $\Theta((b/r) (n + 2^r)) = \Theta((b / \lg n) (2n)) = \Theta(bn / \lg n)$
      - e.g., to sort $n=2^{16}$ ints of $b=32$-bits, use $r=16$-bit digits

Outline for today

  - Proof why comparison sorts must be Ω(n lg n)
- Linear-time non-comparison sorts:
  - Counting sort
  - Radix sort, complexity
  - Bucket sort: proof w/ probabilistic analysis
- Hash tables:
  - Collision handling by chaining
  - Hash functions and universal hashing
  - Collision handling by open addressing

Bucket sort

- Assume: values uniformly distributed over [0,1)
- Idea: Divide range [0,1) into n equal-size buckets
    - e.g., each bucket can be a small array or linked list
  - Distribute input into buckets
  - Sort each bucket
    - e.g., by insertion sort
    - should be fast because we expect small buckets
  - Pull from each bucket in order



Kent U

- Correctness: if $A[i] \le A[j]$, then either:
  - $A[i]/n = A[j]/n$ (same bucket: insertion sort), or
  - $A[i]/n < A[j]/n$ (diff bucket: order of buckets)

Bucket sort: complexity

- Let n_i = # items in i^{th} bucket
  - Intuitively, if items are uniformly distrib, $n_i \approx 1$
  - so whole thing should be T(n) = Θ(n)
- To be rigorous: notice that $T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$
- Find expected value:

$$E[T(n)] = E[\Theta(n) + \sum O(n_i^2)] = \Theta(n) + O(\sum E[n_i^2])$$

- Claim that $E[n_i^2] = 2 - (1/n)$ for all i:
  - if so, then E[T(n)] = Θ(n) + O(Σ (2 – 1/n))
    = Θ(n) + O(2n – 1)
    = Θ(n), and the proof is complete

Bucket sort: $E[n_i^2] = 2 - (1/n)$

- Use indicator variable:
  - $X_{ij} = 1$ if A[j] falls in bucket i, and 0 if not
  - So $n_i = \Sigma_j X_{ij}$ (count of items in this bucket)
- So $E[\ n_i^2\ ] = E[\ (\ \Sigma_j X_{ij}\ )^2\ ]$ (count items)

  $= \Sigma_j E[\ X_{ij}^2\ ] + 2\Sigma_j \Sigma_k E[\ X_{ij}X_{ik}\ ]$ (expand)
- Consider each term separately:
  - Applying probability rules:

    $E[\ X_{ij}^2\ ] = 0^2\ P(X_{ij} = 0) + 1^2\ P(X_{ij} = 1)$

    $= 0^2\ (1 - 1/n) + 1^2\ (1/n) \quad = 1/n$
  - Since items $j \neq k$ are independent:

    $E[\ X_{ij}X_{ik}\ ] = E[\ X_{ij}\ ]\ E[\ X_{ik}\ ] = (1/n)(1/n) = 1/n^2$

Bucket sort: finish proof

- So $E[\ n_i^2\ ] = \Sigma_j\ E[\ X_{ij}^2\ ] + 2\Sigma_j\Sigma_k\ E[\ X_{ij}X_{ik}\ ]$

  $= \Sigma_j\ (1/n) + 2\Sigma_j\Sigma_k\ (1/n^2)$

  $= (1/n)\ \Sigma_j\ (1) + (2/n^2)\ \Sigma_j\Sigma_k\ (1)$

  $= (1/n)(n) + (2/n^2)(\ n(n-1)/2\ )$

  $= 1 + n(n-1)/n^2$

  $= 2 – 1/n$

- Hence expected running time for bucket sort is $E[\ T(n)\ ] = \Theta(n) + \Sigma\ (2 – 1/n)$

  $= \Theta(n) + 2n - 1$

  $= \Theta(n)$, linear time

- Assumptions: input uniformly distributed on $[0,1)$

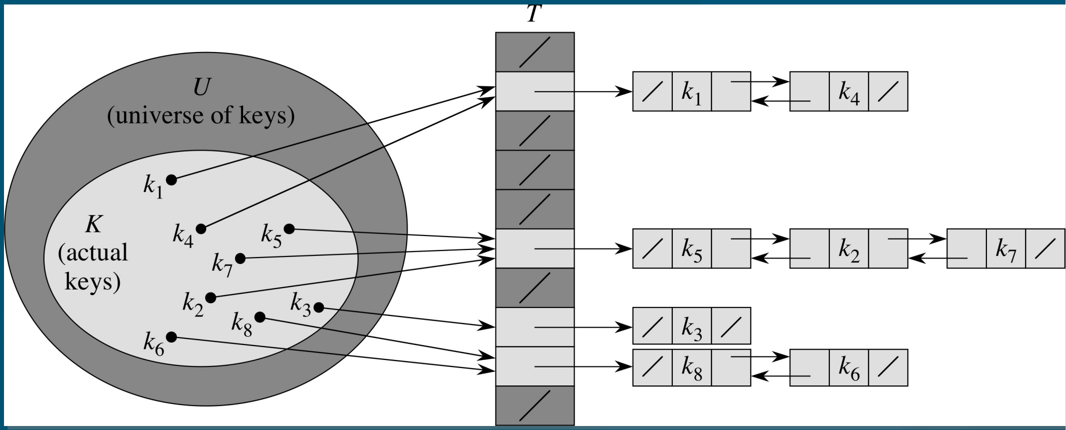
Outline for today

  - Proof why comparison sorts must be Ω(n lg n)
- Linear-time non-comparison sorts:
  - Counting sort
  - Radix sort, complexity
  - Bucket sort: proof w/ probabilistic analysis
- Hash tables:
  - Collision handling by chaining
  - Hash functions and universal hashing
  - Collision handling by open addressing

Hash tables

- Dictionary of key-value pairs, e.g., Python dict
- Interface:
  - insert(T, k, x): add item x with key k
  - search(T, k): find an item with key k
  - delete(T, x): delete specific item x
- Better than regular array (direct addressing) when
  - Range of possible keys is too huge to allocate
  - Actual keys are sparse subset of possible keys
  - e.g., only have items at keys 0, 2, 40201300
    - Regular array would allocate 40201300 entries!

Hashing



- Main idea:
  - Hash function
    h(k): U → {0, …, m-1}
    maps from set U of possible keys
    into a set of m buckets
  - Use h(k) as key instead of k
- Hash collision when two keys hash to same bucket
  - Hopefully, this is rare
  - Chain multiple items via linked list
- Idea is similar to bucket sort, but
  - Don't know distribution or range of keys, so
  - Use hash function to get uniform distribution

Implementing hash tables

- insert(T, k, x):
  - Insert x at the head of the linked list at slot h(k)
  - Complexity: O(1)
  - Assumes x is not already in the list
- search(T, k):
  - Linear search through the list at slot h(k)
  - Complexity: O(*length of list at h(k)*)
- delete(T, x):
  - If given pointer directly to item x, then O(1)
  - If not, then need to do a search first

Hash table load factor

- Efficiency of hash table depends on search()
  - Which depends on # items $n_{h(k)}$ in each bucket
- Load factor $\alpha = n/m$:
  - n = # items currently stored in hash table
  - m = # buckets
- So $E[\ n_{h(k)}\ ] = \alpha$ (average # items per bucket)
- An unsuccessful search takes average $\Theta(1 + \alpha)$:
  - Computing hash function takes $\Theta(1)$
  - Linear search needs to search entire bucket
  - Expected length of bucket is α

Complexity of search()

- A successful search also takes average $\Theta(1 + \alpha)$:
- # items searched = # collisions after x inserted
- Use indicator X_{ij} = { 1 if $h(k_i) = h(k_j)$, 0 else }
  - $E[X_{ij}]$ = (prob. of collision) = $1/m$
- $E[\text{ \# items searched }] = E[(1/n) \Sigma_i(\text{ \#items })]$

$$
\begin{aligned}
&= E[(1/n) \Sigma_i(1 + \Sigma_j X_{ij})] \\
&= (1/n) \Sigma_i(1 + \Sigma_j E[X_{ij}]) \\
&= (1/n) \Sigma_i(1 + \Sigma_j (1/m)) \\
&= 1 + (1/n) \Sigma_i\Sigma_j (1/m) \\
&= 1 + (1/nm)\, n(n-1)/2 \\
&= 1 + \alpha/2 - \alpha/2n
\end{aligned}
$$

Outline for today

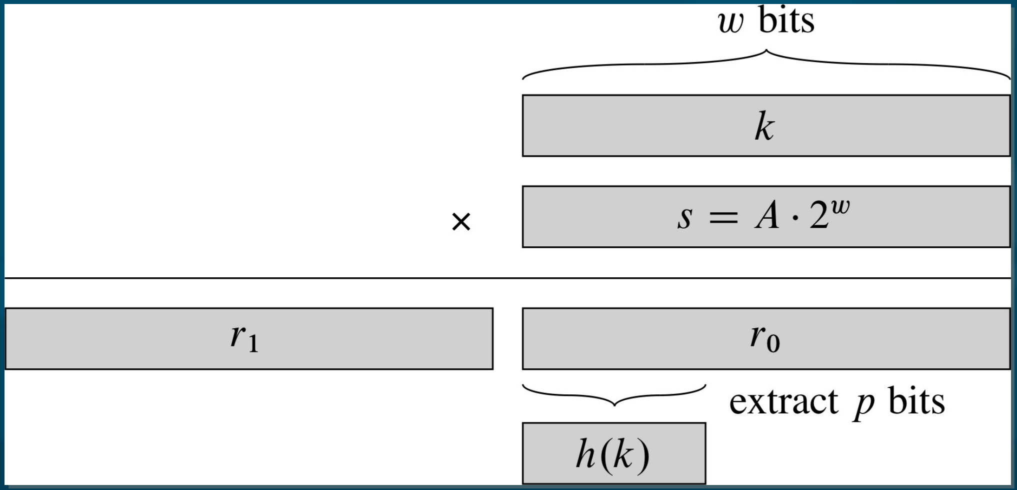
  - Proof why comparison sorts must be Ω(n lg n)
- Linear-time non-comparison sorts:
  - Counting sort
  - Radix sort, complexity
  - Bucket sort: proof w/ probabilistic analysis
- Hash tables:
  - Collision handling by chaining
  - Hash functions and universal hashing
  - Collision handling by open addressing

Hash functions

- Wlog, assume keys k are natural numbers
  - If not, convert (e.g., ASCII codes)
- Want h(k) to be uniformly distributed on 0..m-1
  - But distribution of keys k is unknown
  - Keys k_i and k_j might not be independent
- Division hash: h(k) = k mod m
  - Fast, but if $m=2^p$, this is just the p least-sig bits
    - If k is a string using radix-2^p representation, then permuting the string gives same hash (*11.3-3*)
  - Choose m prime, not too close to a power of 2

Multiplication hash

- Multiplication hash: $h(k) = \lfloor m(kA \bmod 1) \rfloor$, where $0<A<1$ is some chosen constant
- Fast implementation using $m=2^p$:
  - Let w be the native machine word size (#bits)
  - Pick a w-bit integer s in $0 < s < 2^w$, let $A = s/2^w$
  - Multiply $s*k$: product has 2w bits in words r_0, r_1
  - Select the p most-sig bits of the lower word r_0



try $A \approx \varphi - 1$?

Universal hashes

- Any fixed choice of hash function is vulnerable to pathological input specifically designed to obtain many hash collisions
- Keep a pool H of hash functions, randomly select
- Want pool to have the universal hash property:
  - For any two keys $j \neq k$, the number of hash functions in H that cause a collision $h(j) = h(k)$ is $\leq |H| / m$
- Then expected size of buckets is $O(1+\alpha)$, and complexity of search is still $O(1)$.

Outline for today

  - Proof why comparison sorts must be Ω(n lg n)
- Linear-time non-comparison sorts:
  - Counting sort
  - Radix sort, complexity
  - Bucket sort: proof w/ probabilistic analysis
- Hash tables:
  - Collision handling by chaining
  - Hash functions and universal hashing
  - Collision handling by open addressing

Open addressing

- Another way to handle collisions, instead of chain
- Keys stored directly in table, no linked lists
- To search:
  - Probe in slot $h(k)$:
    - if NIL, unsuccessful search (and we're done)
    - if the entry is our key, we've found it
    - if the entry is not our key, we hit a collision:
      - Try again with next entry in probe sequence
- Hash function $h: U \times \{0, \ldots, m-1\} \rightarrow \{0, \ldots, m-1\}$
  - Probe sequence: $h(k,0)$, $h(k,1)$, $h(k,2)$, …
    - Must be a permutation of the slots $\{0, \ldots, m-1\}$
- Hash table may fill/overflow

Probe sequencing

- Ideally, want uniform hashing: each permutation is equally likely to be probe sequence for a key
- Linear probing:
  - First try $h(k)$, then $h(k)+1$, etc (mod m)
  - Long filled runs get longer (more likely to hit)
- Quadratic probing:
  - First try $h(k)$, then jump around quadratically:
    - $h(k, i) = (h(k) + c_1 i + c_2 i^2) \bmod m$
  - Must choose c_1, c_2 to get full permutation
  - Collision on initial $h(k)$ ⇒ full sequence collision

Probe seq.: double hashing

- Use two hash functions h_1 and h_2:
  - Try $h_1(k)$ first, then use h_2 to jump around:
    - $h(k, i) = (h_1(k) + i\ h_2(k)) \bmod m$
  - In order to get full permutation, $h_2(k)$ and m must be relatively prime
    - e.g., let $m=2^p$ and ensure $h_2(k)$ always odd
    - or, let m be prime, and ensure $1 < h_2(k) < m$
- Each combination of $h_1(k)$ and $h_2(k)$ yields a different probe sequence:
  - total # sequences = $\Theta(n^2)$