# Searching – Key Points

- searching vs lookup
  - more efficient algorithms are possible when the collection of elements is fixed (no insertions or deletions)
- sequential search
  - can be applied to any traversable collection
- binary search
  - requires a sorted array
  - deceptively tricky to implement correctly – thorough testing requires searching for every key as well as every option for between keys



# Searching – Key Points

- how many elements?
  - sequential search is fastest for small collections (≤ 20), even if they are already sorted
  - beyond 100 elements (and especially if multiple searches), better to sort first and use binary search
  - for huge data sets (too big for memory), use B-trees or another data structure
- are some elements accessed more often than others?
  - take advantage – order list accordingly or build optimal binary search tree
- do access frequencies change over time?
  - *self-organizing* structures – *move-to-front heuristic* (lists), *splay trees*
- unbounded range, or element thought to be nearby?
  - *one-sided binary search* – repeatedly double the search interval size, extending one side of the interval, until the upper bound is found (then regular binary search)



| Question | build an optimal binary search tree | use B-trees | use one-sided binary search | use a self-organizing structure, such as a list with the move-to-front heuristic or a splay tree | use sequential search | sort, then use binary search | use binary search |
|---|---|---|---|---|---|---|---|
| unequal access frequencies, in a static environment (fixed set of elements, known and stable patterns of access) | ✓ 1 | 2 | 1 | 1 | 1 | 1 | 0 |
| huge data set | 0 | ✓ 5 | 0 | 1 | 0 | 0 | 1 |
| an unbounded search range | 0 | 0 | ✓ 1 | 2 | 1 | 1 | 2 |
| target might be close by | 0 | 0 | ✓ 3 | 0 | 2 | 1 | 1 |
| unequal access frequencies, in a dynamic environment (changing set of elements and/or changing patterns of access) | 1 | 0 | 1 | ✓ 5 | 0 | 0 | 0 |
| a small number of elements, sorted | 1 | 1 | 0 | 0 | ✓ 1 | 1 | 3 |
| a small number of elements, unsorted | 1 | 0 | 0 | 0 | ✓ 4 | 2 | 0 |

optimal BST is optimized to provide fastest possible access for given patterns of access

expensive to compute, so want to use it for many searches

binary search isn't limited to collections – it can be applied to continuous domains, such as numbers – but requires a defined search interval

sequential search is efficient if the target *is* close by; one-sided binary search mitigates the high cost if it isn't

B-trees are a balanced search tree designed for external memory applications

without locality of reference, assumption of all memory accesses being equal time breaks down when external memory is required



| Question | build an optimal binary search tree | use B-trees | use one-sided binary search | use a self-organizing structure, such as a list with the move-to-front heuristic or a splay tree | use sequential search | sort, then use binary search | use binary search |
|---|---|---|---|---|---|---|---|
| unequal access frequencies, in a static environment (fixed set of elements, known and stable patterns of access) | ✓ 1 | 2 | 1 | 1 | 1 | 1 | 0 |
| huge data set | 0 | ✓ 5 | 0 | 1 | 0 | 0 | 1 |
| an unbounded search range | 0 | 0 | ✓ 1 | 2 | 1 | 1 | 2 |
| target might be close by | 0 | 0 | ✓ 3 | 0 | 2 | 1 | 1 |
| unequal access frequencies, in a dynamic environment (changing set of elements and/or changing patterns of access) | 1 | 0 | 1 | ✓ 5 | 0 | 0 | 0 |
| a small number of elements, sorted | 1 | 1 | 0 | 0 | ✓ 1 | 1 | 3 |
| a small number of elements, unsorted | 1 | 0 | 0 | 0 | ✓ 4 | 2 | 0 |

self-organizing works in a static environment, but the gains from using the optimal structure offsets the expense of building it if it will be used many times

self-organizing can extend the size for which something like sequential search is fast enough (less frequently used elements get pushed to the end/bottom so don't get in the way), but "huge" is "too big to fit into memory"

sequential search is simpler than binary search and can be faster for a small collection

it can be used for larger collections, but is likely no longer the best choice

binary search requires a sorted array

sorting and then binary search is always possible, but there may be better options for particular situations

## Selection – Key Points

- selection – find *k*th element (commonly median)
  - searching locates an element by value, selection locates an element by rank
- applications
  - order statistics (*k* = 1, *k* = *n*, *k* = *n*/2 – min, max, median)
  - quantiles – median, quartiles, deciles, percentiles, …
  - filtering – extract or eliminate top/bottom percentage



## Selection – Key Points

- how fast do you need?
  - find *k*th is Θ(1) in a sorted array – can be worthwhile to sort if you need multiple *k* values
  - *quick-select* – Θ(n) expected, Θ($n^2$) worst-case (unlikely)
    - based on quicksort – pick pivot, divide into smaller and bigger groups, but then can determine which group will contain the *k*th element and recurse only on that
- too much data to fit in memory, or to even contemplate storing?
  - *random sampling* to save only a sample of the data set for processing
  - compute quantiles for chunks of data, then combine to approximate the overaall measures



Which of the following are suitable for selection with very large and/or streaming data sets? Choose all that apply.

| | Answer | Respondents | Percentage |
|---|---|---|---|
| ✕ | sort the data, then pick the kth element | 2 | 14% |
| ✕ | use quick select | 1 | 7% |
| ✓ | use a random sample of the data for selection | 5 | 36% |
| ✓ | compute quantiles for blocks of data, then combine those to estimate quantiles for the whole data set | 6 | 43% |

sorting and quick-select lack locality of reference so are not efficient with external memory



## Applications of Hashing

- hashing has (many) applications beyond lookup
  - data integrity and verification
    - checksums
  - data deduplication and managing collections of unique elements
  - pattern matching
    - compare hashes to quickly check if two strings are not equal – only check characters if the hashes match
  - cryptography and data security
    - store hashed passwords instead of plaintext
    - digital signatures
- key properties
  - hash function is deterministic
  - (a good) hash function distributes keys well / few collisions
  - hash function is not reversible

| Answer | Respondents | Percentage |
|---|---|---|
| ✓ finding duplicate elements in a large dataset | 6 | 14% |
| ✓ finding the most common element in a list | 7 | 17% |
| ✓ fingerprinting digital files for fast comparison | 6 | 14% |
| ✓ inexact pattern matching in text search | 3 | 7% |
| ✓ detecting plagiarism in documents | 5 | 12% |
| ✓ verifying data integrity (proof of unmodified content) | 4 | 10% |

similar words hash to the same or nearby values – canonicalization e.g. Soundex, locality-sensitive hashing

break down words into smaller pieces and compare those – n-gram hashing

extensions to string-matching algorithms that using hashing to quickly compare strings (e.g. Rabin-Karp)

generate a hash for the original document and send along with the document receiver generates hash for the received document and compares hashes – very unlikely two different documents will have the same hash



| | | |
|---|---|---|
| ✕ sorting | 1 | 2% |
| ✕ compressing large files to save space | 3 | 7% |
| ✕ encrypting sensitive information | 4 | 10% |
| ✕ generating random numbers | 3 | 7% |

hash functions distribute keys without maintaining any order

hashing is one-way – different things hash to the same value so you can't reconstruct the original from the hash

hash functions may be part of a pseudorandom number generator, but aren't a source of randomness on their own – deterministic, require input, no guarantee of uniform distribution