Algorithms and Data Structures, Fall 2011

Sorting and searching case studies

Rasmus Pagh

Based on slides by Kevin Wayne, Princeton

Today's lecture

We will focus on these intended learning outcomes:

- **Choose** among and make use of the most important algorithms and data structures in libraries, based on knowledge of their complexity.
- **Design** algorithms for ad hoc problems by using and combining known algorithms and data structures.

The lecture will be based on a number of case studies, many of which have previously been posed as problems in the course.

In SW section 3.5 you can find further examples.

Case study: A small database

Employees

Id	Name	Salary
1	Joe White	60,000
2	Will Jones	55,000
…	…	…

ProjectAssignment

Id	BossID	Project
1	1	APL
2	4	GUI
…	…	…

Projects

Id	Year	Profit
APL	1999	-100,000
GUI	2000	30,000
…	…	…

Case study: A small database

Some database queries

- Given the name of an employee, find his/her id and salary.
- Find the total profit of all projects in the year 2000.
- Find employees who have a higher salary than another employee who was their boss in some project.

Some database updates

- Change the salary of an employee.
- Add a new employee or project.
- Assign an employee to work on a project.

Employees

Id	Name	Salary
1	Joe White	60,000
2	Will Jones	55,000
...	...	...

ProjectAssignment

Id	BossID	Project
1	1	APL
2	4	GUI
...	...	...

Projects

Id	Year	Profit
APL	1999	-100,000
GUI	2000	30,000
...	...	...

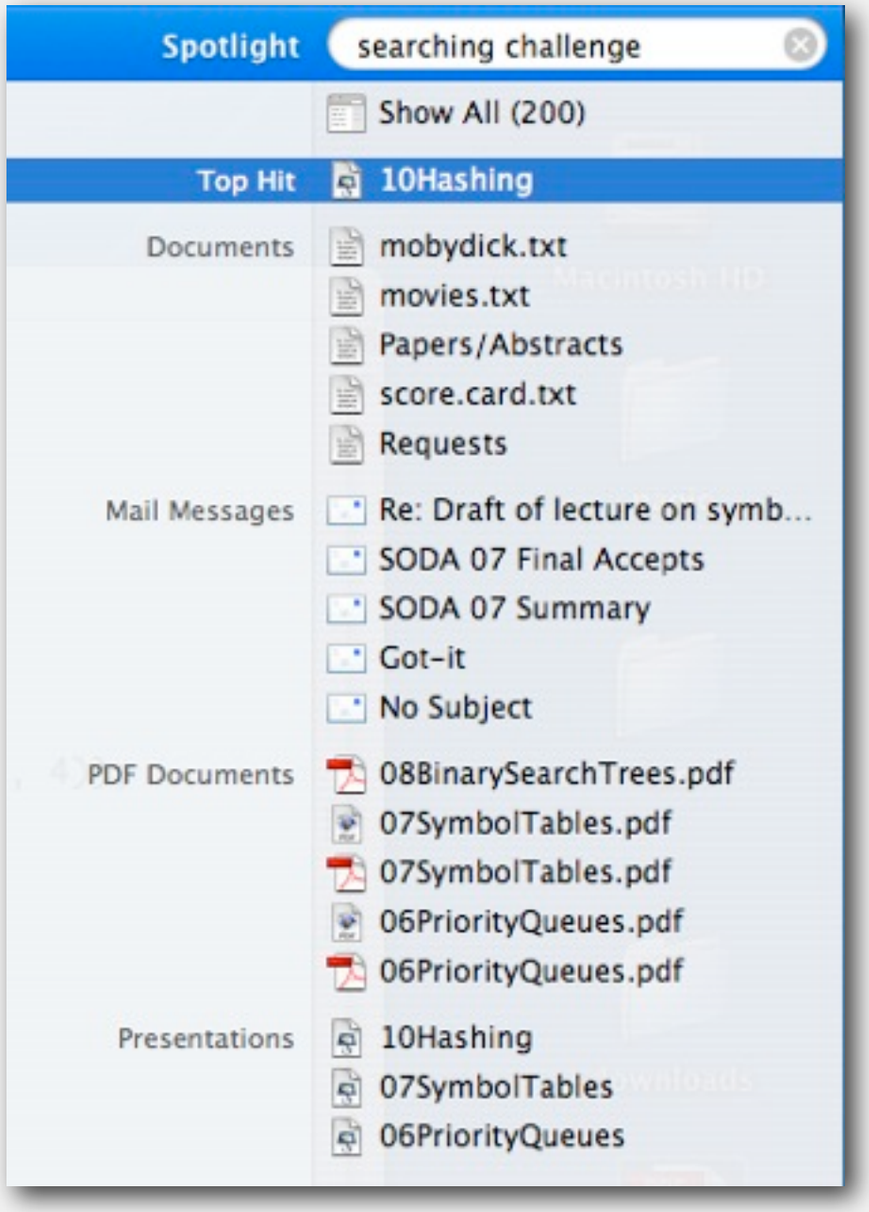
Task: Consider possibilities for indexing.

Notation: na assignments, ne employees.

Case study: File indexing

Goal. Index a PC (or the web).

Simplified goal: Given two or more words, report the list of (text) files in which these words occur.



Case study: File indexing

Inverted index: For each word, list the files that contain it.

Building an inverted index:

Problem session!

Case study: File indexing

Inverted index: For each word, list the files that contain it.

Building an inverted index:

- Replace file names by small ID numbers: Use a symbol table to look up.
- Fill a symbol table:
  - Key = word
  - Value = unbounded array of file IDs

Can also use these symbol tables to answer queries.

Problem: What if data is so large that it does not fit in RAM: Thrashing!

File system model

Page. Contiguous block of data (e.g., a file or 4096-byte chunk).

Probe. First access to a page (e.g., from disk to memory).

slow

fast

Property. Time required for a probe is much larger than time to access data within a page.

Cost model. Number of probes.

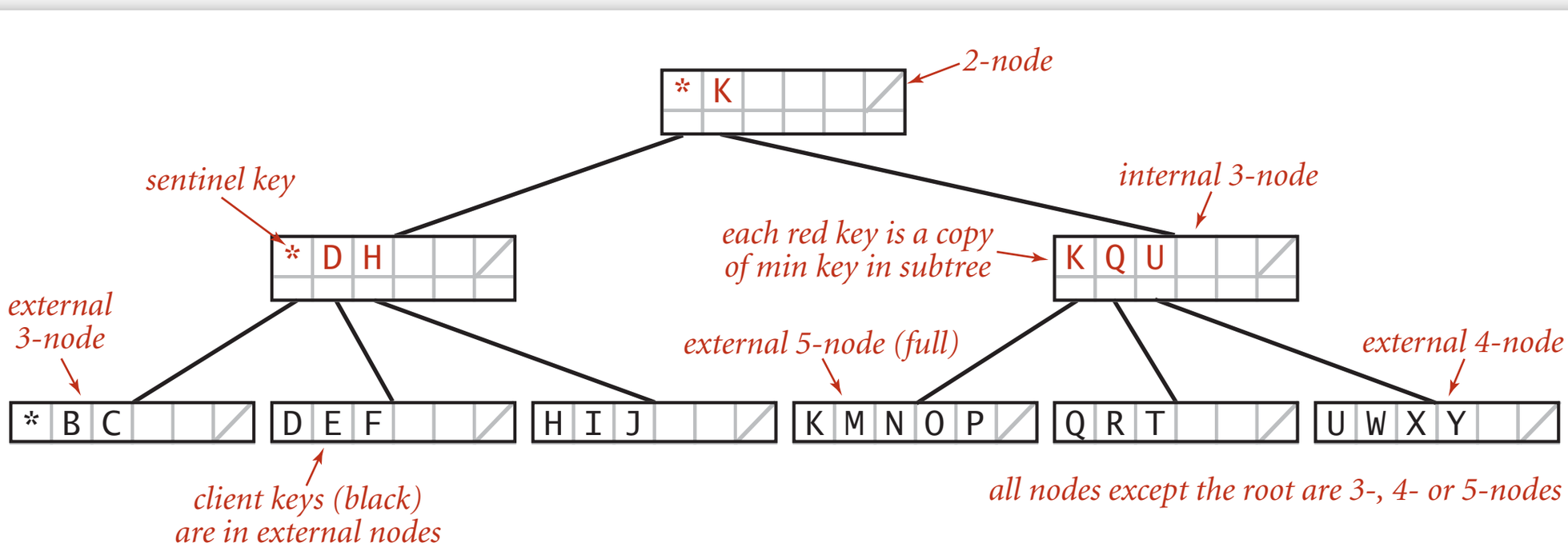
Goal. Access data using minimum number of probes.

B-trees (Bayer-McCreight, 1972)

B-tree. Generalize 2-3 trees by allowing up to $M - 1$ key-link pairs per node.

- At least 2 key-link pairs at root.
- At least $M / 2$ key-link pairs in other nodes.
- External nodes contain client keys.
- Internal nodes contain copies of keys to guide search.

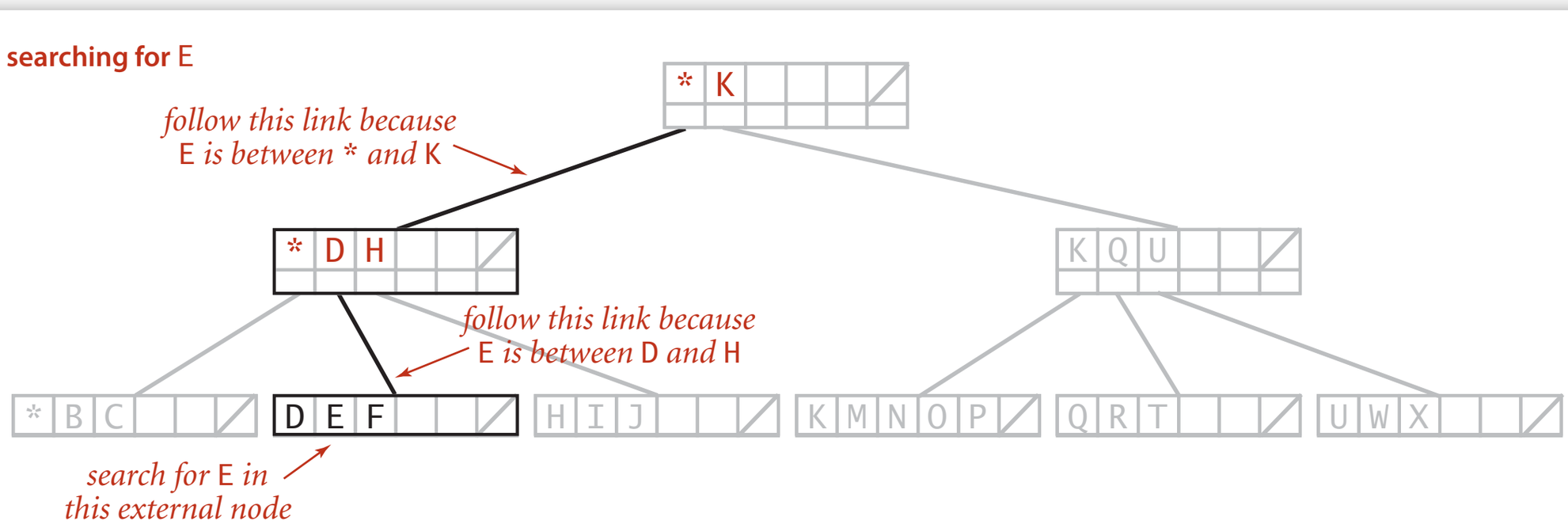
choose M as large as possible so that M links fit in a page, e.g., M = 1024



Anatomy of a B-tree set (M = 6)

Searching in a B-tree

- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.



Searching in a B-tree set (M = 6)

Sorting on disk

Even if B-trees are used, symbol table operations use 1 or more I/Os.

Is there an alternative approach? Yes!

- Replace file names **and words** by small ID numbers: Use a symbol table to look up.
- Create a list of (fileID,wordID) pairs, output it to disk.
- Sort the list according to wordID.
- Traverse the sorted lists to build the inverted index (create symbol table).

Question: How does one best sort data on disk?

- Mergesort, quicksort both read/write items around log n times.

- **Multiway** versions typically read/write each item only twice!

(The amount of CPU work is about the same as if we had all in RAM.)

Checking referential integrity

Suppose you are writing a program to transfer some data about movies, actors, and information about the cast of each movie to a relational database. The program should check *referential integrity*, i.e., whenever actor A stars in movie B it must check that data about actor A and movie B exists before this information can be moved to the database.

1. Describe a solution using binary search trees. What is the asymptotic complexity?
2. Describe an efficient method based on hashing. What can you say about complexity?
3. Describe an efficient method based on sorting that does *not* use binary search. (**Hint:** You need to sort several times.) What is its worst-case time complexity?

Case study: Using trees

For an on-line game site, you maintain a list of each player's best score. Also, the site should be able to display the top 100 scores, scores 101-200, and so on.

Suggest an efficient data structure that allows submission of a new top score (erasing the old top score for that player), and retrieval of selected top scores as described above. You should not design a new data structure, but rely on the toolbox you have seen in the course.

Ordered symbol table API

	keys	*values*
min() →	09:00:00	Chicago
	09:00:03	Phoenix
	09:00:13	Houston
get(09:00:13) →	09:00:59	Chicago
	09:01:10	Houston
floor(09:05:00) →	09:03:13	Chicago
	09:10:11	Seattle
select(7) →	09:10:25	Seattle
	09:14:25	Phoenix
	09:19:32	Chicago
	09:19:46	Chicago
keys(09:15:00, 09:25:00) →	09:21:05	Chicago
	09:22:43	Seattle
	09:22:54	Seattle
	09:25:52	Chicago
ceiling(09:30:00) →	09:35:21	Chicago
	09:36:14	Seattle
max() →	09:37:44	Phoenix

size(09:15:00, 09:25:00) *is* 5

rank(09:10:25) *is* 7

Examples of ordered symbol-table operations

Hashing to identify almost-anagrams

Two strings x_1 and x_2 are anagrams if and only if they contain the same set of letters. This can be checked e.g. by sorting the letters of each string. The strings are *almost-anagrams* if they contain the same letters except for one character change or addition. For example, `test` and `taste` are almost-anagrams, and so are `glad` and `lads`.

We want to use hashing to identify almost-anagrams. As a building block we will use a hashing method to identify anagrams. We note that letters in the English alphabet can be converted to integers in $\{0, \ldots, 255\}$, e.g., in Java one can call `x.getBytes()` to return an array of bytes corresponding to the letters of a string `x`. In the following, we let x_i denote the integer corresponding to the ith character of a string x. We will make use of the prime number $p = 2^{31} - 1$, which in Java can be computed as `p = (1<<31)-1`.

a) Let T be an array of 256 integers in the range $\{0, \ldots, p\}$, and consider the hash function $h(x) = (T[x_1] + T[x_2] + T[x_3] + T[x_4] + T[x_5]) \bmod p$. Observe that if x and x' are anagrams, then $h(x) = h(x')$. Conversely, it is possible to show that if x and x' are not anagrams, and the integers in T are chosen at random, then the probability that $h(x) = h(x')$ is $1/p$. Use this fact to identify all anagrams in the collection of 5-letter words, by inserting their hash values in an associative array. Can you express the expected running time in terms of n (the number of strings), k (the number of anagrams reported), and p?

Anagrams, continued

Note that for any pair x, y of almost-anagrams of length 5, it is possible to remove one letter from each of x and y to produce anagrams (not necessarily English words). For example, `tests` and `stats` can have `a` and `e` removed, respectively, to produce strings that both contain the letters `sstt`. Conversely, if we have two strings x and y such that removing one letter from each gives anagrams, then x and y are almost-anagrams. This suggests that we can identify almost-anagrams in a set S of n of 5-letter strings by identifying anagrams in the set S' of $5n$ 4-letter strings obtained by removing a character from one of the strings in S.

b) Write a procedure that inserts all hash values of elements in S' in an associative array, where each hash value is associated with the set of strings in S that lead to this hash value. For example, `tests` should be in the set of strings associated with $h(\text{test})$, $h(\text{tess})$, $h(\text{tets})$, $h(\text{tsts})$, and $h(\text{ests})$.

Case study: Analysis of trees

```
// input A[1..n], an array of integers
// output x, the integer that occurs the most times in A
//
// The program output is an integer x such that no other integer in A occurs
//    more frequently than x.
// TreeMap implements a balanced search tree with integer keys, and integers
//    as associated information

T:=new TreeMap(Integer,Integer);

for i:=1 to n do {
  if A[i] exists in T then c:=T.lookup(A[i]) else c:=0;
  insert/replace A[i] in T with associated information c+1;
}
x:=A[1];
for i:=1 to n do {
  if (T.lookup(A[i])>T.lookup(x)) then x:=A[i];
}
```

Task:

Analyze the running time of the algorithm.

If needed, suggest an alternative implementation.