Chapter 8

More Computational Complexity: The Searching Problem

Preview

- Overview
- Lecture Notes
  - Lower Bounds for Searching Only by Comparisons of Keys
    - Lower Bounds for Worst-Case Behavior
    - Lower Bounds for Average-Case Behavior
  - Interpolation Search
  - Searching in Trees
    - Binary Search Trees
    - B-Trees
- Quick Check
  - Hashing
  - The Selection Problem: Introduction to Adversary Arguments
    - Finding the Largest Key
    - Finding Both the Smallest and Largest Keys
    - Finding the Second-Largest Key
    - Finding the *k*th-Smallest Key
    - A Probabilistic Algorithm for the Selection Problem
- Quick Check
- Classroom Discussion
- Homework

- ❖ Keywords
- ❖ Important Links

Overview

Like sorting, searching is one of the most useful applications in computer science. The problem is usually to retrieve an entire record based on the value of some key field. We want to analyze the problem of searching and show that we have obtained searching algorithms whose time complexities are about as good as our lower bounds.

In searching for a phone number, Barney Beagle uses an **Interpolation Search**, which does more than just compare keys. When he is looking for Colleen Collie’s number, he starts at the beginning of the phone book because he knows the letter “C” is at the beginning of the book. We present Interpolation Search and show that an array does not meet other needs of certain applications. We also show that although Binary Search is optimal, the algorithm cannot be used for other applications because it relies on array implementation. We show that trees do not meet these needs, and we discuss tree searching. We will also cover searching when it is not important that the data be retrieved in a sorted sequence. We will also discuss hashing in this section.

Lecture Notes

Lower Bounds for Searching Only by Comparisons of Keys

The problem of searching for a key can be described as follows: Given an array *S* containing *n* keys and a key *x*, find an index *i* such that *x* = *S* [*i*] if *x* equals one of the keys; if *x* does not equal one of the keys, report failure.

Binary Search is very efficient for solving this problem when the array is sorted. Its worst-case time complexity is [lg n] +1. As long as we limit ourselves to algorithms that search only by comparisons of keys, there is no improvement possible. Algorithms that search for a key *x* in an array only by comparisons of keys can compare keys with each other or with the search key *x*, and they can copy keys, but they cannot do other operations on them. To assist their search they can use the knowledge that the array is sorted. We will obtain bounds for deterministic algorithms, and we will assume that the keys are distinct.

As we did for deterministic sorting algorithms, we can associate a decision tree with every deterministic algorithm that searches for a key *x* in an array of *n* keys. Refer to Figures 8.1 and 8.2 for illustrations of decision trees based on Binary Search and Sequential Search. In these trees, each large node represents a comparison of an array item with the search key *x*, and each small node (leaf) contains a result that is reported. When *x* is in the array, we report an index of the item that it equals, and when *x* is not in the array, we report an “F” for failure. We assume that a searching algorithm never changes any array values, so these are still the values after the search is completed.

Each leaf in a decision tree for searching *n* keys for a key *x* represent a point at which the algorithm stops and reports an index *i* such that *x* = *si* or reports failure. Every internal node represents a comparison. Every algorithm that searches for a key *x* in an array of *n* keys has a corresponding pruned, valid decision tree. A searching algorithm need not always compare *x* with an array item. It can compare two array items. Because we are assuming that all keys are distinct, the outcome will be equality only when *x* is being compared. In the cases of both Binary Search and Sequential Search, when the algorithm determines that *x* equals an array item, it stops and returns the index of the array item. An inefficient algorithm may continue to do comparisons when *x* equals an array item and return the index later. We can replace such an algorithm with one that stops at this point and returns the index. The new algorithm will be at least as efficient as the original one.

Because there are only three possible results of a comparison, a deterministic algorithm can only take three different paths at each comparison. Each comparison node can only have three children. Because equality must lead to a leaf that returns an index, only two of the children can be comparison nodes. Therefore, the set of the comparison nodes in the tree constitutes a binary tree.

Lower Bounds for Worst-Case Behavior

Because every leaf in a pruned, valid decision tree must he reachable, the worst-case number of comparisons done by such a tree is the number of nodes in the longest path from the root to a leaf in the binary tree consisting of the comparison nodes. This number is the depth of the binary tree plus 1. Therefore, to establish a lower bound on the worst-case number of comparisons, we need only establish a lower bound on the depth of the binary tree consisting of the comparison nodes. Refer to the text for the applicable lemmas and theorem.

Lower Bounds for Average-Case Behavior

A binary tree is called a **nearly complete binary tree** if it is complete down to the depth of *d* – 1. Every essentially complete binary tree is nearly complete, but not every nearly complete binary tree is essentially complete. Refer to the text for a discussion of Lemma 8.3.

Binary Search 's average-case performance is not much better than its worst case. We can see why this is so by looking at Figure 8.1. In that figure, there are more result nodes at the bottom of the tree than there are in the rest of the tree. This is true even if we don't consider unsuccessful searches. Refer to the text for a discussion of Lemmas 8.4, 8.5, and 8.6.

We have established that Binary Search is optimal in its average-case performance given specific assumptions about the probability distribution. It may not be optimal for other probability distributions.

Interpolation Search

Remember that Barney Beagle does more than just compare keys to find Colleen Collie's number in the phone book. He interpolates and starts at the front of the phone book because Colleen's last name begins with "C."

Suppose we are searching 10 integers, and we know that the first integer ranges from 0 to 9, the second from 10 to 19, the third from 20 to 29, . . . , and the tenth from 90 to 99. Then we can immediately report failure if the search key *x* is less than 0 or greater than 99, and, if neither of these is the case, we need only compare x with S [1 + |_x/10_|].

In most cases we do not have this much information. But, it is reasonable to assume that the keys are close to being evenly distributed between the first one and the last one. In such cases, instead of checking whether *x* equals the middle key, we can check whether *x* equals the key that is located about where we would expect to find *x*. The algorithm that implements this strategy is called Interpolation Search. Low is set initially to 1 and *high* to *n*. We then use linear interpolation to determine approximately where we feel x should be located. That is, we compute:

[See equations at top of page 348]

If the keys are close to being evenly spaced, Interpolation Search homes in on the possible location of *x* more quickly than does Binary Search. For instance, in the preceding example, if *x* = 25 were less than S[3], Interpolation Search should reduce the instance of size 10 to one of size 2, whereas Binary Search would reduce it to one of size 4.

A drawback of Interpolation Search is its worst-case performance. In the worst cast, Interpolation Search generates to a sequential search. The worst case happens when *mid* is repeatedly set to *low*. A variation of Interpolation Search called **Robust Interpolation Search** remedies this situation by establishing a variable *gap* such that *mid – low* and *high – mid* are always greater than *gap*. Refer to the text for a more detailed discussion.

Under the assumptions that the keys are uniformly distributed and that the search key *x* is equally likely to be in each of the array slots, the average-case time complexity for Robust Interpolation Search is Y(lg(lg *n*). Its worst-case time complexity is Y (lg *n*)2), which is worse than Binary Search but much better than Interpolation Search.

Searching in Trees

Interpolation Search and Robust Interpolation Search are very efficient. But they cannot be used in many applications because an array is not an appropriate structure for storing data in these applications.

Static searching is a process in which the records are all added to the file at one time, and there is no need to add or delete records later. Many applications require **dynamic searching**, which means that records are added and deleted frequently.

An array structure is inappropriate for dynamic searching because, when we add a record in sequence to a sorted array, we must move all the records following the added record. Binary Search requires that the keys be structured in an array, because there must be an efficient way to find the middle item. This means that Binary Search cannot be used for dynamic searching. Although we can readily add and delete records using a linked list, there is no efficient way to search a linked list. If it is necessary to retrieve the keys quickly in sorted sequence, direct-access storage (**hashing**) will not work. Dynamic searching can be implemented efficiently using a tree structure.

Binary Search Trees

The algorithm that builds a binary tree requires that all keys be added at one time, which means that the application requires static searching. Binary trees are also appropriate for dynamic searching. Using a binary search tree, we can usually keep the average search time low, while being able to add and delete keys quickly. We can quickly retrieve the keys in sorted sequence by doing an **in-order traversal** of the tree. This process involves first visiting all the nodes in the left subtree using an in-order traversal, then visiting the root, and finally visiting all the nodes in the right subtree using an in-order traversal.

Figure 8.5 shows the fundamental relationship between Binary Search and binary search trees. Using a binary search tree, we can usually keep the average search time low, while being able to add and delete keys quickly. The drawback of binary search trees is that when keys are dynamically added and deleted, there is no guarantee that the resulting tree will be balanced. One often obtains a **skewed tree**, which is simply a linked list. If the keys are added at random, the resulting tree will be closer to a balanced tree than it will be closer to a linked list. The search time would usually be efficient. Refer to the text for a discussion of Theorem 8.3.

B-Trees

In many applications, performance can be degraded by a linear search time. For example, the keys for records in large databases often cannot all fit into a computer's high-speed memory (RAM). Multiple disk accesses are needed to accomplish the search (**external search**). When all the keys are simultaneously in memory, it is called an **internal search**. A linear-time search could prove to be unacceptable. One solution is to write a balancing program that takes as its input an existing binary tree and outputs a balanced binary search tree containing the same keys.

In a very dynamic environment, it would be better if the tree never became unbalanced in the first place. Algorithms for adding and deleting nodes while maintaining a balanced binary tree were developed by Adel'son-Velskii and Landis. Thus, balanced binary trees are often called **AVL trees**.

Bayer and McCreight developed an improvement over binary search trees called **B-trees**. When keys are added to or deleted from a B-tree, all leaves are guaranteed to remain at the same level, which is even better than maintaining balance.

B-trees actually represent a class of trees, of which the simplest is a **3–2 tree**. We illustrate the process of adding nodes to such trees. A 3–2 tree is a tree with the following properties:

- Each node contains one or two keys.
- If a non-leaf contains one key, it has two children, whereas if it contains two keys, it has three children.
- The keys in the left subtree of a given node are less than or equal to the key stored at that node.
- The keys in the right subtree of a given node are greater than or equal to the key stored at that node.
- If a node contains two keys, the keys in the middle subtree of the node are greater than or equal to the left key and less than or equal to the right key.
- All leaves are at the same level.

The tree remains balances because the tree grows in depth at the root instead of at the leaves. When it is necessary to delete a node, the tree shrinks in depth at the root. In this way, all leaves remain at the same level, and search, addition, and deletion times are guaranteed to be Y (lg *n*). An in-order traversal retrieves the keys in sorted sequence. B-trees are used most in modern database management systems.

Quick Check

1. A decision tree is called ____for searching n keys for a key x if for each possible outcome there is a path from the root to a leaf that reports that outcome.
   Answer: valid
2. A decision tree is called ____ if every leaf is reachable.
   Answer: pruned
3. A binary tree, which is complete down to a depth of d – 1, is called a(n) ____ binary tree
   Answer: nearly complete
4. A process in which the records are all added to the file at one time and there is no need to add or delete records later is called ____.
   Answer: static searching

Hashing

A **hash function** is a function that transforms a key to an index, and an application of a hash function to a particular key is called "hashing the key." In the case of social security numbers, an example of a hash function would be:

$$h\ (key) = key\ \%\ 100$$

This function simply returns the last two digits of the key. If a particular key hashes to i, we store the key and its record at *S* [*i*]. This strategy does not store the keys in sorted sequence, which means that it is applicable only if we never need to retrieve the records efficiently in sorted sequence. If it is necessary to do this, one of the methods discussed previously should be used.

If no two keys hash to the same index, this method works fine. However, this is rarely the case when there are a substantial number of keys. For example, if there are 100 keys and each key is equally likely to hash to each of the 100 indices, the probability of no two keys hashing to the same index is:

$$100! / 100^{100} \sim= 9.3 \times 10^{-43}$$

We are almost certain that at least two keys will hash into the same index. Such an occurrence is called a **collision** or **hash clash**. There are various ways of resolving a collision. One of the best ways is through **open hashing** or **open addressing**. With open hashing, we create a bucket for each possible hash value and place all the keys that hash to a value in the bucket associates with that value. Open hashing is usually implemented through linked lists.

The number of buckets need not equal the number of keys. The more keys we store, the more likely we are to have collisions. Because a bucket stores only a pointer, not much space is wasted by allocating a bucket. It is often reasonable to allocate at least as many buckets as there are keys.

When searching for a key, it is necessary to do a sequential search through the bucket (linked list) containing the key. If all the keys hash into the same bucket, the search degenerates into a sequential search. It is still almost impossible for all the keys to end up in the same bucket.

Intuitively, it should be clear that the best thing that can happen is for the keys to be uniformly distributed in the buckets. That is, if there are n keys and m buckets, each bucket contains n/m keys. Actually, each bucket will contain exactly n/m keys only when n is a multiple of m. When this is not the case, we have an approximately uniform distribution. When the keys are uniformly distributed, we have a very small search time. Refer to the text for a discussion of Theorems 8.4, 8.5, and 8.6.

The Selection Problem: Introduction to Adversary Arguments

This Selection problem finds the kth-largest or kth-smallest key in a list of n keys. We assume that the keys are in an unsorted array. First we discuss the problem for $k = 1$, which means that we are finding the largest or smallest key. Next we show that we can simultaneously find the smallest and largest keys with fewer comparisons than would be necessary if each were found separately. Then we discuss the problem for $k = 2$, which means that we are finding the second-largest or second-smallest key. Finally, we address the general case.

Finding the Largest Key

[See Algorithm 8.2 on page 363]

The number of comparisons of keys done by the algorithm is given by:

$$T(n) = n - 1.$$

It seems impossible to improve on this performance. We can think of an algorithm that determines the largest key as a tournament among the keys. Each comparison is a match in the tournament, the larger key is the **winner** of the comparison, and the smaller key is the **loser**. The largest key is the winner of the tournament. We use this terminology throughout this section.

Finding Both the Smallest and Largest Keys

[See Algorithm 8.3 on page 364]

Using the above algorithm is better than finding the smallest and largest keys independently, because for some inputs the comparison of $S[i]$ with large is not done for every i. Therefore, we have improved on the every-case performance. But whenever $S[1]$ is the smallest key, that comparison is done for all i. Therefore, the worst-case number of comparisons of keys is given by

$$W(n) = 2(n - 1),$$

which is exactly the number of comparisons done if the smallest and largest keys are found separately. It may seem that we cannot improve on this performance, but we can. The trick is to compare the keys and find which key in each pair is smaller. This can be done with about $n/2$ comparisons. We can then find the smallest of all the smaller keys with about $n/2$ comparisons and the largest of the keys with about $n/2$

comparisons. In this way, we find both the smallest and largest keys with about 3*n*/2 total comparisons. Refer to the text for a discussion of Algorithm 8.4.

We use a method, called an **adversary argument**, to establish our lower bound. Suppose some adversary's goal is to make an algorithm work as hard as possible. At each point where the algorithm must make a decision, the adversary tries to choose a result of the decision that will keep the algorithm going as long as possible. The only restriction is that the adversary must always choose a result that is consistent with those already given. As long as the results are consistent, there must be an input for which this sequence of results occurs. If the adversary forces the algorithm to do the basic instruction *f* (*n*) times, then *f* (*n*) is a lower bound on the worst-case time complexity of the algorithm.
Let us denote the status of each key during the algorithm as follows:

KEY STATUS	**MEANING**
W	Only winning
L	Only losing
WL	Winning and losing
N	Not yet participated in a comparison

Table 8.2 shows an adversary strategy that always discloses a minimal amount of information. Table 8.3 shows an adversary's strategy for foiling Algorithm 8.3 for an input size of 5. Refer to the text for a more detailed discussion.

When developing an adversary strategy, our goal is to make algorithms for solving some problems work as hard as possible. A poor adversary may not actually achieve this goal. Whether or not the goal is reached, the strategy can be used to obtain a lower bound on how hard the algorithms must work.

Finding the Second-Largest Key

To find the second-largest key, we can use Algorithm 8.2 to find the largest key with *n* – 1 comparisons, eliminate that key, and then use Algorithm 8.2 again to find the largest remaining key with *n* – 2 comparisons. Therefore, we can find the second-largest key with 2n - 3 comparisons. We should be able to improve on this, because many of the comparisons done when finding the largest key can be used to eliminate keys from contention for the second largest. That is, any key that loses to a key other than the largest cannot be the second largest. The Tournament method uses this fact.

The **Tournament method** is so named because it is patterned after the method used in elimination tournaments. The Tournament method directly applies only when *n* is a power of 2. When this is not the case, we can add enough items to the end of an array to make the array size a power of 2.

Although the winner in the last round is the largest key, the loser in that round is not necessarily the second-largest key. In Figure 8.10, the second-largest key loses to the largest key in the second round. This is a problem with many actual tournaments because the two best teams do not always meet in the championship game. To find the second-largest key, we can keep track of all the keys that lose to the largest key and then use Algorithm 8.2 to find the largest of those. We have to maintain linked lists, one for each key. After a key loses a match, it is added to the winning key's linked list. Refer to the text for a more detailed discussion of Theorem 8.9.

Finding the *k*th-Smallest Key

In general, the Selection problem entails finding the *k*th-largest or *k*th-smallest key. So far, we've discussed finding the largest key, because it has seemed more appropriate for the terms used. That is, it seems appropriate to call the largest key a winner. Here we discuss finding the *k*th-smallest key because it makes our algorithms more clear. For simplicity, we assume that the keys are distinct.

One way to find the *k*th-smallest key in Y (*n* lg *n*) time is to sort the keys and return the *k*th key. We develop a method that requires fewer comparisons. Recall that procedure partition, which is used in Quicksort, partitions an array so that all keys smaller than some pivot item come before it in the array and all keys larger than that pivot item come after it. The slot at which the pivot item is located is called the *pivotpoint*. We can solve the Selection problem by partitioning until the pivot item is at the *k*th slot.
We do this by recursively partitioning the left subarray if *k* is less than *pivotpoint*, and by recursively partitioning the right subarray if *k* is greater than *pivotpoint*. When *k* = p*ivotpoint*, we're done. This divide-and-conquer algorithm solves the problem by this method.

A Probabilistic Algorithm for the Selection Problem

A **Sherwood algorithm** is a probabilistic algorithm that always gives the correct answer. Such an algorithm is useful when some deterministic algorithm runs much faster on the average than it does in the worst case. The worst-case quadratic time is achieved in that algorithm when the *pivotpoint* for a particular input is repeatedly close to low or high in the recursive calls. This happens when the array is sorted in increasing order and *k* = *n.* Because this is not the case for most inputs, the algorithm's average performance is linear. Suppose that, for a particular input, we choose the pivot item at random according to a uniform distribution. Then, when the algorithm is run for that input, the *pivotpoint* is more likely to be away from the endpoints than close to them. Therefore, linear performance is more likely. Because the number of comparisons is linear when averaged over all inputs, intuitively it seems that the expected value of the number of comparisons of keys done for a particular input should be linear when the pivot item is chosen at random according to a uniform distribution.

You may ask why we would want to use such an Algorithm 8.6 guarantees linear performance. The reason is that Algorithm 8.6 has a high constant because of the overhead needed for approximating the median. For a given input, the Sherwood algorithm runs faster on average than Algorithm 8.6. The decision of whether to use Algorithm 8.6 or the Sherwood algorithm depends on the needs of the particular application. If better average performance is needed, we should use the Sherwood algorithm. If quadratic performance can never be tolerated, we should use Algorithm 8.6. Refer to the text for a more detailed discussion of the Sherwood algorithm.

Quick Check

1. A(n) ____is a function that transforms a key to an index
   Answer: hash function
2. A key of n distinct keys is called ____**,** if half the keys are smaller and half are bigger
   Answer: median

3. A method used to establish our lower bound may be called, ____.
   Answer: adversary argument
4. A(n) ___ is used to obtain a candidate solution that can be used in an induction argument.
   Answer: constructive induction

Classroom Discussion

- Discuss an algorithm that creates a 3-2 tree from a list of keys
- Discuss two situations in which hashing is not appropriate

Homework

Assign Exercises 13, 23, and 31.

Keywords

- **3–2 tree –** is a tree with the following properties:
    1. Each node contains one or two keys.
    2. If a non-leaf contains one key, it has two children, whereas if it contains two keys, it has three children.
    3. The keys in the left sub-tree of a given node are less than or equal to the key stored at that node.
    4. The keys in the right sub-tree of a given node are greater than or equal to the key stored at that node.
    5. If a node contains two keys, the keys in the middle sub-tree of the node are greater than or equal to the left key and less than or equal to the right key.
    6. All leaves are at the same level.
- **Adversary argument –** a theoretical agent that uses information about the past moves to choose the worst-case input.
- **AVL trees** – balanced binary trees.
- **B-trees** – a balanced search tree in which all leaves are guaranteed to remain at the same level.
- **Collision** or **hash clash –** when at least two keys will hash to the same index.
- **Dynamic searching** – records are frequently added and deleted.
- **External search –** a search where multiple disk accesses are needed to accomplish the search
- **Hash function –** a function that transforms a key to an index.
- **Hashing –** an information storage and retrieval technique.
- **In-order traversal –** a process that involves first visiting all the nodes in the left sub-tree using, then visiting the root, and finally visiting all the nodes in the right subtree.
- **Interpolation search –** a search of an assorted array done by estimating the next position to check based on a linear interpolation of the search key and the values at the ends of each interval.
- **Internal search –** a search where all the keys are simultaneously in memory.
- **Loser –** the smaller key of a comparison.
- **Median –** of *n* distinct keys is the key such that half the keys are smaller and half are larger
- **Min-TND –** the minimum of the TND for binary trees containing n nodes.
- **Nearly complete binary tree –** a binary tree that is complete down to a depth of *d* –1.
- **Node distance** – total number of nodes in the path from the root to a node.
- **Open hashing** or **open addressing** – one of the best ways to resolve a collision.
- **Robust Interpolation –** a variation of Interpolation Search that establishes a variable *gap.*
- **Selection problem** – a problem to find the *k*th-largest or *k*th smallest key in a list of *n* keys.
- **Sherwood algorithm –** an algorithm that always gives the correct answer.
- **Skewed tree –** tree whose keys are all added in increasing sequence.
- **Static searching –** a process in which the records are all added to the file at one time and there is no need to add or delete records later.
- **Total node distance (TND)** – sum of the node distances to all nodes.
- **Tournament method –** a method of selecting individuals from a population. The individual keys are chosen at random, with the larger key being identified as the **winner** and the smaller key being identified as the **loser**.
- **Valid decision tree –** a decision tree is valid for sorting n keys if, for each permutation of the n keys, there is a path from the root to a leaf that sorts that permutation.
- **Winner –** the larger key of a comparison.

Important Links

- http://www.nist.gov/dads/ (Dictionary of Algorithms and Data Structures)
- http://www.cs.cornell.edu/boom/2004sp/ProjectArch/Chess/algorithms.html (AI Chess Algorithms)