|  |
| --- |
| *Data Structures and Algorithms* |
| **4 Searching** |

Computer systems are often used to store large amounts of data from which individual records must be retrieved according to some search criterion. Thus the efficient storage of data to facilitate fast searching is an important issue. In this section, we shall investigate the performance of some searching algorithms and the data structures which they use.

**4.1 Sequential Searches**

Let's examine how long it will take to find an item matching a key in the collections we have discussed so far. We're interested in:

1. the average time
2. the worst-case time and
3. the best possible time.

However, we will generally be most concerned with the worst-case time as calculations based on worst-case times can lead to guaranteed performance predictions. Conveniently, the worst-case times are generally easier to calculate than average times.

If there are ***n*** items in our collection - whether it is stored as an array or as a linked list - then it is obvious that in the worst case, when there is no item in the collection with the desired key, then ***n*** comparisons of the key with keys of the items in the collection will have to be made.

To simplify analysis and comparison of algorithms, we look for a dominant operation and count the number of times that dominant operation has to be performed. In the case of searching, the dominant operation is the comparison, since the search requires n comparisons in the worst case, we say this is a ***O(n)*** (pronounce this "big-Oh-n" or "Oh-n") algorithm. The best case - in which the first comparison returns a match - requires a single comparison and is ***O(*1*)***. The average time depends on the probability that the key will be found in the collection - this is something that we would not expect to know in the majority of cases. Thus in this case, as in most others, estimation of the average time is of little utility. If the performance of the system is vital, i.e. it's part of a life-critical system, then we must use the worst case in our design calculations as it represents the best guaranteed performance.

**4.2 Binary Search**

However, if we place our items in an array and sort them in either ascending or descending order on the key first, then we can obtain much better performance with an algorithm called **binary search**.

In binary search, we first compare the key with the item in the middle position of the array. If there's a match, we can return immediately. If the key is less than the middle key, then the item sought must lie in the lower half of the array; if it's greater then the item sought must lie in the upper half of the array. So we repeat the procedure on the lower (or upper) half of the array.

Our FindInCollection function can now be implemented:

static void \*bin\_search( collection c, int low, int high, void \*key ) {

int mid;

/\* Termination check \*/

if (low > high) return NULL;

mid = (high+low)/2;

switch (memcmp(ItemKey(c->items[mid]),key,c->size)) {

/\* Match, return item found \*/

case 0: return c->items[mid];

/\* key is less than mid, search lower half \*/

case -1: return bin\_search( c, low, mid-1, key);

/\* key is greater than mid, search upper half \*/

case 1: return bin\_search( c, mid+1, high, key );

default : return NULL;

}

}

void \*FindInCollection( collection c, void \*key ) {

/\* Find an item in a collection

Pre-condition:

c is a collection created by ConsCollection

c is sorted in ascending order of the key

key != NULL

Post-condition: returns an item identified by key if

one exists, otherwise returns NULL

\*/

int low, high;

low = 0; high = c->item\_cnt-1;

return bin\_search( c, low, high, key );

}

Points to note:

1. bin\_search is recursive: it determines whether the search key lies in the lower or upper half of the array, then calls itself on the appropriate half.
2. There is a termination condition (two of them in fact!)
   1. If low > high then the partition to be searched has no elements in it *and*
   2. If there is a match with the element in the middle of the current partition, then we can return immediately.
3. AddToCollection will need to be modified to ensure that each item added is placed in its correct place in the array. The procedure is simple:
   1. Search the array until the correct spot to insert the new item is found,
   2. Move all the following items up one position *and*
   3. Insert the new item into the empty position thus created.
4. bin\_search is declared static. It is a local function and is not used outside this class: if it were not declared static, it would be exported and be available to all parts of the program. The static declaration also allows other classes to use the same name internally.

|  |
| --- |
| static reduces the visibility of a function an should be used wherever possible to control access to functions! |

**Analysis**

|  |  |
| --- | --- |
| http://www.cs.auckland.ac.nz/%7Ejmor159/PLDS210/fig/bsearch.gif | Each step of the algorithm divides the block of items being searched in half. We can divide a set of ***n*** items in half at most **log2 *n*** times.  Thus the running time of a binary search is proportional to **log *n*** and we say this is a ***O(*log *n)*** algorithm. |

|  |  |
| --- | --- |
| Binary search requires a more complex program than our original search and thus for *small* ***n*** it may run slower than the simple linear search. However, for large ***n***,  http://www.cs.auckland.ac.nz/%7Ejmor159/PLDS210/fig/limit.gif  Thus at large ***n***, **log *n*** is *much* smaller than ***n***, consequently an ***O(*log *n)*** algorithm is *much* faster than an ***O(n)*** one. | http://www.cs.auckland.ac.nz/%7Ejmor159/PLDS210/fig/log_graph.gif  Plot of ***n*** and **log *n*** *vs* ***n*** . |

We will examine this behaviour more formally in a [later section](http://www.cs.auckland.ac.nz/%7Ejmor159/PLDS210/complexity.html). First, let's see what we can do about the insertion (AddToCollection) operation.

In the worst case, insertion may require ***n*** operations to insert into a sorted list.

1. We can find the place in the list where the new item belongs using binary search in ***O(*log *n)*** operations.
2. However, we have to shuffle all the following items up one place to make way for the new one. In the worst case, the new item is the first in the list, requiring ***n*** move operations for the shuffle!

A similar analysis will show that deletion is also an ***O(n)*** operation.

If our collection is static, *ie* it doesn't change very often - if at all - then we may not be concerned with the time required to change its contents: we may be prepared for the initial build of the collection and the occasional insertion and deletion to take some time. In return, we will be able to use a simple data structure (an array) which has little memory overhead.

However, if our collection is large and dynamic, *ie* items are being added and deleted continually, then we can obtain considerably better performance using a data structure called a [tree](http://www.cs.auckland.ac.nz/%7Ejmor159/PLDS210/trees.html).

|  |
| --- |
| **Key terms** |

**Big Oh**

A notation formally describing the set of all functions which are bounded above by a nominated function.

**Binary Search**

A technique for searching an ordered list in which we first check the middle item and - based on that comparison - "discard" half the data. The same procedure is then applied to the remaining half until a match is found or there are no more items left.

|  |  |
| --- | --- |
| Continue on to [Trees](http://www.cs.auckland.ac.nz/%7Ejmor159/PLDS210/trees.html) | Back to the [Table of Contents](http://www.cs.auckland.ac.nz/%7Ejmor159/PLDS210/ds_ToC.html) |

© [John Morris](mailto:morris@ee.uwa.edu.au), 1998