Linear Search, Binary Search and other Searching Techniques

By Prelude

Searching for data is one of the fundamental fields of computing. Often, the difference between a fast program and a slow one is the use of a good algorithm for the data set. This article will focus on searching for data stored in a linear data structure such as an array or linked list. Naturally, the use of a hash table or binary search tree will result in more efficient searching, but more often than not an array or linked list will be used. It is necessary to understand good ways of searching data structures not designed to support efficient search.

Linear Search

The most obvious algorithm is to start at the beginning and walk to the end, testing for a match at each item:

```
bool jw_search ( int *list, int size, int key, int*& rec )
{
  // Basic sequential search
  bool found = false;
  int i;

  for ( i = 0; i < size; i++ ) {
    if ( key == list[i] )
      break;
  }
  if ( i < size ) {
    found = true;
    rec = &list[i];
  }

  return found;
}
```

This algorithm has the benefit of simplicity; it is difficult to get wrong, unlike other more sophisticated solutions. The above code follows the convention of this article, they are as follows:

1. All search routines return a true/false boolean value for success or failure.
2. The list will be either an array of integers or a linked list of integers with a key.
3. The found item will be saved in a reference to a pointer for use in client code.

The algorithm itself is simple. A familiar 0 - n-1 loop to walk over every item in the array, with a test to see if the current item in the list matches the search key. The loop can terminate in one of two ways. If i reaches the end of the list, the loop condition fails. If the current item in the list matches the key, the loop is terminated early with a break statement. Then the algorithm tests the index variable to see if it is less that size (thus the loop was terminated early and the item was found), or not (and the item was not found).

For a linked list defined as:

```
struct node {
  int rec;
  int key;
  node *next;

  node ( int r, int k, node *n )
    : rec ( r )
    , key ( k )
    , next ( n )
  {}
```

```
};
```

The algorithm is equally simple:

```
bool jw_search ( node*& list, int key, int*& rec )
{
  // Basic sequential search
  bool found = false;
  node *i;

  for ( i = list; i != 0; i = i->next ) {
    if ( key == i->key )
      break;
  }
  if ( i != 0 ) {
    found = true;
    rec = &i->rec;
  }

  return found;
}
```

Instead of a counting loop, we use an idiom for walking a linked list. The idiom should be familiar to most readers. For those that are not familiar with it, that is how it is done. :-) The loop terminates if i is a null pointer (the algorithm assumes a null pointer terminates the list) or if the item was found.

The basic sequential search algorithm can be improved in a number of ways. One of those ways is to assume that the item being searched for will always be in the list. This way you can avoid the two termination conditions in the loop in favor of only one. Of course, that creates the problem of a failed search. If we assume that the item will always be found, how can we test for failure?

The answer is to use a list that is larger in size than the number of items by one. A list with ten items would be allocated a size of eleven for use by the algorithm. The concept is much like C-style strings and the nul terminator. The nul character has no practical use except as a dummy item delimiting the end of the string. When the algorithm starts, we can simply place the search key in list[size] to ensure that it will always be found:

```
bool jw_search ( int *list, int size, int key, int*& rec )
{
  // Quick sequential search
  bool found = false;
  int i;

  list[size] = key;
  for ( i = 0; key != list[i]; i++ )
    ;
  if ( i < size ) {
    found = true;
    rec = &list[i];
  }

  return found;
}
```

Notice that the only test in the traversal loop is testing for a match. We know that the item is in the list somewhere, so there's no need for a loop body. After the loop the algorithm simply tests if i is less than size. If it is then we have found a real match, otherwise i is equal to size. Because list[size] is where the dummy item was, we can safely say that the item does not exist anywhere else in the list. This algorithm is faster because it reduces two tests in the loop to one test. It isn't a big improvement, but if jw_search is called often on large lists, the optimization may become noticeable.

Another variation of sequential search assumes that the list is ordered (in ascending sorted order for the algorithm we will use):

```
bool jw_search ( int *list, int size, int key, int*& rec )
{
  // Ordered sequential search
  bool found = false;
  int i;

  for ( i = 0; i < size && key > list[i]; i++ )
```

```
    ;
  if ( key == list[i] ) {
    found = true;
    rec = &list[i];
  }

  return found;
}
```

The performance for a successful search where all keys are equally likely is the same as the basic algorithm. The speed improvement is for failed searches. Because the absence of an item can be determined more quickly, the average speed of a failed search is twice that of previous algorithms on average. By combining the Quick sequential search and the Ordered sequential search, one can have a highly tuned sequential search algorithm.
Exercise 1: The last paragraph suggests an efficient sequential search algorithm. Use what you've learned to implement it.

Exercise 2: Write a test program to verify the correct operation of the functions given.

Exercise 3: Can you think of a more efficient way to perform sequential search? What about a non-sequential search?

Self Organizing Search

For lists that do not have a set order requirement, a self organizing algorithm may be more efficient if some items in the list are searched for more frequently than others. By bubbling a found item toward the front of the list, future searches for that item will be executed more quickly. This speed improvement takes advantage of the fact that 80% of all operations are performed on 20% of the items in a data set. If those items are nearer to the front of the list then search will be sped up considerably.

The first solution that comes to mind is to move the found item to the front. With an array this would result in rather expensive memory shifting:

```
bool jw_search ( int *list, int size, int key, int*& rec )
{
  // Self-organizing (move to front) search
  bool found = false;

  // Is it already at the front?
  if ( key == list[0] ) {
    rec = &list[0];
    found = true;
  }
  else {
    int i;
    for ( i = 1; i < size; i++ ) {
      if ( key == list[i] )
        break;
    }
    if ( i < size ) {
      int save = list[i];
      // Fill the hole left by list[i]
      for ( int j = i; j < size - 1; j++ )
        list[j] = list[j + 1];
      // Make room at the front
      for ( int j = size - 1; j > 0; j-- )
        list[j] = list[j - 1];
      list[0] = save;
      rec = &list[0];
      found = true;
    }
  }

  return found;
}
```

Filling the hole left by removing the found item and then shifting the entire contents of the array to make room at the front is dreadfully expensive and probably would make this algorithm impractical for arrays. However, with a linked list the splicing operation required to restructure the list and send the item to the front is quick and trivial:

```
bool jw_search ( node*& list, int key, int*& rec )
```

```
{
  // Self-organizing (move to front) search
  node *iter = list;
  bool found = false;

  // Is it already at the front?
  if ( key == iter->key ) {
    rec = &iter->rec;
    found = true;
  }
  else {
    for ( ; iter->next != 0; iter = iter->next ) {
      if ( key == iter->next->key )
        break;
    }
    // Was the item found?
    if ( iter->next != 0 ) {
      // Remove the node and fix the list
      node *save = iter->next;
      iter->next = save->next;
      // Place the node at the front
      save->next = list;
      list = save;
      rec = &list->rec;
      found = true;
    }
  }

  return found;
}
```

For a linked data structure, moving an item to a new position over large distances has a constant time complexity, O(1), whereas for contiguous memory such as an array, the time complexity is O(N) where N is the range of items being shifted. A solution that is just as effective, but takes longer to reach the optimal limit is to swap the found item with the previous item in the list. This algorithm is where arrays excel over linked lists for our data set of integers. The cost of swapping two integers is less than that of surgery with pointers. The code is simple as well:

```
bool jw_search ( int *list, int size, int key, int*& rec )
{
  // Self-organizing (swap with previous) search
  bool found = false;
  int i;

  for ( i = 0; i < size; i++ ) {
    if ( key == list[i] )
      break;
  }
  // Was it found?
  if ( i < size ) {
    // Is it already the first?
    if ( i > 0 ) {
      int save = list[i - 1];
      list[i - 1] = list[i];
      list[i--] = save;
    }
    found = true;
    rec = &list[i];
  }

  return found;
}
```

Exercise 4: Does the 80-20 rule really suggest an improvement over previous sequential search algorithms? If so, at what point does the performance increase make the cost of restructuring the list practical?

Exercise 5: Tune source 6 to run as quickly as possible. Would it make sense to use in production code?

Exercise 6: Rewrite source 8 to use linked lists. Compare and contrast the two functions.

Exercise 7: Consider shifting the item over smaller distances, 1/2 or 1/3 of the distance to the front for example. Is such a

change worth the effort? Is it an improvement in any way?

Binary Search

All of the sequential search algorithms have the same problem; they walk over the entire list. Some of our improvements work to minimize the cost of traversing the whole data set, but those improvements only cover up what is really a problem with the algorithm. By thinking of the data in a different way, we can make speed improvements that are much better than anything sequential search can guarantee.

Consider a list in ascending sorted order. It would work to search from the beginning until an item is found or the end is reached, but it makes more sense to remove as much of the working data set as possible so that the item is found more quickly. If we started at the middle of the list we could determine which half the item is in (because the list is sorted). This effectively divides the working range in half with a single test. By repeating the procedure, the result is a highly efficient search algorithm called binary search.

The actual algorithm is surprisingly tricky to implement considering the apparent simplicity of the concept. Here is a correct function that implements binary search by marking the current lower and upper bounds for the working range:

```
bool jw_search ( int *list, int size, int key, int*& rec )
{
  // Binary search
  bool found = false;
  int low = 0, high = size - 1;

  while ( high >= low ) {
    int mid = ( low + high ) / 2;
    if ( key < list[mid] )
      high = mid - 1;
    else if ( key > list[mid] )
      low = mid + 1;
    else {
      found = true;
      rec = &list[mid];
      break;
    }
  }

  return found;
}
```

No explanation will be given for this code. Readers are expected to trace its execution on paper and with a test program to fully understand its elegance. Binary search is very efficient, but it can be improved by writing a variation that searches more like humans do. Consider how you would search for a name in the phonebook. I know of nobody who would start in the middle if they are searching for a name that begins with B. They would begin at the most likely location and then use that location as a gauge for the next most likely location. Such a search is called interpolation search because it estimates the position of the item being searched for based on the upper and lower bounds of the range. The algorithm itself isn't terribly difficult, but it does seem that way with the range calculation:

```
bool jw_search ( int *list, int size, int key, int*& rec )
{
  // Interpolation search
  bool found = false;
  int low = 0, high = size - 1;

  while ( list[high] >= key && key > list[low] ) {
    double low_diff = (double)key - list[low];
    double range_diff = (double)list[high] - list[low];
    double count_diff = (double)high - low;
    int range = (int)( low_diff / range_diff * count_diff + low );
    if ( key > list[range] )
      low = range + 1;
    else if ( key < list[range] )
      high = range - 1;
    else
      low = range;
  }
  if ( key == list[low] ) {
    found = true;
    rec = &list[low];
  }
```

```
    return found;
}
```

Once again, no explanation will be given. If you made it this far you should be writing test code and doing paper runs of any new algorithm you meet as habit. If not, now you have no choice because I am not going to say just how this function works. Its magic. :-)

Interpolation search is theoretically superior to binary search. With an average time complexity of O(log log n), interpolation search beats binary search's O(log n) easily. However, tests have shown that interpolation search isn't significantly better in practice unless the data set is very large. Otherwise, binary search is faster.

Exercise 8: Explain why binary search is so tricky to implement. Show an example of an incorrect implementation.

Exercise 9: Binary search uses a divide-and-conquer algorithm. What other problems can this algorithm solve?

Exercise 10: Determine the lower limit where interpolation search becomes substantially better than binary search.

Exercise 11: Write a search function that uses interpolation search until the lower limit is reached, then switches to binary search. Is the extra performance worth the effort?

Conclusion

Searching is an important function in computer science. Many advanced algorithms and data structures have been devised for the sole purpose of making searches more efficient. And as the data sets become larger and larger, good search algorithms will become more important. At one point in the history of computing, sequential search was sufficient. But that quickly changed as the value of computers became apparent.

Linear search has many interesting properties in its own right, but is also a basis for all other search algorithms. Learning how it works is critical.

Binary search is the next logical step in searching. By dividing the working data set in half with each comparison, logarithmic performance, O(log n), is achieved. That performance can be improved significantly when the data set is very large by using interpolation search, and improved yet again by using binary search when the data set gets smaller.

Related articles

Adventures in debugging binary search

Learn more about sorting algorithms

Learn more about algorithmic efficiency and big-O notation