Searching Techniques

Muhaimin Bin Munir

Searching

The process of finding a particular element in an array is called searching. There two popular searching techniques -

- Linear search
- Binary search

If the search key is a member of the array, typically the location of the search key is reported to indicate the presence of the search key in the array. Otherwise, a *sentinel* value is reported to indicate the absence of the search key in the array.

Linear Search

Each member of the array is visited until the search key is found.

Example

Write a program to search for the search key entered by the user in the following array:

{9, 4, 5, 1, 7, 78, 22, 15, 96, 45}

You can use the linear search in this example.

Binary Search

Given a sorted array, **Binary Search** algorithm can be used to perform fast searching of a search key on the sorted array.

```
int BinarySearch (int A[], int skey) {
        int low = 0, high = SIZE − 1, middle;
        while ( low <= high) {
                middle = (low + high) / 2;
                if ( skey == A[middle] )
                        return middle;
                else if ( skey < A[middle] )
                        high = middle − 1;
                else
                        low = middle + 1;
        }
        return −1;
}
```

Simulation

https://www.cs.usfca.edu/~galles/visualization/Search.html

Binary Search Variants

A[] = {2, 3, 3, 5, 5, 5, 6, 6, 9, 9, 9, 9} //0 based indexing

- **search(key)**: returns any position of the key if found. otherwise returns -1.
- **firstOccurance(key)**: returns first occurance of the key if found. otherwise returns -1.
- **lastOccurance(key)**: returns last occurance of the key if found. otherwise returns -1.
- **leastGreater(key)**: returns the first position of the least element greater than the key.
- **greatestLesser(key)**: returns the last position of the greatest element lesser than the key.
- **searchNear(key)**: returns the position of the key if found otherwise returns nearest position from the key.

function_call	return_value
search(5)	3/4/5
firstOccurance(9)	8
lastOccurance(6)	7
leastGreater(5)	6
leastGreater(9)	12
greatestLesser(2)	-1
greatestLesser(9)	7
searchNear(7)	7

Computational Complexity

- The Computational Complexity of the **Binary Search** algorithm is measured by the maximum (worst case) number of comparisons it performs for searching operations.
- The searched array is divided by 2 for each comparison/iteration. Therefore, the maximum number of comparisons is measured by: $\mathbf{log_2(n)}$, where **n** is the size of the array.

Example:
If a given sorted array 1024 elements, then the maximum number of comparisons required is:

$\log_2(1024) = 10$ (only 10 comparisons is enough)

Computational Complexity

- Note that the Computational Complexity of the **Linear Search** is the maximum number of comparisons you need to search the array. As you are visiting all the array elements in the worst case, then, the number of comparisons required is:

  n (n is the size of the array)

  Example:
  If a given an array of 1024 elements, then the maximum number of comparisons required is:

  n = 1024 (As many as 1024 comparisons may be required)