Data Structures

Array Searching

Array Operations

- **Insertion**
  - Operation of adding another element to an array
  - How many steps in terms of n (number of elements in array)?
    - At the end
    - In the middle
    - In the beginning
  - n steps at maximum (move items to insert at given location)
- **Deletion**
  - Operation of removing one of the elements from an array
  - How many steps in terms of n (number of elements in array)?
    - At the end
    - In the middle
    - In the beginning
  - n steps at maximum (move items back to take place of deleted item)

Array Operations: Search Algorithms

- Operation of locating a specific data item in an array
  - Successful: If location of the searched data is found
  - Unsuccessful: Otherwise

- Complexity (or efficiency) of a search algorithm
  - Number of comparisons $f(n)$ required to locate data within array
  - n is the number of elements within array

- Two algorithms for searching in arrays
  - Linear search (or sequential search)
  - Binary search

Linear Search

- Very intuitive and simple algorithm

Algorithm works as follows:

- Starts from the first element of the array
- Uses a loop to sequentially step through an array
- Compares each element with the data item being searched
- Stops when data item is found or end of array is reached

Linear Search Algorithm

```
// numElems - maximum number of elements in the array
// value    - integer data (item) to be searched
// position - array subscript that holds value (if success)
//            -1 if value not found

int searchList(int list[], int numElems, int value)
{
  int index = 0;      // Used as a subscript to search array
  int position = -1;  // To record position of search value
  bool found = false; // Flag to indicate if the value was found
  while (index < numElelments && !found)
  {
        if (list[index] == value) {
           found = true;
           position = index;
        }
        index++;
   }
   return position;
}
```

Calling Function searchList

```
#include <iostream.h>

// Function prototype
int searchList(int [], int, int);
const int arrSize = 5;

void main(void)
{
    int tests[arrSize] = {87, 75, 98, 100, 82};
    int result;
    result = searchList(tests, arrSize, 100);
    if (result == -1)
        cout << "You did not earn 100 points on any test\n";
    else{
        cout << "You earned 100 points on test ";
        cout << (result + 1) << endl;
    }
}
```

Program Output:
You earned 100 points on test 4.

Discussion

- Advantage of linear search is its simplicity
  - Easy to understand
  - Easy to implement
  - Does not require array to be in order (i.e., sorted)

- Disadvantage is its efficiency (or complexity)
  - Worst case complexity: `f(n) = n+1`
    - ➢ Number of steps are proportional to number n of elements in an array

  - If there are 20,000 items in an array
    - ➢ Searched data item is stored in the 19,999th element
    - ➢ Entire array has to be searched

Binary Search

- Binary search is more efficient than linear search
  - Requires array to be in sorted order (i.e., ascending order)

Algorithm works as follows:

- Starts searching from the middle element of an array

- If value of data item is less than the value of middle element
  - Algorithm starts over searching the first half of the array

- If value of data item is greater than the value of middle element
  - Algorithm starts over searching the second half of the array

- Algorithm continues halving the array until data item is found

Binary Search Algorithm

```
// numElems – maximum number of elements in the array
// value    – integer data (item) to be searched
// position – array subscript that holds value (if success)
//            -1 if value not found
int binarySearch(int array[], int numelems, int value)
{
   int first = 0, last = numelems - 1, middle, position = -
   1;
   bool found = false;
   while (!found && first <= last){
      middle = (first + last) / 2;            // Calculate mid point
      if (array[middle] == value) {           // If value is found at mid
         found = true;
         position = middle;
      }
      else if (array[middle] > value)         //If value is in lower half
          last = middle - 1;
      else
          first = middle + 1;                 // If value is in upper half
   }
   return position;
}
```

Binary Search Example

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
list	4	8	19	25	34	39	45	48	66	75	89	95

Sorted list for binary search

key = 89

Iteration	first	last	mid	list[mid]	
1	0	11	5	39	
2	6	11	8	66	
3	9	11	10	89	Value is found

key = 34

Iteration	first	last	mid	list[mid]	
1	0	11	5	39	
2	0	4	2	19	
3	3	4	3	25	
4	4	4	4	34	Value is found

Calling Function binarySearch

```
#include <iostream.h>
// Function prototype
int binarySearch(int [], int, i
const int arrSize = 20;

void main(void)
{
    int empIDs[arrSize] = {101, 142, 147, 189, 199, 207, 222, 234, 289, 296,
                           310, 319, 388, 394, 417, 429, 447, 521, 536, 600};
    int result, empID;
    cout << "Enter the Employee ID you wish to search for: ";
    cin >> empID;
    result = binarySearch(empIDs, arrSize, empID);
    if (result == -1)
        cout << "That number does not exist in the array.\n";
    else {
        cout << "That ID is found at element " << result;
        cout << " in the array\n";
    }
}
```

Program Output:

Enter the Employee ID you wish to search for: 199
That ID is found at element 4 in the array

Efficiency Of Binary Search

- Much more efficient than the linear search
- How long does this take (worst case)?
  - If the list has 8 elements
    - It takes 3 steps ($2^3 = 8$)
  - If the list has 16 elements
    - It takes 4 steps ($2^4 = 16$)
  - If the list has 64 elements
    - It takes 6 steps ($2^6 = 64$)
- Worst case complexity: $f(n) = \log_2(n)$
  - Takes $\log_2 n$ steps

Any Question So Far?