Sorting and Searching I

DM2233 Advanced Data Structures & Algorithms

Module Schedule

Week	Lecture	Remarks
1	Overloading and Templates I	
2	Overloading and Templates II	Labour Day (Fri) – Lab 2 Make up on 27-Apr
3	Overloading and Templates III	
4	Overloading and Templates IV	
5	Exception Handling I	
6	Exception Handling II	
7	Standard Template Library / Assignment 1	Vesak Day (Mon)
Week 8 and 9: Mid-Sem Break		
10	Sorting and Searching I	
11	Sorting and Searching II	
12	Sorting and Searching III	
13	Binary Tree I	Hari Raya Puasa (Fri)
14	Lab Test	
15	Binary Tree II	
16	Binary Tree III	SG50 Day (Fri)
17	Preprocessing / Assignment 2	National Day (Mon)

Objective

- Introduction to Search Algorithms
- Types of Search Algorithms
  - Linear Search Algorithm
    - Sequential Search Analysis
  - Binary Search Algorithm
  - Comparison

Introduction to Search Algorithms

- A search is required, if we want to locate a particular piece of information
- When a search is performed, the item (target) should have a unique value associated with it in order for the search to be successful
- The unique value is what we call the key of the item

Introduction to Search Algorithms

- The keys of the items in the chunk of data are used in various software operations such as searching, sorting, insertion, and deletion
- A search usually compares these keys against the key to be searched
- If a key matches, the search is successful. Otherwise, the search is unsuccessful

Types of Search Algorithms

- Linear Search Algorithm (Sequential Search)

- Binary Search Algorithm (Binary Search)

Linear Search Algorithm

- Usually applies to an array or linked list
- Start from the front or end of the array or list
- Compare the array or node key against the key to be search
- If not found, advance to next adjacent array component or node and repeat previous step

Linear Search Example

```
int data[] = { 1, 3, 4, 7, 2, 9, 5, 8 };
int key_to_find = 7;

for(int i = 0; i < sizeof(data)/4; i++)
{
     if(data[i] == key_to_find)
     {
          cout << "Found it!" << endl;
          return 0;
     }
}

return -1;
```

Observations

- See that the search starts from first array component (i.e. i = 0, data[0])
- Search ends when the item is found. A 0 is returned
- If not found, array index i would run out of bounds. A -1 is returned
- Array items are not in any particular order

Observations

- cout << "Found it!" << endl; return 0; executes once if item is found

- return -1; executes once if item is not found

Sequential Search Analysis

- Not really interested in statements that executes only once since the computer processing time is small. Hence, the analysis is on:

```
for(int i = 0; i < sizeof(data)/4; i++)
{
   if(data[i] == key_to_find)
   {
         . . .
   }
}
```

Sequential Search Analysis

- For each iteration of the loop, the search item is compared with an element in the list
- Clearly, the loop terminates once the search item is found
- Therefore, the key comparisons have most impact on the loop statement
- In analysis, we count the number of key comparisons because this number gives us the most useful information

Sequential Search Analysis

- Suppose L is list of length n. We want to determine the number of key comparisons made by sequential search when L is searched for a given item
- If item is not in L, we have done n key comparisons
- If item is somewhere in L, the number of key comparisons depends on where in the list L, the search item is located

Sequential Search Analysis

- Worst Case: n key comparisons
- Best Case: 1 key comparison
- The above 2 cases are not likely to occur every time we apply a sequential search
- The more meaningful would be the average behaviour, that is the average number of key comparisons

Sequential Search Analysis

- To determine the average number of comparisons in a successful case
  - Consider all possible cases
  - Find the number of comparisons in each case
  - Total up the number of comparisons and divide by the number of cases

Sequential Search Analysis

- If target (search item) is the first element in L, one key comparison is needed

- If target is the second element in L, two key comparisons is needed

- Suppose there are n elements in the list L, (1 + 2 + 3 + ... + n) key comparisons are done

- Average key comparisons:
  - (1 + 2 + 3 + ... + n) / n

Sequential Search Analysis

- It is known that
  - 1 + 2 + 3 + ... + n = $\frac{n(n+1)}{2}$

- Average key comparisons:
  - (1 + 2 + 3 + ... + n) / n
  - = $\frac{n+1}{2}$

Sequential Search Analysis

- Average Case:
  - $\frac{n+1}{2}$
- Best Case:
  - 1
- Worst Case:
  - n
- Even if the list is ordered, the above cases stand

Types of Search Algorithms

- Linear Search Algorithm (Sequential Search)
- Binary Search Algorithm (Binary Search)

Binary Search Algorithm

- For very large list, sequential search, on average, searches half the list
- Not very efficient
- The binary search is very fast, however, it can only operate on ordered list

Binary Search Algorithm

- Uses divide-and-conquer technique to search the list
- First the search item is compared with the middle element of the list
- If search item is found, the search terminates.
- If search item is less than the middle element, restrict the search to the first half of the list; otherwise, that means search the second half of the list

Binary Search Algorithm

```
int first = 0;
int last = length - 1;
int mid;

while(first <= last)
{
      mid = (first + last) / 2; //calculate mid of list
      if(list[mid] == key_to_find)
            return mid;
      else if(list[mid] > item)
            last = mid - 1; //search first half
      else
            first = mid + 1; //search second half
}

return -1;
```

Binary Search Example

- int data[] = { 1, 3, 9, 12, 16, 23, 35, 42, 55, 59, 67, 99 };

- int key_to_find = 59;

0	1	2	3	4	5	6	7	8	9	10	11
1	3	9	12	16	23	35	42	55	59	67	99

Iteration	first	last	mid	data[mid]	Number of comparisons
1	0	11	5	23	2

Binary Search Example

- int key_to_find = 59;

0	1	2	3	4	5	6	7	8	9	10	11
1	3	9	12	16	23	35	42	55	59	67	99

Iteration	first	last	mid	data[mid]	Number of comparisons
1	0	11	5	23	2
2	6	11	8	55	2

Binary Search Example

- int key_to_find = 59;

0	1	2	3	4	5	6	7	8	9	10	11
1	3	9	12	16	23	35	42	55	59	67	99

Iteration	first	last	mid	data[mid]	Number of comparisons
1	0	11	5	23	2
2	6	11	8	55	2
3	9	11	10	67	2

Binary Search Example

- int key_to_find = 59;

0	1	2	3	4	5	6	7	8	9	10	11
1	3	9	12	16	23	35	42	55	59	67	99

Iteration	first	last	mid	data[mid]	Number of comparisons
1	0	11	5	23	2
2	6	11	8	55	2
3	9	11	10	67	2
4	9	9	9	59	1 (item is found)

Observations

- See that the search starts from the mid of the array
- Depending on the mid component, the search later focuses on lower or upper half of the array

Binary Search Analysis

- Best Case:
  - 1

- Worst Case:
  - Suppose L is a sorted list of size n
  - Each iteration reduces the size of n by half
    - n, $\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \ldots 1$
    - So assume we are so unlucky that the number we want to find does not exists in the array
    - The iterations it takes from n to 1 is $\log_2 n$

Binary Search Analysis

- Average Case:
  - Its still a study
  - Also depends on the data distribution
  - But we can safely assume that it’s also $\log_2 n$

Comparison

- Sequential Search Average Case:
  - $\frac{n+1}{2}$
- Sequential Search Worst Case:
  - n
- Binary Search Average Case:
  - $\log_2 n$
- Binary Search Worst Case:
  - $\log_2 n$
- It can be seen that Binary Search is definitely faster

Summary

- Understand Linear Search Algorithm requirements
- Discussed about linear search algorithm
- Understand Binary Search Algorithm and how to implement it
- Understand how to analyze a algorithm and conclude its efficiency