BITS Pilani
K K Birla Goa Campus

innovate achieve lead

8. Searching

Dr. Swati Agarwal

Agenda

1. Searching
   - Explicit Searching: Requirements
2. Problem Statement
3. Searching Algorithms
   - Linear Search (unsorted array)
   - Sorted/Ordered Linear Search
   - Binary Search
   - Hashing
4. Hash Tables
5. Collisions

Searching

- An important and recurring problem in Computing: *locating information*

- Formally known as *Searching*
  - **Implicit Search:** Collection of data is implicit. For example, find the square root of a given number. You have to perform an exhaustive search on a list of numbers to complete the task
  - **Explicit Search:** Data is explicitly given and search needs to be performed only on that data. For Example: given a record of DSA students find whether there is any student belonging to EEE stream. (focus of the module)

Explicit Searching: Requirements

- Clearly, if an element needs to be searched, it needs to be **represented** first. (need of data structure)
- Once that information is represented, we need a method to **process** (here, processing of interest is that of Searching) it (need of an algorithm)

Explicit Searching: Dummy Example

- Representation: simplest form of data structure: Arrays[1]

Let us assume an integer array $A[10] =$

2	10	5	15	30	7	9	14	16	35

$Search(5) = 2$
$Search(9) = 7$
$Search(35) = 9$
$Search(55) =$ nowhere.

Represent *nowhere* as -1 since we are using simple array data structure.

[1] more complex data structures can also be used. Later modules

Problem Statement

Given an array $A[]$ and an integer x, find an integer i such that

$$i = \begin{cases} -1 & \text{if } \nexists j, A[j] == x \\ j & \text{if } \forall j, A[j] == x \end{cases}$$

- If there is more than one position where x occurs then the algorithm can return any of them.
- For example: if $A[] = [15, 20, 25, 15]$, then for $x = 15$, it can return 0 as well as 3.
- One algorithm may return smallest position at which x occurs and another algorithm may return largest value.

Searching Algorithms

1. Linear Search (unsorted array)
2. Linear Search (sorted array)
3. Binary Search
4. Hashing

1. Unordered Linear Search

- Bruteforce Search method aka British Museum Algorithm
- Numbers in the array can be arranged in any arbitrary order.
- Exhaustive enumeration- go through each element until we find the information (x).

1. Linear Search (unsorted array)

```
function UnOrderedLinearSearch(intA[], n, x)
    for {i=0; i < n; i++}
        if A[i] == x then
            return i
    return −1
```

	Best Case	Average Case	Worst Case
Linear Search	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

2. Sorted/Ordered Linear Search

- The elements in the array are sorted (increasing order)
- In many cases, we do not need to scan the complete array
- At any point of time if the value of $A[i] > x$ then all remaining items are going to be greater as well

2. Sorted/Ordered Linear Search

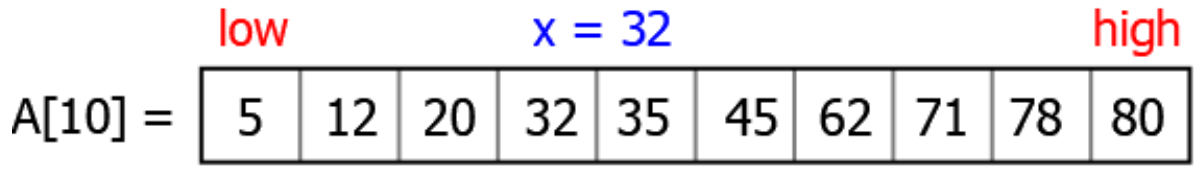
```
function OrderedLinearSearch(intA[], n, x)
    for {i=0; i < n; i++}
        if A[i] == x then
            return i
        else if A[i] > x then
            return −1
    return −1
```

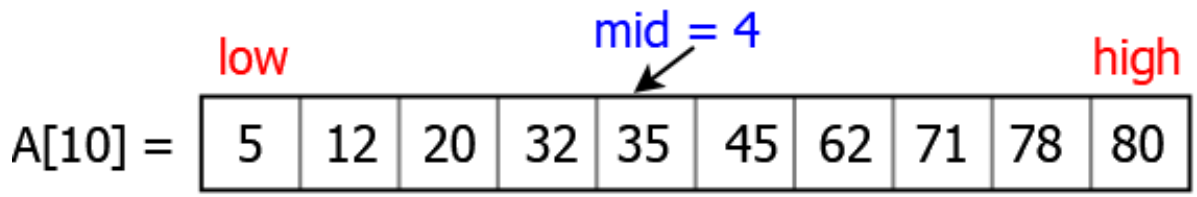
We can make modifications in the algorithm and reduce the complexity by incrementing the index at a faster rate.

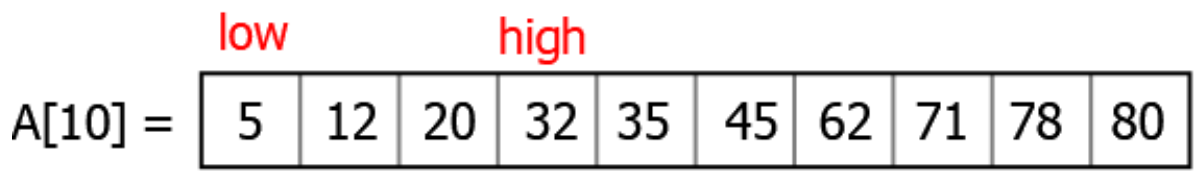
3. Binary Search

- aka Bi-section search
- data is always sorted
- Similar to a dictionary search in real-world
- Search for the element/word in the middle of the collection (array/word dictionary).
- If the element if found. Bingo!
- If the element is lesser than the middle value then apply the same process to first half. Otherwise apply the same process to the second half.
- sounds like we can solve this using recurrence equation as well.

3. Binary Search: Working Example







3. Binary Search Iterative

function BINARYSEARCHITERATIVE($intA[]$, n, x)
    $low = 0, high = n - 1$
    while $low \leq high$
        $mid \leftarrow \lfloor low + (high - low)/2 \rfloor$
        if $A[mid] == x$ **then**
            return mid
        else if $A[mid] < x$ **then**
            $low \leftarrow mid + 1$
        else
            $high \leftarrow mid - 1$
    return -1

	Best Case	**Average Case**	**Worst Case**
Linear Search	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

3. Binary Search Recursive

function BSRecursive(*intA*[], *x*, *low*, *high*)
    $mid \leftarrow \lfloor low + (high - low)/2 \rfloor$
    if $A[mid] == x$ **then**
        return *mid*
    else if $A[mid] < x$ **then**
        return BSRecursive($A[]$, *x*, $mid + 1$, *high*)
    else
        return BSRecursive($A[]$, *x*, *low*, $mid - 1$)
    return -1

4. Hashing- Dictionary Abstract Data Type[2]

- Store items (k,e)
- Search(k), insert (k,e), remove(k)
- size(), isEmpty()
- Ordered Dictionary ADT: maximum(), minimum(), successor(k), predecessor(k)

[2]Hashing slides belong to Dr. A Baskar.

Naive implementation: Log file

- Store items in an array or list
- insert method takes O(1)
- search and remove will take O(n)
- Space requirement is $\Theta(n)$

Look-up Table

- Store items in an array but with an ordering
- Binary search takes $\mathcal{O}(logn)$ time
- But insertion and deletion will take O(n) time
- Look-up table is good for finding minimum, maximum, predecessor and successor
- Can we improve the performance for insertion and deletion?

Hash Tables

- Hash tables are made up of two functions namely hash code and compression map
- Hash code is a function which converts keys to integers
- Compression map is a function which maps integers into the range $[0, N-1]$
- In this lecture we focus on compression maps and refer them as hash functions h

Hash function

- Uses an array of size proportional to the number of keys used
- Instead of using key as an array index, array index is computed using hash function h from the key k
- We use $h(k)$ to denote the hash value of k

Division Method

- $h(k) = k \bmod m$ where m is the size of the table
- Certain values of m may not be good
- Good values of m are prime numbers which are not close to exact powers of 2

Collisions

- Two keys k_1, k_2 may hash into the same slot, i.e., $h(k_1) = h(k_2)$
- Ideally we should try to avoid collision altogether
- But it is not possible when the potential key space is large
- We should find effective techniques to resolve collisions

Collision Resolution: Chaining

- In this method, we put all elements that hash to same slot in a linked list
- Slot j contains a pointer to the head of the list of all stored elements that hash to j
- If there are no such elements, slot j contains NIL

Performance Analysis: Chaining Method

- In the worst case, all the keys are hashed to same slot
- So searching for an element will take $\mathcal{O}(n)$ time
- In the best case, it will take constant time to search
- What about average case?

Average Case Analysis: Chaining Method

- We define load factor α as the average number of keys per slot
- If n is the number of keys to be stored and m be the number of slots, then $\alpha = n/m$
- Assumption 1: $\mathcal{O}(1)$ time to compute $h(k)$
- Assumption 2 (Simple Uniform Hashing): Any key is equally likely hash into any of the m slots, independent of where any other keys hashes to

Expected time: Chaining Method

- Successful search takes expected time $\Theta(1 + \alpha)$
- Unsuccessful search takes expected time $\Theta(1 + \alpha)$
- If n is $\mathcal{O}(m)$, then α is $\mathcal{O}(m)/m$, which is $\mathcal{O}(1)$
- Searching takes constant time on average
- Indeed all dictionary operations take $\mathcal{O}(1)$ time on average

Open Addressing

- Store all keys in the hash table itself
- Each slot contains either a key or NIL
- To search for key k:
  - Examine slot h(k). Examining a slot is known as a probe
  - If slot $h(k)$ contains key k, the search is successful
  - If the slot contains NIL, the search is unsuccessful
  - There's a third possibility: slot $h(k)$ contains a key that is not k
  - Compute the index of some other slot, based on k and which probe we are on
  - Keep probing until we either find key k or we find a slot holding NIL

Linear Probing

- h(k,i)=h'(k)+i mod m
- h'(k) is an auxiliary hash function
- Inserting and searching is straightforward if there are no deletions

Inserting an element

```
Hash-Insert(T, k)
1.  i <- 0
2.  repeat  j <- h(k, i)
3.              if T[j] = NIL
4.                  then T[j] <- k
5.                       return j
6.                  else i <-i + 1
7.  until i = m
8.  error hash table overflow
```

Searching an element

```
Hash-Search (T, k)
1.  i <- 0
2.  repeat  j <- h(k, i)
3.              if T[j] = k
4.                  then return j
5.              i <- i + 1
6.  until T[j] = NIL or i = m
7.  return NIL
```

Deleting an element

- Cannot just turn the slot containing the key we want to delete to contain NIL
- Doing so might make it impossible to retrieve any key k during whose insertion we had probed this slot and found it occupied
- Use a special value DELETED instead of NIL when marking a slot as empty during deletion
- Search should treat DELETED as though the slot holds a key that does not match the one being searched for
- Insert should treat DELETED as though the slot were empty, so that it can be reused

Probe Sequence

- Sequence of slots examined during a key search constitutes a probe sequence
- Probe sequence must be a permutation of the slot numbers
- We examine every slot in the table, if we have to
- We don't examine any slot more than once

Linear Probing

- The initial probe, h(k,0), determines the entire probe sequence
- Hence, only *m* distinct probe sequences are possible
- Suffers from primary clustering:
  - Long runs of occupied sequences build up
  - Long runs tend to get longer, since an empty slot preceded by i full slots gets filled next with probability (i+1)/m
  - Hence, average search and insertion times increase

Quadratic probing

- $h(k,i) = h'(k) + c_1 * i + c_2 * i^2 \mod m$
- The initial probe position is T[h(k)], later probe positions are offset by amounts that depend on a quadratic function of the probe number i
- Must constrain c_1, c_2 and m to ensure that we get a full permutation of $0, 1, \ldots, m-1$
- Can suffer from secondary clustering: If two keys have the same initial probe position, then their probe sequences are the same

Double Hashing

- $h(k, i) = (h_1(k) + i \times h_2(k)) \mod m$
- Here h_1, h_2 are two auxiliary hash functions.
- Choose m to be a power of 2 and have $h_2(k)$ always return an odd number
- Or let m be a prime number and have $1 < h_2(k) < m$
- m^2 different probe sequences and one for each possible combination of $h_1(k)$ and $h_2(k)$
- Close to the ideal uniform hashing and best method available for open addressing

Analysis

- Uniform Hashing: Each key is likely to have any of the $m!$ permutations of $< 0, 1, \ldots, m-1 >$
- Expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$
- Expected number of probes in a successful search is at most $1/\alpha \log(1/(1-\alpha))$