ENG 346 Data Structures and Algorithms for Artificial Intelligence

Searching Techniques

Dr. Kürşat İnce
kince@gtu.edu.tr

Searching

- The fundamental process of locating a specific element or item within a collection of data.
  - arrays, lists, trees, …
- Objective: Determine whether the requested element exists within the data, and if so, identify its precise location or retrieve it.

Why is it important?

- Efficiency: Efficient searching algorithms improve program performance.
- Data Retrieval: Quickly find and retrieve specific data from large datasets.
- Database Systems: Enables fast querying of databases.
- Problem Solving: Used in a wide range of problem-solving tasks.

Applications of Searching

- Information Retrieval: Search engines like Google, Bing, and Yahoo use sophisticated searching algorithms to retrieve relevant information from vast amounts of data on the web.
- Database Systems: Searching is fundamental in database systems for retrieving specific data records based on user queries, improving efficiency in data retrieval.
- E-commerce: Searching is crucial in e-commerce platforms for users to find products quickly based on their preferences, specifications, or keywords.
- Networking: In networking, searching algorithms are used for routing packets efficiently through networks, finding optimal paths, and managing network resources.
- Artificial Intelligence: Searching algorithms play a vital role in AI applications, such as problem-solving, game playing (e.g., chess), and decision-making processes
- Pattern Recognition: Searching algorithms are used in pattern matching tasks, such as image recognition, speech recognition, and handwriting recognition.

Linear Search

- The Algorithm examines each element one by one.
  - If it finds any element that is exactly the same as the key you're looking for, the search is successful, and it returns the index of key.
  - If it goes through all the elements and none of them matches the key, then that means "No match is Found".

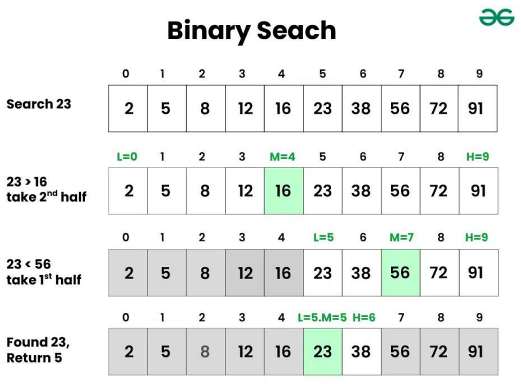
```
LinearSearch(collection, key):
   for each element in collection:
      if element is equal to key:
         return the index of the element
   return "Not found"
```

Time Complexity:
- **Best Case**: O(1)
- **Worst Case:** O(N)
- **Average Case:** O(N)

Binary Search

- Search algorithm used in a sorted data

  ➔Repeatedly divide the search interval in half.



Binary Search – continued

Advantages of Binary Search

- Binary search is faster than linear search, especially for large arrays.
- More efficient than other searching algorithms with a similar time complexity, such as interpolation search or exponential search.
- Binary search is well-suited for searching large datasets that are stored in external memory, such as on a hard drive or in the cloud.

Disadvantages of Binary Search

- The array should be sorted.
- Binary search requires that the data structure being searched be stored in contiguous memory locations.
- Binary search requires that the elements of the array be comparable, meaning that they must be able to be ordered.

Binary Search – Generic Algorithim

```
binarySearch(collection, key):
   left = 0
   right = length(collection) - 1
   while left <= right:
      mid = (left + right) // 2
      if collection[mid] == key:
         return mid
      elif collection[mid] < key:
         left = mid + 1
      else:
         right = mid - 1
   return "Not found"
```

Time Complexity:
- **Best Case**: O(1)
- **Worst Case:** O(log N)
- **Average Case:** O(log N)

Binary Search – Iterative Version

```
# It returns location of x in given array arr
def binarySearch(arr, low, high, x):
    while low <= high:
        mid = low + (high - low) // 2
        # Check if x is present at mid
        if arr[mid] == x:
            return mid
        # If x is greater, ignore left half
        elif arr[mid] < x: low = mid + 1
        # If x is smaller, ignore right half
        else: high = mid - 1

    # Otherwise the element was not present
    return -1
```

```
# Driver Code
if __name__ == '__main__':
    arr = [2, 3, 4, 10, 40]
    x = 10

    # Function call
    result = binarySearch(arr, 0, len(arr)-1, x)
    if result != -1:
        print("Element is present at index", result)
    else:
        print("Element is not present in array")
```

Binary Search – Recursive Version

```
# Returns index of x in arr if present, else -1
def binarySearch(arr, low, high, x):
    # Check base case
    if high >= low:
        mid = low + (high - low) // 2
        # If element is present at the middle itself
        if arr[mid] == x: return mid
        # If element is smaller than mid, then it
        # can only be present in left subarray
        elif arr[mid] > x:
            return binarySearch(arr, low, mid-1, x)
        # Else the element can only be present
        # in right subarray
        else:
            return binarySearch(arr, mid+1, high, x)
    # Element is not present in the array
    else:
        return -1
```

```
# Driver Code
if __name__ == '__main__':
    arr = [2, 3, 4, 10, 40]
    x = 10

    # Function call
    result = binarySearch(arr, 0, len(arr)-1, x)
    if result != -1:
        print("Element is present at index", result)
    else:
        print("Element is not present in array")
```

Hashing

Hash Function

- Function that takes an input (or 'message') and returns a hash code, i.e. number or fixed-size string of bytes.
  - ➔Efficiently map data of arbitrary size to fixed-size values, which are often used as indexes in hash tables.

Key Properties of Hash Functions

- Deterministic: A hash function must consistently produce the same output for the same input.
- Fixed Output Size: The output of a hash function should have a fixed size, regardless of the size of the input.
- Efficiency: The hash function should be able to process input quickly.
- Uniformity: The hash function should distribute the hash values uniformly across the output space to avoid clustering.
- Pre-image Resistance: It should be computationally infeasible to reverse the hash function, i.e., to find the original input given a hash value.
- Collision Resistance: It should be difficult to find two different inputs that produce the same hash value.
- Avalanche Effect: A small change in the input should produce a significantly different hash value.

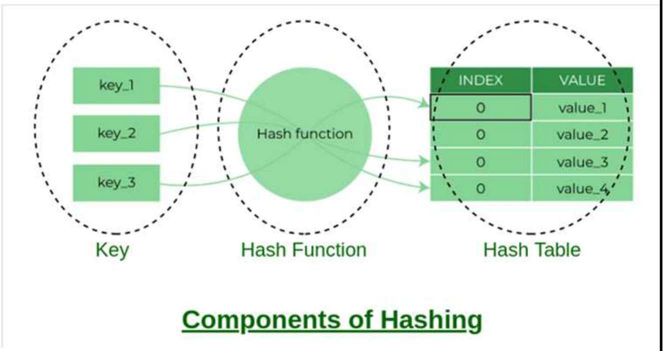
Hashing

Types of Hash Functions

- Division Method: *h(k)=k mod m*
- Multiplication Method: h(k)=⌊m(k*A mod 1)⌋
- Mid-Square Method: Middle digits of k^2
- Folding Method: Fold number on itself
- Cryptographic Hash Functions: MD5, SHA-1, SHA-256
- Universal Hashing: *h(k)=((a*k+b) mod p) mod m*
- Perfect Hashing:

Hash Data Structure

- Store and retrieve <key, value> pairs efficiently
- Key: Any number or string value in <key, value> pair
- Hash Function: Function that returns possible location for key
  - Hash code/index → Data location in a collection
- Hash (table): Data structure that maps keys to values



Components of Hashing

Hash Data Structure

Advantages of Hashing in Data Structures

- **Key-value support:** Hashing is ideal for implementing key-value data structures.
- **Fast data retrieval:** Hashing allows for quick access to elements with constant-time complexity.
- **Efficiency:** Insertion, deletion, and searching operations are highly efficient.
- **Memory usage reduction:** Hashing requires less memory as it allocates a fixed space for storing elements.
- **Scalability:** Hashing performs well with large data sets, maintaining constant access time.
- **Security and encryption:** Hashing is essential for secure data storage and integrity verification.

Hash Table Example

- Data: <Key, Value>
- Hash Function:
- Hash Table:

Collusion Resolution

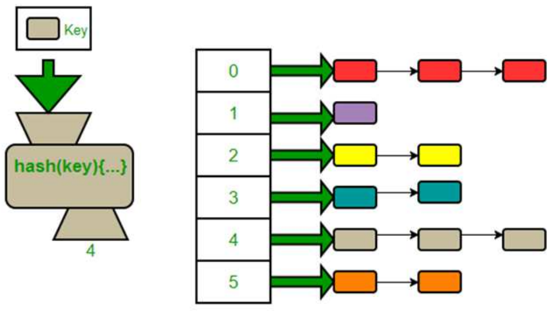
- Collusion occurs when two or more keys have the same hash value

Resolution:

- Linear Probing: Check the next slot until you find an empty slot.
- Quadratic Probing: Check the i^2th slot in the ith iteration.
- Separate Chaing

Separate Chaining

- The array stores Head to linked lists.
- The linked lists store the elements.



Separate Chaining

Advantages:

- Simple to implement.
- Hash table never fills up, we can always add more elements to the chain.
- Less sensitive to the hash function or load factors.
- It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

- The cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
- Wastage of Space (Some Parts of the hash table are never used)
- If the chain becomes long, then search time can become O(n) in the worst case
- Uses extra space for links