RAMCO INSTITUTE OF TECHNOLOGY
Approved by AICTE, New Delhi & Affiliated to Anna University
Accredited by NAAC & An ISO 9001:2015 Certified Institution
NBA Accredited UG Programs: CSE, EEE, ECE and MECH

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

Academic Year: 2021-2022(Even Semester)

Regulation 2021

Subject code & Title: AD3251 - Data Structures Design

Semester/Year : II/ I

Notes

14-06-2022

Prepared by: Dr.M.Kaliappan, Professor/AD
Dr.S.Vimal, Associate Professor/AD

---

2. Searching and Hashing

1. What is linear searching? What is the time complexity?

- A linear search scans one item at a time, without jumping to any item .
- The worst case complexity is O(n), sometimes known an O(n) search
- Time taken to search elements keep increasing as the number of elements are increased

2. What is Binary search?

Binary Search is a searching algorithm for finding an element's position in a sorted array by repeatedly dividing the search interval in half.

Implementation

- Iterative Method
- Recursive Method

3. What are the Applications of Binary search ?

- In libraries of Java, .Net, C++ STL
- While debugging, the binary search is used to pinpoint the place where the error happens.

4. **Why Hashing is needed?**

- After storing a large amount of data. Linear search and binary search perform lookups/search with time complexity of O(n) and O(log n) respectively.
- As the size of the dataset increases, these complexities also become significantly high which is not acceptable.
- We need a technique that does not depend on the size of data. Hashing allows lookups to occur in constant time i.e. O(1).

5 What is Hash Table ?

- Hash table is **a data structure which stores data in an associative manner**. In a hash table, data is stored in an array format, where each data value has its own unique index/hash key value.

6. **What is hash function ?**

A hash function is used for mapping each element of a dataset to indexes in the hash table range $[0,N-1]$, where N is the capacity of the bucket array for a hash table

7. What is Hashing?

Hashing is a technique of mapping a large set of arbitrary data to tabular indexes using a hash function. It is a method for representing dictionaries for large datasets.

8. What is Hash code?

- Hash Code **returns an integer value, generated by a hashing algorithm. T**his integer need not be in the range $[0,N-1]$, and may even be negative.
- The set of hash codes assigned to keys should avoid collisions as much as possible

9. What are the Types of hashcode?

- Bit Representation hash code : 32bit hash code
- Polynomial hash code
- Cyclic-Shift Hash Codes

10. What is Cyclic-Shift Hash Codes?

Polynomial hash code replaces multiplication by a with a cyclic shift of a partial sum by a certain number of bits.

11. Write an algorithm for Cyclic-Shift Hash Codes

```
def hash code(s):
mask = (1 << 32) − 1 # limit to 32-bit integers
h = 0
for character in s:
h = (h << 5 & mask) | (h >> 27)   # 5-bit cyclic shift of running sum
h += ord(character)               # add in value of next character
return h
```

12. What is the use of compression function?

- A compression function is one that minimizes the number of collisions for a given set of distinct hash codes.
- **The Division Method**
  - A simple compression function is the ***division method***, which maps an integer i to i mod N,
- **The MAD Method**
  - It helps eliminate repeated patterns in a set of integer keys, is the ***Multiply-Add-and-Divide*** (or "MAD") method.
  - This method maps an integer i to $[(ai+b) \bmod p] \bmod N$

13. What is the Collision Resolution?

1. Linear Probing
2. Chaining

13. How will you avoid Collision using Linear Probing?

- One way to resolve collision is to find another open slot whenever there is a collision and store the item in that open slot. The search for open slot starts from the slot where the collision happened. It moves sequentially through the slots until an empty slot is encountered. The movement is in a circular fashion. Hence, covering the entire hash table. This kind of sequential search is called Linear Probing.

14 How will you avoid Collision using Chaining?

This allows multiple items exist in the same slot/index. This can create a chain/collection of items in a single slot. When the collision happens, the item is stored in the same slot using chaining mechanism.

15. What is load factor ?

- if there are n entries and b is the size of the array there would be n/b entries on each index. This value n/b is called the **load factor** that represents the load that is there on our map.
- Default Load factor=0.75
- This Load Factor needs to be kept low, so that number of entries at one index is less and so is the complexity almost constant, i.e., O(1)

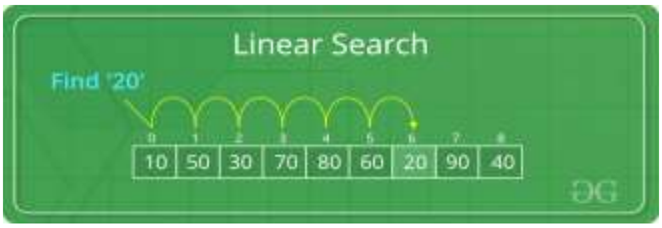
16. What are the applications are hashing?

- Password Verification
- Game Boards
- Graphics
- Message Digest
- Compiler Operation
- Rabin-Karp Algorithm
- Linking File name and path together

PART B

1 Write a python program to implement linear searching

A linear search scans one item at a time, without jumping to any item



```
def search(arr, n, x):
    for i in range(0, n):
        if (arr[i] == x):
            return i
    return -1

arr = [2, 3, 4, 10, 40]
x = 10
n = len(arr)

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

Time Complexity : O(n)

Auxiliary Space: O(1)

---

2. Write an optimized python program to implement linear search?

A linear search scans one item at a time, without jumping to any item

```
def search(arr, search_Element):
    left = 0
    length = len(arr)
    position = -1
    right = length - 1
    for left in range(0, right, 1):  # Run loop from 0 to right
        if (arr[left] == search_Element): # If search_element is found with  left variable
            position = left
            print("Element found in Array at ", position +1, " Position with ", left + 1, " Attempt")
            break
        if (arr[right] == search_Element):  # If search_element is found with  right variable
            position = right
            print("Element found in Array at ", position + 1, " Position with ", length -
 right, " Attempt")
```

```
            break
        left += 1
        right -= 1
    if (position == -1):  # If element not found
        print("Not found in Array with ", left, " Attempt")
arr = [1, 2, 3, 4, 5]
search_element = 5
search(arr, search_element) # Function call
```

Time Complexity : O(n)

Auxiliary Space: O(1)

----

3. Implementation of Binary Search in python using Iterative method

Binary Search is a searching algorithm for finding an element's position in a sorted array by repeatedly dividing the search interval in half

```
def binarySearch(array, x, low, high):    # Repeat until the pointers low and high meet each ot
her
    while low <= high:
        mid = (low + high)//2

        if array[mid] == x:
            return mid
        elif array[mid] < x:
            low = mid + 1
        else:
            high = mid - 1

    return -1

array = [3, 4, 5, 6, 7, 8, 9]
x = 4
result = binarySearch(array, x, 0, len(array)-1)

if result != -1:
    print("Element is present at index " + str(result))
else:
    print("Not found")
```

Time Complexities

- Best case complexity: O(1)
- Average case complexity: O(log n)
- Worst case complexity: O(log n)
- Space Complexity

The space complexity of the binary search is O(1).

---

4. Implementation of Binary Search in python using recursive method

Binary Search is a searching algorithm for finding an element's position in a sorted array by repeatedly dividing the search interval in half

```
# Binary Search in python : Recursive method
def binarySearch(array, x, low, high):
    if high >= low:
        mid = (low + high)//2
        if array[mid] == x: # If found at mid, then return it
            return mid

        elif array[mid] > x: # Search the left half
            return binarySearch(array, x, low, mid-1)
        else:
            return binarySearch(array, x, mid + 1, high) # Search the right half

    else:
        return -1
array = [3, 4, 5, 6, 7, 8, 9]
x = 4

result = binarySearch(array, x, 0, len(array)-1)

if result != -1:
    print("Element is present at index " + str(result))
else:
    print("Not found")
```

Time Complexities

- Best case complexity: O(1)
- Average case complexity: O(log n)
- Worst case complexity: O(log n)
- Space Complexity

The space complexity of the binary search is O(1).

---

5. Implementation of Hash table using python.

Hash table is **a data structure which stores data in an associative manner**. In a hash table, data is stored in an array format, where each data value has its own unique index/hash key value

```
hash_table = [None] * 10     # Implementation of Hashtable  using python
print (hash_table)

def hashing_func(key):
  return key % len(hash_table)
# print the calculated hash key
print (hashing_func(10)) # Output: 0
print (hashing_func(20)) # Output: 0
print (hashing_func(25)) # Output: 5
def insert(hash_table, key, value):
  hash_key = hashing_func(key)
  hash_table[hash_key] = value

insert(hash_table, 10, 'Nepal')
print (hash_table)


insert(hash_table, 25, 'USA')
print (hash_table)
```

Output

['**Nepal**', None, None, None, None, None, None, None, None, None]

['Nepal', None, None, None, None, '**USA**', None, None, None, None]

----

6. How will you avoid Collision using Chaining?

This allows multiple items exist in the same slot/index. This can create a chain/collection of items in a single slot. When the collision happens, the item is stored in the same slot using chaining mechanism.

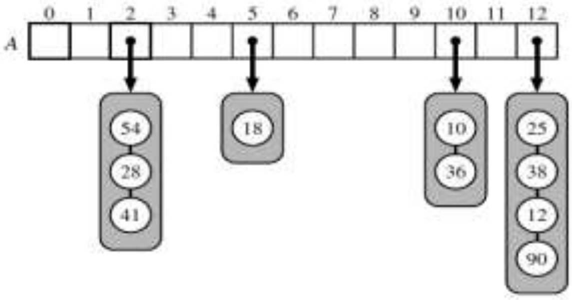


Fig: The compression function is $h(k)=k \bmod 13$

```
HashTable = [[] for _ in range(10)] # Creating Hashtable as a nested list
display_hash (HashTable)
def Hashing(keyvalue): # Hashing Function to return key for every value.
  hash_key=keyvalue % len(HashTable)
  return hash_key

def insert(Hashtable, keyvalue, value):  # Insert Function to add # values to the hash table
  hash_key = Hashing(keyvalue)
  Hashtable[hash_key].append(value)
def display_hash(hashTable): # Function to display hashtable
  for i in range(len(hashTable)):
    print(i, end = " ")

    for j in hashTable[i]:
      print("-->", end = " ")
      print(j, end = " ")

    print()
insert(HashTable, 10, 'Allahabad')
insert(HashTable, 25, 'Mumbai')
insert(HashTable, 20, 'Mathura')
insert(HashTable, 9, 'Delhi')
insert(HashTable, 21, 'Punjab')
insert(HashTable, 21, 'Noida')
display_hash (HashTable)
```

outout

```
0 --> Allahabad --> Mathura
1 --> Punjab --> Noida
2
3
4
5 --> Mumbai
6
7
8
9 --> Delhi
```

----

7. How will you avoid Collision using Chaining?

This allows multiple items exist in the same slot/index. This can create a chain/collection of items in a single slot. When the collision happens, the item is stored in the same slot using chaining mechanism.

```
class hashTable:
  def _init_(self):
    self.table = [None]*10 # initialize hash Table
    self.elementCount =0

  def hashFunction(self, key): # Find Hash code
    return key % self.size

  def insert(self, key, element):  # inserts element into the hash table
    position = self.hashFunction(key)

    if self.table[position] == None:#checking if eth position is empty
      self.table[position] = element
      self.elementCount += 1
    # collision occured hence we do linear probing
    else:
      while self.table[position] != None:
        position += 1

      self.table[position] = element
      self.elementCount += 1
# method that searches for an element in the table returns position of element if found else ret
urns False
  def search(self, key):
    position = self.hashFunction(key)
    if(self.table[position] == element):
      print("element found ")
      return position
    else:
      print("element found ")
  # method to display the hash table
  def display(self):
    print("\n")
    for i in range(self.size):
      print("Hash Value: "+i + "\t\t" + self.table[i])
```

----

8. Write a python program to implement the quadratic probing.

Quadratic probing is an open addressing scheme in computer programming for resolving hash collisions in hash tables. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

```
def hashing(table, tsize, arr, N):   # Iterating through the array
  for i in range(N):      # Computing the hash value
    hv = arr[i] % tsize
    if (table[hv] == -1): # Insert in the table if there is no collision
      table[hv] = arr[i]
    else:
      for j in range(tsize):  # If there is a collision iterating through all possible quadratic values
        t = (hv + j * j) % tsize       # Computing the new hash value
        if (table[t] == -1):
          table[t] = arr[i] # Break the loop after  inserting the value  in the table
          break
```

---

9. Write a program to implement rehashing.

Rehashing means hashing again. Basically, when the load factor increases to more than its pre-defined value (default value of load factor is 0.75), the complexity increases.

```
def __setitem__(self, k, v):
    j = self._hash_function(k)
    self._bucket_setitem(j, k, v)              # subroutine maintains self._n
    if self._n > len(self._table) // 2:         # keep load factor <= 0.5
      self._resize(2 * len(self._table) - 1)      # number 2^x - 1 is often prime

  def __delitem__(self, k):
   j = self._hash_function(k)
    self._bucket_delitem(j, k)                   # may raise KeyError
    self._n -= 1

  def _resize(self, c):
    """Resize bucket array to capacity c and rehash all items."""
    old = list(self.items())       # use iteration to record existing items
    self._table = c * [None]       # then reset table to desired capacity
    self._n = 0                    # n recomputed during subsequent adds
    for (k,v) in old:
      self[k] = v                  # reinsert old key-value pair
def _bucket_setitem(self, j, k, v):
    found, s = self._find_slot(j, k)
    if not found:
```

```
        self._table[s] = self._Item(k,v)          # insert new item
        self._n += 1                              # size has increased
    else:
        self._table[s]._value = v                 # overwrite existing
```
----

10. Discuss about efficiency of hash table.

- If our hash function is good, then we expect the entries to be uniformly distributed in the N cells of the bucket array. Thus, to store n entries, the expected number of keys in a bucket would be n/N, which is $O(1)$ if n is $O(N)$. n/N *is a load factor*. The default load factor is 0.75
- The costs associated with a periodic rehashing, to resize a table after occasional insertions or deletions can be accounted for separately

Operation	List	Hash Table	
		expected	worst case
__getitem__	$O(n)$	$O(1)$	$O(n)$
__setitem__	$O(n)$	$O(1)$	$O(n)$
__delitem__	$O(n)$	$O(1)$	$O(n)$
__len__	$O(1)$	$O(1)$	$O(1)$
__iter__	$O(n)$	$O(n)$	$O(n)$

References:

1. Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser, "Data Structures and Algorithms in Python" (An Indian Adaptation), Wiley, 2021
2. https://www.geeksforgeeks.org/python-oops-concepts/
3. https://www.programiz.com/dsa/circular-queue
4. https://www.techiedelight.com/iterative-merge-sort-algorithm-bottom-up/
5. https://blog.chapagain.com.np/hash-table-implementation-in-python-data-structures-algorithms/
6. https://www.codingninjas.com/codestudio/library/load-factor-and-rehashing
7. https://onecompiler.com/python/3wq9gxuuy
8. Madhavan Mukund, Chennai Mathematical Institute, NPTEL MOOC on Programming Data Structure and Algorithms in Python
9. http://www.cse.unsw.edu.au/~cs2011/05s1/tut/solutions/tut08sol.html