# Chapter Four: Searching and Sorting

Sem. I - 2017

Department of Software Engineering

ITSC-AAIT

Natnael A.(MSc.)

# Searching

- Searching is the algorithmic process of finding a particular item in a collection of items. A search typically answers either True or False as to whether the item is present.
- The Sequential Search
  - Data which are stored in a collection have a linear or sequential relationship.
  - Each data item is stored in a position relative to the others.
  - In Python lists, these relative positions are the index values of the individual items.
  - Sequential Search starts from the first item in the list and proceeds to the consecutive index till it find the target or reached the end of the list.

# Sequential Search

- A python implementation of sequential search is depicted below. It accepts a list and the item to be searched in the list.

```
''' Sequential Search '''
def sequential_search(a_list, item):
    pos = 0
    found = False
    while pos < len(a_list) and not found:
        if a_list[pos] == item:
            found = True
        else:
            pos = pos+1
    return found
```

# Sequential Search

- Three Scenarios while using sequential search
  - Best Case – the item is at the beginning of the list O(1)
  - Average Case - the item is half way to the end of the list T(n)= n/2
  - Worst Case – the item is at the end of the list O(n)
- Hence the complexity of the sequential search is O(n).

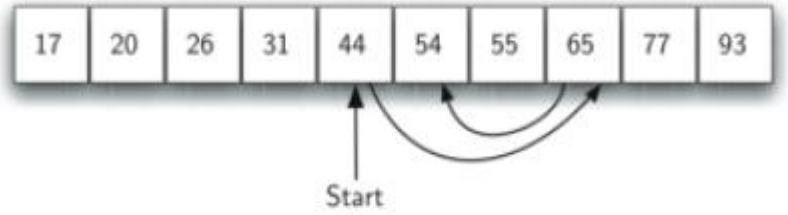
# Sequential Search

- Analysis of Sequential Search
- What would happen to the sequential search if the items were ordered in some way?
  - Computes till it reaches item or the item greater than the target.

```
''' Sequential Search  on a sorted list'''
def ordered_sequential_search(a_list, item):
    pos = 0
    found = False
    stop = False
    while pos < len(a_list) and not found and not stop:
        if a_list[pos] == item:
            found = True
        else:
            if a_list[pos] > item:
                stop = True
            else:
                pos = pos+1
    return found
```

# The Binary Search

- Takes greater advantage of the ordered list.
- Binary Search starts by examining the middle item and exits w/ success if it is the target item.
- If it is not the correct item then the binary search uses the ordered nature of the list to eliminate half of the remaining items.
- Likewise, the search continues till the item is found or reached the last item.
- This algorithm is a great example of a divide and conquer strategy.




```
''' Binary Searching '''
def binary_search(a_list, item):
    first = 0
    last = len(a_list) - 1
    found = False
    while first <= last and not found:
        midpoint = (first + last) // 2 # Integer Division
        if a_list[midpoint] == item:
            found = True
        else:
            if item < a_list[midpoint]:
                last = midpoint - 1
            else:
                first = midpoint + 1
    return found
```

# The Binary Search

- **Analysis of Binary Search**
- Baseline: each comparison eliminates about half of the remaining items from consideration.
- So, the question is how many time can we split the list?

| Comparisons | Approximate Number Of Items Left |
|---|---|
| 1 | $\frac{n}{2}$ |
| 2 | $\frac{n}{4}$ |
| 3 | $\frac{n}{8}$ |
| ... | ... |
| $i$ | $\frac{n}{2^i}$ |

- The number of comparisons necessary to get to this point is $i$ where $n/2^i = 1$.
- Solving for $i$ gives us $i = \log n$. Therefore, the binary search is $O(\log n)$.
  - *Remember: For* $x > 0$ *and* $b > 0$, $b<>1$, $\quad Y = \log_b x$ *is equivalent to* $b^y = x$

# The Binary Search

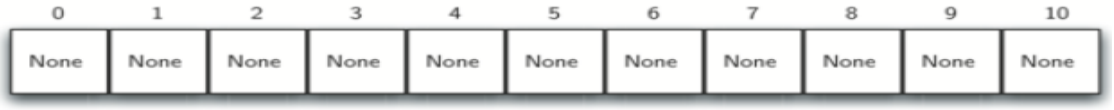
- **Analysis of Binary Search**
- Consider the following recursive solution, it uses the slice operator to create the left half of the list that is then passed to the next invocation (similarly for the right half as well).
- The slice operator in python is O(k).
- Hence, this binary search will not perform in logarithmic time.
- This can be remedied by passing the list along with the starting and ending indices.

```
'''Recursive Binary Search With Slice Operator '''
def binary_search(a_list, item):
    if len(a_list) == 0:
        return False
    else:
        midpoint = len(a_list) // 2
    if a_list[midpoint] == item:
        return True
    else:
        if item < a_list[midpoint]:
            return binary_search(a_list[:midpoint], item)
        else:
            return binary_search(a_list[midpoint + 1:], item)
```

- The cost of sorting makes binary search unsuitable than sequential search in small sized lists.

# Hashing

- Hashing achieves O(1) searching complexity.
- A hash table is a collection of items which are stored in such a way as to make it easy to find them later.
- Slot is a position in the table that can hold an item and it is named by an integer value staring from 0.
- Initially, the hash table contains no items so every slot is empty.

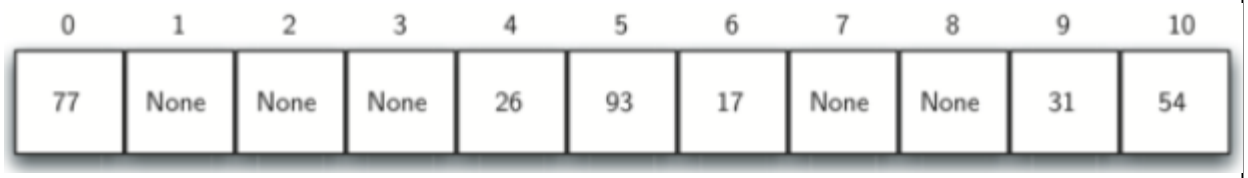



Hash Table with 11 Empty Slots

- The mapping between an item and the slot where that item belongs in the hash table is called the hash function.
- The hash function will take any item in the collection and return an integer in the range of slot names, between 0 and $m - 1$

# Hashing

- **Remainder Method**
- Simply takes an item and divides it by the table size, returning the remainder as its hash value.
- Once the hash values have been computed, we can insert each item into the hash table at the designated position.



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 77 | None | None | None | 26 | 93 | 17 | None | None | 31 | 54 |

- Load factor is the measure of the percentage of occupied items from available hash table slots.

$$\lambda = \frac{\text{number\_of\_items}}{\text{table\_size}}$$

- While searching, the hash function is used to compute the slot name and check the hash table if the item with that name is present with O(1) complexity.

# Hashing

- Some times, two or more items would need to be in the same slot, collision/Clash. In this case, searching with O(1) complexity won't work.
- E.g Using a remainder function(44) on the above function causes a replacement of 77.

❑ **Hash Functions**

- A hash function that maps each item into a unique slot is referred to as a perfect hash function.
- There is no systematic way to construct a perfect hash function for arbitrary collection of items.
- **Solution I**: increase the size of the hash table if the number of items is relatively small.
  - The major goal is to create a hash function that minimizes the number of collisions, is easy to compute, and evenly distributes the items in the hash table. Hence, there must be ways to extend the simple remainder method.

# Hashing

- **The folding method**
- Begins by dividing the item into equal size pieces (the last piece may not be of equal size).
- These pieces are then added together to give the resulting hash value.
  - E.g. an item “436-555- 4601” will be divided in to groups with two characters as 2 (43, 65, 55, 46, 01). And each term will be added yielding 43 + 65 + 55 + 46 + 01 = 210.
- Assuming the size of the hash table, 11, the hash function yields 210%11→ 1.
- Some folding methods go one step further and reverse every piece before the addition. For the above example, we get **43 + 56 + 55 + 64 + 01 = 219** which gives **219%11 = 10**.

# Hashing

- **The mid-square method**
  - First square the item, and then extract some portion of the resulting digits.
  - E.g: extracting the middle two of the squared item; $44^2 = 1,936$ **93%11 → 5**.
  - It is also possible to create hash functions for **character-based items** such as **strings**.
  - The word "cat" can be thought of as a sequence of **ordinal values**.

    **ord('c') → 99**
    **ord('a') → 97**
    **ord('t') → 116**

| Item | Remainder | Mid-Square |
|---|---|---|
| 54 | 10 | 3 |
| 26 | 4 | 7 |
| 93 | 5 | 9 |
| 17 | 6 | 8 |
| 77 | 0 | 4 |
| 31 | 9 | 6 |

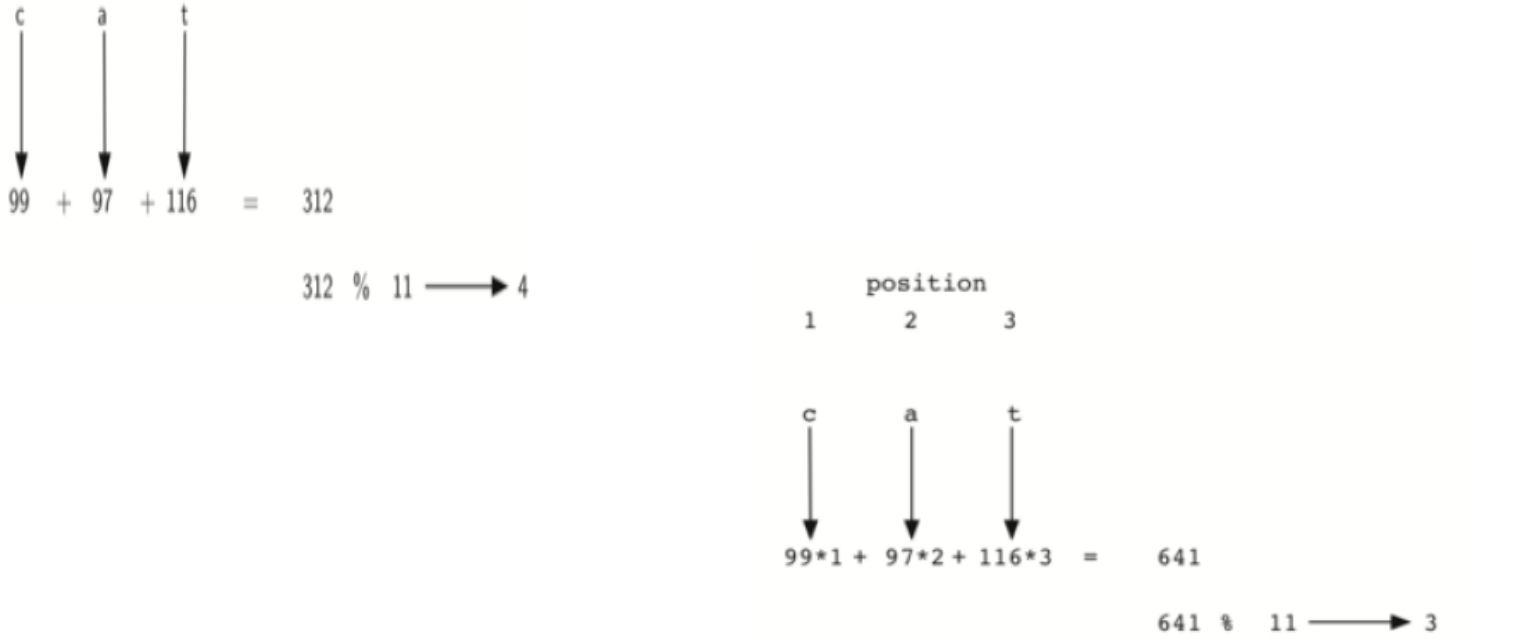
Comparisons of Remainder and Mid-Square Methods

- The Folding Method

```
''' Character Based Hash Function'''
def hash(a_string, table_size):
    sum = 0
    for pos in range(len(a_string)):
        sum = sum + ord(a_string[pos])
    return sum % table_size
```

# Hashing

- Character Based Folding Method
  - Even in this case, anagrams will always be given the same hash value.
  - Using the position of the character as a weight helps to resolve this problem.




Hashing a String Using Ordinal Values with Weighting

# Hashing

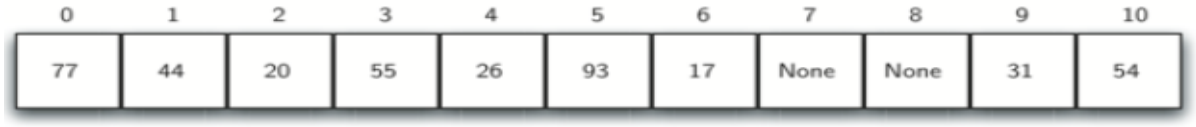
- **Collision Resolution**
  - Is a systematic way to minimize the possibility of two or more items hash in the same slot..
  - **Solution**: simply look in free slots if collision is about to occur.
  - It is also called **Open Addressing**, which start at the original hash value and then move in a sequential manner through the slots until we encounter the first slot that is empty.
  - **Linear Probing** is a technique of systematically visiting each slot one at a time.
  - Searching for item in a Hash table which is constructed by using **open addressing and linear probing** requires utilization of similar methods.
  - So, searching such kind of hash tables requires utilization of sequential search from the default location of an item to the target item if not to empty slot.

# Hashing

- **Collision Resolution**
- A **disadvantage** to linear probing is the tendency for clustering; if many collisions occur at the same hash value, a number of surrounding slots will be filled by the linear probing resolution.
- One way to deal with clustering is to extend the linear probing technique in a way it puts colliding items evenly or dispersing. E.g. collision resolution with plus 3.
- **Rehashing** is the general name of looking for another slot after a collision occurrence.
  - Plus 1 Rehash → new_hash_value = rehash(old_hash_value) where rehash (pos) = (pos + 1)%size_of_table.
  - Plus 3 Rehash → rehash(pos) = (pos + 3)%size_of_table.
- In general,
  - rehash(pos) = (pos + skip) % sizeoftable

# Hashing

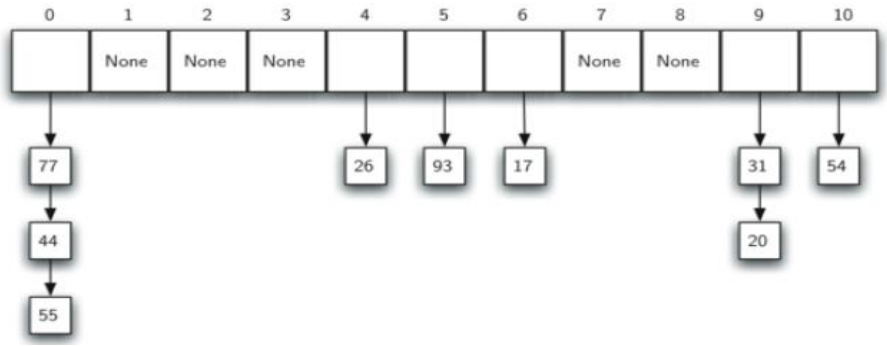
- **Collision Resolution**
- **Quadratic Probing**
  - A variation of the linear probing; no constant skip function is used.
  - The hash value is incremented by using rehash function; This means that if the first hash value is $h$, the successive values are $h$+1, $h$+4, $h$+9, $h$+16, and so on.
  - Quadratic probing uses a skip consisting of successive perfect squares.




Collision Resolution with Quadratic Probing

# Hashing

- **Collision Resolution**
- **Chaining**
  - Is another collision resolution mechanism in which each slot holds a reference to a collection (or chain) of items.
  - As more and more items hash to the same location, the difficulty of searching for the item in the collection increases.




- [LAB] - Implement Hash Table using Map ADT.

# Sorting

- Sorting is the process of placing elements from a collection in some kind of order.
- Sorting is an important area of study in computer science.
- The efficiency of sorting algorithm depends on the number of items being processed.
- A complex sorting algorithm should not be applied to small collections of items.
- **Comparing** and **positioning/rearranging** items in a collection are the major operations performed in any type of sorting algorithms.

# Bubble Sort

- Makes multiple passes through a list.
- It compares adjacent items and exchanges those that are out of order. In each pass, each item "bubbles" up to the location where it belongs.

```
    ''' Bubble Sort '''
def bubble_sort(a_list):]
    :#start , stop, step(backward or forward traversing)
    for pass_num in range(len(a_list) - 1, 0, -1)
        for i in range(pass_num):
            if a_list[i] > a_list[i + 1]:
                temp = a_list[i]
                a_list[i] = a_list[i + 1]
                a_list[i + 1] = temp
```

- The exchange operation, sometimes called a "swap", requires additional memory space.
- Regardless of how the items are arranged in the initial list, $\boldsymbol{n-1}$ passes will be made to sort a list of size $n$. Recall that the sum of the first $n$ integers is $\frac{1}{2}*n^2 + \frac{1}{2}*n$. The sum of the first $n-1$ integers is $\frac{1}{2}*n^2 + \frac{1}{2}*n - n$, which is $\frac{1}{2}*n^2 - \frac{1}{2}*n \rightarrow$ $O(n^2)$.

# Bubble Sort

- The bubble sort is known as the most inefficient sorting algorithm.
- However, a bubble sort could be modified to enable
  - Early stop, if the list is found sorted.

```
    ''' Short Bubble Sort '''
def short_bubble_sort(a_list): #
    exchanges = True
    pass_num = len(a_list) - 1
    while pass_num > 0 and exchanges:
        exchanges = False
        for i in range(pass_num):
            if a_list[i] > a_list[i + 1]:
                exchanges = True
                temp = a_list[i]
                a_list[i] = a_list[i + 1]
                a_list[i + 1] = temp
        pass_num = pass_num - 1
```

# Reading

Problem Solving with Data Structure and Algorithms Page [147-185]