SC1007
Searching

Dr. Liu Siyuan
Email: syliu@ntu.edu.sg
Office: N4-02C-72a

Overview

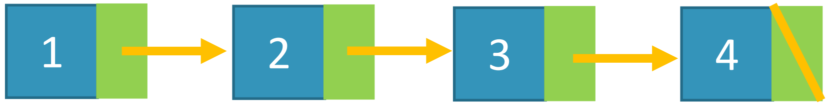
- Exhaustive Algorithm:
  - Sequential Search
- Decrease-and-conquer Algorithm:
  - Binary Search
  - Jump Search

Time Complexity of Sequential Search

```
def search(head, a):
   pt = head
   while pt is not None and pt.key != a:
      pt = pt.next
   return pt
```

c_1

c_2



Assume that the search key ***a*** is in the list

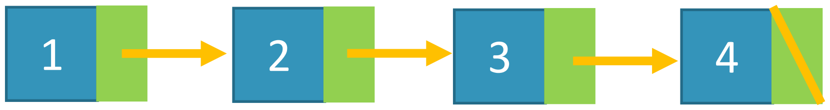
1. Best-case analysis: c_1 when ***a*** is the first item in the list => Θ (1)
2. Worst-case analysis:
3. Average-case analysis:

Time Complexity of Sequential Search

```
def search(head, a):
    pt = head
    while pt is not None and pt.key != a:
        pt = pt.next
    return pt
```

c_1

c_2 (n-1) iterations



Assume that the search key ***a*** is in the list

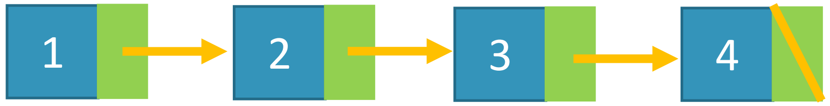
1. Best-case analysis: c_1 when ***a*** is the first item in the list => Θ (1)
2. Worst-case analysis: $c_2 \cdot (n-1) + c_1$ => Θ (n) when ***a*** is the last item in the list
3. Average-case analysis $p_1 \times \textit{time to search for item } 1 + p_2 \times \textit{time to search for item } 2 + \cdots + p_n \times \textit{time to search for item } n$

Time Complexity of Sequential Search

```
def search(head, a):
    pt = head
    while pt is not None and pt.key != a:
        pt = pt.next
    return pt
```

c_1

c_2 (n-1) iterations



Assume that the search key ***a*** is always in the list

1. Best-case analysis: c_1 when **a** is the first item in the list => Θ (1)

2. Worst-case analysis: $c_2 \cdot (n-1) + c_1$ => Θ (n) when **a** is the last item in the list

3. Average-case analysis: $\boldsymbol{p_1 c_1 + p_2(c_1 + c_2) + p_3(c_1 + 2c_2) + \cdots + p_n(c_1 + (n-1)c_2)}$

Assume that every item in the list has an equal probability as a search key, i.e., $p_i = \frac{1}{n}$

$$\boldsymbol{\frac{1}{n}[c_1 + (c_1 + c_2) + (c_1 + 2c_2) + \cdots + (c_1 + (n-1)c_2)] = \frac{1}{n}\sum_{i=1}^{n}(c_1 + c_2(i-1))}$$

$$= \frac{1}{n}[nc_1 + c_2 \sum_{i=1}^{n}(i-1)\ ]$$

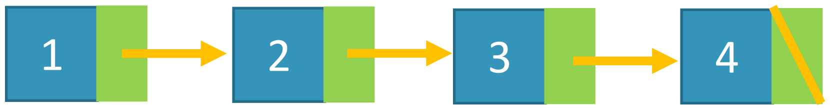
$$= c_1 + \frac{c_2}{n} \cdot \frac{n}{2}(0 + (n-1)) = c_1 + \frac{c_2(n-1)}{2} = \Theta\ (n)$$

Time Complexity of Sequential Search

```
def search(head, a):
  pt = head
  while pt is not None and pt.key != a:
    pt = pt.next
  return pt
```

c_1

c_2



If the search key is in the list, on average: $c_1 + \frac{c_2(n-1)}{2} = \Theta\ (n)$

If the search key, a, is not in the list, then the time complexity is

$$c_1 + nc_2 = \Theta\ (n)$$

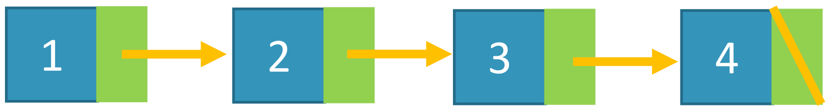
Since the probability of the search key is in the list is unknown, we only can have

$$f(n) = P(a\ in\ the\ list)(c_1 + \frac{c_2(n-1)}{2}) + (1 - P(a\ in\ the\ list))(c_1 + nc_2)$$

It is still a linear function. **Θ (n)**

Time Complexity of Sequential Search

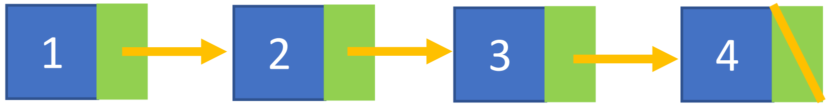
```
def search(head, a):
   pt = head
   while pt is not None and pt.key != a:
      pt = pt.next
   return pt
```



- The data is stored unordered
- To search a key, every element is required to read and compare
- This is a brute-force approach or a näive algorithm
- Its time complexity is **O(n)**
- How can we improve it?

Decrease and Conquer: Binary Search

- Given a sorted list



- Whether a search key ***a*** is in the list?

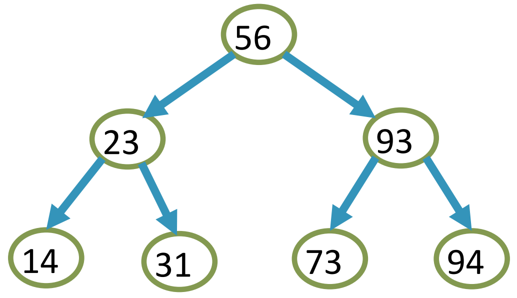
```
def binary_search_recursive(arr, left, right, target):
    if left > right:
        return -1
    mid = left + (right - left) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return binary_search_recursive(arr, mid + 1, right, target)
    else:
        return binary_search_recursive(arr, left, mid - 1, target)
```

Time Complexity of Binary Search

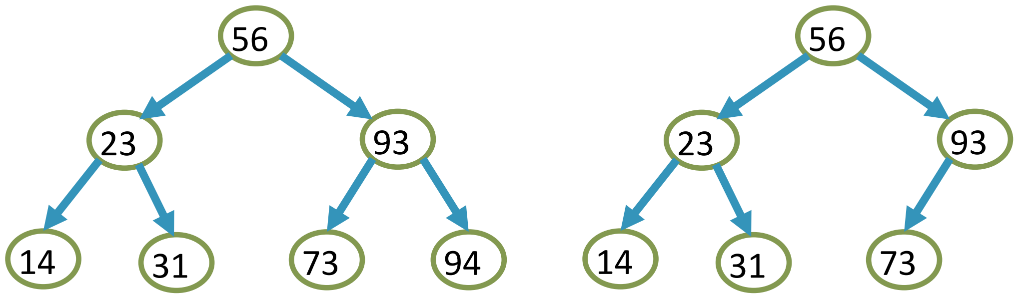
```
def binary_search_recursive(arr, left, right, target):
    if left > right:
        return -1
    mid = left + (right - left) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return binary_search_recursive(arr, mid + 1, right, target)
    else:
        return binary_search_recursive(arr, left, mid - 1, target)
```

```
def binary_search(self, target, current_node):
    if current_node is None:
        return False
    elif target == current_node.data:
       return True
    elif target < current_node.data:
       return self.binary_search(target,current_node.left)
    else:
       return self.binary_search(target,current_node.right)
```

- Given a sorted list, e.g.,
    - 14, 23, 31, 56, 73, 93, 94
- We can build a BST



Terminology



- The Height of a tree: The number of **edges** on the longest path from the root to a leaf
- The Depth of a node: The number of edges from the node to the root of its tree.

For a complete binary tree with height H, we have:

$$\mathbf{2^H\text{-}1 < \mathit{n} \leq 2^{H+1} - 1}$$

where n is an integer and the size of the tree

$$\mathbf{2^H \leq \mathit{n} < 2^{H+1}} \qquad \mathbf{(e.g., 7<n \leq 15 \equiv 8 \leq n<16)}$$

$$\mathbf{H \leq \log_2 n < H+1}$$

If H is an integer, H+1 must be the next integer.

Height = $\lfloor \log_2 n \rfloor$

Binary Search – Worst Case Time Complexity

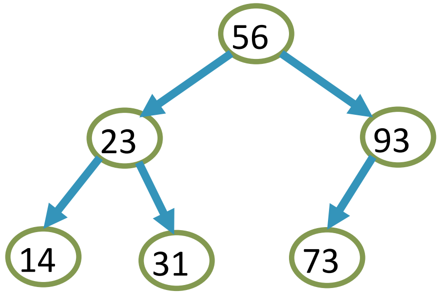
```
def binary_search(self, target, current_node):
    if current_node is None:
        return False
    elif target == current_node.data:
        return True
    elif target < current_node.data:
        return self.binary_search(target,current_node.left)
    else:
        return self.binary_search(target,current_node.right)
```

f(n)

Constant c

$\boldsymbol{f((n-1)/2)}$

$\boldsymbol{f((n-1)/2)}$



- Assume a complete binary tree

$$f(n) = f\left(\frac{n-1}{2}\right) + c \quad = f\left(\frac{\left(\frac{n-1}{2}\right) - 1}{2}\right) + 2c = f\left(\frac{n-1-2}{2^2}\right) + 2c$$

$$= f\left(\frac{\frac{n-1-2}{2^2} - 1}{2}\right) + 3c = f\left(\frac{n-1-2-2^2}{2^3}\right) + 3c$$

…

Binary Search – Worst Case Time Complexity

$$f(n) = f\left(\frac{n-1}{2}\right) + c$$
$$= f\left(\frac{n-(1+2+\cdots+2^{k-2}+2^{k-1})}{2^k}\right) + kc$$
$$= f\left(\frac{n-2^k+1}{2^k}\right) + kc$$
$$= f(1) + kc$$
$$= c + kc$$
$$= (\lfloor \log_2 n \rfloor + 1)c$$
$$= \Theta(\log_2 n)$$

$$0 < \frac{n-2^k+1}{2^k} \leq 1$$
$$0 < \frac{n+1}{2^k} - 1 \leq 1$$
$$1 < \frac{n+1}{2^k} \leq 2$$
$$2^k < n+1 \leq 2^{k+1}$$
$$k < \log_2(n+1) \leq k+1$$
$$\lceil \log_2(n+1) \rceil = k+1$$
$$\lfloor \log_2 n \rfloor + 1 = k+1$$
$$k = \lfloor \log_2 n \rfloor$$

From previous slide:

$2^k \leq n < 2^{k+1} \equiv$
$2^k - 1 < n \leq 2^{k+1} - 1$

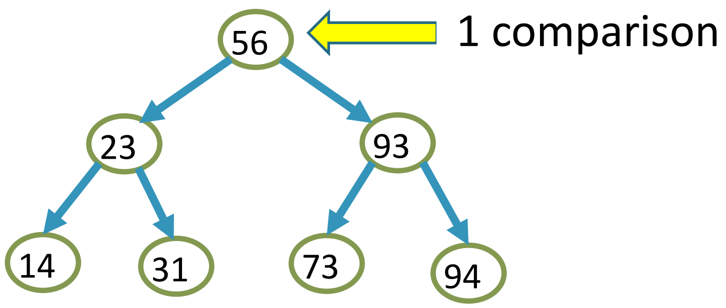
Therefore
$log(n+1) \leq k+1$
$\lceil log(n+1) \rceil = k+1$
$logn \geq k$
$\lfloor logn \rfloor = k$

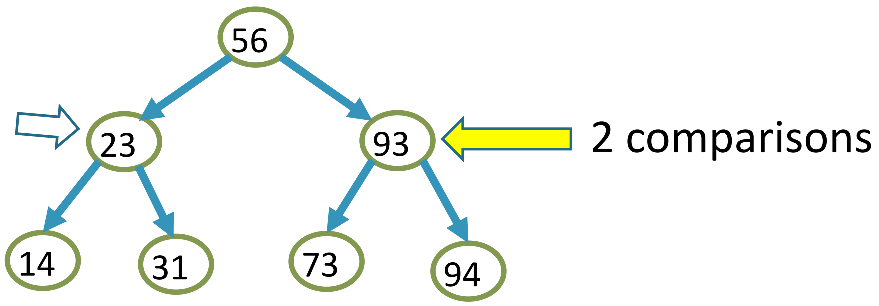
Binary Search – Average Case Time Complexity

- $A_s(n)$: # of comparisons for successful search
- $A_f(n)$: # of comparisons for unsuccessful search (worst case): $\Theta(\log_2 n)$

$$A(n) = qA_s(n) + (1-q)A_f(n)$$

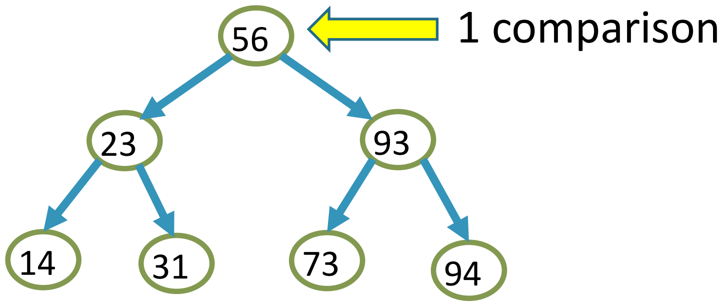
For $A_s(n)$, we assume $n = 2^k - 1$ first

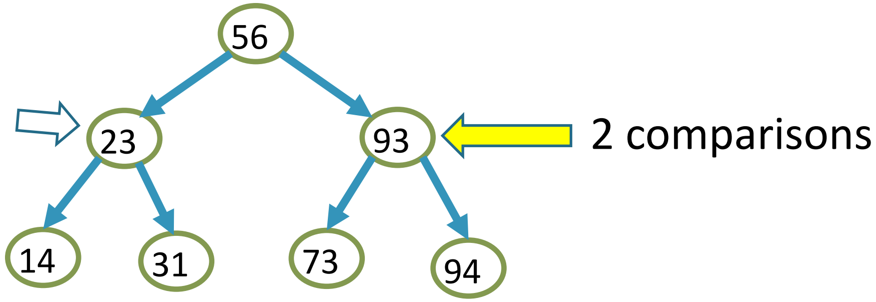




Binary Search – Average Case Time Complexity

$$A(n) = qA_s(n) + (1-q)A_f(n)$$





- For $A_s(n)$, we assume $n = 2^k - 1$ first
- We can observe that:
  - 1 position requires 1 comparison (level 1)
  - 2 positions requires 2 comparisons (level 2)
  - 4 positions requires 3 comparisons (level 3)
  - … …
  - 2^{t-1} positions requires t comparisons ((level t)

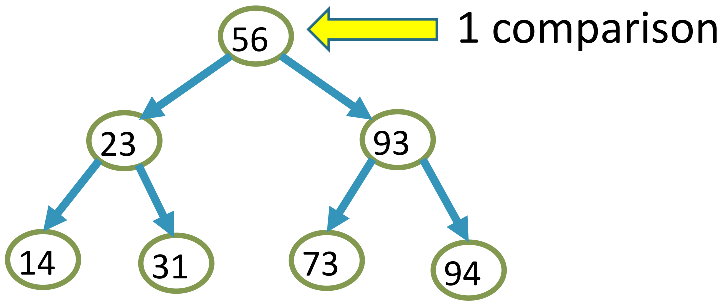
$$A_s(n) = \sum_{t=1}^{k} p_t \times (\#comparisons\ at\ level\ t)$$

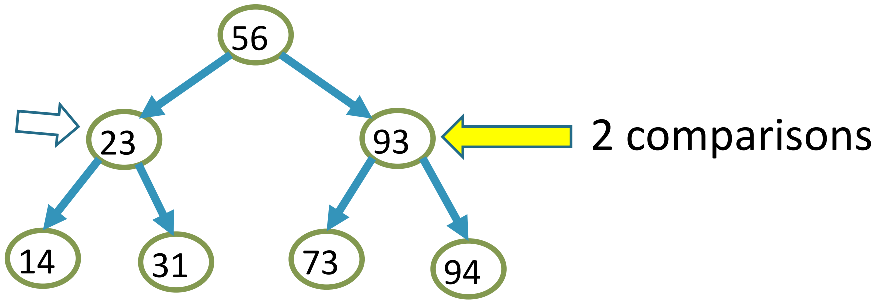
$$= \sum_{t=1}^{k} \frac{1}{n} \times (\#positions\ at\ level\ t) \times (\#comparisons\ at\ level\ t)$$

$$= \sum_{t=1}^{k} \frac{1}{n} \times 2^{t-1} \times t$$

Binary Search – Average Case Time Complexity

$$A(n) = qA_s(n) + (1-q)A_f(n)$$





- Assuming n=2^k-1, we have

$$\sum_{t=1}^{k} t2^{t-1} = 1\cdot 1 + 2\cdot 2 + 3\cdot 4 + 4\cdot 8 + \ldots + k\cdot 2^{k-1}$$

$$2\sum_{t=1}^{k} t2^{t-1} = \quad 1\cdot 2 + 2\cdot 4 + 3\cdot 8 + \ldots + (k-1)\cdot 2^{k-1} + k\cdot 2^{k}$$

$$(2-1)\sum_{t=1}^{k} t2^{t-1} = -1\cdot 1 - 1\cdot 2 - 1\cdot 4 - 1\cdot 8 - \ldots - 1\cdot 2^{k-1} + k\cdot 2^{k} \quad \triangleright \text{eq. 2 - eq. 1}$$

$$\sum_{t=1}^{k} t2^{t-1} = -2^k + 1 + k\cdot 2^k \quad \triangleright \text{geometric series}$$

$$= 2^k(k-1) + 1$$

$$A_s(n) = \frac{1}{n}\sum_{t=1}^{k} t2^{t-1}$$

$$= \frac{(k-1)2^k + 1}{n}$$

$$= \frac{[\log_2(n+1) - 1](n+1) + 1}{n}$$

$$= \log_2(n+1) - 1 + \frac{\log_2(n+1)}{n}$$

Binary Search – Average Case Time Complexity

- The time complexity is

$$
\begin{aligned}
A_q(n) &= qA_s(n) + (1-q)A_f(n) \\
&= q\left[\log_2(n+1) - 1 + \frac{\log_2(n+1)}{n}\right] + (1-q)(\log_2(n+1)) \\
&= \log_2(n+1) - \mathrm{q} + \mathrm{q}\frac{\log_2(n+1)}{n} \\
&= \Theta(\log_2 n)
\end{aligned}
$$

- q is probability which is always ≤ 1
- $\frac{\log_2(n+1)}{n}$ is very small especially when n >> 1
- Binary search does approximately $\log_2(n+1)$ comparisons on average for n elements.

Binary Search – Another Implementation

```
def binary_search_recursive(arr, left, right, target):
    if left > right:
        return -1
    mid = left + (right - left) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return binary_search_recursive
    else:
        return binary_search_recursive
```

```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid  # target found at index mid
        elif arr[mid] < target:
            low = mid + 1  # search right half
        else:
            high = mid - 1  # search left half
    return -1  # target not found
```

Jump Search

```
def jump_search(arr, target):
    n = len(arr)
    step = int(math.sqrt(n))
    prev = 0

    while prev < n and arr[min(step, n) - 1] < target:
        prev = step
        step += int(math.sqrt(n))
        if prev >= n:
            return -1
    for i in range(prev, min(step, n)):
        if arr[i] == target:
            return i
    return -1
```

- When binary search is costly, e.g., searching for an element in a very large sorted dataset stored on a slow storage medium, like a database on disk or an external hard drive

Time Complexity of Jump Search

- Assume that the search key ***a*** is in the list

1. Best-case analysis: $\Theta\,(1)$
2. Worst-case analysis: $\Theta\,(\sqrt{n}) + \Theta(\sqrt{n}) = \Theta\,(\sqrt{n})$
3. Average-case analysis: $\sum_{i=1}^{\sqrt{n}} p_i\, \Theta(\sqrt{n}) = \sum_{i=1}^{\sqrt{n}} \frac{1}{\sqrt{n}} \Theta(\sqrt{n}) = \Theta(\sqrt{n})$

- Assume that the search key ***a*** is not in the list

$\Theta\,(\sqrt{n}) + \Theta(\sqrt{n}) = \Theta\,(\sqrt{n})$

- On average, the time complexity of Jump Search is $\Theta(\sqrt{n})$

Summary

- Exhaustive Algorithm: Sequential Search
  - Time complexity O(n)
- Decrease-and-conquer Algorithm:
  - Binary Search: Time complexity $O(\log_2 n)$
  - Jump Search: Time complexity $O(\sqrt{n})$

	Best Case	Average Case	Worst Case	Overall
Sequential	Θ (1)	Θ (n)	Θ (n)	O(n)
Binary	Θ (1)	Θ (logn)	Θ (logn)	O (logn)
Jump	Θ (1)	Θ ($\sqrt{n}$)	Θ ($\sqrt{n}$)	O ($\sqrt{n}$)