SC1007
Data Structures and Algorithms

Week 12: Revision

College of Computing and Data Science

Instructor: Luu Anh Tuan

Email: anhtuan.luu@ntu.edu.sg

Office: #N4-02c-86

What Is An Algorithm?

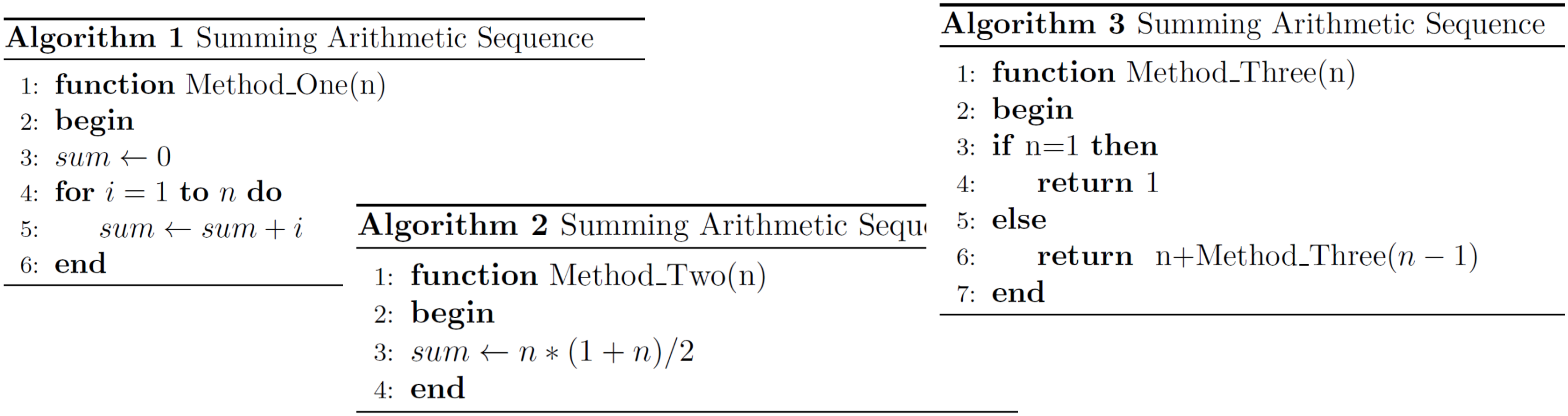
- An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

Introduction to Algorithms

-T. H. Cormen et. al.

Example : Arithmetic Series

- There are many ways (algorithms) to solve a problem
- Summing up 1 to n



Algorithm 1 Summing Arithmetic Sequence

```
function Method_One(n)
begin
$sum \leftarrow 0$
for $i = 1$ to $n$ do
    $sum \leftarrow sum + i$
end
```

Algorithm 2 Summing Arithmetic Sequ

```
function Method_Two(n)
begin
$sum \leftarrow n * (1 + n)/2$
end
```

Algorithm 3 Summing Arithmetic Sequence

```
function Method_Three(n)
begin
if n=1 then
    return 1
else
    return  n+Method_Three($n - 1$)
end
```

Analysis of Algorithms

- The study of the efficiency and performance of algorithms
- Evaluate the speed and scalability of an algorithm
  - How its efficiency changes as input sizes grow
- Identify the most efficient algorithms for a given problem
- Understand the trade-offs between different approaches

Time and space complexities

- Analyze efficiency of an algorithm in two aspects
    - Time
    - Space

- Time complexity: the amount of time used by an algorithm
- Space complexity: the amount of memory units used by an algorithm

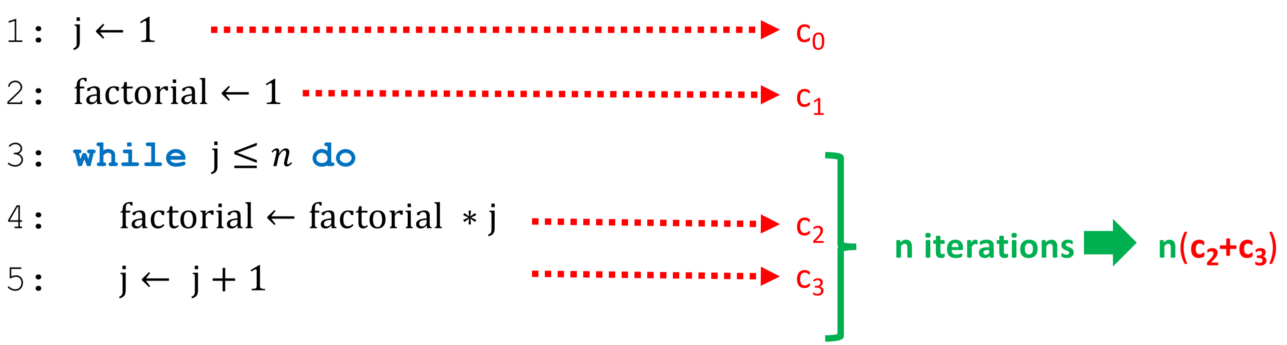
Time Complexity or Time Efficiency

1. Count the number of primitive operations in the algorithm

2. Express it in term of problem size

Time Complexity or Time Efficiency

i. Repetition Structure: for-loop, while-loop



```
j ← 1
factorial ← 1
while j ≤ $n$ do
    factorial ← factorial ∗ j
    j ← j + 1
```

$f(n) = c_0+c_1+n(c_2+c_3)$

The function increases linearly with n (problem size)

Common Complexity Classes

Order of Growth	Class	Example
1	Constant	Finding midpoint of an array
$\log_2 n$	Logarithmic	Binary Search
n	Linear	Linear Search
$n\log_2 n$	Linearithmic	Merge Sort
n^2	Quadratic	Insertion Sort
n^3	Cubic	Matrix Inversion (Gauss-Jordan Elimination)
2^n	Exponential	The Tower of Hanoi Problem
$n!$	Factorial	Travelling Salesman Problem

When time complexity of algorithm A grows faster than algorithm B for the same problem, we say A is inferior to B.

Asymptotic Notations

- Worst-case complexity: Big-Oh (**O**)
- Best-case complexity: Big-Omega (**Ω**)
- Average-case complexity: Big-Theta (**Θ**)

Space Complexity

- Determine number of entities in problem (also called problem size)
- Count number of basic units in algorithm

- Basic units
- Things that can be represented in a constant amount of storage space
- E.g. integer, float and character.

Space Complexity

- Space requirements for an array of n integers - Θ(n)
- If a matrix is used to store edge information of a graph,

  i.e. G[x][y] = 1 if there exists an edge from x to y,

  space requirement for a graph with n vertices is $\Theta(n^2)$

Space/time tradeoff principle

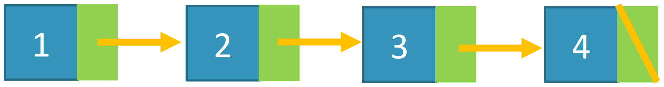
- Reduction in time can be achieved by sacrificing space and vice-versa.

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:

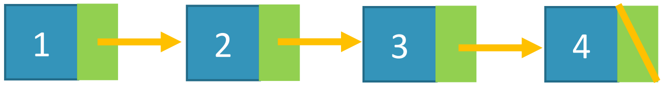
3. Average-case analysis:

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
```

pt = head → c_1

pt = pt.next → 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

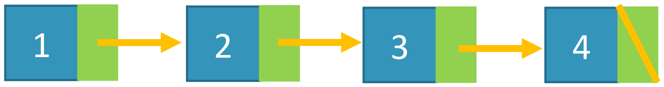
3. Average-case analysis $$\boldsymbol{p_1 \times time\ to\ search\ for\ item\ 1 + p_2 \times time\ to\ search\ for\ item\ 2 + \cdots + p_n \times time\ to\ search\ for\ item\ n}$$

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
```

pt = head → c_1

pt = pt.next → c_2 (n-1) iterations



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

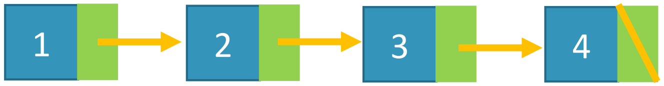
1. Best-case analysis: c_1 when **a** is the first item in the list => Θ (1)
2. Worst-case analysis: $c_2 \cdot (n-1) + c_1$ => Θ (n) when **a** is the last item in the list
3. Average-case analysis

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



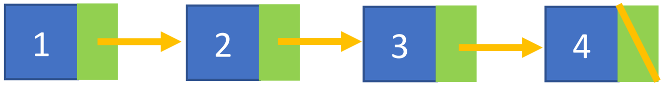
If the search key is in the list, on average: $\boldsymbol{c_1 + \frac{c_2(n-1)}{2}} = \Theta$ (n)

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

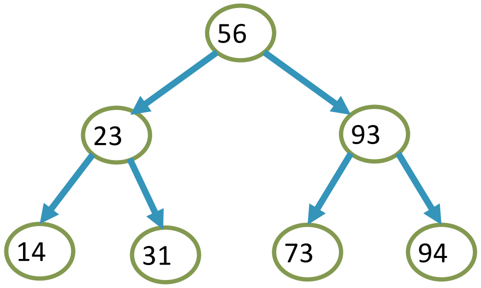
$$\boldsymbol{c_1 + nc_2} = \Theta \text{ (n)}$$

Devide and Conquer: Binary Search

- Given a sorted list



- Whether a search key ***a*** is in the list?
    - Given a sorted list, e.g.,
        - 14, 23, 31, 56, 73, 93, 94
    - We can build a BST

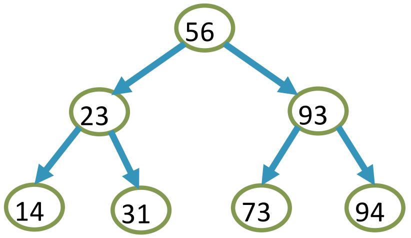


Time Complexity of Binary Search

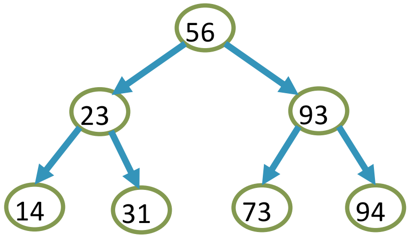
```
def binary_search_recursive(arr, left, right, target):
    if left > right:
        return -1
    mid = left + (right - left) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return binary_search_recursive(arr, mid + 1, right, target)
    else:
        return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

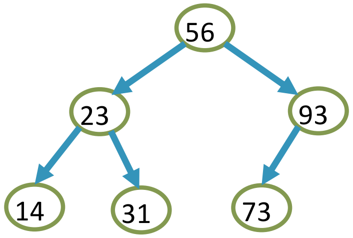
```
def binary_search(self, target, current_node):
    if current_node is None:
        return False
    elif target == current_node.data:
       return True
    elif target < current_node.data:
       return self.binary_search(target,current_node.left)
    else:
       return self.binary_search(target,current_node.right)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

- 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.

Height = $\lfloor \log_2 n \rfloor$

Binary Search – Worst Case Time Complexity

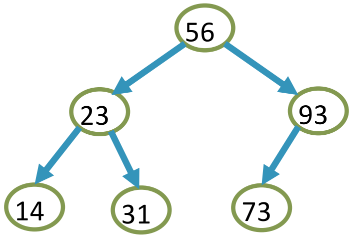
```
def binary_search(self, target, current_node):
    if current_node is None:
        return False
    elif target == current_node.data:
       return True
    elif target < current_node.data:
       return self.binary_search(target,current_node.left)
     else:
       return self.binary_search(target,current_node.right)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

f(n)

Constant c

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

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



- Assume a complete binary tree

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

$$= \Theta(\log_2 n)$$

Binary Search – Average Case Time Complexity

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

$$
\begin{aligned}
A(n) &= qA_s(n) + (1-q)A_f(n) \\
&= \Theta(\log_2 n)
\end{aligned}
$$

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

- 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 $\Theta(\sqrt{n})$

- 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 ($\log_2 n$)	Θ ($\log_2 n$)	O ($\log_2 n$)
Jump	Θ (1)	Θ $(\sqrt{n})$	Θ $(\sqrt{n})$	O $(\sqrt{n})$

Hashing

- Hashing: a typical space and time trade-off in algorithm
- To achieve search time in O(1), memory usage will be increased

What is hashing?

- To reduce the key space to a reasonable size
- Each key is mapped to a unique index (hash value/address)
- Search time remains O(1) on the average

hash function: {all possible keys} → {0, 1, 2, …, h-1}

- The array is called a hash table
- Each entry in the hash table is called a hash slot
- When multiple keys are mapped to the same hash value, a collision occurs
- If there are n records stored in a hash table with h slots, its load factor is $\alpha = \frac{n}{h}$

Hash Functions

- Must map all possible values within the range of the hash table uniquely
- Mapping should achieve an even distribution of the keys
- Easy and fast to compute
- Minimize collision

1. Modulo Arithmetic
2. Folding
3. Mid-square
4. Etc.

Hash Functions

1. Modulo Arithmetic: $\mathrm{H}(k) = k \bmod h$
   - E.g. *h = 13 & k = 37699 → H(k) = 37699 mod 13 = 12*
   - In practice, h should be a prime number, but not too close to any power of 2

2. Folding
   - Partition the key into several parts and combine the parts in a convenient way
   - Shift folding: Divide the key into a few parts and added up these parts
   - X = abc → H(X) = (a + b + c) mod h
   - E.g. *H(123456789) =(123 + 456 + 789) mod 13 = 3*

3. Mid-square
   - The key is squared and the middle part of the result is used as the hash address
   - E.g. *k=3121, $k^2 = 3121^2 = 9740641$ → H(k) = 406*

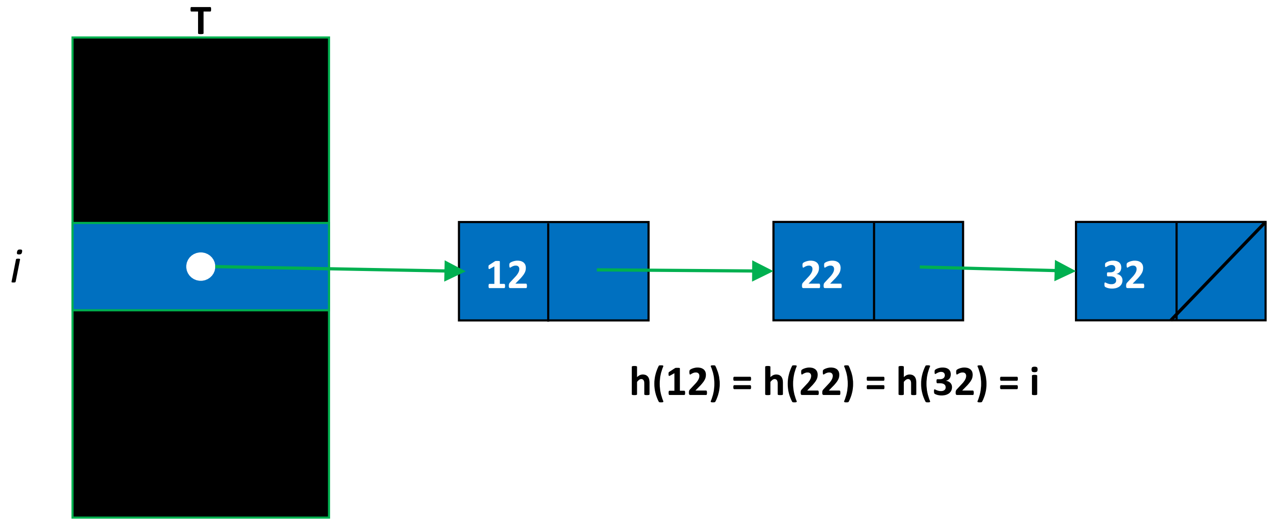
Collision Resolutions

- Closed Addressing Hashing – a.k.a separate chaining

- Open Addressing Hashing
  - Linear Probing
  - Quadratic Probing
  - Double Probing

Closed Addressing: Separate Chaining

- Keys are not stored in the table itself
- All the keys with the same hash address are store in a separate list



h(12) = h(22) = h(32) = i

- During searching, the searched element with hash address i is compared with elements in linked list H[i] sequentially
- In closed address hashing, there will be α number of elements in each linked list on average $\alpha = \frac{n}{h}$

Closed Addressing: Separate Chaining

Time complexity in the worst-case analysis: $\Theta(n)$

Time complexity in the average-case analysis: $\Theta(\alpha)$

Open Addressing

- Keys are stored in the table itself
- α cannot be greater than 1
- When collision occurs, probe is required for the alternate slot
  - Ideally, the probing approach can visit every possible slot

1. **Linear Probing**: probe the next slot

   $H(k, i) = (k + i) \bmod h$ where $i \in [0, h - 1]$

   Primary clustering:

   - A long runs of occupied slots
   - Average search time is increased

Open Addressing

2. Quadratic Probing

$H(k, i) = (k + c_1 i + c_2 i^2) \bmod h$ where c_1 and c_2 are constants, $c_2 \neq 0$

- Secondary Clustering: if two keys have the same initial probe position, their probe sequences will be the same. This will form a clustering.

Open Addressing

3. Double Hashing: a random probing method

$H(k, i) = (k + iD(k)) \bmod h$ where $i \in [0, h-1]$ and $D(k)$ is another hash function

Time Complexity

Linear Probing

- Successful Search: $\frac{1}{2}(1 + \frac{1}{1-\alpha})$
- Unsuccessful Search: $\frac{1}{2}(1 + \left(\frac{1}{1-\alpha}\right)^2)$

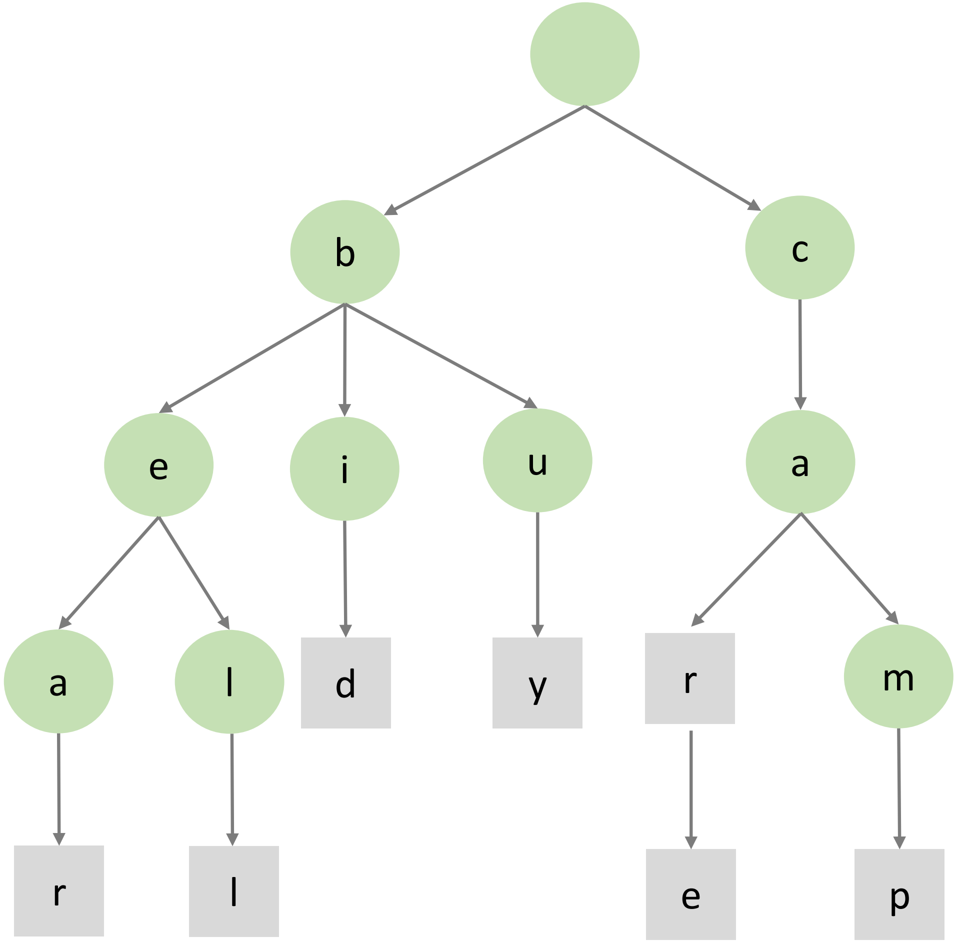
Double Hashing

- Successful Search: $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$
- Unsuccessful Search: $\frac{1}{1-\alpha}$

*Proof can be found in The Art of Computer Programming by Knuth Donald (1973)

What Is a Trie

- A tree-based data structure used for efficient string operations. Also called prefix tree or digital tree.
- It is a specialized search tree data structure used to store and retrieve strings from a dictionary or set.

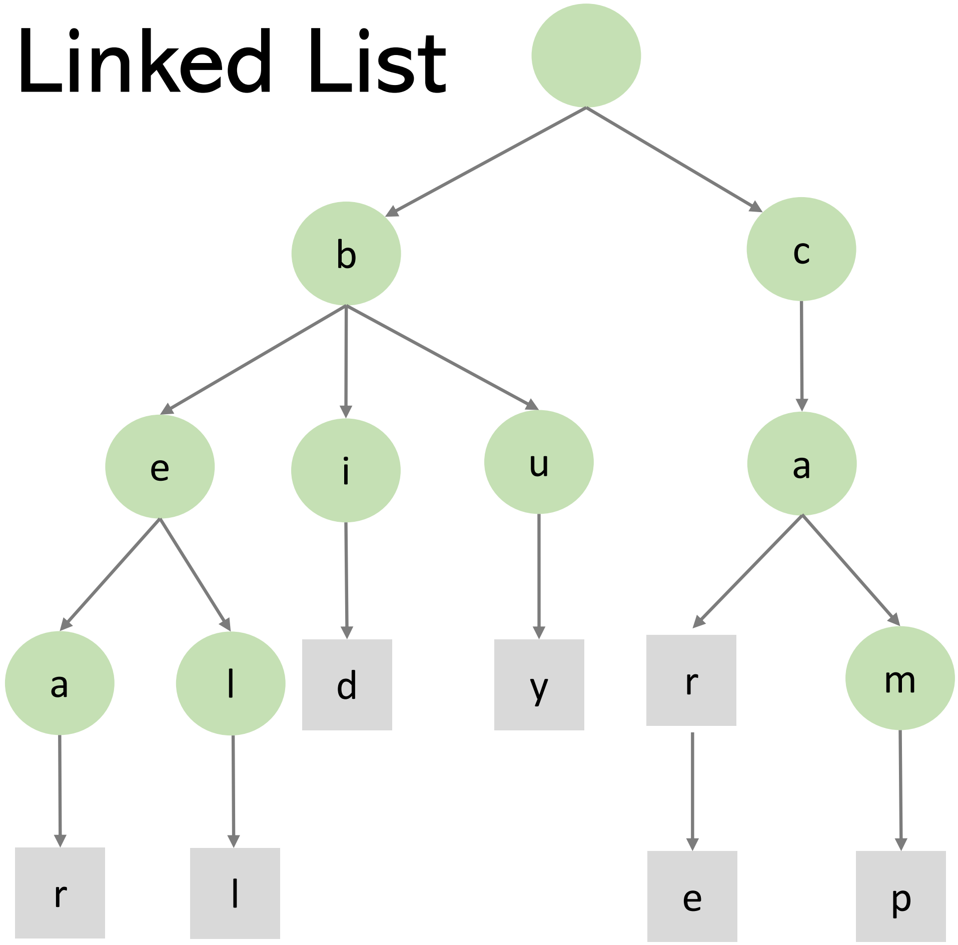


The trie structure for strings: bear, bell, bid, buy, car, care, camp

Implementations with Linked List

```
class TrieNode:
    def __init__(self, char):
        self.char = char
        self.is_end_of_word = False
        self.child = None
        self.next = None
```

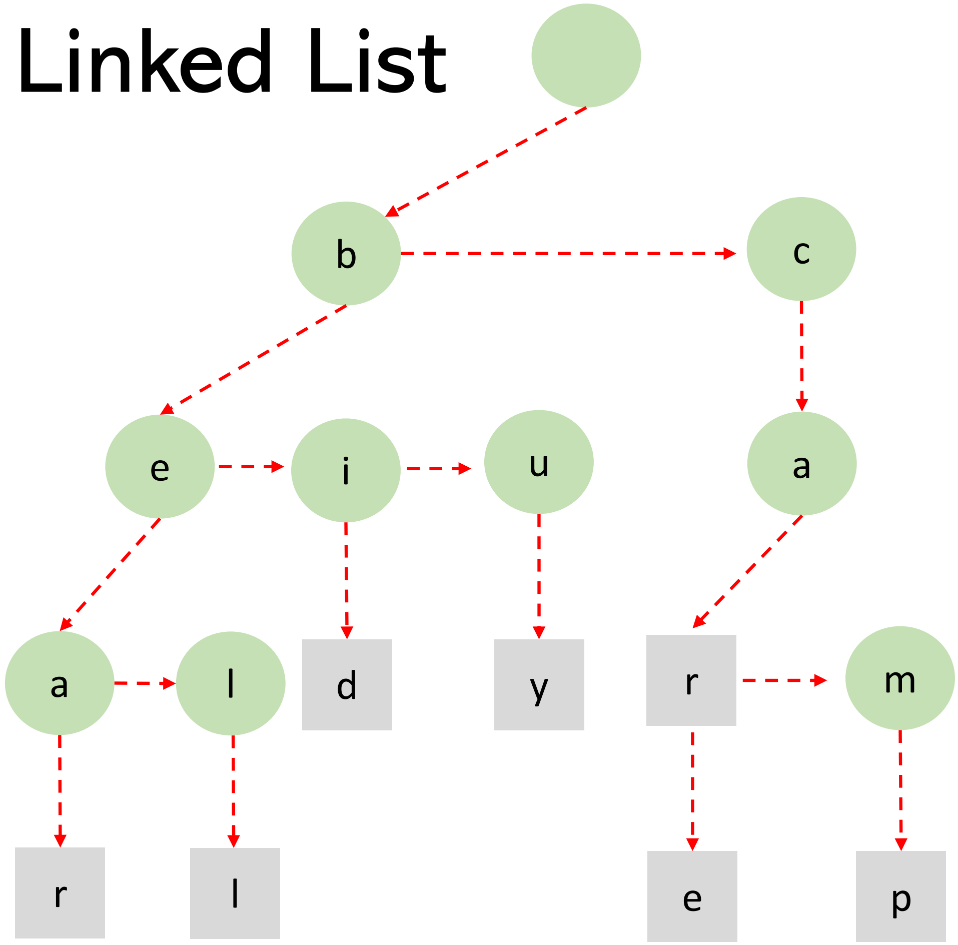
```
char = ‘a’
end_of_word = False
child = TrieNode(‘r’)
next = TrieNode(‘l’)
```



Implementations with Linked List

```
class TrieNode:
    def __init__(self, char):
        self.char = char
        self.is_end_of_word = False
        self.child = None
        self.next = None
```

```
char = ‘a’
end_of_word = False
child = TrieNode(‘r’)
next = TrieNode(‘l’)
```



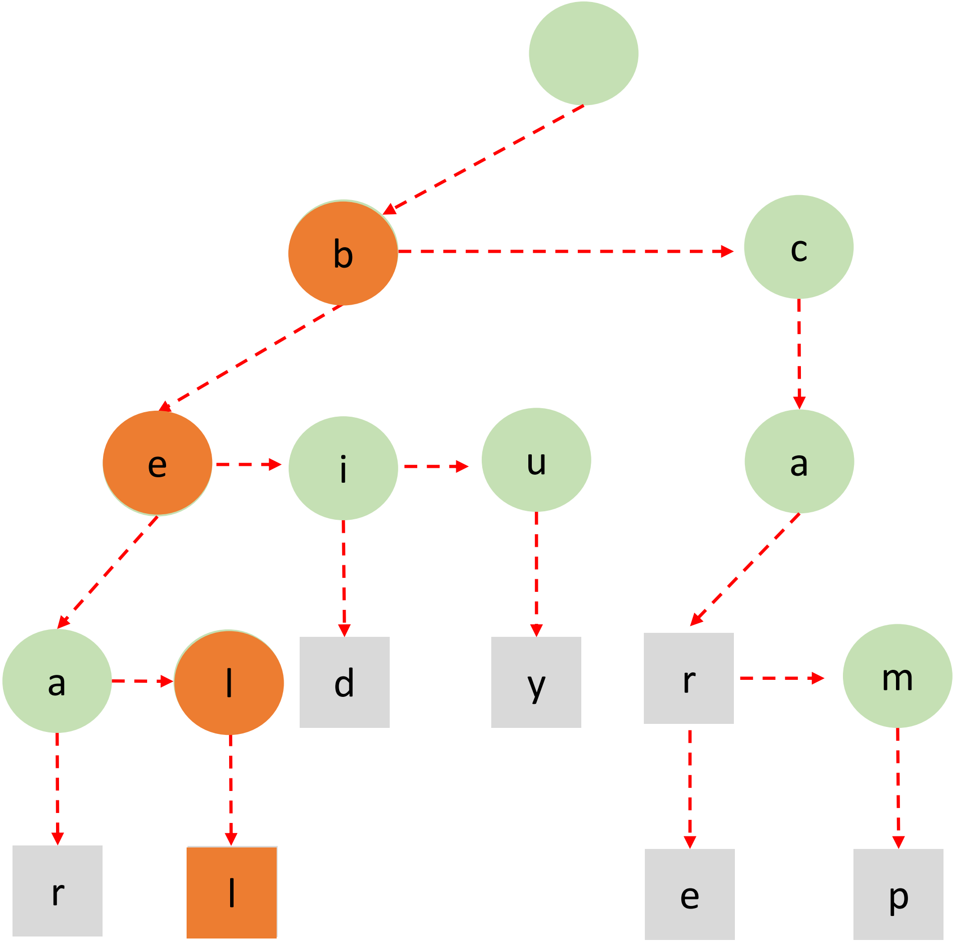
Implementations with Linked List

- The core operations for a trie:
  - Search a word
  - Insert a word
  - Traversal
- Usually we will not delete a word from a trie
  - Dictionaries don’t usually change
  - Deleting from a trie is much more complex than inserting
- The binary tree traversal algorithms can be applied in trie
  - Preorder (dfs)
  - Level-by-level (bfs)

```
class Trie:
    def __init__(self):
        self.root = TrieNode("")
    def search(self, word):
    def insert(self, word):
    def dfs(self,node):
    def bfs(self,node):
```

Search a Word

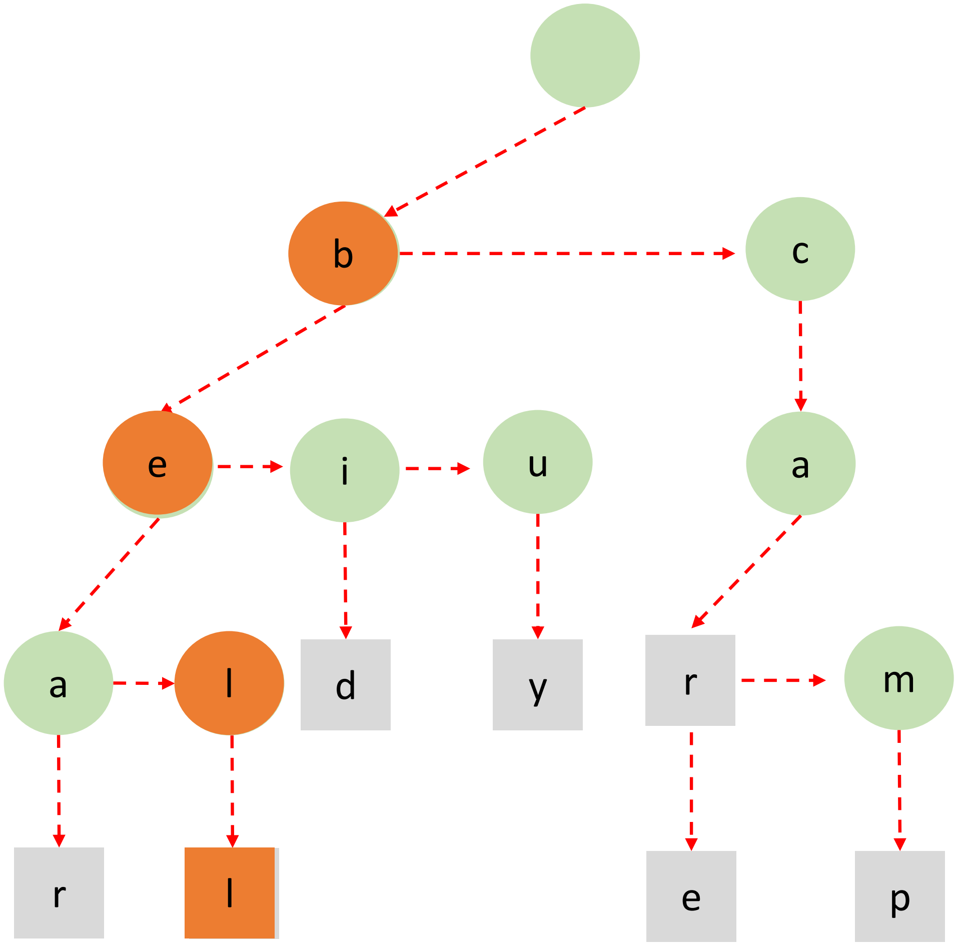
```
parent_node = root
for each character in the word
      if the current character is a child of parent_node:
            parent_node = current_node
            move on to the next character
      else:
            return False
return current_node.is_end_of_word
```



For example, search “bell”

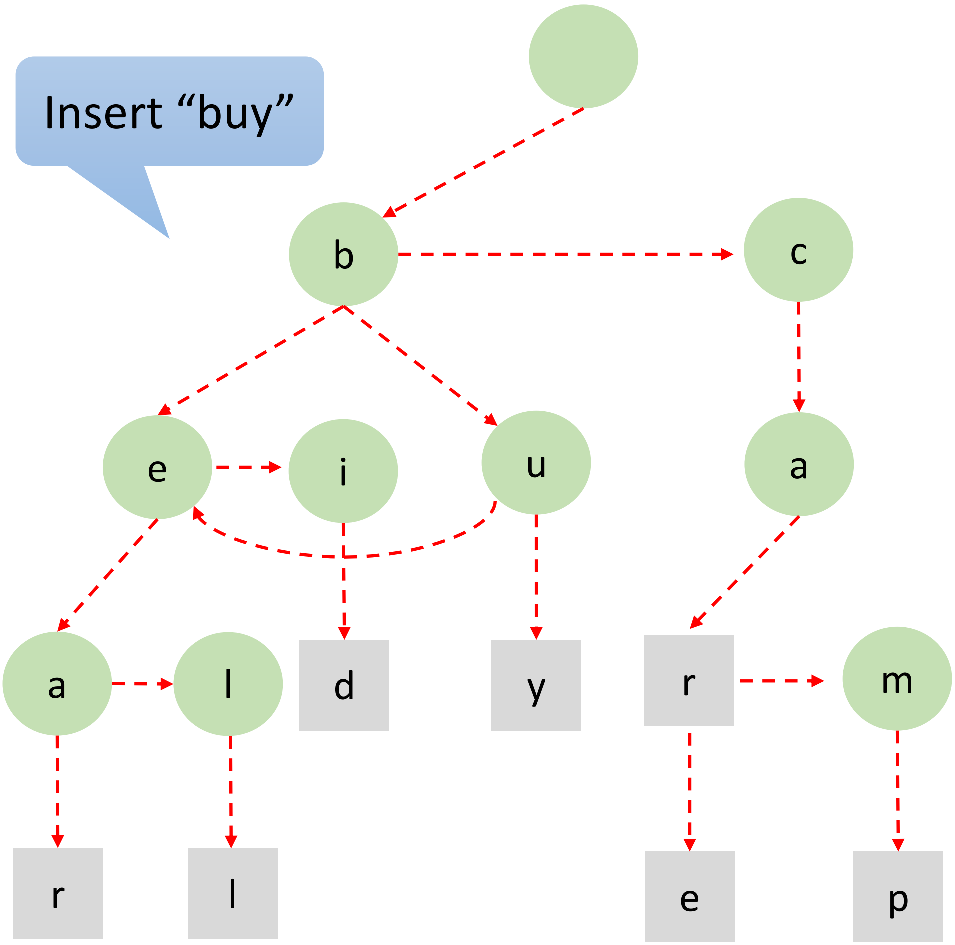
Search a Word

```
def _find_child(self, node, char):
    current = node.child
    while current:
        if current.char == char:
            return current
        current = current.next
    return None

def search(self, word):
    node = self.root
    for char in word:
        node = self._find_child(node, char)
        if not node:
            return False
    return node.is_end_of_word
```



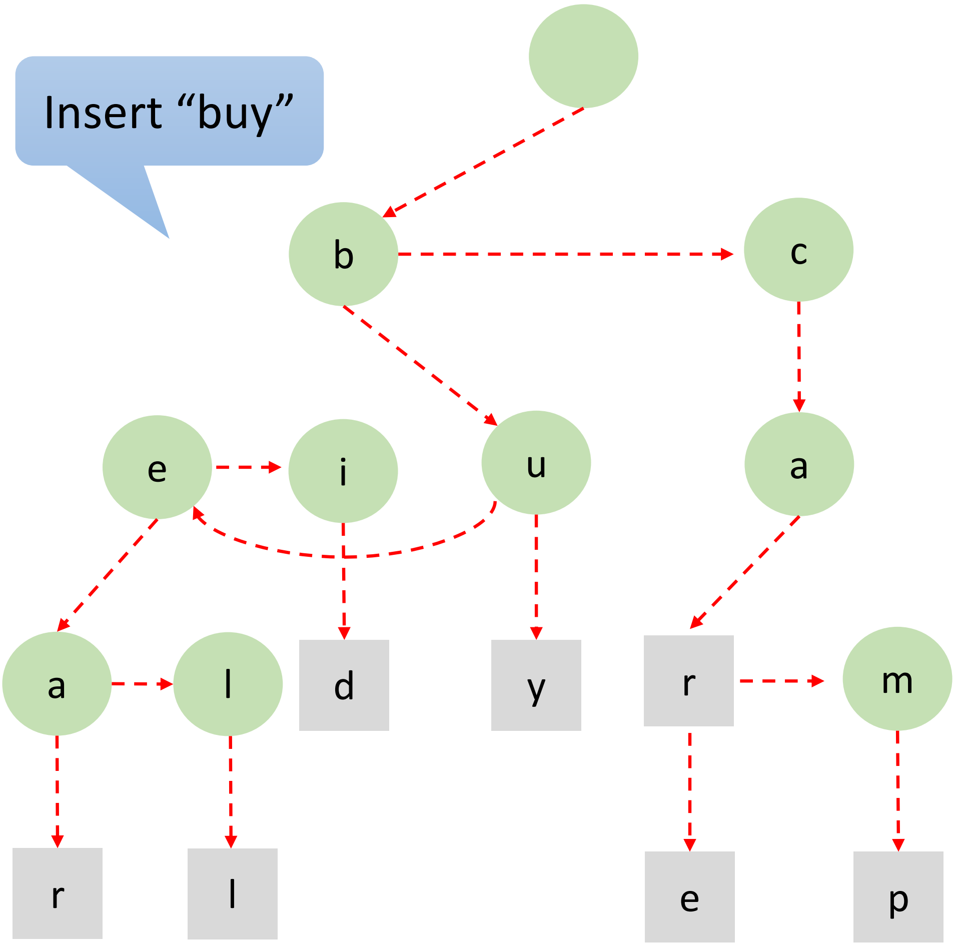
For example, search “bell”

Insert a Word



```
for each character in the word:
    if the character is a child node of
    the parent node:
        move to the next character
    else:
        new_node = create a new TrieNode
        #insert the child at the beginning
        #of the linked list
        set the new_node next be the
        parent_node's first child
        set the parent_node first child be
        the new_node

set end_of_word of the last_node as True
```

Insert a word



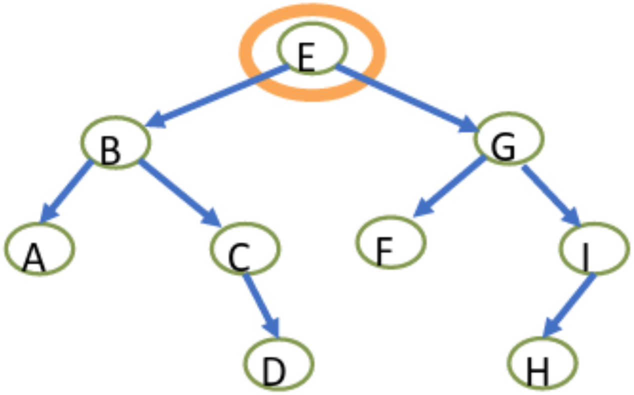
Insert “buy”

```
def _add_child(self, node, char):
     new_node = TrieNode(char)
     new_node.next = node.child
     node.child = new_node
     return new_node

def insert(self, word):
    node = self.root
    for char in word:
        child = self._find_child(node, char)
        if not child:
            child = self._add_child(node, char)
        node = child
     node.is_end_of_word = True
```

Lab Practice

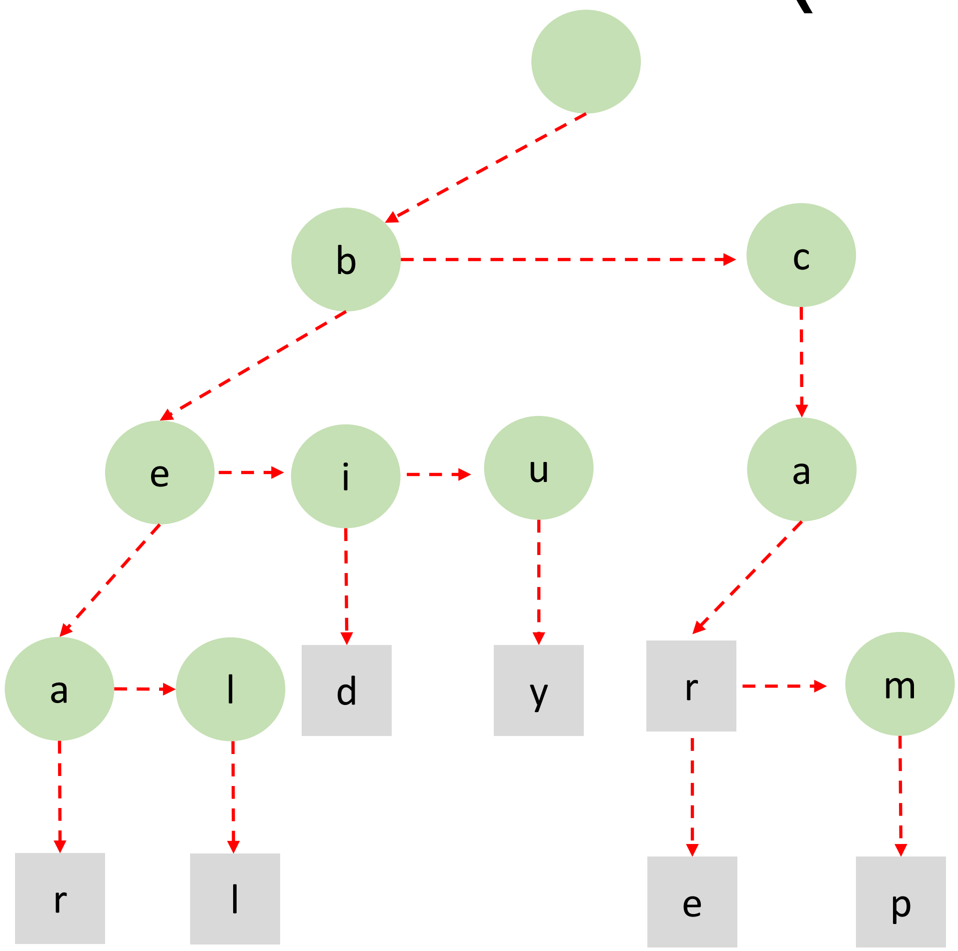
Pre-order Depth First Traversal

- Pre-order
  - **Process the current node's data**
  - **Visit the left child subtree**
  - **Visit the right child subtree**



```
TreeTraversal(Node N):
    Visit N;
    If (N has left child)
        TreeTraversal(LeftChild);
    If (N has right child)
        TreeTraversal(RightChild);
    Return; // return to parent
```

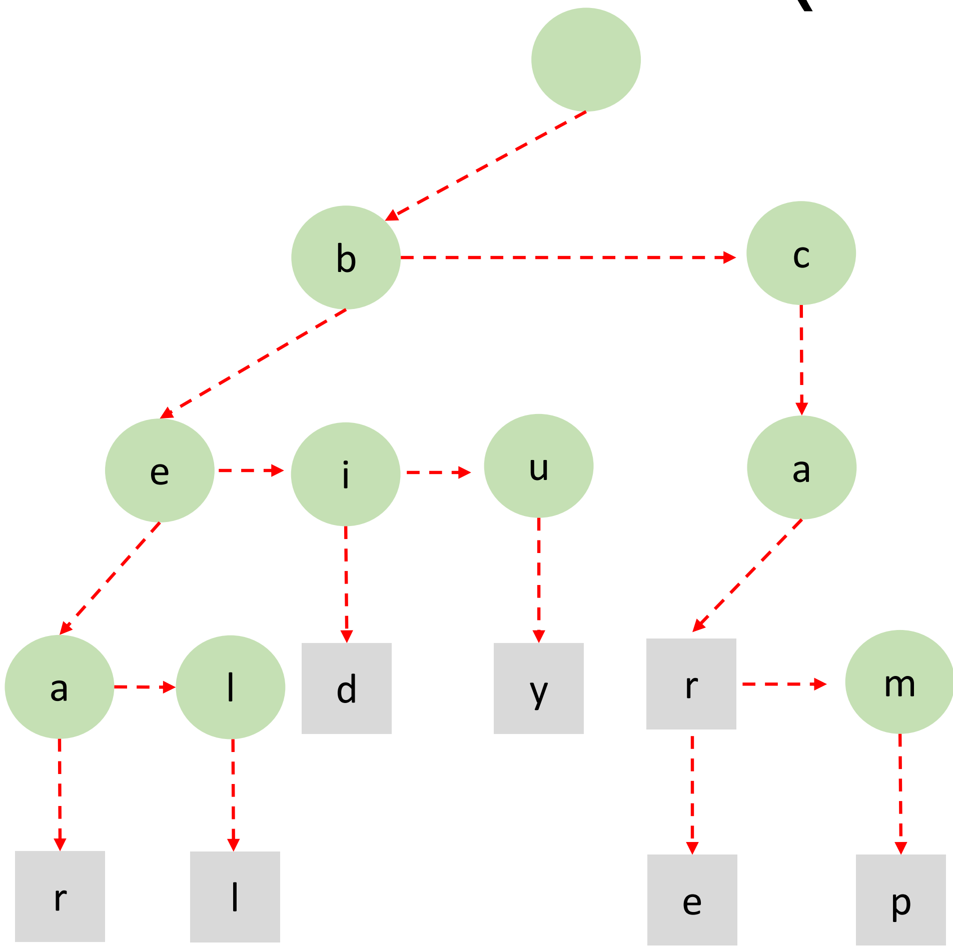
Preorder Traversal (DFS)



Instead of visiting left and right children, visit each child of the TrieNode

```
dfs(TrieNode tn):
    visit tn
    child = tn.child
    while child is not None:
        dfs(child)
        child = child.next
```

Preorder Traversal (DFS)



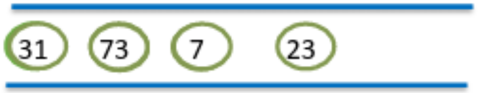
```
def dfs(self, node):
    if node is not None:
        print(node.char, end=" ")
    child = node.child
    while child:
        self.dfs(child)
        child = child.next
```

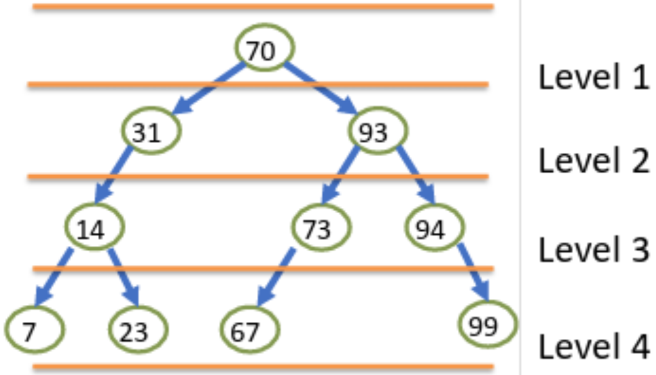
```
None b e a r l l l d u y c a r e m p
```

Breath-first Traversal: Level-by-level

Level-By-Level Traversal:

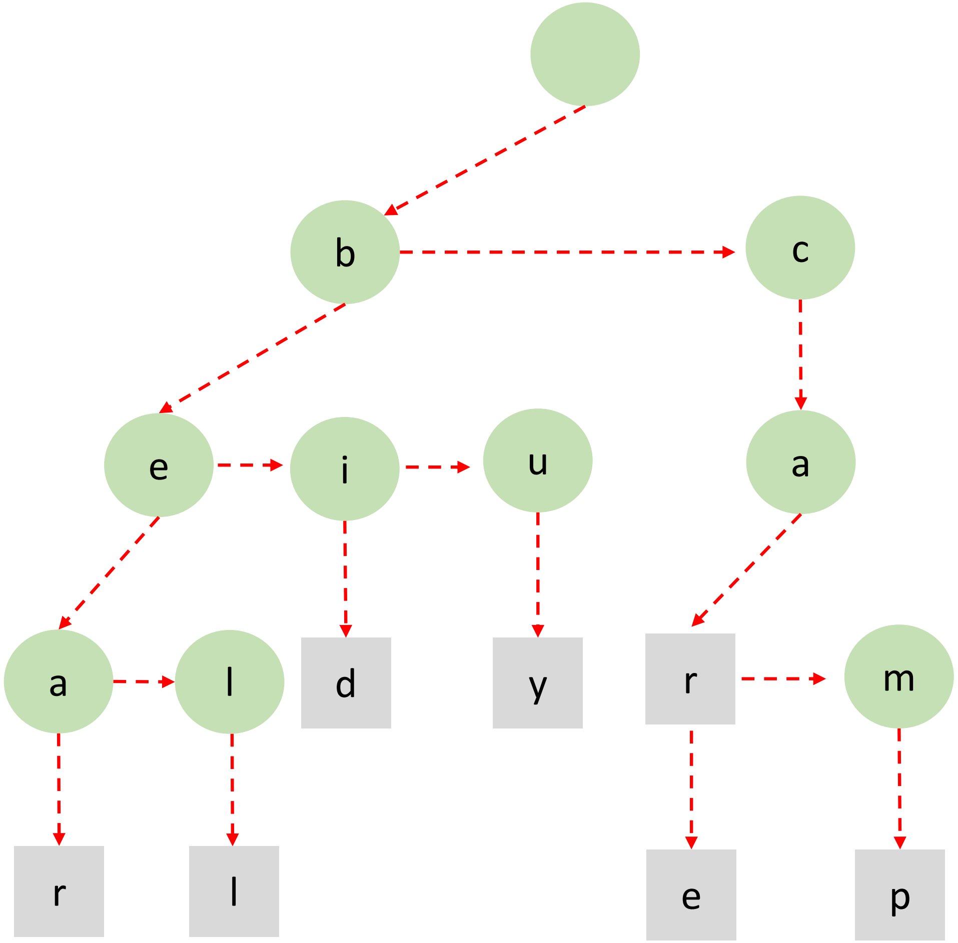
- Visiting a node
- Remember all its children
    - Use a queue (FIFO structure)





1. Enqueue the current node
2. Dequeue a node
3. Enqueue its children if it is available
4. Repeat Step 2 until the queue is empty

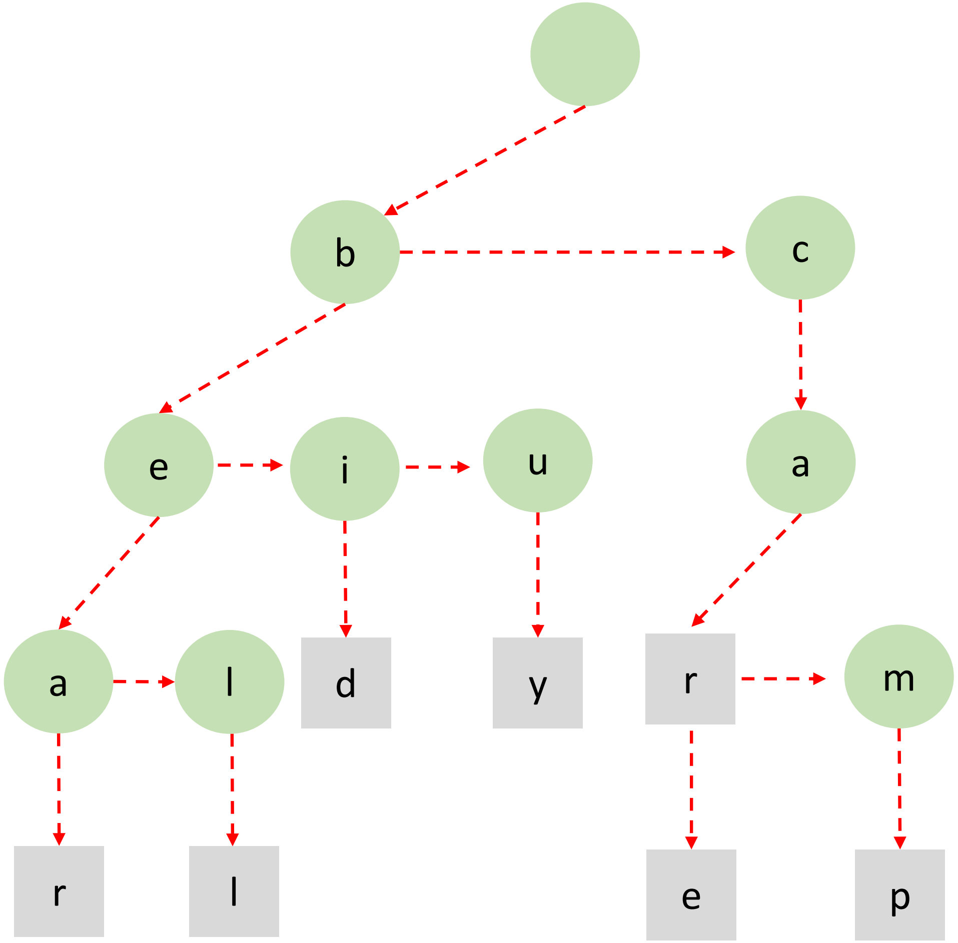
Level-by-Level Traversal (BFS)



```
def bfs(self):
    queue = Queue()
    queue.enqueue(self.root)
    while not queue.is_empty():
        node = queue.dequeue()
        print(node.char, end=" ")
        child = node.child
        while child:
            queue.enqueue(child)
            child = child.next
```

```
None b c e i u a a l d y r m r l e p
```

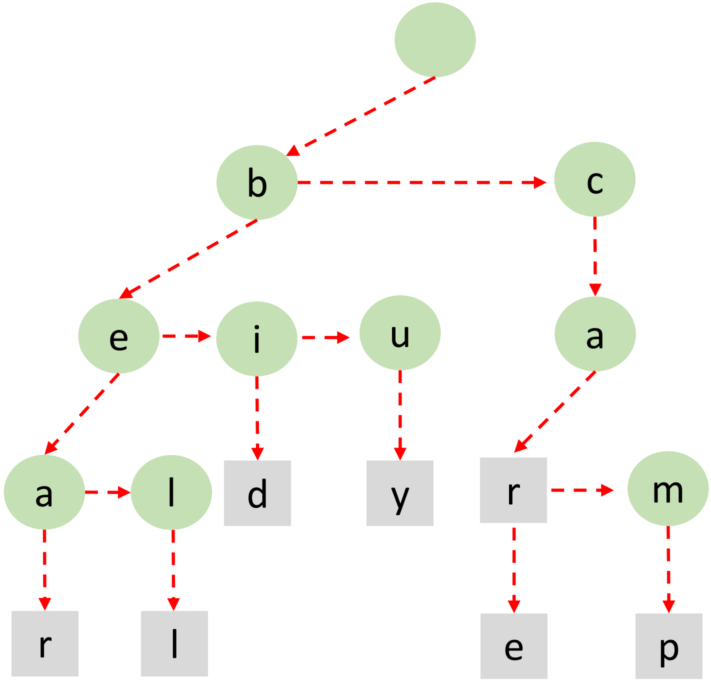
Working Example: Print All Words

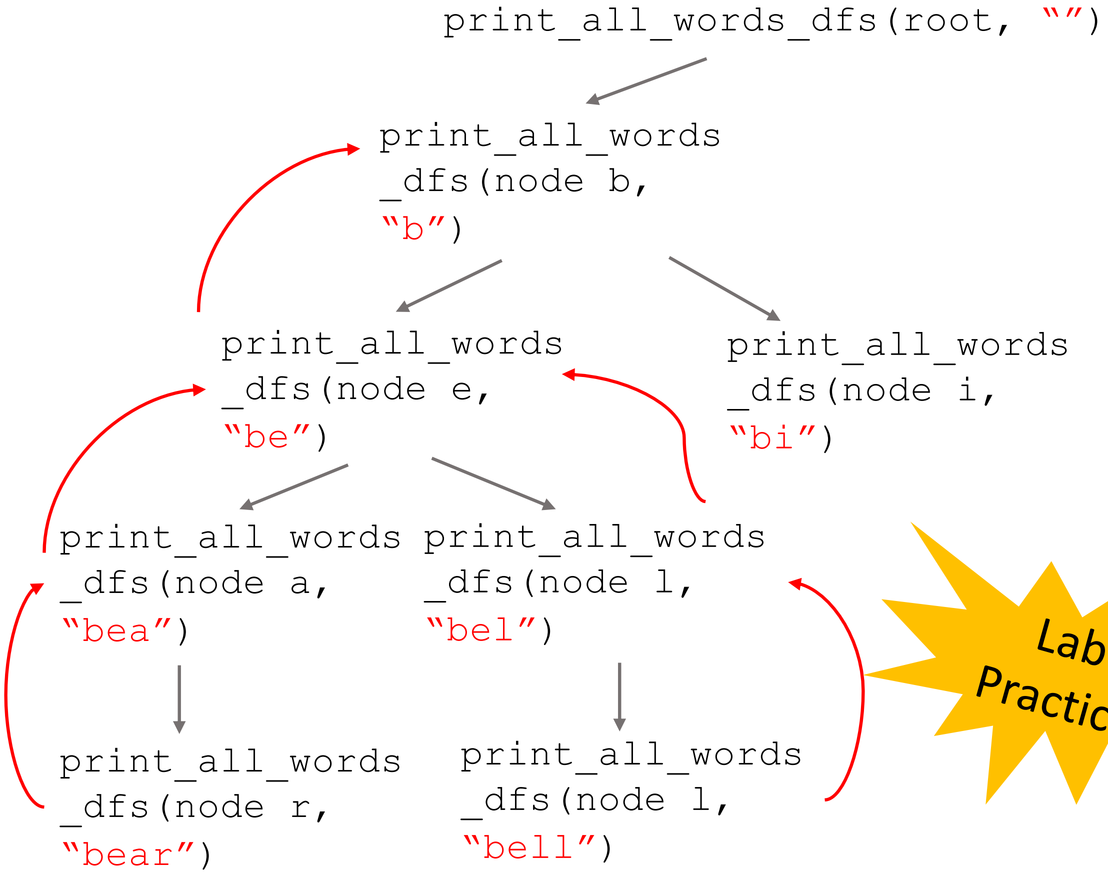


- Apply dfs
- When the node is the end of a word, print it
- Keep track of current nodes’ ancestors

```
def print_all_words_dfs(self, node, prefix):
    if node.is_end_of_word:
        print(prefix)

    child = node.child
    while child:
        self. print_all_words(child,
                              prefix+child.char)
        child = child.next
```

Working Example: Print All Words

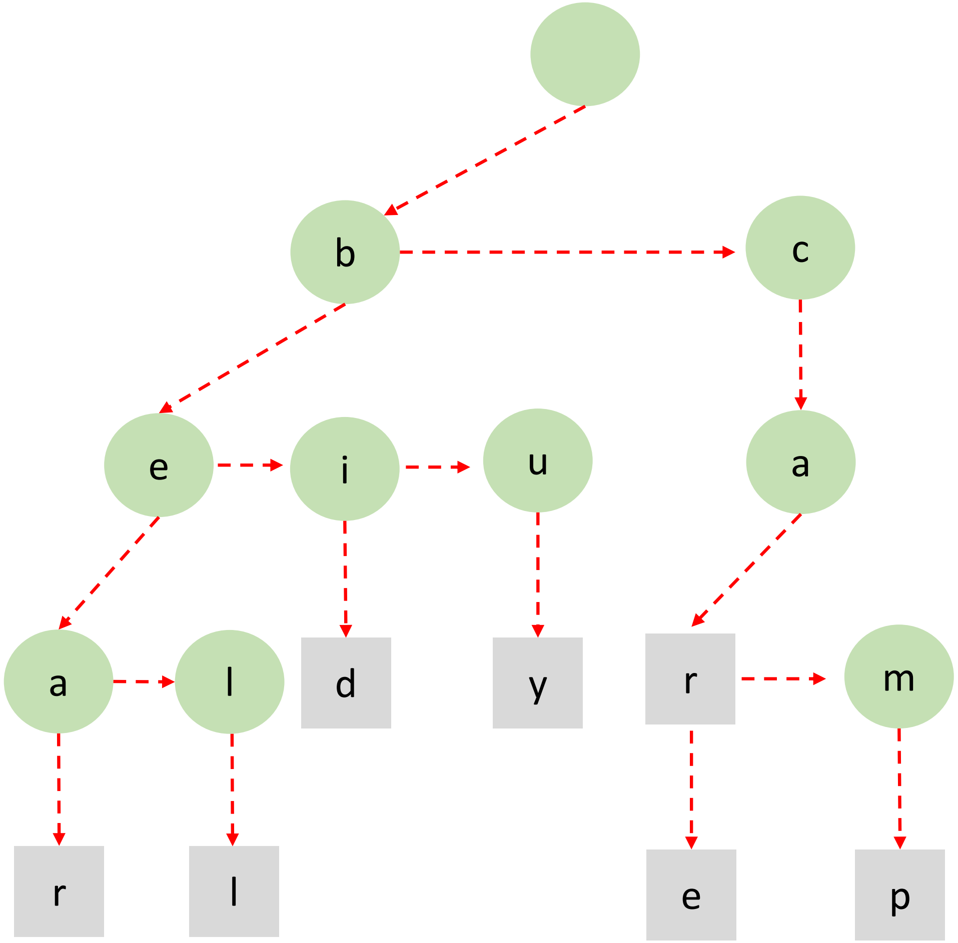




```
print_all_words_dfs(root, "")

print_all_words
_dfs(node b,
"b")

print_all_words
_dfs(node e,
"be")

print_all_words
_dfs(node i,
"bi")

print_all_words
_dfs(node a,
"bea")

print_all_words
_dfs(node l,
"bel")

print_all_words
_dfs(node r,
"bear")

print_all_words
_dfs(node l,
"bell")
```

Lab Practice

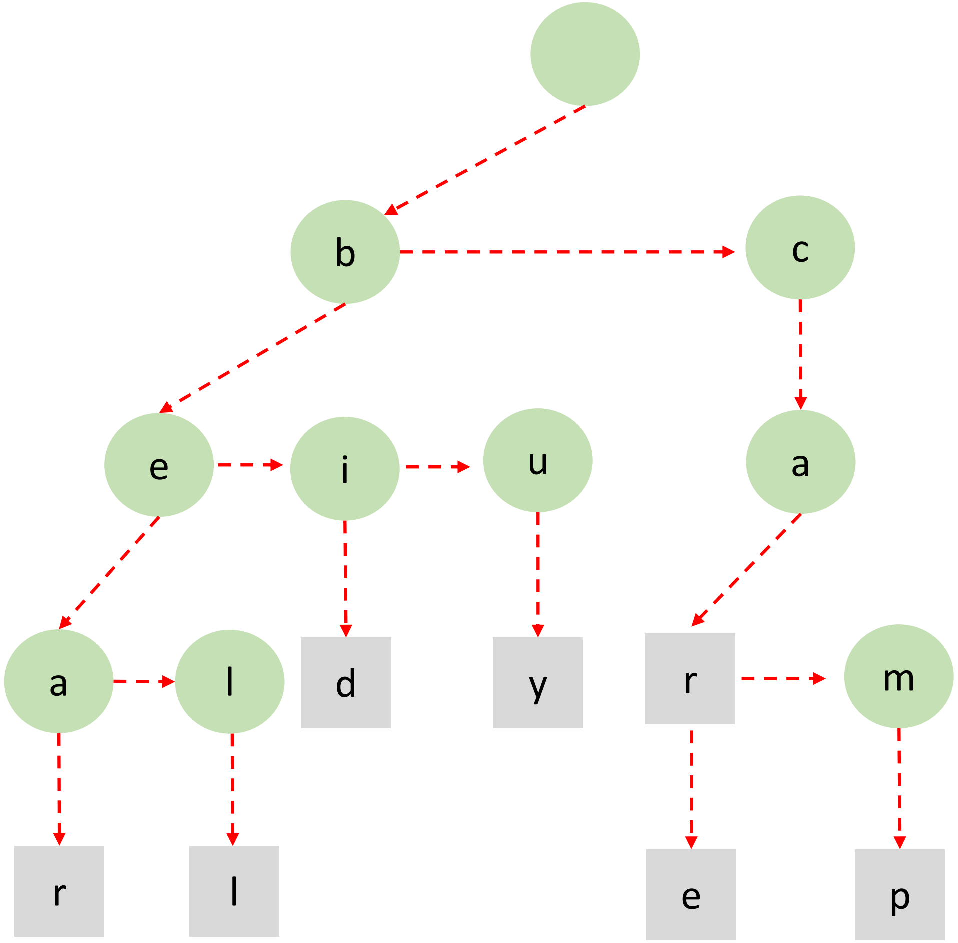
Working Example: Print All Words



- Apply bfs
- When enqueue a node, also enqueue the node's ancestors & the node's character
- When dequeue a node, if the node is end of a word, print the word

```
class Queue:
    def __init__(self):
        self.items = []
    def enqueue(self, item):
        self.items.append(item)
    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        return None
    def is_empty(self):
        return len(self.items) == 0
```

Working Example: Print All Words



```
def print_all_words_bfs(self):
    queue = Queue()
    queue.enqueue((self.root, ""))
    while not queue.is_empty():
        node, prefix = queue.dequeue()
        if node.is_end_of_word:
            print(prefix)
        child = node.child
        while child:
            queue.enqueue((child,
                           prefix + child.char))
            child = child.next
```

Tutorial Practice