

Singly Linked List

Queues

# Data Structures CHEAT SHEET

SCALER  
*Topics*

Time Complexity

Arrays

Arrays											
<b>Description</b> An array is a collection of elements identified by index or key values. The elements are stored contiguously in memory.	<b>Time Complexity</b>  "n" represents the number of elements in the array. <table> <tr> <th>Operation</th><th>Time Complexity</th></tr> <tr> <td>Access</td><td><math>O(1)</math></td></tr> <tr> <td>Search</td><td><math>O(n)</math></td></tr> <tr> <td>Insertion</td><td><math>O(n)</math></td></tr> <tr> <td>Deletion</td><td><math>O(n)</math></td></tr> </table>	Operation	Time Complexity	Access	$O(1)$	Search	$O(n)$	Insertion	$O(n)$	Deletion	$O(n)$
Operation	Time Complexity										
Access	$O(1)$										
Search	$O(n)$										
Insertion	$O(n)$										
Deletion	$O(n)$										
<b>Visualization</b> <pre> arr[0] arr[1] arr[2] arr[3] arr[4] arr[5] arr[6]   8   10   12   7   19   4   8 Array Indices → 0    1    2    3    4    5    6           </pre>											
<b>Space Complexity</b>  $O(n)$	<b>Notable Usages</b>  Efficient random access. Contiguous memory allocation.										

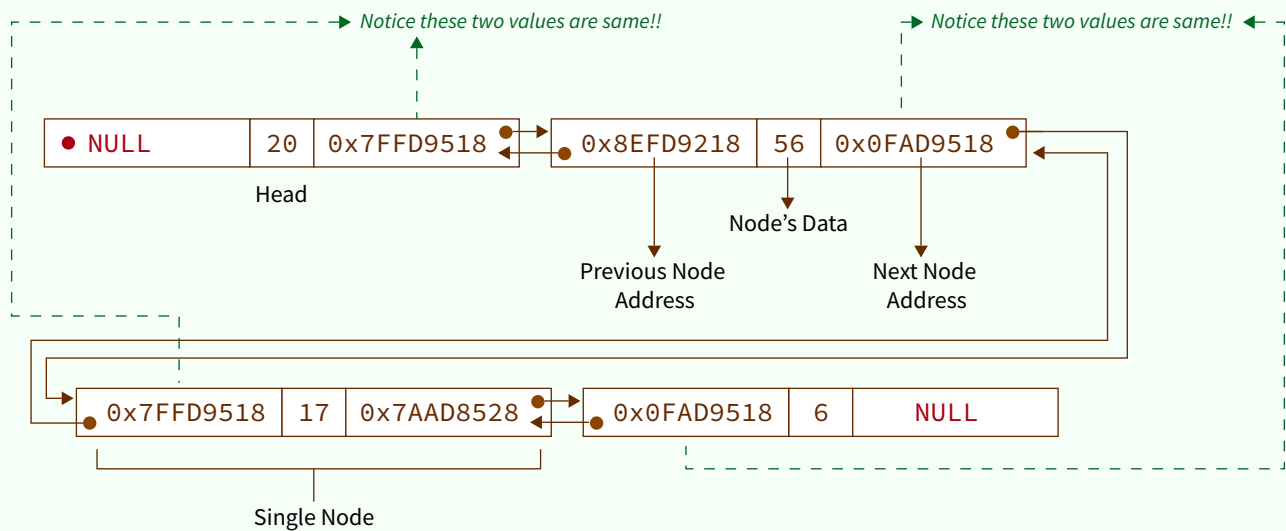
Singly Linked List											
<b>Description</b> A singly linked list contains nodes which have a data part and an address part, i.e., next, which points to the next node in the order of sequence of nodes.											
<b>Visualization</b> <pre> graph LR     Head[20   0x7FFD9518] --&gt; Node2[34   0x8EFD9218]     Node2 --&gt; Node3[8   0x0FAD9518]     Node3 --&gt; Node4[12   0x7AAD8528]     Node4 --&gt; End[98   NULL]     </pre>											
<b>Time Complexity</b>  "n" represents the number of nodes in the Singly Linked List. <table> <tr> <th>Operation</th><th>Time Complexity</th></tr> <tr> <td>Access</td><td><math>O(n)</math></td></tr> <tr> <td>Search</td><td><math>O(n)</math></td></tr> <tr> <td>Insertion</td><td><math>O(1)</math></td></tr> <tr> <td>Deletion</td><td><math>O(1)</math></td></tr> </table>	Operation	Time Complexity	Access	$O(n)$	Search	$O(n)$	Insertion	$O(1)$	Deletion	$O(1)$	<b>Space Complexity</b>  $O(n)$
Operation	Time Complexity										
Access	$O(n)$										
Search	$O(n)$										
Insertion	$O(1)$										
Deletion	$O(1)$										
	<b>Notable Usages</b>  Dynamic memory allocation. Efficient insertion and deletion at the beginning.										

# Doubly Linked List

## Description

In a doubly linked list, each node contains data and two links, the first link points to the previous node and the next link points to the next node in the sequence.

## Visualization



## Time Complexity

"n" represents the number of nodes in the Doubly Linked List.

Operation	Time Complexity
Access	$O(n)$
Search	$O(n)$
Insertion	$O(1)$
Deletion	$O(1)$

## Space Complexity

$O(n)$

## Notable Usages

Bi-directional traversal.  
Efficient deletion of a node.

## Stacks

### Description

A stack is a linear data structure that follows the LIFO (Last In First Out) principle. It has two main operations: push and pop.

### Time Complexity

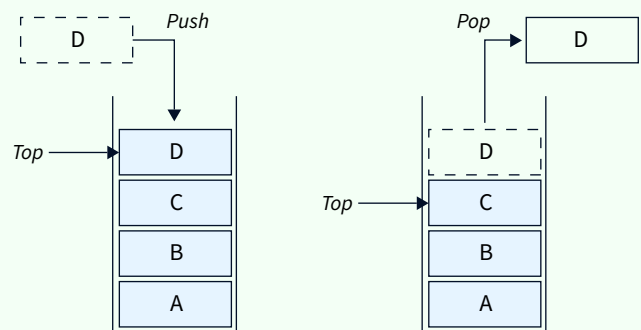
"n" represents the number of elements in the Stack.

Operation	Time Complexity
Access	$O(n)$
Search	$O(n)$
Insertion	$O(1)$
Deletion	$O(1)$

### Space Complexity

$O(n)$

### Visualization



### Notable Usages

Function call management.  
Undo/Redo operations.

## Queues

### Description

A queue is a linear data structure that follows the FIFO (First In First Out) principle. It mainly has two operations: enqueue and dequeue.

### Time Complexity

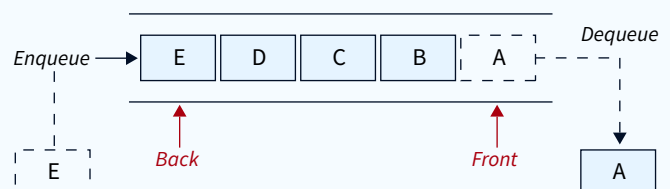
"n" represents the number of elements in the Queue.

Operation	Time Complexity
Access	$O(n)$
Search	$O(n)$
Insertion	$O(1)$
Deletion	$O(1)$

### Space Complexity

$O(n)$

### Visualization



### Notable Usages

Job scheduling.  
Breadth-first search.

## Binary Tree

### Description

A binary tree is a tree-type non-linear data structure with a maximum of two children for each parent.

### Time Complexity

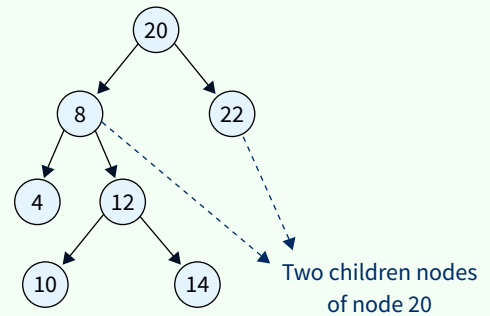
"n" represents the number of nodes in the binary tree.

Operation	Time Complexity
Access	$O(n)$
Search	$O(n)$
Insertion	$O(n)$
Deletion	$O(n)$

### Space Complexity

$O(n)$

### Visualization



### Notable Usages

Hierarchical data representation.  
Quick searching and sorting.

## Binary Search Tree

### Description

A binary search tree, also known as an ordered or sorted binary tree, is a rooted binary tree whose internal nodes each store a key greater than all the keys in the node's left subtree and less than those in its right subtree.

### Worst Case Time Complexity

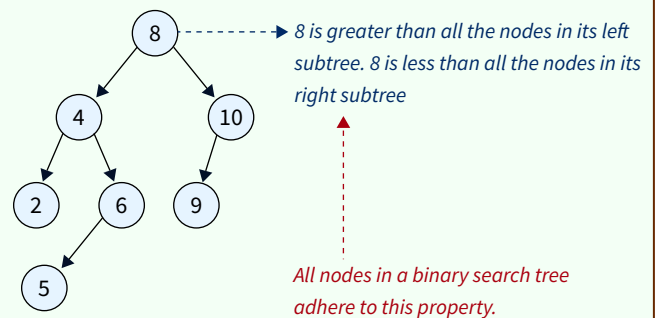
"n" represents the number of nodes in the binary search tree.

Operation	Worst Case Time Complexity
Access	$O(n)$
Search	$O(n)$
Insertion	$O(n)$
Deletion	$O(n)$

### Space Complexity

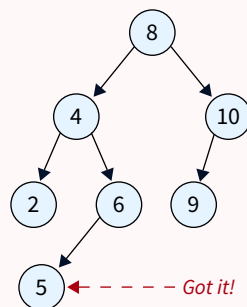
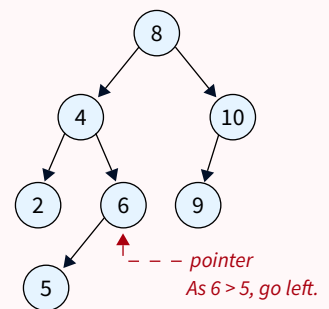
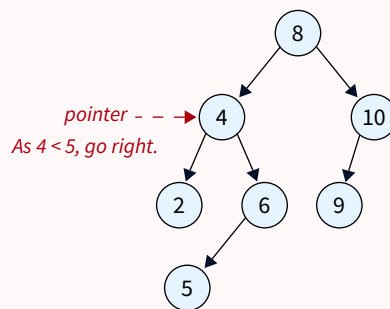
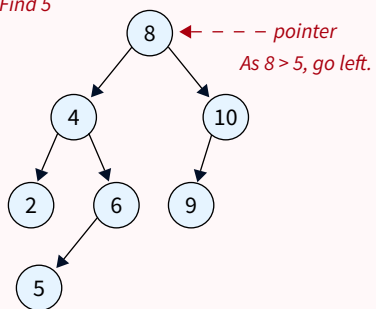
$O(n)$

### Visualization



## Demonstration of Search operation

Find 5



### Notable Usages

Efficient searching.  
In-order traversal.

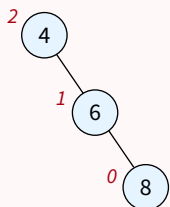
# AVL Tree

## Description

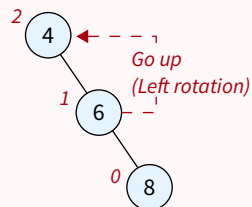
An AVL tree is a self-balancing binary search tree. For every node, the difference between the heights of the left and right subtrees is at most one.

## Visualization

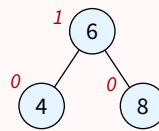
### Concept of Left Rotation



Unbalanced tree  
due to node 4

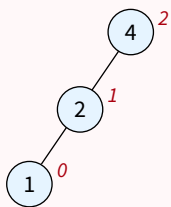


Left Rotation

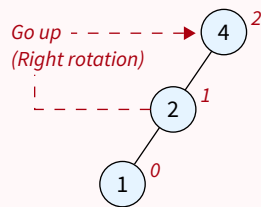


Balanced Tree

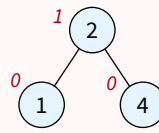
### Concept of Right Rotation



Unbalanced tree  
due to node 4

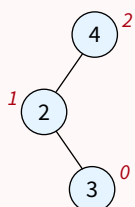


Right Rotation

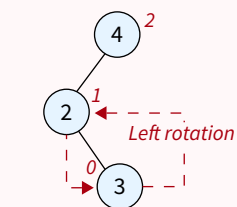


Balanced Tree

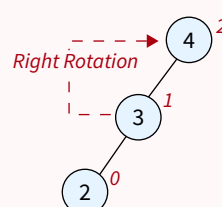
### Concept of Left-Right Rotation



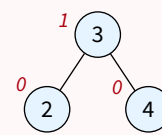
Unbalanced tree  
due to node 4



Left Rotation

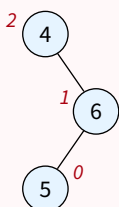


Right Rotation

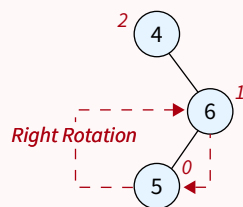


Balanced Tree

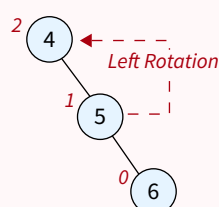
### Concept of Right-Left Rotation



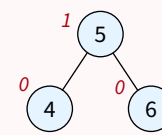
Unbalanced tree  
due to node 4



Right Rotation



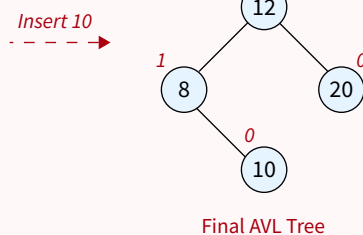
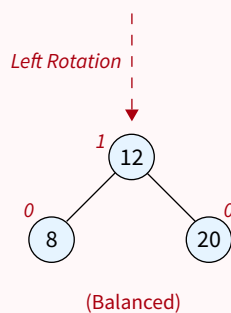
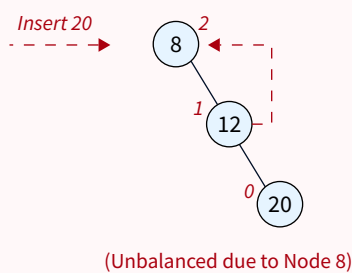
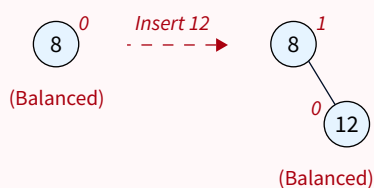
Left Rotation



Balanced Tree

Let's understand insertion in AVL tree: Create an AVL tree using 8, 12, 20, 10.

Insert 8



## Time Complexity

"n" represents the number of nodes in the AVL tree.

Operation	Time Complexity
Access	$O(\log n)$
Search	$O(\log n)$
Insertion	$O(\log n)$
Deletion	$O(\log n)$

## Notable Usages

Self-balancing property.  
Efficient searching and insertion.

## Space Complexity

$O(n)$



# HashTable

## Description

A hash table, also known as a hash map, is a data structure that provides efficient key-value pair storage and retrieval. It uses a hash function to compute an index, called a hash code, which maps the key to a specific location in the underlying array.

## Visualization of Chained Hash Table

HashTable-Size = 5

HashTable function:  $h(\text{key}) = \text{key} \% 5$

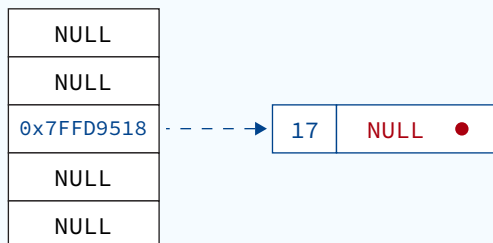
NULL
NULL
NULL
NULL
NULL

Let's insert the following keys into the hashtable.

17  
43  
2  
34  
43  
98  
10

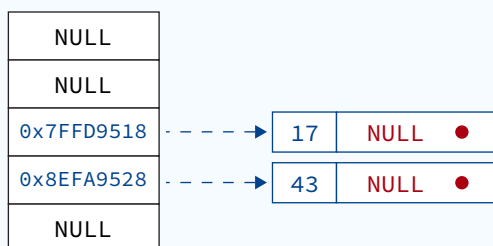
Insert 17

$h(17) = 17 \% 5 = 2$  -----> Insert it into index 2



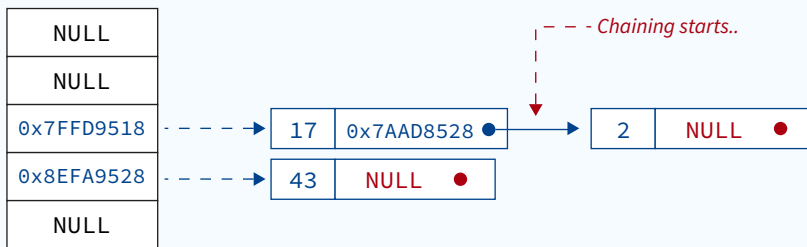
Insert 43

$h(43) = 43 \% 5 = 3$  -----> Insert it into index 3



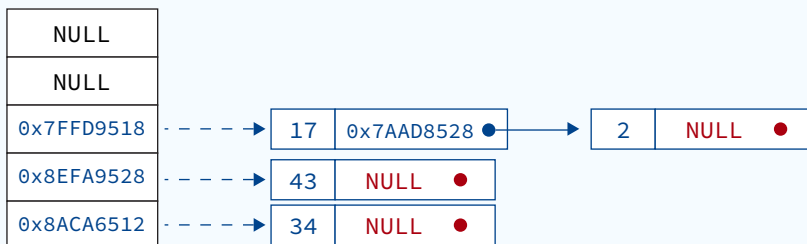
### Insert 2

$h(2) = 2 \% 5 = 2$  -----> Insert it into index 2



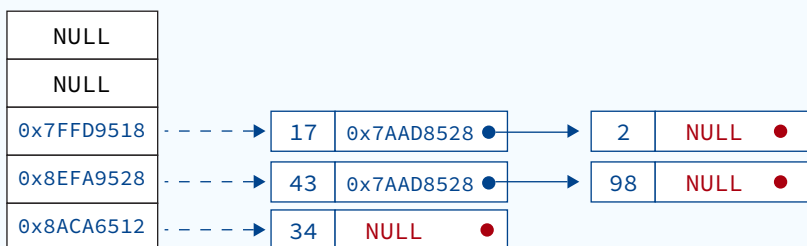
### Insert 34

$h(34) = 34 \% 5 = 4$  -----> Insert it into index 4



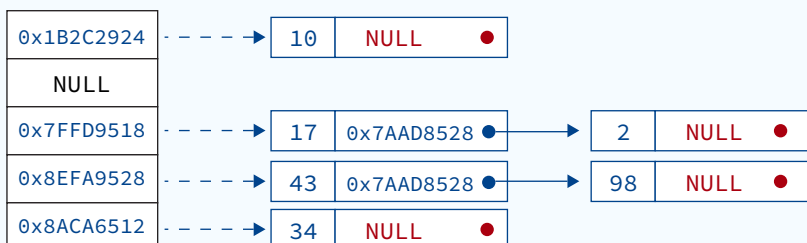
### Insert 98

$h(98) = 98 \% 5 = 3$  -----> Insert it into index 3



### Insert 10

$h(10) = 10 \% 5 = 0$  -----> Insert it into index 0



## Time Complexity

"n" represents the size of hash table and "k" represents the average number of elements in each chain (linked list).

Operation	Best Case Time Complexity	Average Case Time Complexity	Worst Case Time Complexity
Search	$O(1)$	$O(1+k)$	$O(n)$
Insertion	$O(1)$	$O(1+k)$	$O(n)$
Deletion	$O(1)$	$O(1+k)$	$O(n)$

## Space Complexity

$O(n + m)$  where  $n$  is the size of hashtable and  $m$  is the number of elements inserted.

## Notable Usages

Fast data retrieval.  
Key-value storage.

# Trie

## Description

A trie is a tree-like data structure used for efficient retrieval and storage of strings. It organizes strings by their characters in a tree-like structure, making it ideal for tasks such as autocomplete, spell-checking, and searching for words with a common prefix.

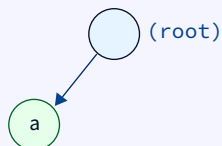
## Visualization of Trie

Let's create a trie out the following strings:

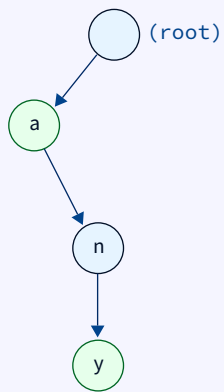
1. a
2. any
3. the
4. there
5. their
6. annoy



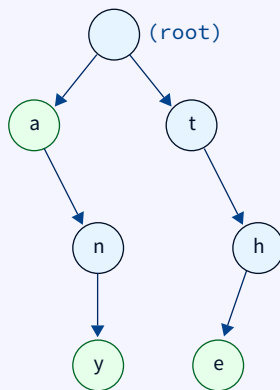
Insert "a"



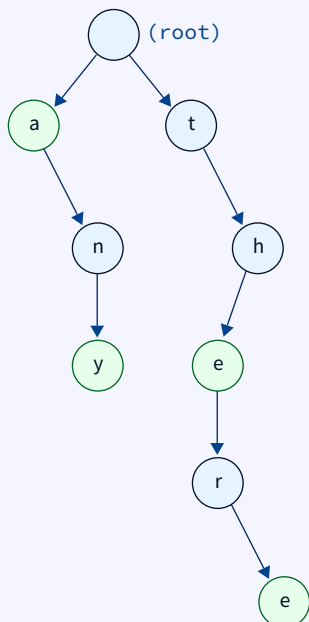
Insert "any"



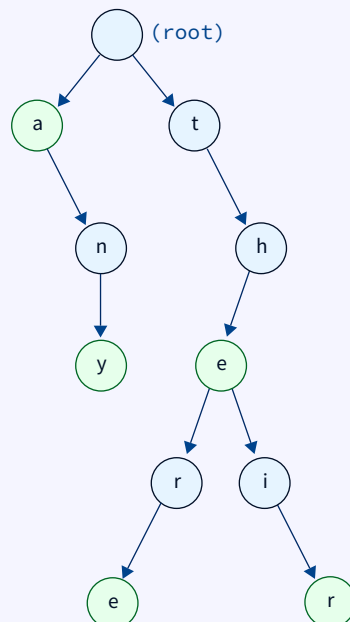
Insert "the"



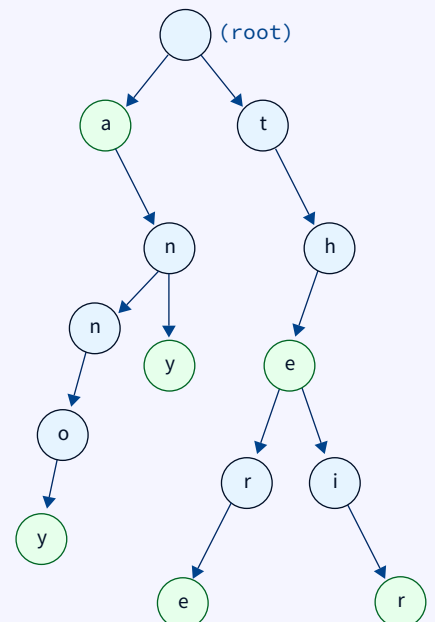
Insert "there"



Insert "their"



Insert "annoy"



Note: All green nodes are leaf nodes representing the tail of a word from the root.

## Time Complexity

"n" represents the length of the key/string under operation.

Operation	Time Complexity
Search	$O(n)$
Insertion	$O(n)$
Deletion	$O(n)$

## Notable Usages

Efficient prefix searching.  
Autocomplete suggestions.



## Space Complexity

$O(n * m)$  where  $n$  is the maximum length of a key and  $m$  is the number of keys inserted.

# SCALER TOPICS



Unlock your potential in software development with  
**FREE COURSES** from **SCALER TOPICS**!

Register now and take the first step towards your future Success!



**PRATEEK NARANG**

**C++ for Beginners**

 5.9k enrolled  **Free**



**TARUN LUTHRA**

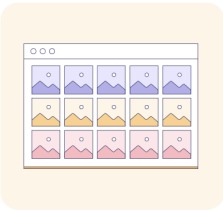
**Java for Beginners**

 6.8k enrolled  **Free**

That's not it. Explore 20+ Courses by clicking below

[Explore Other Courses](#)

Practice **CHALLENGES**  
and become 1% better everyday



**CIFAR-10 Image Classification Using PyTorch**  
Article

 No. Of Questions : 3

[Go to Challenge >](#)



**How to Build a Snake Game in JavaScript?**  
Article

 No. Of Questions : 3

[Go to Challenge >](#)

[Explore Other Challenges](#)