Singly Linked List

Queues

Data Structures CHEAT SHEET

SCALER Topics

Time Complexity

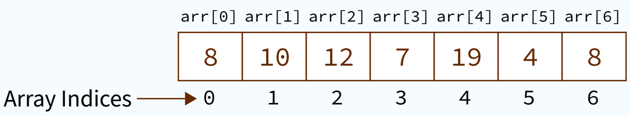
Arrays

Arrays

Description

An array is a collection of elements identified by index or key values. The elements are stored contiguously in memory.

Visualization



Time Complexity

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

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

Space Complexity

O(n)

Notable Usages

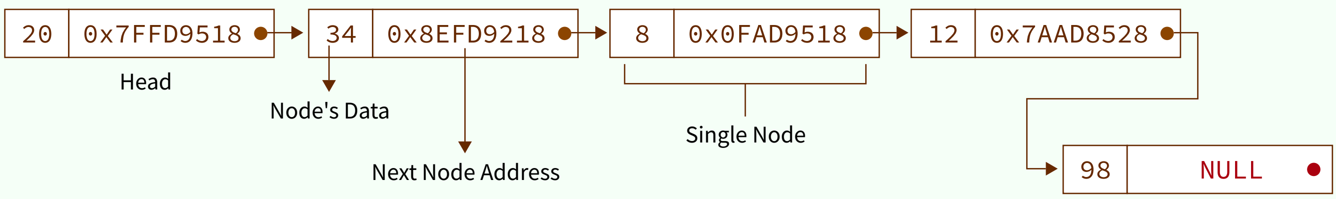
Efficient random access.
Contiguous memory allocation.

Singly Linked List

Description

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.

Visualization



Time Complexity

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

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

Space Complexity

O(n)

Notable Usages

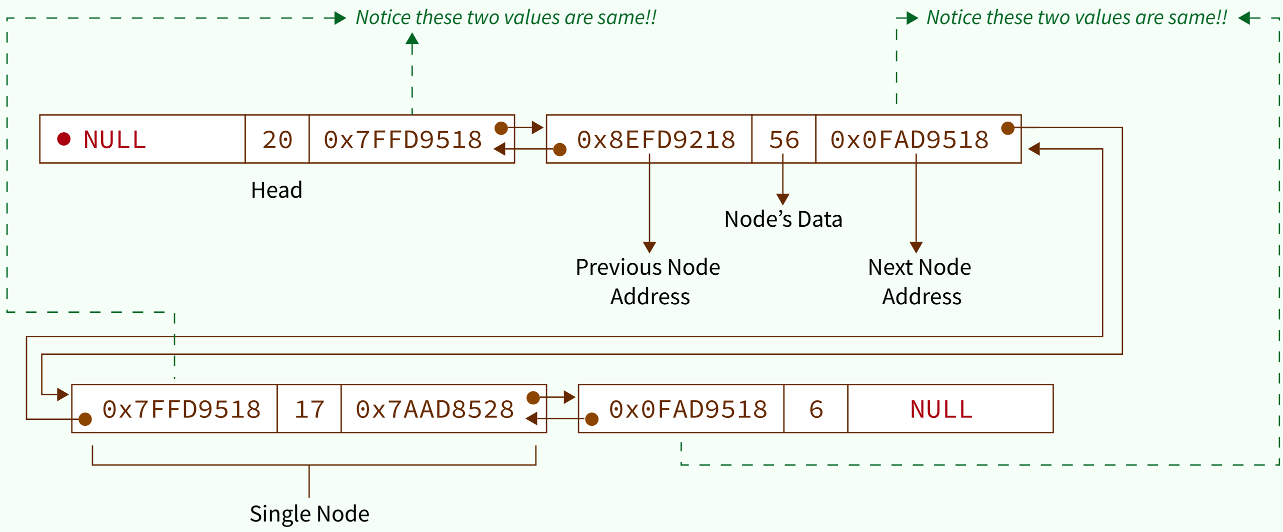
Dynamic memory allocation.
Efficient insertion and deletion at the beginning.

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

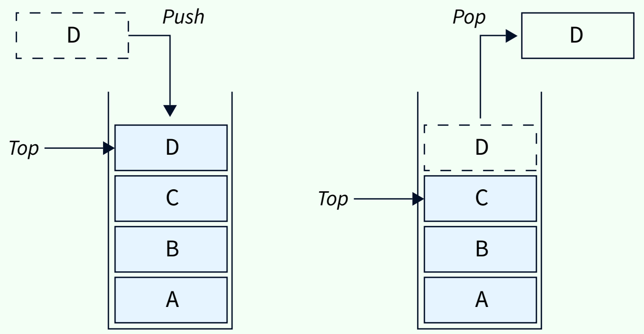
Bi-directional traversal.
Efficient deletion of a node.

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.

Visualization



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)

Notable Usages

Function call management.
Undo/Redo operations.

Space Complexity

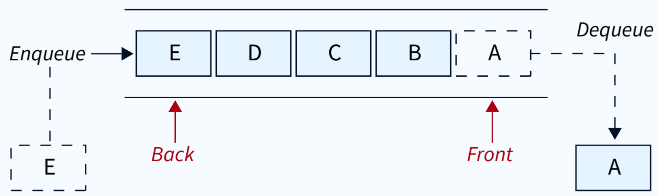
O(n)

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.

Visualization



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)

Notable Usages

Job scheduling.
Breadth-first search.

Space Complexity

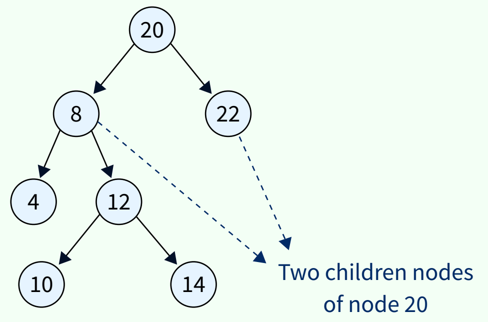
O(n)

Binary Tree

Description

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

Visualization



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)

Notable Usages

Hierarchical data representation.
Quick searching and sorting.

Space Complexity

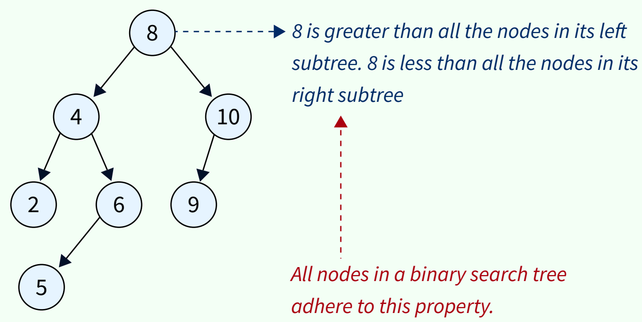
O(n)

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.

Visualization



Worst Case Time Complexity

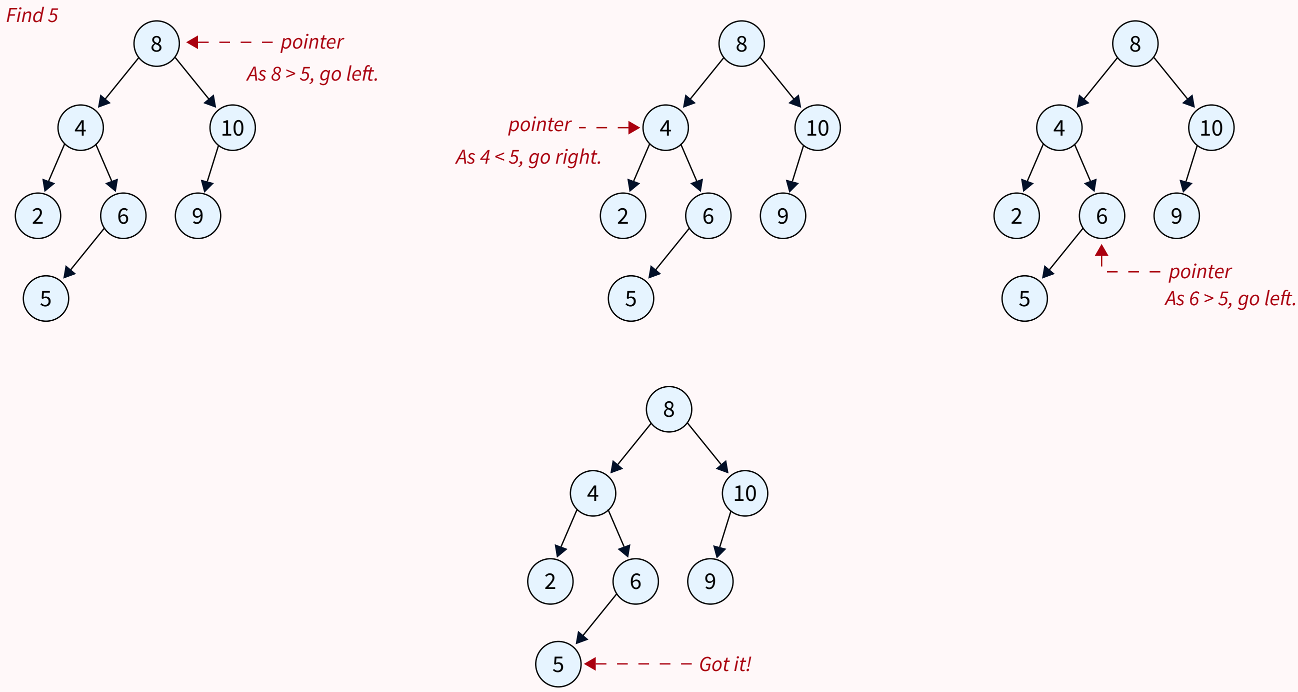
"n" represents the number of nodes in the binary search tree.

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

Space Complexity

O(n)

Demonstration of Search operation



Notable Usages

Efficient searching.
In-order traversal.

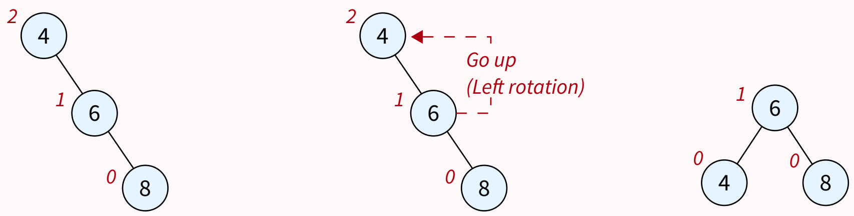
AVL Tree

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

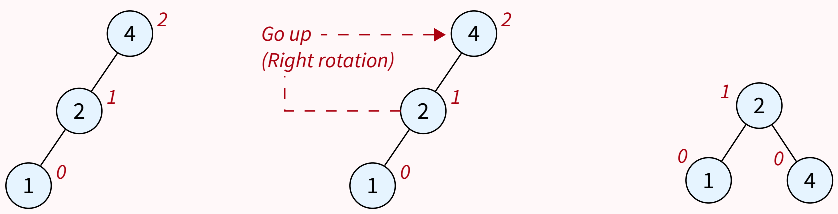


Unbalanced tree due to node 4

Left Rotation

Balanced Tree

Concept of Right Rotation

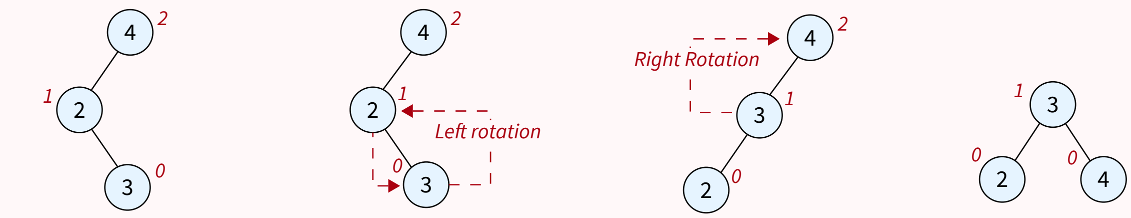


Unbalanced tree due to node 4

Right Rotation

Balanced Tree

Concept of Left-Right Rotation



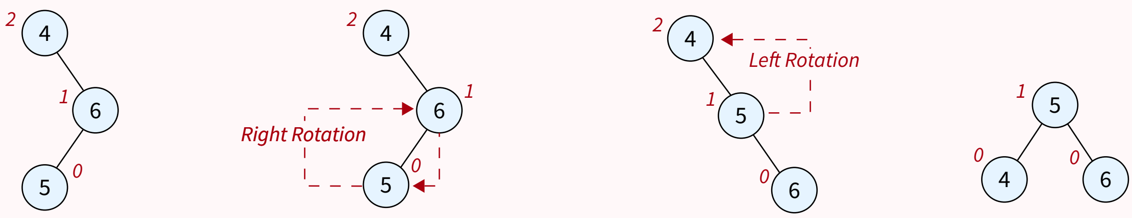
Unbalanced tree due to node 4

Left Rotation

Right Rotation

Balanced Tree

Concept of Right-Left Rotation



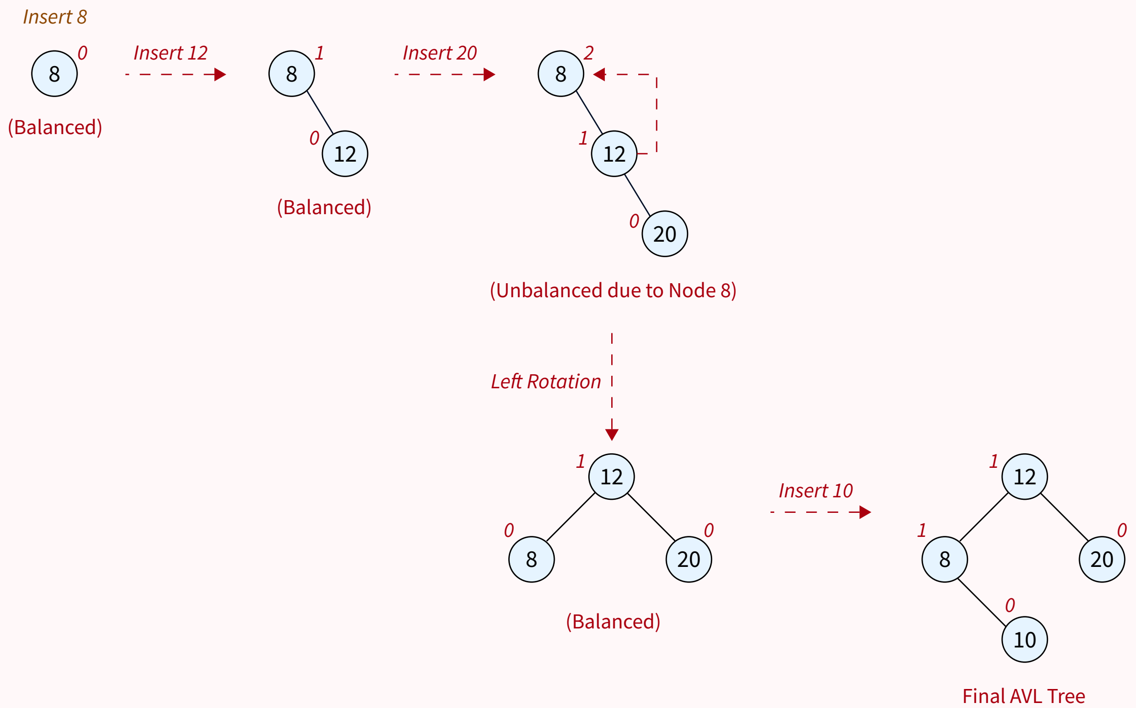
Unbalanced tree due to node 4

Right Rotation

Left Rotation

Balanced Tree

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



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)

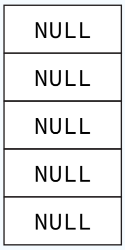
HashTable

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(key) = key % 5

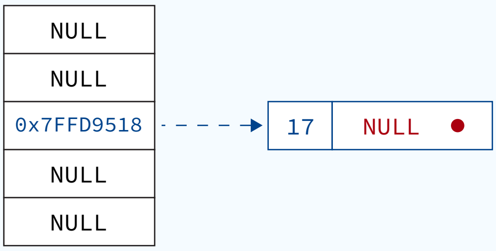


Let's insert the following keys into the hashtable.

```
17
43
 2
34
43
98
10
```

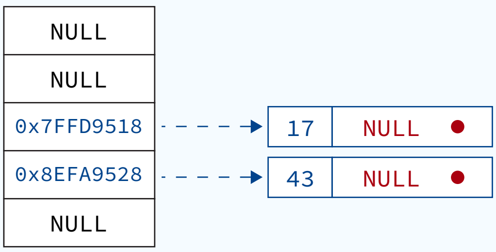
Insert 17

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



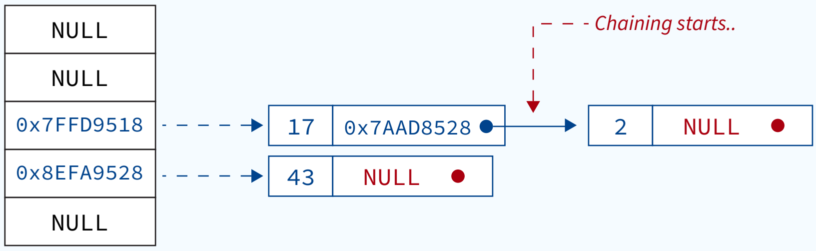
Insert 43

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



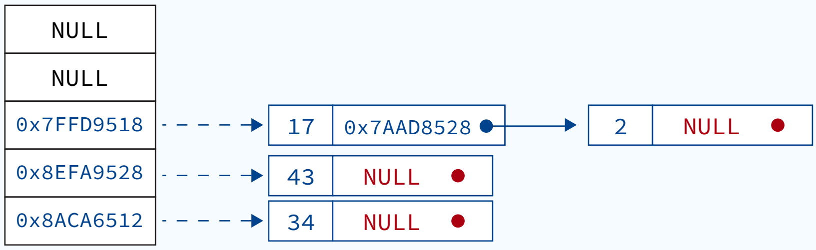
Insert 2

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



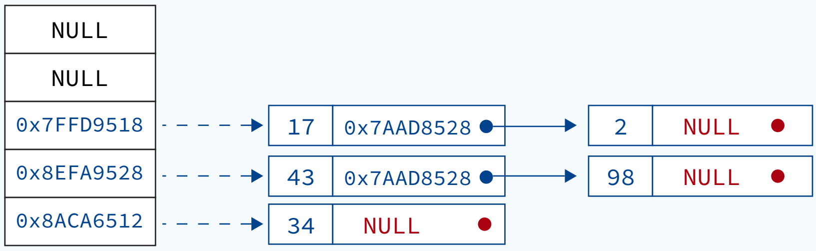
Insert 34

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



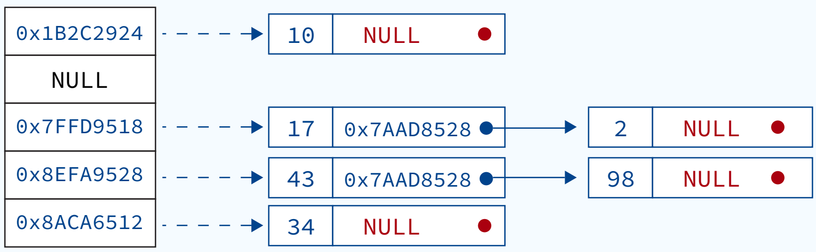
Insert 98

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

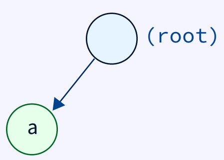
A trie is a tree-like data structure used for efficient retrieval and storage of strings. It organizes strings by their characters in a tree-like structure, making it ideal for tasks such as autocomplete, spell-checking, and searching for words with a common prefix.

Visualization of Trie

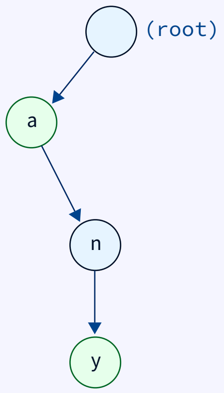
Let's create a trie out the following strings:

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

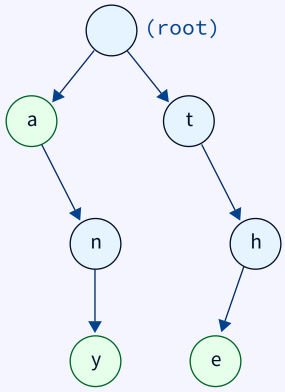
Insert "a"



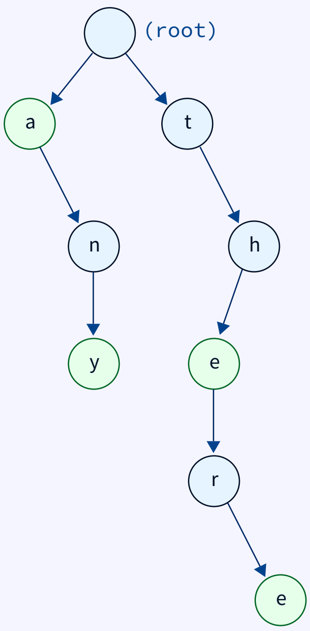
Insert "any"



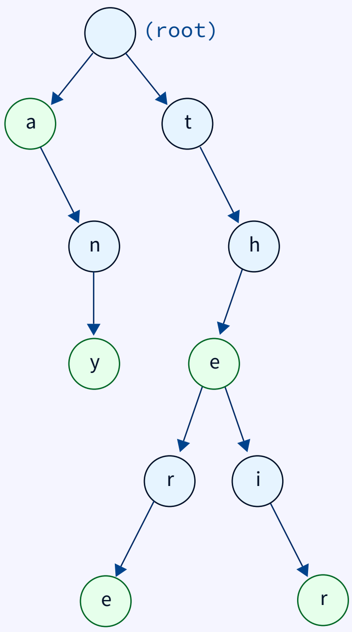
Insert "the"



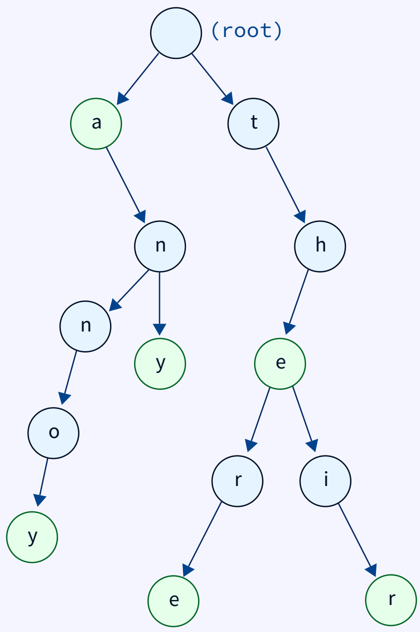
Insert "there"



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)

Space Complexity

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

Notable Usages

Efficient prefix searching.
Autocomplete suggestions.