Essential data structures

Mustafa Hajij
MSDS program
University of San Francisco

UNIVERSITY OF SAN FRANCISCO

Arrays

Arrays:

Definition: An array is a collection of elements, each identified by an index or a key.

Arrays

Arrays:

Definition: An array is a collection of elements, each identified by an index or a key.

Time Complexity:

-Access (Read/Write): O(1) - Accessing an element in an array by index is constant time.

Arrays

Arrays:

Definition: An array is a collection of elements, each identified by an index or a key.

Time Complexity:
-Access (Read/Write): O(1) - Accessing an element in an array by index is constant time.
- Insertion/Deletion at End: O(1) - Adding or removing an element at the end of an array is constant time.

Arrays

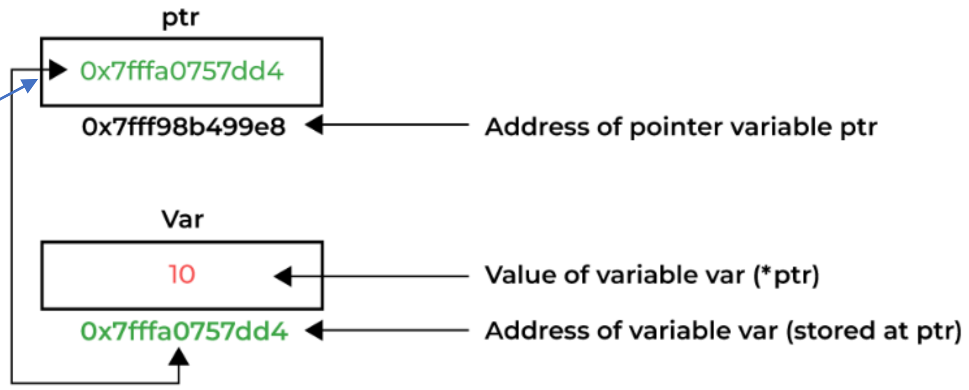
Arrays:

Definition: An array is a collection of elements, each identified by an index or a key.

Time Complexity:
-Access (Read/Write): O(1) - Accessing an element in an array by index is constant time.
- Insertion/Deletion at End: O(1) - Adding or removing an element at the end of an array is constant time.
- Insertion/Deletion at Arbitrary Position: O(n) - Inserting or deleting an element at an arbitrary position requires shifting elements and takes linear time.

Recall Pointers

- A pointer is a variable that stores the memory address of another variable in a programming language. It allows direct access to the memory location, enabling efficient manipulation and referencing of data.

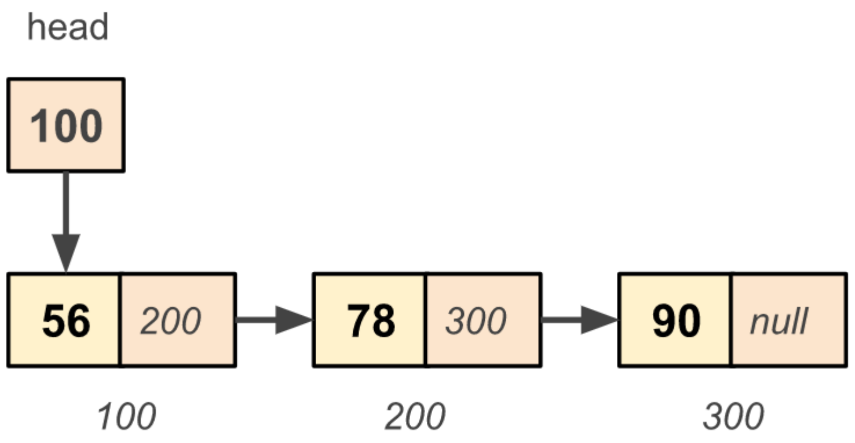

ptr
0x7fffa0757dd4
0x7fff98b499e8
Address of pointer variable ptr
Var
10
Value of variable var (*ptr)
0x7fffa0757dd4
Address of variable var (stored at ptr)

C Pointers - GeeksforGeeks

This pointer ptr stores the address of var

Linked List

A linked list is a linear data structure in which elements are stored in nodes, each containing a data element and a reference (pointer) to the next node,



Linked List

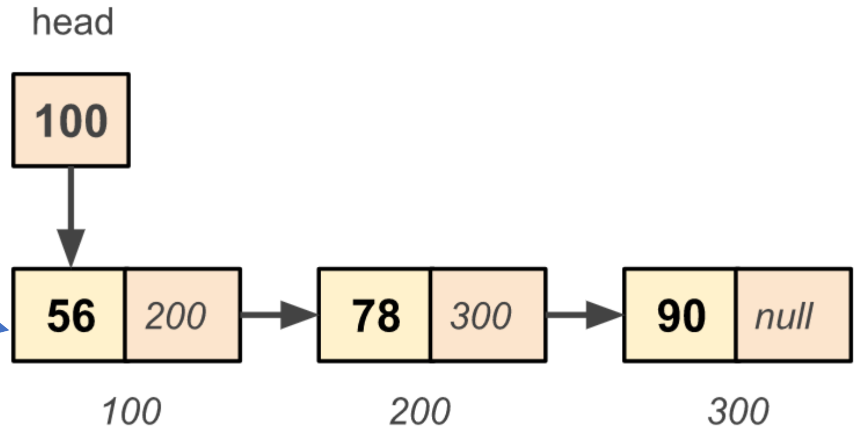
A linked list is a linear data structure in which elements are stored in nodes, each containing a data element and a reference (pointer) to the next node,

Properties :

- Linked lists allow dynamic allocation and efficient insertion/deletion.

- Require sequential traversal for access.

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None
```



Linked List

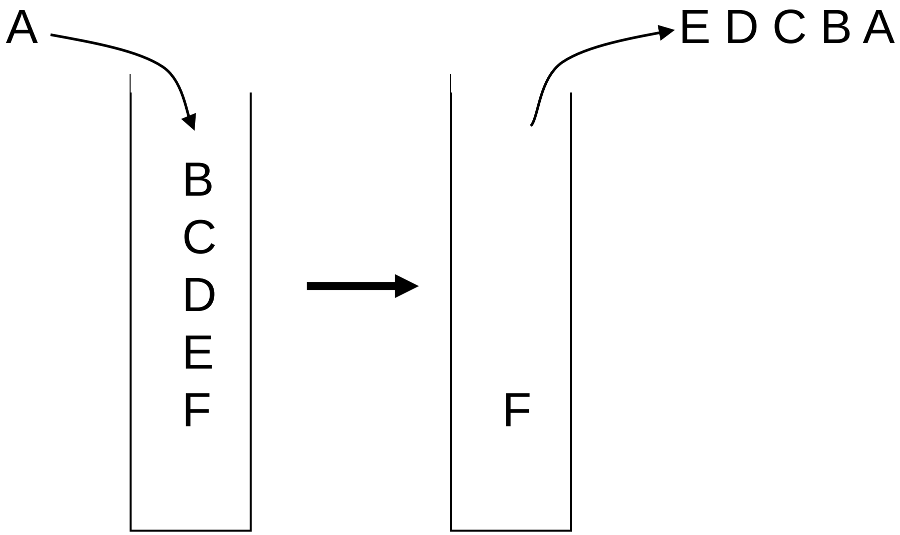
```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None
```

```
class LinkedList:
    def __init__(self):
        self.head = None

    def is_empty(self):
        return self.head is None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

 def display(self):
        current_node = self.head
        while current_node:
            print(current_node.data, end=" -> ")
            current_node = current_node.next
        print("None")
```

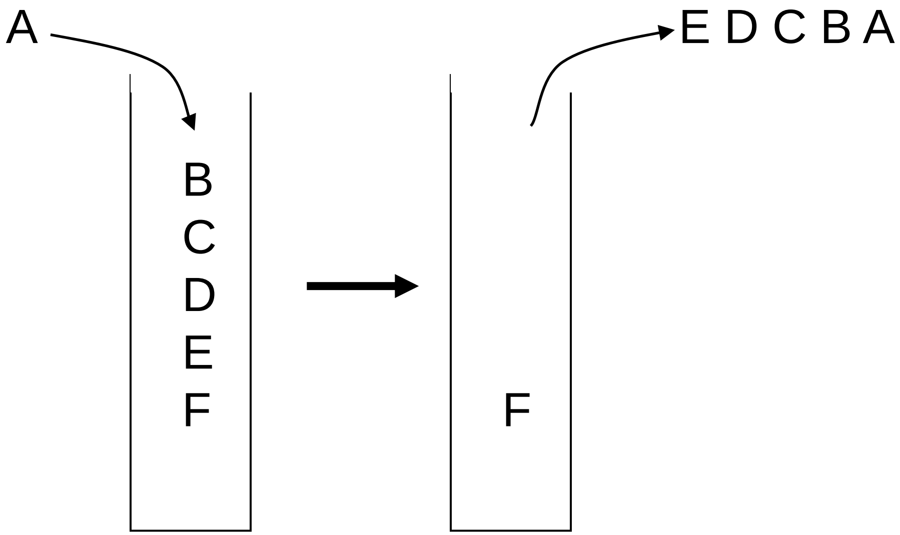
```
# Example usage:
linked_list = LinkedList()
linked_list.append(1)
linked_list.append(2)
linked_list.append(3)
linked_list.prepend(0)
linked_list.display()

# Output: 0 -> 1 -> 2 -> 3 -> None
```

Stack, Last In First Out data structure

- Stack operations
  - create
  - destroy
  - push
  - pop
  - top
  - is_empty



Stack, Last In First Out data structure



- Stack operations
  - create
  - destroy
  - push
  - pop
  - top
  - is_empty
- Stack property: if an element x is pushed into the stack before an element y is pushed, then x will be popped after y is popped.

  This is why this data structure is called a LIFO stands for Last In First Out

```
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            print("Stack is empty. Cannot pop.")

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            print("Stack is empty. Cannot peek.")

    def size(self):
        return len(self.items)
```

```
# Example usage:
stack = Stack()

stack.push(1)
stack.push(2)
stack.push(3)

print("Stack:", stack.items)
print("Size:", stack.size())
print("Peek:", stack.peek())

popped_item = stack.pop()
print("Popped item:", popped_item)
print("Stack after pop:", stack.items)
```

Queue, First In First Out data structure

Queue Operations:

- 1. Create: Initialize an empty queue.
- 2. Destroy: Deallocate resources and remove all elements from the queue.
- 3. Enqueue (Push): Add an element to the rear of the queue.
- 4. Dequeue (Pop): Remove and return the element from the front of the queue.
- 5. Is Empty: Check if the queue is empty.

Queue, First In First Out data structure

Queue Operations:

- 1. Create: Initialize an empty queue.
- 2. Destroy: Deallocate resources and remove all elements from the queue.
- 3. Enqueue (Push): Add an element to the rear of the queue.
- 4. Dequeue (Pop): Remove and return the element from the front of the queue.
- 5. Is Empty: Check if the queue is empty.

- Queue Property:
- If an element x is enqueued into the queue before an element y is enqueued, then x will be dequeued before y is dequeued. This is why this data structure is called a FIFO, which stands for First In First Out.

- Note: In a queue, elements are added at the rear and removed from the front.

```
class Queue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        else:
            print("Queue is empty. Cannot dequeue.")

    def size(self):
        return len(self.items)
```

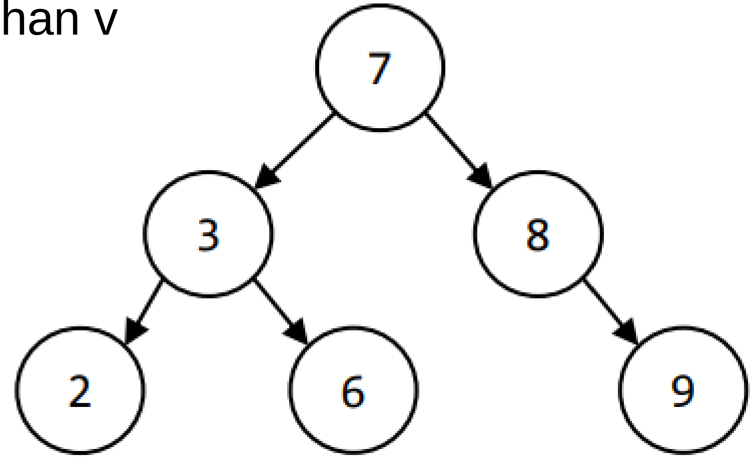
```
# Example usage:
queue = Queue()

queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)

print("Queue:", queue.items)
print("Size:", queue.size())
print("Front:", queue.front())
print("Rear:", queue.rear())

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)
print("Queue after dequeue:", queue.items)
```

Binary Search Tree

- *What is a Binary Search Tree (BST)?*
  - A data structure that organizes elements in a hierarchical tree-like structure.
  - Key feature: Each node has at most two children, with a specific ordering property.

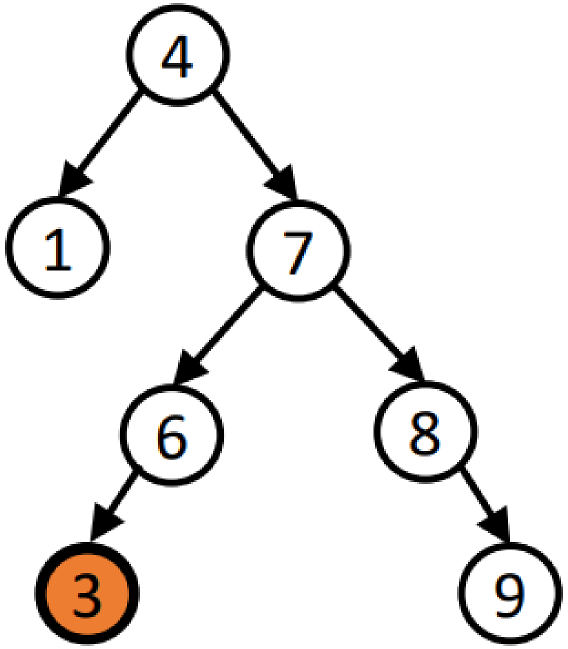
For every node n in a tree which has a value v:

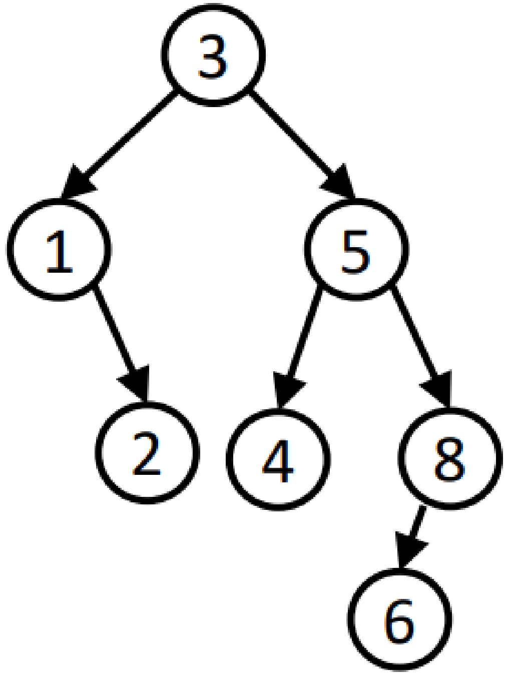
- Each left child (and all its children, etc.) must be strictly less than v
- Each right child (and all its children, etc.) must be strictly greater than v



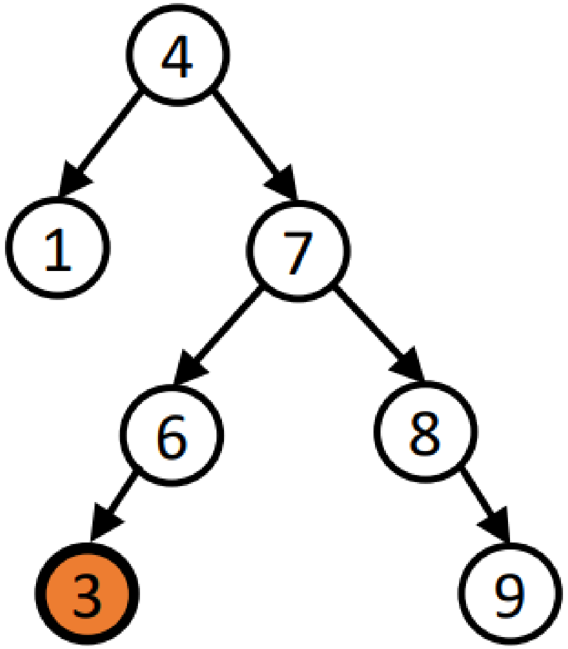
Binary Search Tree

Which binary tree is BST?

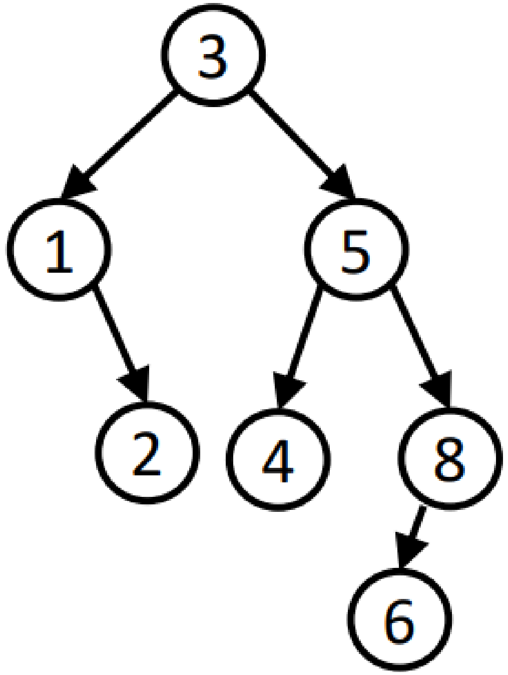




Which binary tree is BST?



no

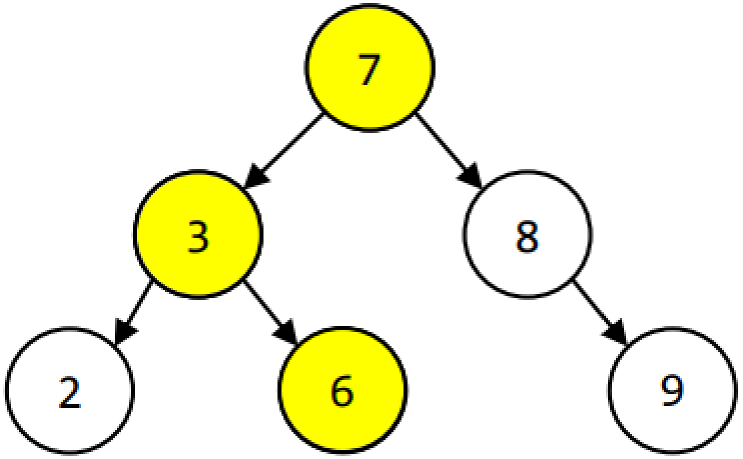


yes

Looking for 5

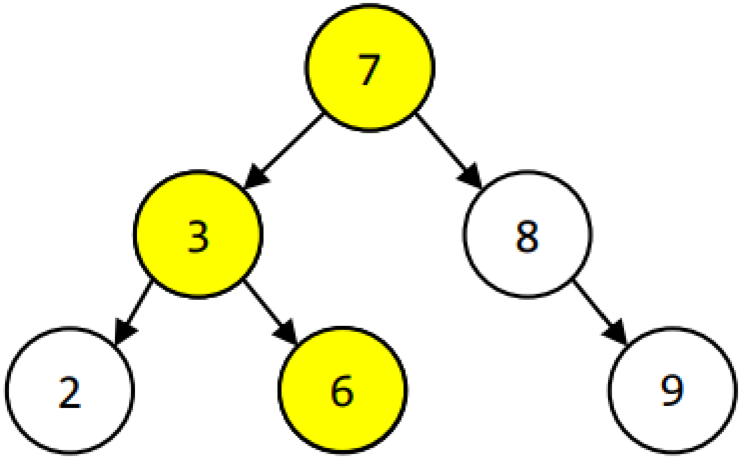
1.Starting Point: Root Node 7

- We initiate the search from the root node, which is 7 in this case.



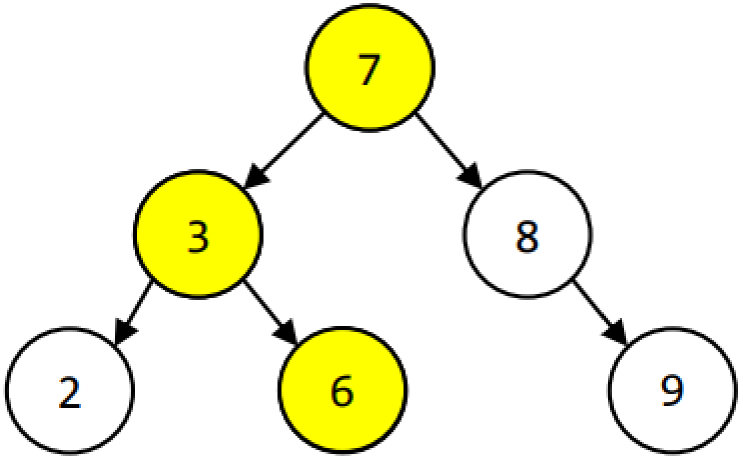
Looking for 5

1. **Starting Point: Root Node 7**
   - We initiate the search from the root node, which is 7 in this case.
2. **Comparison with 7: Move to Left Child**
   - Since all nodes less than 7 are situated in the left subtree, and 5 is indeed less than 7, our search focuses solely on the left child tree.



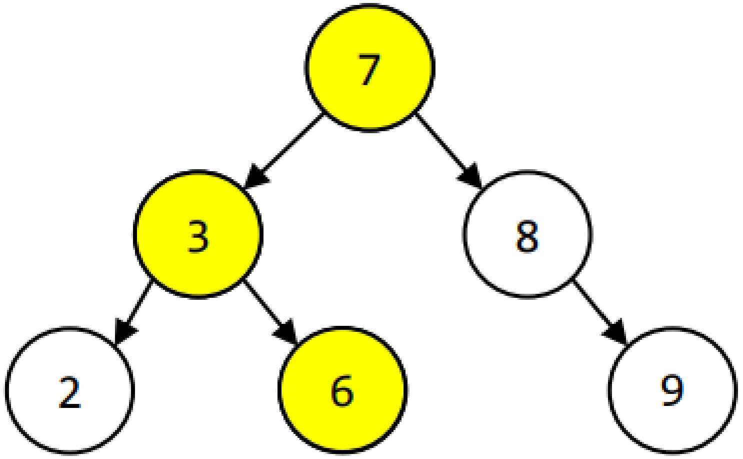
Looking for 5

1. **Starting Point: Root Node 7**
   - We initiate the search from the root node, which is 7 in this case.
2. **Comparison with 7: Move to Left Child**
   - Since all nodes less than 7 are situated in the left subtree, and 5 is indeed less than 7, our search focuses solely on the left child tree.
3. **Comparison with 3: Move to Right Child**
   - Subsequently, as we compare 5 to 3, the realization is that all values greater than 3 but less than 7 must exist in the right subtree of 3. Given that 5 is great than 3, our search narrows down to the right child of 3.



Looking for 5

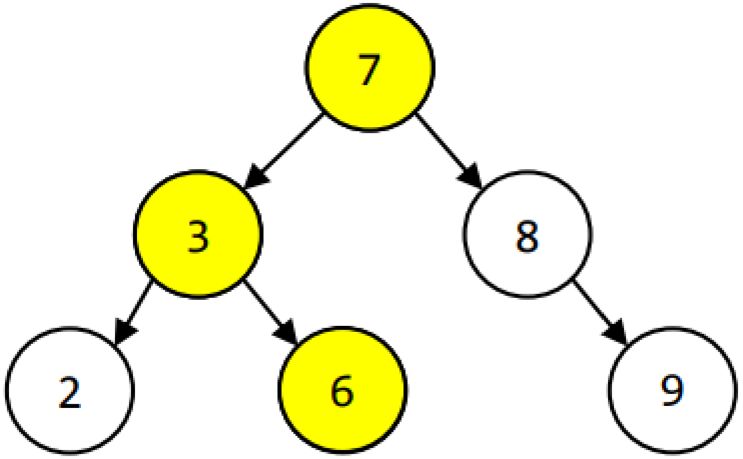
1. **Starting Point: Root Node 7**
   - We initiate the search from the root node, which is 7 in this case.
2. **Comparison with 7: Move to Left Child**
   - Since all nodes less than 7 are situated in the left subtree, and 5 is indeed less than 7, our search focuses solely on the left child tree.
3. **Comparison with 3: Move to Right Child**
   - Subsequently, as we compare 5 to 3, the realization is that all values greater than 3 but less than 7 must exist in the right subtree of 3. Given that 5 is great than 3, our search narrows down to the right child of 3.
4. **Binary Search Principle**
   - The entire process mirrors the essence of binary search, a systematic approac to efficiently locate a specific value within a sorted dataset.



Binary Search Tree

Looking for 5

1. **Starting Point: Root Node 7**
   - We initiate the search from the root node, which is 7 in this case.
2. **Comparison with 7: Move to Left Child**
   - Since all nodes less than 7 are situated in the left subtree, and 5 is indeed less than 7, our search focuses solely on the left child tree.
3. **Comparison with 3: Move to Right Child**
   - Subsequently, as we compare 5 to 3, the realization is that all values greater than 3 but less than 7 must exist in the right subtree of 3. Given that 5 is great than 3, our search narrows down to the right child of 3.
4. **Binary Search Principle**
   - The entire process mirrors the essence of binary search, a systematic approach to efficiently locate a specific value within a sorted dataset.
5. **Conclusion: Binary Search Tree (BST)**
   - Therefore, the search methodology aligns with the principles of a Binary Searc Tree (BST), where nodes are arranged in a hierarchical structure with the left child containing smaller values and the right child containing larger values.



Binary Search Tree : main methods

- *Insertion*
  - Add a new key while maintaining the order of the BST.
- *Deletion*
  - Remove a key, ensuring the BST properties are preserved.
- *Search*
  - Locate a specific key efficiently using the binary search property.

Binary Search Tree

```
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
```

```
class BST:
    def __init__(self):
        self.root = None

    def insert(self, key):
        self.root = self._insert(self.root, key)

    def _insert(self, root, key):
        if root is None:
            return TreeNode(key)
        if key < root.key:
            root.left = self._insert(root.left, key)
        elif key > root.key:
            root.right = self._insert(root.right, key)
        return root

    def search(self, key):
        return self._search(self.root, key)

    def _search(self, root, key):
        if root is None or root.key == key:
            return root
        if key < root.key:
            return self._search(root.left, key)
        return self._search(root.right, key)
```

```
# Example Usage:

bst = BST()
values = [7, 3, 9, 2, 5, 8, 10]

for value in values:
    bst.insert(value)

search_key = 5
result = bst.search(search_key)

if result:
    print(f"Value {search_key} found in the BST.")
else:
    print(f"Value {search_key} not found in the BST.")
```

Let examine search more closely

```
def search(p:TreeNode, x:object):
    if p is None: return None
    if x < p.value:
        return search(p.left, x)
    if x > p.value:
        return search(p.right, x)
    return p
```

Binary Search Tree : complexity

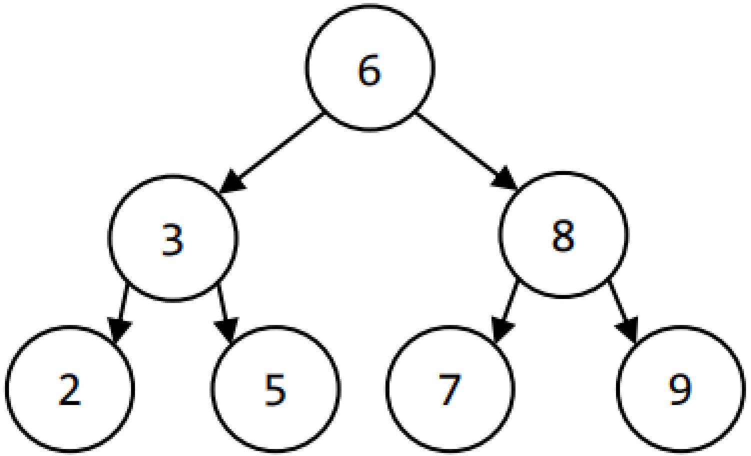
- *Average Case*
  - Search, Insertion, and Deletion: O(log n)
- *Worst Case*
  - Unbalanced Tree: O(n) - Degenerates to a linked list.

Balanced Binary Search Tree

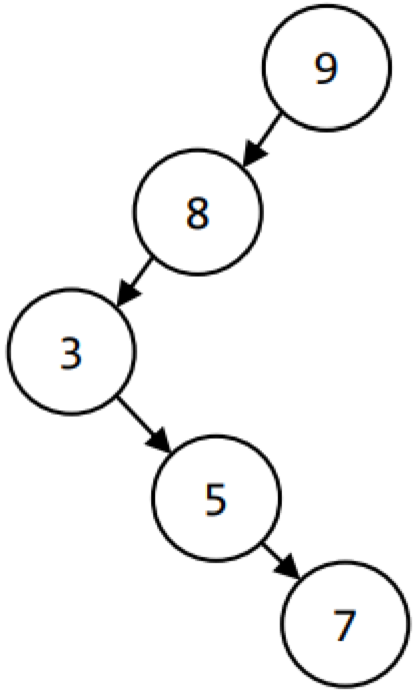
Let's consider the runtime of search on a BST that is balanced.

A tree is balanced if for every node in the tree, the node's left and right subtrees are approximately the same size. This results in a tree that minimizes the number of recursive levels.

Every time you take a search step in a balanced tree, you cut the number of nodes to be searched in half. This means that you'll take O(log n) time, like with ordinary binary search.

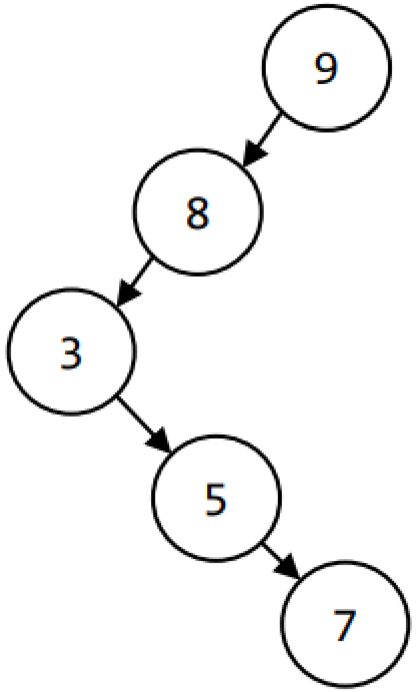


Unbalanced Binary Search Tree



Is this a valid BST?

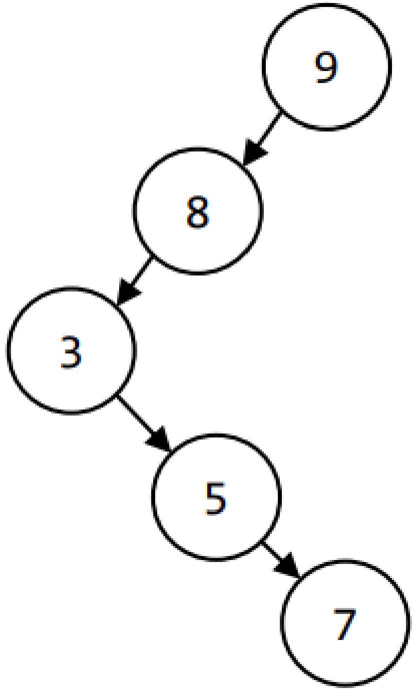
Unbalanced Binary Search Tree



Is this a valid BST? Yes!

Unbalanced Binary Search Tree

A tree is considered unbalanced if at least one node has significantly different sizes in its left and right children. For example, consider the tree on the right.

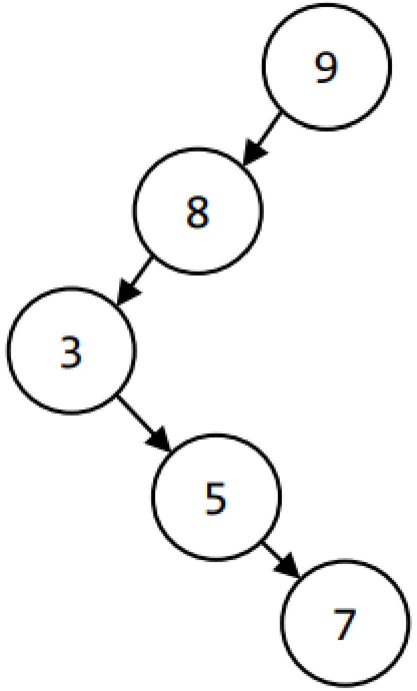


Unbalanced Binary Search Tree

A tree is considered unbalanced if at least one node has significantly different sizes in its left and right children. For example, consider the tree on the right.

This is a valid BST, but it is still difficult to search! If you search it for a number like 6, it can still take O(n) time. When we put data into BSTs, we usually strive to make them balanced, to avoid these edge cases.

When we design the insertion function in that way, we can assume the average runtime will be O(log n), why?.



Dictionaries (Hashtables)

- Dict.Init(): Initialize the dictionary;
- Dict.Insert(Key K, Data D): Insert (key,data) in dictionary;
- bool Dict.Member(Key K): Return true if key K is in dictionary;
- Data Dict.Retrieve(Key K): Return data associated with key K.

Hash table: H[0], H[1], . . . , H[m − 1]. m = Size of hash table. This about it as a list of size m.

Hash functions: h : Key K → {0, 1, . . . , m − 1}.

Examples of hash functions

The set of all Keys K is typically much larger than the size of the table m, could be infinite!

- h(K) = K mod m;
- h(K) = ((aK + b) mod p) mod m for some large prime p >> m

Bad hash functions

Lets consider the Division hashing: h(K) = K mod m.

Problems:

- If m = 1000, then keys (1027, 2027, 3027, . . . , 9027) all map to H[27];

Bad hash functions

Lets consider the Division hashing: h(K) = K mod m.

Problems:

- If m = 1000, then keys (1027, 2027, 3027, . . . , 9027) all map to H[27];

- If m is even and keys K are always even, then h(K) is always even. (Never access the H[i] for i odd.)

Bad hash functions

Lets consider the Division hashing: h(K) = K mod m.

Problems:

- If m = 1000, then keys (1027, 2027, 3027, . . . , 9027) all map to H[27];

- If m is even and keys K are always even, then h(K) is always even. (Never access the H[i] for i odd.)

- If m is a multiple of 10 and keys K are always multiples of 10, then h(K) is always a power of 10. (Only access 1/10’th of the hash table, a huge waste of memory.)

Rule of thumb: m should be a prime.

Good hash functions

- All hash table locations are equally likely to be accessed;

- Keys with a regular pattern are not mapped to the same locations

Hashing : collision

Collisions means : h(K1) = h(K2) but K1 is not equal K2.

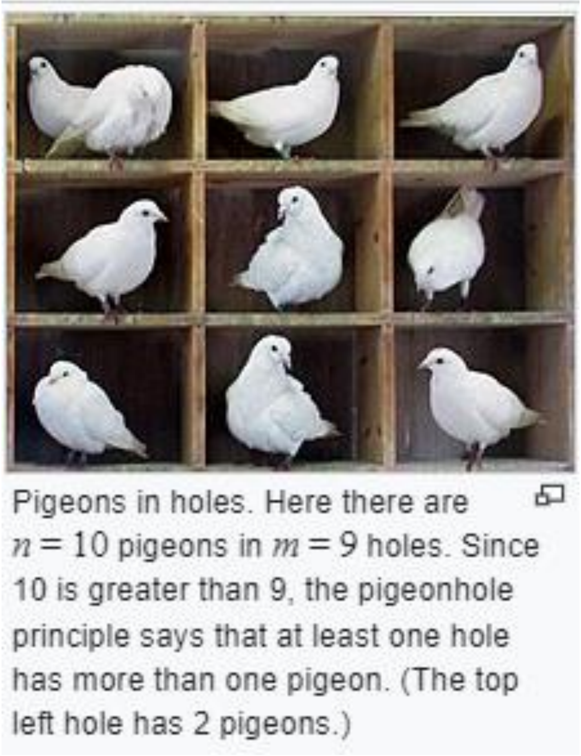
Collisions is unavoidable, why?

Hashing : collision

Collisions means : h(K1) = h(K2) but K1 is not equal K2.

Collisions is unavoidable, why?

The Pigeonhole principle



Pigeons in holes. Here there are $n = 10$ pigeons in $m = 9$ holes. Since 10 is greater than 9, the pigeonhole principle says that at least one hole has more than one pigeon. (The top left hole has 2 pigeons.)

https://en.wikipedia.org/wiki/Pigeonhole_principle

Hashing : collision

Remember that Hash table: H[0], H[1], . . . , H[m − 1].

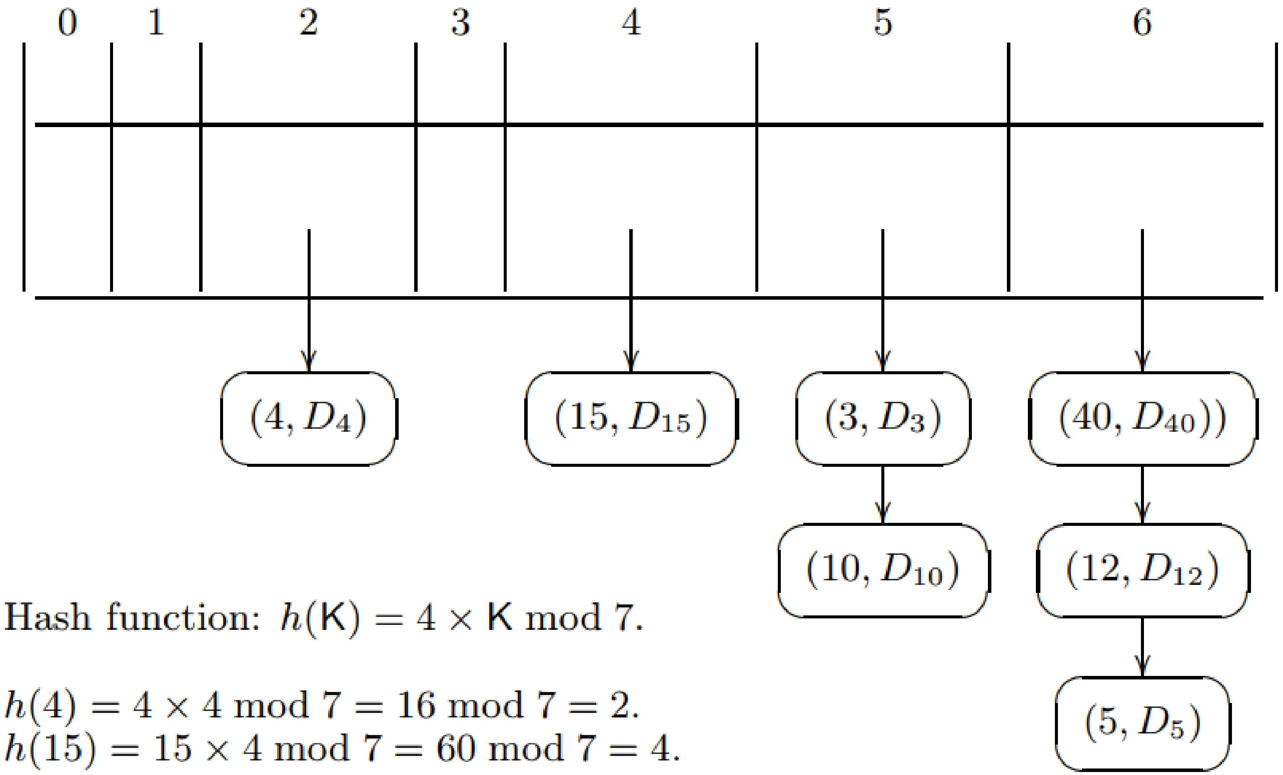
m = Size of hash table.
n = Number of elements in the table.
N = Size of the set containing all possible keys. (e.g., 232 or 264 for 32-bit or 64-bit unsigned integers.) (N could be infinite! For example when the keys are all possible strings.)

Typically, N >> m > n

Hashing : collision

One solution for the collision problem : Chained Hashing.

Each location $H[i]$ is a linked list.



Hash function: $h(\mathsf{K}) = 4 \times \mathsf{K} \bmod 7$.

$h(4) = 4 \times 4 \bmod 7 = 16 \bmod 7 = 2.$
$h(15) = 15 \times 4 \bmod 7 = 60 \bmod 7 = 4.$

Hashing : application

Problem: For each element x in array A, report the number of occurrences of x in A.

Hashing : application

Problem: Return true if there is a duplicate element in a given list.

UNIVERSITY OF SAN FRANCISCO

Graphs

A graph is a data structure that consists of a set of nodes (vertices) and a set of edges connecting these nodes. The edges may have a direction (directed graph) or may not (undirected graph). Graphs are widely used to represent relationships and connections between different entities.

A few terms :

1. Vertex (Node): A fundamental unit in a graph, representing an entity.

2. Edge: A connection between two vertices. It may have a direction (directed) or not (undirected).

3. Directed Graph: A graph in which edges have a direction, indicating a one-way relationship between vertices.

4. Undirected Graph: A graph in which edges have no direction, representing a mutual relationship between vertices.

5. Path: A sequence of vertices where each adjacent pair is connected by an edge.

6. Cycle: A path that starts and ends at the same vertex, forming a closed loop.

Graphs are used in various applications, including social networks, transportation systems, computer networks, and more. They provide a powerful and flexible way to model and analyze relationships between entities.