



CoGrammar

Trees



**SKILLS
FOR LIFE**

SKILLS BOOTCAMPS



Department
for Education

Foundational Sessions Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.
(FBV: Mutual Respect.)
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Open Classes.

You can submit these questions here:

[SE Open Class Questions](#) or [DS Open Class Questions](#)

Foundational Sessions Housekeeping cont.

- For all **non-academic questions**, please submit a query: www.hyperiondev.com/support
- Report a **safeguarding** incident: www.hyperiondev.com/safeguardreporting
- We would love your **feedback** on lectures: [Feedback on Lectures](#)

Reminders!

Guided Learning Hours

By now, ideally you should have 7 GLHs per week accrued. Remember to attend any and all sessions for support, and to ensure you reach 112 GLHs by the close of your Skills Bootcamp.

Progression Criteria

✓ **Criterion 1: Initial Requirements**

- Complete 15 hours of Guided Learning Hours and the first four tasks within two weeks.

✓ **Criterion 2: Mid-Course Progress**

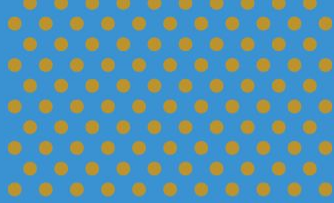
- Software Engineering: Finish 14 tasks by week 8.
- Data Science: Finish 13 tasks by week 8.

✓ **Criterion 3: Post-Course Progress**


- Complete all mandatory tasks by 24th March 2024.
- Record an Invitation to Interview within 4 weeks of course completion, or by 30th March 2024.
- Achieve 112 GLH by 24th March 2024.


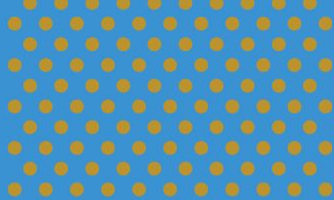
✓ **Criterion 4: Employability**

- Record a Final Job Outcome within 12 weeks of graduation, or by 23rd September 2024.

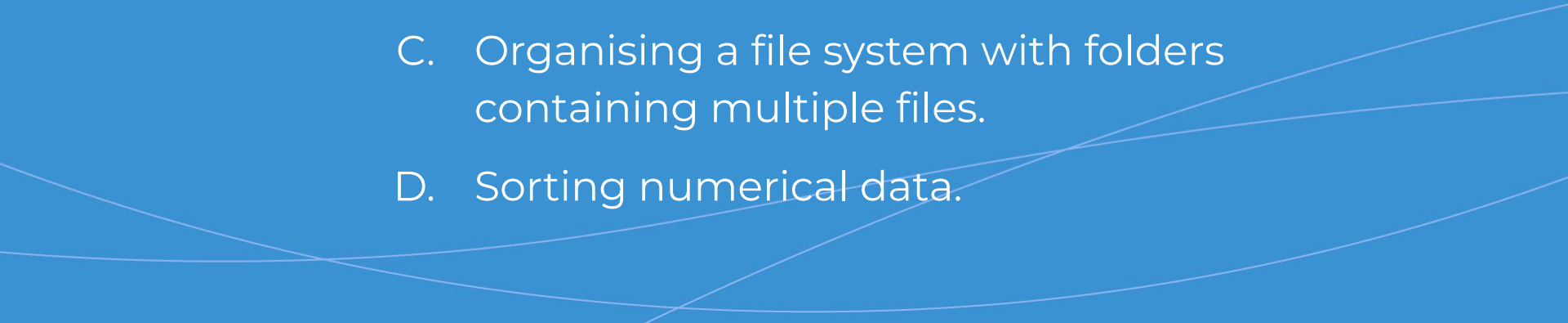



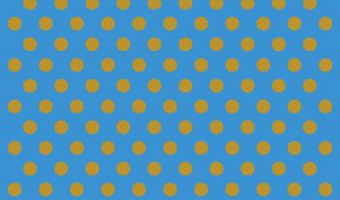
What is the maximum number of children for a node in a binary tree?

- A. One
 - B. Two
 - C. Three
 - D. Unlimited
- 




In which scenario is a rose tree (multi-way tree) more suitable than a binary tree?

- A. Managing a priority queue.
 - B. Representing binary relationships.
 - C. Organising a file system with folders containing multiple files.
 - D. Sorting numerical data.
- 



What is the time complexity of searching for an element in a balanced binary search tree?

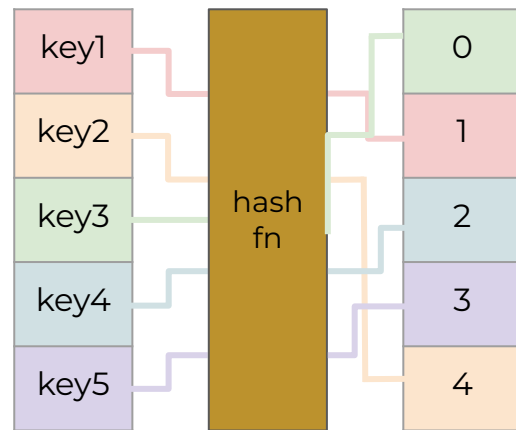
- A. $O(1)$
 - B. $O(\log n)$
 - C. $O(n)$
 - D. $O(n \log n)$
- 

Recap of Hash Tables



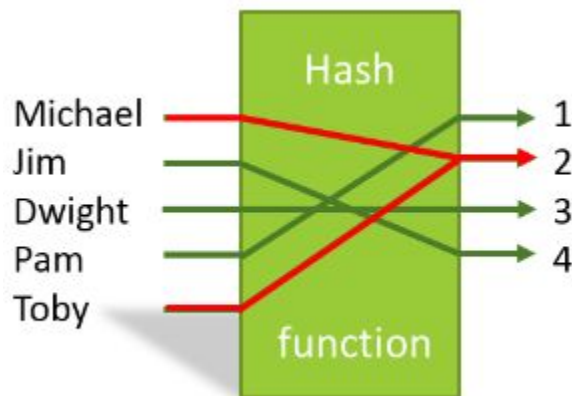
Hash Tables

- A data structure that is used to **store key-value pairs** which uses a **hash function to transform the key into the index of an associated array element** where the data will be stored.
- **Hashing** refers to using the hash function to calculate the hash which is the index of the array element where the key-value pair will be stored and it is **deterministic**.



Hash Collisions

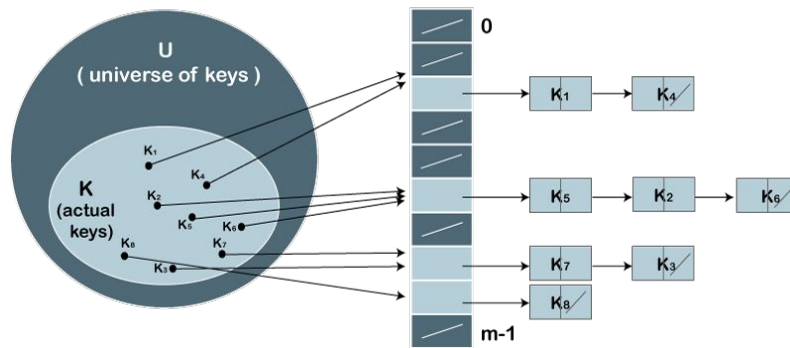
- When **two or more distinct keys** hash to the same array index. We solve this with a **collision resolution process**.



Collision-Resolution Methods

- **Chaining:** Linked Lists are used for each bucket, so multiple key-value pairs can be stored in the same bucket.
 - Pros: Simple to implement, dynamic resizing
 - Cons: Memory overhead implications, space inefficiencies

Collision Resolution by Chaining



- **Linear Probing:** If there is a collision, place the pair in the next available slot in the hash table (linearly).
 - Pros: Simple to implement, memory-efficient
 - Cons: Primary clustering (large blocks of occupied elements), clustering results in more time inefficiencies

insert (28)	
0	
1	
2	
3	
4	
5	
6	
7	
8	28
9	

insert (55)	
0	
1	
2	
3	
4	
5	55,
6	
7	
8	28
9	

insert (71)	
0	
1	71
2	
3	
4	
5	55,
6	
7	
8	28
9	

insert (67)	
0	
1	71
2	
3	
4	
5	55,
6	
7	67
8	28
9	

insert (11)	
0	
1	71
2	11
3	
4	
5	55,
6	
7	67
8	28
9	

Collision

insert (10)	
0	10
1	71
2	11
3	
4	
5	55,
6	
7	67
8	28
9	

insert (90)	
0	10
1	71
2	11
3	90,
4	
5	55,
6	
7	67
8	28
9	

Collision

insert (44)	
0	10
1	71
2	11
3	90,
4	44
5	55,
6	
7	67
8	28
9	

Dictionaries in Python

- They are **implemented internally using a hash table**, which makes use of Python's built-in hash function.
- Dictionaries are the **easiest way to implement a hash table in Python**, since all the complexities involved in ensuring efficiency is abstracted away and is done for us.

Trees Topics

1. Binary Trees
2. Binary Search Trees
3. Rose Trees
4. B-Trees
5. Iterative vs Recursive Heap Implementations using Binary Tree

The Role of Binary and Rose Trees in Data Organisation

Consider an e-commerce platform with an extensive product catalog organised hierarchically.

- Challenges
 - **Efficient** searching and updating products.
 - **Managing complex hierarchical relationships** between products and categories.

- Solution with Binary Trees
 - Ideal for sorted data.
 - Operations like **search, insert, and delete can be optimised.**
- Solution with Rose Trees
 - Perfect for representing multi-level hierarchies.
 - Can **handle multiple child nodes per parent, reflecting complex relationships.**
- Real-World Impact: **Improved performance in querying and managing hierarchical data**, essential for a seamless user experience on the platform.

Example: Visualising Binary Tree

- Throughout the lecture we'll use the phrases “Binary Tree” and “Binary Search Tree” interchangeably, but our focus is on the commonly used Binary Search Tree which has differences to the Binary Tree and will be explained in a few slides
- Let's use a tool developed by [VISUALGO.NET](https://visualgo.net) to show dynamic visualisations (as documented in the slides for convenience 😊)

- Let's assume we have products with ids {5, 6, 2, 1, 8, 10, 9, 4, 7} that were received in this order and need to be documented

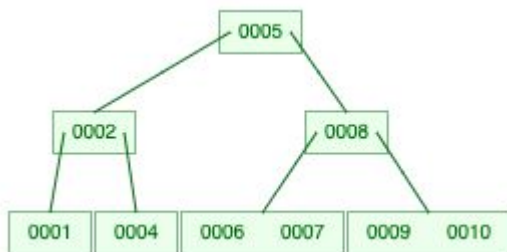


- Notice if we search for 7, we'll search a total of **3 nodes** before finding it, instead of through a set of **9 elements**. Similarly, inserts and deletes are more efficient. For massive datasets this becomes indispensable to a large company

Example: Visualising Rose Tree

- In this lecture we focus on B-trees since they are most common in industry, which are Rose Trees with two special properties
 1. **It is perfectly balanced**
 2. **Every node, except perhaps the root, is at least half-full**
- We'll use a tool developed by [University of San Francisco](#) to show this dynamically

- We'll use the exact same scenario used previously, where we have products with ids {5, 6, 2, 1, 8, 10, 9, 4, 7} that were received in this order and need to be stored in our catalog



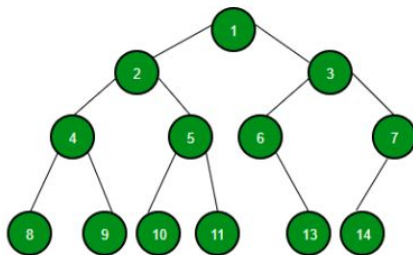
- Even compared to the efficient binary tree, the B-tree is extremely efficient for database management. If we search for 7, we'll find it after passing 2 nodes, although insertions could be a bit more complex than the simple Binary tree

Binary Trees

A tree structure where each node has at most two children

- **Properties**

- Hierarchical structure with a single root.
- Each node has zero, one, or two children.



Source: [GeeksForGeeks](https://www.geeksforgeeks.org/)

Binary Search Trees (BSTs)

A type of binary tree where each node has a key, and every node's key is greater than all keys in its left subtree and less than all keys in its right subtree.

- Here is the implementation of a single node in a BST, which will be useful to demonstrate the functions in the coming slides

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

Insert Function

Time Complexity: $O(n)$

which occurs when the tree becomes unbalanced and takes the form of a linked list

Space Complexity: $O(n)$

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

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

    def _insert_recursive(self, node, key):
        if node is None:
            return BSTNode(key)
        if key < node.val:
            node.left = self._insert_recursive(node.left, key)
        else:
            node.right = self._insert_recursive(node.right, key)
        return node
```


Search Function

Time Complexity: $O(n)$, in a completely unbalanced tree

Space Complexity: $O(n)$

```
def search(self, key):  
    return self._search_recursive(self.root, key)  
  
def _search_recursive(self, node, key):  
    if node is None or node.val == key:  
        return node  
    if key < node.val:  
        return self._search_recursive(node.left, key)  
    return self._search_recursive(node.right, key)
```

For example, let's create a BST from {5, 6, 2, 1, 8, 10, 9, 4, 7} and verify that 7 is in the tree, like the example at the start



```
bst = BinarySearchTree()
bst.insert(5)
bst.insert(6)
bst.insert(2)
bst.insert(1)
bst.insert(8)
bst.insert(10)
bst.insert(9)
bst.insert(4)
bst.insert(7)
print("Found" if bst.search(7) else "Not Found")
# Found
```

Rose Trees

A tree in which each node can have an arbitrary number of children

- Rose Trees are ideal for representing hierarchical structures with varying depths
- The Rose Tree node is similar to the BST node, although it has an array for all it's children, not just the left and right children

```
class RoseTreeNode:  
    def __init__(self, value):  
        self.value = value  
        self.children = []
```

Insert Function

Time Complexity: $O(n)$,
where n is, just as a
reminder, the number of
nodes in the tree

Space Complexity: $O(n)$

```
class RoseTree:
    def insert(self, parent, child_value):
        child = RoseTreeNode(child_value)
        parent.children.append(child)
        return child
```

Search Function

Time Complexity: $O(n)$

Space Complexity: $O(n)$

```
def search(self, node, key):  
    if node.value == key:  
        return node  
    for child in node.children:  
        found = self.search(child, key)  
        if found:  
            return found  
    return None
```

- While an example of Rose Trees could be an intuitive exercise, we will build intuition by way of introducing **B-Trees**, since they are widely used in database management and referred to much more often than “Rose Trees”. But now you have an understanding of what structure B-Trees stem from.

B-Trees

A type of balanced tree data structure, commonly used in databases and file systems. They are a specialized form of a rose tree, where each node can have multiple children

- **Properties:**

- Each node in a B-Tree can **have several children**, determined by the tree's order.
- Nodes keep data in sorted order, facilitating efficient access and insertion.
- **Automatically balances itself** to maintain a low height.

B-Tree Node and B-Tree Constructor

```
class BTreeNode:
    """Node of a B-Tree"""
    def __init__(self, order):
        self.keys = []
        self.children = []
        self.order = order
        self.is_leaf = True

class BTree:
    """B-Tree structure"""
    def __init__(self, order):
        self.root = BTreeNode(order)
        self.order = order
```

Disclaimer: Some complicated functions are omitted because they do not aid in understanding. Feel free to check it out in the script.

In practice you would use **pre-built libraries** to implement these structures, but for better understanding **implementing from scratch is a great exercise.**

Insert Function

Inserts a new key into the B-Tree. If the node is full (determined by the order), it splits the node and then inserts the key in the correct position.

Time Complexity: $O(\log n)$, each insertion may require traversing from root to leaf and possibly splitting nodes.

Space Complexity: $O(1)$, no added space required.

```
def insert(self, key):
    root = self.root
    if len(root.keys) == 2 * self.order - 1:
        temp = BTreeNode(self.order)
        temp.children.insert(0, self.root)
        self._split_child(temp, 0)
        self._insert_non_full(temp, key)
        self.root = temp
    else:
        self._insert_non_full(root, key)
```

Search Function

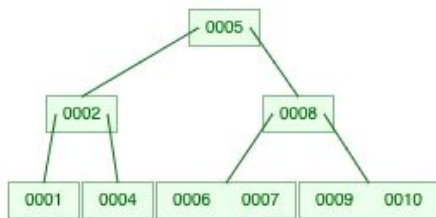
Searches for a key by traversing down the tree nodes. It efficiently narrows down the search path at each node.

Time Complexity: $O(\log n)$, the search operation traverses from root to leaf, quickly narrowing down the search space.

Space Complexity: $O(1)$, no added space required.

```
def search(self, k, x=None):
    """Search key k in the tree"""
    if not x:
        x = self.root
    i = 0
    while i < len(x.keys) and k > x.keys[i]:
        i += 1
    if i < len(x.keys) and k == x.keys[i]:
        return (x, i)
    if x.is_leaf:
        return None
    return self.search(k, x.children[i])
```

For example, let's create a B-Tree from ids {5, 6, 2, 1, 8, 10, 9, 4, 7} and verify that id 7 is in the tree, like the example at the start:



```
btree = BTree(3)
btree.insert(5)
btree.insert(6)
btree.insert(1)
btree.insert(8)
btree.insert(10)
btree.insert(9)
btree.insert(4)
btree.insert(7)
found = btree.search(7)
print("Found" if found else "Not Found")
# Found
```

Notice how, for this example, **the B-Tree is so much more efficient since it balances the tree automatically**, which keeps the height as low as possible and makes insertion and searching simpler.

In large databases with millions of customers, just think about how efficiently you could store their information and retrieve it with this structure instead of simple linear arrays or lists.

BSTs are best suited for **in-memory data storage with frequent updates** where data is relatively balanced or re-balancing is feasible.

B-Trees excel in handling disk-based storage systems where minimizing read/write operations is crucial due to their high I/O cost. **They are the backbone of many database systems and file systems due to their ability to handle large data sets efficiently.**

Iterative vs Recursive Heap Implementations using Binary Tree

Understanding both iterative and recursive heap implementations is crucial for software engineers and data scientists to **grasp underlying algorithmic strategies and memory management**. Iterative methods give insight into **explicit stack management and control flow**, while recursive methods highlight the natural **simplicity of certain algorithms and reliance on the call stack**, we'll learn about trade-offs between readability, performance, and stack memory usage.

Iterative Heap Implementation

Iterative heapify up function: Adjusts the heap from bottom to top, ensuring the **parent node is always smaller than its children** by iteratively swapping the newly added element with its parent when the parent is larger. Notice the stack.

```
def _heapify_up_iterative(self, index):  
    # Iterative approach using a stack to track parent indices  
    stack = []  
    while index != 0:  
        parent_index = (index - 1) // 2  
        stack.append(parent_index)  
        if self.heap[parent_index] > self.heap[index]:  
            self.heap[parent_index], self.heap[index] = self.heap[index], self.heap[parent_index]  
            index = parent_index  
        else:  
            break
```

Iterative heapify down function: After removing the root, it moves the last element to the top and iteratively swaps it with the smaller of its children to maintain the min heap order.

```
def _heapify_down_iterative(self, index):  
    while (index * 2 + 1) < len(self.heap):  
        smallest_child_index = index * 2 + 1  
        if (index * 2 + 2) < len(self.heap) and self.heap[index * 2 + 2] < self.heap[smallest_child_index]:  
            smallest_child_index = index * 2 + 2  
        if self.heap[index] > self.heap[smallest_child_index]:  
            self.heap[index], self.heap[smallest_child_index] = self.heap[smallest_child_index], self.heap[index]  
            index = smallest_child_index  
        else:  
            break
```


Recursive Heap Implementation

Recursive heapify up function: Recursively ensures that **each child node is larger than its parent**, effectively bubbling up the smaller values to maintain the min heap structure. Notice no stack is used.

```
def _heapify_up_recursive(self, index):  
    if index == 0:  
        return  
    parent_index = (index - 1) // 2  
    if self.heap[parent_index] > self.heap[index]:  
        self.heap[parent_index], self.heap[index] = self.heap[index], self.heap[parent_index]  
        self._heapify_up_recursive(parent_index)
```

Recursive heapify down function: Post-removal of the root, it places the last element at the top and recursively swaps it down the heap with its smallest child to preserve the min heap property.

```
def _heapify_down_recursive(self, index):
    smallest = index
    left_child = 2 * index + 1
    right_child = 2 * index + 2

    if left_child < len(self.heap) and self.heap[left_child] < self.heap[smallest]:
        smallest = left_child
    if right_child < len(self.heap) and self.heap[right_child] < self.heap[smallest]:
        smallest = right_child
    if smallest != index:
        self.heap[index], self.heap[smallest] = self.heap[smallest], self.heap[index]
        self._heapify_down_recursive(smallest)
```

Iterative vs Recursive Implementations

Iterative

```
def insert(self, element):  
    self.heap.append(element)  
    self._heapify_up_iterative(len(self.heap) - 1)
```

```
def retrieve(self):  
    if self.heap:  
        value = self.heap[0]  
        if len(self.heap) > 1:  
            self.heap[0] = self.heap.pop()  
            self._heapify_down_iterative(0)  
        else:  
            self.heap.pop()  
        return value  
    return None
```

Recursive

```
def insert(self, element):  
    self.heap.append(element)  
    self._heapify_up_recursive(len(self.heap) - 1)
```

```
def retrieve(self):  
    if self.heap:  
        value = self.heap[0]  
        self.heap[0] = self.heap[-1]  
        self.heap.pop()  
        self._heapify_down_recursive(0)  
        return value  
    return None
```

Iterative

Time Complexity: $O(\log n)$ for insert, $O(\log n)$ for retrieve.

Space Complexity: $O(1)$ since it uses an in-place algorithm without additional space overhead.

Recursive

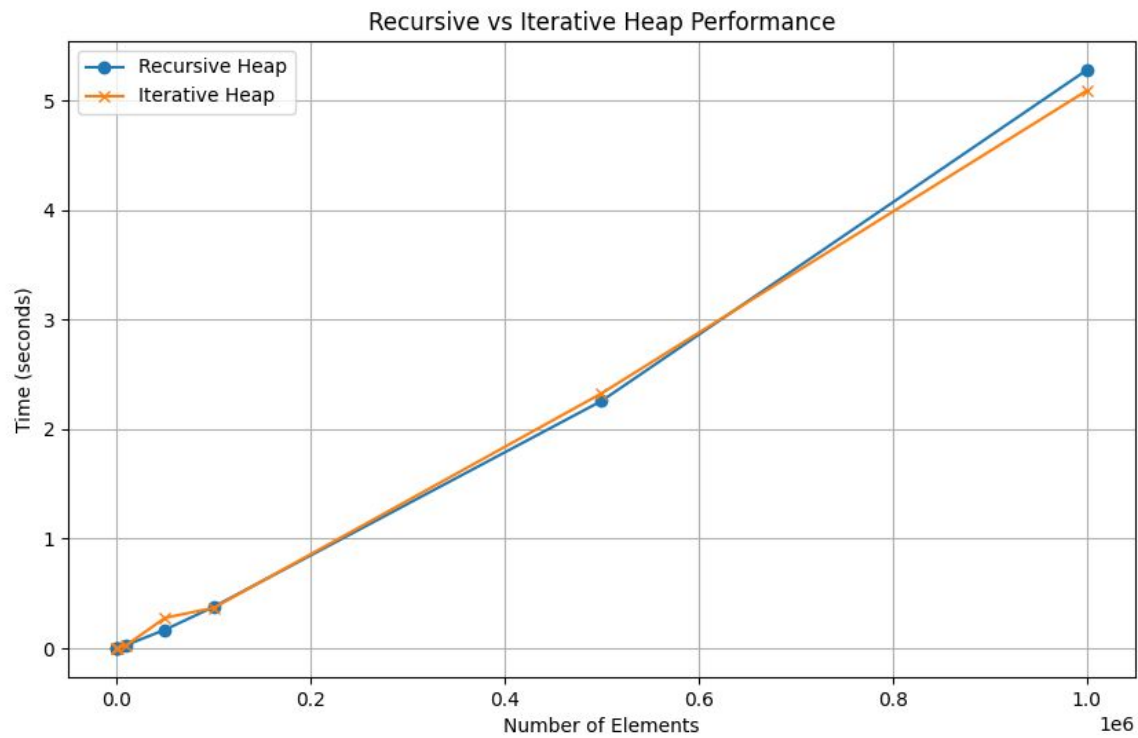
Time Complexity: $O(\log n)$ for insert, $O(\log n)$ for retrieve.

Space Complexity: $O(\log n)$ due to the call stack used in recursive calls, which grows with the height of the heap.

When to use each?

Iterative Implementation: Favor this approach when working with **large datasets** or in environments with **limited stack size** to avoid potential stack overflow errors. It is also **generally considered more space-efficient due to constant space complexity**.

Recursive Implementation: Choose this for **better readability and ease of understanding**, especially when the depth of recursion (and thus the size of the heap) is not prohibitively large. It is typically **more straightforward to implement and can be more natural to reason about**.



To exemplify that what the Heap implemented with a Binary Tree does we could use a [Min Heap Visualiser](#). By tapping “BuildHeap” we could see how the heap sorts to maintain the min heap structure.

To see what retrieve does we could tap “Remove Smallest” and similarly we could add any element we want.

Notice how the tree automatically balances to make the operations more efficient.

Worked Example

You are building a task scheduler for a computer system. **Tasks are being scheduled based on their priority level, with the most critical tasks being executed first.**

1. How would you insert a new task with a given priority into the scheduler using a min heap to ensure the most critical task is addressed first?
2. After several tasks are scheduled, how would you retrieve the next task to be executed, and what is the space complexity of this operation using the iterative approach?

Worked Example

You are building a task scheduler for a computer system. **Tasks are being scheduled based on their priority level, with the most critical tasks being executed first.**

1. How would you insert a new task with a given priority into the scheduler using a min heap to ensure the most critical task is addressed first?

You would use the insert function of the heap to add the new task. The heapify up process would then ensure that the task is placed in the correct position to maintain the min heap property.

2. After several tasks are scheduled, how would you retrieve the next task to be executed, and what is the space complexity of this operation using the iterative approach?

You would use the retrieve function to get the next task, which is the root of the min heap. The space complexity for this operation in an iterative approach is $O(1)$ as it does not require additional space beyond the heap itself.

Summary

Binary Trees

- ★ A tree structure where each node has at most two children
- ★ Time Complexity (Insert/Search in an unbalanced tree): $O(n)$
- ★ Space Complexity: $O(n)$

Binary Search Trees

- ★ A binary tree where each node's key is greater than all keys in its left subtree and less than all keys in its right subtree.
- ★ Time Complexity (Insert/Search in an unbalanced tree): $O(n)$
- ★ Space Complexity: $O(n)$

Summary

Rose Trees

- ★ A tree where each node can have an arbitrary number of children, ideal for representing complex hierarchical structures
- ★ Time Complexity (Insert/Search): $O(n)$
- ★ Space Complexity: $O(n)$

B-Trees

- ★ A balanced tree data structure used in databases and file systems; a specialized form of a rose tree
- ★ Time Complexity (Insert/Search): $O(\log n)$
- ★ Space Complexity: $O(1)$

Summary

Iterative vs Recursive Heap Implementations using Binary Tree



- ★ Both approaches have the same time and space complexities ($O(\log n)$ for time complexity in insert and retrieve, and $O(\log n)$ for space complexity in the iterative approach due to stack usage)

Application Scenarios

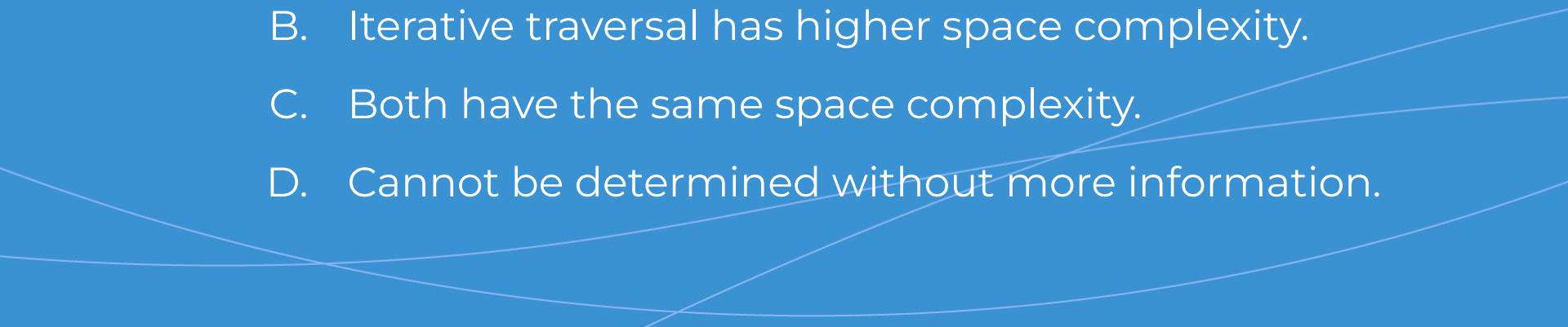
- ★ BSTs are suited for in-memory data storage with frequent updates, whereas B-Trees excel in disk-based storage systems where minimising read/write operations is crucial


Further Learning

- [Programiz](#) - Binary Tree Tutorial
- [VISUALGO](#) - Tree Visualisation Tool
- [Simplilearn](#) - Data Structure and Algorithm Complexity Guide
- [Binari](#) - Interactive Binary and Rose Tree Learning




How does the space complexity of a recursive traversal of a binary tree compare to an iterative traversal using a stack?

- A. Recursive traversal has higher space complexity.
 - B. Iterative traversal has higher space complexity.
 - C. Both have the same space complexity.
 - D. Cannot be determined without more information.
- 



In a project management tool, which data structure would be most efficient for quickly retrieving the highest-priority task?

- A. Binary Search Tree
 - B. Min Heap
 - C. Rose Tree
 - D. Stack
- 



Questions and Answers

Questions around Trees

