Self-Study Report: Binary Tree Data Structure

A **Binary Tree Data Structure** is a hierarchical data structure in which each node has at most two children, referred to as the left child and the right child. It is commonly used for the efficient storage and retrieval of data,

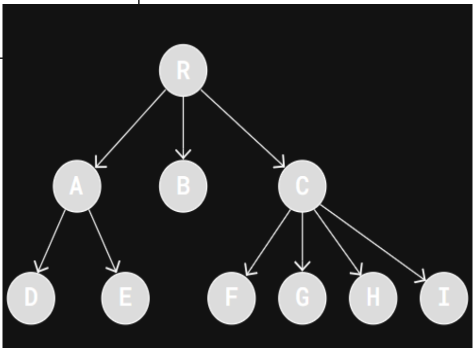
Here are some basic terminologies to better understand a binary tree

- Node: The fundamental part of a binary tree that contains data and links to its children.
- Root: Top node of the binary tree
- Leaf: Node that does not have any children
- Height: The length of the longest path from the root to a leaf
- Depth: The length of the path from the root to a specific node.
- Subtree: A tree consisting of a node and its descendants.

Binary trees are versatile data structures, and here are their main methods and operations:

1. **Insertion**: Add nodes to the tree while maintaining its structure and properties. For example, inserting a value in a binary search tree ensures that smaller values go to the left and larger ones to the right.
2. **Deletion**: Remove nodes from the tree. This can involve reorganizing the tree to preserve its properties.
3. **Traversal**: Access and process nodes in the tree in a specific order. Common traversal methods include:
    - **Inorder Traversal**: Left → Root → Right
    - **Preorder Traversal**: Root → Left → Right
    - **Postorder Traversal**: Left → Right → Root
4. **Search**: Locate a specific node or value within the tree, typically using recursive methods or comparison.
5. **Finding Min/Max**: Retrieve the smallest or largest value in the binary tree, often in a binary search tree.
6. **Balancing**: Adjust nodes to optimize tree performance (applies to AVL trees and other balanced binary trees).

The data structure is called “tree” because it looks like a tree upside down



The Tree data structure can be useful in many cases.

- Hierarchical Data: file systems, organizational models, ECT
- Databases: Used for quick data retrieval
- Routing Tables: Used for routing data in network algorithms
- Sorting/Searching: used for sorting data or searching for data
- Priority Queue: priority queue data structures are commonly implemented using trees

Up above we can see a binary tree, lets do a quick analysis on the tree

The circles are called nodes and the root node is R. the leaf nodes are D,E,B,F,G,H,I they are nodes that do not have any child nodes. The child nodes are everything except for node R as they are a subnode of the parent node. The parent nodes are A,R,C they are the parent of the child subnodes.

The nodes are connected by edges that show they have a relationship between each other.

The tree size is (n=10) because there are a total of 10 nodes. The tree height is

There are multiple types of trees including

Binary Tree: each node has up to two children, the left child node and the right child node. This structure is the foundation for more complex tree types like binary Search Trees and AVL Trees.

Binary Search Trees (BST's): A type of Binary Tree where for each node, the left child node has a lower value, and the right child node has a higher value.

AVL Trees: A type of Binary Search Tree that self-balances so that for every node, the difference in height between the left and right subtrees is at most one. This balance is maintained through rotations when nodes are inserted or deleted.

There are many different applications for the usage of binary trees. The one I want to focus on right now is the Binary Search Tree (BST). The BST is an extension of the binary tree. It's a binary tree with internal ordering. It has a unique binary-search-tree property where the left child of a given node must be smaller than or equal to its parent node, while the right child should be greater than or equal to its parent node. Binary search Trees (BST) are widely used in various applications due to their efficient search, insertion, and deletion operations. Some common uses for a binary search Tree include:

1) Searching for Data
    - BSTs allow for efficient searching of elements. The average time complexity for search operations is (O(\log n)), making it faster than linear search methods.
2) Dynamic set operations
    - BSTs support dynamic set operations such as insertion, deletion, and lookup. This is useful in applications where the dataset changes frequently.
3) Sorted Data Retrieval
    - In-order traversal of a BST retrieves elements in sorted order. This property is useful for applications that require sorted data without needing to sort it explicitly.
4) Implementing Associate Arrays
    - BSTs can be used to implement associative arrays or maps, where keys are stored in the tree, allowing for efficient key-value pair retrieval.
5) Priority Queues
    - While heaps are more commonly used for priority queues, BSTs can also be used to implement priority queues, allowing for efficient retrieval of the minimum or maximum element.
6) Database indexing
    - Many database systems use BSTs or variations (like B-trees) for indexing data, allowing for quick access to records based on key values.

Binary Search Trees are versatile data structures that provide efficient solutions for a variety of problems involving dynamic datasets, sorted data retrieval, and associative arrays. Their logarithmic time complexity for key operations makes them a popular choice in many applications across different domains.

CONCLUSION

A binary tree is a fundamental data structure that provides a way to organize and manage data hierarchically. Its properties and traversal methods make it suitable for a wide range of applications.

Self-Study Report: Shell Sort

Introduction

Shell Sort, as the name indicates, is a comparison-based sorting algorithm and a variant of insertion sort named after its creator Donald Shell. Shell Sort is an extension of insertion sort with the additional facility of swapping elements that are far from one another with the least amount of overall shifts or swaps. Shell Sort can efficiently sort medium-sized data and is simple to implement.

Algorithm Overview

Shell Sort works by arranging the data sequence into subsets (or "gaps") and sorting those subsets using insertion sort. The gap size is gradually reduced until it becomes 1, at which point a final pass of insertion sort ensures the sequence is fully sorted.

Steps of Shell Sort

1. Choose a gap sequence: Start with a large gap and progressively reduce it (e.g., divide the gap by 2 in each iteration).
2. Sort subsets for each gap: Insertion sort for elements far apart by the gap.
3. Reduce the gap: Keep repeating the above until a gap of 1 is obtained.
4. Final pass: Perform a standard insertion sort with a gap of 1.

The idea here is that sorting elements far apart (with large gaps) reduces the number of swaps to be performed in the final insertion sort.

Time Complexity Analysis

The time complexity of Shell Sort depends on the choice of the gap sequence, as it affects the efficiency of sorting within subsets. Commonly used gap sequences include:

1. Original Shell's Sequence (gap = n/2, n/4, ..., 1)
2. Knuth's Sequence (gap = 3k - 1, when k is a positive integer)
3. Hibbard's Sequence (gap = 2k - 1)

- Best Case: O(n log n) (using optimized gap sequences)
- Average Case: $O(n^{1.25})$ to $O(n^{1.5})$, depending on the gap sequence
- Worst Case: $O(n^2)$(with poor gap sequences like Shell's original)

Shell Sort is not as efficient as modern algorithms like Merge Sort or Quick Sort for large datasets, but it performs well for smaller or moderately sized arrays.

How shell sort works:

For example:

The initial list = [2 , 8, 36, 4, 45]

1. First, Gap = 2
   - i. 2 compare to 36, 2 < 36 , remain in same [2, 8, 36, 4, 45]
   - ii. 8 compare to 4, 8>4 , swap [2, 4, 36, 8, 45]
   - iii. 36 compare to 45, remain in same [2, 4, 36, 8, 45]
2. Then, Gap = 1
   - i. 2 compare to 4, 2<4 , remain in same [2, 4, 36, 8, 45]
   - ii. 4 compare to 36, 4<36, remain in same [2, 4, 36, 8, 45]
   - iii. 36 compare to 8, 36>8, swap [2, 4, 8, 36, 45]

The array is sorted now [2, 4, 8, 36, 48]

Shell Sort Advantages

1. Fast for mid-sized arrays: Since less shifting is needed, Shell Sort is quicker than insertion sort.

2. In-place sorting: No additional memory space is needed, and thus space-efficient (O(1) space complexity).

3. Simple implementation: Simple to write and use.

Disadvantages

1. Not stable: Shell Sort does not keep the relative ordering of equal elements.

2. Worst case for large datasets: It has poor performance with time complexity in comparison to well-designed algorithms such as Quick Sort or Merge Sort.

Conclusion

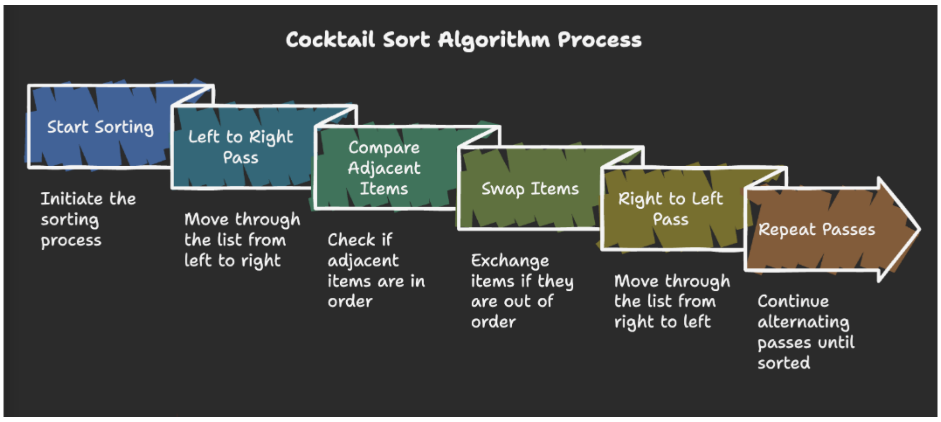
Shell Sort is a fast and good sorting algorithm for small to medium-sized data sets where memory and simplicity are concerns. While it is less suited for big data sets, it can be significantly improved with well-constructed gap sequences.

Introduction:

As part of this self-study, I have chosen to investigate the Cocktail Sort algorithm, also known as bidirectional bubble sort or shaker sort. This algorithm is a variation of the fundamental Bubble Sort and offers a slightly optimized approach by sorting in both directions during each pass. This report will go through into the details of Cocktail Sort to provide a comprehensive understanding of its mechanics and performance.

Algorithm Description and Working Principle:

Cocktail Sort is a comparison-based sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The passes through the list are done alternately from left to right and from right to left.


Cocktail Sort Algorithm Process
Start Sorting
Initiate the sorting process
Left to Right Pass
Move through the list from left to right
Compare Adjacent Items
Check if adjacent items are in order
Swap Items
Exchange items if they are out of order
Right to Left Pass
Move through the list from right to left
Repeat Passes
Continue alternating passes until sorted

The algorithm proceeds as follows:

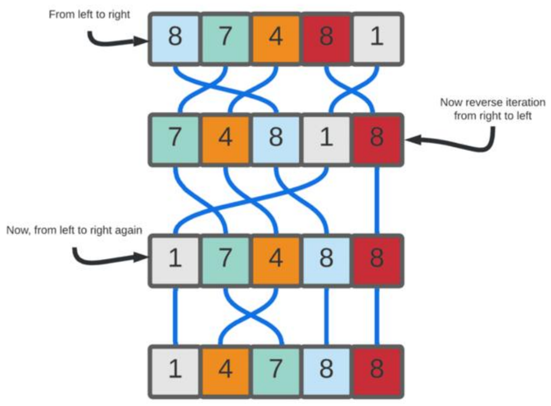
1. **Left-to-Right Pass (Bubble Up):** Starting from the beginning of the list, compare each element with its next neighbor. If the current element is greater than the next, swap them. This pass moves the largest unsorted element to its correct position at the end of the unsorted portion.
2. **Right-to-Left Pass (Sink Down):** After the left-to-right pass, start from the end of the list (excluding the last already sorted element) and move towards the beginning. Compare each element with its previous neighbor. If the current element is smaller than the previous, swap them. This pass moves the smallest unsorted element to its correct position at the beginning of the unsorted portion.
3. Repeat steps 1 and 2, each time reducing the range of elements being considered (as the smallest and largest elements are placed correctly in each bidirectional pass).
4. The algorithm terminates when a full bidirectional pass occurs without any swaps, indicating that the list is sorted.

Examples of How the Algorithm Works:

To illustrate the algorithm's operation, let's use the example array: [5, 1, 4, 2, 8, 0]

Pass 1:

- **Left to Right:**
    - [5, 1, 4, 2, 8, 0] -> [1, 5, 4, 2, 8, 0] (swap 5 and 1)
    - [1, 5, 4, 2, 8, 0] -> [1, 4, 5, 2, 8, 0] (swap 5 and 4)
    - [1, 4, 5, 2, 8, 0] -> [1, 4, 2, 5, 8, 0] (swap 5 and 2)
    - [1, 4, 2, 5, 8, 0] -> [1, 4, 2, 5, 0, 8] (swap 8 and 0)


From left to right
Now reverse iteration from right to left
Now, from left to right again
8 7 4 8 1
7 4 8 1 8
1 7 4 8 8
1 4 7 8 8

- **Right to Left:**
    - [1, 4, 2, 5, 0, 8] -> [1, 4, 2, 0, 5, 8] (swap 5 and 0)
    - [1, 4, 0, 2, 5, 8] (swap 2 and 0)
    - [1, 0, 4, 2, 5, 8] (swap 4 and 0)
    - [0, 1, 4, 2, 5, 8] (swap 1 and 0)

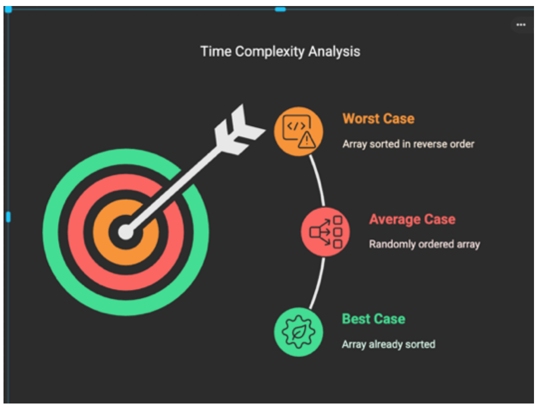
Pass 2:

- **Left to Right (excluding first and last):**
    - [0, 1, 4, 2, 5, 8] -> [0, 1, 2, 4, 5, 8] (swap 4 and 2)
    - [0, 1, 2, 4, 5, 8] (no swap between 4 and 5)
- **Right to Left (excluding first and last):**
    - [0, 1, 2, 4, 5, 8] (no swap between 5 and 4)
    - [0, 1, 2, 4, 5, 8] (no swap between 4 and 2)

Since no swaps occurred in the second full bidirectional pass, the array is sorted: [0, 1, 2, 4, 5, 8]

Analysis of Time Complexity:

An analysis of the time complexity for the chosen sorting algorithm is given below:

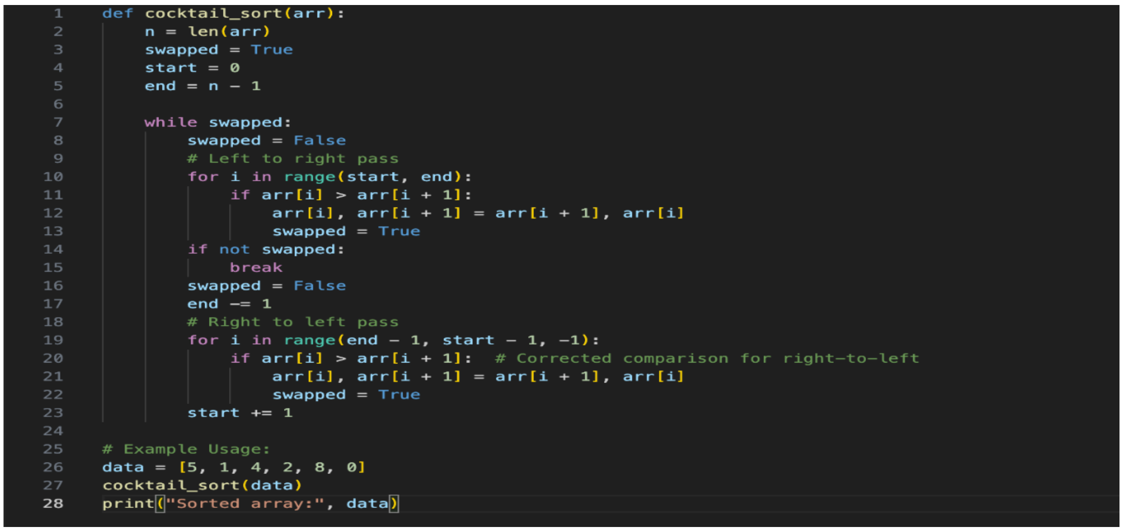


- **Best Case:** O(n) - This occurs when the input array is already sorted. In this scenario, the algorithm will perform one full bidirectional pass without any swaps, and the swapped flag will cause the loop to terminate.

- **Average Case:** O(n^2) - For a randomly ordered array, the number of comparisons and swaps is typically proportional to the square of the number of elements.

- **Worst Case:** O(n^2) - This occurs when the array is sorted in reverse order. Similar to Bubble Sort, every pair of adjacent elements might need to be compared and swapped multiple times.

<u>Code Snippets and Explanation:</u>



```python
def cocktail_sort(arr):
    n = len(arr)
    swapped = True
    start = 0
    end = n - 1

    while swapped:
        swapped = False
        # Left to right pass
        for i in range(start, end):
            if arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
        if not swapped:
            break
        swapped = False
        end -= 1
        # Right to left pass
        for i in range(end - 1, start - 1, -1):
            if arr[i] > arr[i + 1]:  # Corrected comparison for right-to-left
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
        start += 1

# Example Usage:
data = [5, 1, 4, 2, 8, 0]
cocktail_sort(data)
print("Sorted array:", data)
```

Explanation:

The Python code defines a function cocktail sort that takes an array(arr) as input.

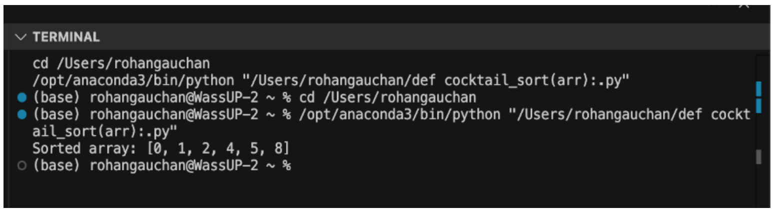
- n stores the length of the array.

- swapped is a Boolean flag that tracks if any swaps occurred during a pass. If no swaps occur in a full bidirectional pass, it indicates that the array is sorted, and the algorithm can terminate.

- start and end variables define the portion of the array that is currently being considered for sorting. In each pass, the sorted elements at the beginning and end are effectively excluded by adjusting these pointers.

- The while swapped: loop continues as long as the swapped flag is true from the previous pass.

- The first for loop iterates from start to end (left to right), performing the bubble-up operation. If adjacent elements are in the wrong order, they are swapped, and the swapped flag is set to True.
- The if not swapped: break statement checks if any swaps occurred in the left-to-right pass. If not, the array is sorted, and the loop breaks.
- The end pointer is decremented because the largest element in the current unsorted portion has now reached its correct position at the end.
- The second for loop iterates from end - 1 down to start (right to left), performing the sink-down operation. If adjacent elements are in the wrong order, they are swapped, and the swapped flag is set to True.
- The start pointer is incremented because the smallest element in the current unsorted portion has now reached its correct position at the beginning.



```
cd /Users/rohangauchan
/opt/anaconda3/bin/python "/Users/rohangauchan/def cocktail_sort(arr):.py"
● (base) rohangauchan@WassUP-2 ~ % cd /Users/rohangauchan
● (base) rohangauchan@WassUP-2 ~ % /opt/anaconda3/bin/python "/Users/rohangauchan/def cockt
ail_sort(arr):.py"
Sorted array: [0, 1, 2, 4, 5, 8]
○ (base) rohangauchan@WassUP-2 ~ %
```

The example usage demonstrates how to call the cocktail sort function with an unsorted array, and the output will be the sorted array shown in the picture.

Conclusion:

In summary, this self-study report has thoroughly explored the Cocktail Sort algorithm, covering its operation, time complexity, development, and a practical example. Studying Cocktail Sort has provided a clear understanding of this sorting method and its relationship to Bubble Sort. Furthermore, working on the Python implementation has significantly improved my understanding and practical skills in Python programming.

Self-Study Report: Heap Sort Algorithm

1. Introduction

In this self-study report, I explore the **Heap Sort** algorithm, which was not covered in the course syllabus. While the course introduced fundamental data structures such as stacks, queues, linked lists, and binary trees, the Heap stands out for its application in **priority-based operations** and efficient **sorting mechanisms**.

Heap Sort is a **comparison-based sorting algorithm** that leverages a specialized **binary heap data structure**. It is an improvement over selection sort, where we repeatedly select the maximum element and move it to the end of the array. What makes Heap Sort efficient is that finding the maximum element is done in **O(log n)** time using a **binary heap**, resulting in an overall time complexity of **O(n log n)**.

Heap Sort is commonly used in **system-level applications**, **priority queues**, and scenarios where **worst-case performance predictability** and **low memory usage** are essential.

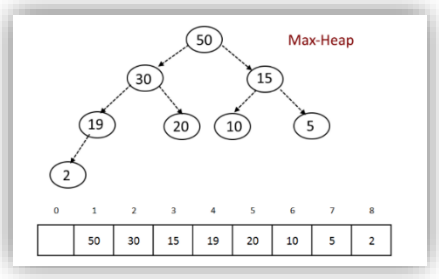
2. Understanding the Heap Data Structure

A **Heap** is a special tree-based data structure that satisfies the **heap property**. The heap property can be defined as:

- **Max-Heap**: The value of each parent node is greater than or equal to the values of its children.
- **Min-Heap**: The value of each parent node is less than or equal to the values of its children.

In a heap, the highest (or lowest) value is always at the root, which makes heaps useful for **priority queues**. For example, Heap sort uses a Max Heap to sort the array in ascending order.

Heap Representation:



A binary heap is a complete binary tree, meaning all levels are filled except possibly the last, which is filled from left to right.

Heaps are typically implemented using arrays, where:

- Parent at index i
- Left child at 2*i + 1
- Right child at 2*i + 2

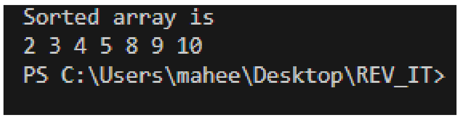
For an array of size **n**, the root is at index **0**, the left child of an element at index **i** is at **2i + 1**, and the right child is at **2i + 2**. This diagram is an example of a max Heap.

3. Steps Involved in Heap Sort

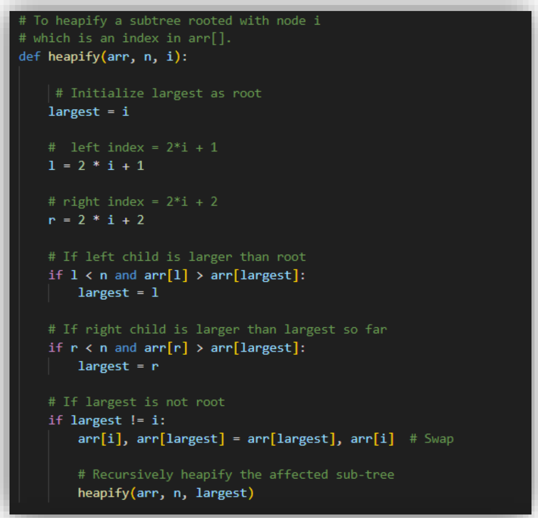
1. **Build a Heap**: Convert the input array into a heap (max-heap for ascending order or min-heap for descending order).
2. **Extract Max/Min**: The largest (or smallest) element will be at the root of the heap. Swap it with the last element of the heap, then reduce the heap size.
3. **Heapify**: After extracting the root, the heap property must be restored. This process is called heapifying the heap.
4. **Repeat**: Repeat the extraction and heapify steps until the heap is empty.

4. Implementation of Heap Sort in Python

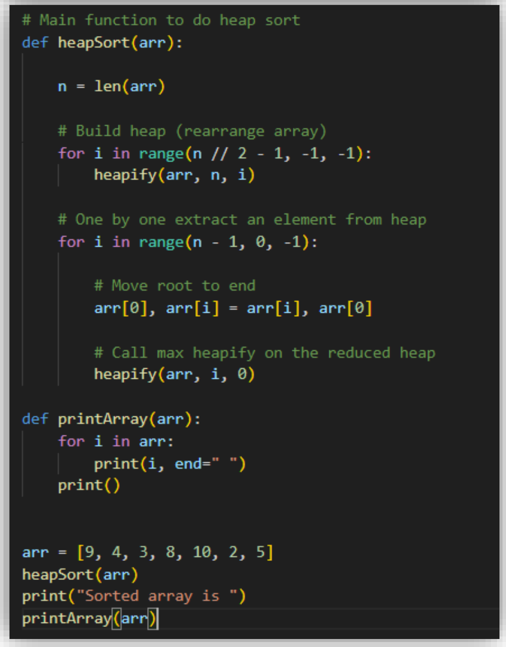
Output is:



```
Sorted array is
2 3 4 5 8 9 10
PS C:\Users\mahee\Desktop\REV_IT>
```



```python
# To heapify a subtree rooted with node i
# which is an index in arr[].
def heapify(arr, n, i):

    # Initialize largest as root
    largest = i

    #  left index = 2*i + 1
    l = 2 * i + 1

    # right index = 2*i + 2
    r = 2 * i + 2

    # If left child is larger than root
    if l < n and arr[l] > arr[largest]:
        largest = l

    # If right child is larger than largest so far
    if r < n and arr[r] > arr[largest]:
        largest = r

    # If largest is not root
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # Swap

        # Recursively heapify the affected sub-tree
        heapify(arr, n, largest)
```



```python
# Main function to do heap sort
def heapSort(arr):

    n = len(arr)

    # Build heap (rearrange array)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # One by one extract an element from heap
    for i in range(n - 1, 0, -1):

        # Move root to end
        arr[0], arr[i] = arr[i], arr[0]

        # Call max heapify on the reduced heap
        heapify(arr, i, 0)

def printArray(arr):
    for i in arr:
        print(i, end=" ")
    print()


arr = [9, 4, 3, 8, 10, 2, 5]
heapSort(arr)
print("Sorted array is ")
printArray(arr)
```

1. The heapify() function:
- Ensures that the property is maintained, since for a max heap, the parent must be greater than both children.
- It checks if the current node (i) is the largest compared to its children. If it's not, it swaps the current node with the largest child and recursively calls itself to fix the sub-tree rooted at that child.

2. The heapsort() function:
- It first builds a Max Heap from the unsorted input. This is done by calling heapify() starting from the non-leaf node (index n//2 - 1) up to the root.
- Then, it repeatedly swaps the root of the heap (the largest element) with the last element in the array.
- After each swap, it reduces the heap size by 1 and calls heapify() again to restore the heap property.

3. Lastly The printArray() function:
- Prints the final sorted array

5. Time Complexity of Heap Sort

- **Building the Heap**: The time complexity for building the heap is **O(n)**.

- **Extracting Elements**: The time complexity for extracting each element from the heap is **O(log n)**. Since we perform the extraction for all elements, the overall time complexity for the sorting process is **O(n log n)**.

Therefore, the **overall time complexity of Heap Sort** is **O(n log n)**, making it more efficient than algorithms like bubble sort and selection sort, which have a time complexity of **O(n^2)**.

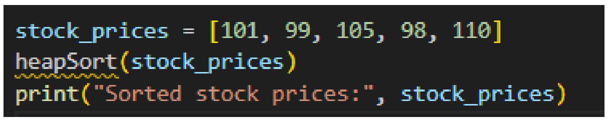
Space Complexity:

Auxiliary Space: Typically O(log n) due to recursion.

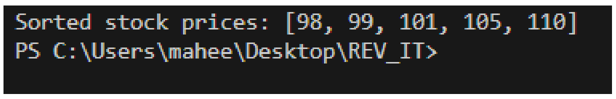
However, it can be reduced to O(1) using an iterative version of heapify(), making Heap Sort an in-place sorting algorithm.

6. **Applications of Heaps**

- **Priority Queue**: Heaps are widely used in priority queues, where the highest (or lowest) priority element is always at the front.
- **Graph Algorithms**: Heaps are used in Dijkstra's and Prim's algorithms for finding the shortest path and minimum spanning tree, respectively.
- **Sorting Stock Prices**: In a stock monitoring system (like the one we developed in our group project), we might want to sort stock prices for analysis or display.



```
stock_prices = [101, 99, 105, 98, 110]
heapSort(stock_prices)
print("Sorted stock prices:", stock_prices)
```



```
Sorted stock prices: [98, 99, 101, 105, 110]
PS C:\Users\mahee\Desktop\REV_IT>
```

7. Advantages and Disadvantages of Heap Sort:

Advantages:

- Consistent Time Complexity: O(n log n) across all cases.
- In-Place: which means it does not require additional memory.
- Simple to implement and understand compared to Quick Sort, Merge Sort, and other equally efficient sorting algorithms since it does not have recursion concepts.

Disadvantages:

- Not Stable: Equal elements may not retain their original order.
- Less Efficient in Practice: More swaps and lower performance compared to Quick Sort.
- Larger constant factors make it slower for small arrays.

8. Conclusion

Heap Sort is a powerful, efficient, and memory-friendly sorting algorithm. Its performance is consistent across all input types, making it great for real-time applications. While not the fastest compared to Quick Sort, it's still used in systems requiring priority management (e.g., task schedulers).

Self-Study Report: Matrix Data Structure

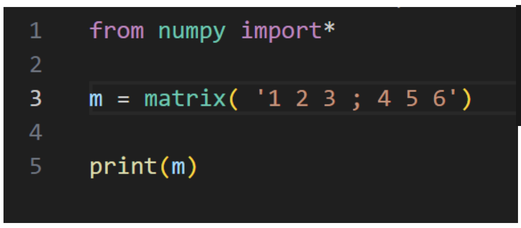
Introduction

A two-dimensional array of numbers set up in rows and columns is called a matrix. It is a basic data structure in mathematics and computer science that is widely used to solve equation systems, carry out operations in linear algebra, and represent linear transformations. In many fields, such as scientific computing, machine learning, and graphics, matrices are crucial.

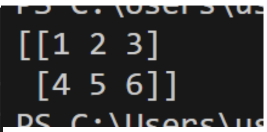
Key Characteristics of Matrices

- **Dimensions**: A matrix is defined by its dimensions, denoted as m×nm \times nm×n, where mmm is the number of rows and nnn is the number of columns.
- **Element Access**: Elements in a matrix can be accessed using two indices: A[i][j]A[i][j]A[i][j] refers to the element located at the ithi^{th}ith row and jthj^{th}jth column.
- **Types**: Matrices can be classified as square (same number of rows and columns), rectangular, sparse (majority of elements are zero), and dense (majority of elements are non-zero).

Some of the Commonly Used Method



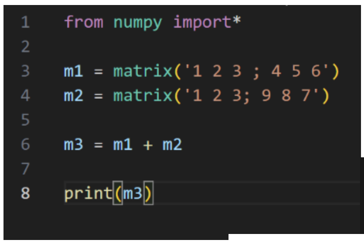
```
from numpy import*

m = matrix( '1 2 3 ; 4 5 6')

print(m)
```



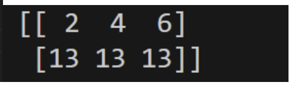
```
[[1 2 3]
 [4 5 6]]
```

We can use Numpy libraries to access matrix. Then we make the matrix as it’s displayed in the code. And to differentiate the row. The users can “;” to put them in a row.

Addition & Subtraction



```python
from numpy import*

m1 = matrix('1 2 3 ; 4 5 6')
m2 = matrix('1 2 3; 9 8 7')

m3 = m1 + m2

print(m3)
```

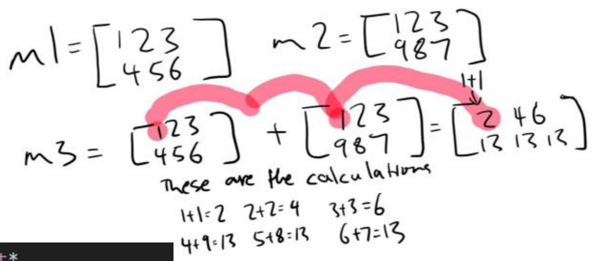


```
[[ 2  4  6]
 [13 13 13]]
```

Now Let’s Make another matrix with the variable name m2.

Then we add m1 with m2 and the outcome will be m3.

The process behind the code;

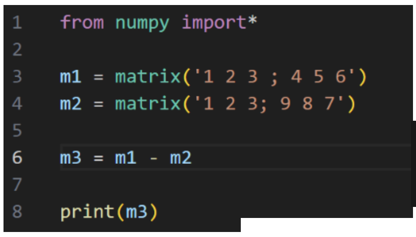


$$m1 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad m2 = \begin{bmatrix} 1 & 2 & 3 \\ 9 & 8 & 7 \end{bmatrix}$$

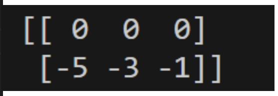
$$m3 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 1 & 2 & 3 \\ 9 & 8 & 7 \end{bmatrix} = \begin{bmatrix} \overset{1+1}{2} & 4 & 6 \\ 13 & 13 & 13 \end{bmatrix}$$

These are the calculations

1+1=2 2+2=4 3+3=6
4+9=13 5+8=13 6+7=13



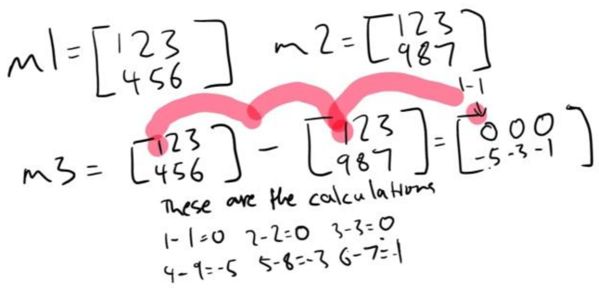
```python
from numpy import*

m1 = matrix('1 2 3 ; 4 5 6')
m2 = matrix('1 2 3; 9 8 7')

m3 = m1 - m2

print(m3)
```



```
[[ 0  0  0]
 [-5 -3 -1]]
```

Now Let’s take the same two matrix and we minus m1 with m2.

The process behind the code;



$$m1 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad m2 = \begin{bmatrix} 1 & 2 & 3 \\ 9 & 8 & 7 \end{bmatrix}$$

$$m3 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} - \begin{bmatrix} 1 & 2 & 3 \\ 9 & 8 & 7 \end{bmatrix} = \begin{bmatrix} \overset{1-1}{0} & 0 & 0 \\ -5 & -3 & -1 \end{bmatrix}$$

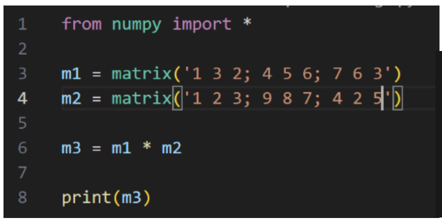
These are the calculations

1-1=0 2-2=0 3-3=0
4-9=-5 5-8=-3 6-7=-1

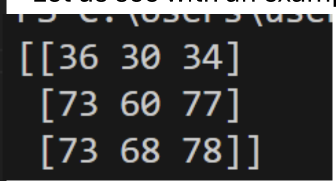
Note: subtracting is actually defined as the **addition** of a negative matrix: A + (−B)

Multiplication

Multiplying Matrix with another matrix. To multiply a matrix **by another matrix** we need to do the “dot product" of rows and columns ... what does that mean? Let us see with an example:

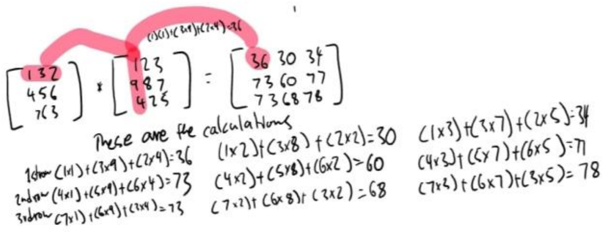


```python
from numpy import *

m1 = matrix('1 3 2; 4 5 6; 7 6 3')
m2 = matrix('1 2 3; 9 8 7; 4 2 5')

m3 = m1 * m2

print(m3)
```



```
[[36 30 34]
 [73 60 77]
 [73 68 78]]
```

Now let’s have (3 x 3) Matrix Multiplying to each other



$$\begin{bmatrix} 1 & 3 & 2 \\ 4 & 5 & 6 \\ 7 & 6 & 3 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 9 & 8 & 7 \\ 4 & 2 & 5 \end{bmatrix} = \begin{bmatrix} 36 & 30 & 34 \\ 73 & 60 & 77 \\ 73 & 68 & 78 \end{bmatrix}$$

(1x1)+(3x9)+(2x4)=36

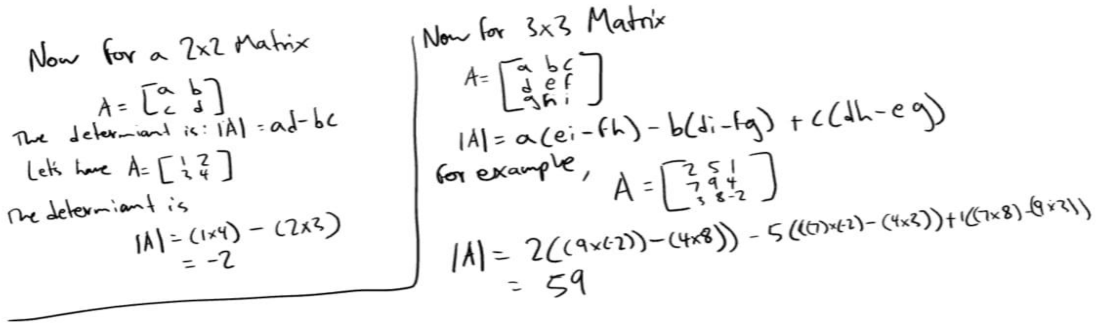
These are the calculations

1st row (1x1)+(3x9)+(2x4)=36 (1x2)+(3x8)+(2x2)=30 (1x3)+(3x7)+(2x5)=34
2nd row (4x1)+(5x9)+(6x4)=73 (4x2)+(5x8)+(6x2)=60 (4x3)+(5x7)+(6x5)=77
3rd row (7x1)+(6x9)+(3x4)=73 (7x2)+(6x8)+(3x2)=68 (7x3)+(6x7)+(3x5)=78

As you can see, to work out the answer for the 1st row and 1st column. The "Dot Product" is where we multiply matching members, then sum up.

Determinant

The determinant helps us find the inverse of a matrix, tells us things about the matrix that are useful in system of linear equation, calculus and more. When calculation all the matrix should ne square for example it should have the same amount of rows and columns.



Now for a 2x2 Matrix

$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$

The determiant is: $|A| = ad - bc$

Let's have $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

The determiant is

$|A| = (1 \times 4) - (2 \times 3)$
$= -2$

Now for 3x3 Matrix

$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$

$|A| = a(ei - fh) - b(di - fg) + c(dh - eg)$

for example, $A = \begin{bmatrix} 2 & 5 & 1 \\ 7 & 9 & 4 \\ 3 & 8 & -2 \end{bmatrix}$

$|A| = 2((9 \times (-2)) - (4 \times 8)) - 5(((7) \times (-2)) - (4 \times 3)) + 1((7 \times 8) - (9 \times 3))$
$= 59$

Development of the Matrix Data Structure

The concept of matrices dates back to ancient civilizations, where they were used for solving linear equations. In computer science, matrices gained prominence with the advent of digital computing and the need for efficient numerical methods.

- **Early Developments**: Researchers like Carl Friedrich Gauss and Évariste Galois contributed to the mathematical foundation, which laid the groundwork for algorithms to manipulate matrices.
- **Computer Algebra Systems**: In the 1960s and 1970s, the rise of computer algebra systems introduced methods for symbolic manipulation of matrices.
- **Modern Libraries**: Today, libraries such as NumPy (Python), Eigen (C++), and MATLAB offer extensive functionalities for matrix operations, making it easier for developers to work with matrices in applications ranging from data analysis to machine learning.

Conclusion

With numerous uses in data science, engineering, and mathematics, the matrix data structure is a strong and adaptable computing tool. Developers can effectively address complex issues by comprehending matrix operations and how they are implemented, especially in domains like image processing, machine learning, and numerical simulations. The use of matrices in data representation and manipulation will only increase in significance as technology advances.