Data Structures and Algorithms

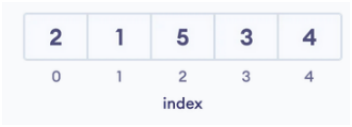
Contents

Linear data structures: 2
Non-Linear data structures:....3
Solving Big O notations:....4
Array Data Structure: 5
Stack Data Structure (LIFO):....5
Queue Data Structure (FIFO): 5
Circular Queue 6
Priority Queue....6
Deque (Double Ended Queue) 6
Linked List Data Structure:....7
Singly Linked List 7
Doubly Linked List 7
Circular Linked List 7
Hash Table: 8
Heap: 9
Binary Tree:....9
Binary Search Tree (BST): 10
AVL Tree: 10
Graph Data Structure: 11
Depth First Search (DFS):....12
Breadth First Search (BFS): 12
Binary Search: 13
Merge Sort Algorithm:....13
QuickSort Algorithm:....13
HeapSort Algorithm: 14
Dijkstra's Algorithm:....14
Dynamic Programming:....15
Backtracking Algorithm: 15

Linear data structures:

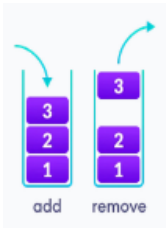
Elements are arranged in sequence one after the other, which results in an increased time complexity with the increase in data size.

i) Array:



Store data sequentially in the memory.

Elements are of the same type.



ii) Stack:

LIFO principle.

iii) Queue:

FIFO principle.

iv) Linked List:

Elements are connected through a series of nodes.

Each node contains data items and address to the next node.

v) Hash Table:

Stores elements in key-value pairs,

Key is a unique integer used for indexing the values

Value is the data associated with keys.

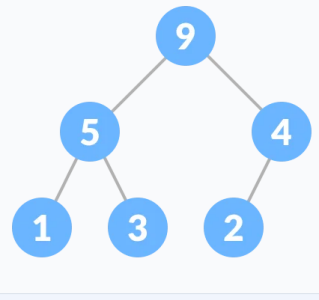
vi) Heap:

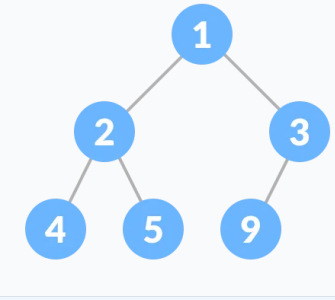
This is a complete binary tree structure that satisfies the heap property.

There are two types:

Max heap – any given node is always greater than its child and the root is the largest.

Min heap – any given node is always smaller than its child and the root is the smallest.

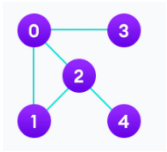




Non-Linear data structures:

Elements are not in any sequence. Arranged in a hierarchical manner, which allow quicker and easier access to the data.

i) **Graph:**



Nodes are called vertices.

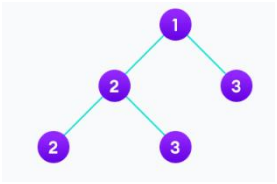
Vertices are connected through Edges.

- Spanning Tree and Minimum Spanning Tree.
- Strongly Connected Components.
- Adjacency Matrix.
- Adjacency List.

ii) **Trees:**

A tree is a collection of Vertices and Edges.

Only one edge between two vertices.



- Binary Tree
- Binary Search Tree
- AVL Tree

Solving Big O notations:

1- Iterative Method:

- Check 3 iterations manually.

- Substitute the found iterations in the first iteration.

- Find the "i^{th}" iteration.

- Using the former step, find the stopping condition for the recursion.

- Substitute the found condition and reach the answer.

$$\mathrm{T}\left(\frac{n}{2^i}\right) \rightarrow\ n = 2^i \rightarrow\ \log_2 n = i \rightarrow\ \theta(logn)$$

1.5- Variables Substitution:

$$t(n) = 2t\left(\sqrt{n}\right) + logn \rightarrow\ m = logn \rightarrow\ n = 2^m \rightarrow\ t(2^m) = 2t\left(2^{m\,0.5}\right) + m = 2t\left(2^{\frac{m}{2}}\right) + m \rightarrow$$

$$\rightarrow\ s(m) = t(2^m) \rightarrow\ s(m) = 2s\left(\frac{m}{2}\right) + m \rightarrow\ \theta(mlogm) \rightarrow\ \theta(lognloglogn)$$

2- Master Theorem:

- a >= 1 & b > 1

- T(n) is defined by positive integers, as follows: $T(n) = a * T\left(\frac{n}{b}\right) + f(n)$

I) if, $f(n) = O\left(n^{\log_b(a-\varepsilon)}\right), \varepsilon > 0$

then, $T(n) = \theta(n^{\log_b(a)})$

II) if, $f(n) = \theta\left(n^{\log_b(a)}\right)$

then, $T(n) = \theta\left(n^{\log_b(a)} * \log(n)\right)$

III) if, $f(n) = \Omega\left(n^{\log_b(a+\varepsilon)}\right), \varepsilon > 0$ AND $a * f\left(\frac{n}{b}\right) \leq c * f(n)$ (regularity condition)

then, $T(n) = \theta(f(n))$

Array Data Structure:

For one dimensional, a simple type varName[] (JAVA) \ var_name = […] (Python).

Stack Data Structure (LIFO):

Operations of stack:

- push()
- pop()
- isEmpty()
- isFull()
- peek(), returns the value of the top element (without actual popping)

Stack.java


Stack.py

Queue Data Structure (FIFO):

Simple Queue - This DS has its limitations.

Until the queue is reset (emptied), the dequeued indexes can't be re-used.

Operations of simple queue:

- enqueue()
- dequeue()
- isEmpty()
- isFull()
- peek(), returns the value of the front element (without removing it)

Queue.java


Queue.py

Circular Queue - It's advantage over the simple queue is better memory utilization.

If the last position is full, and first is empty -> insert element in first.


CircularQueue.java


CircularQueue.py

Priority Queue -

Different methods of implementing (I implemented using LL and Binary Heap, since complexities for Heap and BST are similar):

Operations	peek	insert	delete
Linked List	`O(1)`	`O(n)`	`O(1)`
Binary Heap	`O(1)`	`O(log n)`	`O(log n)`
Binary Search Tree	`O(1)`	`O(log n)`	`O(log n)`


PriorityQueueLinkedList.java


PriorityQueueLinkedList.py

*Simple use of classes:


PriorityQueueHeap.java


PriorityQueueHeap.py

Deque (Double Ended Queue) –

Allows the user to insert\remove elements from both the front and the rear.


DequeCircularQueue.java

*Like CircularQueue.py but also uses insert(index, element) method

Linked List Data Structure:

Operations of Linked Lists:

- insert() – head, tail, and after a specific node
- delete() – delete by position
- search()
- sort() – Python use sort() (uses Timsort algorithm)– Best O(n), Worst O($n \log n$)

Three types of linked lists:

Singly Linked List

SinglyLinkedList.java SinglyLinkedList.py

Doubly Linked List

DoublyLinkedList.java DoublyLinkedList.py

Circular Linked List

Singly LL will also have a pointer from tail to head (doubly LL with an additional pointer from head to tail):

Hash Table:

A data structure which stores elements in pairs – key & value.

Keys are unique, thus allowing us to find their value (by index) fast.

We'll be able to index our values by using a `hash function` (the process is called `hashing`), so `h(k)` (where h() is the hash function and k is the key) will return an index which our associated value will be stored in.

Hash collisions (hash function returns the same index for multiple keys) is usually resolved by two techniques:

- Collision resolution by Chaining:
  - Each of the indexes (slots of the hash table) produced by the hash function, points to the head of a Doubly Linked List. This technique allows us to store multiple values having the same index in a Doubly-LL.
- Open Addressing: Linear Probing, Quadratic Probing and Double Hashing:
  - Each index (slot) is either filled with a single value (which is produced by one key) or left empty (null).
    - i) Linear Probing:
      $h(k,i) = (h'(k) + i)\ mod\ m$
      where $i = \{0, 1, 2, \dots\}$
      and h'(k) is a new hash function
      Meaning if a collision occurs at h(k, 0), check if h(k, 1) is available, and so on.
      BUT what if all elements have keys which produce the same index? Inserting time complexity will be inflected badly.
    - ii) Quadratic Probing:
      Like Linear Probing, but the spacing between the slots is increased. $h(k,i) = (h'(k) + i^2)\ mod\ m$
    - iii) Double Hashing:
      If a collision occurs after applying $h_1(k)$ then use another hash function, as follows:
      $h(k,i) = (h_1(k) + i * h_2(k))\ mod\ m$
      where $i = \{0, 1, 2, \dots\}$
  - An example of a good hash function is the Division Method:
    $h(k) = k\ mod\ m$
- It is always preferred to use 'm' which is a prime number (derived from the hash table size), this will reduce the number of collisions!

HashTableUse.java


SimpleHashTable.py

Heap:

Operations of Heap:

- heapify()
- insert()
- delete()
- peek(), find max/min (peek at root)
- extract(), removes max/min and returns its value (extract root)


MaxHeap.java


MaxHeap.py

- For MinHeap implementation, simply switch the following in heapify() method in the "if statements":
  $arr[largest] < arr[left]$
  $arr[largest] < arr[right]$
  to:
  $arr[largest] > arr[left]$
  $arr[largest] > arr[right]$

Binary Tree:

Tree Traversal:

In-order:

(i) Visit all nodes in the left subtree
(ii) Visit root node
(iii) Visit all nodes in the right subtree

Pre-order:

(i) Visit root node
(ii) Visit all nodes in the left subtree
(iii) Visit all nodes in the right subtree

Post-order:

(i) Visit all nodes in the left subtree
(ii) Visit all nodes in the right subtree
(iii) Visit root node


Node.java


TreeTraversal.java


TreeTraversal.py

Full Binary Tree:

Every parent node has either two or no children nodes.

Perfect Binary Tree:

Every parent node has exactly two child nodes and all leaf nodes are at the same level.

Complete Binary Tree:

Every level is filled with nodes, except the last one which is filled starting from the left most node, but not necessarily all the way to the right most.


BinaryTree.java


BinaryTree.py

Balanced Binary Tree:

The height of the left and right subtree of any node differs by not more than 1!


BalancedBT.java


BalancedBT.py

Binary Search Tree (BST):

All child nodes in the left subtree are smaller than their parent node.

All child nodes in the right subtree are bigger than their parent node.


BinarySearchTree.java


BinarySearchTree.py

AVL Tree:

This is a BST with an extra condition, each node contains an extra field called Balance Factor (like balanced BT concept) whose value is one of the following: -1, 0, 1.

To maintain this, it has to self-balance upon inserting\removing elements. This is done by rotating the subtrees – Left Rotate, Right Rotate, Left Right Rotate, and Right Left Rotate.

- Not planning to implement this DS now (maybe in the future).

Graph Data Structure:

Adjacency:

A vertex is adjacent to another vertex if there is an edge connecting them.

Directed Graph:

This is a graph in which having an edge from `u` to `v` vertices doesn't necessarily mean that there is an edge from `v` to `u`.

Undirected Graph:

Unlike directed graphs, having an edge from `u` to `v` here, does mean there's an edge from `v` to `u` as well.

Connected Graph:

This is a graph in which there is always a path from a vertex to any other vertex.

Graphs are usually represented in one of the following techniques:

i) Adjacency Matrix:
This technique requires more space, but it's simple. Create a 2-D array, and every [row][column] marked with `true`, means there is an edge between those vertices (vertex one being the row index, and the second is the column index). If there is no edge, then the value will be `false`.

ii) Adjacency List:
This technique saves more space because it builds the graph as an array of linked lists. The index of each array represents the vertex and the elements in it's linked list represents the vertices which are connected to it by an edge.

Spanning Tree:

This is a sub-graph of an undirected connected graph, while having the minimum possible number of edges.

The total number of spanning trees with `n` vertices is equal to $n^{(n-2)}$.

Minimum Spanning Tree:

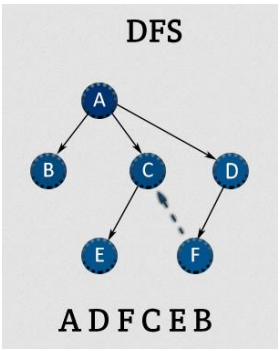
This is a spanning tree in which the sum of the weight of the edges is the possible minimum.

Strongly Connected Components:

This is the portion of a directed graph in which there is a *path from each vertex to another vertex*.

Depth First Search (DFS):

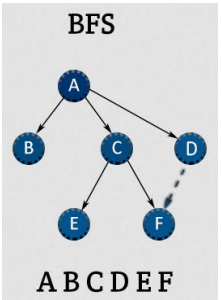
The algorithm marks each vertex as visited while avoiding cycles. DFS traverses as deep as it can reach from the selected start vertex, then continues to all other nodes it can reach. A run example:



Graph.java DFS.java DFS.py

Breadth First Search (BFS):

Like DFS's concept, but first traverses to the neighbors of the starting node, and "expands" in those layers one by one. An example:



BFS.java BFS.py

Binary Search:

This searching algorithm works only on a sorted array.

It allocates two pointers – one on arr[0] and another on arr[arr.length - 1], finds the middle element (between those two pointers) and keeps comparing whether the element to find is bigger\smaller than our current middle while changing the position of the pointers accordingly. Binary Search returns the element index.


BinarySearch.java


BinarySearch.py

Merge Sort Algorithm:

Based on the principle of Divide and Conquer Algorithm. A problem is divided into multiple sub problems, where each sub problem is solved individually. Finally, all sub problems are combined into the final solution.

This sorting algorithm is ***stable***, meaning if there are duplicate elements in the array, for instance two 3s', then the first 3 is still the first in the sorted array as well.

It divides the array into two halves until the sub arrays are of size 1, after that the merge method combines the sorted arrays (an array of size 1 is of course, a sorted array) until the whole array is merged once again.


MergeSort.java


MergeSort.py

QuickSort Algorithm:

Also based on the principle of D&C Algorithm.

The array is divided into sub arrays by selecting a pivot (from the array). Values larger than the pivot element are moved to its right side, and values smaller are moved to the pivot's left side. This step is repeated until each sub array contains only one element.

Finally, the elements are combines into one sorted array.

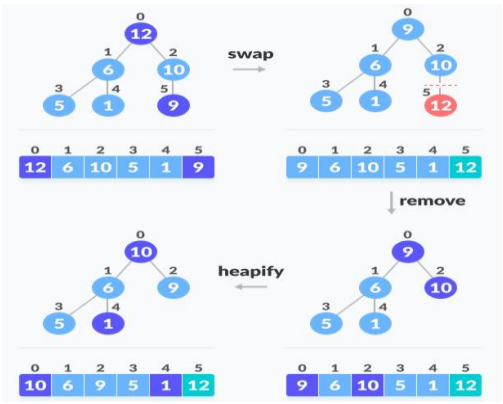
QS is ***not a stable*** algorithm.


QuickSort.java


Quicksort.py

HeapSort Algorithm:

This sorting algorithm (not stable) depends on the principles of the heap data structure. Basically, it will swap the first value (root) of the list with the last value (right most leaf), remove it, and lastly heapify then repeat. An example:



HeapSort.java HeapSort.py

Dijkstra's Algorithm:

Finds the *shortest path* from a source vertex to all other nodes in a graph (if such a path exists). ***No negative*** cycles must exist!

Dijkstra.java Dijkstra.py

Dynamic Programming:

If any problem can be divided into sub problems, and there are overlapping among those sub problems, then the solutions for these sub problems can be saved for future reference. This method improves the overall runtime and is used for optimization.

Dynamic Programming works by storing the result of sub problems so that when their solutions are used, they are already saved and there won't be a need to recalculate them.

This technique can be done using a top-down approach, by starting from the base case and working towards the solution, or a bottom-up manner which is the opposite.

It is mostly applied to recursive algorithms, but of course not to all. A famous example of the latter option is the Fibonacci sequence, which involves the presence of overlapping sub problems, thus a recursion can only reach the solution using a D&C approach.

Backtracking Algorithm:

Backtracking algorithm is a problem-solving algorithm that uses a brute force approach for finding the desired output. If the current solution is not suitable, then backtrack and try other solutions. Meaning a recursive approach is used in this algorithm.

Backtracking is used to solve problems which have multiple solutions.

One example (among many others) for this algorithm, is the "Eight queens puzzle".