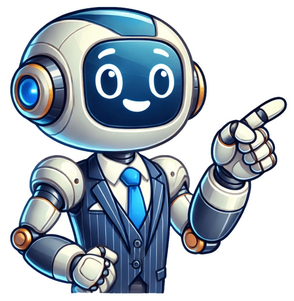
Continue



This is a list of well-known data structures. For a wider list of terms, see list of terms relating to algorithms and data structures. For a comparison of running times for a subset of this list see comparison of data structures. Main article: Primitive type Boolean, true or false. Character Floating-point representation of a finite subset of the rationals. Including single-precision and double-precision IEEE 754 floats, among others Fixed-point representation of the rationals Integer, a direct representation of either the integers or the non-negative integers Reference, sometimes erroneously referred to as a pointer or handle, is a value that refers to another value, possibly including itself Symbol, a unique identifier Enumerated type, a set of symbols Complex, representation of complex numbers Main article: Composite type Array, a sequence of elements of the same type stored contiguously in memory Record (also called a structure or struct), a collection of fields Product type (also called a tuple), a record in which the fields are not named String, a sequence of characters representing text Union, a datum which may be one of a set of types Tagged union (also called a variant, discriminated union or sum type), a union with a tag specifying which type the data is Main article: Abstract data type Container List Tuple Associative array, Map Multimap Set Multiset (bag) Stack Queue (example Priority queue) Double-ended queue Graph (example Tree, Heap) Some properties of abstract data types: Structure Ordered? Uniqueness? List yes no Associative array no keys (indexes) only Set no yes Stack yes no Multimap no no Multiset (bag) no no Queue yes no "Ordered" means that the elements of the data type have some kind of explicit order to them, where an element can be considered "before" or "after" another element. This order is usually determined by the order in which the elements are added to the structure, but the elements can be rearranged in some contexts, such as sorting a list. For a structure that isn't ordered, on the other hand, no assumptions can be made about the ordering of the elements (although a physical implementation of these data types will often apply some kind of arbitrary ordering). "Uniqueness" means that duplicate elements are not allowed. Depending on the implementation of the data type, attempting to add a duplicate element may either be ignored, overwrite the existing element, or raise an error. The detection for duplicates is based on some inbuilt (or alternatively, user-defined) rule for comparing elements. A data structure is said to be linear if its elements form a sequence. Array Associative array Bit array Bit field Bitboard Bitmap Circular buffer Control table Image Dope vector Dynamic array Gap buffer Hashed array tree Lookup table Matrix Parallel array Sorted array Sparse matrix Iliffe vector Variable-length array Doubly linked list Array list Linked list also known as a Singly linked list Association list Self-organizing list Skip list Unrolled linked list VList Conc-tree list Xor linked list Zipper Doubly connected edge list also known as half-edge Difference list Free list Main article: Tree (data structure) Trees are a subset of directed acyclic graphs. AA tree AVL tree Binary search tree Binary tree Cartesian tree Conc-tree list Left-child right-sibling binary tree Order statistic tree Pagoda Randomized binary search tree Red–black tree Rope Scapegoat tree Self-balancing binary search tree Splay tree T-tree Tango tree Threaded binary tree Top tree Treap WAVL tree Weight-balanced tree Zip tree B-tree B+ tree B*-tree Dancing tree 2–3 tree 2–3–4 tree Queap Fusion tree Bx-tree Heap Min-max heap Binary heap B-heap Weak heap Binomial heap Fibonacci heap AF-heap Leonardo heap 2–3 heap Soft heap Pairing heap Leftist heap Treap Beap Skew heap Ternary heap D-ary heap Brodal queue In these data structures each tree node compares a bit slice of key values. Radix tree Suffix tree Suffix array Compressed suffix array FM-index Generalised suffix tree B-tree Judy array Trie X-fast trie Y-fast trie Merkle tree Ternary search tree Ternary tree K-ary tree And–or tree (a,b)-tree Link/cut tree SPQR-tree Spaghetti stack Disjoint-set data structure (Union-find data structure) Fusion tree Enfilade Exponential tree Fenwick tree Van Emde Boas tree Rose tree These are data structures used for space partitioning or binary space partitioning. Segment tree Interval tree Range tree Bin K-d tree Implicit k-d tree Min/max k-d tree Relaxed k-d tree Adaptive k-d tree Quadtree Octree Linear octree Z-order UB-tree R-tree R+ tree R* tree Hilbert R-tree X-tree Metric tree Cover tree M-tree VP-tree BK-tree Bounding interval hierarchy Bounding volume hierarchy BSP tree Rapidly exploring random tree Abstract syntax tree Parse tree Decision tree Alternating decision tree Minimax tree Expectiminimax tree Finger tree Expression tree Log-structured merge-tree PQ tree Approximate Membership Query Filter Bloom filter Cuckoo filter Quotient filter Count–min sketch Distributed hash table Double hashing Dynamic perfect hash table Hash array mapped trie Hash list Hash table Hash tree Hash trie Koorde Prefix hash tree Rolling hash MinHash Ctrie Many graph-based data structures are used in computer science and related fields: Graph Adjacency list Adjacency matrix Graph-structured stack Scene graph Decision tree Binary decision diagram Zero-suppressed decision diagram And-inverter graph Directed graph Directed acyclic graph Propositional directed acyclic graph Multigraph Hypergraph Lightmap Winged edge Quad-edge Routing table Symbol table Piece table E-graph List of algorithms Purely functional data structure Blockchain, a hash-based chained data structure that can persist state history over time Tommy Benchmarks Comparison of several data structures. Retrieved from " roadmap.sh by Community created roadmaps, best practices, projects, articles, resources and journeys to help you choose your path and grow in your career. The top DevOps resource for Kubernetes, cloud-native computing, and large-scale development and deployment. Data Structures & Algorithms What Is Data Structure? Definition, Types, Applications & More What Is Data Structure?Key Features Of Data StructuresBasic Terminologies Related To Data StructuresTypes Of Data StructuresWhat Are Linear Data Structures & Its Types?What Are Non-Linear Data Structures & Its Types?Importance Of Data Structure In ProgrammingBasic Operations On Data StructuresApplications Of Data StructuresReal-Life Applications Of Data StructuresLinear Vs. Non-linear Data StructuresWhat Are Algorithms? The Difference Between Data Structures & AlgorithmsConclusionFrequently Asked Questions Learn About Asymptotic Notations (+ Graphs & Real-Life Examples) What Is Asymptotic Notation?How Asymptotic Notation Helps In Analyzing PerformanceTypes Of Asymptotic NotationBig-O Notation (O)Omega Notation (Ω)Theta Notation (Θ) Little-O Notation (o)Little-Omega Notation (ω)Summary Of Asymptotic NotationsReal-World Applications Of Asymptotic NotationConclusionFrequently Asked Questions Big O Notation | Complexity, Applications & More (+Examples) Understanding Big O NotationTypes Of Time ComplexitySpace Complexity In Big O NotationHow To Determine Big O ComplexityBest, Worst, And Average Case ComplexityApplications Of Big O NotationConclusionFrequently Asked Questions Time Complexity Of Algorithms: Types, Notations, Cases, and More What Is Time Complexity? Why Do You Need To Calculate Time Complexity? The Time Complexity Algorithm Cases Time Complexity: Different Types Of Asymptotic Notations How To Calculate Time Complexity? Time Complexity Of Sorting Algorithms Time Complexity Of Searching Algorithms Writing And optimizing An algorithm What Is Space Complexity And Its Significance? Relation Between Time And Space Complexity Conclusion What Is Linear Data Structure? Types, Uses & More (+ Examples) What Is Linear Data Structure?Key Characteristics Of Linear Data StructuresWhat Are The Types Of Linear Data Structures?What Is An Array Linear Data Structure?What Are Linked Lists Linear Data Structure?What Is A Stack Linear Data Structure?What Is A Queue Linear Data Structure?Most Common Operations Performed in Linear Data StructuresAdvantages Of Linear Data StructuresWhat Is Nonlinear Data Structure?Difference Between Linear & Non-Linear Data StructuresWho Uses Linear Data Structures?ConclusionFrequently Asked Questions What is The Difference Between Linear And Non Linear Data Structure? What is a linear data structure? What is a non-linear data structure? Difference between linear data structure and non-linear data structure FAQs about linear and non-linear data structures What Is Linear Search? Algorithm, Working, Complexity & Examples What Is Search?What Is Linear Search In Data Structure?What Is Linear Search Algorithm? Working Of Linear Search AlgorithmComplexity Of Linear Search Algorithm In Data StructuresImplementations Of Linear Search Algorithm In Different Programming LanguagesReal-World Applications Of Linear Search In Data StructureAdvantages & Disadvantages Of Linear SearchBest Practices For Using Linear Search AlgorithmConclusionFrequently Asked Questions Binary Search Algorithm | Iterative & Recursive With Code Examples What Is The Binary Search Algorithm?Conditions For Using Binary SearchSteps For Implementing Binary SearchIterative Method For Binary Search With Implementation ExamplesRecursive Method For Binary SearchComplexity Analysis Of Binary Search AlgorithmIterative Vs. Recursive Implementation Of Binary SearchAdvantages & Disadvantages Of Binary SearchPractical Applications & Real-World Examples Of Binary SearchConclusionFrequently Asked Questions Jump Search Algorithm | Working, Applications & More (+Examples) Understanding The Jump Search AlgorithmHow Jump Search Works?Code Implementation Of Jump Search AlgorithmTime And Space Complexity AnalysisAdvantages Of Jump SearchDisadvantages Of Jump SearchApplications Of Jump SearchConclusionFrequently Asked Questions Sorting In Data Structures - All Techniques Explained (+Examples) What Is Sorting In Data Structures? What Is Bubble Sort?What Is Selection Sort?What Is Insertion Sort?What Is Merge Sort?What Is Quick Sort?What Is Heap Sort?What Is Radix Sort?What Is Bucket Sort?What Is Counting Sort?What Is Shell Sort?Choosing The Right Sorting AlgorithmApplications Of SortingConclusionFrequently Asked Questions Learn All About Bubble Sort Algorithm (With Code Examples) Understanding Bubble Sort AlgorithmBubble Sort AlgorithmImplementation Of Bubble Sort In C++Time And Space Complexity Analysis Of Bubble Sort AlgorithmAdvantages Of Bubble Sort AlgorithmDisadvantages Of Bubble Sort AlgorithmApplications of Bubble Sort AlgorithmsConclusionFrequently Asked Questions Merge Sort Algorithm | Working, Applications & More (+Examples) Understanding The Merge Sort AlgorithmAlgorithm For Merge SortImplementation Of Merge Sort In C++Time And Space Complexity Analysis Of Merge SortAdvantages And Disadvantages Of Merge SortApplications Of Merge Sort ConclusionFrequently Asked Questions What Is Selection Sort Algorithm? Explained With Code Examples Understanding The Selection Sort AlgorithmAlgorithmic Approach To Selection SortImplementation Of Selection Sort AlgorithmComplexity Analysis Of Selection SortComparison Of Selection Sort With Other Sorting AlgorithmsAdvantages And Disadvantages Of Selection SortApplication Of Selection SortConclusionFrequently Asked Questions Insertion Sort Algorithm - Working Explained (+Code Examples) What Is Insertion Sort Algorithm?How Does Insertion Sort Work? (Step-by-Step Explanation)Implementation of Insertion Sort in C++Time And Space Complexity Of Insertion SortApplications Of Insertion Sort AlgorithmComparison With Other Sorting AlgorithmsConclusion Frequently Asked Questions Quick Sort Algorithm | Working, Applications & More (+Examples) Understanding Quick Sort AlgorithmStep-By-Step Working Of Quick SortImplementation Of Quick Sort AlgorithmTime Complexity Analysis Of Quick SortAdvantages Of Quick Sort AlgorithmDisadvantages Of Quick Sort AlgorithmApplications Of Quick Sort AlgorithmConclusionFrequently Asked Questions Heap Sort Algorithm - Working And Applications (+ Code Examples) Understanding The Heap Data StructureWorking Of Heap Sort Algorithm Implementation Of Heap Sort AlgorithmAdvantages Of Heap SortDisadvantages Of Heap SortReal-World Applications Of Heap SortConclusionFrequently Asked Questions Counting Sort Algorithm In Data Structures (Working & Example) Understanding The Counting Sort AlgorithmConditions For Using Counting Sort AlgorithmHow Counting Sort Algorithm Works?Implementation Of Counting Sort AlgorithmTime And Space Complexity Analysis Of Counting SortComparison Of Counting Sort With Other Sorting AlgorithmsAdvantages Of Counting Sort AlgorithmDisadvantages Of Counting Sort AlgorithmApplications Of Counting Sort Algorithm ConclusionFrequently Asked Questions Shell Sort Algorithm In Data Structures (With Code Examples) Understanding Shell Sort AlgorithmWorking Of Shell Sort AlgorithmImplementation Of Shell Sort AlgorithmTime Complexity Analysis Of Shell Sort AlgorithmAdvantages Of Shell Sort AlgorithmDisadvantages Of Shell Sort AlgorithmApplications Of Shell Sort AlgorithmConclusionFrequently Asked Questions Linked List In Data Structures | Types, Operations & More (+Code) What Is Linked List In Data Structures?Types Of Linked Lists In Data StructuresLinked List Operations In Data StructuresAdvantages And Disadvantages Of Linked Lists In Data StructuresComparison Of Linked Lists And ArraysApplications Of Linked Lists ConclusionFrequently Asked Questions Single Linked List In Data Structure | All Operations (+Examples) What Is A Singly Linked List In Data Structure?Insertion Operations On Singly Linked ListsDeletion Operation On Singly Linked ListSearching For Elements In Single Linked ListCalculating Length Of Single Linked ListPractical Applications Of Singly Linked Lists In Data StructureCommon Problems With Singly Linked ListsConclusionFrequently Asked Questions (FAQ) Reverse A Linked List | All Approaches Explained +Code Examples What Is A Linked List?Reverse A Linked ListHow To Reverse A Linked List? (Approaches)Recursive Approach To Reverse A Linked ListIterative Approach To Reverse A Linked ListUsing Stack To Reverse A Linked ListComplexity Analysis Of Different Approaches To Reverse A Linked ListConclusionFrequently Asked Questions Stack In Data Structures | Operations, Uses & More (+Examples) What Is A Stack In Data Structure?Understanding Stack OperationsStack Implementation In Data StructuresStack Implementation Using ArraysStack Implementation Using Linked ListsComparison: Array vs. Linked List ImplementationApplications Of Stack In Data StructuresAdvantages And Disadvantages Of Stack Data StructureConclusionFrequently Asked Questions Graph Data Structure | Types, Algorithms & More (+Code Examples) What Is A Graph Data Structure?Importance Of Graph Data StructuresTypes Of Graphs In Data StructureTypes Of Graph AlgorithmsApplication Of Graphs In Data StructuresChallenges And Complexities In GraphsConclusionFrequently Asked Questions What Is Tree Data Structure? Operations, Types & More (+Examples) What Is Tree Data Structure?Terminologies Of Tree Data Structure: Different Types Of Tree Data StructuresBasic Operations On Tree Data StructureApplications Of Tree Data StructuresComparison Of Trees, Graphs, And Linear Data StructuresAdvantages Of Tree Data StructureDisadvantages Of Tree Data StructureConclusionFrequently Asked Questions Dynamic Programming - From Basics To Advanced (+Code Examples) What Is Dynamic Programming?Real-Life Example: The Jigsaw Puzzle AnalogyHow To Solve A Problem Using Dynamic Programming?Dynamic Programming Algorithm TechniquesAdvantages Of Dynamic ProgrammingDisadvantages Of Dynamic ProgrammingApplications Of Dynamic Programming ConclusionFrequently Asked Questions Sliding Window Algorithm - Working Explained With Code Examples Understanding The Sliding Window AlgorithmHow Does The Sliding Window Algorithm Works?How To Identify Sliding Window Problems?Fixed-Size Sliding Window Example: Maximum Sum Subarray Of Size kVariable-Size Sliding Window Example: Smallest Subarray With A Given SumAdvantages Of Sliding Window TechniqueDisadvantages Of Sliding Window TechniqueConclusionFrequently Asked Questions 55+ Data Structure Interview Questions For 2025 (Detailed Answers) Introduction To Data StructuresData Structures Interview Questions: BasicsData Structures Interview Questions: IntermediateData Structures Interview Questions: AdvancedConclusion Data structures are the fundamental building blocks of computer programming. They define how data is organized, stored, and manipulated within a program. Understanding data structures is very important for developing efficient and effective algorithms. What is Data Structure?A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So we must have good knowledge about data structures. Classification of Data StructureClassification of Data StructureLinear Data Structure: Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure. Example: Array, Stack, Queue, Linked List, etc.Static Data Structure: Static data structure has a fixed memory size. It is easier to access the elements in a static data structure. Example: array.Dynamic Data Structure: In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code. Example: Queue, Stack, etc.Non-Linear Data Structure: Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only. Examples: Trees and Graphs.Please refer Complete Data Structures & Algorithms Tutorial for topic-wise guide, practice problems and interview questions. What are data structures?What are algorithms?Why learn data structures and algorithms?When are you ready to learn about data structures and algorithms? Understanding data structures1. ArraysOperations and time complexity2. Linked listsTypes of linked listsSingly linked listDoubly linked listCircular linked listOperations and time complexity3. StacksOperations and time complexity4. QueuesOperations and time complexity5. TreesTypes of treesBinary treeBinary search tree (BST)AVL tree (Balanced BST)Heap (Min-heap, max-heap)Operations and time complexityTrieOperations and time complexity6. GraphsTypes of graphsDirected graph (Digraph)Undirected graphWeighted graphAcyclic graphGraph representationExploring algorithms1. Searching algorithmsLinear searchBinary search2. Sorting algorithmsBubble sortMerge sort3. Graph algorithmsDepth-first search (DFS)Breadth-first search (BFS)4. Dynamic programmingA 6-step roadmap to building DSA proficiencyBuilding a strong foundation Data structures are the fundamental building blocks of computer programming. They define how data is organized, stored, and manipulated within a program. Understanding data structures is very important for developing efficient and effective algorithms. What is Data Structure?A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So we must have good knowledge about data structures. Classification of Data StructureClassification of Data StructureLinear Data Structure: Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure. Example: Array, Stack, Queue, Linked List, etc.Static Data Structure: Static data structure has a fixed memory size. It is easier to access the elements in a static data structure. Example: array.Dynamic Data Structure: In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code. Example: Queue, Stack, etc.Non-Linear Data Structure: Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only. Examples: Trees and Graphs.Please refer Complete Data Structures & Algorithms Tutorial for topic-wise guide, practice problems and interview questions. Data structures are the fundamental building blocks of computer programming. They define how data is organized, stored, and manipulated within a program. Understanding data structures is very important for developing efficient and effective algorithms. What is Data Structure?A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So we must have good knowledge about data structures. Classification of Data StructureClassification of Data StructureLinear Data Structure: Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure. Example: Array, Stack, Queue, Linked List, etc.Static Data Structure: Static data structure has a fixed memory size. It is easier to access the elements in a static data structure. Example: array.Dynamic Data Structure: In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code. Example: Queue, Stack, etc.Non-Linear Data Structure: Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only. Examples: Trees and Graphs.Please refer Complete Data Structures & Algorithms Tutorial for topic-wise guide, practice problems and interview questions. Data structures are the fundamental building blocks of computer programming. They define how data is organized, stored, and manipulated within a program. Understanding data structures is very important for developing efficient and effective algorithms. What is Data Structure?A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So we must have good knowledge about data structures. Classification of Data StructureClassification of Data StructureLinear Data Structure: Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure. Example: Array, Stack, Queue, Linked List, etc.Static Data Structure: Static data structure has a fixed memory size. It is easier to access the elements in a static data structure. Example: array.Dynamic Data Structure: In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code. Example: Queue, Stack, etc.Non-Linear Data Structure: Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only. Examples: Trees and Graphs.Please refer Complete Data Structures & Algorithms Tutorial for topic-wise guide, practice problems and interview questions. Share — copy and redistribute the material in any medium or format for any purpose, even commercially. Adapt — remix, transform, and build upon the material for any purpose, even commercially. The licensor cannot revoke these freedoms as long as you follow the license terms. Attribution — You must give appropriate credit , provide a link to the license, and indicate if changes were made . You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use. ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits. You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation . No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material. Cookies are disabled in your browser settings. To fully access our website and its features, please enable cookies and Refresh the page. By javinpaul Data structures and algorithms are some of the most essential topics for programmers, both to get a job and to do well on a job. Good knowledge of data structures and algorithms is the foundation of writing good code. If you are familiar with essential data structures e.g. array, string, linked list, tree, map, and advanced data structures like tries, and self-balanced trees like AVL trees, etc., you'll know when to use which data structure and compute the CPU and memory cost of your code. Even though you don't need to write your own array, linked list, or hashtable, given every major programming SDK provides them, e.g. JDK or C++ STL library, you will need to understand them so that you can use them in the right place. Using the right data structure can drastically improve the performance of an algorithm. Ideally, we should all learn data structures and algorithms in our schools and colleges, but it's rarely ever covered. Most of the programmers, including myself, only get introduced to a data structure in our computer science courses, but we didn't really learn the real-world importance of them, and that's why we didn't understand them better. For us, they are just the algorithms and data structures e.g. some concept, not a tool that you can use to write good programs. We didn't know that Facebook would use them to store our details or that Google would use them to store web pages and link to search queries. Anyway, it's never too late. If you think that your data structure knowledge is not up to par or you want to improve your data structure knowledge, then you have come to the right place. Earlier, I shared some free books to learn data structures and algorithms, and in this article, you will learn about some of the best data structure and algorithm courses that are available for free online. Here is my list of some of the best courses to learn data structures and algorithms, which are also free. Many programmers think that free resources are not good, which is not true. Even though they sometimes don't match the quality and coverage of paid resources, they are, in fact, the best resources to start with. You can use these courses to familiarize yourself with the essential data structures and learn some basics. Some of them are particularly good from an interview point of view so you can use them alongside any book you are using to prepare for your programming job interview. This is another free, online algorithm and data structure training course, which aims to teach basic data structures in computer programming. The data structures taught in the course include Stack, Queue, and Linked List using the C programming language. The primary goal of this course is to make students and software engineers visualize how different data structures work. Data Structures Concepts in C This is not an exhaustive course, but you will learn about Stack, Queue, and Linked List. In short, it is a great course for programmers new to data structures and those who have just learned a programming language and want to learn the basics of how data structures work. This is the first part of a two-part series of courses covering algorithms and data structures on Courera by Robert Sedgewick. In this part, you will learn essential data structures like linked lists, stacks, queues, binary trees, and hash tables, and searching and sorting algorithms like binary search, quicksort, mergesort, insertion sort etc. Algorithms Part 1 You will also learn about core data structures and algorithms used in everyday applications and understand the trade-offs involved with choosing each data structure along with traversal, retrieval, and update algorithms. All the features of this course are available for free but it does not offer a certificate upon completion. This is the second part of a two-part series of free online Coursera courses covering data structures and algorithms by Robert Sedgewick and Kevin Wayne, both are professors of Computer Science. Robert Sedgewick is also the author of Algorithms (4th Edition) book, one of the most popular books on Algorithms for Java developers. In this part, you will learn about the graph- and string-processing algorithms. You will also learn some advanced data structures and algorithms used in application development. Talking about the social proof, both Algorithms Part 1 and Algorithms Part 2 are highly recommended courses and have impressive reviews and ratings. Algorithms — Part 2 This is an introductory online course on data structures, algorithms, and complexity analyses. It will teach you design, implementation, and analyses of basic data structures using the Java language. The best part of this course is the number of practical examples and that it focuses on intuition rather than formulas and mathematical proofs. Data Structure in Java The course provides a good introduction for "complexity analysis," which makes it possible to spot a poorly (and a great) performing program, even without the need for executing it. That's all about some of the free data structure and algorithm courses available online. You can use these courses to learn data structures and algorithms at your own pace. They may not all be comprehensive, but they do provide a good introduction. Once you go through these courses, you can pick a good book like Introduction to Algorithms to further supplement your knowledge. Update: this course is no longer free, but it's a great course and may be worth it for you. This data structure and algorithm courses from Udemy is for all those people who want to learn data structures from an absolute basic to advanced level. This course doesn't expect you to have any prior knowledge of data structures, but a basic prior knowledge of Java is helpful. The author @William Fiset is a Software Engineer at Google and a former ACM-ICPC world finalist and has excellent computer programming and problem-solving skills. Talking about social proof, the course has more than 25K students and an average 4.1 ratings from 683 rating which is impressive. In short, it's a complete guide to learning everything there is to know about data structures. Easy to Advanced Data Structures This is also an ideal course for computer science students and working software professionals who want to learn data structures and algorithms from an interview perspective. So, if you are looking to get a job with product-based companies like Amazon, Google, Microsoft, or Facebook, you can use this course to kick-start your preparation. Thanks. You made it to the end of the article. Good luck with your programming Journey! It's certainly not going to be easy, but by following these courses, you are one step closer to mastering data structure and algorithms than others. If you like this article, then please share with your friends and colleagues, and don't forget to follow javinpaul on Twitter! How can financial brands set themselves apart through visual storytelling? Our experts explain how.Learn MoreThe Motorsport Images Collections captures events from 1895 to today's most recent coverage.Discover The CollectionCurated, compelling, and worth your time. Explore our latest gallery of Editors' Picks.Browse Editors' FavoritesHow can financial brands set themselves apart through visual storytelling? Our experts explain how.Learn MoreThe Motorsport Images Collections captures events from 1895 to today's most recent coverage.Discover The CollectionCurated, compelling, and worth your time. Explore our latest gallery of Editors' Picks.Browse Editors' FavoritesHow can financial brands set themselves apart through visual storytelling? Our experts explain how.Learn MoreThe Motorsport Images Collections captures events from 1895 to today's most recent coverage.Discover The CollectionCurated, compelling, and worth your time. Explore our latest gallery of Editors' Picks.Browse Editors' Favorites MODULE 1Introduction of Data Structure and algorithmMODULE 2Complexity analysis of Data Structure and AlgorithmsMODULE 3Common Problem Solving TechniquesMODULE 4Array Data StructureMODULE 5Searching AlgorithmsMODULE 6Sorting AlgorithmsMODULE 7Array Advanced AlgorithmsMODULE 8String Data StructureMODULE 9String Pattern MatchingMODULE 10String Advanced AlgorithmsMODULE 12Stacks and QueuesMODULE 15Greedy Algorithms and Dynamic ProgrammingMODULE 17Graph Connected ComponentsMODULE 19Shortest Path AlgorithmsMODULE 20Graph flow algorithmsMODULE 21Graph Advanced AlgorithmsMODULE 22Maths and AlgebraYou can claim your course certificate upon course completion. You would be able to use this certificate on your resume, Linkedin profile or your website.Learn MoreHave you ever wondered why most of the product-based companies focus so much on data structures and algorithms in their interviews for positions like Software Development Engineer, Data Scientist, Machine Learning Engineer, and so on? If yes, all the answers to your questions lie here.It's always crucial to understand why to learn data structures and algorithms, their needs and applications, and other questions related to data structures and algorithms. Let's deep dive into it. Data Structure is a way to organize and store data in a computer so that it can be accessed and used efficiently. Data structures provide logical and organized way of representing and manipulating data.As an example consider a phonebook where we can search for a person's phone name by looking up their phone number. Data Structures can be classified in two categories: Primitive data structure: It can store simple values such as integers, characters, and floating-point numbers. Non-primitive data structure: It can store multiple data elements of different types. They are furthur divided in two categories: Linear: In linear data structures, the data elements are arranged in a sequential manner, such as arrays, linked lists, and stacks. Non-Linear: In non-linear data structures data elements are not arranged in a sequential manner, such as trees and graphs. Each data structure has its own advantages and disadvantages, and choosing the right one for a specific application is essential for efficient processing and storage of data. Consider the above example of PhoneBook. It can be implemented using Hash-tables where each entry consists of a name and phone number. Data Structure is something that can be used to store and organize data in a particular fashion. And, now comes the algorithm. An algorithm is a step-by-step set of instructions to solve a particular problem.In simple words, you can say that Data Structures are nothing but "meaningful" arrangements of data that algorithms can use to solve any particular problem!Can you imagine going to a library and finding all 10,000 books stored randomly? No! It will be a very hectic task to find the book you want. So we need to create an better or optimized solution to store and search the books instead of a simple solution of randomly searching. And that's why we need to learn data structures and algorithms and understand their tradeoffs for different situations to be able to create optimized solutions To solve some real-world complex problems: Yes you heard it right. Consider the above example of searching books in the library, there you can't search books randomly, you need a proper approach to search books in order to save time, and here data structures and algorithms came into play to solve some real-life based problems. Optimization and Scalability: Once you have knowledge of data structures and algorithms, you can easily decide which data structure can be used at which place and which algorithm will be best for your use case. This helps in writing more optimized and scalable code. Improving your problem solving skills: DSA is your toolbox for tackling some of the toughest challenges in the tech world. From your WhatsApp chat to LinkedIn Feed - everything uses DSA user the hood in some form. Whether you build your own projects, participate in competitive coding contests or work as a software developer - the knowledge of DSA is always helpful. For job opportunities: Another point is that these days most product-based companies ask DSA and algorithms in their interviews as they want to judge the problem-solving skills of the candidate. So learning DSA and algorithms will give you an advantage during the interviews and hence can land you in your dream company. Reducing time complexity- DSA plays a major role in reducing the time complexity of the code. A problem can be solved using various approaches, but you have to pick the optimized one in order to be more productive and solve the problem in lesser time. It can be done through learning data structures and algorithms. The core of computer science - Data structures and algorithms are considered to be the foundation of computer science. With advancements in technologies, more and more data is getting stored. A huge amount of data can slow down the processing speed of computer systems. This is where data structures can help us. They can improve the processing power of the computer by the effective utilization and storage of data. Learn DSA from Scaler: You can learn DSA through the Scaler Topics. We offer a complete series of in depth DSA tutorials along with suitable real life examples. These are targeted for absolute beginners who want to dive into the field of data structures and algorithms. Learn through books: You can learn DSA through various available books also. A few famous books are: "Introduction to Algorithms" by Thomas H. Cormen, "The Algorithm Design Manual" by Steven S. Skiena, "Algorithms" by Robert Sedgewick, and Kevin Wayne, and many more. There are a lot of real-life applications of Data structures and algorithms you can see around you. Like Facebook, how that connection and friends logic is built. All that logic is built through Graph data structure.Similarly, Google maps uses the Graph data structure internally. So there are tons of applications of data structures and algorithms which you can see all around you.Some of the problems that can be solved using DSA and Algorithms are- Knapsack problem Tower of Hanoi Shortest distance between two points Project scheduling and many more... The target audience of this tutorials are Computer Science graduates as well as Software Professional who wants to learn data structures and algorithm programming in simple and easy manner. Prerequisites for these tutorials are: You should have basic understanding of any one of below languages: Hope you get an idea about Data structures and algorithms and their importance and need. One should learn DSA in order to enhance their problem-solving skills and for better job opportunities in good companies. Python has been used worldwide for different fields such as making websites, artificial intelligence and much more. But to make all of this possible, data plays a very important role which means that this data should be stored efficiently and the access to it must be timely. So how do you achieve this? We use something called Data Structures. With that being said, let us go through the topics we will cover in Data Structures in Python. The article has been broken down into the following parts:So, let's get started :)Organizing, managing and storing data is important as it enables easier access and efficient modifications. Data Structures allows you to organize your data in such a way that enables you to store collections of data, relate them and perform operations on them accordingly. It is one of the most basic concepts that beginners get to know about when learning the best Python course online for exams like PCEP, PCAP, PCPP. Python has implicit support for Data Structures which enable you to store and access data. These structures are called List, Dictionary, Tuple and Set.Python allows its users to create their own Data Structures enabling them to have full control over their functionality. The most prominent Data Structures are Stack, Queue, Tree, Linked List and so on which are also available to you in other programming languages. So now that you know what are the types available to you, why don't we move ahead to the Data Structures and implement them using Python.As the name suggests, these Data Structures are built-in with Python which makes programming easier and helps programmers use them to obtain solutions faster. Let's discuss each of them in detail.ListsLists are used to store data of different data types in a sequential manner. There are addresses assigned to every element of the list, which is called as Index. The index value starts from 0 and goes on until the last element called the positive index. There is also negative indexing which starts from -1 enabling you to access elements from the last to first. Let us now understand lists better with the help of an example program.Creating a listTo create a list, you use the square brackets and add elements into it accordingly. If you do not pass any elements inside the square brackets, you get an empty list as the output. my_list = [] #create empty list print(my_list) my_list = [1, 2, 3, 'example', 3.132] #creating list with data print(my_list) Output: [] [1, 2, 3, 'example', 3.132]Adding ElementsAdding the elements in the list can be achieved using the append(), extend() and insert() functions.The append() function adds all the elements passed to it as a single element. The extend() function adds the elements one-by-one into the list. The insert() function adds the element passed to the index value and increase the size of the list too. my_list = [1, 2, 3] print(my_list) my_list.append([555, 12]) #add as a single element print(my_list) my_list.extend([234, 'more_example']) #add as different elements print(my_list) my_list.insert(1, 'insert_example') #add element i print(my_list) Output: [1, 2, 3] [1, 2, 3, [555, 12]] [1, 2, 3, [555, 12], 234, 'more_example'] [1, 'insert_example', 2, 3, [555, 12], 234, 'more_example']Upskill for Higher Salary with Python Programming CoursesDeleting ElementsTo delete elements, use the del keyword which is built-in into Python but this does not return anything back to us. If you want the element back, you use the pop() function which takes the index value. To remove an element by its value, you use the remove() function.Example: my_list = [1, 2, 3, 'example', 3.132, 10, 30] del my_list[5] #delete element at index 5 print(my_list) my_list.remove('example') #remove element with value print(my_list) a = my_list.pop(1) #pop element from list print('Popped Element: ', a, ' List remaining: ', my_list) my_list.clear() #empty the list print(my_list) Output: [1, 2, 3, 'example', 3.132, 30] [1, 2, 3, 3.132, 30] Popped Element: 2 List remaining: [1, 3, 3.132, 30] []Accessing ElementsAccessing elements is the same as accessing Strings in Python. You pass the index values and hence can obtain the values as needed. my_list = [1, 2, 3, 'example', 3.132, 10, 30] for element in my_list: #access elements one by one print(element) print(my_list) #access all elements print(my_list[3]) #access index 3 element print(my_list[0:2]) #access elements from 0 to 1 and exclude 2 print(my_list[::-1]) #access elements in reverse Output: 1 2 3 example 3.132 10 30 [1, 2, 3, 'example', 3.132, 10, 30] example [1, 2] [30, 10, 3.132, 'example', 3, 2, 1]Other FunctionsYou have several other functions that can be used when working with lists.The len() function returns to us the length of the list. The index() function finds the index value of value passed where it has been encountered the first time. The count() function finds the count of the value passed to it. The sorted() and sort() functions do the same thing, that is to sort the values of the list. The sorted() has a return type whereas the sort() modifies the original list. my_list = [1, 2, 3, 10, 30, 10] print(len(my_list)) #find length of list print(my_list.index(10)) #find index of element that occurs first print(my_list.count(10)) #find count of the element print(sorted(my_list)) #print sorted list but not change original my_list.sort(reverse=True) #sort original list print(my_list) Output:6 3 2 [1, 2, 3, 10, 10, 30] [30, 10, 10, 3, 2, 1]String A string is a sequence of characters in Python used to represent text. Strings are immutable, meaning they cannot be changed after they are created.Creating a String:Creating a string is as easy as enclosing characters in single or double quotes. my_string = "Hello, World!" print(my_string) # Output: Hello, World! String Operations:Concatenation: Joining two or more strings together.str1 = "Hello" str2 = "Python" result = str1 + " " + str2 print(result) Output: Hello PythonRepetition: Repeating a string multiple times.repeated = "Hi! " * 3 print(repeated) Output: Hi! Hi! Hi!Slicing: Extracting a portion of a string using index positions.text = "Programming" print(text[0:5]) Output: ProgrLength: Finding the number of characters in a string.print(len(text)) Output: 11Changing Case: Convert a string to uppercase or lowercase.print(text.upper()) Output: PROGRAMMINGByteArray A bytearray in Python is a mutable sequence of bytes. It allows you to work with binary data and modify the bytes directly. It's similar to a list of integers where each integer represents a byte (0 to 255).Example: Creating, Modifying, and Adding Elements to a bytearray# Creating a bytearray from a string data = bytearray("Hello", 'utf-8') print("Original bytearray:", data.decode('utf-8')) # Output: Hello # Modifying the first byte (H -> J) data[0] = ord('J') print("After modification:", data.decode('utf-8')) # Output: Jello # Adding a single byte ('!') data.append(ord('!')) # Extending the bytearray with another byte sequence (' World') data.extend(bytearray(" World", 'utf-8')) # Final output print("Final bytearray:", data.decode('utf-8')) # Output: Jello! World Output:Original bytearray: HelloAfter modification: JelloFinal bytearray: Jello! WorldThis example covers the creation, modification, and addition of elements to a bytearray.Collections Module:The collections module in Python provides specialized container data types that extend the functionality of built-in types like lists, dictionaries, and tuples.Here are some key data structures from the collections module, along with code examples and its output:CounterCounter is a subclass of dict used to count hashable objects. It counts the frequency of elements in an iterable.from collections import Counter# Example: Count frequencies of characters in a string data = "apple" counter = Counter(data) print(counter) Output:Counter({'p': 2, 'a': 1, 'l': 1, 'e': 1})OrderedDictOrderedDict is a dictionary subclass that remembers the order in which keys were inserted.from collections import OrderedDict# Example: Create an ordered dictionary ordered_dict = OrderedDict() ordered_dict['first'] = 1 ordered_dict['second'] = 2 ordered_dict['third'] = 3 print(ordered_dict) Output:OrderedDict([('first', 1), ('second', 2), ('third', 3)])Defaultdictdefaultdict is a dictionary that returns a default value if the key is not found. You can specify the default type (like int, list, etc.).from collections import defaultdict# Example: defaultdict with int (default value 0) dd = defaultdict(int) dd['a'] += 1 dd['b'] += 2 print(dd) Output:defaultdict(, {'a': 1, 'b': 2})ChainMapChainMap groups multiple dictionaries into a single view, allowing lookup across dictionaries.from collections import ChainMap# Example: Chain two dictionaries dict1 = {'a': 1, 'b': 2} dict2 = {'b': 3, 'c': 4} chain_map = ChainMap(dict1, dict2) print(chain_map['b']) # Value from dict1 takes precedence print(chain_map['c']) # Value from dict2 Output:24NamedTupleNamedTuple is a lightweight, immutable object that provides named fields for accessing data like a class.from collections import named tuple# Example: Create a named tuple for a point in 2D space Point = namedtuple('Point', ['x', 'y']) p = Point(10, 20) print(p.x, p.y) Output:10 20Dictdict is the built-in Python dictionary, a mutable data type used to store key-value pairs.# Example: Basic dictionary usage dict_example = {'name': 'Alice', 'age': 25} print(dict_example['name']) dict_example['age'] = 26 print(dict_example) Output:Alice{'name': 'Alice', 'age': 26}These are some of the most commonly used data structures from the collections module in Python. Each offers specialized functionality to simplify certain tasks in Python programming.UserDictUserDict is a wrapper around the regular dictionary (dict) that makes it easier to subclass and customize dictionary behavior.from collections import UserDict# Example: Custom dictionary that converts keys to uppercase class MyDict(UserDict): def __setitem__(self, key, value): super().__setitem__(key.upper(), value) my_dict = MyDict() my_dict['name'] = 'Alice' print(my_dict) Output:{'NAME': 'Alice'}UserListUserList is a wrapper around the standard list that allows you to modify list behavior when subclassing.from collections import UserList# Example: Custom list that prevents adding negative numbers class MyList(UserList): def append(self, item): if item >= 0: super().append(item) my_list = MyList() my_list.append(10) my_list.append(-5) print(my_list) Output:[10]UserStringUserString is a wrapper around the standard string (str) that makes it easier to subclass and extend string behavior.from collections import UserString# Example: Custom string that converts to lowercase class MyString(UserString): def __init__(self, data): super().__init__(data.lower()) my_string = MyString("HELLO WORLD") print(my_string) Output:hello worldThese classes make it easier to create custom objects that behave like built-in data types but with added functionality.DictionaryDictionaries are used to store key-value pairs. To understand better, think of a phone directory where hundreds and thousands of names and their corresponding numbers have been added. Now the constant values here are Name and the Phone Numbers which are called as the keys. And the various names and phone numbers are the values that have been fed to the keys. If you access the values of the keys, you will obtain all the names and phone numbers. So that is what a key-value pair is. And in Python, this structure is stored using Dictionaries. Let us understand this better with an example program.Creating a DictionaryDictionaries can be created using the flower braces or using the dict() function. You need to add the key-value pairs whenever you work with dictionaries. my_dict = {} #empty dictionary print(my_dict) my_dict = {1: 'Python', 2: 'Java'} #dictionary with elements print(my_dict) Output: {} {1: 'Python', 2: 'Java'}Changing and Adding key, value pairsTo change the values of the dictionary, you need to do that using the keys. So, you firstly access the key and then change the value accordingly. To add values, you simply just add another key-value pair as shown below. my_dict = {'First': 'Python', 'Second': 'Java'} print(my_dict) my_dict['Second'] = 'C++' #changing element print(my_dict) my_dict['Third'] = 'Ruby' #adding key-value pair print(my_dict) Output: {'First': 'Python', 'Second': 'Java'} {'First': 'Python', 'Second': 'C++'} {'First': 'Python', 'Second': 'C++', 'Third': 'Ruby'}Deleting key, value pairsTo delete the values, you use the pop() function which returns the value that has been deleted. To retrieve the key-value pair, you use the popitem() function which returns a tuple of the key and value. To clear the entire dictionary, you use the clear() function. my_dict = {'First': 'Python', 'Second': 'Java', 'Third': 'Ruby'} a = my_dict.pop('Third') #pop element print('Value:', a) print('Dictionary:', my_dict) b = my_dict.popitem() #pop the key-value pair print('Key, value pair:', b) print('Dictionary', my_dict) my_dict.clear() #empty dictionary print('n', my_dict) Output:Value: Ruby Dictionary: {'First': 'Python', 'Second': 'Java'}Key, value pair: ('Second', 'Java') Dictionary {'First': 'Python'}{}Accessing ElementsYou can access elements using the keys only. You can use either the get() function or just pass the key values and you will be retrieving the values. my_dict = {'First': 'Python', 'Second': 'Java'} print(my_dict['First']) #access elements using keys print(my_dict.get('Second')) Output: Python JavaOther FunctionsYou have different functions which return to us the keys or the values of the key-value pair accordingly to the keys(), values(), items() functions accordingly. my_dict = {'First': 'Python', 'Second': 'Java', 'Third': 'Ruby'} print(my_dict.keys()) #get keys print(my_dict.values()) #get values print(my_dict.items()) #get key-value pairs print(my_dict.get('First')) Output: dict_keys(['First', 'Second', 'Third']) dict_values(['Python', 'Java', 'Ruby']) dict_items([('First', 'Python'), ('Second', 'Java'), ('Third', 'Ruby')]) PythonTupleTuples are the same as lists are with the exception that the data once entered into the tuple cannot be changed no matter what. The only exception is when the data inside the tuple is mutable, only then the tuple data can be changed. The example program will help you understand better.Creating a TupleYou create a tuple using parenthesis or using the tuple() function. my_tuple = (1, 2, 3) #create tuple print(my_tuple) Output: (1, 2, 3)Accessing ElementsAccessing elements is the same as it is for accessing values in lists. my_tuple2 = (1, 2, 3, 'edureka') #access elements for x in my_tuple2: print(x) print(my_tuple2) print(my_tuple2[0]) print(my_tuple2[:]) print(my_tuple2[3][4]) Output: 1 2 3 edureka (1, 2, 3, 'edureka') 1 (1, 2, 3, 'edureka') eAppending ElementsTo append the values, you use the '+' operator which will take another tuple to be appended to it. my_tuple = (1, 2, 3) my_tuple = my_tuple + (4, 5, 6) #add elements print(my_tuple) Output: (1, 2, 3, 4, 5, 6)Other FunctionsThese functions are the same as they are for lists. my_tuple = (1, 2, 3, ['hindi', 'python']) my_tuple[3][0] = 'english' print(my_tuple) print(my_tuple.count(2)) print(my_tuple.index(['english', 'python'])) Output: (1, 2, 3, ['english', 'python']) 1 3SetsSets are a collection of unordered elements that are unique. Meaning that even if the data is repeated more than one time, it would be entered into the set only once. It resembles the sets that you have learnt in arithmetic. The operations also are the same as is with the arithmetic sets. An example program would help you understand better.Creating a setSets are created using the flower braces but instead of adding key-value pairs, you just pass values to it. my_set = {1, 2, 3, 4, 5, 5, 5} #create set print(my_set) Output: {1, 2, 3, 4, 5}Adding elementsTo add elements, you use the add() function and pass the value to it. my_set = {1, 2, 3} my_set.add(4) #add element to set print(my_set) Output: {1, 2, 3, 4}Operations in setsThe different operations on set such as union, intersection and so on are shown below. my_set = {1, 2, 3, 4} my_set_2 = {3, 4, 5, 6} print(my_set.union(my_set_2), '----------', my_set | my_set_2) print(my_set.intersection(my_set_2), '----------', my_set & my_set_2) print(my_set.difference(my_set_2), '----------', my_set - my_set_2) print(my_set.symmetric_difference(my_set_2), '----------', my_set ^ my_set_2) my_set.clear() print(my_set) The union() function combines the data present in both sets. The intersection() function finds the data present in both sets only. The difference() function deletes the data present in both and outputs data present only in the set passed. The symmetric_difference() does the same as the difference() function but outputs the data which is remaining in both sets.Output: {1, 2, 3, 4, 5, 6} ——–- {1, 2, 3, 4, 5, 6} {3, 4} ——–- {3, 4} {1, 2} ——–- {1, 2} {1, 2, 5, 6} ——–- {1, 2, 5, 6} set()Frozen SetA frozen set in Python is an immutable version of a regular set, meaning its elements cannot be changed after creation (no adding, removing, or modifying elements). It's useful when you need a set that should not be altered, like using it as a key in a dictionary.Creating a Frozen Set:frozen = frozenset([1, 2, 3, 4])print(frozen)Adding Elements:You cannot add elements to a frozen set after its creation since it's immutable.Operations on Frozen Sets:You can still perform set operations like:Union: frozen1 | frozen2 Intersection: frozen1 & frozen2 Difference: frozen1 – frozen2 Symmetric Difference: frozen1 ^ frozen2Example:frozen1 = frozenset([1, 2, 3])frozen2 = frozenset([3, 4, 5])# Unionprint(frozen1 | frozen2) Output: frozenset({1, 2, 3, 4, 5})# Intersectionprint(frozen1 & frozen2) Output: frozenset({3})Now that you have understood the built-in Data Structures, let's get started with the user-defined Data Structures. User-defined Data Structures, the name itself suggests that users define how the Data Structure would work and define functions in it. This gives the user whole control over how the data needs to be saved, manipulated and so forth.Explore top Python interview questions covering topics like data structures, algorithms, OOP concepts, and problem-solving techniques. Master key Python skills to ace your interview and secure your next developer role.Let us move ahead and study the most prominent Data Structures in most of the programming languages.Arrays vs. ListsArrays and lists are the same structure with one difference. Lists allow heterogeneous data element storage whereas Arrays allow only homogenous elements to be stored within them.StackStacks are linear Data Structures which are based on the principle of Last-In-First-Out (LIFO) where data which is entered last will be the first to get accessed. It is built using the array structure and has operations namely, pushing (adding) elements, popping (deleting) elements and accessing elements only from one point in the stack called as the TOP. This TOP is the pointer to the current position of the stack. Stacks are prominently used in applications such as Recursive Programming, reversing words, undo mechanisms in word editors and so forth.QueueA queue is also a linear data structure which is based on the principle of First-In-First-Out (FIFO) where the data entered first will be accessed first. It is built using the array structure and has operations which can be performed from both ends of the Queue, that is, head-tail or front-back. Operations such as adding and deleting elements are called En-Queue and De-Queue and accessing the elements can be performed. Queues are used as Network Buffers for traffic congestion management, used in Operating Systems for Job Scheduling and many more.TreeTrees are non-linear Data Structures which have root and nodes. The root is the node from where the data originates and the nodes are the other data points that are available to us. The node that precedes is the parent and the node after is called the child. There are levels a tree has to show the depth of information. The last nodes are called the leaves. Trees create a hierarchy which can be used in a lot of real-world applications such as the HTML pages use trees to distinguish which tag comes under which block. It is also efficient in searching purposes and much more.Linked ListLinked lists are linear Data Structures which are not stored consequently but are linked with each other using pointers. The node of a linked list is composed of data and a pointer called next. These structures are most widely used in image viewing applications, music player applications and so forth.GraphGraphs are used to store data collection of points called vertices (nodes) and edges (edges). Graphs can be called as the most accurate representation of a real-world map. They are used to find the various cost-to-distance between the various data points called as the nodes and hence find the least path. Many applications such as Google Maps, Uber, and many more use Graphs to find the least distance and increase profits in the best ways.HashMapsHashMaps are the same as what dictionaries are in Python. They can be used to implement applications such as phonebooks, populate data according to the lists and much more.That wraps up all the prominent Data Structures in Python. I hope you have understood built-in as well as the user-defined Data Structures that we have in Python and why they are important.Now that you have understood the Data Structures in Python, check out the Data Science with Python Course Online available on Edureka, a trusted online learning company with a network of more than 250,000 satisfied learners spread across the globe.Edureka's Python Programming Certification Training course is designed for students and professionals who want to be a Master in Python programming. The course is designed to give you a head start into Python programming and train you for both core and advanced concepts.Stay ahead of the curve in technology with This Post Graduate Program in AI and Machine Learning in partnership with E&ICT Academy, National Institute of Technology, Warangal. This Artificial intelligence course is curated to deliver the best results.Got a question for us? Please mention it in the comments section of this "Data Structures You Need to Learn with Python" blog and we will get back to you as soon as possible. Linear Data Structures are a type of data structure in computer science where data elements are arranged sequentially or linearly. Each element has a previous and next adjacent, except for the first and last elements. Sequential Organization: In linear data structures, data elements are arranged sequentially, one after the other. Each element has a unique predecessor (except for the first element) and a unique successor (except for the last element)Order Preservation: The order in which elements are added to the data structure is preserved. This means that the first element added will be the first one to be accessed or removed, and the last element added will be the last one to be accessed or removed.Fixed or Dynamic Size: Linear data structures can have either fixed or dynamic sizes. Arrays typically have a fixed size when they are created, while other structures like linked lists, stacks, and queues can dynamically grow or shrink as elements are added or removed.Efficient Access: Accessing elements within a linear data structure is typically efficient. For example, arrays offer constant-time access to elements using their index. Linear data structures are commonly used for organising and manipulating data in a sequential fashion. Some of the most common linear data structures include: Arrays: A collection of elements stored in contiguous memory locations.Linked Lists: A collection of nodes, each containing an element and a reference to the next node.Stacks: A collection of elements with Last-In-First-Out (LIFO) order.Queues: A collection of elements with First-In-First-Out (FIFO) order.1. Array An array is a collection of items of same data type stored at contiguous memory locations. Array Characteristics of Array Data Structure: Homogeneous Elements: All elements within an array must be of the same data type.Contiguous Memory Allocation: In most programming languages, elements in an array are stored in contiguous (adjacent) memory locations. Zero-Based Indexing: In many programming languages, arrays use zero-based indexing, which means that the first element is accessed with an index of 0, the second with an index of 1, and so on. Random Access: Arrays provide constant-time (O(1)) access to elements. This means that regardless of the size of the array, it takes the same amount of time to access any element based on its index.Types of arrays: One-Dimensional Array: This is the simplest form of an array, which consists of a single row of elements, all of the same data type. Elements in a 1D array are accessed using a single index.One-Dimensional ArrayTwo-Dimensional Array: A two-dimensional array, often referred to as a matrix or 2D array, is an array of arrays. It consists of rows and columns, forming a grid-like structure. Elements in a 2D array are accessed using two indices, one for the row and one for the column. Two-Dimensional Array:Multi-Dimensional Array: Arrays can have more than two dimensions, leading to multi-dimensional arrays. These are used when data needs to be organized in a multi-dimensional grid. Multi-Dimensional Array Types of Array operations:Accessing Elements: Accessing a specific element in an array by its index is a constant-time operation. It has a time complexity of O(1).Insertion: Appending an element to the end of an array is usually a constant-time operation, O(1) but insertion at the beginning or any specific index takes O(n) time because it requires shifting all of the elements.Deletion: Same as insertion, deleting the last element is a constant-time operation, O(1) but deletion of element at the beginning or any specific index takes O(n) time because it requires shifting all of the elements.Searching: Linear Search takes O(n) time which is useful for unsorted data and Binary Search takes O(logn) time which is useful for sorted data.2. Linked List A Linked List is a linear data structure which looks like a chain of nodes, where each node contains a data field and a reference(link) to the next node in the list. Unlike Arrays, Linked List elements are not stored at a contiguous location. Common Features of Linked List:Node: Each element in a linked list is represented by a node, which contains two components:Data: The actual data or value associated with the element.Next Pointer(or Link): A reference or pointer to the next node in the linked list.Head: The first node in a linked list is called the "head." It serves as the starting point for traversing the list.Tail: The last node in a linked list is called the "tail." Types of Linked Lists:Singly Linked List: In this type of

linked list, every node stores the address or reference of the next node in the list and the last node has the next address or reference as NULL. For example: 1->2->3->4->NULL Singly Linked List Doubly Linked Lists: In a doubly linked list, each node has two pointers: one pointing to the next node and one pointing to the previous node. This bidirectional structure allows for efficient traversal in both directions. Doubly Linked Lists Circular Linked Lists: A circular linked list is a type of linked list in which the first and the last nodes are also connected to each other to form a circle, there is no NULL at the end. Circular Linked Lists Types of Linked List operations:Accessing Elements: Accessing a specific element in a linked list takes O(n) time since nodes are stored in non conitgous locations so random access if not possible.Searching: Searching of a node in linked list takes O(n) time as whole list needs to travesed in worst case.Insertion: Insertion takes O(1) time if we are at the position where we have to insert an element.Deletion: Deletion takes O(1) time if we know the position of the element to be deleted.3. Stack Data StructureA stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle, meaning that the last element added to the stack is the first one to be removed. Stack Data structure Types of Stacks:Fixed Size Stack: As the name suggests, a fixed size stack has a fixed size and cannot grow or shrink dynamically. If the stack is full and an attempt is made to add an element to it, an overflow error occurs. If the stack is empty and an attempt is made to remove an element from it, an underflow error occurs.Dynamic Size Stack: A dynamic size stack can grow or shrink dynamically. When the stack is full, it automatically increases its size to accommodate the new element, and when the stack is empty, it decreases its size. This type of stack is implemented using a linked list, as it allows for easy resizing of the stack.Stack Operations:push(): When this operation is performed, an element is inserted into the stack.pop(): When this operation is performed, an element is removed from the top of the stack and is returned.top(): This operation will return the last inserted element that is at the top without removing it.size(): This operation will return the size of the stack i.e. the total number of elements present in the stack.isEmpty(): This operation indicates whether the stack is empty or not.4. Queue Data StructureA queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. In a queue, the first element added is the first one to be removed. Queue Data Structure Types of Queue:Input Restricted Queue: This is a simple queue. In this type of queue, the input can be taken from only one end but deletion can be done from any of the ends.Output Restricted Queue: This is also a simple queue. In this type of queue, the input can be taken from both ends but deletion can be done from only one end.Circular Queue: This is a special type of queue where the last position is connected back to the first position. Here also the operations are performed in FIFO order. To know more refer this.Double-Ended Queue (Dequeue): In a double-ended queue the insertion and deletion operations, both can be performed from both ends. To know more refer this.Priority Queue: A priority queue is a special queue where the elements are accessed based on the priority assigned to them. To know more refer this.Queue Operations:Enqueue(): Adds (or stores) an element to the end of the queue..Dequeue(): Removal of elements from the queue.Peek() or front(): Acquires the data element available at the front node of the queue without deleting it.rear(): This operation returns the element at the rear end without removing it.isFull(): Validates if the queue is full.isNull(): Checks if the queue is empty.Advantages of Linear Data StructuresEfficient data access: Elements can be easily accessed by their position in the sequence.Dynamic sizing: Linear data structures can dynamically adjust their size as elements are added or removed.Ease of implementation: Linear data structures can be easily implemented using arrays or linked lists.Versatility: Linear data structures can be used in various applications, such as searching, sorting, and manipulation of data.Simple algorithms: Many algorithms used in linear data structures are simple and straightforward.Disadvantages of Linear Data StructuresLimited data access: Accessing elements not stored at the end or the beginning of the sequence can be time-consuming.Memory overhead: Maintaining the links between elements in linked lists and pointers in stacks and queues can consume additional memory.Complex algorithms: Some algorithms used in linear data structures, such as searching and sorting, can be complex and time-consuming.Inefficient use of memory: Linear data structures can result in inefficient use of memory if there are gaps in the memory allocation.Unsuitable for certain operations: Linear data structures may not be suitable for operations that require constant random access to elements, such as searching for an element in a large dataset.