UVA DATA SCIENCE

Foundation of Computer Science for Data Science

Linear data structure: Array, Dynamic Array, Linked list

Mai Dahshan

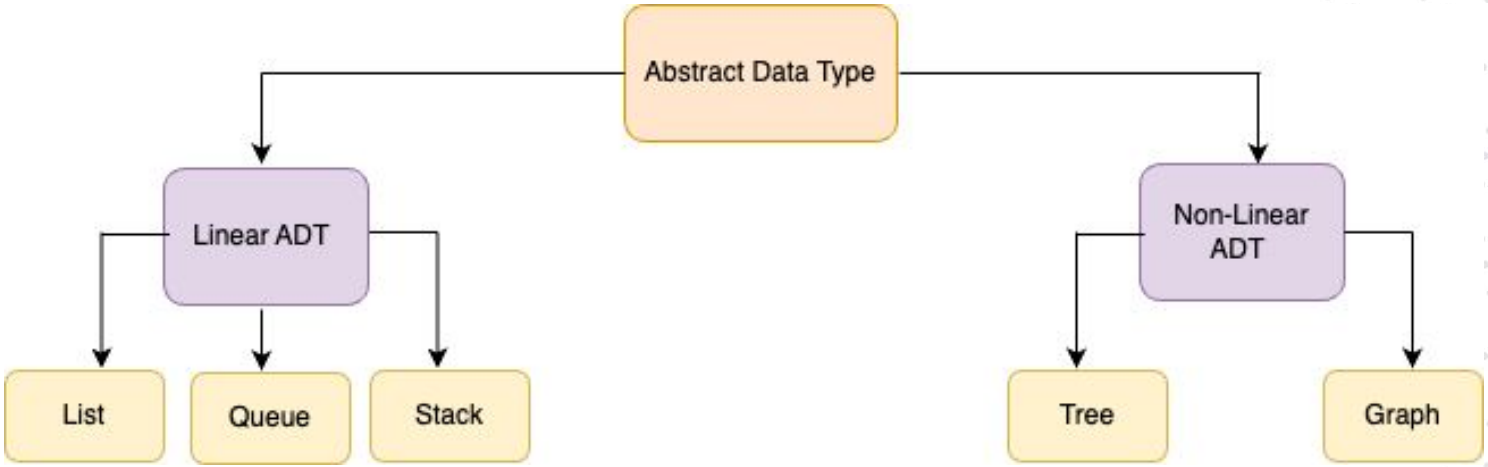
September 9, 2024

UVA DATA SCIENCE

Learning Objectives

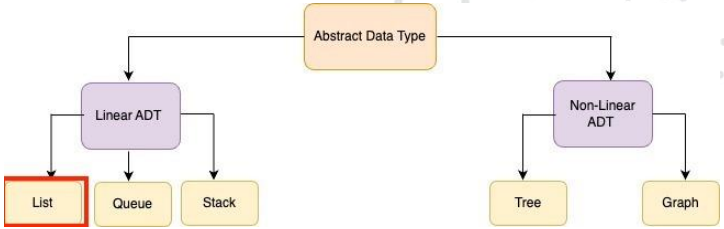
- Implement Linear ADTs
- Understand linear data structures
- Analyze the performance of linear data structures
- Implement linear data structures in Python
- Apply linear data structures to Data Science Problems
- Understand and implement searching algorithms for linear data structures

ADTs


Abstract Data Type
Linear ADT
Non-Linear ADT
List
Queue
Stack
Tree
Graph

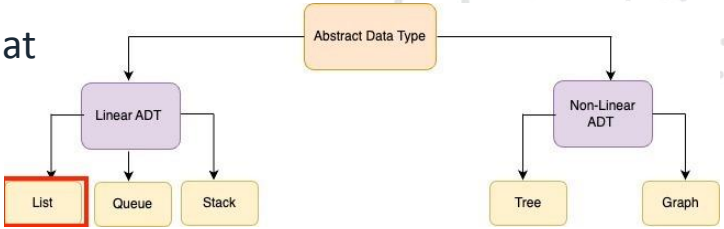
ADTs

- **List Abstract Data Type (ADT)** defines a collection of elements arranged in a sequential order. It abstracts the operations that can be performed on a list, without specifying the details of how these operations are implemented



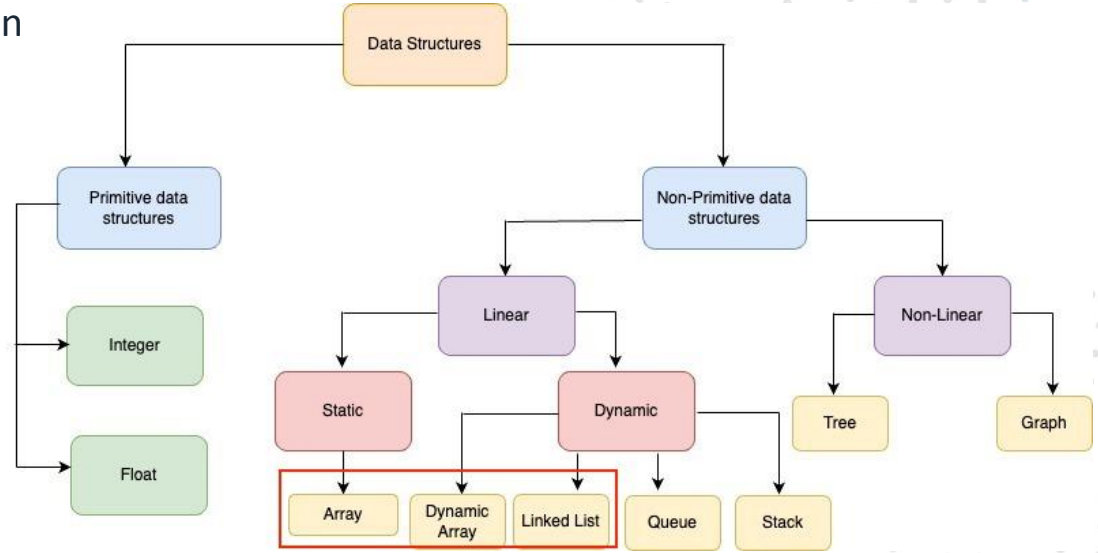
ADTs

- Key Operations of a List ADT
  - **Insertion**: Add an element at a specific position, at the beginning, or at the end of the list
  - **Deletion**: Remove an element from a specific position or based on the element's value
  - **Traversal**: Access each element in the list in order.
  - **Search**: Find an element in the list that matches a given value
  - **Sorting:-**Arranging the elements in logical order
  - **Update**: Modify the value of an element at a specific position



ADTs

- List ADT Implementations
  - Array-based List Implementation
  - Linked List Implementation

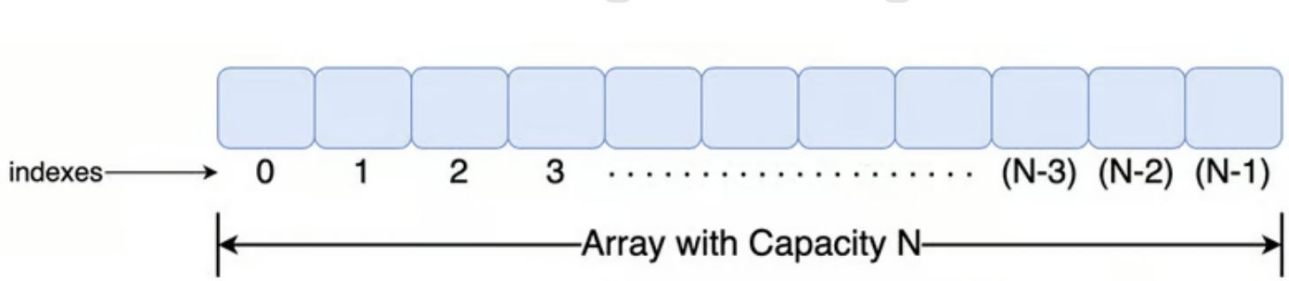


Linear Data Structures

- Data elements are arranged in a linear sequence.
- Iterate over all elements in a single run
- Elements can be accessed in a specific, linear order
- Data elements are organized in a single level, without any hierarchical or nested structure

Array

- An array is a static linear data structure with a fixed-size, indexed sequence of homogeneous data items
- Array elements are stored in consecutive memory locations
- The array length is fixed upon creation and cannot be altered



Example of 1D array of size n

Array Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Access			
Insertion at the end			
Insertion at beginning/middle			
Deletion at the end			
Deletion at beginning/middle			
Update value at given index			

Array Operations Visualization: https://visualgo.net/en/array

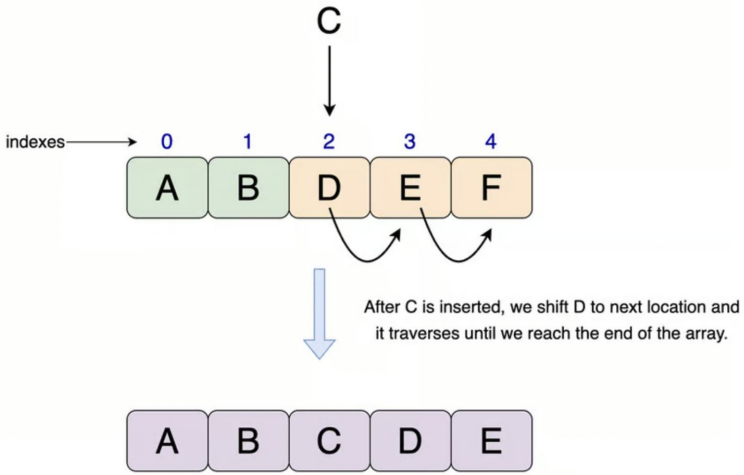
Array Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Access	O(1)	O(1)	O(1)
Insertion at the end	O(1)	O(1)	O(1)
Insertion at beginning/middle	O(n)	O(n)	O(n)
Deletion at the end	O(1)	O(1)	O(1)
Deletion at beginning/middle	O(n)	O(n)	O(n)
Update value at given index	O(1)	O(1)	O(1)

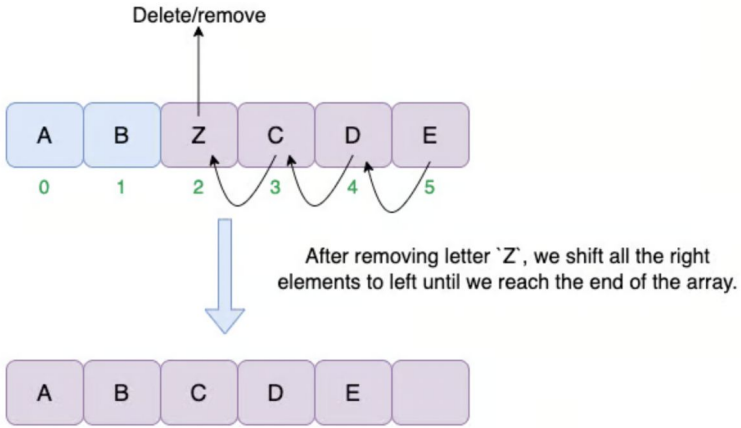
Array Operations Visualization: https://visualgo.net/en/array

Array Insertion and Deletion

Inserting an element in middle



Deleting an element in middle



Pros and Cons of Array

- Pros:
  - Fast lookups (random access)
  - Fast appends
  - Cache friendliness
- Cons:
  - Fixed size.
  - Memory unused or wasted.
  - Costly inserts
  - Costly deletes

Python Implementations of Array

- NumPy Array
- Array Module
- Tuple is like an array but w/ parentheses. Ex: myTuple = (1, 2, 3, 4)
  It is immutable and you can only index it or slice it
- Tensorflow, Pytorch tensor

Tensors are either mutable (TensorFlow) or immutable (Pytorch)

Python Implementation of Array

Feature	NumPy Array	Array Module	Tuple	Tensor
Mutability	Mutable elements can be changed	Mutable elements	Immutable	Mutable or immutable depending on the framework (e.g., TensorFlow, PyTorch)
Data Type Support	Supports multiple homogeneous data types, including complex numbers	Supports homogeneous data types	Supports heterogeneous data types	Supports homogeneous data types
Operations	Supports a wide range of mathematical and statistical operations	Limited operations; mainly basic array manipulations	Limited operations; mostly indexing and slicing	Supports a wide range of mathematical operations, including gradient computation in deep learning frameworks

Python Implementation of Array

Feature	NumPy Array	Array Module	Tuple	Tensor
Performance	Optimized for numerical operations with efficient performance	Less optimized suitable for basic needs	Not optimized for numerical operations	Optimized for high-performance computations in deep learning frameworks
Use Cases	Scientific computing, data analysis, numerical computations	Basic array manipulations, simple data storage	Data storage, fixed collections of elements	Machine learning, deep learning, high-performance computing

Uses of Array in Data Science

Q: What applications can we use arrays for in data science?

- **Text analysis**: Storing text data as arrays of word vectors, enabling operations like sentiment analysis for small/medium datasets
- **Regression analysis**: Using arrays to store both feature vectors and target values for training a regression model
- **Gradient Calculations**: In backpropagation of deep learning models, gradients of the loss function with respect to network parameters (weights and biases) are stored as arrays for efficient computation and updates
- **Deep learning libraries** like TensorFlow and PyTorch leverage the power of array operations to perform computations across entire arrays simultaneously, significantly improving processing speed compared to element-wise operations

If we're dealing with large datasets, you should use a numpy array or tensors (mutables). Arrays are great for smaller datasets.

Array

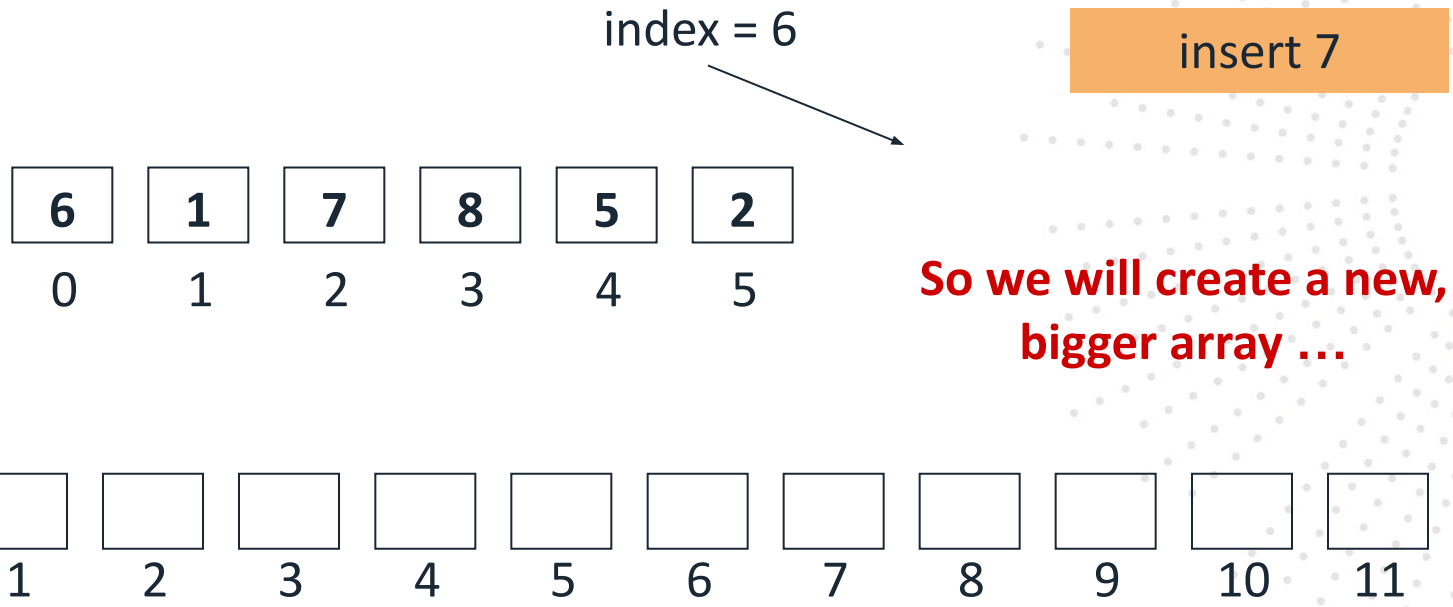
insert 2

Array

insert 7

The array is full and there is no room for a new item!

Array


index = 6
insert 7
6
1
7
8
5
2
0
1
2
3
4
5
So we will create a new, bigger array …
0
1
2
3
4
5
6
7
8
9
10
11

Dynamic Array

- A dynamic arrays is a linear data structure that allows resizing during runtime
  at some point in time, when you're array is full, you will need to double its size so that your data is still stored in contiguous memory
- They can grow or shrink as needed, making them more flexible than static arrays
- Like static arrays, elements are stored in contiguous memory locations

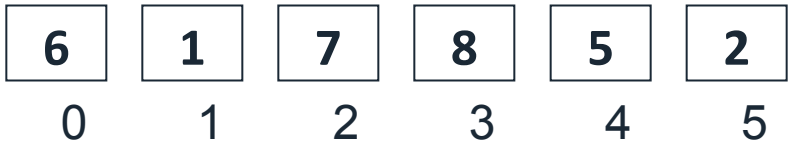
difference between dynamic array and reg array is that dynamic can grow and shrink based on the number of elements inside your array.

Dynamic Array

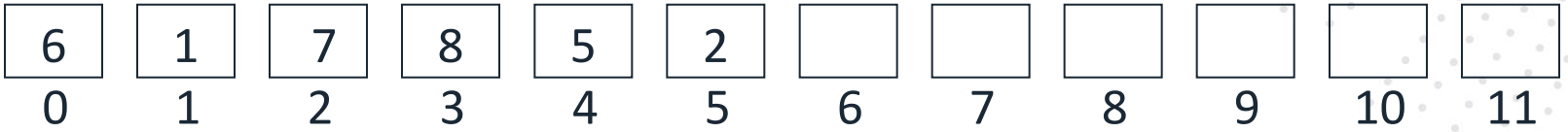
- How Dynamic Arrays Work
  - Typically implemented using a static array that doubles in size when capacity is exceeded.
  - When full, a new array is created, and all elements are copied over.
    - Usually doubles the size (growth factor of 2), but can vary.
    - It’s good, but… copying an array is computationally burdensome: O(n)

Dynamic Array

insert 7



… copy the elements of the old array into it new old

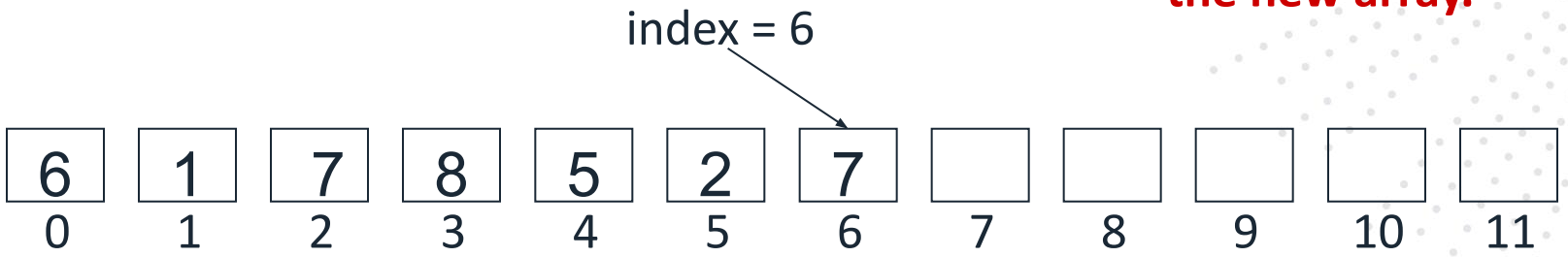


Dynamic Array

The old array will eventually be deleted by garbage collector

insert 7

… and finally insert 7 into the new array.



Dynamic Array

- How to minimize the number of copies
  - Difficult to do without knowing the max capacity needed
- Exponential Growth Scheme
  - When attempting to add element and capacity is full, allocate a new array with double capacity and copy.
  - How many copies are needed if we follow the above re-sizing rule?
    - Assume the max capacity is (which is unknown apriori)
    - Assume array is initialized to size *i*.
    - Total number of allocation-copy events: $i * log_2(N - i)$

Dynamic Array Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Access			
Insertion at the end			
Insertion at beginning/middle			
Deletion at the end			
Deletion at beginning/middle			
Update value at given index			
Resizing			

Dynamic Array Operations Visualization: https://visualgo.net/en/array

Dynamic Array Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Access	O(1)	O(1)	O(1)
Insertion at the end	O(1)	O(1) amortized cost	O(n)
Insertion at beginning/middle	O(n)	O(n)	O(n)
Deletion at the end	O(1)	O(1)	O(1)
Deletion at beginning/middle	O(n)	O(n)	O(n)
Update value at given index	O(1)	O(1)	O(1)
Resizing	O(1)	O(1) amortized cost	O(n)

Dynamic Array Operations Visualization: https://visualgo.net/en/array

Pros and Cons of Dynamic Array

- Pros:
  - Fast lookups (random access)
  - Flexible Size: Can dynamically adjust to accommodate new elements.
  - Cache friendliness
- Cons:
  - Resizing Cost
  - Memory Allocation

Python Implementation of Dynamic Array

- List
- Python using custom classes

Applications of Dynamic Array in Data Science

- In **online learning or real-time ML applications**, dynamic arrays can efficiently handle incoming data streams, allowing models to learn from data as it arrives.
- In **natural language processing (NLP) and time series analysis**, dynamic arrays store sequences of varying lengths, such as sentences or time series data points,

Dynamic Array

- What if we have a massive list and constantly need to add new data?
  No, b/c you are taking a lot of time copying items into the list. In this case, we need to avoid copying w/ linked lists!
- The copying process when the list reaches capacity can become cumbersome.
  - Is there a way to avoid this issue?

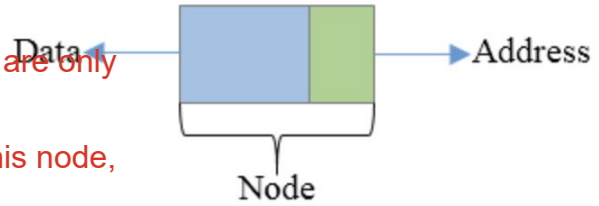
Linked List

- A linked list is a linear data structure where elements, called nodes, are stored in non-contiguous memory locations.
- Each node contains a data value and a reference(s) (or pointer(s))

W/ linked lists, there are elements scattered everywhere in memory and are connected with nodes.
A node has two components: 1) Value (data) and 2) Address

So they don't need to be in contiguous memory. There are only 2 steps

1) create the new node, 2) connect the pointer to the this node,

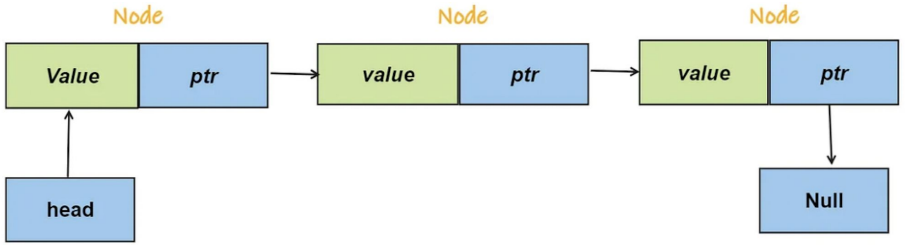


Types of Links List

- **Singly Linked List** has a head pointer only
- **Doubly Linked List** has a head and tail pointer
- **Circular Linked List**
- Skip List (Advanced)
- Unrolled Linked List (Advanced)

Singly Linked List

- A singly linked list is a linear data structure where each element (node) points to the next node in the sequence.
- It consists of a series of nodes connected in a single direction



Singly Linked List Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Access			
Insertion at the head			
Deletion at the head			

Singly linked list Operations Visualization: https://csvistool.com/LinkedList
https://visualgo.net/en/list

Singly Linked List Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Access	O(1)	O(n)	O(n)
Insertion at the head	O(1)	O(1)	O(1)
Deletion at the head	O(1)	O(1)	O(1)

Singly linked list Operations Visualization: https://csvistool.com/LinkedList
https://visualgo.net/en/list

Singly Linked List Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Insertion at the middle			
Insertion at the end			
Deletion at the middle			
Deletion at the end			

*Assume no direct reference

Singly Linked List Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Insertion at the middle	O(n)	O(n)	O(n)
Insertion at the end	O(n)	O(n)	O(n)
Deletion at the middle	O(n)	O(n)	O(n)
Deletion at the end	O(n)	O(n)	O(n)

*Assume no direct reference

- Inserting a new node is an O(1) operation if you have a reference to the node before the insertion point
- Deleting a node is O(1) if you have a reference to the node before the deletion point
- Finding the position in the middle or tail of the list dominates the time complexity, making it O(n)

Python Implementation of Singly Linked List

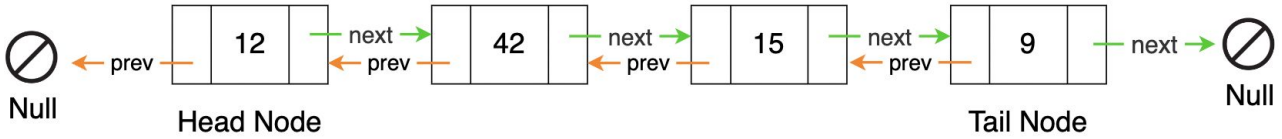
- Python using custom classes

Pros and Cons of Singly Linked List

- Pros:
  - Is able to grow in size as needed
  - Does not require the shifting of items during insertions and deletions
- Cons
  - Increased Memory Usage
  - Increased Overhead
  - Unidirectional Traversal
  - Accessing an element does not take a constant time

Doubly Linked List

- A doubly linked list is a linear data structure where each element (node) points to the next node and previous node in the sequence



Doubly Linked List Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Access			
Insertion at the head			
Deletion at the head			

Doubly linked list Operations Visualization:https://csvistool.com/DoublyLinkedList
https://visualgo.net/en/list

Doubly Linked List Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Access	O(1)	O(n)	O(n)
Insertion at the head	O(1)	O(1)	O(1)
Deletion at the head	O(1)	O(1)	O(1)

Doubly linked list Operations Visualization:https://csvistool.com/DoublyLinkedList
https://visualgo.net/en/list

Doubly Linked List Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Insertion at the middle			
Insertion at the end			
Deletion at the middle			
Deletion at the end			

*Assume tail pointer

Doubly linked list Operations Visualization:https://csvistool.com/DoublyLinkedList
https://visualgo.net/en/list

Doubly Linked List Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Insertion at the middle	O(n)	O(n)	O(n)
Insertion at the end	O(1)	O(1)	O(1)
Deletion at the middle	O(n)	O(n)	O(n)
Deletion at the end	O(1)	O(1)	O(1)

*Assume tail pointer

Doubly linked list Operations Visualization:https://csvistool.com/DoublyLinkedList
https://visualgo.net/en/list

Doubly Linked List Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Insertion at the middle			
Insertion at the end			
Deletion at the middle			
Deletion at the end			

*Assume no direct reference

Doubly linked list Operations Visualization:https://csvistool.com/DoublyLinkedList
https://visualgo.net/en/list

Doubly Linked List Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Insertion at the middle	O(n)	O(n)	O(n)
Insertion at the end	O(n)	O(n)	O(n)
Deletion at the middle	O(n)	O(n)	O(n)
Deletion at the end	O(n)	O(n)	O(n)

Insertion cost + Traversing cost

Deletion cost + Traversing cost

*Assume no direct reference

Doubly linked list Operations Visualization:https://csvistool.com/DoublyLinkedList
https://visualgo.net/en/list

Pros and Cons of Doubly Linked List

- Pros:
  - Is able to grow in size as needed
  - Does not require the shifting of items during insertions and deletions
  - Able to traverse the list in any direction
  - Reversing the list is simple
- Cons
  - Increased Memory Usage
  - Increased Overhead
  - Accessing an element does not take a constant time

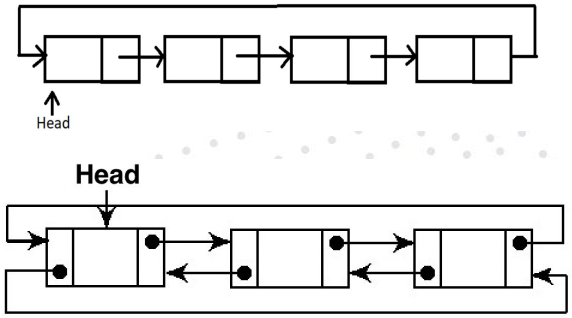
Python Implementation of Doubly Linked List

- Python using custom classes

Circular Linked List

- A circular linked list is a linear data structure where the last node points back to the first node, forming a circle

- Types of Circular Linked Lists
  - Singly Circular Linked List
  - Doubly Circular Linked List



Applications of Linked list in Data Science

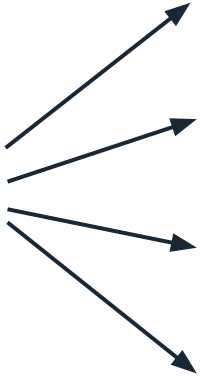
- **Text Processing:** Linked lists can be used to represent and manipulate strings or large texts where frequent insertions and deletions are required.

- **Replay Buffers:** In reinforcement learning, linked lists can be used to implement replay buffers that store past experiences or states for training.

Applications of Linear Data Structures in Data Science

- Q: Image classification, how many way to implement?

Is it a cat or dog?



1D Array: Feed forward net, Traditional ML

2D, 3D Array: CNN

Linked list: Recurrent Neural Net

Graph: Graph Neural Net

Linear Data Structure Activity

- For each case, identify the most suitable linear data structure to use
  - **Case 1:** You were asked to design an e-commerce application that allows users to freely modify their cart contents. The shopping cart should automatically update in real-time as items are added or removed. Which data structure would you use to store the cart contents?
  - **Case 2:** You were asked to design a weather monitoring system where data collection is restricted to a fixed timeframe. The system should store daily temperature readings for a specific city over the course of a week. Which data structure would you use to store the temperature?

Linear Data Structure Activity

- **Case 3:** You were asked to design an application that tracks visited URLs in the order they were accessed, allowing users to navigate backward or forward through their browsing history. Which data structure would you use to store visited URLs?
- **Case 4:** You are asked to design a web application that manages and displays items such search results, user comments, or product listings, which data structure would you use to store these items?
- **Case 5:** You are asked to design a music player program that handles a playlist, allowing users to frequently add and remove tracks, and also manages metadata for each track. which data structures would you use in this case?

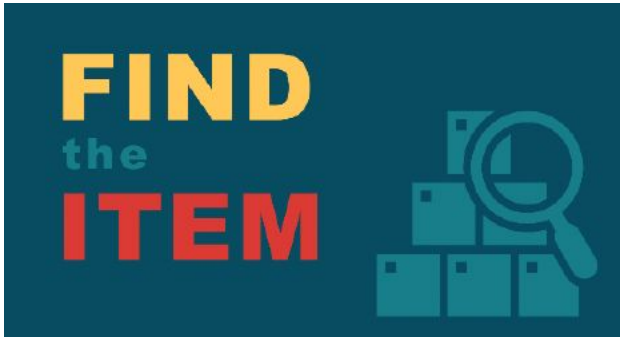
UVA DATA SCIENCE

Foundation of Computer Science for Data Science

Searching linear data structure

Searching

- As we gather and manage data, we want to search for specific items within a data container



Key Questions to Address Before Searching

- Before choosing a search algorithm, the following key questions need to be addressed:
  - What is the structure of the data (how is the data stored)?
  - What is to be searched: numbers, text or graphs?
  - How large is the set of elements to be searched?
  - Is data presorted in some order such as ascending or descending order?
  - What are the performance requirements? Are there constraints on time complexity (speed) or space complexity (memory usage)?

Why Learn Search Algorithms?

- Searching for data is one of the fundamental operations in computing
- Often, the efficiency of a search algorithm is what distinguishes a fast program from a slow one.
- In the age of big data, efficiently searching through data is crucial for maintaining a competitive edge

Searching Linear Data Structures

- Parameters that affect search algorithm selection:
  - Data size
  - Whether the list is **sorted**
  - Whether all the elements in the list are **unique** or have **duplicate** values
  - whether the search is performed frequency or not

Sequential/Linear Search

- Sequential/Linear search is a searching algorithm that finds a target value by sequentially checking each element in a list until the target is found or the end is reached.
    - **Input:** A collection of data (e.g., a list, array, or dataset) and a target value that needs to be located within this collection.
    - **Output:** The position or index of the target value in the collection if it is present; otherwise, a special value indicating that the target value is not found in the collection

Sequential/Linear Search

Algorithm

- Check if the list is Empty: if the list is empty, return -1 (sentinel value)
- Initialization: Start at the first element of the list (index 0)
- Iterate through list: Traverse each element of the list
  - Compare and check the current element with the target element.
    - If the current element matches the target element, return the index of the matching element
    - If the current element does not match the target element, move to the next element in the list
- If the end of the list is reached without finding the target element, the search is unsuccessful, and the algorithm returns a value indicating that the element is not in the list (-1 or 'None')

Search(A,n)
Input: An array $A[n]$, where $n \geq 1$; an item x
Output: Index where x occurs in A, or -1

for $i \leftarrow 0$ to $n-1$ do
    if $A[i] = x$ then return(i);
return(-1);

Sequential/Linear Search

- Examples

Sequential/Linear Search: Time Complexity

Operation	Best Case	Average Case	Worst Case
Item present			
Item not present			

Linear Search Visualization: https://www.cs.usfca.edu/~galles/visualization/Search.html

Sequential/Linear Search: Time Complexity

Operation	Best Case	Average Case	Worst Case
Item present	O(1)	O(n/2) = O(n)	O(n)
Item not present	O(n)	O(n)	O(n)

Linear Search Visualization: https://www.cs.usfca.edu/~galles/visualization/Search.html

Sequential/Linear Search

- Pros:
  - Linear search is easy to understand and implement.
  - Works on unsorted lists, so there's no need for pre-processing or sorting the data.
  - Can be used on any data structure that allows sequential access, such as arrays, linked lists, or files. Efficient for small lists
- Cons
  - The time complexity is O(n), which means the time taken grows linearly with the number of elements. This can be very slow for large lists.
  - Not Optimal for Repeated Searches

Binary Search

- Binary search is a searching algorithm for finding a target value within a sorted array or list by repeatedly dividing the search interval in half.
  - **Input:** A collection of sorted data (e.g., a list, array, or dataset) and a target value that needs to be located within this collection.
  - **Output:** The position or index of the target value in the collection if it is present; otherwise, a special value indicating that the target value is not found in the collection

Binary Search

Algorithm

- Check for Empty list: if the list is empty, return -1
- Initialize two variables: **low** set to index 0(lower bound of the search space), and **high** set to n-1 (upper bound of the search space).
- Traverse through the list while low<=high
  - Calculate the Midpoint (mid= floor((low+high)/2)
  - Compare the Midpoint Value with the Target
    - if target value == list[mid], then target found at index (mid). Return mid
    - if target value > list[mid], then low=mid+1
    - if target value < list[mid], then high=mid-1
- If low becomes greater than high, it means the target value is not present in the list. Return -1

```
BinarySearch(A[0..n − 1], K)
    //Implements nonrecursive binary search
    //Input: An array A[0..n − 1] sorted in ascending order and
    //       a search key K
    //Output: An index of the array's element that is equal to K
    //       or −1 if there is no such element
    l ← 0;   r ← n − 1
    while l ≤ r do
        m ← ⌊(l + r)/2⌋
        if K = A[m] return m  # Target found at index (mid)
        else if K < A[m] r ← m − 1 # Discard right part
        else l ← m + 1 # Discard left part
    return −1
```

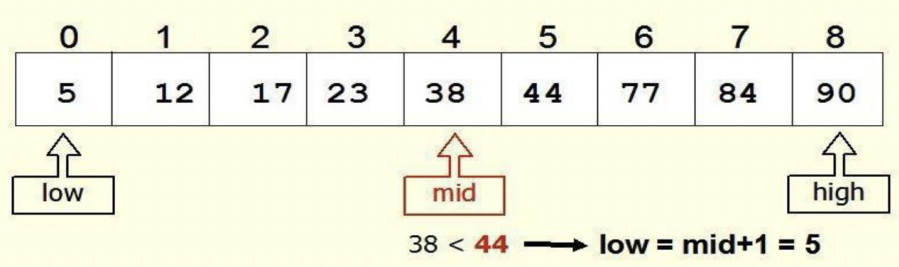
Binary Search

- How to find the midpoint?

$$\text{mid} = \left\lfloor \frac{\text{low}+\text{high}}{2} \right\rfloor$$

  - low: The starting index of the search range
  - high: The ending index of the search range

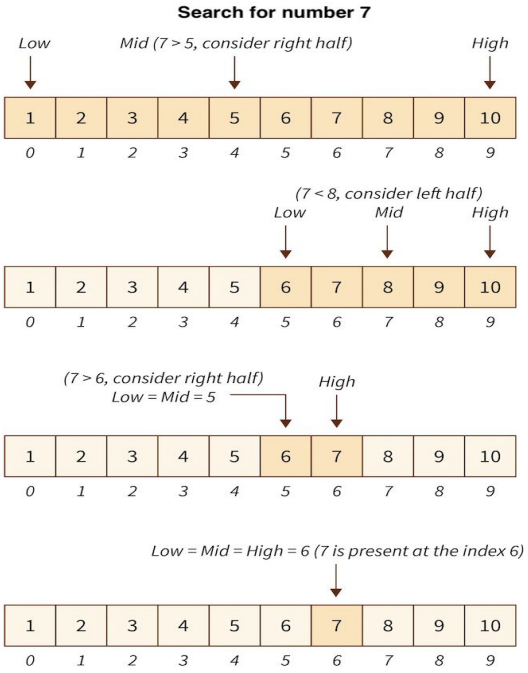
- Example



$$\text{mid} = \frac{0+8}{2} = \frac{8}{2} = 4$$

Binary Search

- Example



Binary Search: Time Complexity

Operation	Best Case	Average Case	Worst Case
Item present			
Item not present			

Binary Search Visualization: https://www.cs.usfca.edu/~galles/visualization/Search.html

Binary Search: Time Complexity

Operation	Best Case	Average Case	Worst Case
Item present	O(1)	O(log n)	O(log n)
Item not present	O(log n)	O(log n)	O(log n)

Binary Search Visualization: https://www.cs.usfca.edu/~galles/visualization/Search.html

Binary Search

- The data must be SORTED !
  - Before implementing Binary search, there is a non-optional pre-processing step that is required on the input data (in an array or list...) if not sorted
- Binary Search is also called half-interval search
  - No matter what is the size of the data, each iteration reduces the number of items to be searched by half
  - It’s a good example of the divide-and-conquer strategy (Complexity: O(lg n))

Binary Search

- Much more efficient than sequential search
  - Especially if search is done many times (sort once, search many times)
  - Might be worth the overhead (once) if you will use this algorithm multiple times afterwards

Binary Search Activity

- Some binary search inputs will be provided
  - Array of int values, plus a **target** to find
- Respond with:
  - What **index** is returned, and
  - The **sequence of mid index values** that were compared to the target to get that answer

Binary Search Example #1

- Input: | -1 | 4 | 5 | 11 | 13 | and target 4

  Index returned is?

  mid indices (positions) compared to target-value:

1) -1 4
2) 0 1
3) 5 -1 4
4) 2 0 1
5) other

Binary Search Example #1 Solution

- Input: | -1 | 4 | 5 | 11 | 13 | and target 4

  Index returned is? **1**

  mid indices (positions) compared to target-value:

1) -1 4
2) 0 1
3) 5 -1 4
4) **2** **0** **1**
5) other

Binary Search Example #2

- Input:

-5	-2	-1	4	5	11	13	14	17	18

and target 3

Index returned is?

mid indices (positions) compared to target-value

1) 5 2 3 4
2) 4 1 2 3
3) 4 2 3 4
4) other

Binary Search Example #2 Solution

- Input:

-5	-2	-1	4	5	11	13	14	17	18

and target 3

Index returned is? **-1**

mid indices (positions) compared to target-value

1) 5 2 3 4
2) **4 1 2 3**
3) 4 2 3 4
4) other

Binary Search Example

- Input: | -1 | 4 | 5 | 11 | 13 | and target 13

  Index returned is?

  mid indices (positions) compared to target-value:

1) 0 1 2 3 4
2) 2 4
3) 2 3 4
4) other

Binary Search Example #3 Solution

- Input: | -1 | 4 | 5 | 11 | 13 | and target 13

  Index returned is? **4**

  mid indices (positions) compared to target-value:

1) 0 1 2 3 4
2) 2 4
3) **2** **3** **4**
4) other