A Starter Guide to Data Structures for AI and Machine Learning - KDnuggets

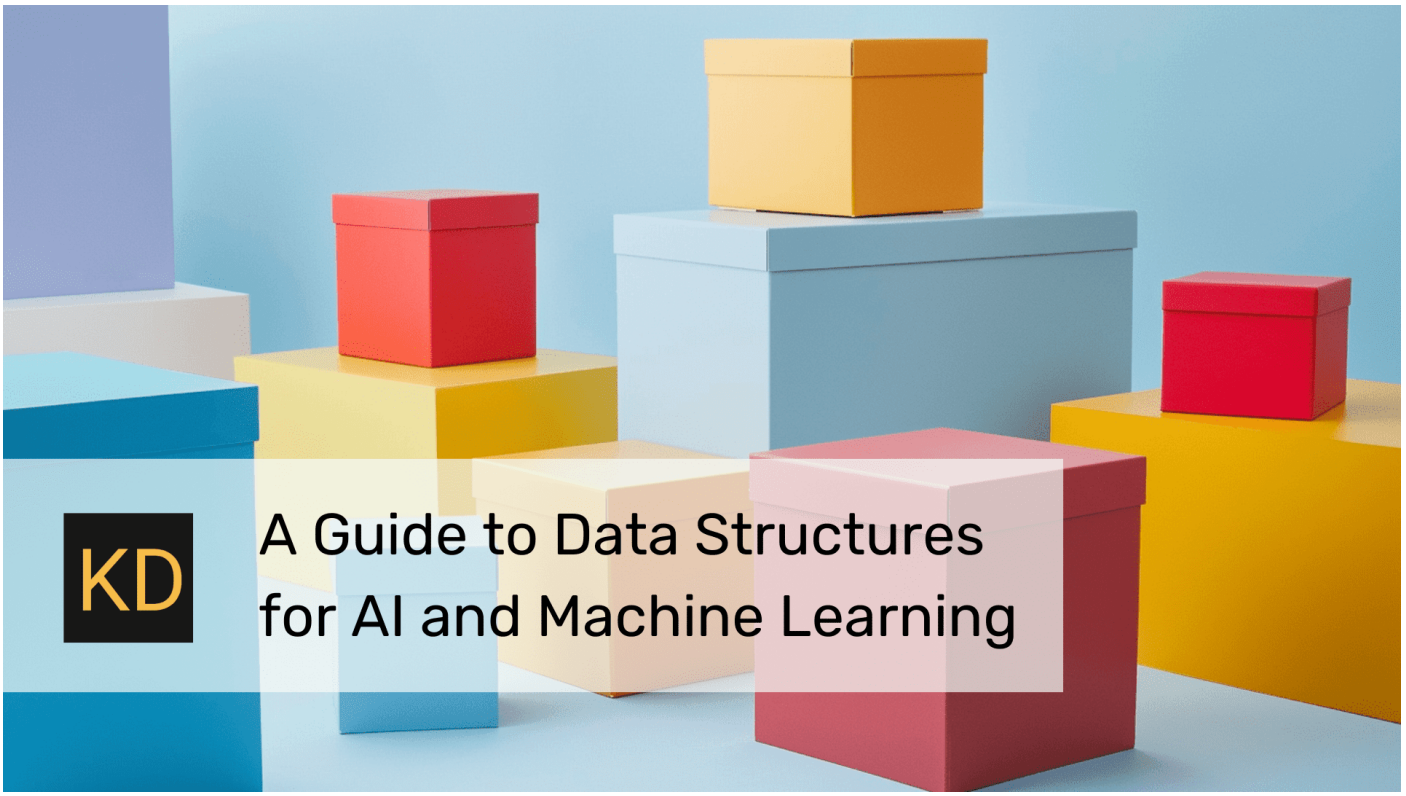

Image created by Author

Introduction

Data structures are, in a sense, the building blocks of algorithms, and are critical for the effective functioning of any AI or ML algorithm. These structures, while often thought of as simple containers for data, are more than that: they are incredibly rich tools in their own right, and can have a greater effect on the performance, efficiency, and overall computational complexity of algorithms than has been given credit. Choosing a data structure is therefore a task that requires careful thought, and can be determinate of the speed with which data can be processed, the scale to which an ML model can operate, or even of the feasibility of a given computational problem.

This article will introduce some data structures of importance in the fields of AI and ML and is aimed at both practictioners and students, as well as AI and ML enthusiasts. It is our hope in writing this article to supply some knowledge of important data structures in the AI and ML realms, as well as to provide some guidelines as to when and how these structures can be used effectively to their best advantage.

As we go through each of a series of data structures, examples will be given of AI and ML scenarios

in which they might be employed, each structure possessing its own set of strengths and weaknesses. Any implementations will be given in Python, a language of enormous popularity in the data science field, and are suitable for a variety of tasks in AI and ML. Mastering these core building blocks is essential for a variety of tasks that data scientists might face: sorting large data sets, creating high-performing algorithms that are both fast and light on memory, and maintaining data structures in a logical and efficient way to name but a few.

After starting with the basics of simple arrays and dynamic arrays, we will move on to more advanced structures, such as linked lists and binary search trees, before wrapping up with hash tables, a structure that is both very useful and can provide an excellent return on the investment of learning. We cover both the mechanical production of these structures, as well as their real-world use in AI and ML applications, a combination of theory and practice that provides the reader with the understanding needed to decide which is best for a particular problem, and to implement those structures in a robust AI system.

In this article we will dive into the various data structures pivotal for AI and machine learning, starting with arrays and dynamic arrays. By understanding the characteristics, advantages, and limitations of each data structure, practitioners can make informed choices that enhance the efficiency and scalability of their AI systems.

1. Arrays and Dynamically-Sizing Arrays

Perhaps the most basic of computer science data structures, an array is a collection of elements of the same type stored in adjacent memory locations, allowing direct random access to each element. Dynamic arrays, like the lists in Python, build on simple arrays, but adding automatic resizing, where additional memory is allocated as elements are added or removed. This auto-memory-allocating ability is at the heart of dynamic arrays. A few basic suggestions as to when arrays are best to use might include problems with a seemingly linear traversing of data or where the number of elements does not fluctuate in the slightest, such as datasets of unchangeable sizes that Machine Learning algorithms might ingest.

Let’s first discuss the upsides:

Easy access to elements by index: Quick retrieval operations, which is crucial in many AI and ML scenarios where time efficiency is key

Good for known or fixed-size problems: Ideal for when the number of elements is predetermined or changes infrequently

And the downsides:

Fixed size (for static arrays): Requires knowing the maximum number of elements in advance, which can be limiting

Costly insertions and deletions (for static arrays): Each insertion or deletion potentially requires

shifting elements, which is computationally expensive

Arrays, possibly because they are simple to grasp and their utility, can be found nearly anywhere in computer science education; they are a natural classroom subject. Having O(1), or constant, time-complexity when accessing a random element from a computer memory location endears it to systems where runtime efficiency reigns supreme.

In the world of ML, the array and dynamic array are crucial for being able to handle datasets and, usually, to arrange feature vectors and matrices. High-performance numerical libraries like NumPy use arrays in concert with routines that efficiently perform task across datasets, allowing for rapid processing and transformation of numerical data required for training models and using them for predictions.

A few fundamental operations performed with Python's pre-built dynamic array data structure, the list, include:

```
# Initialization
my_list = [1, 2, 3]

# Indexing
print(my_list[0])        # output: 1

# Appending
my_list.append(4)        # my_list becomes [1, 2, 3, 4]

# Resizing
my_list.extend([5, 6])   # my_list becomes [1, 2, 3, 4, 5, 6]
```

2. Linked Lists

Linked lists are another basic data structure, one consisting of a sequence of nodes. Each node in the list contains both some data along with a pointer to the next node in the list. A singly linked list is one that each node in the list has a reference to just the next node in the list, allowing for forward traversal only; a doubly linked list, on the other hand, has a reference to both the next and previous nodes, capable of forward and backward traversal. This makes linked lists a flexible option for some tasks where arrays may not be the best choice.

The good:

They are: dynamic expansions or contractions of linked lists occur with no additional overhead of reallocating and moving the entire structure

They facilitate fast insertions and deletions of nodes without requiring further node shifting, as an array might necessitate

The bad:

The unpredictability of the storage locations of elements creates poor caching situations, especially in contrast to arrays

The linear or worse access times required to locate an element by index, needing full traversal from head to find, are less efficient

They are especially useful for structures where the number of elements is unclear, and frequent insertions or deletions are required. Such applications make them useful for situations that require dynamic data, where changes are frequent. Indeed, the dynamic sizing capability of linked lists is one of their strong points; they are clearly a good fit where the number of elements cannot be predicted well in advance and where considerable waste could occur as a result. Being able to tweak a linked list structure without the major overhead of a wholesale copy or rewrite is an obvious benefit, particularly where routine data structure adjustments are likely to be required.

Though they have less utility than arrays in the realm of AI and ML, linked lists do find specific applications wherein highly mutable data structures with rapid modifications are needed, such as for managing data pools in genetic algorithms or other situations where operations on individual elements are performed regularly.

Shall we have a simple Python implementation of linked list actions? Sure, why not. Note that the following basic linked list implementation includes a Node class to represent each list element, and a LinkedList class to handle the operations on the list, including appending and deleting nodes.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node
```

```
    def delete_node(self, key):
        temp = self.head
        if temp and temp.data == key:
            self.head = temp.next
            temp = None
            return
        prev = None
        while temp and temp.data != key:
            prev = temp
            temp = temp.next
        if temp is None:
            return
        prev.next = temp.next
        temp = None

    def print_list(self):
        current = self.head
        while current:
            print(current.data, end=' ')
            current = current.next
        print()
```

Here is an explanation of the above code:

This LinkedList class is responsible for managing the linked list, which includes creation, appending data, deleting nodes, and displaying the list, and when initialized creates the head pointer, head, marks an empty linked list by default

The append method appends data to the end of a linked list, creating a new node either at the head of the list when it's empty, or traversing to the end of a non-empty list to add the new node

The delete_node method removes a node with a given key (data) by considering these three cases: target key is in the head node; target key is in another node in the list; no node holds the key

By setting pointers correctly, it is able to take out a node without sacrificing the order of remaining nodes

The print_list method walks the list starting at the head, printing the contents of each node, in sequence, allowing for a simple means of understanding the list

Here is an example of the above LinkedList code being used:

```
# Create a new LinkedList
```

```
my_list = LinkedList()

# Append nodes with data
my_list.append(10)
my_list.append(20)
my_list.append(30)
my_list.append(40)
my_list.append(50)

# Print the current list
print("List after appending elements:")
my_list.print_list()       # outputs: 10 20 30 40 50

# Delete a node with data '30'
my_list.delete_node(30)

# Print the list after deletion
print("List after deleting the node with value 30:")
my_list.print_list()       # outputs: 10 20 40 50

# Append another node
my_list.append(60)

# Print the final state of the list
print("Final list after appending 60:")
my_list.print_list()       # outputs: 10 20 40 50 60
```

3. Trees, particularly Binary Search Trees (BST)

Trees are an example of a non-linear data structure (compare with arrays) in which parent-child relationships exist between nodes. Each tree has a root node, and nodes may contain zero or more child nodes, in a hierarchical structure. A Binary Search Tree (BST) is a kind of tree that allows each node to contain up to two children, generally referred to as the left child and right child. In this type of tree, keys contained in a node must, respectively, either be greater than or equal to all nodes contained within its left subtree, or less than or equal to all nodes contained in its right subtree. These properties of BSTs can facilitate more efficient search, insert, and remove operations, provided that the tree remains balanced.

BST pros:

With respect to more commonly used data structures such as arrays or linked lists, BSTs facilitate

quicker access, insertion and deletion

And BST cons:

However, previously mentioned that BSTs will provide decreased performance when unbalanced/ skewed

This can cause operation time complexity to degrade to O(n) in the worst case

BSTs are particularly effective when many search, insert, or delete operations are required with respect to the dataset they are handling. They are certainly more appropriate when the data is accessed frequently in a dataset that undergoes frequent changes.

Moreover, trees represent an ideal structure for describing hierarchical data in a way creating a tree-like relationships between data, like files system or organizational chart. This makes them particularly useful in applications where this sort of hierarchical data structuring is of interest.

BSTs are able to assure search operations are quick due to their average O(log n) time complexity for access, insert, and delete operations. This makes them of particular interest for applications where swift data access and updates are necessary.

Decision trees, a type of tree data structure widely used for classification and regression tasks in machine learning, enable models to be constructed which predict the based off target variable from rules determined by the features. Trees also see wide use in AI, such as game programming; particularly in the case of games of strategy such as chess, trees are used to simulate scenarios and determine constraints which dictate optimal moves.

Here is an overview of how you can implement a basic BST, including insert, search and delete methods, using Python:

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

def insert(root, key):
    if root is None:
        return TreeNode(key)
    else:
        if root.val < key:
            root.right = insert(root.right, key)
        else:
            root.left = insert(root.left, key)
    return root
```

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

def deleteNode(root, key):
    if root is None:
        return root
    if key < root.val:
        root.left = deleteNode(root.left, key)
    elif(key > root.val):
        root.right = deleteNode(root.right, key)
    else:
        if root.left is None:
            temp = root.right
            root = None
            return temp
        elif root.right is None:
            temp = root.left
            root = None
            return temp
        temp = minValueNode(root.right)
        root.val = temp.val
        root.right = deleteNode(root.right, temp.val)
    return root

def minValueNode(node):
    current = node
    while current.left is not None:
        current = current.left
    return current
```

Explanation of the above code:

The foundation of a Binary Search Tree is the TreeNode class, which houses the node's value (val) and its left and right child node pointers (left and right)

The insert function is an implementation of the recursive strategy of inserting a value into the BST:

in the base case in which no root exists it creates a new TreeNode, and otherwise it puts keys larger than itself to its right subtree, and smaller nodes to the left, preserving the BST's structure

The search function handles the base cases of no node with the specified value being found and not finding the specified root's value, and then searches recursively in the correct subtree based on the value of the key being compared to the current node

The delete_node method can be split into three cases: like a delete call for a key without children (replaced by the right child); one without a right child (replaced by the left child); and delete on a node with two children (replaced by its 'inorder successor', the smallest value in its right subtree), making the recursive node deletions and maintaining BST structure

A helper function is that of finding the minimum-value node (i.e. the leftmost node) of a subtree, which is utilized during the deletion of a node with two children

Here is an example of the above BST code implementation being used.

```
# Create the root node with an initial value
root = TreeNode(50)

# Insert elements into the BST
insert(root, 30)
insert(root, 20)
insert(root, 40)
insert(root, 70)
insert(root, 60)
insert(root, 80)

# Search for a value
searched_node = search(root, 70)
if searched_node:
    print(f"Found node with value: {searched_node.val}")
else:
    print("Value not found in the BST.")

# output -> Found node with value: 70

# Delete a node with no children
root = deleteNode(root, 20)

# Attempt to search for the deleted node
searched_node = search(root, 20)
if searched_node:
```

```
    print(f"Found node with value: {searched_node.val}")
else:
    print("Value not found in the BST - it was deleted.")

# output -> Value not found in the BST - it was deleted.
```

4. Hash Tables

Hash tables are a data structure well-suited to rapid data access. They harness a hash function to compute an index into a series of slots or buckets, out of which the desired value is returned. Hash tables can deliver almost instant data access thanks to these hash functions, and can be used to scale to large datasets with no decrease in access speed. The efficiency of hash tables relies heavily on a hash function, which evenly distributes entries across an array of buckets. This distribution helps to avoid key collisions, which is when different keys resolve to the same slot; proper key collision resolution is a core concern of hash table implementations.

Pros of hash tables:

Rapid data retrieval: Provides average-case constant time complexity (O(1)) for lookups, insertions, and deletions

Average time complexity efficiency: Mostly consistently swift, which makes hash tables suited to real-time data handling in general

Cons of hash tables:

Worst-case time complexity not great: Can degrade to O(n) if there are many items hashing to the same bucket

Reliant on a good hash function: The importance of the hash function to hash table performance is significant, as it has a direct influence on how well the data is distributed amongst the buckets

Hash tables are most often used when rapid lookups, insertions, and deletions are required, without any need for ordered data. They are particularly useful when quick access to items via their keys is necessary to make operations more rapid. The constant time complexity property of hash tables for their basic operations makes them extremely useful when high performance operation is a requirement, especially in situations where time is of the essence.

They are great for dealing with massive data, since they provide a high speed way for data lookup, with no performance degredation as the size of the data grows. AI often needs to handle huge amounts of data, where hash tables for retrieval and lookup make a lot of sense.

Within machine learning, hash tables help with feature indexing large data collections - in preprocessing and model training, quick access and data manipulation facilitated via hash tables. They can also make certain algorithms perform more efficiently - in some cases, during k-nearest neighbors calculation, they can store already computed distances and recall them from a hash table

to make large dataset calculations quicker.

In Python, the dictionary type is an implementation of hash tables. How to make use of Python dictionaries is explained below, with a collision handling strategy as well:

```
# Creating a hash table using a dictionary
hash_table = {}

# Inserting items
hash_table['key1'] = 'value1'
hash_table['key2'] = 'value2'

# Handling collisions by chaining
if 'key1' in hash_table:
    if isinstance(hash_table['key1'], list):
        hash_table['key1'].append('new_value1')
    else:
        hash_table['key1'] = [hash_table['key1'], 'new_value1']
else:
    hash_table['key1'] = 'new_value1'

# Retrieving items
print(hash_table['key1'])

# output: can be 'value1' or a list of values in case of collision

# Deleting items
del hash_table['key2']
```

Conclusion

An investigation of a few of the data structures underpinning AI and machine learning models can show us what some of these rather simple building blocks of the underlying technology are capable of. The inherent linearity of arrays, the adaptability of linked lists, the hierarchical organization of trees, and the O(1) search time of hash tables each offer different benefits. This understanding can inform the engineer as to how they can best leverage these structures %mdash; not only in the machine learning models and training sets they put together, but in the reasoning behind their choices and implementations.

Becoming proficient in elementary data structures with relevance to machine learning and AI is a skill that has implications. There are lots of places to learn this skill-set, from university to workshops to online courses. Even open source code can be an invaluable asset in getting familiar

with the disciplinary tools and best practices. The practical ability to work with data structures is not one to be overlooked. So to the data scientists and AI engineers of today, tomorrow, and thereafter: practice, experiment, and learn from the data structure materials available to you.

Matthew Mayo (**@mattmayo13**) holds a Master's degree in computer science and a graduate diploma in data mining. As Managing Editor, Matthew aims to make complex data science concepts accessible. His professional interests include natural language processing, machine learning algorithms, and exploring emerging AI. He is driven by a mission to democratize knowledge in the data science community. Matthew has been coding since he was 6 years old.