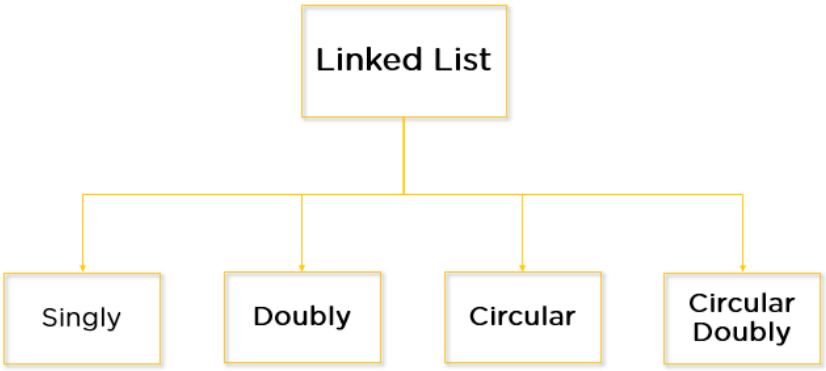
1. **What is a linked list? Compare and contrast different types of linked lists.**

A *linked list* is a data structure that stores information in sequential order. It is made up of a series of nodes. Each node contains two pieces of information: a value representing the data we want to store and a link to the node after it in the list. Pointers are used to create links between nodes, which explicitly refer to the memory locations where the nodes are stored (Fossati, 2008).

The primary advantage of a linked list over an array is that it can grow and shrink dynamically as elements are added or removed. In contrast, the size of an array is fixed, and if you want to add or remove parts, you may need to allocate a new collection and copy the existing elements over (GeeksforGeeks, 2011).

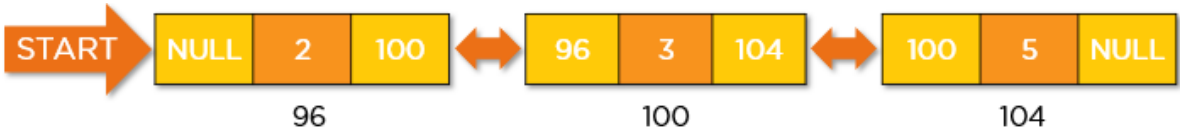
Because the elements of linked lists are not stored in contiguous memory locations, they differ from arrays and other sequential data structures. Instead, each node can be located anywhere in memory, and each node only stores a reference to the next node; this means linked lists can be more efficient than arrays for specific operations, such as inserting or deleting elements in the list's middle.

According to Ravikiran (2023), there are various linked lists, each with a different number of pointers and a different direction in which the pointers point. There are four types of linked lists, including:

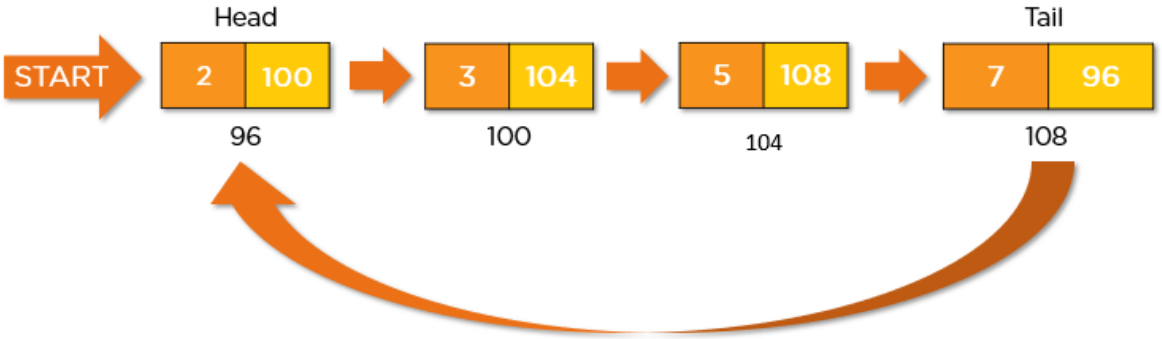


Singly-linked list: Each node contains a value and a pointer to the next node; this is the simplest form of a linked list and is commonly used for implementing stacks, queues, and hash tables.

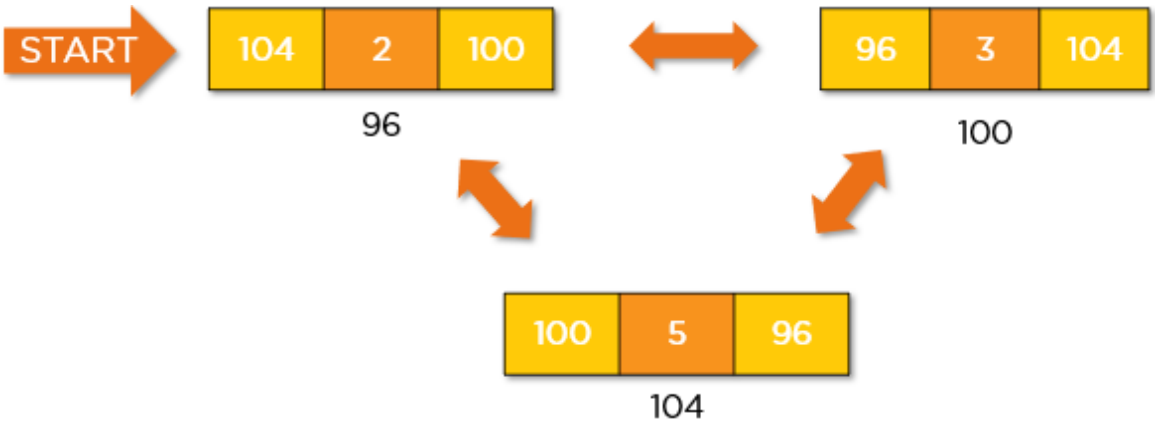
Doubly linked list: Each node contains a value, a pointer to the next node, and a pointer to the previous node; this allows for easy traversal in both directions but requires more memory to store the extra tips.



Circular linked list: A linked list variant in which the last node in the list points back to the first, forming a circular chain; this means that the last node's next pointer points to the first node instead of null.



A circular doubly linked list is a hybrid of a doubly and a circular linked list. Like the doubly linked list, it has an extra pointer called the previous pointer, and its last node, like the circular linked list, points at the head node. This kind of linked list is bi-directional. As a result, you can travel in both directions.



In conclusion, linked lists are versatile data structures representing various collections. Linked lists offer different trade-offs regarding memory usage, traversal efficiency, and implementation ease.

2. **Why would you prefer to implement a linked list over an array? What are the problems it which may be encountered when implementing a linked list over an array?**

For several reasons, you might prefer to implement a linked list over an array. When it comes to memory allocation, linked lists outperform arrays. In contrast to arrays, the size of a linked list is not fixed, allowing it to grow or shrink as the programme runs. This is possible because pointers must be updated when inserting or deleting from a linked list (Krohn, 2019).

Linked lists also have a variable size and are simple to insert/delete. However, with linked lists, random access is not permitted. We must access elements sequentially, beginning with the first node. As a result, we cannot perform a binary search using linked lists (GeeksforGeeks, 2011).

In summary, linked lists are a strong data structure that, in some cases, can be more flexible and efficient than arrays, mainly when dealing with dynamic collections. However, they have drawbacks, such as slower random access and additional memory overhead.

3. **Discuss the advantages and disadvantages of stacks and queues implemented using a linked list.**

According to Karumanchi (2015), stacks and queues are abstract data types that can be implemented with the help of a linked list. For instance, there are several reasons why you should use a linked list instead of an array:

- **3.1.1 Dynamic size:** A linked list can grow or shrink dynamically as elements are added or removed. On the other hand, an array has a fixed size that must be specified in advance, and expanding or contracting it necessitates creating a new array and copying the elements over.
- **3.1.2 Efficient insertion and deletion:** Inserting or deleting an element in a linked list requires only a few pointers to be updated. However, doing the same operation in an array may necessitate shifting many elements.
- **3.1.3 Memory efficiency:** Because linked lists do not require a contiguous memory block, they can be more memory efficient than arrays.

However, using a linked list instead of an array has some drawbacks:

- **3.2.1 Random access is slower:** Unlike an array, where you can directly access any element, a linked list requires you to traverse the list from the beginning to find a specific element. This slows down random access.
- **3.2.2 Additional memory overhead:** Each node in a linked list requires additional memory to store the pointer to the next node, whereas an array does not.
- **3.2.3 Cache inefficiency:** Because arrays are more cache-friendly than linked lists, accessing elements in an array can be faster. When you access an array element, the adjacent elements are likely also in the cache. On the other hand, accessing an element in a linked list may necessitate loading the entire node into the cache, which may require more efficient caching.

In conclusion, linked list implementations of stacks and queues can provide dynamic size and efficient insertion and deletion, but at the expense of additional memory overhead, slower access time, and cache inefficiency. The choice between using a linked list or an array to implement stacks and queues is determined by the application's specific requirements, such as the collection size, the frequency of access and modification operations, and the system's memory constraints.

4. **Construct a tree for the given inorder and postorder traversals:**

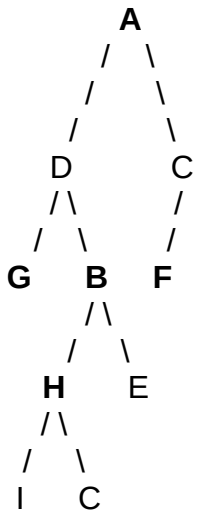
Inorder: D G B A H E I C F

Postorder: G D B H I E F C A

As explained by Thapar (N.D.), to construct a binary tree from its inorder and postorder traversals, we can follow these steps:

4.1 The tree's root is always the last element in the postorder traversal. The root is "A" in this case.
4.2 In the inorder traversal, find the index of the root. The left subtree is everything to the left of the root index, and everything to the right of the root index is the right subtree.
4.3 Build the left subtree recursively using the elements to the left of the root index and the right subtree recursively using the elements to the right of the root index.

Using these steps, we can construct the following tree for the given inorder and postorder traversals:

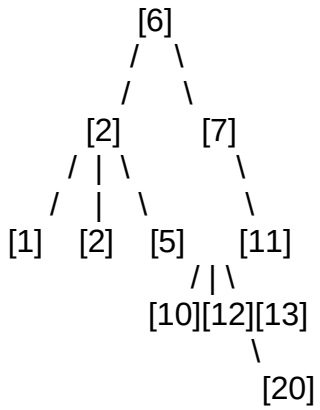


5. **Multiway search tree is used for disc access. Construct a B tree of order 5 for the following data:** 1 6 7 2 11 5 10 13 12 20

We can use the following steps to build a B-tree of order 5 for the given data based on the ideas of Walker (2020):

1. Begin with an empty order 5 B-tree.
2. Insert the first key, 1, at the tree's root.

3. In the root node, enter the second key, 6.
4. In the root node, enter the third key, 7.
5. In the root node, enter the fourth key, 2.
6. Divide the root node into two parts using the keys [2, 6] and [1, 7].
7. The median key, 6, is promoted to the parent node.
8. Insert the fifth key, 11, into the root's right child node.
9. Insert the sixth key, 5, into the root's left child node.
10. Divide the root's right child node into two nodes using the keys [5, 10, 11] and [12, 13, 20].
11. The median key, 11, is promoted to the parent node.
12. Insert the seventh key, 10, into the left child node of the root's right child node.
13. Insert the eighth key, 13, into the right child node of the root's right child node.
14. Insert the ninth key, 12, into the left child node of the root's right child node.
15. Insert the tenth key, 20, into the right child node of the root's right child node.
16. The B-tree is now finished.



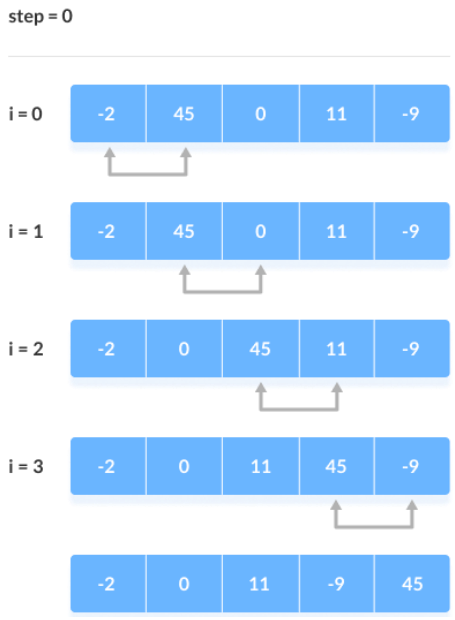
6. Write the steps for the bubble sort using natural language or flowchart and then write an algorithm for bubble sort using appropriate notations, i.e., pseudocode.

Bubble sort is a simple sorting algorithm that steps through the list of elements to be sorted repeatedly, compares each pair of adjacent elements, and swaps them if they are out of order. The list is passed through again and again until it is sorted. The algorithm gets its name from how more minor elements "bubble" to the top of the list as passed through multiple times (Min, 2010).

6.1 The steps for bubble sort are as follows:

1. Start from the first element of the array and compare it with the next element.
2. If the first element is greater than the next element, swap them.
3. Move to the next element and repeat step 2.
4. Continue until the end of the array.

5. Repeat steps 1 to 4 for the remaining unsorted portion of the array until the entire array is sorted.



(programiz, 2020).

6.2 Here is an example of a Bubble Sort Algorithm:

```
bubbleSort(array)
  for i <- 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap leftElement and rightElement
end bubbleSort
```

6.3 Bubble Sort Code in Python:

```
# Bubble sort in Python

def bubbleSort(array):

  # loop to access each array element
  for i in range(len(array)):

    # loop to compare array elements
    for j in range(0, len(array) - i - 1):

      # compare two adjacent elements
      # change > to < to sort in descending order
      if array[j] > array[j + 1]:

        # swapping elements if elements
        # are not in the intended order
        temp = array[j]
        array[j] = array[j+1]
        array[j+1] = temp


data = [-2, 45, 0, 11, -9]

bubbleSort(data)
```

7. **Write the steps for a quick sort.**

Quick Sort is a widely used sorting algorithm that employs a divide-and-conquer strategy to sort an array (Programiz, N.D.). For instance, here are the steps for Quick Sort:

1. Select a pivot element from the array. The pivot element can be any element in the array.
2. Partition the array around the pivot element such that all the elements less than the pivot are moved to the left of the pivot, and all the elements greater than the pivot are moved to the right of the pivot.
3. Recursively apply steps 1 and 2 to the left and right sub-arrays until the entire array is sorted.

8. **Write an algorithm for a sequential search.**

Sequential Search (also known as Linear Search) is a simple searching algorithm that works by iterating through an array of elements and comparing each element with the target element until a match is found or the end of the array is reached (Programiz, N.D.). For instance, here is the algorithm for Sequential Search:

```
LinearSearch(array, key)
  for each item in the array
    if item == value
      return its index
```

9. **Write steps for a binary search using natural language or flowchart and then write an algorithm for binary search using appropriate notation, i.e., pseudocode.**

Binary Search is a searching algorithm for determining the position of an element in a sorted array (Programiz, N.D.). For instance, here are the steps for Binary Search:

1. Begin with the array's middle element.
2. Return the index of the middle element if it equals the target value.
3. If the middle element is greater than the target value, repeat step 1 with the array's left half.
4. If the middle element is less than the target value, repeat step 1 with the array's right half.

5. If there are no more elements to search in the array, return -1.

Here is an example of binary search algorithm pseudocode:

```
function binarySearch(array, target):
    left = 0
    right = length(array) - 1
    while left <= right:
        mid = floor((left + right) / 2)
        if array[mid] == target:
            return mid
        else if array[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

10. Suppose you are doing a sequential search of the list [15, 18, 2, 19, 18, 0, 8, 14, 19, 14]. How many comparisons would you need to do in order to find the key 18? Show the steps.

Using sequential search to find the key 18 in the list [15, 18, 2, 19, 18, 0, 8, 14, 19, 14], we would start at the first element and compare it to the key. If it does not match, we move to the next element and repeat the process until we find the key or reach the end of the list.

Here are the steps to find key 18 using sequential search:

Index: 0 1 2 3 4 5 6 7 8 9
Value: [15, 18, 2, 19, 18, 0, 8, 14, 19, 14]

Comparison 1: Compare the key with the first element (15). Not a match.
Comparison 2: Compare the key with the second element (18). Match found! Return index of the second element (1).

Therefore, we only needed one comparison to find the key 18 in this list using sequential search.

11. Suppose you have following list of numbers to sort [19, 1, 9, 7, 3, 10, 13, 15, 8, 12]. Show the partially sorted list after three complete phases of bubble sort.

After three complete phases of bubble sorting, we must perform the following steps to partially sort the list [19, 1, 9, 7, 3, 10, 13, 15, 8, 12]:

First Phase:

- Compare 19 with 1, swap (1, 19)
- Compare 19 with 9, swap (9, 19)
- Compare 19 with 7, swap (7, 19)
- Compare 19 with 3, swap (3, 19)
- Compare 19 with 10, swap (10, 19)
- Compare 19 with 13, swap (13, 19)
- Compare 19 with 15, swap (15, 19)
- Compare 19 with 12, swap (12, 19)

List after first phase: [1, 9, 7, 3, 10, 13, 15, 12, 19, 8]

Second Phase:

- Compare 1 with 9, no swap
- Compare 9 with 7, swap (7, 9)
- Compare 9 with 3, swap (3, 9)
- Compare 10 with 13, no swap
- Compare 13 with 15, no swap
- Compare 15 with 12, swap (12, 15)
- Compare 19 with 8, swap (8, 19)

List after second phase: [1, 7, 3, 9, 10, 13, 12, 8, 15, 19]

Third Phase:

- Compare 1 with 7, no swap
- Compare 7 with 3, swap (3, 7)
- Compare 7 with 9, no swap
- Compare 9 with 10, no swap
- Compare 10 with 13, no swap
- Compare 13 with 12, swap (12, 13)
- Compare 15 with 8, swap (8, 15)

List after third phase: [1, 3, 7, 9, 10, 12, 8, 13, 15, 19]

After three complete phases of bubble sorting, the partially sorted list would be [1, 3,

7, 9, 10, 12, 8, 13, 15, 19].

12. Given the statement below

```
x = BinaryTree('a')
insert_left(x,'b') insert_right(x,'c')
insert_right(get_right_child(x),'d')
insert_left(get_right_child(get_right_child(x)),'e')
```

Which of these answers is the correct representation of the tree? Show your reasoning for your chosen solution.

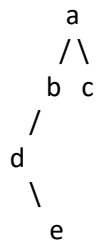
A. ['a', ['b', [], []], ['c', [], ['d', [], []]]]

B. ['a', ['c', [], ['d', ['e', [], []], []]], ['b', [], []]]

C. ['a', ['b', [], []], ['c', [], ['d', ['e', [], []], []]]]
D. ['a', ['b', [], ['d', ['e', [], []], []]], ['c', [], []]]

C is the correct tree representation. Here is the explanation:

The first line generates a binary tree with the node 'a' as the root.
The second line adds a left child 'b' to the left child 'a'. The third line adds a right child from 'c' to 'a'.
The fourth line adds a right child from 'd' to 'c'. Finally, a left child 'e' to 'd' is inserted in the fifth line.
As a result, the tree looks like this:

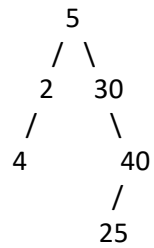


Here, 'a' is the tree', with a left child, 'b' and a right child 'c'. The right child 'c' has a right child 'd', and 'd' has a left child 'e'. This matches the statements given:

```
x = BinaryTree('a')
insert_left(x, 'b')                                  #inserts 'b' as the left child of 'a'
insert_right(x, 'c')                                 #inserts 'c' as the right child of 'a'
insert_right(get_right_child(x), 'd')                #inserts 'd' as the right child of 'c'
insert_left(get_right_child(get_right_child(x)), 'e')    #inserts 'e' as the left child of 'd'
```

13. **Draw a tree showing a correct binary search tree given that the keys were inserted in the following order 5, 30, 2, 40, 25, 4.**

Here is a binary search tree that meets the specified insertion order:



A node's left child has a key less than its parent node, and its right child has a key greater than its parent node in a binary search tree. As a result, when inserting the keys in the specified order, we begin with 5 as the root node. Then, because 2 is less than 5, it is inserted as the left child of 5. Then, 30 is added as the right child of 5,

because 30 is greater than 5. Then, because 40 is greater than 30, it is inserted as the right child of 30. Because 25 is less than 40, it is inserted as the left child of 40. Finally, because 4 is greater than 2, it is inserted as the right child of 2.

References:

Fossati, D., Di Eugenio, B., Brown, C. & Ohlsson, S., 2008. Learning linked lists: Experiments with the iList system. In *Intelligent Tutoring Systems: 9th International Conference, ITS 2008, Montreal, Canada, June 23-27, 2008 Proceedings 9* (pp. 80-89). Springer Berlin Heidelberg.

GeeksforGeeks. (2011). *Linked List vs Array - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/linked-list-vs-array/.

Ravikiran, A. S. (2023). *Types of Linked List in Data Structures | Simplilearn*. [online] Available at: https://www.simplilearn.com/tutorials/data-structure-tutorial/types-of-linked-list.

Krohn, H. (2019). *Linked Lists vs. Arrays*. [online] Medium. Available at: https://towardsdatascience.com/linked-lists-vs-arrays-78746f983267.

GeeksforGeeks. (2011). *Linked List vs Array - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/linked-list-vs-array/.

Karumanchi, N. (2015). *Data Structure and Algorithmic Thinking with Python: Data Structure and Algorithmic Puzzles*. CareerMonk Publications

Thapar, S. (N.D). *Construct Binary Tree From Inorder and Postorder traversal (Recursive method) | Trees*. [online] Available at: https://www.youtube.com/watch?v=rY9ejIY9Osw [Accessed 25 Apr. 2023].

Walker, A. (2020). *B TREE in Data Structure: Search, Insert, Delete Operation Example*. [online] www.guru99.com. Available at: https://www.guru99.com/b-tree-example.html [Accessed 26 Apr. 2023].

Min, W. (2010). Analysis on bubble sort algorithm optimization. In *2010 International forum on information technology and applications* (Vol. 1, pp. 208-211). IEEE.

Programiz (2020). *Bubble Sort Algorithm*. [online] Programiz.com. Available at: https://www.programiz.com/dsa/bubble-sort.

Programiz (N.D.). *QuickSort Algorithm*. [online] Available at: https://www.programiz.com/dsa/quick-sort.

Programiz (N.D.). *Linear Search*. [online] Available at: https://www.programiz.com/dsa/linear-search.