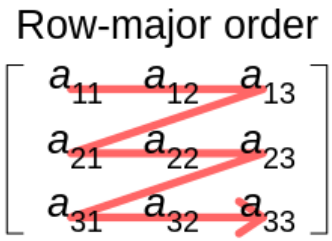
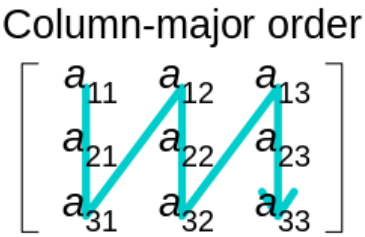
Lab exam 2

Arrays

- Arrays provide constant time access to any element
- There are two ways to index a multidimensional array, row-major and column-major:





- Time complexity of array operations:

Operation	Big O Notation	Description
Add (End)	O(1)	Adding an element at the end of the array.
Add (Middle)	O(n)	Adding an element at the middle of the array.
Add (Begin)	O(n)	Adding an element at the beginning of the array.
Subtract (End)	O(1)	Removing an element from the end of the array.
Subtract (Middle)	O(n)	Removing an element from the middle of the array.
Subtract (Begin)	O(n)	Removing an element from the beginning of the array.
Search (End)	O(n)	Searching for an element at the end of the array.
Search (Middle)	O(n)	Searching for an element in the middle of the array.
Search (Begin)	O(n)	Searching for an element at the beginning of the array.

Lists

- A list is a collection of elements where each element has a successor element.

- In a singly linked list, each element points to its successor, forming a chain. In a doubly linked list, each element points both to its successor and its predecessor, forming a bidirectional chain.

Trees

- Variants of trees:
  - binary trees - each node has at most two children
  - binary search trees - same as binary but now the left child holds a value less than the parent, and the right child holds a value greater than the parent.
  - balanced trees - a tree where the depth of any two leaves (length of a path from a node to the root) differs by at most one.
- Three variants of binary tree traversal:
  - Preorder: Root → Left → Right
  - Inorder: Left → Root → Right
  - Postorder: Left → Right → Root

Tries

- Also known as prefix trees
- It is a tree like data structure, where each edge is a character in a string, keys stored in that data structure are created by traversing this tree from the root, and each node stores a value associated with this key.

AVL Tree

- Rebalancing an AVL tree is performed after every deletion/insertion. The procedure is:
  - Update the balance factor of every note (height of left subtree - height of the right subtree)

    - For every node (from bottom to the top) that has a balance factor > 1 or < -1 :
        - If balance factor > 1 and `newNode` < `leftChild` then do right rotation
        - if balance factor > 1 and `newNode` ≥ `leftChild` then do left-right rotation
        - if balance factor < -1 and `newNode` > `rightChild` then do left rotation
        - if balance factor < -1 and `newNode` ≤ `rightChild` then do right-left rotation
    - After every iteration update the balance factors and repeat until the tree is balanced.
- Deleting a node is performed by swapping it's value with the node that has the smallest value in the node's right subtree (smallest node greater than the node being deleted).

Graphs

- Graph representations:
    - Adjacency matrix - a matrix where if there is a 1 at position `(i,j)` that means there is an edge from i to j. For undirected graphs this matrix is symmetric.
    - Incidence matrix - a matrix where each row represents a vertex and each column represents an edge. The value at position `(i,j)` indicates that vertex i is incident to edge j.
    - Edge list - a simple list of tuples where each tuple represents an edge in the graph.
    - Adjacency/Successor list - a dictionary where keys are vertices and values are lists of neighbouring vertices.

Tree Traversal

- Depth first search

```
def dfs(a, v):
    visited = []
    def traverse(a, v):
        visited.append(v)
        for j,k in enumerate(a[v]):
            if k == 0: continue
            if j not in visited: traverse(a, j)
    traverse(a, v)
    print(visited)
```

- Breadth first search

```
def bfs(a, v):
    visited = [v]
    temp = [j for j,k in enumerate(a[v]) if k == 1]
    while len(temp) > 0:
        visited.append(temp[0])
        for j,k in enumerate(a[temp[0]]):
            if k == 1 and j not in visited:
                temp.append(j)
        temp.pop(0)
    print(visited)
```

Two other binary search tree traversal methods:

- Binary tree level order traversal - visits all notes level by level from left to right.
- Spiral order traversal - start from the root then alternate between left-to-right and right-to-left traversal for each level. (Implemented by maintaining two stacks until both are empty)