Theory

1.

Array

- An array's size is fixed as prescribed by it's declaration and a one block of memory is allocated according to this size when it is created.
- Elements exist in serial positions in this memory block and can be accessed by their indexes.
- Elements are all of the same size.
- Looking up an element by its index is fast due to the nature of the memory arrangement.
- Removing or deleting elements in an array requires that other elements behind the position of that element has to be shifted by one position. This places a burden on processing time.

List

- A linked list is dynamic in nature and memory is allocated to it when needed.
- An element or node consists of two parts, the value stored and a pointer to memory address of the next element in the list.
- The values in the list can be of any type or size.
- Deleting or inserting elements requires only that requires that the address pointers of the elements needs to be updated, which pointers to be updated determined by whether the insertion/deletion is at the head, middle or end of the list.
- Looking up elements takes longer than looking up elements in an array since the list has to be traversed from pointer to pointer until the desired value is found.

When to use which and when not:

Array:
- Use when frequent lookups are needed and, bonus if the position of the value looked up is known.
- Do not use if frequent insertions or deletions will happen.

Linked list:
- Use when it is not known how many values will be added or deleted to the collection, or when it is known that addition and deletion of values will happen often.
- Do not use when the collection is likely to remain static and frequent lookups of values are required.

2.

Dictionary

A dictionary is a collection of key-value pairs and is a higher level language implementation of a hash table. Insertion, deletion and access time on average is O(1) but could degrade to O(n) where n = (number of key-value pairs).
Use when data needs to be retrieved easily without having to know the underlying hash functions or specific indices of values.

Stack

A stack is a collection of values ordered in the sequence in which the values were added (popped) onto the stack.
A stack can hold only a predefined amount of values and this needs to be checked before pushing or popping values onto or from the stack.
Values can be pushed (added) onto the list or popped (removed) from the list.
The last value popped onto the list will be the same value that will be removed with a following pop command.
Use in cases where parenthesis/brackets/braces pairs needs to match in an input string, reverse a word, computations needs to be done in reverse order, reclaim memory after the last instruction on the stack was completed.

Queue

A queue is a list of values queued in the order in which it was added (enqueued).
The first value in the queue (added to the queue with enqueue) will also be the first value that will be removed with a dequeue command.
Use when the collection of values need to be acted on in the sequence in which they were added. (For instance a group chat where the original sender of a message should be the first person to see his typed message appear in the chat after pressing enter)

3. Hash Table

a) Use the remainder of the division of the input key by size of the array and use the result to store the value in the bucket with the corresponding index.
Array size = 10

10, “first value” → (10 % 10) (store/find value at index 0)

b) Convert the input to the total ascii value of the characters in the string and apply same formula as above.

c)
Linked list buckets: turn the bucket where the collision occurs into a linked list where the first value also contains the address of the next value in the same bucket and so on. This implies a linear search in this bucket.
Linear probing: use hash to compute the index, if a bucket is occupied, place it in the next available empty bucket to the right (the first empty bucket with a higher index). This can become inefficient with large arrays since a linear search for the next empty bucket will increase in time.
Double hashing: Jump ahead (n times) from the position of the occupied/collision bucket until an empty bucket is found.

4. Skip List

It is a dynamic, randomized data structure of linked lists that uses a probability distribution algorithm for its search function with the efficiency comparable to balanced trees but with lower overhead.

It consists of an ordered link list where each node has one of more forward pointers (express lanes) which allows searches to skip ahead to the relevant sections of the list where the value searched resides.

The number of levels will depend on the chosen chance to advance a node during insertion.

This improves the O(n) complexity of normal linked lists to O(log n) (worst-case) where n = number of lanes.

How it works:

Search:

Normal linked list: 30 -> 40 -> 50 -> 60

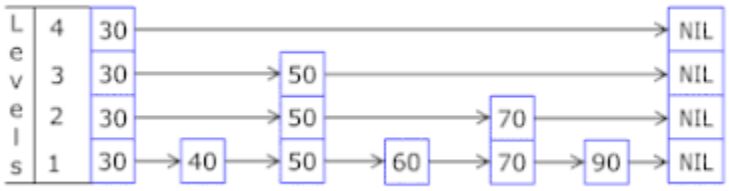
Skip list: 30 -> 50 -> 60

To search for an element 60, begin at the topmost level of the header.

Go through the list in this level moving to the right (forward or next) until the node with the largest element that is smaller than 60 is found (50)

Then go to the level below and, beginning from the node found in the level above, search again for the node with the largest element that is smaller than 60.

When such a node is found, go down again and repeat down to the bottom level. Here 50 is still the largest node that is smaller than 60 and 60 < 70.



Insert:

Do a search first to check that the node does not exist.

Insertion consists in deciding the height of the new node, where a new level is created at random (maybe there will be a new level, maybe not)

For each of the levels up to this height, insert this new node after the node with the largest element that is smaller than the new node's element.

Deletion:

Find the positions of the node to be deleted in all the levels it resides along with the positions of all the nodes with the largest elements that are less than the element of the node to be updated (nodes left of element in each level)

Set nodeLeft.next = nodeToDele.next in each level.

Update:

Search for the node in all levels, update node value/element.

Advantages:

- Has the flexibility of a linked list but with improved search speed.
- Useful when concurrent access to data is needed.
- On insertion only the adjacent nodes are affected.
- Automatic sorting of inserted elements.
- If one part of the list is modified, data can still be accessed in other parts.

- Don’t need to know the total memory needed beforehand.

Practical

Section A

Data Structures 1

https://github.com/isabellavs/avochoc/blob/main/palindrome.py

Data Structures 2

https://github.com/isabellavs/avochoc/blob/main/contactsList.py

Data Structures 3

https://github.com/isabellavs/avochoc/blob/main/movieList.py

Section B

Algorithms 0

https://github.com/isabellavs/avochoc/blob/main/calcAvg.py

Algorithms 1

https://github.com/isabellavs/avochoc/blob/main/sortNum.py

Section C

Research

Level 1

https://github.com/isabellavs/avochoc/blob/main/calcSum.py

Level 2 (Choose one)

Section D

Logic

Easy (Choose one)

Medium (Choose one)
Hard (Choose 1)

DB Design