**Data Structures & Algorithms**

**Algorithms**

* Algorithms – steps needed to complete a task
* A problem can be solved using various algorithms.
* An algorithm can have many implementations

**Big O Notation / Time Complexity:**

Time Complexity – The amount of time taken by an algorithm to run.

Memory Complexity – The amount of memory taken by an algorithm to run. (not as important now because memory is cheap).

Table

Description automatically generated

Listed from best to worst.

Chart, line chart

Description automatically generated

Constant O(1) : As the number of items increases the number of steps remain the same (constant). This is the best-case scenario.

As you go down the table the number of steps drastically increase as the number of items increase.

**Arrays**

* Size of an array is static
* Contiguous block in memory
  + Memory is allocated on declaration of the size of an array.
  + Every element occupies the same amount of space in memory.
  + In the case of non-primitive types (objects) the array contains references to those types of same size.

Benefits:

* Useful when you know the indices of the element you want to retrieve
  + O(1) constant time complexity
* Memory Efficient

Cons:

* When indices of element is unknown it becomes:
  + O(n) linear time complexity

Table

Description automatically generated

**Unstable vs Stable Algorithms:**

Table, calendar

Description automatically generated

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Graphical user interface, application, table

Description automatically generated

Stable Algorithms are usually preferred in some cases such as sorting objects.

**Sorting Algorithms**

**Bubble Sort Theory**

* Performance degrades quickly as the items needed to sort grows.
* In-place algorithm (sorted items occupy the same storage as the original items)
* O(n2) – quadratic time complexity
* Stable algorithm
* Grows right to left
* Traversing through the unsorted partition
* Uses two variables
  + Unsorted partition index
  + Traverse index

Graphical user interface, application

Description automatically generated

Diagram, table

Description automatically generated with medium confidence

* In general, the time complexity can be determined by looking at the amount of “for” loops the algorithm has as they correspond to “n”.
  + 2 for loops (quadratic algorithms)

**Selection Sort Theory**

* In-place algorithm
* O(n2) – quadratic time complexity
* Less swapping than bubble sort
* Unstable algorithm
* Grows right to left
* Traversing through the unsorted partition
* Uses 3 variables
  + Unsorted partition index
  + Traverse index
  + Largest value index

Graphical user interface, application

Description automatically generated

**Insertion Sort (Theory)**

* In place algorithm
* O(n2) – quadratic time complexity
* Stable
* Grows from left to right
* Work with the sorted partition
* Uses 3 variables
  + Unsorted partition index
  + Traverse index
  + New Element – value we want to insert

Graphical user interface, application

Description automatically generated

**Shell Sort (Theory)**

* In place algorithm
* Time complexity is dependent on gap
  + Worst case: O(n2)
* Unstable
* Variation of Insertion Sort
* Shell Sort starts out using a larger gap value
  + Insertion sort chooses which element to insert using a gap of 1
  + Goal is to reduce amount of shifting required
* As the algorithm runs the gap is reduced
  + Last gap value is always 1, which is ultimately an insertion short

Table

Description automatically generated

* Gap is calculated using (3k-1) / 2
* k is set based on the length of the array
* We want the gap to be as close as possible to the length of the array, without being greater than the length.

Graphical user interface, application

Description automatically generated with medium confidence

**Merge Sort (Theory)**

* Not in place
* O (n log n)
* Stable
* Divide and conquer algorithm (recursion)
* Two phases:
  + Splitting
  + Merging
* Splitting leads to faster sorting during the merging phase
* Splitting is logical, not creating new arrays

**Splitting Phase**

* Start with an unsorted array
* Divide the array into 2 arrays, which are also unsorted
  + First array – left array
  + Second array – right array
* Keeping splitting the subarrays until they become arrays of one element
* The one element arrays are considered sorted

Diagram

Description automatically generated

**Merging Phase**

* Merge every left/right pair of sibling arrays into a sorted array
* After the first merge, we’ll have a bunch of 2-element sorted arrays
* Then merge those arrays (left/right) siblings to end up with a bunch of 4-element sorted arrays
* Repeat until you have a single sorted array

Not in-place. (uses temporary arrays).

Diagram

Description automatically generated

**Quick Sort (Theory)**

* In place
* O (n log n)
* Non-stable
* Divide and conquer algorithm (recursion)
* Three steps:
  + Identify pivot
  + Partition
  + Recursive Call

**Lists**

* Lists are an abstract data type.
* Classes that implement the list interface are an order collection (a sequence).
* Common methods:

|  |  |  |
| --- | --- | --- |
| Modifier and Type | Method | Description |
| Boolean | add(E e) | Appends the specified element to the end of the list |
| Boolean | contains(Object o) | Returns true if this list contains the specified element |
| E get | (int index) | Return the element at the specified position of the list |
| Int | indexOf(Object o) | Returns index of the first occurrence of the specified element in the list or -1 if does not contain. |
| Boolean | isEmpty() | Returns true if list has no elements |
| Boolean | remove(Object o) | Removes the first occurrence of the specified element in the list, if it is present |
| Int | size() | Returns the number of elements in the list |
| Object[] | toArray() | Returns an array containing all of the elements in this list in proper sequence (from first to last element) |

Lists Documentation: <https://docs.oracle.com/javase/9/docs/api/java/util/List.html>

**Abstract Data Type**

* Doesn’t dictate how the data is organized
* Dictates the operations you can perform
* Concrete data structure is usually a concrete class
* Abstract data type is usually an interface
* Lists, Stacks, Queues are all abstract data types
  + They display functionality. However, only a class that implements these interfaces can show implementation of these abstract data types.
  + E.g., ArrayList class implements the List interfaces and Array Data Structure.
    - The implementation of the List abstract data type in the ArrayList class is to add Lists functionality to Array Data Structures.

**Array Lists**

* A resizable array implementation of the List interface
* The data in the list is being stored in an array (called the backing array)
* If the backing array you want to add items to isn’t large enough, then it will negatively affect the performance as the implementation would have to resize the backing array to accommodate the new items.
* Same with removing, as it would have to shift items to remove any empty space.
* If add/removing, then it’s best to have an idea of how many items are going to be on the list.
* This is done with in the constructor that takes in the initial capacity shown below:
  + ArrayList(int initialCapacity) – constructs an empty list with the specified initial capacity
* Difference between Size vs. Capacity
  + Capacity is the maximum number of items that the list can hold before it’s going to have to be resized
  + Size is the actual number of items in the list
* Pros:
  + Good for iterating items through the list
* Cons:
  + Bad for inserting items into the list in any position other than the end
  + Not good for deletion/removal
  + Not good for accessing an item without the index

**Vectors**

* Vector is a thread safe ArrayList class
* Synchronized
* If a thread-safe implementation is not needed, then ArrayList is a better option
* Can be assigned to the interface List rather than using the specific implementation “Vector”, This is good practice as it allows switching between other implementations of that interface

**Linked List**

* Pros
  + Does not need to be resized
  + Can be O(1) complexity if only need to add items to the front of the list
* Cons
  + Bounded by memory
  + Not good for random access

**Singly Linked List**

* Each Item in the list is called a node
* The first item in the list is the head of the list
* In java, you wouldn’t implement a linked list class yourself, there’s a linked list class

Chart

Description automatically generated

Adding Bill to Head

* When it comes to inserting in a LinkedList you always want to insert at the front of the list.
* Inserting in a Singly LinkedList
  + Create a new node “Bill”
  + Assign “Jane” to the next field
  + Assign head to “Bill”
  + Will be O(1) complexity
* Deleting in a Singly LinkedList
  + Assign “Bill” to temporary variable “removedNode”
  + Assign “head” to “Jane”
  + Return “removedNode”
  + Will be O(1) complexity
  + If you want to do cleanup you’d set Bill next field to null

**Doubly Linked List**

* Each item in the list has a next and previous fields
* Has a head and tail
* Can traverse from front to back/back to front

Chart

Description automatically generated

Adding Bill to head

* Inserting at the head of a Doubly LinkedList
  + Create a new node “Bill”
  + Assign “Jane” to Bill’s next field
  + Assign whatever Jane is pointing to as previous to Bill’s previous field
  + Assign “Bill” to Jane’s previous field
  + Assign head to “Bill”
  + O(1) complexity

Chart

Description automatically generated

Adding Bill to tail

* Inserting at the tail of a Doubly LinkedList
  + Create a new node “Bill”
  + Assign tail’s next field to Bill’s next field
  + Assign tail to Bill’s previous field
  + Assign tail’s next field to “Bill”
  + Assign tail to “Bill”
  + O(1) complexity

Chart

Description automatically generated

Removing Jane from head

* Deleting from head of a Doubly LinkedList
  + Assign “Jane” to “removedNode”
  + Assign John’s previous field to Jane’s previous field
  + Assign head to Jane’s next field
  + Return “removedNode” from the method
  + If you want to do cleanup you’d set Jane’s next field to null
  + O(1) complexity

Chart

Description automatically generated

Removing Bill from tail

* Deleting from tail of a Doubly LinkedList
  + Assign “Bill” to “removedNode”
  + Assign Mike’s next field to Bill’s next field
  + Assign tail to Bill’s previous field
  + Return “removedNode” from the method
  + If you want to do cleanup you’d set Bill’s next field to null
  + O(1) complexity
* Inserting a node A between B and C
  + Assign A’s next field to B’s next field
  + Assign A’s previous field to C’ s previous field
  + Assign B’s next field to A
  + Assign C’s previous field to A
  + O(n) complexity
* Removing a node A between B and C
  + Assign A to “removedNode”
  + Assign C’s previous field to A’s previous field
  + Assign B’s next field to A’s next field
  + Return A from the method
  + If you want to do cleanup you’d set A’s next to field to null
  + O(n) complexity