

Question One:

a) Running time:

$$RT = n^2 + n \log n + 3n + 2.5.$$

Within this algorithm

2.5 is $O(1)$ so is the lowest complexity.

~~n^2 is quadratic complexity~~
 ~~$n \log n$ is the next most complex~~
 $n \log n$ is the next most complex
 n^2 is quadratic complexity and the next most complex
 $3n$ is exponential complexity, this represents the
highest complexity in our running time at $O(2^n)$
Therefore the overall worst case complexity of this
algorithm is $O(2^n)$

b) $O(1)$: This is constant time.

$O(1) =$ $x = 1$
 $\text{print}(x)$

For every element added
the run time will always
be the same

$O(n) =$ this is linear time

Array = []

for i in array:
print i

For every additional
element in the array
the time increases by the same amount

$O(n^2) =$ this is quadratic time.

Array = []

~~def~~ ~~for~~ ~~in~~ PrintArray(x)
for i in array
if i == x
print i

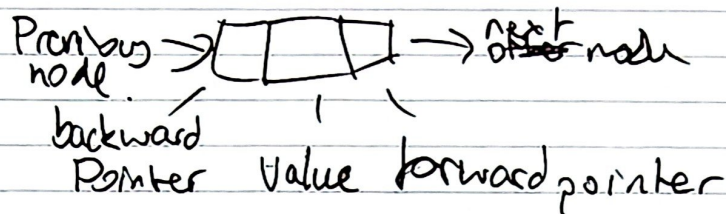
For every additional element in
an array we must loop and
check in an if statement

Question one!

c)

Dynamic memory allocation allows us to allocate memory during run time.

a doubly linked list has nodes that have a forward + backward pointer



To add an item to a double linked list we must allocate the ~~node~~ preceeding the node at which we wish to place our new node, a pointer to the new node. Allocate our new node a pointer to the node it is.

Question two:

a) Hashing is a method used to determine the location of an element in a table. In order to carry out hashing we develop a hashing algorithm, such as;

$$h(k) = k \bmod m$$

where k represents the value of our element and m calculates the location in our hash table at which the value should be located. This allows us to quickly find values in a table, in constant time ($O(1)$) as we simply must pass our value through the algorithm.

b) our hashing algorithm/hash function is

$$h(k) = k \bmod m$$

$$m = 7 \text{ (table size)}$$

meaning that for a key of 17 we must divide it by 7 and calculate the remainder.

$$17 / 7 = 2 \text{ remainder } 3$$

Table after
insertion

0	
1	
2	
3	17
4	
5	
6	

We then wish to add 38 to our table

so:

$$38 / 7 = 5 \text{ remainder } 3$$

so if we attempt to place 38 in a static array, the values of 17 and 38 collide. However, when using separate chaining values are stored in a linked list structure and we simply add 38 as a node connected to the table element 3. as shown below

0	
1	
2	
3	→ 17 → 38
4	
5	
6	

Question Two:

C) Separate chaining is an effective method used in hashing in order to prevent issues that arise from collisions.

By storing values in linked lists, attached to each element in the hash table, we are able to always add a value to its corresponding element, derived from a hash function.

This method has distinct advantages over other approaches such as open addressing;

- Linked lists are dynamically allocated, therefore this method allows a hash table to grow to any size with little overhead.
- Collisions largely become a non-issue.
- Unlike open addressing there is no need to account for clustering, removing the need for ~~complicated~~ complex probing such as quadratic probing.
- Work well in high load density hash tables.

Despite advantages over other methods, there are some disadvantages;

- Dynamic structure means that search operations are complicated vs arrays.
- Caching function of static arrays is far better.
- When lists grow, search functions can become far slower than static arrays. In open addressing with static array search complexity remaining at $O(1)$ and ~~that~~ separate chaining being $O(n)$.
- Stroustrup in his 2015 lecture on C++ best practices states that ~~linked~~ lists should be avoided, particularly when large data sets are involved due to search complexity, in addition to high memory overhead due to the need for pointers at each value.

Question 4a continued:

The value after 74 is 75, this should be the parent node of 74.

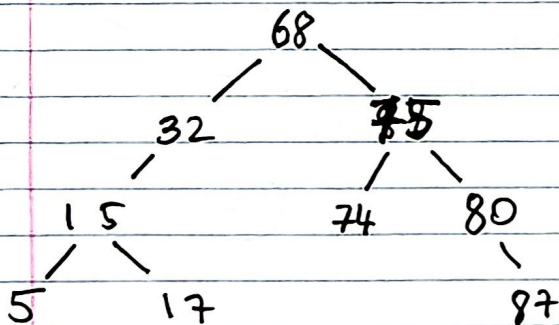
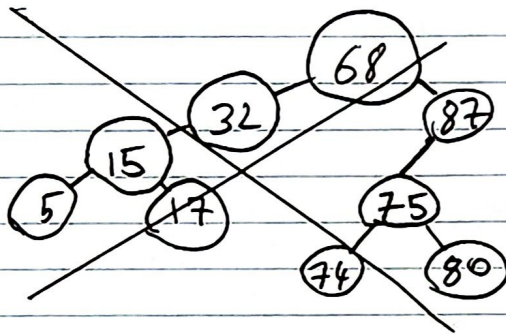
The next value sequentially in our ordered list is 80 so it should be a right child of 75.

Finally 87 is greater than 75, it should in order to create a balanced tree have 75 as a left child. But 87 can be a right child of 80. This is our final node so therefore is linked back child of 80.
~~to our root~~

Our final tree is

To check this an in order search result is:

5, 15, 17, 32, 68, 74, 75, 80, 87



Question four b:

In order traversal: 1, 14, 15, 17, 25, 34, 36, 63, 81, 87

Pre order traversal: 36, 25, 14, 17, 15, 34, 81, 63, 87

Post order traversal: 1, 15, 17, 14, 34, 25, 63, 87, 81, 36.

An inorder traversal ~~traverses~~ is: Left Root Right

A pre-order traversal is: Root Left Right

A postorder traversal is: Left Right Root

Question four C:

In order to delete a value from a binary tree we must first take the nature of the node we wish to delete into consideration. If the node is a leaf node it can simply be deleted, however if it is a parent such as 81, the process is more complicated. We must delete the node in a way that does not disrupt the function of the tree.

We must ~~first~~ ensure that when an inorder traversal is carried out, the values are ordered in weight. In order to do this we can use an algorithm such as the one below.

~~tree(x)~~
delete(x):

for i in tree:
if ~~i~~ = x:

delete(x):

for child of x:
if child < x:
x = child

By locating the smallest valued child of 81 we can then assign that value to the node that contains 81 and remove the node containing its smallest child (63)