

Level 5 time restricted assessment specification

Part a

1. Compare and contrast the benefits and drawbacks of using arrays and linked lists.

Arrays and linked lists

An array is a basic data structure that stores data elements in continuous memory locations. An array can only store data of the same data type e.g. integer or float. An array will always be defined with a fixed length when initialised: any elements which are added which exceed the specified size, will cause an array out of bounds exception (this may be slightly different depending on language and compiler) and will cause the program to crash, providing the exception is not handled using a catch clause.

A linked list is conceptually similar to an array, in that it is used to store similar types of data, although link lists are not restricted to the same data type as arrays. Unlike arrays, linked lists do not store the data elements in successive memory locations. Rather, it uses a series of nodes which are linked via pointers.

Search times

One of the main benefits an array has over a linked list is faster search times. This is because any array element can be accessed by using the offset to the location of the first element, as such, arrays have a constant search time, which can be expressed in terms of the Big O notation: $O(1)$. Linked lists, in contrast to a standard array, have significantly slower search times. Indeed, unlike arrays, linked list elements cannot be searched by index, but rather uses iteration. This means that a pointer to the first node would need to iterate along each element sequentially to reach the specified element in the list. This inevitably has consequences each time an extra element is added to the list, as the iterator will have an extra element to iterate over when searching for the specified node. Given this, a linked list can be said to have a linear search time which can be expressed in the big O notation as $O(n)$.

Memory usage

A static array has the potential to waste much more memory than a linked list. This is because it is not always possible to determine how much memory needs to be allocated during initialisation, for example: for an array that stores X amount user inputs, if the array is declared to have 1000 elements and the user only ever inputs 2 elements, the remaining allocated space is

wasted. However, it is possible to address this by introducing dynamic arrays or data elements that lend from arrays such as vectors in C++. Linked lists are much more memory efficient, at least from the perspective of memory allocation, as the size is not pre-defined at the point of initialisation. This effectively allows nodes to be removed and added during the program's runtime. An advantage an array does have over linked lists in terms of memory usage, is that it requires less memory per element, as a linked list needs to allocate additional resource to store the pointers.

Insertion and delete time

An advantage that linked lists have over arrays is faster insertion and deletion of data. Nodes can be added or deleted to or from the start or the end of the list, using constant time ($O(1)$). Adding nodes to the middle of the list is slightly slower due to the iterations required to reach the specified node. This is done using a linear time ($O(n)$). In contrast to linked lists, arrays have an insertion time of n ($O(n)$) or $O(1)$ if deleting or inserting from the last element of the array. The slower timings can be understood when considering that every element after the specified index must be shifted to make space for the insertion. Moreover, the elements will again need to be shifted in the same way for a delete.

2. We wish to record thousands of new names being inserted every day. Would it be preferable to use an array or a linked list? Provide reasons for your choice.

A linked list would offer some benefits over an array in this situation. For example a static array would require a set number of elements to be defined at initialisation – and there is currently no information on how many records will be input in total, so it is impossible to determine how big the array needs to be, whereas a linked list can shrink or grow each time a record is added or deleted: this has implications for memory management, as any array elements which are declared and not used are a wasted resource, unlike in a linked list where there will be no empty nodes. This will offset the additional cost of storing pointers in a list compared to the lower cost elements in an array. Another main benefit of using a linked list over an array is faster data insertion and deletion. This is important as we will be entering thousands of records each day and as discussed in question A: linked lists have a constant ($O(1)$) insert and delete time (from start or end of list). There is also no requirement to regularly retrieve or lookup data (the question only mentions enter) – as such the benefit of the constant search times ($O(1)$) provided by an array are not likely to be utilised.

Assume we wish to use a hybrid approach where we create an array of linked lists as shown in figure 1 below.

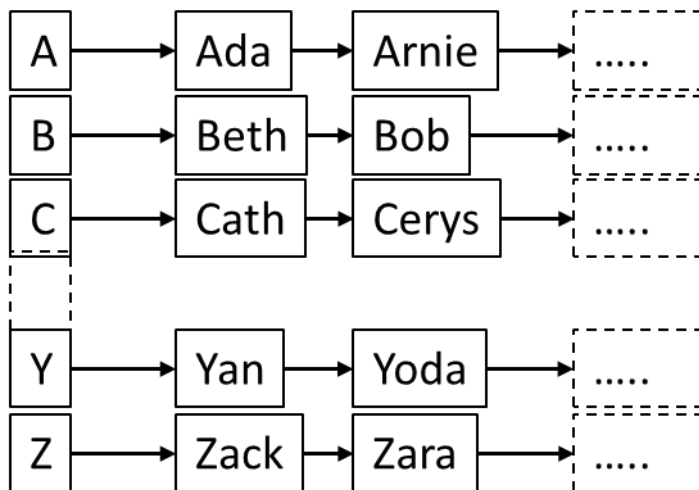


Figure 1 Hybrid Data Structure

Evaluate the probable performance of this data structure. You do not need to provide Big O calculations for this analysis. If a significant percentage of the names are found to start with just 5 letters, what problems might this cause? For example, names beginning with H may be very popular this year.

The performance of the structure for this scenario is suitable, as it segments lists by alphabetical order by storing them in individual array elements. By using this data structure, we can reduce the size of an individual linked list into a more manageable chunks – this will improve the look up speed over storing all records in just the one standard list as the iterator will have less nodes to iterate over to find the correct one. It also solves the issue with the static array only storing a specified number of elements, as we know that we only need 26 (length of the alphabet) – this helps when controlling memory allocation as no elements in the array are being wasted. Moreover, as array elements are indexed, we can perform fast lookups to identify each list using the constant lookup time or big O notation ($O(1)$).

The problem that would arise from having a having a significant volume of data for just the one type of name, for example starting with H, is that all searches against names on that particular list, will return slower than the records from the trees that have fewer records, this is because a list uses an iterator which runs from the head of the list and the more nodes there are, the longer the search result will take to return, as linked lists have a linear search time: expressed as a Big O notation of n ($O(n)$).

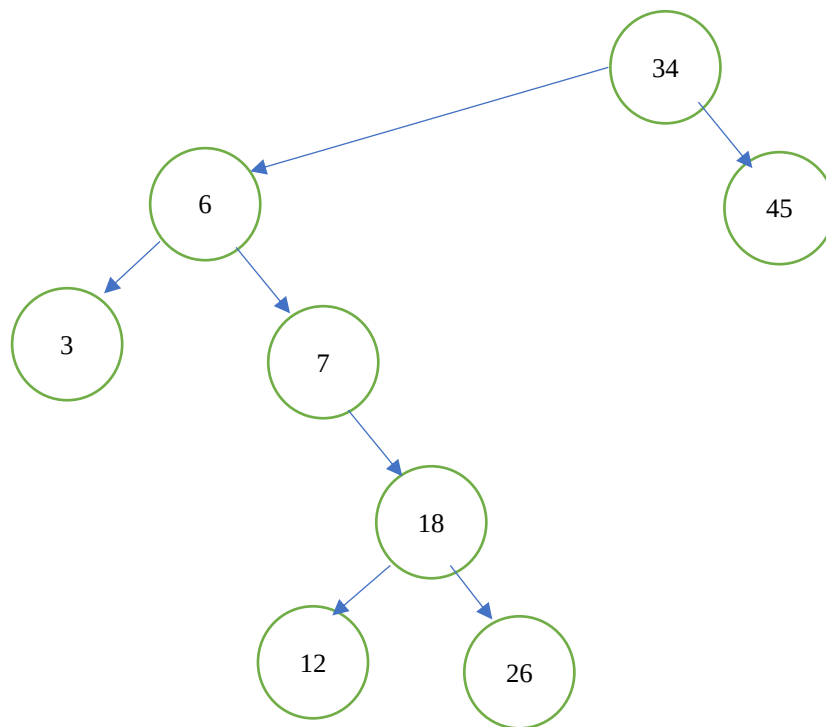
3. Demonstrate how a binary search tree is constructed for the values 34, 6, 7, 18, 45, 12, 3, 26 and explain the need to maintain a balanced binary tree.

To understand how a binary search tree is constructed, is important to consider the four following rules:

1. Nodes to the left of the subtree must contain only nodes with keys which are lesser than the nodes key.

2. Nodes to the right of the subtree must contain only nodes with keys greater than the nodes key.
3. Both subtrees are binary search trees.
4. There can be no identical nodes

Given this information a binary search tree for the above values would look like this:



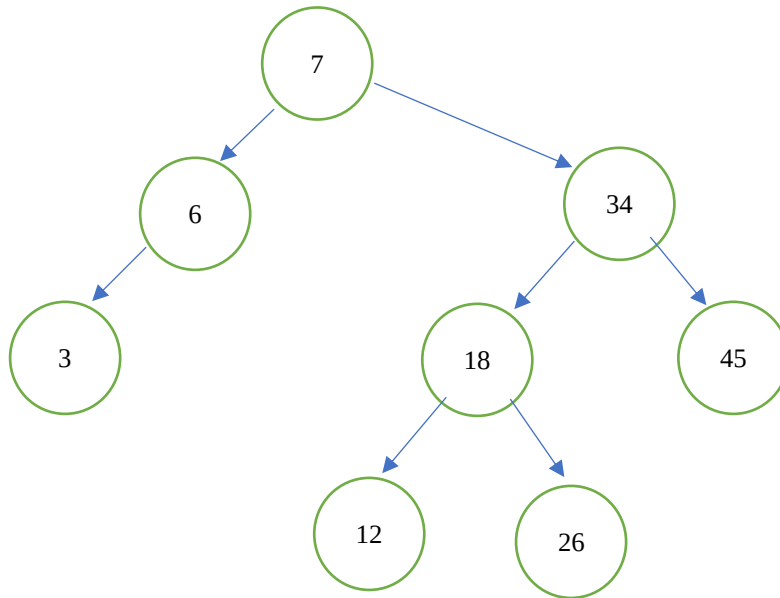
While the above tree is correct, it is not particularly efficient as the left subtree is evidently larger than the right subtree. This would result in good performance if running a search to the right of the root node as it only has the one level to search, but slower performance if the search were to the left of the root node as it has three extra levels to search. This is clearly inefficient and a waste of processing power and time, something which will compound if the above tree continues to grow in its current form. There is a solution which involves balancing the tree which should provide at least $O(\log n)$ in the worst case scenario. An AVL (Adelson-Velsky and Landis) tree is an example of a height balanced tree which can be used to help us achieve this. In an AVL binary search tree, the differences between the height on the right and left subtrees cannot be more than one for all the nodes. This is achieved by considering the following formula:

$$BF = \text{Height}(\text{LeftSubTree}(X)) - \text{Height}(\text{RightSubTree}(X))$$

The balance factor for the binary tree above can be calculated as: $BFA(A) = 4 - 1 = 3$.

In an AVL tree, there must be a branch factor of either (1, 0 or -1), as demonstrated the above tree would not meet this criteria.

By applying the logic required for an AVB balanced search tree, the above tree can be rebalanced as shown below:



The above tree now meets the criteria for faster search operations $O(\log n)$ with a branch factor of: $BFA(A) = 2-3=1$.

Part B

1) Dijkstra's Algorithm: Directed graph diagrams

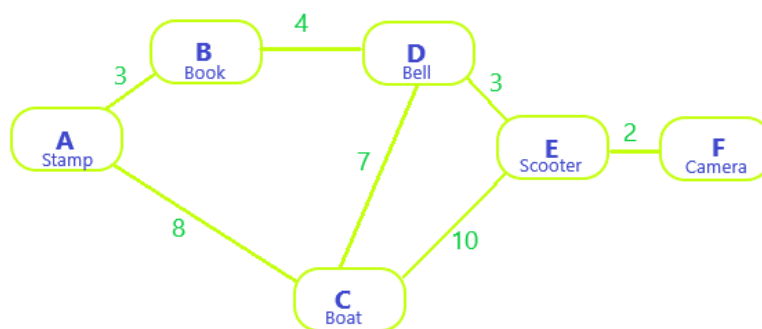
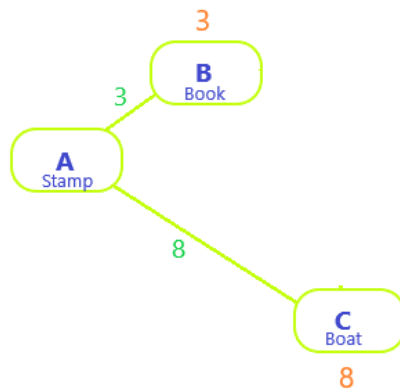
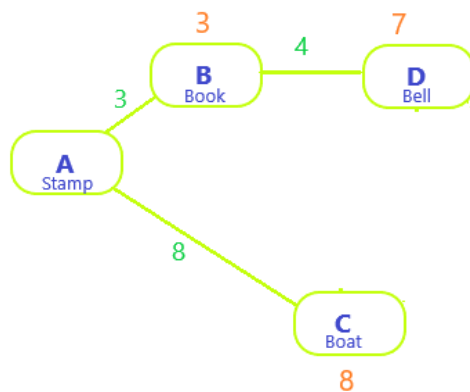


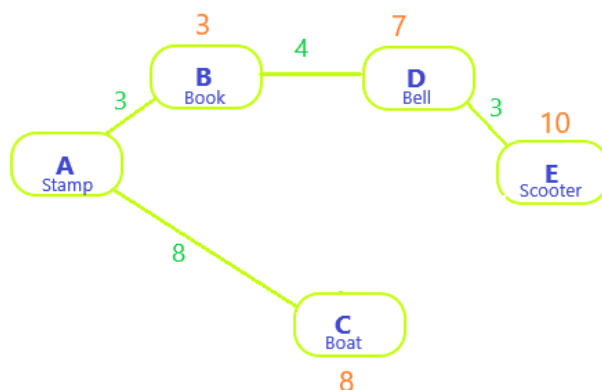
Diagram to show all “trade” arcs and vertices.



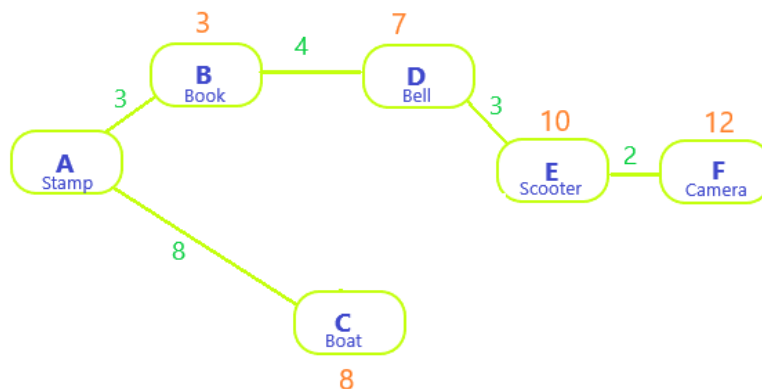
Step 1: Vertex A is the source vertex with a value of 0. The lowest value vertex is selected — Vertex B is given a value of three. The adjacent vertex (C) is updated to give a value of eight.



Step 2: The minimum value vertex is selected — Vertex D is given a value of seven.



Step three: The minimum value vertex is selected — Vertex E is given a value of ten.



Step four: The minimum value vertex is selected — Vertex F is given a value of 12.

	Vertex	A	B	C	D	E	F
Start	Parent	null	null	null	null	null	null
	Cost	0	∞	∞	∞	∞	∞
1	Parent	null	A	A	null	null	null
	Cost	0	3	8	∞	∞	∞
2	Parent	null	A	A	B	null	null
	Cost	0	3	8	7	∞	∞
3	Parent	null	A	A	B	D	null
	Cost	0	3	8	7	10	∞
4	Parent	null	A	A	B	D	null
	Cost	0	3	8	7	10	∞
5	Parent	null	A	A	B	D	E
	Cost	0	3	8	7	10	12

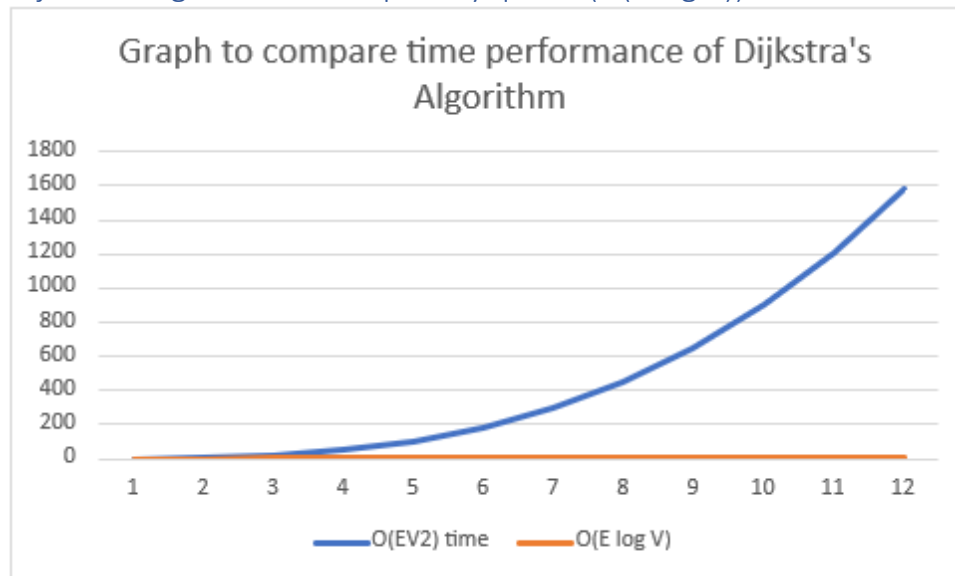
Table to show Vertex values being processed.

2) Table to compare the time performance of Dijkstra's Algorithm $O(EV^2)$ against using Dijkstra's Algorithm with a priority queue ($O(E \log V)$)

Vertices (V)	Edges (V-1)	$O(EV^2)$	$O(E \log V)$
1	0	0	0
2	1	4	0.30103
3	2	18	0.95424251
4	3	48	1.80617997
5	4	100	2.79588002
6	5	180	3.89075625
7	6	294	5.07058824
8	7	448	6.32162991
9	8	648	7.63394008
10	9	900	9

11	10	1210	10.4139269
12	11	1584	11.8709937

Graph to compare the time performance of Dijkstra's Algorithm $O(EV^2)$ against using Dijkstra's Algorithm with a priority queue ($O(E \log V)$)



Part C

Minimax algorithm

The minimax algorithm is a backtracking algorithm that is used in two-player board games, such as tic-tac-toe and connect four, to determine the optimum move for a player based on the assumption that the opponent player plays optimally too. The algorithm searches the game tree using recursion to make a decision for the next move based on the current game state.

The game tree root node represents the current state of the game board, and the branches to subsequent nodes represent the possible game board states for each possible move by the player. The level below this represents all possible outcome game board states for the opponent players next move, and so on. The game tree branches out in this manner, mapping each possible move in turn, and ultimately, each possible outcome until one player wins, i.e., the terminal nodes.

In order to make a decision about which move is optimal for the computer, each possible move is scored for "utility". Utility is a weighting given to each node (i.e. possible game state) by the programmer based on factors such as location of opponent pieces, potential for winning positions by either player, and the number of blank spaces remaining on the board. The first player is the *maximiser*, which attempts to attain the highest possible score, while the other is the *minimiser*, who attempts to attain the lowest possible score. The algorithm calculates its next best move by determining the utility score for each node. It then calculates what the next likely move taken by the opponent will be, assuming they play optimally, and continues this way until it reaches the terminal node. The node that returns the highest utility score, when considering the moves taken by the opponent, is the optimal one and is selected. This is a depth-first search, where the algorithm searched vertically down the tree,

before backtracking and cycling recursively until all nodes of the entire tree has had a utility value assigned and a decision is made.