**Linked-Lists**

**4.0 Nodes Introduction**

**Nodes**

* Nodes: are what linked lists comprised off. Nodes are the building blocks of Singly Linked Lists. They are a data structure which holds a few key pieces of information.
* A node is a locational object, and it has two desirable features:
* 1. It stores the address of elements of an array, or RAM or other memory within our computer. Therefore, when you call on a node it can call on these elements, because it stores their addresses. Each node can store the address of the data from different locations within the memory. Then, the linked lists can connect all these notes, meaning it connecting data from various locations within the memory of the computer
* 2. nodes can be connected, when connected they make up a linked list. Therefore, they are able to call on different combinations of elements from the computer’s memory, because each node has some sort of elements; addresses

**4.1 Linked Lists**

**General Info**

* Linked Lists are a core data structure that can link data together from multiple locations. They are used by queues, trees, data storage, and even web searches when you arrows that you can use to switch between different pages, and many more applications
* Each node can store the address of the data from different locations within the memory. Then, the linked lists can connect all these notes, meaning it connecting data from various locations within the memory of the computer
* The start of every list is known. However, the end is unknown. Therefore, we can use tail pointers point at the end of linked lists. Having tail pointers improve the run time of inserting, deleting nodes in linked list from the back and regardless, if we have singly or doubly linked lists the run times with tail pointers for the mentioned operations will be n(1)
* *Benefits*
* Linked Lists help us link data together from separate locations. In this way they are infinitely scalable and can contain multiple different types of data. This gives us a flexibility that arrays don’t give us.
* *Cons*
* The main con between linked lists and is the difference between linked lists and arrays is that the data in the linked list aren’t directly accessible.
* For example, if we had an array of tempArray = [H,E,A,P], we could access the P by calling tempArray[3]. This would instantly return us the P.
* With a linked list however, we can’t do that. We only have the first node, and then where that node points to. This means that we have to traverse the entire list before we get to the P. So, if we want the 4###sup/sup### element, we will have to touch the H, E, and A before we get to it.

**A Singly Linked List**

* *General Info*
* A singly linked list follows a specific pattern of nodes. Starting from the first node and going through each other node in order up to the last node then the last point
* Example of a linked list, each of the input is a node that contains data such as an array or a list
* Start -> X - >Y ->a->t -> Null
* The last point is known as “Null”, which means nothing
* *Adding nodes to end of a single linked list*
* When you wish to add a new node to the singly linked list, you replace the “Null” last point space by the new node. Then add a new last point end space also known as “Null”
* *Removing nodes to a single linked list*
* Nodes are dependent on each other. Therefore, the removal of nodes is a tricky process
* Steps of deletion:
* 1. Prior to deleting a specific node. Make sure all of its dependencies are pointing to other nodes, so you do not disturb the chain
* For Example: Delete a in 🡪 Start -> X - >Y ->a->t -> Null
* a is dependent on Y and a provides information to t. Therefore, prior to deleting a. We must create a link between Y and t. Therefore, when we remove a it does not disrupt anything
* 
* *Inserting nodes to the middle of single linked list*
* Nodes are dependent on each other. Therefore, the addition of nodes is a tricky process
* For example, 🡪 insert k between Y and t Start -> X - >Y ->t -> Null
* Steps:
* 1. When adding anything it is not added alone it is always added with “Null” so k, looks more like k-> Null. First connect k to t with an arrow and remove k’s end point “Null”
* 2. Remove the arrow between Y and t and make an arrow goes from Y to k.
* Note: if we did 2 before 1. Theen the linked list would have lost the information of everything that is after Y
* *Run time of Single Linked Lists*
* The average run time it takes the computer to perform to a specific operation on a *Singly* linked lists are presented below:

|  |  |  |
| --- | --- | --- |
| Operation on a fixed array | Run Time | Explained |
| Insert a node at a random location | O(n) | We must go through all the nodes to insert a new node at a random location |
| Insert a node at the front | O(1) | Will need to only edit the where the **new node** will point, and we need to set the **head** to point to our **new node** |
| Insert a node at the end | O(n) | We must go through all the nodes to insert a new node at the end |
| Delete a node from a  Random location | O(n) | We must go through all the nodes to delete the last element from a middle random location |
| Delete a node from the front | O(1) | Will need to make the **head** point at **the second node**. Then delete the head arrow to the **to be deleted fist node**. Then delete the first node |
| Delete a node from the Back | O(n) | We must go through all the nodes to delete the last element from the end |
| Search for a node in an unsorted linked list | O(n) | We must go through all the nodes to search for a node in an unsorted linked list |
| Search for a node t in unsorted linked list | O(n) | We must go through all the nodes to search for a node in an unsorted linked list |
| access time(how long it takes to get to a node within a linked list | O(n) | We must go through all the nodes before a specific location before getting into that specific location. Therefore, at worst case scenario we would need to cover all the nodes |

* *Benefits of Singly Linked lists*
* You can continuously add arrays to the linked list without the need to continuously expand one array. Without worrying to overwriting data and allocating extra unnecessary space

**Doubly Linked List**

* *General Info*
* Each node in a doubly linked list has two arrows. One arrow pointing forward and one arrow pointing backward. This this will typically increase the memory needed for the list, as each node now has another pointer to keep track of
* Doubly linked lists make it easier to get from one node to the other. Instead of covering the while list every time. Therefore, the run time will improve but it won’t improve to the point where it will be lower than O(n)
* All the run times between the singly and double linked list are the same
* The start of every list is known. However, the end is unknown. Therefore, we can use tail pointers point at the end of linked lists. Having tail pointers improve the run time of inserting, deleting nodes in linked list from the back and regardless, if we have singly or doubly linked lists the run times with tail pointers for the mentioned operations will be n(1)
* *Removing nodes to a Doubly linked list*
* While removing a node in a singly linked list we must undergo 3 steps (1. Find the node)(2. Create a connection with the node’s connections, so we do not lose them)(3. Delete the nodes).
* However doubly linked lists make the removing process consist of only 2 steps (1. Find the node)(2.delete the node).
* This is because each node in the doubly linked list has two connections so it can not be lost. This will not improve the run time significantly, but it will produce a better run time still

**Tail Pointers**

* *General Info*
* Tail pointers allows to point to the very end of our lists
* The start of every list is known. However, the end is unknown. Therefore, we can use tail pointers point at the end of linked lists
* The tail pointer continuously points to the end of the linked list, and it changed positions every time a node is added
* Having tail pointers improve the run time of inserting, deleting nodes in linked list from the back and regardless, if we have singly or doubly linked lists the run times with tail pointers for the mentioned operations will be n(1)