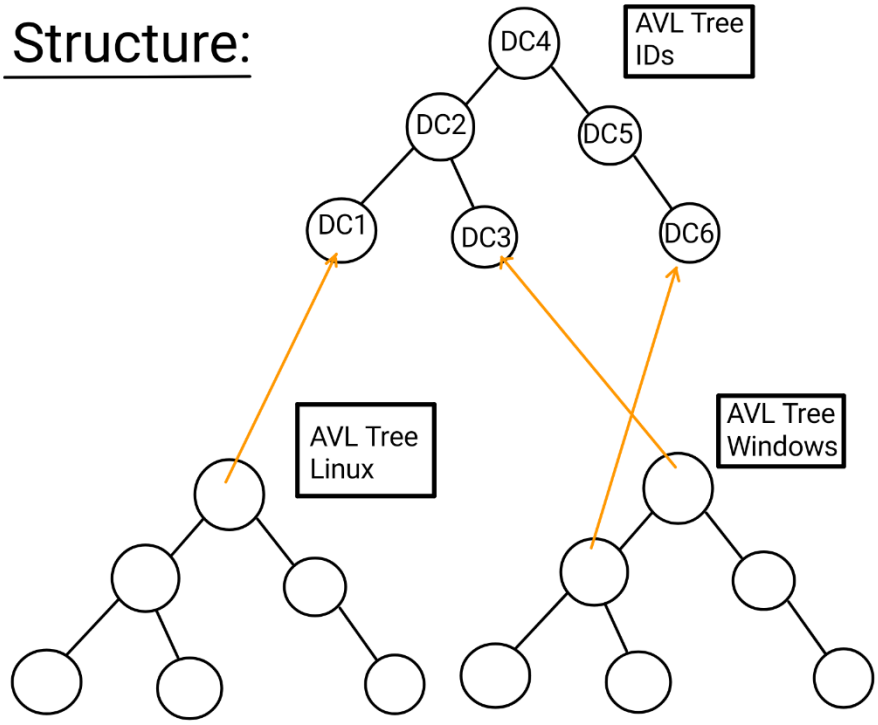
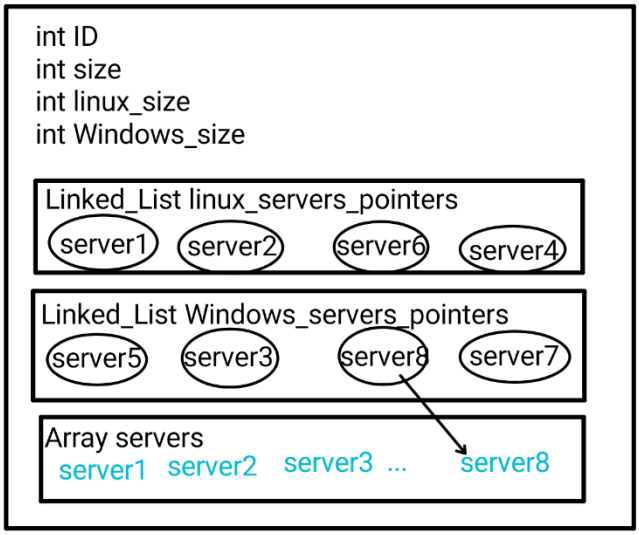
Code structure:

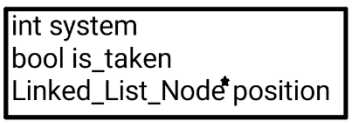
Structure:



DataCenter:



Server:



Our code includes an interface class DataCenterManager, which works with the following underlying structure:

- An AVL Tree (marked "IDs" in sketch) that stores the DataCenters, ordered by ID. This tree is for retrieval of arbitrary DataCenters in O(logn) time complexity.
- An AVL Tree (marked "Linux" in sketch) that stores pointers (actually, wrapper classes for pointers, with minimal functionality) to DataCenters, ordered by number of Linux-running servers in a given DataCenter (and secondarily by ID). This tree is for maintaining a sorted ordering of the DataCenters in preparation for GetDataCenterByOS function call, and is revised every time a server changes operating system to maintain specified ordering.
- An AVL Tree (marked "Windows" in sketch) that stores pointers (actually, wrapper classes for pointers, with minimal functionality) to DataCenters, ordered by number of Windows-running servers in a given DataCenter (and secondarily by ID). This tree is for maintaining a sorted ordering of the DataCenters in preparation for GetDataCenterByOS function call, and is revised every time a server changes operating system to maintain specified ordering.

Structural Classes:

- DataCenter
    - Remembers its ID, size (total number of servers), linux_size (number of Linux-based servers), and windows_size (number of Windows-based servers)
    - Maintains an immutable array of its servers, sorted by server_ID. This is to enable O(1) time complexity retrieval of servers.
    - 2 Linked lists containing pointers to servers (that actually reside in servers array), one for each operating system, sorted by order described in assignment. This is to allow O(1) retrieval of proper alternative server (as defined in assignment) in case requested server is already taken. Proper ordering is maintained by:
        - moving servers over to proper list when they change operating system,
        - always adding newly released serves to tail of linked list (queue-style) and always popping servers from the head, to maintain FIFO ordering (as described in assignment with "priorities". Here priorities are defined as "distance from the tail of linked list"),
        - always removing successfully requisitioned servers from list, so that only available servers remain in list. Servers are returned to appropriate list when they are freed.
- Server
    - Remembers which operating system it is running, its availability status, and where it is located on the pair of linked lists its host DataCenter maintains (a pointer to the Node containing a pointer to said server). The location is to allow proper maintenance of the host DataCenter's linked lists in O(1) time complexity.

Space complexity

The structure and algorithms we selected maintain O(m+n) space complexity (n=number of DataCenters, m=total number of servers) because:

1. We used a constant number of AVL Trees, each of which has one node per DataCenter, for a total of O(n) space complexity. Each Server has a total of 3 pointers to it (one in each linked list in its host and one in its host's servers array), for a total space complexity of O(m) space complexity. Thus a total of O(m+n) space complexity.
2. All recursive algorithms run in logarithmic time (which is less than required linear time) because they operate on trees whose depth is logarithmic and move down a level at each recursion step (thus are sure to come to a stop at the bottom of the tree) and all recursive steps take O(1) time (besides the recursive call, of course) .

Functions and time complexity:

- Init
    - Time complexity: O(1)
    - Init sets up the structure described, setting up a constant number of data structures in O(1) time, thus taking O(1) time. Init initializes the DataCenterManager class with 3 empty AVL Trees (empty, thus building time is unrelated to future content) at $3 * O(1) = O(1)$ time complexity for each tree initialization.
- AddDataCenter
    - Time complexity: O(logn + m)
    - AddDataCenter initializes a DataCenter and adds it to the IDs AVL Tree (taking O(m) time to initialize the DataCenter with all m of the Servers and O(logn) time to add to IDs tree, as proven in lecture – "adding an element to an AVL Tree takes logarithmic time complexity"). It also adds AuxDataCenters (aforementioned classes acting as wrappers for DataCenter pointers, requiring only O(1) time to initialize) to Linux and Windows trees, also taking logarithmic time, for a total of $O(3logn + m) = O(logn + m)$ time complexity. Note: each AVL Tree holds a representative of each and every DataCenter, thus has n elements in it, implying the logarithmic time complexity cited.
- RemoveDataCenter
    - Time complexity: O(logn + m)
    - RemoveDataCenter removes a DataCenter from all 3 trees, taking O(logn) time to find and remove the DataCenter from each tree (as proven in lecture – "finding an element in an AVL Tree takes logarithmic time complexity"). DataCenter's destructor (called once) calls Array's destructor and LinkedList's destructor (twice), each of which take O(m) time complexity, destroying each of the m nodes/Servers in O(1) time. This gives a total of O(logn + m) time complexity for removal from IDs tree and O(logn) time complexity for removal from Linux and Windows trees, trees, for a grand total of $O(3logn + m) = O(logn + m)$ time complexity.
- RequestServer
    - Time complexity: O(logn)
    - RequestServer takes O(logn) time to locate host DataCenter in all 3 AVL Trees (retrieving an element from an AVL Tree takes O(logn) time complexity, as proven in lecture). It then uses the serverID to locate in O(1) time complexity the requested server in Servers array (ID gives a direct mapping to proper cell in Servers array), and after various

maintenance operations (expanded upon in next bullet) returns Server to user after installation (in O(1), by simply toggling the OS member of Server instance) of requested OS on server if necessary, if Server is available. Each server knows if it is available (has a boolean member for the purpose), thus ascertaining availability takes O(1) time complexity. If requested Server is unavailable, the function pops the highest priority server from the appropriate linked list (if non-empty; otherwise it pops a server from the other linked list and installs requested OS on server) in O(1) time complexity (made possible by a direct pointer to the head of the linked list!) and performs the same maintenance operations as the standard request for a server.

  - Maintenance operations include removal of server from host DataCenter's linked lists (to ensure linked lists always contain only available servers) and updating server that it is unavailable. Removal of the server from the host's linked lists is possible despite possessing only a pointer to the node containing the server itself, because aforementioned linked lists are of the doubly linked lists variant. Additionally, every installation of a different OS on a server updates the positioning of the representative AuxDataCenter in both Linux and Windows AVL Trees, by first removing AuxDataCenter from tree (in O(logn) time, as proven in lecture), performing OS installation on server (in O(1) time, as mentioned), and then re-inserting AuxDataCenter back into tree (in O(logn) time, as proven in lecture) so that it will be emplaced properly in tree by insertion algorithm, taking a total of $O(2logn) = O(logn)$ time complexity.
  - Finding the representatives of the host in all 3 trees takes $O(3logn) = O(logn)$ time, locating and preparing the server takes O(1) time, and updating the positioning of the host DataCenter in each of the OS trees takes O(logn) time complexity, for a total of O(logn) time complexity.

- FreeServer
  - Time complexity: O(logn)
  - FreeServer takes O(logn) time to locate host DataCenter in all 3 AVL Trees (retrieving an element from an AVL Tree takes O(logn) time complexity, as proven in lecture). It then uses the serverID to locate in O(1) time complexity the requested server in Servers array (ID gives a direct mapping to proper cell in Servers array), and performs the necessary maintenance operations on the server.
  - The maintenance operations mentioned include:
    - updating the server that it is now available (updating the boolean availability member takes O(1) time complexity)
    - adding the server back into its appropriate linked list, at the tail (to represent lowest priority), taking O(1) time complexity due to a pointer to the tail of the linked list (note: tail pointer gets updated with each addition to the linked list).
  - Time complexity is O(logn) to locate the host DataCenter, O(1) to locate the server, O(1) to update the server's availability status and return it to the host DataCenter's appropriate linked list, and $O(2logn) = O(logn)$ time complexity to update DataCenter's placement in Linux and Windows trees, for a total time complexity of O(logn).
- GetDataCentersByOS
  - Time complexity: O(n)

- GetDataCenterByID simply returns an array containing the InOrder flattening of the appropriate tree (Linux or Windows, as per request). The trees are AVL trees (thus in particular are binary search trees) ordered by number of Windows/Linux servers, so the InOrder flattening gives the requested ordering. GetDataCentersByOS operates on a tree bearing n nodes (1 AuxDataCenter for each DataCenter added) by recursive algorithm demonstrated in the tutorial, with O(n) time complexity as show in the tutorial.

- Quit
  - Time complexity: O(n+m)
  - Quit calls a DataCenterManager's destructor, which calls each tree's destructor (3 trees), which calls the root node's destructor, which recursively destroys its left and right subtrees and then itself, thus destroying the tree in PostOrder. Similarly to proof of time complexity for PostOrder dumping of tree into an array (the only difference between the algorithms is the O(1) time complexity operations to be performed on each node, of which there is a constant number), each node on each tree is reached exactly once, thus calling each of n DataCenter's destructors, each of which destroys all of its own servers (with a total of m servers destroyed), for a total complexity of O(n+m) – m servers destroyed (each in O(1) time complexity) and n DataCenters destroyed (each in O(1) time complexity besides for the destruction of servers, whose destruction is being counted separately). This gives O(n+m) time complexity destruction for each of the 3 trees in the code structure, for a total time complexity for Quit of $O\big(3(n+m)\big) = O(n+m)$.