CS2302 - Data Structures

Summer 2020

Practice Final Exam

1. Write the recursive function *smaller(L,i)* that receives a (native) Python list L and an integer i and returns a Python list containing all items in L are smaller than i, in the reverse order than they appear in L and without modifying L.

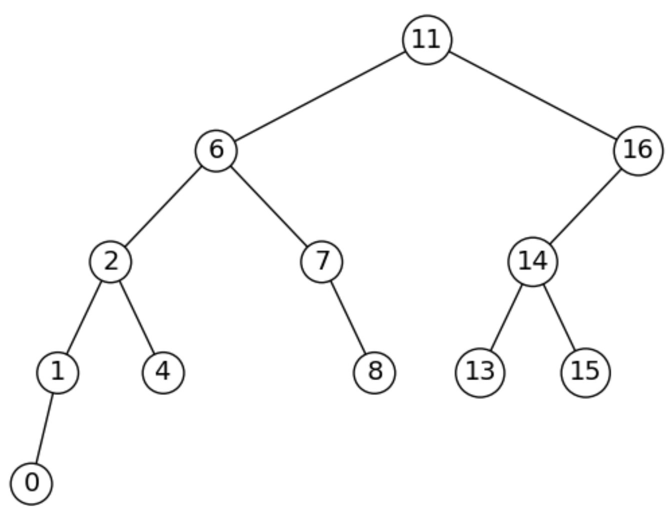
2. Write the function *remove_second(L)* that receives a reference to a List object L (as defined in singly_linked_list.py) and removes the second item in L. If L has less than two items, your function should do nothing. Make sure the *tail* attribute is set correctly. Also, make sure your function runs in $O(1)$ time.

3. The cumulative sum of a list A is a list C of the same length as L where $C[i]$ contains $A[0]+A[1]+...+A[i]$. Thus $C[0] = A[0]$, $C[1] = A[0] + A[1] = C[0] + A[1]$, and, in general $C[i] = C[i-1] + A[i]$. For example, if $A = [2, 3, 1, 4]$, then $C = [2, 5, 6, 10]$. Write the function *cumulative_sum(L)* that receives a reference to a List object L (as defined in singly_linked_list.py) and builds and returns a List object containing the cumulative sum of L.

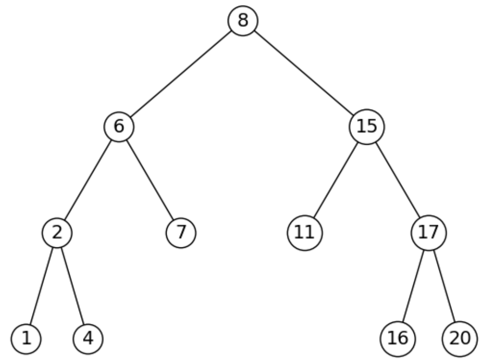
4. Write the function *equal_row(A)* that receives a 2D numpy array A and returns a list containing the indices of the rows in A where all elements are equal.

5. Write the function *sorted_row(A)* that receives a 2D numpy array A and returns a list containing the indices of the rows in A that are sorted in ascending order.

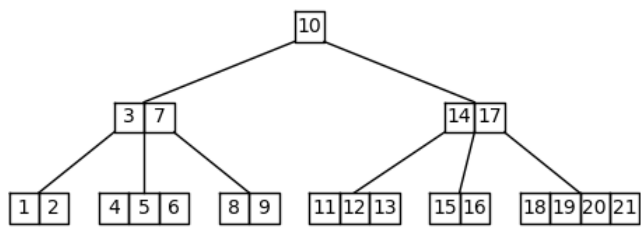
6. Write the function *max_at_depth_bst(T,d)* that receives a reference to the root of a binary search tree T and an integer d and returns the largest element in the tree that has depth d or *-math.inf* if the tree has no elements at depth d. For example, if T is the root of the tree in the figure, *max_at_depth_bst(T,0)* should return 11, *max_at_depth_bst(T,1)* should return 16, *max_at_depth_bst(T,2)* should return 14, *max_at_depth_bst(T,3)* should return 15, *max_at_depth_bst(T,4)* should return 0 and *max_at_depth_bst(T,5)* should return *-math.inf*.



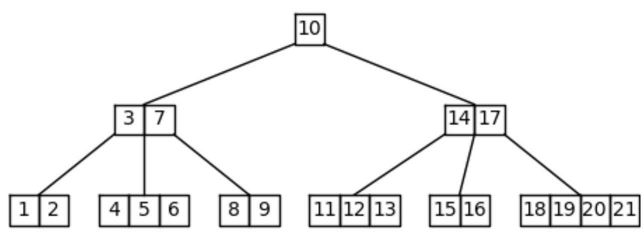
7. Write the function *in_leaves(T)* that receives a reference to the root of a binary search tree T and returns a list containing the items that are stored in leaf nodes in the tree. For example, if T is the root of the tree in the figure, *in_leaves(T)* should return $[1, 4, 7, 11, 16, 20]$.



8. Write the function *max_at_depth_btree(T,d)* that receives a reference to the root of a B-tree T and an integer d and returns the largest element in the tree that has depth d or *-math.inf* if the tree has no elements at depth d. For example, if T is the root of the tree in the figure, *max_at_depth_btree(T,0)* should return 10, *max_at_depth_btree(T,1)* should return 17, *max_at_depth_btree(T,2)* should return 21, and *max_at_depth_btree(T,5)* should return *-math.inf*.



9. Write the function *internal(T)* that receives a reference to the root of a B-tree T and returns a list containing the items that are stored in internal (non-leaf) nodes in the tree. For example, if T is the root of the tree in the figure, *internal(T)* should return $[3, 7, 10, 14, 17]$.

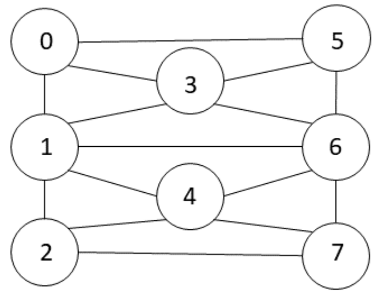


10. A (much) simpler version of *subsetsum* consists of, given a list of integers S and a goal k, determining if there are two elements of S that add up to k. This problem can be solved in $O(n)$ using a hash table, as done by the function *find_sum_pair(S,k)*. The function provided returns *True* if the pair of numbers exists and *False* otherwise. Modify it to return a list containing the two numbers, if they exist, and *None* otherwise. For example, if $S = [1, 3, 6]$, *find_sum_pair(S,7)* should return $[1, 6]$ and *find_sum_pair(S,10)* should return *None*.

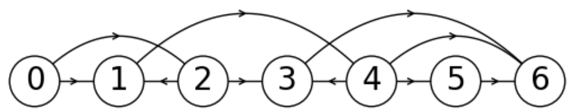
11. Write the function *remove_duplicates(L)* that receives a list L and returns a list containing the elements of L after removing duplicates, in the same order as they appear in L. For example, if $L = [4, 2, 7, 9, 7, 8, 1, 9, 2, 4]$, your function should return the list $[4, 2, 7, 9, 8, 1]$. Your function **must** run in $O(n)$ time and use a hash table with chaining, as implemented in *hash_table_chain.py*.

12. Three vertices u, v, w form a clique in an undirected graph $G = (V, E)$ if there are edges connecting every pair of vertices in (u, v, w) (that is, $(u, v) \in E$, $(u, w) \in E$, and $(v, w) \in E$. Write the function

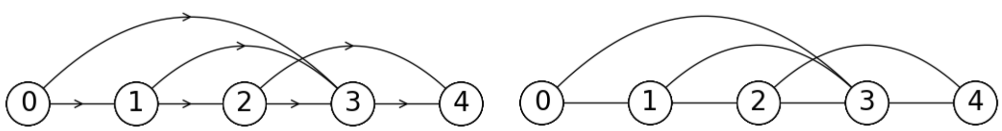
clique(G, u, v, w) that receives a graph represented as an adjacency matrix and vertex indices u, v and w and determines if they form a clique. For example, in the graph below, $(0, 1, 3)$ form a clique (thus *clique*$(G, 0, 1, 3)$ should return True) and $(0, 1, 2)$ do not form a clique.



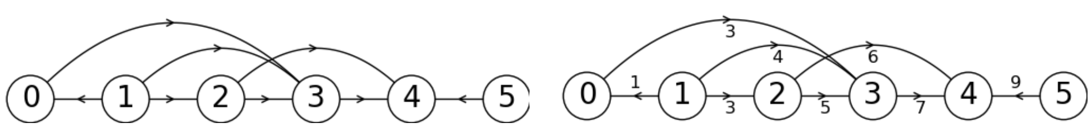
13. Write the function *first_ts(G)* that receives a directed graph $G = (V, E)$ represented as an adjacency list and returns a list containing the vertices that could start a topological sort of G. Hint: what feature does the first vertex in a topological sort have? Hint 2: you don't need to find the topological sort, or verify if one exists. For example, in the graph below, *first_ts(G)* should return [0].



14. Write the function *make_undirected(G)* that receives a directed graph G represented as an adjacency matrix and converts G to an undirected graph. For example, if G is the graph on the left, after executing *make_undirected(G)*, G should be the graph on the right.



15. Write the function *make_weighted(G)* that receives an unweighted graph G represented as an adjacency list and converts G to a weighted graph, where the weight of an edge is the sum of the indices of the vertices it connects. For example, if G is the graph on the left, after executing *make_weighted(G)*, G should be the graph on the right.



16. Write the function *am_to_el(G)* that receives a graph represented as an adjacency matrix and builds and returns the edge list representation of the same graph.

17. The function *subsetsum(S,g)* returns *True* if there is a subset of the set of positive integers S that adds up to goal g. Write the function *subsetsum_count(S,g)* by modifying *subsetsum(S,g)* to return the number of subsets of S that add up to g.

18. Write the function *edit_distance_with_wildcard(s1,s2)* that is identical to *edit_distance(s1,s2)* but takes the character '*' as a wildcard, which matches any character in the other string.