IntroCS: Graphs & (Christmas) Trees

Maximilian Harl*, Sebastian Dunzer†

Winter Term 19/20

This Christmas Set contains 6 pages (including this cover page) and 6 questions.

This is the Christmas Set from "Introduction to Computer Science" for International Information Systems Students from the Winter Semester 2019/2020. It is supposed to revise sorting algorithms, trees (binary, search, etc.) and graphs.

Merry Christmas from the IntroCS Team!

---

1. Quick[ly grabbing all presents] Sort.
   QuickSort is a sorting algorithm that is in a similar tier like merge sort. It is a divide & conquer sorting algorithm which uses comparison to sort.
   QuickSort relies on so-called "pivot-elements". In our Christmas setting, you could imagine a child sorting presents. The child chooses any "pivot"-present and sorts all its other presents by pushing larger presents to its right and smaller ones to its left (compared to the "pivot"-element), until they are all sorted. Then, the child proceeds on both "sides" of the pivot present in the same way.
   As a simplification you can use the last element of each block as the pivot element. You are only required to implement the ascending sorting.
   You are completely free in your implementation, but to provide you some orientation and the possibility to check you code with our test cases, we suggest you stick to the following specifications:
   Your module should read $argv[1]$ and sort the given string.
   You should create a function $quickSort(arr, lower, upper)$ where arr is the list that is to be sorted, lower the starting index for the sorting and upper the end index for the sorting.
   You should implement everything in a file called *qsort.py*. Take a look at the pseudo code below.

*maximilian.harl@fau.de
†sebastian.dunzer@fau.de

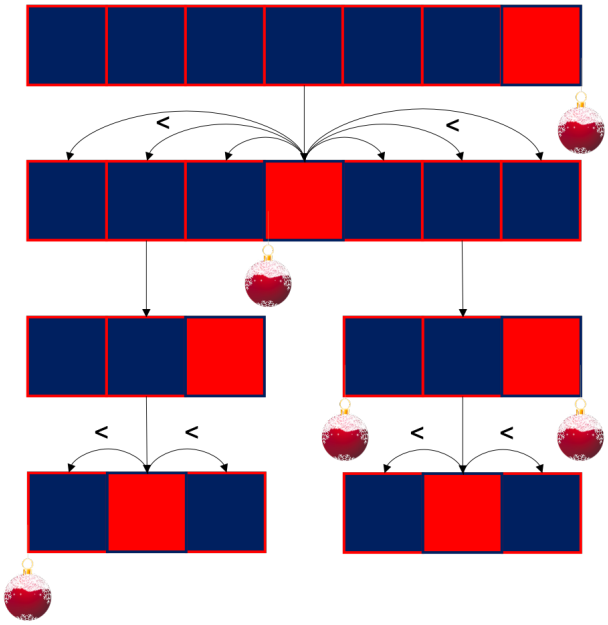

Figure 1: The pivot-element and its position changes during sorting.

Data: An array arr that is to be sorted
Result: The sorted array arr
if $len(arr) <= 1$ **then**
    return arr;
else
    Divide:
    Select some pivot-element $pivot$ from arr.
    Calculate the subsequence arr_1 from arr, with the elements, whose values are $<= pivot$, and subsequence arr_2 from arr, with the elements $> pivot$.;
    Conquer:
    $arr_{1'} := QuickSort(arr_1)$;
    $arr_{2'} := QuickSort(arr_2)$;
    Merge:
    return $concat(arr_{1'}, arr_{2'})$;
end

Algorithm 1: QuickSort

2. Binary Trees.
   This task familiarises you with binary trees.
   A binary tree is a tree, where each node has at most two children. You can imagine a tree to be the triple $(value, left\_child, right\_child)$.
   Your task is to create a binary tree that is complete from a given array.
   E.g: INPUT: {1, 2, 3, 4, 5, 6, 7, 8}
   OUTPUT:

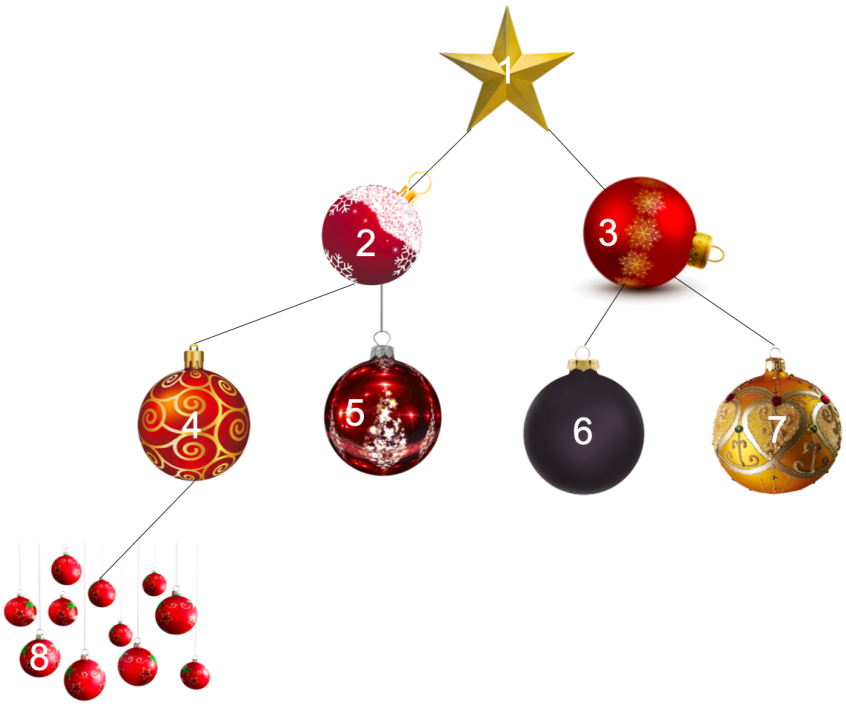

Figure 2: Ho Ho Ho!

You are not required to create a graphical output of the tree. You are, apart from the tree implementation and representation in python, required to create function $tree\_print(bin\_tree)$ that traverses the tree and returns the nodes of the tree in inorder sequence. The in-order sequence for the example tree is $< 8, 4, 2, 5, 1, 6, 3, 7 >$. (Hint: remember the array-embedding of a tree) Note, that the other ways to traverse a tree are preorder and postorder.
To check your code, implement everything in a class $BinTree$, in a file $bin\_tree.py$. As node you should create another file $Node.py$ that looks as follows:

```
class Node:
    def __init__(self, payload):
        self.payload = payload
        self.left = None
        self.right = None
```

3. Binary Search Trees.
   The third tree task is the implementation of a binary search tree.
   Reminder: A binary search tree is a tree where all values that are less than the value of a given node are in its left sub-tree and all values that are greater than the value of a given node are in its right sub-tree.
   You need to set up a binary search tree from a given array. Keep in mind, that the order of the inserted values is very important!
   Implement everything in a file called *bin_search_tree.py* in a class called *BST*. You should implement an *add()*, *delete()*, *bfs_it()* and *dfs_rec()* function.

   i) *void add(Node root, int value)*:
   Adds a node with the value value to the tree at its correct position according to the binary search tree property.

   ii) *void delete(Node root, int value)*:
   Deletes the node with the value value from the binary search tree and makes sure, that its properties as a search tree are still intact.

   iii) *void bfs_it(Node root)*:
   Iterates over the tree specified by root iteratively in a breath-first manner.

   iv) *void dfs_rec(Node root)*:
   iterates the tree specified by root recursively in a depth-first manner.

   helper) *void find(Node root, int value)*:
   searches for the Node specified by the value value in the tree and returns its node.

   You are allowed to implement as many helper functions as you like.

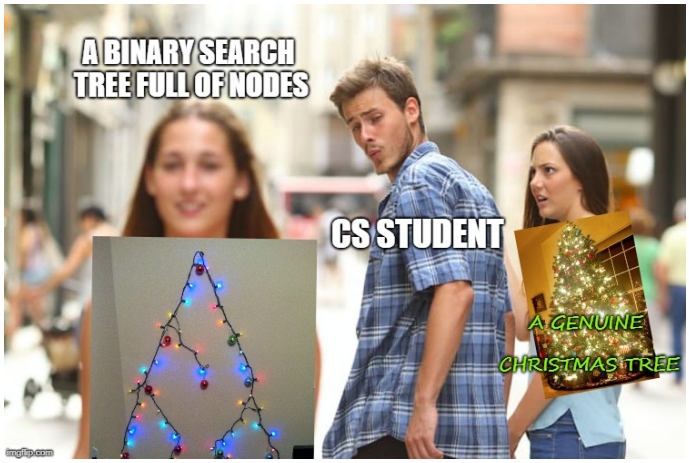

Figure 3: Merry Christmas! Celebrate the holidays!

4. AVLTree.
   In the lecture and through our shorts you got to know AVL-Trees. Now it is your task to implement AVL-Trees. The specific functions you need to implement are:

   $insert(self, root, value)$ A dynamic function that inserts a new node with the value $value$ and returns the new root of the tree (since the root can change due to rotations).

   $delete(self, root, value)$ A dynamic that function deletes the node specified by $value$ and returns the new root of the tree (since the root can change due to rotations).

   These function should be implemented in a file called $avl.py$ and in a class called avl_tree. A node of the tree (specified in the file $Node.py$) looks as follows:

```
class Node:
    def __init__(self, payload):
        self.payload = payload
        self.left = None
        self.right = None
        self.height = 1
```

   You can add any properties that you think are useful. With this type of node think about how to calculate the height of the with less effort (trade-off between speed and memory). Furthermore, use helper functions where possible, to shorten your code (e.g. $leftRotate()$, $rightRotate()$, $getHeight()$, $getBalance()$, ...)

5. DFS on general graphs.
   After having watched the shorts on depth-first search on graphs, you should understand the general concept. To enforce the knowledge, you program a DFS traversal in an iterative way using a "stack". Your dfs should accept two arguments.

   - node to start from.
   - node to find.

   if *found the element* **then**
   | return the number of nodes that your DFS crossed until you found the element and the path it took there as list of nodes
   else
   | return -1, [.]
   end

   The constructor of your class should accept a list $[node, directed_edge]$ from which it constructs its graph representation. Your class should contain a function $dfs(self, root, tofind)$ that returns the values as stated above.
   For testing purposes your class should be called $Graph$ in a file $dfs\_gr.py$ and additionally contain a function $add(self, start, end)$ that adds an edge from the node with value $start$ to the node with value end to the graph.

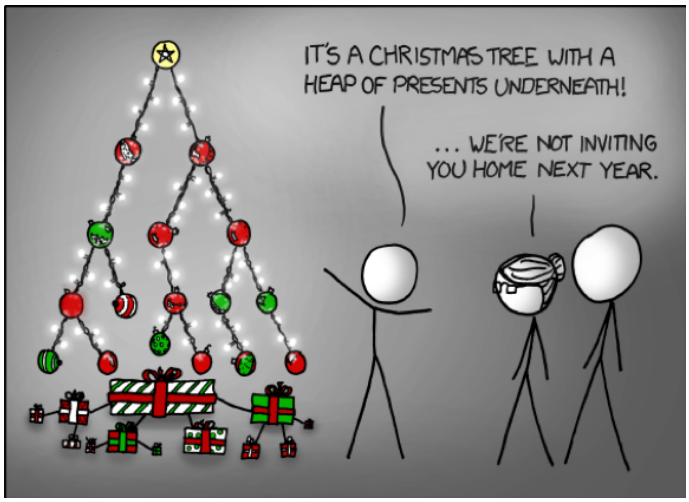

Figure 4: Sorry, that there is no heap in this problem set. Do you start to feel the way we do since we had our own "IntroCS"?

6. (3 karma points) In no more than one paragraph, proof that Santa Clause or the Christkind exists. And that Christmas is more than a commercial exploitation of the public by corporate greet. Also explain the spirit of Christmas and prove that or explain why the meaning of life is 42.