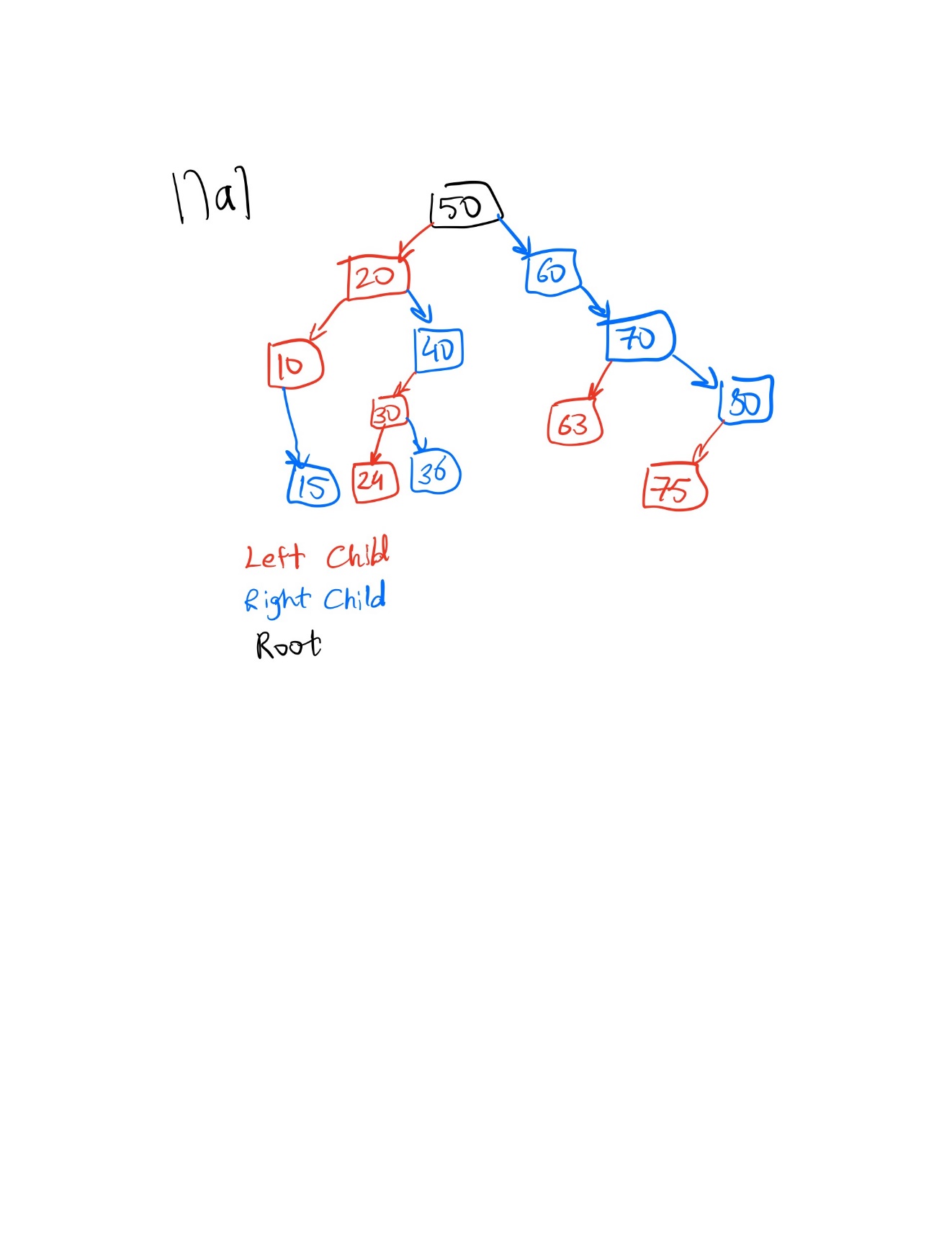
1.a. picture below

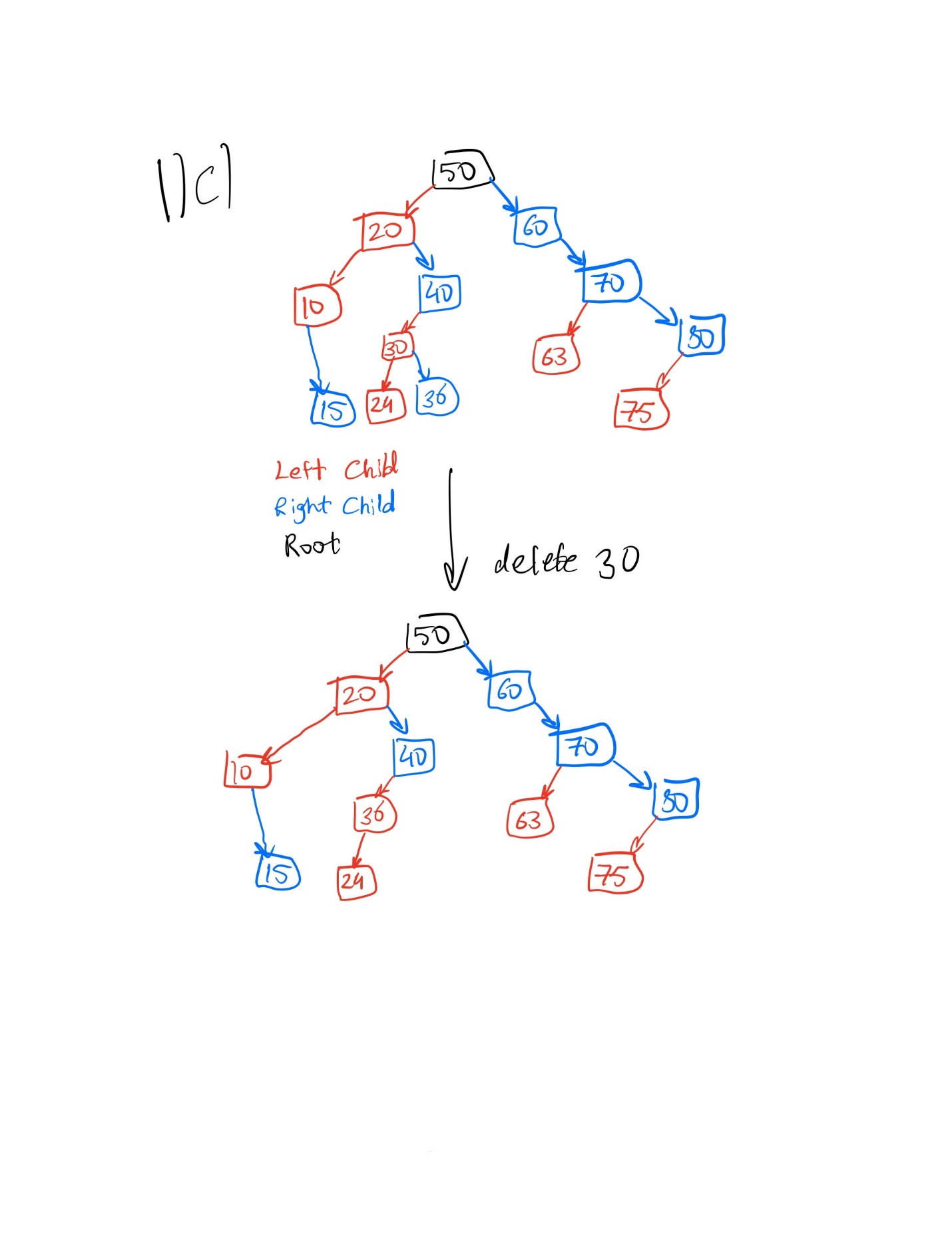


1.b.

Inorder traversal: 10 -> 15 -> 20 -> 24 -> 30 -> 36 -> 40 -> 50 -> 60 -> 63 -> 70 -> 75 -> 80

preorder traversal: 50 -> 20 -> 10 -> 15 -> 40 -> 30 -> 24 -> 36 -> 60 -> 70 -> 63 -> 80 -> 75

postorder traversal: 15 -> 10 -> 24 -> 36 -> 30 -> 40 -> 20 -> 63 -> 75 -> 80 -> 70 -> 60 -> 50

1.c. Pictures below Diagram

Description automatically generated with medium confidence

1. In some binary search trees, each node has a left child pointer, a right child pointer and a parent pointer. The parent pointer of a node points to its parent (duh!), or is nullptr if the node is the root node. This problem will examine such trees.
   1. Show a C++ structure/class definition for a binary tree node that has both child node pointers and a parent node pointer. Assume the data stored in each node is an int.

**class** Node

{

**public**:

add(const int& newdata, Node\* rootPtr);

**int** data;

Node\* parent

    Node\* left;

    Node\* right;

};

* 1. Suppose this tree is being used to represent a set of integers. Write pseudocode to insert a new node into a binary search tree with parent pointers. Since this is for a set, attempting to insert a value already in the tree does nothing. It is your choice what parameters the function takes, whether it is recursive or not, whether it uses an auxiliary function you write, etc. (Hint: You can find binary search tree insertion code on pp. 471-473 of the Carrano book).

Node::add(const int& newData, Node\* rootPtr)

{

Node\* newNodePtr = new Node;

Set the data member variable of newNodePtr to newData and its left, right, and parent pointers to nullptr

rootPtr = insertInorder(rootPtr, newNodePtr);

return true;

}

insertInorder(Node\* rootPtr, Node\* newNodePtr)

{

Search the tree pointed to by rootPtr for the item in the node pointed

to by newNodePtr (newNodePtr’s data)

if (the search terminates at parentNode’s left subtree )

Set the left member variable of parentNode to newNodePtr

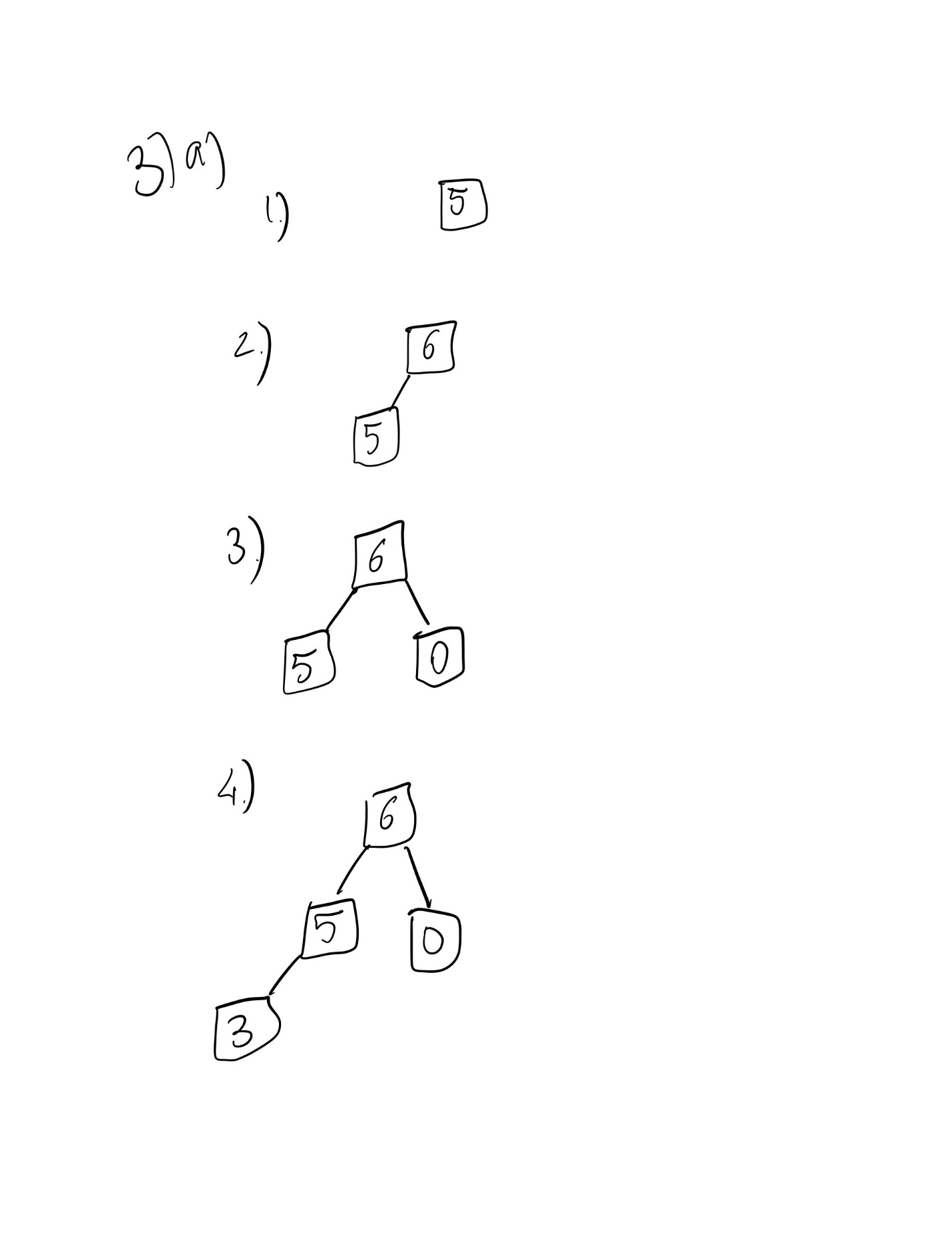
And set newNodePtr’s parent member variable to point to parentNode

else

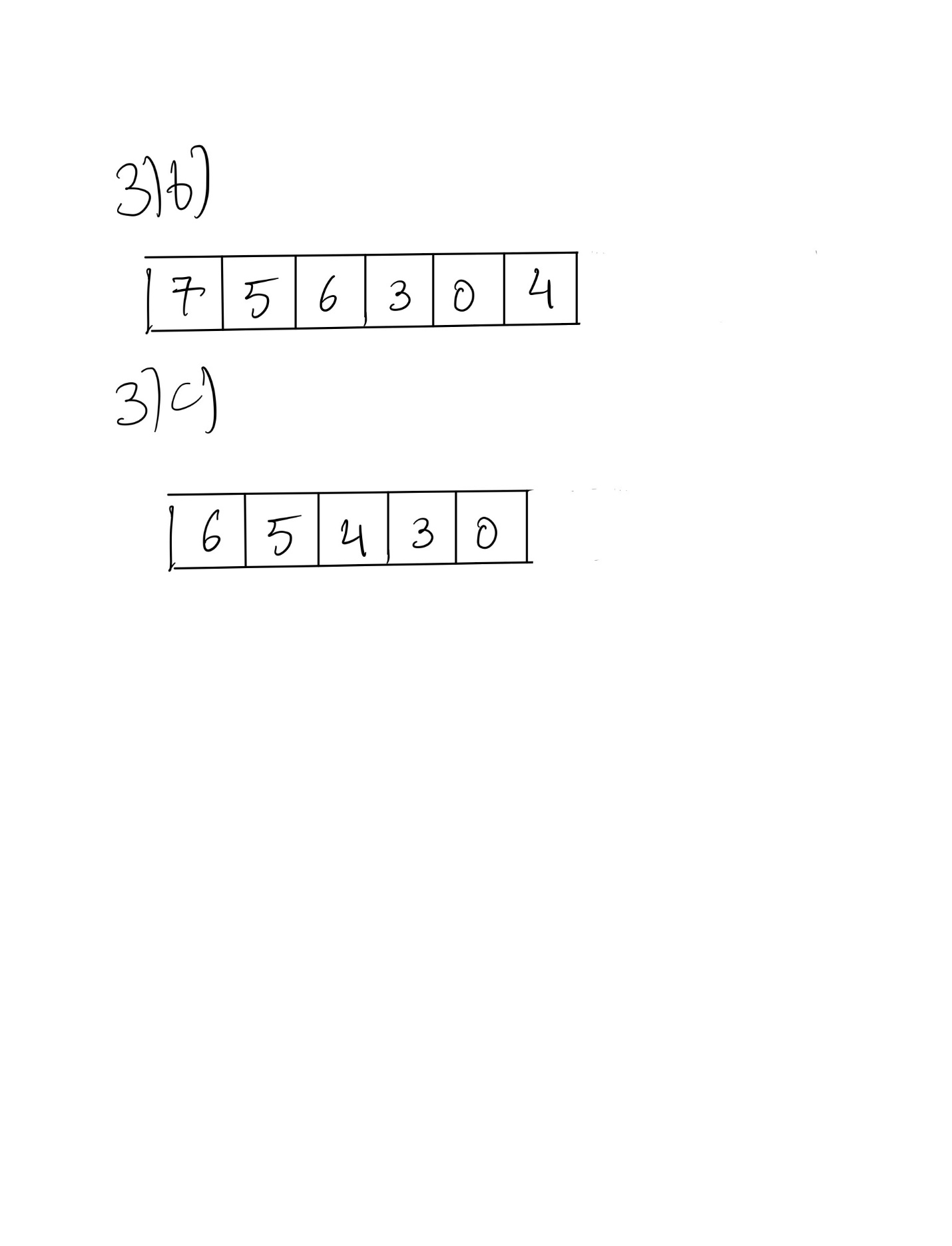
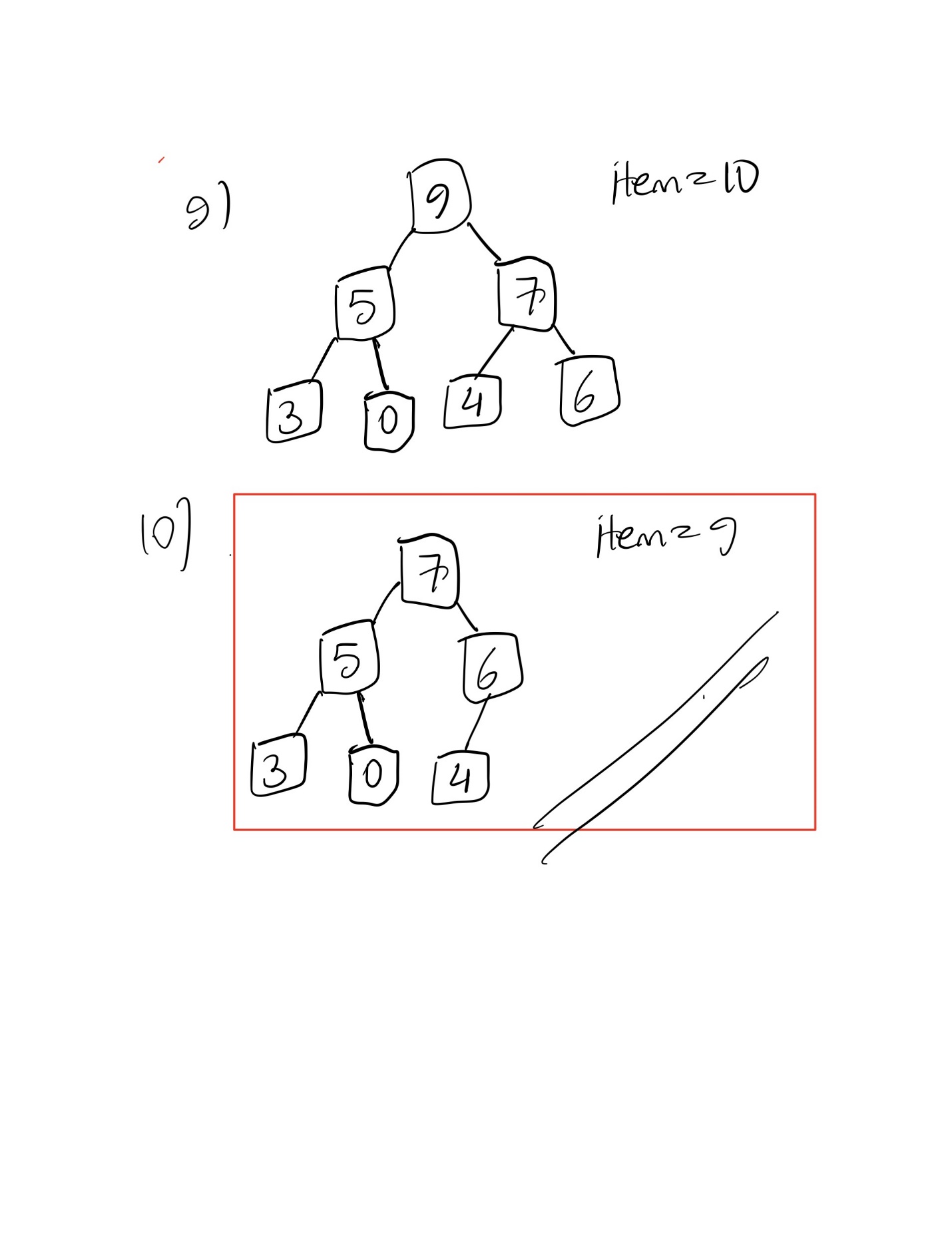
Set the right member variable of parentNode to newNodePtr

And set newNodePtr’s parent member variable to point to parentNode

}

 Diagram, schematic

Description automatically generated Diagram

Description automatically generated

1. Note: A pair<T1, T2> is a simple struct with two data members, one of type T1 and one of type T2. A set<K> and a map<K, V> are organized as approximately balanced binary search trees; an unordered\_set<K> and an unordered\_map<K, V> are organized as hash tables that never allow the load factor to exceed some constant, and a loop that visits every item in a hash table of N items is O(N). For the keys to be hashed, the hash function used produces uniformly distributed results.

Suppose UCLA has **C** courses each of which has on average **S** students enrolled. For this problem, courses are represented by strings (e.g. "CS 32"), and students by their int UIDs. We will consider a variety of data structures, and for each determine the big-O time complexity of the appropriate way to use that data structure to determine whether a particular student *s* is enrolled in course *c*. For example, if the data structure were vector<pair<string, vector<int>>>, where each pair in the outer vector represents a course and all the students in that course, with those students being sorted in order, then if the pairs are in no particular order in the outer vector, the answer would be O(C + log S). (The reason is that we'd have to do a linear search through the outer vector to find the course, which is O(C), and then after that do a binary search of the S students in the sorted vector for that course, which is O(log S).) In these problems, we're just looking for the answer; you don't need to write the reason.

* + 1. vector<pair<string, list<int>>>, where each pair in the outer vector represents a course and all the students in that class, with those students being sorted in order. The pairs are in no particular order in the outer vector. What is the big-O complexity to determine whether a particular student *s* is enrolled in course *c*?

**O(C + S)**

* + 1. map<string, list<int>>, where the students in each list are in no particular order. What is the big-O complexity to determine whether a particular student *s* is enrolled in course *c*?

**O(log(C) + S)**

* + 1. map<string, set<int>>. What is the big-O complexity to determine whether a particular student *s* is enrolled in course *c*?

**O(log(C) + log(S))**

* + 1. unordered\_map<string, set<int>>. What is the big-O complexity to determine whether a particular student *s* is enrolled in course *c*?

**O(1 + log(S))**

Note: accounting for collisions, the hash table entry may have a linked list of size N. In this case, the worst case time complexity would be

O(N + log(S))

where for the given key (in this case the course name) we visited every item in a hash table of N items

* + 1. unordered\_map<string, unordered\_set<int>>. What is the big-O complexity to determine whether a particular student *s* is enrolled in course *c*?

**O(1 + 1) approx.. O(1)**

Note: accounting for collisions, the hash table entries for the tables may have a linked list of size N1 and N2 respectively. In this case the, the worst case time complexity would be

O(N1 +N2)

where for the given key (in this case the course name and the student id) we visited every item in a hash table of N1 and N2 items for the unordered\_map and unordered\_set respectively.

* + 1. Suppose we have the data structure map<string, set<int>> and we wish for a particular course *c* to write the id numbers of *all* the students in that course in sorted order. What is the big-O complexity?

**O(log(C) + S\*log(S))**

* + 1. Suppose we have the data structure unordered\_map<string, unordered\_set<int>> and we wish for a particular course *c* to write the id numbers of *all* the students in that course in sorted order (perhaps using an additional container to help with that). What is the big-O complexity?

**O(1 + 1\*S+S)** **approx.. O(S)**

// assuming the sorted container is given to us // +S for looking up students in the unordered\_set, +S for iterating through the additional container

Note: assuming we use quicksort to sort the additional container our answer would be O(1 + 1\*S+S\*log(S)+S)

Note: in the worst case we would have

O(N1 + N2\*S+S)

assuming we use quicksort to sort the additional container our answer would be O(N1 + N2\*S+S\*log(S)+S)

where for the given key (in this case the course name and the student id) we visited every item in a hash table of N1 and N2 items for the unordered\_map and unordered\_set respectively

* + 1. Suppose we have the data structure unordered\_map<string, set<int>> and we wish for a particular student *s* to write *all* the courses that student is enrolled in, in no particular order. What is the big-O complexity?

**O(C+C\*log(S))**

Note: in the worst case we will have a time complexity of

O(C\*N1+C\*log(S))

where for the given key (in this case the course name) we visited every item in a hash table of N1 (on average since the number of collisions might be different for different course names) items for the unordered\_map.