BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

CS-F211: Data Structures and Algorithms

Lab 10: AVL Trees and 2-4 Trees

Introduction

Welcome to week 10 of Data Structures and Algorithms! Last week, we studied a riveting data structure - Binary Search Trees. We studied various operations that can be performed on a BST such as insertion, querying, deletion, retrieving the k^{th} smallest element, etc. For most of these operations, for a tree having `n` nodes, the time complexity came out to be `O(h)`, where h is the height of the tree. While this height can be `O(log n)` when the tree is balanced, in the worst case it can go to `O(n)` when the tree is extremely skewed. We would be studying how to ensure `O(log n)` depth today using two different methods, leading to two data structures - AVL Trees and 2-4 Trees.

Topics to be covered in this labsheet

- Introduction to AVL Trees
    - Rotations
    - Insertion
    - Deletion
- Introduction to 2-4 Trees
    - Insertions
    - Deletions

Introduction to AVL Trees

An AVL tree (named after inventors Adelson-Velsky and Landis) is a self-balancing binary search tree. It is a binary search tree that has to satisfy the height balancing property. The **height balancing property** says that the heights of the two child subtrees of any node differ by at most one.

Any binary search tree T that satisfies the height balancing property is said to be an AVL tree.

An example of an AVL tree is shown in Figure 1. The keys of the entries are shown inside the nodes, and the heights of the nodes are shown above the nodes. You can appreciate the height balancing property here, as for any node, the heights of the children differ by at most 1, which leads to an *almost* balanced tree and ensures logarithmic height[1].

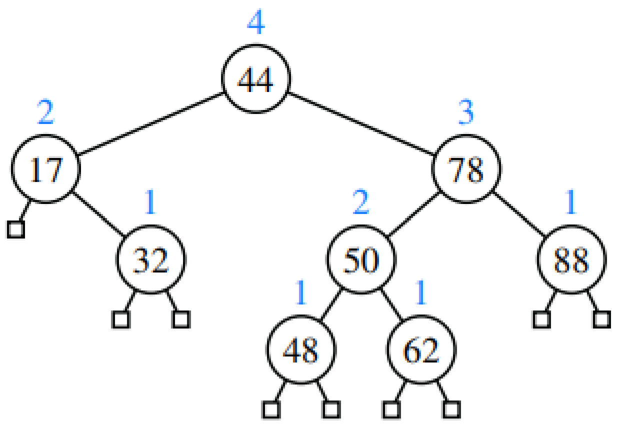

Figure 1: An example of an AVL tree.

Home Exercise 1: Write a function that takes a pointer to a BST (as defined in lab sheet 9) and checks if it is an AVL tree.

```
int is_avl(struct bst *bst);
```

The above function must return 0 when the tree is not an AVL tree and 1 when it is an AVL tree. You might make use of a helper function to check if a node is height-balanced as follows:

```
int is_height_balanced(struct node *node) // Returns -1 if not balanced, otherwise returns height. The height of leaf is 0.
```

[1] Refer to the proof of Theorem 3.2 as given in Michael T. Goodrich, Roberto Tamassia - Algorithm Design. Foundations, Analysis, and Internet Examples-Wiley (2001)

The `is_height_balanced()` function can be implemented in a recursive fashion and thus `is_avl()` is now reduced to `is_height_balanced(bst->root).` Complete the function definitions and test them out using an appropriate `main().`

Now, the difficult part is ensuring that this property is preserved while inserting and deleting nodes into an AVL tree[2]. For this, we define a new useful operation on a tree called rotations.

[2] You can visualize the various AVL tree operations on the following website - https://visualgo.net/en/bst?slide=14

Rotations

The primary operation to rebalance a binary search tree is known as a rotation. During a rotation, we *rotate* a parent to be below its child, as illustrated in Figure 2.

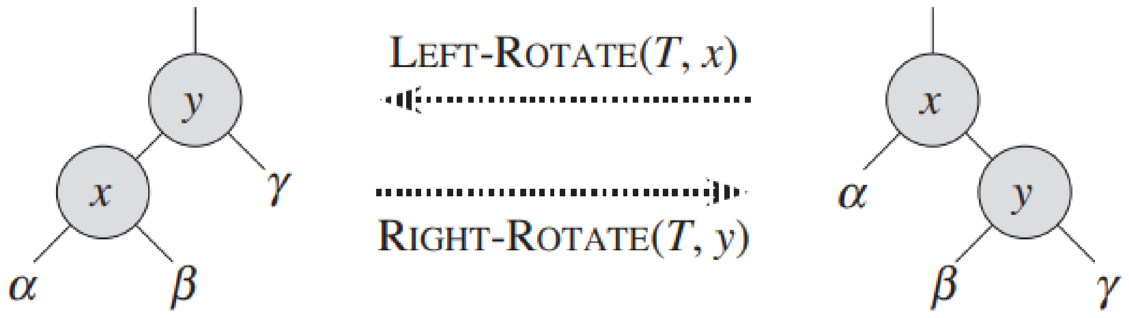

Figure 2: Rotation operations illustrated in a binary search tree

The operation LEFT-ROTATE(T,x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers. The inverse operation RIGHT-ROTATE(T,y). The letters α, β and γ represent arbitrary subtrees. The rotation operation preserves the binary-search-tree property: the keys in α precede x.key, which preceded the keys in β, which precede y.key, which preceded the keys in γ.

The mechanics of the rotate-left operation is illustrated in code snippet 1 given below:

```c
struct node *rotate_left(struct node *x)
{
    struct node *y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}
```

Code Snippet 1: Rotate left

You can write the rotate_right function which works in a similar manner by yourself.

Building an AVL Tree

We will first define an insert() method that inserts a node into an AVL tree while ensuring that the height balance property is maintained, and the resulting tree is also an AVL tree. Then we can create an AVL tree by simply inserting the elements one at a time while ensuring that at each step, the tree is an AVL tree.

First, we insert a node as we do for a usual binary search tree. Then we perform rotations on the tree to ensure that it satisfies the height balance property. When a node is inserted, the height balance property may be violated for multiple nodes. However, an interesting thing is that all of these nodes where the property can be violated are the ancestors of the newly inserted node.

So we need to traverse up from the newly inserted node up to potentially the root and perform rotations whenever we encounter a height imbalance.

For example, consider the following figure:

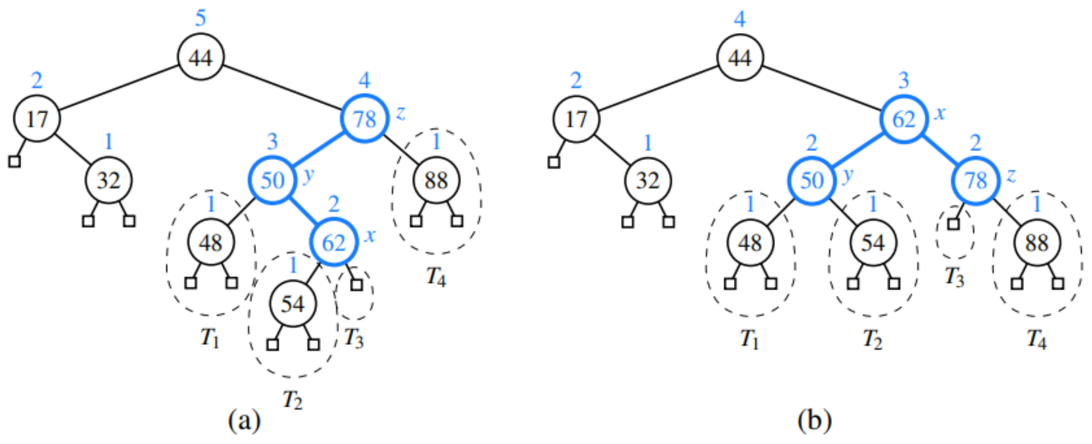

Figure 3: Height imbalance created due to the insertion of 54

Here, when node 54 is inserted into an AVL tree, there is an imbalance created at node 78 as the heights of the ancestors of 54: 62, 50, 78 and 44 are affected. So we perform a left rotation on 50 followed by a right rotation on 78 to transform tree (a) into tree (b) which is now an AVL tree.

There are different kinds of imbalances based on the structure of the child and the grandchild towards the inserted node from the first node of imbalance encountered while traversing upwards.

We can achieve this by making the insertAVL() a recursive function, which starts at the root, adds to either subtree and at every step, once the control returns to the calling function after the recursive call, can now perform rotations to balance the tree.

There are four types of imbalances that can possibly be created after the insertion of a node:

1. **LL Imbalance (or similarly RR Imbalance):** Here, the new node is part of the left subtree of the left child of the first unbalanced node encountered while traversing upwards from the added node to the root.

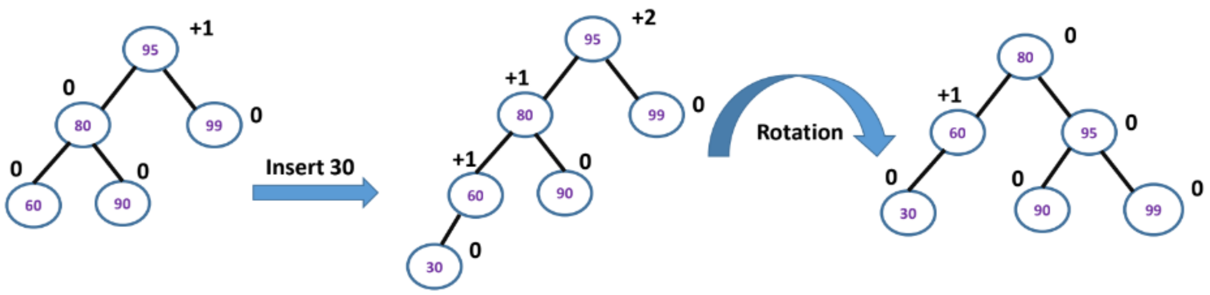

Figure 4: LL imbalance being solved by a single right rotation

Here, a single right rotation can balance the node up to this level.

Similarly, a single left rotation can balance an RR imbalance.

2. **LR Imbalance (or similarly RL imbalance):** Here, the new node is a part of the right subtree of the left child (say **X**) of the first imbalanced node (say **Z)** encountered while traversing towards the root. Here, two rotations are required to balance this level. First, a left rotation is required on **X** to convert the LR imbalance into an LL imbalance. Now a right rotation on **Z** as above can fix the imbalance at this level and we can continue moving upwards to fix any other unbalanced nodes.

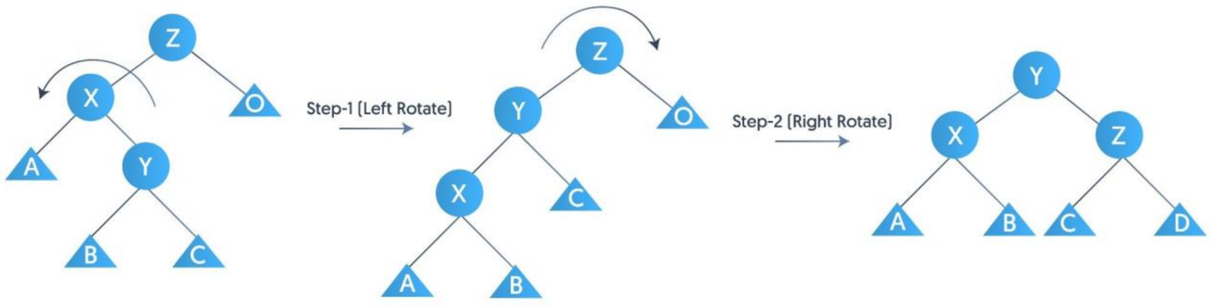

Figure 5: LR Imbalance being solved by two rotations

Here, node Z is imbalanced and the newly added node is a part of the subtree rooted at Y. So the way to rebalance the tree is to first left-rotate X followed by right-rotating Z.

Similarly, an RL imbalance can be solved by a left rotation followed by a right rotation.

So, after the node has been inserted into an AVL we move upwards towards the root and at every imbalanced node, apply appropriate rotations to restore the AVL property to the tree.

Consider code snippet 2 given below:

```
struct node *insertAVL(struct node *node, int value)
{
    if (node == NULL)
    {
        node = new_node(value);
    }
    else if (value < node->value)
    {
        node->left = insertAVL(node->left, value);
    }
    else
    {
        node->right = insertAVL(node->right, value);
    }
    int balance = is_height_balanced(node);
    if (balance == -1)
    {
        if (value < node->value)
        {
            if (value < node->left->value)
            {
                // LL imbalance
                node = rotate_right(node);
            }
            else
            {
                // LR imbalance
                node->left = rotate_left(node->left);
                node = rotate_right(node);
            }
        }
        else
        {
            /*
                Complete the code for the following cases:
                RR imbalance
                RL imbalance
            */
        }
    }
    return node;
}
```

Code Snippet 2: Incomplete implementation of insertAVL()

Task 1: To the *bst.c* file, **add functions rotate_left() and rotate_right()**. Now **complete the insertAVL() function** as per code snippet 2 given above using appropriate calls to rotate_left() and rotate_right() as per the different imbalance conditions.

Now, **write a main()** that adds nodes in the order: 1, 2, 3, 4, 5, 6, 7, 8, 9; using the insertAVL() function as defined here. **Check** whether the resulting BST is an AVL tree or not. Also, observe the structure of the tree by **printing the nodes in breadth-first order.**

You can use the function given in code snippet 3 below for the breadth-first traversal of the tree:

```
void traverse_bfs(Node *node)
{
    if (node == NULL)
    {
        return;
    }
    Node *queue[100];
    int front = 0;
    int back = 0;
    queue[back++] = node;
    while (front != back)
    {
        Node *current = queue[front++];
        printf("%d ", current->value);
        if (current->left != NULL)
        {
            queue[back++] = current->left;
        }
        if (current->right != NULL)
        {
            queue[back++] = current->right;
        }
    }
}
```

Code Snippet 3: Breadth-First search

Note, the size of the queue (taken as 100 here) should in general be a function of the size of the tree. You can use a *better* queue implementation based on the techniques discussed in Week 3.

Task 2: Observe the insertAVL() function as implemented in the snippet. What is the time complexity of inserting one node into the AVL tree? It looks O(height) right? However, it is not. If you go through the code carefully, you would observe that each iteration of insertAVL() makes a call to is_height_balanced(), which itself has a complexity of O(height).

This results in a complexity of O($\log^2$n) which is not optimal. You can reduce this complexity by storing some additional information about the height in each node.

Add a **`height`** parameter to your bst node structure and modify the `insertAVL()` function so that it now updates this parameter and reduces the running time from `O(log`^{2}`n)` to `O(log n)`. Consider the height of leaf nodes to be **1**. (Yes, this is another example of the space-time complexity tradeoff).) You can also use a `balance_factor()` function as a sub-routine that calculates the balance factor of a node as the `left_height - right_height`.

Home Exercise 2: We saw the traverse_bfs() function in Task 1. This function is called breadth-first (or level order) because it prints all the nodes of each level before proceeding to the next. An another way of traversing the tree is called depth-first traversal. In this, one path upto the leafis completely traversed before you proceed along another path. All the walks we saw last week - in-order, pre-order and post-order, are versions of DFS.

You had implemented these walks last week. Now complete the program provided in the file *hw2.c* that constructs a binary tree given the in-order and post-order traversals.

Assume you are given three lines of input.
The first line contains the total number of nodes
The second line contains the in-order walk
The third line contains the post-order walk

Print the bfs output of the resulting tree on stdout.

Sample Input 1

```
8
4 8 2 5 1 6 3 7
8 4 5 2 6 7 3 1
```

Sample Output 1

```
1 2 3 4 5 6 7 8
```

Sample Input 2

```
9
1 3 4 5 6 7 8 9 10
1 3 5 7 6 10 9 8 4
```

Sample Output 2

```
4 3 8 1 6 9 5 7 10
```

Deletion from an AVL Tree

Similar to insertion, in deletion as well we delete the node normally from the BST and then move upwards fixing the tree where needed. In this section, we will consider that the tree has been augmented with the height parameter as described in Task 2. After deleting a node, we move upwards, checking the balance_factor at each level. Based on the balance_factor, we can determine if the tree is still balanced or not. If the balance_factor is -2 or 2, then we need to perform a rotation to restore the balance. There are four possible cases of imbalances: left-left, left-right, right-left and right-right.

In order to find which case of rotation needs to be performed, we check for the longer child of the current node, and its longer child. Based on their locations (left or right), we can identify the kind of imbalance present.

The implementation of deleteAVL() is given below in code snippet 4. Study it carefully to understand the motivation.

```
Node *deleteAVL(Node *node, int value)
{
  if (node == NULL)
  {
    // value not found
    return NULL;
  }
  else if (value < node->value)
  {
    // value is in left subtree
    node->left = deleteAVL(node->left, value);
    node->height = 1 + height(node->left) > height(node->right) ?
height(node->left) : height(node->right);
  }
  else if (value > node->value)
  {
    // value is in right subtree
    node->right = deleteAVL(node->right, value);
    node->height = 1 + height(node->left) > height(node->right) ?
height(node->left) : height(node->right);
  }
  else
  {
    // value is at this node
    if (node->left == NULL && node->right == NULL)
    {
      // node is a leaf
      free(node);
      node = NULL;
    }
    else if (node->left == NULL)
    {
      // node has only right child
      struct node *temp = node;
      node = node->right;
      free(temp);
    }
    else if (node->right == NULL)
    {
      // node has only left child
      struct node *temp = node;
      node = node->left;
      free(temp);
    }
    else
    {
      // node has both children
      struct node *temp = predecessor(node);
      node->value = temp->value;
      node->left = deleteAVL(node->left, temp->value);
      node->height = 1 + height(node->left) > height(node->right) ?
height(node->left) : height(node->right);
    }
  }

  int balance = balance_factor(node);
  // balance factor is left height - right height

  if (balance > 1)
  {
    // left subtree is longer
    if (balance_factor(node->left) >= 0)
    {
      // LL imbalance
      node = rotate_right(node);
      node->right->height = 1 + (height(node->right->left) > height(node->right->right) ? height(node->right->left) : height(node->right->right));
      node->height = 1 + (height(node->left) > height(node->right) ? height(node->left) : height(node->right));
    }
    else
    {
      // LR imbalance
      node->left = rotate_left(node->left);
      node->left->left->height = 1 + (height(node->left->left->left) > height(node->left->left->right) ? height(node->left->left->left) : height(node->left->left->right));
      node->left->height = 1 + (height(node->left->left) > height(node->left->right) ? height(node->left->left) : height(node->left->right));
      node = rotate_right(node);
      node->right->height = 1 + (height(node->right->left) > height(node->right->right) ? height(node->right->left) : height(node->right->right));
      node->height = 1 + (height(node->left) > height(node->right) ? height(node->left) : height(node->right));
    }
  }
  else if (balance < -1)
  {
    // right subtree is longer
    if (balance_factor(node->right) <= 0)
    {
      // RR imbalance
      node = rotate_left(node);
      node->left->height = 1 + (height(node->left->left) > height(node->left->right) ? height(node->left->left) : height(node->left->right));
      node->height = 1 + (height(node->left) > height(node->right) ? height(node->left) : height(node->right));
    }
    else
    {
      // RL imbalance
      node->right = rotate_right(node->right);
      node->right->right->height = 1 + (height(node->right->right->left) >
height(node->right->right->right) ? height(node->right->right->left) :
height(node->right->right->right));
      node->right->height = 1 + (height(node->right->left) > height(node-
>right->right) ? height(node->right->left) : height(node->right->right));
      node = rotate_left(node);
      node->left->height = 1 + (height(node->left->left) > height(node-
>left->right) ? height(node->left->left) : height(node->left->right));
      node->height = 1 + (height(node->left) > height(node->right) ?
height(node->left) : height(node->right));
    }
  }
  return node;
}
```

Code Snippet 4: AVL Tree Deletion

Home Exercise 3: Consider the `insertAVL()` and `deleteAVL()` functions above. Both functions are recursive. Write a non-recursive version of `insertAVL()` and `deleteAVL()` that uses a stack to store the nodes that are visited during the traversal. The stack should be implemented using a linked list. The stack should store the addresses of the nodes that are visited during the traversal. The non-recursive versions of `insertAVL()` and `deleteAVL()` should have the same functionality as the recursive versions of `insertAVL()` and `deleteAVL()`.

Task 3: Add a **`parent`** attribute to the **`node`** structure. Now implement iterative versions of `insertAVL()` and `deleteAVL()` without using the stack.

In comparison with the implementation in Home Exercise 3 above using the stack, which iterative version had a larger space overhead? Which was faster?

Building an AVL Tree using the Restructure Algorithm

Algorithm restructure(x):
Input: A node x of a binary search tree 7 that has both a parent y and a grand-parent z
Output: Tree T after a trinode restructuring (which corresponds to a single or double rotation) involving nodes x, y, and z

1: Let (a,b,c) be a left-to-right (inorder) listing of the nodes x, y, and z, and let (To, T1, T2, T3) be a left-to-right (inorder) listing of the four subtrees of x, y, and z not rooted at x, y, or z.
2: Replace the subtree rooted at z with a new subtree rooted at b.
3: Let a be the left child of b and let To and T_1 be the left and right subtrees of a, respectively.
4: Let c be the right child of b and let T_2 and T3 be the left and right subtrees of c, respectively.

Task 4: Construct an AVL Tree that maintains balance after insertions using the restructure(x) algorithm for trinode restructuring.

Introduction to (2,4) Trees

Now we move from binary search trees to multiway search trees. A multiway search tree has more than two children. Obviously, as we keep increasing the width of the tree, we can have shorter trees which might suggest lower complexity of search, insert, delete and other operations. However, with the time we save in height, we end up losing in width as the complexity of looking for the correct subtree at each level increases. Here in particular we consider (2, 4) trees. A (2, 4) tree has the following property:

- Every node has at least 2 keys
- Every internal node with k keys has k+1 child nodes
- Every node has at most 3 keys (and 4 children)
- The elements in each node are sorted
- All leaf nodes are at the same level

If we compare (2,4) trees with AVL trees, while both of them ensure logarithmic depth, (2,4) trees are in a sense less strict with respect to their balancing criteria than AVL trees.

Insertion

In insertion, first we find the appropriate leaf node. Then the insertion operation is always performed on this leaf node. Whenever we encounter a 4 node (node having 4 children) while traversing towards the leaf, we split preemptively it so that we can avoid the traversal back to the root after insertion.

The file *two_four.c* has been shared with you that contains the code for insertion. Let us try to understand it in parts.

After defining the struct node and the tree, we begin the function insert_24()

```
void insert_24(Tree *tree, int val)
```

1) In case the tree is empty, create a root node and add the element there:

```
{
    Node *temp = tree->root;
    if (node == NULL)
    {
        Node *myNode = new_node();
        myNode->isLeaf = 1;
        myNode->keys[0] = val;
        myNode->num_keys = 1;
        tree->root = myNode;
        return;
    }
```

2) Otherwise, we traverse up to the leaf, splitting every 4-node that we encounter

```
    Node *parent = NULL;
    while (temp)
    {
        if (temp->num_keys == 3)
        {
            // split 4-node
            Node *newNode = new_node();
            newNode->isLeaf = temp->isLeaf;
            newNode->children[0] = temp->children[2];
            newNode->children[1] = temp->children[3];
            newNode->keys[0] = temp->keys[2];
            newNode->num_keys = 1;
            temp->children[2] = NULL;
            temp->children[3] = NULL;
            temp->num_keys = 1;
```

2.i) Now we insert the new node into the parent

```
            if (parent == NULL)
            {
                parent = new_node();
                parent->isLeaf = 0;
                parent->children[0] = temp;
                parent->children[1] = newNode;
                parent->keys[0] = temp->keys[1];
                parent->num_keys = 1;
                tree->root = parent;
                printf("Created new root node\n");
            }
            else
           // The parent must have 1 or 2 keys since all 4-nodes are split
            {
                if (parent->num_keys == 1)
                // {...}
                // 2 keys in the parent
                else
                // {...}
```

3) Now we update temp and parent for the next iteration

```
            // Find the correct parent and child for the next iteration
            for (int i = 0; i <= parent->num_keys; i++)
            {
                // First find the correct parent. If you reach the end of
the keys, the child is the last child
                // Else, if the value is less than the key, the child is
the current child
                if (i == parent->num_keys || val < parent->keys[i])
                {
                    parent = parent->children[i];
                    // Now that we have the correct parent, find the correct
child

                    for (int j = 0; j <= parent->num_keys; j++)
                    {
                        // Same logic as above. Now update temp
                            {...}
                    }
                    break;
                }
            }
        }
```

4) If not a 4-node, just traverse to the correct child and end while

```
        else
        {
            parent = temp;
            // Find the correct child for the next iteration
            for (int i = 0; i < parent->num_keys; i++)
            {
                if (val < parent->keys[i])
                {
                    temp = parent->children[i];
                    break;
                }
            }
            if (parent == temp)
            {
                temp = parent->children[parent->num_keys];
            }
        }
    }
```

5) After the while loop, just insert into the correct leaf node, now pointed to by `parent`

```
    // Insert the value into the leaf
    for (int i = parent->num_keys - 1; i >= 0; i--)
    {
        if (val < parent->keys[i])
        {
            parent->keys[i + 1] = parent->keys[i];
        }
        else
        {
            parent->keys[i + 1] = val;
            break;
        }
        if (i == 0)
        {
            parent->keys[i] = val;
        }
    }
    parent->num_keys++;
}
```

Thus, in this way, we can ensure that all nodes are at the same level in a two-four tree as well as ensuring the depth of the tree is O(`log(N))`.

Task 5: Write a function to search for a given value in a (2, 4) tree. While searching in both (2,4) trees and AVL trees are `O(log N)`, which is faster? Construct an AVL tree and a (2,4) tree with integers from 1 to 100000. Now perform 5000 random searches of numbers in this range on each tree and compare the avg time taken in each tree.

Home Exercise 4: You have a database of people called *people.csv* with the following fields:

- name
- user_id
- salary
- age

Read the file and populate the data in an array.
Now, imagine you know you are going to receive many queries to retrieve records from the array using both the user_id and the salary. In case you sort it based on one of the fields for faster retrieval via binary search or store it in a tree, querying via the other field would be costly. Also, it is not practical to store two copies of the records sorted on different fields as you want changes made by one to be reflected in the database. To solve this problem we can create indices for the data.

An index is a data structure that allows you to search for a specific value in a database in O(log n) time. For each key in the index, you store a pointer to the record in the database that has that key. Here, the pointer is the index of the record in the array.

You can use two (2,4) trees as indices for the user_id and salary fields. You can assume the both these fields have unique entries in the file. Thus you can search for people by their user_id or salary in `O(log n)` time using these trees.

The user_id tree is made up of the following node:
```
struct user_id_pointer
{
    int user_id;
    int index;
};
struct user_id_node
{
    int num_keys;
    struct user_id_pointer keys[3];
    int children[4];
    int isLeaf;
};
```

Similarly, define a salary_node and construct (2,4) trees for both of them.

Task 6: We saw insertion into (2,4) trees above. Deletion from (2,4) trees follows a similar spirit of preemptively adjusting the upper nodes as we move below in order to maintain the property of the tree. While we were splitting near full nodes above, we would be merging nodes having space here. Similar to insertion you need to traverse up to the appropriate node containing the key to be deleted, performing the following steps:

1. If the element, k is in the node and the node is a leaf containing at least 2 keys, simply remove k from the node.
2. If the element, *k* is in the node and the node is an *internal node* perform *one* of the following:
    a. If the element's left child has at least 2 keys, replace the element with its predecessor, p, and then recursively delete p.
    b. If the element's right child has at least 2 keys, replace the element with its successor, s, and then recursively delete s.
    c. If both children have only 1 key (the minimum), merge the right child into the left child and include the element, k, in the left child. Free the right child and recursively delete k from the left child.
3. If the element, k, is not in the internal node, follow the proper link to find k. To ensure that all nodes we travel through will have at least 2 keys, you may need to perform one of the following before descending into a node. Then, you will descend into the corresponding node. Eventually, case 1 or 2 will be arrived at (if k is in the tree).
    a. If the child node (the one being descending into) has only 1 key and has an immediate sibling with at least 2 keys, move an element down from the parent into the child and move an element from the sibling into the parent.
    b. If both the child node and its immediate siblings have only 1 key each, merge the child node with one of the siblings and move an element down from the parent into the merged node. This element will be the middle element in the node. Free the node whose elements were merged into the other node.

You can refer to this for an illustration of the above algorithm.[3]

Implement deletion in 2-4 trees following the above cases.

[3] Matt Klassen's lecture notes on 2-3-4 trees