Construct BST from given sorted arrays

Given a sorted array. Write a function that creates a Balanced Binary Search Tree using array elements.

Examples:
Input: *arr[] = {1, 2, 3}*
Output: *A Balanced BST*

```
   2
  / \
 1   3
```

Explanation: *all elements less than 2 are on the left side of* ***2*** *, and all the elements greater than* ***2*** *are on the right side*
Input: *arr[] = {1, 2, 3, 4}*
Output: *A Balanced BST*

```
     3
    / \
   2   4
  /
 1
```

Sorted Array to Balanced BST By Finding The middle element
The idea is to find the middle element of the array and make it the root of the tree, then perform the same operation on the left subarray for the root's left child and the same operation on the right subarray for the root's right child.

Follow the steps mentioned below to implement the approach:

- Set The middle element of the array as root.
- Recursively do the same for the left half and right half.
    - Get the middle of the left half and make it the left child of the root created in step 1.
    - Get the middle of the right half and make it the right child of the root created in step 1.
- Print the preorder of the tree.

Below is the implementation of the above approach:

C++

```
// C++ program to print BST in given range

#include <bits/stdc++.h>

using namespace std;


/* A Binary Tree node */

class TNode {

public:

    int data;

    TNode* left;

    TNode* right;

};


TNode* newNode(int data);


/* A function that constructs Balanced

Binary Search Tree from a sorted array */

TNode* sortedArrayToBST(int arr[], int start, int end)

{

    /* Base Case */

    if (start > end)

        return NULL;


    /* Get the middle element and make it root */

    int mid = (start + end) / 2;

    TNode* root = newNode(arr[mid]);


    /* Recursively construct the left subtree
    and make it left child of root */
    root->left = sortedArrayToBST(arr, start, mid - 1);

    /* Recursively construct the right subtree
    and make it right child of root */
    root->right = sortedArrayToBST(arr, mid + 1, end);

    return root;
}

/* Helper function that allocates a new node
with the given data and NULL left and right
pointers. */
TNode* newNode(int data)
{
    TNode* node = new TNode();
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return node;
}

/* A utility function to print
preorder traversal of BST */
void preOrder(TNode* node)
{
    if (node == NULL)

        return;

    cout << node->data << " ";

    preOrder(node->left);

    preOrder(node->right);

}


// Driver Code

int main()

{

    int arr[] = { 1, 2, 3, 4, 5, 6, 7 };

    int n = sizeof(arr) / sizeof(arr[0]);


    /* Convert List to BST */

    TNode* root = sortedArrayToBST(arr, 0, n - 1);

    cout << "PreOrder Traversal of constructed BST \n";

    preOrder(root);


    return 0;

}
```

Output

```
PreOrder Traversal of constructed BST

4 2 1 3 6 5 7
```

Time Complexity: O(N)
Auxiliary Space: O(H) ~= O(log(N)), for recursive stack space where H is the height of the tree.

--

Approach 2: Using queue – Iterative Approach

1. First initialize a queue with root node and loop until the queue is empty.
2. Remove first node from the queue and find mid element of the sorted array.
3. Create new node with previously find middle node and set left child to the deque node left child if present and also set the right child with deque node right child. Enqueue the new node onto the queue. Set the right child of the dequeued node to the middle element on the left side of the sorted array. If the dequeued node has a left child, enqueue it onto the queue. Return the root node.

Below is the implementation of the above approach:

C++

```cpp
#include <iostream>

#include <queue>

#include <vector>


using namespace std;


// structure of the tree node

struct Node {

    int val;

    Node* left;

    Node* right;

};


// function to convert the array to BST

// and return the root of the created tree

Node* sortedArrayToBST(vector<int>& nums)

{

    // if the array is empty return NULL
```

```
if (nums.empty()) {
    return NULL;
}

int n = nums.size();
int mid = n / 2;
Node* root = new Node{ nums[mid], NULL, NULL };
// initializing queue
queue<pair<Node*, pair<int, int> > > q;
// push the root and its indices to the queue
q.push({ root, { 0, mid - 1 } });
q.push({ root, { mid + 1, n - 1 } });

while (!q.empty()) {
    // get the front element from the queue
    pair<Node*, pair<int, int> > curr = q.front();
    q.pop();

    // get the parent node and its indices
    Node* parent = curr.first;
    int left = curr.second.first;
    int right = curr.second.second;

    // if there are elements to process and parent node
    // is not NULL
    if (left <= right && parent != NULL) {
        int mid = (left + right) / 2;
```

```
            Node* child = new Node{ nums[mid], NULL, NULL };

            // set the child node as left or right child of
            // the parent node
            if (nums[mid] < parent->val) {
                parent->left = child;
            }
            else {
                parent->right = child;
            }

            // push the left and right child and their
            // indices to the queue
            q.push({ child, { left, mid - 1 } });
            q.push({ child, { mid + 1, right } });
        }
    }

    return root;
}

// function to print the preorder traversal
// of the constructed BST
void printBST(Node* root)
{
    if (root == NULL)
        return;
    cout << root->val << " ";

    printBST(root->left);

    printBST(root->right);

}


// Driver program to test the above function

int main()

{

    // create a sorted array

    vector<int> nums = { 1, 2, 3, 4, 5, 6, 7 };

    // construct the BST from the array

    Node* root = sortedArrayToBST(nums);

    // print the preorder traversal of the BST

    printBST(root); // Output: 4 2 1 3 6 5 7

    return 0;

}
```

Output

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

Time Complexity: O(N), where N is the number of elements in array.
Auxiliary Space: O(N)