Lab 3 - Binary Search Tree

Tree

A tree is a non-linear data structure where the data are organized in a hierarchical manner.
A tree is a finite set of one or more nodes such that

- There is a specially designated node called the root.
- The remaining nodes are partitioned into n ≥ 0 disjoint sets T1, T2, . . . , Tn, where each of these sets is a tree. T1, T2, . . . , Tn are called the subtrees of the root.

Binary Search Tree (BST)

A binary search tree (BST) is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

- Each node has exactly one key and the keys in the tree are distinct.
- The keys (if any) in the left subtree are smaller than the key in the root.
- The keys (if any) in the right subtree are larger than the key in the root.
- The left and right subtrees are also binary search trees.

Functions:

(a) isEmpty():

- Returns true if the tree is empty, and false otherwise

```
// Check if the tree is empty
bool ArrayBinarySearchTree::isEmpty()
{
    if (data_user == nullptr || nodeCount == 0)
    {
        return true;
    }
    return false;
}
```

(b) addBST(data):

- Inserts an element to the BST

```
// Add the given element to the tree
void ArrayBinarySearchTree::addBST(int data)
{
    if (data_user == nullptr || isFull())
    {
        if (!resize())
        {
            throw "Error allocating space";
        }
    }
    nodeCount++;
    if (nodeCount == 0)
    {
        data_user[0] = new ArayNode;
        data_user[0]->data = data;
        depth = 0;
        return;
    }
    depth = static_cast<int>(log2(nodeCount));

    int i{1};
    while (i <= size)
    {
        int left = 2 * i;
        int right = left + 1;
        if (left >= size)
        {
            resize();
        }
        if (data_user[left] == nullptr)
        {
            data_user[left] = new ArayNode;
            data_user[left]->data = data;
            std::cout << "Added to " << left << std::endl;
            return;
        }
        if (right >= size)
        {
            resize();
        }
        if (data_user[right] == nullptr)
        {
            data_user[right] = new ArayNode;
            data_user[right]->data = data;
            std::cout << "Added to " << right << std::endl;
            return;
        }
        i++;
    }
}
```

Lab 3 - Binary Search Tree

(c) removeBST(keyToDelete):

- Removes the node with the given key from the BST

```
// Remove the given element from the tree
void ArrayBinarySearchTree::removeBST(int key)
{
    if (isEmpty())
    {
        throw "Empty tree cannot be removed";
    }
    int i;
    // Find the index of the node with the given key
    for (i = 0; i < size; i++)
    {
        if (data_user[i] && data_user[i]->data == key)
        {
            break;
        }
    }

    if (i < size && data_user[i])
    {
        // Replace the node to be removed with the last node
        auto temp = data_user[size - 1];
        data_user[size - 1] = data_user[i];
        data_user[i] = temp;

        // Delete the last node
        delete data_user[size - 1];
        data_user[size - 1] = nullptr;
        nodeCount--;

        // Re-balance the tree
        heapify(i);
    }
    throw "Key not found";
}
```

(c) searchBST(targetKey):

- Returns true if the key exists in the tree, and false otherwise

```
// Search for the given element in the tree
int ArrayBinarySearchTree::searchBST(int key)
{
    if (isEmpty())
    {
        throw "Empty tree cannot be searched";
    }
    int left = 0;
    int right = size - 1;
    // Binary search using iterative approach
    while (left <= right)
    {
        int mid = left + (right - left) / 2;

        if (data_user[mid] != nullptr && data_user[mid]->data == key) // Key found
        {
            return mid;
        }
        else if (data_user[mid] != nullptr && data_user[mid]->data < key) // Search in the right subtree
        {
            left = mid + 1;
        }
        else // Search in the left subtree
        {
            right = mid - 1;
        }
    }

    return -1; // Key not found
}
```

Lab 3 - Binary Search Tree

Main Functions and it's output:

Main:

```
void Main_for_ArrayBst()
{
    ArrayBinarySearchTree bst;
    std::cout << "Checking if this is empty: " << std::boolalpha << bst.isEmpty() << std::endl;
    bst.addBST(0);
    bst.addBST(1);
    bst.addBST(2);
    bst.addBST(3);

    std::cout << "Checking if this is empty: " << std::boolalpha << bst.isEmpty() << std::endl;
    std::cout << "Searching for 2: " << bst.searchBST(2) << std::endl;
    std::cout << "Searching for 3: " << bst.searchBST(3) << std::endl;
    std::cout << "Searching for 4: " << bst.searchBST(4) << std::endl;
    try
    {
        bst.removeBST(2);
    }
    catch (const char *e)

    {
        std::cerr << e << '\n';
    }
    try
    {
        std::cout << "Searching for 2(after removing): " << bst.searchBST(2) << std::endl;
        bst.~ArrayBinarySearchTree();
    }
    catch (const char *e)
    {
        std::cerr << e << '\n';
    }

    try
    {

        std::cout << "Checking if this is empty(after emptying): " << std::boolalpha << bst.isEmpty() << std::endl;
    }
    catch (const char *e)
    {
        std::cerr << e << '\n';
    }
    std::cout << "trying to search after clearing the tree: " << std::endl;
    try
    {
        bst.searchBST(2);
    }
    catch (const char *e)
    {
        std::cerr << e << '\n';
    }
    std::cout << "trying to remove after clearing the tree: " << std::endl;
    try
    {
        bst.removeBST(2);
    }
    catch (const char *e)
    {
        std::cerr << e << '\n';
    }
    std::cout << "End of Array BST" << std::endl;
    return;
}
```

:

Lab 3 - Binary Search Tree

Output:

```
Checking if this is empty: true
Added to 2
Added to 3
Added to 4
Added to 5
Checking if this is empty: false
Searching for 2: 4
Searching for 3: 5
Searching for 4: -1
Key not found
Searching for 2(after removing): -1
Checking if this is empty(after emptying): true
trying to search after clearing the tree:
Empty tree cannot be searched
trying to remove after clearing the tree:
Empty tree cannot be removed
End of Array BST
```

Other important functions:

Heapify(index):

- This function will take an index and maintain the heap structure

```
void ArrayBinarySearchTree::heapify(int index)
{

    // Implement heapify to maintain the binary search tree property
    int left = 2 * index + 1;
    int right = 2 * index + 2;
    int smallest = index;

    if (left < size && data_user[left] && data_user[smallest] && data_user[left]->data < data_user[smallest]->data)
    {
        smallest = left;
    }

    if (right < size && data_user[right] && data_user[smallest] && data_user[right]->data < data_user[smallest]->data)
    {
        smallest = right;
    }

    if (smallest != index)
    {
        auto temp = data_user[index];
        data_user[index] = data_user[smallest];
        data_user[smallest] = temp;
        heapify(smallest);
    }
}
```

~ArrayBinarySearchTree()

- This function will remove all the elements stored in the “data_user”

```
~ArrayBinarySearchTree()
{
    for (int i = 0; i < size; i++)
    {
        delete data_user[i];
    }
    delete[] data_user;
    data_user = nullptr;
}
```

Lab 3 - Binary Search Tree

Resize():

- This function will resize the array that is storing the data.

```
// Helper function to resize the array
bool ArrayBinarySearchTree::resize()
{
    return resize(0);
}

// main function to resize the array
bool ArrayBinarySearchTree::resize(int temp_size)
{
    ArayNode *temp = nullptr; // Temporary pointer to store the data
    try
    {
        if (data_user == nullptr)
        {
            data_user = new ArayNode *[10]; // Initial size of the array
            size = 10;                      // Storing size of the array
            for (int i = 0; i < 10; i++)
            {
                data_user[i] = nullptr; // Initialize all the pointers to nullptr
            }
            return true;
        }
        if (temp_size == 0) // If the size is not specified
        {
            temp_size = pow(2, depth) == 1 ? 1 : pow(2, depth + 1) + 1; // Calculate the size of the array
            depth++;                                                    // Increase the depth
            data_user = new ArayNode *[temp_size];                      // Create a new array with the new size
            for (int i = 0; i < temp_size; i++)
            {
                data_user[i] = nullptr; // Initialize all the pointers to nullptr
            }
            size = temp_size;
            return true;
        }
        temp = new ArayNode[temp_size * 2];
        int i;
        for (i = 0; i < size; i++)
        {
            temp[i] = *data_user[i]; // Copy the data to the temporary pointer array
        }
        for (i = 0; i < size; i++)
        {
            delete data_user[i]; // Delete the data from the original array
        }
        delete[] data_user;

        data_user = new ArayNode *[temp_size * 2];
        for (int i = 0; i < temp_size; i++) // re save the data
        {
            data_user[i] = &temp[i];
        }
        size *= 2;
        delete[] temp;
        return true;
    }
    catch (std::exception &e)
    {
        std::cout << e.what() << std::endl;
        return false;
    }
}
```

Note:

- This is just the Array implementation of the binary search tree since my roll number is odd (27).