Binary Search Trees

Kilaru Yasaswi Sri Chandra Gandhi[*]

November 19, 2019

Contents

1 Introduction

Hi everyone, today we shall be looking at a popular data structure that you've probably seen or heard about before. It is called the *Binary Search Tree* and it builds on a fundamental divide and conquer algorithm, namely binary search that all of us are familiar with. The binary search algorithm is associated with a data structure called the BST, the Binary Search Tree. Let's dive in.

[*]Indian Institute of Technology, Roorkee (Website)

1.1 Motivating Binary Search Trees

All data structures were invented to solve problems and the same goes with BST's. Let us look at binary search trees by first considering a popular problem whose solution is best achieved by means of a BST. It is a problem that exists in all sorts of scheduling problems. What we'll do after defining this problem and talking about how we could possibly solve it with the data structures you already know like arrays, lists and also heaps and hopefully motivate you into the reason behind the existence of binary search trees, because they are kind of the perfect data structure for this particular problem. So let's dive into what the runway reservation system looks like. And it's your basic scheduling problem.

1.1.1 Runway Reservation System Problem

Assume we have an airport that has a single runway. We can understand that this runway would be pretty busy, and there would be safety issues associated with landing planes, and planes taking off. And so there are constraints associated with the system, that have to be obeyed. And you have to build these constraints in– and the checks for these constraints– into your data structure.

The problem statement here is that each plane will have a landing time at 't' minutes and we will have to fit this new number 't' into the already existing set of times. Let's call this set R. And we will need to make sure that no two planes land within less than 'k' minutes of each other for safety reasons.

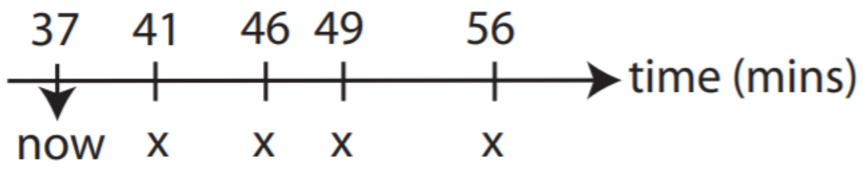

Figure 1: Set of scheduled times

In figure 1, the set $R = (41, 46, 49, 56)$ gives the queue and $k = 3$ is the intermediate time. Let us look at some valid and invalid time requests:

- 44 is not allowed because $46 \in R$ and $|46 - 44| < k$.
- 53 is allowed as $|56 - 53| = k$.
- 20 is not allowed as it is in the past.

So this is about adding to the data structure. And so an insert operation, if you will, that has a constraint associated with it that you need to check. Similarly we delete every entry after the plane has landed, so on and so forth.

1.1.2 Solving using unsorted arrays

Question: Let's say you have an unsorted list or an array corresponding to R. That's all you have. What's wrong with this data structure from an efficiency standpoint?

Answer: Everything that we will have to will take $O(n)$ time. Let us see how that happens: Adding to an unsorted array takes $O(1)$ time but performing the 'k' minute check in an unsorted array will require us to scan through the entire array.

1.1.3 Solving using sorted arrays

Question: What will happen if we have a sorted array corresponding to R? Think of both the k-minute check and insertion.

Answer: All of you who thought of binary search to perform the k minute check, you were right. So, within logarithmic time, we manage to perform the check. But we still have to perform insertion.

Question: How long will insertion take in a sorted array provided that our k-minute check has been passed?

Answer: $O(n)$ is the correct answer. Here, we are supposed to shift every single element by 1 everytime we have to insert. Hence the linear runtime.

1.1.4 Solving using lists

Question: Now you could imagine that you had a sorted list, if you have a list, it's got a bunch of pointers in it and if we've found the insertion point, we can insert by moving pointers in constant time. But what's the problem with the list?

Answer: You cannot do binary search on a list, there's no notion of going to the $\frac{n}{2}$ index and doing random access on a conventional list.

1.1.5 The BST solution

After all these approaches, I imagine you can appreciate the issues with solving what seems to be a trivial job for humans. And we saw that that the sorted array came pretty close except for the insertion. So, if we can solve insertion in $O(\log(n))$ time, we are good to go.

1.2 Tree Terminology

1.2.1 Generic Tree Structure and Terms

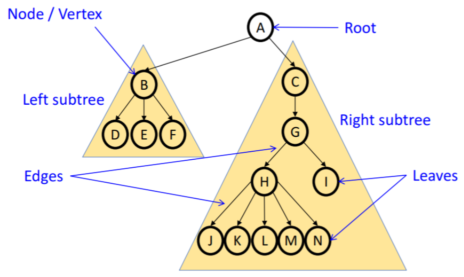

Figure 2: Generic tree terms

In figure 2 we see a generic tree. All the points are called nodes/vertices and lines joining nodes are called edges. The topmost node is called a root and the bottom-most set of nodes are called leaves. Each branch is called a sub-tree and in figure 2 we see a left sub-tree and a right sub-tree.

1.2.2 Binary Tree

A binary tree is a special kind of tree where every node has at most two children (branching factor not greater than 2). In more technical terms, a binary tree is a root that *has* some data in it with a left sub-tree (which may be empty) and a right sub-tree (which also may be empty).

Figure 3 shows a typical binary tree, we can see that the root has data, the left most node is empty, the node with '6' in it is complete and the right most node only has a left sub-tree.

1.2.3 Tree Traversals

A **traversal** is an order for visiting all the nodes of a tree.

- **Pre-order:** Root $\rightarrow$ Left Sub-tree $\rightarrow$ Right subtree

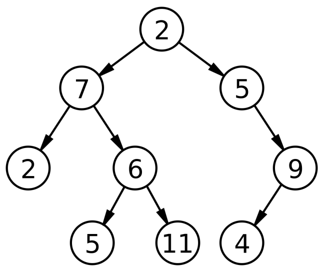

Figure 3: Binary Tree

- **In-order:** Left Sub-tree → Root → Right subtree
- **Post-order:** Left Sub-tree → Right Sub-tree → Root

2 Binary Search Trees

With binary search trees we try to obtain efficient insert and search times for the associated arrays as depicted by the scheduling problem explained before. We will show that by using an ordering invariant, we will be able to achieve $O(\log(n))$ worst-case asymptotic complexity for insert and search. This also extends to deletions.

2.1 The Ordering Invariant

First, we will look at a formal definition and then attempt to understand it using some examples. At the core of binary search trees is the *ordering invariant.*

Ordering Invariant: At any node with key k in a binary search tree, all keys of the elements in the left sub-tree are strictly less than k, while all keys of the elements in the right sub-tree are strictly greater than k.

2.1.1 Practice

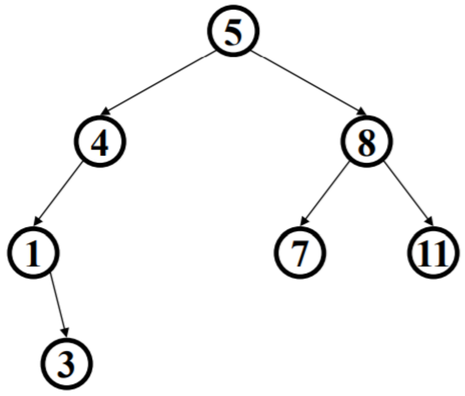

Figure 4: Binary Tree

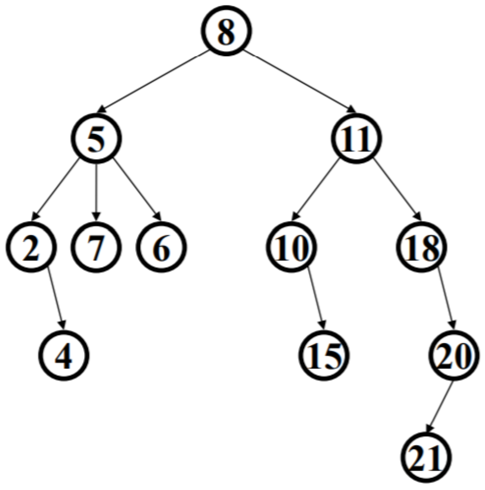

Figure 5: Not a Binary Tree

We can see that 4 does not violate the *Ordering Invariant* but figure 5 has multiple violations in node 5 (more than 2 children), node 20 & 11 (left sub-tree has an element greater than root)

2.1.2 Representation using pointers

Unlike heaps, we cannot easily represent binary search trees with arrays and keep them balanced. This is because heaps have a lot of flexibility where to insert new elements, while with binary search trees the position of the new element is determined very rigidly.

So we will use a pointer-based implementation where every node has two pointers: one to its left child and one to its right child. A missing child is represented as NULL, so a leaf just has two null pointers.

```
typedef struct tree* tree;
struct tree
{
    elem data;
    tree left;
    tree right;
};
```

Listing 1: C++ example

As usual, we have a *header* which in this case just consists of a pointer to the root of the tree. We often keep other information associated with the data structure in these headers, such as the size.

```
typedef struct tree* tree;
struct bst
{
    tree root;
};
```

Listing 2: C++ example

2.1.3 Runtime

Binary search trees support several operations, including Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete. These operations run in time proportional to the height of the tree. In the best case scenario, the tree is a complete binary tree, and the height of the tree is $\Theta(\log(n))$. In the worst case scenario, the tree is a linear chain, so the height of the tree is $\Theta(n)$.

2.1.4 Inorder tree walk

Question: Given a binary search tree, write a program that prints the keys in the binary search tree in sorted order.

Answer: The below function will give us the required order.

```
void printSorted(int arr[], int start, int end)
{
    if(start > end)
        return;

    // print left subtree
    printSorted(arr, start * 2 + 1, end);

    // print root
    cout << arr[start] << " ";

    // print right subtree
    printSorted(arr, start * 2 + 2, end);
}
```

Listing 3: C++ example

Runtime: This program runs in $\Theta(n)$ time. It visits all n nodes of the subtree, so $T(n) = \Omega(n)$.

2.2 Operations on Binary Search Trees

In the next few sections, we shall discuss the operations that we can perform on BSTs. As we already know the requirements of the data structures, most of us know the basic operations that we need to test out like searching, insertion and deletion. We shall also briefly look at some other trivial operations.

2.2.1 Search

Searching is quite straightforward, we initially compare the value of a node with the root. If equal, return true. If less, recursively look in the left-hand subtree. If greater, recursively look in the right-hand subtree.

```
boolean search(V, N)
{ // search for value V in the subtree with root N;
   if (N == null)
       return false;
   else if (V = N.value)
       return true;
   else if (V < N.value)
       return search(V, N.left);
   else
       return search(V, N.right);     // V > N.value
}
```

Listing 4: C++ example

2.2.2 Insertion

Suppose we want to add a value V, search for V in the tree as explained before. If V is there already, do nothing. If it is not, add it at the point where the search ran into a NULL pointer.

```
void add(V,N)
{
    if (V < N.value)
    {
        if (N.left != null)
            add(V, N.left)
        else
        {
            //create new node P for V;
            N.left = P;
        }
    }
    else if (V > N.value)
        if (N.right != null)
            add(V, N.right)
    else
    {
        //create new node P for V;
        N.right = P;
    }
}
```

Listing 5: C++ example

2.2.3 Deletion

Suppose we have to delete element V, find V using the search algorithm we looked at earlier. Now there are five possibilities:

- **Case 1:** If V is not in the tree, do not do anything
- **Case 2:** V is at node P which is a leaf. Delete P.
- **Case 3:** V is at an internal node P with only one child C. Move C into P's position.
- **Case 4:** V is at an internal node P with two children. Let Q be either the node immediately before P, which is the rightmost descendant of *P.left*, or the node immediately after P, which is the left-most descendant of *P.right*. In either case, Q does not have two children (if it did, it wouldn't be leftmost/rightmost) so either:
  - **Case 4a:** Q is a leaf. Move *Q.value* into *P.value*, and delete Q.
  - **Case 4b:** Q has one child C. This must be a left child, since Q is the rightmost descendent. Move *Q.value* into *P.value* and move C into Q's position.

I know that was a lot, but looking at the code with a simple BST at hand will solve most doubts.

```
void delete(V,N)
{
    if (N == null)
        return;                          // Case 1
    if (V  < N.value)
        delete(V, N.left);               // recursively go down the tree
    else if (V > N.value)
        delete(V, N.right)               // recursively go down the tree
    else                                 // V == N.value
        if (N.left == null && N.right == null)
            deleteLeaf(N);               // Case 2
    else if (N.left == null)             // Case 3
            replaceNode(N.right, N);
    else if (N.right == null)            // Case 3
            replaceNode(N.left, N);
    else
    {                                    // Case 4
        Q = N.left.rightmostDescendant();
        N.value = Q.value;
        if (Q is a leaf)                 // Case 4.A
            deleteLeaf(Q);
        else                             // Case 4.B
            replaceNode(Q.left, Q)
    }
}

void deleteLeaf(L)
{
    P = L.parent;
    if (L == P.left)
        L.left = null;
    else
        L.right = null;
}

void replaceNode(C,N)
{                                        // Replace N by C;
    P = N.parent;
    C.parent = P;
    if (N == P.left)
        P.left = C;
    else P.right = C;
}
```

Listing 6: C++ example

2.3 Rounding Up

Here, we shall discuss some minor trivialities that will complete our understanding of BSTs. These will mostly involve minor algorithms and runtime technicalities.

2.3.1 Minimum / Maximum

To find the minimum element in a tree, we start at the root, and take its left child. Then we look at the left child of that left child, etc. We keep going until we reach NIL, and then we return the last non-NIL node in the sequence.

Question: Write the implementation using the ideas covered in the lecture.

Question: Find worst-case runtime of both maximum and minimum operations

2.3.2 Successsor / Predecessor

The successor of a node is the smallest node greater than that node (when ordered by key). We can use the tree-walk covered in the beginning and check whether x has a right subtree or not.

Question: Write the implementation using the ideas covered in the lecture.

Question: Find worst-case runtime of both successor and predecessor operations.

2.3.3 Time Requirement

All three main operations (search, insert and delete) require climbing down from the root to a lower node in the tree. Therefore in the worst case they take time proportional to the height of the tree.

2.3.4 Height to number of elements relation

Question: How is the height of the tree H related to the number of elements N?

Answer: Well, that depends. In the best case, every internal node (except possibly one in second-to-last row) has two children. Then N is between 2^{H-1} and $2^H - 1$, so H is about $\log_2(N)$ plus or minus 1.

In the worst case, every internal node has exactly one child, and there is only one leaf in the tree. In that case the tree is essentially just a linked list and $H = N$.

3 Summary

So in this lecture, we started of by looking at a problem that would have extremely expensive to solve with arrays, lists and even heaps. We then looked at what we required to solve that problem efficiently and we came up with a tree structure that had binary search as its backbone.

Then we looked at what trees are and explained the structural concept of a binary tree and how each node cannot have more than 2 children. We then looked at the ordering invariant which is the formal definition of a BST. After that, we went through a simple walkthrough algorithm that allowed us to access every element. Walkthrough algorithms are extremely important when you're learning any data structure because you have to know how to access memory.

We then looked at representation using pointers after understanding why traditional approaches don't work. We also looked at a preliminary runtime analysis. We then dived into the technical algorithms and started with search which was easy, insertion and deletion took some understanding but after that, we now have a grasp over BST's.

In the final section, we looked at some trivial algorithms and some questions for homework practice. We looked at time requirements and a very important height to node relation.

At this point, we are very confident in our understanding of BSTs and we can move on to red-black trees and AVL trees. To get a much better understanding, solving problems and implementing all the algorithms on your own is recommended. We hope you had as much fun throughout this lecture as we did while preparing the material.