Binary Search Tree (I): Introduction

CSCD 300 – Data Structures

Eastern Washington University

Goal

We will learn the motivation and basic concepts of the binary search tree data structure.

Outline

1. The problems with the array, linked lists, and hash table data structure
2. The motivation for Binary Search Tree
3. The overview of Binary Search Tree

The problem with Arrays

- If all the elements are saved in ascending (or descending) order:
  - `Search` is fast and takes $O(\log n)$ time [1] by using the binary search.
  - `Insertion` is slow and take $O(n)$ time, because the new element needs to occupy a particular array location and all elements on and after that particular array location need to move to the next right array location.
  - `Delete` is slow and take $O(n)$ time, because all the elements that are after the array location where the deleted element occupied need to move to the next left array location.
- If all the elements are saved in any arbitrary order:
  - `Search` is slow and takes $O(n)$ time, because we have to linearly scan the whole array in the worst case.
  - `Insert` is fast and takes $O(1)$ time by simply inserting the new element right after the current rightmost element.
  - `Delete` is slow and takes $O(n)$ time, because we first need to search for the to-be-deleted element which takes $O(n)$ time, we can then move the rightmost element to the array location where an element is just deleted. Of course, if the to-be-deleted element is the rightmost element, we just simply delete it.

[1] We use n to denote the data set size.

The problem with Linked Lists

- If all the elements are saved in ascending (or descending) order:
  - `Search` is slow and takes $O(n)$ time, because we have to linearly scan the whole list in the worst case.
  - `Insertion` is slow and take $O(n)$ time, because we first need to find the right position to insert the new element, which takes $O(n)$ time.
  - `Delete` is slow and take $O(n)$ time, because we first need to find the to-be-deleted element, which takes $O(n)$ time.
- If all the elements are saved in any arbitrary order:
  - `Search` is slow and takes $O(n)$ time, because we have to linearly scan the whole list in the worst case.
  - `Insert` is fast and take $O(1)$ time, because we can just simply insert the new element at the head (or tail, if its reference is maintained) of the linked list.
  - `Delete` is slow and take $O(n)$ time, because we first need to find the to-be-deleted element, which takes $O(n)$ time.

The problem with Hash Tables

In average, the `search`, `insert`, and `delete` operations are fast and take $O(1)$ time, but the data set size must be well pre-determined and cannot grow to be infinitely large. Otherwise, the search performance of the hash table will become worse and worse.

You can feel free to think of any other disadvantages of the array, linked list, and hash table as a data structure for indexing a data set.

The motivation for Binary Search Tree

Question

Can we have a data structure, such that (1) the time costs of `search`, `insert`, and `delete` are all sublinear of the data set size, and (2) it supports a data set of any size as long as the memory capacity allows ?

Answer

Yes, one example of such data structures is the Binary Search Tree (BST).

- The time cost of `search`, `insert`, and `delete` operations of BST is still $O(n)$ in the worst case.
- However, in practice and for the real-world data, the time cost of those three operations will be $O(\log n)$.
- Further, balanced BSTs exist but are more complicated, which however guarantee $O(\log n)$ time cost for all the three operations even in the worst case.
- We will only discuss the basic BST in this course, and will discuss the the balanced BSTs in CSCD320.

The overview of Binary Search Tree

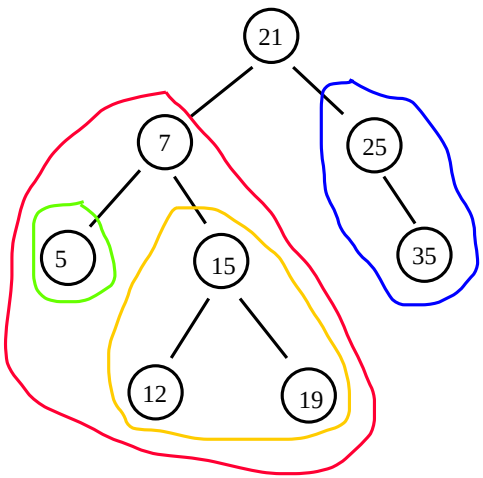
The BST can be viewed as a 2-d linked list, such that

- The overall topology of the BST is an upside-down tree.
- Each node has a data field, a left child link, a right child link, and also a parent link if needed in applications.
- Each node cannot have more than two children: zero child, one child, or two children.
- The key at the root node is larger than any key in the left subtree and is smaller than any key in the right subtree. This is also true for any subtree in the whole BST, meaning any subtree of a BST is also a BST.

An example follows ...

EASTERN WASHINGTON UNIVERSITY
start something big

An example and a few terminologies



An example BST

- Node 21's left subtree is the red subtree; Node 21's right subtree is the blue subtree. Similarly, node 7's left subtree is the green subtree; Node 7's right subtree is the yellow subtree.
- Node 21's left child is node 7, right child is node 25. Similarly, Node 25's left child is `null`, right child is node 35.
- Node 21 is called the root of the whole BST. Similarly, node 7 is the root of the red subtree.
- Node 21's grand children are nodes 5, 15, and 35.
- Node 15's parent is node 7. Node 21's parent is `null`.
- Any node who does not have any child node is called a leaf node in the tree. In this example, node 5, 12, 19, 35 are leaf nodes.
- Node 21's descendants are all those nodes in the subtree rooted on node 21 and are below node 21. Similarly, some other node in this example tree has descendants. Leaf nodes do not have descendants.
- Node 15's ancestors are all the nodes on the path from the global root toward node 15: node 21 and 7 in this example.
- Node 15's sibling is the node 5, who shares the same parent node.

The structure of each node

In order to connect all the nodes into such an upside-down tree structure, each node has to have at least four attributes:

```
class BST_Node{
   int key; //the key saved in the node. You can have any other fields
            //of any type within a node, depending on your applications.
   BST_Node parent; // reference to the parent node.
   BST_Node left;   // reference to the left child node.
   BST_Node right;  // reference to the right child node.

   /* constructor */
   BST_Node(int k){ key = k; parent = left = right = null; }

   /* other operations are here */
}
```

Note: the parent link may not be needed if the set of BST operations in your application do not need the parent links to travel upward inside the tree. However, for general purpose, we include the parent link here.

The Binary Search Tree

```
class BST{
   BST_Node root;   //the root of the BST

   /* You can add any other member fields needed by your app here */

   /* the constructor */
   BST(){ root = null;}


   /* Other methods will follow here  */
}
```

In the next lecture, we are going to use the BST node structure to construct a BST over a given data set. Essentially, we need to know how to insert a new key into a BST, which can be even empty. Then we can construct the BST for the given data set by inserting keys one by one.

EASTERN WASHINGTON UNIVERSITY
start something big