Engineering and Applied Science Programs for Professionals
Whiting School of Engineering
Johns Hopkins University
685.621 Algorithms for Data Science
Data Structures II - Binary Search Trees

This document provides a rollup of data structures and the use in the CS which is heavily used in Data Science. Search trees are data structures that support many dynamic-set operations which can be used as both a dictionary and as a priority queue. The basic operations take time proportional to the height of the tree, for example a complete binary tree with n nodes the worst case $\Theta(\ln(n))$ and for linear chain of n nodes the worst case $\Theta(n)$. The different types of search trees include binary search trees (covered in Chapter 12), red-black trees (covered in Chapter 13), and B-trees (covered in Chapter 18). We will cover binary search trees, tree walks, and operations on binary search trees.

Contents

Throughout this course, we will use a wide variety of trees to solve problems with the algorithms we study. One of the more commonly used tree structures is the binary search tree. A binary search tree is a tree structure where the keys are rooted in the tree in such a way to facilitate easy search. For this to happen, the binary search tree imposes a particular property on the orientation of the key values in the tree.

1 Binary Search Trees

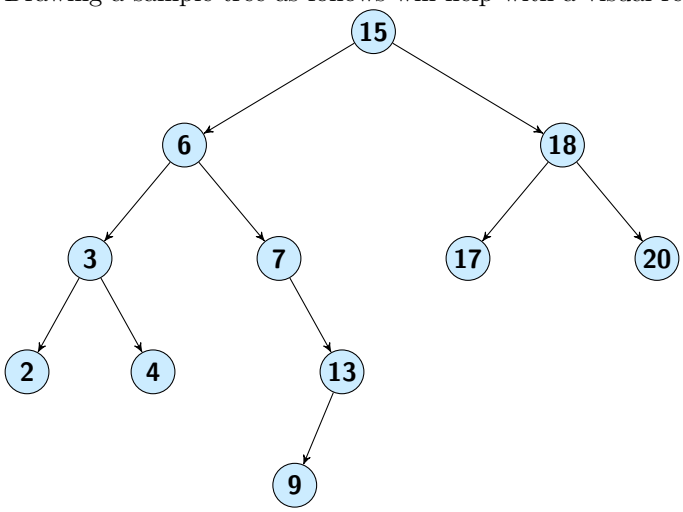
Binary search trees are an important data structure for dynamic sets. The binary search tree property is a property imposed on every vertex of a binary search tree such that the following conditions hold.

- Accomplish many dynamic-set operations in $O(h)$ time, where h = height of tree.
- A binary tree is represented by a linked data structure in which
- $T.root$ points to the root of tree T.
- Each node contains the attributes
  - $T.key$ (and possibly other satellite data).
  - $T.left$ points to left child.
  - $T.right$ points to right child.
  - $T.p$ points to parent. $T.root.p = NIL$.

Stored keys must satisfy the binary-search-tree property.

- If y is located in the left subtree of x, then $y.key \leq x.key$.
- If y is located in the right subtree of x, then $y.key \geq x.key$.

Drawing a sample tree as follows will help with a visual representation.



This is Figure 12.1(a) from the text. It's OK to have duplicate keys, though there are none in this example. For the sake of discussion, assume all key values are distinct (although, in general, this is not required). To show that the binary-search-tree property holds we can use the inorder tree traversal algorithm as will be seen in the Tree Traversal section.

2 Tree Traversal

The binary-search-tree property allows us to print keys in a binary search tree in order, recursively, using an algorithm called an inorder tree walk. Elements are printed in monotonically increasing order.

How INORDER-TREE-WALK works:

- Check to make sure that x is not NIL.
- Recursively, print the keys of the nodes in x's left subtree.
- Print x's key.
- Recursively, print the keys of the nodes in x's right subtree.

INORDER-TREE-WALK(x)

```
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print(key[x])
4      INORDER-TREE-WALKK(x.right)
```

Example
Do the inorder tree walk on the example above, getting the output 2-3-4-6-7-9-13-15-17-18-20.

Correctness
Follows by induction directly from the binary-search-tree property.

Time
Intuitively, the walk takes $\Theta(n)$ time for a tree with n nodes, because we visit and print each node once. [Book has formal proof.]

The tree traversal corresponds to visiting all of the vertices in a tree in some order. Three approaches that yield interesting and useful results are:

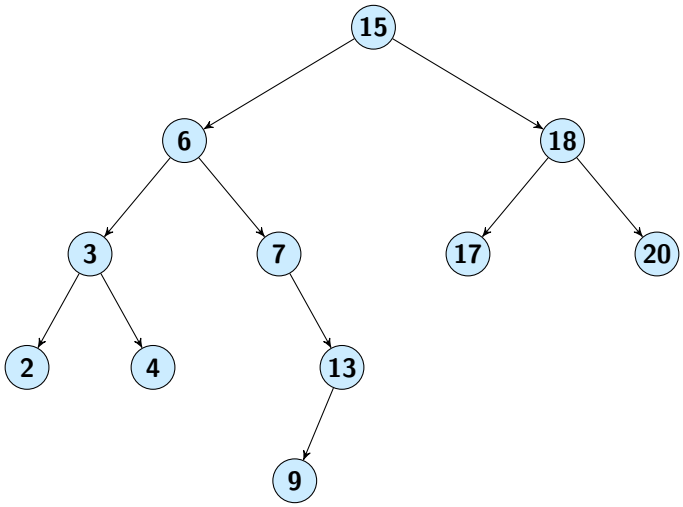
- Inorder traversal
- Preorder traversal
- Postorder traversal

Each of these traversals can be performed in $\Theta(n)$ time where $n = |T|$, and we can represent these choices using the following single algorithm:

TREETRAVERSE($root, type$)

```
1  if root = NIL then return
2  if type = "preorder" then
3      print(key[root])
4  TREETRAVERSE(root.left, type)
5  if type = "inorder" then
6      print(key[root])
7  TREETRAVERSE(root.right, type)
8  if type = "postorder" then
9      print(key[root])
```

Interpreting this algorithm, we note that **root** corresponds to a pointer to the root of a subtree, and type specifies the type of traversal from the set of values (**preorder, inorder, postorder**). From this, we see that a preorder traversal involves "visiting" the root of the tree first (indicated by printing that nodes key value; however, any operation can be performed during a "visit"), then traversing left subtree and traversing right subtree. Consider the following tree:



In this tree, a preorder traversal yields the sequence 8–3–2–6–4–12–9. Before looking at the inorder traversal, let's consider postorder. In a postorder traversal, from a root node, we traverse the left subtree, then traverse the right subtree, and then visit the root. Again, using the tree above, we see that postorder traversal yields the sequence 2–4–6–3–9–12–8.

Finally, let's consider the inorder traversal. For this traversal, we begin by traversing the left subtree from the root, then we visit the root node, and finally we traverse the right subtree. The reason we saved this for last is that, when applying an inorder traversal to a binary search tree, something interesting happens.

For example, consider the tree above. An inorder traversal of this tree yields the sequence 2–3–4–6–8–9–12. In other words, an inorder traversal of any binary search tree will visit the vertices of the tree in non-decreasing order of the key values stored in the tree.

3 Querying a Binary Search Tree

All dynamic-set search operations can be supported in $O(h)$ time. For a balanced binary tree the hight h has a running time of $h = \Theta(\log(n))$ (and for an average tree built by adding nodes in random order.) An unbalanced tree that resembles a linear chain of n nodes in the worst case is $h = \Theta(n)$.

3.1 Searching

TREE-SEARCH(x, k)

```
1  if x == NIL or k == key[x]
2        return x
3  if k < x.key
4        return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

The initial call is to TREE-SEARCH$(T.root, k)$.

Example
Search for values D and C in the example tree from above.

Time
The algorithm recurses, visiting nodes on a downward path from the root. Thus, running time is $O(h)$, where h is the height of the tree.

[The text also gives an iterative version of TREE-SEARCH, which is more efficient on most

3.2 Minimum and Maximum

The binary-search-tree property guarantees that:

- the minimum key of a binary search tree is located at the leftmost node, and
- the maximum key of a binary search tree is located at the rightmost node.

Traverse the appropriate pointers ($left$ or $right$) until NIL is reached.

TREE-MINIMUM(x)
1 **while** $x.left \neq$ NIL
2 $\quad x = x.left$

TREE-MAXIMUM(x)
1 **while** $x.right \neq$ NIL
2 $\quad x = x.right$

Time
Both procedures visit nodes that form a downward path from the root to a leaf. Both procedures run in $O(h)$ time, where h is the height of the tree.

3.3 Successor and Predecessor

Assuming that all keys are distinct, the successor of a node x is the node y such that $y.key$ is the smallest $key > x.key$. (We can find x's successor based entirely on the tree structure. No key comparisons are necessary.) If x has the largest key in the binary search tree, then we say that x's successor is NIL.

There are two cases:

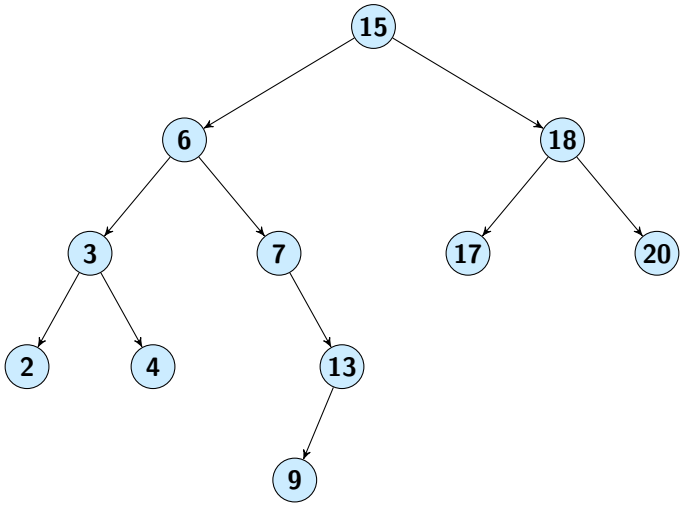
- If node x has a non-empty right subtree, then x's successor is the minimum in x's right subtree.
- If node x has an empty right subtree, notice that:
  - As long as we move to the left up the tree (move up through right children), we're visiting smaller keys.
  - x's successor y is the node that x is the predecessor of (x is the maximum in y's left subtree).

TREE-SUCCESSOR(x)
1 **if** $x.right \neq$ NIL
2 $\quad$ **return** TREE-MINIMUM($x.right$)
3 **while** $y \neq$ NIL and $x == y.right$
4 $\quad x = y$
5 $\quad y = y.p$ **return** y

TREE-PREDECESSOR is symmetric to TREE-SUCCESSOR as shown in the algorithm below:

```
TREE-PREDECESSOR(x)
1  if x.left ≠ NIL
2        return TREE-MINIMUM(x.left)
3  while y ≠ NIL and x == y.left
4        x = y
5        y = y.p return y
```

Example



- Find the successor of the node with key value 15. (Answer: Key value 17)
- Find the successor of the node with key value 6. (Answer: Key value 7)
- Find the successor of the node with key value 4. (Answer: Key value 6)
- Find the predecessor of the node with key value 6. (Answer: Key value 4)

Time

For both the TREE-SUCESSOR and TREE-PREDECESSOR procedures, in both cases, we visit nodes on a path down the tree or up the tree. Thus, running time is $O(h)$, where h is the height of the tree.

3.4 Insertion and Deletion

Insertion and deletion allows the dynamic set represented by a binary search tree to change. The binary-search-tree property must hold after the change. Insertion is more straightforward than deletion.

3.4.1 Insertion

TREE-PREDECESSOR(x)

```
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.p = y
 9  if y == NIL
10        T.root = z // tree T was empty
11  elseif z.key < y.key
12        y.ledt = z
13  else y.right = z
```

- To insert value into the binary search tree, the procedure is given node z, with $z.key = v, z.left =$ NIL, and $z.right =$ NIL.
- Beginning at root of the tree, trace a downward path, maintaining two pointers.
  - Pointer x: traces the downward path.
  - Pointer y: "trailing pointer" to keep track of parent of x.
- Traverse the tree downward by comparing the value of node at x with v, and move to the left or right child accordingly.
- When x is NIL, it is at the correct position for node z.
- Compare z's value with y's value, and insert z at either y's $left$ or $right$, appropriately.

Analysis of Insertion

- The initialization takes $O(1)$ time to complete
- The while loop in lines 3-7 searches for place to insert z, maintaining parent y which takes $O(h)$ time.
- Lines 8-13 insert the value and takes $O(1)$

It should be noted that this is the same as TREE-SEARCH. On a tree of height h, the procedure takes $O(h)$ time. TREE-INSERT can be used with INORDER-TREE-WALK to sort a given set of numbers. (See Exercise 12.3-3.)

3.4.2 Deletion

[Deletion from a binary search tree changed in the third edition. In the first two editions, when the node z passed to TREE-DELETE had two children, z's successor y was the node actually removed, with y's contents copied into z. The problem with that approach is that if there are external pointers into the binary search tree, then a pointer to y from outside the binary search tree becomes stale. In the third edition, the node z passed to TREE-DELETE is always the node actually removed, so that all external pointers to nodes other than z remain valid.]

Conceptually, deleting node z from binary search tree T has three cases:

- If z has no children, just remove it.

- If z has just one child, then make that child take z's position in the tree, dragging the child's subtree along.
- If z has two children, then find z's successor y and replace z by y in the tree. y must be in z's right subtree and have no left child. The rest of z's original right subtree becomes y's new right subtree, and z's left subtree becomes y's new left subtree.

This case is a little tricky because the exact sequence of steps taken depends on whether y is z's right child. The code organizes the cases a bit differently. Since it will move subtrees around within the binary search tree, it uses a subroutine, TRANSPLAN, to replace one subtree as the child of its parent by another subtree.

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2        T.root = z
3  elseif u == u.p.left
4        u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7        v.p = u.p
```

TRANSPLANT(T, u, v) replaces the subtree rooted at u by the subtree rooted at v:

- Makes u's parent become v's parent (unless u is the root, in which case it makes the root).
- u's parent gets as either its left or right child, depending on whether u was a left or right child.
- Doesn't update $v.left$ or $v.right$, leaving that up to TRANSPLANT's caller.

```
TREE-PREDECESSOR(x)
1  if x.left ≠ NIL
2        return TREE-MINIMUM(x.left)
3  while y ≠ NIL and x == y.left
4        x = y
5        y = y.p return y
```

Time

$O(h)$, on a tree of height h. Everything is $O(1)$ except for the call to TREE-MINIMUM.

Minimizing running time

We've been analyzing running time in terms of h (the height of the binary search tree), instead of n (the number of nodes in the tree).

- Problem: Worst case for binary search tree is $\Theta(n)$—no better than linked list.
- Solution: Guarantee small height (balanced tree)—$h = O(\lg(n))$.

In later chapters, by varying the properties of binary search trees, we will be able to analyze running time in terms of n.

- Method: Restructure the tree if necessary. Nothing special is required for querying, but there may be extra work when changing the structure of the tree (inserting or deleting).

Red-black trees are a special class of binary trees that avoids the worst-case behavior of $O(n)$ that we can see in "plain" binary search trees. Red-black trees are covered in detail in Chapter 13.

4 Conclusion

Binary search trees are viewed today as data structures that can support dynamic set operations, e.g., Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete. They can be used to build Dictionaries and Priority Queues. The basic operations take time proportional to the height of the tree – $O(h)$.

5 References

[1] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronal L., and Stein, Clifford, *Introduction to Algorithms*, 3rd Edition, MIT Press, 2009

[2] MIT Prof. Erik D. Demaine and MIT Prof. Charles E. Leiserson, Lecture Notes, Slides and Videos