Binary Search Trees

Adnan Aziz

1 BST basics

Based on CLRS, Ch 12.

Motivation:

- Heaps — can perform extract-max, insert efficiently $O(\log n)$ worst case
- Hash tables — can perform insert, delete, lookup efficiently $O(1)$ (on average)

What about doing searching in a heap? Min in a heap? Max in a hash table?

- Binary Search Trees support search, insert, delete, max, min, successor, predecessor
  - time complexity is proportional to height of tree
    * recall that a complete binary tree on $n$ nodes has height $O(\log n)$

Basics: A BST is organized as a binary tree

- added caveat: keys are stored at nodes, in a way so as to satisfy the BST property:
  - for any node $x$ in a BST, if $y$ is a node in $x$’s left subtree, then $key[y] \leq key[x]$,
    and if $y$ is a node in $x$’s right subtree, then $key[y] \geq key[x]$.

Implementation — represent a node by an object with 4 data members: key, pointer to left child, pointer to right child, pointer to parent (use NIL pointer to denote empty)
Figure 1: BST examples

Figure 2: BST working example for queries and updates
1.1 Query operations

1.1.1 Print keys in sorted order

The BST property allows us to print out all the keys in a BST in sorted order:

INORDER-TREE-WALK(x)

if x != NIL
    then INORDER-TREE-WALK(left[x])
    print key[x]
    INORDER-TREE-WALK(right[x])

1.1.2 Search for a key

TREE-SEARCH(x,k)

if x = NIL or k = key[x]
    then return x

if k < key[x]
    then return TREE-SEARCH(left[x],k)
else return TREE-SEARCH(right[x],k)

Try—search 13, 11 in example

1.1.3 Min, Max

TREE-MINIMUM(x)

while (left[x] != NIL)
    do x <- left[x]

return x

- Symmetric procedure to find max.

Try—min=2, max=20
1.2 Successor and Predecessor

Given a node $x$ in a BST, sometimes want to find its “successor”

- node whose key appears immediately after $x$’s key in an in-order walk

Conceptually: if the right child of $x$ is not NIL, get min of right child, otherwise examine parent

- Tricky when $x$ is left child of parent, need to keep going up the search tree

TREE-SUCCESSOR($x$)

```plaintext
if right[$x$] != NIL
    then return TREE-MINIMUM( right[$x$] )

y <- p[$x$]
while y != NIL and x = right[y]
    do x <- y
    y <- p[y]
```

Symmetric procedure to find pred.

Try—succ 15, 6, 13, 20; pred 15, 6, 7, 2

Theorem: all operations above take time $O(h)$ where $h$ is the height of the tree.

1.3 Updates

1.3.1 Inserts

**Insert**: Given a node $z$ with key $v$, left[$z$],right[$z$],p[$z$] all NIL, and a BST $T$

- update $T$ and $z$ so that updated tree includes the node $z$ with BST property still true

Idea behind algorithm: begin at root, trace path downward comparing $v$ with key at current node.

- get to the bottom $\Rightarrow$ insert node $z$ by setting its parent to last node on path, update parent left/right child appropriately

Refer to TREE-INSERT($T,z$) in CLRS for detailed pseudo-code.

Try—insert keys 12, 1, 7, 16, 25 in the example
1.3.2 Deletions

Given a node $z$ (assumed to be a node in a BST $T$), delete it from $T$.
Tricky, need to consider 3 cases:

1. $z$ has no children
   - modify $p[z]$ to replace the $z$-child of $p[z]$ by NIL

2. $z$ has one child
   - “splice” $z$ out of the tree

3. $z$ has two children
   - “splice” $z$’s successor (call it $y$) out of the tree, and replace $z$’s contents with those of $y$. **Crucial fact:** $y$ cannot have a left child

Refer to TREE-DELETE($T, z$) in CLRS for detailed pseudo-code.
Try—deleting 9, 7, 6 from example

**Theorem:** Insertion and deletion can be performed in $O(h)$ time, where $h$ is the height of the tree.
   - note how we crucially made use of the BST property

2 Balanced binary search trees

**Fact:** if we build a BST using $n$ inserts (assuming insert is implemented as above), and the keys appear in a random sequence, then the height is very likely to be $O(\log n)$ (proved in CLRS 12.4).
   - random sequence of keys: extremely unrealistic assumption

Result holds only when there are no deletes—if there are deletes, the height will tend to $O(\sqrt{n})$. This is because of the asymmetry in deletion—the predecessor is always used to replace the node. The asymmetry can be removed by alternating between the successor and predecessor.

Question 1. Given a BST on $n$ nodes, can you always find a BST on the same $n$ keys having height $O(\log n)$?
• Yes, just think of a (near) complete binary tree

Question 2. Can you implement insertion and deletion so that the height of the tree is always \(O(\log n)\)?

• Yes, but this is not so obvious—performing these operations in \(O(\log n)\) time is a lot trickier.

Broadly speaking, two options to keeping BST balanced

• Make the insert sequence look random (treaps)
• Store some extra information at the nodes, related to the relative balance of the children
  – modify insert and delete to check for and correct skewedness
  – several classes of “balanced” BSTs, e.g., AVL, Red-Black, 2-3, etc.

2.1 Treaps

Randomly built BSTs tend to be balanced ⇒ given set \(S\) randomly permute elements, then insert in that order. (CLRS 12.4)

• In general, all elements in \(S\) are not available at start

Solution: assign a random number called the “priority” for each key (assume keys are distinct). In addition to satisfying BST property (\(v\) left/right child of \(u\) ⇒ \(key(v) < / > key(u)\)), require that if \(v\) is child of \(u\), then priority(\(v\)) > priority(\(u\)).

• Treaps will have the following property: If we insert \(x_1, \ldots, x_n\) into a treap, then the resulting tree is the one that would have resulted if keys had been inserted in the order of their priorities

Treap insertion is very fast—only a constant number of pointer updates. It is used in LEDA, a widely used advanced data structure library for C++.

2.2 Deterministic balanced trees

Many alternatives: AVL, Red-Black, 2-3, B-trees, weight balance trees, etc.

• most commonly used are AVL and Red-Black trees.
Figure 3: Treap insertion
2.3 AVL Trees

Keep tree height balanced—for each node heights of left and right subtrees differ by at most 1.

Implement by keeping an extra field $h[x]$ (height) for each node.

Height of AVL tree on $n$ nodes is $O(\lg n)$.

- Follows from the following recurrence: If $M_h$ is the minimum number of nodes in an AVL tree of height $h$, $M_h = 1 + M_{h-1} + M_{h-2}$. The minimum number of nodes in an AVL tree of height 0 or 1 is 1.

The recurrence $F_h = F_{h-1} + F_{h-2}$ with $F_0 = F_1 = 1$ is known yields what are known as the “Fibonacci numbers”. These numbers have been studied in great detail; in particular, $F_h = \frac{\phi^h - (1-\phi)^h}{\sqrt{5}}$, where $\phi = (1 + \sqrt{5})/2$. Clearly, the Fibonacci numbers grow exponentially. Since $M_h > F_h$, this guarantees that $M_h$ grows exponentially with $h$.

Ordinary insertion can throw off balance by at most 1, and only along the search path.

- moving up from added leaf, find first place along search path where tree is not balanced
- re-balance the tree using rotations (Figure 4)

Upshot: insertion takes $O(\lg n)$ time, and the number of rotations is $O(1)$.

- Proof: based on figures

Deletions, as usual, are more complicated. Rebalancing cannot in general be achieved with a single rotation—there are cases where $\Theta(\lg n)$ rotations are required.

AVL trees are very efficient in practice—the worst cases require 45% more compares than optimal trees. Empirical studies show that they require $\lg n + 0.25$ compares on average.
Figure 5: AVL: Two possible insertions that invalidate balances. There are two symmetric cases on the right side. \(A\) is the first node on the search path from the new node to the root where the height property fails.

Figure 6: AVL: single rotation to rectify invalidation
2.4 Red-black trees

CLRS Chapter 13
RB-tree is a BST with an additional bit of storage per node—its color, which can be red or black.
Needs to satisfy the following properties:

P1 Every node is colored red or black
P2 The root is black
P3 Every leaf (NIL) is black
P4 If a node is red, both its children are black
P5 For every node, all paths to descendant leaves have the same number of black nodes

Define the black height of node $x$, $bh(x)$, to be the number of black nodes, not including $x$, on any path from $x$ to leaf (note that P5 guarantees that the black height is well defined).

Lemma 1 A red-black tree with $n$ internal nodes has height at most $1 + 2 \lg n$. 
Proof: Subtree rooted at any node \( x \) has at least \( 2^{bh(x)} - 1 \) internal nodes—this follows directly by induction.

Then note that P4 guarantees at least half the nodes on any path from root to leaf are black (including root). So the black height is at least \( h/2 \), hence \( n \geq 2^{h/2} - 1 \); result follows from moving the \(-1\) to left side, and taking logs.

From the above it follows that the insert and delete operations take \( O(\lg n) \) time.

However, they may leave result in violations of some of the properties P1–P5. Properties are restored by changing colors, and updating pointers through rotations. Conceptually rotation is simple, but code looks a mess—see page 278, CLRS.

Example of update sequence for insertion in Figure 9.

2.5 Splay trees

Based on Tarjan, “Data Structures and Network Algorithms,” 1983, Chapter 4, Section 3.

Key idea—adjust BST after each search, insert, and delete operation. (Tarjan actually allows a couple more operations, specifically, joining two trees, and splitting a tree at a node, but we’ll stick to the search/insert/delete ops.)

Key result—although a single operation may be expensive (\( \Theta(n) \), where \( n \) is the number of keys in the tree), a sequence of \( m \) total operations, of which \( n \) are inserts will complete in \( O(m \lg n) \).

Basic operation: “splaying”.

Figure 8: RB tree: Thick circles represent black nodes; we’re ignoring the NIL leaves.
Figure 9: Insert in RB-tree
As part of search/insert/update, perform splaying on each node on path from $x$ to root.

Definition of splaying at a node $x$:

- if $x$ has a parent, but no grandparent, rotate at $p(x)$

- if $x$ has a grandparent (which implies it has a parent), and both $x$ and $p(x)$ are left children or both are right children, then rotate at $p(p(x))$ then at $p(x)$

- if $x$ has a grandparent and $x$ is a left child, and $p(x)$ is a right child or vice versa, rotate at $p(x)$ and then the new $p(x)$ (which will be the old grandparent of $x$)

Search/update node $x$—splay at $x$
Delete $x$—splay its parent just prior to deletion

3 Augmenting data structures

CLRS Chapter 14
Typical engineering situation: rarely the case that cut-and-paste from textbook suffices.

- sometimes need a whole new data structure

- most often, augment existing data structure with some auxiliary information

Two examples—dynamic order statistics, and interval trees

3.1 Dynamic order statistics

In addition to usual BST operations (insert, delete, lookup, succ, pred, min, max), would like to have a rank operation:

- rank of element is its position in the linear order of the set

One approach to return fast rank information—balanced BST with additional size field

- $size[x] =$ number of nodes in subtree rooted at $x$, including $x$ itself

- $size[x] = size[left[x]] + size[right[x]] + 1$, where $size[nil] = 0$

Two kinds of queries:
- Retrieve element with given rank
- Determine rank of element

Let's see how to compute rank information using size field; later will show how size can be updated through inserts and deletes.

// Return element with rank i in tree rooted at node x
OS_SELECT(x, i)
    r <- size[left[x]] + 1
    if ( i = r )
        then return x;
    elseif ( i < r )
        then return OS_SELECT(left[x], i)
    else return OS_SELECT(right[x], i-r)

Idea: case analysis

// Return rank of element at node x in tree T
OS_RANK(T, x)
    r <- size[left[x]] + 1
    y <- x
    while ( y != root[T] )
        do if ( y = right[p[y]] )
            then r <- r + size[left[p[y]]] + 1
            y <- p[y]
    return r

Idea: at start of each iteration of while loop, r is rank of key[x] in subtree rooted at node y

Both rank algorithms are \( O(h) \) (= \( O(\lg n) \) for balanced BST).
How to preserve size field through inserts and deletes?
• Insert
  – before rotations: simply increment size field for each node on search path
  – rotations: only two nodes have field changes
  – upshot: insert remains $O(h)$
• Deletion: similar argument

3.2 Interval trees

Augment BSTs to support operations on intervals. A closed interval $[t_1, t_2]$ is an ordered pair of real numbers $t_1 \leq t_2$; it represents the set \{ $t \in R \mid t_1 \leq t \leq t_2$ \}. (Can also define open and half-open intervals, but no real difference).

Useful for representing events which occupy a continuous period of time.

• Natural query: what events happened in a given time?

Nice solution via BSTs.

Represent interval $[t_1, t_2]$ as an object $i$, with fields $low[i] = t_1$ and $high[i] = t_2$.

Fact: intervals satisfy trichotomy

• if $i, i'$ are intervals, then either they overlap, or one is to the left of the other

Interval tree—BST with each node containing an interval.

• given node $x$ with interval $int[x]$, BST key is $low[int[x]]$

• store additional information: $max[x]$, maximum value of any endpoint stored in subtree rooted at $x$

• must update information through inserts and deletes
  – can be done efficiently: $max[x] = \max(high[int[x]], max[left[x]], max[right[x]])$

New operation: interval search
// find node in T whose interval overlaps with interval i

INTERVAL_SEARCH( T, i )

x <- root[T]
while ( x != nil[T] ) && ( i does not overlap int[x] )
do if ( left[x] != nil[T] && ( max[left[x]] >= low[i] )
    then x <- left[x]
    else x <- right[x]
return x

Why does this work?

• If tree T contains an interval that overlaps i, then there is such an interval in the subtree rooted at T