Algo Lecture 4

Constantin Gierczak–Galle

October 26, 2020

Abstract

RED-BLACK trees and integer storing algorithms

Plan :

- Red-black trees
- Lower bound for searching algorithms
- Predecessor problem

I] Binary search trees.

It's a tree storing **integers** in its nodes. It must satisfy two *properties* :

- It must be **binary** ie. each non-leaf node must have *two children*.
- The **left** (resp. right) **subtree** of each node must contain elements that are $\leq$ (resp. $>$) than the node

Access, *insertion* and *deletion* of a gibven element are in $O(h)$ time where h is the *height* of the tree.

A **red-black tree** is a *BST* where each node is colored red or black. By constraigning the node colors on any simple path from root to leaf, we ensure that **no such path can be more than twice as long as any other** –> approximate balance

Properties :

- Every node is either *red* or *black*
- the *root* is **black**
- every *leaf* (NIL) is **black**
- is a *node* is **red**, then **both of its children are black**

- for each node, all simple path from it to descendant leaves contain the same number of black nodes

LEMMA : The height of such a tree with a node is at most $2log(n+1)$

PROOF : By *induction* : Let $bh(x)$ be the number of black nodes in a path from x to any leaf below it. By induction on $bh(x)$: the subtree rooted at x contains at least $2^{bh(x)} - 1$ nodes

Hence, if h is the black height of the tree, $n \geq 2^h - 1$ ie. $h \leq log(n+1)$

Since in any root-to-leaf path the number of red nodes is at most twice the number of black nodes, the lemma follows. **QED**

The *search* operation is therefore in $O(log(n))$ time. *Insert* and *delete* operation are a bit trickier 'cuz they could violate the red-black properties.

We hence have to *change color* of some nodes and perform *right or left rotations*.

1) Insertion

- We start by inserting the node n into the tree T as if we were in an ordinary search tree
- Then we color it red
- To guarantee the preservation of the red-black properties, we call an auxiliary procedure *RB-INSERT-FIXUP* to recolor nodes and perform rotations.

1.2) RB-INSERT-FIXUP

We now need to show hopw to restore the red-black properties after insertion. Let z be the node that violates the properties.

Case 1 : z's uncle y is red **Case 2 :** z's uncle y is black and z is a right child **Case 3 :** z's uncle y is black and z is a left child

Case 1 : color z.p black color y black color z.p.p red *That is we invert color of the parent, grandparent and uncle of z Then we treat z.p.p recursively*

Case 2 : We perform left-rotation on the edge z.p_z *We are now in case 3*

Case 3 : color z.p black color z.p.p red right-rotate on the edge z.p_z.p.p *That is we invert the color of the parent and grand-parent of z. Then, the red-black properties are restored*

Pseudo-code :

```
RB-INSERT-FIXUP(T, z):
  while z.p.color == RED:
    if z.p == z.p.p.left:
      y <- z.p.p.right
```

```
    if y.color == RED:  # Case 1
      z.p.color <- BLACK
      y.color <- BLACK
      z.p.p.color <- RED
      z <- z.p.p
    elif z == z.p.right # Case 2
      z <-s z.p
      LEFT-ROTATE(T, z)
      z.p.color <- BLACK
      z.p.p.color <- RED
      RIGHT-ROTATE(T, z.p.p)
    else:               # Case 3
      z.p.color <- BLACK
      z.p.p.color <- RED
      RIGHT-ROTATE(T, z.p.p)
  else
    same but inverting "right" and "left"
T.root.colro <- BLACK
```

Analysis : The while loop maintains the following three-part invariant. At the start of each iteration of the loop :

1. Node z is RED.
2. if z.p is the root, then z.p is BLACK
3. If there is a violation of the red-bblack properties, there is at most one violation and it is of either :
   - Prt 2 : The root is black. -> Violated chen z is the root and is red –> We color z black
   - Prt 4 : Is a node is RED, the nboth of its children are BLACK.
   - Violated when z and z.p are red. –> The depth of z can only decrease, this happen at least $O(log(n))$ times

Insertion therefore happens in O(log(n)) time.

2) Deletion

It's a bit trickier than the insertion We have *three cases :*

- The node has no children -> Simply delete it
- The node has exactly one child -> replace it with its child

- The node has two children -> replace it with its "successor" –> the bottom left element (hence minimum value of) of its right child

Then if the replacement is black, `RB-DELETE-FIXUP(T, x)` 'cuz the properties could be violated where x = y.left if $!= T.NIL else y.right where y = z if one the z's children is T.NIL else the successor (case 3)

If y.color == BLACK, we can violate these three properties :

- Prt 2 : the root is black -> when y si the root and we move y's child x, possibly RED, at its place)
- Prt 4 : if a node is RED, then both its children are BLACK
- Prt 5 : for each node, all simple paths from it to descendant leaves contain the same number of BLACK nodes -> every root-to-leaf path containing node y now contains one less black node

We consider 4 cases :

- x's sibling w is RED
- x's sibling w is BLACK and both children of w are BLACK
- x's sibling w is BLACK, w's left child is RED and right child is BLACK
- x's sibling w is BLACK and w's right child is RED

Case 1 : Rotation on edge w_w.p reduces to cases 2-4 on x and it's new sibling

Case 2 : Color w in RED reduces to 1

Case 3 : Right-rotation on w_(w's left child) to reduce to case 4

Case 4 : Rotation on w_w.p color w's right child to BLACK Now there's no violation anymore

DELETION is also in $O(log(n))$ time.

II] Dynamic optimality problem

In practice, we access some elements **more often than others**… -> Can we use that to make the **search faster**?

For a fixed access sequence X, let $OPT(X)$ be the number of unit-cost operations made by the fastest **BST** for X.

A BST is `c-competitive` is it executes all sufficiently long sequences X in time at most $c \times OPT(X)$

TANGO TREES are $O(log\ log\ n)$-competitive. **SPLAY TREES** are conjectured to be $O(1)$-competitive BUT this hasn't been proved.

A **RED-BLACK tree** can be used to sort a set of n integers in $O(n \times log\ n)$ time and $O(n)$ space It's not the best algorithm in practice (*large hidden constants*) -> DFT Assuming we're only allowed to compare integers, is is **asymptotically optimal**.

III] LOWER BOUND FOR INTEGER SORTING

Can we sort in $o(n \times log\ n)$? If only comparisons are allowed, **NO** Any **comparison-based sorting program** can be thought of a **decision tree of possible executions**. Running the same program twice on the same permutation causes it to do *exactly the same thing*, but running it on different permutations of the same data causes a different sequance of comparisons to be made on each.

CLAIM :

The **minimum height** of a decision tree is the **worst-case complexity of comparison-based sorting**.

LEMMA :

The height of any decision tree is $\Omega(n \times log\ n)$.

PROOF :

Since any two different permutations of n elements require a different sequence of steps to sort, there must be at least $n!$ different paths from the root to leavs in the decision tree.

Thus there must be at least n! leaves in this binary tree. Since a binary tree of height h has 2^h leaves, we know that : $n! \leq 2^h$ ie. $h \geq log(n!)$

Finally, $log(n!) = \Omega(n \times log\ n)$ QED.

IV] PREDECESSOR PROBLEM

Maintain a set S of **integers** from $U = [\![1; u]\!]$ subject to :

- *insertions*
- *deletions*
- **predecessor queries** : find the largest smallest element
- **successor queries** : find the smallest largest element

We can use **BST** for $O(n)$ space and $\Theta(log\ n)$ time –> Optimal in the comparison model or **van Emde Boas trees** for $O(u)$ space and $O(log\ log\ u)$ time.

VAN EMDE BOAS TREES :

Idea : split the universe U in $\sqrt{u}$ chunks of size $\sqrt{u}$ each and recurse for each chunk.

If the time per operation satisfies

$T(u) = T(\sqrt{u}) + O(1)$, $T(u) = O(log\ log\ u)$

T.summary is a **vBBT** of size $\sqrt{u}$ containing all i such that the i-th chunk is not empty.

for each $1 \leq i \leq \sqrt{u}$, T.chunk[i] is a **vBBT** of size $\sqrt{u}$ containing $x\%\sqrt{u}$ for each x in the i-th chunk

T.min :

the smallest element in T, not stored recursively (only stored once)

T.max :

the largest element in T

We introduce **hierchical coordinates :** Represent each integer $x = (c, i)$ where

- c is the chunk containing x ($c = x//\sqrt{u}$)
- i is the position of x in that chunk ($i = x\%\sqrt{u}$)

```
SUCCESSOR(T, x = (c, i)):
    if x < T.min:
      return T.min
    if i < T.chunk[c].max:  # successor has to be searched in the same chunk&
      return (c, SUCCESSOR(T.chunk[c], i))
    else:
      c' = SUCESSOR(T.summary, c) # successor has to be searched in the following chunk
      return (c', T.cluster[c'].min)
```

Analysis : in each case, we have $T(u) = T(\sqrt{u}) + 1$

```
INSERTION(T, x = (c, i)):
  if T.min = None: # if the tree is empty
    T.min = T.max = x
    return
  if x < T:
    swap(x, T.min)
  if x > T.max:
    T.max = x
  if T.chunk[c].min = None: # bucket is empty
    INSERT(T.summary, c) # Next call in constant time! That's because it will store once the
  INSERT(T.chunk[c], i)
```

```
DELETION(T, x = (c, i)):
  if x = T.min:
    c = T.summary.min
    if c = None:  # T only contains T.min
      T.min = None
    x = T.min = (c, i=T.chunk[c].min) # unstore new min
  DELETE(T.chunk[c], i)
  if T.chunk[c].min = None: # ie. that chunk is now empty
    DELETE(T.summary, c)    # we then delete chunk c from the summary
  if T.summary.min = None:  # if T only contains one element : T.min
    T.max = T.min
  else:                     # update the maximum after deletion
    c' = T.summary.max
    T.max = (c', T.chunk[c'].max)
```

We therefore have :

- $T(u) = T(\sqrt{u}) + O(1)$ ie. $T(u) = O(log\ log\ u)$
- $S(u) = (\sqrt{u} + 1) \times S(\sqrt{u}) + O(1)$ ie. $ S(u) = O(u)$

Can we use instead **space** in $O(n)$?

We construct a **perfect binary tree** whose *leafs* are all **elements from** U

Each leaf holds a *bit*. 1 means the **element is in the tree**. We then define the bit for every node : the `OR` of its children. It thus has a height in $O(log\ u)$.

Then, to get the *predecessor/successor* of an element x, just follow the path from the root to x. Once you encounter a 0, juste keep on following the ones. At the end, you get the **predecessor** or the **successor**.

If the tree is stored in a **double linked list**, we can thus get the predecessor or the successor in $O(log\ log\ u)$ time (store it like with a **heap**)

Better : X-FAST TRESS :

Store every *root-to-leaf* path corresponding to a present element viewed in *binary* (left = 0, right = 1), using **cuckoo hashing**. **Predecessor queries** in $O(log\ log\ u)$ time, **update** in $O(log\ u)$ expected amortised time and **space** $O(n\ log\ u)$

Even better : Y-FAST TREES :

We create one big tree containing $O(n)$ nodes and small trees containing between $[\frac{1}{2}, 2] \times log(u)$ elements

The small trees work like BST $\rightarrow$ groups $\rightarrow$ a representative for each group is stored in the big tree. $\rightarrow O(\frac{n}{log\ u]})$ elements in the big tree

We then get :

- Predecessor queries in $O(log\ log\ u)$ time
- Updates in $O(log\ log\ u)$ expected amortised time
- Space : $O(n)$

How to maintain the $[\frac{1}{2}, 2] \times log(u)$ elements in each group?

If there are fewer than $\frac{log(u)}{2}$ elements in total, store them in a single **BST**.

Otherwise, suppose we add/delete an element. If a group becomes too large, split it in two. If a group becomes too small, merge it with its neighbour then split it if needed.