Binary Search Trees

Nicolò Felicioni[1]

Dipartimento di Elettronica e Informazione
Politecnico di Milano

nicolo . felicioni @ polimi . it

June 1, 2021

[1] Based on Nicholas Mainardi's material, enriched with few additional examples.

Binary Trees

Binary Tree: Definition

A tree is a pair of sets (nodes V and edges E) with a particular structure. In particular, there are no cycles and edges are directed, defining a *parent relationship*. That is, $\forall i, j \in V((i, j) \in E \Rightarrow i$ is parent of $j)$. Each tree has a single root node, which is the only one with no parents, and leaves, which are nodes with no children. A tree is n-ary if its nodes can have at most n children. Therefore, in a binary tree nodes can have at most 2 children

- Trees are useful to model different real world situations, especially hierarchical ones
- Binary Search Trees (BST) are perfectly suitable for searching operations

Binary Search Trees

Definition

In a Binary Search Tree (BST), for any node with key k, all nodes in the subtree rooted in its left child have a key $\leq k$, while all nodes in the subtree rooted in its right child have a key $\geq k$.

Operations

- $\mathrm{SEARCH}, \mathrm{INSERT}, \mathrm{DELETE}$
- MIN = leftmost leave, MAX = rightmost leave
- These operations are $O(h)$, where h is the height of the tree: typically $\log(n)$ if the tree is balanced
- All visit operations are $O(n)$:
  - $\hookrightarrow$ $\mathrm{IN\text{-}ORDER\text{-}VISIT}$: nodes are visited in increasing order
  - $\hookrightarrow$ $\mathrm{PRE\text{-}ORDER\text{-}VISIT}$: parents are visited before children
  - $\hookrightarrow$ $\mathrm{POST\text{-}ORDER\text{-}VISIT}$: parents are visited after children

Create a balanced BST

Problem

Given an array A, create a balanced BST with all the keys contained in A

Idea for the Solution - 1

- Maybe if we sort A ($O(n \log(n))$) the problem is simpler
- But we can not insert the elements of A in an empty BST after the sorting, otherwise we will end up with a completely unbalanced tree!
- We should insert the central element first
- Then, apply the same policy to the right sub-array and the left sub-array

Create a balanced BST

Problem

Given an array A, create a balanced BST with all the keys contained in A

Idea for the Solution - 1 - Complexity

- The sorting has a worst-case complexity of $\Theta(n\log(n)))$
- For each element $i+1$, we must insert the element in a BST with i elements already present

$$T(n) = \Theta(n\log(n)) + \sum_{i=1}^{n} \log(i) =$$

$$\Theta(n\log(n)) + \log(n!) = \Theta(n\log(n))$$

(We used the Stirling approximation)

Create a balanced BST

Problem

Given an array A, create a balanced BST with all the keys contained in A

Idea for the Solution - 2

- First, sort the array
- Then, create a node with a key equals to the central element of the array
- At this point, recursively repeat the procedure for the left sub-array, and make the node returned by the recursive call be the left child of the current node
- Apply recursively the procedure also to the right sub-array, and make the node returned by the recursive call be the right child of the current node

Create a balanced BST

Problem

Given an array A, create a balanced BST with all the keys contained in A

Idea for the Solution - 2 - Pseudocode

$\textsc{Build}(A, l, r)$

1 **if** $l > r$
2 **then return** NIL
3 $mid \leftarrow (l + r)/2$
4 $T \leftarrow BST()$
5 $T.key \leftarrow A[mid]$
6 $T.left \leftarrow \textsc{Build}(A, l, mid - 1)$
7 $T.right \leftarrow \textsc{Build}(A, mid + 1, r)$
8 **return** T

Create a balanced BST

Problem

Given an array A, create a balanced BST with all the keys contained in A

Idea for the Solution - 2 - Complexity

In order to solve the problem we will call $\textsc{Build}(A, 1, A.len)$.
In this case, the complexity of $\textsc{Build}$ will be:

$$T_{build}(n) = 2T(n/2) + \Theta(1)$$

Hence, according to the Master's Theorem, $T_{build}(n) = \Theta(n)$.
The total complexity (including the sorting phase) will be:

$$T(n) = T_{sort}(n) + T_{build}(n) = \Theta(n \log(n)) + \Theta(n) = \Theta(n \log(n))$$

Minimum Common Ancestor

Problem

A node is a common ancestor of 2 nodes if it is an ancestor for both of them. The minimum common ancestor is the closest to the nodes, that is the one furthest from the root of the tree. Suppose there is a BST where each node has a distinct key. How to find the minimum common ancestor of two nodes with keys k_1, k_2?

Idea for the Solution

- Where is the minimum common ancestor w.r.t. the 2 nodes?
- If one of the 2 nodes is an ancestor of the other one, then the former is surely the minimum common ancestor
- In the other case, it means that the nodes are in different branches of the tree
- Therefore, the minimum common ancestor is the node where the 2 different branches fork

Minimum Common Ancestor

Algorithm Design

- Can we identify the forking point?
- For nodes before the forking point, either both the nodes are in the left subtree or both the nodes are in the right subtree
- Therefore, their keys are either both smaller or both larger than the 2 nodes
- The forking node is the first one having its key in between the keys of the 2 nodes, since one node is in the left subtree and the other node is in the right subtree
- Note that this property is true also in the case where one node is an ancestor of the other, since the key of the minimum common ancestor is equal to one of the 2 nodes
- Thus, we can look for such a node starting from the root of the tree

Minimum Common Ancestor

Algorithm Design

- We need to choose the correct branches in order to get to the minimum common ancestor
- In particular, we choose the left subtree if the keys of both nodes are smaller than the current one, and vice versa

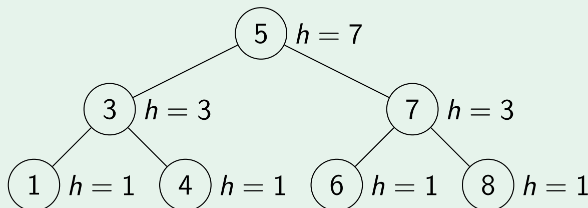
```
MINIMUM COMMON ANCESTOR(T, k1, k2)
1  mca ← T.root
2  while mca ≠ NIL
3       do if mca.key > k1 ∧ mca.key > k2
4              then mca ← mca.left
5              else  if mca.key < k1 ∧ mca.key < k2
6                        then mca ← mca.right
7                        else  return mca
```

Number of Heirs

Problem

Design an algorithm which computes the sum of the number of heirs for each node of the tree

- Heirs are the opposite of ancestors
- Each node is heir of all its ancestors, thus the sum will be larger than the number of nodes of the tree



In this example, the number of heirs of each node is written on the right. The sum of them is 17

Number of Heirs

Algorithm Design

- First of all, we need to count the number of heirs of a given node
- This is equivalent to count the number of nodes of the subtree rooted at the given node
- Recursive Method to do it?

CountElements(*node*)

1 $s_1 \leftarrow s_2 \leftarrow 0$
2 **if** $node.left \neq NIL$
3 **then** $s_1 \leftarrow$ CountElements($node.left$)
4 **if** $node.right \neq NIL$
5 **then** $s_2 \leftarrow$ CountElements($node.right$)
6 **return** $s_1 + s_2 + 1$

Number of Heirs

Algorithm Design

- From this algorithm, we can get the sum of all heirs:
  SumHeirs(*node*)

  1 $acc_1 \leftarrow acc_2 \leftarrow 0$
  2 **if** $node.left \neq NIL$
  3 **then** $acc_1 \leftarrow$ SumHeirs($node.left$)
  4 **if** $node.right \neq NIL$
  5 **then** $acc_2 \leftarrow$ SumHeirs($node.right$)
  6 **return** $acc_1 + acc_2 +$ CountElements($node$)

- For each node, we count its heirs, leveraging CountElements, and we add it to the accumulator storing the sum of all heirs, which is recursively computed

Number of Heirs

Complexity

- Complexity of CountElements: we visit each node once, performing only additions $\Rightarrow O(m)$, with m being the size of the tree rooted in *node*
- Complexity of SumHeirs: recursive algorithm:
  - the non recursive part is mainly the execution of CountElements, which has linear complexity
  - the number of recursive calls and the size of the subproblem depend on the structure of the tree
  - The best case is a perfectly balanced tree, thus the recurrence equation is $T(n) = 2T(\frac{n}{2}) + O(n) \Rightarrow T(n) = O(n\log(n))$
  - The worst case is a completely unbalanced tree, thus the recurrence equation is $T(n) = T(n-1) + O(n) \Rightarrow T(n) = O(n^2)$

Number of Heirs

Algorithm Design

- Can we do better? It seems we are "wasting" recursion calls
- Indeed, we use a recursive algorithm which calls at each iteration another recursive procedure which has actually the same recursion scheme
- Maybe we can merge these 2 routines employing a single recursion tree
- We would need to compute the sum while counting the nodes
- We employ an accumulator variable, returning 2 values from the recursive calls
- We can do it in pseudocode since we can always do it in a programming language (return a pointer to a structure with 2 integers)

Number of Heirs

Pseudocode

$\textsc{SumHeirs}(node)$

1. $acc_1 \leftarrow acc_2 \leftarrow 0$
2. $s_1 \leftarrow s_2 \leftarrow 0$
3. **if** $node.left \neq NIL$
4. **then** $(s_1, acc_2) \leftarrow \textsc{SumHeirs}(node.left)$
5. **if** $node.right \neq NIL$
6. **then** $(s_2, acc_2) \leftarrow \textsc{SumHeirs}(node.right)$
7. $s \leftarrow s_1 + s_2 + 1$
8. **return** $(s, acc_1 + acc_2 + s)$

Time complexity is $O(n)$, since we visit once all the nodes of the tree performing only additions

An Interesting Property

We are willing to understand if the following property holds for a binary search tree:

Definition

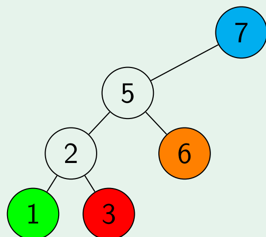
Consider a leaf node with key k. The search path for key k is the sequence of nodes visited during a search for k. Define A the set of nodes on the left of the search path for k, B the set of nodes belonging to the search path, and C the set of nodes on the right of the search path. In particular, a node is on the left (right) of the search path if it belongs to a left (right) subtree which has been discarded during search for key k. Is it true that $\forall a \in A, b \in B, c \in C (a \leq b \leq c)$?

An Interesting Property

Intuitively, this property seems to hold!

- If a node c is on the right of a search path, it means that at some point in the tree we chose the left subtree, thus all values in this subtree are smaller than $c \ldots$
- However, this right subtree may be in the left subtree of another node in B, thus $c \leq b$!

Counterexample



- For $k = 3$, the search path is $7, 5, 2, 3$
- Consider $a = 1$, $b = 7$, $c = 6$
- In this example, it holds $a \leq b$, but $c < b$!
- Thus, the property does not hold!

Non Binary Search Trees

Problem

Consider a tree which is not binary. Each node n of the tree has the following attributes:

- $key[n]$ is the value of the key associated to the node
- $lc[n]$ is the pointer to the leftmost child
- $rs[n]$ points to the closest right sibling of the node
- $ls[n]$ points to the closest left sibling of the node
- $p[n]$ points to the parent of the node

The root node has clearly no parent, neither siblings.

Question: Is it possible to employ this type of tree as a search one? How?

Non Binary Search Trees

What does It mean having a search tree?

- We can discard a branch of the subtree rooted in current node depending on its comparison with the element to be searched
- Since there are only 2 outcomes for the comparison not yielding the searched element, we need only 2 possible branches for each node out of the 4 possible ones for a generic node $(lc[n], rs[n]ls[n], p[n])$. How to select them?
- Consider the root element: only $lc[n]$ is not NIL: trivial choice
- Consider now the leftmost child: $ls[n]$ is NIL, $p[n]$ is the root, which has already been visited in the look-up, thus there are only 2 alternatives: $lc[n]$ and $rs[n]$
- Consider now a generic node: both $ls[n]$ and $p[n]$ have already been visited, since we have no other paths to this element from the root, thus there are only 2 choices: $lc[n]$ and $rs[n]$

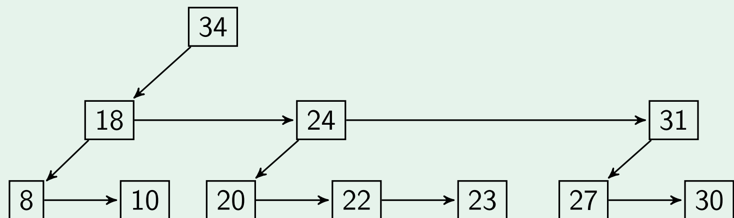
Non Binary Search Trees

Data Structure Design

We can organize the tree as follows

- Given a node n, all the elements stored in the sub-trees rooted in n's children are smaller than n
- All the elements stored in the sub-trees rooted in n's right siblings are greater than n

An Example



Non Binary Search Tree

- Which element is stored in the root? $\Rightarrow$ The maximum!
- Search strategy: choose the leftmost child if the searched key is smaller than the current node, the right sibling otherwise

Searching Algorithm

SEARCH(T, k)

1 $node \leftarrow T.root$
2 **while** $node \neq NIL$
3   **do if** $key[node] = k$
4   **then return** $node$
5   **if** $k < key[node]$
6   **then** $node \leftarrow lc[node]$
7   **else** $node \leftarrow rs[node]$
8 **return** NIL

Non Binary Search Tree

Successor Algorithm

Where is the successor of a node? There are 2 cases:

1. The node has a right sibling: the successor is the minimum of the subtree rooted in the right sibling
2. The node has no right sibling: the successor is among its ancestors, since the children are always smaller
   - Which ancestor? The parent! Indeed, the left siblings are smaller, while the parent is the closest ancestor being greater than the node

SUCCESSOR(T, z)

```
1  if rs[z] = NIL
2     then return p[z]
3  node ← rs[z]
4  while lc[node] ≠ NIL
5        do node ← lc[node]
6  return node
```

Non Binary Search Trees

Predecessor Algorithm

Where is the predecessor of a node? There are 2 cases:

1. The node has children: the predecessor is the rightmost one
2. The node has no children: as parent nodes are all greater, the predecessor is the first left sibling found among its ancestors

PREDECESSOR(T, z)

```
1   if lc[z] = NIL
2       then node ← z
3            while node ≠ NIL
4                 do if ls[node] ≠ NIL
5                       then return ls[node]
6                    node ← p[node]
7            return NIL
8   node ← lc[z]
9   while rs[node] ≠ NIL
10      do node ← rs[node]
11  return node
```

Non Binary Search Trees

Insertion Algorithm

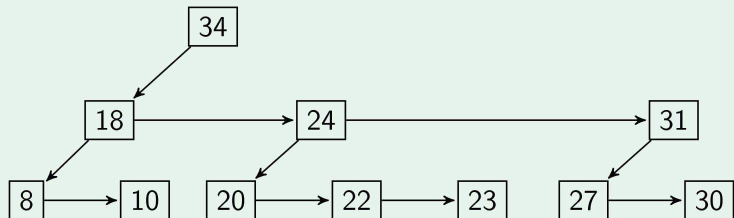
INSERT(T, z)

```
 1  if T.root = NIL
 2     then T.root = z
 3     else if key[z] > key[T.root]  ▷ Insert in place of the root
 4             then p[T.root] ← z
 5                  lc[z] ← T.root
 6                  T.root ← z
 7             else ▷ Place the node where it will be found by SEARCH
 8                  node ← T.root
 9                  while node ≠ NIL
10                        do pred ← node
11                           if key[z] < key[node]
12                              then node ← lc[node]
13                              else node ← rs[node]
14                  if key[z] < key[pred]
15                     then lc[pred] ← z
16                          p[z] ← pred
17                     else rs[pred] ← z
18                          ls[z] ← pred
19                          p[z] ← p[pred]
```

Non Binary Search Trees

Deletion

Consider again our example tree:



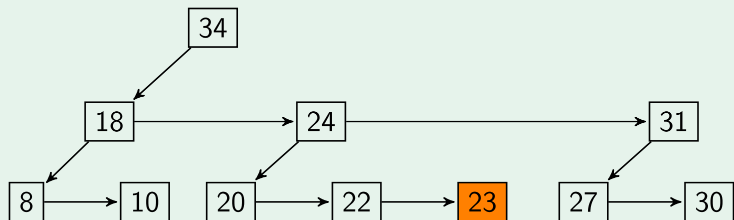
How do we delete a node? $\Rightarrow$ We have different cases! Let's tackle them...

Non Binary Search Tree

Case 1

Delete a node with no subsequent nodes (neither children, nor siblings)

- This is the simplest one: simply deletes the node

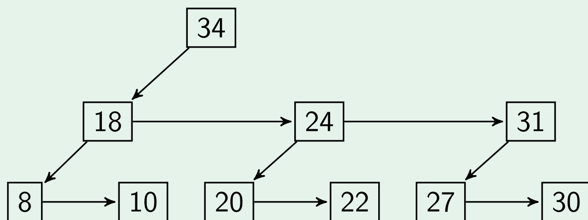


Non Binary Search Tree

Case 1

Delete a node with no subsequent nodes (neither children, nor siblings)

- This is the simplest one: simply deletes the node

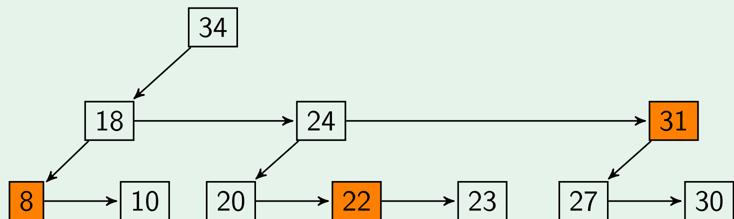


Non Binary Search Tree

Case 2

Delete a node with one subsequent node

- We need to link its single subtree to the previous node in the tree

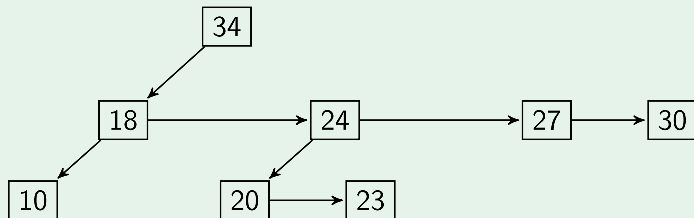


Non Binary Search Tree

Case 2

Delete a node with one subsequent node

- We need to link its single subtree to the previous node in the tree

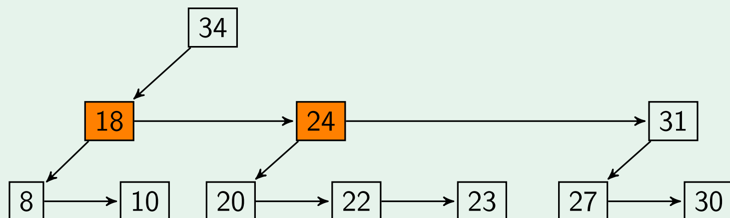


Non Binary Search Tree

Case 3

Delete a node with 2 subsequent nodes

- In this case, we need to replace the node either with its successor or with its predecessor
- The successor (resp. predecessor) is then erased using case 2 algorithm, since we know it has no children (resp. no right sibling)

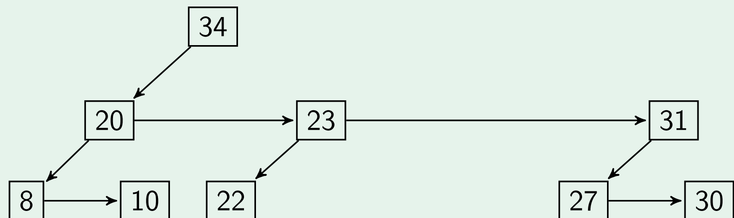


Non Binary Search Tree

Case 3

Delete a node with 2 subsequent nodes

- In this case, we need to replace the node either with its successor or with its predecessor
- The successor (resp. predecessor) is then erased using case 2 algorithm, since we know it has no children (resp. no right sibling)



Non Binary Search Tree

Case 4?

- It seems all possible cases are covered
- However, there is still a particular case to be handled
- What happens when we want to erase the root of the tree?

Erasing the Root

Since the root node cannot have siblings, we know that there are only 2 possibilities:

1. The root node has no left child: it is the only node of the tree, thus we simply erase it
2. The root node has a left child: Can we apply case 2 algorithm?

Non Binary Search Tree

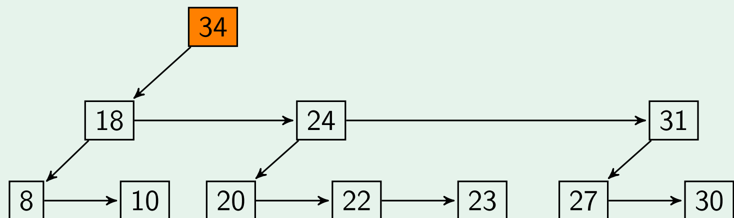
Erasing The Root

- The root has no previous node, indeed both $ls[n]$ and $p[n]$ are NIL
- If we simply erase it, and update the $T.root$ pointer to the leftmost child, we get an incorrect tree if this child has siblings
- Which node to be placed in the root?
- Recall one property of this type of tree: which element is stored in the root? The maximum!
- Therefore, if we erase the root element, the new one must be the bigger element apart from the erased one $\Rightarrow$ predecessor of the root!
- Where is the predecessor? It is the rightmost sibling among the root's children

Non Binary Search Tree

Erasing The Root

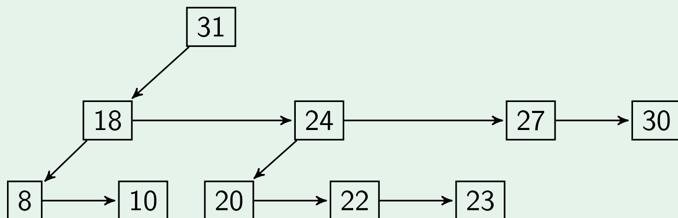
- Therefore, if the element to be erased is the root and it has a left child, we search for the predecessor, we replace the root with its key and we erase it
- As for the successor, we know the predecessor has no right sibling, thus we can apply the algorithm for case 2



Non Binary Search Tree

Erasing The Root

- Therefore, if the element to be erased is the root and it has a left child, we search for the predecessor, we replace the root with its key and we erase it
- As for the successor, we know the predecessor has no right sibling, thus we can apply the algorithm for case 2

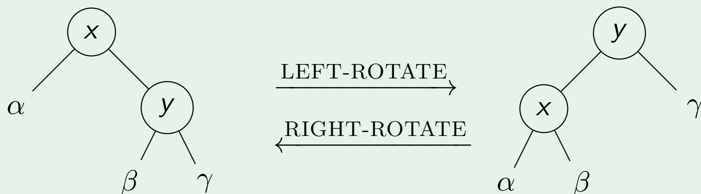


Number of Rotations

Problem

Given a binary search tree with n nodes, how many rotations are possible?

Rotations Recap



Number of Rotations

Solving the Problem

- Let's try to think about what is actually rotating in rotations
- Basically, an edge between 2 nodes move from a left subtree to a right one (RIGHT-ROTATE), and vice versa (LEFT-ROTATE), rotating the 2 nodes altogether
- Can the transformation be applied to every edge?
- Yes, just the 2 nodes x, y are needed
- Hence, there is a possible rotation for each edge in a binary search tree. Thus, the solution of the problem is found if we know the number of edges
- In a tree, a node has exactly one ingoing edge, except for the root
- Therefore, $n - 1$ links exist $\Rightarrow n - 1$ rotations are possible

Persistent Trees

Problem

Sometimes, it may be useful to keep track of all the past versions of a set of elements. One trivial solution is duplicate the set for each modification performed and apply it on the copy. However, this solution increases a lot the memory consumption.
Nevertheless, we can try to minimize the number of elements to be copied and modified by taking into account only the ones affected by modifications of the set. Suppose the set is stored in a binary search tree. A data structure which keeps track of all the past versions of the tree is called a **persistent tree**. Which nodes need to be modified (and thus duplicated) during the common tree operations (SEARCH, INSERT, DELETE) to get a persistent tree?

Persistent Trees

An Unpleasant Outcome

- Consider an INSERT operation
- For sure, we introduce a new element in the tree, which needs to be linked as a child of its parent node
- Hence, we have to modify, and thus duplicate, its parent too
- Moreover, if the parent has another child, we need to update its parent reference too
- Basically, if a node is modified, we need to modify both its parent and its children, thus it turns out we have to modify each node of the tree!
- Therefore, for a single insertion, a whole copy of the tree is needed $\Rightarrow$ the space consumption increases by n for each modification

Persistent Trees

Improving the Solution

- Usually, in a binary search tree there is a bit of redundancy
- In particular, parent pointers are surely useful, but the fundamental tree operations can be performed without them
- Indeed, $\text{INSERT}, \text{SEARCH}$ and DELETE always go down in the tree, whilst visit algorithms aptly use recursion
- TREE-SUCCESSOR may be re-written without employing the parent pointer still achieving $O(h)$ complexity
- Which benefits do we have for a persistent tree if we remove parent pointers?
- When we edit a node, we no longer need to update its children, since there is no more a parent pointer to be updated!
- How many nodes do we need to modify now?

Persistent Trees

Insertion

- During insertion, the parent of the new node must be updated with the reference to its new child
- Moreover, the grandfather of the inserted node needs to be updated too, since its child was modified
- With the same line of reasoning, all the ancestors up to the root needs to be updated, thus at most $O(h)$ nodes
- We can copy all the visited nodes during the descendant searching for the place where to insert the new node, still retaining $O(h)$ complexity

Persistent Trees

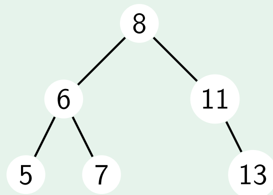
PERSITENT-TREE-INSERT

PERSISTENT-TREE-INSERT(T, z)

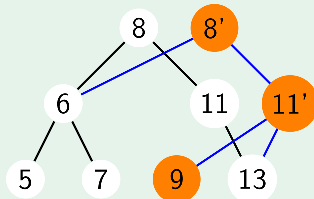
```
 1  if T.root = NIL
 2     then T'.root ← z
 3     else  T'.root ← COPY-NODE(T.root)
 4           x ← T'.root
 5           while x ≠ NIL
 6               do y ← x
 7                  if z.key < x.key
 8                     then x ← COPY-NODE(x.left)
 9                          y.left ← x
10                     else  x ← COPY-NODE(x.right)
11                          y.right ← x
12           if z.key < y.key
13              then y.left ← z
14              else  y.right ← z
15  return T'.root
```

Persistent Trees

Insertion Example - Original Graph



Insertion Example - Inserting 9



Persistent Trees

Deletion

- For deletion, we have 2 cases:
  1. The node to be erased has less than 2 children
  2. The node to be erased has 2 children
- In the first case, we simply update pointers of the parent node, and thus of all the ancestors, which are at most $O(h)$
- In the second case, we replace the erased node with the successor and we erase the successor, thus both its ancestors and the ancestor nodes of its successor need to be modified
- However, if z has 2 children, it is an ancestor of its successor y, hence only the ancestors of y need to be updated $\Rightarrow$ still $O(h)$ nodes

Persistent Trees

Can We Get Persistent Red and Black Trees?

- They employ additional operations: Rotations and fixup algorithms
- Both these algorithm leverages the parent pointer
- Therefore, can we get parent nodes in $O(1)$ still retaining $O(\log(n))$ space?
- The key idea is that we can build a stack of all the ancestors of the node to be inserted/erased with complexity $O(\log(n))$
- Since these algorithms are used in INSERT/DELETE procedures, building this stack does not increase asymptotic complexity
- How many nodes need to be updated by these operations?

Nodes Affected By Rotations

- Rotations involve 2 nodes x, y, where x is the parent of y, and their ancestors
- Indeed, after the rotation, y becomes the children of the parent of x, thus the parent must be updated
- Conversely, moving the subtrees of x, y does not require to update the children nodes, since just the pointers in x and y are modified
- Since x is the parent of y, they share the same ancestors
- In conclusion, rotations modify $O(\log(n))$ nodes

Nodes Affected by Fixup Algorithms

- Fixup perform at most 3 rotations, involving $O(\log(n))$ nodes
- Additionally, fixup may change the color of the sibling node (which is not an ancestor of the inserted/erased node z) and the two nephews (in case of DELETE-FIXUP). Thus all their ancestors need to be updated. However, the ancestors of these nodes are a subset of the ones of z
- A single call of fixup algorithm modifies $O(\log(n))$ nodes
- However, fixup algorithms are recursive: they start from the inserted/erased node, and then they may be invoked on its ancestor, up to the root: $O(\log(n))$ recursive calls
- Each of this call adds at most 3 nodes to the set of modified nodes (the sibling/nephews of the current fixed-up node)
- In conclusion, the nodes to be modified are at most $O(\log(n))$