A Simple and Efficient Union-Find-Delete Algorithm

Amir Ben-Amram, Simon Yoffe

ACAC
Athens 2010

Union Find data structure

Maintain a collection of disjoint sets under the operations:

Makeset(a) – create new set for element a

Union(A, B) – destructive union of sets A and B

Find(a) – find the set containing a

Union Find Delete data structure

Maintain a collection of disjoint sets under the operations:

Makeset(a) – create new set for element a

Union(A, B) – destructive union of sets A and B

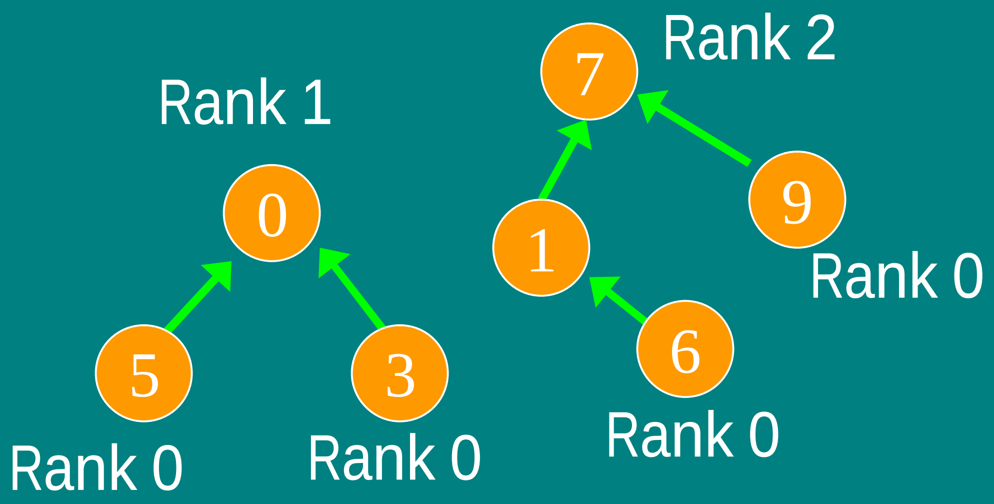
Find(a) – find the set containing a

Delete(a) – remove a from its containing set

The classic data structure

Represent each set A as rooted tree T_A

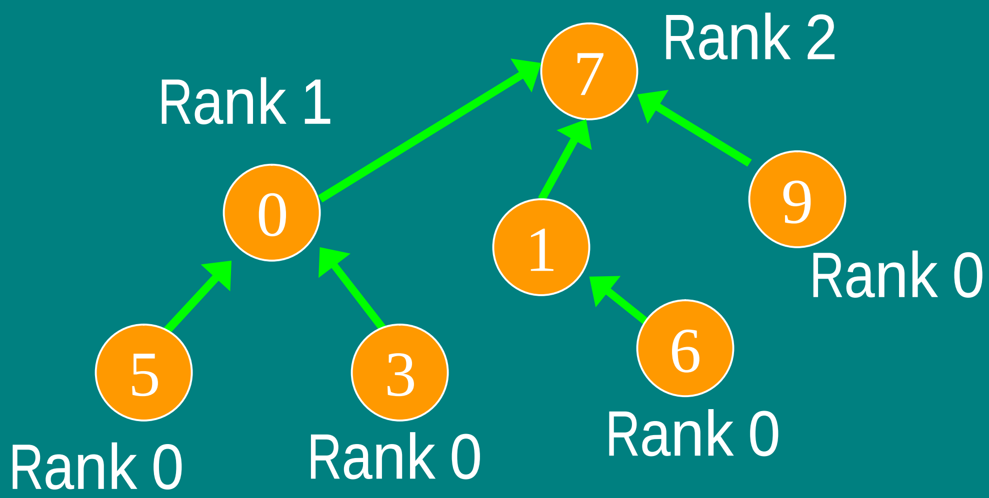
Union links the root of the shallower tree to the root of the taller tree (by rank)



The classic data structure

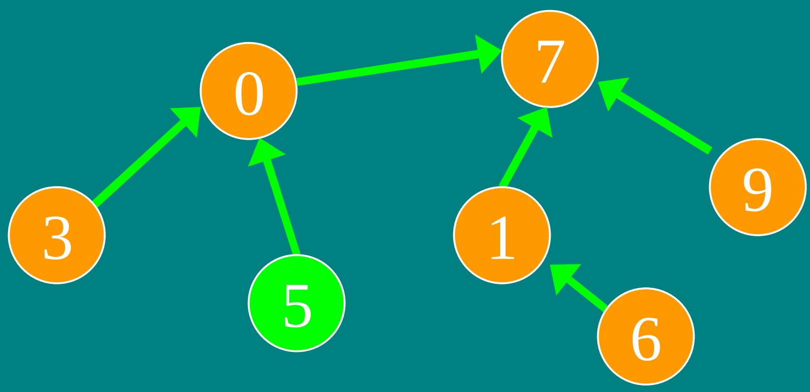
Represent each set A as rooted tree T_A

Union links the root of the shallower tree to the root of the taller tree (by rank)

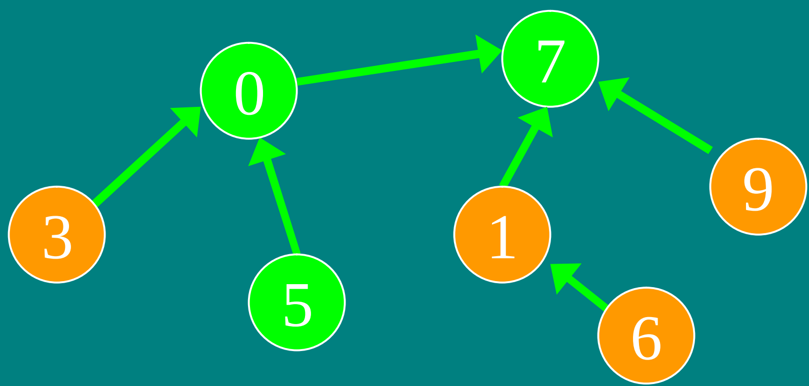


O(1)

Find climbs from the provided element up to the root and returns the root as the set identifier

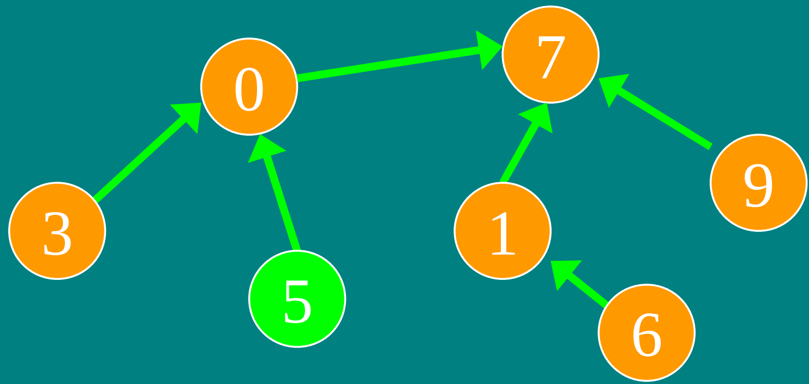


Find climbs from the provided element up to the root and returns the root as the set identifier



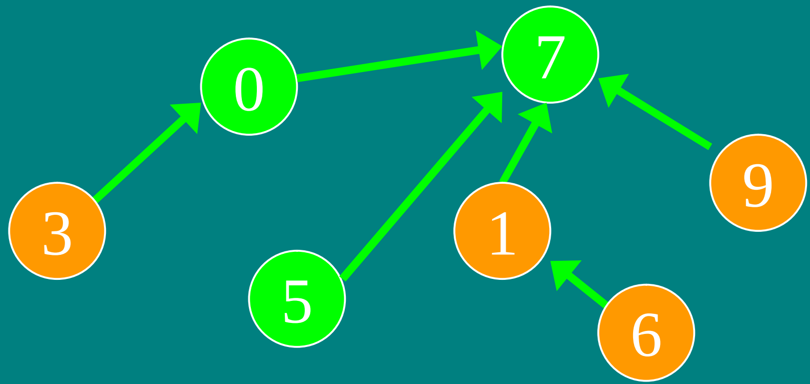
O(log(n))

To increase the amortized efficiency of the find operation we perform path compression



Link all the nodes in the path directly to the root

To increase the amortized efficiency of the find operation we perform path compression



O(α(n)) amortized

Link all the nodes in the path directly to the root

These results go back to...

- R.E. Tarjan. Efficiency of good but not linear set union algorithm. Journal of the ACM, 22:215-225, 1975
- R.E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. Journal of the ACM, 31(2):245-281, 1984

- These works did not consider delete

Purpose of this talk:

Describe a simple and efficient way to incorporate the delete operation.

Goals:

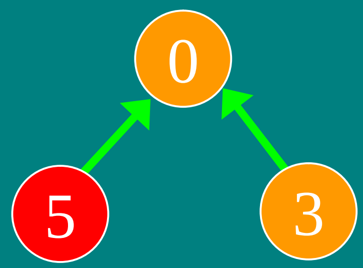
- Delete operation in constant time
- Other operations preserve their current efficiency (time and space)

Applications

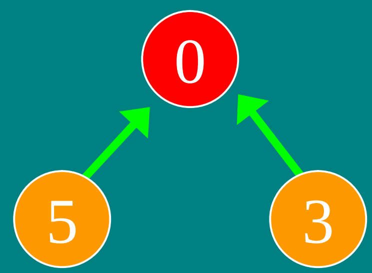
- Implementing meldable priority queues (Haim Kaplan, Nira Shafrir, and Robert Endre Tarjan SODA 2002)
- Implementing uniqueness and ownership transfer in the universe type system (Yoshimi Takano 2007)

Delete operation

Deleting a leaf node is easy (constant time)

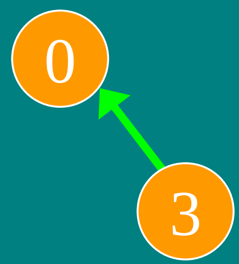


A problem arises when trying to delete a non-leaf node

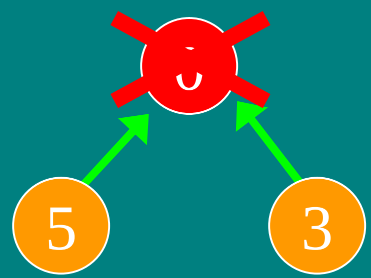


Delete operation

Deleting a leaf node is easy (constant time)

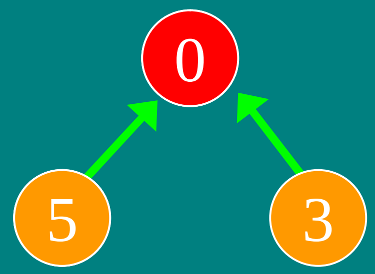


A problem arises when trying to delete a non-leaf node



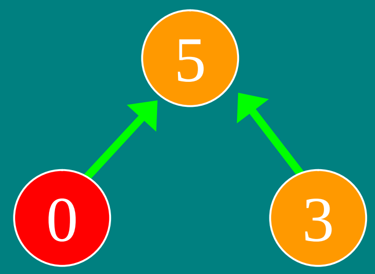
Possible Solution

Find a leaf, switch the elements between the nodes and delete the leaf



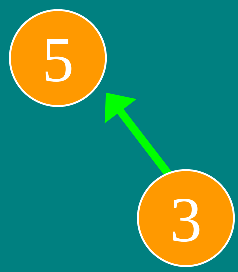
Possible Solution

Find a leaf, switch the elements between the nodes and delete the leaf



Possible Solution

Find a leaf, switch the elements between the nodes and delete the leaf



How to find a leaf?

The answer is that not that simple

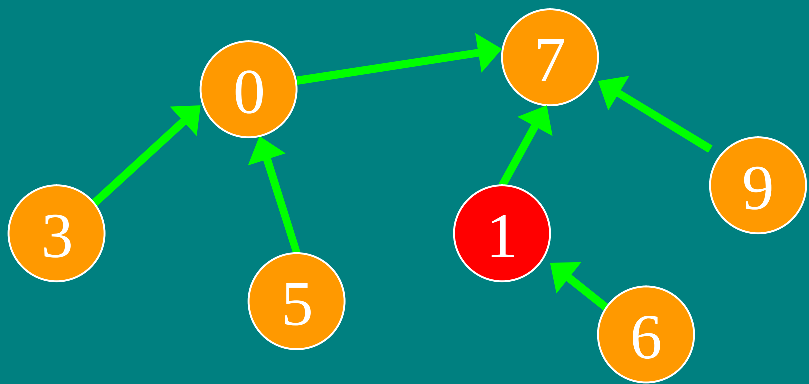
Problems in Finding a Leaf

- The straight-forward idea is to maintain a leaf collection
- Each node should have some kind of a link to the leaf collection
- The problem is how to handle those links during updates
- The straight-forward ideas don't work in constant time

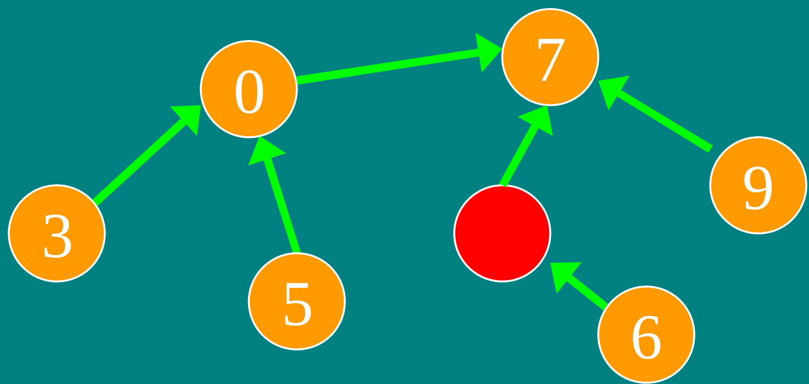
Previous Work

- H. Kaplan, N. Shafrir, R.E. Tarjan: Union-Find with deletions. SODA 2002
- S. Alstrup, I.L. Gørtz, T. Rauhe, M. Thorup, U. Zwick: Union-Find with Constant Time Deletions, ICALP 2005

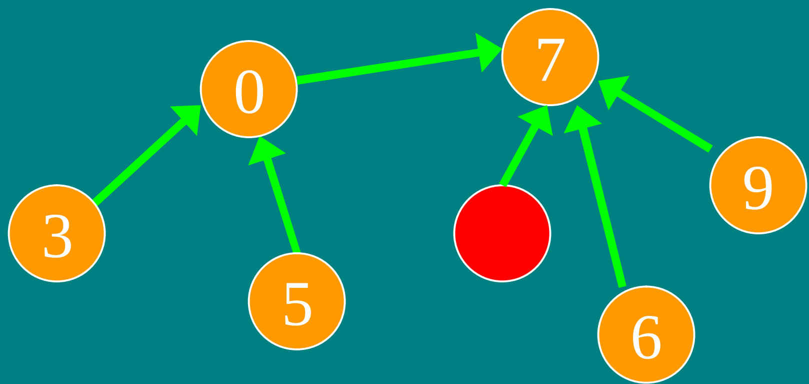
- The solution instead of finding a leaf uses vacant nodes (empty nodes)
- Constant time "tidy" and "local compress" operations for preserving efficiency
- Controlling the amount of the vacant nodes



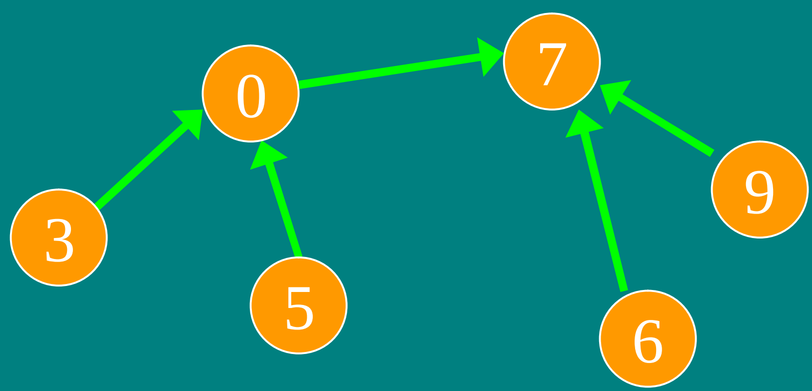
- The solution instead of finding a leaf uses vacant nodes (empty nodes)
- Constant time "tidy" and "local compress" operations for preserving efficiency
- Controlling the amount of the vacant nodes



- The solution instead of finding a leaf uses vacant nodes (empty nodes)
- Constant time "tidy" and "local compress" operations for preserving efficiency
- Controlling the amount of the vacant nodes



- The solution instead of finding a leaf uses vacant nodes (empty nodes)
- Constant time "tidy" and "local compress" operations for preserving efficiency
- Controlling the amount of the vacant nodes



Our Solution

Finding a leaf in a constant worst-case time

To accomplish that we extend the data structure as follows:

- Each node holds an ordered doubly linked list of its children
- The root holds a doubly linked list of a non leaf children
- Each tree holds a cyclic doubly linked list of the tree nodes in right-to-left DFS order

Tree Nodes in DFS Order

The predecessor of a given node can be one of two:

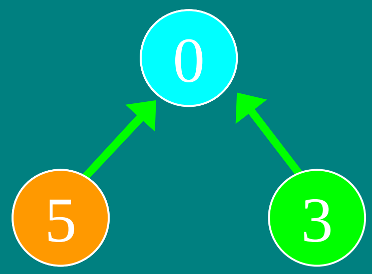
- The parent node in the tree

DFS

0

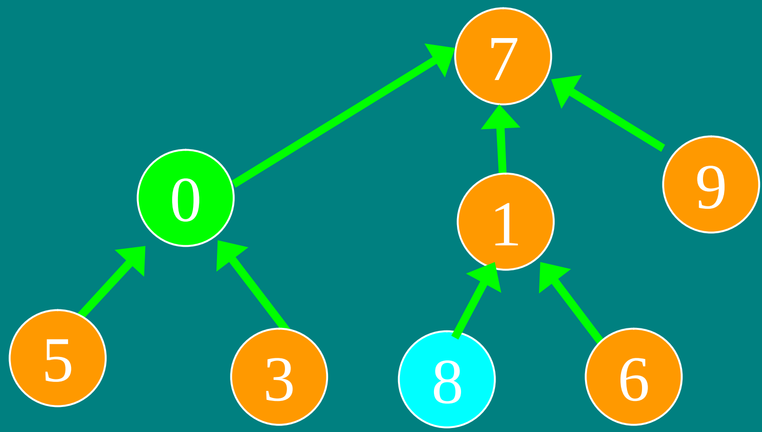
3

5



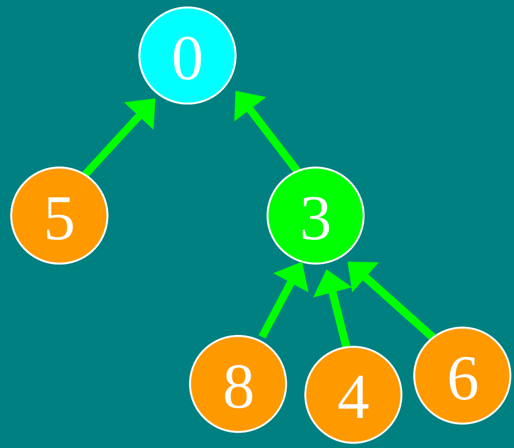
- The leftmost leaf in the sub tree of the right sibling of the given node

DFS
7
9
1
6
8
0
3
5



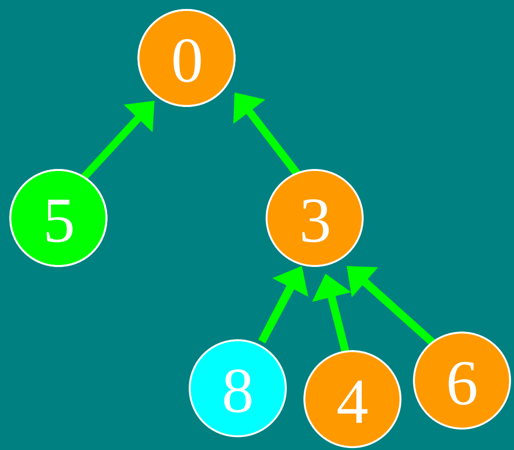
- The predecessor is a parent or a leaf

DFS
0
3
6
4
8
5



- If it is a parent we can examine the left sibling of the node

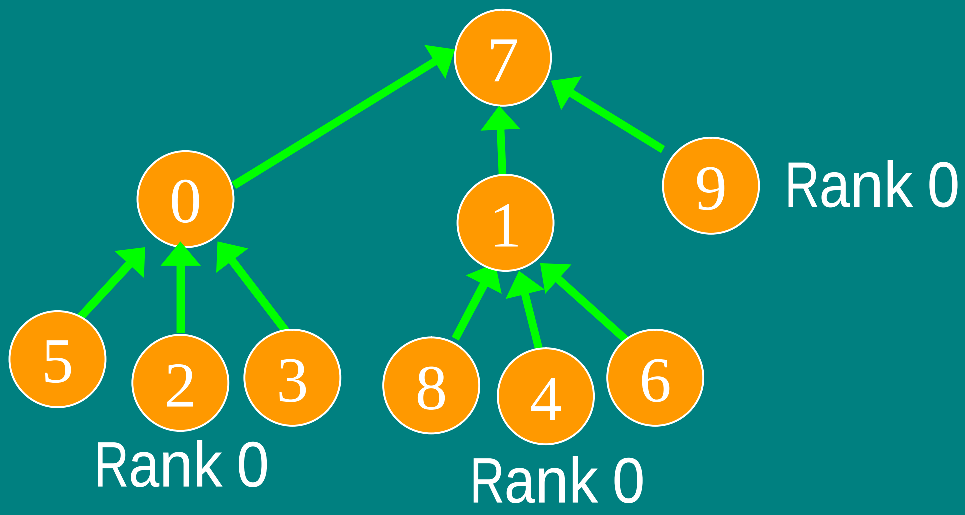
DFS
0
3
6
4
8
5



Full or Reduced Trees

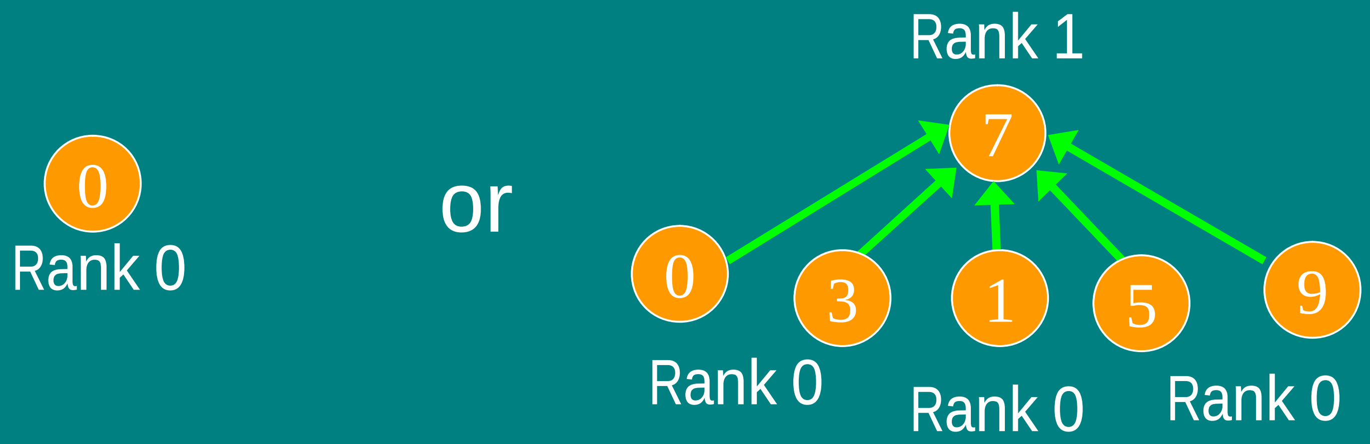
We maintain certain invariants to achieve the desired efficiency. Every tree will be either

- Full – each node is either a leaf of rank 0, or a parent with at least three children



or

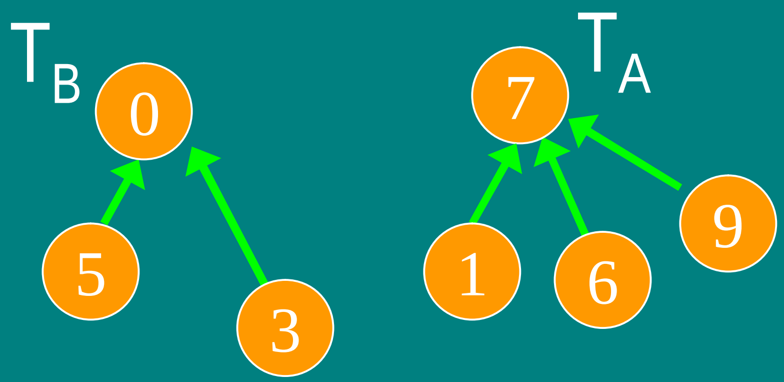
- Reduced – a single node of rank 0,
- or a root of rank 1 with leaves of rank 0



Implementing Union

One of the trees is of size < 4 :

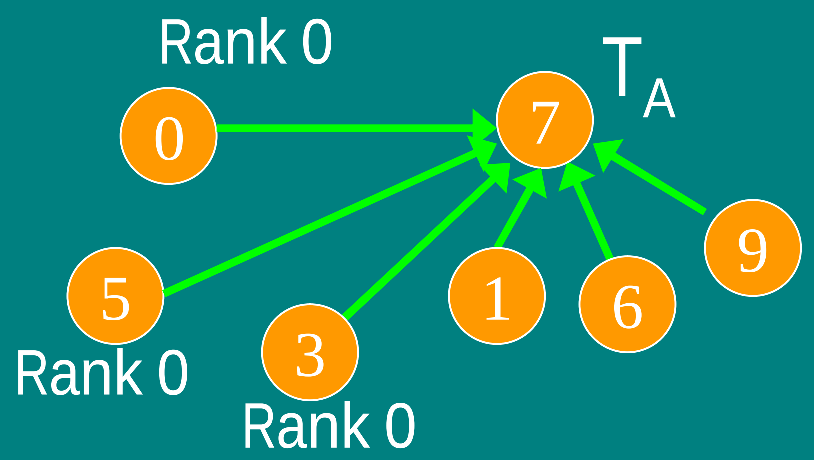
Hang all its nodes on the other root



Implementing Union

One of the trees is of size < 4 :

Hang all its nodes on the other root

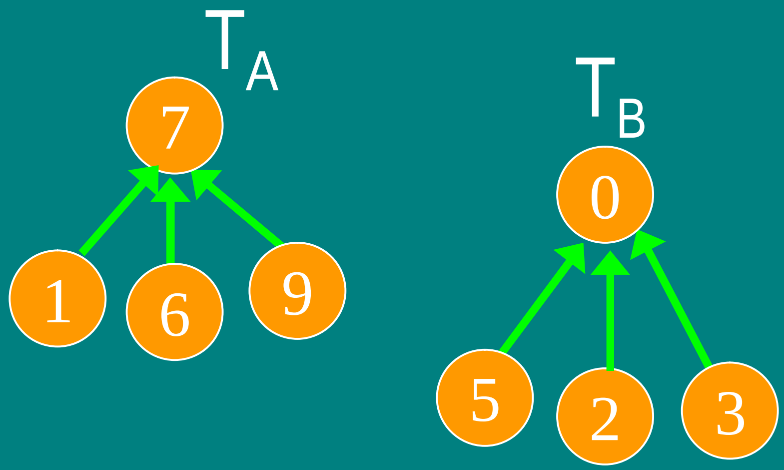


Both trees are of size ≥ 4 :

Union by rank

Update our additional lists

DFS	DFS
7	0
9	3
6	2
1	5



Both trees are of size ≥ 4 :

Union by rank

Update our additional lists

DFS

7

0

3

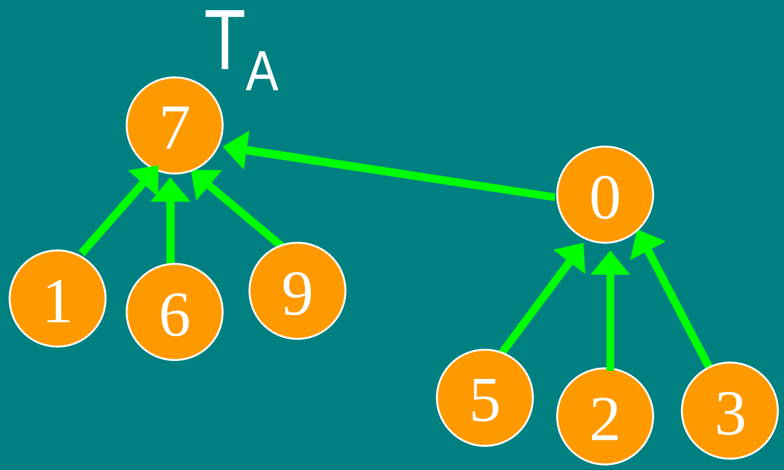
2

5

9

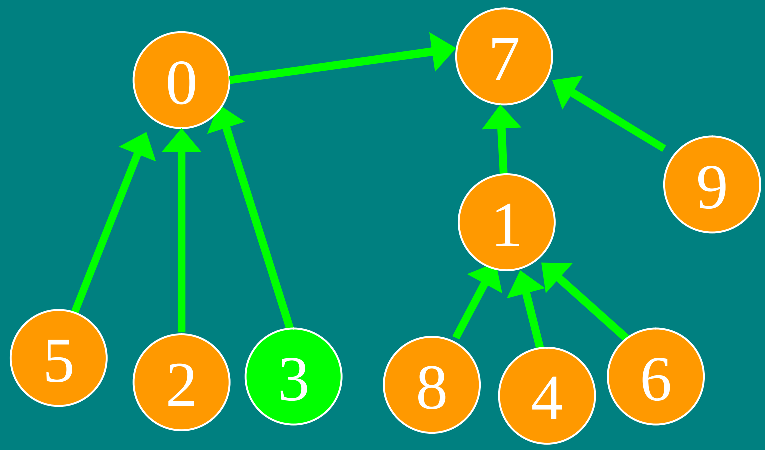
6

1



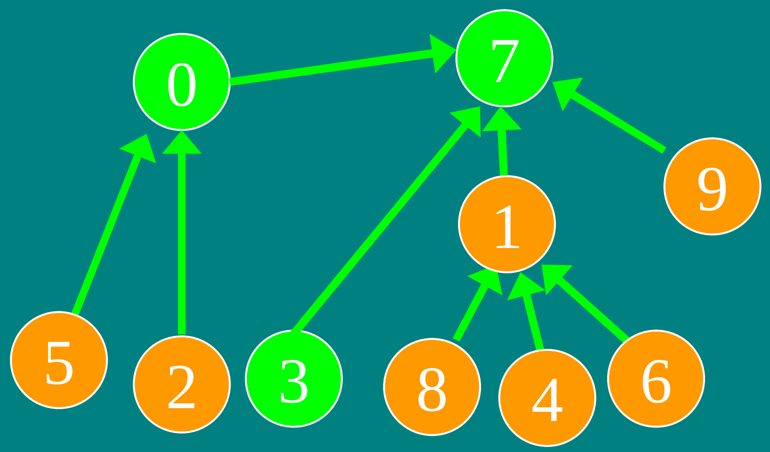
Implementing Find

- Instead of path compression we use *path splitting* [Tarjan and van Leeuwen]
- Each node in the path is moved from its parent to its grand-parent
- If the parent now has less than three children we move them as well



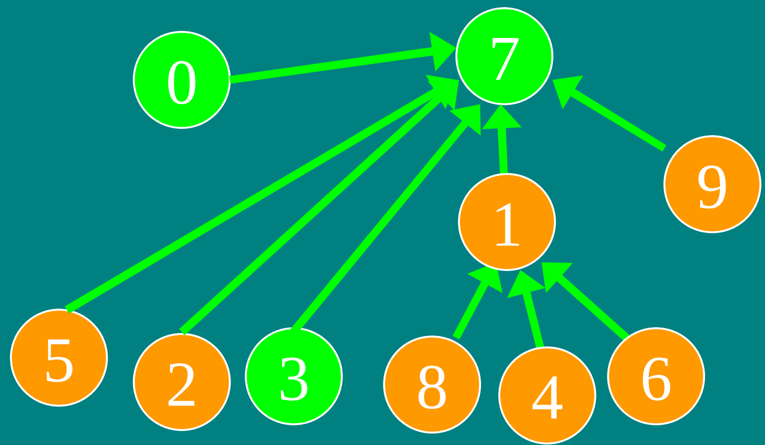
Implementing Find

- Instead of path compression we use *path splitting* [Tarjan and van Leeuwen]
- Each node in the path is moved from its parent to its grand-parent
- If the parent now has less than three children we move them as well



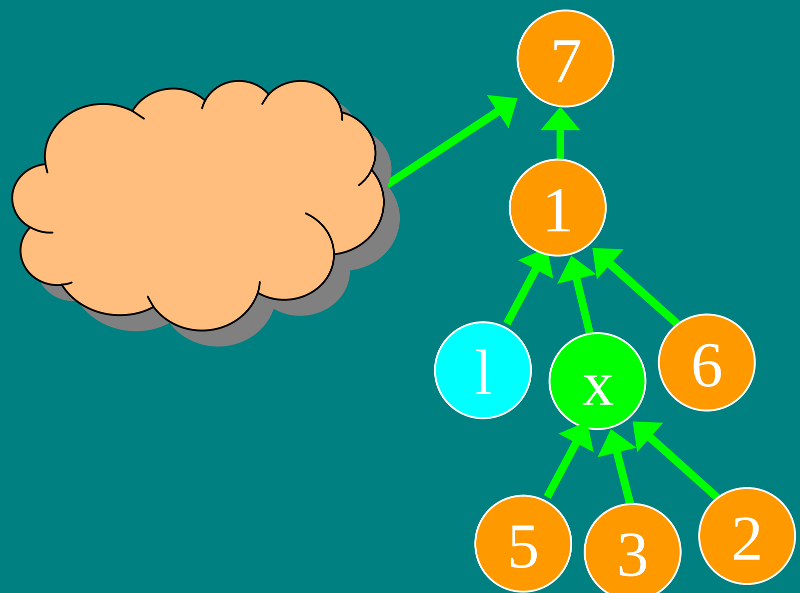
Implementing Find

- Instead of path compression we use *path splitting* [Tarjan and van Leeuwen]
- Each node in the path is moved from its parent to its grand-parent
- If the parent now has less than three children we move them as well



Fixing The DFS Order

- If node x has a left sibling (l), it means that the sub tree that starts at x is represented in the DFS order by segment [x, l)
- It is simple to disconnect the segment and insert it before the parent of x in the DFS order

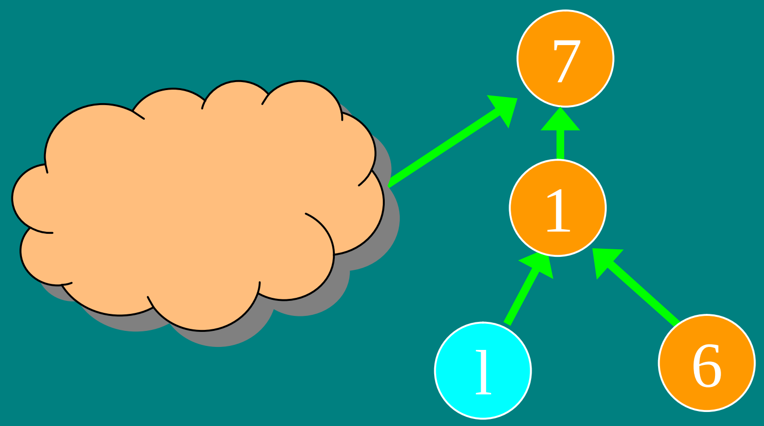


DFS

7
1
6
x
2
3
5
l
...

Fixing The DFS Order

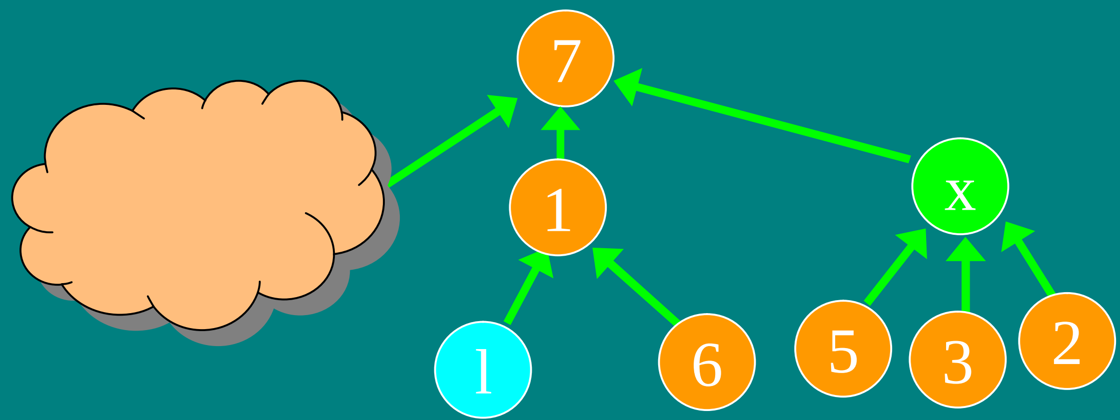
- If node x has a left sibling (l), it means that the sub tree that starts at x is represented in the DFS order by segment [x, l)
- It is simple to disconnect the segment and insert it before the parent of x in the DFS order



<u>DFS</u>
7
1
6
l
...

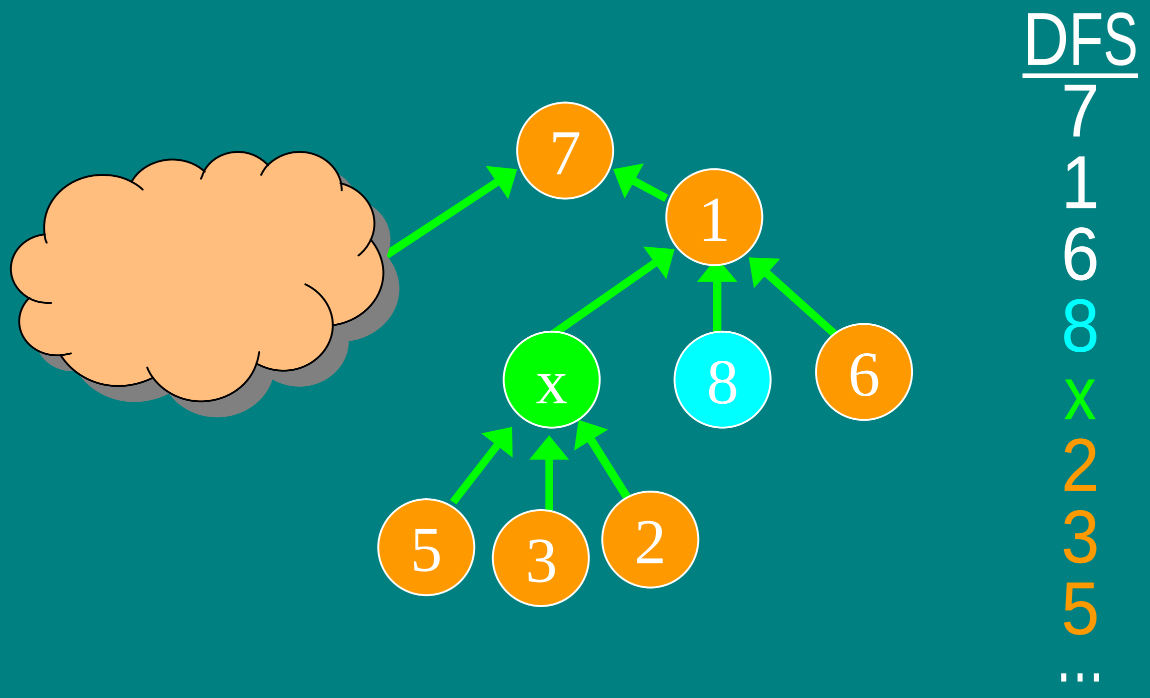
Fixing The DFS Order

- If node x has a left sibling (l), it means that the sub tree that starts at x is represented in the DFS order by segment [x, l)
- It is simple to disconnect the segment and insert it before the parent of x in the DFS order

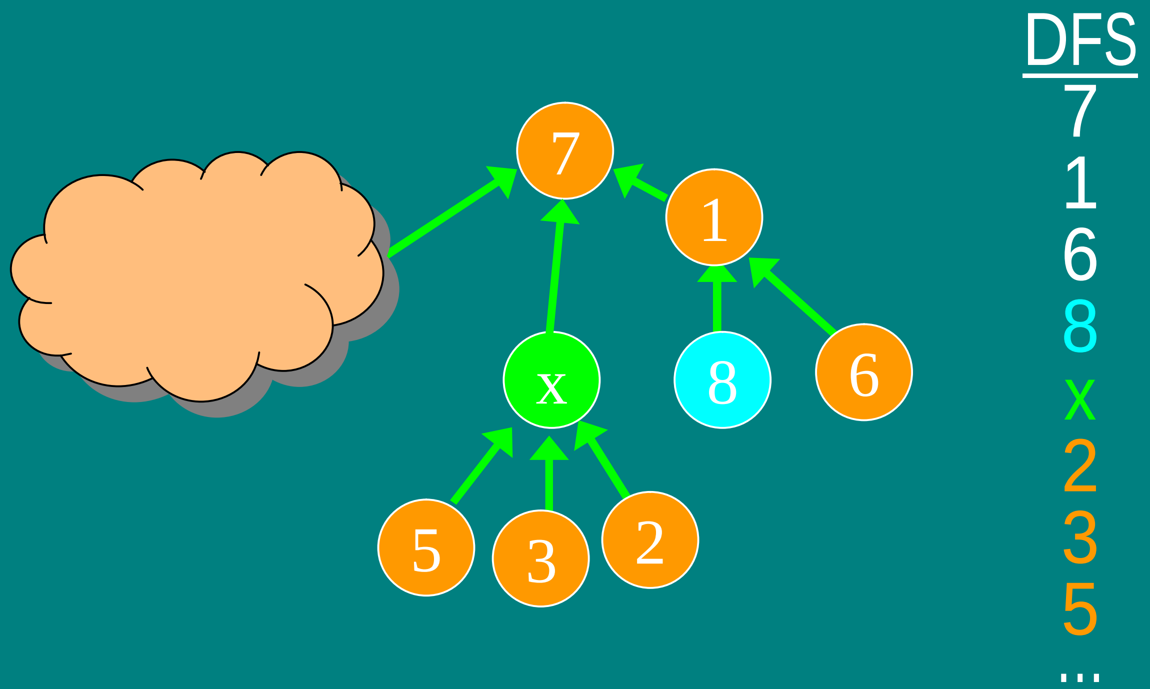


DFS
7
x
2
3
5
1
6
l
...

- If node x is the leftmost child of its parent, we do not have to change the DFS order at all



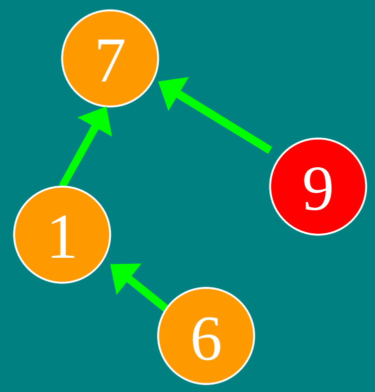
- If node x is the leftmost child of its parent, we do not have to change the DFS order at all



Implementing Delete

Tree is of size ≤ 4 :

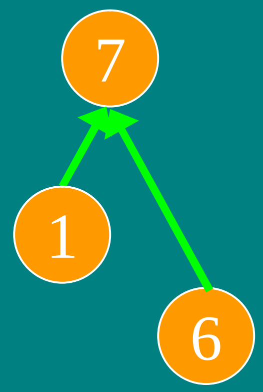
Rebuild the tree in to reduced from



Implementing Delete

Tree is of size ≤ 4 :

Rebuild the tree in to reduced from



Tree of size > 4 :

Find a leaf, switch elements with the leaf node

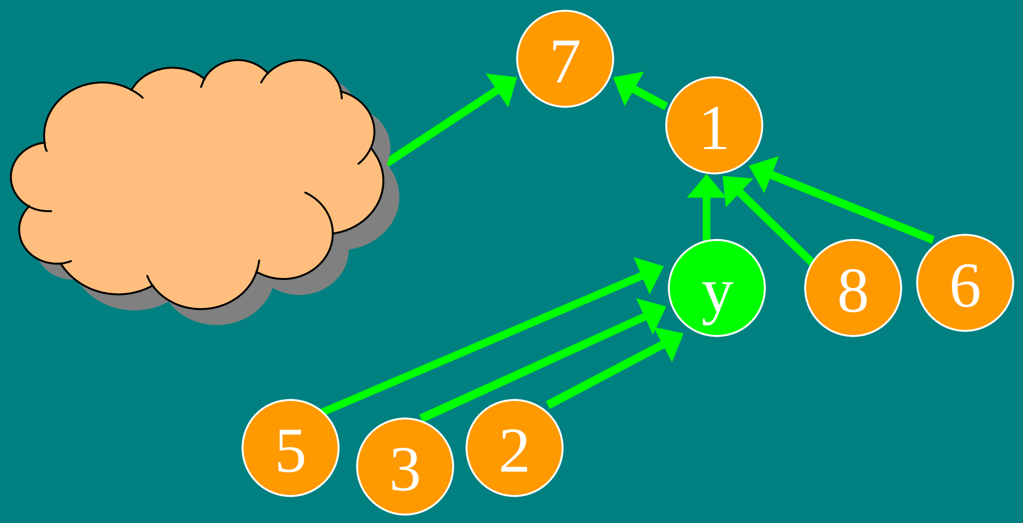
Delete the leaf node

Update additional lists

If the tree is not reduced as the result of the deletion apply local rebuild to the parent of the deleted node (or the root)

Local Rebuild

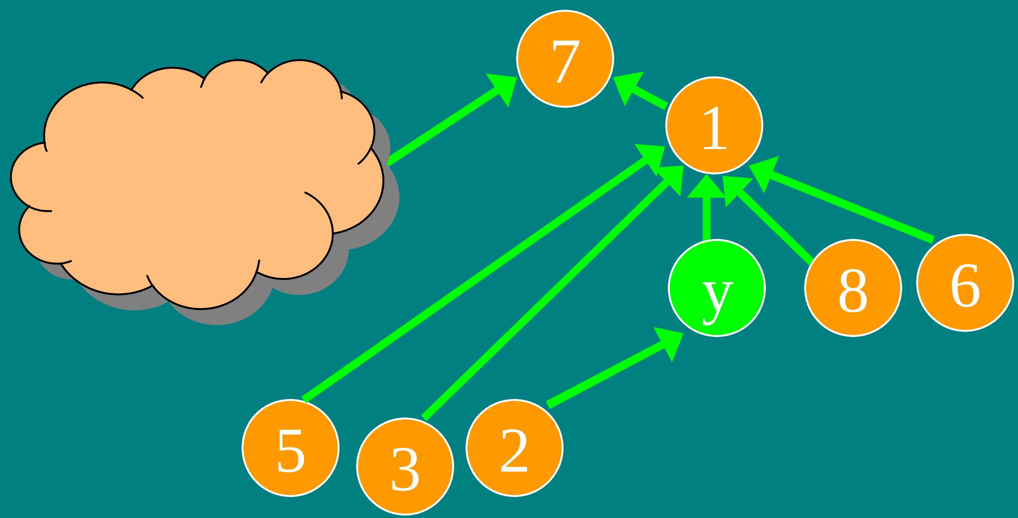
If node y is not the root, move 2 leftmost children of y to its parent



If less than 3 children remain, move the rest of the children as well

Local Rebuild

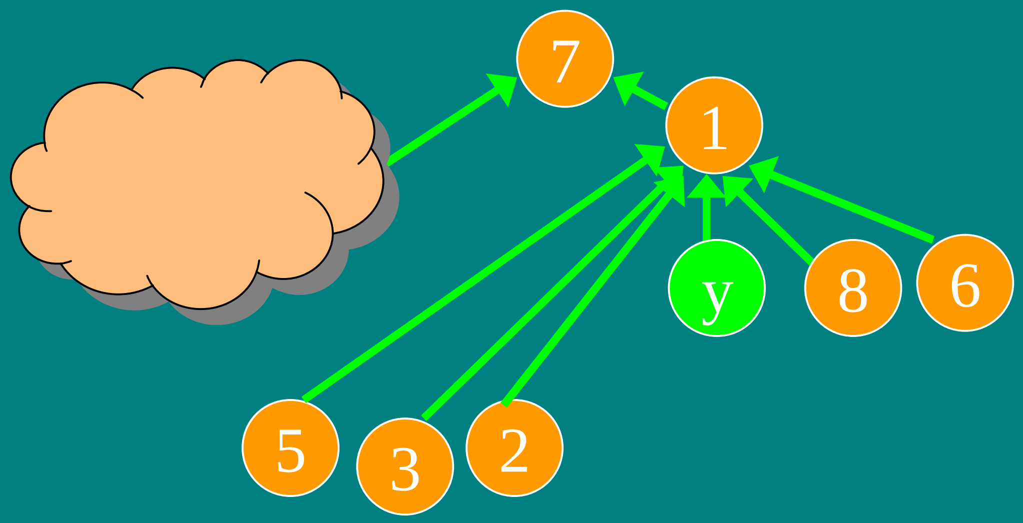
If node y is not the root, move 2 leftmost children of y to its parent



If less than 3 children remain, move the rest of the children as well

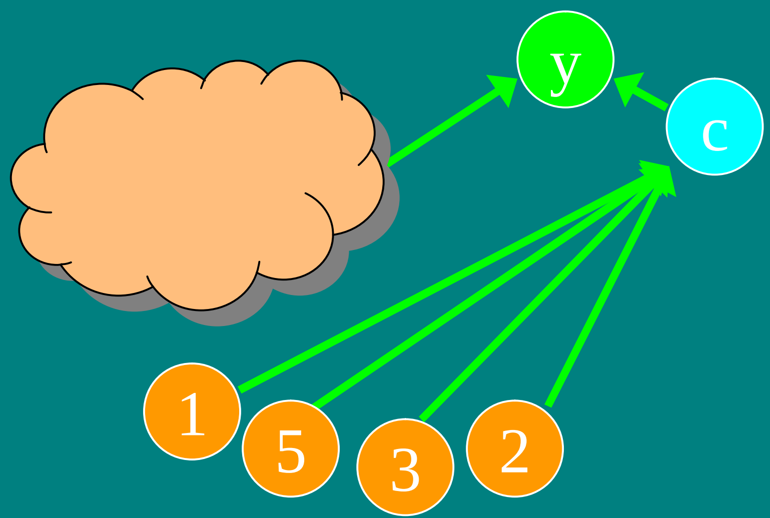
Local Rebuild

If node y is not the root, move 2 leftmost children of y to its parent



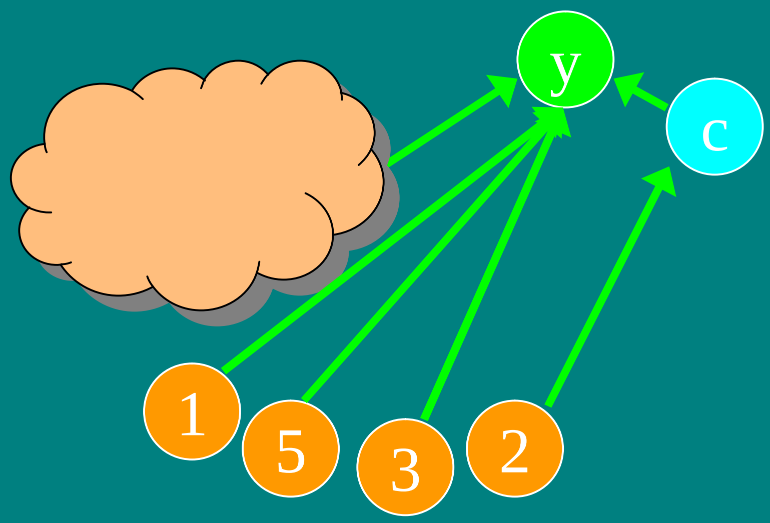
If less than 3 children remain, move the rest of the children as well

If node y is the root and node c is a non-leaf child of y, move three leftmost children of c to y



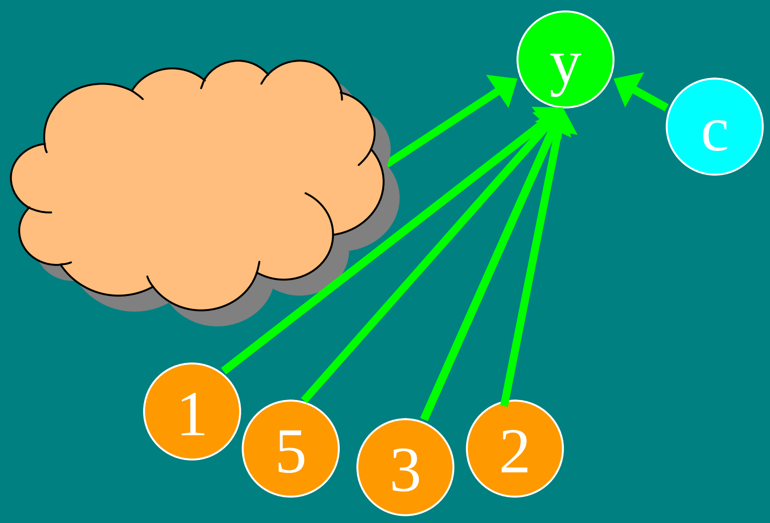
If less than 3 children remain, move the rest of the children as well

If node y is the root and node c is a non-leaf child of y, move three leftmost children of c to y



If less than 3 children remain, move the rest of the children as well

If node y is the root and node c is a non-leaf child of y, move three leftmost children of c to y



If less than 3 children remain, move the rest of the children as well

Analysis

Our asymptotic worst case and amortized complexity is similar to that of Alstrup et al.

We actually reuse parts of their analysis.

The notable difference is a conceptual simplification (no vacant nodes) and a much simpler “local compression”

+ a simplified analysis (with smaller constant factors)

Future Research

- Can we find a leaf in the sub tree of the node requested for deletion?
- Can we reduce the memory usage of the data structure?

Ευχαριστω

Thank You