ECE 250 Algorithms and Data Structures

Splay Trees

Douglas Wilhelm Harder, M.Math. LEL
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

Outline

This topic covers splay trees

- A binary search tree
- An alternate idea to optimizing run times
- A possible height of $\mathrm{O}(n)$ but amortized run times of $\Theta(\ln(n))$
- Each access or insertion moves that node to the root
- Operations are *zig-zag* and *zig-zig*
- Similar to, but different from, AVL trees

Background

AVL trees and red-black trees are binary search trees with logarithmic height

- This ensures all operations are $\mathrm{O}(\ln(n))$

An alternative to maintaining a height logarithmic with respect to the number of nodes, an alternative idea is to make use of an old maxim:

> Data that has been recently accessed is more likely to
> be accessed again in the near future.

Background

Accessed nodes could be rotated or *splayed* to the root of the tree:

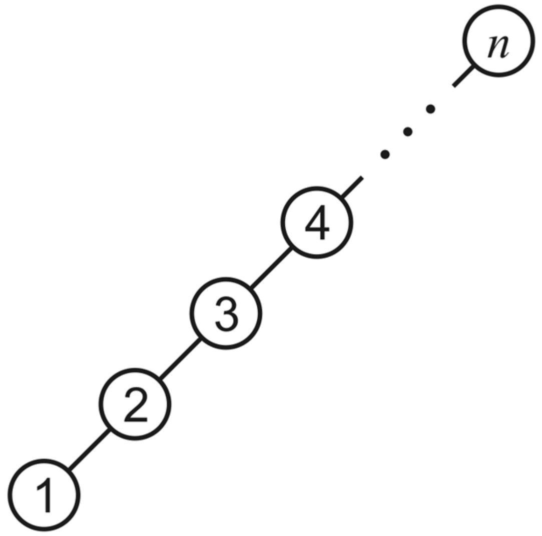
- Accessed nodes are splayed to the root during the count/find operation
- Inserted nodes are inserted normally and then splayed
- The parent of a removed node is splayed to the root

Invented in 1985 by Daniel Dominic Sleator and Robert Endre Tarjan

Insertion at the Root

Immediately, inserting at the root makes it clear that we will still have access times that are $O(n)$:

- Insert the values 1, 2, 3, 4, …, n, in that order

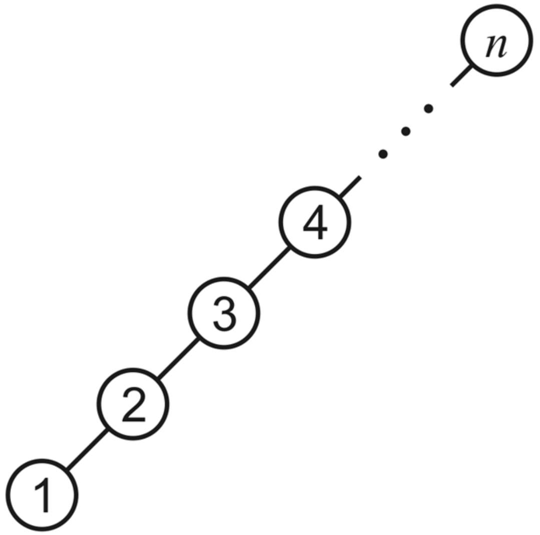


- Now, an access to 1 requires that a linked list be traversed

Inserting at the Root

However, we are interested in amortized run times:

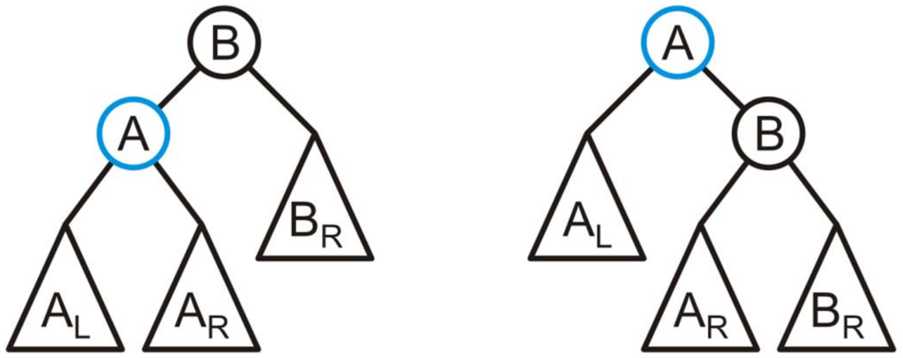
- We only require that n accesses have $\Theta(n \ln(n))$ time
- Thus $O(\ln(n))$ of those accesses could still be $O(n)$



Inserting at the Root

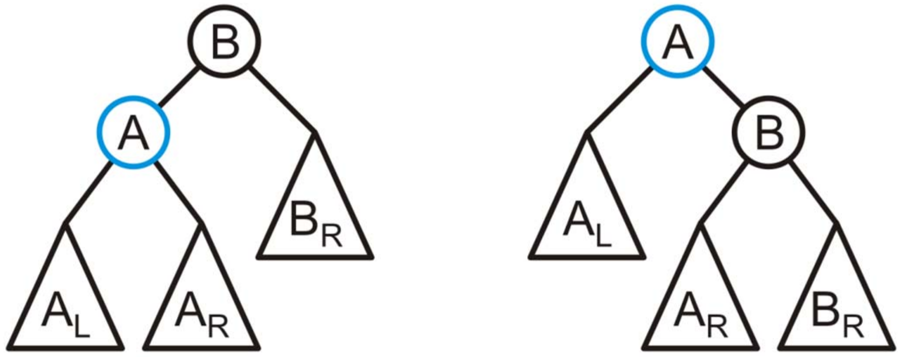
Before we consider insertions, how can we simply move an access node to the root?

– We could consider AVL rotations, the simplest of which is:



Single Rotations

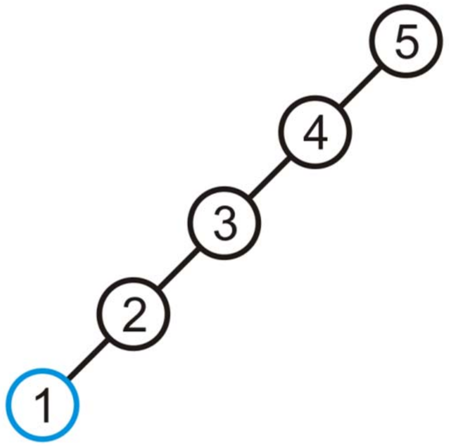
Unfortunately, as we will see, using just single rotations **does not work**



Single Rotations

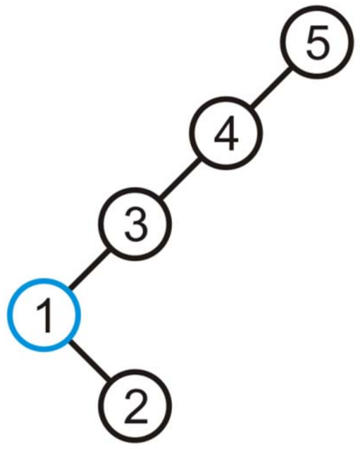
Consider this splay tree with five entries

- They were inserted in the order 1, 2, 3, 4 and 5
- Let us access 1 by find it and then rotating it back to the root



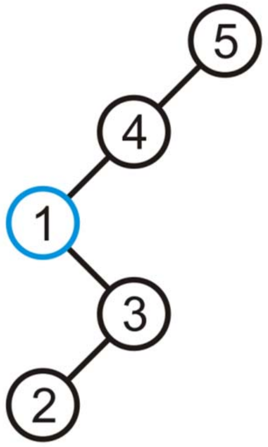
Single Rotations

Rotating 1 and 2



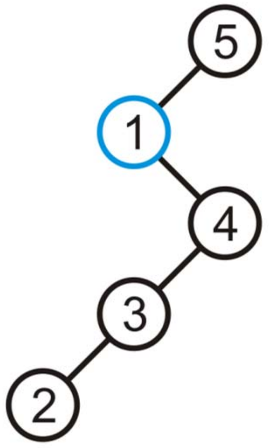
Single Rotations

Rotating 1 and 3



Single Rotations

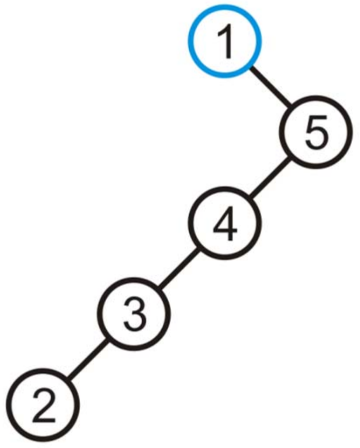
Rotating 1 and 4



Single Rotations

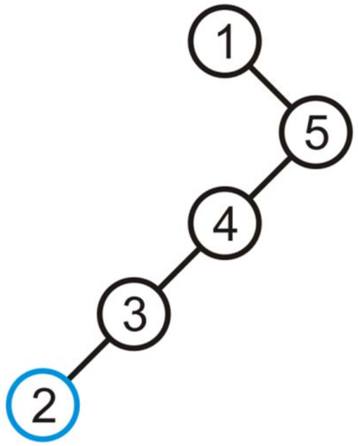
Rotating 1 and 5

- The result still looks like a linked list



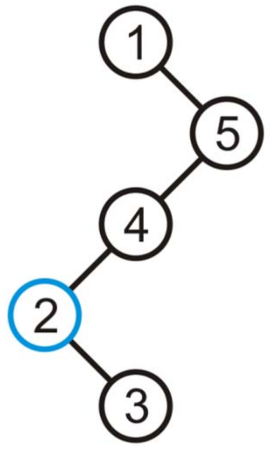
Single Rotations

Accessing 2 next doesn't do much



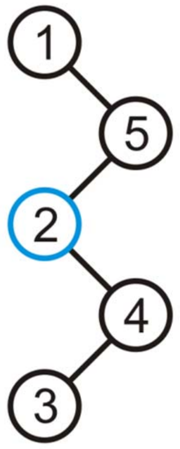
Single Rotations

Accessing 2 next doesn't do much



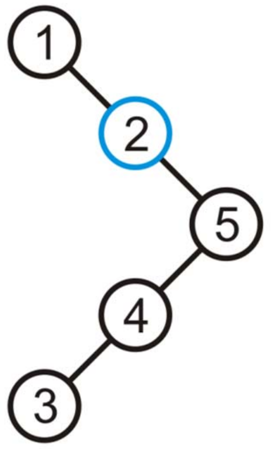
Single Rotations

Accessing 2 next doesn't do much



Single Rotations

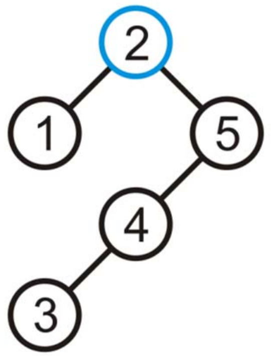
Accessing 2 next doesn’t do much



Single Rotations

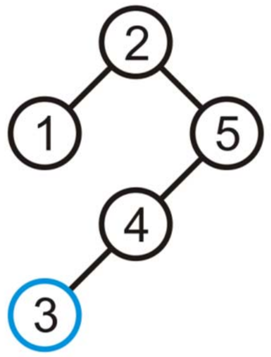
Accessing 2 next doesn't do much

- The resulting tree is shallower by only 1



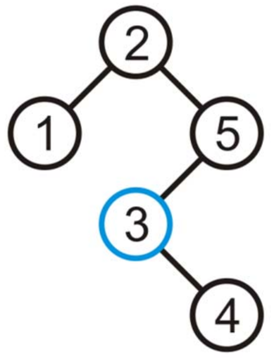
Single Rotations

Accessing 3 isn't significant, either



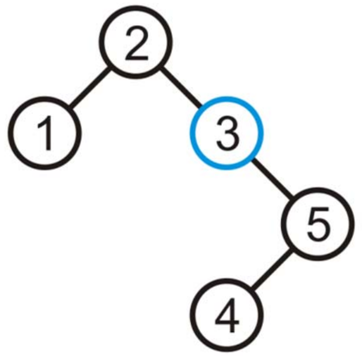
Single Rotations

Accessing 3 isn't significant, either



Single Rotations

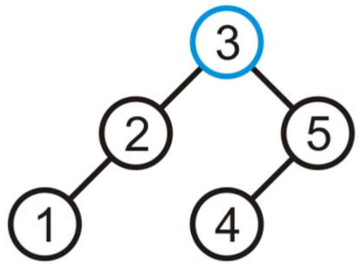
Accessing 3 isn't significant, either



Single Rotations

Accessing 3 isn't significant, either

– Essentially, it is two linked lists and the left sub-tree is turning into the original linked list



Single Rotations

In a general splay tree created in the order

$$1, 2, 3, 4, \ldots, n$$

and then accessed repeated in the order

$$1, 2, 3, 4, \ldots, n$$

will require

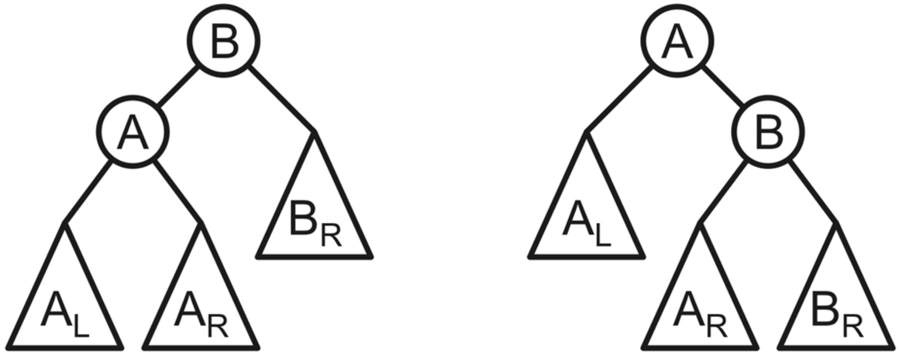
$$\sum_{k=1}^{n}(n-k) = n^2 - \sum_{k=1}^{n} k = n^2 - \frac{n(n+1)}{2} = \frac{n(n-1)}{2} = \mathrm{O}(n^2)$$

comparisons—an amortized run time of $\mathrm{O}(n)$

Single Rotations

Thus, a single rotation will not do

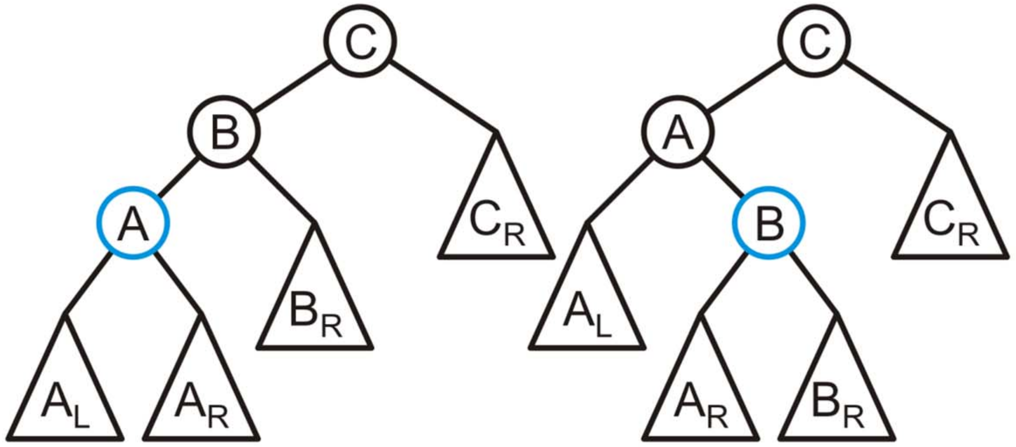
– It can convert a linked list into a linked list



Depth-2 Rotations

Let’s try rotations with entries at depth 2

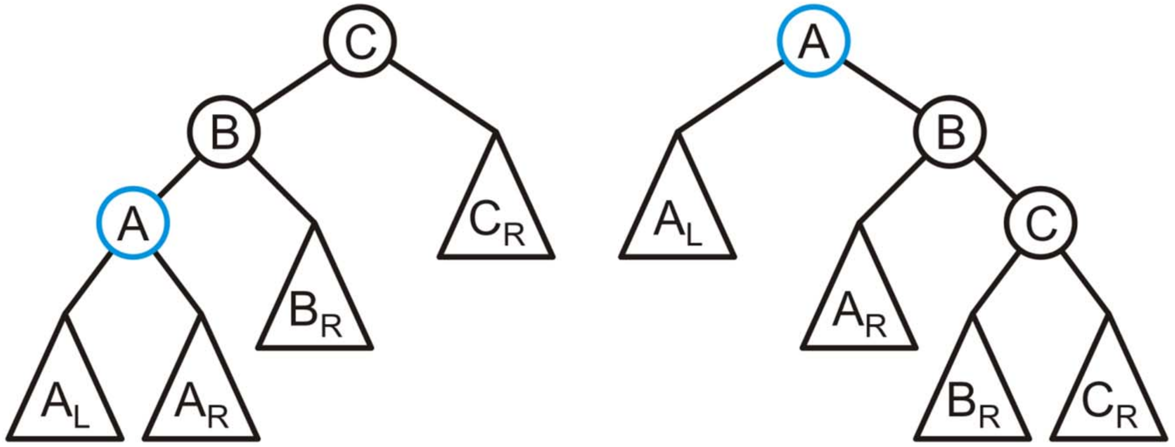
- Suppose we are accessing A on the left and B on the right



Depth-2 Rotations

In the first case, two rotations at the root bring A to the root

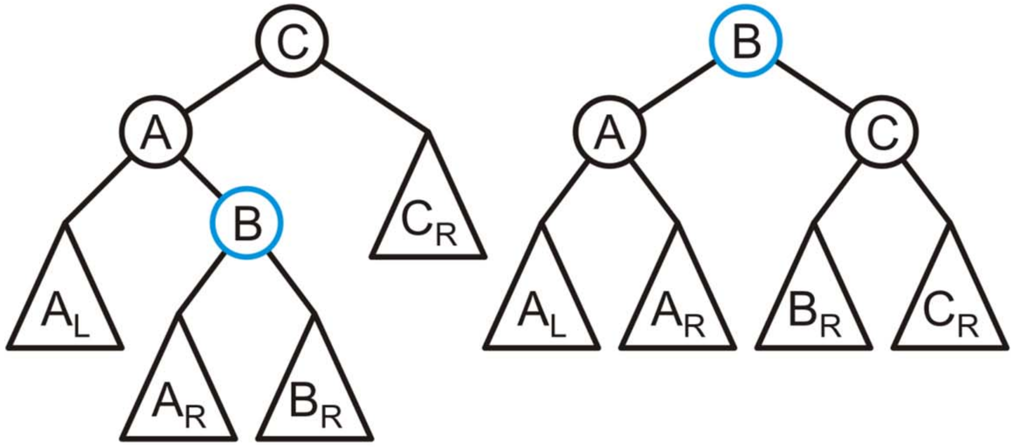
- We will call this a *zig-zig rotation*



Depth-2 Rotations

In the second, two rotations bring B to the root

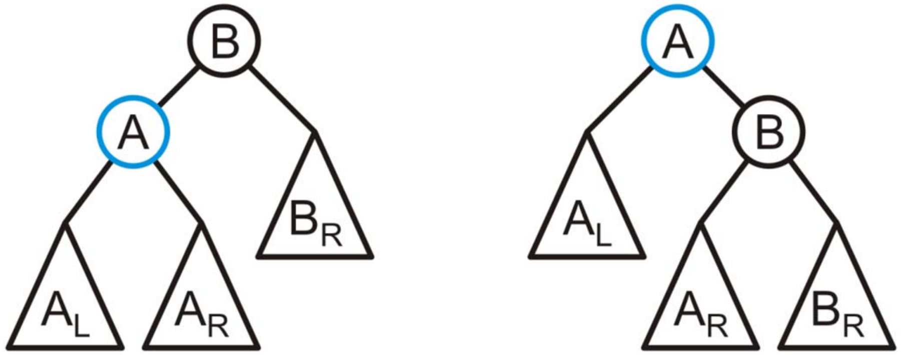
- It doesn't seem we've done a lot…
- We will call this a *zig-zag rotation*



Depth-2 Rotations

If the accessed node is a child of the root, we must revert to a single rotation:

– A *zig* rotation



Operations

Accessing any node splays the node to the root

Inserting a new element into a splay tree follows the binary search tree model:

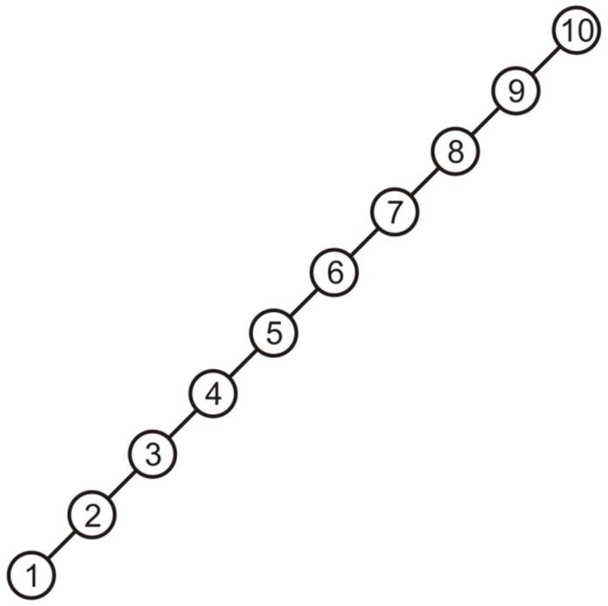
– Insert the node as per a standard binary search tree
– Splay the object to the root

Removing a node also follows the pattern of a binary search tree

– Copy the minimum of the right sub-tree
– Splay the parent of the removed node to the root

Examples

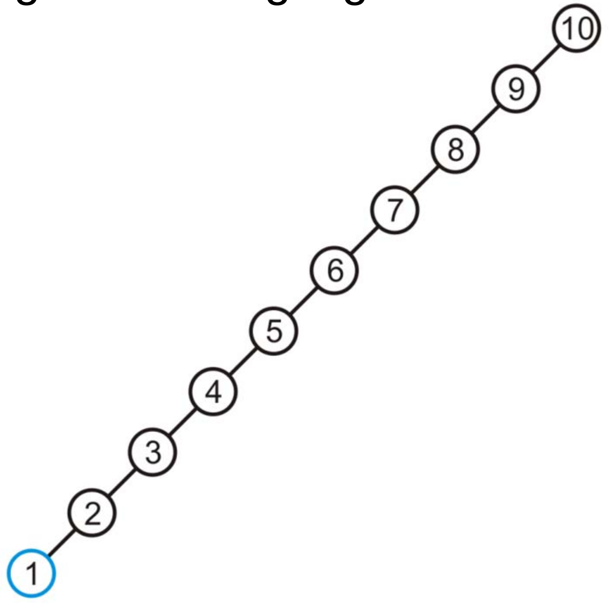
With a little consideration, it becomes obvious that inserting 1 through 10, in that order, will produce the splay tree



Examples

We will repeatedly access the deepest node in the tree

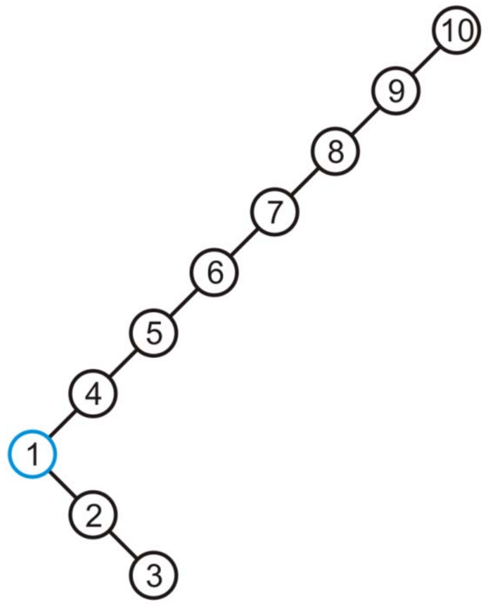
- With each operation, this node will be splayed to the root
- We begin with a zig-zig rotation



Splay Trees

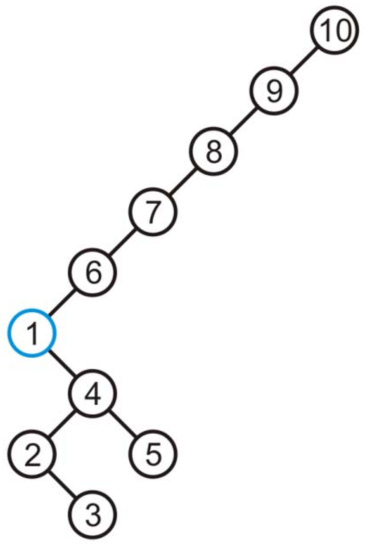
Examples

This is followed by another zig-zig operation...



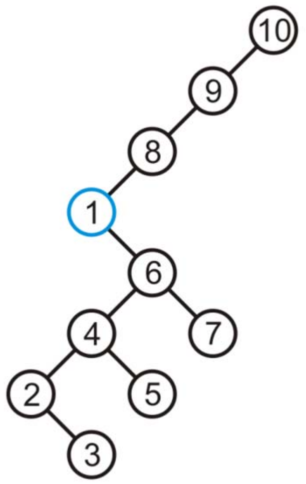
Examples

...and another



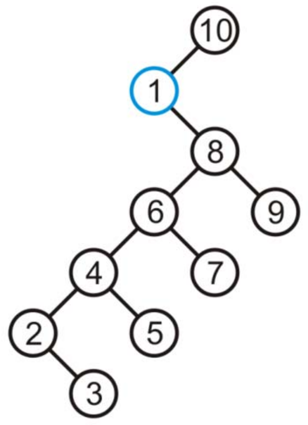
Examples

...and another



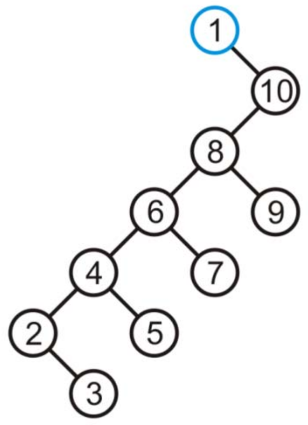
Examples

At this point, this requires a single zig operation to bring 1 to the root



Examples

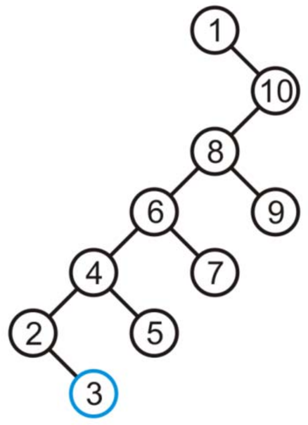
The height of this tree is now 6 and no longer 9



Examples

The deepest node is now 3:

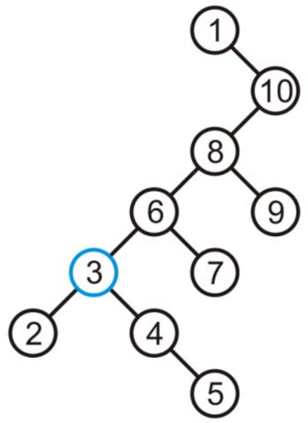
- This node must be splayed to the root beginning with a zig-zag operation



Examples

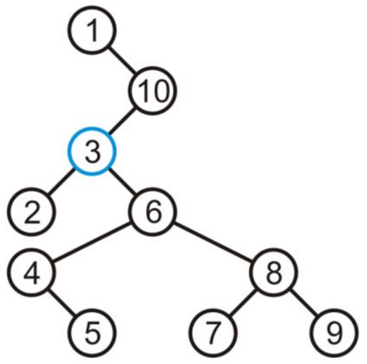
The node 3 is rotated up

– Next we require a zig-zig operation



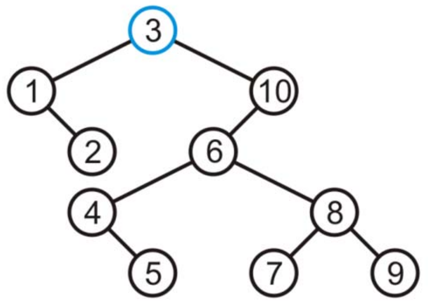
Examples

Finally, to bring 3 to the root, we need a zig-zag operation



Examples

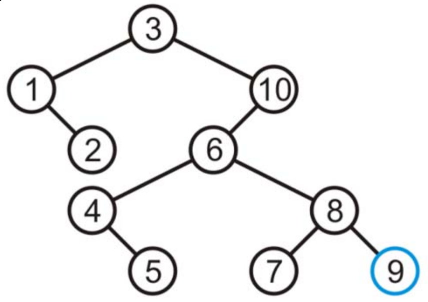
The height of this tree is only 4



Examples

Of the three deepest nodes, 9 requires a zig-zig operation, so will access it next

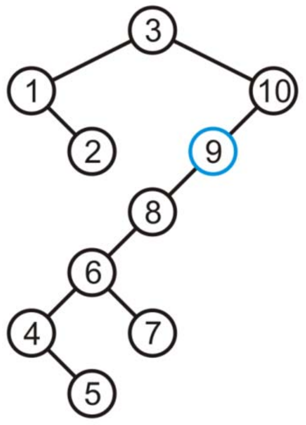
– The zig-zig operation will push 6 and its left sub-tree down



Examples

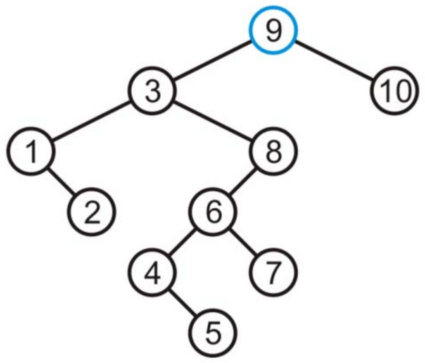
This is closer to a linked list; however, we're not finished

- A zig-zag operation will move 9 to the root



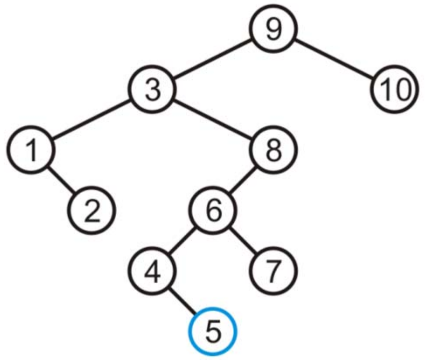
Examples

In this case, the height of the tree is now greater: 5



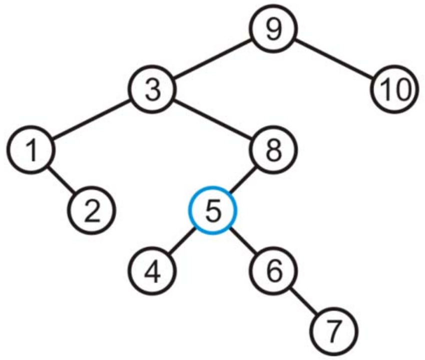
Examples

Accessing the deepest node, 5, we must begin with a zig-zag operation



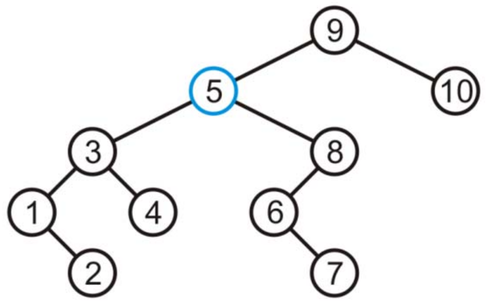
Examples

Next, we require a zig-zag operation to move 5 to the location of 3



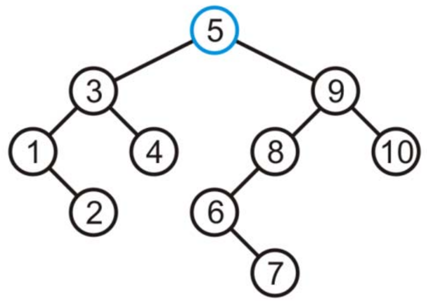
Examples

Finally, we require a single zig operation to move 5 to the root



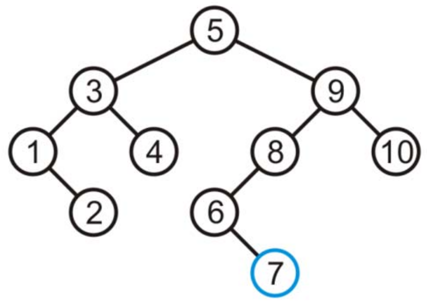
Examples

The height of the tree is 4; however, 7 of the nodes form a perfect tree at the root



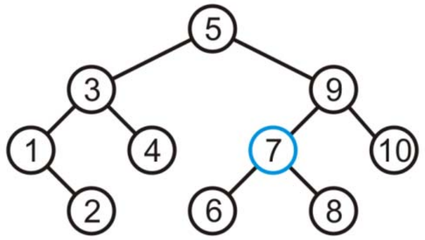
Examples

Accessing 7 will require two zig-zag operations



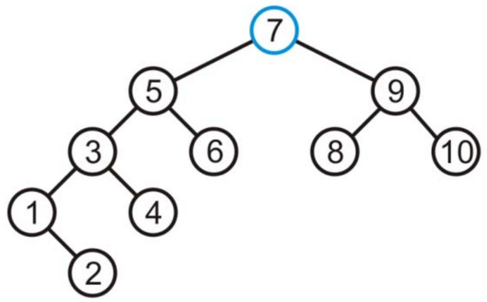
Examples

The first zig-zag moves it to depth 2



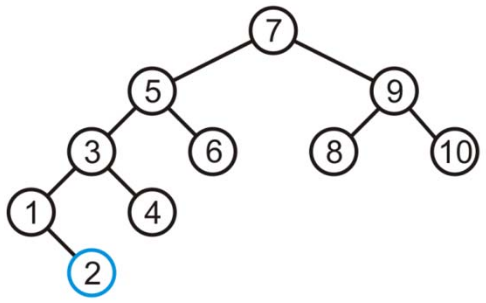
Examples

7 is promoted to the root through a zig-zag operation



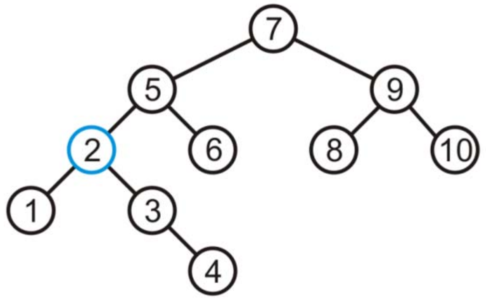
Examples

Finally, accessing 2, we first require a zig-zag operation



Examples

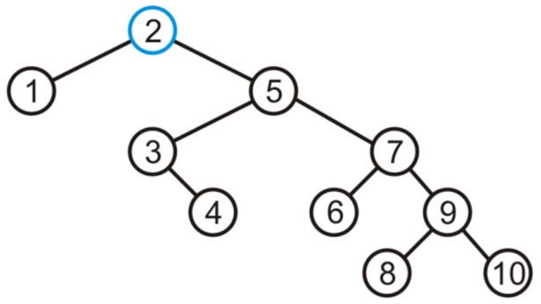
This now requires a zig-zig operation to promote 2 to the root



Examples

In this case, with 2 at the root, 3-10 must be in the right sub-tree

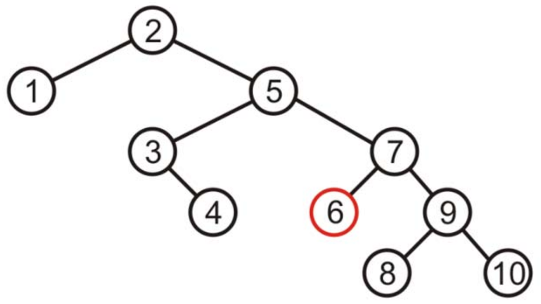
– The right sub-tree happens to be AVL balanced



Examples

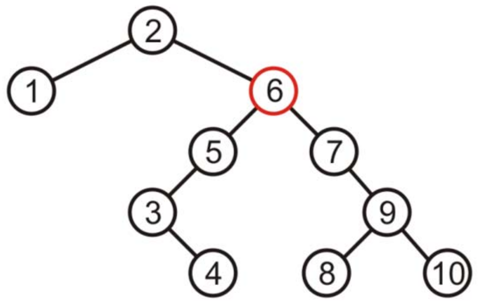
To remove a node, for example, 6, splay it to the root

- First we require a zig-zag operation



Examples

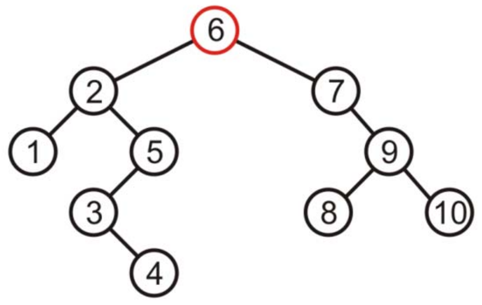
At this point, we need a zig operation to move 6 to the root



Examples

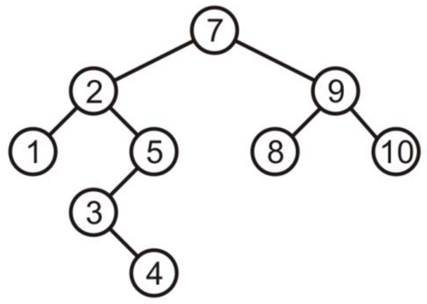
We will now copy the minimum element from the right sub-tree

- In this case, the node with 7 has a single sub-tree, we will simply move it up



Examples

Thus, we have removed 6 and the resulting tree is, again, reasonably balanced



Performance

It is very difficult with small trees to demonstrate the amortized logarithmic behaviour of splay trees

The original ACM article proves the *balance theorem*:

> The run time of performing a sequence of m operations on a splay tree with n nodes is $\mathrm{O}(\ m(1 + \ln(n)) + n \ln(n)\ )$.

Therefore the run time for a splay tree is comparable to any balanced tree assuming at least n operations

Performance

From the time of introducing splay trees (1985) up till today
the following conjecture (among others) remains unproven.

Dynamic optimality conjecture[2]

Consider any sequence of successful accesses on an *n*-node search tree. Let A be any algorithm that carries out each access by traversing the path from the root to the node containing the accessed item, at a cost of one plus the depth of the node containing the item, and that between accesses performs an arbitrary number of rotations anywhere in the tree, at a cost of one per rotation. Then the total time to perform all the accesses by splaying is no more than O(*n*) plus a constant times the time required by algorithm A.

Performance

The ECE 250 web site has an implementation of splay trees at

http://ece.uwaterloo.ca/~ece250/Algorithms/Splay_trees/

It allows the user to export trees as SVG files

Comparisons

Advantages:

– The amortized run times are similar to that of AVL trees and red-black trees

– The implementation is easier

– No additional information (height/colour) is required

Disadvantages:

– The tree will change with read-only operations

Summary

This topic covers splay trees

- A binary search tree
- Splay accessed or inserted nodes to the root
- The height is $O(n)$ but amortized run times of $\Theta(\ln(n))$ for $\Omega(n)$ operations
- Operations are termed *zig*, *zig-zag* and *zig-zig*
- Requires no additional memory

References

[1] Weiss, Data Structures and Algorithm Analysis in C++, 3rd Ed., Addison Wesley, §4.5, pp.149-58.

[2] Daniel D. Sleator and Robert E. Tarjan, "Self-Adjusting Binary Search Trees", Journal of the ACM 32 (3), 1985, pp.652-86.

Usage Notes