-CPSC 3120 Midterm Exam Study Guide

Brendan McGuire (bmmcgui@clemson.edu), Patrick Smathers (psmathe@clemson.edu)

Disclaimer: While I attempt to be accurate, I am a student like you. Please let me know if you notice anything incorrect or misleading in this document, and I can correct it!

Characterizing Algorithms **2**
- RAM Model of Computation 2
- Asymptotic Notation 2
  - Big O Notation 2

Amortized Analysis **3**
- Accounting Method 3
  - Stack Example 3
- Potential Method 4
  - Stack Example 4

Union-Find Data Structures **5**
- Overview of Data Structure 5
- List-Based Implementation 5
- Tree-Based Implementation 7
  - Path Compression 8
    - Ackerman's Function 8

Binary Trees **9**
- Properties of Binary Trees (We Care About) 9
  - Balance 9
  - Proper/Full 9
  - Complete 9
- Operations (We Care About) 9
- AVL Trees 10
  - Rotations 10
- Red Black Trees 11
  - Insertion Operation 11
  - Fix Up 11
- Case 4: Parent is Red, Parent's Sibling is Black, New Node is Inner Grandchild 11
- Splay Trees 14
  - Splay Operation 14

Characterizing Algorithms

RAM Model of Computation

We use a Random Access Machine model of computation to estimate the complexity of our algorithms:

- Infinite Memory
- Single CPU
- All operations are roughly equivalent in cost, counting total units of work.
- From these assumptions, we can generate a function that estimates the number of operations performed as a function of the input size.

For example, consider the following factorial function.

```
function factorial(n: number) {

    let total = 1; // 1 unit
    for (let num = n; n > 0; n--) { // 1 unit
        total = total * n; // 4 units
    }

    return total; // 1 unit
}
```

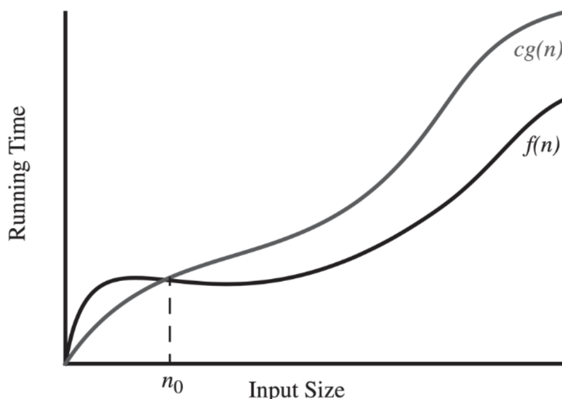
Therefore, we can estimate the number of operations this algorithm performs as 4n + 3.

Asymptotic Notation

However, it's not always possible or feasible to compute the exact number of operations needed as a function of the input size. It quickly becomes impractical to go through the computations for more complex problems. Thankfully, it is often unnecessary. Instead, computer scientists use Big O notation to analyze algorithmic complexity.

Big O Notation

When using Big O notation, we only care about how the number of operations increases with increasing large inputs. More formally, we consider a function f(n) on the order of g(n) (f(n) is O(g(n)), is every value differs by only a constant factor. Big O notation is usually used to evaluate performance for functions as your input gets arbitrarily large.



Example

4n + 3 is **O(N)**

100000 is **O(1)**

$2n^2 + 122n$ is **O(n²)**

Amortized Analysis

This allows us to consider how the runtime of a *series* of operations grows as the input gets larger; it can be extremely useful when your data structure needs to perform some sort of expensive operation *occasionally*. We can use amortized analysis to show that the average cost of an operation is small, even if the cost occasionally is big. For example, consider a dynamic array with initial capacity k and n elements.

32	10	45	345	23			

A dynamic array with capacity k = 8 and n = 5.

Adding a new element to the dynamic array is typically an **O(1)** operation, but occasionally the array fills. When the array is full, we must allocate a new array and copy every element over before inserting the new element. This means that insert is occasionally **O(N)**.

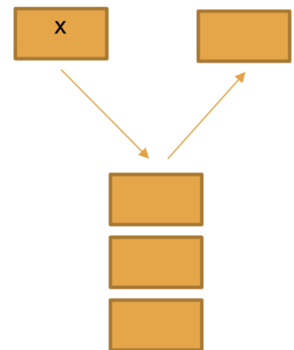
Traditional algorithmic analysis requires us to estimate the cost of insert as **O(N)** and the cost of inserting k elements as **O(N^2)**. However, this is a dramatic overestimation because reallocations should not occur on every insert. We can use amortized analysis to get a better estimate.

Accounting Method

In the accounting method, we assign different costs to operations, with some operations intentionally over- or under-charged. This adjusted cost is used to account for future work, and is called the **amortized cost**. In order to properly estimate the cost, we must ensure that the total amortized cost of a sequence of operations does not exceed the total actual cost.

Stack Example

Consider a stack data structure with the following operations. The naive cost of each operation is shown below.



Push - O(1): Adds a single element to the stack.

Pop - O(1): Removes a single element from the stack.

Multipop - O(k): Removes all elements in the stack up to a maximum of k elements.

So the naive cost of a sequence of k operations is **O(k^2)**. Let's use the accounting method to get a more accurate bound. Here, we assign a schema cost only to the push operation, but with a cost of 2 per insert: 1 unit for insertion and 1 unit for removal.

Operation	Real Cost	Schema Cost	Amortized Cost
Push	1	2	O(1)
Pop	1	0	O(1)
Multipop	k	0	O(1)

It is trivial to see that this schema exceed the total actual cost for a series of operations. We can use the updated amortized costs to estimate the cost of a sequence of k operations as **O(k).**

Potential Method
This is a more formal, rigorous method, better suited for more involved algorithms. Instead of assigning our own schema, we define a **potential function** on the data structure. This function represents the amount of prepaid work that can be released to pay for future operations.

For example, consider the stack example above. As the number of elements in the stack increased, a potential function's value would increase, indicating a higher level of tension within the system. Popping the stack would allow you to release some of that tension.

A potential function is more formally defined as relation that maps each data structure D_i to a real number, the potential associated with that data structure. Then, we can estimate a function's amortized cost using the formula below:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

An estimate for the amortized cost of a function ($\hat{c}_i$) as a function of the real cost of the operation plus any gain of potential energy as a result of that operation.

Stack Example
Consider the stack example above. Let's define the potential function **ϕ** as the number of elements in the stack. Now, we can estimate the amortized cost of each operation.

Operation	**Amortized Cost**
Push	*1 + (k' + 1) - k = 2* *The push operation adds one element to the stack, increasing the potential function by 1. Since the real cost of this operation is 1, this means the amortized cost is 1 + 1 = 2.*
Pop	*1 + (k' - 1) - k = 0* *The push operation removes one element to the stack, decreasing the potential function by 1. Since the real cost of this operation is 1, this means the amortized cost is 1 - 1 = 0.*
Multipop	*k + (k' - k) - k' = 0* *The multipop operation removes at most k elements from the stack, decreasing the potential function by k. Since the operation's real cost is k, the amortized cost is k - k =0.*

Union-Find Data Structures

Overview of Data Structure

Union and Find operations are involved in a number of applications, such as social network analysis, relationship among segments, graph analysis, etc.

There are generally two ways to implement the union-find operations: either a List-based implementation, or a Tree-based implementation)

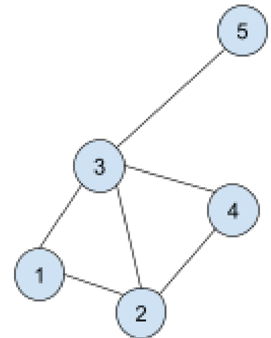
Union-Find data structures always include three operations:

- MakeSet(e) - create a singleton with one element e
- Union(A,B) - generate A ∪ B from A and B *(∪ is the Union symbol)*
- Find(e) - Return the name of the set containing e

Note: MakeSet should be called on each node to create a singleton set so that it can be joined as a Union and found using Find.

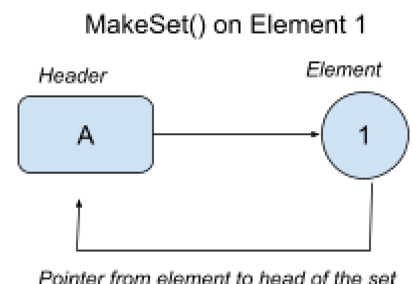
List Based Implementation

```
Class Header {
      List of elements  //list of elements belonging to the set
      Name              //label of the set
}

Class Element {
      Head //Pointer to the head of the set
}
```



Function that when given a set of notes and arcs in the graph, union join them

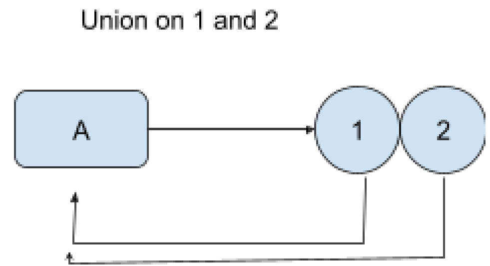
```
function UFConnectedComponents (N,A):
       Input - N set of nodes of the graph
       Input - A set of arcs of the graph

       for each n in N
              makeSet(n) //each element must be turned into a singleton set first
- O(N)

       for each (x,y) in A
                 If (find(x) != find(y)) //for each set in A, if the items are
not already part of the same set
                          union(find(x),find(y)) //join
them - O(A*max(find_c, union_c))
```



So let's break this down into 2 steps:

1. Make each element a singleton set. This allows it to be found using Find() and joined using Union() since it would then be a set.

2. Then, go through each pair in the set of arcs and find both elements header
3. If they are not in the same set, join the sets

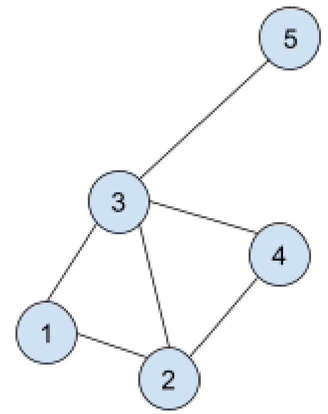



```
function makeSet(e) O(c)
      header h
      h.name = //name for the new set
      h.list = [] //empty list

      Element el = Element(e) O(c)
      h.list.push(el)
      el.head = h

function find(Element el)
      return el.head

function union(Header h1, Header h2) O(N)
      for each element in h2.list
            h1.list.push(element)
            element.head = h1

function unionBySize(Header h1, Header h1) O(logN)
      if(h1.list.size > h2.list.size)
            for each element in h2.list
            h1.list.push(element)
            element.head = h1
else
      for each element in h1.list
            h2.list.push(element)
            element.head = h2
```

Q: What is the cost of running a sequence of N of these operations?

A: $O(N^2)$ - or O(NlogN) if using Union By Size

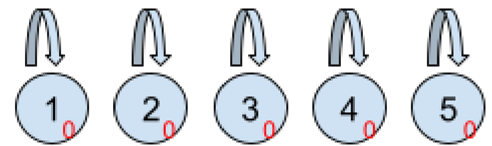
Tree Based Implementation

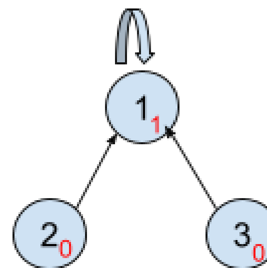


Before makeset is called

```
Class Node{
      Value el //the element in the problem set
      Node* ptr //pointer to the parent node
      rank //informally store the size of a tree - THIS IS ONLY
USED IN RANKED TREES
}
Function makeSet(v}{    //O(c)
      Node node
      node.el = v
      node.ptr = node
      node.rank = 0 //only used when doing ranked trees
}
Function union(A,B){    //O(c)
      B.ptr = A
}
Function find(node){    //O(log n)
      Node n = node
      while(n.ptr != n){  //the lecture slides have an example since he said we repeat
until we have the pointer point to itself
            n = n.ptr
      }
      return node
}

Function union(A,B) {//using rank   O(c)
      if (A.rank < B.rank){
            A.ptr = B
      }else {
           if(A.rank == B.rank){
                A.rank = A.rank+1
           }
           B.ptr = A
      }
}
```

Ranked Example



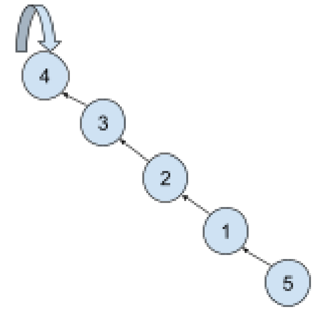


Union(1, 2)
Union(1, 3)

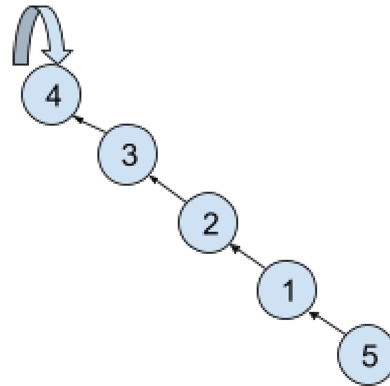
Note: these trees are not ordered and may not be balanced

Path Compression

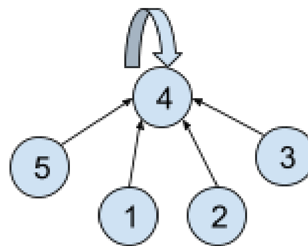
Say we have a tree that is a chain of 5 nodes. If we wanted to find the head of, say, the node 5, we would have to traverse all the way up the tree through pointers. **Path compression** would allow us to reduce the tree's height and make find operations faster. We do this by modifying the Find() function to move the nodes right under the header when the parent is found.




```
Function find(node){
      Node root = node
      while(root.ptr != root){
            root = root.ptr
      }

      Node n = node
      Node next
      while(n.ptr != root){
            Next = node.ptr
            node.ptr = root
            node = next
      }

      return root
}
```



Find(5)



Ackerman's Function

Originally, it was believed that the amortized cost was O(I), but it was proved false. However, it is still better than O(log n)

A(m,n) is known as Ackerman's function, which is the fastest growing function known

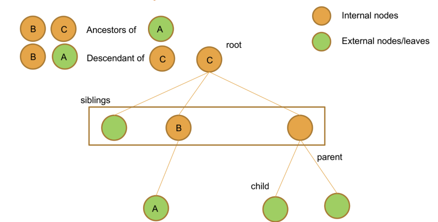
$A(1,\ j)\ =\ 2^j \quad for\ i\ \geq\ 1,$

$A(i, 1)\ =\ A(i-1, 2) \quad for\ i\ \geq 2,$

$A(i,\ j)\ =\ A(i-1,\ A(i, j-1)) \quad for\ i, j\ \geq\ 2$

Binary Trees

A tree is a set of nodes linked with a parent-child relationship. Each tree has a special parent node R, the **Root**, which has no parent nodes. Parents can have many children, but for this section we will consider only binary trees, where parents have at most two children.

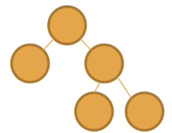


Binary Trees are often used to store sets of numbers, where the *left child* of a node is strictly less than the parent node, and the right child is strictly greater than the child node.

<u>*Properties of Binary Trees (We Care About)*</u>

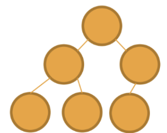
Balance

A binary tree is *balanced* if the left and right subtrees are both balanced, and their height differs by at most one. A balanced binary tree has a height of **log (N)**, which is useful because most operations depend on the tree's height.



Proper/Full

A tree is proper/full if every node other than leaves has two children.



Complete

A tree in which every level, except possibly the last, is completely filled. All nodes are as far left as possible

<u>*Operations (We Care About)*</u>

Insert - O(log N)*

Remove - O(log N)*

Find - O(log N)*

Traversal - O(N)

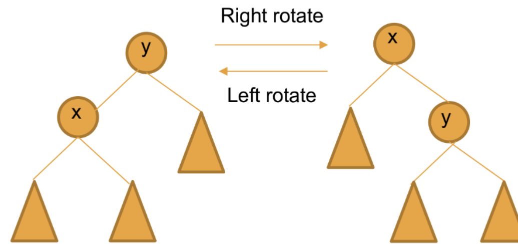
```
void print_inorder(Node *T) {
  print_inorder(T->left);
  print(T);
  print_inorder(T->right);
}
void print_preorder(Node *T) {
  print(T);
  print_inorder(T->left);
  print_inorder(T->right);
}
void print_postorder(Node *T) {
  print_inorder(T->left);
  print_inorder(T->right);
  print(T);
}
```

Most of the operations we care about in binary trees are **O(log N) if the tree is balanced, so the focus of these variants of trees is to keep trees balanced. This is true because these operations mostly depend on the tree's height, log N in a balanced tree. The worst case for these operations when the tree is not balanced is O(N). This is the case when the Binary Tree is essentially a Linked List.*

AVL Trees

An AVL Tree is perhaps the most literal way to ensure balance in all operations by utilizing **rotations** to ensure that the height of the left and right subtrees of every parent of the most recently touched node differ by no more than 1.

Rotations



They accomplish this balance by defining a **balance** operation applied to the most recently touched node and all of its parent nodes up to the root.

Balance Node N

1. Get the height of N's left and right subtree. If they differ by more than 1, this N is *unbalanced.* Get the children and grandchildren of N.
    a. If both are on the left, apply the right rotation
    b. If both are on the right, apply left rotation
    c. If one is right and the other is left, apply left-right rotation
    d. If one is left and the other is right, apply right-left rotation
2. Update stored height property to reflect the new reality

Red Black Trees

In order to maintain balance, a Red Black Tree introduces additional metadata into each node: the node's **color**, which can be either red or black. Then, a tree will be balanced if it follows these rules.

- The root of the tree is black
- Every leaf is black. In many implementations, the "leaves" are the NULL pointers on the nodes without children.
- If a node is red, then it's children are black
- For each node, all simple paths from the node to the descendent leaves contain the same number of black nodes

Insertion Operation

To insert into a Red Black Tree, insert like normal, coloring the new node **red**. Then call the fix up function to get the red black tree back in compliance with the properties of red black trees, ensuring balance.

Fix Up

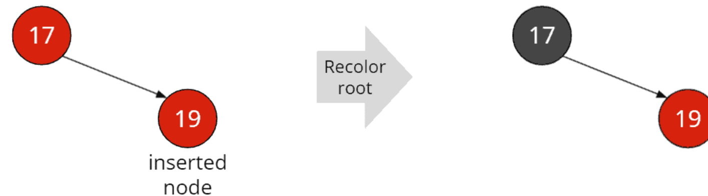
After a new node is inserted, fix up can be used to maintain the red black tree's properties, *rebalancing* the tree. When doing this, one must consider a number of cases.

Case 1: New node is the root

Color the new node black.

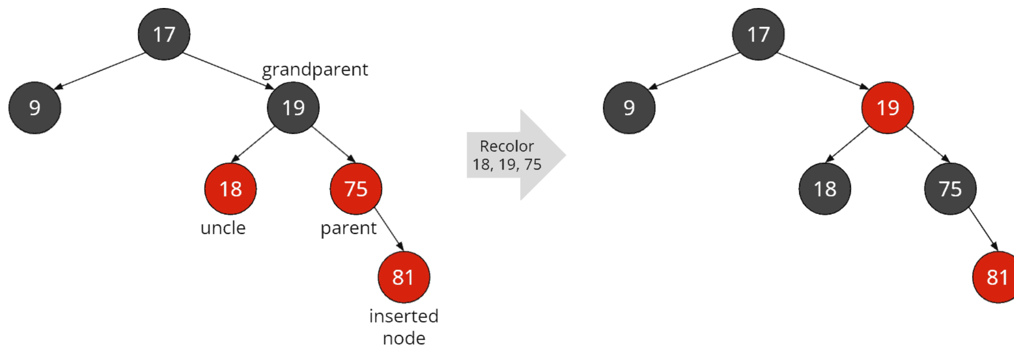
Case 2: Parent Node is Red, and The Root

Color the root node black.



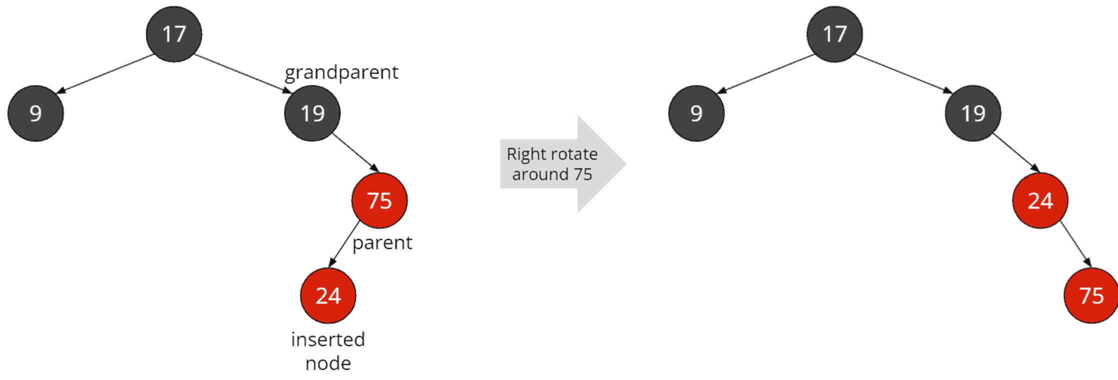
Case 3: Both Parent and Parent's Sibling Are Red

Paint the parent and parent's sibling black, and paint the grandparent red.

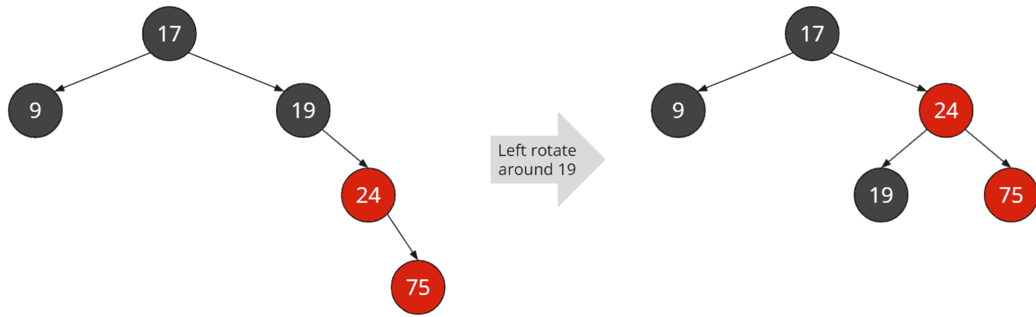


Case 4: Parent is Red, Parent's Sibling is Black, New Node is Inner Grandchild

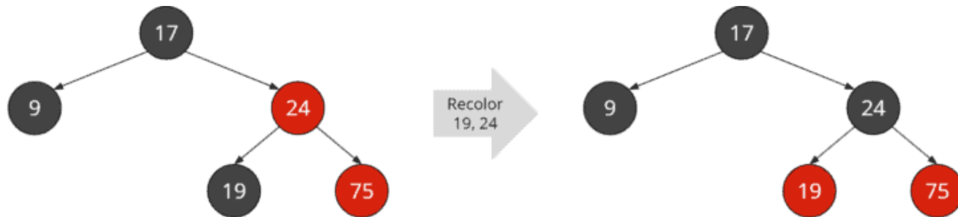
Inner Grandchild means that parent is a left child and new node is a right child, or vise versa. Rotate at the parent node in the opposite direction of the inserted node.



Next, rotate at the grandparent node opposite to the previous rotation.

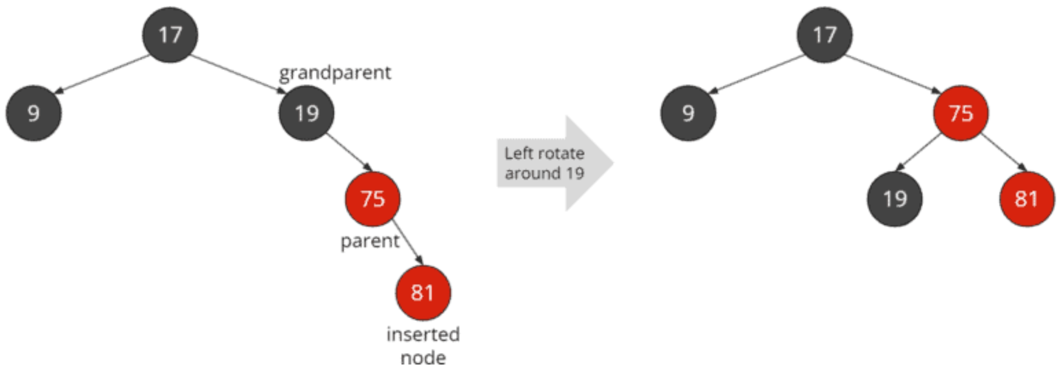


Finally, color the newly inserted node black and the original grandparent red.



Case 5: Parent Node Is Red, Parent's Sibling Node Is Black, New Node Is "Outer Grandchild"

Outer Grandchild means the new node and parent are both left or right children. Rotate at the grandparent in the opposite direction of the parent and inserted node (if the parent and new node are both right children, then rotate left at the grandparent).



Then, recolor the former parent black and the former grandparent red.

Splay Trees

In a Splay Tree, we ensure balance by performing a series of splay operations on the most recently touched node in order to rotate it up to the root. In addition to maintaining balance, this moves frequently accessed nodes up the binary search tree, implementing a soft LRU cache. As an additional benefit, splay trees do not need any additional metadata stored in each node to maintain balance, unlike Red Black Trees or AVL Trees.

Splay Operation

When a node x is accessed, a splay operation is performed on x to move it to the root. To perform a splay operation, we carry out a sequence of steps, each moving x closer to the root. By performing a splay operation on the node of interest after every access, the recently accessed nodes are kept near the root, and the tree remains roughly balanced.

Each splay step depends on several factors:

1. Whether x is the left or right child of its parent, *p*
2. Whether *p* is the root
3. Whether *p* is the left or right child of its parent, *g* (grandparent of x).

When	What	Picture
p is the root *x* is left child	Zig Left	
p is the root *x* is right child	Zig Right	
p is not root x and p are both left children	Zig Zig Left	
p is not root x and p are both right children	Zig Zig Right	
p is not root x is left child, p is right child	Zig Zag Left	
p is not root x is right child, p is left child	Zig Zag Right	