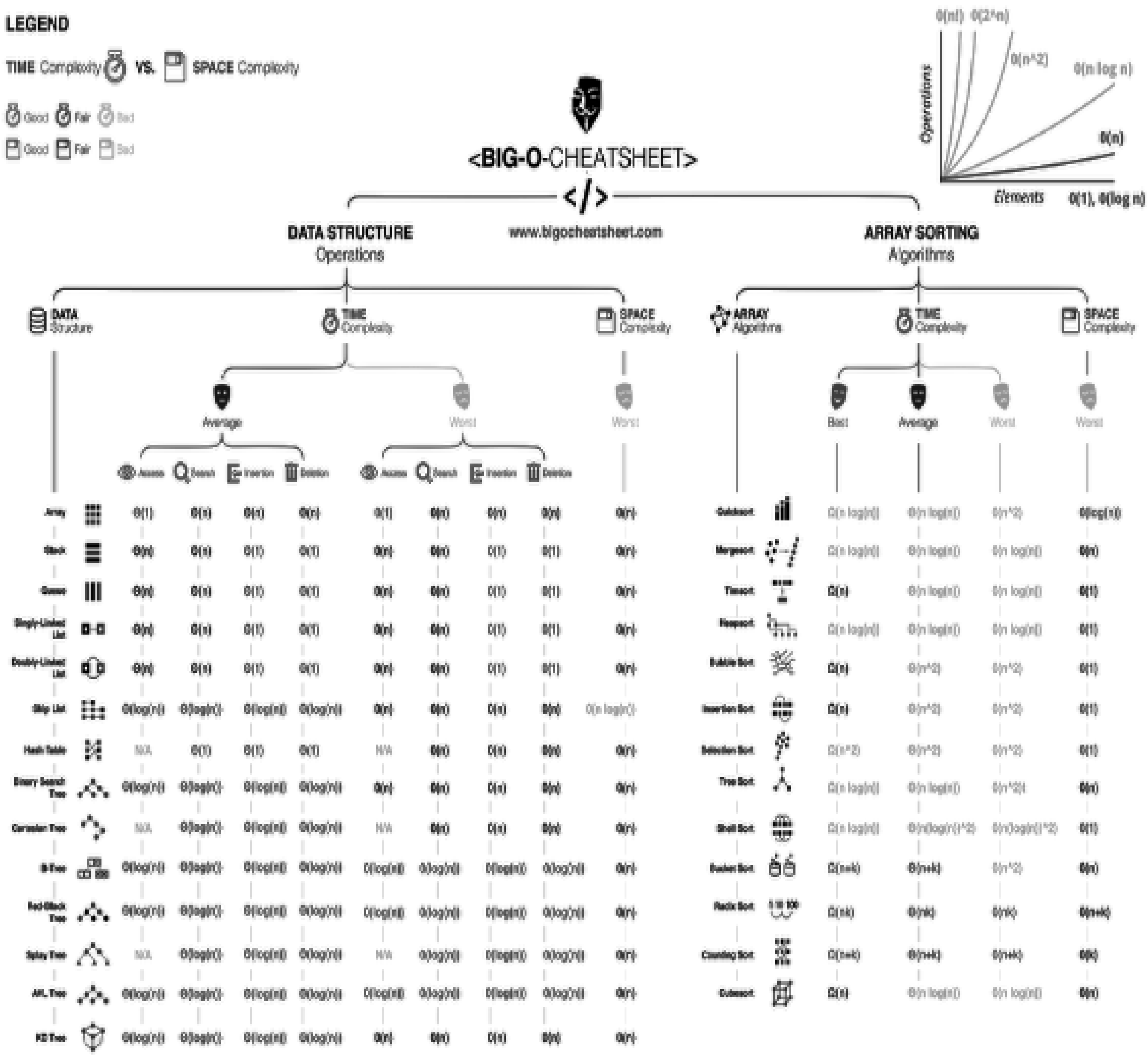
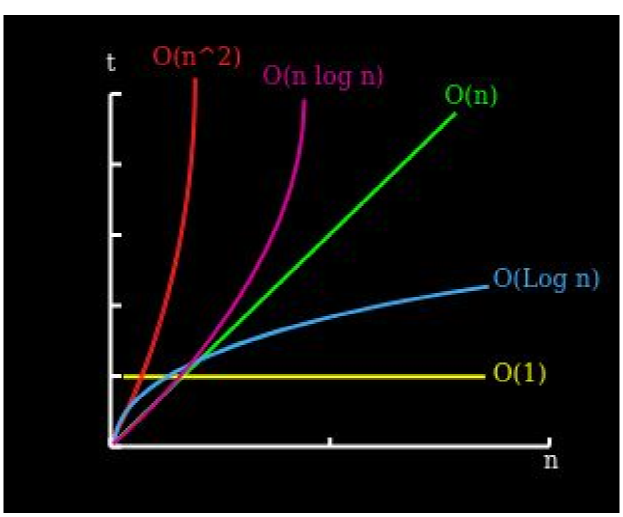
`





Constant Time O(1)
Linear Time O(n)
Quadratic Time O(n^2)
Logarithmic Time O(Log n)
Linearithmic(n*Log n)
Runtime

f(n)	>=	O(g(n))
f(n)	>	o(g(n))
f(n)	=	θ(g(n))
f(n)	<	ω(g(n))
f(n)	<=	Ω(g(n))

Indicate for each of the statements below whether the statement is true or false.

You do not need to show work. (2 points each)

a) n^3 is in $\Omega(n^4)$ False

b) $2n^3 + n^2$ is in $\theta(n^3)$ True

c)

while

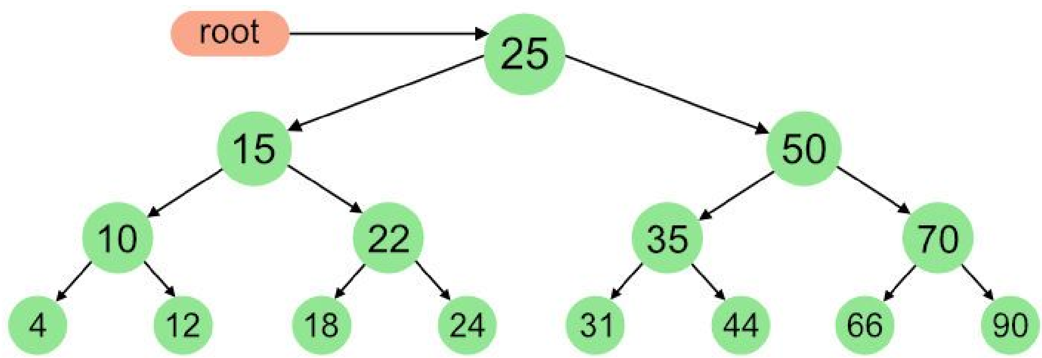
InOrder(root) visits nodes in the following order:

4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

(n>0) { is in Ω(n) and O(n)
n--;
} //True
d) while (n^2>0) {is in Ω(n) and n--; } //False!!!
e) buildHeap() is in O(n) True
f) AVL height is O(logn) True
g) AVL height is O(n) True
h) BST height is best case O(n) True

<u>Tree Traversal</u>
InOrder: (LPR) O(n)
PostOrder: (LRP)
PreOrder: (PLR)

Call recursively for tree traversal.
LPR(Left Print Right)
LRP(Left Right Print)
PLR(Print Left Right)



<u>Hash Table</u>
Consider inserting data with integer keys 22, 55, 33, 2, 26, 12 in that order into a hash table of size 11 where the hashing function is h(key) %11.

Mod		Chaining Hash		QP		Linear f(i)=i h(k),…,h(k)+ i		Double Hash f(i)=i*g(k) h(k),…,h(k)+i*g(k)
22 % 11 = 0	0	33 -> 55 -> 22	0	22	0	22	0	22
55 % 11 = 0	1	12	1	55	1	55	1	55
33 % 11 = 0	2	2	2	2	2	33	2	33
2 % 11 = 2	3		3		3	2	3	12
26 % 11 = 4	4	26	4	33	4	26	4	26
12 % 11 = 1	5		5	26	5	12	5	
	6		6		6		6	
	7		7		7		7	24
	8		8		8		8	
	9		9		9		9	
	10		10	12	10		10	

Programming Questions

```
IF tree is right heavy
{
  IF tree's right subtree is left heavy
  {
     Perform Double Left rotation
  }
  ELSE
  {
     Perform Single Left rotation
  }
}
ELSE IF tree is left heavy
{
  IF tree's left subtree is right heavy
  {
     Perform Double Right rotation
  }
  ELSE
  {
     Perform Single Right rotation
  }
}
```

```java
public static Integer last_printed = null;
public static boolean checkBST(TreeNode n){
    if (n == null) return true;
    // check / recurse left
    if (!checkBST(n.left)) return false;

    // check current
    if( last_printed != null && n.data <= last_printed) {
      return false;
    }

    //check / recurse right
    if (!checkBST(n.right)) return false;

    return true;
 }
```

```java
boolean checkBST(TreeNode n){
    return checkBST(n, null, null);
}

boolean checkBST(TreeNode n, Integer min, Integer max){
    if(n == null){
        return true;
    }

    if((min != null && n.data <= min) ||
       (max !=null && n.data > max)){
        return false;
    }

    if(!checkBST(n.left, min, n.data) ||
       !checkBST(n.right, n.data, max)){
        return false;
    }
    return true;
}
```

```java
/* Java program to checks if a binary tree is max heap or not */
// A Binary Tree node
class Node
{
	int key;
	Node left, right;
	Node(int k)
	{
		key = k;
		left = right = null;
	}
}

class Is_BinaryTree_MaxHeap
{
	/* This function counts the number of nodes in a binary tree */
	int countNodes(Node root)
	{
		if(root==null)
			return 0;
		return(1 + countNodes(root.left) + countNodes(root.right));
	}

	/* This function checks if the binary tree is complete or not */
	boolean isCompleteUtil(Node root, int index, int number_nodes)
	{
		// An empty tree is complete
		if(root == null)
			return true;

		// If index assigned to current node is more than
```

```
        // number of nodes in tree, then tree is not complete
        if(index >= number_nodes)
            return false;

        // Recur for left and right subtrees
        return isCompleteUtil(root.left, 2*index+1, number_nodes) &&
            isCompleteUtil(root.right, 2*index+2, number_nodes);

    }

    // This Function checks the heap property in the tree.
    boolean isHeapUtil(Node root)
    {
        // Base case : single node satisfies property
        if(root.left == null && root.right==null)
            return true;

        // node will be in second last level
        if(root.right == null)
        {
            // check heap property at Node
            // No recursive call , because no need to check last level
            return root.key >= root.left.key;
        }
        else
        {
            // Check heap property at Node and
            // Recursive check heap property at left and right subtree
            if(root.key >= root.left.key && root.key >= root.right.key)
                return isHeapUtil(root.left) && isHeapUtil(root.right);
            else
                return false;
        }
    }
    // Function to check binary tree is a Heap or Not.
    boolean isHeap(Node root)
    {
        if(root == null)
            return true;

        // These two are used in isCompleteUtil()
        int node_count = countNodes(root);

        if(isCompleteUtil(root, 0 , node_count)==true && isHeapUtil(root)==true)
            return true;
        return false;
    }
    // driver function to test the above functions
    public static void main(String args[])
    {
        Is_BinaryTree_MaxHeap bt = new Is_BinaryTree_MaxHeap();

        Node root = new Node(10);
        root.left = new Node(9);
        root.right = new Node(8);
        root.left.left = new Node(7);
```

```java
            root.left.right = new Node(6);
            root.right.left = new Node(5);
            root.right.right = new Node(4);
            root.left.left.left = new Node(3);
            root.left.left.right = new Node(2);
            root.left.right.left = new Node(1);
                  if(bt.isHeap(root) == true)
                  System.out.println("Given binary tree is a Heap");
            else
                  System.out.println("Given binary tree is not a Heap");
      }
} //End Heap Confirmation program
// Java Program for Lowest Common Ancestor in a Binary Tree
// A O(n) solution to find LCA of two given values n1 and n2
import java.util.ArrayList;
import java.util.List;

// A Binary Tree node
class Node {
      int data;
      Node left, right;

      Node(int value) {
            data = value;
            left = right = null;
      }
}

public class BT_NoParentPtr_Solution1
{
      Node root;
      private List<Integer> path1 = new ArrayList<>();
      private List<Integer> path2 = new ArrayList<>();

      // Finds the path from root node to given root of the tree.
      int findLCA(int n1, int n2) {
            path1.clear();
            path2.clear();
            return findLCAInternal(root, n1, n2);
      }

      private int findLCAInternal(Node root, int n1, int n2) {

            if (!findPath(root, n1, path1) || !findPath(root, n2, path2)) {
                  System.out.println((path1.size() > 0) ? "n1 is present" : "n1 is missing");
                  System.out.println((path2.size() > 0) ? "n2 is present" : "n2 is missing");
                  return -1;
            }

            int i;
            for (i = 0; i < path1.size() && i < path2.size(); i++) {
            // System.out.println(path1.get(i) + " " + path2.get(i));
                  if (!path1.get(i).equals(path2.get(i)))
                        break;
            }
            return path1.get(i-1);
```

```
    }

    // Finds the path from root node to given root of the tree, Stores the
    // path in a vector path[], returns true if path exists otherwise false
    private boolean findPath(Node root, int n, List<Integer> path)
    {
        // base case
        if (root == null) {
            return false;
        }

        // Store this node . The node will be removed if
        // not in path from root to n.
        path.add(root.data);

        if (root.data == n) {
            return true;
        }

        if (root.left != null && findPath(root.left, n, path)) {
            return true;
        }

        if (root.right != null && findPath(root.right, n, path)) {
            return true;
        }

        // If not present in subtree rooted with root, remove root from
        // path[] and return false
        path.remove(path.size()-1);

        return false;
    }

    // Driver code
    public static void main(String[] args)
    {
        BT_NoParentPtr_Solution1 tree = new BT_NoParentPtr_Solution1();
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);
        tree.root.right.left = new Node(6);
        tree.root.right.right = new Node(7);

        System.out.println("LCA(4, 5): " + tree.findLCA(4,5));
        System.out.println("LCA(4, 6): " + tree.findLCA(4,6));
        System.out.println("LCA(3, 4): " + tree.findLCA(3,4));
        System.out.println("LCA(2, 4): " + tree.findLCA(2,4));
    }
} //End LCA program
```

Meet the Family

- **O(f(n))** is the set of all functions asymptotically less than or equal to f(n)
  - **o(f(n))** is the set of all functions asymptotically strictly less than f(n)
- **Ω(f(n))** is the set of all functions asymptotically greater than or equal to f(n)
  - **ω(f(n))** is the set of all functions asymptotically strictly greater than f(n)
- **θ(f(n))** is the set of all functions asymptotically equal to f(n)

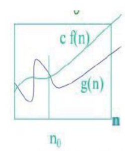
Asymptotic Analysis

- Eliminate low order terms
  - 4n + 5 =>
  - 0.5 n log n + 2n + 7 =>
  - n^3 + 2^n + 3n =>
- Eliminate coefficients
  - 4n =>
  - 0.5 n log n =>
  - n log n^2 =>

Definition of Order Notation

- Upper bound: T(n) = O(f(n)) Big-O
  - Exist constants c and n' such that
    - T(n) <= c f(n) for all n >= n'
- Lower bound: T(n) = Ω(g(n)) Omega
  - Exist constants c and n' such that
    - T(n) >= c g(n) for all n>= n'
- Tight bound T(n) = Θ(f(n)) Theta
  - When both hold: T(n) = O(f(n)) and T(n) = Ω(f(n))

Example

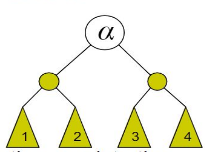


- $g(n) = 1000n$ vs. $f(n) = n^2$
- Is $g(n) \in O(f(n))$?
  - Pick: $n_0 = 1000$, $c = 1$
  - $1000n \le 1 * n^2$ for all $n \ge 1000$
  - So $g(n) \in O(f(n))$
- Small cases, really don't matter. As long as it's eventually an upper bound, it fits the defintion
- If f(n) is in O(n)... what about is f(n) in $O(n^2)$?

Slowest to Fastest Growing	Function Type	Example
1	Constant Functions	5, 25, 6000
2	Logarithmic Functions	$\log_5 n$, log n
3	Linear Functions	5n, 25n, 6000n
4	Linearithmic Functions	n $\log_5 n$, n log n
5	Polynomial Functions	$5n^2$, $25n^4$, $6000n^{12}$
6	Exponential Functions	5^n, 25^{6000n}

Balancing AVL Trees, *cont'd*

- Therefore, one of the following had to occur:
  - Case 1 (outside left-left): The insertion was into the left subtree of the left child of α.
  - Case 2 (inside left-right): The insertion was into the right subtree of the left child of α.
  - Case 3 (inside right-left): The insertion was into the left subtree of the right child of α.
  - Case 4 (outside right-right): The insertion was into the right subtree of the right child of α.



Cases 1 and 4 are mirrors of each other, and cases 2 and 3 are mirrors of each other.

Balancing AVL Trees: Case 1

- Case 1 (outside left-left): Rebalance with a single right rotation.

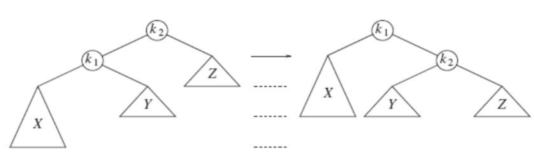
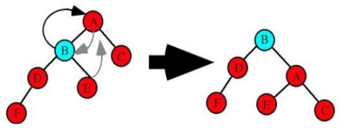


Figure 4.31 Single rotation to fix case 1

Balancing AVL Trees: Case 1, *cont'd*

- Case 1 (outside left-left):
  Rebalance with a single right rotation.



Node A is unbalanced.
Single right rotation: A's left child B becomes the new root of the subtree.
Node A becomes the right child and adopts B's right child as its new left child.

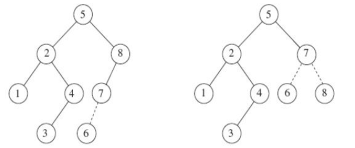


Figure 4.32 AVL property destroyed by insertion of 6, then fixed by a single rotation

Node 8 is unbalanced.
Single right rotation: 8's left child 7 becomes the new root of the subtree.
Node 8 is the right child.

Balancing AVL Trees: Case 2

- Case 2 (inside left-right):
  Rebalance with a double left-right rotation.

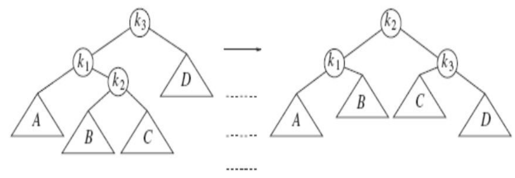
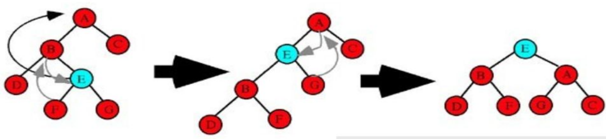


Figure 4.35 Left–right double rotation to fix case 2

Balancing AVL Trees: Case 2, *cont'd*

- Case 2 (inside left-right):
  Rebalance with a double left-right rotation.



http://www.cs.uah.edu/~rcoleman/CS221/Trees/AVLTree.htm

Node A is unbalanced.
Double left-right rotation: E becomes the new root of the subtree after two rotations. Step 1 is a single left rotation between B and E. E replaces B as the subtree root. B becomes E's left child and B adopts E's left child F as its new right child. Step 2 is a single right rotation between E and A. E replaces A is the subtree root. A becomes E's right child and A adopts E's right child G as its new left child.

Balancing AVL Trees: Case 3

- Case 3 (inside right-left):
  Rebalance with a double right-left rotation.

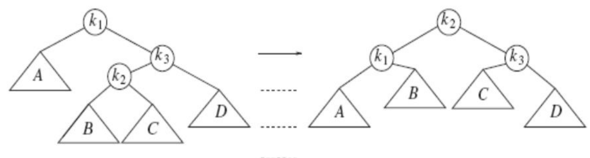
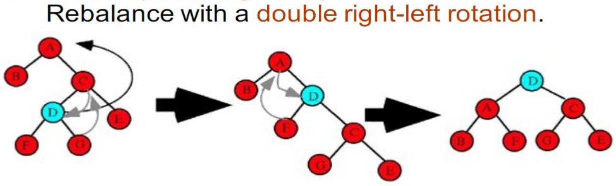


Figure 4.36 Right–left double rotation to fix case 3

Balancing AVL Trees: Case 3, *cont'd*

http://www.cs.uah.edu/~rcoleman/CS221/Trees/AVLTree.ht

- Case 3 (inside right-left):
  Rebalance with a double right-left rotation.



Node A is unbalanced.
Double right-left rotation: D becomes the new root of the subtree after two rotations. Step 1 is a single right rotation between C and C. D replaces C as the subtree root. C becomes D's right child and C adopts D's right child G as its new left child. Step 2 is a single left rotation between D and A. D replaces A is the subtree root. A becomes D's left child and A adopts D's left child F as its new right child.

Balancing AVL Trees: Case 4

- Case 4 (outside right-right):
  Rebalance with a single left rotation.

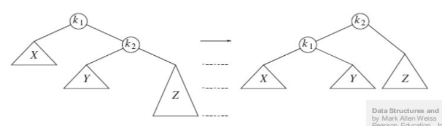
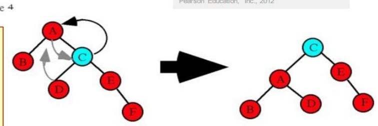


Figure 4.33 Single rotation fixes case 4

Data Structures and Algorithms in Java, 3rd ed. by Mark Allen Weiss Pearson Education, Inc., 2012

Node A is unbalanced.
Single left rotation: A's right child C becomes the new root of the subtree.
Node A becomes the left child and adopts C's left child as its new right child.



11

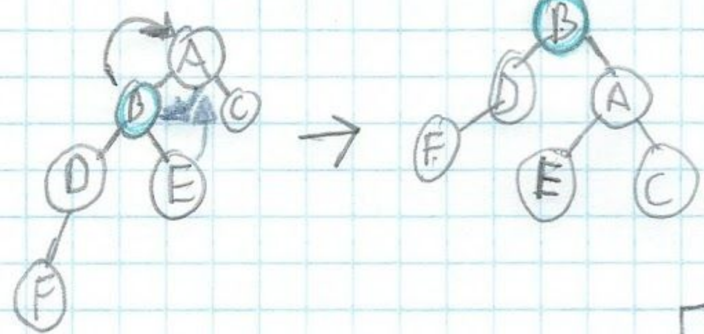
AVL Trees (N> root = Right Child, N<root = Left Child)

AVL's are balanced as long as the heights differ by 1. Not only for the entire tree, but each subtree as well.

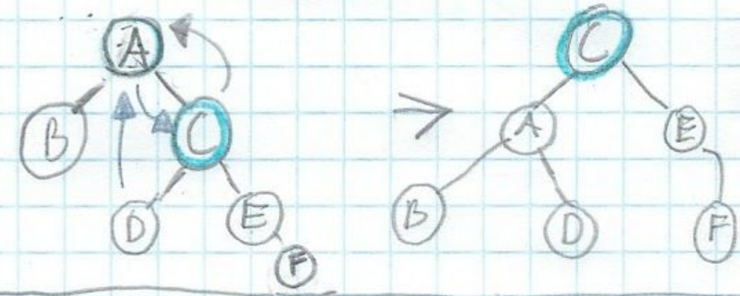
If the height differs by 2 or more, then unbalanced, and we'll need a rotation. We look at the root and call it α.

AVL trees

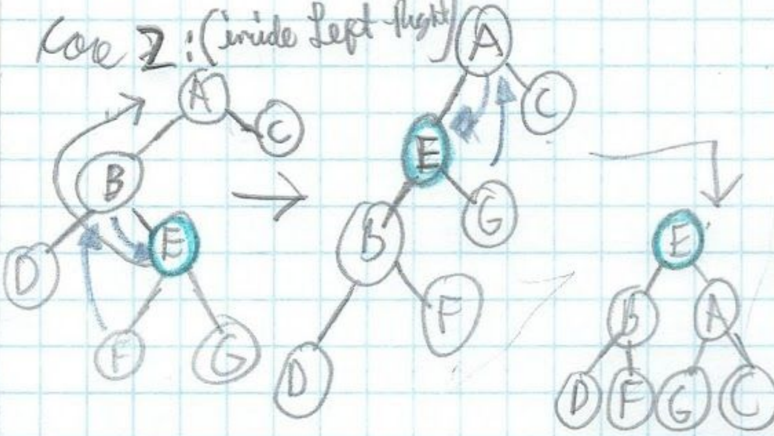
Case 1: (Left-Left) outside



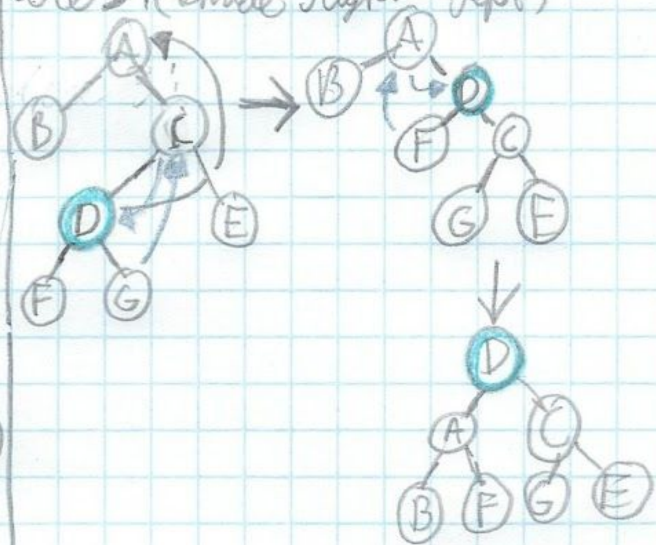
Case 4 (Right-Right) outside



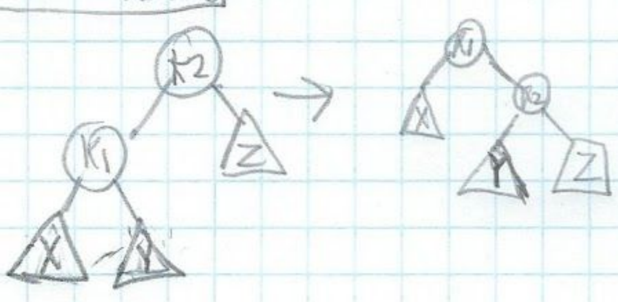
Case 2: (inside Left-Right)



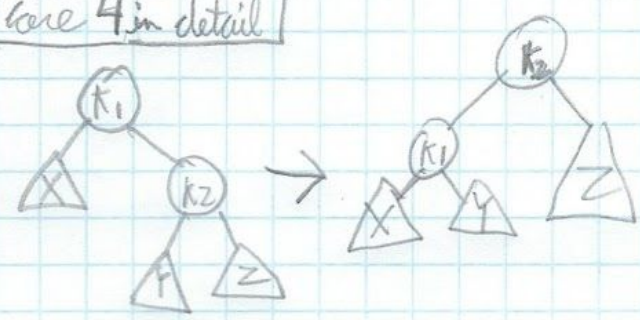
Case 3: (inside right-Left)



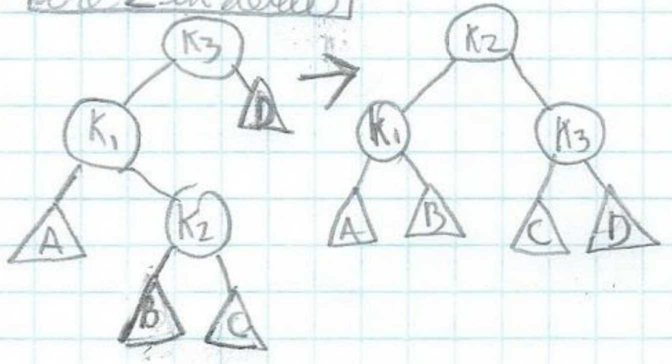
Case 1 in detail



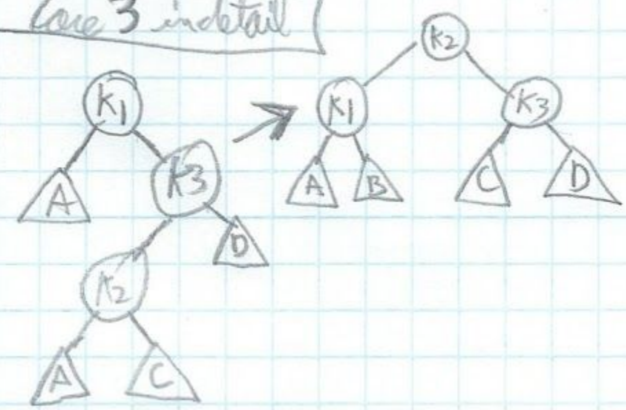
Case 4 in detail



Case 2 in detail



Case 3 in detail



Sorting Summary

- Simple $O(n^2)$ sorts can be fastest for small n
  - selection sort, insertion sort (latter linear for mostly-sorted)
  - good for "below a cut-off" to help divide-and-conquer sorts
- $O(n \log n)$ sorts
  - heap sort, in-place but not stable nor parallelizable
  - merge sort, not in place but stable and works as external sort
  - quick sort, in place but not stable and $O(n^2)$ in worst-case
    - often fastest, but depends on costs of comparisons/copies
- $\Omega(n \log n)$ is worst-case and average lower-bound for sorting by comparisons
- Non-comparison sorts
  - Bucket sort good for small number of key values
  - Radix sort uses fewer buckets and more phases
- Best way to sort? It depends!

Demo

```java
boolean isPalindrome(LinkedListNode head){
    LinkedListNode fast = head;
    LinkedListNode slow = head;

    Stack<Integer> stack = new Stack<Integer>();

    //push elements from first half of linked list onto stack
    while(fast != null && fast.next != null){
        stack.push(slow.data);
        slow = slow.next;
        fast = fast.next.next;
    }

    //has odd num of elements, so skip the middle
    if (fast != null){
        slow = slow.next;
    }

    while(slow != null){
        int top = stack.pop().intValue();

        //if values are different, then it's not a palindrome
        if(top != slow.data)
            return false;

        slow = slow.next;
    }
    return true;
}
```

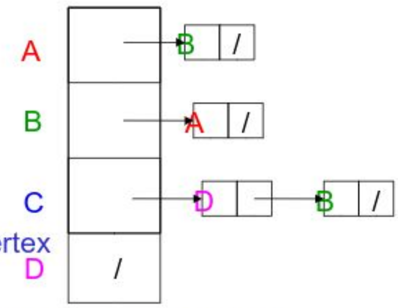
Palindrome might be on test,
but put it in just in case.

Heaps [Index Zero is always empty]
Min Heap
i is the Current Index in the array.
2i will give the left child
2i+1 Right Child

DeleteMin:
Deletes lowest value (the root) and creates a whole.
Next Lowest Value moves up to the root.
Then the next lowest root becomes the new
root of sub tree.

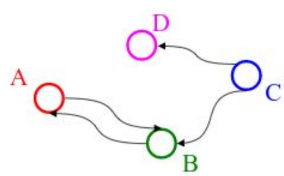
Inserting Values into Min Heap
Add a hole at the end of the tree and place the new value. Compare the new value to the upper values, if it's less than them, keep moving up. If the new value is greater than the upper root, then it stays.

Adjacency List Properties



- Running time to:
  - Get all of a vertex's out-edges:
    $O(d)$ where d is out-degree of vertex
  - Get all of a vertex's in-edges:
    $O(|E|)$ (but could keep a second adjacency list for this!)
  - Decide if some edge exists:
    $O(d)$ where d is out-degree of source
  - Insert an edge: $O(1)$ (unless you need to check if it's there)
  - Delete an edge: $O(d)$ where d is out-degree of source
- Space requirements:
  1. $O(|V|+|E|)$
- Best for dense or sparse graphs?
  1. Best for sparse graphs, so usually just stick with linked lists

More notation



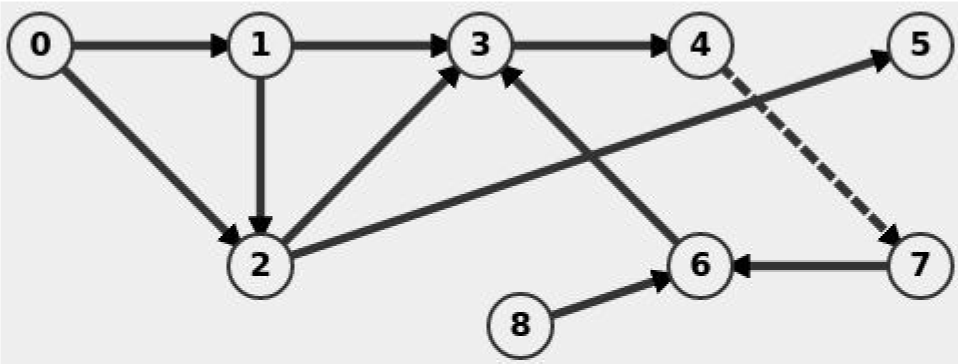
For a graph **G=(V,E)**:

- **|V|** is the number of vertices
- **|E|** is the number of edges
  - Minimum? 0
  - Maximum for undirected? $|V||V+1|/2 \times O(|V|^2)$
  - Maximum for directed? $|V|^2 \times O(|V|^2)$
    (assuming self-edges allowed, else subtract **|V|**)
- If **(u,v)** × **E**
  - Then **v** is a neighbor of **u**, i.e., **v** is adjacent to **u**
  - Order matters for directed edges
    - **u** is not adjacent to **v** unless **(v,u)** × **E**

Adjacency Matrix Properties

	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

- Running time to:
  - Get a vertex's out-edges: $O(|V|)$
  - Get a vertex's in-edges: $O(|V|)$
  - Decide if some edge exists: $O(1)$
  - Insert an edge: $O(1)$
  - Delete an edge: $O(1)$
- Space requirements:
  1. $|V|^2$ bits
- Best for sparse or dense graphs?
  1. Best for dense graphs



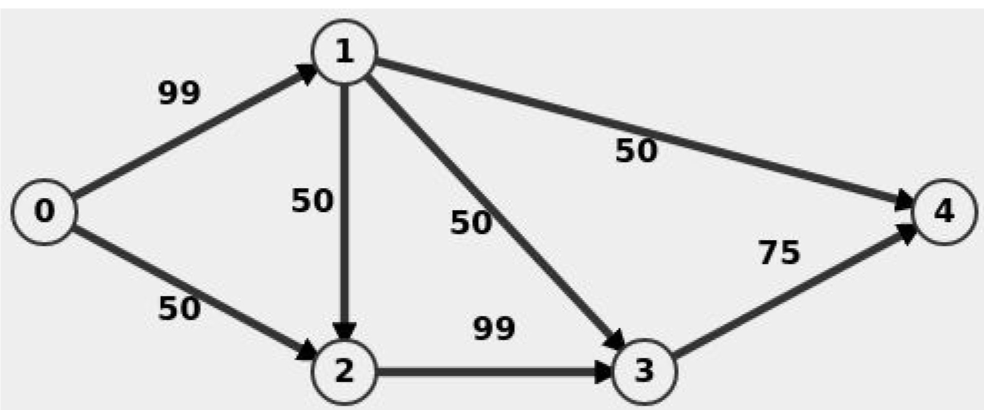
• **V=9, E=11** • Tree? **N/A** • Complete? **No** • Bipartite? **N/A** • DAG? **No**

Edge List		
0:	0	1
1:	0	2
2:	1	2
3:	1	3
4:	2	3
5:	2	5
6:	3	4
7:	4	7
8:	6	3
9:	7	6
10:	8	6

Adjacency Matrix

	0	1	2	3	4	5	6	7	8
0	0	1	1	0	0	0	0	0	0
1	0	0	1	1	0	0	0	0	0
2	0	0	0	1	0	1	0	0	0
3	0	0	0	0	1	0	0	0	0
4	0	0	0	0	0	0	0	1	0
5	0	0	0	0	0	0	0	0	0
6	0	0	0	1	0	0	0	0	0
7	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	1	0	0

Adjacency List		
0:	1	2
1:	2	3
2:	3	5
3:	4	
4:	7	
5:		
6:	3	
7:	6	
8:	6	



• **V=5, E=7** • Tree? **N/A** • Complete? **No** • Bipartite? **N/A** • DAG? **Yes**

Edge List			
0:	0	1	99
1:	0	2	50
2:	1	2	50
3:	1	3	50
4:	1	4	50
5:	2	3	99
6:	3	4	75

Adjacency Matrix

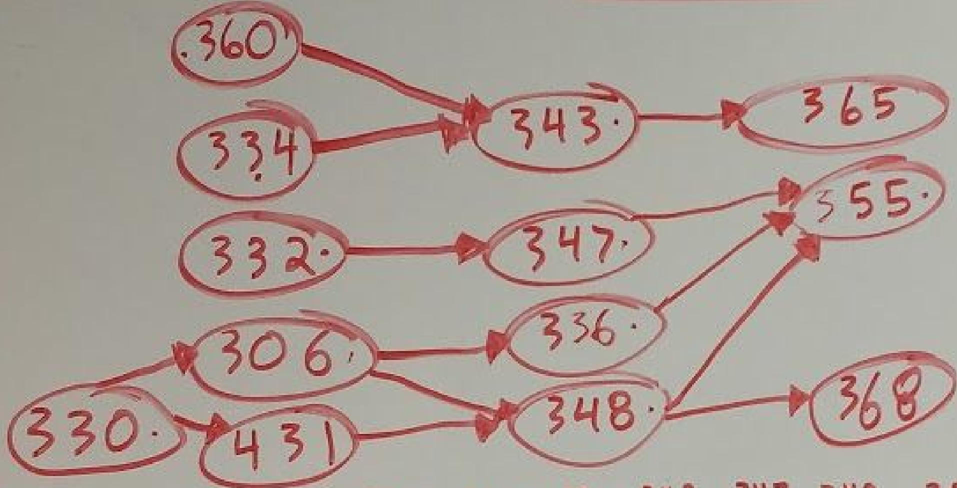
	0	1	2	3	4
0	0	99	50	0	0
1	0	0	50	50	50
2	0	0	0	99	0
3	0	0	0	0	75
4	0	0	0	0	0

Adjacency List			
0:	(1, 99)	(2, 50)	
1:	(2, 50)	(3, 50)	(4, 50)
2:	(3, 99)		
3:	(4, 75)		
4:			

Topo Sort

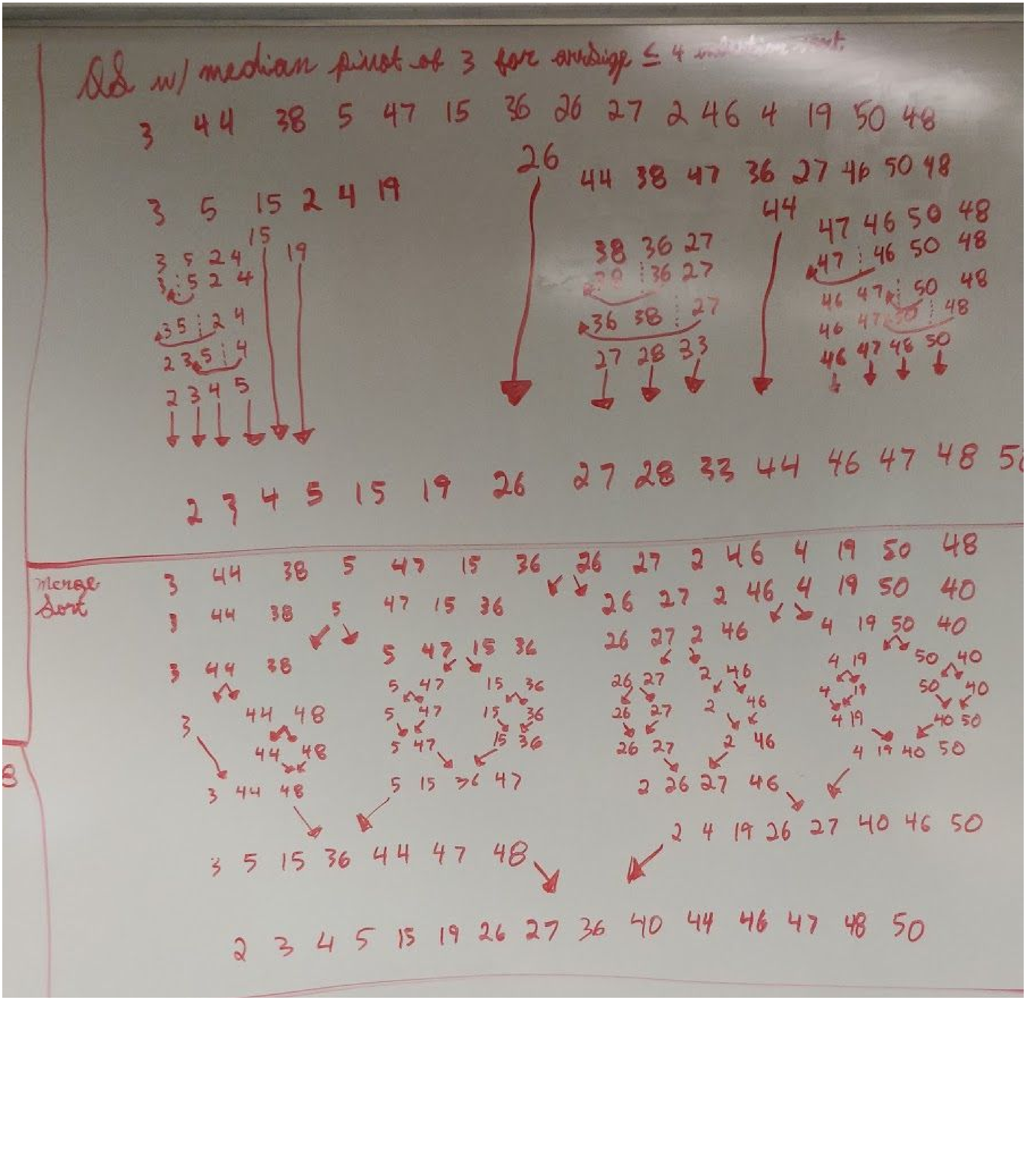
node	142	143	311	312	331	332	333	341	351	352	440
in-deg	0 X	1 0 X	1 1 1 1 0 X	2 2 2 2 2 1 0	1 1 1 0 0 X	1 1 1 1 0 X	2 2 2 2 2 2 1 1 0 X	1 1 1 0 0 0 X	1 1 1 0 0 0 0 X	2 2 2 2 1 1 1 0 X	1 1 1 1 1 0 0 0 X

out: 142, 143, 311, 331, 332, 341, 351, 333, 352, 440



out: 330 306 332 334 336 347 360 343 365 431 348 368 355

node	306	330	332	334	336	343	347	348	355	360	365	368	431
in-deg	1 0 X	0 X	0 0 0 X	0 0 0 0 X	1 1 0 0 0 X	2 2 2 2 1 1 1 0 X	1 1 1 0 0 0 X	2 2 1 1 1 1 1 1 1 0 X	3 3 3 3 3 2 1 1 1 1 1 0	0 0 0 0 0 0 0 X	1 1 1 1 1 1 1 0 X	1 1 1 1 1 1 1 1 1 1 0	1 0 0 0 0 0 0 0 0 0 X



QS w/ median pivot of 3 for arrsize ≤ 4 insertion sort.
3 44 38 5 47 15 36 26 27 2 46 4 19 50 48
26
44 38 47 36 27 46 50 48
3 5 15 2 4 19
44
47 46 50 48
38 36 27
2 3 4 5 15 19 26 27 28 33 44 46 47 48 50
Merge Sort
3 44 38 5 47 15 36 26 27 2 46 4 19 50 48
3 5 15 36 44 47 48
2 4 19 26 27 40 46 50
2 3 4 5 15 19 26 27 36 40 44 46 47 48 50