COMP 251

CSUS REVIEW SESSION

BY NESHMA & FIRZANA

Email:

- firzana.sadik@mail.mcgill.ca
- neshma.metri@mail.mcgill.ca

CONTENTS

Data Structures

- Hashing
- Disjoint Sets
- Heaps and Heapsort
- Red Black Trees
- BST and AVL Trees

Design and Analysis

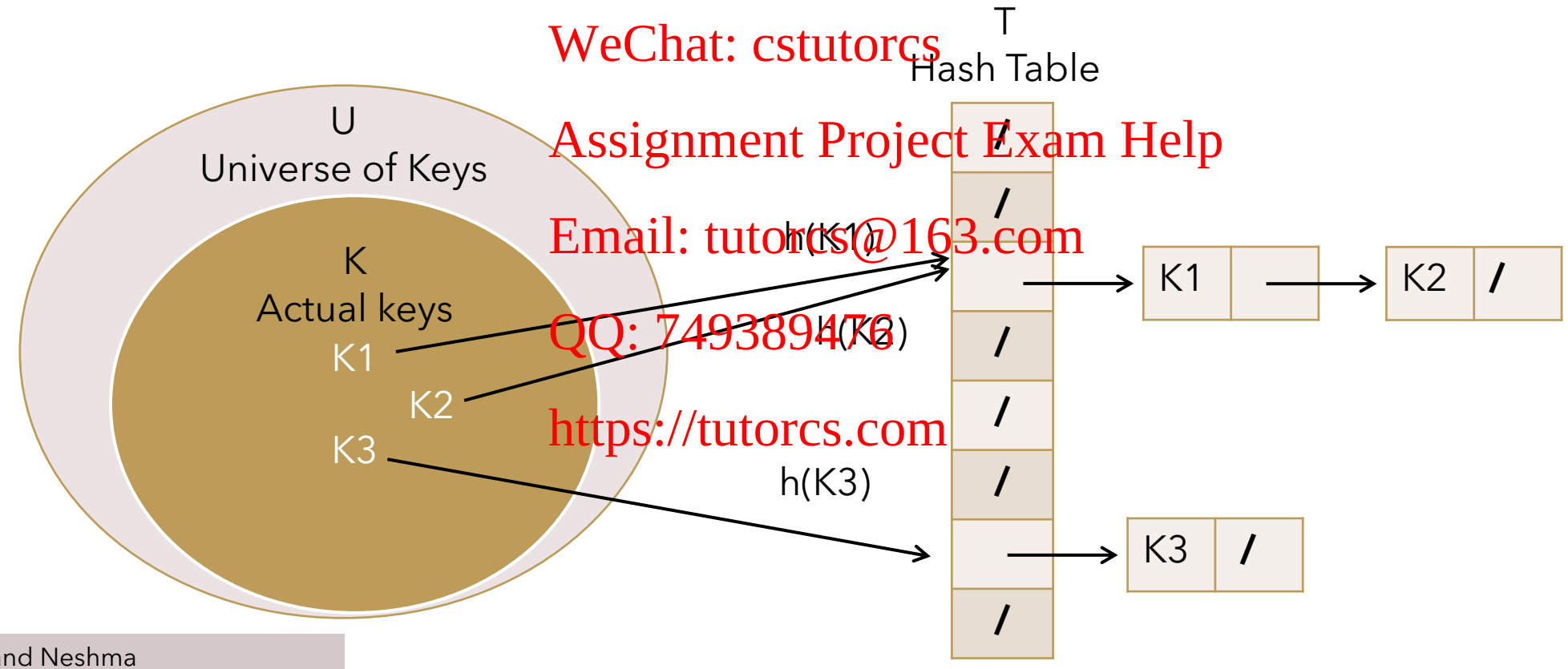
- Algorithms and Paradigm
- Divide and Conquer
- Dynamic Programming
- Greedy Algorithm
- Amortized Analysis

- Quizzes
- KAHOOT
- Q&A

Hash table

- **What**: A list-based **data structure** that uses a hashing function to store key value pairs
- **Why:** Used to **reduce storage s** **O(n)** for n keys :unlike direct address table (that can have a lot of wasted space comp number of keys saved in it)and **search time to O(1)**

Hash function h used below h: U {0, 1, ..., m-1} for table of length m and uses **chaining**



Hash function

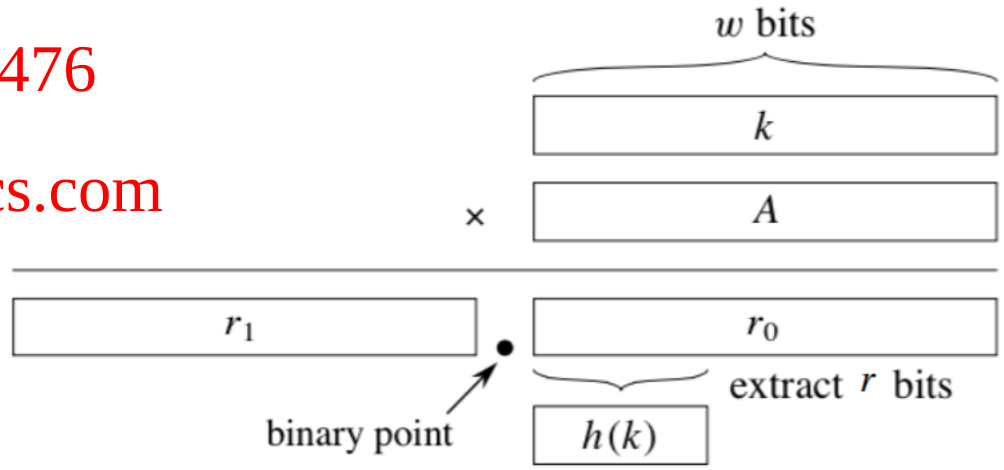
- **Properties:**
- Uniform distribution of keys in the table
- Regularity in key distribution should not affect the above property
- **Examples:**
- Division method
- Multiplication method
- Open addressing

Hash function

- Division method
- $h(k) = (k \ mod \ d)$
  - d must be chosen carefully
  - Tip: choose prime d not too close to a power of 2

- Multiplication method
- $h(k) = (A.k \ mod \ 2^w) \gg (w - r)$
  - A, k are both w bit numbers
  - $(2^{w-1} < A < 2^w)$
  - Slower to compute but effective

REMINDER:
BIT SHIFT:
101101 >> 1 = 10110
(right shift)
Division by 2^k



Open addressing

- No chaining
- **How?:**
  1. Probe the table
  2. Insert if slot empty else calculate anc nction
- More slots than keys
- **Deletion** is difficult and deleted cell must be marked deleted not NIL

REMINDER!
Search must use the same probe sequence

Possible hash functions

Linear probing

$$h(k,i) = \left(h'(k) + i\right) \bmod m$$

- Tendency to clustering

Quadratic probing

$$h(k,i) = \left(h'(k) + i.c_1 + i^2.c_2\right) \bmod m$$

- Must guarantee full permutation
- Secondary clustering

Double hashing

$$h(k,i) = \left(h_1(k) + i.h_2(k)\right) \text{moe } m$$

- Second hash function must be relatively prime to guarantee full permutation

Analysis: Hashing with chaining, open addressing

NOTE: Average case analysis, what about worst case

Hashing with chaining

- Note: Assuming uniform hashi
- **Insertion:** $O(1)$
- **Deletion:** $O(1) + search\ time$
- **Search:**
  - **Worst case: :** $O(n)$
  - **Average case:** $\Theta(1+\alpha)$

shing with open addressing

Note: Assuming uniform hashing

- Expected number of probe in

1. **Unsuccessful search:** $\frac{1}{(1-\alpha)}$

Note: Expected number of probes to insert is at most the same as above

2. **Successful search:** $\frac{1}{\alpha}.\ log\left(\frac{1}{(1-\alpha)}\right)$

REMINDER!
Load factor : $\alpha = \frac{n}{m}$
n keys and m slots

Disjoint Sets

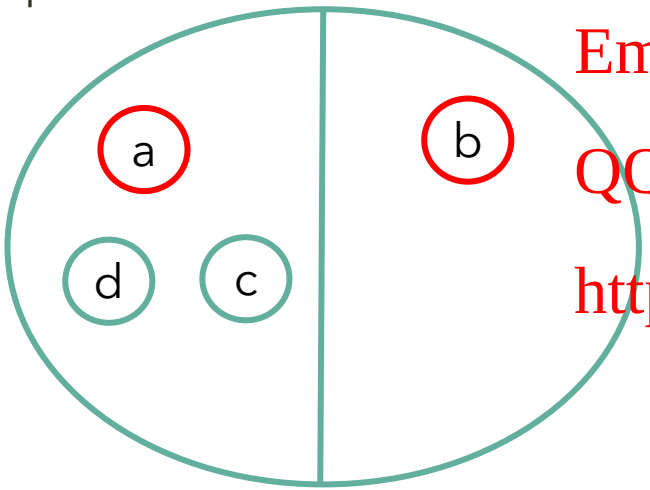
Waldispühl.J. (2020). *COMP251:* Disjoint Sets [PowerPoint slides]. https://www.cs.mcgill.ca/~jeromew/teaching/251/F2020/COMP251_Lecture9_F2020.pdf

Some definitions

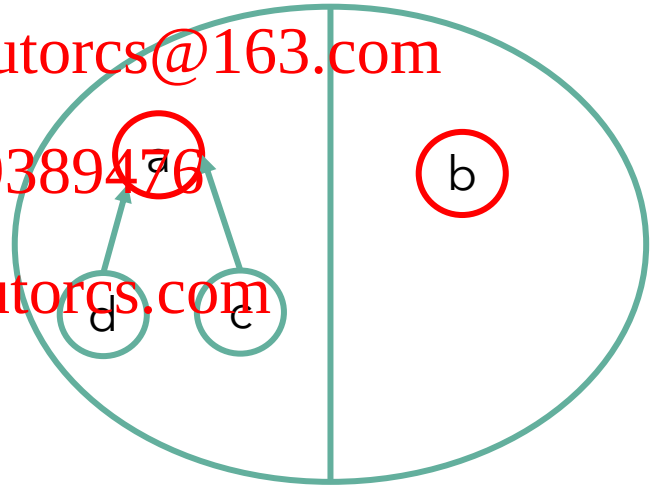
- **Connected components:** s[illegible]s connected by a path (or a single node by itself)
- **Partition:** For $B_1 \cup B_2 \cup B_3 \ldots \cup B_n = \mathbf{S}$, for $\mathbf{S}$ being the sample space and $B_i \cap B_j = \emptyset \; iff \; i \neq j$ and none of the subsets are empty

Disjoint sets

- **What**: A tree or list-based al**ta structure**
- **How:** Each node in the part**des has a representative node**
- **Has operations:**
  1. **Find(i):** returns the representative of the subset (or partition) that contains i
  2. **Union(i,j):** merges set containing i and j, the choice of representative is decided by implementation



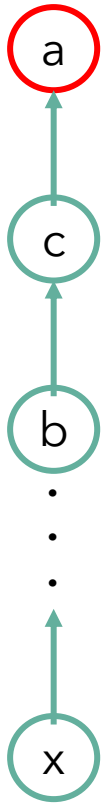
Set representation



Tree representation

Union: Union by size Union by height

- Worst case



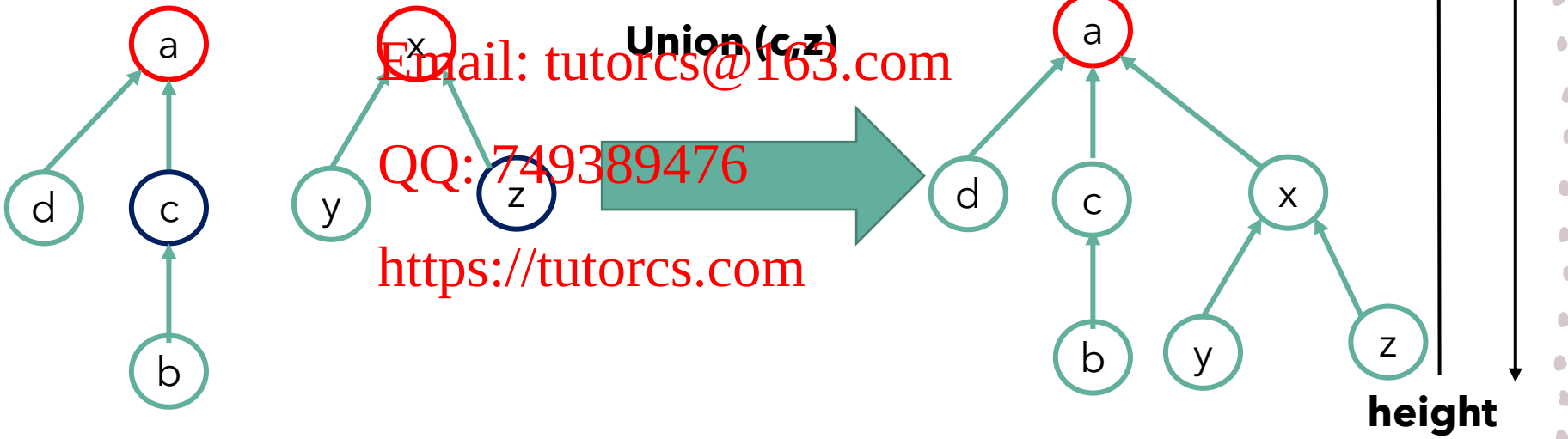
Find(x) is O(n)

After union:

- The depth of any node after union is at most $\log n$ for n nodes in the set

After union:

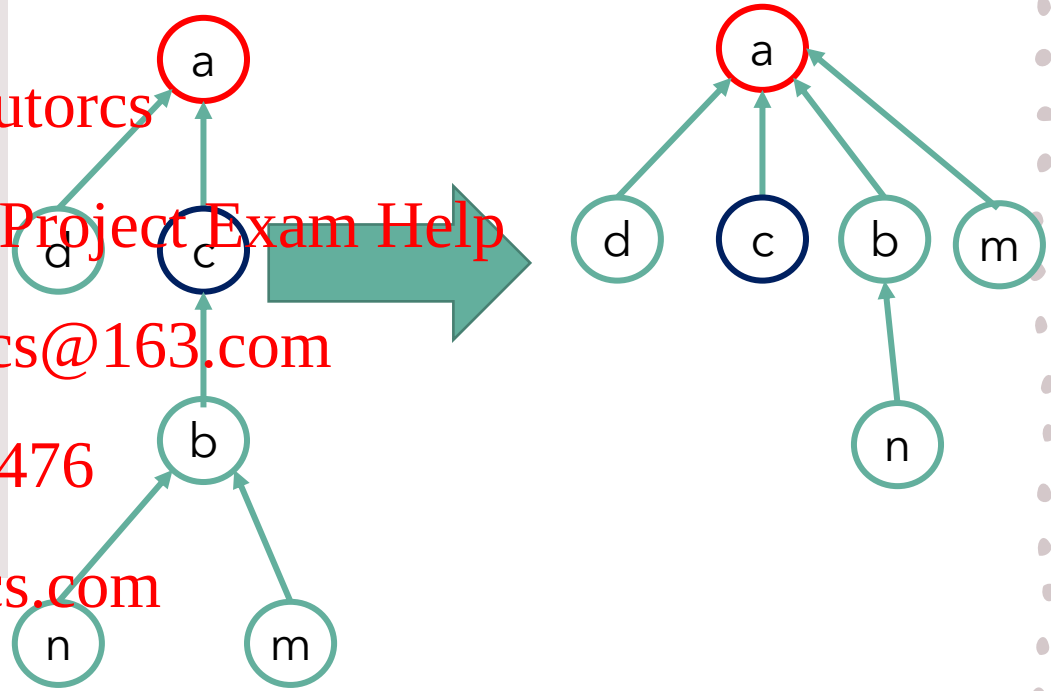
- The height of tree is at most log n for n nodes in the set
- The tree of height h has at least $n_h \geq 2^h$ nodes



Path compression


```
find(i) {
        if parent[i] == i {
                return i;
        }
        else {
                parent[i] = find(parent[i]);
                return parent[i]
        }
}
```


Example: find(m)



Running time

	Find(i)	Union(i,j)
Quick find	$O(1)$	$O(n)$
Union by size/ height	$O(\log n)$	$O(\log n)$

- Union by size with path compression: $O(\log n)$ shown to take $O(m\,\alpha(n))$
- Note : $\alpha(n))$ for n $2048\ -A_{4(1)}$; $\alpha(n)$ is 4

More resources for disjoint sets

- Good resource : http://www.[illegible]ll.ca/~langer/251/5-disjointsets.pdf

Heaps and Heapsort

Heaps

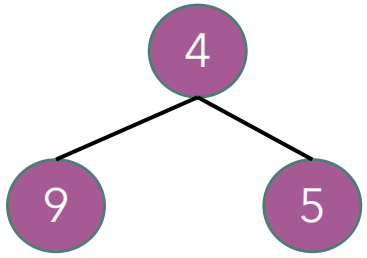
- **What**: A tree-based **data structure** (complete binary tree)
- **How:**

  We fill the tree from top to bottom, left to right.
  Represent using **arrays**
- **Two Types:**

REMINDER!
Does **not** follow BST properties

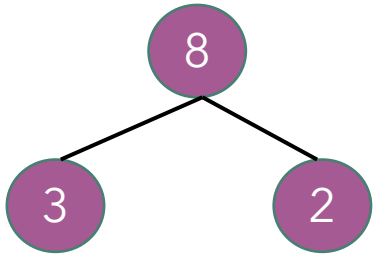
Min Heap

- Root: Node with **minimum** value
- Property: Every node is **smaller** than their children



Max Heap

- Root: Node with **maximum** value
- Property: Every node is **greater** than their children



Heaps: Height and Arrays

REMINDER!
Root at element
1 NOT 0

Height Properties:

- Number of edges in the **longe** path from that node to a leaf
- Heap height = root height: **Θ (log n)**
- Basic operations: Θ (log n) time

Example:

- Type: Minimum Heap
- Height = 3
- Array of length 12:

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Value	-	4	9	5	25	13	8	12	40	26	15	30

Array Properties:

- (int[] A, Index i)
  - Root: A[1]
  - Parent[i]: A[Floor(i/2)]
  - LeftChild[i]: A[2i]
  - RightChild[i]: A[2i+1]

Heaps: Build Heap

Provided an array, build MaxHeap:

1. Visit element at current index (starting: l
2. Check if heap property is met:
   i. **If satisfied:** continue to 3
   ii. **If not satisfied:** Heapify and go to 2
3. Stop if index = 0, if not, go to next element i-1 (repeat)

MaxHeapify(Array A, Index i):

- If the element at index i does not satisfy heap property
  - LeftChild[i] or RightChild[i] > current
- Let the index k, be the (left or right) child whose value is greatest
- Change the value of the element at index i with the value of element at index k
  - Example:

Provided a node, build MinHeap:

1. Add it to the most left spot that is available.
2. Check if heap property is met
   i. **If satisfied**: continue
   ii. **If not satisfied**: Heapify
3. Stop if no more nodes to add or (repeat)

MinHeapify:

- If the value of node does not satisfy heap property
  - CurrentNode < parentNode
- Check if current node is root
  - If not: Switch node with parent node and now currentNode = ParentNode (repeat)
- If yes: Stop

Heaps: Example Max-Heap (provided Array)

START

3	4				12	15	7	10
1	2				5	6	7	8

3	4	8	10	12	15	7	
1	2	3	4	5	6	7	8

MaxHeapify(4), swap(4,8), heap satisfied at index 8

3	4	15	10	12	8	7	9
1	2	3	4	5	6	7	8

MaxHeapify(3), swap(3,6), heap satisfied at index 6

3	12	15	10	4	8	7	9
1	2	3	4	5	6	7	8

MaxHeapify(2), swap(2,5), heap satisfied at index 5

15	12	3	10	4	8	7	9
1	2	3	4	5	6	7	8

*MaxHeapify(1), swap(1,3), heap **not** satisfied at index 3*

15	12	8	10	4	3	7	9
1	2	3	4	5	6	7	8

MaxHeapify(3), swap(3,6), heap satisfied at index 6

15	12	8	10	4	3	7	9
1	2	3	4	5	6	7	8

FINISH

Heapsort (Ascending)

REMINDER! Descending order: Build Min-Heap

- Steps: Given an array A size n
  1. Build a **Max-Heap**
  2. Start with k = n-1
  3. Swap the element at index 1 with element k of array A
     i. Elements from index n-k up to n is sorted
     ii. Remaining elements from 1 up to n-k do not satisfy heap property
     iii. MaxHeapify(1); the root.
     iv. k = k -1
     v. If: k!=1
        i. Repeat step 3
     i. Else: Stop

Example Heapsort (provided Max-Heap)

15	12	8	10	4	START
1	2	3	4	5	

12	4	8	10	15	then MaxHeapify(1)	K =5
1	2	3	4	5		
12	10	8	4	15	MaxHeapify(2)	
1	2	3	4	5		
10	4	8	12	15	swap(1,4) then MaxHeapify(4): Property Met	K =4
1	2	3	4	5		
8	4	10	14	15	swap(1,3) then MaxHeapify(1): Property Met	K =3
1	2	3	4	5		
4	8	10	14	15	swap(1,2) then MaxHeapify(1): Property Met	K =2
1	2	3	4	5		
						K =1

4	8	10	12	15	FINISH
1	2	3	4	5	

Analysis: Max-Heapify

REMINDER!
# of nodes at height h in binary tree
$n = 2^{h+1} - 1$

Max-Heapify(Array A, size n)

- Worst Case:
  - Size of largest **subtree** $\leq \frac{2n}{3}$

    (Last row of tree is half full)
    - $T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$
    - $T(n) = O(logn)$
  - Or equivalently,
    - $O(h)$
      - h: the height of the node of MaxHeapify

Total n in heap

# of nodes in left subtree* + # nodes in right subtree +1

$= (2^{h+2} - 1) + (2^{h+1} - 1) + 1$

$= 3 * 2^{h+1} - 1$

Rearrange:

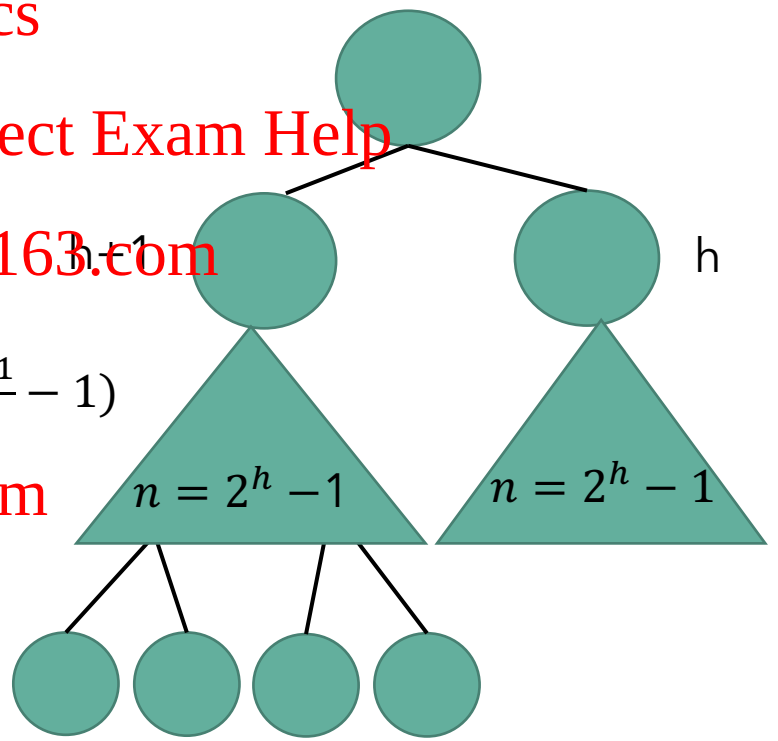
$2^{h+1} = \frac{n+1}{3}$

h+1

Substitute into *

$(2^{h+2} - 1) = (2 * \frac{n+1}{3} - 1)$

$= \frac{2n-1}{3} < \frac{2n}{3}$



Analysis: Build Heap, HeapSort

Build Maxheap

$$Cost\ of\ a\ Maxheapify * \#\ MaxHeap$$

Loose Upper Bound:

- Cost of Maxheap: $O(logn)$
- # of Calls: $O(n)$
- $= O(logn) * O(n)$
- $= O(nlogn)$

Tight Upper bound:

- Cost of Maxheap: $O(h)$
- # of Calls:
  - # nodes at height h $\leq$ Ceiling$(\frac{n}{2^{h+1}})$
  - $= \sum_{h=0}^{h=logn} \frac{n}{2^{h+1}}$
- $\sum_{h=0}^{h=logn} \frac{n}{2^{h+1}} * O(h)$
- $O\left(\frac{n}{2} \sum_{h=0}^{h=logn} \frac{h}{2^h}\right)$ //series
- $O\left(\frac{2}{2} n\right) = O(n)$

HeapSort

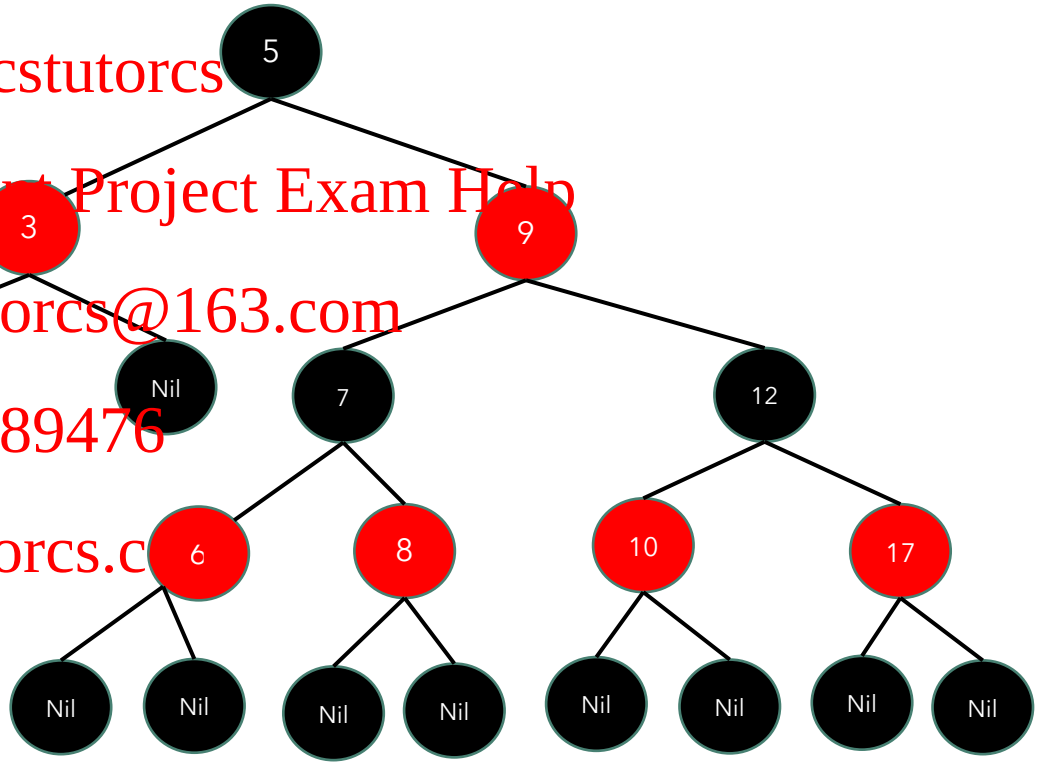
$$Build\ MaxHeap + (Exchange\ element * n) + (MaxHeapify * n)$$

- (For loop n-1 times)
  - Build MaxHeap: $O(n)$
  - Exchange element: $O(1)$
  - Max-Heapify: $O(logn)$
  - $n + n + nlogn$
  - $= O(nlogn)$

Red Black Trees

Red Black Trees

- **What:** A variation of a **binar** **tree** (BST) that is **self-balancing**
  - BST attributes + **1 bit** (represen black/red)
  - All leaves / empty trees: color b ent using *Nil, color[Nil] = black)*
- **Operations:** $O(logn)$
- **Height:** $O(logn)$ // n number of nodes



REMINDER!
Don't usually show sentinels

Red Black Trees: Properties & Height

Properties

1. A node is either Red or Black
2. The Root is Black
3. All leaves (nil) are Black
4. A Red node has children nodes colored black / No consecutive red nodes
5. Each node: Every path to a leaf has same # of black nodes / same bh

ht

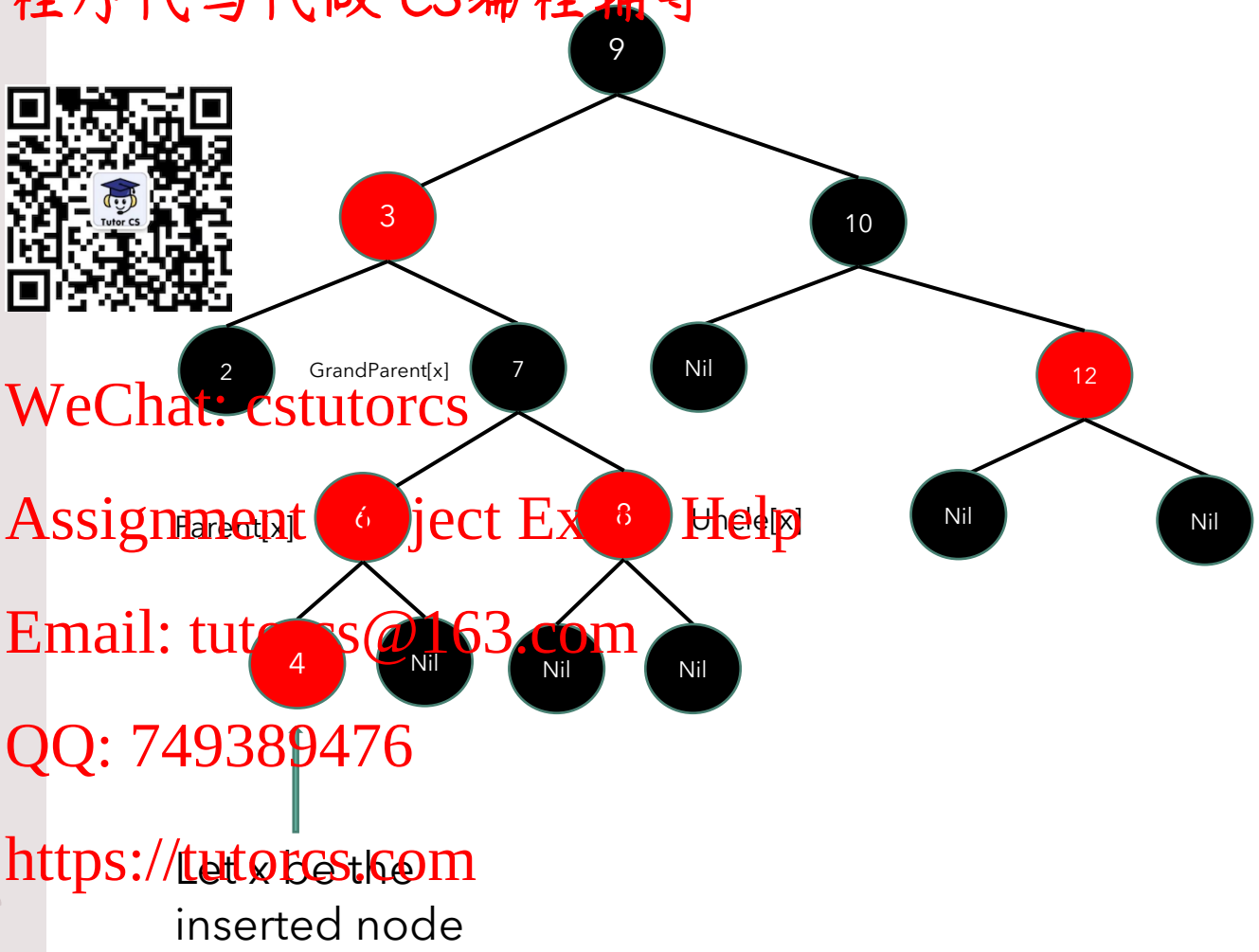
ode

- **Height** of node: h(x)
  - # edges in the longest path to a leaf
- **Black height** of node: bh(x)
  - # of black nodes on the path from node to leaf
  - Not including x
- Black height of tree = bh(root)
- **Lemma1:** Node with h(x), $\boldsymbol{bh(x) \geq \frac{h(x)}{2}}$
- **Lemma2:** Rooted at node x, subtree has $\geq \boldsymbol{2^{bh(x)} - 1}$. internal nodes
- **Lemma3:** T with n internal nodes: height $\mathbf{h} \leq \boldsymbol{2 * log(n+1)}$

Red Black Trees: Insert Node

1. Insert node like you would w
   I. Compare with root all the way t
   II. Inserted to a leaf node
2. Color node to be red
3. RB-Fixup: Check Red Black Tree Properties by: // While the parent[x] is Red
   I. Re-Coloring Nodes
   II. Rotation

Red Black Tree Example



Case 1

Property 4 violated

- The Uncle is Red
- The GrandParent[x] is Black

Steps:

1. Change Parent[x] and Uncle[x] to Black
   1. *Property 5 violated!*
2. Change GrandParent[x] to Red
   1. *Property 5 restored*
3. x pointer moves 2 level up
   1. x = GrandParent[x]

REMINDER!
Applies to right child

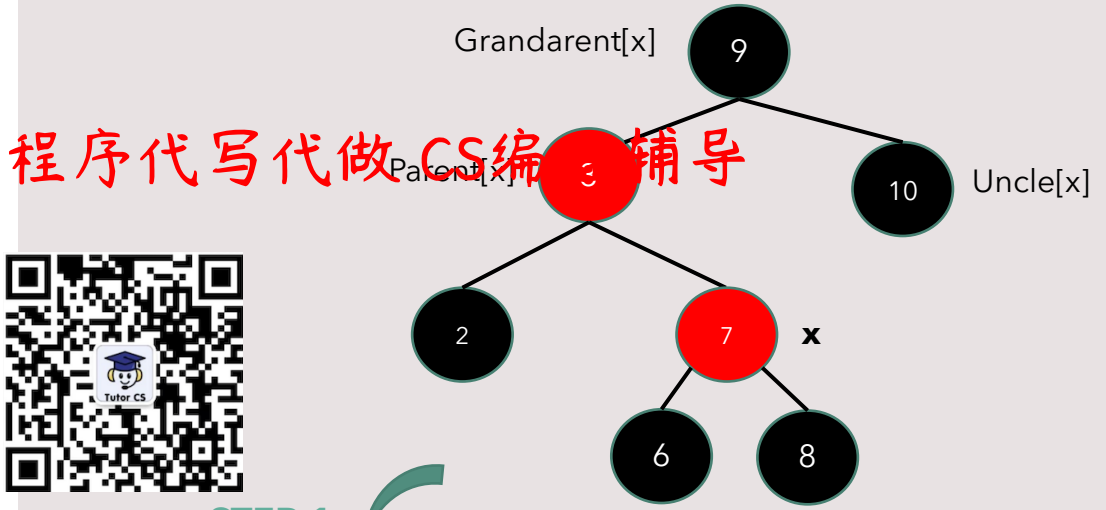
Case 2

Property 5 is violated!

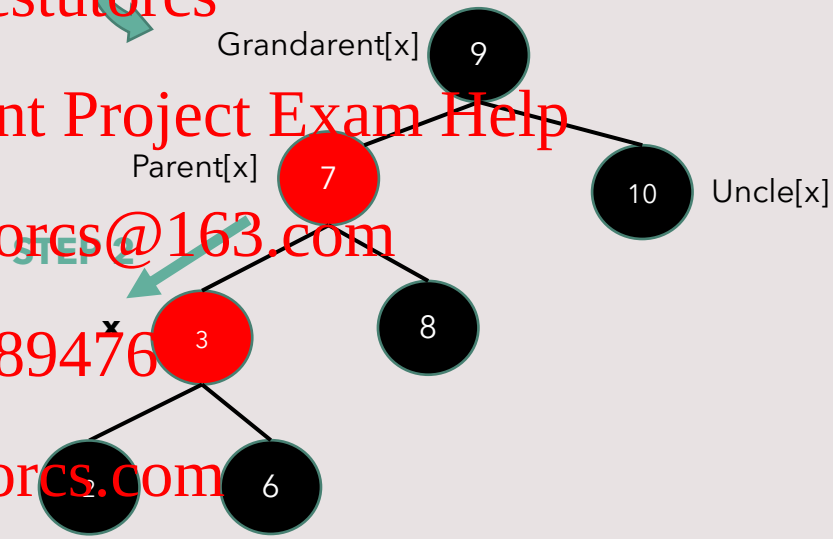
- The Uncle[x] is black
- x is right child

Steps

1. Left Rotate at Parent[x]
2. Parent and x switch
    1. Parent[x] = x,
    2. x = Parent[x]
3. Case 3



STEP 1



STEP 2

REMINDER!
x remains the same level

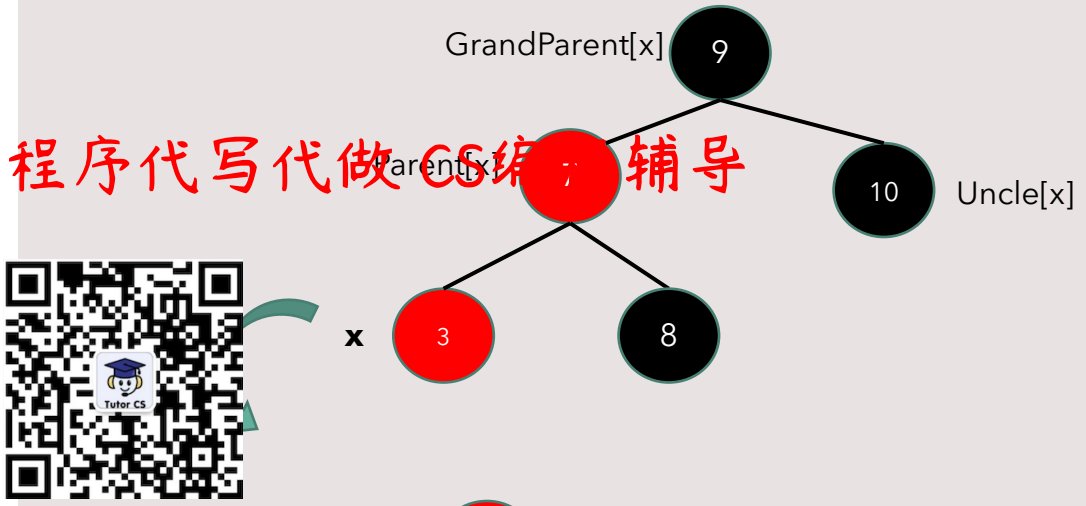
Case 3

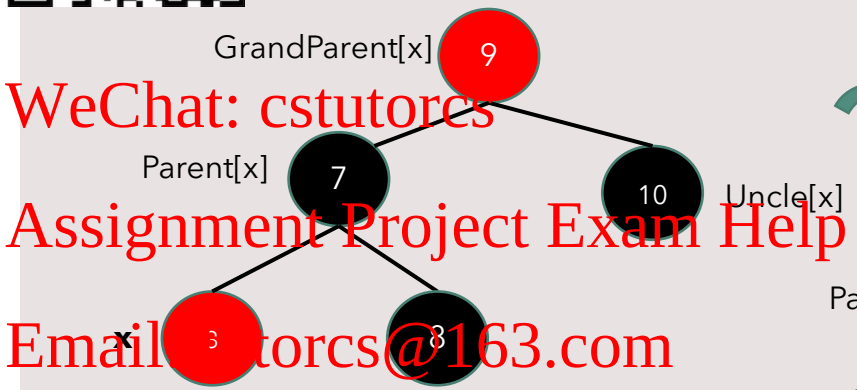
Property 5 is violated!

- The Uncle[x] is black
- x is a left child

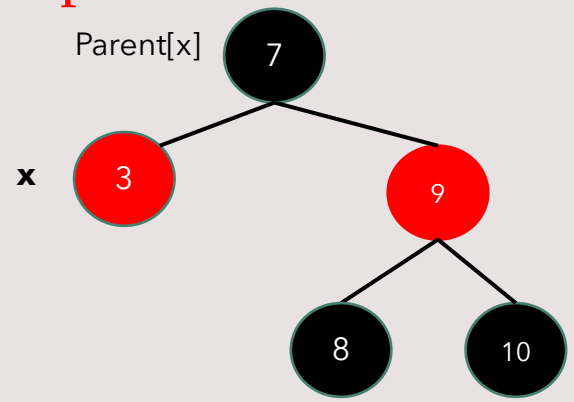
Steps

1. Color Parent[x] Black, GrandParent[x] Red
   I. *Property 4 violated!*
2. Rotate Right on GrandParent[x]
   I. *Property 4 is met.*
3. Parent[x] is black so stop





STEP 2



Red Black Tree: Insert Analysis

- To insert (before fix-up) / Ins / height of RB Tree: $O(logn)$
- Change colour to Red: $O(1)$
- RB-Fixup:
  - While loop if Case 1
    - Recoloring: $O(1)$
    - Case 1: x moves 2 levels up
    - At most 2 rotations: Case 2 and Case 3
  - At most loops: $O(logn)$
- $\boldsymbol{O(\log n)\ time}$

BST & AVL Trees

Waldispühl.J. (2020). *COMP251:* Binary search trees, AVL trees & AVL sort [PowerPoint slides]. https://www.cs.mcgill.ca/~jeromew/teaching/251/F2020/COMP251_Lecture6_F2020.pdf

Binary Search Tree (BST)

- **Properties:**

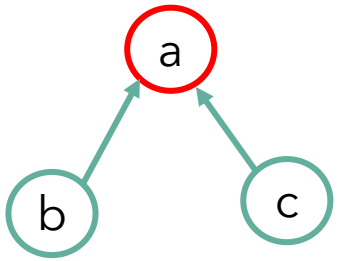
  Nodes smaller than the parent are in left subtree

  Nodes bigger than the parent are in right subtree

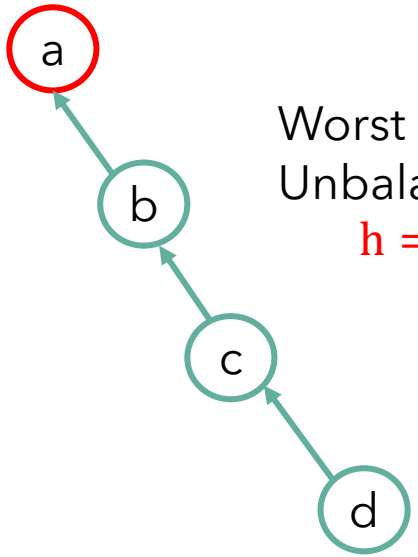
- **Operations:**

Operations	Time taken for tree with height h
Search (Tree, Key)	$\boldsymbol{\Theta(h)}$
Insert (Tree, Key)	$\boldsymbol{\Theta(h)}$
Delete (Tree, Key)	$\boldsymbol{\Theta(h)}$

Note: h = 1 + max (height(left child) , height(right child))



Best case:
Balanced tree
$\mathrm{h} = \boldsymbol{\Theta(log\, n)}$

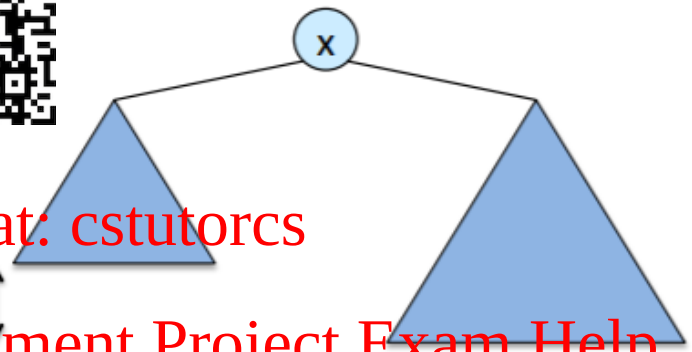


Worst case:
Unbalanced tree
$\mathrm{h} = \boldsymbol{\Theta(\, n)}$

Slides by Fizzana and Neshma

Waldispühl, J. (2020). *COMP251: Binary search trees, AVL trees & AVL sort* [PowerPoint slides]. https://www.cs.mcgill.ca/~jeromew/teaching/251/F2020/COMP251_Lecture6_F2020.pdf

REMINDER!
Height of AVL tree is always O(log n)



$|h_{left}-h_{right}| \leq 1$

AVL trees

BST SUCH THAT HEIGHT OF ANY TWO CHILD SUBTREES DIFFERS BY AT MOST ONE

Waldispühl J. (2020). *COMP251: Binary search trees, AVL trees & AVL sort* [PowerPoint slides]. https://www.cs.mcgill.ca/~jeromew/teaching/251/F2020/COMP251_Lecture6_F2020.pdf

程序代写代做 CS编程辅导

WeChat: cstutorcs

Assignment Project Exam Help

Violates AVL property!

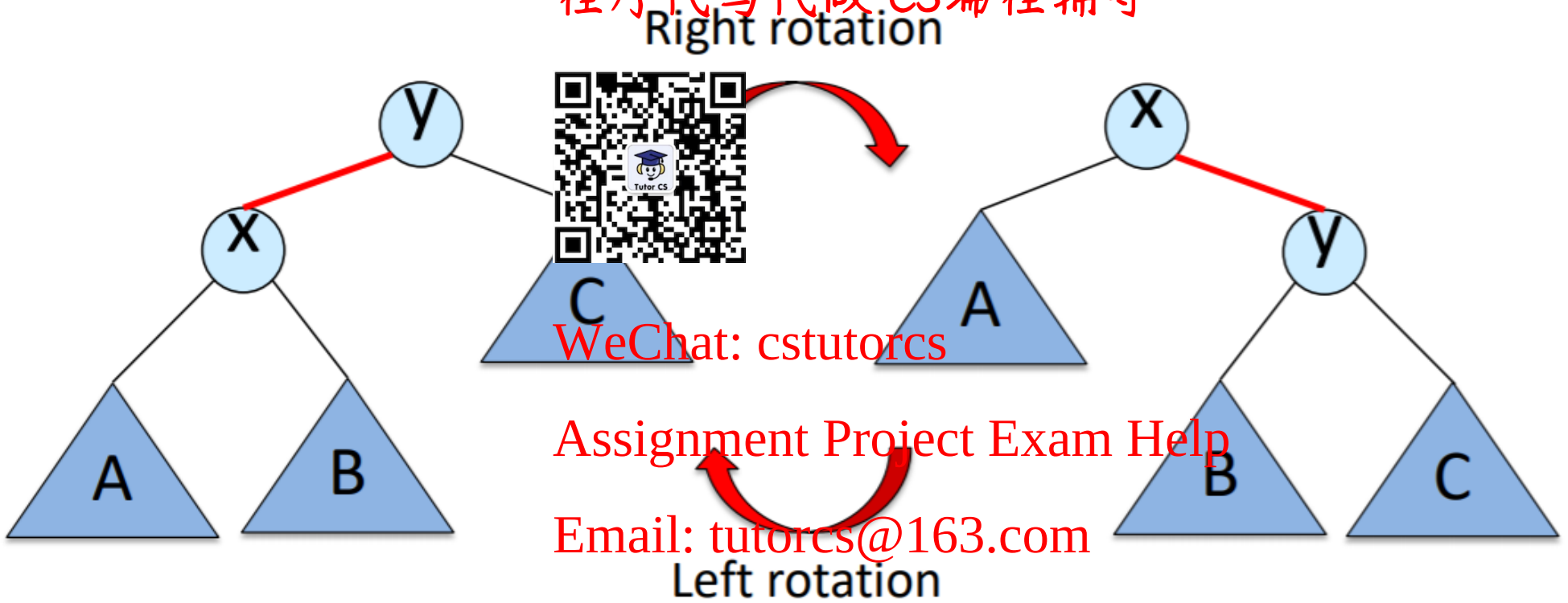
Email: tutorcs@163.com

QQ: 749389476

https://tutorcs.com

Balance factor

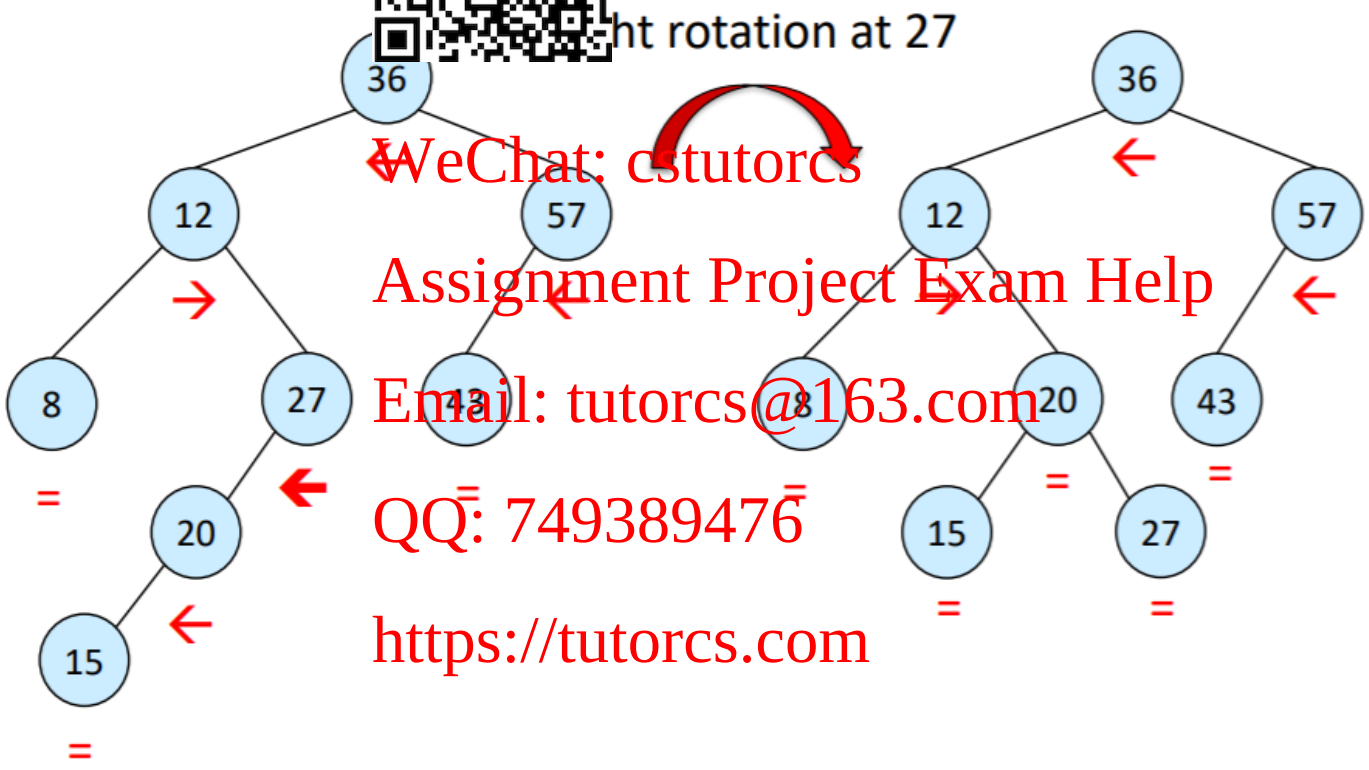
Waldispühl.J. (2020). *COMP251: Binary search trees, AVL trees & AVL sort* [PowerPoint slides]. https://www.cs.mcgill.ca/~jeromew/teaching/251/F2020/COMP251_Lecture6_F2020.pdf



Rotations

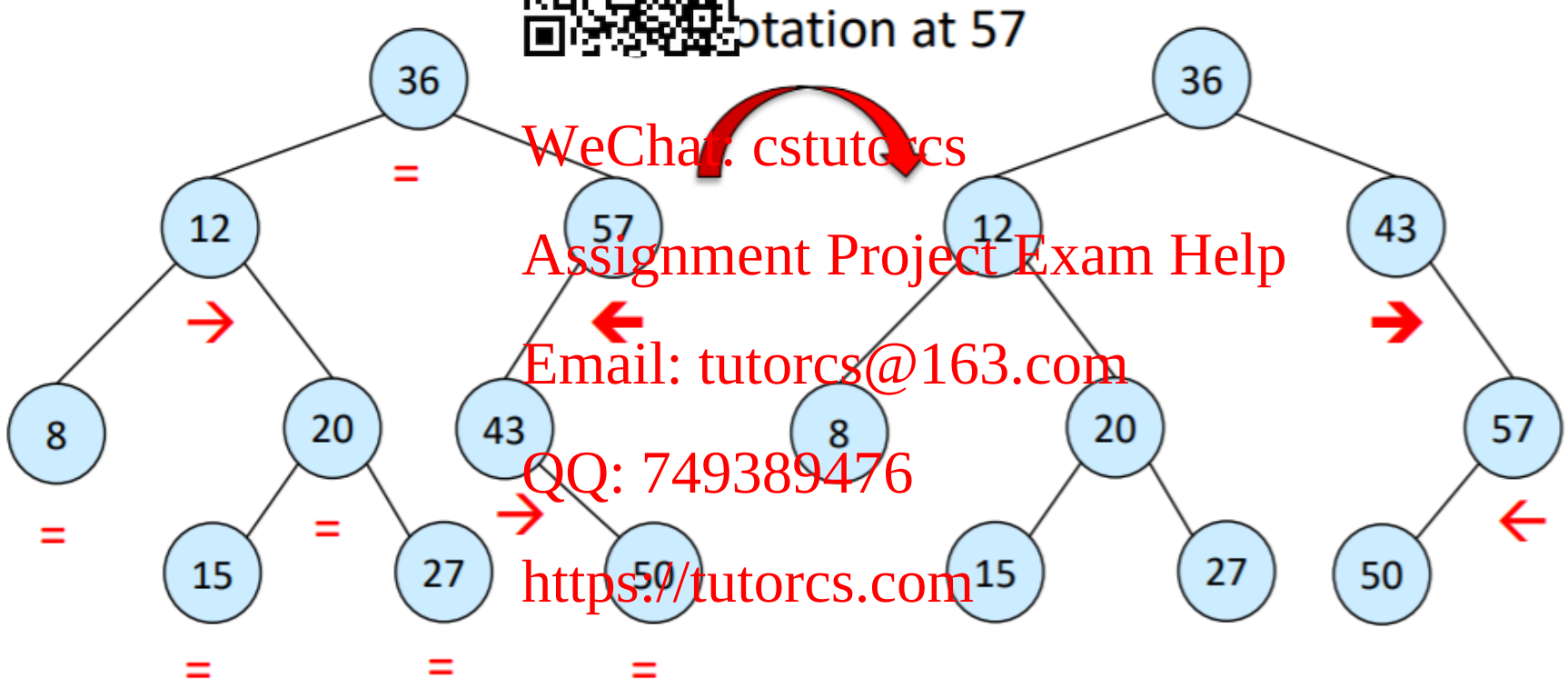
Waldispühl.J. (2020). *COMP251: Binary search trees, AVL trees & AVL sort* [PowerPoint slides]. https://www.cs.mcgill.ca/~jeromew/teaching/251/F2020/COMP251_Lecture6_F2020.pdf

Example 1: insert (15)

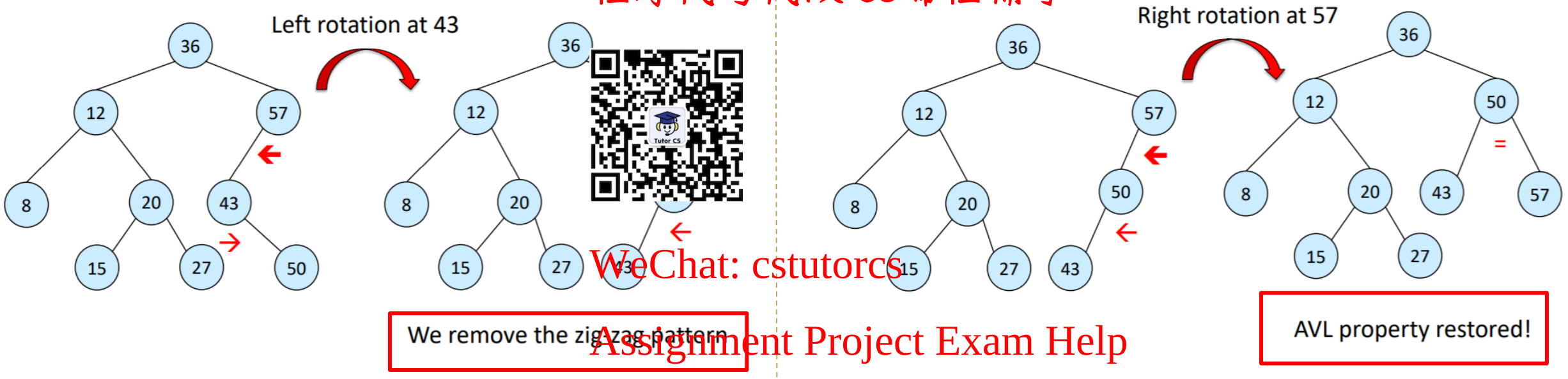


Waldispühl.J. (2020). *COMP251: Binary search trees, AVL trees & AVL sort* [PowerPoint slides]. https://www.cs.mcgill.ca/~jeromew/teaching/251/F2020/COMP251_Lecture6_F2020.pdf

Example 2: insert (50) fail



Waldispühl.J. (2020). *COMP251: Binary search trees, AVL trees & AVL sort* [PowerPoint slides]. https://www.cs.mcgill.ca/~jeromew/teaching/251/F2020/COMP251_Lecture6_F2020.pdf



Example 2: insert(50) success

Waldispühl.J. (2020). *COMP251: Binary search trees, AVL trees & AVL sort* [PowerPoint slides]. https://www.cs.mcgill.ca/~jeromew/teaching/251/F2020/COMP251_Lecture6_F2020.pdf

Algorithm : AVL insert

Suppose **x** is **lowest** node **violating**

- If **x** is **right-heavy**:

  If **x's right child** is **right-heavy** or **balanced**: **Left rotation**

  Else: **Right** followed by **left rotation**

- If **x** is **left-heavy**:

  If **x's left child** is **left-heavy** or **balanced**: **Right rotation**

  Else: **Left** followed by **right rotation**

4. Then continue up to **x's ancestors**.

Slides by Hrzana and Neshma

Waldispühl J. (2020). *COMP251: Binary search trees, AVL trees & AVL sort* [PowerPoint slides]. https://www.cs.mcgill.ca/~jeromew/teaching/251/F2020/COMP251_Lecture6_F2020.pdf

Run time BST/AVL sort

- **BST sort:**

1. Build a BST out of the given keys (unsorted)
2. Run an in-order traversal to print the keys in sorted order (run time $\boldsymbol{\Theta(n)}$)
3. **Best case:**

   $\boldsymbol{\Omega(n\ log\ n)}$ for insertion $\boldsymbol{\Theta(log\ n)}$

1. **Worst case:**

   $\boldsymbol{\Theta(n^2)}$ for insertion $\boldsymbol{\Theta(n)}$

- **AVL sort:**

1. Same as BST sort but with AVL tree
2. **Best/worst case:**

   $\boldsymbol{O(n\ log\ n)}$ for insertion $\boldsymbol{\Theta(log\ n)}$

KAHOOT !

Data Structures

Site: kahoot.it

PIN:

Algorithms Paradigm

Recursive backtracking example

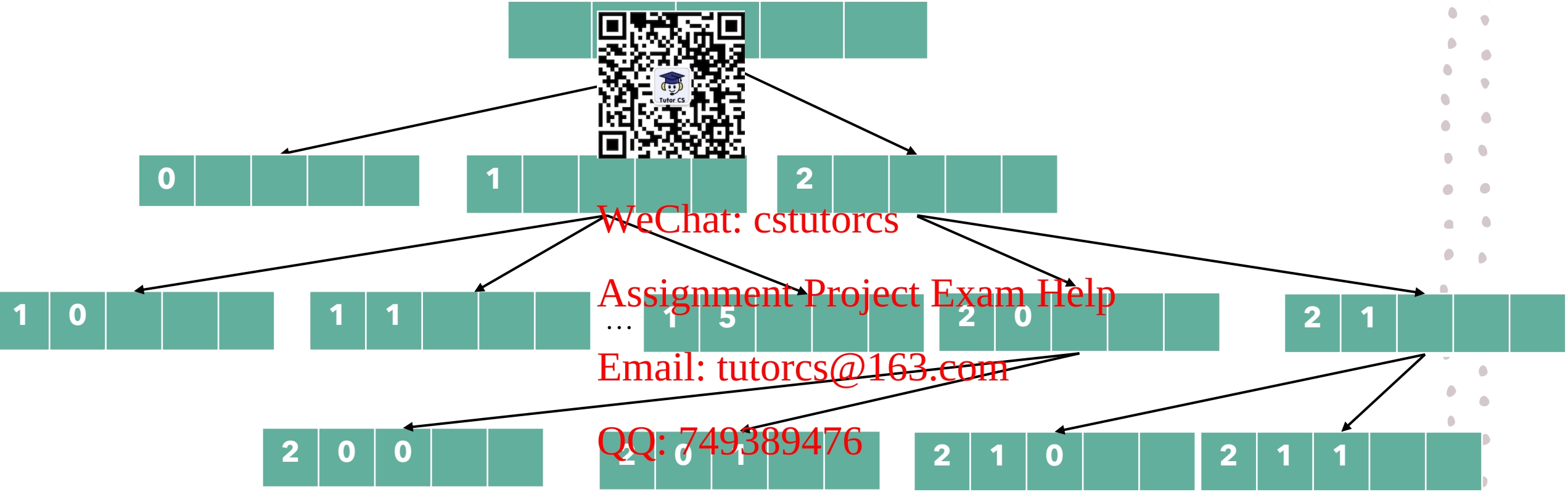
- An autobiographical number is a number such that the first digit is the number of zeros in the number, the second digit is the number of ones in the number, …, and the tenth digit is the number of 9's in the number. The first digit must be non-zero.
- For example: 1210.
- Question: find a 5-digit autobiographical number

Question: find a 5-digit autobiographical number

What do we know?

1. Each digit has 5 options [0,1,2….4] Please note that in the recording I said it has 10 options and that is incorrect!!
2. First digit needs to be non-zero
3. Value of each digit depends on all other digits for example if the first digits is 8 then there has to be 8 zeroes in the number.
4. All the digits must add to 5

Question: find a 5-digit autobiographical number



Divide and Conquer

Waldispühl.J. (2020). *COMP251:* Divide-and-Conquer (1) [PowerPoint slides]. https://www.cs.mcgill.ca/~jeromew/teaching/251/F2020/COMP251_Lecture21_F2020.pdf
Waldispühl.J. (2020). *COMP251:* Divide-and-Conquer (2) [PowerPoint slides]. https://www.cs.mcgill.ca/~jeromew/teaching/251/F2020/COMP251_Lecture22_F2020.pdf

Divide and Conquer

- **Divide** the problem into sub-problems that are like the original problem but smaller in size
- **Conquer** the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
- **Combine** the solutions to create a solution to the original problem

Sorting application

Obvious applications	Sorting problems become easy once elements are sorted	Non-obvious applications.
Organize an MP3 library.	Identify statistical outliers.	Convex hull.
Display Google PageRank results.	Binary search in a database.	Closest pair of points
List RSS news items in reverse chronological order	Remove duplicates in a mailing list.	Minimum spanning trees (Kruskal's algorithm).

Merge sort

```
ergeSort (Array, p , r) {
    if (p < r)
    then q = ⌊(p+r)/2⌋
        MergeSort( A, p , q)
        MergeSort(A, q+1, r)
        Merge (A,p ,q, r)
}
```

Initial call: MergerSort (A, 1, n)

REMINDER!
Do practice Loop invariant property for Merge or merge-sort

Analysis of Merge Sort

- **Divide**: computing middle t
- **Conquer**: solving two sub problems takes $2\ T(\boldsymbol{n/2})$
- **Combine** : merging n elements takes $\boldsymbol{\Theta(n)}$
- Running time:

  $\boldsymbol{T(n) = \Theta(1)}$ for n=1

  $\boldsymbol{T(n)} = 2\ T\left(\frac{\boldsymbol{n}}{\boldsymbol{2}}\right) + \boldsymbol{\Theta(n)}$ for n>1

  Thus, $\boldsymbol{T(n) = \Theta(n\ log\ n)}$

Divide and conquer
Master theorem

Master method

Goal. Recipe for solving common divide-and-conquer recurrences:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

Terms.

- $a \geq 1$ is the number of subproblems.
- $b > 0$ is the factor by which the subproblem size decreases.
- $f(n)$ = work to divide/merge subproblems.

Recursion tree.

- $k = \log_b n$ levels.
- a^i = number of subproblems at level i.
- n / b^i = size of subproblem at level i.

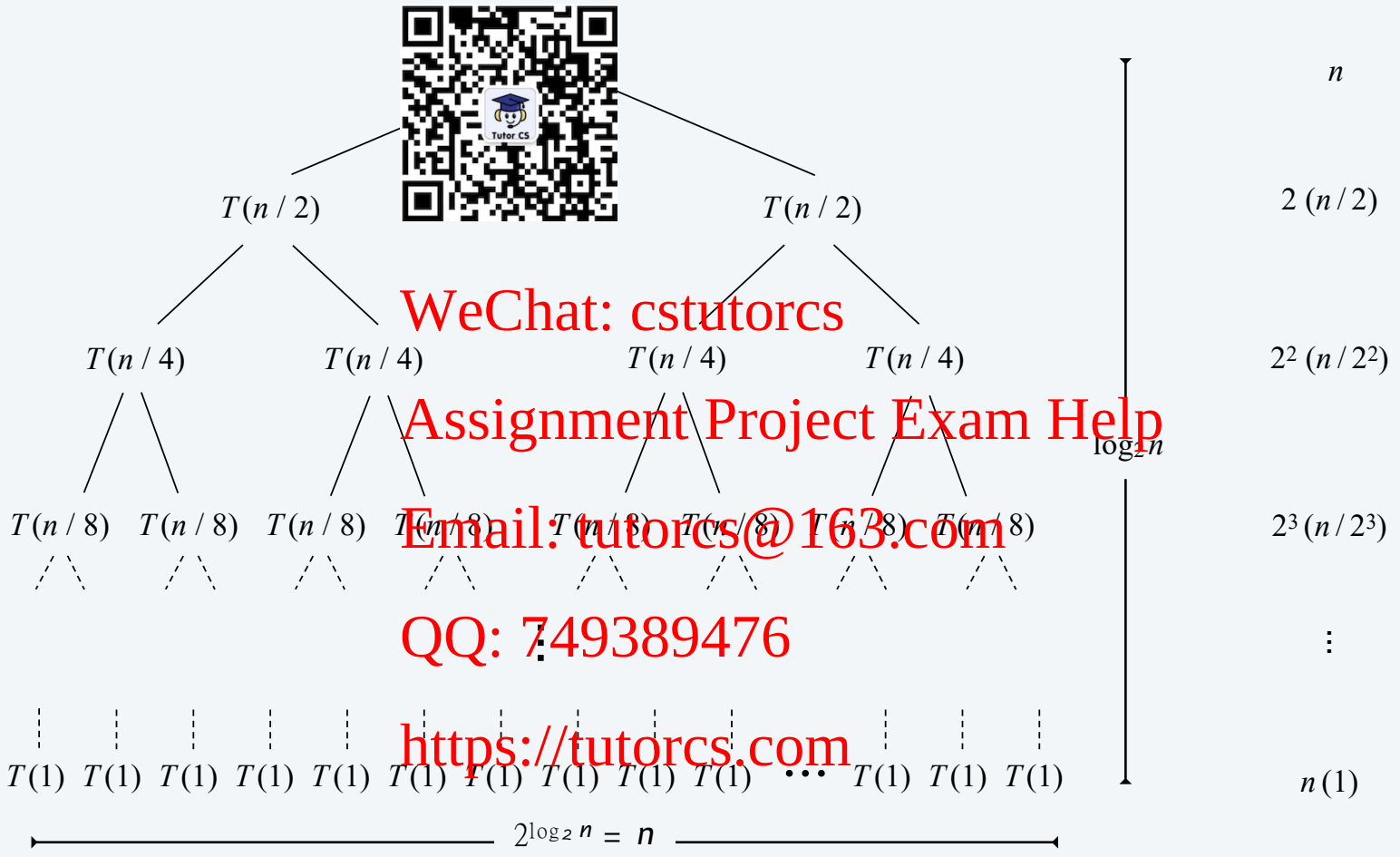
Case 1: total cost dominated by cost of leaves

Ex 1. If $T(n)$ satisfies $T(n) = 3\,T(n/2) + n$, with $T(1) = 1$, then $T(n) = \Theta(n^{\lg 3})$.

$$r = 3/2 > 1 \qquad T(n) = (1 + r + r^2 + r^3 + \ldots + r^{\log_2 n})\,n = \frac{r^{1+\log_2 n} - 1}{r - 1}\,n = 3n^{\log_2 3} \quad 2n$$

Case 2: total cost evenly distributed among levels

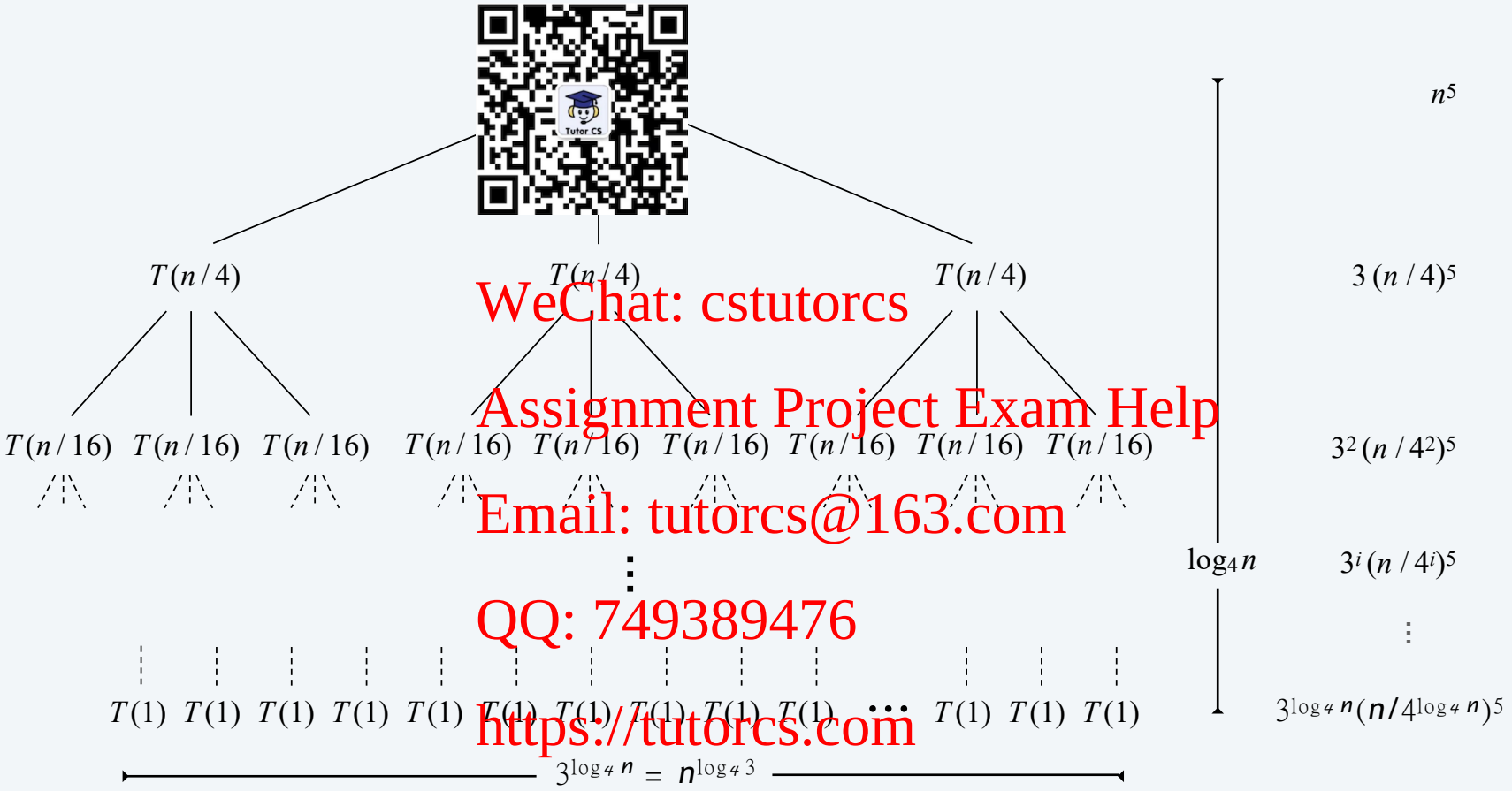
Ex 2. If $T(n)$ satisfies $T(n) = 2\,T(n/2) + n$, with $T(1) = 1$, then $T(n) = \Theta(n \log n)$.



$r = 1$ $\qquad T(n) = (1 + r + r^2 + r^3 + \ldots + r^{\log_2 n})\, n = n\,(\log_2 n + 1)$

Case 3: total cost dominated by cost of root

Ex 3. If $T(n)$ satisfies $T(n) = 3\,T(n/4) + n^5$, with $T(1) = 1$, then $T(n) = \Theta(n^5)$.



$$r = 3/4^5 < 1 \qquad n^5 \le T(n) \le (1 + r + r^2 + r^3 + \ldots)\,n^5 \le \frac{1}{1-r}\,n^5$$

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers that satisfies the recurrence

$$T(n) = a\,T(n/b) + f(n)$$

where n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Let $k = \log_b a$. Then,

Case 1. If $f(n) = O(n^{k-\varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^k)$.

Ex. $T(n) = 3\,T(n/2) + n$.

- $a = 3,\ b = 2,\ f(n) = n,\ k = \log_2 3$.
- $T(n) = \Theta(n^{\lg 3})$.

The formula works with $\varepsilon = \log_2 3 - 1 > 0$

$$f(n) = n = O\left(n^{\log_2 3 - (\log_2 3 - 1)}\right)$$

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers that satisfies the recurrence

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

where n/b means $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Let $k = \log_b a$. Then,

Case 2. If $f(n) = \Theta(n^k \log^p n)$, then $T(n) = \Theta(n^k \log^{p+1} n)$.

Ex. $T(n) = 2\,T(n/2) + \Theta(n \log n)$.

- $a = 2,\ b = 2,\ f(n) = n \log n\ k = \log_2 2 = 1,\ p = 1.$
- $T(n) = \Theta(n \log^2 n)$.

$$f(n) = \Theta(n \log n) = \Theta(n^{\log_2 2} \log n)$$

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers that satisfies the recurrence

$$T(n) = a\,T(n / b) + f(n)$$

where n / b means either $\lceil n / b \rceil$ or $\lfloor n / b \rfloor$. Let $k = \log_b a$. Then,

regularity condition holds if f(n) = Θ(n^{k+ε})

Case 3. If $f(n) = \Omega(n^{k+\varepsilon})$ for some constant $\varepsilon > 0$ and if $a\,f(n / b) \le c\,f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Ex. $T(n) = 3\,T(n / 4) + n^5$.

- $a = 3,\ b = 4,\ f(n) = n^5,\ k = \log_4 3$.
- $T(n) = \Theta(n^5)$.

1st property satisfied with $\varepsilon = 1 - \log_4 3$

$$f(n) = n^5 = \Omega(n^{(\log_4 3 + (1 - \log_4 3))})$$

2nd property satisfied with $c = \frac{3}{4}$

$$3.\left(\frac{n}{4}\right)^5 \le c.n^5$$

Master theorem

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers that satisfies the recurrence

$$T(n) = a\,T(n/b) + f(n)$$

where n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Let $k = \log_b a$. Then,

Case 1. If $f(n) = O(n^{k-\varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^k)$.

Case 2. If $f(n) = \Theta(n^k \log^p n)$, then $T(n) = \Theta(n^k \log^{p+1} n)$.

Case 3. If $f(n) = \Omega(n^{k+\varepsilon})$ for some constant $\varepsilon > 0$ and if $a\,f(n/b) \le c\,f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Applications

a) $T(n/2) + n^2$

$\Theta(n^2)$ (case 3)

b) $T(n) = T(n/2) + 2^n$

$\Rightarrow T(n) = \Theta(2^n)$ (case 3)

c) $T(n) = 16 * T(n/4) + n$

$\Rightarrow T(n) = \Theta(n^2)$ (case 1)

d) $T(n) = 2 * T(n/2) + n \log n$

$\Rightarrow T(n) = n \log^2 n$ (case 2)

e) $T(n) = 2^n * T(n/2) + n^n$

$\Rightarrow$ Does not apply!!

Dynamic Programming

Dynamic Programming

- **What:** An algorithmic technique **ks down** an optimization problem into (all possible) **simple sub-problems**.
- **Why:** Avoid recomputing, as we store the solution to the sub-problems
- **Achieves:** Correctness and Efficiency
- **When:**
  - Optimal Substructure Problem
  - Overlapping sub-problems
- **Types of approaches:** Top-Down and Bottom-up
  - Top Down: Uses Memoization
  - Bottom Up: Uses Tabulation

REMINDER!
Optimal substructure: the optimal solution can be constructed from the optimal solution of the sub-problem

DP: 1-D, 2-D, Tree, Interval, Subset

- **1-Dimensional:** Only need 1-D [illegible] tore the states, depends on one variable

  Examples: Fibonacci, # ways to write n as the sum of numbers,

- **2-Dimensional:** Need 2-D memo to store states, depends on two variables

  Examples: Knapsack: Value and weight, Stocks profit: days, transactions, Dissimilarity between 2 strings: string 1 and string 2

- **Tree:** Using a tree data structure to solve the DP problem

  Examples: Find largest Independent Sets, maximum sum of values from root to node without revisiting

- **Interval:** Implementing DP on intervals

  Examples: Weighted interval scheduling,

DP: Top-Down

Advantages

- Easier to understand and apply
- Compute sub-problems when necessary

Disadvantages

- Slow with revisited sub-problems
- Memory problems

- **Idea:** This is a recursive approach in which the method is called recursively until the exhaustion of subproblems.

- **Steps (implementing):**
  1. Initialize the memoization table: m
     - Values: -1
     - Size: The number of distinct sub-problems
  2. Check if a sub-problem has been computed at the start of recursive function
     - True: Get information from m
     - False: Compute the problem and store in m

DP: Bottom-up

Advantages

- Revisiting subproblems: faster
- Save memory space using "on th

Disadvantages

- Not intuitive
- Visits and Fills up all the states

- **Idea:** Starting at the smallest value, then iteratively compute values at each step

- **Steps (Implementation)**

1. Identify the Complete Search recurrence.
2. Initialize some parts of the DP table with known initial values.
3. Determine how to fill the rest of the DP table based on the Complete Search recurrence, usually involving one or more nested loops to do so.

Intuitive Steps Top-Down and Bottom-up

1. Identify what the sub-problem is
   - They are often similar so look for patterns
2. Write problem as a recurrence
   - What decision you make at every step
   - Currently at step i, what info you needed for i-1, and what you need for step i+1?
3. Solve original problem using step 1, 2 and find the base case
4. Dimension of memoization array & direction it is filled

1. Identify what the sub-problem is
   - They are often similar so look for patterns
2. Write problem using cached results
   - The inner loop should go through all possibilities from the start, up to the level of of the outer-loop
3. Write step 2 mathematically
   - Keep track of optimal result
4. Dimension of tabulation array & direction it is filled
   - Store the result of the optimal solutions for the sub-problems

DP: Example

2-D & Bottom-up

Problem (LeetCode):

Design an algorithm to find the maximum profit. Given an integer array *prices* (the i-th element represents the price of a share on day **i** and the maximum number of transactions you can do is **k**. Multiple transactions at once is not allowed (you must sell before buying again).

Constraints:

- $0 \leq k \leq 100$ *// # transactions*
- $1 \leq prices.length \leq 1000$ *// # of predicted prices*
- $1 \leq prices[i] \leq 1000$ *// maximum price*

- Example:

input: `k = 2`
`prices = [2,4,1]`

output: **`profit = $2`**

Explain: Buy share at price $2 on day one and sell at price $4 at day two (4-2=2)

Optimal sub-structure Problem? Yes.

We are trying to find the maximum profit we can make given the constraints.

1. Identify sub-problems:

The goal is to find the max profit with at most:

- **k** transactions

Which is *similar* to finding max profit for...

- **k-1** transactions *// Sub-problem*
- **k-2** transactions *// Sub-problem of sub-problem*
- etc.

In other words,

We want to get the profits for:

0,1, 2, 3, ..., k transaction(s)

2. W…oblem based on cached results:

- …ion do I make at every step?

…rithm chooses to **buy** or **sell** or **do nothing**

- Currently at step k,
  - If we did **no transaction** at step k on day i:
    - Profit is the same profit as on transaction k on day i-1
  - If we **sell a stock** at step k:
    - The profit for selling on day i depends on the day, j, we bought the stock (get the maximum range by choosing the day where price is lowest).
    - And in addition to the profit we made day j with one less transaction

- Mathematically represent the problem:
  - `Profit(k,i)`: The maximum profit with up to day i
  - `Profit(k,i-1)`: The maximum profit w ons up to the day before
  - `Profit(k-1,j)`: The maximum profit if ess transactions up to day j
  - `prices[i] – prices[j]`: If we sold on day i, we bought on a day j (i>j) such that it gives the maximum profit.


```
Profit(k,i) = max(profit(k,i-1), profit(k-1,j) + prices[i]-prices[j])
```


Choose the **maximum** from:

- Getting the profit from the day before

- Getting the profit on the day we bought the stock at one less transaction +

the profit made by the current transaction of selling that stock on current day i.

DP: Example Solution


```
if prices.length == 0 || k == 0,                    with edge case
        return 0
int[][] profits = new int[k+1][n]                   lation Table (transaction size: [0,k], n days)
for i=0 to i=k+1, profits[i][0] = 0  // Initialize 0 profits on 0th day
for i=0 to i=n, profits[0][i] = 0    // Initialize 0 profits on 0th transaction

for t=1 to k+1                          // loop through sub problems k, k-1, … 1
  for i=1 to n                          //loop through every day for t transactions
    profits[t][i] = profits[t][i-1] // Initialize the profit (if no transaction)
    for j=1 to i                        // Find maximum profit from day 0 upto day i

        profits[t][i] = max(profits[t][i], prices[i] + profits[t-1][j] + prices[i] – prices[j])

return profits[k][n-1]                  // Max profit (bottom right of memoization table)
```

K = 2

prices = [2,4,1]

nitialization

		Days		
		0	0,1	0,1,2
transaction	0	0	0	0
	1	0	2	2
	2	0	2	**2**

Profits[1][1]

Max of:

- Profits[1][0] = 0
- Profits[0][0] + Prices[1] – Prices[0] = 0+2 = **2**

Profits[1][2]

Max of:

- Profits[1][1] = **2**
- Profits[0][0] + Prices[2] – Prices[0] = 1-2 = -1

Profits[2][1]

Max of:

- Profits[2][0] = 2
- Profits[1][0] + Prices[1] – Prices[0] = 4-2 = **2**

Profits[2][2]

Max of:

- Profits[2][1] = **2**
- Profits[1][1] + Prices[2] – Prices[0] = 2+ (-1) = 1

Greedy Algorithm

Greedy Algorithm

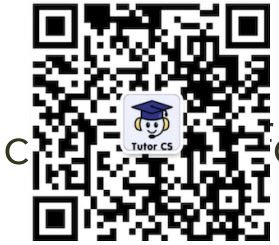
REMINDER!
Never goes back to change the choice it makes

- **What:** An algorithmic technique for optimization problems (optimal substructure)
- **Why:** Can be trivial to come up with a greedy choice
- **Why not:** Does not guarantee optimal solution for the overall problem
  - Proving the greedy choice gives optimal solution is difficult
- **Intuition:** What do we mean by "Greedy Choice" ?

  If we have a choice to make: make the **best** one currently

  If we make the **optimal choice locally**, hopefully we get a global optimal solution?

Greedy Algorithm: Steps



1. Check if you need to proc ormation before running greedy algorithm
   - Sorting?
2. Figure out what the greedy choice is (prove it is optimal locally)
3. Make that greedy choice when you have to, and now solve the sub-problem
   - Using top down approach
4. Show that with the greedy choice made, the subproblems, we have in the end an optimal solution to the problem

Greedy Algorithm: Example

Activity Selection Problem

Data Compression

- Specimen problem
- Kruskal MST
- Dijkstra's Single source Shortest Path Algorithm

Greedy Algorithm: Huffman Encoding

- **Definitions:**

  Code: Map of a alphabet charact[illegible]ry code-word

  Pre-fix code: A binary code such that no code-word is the prefix of another code-word

  Encoding Tree (full binary tree): Represents a prefix code

    Characters: Stored in leaf

    Code word: The path from the root to leaf (0 if left child, 1 if right child)

- **Idea:** Assign a character to a binary string whose length depends on the frequency of that character

    Encode high-frequency characters with short code words

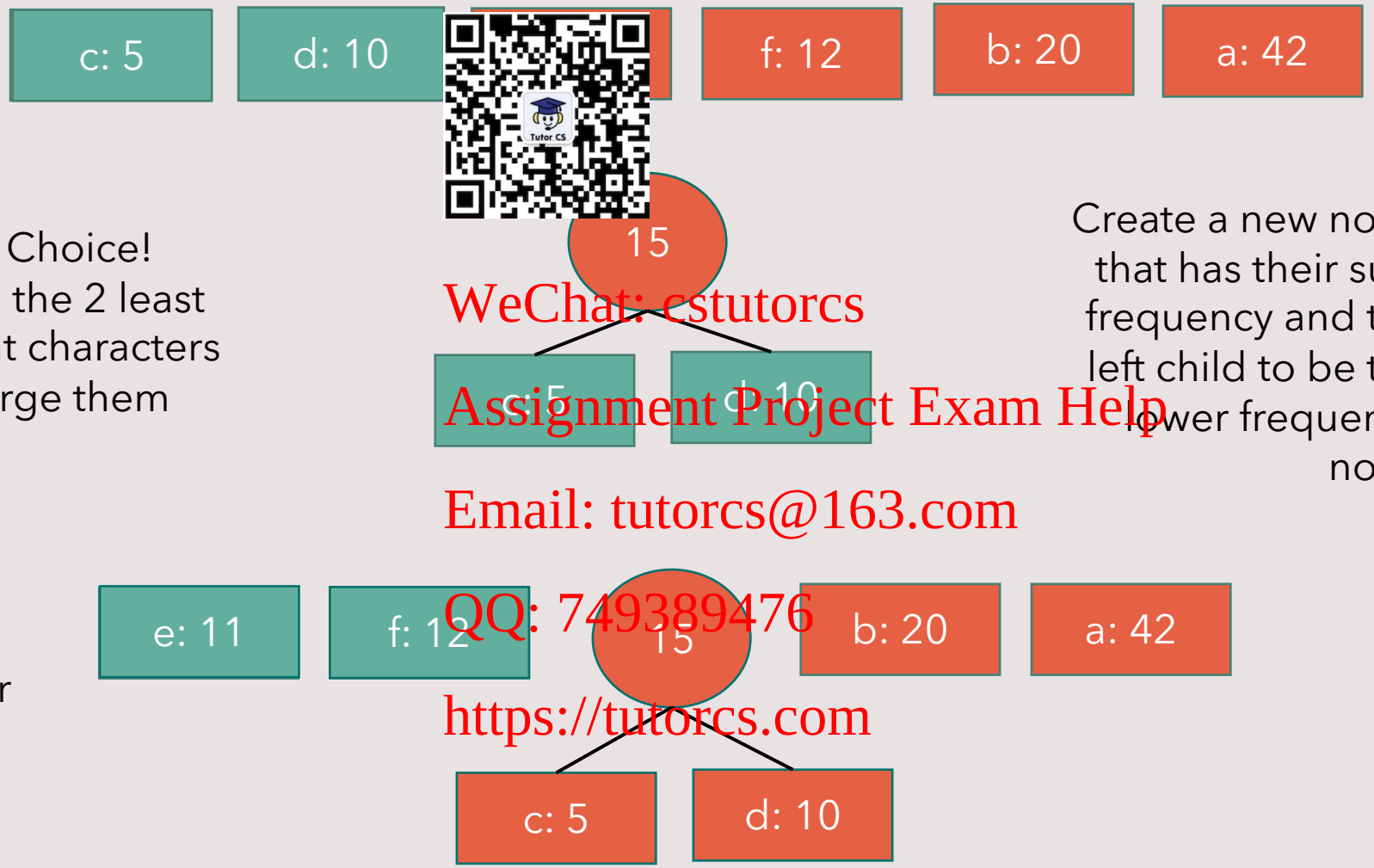
    No code word is a prefix for another code

The greedy algorithm used:

- **Greedy Choice:** Choose the two least frequent characters (then merge)
- Optimal prefix: Search in a binary tree T, and label the leaves with characters, that minimizes the number of bits used (length of the path).
- In other words: Assign a shorter codeword length to a more frequently used character
- Results: This will save us maximum space when storing this word

Total # bits used by the word = $\sum_{i=1}^{n} f[i] \cdot depth(i)$

Example

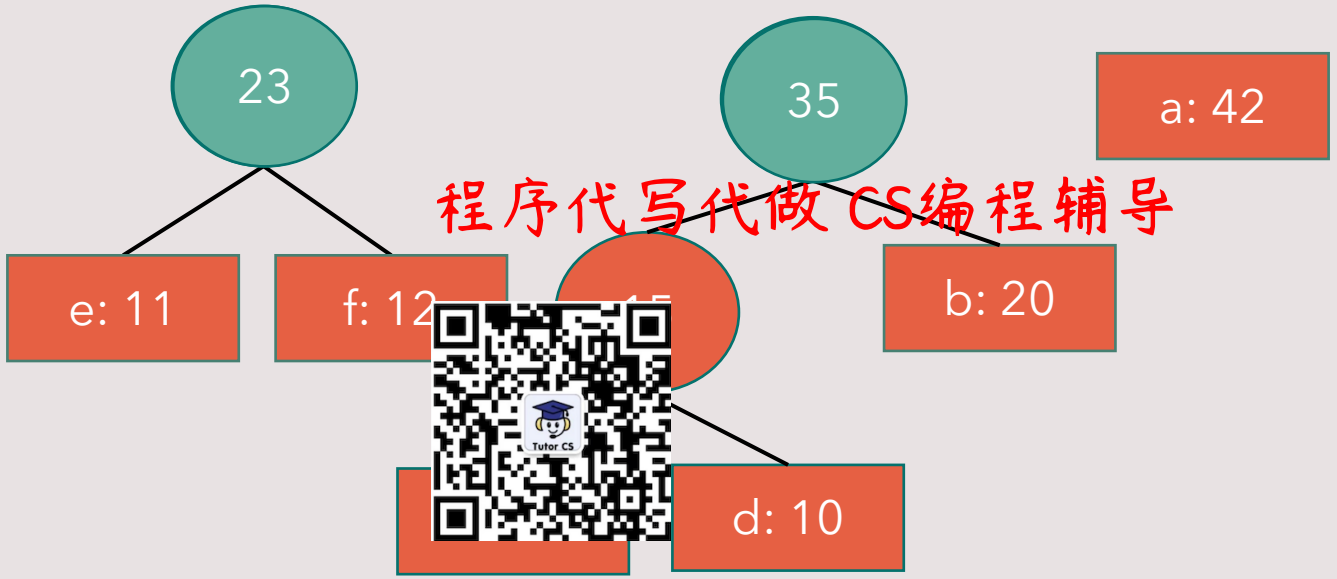


Greedy Choice! Choose the 2 least frequent characters and merge them

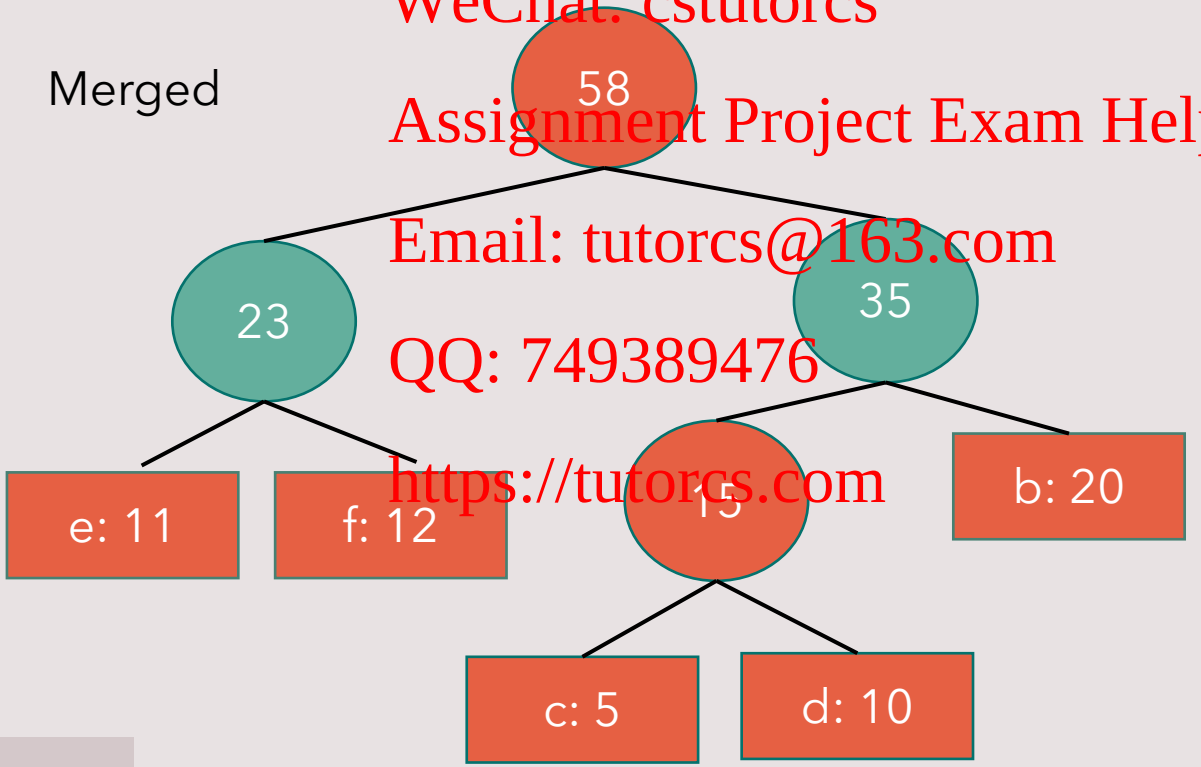
Create a new node that has their sum frequency and the left child to be the lower frequency node

Put the new node into correct order

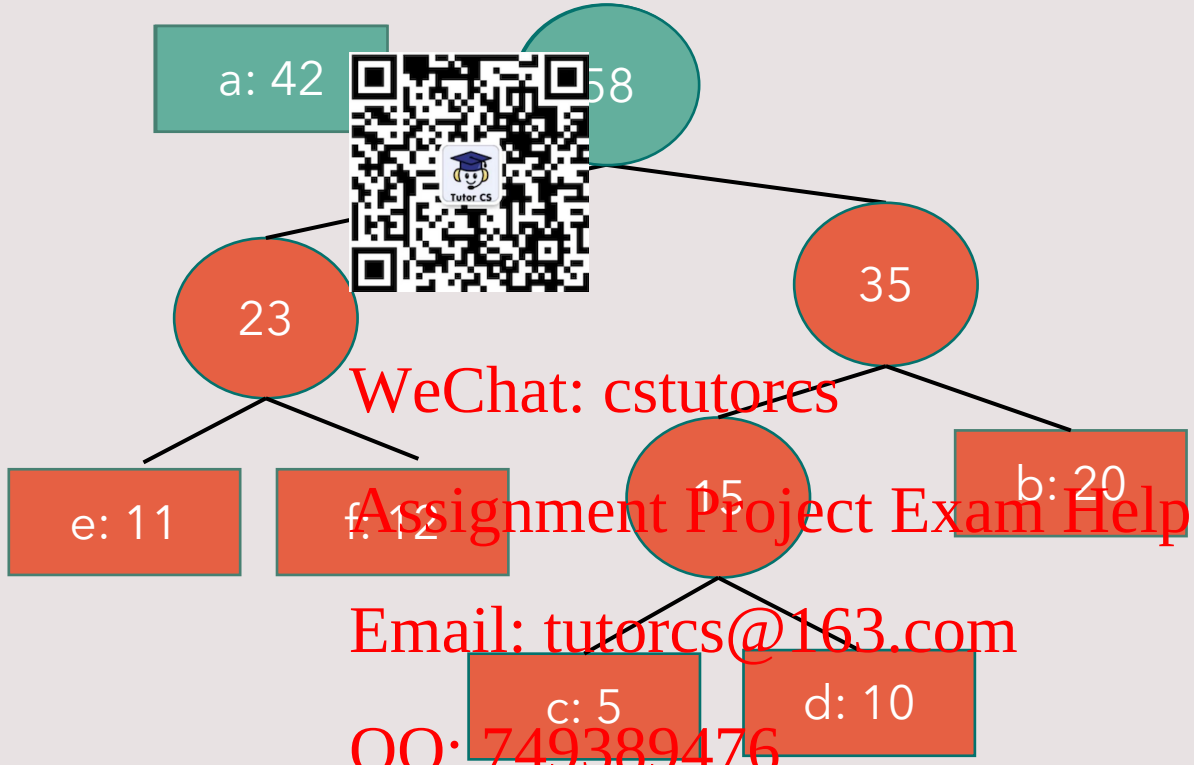
Sorted



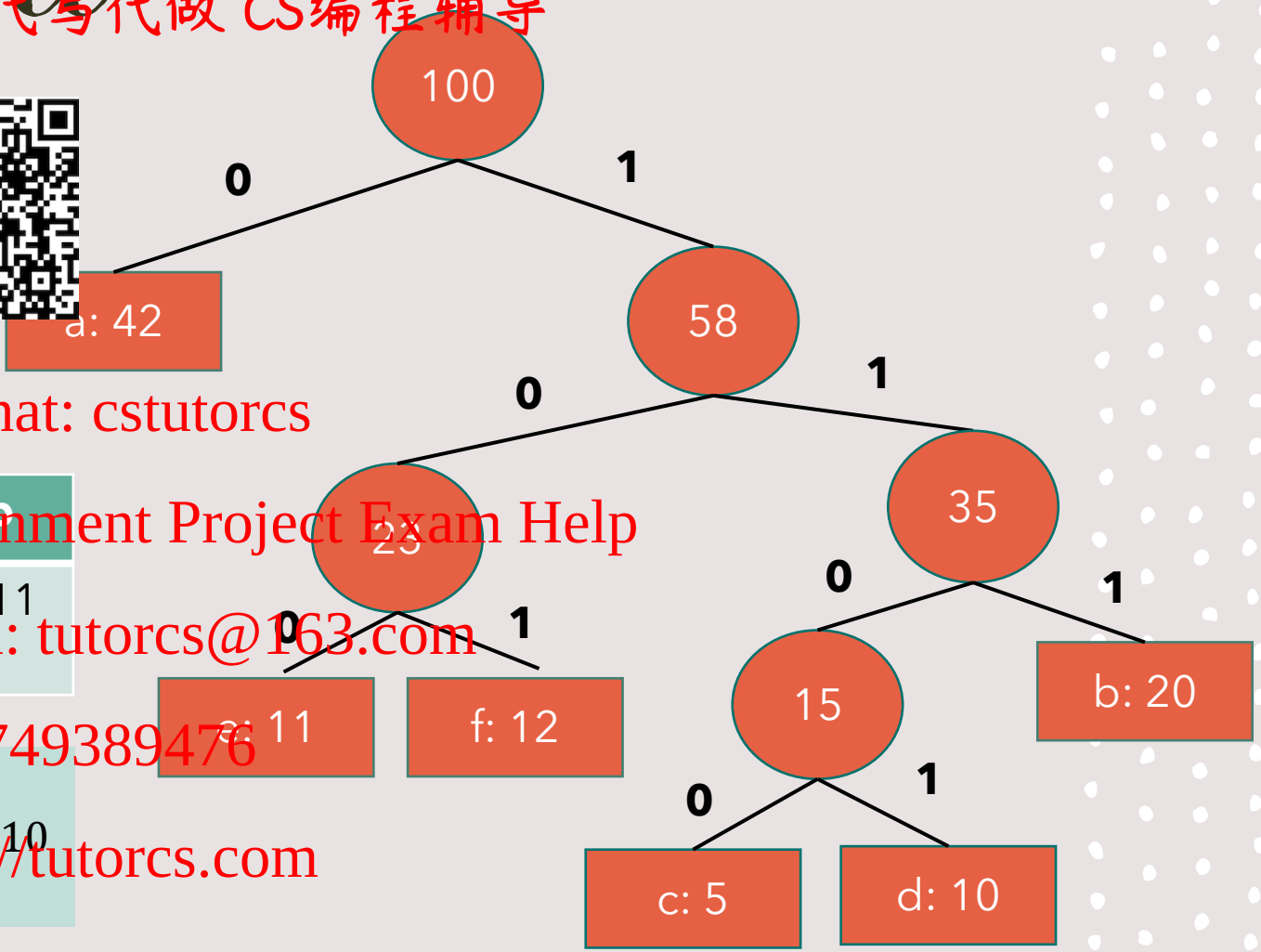
Merged



Sorted



The final Encoding Tree



char	a	e	f	c	d	b
code	0	100	101	1100	1101	111

Total # bits:

$= 1 * 42 + 3 * 11 + 3 * 12 + 4 * 5 + 3 * 20 + 4 * 10$

$= 231$

Amortized Analysis

Waldispühl.J. (2020). *COMP251*Amortized analysis [PowerPoint slides]. https://www.cs.mcgill.ca/~jeromew/teaching/251/F2020/COMP251_Lecture20_F2020.pdf

Amortised analysis?

- **Why?:** To show that on average the cost per operation is much smaller compared to considering the individual cost
- **How?:**
  1. **Aggregate analysis**
  2. **Accounting method**
  3. **Potential method (not covered here)**

Aggregate analysis

- **Step 1:** Show that all n opera $T(n)$ time
- **Step 2:** Show that each operation ,thus, takes $\frac{T(n)}{n}$ that is **constant time.**
- **NOTE:** Different operation can have **different costs**

Accounting method

- **Step 1:** Assign charges to op some might have higher charge other might have lower change than actual cost. Amortised cost = amount we charge.
- **Step 2:** When Amortised cost > Actual cost, store the difference as **credit**
- **Step 3:** Use **credit** later when Amortised cost < Actual cost
- **NOTE:** Different operation have **same cost**

Example problem: modified stack

- A stack with following opera

1. ***pop*()**: $pop()$ has time complexity of $O(1)$
2. ***push*()**: $push()$ has time complexity of $O(1)$
3. $multipop(n)$ **:** calls pop n-times or stop if less than n elements present in stack

Question: Run time of multipop(k)?

Naive

- $multipop(n)$ has time complexity of $O(n)$
- Thus , $n\ multipop(n)$ will take $O(n^2)$

Aggregate method

- Each object is popped only once for each time it is pushed to the stack
- Then, if number of pushes $\leq n$ then number of pops $\leq n$ that is we can only run one $\boldsymbol{multipop(k)}$ for k $\leq n$
- Therefore, total cost is $\boldsymbol{O(n)}$
- Amortized cost $= \frac{\boldsymbol{O(n)\ cost\ each}}{\boldsymbol{n\ operations}} = \boldsymbol{O(1)}$

Accounting method

- Costs are as follows:
  - Push = \$2
  - Pop = \$0
  - Multipop = \$0
- Note: actual cost of multipop is variable whereas her we have set it constant.
- \$1 for Push and \$1 prepayment for any pop from either pop() or multipop()
- Amortized cost $= \boldsymbol{O(1)}$

Amortized Analysis. *Brilliant.org*. Retrieved 15:10, March 6, 2021, from https://brilliant.org/wiki/amortized-analysis/

THANK YOU!

Kahoot Solutions: Data structures 1, 2, 3,

- Question 1
- What is FALSE about Red Black Trees?
- The black height of a RBT is the black height of the root
- The Root's parent is *nil*[T]
- At most half of the nodes in a path from root to leaf is black
- In RB-Fixup, the loop terminates when the parent[x] is black
- Answer:
- **c**

  "At LEAST half "

  This is due to the RB property of no 2 consecutive nodes can be red.

- Q
- **What is the next array we have to sort if we are using Heapsort (+MaxHeapify)?**

7	4	3	1	2

- Answer:

4	2	3	1	7

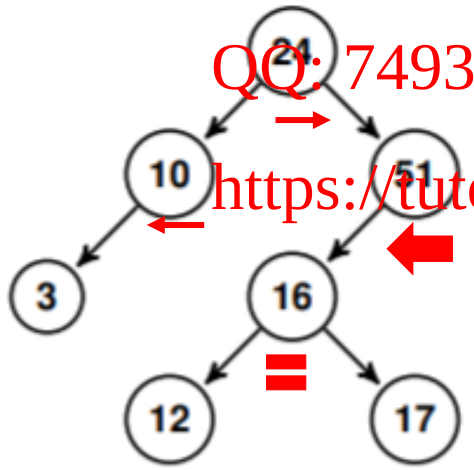
- Swap(1,5), maxheapify(1)

- Question 3
- Performing one rotation always preserves the AVL property.

Answer: False, we can see it from the last AVL tree example we looked at today

Kahoot Solutions: 4, 5

- Question 4
- Consider the AVL tree below. Indicate one node that violates the AVL property (give the key used to label the node).
- Answer: 51 (**Note**: please check slides for the meaning of arrows)



- Question 5
- **What is TRUE about building a heap?**
- We insert a node the same way as BST (Does not follow BST properties)
- Time complexity is only based on MaxHeapify
- The time complexity is O(logn) (It is nlogn)
- Given an array, we first call MaxHeapify on index 1 (We start at: A.length/2)

Answer: **Time complexity is only based on MaxHeapify**

#. Calls of MaxHeapify * MaxHeapify

Kahoot Solutions: Design and Analysis 1, 2

For Question 2 explanation please visit: http://www.cim.mcgill.ca/~langer/251/5-disjointsets.pdf

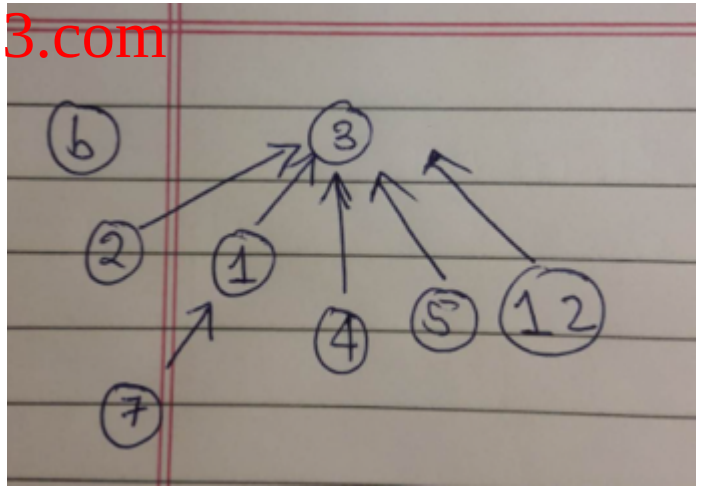
- Question 1
- All dynamic programming algorithms satisfy an optimal substructure property.
- Answer: True

Question 2:

Give the parent map that results from the following sequence for union by size with path compression

union(1,2), union(3,4), union(3,5), union(1,7), union(3,12), union(7,4), union(2,3)

Answer:



Kahoot Solutions: 3,4

- Question 3
- Which case applies to the following:

  T(n) = sin(n) *T(n/2) + sin(n)
- Answer: Does not apply

  Since f(n) = sin(n) which is not a polynomial.

  NOTE: It is fine if f(n) = 2^n

- Question 4
- Is the statement true or false:

  The sum over all operations, of the amortized cost per operation gives a lower bound on the actual cost of the sequence of operation.
- Answer: False, it gives an upper bound