Questions

1.

a. Dijkstra’s algorithm for the shortest path and Prim’s minimum spanning tree algorithm have the same big-Oh worst case running time. is this true?

Dijkstra's algorithm has a time complexity of O((V + E) log V), where V is the number of vertices and E is the number of edges. The main factor contributing to the time complexity is using a priority queue to extract the minimum distance vertex in each iteration efficiently.

On the other hand, Prim's algorithm has a time complexity of O(E + V log V) with a binary heap implementation, where V is the number of vertices and E is the number of edges. The main operations driving the time complexity are the decreased key operations in the priority queue.

Although both algorithms involve priority queues and have a logarithmic factor due to the priority queue operations, the specific operations and their implementations lead to different overall complexities. Therefore, the statement that Dijkstra's and Prim's algorithms have the same big-Oh worst-case running time is inaccurate.

2.

Q7: **Can you do *Iterative Pre-order Traversal* of a *Binary Tree* without *Recursion*?**

Mid Binary Tree

**Problem**

Given a binary tree, return the preorder traversal of its nodes' values. Can you do it without recursion?

**Answer**

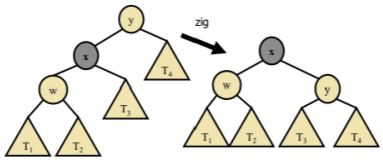
Let's use **Stack**:

1. Set current node as `root`
2. Check if current node is `null` then return
3. Store current node value in the container
4. Put `right` on stack
5. Put `left` on stack
6. While stack is *not empty*
   i. `pop` from stack into current node. Note, the `left` will be on top of stack, and will be taken first
   ii. Repeat from step 2

3. Topics in Trees
   BST, AVL, splay, (2,4), red-black:
   insert/delete
   Heapify (min- and max-heap)
   Splay (zig-zig and zig-zag)

## Cost per zig

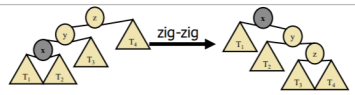
Rank of key k the number of keys that are less than k. In other words, it returns the index of k in a sorted array, or an ordered tree.



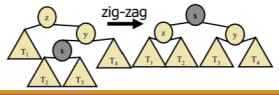

Doing a zig at x costs at most rank' (x) - rank(x):

- cost = rank' (x) + rank' (y) - rank(y) - rank(x) ≤ rank' (x) - rank(x).

## Cost per zig-zig and zig-zag




Doing a zig-zig or zig-zag at x costs at most 3(rank' (x) - rank(x)) - 2




45

## Cost of Splaying

Cost of splaying a node x at depth d of a tree rooted at r:

- at most 3(rank(r) - rank(x)) - d + 2:
- Proof: Splaying x takes d/2 splaying substeps:

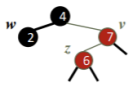
$$\begin{aligned}\text{cost} &\le \sum_{i=1}^{d/2} \text{cost}_i \\ &\le \sum_{i=1}^{d/2} (3(\text{rank}_i(x) - \text{rank}_{i-1}(x)) - 2) + 2 \\ &= 3(\text{rank}(r) - \text{rank}_0(x)) - 2(d/d) + 2 \\ &\le 3(\text{rank}(r) - \text{rank}(x)) - d + 2.\end{aligned}$$

Red-Black (remedy double reds, restucturing)

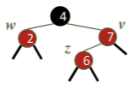
# Remedying a Double Red

Consider a double red with child $z$ and parent $v$, and let $w$ be the sibling of $v$

Case 1: $w$ is black (Aunt is Black)




Case 2: $w$ is red (Aunt is Red)

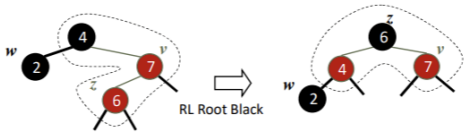



68

# Restructuring

A restructuring remedies a child-parent double red when the parent red node has a black sibling. (Aunt is Black)

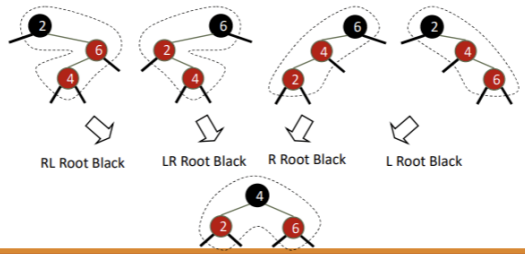
The internal property is restored and the other properties are preserved




69

# Restructuring (cont.)

There are four restructuring configurations depending on whether the double red nodes are left or right children




70

4. **How would you check if a binary tree is a subtree of another binary tree?**

   To check if a binary tree is a subtree of another, we can use the Depth-First Search (DFS) algorithm. First, traverse through the larger tree T using DFS. For each node in T that matches the root of the smaller tree S, call a separate method to validate if S is indeed a subtree starting from that point. This validation method will simultaneously traverse both trees and check for structural and value equality at each node. If all nodes match, return true; otherwise, false. The time complexity of this approach is O(m*n), where m and n are the number of nodes in the two trees respectively.

```
function isSubtree(largerTree, smallerTree):
    if largerTree is null:
        return false

    if isIdentical(largerTree, smallerTree):
        return true

    return isSubtree(largerTree.left, smallerTree) OR
isSubtree(largerTree.right, smallerTree)

function isIdentical(tree1, tree2):
    if tree1 is null AND tree2 is null:
        return true
    if tree1 is null OR tree2 is null:
        return false

    return (tree1.value == tree2.value) AND isIdentical(tree1.left,
tree2.left) AND isIdentical(tree1.right, tree2.right)
```

5. **Can you explain the concept of a Binary Heap and how it differs from a Binary Search Tree?**

   A Binary Heap is a complete binary tree, which maintains the heap property. It's either Min-Heap or Max-Heap where parent nodes are respectively smaller or larger than their children. This ensures efficient retrieval of minimum or maximum elements.

   In contrast, a Binary Search Tree (BST) doesn't necessarily maintain completeness and has an ordering property where each node's left child is less than the node, and right child is greater. This facilitates efficient search, insertion, and deletion operations.

   The key difference lies in their properties and usage. A Binary Heap is used when we need quick access to the smallest or largest element, like in Priority Queues. BSTs are preferred for searching as they provide ordered sequence of elements

6. **How would you approach finding the second largest element in a Binary Search Tree?**

To find the second largest element in a Binary Search Tree (BST), we can use an in-order traversal method. This approach visits nodes in ascending order, so the second last node visited will be the second largest. We start from root and move to the rightmost node which is the largest. Then, we step back to its parent if it has no left child. If it does have a left child, we then find the rightmost node of this left subtree. The key here is that the second largest node must either be the parent of the largest node or the largest node in the left subtree of the largest node.

7. **Consider the following infix expression and convert into reverse polish notation using stack. A + (B * C- (D/E A F)* H)**

$A + (B*C - (D/E \wedge F)*H)$

| Character | Stack | Postfix |
|---|---|---|
| A | ( | A |
| + | (+ | A |
| ( | ( + ( | A |
| B | ( + ( | AB |
| * | ( + (* | AB |
| C | ( + (* | ABC |
| – | ( + (– ( | ABC* |
| ( | ( + ( – ( | ABC* |
| D | ( + (– ( | ABC*D |
| / | ( + (– ( / | ABC*D |
| E | ( + (– (/ | ABC*DE |
| ^ | ( + (– (/^ | ABC*DE |
| F | ( + (– (/^ | ABC*DEF |
| ) | ( + (– (/^ | ABC*DEF |
| * | ( + (– * | ABC*DEF ^/ |
| H | ( + (– * | ABC*DEF ^/ H |

8. **Write the procedures for insertion, deletion and traversal of a queue.**

*1. Insertion:* Insert in Q (Queue, Max, Front, Rear, Element)

Let Queue is an array, Max is the maximum index of array, Front and Rear to hold the index of first and last element of Queue respectively and Element is value to be inserted.

**Step1:** If Front = 1 and Rear = Max or if Front = Rear + 1

Display "Overflow" and Return

**Step2:** If Front = NULL [Queue is empty]

Set Front = 1 and Rear = 1

else

if Rear = N, then
Set Rear = 1
else Set Rear = Rear +1
[End of if Structure]

**Step 3:** Set Queue [Rear) = Element [This is new element
**Step4:** End

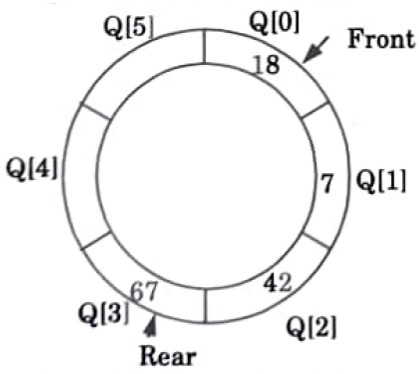
*2. Deletion:* Delete from Q (Queue, Max, Front, Rear, Item)
**Step1:**If Front = NULL [Queue is empty]
display Underflow" and Return
**Step 2:** Set Item = Queue [Front]
**Step 3**: If Front = Rear [Only one element}
Set Front = Rear and Rear = NULL
Else if Front = N, then
Set Front =1
Else Set Front = Front+ 11
[End if structure]
**Step 4:** End

*3. Traversal of a queue:* Here queue has Front End FE and Rear End RE. This algorithm traverses queue applying an operation PROCESS to each element of the queue:

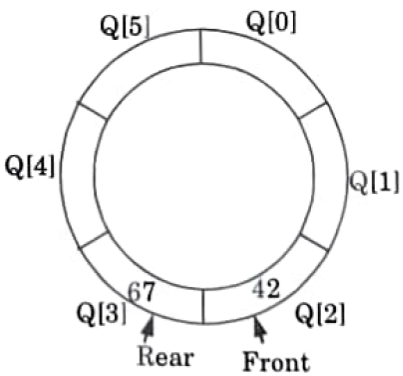
**Step 1:** [Initialize counter] Set K=FE
**Step 2:** Repeat step 3 and 4 while K≤RE
**Step 3:** [Visit element] Apply PROCESS to queue [K]
**Step 4:** [Increase counter] Set K= K+1
[End of step 2 loop]
**Step5:** Exit

**9. What is a circular queue? Write a C code to insert an element in a circular queue. Write all the condition for overflow.**

1. A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location at the queue is full.
2. In circular queue, the elements Q[0], Q[1], Q[2]…Q[n-1] is represented in a circular fashion.
For example: Suppose Q is a queue array of six elements.
3. PUSH and POP operation can be performed on circular queue. Fig. will illustrate the same.




(a) A circular queue after inserting 18, 7 , 42, 67.




(b) A circular queue after popping 18, 7.



```
public class CircularQueue {
    private int front, rear;
    private int maxSize;
    private int[] queue;
    public CircularQueue(int size) {
        maxSize = size + 1; // One extra space to differentiate between full
and empty conditions
        queue = new int[maxSize];
        front = rear = 0;
    }
    // Method to check if the queue is empty
    public boolean isEmpty() {
        return front == rear;
    }
    // Method to check if the queue is full
    public boolean isFull() {
        return (rear + 1) % maxSize == front;
    }
    // Method to insert an element into the circular queue
    public void enqueue(int item) {
        if (isFull()) {
            System.out.println("Queue is full. Cannot insert element:"+ item);
            return;
        }
        rear = (rear + 1) % maxSize;
        queue[rear] = item;
        System.out.println("Inserted element: " + item);
    }
    public static void main(String[] args) {
        CircularQueue circularQueue = new CircularQueue(5);
        circularQueue.enqueue(1);circularQueue.enqueue(2);
        circularQueue.enqueue(3);circularQueue.enqueue(4);
        circularQueue.enqueue(5);circularQueue.enqueue(6); // This will print
an error message as the queue is full
    }
```

```
}
```

Conditions for overflow: There are two conditions:

1. (front = 0) and (rear = Max – 1)
2. front = rear + 1

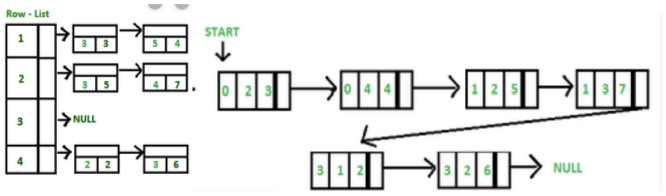
**10.**

You are supposed to design an efficient data structure based on linked lists that stores a sparse matrix. For the given sparse matrix answer the below questions:

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$

a) Draw your data structure. Please be neat and show all the field values of each node. Any unreadable solutions will not be graded.

Any below solution or any other reasonable solution is acceptable:




b) Give the space complexity of your structure and the time complexity of finding a value in a given (x,y) location and compare it with the array based matrix structure.

O(n)

**11. Explain a real-life scenario that can better be modeled with a fixed-sized array rather than a linked list. Explain why the array is better than a linked list in this case.**

Storing the values of a fixed number of things is better done with a fixed size array. For example, storing the grades of a class of students ordered in a specific way. This way, the grade of any student can be accessed or modified in constant time. The linked list would be much slower to do these.