Lecture 1

12th August 2025

What is an Algorithm?

An **algorithm** is a finite sequence of well-defined instructions to solve a given problem. It must:

1. Terminate after a finite number of steps.
2. Produce the correct output for all valid inputs.

What are Data Structures?

Data structures are methods to organize and store data in a computer so that it can be accessed and used efficiently. **Key goal:** Balance the trade-off between *storing* and *retrieving* data.

Example:

- **Linked List:** Storing a node is $O(1)$, but retrieval is $O(n)$ (must traverse nodes).
- **Array:** Retrieval by index is $O(1)$, but inserting in the middle is $O(n)$ due to shifting.

Abstract Data Type (ADT)

An **Abstract Data Type** is a conceptual model that defines:

- The operations that can be performed.
- The behavior of those operations.

It does not define implementation details or how data is stored in memory.

Example: The `List` ADT supports inserting, deleting, and accessing elements by position. A linked list is *not* an ADT — it is a concrete data structure that can be used to implement the List ADT.

Measuring Algorithm Efficiency

One way: Count the number of **basic operations** needed to solve the problem as a function of input size n.

This allows comparing algorithms and predicting which one scales better for large inputs.

Asymptotic Notations

We study algorithm growth for large n.

Big-O

$f(n) = O(g(n))$ if $\exists$ constants $c > 0$, $n_0 \geq 0$ such that:

$$f(n) \leq c \cdot g(n), \quad \forall n \geq n_0$$

Examples: $\sqrt{n} = O(n), \quad n = O(n \log n), \quad \log n = O(n).$

Big-Omega

$f(n) = \Omega(g(n))$ if $\exists$ constants $c > 0$, $n_0 \geq 0$ such that:

$$c \cdot g(n) \leq f(n), \quad \forall n \geq n_0$$

Big-Theta

$f(n) = \Theta(g(n))$ if:

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

That is, $g(n)$ tightly bounds $f(n)$ above and below.

Small-o

$f(n) = o(g(n))$ if:

$$\forall c > 0, \ \exists n_0 : f(n) < c \cdot g(n), \ \forall n \geq n_0$$

Standard Complexity Classes

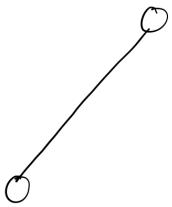
$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(2^n) \subset O(n!)$$

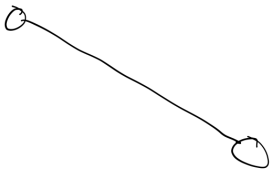
Types of Algorithm Analysis

1. **Worst Case Analysis**
2. **Average Case Analysis**
3. **Best Case Analysis**
4. **Amortized Analysis**

Homework

1. Define a Binary Tree.
2. Check whether the given diagrams are Binary Trees.





Lecture 2

14^{th} August 2025

Refer sir's notes for this lecture. Topics that were covered in the class:

- Queues operations
- Algorithm to add and delete an element in simple and circular queue.(both array and linked list implementation)
- Stack operations

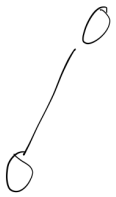
Lecture 3

19th August 2025

Binary Tree

A binary tree is either empty or it has:

- A root node.
- Two children (left subtree & right subtree).
- Every child is itself a binary tree.



Above is a binary tree but right subtree of the root node is empty

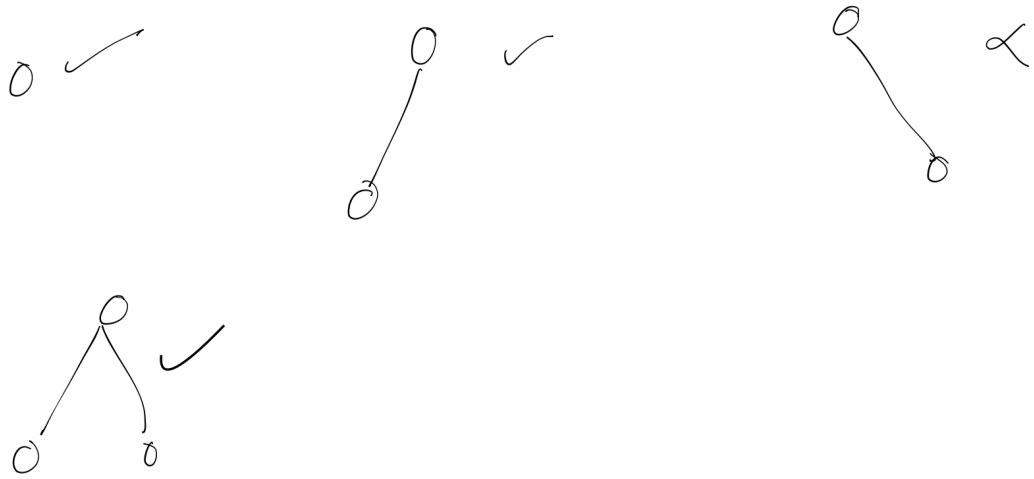
- (The right subtree of a root node can be empty.)
- Maximum number of nodes at level i is at most 2^i.
- Height h implies maximum nodes $2^{h+1} - 1$.
- n nodes correspond to $\lceil \log_2(n+1) \rceil$ levels.

Questions:

- How many binary trees can you make with n nodes?
  Types: Full Binary Tree, Complete Binary Tree.
- Complete Binary Tree:
  All levels completely filled except possibly the last, which is filled from left to right.
- How many complete binary trees can you construct with n nodes?
  Answer: 1

Question

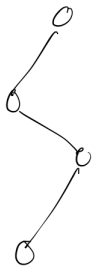
Which of the following is a **complete binary tree**?



Benefits of Complete Binary Trees

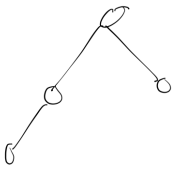
- They can be stored efficiently.
- Example: Storing a sparse binary tree in an array waste memory, but complete binary trees require minimal memory spaces.
- Complete binary trees also reduce worst-case height (the height of the leaf farthest from the root node).

Example



If we want to store this tree in an array, we would have to leave some memory blank (we need at least 6 spaces).

But in a **Complete Binary Tree**:



We would just need 4 spaces.

Perfect Binary Tree

All nodes have two children and all leaf nodes are at the same level.

Node Structure

- `data`
- `left` pointer
- `right` pointer

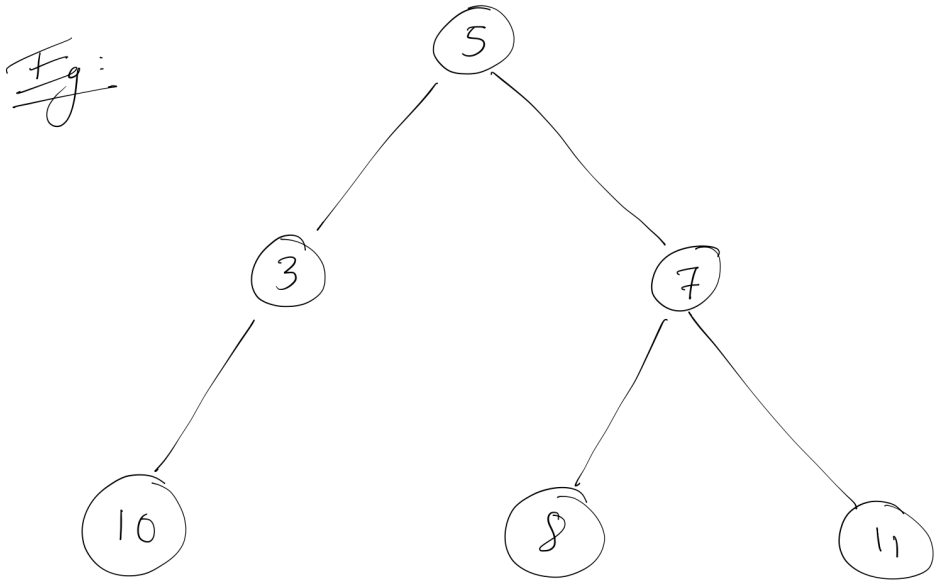
(Conventional order: left subtree before right subtree)

Tree Traversals

- Pre-order: root → left → right
- In-order: left → root → right
- Post-order: left → right → root

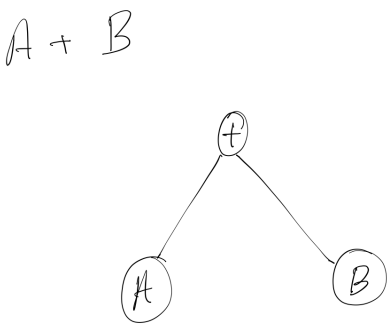
Preorder Traversal Function

```
Function Preorder(node)
  if node == NULL
    return
  Print(node.data)
  Preorder(node.left)
  Preorder(node.right)
```



- Preorder: $5 \to 3 \to 10 \to 7 \to 8 \to 11$
- Inorder: $10 \to 3 \to 5 \to 8 \to 7 \to 11$
- Postorder: $10 \to 3 \to 8 \to 11 \to 7 \to 5$

Algebraic Expression for Binary Tree



Level Order Traversal

```
Function LevelOrder(root)
  if root == NULL
    create an empty queue Q
    return
  Add(Q, root)
  while Q is not empty
    node = Delete(Q)
    Print(node.data)
    if node.left != NULL
      Add(Q, node.left)
    if node.right != NULL
      Add(Q, node.right)
```

Binary Search Tree (BST)

A BST is a special binary tree where:

- Each node contains a key.
- For every node, the keys in the Left Subtree are less than the node's key.
- The keys in the Right Subtree are greater than the node's key.

BST Operations:

- Insertion
- Deletion
- Search
- Traversal

Insertion in BST

```
Insert(Node root, key x)
  if root == NULL
    root = new Node(x)
    return root
  if x < root.key
    root.left = Insert(root.left, x)
  else if x > root.key
    root.right = Insert(root.right, x)
  return root
```

(Note: This implementation ignores duplicates.)

Deletion in BST

```
Delete(Node root, key x)
  if root == NULL
    return root
  if x < root.key
    root.left = Delete(root.left, x)
  else if x > root.key
    root.right = Delete(root.right, x)
  else
    # Node to be deleted found:
    # Case 1: Node with no children (leaf)
    # Case 2: Node with one child
    # Case 3: Node with two children (replace with inorder successor)
```

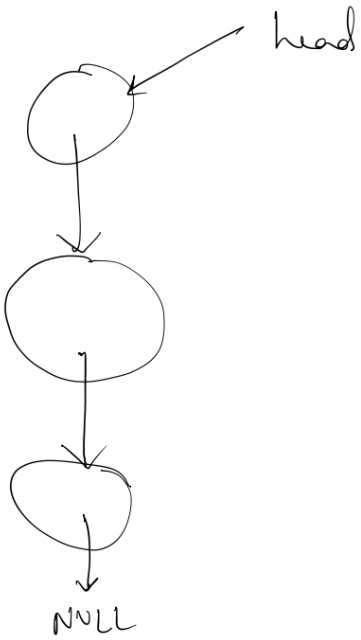
Lecture 4

24th August 2025

Linked List: Motivation

Arrays have certain shortcomings, such as fixed size and inefficient memory allocation. A **Linked List** overcomes these by dynamically allocating memory and allowing efficient insertion/deletion.

- Each node in a linked list contains two parts: data and a pointer to the next node.
- The first node is called the **head**. If the list is empty, `head = NULL`.



Linked List as an Abstract Data Type (ADT)

A **Linked List ADT** defines:

- The operations that can be performed (create, insert, delete, search, traverse, check empty, find length).
- The behavior of these operations, not the implementation details.

Basic Operations and their Algorithms

Create

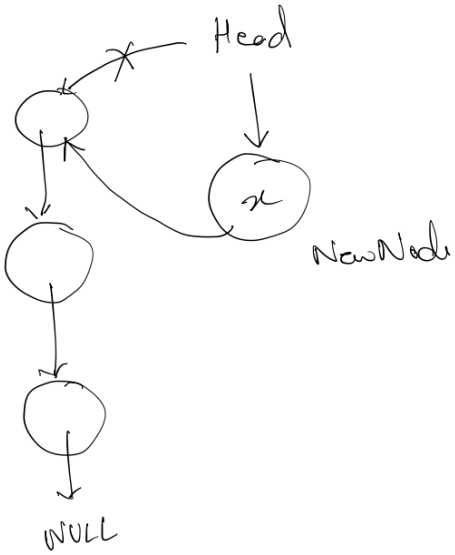
`CREATE()`
Initializes an empty list.

```
head = NIL
return head
```

Insert at Beginning

`INSERTBEGIN(head, x)`
Inserts node with value x at the start.

```
newNode = allocate node
newNode.data = x
newNode.next = head
head = newNode
return head
```



Complexity: $O(1)$

Insert at End

`INSERTEND(head, x)`
Inserts node with value x at the end.

```
newNode = allocate node
newNode.data = x
newNode.next = NIL
if head == NIL
    head = newNode
    return head
temp = head
while temp.next != NIL
    temp = temp.next
```

```
temp.next = newNode
return head
```

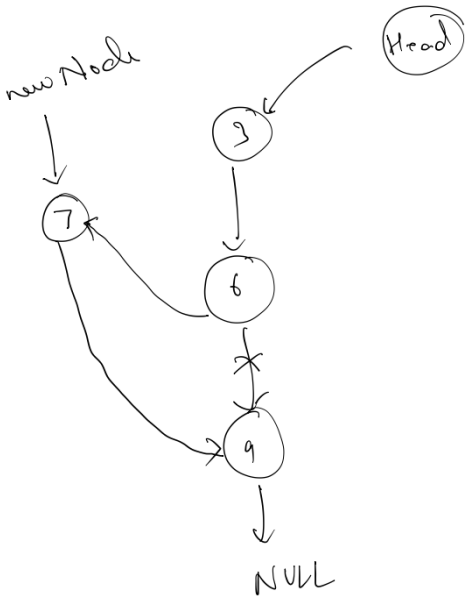
Complexity: $O(n)$

Insert After a Node

`INSERTAFTER(node, x)`
Inserts x after a given node.

```
if node == NIL
    print "Invalid position"
    return
newNode = allocate node
newNode.data = x
newNode.next = node.next
node.next = newNode
```

Example 1. *We want to insert* 7 *in the list* $\{3, 6, 9\}$ *preserving the order, using a linked list.*

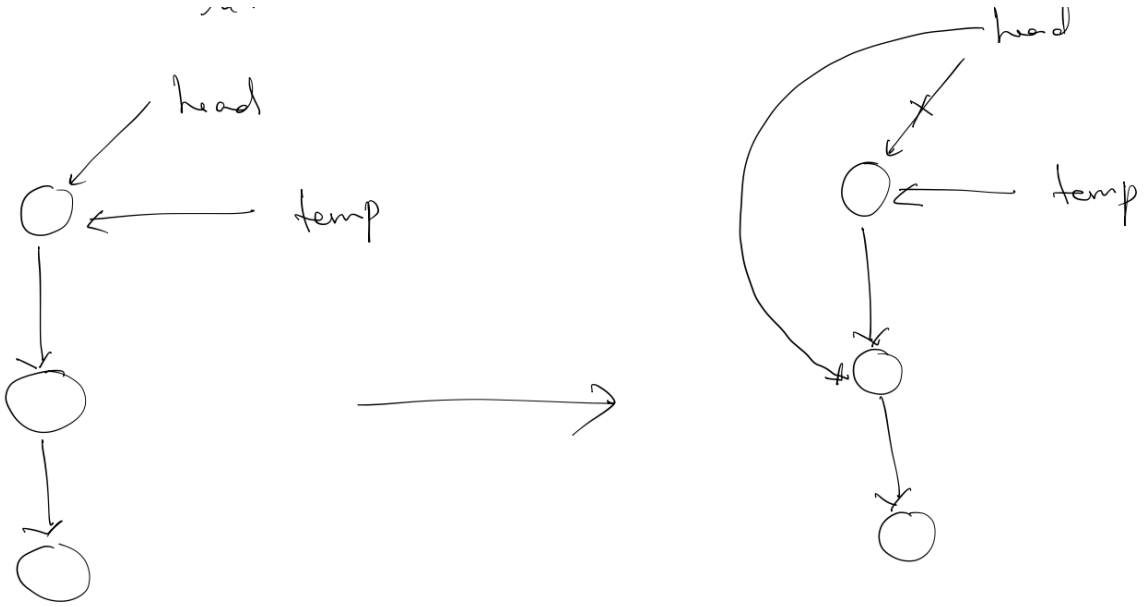


Delete from Beginning

`DELETEBEGIN(head)`
Deletes node at the start.

```
if head == NIL
    print "Underflow"
    return NIL
temp = head
```

```
head = head.next
free(temp)
return head
```



Complexity: $O(1)$

Search for Value

`SEARCH(head, x)`
Searches for value x in the list.

```
pos = 1
temp = head
while temp != NIL
    if temp.data == x
        return pos
    temp = temp.next
    pos = pos + 1
return -1
```

Complexity: $O(n)$

Traverse and Length

- **Traverse**: Walk through each node (implement as exercise).
- **Length**: Count the number of nodes (implement as exercise).

Limitations of Singly Linked List

- Cannot move backward; only forward traversal is possible.
- Doubly linked lists allow bidirectional movement.

Doubly Linked List (Concept)

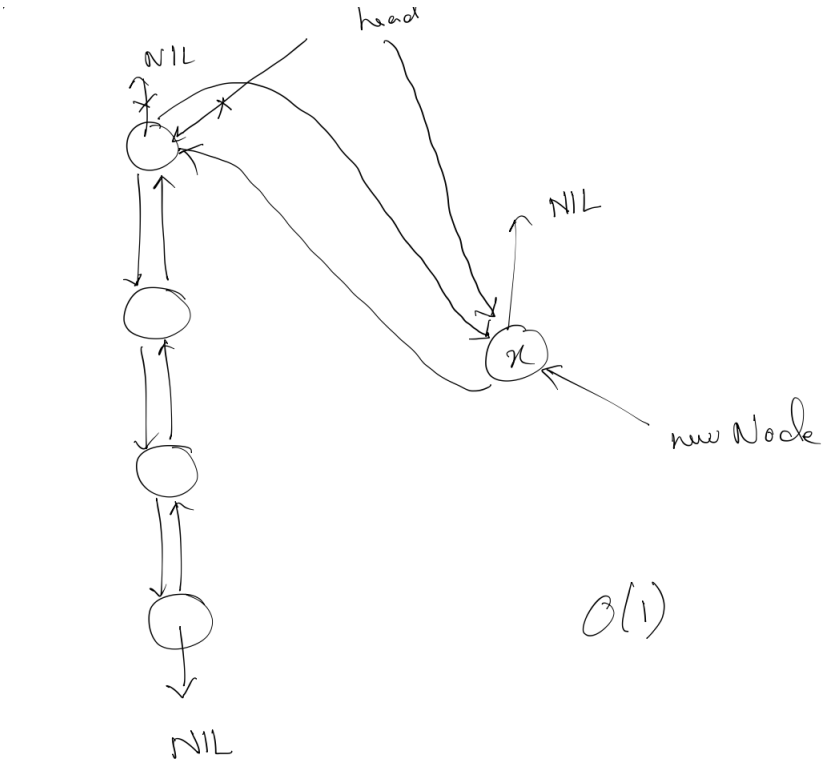
A **Doubly Linked List** node contains three parts: data, pointer to next node, pointer to previous node.

- Insertions and deletions at both ends can be done efficiently.

Insert at Beginning (Doubly Linked List):

InsertBegin(head, x):

```
newNode = allocate node
newNode.data = x
newNode.prev = NIL
newNode.next = head
if head != NIL
    head.prev = newNode
head = newNode
return head
```



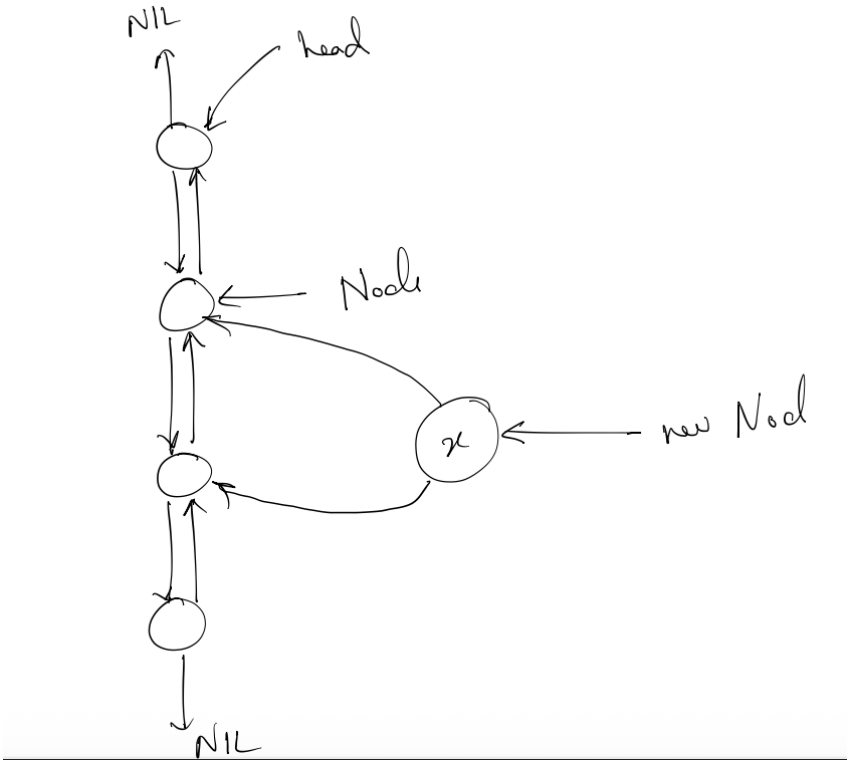
Inserting After a given node

InsertAfter(node, X)

```
if node == NIL
    print "Invalid pointer"
    return

newNode      <-  Node(X)
newNode.next <-  node.next
newNode.prev <-  node

if node.next != NIL
    node.next.prev <- newNode
node.next    <- newNode
```



D.S.A 26.8.25

Algorithm 1 Delete(head, X)

```
1: temp ⇐ head
2: while temp ≠ NIL and temp.data ≠ X do
3:     temp ⇐ temp.next
4: end while
5: if temp == NIL then
6:     print "Not Found"
7:     return head
8: end if
9: if temp.prev ≠ NIL then
10:    temp.prev.next ⇐ temp.next
11: else
12:    head ⇐ temp.next
13: end if
14: if temp.next ≠ NIL then
15:    temp.next.prev ⇐ temp.prev
16: end if
17: return head
```

C.S.L.L(Circular Singly Linked List)

seq. of nodes (data , next)
with last.next = head

seq. of nodes (data , next)
with last.next = head

please note:

In the C.S.L.L part,the thing written under it explain how it works , so it basically gives an idea about C.S.L.L.

Algorithm 2 Delete(head, X)

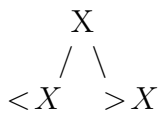
```
1: if head == NULL then
2:    return NULL
3: end if
4: curr ⇐ head ; prev ⇐ NULL
5: repeat
6:    if curr.data = X then
7:       if prev == NULL then
8:          temp ⇐ head
9:          while temp.next ≠ head do
10:            temp ⇐ temp.next
11:         end while
12:         if head = head.next then
13:            head ⇐ NULL
14:         else
15:            temp.next ⇐ head.next
16:            head ⇐ head.next
17:         end if
18:      else
19:         prev.next ⇐ curr.next
20:      end if
21:      return head
22:   end if
23:   prev ⇐ curr
24:   curr ⇐ curr.next
25: until curr ≠ head
26: return head
```

Algorithm 3 Insert-Beginning(head, X)

```
1: newNode ⇐ Node(X)
2: if head = NULL then
3:    newNode.next ⇐ newNode
4:    head ⇐ newNode
5:    return head
6: end if
7: temp ⇐ head
8: while temp.next ≠ head do
9:    temp ⇐ temp.next
10: end while
11: newNode.next ⇐ head
12: temp.next ⇐ newNode
13: head ⇐ newNode
14: return head
```

BST (Binary Search Tree)

Example of a Binary Search Tree



- In BST, the difference between the left & right child can be at most 1.
- What is the minimum no. of nodes if the height given is h?

$$N_H = \text{min. no. of nodes in a height-}h \text{ balanced tree.}$$

$$N_H = N_{H-1} + N_{H-2} + 1 \; > \; 2N_{H-2}$$

$$\Rightarrow \; N_H > 2^{H/2}$$

$$\Rightarrow \; H < 2 \log N_H$$

$$N_H = F_{H+2} - 1$$

Please Note:

The $<$ and $>$ signs given in the picture of "example of a binary tree" means less and greater than , in the usual sense.

Lecture 6

28th August 2025

Recap

If the balancing property of a binary tree is satisfied, the height is $O(\log n)$ (where n is number of nodes).

AVL Insert

1. Insert in a simple BST.
2. Work your way up restoring AVL property and updating heights as you go.

Balancing

Balance factor = height of left subtree - height of right subtree

A tree is balanced iff balance factor of all node is -1, 0 or 1.

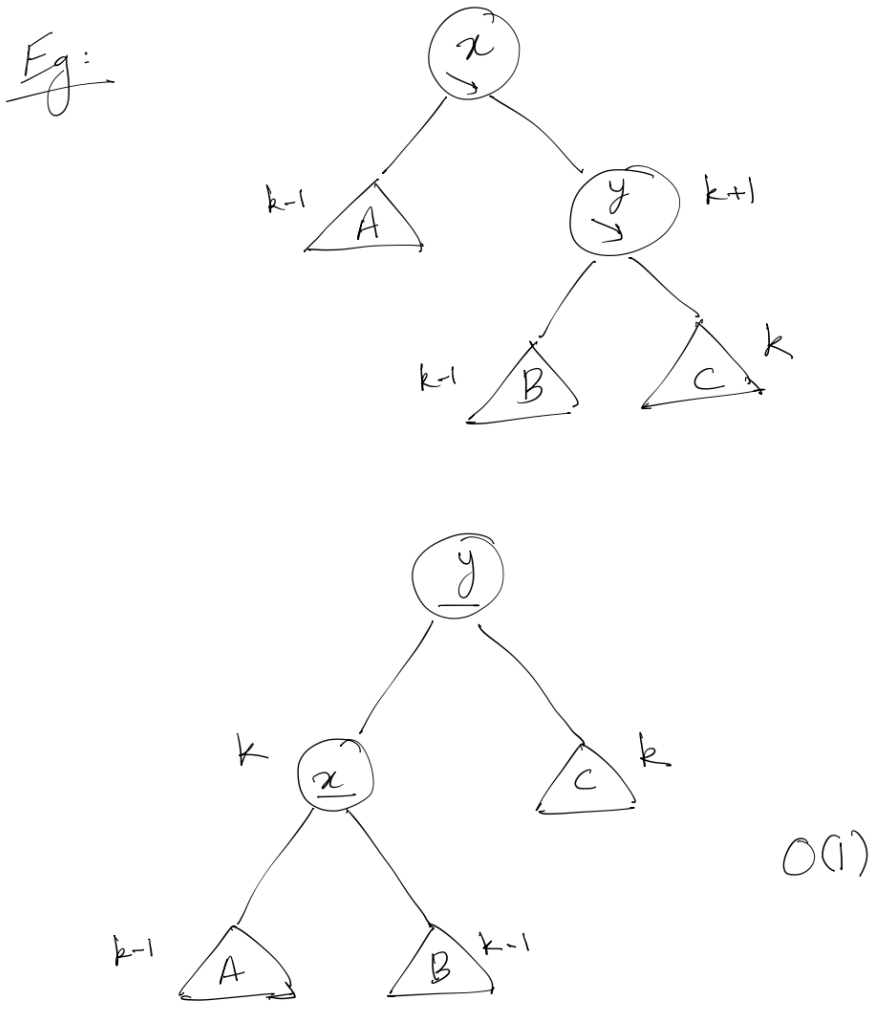
Rotations

The balancing procedure involves:

- Left rotation
- Right rotation
- Right-left rotation
- Left-right rotation

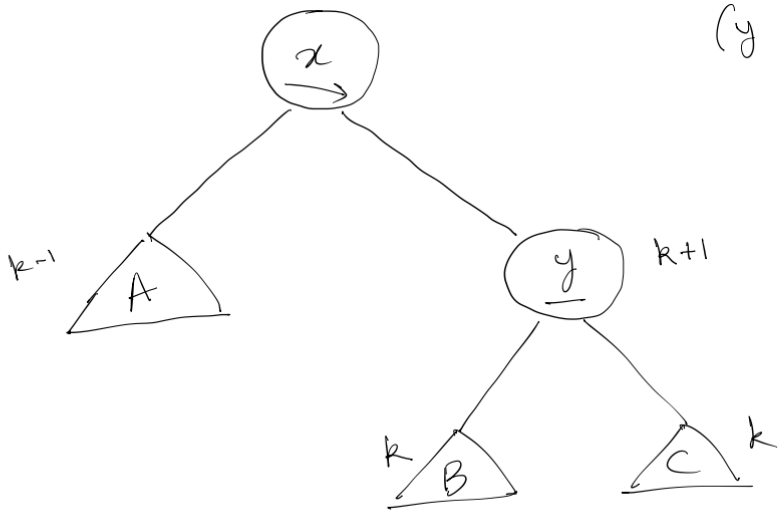
The above operations take $O(1)$ time.

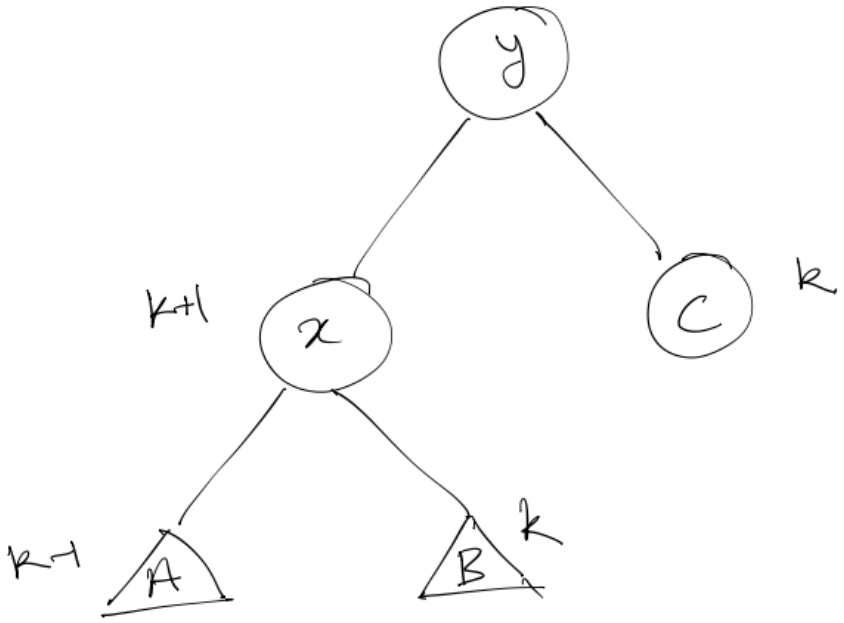
Now, suppose x is the lowest node violating AVL property and x is right heavy.

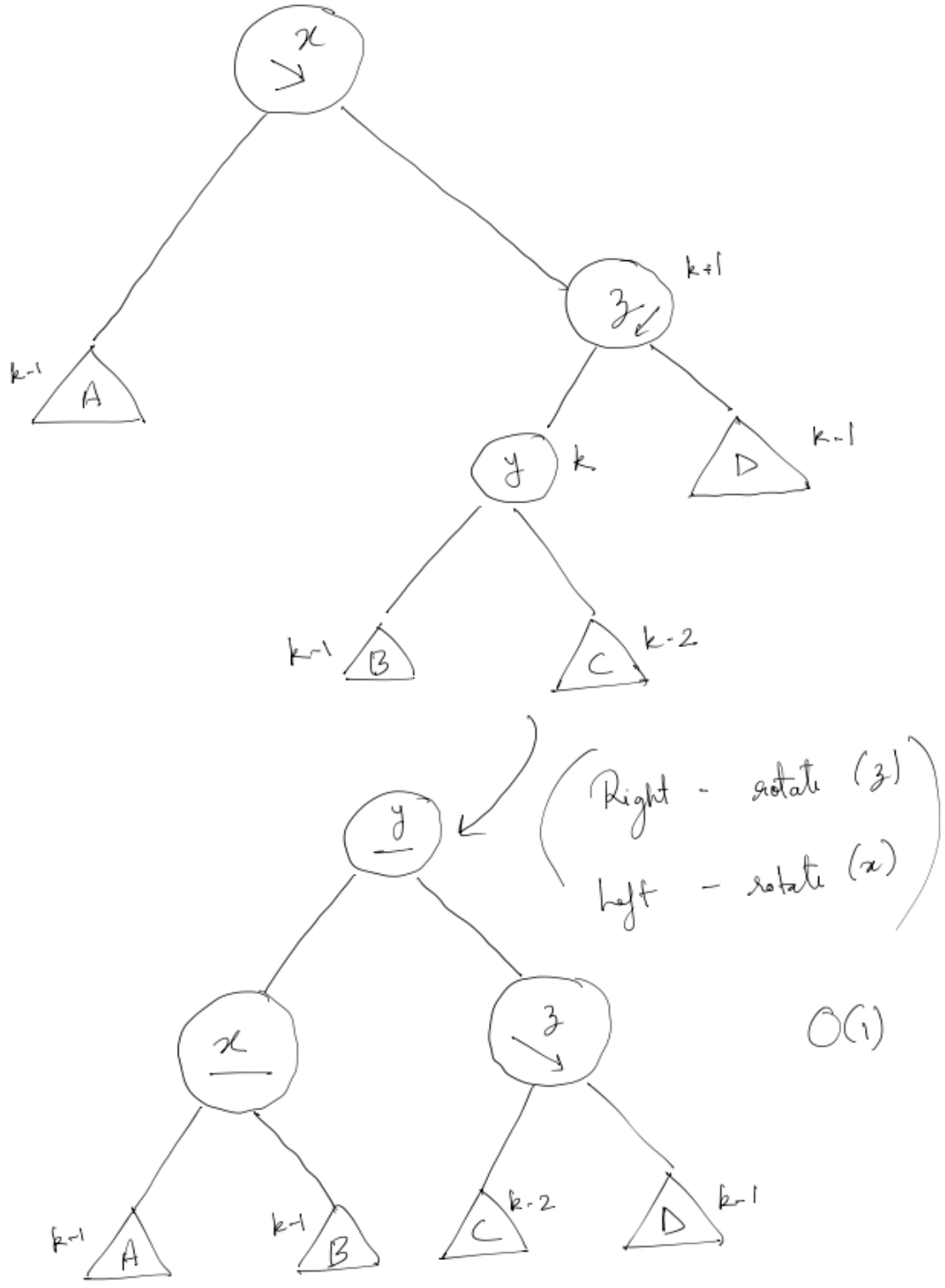


Recall AVL tree has to be BST and it should be balanced. The above procedure is called left-rotate in x.

Case II (y is Balanced)

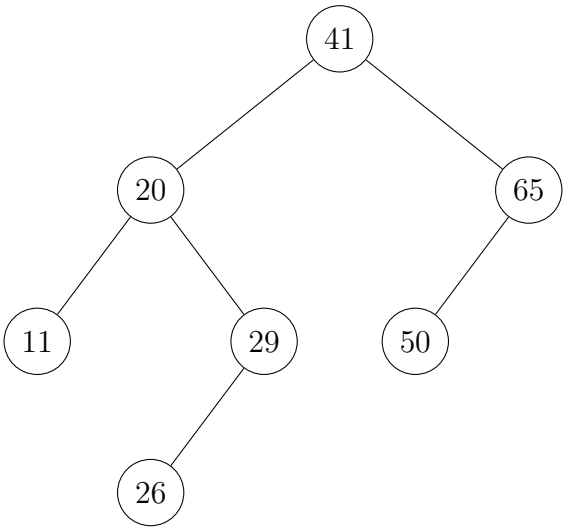




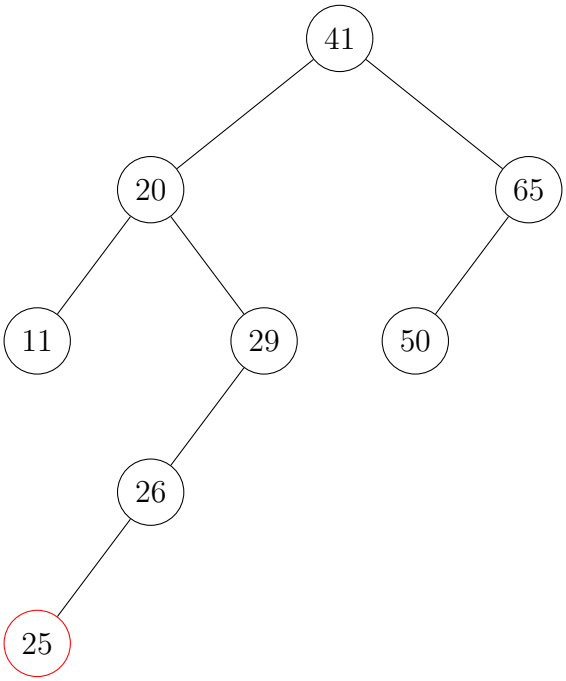


$$\text{Cost of insertion} = \underbrace{\log n}_{\text{search}} + \underbrace{2}_{\text{rotation}} + \underbrace{\log n}_{\substack{\text{going} \\ \text{back}}}$$

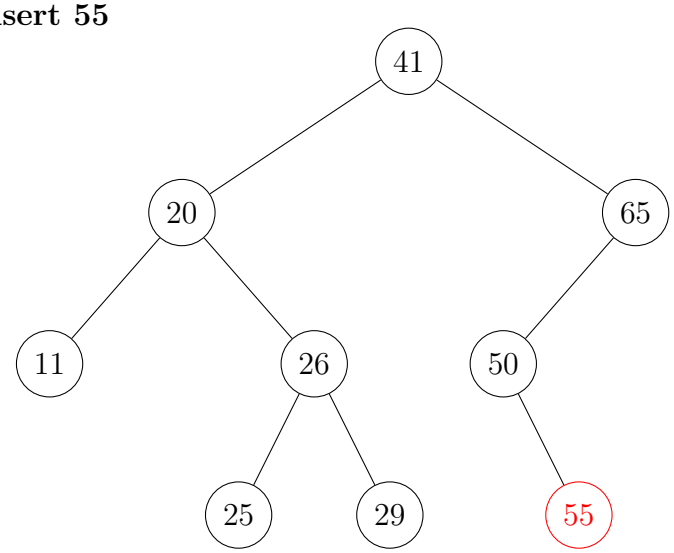
Example



Insert 25



Insert 55



Heap

- Min heap: Parent node is less than child nodes.
- Max heap: Parent node is greater than child nodes.
- Heap sort has complexity $O(n \log n)$.

Deleting a Node in BST

1. If leaf node: just delete.
2. If one child: replace the node with its child.
3. Other cases connected to AVL balancing.

dsa lec 2/9/25

AUTHOR
abhishek

Delete(note root , key x)

```
if root == NULL
  return root
if x < root.key
  root.left=Delete(root.left,x)
else if x  > root.key
  root.right = Delete(root.right,x)
else
  if root.left == NULL
    return root.right
  else root.right == NULL
    return root.left
  temp = FindMin(root,right)
  root.key = temp.key
  root.right = Delete(root.right,temp.key)
return root
```

FindMin -> this gives the inorder successor of the Root

if we think of linearly it's just replacing the the value with it successor

Hashing

Def- A Hash function maps the key to an index in the table

e.g. let say table size is 10 , what will be the hash value of this key = 12345 ? it will be **5**

A 'good' Hash function has the following properties :

- Fast to compute
- Uniform distributes keys
- Minimize collisions

1. Division : -h(k) = k -(m is prime?)

2. Multiplication

3. Mid square method

4. Folding method

Open Hashing (Chaining)

- each slot point to a linked list of entries
- Insertion : append to the list
- search : traverse list
- Performance ?Load factor : $\frac{m}{n} = \alpha$, m = keys , n = slots
- expected search cost $\approx 1 + \alpha$

Closed Hashing (Open Addressing)

- Linear Probing
- quadratic Probing ($h_i(k) = (h(k) + c_1 i + c_i i^2)(mod\ m)$)
- Double Hashing ($h_i = (h_1(k) + i h_2(k)(mod\ m)$)

---

title: “Dsa lec7 4/9” author: “Smarak” format: html editor: visual —

AVL Deletion

Steps for deleting a node in an **AVL Tree**:

1. Perform normal **BST Deletion**.
2. **Update Heights** of affected nodes.
3. **Check Balance Factor (BF)** at each node.
4. **Rebalance** using appropriate rotation(s).

Step 2: Update Heights

```
root.height = 1 + max(height(root.left), height(root.right))

height(node):
  if node == NULL:
    return 0
  return node.height
```

Step 3: Check Balance Factor

```
balance = height(root.left) - height(root.right)
```

Step 4: Rebalance with Rotations

- Four imbalance cases:

```
# LL Rotation
if balance > 1 AND Get_balance(root.left) >= 0:
  return Right_Rotate(root)

# LR Rotation
if balance > 1 AND Get_balance(root.left) < 0:
  root.left = Left_Rotate(root.left)
  return Right_Rotate(root)

# RR Rotation
if balance < -1 AND Get_balance(root.right) <= 0:
  return Left_Rotate(root)
```

```
# RL Rotation
if balance < -1 AND Get_balance(root.right) > 0:
  root.right = Right_Rotate(root.right)
  return Left_Rotate(root)
```

Complexity Analysis (Master Theorem)

General recurrence: $T(n) = aT\left(\frac{n}{b}\right) + cn^k$

Case 1: $a > b^k$

- Dominated by recursive calls.
- $T(n) = \Theta(n^{\log_b a})$

Case 2: $a = b^k$

- Balanced case.
- $T(n) = \Theta(n^k \log n)$

Case 3: $a < b^k$

- Dominated by leaf-level work.
- $T(n) = \Theta(n^k)$

Extended Master Theorem (general $f(n)$)

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & f(n) = O(n^{\log_b a - \epsilon}) \\ \Theta(n^{\log_b a} \log^{k+1} n) & f(n) = \Theta(n^{\log_b a} \log^k n) \\ \Theta(f(n)) & f(n) = \Omega(n^{\log_b a + \epsilon}), af(n/b) \leq cf(n) \end{cases}$$

Tutorial Recap

Asymptotic Notations

- $O(\leq)$: Upper bound
- $o(<)$: Strict upper bound
- $\Omega(\geq)$: Lower bound
- $\Theta(=)$: Tight bound

Data Structures Covered

- Arrays, Linked Lists
- Trees (Binary, AVL)
- Stacks (LIFO)
- Queues (FIFO)

Hashing

Open Hashing (Chaining)

- Store colliding elements in linked lists at the same hash bucket.

Linear Probing

- Probe sequence: $(h(k) + i) \mod m$, increment i by 1 each time.
- Simple but clustering occurs.

Quadratic Probing

- Probe sequence: $(h(k) + i^2) \mod m$.
- Reduces clustering compared to linear probing.

Double Hashing

- Use two hash functions: $h\_1(k)\$and\$h\_2(k)$.
- Probe sequence: $(h\_1(k) + i \cdot h\_2(k)) \mod m$.
- Effective in reducing clustering.

Heaps

- **Max Heap**: Each parent ≥ its children.
- **Min Heap**: Each parent ≤ its children.
- Supports efficient priority queue operations (insert, extract-min/max).

---

dsa 9/9

AUTHOR
smarak

Sorting

BST- all values in the left are lesser and vise versa searching , so while searching in a BST Time taken is $T(n) = T(n/2) + 1$ for once then total will be in order of $log\ (n)$ (We are asumming that the searching happnes in the array...not a linked list)

(**Stable sort**- keeps the relative order of equal elements the same as they were before sorting.)imp when non distinct values with example: if two have same vlaue their ordering will still be same after soting i.e. if X_i and X_j both have equal value and i and j and $i < j$ are the order/linear adressing of the $X's$ then after sorting $X_i \rightarrow X_a$ and $X_j \rightarrow X_b$ then $a < b$ ## Selection sort

```
for (i=0 to n-2) do
  minIndex <- i
  for j = i+1 to n-1 do
    if A[j] < A[minIndex] then
      minIndex <- j
  swap A[i] with A[minIndex]
```

-number of comparisions is n^2 -this is a inplace algorithm as space complexity is 1 -minIndex <- stores index of the minimum element in the array -This is a stable sort ?(can be acheaved...)

Bubble Sort

```
for (i=0) to (n-2) do
  Xswapped = Flase
  for (j=0) (to n-2-i) do
    if A[j] > A[j+1] then
      swap A[j] with A[j+1]
      Xswapped = True then
  Xswapped == False then
    Break
```

- space complexity is order of 1 so inplace algorhythm
- time complexity :worst case order n^2 , best case n
- Xswapped is a flag variable to check if we need to continue sorting i.e. stop if it's already sorted
- stable sort ? (Yes)

Build Tournament Tree (A)

```
leaves <- A
while N > 1 do    #N <- Number of nodes
  pair nodes,compute winer, make parrent
retrun root
```

Update Tournament Tree (leaf,T)

```
UpdateTournamentTree(leaf,T){
  while (leaf las parent) do
    parent.value <- min(leaf,sibbling)
    leaf <- parent
}
```

tournament Sort (A[0,1.....,n-1])

```
T <- BuildTournamnetTree(A)
sortedList <- []
for (i=0) to (n-1) do
  winner <- T.root.value
  append winner to SortedList
  leaf <- leaf corresponding to winner
  leaf.value <- inf     #inf <- infinity
  UpdateTournamentTree(leaf,T)
```

- building a tournamnet <- is order n
- time complexity : $n.log(n)$
- space complexity : n

Merge sort

Merge Algo

```
Merge(A,left,mid,right)
  n_1 <-mid - left + 1
  n_2 <- right-mid
  L[0,1...,n_1 - 1] <- A[left,....,mid]
  R[0,1....n_2 - 1] <- A[mid+1,....,right]
  i=0,j=0,k=left
  while (i < n_1) and (j < n_2)
    if L[i] <= R[j]
      A[k] = L[i]
      i = i+1
    else
      A[k] = R[i]
      j = j+1
```

```
    k = k+1
  while (i <n_1)
    A[k] = L[i]
    i = i+1
    k = k+1
  while (j<n_2)
    A[k] = R[j]
    j = j+1
    k = k+1
```

The sort Algo

```
MergeSort(A,left,right)

  if left < right
    mid = (left + right)/2
    MergeSort(A,left,mid)
    MergeSort(A,mid+1,right)
    Merge(A,left,mid,right)
```

- Time complexity : $2.T(n/2) + O(n)$ ($O(n)$i so from the merge step) solving gives $n.log(n)$
- space complexity : order n

DSA lec9 11.9

AUTHOR
SMARAK

#Class

Merge sort

- we split it into 2 halves
- we sort them individually and then merge them together
- Space complexity is n
- time complexity is $n.log(n)$

Quick Sort

```
Partition(A,Low,High)
  pivot <- A[high]
  i <- Low -1
  for j <- Low to High-1
    if A[j] <= pivot then
      i = i+1
      swap A[i]&A[j]
  swap A[i+1] & A[High]
  return i+1
```

1. Divide into 5 groups
2. Sort each group
3. Pick th median of each group
4. Recursively find the median of medians <- pivot
5. partition the array aroud the pivot
6. Recurse in the part containing the $ k^{th}$ element

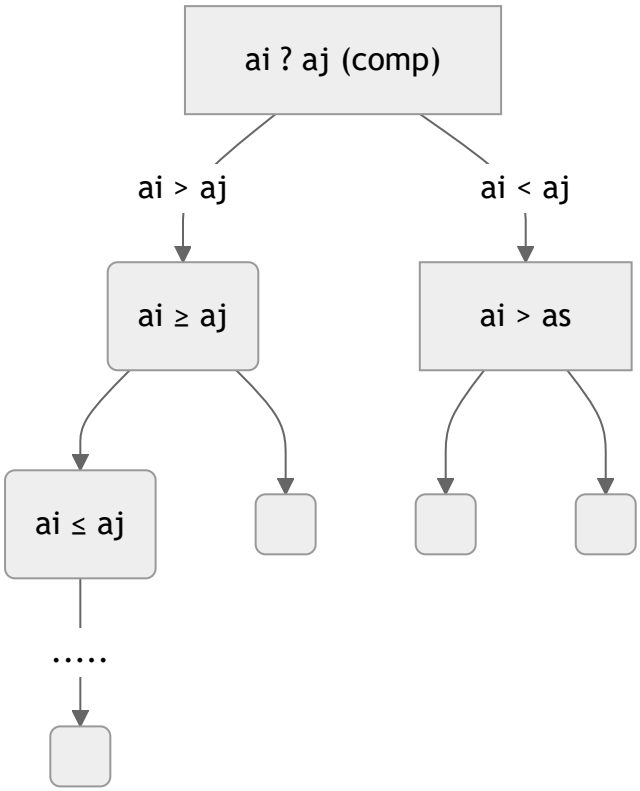
```
Select(A,k)
  if length(A) <= 5
    Sort A
    return A[k]
  groups <- Split A into groups of 5
  medians = []
  for each group in groups
    sort(group)
    Append median(group) to medians
  pivot = Select(medians,length(medians)/2)
  L = [x in A| x < pivot]
  E = [x in A| x = pivot]
  G = [x in A| x > pivot]
  if k <= |L|
    return Select(L,K)
  else if k <= |L|+|E|
    return pivot
```

```
else
  return Select(G,k-|L|-|E|)
```

1. Find the max no. to know the no. of digits d
2. Starting form the LSD use a stable sort to sort the no. acc. to the digits
3. Move to the next significant digit and repat
4. After d passes , the array is sorted

DSA lec10 16.9

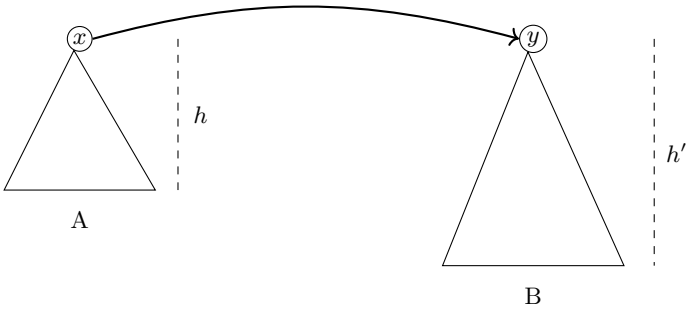
AUTHOR
Smarak



Atleast $n!$ leaves $\implies$ {Height} $\geq \log_2 n!$

$\implies O(\log_2 n)$

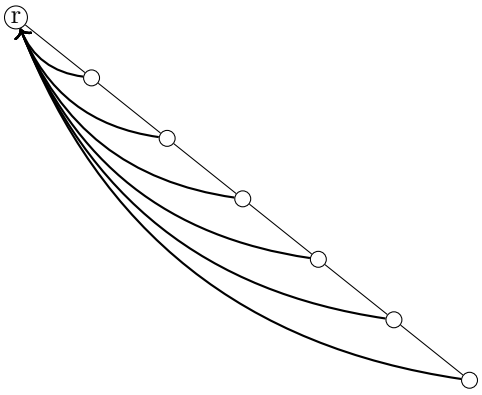
Combined Trees



$$h' > h \quad \text{or} \quad h' = h$$

Amortized Analysis

Worst case analysis for a group of operations.



$$A_0(x) = 1 + x$$

$$A_{k+1}(x) = A_k^x(x)$$

$$A_k^i = \underbrace{A_k \circ A_k \circ \cdots \circ A_k}_{i \text{ times}}$$

$$A_k^0 = \text{Identity}, \quad A_k^{i+1} = A_k \circ A_k^i$$

Compute

$$A_{k+1}(x) \quad \text{Start with } x, \quad A_k(1) = A_k(x)$$

Apply A_k x times.

Claim

If (i) x $\leq y \implies A_k(x) \leq A_k(y)$

$$(ii) x \leq A_k(x)$$

Examples

$$A_0(x) = x + 1, \quad x \geq 2$$

$$A_1(x) = A_0^x(x) = 2x$$

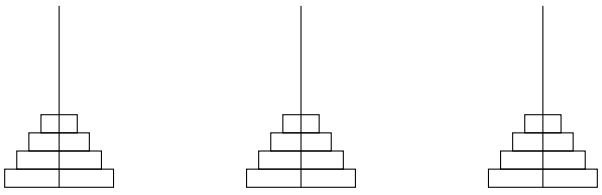
$$A_2(x) = x \cdot 2^x$$

$$A_3(x) \geq \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}}_{x \text{ times}} = 2 \uparrow x$$

$$A_4(x) \geq 2 \uparrow \big(2 \uparrow \big(\cdots \uparrow (2 \uparrow 2)\big)$$

$$\alpha(n) = \textit{the least } k \textit{ such that } A(k) \geq n$$

Tower of Hanoi



$$T_t(u) = \text{subtree rooted at } u \text{ at time } t$$

$$\text{rank}(u) = 2 + \text{height}\big(T_m(u)\big)$$

Theorem. A sequence of m union and find operations starting with n singleton sets takes time at most $\text{O}\big((\text{m+n})\,\alpha(n)\big)$.

Lecture 13

23rd September 2025

Refer sir's notes for this lecture. Topics covered - counting sort and its time complexity.