# The WRONG Structure Of Data Structures

La Structure Des Structures de Données

*COMP-2412*

2nd Edition

# Big O notation

- Describes the performance of an algorith as the amount of data increases
"how code slows as it grows"
- worst case scenario

Examples
- $O(1)$
- $O(n)$
- $O(\log n)$
- $O(n^2)$

$n$ = amount of data

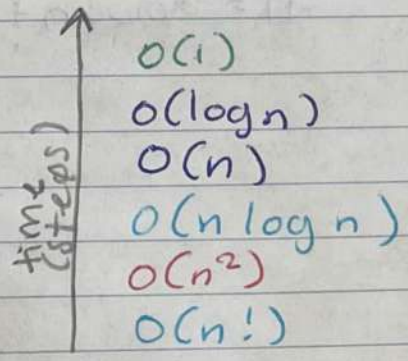

$O(1)$ constant time
- fastest an algorithm can operate
- the operation takes the same amount of time, no matter how much data ($n$) there is.

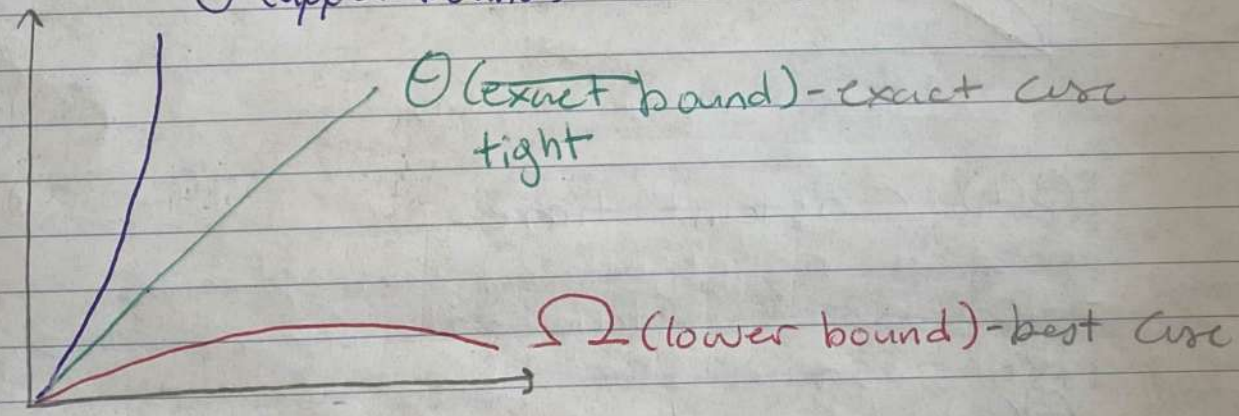
$O(n)$ linear time
- the amount of data increases the time of the operation
- the operation time and data grow at the same rate

$O(n^2)$ quadratic time
- every single item in a data set needs to interact with every item
- really slow

$O(\log n)$ logarithmic time
- cutting time of operation in half each time.

order of time




$O \rightarrow g(n) \in O(f(n))$ if and only if $f(n) \geq C \cdot g(n)$ for $n \geq n_0$

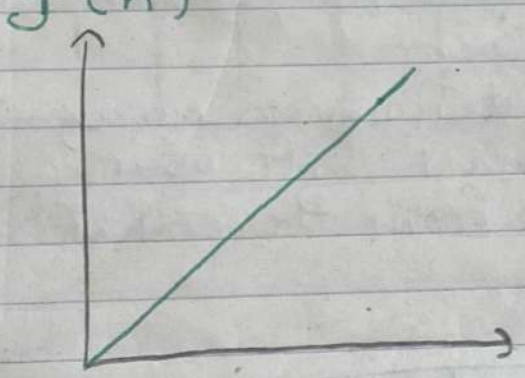
- After the input size gets big enough, $f(n)$ grows at least as fast as $g(n)$, up to a constant factor

$g(n)$ - actual running time of algorithm
$f(n)$ - Simplified upper bound to compare against $g(n)$
$C$ (constant) - fixed positive number
$n_0$ - starting point

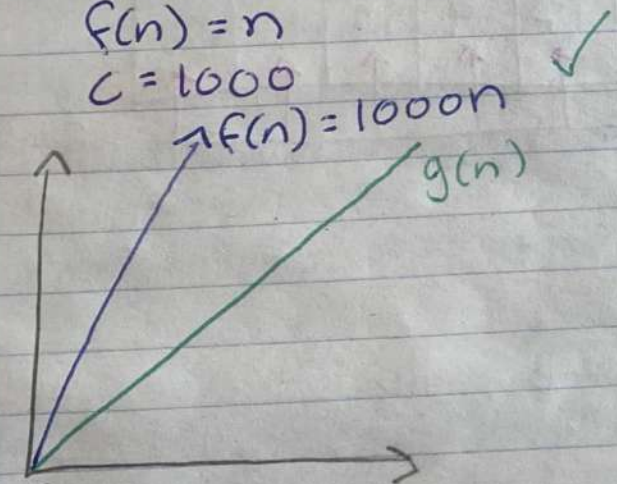
$f(n) \geq C \cdot g(n)$ for $n \geq n_0$

- Once $n$ is big enough, $f(n)$ will always stay above $g(n)$ (or grow just as fast), if we allow ourselves to stretch $g(n)$ by a constant

Ex.

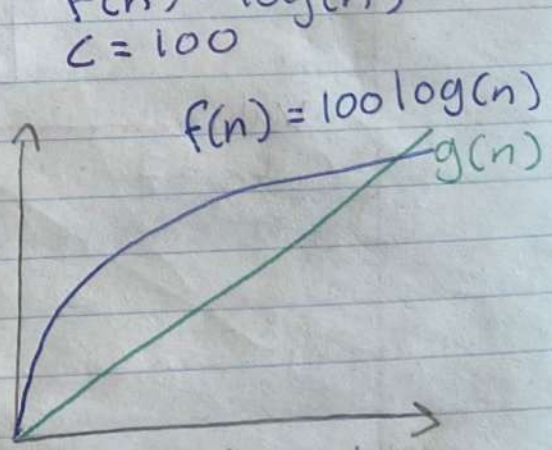



if $f(n)$ eventually dominate $g(n)$, then $g(n) \in O(f(n))$




✓

$O(n)$ is an upper bound for $g(n)$




✗

$O(\log n)$ is not an upper bound for $g(n)$

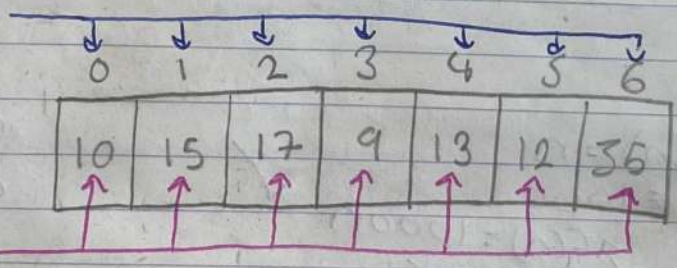
Hilroy

# Arrays

- each element is placed side by side in memory
- each element has a specific index, retrieving a value is extremly fast. you can go directly to its position without searching.
- arrays are a fixed size

- Accessing an array by an index → O(1)
- Insertion if replacing an existing element → O(1)
- Inserting while the array is full → O(n)
  - new array is required with previous elements copied over
- deleting an element from an array → O(n)
  - all other elements must shift over
- deleting an element from the end of an array → O(1)

Ex.




size = 7

# Memory

- integers can be stored on a computer as 32 bits

bit = 1 or 0

byte: small unit of data = 8 bits

Ex.

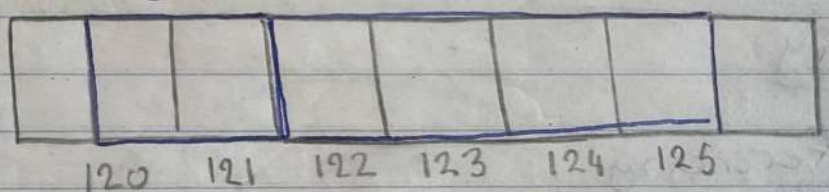
1 ⟶ 0000...01

2 ⟶ 0000...10

memory = a long tape of bytes

Ex.

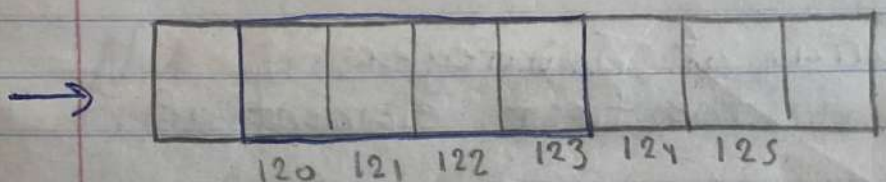
byte = 10010101 byte = 00100010




Storing addresses

- Integers take up 4 bytes (4 · 8 = 32)

Ex. 1 ⟶ 32 bits (0000...01) ⟶ bytes (4)




integer 1 takes up 4 bytes in memory

# Pointers

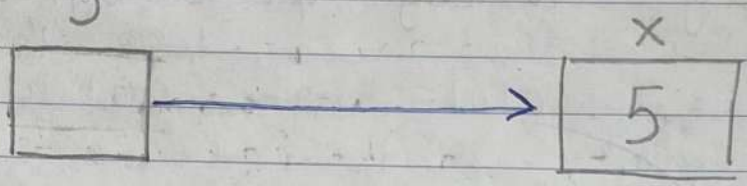
- An integer that stores a memory address

& = address of operator

Ex.

```
int x = 5;
int *y = &x;
```

y pointing towards memory address of x




x == *y

dereferencing operator

Ex.

```
int A[3];
int* Ip = &A[0];
```

- A is an array of 3 integers
- Ip points to the first element of A

```
print("%d", *Ip);
```

- Dereferencing Ip gives the value at A[0]

## Basic operation

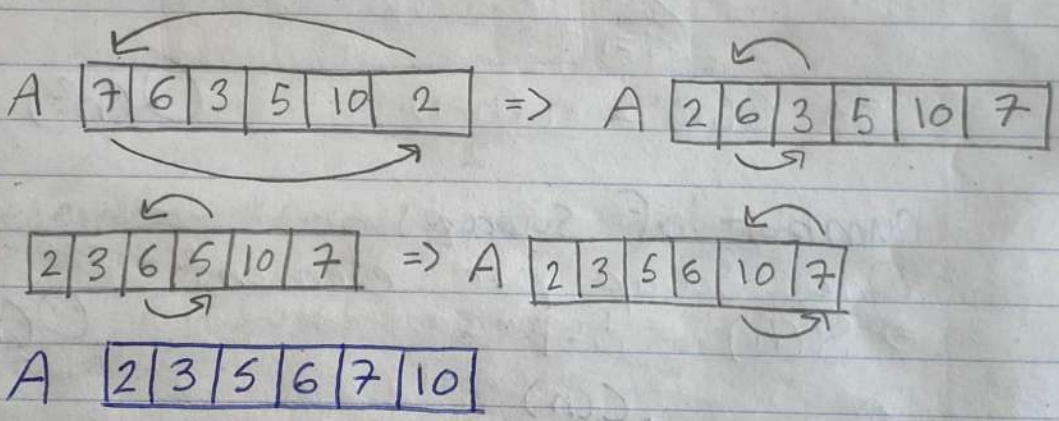
- A basic operation is the one action that dominates the running time of the algorithm
  - the operation the program performs most often.

## Sorting

### Selection sort

- Repeatedly finds the smallest element in the unsorted part of an array
- Swaps it with the first unsorted position
- Grows a sorted section from left to right

Ex.




### Not stable

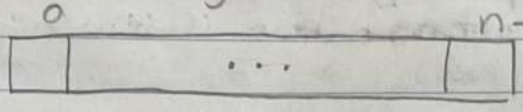
- swapping elements can change the relative order of equal elements.

Ex.

$a_1, b, a_2 \Rightarrow a_2, b, a_1$

## Number of Comparisons

- for an array that contains n elements




1st element => n-1 comparisons
2nd element => n-2
3rd element => n-3
...
n-2 element => 1

- total comparisons:

$C(n) = (n-1) + (n-2) + \cdots + 1$

$$\Rightarrow \sum_{i=1}^{n} i = 1+2+3+4+5 \cdots n$$

$$\Rightarrow C(n) = \frac{n(n-1)}{2} \qquad O(n^2)$$

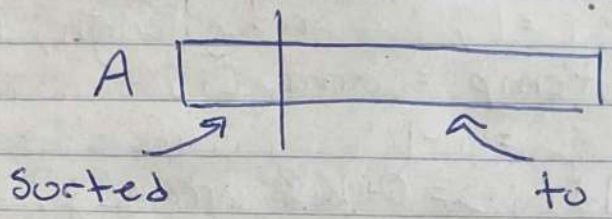
## number of swaps

- One swap per element

$\Rightarrow S(n) = n-1$ $\qquad O(n)$

# Insertion Sort

- A sorting algorithm that builds a sorted array one element at a time




- Each new element is inserted into its correct position by
  - Comparing
  - Shifting elements to the right
  - placing the element where it belongs

Ex.

A | 7 | 6 | 1 | 5 | 10 | 2 |

=> A | 7 | 6 | 1 | 5 | 10 | 2 |

First element considered sorted

= A | 6 | 7 | 1 | 5 | 10 | 2 |

=> A | 1 | 6 | 7 | 5 | 10 | 2 |

Compare

=> A | 1 | 5 | 6 | 7 | 10 | 2 |

=> A | 1 | 5 | 6 | 7 | 10 | 2 |

=> A | 1 | 2 | 5 | 6 | 7 | 10 |

Stable - keeps equal elements in their original order
- elements are shifted, not swapped.

Hilroy

## Code example

```
insertionSort(int[] array) {
    for (int i = 1; i < array.length; i++) {
        int temp = array[i]
        int j = i-1
        while (j >= 0 && array[j] > temp) {
            array[j+1] = array[j]
            j--
        }
        array[j+1] = temp
    }
}
```

## Number of comparisons

2nd element → at most 1
3rd element → at most 2
...
n element → at most n-1

$$\Rightarrow C(n) = 1+2+3+\dots+(n-1) = \frac{n(n-1)}{2}$$

- Best case (already sorted) → $O(n)$
- Average case → $O(n^2)$
- Worst case (reverse order) → $O(n^2)$

## Shifts

- Each comparison may cause a shift
  $S(n) \in O(n^2)$

The number of shifts grows on the order of $n^2$ in the worst case.

# Linear Data Structures

- A linear data structure stores elements in a sequence, where each element has a single predecessor and successor (except ends)

## Examples:

- Arrays
- Lists
- Linked Lists

## Core operations

- Insert
  - Insert(e, index) → insert element at position
  - Insert(e) → Insert at the end

- Delete
  - Delete(index) → remove element at position
  - Delete(e) → Remove by value

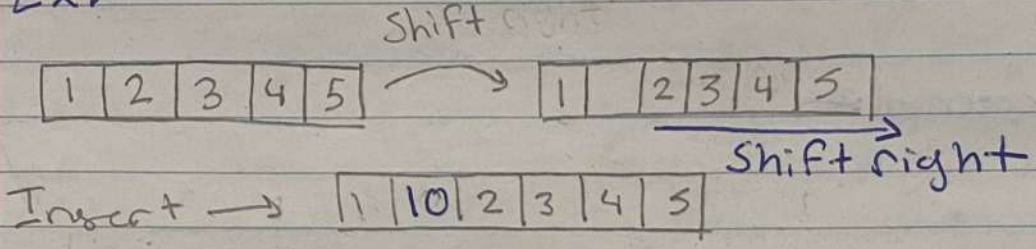
- Access and Search
  - Search(e) → find element
  - At(index) → Get element at index
  - Update(index) → Modify value at index

- State checks
  - Empty() → checks if list has no elements
  - Full() → check if list is full (array-based)

## Insert Operation (Array-Based List)

- check if list is full
- Shift elements right to make space
- Insert elements

Ex.




$\Theta(n)$ → must shift elements

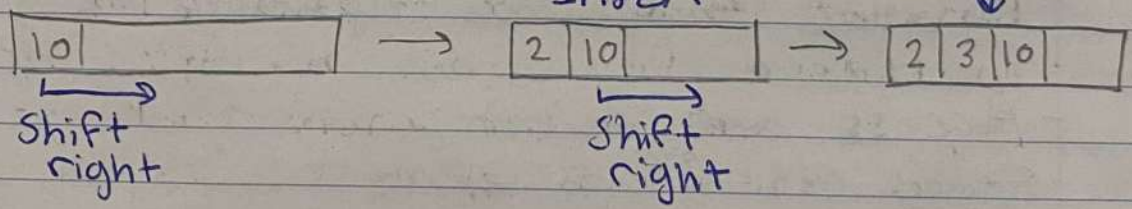
## Sorted List insertion

① Find index where element belongs
② Shift elements to the right
③ Insert element

$\Theta(n)$ or $\Theta(\log n)$

Ex.

- Find position between 2 and 10
- Insert




## Time complexity

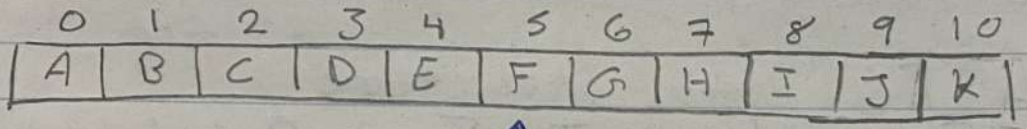
find index → $\Theta(n)$ or $\Theta(\log n)$
Shift elements → $\Theta(n)$
Insert → $\Theta(1)$
$= \Theta(n)$

# Binary Search

- Search algorithm that finds the position of a target value within a sorted array. Half of the array is eliminated during each "step".

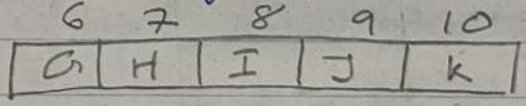
## Ex.

value = "H"
index = ?

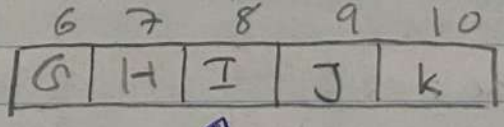



check if value is equal to middle (if yes, return index)

else

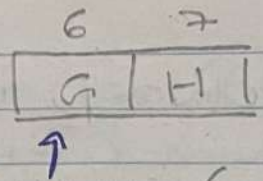



- Check to see if value is bigger or smaller than the middle value
- Disregard first half of list (since the list is sorted, we know that the value could not possibly be in the first half of the list)




check middle

Bigger than value, disregard second half of list.




check middle (round down to index 6)

- Disregard middle value, return index 7.

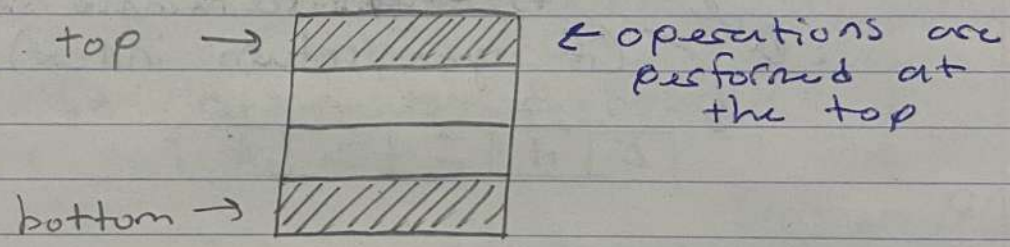
Time complexity → $\Theta(\log n)$

## Stacks

- LIFO data structure

LIFO → Last In, First Out

- Stores objects into a vertical tower




push() → adds to the top of the stack
pop() → removes and returns the top
top() → returns top of stack

## Bad Stack method implementation

```
pop()
    t = top()
    for(i = 1; i < bottom; i++){
      S[i-1] = S[i]
    }
    return t
    bottom--
```

- Removes the top element by shifting all elements one position to the left

Time Complexity → $\Theta(n)$ due to shifting

```
push(e)
    for(i = bottom; i > 0; i--){
        S[i+1] = S[i]
    }
    bottom++
    S[0] = e
```

- Inserts a new element at the bottom by shifting all elements one position to the right

Time Complexity → $\Theta(n)$ due to shifting

Effieient Stack implementation

```
pop()
    t = top()
    S[top] = null
    top--
    return t
```

- Removes the top element by making it null and decrementing the top pointer

Time Complexity → $\Theta(1)$

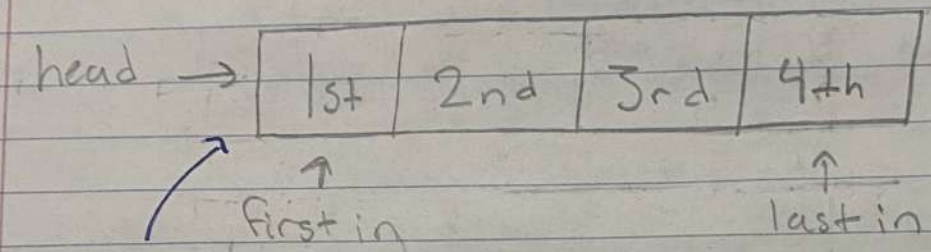
```
push(e)
    S[top+1] = e
    top++
```

- Adds a new element at the next top position

Time Complexity → $\Theta(1)$

# Queues

- FIFO Data Structure
  - FIFO: First In, First Out
  - A collection designed for holding elements prior to processing.




First in is first to be removed.

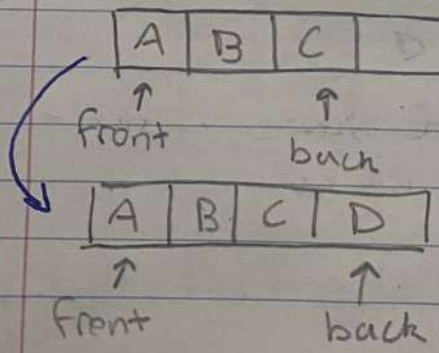
enqueue(e) → adds element e to the back of the Queue

dequeue() → removes and returns the element in the front of the Queue.

```
enqueue(e)
    if (not full)
        Q[back+1] = e
        back++
```

- adds an element to the back of the queue

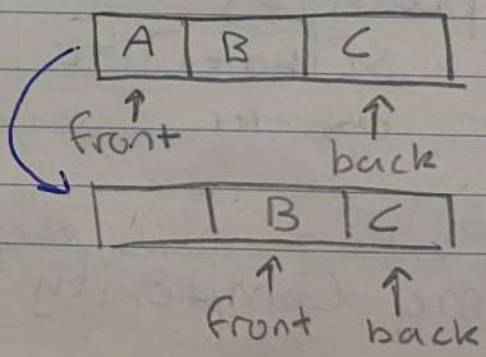
Time Complexity → O(1)




```
dequeue()
    e = Q[front]
    Q[front] = null
    front++
    return e
```

- Removes an element from the front of the queue

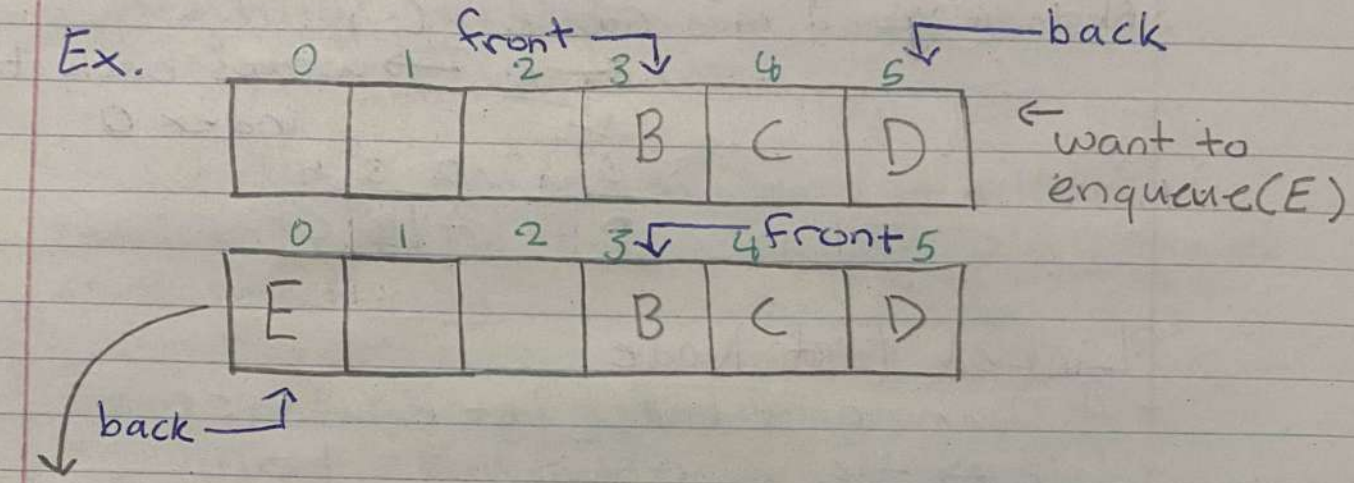
Time Complexity → O(1)



# Circular Queues

- When the pointer pointing towards the back of the queue reaches the end of the array, it wraps back around to index 0.

Ex.

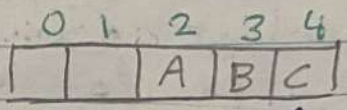



```
enqueue(e)
    if (num_elements < capacity)
      back = (back+1) mod capacity
      Q[back] = e
      num_elements ++
```

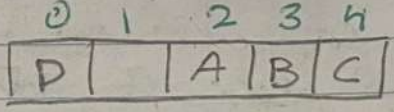
```
dequeue()
    if (num_elements ≥ 1)
    e = Q[front]
    front = (front +1) mod capacity
    num_elements --
    return e
```

Note: we use mod (%) so the queue wraps around to the beginning of the queue

Ex.
Capacity = 5




enqueue(D)
back = 4
(back(4) + 1) mod capacity(5)
= 5 % 5 = 0 → wraps back to index 0




## Linked List Node

- A Node in a linked list contains:
  - data → the value stored
  - next → a pointer that stores the memory address of the next node

```
struct node {
    int data;            ← 32 bits (4 bytes)
    struct node *next;   ← 64 bits (8 bytes)
}                          because it stores the
                           memory address
```

## Example of Nodes linking

```
struct node mynode;   } node variables
struct node nn2;
mynode.next = &nn2;   ← store the memory address of nn2.
```

- mynode.next points to nn2. This links the two nodes together.

Struct node* tp = malloc()

- tp is a pointer to a node
- malloc() allocates memory dynamically

## Creating a Node in C++

node* tp = new node

- Same idea as malloc(), as it allocates memory, except it calls the constructor automatically

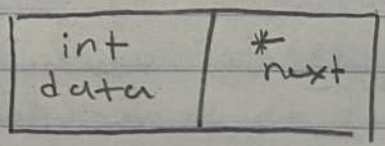
Shortcut for

tp→next = &mynode

(*tp).next = &mynode

- Go to the node tp points to, then access its next field

### Node

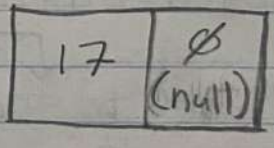



Stores actual value

pointer to the next node in the list

(if last node, next is nullptr)

Ex. mynode



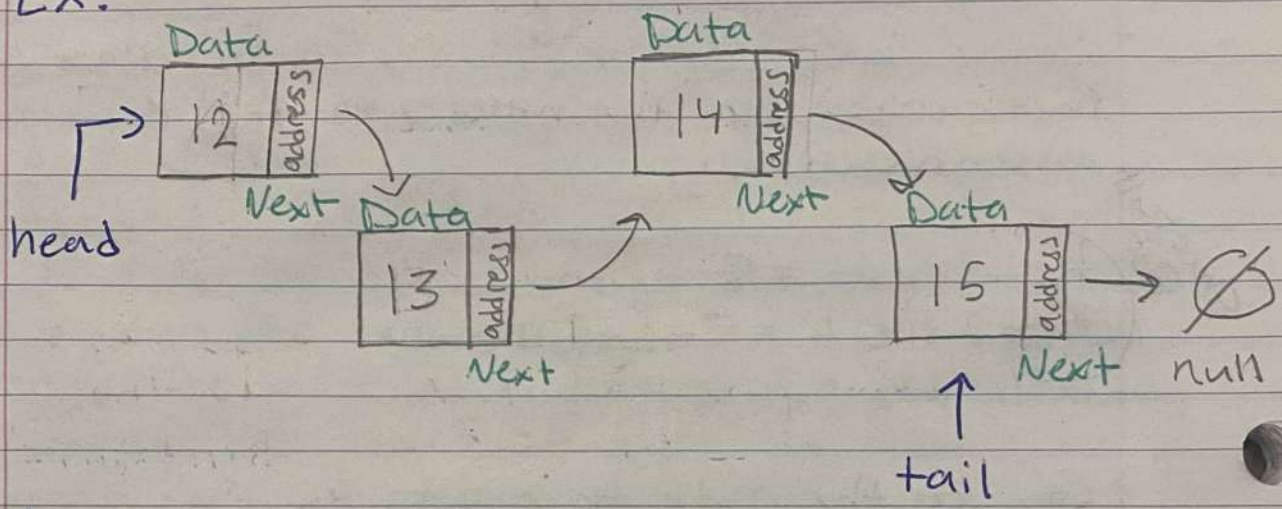

⇒

```
node mynode;
mynode.data = 17;
mynode.next = nullptr;
```

# Linked lists

- A linked list is made up of a long chain of nodes. Each node contains two parts: data we need to store, and an address to the next node in line.

Ex.

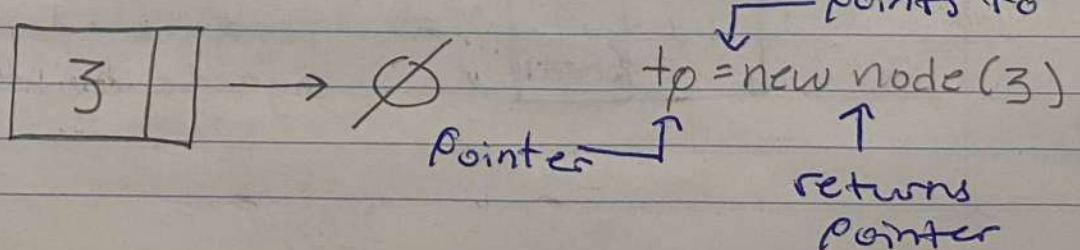



head: points to the first node in a linked list
tp: temporary pointer
tail: points to the last node

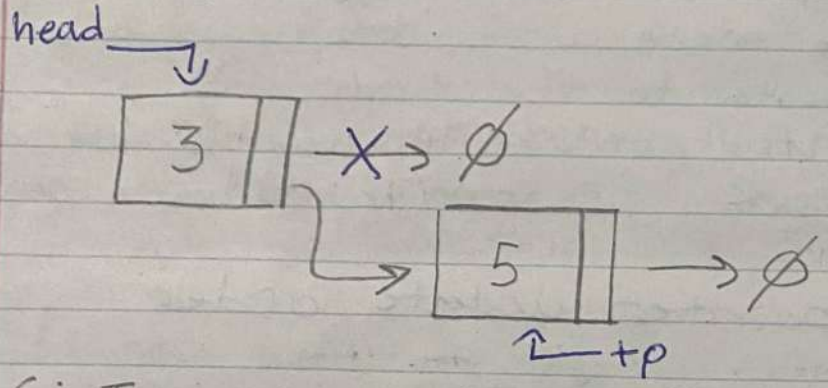
Note: A node needs to have a pointer referencing it. If a node doesn't have a pointer referencing it, it is lost (memory leak).

## Insert

A: insert in an empty list

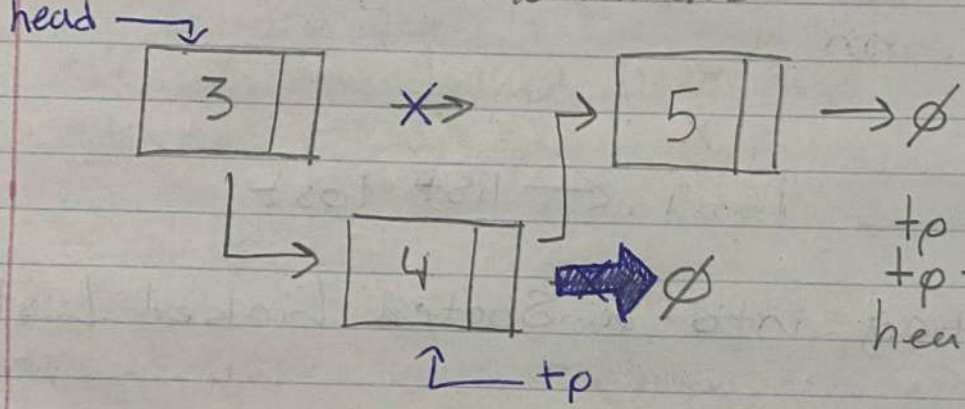


B: insert at tail



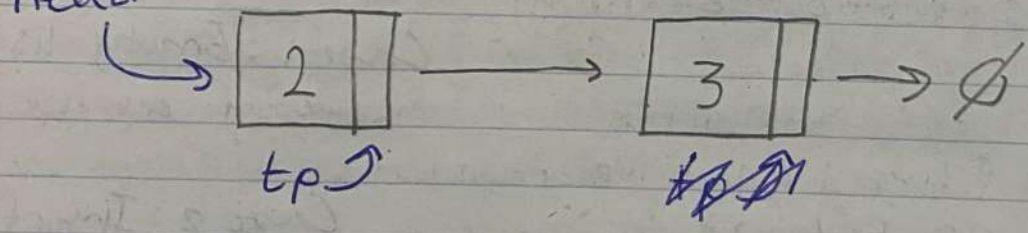

```
tp = new node(5)
head->next = tp
```

C: Insert in the middle



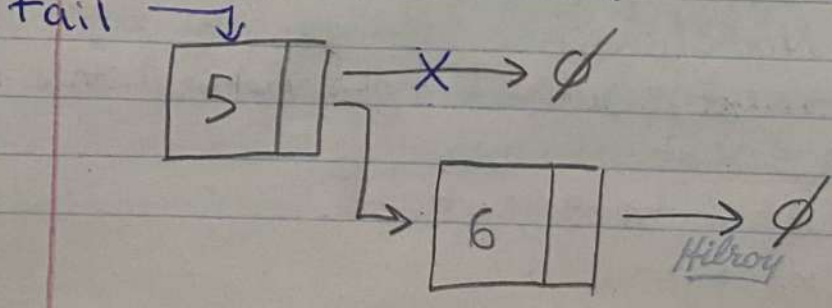

```
tp = new node(4)
tp->next = head->next
head->next = tp
```

D: Insert at head




```
tp = new Node(2)
tp->next = head
head = tp
```

E: Insert at tail




```
tp = new node(6).
tail->next = tp
tail = tp
```

# Inserting

## Basic Steps for ~~Shifting~~

- allocate memory for the new node
- assign data to the node
- Update next pointer to maintain list integrity
- Use temporary pointers (tp) to track positions

## Correct pointer update order ✓

```
nn->next = head
head = nn
```

## ✗

```
head = nn
nn->next = head   ← list lost
```

## Inserting into a Sorted Linked List

inserting a number into a linked list while keeping it sorted.

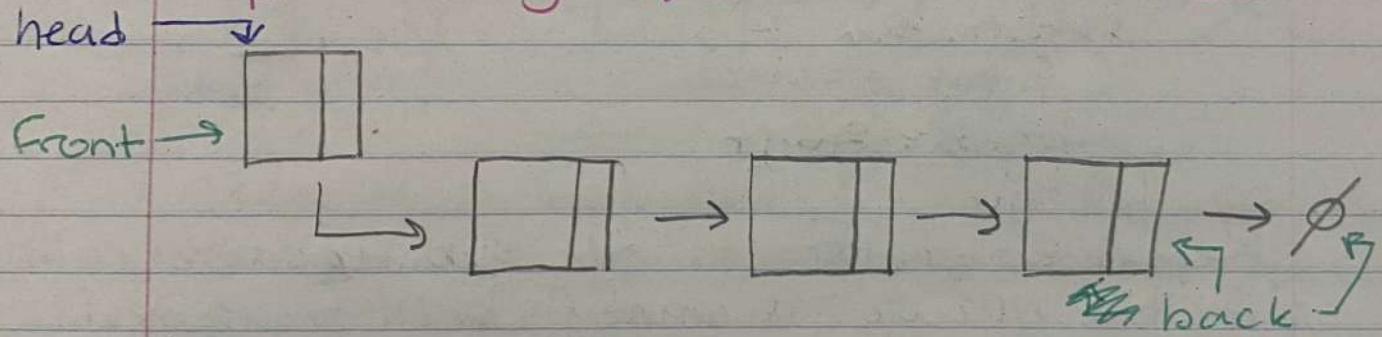
```
InsertSorted(num)
  tp = head                          Case 1: Empty list
  if(tp == null):                      insert as head
    head = new Node(num)
  else if(tp->data > num):           Case 2: Insert before
    nn = new Node(num)                 head
    nn->next = tp                      new node becomes head
    head = nn
    return                           Case 3: Insert in middle
  else:                                or end
    nn = new Node(num)                 traverse to correct spot
    while(tp->next != null && tp->next->data < num)
      tp = tp->next
      nn->next = tp->next
      tp->next = nn
```

# Comparing Arrays and Linked Lists

- Arrays
  - Stored in contiguous memory
  - Supports binary search → $O(\log n)$
  - Inserting / deleting requires shifting → $O(n)$

- Linked Lists
  - Stored in non-contiguous memory
  - no binary search because you must traverse through the linked lists and find the specific address → $O(n)$
  - if you know specific address of node → $O(1)$

# Implementing a Linked List with Queues




- Elements are added at the back (enqueue) and removed from the front (dequeue)

## Operations

enqueue(e) → back
dequeue() → front

```
enqueue(e)
    nn = new Node(e)
    back->next = nn
    back = nn
    if front == ∅
      front = nn
```

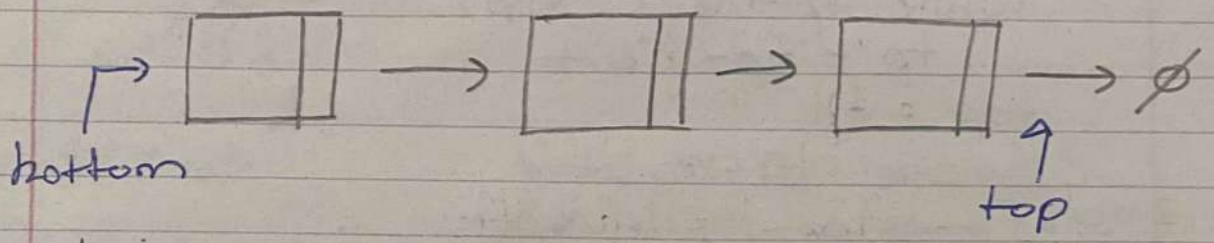
- Adds a new node at the back
- If the queue was empty, both front and back point to the new node.

```
dequeue()
    e = front -> data
    newfront = front->next
    delete front
    front = newfront
    return e
    if front == back
      back = front
```

- Removes the front node and updates pointers
- if the queue is empty, both front and back pointers point towards null.

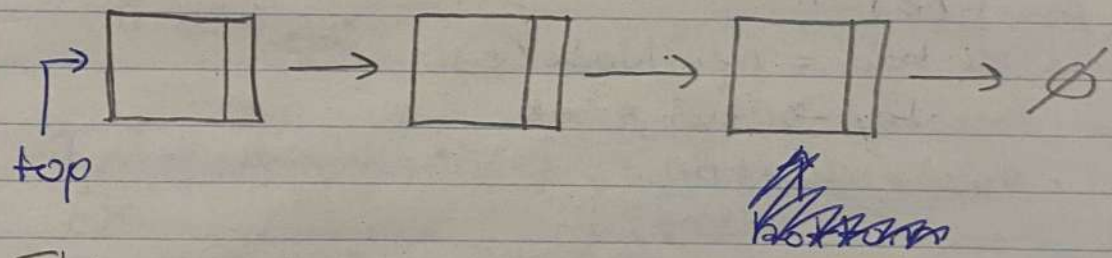
# Implementing linked lists using stacks

## Bad implementation




- traversing a stack from bottom to top is bad for implementation of operations (push(e), pop()) because you need to traverse the entire list ($\Theta(n)$)

## Good implementation




- This implementation is good because the top pointer for the linked list is at the head

## Bad implementation of operations

```
bad push(e)
    new top = new Node(e)
    top→next = new top
    top = new top
```

- Does not account for if top is null

```
badpop()
    e = top->data
    tp = bottom
    while(tp->next ≠ top)
        tp = tp->next
    top = tp
    delete tp->next
    tp->next = nullptr
    return e
```

- this deletes the last node (top) by travelling through the list from bottom to top → O(n)

## Good implementation of operations

```
push(e)
    newtop = newNode(e)
    newtop->next = top
    top = newtop
```

- A new node is created and linked, then and only then is the top pointer assigned.

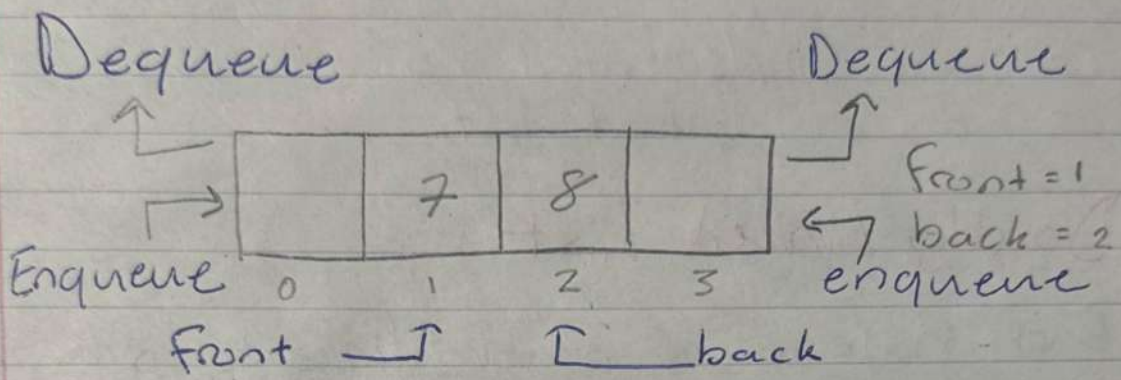
```
pop()
    tp = top
    top = top->next
    e = tp->data
    delete tp
    return e
```

- top node is removed and it's value is returned.

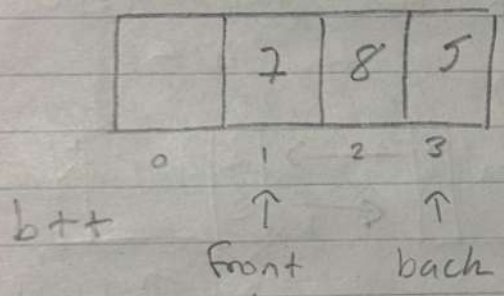
# Double Ended Queue

- enqueue and dequeue can occur from both the back and the front

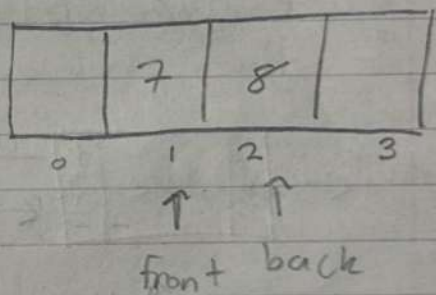
Ex.



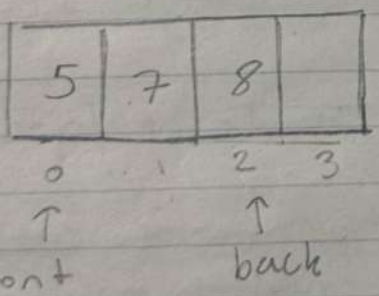

Add to back (5)



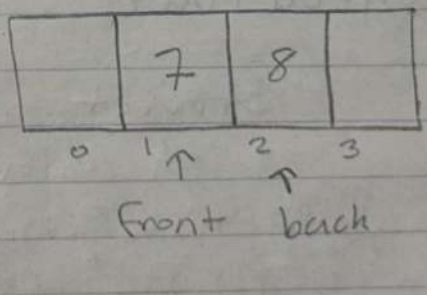

Delete from back. ()




Add to front (5)




Delete from front




- If the queue is empty, reset the front and back to -1 to not waste space

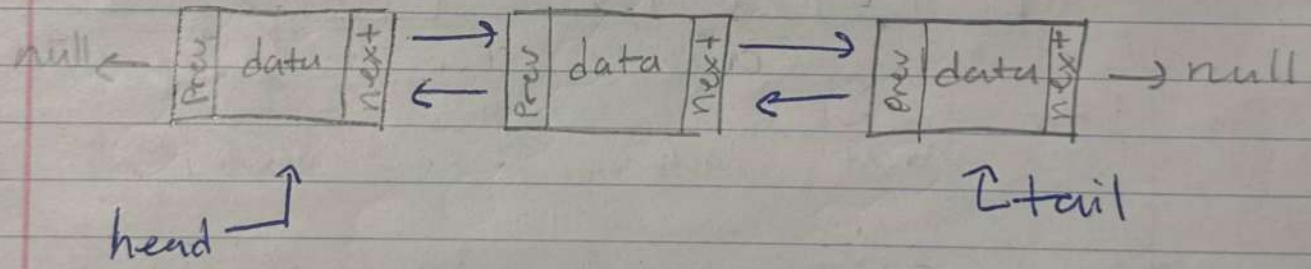
## Basic Operations

- getFront()
- getBack()
- isFull()
- isEmpty()
- insert at front
- insert at back
- delete from front
- delete from back

- Can be implemented as a circular array or doubly linked list

## Doubly Linked List

- Each node in a doubly linked list contains data and two reference variables
- One reference variable refers to the next node in the list, the other refers to the previous node in the list.




## Structure

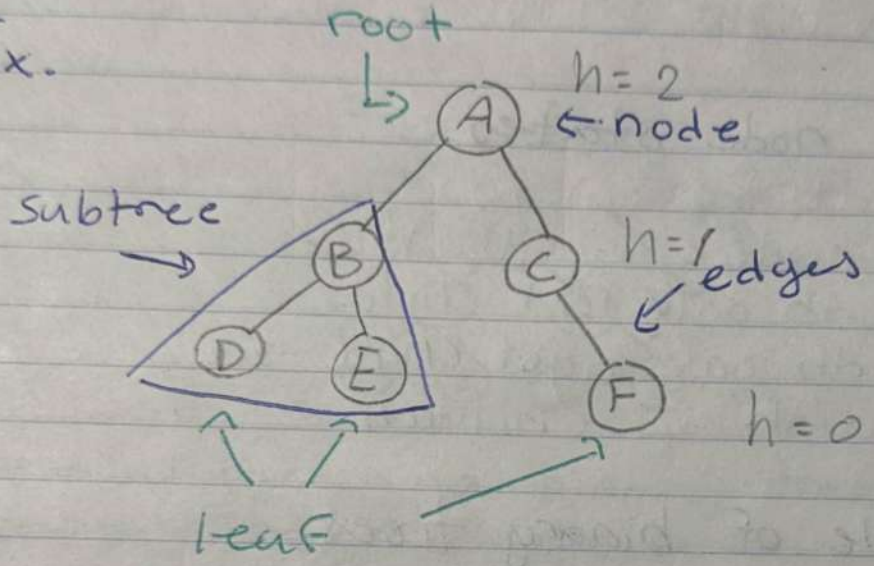
```
Struct node {
  int data;
  node* next;  ← pointer to next node
  node* prev;  ← pointer to previous node
}
```

# Hiarchial Data Structurs

## Trees

- A tree is a hierachical data structure made up of nodes

Ex.




root : The top node of a tree
leaf node : a node where the left and right children are null.
Subtree : a smaller tree held within a larger tree
height : number of edges above furthest leaf node

## Implementation

```
Struct Node {
    int data;
    struct node* children;
    int num_of_children;
}
```

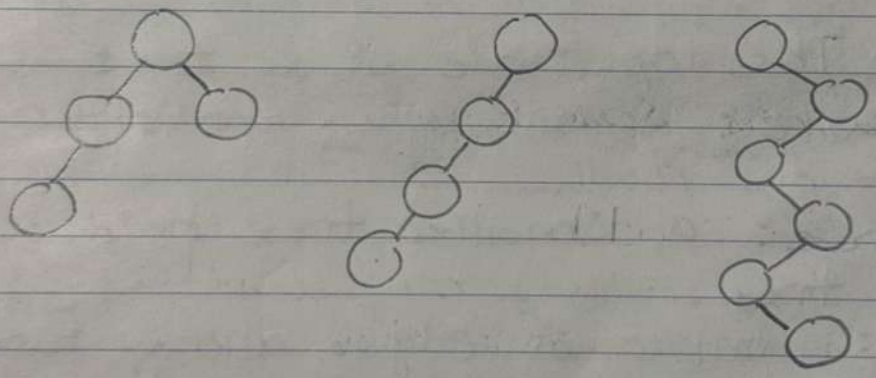
# Binary Tree

- A tree where each node in the tree has no more than 2 children.
- Each node has at most a left child and a right child

## Possible node states

- empty tree
- Single node
- Node with only left child
- Node with only right child
- Node with 2 children

## Example of binary trees



## Structure

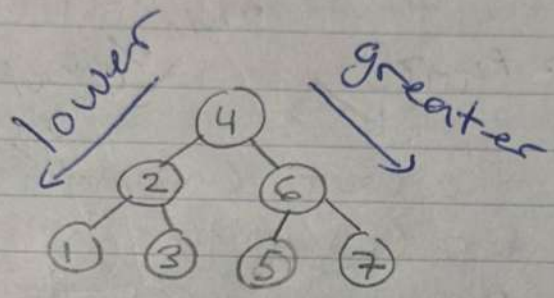
```
Struct node {
  int data
  Struct node* left
  Struct node* right
```

# Binary Search Trees (BST)

## BST property

- Starting at the root node, the left most child should have the lowest value, and the rightmost child should have the greatest value.

Ex.




any node in a binary tree needs to follow the BST property

## BST Search

Recursively searching

```
Search(e, root):
    if (root == null) return null
    if (root → data == e) return root
    if (e < root → data)
        return Search(e, root → left)
    if (e > root → data)
        return search(e, root → right)
```

Iterative Search

```
IterSearch(e, troot):
    while (troot ≠ null)
        if (troot == e) return troot
        if (e < troot → data)
            troot = troot → left
        if (e > troot → data)
            troot = troot → right
    return null
```

- Start at the root. Compare the value being searched for (e) with the temporary pointer troot → data. Search the left side if e is smaller, search the right side if e is larger. Stop and return if found or null.

## how to account for duplicate data

- add a count field

```
struct node {
    data
    Struct Node* left
    Struct Node* right
    Count ← Count the amount of duplicates
}
```

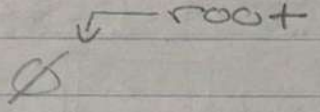
# BST insert

recursive insert

```
BstInsert(e, root)
  if (root == null) root = new Node(e)
  if (e < root->data)
    root->left = BstInsert(e, root->left)
  else if (e > root->data)
    root->right = BstInsert(e, root->right)
  else
    root->count++ ← handle duplicates
  return root
```

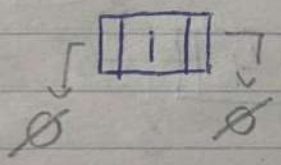
Ex.

We want to insert [1, 2, 3, 7, 4]

① 


② create node 1 and insert

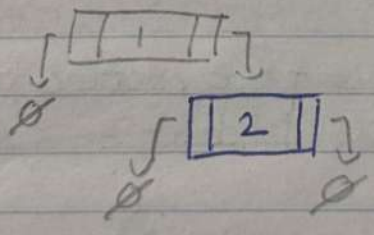
BstInsert(1, root)




③ Create node 2 and insert

2 > 1 → go right

BstInsert(2, root)

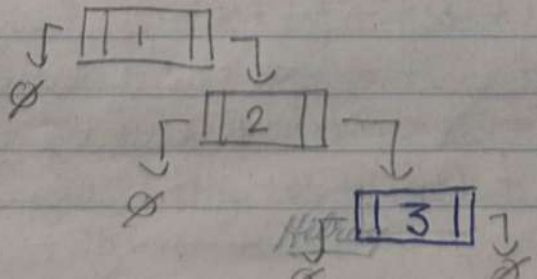



④ Create node 3 and insert

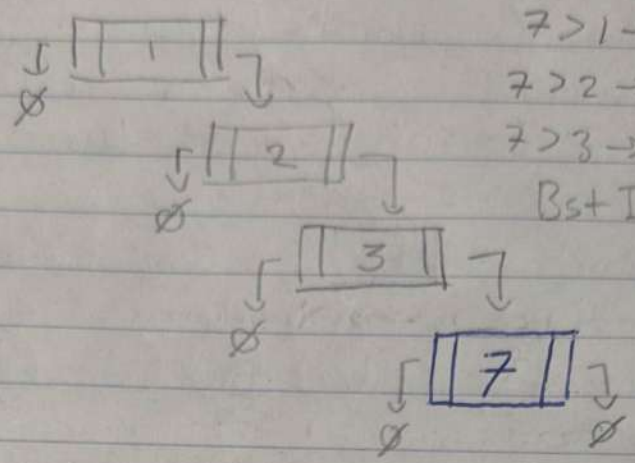
3 > 1 → go right

3 > 2 → go right

BstInsert(3, root)

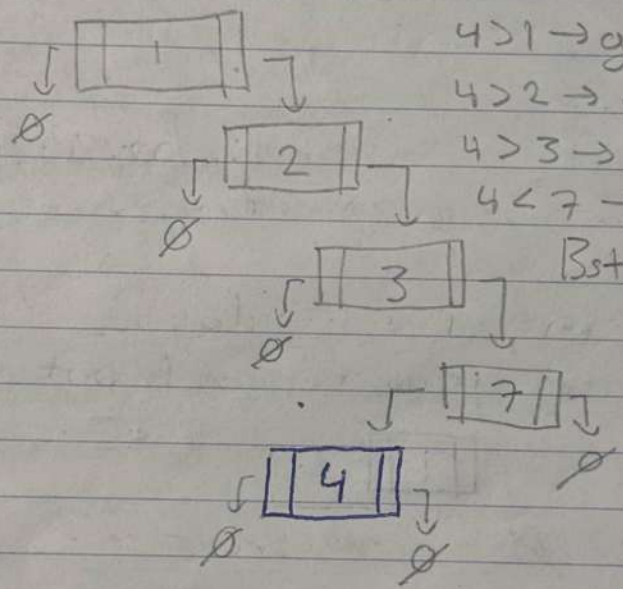


⑤ create node 7 and insert




7 > 1 → go right
7 > 2 → go right
7 > 3 → go right
BstInsert(7, root)

⑥ create node 4 and insert




4 > 1 → go right
4 > 2 → go right
4 > 3 → go right
4 < 7 → go left
BstInsert(4, root)

## Iterative Search

```
IterativeBstInsert(e, troot)
  if (troot == null) troot = new Node(7)
  while(1)
  if (troot->data == e) return
  if (troot->data > e)
    if (troot->left == 0)
      troot->left = new Node(e); return
      troot = troot->left; continue
  else if (troot->right == null) troot->right = new Node(e)
    troot = troot->right
```

# Traversal

- The process of visiting all of the nodes in a tree
- There a 3 types of traversal: In order, pre order and post order.

## Pre order Traversal visit order

root → left → right

## Inorder Traversal visit order

left → root → right

## Postorder Traversal visit order

left → right → root

## Inorder Traversal

```
inorderTraversal(root):
    if(root == null) return
    inOrderTraversal(root->left)
    Print(root->data)
    inOrderTraversal(root->right)
```

## Pre Order Traversal

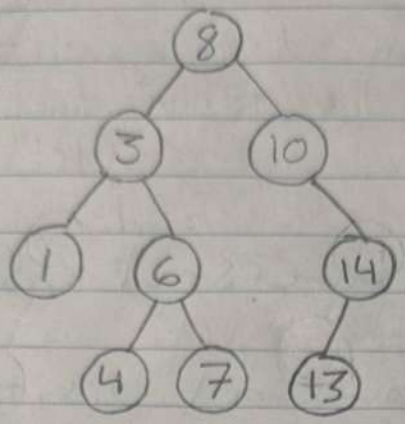
```
preOrderTraversal(root):
    if(root == null) return
    print(root->data)
    preOrderTraversal(root->left)
    preOrderTraversal(root->right)
```

## Post order Traversal

```
postOrderTraversal(root):
    if(root == null) return
    postOrderTraversal(root->left)
    postOrderTraversal(root->right)
    print(root)
```

Hilroy

Ex.




Inorder Traversal output
1, 3, 4, 6, 7, 8, 10, 13, 14

Preorder Traversal output
8, 3, 1, 6, 4, 7, 10, 14, 13

Postorder Output
1, 4, 7, 6, 3, 13, 14, 10, 8

Finding the smallest and largest node

- Smallest: traverse left until null

```
Smallest (root):
  while (root->left ≠ null)
    root = root->left
  return root
```

- largest: traverse right until null

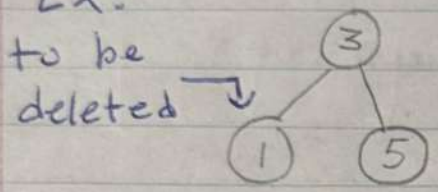
```
largest (root):
  while (root->right ≠ null)
    root = root->right
  return root
```

# Deletion

- Deletion with BST has 3 cases:
  - The Node has no children (leaf)
  - The Node has either a left or right child
  - The Node has 2 children

## Case 1: no children

Ex.



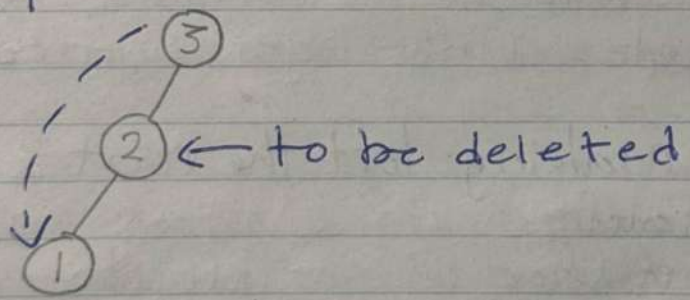

- we want to delete 1

Steps

- Find 1
- create to be deleted pointer
- set parent's pointer to null
- delete 1

## Case 2: 1 child

Ex. 1

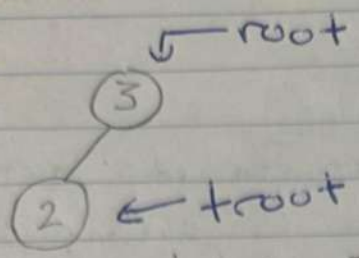



- we want to delete 2

Steps

- Find 2
- connect 2's parent directly to 2's child
- delete 2

Ex. 2

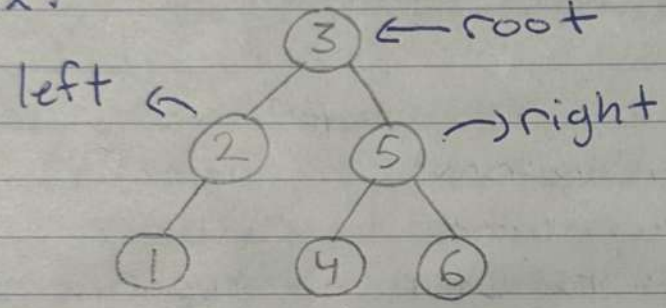



- we want to delete 3

Steps
- assign a temporary pointer (troot) to new root
- delete 3
- point root pointer to new root

Case 3: 2 children

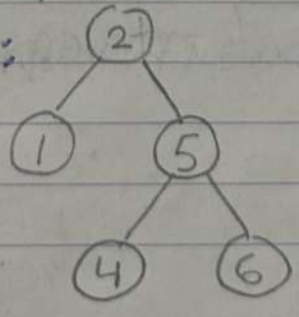
Ex.




- we want to delete 3
- to delete 3, we must either replace it the largest node in the left subtree, or the smallest node in the right subtree

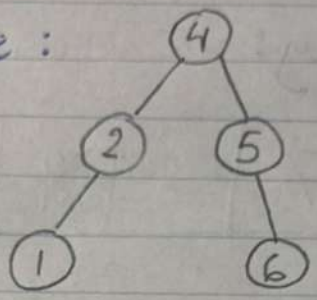
Option 1: making left the root
- go to the left
- go right as far as possible to get largest node in the left subtree
- In this example, delete 3, then replace the root with 2.

new tree:



Option 2 : making right the root
- go to the right
- go to the left as far as possible to get smallest node in right subtree
- In this example, delete 3, then replace the root with 4

New tree :




Deletion algorithm
1. Check if the root is null
2. Find node to delete
3. Evaluate the cases:
- If the node has no children, delete it directly
- If the node has one child, connect its parent to its child
- If the node has 2 children:
- Find the parent of root replacement, either:
- largest among smaller nodes than root
- Smallest among larger nodes than root
- adjust the parent's pointers
- largest: parent_largest->right = largest->left
- Smallest: parent_smallest->left = smallest->right
- replace the root with either the largest or the smallest
- keep original children attached

# BST Sort

① Insert elements from an array into BST
② Perform In Order traversal to copy the elements from the BST back into an array

Ex. array:

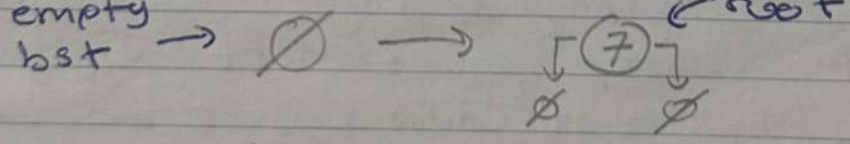
| 7 | 8 | 9 | 5 | 2 | 3 |
|---|---|---|---|---|---|

## Step 1: Insert elements into BST

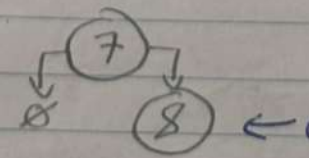
- Use BST insert to start inserting elements into the BST
- If the tree is empty, make the element the root
- Compare the new node with the existing nodes of the tree
  - → if the value is smaller, insert it on the left
  - → if the value is larger, insert it on the right
- repeat until null, then insert

Ex.

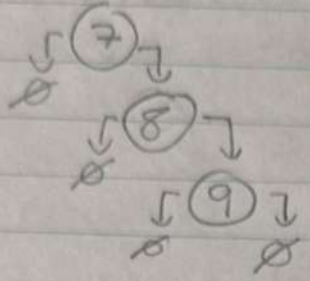
- Start with 7, set as root



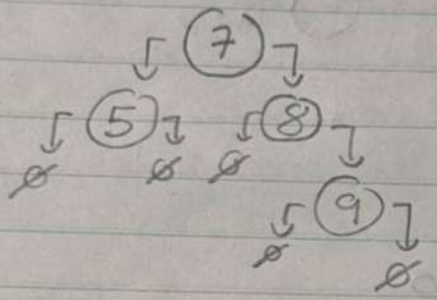

- Compare 8 with the root, It is larger so we move it to the right pointer



- repeat with 9

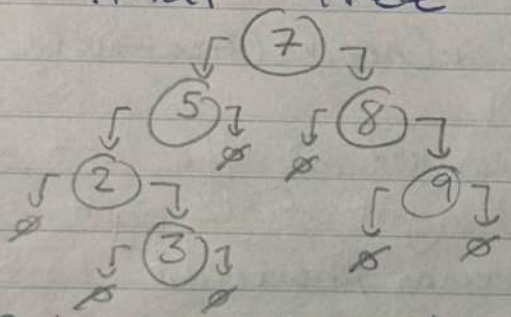



- add 5 to left because it is smaller than the root (7)




- repeat for 2 and 3

Final tree




Step 2: use in order traversal

- we use in order traversal to traverse throught the tree

Result: 2, 3, 5, 7, 8, 9

Time Complexity

insert: $O(n \times n)$ => $O(n^2)$

↑ number of elements ↑ Worst case insertion

inorder traversal: $O(2n)$ => $O(n)$

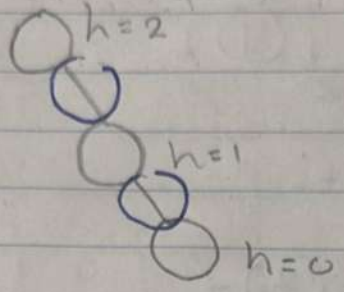
↑ number of comparisons

Hilroy

## Binary tree height

- number of edges that it takes to get from a node to a leaf node

Ex.




height(null) = -1
height(leaf) = 0

## height function

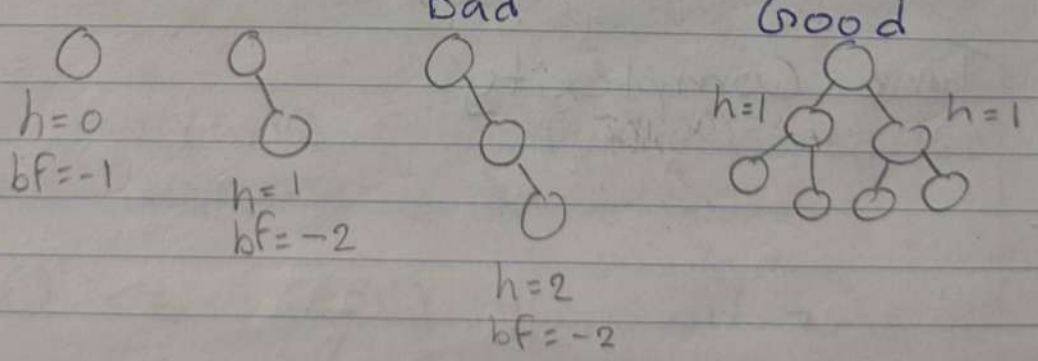
```
height(tree):
    if(tree == null) return -1
    return max(height(tree->left, height(tree->right))+1
```

- Compute the height of the left subtree, compute the height of the right subtree. Take the maximum, then add 1 for the current edge.

## Balance Factor

bf = height(left) - height(right)

Ex.




- Searches become O(n) when trees aren't balenced

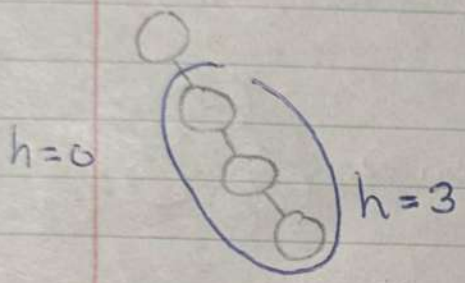
# AVL Trees

## AVL property

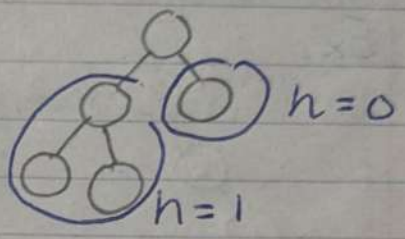
$$-1 \leq bf \leq 1$$

- when the balance factor exits from this range, a rotation must be performed

Ex.




bf = (0) - (3) = -3
bf of -3 at the root
- This tree violates the AVL property




bf = (1) - (0) = 1
bf of 1 at the root
- This tree respects the AVL property

Note left-heavy => +, right-heavy => −

## AVL node structure

```
Struct Node {
    int data
    Struct node* left
    Struct node* right
    int bf
    int height
}
```

# AVL Rotation

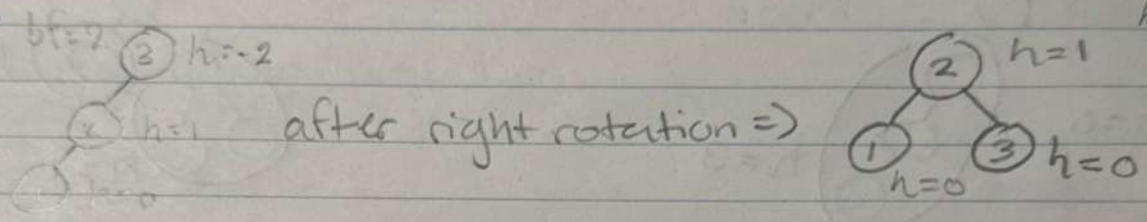
- when the bf becomes ±2, we rotate to fix imbalences

## Types of rotation

### case 1 : right rotation

- This case happens when the root is left heavy (bf+2) and the left child is left heavy (bf+1)

Ex.



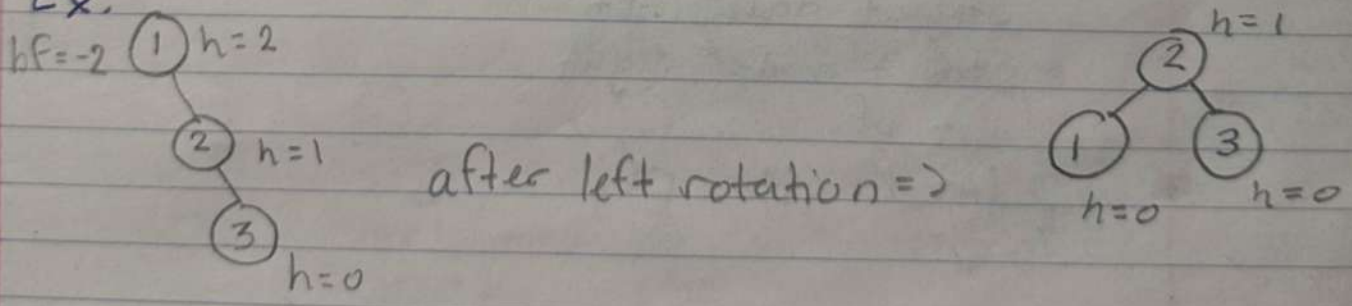

### rotation Steps

tp = root → left
root → left = tp → right
tp → right = root
root = tp

### Case 2 : left rotation

- This case happens when the root is right heavy (bf = -2) and the right child is right heavy (bf = -1)

Ex.



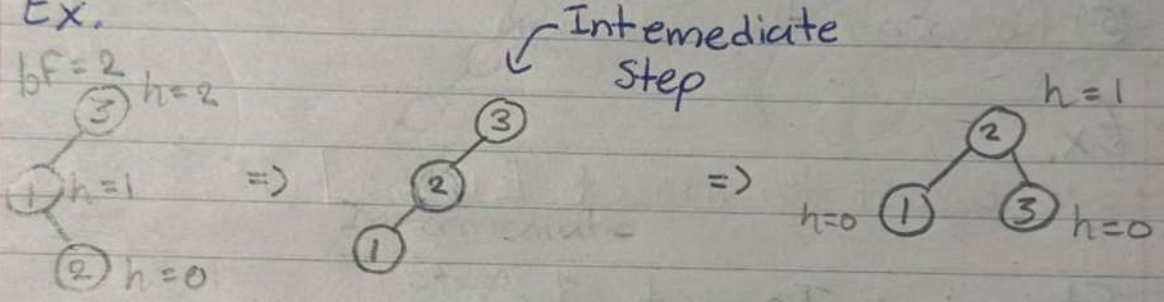
Rotation Steps

```
tp = root->right
root->right = tp->left
tp->left = root
root = tp
```

## Case 3: left-right rotation

- This case happens when the root is left heavy (bf = +2) and the left child is right heavy (bf = -1)

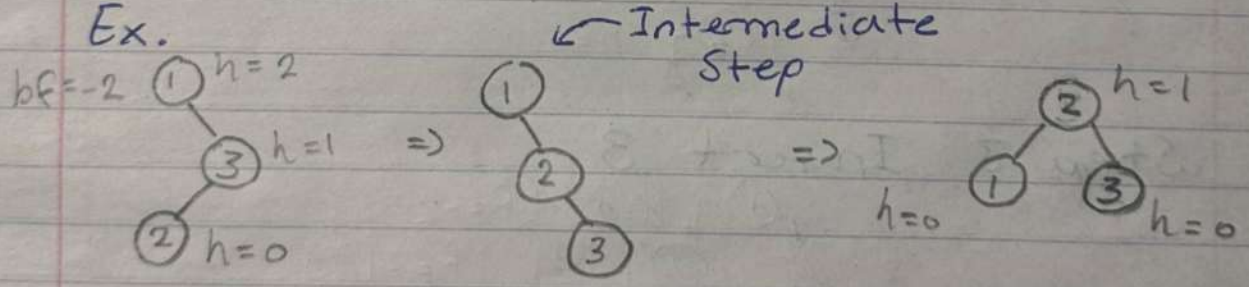
Ex.




## Case 4: right-left rotation

- This case happens when the root is right heavy (bf = -2) and the right child is left heavy (bf = +1)

Ex.



# AVL Insertion

- with AVLBST, we insert like a normal BST, and when bf isn't between -1 and 1, we rotate appropriatly.

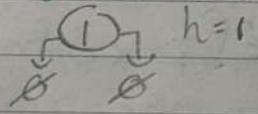
## Insertion steps

1. Create node
2. Insert the node into the tree
3. update the bf of each of the nodes
4. Detect the type of imbalence
5. Perform the rotation

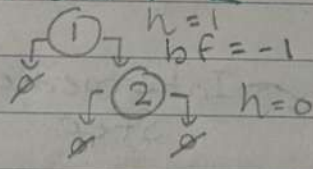
## Ex.

want to insert [1, 2, 3]

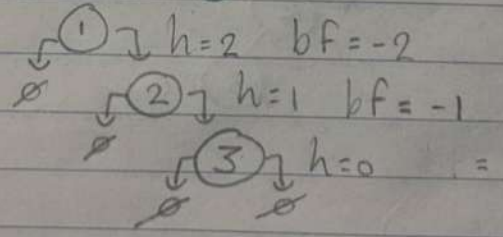
Step 1 : Insert 1




Step 2 : Insert 2

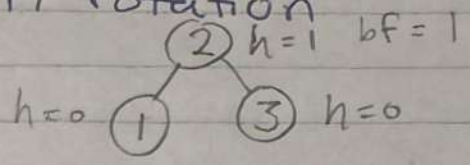



Step 3 : Insert 3




- Since the root (1) is right heavy and the right child (2) is right heavy, we need to perform a left rotation

after left rotation




Time Complexity

| operations | BST | AVL BST |
|---|---|---|
| Insert | $O(n)$ | $O(\log n)$ |
| Remove | $O(n)$ | $O(\log n)$ |
| Search | $O(n)$ | $O(\log n)$ |
| largest/smallest | $O(n)$ | $O(\log n)$ |
| next largest/smallest | $O(n)$ | $O(\log n)$ |

BST Sort: $O(n^2)$
AVL Sort: $\Theta(n \log n)$