:) Time Complexity : # of computations done based on the size of input

:) Arrays : A contiguous block of memory
(contiguous: no gap)

- Drawbacks of array :
  - size is fixed
  -

:) List (ADT) : An ordered, 0-aligned, contiguous collection of data
(0-aligned: start from 0; contiguous: no gaps)

- Two standard implementations of List :
  - Array List
  - Linked List

:) Array List : Use an array as an underlying structure

- AddBack() : O(1)
  - If the underlying array is full → make the new array with similar data but larger capacity : O(n)

We denote this situation as $O(1)^*$, where * = "armortized" = "average worst case" mostly fast, sometimes slow

- Worst case of adding back n times (assume the underlying array has size n) $= \frac{O(n) + (n-1)O(1)}{n} = O(1)$

- AddFront() : O(n)

- AddAtIndex() : O(n) (the worst case is that user want to add front)

- RemoveBack() : remove the item (change it to null)

Note : Extra work is okay if it is O(1)

) Prefer Array over ArrayList when:
  - Data is fixed size
  - Want gaps
  - Save memory

) Single Linked List : each data is stored with a pointer to the next node

```
Def : class LinkedNode<T> {
        T data
        LinkedNode<T> next
      }

      class LinkedList<T> {
        LinkedNode<T> head
        int size
      }
```

- Add to front ():

```
newNode = new LinkedNode(data)
newNode.next = head        (head now become newNode.next)
head = newNode             (re assign head)
```

- size = 0 is not a problem

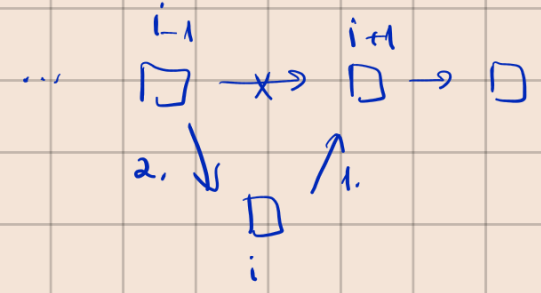
⇒ O(1)

- Add Back ():

```
// error check
curr = head
while curr.next != null:
    curr = curr.next

newNode = new LinkedNode(data)
curr.next = newNode
```

→ O(n)

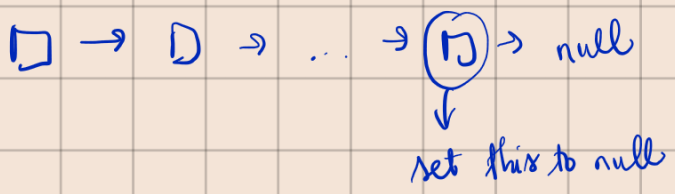
- AddAtIndex (i)



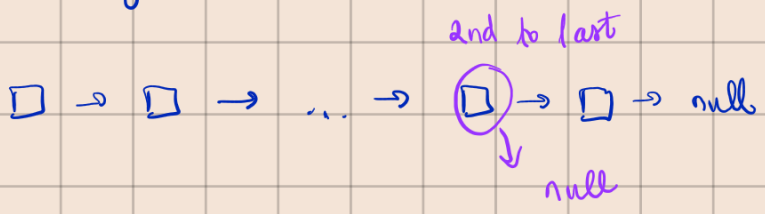
- Remove First:

waste? No if we are using java because it has a garbage collector that take care of objects that is no longer accessed by both client & programmer

- Remove Last:



→ wrong



How to acces the 2nd to last ? curr.next.next == null ?

- size = 0 → Input validation
- size = 1 → Null PT exception

→ O(n)

- Remove At Id : Worst-case is n − 1 → O(n)

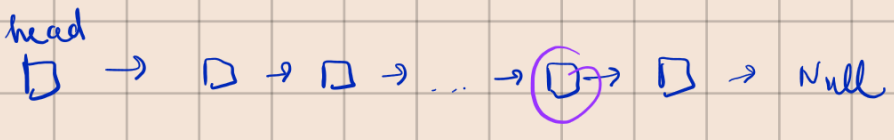
AddFront, RemoveFront → fast
AddBack, RemoveBack → slow
→ We can add a "tail"

- class LinkedList <T> {
  LinkedList <T> head
  LinkedNode <T> tail
  int size

- Add Back w/ tail

```
tail.next = NewNode(data)
tail = tail.next
```

- Remove Back w/ tail:



need 2nd to last Node
→ has to traverse the list again

why don't we a pretail pointer ?

After pretail → null
tail → pretail
pretail → ??? broken

- How to step backwards to the previous node

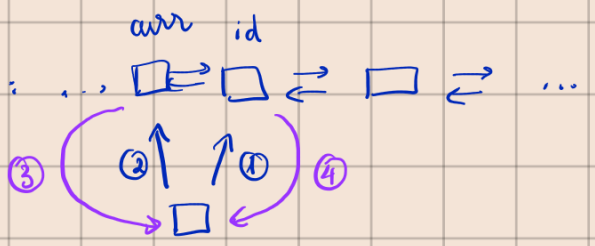
- Doubly-Linked list : each node contains references to both the next and previous node

- Remove Back ():

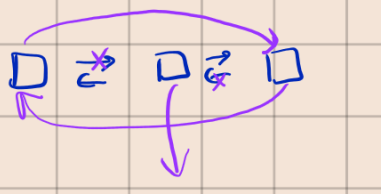
  tail = tail . previous

  tail . next = null

) Add At Index :



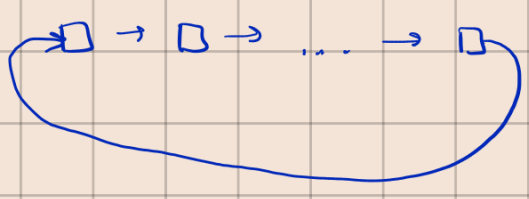
) Remove At Index :



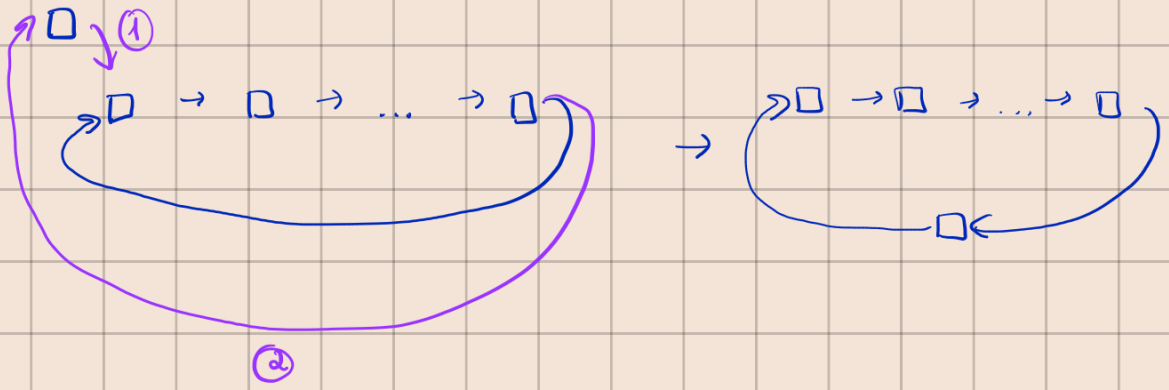
) Why prefer SLL over DLL :

- DLL uses more memory
- can hold more nodes in cache
- less complicated code

) Circular SLL :



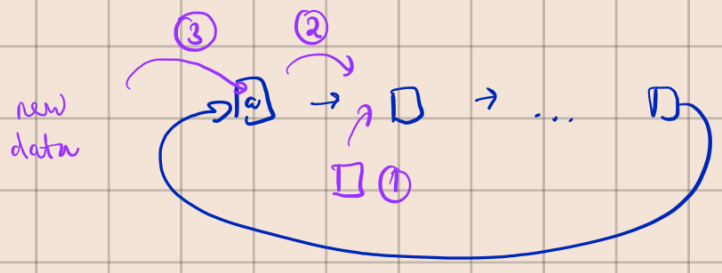
- Add Front :



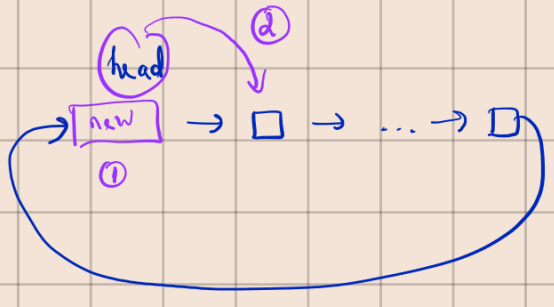
How to do ② ?

① iterating to the end, change the ptr → O(n)

② We can move data between node :

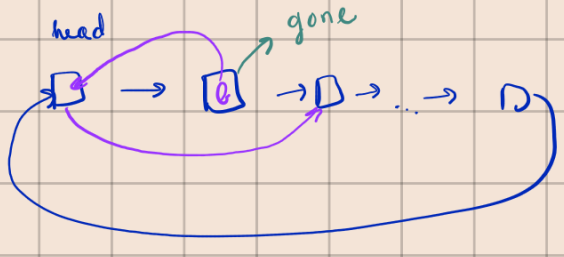


- Add Back:
  - ① addFront (data)
  - ② moveFrontToBack ()



head = head.next

- RemoveFront :



head.data = head.next.data

head.next = head.next.next

Not worked for CSLL with 1 node

- Remove Back : Need to identify the 2nd-to-last

→ Iterate, stop at curr.next.next = head

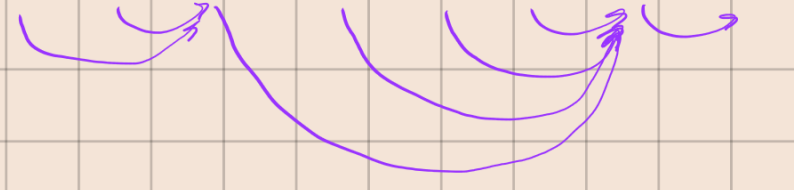
- Recursion : "ptr reinforcement" - each node determines whether it remains in the list

- Remove Even : 1 → 2 → 3 → 4 → 7 → 9

↳ 1 → 3 → 7 → 9

a node v has even data, the ptr points to the node must point somewhere else

Ex of running 1 → 4 → 7 → 6 → 2 → 8 → 3 → null



⇒ 1 → 7 → 3

The method Recursive Helper should return the node that a node should point to

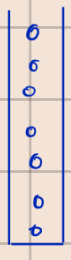
Divide : $T(n) = T(n-1)$

Conquer :

curr.next

·) Puzzle : only remove/add from top

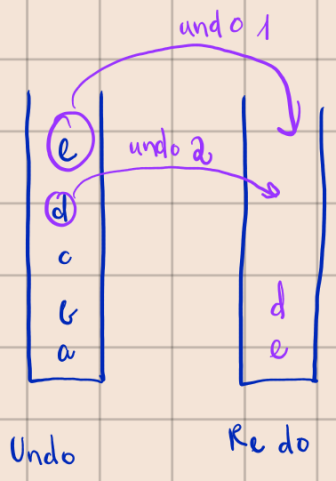
·) Stack :



- Always remove the most recently added "newest" data
  - The data at the bottom stay there the longest

(Last In First Out - LIFO)

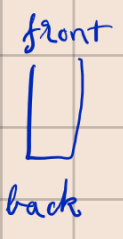
- Ex : Undo / Redo



.) Behavior : add Top = push () No add Bottom
remove Top = pop () remove Bottom
get Top = peek ()

- Array Stack
- Linked Stack

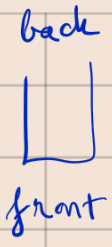
.) Linked Stack : use Linked List



- top = front

⇒ { push = add Front O(1)
pop = remove Front O(1)

- Implementation for Linked Stack is the same with Linked List



-) Array Stack : use Array List

  . top = back ⇒ { push = add Back
  pop = remove Back

-) Problem of Stack : don't have remove / add At Index

  · If don't need those, then choose Stack

-) Queue : → ——— → (back, front)

  Add To Back = enqueue No add / remove
  Remove From Front = dequeue At Index

  · Linked Queue : try using a linked list

    · add To Front { enqueue = add Front
    dequeue = remove Back (hard)

    · add To Back { enqueue = add Back O(1) → need "tail"
    dequeue = remove Front O(1)

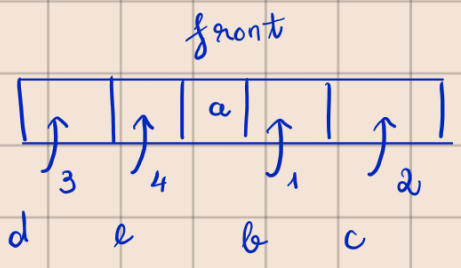
  · Array Queue :

    · add To Front ⇒ { enqueue = add To Front O(n)
    dequeue = remove From Back O(1)

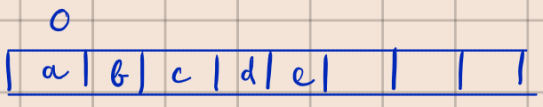
    · add To Back ⇒ { enqueue = add to Back O(1)*
    dequeue = remove Front O(n)

  ⇒ Bad ⇒ Cannot use Array List

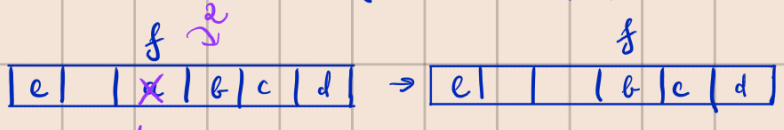
- Circular Array : contiguous, don't need to be 0-aligned, but must keep track of index "front"



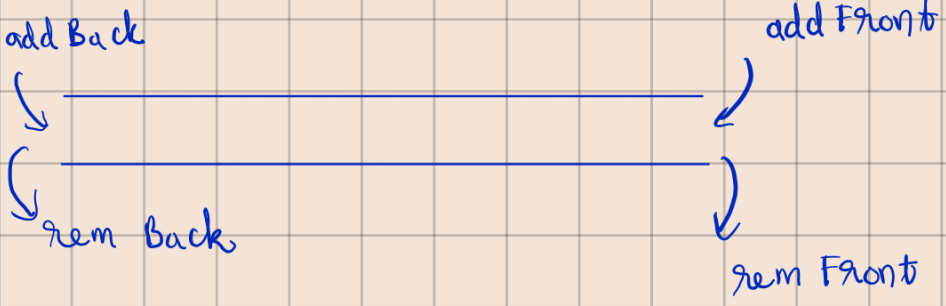
(front + id) % len

- resize () → 

- enqueue () = addBack ((front + size) % len (backArr))
- dequeue () = removeFront ()



- Deques (Double Ended Queue)



- stairs : deque
- escalator : queue
- elavator : stack

- Linked Deque :

- LinkedStack : SLL
- Linked Queue : SLL w/ tail, CSLL
- Linked Deque : DLL
  ↓ make remBack() O(1) too

→ Linked Deque is DLL without addIndex() & remIndex()

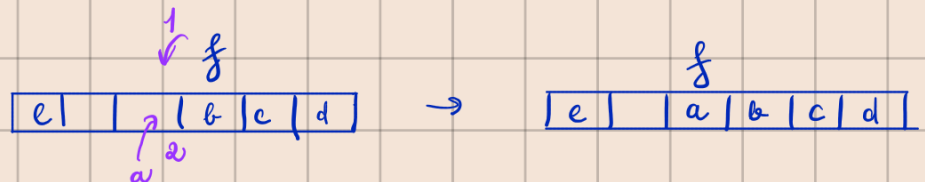
) Array Deque :

- Array Stack : array List
- Array Queue : circular array
- Array Deque : circular array

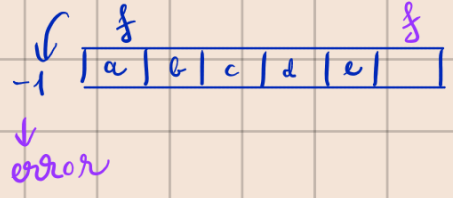
- addBack()
- remFront()

) from LinkedQueue

- addFront() :



- Note :



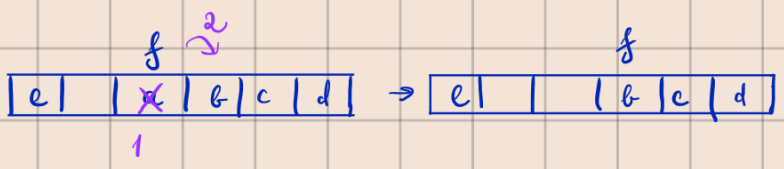
Move f to front in CArr : $f = (f - 1) \% \text{len}(\text{backArr})$

Problem : "%" in java is not mod, it's remainder

Sol : $f = f - 1$

if $(f == -1)$: $f = \text{len}(\text{backArr}) - 1$

- remBack() :



$f = (f+1) \% \text{ len(backarr)}$

$\geq 0 \rightarrow$ no problem

Why don't use CArr instead of ArrList ?

- Doing mod is more expensive than just +/-

:) Tree : nodes connected by edges where :

- parent
  O
  ↓
  O child

- A node can have any # of children
- _____ can only have 0 or 1 parent
- Exactly 1 node with 0 parent → "root"
- No cycles

:) Leaf : Node that has 0 children

- Internal node : node that is not a leaf

:)

depth ↓ height ↑

A, B are the ancestors of C

B, C _____ descendants of A

C, D are siblings

depth of C is 2 , height of A is 2

- Subtree

- Depth : depth of parent + 1

- Height : max (heights of kids) + 1

- Binary Tree : At most 2 children per node, "left" and "right"

  - Full : BT is full if each node has exactly 0 or 2 kids

  - Complete : BT is complete if

    lowest depth

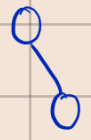
    (1) Each depth of the tree except the lowest must have maximum # nodes

    (2) Lowest depth must be filled from left → right

    → For each n, there is exactly one shape a complete tree has

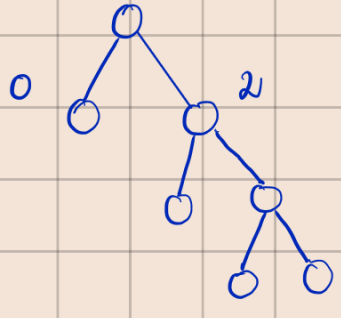
  - Balanced : A node is balanced if its children have height that differ by 0 or 1. A tree is balanced if every node is balanced
    - The height of null node is -1
    - Leaf nodes are always balanced

balanced but not complete



complete but not full

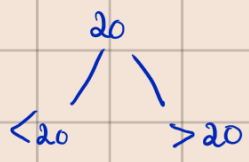
full but not balanced



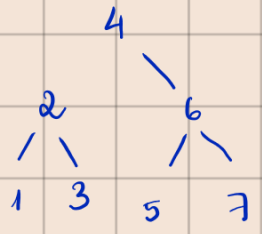
- Heap : a data structure that stores data in a tree-based structure

- Binary Search Tree : A binary tree where :

- Each node contains x
- All the data in the left subtree must $< x$
  right subtree $> x$



) Pre-order traversal : go through all nodes of BST from ↓ , →



→ (4, 2, 1, 3, 6, 5, 7) : Pre OT

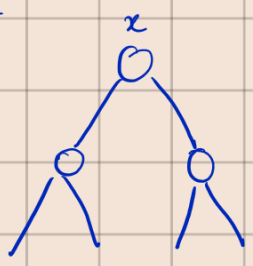
- Each BST has a unique ordered Pre OT

) Post-order traversal : go through all nodes of BST from ↑ , →

- if we remove on Post OT, we will only remove leaves

) In-order traversal : left subtree → root → right subtree

- Add () : we add to leaf



goes down where x should be

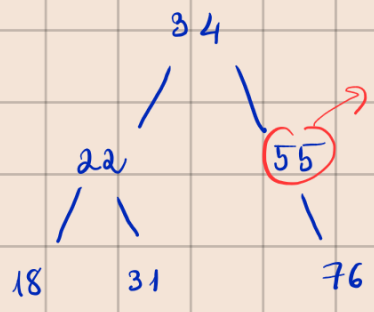
```
public void Add(T data)
  root = addH(root, data)

private Node addH(Node curr, T data):
  if curr == null:
    Node newNode = new Node(data)
    return newNode
  if curr.data == data:
    //do nothing, data already in tree
  if curr.data < data:
    //recurse right
    curr.right = addH(curr.right, data)
  if curr.data > data:
    //recurse left
    curr.left = addH(curr.left, data)

  return curr
```

return node replaces curr

ptr reinforcement

replace null w/ new Node

- Remove () :
  - If no children : similar to add
  - Has 1 child : remove 55



Solution : connect 34 to 76

  - Has 2 children :
    - Successor of x is the smallest data item in tree > x
    - Predecessor of x is the largest _____ < x
    - Finding successor : 1 right → left until goes to the end
      predecessor : 1 left → right

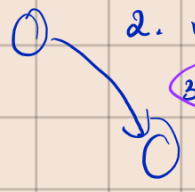
- When removing a node with 2 children, the node doesn't go away, but replaced wit predecessor / successor 's data. The predecessor / successor that we used will be removed (removing with 1 / 0 child this time)

- Problem with coding in Java regarding ptr reinforcement:

remove

1. remove x
2. what pt at?
3. what was removed ?

) return 2 things → hard to do in Java



↓

store in a dummy node

) Priority Queue: each item has some assigned priority

- enqueue ()
- dequeue () → removes highest prio data

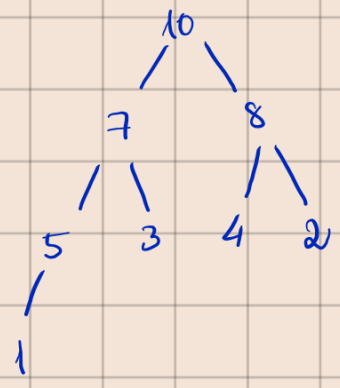
enqueue (14)

↓

- using heap
- using BST

) Heap: 2 conditions

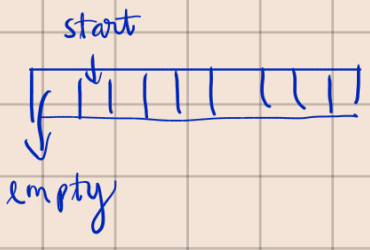
- Complete tree
- Each node must be higher order than its children



max Heap : node > children

min Heap : node < children

- Use an array to store the heap data



kids of data at index i : $2i$, $2i+1$

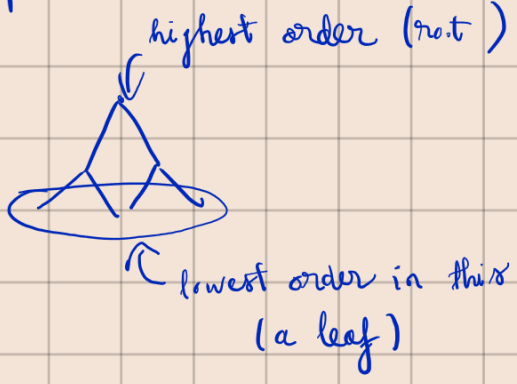
parent _____ : $\lfloor \frac{i}{2} \rfloor$

- Add() :
  - Maintain completeness
  - Fix order ↓ (Up Heap)

terminating conditions [in good relationship w/ parent
at root

Runtime : $O(\log(n))^*$

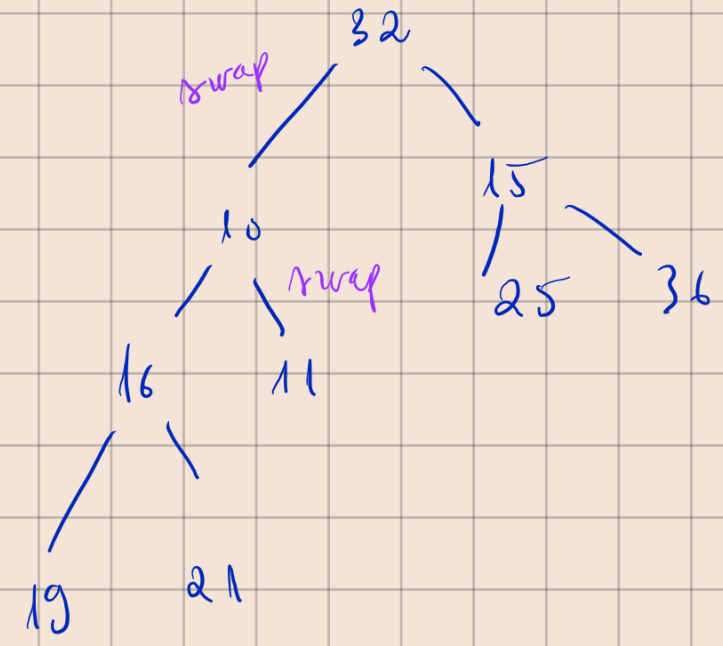
:) Max heap

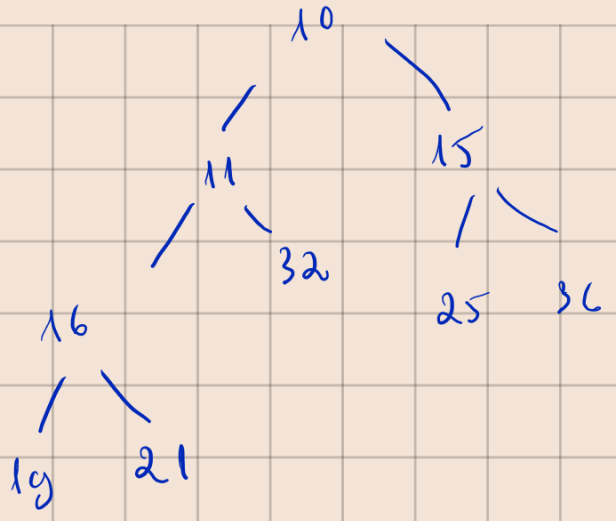


:) Remove () :

- Remove root
- DownHeap: 3 cases for a the right most leaf when it becomes the new root ✓
  - 2 good kids → done
  - 1 good, 1 bad → swap with bad
  - 2 bad → swap with higher order
- stop if { becomes a leaf (has data at index 2i, 2i+1) / good relation ship

:) Run remove() :





-) BuildHeap() : Input : data Arr [n]
  Output : Heap with the data

  Method 1 : add n times ⇒ O(n log n)

  Method 2 :
  For each internal node ↑ in reverse order :
  down Heap (n)

  ⇒ O(n)

  Binary
-) Given a tree, how many swaps maximum to get the binary
  → heap ?

  → Σ heights()

-) Hash Map : store <key, value> pairs

  • Rules for key : must be unique, immutable

  • put key() : add new (key, value) to the Map

if already has the key → replace old value with new value

- getKey()

- putKey() : use a backing Array (get() of arr is O(1))

  the index derived from the hash code of the object

  - what if key is not an int ? → use hashCode()

- hashCode() : 2 properties

  - Mandatory : 2 equals() object has the same hash code

  - Nice to have : 2 diff object has diff hash code

- hashcode → index : abs (hashcode % arr.length)

- collision : trying to put key into an index where there is already a key

  - We can resize the back arr to reduce collision rate
    → resize before full (the proportion we should resize at is called load factor)

:) Quadratic probing : $k + i^2$ for $i = 0, 1, \ldots$

- Increment : when node $k+i^2$ is occupy
- If $k+i^2 >$ len → index = $(k+i^2)$ % len

→ solve prob of lin. probing, but have to resize many times

Better plan :

:) Run Binary search on LinkedList :

- One approach : 2 linked List stacked

→ Still O(n) (no middle ptr for left / right linkedList)

:) SkipList : Stacking several linked lists on top of each other

- Each node contain ref s to : next, previous, above, below
- Node must obey 2 rules :
  - All data must be in lowest level
  - If in levels, must be in all lower levels

(name of creators)

:) AVL : self - balancing BST

:) balance factor := node.left.height − node.right.height

- each node contains height and bf

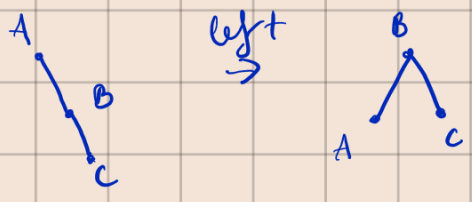
- null node has height - 1

- [-1, 1] : good
  -2 or 2 : unbalanced → not good → have to rebalance

) AVL's add() : same as BST + fix unbalanced nodes

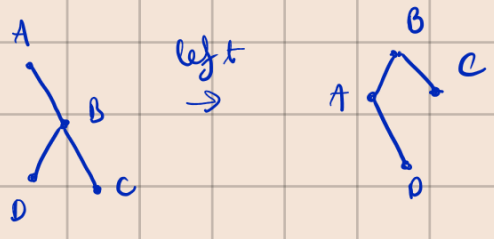
) Fixing unbalance : Rotation the unbalanced node's subtree

- left :

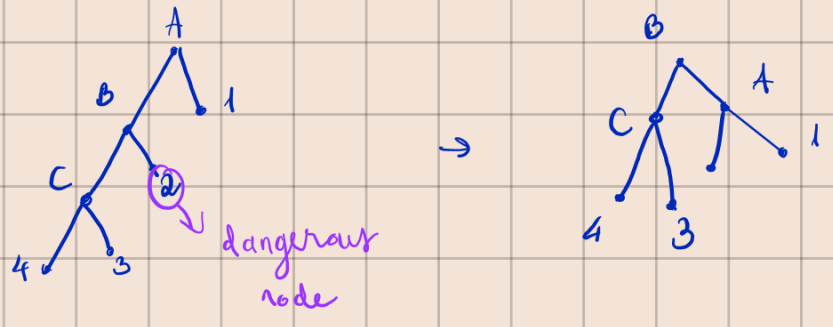


Do when node.bf = -2
node.right.bf = -1 or 0

- Note :

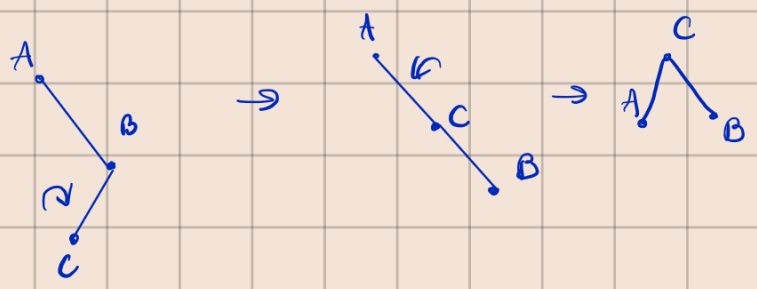


- right :



- Note :

- right - left rotation : node . bf = -2, node . right . bf = 1



- There may be 2 dangerous nodes

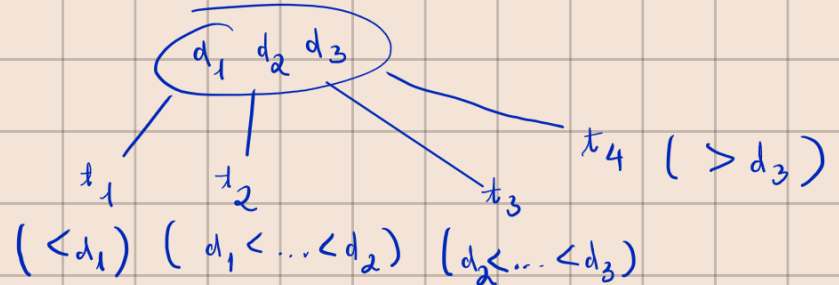
- left - right rotation : node . bf = +2
  node . left . BF = -1

:) 2-4 trees : tree, but each node can have 1, 2, or 3 items, sorted, and

kids = # data + 1

- Shape property : all leaves at the same depth

- order property :

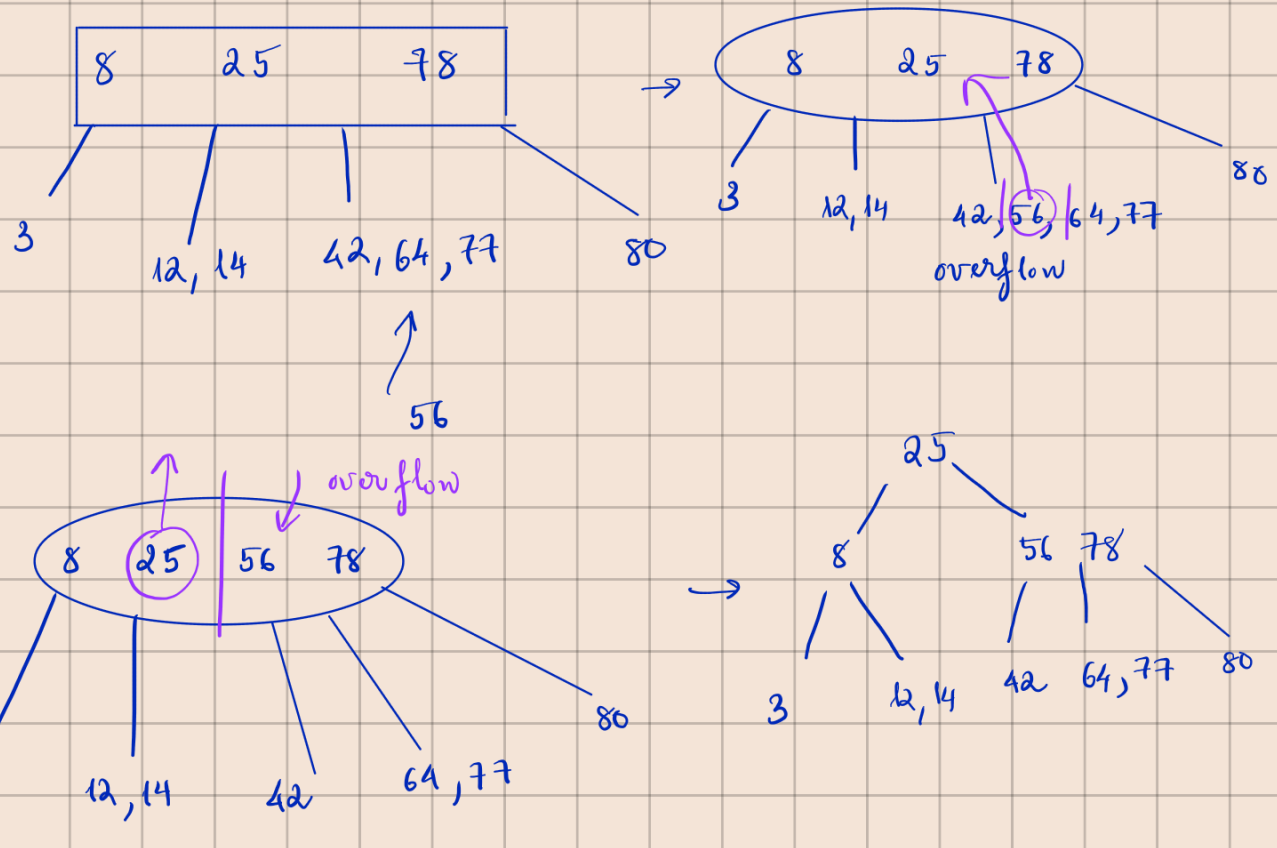


:) 2-4 add : • Use 2-4 contains() to find node
• Add to it (still

• What if the leaf is full (i.e. 3 items)

→ still add, temporarily "overflow" → push 2nd / 3rd data to parent and split the current node to 2 nodes

• What if parent is full ? → do the same thing and we have a new parent



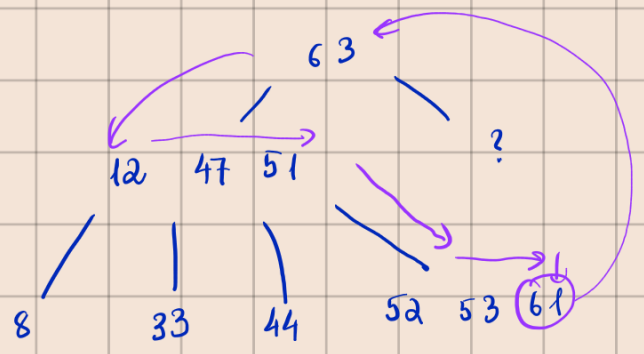
:) 2-4 remove :

Case 1 : remove data from leaf with 2 or more data
→ Just delete

Case 2 : remove from internal nodes

→ replace data with predecessor or successor

- Re / Pre in a leaf with other data

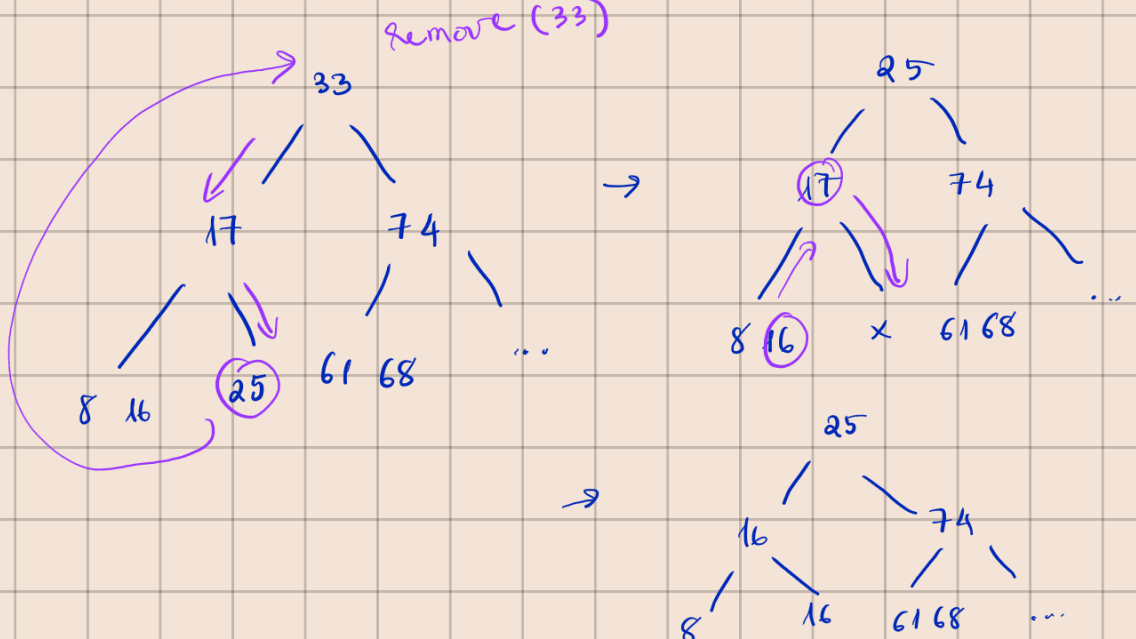


- Re / Pre is the only data in a leaf → Case 3 and 4

- Case 3 : data is in leaf, leaf has only 1 data, immediate sibling (same parent) has 2, 3 data

→ Transfer : parent ↓, sibling data ↑

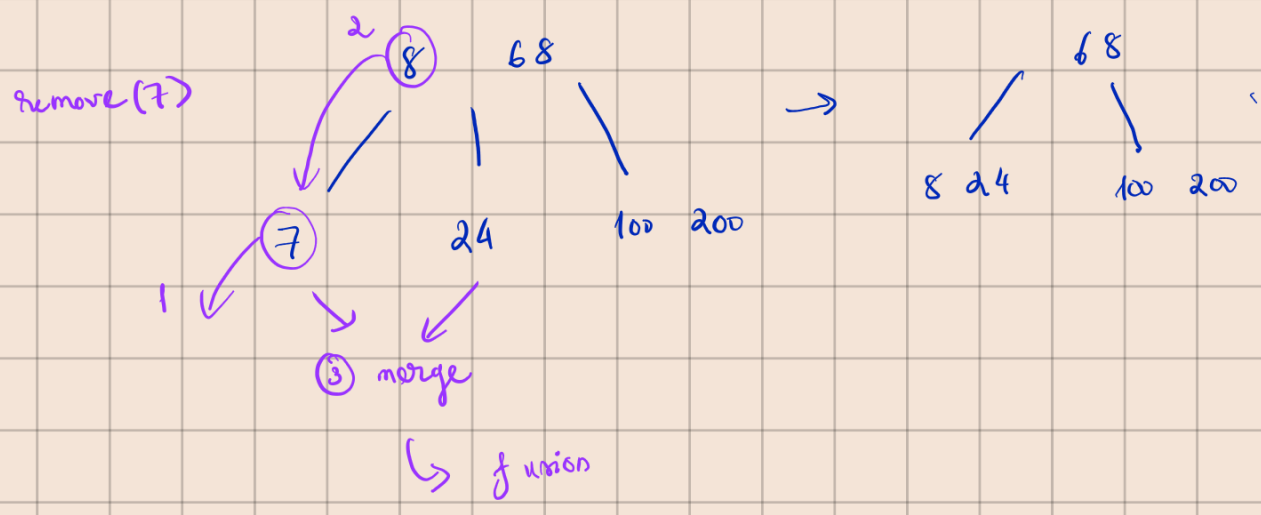
empty ↓ closest sibling



Case 4 : leaf has 1 data, siblings also have 1 data

- "underflow" : empty node, no ability to transfer, because sibling all have 1 data

- We will treat underflow exactly like overflow, but reverse



- overflow : pushing to parent can make it too full

- underflow : pull out parent could make parent empty

if parent is empty

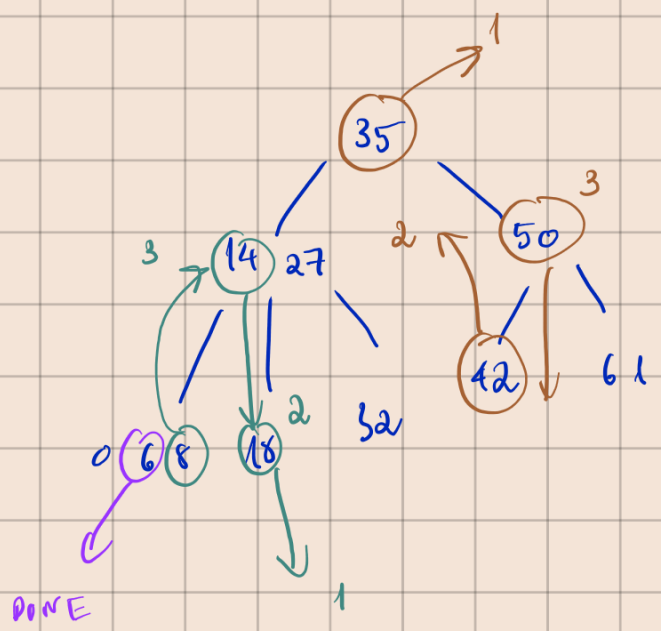
if sibling of parent has spare data → Transfer → done

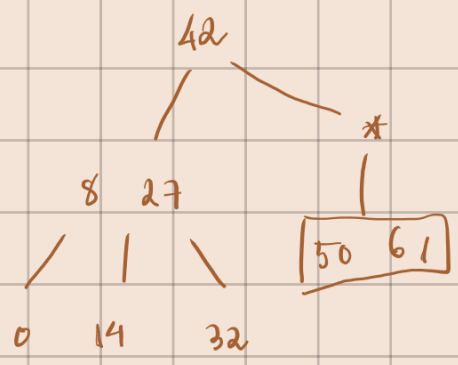
if sibling of parent all have 1 data → fusion again

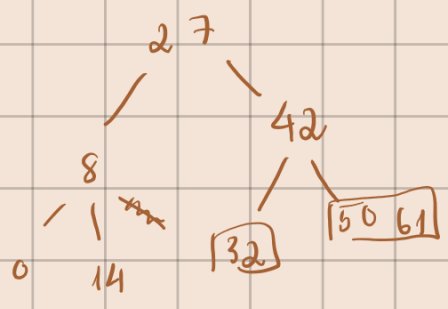
↓

can create more work :

- empty root : remove it


35
1
14
27
3
50
3
2
42
61
3
2
32
0
6
8
18
2
DONE
1


42
*
8
27
50
61
0
14
32


27
8
42
0
14
32
50
61