- Amortized Cost = $\frac{\text{Total cost of operations}}{\#\text{operations}}$

- Pointer: a variable that stores the memory address of an object

- Hard removal: The data removed is completely removed from the backing structure
  - Set a position to null is a hard removal method

- Soft removal: The data is still inside the backing data structure, but cannot be accessed by user

---

DATA STRUCTURE

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

  - 2 standard implementation of list: Array List, Linked List

- Array List: CS 1332 lecture note

- Linked List: 3 different implementations: Singly-Linked List, Doubly-Linked List, Circularly Linked List

- Singly-Linked List: CS 1332 lecture

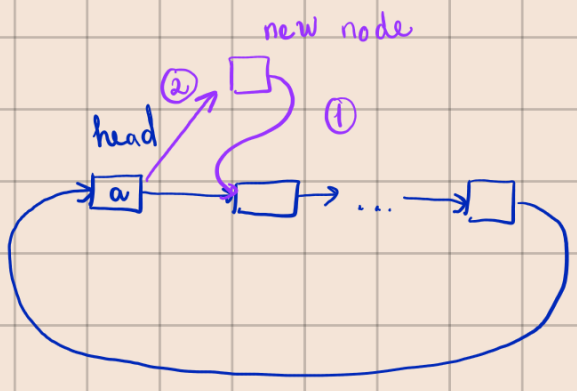
  - Methods that require going to the end of the list takes O(n)
    - AddBack() → tackled using tail pointer
    - RemoveBack() → a problem where it's impossible to assign "tail" to 2nd-to-last node
      → cannot go backwards

- Note on removeLast(): cannot directly set the data of "tail" to null because the node itself still there, meaning the 2nd-to-last node is not pointing to null
  - Instead, have to traverse the list to the 2nd-to-last node, then set its pointer to point to null

) Doubly Linked List: Can do all methods in O(1) (except addAtIndex($\frac{n}{2}$)), but cost memory and more complex

) Circular Singly Linked List: like a singly linked list, but the last node points to the 1st node
- "tail" is not needed

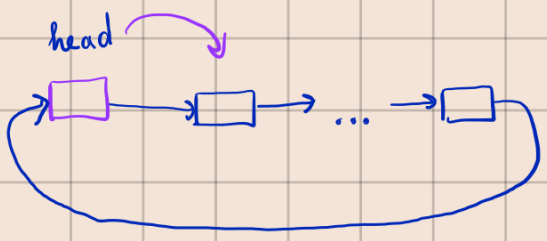
- Add Front():



③ Move old head's data "a" to new node

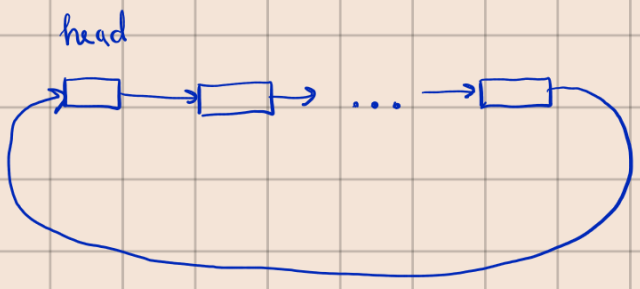
④ Move new head's data to old head

- Add Back():
  1. Add Front (data)
  2. Move Front To Back()



head = head . next

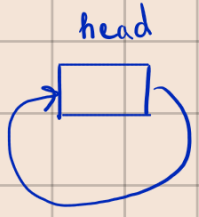
- Remove Front () :



  - Remove current head is hard (because of the "tail" pointer)
  - Remove 2nd node is easy → move the data of 2nd node to head then remove 2nd node

```
head . data = head . next . data
head . next = head . next . next
```

(*) Not worked if CSLL only has 1 node

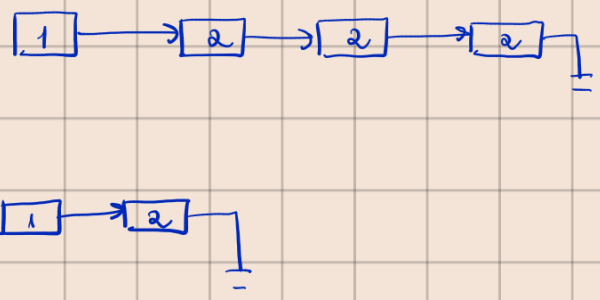


- Remove Back () : iterate to the 2nd-to-last node

(Recursion)

·) Pointer Reinforcement : each node determine whether it remains in the list

Ex : remove duplicates in a Linked List



→


```
1  public void removeDuplicates() {
2      head = rRemove(head);
3  }
4
5  private Node<T> rRemove(Node<T> curr) {
6      if (curr == null) {
7          return null;
8      }
9      curr.next = rRemove(curr.next);
10     if (curr.next != null
11     && curr.data.equals(curr.next.data)) {
12         return curr.next;
13     }
14     return curr;
15 }
```

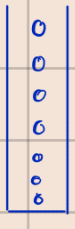
return → return the node that replaces curr in the list

Base case → return the node that replaces curr.next in the list (recursive step)

condition for curr to be in the list

only remove / add from top

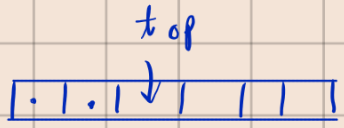
·) Stacks :



- Data at the bottom stay in the stack the longest (Last In First Out - LIFO)
- push () := addTop ()
  pop () := removeTop ()
  peek () := getTop

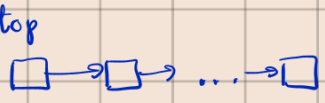
- Problem : don't have addIndex () and remIndex ()

·) Array Stack :



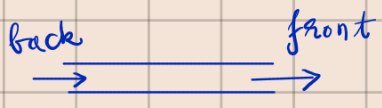
- top : backing ArrayList 's back
- push() := addBack()
- pop() := removeBack()
- peek() := getBack()

·) LinkedStack :



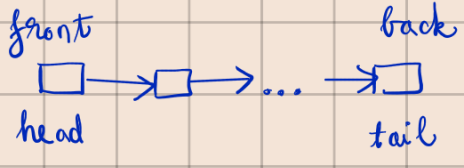
- top : backing LinkedList 's head
- push() := addFront()
- pop() := remFront()
- peek() := getHead()

·) Queue :



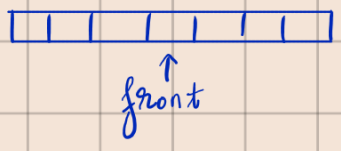
- Features : addToBack(), remFromFront()
  enqueue() dequeue()

·) Linked Queue :



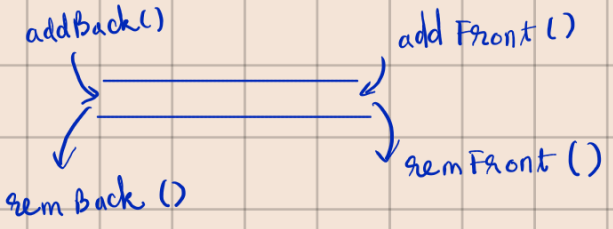
- back : backing LinkedList 's tail
- front : backing LinkedList 's head

⇒ { enqueue() := addBackWithTail()
  { dequeue() := remFront()

- Array Queue : 

(circular Array)

  - Backing DS is circular Array with front index
    - back = (front + size - 1) % backing Arr . length
  - When resize backing Arr, set front back to 0
  - enqueue () := addIndex ((front + size - 1) % backing Arr . length)
    dequeue () :
      - Set front's data to null
      - front := (front + 1) % backing Arr . length

- Deque : 

- LinkedDeque : Have to deal with O(n) method (removeBack()) if using SLL
  → Use DLL as the backing DS

- ArrayDeque : Use circular Array as the backing DS

- Trees : G is a tree if G is loop-free and has no cycles
  - Parent & Child : 

  - A node can have any # of children
  - Exactly 1 node w/ 0 parent

- Leaf : Node that has 0 children

(BT)

- Binary Tree : Each node has at most 2 children : "left" and "right"

  - Full BT : each node has exactly 0 or 2 children

  - Complete BT : satisfy 2 conditions :
    1. Each depth of the tree except the lowest depth must have maximum # nodes
    2. Lowest depth must be filled from left → right

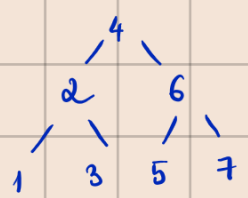
  → For each n, there is exactly one shape a complete tree has

  - Balanced BT : A node is balenad if its children have height that differ by 0 or 1 . A tree is balanced if every node is balanced

- Binary Search Tree (BST) : A binary tree where for a node with data x , all the data in the left subtree $< x$ and all the data in the right subtree $> x$.

- Pre - order traversal : parent → left subtree → right subtree

Ex :

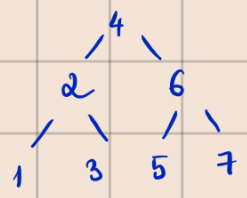


→ (4, 2, 1, 3, 6, 5, 7)

  - Pre-order traversal is unique

- In - order traversal : left subtree → parent → right subtree

Ex :



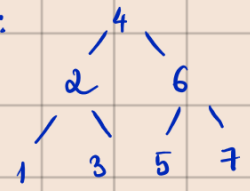
→ (1, 2, 3, 4, 5, 6, 7)

ordered

→ This outputs ordered list of data

- Post-order traversal : left subtree → right subtree → parent

Ex :

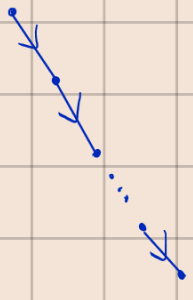


→ (1, 3, 2, 5, 7, 6, 4)

- Tree Operations :

  - contains () :
    - If data > currNode . data : search right
    - If data < currNode . data : search left

Worst case :



→ O(n)

Generally runtime is O(log n)

- add () : Any data can be added to leaf positions

- If data > currNode.data : go right
- If data < currNode.data : go left

```
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 () : There are 3 cases :

- Case 1 : The removed data has no children :
  - Ptr reinforcement :
    if currNode.data == data : return null

- Case 2 : The _____ 1 child :
  - Ptr reinforcement :
    if currNode.data == data : return currNode.nextNode

- Case 3 : The _____ 2 children :
  - Find the node containing the data using ptr reinforcement
  - Delete the node's data
  - Replace with its successor / predecessor
    successor: smallest data > curr.data
    predecessor: largest data < curr.data
  - Remove the S / PS (1 or 0 child this time)

return node replaces curr

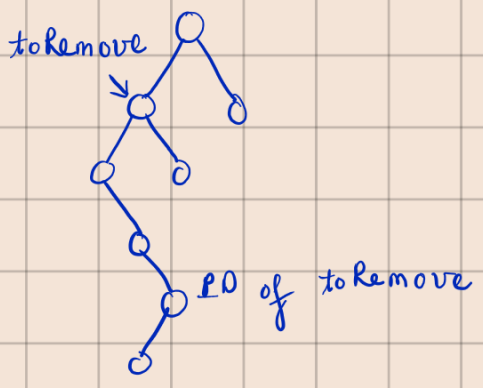
```
public T remove(T data):
  Node dummy = new Node(null)
  root = removeH(root, data, dummy)
  return dummy.data

private Node removeH(Node curr, T data, Node dummy):
  if curr == null:
    //data not in tree
    throw exception
  if curr.data < data:
   //recurse right
  if curr.data > data:
   //recurse left
  if curr.data == data:
    dummy.data = curr.data
    if curr has 0 kids:
      return null
    if curr has 1 kid:
     return curr.kid
    if curr has 2 kids:
      //get predecessor
      Node dummy2 = new Node(null)
      curr.left = getPredecessor(curr.left, dummy2)
      curr.data = dummy2.data

  return curr
```

↑ Left

→ as R as possible

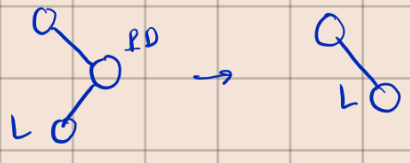


→ return the node replacing curr

```
private Node getPredecessor(Node curr, Node dummy):
  // find the rightmost descendant
  if curr.right == null: → curr is PD
    //found predecessor
    dummy.data = curr.data
    return curr.left
  else:
    curr.right = getPredecessor(curr.right, dummy)
    return curr
```

→ ptr reinforcement as well

→ clever trick to return more than one value



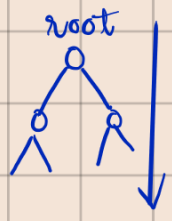
- Base case of getPredecessor(): we want to remove the PD as well, therefore the parent of the PD will point to the left (L) node instead

- getPredecessor() will reconstruct the path it travels through to get to the PD of toRemove.

→ Reading recursion: checking the base case n = 0, then check the case n = 0+1, ..., until you get the concept

) Heap : A tree data structure where for every node, the value of its children is greater or smaller than its own value

- Max Heap : 

decreasing
- Min Heap : 

increasing

- Heap is a complete tree
- order : for a max heap, node with greater value has higher order
  ____ min heap, ____ smaller ____
- parent has higher order than children
- Backing DS : array (because there is no gaps between data)
  - Storing data of a heap into array : use level-order traversal
  - The 0-index element is null

Let A be the backing array

- add () :
  1. A.addBack() → now heap may be out-of-order
  2. Upheap :

```
while newNode has higher order than parent
    swap (newNode, parent)
    if newNode is root :
        break
```

- remove () : remove root (highest order)
  1. remove A[1]
  2. move A[size] to index 1 → newNode

③ Down heap :

while newNode has lower order than it's children

if only 1 bad child :

swap (newNode, bad child)

if 2 bad children :

swap (newNode, bad child with higher order

if newNode is leaf (have 0 children) :

break

- buildHeap() : starting from right → left

for each internal node v in reverse order level :

downHeap (v)

Runtime : O(n)

:) HashMap : A data structure that stores (key, value) pairs

Let M be a hashmap

- Rules : keys must be unique and immutable
- put (key, value) : put (key, value) pair into M, if key already existed, replace old value with new input value
- get (key) : return associated value

We want put() and get() to be O(1) ⇒ hashCode() and backing array

- hashCode() : map data of arbitrary size to a fixed-size value (hash value)
  - Two objects are equal ⇒ they have the same hash value
  - Reference : 2 diff objects have diff hash code
  - index = | object.hashCode() % A.len |

- Collisions : put 2 diff keys into the same spot of backing array
  - good hashCode() can reduce this
  - increase size of backing array

- Load factor : a value indicate when to resize, i.e. when $\frac{\#\text{ items}}{A.\text{len}}$ $\geq$ load factor

- Resize : when doing resize, len changes so indices of keys must be recalculated

- Open addressing : 1 item per index $\longrightarrow$ Linear / Quadratic Probing
- Closed addressing : $\geq$ 2 items per index $\longrightarrow$ External chaining

- External chaining : one index hold a linked list (no back ptr)
  - put() $\rightarrow$ O(n) (adding to back)
  - resize cost : O(n)

- Linear / Quadratic probing : keep track of deleted items (DEL mark)

  - put() : stop when seeing size non-removed entry or meeting removed entry with key-value (*)

    ① Compute id

    ② At id :

    if key already existed : update value
    if diff keys : probe right (id += 1)
    if DEL : remember if it the 1st one
    if null :
    if 1st DEL = null : put key here
    else : put key into 1st DEL

Runtime : Worst case $O(n)$

- get () : (*)
  1. Compute id
  2. At id :
     - if keys are same : return key's value
     - if diff keys : probe right
     - if DEL : probe right
     - if null : stop, no found

Runtime : Worst case $O(n)$

- resize : everything can collide after resize → $O(n^2)$

- Quadratic probing can help reduce runtime by creating gaps, but this causes searching for empty space to put items in inefficient

- What do we do when we cannot find an empty space? resize (not once)
- Why the max # attempts is $\frac{arr.len}{2}$? After that the indices will start repeating

resize once may not be enough

) SkipList : 2D linked list, where a node has 4 ptrs : next, previous, above, below

- Rules :
  1. All data must be in lowest level
  2. If in level, must be lower levels (no data can fall between 2 empty levels)

- contains () :
  - if data > curr's data : go right
  - else if data < _____ : go down

if hit null : not in list

- add () : most leftward item is $-\infty$, most rightful item is ∞
  1. Flip coin
  2. if head : # layers += 1 (add on top)
     if tail : start adding
  3. Adding : if data > curr's data : go right
     if data < curr's data :
     add before curr
     go up
     continue until fall out of the list

- remove () : similar to contains (), but after found and remove the element at the current level, go down and repeat until fall out of list

Runtime :

- for add () : random
- for contains () :
  - Best case is similar to BST → $O(\log n)$
  - Worst case : all data is in a single linked list ↑ horizontal (all HEADS) ↓ vertical (all TAILS)

  ⇒ $O(n)$
- for removes() : $O(n)$

) AVL : self-balancing binary search tree

- Balance : A node is balanced if it children have heights diff by 0 or 1. A tree containing all balanced nodes is balanced

- null node has height -1

BF

- Balance factor := node.left.height − node.right.height
  - BF ∈ {−2, −1, 0, 1, 2}
  - Ex: 

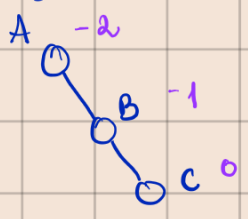
root.bf = 2 → lean too much to the left

- Fixing unbalanced subtree: Let A, B, C form a subtree, we have 4 cases:

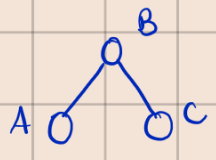
① Rotate left:

A.BF = −2

A.right.BF = −1

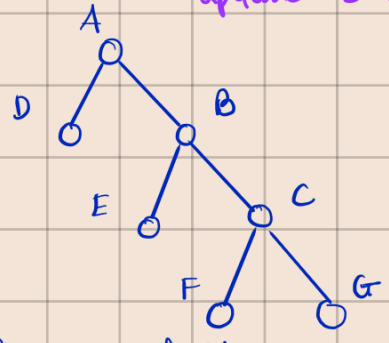


Rotate left:

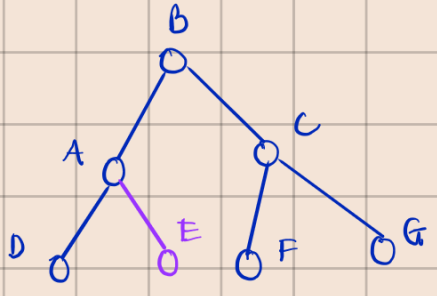


Notice that height and BF of C doesn't change → only need to update B and A

What if B already had 2 children?



→

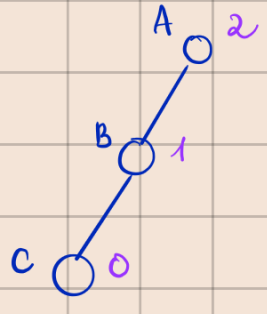


: attach left child to old root as right child

tip when coding: create temp node point to B to keep track of what we want to return

- What node need to update? A and B

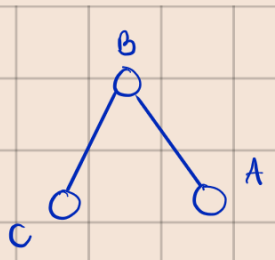
② Rotate right:



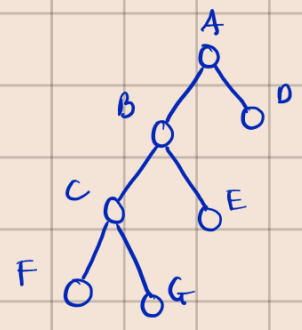
A.BF = 2

A.left.BF = 1

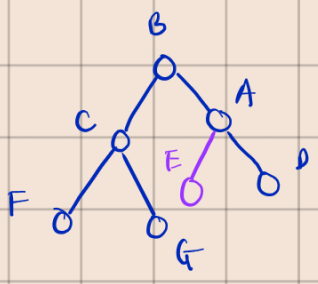
Rotate right :



When B has 2 children :

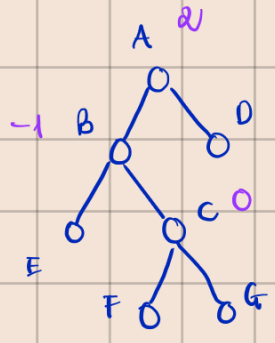


→



: attach right child to old root as left child

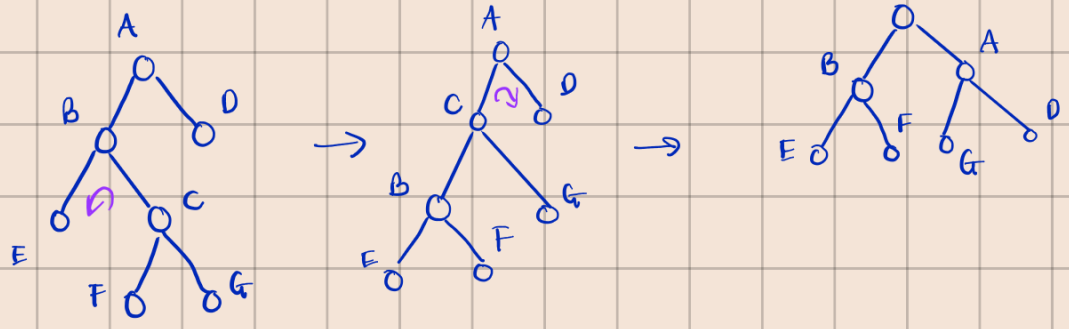
③ Rotate left - right



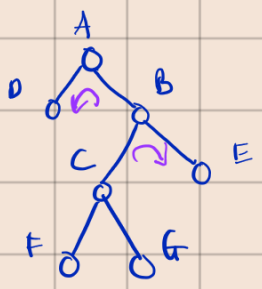
A . BF = 2

A . left . BF = -1

Rotate left-right : rotate left for sub-tree rooted with B, then rotate right for sub tree rooted at A



④ Rotate right - left :



- Add() :
  1. BST. Add ()
  2. Update height (from newly add node → root)) can be grouped together as update()
  3. Update BF
  4. Fix unbalanced subtree
     - Since the method to fix unbalance subtree it used together with ptr reinforcement, it is more convenient to design the method to return the rebalanced root

       Node rebalance (Node parent)

     - Where to put it ? When we want to rebalance a subtree rooted at parent where BF of parent = ± 2
       → It makes sure that the children and grandchildren node are balanced
       → Makes sure that parent's BF is always ± 2

- Remove() :
  1. BST. Remove() ↗ need to remove the successor / predecessor, so also need to update height / BF here
  2. Update height (from newly add node → root)
  3. Update BF
  4. Fix unbalanced node

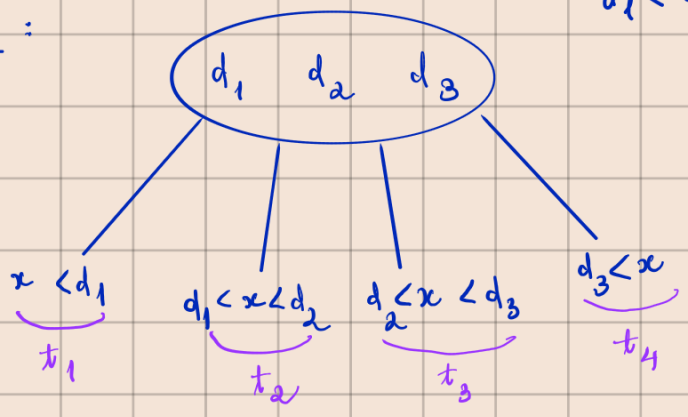
Runtime : all O(log n)

- 2-4 Tree : A tree where each node has 1, 2, or 3 data

  - # kids = # data + 1
  - Shape property : all leaves are at the same depth

  - Order property :

$d_1 < d_2 < d_3$



  - Contains () :

    if data $< d_1$: search t_1
    else if $d_1 <$ data $< d_2$: search t_2
    else if $d_2 <$ data $< d_3$: search t_3
    else if $d_3 <$ data : search t_4

- Add () :

  ① Use contain () to find leaf node
  ② Add to existing node

  Case : leaf node is already full, adding to it causes "overflow"
  → still add, then push 2nd / 3rd data to parent → promotion
  → Split leaf node into 2 new nodes

  - What if parent is overflow ? continuing do promotion for parent node

- Remove () :

① Use contain() to find node containing the data

Case 1 : data in leaf node with ≥ 2 data
→ Just remove

Case 2 : data in leaf node with only 1 data
→ Look into sibling nodes to see if they have ≥ 2 data
- If yes, then after removing the data in leaf node, move parent node to the now-empty leaf node, and push sibling data to the place left over by parent (technically rotation)

- If no, then we encounter "underflow"
→ Pull data from parent down to now-empty node, then merge with sibling → fusion

What if parent is empty? continuing to fusion in parent node

Case 3 : data in internal node and successor/predecessor of data have ≥ 2 data
→ Replace data with successor / predecessor
→ Remove the succ / pred (Guarantee to be in leaf node)
→ The rest is case 1

Case 4 : data in internal node and succ/pred of data is alone in its node
→ Replace data w/ succ / pred
→ Remove the succ / pred the same way in case 2

- Runtime :
  - Add() : contain() : $O(\log n)$
    overflows : at most $\log n$ overflow → $O(\log n)$

$\Rightarrow O(\log n)$

- **Remove()** : contain() : $O(\log n)$
  transfer : at most once
  underflow : at most $\log n$ underflows $\rightarrow O(\log n)$

- **Contain()** : $O(\log n)$

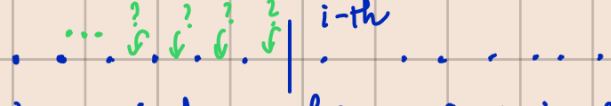
ALGORITHM

- Sorting : there are 3 criteria for sorting algos :
  - In-place : Doesn't need additional data structure to sort
  - Adaptability : If data is partial sorted, then runtime improves
  - Stability : Items with the same value will stay in the same relative order after sorting

- In-place and Adaptability are easy to detect
- For Stability : if swap only adjacent elements → probably stable
  if ____ further apart → probably not stable

- Bubble Sort : For each pair of adjacent items, if out of order, swap
  - After i-th iteration, the item in index size - i will be correct
  - Optimization :
    - Keep track of the index of the last item that got swapped → k
    → For each following iteration, stop at k (indices from k → end are correct)
    - If no swaps made during an iteration → finish
  - In-place, Adapt, and Stable
  - Runtime : $O(n^2)$

- Insertion Sort : In i-th iteration, put i-th element into the sorted portion of the list



  - At i-th iteration, indices from 0 → i are sorted
  - Runtime : $O(n^2)$
  - Stable, Adaptive, In-place

- Selection Sort : In i-th iteration, put the i-th largest element to correct position
  - Actually the one that has the fewest # swaps
  - Not stable, Not adaptive, In-place

- CocktailShaker Sort : Run bubble sort from left → right then right to left
  - The next iteration of BB starts at the element right after the last swap position of current BB
  - If no swap in one iteration → Finish

  - Stable, Adaptive, In-place
  - Runtime : $O(n^2)$

- Best iterative sorts : Insertion Sort (fewest compares)

- HeapSort : Add all data to heap, then remove item by item
  - Note : Use BuildHeap() to add data
  - Not Stable, not adaptive, not In-place
  - Runtime : $O(n \log n)$

- MergeSort : Recursively sort left half, and then right-half, and then merge left and right halves

  - Note : • left = data [0 : size/2] (right-exclusive)
    right = data [size/2 : size]
    • For merge, create a new temp array to hold the sorted elements, then copy them into the original portion

  - Stable, not adaptive, not In-place

:) Fred Sort : break data into lists of size $\sqrt{n}$, then run insertion sort on each list, then merge them into a single sorted list

- Stable, not Adaptive, not In-place
- Runtime : - Insertion sorts : $O(n)$
  - Add 1 element to temp & need to compare with $\sqrt{n}$ others (# lists)
    $\rightarrow O(n\sqrt{n})$

  $\Rightarrow O(n\sqrt{n})$

:) Quick Sort : recursively partition the left sub-array and right subarray according to a pivot chosen at random

$$< \text{pivot} \mid \text{pivot} \mid > \text{pivot}$$

- Partition () :
  1. Swap (pivot, A[start])
  2. Let $i = 1$, j = size -1
     - Find i s.t $A[i] >$ pivot from index 1 $\rightarrow$ end
     - Find j s.t $A[j] <$ pivot from index end $\rightarrow$ 1
     - Swap ($A[i]$, $A[j]$)
     - Repeat until $j < i$ (j moves to the region containing elements $<$ pivot)
  3. Swap (pivot, $A[j]$)

- Not stable, not adaptive, in-place
- Runtime : average $O(n \log n)$
  worst $O(n^2)$ (keep choosing pivot which is the min / max element)

:) Quick Select : Similar to Quick Sort, but only need to work on one side
$\rightarrow O(n)$ runtime

- Radix Sort : Sort integers by 10's places (each place can only hold digit from $0 \to 9$, except the largest place which may include negative digits , so digit from $-9 \to 9$

  - # iterations : # digits of the longest int = k

  - Represent each digit by a Bucket (linkedList) containing the integers that has that digit at the current 10's place.

  - Runtime : $O(kn)$

  - Stable , not adaptive, not in-place

- Pattern matching : given a string text and string pattern, find all occurrences of pattern in text

- Booyer Moore :

  - Last Occurrence Table : mapping each char in the pattern to the last index it appears in the pattern , and all other chars got mapped to -1 (Use hashmap)

  - Searching algorithm :

    ① Starting from the end of the pattern, compare char-by-char with the crpd portion of the text

    ② If every char matches : add the start index of the crpd portion of the text to result , then move start forward 1 position

    ③ Else : Let failtable be the failure table

Let k be the index of the char of text that fails

lastId
!!
If failtable.getKey(text[k]) < k :
    start += pattern.length - lastId

Else : start += 1

) KMP : relies on prefix & suffix of the pattern to determine shifting

- Failure Table : record the length of the longest prefix that is also the suffix of the pattern [0 ... i] (right inclusive)
  - Algorithm for creating FT :

FT[0] = 0

let i := the starting index of prefix
    j := _____ suffix

i j (at both heads)
↓ ↓
i = 0 , j = 1 (the substring is $c_1 c_2$)

While j < pat.length :
    If pat[i] == pat[j] : write i+1 to j (i+1 = length of prefix)
                          i++, j++

Else :
    If i ≠ 0 : i = FT[i-1]
    If i = 0 : FT[j] = 0 , j++

(i, j now at both heads again and pat[i] ≠ pat[j])

·) Searching algorithm: Acts like normal Brute Force, except:

- Comparing go from left → right of pattern
- When a mismatch occur at index k of pattern

  start += k - FT[k-1]

  For next comparing iteration, start at id FT[k-1] of pattern

- If all chars match, treat the next char in text at mismatch and shifts like normal

- If mismatch right at start: start ++ (shift 1 pos)

·) Rabin-Karp: Compare the whole pattern with the crpd portion of text using hash

- Searching algorithm:

  If hash(Pat) == hash(Por): (equal hashes doesn't mean equal objects)

  → Compare Pat and Por char by char

  Else: shift pat by 1, hash of new portion computed using rolling hash

  pattern = $\overline{c_n \dots c_0}$

- Coding Tips: $hash(pat) = \sum_{i=0}^{n} c_i \cdot base^i$

  ① Have to compute hash of $\overline{c_0 \dots c_l}$ where l = pat.length manually (can't use rolling hash)

  - hashFirst() return hash of $\overline{c_0 \dots c_l}$ and $base^n$ (don't need to use Math.exp())

② Rolling hash : let $h = hash(\overline{c_i \dots c_j})$

$hash(\overline{c_{i+1} \dots c_{j+1}}) = (hash(\overline{c_i \dots c_j}) - c_i * base^n) * base + c_{j+1}$

③ Stop when $i == end - l + 1$

:) Galil Rule : A rule that can optimize Bayer Moore

Ex : text = a a a a a a a ... a (length n)

pattern = a a a ... a (length m) $(m < n)$

=) Runtime = $O(mn)$ (just like Brute Force)

- We can make pattern become g g g ... g where g = a ... a where g.length = pat.length - FT[end]

- When we get a complete match between pat and text, treat this like the way KMP does : shift by period (= pat.length - FT[end], next char in text seen as mismatch) and start to check at id pat.length - period (= FT[end])

a b a b a b

Ex FT:

0	1	2	3	4	5
0	0	1	2	3	4

period = ab

period.length = 2

:) Depth - First Search (DFS) : An algorithm to traverse the graph, where the visited vertices are marked to avoid later. The order of visits is similar to the process of build up and tear down a stack

- Runtime : $O(|V| + |E|)$
- Space-complex : $O(\log n)$

- Breadth - First Search (BFS): Traverse the graph layer by layer, where visited are marked to avoid later. The order of visits is similar to the process of build up and tear down a queue. Stop when the queue is empty.

  - Runtime: $O(|V| + |E|)$
  - Space complex: $O(n/2)$

- Dijkstra: Find the shortest path from a node to all other nodes in a weight graph. Use priority queue to keep track of the shortest path. Vertices returned by priority queue's remove() has value of min distance to the starting node and thus marked as visited. Stop when the queue is empty or all vertices are visited.

  - Runtime: $O((V+E)\log V)$

- Prim: Find the MST by using a set of visited vertices S containing vertices reached by the chosen edges. Use priority queue to keep track of the shortest edge that incident to the vertices of S (to be added). Stop when all edges visited or the queue is empty

  - Runtime: $O(|E|\log|E|)$

- Kruskal: Find the MST by picking the shortest edges possible throughout the graph that don't form a cycle. Use union-find data structure to keep track of cycles.

  - Runtime: union() / find() $= O(\alpha(V)) \approx O(1)$ (inverse Ackermann func)

  $\Rightarrow O(|E|\log|E|)$