Data Structures and Algorithms

- Lini Thomas + Kshitij Gajjar + Venkatesh Chopella

[redacted]

03/01/25

→ Sorting:

1) insertion sort: make room for the new element, then insert it
i.e. elements get shifted one-by-one till we find the proper position for required element

[redacted]

✦ H.W.
1. Code Insertion Sort
2. Code Binary Search
3. Bubble Sort
4. Selection Sort

```
int arr[n]
int temp;
pseudocode: for (i=1; i<n; i++){
              temp = arr[i];
              for (j=i-1; j>=0; j--){
                if (temp < arr[j]){
                  arr[j+1] = arr[j];
                } else {
                  arr[j+1] = temp;
                }
              }
            }
```

complexity: $O(n^2)$ as in worst case, we are running one n-sized loop, and for every element in that n-sized loop, we run another n-sized loop [redacted]

It is $O(n)$ in the best case i.e. an already sorted array, as the second n-sized loop doesn't need to run at all.

• Binary search

→ jump to the middle element of a sorted array. check if that number is less than or greater than required element:

a b c d e

search this if c > required

else search this

∴ number of steps halved each time.

i.e. complexity = $O(\log_2 n)$.

↳ because $1/2$

→ This is better than scanning the array fully, which is an $O(n)$ process.

→ Q. Can Binary Search help us with insertion sorting?

A. No, even though finding the location will be faster, we still have to shift the element there.

i.e. insertion sort using binary search

$O(n \times n\log n)$

↓ shifting + placing

finding position × n values of i-loop

Q. An inversion: if $i<j$ and $a[i] > a[j]$., $(a[i], a[j])$ → inversion. If there are $f(n)$ inversions, what is the relation between inversions and insertion sort?

A. number of steps in an insertion sort = number of inversions in the permutation.

At any point, we only need to focus on the inversions of the number that lie before the number, as the inversions after the number are covered by the numbers after it

||

we check for all the numbers greater than current i value (to shift it).

∴ checking for inversions = checking for insertion sort.

∴ number is the same

ii) Bubble sort : swapping consecutive pairs of elements from the first pair (0^{th}-1^{st} element), moving till the $(n-i)^{th}$ position in each iteration i.e there are n-1 iterations.

Best case complexity is O(n), where the array is already sorted, and further iterations can be stopped.

iii) Selection sort : select the maximum number, put it in the last place of the current iteration : $(n-i)^{th}$ position, i - iteration number, and placing the number currently at last position to the original position of the maximum.

(or, find smallest & put in first position)

contains minimum swaps.

Q. Let A be an array containing integer values. dist(A) = minimum no. of swaps required to sort array.

dist (2, 5, 3, 1, 4, 2, 6)

A. 2 5 3 1 4 2 6

2 2 3 1 4 5 6

2 2 1 3 4 5 6 = 4.

2 1 2 3 4 5 6

1 2 2 3 4 5 6

Q. Which sort uses least amt. of comparisons to sort:

23 32 45 69 72 73 89 97

A. Insertion or Bubble sort

Q. number of comparisons to find an element in a list of n (distincts) elements that is neither minimum nor maximum?

A. Look at first 3 elements, pick the middle valued one.

Q. Best Algorithm to find number of leaders (numbers that are greater than all numbers to its right).

A. Move from right to left, find no. of (new) maximums.

Q. Best algo to shift all negative elements to the left & all positive elements to the right contains how many iterations?

07/01/25

→ Merge Sort: merging two sorted lists into one (divide and conquer algorithm)
↳ minimum time complexity of any sorting algo: nlogn

Ex: 99 6 86 15 58 35 86 4 0

99 6 86 15 | 58 35 86 4 0

99 6 | 86 15 | 58 35 | 86 4 0

99 | 6 | 86 | 15 | 58 | 35 | 86 | 4 0

99 | 6 | 86 | 15 | 58 | 35 | 86 | 4 | 0

→ after getting multiple 1-element (i.e. sorted) arrays, we can merge sort them to get the final sorted array.

Q. 56, 5, 7, 2, 99, 17, 3, 9, 85

A.

56, 5, 7, 2 99, 17, 3, 9, 85

56, 5 7, 2 99, 17 3, 9, 85

56 5 7 2 99 17 3 9, 85

56 5 7 2 99 17 3 9 85

5, 56 2, 7 17, 99 3, 9, 85

2, 5, 7, 56 3, 9, 17, [illegible] 85, 99

2, 3, 5, 7, 9, 17, 56, [illegible] 85, 99

2, 3, 5, 7, 9, 17, 56, 85, 99

unsorted array

↓ split

obtained 1-element arrays

↓ merge (while sorting)

sorted array

both arrays done simultaneously. *technically incorrect flow*

we merge back what we had split.

i.e. 3, 9, 85

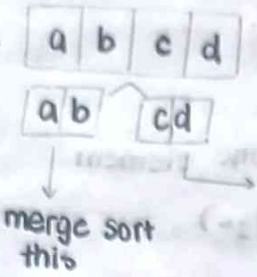
3 | 9, 85

9 | 85

9, 85

3, 9, 85

→ recursive algorithm: for every new array created due to splitting, merge sort is called on that new array.

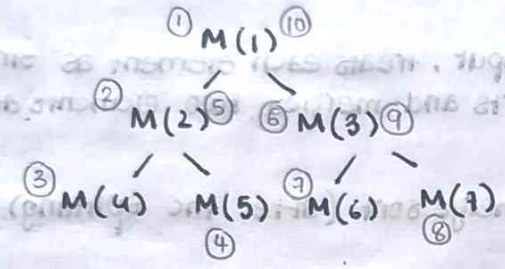
i.e. : 

3. merge the sorted halves together

• Two-way merge efficiency compared to merge sort?
• Prove that merge sort by splitting >2 is more costly

merge sort → splitting of array → merge sort of new arrays
↑ recursion

→ flow/sequence of recursion



→ while merge-sorting, we can either create a temporary array for each step, which takes up a lot of space. or we can use the same array repeatedly

i.e. take array of size n, and fill its positions with the appropriate values.

this is called double storage merging.

→ complexity of merge-sort algorithm: there is no difference in no. of steps b/w the best and worst case ∴ complexity is $n \log n$ in any case.

↳ Time to sort n elements = $T(n)$

$$T(n) = T(n/2) + T(n/2) + cn$$

- $T(n/2)$: Time to sort first half
- $T(n/2)$: Time to sort second half
- cn: some time to merge the sorted halves. $\propto n$, because merging is $O(n)$ process.

$$= T(n/4) + T(n/4) + c \cdot n/2$$

$$\vdots$$

$$T(n) = 2^k T\left(n/2^k\right) + kc \cdot n \text{ in the } k^{th} \text{ step}$$

we know the process goes on until n cannot be divided anymore

$\therefore \; n/2^k \leq 1$

$n \leq 2^k$

$k \geq \log_2 n$

$\therefore \; \therefore k = \log_2 n$ # where arrays all have only one element

(i.e. $T(1)$ $T(n/2^k)$

$\therefore n/2^k = 1$)

$\therefore T(n) = 2^{\log_2 n} T\left(n/2^{\log_2 n}\right) + c \cdot \log_2 n \cdot n$

$T(n) = n\,T(1) + cn\log_2 n$

$T(1) \to 0$

$\therefore T(n) = n \log n \quad (c=1, \text{let.})$

→ two-way merge: takes array as input, treats each element as single element array, and sorts and merges two elements at a time

i.e. bottom → top of merge sort (after the splitting)

Q. What if we split into >2 parts?

A. It is possible to sort the array this way, and the no. of steps will be less, but there are more complex operations (comparison) etc. that are now required.

Thus, it is more costly.

→ Binary vs ternary search

$T_B(n) = T_B(n/2) + 2c$ $\qquad$ $T_t(n) = T_t(n/3) + 4c$

$\vdots$ $\qquad$ $\vdots$

$T_B(n) = 2c\log_2 n$ $\qquad$ $T_t(n) = 4c\log_3 n$

$2c\log_2 n < 4c\log_3 n$

$\therefore$ Binary search is less costly + more efficient than ternary search.

Q. $n=64$, $T(n)=30s \Rightarrow n$ for $T(n)=360s$?

A. $T(n) = c \cdot n \cdot \log n$

$5\ 30 = c \cdot 64 \cdot \log 64 = c \cdot 64 \times 6$

$c = 5/64$

$\therefore 360 = \frac{5}{64} \times n \times \log_2 n$

i.e. $n \times \log_2 n = 72 \times 64$

$= 9 \times 512$

$\therefore n = 512$

$\log_2 n = 9$

$\therefore$ max. input size = 512 elements

Q. 20, 47, 15, 8, 9, 4, 40, 30, 12, 17. straight two-way merge sort (ignore the word straight)

A. P1) 20, 47, 8, 15, 4, 9, 30, 40, 12, 17

P2) 8, 15, 20, 47, 4, 9, 30, 40, 12, 17

$\therefore$ 8, 15, 20, 47, 4, 9, 30, 40, 12, 17.

10/01/24

- Quicksort Algorithm

→ Given an array of n elements,

- if 1 element, return
- if >1 elements, choose a pivot element, then split array into:
  - elements ≤ pivot
  - elements > pivot
- Quicksort the new arrays
- Return results.

Ex. 40 20 10 80 60 50 7 30 100

pointer 1 (next elem) ↓ pointer 2 (last element) ↓

↓ Select first num as pivot, place it in the position where

nums <40 | 40 | nums >40

pointer 1 moves rightwards as long as it points to a value <40, and stops when it reaches a number >40

pointer 2 moves leftward similarly till we reach <40

then, values pointed by 1 and 2 are swapped, and we continue.

we stop moving the pointers once they cross each. and swap values of pivot and pointer 2

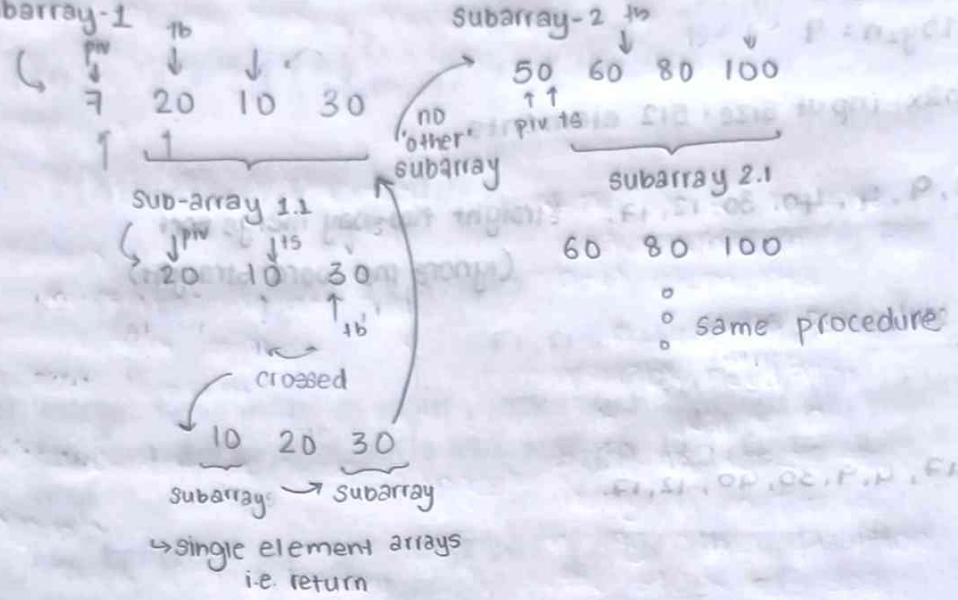
↳ 40 20 10 80 60 50 7 30 100

40 20 10 30 60 50 7 80 100

40 20 10 30 7 50 60 80 100

pointers crossed

7 20 10 30 40 50 60 80 100



= 7 10 20 30 40 50 60 80 100

→ what will the time complexity be if we somehow manage to choose the middle element as the pivot?

$T(n) = \underbrace{T(n/2)}_{\text{Subarray 1}} + \underbrace{T(n/2)}_{\text{Subarray 2}} + \underbrace{cn}_{\text{swaps}}$

$T(n) = n\log n$ in this case

☆ $T(n) = 2T(n/2) + cn$

$\Rightarrow T(n) = n\log n$

if element was first / last:

$T(n) = T(n-1) + cn$ (+ $T(0)$, which is $= 0$)

$T(n) = T(n-2) + c(n-1) + cn$

$T(n) = T(1) + c(2+3+4 \cdots n)$

$T(n) = T(0) + c(1+2+3 \cdots n)$

$T(n) = \frac{n(n-1)}{2} = \frac{n^2-n}{2} \approx \frac{n^2}{2} \approx n^2$ ($c' = c/2$)

$n^2/2$ is equivalent to n^2

since $-n/2 < n^2/2$, we can ignore it

→ the time complexity of quicksort tends to n^2, and the more sorted the array, the more time complexity it has.

→ To improve pivot selection, we choose pivot as the median of data[0], data[$n/2$] and data[n-1].

Q. Possibility that pivot gets placed in worst possible location in first round of partitioning when sorting 25 elements using quicksort.

A. worst possible location = first element / last element

$= 2/25 = 0.08$

Q. If input of sorting is already sorted:

1. Quicksort runs in $O(n^2)$ time
2. Bubblesort runs in $O(n)$ time ⇒ only one iteration, we break if no swaps made.
   or $O(n^2)$ time if not optimised ↳ optimised bubble sort
3. Mergesort runs in $O(n \log n)$ time. ⇒ mergesort is always nlogn
4. Insertion sort runs in $O(n)$ time

Q. How many swaps in selection sort in worst case?

A. n swaps.

Q. Given an unsorted array of integers, give the maximum perimeter. (expected: O(nlogn))

A. 1. Sort it using merge sort

2. Go through from the last element, finding first element s.t. the sum of the elements after it > element itself

i.e. [33, 6, 20, 1, 8, 12, 5, 55, 4, 9]

↓ merge sort

[1, 4, 5, 6, 8, 9, 12, 20, 33, 55]

55 > 20 + 33

33 > 12 + 20

20 < 12 + 9

∴ 20, 12, 9

∴ perimeter = 20 + 12 + 9 = 41

Thus, we ensure maximum + that sides actually form a triangle.

Since 1 is O(nlogn), 2 is O(n), overall complexity is O(nlogn).

✦ Q. Given an unsorted array of integers, O(n) Sort the array in a wave array i.e. $a1 \leq a2 \geq a3 \leq a4 \ldots$

———— X ————

17/01/25

• Stacks

→ called an abstract data structure: it is only concerned with executing the operation, not how it is executing.

→ push: adding elements to stack → you cannot push to a full stack
pop: removing elements from stack → you cannot pop an empty stack

→ implementing operations via arrays:

1) constructing the array + flag variable
↳ i.e. set myTop = -1

2) empty():
check if flag variable = -1

3) Push():
check if array not full
increment myTop
add element
↳ else: output "out of space"

4) pop():
check if array empty
print current element
decrement myTop.
↳ else: output "nothing to pop"

* The stack can have space while array has (n-1) elements:
because the flag variable need not be equal to n-1 i.e. be at the top of the array. If myTop is at any other position, the stack has space even if the array is technically filled.

⇒ what happens if you run out of space in the array i.e. initial array taken is too small? → not an issue if we use linked lists.

option i) everytime you run out of space, add a fixed 'c' amount of space

option ii) everytime you run out of space, double the current amount of space.

→ it takes 'nc' operations in the n^{th} step to execute this → includes operations req. for:
1) creating new array
2) copying old elements
3) filling new elements

∴ $\frac{c(n)(n+1)}{2}$ operations in total i.e. $O(n^2)$ time complexity.

→ it takes $c \cdot 2^{(n-1)}$ operations in the n^{th} step

∴ $O(n \log n)$ time complexity

∴ option ii is better.

→ Applications:

1) Balancing paranthesis:

Ex: ([])({[{}]{}}). suggest an algorithm that determines whether the brackets are balanced or not.

→ iterate through, everytime you see a new character:

i) if it is an open bracket, push it to the top of the stack

ii) if it is a closed bracket, check the top of the stack for the corresponding open bracket.

if you find the correct bracket in the stack, pop it
else, return an error

iii) after iterating through the entire string, if there are any elements remaining in the stack, return an error.

2) converting infix → postfix

a+b → ab+

It is easier to write algo that tells a computer to solve postfix notation.

Ex) a) (a+b-c)*d-(e+f) = ab+c-d*ef+-

b) 4^2*3-3+8/4/(1+1) = 42^3*3-84÷11+÷+

c) a+b/c*(d+e)-f = abc÷de+*+f-

→ iterate through, and:

i) if you see operand, append to end of output list

ii) if you see '(', push it onto the stack

iii) if you see ')', pop the stack till you have popped the left paranthesis ('('). Append the operators popped to the end of the output list.

iv) if you see an operator, push it on the stack. Before that, append any higher/precedence operators to the end of the output list, or equal and pop them from the stack.

Precedence:

^ > */÷ > +/-

At the end of the iteration, any elements still on the stack can be popped and appended to the output list.

Ex) b) 4^2*3-3+8/4/(1+1) = 42^3*3-84÷11+÷+

d) A^{b^c} = A^b^c → acc to above method, Ab^c^

→ actual answer: Abc^^

✦exception:

∴ in the case of exponentials, only pop things of higher precedence (NOT higher and equal) before placing on stack.

→ to solve postfix notation using stack, push operands on stack, and when you encounter an operator, pop the previous two elements on the stack, apply the operator, and push the result back onto the stack. iterate through the entire string

Once we iterate through the entire string, popping the final element gives the final result

X

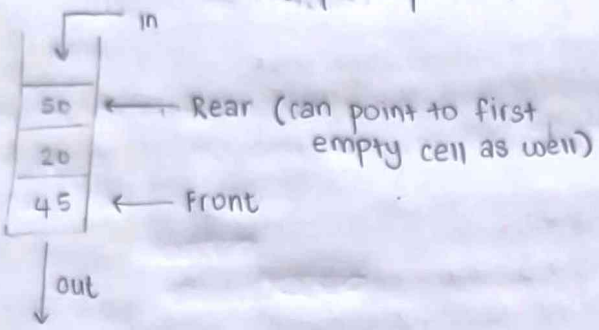
21/01/24

• Queues

→ Enqueue: entering the queue
Dequeue: leaving the queue

✦ Implement stacks and queues

→ First-in, first out principle is followed



As with stacks, queues can have space even if the array it uses is full.

Pointers shld be able to shift from the last to the first element

→ For a linear queue,

i) Enqueue:

```
enqueue (queue){
  if (R==(n-1)){        // length of array
    // full
  } else {
    if (F==-1){
      F++;
    }
    R++;
    queue (R) = a;
}
```

ii) Dequeue:

```
dequeue (queue){
  if (F==R+1 || F==-1){
    // empty
  } else{
    printf("%d", queue (F));
    F++;
  }
```

→ For a circular queue,

↳ circular queue

i) Enqueue:

```
enqueue (queue){
  if ((R+1)%n == F && R>F){
  // queue full
  } else {
  if (F == -1){
    F++;
  }
  R++;
  scanf ("%d", &queue[R]);
}
```

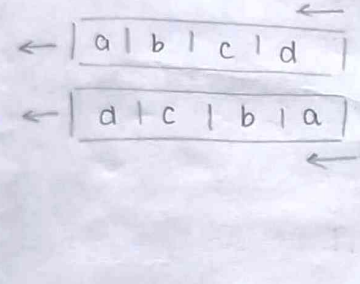
ii) Dequeue:

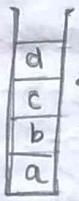
```
dequeue (queue){
  if (((R+1)%n == F && R<F) ||
  // queue empty        F == -1){
  } else {
  printf ("%d", queue[F]);
  F++;
  }
```

→ if we use linked lists to implement queues, we need not use circular linked lists, but if we use arrays, we need to use circular arrays.

Q. Implement a stack using 2 queues:

A.





O(1) push = enqueue Q1.

O(n) pop = dequeue Q2 once, dequeue Q1 till empty, enqueue Q2 in reverse order to Q1.

to print stack, print Q1.

we can make popping the cheaper operation instead as well

Q. Implement a queue using two stacks: HW.

A.

24/01/25

→ A graph is a set of nodes which are connected such that ∃ a root node, children of the root node and the connections here are called 'edges'.

If there is one unique path from the root node to any node in the graph, it is called a tree.

→ A Binary Tree is one where each node has at most 2 children.

→ i) depth of a node = no. of edges in the unique path from the root node to it

ii) height of a node = length of the longest path from a node to a leaf node that is its child.

ii) If n2 is along the path of one of a children of n1, n1 is an ancestor of n2.

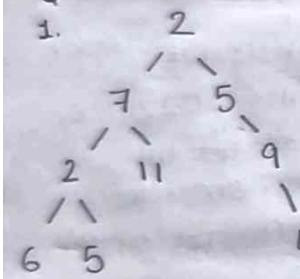
→ implementation:

⇒ when we do not known no. of children of nodes, we can define a

```
struct TreeNode {
    struct TreeNode* FirstChild;
    struct TreeNode* Next_Sibling;
};
```

Q

1.



→ root is pre i.e. Root-L-R

pre-order : 2-2-6-5-11-7-5-9-4

root is in' i.e. ← in-order : 6-2-5-11-7-2-5-9-4

post-order : 6-5-2-11-7-4-9-5-2

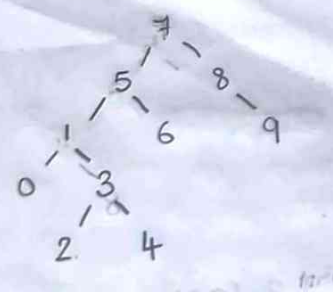
→ root is post i.e. L-R-Root

L-Root-R

→ level-order : elements in the same are taken left to right.

2. 7, 5, 1, 8, 3, 6, 0, 9, 4, 2 added in seq. to Binary Search Tree. what is its inorder traversal?

↳ A binary tree where no.s <root are on the left and > root are on the right



↳ in-order traversal of a BST always returns sorted order of elements

in-order =

0-1-2-3-4-5-6-7-8-9

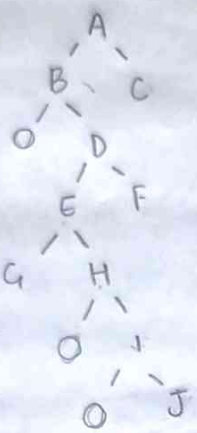
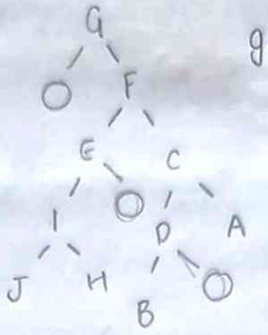
(i.e. in ascending order)

3. pre-order traversal = ABECD (root-left-right)
in order traversal = BEADC.

level order sequence?

∴ level order sequence = A-B-C-E-D

✦ Given in-order + post-order & pre-order + post-order, can you find the tree?

4. Which ordering of  and  gives same answer?

pre: A-B-D-E-G-H-I-J-F-C
in: B-G-E-H-I-J-D-F-A-C
post: G-J-I-H-E-F-D-B-C-A

pre: G-F-E-I-J-H-C-D-B-A
in: G-J-I-H-E-F-B-D-C-A
post: J-H-I-E-B-D-A-C-F-G

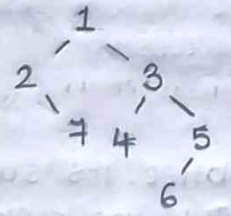
none of them match :-

5. In order traversal of a tree: FBGADCE. Pre-order traversal of same tree would be?

✦ note down options and try

a.
b.
c.
d.

6. Pre order: 1,2,7,3,4,5,6; in order: 2,7,1,4,3,6,5. what is the tree?



↳ get root from pre-order, and use in-order to find all the elements on the root and all the elements on the right

i.e. if inorder = [a b] c [d e]
a b: left of root in tree
c: root
d e: right of root in tree

→ A proper binary tree is a tree in which every node has exactly 2 or 0 children nodes.

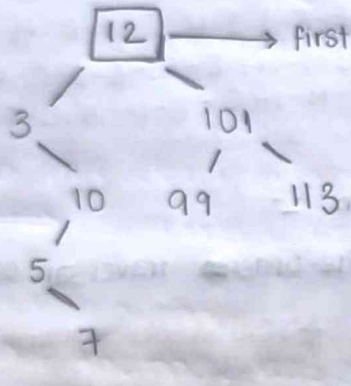
Q Pre-order 1,2,3,4,5,6,7
Post-order 2,4,6,7,5,3,1 } for a full/proper BST.

construct the tree

X

Q Create a BST using the numbers: 12, 3, 10, 5, 7, 101, 99, 113

A.



all nodes in the left subtree of a nodes are smaller than it, and all nodes in the right subtree are greater

→ the smallest element is always at the leftmost node.
similarly, the largest element is always at the rightmost node

→ the successor / predecessor is the number immediately after / before the given number in the sorted list of numbers.

to find successor / predecessor:

S: i) if a node has a right subtree, its successor is the leftmost node of that subtree. ii) If the right subtree of the node doesn't exist, we must traverse backwards through the tree.

i) if we are on the left subtree, the first node that has a left subchild is the root. At worst, this node is the root

ii) If we are on the right subtree, the first node with a left subchild is the successor. If there is no such node in the entire right subtree, the node we have has the greatest value, and successor does not exist.

↳ if at root in case (i), root = successor
(ii), successor doesn't exist.

P: (i) If the node has a left subtree, check for the rightmost node in that subtree to get predecessor.

(ii) if the left subtree of the node doesn't exist, we traverse backwards in the main tree, and try to find the first node which has a right subchild. This node is the predecessor.

If we are on the left subtree and reach the root in this manner, we say the predecessor doesn't exist.

→ Basic operations like finding minimum/maximum node, succesor, predecessor take time proportional to the height of the tree

i.e. $O(h)$

→ This method is preferable to doing in-order traversal, then going through the result (sorted list) for single / small no. of queries as the first process is $O(n)$ best case, while second process is $O(n)$ worst case.

for a small no./ 1 query.

→ Inorder traversal of a BST:

```
Inorder (x)
  if x ≠ NIL;  ⟶ case to ensure tree exists
  then Inorder (Left[x]) ⟶ inorder traversal of left subtree
       print key[x] ⟶ print element.
       Inorder (Right[x]) ⟶ inorder traversal of right subtree
```

→ Tree Search: → root

⇒ Tree-search (x, k) → num to search for.
if x = NIL or k = key[x] } to check if x < k / x doesn't exist,
then return x
if k < key[x]
then return Tree-Search (left[x], k) → if k < x, search left subtree of x for k
else return Tree-Search (Right[x], k)
↳ if x < k, search right subtree of x for k

⇒ This process runs in $O(h)$.

⇒ there is also an iterative approach to tree search, which is more efficient

→ To find minimum / maximum, keep going to the left/right subchild of each node. If the current node has no left/subchild, then it is the minima/maxima at any point.

→ Insertion and deletion of nodes
↳ insert nodes acc. to the BST property.

→ deletion of nodes: → this is the way that has minimal computation.
i) if the node has only one child, link parent of that node to the child of the node.
↳ takes up the same spot the parent did.

ii) if the node has no children, just delete it.
i.e leaf

iii) if the node has 2 children, replace current node with successor, and delete the successor.
↳ it is either case (i) or (ii), so we know alr. how to delete it.