DES 221

Data structure : Big O to Stack

- Big-O Algorithm
  - algorithm can be measured by running time and space * less is better
  - Counting Primitive Operations
    - ↳ count : assigning / calling method / arithmetic / comparison / indexing / object reference / returning
  - Complexity function : f(n)
  - Big-O Notation → tells the nearest bounded function when data size (n) approaches ∞.
    - → If f(n) is the time or space used by the algorithm,
      f(n) is O(g(n)) if constant $c > 0$ and $n_0 \geq 1$ such that
      $f(n) < c * g(n)$ for $n \geq n_0$ *
    - Big-O is a * growth rate * implies how much space and time needed to run.

$O(1) \rightarrow O(\log n) \rightarrow O(n) \rightarrow O(n \log n) \rightarrow O(n^2) \rightarrow O(2^n)$

constant — i=i+b in for loop — single loop — divide and conquer — nested loop

- Trick : list one-time and loop sepearately / nested loop count inside first / last comparison count seperately / i++ counted as 2 operations etc.

- Recursion → Easier to visualize & proof (ex. tree) / some shorther than iterative / nontail take more pops than iterative
  - make a method that call itself recursively.
  - Including : Base case = stopper / General case = continuer. and passed to next call

* law of recursion

1) recursive must have a base case
2) recursive must change its state and toward to base case
3) recursive must call itself, recursively (lol)

* Types of recursion

- Tail recursion → Can run as quick as iteration
  - last instruction executed by function
  - result is immediately returned.
- Non-tail recursion → can be worse than iteration if input size increase
  - not the last instruction by function
  - cannot partially return / save partially onto a memory stack

DES221

L1-5

No. DES 221

Date :

Array!

- always stored in consecutive memory locations
- required to have the same size and use the same data representation
- size is fixed / start the index at 0 end with length - 1 (size-1)
- if array is passed to use in method, it will pass the reference not memory's element.

▸ load : the number of actual elements in array (load <= size)

* insertion : add an element *

concept : want to add at i → move all element after i to the right → A[i] = value → load ++

O(n) case 1 : at first ; move all to right (A[i+1] = A[i]) → A[0] = value → load ++

O(1) 2 : at last ; A[load ++] = value

O(n) 3 : at i ; move element from i to load-1 to right (A[i+1] = A[i]) → A[i] = value → load ++

* deletion : remove an element *

concept : want to remove at i → store A[i] to temp → move all element to the left → load-- → return temp

O(n) case 1 : at first ; temp = A[0] → move right to left (A[i] = A[i+1]) → A[load-1] = 0 → load-- → return

O(1) case 2 : at last ; temp = A[load-1] → A[load-1] = 0 → load-- → return

O(n) case 3 : at i ; temp = A[i] → move right to left (A[i] = A[i+1]) → A[load-1] = 0 → load-- → return

O(1) * get and set * → A[i] = value / call by A[i]

- no gap/hole in the array / static data structure / only hold same data type.

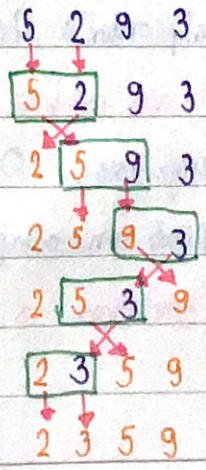
Sorting!

* Partitioning * $O(n^2)$ [concept : "partitioned" into sorted and unsorted group]

Type 1 : Insertion Sort

- let first element be sorted group.
- while unsorted is not empty :
  - compare with each element in sorted
  - if sorted > unsorted : chage position
  - otherwise stop.

example :



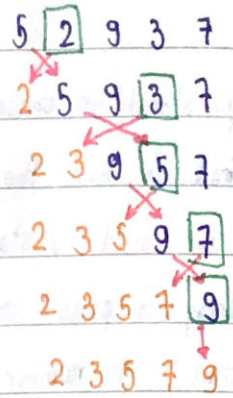
already in order : O(n) / reverse order : $O(n^2)$

average case : $O(n^2)$

depends on comparison and swapping

Type II : Selection Sort

example :



while unsorted group is not empty :

find smallest element

swap position of smallest and first position of unsorted

smallest element is now in sorted group.

already order : $O(n^2)$ / reverse order : $O(n^2)$

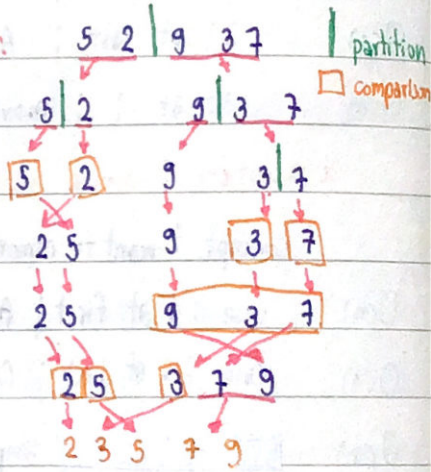
average case : $O(n^2)$

depends on comparison and array shifting

* Divide and Conquer Method * $O(n \log n)$ [concept : divide to smaller digestible problem]

Type I : Merge Sort

example :



- divide the list into group of 1 element
- compare each adjacent and merge

* if size of divided is odd → first will be less than second

already order : $O(n \log n)$ / reverse order : $O(n \log n)$

average case : $O(n \log n)$

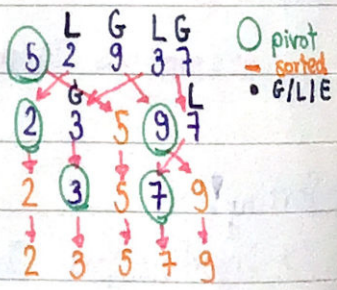
depends on comparison, assign data to temp array,

copying to original array.

Type II : Quick Sort

* first as pivot

example :



- select pivot and split to a sequence (less, equal, greater)
- move less to front of pivot, then equal, then greater
- do the same thing in less and greater until all is sorted

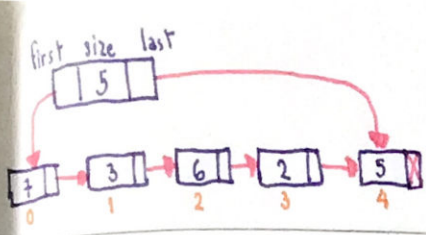
already order : $O(n^2)$ / reverse order : $O(n^2)$

↳ because need to move all data (loop in loop)

average case : $O(n \log n)$

depends on comparison, assign data

* limitation is selecting a good pivot / extra storage for temporary array



No.

Date :

linked list (Singly / SList)

- linear-fasion / linked — each data kept in node / each node has info of what's next node

* metadata : kept for fast access (include starting node / size / ending node)

* SList is dynamic data structure / transverse only one direction

* Insertion : add a node *

O(1) Case 1: at front | 1. create a newnode and put element in it.

O(1) 2. newnode.next = first → first = newnode → size++

O(n) Case 2: at i / at last | 1. create a newnode and walker (current)

O(n) O(n)

2. use loop to put walker at the designated index (stop at i-1) / current=current.next

3. newnode.next = current.next → current.next = newnode → size++

4. if put at last ; last = newnode

* Deletion : remove a node *

O(1) Case 1: at front | 1. create a temporary node

O(1) 2. temp = first / first = temp.next / temp.next = null / size--

3. return temp.element

* If size = 0 → return null / size = 1 → first = null, last = null, size = 0 before return.

O(n) case 2 : at last / at i | 1. create a temporary node (temp) and walker.

O(n) O(n) 2. use loop to put walker at the designated index / current = current.next

3. temp = current.next / current.next = temp.next / temp.next = null / size--

4. return temp.element

* If size = 0 → return null / size = 1 → first = null, last = null, size = 0 before return.

* If remove at last : temp = current / last = current / current.next = null / size-- in step 3

O(n) * Search : find the corresponding element and return index of that *

O(n)

1. index = -1 (set like this because if not found just return -1) and walker = first

2. loop until last element : If walker.element = key ⇒ index = i and break the loop.

walker = walker.next

3. return index.

* dList = double linked list / pointed in 2 direction | circular linked list → last.next = first

↳ complexity decrease, but more memory usage

Stack

- Special type of SList * Accessed/Inserted/Removed from only one end *

* Insertion : PUSH * → addFirst method of SList

* Deletion : POP * → removeFirst method of SList

* Search : PEEK * → first.element / getElementAtIndex(0) of SList

Application

I. binary conversion : stack.push(input%2) → input /= 2 → loop until input = 0 → print(Stack.pop()) until stack empty

II. check palindrome : have 2 stacks of input (one is normal, other is reversed) then pop. each of both stack and check if stack is empty then it is palindrome if found contradiction break the loop and it is not palindrome.

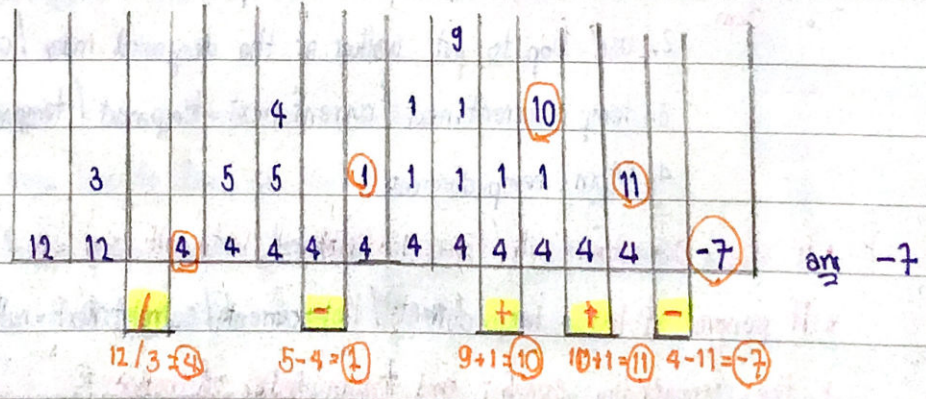
* III. Postfix Expression : parenthesis free expression & 2 operands followed by 1 operator

Read element one by one > not operator > push to operand stack

an operator > pop 2 elements and evaluate infix expression then, push the result to the stack

repeat until prefix runs out.

example : 12 3 / 5 4 - 1 9 + + -



* IV. Infix expression : operator between operands / may come with parenthesis / calculate based on priority (*/%) then (+,-)

* use 2 stacks / operands and operators stack

* stack the parenthesis in operator stack too!

No.

Date :

How to Run Infix Expression:

1. Create operands and operator stack and read element one by one >
2. element is not operator > push to operand stack

is an operator > operator is higher priority than top > push to operator stack

is equal or lower than top > pop 2 from operand stack & 1 from operator stack

then evaluate and push back to operand.

is an open parenthesis "(" > push to operator stack

is a close parenthesis ")" > pop 2 from operand and 1 from operator, evaluate then push to operand

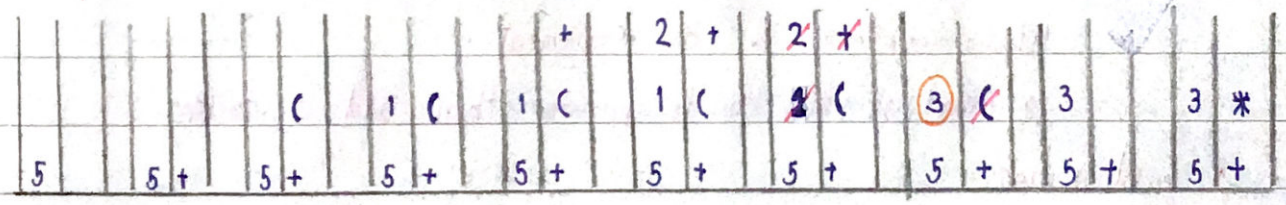
repeat until top is open parenthesis after that pop "(".

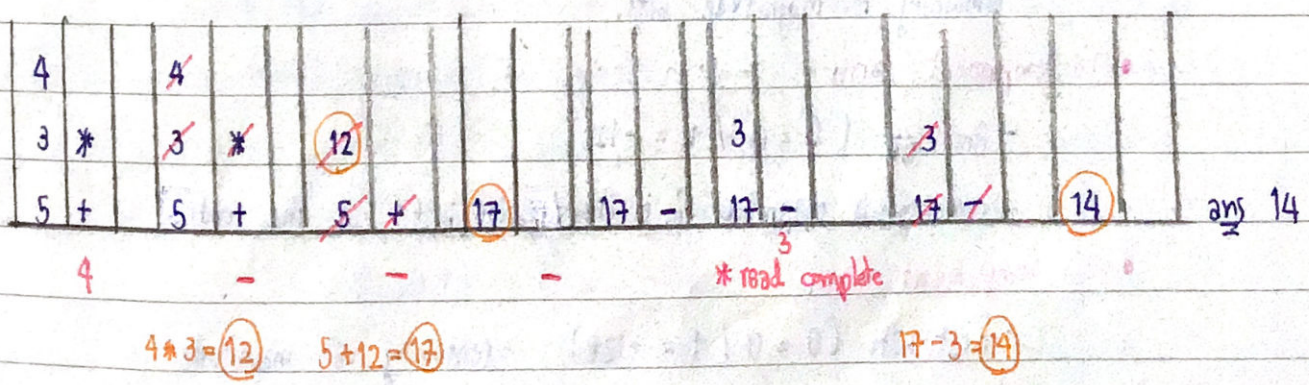
3. Once all completed but operator stack is not empty > pop 2 operands and 1 operator

evaluate and push to operand stack

repeat until operator stack is empty.

example: 5 + (1 + 2) * 4 - 3





No.

Date :

• DES221

Data Structure : Queue to Hashing

▼ Queue - another special type of SList * FIFO : first in, first out *

* Insertion : enqueue * → SList addLast (element)

* Deletetion : dequeue * → SList removeFirst()

* get value : queueFront , queueRear * → SList list.first.element , list.last.element.

▼ Application

* I. Prefix Expression : operands after operator (eg. +23) / parenthesis-free

Evaluation : Create Queue > Load element > Dequeue 3 elements from Queue

→ Check if dequeued form a prefix → [T] evaluate and enqueue result to queue

→ [F] enqueue the first, dequeue one from queue

ex. + + - 6 7 + 3 2 8

(+) + - 6 7 + 3 2 8

+ - 6 7 + 3 2 8 (+)

6-7 = -1 - 6 7 + 3 2 8 + +

3+2 = 5 + 3 2 8 + + -1

(8) + + -1 5

(+) + -1 5 (8)

-1+5 = 4 + -1 5 8 +

(8) + 4

4+8 = 12 + 4 8

12 → ans *

① throw dequeued to enqueue

---- throw line

formed prefix

check tell if it prefix or not

• II : Round Robin Scheduler : used for fairly allocate a resource wl same amount in each round.

Algorithm : ① dequeue Person and amount ② deduce number by specify amount ③ if amount = 0 no need to enqueue, otherwise enqueue back ④ repeat until resource ran out.

▶ Hashing - Technique used for insertions, deletions, searching; at *a constant time*
- key-to-address mapping process (data-to-location)
- Array to store → hash table size of table → hash function
  location of data → address multiple key to same address → collision

* Hashing function
- Division H(key) = key % hashSize
- Folding H(key, C) = (⌊key/C⌋ + key%C) % hashSize
  = (floor(key/C) + key%C) % hashSize
  * C is constant, usually hashSize is Prime number

* Collision Resolution
- Separate Chaining : use list to keep elements that have same hash value
- Open Addressing : relocate to a new cell when there is a collision
  ↳ H(key, i) = (H(key) + f(i)) % hashSize ; f(0) = 0
  - Linear Probing : f(i) = i
  - Quardradic Probing : f(i) = i^2
  - Double hashing : f(i) = i * H2(key) H2 is another hash function
    i start at 0 and in succession (a.k.a. $i \in [0, \alpha)$ & $i \in \mathbb{Z}$

* Load factor (λ)
- how full of your hash table
- λ = number of data / hashsize
- if λ closer to 1, more likely that collision will happened.
  for $\lambda < 0.5$ should be in open addressing
  $\lambda < 0.9$ should be for seperate chaining.

* Rehashing
- use for migrate data to different hash table w/ bigger size
- use when hashTable is full or nearly full
- define new hash function for new table.