```
newNode -> next = temp -> next;
newNode -> prev = temp;
temp -> next -> prev = newNode;
temp -> next = newNode;
```

DELETION

Deletion of inner node

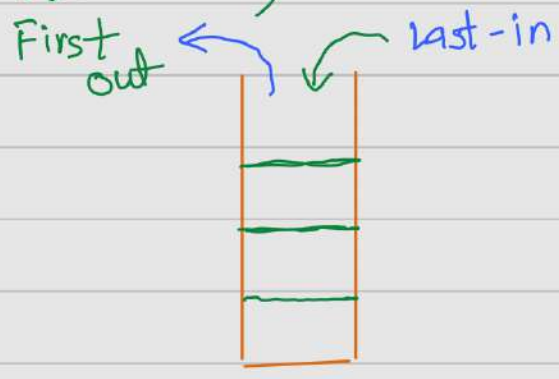

```
temp->next = temp -> next -> next;
temp->next->prev = temp;
```

} pointer changes

```
HEAD = func ( )
temp -> next = newNode;
```

STACK

Linear data structure following principle of (LIFO)
(Last-in First-out)



Putting an item on top is called "push" & removing is called "pop"

Main Operations:

* Push - adds element on top

* Pop - Removes element from top
* IsEmpty - check is stack is empty
* Is Full - If stack is full
* Top - returns the top element

Working

→ A pointer TOP keeps track of top element
→ When intialising the stack, we set TOP value to -1 so as to check for emptiness
→ Pushing the value we increase the value of top & place new element at position pointed by TOP
→ On popping remove the element from top.
→ Before pushing check if it's full
→ Before popping check if it's empty

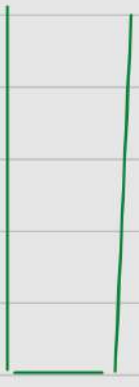
Application → Used in function calls, Recursion Expression evaluation.

Balanced Brackets

Consider a string " {{{{}}{{}}} ". You need to check if all pairs of brackets "{}" are paired up & in-order.

Eg. " {{{{{{{}} " not balanced

" {{}}}{{ " not balanced



Next bigger element:

Given an array find the next bigger element of every element & if there doesn't exist any mark it -1.

Original Ar: 6 7 4 5 1 2 10

Ans: 7 10 5 10 10 10 -1

idea: Use 2 loops 1st iterate over all elements & then for each element find the next greater element.

```
for(int i=0; i<n; ++i)
{ for(int j=i+1; j<n; ++j)
    { =; }
}
```

Time complexity: $O(n^2)$

$$f(n) = \sum_{i=0}^{n-1} (n-1)-(i+1)$$
$$= (n-1)^2 - \sum_{i=0}^{n-1} (i+1)$$
$$\approx O(n^2)$$

Better idea: Using Stack: Time Complexity: $O(n)$

Starting from end keep a stack of next greater element possibilities

You have answer till here!

6 7 4 5 1 2 10

7 10 5 10 2 10 -1

could you ans. for this index?

if a[i] < a[i+1] a[i+1] is the ans

otherwise check if with ans. of a[i+1] is ans. or not & similarly move until you get the ans.

In other words you compare first with last possible ans.

6	7	4	5	1	2	10
7	10	5	10	2	10	-1

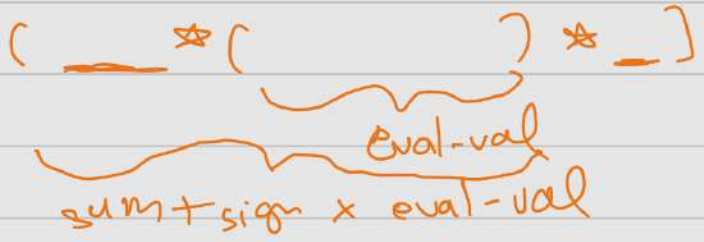
Calculator problem

$1 + (2+3 - (4+5) - 6) + 7$

keep sum & sign variables extract number add them to sum variable acc. to their sign as soon as you encounter "(" put the current sum & sign as a pair in the stack. & set sum = 0. Then as you encounter ")" take out top of the stack & make ten

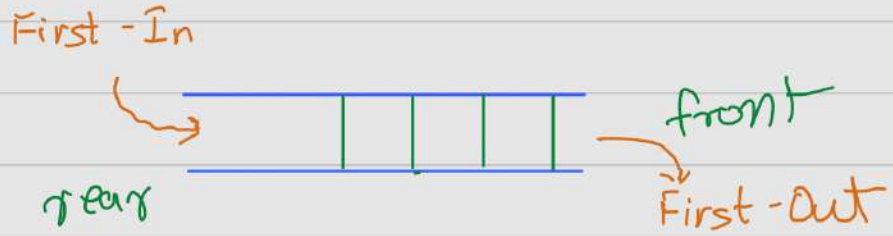
sum = Top.first + Top.second * sum

Remember if you encounter '-' change the sign



QUEUE

Queue is a linear data structure following FIFO rule (First-In First-Out)



Putting items in queue is called "enqueue" & removing these items is called "dequeue"

Basic Operations

* Enqueue → add element to the end
* Dequeue → Remove an element from front
* IsEmpty → If it's empty returns true else false
* IsFull → If it's full
* Front → returns you the front element

Working

→ two pointers FRONT & REAR.
→ FRONT tracks first element
→ REAR tracks last element
→ Intially both set to -1.

Enqueue

→ Check if queue is full
→ For 1st element i.e. if it had been empty set FRONT to 0

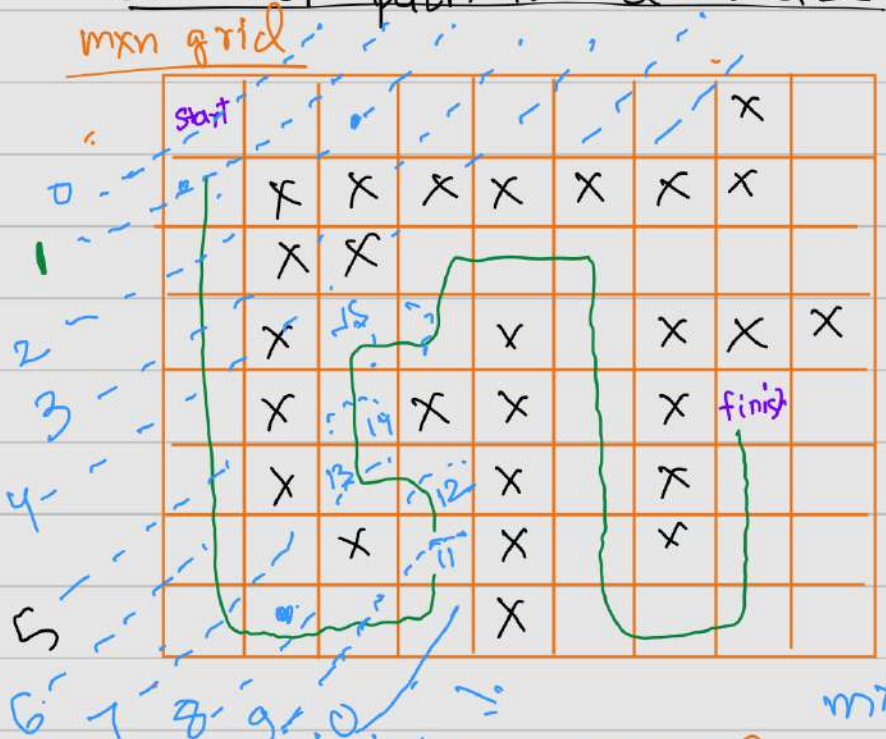
→ increase REAR by 1.
→ add element in position of REAR

Dequeue

→ check if queue is empty
→ return value pointed by front
→ increase FRONT by 1 index
→ For last element, reset value of FRONT & REAR to -1.

Shortest path in a maze

mxn grid



mxn

Time complexity : O(mn)

x : obstacles

Idea: Inspired from water ripples
Starting from "Start" travel like ripple & update the distance if it's not a obstacle & mark visited so that you don't visit it again for updation (1st updation will give the min dist)

dist[start] = 0

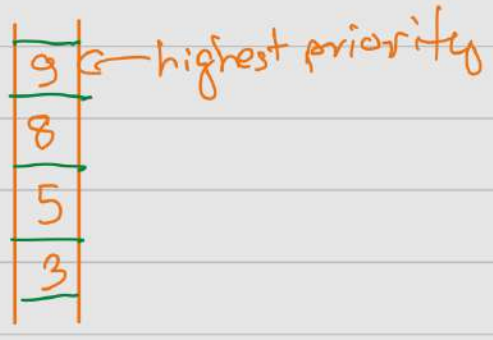
neigh valid → ~~obstacle~~
vist → X

dis[i][j] = min distance of i, j from start

vis[i][j] = (i,j) has been visited / distance has been updated.

PRIORITY QUEUE

It is a special type of queue in which each element is associated with a priority value and elements with higher priority are served first.



c++ Priority queue methods:

push(): inserts element in priority queue
pop(): removes the element with highest priority
top(): returns the element with highest priority
size(): size of priority queue
empty(): if priority queue is empty or not

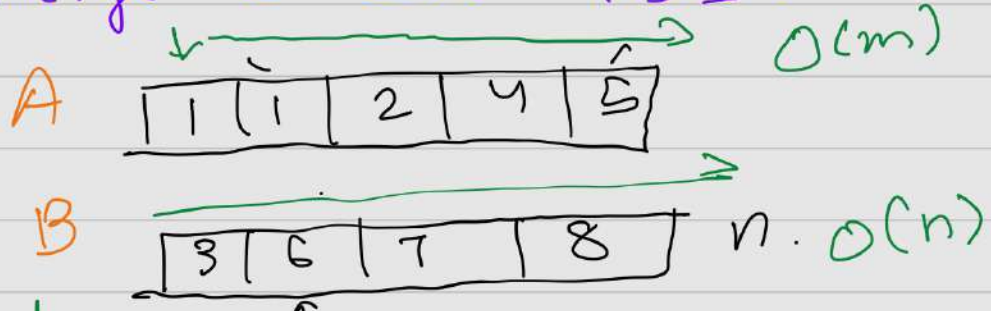
By-default priority: bigger-element is given higher

It is implemented by heap which we will. → You will learn more about this later

" Used to work with highest priority element at

the moment "

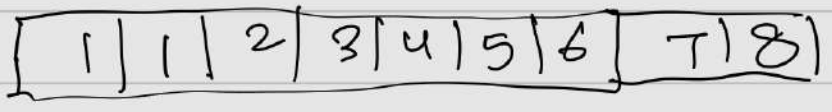
Qn: Merge 2 sorted lists.



Output merge sorted array

trivial idea

$O((m+n)\log(m+n))$ Merge them into one array & the sort the complete merged array



$O(m+n)$

Qn: Merge k sorted lists

priority-queue

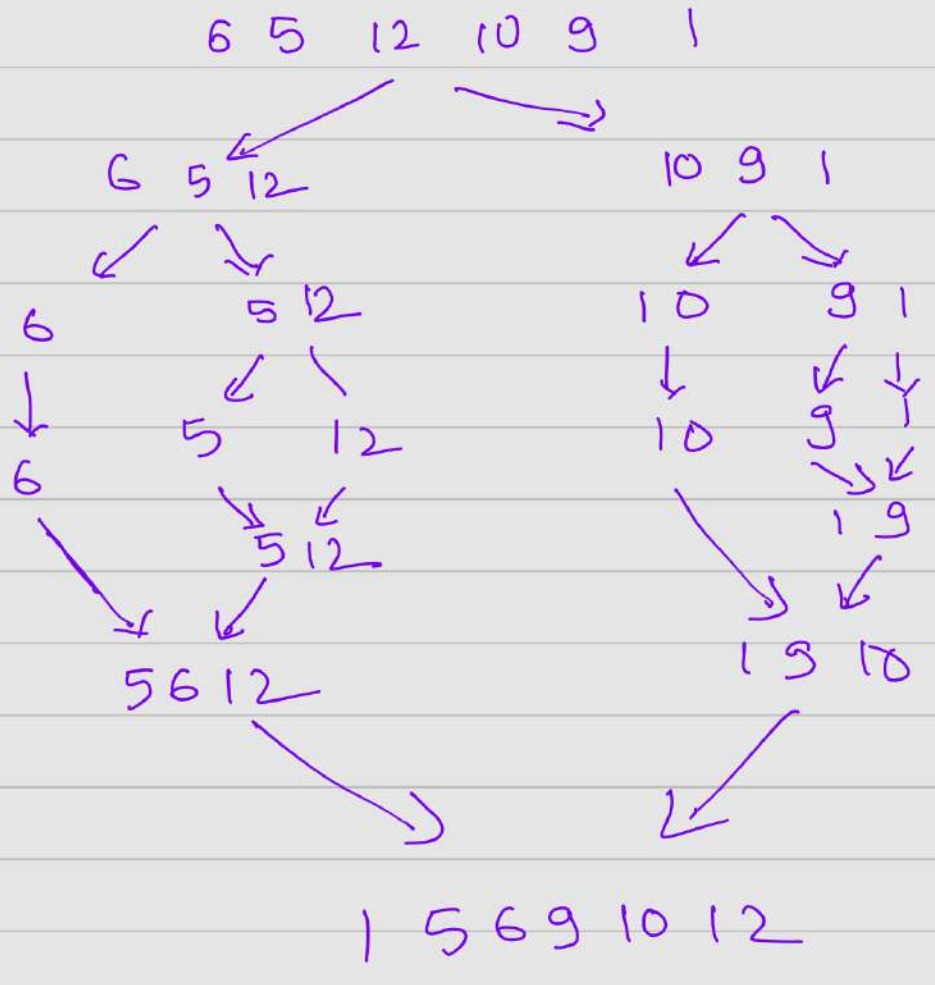
(—, 1)

take all the 1st element

Merge Sort

Divide & conquer Strategy

→ Divide the two arrays in two halves ← Divide
→ Recursively sort the two parts ← Conquer
→ Merge them to form final array ← Combine



Pseudo code:

```
mergeSort (A, p, r) {          → T(n)
    if p > r
        return
    q = (p+r)/2
    mergesort(A, p, q)         — T(n/2)
    mergesort(A, q+1, r)       — T(n/2)
    merge (A, p, q, r) }       — O(n)
```

$T(n) = 2T\left(\frac{n}{2}\right) + cn$ $\qquad T(1) = 1$

Time complexity : $O(n \log n)$

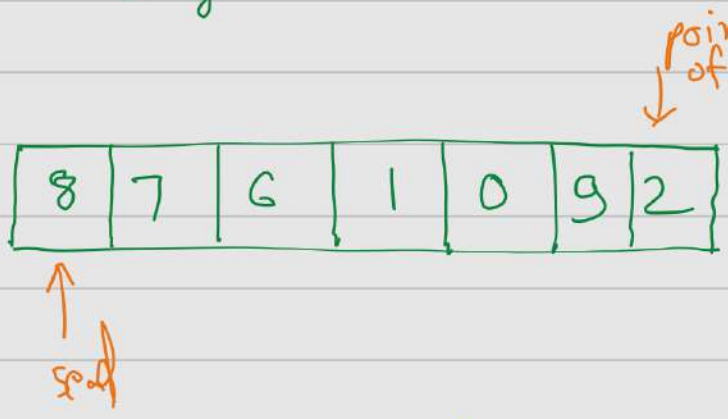
Quick sort

- Divide & conquer approach

→ Choose pivot element. Partition the array s.t. all smaller elements belong to one side & other elements bigger than pivot on the other side.

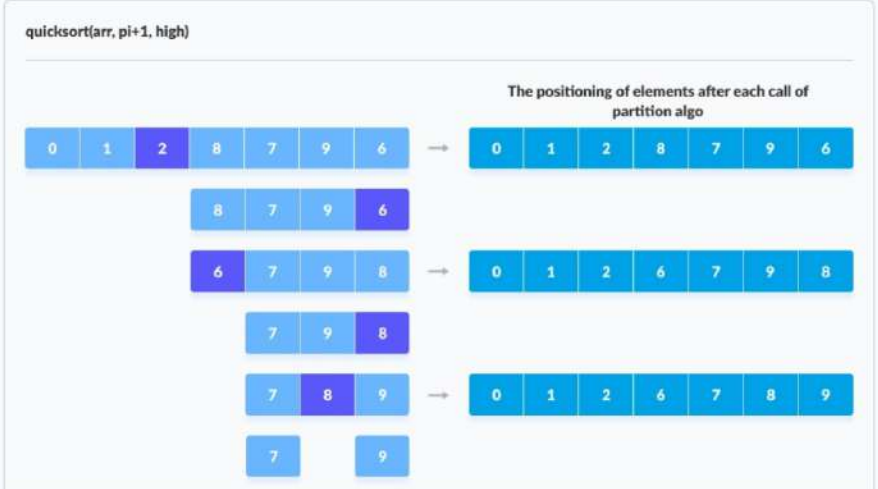
→ Do this recursively for each side of the pivot. This continues until each subarray contains a single element.

partitioning of array in $O(n)$




```
Quick Sort ( A, left, right)
    if ( left < right )
        { pivot_ind ← partition ( A, left, right)
          QuickSort ( A, left, pivot_ind - 1 )
          QuickSort ( A, pivot_ind, right )
        }
```

Sorting the elements on the left of pivot using recursion



Sorting the elements on the right of pivot using recursion

Please Read & Practice these two sortings. write their implementation once.