Page No. :
Date :

Segment Tree.

5/08/2020

1	3	2	4	3	-2	6
0	1	2	3	4	5	6

Q: Range minimum query.
minimum element in a range.

eg. 0-3 : 1
2-4 : 2
4-6 : -2.

How we did.

① using linear search. L-R.
O(N) for 1 query.
O(NQ) for N queries.

Q↑ & T↑.

so Segment tree will reduce query time.

② Pre computation. ($O(n^2)$ time to build)

	0	1	2	3	4	5	6
0							
1	x						
2	x	x	.				
3	x	x	x				
4	x	x	x	x			
5	x	x	x	x	x		
6	x	x	x	x	x	x	

query o(1) time.

① Queries o(1).

② Updates o(n²).

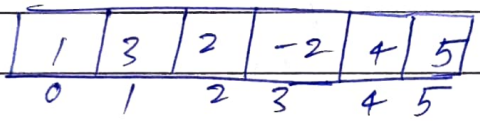
(In 2nd approach)

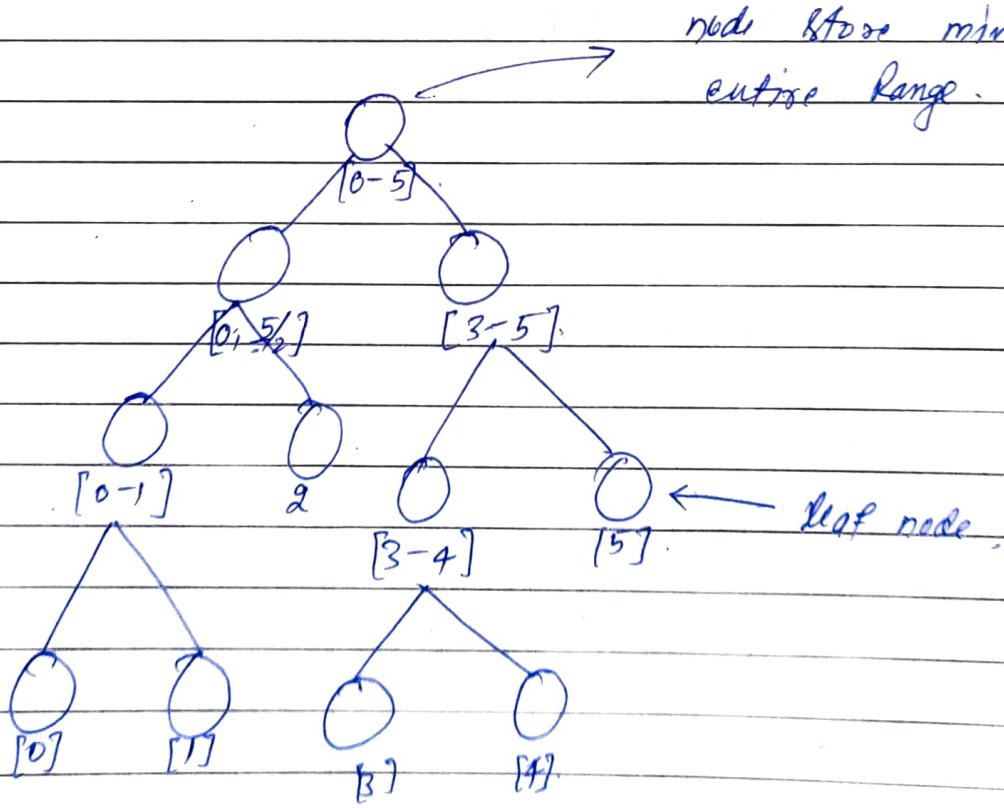
1000 q. o(1).

1000 Update. o(N²).

① Data structure.

Segment Tree for above problem. (Complete b.t).

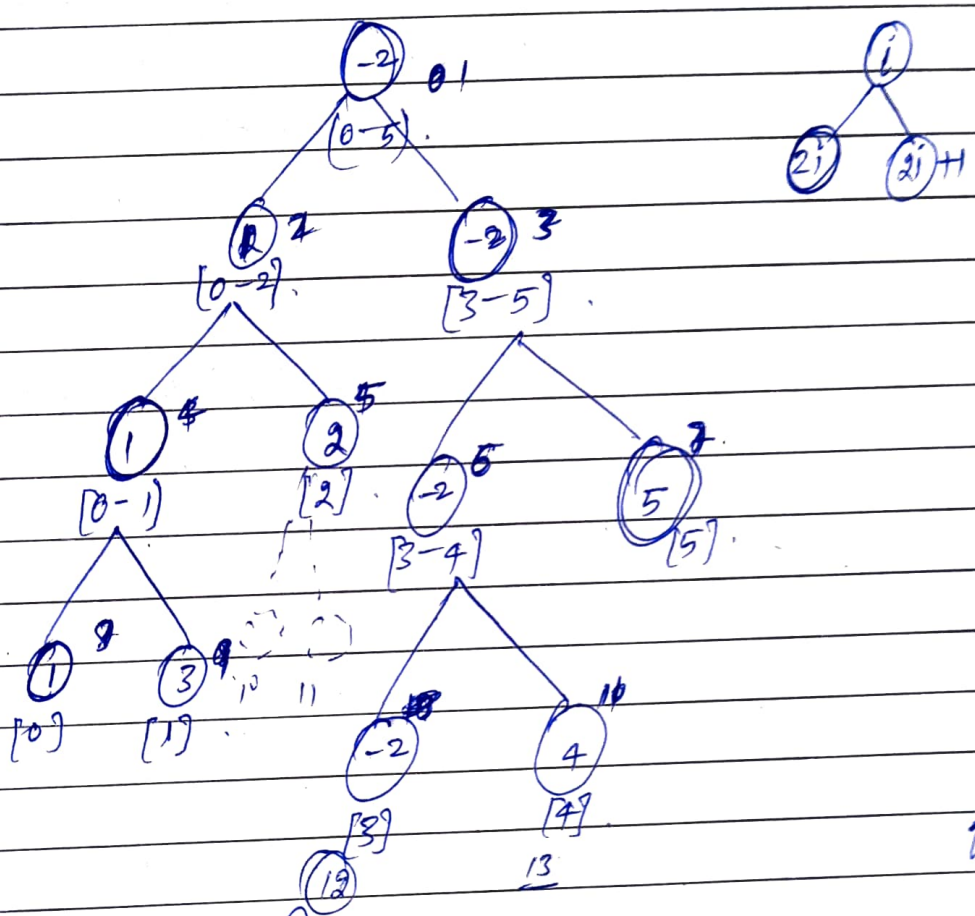




Page No. :
Date :

Segment Tree (based on divide & conquer).

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



① leaf node fill किया
② bottom up way में सभी ranges को fill किया

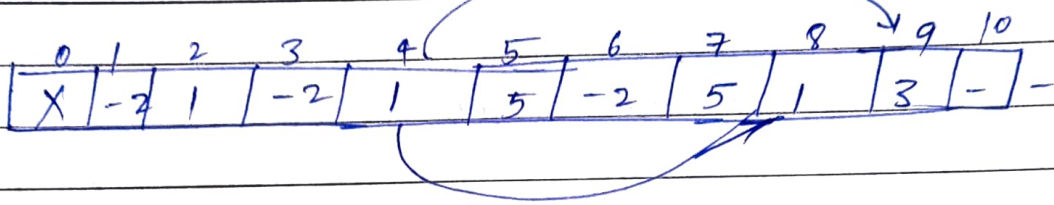
① Build
② query
③ Update

Page No. :
Date :

① Array to build tree.

(Bottom up way
Recursion)

→ used to store data in hierarchal tree.



Tree ⟶ Hierarchial info to store करना

↓

(Array Relationship.)

★★ no. of elements = $2n-1$

= n leaf nodes.

= n-1 internal nodes.

Array size = $2 * 2^{\lceil \log_2 N \rceil} - 1$

not need to

Max array size = $4*N+1$

to build a tree of size N

→ we can start numbering from 0 as well.

ⓒ actual:

Segment → $2N - 1$

$= 2 \times 6 - 1$

$= 11$

so worst case # numbering will go to.

$4 \times 6 - 1$

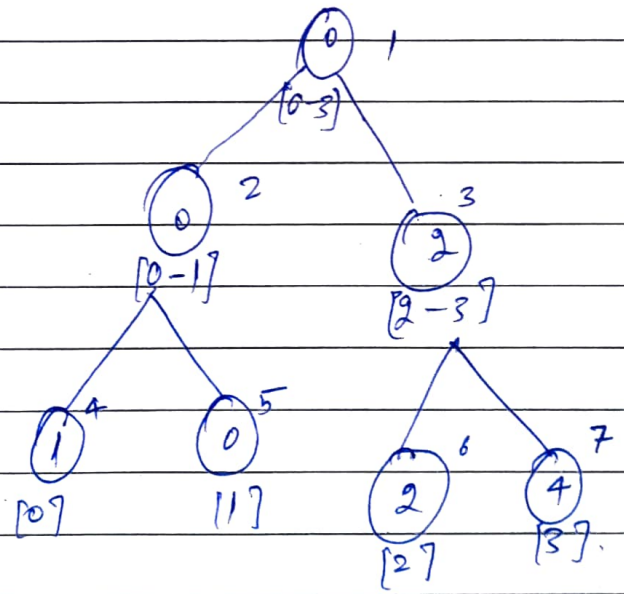
$= 24 + 1$ → if start from 1

$= 23$

$= 4N * 1$ → if start from 0.

eg.

1	0	2	4.
0	1	2	3.



tree node = $2N - 1$

$= 7.$

(4N+1).

↓

for array from tree

Page No. :
Date :

Divide & Conquer :- Problem solving technique.

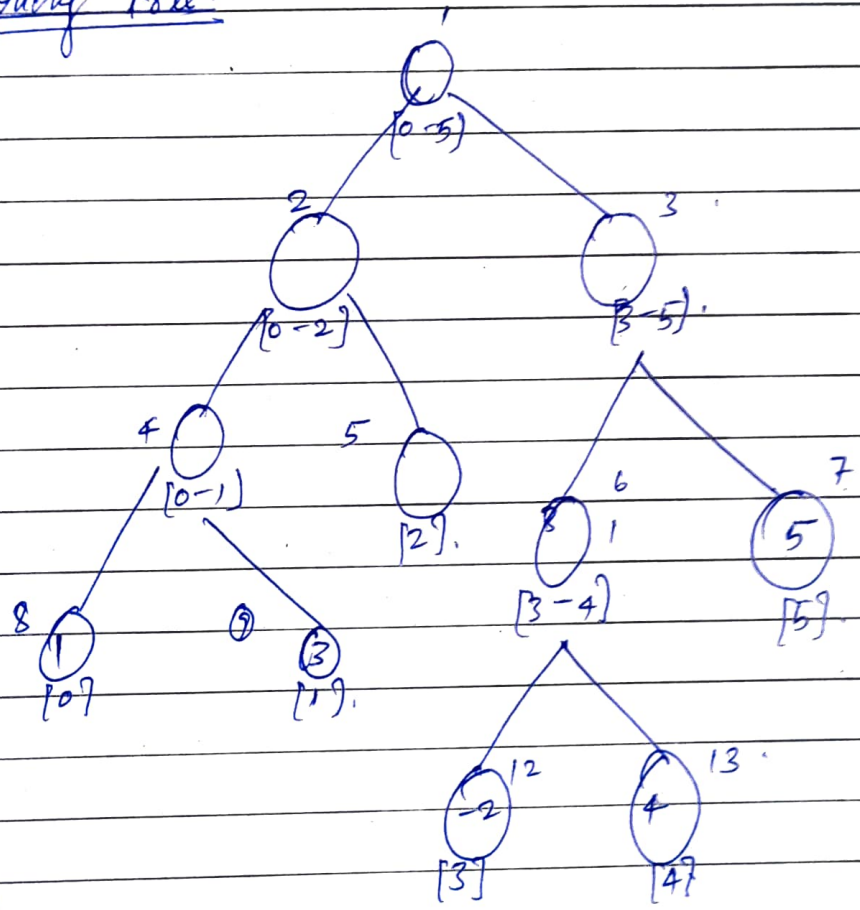
Segment Tree :- Data Structure.

```
void build tree (int *tree, int *a, int index, int s, int e)
{
    // Base case.
    if (s > e) {
        return;
    }

// base case for leaf node.

    if (s == e) {
        tree [index] = a[s];
        return;
    }

// Recursive calls.

    int mide = s+e / 2.

// left sub tree.

    buildtree ( tree, a, 2 * index+1 , s, mid, mid++, e );

Right

    buildTree (tree, a, 2*index+1, mid+1, e );
```

```
int left = tree[2*index];
int right = tree[2*index+1];

tree[index] = min(left, right);
}
```

Range पर कोई query करने पे segment tree is used.

Query tree:

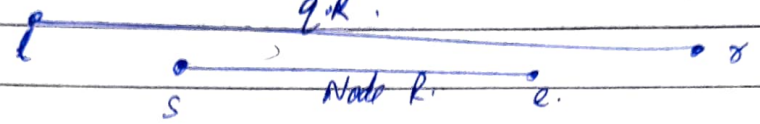


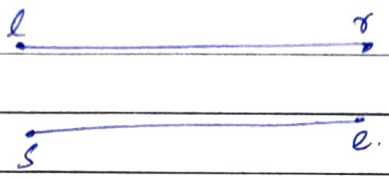
3 case:
we are finding query → Min but can be anything.

Page No. :
Date :

query Range, node Range.

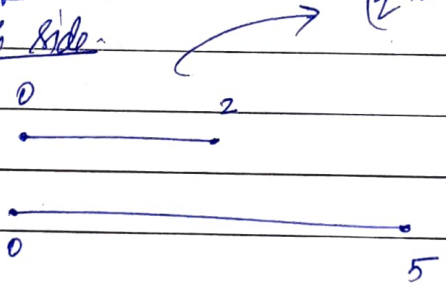
① complete overlap.



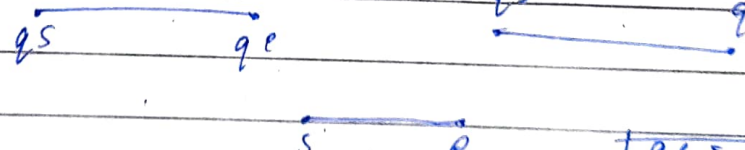


In that case Return tree[index] जहाँ पर है वहाँ

② Partial Overlap.
Left on both Side.



③ No overlap
return '∞'



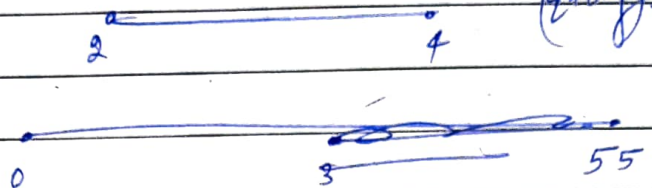
qs > qe
qe < s
(No overlap)

return tree[index]

Page No. :
Date :

query of range partially lies on node of range.

(3)



(complete overlap) means both calls).

// return a min element from the tree lying in range qs and qe.

int query(int *tree, int index, int s, int e, int qs, int qe){

// 3 cases.

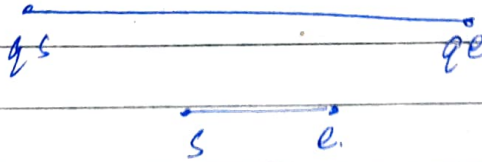
// 1. No overlap.

if (qs > e || qe < s) {

return INT_MAX;

}

// 2. Complete overlap.

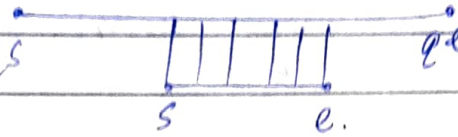


s >= qs && e <= qe.

index = shows current array index

Page No.:
Date:

3rd Node ka range lies within query range, then only will be complete overlap.



$s \geq qs \;\&\&\; e \leq qe$

```
if (s>=qs && e<=qe){
    return tree[index];
}
```

3. Partial overlap — call both side.

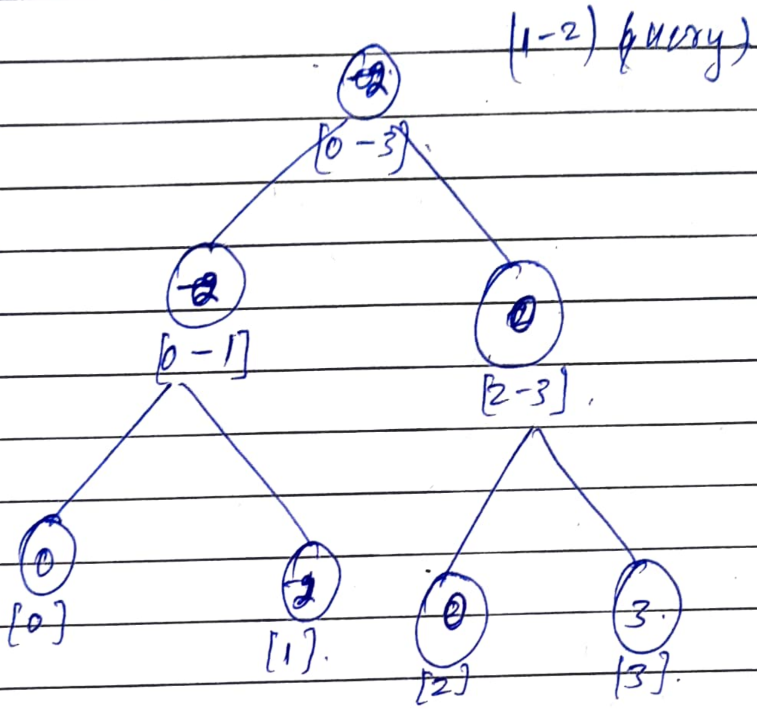
```
int mid = (s+e)/2;
int mid = (s+e)/2;
int leftAns = query(tree, 2*index, s, mid, qs, qe);
int rightAns = query(tree, 2*index+1, mid+1, e, qs, qe);
return min(left, right);
}
```

Page No.:

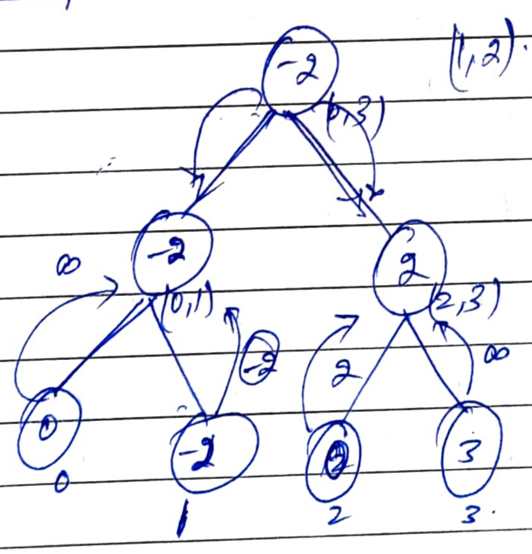
Date:

dry Run:

1	-2	0	3
0	1	2	3



Partial overlap



if query Rang out of helper Range.

check करना कोई seperate helper fn के through

query(tree, 1, s, e, qs, qe);.

Page No. :
Date :

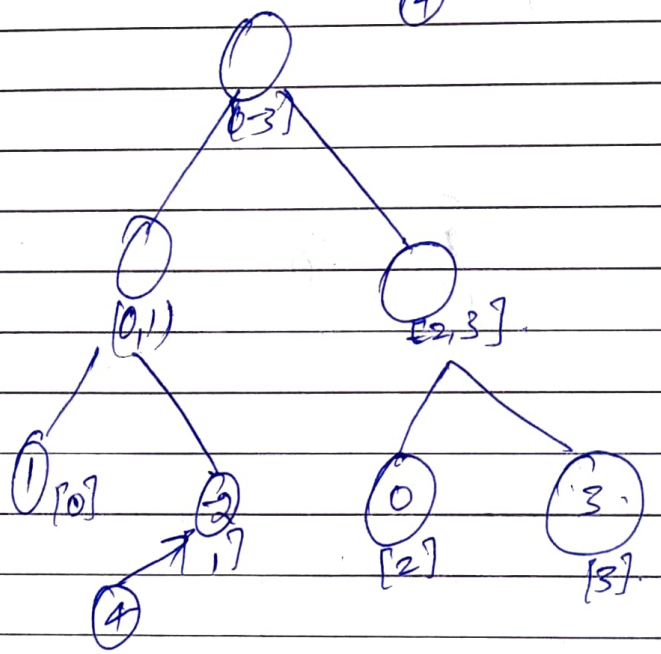
** Update function.

1	0	3	-2.

how to update

① Node update
② Range update.

1	-2	0	3.



i = 1, V = 4.

i = जिस node को update करना है, (value = जो नई value है)

void updateNode (int *tree, int index, int s, int e, int i, int value).

// no overlap

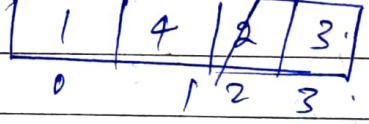
```
if(i<s || i>e){
    return;
}
```

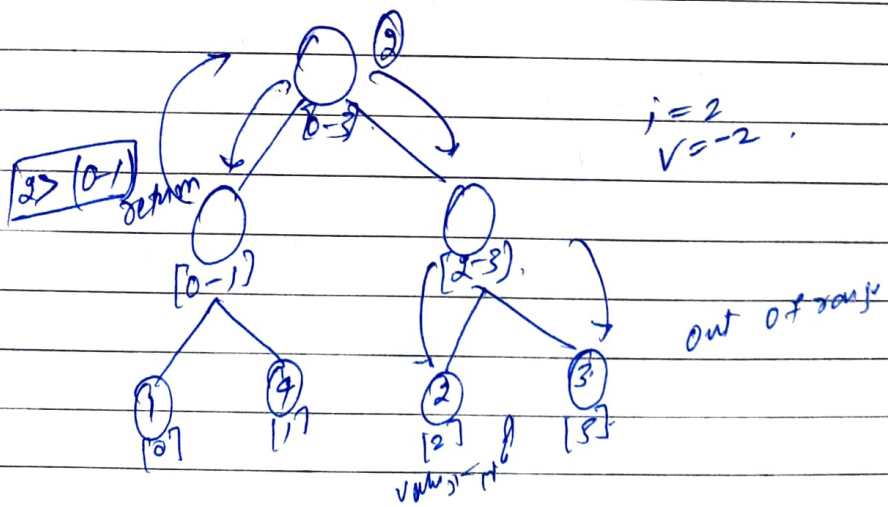
// Reached leafNode.

```
if(s==e){
    tree[index] = value;
    return;
}
```

// Partial overlap. 'i' is being in b/w (s,e) range.

```
else int mid = (s+e)/2;
updateNode(tree, 2*index, s, mid, i, value);
updateNode(tree, 2*index+1, mid+1, e, i, value);
tree[index] = min(tree(2*index), tree[2*index+1]);
}
```





Page No. :
Date :

Range Update

↑ment every value in a range by a value 'v'.

Seg tree operation Supported

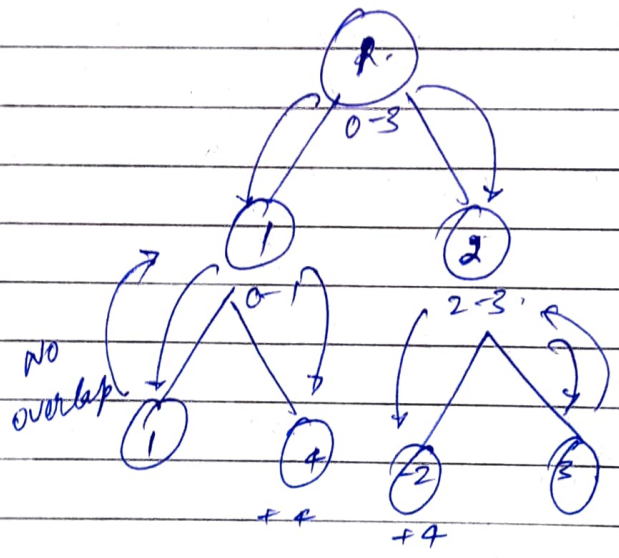
1. Range update
2. query
3. Build a segment tree.
4. Node update.

Delete is not function of segment tree.

RU(1,2) = 4 ↑ment

1	4	-2	3
0	1	2	3

1	8	2	3



time = O(1).

time(s,e) = O(n).

worst Case.

Page No. :
Date :

```
void updateRange(int *tree, int index, int s, int e, int rs, int re,
                 int inc){
  // No overlap
  if(re < s, rs > e){
    return;
  }
  // Reached to leaf node.
  if(s == e){
      tree[index] += inc;
    return;
  }
  // lying in range call both side.
  int mid = (s+e)/2;
  updateRange(tree, 2*index, s, mid, rs, re, inc);
  updateRange(tree, 2*index+1, mid+1, e, rs, re, inc);
  tree[index] = min(tree[2*index], tree[2*index+1]);
  return;
}
```

```
int main() {

    int a[] = {1, 4, -2, 3};
    int n = 4;

    // Build the tree array.

    int *tree = new int [4*n+1];
    int index = 1;
    int s = 0;
    int e = n-1;

    buildtree(tree, a, index, s, e);

    int no_of_q;

    cin>> no_of_q;

    updateNode(tree, 1, s, e, 1);

    while (no_of_qu--) {
        int qs, qe;
        cin >>qs >> qe;

        cout << "Min value b/w range is";

        cout << query(tree, 1, s, e, qs, qe) << endl;
    }

    return 0;
```

Page No. :
Date :

① Build = $O(N)$

because $4N+1$ Nodes are build.

So,

$O(N)$.

② query = $\log(N)$.

③ Update = $\log(N)$.

④ Range Update = $\log(N)$.

$O(s,e) = O(N)$

→ In worst case

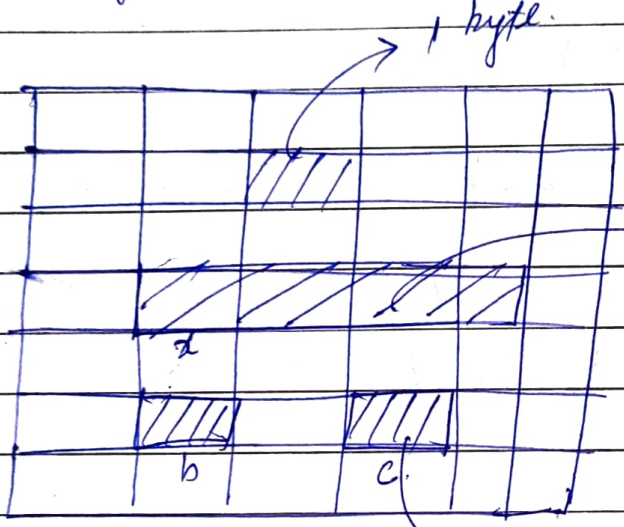
Page No. :
Date :

Pointers.

int 4 byte.
bool 1 byte.
char. 1 byte.
float 4 byte.

1 byte = 8 bits.

Memory

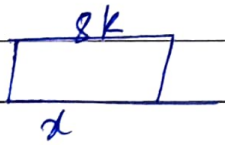


each block has.

int a = 10.

add in terms of hexa decimal (0-9 and a-f).

& operator called addres of operator.



if we write  → g

& operator gives add

→ whole goes and find addres of a in terms of hexadecimal format

Page No. :
Date :

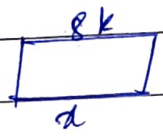
Pointer :- Pointer is a method by which we stores these address value.

```
int * xptr ;
```

→ Says we are going to make pointer 'xptr' which is going to stores addren of integer type variable.

```
int* xptr = &x.
```

① to store integer we write 'int'
② for boolean type we write bool.
③ similarly to store address we write '*'
int* xptr

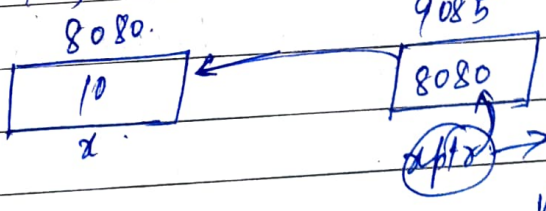


`int* xptr = &x.` or `int *xptr` both same.

*return 8k.

xptr stores 8k.

```
int x = 10;
```



→ gives add of value at which it was pointing.

Page No. :
Date :

Same for float & boolean as well.

* generic pointer :-

T Var.

↑ Data type

T* Varptr = &Var

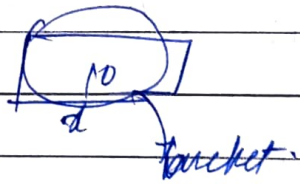
↑ stores add of generic data type

** Dereference operator :-

& → add of variable.

*ptr denotes the variable not address.

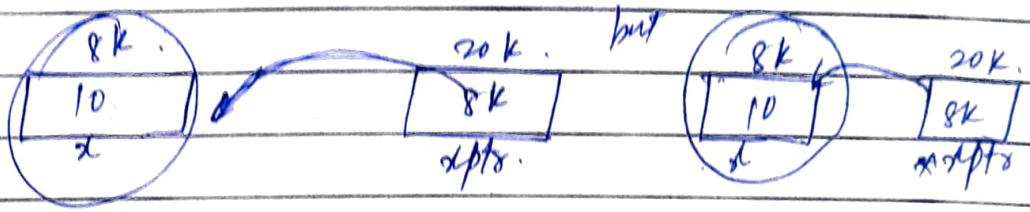
Var. | bucket (a) → address.



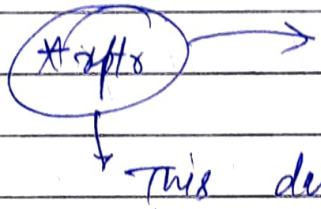
address —*→ bucket

dereference. Can get bucket.

Print(*xptr) // gives value.



xptr → access memory 8k & by * we (extract the variable) fetch value from that address. and using it



cout << (*xptr) << endl ; // print value.

eg.

*xptr = *xptr + 1

is equivalent to

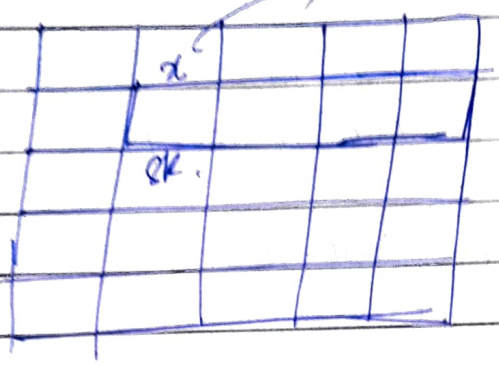
x = x + 1.

Page No.:
Date:

Var.

T* ptr = &var.

if x is integer.



so, int* xptr = &x.

extract 4 byte.

but if it is char type then only 1 memory add is allocated.

char x = 'a';

char *xptr = &x.

** sizeof(ptr) 32 bit system = 4 byte.

64 " " = 8 byte.