HQ17

(a) // x[1..n] is an array of bits, that will be used in the following operations.

(i)
```
def Init(n):
    for i = 1 to n
        x[i] = 0
```

(ii)
```
def Set(i):
    if( i >= 1 && i <= n-1)
        x[i] = 1
```

(iii)
~~def IsSet(i):~~
~~return x[i]~~

```
def IsSet(i):
    if( i >= 1 && i <= n)
        return x[i]
```

(iv)
```
def NextUnset(i):
    for k = i to n
        if( x[k] == 0)
            return k
```

time complexity of the above operations —

(i) $\Theta(n)$ // given n, the for loop will run for all the n bits and set it to value 0

(ii) $\Theta(1)$ // constant operation

(iii) $\Theta(1)$ // constant operation

(iv) $O(n)$ // we will return k as soon as we find x[k] > 0 and the for loop will terminate

(b) (i) def Init(i):

```
for( int i = 1 to n)
    makeSet(i)
    i.val = 0
    i.ptr = i
```

(ii) def Set(i):

```
if ( i >= 1 && i <= n-1 && i.val == 0)
```

(a) temp = find(i+1).ptr

(b) i.val = 1

(c) union(i+1, i)

(d) find(i).ptr = temp.

(iii) def ISet(i):

```
if ( i >= 1 && i <= n)
    return i.val
```

(iv) def NextUnset(i):

```
if ( i >= 1 && i <= n)
    return find(i).ptr
```

In the disjoint set, each element will correspond to the index of the array of bits $x[1 \cdots n]$. Each disjoint set element is an object which has 2 fields named .val ~~and .ptr~~ which stores the value of the element corresponding to the array and .ptr which stores ~~the~~ / points to the next element which has a value of 0; including itself i.e if an element has 0 value .ptr will point to itself.

The intuition behind the approach is, ·pts will always point to the element which has a value of 0 including itself i.e if i is an element with value 0 then its ·ptr will point to itself and if not then its ·ptr will point to some greater index than i s.t that index's value is 0. Now whenever we are performing a union() when an element's value is set to 1 and we are doing it with the next element of i i.e we always do a union of i & i+1, thus ensuring (i+1)'s leader.ptr will always give me the index of element with value 0 and which will be greater than i. So whenever a new element is combined in a set we make use of the above fact and store that element with value 0 in the leader.pts of the newly formed set. So all the element with value set as 1 are combined together & s.t the leader of this set points to the next higher index or equal element with value 0.

First when Init(i) is called we make singleton sets and also we store the value of such elements as 0 (as per the definition of Init()) and point the pts pointer to itself cause as per the pts pointer definition above, we are marking / pointing pts to the next element which has 0 value including itself. These operations we do in (b) (i) as written above.

Now ~~when~~ when Set(i) is called, after checking the array ~~bound~~ size bounds, we first store the (i+1)'s leader. pts in a variable. We do so because i+1 > i & and even if (i+1) element is in a ~~set of~~ singleton set or not its leader ie find(i+1). ~~until~~ pts will always point to the next element having value 0 including itself. Now we do a union of i+1 and i element., thus ensuring contiguous bits of 1 are combined in one set. But again in this newly formed set, the leader ~~might~~ must point to the ~~next~~ element with higher index with value = 0

Thus we take find(i).pts = which is ~~find~~ leader(i).pts & store temp in this. temp will always ensure that an index of highest value is stored as temp ~~itself was~~ = find(i+1).pts where .pts can point to element itself or any element of a greater index. We do these operations in (b)(ii)

In (b)(iii) after checking the array size bounds, Is Set() simply returns the value stored in the ith element, where i is the passed parameter.

When NextUnset(i) is called, we simply check the array size bounds and return find(i).pts. If i is a singleton set ie i.val=0 then .pts will point to itself and thus it will return itself, which is true as per the definition of the NextInit(). Now if i is not a singleton set i.e it is a part of some other bigger set then its val must be equal to 1 as we perform a union only when we set value of element to 1. So, find(i) i.e leader of i's .pts value will point to the element of greater than equal to index whose value is 0. Thus we return find(i).pts and so it matches the return definition of NextUnset as per the question.

(c)

Init (9)

(1) ↻ .val = 0, .ptr = 1 ; (2) ↻ .val = 0, .ptr = 2 ; ... ; (9) ↻ .val = 0, .ptr = 9

where ~~arrow~~ arrow represents the leader.

IsSet (3)

will return 3.val which is 0

Set(4)

temp = find(i+1).ptr. —— (b)(ii)(a)

= find(5).ptr

= 5.ptr

= 5 // 5's leader and ptr are both itself

i.val = 1 —— (b)(ii)(b)

4.val = 1 // 4.val is updated to 1

(4) ↻ .val = 1, .ptr = 4

union(i+1, i) —— (b)(ii)(c)

union(5, 4) // (5) and (4) are combined in one set {5, 4} where leader is implementation dependent .val and .ptr remains unchanged for each element in this step.

find(i).ptr = temp —— (b)(ii)(d)

find(4).ptr ⇒ 4 ~~set~~ leader.ptr will now have value of temp = 5

Next Unset(4)

it will return find(i).pts as per (b)(iv) so,

⇒ find(i).pts

⇒ leader(4).pts

⇒ 5 (as mentioned in the last step of set(4)).

Thus 5 is returned.

Set(5)

temp = find(i+1).pts. —— (b)(ii)(a)

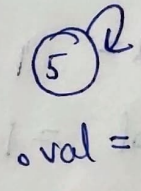
= find(6).pts

= 6.pts

= 6 // 6's leader and pts are both itself

i.val = 1 —— (b)(ii)(b)

5.val = 1 // 5.val is updated to 1



union(i+1, i) —— (b)(ii)(c)

union(6, 5) // 5 and 6 are now combined to form one set. i.e {4, 5, 6} are now combined into a single set. .val & .pts in this step remains unchanged for each element.

find(i).pts = temp. —— (b)(ii)(d)

find(5).pts ⇒ 5, sets leader.pts will now have value of temp = 6

leader is implementation dependant & can be any one of the above 3 elements.

NextUnset(4)

it will return find(i).ptr as per (b)(iv), so,

2) find(4).ptr

now 4 belongs to the set {4, 5, 6}

this sets leader.ptr has a value of 6 as mentioned in the last ~~8~~ step of the last operation.

Thus 6 will be returned.

Set(7)

temp = find(i+1).ptr — (b)(ii)(a)

= find(8).ptr

= 8 // 8's .ptr and leader is itself.

i.val = 1 — (b)(ii)(b)

~~7~~.val = 1 // 7.val is updated to 1

union(i+1, i) — (b)(ii)(c)

union(8, 7) // now 7 & 8 are combined in one set. leader is implementation dependant ie can be either of the two. .val & .ptr of each element remains unchanged in this step.

find(i).ptr = temp — (b)(ii)(d)

find(7).ptr => leader of 7's set .ptr will now have value temp = 8.

Set(8)

temp = find(i+1).ptr — (b) (ii) (a)

= find(9).ptr

= 9 // 9's .ptr and ~~not~~ leader are itself.

i.val = 1 — (b) (ii) (b)

8.val = 1 // 8.val is set to 1 now.

union(i+1, i) — (b) (ii) (c)

union(9, 8) // now 9 & 8 are combined into one set. ie {7, 8, 9} are combined into one set as 7 & 8 were already a part of one set. .val & .ptr of each element remains unchanged in this step.

find(i).ptr = temp — (b) (ii) (d)

find(8).ptr ⇒ leader of 8's set .ptr will now point to temp ie <u>temp = 9.</u>

leader is implementation dependant & thus can be any one of the 3 above elements.

IsSet(4)

will return 4.val which is 1; updated in Set(4) call.

Set(6)

temp = find(i+1).ptr — (b) (ii) (a)

=> find(7).ptr

=> 9 // 7 belongs to set {7,8,9} whose leader.ptr contains value of 9 as ~~dis~~ mentioned in last step of set(8) call.

i.val = 1 — (b) (ii) (b)

6.val = 1 // 6.val is set to 1.

union(i+1, i) — (b) (ii) (c)

union(7, 6) // 7 & 6 are combined into one set. ie 7 belonged to set {7,8,9} and 6 belonged to set {4,5,6} thus these 2 sets are joined / combined into {4,5,6,7,8,9} where leader is implementation dependant. .val and .ptr of each element remains unchanged in this step.

find(i).ptr = temp — (b) (ii) (d)

find(6).ptr => 6's set's leader which can be any one of the above element, so .ptr value ie leader.ptr value will be temp which is 9

Next Unset (4)

it will return find(i).pts as per (b)(iv), so

2) find(4).pts.

now 4 belongs to the set {4,5,6,7,8,9} whose leader.pts has value of 9 (as mentioned in the last step of last function call).

Thus it will return 9

(d) Time complexity.

Direct (as in (a))

already mentioned in (a)

Reversed tree.

Init (i): $\theta(n)$

Set (i): $\theta(n)$

Is Set (i): $\theta(1)$

Next Unset (i): $\theta(n)$.

Reverse tree + union by depth

Init (i): $\theta(n)$

Set (i): $O(\log n)$

Is Set (i): $\theta(1)$

Next Unset (i): $O(\log n)$

Shallow tree + threading

Init (i): $\theta(n)$

Set (i): $\theta(n)$ on average

Is Set (i): $\theta(1)$

Next Unset (i): $\theta(1)$

Shallow tree + threading + union by size

Init (i): $\theta(n)$

Set (i): $O(\log n)$

Is Set (i): $\theta(1)$

Next Unset (i): $\theta(1)$