```
int main()
{ enqueue(10);
  enqueue(20);
  enqueue(30);
  enqueue(40);
  enqueue(50);
  enqueue(60);
  dequeue();
  dequeue();
  dequeue();
  dequeue();
  dequeue();
  dequeue();
  return 0;
}
```

O/P

```
Error Q is full
10
20
30
40
50
Q is empty
```

Practical Application of Queue.

x Processes coming to processor → h/w

* ↳ ready queue

↳ process which are ready for execution

× printing
↳ SPOOL - Simultaneous Peripheral Operations Online
↳ Printer

6/9/2021

Dynamic Memory Allocation

× static memory: memory allocated at compile time
× dynamic memory: memory allocated at execution time

reallocation, forward reference.

× declarative stmt
× assignment stmt
× I/P O/P stmt
× execution stmt
× compilation stmt
× conditional stmt (if)
× iterative stmt (for)

→ dynamic memory allocation: memory allocated during run time.

→ since one of the stmts is the pgm allocates the memory, its the responsibility of pgmmer to release the memory location.

dynamic memory allocation functions

① malloc()

* compile time memory released by the s/m itself
* (datatype *) malloc(sizeof(datatype));
* Eg: int *p;
p=(int *) malloc(sizeof(int));

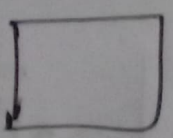
OR

p=(int *) malloc(sizeof(2));

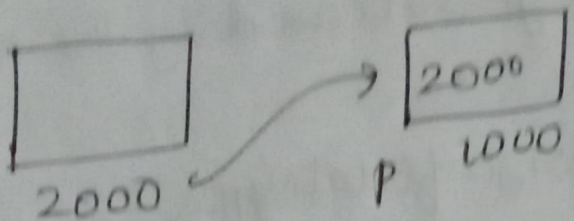
② free():

Program

```
int main()
{
    int *p;
    p=(int *) malloc(sizeof(int));
    scanf("%d", p);
    printf("%d", *p);
    free(p);
  return 0;
}
```

① 

② create a new memory allocation equal to size of int and provide a pointer to the location to pointer p



garbage value

int a; [123] 1000

&a = 1000

a = 123

(3) &p = 1000 ; p = 2000

contains address of another memory location

no & because value should go to address 2000 pointed by

(4) *p = value at address to p

123 ⇒ printf("%d", *p);

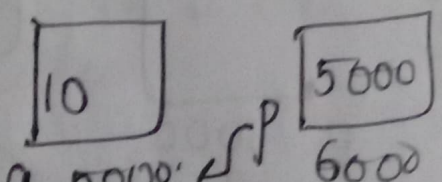
(5) free(p); // free(2000);

free(&p) // free(1000);

2nd stmt memory allocated, (5) memory deallocated.

no stmt to release memory location 1000. because compiler is allocating the memory hence deallocated by s/m itself.

```
void main()
{
  int a; int *p;
  a = 10; p = &a;
  printf("%d", *p); // 10
}
```



↳ ptr pointing to already allocated memory location

free(&p);// inconsistency or memory leak or bad sectors

x malloc for allocating array memory locations

```
int main()
{
  int *p,n,i;
   printf("Enter n:");
   scanf("%d",&n);
p=(int*)malloc(sizeof(int)*n);
for(i=0;i<n;i++)
  {
   printf("Enter p[%d]",i);
   scanf("%d",(p+i));
  }
for(i=0;i<n;i++)
  {
   printf("%d",*(p+i));
  }
  free(p);
 return 0;
}
```

interview or viva

output

Enter n 4

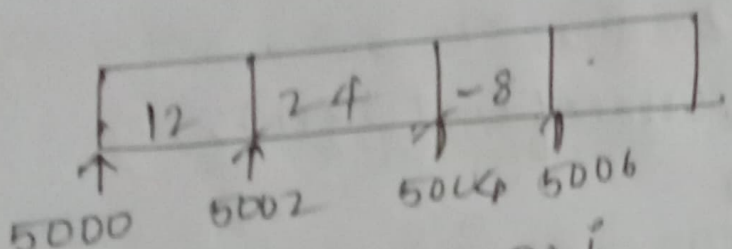
Enter p[0] 12

Enter p[1] 23
Enter p[2] 34
Enter p[3] 45

12 23 34 45

p => starting address

p [5000] (1000) [4] (n 2000)



	p	i	p+i
~~i=0~~	5000	0	5000
	5000	1	5002 (next address)

(p+i) → pointer arithmetic

Interview: what is the diff b/w integer arithmetic and pointer arithmetic

float *f;
f = (float *) malloc (sizeof(float)*5);

	f	p+f
p	0	5000
5000	1	5004
5000	2	5008
5000		

Interview Give an eg in which integer arithmetic is diff from ptr arithmetic

no & symbol in scanf ("%d", p+i);
p is a pointer, in printf we give * to get value

*(5000) = 12

*(5000+1) = *(5002) = 24

free(p); // release memory location by specifying starting address

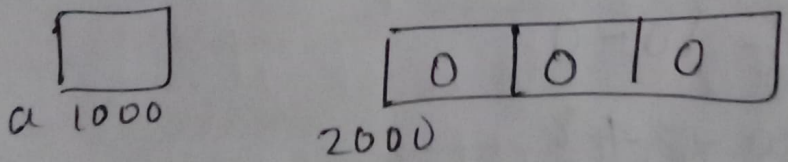
1 byte = 8 bit.

4 bit = nibble

③ calloc()

(datatype *) calloc (sizeof(datatype), noof location);

Eg: int *a;
a = (int *) calloc(sizeof(int), 3);



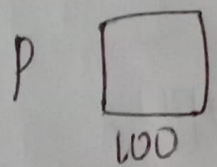
* In calloc, all memory locations are initialized with zero

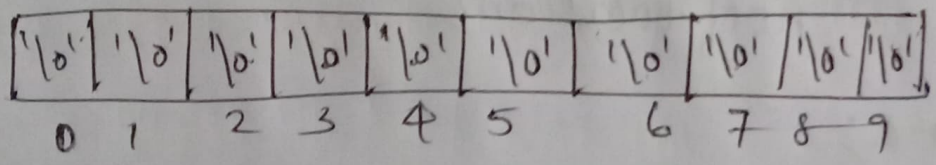
Interview whats the diff b/w malloc and calloc?

* Malloc, memory locations have garbage value.
* calloc have two arguements
* malloc have one arguements
* For exam, write example of separate foreach

Eg: char *p;

p=(char *)calloc(sizeof(char),10);





★ Why initalise with \0?

? ASCII value of \0 - 0

★ ASCII value of 0 - 48

11/10/2021

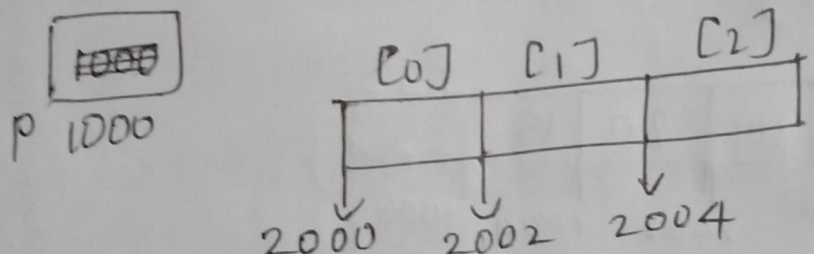
④ realloc():

Interview | malloc() , calloc()

dynamic memory allocation function used to expand or shrink memory location already done by pointer variable using malloc and calloc()

```
int *p;
p=(int *)malloc(sizeof(int)*3);
```

2×3=6 bytes



I) Increasing memory - from 3 to 5

```
p=(int *)realloc(p, sizeof(int)*5);
```

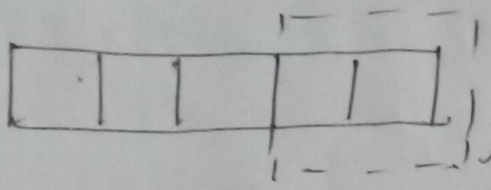
p ⇒ increase the memory location where p is pointing to

3 locations already present → 3×2=6 bytes

↓ increase to

5 locations → 5×2=10 bytes } 4 bytes

10-6=4 bytes additionally allocated or 2 int locations



Issues in this case of ↑ing memory location

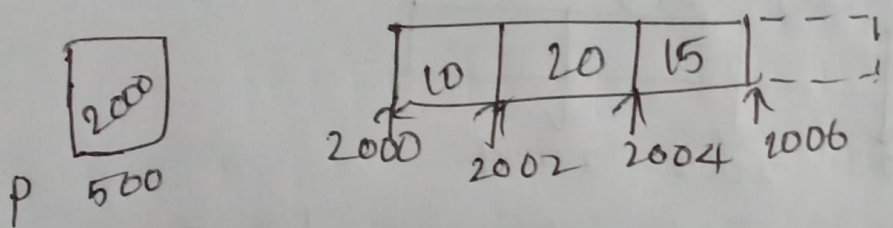
Consider: int *p, *q;

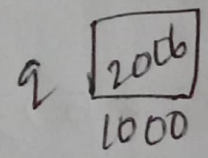
p = (int *) malloc (sizeof (int) * 3);

q = (int *) malloc (sizeof (int) * 2);

*p = 10; *(p+1) = 20;

*(p+2) = 15;





p = (int *) realloc (p, sizeof (int) * 5);

↳ it is not able because q is allocated continuously to p.

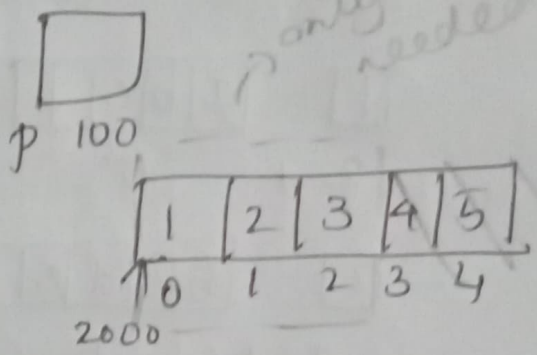
× new space should be created and the values should be copied and the new memory's starting address should be stored in pointer and older memory location should be released

(9) Explain working of realloc

* in realloc if possible continuously memory will be allocated.

II) Decrease Memory location

```
int *p;
p=(int *)calloc(sizeof(int),5);
for(i=0;i<5;i++)
  { *(p+i)=i+1;
  }
```




```
p=(int *)realloc(p,6);
```

here size is mentioned

Locations : 5

size : 5 x 2 = 10 bytes

(p,6) : 2 x 3 = 6 bytes

10 → 6 (bytes)

5 → 3 (locations number)

memory released from last automatically

EG Dynamic Memory Allocation

→ defn

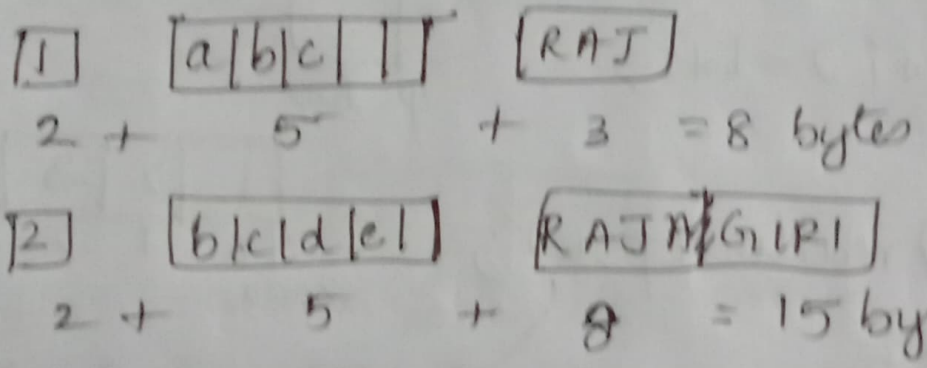
→ purpose

→ fns used

→ explain each fn
→ realloc
  ↳ expand
  ↳ shrinks
→ diff b/w malloc & calloc
→ pgm with each

LR

no integer
name char(6)
address varchar(10)



varchar — used only memory locations which is available

char — irrespective of the value size is fixed (what declared initially)

Implementation of VARCHAR

12/10/2021

Priority Queue

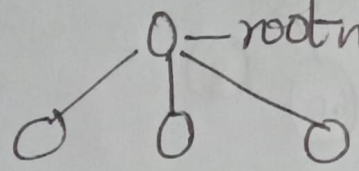
Implement using Arrays.

Heap ppty, Heap Sort, Heap Tree

LQ Implement heap sort
Implement priority queue } Answer is same

Heap Tree

- Tree has root node and children



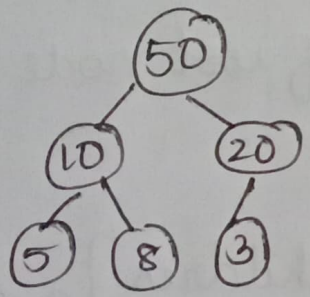
⇒ General Tree

child node

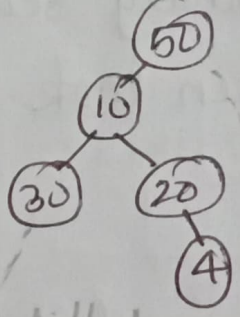
Binary Tree

- A tree having two children or almost two children is called binary tree

Complete Binary Tree



↳ Complete Binary Tree

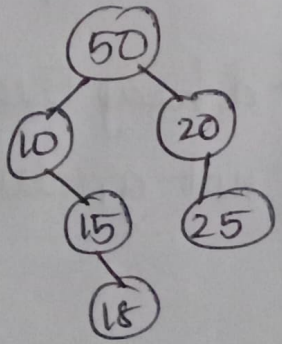


not filled

⇒ Not a complete binary tree

not filled

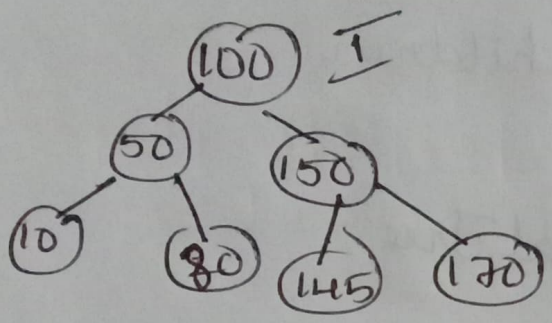
↳ all levels should be filled

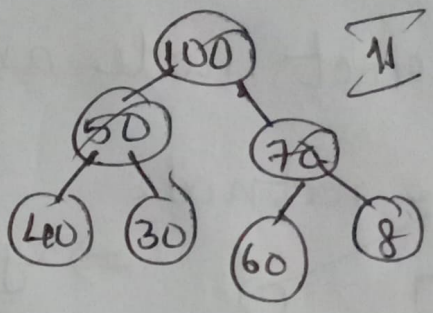


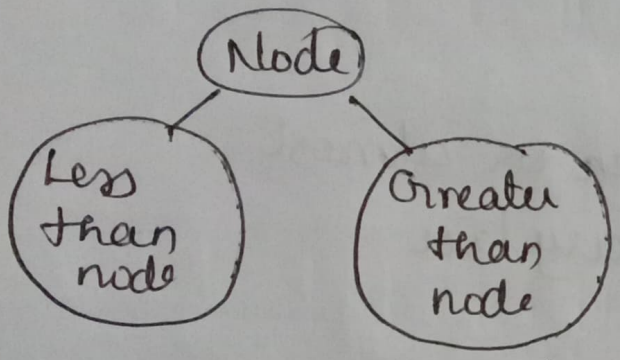
⇒ not a complete binary tree

★ Heap tree is always a complete binary tree

☆ A binary is a heap tree if it satisfies the heap property.







☆ i.e left node < root node
and
right node > root node

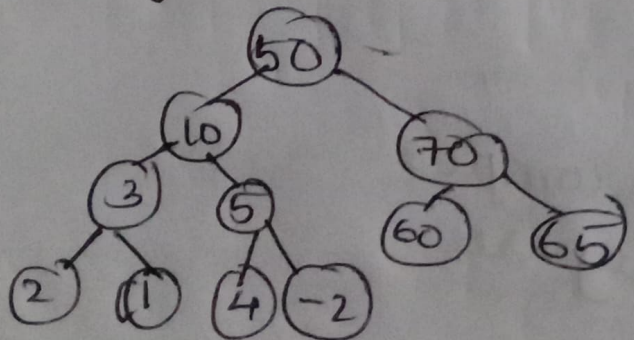
I example for binary search tree

↳ search for 75, then look left of root node

II ⇒ Heap Tree

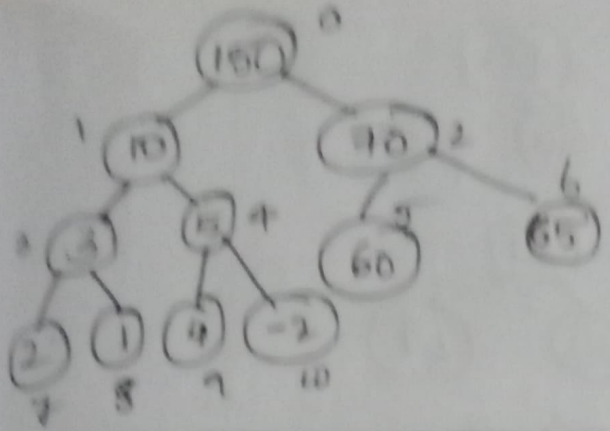
- Value of root node > value of its children (Here no case like ☆)
- Every node should satisfy the above ppty

} Heap ppty



⇒ Not a heap tree

50 not greater than 70



Consider this as an array

100	10	70	3	5	60	65	2	1	4	-2
0	1	2	3	4	5	6	7	8	9	10

- If you are in i^{th} node, then left child of i^{th} node : $2 \times i + 1$
- Then right child of i^{th} node } $2 \times i + 2$

i	l	r
0	1	2
1	3	4
2	5	6

$i = 0$

$2 \times 0 + 1 = 1 = l$

$i = 1$

$2 \times 1 + 1 = 3 = l$

$i = 0$

$2 \times 0 + 2 = 2 = r$

$i = 1$

$2 \times 1 + 2 = 4 = r$

two times values can't be repeating

- Parent for i^{th} node : $(i-1)/2$

⇓

integer division

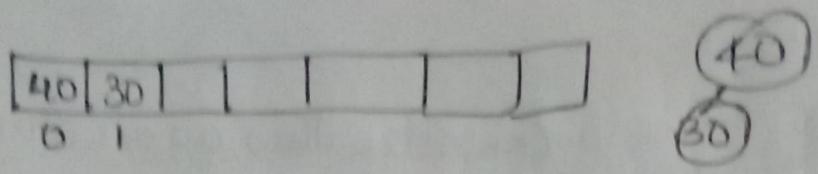
Create Heap Tree

(40)

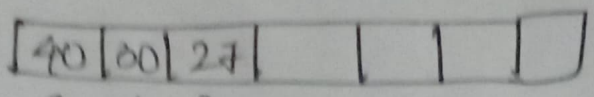
1) | 40 | | | | | | |

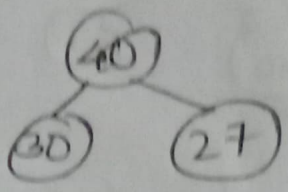
0

2)

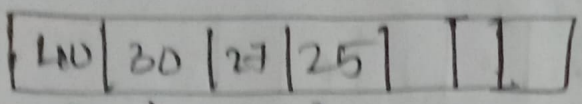


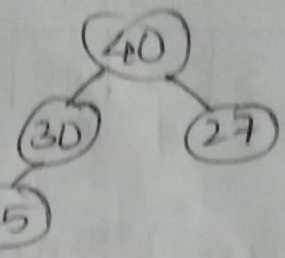
3)



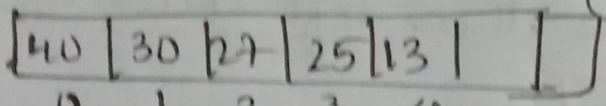


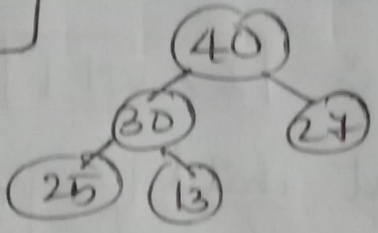
4)



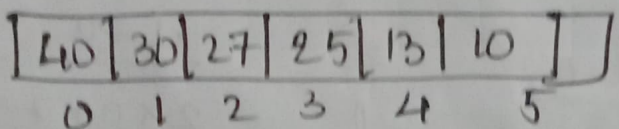


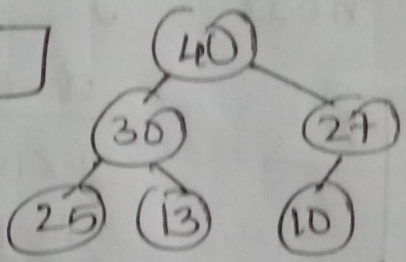
5)



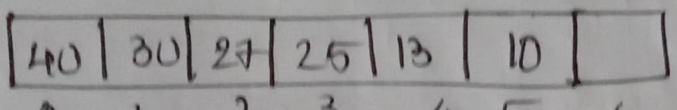


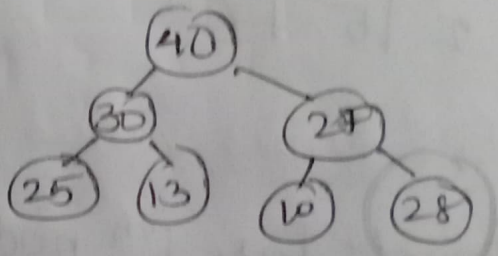
6)





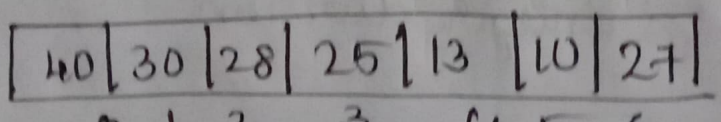
7)

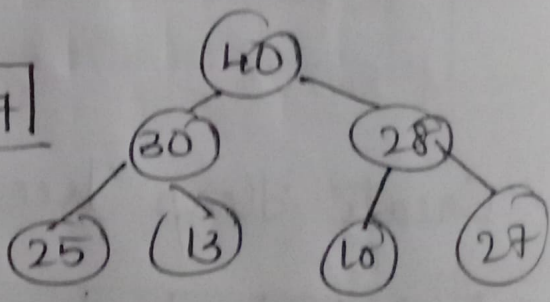


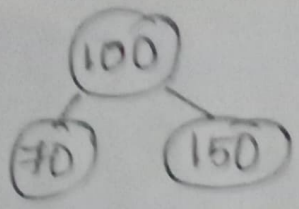


swap(2,6) ⇒ swap(27, 28)

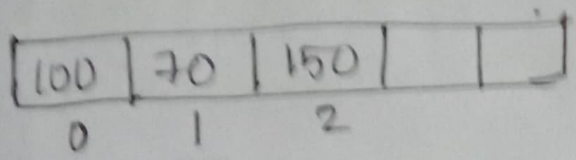
8)



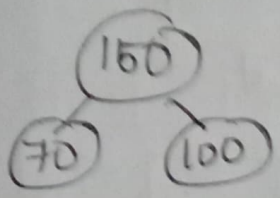


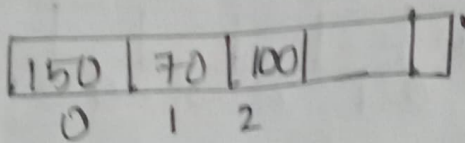
1) 

× 100<150
violated heap ppty

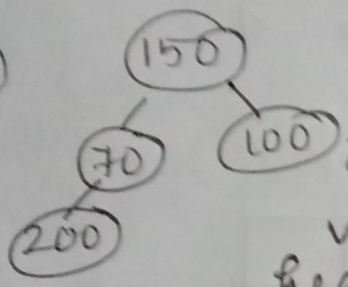


2) swap(100,150)

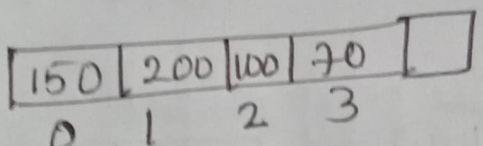




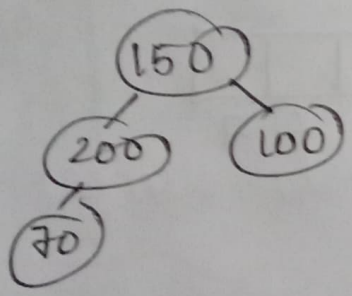
✓ heap tree

3) 

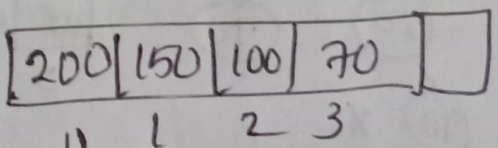
× 70<200
violated
heap ppty

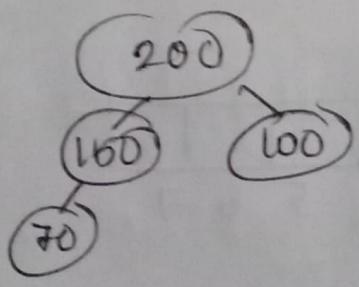


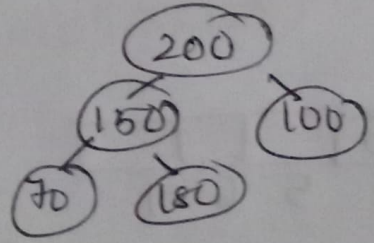
swap(70,200)

4) 

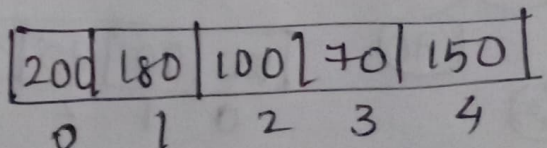
× 150<200
swap(150,200)

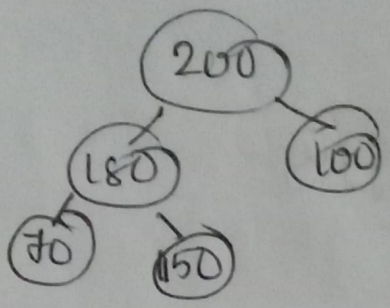




5) 

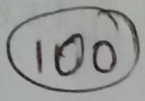
× swap(150,180)

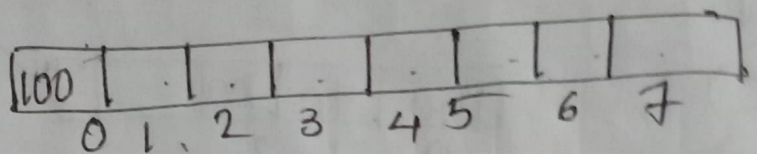


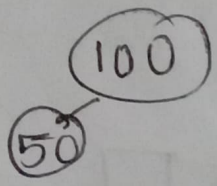


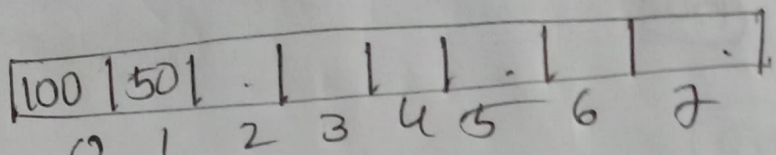
✓ Heap Tree

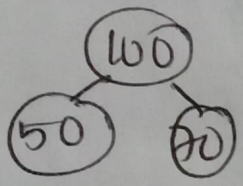
Q) 100, 50, 70, 90, 55, 200, 20, 150

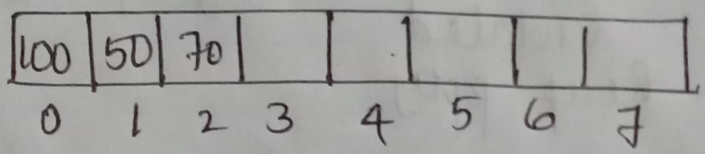


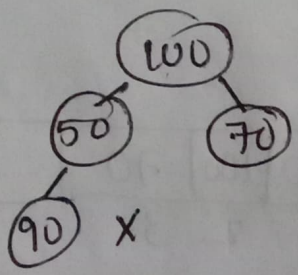


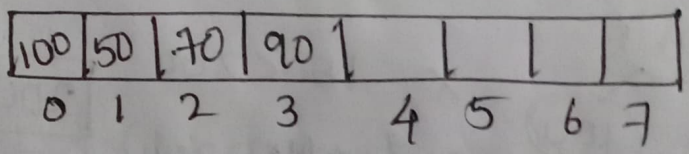




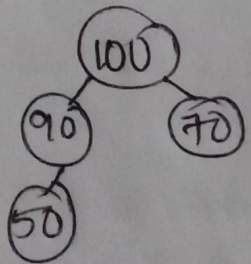


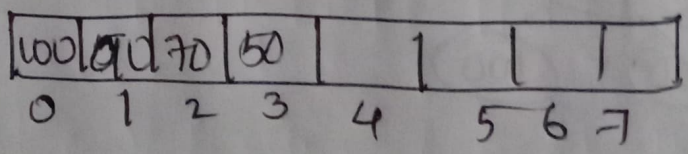


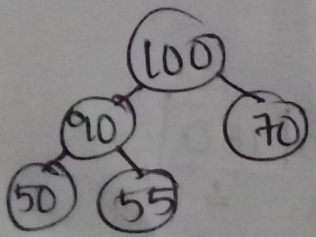


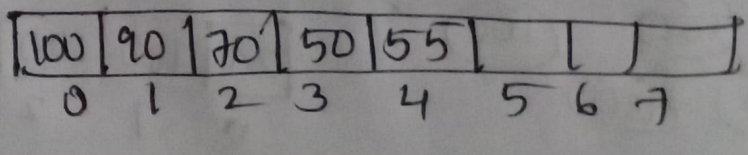


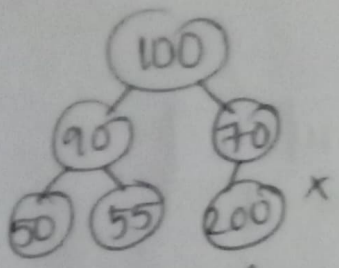
swap (50, 90)

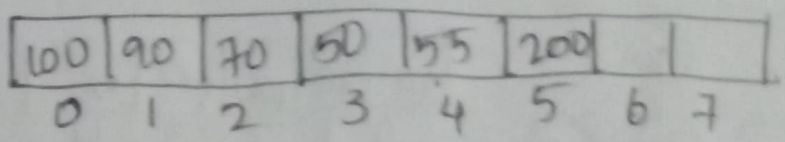




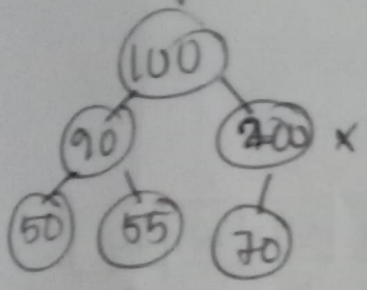


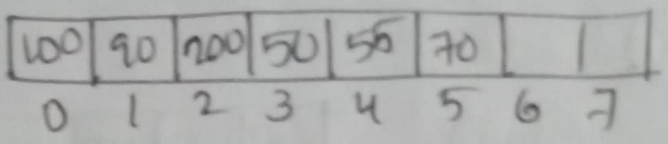




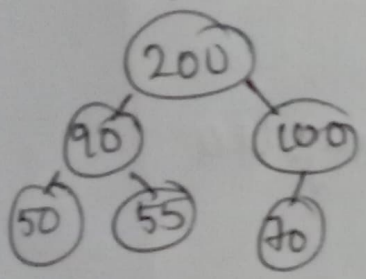


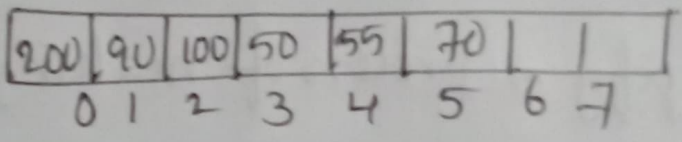
swap (70, 200)

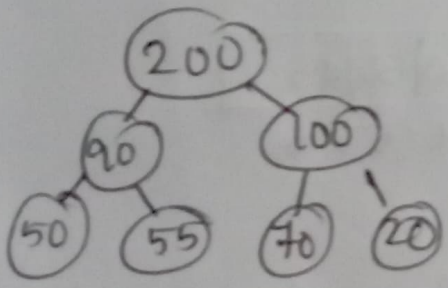


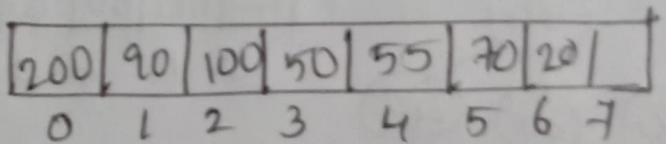


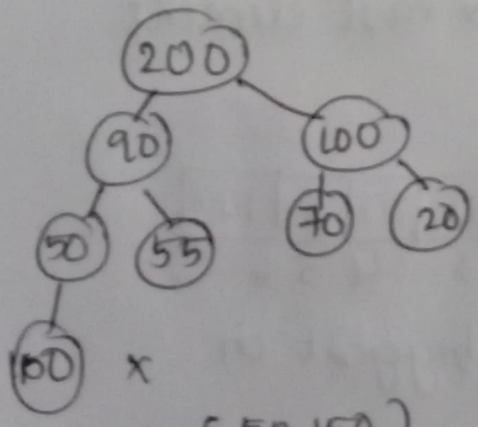
swap (100, 200)

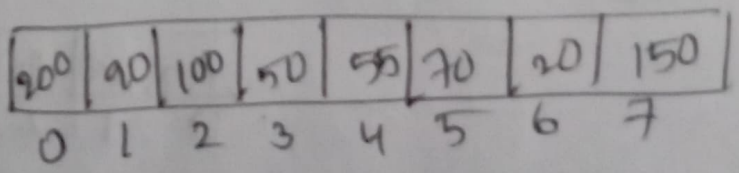




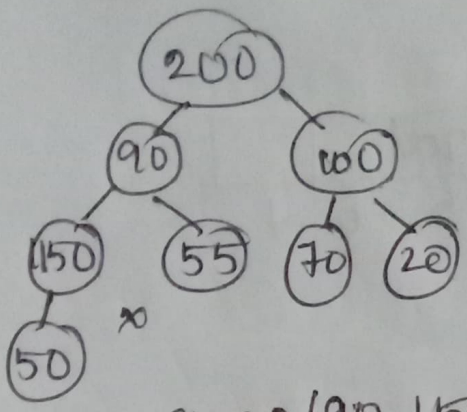






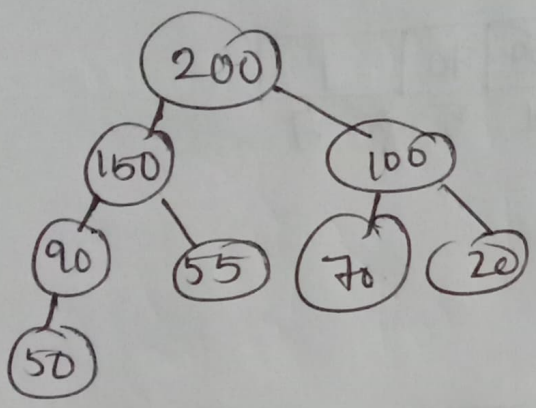


swap (50, 150)



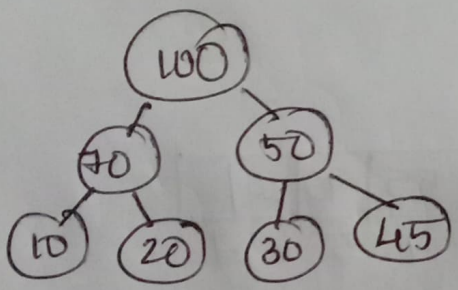
200	90	100	150	55	70	20	50
0	1	2	3	4	5	6	7

swap (90, 150)



200	150	100	90	55	70	20	50
0	1	2	3	4	5	6	7

- Heapify ⇒ process of making heap tree



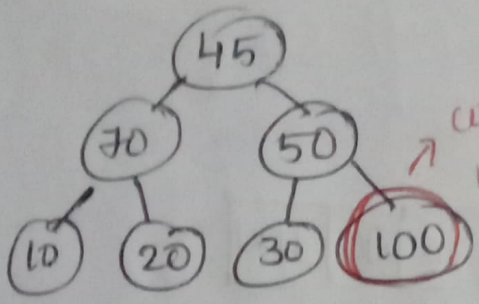
100	70	50	10	20	30	45
0	1	2	3	4	5	6

- Interchange root element & last element

swap (100, 45)

45	70	50	10	20	30	100
0	1	2	3	4	5	6

always value at root node is biggest in heap tree.

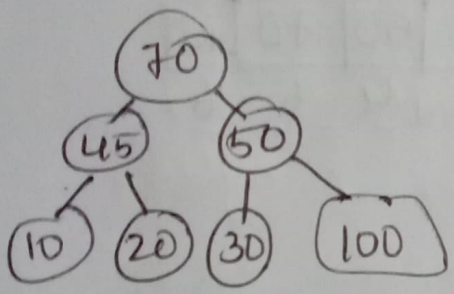


↗ correct place when array is sorted

45 not greater than 70 and 50.

Biggest child of root node is to be found

swap(45, 70)

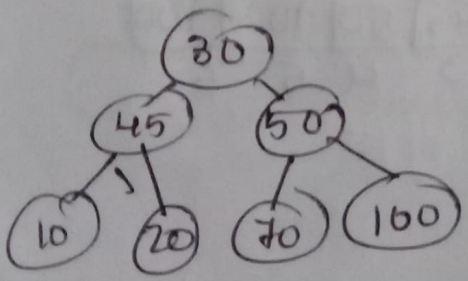


70	45	50	10	20	30	100
0	1	2	3	4	5	6

Now 100 in correct position when array is sorted and heapify done on rest of the elements.

Now 70 is the second largest element, second last position.

swap(70, 30) → with last element in heap (dont consider 100)



30	45	50	10	20	70	100
0	1	2	3	4	5	6

Rest is not a heap tree.

30 not greater than 45 & 50