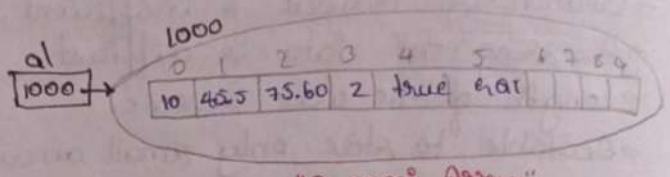
1) ArrayList class:



"Dynamic Array"

```
import java.util.ArrayList;
class Launch
{
    psvm(String[] args)
    {
        ArrayList al=new ArrayList();
        al.add(10);
        al.add(45.5f);
        al.add(75.60);
        al.add('Z');
        al.add(true);
        al.add("eQT");
        S.o.p(al);
    }
}
```

o/p: [10, 45.5, 75.60, 'Z', true, eQT]

Default capacity = 10

New capacity = 50% more than old capacity.
= old capacity + 50% of old capacity
= 10 + 5
= 15

New capacity = 15 + 7
= 22

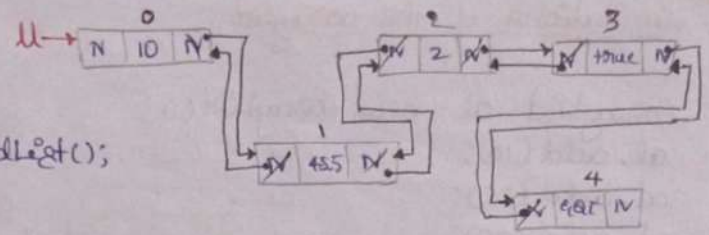
* ArrayList internally makes use of dynamic array.
* Hence, two of the limitations of the array approach namely,
  → size being fixed.
  → not being able to store heterogenous data, can be easily overcome using the ArrayList class as shown above.
* However, ArrayList still expects contiguous memory location on the RAM. as it makes use of dynamic array internally.

2) LinkedList class:

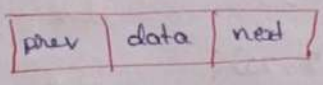
* LinkedList internally makes use of "doubly linked list" DS. Hence, it does not expect contiguous memory locations instead it can utilized dispersed memory location on the RAM (through pointers).
* Hence, all the limitations of the array DS can be overcome by using the LinkedList class.

```
import java.util.LinkedList;
class Launch
{
    psvm(String[] args)
    {
        LinkedList ll=new LinkedList();
        ll.add(10);
        ll.add(45.5f);
        ll.add('Z');
        ll.add(true);
        ll.add("eQT");
        S.o.p(ll);
    }
}
```

o/p: [10, 45.5, Z, true, eQT]



"Dubly Linked List"

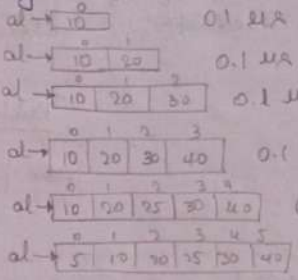


LinkedLists:
1) Singly LL
2) Doubly LL
3) CircularLL

Limitations of the arraylist:

03/5/23

```
ArrayList al = new ArrayList();
al.add(10);
al.add(20);
al.add(30);
al.add(40);
al.add(2,25);
al.add(0,5);
```



0.1 μs
0.1 μs
0.1 μs
0.1 μs
0.3 μs
0.6 μs

* ArrayList is extremely efficient at performing rear-end insertion. However it is inefficient at performing insertion at intermediary positions and it is extremely inefficient at performing front-end insertion.

* The limitation of front-end insertion being inefficient in ArrayList can be overcome using the ArrayDeque class.

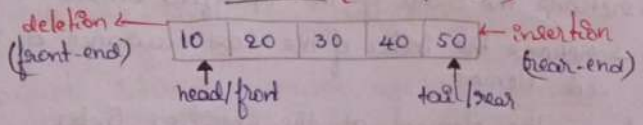
3. ArrayDeque class:

* ArrayDeque is internally makes use of double ended queue DS.
* Default capacity is 16. (increases as a power of 2)

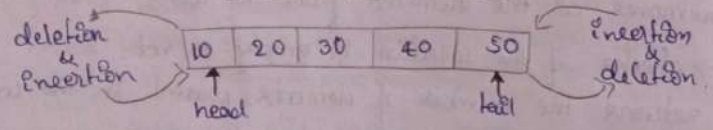
Array

FILO/[IFO × Random-insertion × LILO/
"Stack" × Random-deletion × FIFO "Queue"

Queue [FIFO / LILO]

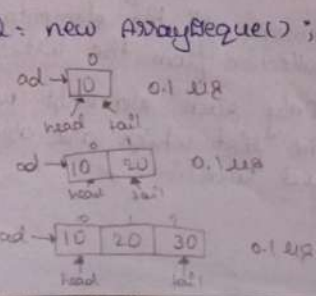


Double-ended Queue



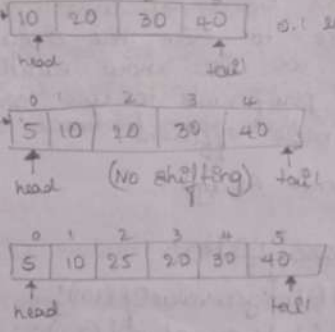
* A double-ended queue is equally efficient at performing insertion at both front-end as well as rear-end

```
ArrayDeque ad = new ArrayDeque();
ad.add(10);
ad.add(20);
ad.add(30);
```



0.1 μs
0.1 μs
0.1 μs

```
ad.addLast(40);
ad.addFirst(5);
ad.add(2,25);
```



0.1 μs
0.1 μs
0.4 μs

* As noticed, ArrayDeque is inefficient at performing insertion at intermediary positions.

* If insertion operation must be equally efficient at any given random position then LinkedList class can be used.

4) PriorityQueue class:

* PriorityQueue internally makes use of 'Min-Heap' DS.
* A PriorityQueue is used when the objects are supposed to be processed based on the priority.

Example:

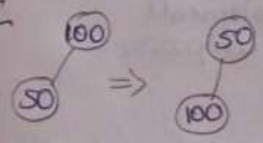
→ Task scheduling by OS.

→ Emergency rooms in a hospital etc.

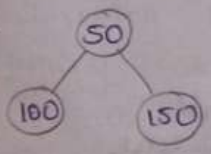
* In PriorityQueue, the highest priority object (least minimum) would be readily available at the front of the queue.

```
PriorityQueue pq = new PriorityQueue();
pq.add(100);
pq.add(50);
pq.add(150);
pq.add(25);
pq.add(75);
pq.add(125);
pq.add(175);
S.o.p(pq); //[25,50,125,100,75,150,175]
```

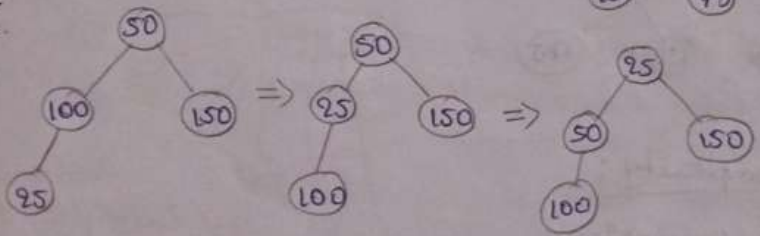
I.



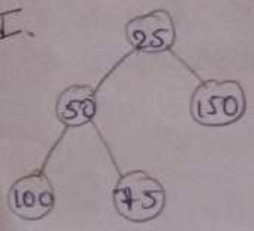
II.



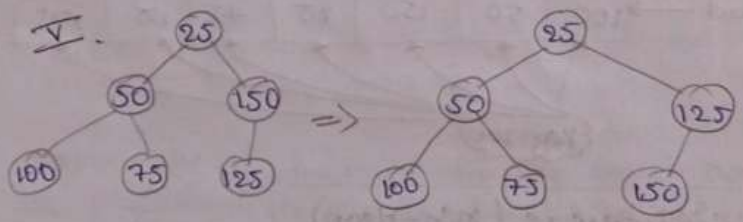
III.



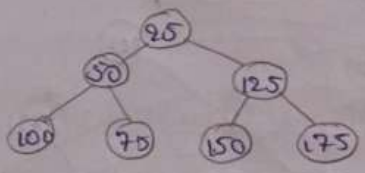
IV.



V.

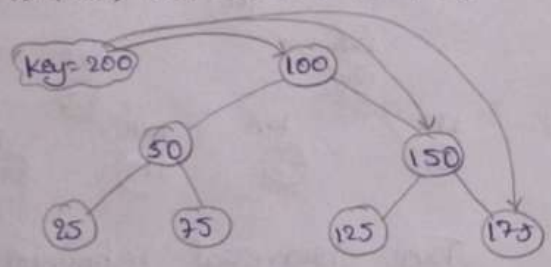


VI.



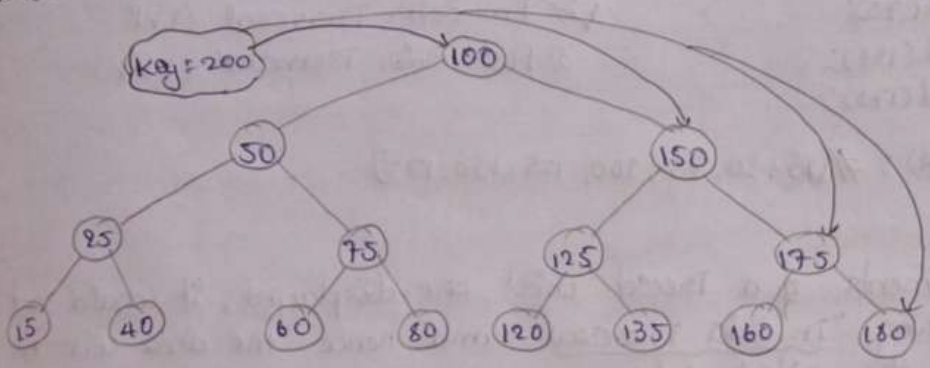
5). TreeSet [Balanced Binary Search Tree (BST)]:

Data: 100, 50, 150, 25, 75, 125, 175



7 - elements
3 - comparisons.

Data: 100, 50, 150, 25, 75, 125, 175, 15, 40, 60, 80, 120, 135, 160, 180



15 - elements
4 - comparisons

No. of elements

$7 \simeq 8 = 2^3$

$15 \simeq 16 = 2^4$

$65535 \simeq 65536 = 2^{16}$

n-elements

No. of comparisons

3 - comparisons

4 - comparisons

16 - comparisons

log n - comparisons

$n = 2^x$

Raise $\log_2$ on B.S.

$\log n = \log 2^x$

$\log n = x \cdot \log_2 2$

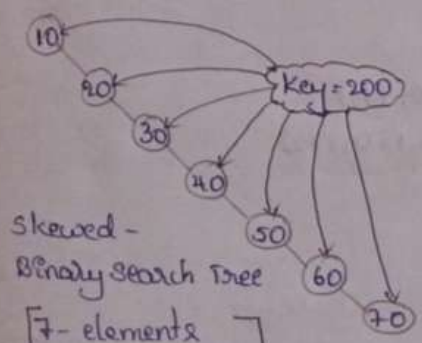
$\log n = x \cdot 1$

$\boxed{x = \log n}$

For n elements, $O(\log n)$

* TreeSet class internally makes use of Balanced Binary search tree DS. using "Red-Black Tree" algorithm. Hence it is efficient at performing search operations.

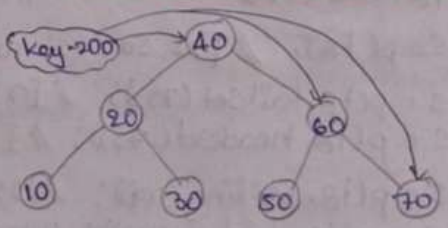
Data: 10, 20, 30, 40, 50, 60, 70



Skewed - Binary Search Tree

[7 - elements
7 - comparisons]

Red - Black Tree Algorithm

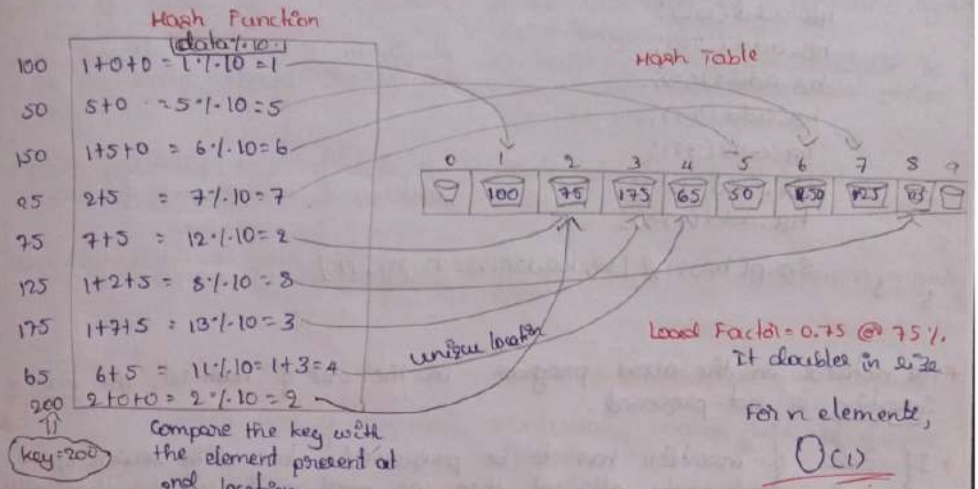


Balanced Binary search Tree

[7 - elements
3 - comparisons]

6) HashSet class [Hashing]:

Data : 100, 50, 150, 25, 75, 125, 175, 65



* Hashset class internally makes use of the Hashing algorithm. Hashing algorithm could contain a Hash Function and an associated Hash table with it.
* The duty of the Hash Function is to calculate the bucket location on the Hash table into which the data has to be stored.
* If two or more data is hashed to the same bucket location then we call it as collision of data.
* One of the responsibilities of the hash function is to calculate the bucket location in such a manner that collision does not occur.
* The Hash function would maintain a load factor which is 0.75 or 75%. As more and more data is stored in the hash table, the chances of collision increases and hence the movement the capacity of the hash table is filled to 75% of its capacity, automatically its size will be would be doubled.

Program:

```
import java.util.HashSet;
class Launch
{
    psvm(String[] args)
    {
        HashSet hs = new HashSet();
        hs.add(100);
        hs.add(50);
        hs.add(150);
        hs.add(25);
        hs.add(75);
        hs.add(125);
        hs.add(175);
        SO.p(hs); // [50, 100, 150, 25, 75, 125, 175]
    }
}
```

* As noticed in the above program, in the case of hash set, the order of insertion is not preserved.
* If order of insertion has to be preserved along with search operation being extremely efficient then we must make use of a class called the LinkedHashSet class.

Variable Approach

Disadv:

1. Creation is difficult
2. Accessing is difficult

Array Approach

Adv:

1. Creation is simple
2. Accessing is easy

Disadv:

1. Size is fixed
2. Homogeneous data
3. Contiguous memory locations on RAM

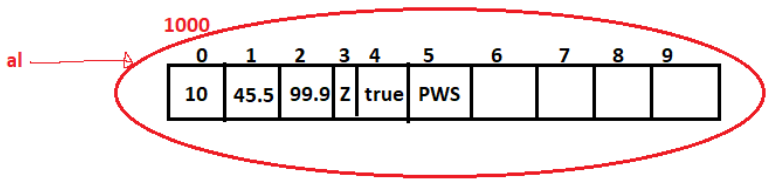
Collections Framework

1. ArrayList class - Dynamic Array
2. LinkedList class -
3. ArrayDeque class
4. PriorityQueue class
5. TreeSet class
6. HashSet class
7. LinkedHashSet class

Collections in Java

ArrayList class [Dynamic Array]

```
ArrayList al = new ArrayList();
    al.add(10);
    al.add(45.5f);
    al.add(99.9);
    al.add('Z');
    al.add(true);
    al.add("PWS");
S.o.p(al); //[10, 45, 99.9, Z, true, PWS]
```



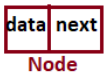
Default Initial Capacity = 10

New Capacity = 50% more than old capacity
= 10 + 5 = 15

New Capacity = 15 + 7 = 22

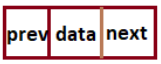
LinkedList class [Doubly Linked List]

Singly LL



Node

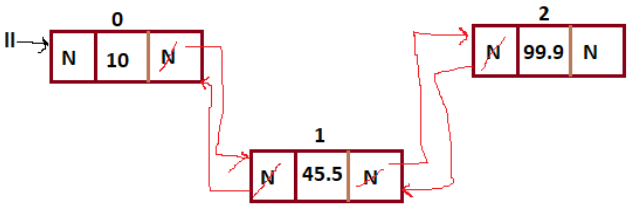
Doubly LL




```
LinkedList ll = new LinkedList();

ll.add(10);
ll.add(45.5f);
ll.add(99.9);
ll.add('Z);
ll.add(true);
ll.add("PWS");

S.o.p(ll);
//[10, 45.5, 99.9, Z, true, PWS]
```

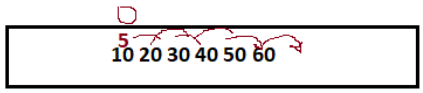


ArrayDeque [Double Ended Queue]

Front-end as well as rear end insertion should be efficient

```
ArrayDeque ad = new ArrayDeque();

ad.add(10);
ad.add(45.5f);
ad.add(5, 0);
```



ArrayList - Rear end insertion

```
al.add(60);  //0.1 Ms
             //0.5 Ms
al.add(5, 0); //0.7 Ms
```