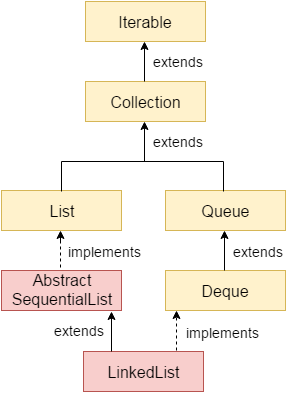
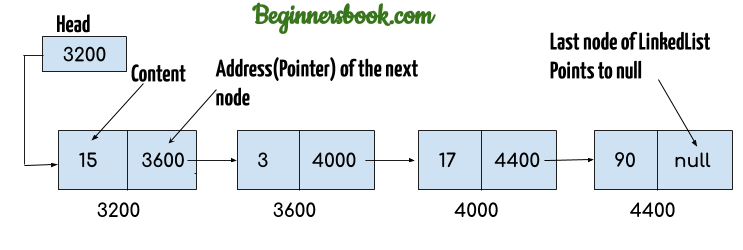
**Hierarchy of LinkedList class in Java**



**LinkedList representation**

Each element in the LinkedList is called the **Node**. Each Node of the LinkedList contains two items: 1) Content of the element 2) Pointer/Address/Reference to the Next Node in the LinkedList.

**This is how a LinkedList looks:**  


**Note:**  
1. **Head** of the LinkedList only contains the Address of the **First element** of the List.  
2. The Last element of the LinkedList contains **null** in the pointer part of the node because it is the end of the List so it doesn’t point to anything as shown in the above diagram.  
3. The diagram which is shown above represents a **singly linked list**. There is another complex type variation of LinkedList which is called **doubly linked list**, node of a doubly linked list contains three parts: 1) Pointer to the previous node of the linked list 2) content of the element 3) pointer to the next node of the linked list.

**Why do we need a Linked List?**

You must be aware of the arrays which is also a linear data structure but **arrays have certain limitations such as:**  
1) **Size of the array is fixed** which is decided when we create an array so it is hard to predict the number of elements in advance, if the declared size fall short then we cannot increase the size of an array and if we declare a large size array and do not need to store that many elements then it is a waste of memory.

2) Array elements **need contiguous memory locations** to store their values.

3) **Inserting an element in an array is performance wise expensive** as we have to shift several elements to make a space for the new element. For example:  
Let’s say we have an array that has following elements: 10, 12, 15, 20, 4, 5, 100, now if we want to insert a new element 99 after the element that has value 12 then we have to shift all the elements after 12 to their right to make space for new element.

Similarly **deleting an element** from the array is also a performance wise expensive operation because all the elements after the deleted element have to be shifted left.

**These limitations are handled in the Linked List by providing following features:**  
1. Linked list allows **dynamic memory allocation**, which means memory allocation is done at the run time by the compiler and we do not need to mention the size of the list during linked list declaration.

2. Linked list elements **don’t need contiguous memory locations** because elements are linked with each other using the reference part of the node that contains the address of the next node of the list.

3. Insert and delete operations in the Linked list are not performance wise expensive because adding and deleting an element from the linked list does’t require element shifting, only the pointer of the previous and the next node requires change.

**Java Linked List example of adding elements**

In the following example we are using add(), addFirst() and addLast() methods to add the elements at the desired locations in the LinkedList, there are several such useful methods in the LinkedList class which I have mentioned at the end of this article.

package com.beginnersbook;

import java.util.\*;

public class JavaExample{

public static void main(String args[]){

LinkedList<Stiring> list=new LinkedList<String>();

//Adding elements to the Linked list

list.add("Steve");

list.add("Carl");

list.add("Raj");

//Adding an element to the first position

list.addFirst("Negan");

//Adding an element to the last position

list.addLast("Rick");

//Adding an element to the 3rd position

list.add(2, "Glenn");

//Iterating LinkedList

Iterator<String> iterator=list.iterator();

while(iterator.hasNext()){

System.out.println(iterator.next());

}

}

}

## Java example of removing elements from the LinkedList

In the following example we are checking out the few popular **remove methods** in the LinkedList that are used to remove elements from certain positions in the LinkedList. Detailed explanation of these methods along with examples are covered in the separate tutorials, links are provided at the end of this article.

package com.beginnersbook;

import java.util.\*;

public class JavaExample{

public static void main(String args[]){

list<String> list=new LinkedList<String>();

//Adding elements to the Linked list

list.add("Steve");

list.add("Carl");

list.add("Raj");

list.add("Negan");

list.add("Rick");

//Removing First element

//Same as list.remove(0);

list.removeFirst();

//Removing Last element

list.removeLast();

//Iterating LinkedList

Iterator<String> iterator=list.iterator();

while(iterator.hasNext()){

System.out.print(iterator.next()+" ");

}

//removing 2nd element, index starts with 0

list.remove(1);

System.out.print("\nAfter removing second element: ");

//Iterating LinkedList again

Iterator<String> iterator2=list.iterator();

while(iterator2.hasNext()){

System.out.print(iterator2.next()+" ");

}

}

}

## Example of LinkedList in Java

import java.util.\*;

public class LinkedListExample {

public static void main(String args[]) {

/\* Linked List Declaration \*/

LinkedList<String> linkedlist = new LinkedList<String>();

/\*add(String Element) is used for adding

\* the elements to the linked list\*/

linkedlist.add("Item1");

linkedlist.add("Item5");

linkedlist.add("Item3");

linkedlist.add("Item6");

linkedlist.add("Item2");

/\*Display Linked List Content\*/

System.out.println("Linked List Content: " +linkedlist);

/\*Add First and Last Element\*/

linkedlist.addFirst("First Item");

linkedlist.addLast("Last Item");

System.out.println("LinkedList Content after addition: " +linkedlist);

/\*This is how to get and set Values\*/

Object firstvar = linkedlist.get(0);

System.out.println("First element: " +firstvar);

linkedlist.set(0, "Changed first item");

Object firstvar2 = linkedlist.get(0);

System.out.println("First element after update by set method: " +firstvar2);

/\*Remove first and last element\*/

linkedlist.removeFirst();

linkedlist.removeLast();

System.out.println("LinkedList after deletion of first and last element: " +linkedlist);

/\* Add to a Position and remove from a position\*/

linkedlist.add(0, "Newly added item");

linkedlist.remove(2);

System.out.println("Final Content: " +linkedlist);

}

}

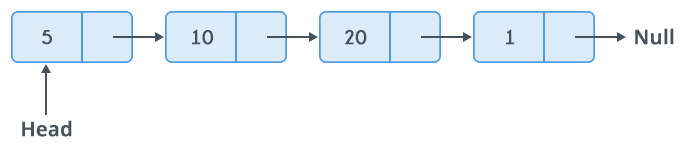
A **linked list** is a way to store a collection of elements. Like an array these can be character or integers. Each element in a linked list is stored in the form of a **node**.

**Node**:



A node is a collection of two sub-elements or parts. A **data** part that stores the element and a **next** part that stores the link to the next node.

**Linked List**:



A linked list is formed when many such nodes are linked together to form a chain. Each node points to the next node present in the order. The first node is always used as a reference to traverse the list and is called **HEAD**. The last node points to **NULL**.

**Why Linked List?**  
Arrays can be used to store linear data of similar types, but arrays have the following limitations.  
**1)** The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.  
**2)** Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.

**Advantages over arrays**  
**1)** Dynamic size  
**2)** Ease of insertion/deletion

**Drawbacks:**  
**1)** Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation. Read about it [here](https://www.geeksforgeeks.org/binary-search-on-singly-linked-list/" \t "_blank).  
**2)** Extra memory space for a pointer is required with each element of the list.  
**3)** Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

**Representation:**  
A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head is NULL.  
Each node in a list consists of at least two parts:  
1) data  
2) Pointer (Or Reference) to the next node

A **D**oubly **L**inked **L**ist (DLL) contains an extra pointer, typically called previous pointer, together with next pointer and data which are there in singly linked list.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/gq/2014/03/DLL1.png)

**Advantages over singly linked list**  
**1)** A DLL can be traversed in both forward and backward direction.  
**2)** The delete operation in DLL is more efficient if pointer to the node to be deleted is given.  
**3)**We can quickly insert a new node before a given node.  
In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

**Disadvantages over singly linked list**  
**1)** Every node of DLL Require extra space for an previous pointer. It is possible to implement DLL with single pointer though (See [this](https://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-1/" \t "_blank)and [this](https://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-2/" \t "_blank)).  
**2)** All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with next pointers. For example in following functions for insertions at different positions, we need 1 or 2 extra steps to set previous pointer.

**Insertion**  
A node can be added in four ways  
**1)**At the front of the DLL  
**2)** After a given node.  
**3)** At the end of the DLL  
**4)** Before a given node.

Vector implements List Interface. Like ArrayList it also maintains insertion order but it is rarely used in non-thread environment as it is synchronized and due to which it gives poor performance in searching, adding, delete and update of its elements.

#### Three ways to create vector class object:

**Method 1:**

Vector vec = new Vector();

It creates an empty Vector with the default initial capacity of 10. It means the Vector will be re-sized when the 11th elements needs to be inserted into the Vector. Note: By default vector doubles its size. i.e. In this case the Vector size would remain 10 till 10 insertions and once we try to insert the 11th element It would become 20 (double of default capacity 10).

**Method 2:**  
Syntax: Vector object= new Vector(int initialCapacity)

Vector vec = new Vector(3);

It will create a Vector of initial capacity of 3.

**Method 3:**  
Syntax:

Vector object= new vector(int initialcapacity, capacityIncrement)

Example:

Vector vec= new Vector(4, 6)

Here we have provided two arguments. The initial capacity is 4 and capacityIncrement is 6. It means upon insertion of 5th element the size would be 10 (4+6) and on 11th insertion it would be 16(10+6).

## Complete Example of Vector in Java:

import java.util.\*;

public class VectorExample {

public static void main(String args[]) {

/\* Vector of initial capacity(size) of 2 \*/

Vector<String> vec = new Vector<String>(2);

/\* Adding elements to a vector\*/

vec.addElement("Apple");

vec.addElement("Orange");

vec.addElement("Mango");

vec.addElement("Fig");

/\* check size and capacityIncrement\*/

System.out.println("Size is: "+vec.size());

System.out.println("Default capacity increment is: "+vec.capacity());

vec.addElement("fruit1");

vec.addElement("fruit2");

vec.addElement("fruit3");

/\*size and capacityIncrement after two insertions\*/

System.out.println("Size after addition: "+vec.size());

System.out.println("Capacity after increment is: "+vec.capacity());

/\*Display Vector elements\*/

Enumeration en = vec.elements();

System.out.println("\nElements are:");

while(en.hasMoreElements())

System.out.print(en.nextElement() + " ");

}

}

**Points to Note about HashSet:**

1. HashSet doesn’t maintain any order, the elements would be returned in any random order.
2. HashSet doesn’t allow duplicates. If you try to add a duplicate element in HashSet, the old value would be overwritten.
3. HashSet allows null values however if you insert more than one nulls it would still return only one null value.
4. HashSet is non-synchronized.
5. The iterator returned by this class is fail-fast which means iterator would throw ConcurrentModificationException if HashSet has been modified after creation of iterator, by any means except iterator’s own remove method.

**HashSet Example**

import java.util.HashSet;

public class HashSetExample {

public static void main(String args[]) {

// HashSet declaration

HashSet<String> hset = new HashSet<String>();

// Adding elements to the HashSet

hset.add("Apple");

hset.add("Mango");

hset.add("Grapes");

hset.add("Orange");

hset.add("Fig");

//Addition of duplicate elements

hset.add("Apple");

hset.add("Mango");

//Addition of null values

hset.add(null);

hset.add(null);

//Displaying HashSet elements

System.out.println(hset);

}

}

TreeSet is similar to [HashSet](https://beginnersbook.com/2013/12/hashset-class-in-java-with-example/" \t "_blank) except that it sorts the elements in the ascending order while HashSet doesn’t maintain any order. TreeSet allows null element but like HashSet it doesn’t allow. Like most of the other collection classes this class is also not synchronized, however it can be synchronized explicitly like this: SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));

## TreeSet Example:

In this example we have two TreeSet (TreeSet<String> & TreeSet<Integer>). We have added the values to both of them randomly however the result we got is sorted in ascending order.

import java.util.TreeSet;

public class TreeSetExample {

public static void main(String args[]) {

// TreeSet of String Type

TreeSet<String> tset = new TreeSet<String>();

// Adding elements to TreeSet<String>

tset.add("ABC");

tset.add("String");

tset.add("Test");

tset.add("Pen");

tset.add("Ink");

tset.add("Jack");

//Displaying TreeSet

System.out.println(tset);

// TreeSet of Integer Type

TreeSet<Integer> tset2 = new TreeSet<Integer>();

// Adding elements to TreeSet<Integer>

tset2.add(88);

tset2.add(7);

tset2.add(101);

tset2.add(0);

tset2.add(3);

tset2.add(222);

System.out.println(tset2);

}

}

## HashSet vs TreeSet

1) [HashSet](https://beginnersbook.com/2013/12/hashset-class-in-java-with-example/" \o "HashSet Class in Java with example" \t "_blank) gives better performance (faster) than [TreeSet](https://beginnersbook.com/2013/12/treeset-class-in-java-with-example/" \o "TreeSet Class in Java with example" \t "_blank) for the operations like add, remove, contains, size etc. HashSet offers constant time cost while TreeSet offers log(n) time cost for such operations.

2) HashSet does not maintain any order of elements while TreeSet elements are sorted in ascending order by default.

**Similarities**:

1) Both HashSet and TreeSet does not hold duplicate elements, which means both of these are duplicate free.

2) If you want a sorted Set then it is better to add elements to HashSet and then [convert it into TreeSet](https://beginnersbook.com/2014/08/how-to-convert-a-hashset-to-a-treeset/" \t "_blank" \o "How to convert a HashSet to a TreeSet) rather than creating a TreeSet and adding elements to it.

3) Both of these classes are non-synchronized that means they are not thread-safe and should be synchronized explicitly when there is a need of thread-safe operations.

**Examples:**

#### HashSet example

import java.util.HashSet;

class HashSetDemo{

public static void main(String[] args) {

// Create a HashSet

HashSet<String> hset = new HashSet<String>();

//add elements to HashSet

hset.add("Abhijeet");

hset.add("Ram");

hset.add("Kevin");

hset.add("Singh");

hset.add("Rick");

// Duplicate removed

hset.add("Ram");

// Displaying HashSet elements

System.out.println("HashSet contains: ");

for(String temp : hset){

System.out.println(temp);

}

}

}

#### TreeSet example

import java.util.TreeSet;

class TreeSetDemo{

public static void main(String[] args) {

// Create a TreeSet

TreeSet<String> tset = new TreeSet<String>();

//add elements to TreeSet

tset.add("Abhijeet");

tset.add("Ram");

tset.add("Kevin");

tset.add("Singh");

tset.add("Rick");

// Duplicate removed

tset.add("Ram");

// Displaying TreeSet elements

System.out.println("TreeSet contains: ");

for(String temp : tset){

System.out.println(temp);

}

}

}

1. HashSet doesn’t maintain any kind of order of its elements.
2. TreeSet sorts the elements in ascending order.
3. LinkedHashSet maintains the insertion order. Elements gets sorted in the same sequence in which they have been added to the Set.

**Example of LinkedHashSet:**

import java.util.LinkedHashSet;

public class LinkedHashSetExample {

public static void main(String args[]) {

// LinkedHashSet of String Type

LinkedHashSet<String> lhset = new LinkedHashSet<String>();

// Adding elements to the LinkedHashSet

lhset.add("Z");

lhset.add("PQ");

lhset.add("N");

lhset.add("O");

lhset.add("KK");

lhset.add("FGH");

System.out.println(lhset);

// LinkedHashSet of Integer Type

LinkedHashSet<Integer> lhset2 = new LinkedHashSet<Integer>();

// Adding elements

lhset2.add(99);

lhset2.add(7);

lhset2.add(0);

lhset2.add(67);

lhset2.add(89);

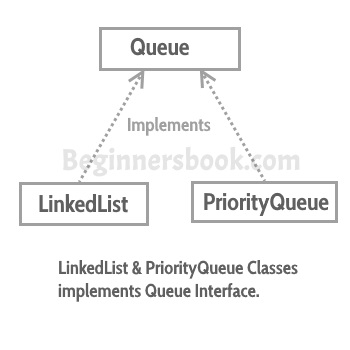
lhset2.add(66);

System.out.println(lhset2);

}

}

A **Queue** is designed in such a way so that the elements added to it are placed at the end of Queue and removed from the beginning of Queue. The concept here is similar to the queue we see in our daily life, for example, when a new iPhone launches we stand in a queue outside the apple store, whoever is added to the queue has to stand at the end of it and persons are served on the basis of FIFO (First In First Out), The one who gets the iPhone is removed from the beginning of the queue.

  
Queue interface in Java collections has two implementation: LinkedList and PriorityQueue, these two classes implements Queue interface.  
**Queue is an interface** so we cannot instantiate it, rather we create instance of LinkedList or PriorityQueue and assign it to the Queue like this:

Queue q1 = new LinkedList();

Queue q2 = new PriorityQueue();

## Java Queue Example

import java.util.\*;

public class QueueExample1 {

public static void main(String[] args) {

/\*

\* We cannot create instance of a Queue as it is an

\* interface, we can create instance of LinkedList or

\* PriorityQueue and assign it to Queue

\*/

Queue<String> q = new LinkedList<String>();

//Adding elements to the Queue

q.add("Rick");

q.add("Maggie");

q.add("Glenn");

q.add("Negan");

q.add("Daryl");

System.out.println("Elements in Queue:"+q);

/\*

\* We can remove element from Queue using remove() method,

\* this would remove the first element from the Queue

\*/

System.out.println("Removed element: "+q.remove());

/\*

\* element() method - this returns the head of the

\* Queue. Head is the first element of Queue

\*/

System.out.println("Head: "+q.element());

/\*

\* poll() method - this removes and returns the

\* **head** of the Queue. Returns null if the Queue is empty

\*/

System.out.println("poll(): "+q.poll());

/\*

\* peek() method - it works same as element() method,

\* however it returns null if the Queue is empty

\*/

System.out.println("peek(): "+q.peek());

//Again displaying the elements of Queue

System.out.println("Elements in Queue:"+q);

}

}

Queue serves the requests based on FIFO(First in First out). Now the question is: **What if we want to serve the request based on the priority rather than FIFO?** In a practical scenario this type of solution would be preferred as it is more dynamic and efficient in nature. This can be done with the help of PriorityQueue, which serves the request based on the priority that we set using [Comparator](https://beginnersbook.com/2017/08/comparator-interface-in-java/).

## Java PriorityQueue Example

In this example, I am adding few Strings to the PriorityQueue, while creating PriorityQueue I have passed the Comparator(named as MyComparator) to the PriorityQueue constructor.  
In the MyComparator java class, I have sorted the Strings based on their length, which means the priority that I have set in PriorityQueue is String length. That way I ensured that the smallest string would be served first rather than the string that I have added first.

**PriorityQueueExample.java**

import java.util.PriorityQueue;

public class PriorityQueueExample

{

public static void main(String[] args)

{

PriorityQueue<String> queue =

new PriorityQueue<String>(15, new MyComparator());

queue.add("Tyrion Lannister");

queue.add("Daenerys Targaryen");

queue.add("Arya Stark");

queue.add("Petyr 'Littlefinger' Baelish");

/\*

\* What I am doing here is removing the highest

\* priority element from Queue and displaying it.

\* The priority I have set is based on the string

\* length. The logic for it is written in Comparator

\*/

while (queue.size() != 0)

{

System.out.println(queue.poll());

}

}

}

**MyComparator.java**

import java.util.Comparator;

public class MyComparator implements Comparator<String>

{

@Override

public int compare(String x, String y)

{

return x.length() - y.length();

}

}

**Deque**

**Deque** is a Queue in which you can add and remove elements from both sides. we have seen that the Queue follows FIFO (First in First out) and in [PriorityQueue](https://beginnersbook.com/2017/08/java-collections-priorityqueue-interface/) example we have seen how to remove and add elements based on the priority. In this tutorial, we will see how to use Deque.

Deque is an interface and has two implementations: LinkedList and ArrayDeque. By implementation I refer that these classes LinkedList and ArrayDeque implements Deque interface, so we can create the instance of these and assign it to the Deque like this:

Deque dq = new LinkedList();

Deque dq = new ArrayDeque();

## Java Deque Interface Example

import java.util.Deque;

import java.util.ArrayDeque;

public class ArrayDequeExample {

public static void main(String[] args) {

/\*

\* We cannot create instance of a Deque as it is an

\* interface, we can create instance of ArrayDeque or

\* LinkedList and assign it to Deque

\*/

Deque<String> dq = new ArrayDeque<String>();

/\*

\* Adding elements to the Deque.

\* addFirst() adds element at the beginning

\* and addLast() method adds at the end.

\*/

dq.add("Glenn");

dq.add("Negan");

dq.addLast("Maggie");

dq.addFirst("Rick");

dq.add("Daryl");

System.out.println("Elements in Deque:"+dq);

/\*

\* We can remove element from Deque using remove() method,

\* we can use normal remove() method which removes first

\* element or we can use removeFirst() and removeLast()

\* methods to remove first and last element respectively.

\*/

System.out.println("Removed element: "+dq.removeLast());

/\*

\* element() method - returns the head of the

\* Deque. Head is the first element of Deque

\*/

System.out.println("Head: "+dq.element());

/\*

\* pollLast() method - this method removes and returns the

\* tail of the Deque(last element). Returns null if the Deque is empty.

\* We can also use poll() or pollFirst() to remove the first element of

\* Deque.

\*/

System.out.println("poll(): "+dq.pollLast());

/\*

\* peek() method - it works same as element() method,

\* however it returns null if the Deque is empty. We can also use

\* peekFirst() and peekLast() to retrieve first and last element

\*/

System.out.println("peek(): "+dq.peek());

//Again printing the elements of Deque

System.out.println("Elements in Deque:"+dq);

}

}