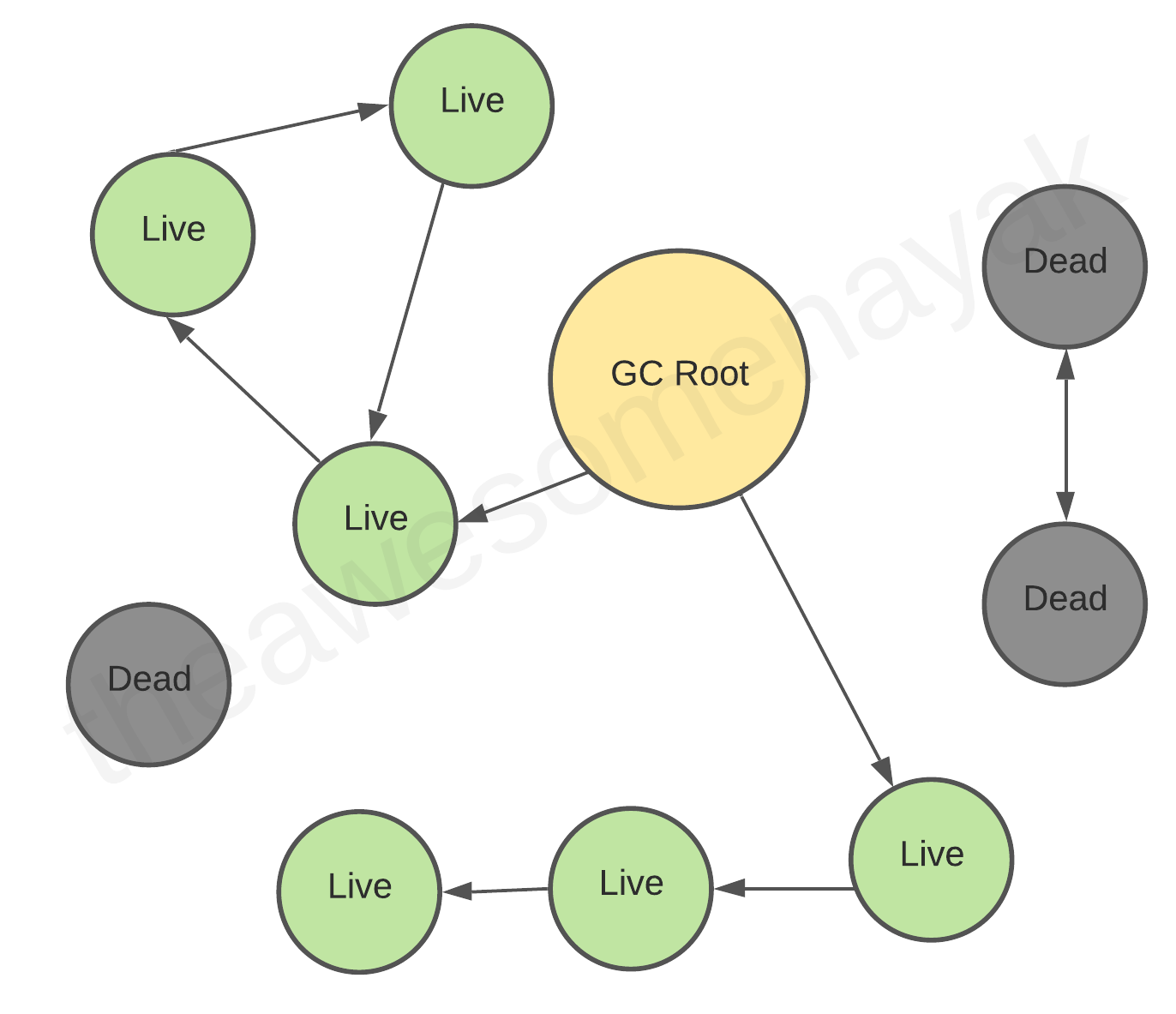
**Garbage Collector**You can say that at any point in time, the heap memory consists of two types of objects:

* *Live* - referenced from somewhere
* *Dead* - not reference anymore

*Garbage Collection Roots* (GC Roots) 

***How to define garbage?***

**Reference Counting Algorithm**

The [Reference Counting Algorithm](https://blogs.msdn.microsoft.com/abhinaba/2009/01/27/back-to-basics-reference-counting-garbage-collection/) allocates a field in the object header to store the reference count of the object. When the reference count of this object drops to zero, the object will be garbage-collected.

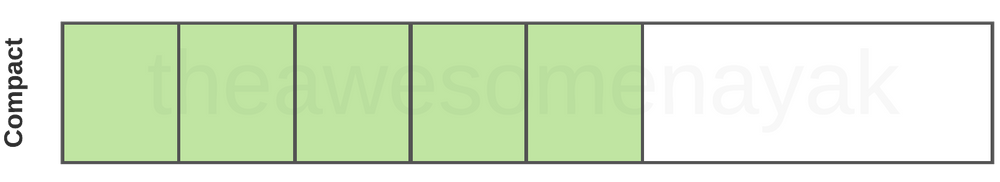
## **Reachability Analysis Algorithm**

start from GC roots.GC traverses the whole object graph in the memory, starting from these roots an object has no reference chain to the GC roots collect.

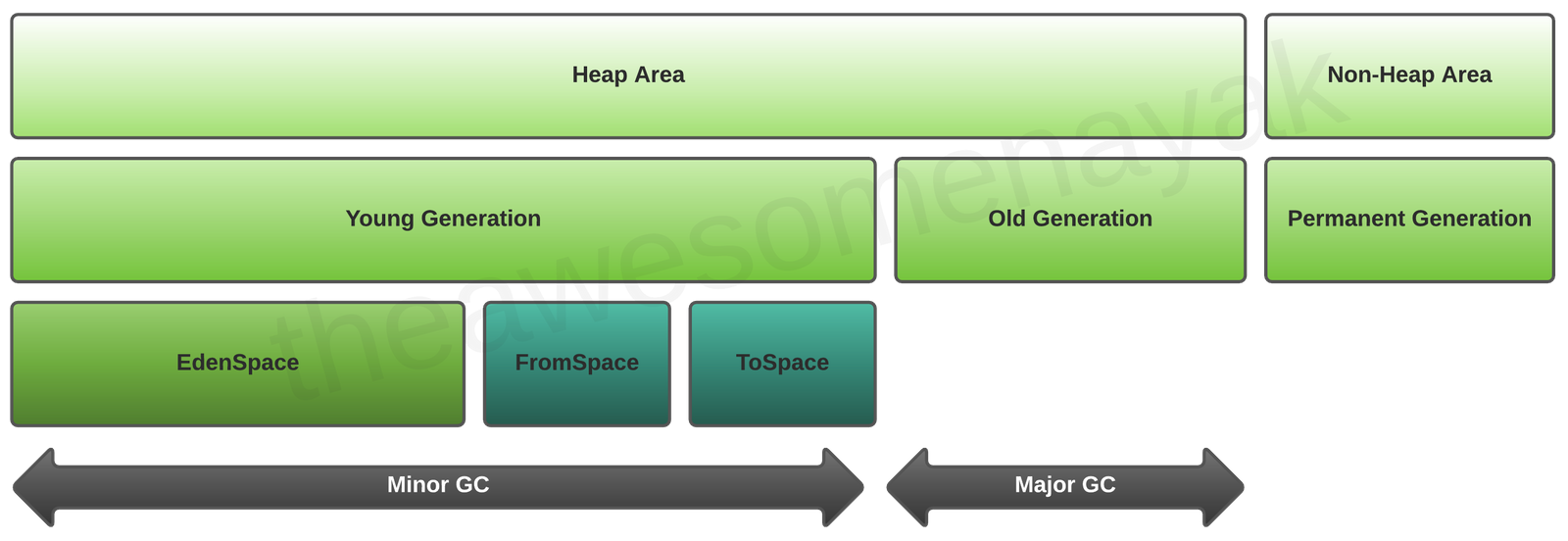
Phases of GC:

1. Mark as alive: By traversing the graph. When an object is visited it is marked as alive.
2. Sweep Dead Objects: Releases the memory fragments that contain dead objects.
3. Compact Remaining Objects In Memory: When sweeped, objects may not be next to each other which results in fragmented memory. MEmory should be compact for the incoming objects.





**Generational Garbage Collection in Java**



Young Generation: For newly created objects. 2 sub:

1. Eden: All new objects start here. Initial place.
2. Survivor: Objects come here from eden after surviving one garbage collection.

When objects are garbage collected from the Young Generation, it is a *minor garbage collection event*.

When Eden space is filled with objects, a Minor GC is performed.

So, at any time, one of the survivor spaces is always empty.

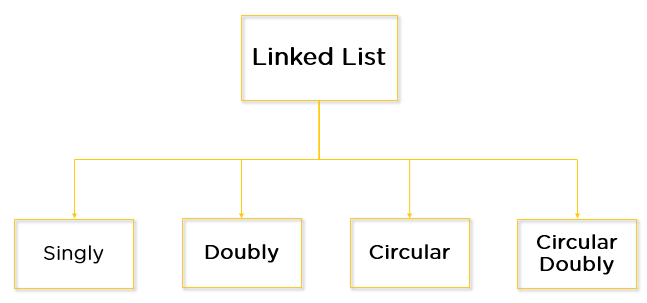
Old Generation: Objects that are long-lived. contains objects that have remained in the survivor spaces for a long time. (There is a threshold defined for how many cycles considered as long)

When objects are garbage collected from the Old Generation, it is a *major garbage collection event*.

Types of garbage collections in JVM:

1. Serial GC: All garbage collection events are conducted serially in one thread. Compaction is executed after each garbage collection."stop the world" event where the entire application is paused.
2. Parallel GC: run on multiprocessor or multithreaded hardware. default implementation. Multiple threads are used for minor garbage collection in the Young Generation. A single thread is used for major garbage collection in the Old Generation.
3. Parallel Old GC: Paralel for both young and old generations.
4. Concurrent Mark Sweep (CMS): Multiple threads are used for minor garbage collection using the same algorithm as Parallel. processes to minimize “stop the world” events. (Doesnt stop the whole app, runs on the side paralel)
5. G1: Although G1 is also generational, it does not have separate regions for young and old generations. Instead, each generation is a set of regions, which allows resizing of the young generation in a flexible way.
6. Epsilon: It handles memory allocation but does not implement any actual memory reclamation mechanism. Once the available Java heap is exhausted, the JVM shuts down.where developers know the application memory footprint exactly,
7. Shenandoah: G1 can evacuate its heap regions only when the application is paused, while Shenandoah can relocate objects concurrently with the application.Shenandoah can compact live objects, clean garbage, and release RAM back to the OS almost immediately after detecting free memory
8. ZGC.

**Linked List Types**

****

**Singley:**

Traversel in one way only. Node points to next node.

Node Structor:

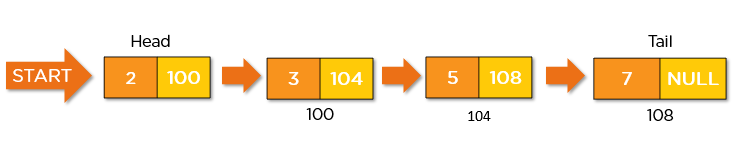
**static class Node {**

**int data;**

**// Pointer to next node in LL**

**Node next;**

**};**

****

**Examples**: Implementation of [stacks](https://www.geeksforgeeks.org/stack-data-structure/) and [queues](https://www.geeksforgeeks.org/queue-data-structure/), Maintaining a directory of names, Implementation of graphs: [Adjacency list representation of graphs](https://www.geeksforgeeks.org/graph-and-its-representations/) is the most popular which uses a linked list to store adjacent vertices.

**Doubly:**

Traversal in both direction. Node can point to next and prev.

**Node Stucture:**

**static class Node {**

**int data;**

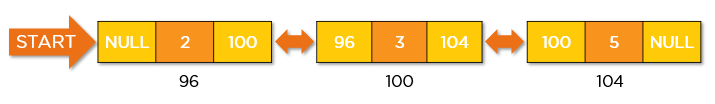
**// Pointer to next node in DLL**

**Node next;**

**// Pointer to the previous node in DLL**

**Node prev;**

**};**

****

**Examples: mage viewer – Previous and next images are linked and can be accessed by the next and previous buttons.**

***Design a data structure that supports following operations efficiently.***

1. ***getMin***
2. ***extractMin***
3. ***getMax***
4. ***extractMax***
5. ***Insert***
6. ***Redo and undo functionality.***
7. ***Use of the Back and forward button in a browser.***
8. ***The most recently used section is represented by the Doubly Linked list.***
9. ***Other Data structures like Stack, HashTable, and BinaryTree can also be applied by Doubly Linked List.***

**Circular:**

**Last node contains a pointer to the first node. No beginning or end. Can start from any node to traverse. Can traverse the list in any direction.**

**data can be added and removed from the list at any time. deal for applications where data needs to be constantly added or removed,**

**Node Structure:**

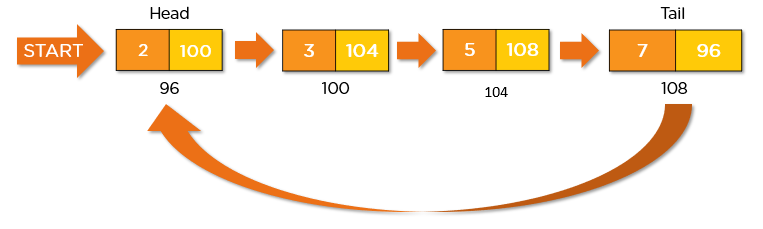
**static class Node {**

**int data;**

**// Pointer to next node in CLL**

**Node next;**

**};**

****

**Examples:**

**Useful for implementation of a queue.**

**Circular lists are useful in applications to go around the list repeatedly.**

**Circular Doubly:**

Same logic as circular but node contains next and prev pointers. More complex version. The difference between the doubly linked and circular doubly list is the same as that between a singly linked list and a circular linked list. The circular doubly linked list does not contain null in the previous field of the first node.

Node Sturcture:

static class Node

{

int data;

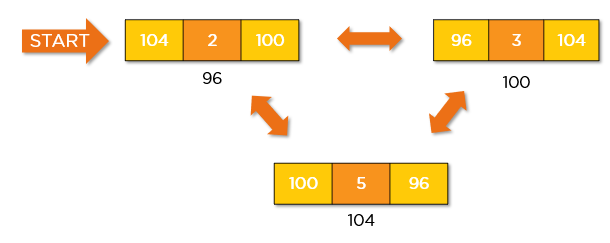
// Pointer to next node in DCLL

Node next;

// Pointer to the previous node in DCLL

Node prev;

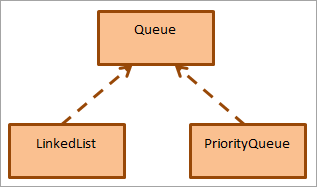
};

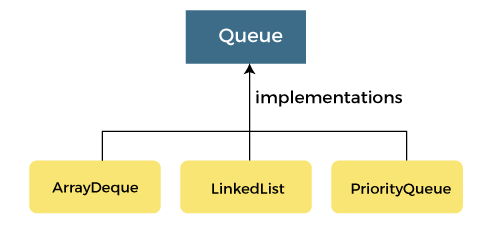
****

**Examples: used for the implementation of advanced data structures like the** [**Fibonacci Heap**](http://en.wikipedia.org/wiki/Fibonacci_heap)**.**

**Queue Types**

* **Simple Queue:** nsertion takes place at the rear and removal occurs at the front. It strictly follows the FIFO (First in First out) rule.
* **Circular Queue:** he last element points to the first element making a circular link.
* **Priority Queue**: A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.
* **Double Ended Queue:** In a double ended queue, insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow the FIFO (First In First Out) rule.

****

****

* [LinkedBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingQueue.html) — an optionally bounded FIFO blocking queue backed by linked nodes
* [ArrayBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html) — a bounded FIFO blocking queue backed by an array
* [PriorityBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/PriorityBlockingQueue.html) — an unbounded blocking priority queue backed by a heap