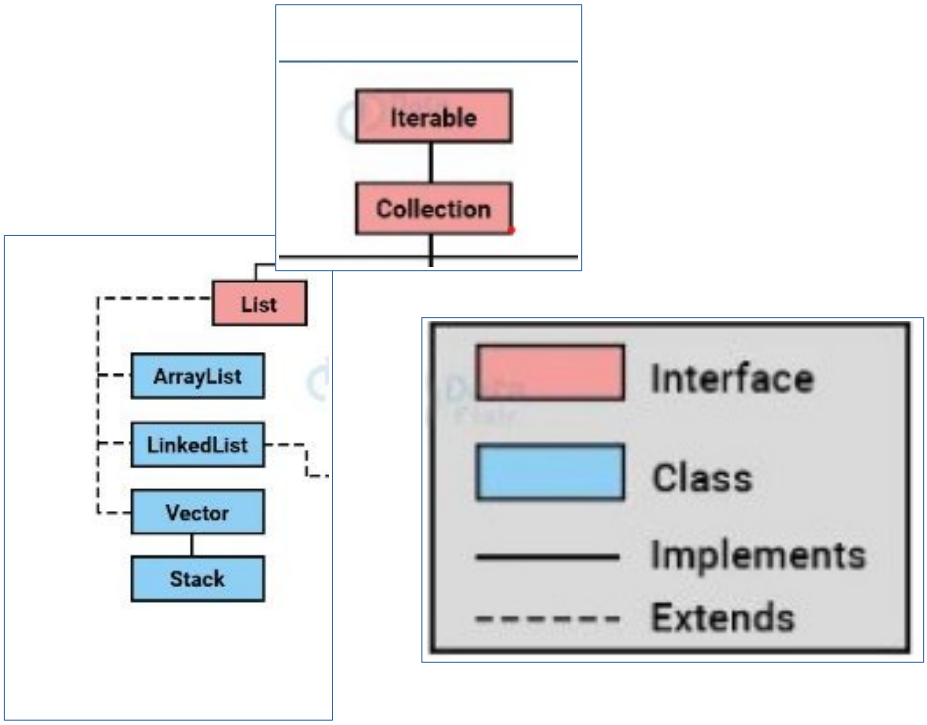


Listen (Lists) sind eine der grundlegendsten und vielseitigsten Datenstrukturen im **Java Collections Framework**. Im Gegensatz zu **Sets**, die sich auf einzigartige Elemente konzentrieren, und **Maps**, die Schlüssel-Wert-Paare speichern, erlaubt eine **List** das Speichern von **mehrfach vorkommenden Elementen** und bietet **Index-basierten Zugriff**

Grundkonzepte von Listen (List)

1. Geordnete Sammlung:

- Eine **List** speichert ihre Elemente in einer **bestimmten Reihenfolge**. Diese Reihenfolge bleibt fest, es sei denn, sie wird explizit geändert (durch Hinzufügen, Entfernen oder Sortieren der Elemente).
- Die Position der Elemente wird durch **Indizes** repräsentiert, beginnend bei 0.

2. Zugriff über Index:

- Im Gegensatz zu **Sets**, die keinen Zugriff über einen Index bieten, erlaubt eine **List** den direkten **Zugriff auf Elemente über einen Index**. Du kannst also auf das erste, zweite oder beliebige n-te Element zugreifen, ohne die Liste durchlaufen zu müssen.

3. Duplikate erlaubt:

- **Listen erlauben Duplikate**, was bedeutet, dass das gleiche Element **mehrfach** in der Liste vorhanden sein kann. Die Reihenfolge, in der die Elemente hinzugefügt wurden, bleibt erhalten, und du kannst sie über ihren Index ansprechen.

4. Null-Werte:

- Listen erlauben in der Regel die Speicherung von **null-Werten**. Je nach Implementierung kann eine Liste einen oder mehrere null-Werte enthalten.

Wichtige List-Typen

1. ArrayList:

- **ArrayList** ist die am häufigsten verwendete List-Implementierung. Sie verwendet intern ein **dynamisches Array**, das sich bei Bedarf automatisch vergrößert.
- Vorteile:
  1. Sehr schneller **Zugriff auf Elemente** über den Index (O(1)).
  2. Ideal für Szenarien, bei denen häufig auf Elemente zugegriffen wird.
- Nachteile:
  1. Das **Einfügen und Entfernen** von Elementen in der Mitte der Liste ist langsamer, da alle nachfolgenden Elemente verschoben werden müssen (O(n)).
  2. Nicht synchronisiert (nicht thread-sicher).

2.LinkedList:

- **LinkedList** ist eine Implementierung der `List`-Schnittstelle, die auf einer **doppelt verketteten Liste** basiert. Jedes Element in der Liste ist mit dem vorherigen und dem nächsten Element verknüpft.
- **Vorteile:**

  Einfügen und Entfernen von Elementen ist sehr schnell, insbesondere am Anfang oder Ende der Liste (O(1)), da keine Verschiebungen wie bei der `ArrayList` erforderlich sind.
- **Nachteile:**

  Zugriff auf Elemente über den Index ist langsamer (O(n)), da die Liste von Anfang an durchlaufen werden muss.

3. Vector:

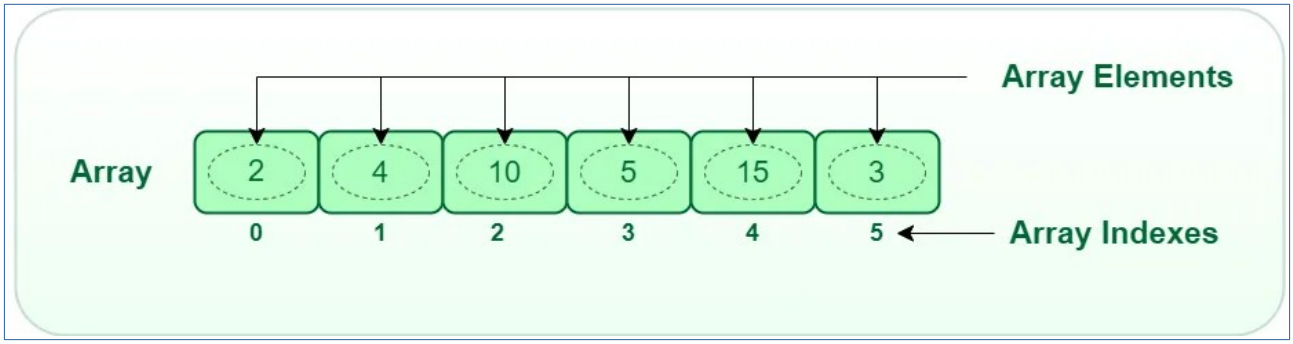
- **Vector** ist eine ältere Implementierung, die ebenfalls ein **dynamisches Array** verwendet, aber im Gegensatz zur `ArrayList` ist es **synchronisiert**. Das bedeutet, dass `Vector` **thread-sicher** ist, aber langsamer im Vergleich zur `ArrayList`.

4. Stack:

- **Stack** ist eine spezielle Art von `Vector`, die nach dem **LIFO-Prinzip** (Last In, First Out) arbeitet. Elemente werden oben auf den Stapel gelegt und auch wieder von dort entfernt.

1.1 Array:

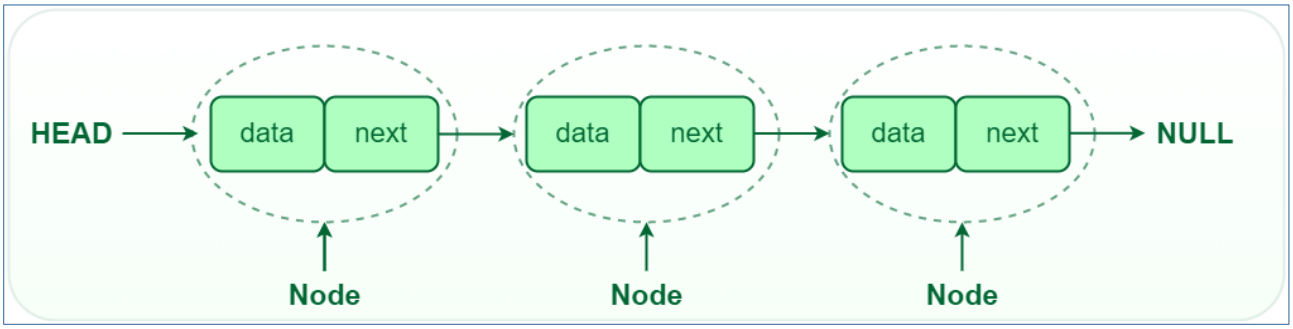
- **Indexed**: Index 0 => bin n -1
- **fetse Größe**, die nicht verändert kann
- kann nur **Elemente desselben Datentyps** speichern
- Zugriff erfolgt **direkt** über seinen Index
- sind **änderbar** (mutable)
- alle Elemente eines Arrays **nebeneinander im Speicher** liegen.
- alle seine Elemente automatisch auf **Standardwerte** Typs gesetzt
- **Iteration** durch Schleifen (wie `for`, `for-each`, `while`).
- Speicherkomplexität eines Arrays ist **O(n)**
- In Java sind auch **mehrdimensionale Arrays möglich**
- Arrays ermöglichen **zufälligen Zugriff (random access)** auf ihre Elemente



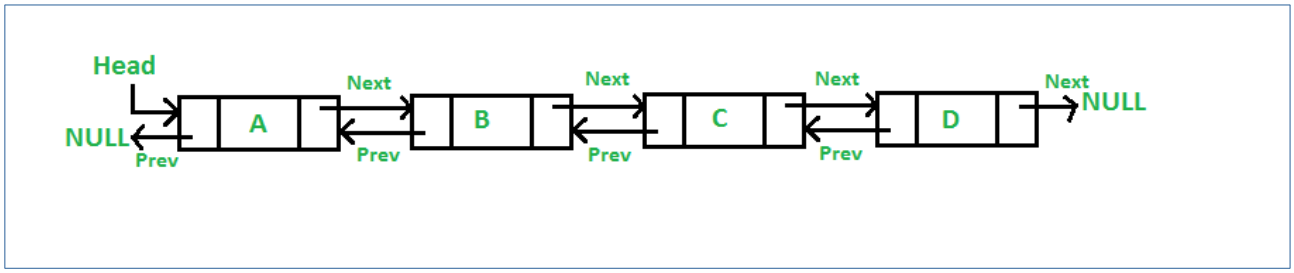
1.2 Linked List:

- **Nicht Indexed**: kann nicht direkt auf das n-te Elemente zugreifen
- **Dynamische Größe:** man soll keine Größe eingeben.
- Speicherkomplexität einer LinkedList **O(n)**
- **Elemente bestehen aus Knoten:**
  1. **Daten(Knote):** das eigentliche Element, das man speichern möchte
  2. **Verweis (next Knote):** auf den nächsten Knoten in der Liste
- alle Elemente eines LinkedList **nicht nebeneinander im Speicher** liegen.
- unterstützt keinen **zufälligen Zugriff** auf ihre Elemente,
- **Verschiedene Varianten der Linked List:**
  1. **Singly Linked List (einfach verkettete Liste):** Jeder Knoten enthält einen Verweis auf den nächsten Knoten.
  2. **Doubly Linked List (doppelt verkettete Liste):** Jeder Knoten enthält Verweise auf den nächsten und vorherigen Knoten
  3. **Circular Linked List (zirkulär verkettete Liste):** In dieser Variante zeigt der letzte Knoten wieder auf den ersten,

Singly Linked List:

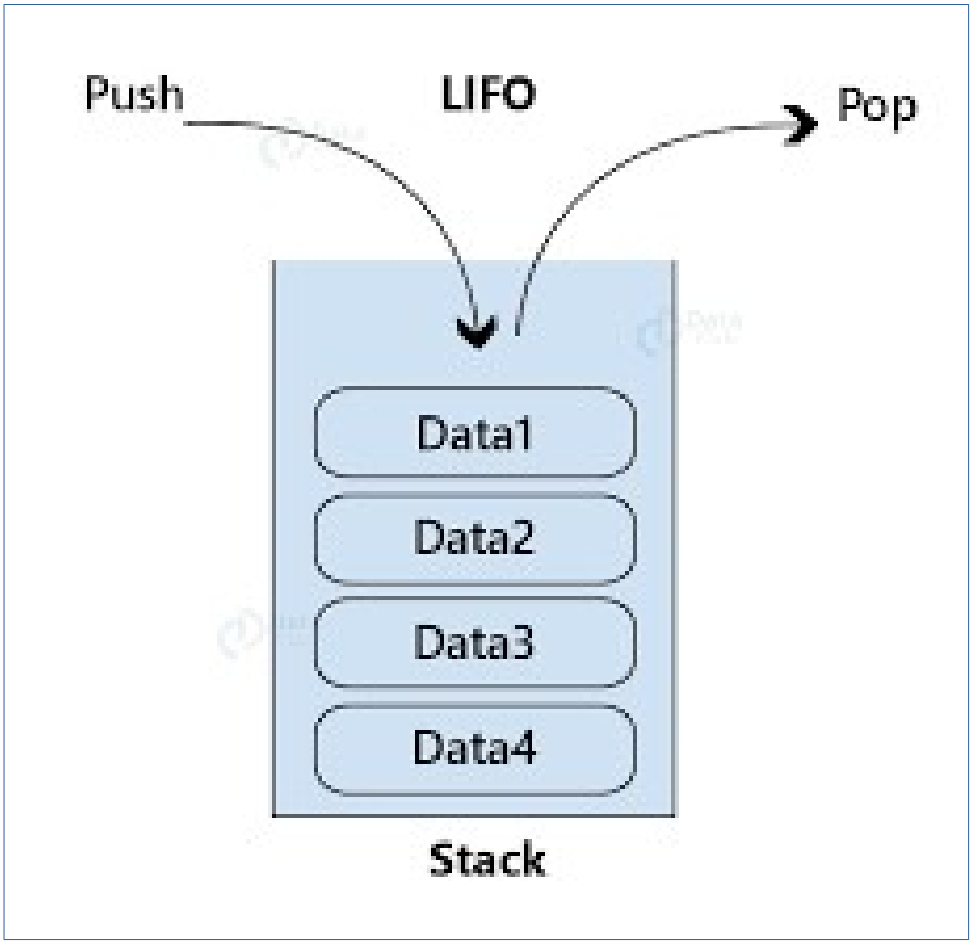


Doubly Linked List:



1.3 Stack:

- **LIFO-Prinzip (Last In, First Out):** das **letzte Element**, das in den Stack **eingefügt wurde**, als **erstes wieder entfernt wird.**
- erlaubt **nur den Zugriff auf das oberste Element** (genannt **peek** -Operation)
- **Hinzufügen** eines Elements (genannt **push**-Operation) Zeitkomplexität **O(1)**
- **Entfernen** eines Elements (genannt **pop**-Operation) Zeitkomplexität **O(1)**
- **Suchen** eines Elements, gibt entweder das Index des Elements oder ‚-1'
- kann als **Array-basiert** oder **Linked-List-basiert** implementiert werden und
- **wächst dynamisch**
- **keinen direkten Zugriff auf die unteren Elemente**
- **Speicherkomplexität O(n)**
- erlaubt **keinen zufälligen Zugriff** auf seine Elemente



Wichtige Unterschiede zwischen List-Typen

List-Typ	Synchronisiert	Indexzugriff	Zugriffszeit (über Index)	Einfügen/Entfernen in der Mitte	Verwendung
ArrayList	Nein	Ja	O(1)	O(n)	Häufiger Zugriff auf Elemente
LinkedList	Nein	Ja	O(n)	O(1)	Häufiges Einfügen/Löschen von Elementen
Vector	Ja	Ja	O(1)	O(n)	Synchronisierte Variante von `ArrayList`
Stack	Ja	Nein	-	-	LIFO-Verhalten (Last In, First Out)

1.5 Trees:

1. **Binärbaum:**

- Ein **Binärbaum** ist ein spezieller Baum, bei dem jeder Knoten maximal zwei Kinder hat: einen **linken** und einen **rechten** Kindknoten.
- Ein Binärbaum kann voll, komplett, perfekt oder unbalanciert sein:
    - **Voller Binärbaum**: Jeder Knoten hat entweder 0 oder 2 Kinder.
    - **Perfekter Binärbaum**: Alle Ebenen sind vollständig gefüllt.
    - **Kompletter Binärbaum**: Alle Ebenen außer der letzten sind vollständig gefüllt, und alle Knoten in der letzten Ebene sind so weit wie möglich links ausgerichtet.

2. Suchbaum (Binary Search Tree, BST):

- Ein **Binärer Suchbaum** (BST) ist ein Binärbaum, bei dem die Daten in einer bestimmten Reihenfolge gespeichert werden:
    - Für jeden Knoten gilt, dass alle Werte im **linken Unterbaum** kleiner sind als der Wert des Knotens, und alle Werte im **rechten Unterbaum** größer.
- Diese Struktur ermöglicht effiziente Suchoperationen mit einer durchschnittlichen Zeitkomplexität von **O(log n)**.

3. AVL-Baum:

- Ein **AVL-Baum** ist ein selbstbalancierender Binärbaum, bei dem die Höhe der beiden Teilbäume jedes Knotens sich höchstens um 1 unterscheidet.
- Dadurch wird garantiert, dass die Höhe des Baums logarithmisch bleibt, was die Leistung der Such-, Einfüge- und Löschoperationen bei **O(log n)** hält.

4. Rot-Schwarz-Baum

- Ein **Rot-Schwarz-Baum** ist ein weiterer selbstbalancierender Binärbaum, bei dem jeder Knoten entweder **rot** oder **schwarz** gefärbt ist, um eine bestimmte Balance zu wahren.
- Er garantiert ebenfalls eine logarithmische Höhe und bietet Such-, Einfüge- und Löschoperationen in **O(log n)**.

5. Heap (Halbgeordneter Baum):

- Ein **Heap** ist ein spezieller Binärbaum, der entweder ein **Max-Heap** oder ein **Min-Heap** sein kann.
    - Im **Max-Heap** ist der Wert eines Elternknotens immer größer oder gleich den Werten seiner Kinder.
    - Im **Min-Heap** ist der Wert eines Elternknotens immer kleiner oder gleich den Werten seiner Kinder.
- Heaps werden typischerweise zur Implementierung von **Prioritätswarteschlangen** verwendet.

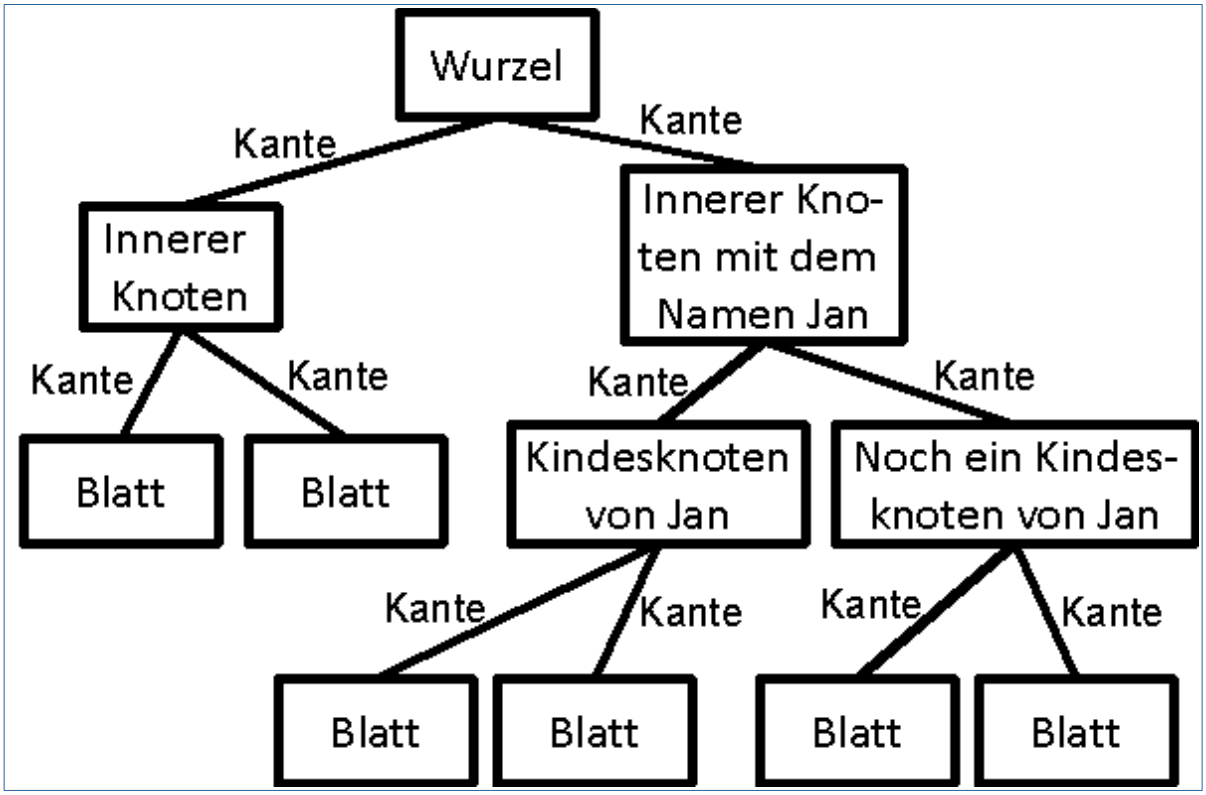
6. B-Baum:

- Ein **B-Baum** ist ein selbstbalancierender Baum, der in Datenbanken und Dateisystemen verwendet wird. Er kann mehr als zwei Kinder pro Knoten haben und sorgt für eine effiziente Datenspeicherung und -suche auf Festplatten.

- Jeder Knoten kann mehrere Schlüssel enthalten, und der Baum wächst und schrumpft, um die gespeicherten Daten effizient zu organisieren.

7. Trie (Prefix Tree):

- Ein **Trie** ist ein spezieller Baum, der zur Speicherung von Zeichenfolgen verwendet wird. Jeder Pfad von der Wurzel zu einem Blatt repräsentiert eine Zeichenfolge, wobei jeder Knoten ein einzelnes Zeichen speichert.
- Tries sind nützlich für effiziente Suchvorgänge, z.B. in Autovervollständigungssystemen oder Wörterbüchern.



1.6 Graphen: