Java Data Structures
    # ArrayList
    # LinkedList
    # Stack
    # Queue
    # Deque
    # Set
    # Map
    # HashTable
    # String, StringBuilder, StringBuffer
    # Arrays
    # Stream
    # Big Decimal
    # Timer
    # Tree
        Red-Black tree

H3 Java Data Structures

H4 # ArrayList

- Slower, but offers flexibility in manipulations.
- It is initialized by size, and can grow dynamically
- Supports random access
- Doesn't support primitives

```
//instance default
ArrayList<E> al = new ArrayList<E>();
ArrayList<E> al = new ArrayList<E>(size);

//instance from stream using list
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
List<E> list = stream.collect(Collectors.toList());
ArrayList<E> al = new ArrayList<E>(list);

//instance from stream using toCollection method
ArrayList<T> al = stream.collect(Collectors.toCollection(ArrayList::new));

//methods
al.clear();
al.isEmpty();

al.add(E element);
```

```
al.set(index, E element);

al.get(index);
al.contains(E element);
al.indexOf(E element);

al.lastIndexOf(E element);
al.ensureCapacity(newSize);
al.trimToSize();

al.remove(index);
al.remove(E element);

//convert
Object[] = al.toArray();

//iterations
al.forEach(n -> Sout...)
al.removeIf(n -> (n % 3 == 0));

ListIterator<E> it = al.listIterator();
ListIterator<E> it = al.listIterator(fromIndex);
while (it.hasNext()) {
  System.out.println("Value is : "+ it.next());
}
```

H4 # LinkedList

```
LinkedList<E> ll = new LinkedList<E>();

// add vs offer
// On failure, add throws exception, offer returns false.

ll.add(index, element); // tail
ll.addFirst(element);
ll.addLast(element); // last element

ll.offer(element);
ll.offerFirst(element);
ll.offerLast(element);

ll.removeFirst(element);
ll.removeLast(element);
getFirst();
getLast();

```

```
push(element);
peek(); // head, don't remove
poll(); // head, remove

peekFirst();
peekLast();

pollFirst();
pollLast();

```

Stack

- LIFO

```
Stack<E> stack = new Stack<E>();
push(element);
peek();
pop();
search(element);
removeElementAt(index);
```

Queue

- FIFO - Insert at tail, remove at head
- Implemented by PriorityQueue and LinkedList

Deque

- FIFO and LIFO
- Offers insert, remove at both ends
- Faster than stacks and linked list

Set

- Unordered list, dups are not allowed
- HashSet, LinkedHashSet, and TreeSet (sorted)
- HashSet
  - Doesn't preserve insertion order
  - Search is O(1)
- TreeSet
  - Doesn't preserve the insertion order, instead elements are sorted
  - Search is O(Log(n))

Map

- Key/Value

- No Dups
- TreeMap and LinkedListMap maintain order, HashMap doesn't

HashTable

- Same as Map, but synchronized

String, StringBuilder, StringBuffer

- String is immutable
- StringBuilder is mutable
- StringBuffer is thread safe implementation of StringBuilder

```
StringBuffer sbr = new StringBuffer(str);
sbr.append(str);

StringBuilder sbl = new StringBuilder(str);
sbl.append(str);
sbl.reverse(str);

sbr.toString();
sbl.toString();

Integer.toString(int);
String.valueOf(int);
Integer(n).toString();

int e = 12345;
DecimalFormat df = new DecimalFormat("#,###");
String Str5 = df.format(e);
System.out.println(Str5); // 12,345

int h = 255;
Integer.toBinaryString(h); // 11111111
Integer.toHexString(j); // FF
Integer.toOctalString(i); // 377

Integer.toString(int, 7); // to base 7

```

Arrays

```
import java.utils.Array;

int arr[] = { 19, 20};
Arrays.toString(arr);
Arrays.asList(arr);
Arrays.toStream(arr);
```

```
Arrays.sort(arr);
Arrays.binarySearch(arr, key);
Arrays.binarySearch(arr, fromIndex, toIndex, key);
Arrays.compare(arr1, arr2);

// Set of Integers to Set of Strings
Set<Integer> setOfInteger =
  new HashSet<>(Arrays.asList(1, 2, 3, 4, 5));
Set<String> setOfString = setOfInteger
  .stream()
  .map(String::valueOf)
  .collect(Collectors.toSet());


```

H4 # Stream

```
// from collection
List<String> list = new ArrayList<>();
Stream<T> stream = list.stream();
Iterator<T> it = stream.iterator();
while (it.hasNext()) {
  System.out.print(it.next() + " ");
}

// from values
Stream<Integer> stream
  = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
stream.forEach(p -> System.out.print(p + " "));

// from array
Stream<T> streamOfArray = Arrays.stream(arr);
Stream<String> streamOfArray = Stream.empty();

// fluid
Stream.Builder<String> builder = Stream.builder();
Stream<String> stream = builder.add("a")
  .add("b")
  .add("c")
  .build();

// infinite
Stream.iterate(seedValue = 2, (Integer n) -> n * n)
  .limit(limitTerms = 5)
  .forEach(System.out::println);
// 2, 4, 16, 256, 65536
```

```
30
31  // generate
32  Stream.generate(Math::random)
33    .limit(limitTerms = 5)
34    .forEach(System.out::println);
35
36  // filter
37  list.stream()
38    .filter(p.asPredicate()) // p is a pattern
39    .forEach(System.out::println);
```

Big Decimal

- Java Double and Float are floating point numbers stored as binary representation of fraction and exponent. Results have small errors 0.9999999 etc.
- Big Decimal provide accurate representation .` `

Timer

```
1   long start = System.currentTimeMillis();
2   // ...
3   long finish = System.currentTimeMillis();
4   long timeElapsed = finish - start;
5
6   long start = System.nanoTime();
7   // ...
8   long finish = Sy    stem.nanoTime();
9   long timeElapsed = finish - start;
10
11  Instant start = Instant.now();
12  // ...
13  Instant finish = Instant.now();
14  long timeElapsed =
15    Duration.between(start, finish).toMillis();
```

Tree

- Heavy mutations - use Red-Black tree
- Low mutations - use AVL tree

Red-Black tree

- Every node is either black or red
- All roots aree black
- A red node can never have red parent
- All routes from root to leaf will have the same number of black nodes - also called the black height.
- Tree height = 2Log(n+1). Black height is (tree height) / 2