Data Structures

Lists, Maps, and Sets

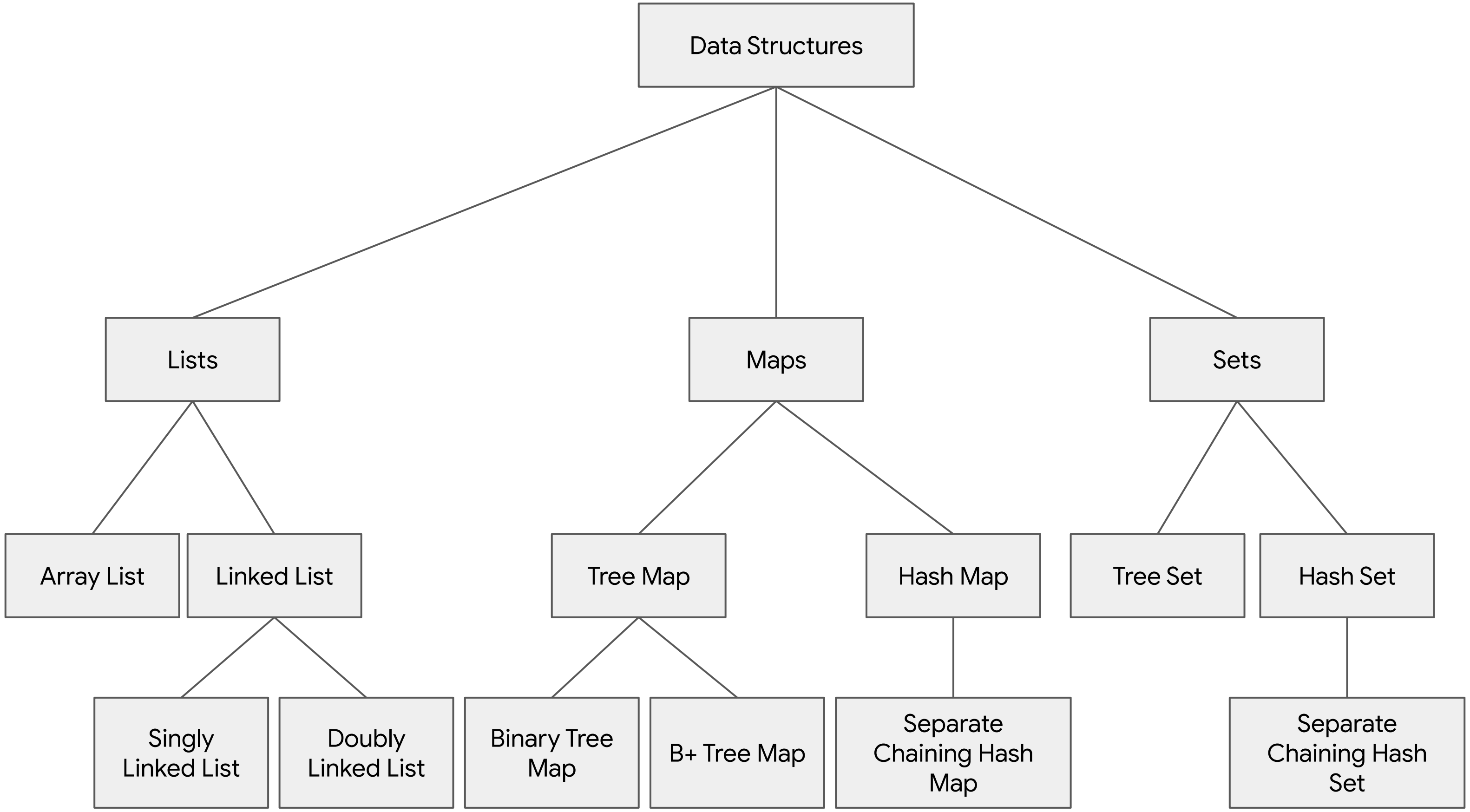
Objectives

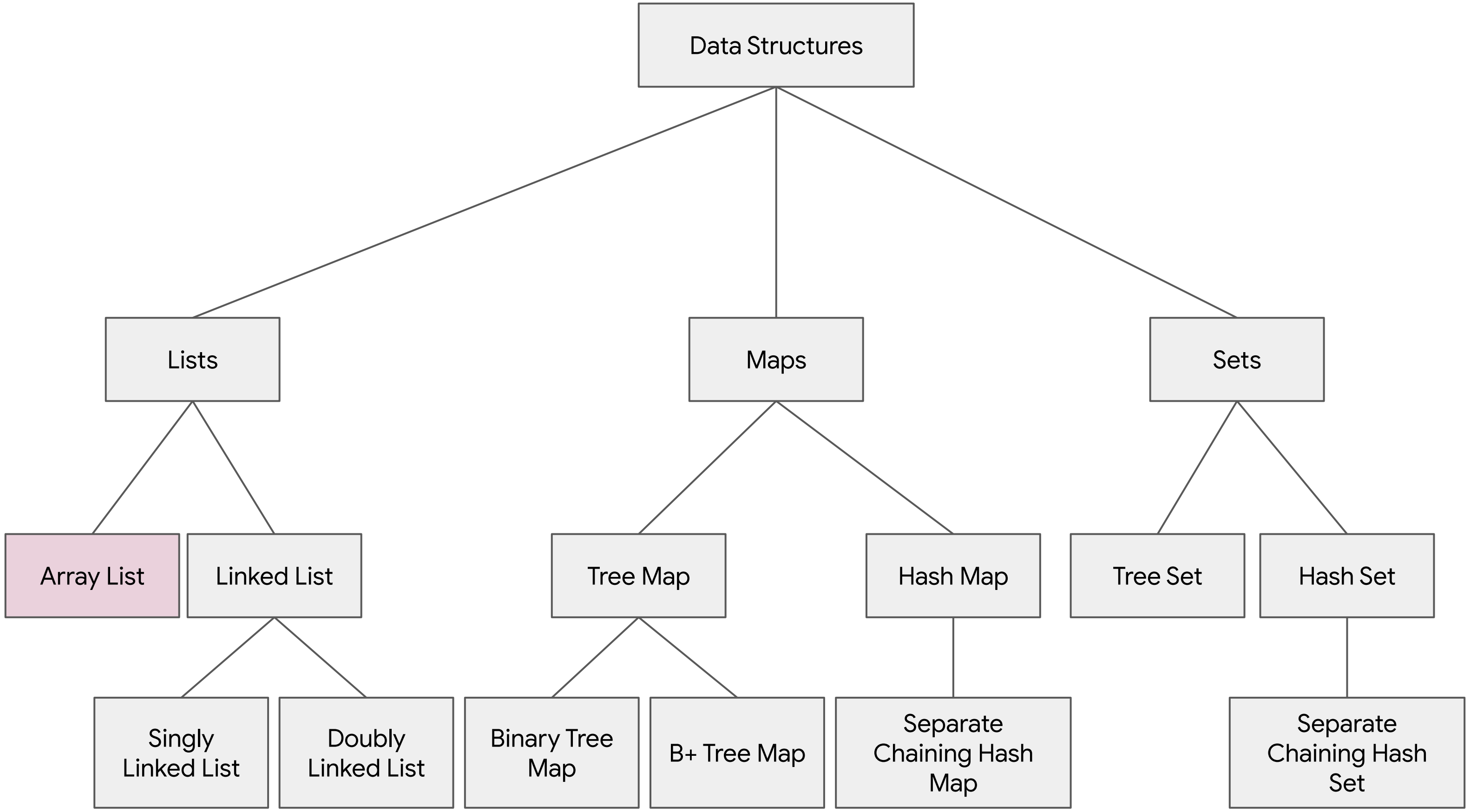
Primary Objectives

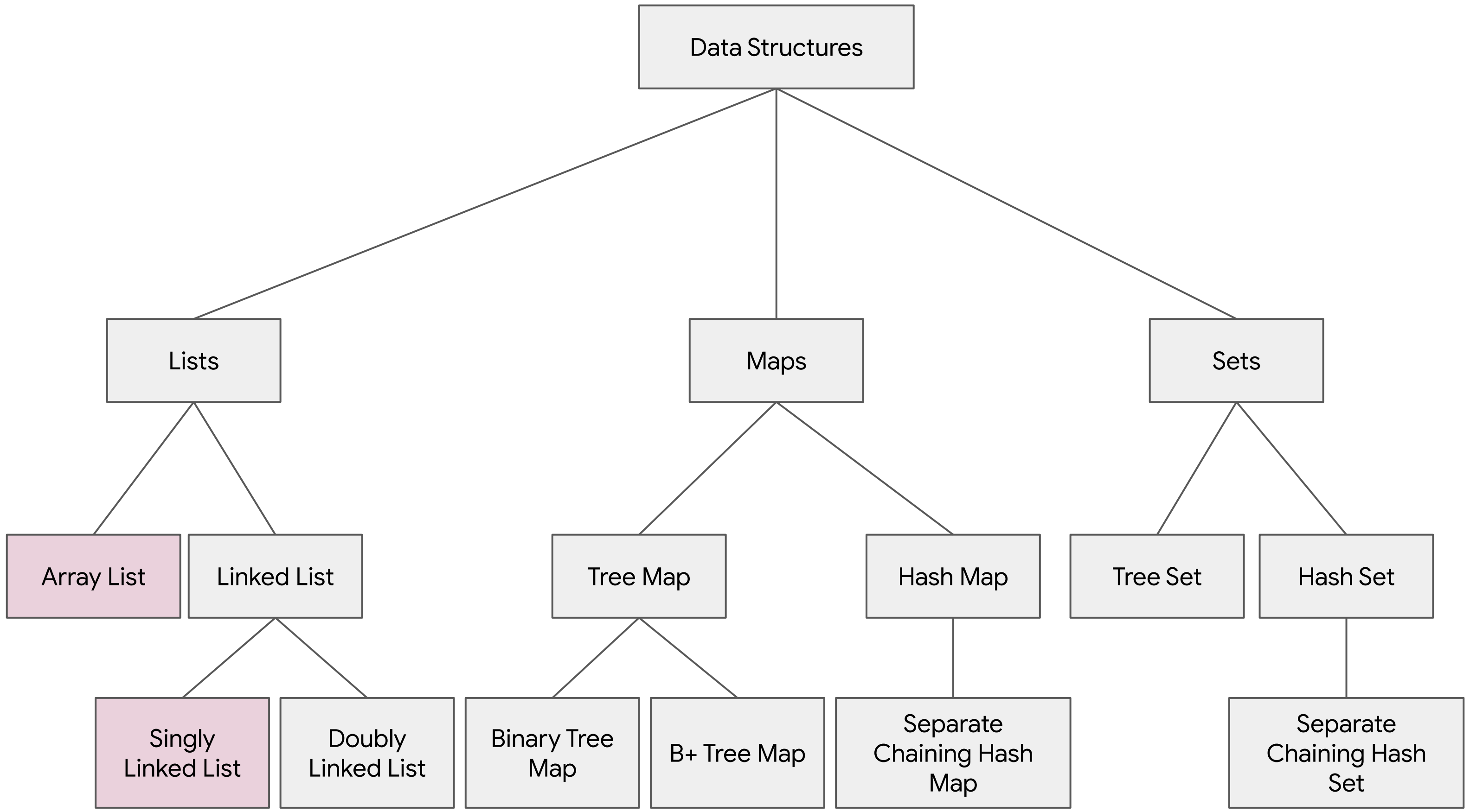
- What is a **list** and when could I use it?
- What is a **set** and when could I use it?
- What is a **map** and when could I use it?

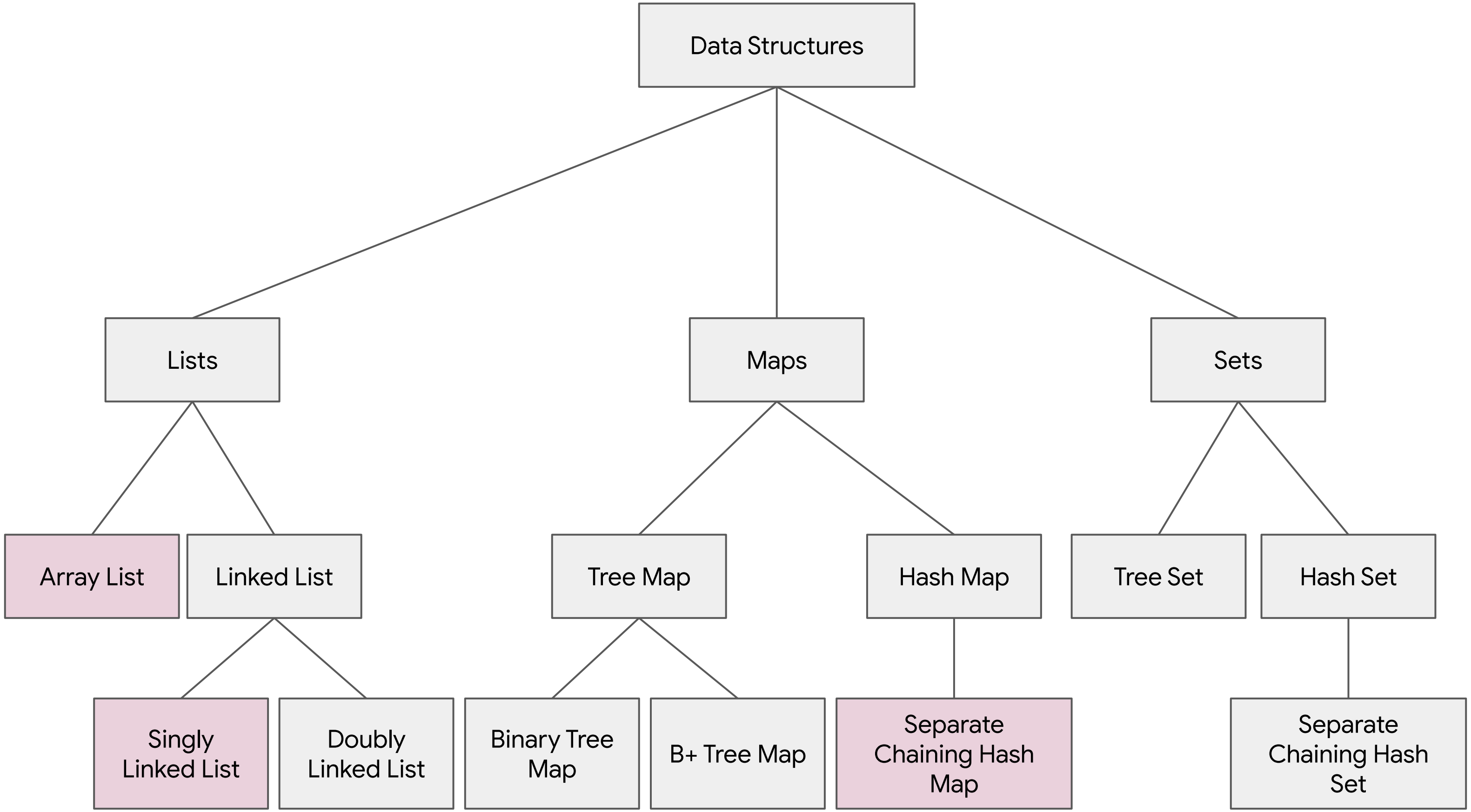
Secondary Objectives

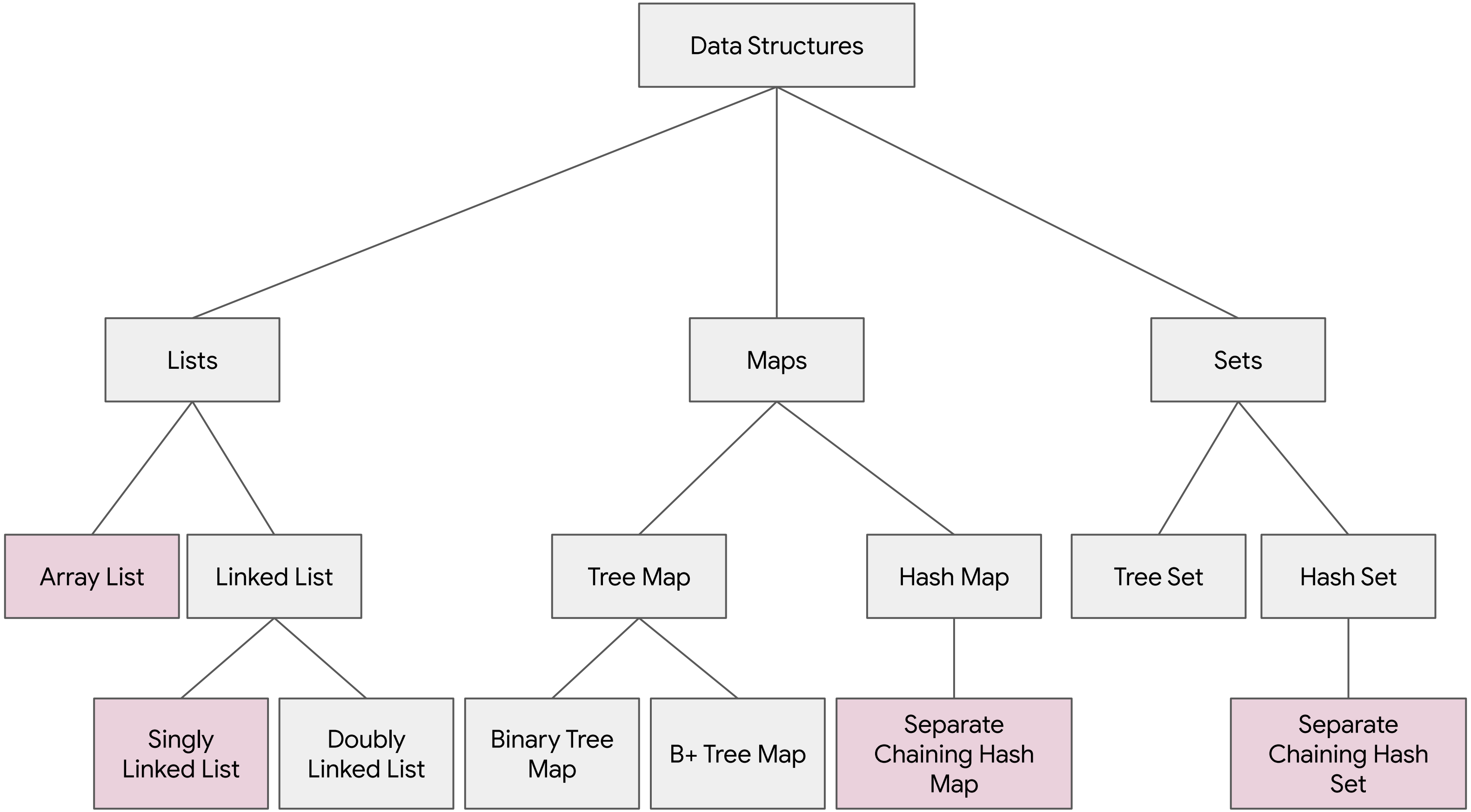
- What trade-offs am I making when I use a **list**?
- What trade-offs am I making when I use a **set**?
- What trade-offs am I making when I use a **map**?


Data Structures
Lists
Maps
Sets
Array List
Linked List
Tree Map
Hash Map
Tree Set
Hash Set
Singly Linked List
Doubly Linked List
Binary Tree Map
B+ Tree Map
Separate Chaining Hash Map
Separate Chaining Hash Set


Data Structures
Lists
Maps
Sets
Array List
Linked List
Tree Map
Hash Map
Tree Set
Hash Set
Singly Linked List
Doubly Linked List
Binary Tree Map
B+ Tree Map
Separate Chaining Hash Map
Separate Chaining Hash Set


Data Structures
Lists
Maps
Sets
Array List
Linked List
Tree Map
Hash Map
Tree Set
Hash Set
Singly Linked List
Doubly Linked List
Binary Tree Map
B+ Tree Map
Separate Chaining Hash Map
Separate Chaining Hash Set


Data Structures
Lists
Maps
Sets
Array List
Linked List
Tree Map
Hash Map
Tree Set
Hash Set
Singly Linked List
Doubly Linked List
Binary Tree Map
B+ Tree Map
Separate Chaining Hash Map
Separate Chaining Hash Set


Data Structures
Lists
Maps
Sets
Array List
Linked List
Tree Map
Hash Map
Tree Set
Hash Set
Singly Linked List
Doubly Linked List
Binary Tree Map
B+ Tree Map
Separate Chaining Hash Map
Separate Chaining Hash Set

Lists

Characteristics

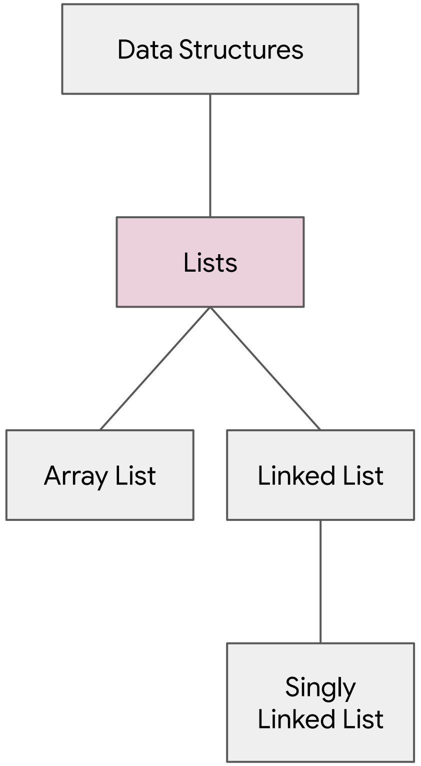
- Ordered indexable sequences
- Can contain duplicate values

General Strengths

- Good insertion
- Good iteration

General Weaknesses

- Poor look-up
- Poor removal



Array Lists

Structure

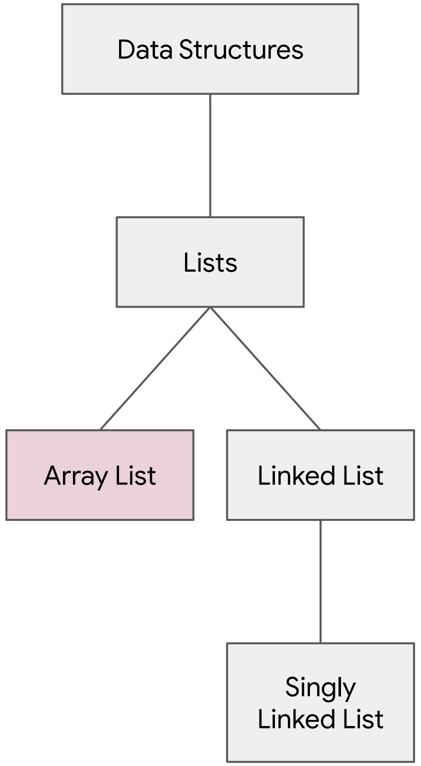
- Logic + Array

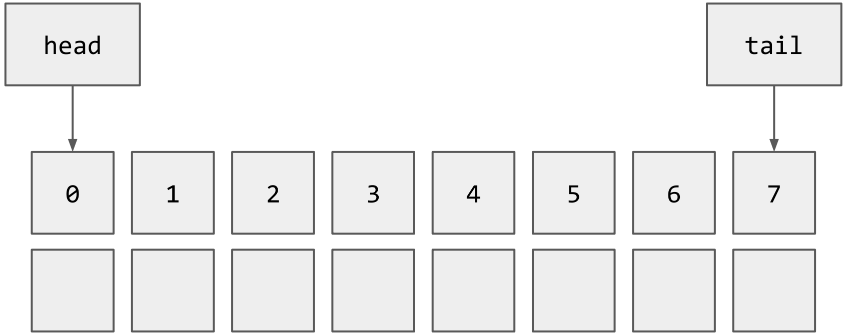
General Strengths

- Fast iterations
- Fast Indexing

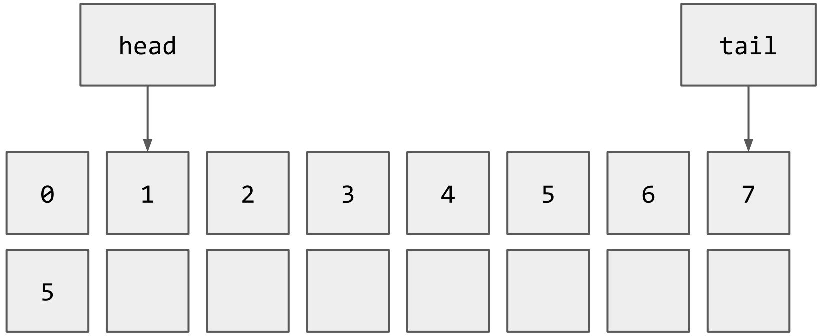
General Weaknesses

- "Growing" the array is slow

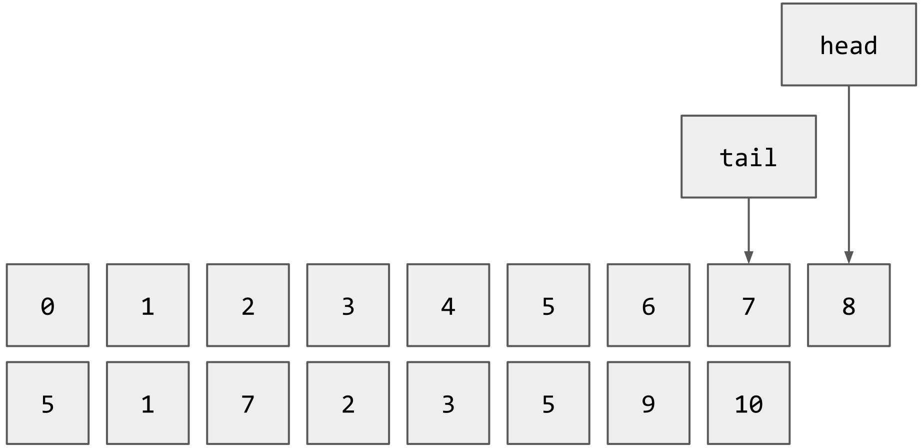


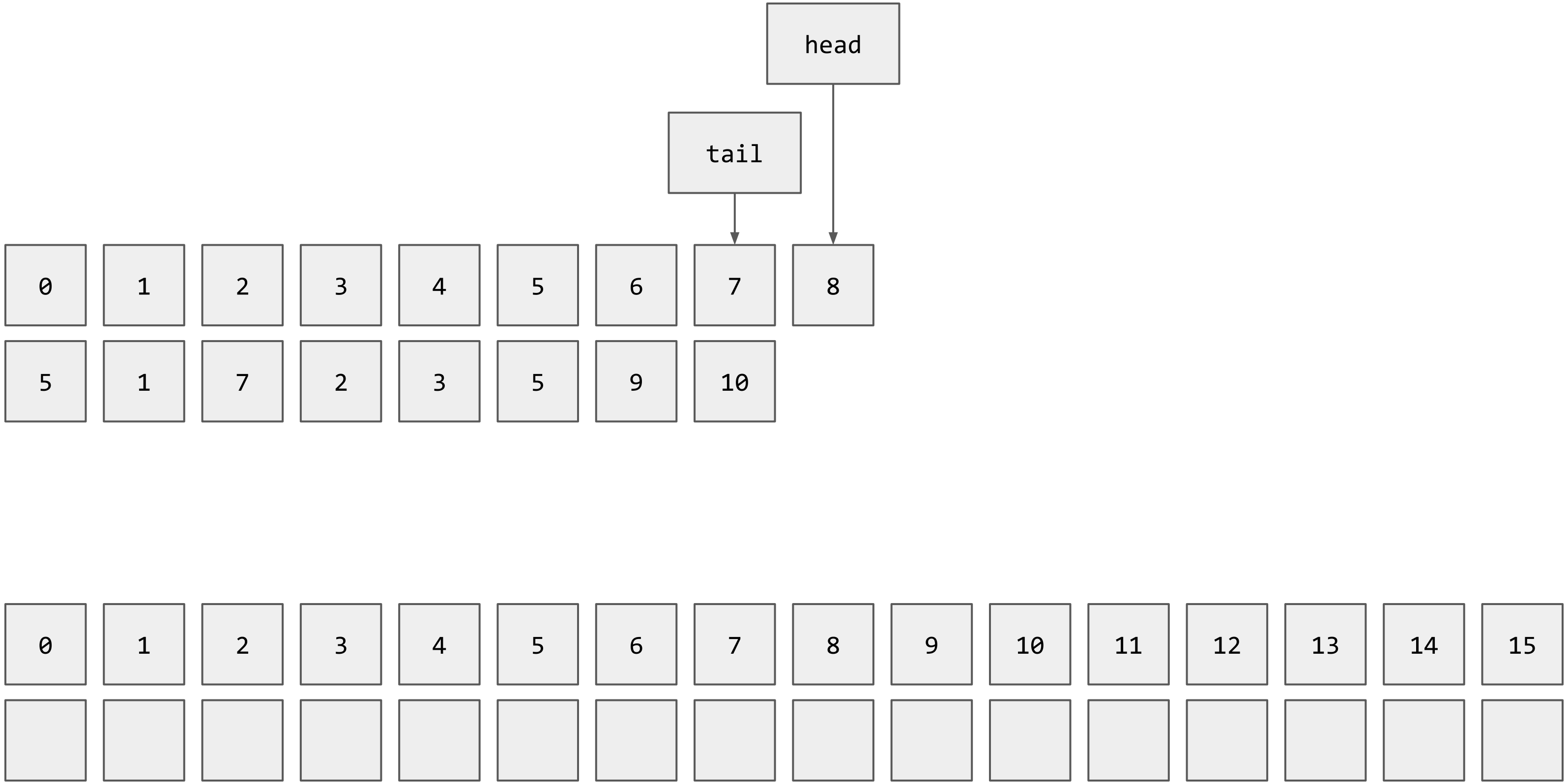

head
tail
0
1
2
3
4
5
6
7

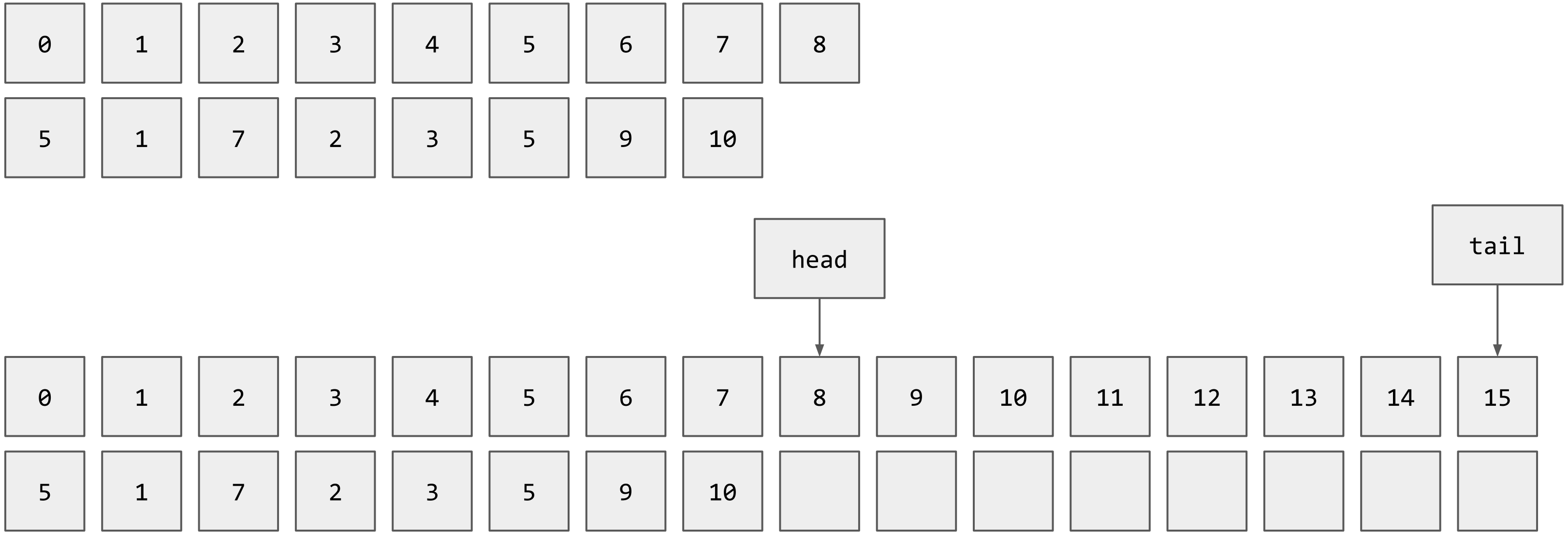
```
ArrayList<Integer> list = new ArrayList<>();

list.add(5);

list.size();  // 1
```



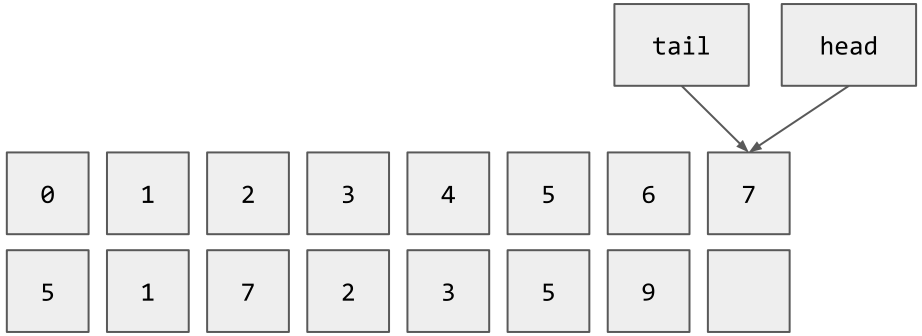
```
ArrayList<Integer> list = new ArrayList<>();

list.add(5);
list.add(1);
list.add(7);
list.add(2);
list.add(3);
list.add(5);
list.add(9);
list.add(10);

list.add(13); // There is no room!
```



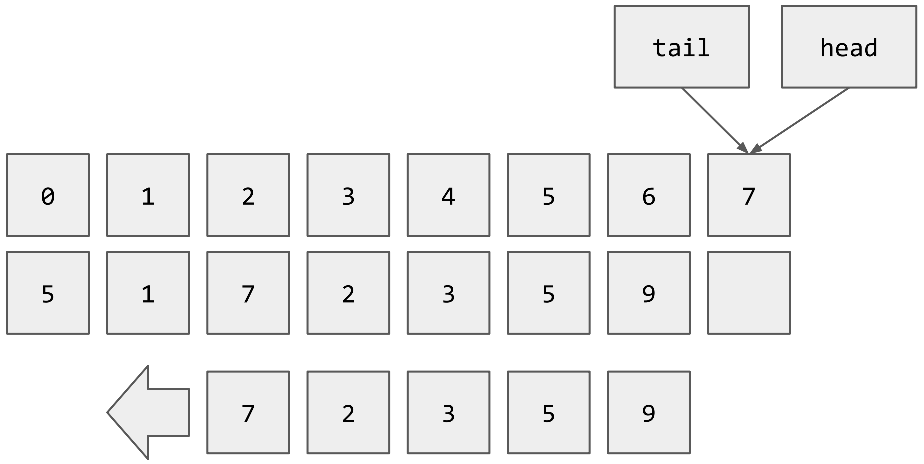

head
tail
0 1 2 3 4 5 6 7 8
5 1 7 2 3 5 9 10
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


0
1
2
3
4
5
6
7
8
5
1
7
2
3
5
9
10
head
tail
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
5
1
7
2
3
5
9
10

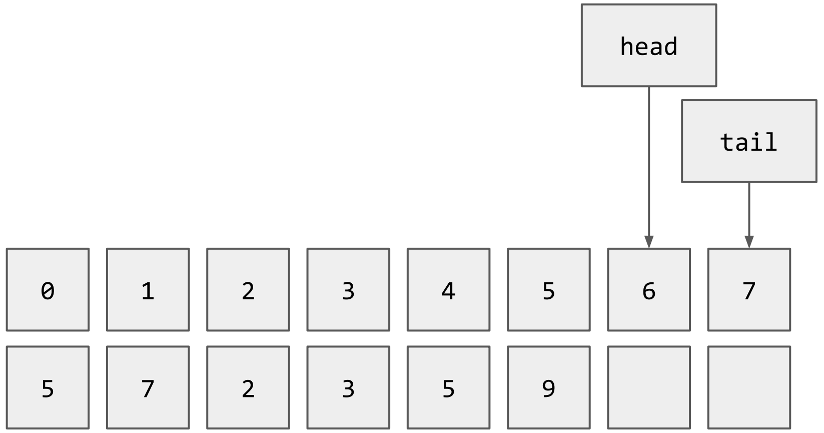
```
ArrayList<Integer> list = new ArrayList<>();

list.add(5);
list.add(1);
list.add(7);
list.add(2);
list.add(3);
list.add(5);
list.add(9);
list.add(10);

list.remove(1);
```



```
ArrayList<Integer> list = new ArrayList<>();

list.add(5);
list.add(1);
list.add(7);
list.add(2);
list.add(3);
list.add(5);
list.add(9);
list.add(10);

list.remove(1);
```



```
ArrayList<Integer> list = new ArrayList<>();

list.add(5);
list.add(1);
list.add(7);
list.add(2);
list.add(3);
list.add(5);
list.add(9);
list.add(10);

list.remove(1);
```



Array List Summary

General Strengths

- Good insertion
- Good (Fast) iteration
- Fast Indexing

General Weaknesses

- Poor look-up
- Poor removal
- "Growing" the array is slow

Runtime Analysis	
add value	O(1) amortized
remove by index	O(n)
get by index	O(1)
contains value	O(n)

Singly Linked Lists

Structure

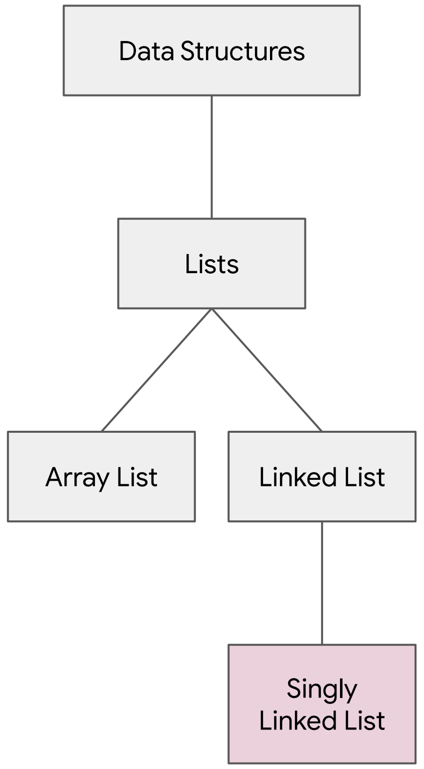
- Chaining "nodes" together

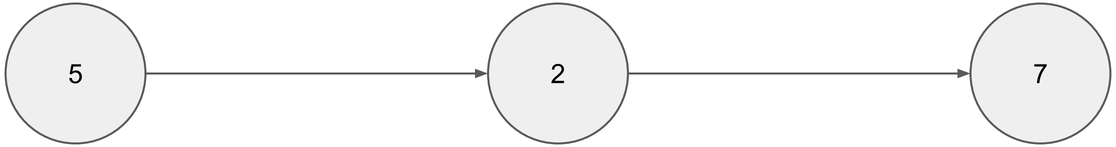
General Strengths

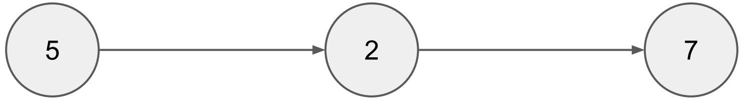
- Good middle-of-list insertion
- Good middle-of-list removals

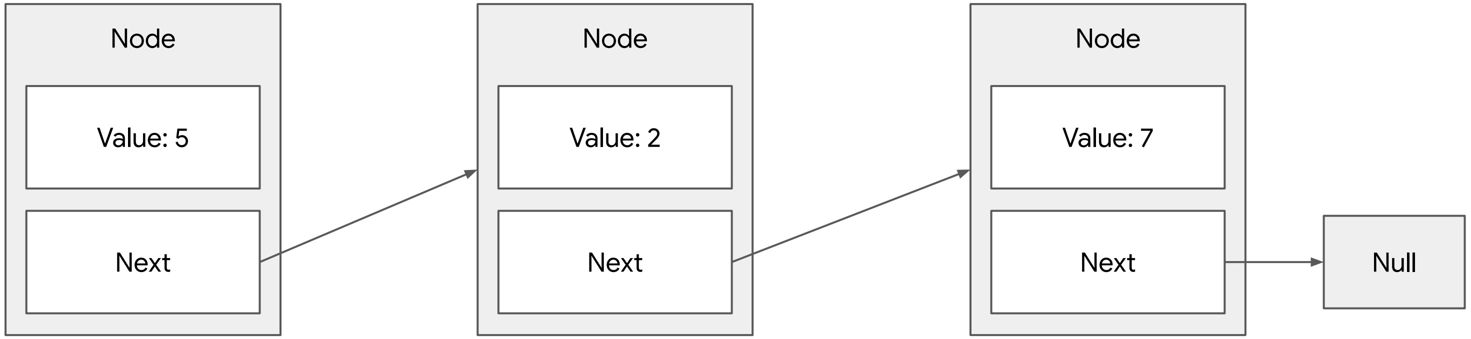
General Weaknesses

- Poor memory locality
- Poor indexing

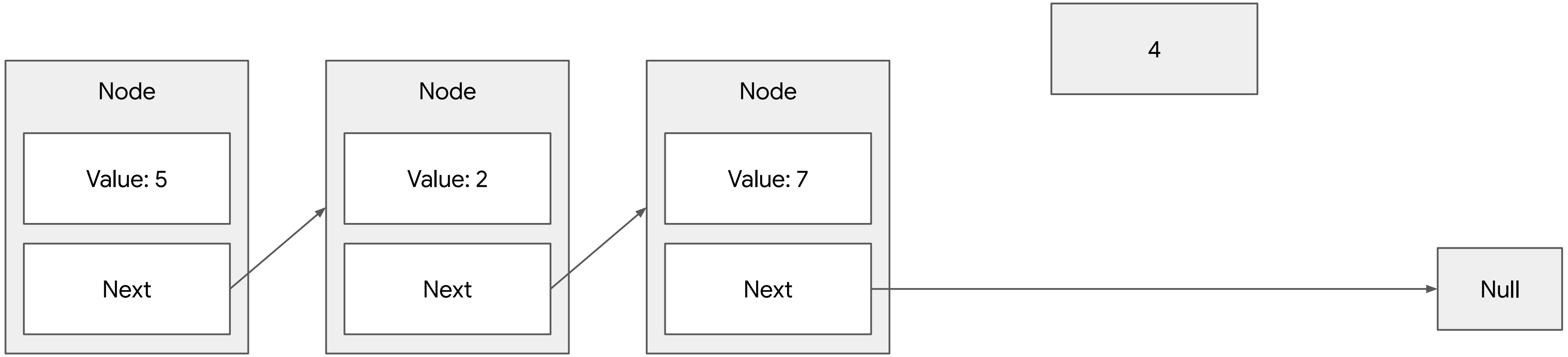



5
2
7

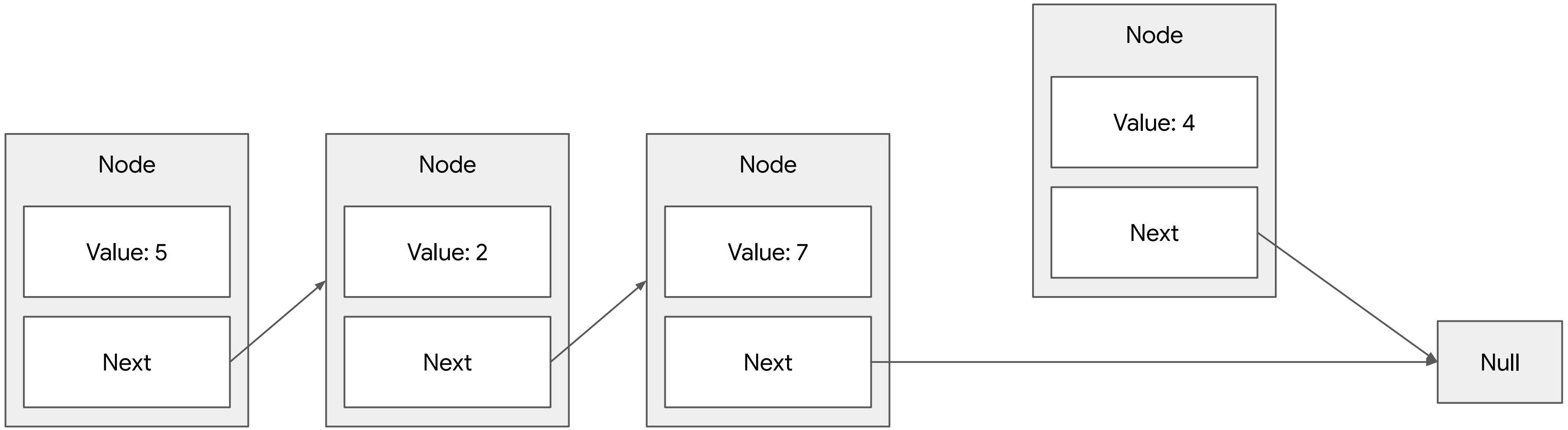

5
2
7


Node
Value: 5
Next
Node
Value: 2
Next
Node
Value: 7
Next
Null

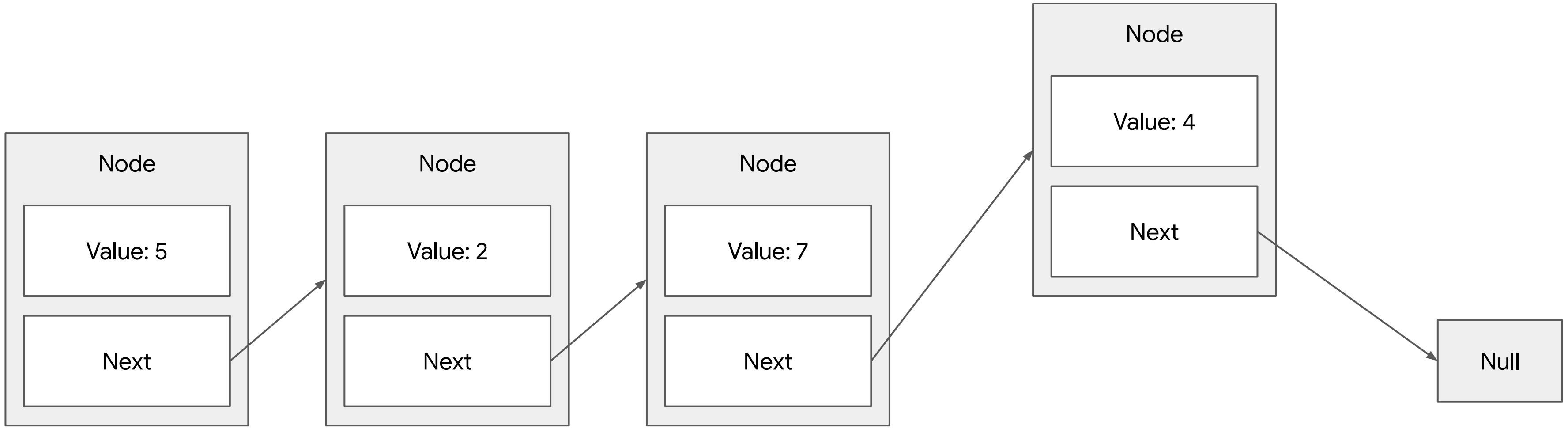
list.add(4)



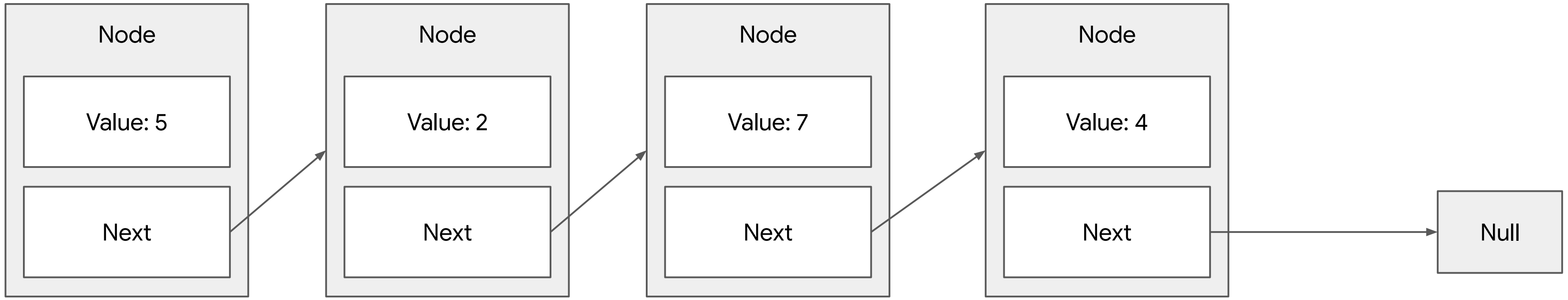
list.add(4)


Node
Value: 5
Next
Node
Value: 2
Next
Node
Value: 7
Next
Node
Value: 4
Next
Null

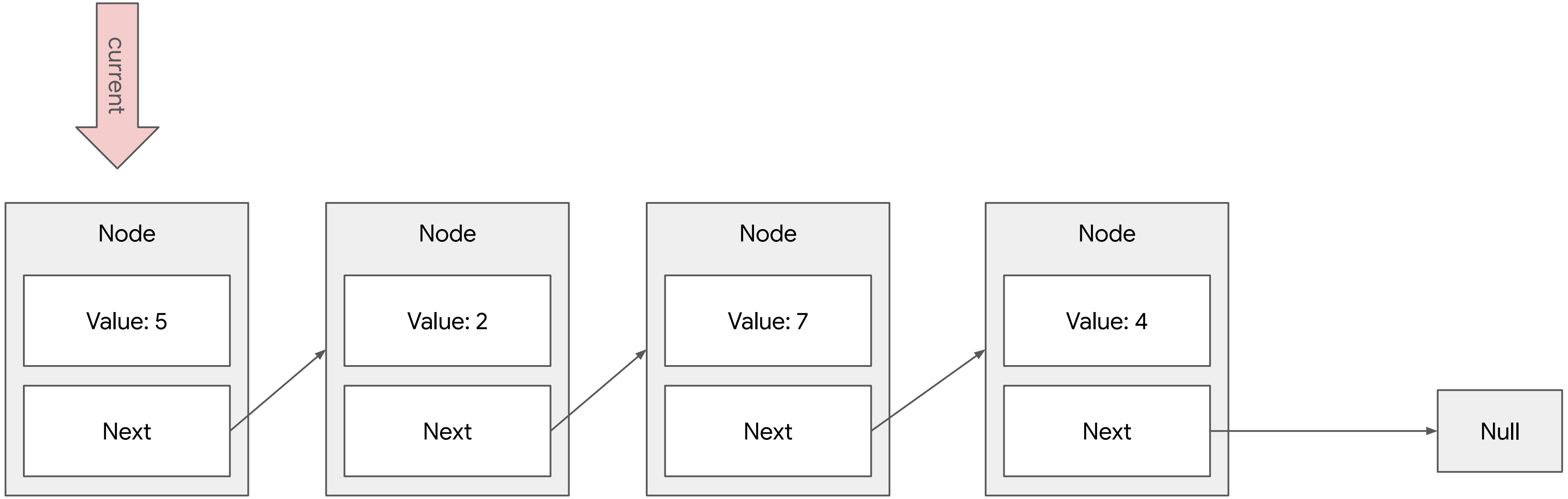
list.add(4)


Node
Value: 5
Next
Node
Value: 2
Next
Node
Value: 7
Next
Node
Value: 4
Next
Null

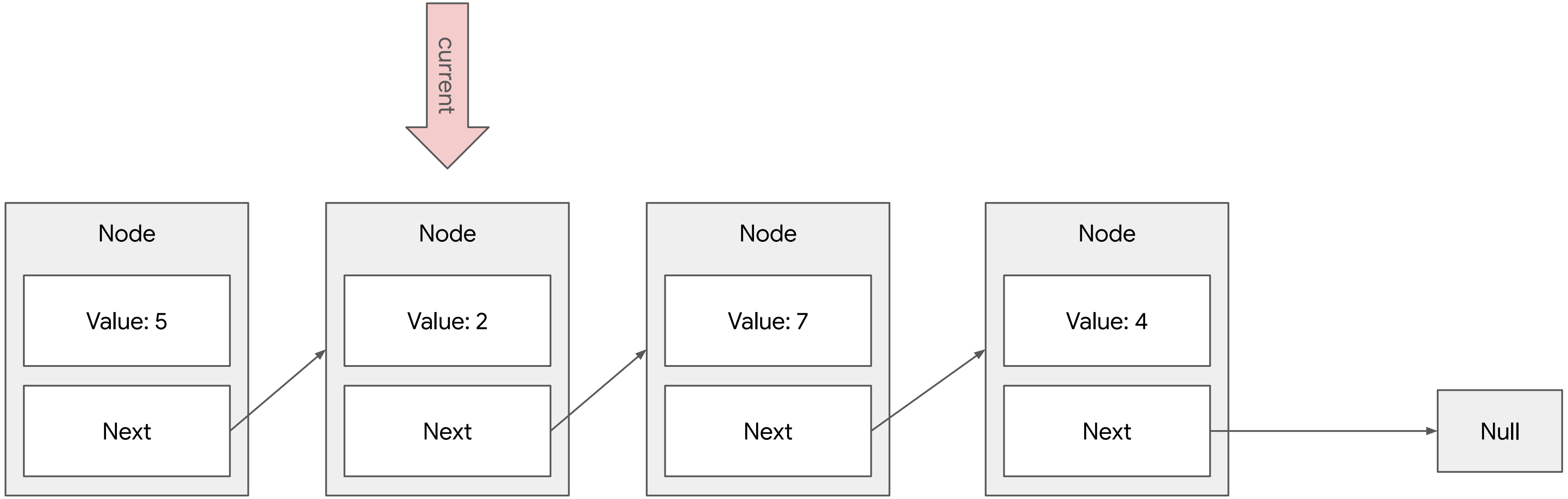
list.remove(7)



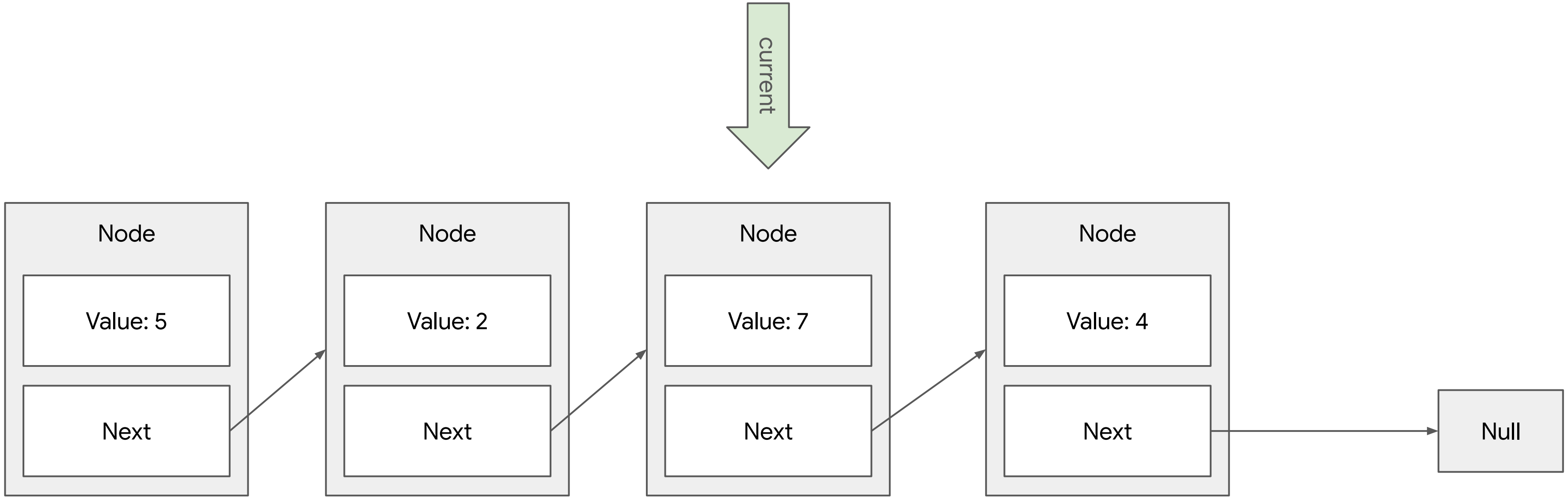
list.remove(7)



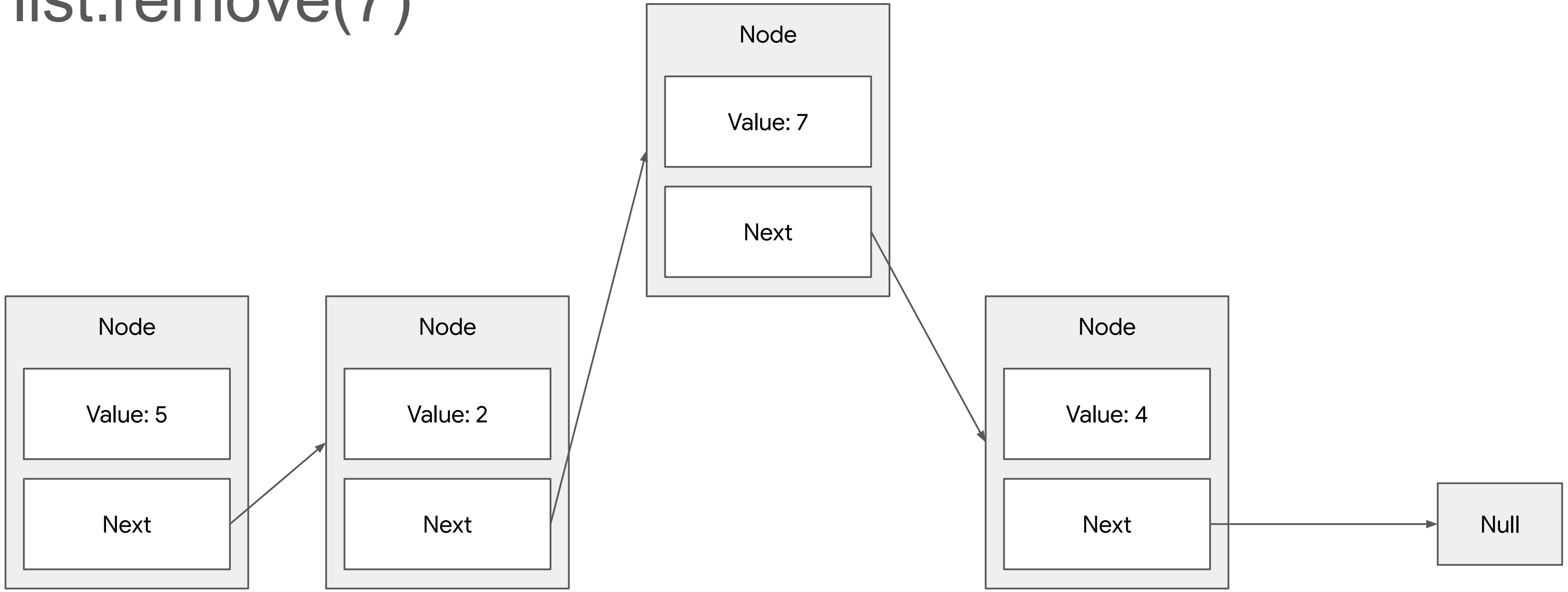
list.remove(7)



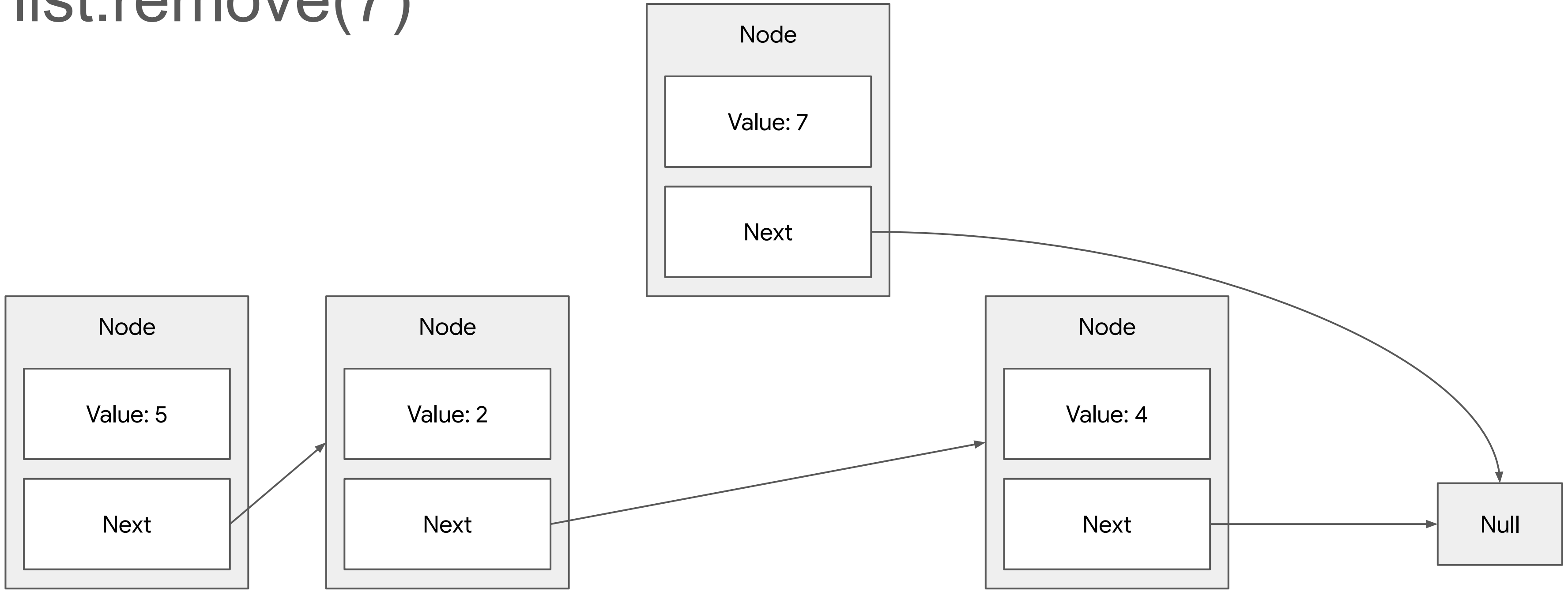
list.remove(7)



list.remove(7)



list.remove(7)



Singly Linked List Summary

General Strengths

- Good insertion
- Good (okay) iteration

General Weaknesses

- Poor indexing
- Poor look-up
- Poor removal
- Poor memory locality

Runtime Analysis	
add value	O(1)
remove by index	O(n)
get by index	O(n)
contains value	O(n)

Maps

Characteristics

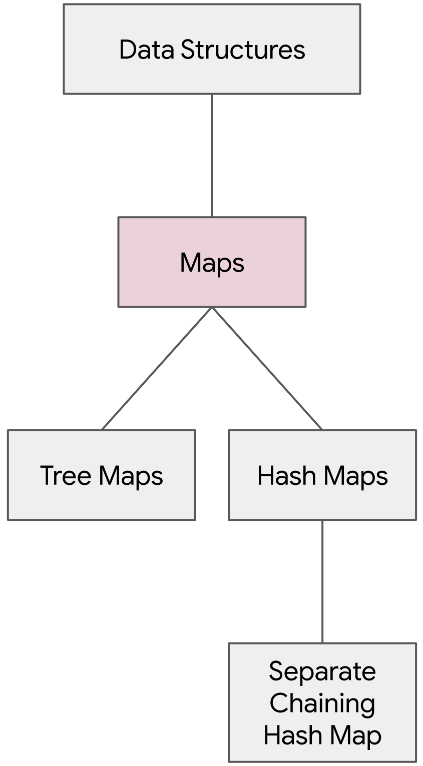
- Create associations between unique keys and non-unique values

General Strengths

- Good insertion
- Good removal
- Good look-up by value

General Weaknesses

- Poor iteration
- Poor memory locality
- Poor memory usage



Separate Chaining Hash Map

Characteristics

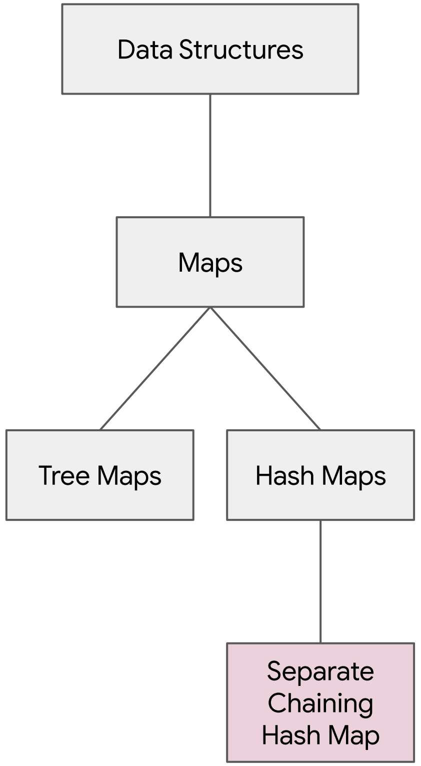
- Create associations between unique keys and non-unique values
- Unordered

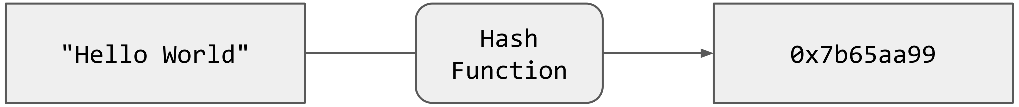
General Strengths

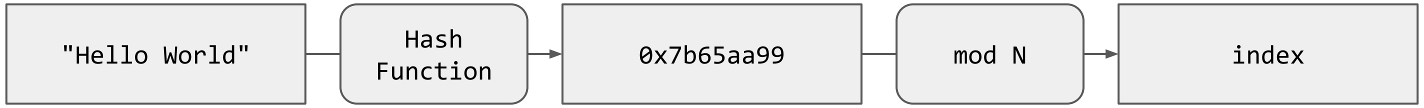
- Good insertion
- Good removal
- Good look-up by value

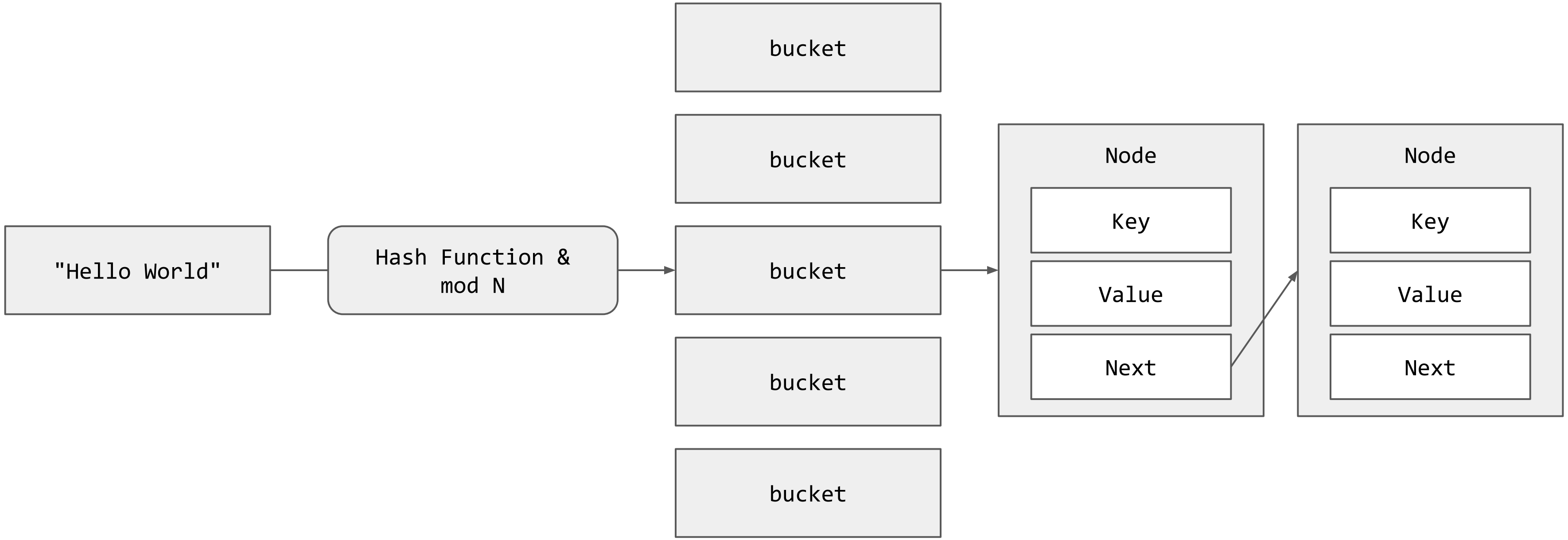
General Weaknesses

- Poor iteration
- Poor memory locality
- Poor memory usage




"Hello World"
Hash Function
0x7b65aa99


"Hello World"
Hash Function
0x7b65aa99
mod N
index


"Hello World"
Hash Function & mod N
bucket
bucket
bucket
bucket
bucket
Node
Key
Value
Next
Node
Key
Value
Next

Separate Chaining Hash Map

General Strengths

- Good insertion
- Good removal
- Good look-up by value

General Weaknesses

- Poor iteration
- Poor memory locality
- Poor memory usage

Runtime Analysis	
add value	O(1)*
remove by key	O(1)*
get by key	O(1)*
contains key	O(1)*
contains value	O(n)

Sets

Characteristics

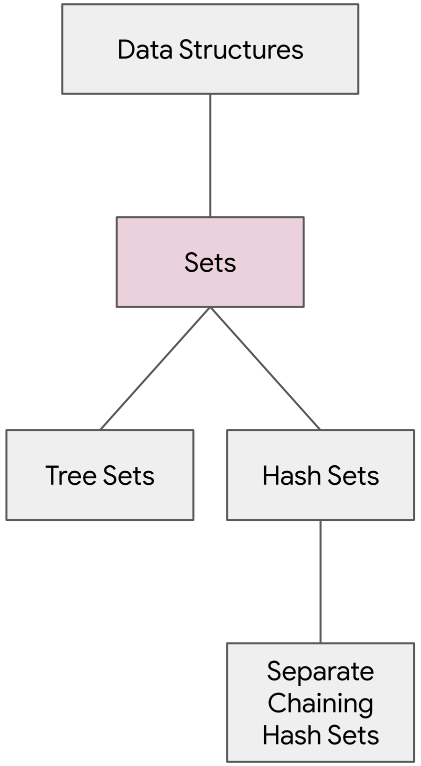
- Store unique values

General Strengths

- Good insertion
- Good removal
- Good look-up

General Weaknesses

- Poor iteration
- Poor memory locality
- Poor memory usage



Separate Chaining Hash Sets

Characteristics

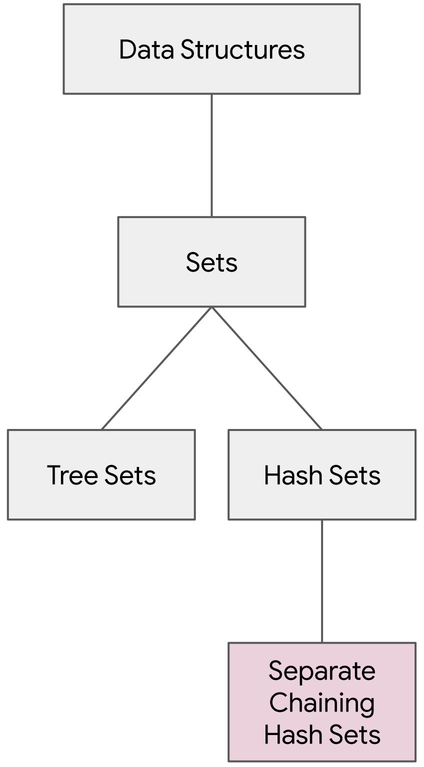
- Store unique values
- Unordered

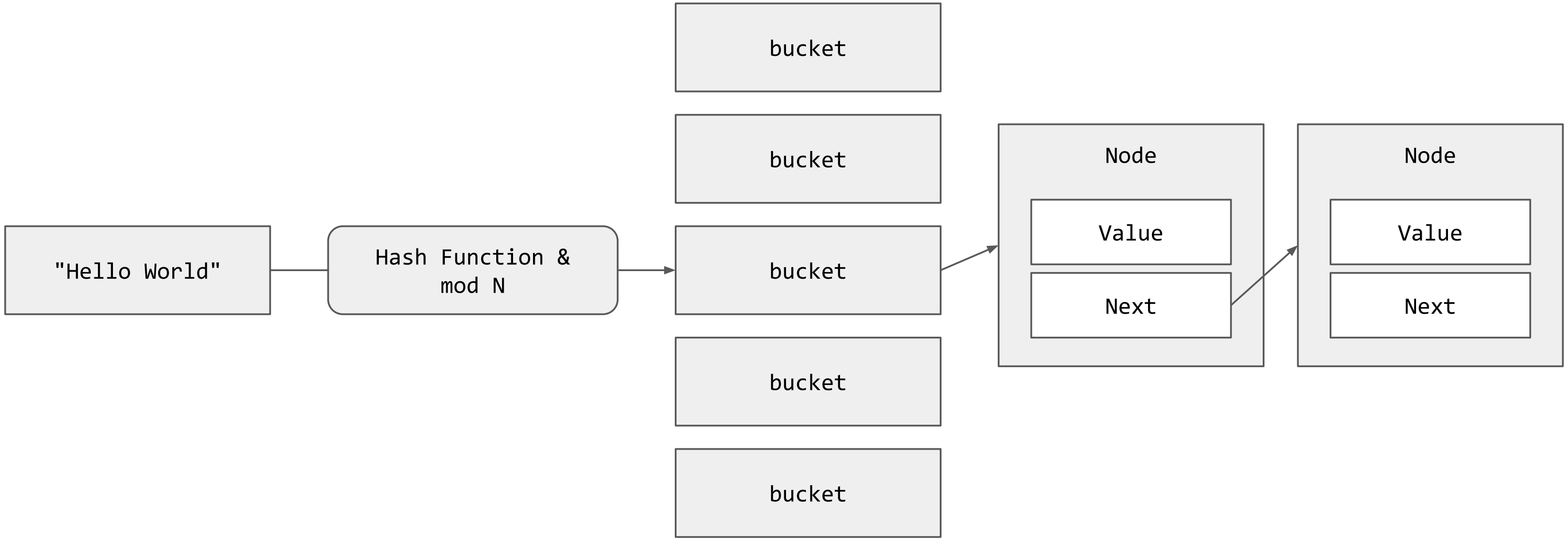
General Strengths

- Good insertion
- Good removal
- Good look-up

General Weaknesses

- Poor iteration
- Poor memory locality
- Poor memory usage




"Hello World"
Hash Function & mod N
bucket
bucket
bucket
bucket
bucket
Node
Value
Next
Node
Value
Next

Separate Chaining Hash Set

General Strengths

- Good insertion
- Good removal
- Good look-up

General Weaknesses

- Poor iteration
- Poor memory locality
- Poor memory usage

Runtime Analysis	
add value	O(1)*
remove value	O(1)*
contains value	O(1)*

Hash Functions

Objective

- Take arbitrary length input and produce fixed-length output.
- Have near random association with input and output.
- Deterministic

Important Notes

- Context
- Good Enough

(Poor) Hashing Examples

```
int simpleHash(char[] string) {
  int hash = 0;

  for (char c : string) {
    hash += c;
  }

  return hash;
}
```

What does it do?

Speed?

Collisions?

Final opinion?

(Poor) Hashing Examples

```
int bitRotateHash(char[] string) {
  int hash = 0;

  for (char c : string) {
    int high = (hash >> 24) & (0xFF);
    hash = (hash << 8) | (high ^ c);
  }

  return hash;
}
```

What does it do?

Speed?

Collisions?

Final opinion?

Reflection

Primary Objectives

- What is a **list** and when could I use it?
- What is a **set** and when could I use it?
- What is a **map** and when could I use it?

Secondary Objectives

- What trade-offs am I making when I use a **list**?
- What trade-offs am I making when I use a **set**?
- What trade-offs am I making when I use a **map**?