Datastructures
Lecture 11

Andreas Wieden

Outline

- Heap sort.
- Bucket sort
- Quick sort.

Why study Heapsort?

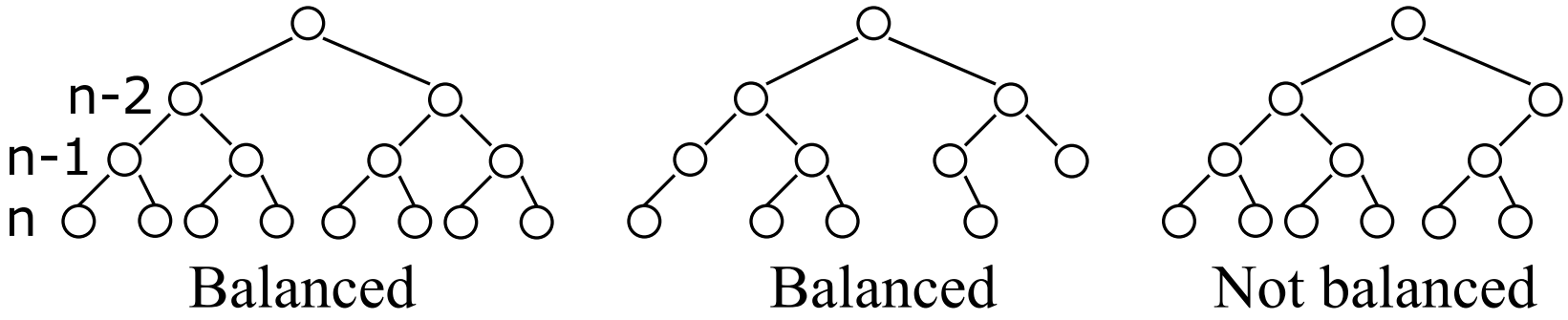
- It is a well-known, traditional sorting algorithm you will be expected to know
- Heapsort is *always* O(n log n), even in worst case.

What is a “heap”?

- Definitions of heap:
  1. A large area of memory from which the programmer can allocate blocks as needed, and deallocate them (or allow them to be garbage collected) when no longer needed
  2. A balanced, left-justified binary tree in which no node has a value greater than the value in its parent
- These two definitions have little in common
- Heapsort uses the second definition

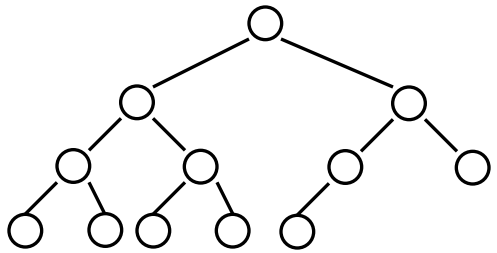
Balanced binary trees

- Recall:
  - The depth of a node is its distance from the root
  - The depth of a tree is the depth of the deepest node
- A binary tree of depth n is balanced if all the nodes at depths 0 through n-2 have two children

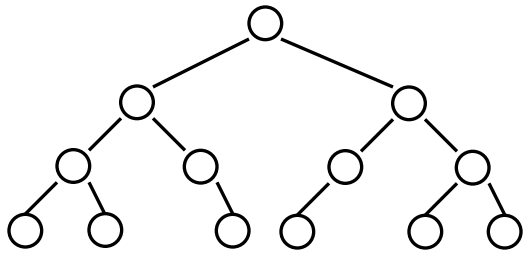


Left-justified binary trees

- A balanced binary tree is left-justified if:
  - all the leaves are at the same depth, or
  - all the leaves at depth n+1 are to the left of all the nodes at depth n



Left-justified



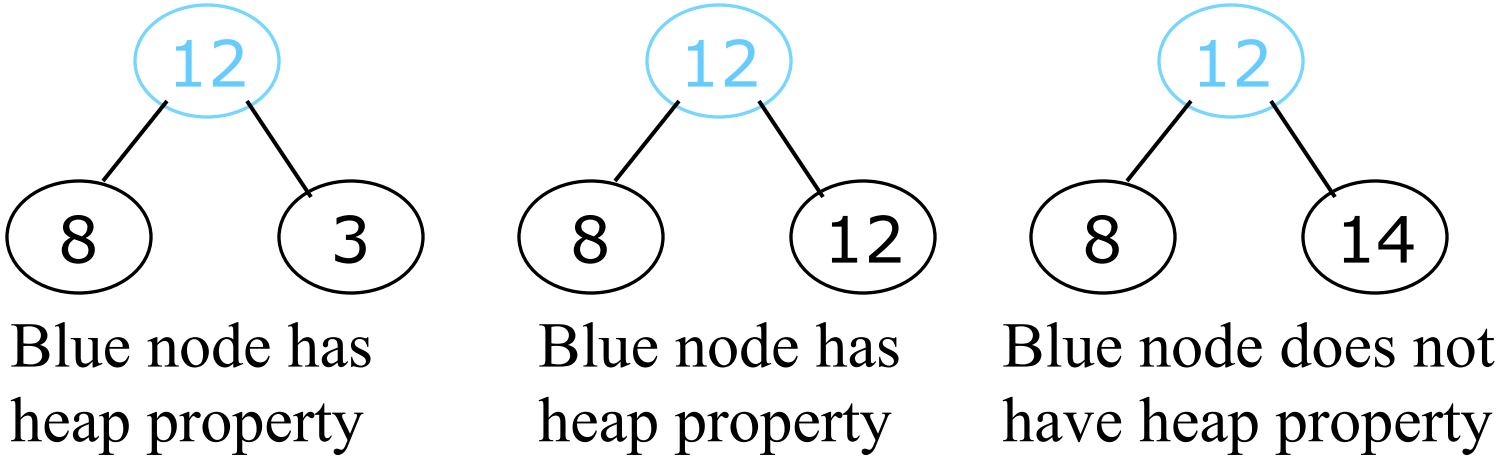
Not left-justified

Plan of attack

- First, we will learn how to turn a binary tree into a heap
- Next, we will learn how to turn a binary tree *back* into a heap after it has been changed in a certain way
- We will use these ideas to sort an *array*

The heap property

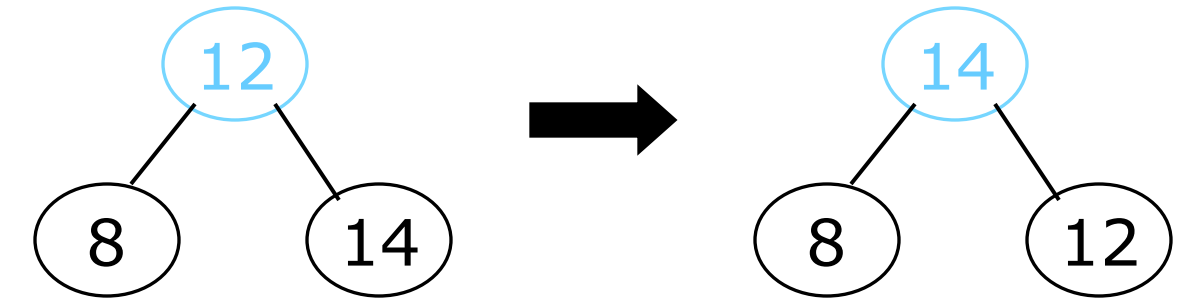
- A node has the heap property if the value in the node is as large as or larger than the values in its children



- All leaf nodes automatically have the heap property
- A binary tree is a heap if *all* nodes in it have the heap property

Bubble up

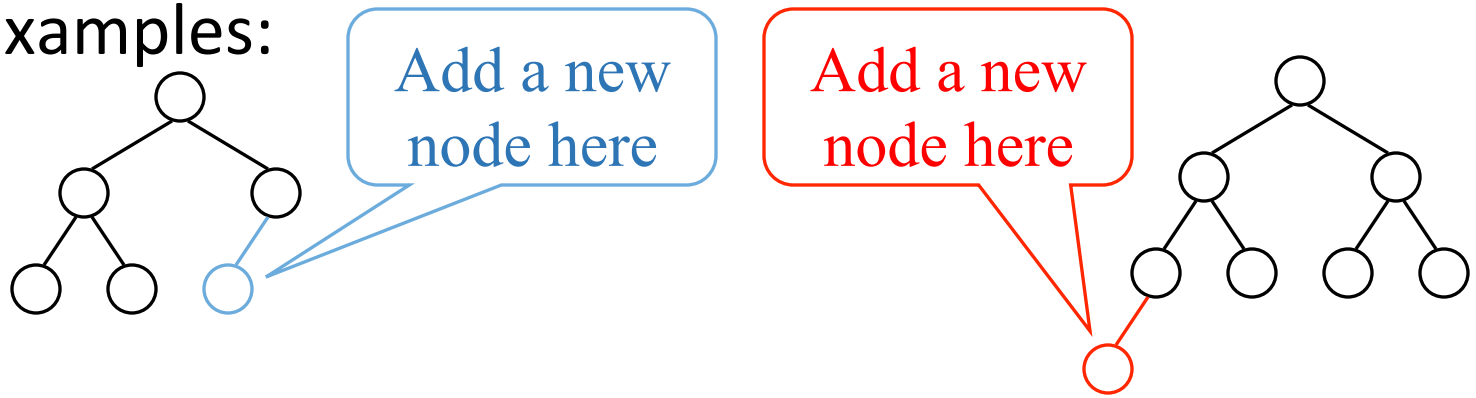
- Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child



- This is sometimes called bubble up
- Notice that the child may have *lost* the heap property

Constructing a heap I

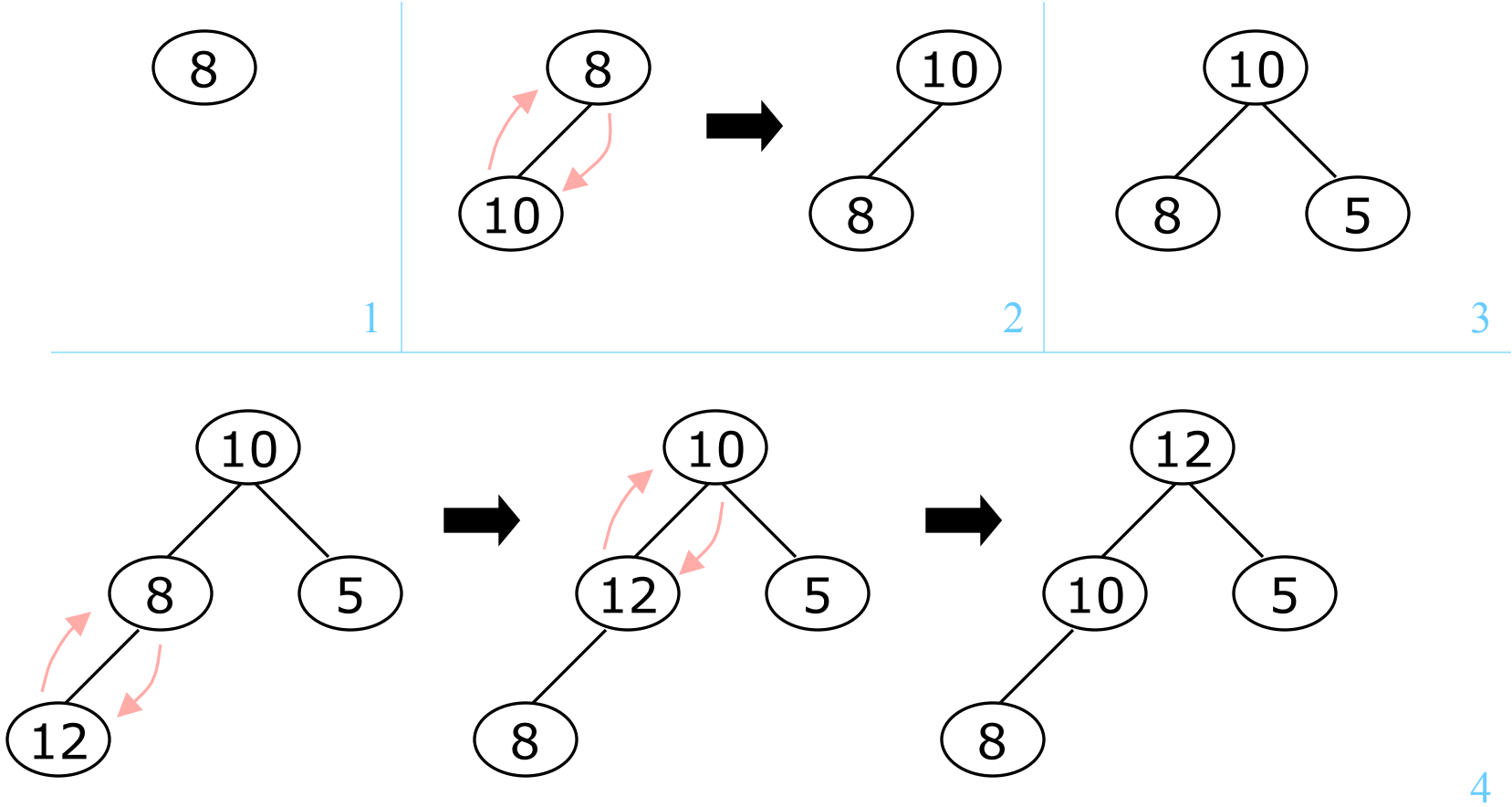
- A tree consisting of a single node is automatically a heap
- We construct a heap by adding nodes one at a time:
  - Add the node just to the right of the rightmost node in the deepest level
  - If the deepest level is full, start a new level
- Examples:



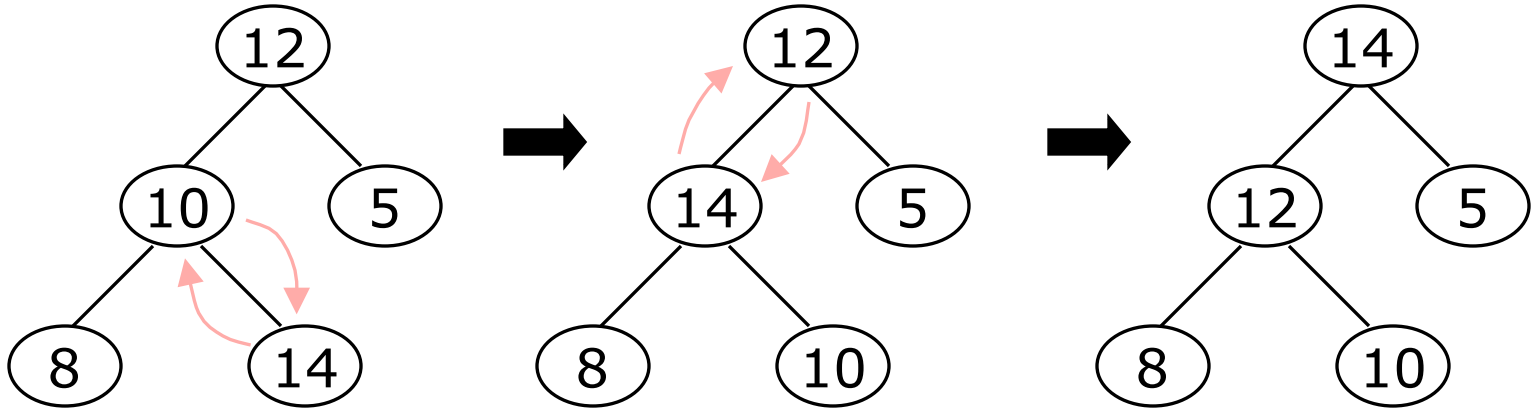
Constructing a heap II

- Each time we add a node, we may destroy the heap property of its parent node
- To fix this, we bubble up
- But each time we bubble up, the value of the topmost node in the swap may increase, and this may destroy the heap property of *its* parent node
- We repeat the bubble up process, moving up in the tree, until either
  - We reach nodes whose values don't need to be swapped (because the parent is *still* larger than both children), or
  - We reach the root

Constructing a heap III



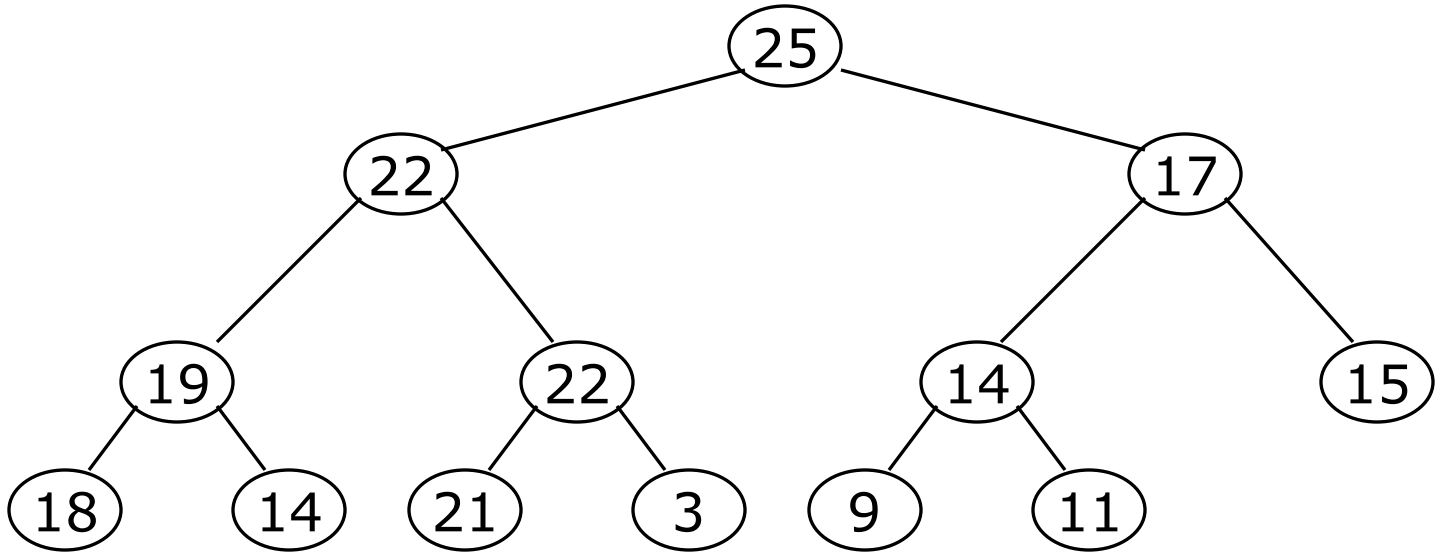
Other children are not affected



- The node containing 8 is not affected because its parent gets larger, not smaller
- The node containing 5 is not affected because its parent gets larger, not smaller
- The node containing 8 is still not affected because, although its parent got smaller, its parent is still greater than it was originally

A sample heap

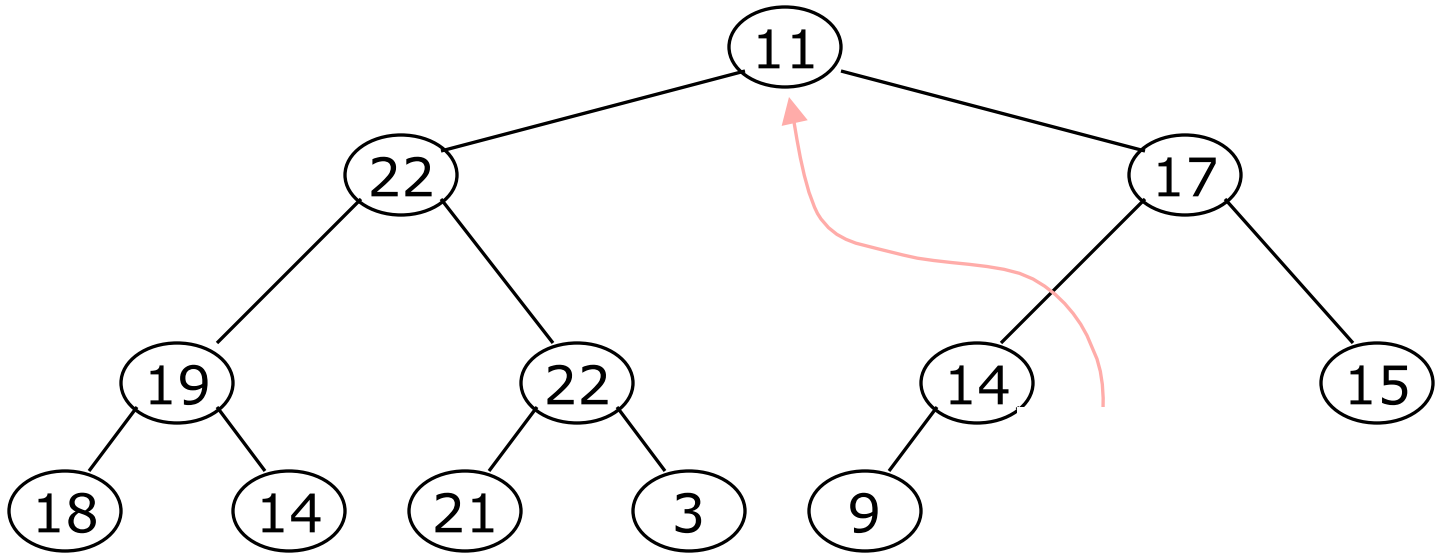
- Here's a sample binary tree after it has been heapified



- Notice that heapified does *not* mean sorted
- Heapifying does *not* change the shape of the binary tree; this binary tree is balanced and left-justified because it started out that way

Removing the root

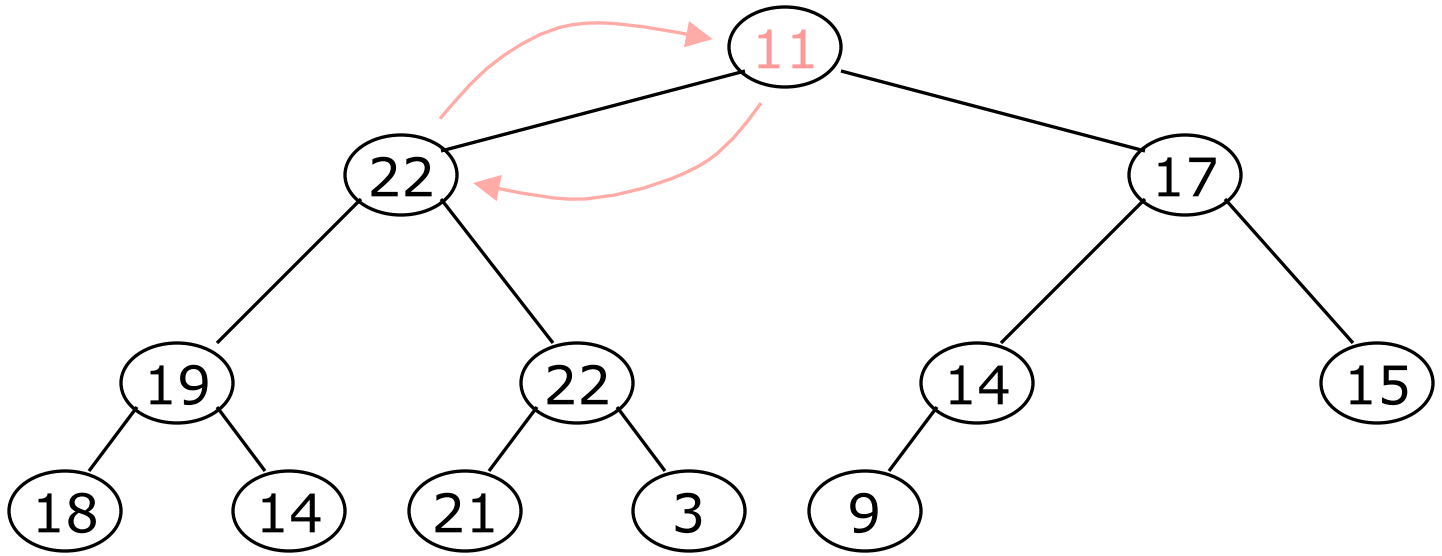
- Notice that the largest number is now in the root
- Suppose we *discard* the root:



- How can we fix the binary tree so it is once again *balanced and left-justified?*
- Solution: remove the rightmost leaf at the deepest level and use it for the new root

The **reHeap** method I

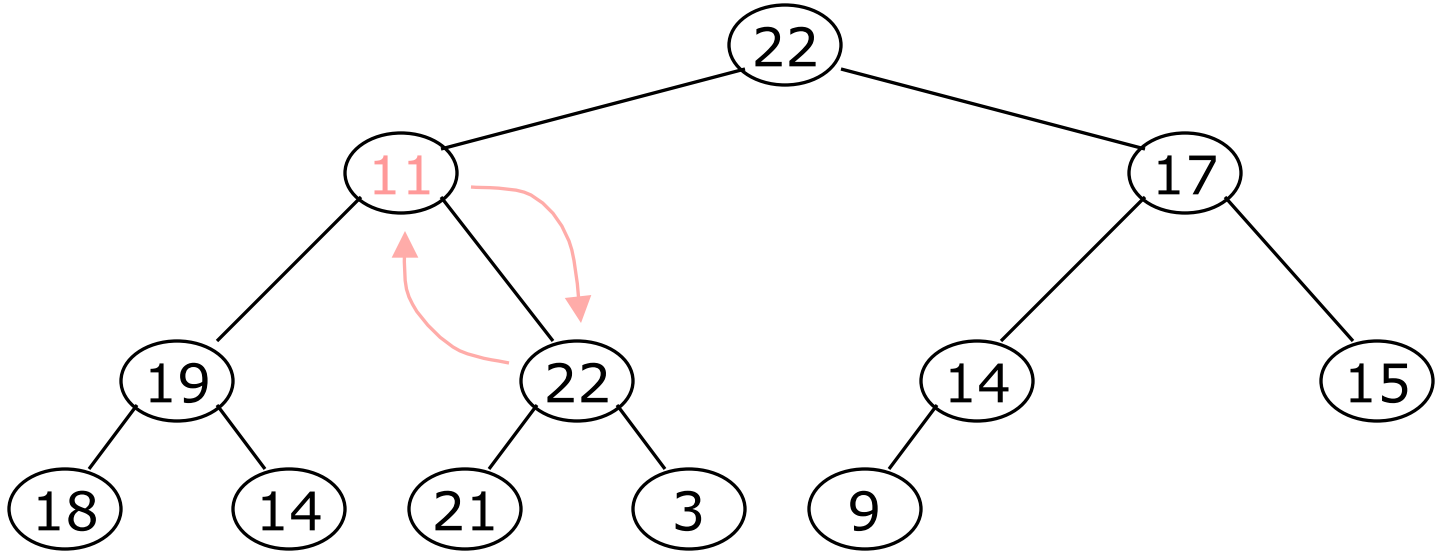
- Our tree is balanced and left-justified, but no longer a heap
- However, *only the root* lacks the heap property



- We can bubble down the root
- After doing this, one and only one of its children may have lost the heap property

The **reHeap** method II

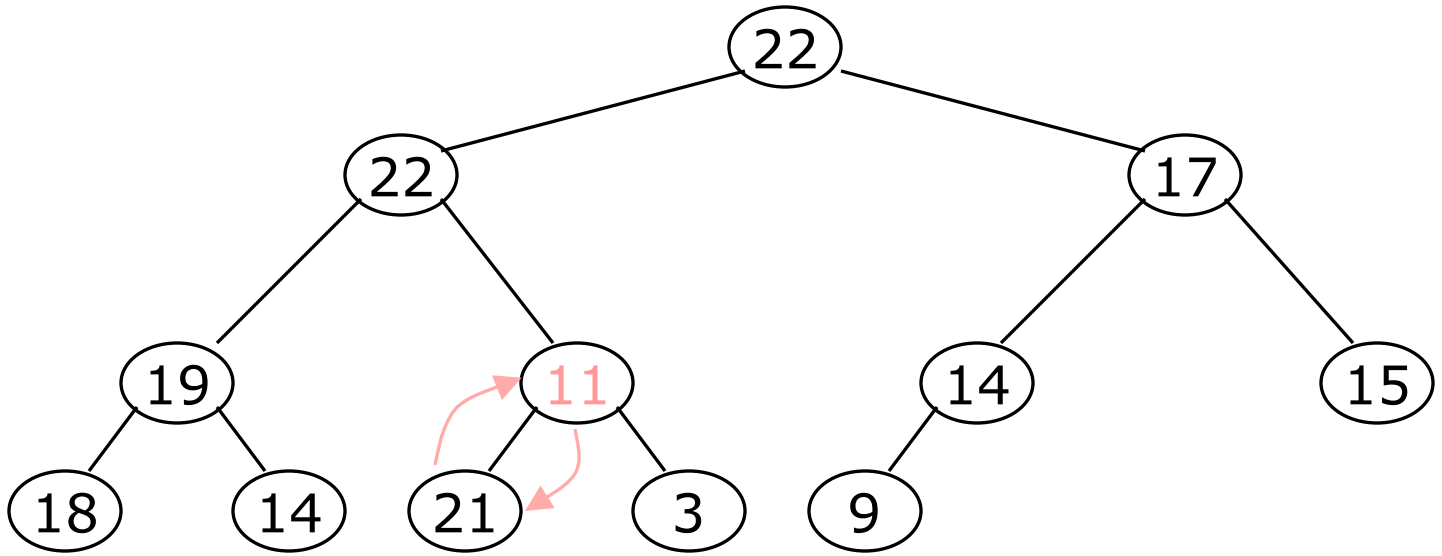
- Now the left child of the root (still the number `11`) lacks the heap property



- We can bubble down this node
- After doing this, one and only one of its children may have lost the heap property

The **reHeap** method III

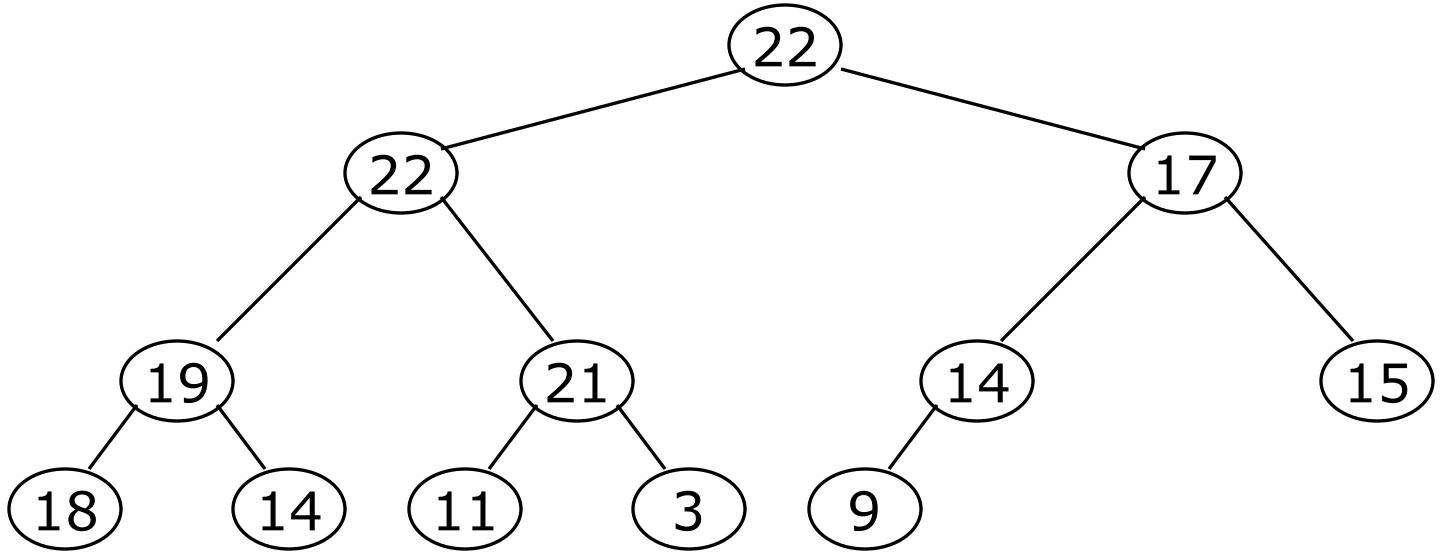
- Now the right child of the left child of the root (still the number `11`) lacks the heap property:



- We can bubble down this node
- After doing this, one and only one of its children may have lost the heap property —but it doesn't, because it's a leaf

The **reHeap** method IV

- Our tree is once again a heap, because every node in it has the heap property



- Once again, the largest (or *a* largest) value is in the root
- We can repeat this process until the tree becomes empty
- This produces a sequence of values in order largest to smallest

Heap sort

A simple sorting algorith

- Insert each element into a binary heap.
- Execute delete-min and insert into an new array until the binary heap is empty.

Heap sort

One variant is to use the same array by inserting the removed elements in the end of the array.

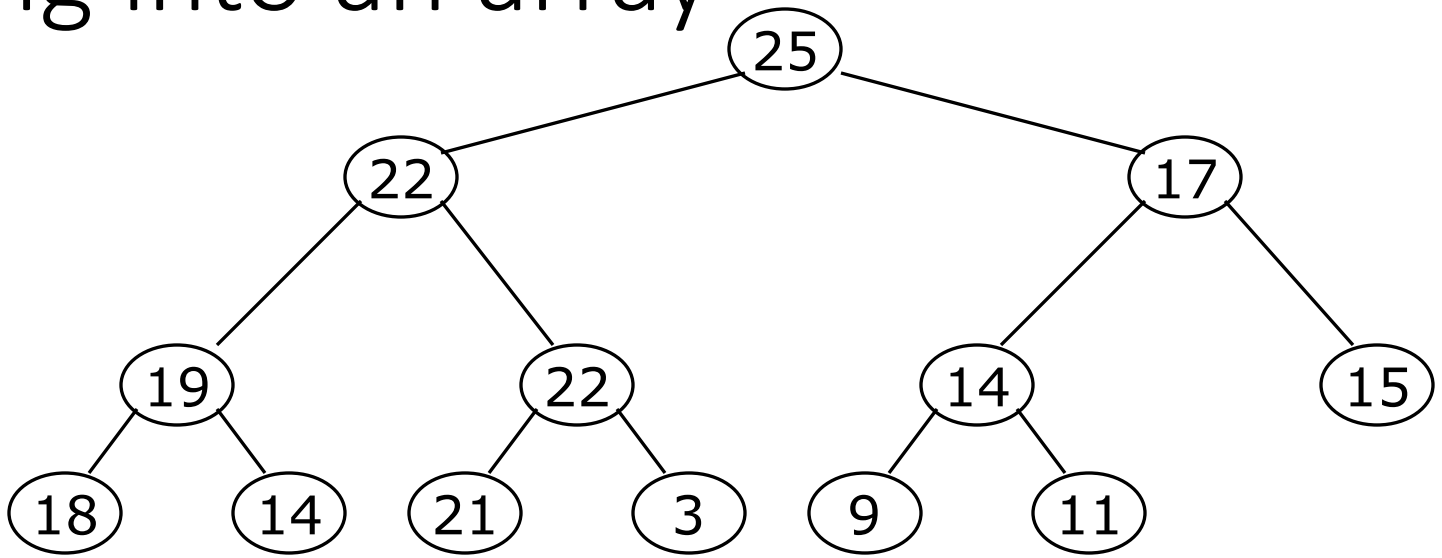
Sorting

What do heaps have to do with sorting an array?

- Because the binary tree is *balanced* and *left justified,* it can be represented as an array
- All our operations on binary trees can be represented as operations on *arrays*
- To sort:

```
heapify the array;
while the array isn't empty {
    remove and replace the root;
    reheap the new root node;
}
```

Mapping into an array

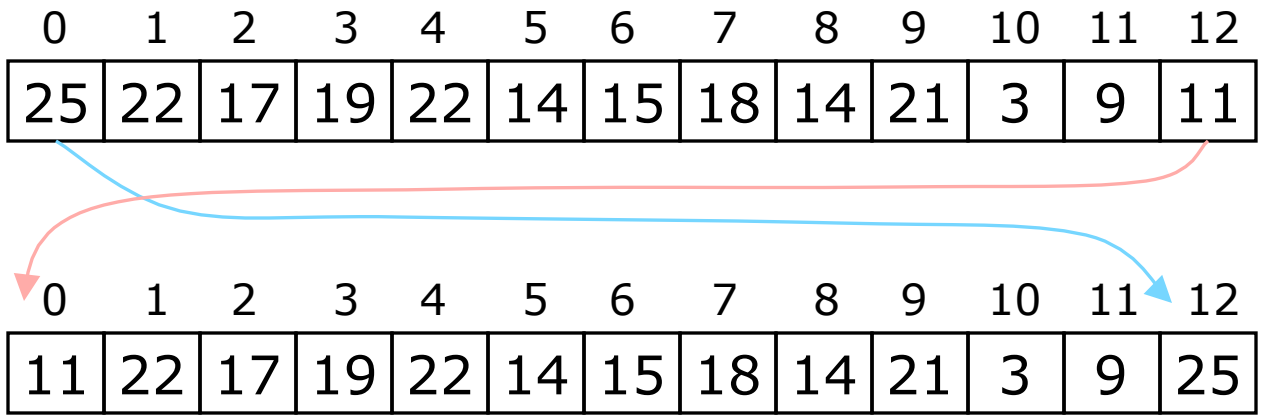


0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

- Notice:
  - The left child of index i is at index 2*i+1
  - The right child of index i is at index 2*i+2
  - Example: the children of node 3 (19) are 7 (18) and 8 (14)

Removing and replacing the root

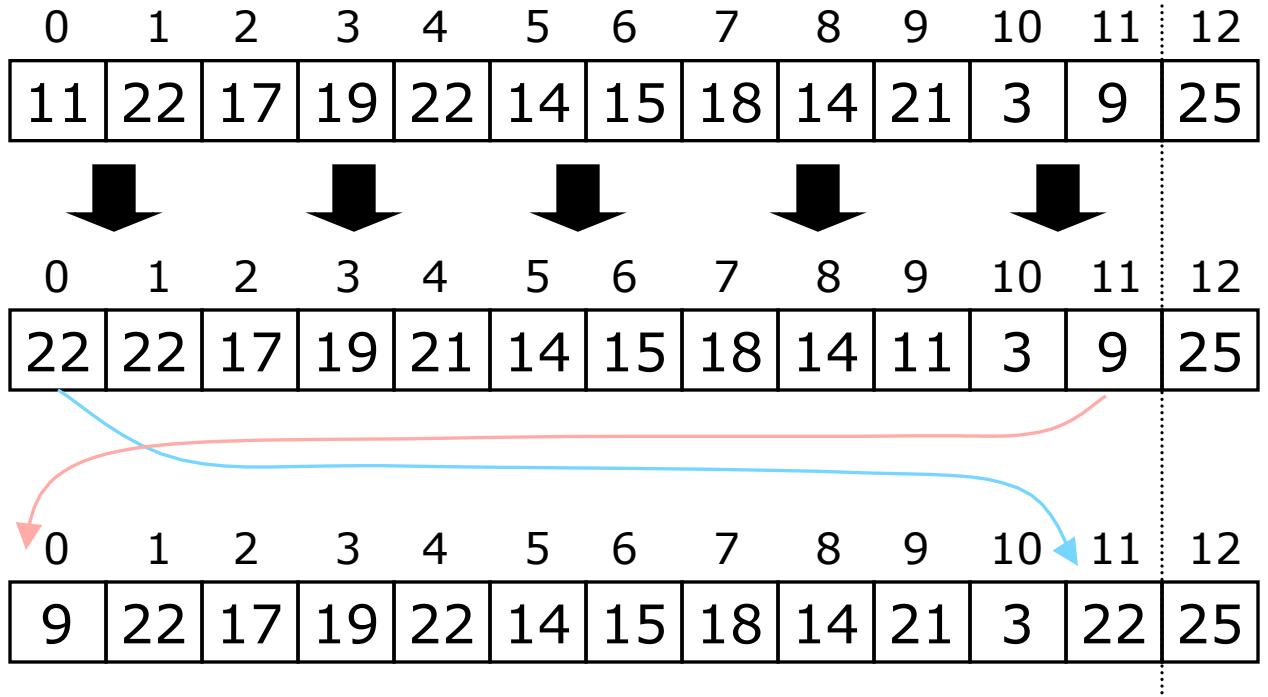
- The “root” is the first element in the array
- The “rightmost node at the deepest level” is the last element
- Swap them...



- ...And pretend that the last element in the array no longer exists—that is, the “last index” is `11` (9)

Reheap and repeat

- Reheap the root node (index 0, containing 11)...



- ...And again, remove and replace the root node
- Remember, though, that the “last” array index is changed
- Repeat until the last becomes first, and the array is sorted!

Time Complexity

Assumption: Compare operations take constant time.

Time Complexity

Compare operations take constant time.

The depth of a specifik leaf= The number of comparisons to sort a list.

- Hight of the tree = Worst case
- The depth of the depest node is log n.

Time Complexity

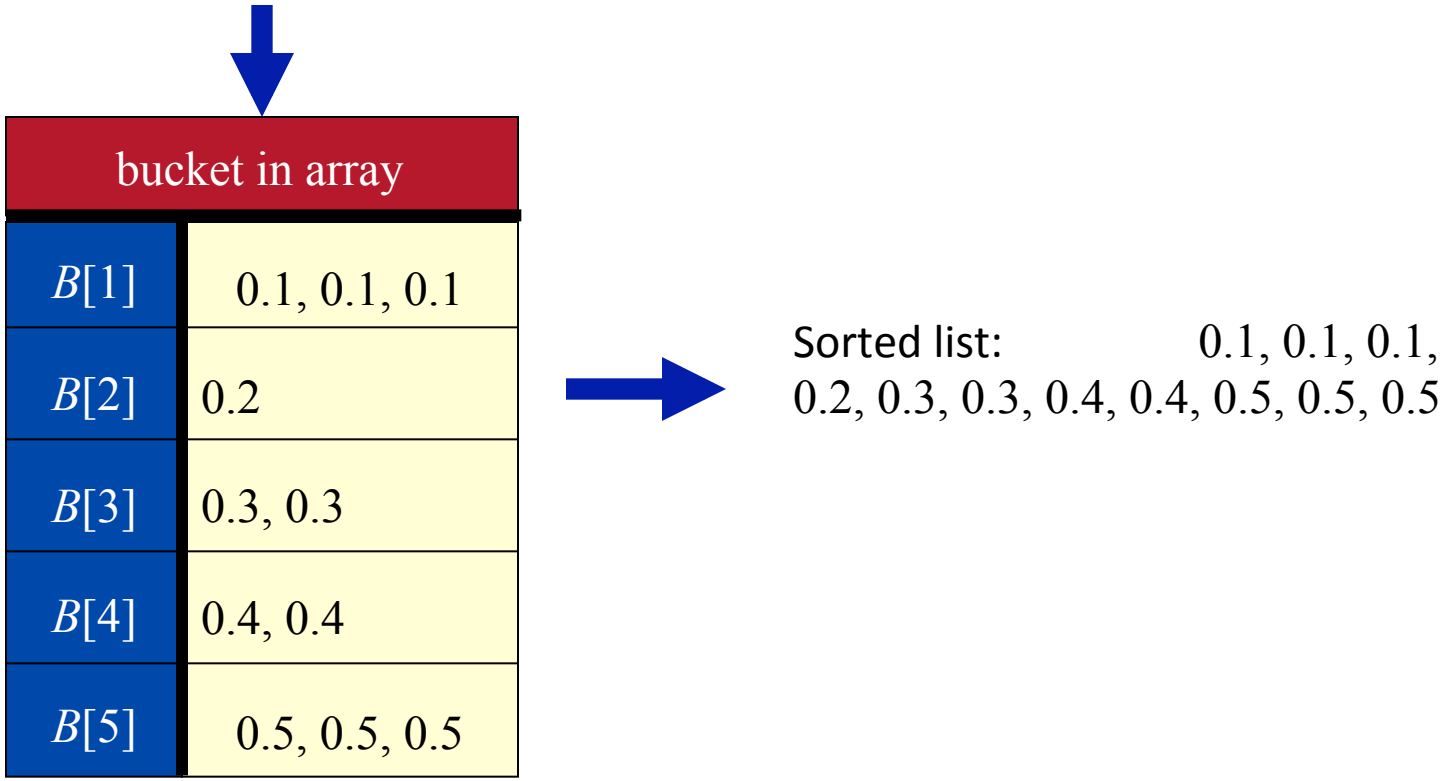
Sorting using Heapsort is O(n log n)

Bucket Sort

- Given that, we have N integers in the range 0 to M-1
- Maintain an array count of size M, which is initialized to zero. Thus, count has M cells (buckets).
- Read A_i
- Increment count[A_i] by 1
- After all the input array is read, scan the count array, printing out a representation of sorted list.

Bucket Sort Example

- A=(0.5, 0.1, 0.3, 0.4, 0.3, 0.2, 0.1, 0.1, 0.5, 0.4, 0.5)



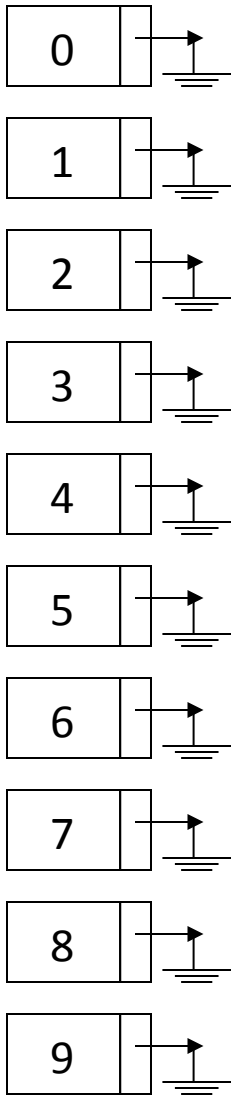
Radix Sort

- Radix sort is generalization of bucket sort.
- It uses several passes of bucket sort
- Perform the bucket sorts by “least significant digits”
  - First sort by digit in units place
  - Second sort by digit in tens place
  - Third sort by digit in hundreds place
  - …….

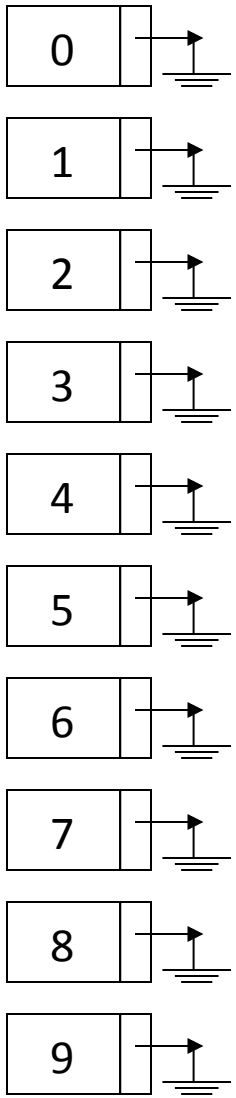
Radix sort, an example:

64	8	216	512	27	729	0	1	343	125

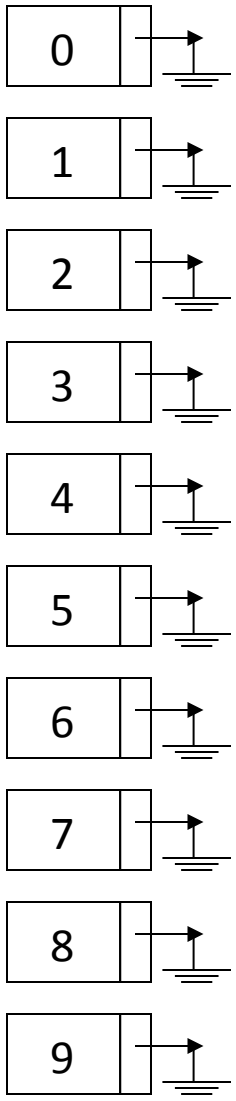
Pass 1


0
1
2
3
4
5
6
7
8
9

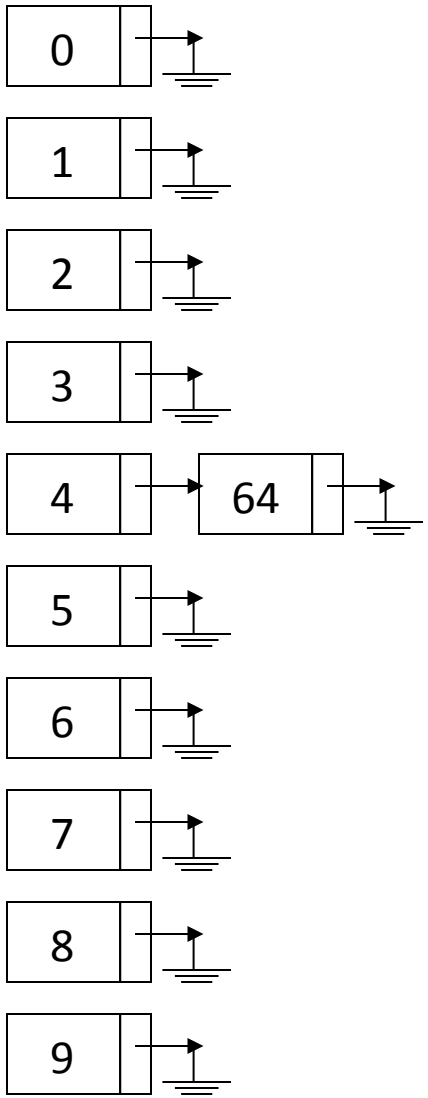
64	8	216	512	27	729	0	1	343	125



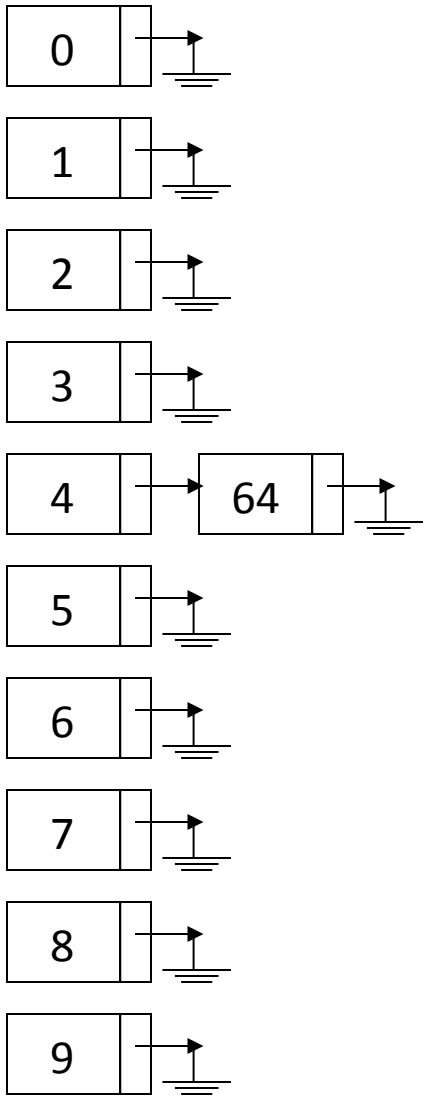
64	8	216	512	27	729	0	1	343	125



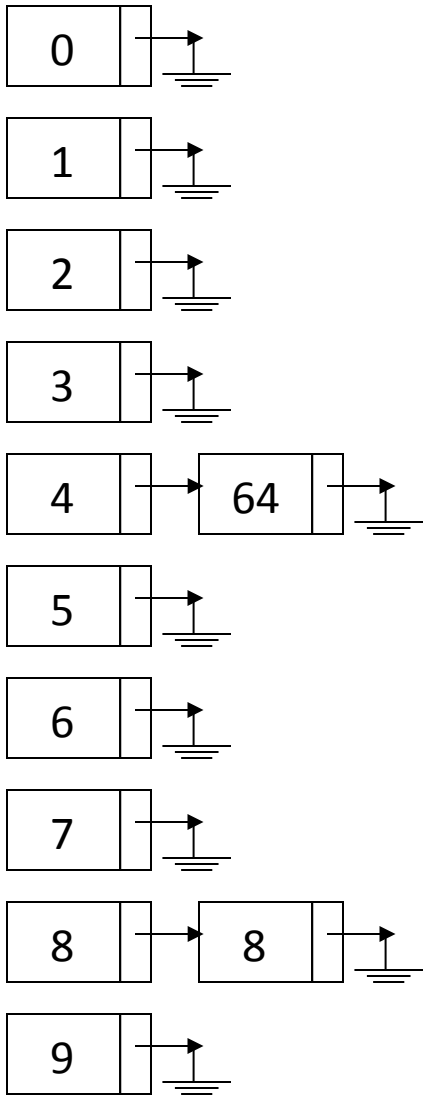
64	8	216	512	27	729	0	1	343	125



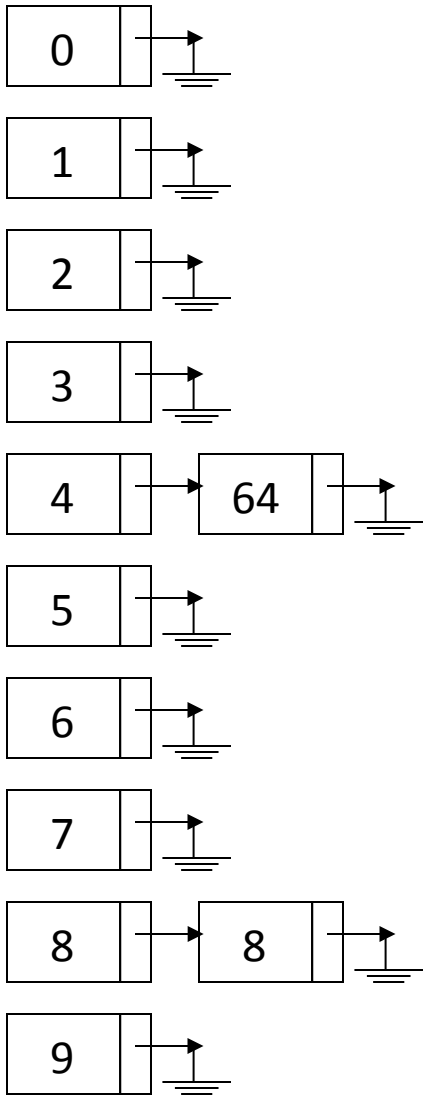
64	8	216	512	27	729	0	1	343	125



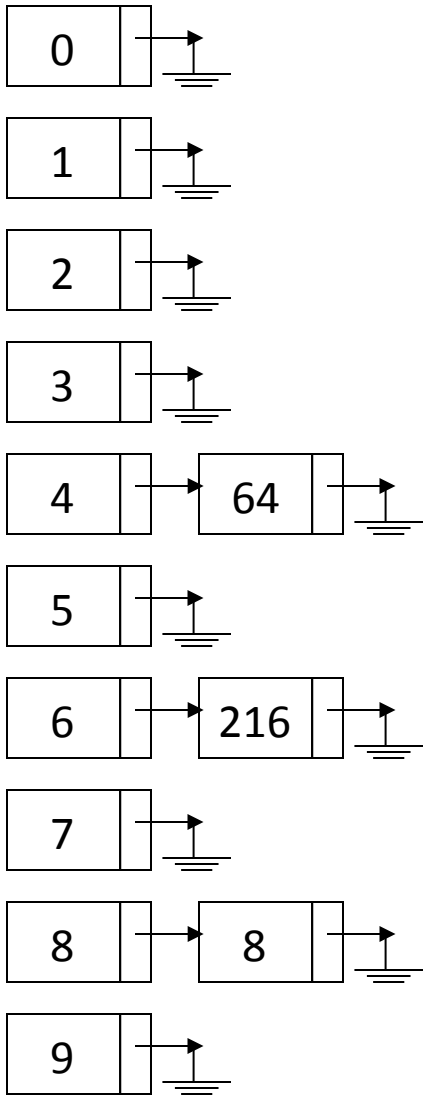
64	8	216	512	27	729	0	1	343	125



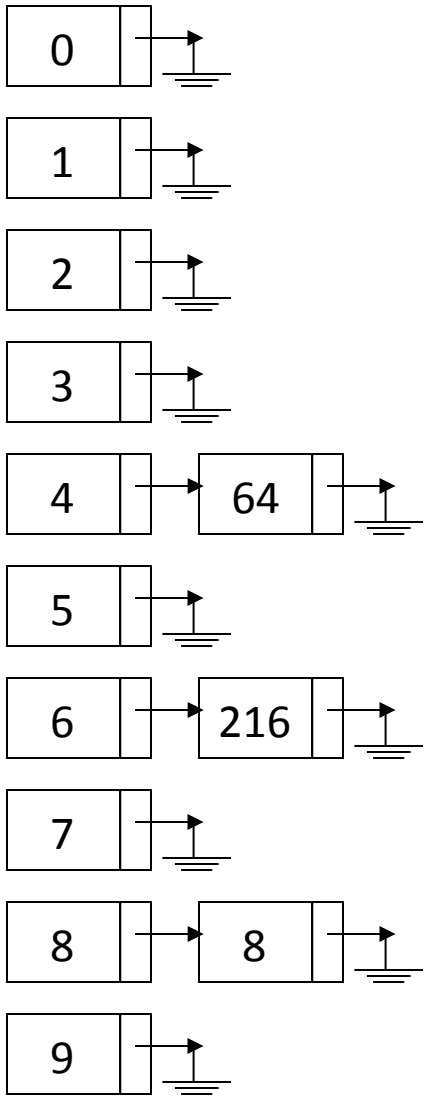
64	8	216	512	27	729	0	1	343	125



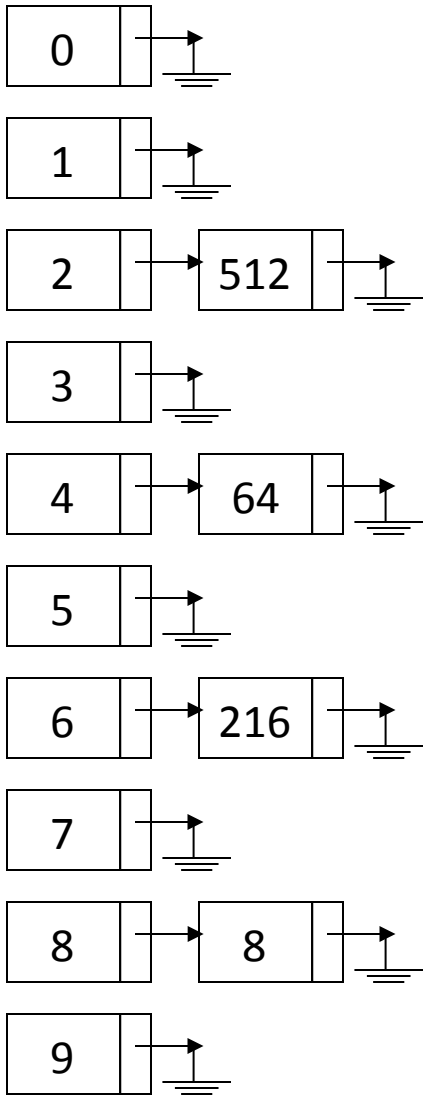
64	8	216	512	27	729	0	1	343	125



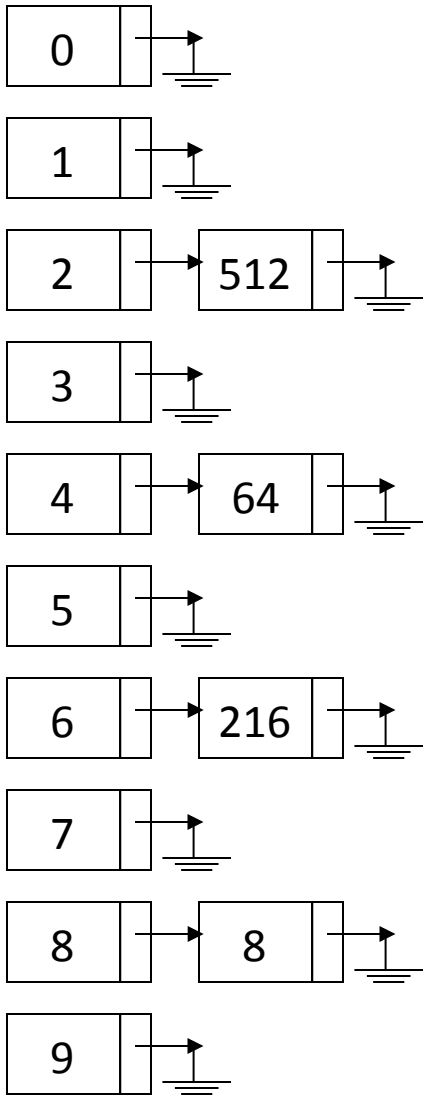
64	8	216	512	27	729	0	1	343	125



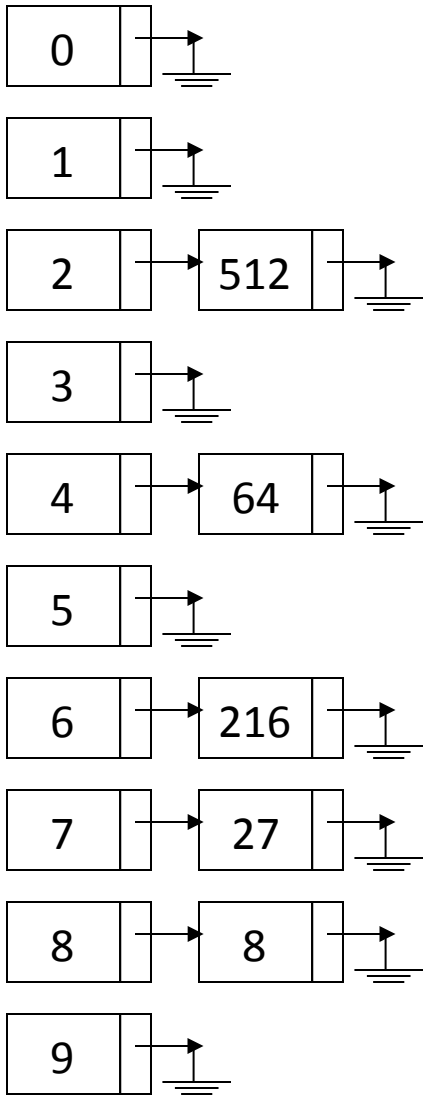
64	8	216	512	27	729	0	1	343	125



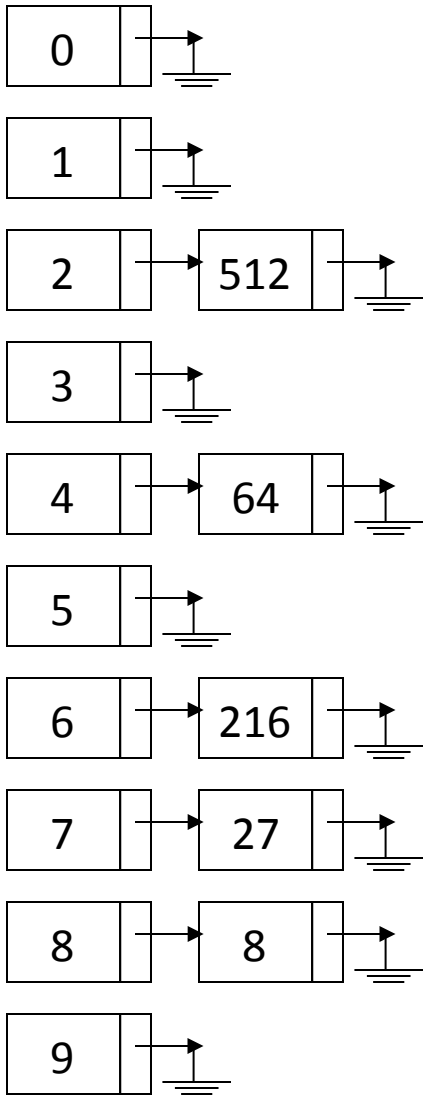
64	8	216	512	27	729	0	1	343	125



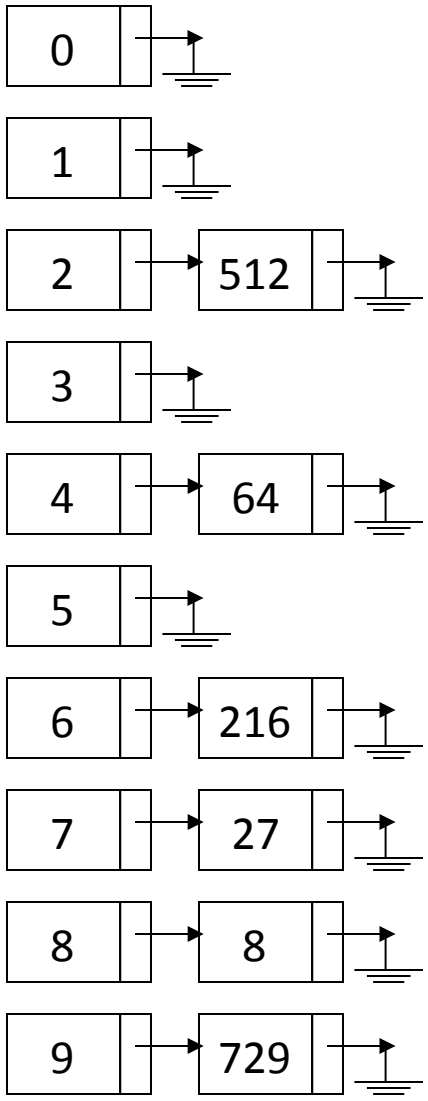
64	8	216	512	27	729	0	1	343	125



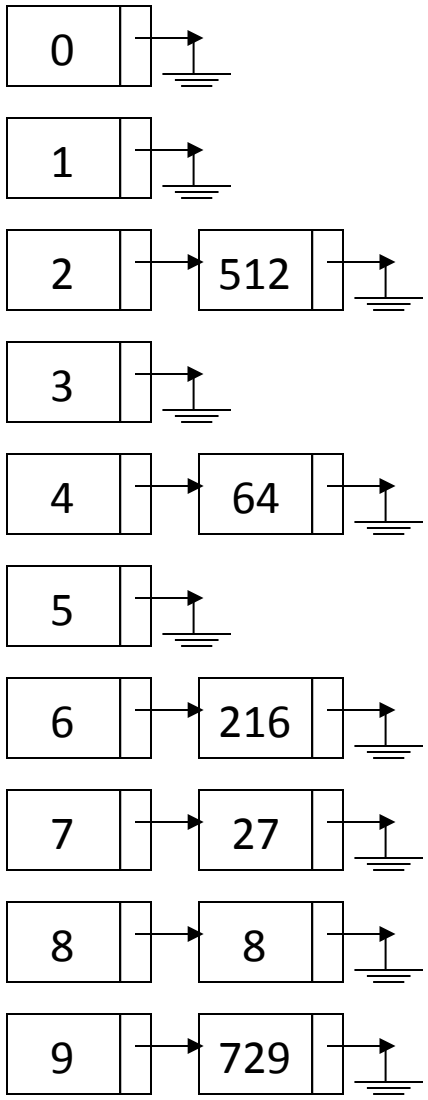
64	8	216	512	27	729	0	1	343	125



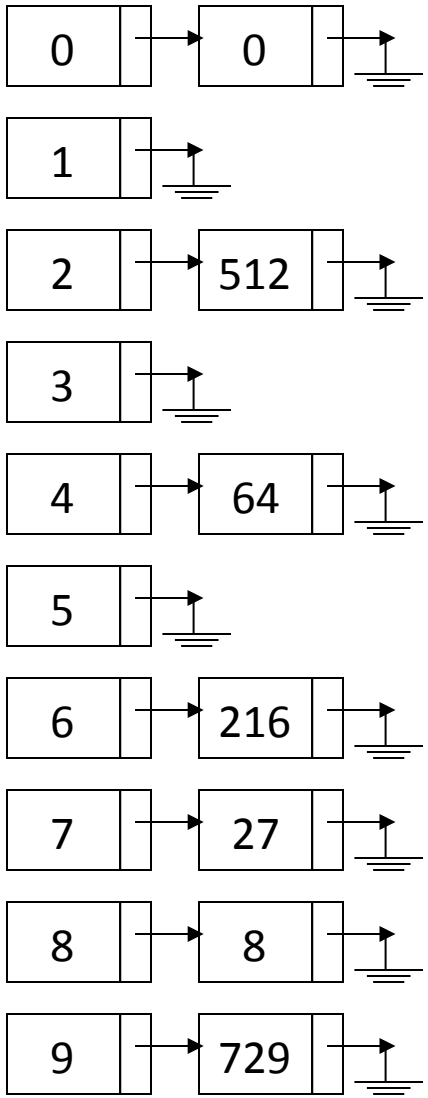
64	8	216	512	27	729	0	1	343	125



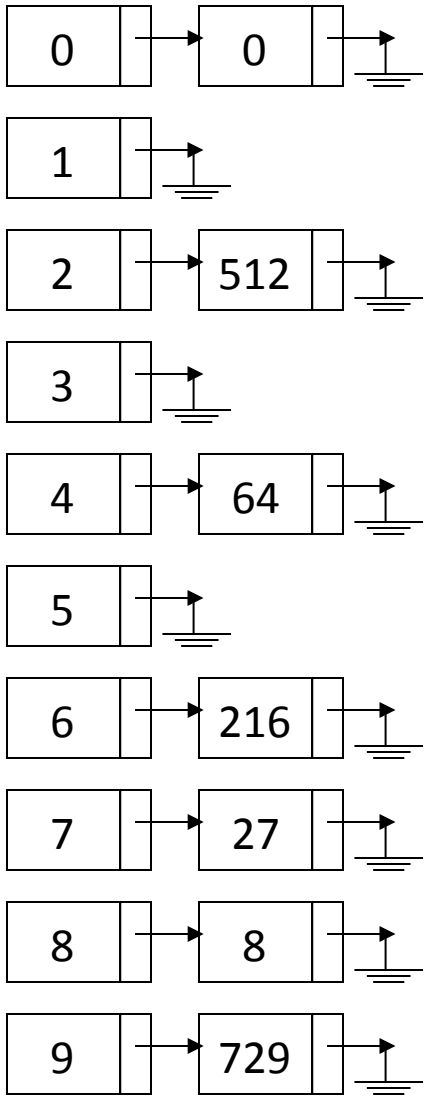
64	8	216	512	27	729	0	1	343	125



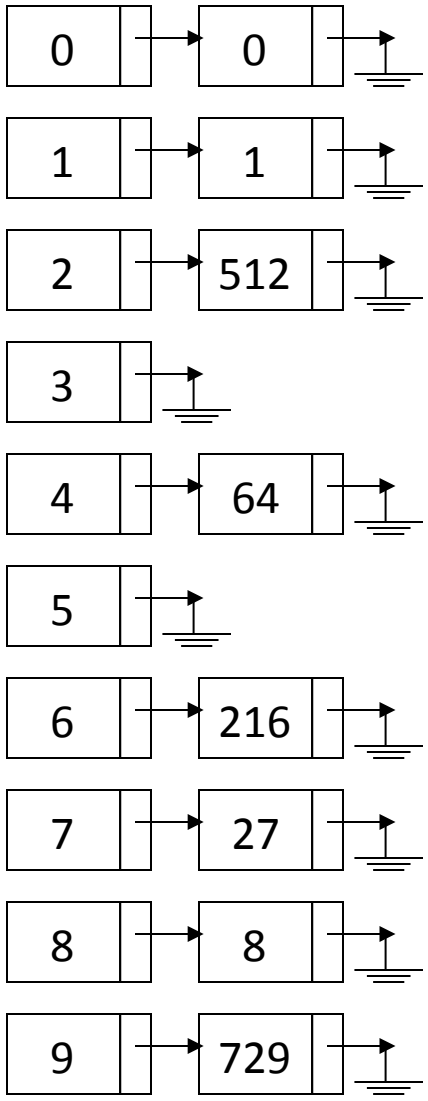
64	8	216	512	27	729	0	1	343	125



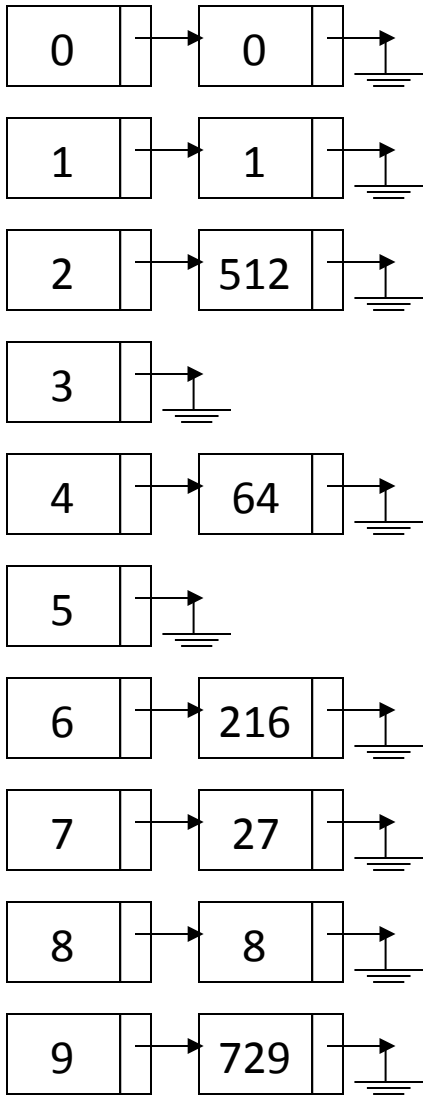
64	8	216	512	27	729	0	1	343	125



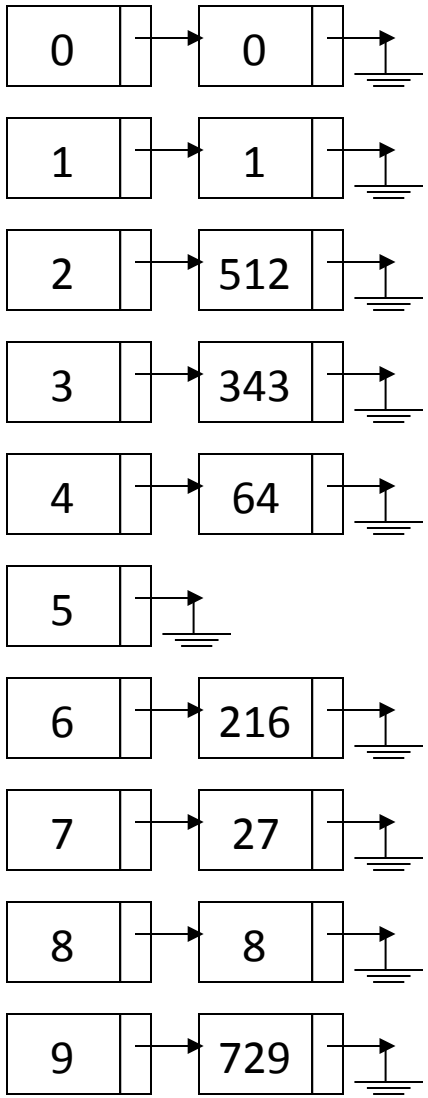
64	8	216	512	27	729	0	1	343	125



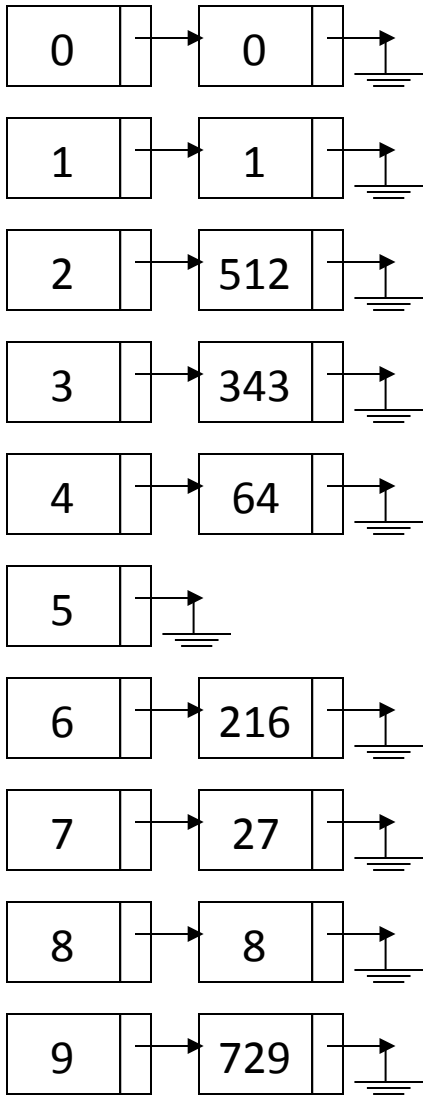
64	8	216	512	27	729	0	1	343	125



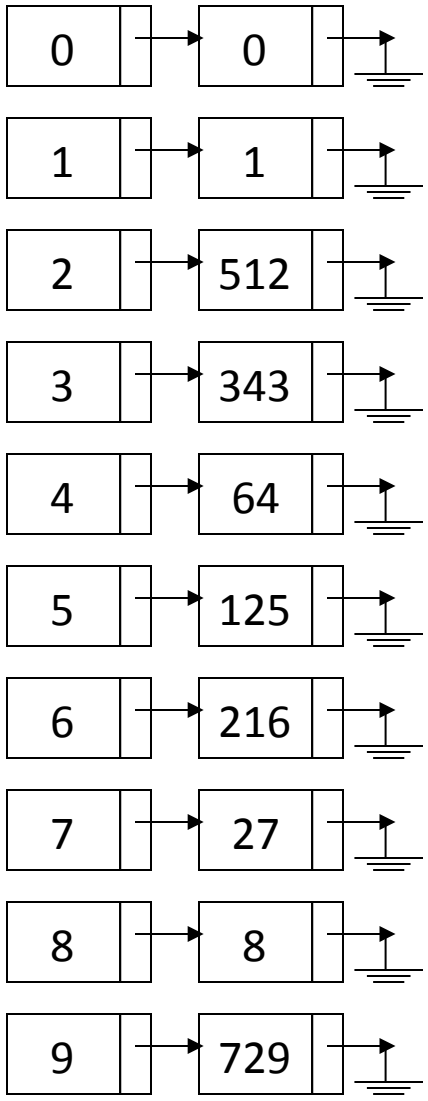
64	8	216	512	27	729	0	1	343	125



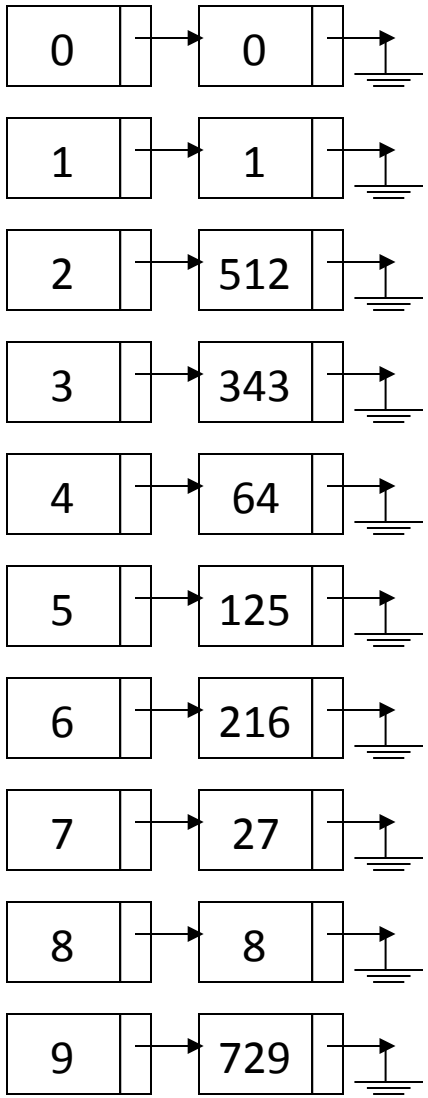
64	8	216	512	27	729	0	1	343	125

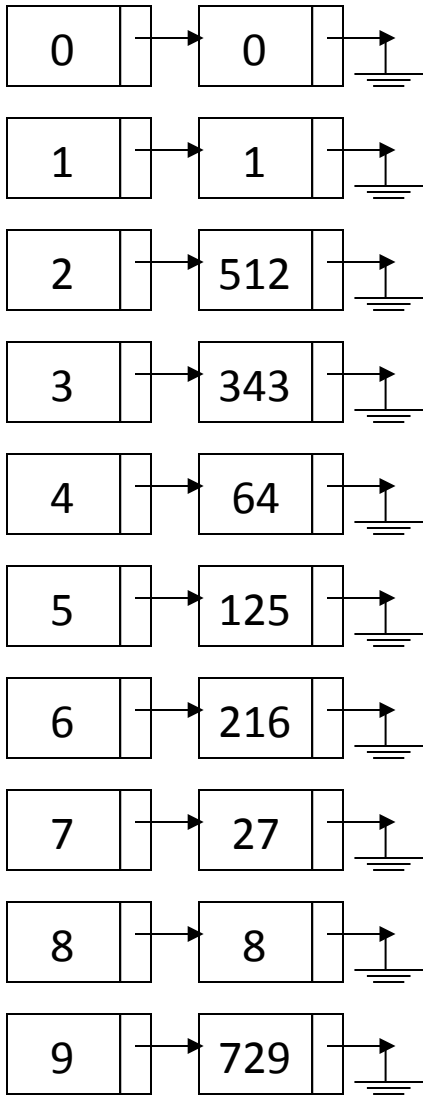


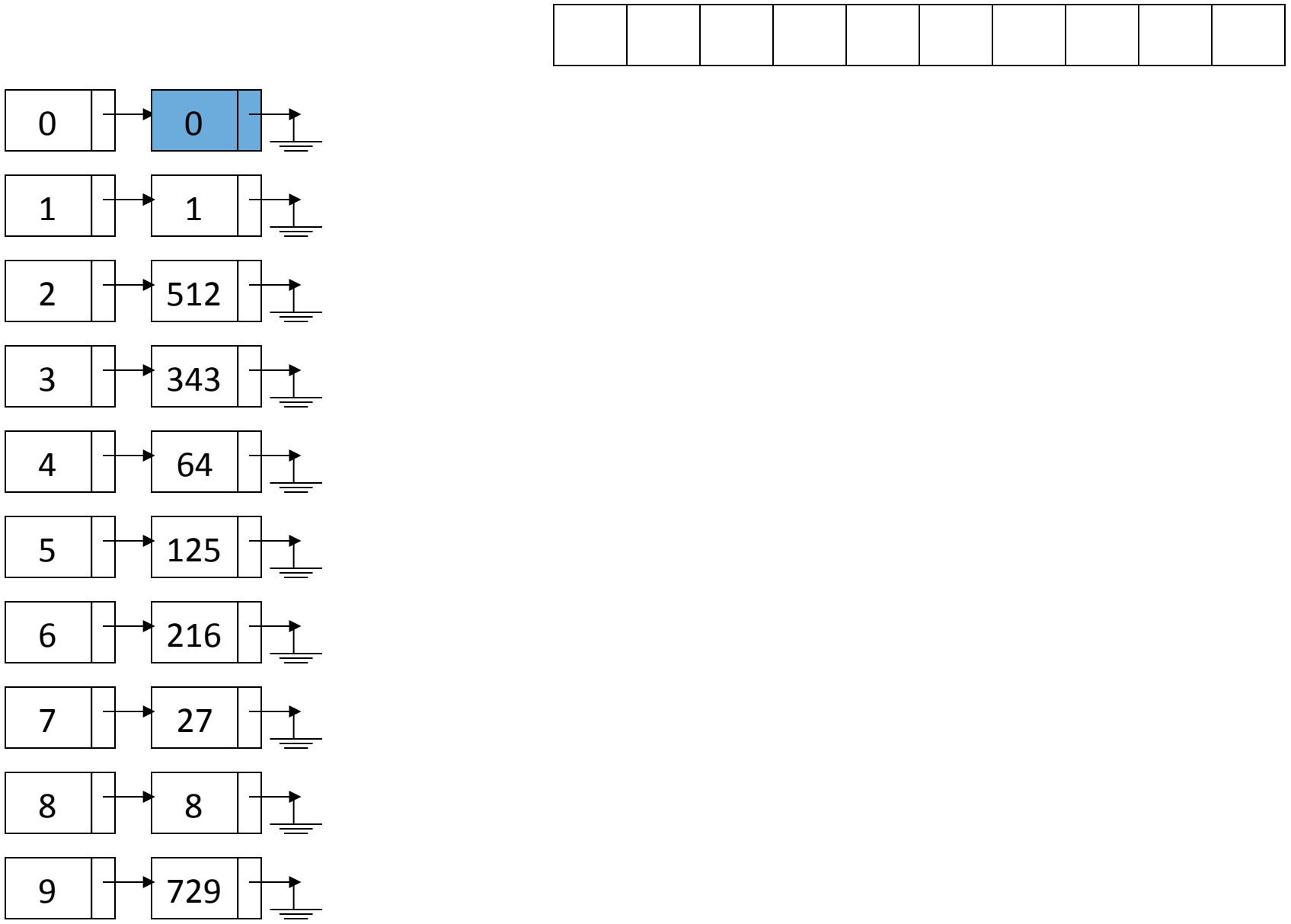
64	8	216	512	27	729	0	1	343	125

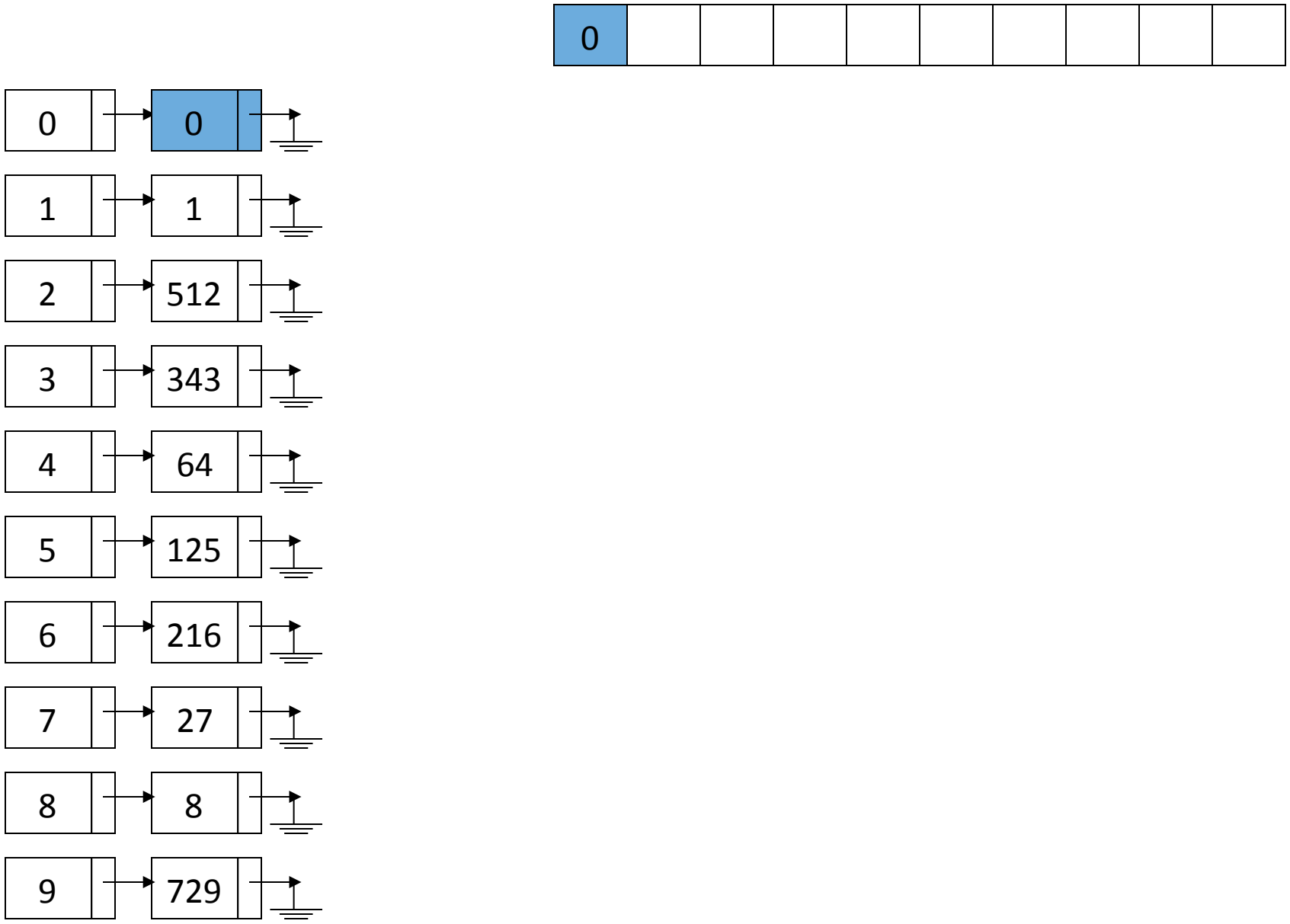


64	8	216	512	27	729	0	1	343	125

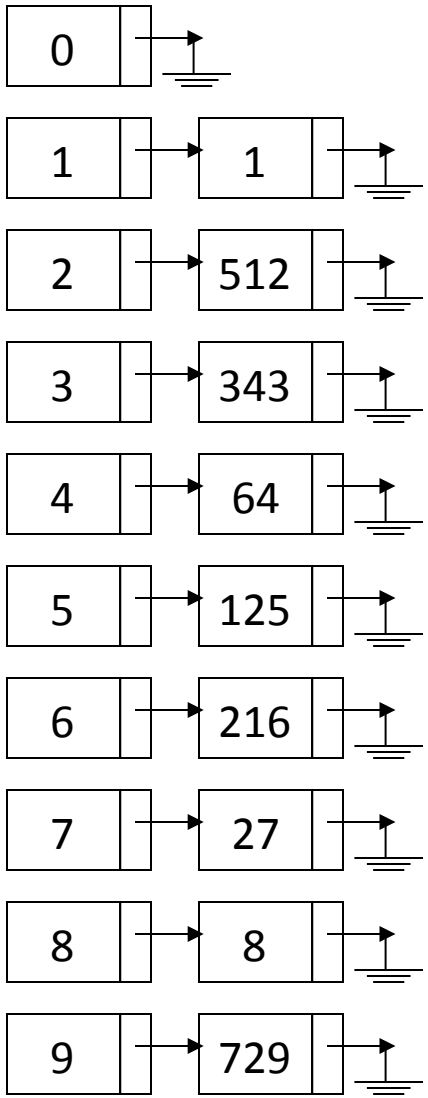


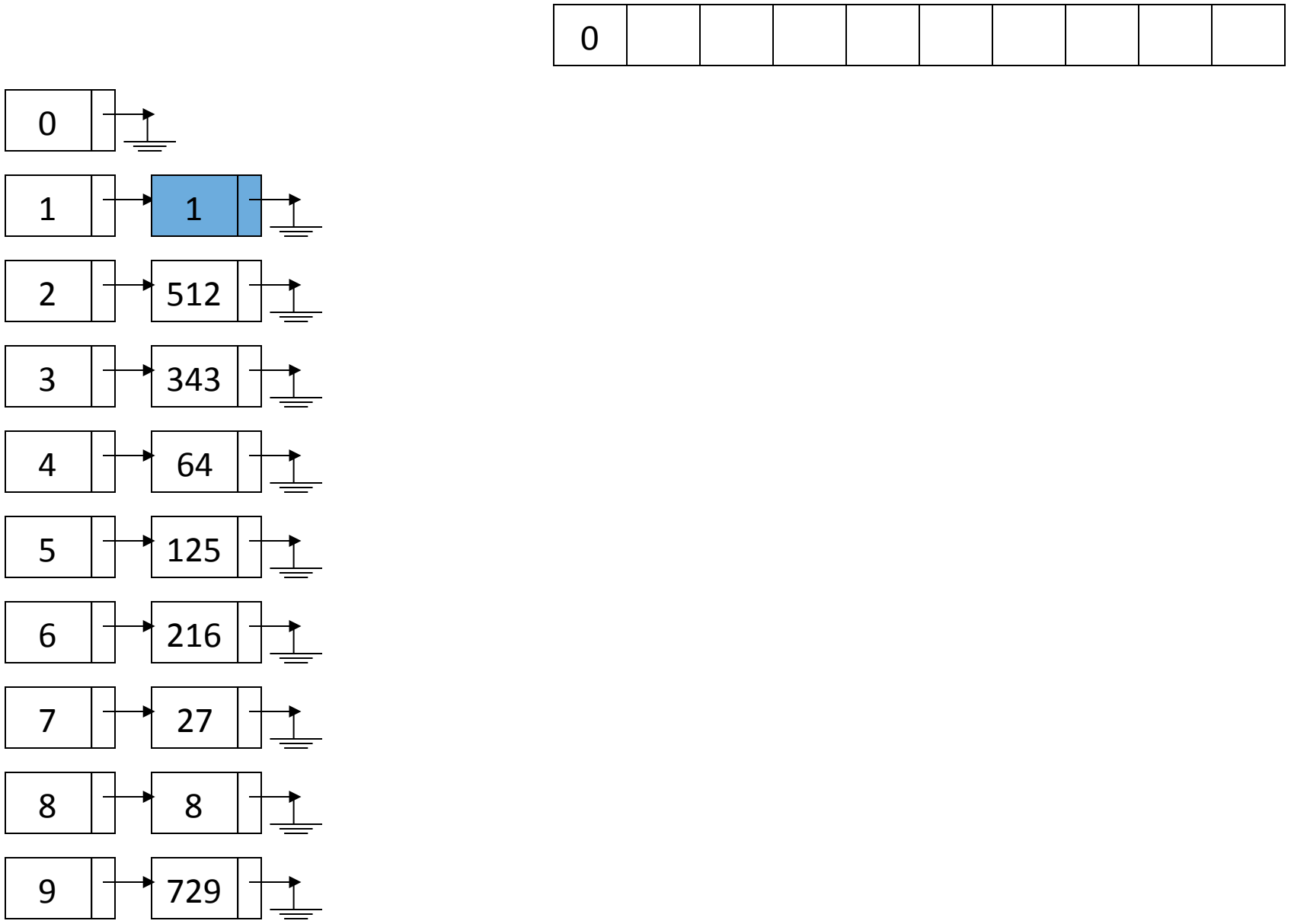

0
0
1
1
2
512
3
343
4
64
5
125
6
216
7
27
8
8
9
729


0
0
1
1
2
512
3
343
4
64
5
125
6
216
7
27
8
8
9
729

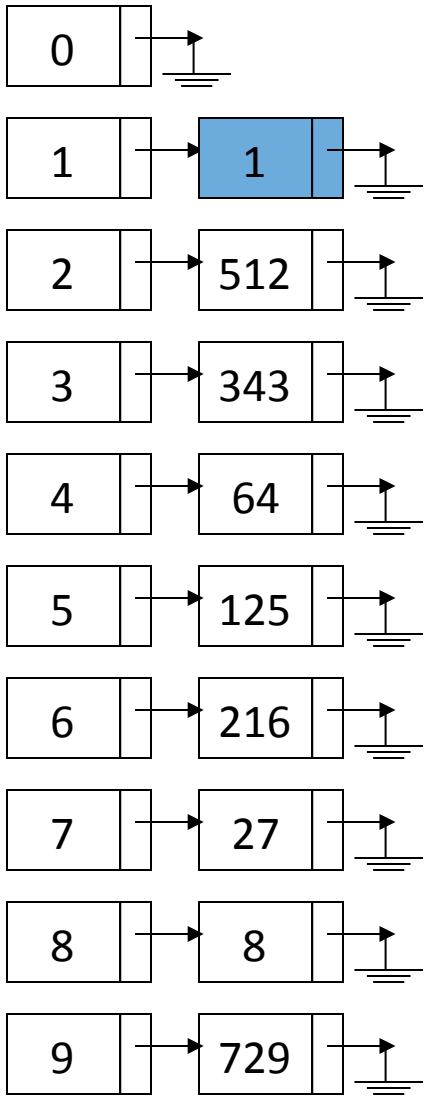

0
0
1
1
2
512
3
343
4
64
5
125
6
216
7
27
8
8
9
729
0


0

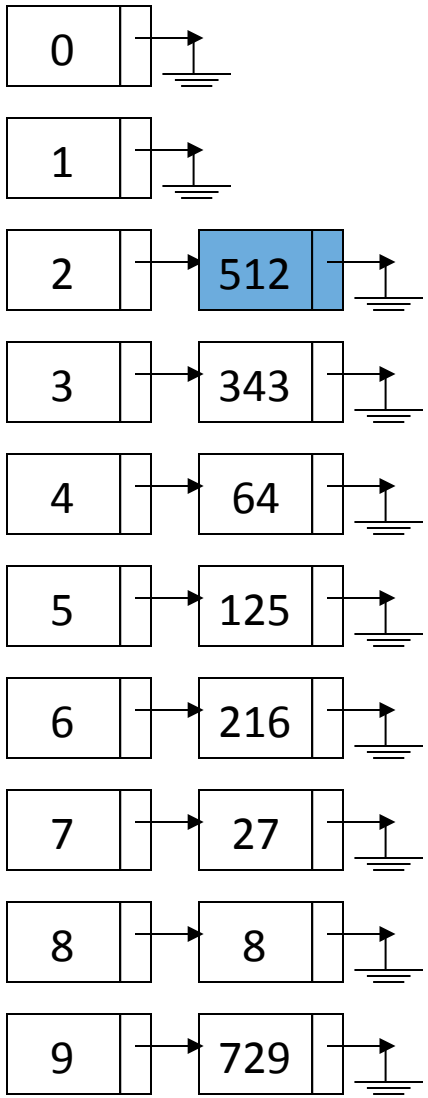

0
1
1
2
512
3
343
4
64
5
125
6
216
7
27
8
8
9
729


0
0
1
1
1
2
512
3
343
4
64
5
125
6
216
7
27
8
8
9
729

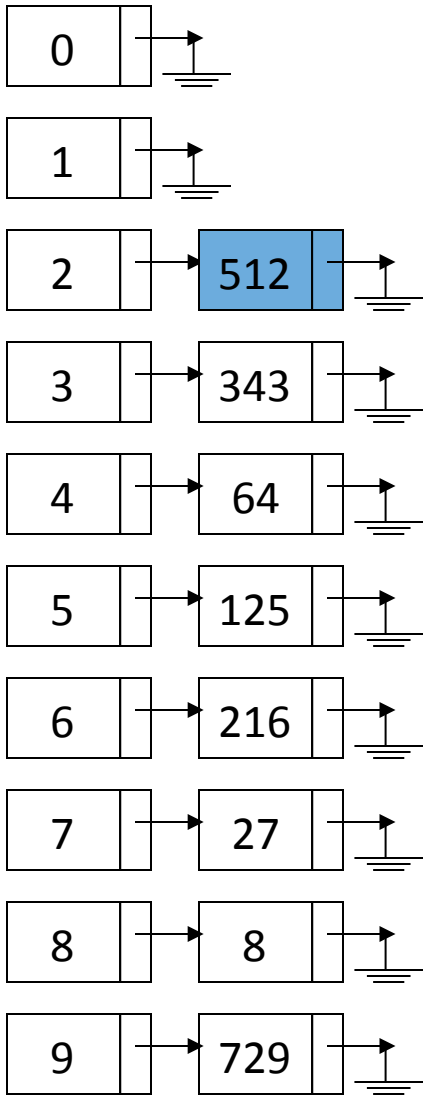

0
1


0
1
1
2
512
3
343
4
64
5
125
6
216
7
27
8
8
9
729

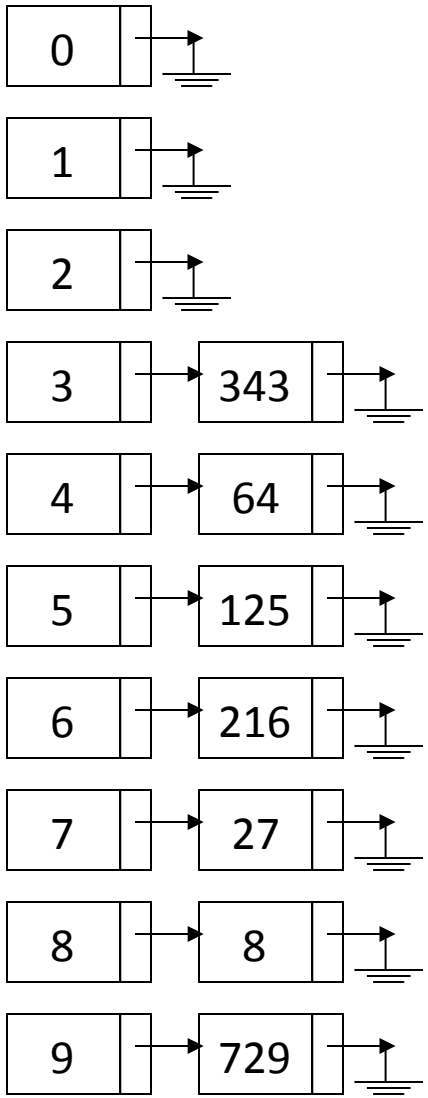
0	1								



0	1	512							

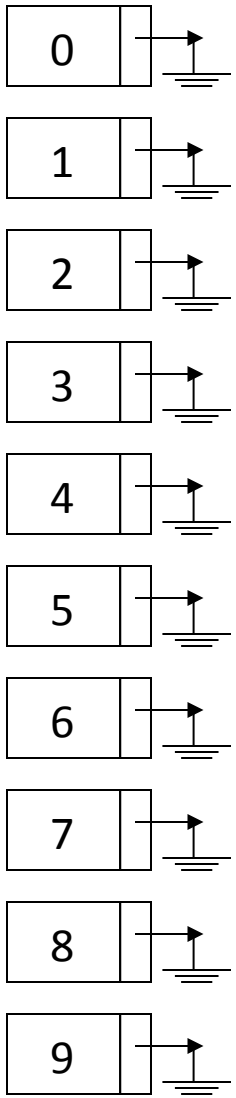


0	1	512							

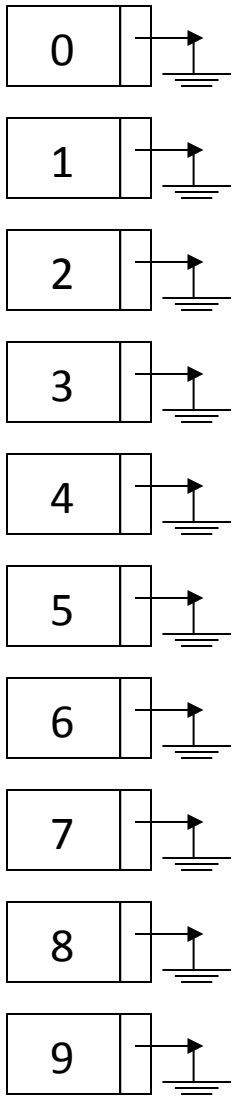


Pass 2

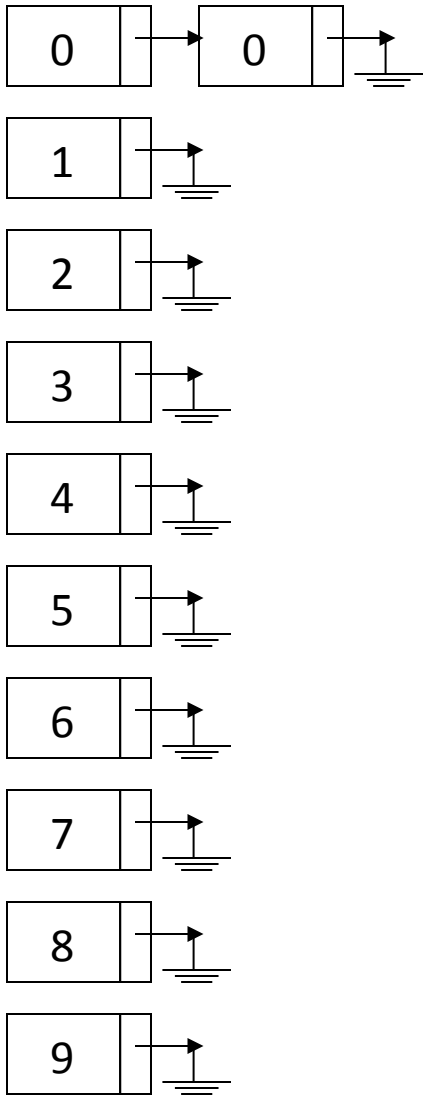
0	1	512	343	64	125	216	27	8	729



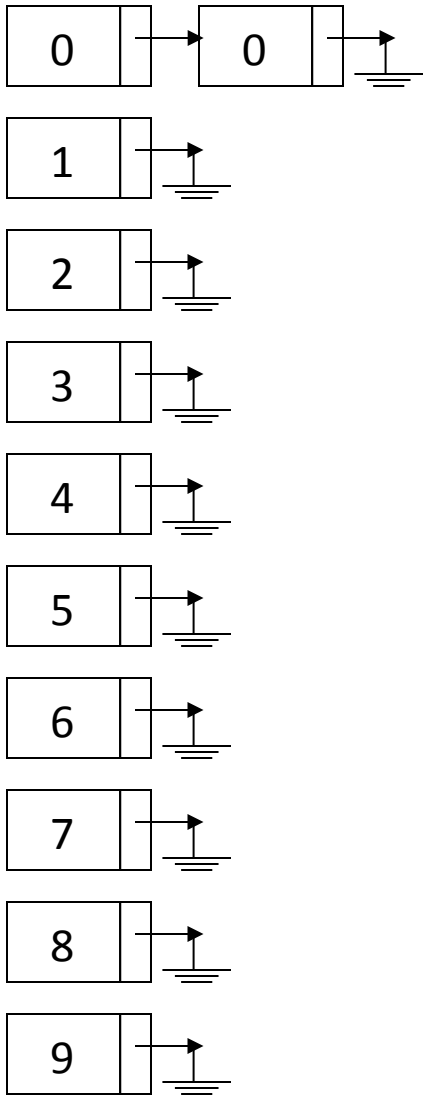
00	1	512	343	64	125	216	27	8	729



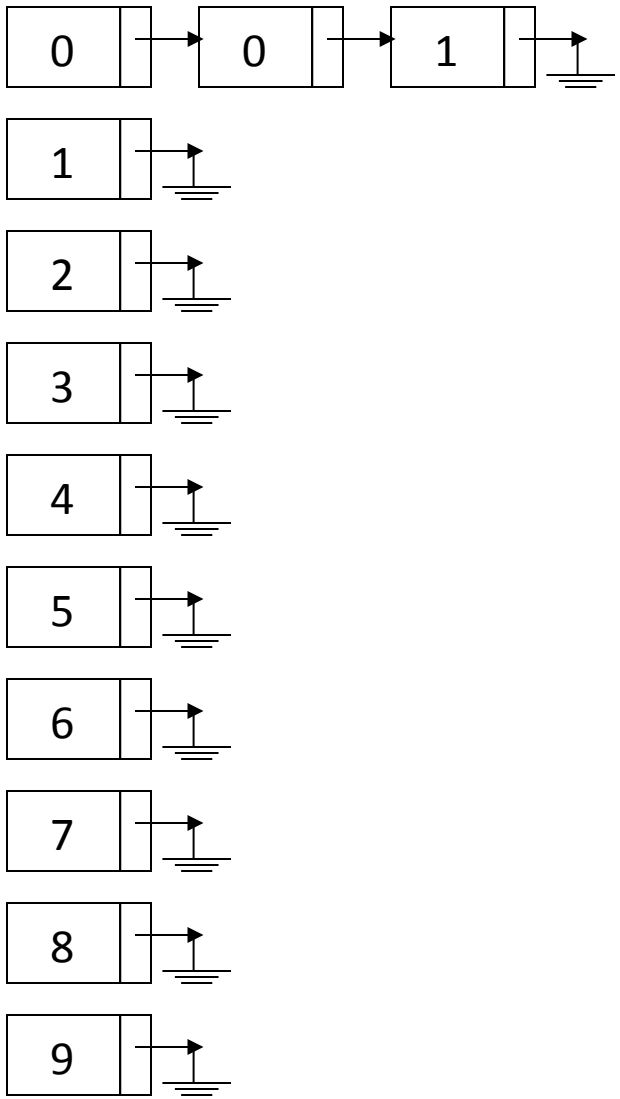
0	1	512	343	64	125	216	27	8	729



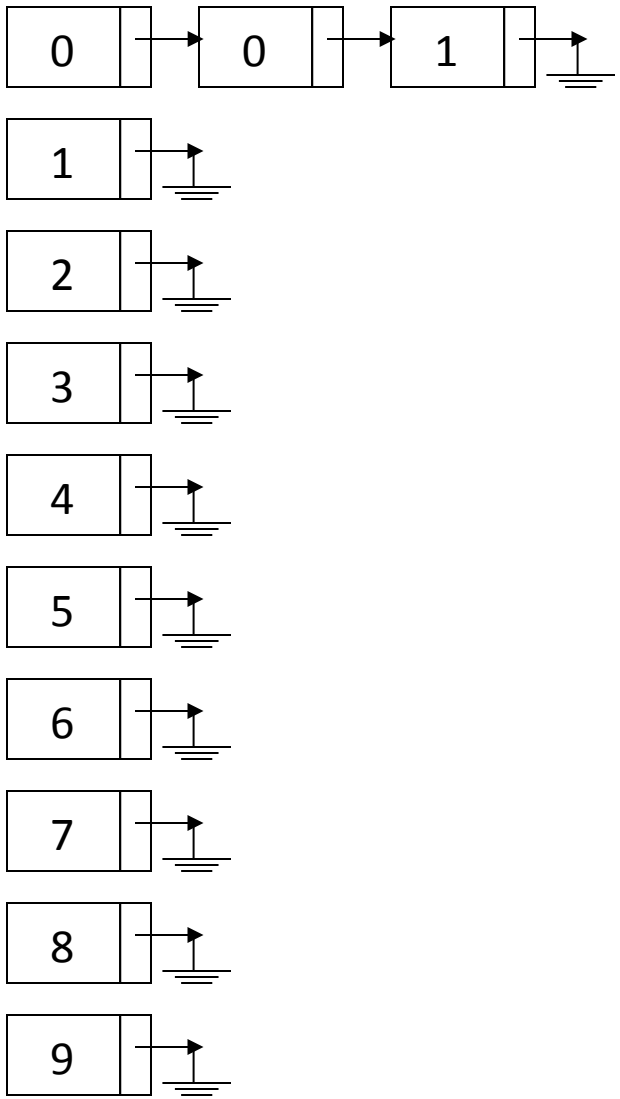
0	01	512	343	64	125	216	27	8	729



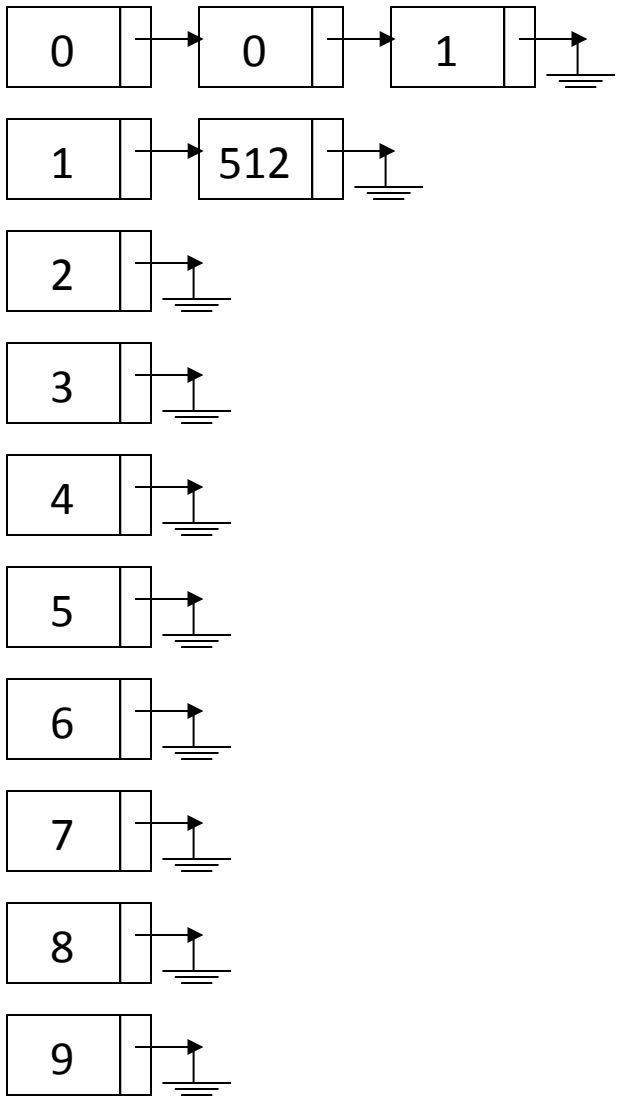
0	1	512	343	64	125	216	27	8	729



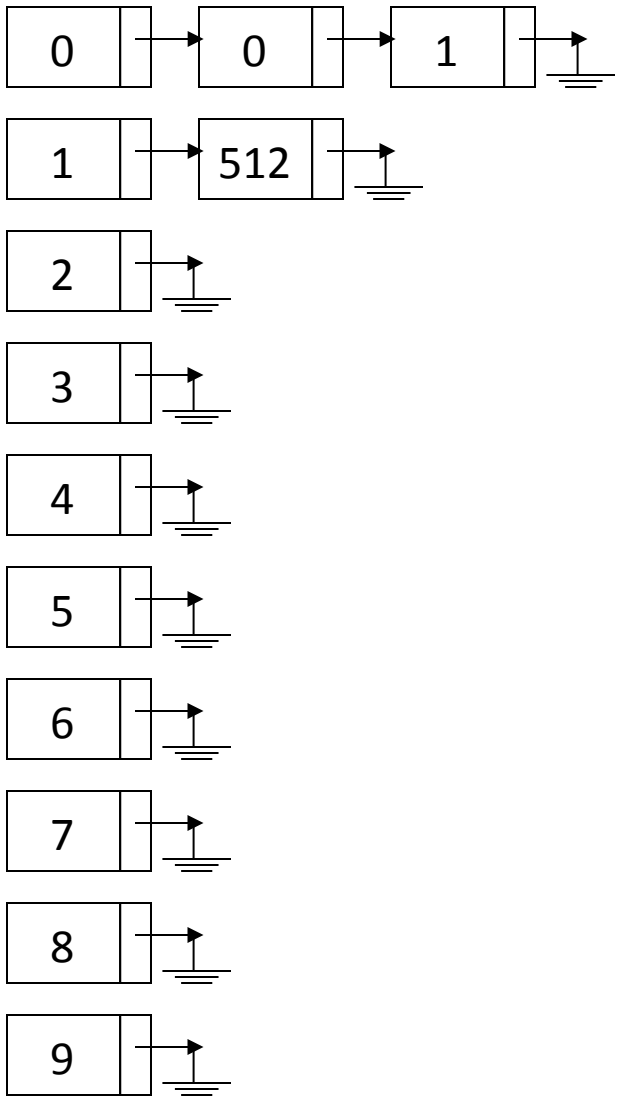
0	1	512	343	64	125	216	27	8	729



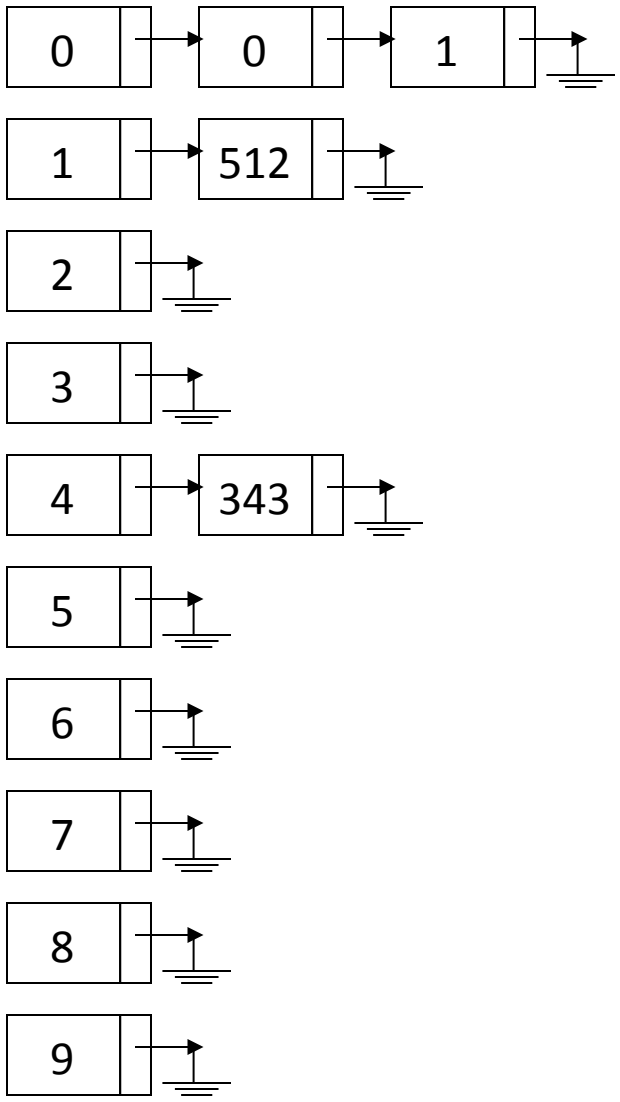
0	1	512	343	64	125	216	27	8	729



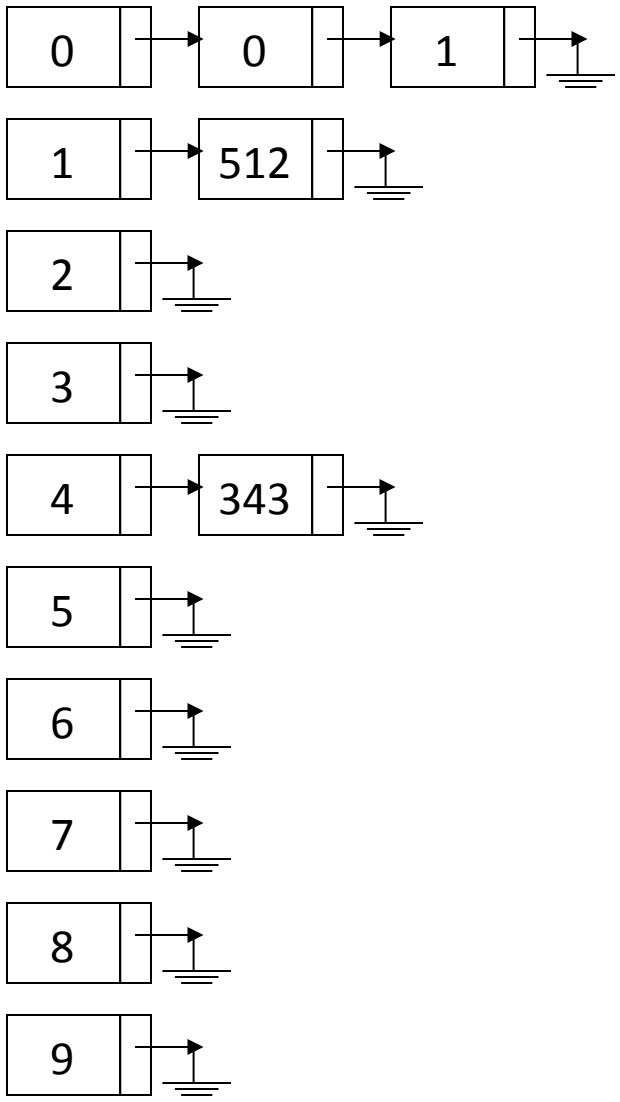
0	1	512	343	64	125	216	27	8	729



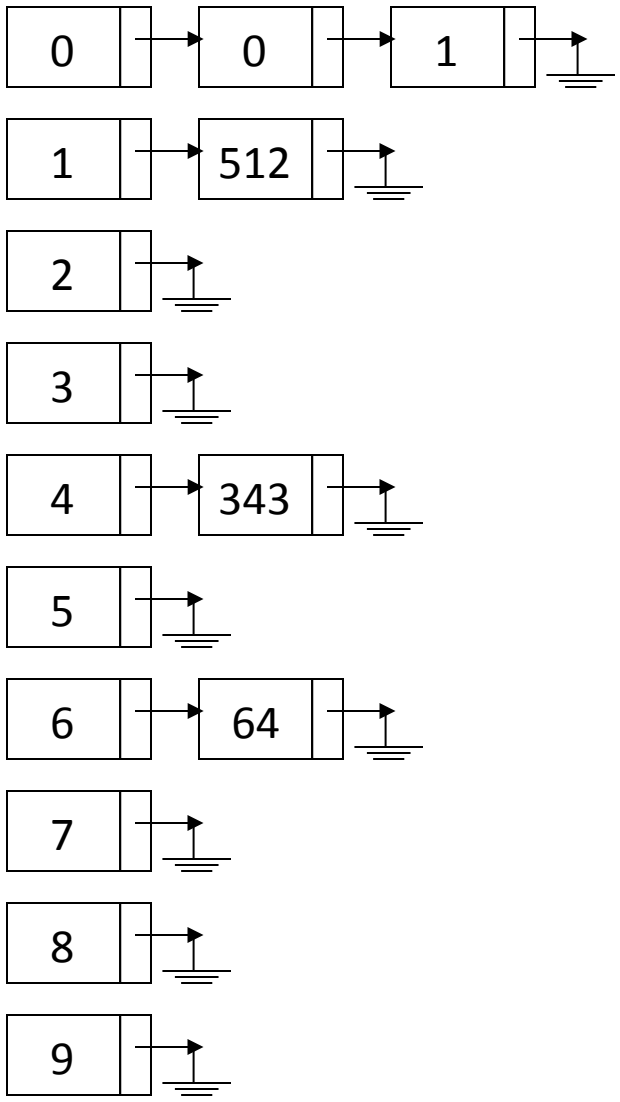
0	1	512	343	64	125	216	27	8	729



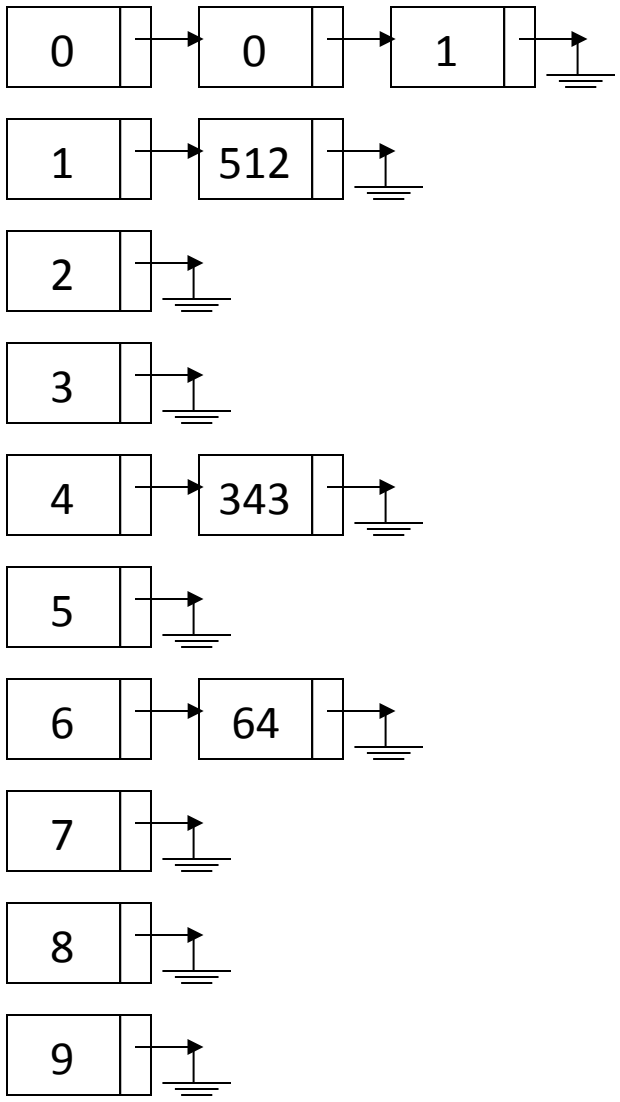
0	1	512	343	64	125	216	27	8	729



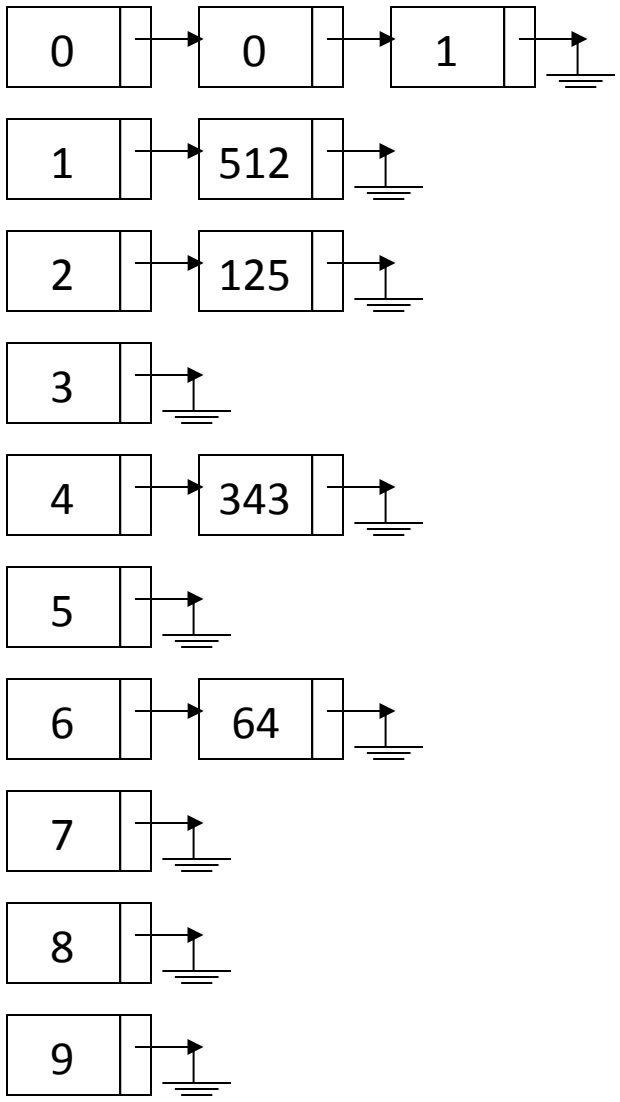
0	1	512	343	64	125	216	27	8	729



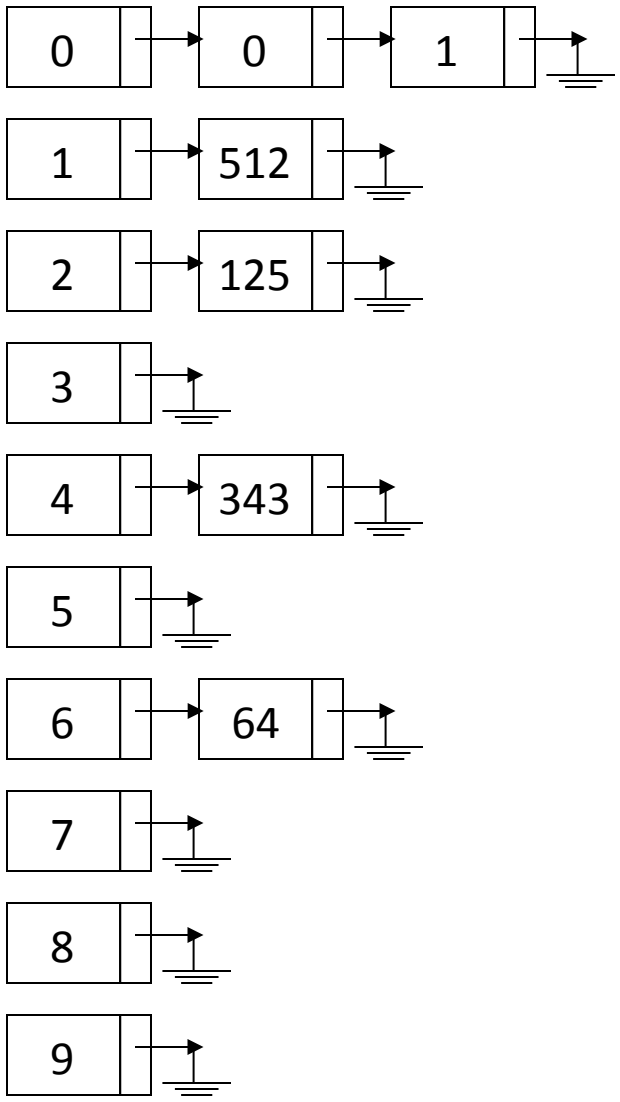
0	1	512	343	64	125	216	27	8	729



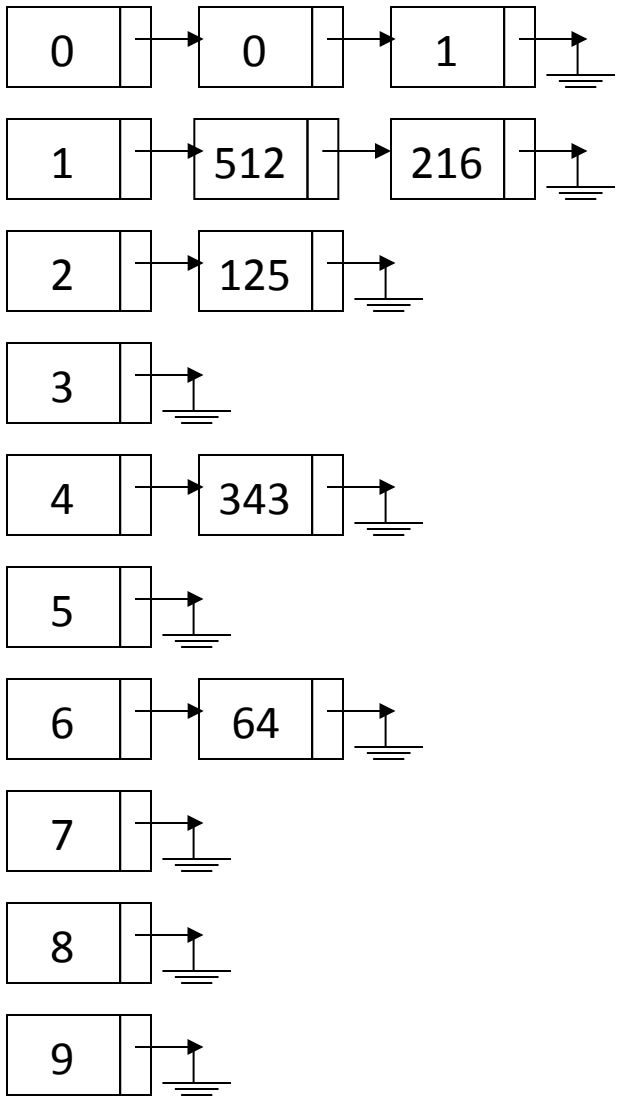
0	1	512	343	64	125	216	27	8	729



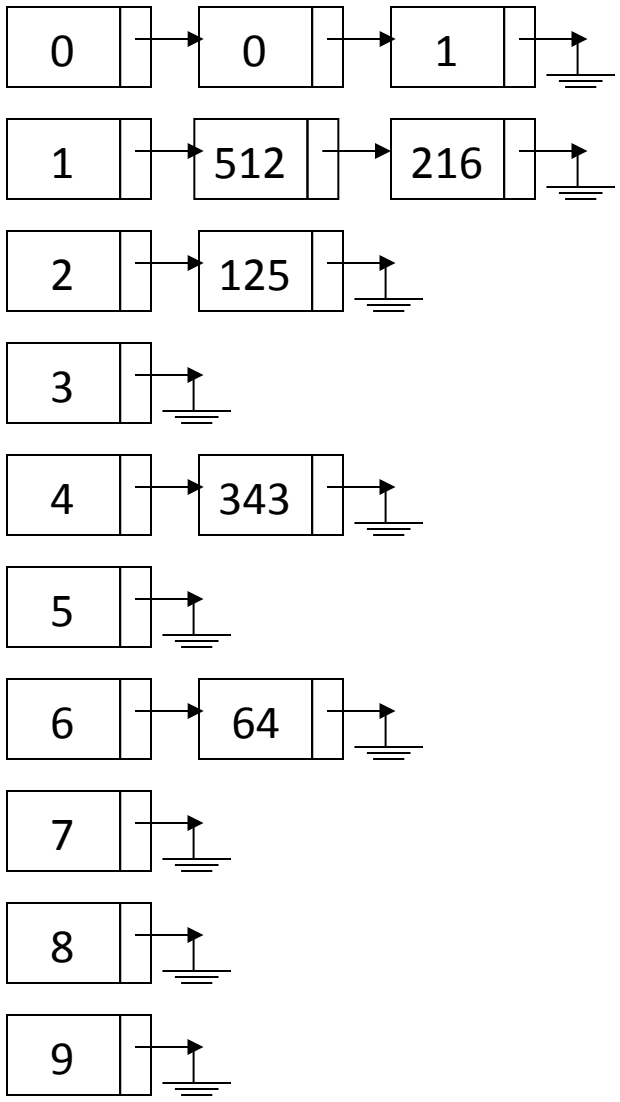
0	1	512	343	64	125	216	27	8	729



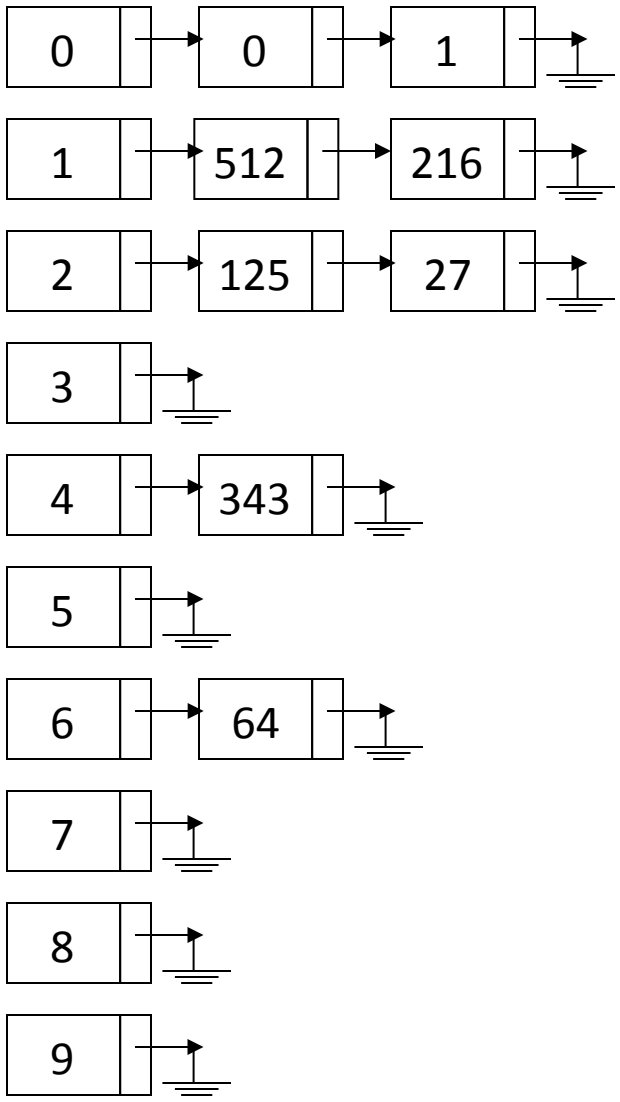
0	1	512	343	64	125	216	27	8	729



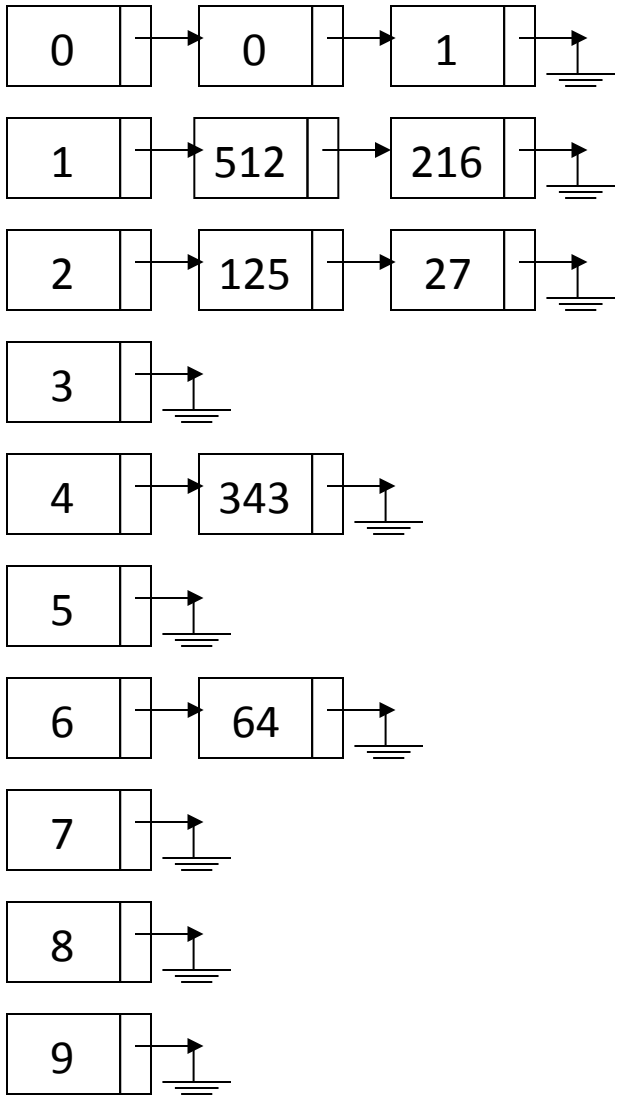
0	1	512	343	64	125	216	27	8	729

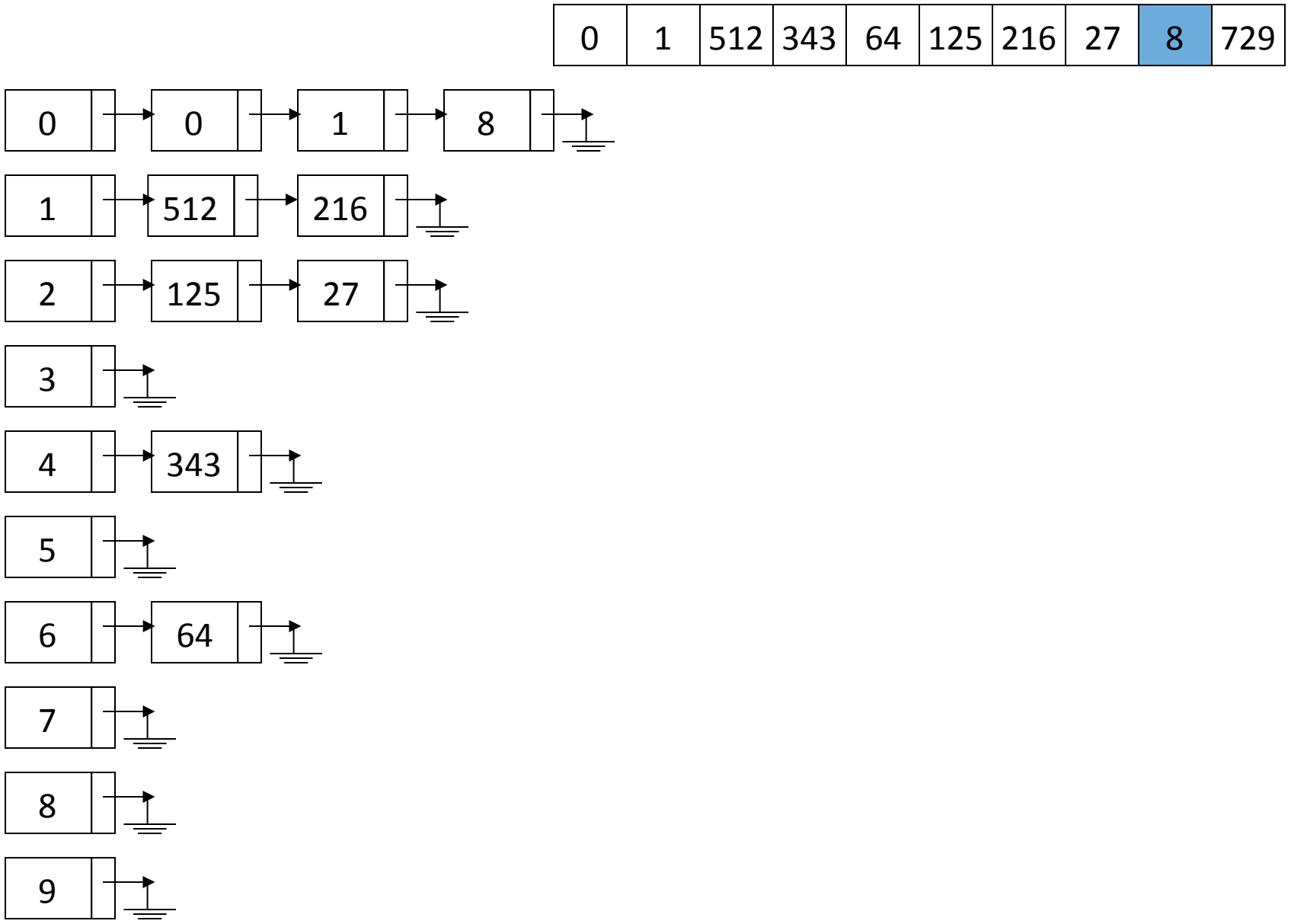


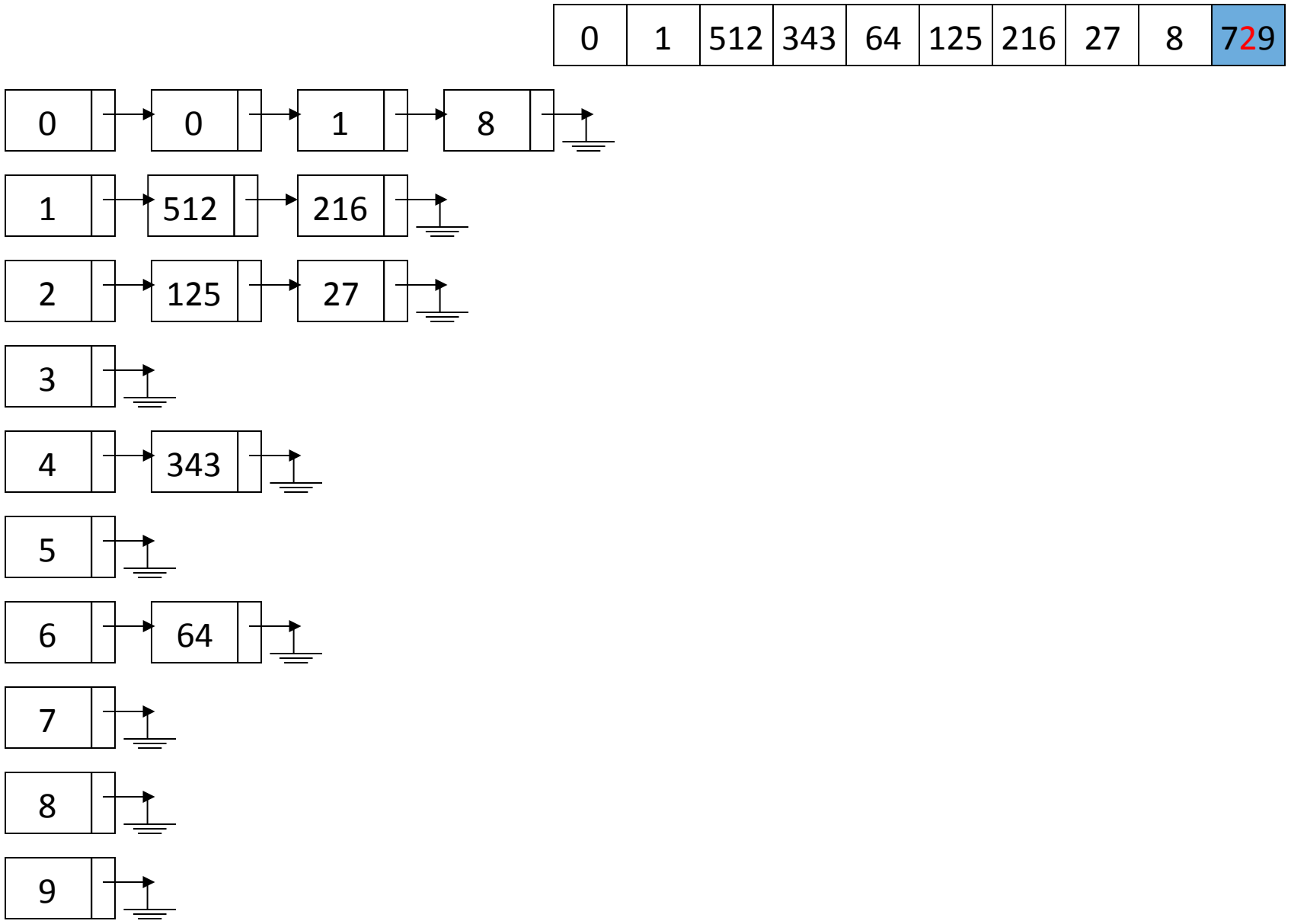
0	1	512	343	64	125	216	27	8	729

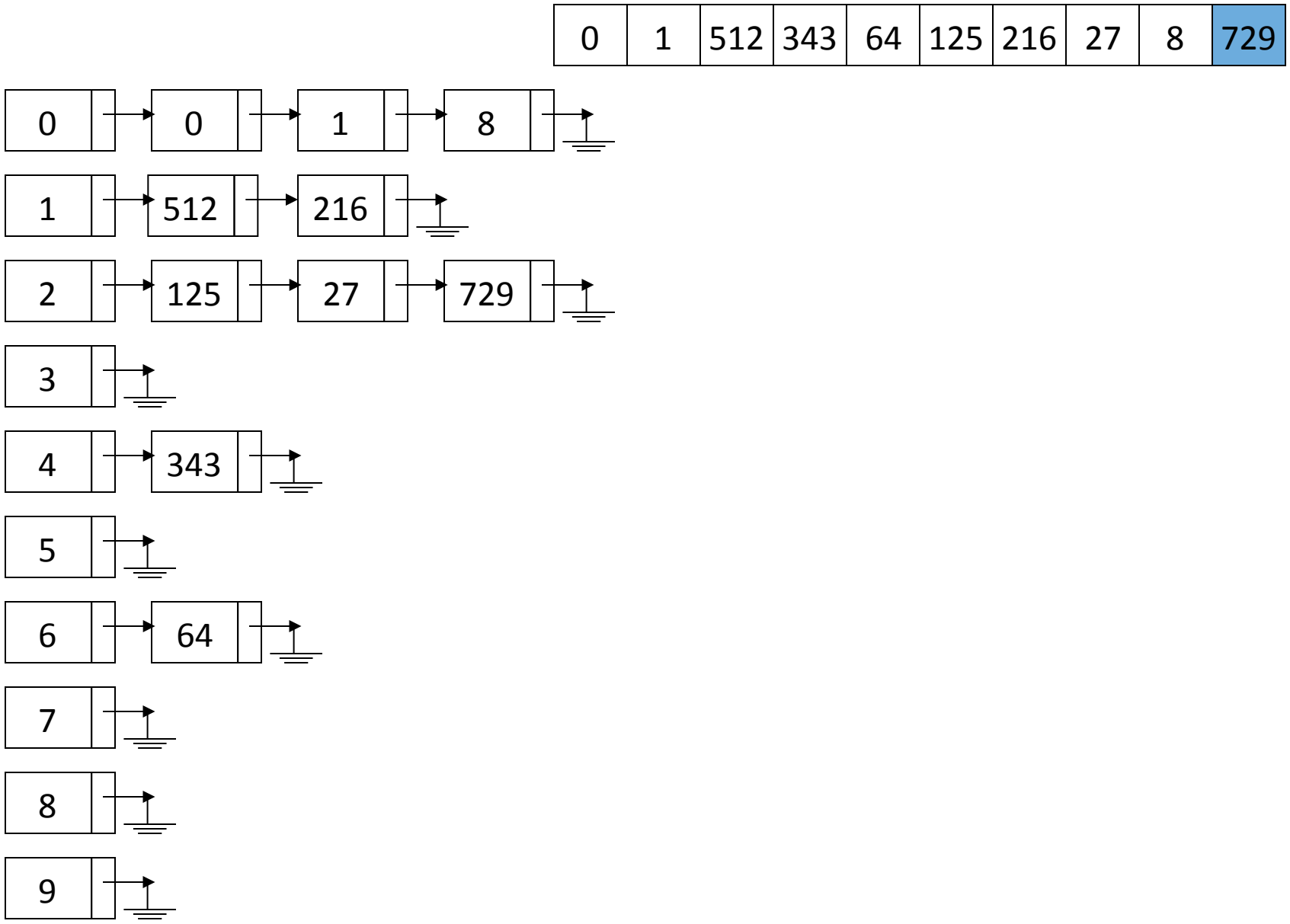


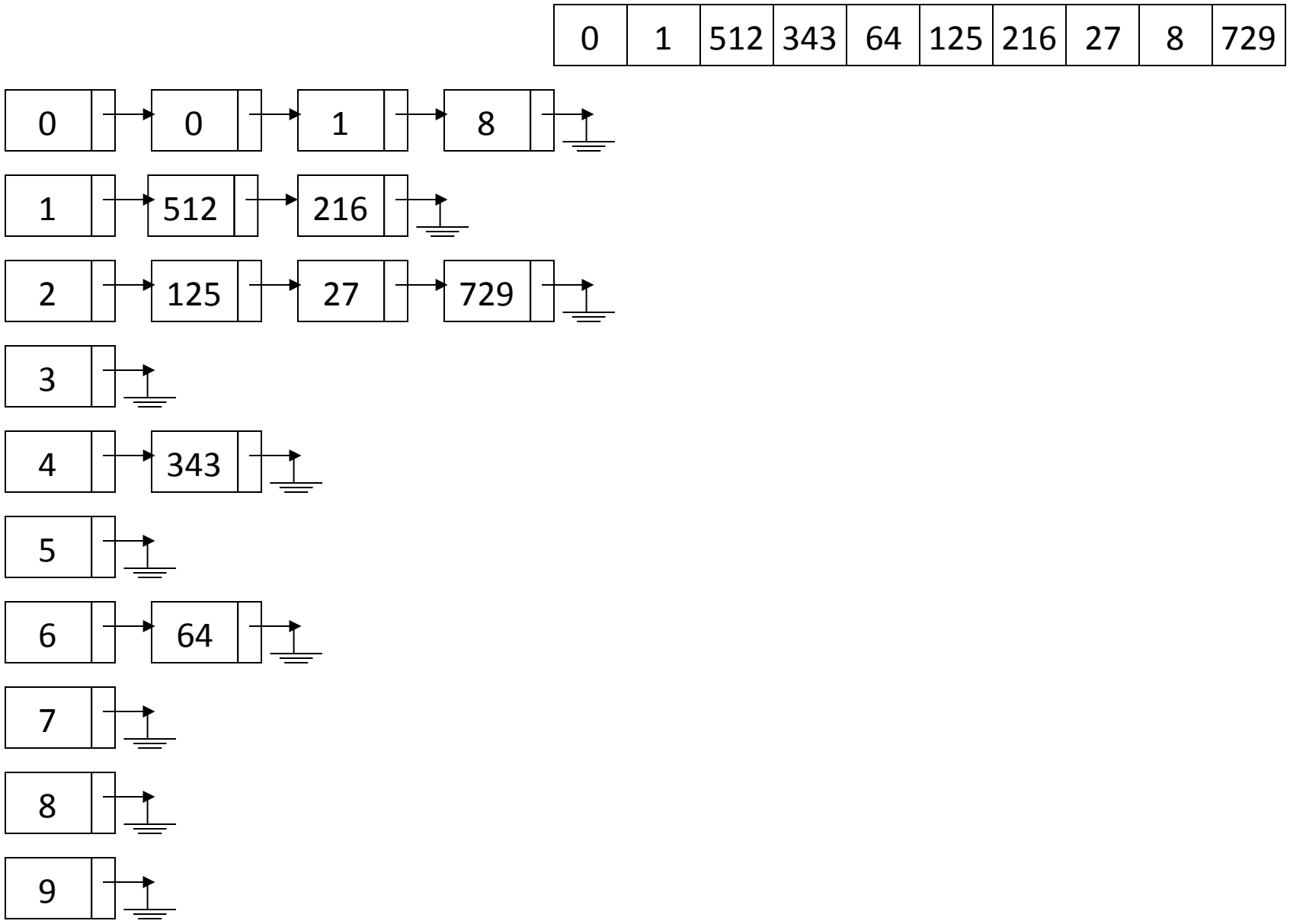
0	1	512	343	64	125	216	27	08	729

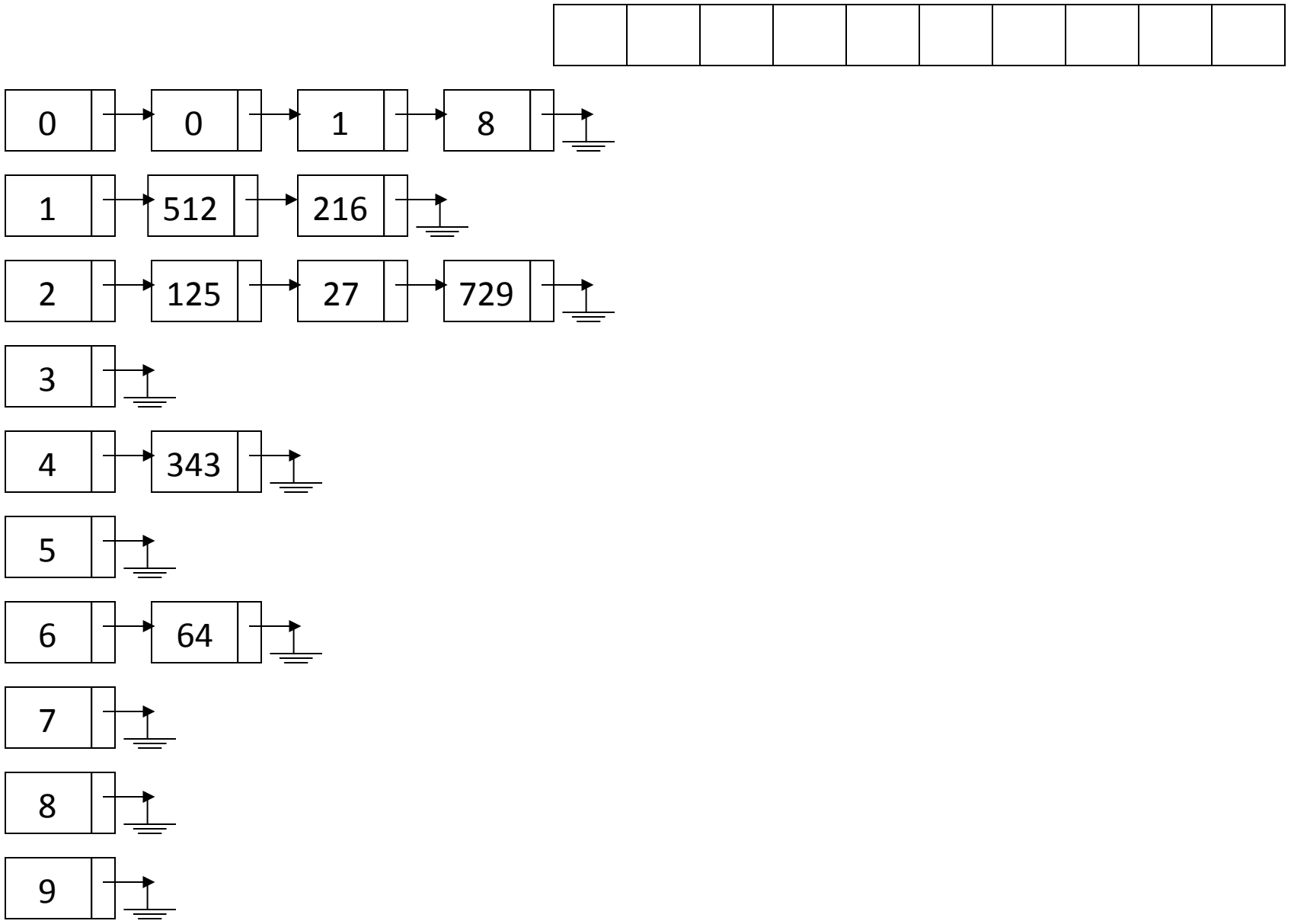


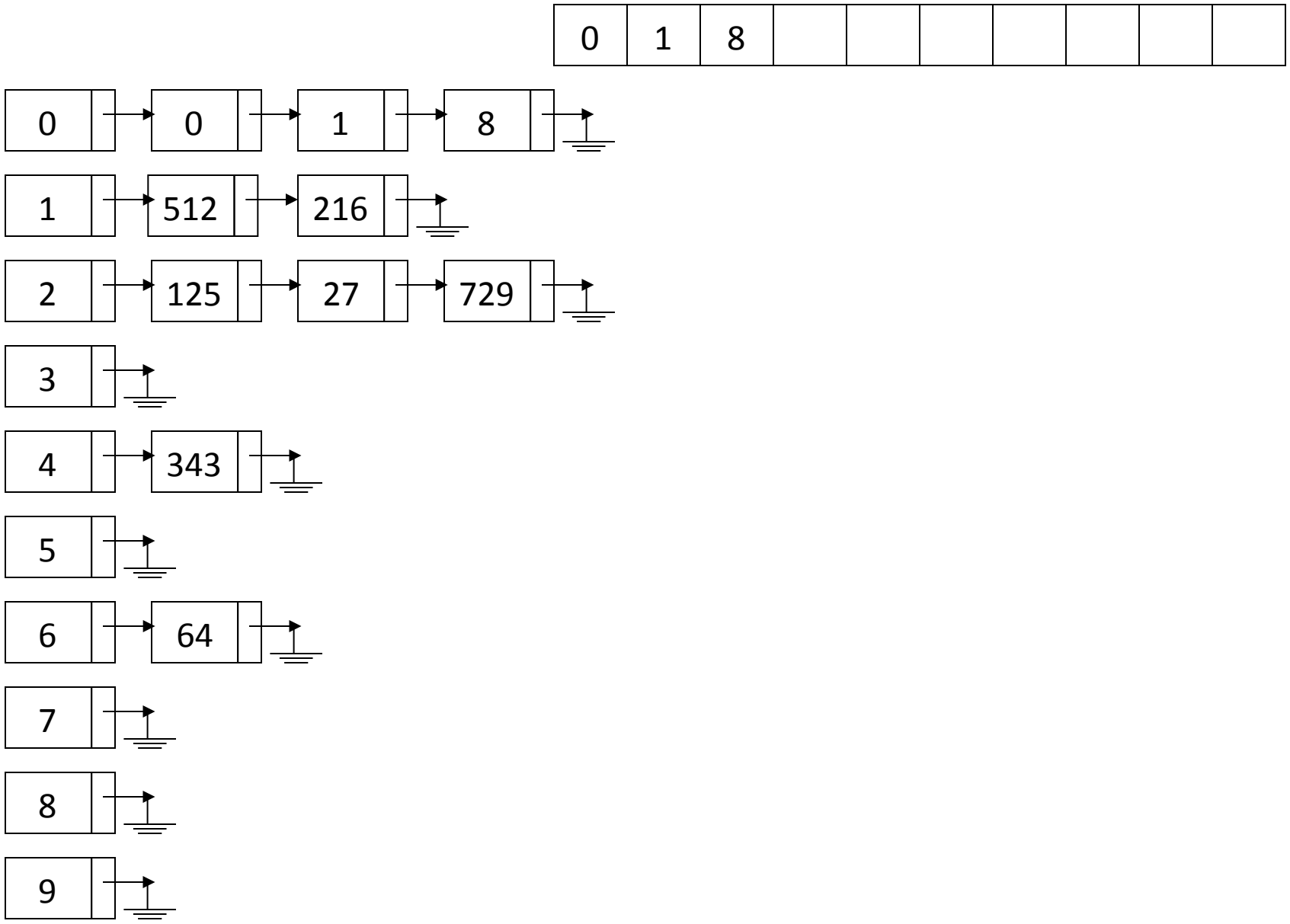

0
1
512
343
64
125
216
27
8
729
0
0
1
8
1
512
216
2
125
27
3
4
343
5
6
64
7
8
9

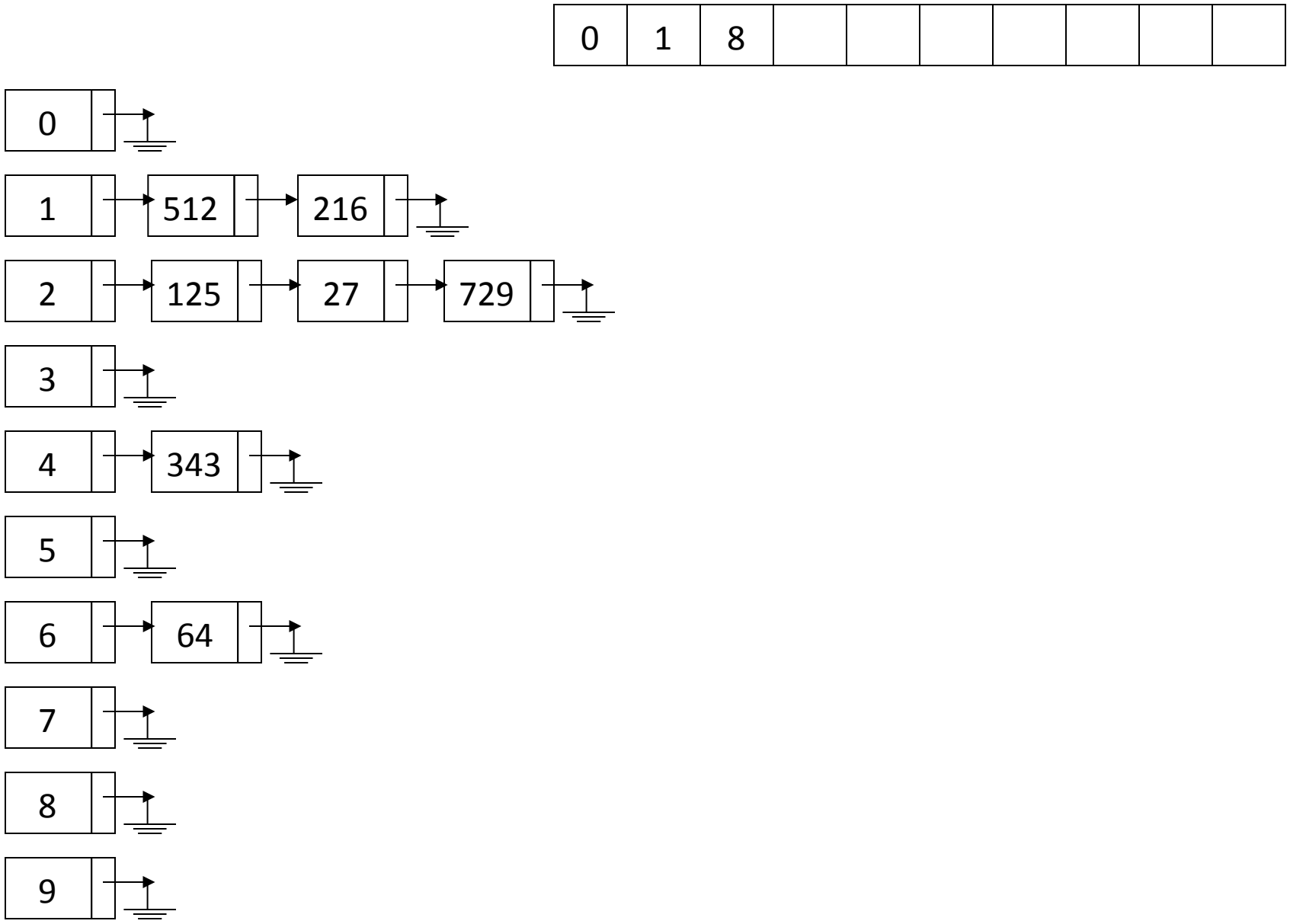

0 1 512 343 64 125 216 27 8 729
0 0 1 8
1 512 216
2 125 27
3
4 343
5
6 64
7
8
9


0
1
512
343
64
125
216
27
8
729
0
0
1
8
1
512
216
2
125
27
729
3
4
343
5
6
64
7
8
9

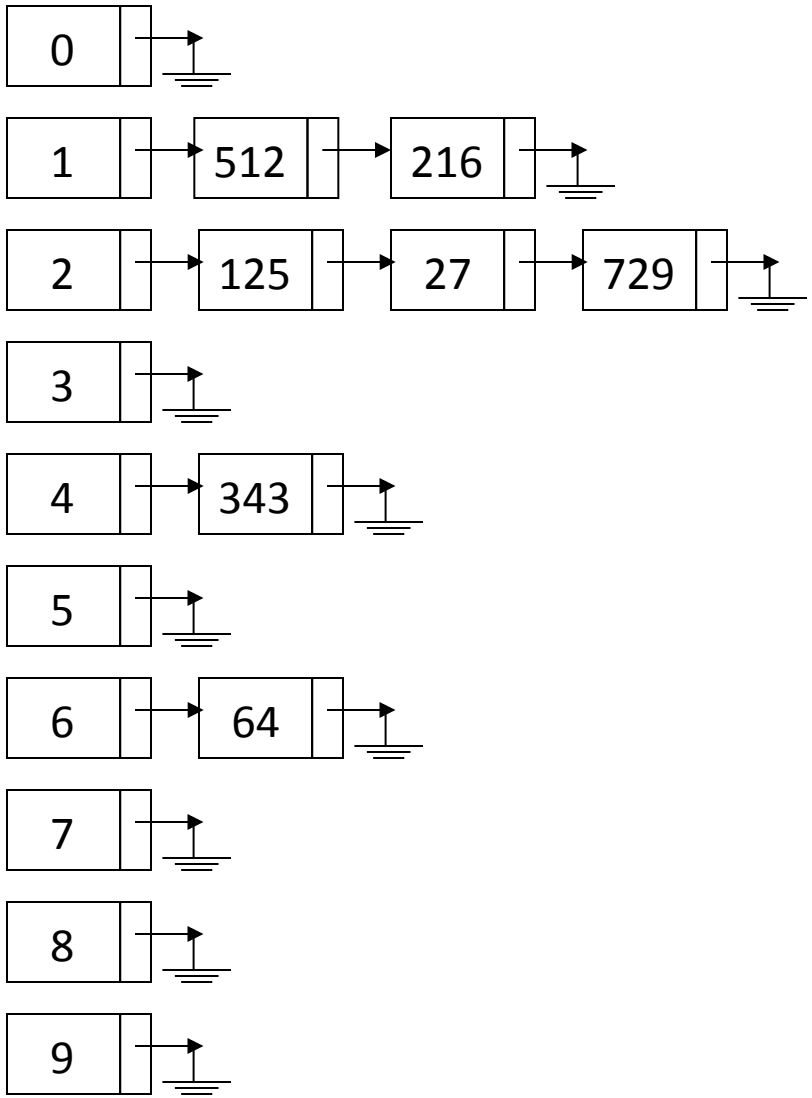

0 1 512 343 64 125 216 27 8 729
0 0 1 8
1 512 216
2 125 27 729
3
4 343
5
6 64
7
8
9


0
0
1
8
1
512
216
2
125
27
729
3
4
343
5
6
64
7
8
9

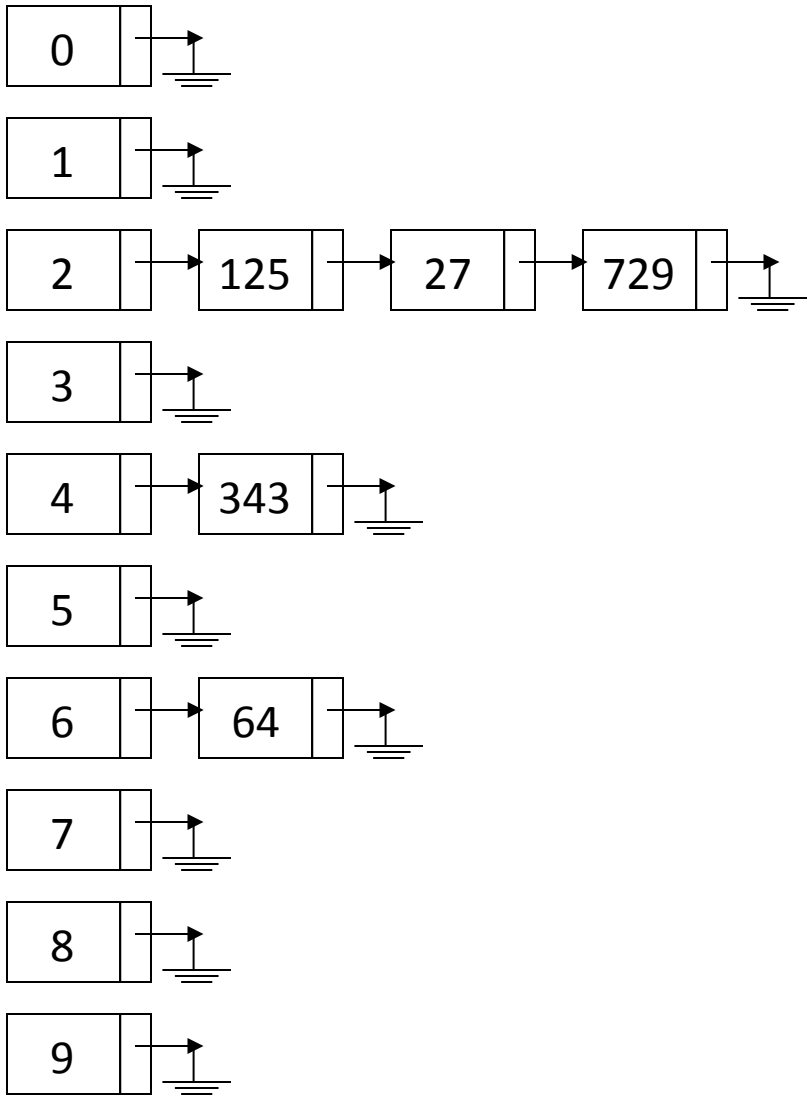

0
1
8
0
0
1
8
1
512
216
2
125
27
729
3
4
343
5
6
64
7
8
9


0
1
8
0
1
512
216
2
125
27
729
3
4
343
5
6
64
7
8
9

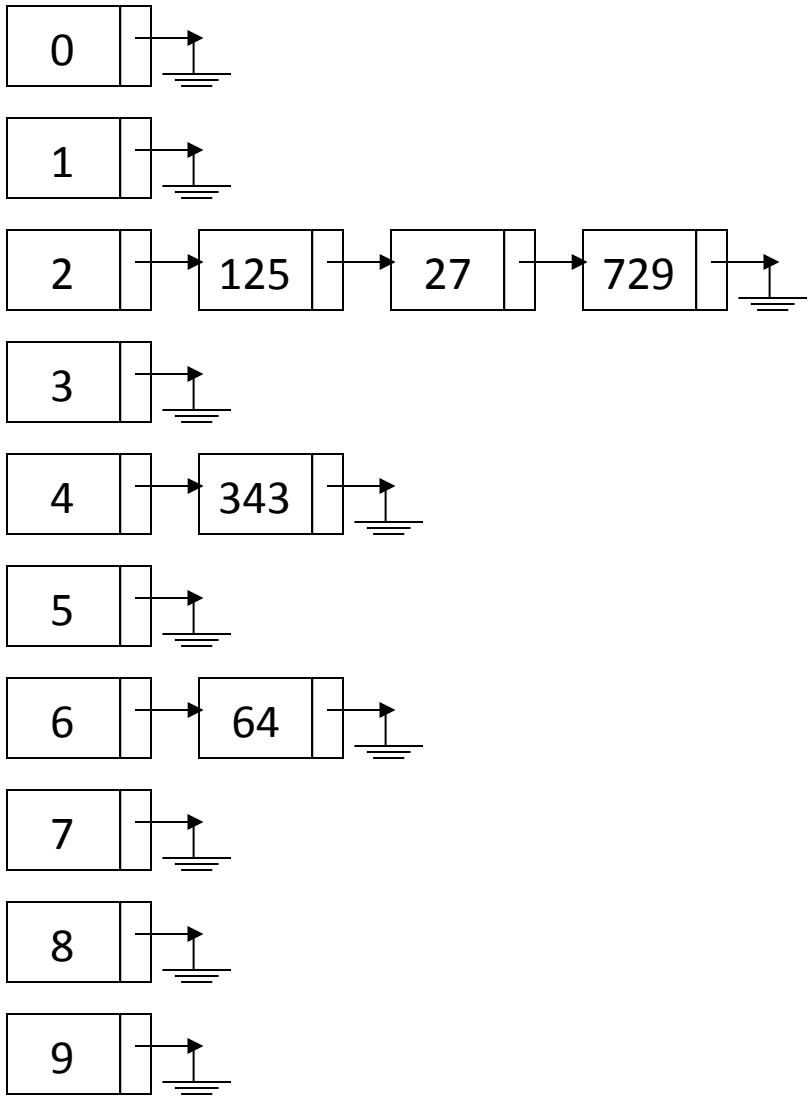
0	1	8	512	216					



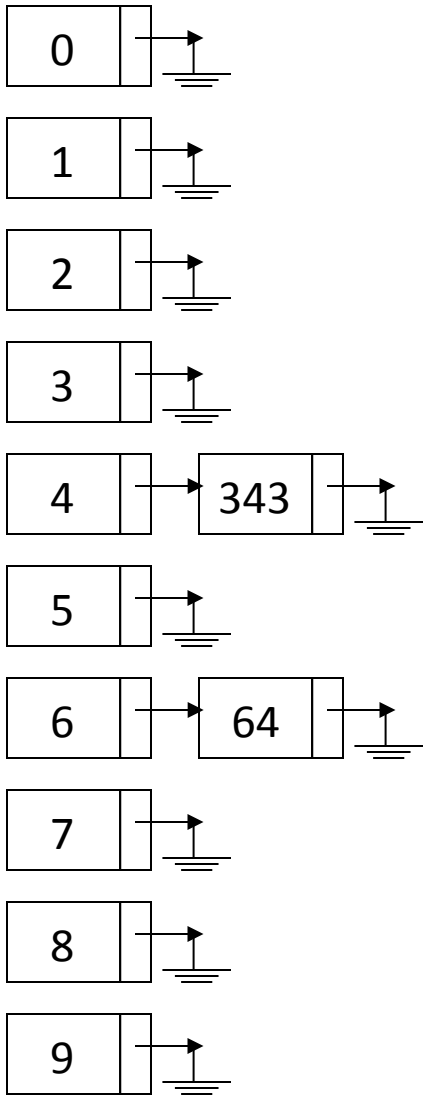
0	1	8	512	216					



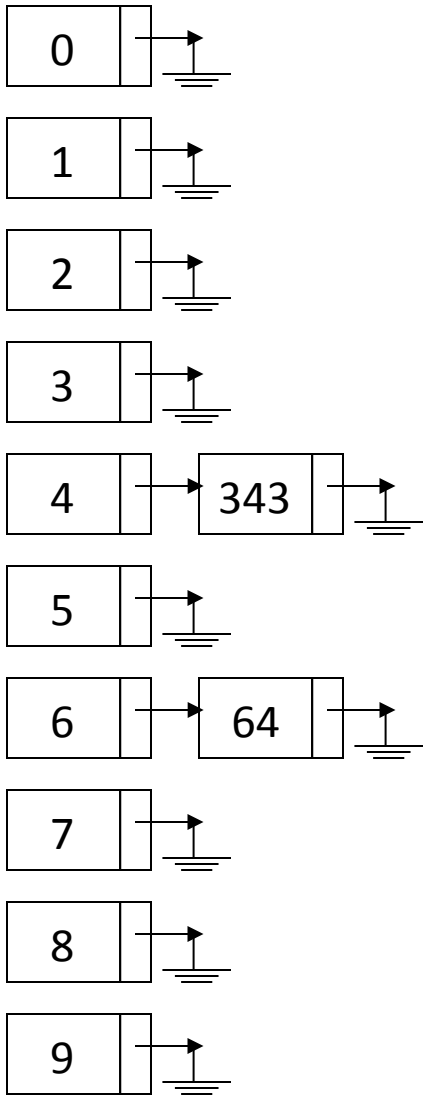
0	1	8	512	216	125	27	729		



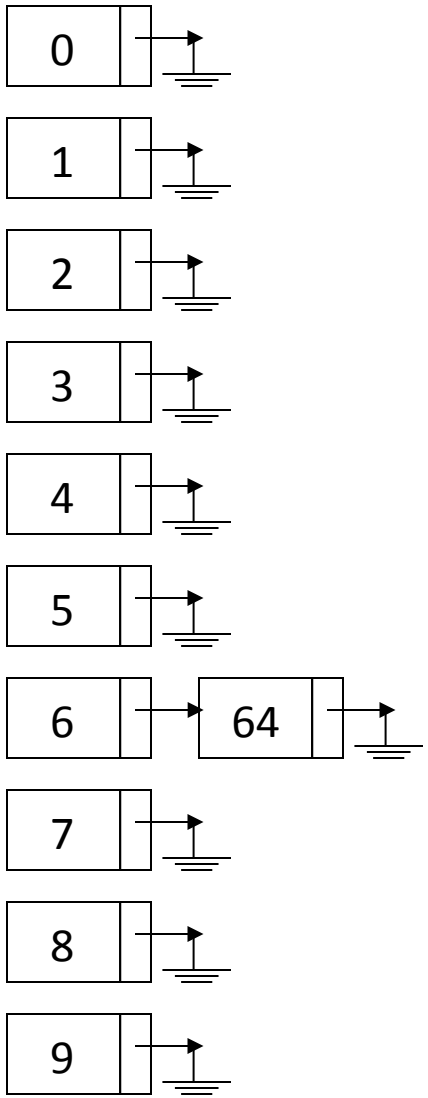
0	1	8	512	216	125	27	729		



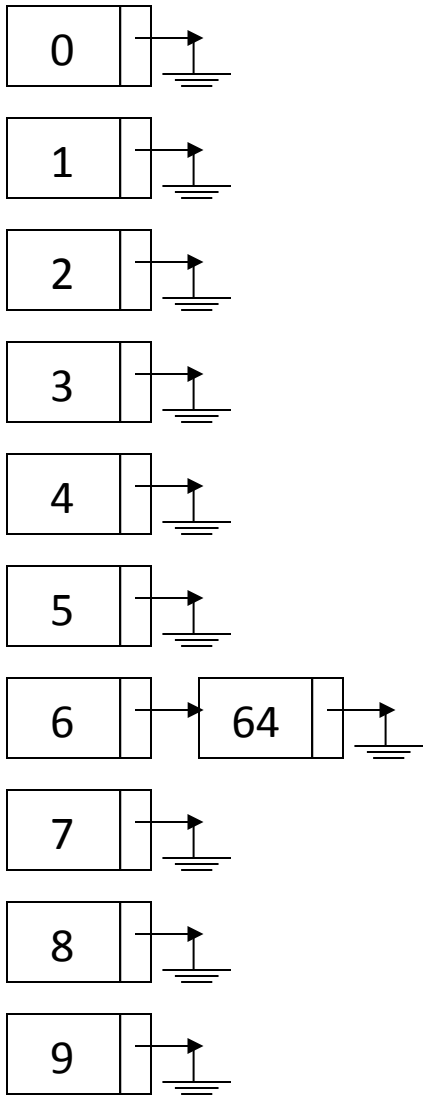
0	1	8	512	216	125	27	729	343	



0	1	8	512	216	125	27	729	343	

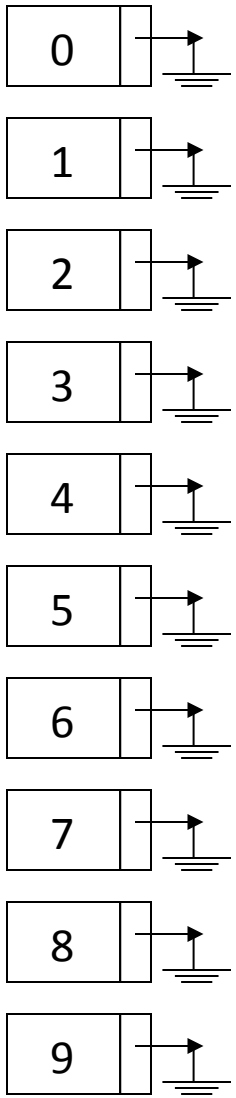


0	1	8	512	216	125	27	729	343	64

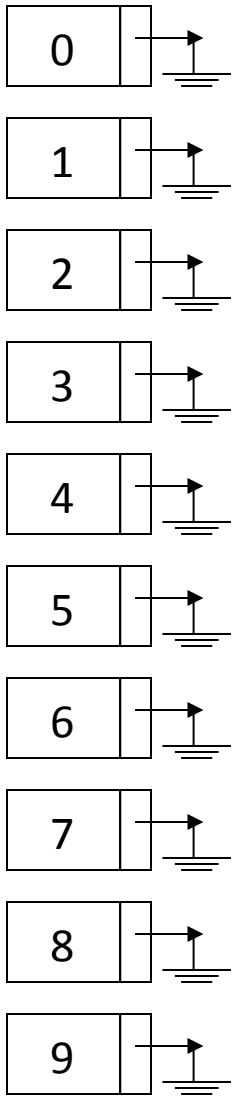


Pass 3

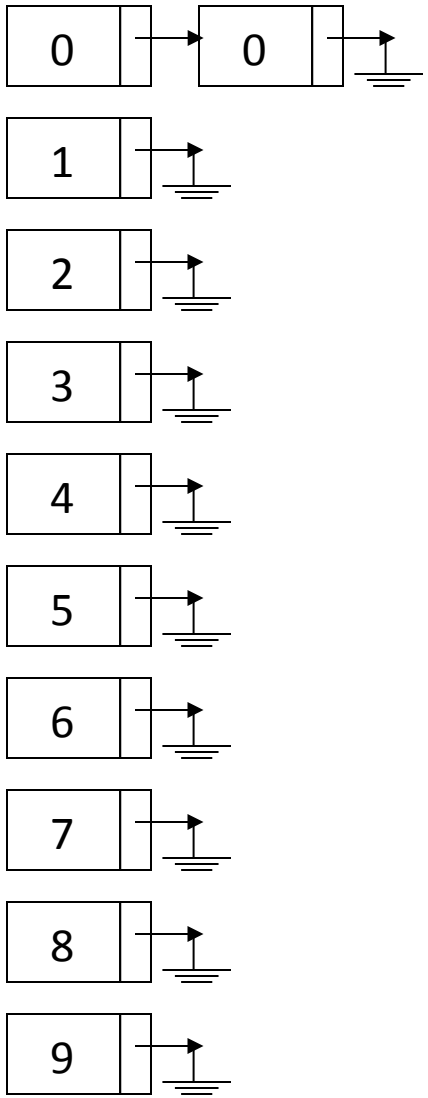
0	1	8	512	216	125	27	729	343	64



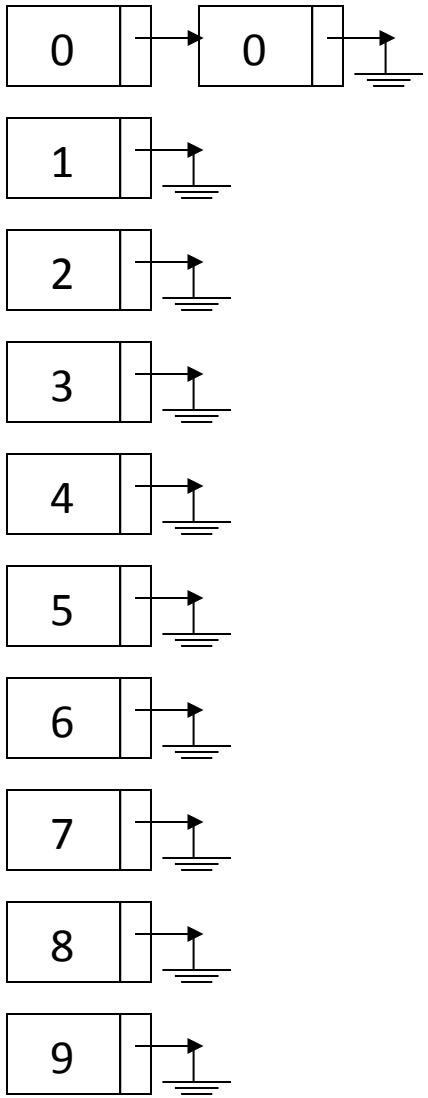
000	1	8	512	216	125	27	729	343	64



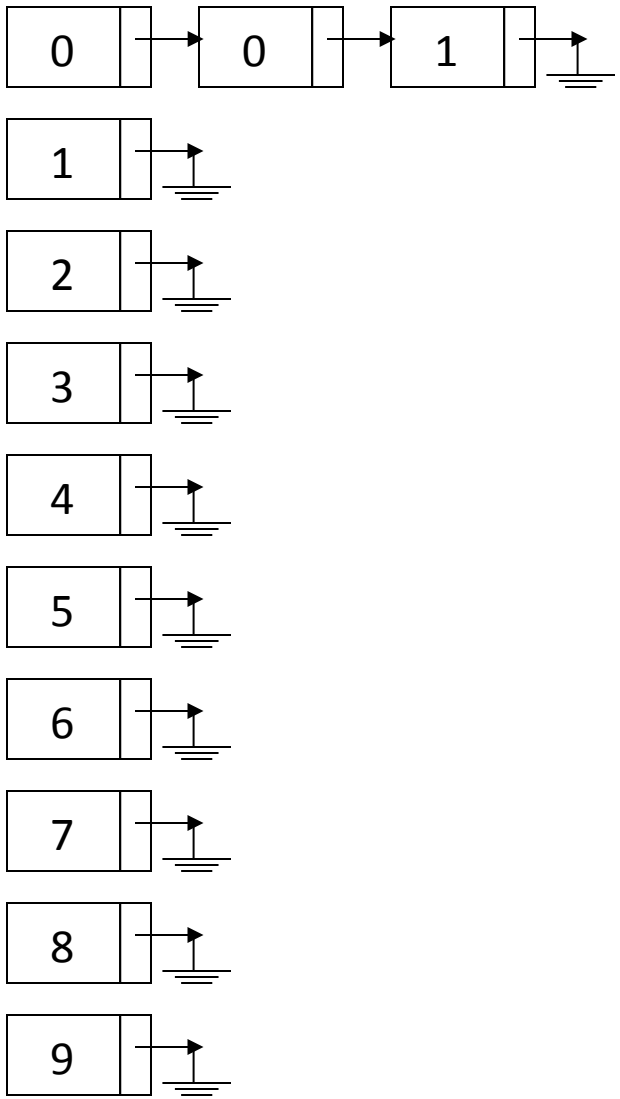
0	1	8	512	216	125	27	729	343	64



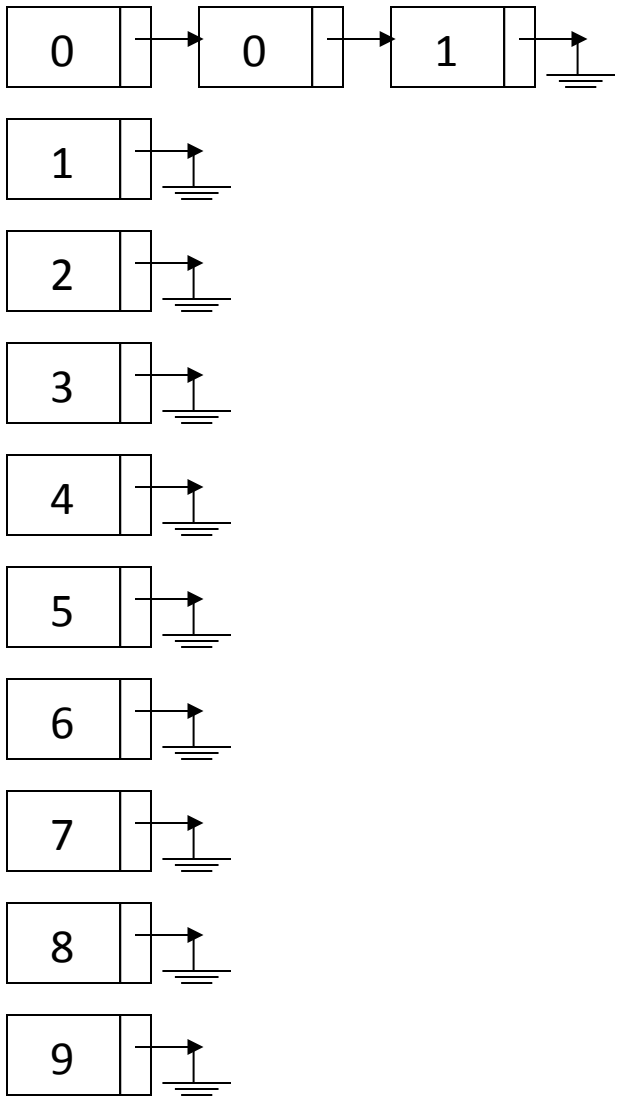
0	001	8	512	216	125	27	729	343	64

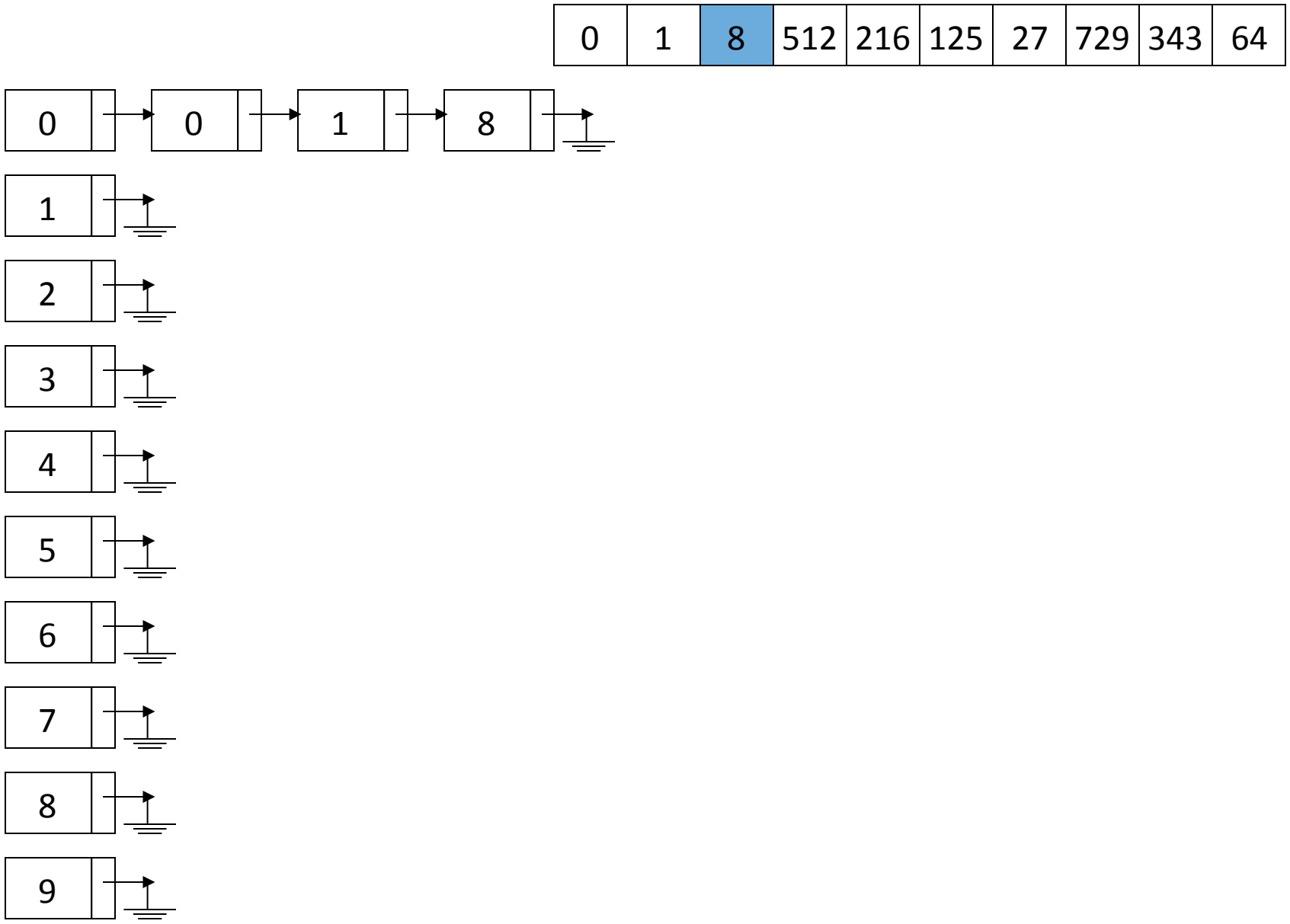


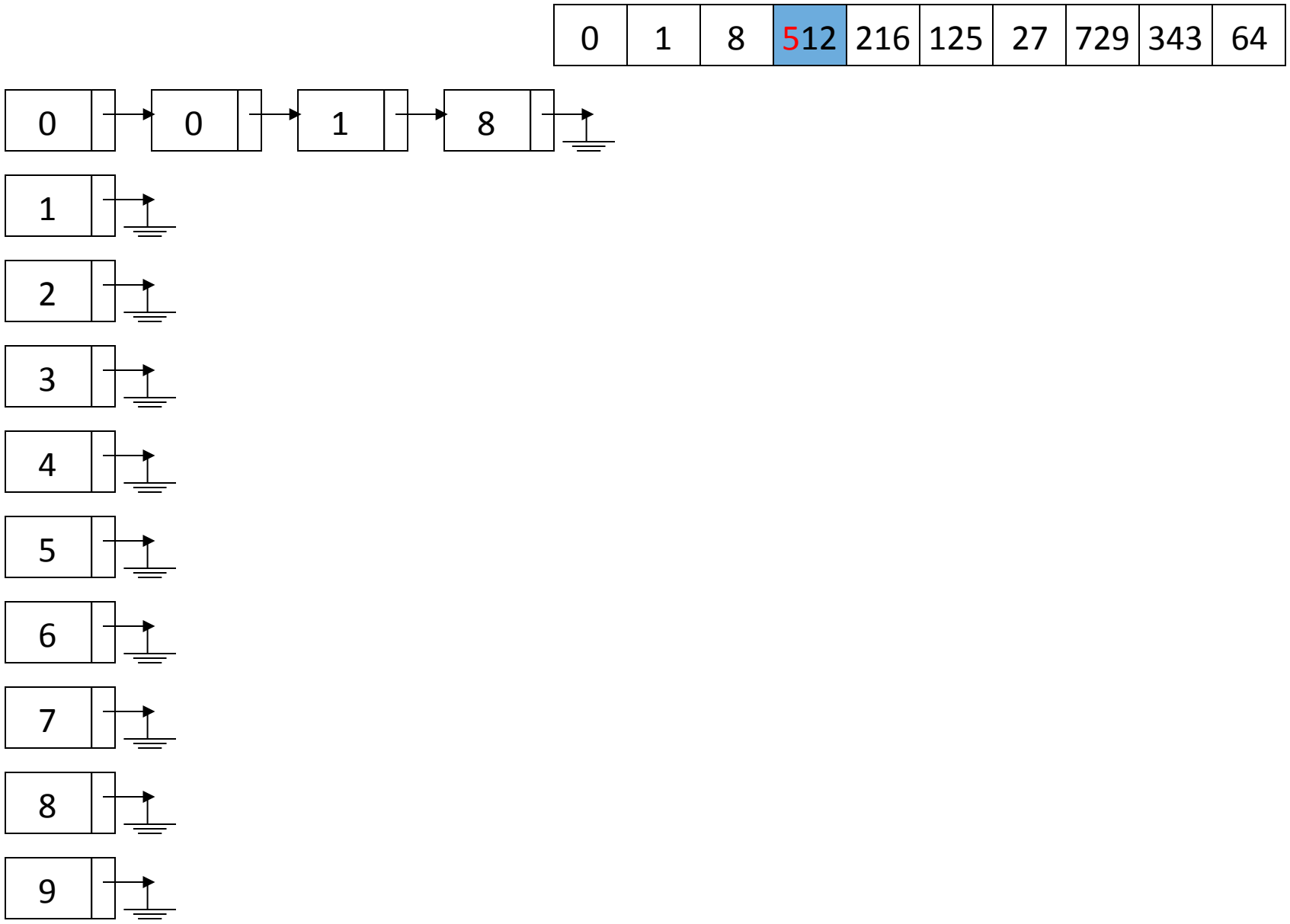
0	1	8	512	216	125	27	729	343	64

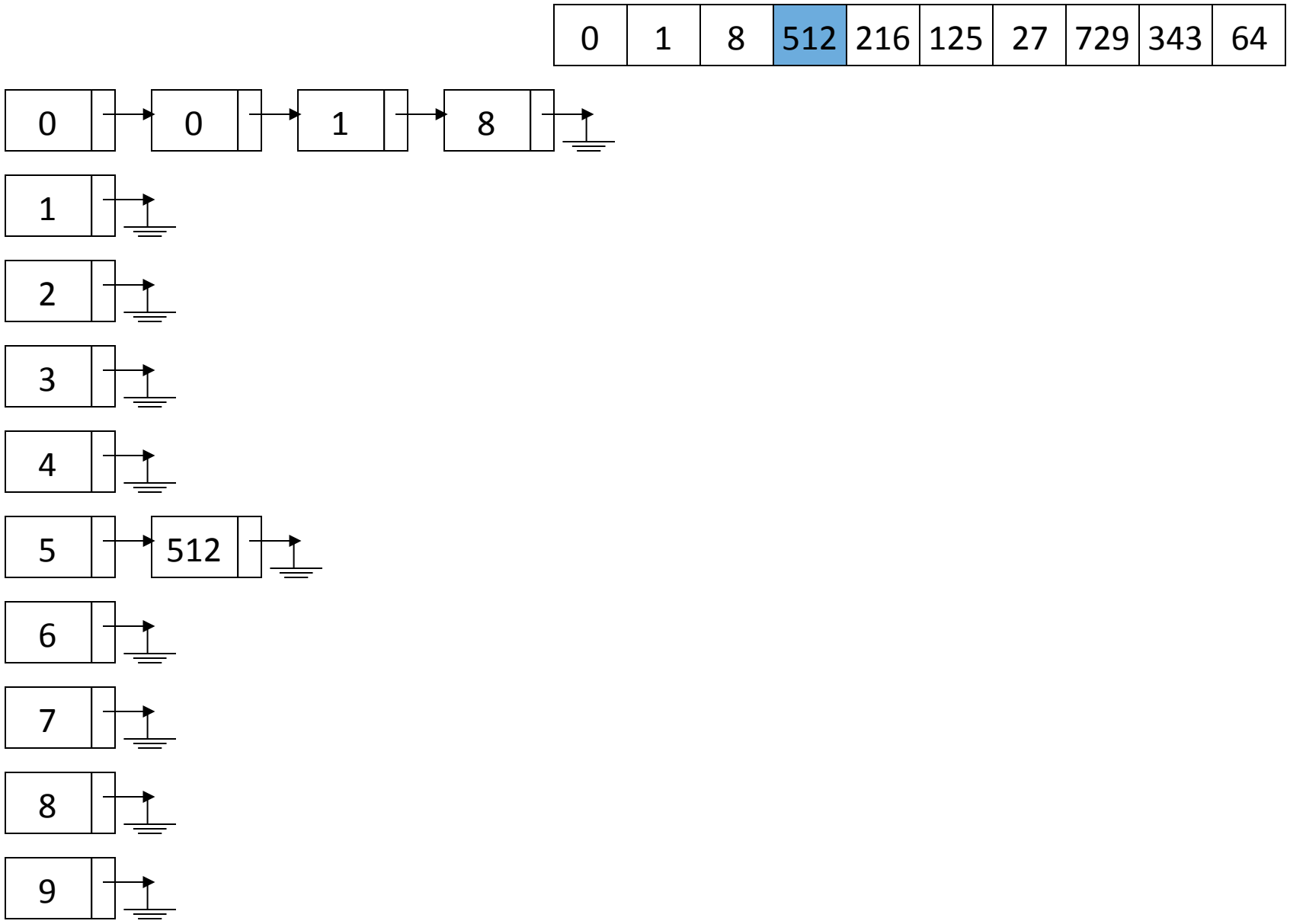


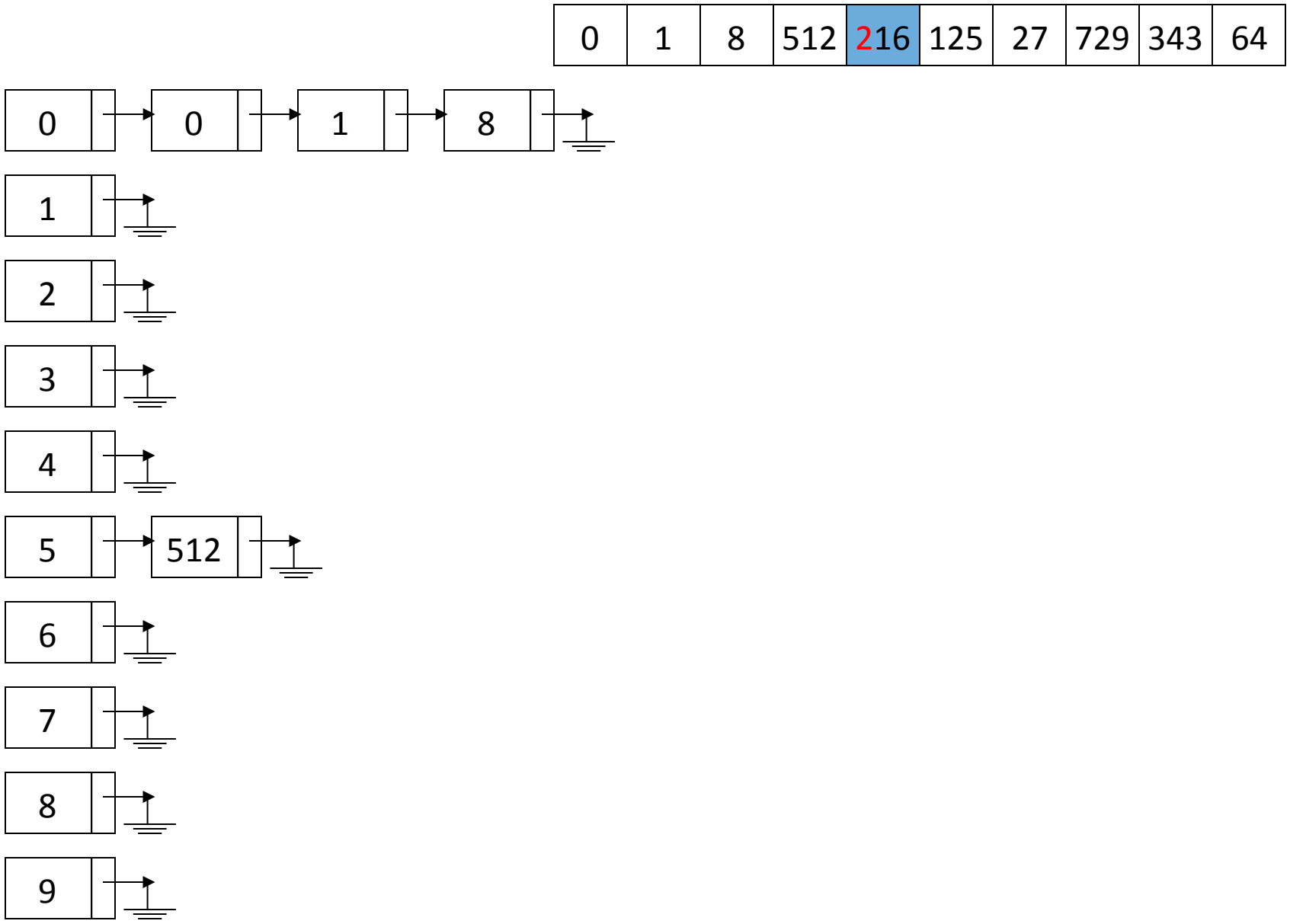
0	1	008	512	216	125	27	729	343	64

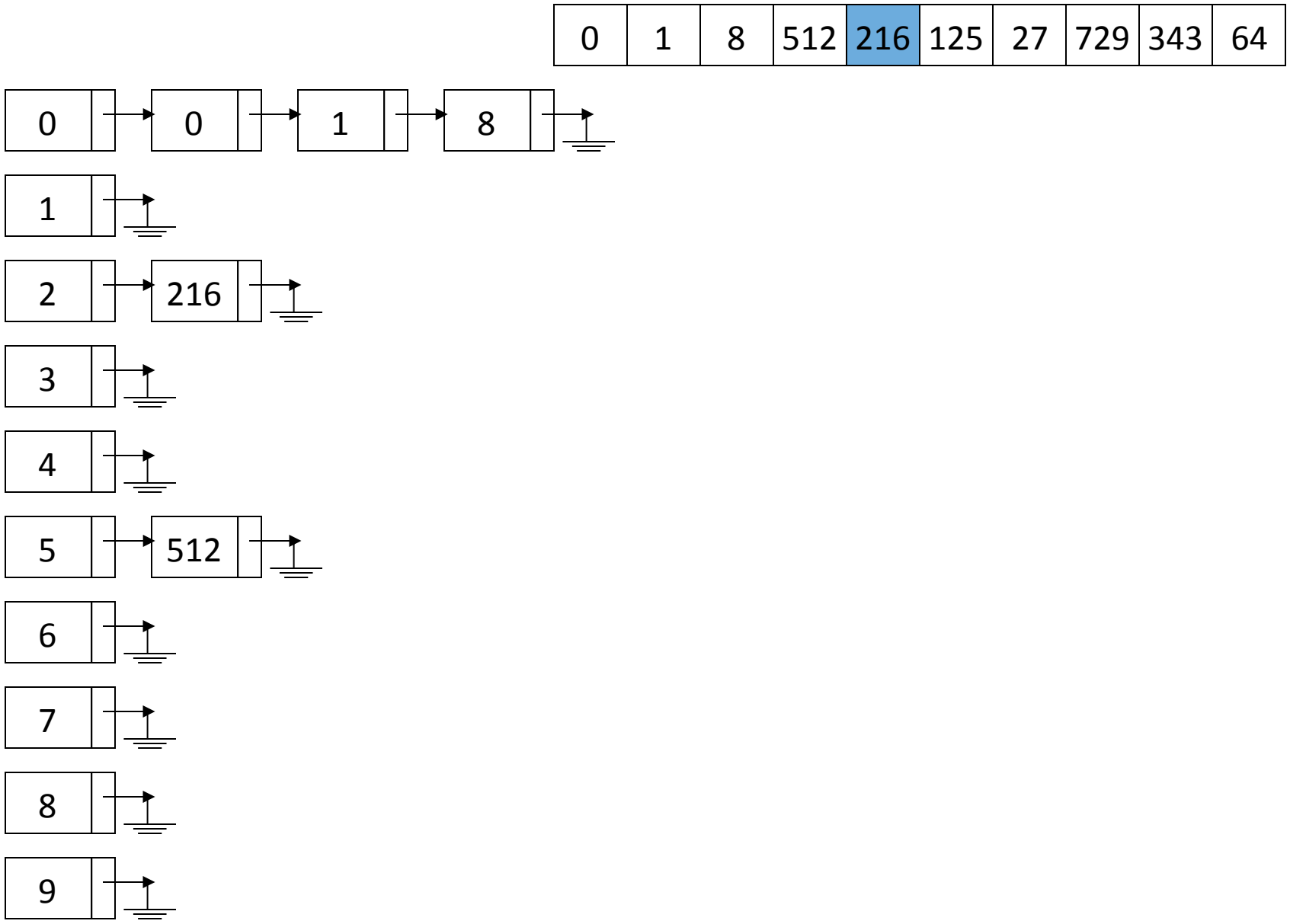


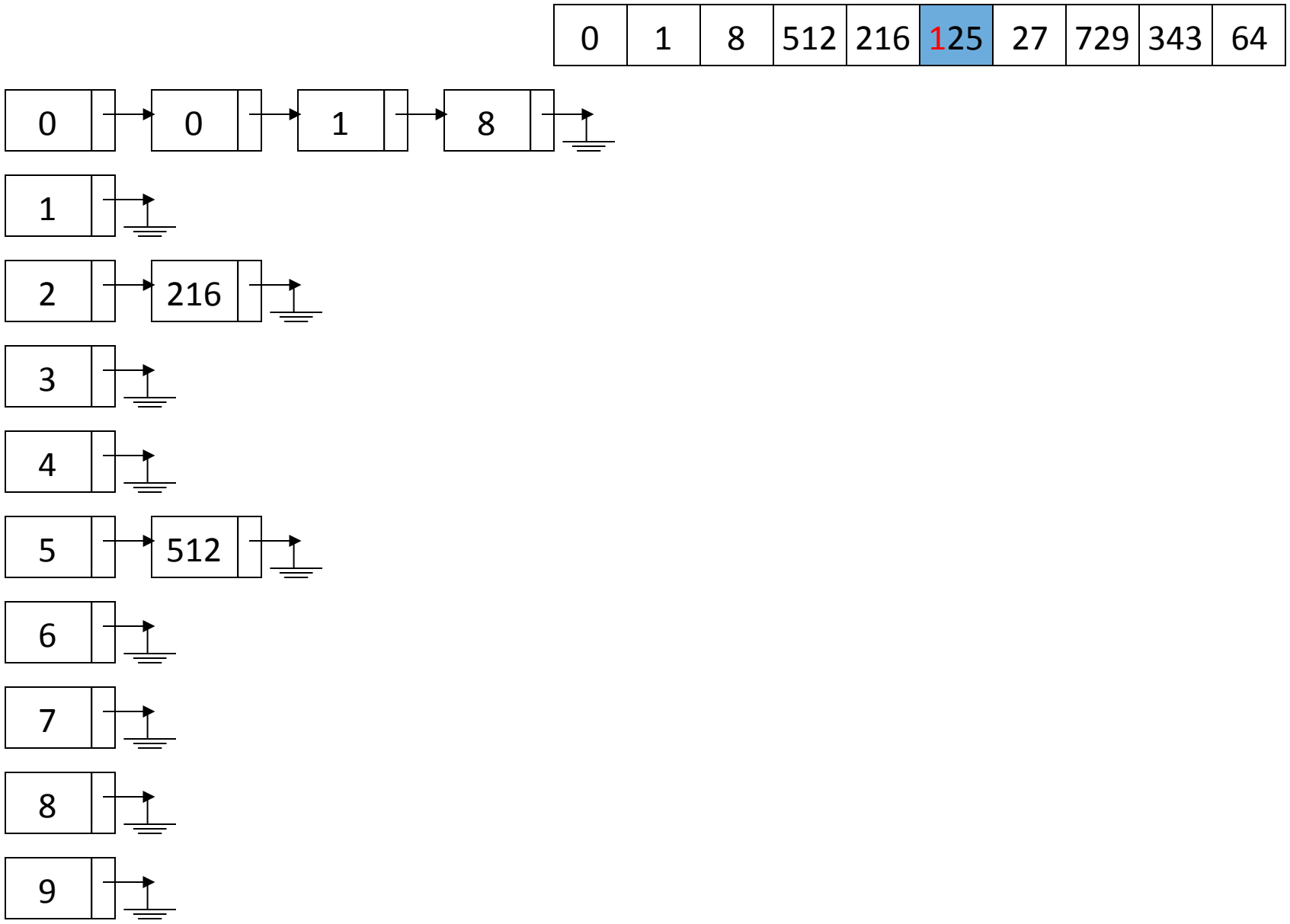

0
1
8
512
216
125
27
729
343
64
0
0
1
8
1
2
3
4
5
6
7
8
9

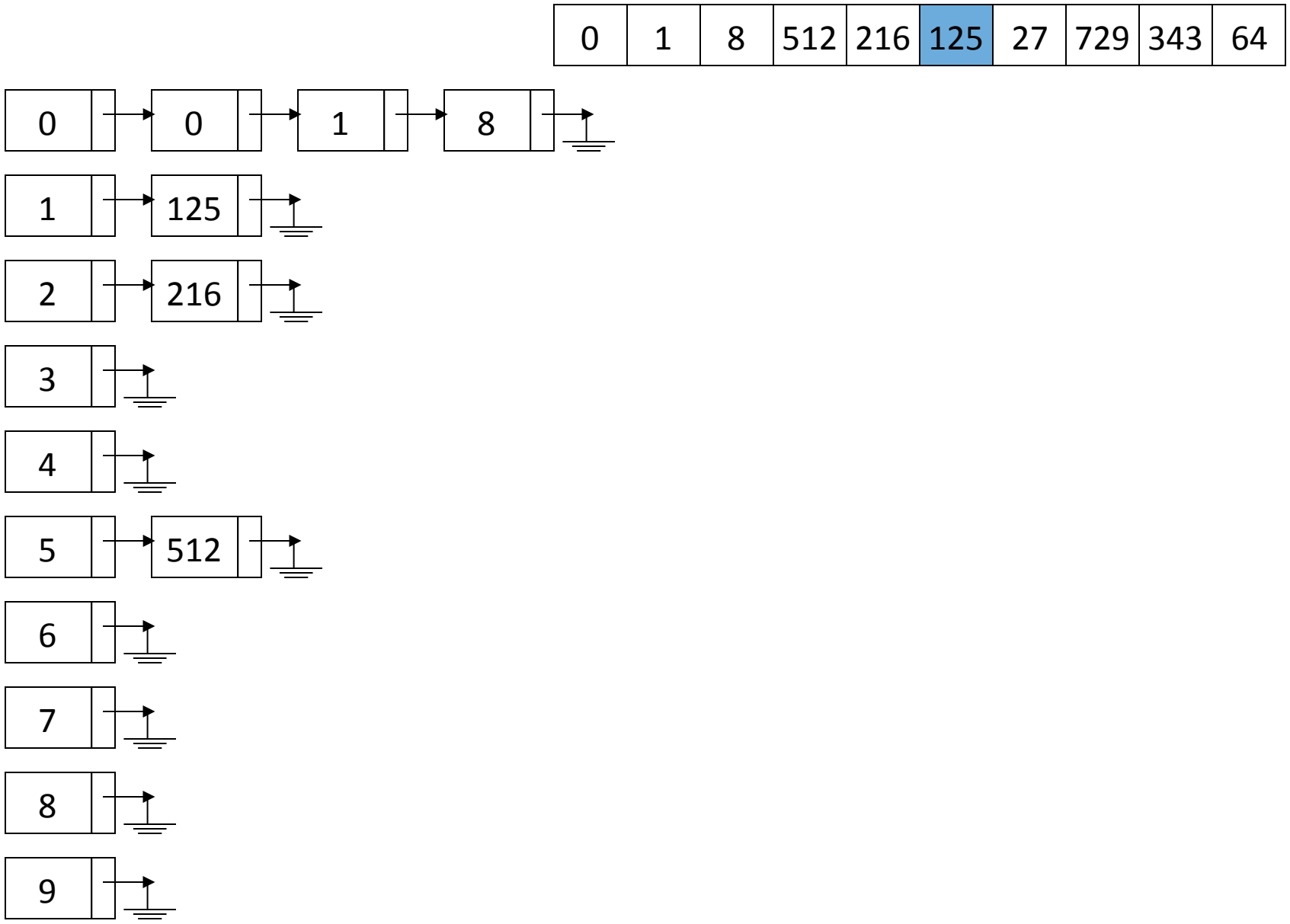

0
1
8
512
216
125
27
729
343
64
0
0
1
8
1
2
3
4
5
6
7
8
9

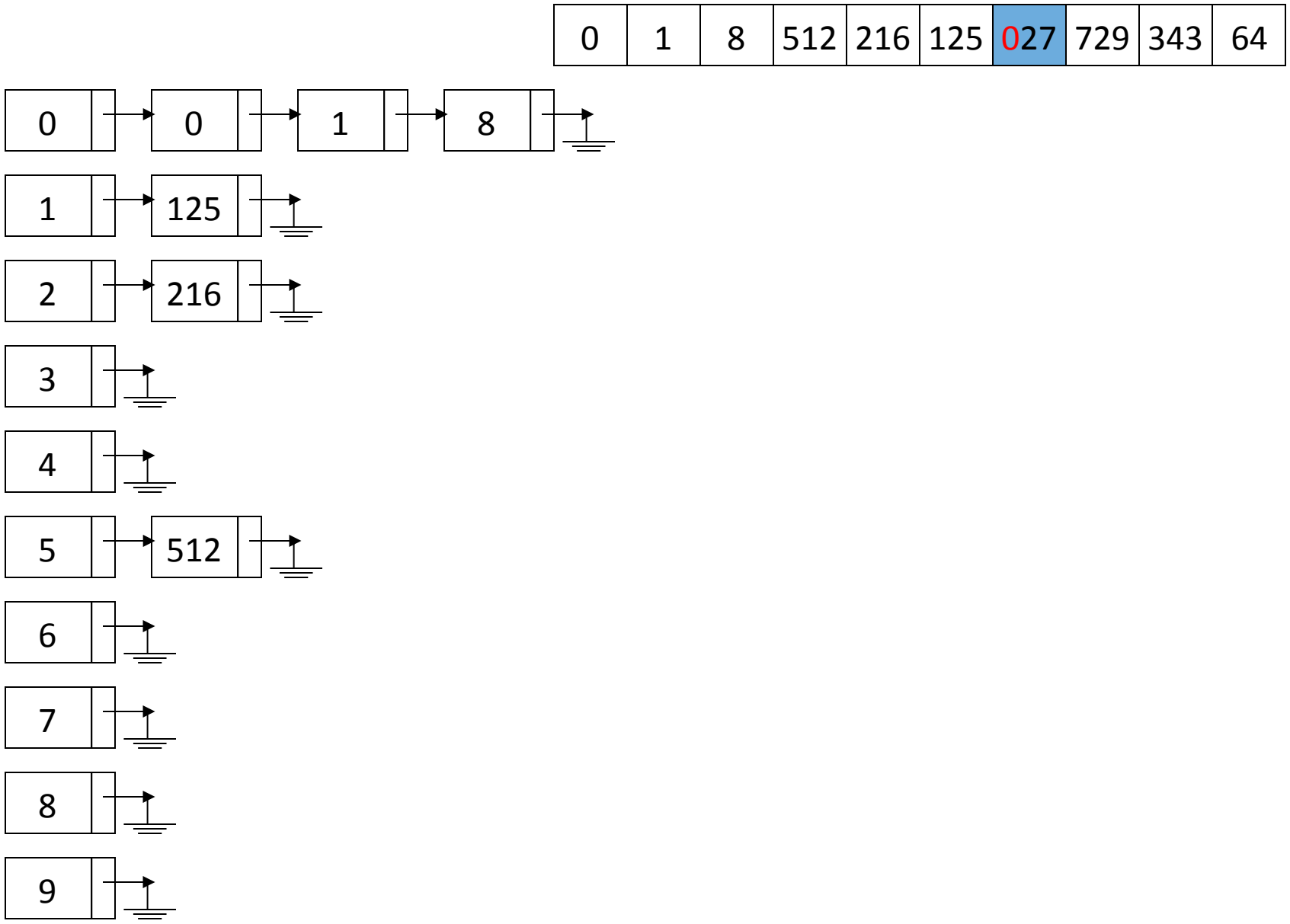

0
1
8
512
216
125
27
729
343
64
0
0
1
8
1
2
3
4
5
512
6
7
8
9

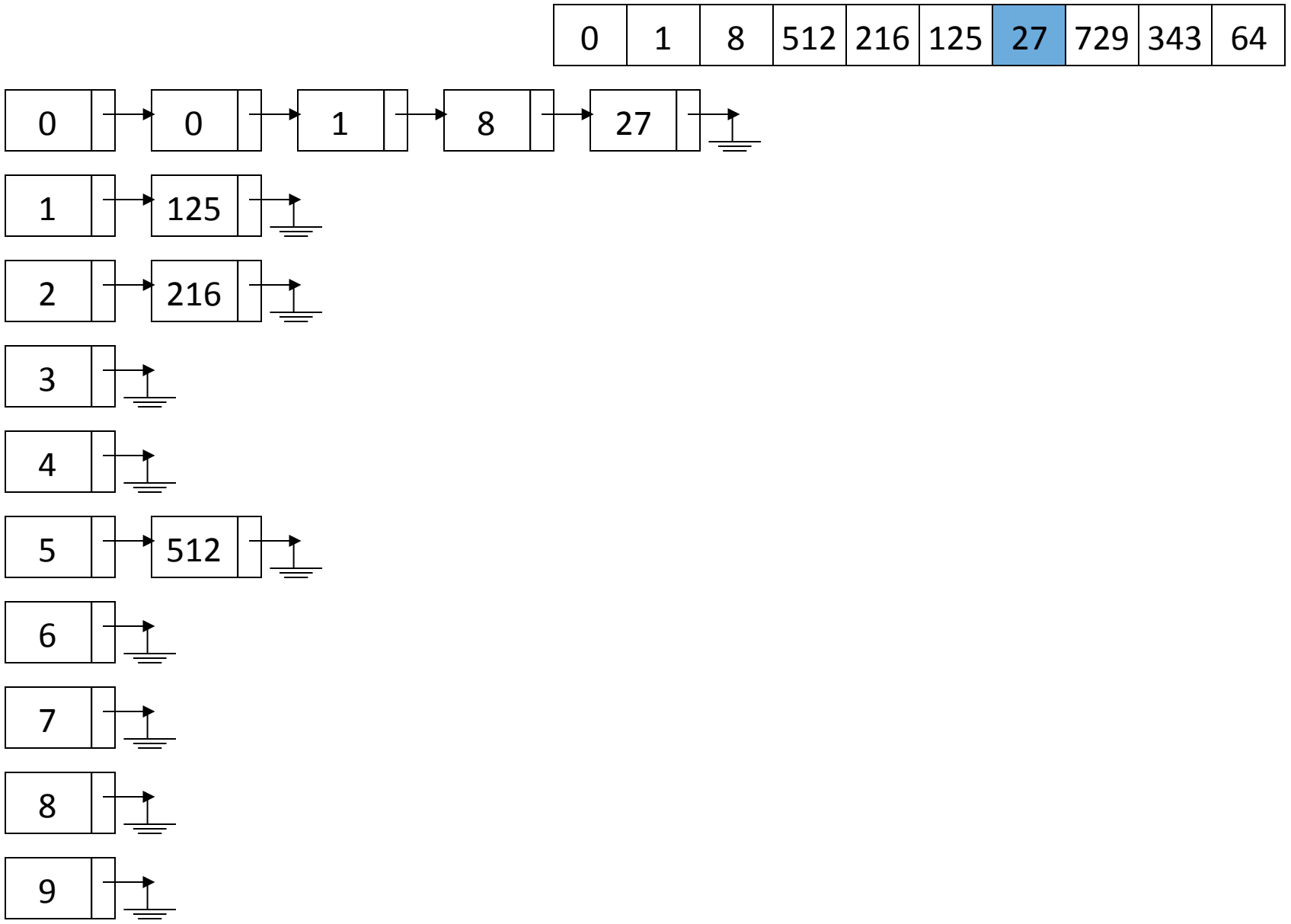

0
1
8
512
216
125
27
729
343
64
0
0
1
8
1
2
3
4
5
512
6
7
8
9

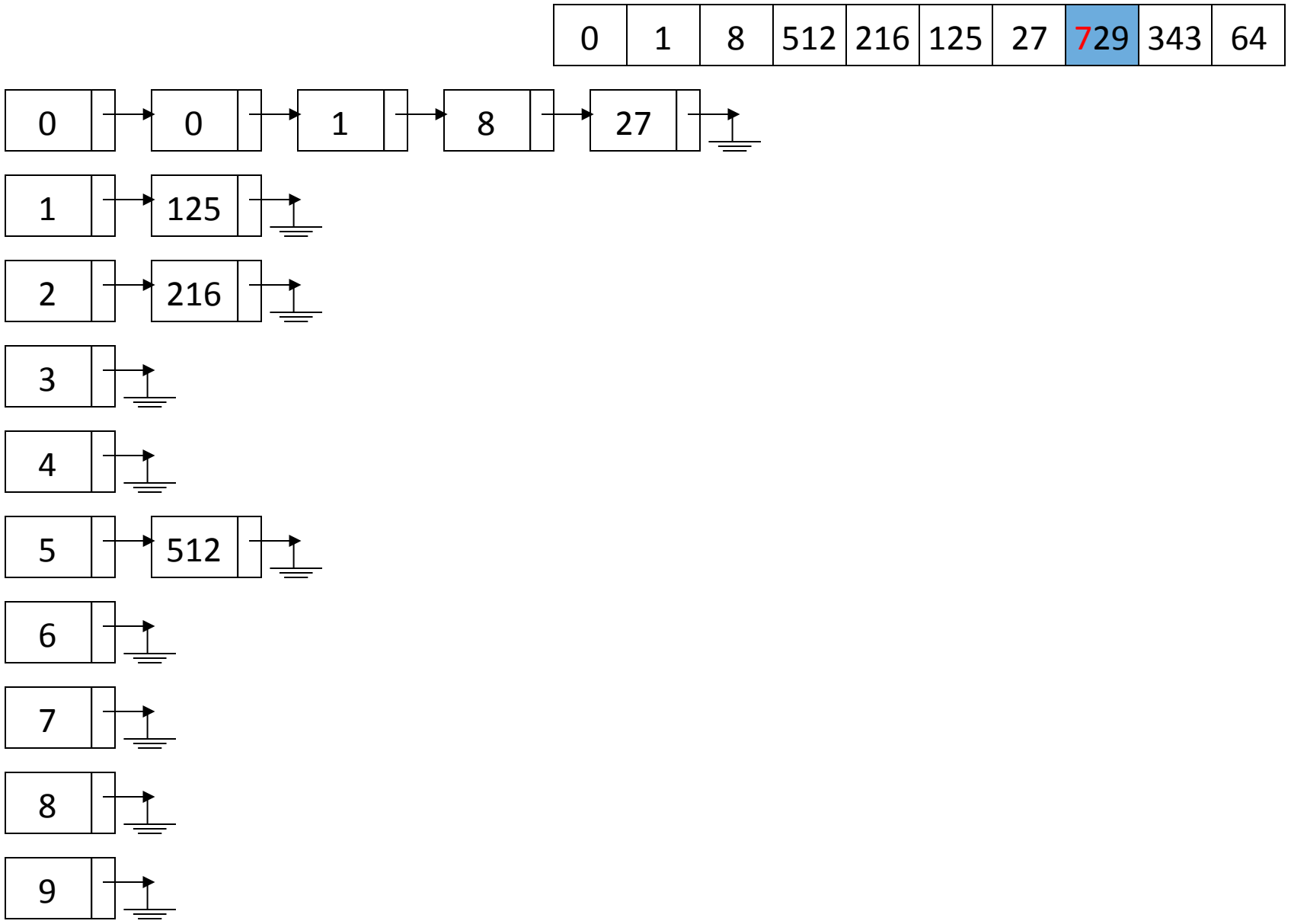

0
1
8
512
216
125
27
729
343
64
0
0
1
8
1
2
216
3
4
5
512
6
7
8
9

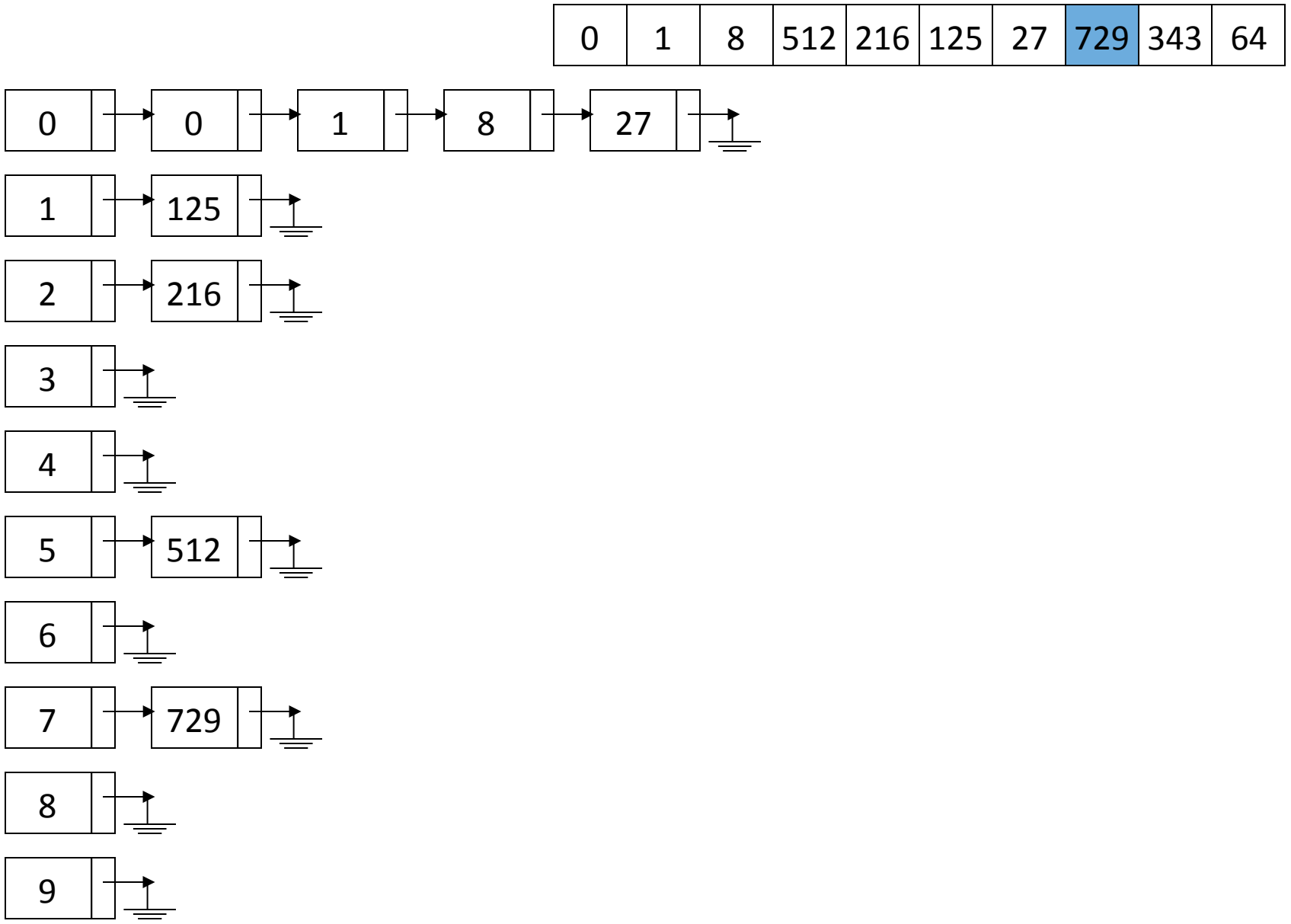

0
1
8
512
216
125
27
729
343
64
0
0
1
8
1
2
216
3
4
5
512
6
7
8
9

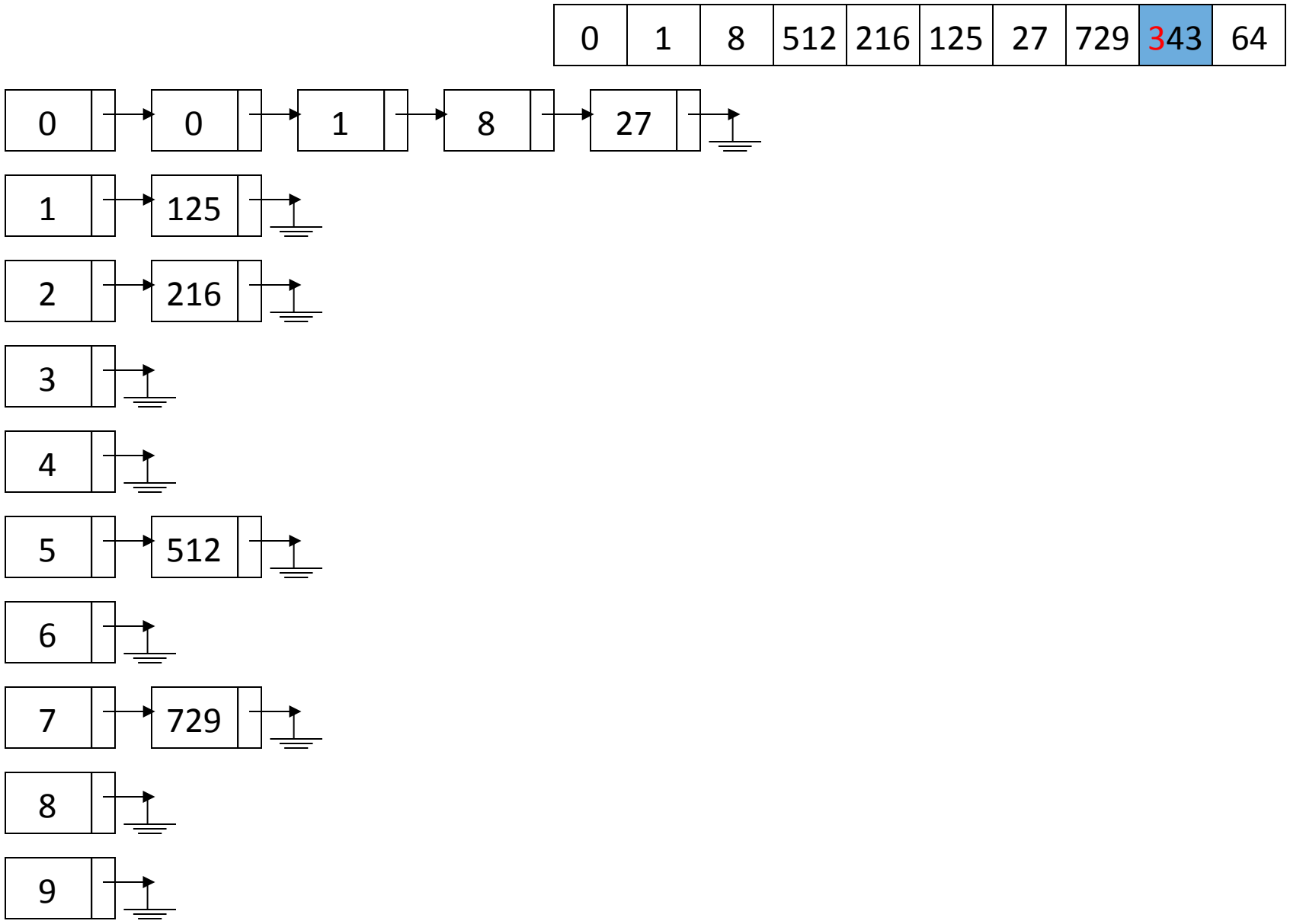

0
1
8
512
216
125
27
729
343
64
0
0
1
8
1
125
2
216
3
4
5
512
6
7
8
9

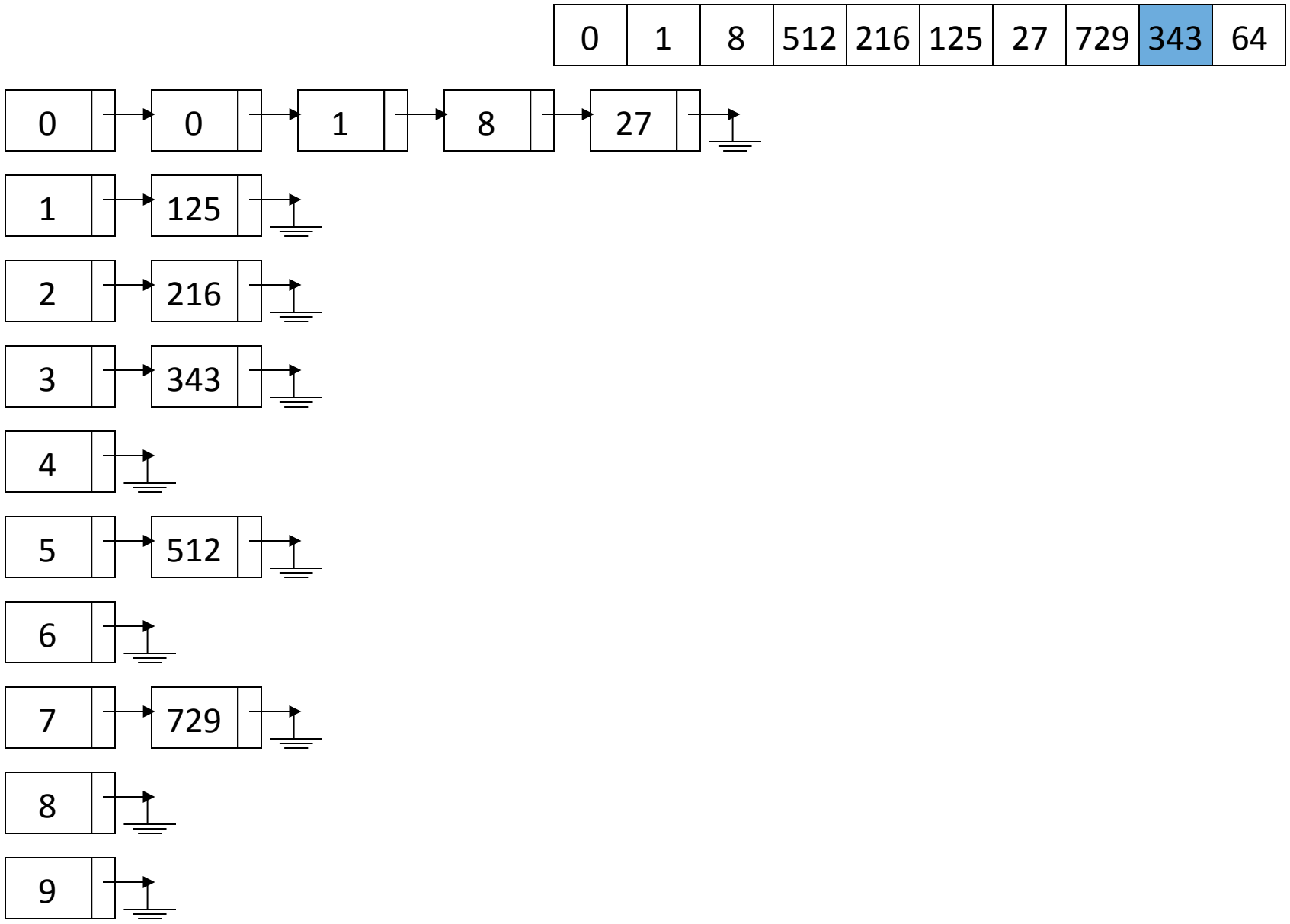

0
1
8
512
216
125
027
729
343
64
0
0
1
8
1
125
2
216
3
4
5
512
6
7
8
9

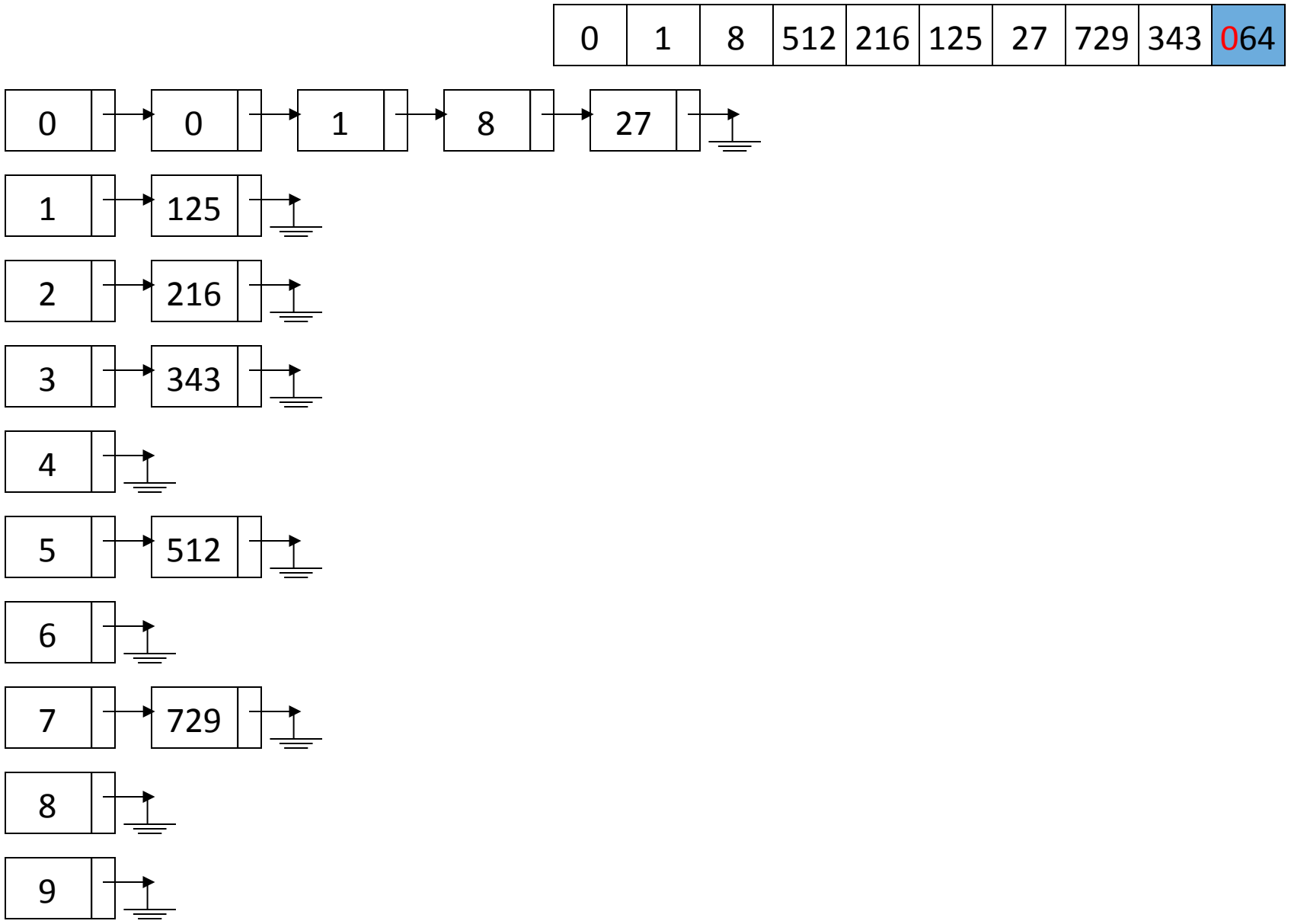

0
1
8
512
216
125
27
729
343
64
0
0
1
8
27
1
125
2
216
3
4
5
512
6
7
8
9

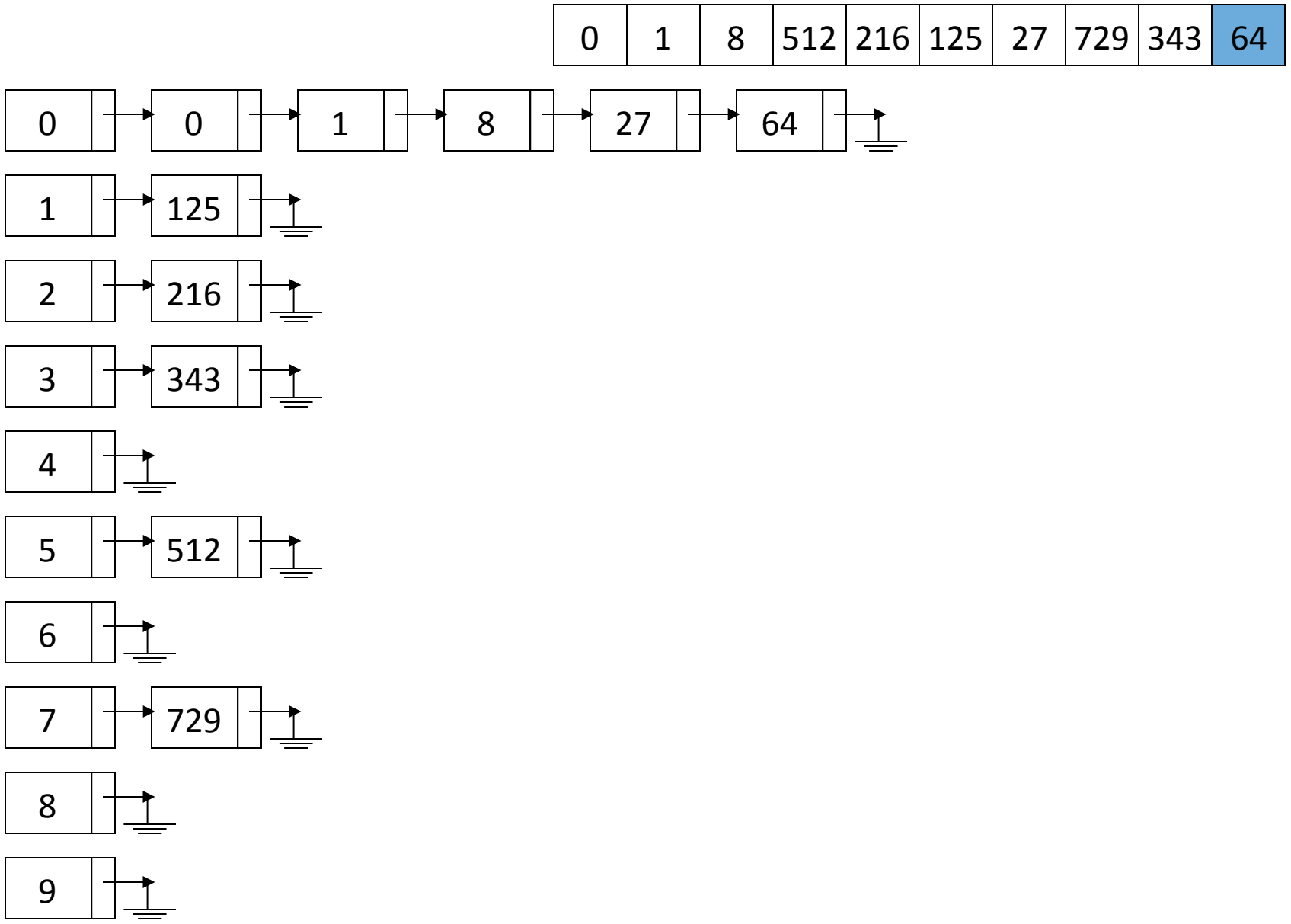

0
1
8
512
216
125
27
729
343
64
0
0
1
8
27
1
125
2
216
3
4
5
512
6
7
8
9

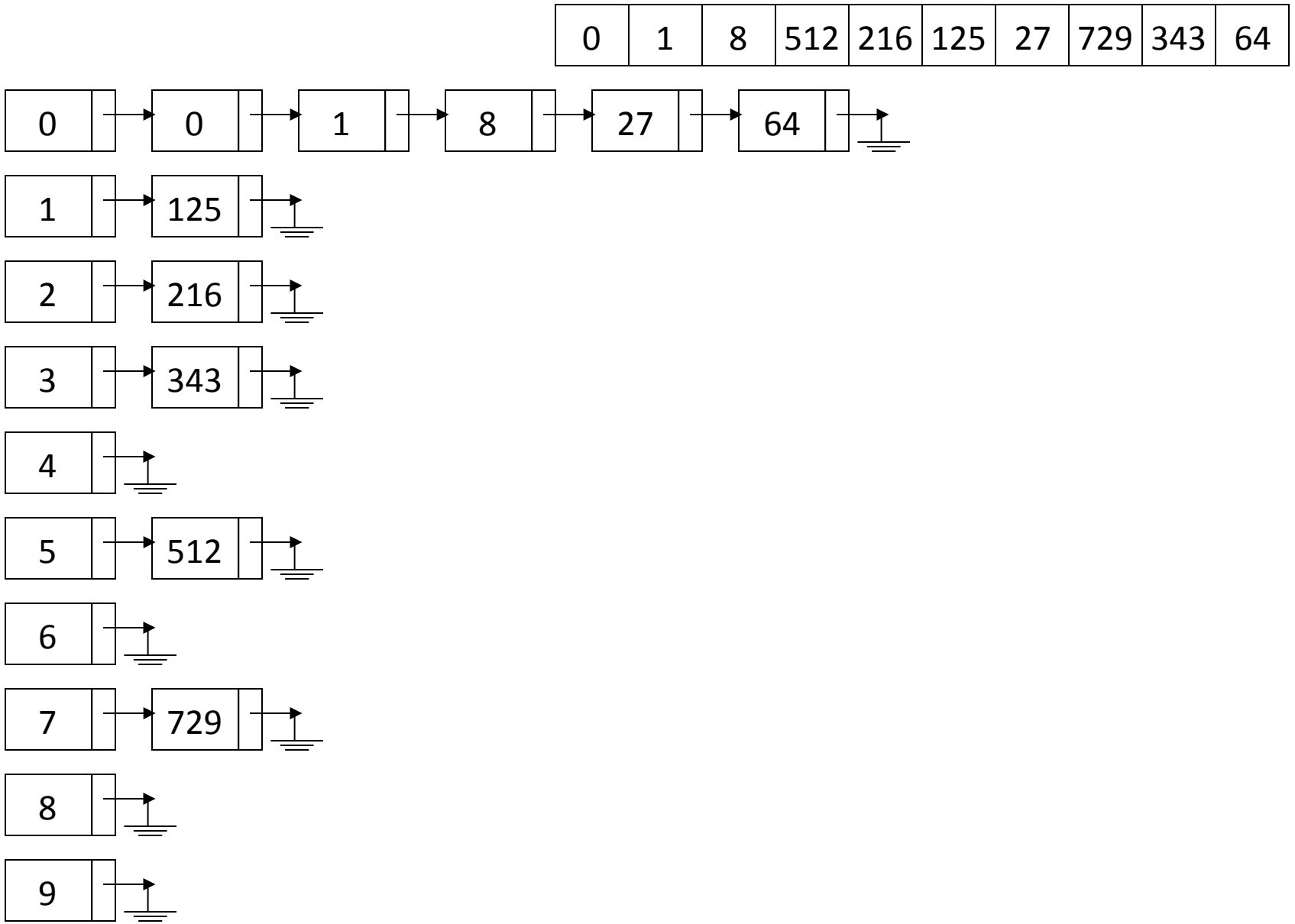

0
1
8
512
216
125
27
729
343
64
0
0
1
8
27
1
125
2
216
3
4
5
512
6
7
729
8
9

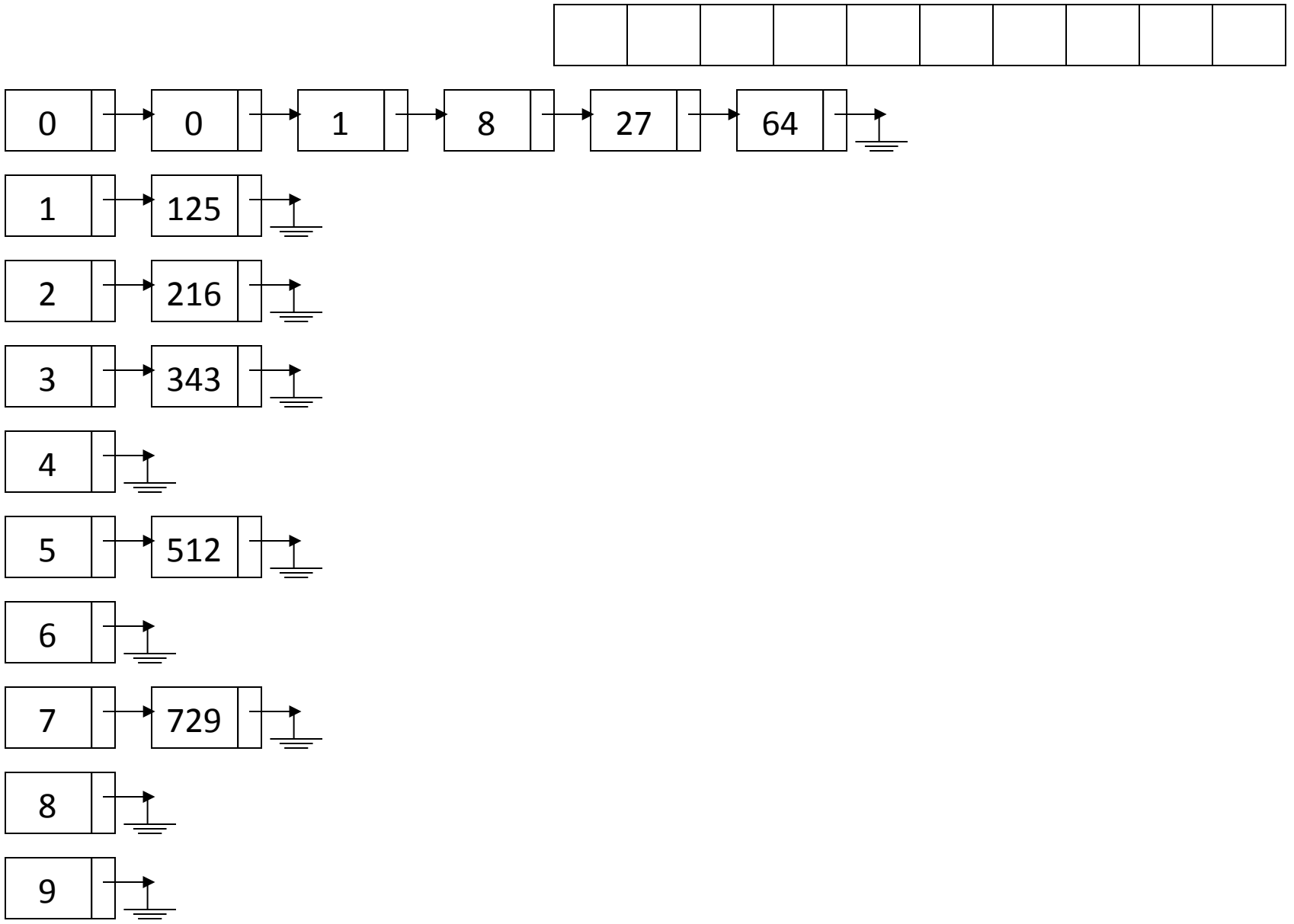

0
1
8
512
216
125
27
729
343
64
0
0
1
8
27
1
125
2
216
3
4
5
512
6
7
729
8
9

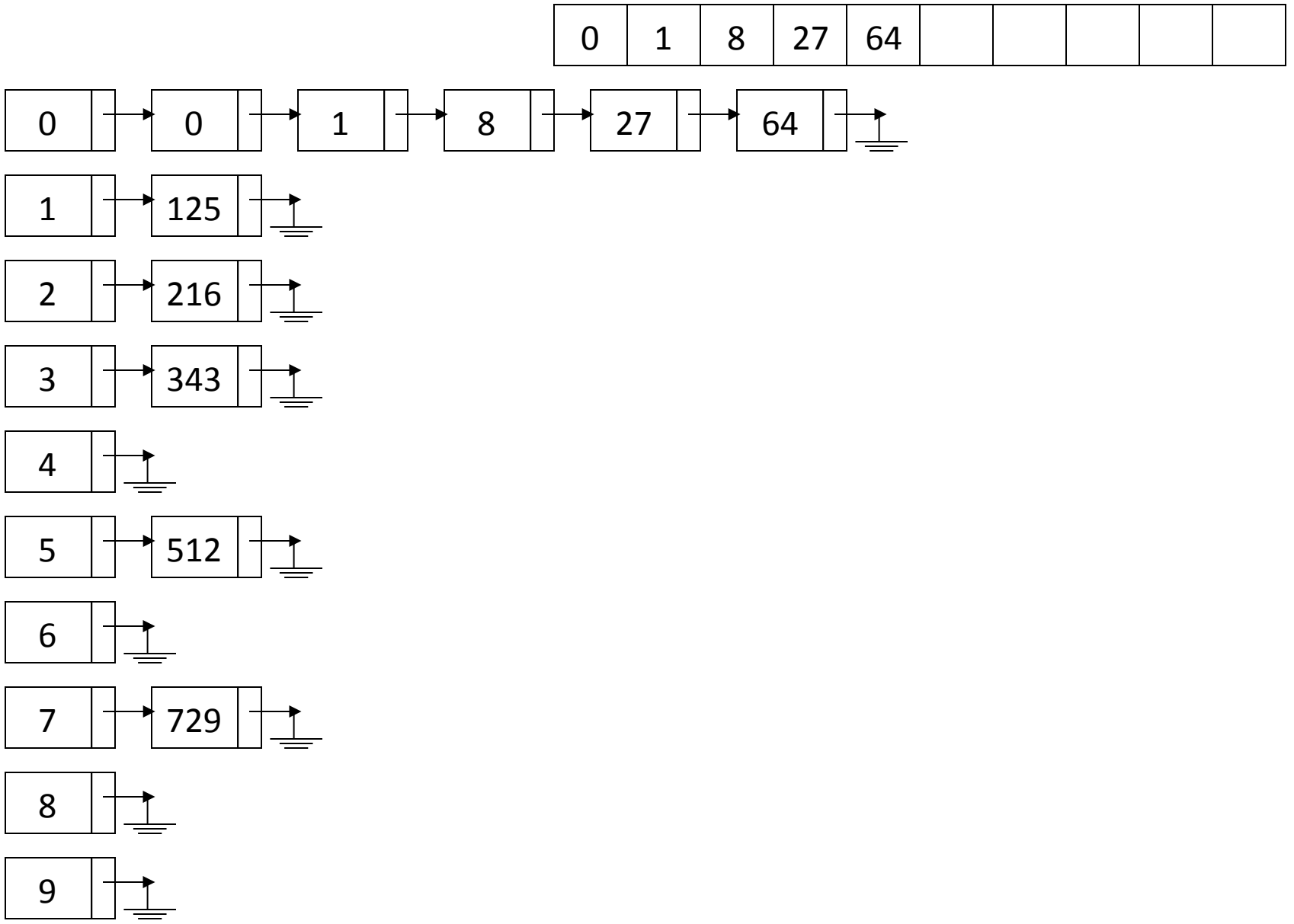

0
1
8
512
216
125
27
729
343
64
0
0
1
8
27
1
125
2
216
3
343
4
5
512
6
7
729
8
9


0 1 8 512 216 125 27 729 343 064
0 0 1 8 27
1 125
2 216
3 343
4
5 512
6
7 729
8
9

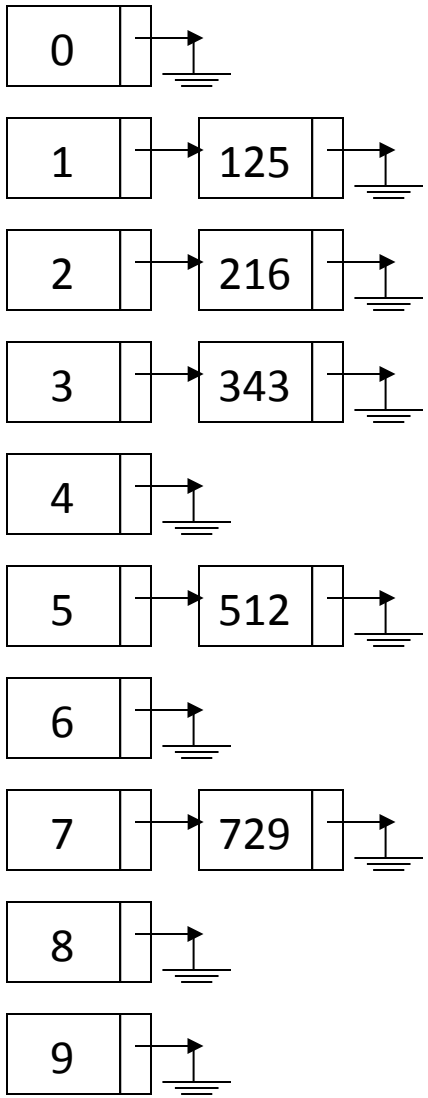

0
1
8
512
216
125
27
729
343
64
0
0
1
8
27
64
1
125
2
216
3
343
4
5
512
6
7
729
8
9


0 1 8 512 216 125 27 729 343 64
0 0 1 8 27 64
1 125
2 216
3 343
4
5 512
6
7 729
8
9

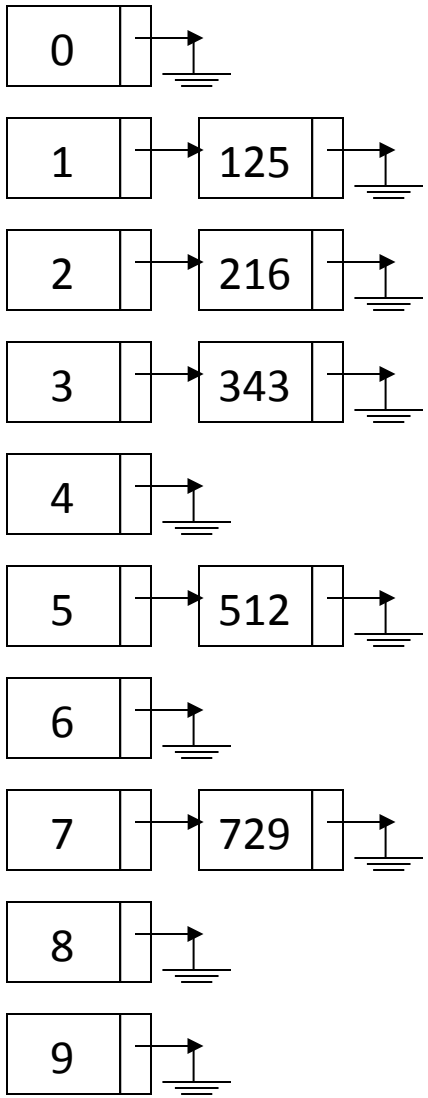

0
0
1
8
27
64
1
125
2
216
3
343
4
5
512
6
7
729
8
9


0
1
8
27
64
0
0
1
8
27
64
1
125
2
216
3
343
4
5
512
6
7
729
8
9

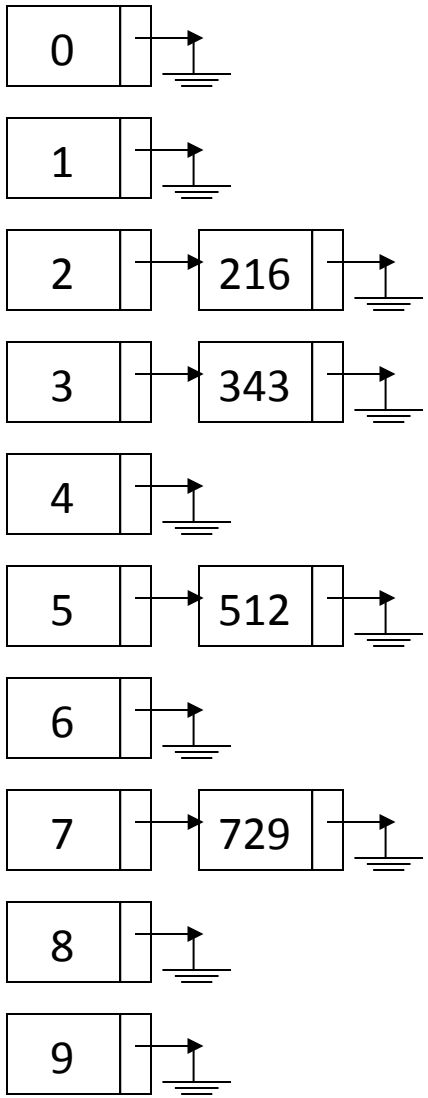
0	1	8	27	64					



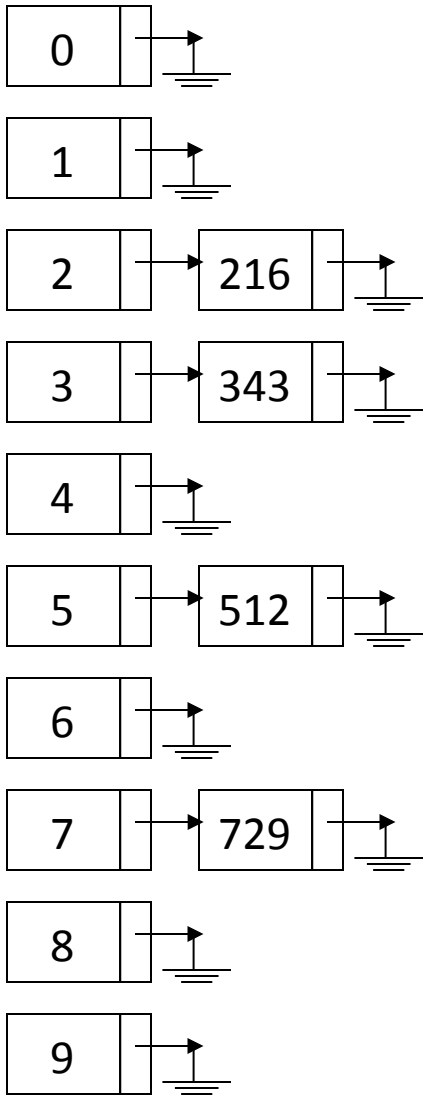
0	1	8	27	64	125				



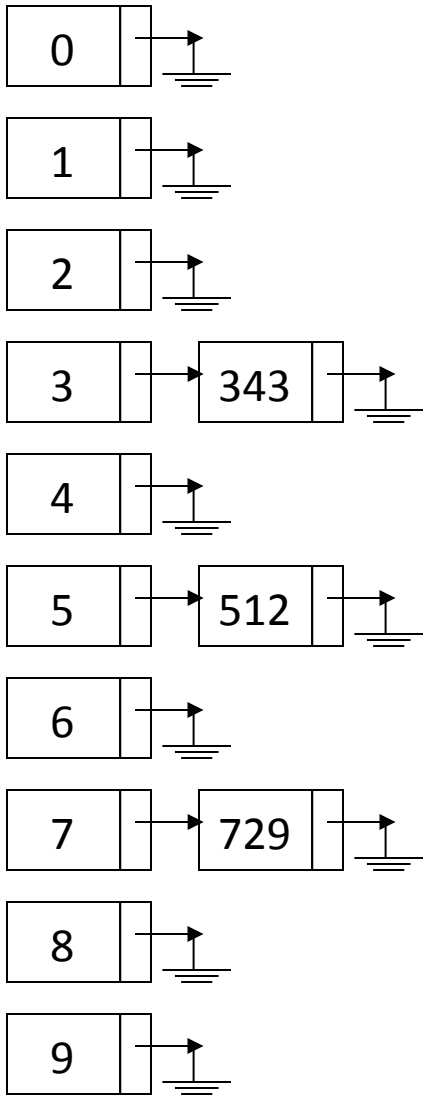
0	1	8	27	64	125				



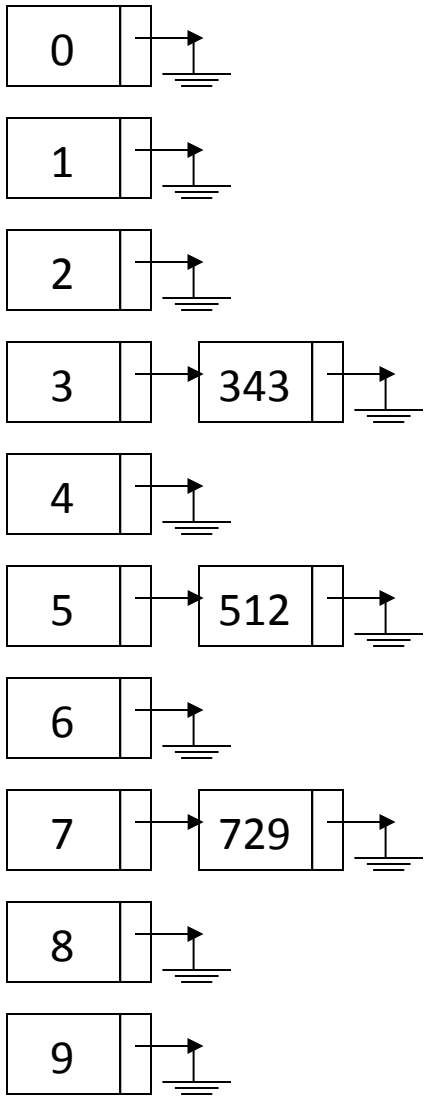
0	1	8	27	64	125	216			



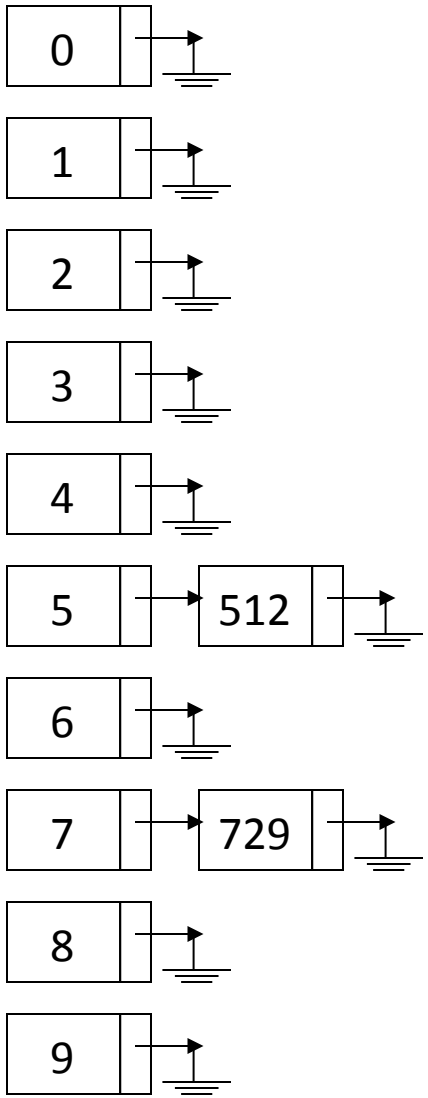
0	1	8	27	64	125	216			



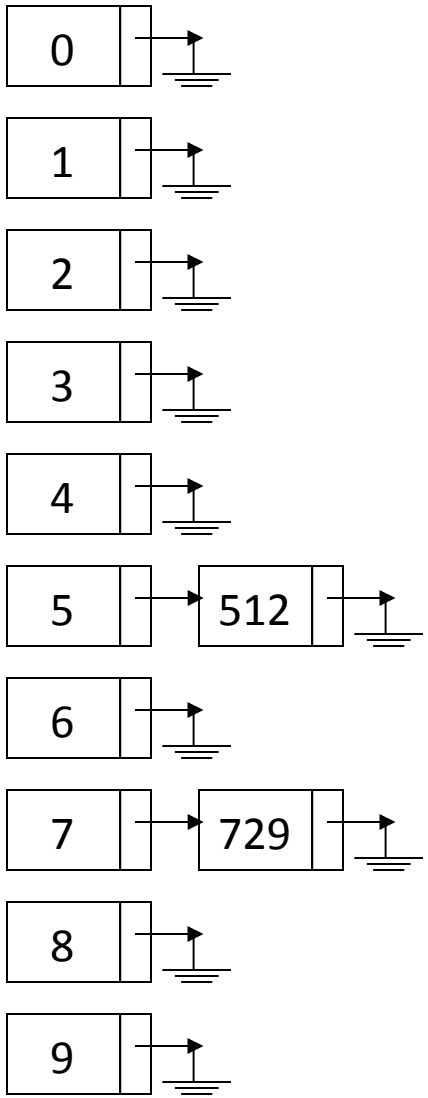
0	1	8	27	64	125	216	343		



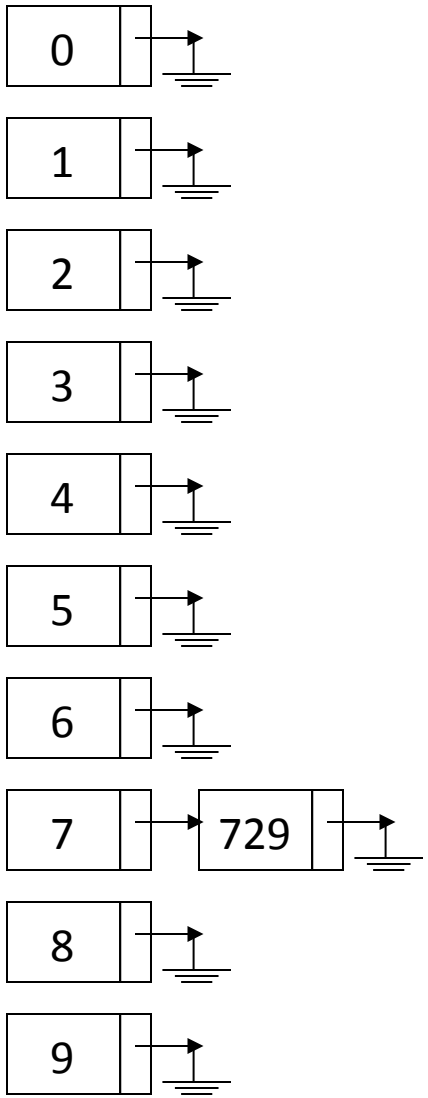
0	1	8	27	64	125	216	343		



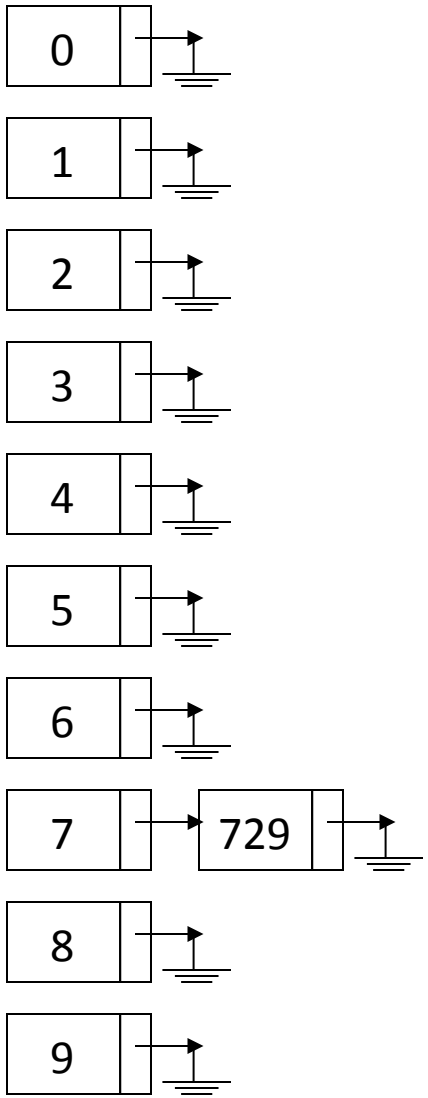
0	1	8	27	64	125	216	343	512	



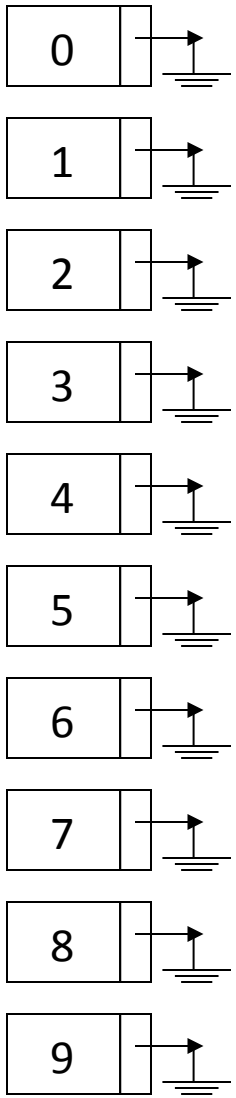
0	1	8	27	64	125	216	343	512	



0	1	8	27	64	125	216	343	512	729



0	1	8	27	64	125	216	343	512	729



What is the time Complexity?

- O(k n).
- O(k log n).
- O((k + n) log(k + n)).
- O(k + n).

Question:

Why use Radixsort?

Exercise, sort number using bucketsort

- 006, 081, 016, 572, 023, 999, 040, 101, 043, 425

Quick sort

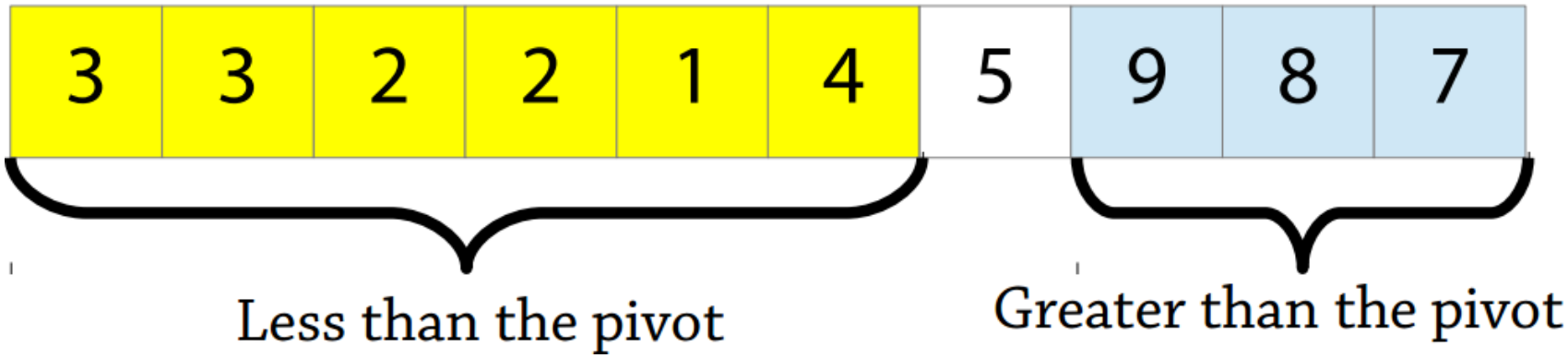
- Pick an element from the array, called the pivot and partition the array in two parts:
- First come all the elements smaller than the pivot, then the pivot, then all the elements greater than the pivot
- Recursively quicksort the two partitions

Quick sort

5	3	9	2	8	7	3	2	1	4

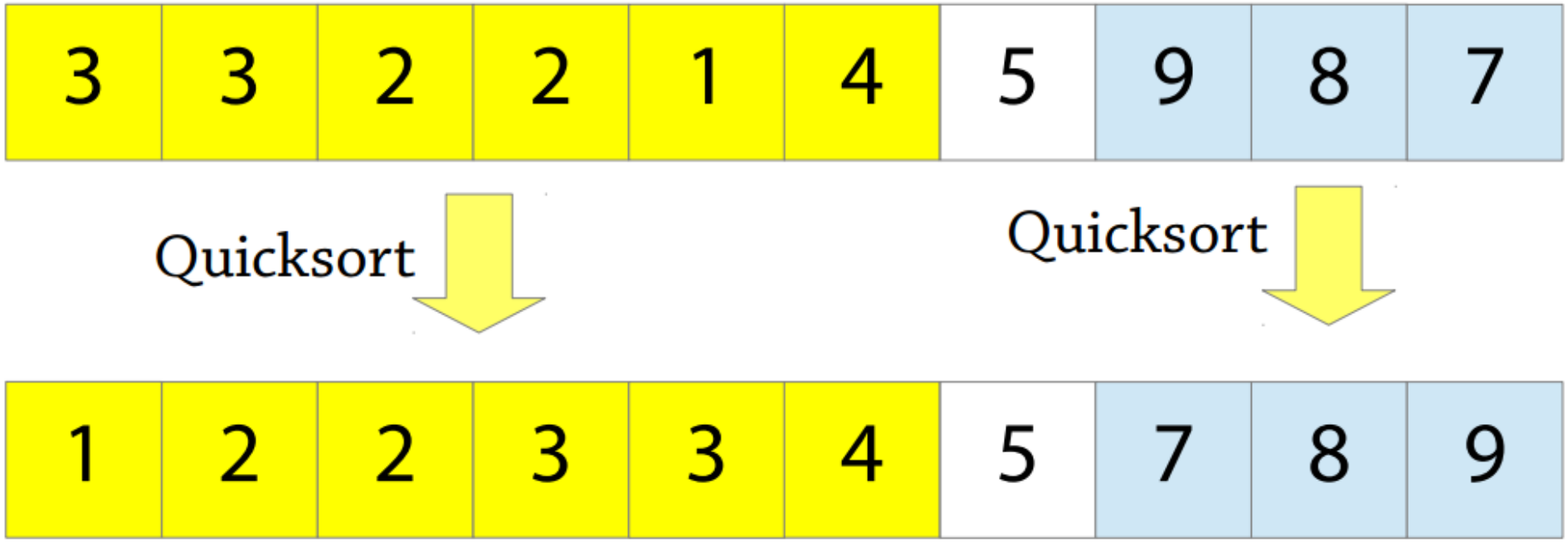
Say the pivot is 5.

Partition the array into: all elements less than 5, then all elements greater than 5



Quick sort

Now recursively quicksort the two partitions!



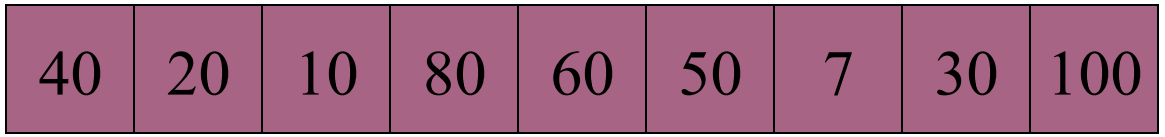
Quicksort Algorithm

Given an array of *n* elements (e.g., integers):

- If array only contains one element, return
- Else
  - pick one element to use as *pivot.*
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results

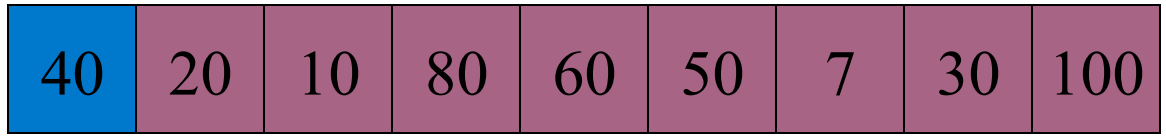
Example

We are given array of n integers to sort:



Pick Pivot Element

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:



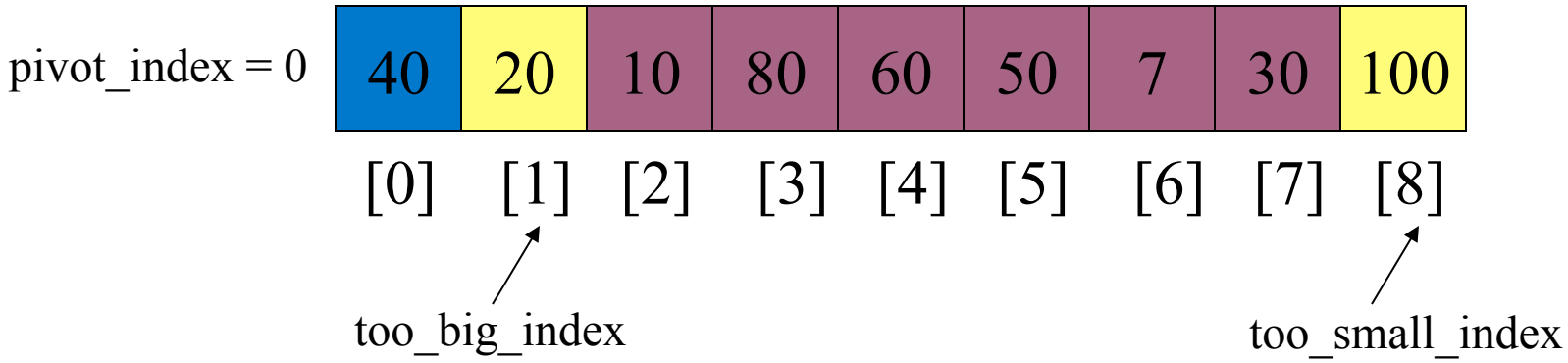
Partitioning Array

Given a pivot, partition the elements of the array such that the resulting array consists of:

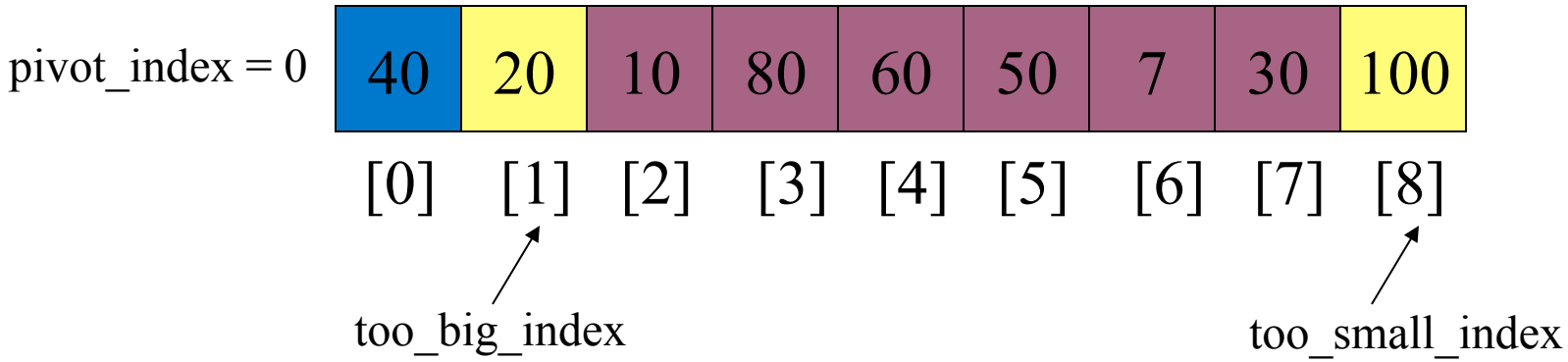
1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

The sub-arrays are stored in the original data array.

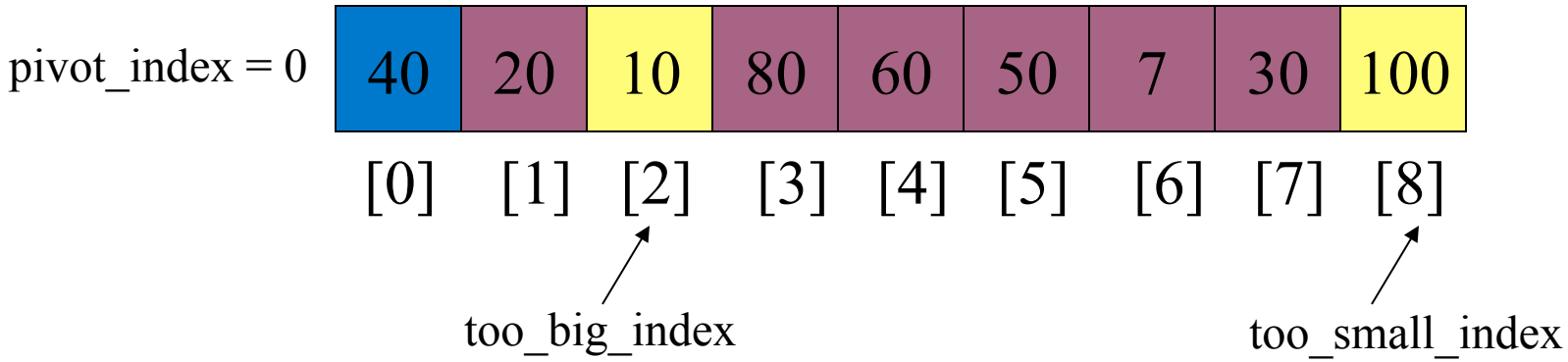
Partitioning loops through, swapping elements below/above pivot.


pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
too_big_index
too_small_index

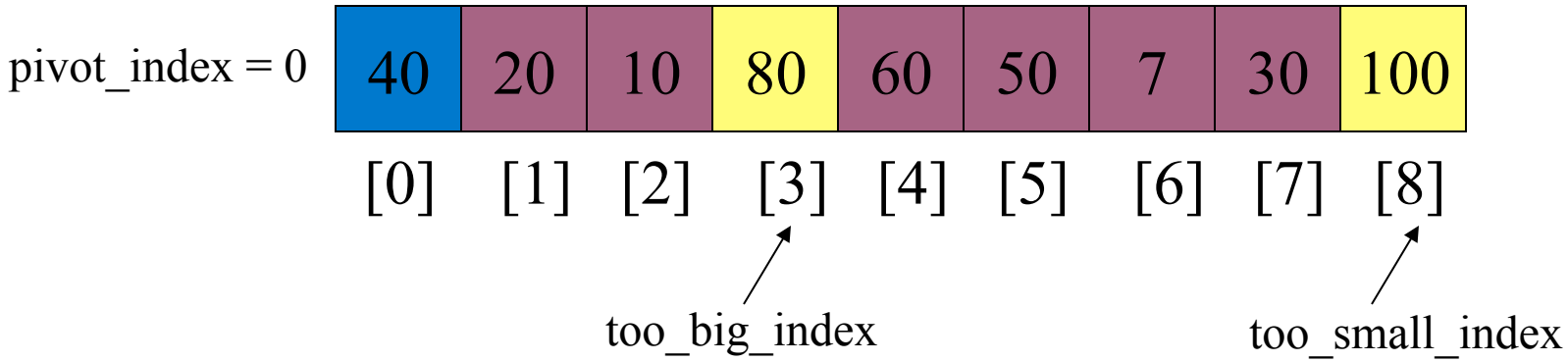
1. While data[too_big_index] <= data[pivot]
   ++too_big_index



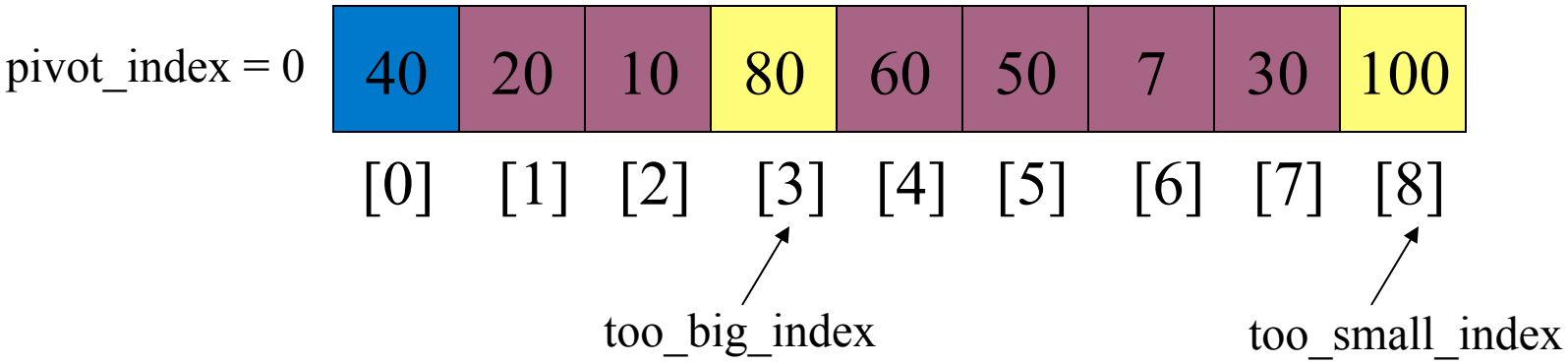
1. While data[too_big_index] <= data[pivot]
   ++too_big_index



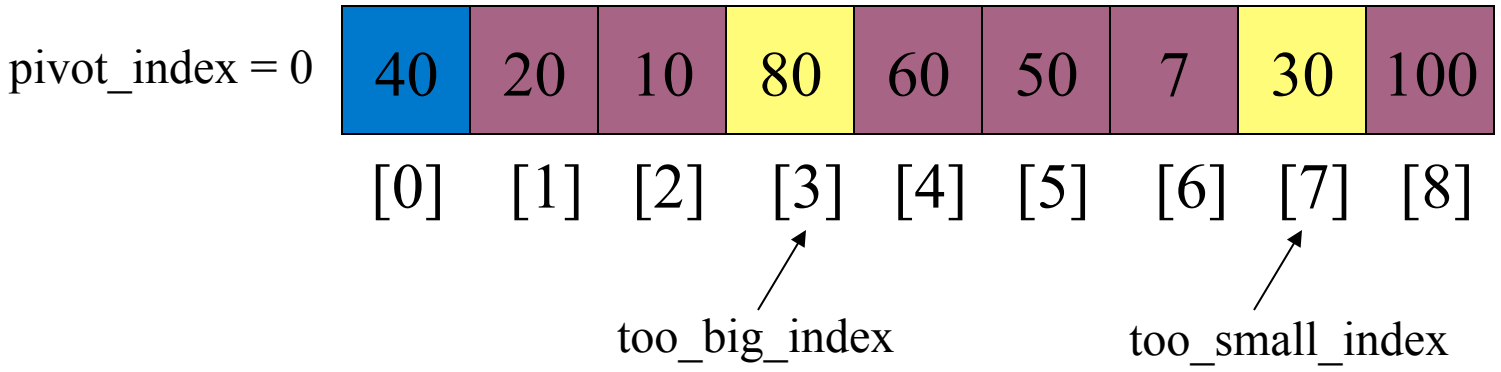
1. While data[too_big_index] <= data[pivot]
   ++too_big_index



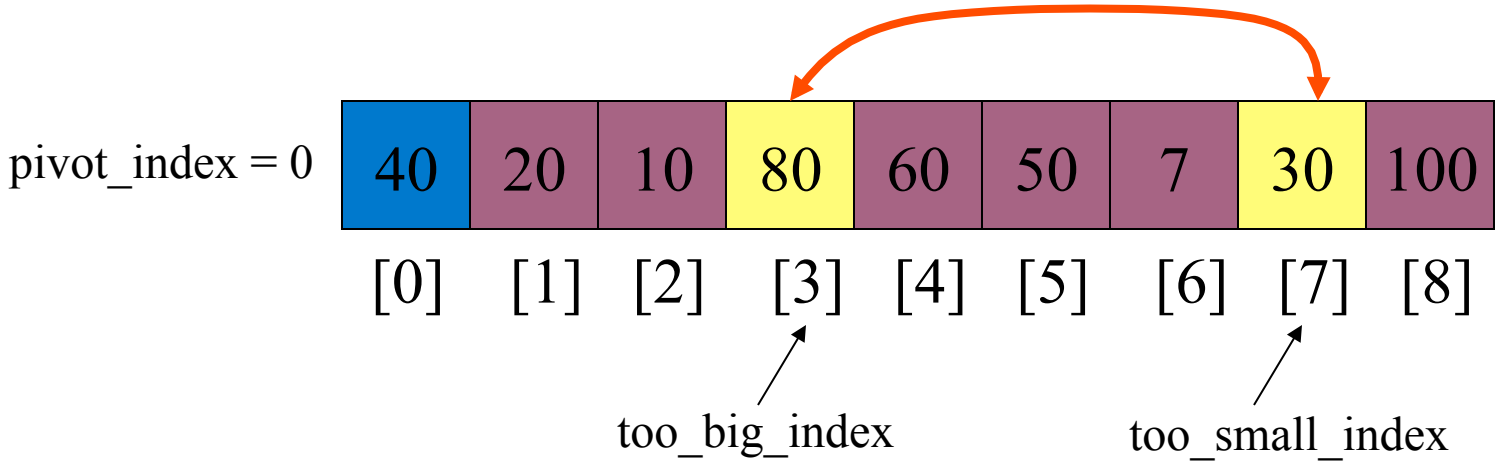
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index



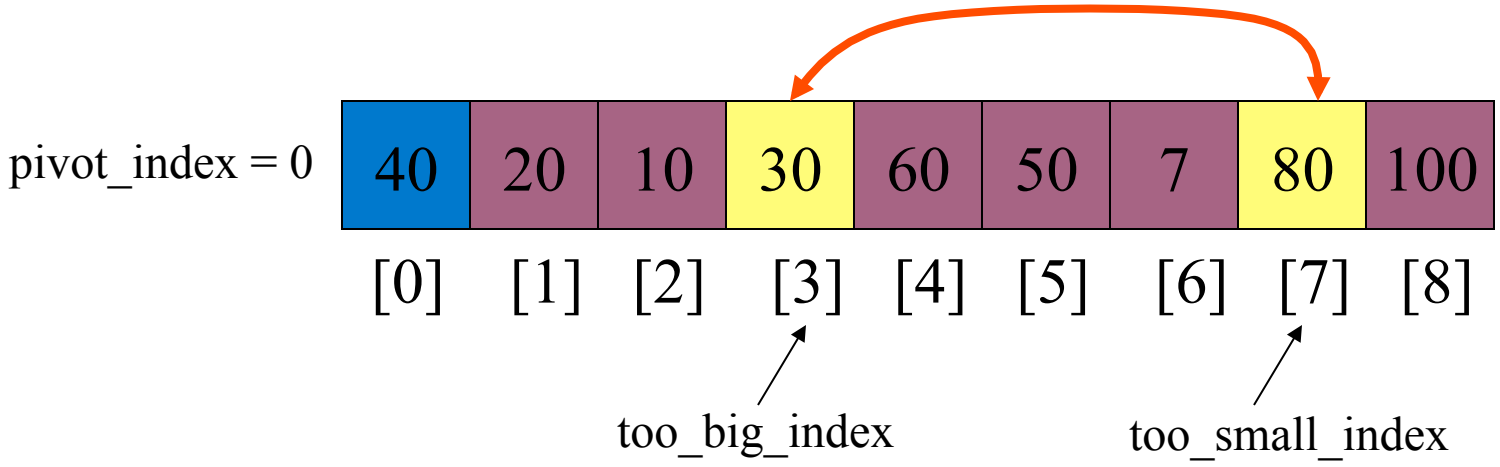
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index



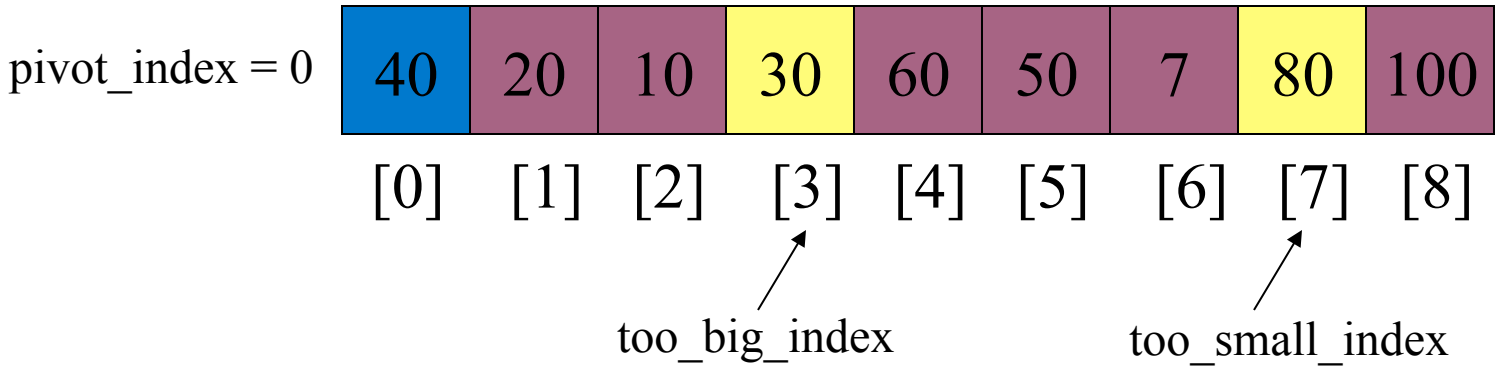
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



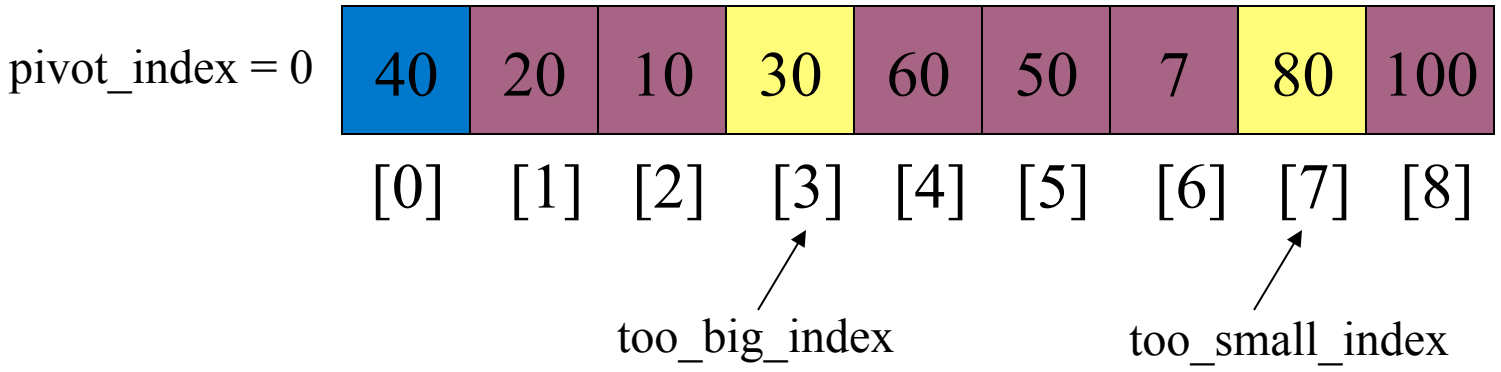
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



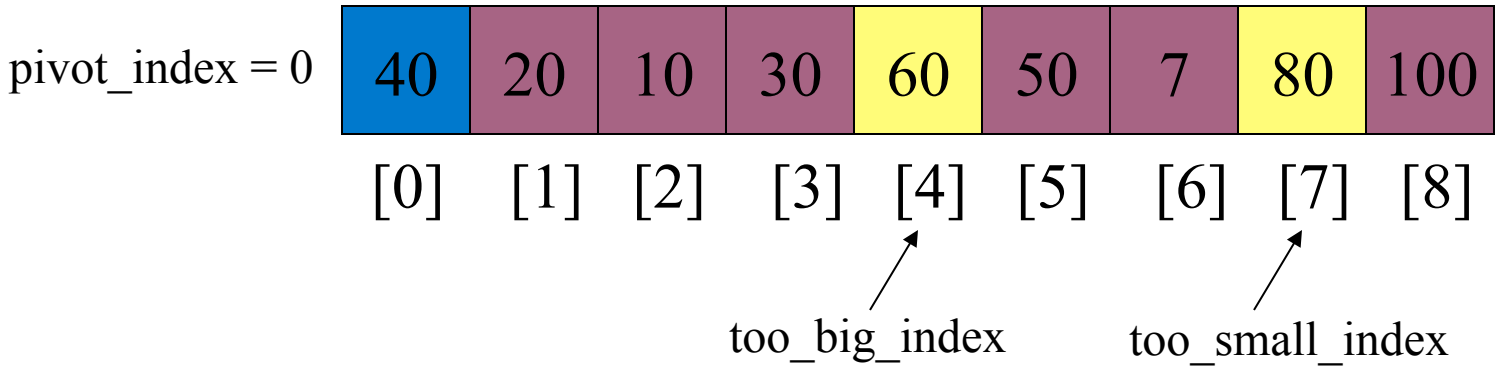
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, repeat 1.



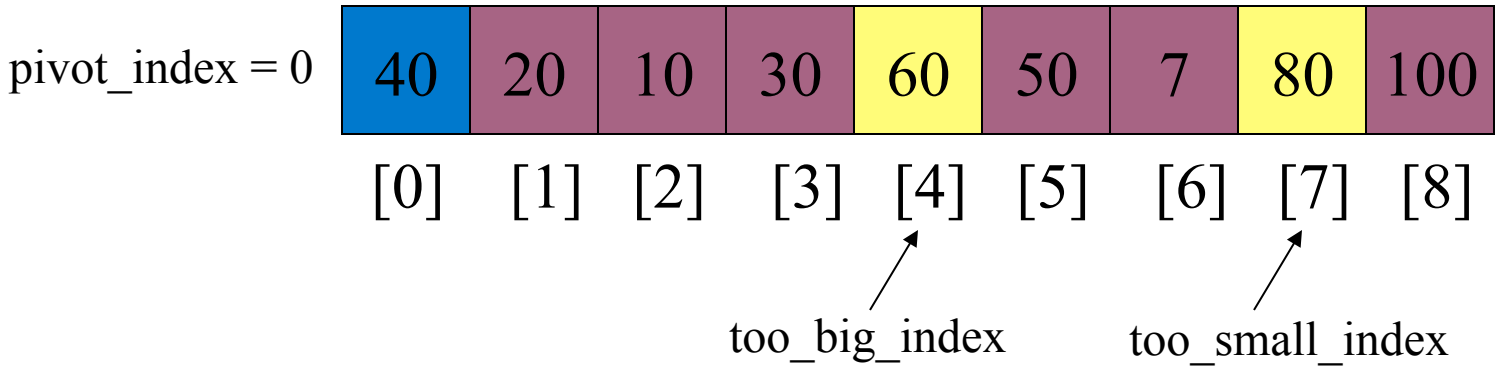
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, repeat1.



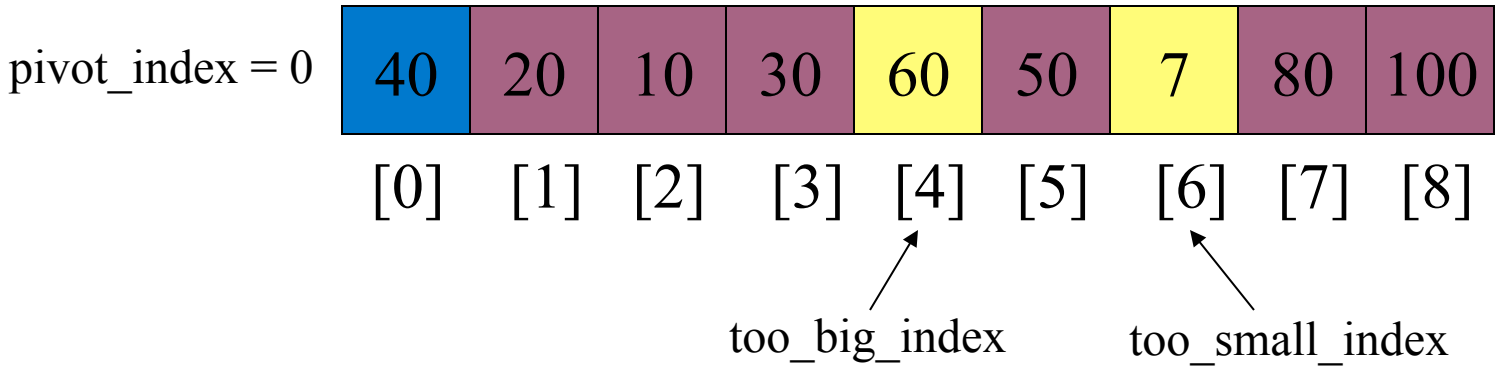
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



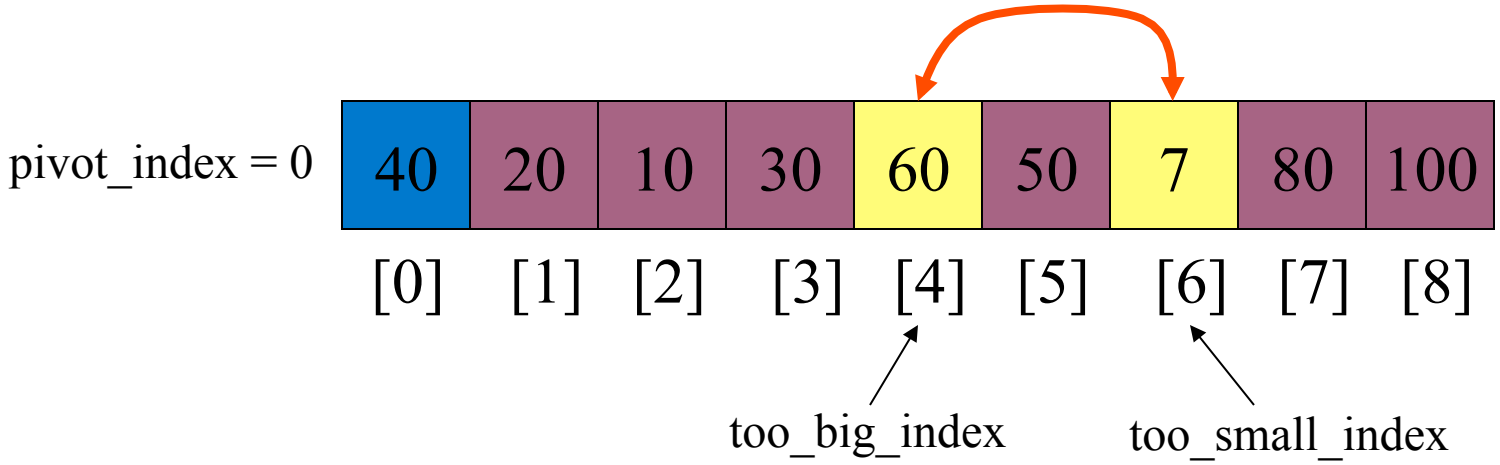
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



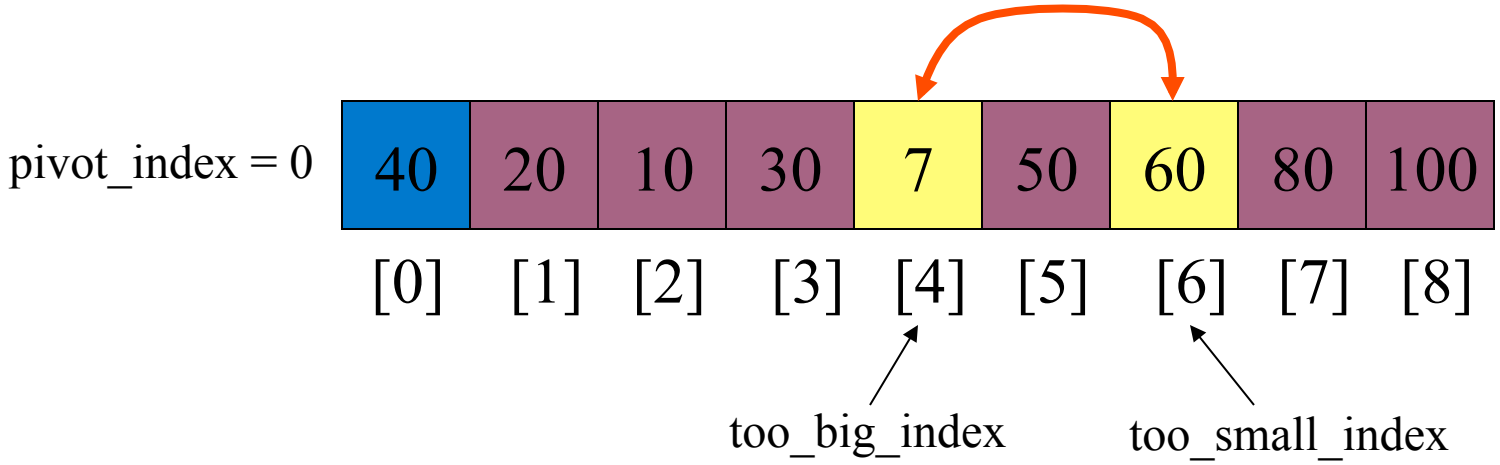
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



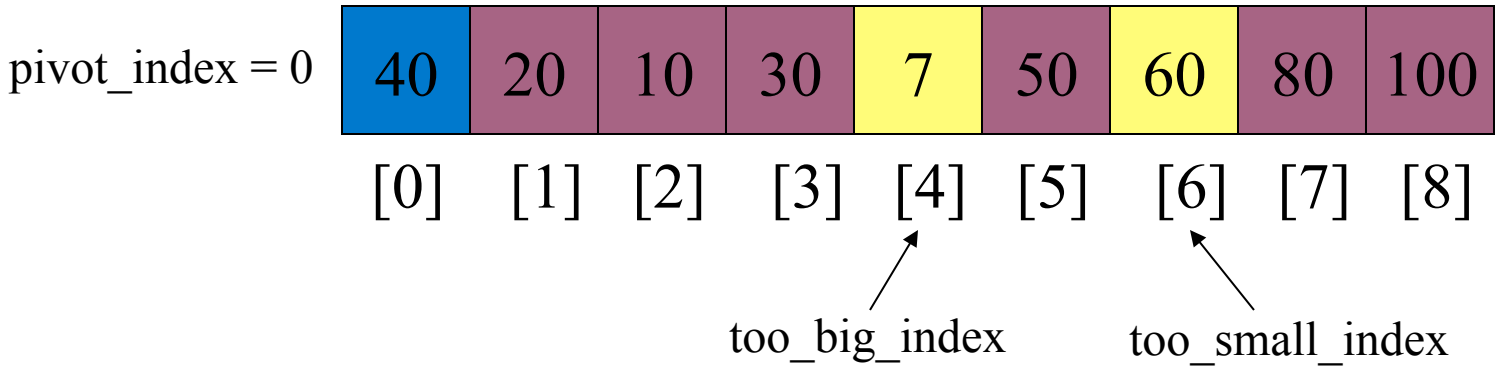
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



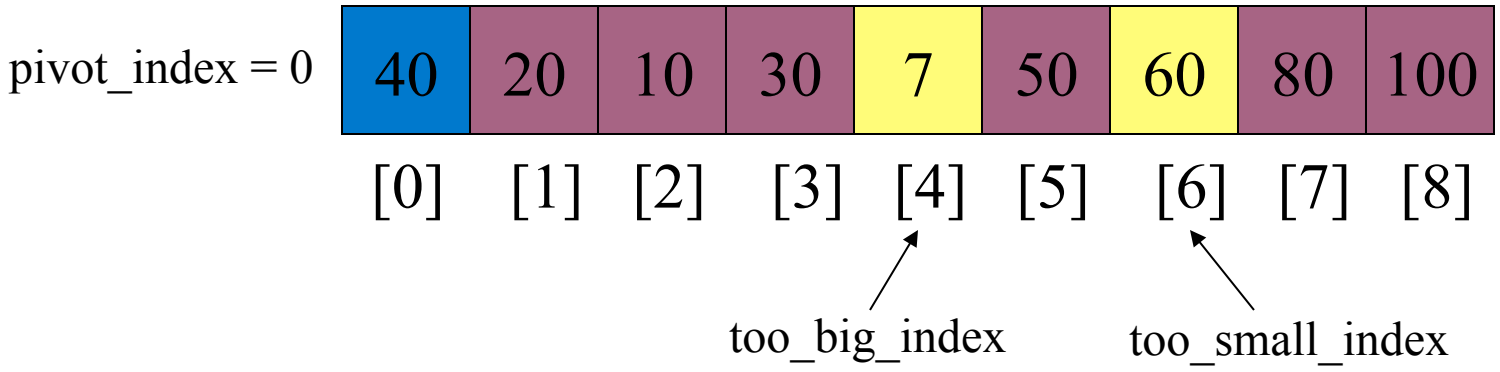
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



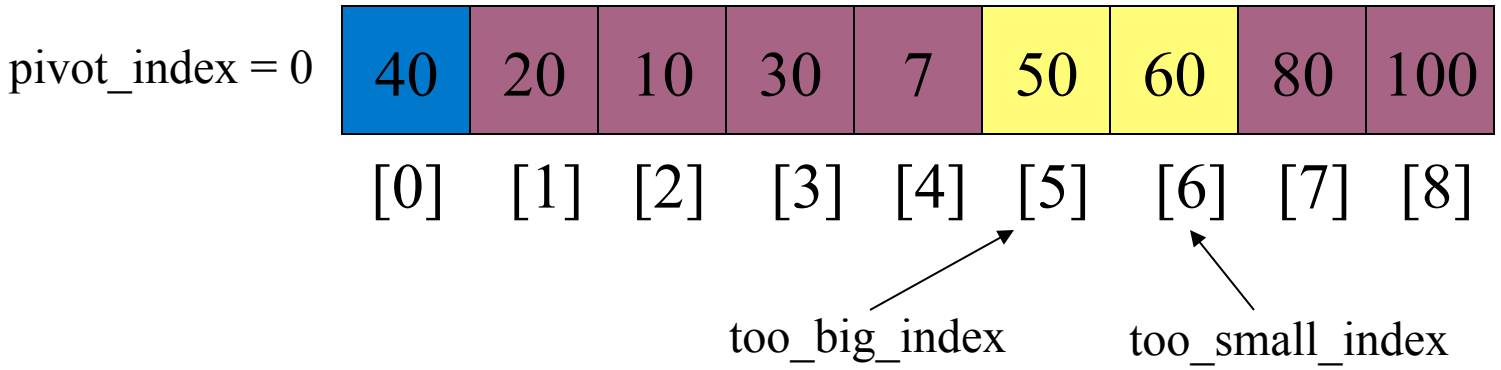
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



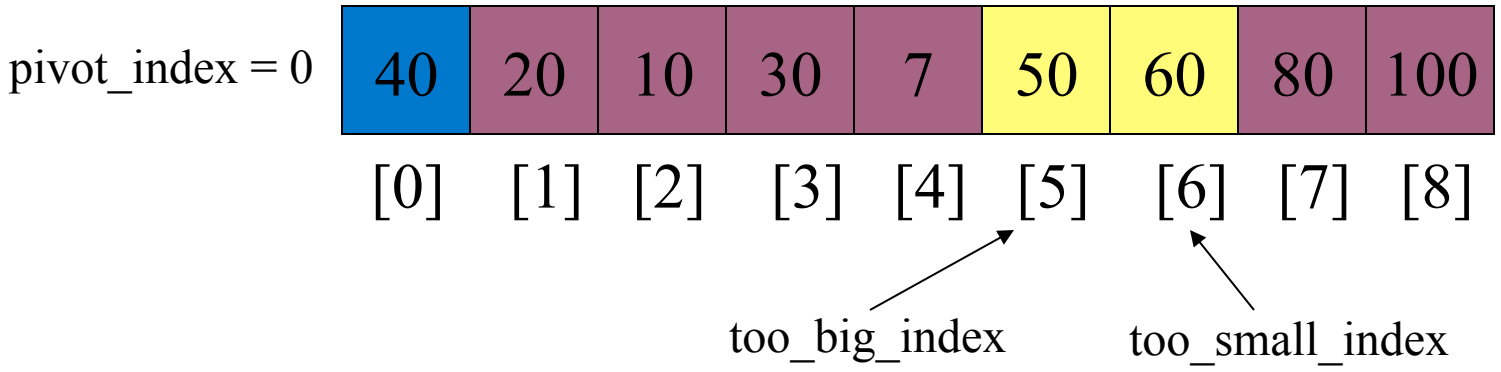
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



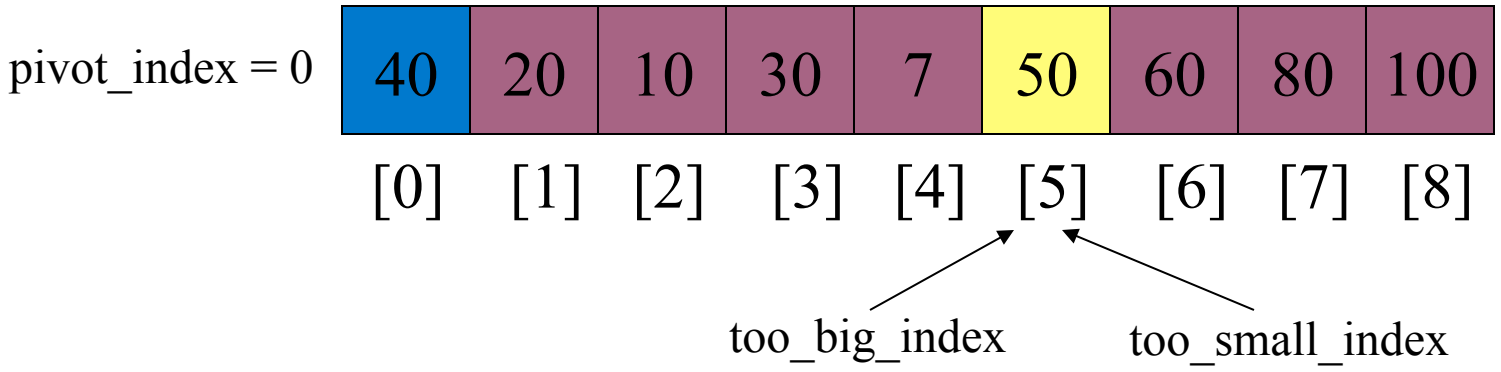
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



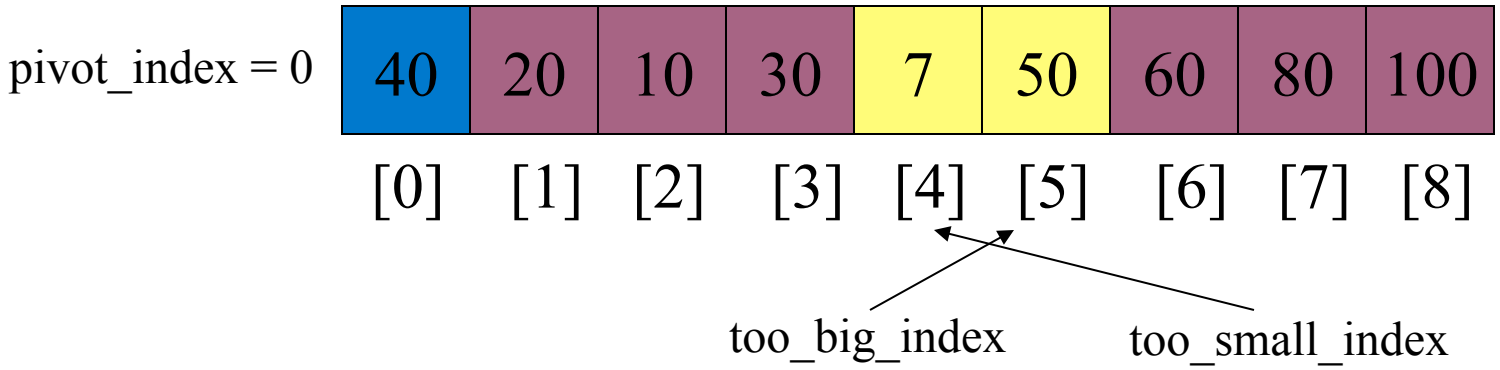
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



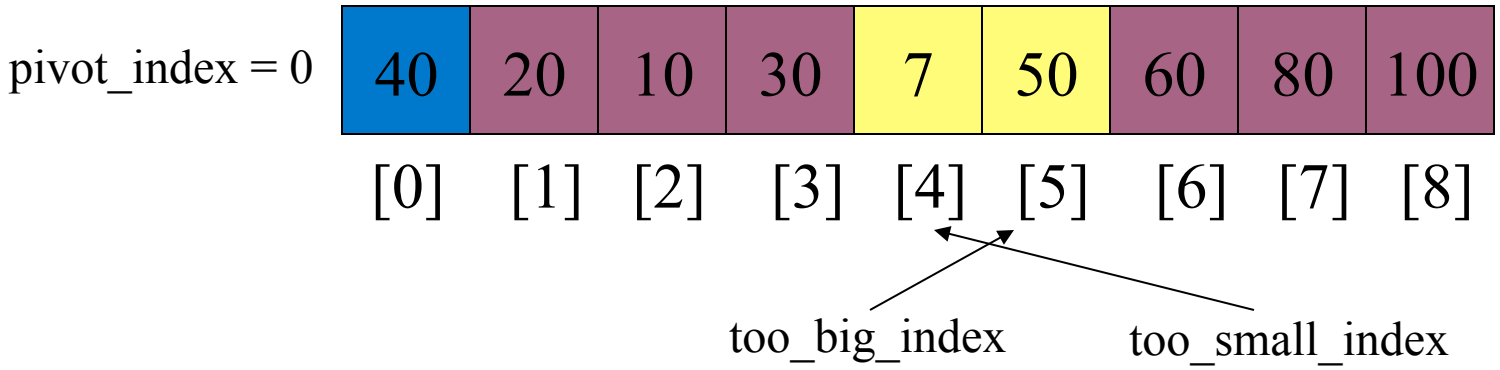
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



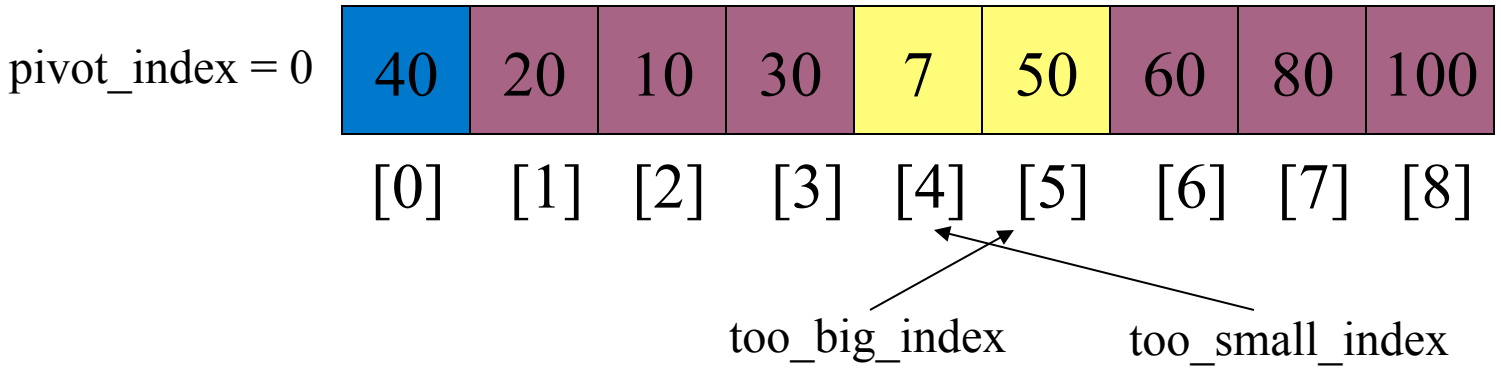
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



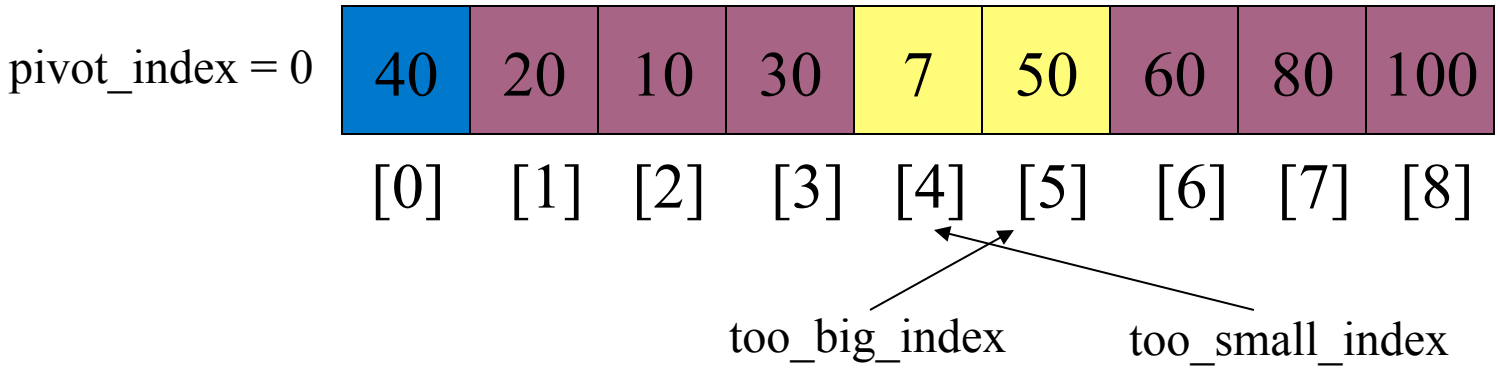
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



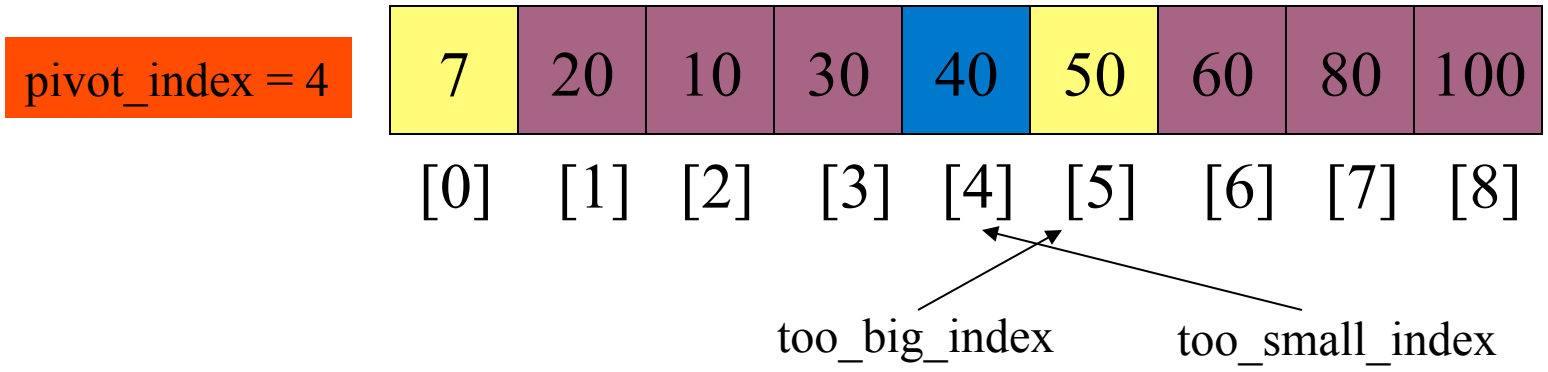
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



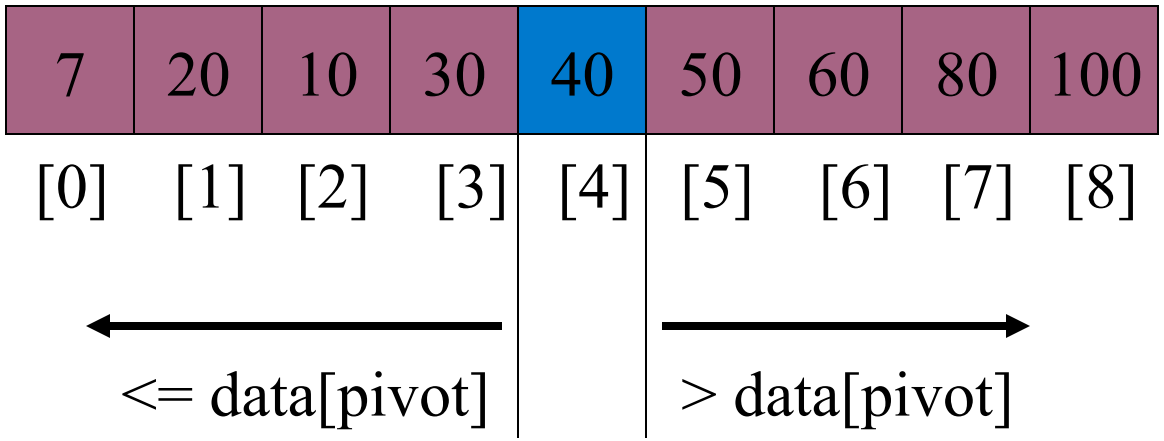
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



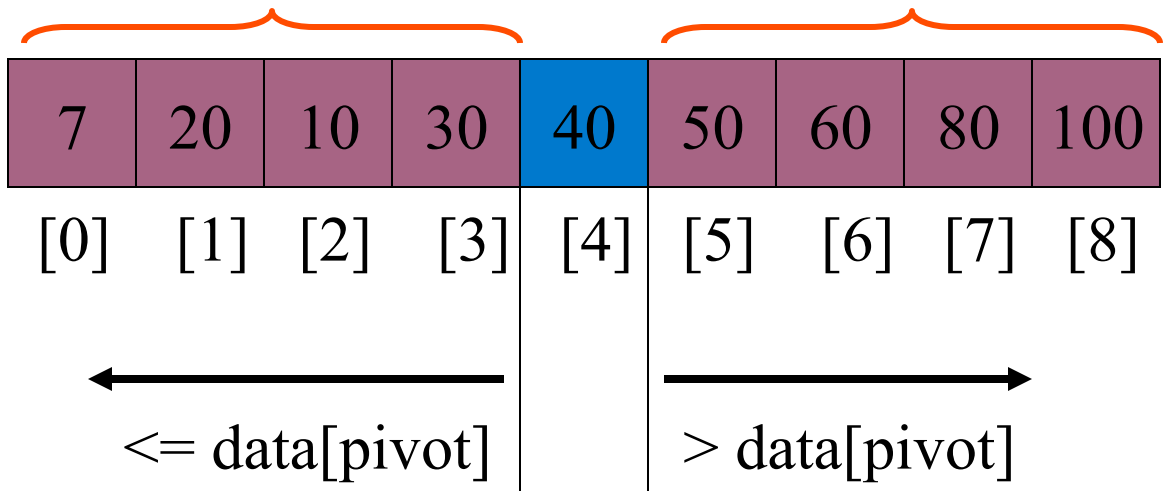
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



Partition Result



Recursion: Quicksort Sub-arrays



Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$
  - Number of accesses in partition?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$
  - Number of accesses in partition? O(n)

Quicksort Analysis

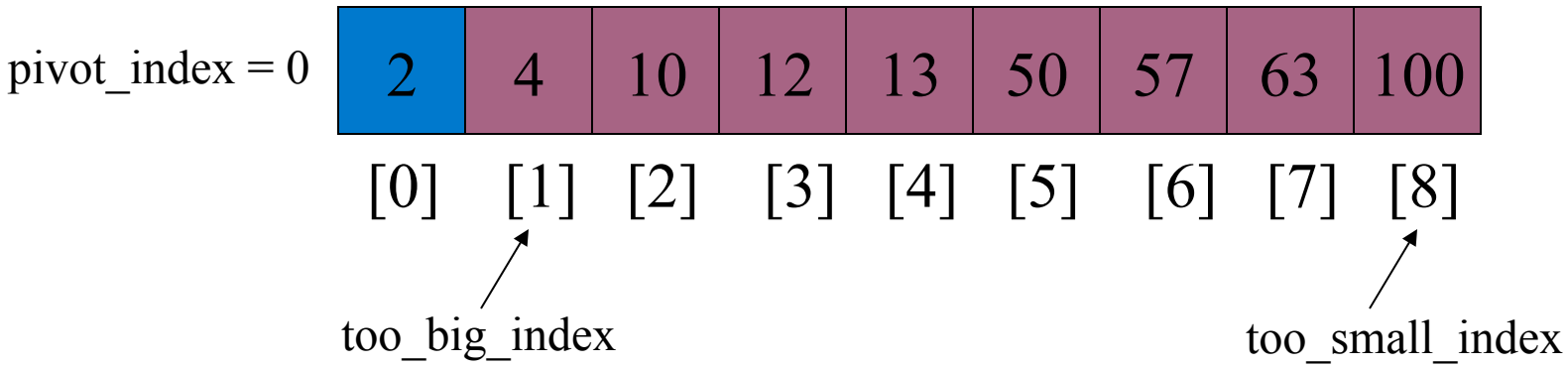
- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$

Quicksort Analysis

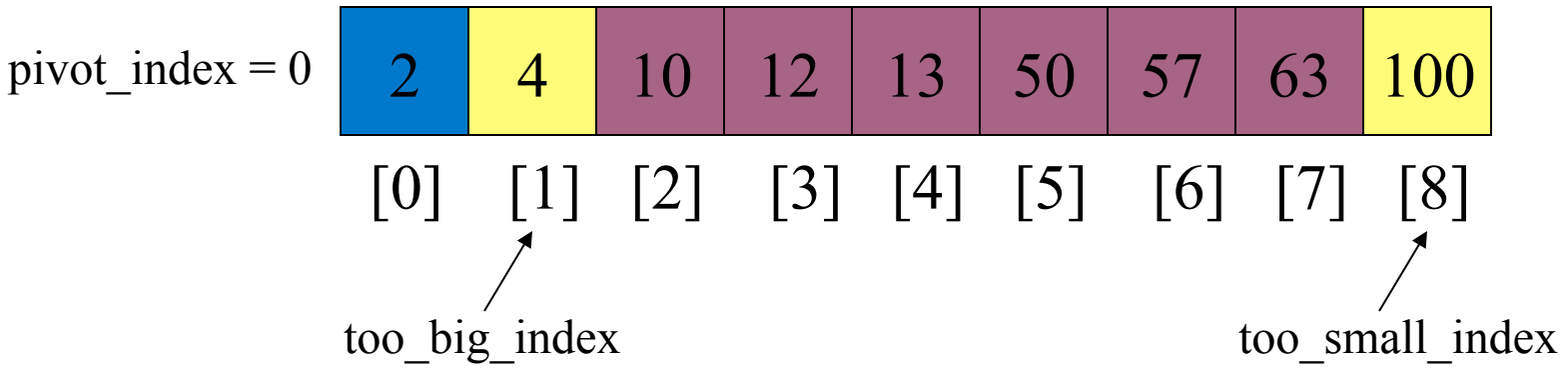
- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?

Quicksort: Worst Case

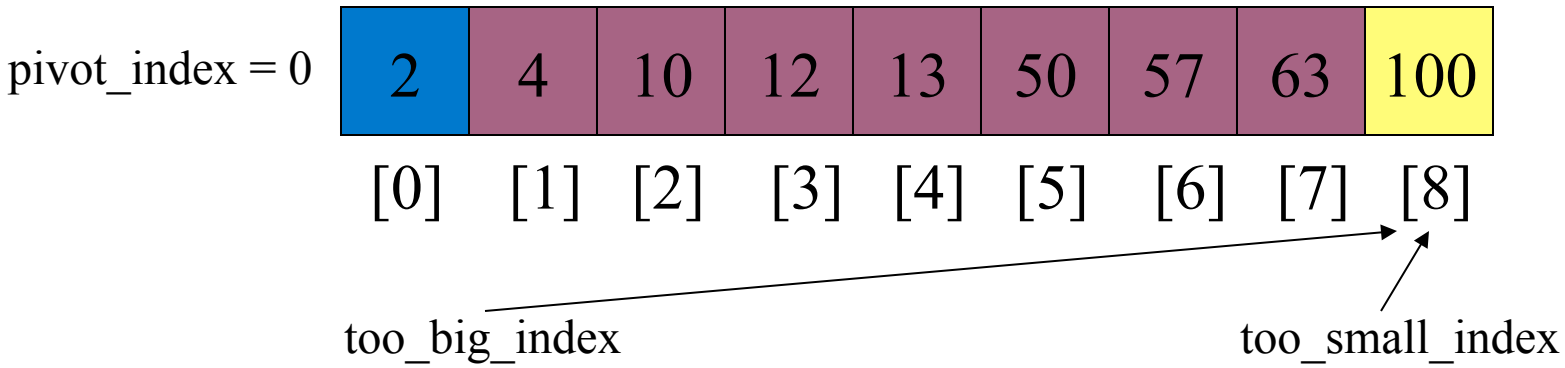
- Assume first element is chosen as pivot.
- Assume we get array that is already in order:



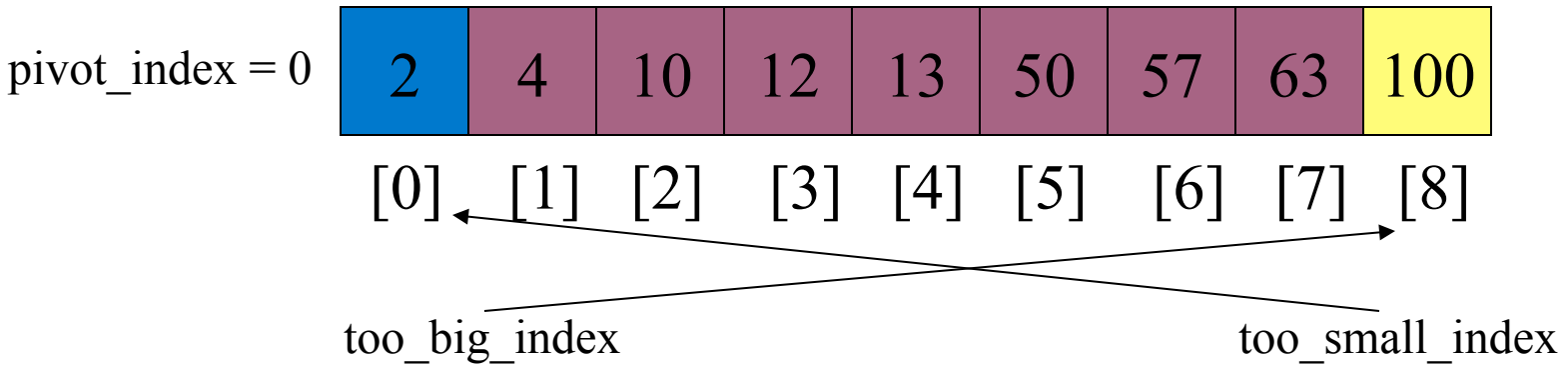
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



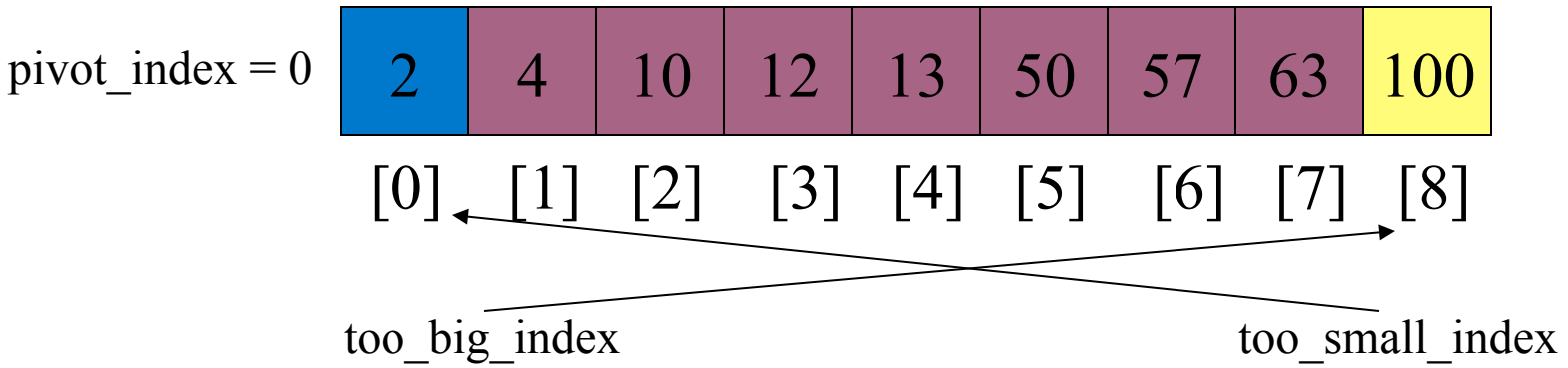
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



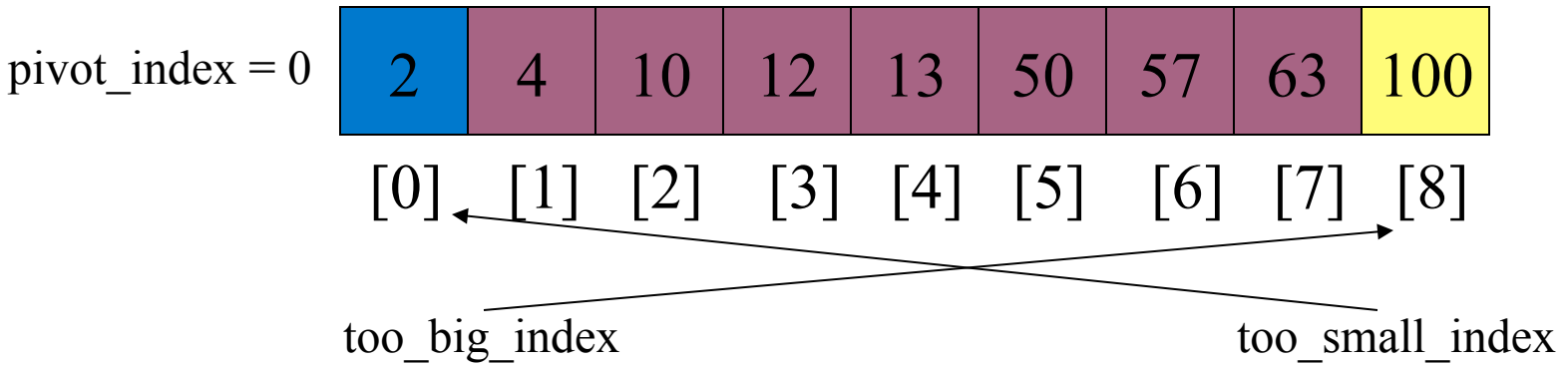
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



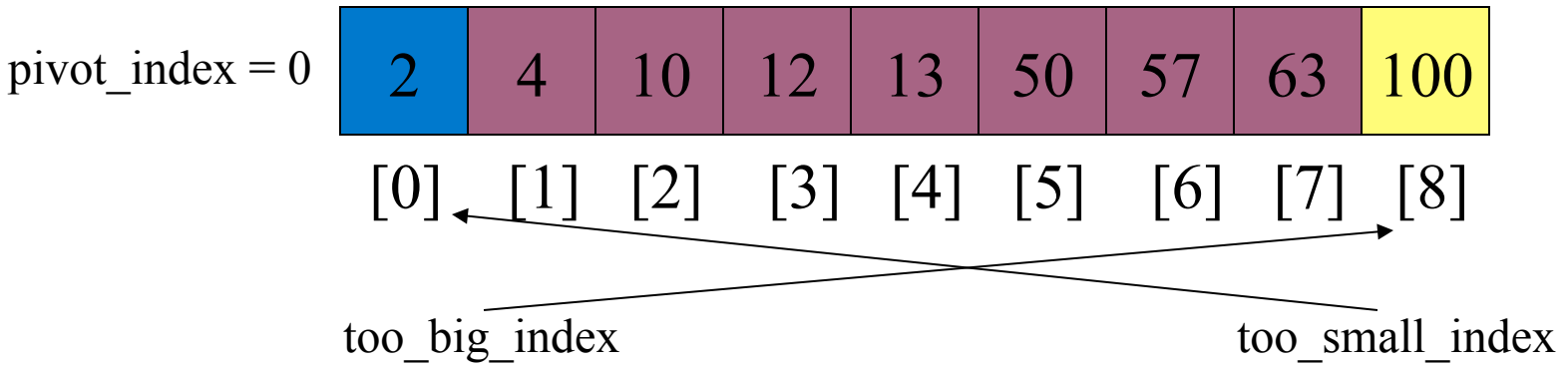
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



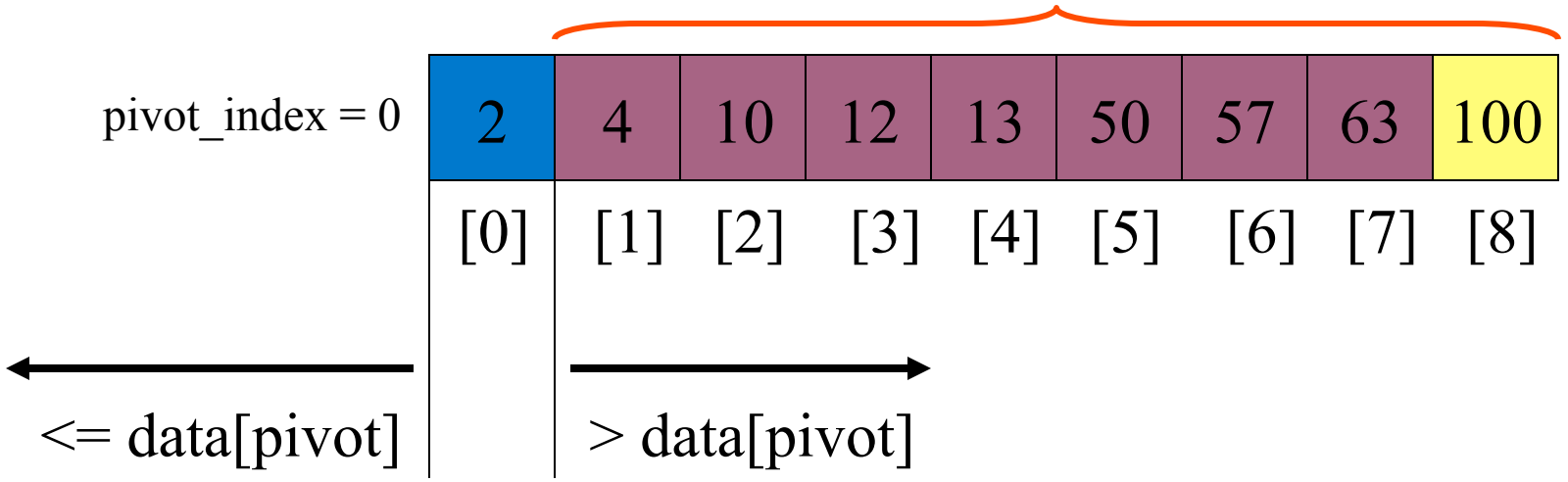
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)
  - Number of accesses per partition?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)
  - Number of accesses per partition? O(n)

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!
- What can we do to avoid worst case?

Improved Pivot Selection

Pick median value of three elements from data array:

data[0], data[n/2], and data[n-1].

Use this median value as pivot.

Question:

- Heapsort has a worst case of O(n log n)
- Quick sort has $O(n^2)$

Should’nt it be better to use Heapsort over Quicksort?

Answer:

- One of the major factors is that quicksort has better *locality of reference*. Statisticly the next data to be accessed is usually close in memory to the data you just looked at. Data that has god locality is more likely be cached together, and therefore quicksort tends to be faster.
- On the other hand, if you want to perform good even in worst case then heapsort is a better choice. This give a better control of keeping critical deadlines.