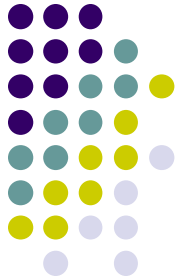
Queues

Data Structures

Dr. Gurpreet Singh Lehal,
Department of Computer Science,
Punjabi University

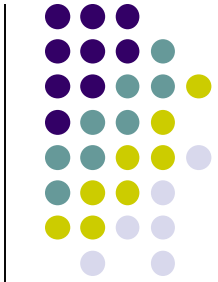
DEFINITION OF QUEUE



- A **Queue** is an ordered collection of items from which items may be deleted at one end (called the ***front*** of the queue) and into which items may be inserted at the other end (the ***rear*** of the queue).
- The first element inserted into the queue is the first element to be removed. For this reason a queue is sometimes called a ***FIFO*** (first-in first-out) list as opposed to the stack, which is a ***LIFO*** (last-in first-out).

Queues

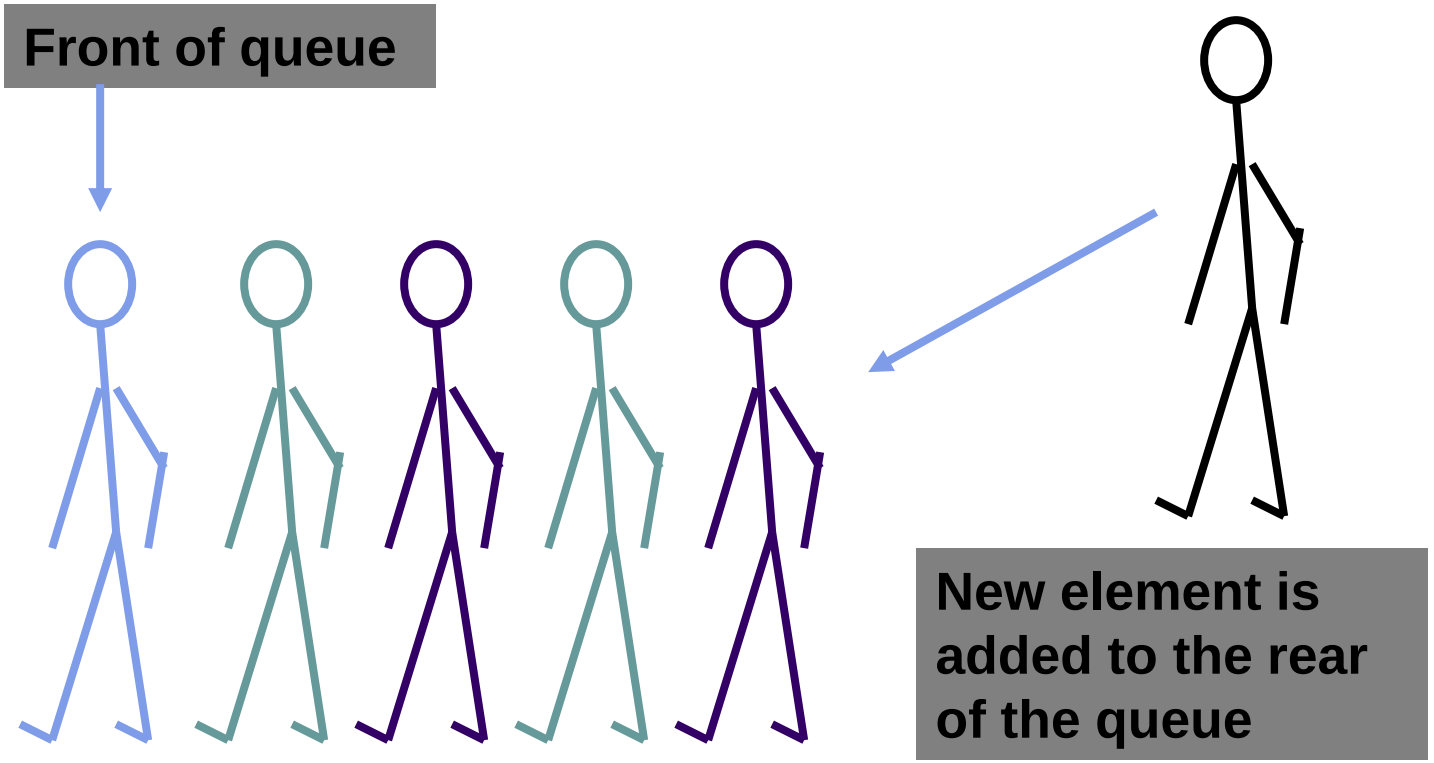
Basic Queue Operations



- There are two basic operations that can be performed on a queue:
- **Enqueue:** This operation inserts or adds data in the queue. The data will be added to the rear of the queue.
- **Dequeue:** This operation removes the data, which has been in the queue for the longest time. The data is present in the front of the queue.

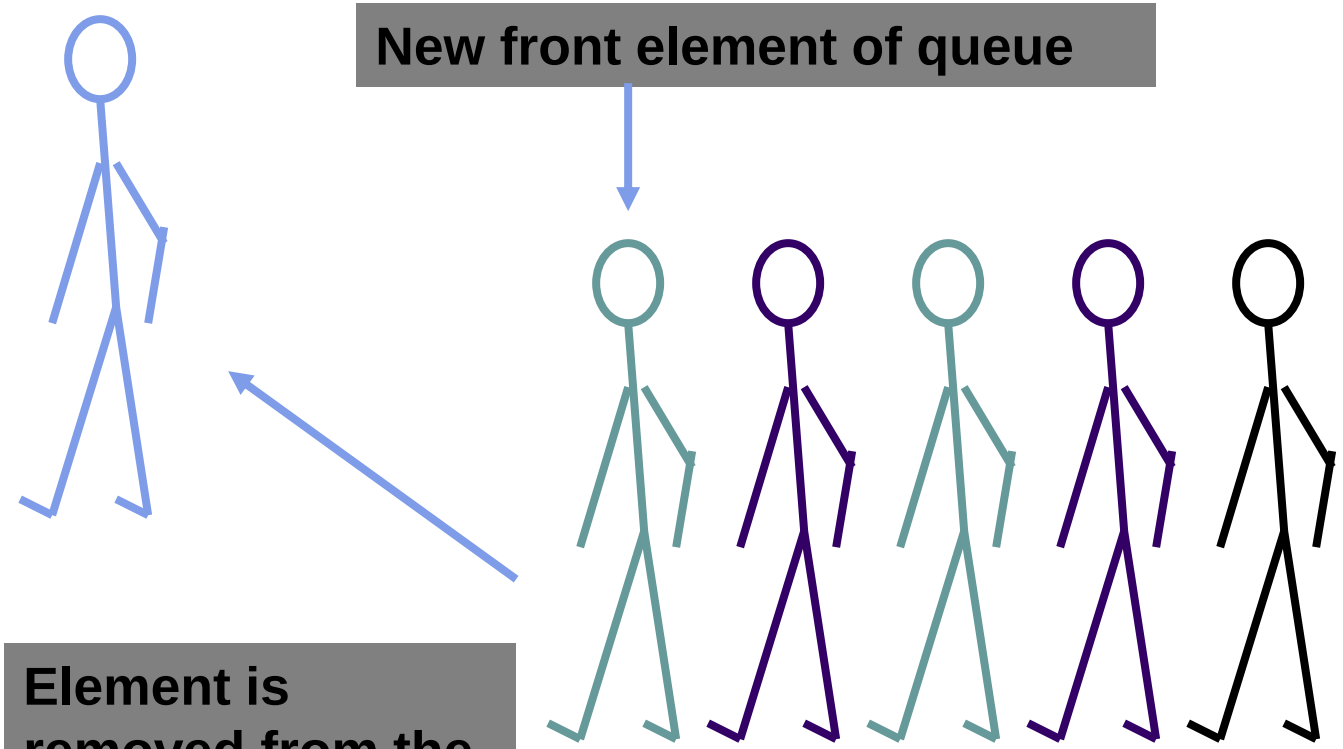
Enqueue

Adding an element

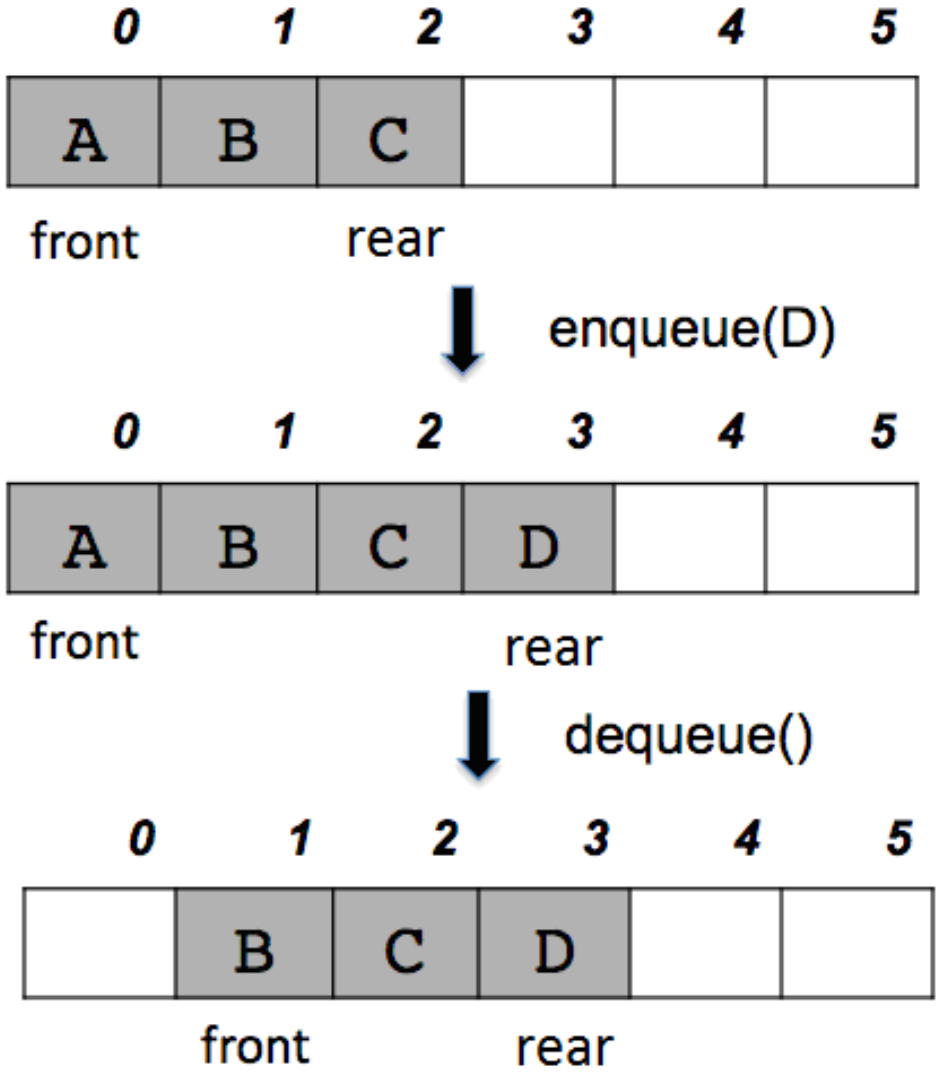


Dequeue

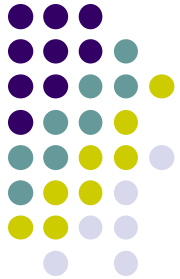
Removing an element



Basic Queue Operations



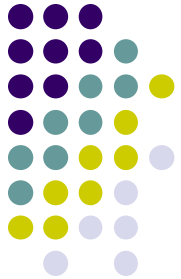
Implementation of the Queues



There are many possible ways to implement the queues:

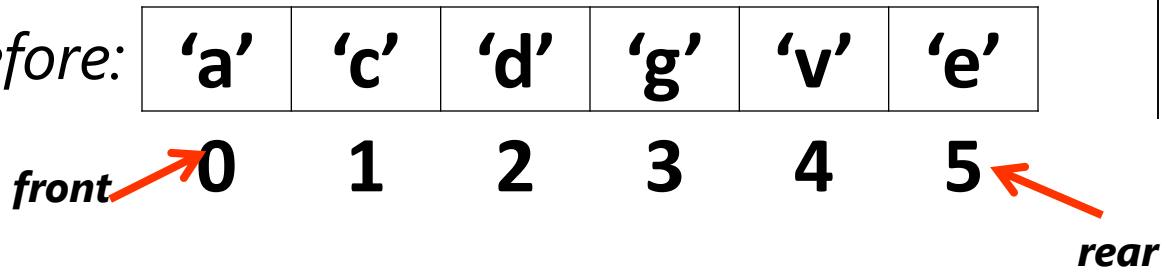
- An array with front at index 0
- An array with floating front and rear
- A circular array with floating front and rear (wrap around array)
- A singly linked list with front
- A singly linked list with front and rear
- A doubly linked list

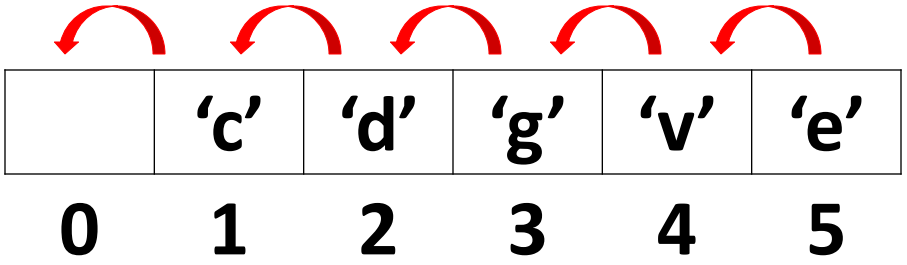
Array Implementation of the Queues



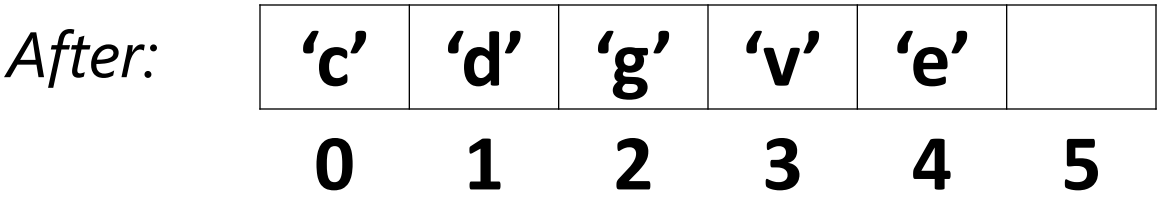
- We must keep track of both the ***front*** and the ***rear*** of the queue. One method is to keep the ***front*** of the array in the first location on the array. Then, we can simply increase the counter of the array to show ***the rear***.
- Nevertheless, to **delete an entry from this queue is very expensive**, since after the first entry was served, all the existing entry need to be move back one position to fill in the vacancy.
- With a long queue this process can lead to poor performance.

Array with front at index 0

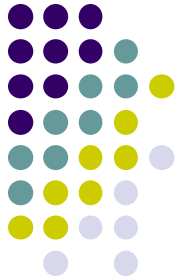




'a' at index 0 is deleted



Linear Implementation of Queues

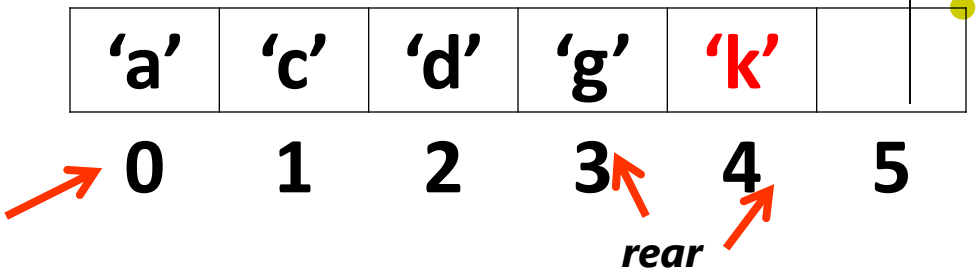


- *Indicate the front and rear of the queue*. We can keep track the entry of the queue without moving any entries.
  - Append an entry: increase the rear by one.
  - To remove the entry: increase the front by one.

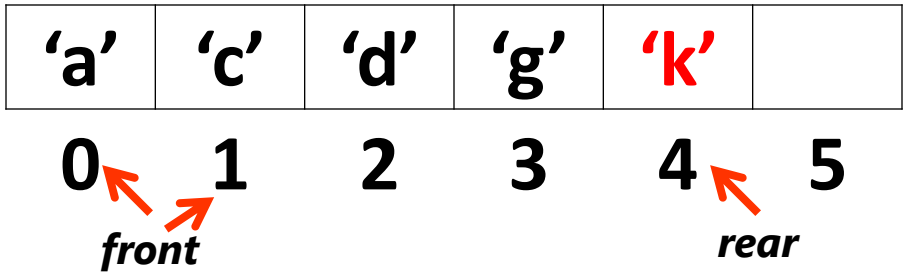
- *Problem:*
  - Position of front will increase and never decrease
  - This leads to the end of the storage capacity

Linear Implementation of Queues

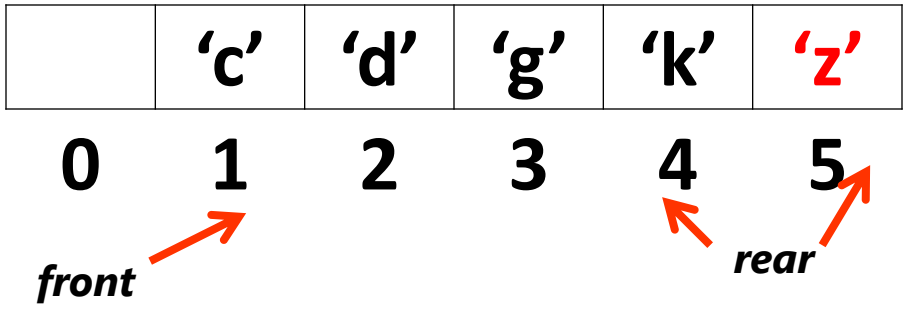
Add 'k' to the queue:
- rear + 1 (increase)



Delete 'a' in queue:
- front + 1 (increase)

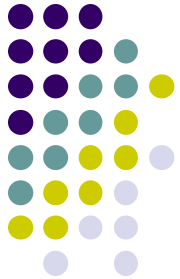


Add 'z' to the queue:
-rear + 1 (increase)



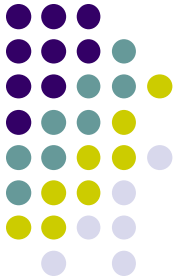
Reach end of storage!!

Need for Circular Queues



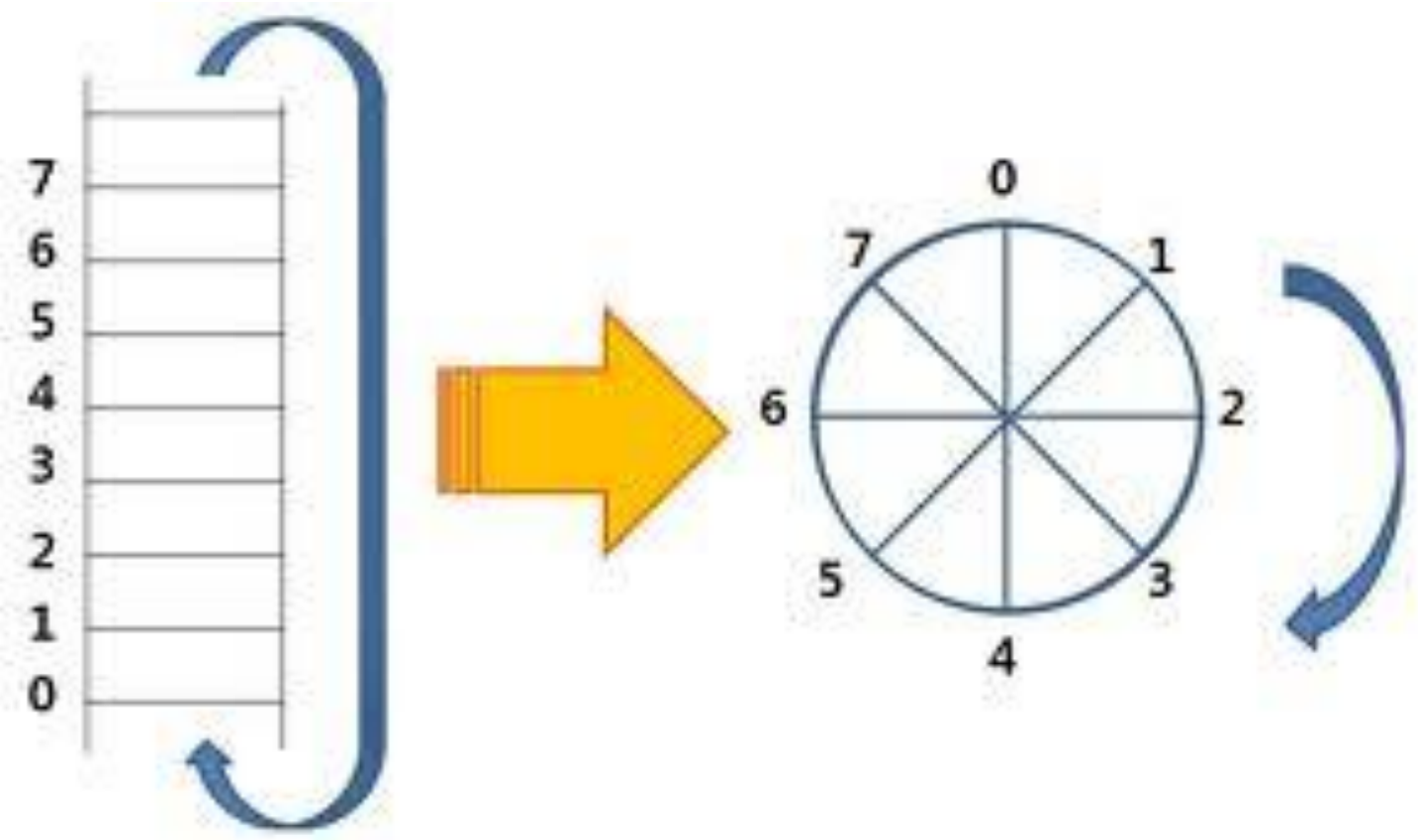
- Queues implemented as linear arrays have the drawback that once the queue is FULL, even though we delete few elements from the "front" and relieve some occupied space, we are not able to add anymore elements, as the "rear" has already reached the Queue's rear most position.

Need for Circular Queues

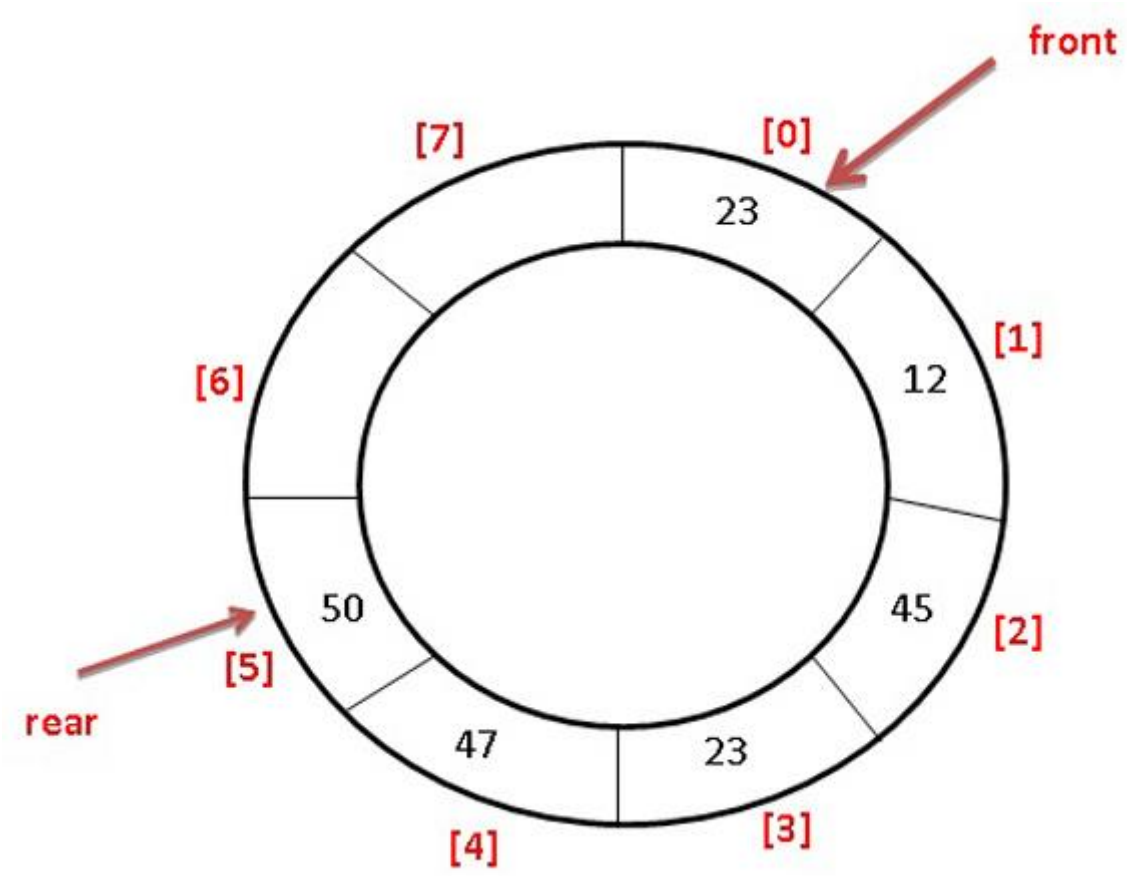


- To solve this problem, queues implement wrapping around. Such queues are called *Circular Queues*.
- Both the front and the rear pointers wrap around to the beginning of the array.
- Circular queues are better than normal queues as they effectively utilise the memory space.

Circular Queues

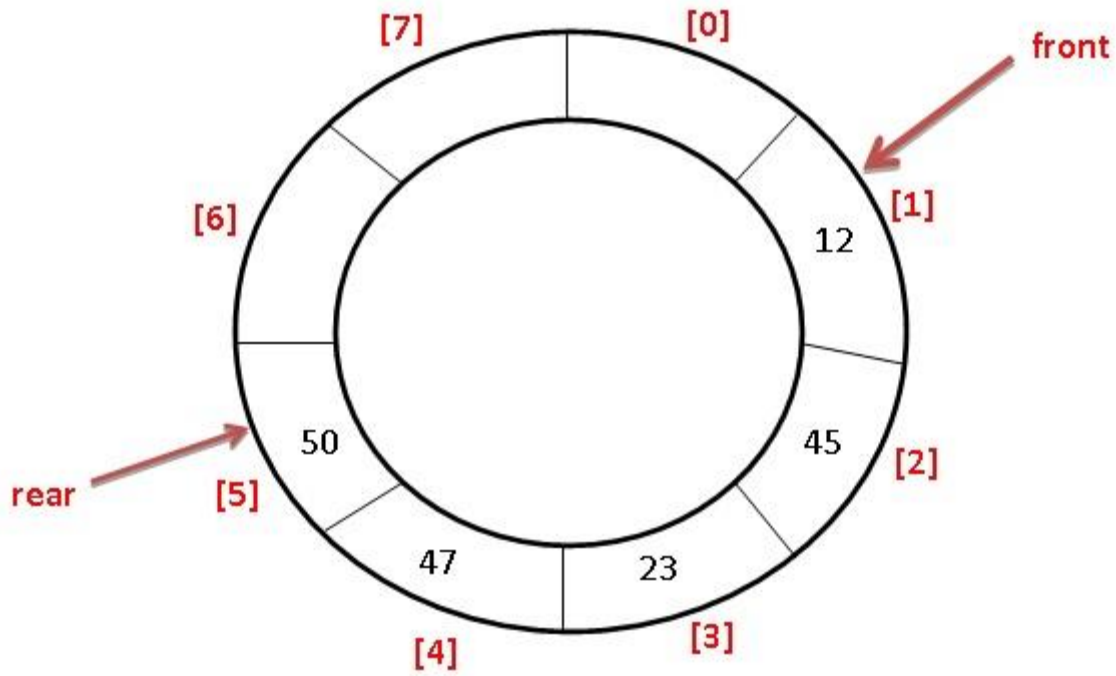


Circular Queues



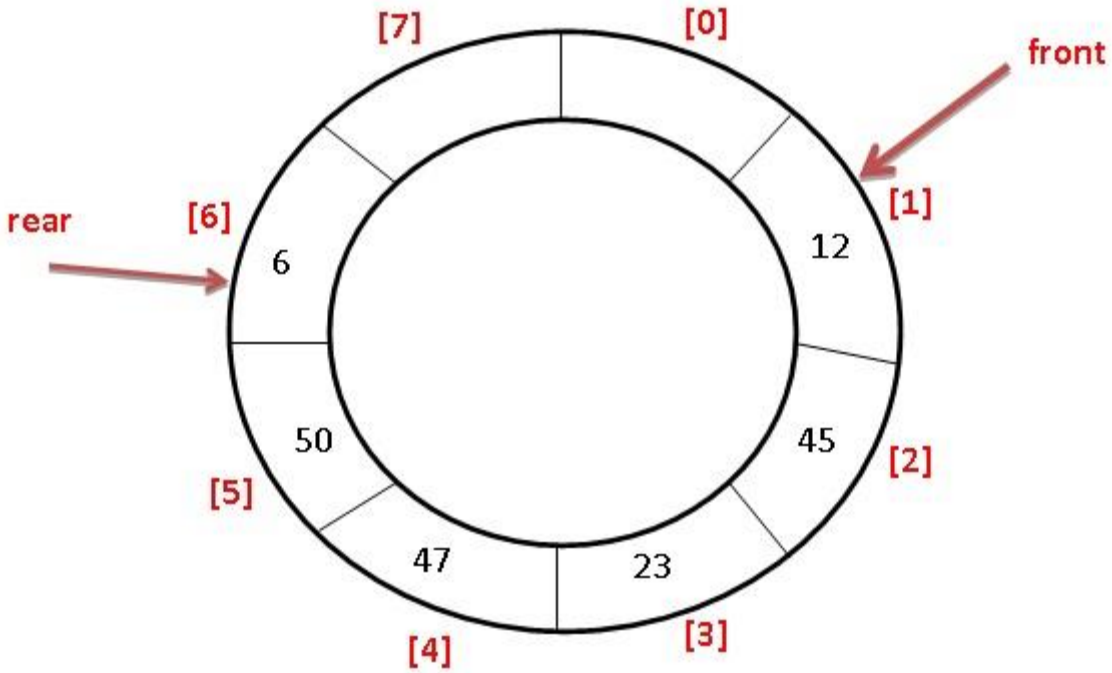
Queue with 6 elements

Circular Queues



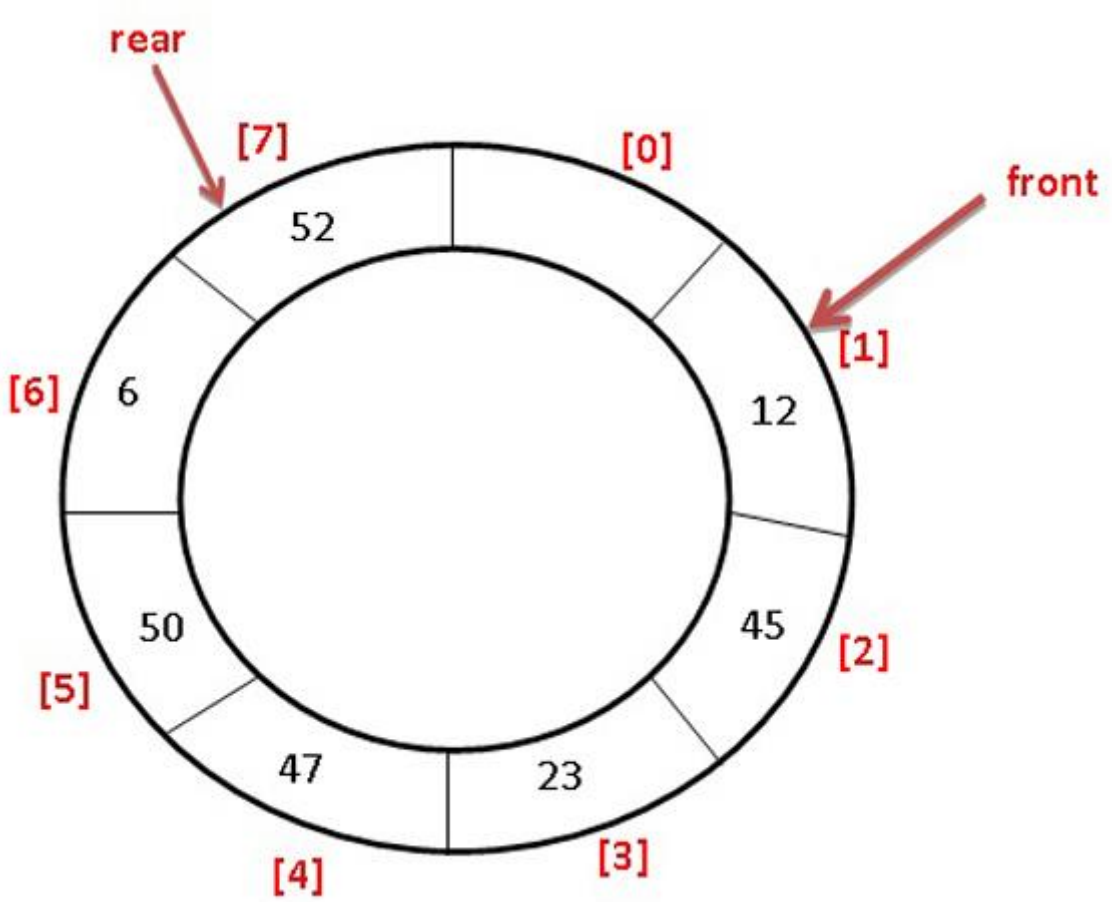
dequeue()

Circular Queues



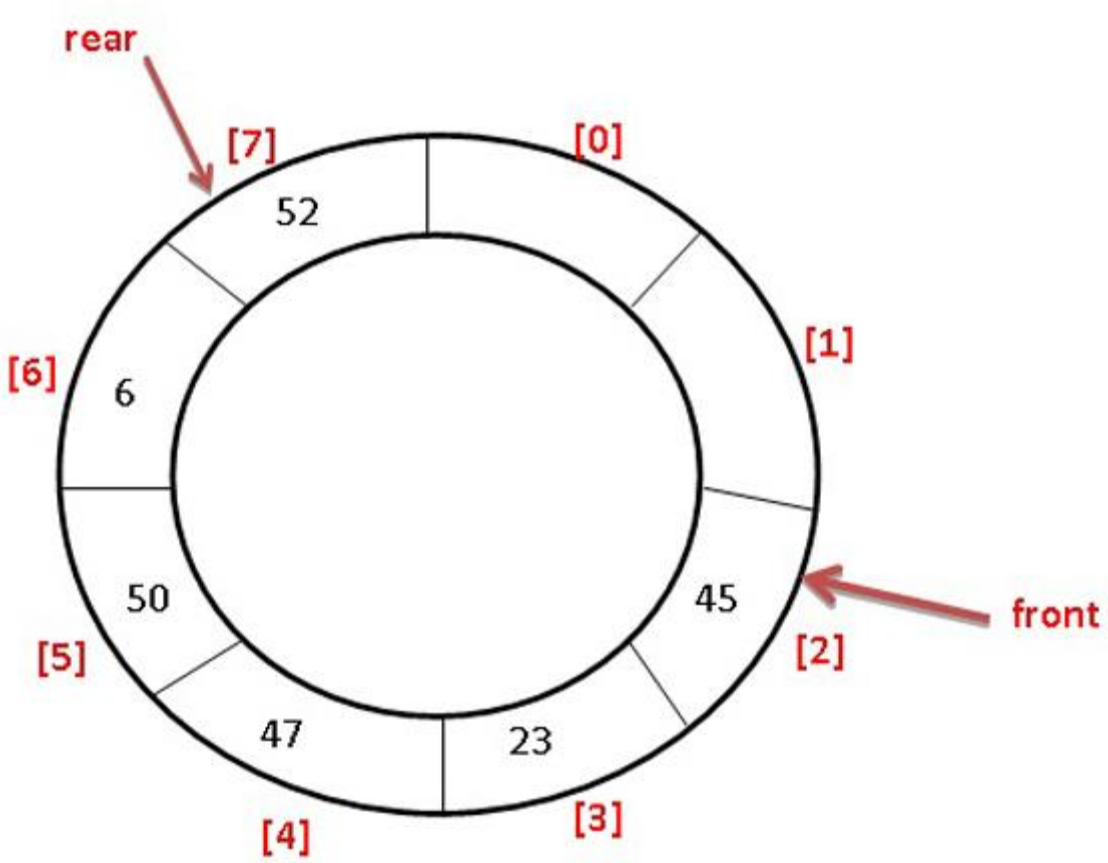
enqueue(6)

Circular Queues



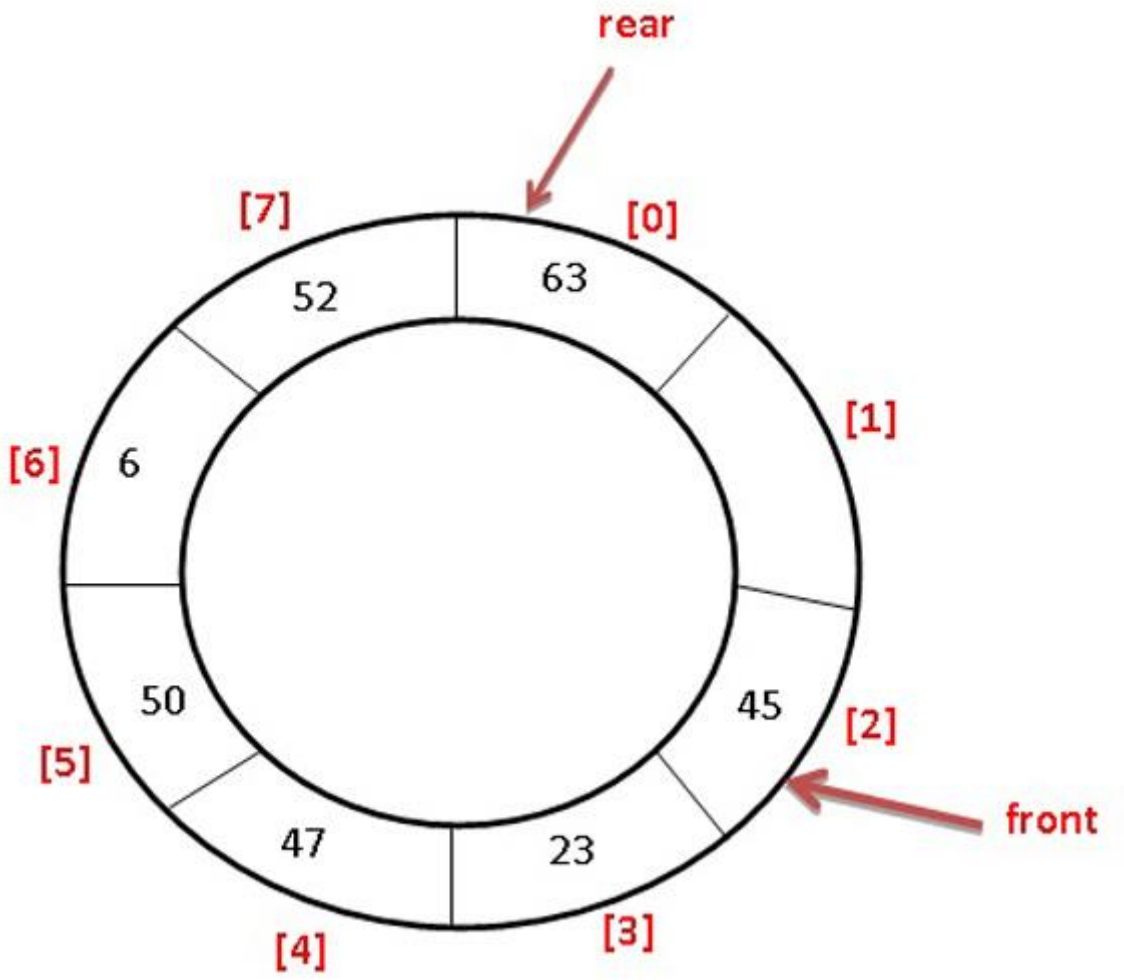
enqueue(52)

Circular Queues



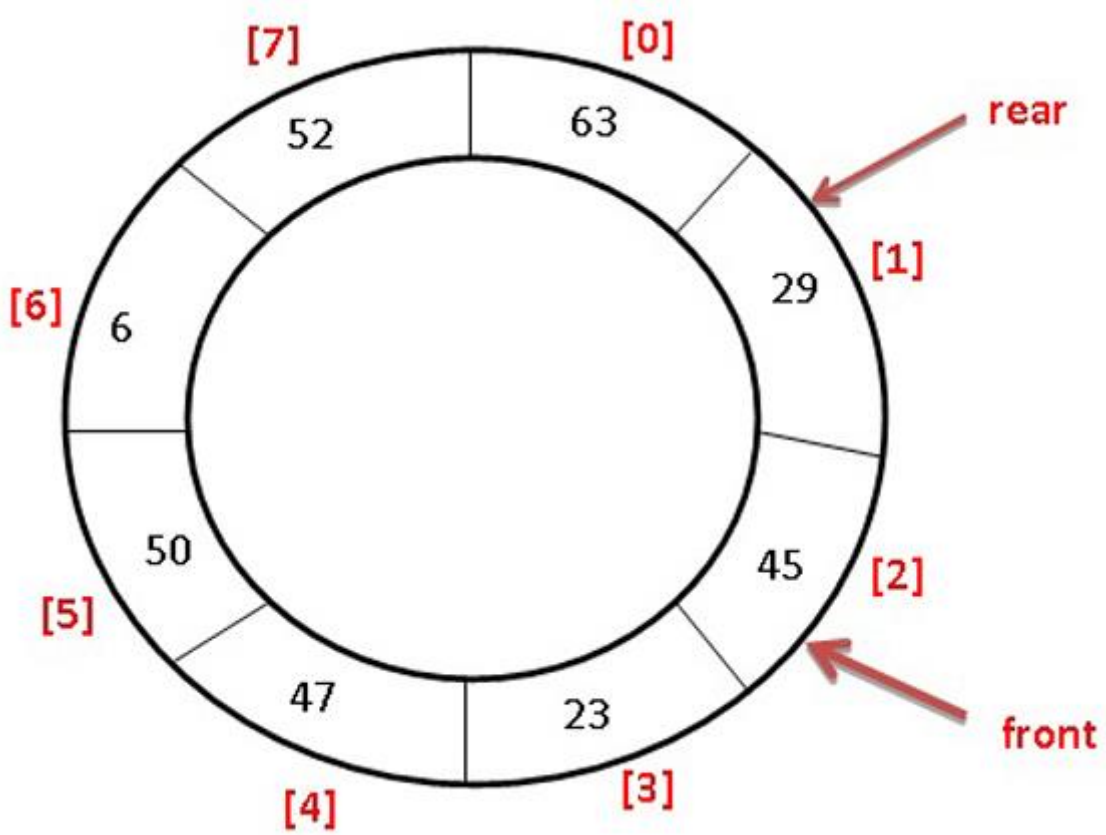
dequeue()

Circular Queues



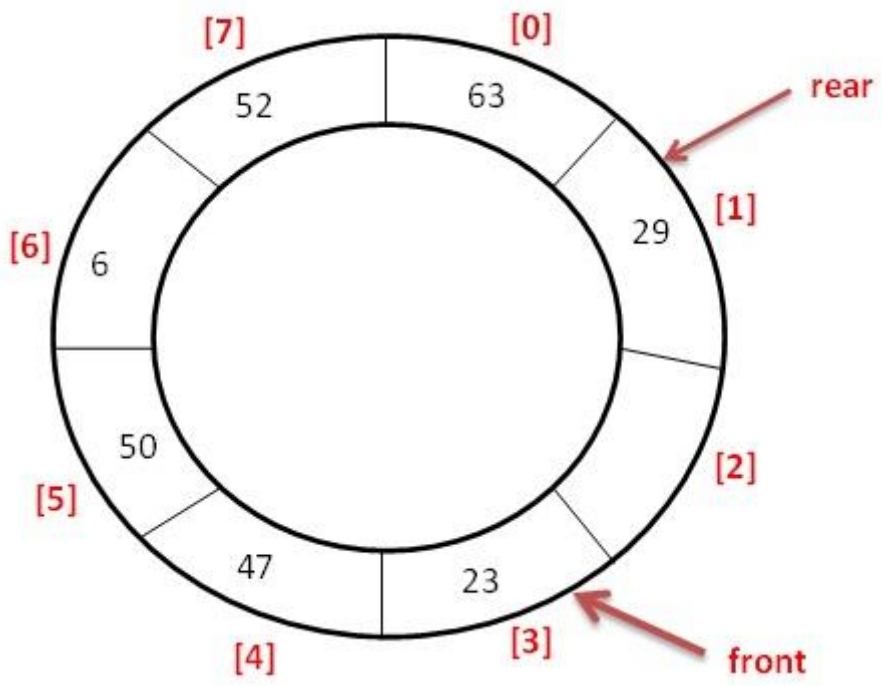
enqueue(63)

Circular Queues



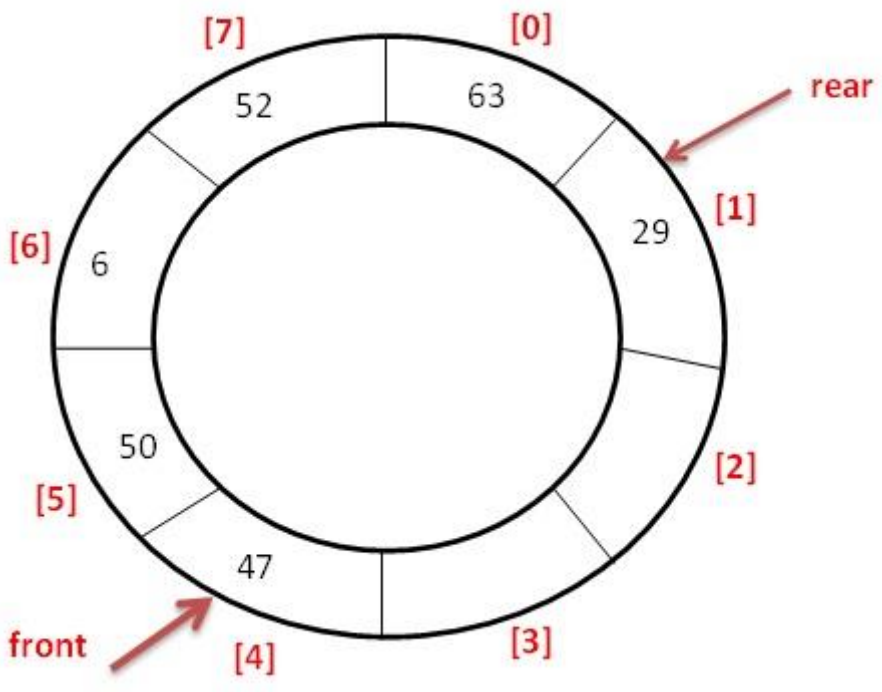
enqueue(29)

Circular Queues



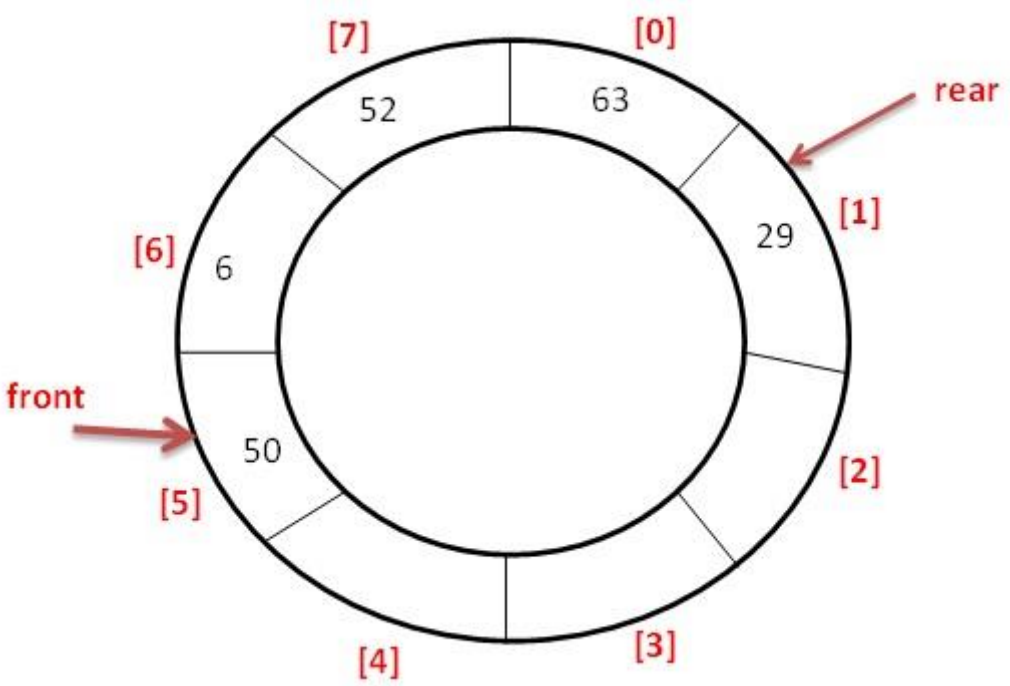
dequeue()

Circular Queues



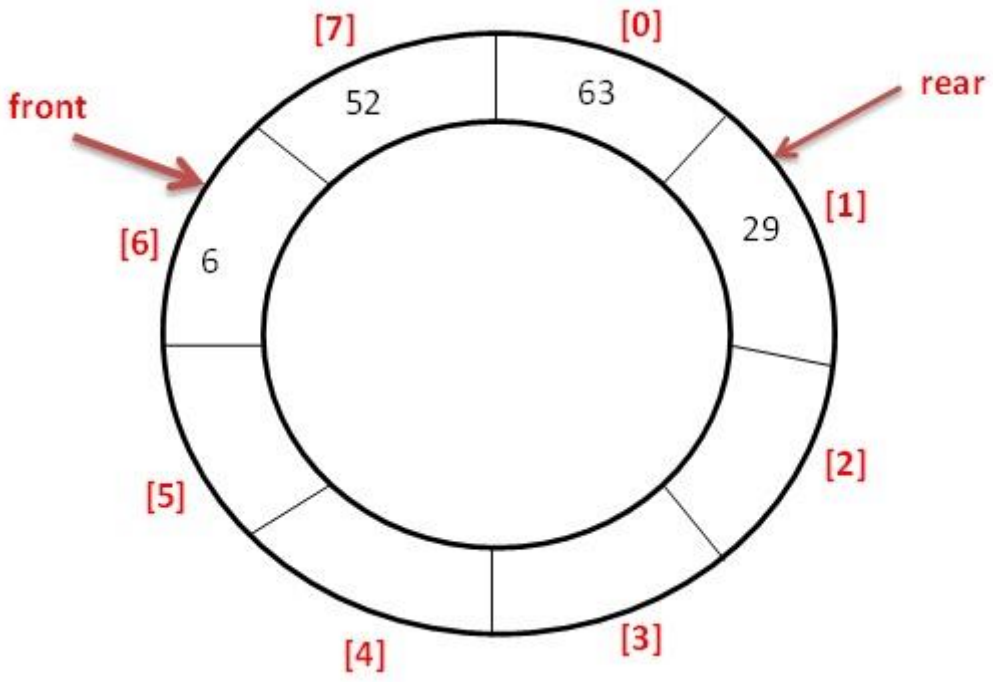
dequeue()

Circular Queues



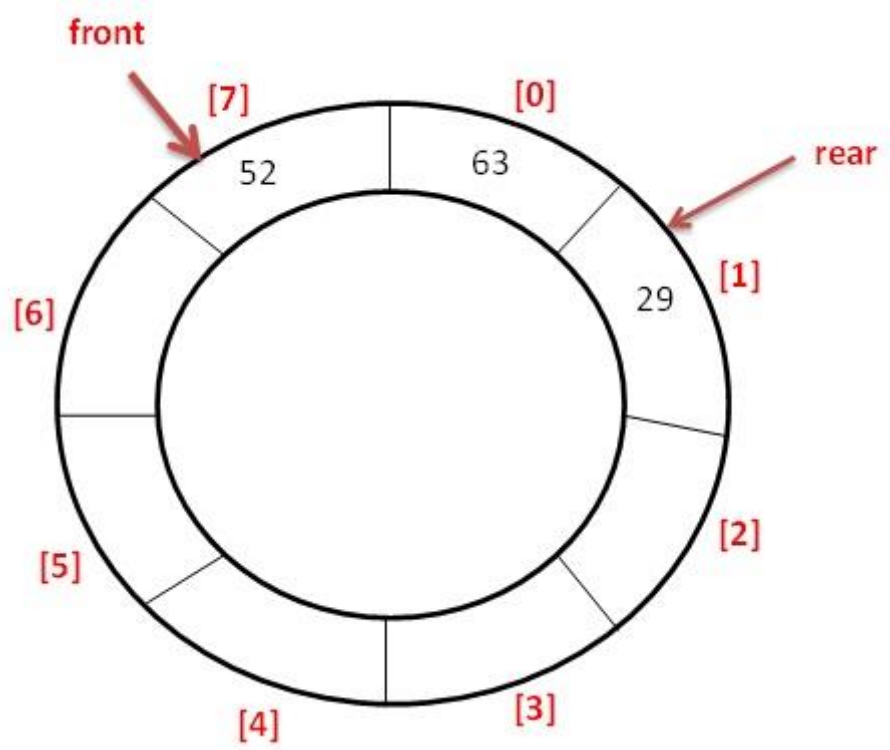
dequeue()

Circular Queues



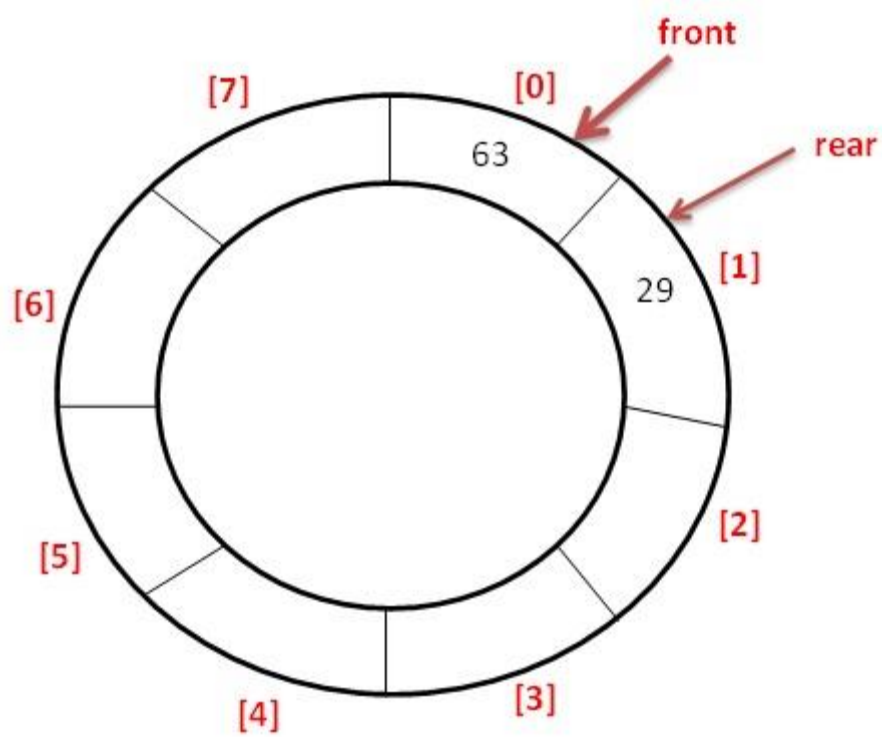
dequeue()

Circular Queues



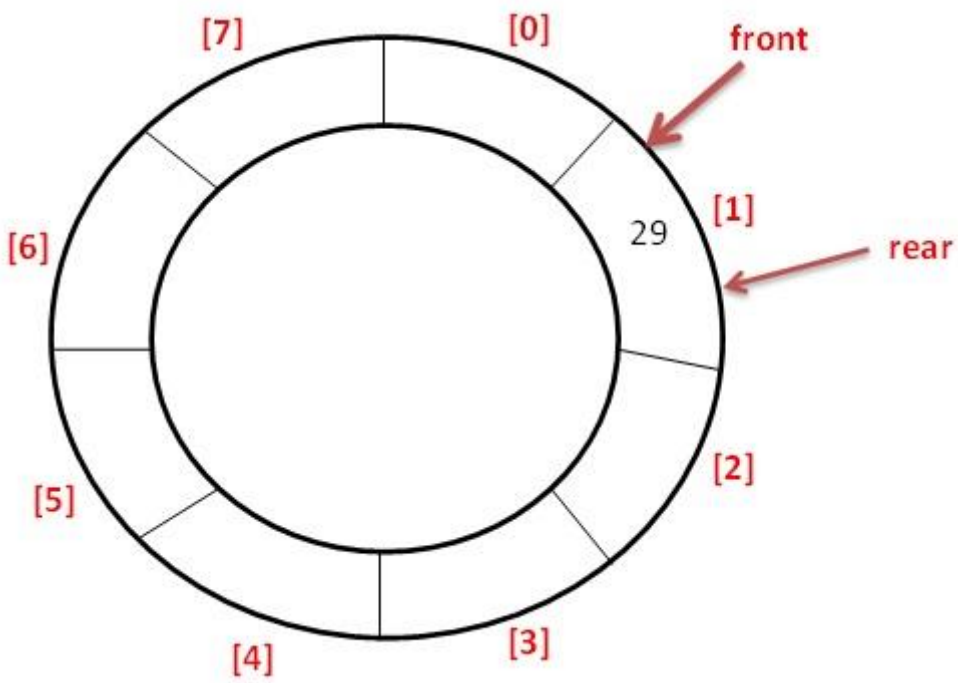
dequeue()

Circular Queues



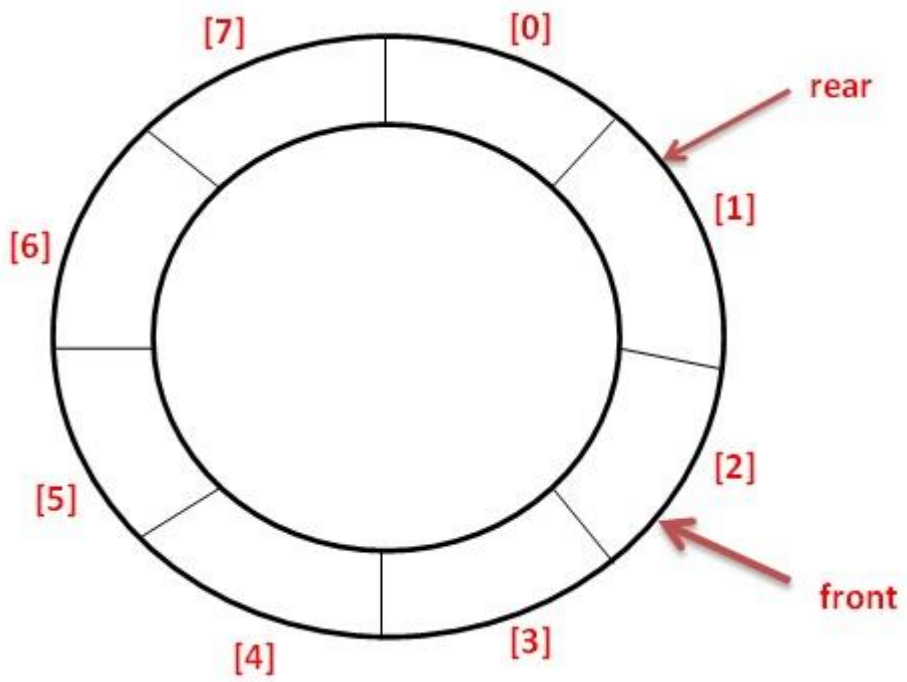
dequeue()

Circular Queues



dequeue()

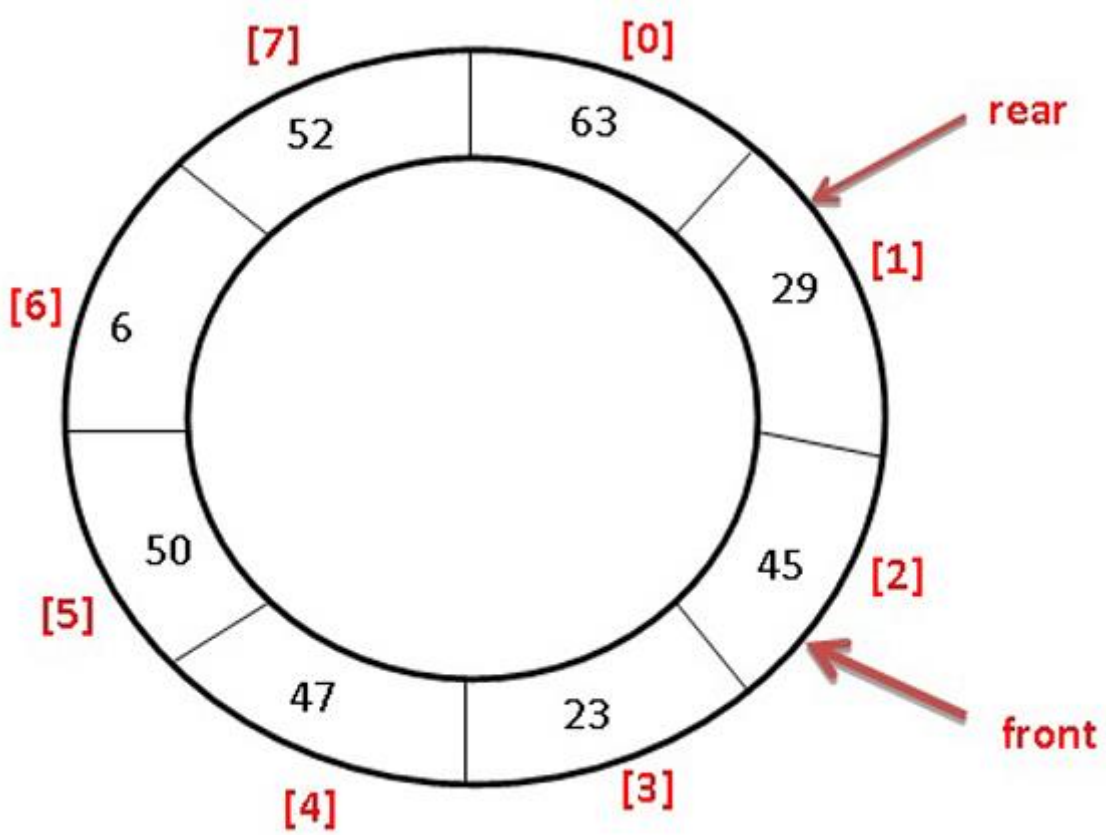
Circular Queues



Empty Queue

dequeue()

Circular Queues

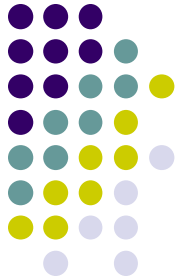


Full Queue

Array Implementation of the Circular Queues

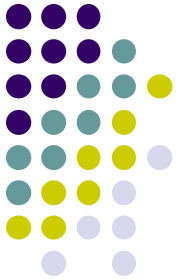
```
typedef char QueueItemType;
class Queue{
public:
   Queue(int size);
   ~Queue();
   bool isEmpty();
   bool isFull();
   bool enqueue(QueueItemType newItem);
   bool dequeue(QueueItemType *QueueTop);
private:
   QueueItemType *items;
   int front, rear, count;
   int MaxQueue;
};
```

Array Implementation of the Circular Queues



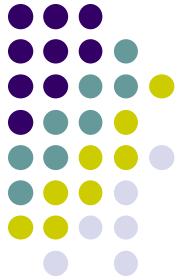
```
Queue::Queue(int size) {
        items = new QueueItemType[size];
        MaxQueue = size;
        front = 0;
        rear = -1;
        count = 0;
   }

Queue:: ~Queue(){
        delete [] items;
        items = NULL;
}
```

Array Implementation of the Circular Queues



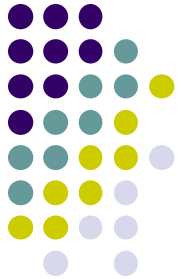
```
bool Queue::isEmpty() {
   return count <= 0;
}

bool Queue::isFull() {
   return count >= MaxQueue-1;
}
```

Array Implementation of the Circular Queues

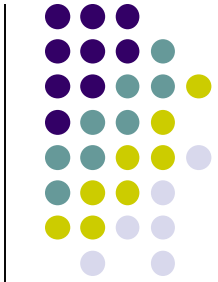


```
bool Queue::enqueue(QueueItemType newItem){
   if (isFull())
      return false;
   else{
      rear = (rear + 1) % MaxQueue;
           items[rear] = newItem;
           count++;
      return true;
   }
}
```

Array Implementation of the Circular Queues

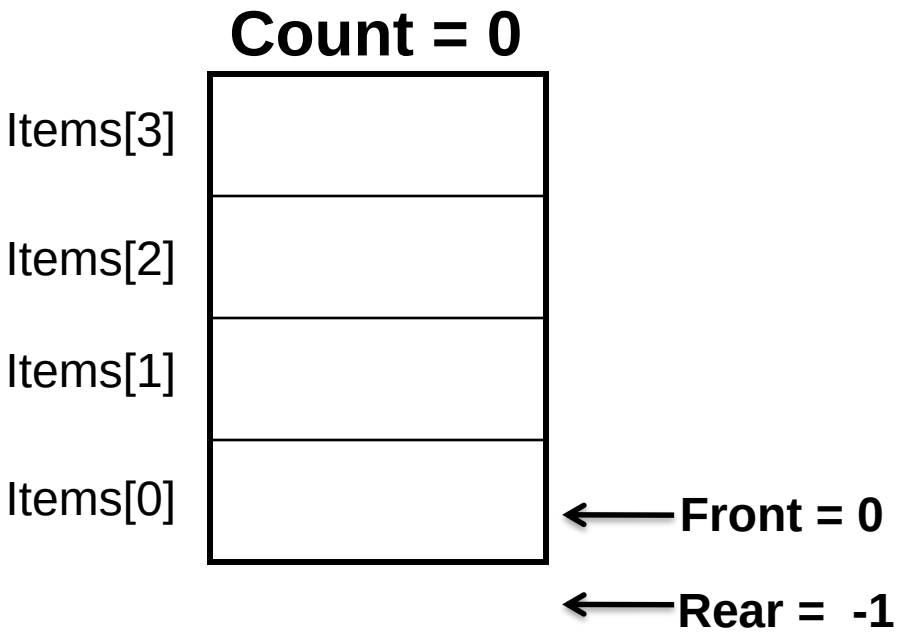


```
bool Queue::dequeue   (QueueItemType
*QueueTop){
   if (isEmpty())
      return false;
   else {
        count--;
         *QueueTop = items[front];
         front = (front + 1) % MaxQueue;
        return true;
   }
}
```

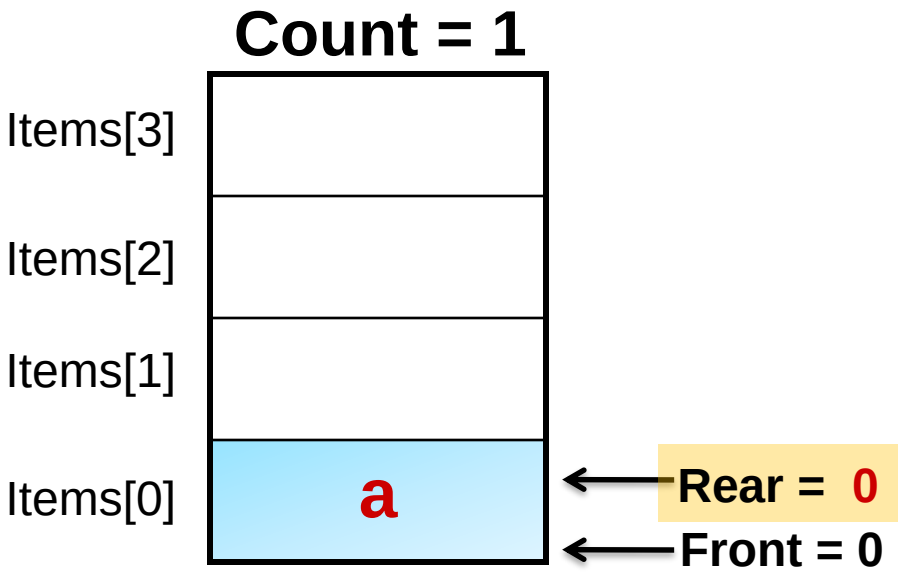


```
main(){
        QueueItemType c;
        Queue Queue(4);
        Queue.enqueue('a');
        Queue.enqueue('b');
        Queue.enqueue('c');
        Queue.dequeue(&c);
        printf("%c\n",c);
        Queue.dequeue(&c);
        printf("%c\n",c);
        Queue.enqueue('d');
        Queue.dequeue(&c);
        Queue.enqueue(‘e');
        printf("%c\n",c);
        Queue.dequeue(&c);
        printf("%c\n",c);
        Queue.dequeue(&c);
        printf("%c\n",c);
}
```

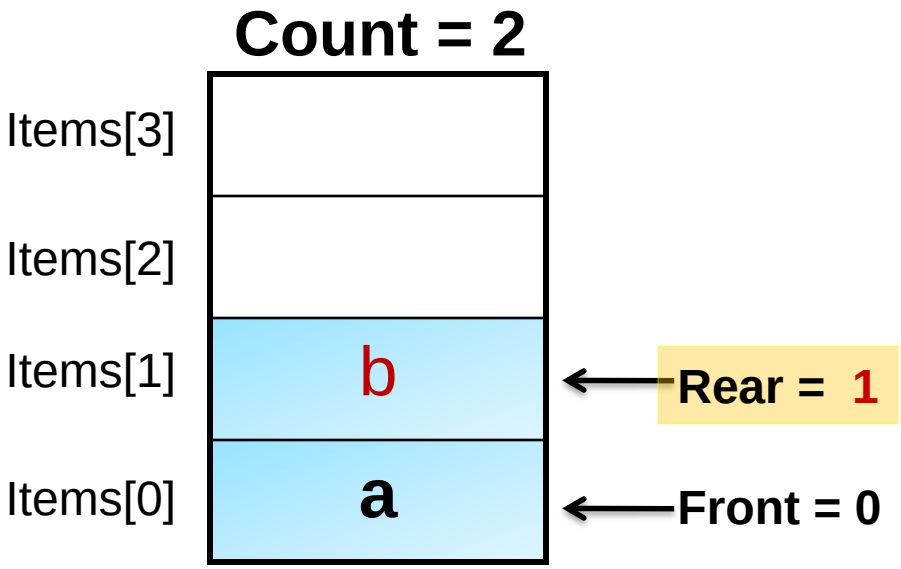
queue(4)



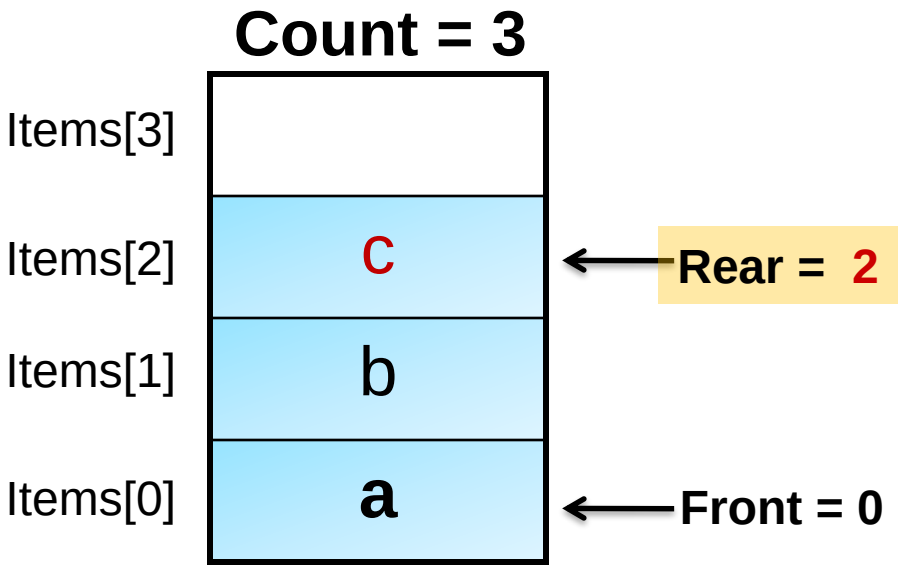
enqueue ('a')



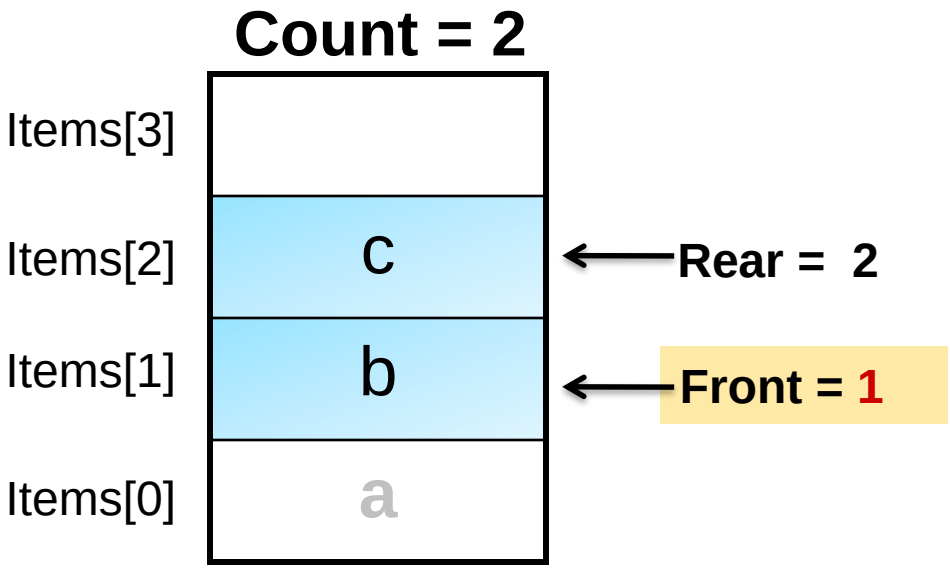
enqueue ('b')



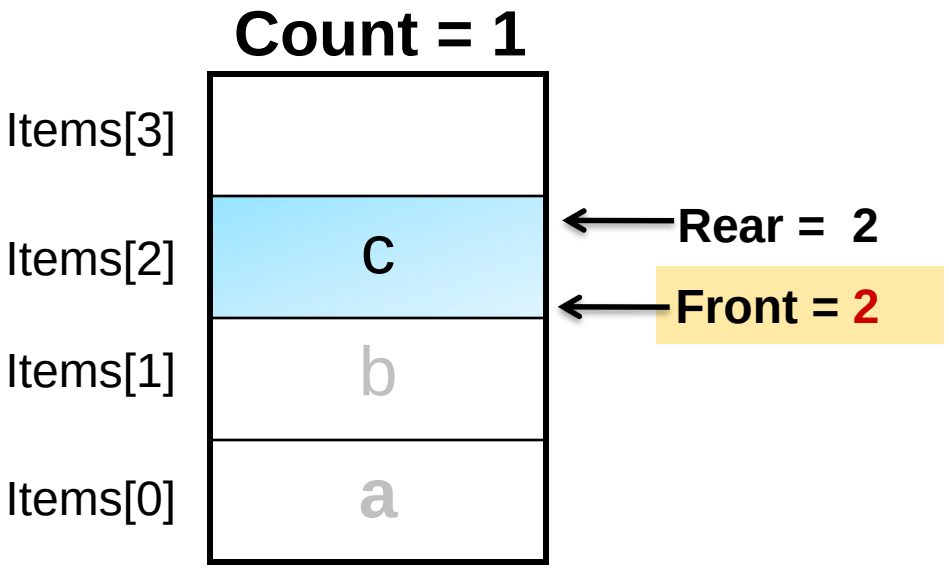
enqueue ('c')



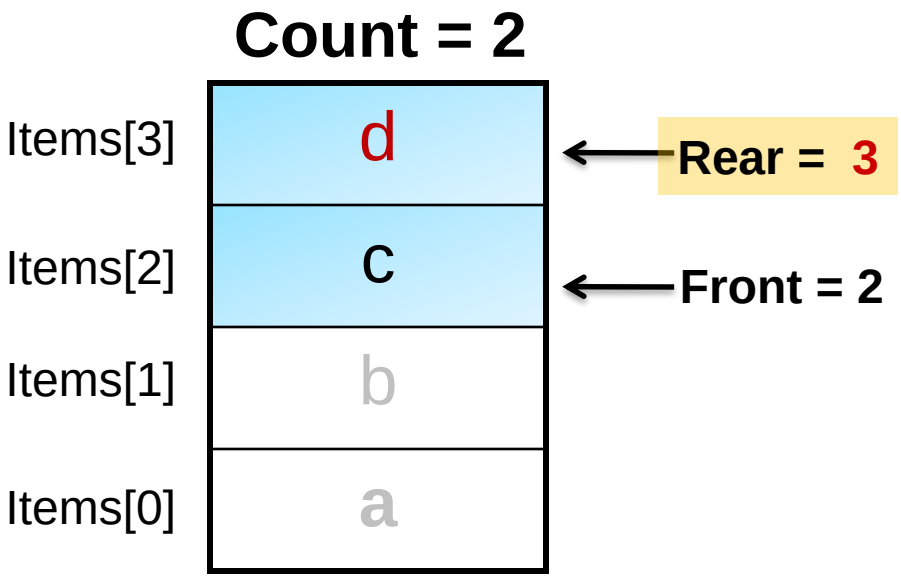
dequeue (&c)



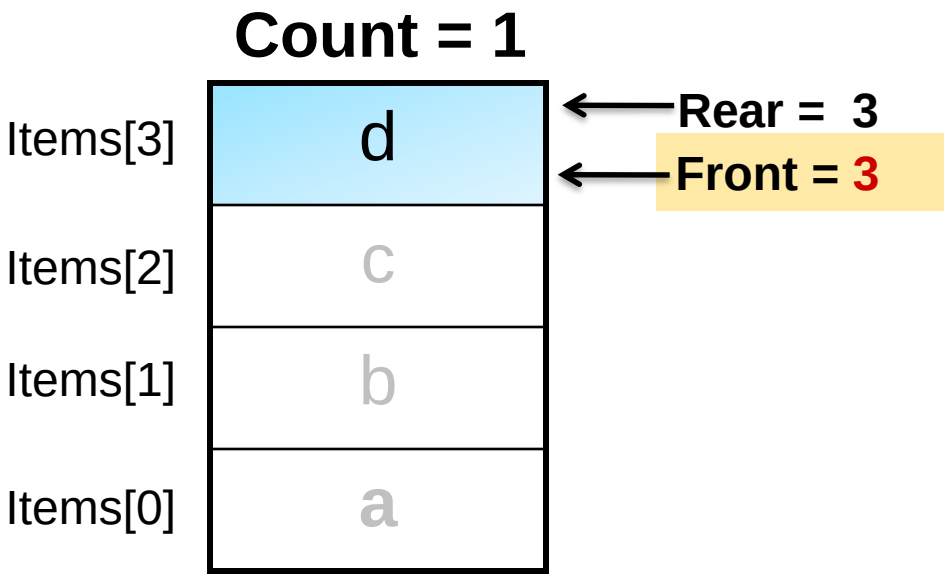
dequeue (&c)



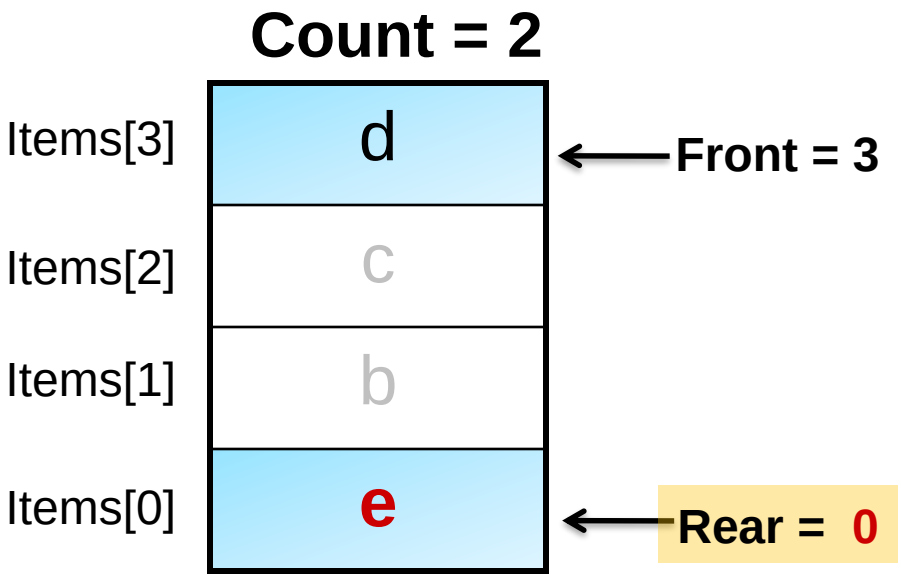
enqueue ('d')



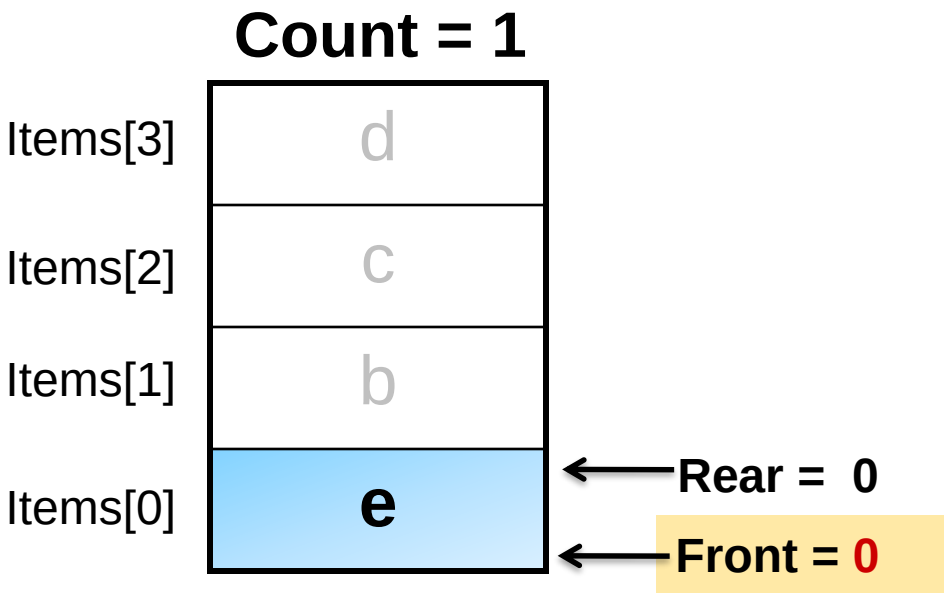
dequeue (&c)



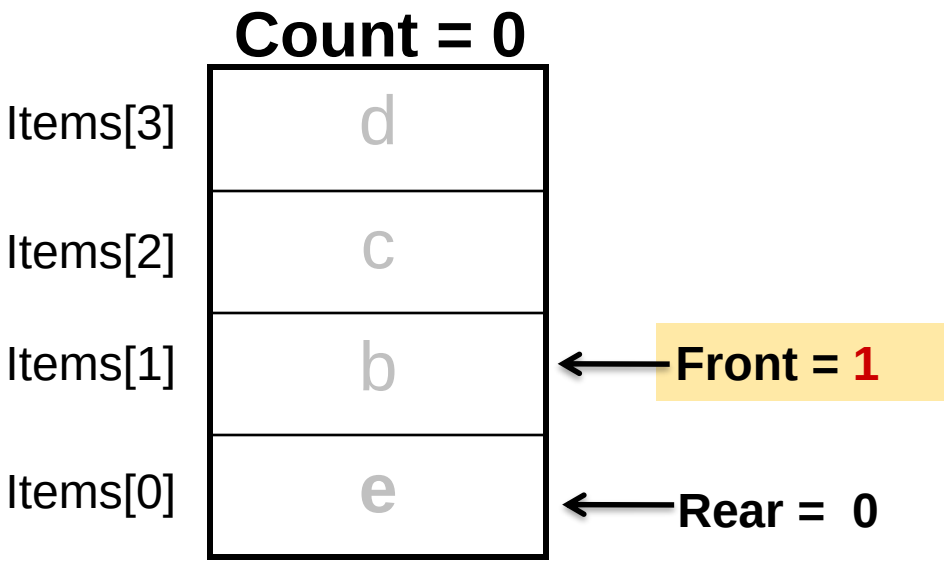
enqueue ('e')



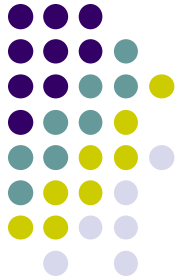
dequeue (&c)



dequeue (&c)

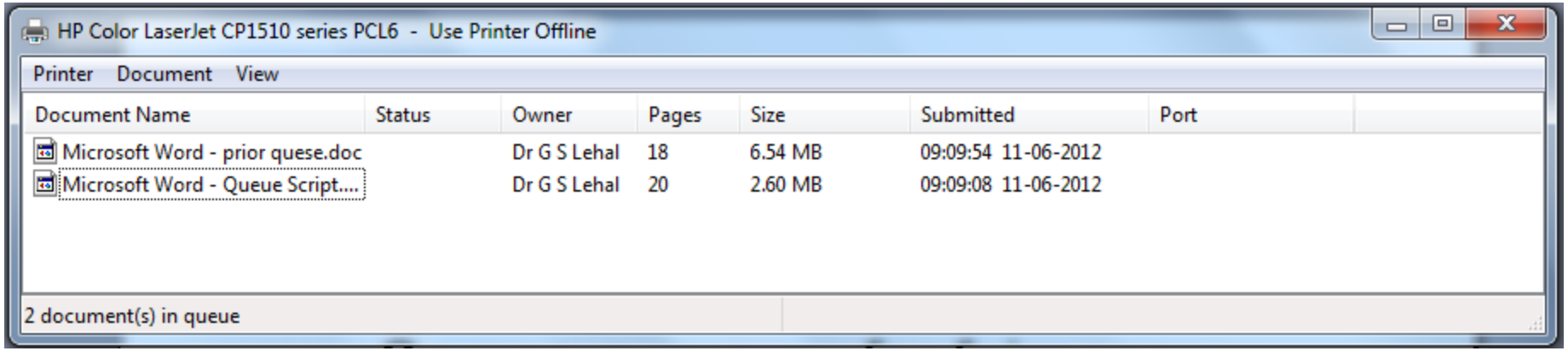


Applications of Queues

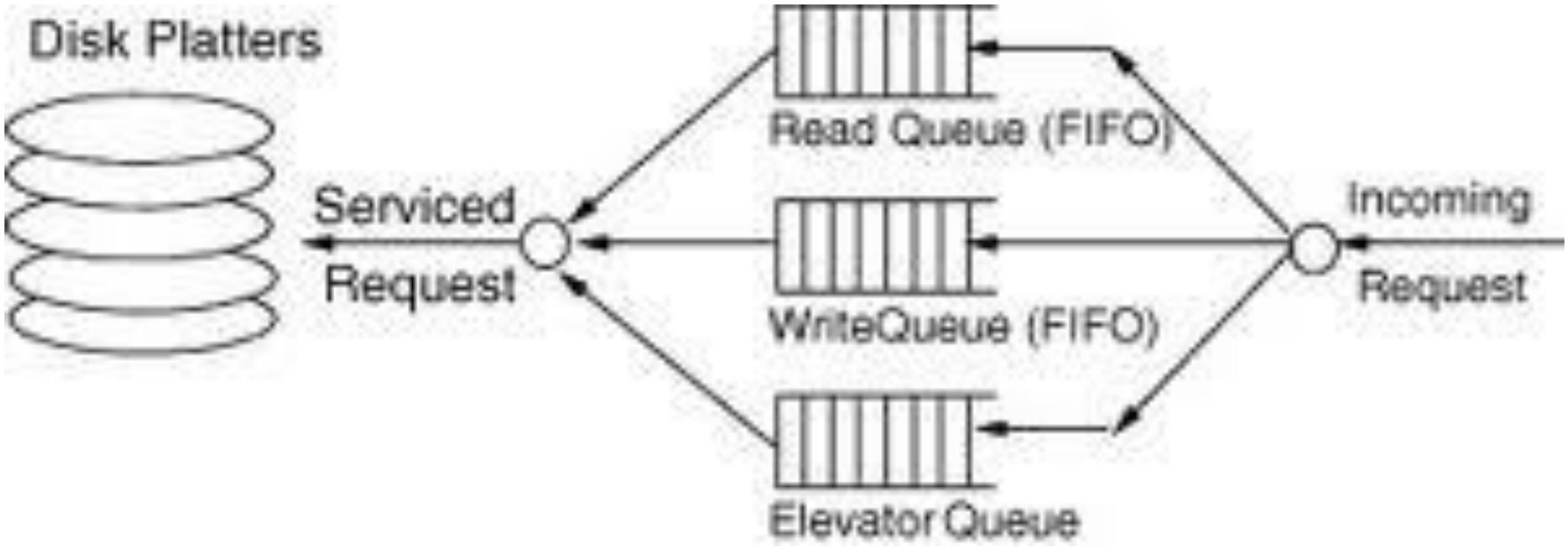


- Queues are used in operating systems, for controlling access to shared system resources such as:
  - Printer
  - Disk access on a network
  - CPU

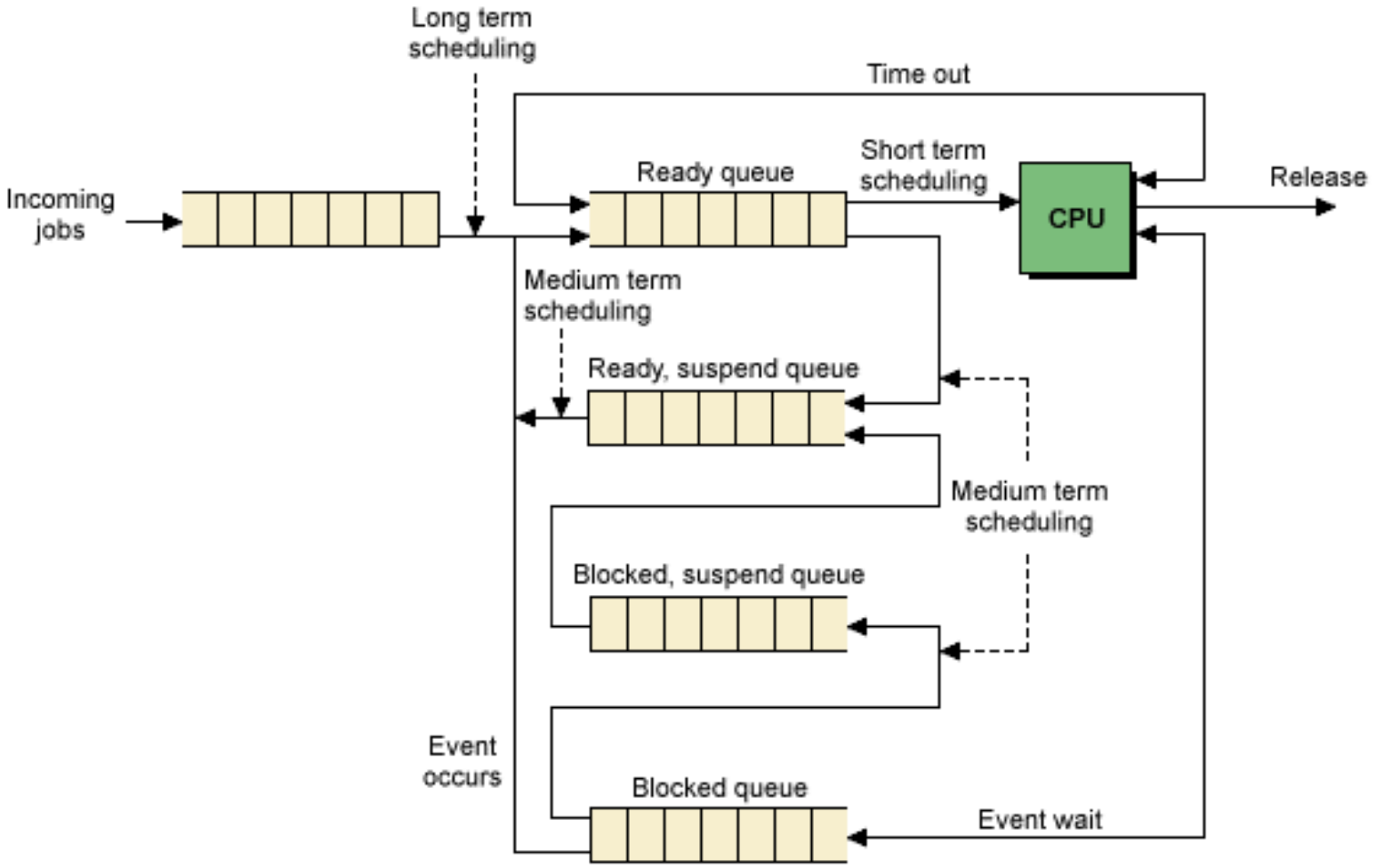
Printer Queue



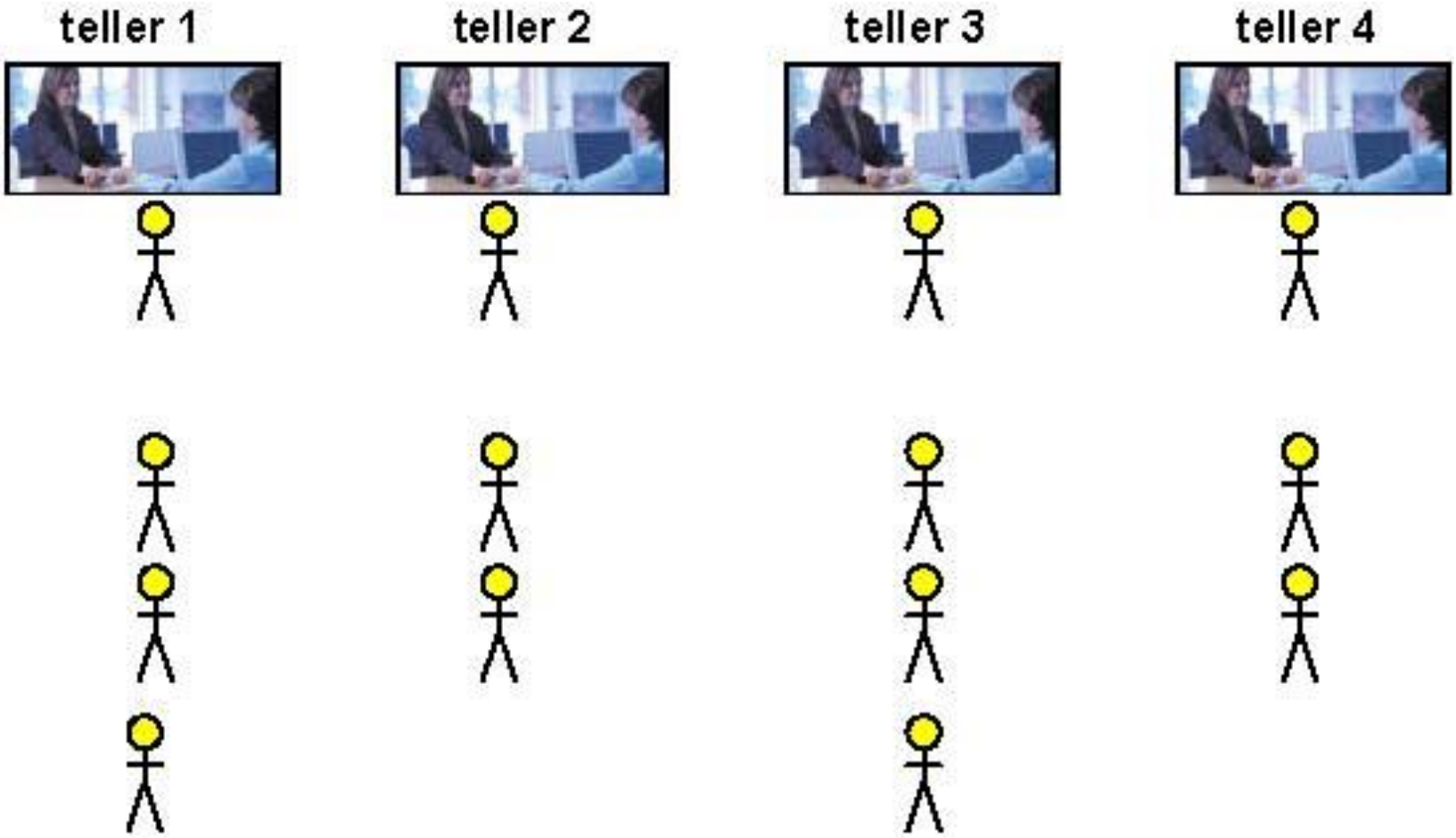
Disk Queue


Disk Platters
Serviced
Request
Read Queue (FIFO)
WriteQueue (FIFO)
Elevator Queue
Incoming
Request

Process Queues



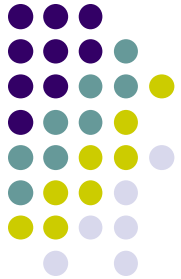
Queues in Simulation



Simulation of Airplane Traffic

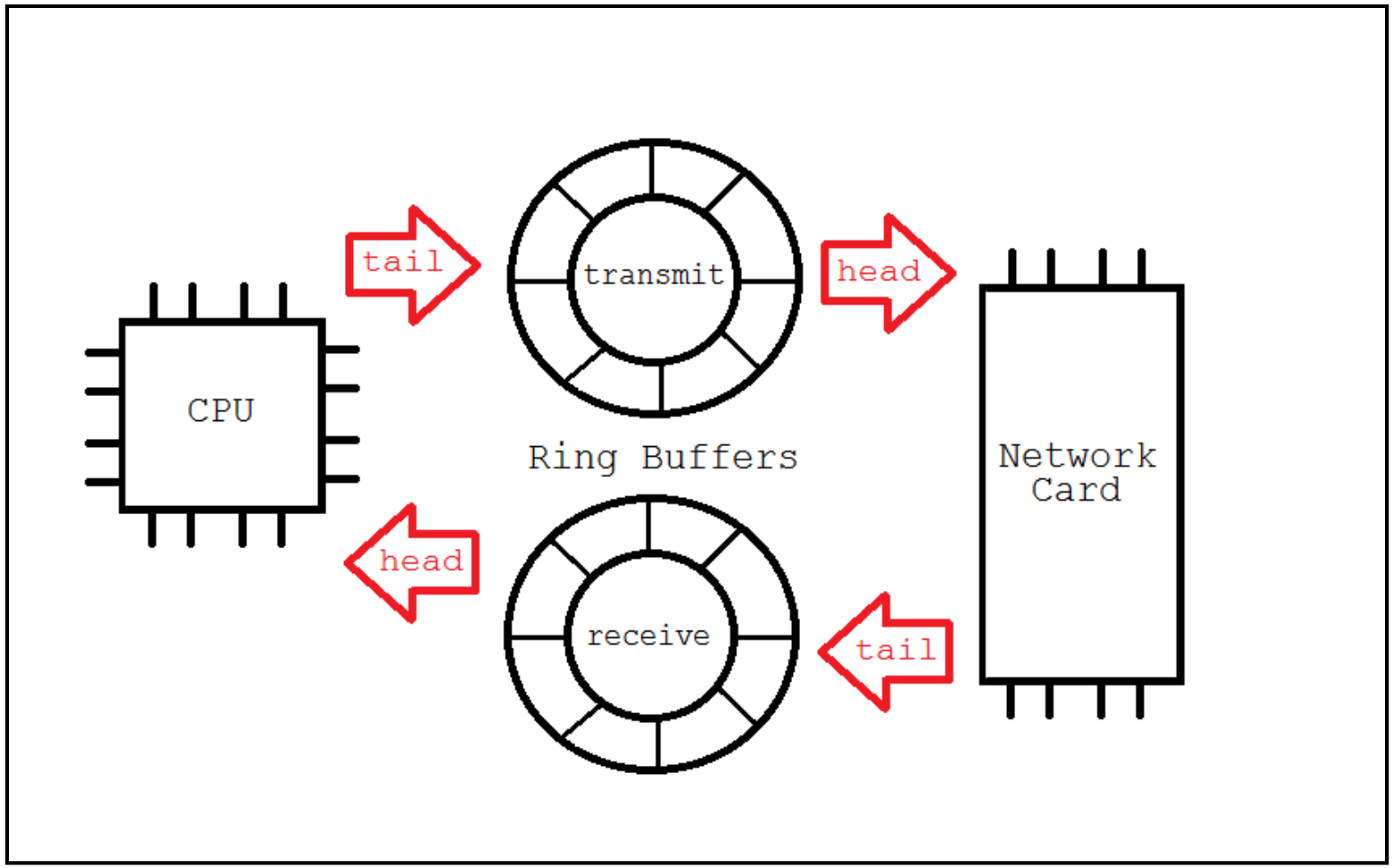
```
            Landing queue
            ----------------------------
Plane ---> )Plane ) Plane ) ... ) Plane) ---+
            ----------------------------     \ lands
                                              \
                                               v
                                             Runway --->
(exits simulation)
                                              ^
                    Take-off queue           / leaves
                    --------------------     /
          Plane ---> )Plane ) ... ) Plane)----+
```

Other Applications of Queues

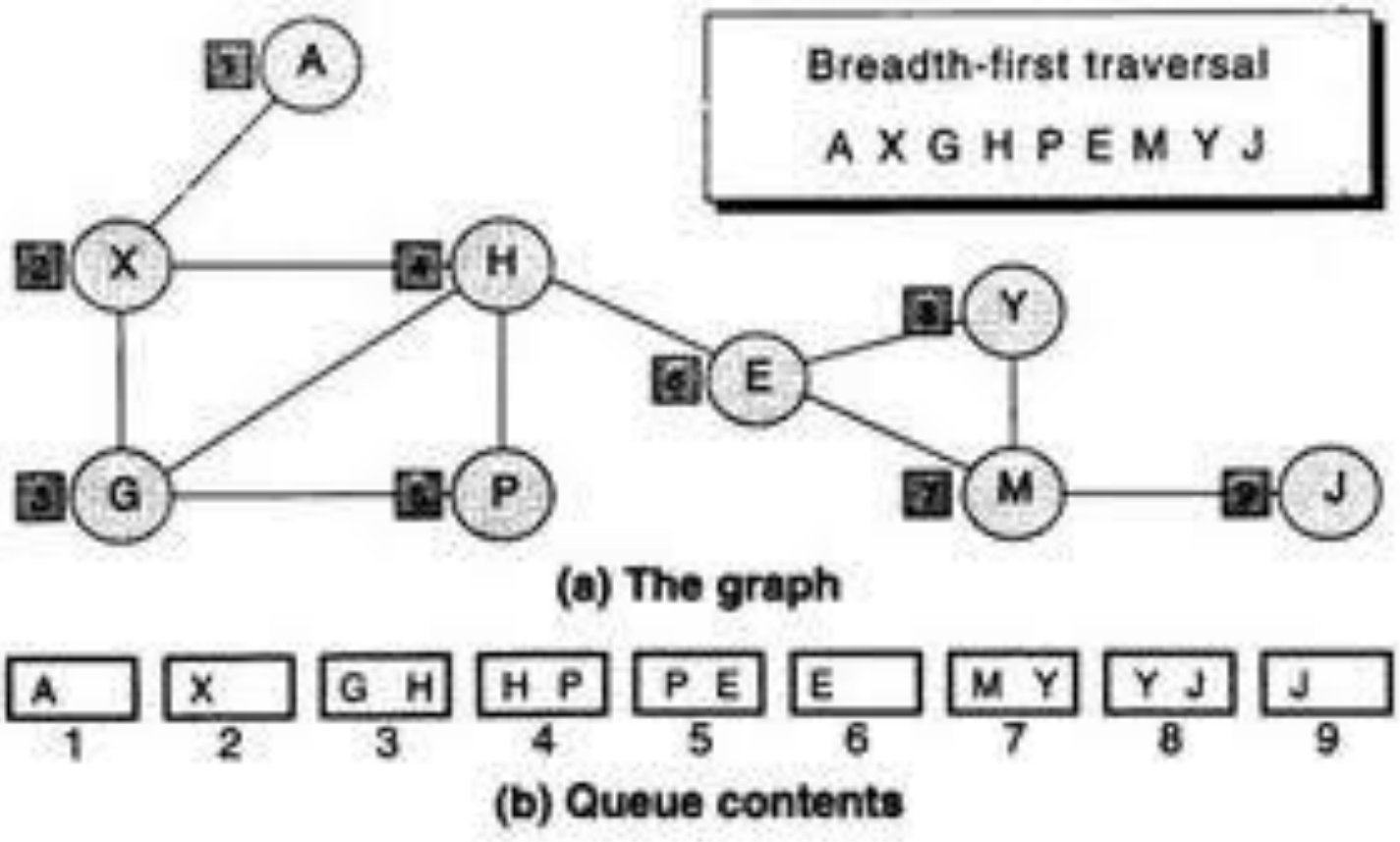


- Reading from the keyboard
- Implementing buffer
- Level order traversal of trees or breadth first search in a graph
- Implementing Radix Sort

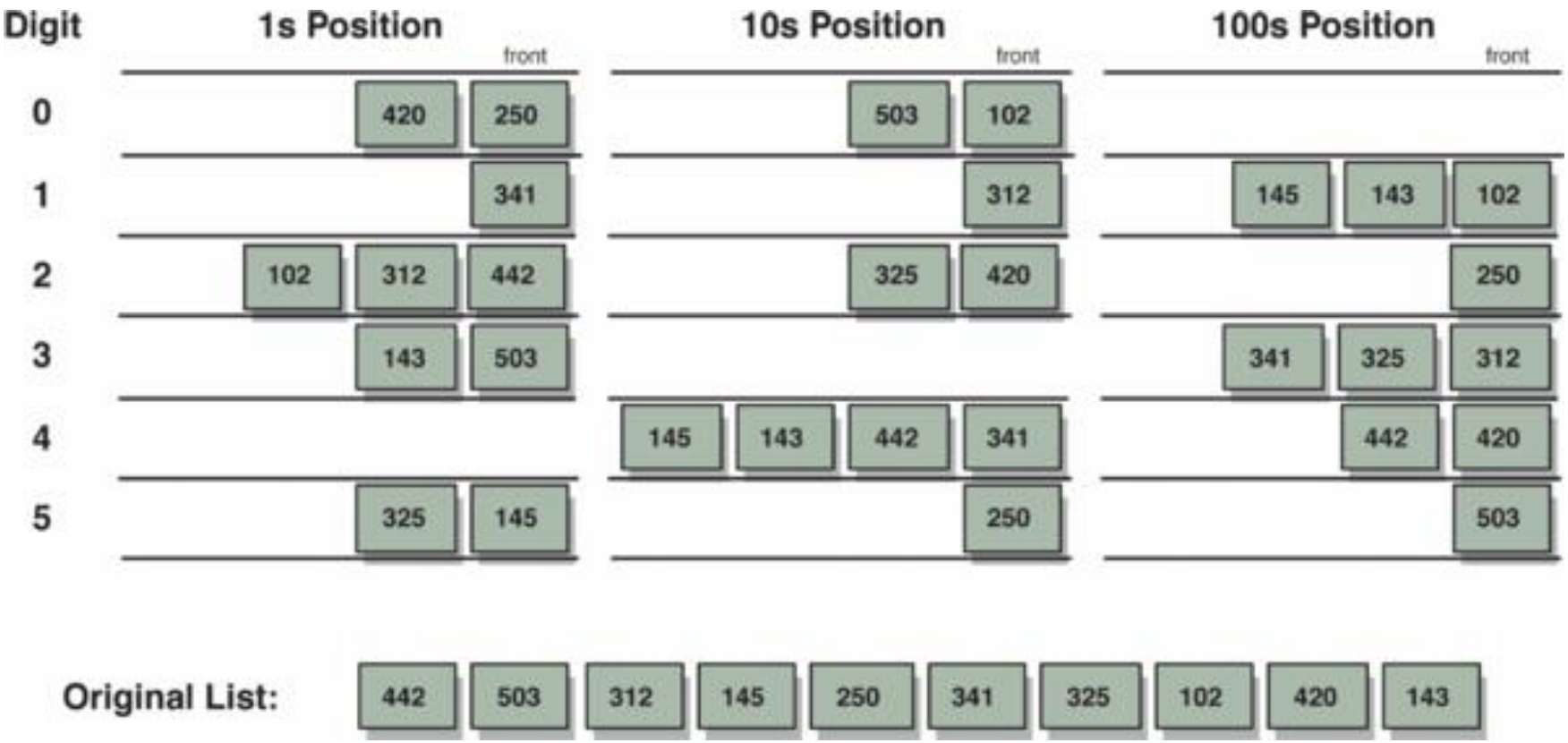
Buffers


tail
transmit
head
CPU
Ring Buffers
Network
Card
head
receive
tail

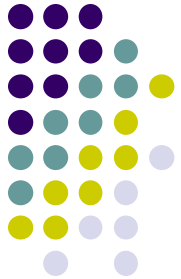
Breadth First Traversal of Graph



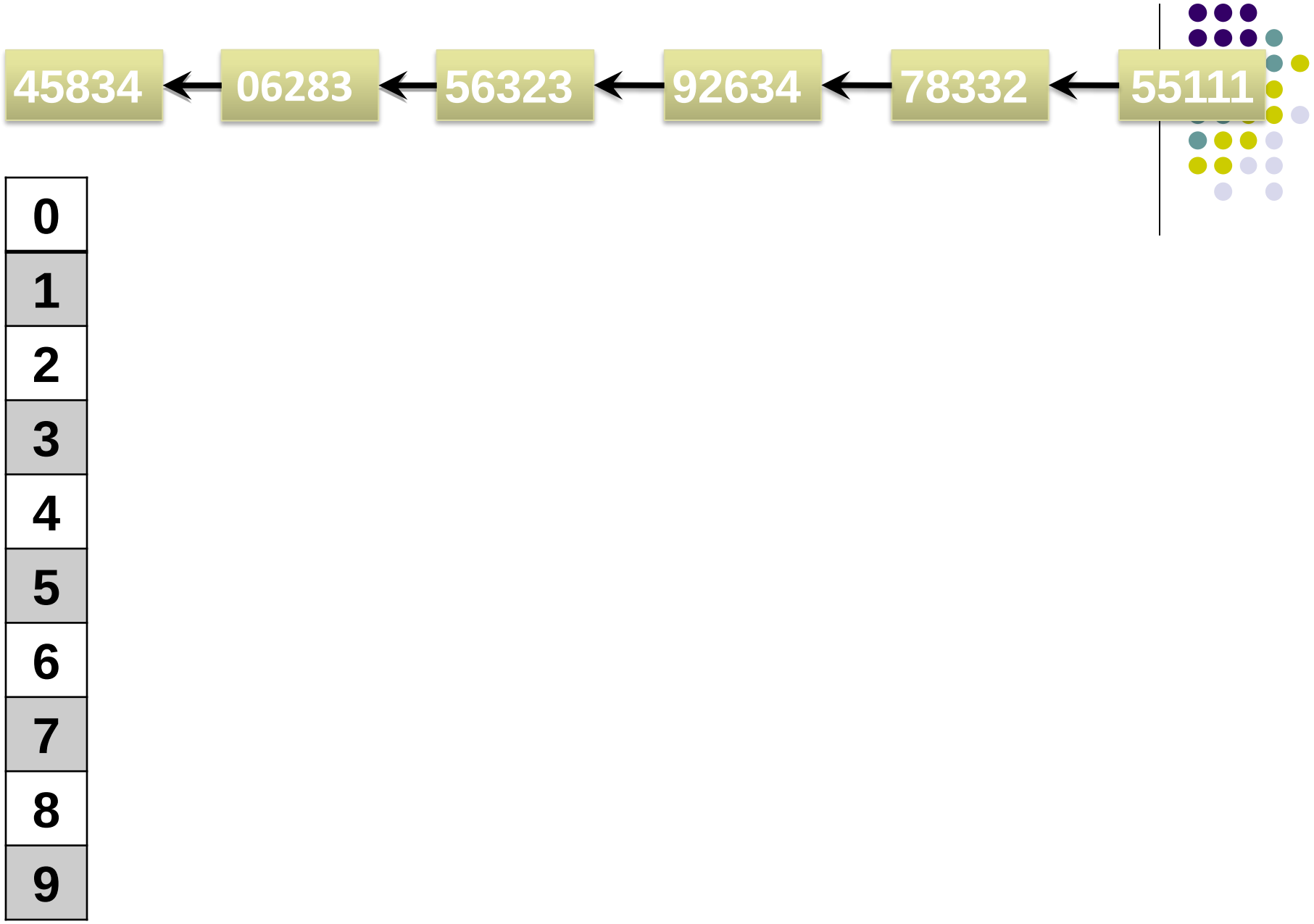
Radix Sort

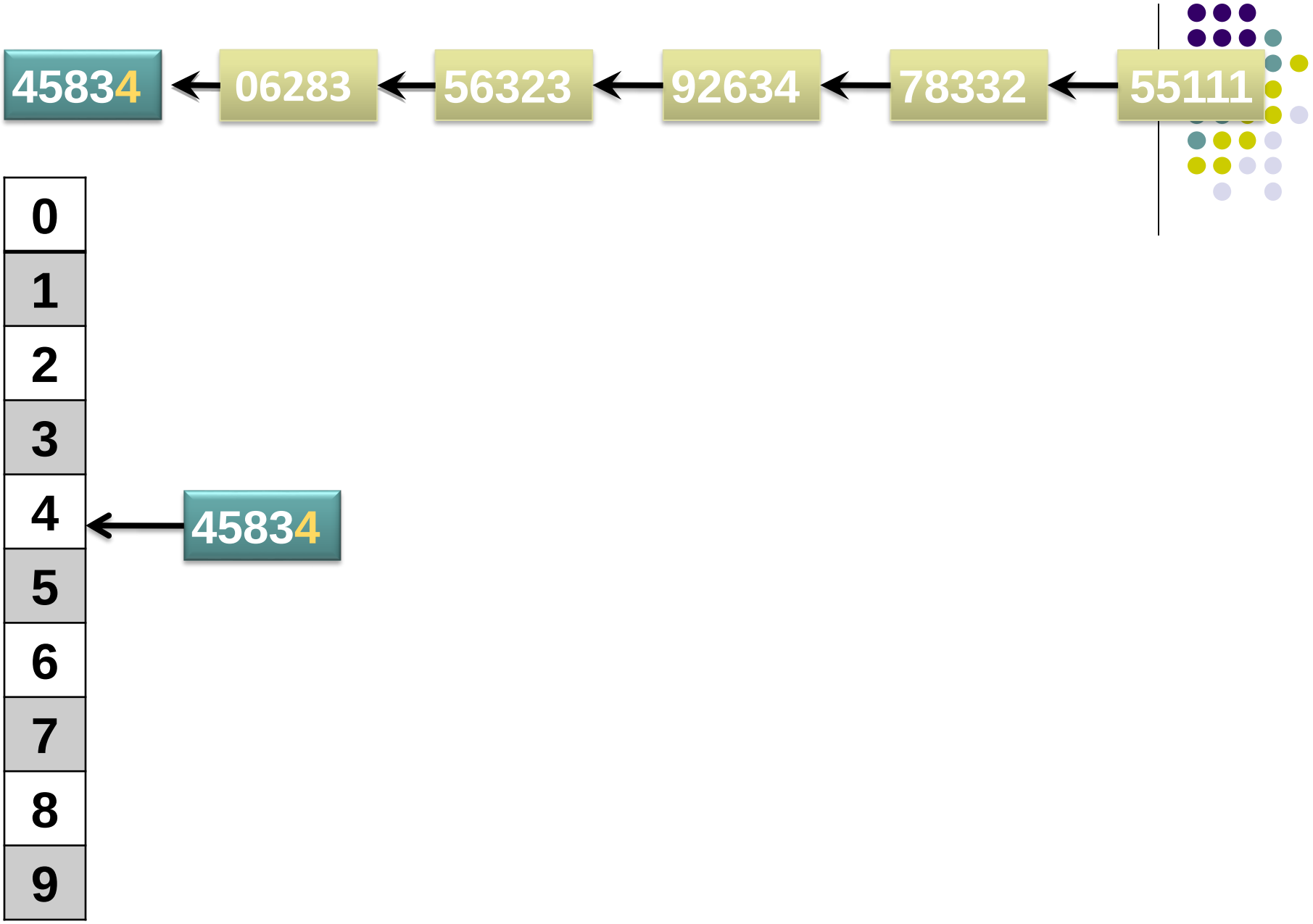

Digit
1s Position
10s Position
100s Position
front
0
1
2
3
4
5
420
250
341
102
312
442
143
503
325
145
503
102
312
325
420
145
143
442
341
250
145
143
102
250
341
325
312
442
420
503
Original List:
442
503
312
145
250
341
325
102
420
143

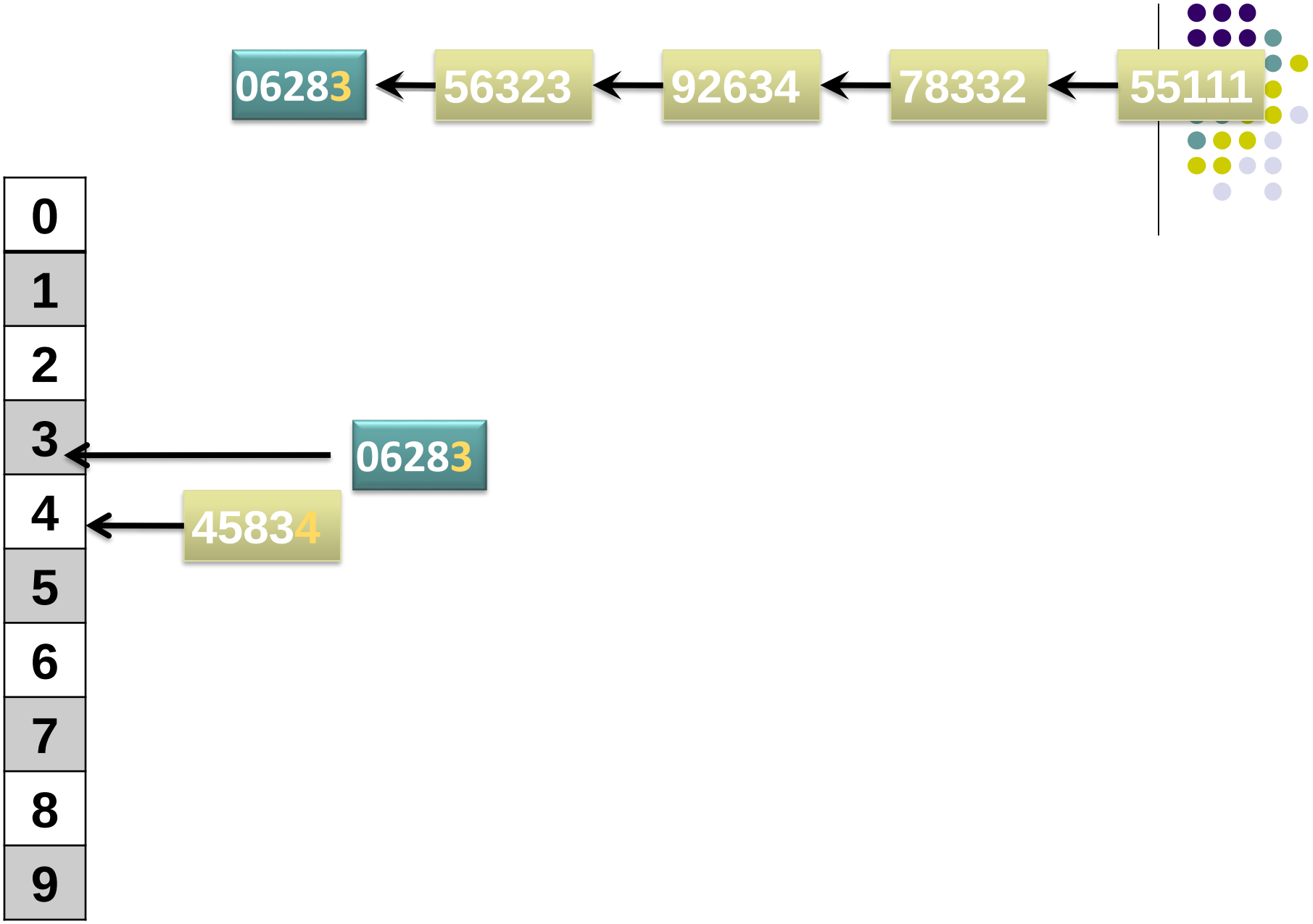
Radix Sort Algorithm

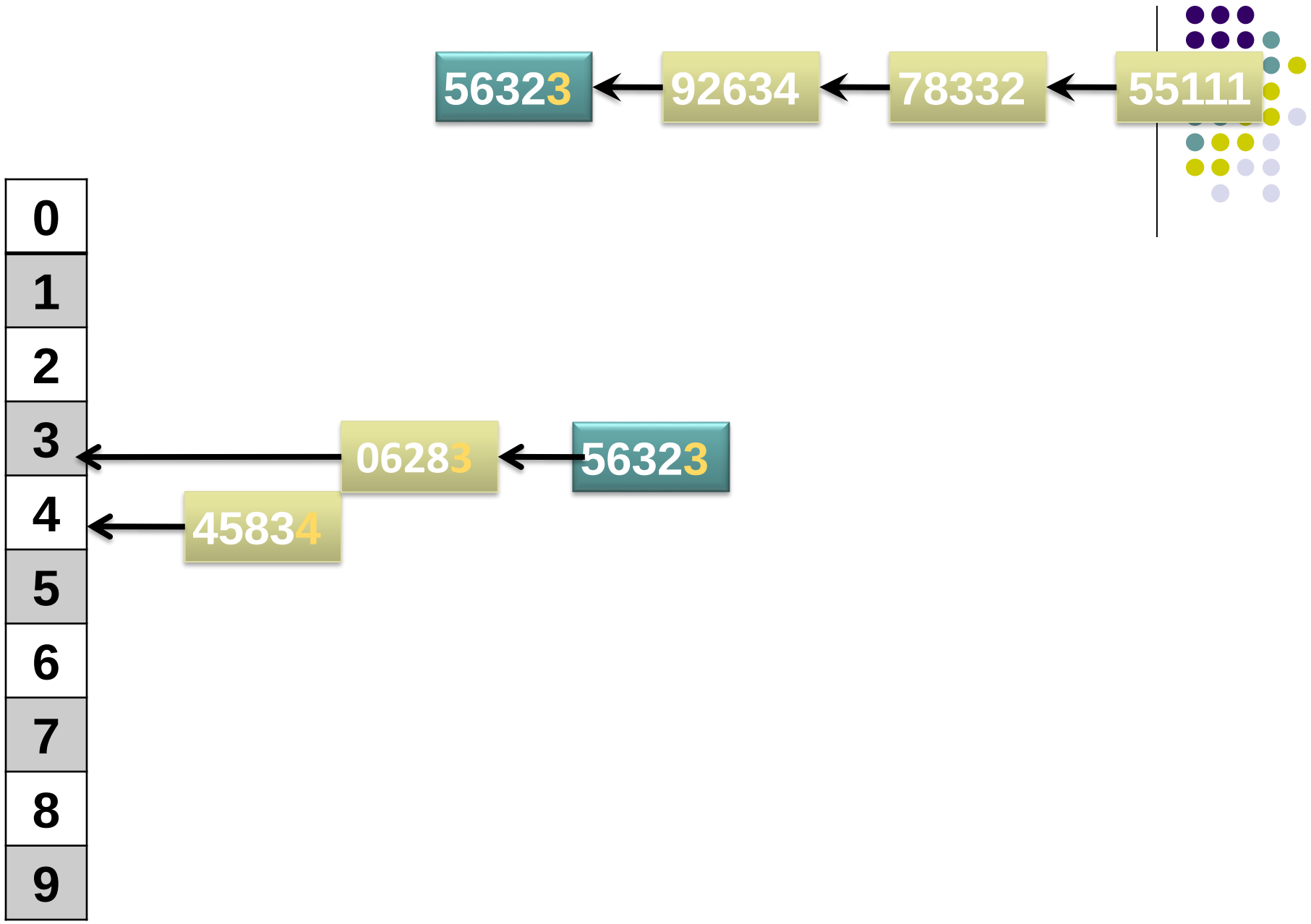


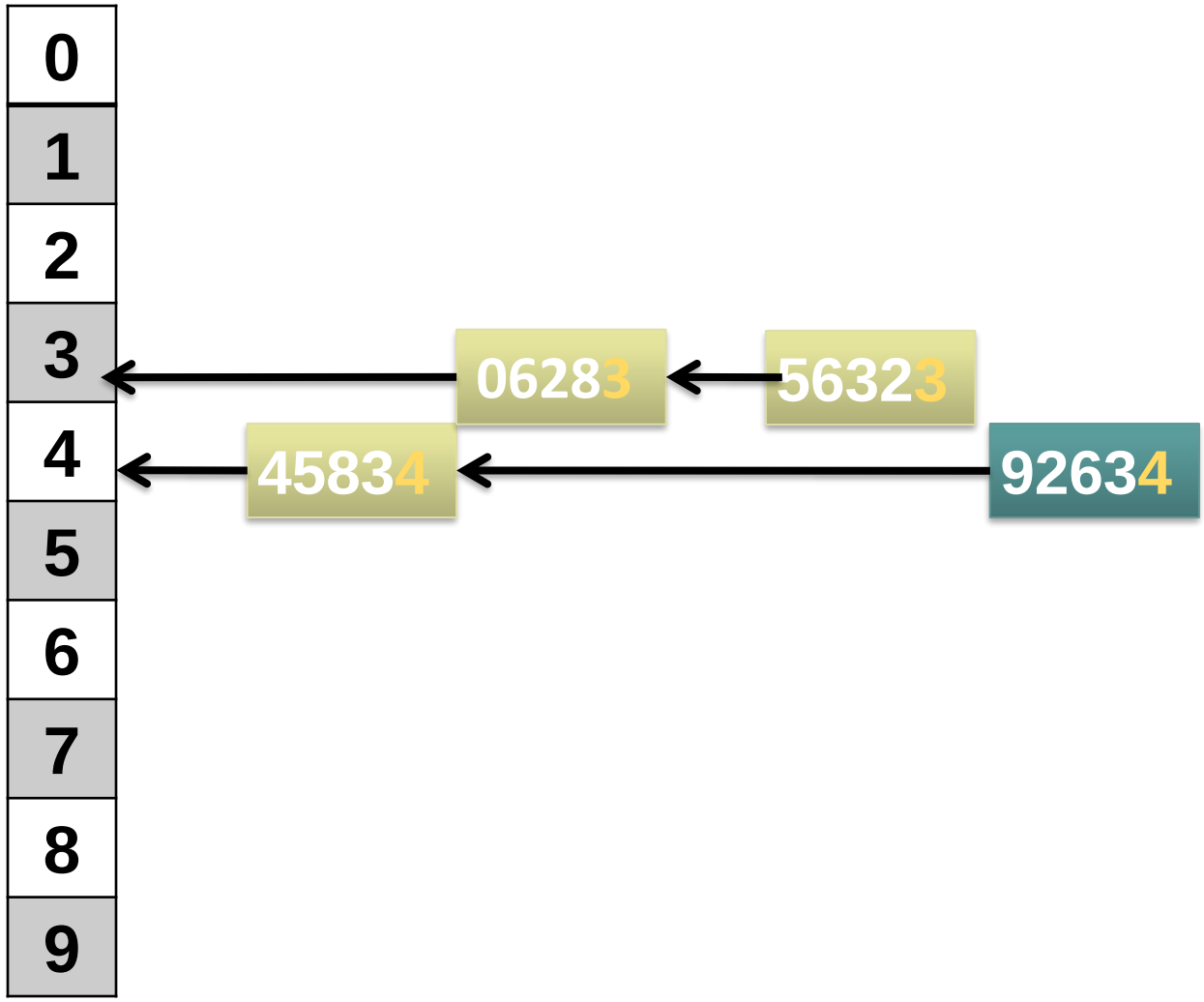
1. Place all the integers in the main queue.
2. Remove each value in the main queue and place it in a digit queue corresponding to the digit being considered, starting with the least significant digit.
3. Once all the values are placed in the appropriate digit queue, collect the values from queue 0 to queue 9, and place them back in the main queue.
4. Repeat steps 2 and 3 with the tens digit, the hundreds digit, and so on. After the last digit is processed, the main queue contains the values in ascending order.

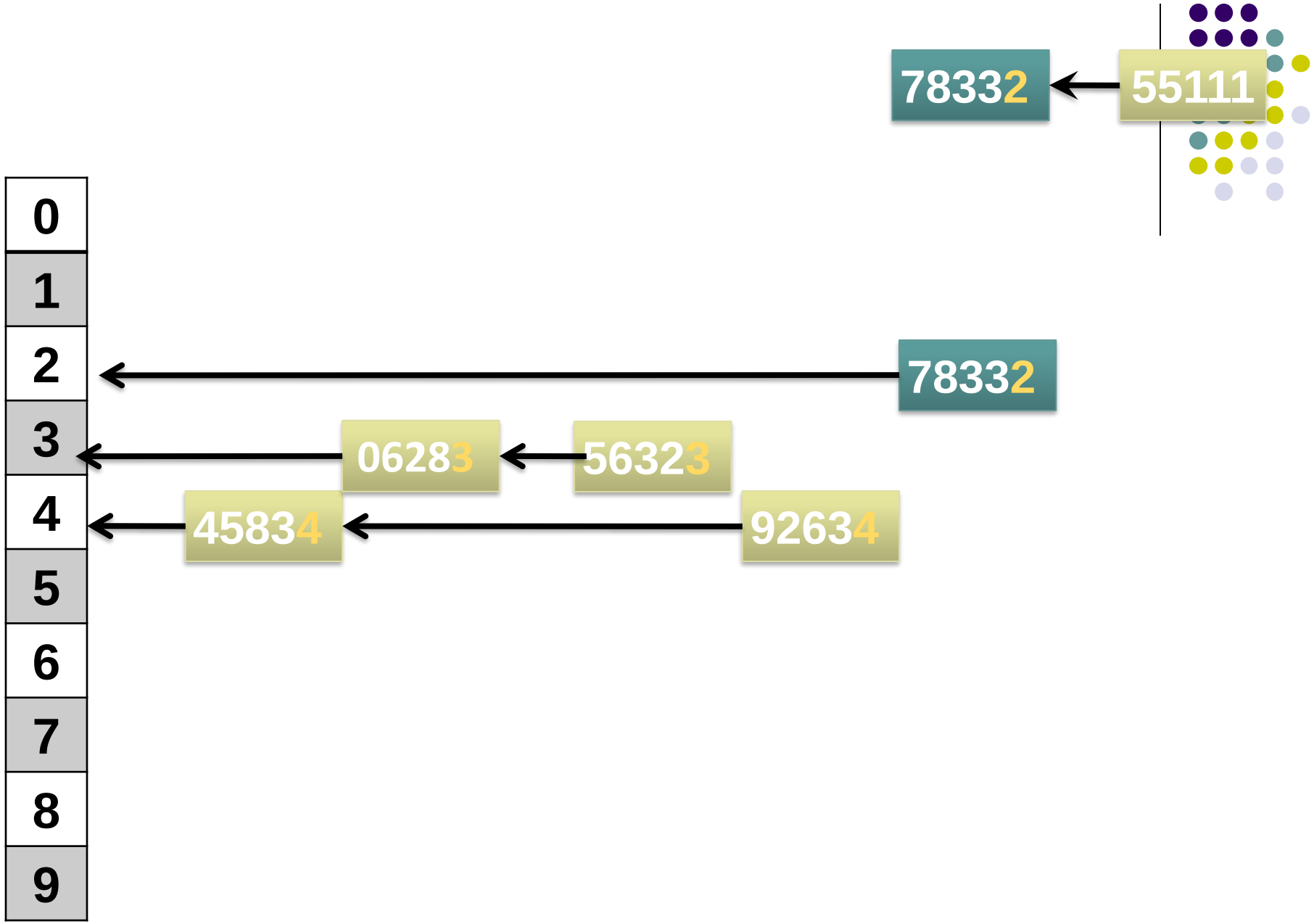

45834
06283
56323
92634
78332
55111
0
1
2
3
4
5
6
7
8
9

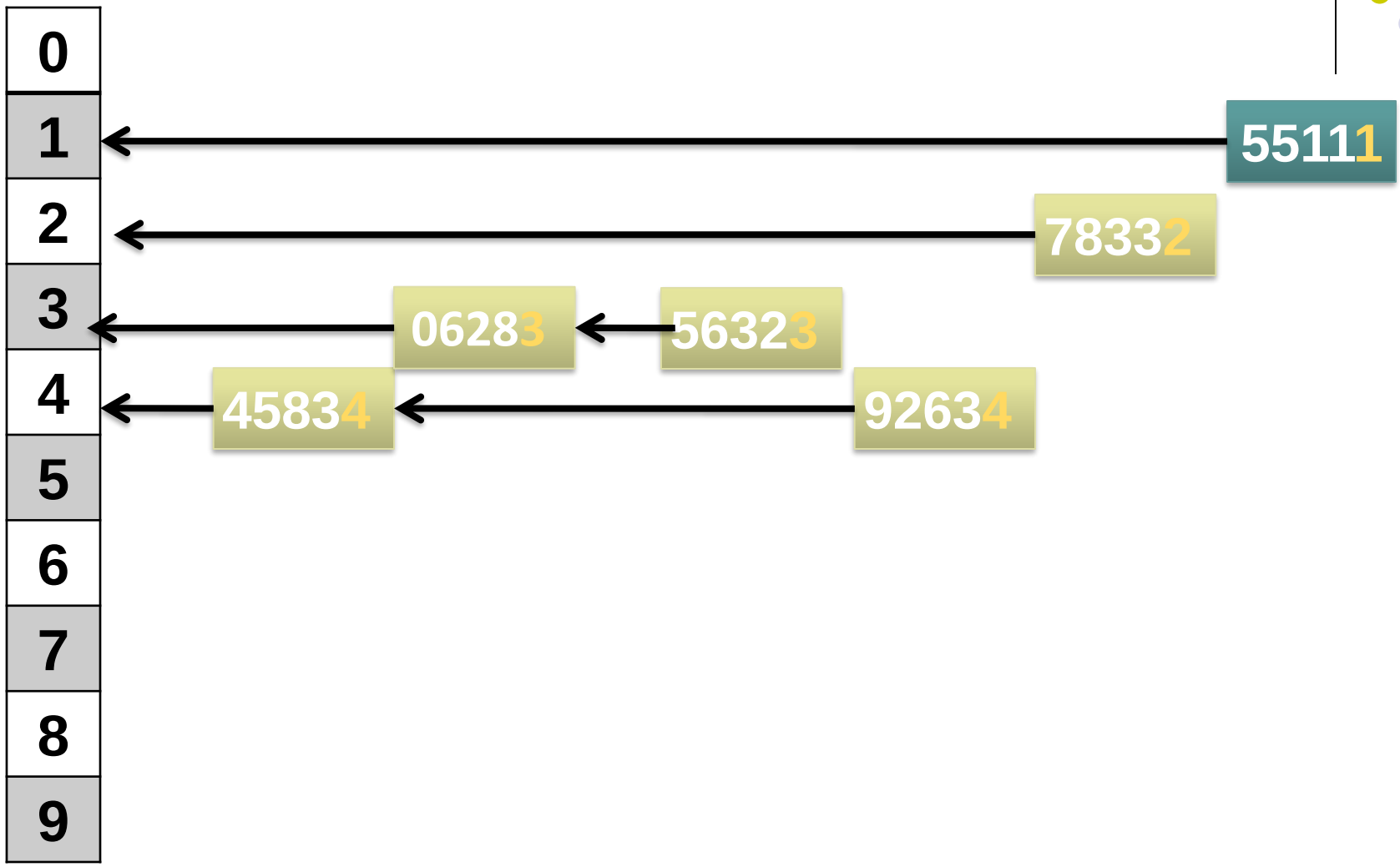

45834
06283
56323
92634
78332
55111
0
1
2
3
4
5
6
7
8
9
45834

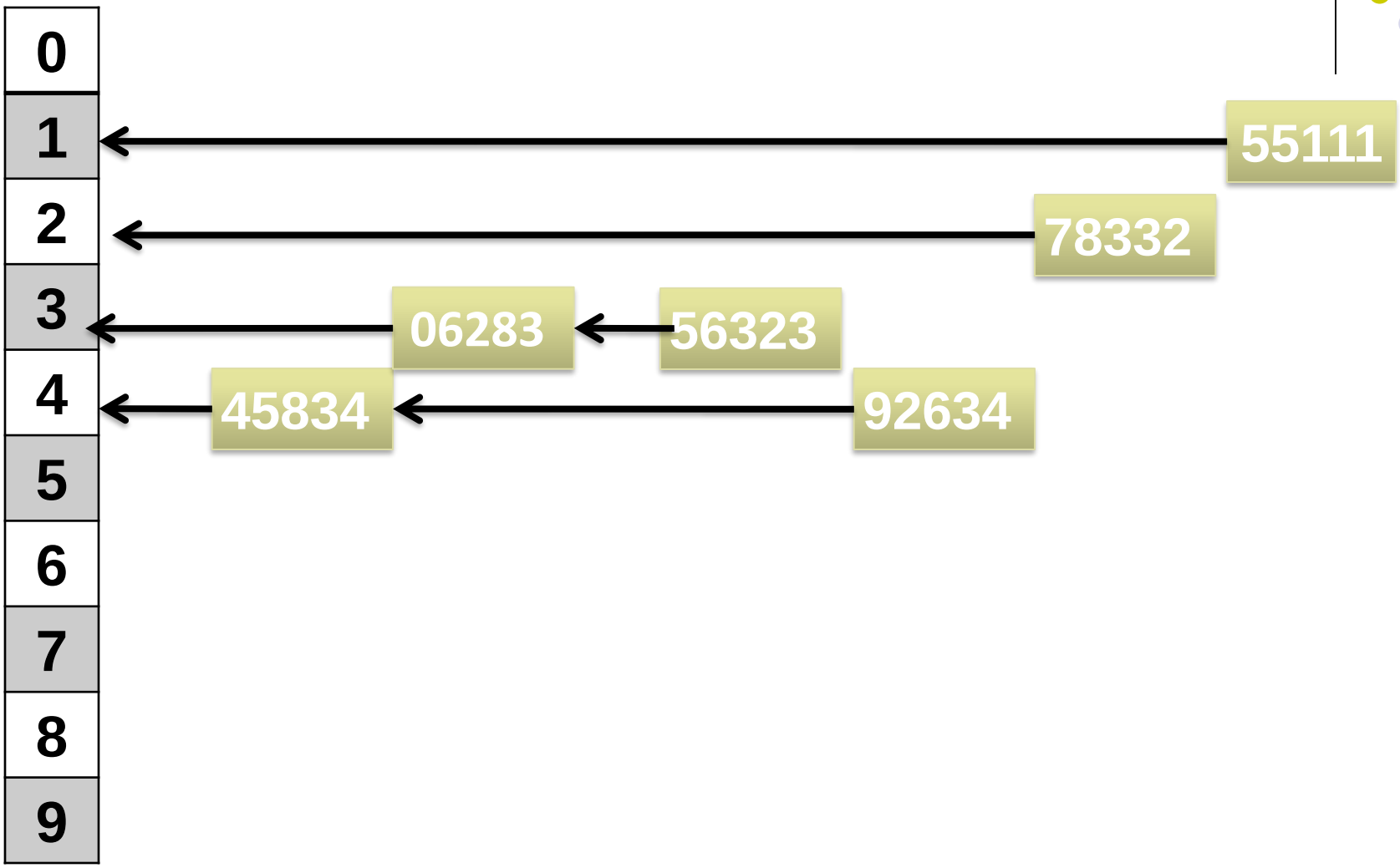

06283
56323
92634
78332
55111
0
1
2
3
4
5
6
7
8
9
06283
45834


56323
92634
78332
55111
0
1
2
3
4
5
6
7
8
9
06283
56323
45834

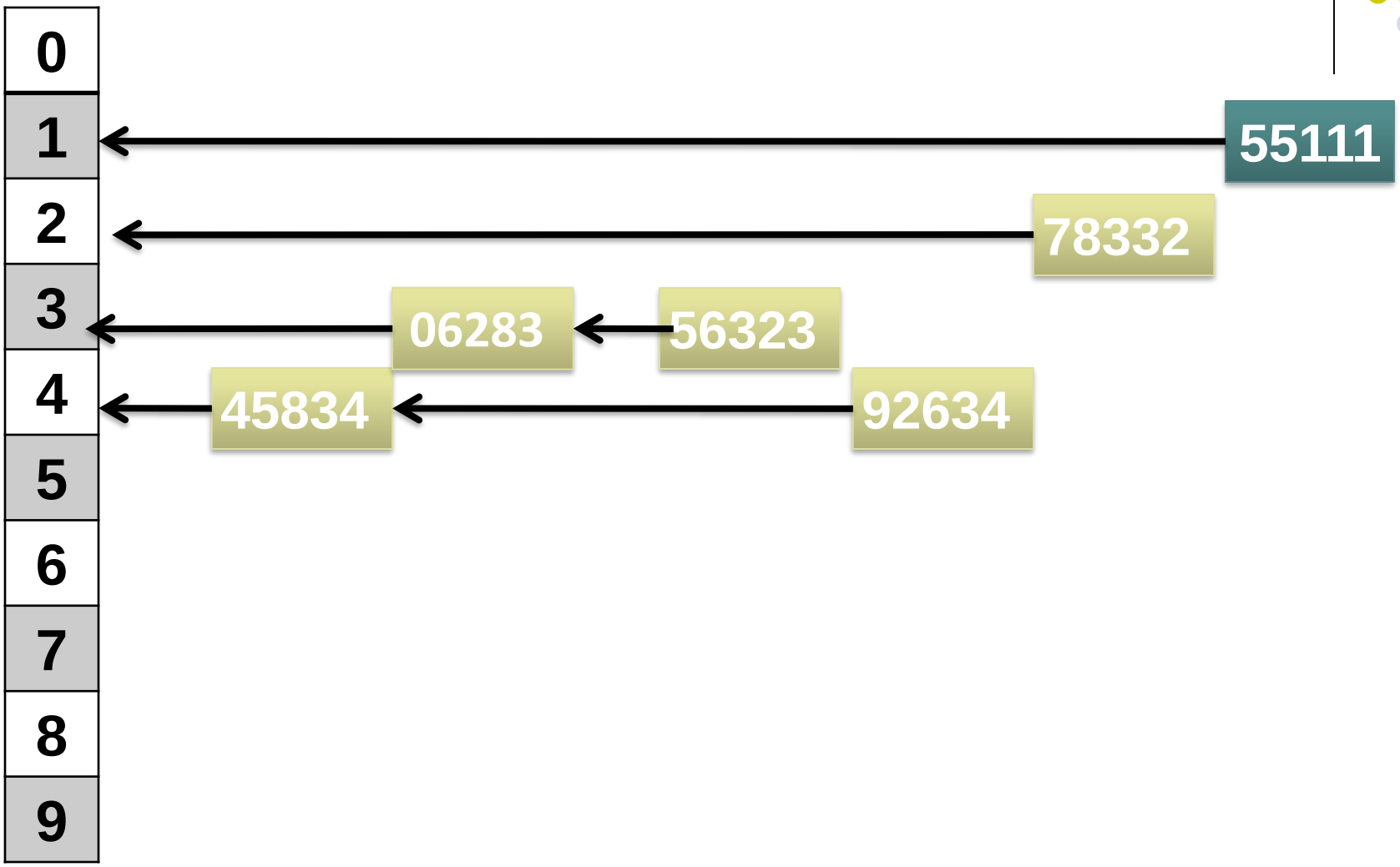

92634
78332
55111
0
1
2
3
4
5
6
7
8
9
06283
56323
45834
92634

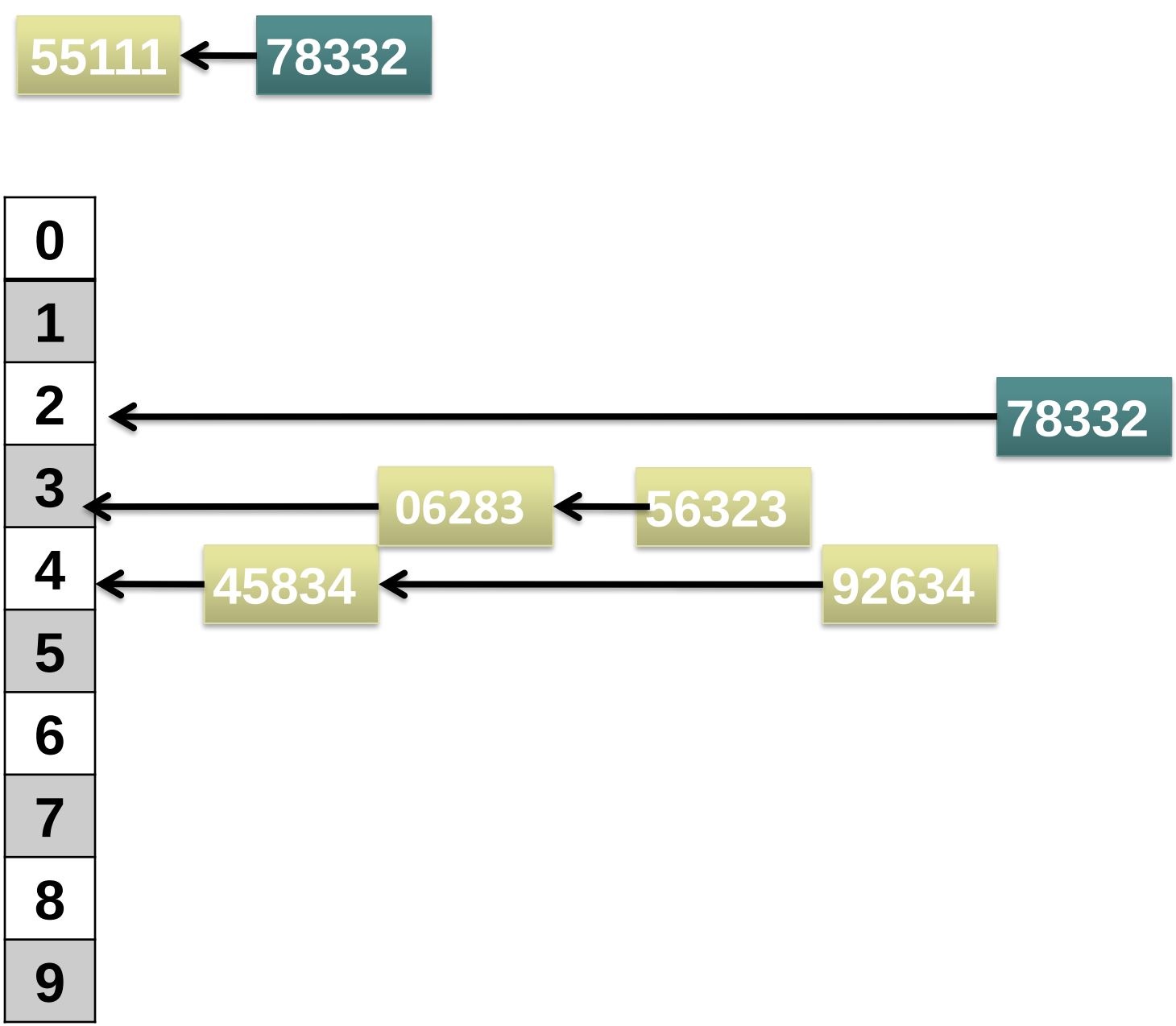

78332
55111
0
1
2
3
4
5
6
7
8
9
78332
06283
56323
45834
92634

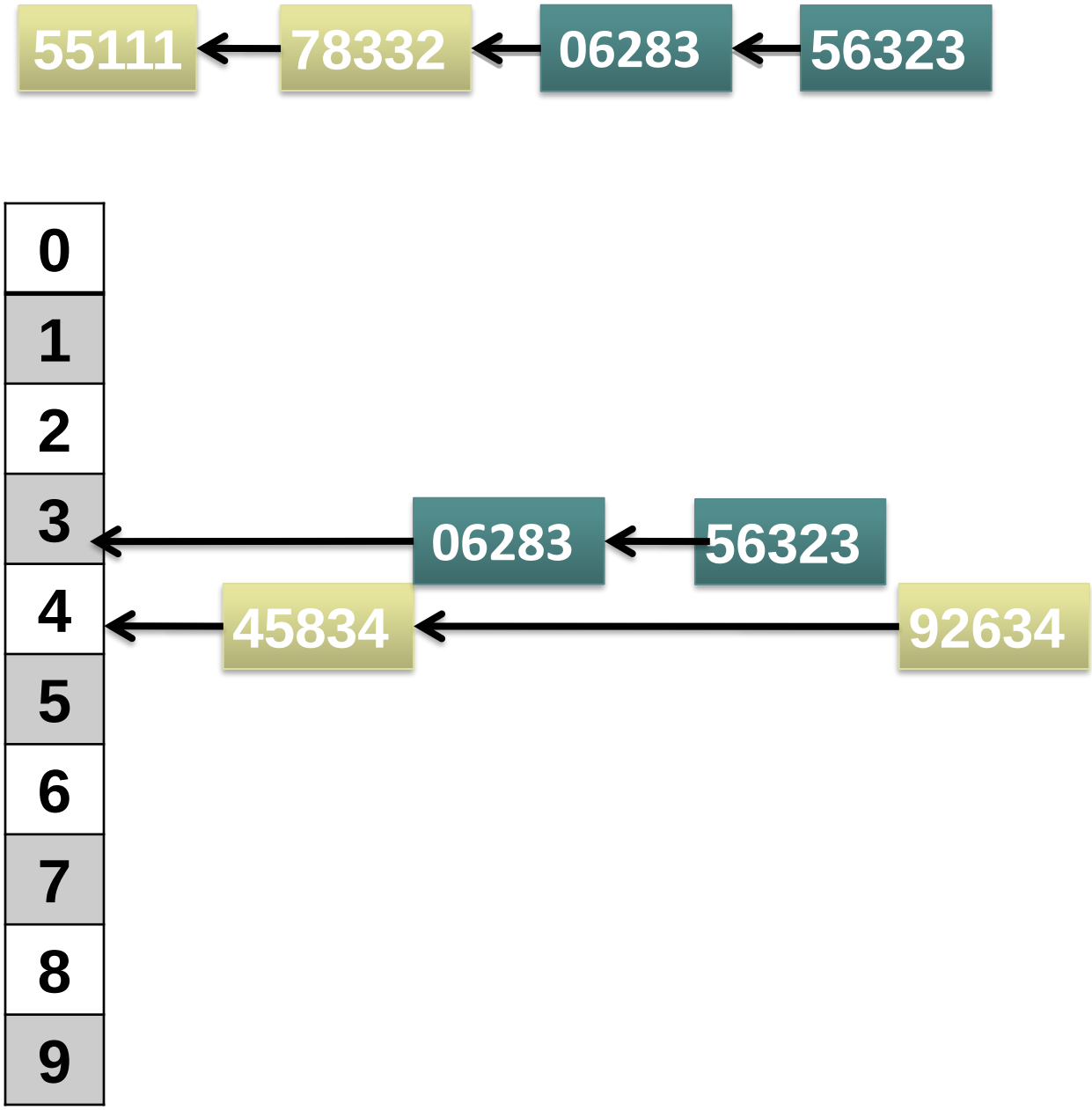

55111
0
1
2
3
4
5
6
7
8
9
55111
78332
06283
56323
45834
92634

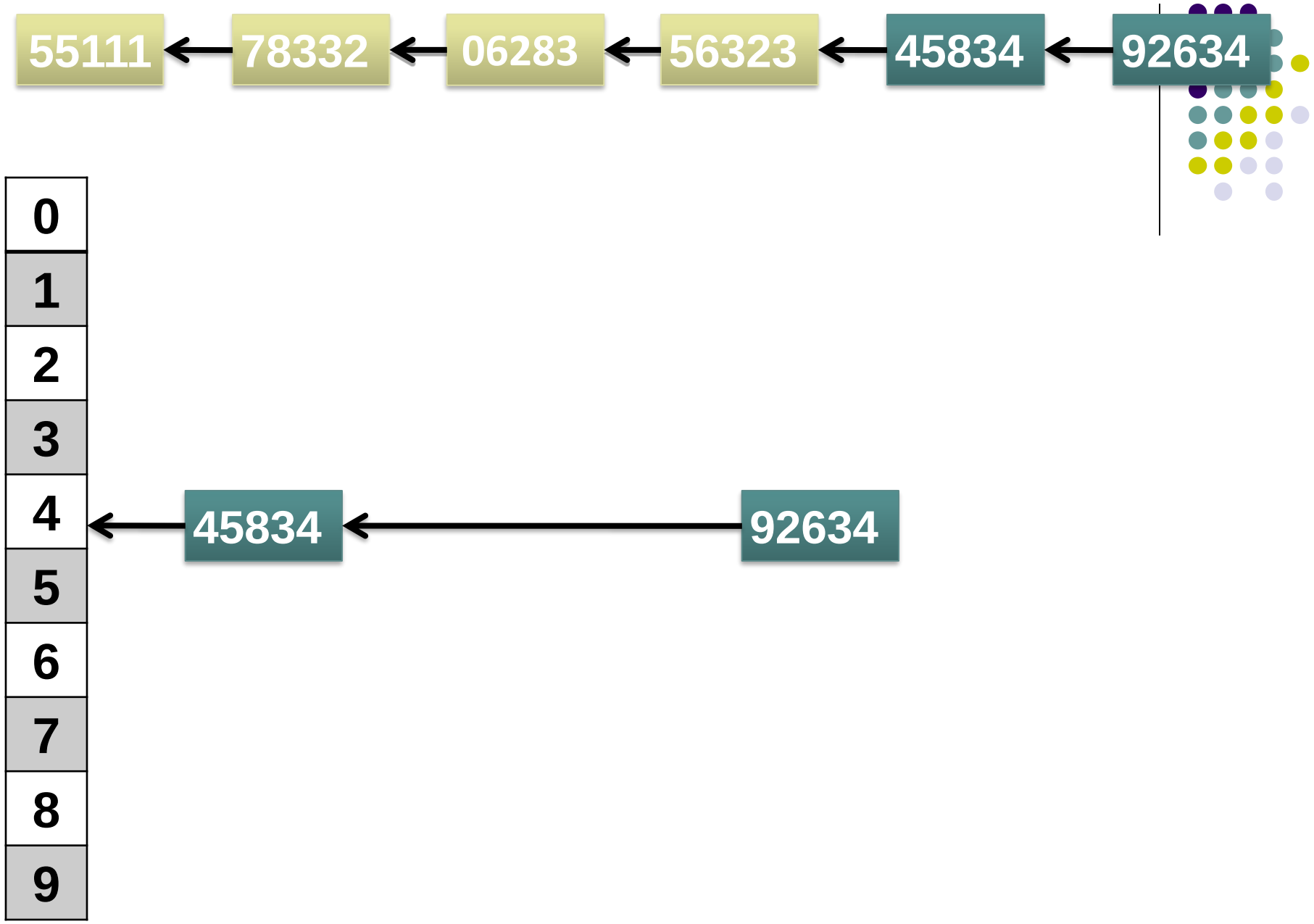

0
1
2
3
4
5
6
7
8
9
55111
78332
06283
56323
45834
92634

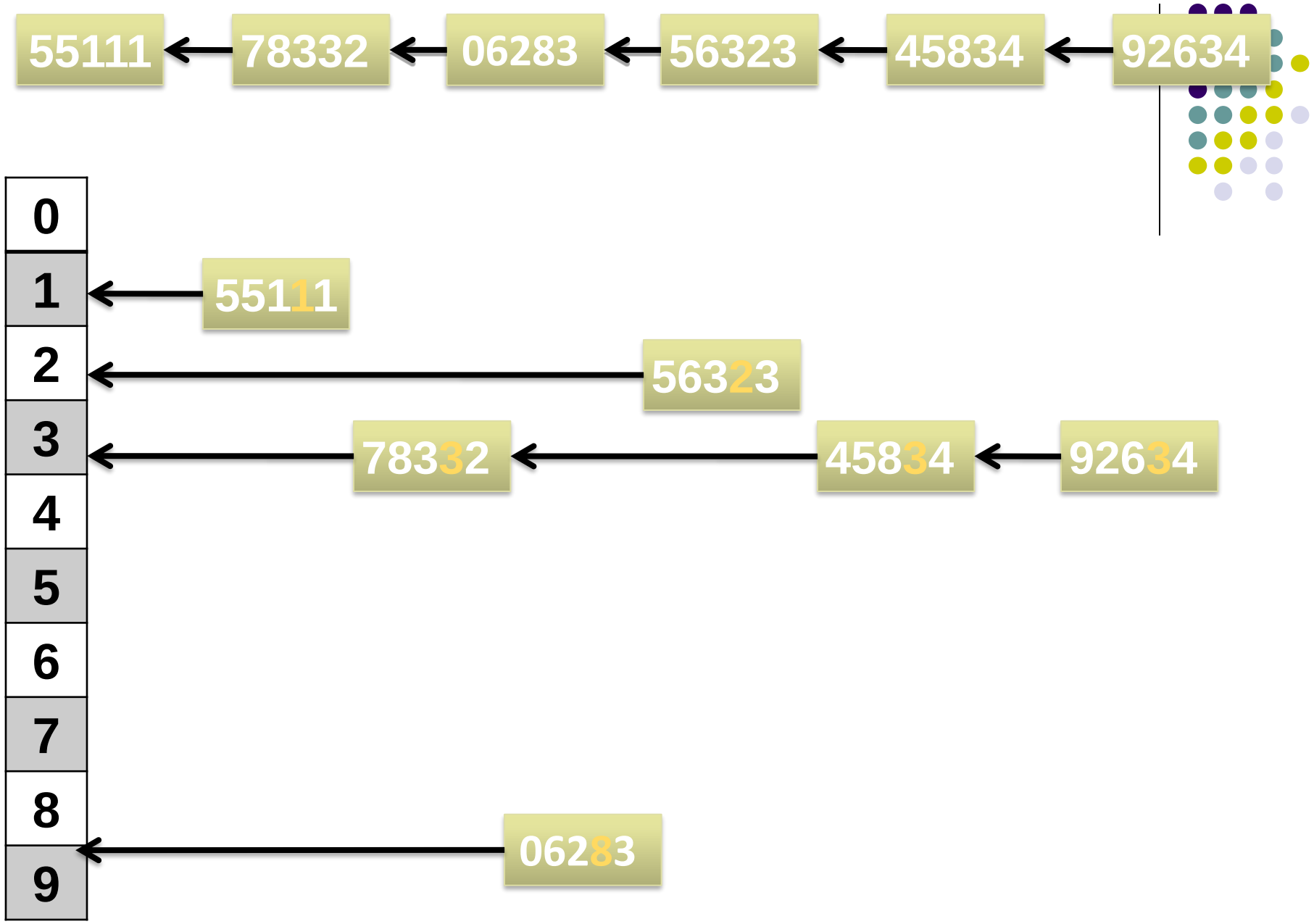
55111

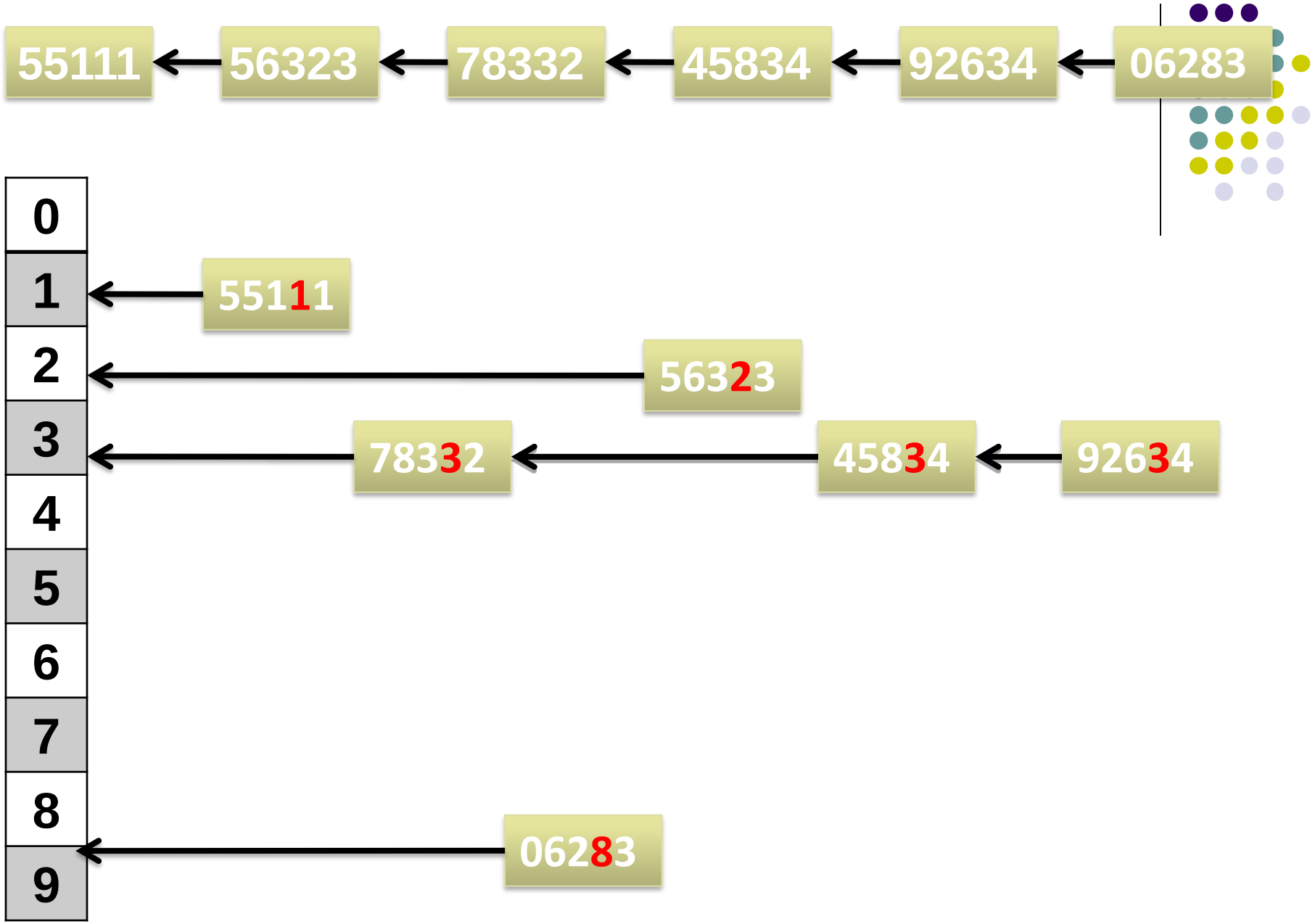


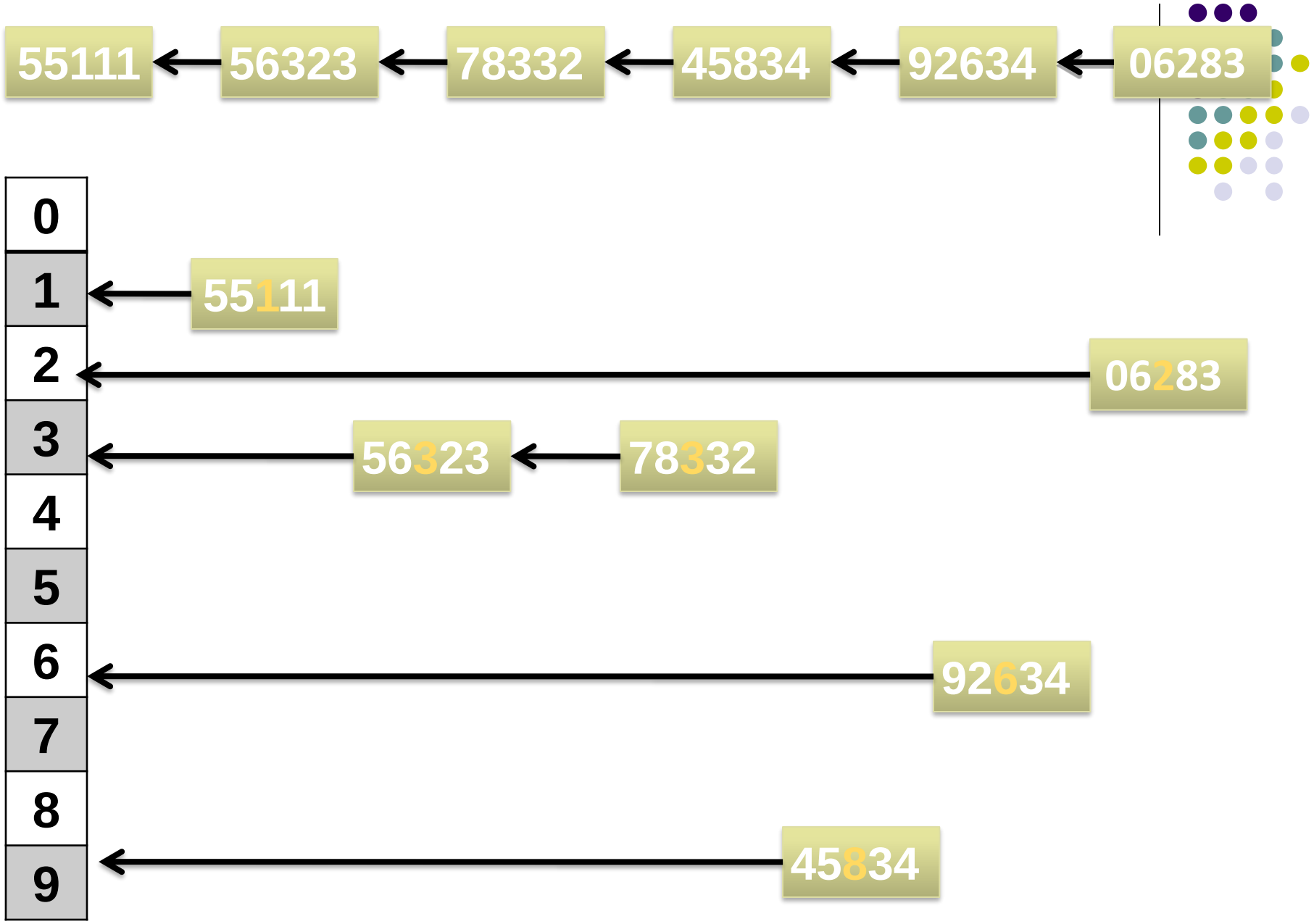

55111
78332
0
1
2
3
4
5
6
7
8
9
78332
06283
56323
45834
92634

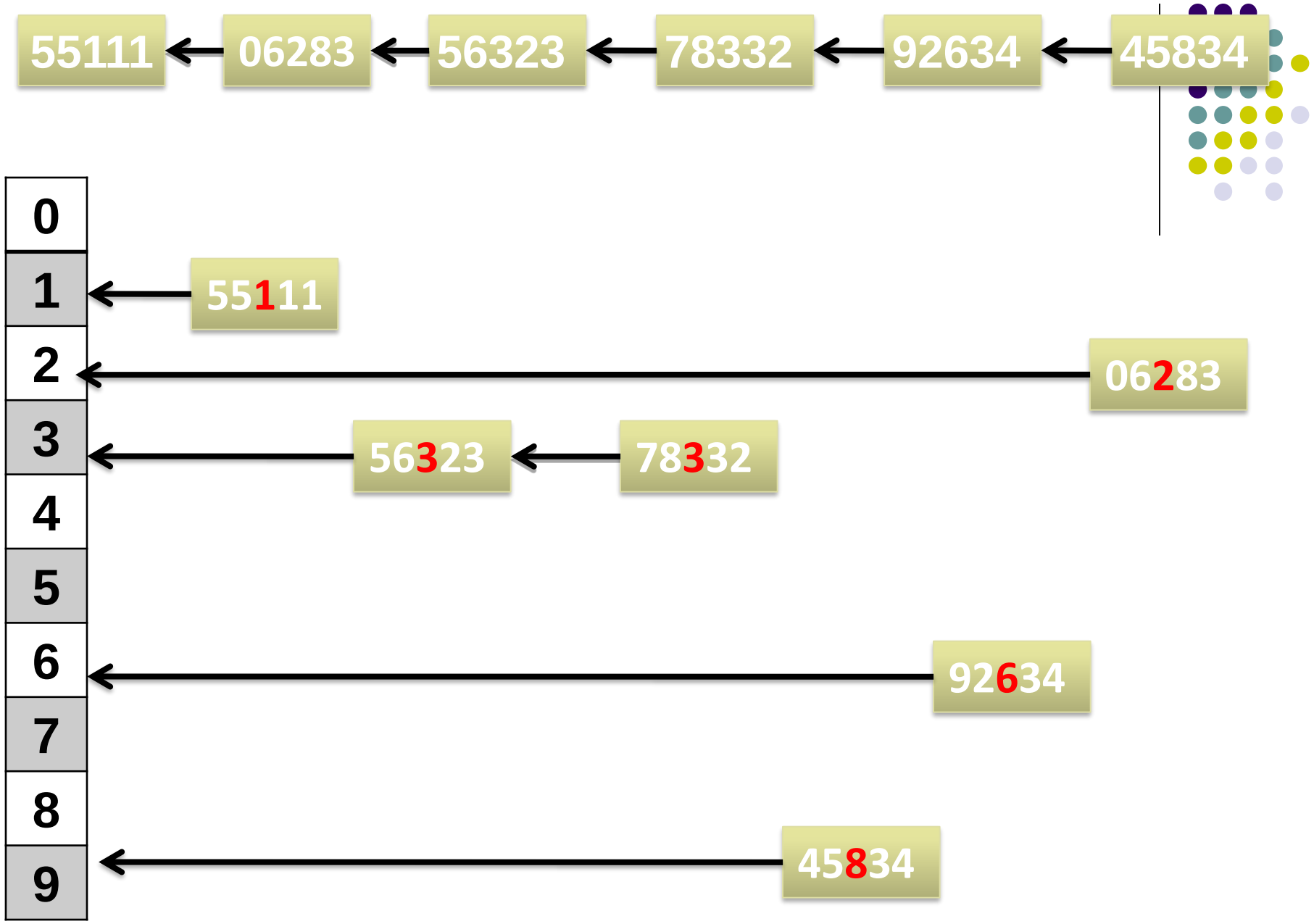

55111
78332
06283
56323
0
1
2
3
4
5
6
7
8
9
06283
56323
45834
92634

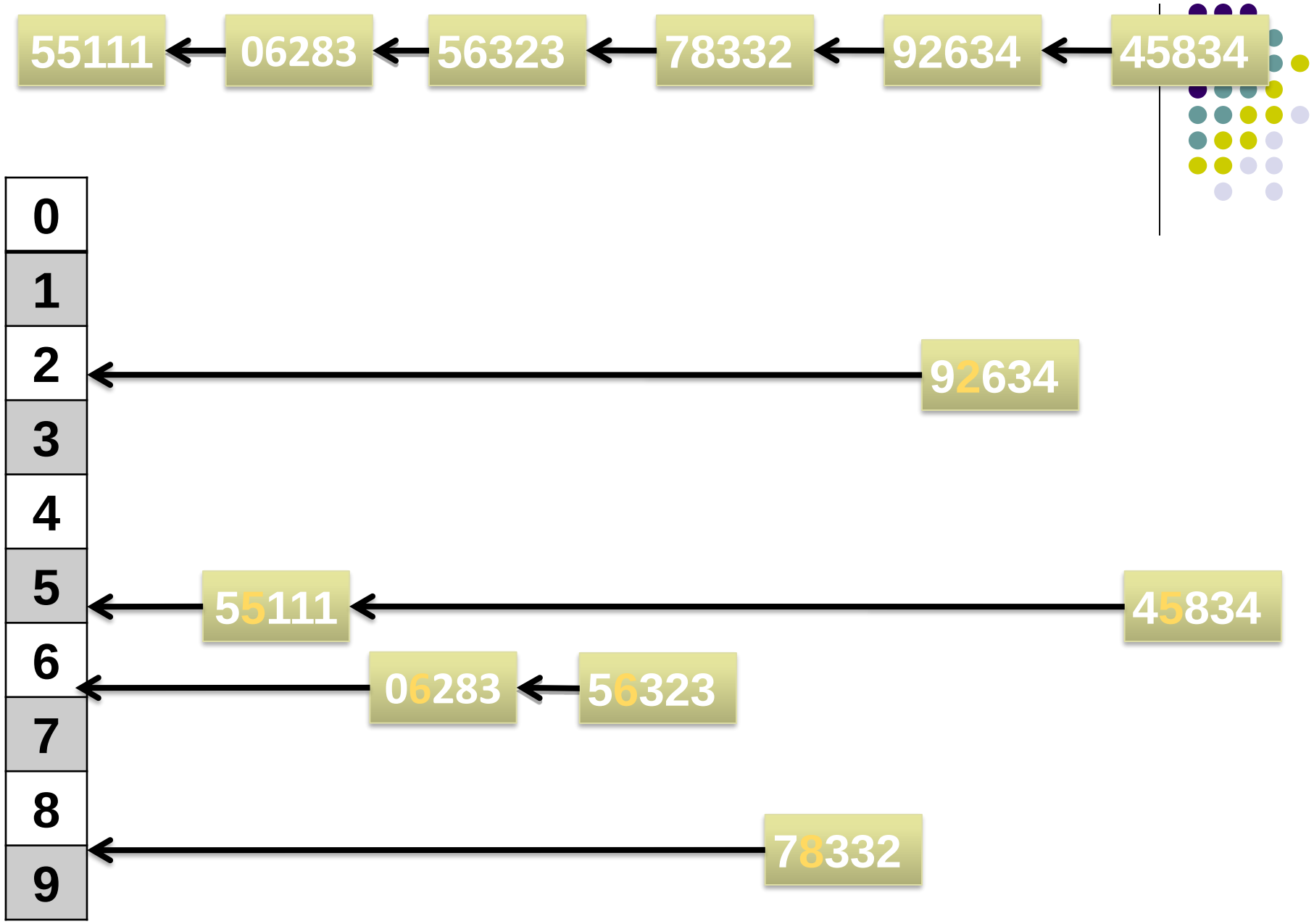

55111
78332
06283
56323
45834
92634
0
1
2
3
4
5
6
7
8
9
45834
92634

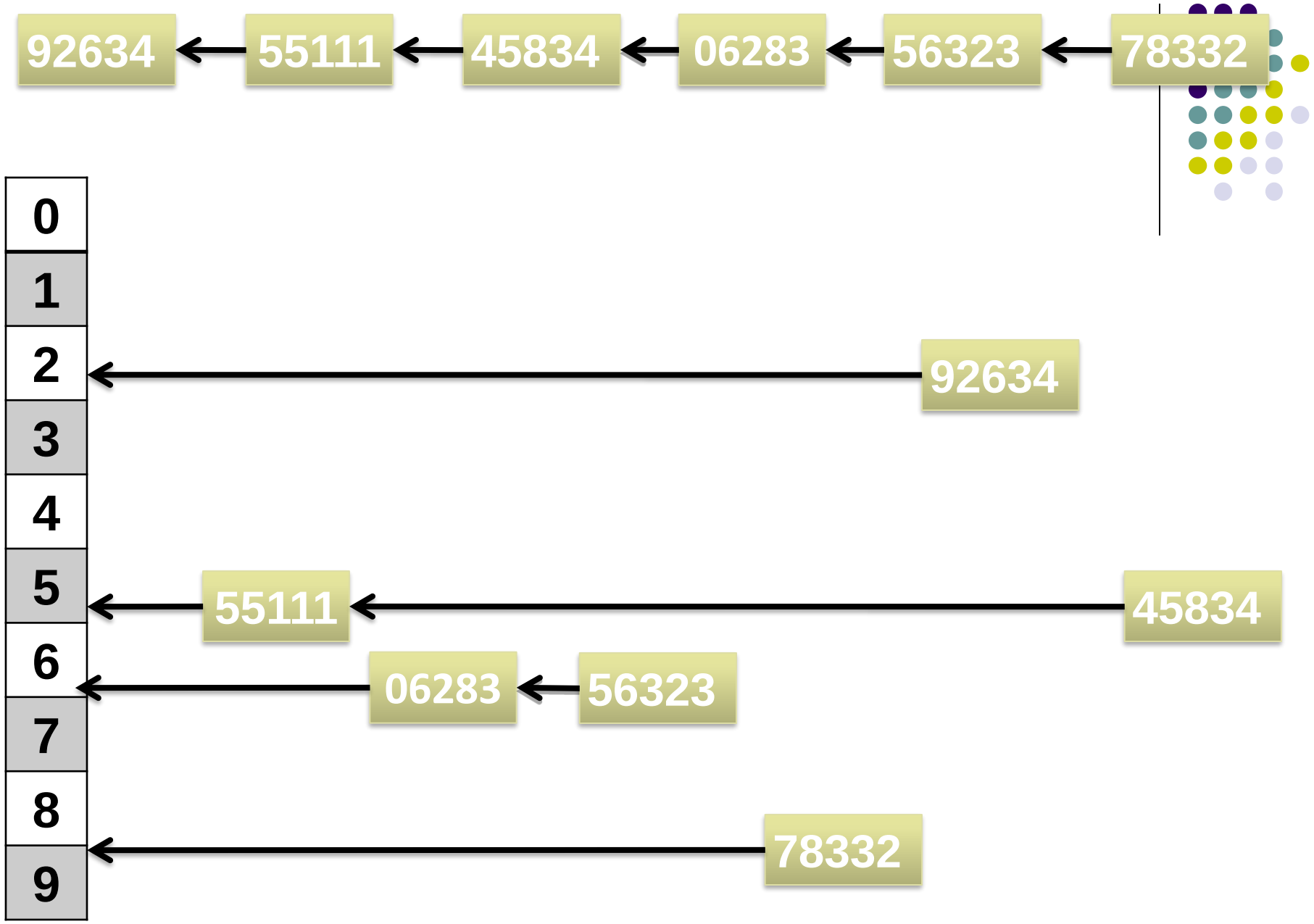

55111
78332
06283
56323
45834
92634
0
1
2
3
4
5
6
7
8
9
55111
56323
78332
45834
92634
06283

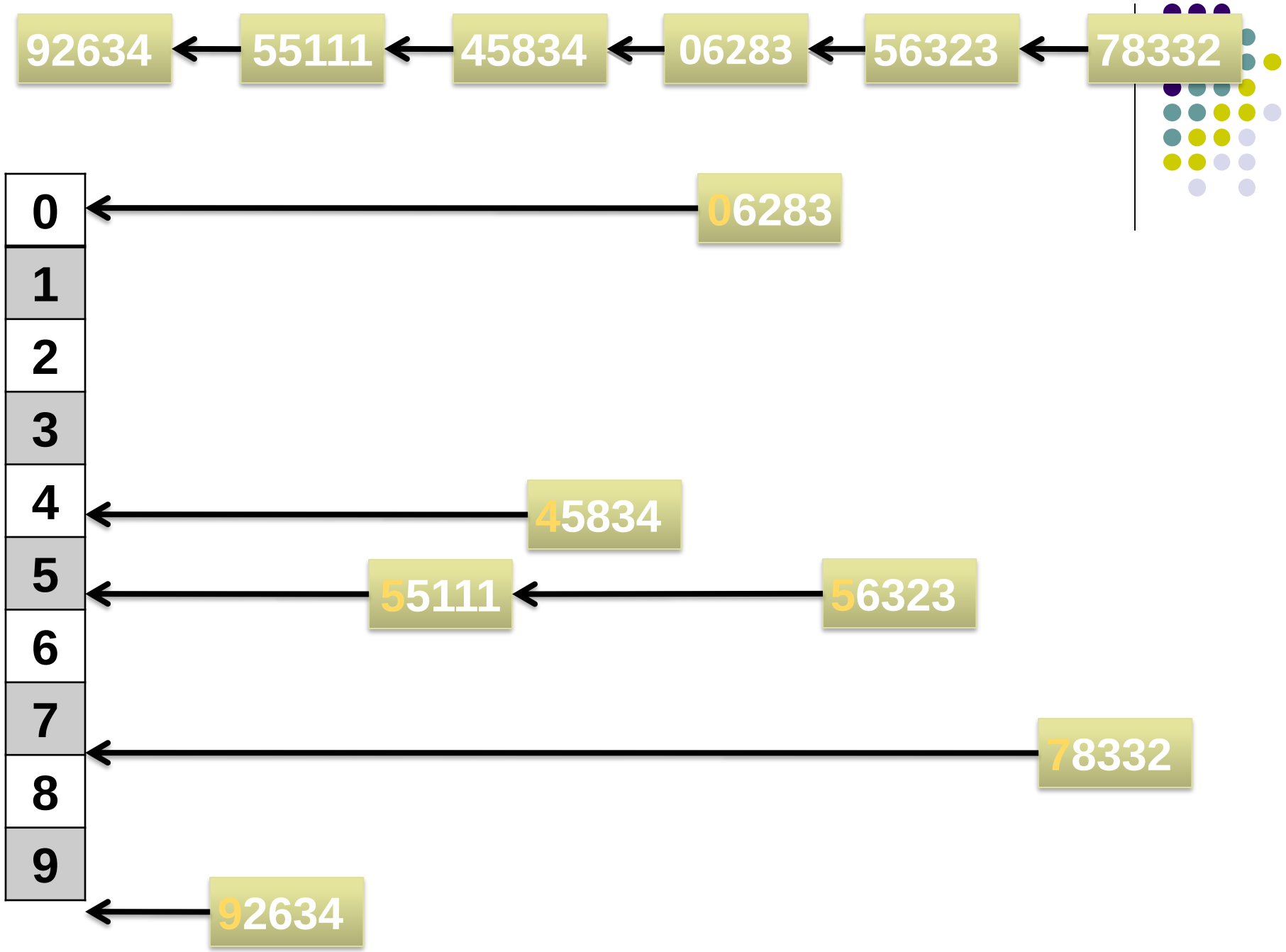

55111
56323
78332
45834
92634
06283
0
1
2
3
4
5
6
7
8
9
55111
56323
78332
45834
92634
06283

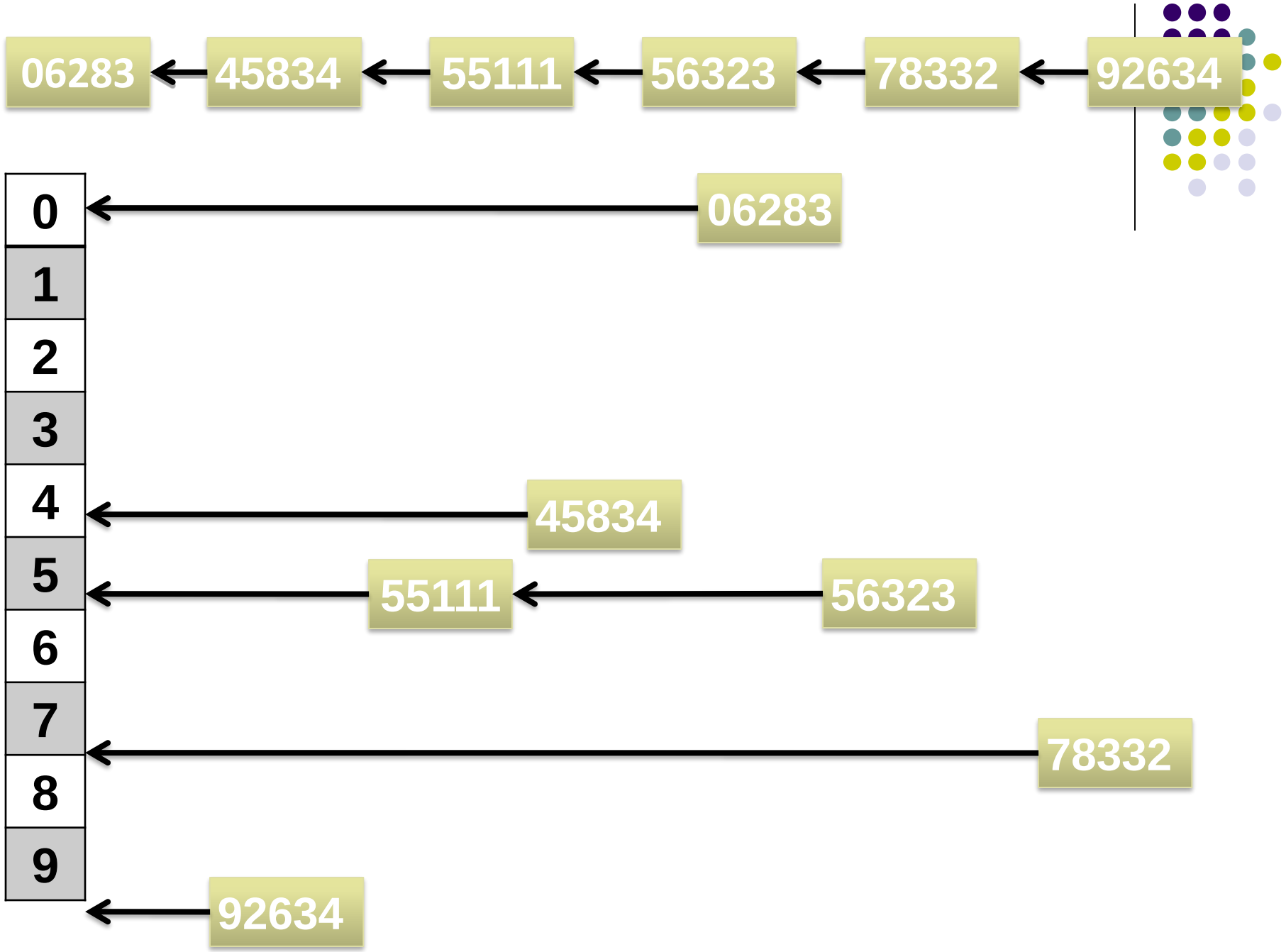

55111
56323
78332
45834
92634
06283
0
1
2
3
4
5
6
7
8
9
55111
06283
56323
78332
92634
45834

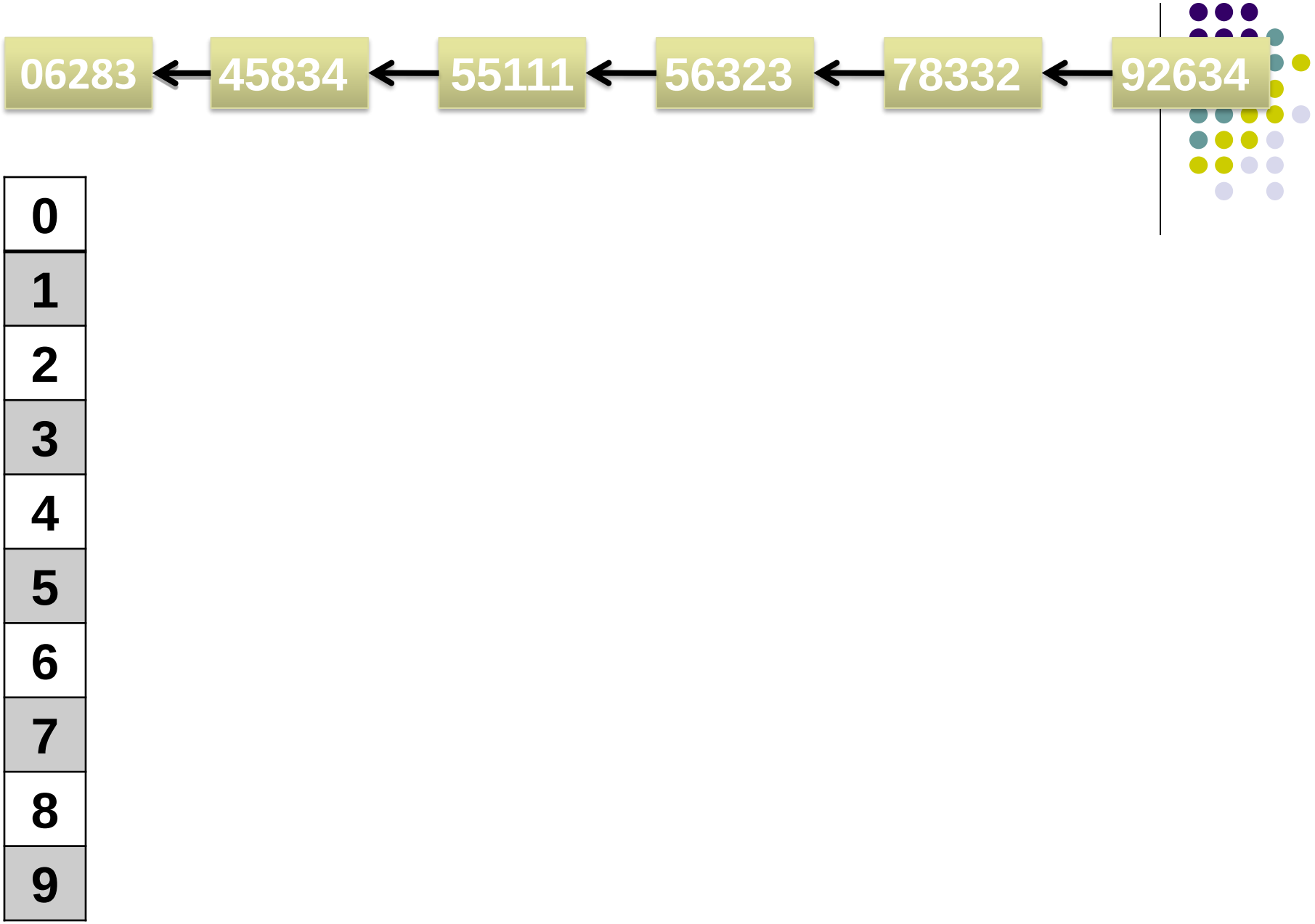

55111
06283
56323
78332
92634
45834
0
1
2
3
4
5
6
7
8
9
55111
06283
56323
78332
92634
45834


55111
06283
56323
78332
92634
45834
0
1
2
3
4
5
6
7
8
9
92634
55111
45834
06283
56323
78332


92634
55111
45834
06283
56323
78332
0
1
2
3
4
5
6
7
8
9
92634
55111
45834
06283
56323
78332


92634
55111
45834
06283
56323
78332
0
1
2
3
4
5
6
7
8
9
06283
45834
55111
56323
78332
92634


06283
45834
55111
56323
78332
92634
0
1
2
3
4
5
6
7
8
9
06283
45834
55111
56323
78332
92634


06283
45834
55111
56323
78332
92634
0
1
2
3
4
5
6
7
8
9