数据结构与算法

复习、巩固、联动

Cat 1.04

斐波那契数列

- 假设一对兔子每年生一对孩子
- 兔宝宝在两年之后才能长大，才能有自己的孩子
- 兔子永远不会死

N年后会有多少只兔子?

F(n) = F(n-1) + F(n-2)

1 1 2 3 5 8 13 21

如何计算 F(n)?

斐波那契数列

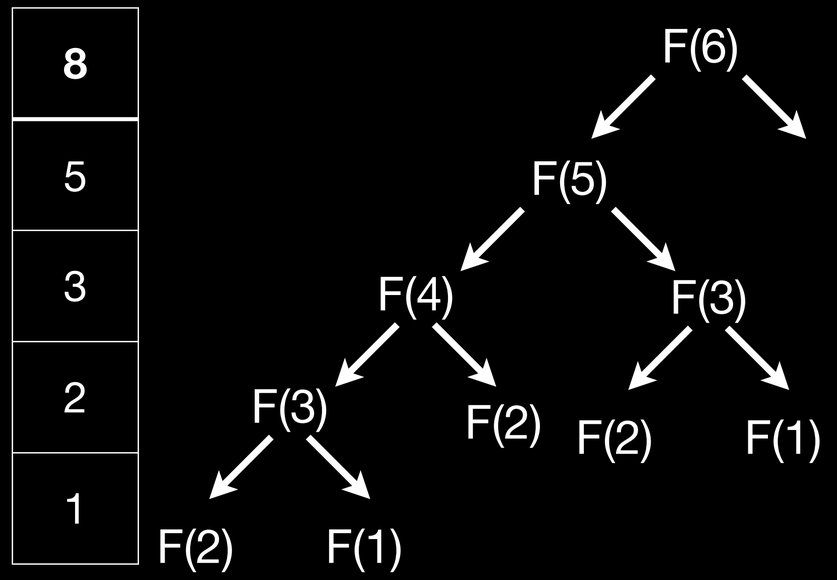
```
int fib(int n)
{
    if (n <= 2) return 1;
    else return fib(n-1) + fib(n-2);
}
```

- 这个算法需要多长时间?
- 这个算法需要多长性能?
- 以执行的指令为标准

斐波那契数列

- 假设一对兔子每年生一对孩子
- 兔宝宝在两年之后才能长大，才能有自己的孩子
- 兔子永远不会死

N年后会有多少只兔子?



一共执行13次 执行n次 = F(n) + 2F(n) - 2

DP

为什么会这么慢，原因在于不断地一遍又一遍地重复计算已经知道答案的子问题（overlap sub-problem）

```
int fib(int n)
{
    int f[n+1];
    f[1] = f[2] = 1;
    for (int i = 3; i <= n; i++)
        f[i] = f[i-1] + f[i-2];
    return f[n];
}
```

执行n次 =n-1 + n-2 + 3 = 2n

降低空间复杂度

需要去分析程序使用的内存量。一个程序虽然花费了很多时间，但是仍然可以运行它，只是需要等待更长的时间。但是，如果一个程序占用大量内存，可能根本无法运行它。需要降低空间复杂度：

```
int fib3(int n)
{
   int a = 1, b = 1;
   for (int i = 3; i <= n; i++) {
       int c = a + b;
       a = b;
       b = c;
 }
 return b;
```

执行n次 = 4n - 3

大O符号

算法3比算法2慢一些，但是如果把算法3放到CPU更快的计算机上执行呢?

所以需要有一种确切的方式描述我们对算法分析的结构：大O符号!

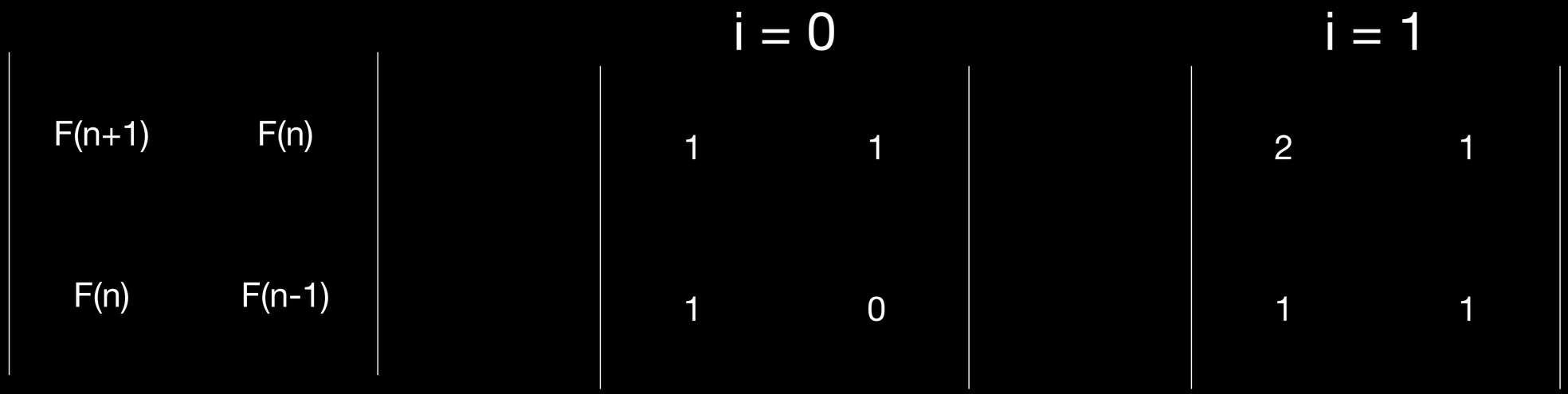
O符号的意义何在?

允许我们对算法行为的所有细节不那么在意?

像算法3虽然比算法2慢一些，但是实际开发中算法 2 中分配数组也需要一部份时间。可能意味着在实际时间中，算法彼此更接近。

但是对于算法1和算法3，会很明显得出4n 都比 3F(n)-2 好得多。

降低时间复杂度



降低时间复杂度

- n = 9
- $2^9 = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 = 512$
- $2^2 = (2^1)^2 = 4$
- $4^2 = (2^2)^2 = 16$
- $16^2 = (2^4)^2 = 256$
- $256 * 2 = 512$
- $2^n = (2^m)^2$ 如果n是偶数，m = n/2
- $2^n = (2^m)^2 * 2$ 如果n是奇数，m = n/2

通项公式

线性递推数列

$$a_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right].$$

总结

- 先分析，分析啥？想啥就分析啥
- 先按自己能理解的去实现
- 观察，优化，再分析
- 能力范围之内实现的有哪些？
- 能力范围之外还有哪些好方式？学习，记录
- 实际应用？结合具体场景分析
- 并不一定最优秀的算法就是最合适的

Linked List & Array区别

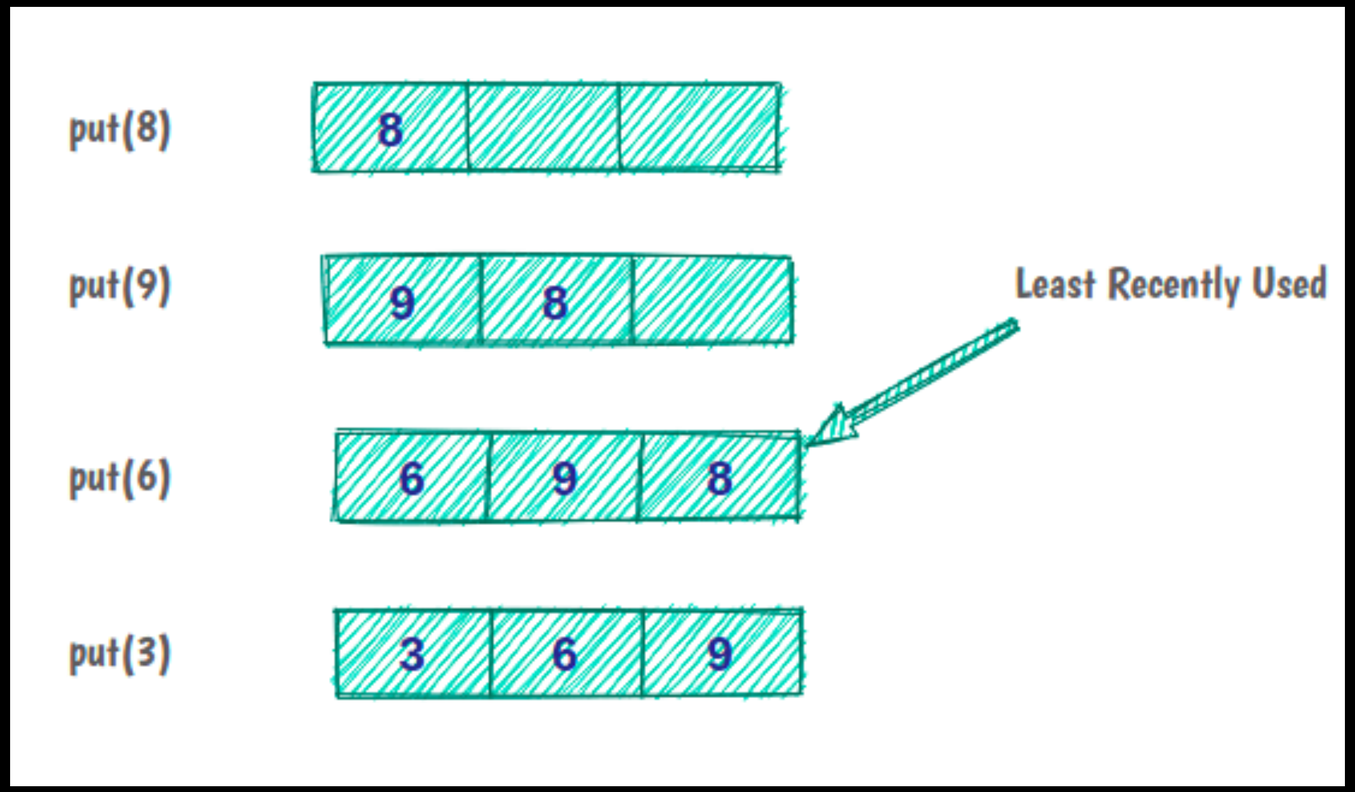
Linked List	Array
数据可以分散存储	只能存储在连续的内存中，大小固定
运行时可以动态改变大小	一般情况下需要提前开辟固定内存空间
相同单一数据，链表结点占用内存更多	占用内存小
整体占用内存小	需提前分配，分配内存又可能未使用
访问速度慢，需遍历先前结点，不能随机访问	通过索引直接访问，连续内存，访问快
修改某一结点慢，插入删除快	修改快，插入删除慢，需开辟新空间，移动元素
使用指针难度高，寻址困难	直接操作地址，寻址方便
不能使用相关查找算法	可以使用查找算法

数组特点：查找容易，插入和删除困难

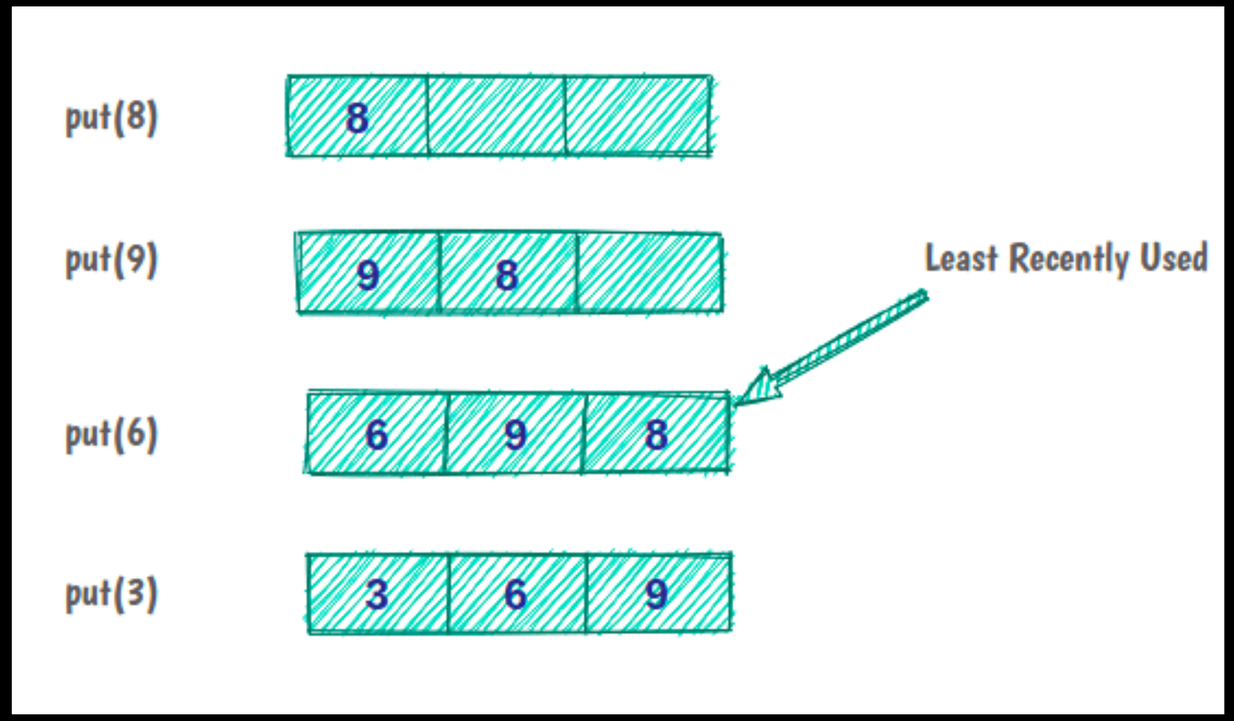
链表特点：查找麻烦，插入和删除容易

LRU

- LRU(最近最少使用)，是一种缓存淘汰算法。顾名思义，最长时间没有被使用的元素会被从缓存中淘汰
- 例如，如果我们有一个容量为3的缓存：



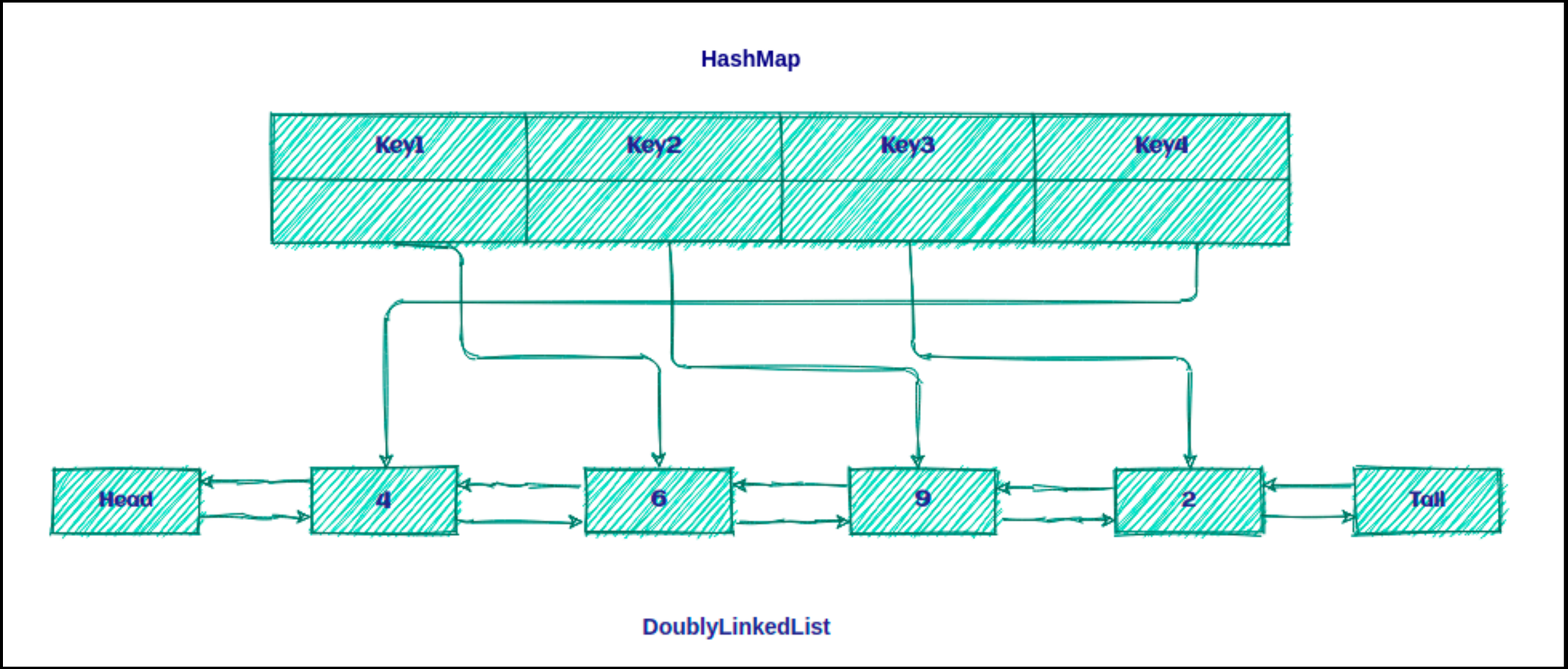
LRU



- Queue（队列）：此队列将具有特定容量，因为缓存的大小有限。当引入一个新元素时，它就会被添加到队列的尾部。需要淘汰的元素放在队列的头部；
  - 先进先出
  - 每一项存储的是key-value，通过key取value时，需要遍历
  - 时间复杂度是：O(n)
  - 空间复杂度是：O(n)
  - 如何减小缓存命中时间，减小到O(1)?

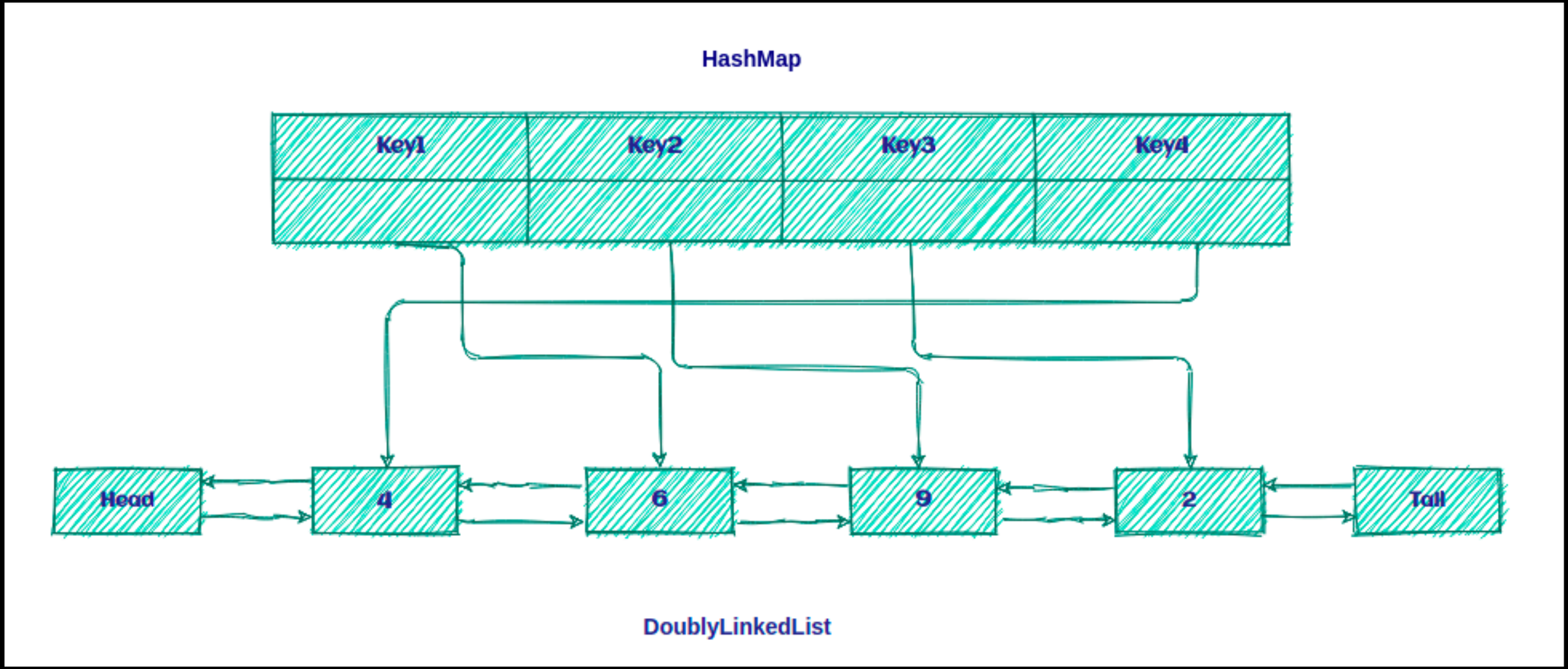
LRU

- 通过链表与Hash表。Hash表用来存储key-value，减小缓存命中时间。 链表负责管理缓存元素的顺序
  - 时间复杂度是：O(1)
  - 空间复杂度是：O(n)



LRU

- 通过双向链表与Hash表。Hash表用来存储key-value，减小缓存命中时间，但哈希表是无序的。 所以使用链表负责管理缓存元素的顺序
  - 单向链表？删除节点需要访问前驱节点，O(n)
  - 双向链表，结点有前驱指针，删除/移动节点都是纯指针变动，O(1)



LRU

问题来了，是不是LRU缓存算法一定要使用双向链表和哈希表？如果数据量很小呢?

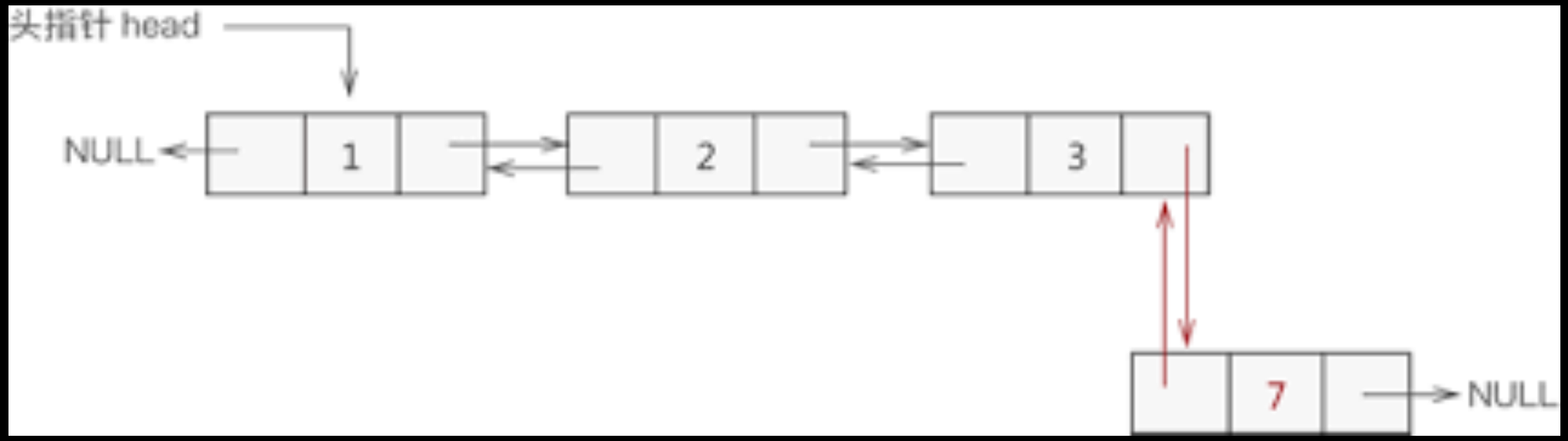
LRU


颜色种类
#FF00FF
red
SYColor
rgb(100%, 0%, 100%)
UIColor
rgba(255, 0, 255, 0.8)
blend color

Linked List

- 链表是动态的线性数据结构，不可以随机访问数据，需要从头遍历。链表中的元素通常不存储在连续的位置。所以链表中的结点通常有数据 + 指针组成
- 链表种类：
    - 单链表：结点 = 数据 + 指针。指针指向下一个结点。单链表中的第一个结点和最后一个结点，分别叫做头结点和尾结点，只能从前往后遍历。尾结点是一个特殊的节点，它的指针总是指向NULL，代表结尾。头节点分为两种：
        - 有虚拟头结点
        - 无虚拟头结点
        - 虚拟头结点使用最多，方便统一结点操作方式
    - 双向链表：结点 = 前指针 + 数据 + 后指针，前指针和后指针分别指向当前结点的前趋结点和后继结点，允许我们从两个方向进行操作。头结点和尾结点都是虚拟结点
    - 循环链表：跟单链表相比，尾结点不在指向NULL，而是指向头结点
    - 对链表的操作有：
        - Insert插入结点
        - Delete删除结点
        - Search搜索结点

双链表



malloc & calloc

- malloc：分配给定大小（以字节为单位）的内存块并返回指针， 不会初始化分配的内存。这意味着如果这部分内存曾经被分配过，则其中可能遗留有各种各样的数据，读取的值有可能是垃圾值
  - memset：初始化内存，但会强制虚拟内存系统将相应的页面映射到物理内存中
- calloc：分配内存并将分配内存中的每个字节初始化为 0
  - 相当于malloc + memset，但是有优化
  - 不会立即分配请求的内存，而是映射到操作系统提供的零页（page）。当系统在使用页面时对其进行初始化，而不是一次全部初始化