前端算法和数据结构

1. 课前准备

2. 课堂主题

3. 课堂目标

4. 知识点

复杂度

空间复杂度，时间复杂度

数组

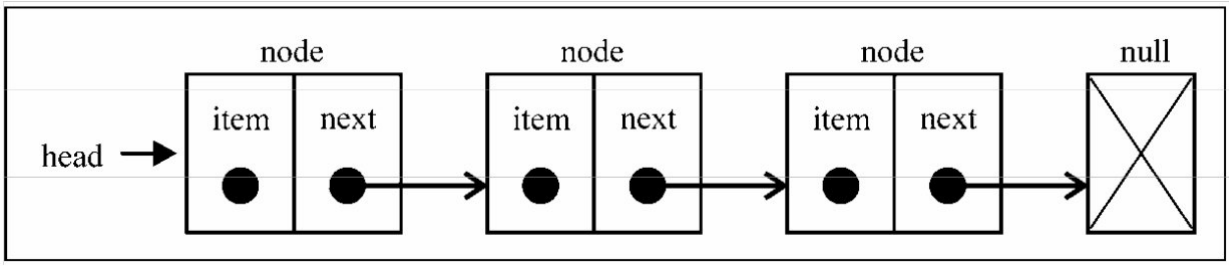
搜索复杂度

删除复杂度

新增复杂度

日常用的最多

链表



搜索复杂度

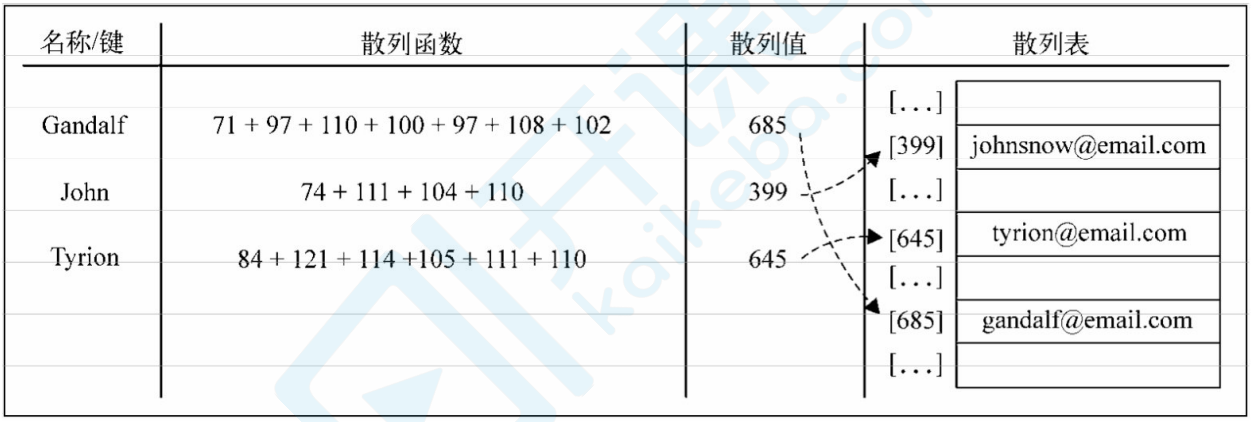
删除复杂度

新增复杂度

集合

hash表

js的对象，就是hashTable的一种实现



hash碰撞

存储复杂度

栈

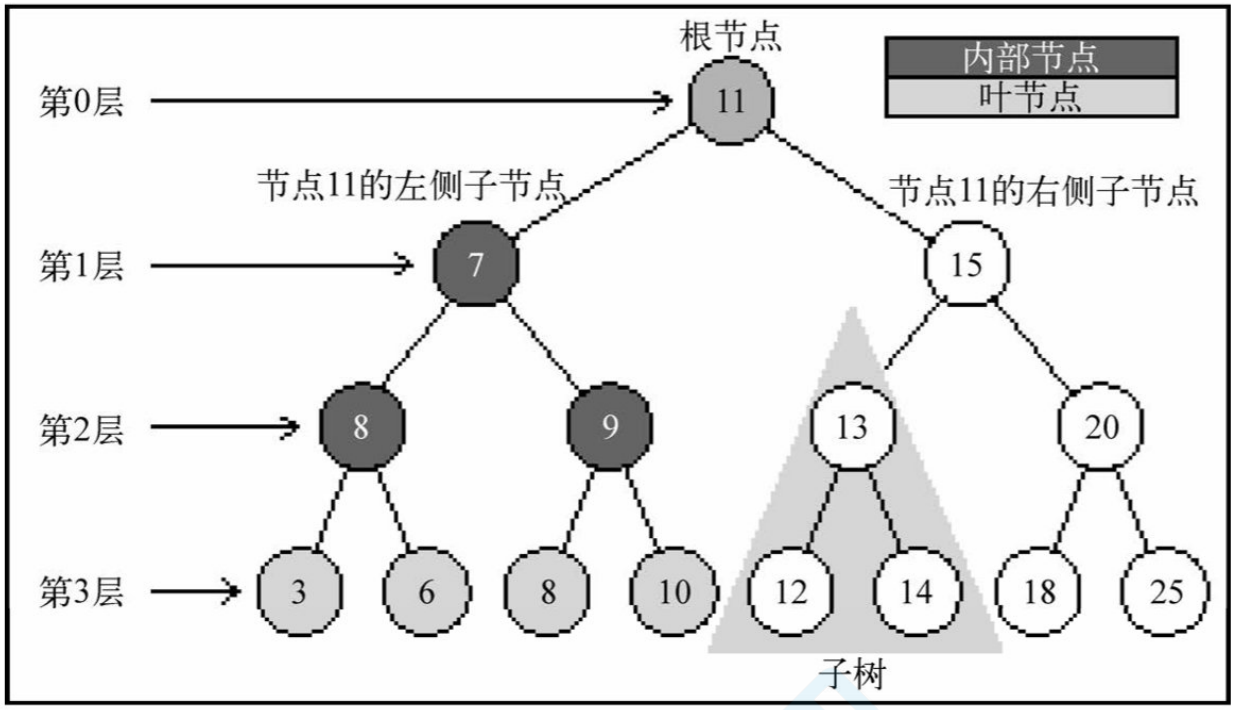
栈是一种遵从先进后出 (LIFO) 原则的有序集合

方法调用，作用域

队列

栈是一种遵从先进先出 (LIFO) 原则的有序集合

树



树的遍历 比如虚拟dom树

图

闭合的树

排序

冒泡

依次遍历，交换位置

```
function bubble_sort(arr){
    for(let i=0;i<arr.length-1;i++){
      for(let j=0;j<arr.length-i-1;j++){
        if(arr[j]>arr[j+1]){
          let swap=arr[j];
          arr[j]=arr[j+1];
          arr[j+1]=swap;
        }
      }
    }
  }

  let arr=[3,1,5,7,2,4,9,6,10,8];
  bubble_sort(arr);
  console.log(arr);
```

快速排序

二分，递归

```
function quick_sort(arr) {
    if (arr.length <= 1) {
        return arr;
    }


    let pivot = arr[0]

    let left = [];
    let right = [];
    for (let i = 1; i < arr.length; i++) {
        if (arr[i] < pivot) {
            left.push(arr[i]);
        } else {
            right.push(arr[i]);
        }
    }

    return quick_sort(left).concat([pivot], quick_sort(right));
}

var arr = [5,4,6,7,1,2,8,9,3];
console.log(quick_sort(arr));
```

原地快排序

不占用额外存储空间 原地交换位置

```
function quick_sort1(arr) {
    if (arr.length <= 1) {
        return arr;
    }
    let pivot = arr[0]

    let i = 1
    let j = arr.length-1
    while(i<j){
        let pivot = arr[0]
        while(arr[j]>=pivot && i<j){
            j --
        }
        while(arr[i]<=pivot  && i<j){
            i ++
        }
        let temp = arr[i]
        arr[i] = arr[j]
        arr[j] = temp
```

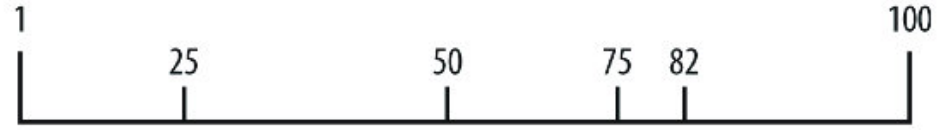
```
    }
    let left = arr.slice(1,i+1)
    let right = arr.slice(j+1)
    return [...quick_sort1(left), pivot, ...quick_sort1(right)]

}

console.log(quick_sort1(arr));
```

。。。选择，希尔，归并，堆，桶，基数等等

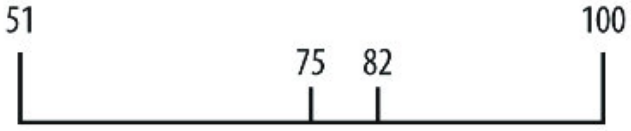
排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

.二分搜索

猜数字游戏，目标数字82

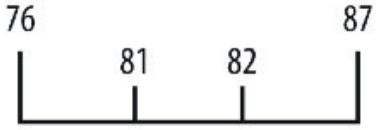


第一次猜测：50，回应：太小

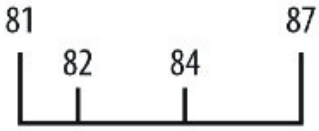


第二次猜测：76，回应：太小

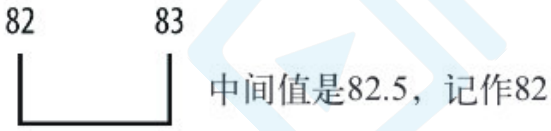
第三次猜测：88，回应：太大



第四次猜测：81，回应：太小



第五次猜测：84，回应：太大



第六次猜测：82，回应：正确

1.将数组的第一个位置设置为下边界（0） 2.将数组最后一个元素所在的位置设置为上边界（数组的长度减1）。 3.若下边界等于或小于上边界，则做如下操作。

- 将中点设置为（上边界加上下边界）除以2
- 如果中点的元素小于查询的值，则将下边界设置为中点元素所在下标加1
- 如果中点的元素大于查询的值，则将上边界设置为中点元素所在下标减1
- 否则中点元素即为要查找的数据，可以进行返回。