数据结构与算法

- 数据结构
  - 线性
    - 线性表
      - 数组
        - 特点：相通元素存储在连续的空间内、因而随机访问性能较好，但是插入和删除性能较差。
        - 遍历：
          - 方式:循环
          - 遍历时间：O(n)
        - 时间复杂度
          - 随机访问
            - 最好时间复杂度：O(1)
            - 平均时间复杂度：O(1)
            - 最坏时间复杂度:O(1)
          - 删除
            - 最好时间复杂度：O(n)
            - 平均时间复杂度：O(n)
            - 最坏时间复杂度:O(n)
          - 插入
            - 最好时间复杂度：O(n)
            - 平均时间复杂度：O(n)
            - 最坏时间复杂度:O(n)
      - 链表
        - 特点：存在不连续的空间内，随机访问性能较差，但是插入和删除性能较高。
        - 遍历
          - 方式：循环
          - 遍历时间：O(n)
        - 时间复杂度（单链表为例）
          - 随机访问
            - 最好时间复杂度：O(1)
            - 平均时间复杂度：O(n)
            - 最坏时间复杂度:O(n)
          - 删除
            - 最好时间复杂度：O(1)
            - 平均时间复杂度：O(1)
            - 最坏时间复杂度:O(1)
          - 插入
            - 最好时间复杂度：O(1)
            - 平均时间复杂度：O(1)
            - 最坏时间复杂度:O(1)
        - 种类
          - 单链表
          - 双向链表
          - 循环链表
          - 双向循环链表
    - 队列　特点：先进先出的线性表
    - 栈　特点：先进后出的线性表
    - HashMap　特点：通过特殊的哈希算法，将key换成成Value的对应位置。
  - 树　二叉树
    - 遍历顺序：
      - 前序遍历：根-左-右
      - 中序遍历：左-根-右
      - 后序遍历：左-右-根
    - 遍历
      - 方式：递归
      - 时间复杂度：O(logn)
    - 特殊分类　二叉搜索树
      - 特点：指一一棵空树或者具有下列列性质的二二叉树
      - 1. 左子子树上所有结点的值均小于它的根结点的值；
      - 2. 右子子树上所有结点的值均大大于它的根结点的值
  - 图
- 算法
  - 分支判断
  - 循环　特殊使用场景　搜索
    - 深度优先搜索

```python
def DFS(self, tree):

    if tree.root is None:
        return []

    visited, stack = [], [tree.root]

    while stack:
        node = stack.pop()
        visited.add(node)

        process(node)
        nodes = generate_related_nodes(node)
        stack.push(nodes)

    # other processing work
    ...
```

    - 广度优先搜索

```python
def BFS(graph, start, end):

    queue = []
    queue.append([start])
    visited.add(start)

    while queue:
        node = queue.pop()
        visited.add(node)

        process(node)
        nodes = generate_related_nodes(node)
        queue.push(nodes)

    # other processing work
    ...
```

  - 递归

```
public void recur(int level, int param) {
  // terminator
  if (level > MAX_LEVEL) {
    // print result
    return;
  }

  // process current logic
  process(level, param);

  // drill down
  recur( level + 1, newParam);

  // restore current status
  param = ....
}
```

    - 特殊使用场景
      - 分治

```
def divide_conquer(problem, param1, param2, ...):

  # recursion terminator
  if problem is None:
    print_result
    return

  # prepare data
  data = prepare_data(problem)
  subproblems = split_problem(problem, data)

  # conquer subproblems
  subresult1 = self.divide_conquer(subproblems[0], p1, ...)
  subresult2 = self.divide_conquer(subproblems[1], p1, ...)
  subresult3 = self.divide_conquer(subproblems[2], p1, ...)
  ...

  # process and generate the final result
  result = process_result(subresult1, subresult2, subresult3, ...)
```

      - 搜索
        - 深度优先搜索

```
visited = set()
def dfs(node, visited):
    visited.add(node)
    # process current node here.
    ...
    for next_node in node.children():
        if not next_node in visited:
            dfs(next_node, visited)
```

        - 广度优先搜索
  - 动态规划
  - 二分查找
  - 贪心算法

XMind:ZEN | 试用模式