嵌入式与软件开发基础学习报告

1. 嵌入式系统概论

1.1 概念

嵌入式系统是一种**专用计算机系统**，它作为设备或装置的一部分，通常用于控制、监视或协助操作设备和机器。它与通用计算机（如 PC、服务器）的核心区别在于其**专用性**。

核心特征：

- **专用性：** 为特定任务而设计，功能固定
- **资源受限：** 处理器性能、内存、存储空间通常远低于通用计算机
- **实时性：** 许多嵌入式系统要求在规定时间内响应外部事件
- **低功耗：** 尤其是电池供电的设备，对功耗极为敏感
- **成本敏感：** 硬件和软件设计都需考虑成本控制
- **直接硬件交互：** 软件需要直接操作寄存器、外设，与硬件紧密耦合

2. Markdown、Git、GitHub

2.1 Markdown 语法

元素	Markdown 语法	效果	快捷键
标题	`# H1` `## H2` `### H3`	不同级别的标题	`Ctrl` + 数字
加粗	`**加粗文本**`	**加粗文本**	`Ctrl` + `B`
斜体	`*斜体文本*`	*斜体文本*	`Ctrl` + `I`
代码	\`单行代码\`	`int a = 10;	
代码块	\`\`\`c  代码...  \`\`\`	语法高亮的代码块	`Ctrl` + `Shift` + `K`
链接	`[链接文本](URL)`	百度	
图片	`![替代文本](图片URL)`	显示图片	
列表	`- 项1` 或 `1. 项1`	无序或有序列表	
引用	`> 引用文字`	> 引用文字	
表格	使用 `-` 和 `|`	如上所示	
分割线	`***` 或 `---`	一条水平线	

2.2 Git 与 GitHub

- **Git：** 一个开源的**分布式版本控制系统。**它记录文件的所有历史变更，允许回溯、分支和协作
- **GitHub：** 一个基于 Git 的**代码托管平台**，提供了协作、代码审查、问题跟踪等社交化功能

核心概念与工作流：

1. **仓库：** 一个被 Git 管理的项目文件夹
2. **提交：** 一次代码的版本存档，包含变更描述和唯一 ID
3. **分支：** 从主线上分离开来的“副本”，用于开发新功能而不影响主线
4. **合并：** 将一个分支的修改整合到另一个分支

常用命令：

```
# 初始化本地仓库
git init

# 克隆远程仓库
git clone <url>

# 查看文件状态
git status

# 将文件添加到暂存区
git add <filename> 或 git add .

# 提交变更到本地仓库
git commit -m "提交信息"

# 查看提交历史
git log

# 创建并切换分支
git checkout -b <branch-name>

# 切换分支
git checkout <branch-name>

# 合并分支
git merge <branch-name>

# 将本地提交推送到远程仓库
git push origin <branch-name>

# 从远程仓库拉取更新
git pull origin <branch-name>
```

3. 数据结构

3.1 栈

- **特点：** **后进先出**
- **操作：**
  - `Push`： 将元素压入栈顶
  - `Pop`： 从栈顶弹出元素
- **实现方式：** 数组或链表
- **嵌入式应用场景：**
  - 函数调用时存储返回地址、局部变量（由编译器自动管理）
  - 表达式求值
  - 中断处理时的上下文保存

3.2 队列

- **特点：** **先进先出**
- **操作：**
  - `Enqueue`： 将元素加入队尾
  - `Dequeue`： 从队头取出元素
- **实现方式：** 数组（循环队列）或链表
- **嵌入式应用场景：**
  - 串口数据接收缓冲區（生产者-消费者模型）
  - RTOS 中的任务间消息传递
  - 按键事件的缓冲

3.3 链表

- **特点：** 由一系列节点组成，每个节点包含数据和指向下一个节点的指针。可以动态地增加或删除节点
- **类型：** 单向链表、双向链表、循环链表
- **操作：** 插入、删除、遍历
- **嵌入式应用场景：**
  - 动态管理不定数量的对象（如 TCP 连接池）
  - 实现更复杂的数据结构（如队列、栈）
  - 内存管理（如 FreeRTOS 的堆内存管理）

实现示例（C 语言）：

```c
#include <stdio.h>
#include <stdlib.h>

// 定义链表节点
struct Node {
    int data;           // 数据域
    struct Node* next;  // 指针域，指向下一个节点
};

// 创建一个新节点
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
```

```
        printf("Memory allocation failed!\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// 在链表头部插入节点
void insertAtHead(struct Node** headRef, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = *headRef;
    *headRef = newNode;
}

// 遍历并打印链表
void printList(struct Node* node) {
    while (node != NULL) {
        printf("%d -> ", node->data);
        node = node->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL; // 链表头指针

    insertAtHead(&head, 3);
    insertAtHead(&head, 2);
    insertAtHead(&head, 1);

    printList(head);

    return 0;
}
```

3.4 图

图由**顶点**和**边**组成，用于表示多对多的关系。

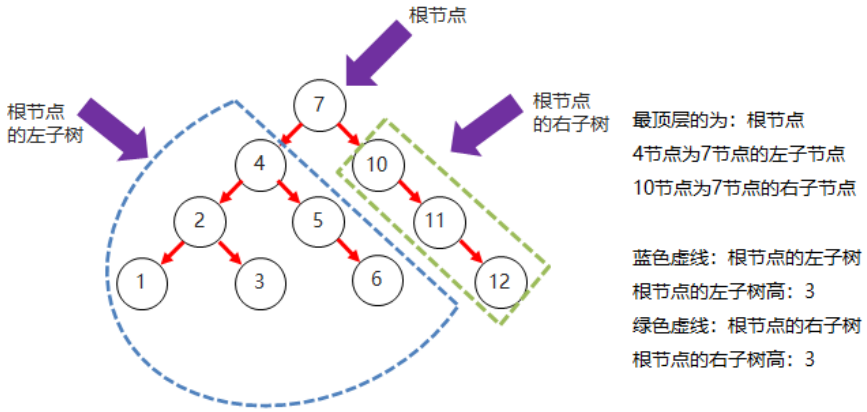
重要术语：

- **顶点：** 图中的节点
- **边：** 顶点之间的连接
- **有向图/无向图：** 边是否有方向
- **权重：** 边上的数值
- **路径：** 顶点序列
- **环：** 起点和终点相同的路径

3.5 翻看之前相关笔记

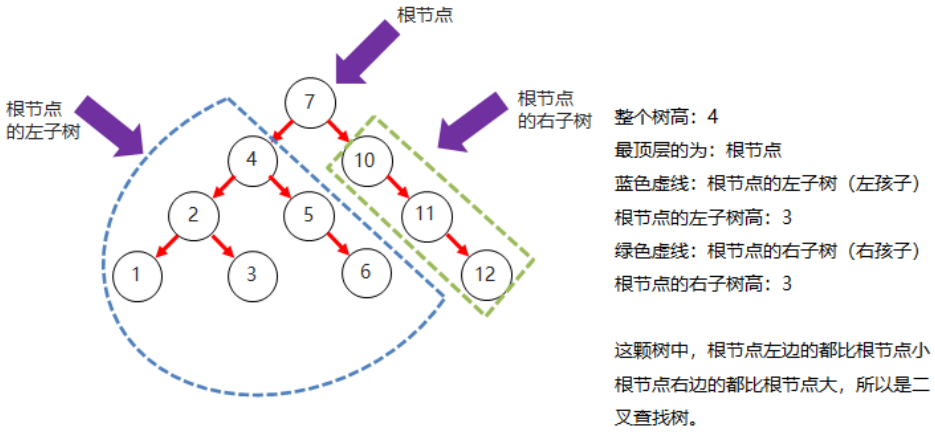
3.5.1 二叉树

- 二叉树的特点
  - 二叉树中，任意一个节点的度要小于等于 2
    - 节点：在树结构中，每一个元素称之为节点
    - 度：每一个节点的子节点数量称之为度
- 二叉树结构图

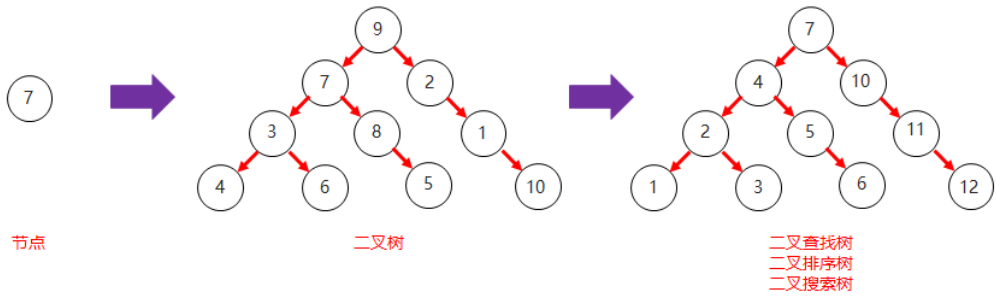


3.5.2 二叉查找树

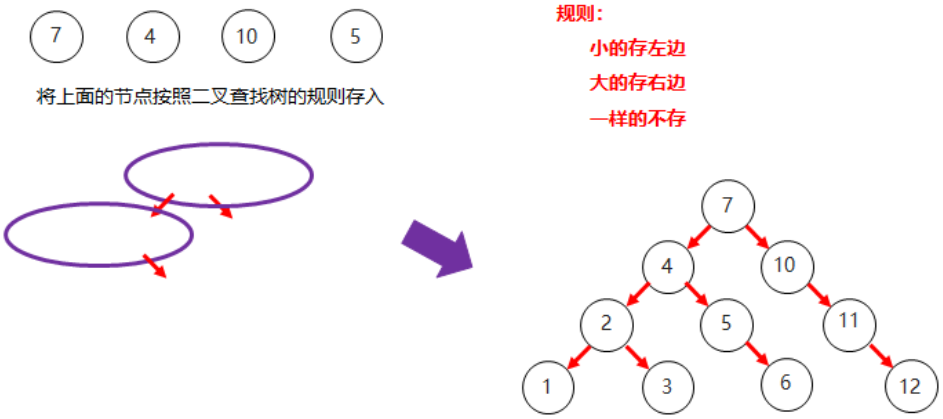
- 二叉查找树的特点
  - 二叉查找树，又称二叉排序树或者二叉搜索树
  - 每一个节点上最多有两个子节点
  - 左子树上所有节点的值都小于根节点的值
  - 右子树上所有节点的值都大于根节点的值
- 二叉查找树结构图



- 二叉查找树和二叉树对比结构图



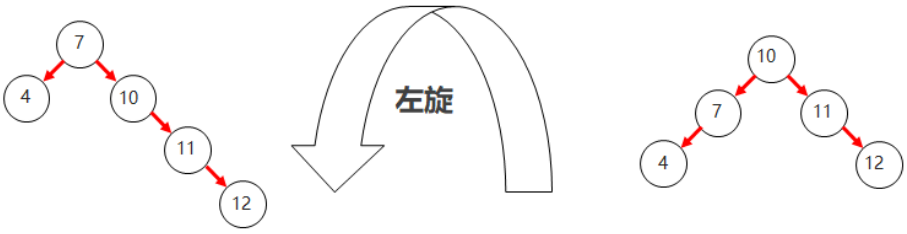
- 二叉查找树添加节点规则
  - 小的存左边
  - 大的存右边
  - 一样的不存

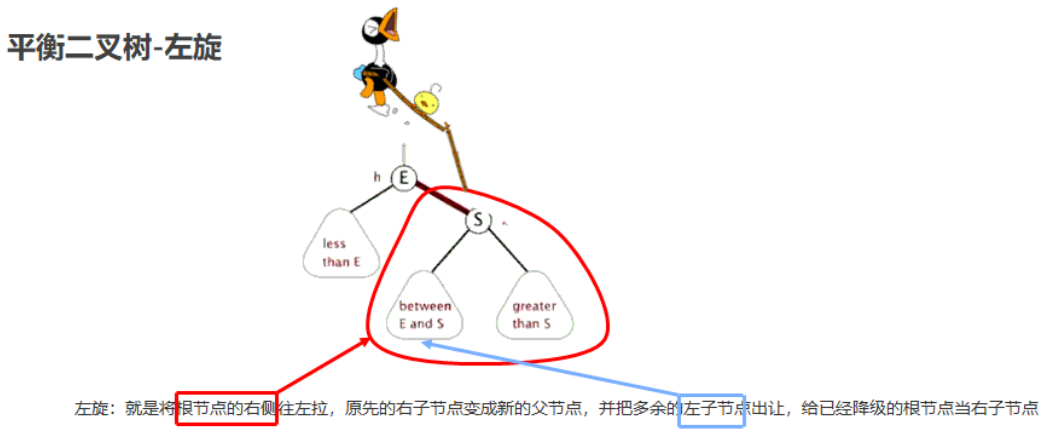


3.5.3 平衡二叉树【理解】

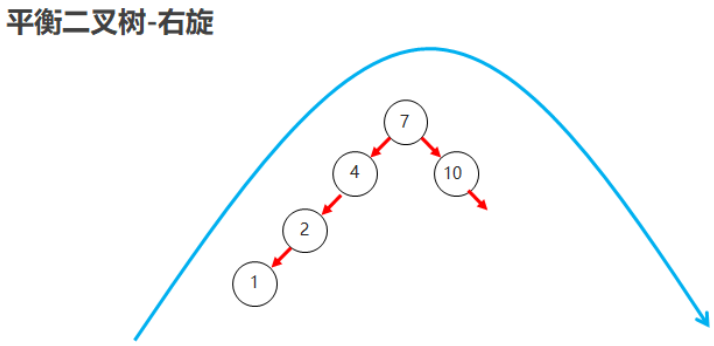
- 平衡二叉树的特点
  - 二叉树左右两个子树的高度差不超过 1
  - 任意节点的左右两个子树都是一颗平衡二叉树
- 平衡二叉树旋转
  - 旋转触发时机
    - 当添加一个节点之后，该树不再是一颗平衡二叉树
  - 左旋
    - 就是将根节点的右侧往左拉，原先的右子节点变成新的父节点，并把多余的左子节点出让，给已经降级的根节点当右子节点

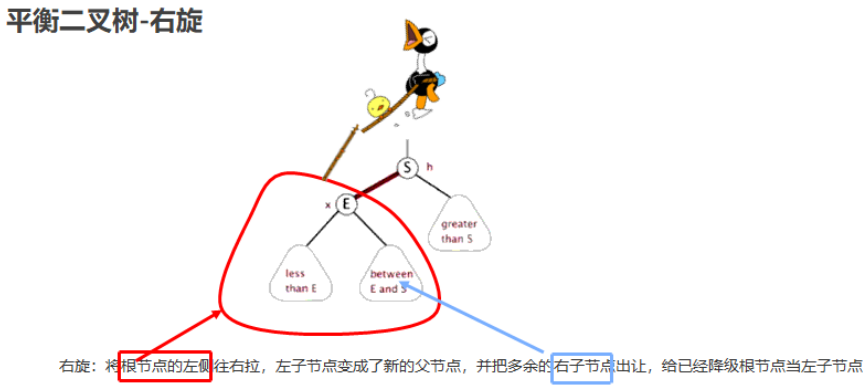
平衡二叉树-左旋



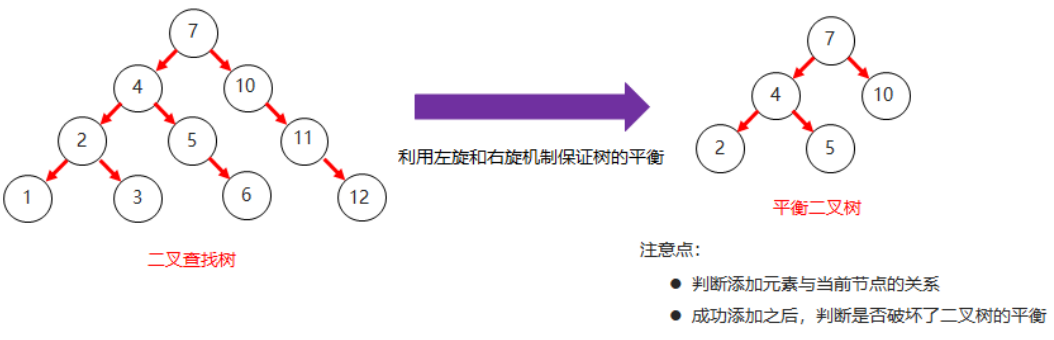


- 右旋
  - 就是将根节点的左侧往右拉，左子节点变成了新的父节点，并把多余的右子节点出让给已经降级根节点当左子节点



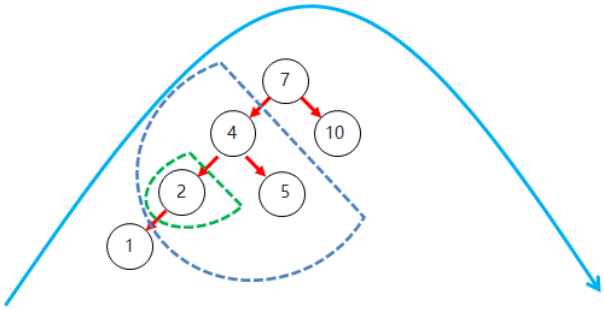


- 平衡二叉树和二叉查找树对比结构图

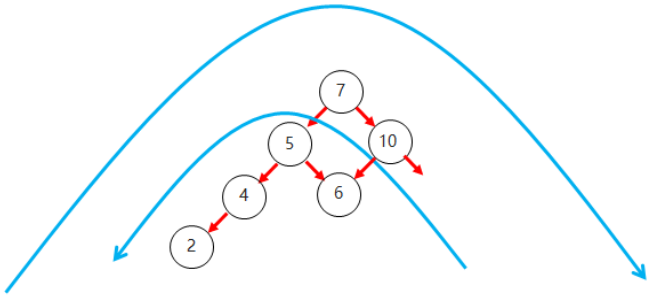


- 平衡二叉树旋转的四种情况

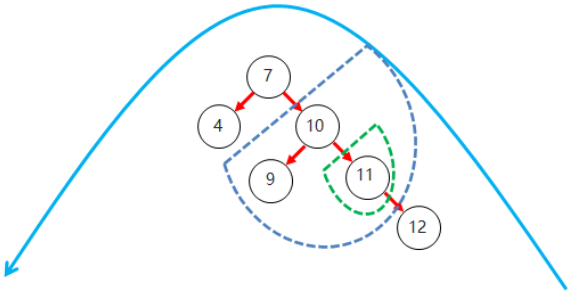
- 左左
  - 左左：当根节点左子树的左子树有节点插入，导致二叉树不平衡
  - 如何旋转：直接对整体进行右旋即可



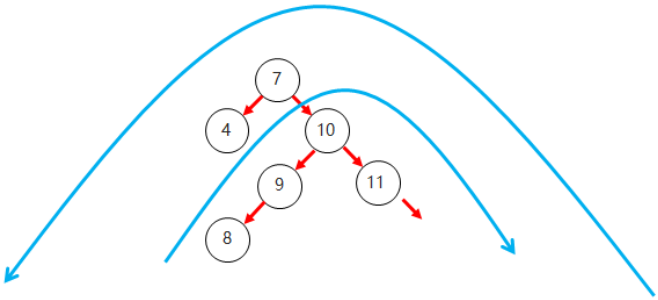
- 左右
  - 左右：当根节点左子树的右子树有节点插入，导致二叉树不平衡
  - 如何旋转：先在左子树对应的节点位置进行左旋，在对整体进行右旋



- 右右
  - 右右：当根节点右子树的右子树有节点插入，导致二叉树不平衡
  - 如何旋转：直接对整体进行左旋即可

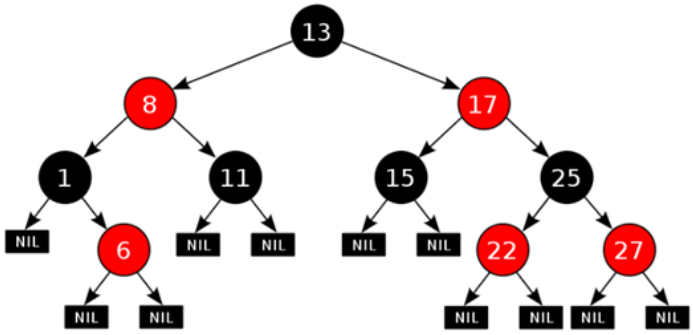


- 右左
  - 右左：当根节点右子树的左子树有节点插入，导致二叉树不平衡
  - 如何旋转：先在右子树对应的节点位置进行右旋，在对整体进行左旋

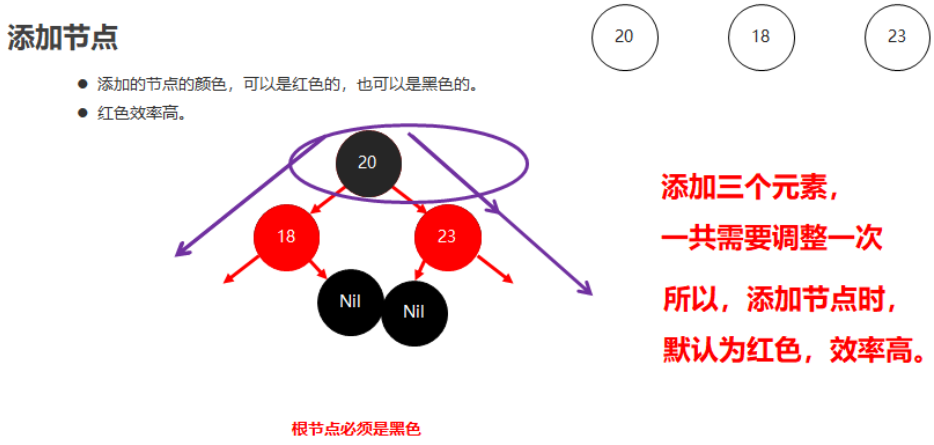


3.5.3 红黑树【理解】

- 红黑树的特点
  - 平衡二叉 B 树
  - 每一个节点可以是红或者黑
  - 红黑树不是高度平衡的，它的平衡是通过自己的红黑规则进行实现的
- 红黑树的红黑规则有哪些
  1. 每一个节点或是红色的，或者是黑色的
  2. 根节点必须是黑色
  3. 如果一个节点没有子节点或者父节点，则该节点相应的指针属性值为 Nil，这些 Nil 视为叶节点，每个叶节点是黑色的
  4. 如果某一个节点是红色，那么它的子节点必须是黑色（不能出现两个红色节点相连的情况）
  5. 对每一个节点，从该节点到其所有后代叶节点的简单路径上，均包含相同数目的黑色节点



- 红黑树添加节点的默认颜色
  - 添加节点时，默认为红色，效率高



- 红黑树添加节点后如何保持红黑规则
  - 根节点位置
    - 直接变为黑色
  - 非根节点位置
    - 父节点为黑色
      - 不需要任何操作，默认红色即可
    - 父节点为红色
      - 叔叔节点为红色
        1. 将“父节点”设为黑色，将“叔叔节点”设为黑色
        2. 将“祖父节点”设为红色
        3. 如果“祖父节点”为根节点，则将根节点再次变成黑色

- 叔叔节点为黑色
    1. 将“父节点”设为黑色
    2. 将“祖父节点”设为红色
    3. 以“祖父节点”为支点进行旋转

4. 总结

本次学习涵盖了从底层嵌入式概念到上层开发工具的完整链条：

1. **嵌入式**是软硬件的交汇点，要求开发者既能驾驭 C 语言和数据结构，又能理解硬件原理
2. **Markdown、Git、GitHub** 是现代化、高效率开发的“标配”工具链，保证了代码和文档管理的规范性与协作性
3. **数据结构（栈、队列、链表）** 是程序的骨架，在资源受限的嵌入式系统中，其选择和实现直接决定了程序的性能和稳定性