树的双亲表示法与其结构解析

1. 双亲表示法简介

双亲表示法是一种常用的树存储结构，它通过一个数组存储树的所有节点，并记录每个节点的父节点位置。每个节点包含两个核心信息：

- **数据域**：存储节点本身的数据值。
- **双亲域**：记录节点父节点在数组中的位置索引。

双亲表示法通过紧凑的结构，快速地实现父节点的查询，同时限制了空间使用，是适用于固定或少量动态操作场景的存储方式。

---

2. 代码结构与宏定义解析

（1）宏定义：`#define MAX_TREE_SIZE 100`

```
#define MAX_TREE_SIZE 100
```

`MAX_TREE_SIZE` 定义了树的最大节点数，用于限制节点数组的大小。在这里，树最多可容纳 100 个节点。如果需要更大的树，可以调整该值。

（2）定义节点数据类型：`TElemType`

```
typedef int TElemType;
```

`TElemType` 是节点数据类型的别名。在当前代码中，节点数据类型为 `int`，表示每个节点存储整数值。如果需要存储其他类型的数据（如字符或浮点数），只需修改 `TElemType` 的定义即可。

---

3. 结构定义与详细解析

（1）节点结构定义：`PTNode`

```
typedef struct PTNode {
    TElemType data;   /* 节点数据 */
    int parent;       /* 双亲位置 */
} PTNode;
```

- `data`：表示节点的数据域，数据类型为 `TElemType`（这里为 `int`）。

- `parent`：记录节点的父节点在数组中的索引。如果 `parent = -1`，表示该节点是树的根节点（没有父节点）。

节点结构通过紧凑的方式表示节点的基本信息，双亲域直接关联到数组中的其他节点。

（2）树结构定义：`PTree`

```
typedef struct {
    PTNode nodes[MAX_TREE_SIZE]; /* 节点数组 */
    int r;                       /* 根节点位置 */
    int n;                       /* 节点总数 */
} PTree;
```

- `nodes[MAX_TREE_SIZE]`：用一个固定大小的数组存储树的所有节点，每个节点为 `PTNode` 类型。
- `r`：根节点在数组中的索引。例如，如果根节点在 `nodes[0]`，则 `r = 0`。
- `n`：当前树的节点总数，动态维护树的大小。

通过 `PTree` 结构，整棵树的信息得以统一管理。

---

4. 双亲表示法的优缺点

优点

1. **节省空间**：使用数组存储节点，无需额外指针。
2. **快速查找父节点**：通过节点的 `parent` 字段直接定位父节点索引。

缺点

1. **子节点查询效率低**：需遍历整个数组查找 `parent` 值等于目标节点索引的节点。
2. **动态性较差**：节点的插入、删除操作可能导致数组重新排列或空间浪费。

---

5. `r` 和 `n` 的赋值与作用

（1）`r` 的作用与赋值

`r` 代表树的根节点在 `nodes` 数组中的索引。

- **初始化时**：
  - 根节点的 `parent` 值为 `-1`，表示没有父节点。
  - `r` 被赋值为根节点在 `nodes` 数组中的位置索引。

示例代码：

```
PTree tree;
tree.nodes[0].data = 'A';  // 根节点数据
tree.nodes[0].parent = -1; // 根节点没有父节点
tree.r = 0;                // 根节点索引为 0
tree.n = 1;                // 当前树节点数为 1
```

(2) `n` 的作用与赋值

`n` 表示树中当前节点的总数。

- **初始化时**:
  - 树为空时，`n = 0`。
  - 添加节点后，`n` 的值随之增加。
- **动态更新**:
  - **插入节点**: `n += 1`
  - **删除节点**: `n -= 1`

示例代码:

```
tree.n = 0;                // 初始化树为空
tree.nodes[0].data = 'A';  // 添加根节点
tree.nodes[0].parent = -1;
tree.r = 0;
tree.n += 1;               // 节点数增加到 1
```

---

6. 使用示例：树的构建与节点存储

树结构示例

假设我们有以下树:

```
    A
   /|\
  B C D
```

双亲表示法存储

- 根节点 (A): 无父节点，`parent = -1`。
- 节点 (B, C, D): 父节点为 (A)，其 `parent = 0`。

索引	数据 (`data`)	父节点 (`parent`)
0	A	-1

索引	数据 (data)	父节点 (parent)
1	B	0
2	C	0
3	D	0

构建代码

```
PTree tree;
tree.n = 0;                 // 初始化节点总数为 0

// 添加根节点 A
tree.nodes[0].data = 'A';
tree.nodes[0].parent = -1;
tree.r = 0;
tree.n += 1;

// 添加子节点 B
tree.nodes[1].data = 'B';
tree.nodes[1].parent = 0;
tree.n += 1;

// 添加子节点 C
tree.nodes[2].data = 'C';
tree.nodes[2].parent = 0;
tree.n += 1;

// 添加子节点 D
tree.nodes[3].data = 'D';
tree.nodes[3].parent = 0;
tree.n += 1;
```

---

7. 总结与应用场景

双亲表示法是一种紧凑的树存储方式，适用于以下场景：

1. **节点数固定**：树的节点数不会频繁增加或减少。
2. **父节点查询频繁**：需要快速定位父节点，而对子节点查询需求较少。

虽然双亲表示法在动态操作中存在局限性，但其结构简单、实现容易，是树的初学者和某些特定场景的理想选择。在实际应用中，其他存储方式（如孩子链表或孩子兄弟表示法）可弥补其不足。