
# 数据结构与算法

## 第 5 讲：树与二叉树

韩文弢

清华大学

2024—2025 学年度春季学期

# 本讲内容

# 5.1 问题引入

# 5.1.1 层次结构

日常生活和科学技术中有很多具有层次结构的事物。

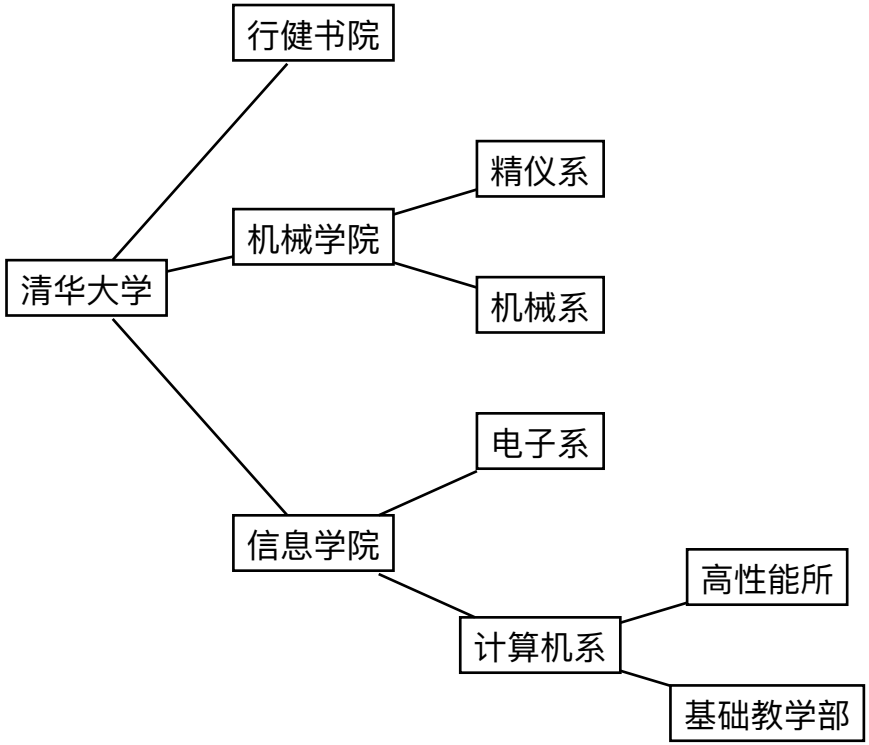



图 5.1　学校组织结构（部分）

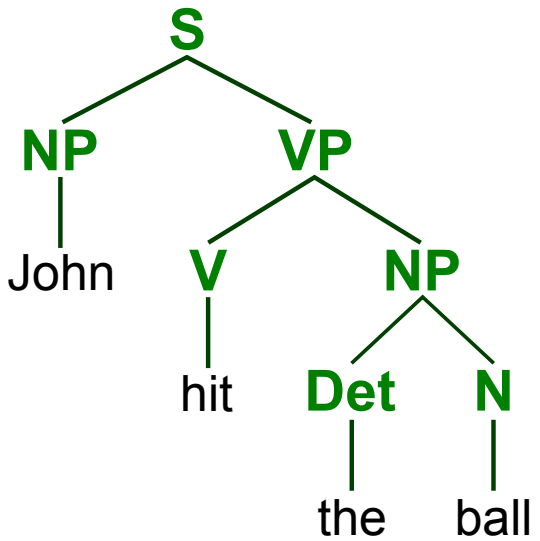
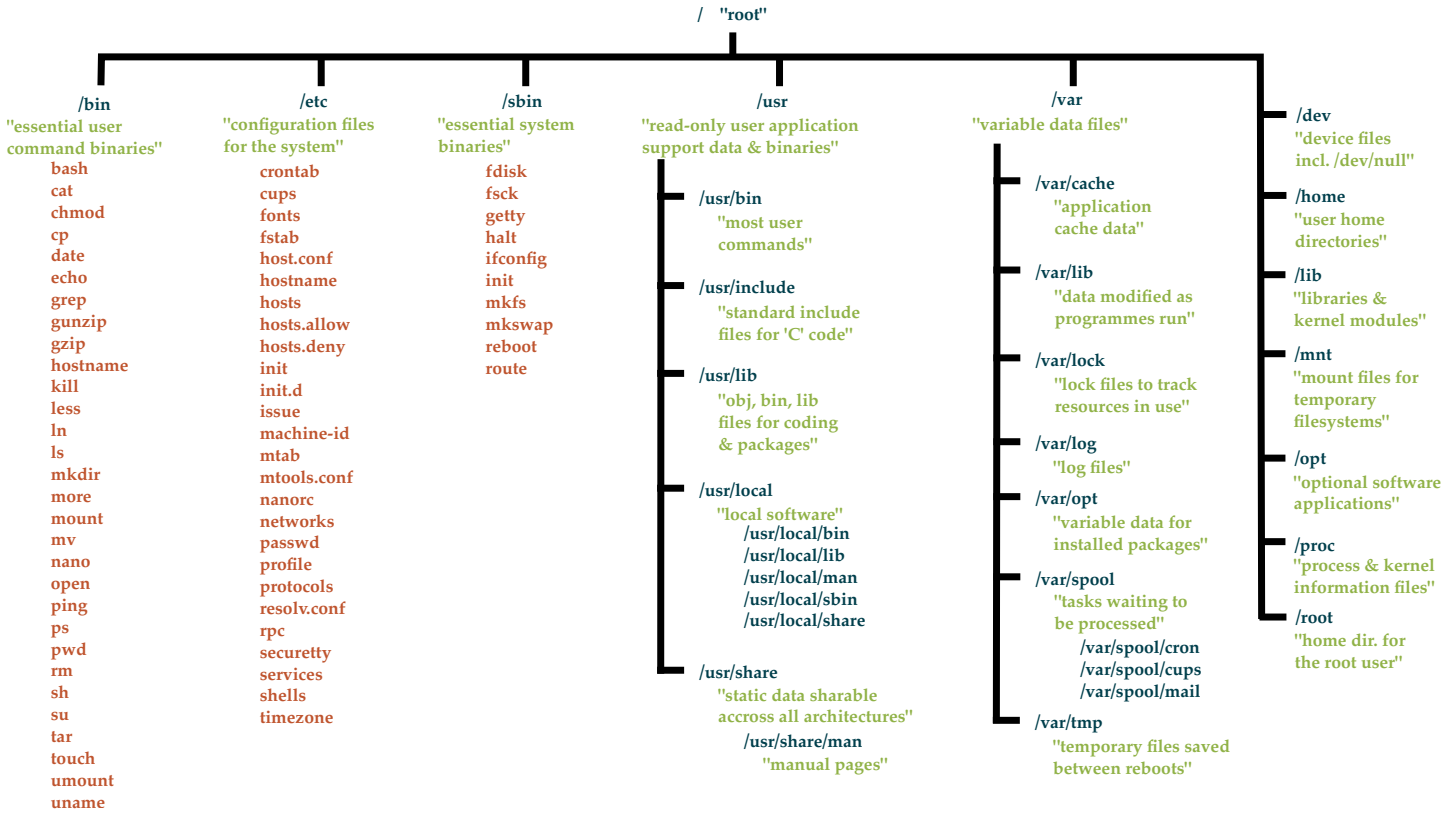



图 5.2　语法结构

# 5.1.2 计算机中的层次结构：文件系统




如何在计算机中表示和存储层次结构？能否使用线性结构？如何查询某个元素？

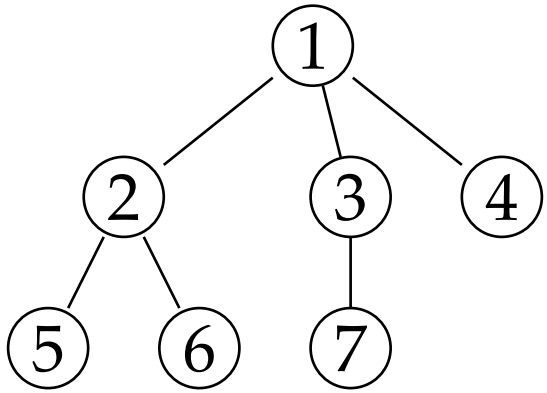
# 5.2 树的概念

# 5.2.1 树的定义

**定义 5.1 (树)** 树是由结点组成的有限集合 $T$，令 $|T|$ 表示结点的数量，则有：

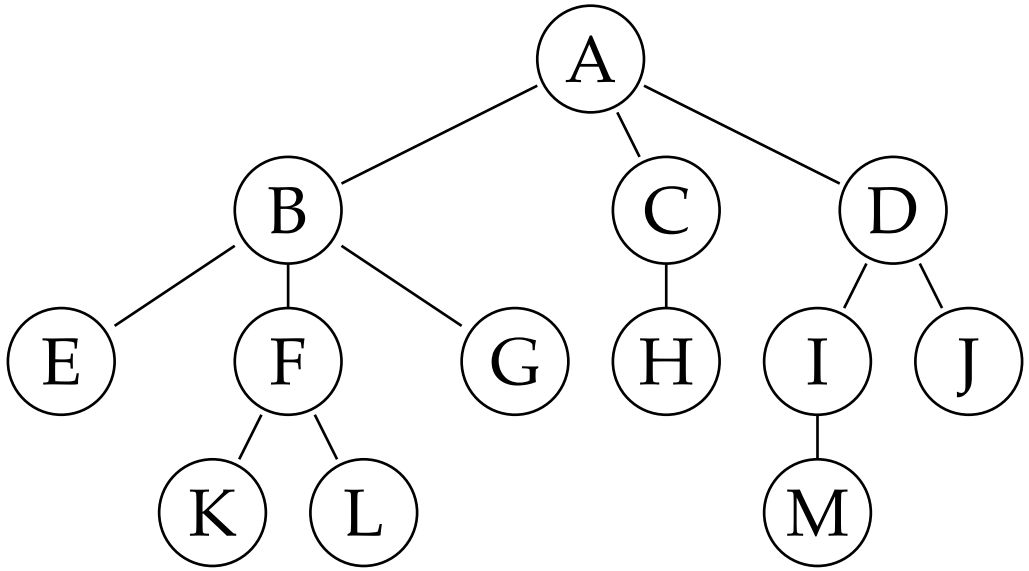
1. 若 $|T| = 0$，则称 $T$ 为**空树**；
2. 若 $|T| > 0$，则
   - $T$ 中有且仅有一个特殊结点 $r$，称为**根结点**；
   - 其余结点的集合 $T \setminus \{r\}$ 可划分为 $m(m \geq 0)$ 个互不相交的子集 $T_1, T_2, \cdots, T_m$，每个子集 $T_i(1 \leq i \leq m)$ 也是树，称为 $r$ 的**子树**；
   - 若 $|T_i| > 0$，则子树 $T_i$ 的根结点 $r_i$ 是 $r$ 的**子结点**，$r$ 是 $r_i$ 的**父结点**。

# 5.2.2 树的性质




- 树是**递归**定义的。
- 树上两个结点间的父子关系属于二元关系。
- 由 $n$ 个结点构成的树包含 $n-1$ 对父子关系。
- 用**边**连接有父子关系的结点，共有 $n-1$ 条边。

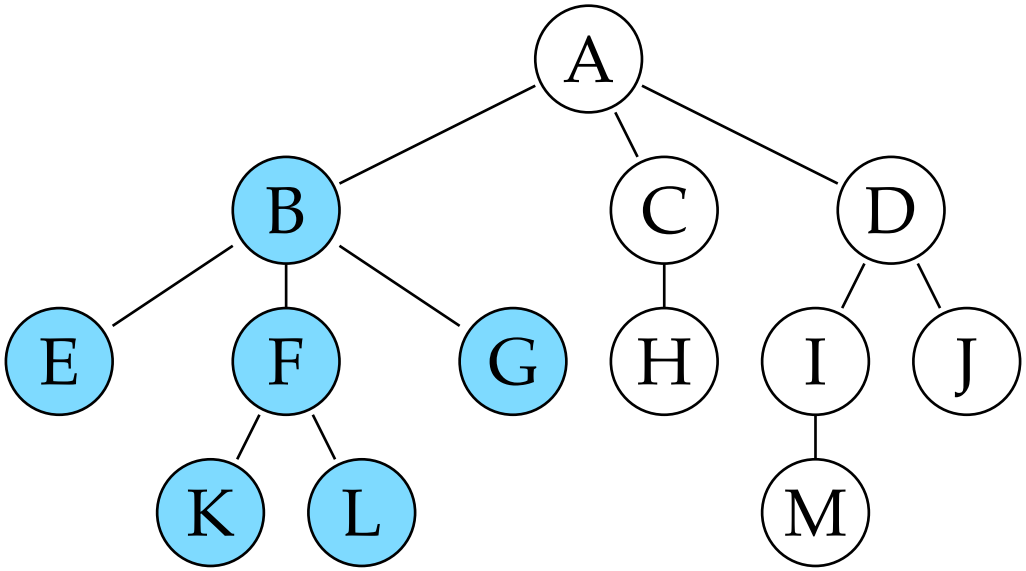
# 5.2.3 树的示例




树 $T = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}, \mathrm{G}, \mathrm{H}, \mathrm{I}, \mathrm{J}, \mathrm{K}, \mathrm{L}, \mathrm{M}\}$，根为 A，子树有：

- $T_1 = \{\mathrm{B}, \mathrm{E}, \mathrm{F}, \mathrm{G}, \mathrm{K}, \mathrm{L}\}$，根为 B
- $T_2 = \{\mathrm{C}, \mathrm{H}\}$，根为 C
- $T_3 = \{\mathrm{D}, \mathrm{I}, \mathrm{J}, \mathrm{M}\}$，根为 D

# 5.2.4 树的示例




子树 $T_1 = \{\mathrm{B}, \mathrm{E}, \mathrm{F}, \mathrm{G}, \mathrm{K}, \mathrm{L}\}$ 同样也是一棵树，根为 B，子树有：

- $T_{1,1} = \{\mathrm{E}\}$
- $T_{1,2} = \{\mathrm{F}, \mathrm{K}, \mathrm{L}\}$
- $T_{1,3} = \{\mathrm{G}\}$

# 5.2.5 树的基本术语

**结点的度** 子结点或非空子树的个数

**树的度** 树中所有结点的度的最大值

**叶结点** 树中度为 $0$ 的结点

**中间结点** 树中叶结点以外的结点，亦称内部结点

**兄弟结点** 父结点相同的结点彼此是兄弟结点

**结点的层次** 根结点在第 $1$ 层；如果结点的层次是 $k(k \geq 1)$，则其子结点都在第 $k+1$ 层；亦称结点的深度

**结点的高度** 叶结点的高度等于 $1$；中间结点的高度等于其所有子结点的高度的最大值加 $1$

**树的高度** 根结点的高度，亦称树的深度

# 5.2.6 树的基本术语

**有序树** 树中各结点的子树从左向右依次排列，不能交换次序；否则称作**无序树**

**祖先结点** 根结点没有祖先；父结点以及父结点的祖先都是该结点的祖先结点

**子孙结点** 叶结点没有子孙；中间结点的各子结点以及这些子结点的子孙都是该结点的子孙结点

# 5.2.7 树的抽象数据类型

**伪码 5.1:** 树的 ADT

**元素**：$D = \{a_i \mid a_i \in U, i = 0, 1, 2, ..., n-1\}$，$U$ 是元素全集，$n$ 是元素个数，$a_0$ 若存在是根结点

**关系**：$R = \left\{\left(a_{p_i}, a_i\right) \mid i = 1, 2, ..., n-1, \text{结点 } p_i \text{ 是结点 } i \text{ 的父结点}\right\}$

**基本操作**：

> Initialize($T$): 初始化一棵空的树 $T$
>
> Destroy($T$): 销毁树 $T$
>
> IsEmpty($L$): 返回树 $T$ 是否为空
>
> Root($T$): 返回树 $T$ 的根结点，若树为空则返回 $\emptyset$
>
> Parent($T$, $p$): 返回树 $T$ 上结点 $p$ 的父结点，若 $p$ 为根则返回 $\emptyset$
>
> Degree($T$, $p$): 获得树 $T$ 上结点 $p$ 的度数
>
> Get($T$, $p$, $i$): 获得树 $T$ 上结点 $p$ 的第 $i$ 个子结点
>
> Insert($T$, $p$, $i$, $x$): 在树 $T$ 上结点 $p$ 的第 $i$ 个子结点之前插入元素 $x$ 作为新的结点
>
> Remove($T$, $p$): 删除树 $T$ 上以结点 $p$ 为根的子树
>
> Traverse($T$, Visit): 遍历树中的每个结点，对每个结点调用 Visit 操作

# 5.2.8 二叉树的定义

二叉树是最简单、最常用的树。

**定义 5.2 (二叉树)** 二叉树是由结点组成的有限集合 $T$，则有：

1. 若 $|T| = 0$，则称 $T$ 为**空二叉树**；
2. 若 $|T| > 0$，则
   - $T$ 中有且仅有一个特殊结点 $r$，称为**根结点**；
   - 其余结点的集合 $T \setminus \{r\}$ 可划分为两个互不相交的子集 $T_{\mathrm{L}}$ 和 $T_{\mathrm{R}}$，分别称为 $r$ 的左子树和右子树。

注意：二叉树的左右子树位置不能交换。

# 5.2.9 二叉树的五种基本形态


$\emptyset$

①空二叉树　②左右子树都为空的二叉树　③右子树为空的二叉树　④左子树为空的二叉树　⑤左右子树都非空的二叉树

注意：形态③和④是不同的。

# 5.2.10 二叉树的基本性质

**命题 5.3** 设非空二叉树上度为 $0, 1, 2$ 的结点个数分别为 $n_0, n_1, n_2$，则有 $n_0 = n_2 + 1$。

证明. 根据命题，二叉树的结点总数 $n = n_0 + n_1 + n_2$。

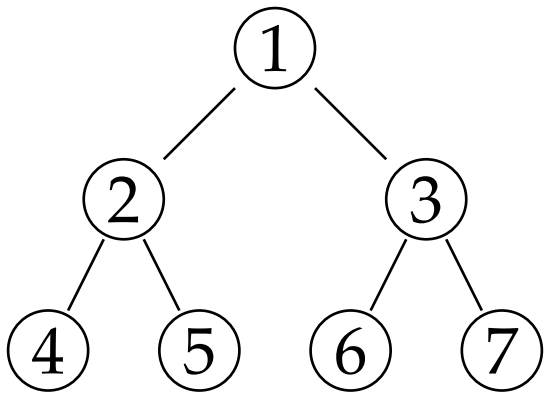
考虑每个结点向子结点引出的边，可知总边数为 $0 \times n_0 + 1 \times n_1 + 2 \times n_2 = n_1 + 2n_2$。

另一方面，由树的性质可知边数为 $n - 1$，因此 $n_1 + 2n_2 = n - 1 = n_0 + n_1 + n_2 - 1$，化简可得 $n_2 = n_0 - 1$，即 $n_0 = n_2 + 1$。 □

# 5.2.11 二叉树的基本性质

**命题 5.4** 二叉树的第 $i$ 层最多有 $2^{i-1}$ 个结点（$i \geq 1$）。

**命题 5.5** 深度为 $d$ 的二叉树最多有 $2^d - 1$ 个节点（$d \geq 1$）。



# 5.2.12 二叉树的抽象数据类型

**伪码 5.2:** 二叉树的 ADT

**元素**：$D = \{a_i \mid a_i \in U, i = 0, 1, 2, ..., n-1\}$，$U$ 是元素全集，$n$ 是元素个数，$a_0$ 若存在是根结点

**关系**：$R = \left\{\left(a_{p_i}, a_i\right) \mid i = 1, 2, ..., n-1, \text{结点 } p_i \text{ 是结点 } i \text{ 的父结点}\right\}$

**基本操作**：

Initialize($T$): 初始化一棵空的二叉树 $T$

Destroy($T$): 销毁二叉树 $T$

IsEmpty($L$): 返回二叉树 $T$ 是否为空

Root($T$): 返回二叉树 $T$ 的根结点，若树为空则返回 $\emptyset$

Parent($T$, $p$): 返回二叉树 $T$ 上结点 $p$ 的父结点，若 $p$ 为根则返回 $\emptyset$

GetLeft/Right($T$, $p$): 获得二叉树 $T$ 上结点 $p$ 的左/右子结点

InsertLeft/Right($T$, $p$, $x$): 在二叉树 $T$ 上结点 $p$ 的左/右子树位置插入元素 $x$ 作为新的结点

Remove($T$, $p$): 删除二叉树 $T$ 上以结点 $p$ 为根的子树

TraversePreorder/Inorder/Postorder($T$, Visit): 前/中/后序遍历树中的每个结点

# 5.3 树的实现

# 5.3.1 树的存储方式

由于树的结构不是线性的，因此一般情况下无法使用顺序存储方式。

通常使用链式存储方式，给每个结点添加指向子结点和/或父结点的指针域。具体如何选取要根据操作需要和存储空间进行权衡。

对于一般的树，由于度数不确定，需要使用支持变长的序列来存储子结点的指针域。或者使用孩子兄弟表示法转化为二叉树。

在结点总数已知的情况下，也可以采用静态分配数组，使用数组下标来代替指针。

# 5.3.2 二叉树的存储方式

**伪码 5.3:** 二叉树结点的链式存储

BinaryTreeNode:

> value: 结点所带的值
>
> left: 指向左子结点的指针
>
> right: 指向右子结点的指针
>
> parent: 指向父结点的指针（可选）

对于一棵二叉树，设置 root 指针指向根结点。

# 5.4 二叉树的遍历

# 5.4.1 遍历的概念

**遍历**是树和二叉树上一种重要的基本操作，要求按照某种顺序访问每一个结点，**既不重复，也不遗漏**。遍历操作能够将树这样的非线性结构转化为线性序列。

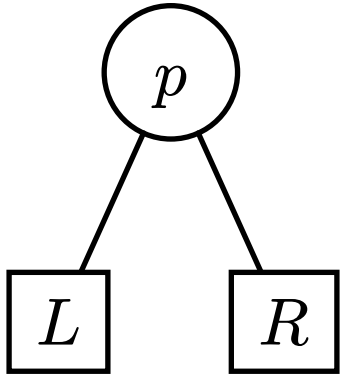
结合二叉树的递归定义，设根结点为 $p$，左右子树分别为 $L$ 和 $R$，则一共有 6 种不同的访问顺序：

$$pLR \qquad pRL \qquad LpR \qquad LRp \qquad RpL \qquad RLp$$

再规定左子树必须在右子树之前访问，于是剩下 3 种遍历方式：

$pLR$：**前序** $LpR$：**中序** $LRp$：**后序**

# 5.4.2 二叉树遍历的递归算法




前序遍历过程：

1. 访问根结点 $p$
2. 前序遍历左子树 $L$
3. 前序遍历右子树 $R$

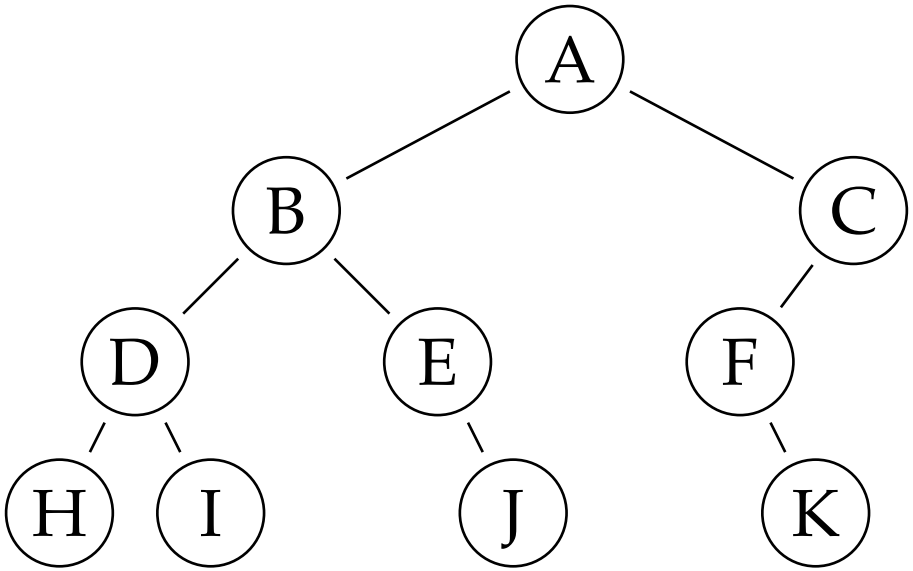
中序遍历过程：

1. 中序遍历左子树 $L$
2. 访问根结点 $p$
3. 中序遍历右子树 $R$

后序遍历过程：

1. 后序遍历左子树 $L$
2. 后序遍历右子树 $R$
3. 访问根结点 $p$

# 5.4.3 二叉树遍历示例




按不同的序遍历二叉树并按序输出结点编号：

- 前序遍历：
  A, B, D, H, I, E, J, C, F, K
- 中序遍历：
  H, D, I, B, E, J, A, F, K, C
- 后序遍历：
  H, I, D, J, E, B, K, F, C, A

观察根结点在遍历结果中的位置。

# 5.4.4 前序遍历算法

**伪码 5.4:** 二叉树前序遍历算法

```
TraversePreorder(T, Visit):
1   if T = ∅
2       return
3   Visit(T)
4   TraversePreorder(T.left)
5   TraversePreorder(T.right)
```

# 5.4.5 中序遍历算法

**伪码 5.5:** 二叉树中序遍历算法

```
TraverseInorder(T, Visit):
1   if T = ∅
2       return
3   TraverseInorder(T.left)
4   Visit(T)
5   TraverseInorder(T.right)
```

# 5.4.6 后序遍历算法

**伪码 5.6:** 二叉树后序遍历算法

```
TraversePostorder(T, Visit):
1   if T = ∅
2       return
3   TraversePostorder(T.left)
4   TraversePostorder(T.right)
5   Visit(T)
```

# 5.4.7 二叉树求高度算法

**伪码 5.7:** 二叉树求高度算法

```
Height(T):
    if T = ∅
        return 0
    h_L ← Height(T.left)
    h_R ← Height(T.right)
    return max(h_L, h_R) + 1
```

# 5.4.8 从遍历结果还原二叉树

问题：能否从遍历结果还原二叉树的结构?

考虑以下两棵二叉树的前序遍历结果。


结果均为 A, B。只用中序或后序能还原吗？问题在哪里?

思考题：如果用前序和中序两种遍历结果可以吗？后序和中序呢？前序和后序呢?

# 5.5 二叉树的应用

# 5.5.1 表达式解析与求值

问题. 给出一个由数、运算符和小括号组成的表达式，其中数由 0 到 9 的数字组成；运算符有加减乘除（除法结果取整），先乘除后加减；小括号可以嵌套。将该表达式解析成二叉树的形式，注意表达式可能会不合法，要能报错。

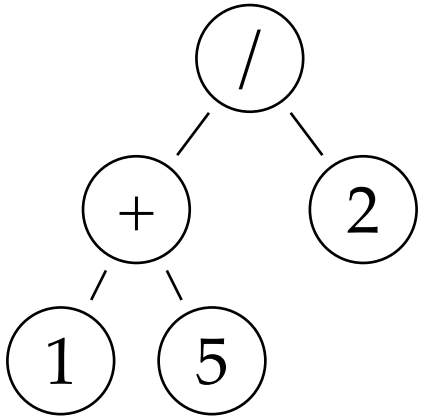
例如：

- `1+1` $\longrightarrow$ `2`
- `(1+5)/2` $\longrightarrow$ `3`
- `(8+0)/(9-(2*3+1))` $\longrightarrow$ `4`
- `(1+2` $\longrightarrow$ 报错

# 5.5.2 表示式树

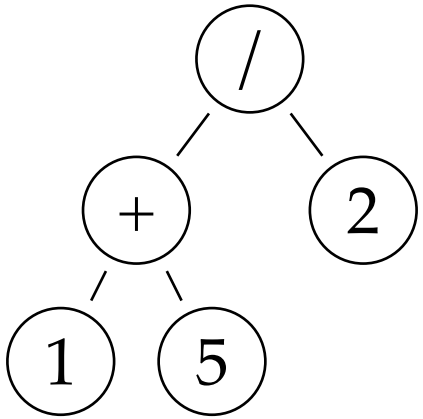
表达式具有层次结构，对于由二目运算构成的表达式，可以用二叉树来表示，称为**表达式树**。

例. 对于表达式 `(1+5)/2`，可以用如下的二叉树来表示：




在表达式树中，叶结点是参与运算的数，中间结点是运算符。

# 5.5.3 表达式树的遍历




对表达式树进行遍历：

- 前序遍历：`/ + 1 5 2`，**前缀表达式**
- 中序遍历（加括号）：`((1+5)/2)`，**中缀表达式**
- 后序遍历：`1 5 + 2 /`，**后缀表达式**

# 5.5.4 用二叉树进行表达式解析和计算

C++

```cpp
#include <iostream>
#include <optional>
#include <string>
#include <string_view>
#include <variant>

// 计算两个数的四则运算
std::optional<int> Calculate(int lhs, char op, int rhs) {
  switch (op) {
    case '+':
      return lhs + rhs;
    case '-':
      return lhs - rhs;
    case '*':
      return lhs * rhs;
    case '/':
```

# 5.5.4 用二叉树进行表达式解析和计算

```
17          if (rhs != 0) {
18            return lhs / rhs;
19          } else {
20            return {};  // 被零除
21          }
22        default:
23          return {};  // 不支持的运算
24      }
25  }
26
27  // 表达式树上结点的值：整型（操作数）或字符型（运算符）
28  using Token = std::variant<int, char>;
29
30  // 表达式树结点类
31  class Node {
32   public:
```

# 5.5.4 用二叉树进行表达式解析和计算

```
33      explicit Node(const Token& value, Node* left = nullptr, Node* right = nullptr)
34          : value_(value), left_(left), right_(right) {}
35
36      // 显示后缀表达式
37      void DumpPostfix() const {
38        if (left_) left_->DumpPostfix();
39        if (right_) right_->DumpPostfix();
40        if (std::holds_alternative<int>(value_)) {
41          std::cout << std::get<int>(value_) << ' ';
42        } else {
43          std::cout << std::get<char>(value_) << ' ';
44        }
45      }
46
47      // 表达式求值
48      std::optional<int> Evaluate() const {
```

# 5.5.4 用二叉树进行表达式解析和计算

```
49      if (std::holds_alternative<int>(value_)) {
50        return std::get<int>(value_);
51      } else {
52        auto lhs = left_->Evaluate();
53        if (!lhs) return {};  // 左子表达式计算错误
54        auto rhs = right_->Evaluate();
55        if (!rhs) return {};  // 右子表达式计算错误
56        return Calculate(*lhs, std::get<char>(value_), *rhs);
57      }
58    }
59
60    static Node* Parse(std::string_view expr);
61
62  protected:
63    Token value_;
64    Node* left_;
```

# 5.5.4 用二叉树进行表达式解析和计算

```
65    Node* right_;
66  };
67
68  // 解析一个表达式
69  Node* Node::Parse(std::string_view expr) {
70    if (expr.empty()) {
71      return nullptr;  // 非法表达式: 空串
72    }
73    int level = 0;  // 当前括号嵌套层数
74    std::size_t pos = std::string::npos;
75    for (std::size_t i = 0; i < expr.length(); i++) {
76      char ch = expr[i];
77      if (ch == '(') {
78        ++level;
79      } else if (ch == ')') {
80        if (--level < 0) {
```

# 5.5.4 用二叉树进行表达式解析和计算

```
81            return {};  // 非法表达式：括号不匹配
82          }
83        } else if (level == 0) {
84          if (ch == '+' || ch == '-') {
85            pos = i;  // 如果是加减号，一定比前面出现的优先级低
86          } else if ((ch == '*' || ch == '/') &&
87                     (pos == std::string::npos || expr[pos] == '*' ||
88                      expr[pos] == '/')) {
89            pos = i;  // 如果是乘除号，只比前面出现的乘除号优先级低
90          }
91        }
92      }
93      if (level != 0) {
94        return nullptr;  // 非法表达式：括号不匹配
95      }
96      if (pos != std::string::npos) {  // 找到了最低优先级的运算符
```

# 5.5.4 用二叉树进行表达式解析和计算

```
97        auto left = Parse(expr.substr(0, pos));
98        if (!left) {
99          return nullptr;  // 左子表达式解析错误
100       }
101       auto right = Parse(expr.substr(pos + 1));
102       if (!right) {
103         return nullptr;  // 右子表达式解析错误
104       }
105       return new Node(expr[pos], left, right);
106     } else if (expr.front() == '(') {  // (xxx) 的形式
107       return Parse(expr.substr(1, expr.length() - 2));
108     } else {  // 数
109       int num = std::stoi(std::string(expr));
110       return new Node(num);
111     }
112     return nullptr;  // 其他错误
```

# 5.5.4 用二叉树进行表达式解析和计算

```
113 }
114
115 int main() {
116   std::string expr;
117   std::getline(std::cin, expr);
118   auto tree = Node::Parse(expr);
119   if (tree) {
120     std::cout << "Postfix expression: ";
121     tree->DumpPostfix();
122     std::cout << '\n';
123     auto result = tree->Evaluate();
124     if (result) {
125       std::cout << "Result: " << *result << std::endl;
126     } else {
127       std::cout << "Invalid calculation" << std::endl;
128     }
129   } else {
```

# 5.5.4 用二叉树进行表达式解析和计算

```
130         std::cout << "Invalid expression" << std::endl;
131     }
132 }
```

# 5.5.5 表达式树的用途

表达式树能够天然地表示表达式的层次结构，便于后续的各种处理：求值、等价变换等。

对于更加复杂的语法，同样可以用树的结构来进行表示，称为**抽象语法树**（abstract syntax tree, AST），是编程语言的编译器和解释器内部实现的重要组成。

# 5.6 小结

# 5.6.1 本讲小结

- 树与二叉树的概念
- 树的实现：链式存储、静态存储
- 二叉树的遍历：前序、中序后序
- 二叉树的应用：表达式树