第八章 动态内存与数据结构

目录

学习目标

1. 掌握动态内存分配与回收方法以及智能指针的使用；
2. 掌握对象的拷贝控制方法；
3. 掌握线性链表、链栈和二叉树的特点及常用操作。

学习目标

1. 掌握动态内存分配与回收方法以及智能指针的使用；
2. 掌握对象的拷贝控制方法；
3. 掌握线性链表、链栈和二叉树的特点及常用操作。

问题

使用数组存放数量未知的元素时，我们必须采用大开小用的策略，这种策略不能实现按需分配，会造成存储空间的浪费

学习目标

1. 掌握动态内存分配与回收方法以及智能指针的使用；
2. 掌握对象的拷贝控制方法；
3. 掌握线性链表、链栈和二叉树的特点及常用操作。

问题

使用数组存放数量未知的元素时，我们必须采用大开小用的策略，这种策略不能实现按需分配，会造成存储空间的浪费

答案

本章介绍的动态内存分配技术的提出就是了为了解决这个问题

8.1 动态内存

下面两组示例代码中定义的对象的存储类型和生命周期有什么区别?

示例一

```
void fun(){
    int a(10);
    cout << a << endl;
}
```

示例二

```
struct A {
    const static double PI(3.14);
}
const double E = 2.72;
void fun(){
    static int a(10);
}
```

8.1 动态内存

下面两组示例代码中定义的对象的存储类型和生命周期有什么区别?

示例一

```
void fun(){
    int a(10);
    cout << a << endl;
}
```

局部自动对象

- 自动存储周期
- 在栈区被分配存储空间

示例二

```
struct A {
    const static double PI(3.14);
}
const double E = 2.72;
void fun(){
    static int a(10);
}
```

8.1 动态内存

下面两组示例代码中定义的对象的存储类型和生命周期有什么区别?

示例一

```
void fun(){
    int a(10);
    cout << a << endl;
}
```

局部自动对象

- 自动存储周期
- 在栈区被分配存储空间

示例二

```
struct A {
    const static double PI(3.14);
}
const double E = 2.72;
void fun(){
    static int a(10);
}
```

静态和全局对象

- 静态存储周期
- 在全局数据区被分配存储空间

8.1.1 创建动态对象

动态对象

由程序员创建并负责回收的对象，在动态内存（也称为自由存储区或堆）中被创建

- 动态存储周期
- 在全局数据区被分配存储空间

8.1.1 创建动态对象

动态对象

由程序员创建并负责回收的对象，在动态内存（也称为自由存储区或堆）中被创建

- 动态存储周期
- 在全局数据区被分配存储空间

C++ 语言使用 运算符 `new` 来分配动态内存：

8.1.1 创建动态对象

动态对象

由程序员创建并负责回收的对象，在动态内存（也称为自由存储区或堆）中被创建

- 动态存储周期
- 在全局数据区被分配存储空间

C++ 语言使用 运算符 `new` 来分配动态内存：

创建动态对象

```
int *pi = new int; // pi指向一个未初始化的int类型对象
```

说明

`new` 语句在自由存储区创建一个无名的 `int` 类型对象，并返回该对象的地址，存放于指针对象 `pi` 中

8.1.1 创建动态对象

动态对象

由程序员创建并负责回收的对象，在动态内存（也称为自由存储区或堆）中被创建

- 动态存储周期
- 在全局数据区被分配存储空间

C++ 语言使用 运算符 `new` 来分配动态内存：

创建动态对象

```
int *pi = new int; // pi指向一个未初始化的int类型对象
```

创建并初始化动态对象

```
int *pi = new int(10);
string *ps = new string("C++");
cout << *pi << "␣" << *ps << endl; // 输出: 10 C++
```

说明

`new` 语句在自由存储区创建一个无名的 `int` 类型对象，并返回该对象的地址，存放于指针对象 `pi` 中

8.1.2 释放动态内存

C++ 语言使用 运算符 `delete` 来释放动态内存：

8.1.2 释放动态内存

C++ 语言使用 运算符 `delete` 来释放动态内存：

释放动态内存—

```
TYPE *P = new TYPE;
delete p;
```

说明

- `p` 必须为一个指向动态对象的指针或者空指针
- 如果 `p` 指向的是类类型对象则调用其析构函数

8.1.2 释放动态内存

C++ 语言使用 运算符 `delete` 来释放动态内存：

释放动态内存—

```
TYPE *P = new TYPE;
delete p;
```

说明

- `p` 必须为一个指向动态对象的指针或者空指针
- 如果 `p` 指向的是类类型对象则调用其析构函数

释放一块非 `new` 分配的内存、同一内存多次释放或者使用一个已经释放的内存，其行为都是未定义的：

8.1.2 释放动态内存

C++ 语言使用 运算符 delete 来释放动态内存：

释放动态内存一

```
TYPE *P = new TYPE;
delete p;
```

说明

- p 必须为一个指向动态对象的指针或者空指针
- 如果 p 指向的是类类型对象则调用其析构函数

释放一块非 new 分配的内存、同一内存多次释放或者使用一个已经释放的内存，其行为都是未定义的：

释放动态内存二

```
int i, *p1 = &i, *p2 = new int(10);
delete p1; // 未定义，p1指向的对象为局部对象
p1 = p2;   // p1和p2指向同一个动态内存空间
delete p1; // 正确，释放p1所指向的动态内存空间
delete p2; // 错误：p2指向的动态内存已经被释放
```

8.1.2 释放动态内存

C++ 语言使用 运算符 `delete` 来释放动态内存：

释放动态内存一

```
TYPE *P = new TYPE;
delete p;
```

说明

- p 必须为一个指向动态对象的指针或者空指针
- 如果 p 指向的是类类型对象则调用其析构函数

释放一块非 new 分配的内存、同一内存多次释放或者使用一个已经释放的内存，其行为都是未定义的：

释放动态内存二

```
int i, *p1 = &i, *p2 = new int(10);
delete p1; // 未定义，p1指向的对象为局部对象
p1 = p2;   // p1和p2指向同一个动态内存空间
delete p1; // 正确，释放p1所指向的动态内存空间
delete p2; // 错误：p2指向的动态内存已经被释放
```

注意

编译器不能分辨指针所指向的对象是否为动态对象，也不能判断指针所指向的动态内存是否被释放

8.1.2 释放动态内存

空悬指针

对于一个指向动态内存的指针，在 `delete` 之后会依然保存已经释放的内存地址，此时的指针也称为空悬指针

8.1.2 释放动态内存

空悬指针

对于一个指向动态内存的指针，在 `delete` 之后会依然保存已经释放的内存地址，此时的指针也称为空悬指针

重置空悬指针

```
delete p;
p = nullptr; //p 不再指向任何对象
```

说明

空悬指针的危害类似于未初始化的野指针，应重置该指针为 `nullptr`

8.1.2 释放动态内存

内存泄漏

在使用动态对象的过程中，由于疏忽或错误造成无法释放已经不再使用的内存的情况称为内存泄漏

8.1.2 释放动态内存

内存泄漏

在使用动态对象的过程中，由于疏忽或错误造成无法释放已经不再使用的内存的情况称为内存泄漏

内存泄漏示例一

```
int i, *q = new int(2);
q = &i; // 错误: 发生内存泄漏
```

说明

当 q 指向对象 i 时，q 原来所指向的动态内存无法释放

8.1.2 释放动态内存

内存泄漏

在使用动态对象的过程中，由于疏忽或错误造成无法释放已经不再使用的内存的情况称为内存泄漏

内存泄漏示例一

```
int i, *q = new int(2);
q = &i; // 错误: 发生内存泄漏
```

说明

当 q 指向对象 i 时，q 原来所指向的动态内存无法释放

内存泄漏示例二

```
foo(614); //正确: 无内存泄漏
foo(105); //错误: 发生内存泄漏
```

foo 函数定义

```
void foo(int i) {
    int *p = new int(207);
    if ( *p > i) return;
    delete p;
}
```

说明

当调用 foo 函数的实参值小于 207 时，foo 函数体中 p 所指向的动态内存无法释放。

8.1.3 智能指针

通过 `new` 和 `delete` 分配和释放动态内存很容易产生空悬指针或内存泄漏

8.1.3 智能指针

通过 `new` 和 `delete` 分配和释放动态内存很容易产生空悬指针或内存泄漏

智能指针

在 C++11 新标准引入，用于控制动态对象的生命期，能够确保正确地自动释放动态内存，从而防止内存泄漏

8.1.3 智能指针

通过 `new` 和 `delete` 分配和释放动态内存很容易产生空悬指针或内存泄漏

智能指针

在 C++11 新标准引入，用于控制动态对象的生命期，能够确保正确地自动释放动态内存，从而防止内存泄漏

新标准在 `memory` 头文件中定义了三种不同类型的智能指针：

1. `unique_ptr` 独占所指向的对象
2. `shared_ptr` 允许多个指针指向一个对象
3. `weak_ptr` 是一种不控制所指对象生命期的智能指针，指向 `shared_ptr` 所管理的对象

8.1.3 智能指针 — unique_ptr

初始化一个 unique_ptr 必须采用直接初始化方式，因为接受指针参数的智能指针的构造函数为 explicit：

8.1.3 智能指针 — unique_ptr

初始化一个 unique_ptr 必须采用直接初始化方式，因为接受指针参数的智能指针的构造函数为 explicit：

unique_ptr 初始化

```
{
    unique_ptr<string> p1; // p1为nullptr
    unique_ptr<int> p2(new int(207));
} // p1和p2离开作用域，被销毁，同时释放其指向的动态内存
```

说明

当 p2 消亡时，p2 所指向的对象也会消亡，完成动态内存的自动释放

8.1.3 智能指针 — unique_ptr

初始化一个 unique_ptr 必须采用直接初始化方式，因为接受指针参数的智能指针的构造函数为 explicit：

unique_ptr 初始化

```
{
    unique_ptr<string> p1; // p1为nullptr
    unique_ptr<int> p2(new int(207));
} // p1和p2离开作用域，被销毁，同时释放其指向的动态内存
```

说明

当 p2 消亡时，p2 所指向的对象也会消亡，完成动态内存的自动释放

智能指针的使用和普通指针类似，解引用时返回其指向的对象：

8.1.3 智能指针 — unique_ptr

初始化一个 unique_ptr 必须采用直接初始化方式，因为接受指针参数的智能指针的构造函数为 explicit：

unique_ptr 初始化

```
{
    unique_ptr<string> p1; // p1为nullptr
    unique_ptr<int> p2(new int(207));
} // p1和p2离开作用域，被销毁，同时释放其指向的动态内存
```

说明

当 p2 消亡时，p2 所指向的对象也会消亡，完成动态内存的自动释放

智能指针的使用和普通指针类似，解引用时返回其指向的对象：

unique_ptr 使用一

```
unique_ptr<string> p1(new string("Mandy"));
if(p1 && p1->empty()) // 指针p1非空且其指向非空string
    *p1 = "Lisha"; // *p1为解引用
```

8.1.3 智能指针 — unique_ptr

unique_ptr 独自拥有所指向的动态对象，也就是说只能有一个 unique_ptr 指向给定的对象：

8.1.3 智能指针 — unique_ptr

unique_ptr 独自拥有所指向的动态对象，也就是说只能有一个 unique_ptr 指向给定的对象：

unique_ptr 使用二

```
unique_ptr<int> p1(new int(207));
unique_ptr<int> p2(p1); //错误
unique_ptr<int> p3;
p3 = p2; //错误
```

说明

unique_ptr 独占所指向的对象，不支持拷贝和赋值

8.1.3 智能指针 — unique_ptr

可以通过 release 或 reset 将一个动态内存的所有权从一个 unique_ptr 转移给另外一个 unique_ptr：

8.1.3 智能指针 — unique_ptr

可以通过 release 或 reset 将一个动态内存的所有权从一个 unique_ptr 转移给另外一个 unique_ptr：

unique_ptr 对象调用 release 成员函数

```
unique_ptr<int> p1(new int(207));
unique_ptr<int> p2(p1.release());
```

说明

release 函数将 p1 置为 nullptr 并返回 p1 原来的指针

8.1.3 智能指针 — unique_ptr

可以通过 release 或 reset 将一个动态内存的所有权从一个 unique_ptr 转移给另外一个 unique_ptr：

unique_ptr 对象调用 release 成员函数

```
unique_ptr<int> p1(new int(207));
unique_ptr<int> p2(p1.release());
```

说明

release 函数将 p1 置为 nullptr 并返回 p1 原来的指针

unique_ptr 对象调用 reset 成员函数

```
unique_ptr<int> p3(new int(105));
p3.reset(p2.release());
```

说明

reset 函数释放 p3 原来的动态内存，并指向 p2 释放出来的内存

8.1.3 智能指针 — shared_ptr

同 unique_ptr，必须使用直接初始化的形式来初始化一个 shared_ptr：

8.1.3 智能指针 — shared_ptr

同 unique_ptr，必须使用直接初始化的形式来初始化一个 shared_ptr：

shared_ptr 初始化一

```
shared_ptr<int> p1 = new int(105); //错误
shared_ptr<int> p2(new int(614)); //正确
```

8.1.3 智能指针 — shared_ptr

同 unique_ptr，必须使用直接初始化的形式来初始化一个 shared_ptr：

shared_ptr 初始化一

```
shared_ptr<int> p1 = new int(105); //错误
shared_ptr<int> p2(new int(614)); //正确
```

更安全的分配和使用动态内存的方法是调用 make_shared 标准库函数：

8.1.3 智能指针 — shared_ptr

同 unique_ptr，必须使用直接初始化的形式来初始化一个 shared_ptr：

shared_ptr 初始化一

```
shared_ptr<int> p1 = new int(105); //错误
shared_ptr<int> p2(new int(614)); //正确
```

更安全的分配和使用动态内存的方法是调用 make_shared 标准库函数：

shared_ptr 初始化二

```
shared_ptr<int> pi = make_shared<int>(10);
// 或者利用 auto 进行类型自动推导，简化书写：
auto pi = make_shared<int>(10);
```

说明

make_shared 是函数模板，在使用时必须要在尖括号中指定想要创建的对象类型

8.1.3 智能指针 — shared_ptr

与 unique_ptr 不同，shared_ptr 允许复制或赋值：

8.1.3 智能指针 — shared_ptr

与 unique_ptr 不同，shared_ptr 允许复制或赋值：

shared_ptr 对象调用成员函数 use_count —

```
auto p1 = make_shared<int>(10); // p1指向的对象只有p1
  一个引用者
cout << p1.use_count() << endl; // 输出1
auto p2(p1); // p1和p2共同指向同一个对象
cout << p1.use_count() << endl; // 输出2
```

说明

成员函数 use_count 返回与当前 shared_ptr 共享内存的智能指针的数量

8.1.3 智能指针 — shared_ptr

与 unique_ptr 不同，shared_ptr 允许复制或赋值：

shared_ptr 对象调用成员函数 use_count 一

```
auto p1 = make_shared<int>(10); // p1指向的对象只有p1
  一个引用者
cout << p1.use_count() << endl; // 输出1
auto p2(p1); // p1和p2共同指向同一个对象
cout << p1.use_count() << endl; // 输出2
```

shared_ptr 对象调用成员函数 use_count 二

```
auto p3 = make_shared<int>(11), p4(p3);
cout << p4.use_count() << endl; // 输出2
p3 = p1;
cout << p4.use_count() << "␣" << p1.use_count() <<
  endl; // 输出1 3
```

说明

成员函数 use_count 返回与当前 shared_ptr 共享内存的智能指针的数量

说明

对一个 shared_ptr 类型对象进行赋值时，赋值操作符将左操作数所指对象的引用计数减 1（如果引用计数减至为 0，则消亡其指向的对象），并将右操作数所指对象的引用计数加 1

8.1.3 智能指针 — weak_ptr

weak_ptr 是一种指向由 shared_ptr 管理的对象的智能指针，它的使用和析构都不会改变 shared_ptr 的引用计数

8.1.3 智能指针 — weak_ptr

weak_ptr 是一种指向由 shared_ptr 管理的对象的智能指针，它的使用和析构都不会改变 shared_ptr 的引用计数

weak_ptr 使用一

```
auto ps = make_shared<int>(10);
weak_ptr<int> pw(ps);
```

说明

ps 的引用计数不会改变

8.1.3 智能指针 — weak_ptr

weak_ptr 是一种指向由 shared_ptr 管理的对象的智能指针，它的使用和析构都不会改变 shared_ptr 的引用计数

weak_ptr 使用一

```
auto ps = make_shared<int>(10);
weak_ptr<int> pw(ps);
```

说明

ps 的引用计数不会改变

由于 weak_ptr 不会管理所指向对象的生命期，它所指向的对象可能是不存在的，因此不能直接使用 weak_ptr 访问对象，而必须调用 lock 函数

8.1.3 智能指针 — weak_ptr

weak_ptr 是一种指向由 shared_ptr 管理的对象的智能指针，它的使用和析构都不会改变 shared_ptr 的引用计数

weak_ptr 使用一

```
auto ps = make_shared<int>(10);
weak_ptr<int> pw(ps);
```

说明

ps 的引用计数不会改变

由于 weak_ptr 不会管理所指向对象的生命期，它所指向的对象可能是不存在的，因此不能直接使用 weak_ptr 访问对象，而必须调用 lock 函数

weak_ptr 使用二

```
if(auto p = pw.lock())
cout << *p;
```

说明

lock 函数检查指向的对象是否存在，如果对象存在则返回一个可用的 shared_ptr，否则返回一个存储 nullptr 的 shared_ptr

8.1.4 动态数组

使用 new 创建一个动态数组，例如：

8.1.4 动态数组

使用 new 创建一个动态数组，例如：

创建动态数组

```
int n = 5;
int *pa = new int[n];
```

说明

new 为动态数组分配指定大小的内存，并返回第一个元素的地址

8.1.4 动态数组

使用 new 创建一个动态数组，例如：

创建动态数组

```
int n = 5;
int *pa = new int[n];
```

创建并初始化动态数组

```
int *pa1 = new int[5]; // 5个未初始化的 int
int *pa2 = new int[5](); // 5个值为0的 int
int *pa3 = new int[5]{1,2,3,4,5};
```

说明

new 为动态数组分配指定大小的内存，并返回第一个元素的地址

说明

新标准下可以使用花括号来执行数组元素的初始化

8.1.4 动态数组

使用 new 创建一个动态数组，例如：

创建动态数组

```
int n = 5;
int *pa = new int[n];
```

说明

new 为动态数组分配指定大小的内存，并返回第一个元素的地址

创建并初始化动态数组

```
int *pa1 = new int[5]; // 5个未初始化的 int
int *pa2 = new int[5](); // 5个值为0的 int
int *pa3 = new int[5]{1,2,3,4,5};
```

说明

新标准下可以使用花括号来执行数组元素的初始化

动态数组的释放需要在 delete 前面加上一个空方括号：

8.1.4 动态数组

使用 new 创建一个动态数组，例如：

创建动态数组

```
int n = 5;
int *pa = new int[n];
```

说明

new 为动态数组分配指定大小的内存，并返回第一个元素的地址

创建并初始化动态数组

```
int *pa1 = new int[5]; // 5个未初始化的 int
int *pa2 = new int[5](); // 5个值为0的 int
int *pa3 = new int[5]{1,2,3,4,5};
```

说明

新标准下可以使用花括号来执行数组元素的初始化

动态数组的释放需要在 delete 前面加上一个空方括号：

释放动态数组

```
delete [] pa1;
```

说明

将逆序释放 pa1 指向的动态数组的每一个元素

8.2 拷贝控制

当定义一个类时，编译器将为我们自动合成默认的复制构造函数、赋值运算符和析构函数。

8.2 拷贝控制

当定义一个类时，编译器将为我们自动合成默认的复制构造函数、赋值运算符和析构函数。

如果类的数据成员含有动态对象，使用这些默认成员函数会有什么问题?

A 类型定义

```
class A {
    int *m_array; // 指向动态整型数组
public:
    A(size_t size) : m_array(new int[size]) {}
    A(const A& rhs) : m_array(rhs.m_array) {}
    ~A() {}
};
```

复制 A 类型对象

```
A a1(10); // 创建并初始化A类型对象a1
{
    A a2(a1); // 用a1复制构造a2
} // 出现错误
```

8.2 拷贝控制

当定义一个类时，编译器将为我们自动合成默认的复制构造函数、赋值运算符和析构函数。

如果类的数据成员含有动态对象，使用这些默认成员函数会有什么问题?

A 类型定义

```
class A {
    int *m_array; // 指向动态整型数组
public:
    A(size_t size) : m_array(new int[size]) {}
    A(const A& rhs) : m_array(rhs.m_array) {}
    ~A() {}
};
```

复制 A 类型对象

```
A a1(10); // 创建并初始化A类型对象a1
{
    A a2(a1); // 用a1复制构造a2
} // 出现错误
```

答案

- a2 执行默认的析构函数不会释放动态内存
- 执行默认复制构造后，a2 和 a1 的 m_array 指向同一个内存单元，若 a2 的析构函数正确，成功释放动态内存，则 a1 的 m_array 成为野指针，a1 析构时无法再次释放该内存地址

8.2.1 简单字符串类

定义一个简单的字符串类 MyStr：

MyStr 类定义

```
class MyStr {
    int m_length; // 字符数组的长度
    char *m_buff; // 指向动态字符数组
private:
    // 私有静态成员函数
    static int strlen(const char *ptr);
    static void strncpy(char *dest, const char *src,
      int n);
public:
    MyStr(const char *val=nullptr); // 默认构造函数
    ~MyStr() { delete[] m_buff; }
    int size() { return m_length; };
// 辅助函数声明
friend ostream& operator<<(ostream&, const MyStr&);
friend MyStr operator+(const MyStr&, const MyStr&);
};
```

说明

- strlen 获取 c 风格字符串长度
- strncpy 复制 src 指向的数组中前 n 个字符到 dest 指向的数组中
- operator<< 打印字符串
- operator+ 重载字符串相加运算
- 析构函数将动态数组 m_buff 释放

8.2.1 简单字符串类

MyStr 类的默认构造函数定义如下：

MyStr 类默认构造函数定义

```
MyStr::MyStr(const char *val):m_length(strlen(val)),
    m_buff(m_length>0?new char[m_length]:nullptr){
    strncpy(m_buff,val,m_length);
}
```

strlen 和 strncpy 函数定义

```
int MyStr::strlen(const char *ptr) {
    int len = 0;
    while (ptr && *ptr++ != '\0')
        ++len;
    return len;
}

void MyStr::strncpy(char *dest, const char *src, int
   n) {
    for (int i = 0; i < n; ++i)
        dest[i] = src[i];
}
```

说明

- 通过 strlen 函数获取形参 val 指针指向的字符串中字符的个数
- 如果 val 指向非空字符串，则利用 new 函数分配相应大小的内存
- 如果 val 为空指针，m_buff 则为空，将不执行动态内存分配
- 利用函数 strncpy 完成字符串的复制

8.2.1 简单字符串类

MyStr 类的辅助函数定义如下：

MyStr 类辅助函数定义

```
ostream& operator<<(ostream& os, const MyStr& s){
    for (int i = 0; i < s.m_length; ++i)
        os << s.m_buff[i];
    return os;
}

MyStr operator+(const MyStr &s1, const MyStr &s2){
    MyStr res;
    res.m_length = s1.m_length + s2.m_length;
    res.m_buff = new char[res.m_length];
    MyStr::strncpy(res.m_buff, s1.m_buff, s1.m_length
      );
    MyStr::strncpy(res.m_buff + s1.m_length, s2.
      m_buff, s2.m_length);
    return res; //返回局部对象 res
}
```

说明

- 在输出运算符 << 函数体内，动态字符数组中的字符逐个写入到输出流对象 os 中，并返回 os 的引用
- 重载的 MyStr 类运算符 +，将两个形参 MyStr 类对象中的字符连接起来，形成一个新的 MyStr 类对象 res，并以值的形式返回 res 的副本

8.2.2 复制与赋值

回到字符串类 MyStr 的定义：

MyStr 类定义

```
class MyStr {
    int m_length; // 字符数组的长度
    char *m_buff; // 指向动态字符数组
    // 其他成员
    ...
};
```

问题

MyStr 类含有动态对象数据成员 m_buff，默认的复制构造将如之前 A 类型对象的复制构造一样出现问题

8.2.2 复制与赋值

回到字符串类 MyStr 的定义：

MyStr 类定义

```
class MyStr {
    int m_length; // 字符数组的长度
    char *m_buff; // 指向动态字符数组
    // 其他成员
    ...
};
```

问题

MyStr 类含有动态对象数据成员 m_buff，默认的复制构造将如之前 A 类型对象的复制构造一样出现问题

MyStr 类对象复制

```
{
    MyStr s1("dynamic"), s2(s1), s3;
    s3 = s1;
} // 错误
```

说明

- 对于指针成员 m_buff，将复制指针本身的值，而非指针所指向的对象的值
- s1、s2 和 s3 的 m_buff 指向同一个内存地址，析构时重复释放

8.2.2 复制与赋值

为了解决上述问题，需要显式定义 `MyStr` 类复制构造函数和赋值运算符重载：

8.2.2 复制与赋值

为了解决上述问题，需要显式定义 MyStr 类复制构造函数和赋值运算符重载：

MyStr 类复制构造函数定义

```
MyStr::MyStr(const MyStr &rhs):m_length(rhs.m_length),
    m_buff(m_length>0 ? new char[m_length] : nullptr){
    strncpy(m_buff, rhs.m_buff, m_length); //复制数据
}
```

说明

首先要为待创建对象分配内存空间，然后将目标对象的内容复制到待创建对象中

8.2.2 复制与赋值

为了解决上述问题，需要显式定义 MyStr 类复制构造函数和赋值运算符重载：

MyStr 类复制构造函数定义

```
MyStr::MyStr(const MyStr &rhs):m_length(rhs.m_length),
    m_buff(m_length>0 ? new char[m_length] : nullptr){
    strncpy(m_buff, rhs.m_buff, m_length); //复制数据
}
```

说明

首先要为待创建对象分配内存空间，然后将目标对象的内容复制到待创建对象中

MyStr 类 operator= 运算符重载

```
MyStr& MyStr::operator=(const MyStr &rhs){
    if (this != &rhs){ // 此判断不能缺少
        delete [] m_buff; // 释放原来的内存
        m_length = rhs.m_length;
        m_buff = new char[m_length]; // 重新分配内存
        strncpy(m_buff, rhs.m_buff, m_length);//复制数据
    }
    return *this;
}
```

8.2.2 复制与赋值

为了解决上述问题，需要显式定义 MyStr 类复制构造函数和赋值运算符重载：

MyStr 类复制构造函数定义

```
MyStr::MyStr(const MyStr &rhs):m_length(rhs.m_length),
    m_buff(m_length>0 ? new char[m_length] : nullptr){
    strncpy(m_buff, rhs.m_buff, m_length); //复制数据
}
```

MyStr 类 operator= 运算符重载

```
MyStr& MyStr::operator=(const MyStr &rhs){
    if (this != &rhs){ // 此判断不能缺少
        delete [] m_buff; // 释放原来的内存
        m_length = rhs.m_length;
        m_buff = new char[m_length]; // 重新分配内存
        strncpy(m_buff, rhs.m_buff, m_length);//复制数据
    }
    return *this;
}
```

说明

首先要为待创建对象分配内存空间，然后将目标对象的内容复制到待创建对象中

问题

为什么不能缺少 if 语句?

8.2.2 复制与赋值

为了解决上述问题，需要显式定义 MyStr 类复制构造函数和赋值运算符重载：

MyStr 类复制构造函数定义

```
MyStr::MyStr(const MyStr &rhs):m_length(rhs.m_length),
    m_buff(m_length>0 ? new char[m_length] : nullptr){
    strncpy(m_buff, rhs.m_buff, m_length); //复制数据
}
```

MyStr 类 operator= 运算符重载

```
MyStr& MyStr::operator=(const MyStr &rhs){
    if (this != &rhs){ // 此判断不能缺少
        delete [] m_buff; // 释放原来的内存
        m_length = rhs.m_length;
        m_buff = new char[m_length]; // 重新分配内存
        strncpy(m_buff, rhs.m_buff, m_length);//复制数据
    }
    return *this;
}
```

说明

首先要为待创建对象分配内存空间，然后将目标对象的内容复制到待创建对象中

问题

为什么不能缺少 if 语句?

答案

避免对自己的复制，否则此情况下会对已被释放的动态内存进行复制，引发错误

8.2.3 移动对象

从代码性能角度考虑，创建 s3 的过程有什么不足?

创建 MyStr 类对象

```
MyStr s1("move␣"), s2("constructor");
MyStr s3(s1+s2);
```

MyStr 类 operator+ 运算符重载部分定义

```
MyStr operator+(const MyStr &s1, const MyStr &s2){
    MyStr res;
    /*...*/
    return res;
}
```

MyStr 类复制构造函数定义

```
MyStr::MyStr(const MyStr &rhs):m_length(rhs.m_length),
    m_buff(m_length>0 ? new char[m_length] : nullptr){
    strncpy(m_buff, rhs.m_buff, m_length);
}
```

8.2.3 移动对象

从代码性能角度考虑，创建 s3 的过程有什么不足?

创建 MyStr 类对象

```
MyStr s1("move␣"), s2("constructor");
MyStr s3(s1+s2);
```

MyStr 类 operator+ 运算符重载部分定义

```
MyStr operator+(const MyStr &s1, const MyStr &s2){
    MyStr res;
    /*...*/
    return res;
}
```

MyStr 类复制构造函数定义

```
MyStr::MyStr(const MyStr &rhs):m_length(rhs.m_length),
    m_buff(m_length>0 ? new char[m_length] : nullptr){
    strncpy(m_buff, rhs.m_buff, m_length);
}
```

答案

- s1+s2 返回的是一个临时对象，属于右值
- 在复制构造中，s3 新开辟了动态内存空间，复制临时对象申请的动态内存中的内容，而临时对象申请的动态内存又马上被释放

8.2.3 移动对象 — 移动构造函数

为了提高性能，应该定义更精准匹配的参数为右值引用的移动构造：

MyStr 类移动构造函数定义

```
MyStr::MyStr(MyStr &&rhs):m_length(rhs.m_length),
    m_buff(rhs.m_buff){
    rhs.m_buff = nullptr; // 置为空指针
    rhs.m_length = 0;
}
```

8.2.3 移动对象 — 移动构造函数

为了提高性能，应该定义更精准匹配的参数为右值引用的移动构造：

MyStr 类移动构造函数定义

```
MyStr::MyStr(MyStr &&rhs):m_length(rhs.m_length),
    m_buff(rhs.m_buff){
    rhs.m_buff = nullptr; // 置为空指针
    rhs.m_length = 0;
}
```

说明

相比复制构造函数，直接接管给临时对象分配的动态内存。没有分配新的动态内存，也没有对动态内存数据的复制。

MyStr 类复制构造函数定义

```
MyStr::MyStr(const MyStr &rhs):m_length(rhs.m_length),
    m_buff(m_length>0 ? new char[m_length] : nullptr){
    strncpy(m_buff, rhs.m_buff, m_length);
}
```

8.2.3 移动对象 — 移动构造函数

为了提高性能，应该定义更精准匹配的参数为右值引用的移动构造：

MyStr 类移动构造函数定义

```
MyStr::MyStr(MyStr &&rhs):m_length(rhs.m_length),
    m_buff(rhs.m_buff){
    rhs.m_buff = nullptr; // 置为空指针
    rhs.m_length = 0;
}
```

说明

相比复制构造函数，直接接管给临时对象分配的动态内存。没有分配新的动态内存，也没有对动态内存数据的复制。

MyStr 类复制构造函数定义

```
MyStr::MyStr(const MyStr &rhs):m_length(rhs.m_length),
    m_buff(m_length>0 ? new char[m_length] : nullptr){
    strncpy(m_buff, rhs.m_buff, m_length);
}
```

问题

移动构造中若没有置空临时对象 rhs 的指针成员会怎样?

8.2.3 移动对象 — 移动构造函数

为了提高性能，应该定义更精准匹配的参数为右值引用的移动构造：

MyStr 类移动构造函数定义

```
MyStr::MyStr(MyStr &&rhs):m_length(rhs.m_length),
    m_buff(rhs.m_buff){
    rhs.m_buff = nullptr; // 置为空指针
    rhs.m_length = 0;
}
```

MyStr 类复制构造函数定义

```
MyStr::MyStr(const MyStr &rhs):m_length(rhs.m_length),
    m_buff(m_length>0 ? new char[m_length] : nullptr){
    strncpy(m_buff, rhs.m_buff, m_length);
}
```

说明

相比复制构造函数，直接接管给临时对象分配的动态内存。没有分配新的动态内存，也没有对动态内存数据的复制。

问题

移动构造中若没有置空临时对象 rhs 的指针成员会怎样?

答案

新建对象的 m_buff 成为野指针

8.2.3 移动对象 — 移动赋值运算符

和移动构造函数的思想类似，可以为 MyStr 类定义一个移动赋值运算符：

MyStr 类重载移动赋值运算符

```
MyStr& MyStr::operator=(MyStr &&rhs) {
    if (this != &rhs) {
        delete[] m_buff;
        m_length = rhs.m_length;
        m_buff = rhs.m_buff;
        rhs.m_buff = nullptr; // 置为空指针
        rhs.m_length = 0;
    }
    return *this;
}
```

说明

和移动构造函数类似，移动赋值运算符也可以避免数据的复制，提高程序的性能

调用 MyStr 类移动赋值运算符

```
MyStr s1("move␣"), s2("assignment"), s3;
s3 = s1 + s2;
```

8.3 线性链表

线性链表

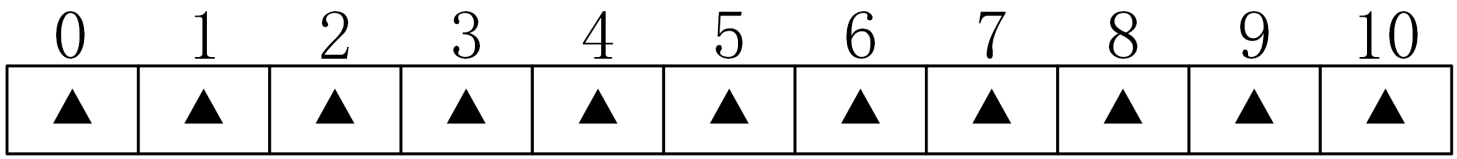
也称为单链表，是由有限个元素组成的有序集合，除了第一个元素和最后一个元素外，每个元素均有一个前驱和一个后继。

8.3 线性链表

线性链表

也称为单链表，是由有限个元素组成的有序集合，除了第一个元素和最后一个元素外，每个元素均有一个前驱和一个后继。

数组是一种线性结构，在逻辑结构上相邻的元素在物理结构上也相邻：

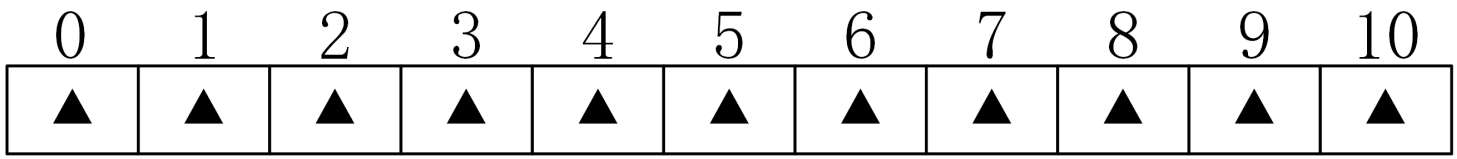


8.3 线性链表

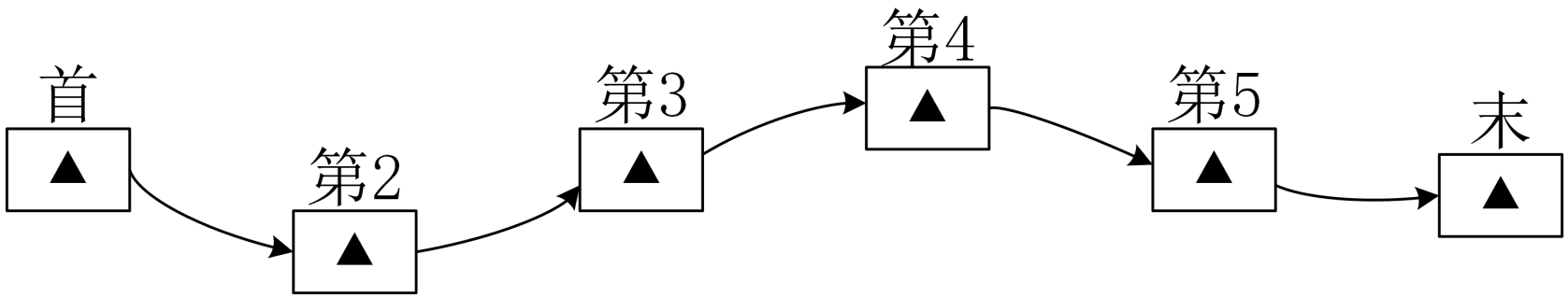
线性链表

也称为单链表，是由有限个元素组成的有序集合，除了第一个元素和最后一个元素外，每个元素均有一个前驱和一个后继。

数组是一种线性结构，在逻辑结构上相邻的元素在物理结构上也相邻：



线性链表为链式结构，在逻辑结构上相邻的元素在物理结构上不要求相邻：



8.3.1 链表表示

每个数据元素占用一个结点，一个结点包含一个数据域和一个指针域，其中指针域存放下一个结点的地址：

8.3.1 链表表示

每个数据元素占用一个结点，一个结点包含一个数据域和一个指针域，其中指针域存放下一个结点的地址：

利用类模板来定义一个结点：

Node 类模板定义

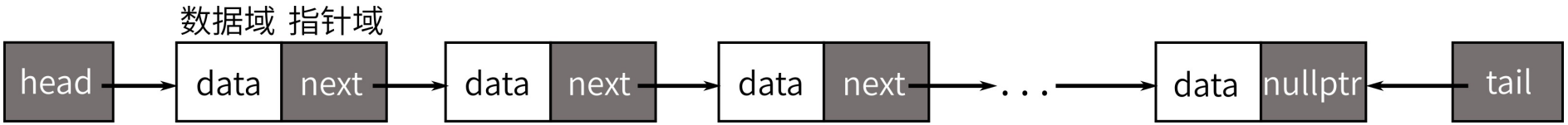
```
template<typename T>
class Node{
    T m_data; //数据域
    Node *m_next = nullptr; //指向下一个结点的指针
public:
    Node(const T &val) :m_data(val) { }
    const T& data() const{ return m_data; }
    T& data() { return m_data; }
    Node* next() { return m_next; }
};
```

说明

- 成员 m_next 为指向 Node 类型的指针。类允许包含指向其自身类型的指针或引用
- 提供两个版本的 data 函数以支持 const 和非 const 对象的数据访问

8.3.1 链表表示

单链表的成员包含两个指针，指针 head 指向表头结点，指针 tail 指向表尾结点：



8.3.1 链表表示

单链表的成员包含两个指针，指针 head 指向表头结点，指针 tail 指向表尾结点：

单链表类模板的定义如下：

SList 类模板定义

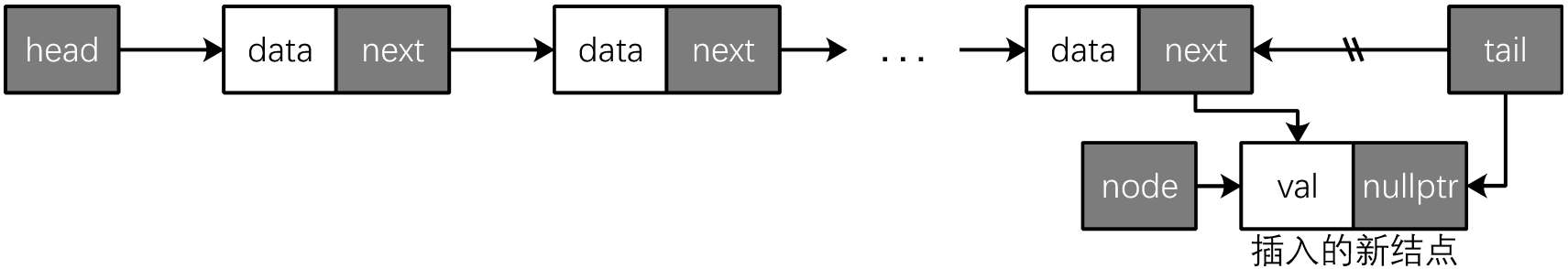
```
template<typename T>
class SList {
    Node<T> *m_head= nullptr, *m_tail= nullptr;
public:
    SList()= default; // 使用默认构造函数
    ~SList();
    void clear();
    void push_back(const T &val);
    Node<T>* insert(Node<T> *pos, const T &val);
    void erase(const T &val);
    Node<T>* find(const T &val);
};
```

说明

- clear 函数用来清空链表所有元素
- push_back 函数为尾插操作
- insert 函数在位置 pos 后插入一个新结点
- erase 函数删除第一个元素值为 val 的元素
- find 函数返回第一个值为 val 的元素的地址

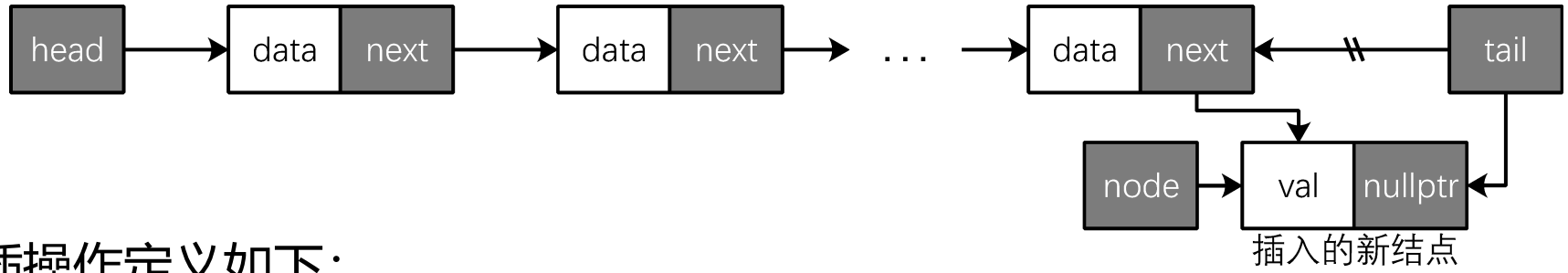
8.3.2 插入操作 — 尾插

尾插操作将新结点插入到链表的表尾：



8.3.2 插入操作 — 尾插

尾插操作将新结点插入到链表的表尾：



尾插操作定义如下：

push_back 函数定义

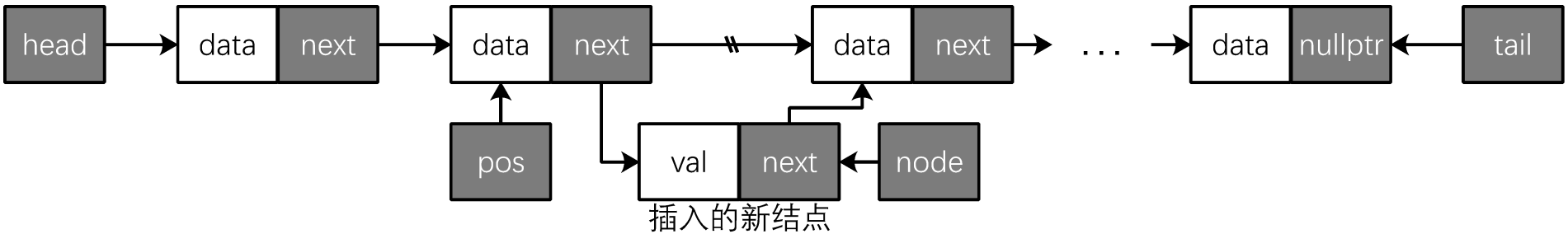
```
template<typename T>
void SList<T>::push_back(const T &val) {
    Node<T> *node = new Node<T>(val); // 创建新结点
    if (m_head == nullptr)
        m_head = m_tail= node;
    else {
        m_tail->m_next = node;
        m_tail = node;
    }
}
```

说明

- 使用形参的数据创建新结点
- 如果为空，将创建的结点作为头结点（也是尾结点）
- 否则，将尾结点指向该结点，并将尾指针后移，使其指向新的尾结点

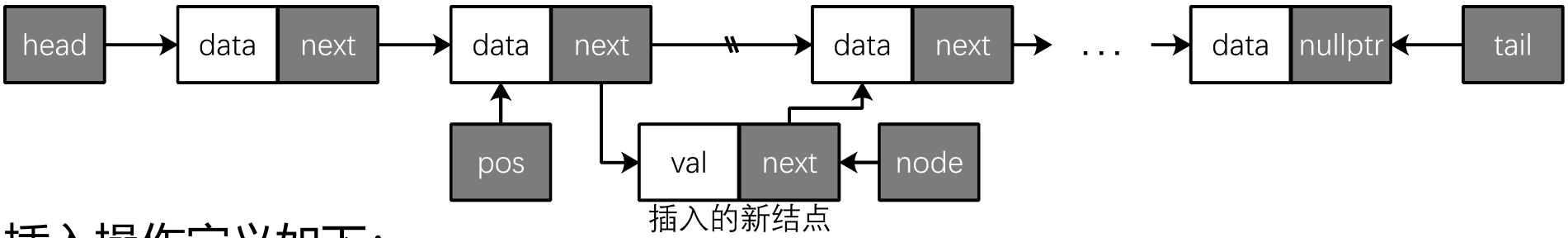
8.3.2 插入操作 — 指定位置插入

插入操作将新结点插入到链表的指定位置：



8.3.2 插入操作 — 指定位置插入

插入操作将新结点插入到链表的指定位置：



插入操作定义如下：

insert 函数定义

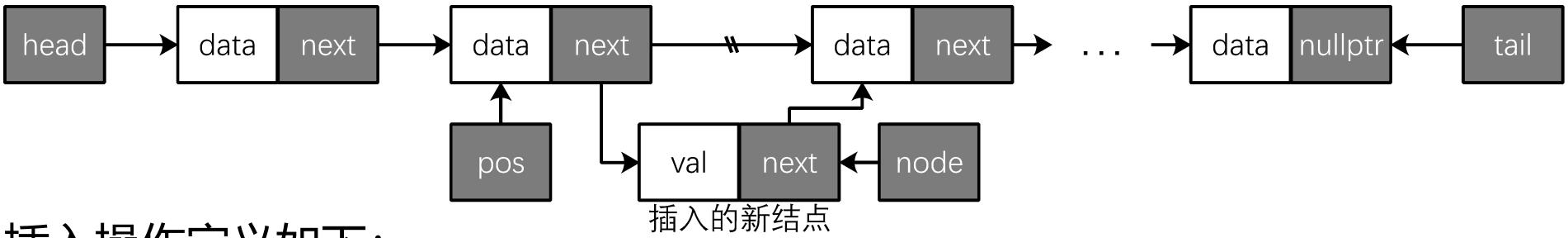
```
template<typename T>
Node<T>* SList<T>::insert(Node<T> *pos, const T &val
  ) {
    Node<T> *node = new Node<T>(val); // 创建新结点
    node->m_next = pos->m_next;
    pos->m_next = node;
    if (pos == m_tail) // 判断pos是否为尾结点
        m_tail = node;
    return node;
}
```

说明

- 将新结点的指针域指向 pos 的后继，再将 pos 的后继修改为 node
- 如果 pos 为尾结点，需要修改尾指针指向新结点

8.3.2 插入操作 — 指定位置插入

插入操作将新结点插入到链表的指定位置：



插入操作定义如下：

insert 函数定义

```
template<typename T>
Node<T>* SList<T>::insert(Node<T> *pos, const T &val
  ) {
    Node<T> *node = new Node<T>(val); // 创建新结点
    node->m_next = pos->m_next;
    pos->m_next = node;
    if (pos == m_tail) // 判断pos是否为尾结点
        m_tail = node;
    return node;
}
```

说明

- 将新结点的指针域指向 pos 的后继，再将 pos 的后继修改为 node
- 如果 pos 为尾结点，需要修改尾指针指向新结点

注意

pos 必须为非空链表的某一个结点指针

8.3.2 插入操作 — 指定位置插入

利用成员函数 find 找到要插入的位置，find 的实现如下：

insert 函数定义

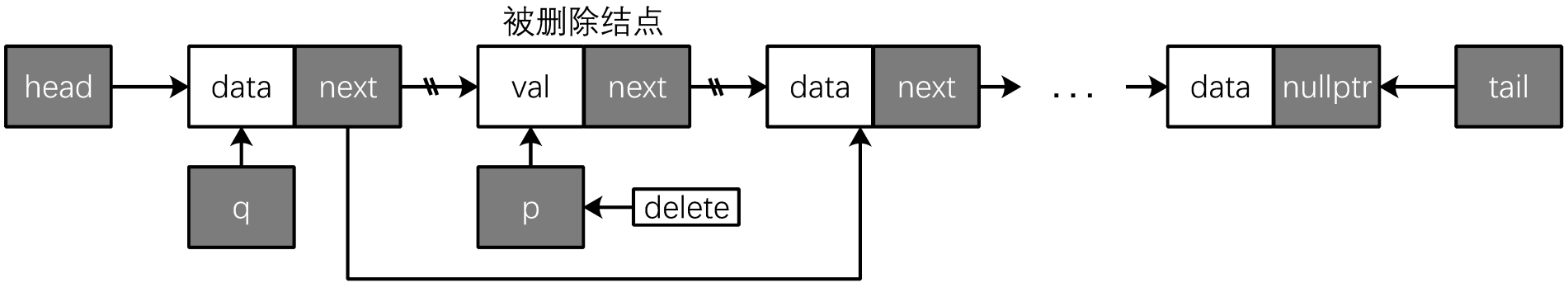
```
template<typename T>
Node<T>* SList<T>::find(const T &val) {
    Node<T> *p = m_head;
    while (p != nullptr && p->m_data != val)
        p = p->m_next;
    return p;
}
```

说明

- 从表头开始扫描，逐个元素进行匹配
- 如果找到则返回此元素的地址
- 否则返回一个空指针

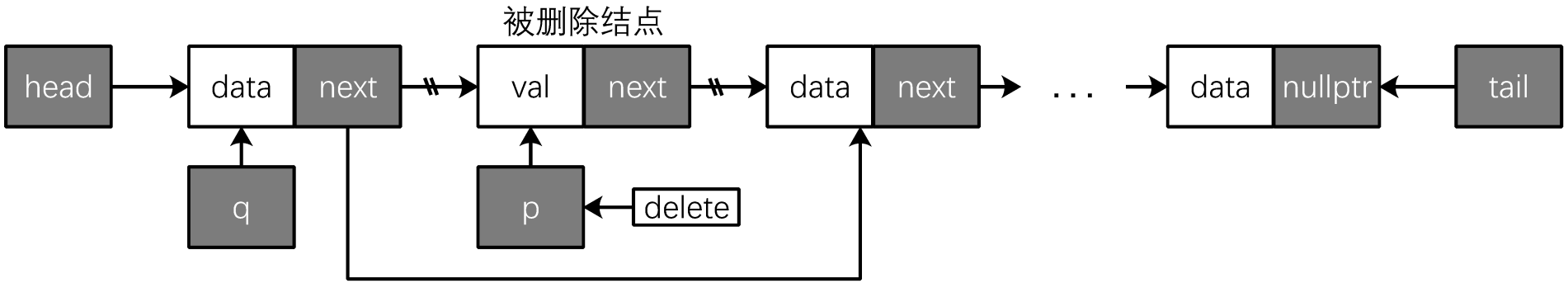
8.3.3 删除操作

成员函数 `erase` 根据指定的内容，删除在链表中第一次出现的元素：



8.3.3 删除操作

成员函数 erase 根据指定的内容，删除在链表中第一次出现的元素：



erase 函数定义

```
template<typename T>
void SList<T>::erase(const T &val) {
    Node<T> *p = m_head, *q = p;
    while (p != nullptr && p->m_data != val) {
        q = p; // 指针q指向p
        p = p->m_next; // 指针p后移
    }
    if (p) q->m_next = p->m_next;
    if (p == m_tail) m_tail = q;
    if (p == m_head) m_head = nullptr;
    delete p;
}
```

说明

- 如果找到，即指针 p 非空，将其从链表中移除
- 如果 p 为表尾元素，修改 tail 指针
- 如果 p 为表头元素，修改 head 指针为空指针

8.3.4 清空链表

成员函数 clear 实现从表头开始，逐个移除链表中每个结点并释放其内存

clear 函数定义

```
template<typename T>
void SList<T>::clear() {
    Node<T> *p = nullptr;
    while (m_head != nullptr) {
        p = m_head; //p 指向当前表头结点
        m_head = m_head->m_next; //表头结点后移
    delete p; //释放 p 所指向的内存
    }
    m_tail = nullptr;//将尾指针 tail 置空
}
```

8.3.4 清空链表

成员函数 clear 实现从表头开始，逐个移除链表中每个结点并释放其内存

clear 函数定义

```
template<typename T>
void SList<T>::clear() {
    Node<T> *p = nullptr;
    while (m_head != nullptr) {
        p = m_head; //p 指向当前表头结点
        m_head = m_head->m_next; //表头结点后移
    delete p; //释放 p 所指向的内存
    }
    m_tail = nullptr;//将尾指针 tail 置空
}
```

在析构函数里面，可以直接调用 clear 函数释放链表的内存空间：

SList 析构函数定义

```
template<typename T>
SList<T>::~SList() {
    clear();
}
```

8.3.5 打印链表

为了像内置类型一样输出，需要重载输出运算符，并将其声明为 SList 的友元：

重载输出运算符声明及友元声明

```
template<typename T>
ostream& operator<<(ostream&,const SList<T>&);

template<typename T>
class SList {
friend ostream& operator<< <T>(ostream&,const SList<T>&);
//其它成员定义保持不变
};
```

重载输出运算符定义

```
template<typename T>
ostream& operator<<(ostream &os, const SList<T>& list) {
    Node<T> *p = list.m_head;
    while (p != nullptr) {
        os << p->data() << "␣";
        p = p->next();
    }
    return os;
}
```

8.3.5 打印链表

为了像内置类型一样输出，需要重载输出运算符，并将其声明为 SList 的友元：

重载输出运算符声明及友元声明

```
template<typename T>
ostream& operator<<(ostream&,const SList<T>&);

template<typename T>
class SList {
friend ostream& operator<< <T>(ostream&,const SList<T>&);
//其它成员定义保持不变
};
```

注意

友元关系被限定在相同类型实例化的输出运算符和 SList 之间

重载输出运算符定义

```
template<typename T>
ostream& operator<<(ostream &os, const SList<T>& list) {
    Node<T> *p = list.m_head;
    while (p != nullptr) {
        os << p->data() << "␣";
        p = p->next();
    }
    return os;
}
```

8.3.6 拷贝控制与友元声明

回顾类模板 Node 的定义，如果使用默认的复制与赋值操作会有什么后果?

Node 类模板部分定义

```
template<typename T>
class Node{
    T m_data; //数据域
    Node *m_next = nullptr; //指针域
    /*...*/
};
```

8.3.6 拷贝控制与友元声明

回顾类模板 Node 的定义，如果使用默认的复制与赋值操作会有什么后果?

Node 类模板部分定义

```
template<typename T>
class Node{
    T m_data; //数据域
    Node *m_next = nullptr; //指针域
    /*...*/
};
```

答案

根据链表中的一个结点创建一个新结点（或赋值操作）时，会导致两个结点的指针域指向链表中的同一个结点

8.3.6 拷贝控制与友元声明

回顾类模板 Node 的定义，如果使用默认的复制与赋值操作会有什么后果?

Node 类模板部分定义

```
template<typename T>
class Node{
    T m_data; //数据域
    Node *m_next = nullptr; //指针域
    /*...*/
};
```

答案

根据链表中的一个结点创建一个新结点（或赋值操作）时，会导致两个结点的指针域指向链表中的同一个结点

应该利用 delete 关键字禁止 Node 类型实例的复制与赋值：

Node 类模板部分定义

```
template<typename T>
class Node {
public:
    Node(const Node &rhs) = delete;
    Node& operator =(const Node &rhs) = delete;
    // 其它成员定义保持不变
};
```

8.3.6 拷贝控制与友元声明

类似的，也不允许 SList 类型实例的复制与赋值：

SList 类模板部分定义

```
template<typename T>
class SList {
public:
    SList(const SList &) = delete;
    SList& operator=(const SList &) = delete;
    //其它成员定义保持不变
};
```

8.3.6 拷贝控制与友元声明

类似的，也不允许 SList 类型实例的复制与赋值：

SList 类模板部分定义

```
template<typename T>
class SList {
public:
    SList(const SList &) = delete;
    SList& operator=(const SList &) = delete;
    //其它成员定义保持不变
};
```

此外，还需要将类模板 SList 声明为 Node 的友元，否则有什么问题？

Node 类模板部分定义

```
template<typename T>
class SList; //前向声明

template<typename T>
class Node {
friend class SList<T>; // 将SList声明为Node的友元
    // 其它成员定义保持不变
};
```

8.3.6 拷贝控制与友元声明

类似的，也不允许 SList 类型实例的复制与赋值：

SList 类模板部分定义

```
template<typename T>
class SList {
public:
    SList(const SList &) = delete;
    SList& operator=(const SList &) = delete;
    //其它成员定义保持不变
};
```

此外，还需要将类模板 SList 声明为 Node 的友元，否则有什么问题？

Node 类模板部分定义

```
template<typename T>
class SList; //前向声明

template<typename T>
class Node {
friend class SList<T>; // 将SList声明为Node的友元
    // 其它成员定义保持不变
};
```

答案

在 SList 的成员函数中将没有权限直接使用 m_next

8.3.6 拷贝控制与友元声明

创建一个存放整型元素的单链表对尾插、指定位置插入、删除等操作进行测试：

使用 SList 类模板

```
SList<int> l;
int val;
while (cin >> val) { // 输入10 20 30三个数据
    l.push_back(val);
}
cout << l << endl;
Node<int> *pos = l.find(20);
l.insert(pos, 25);
cout << l << endl;
l.erase(25);
cout << l << endl;
```

问题

输出结果是什么？

8.3.6 拷贝控制与友元声明

创建一个存放整型元素的单链表对尾插、指定位置插入、删除等操作进行测试：

使用 SList 类模板

```
SList<int> l;
int val;
while (cin >> val) { // 输入10 20 30三个数据
    l.push_back(val);
}
cout << l << endl;
Node<int> *pos = l.find(20);
l.insert(pos, 25);
cout << l << endl;
l.erase(25);
cout << l << endl;
```

问题

输出结果是什么?

答案

输出结果为：

10 20 30

10 20 25 30

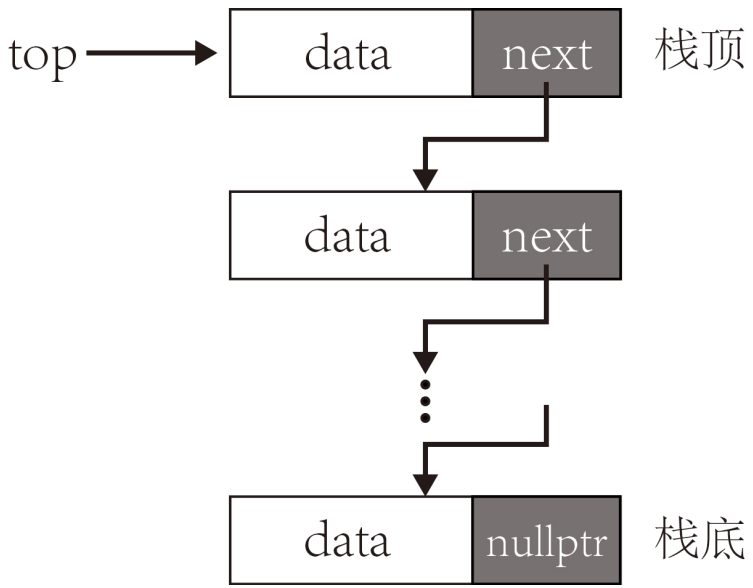
10 20 30

8.4 链栈

链栈

栈是一种只能在一端进行插入和删除操作的线性表。栈也称为后进先出线性表。

允许进行插入和删除操作的一端称为栈顶，另一端称为栈底。



8.4.1 链栈表示与操作

链栈支持进栈、出栈、清空、取栈顶元素和判断是否为空等操作。
模板类 Stack 定义如下：

Stack 类模板定义

```
template<typename T>
class Stack {
    Node<T> *m_top = nullptr;
public:
    Stack() = default; //使用默认构造函数
    Stack(const Stack &) = delete;
    Stack& operator=(const Stack &) = delete;
    ~Stack();
    void clear();
    void push(const T &val);
    void pop();
    bool empty() const { return m_top == nullptr; }
    const T& top() { return m_top->m_data; }
};
```

说明

- 类似 SList，Stack 类模板禁止复制和赋值操作
- clear 函数执行清空栈操作
- push 函数执行进栈操作
- pop 函数执行出栈操作
- empty 函数判断栈是否为空
- top 函数取栈顶元素。返回栈顶元素的 const 引用，意味着只能对栈顶元素进行读操作，不能执行写操作

8.4.1 链栈表示与操作 — 进栈与出栈操作

进栈操作的实现如下：

push 函数定义

```
template<typename T>
void Stack<T>::push(const T &val) {
    Node<T> *node = new Node<T>(val);
    node->m_next = m_top;
    m_top = node;
}
```

说明

创建一个新结点 node，然后将结点 node 压栈，最后修改栈顶指针，使其指向新的栈顶结点

8.4.1 链栈表示与操作 — 进栈与出栈操作

进栈操作的实现如下：

push 函数定义

```
template<typename T>
void Stack<T>::push(const T &val) {
    Node<T> *node = new Node<T>(val);
    node->m_next = m_top;
    m_top = node;
}
```

说明

创建一个新结点 node，然后将结点 node 压栈，最后修改栈顶指针，使其指向新的栈顶结点

出栈操作的实现如下：

pop 函数定义

```
template<typename T>
void Stack<T>::pop() {
    Node<T> *p = m_top;
    m_top = m_top->m_next;
    delete p;
}
```

说明

先把栈顶元素地址保存起来，然后修改栈顶指针，使其指向新的栈顶元素，最后通过保存的指针释放原来栈顶元素的内存

8.4.1 链栈表示与操作 — 清空操作

清空操作的实现如下：

push 函数定义

```
template<typename T>
void Stack<T>::clear() {
    Node<T> *p = nullptr;
    while (m_top != nullptr) {
        p = m_top;
        m_top = m_top->m_next;
        delete p;
    }
}
```

说明

利用出栈的操作，逐个释放每个元素的内存空间

8.4.1 链栈表示与操作 — 清空操作

清空操作的实现如下：

push 函数定义

```
template<typename T>
void Stack<T>::clear() {
    Node<T> *p = nullptr;
    while (m_top != nullptr) {
        p = m_top;
        m_top = m_top->m_next;
        delete p;
    }
}
```

说明

利用出栈的操作，逐个释放每个元素的内存空间

在析构函数中调用 clear 函数释放所有结点的内存：

Stack 析构函数定义

```
template<typename T>
Stack<T>::~Stack() {
    clear();
}
```

8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

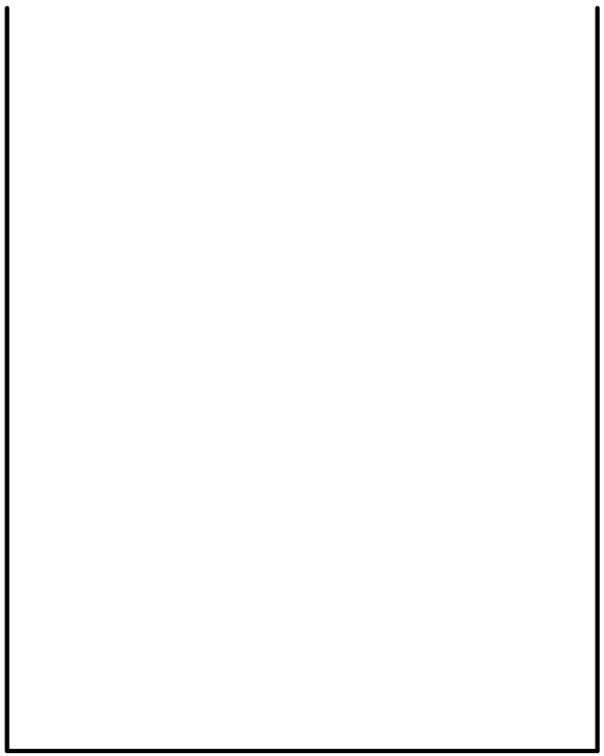
```
a - b / c + d * e =
```

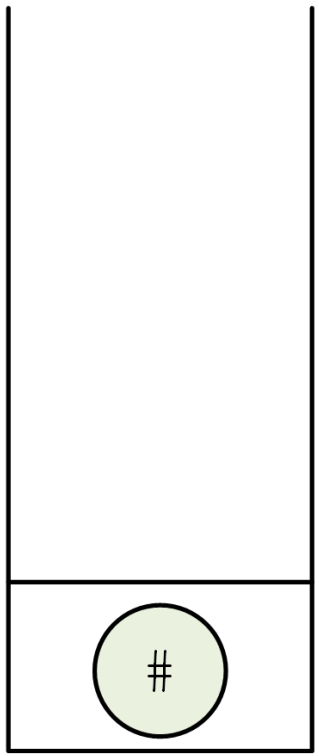
8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

```
a - b / c + d * e =
```



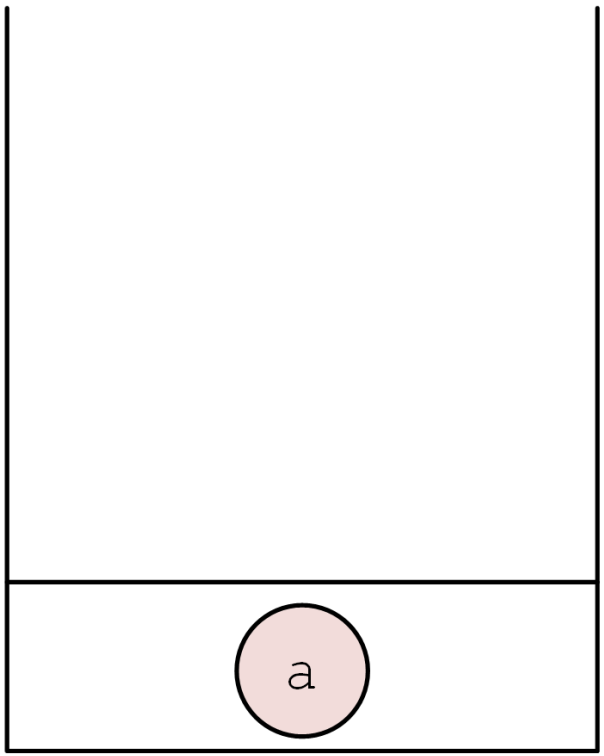


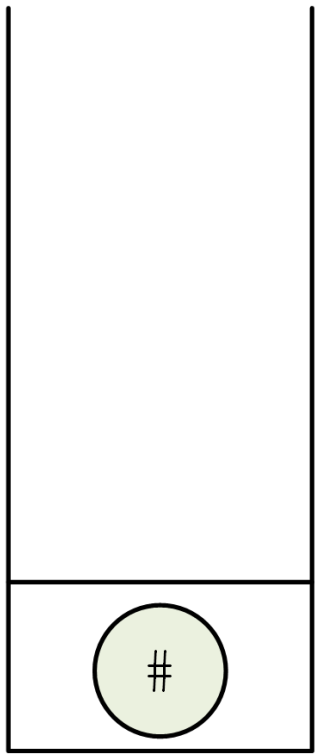
8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

a - b / c + d * e =



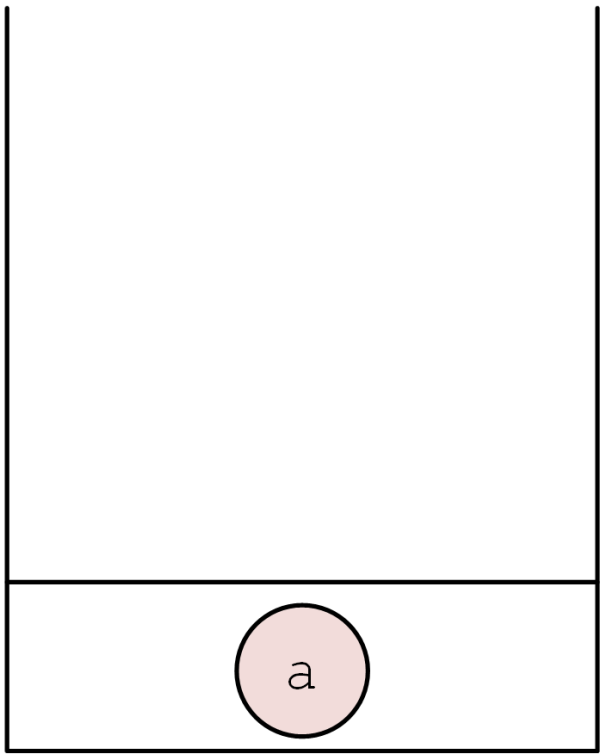


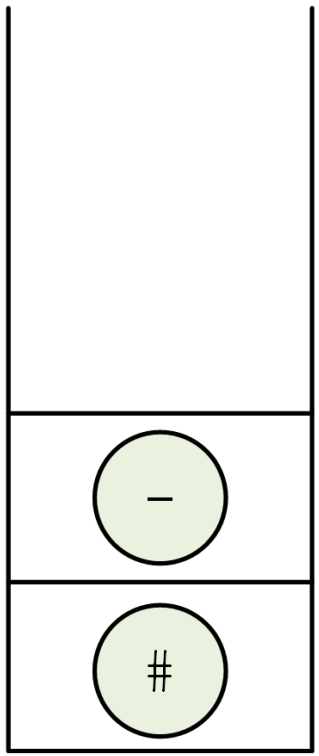
8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

```
a - b / c + d * e =
```



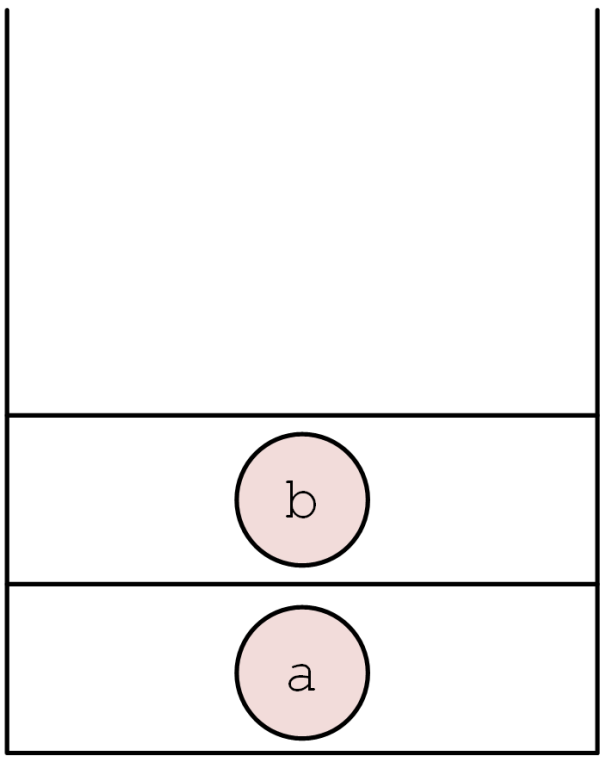


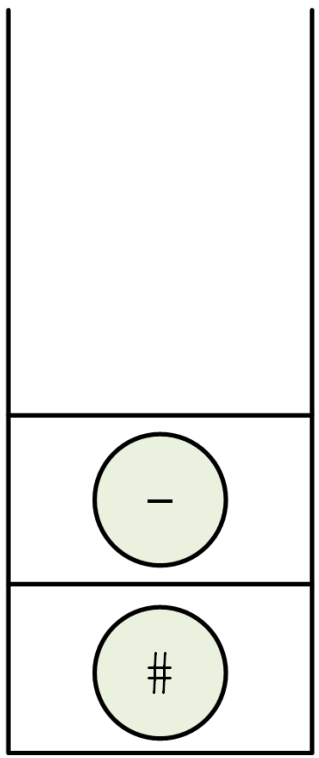
8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

```
a - b / c + d * e =
```



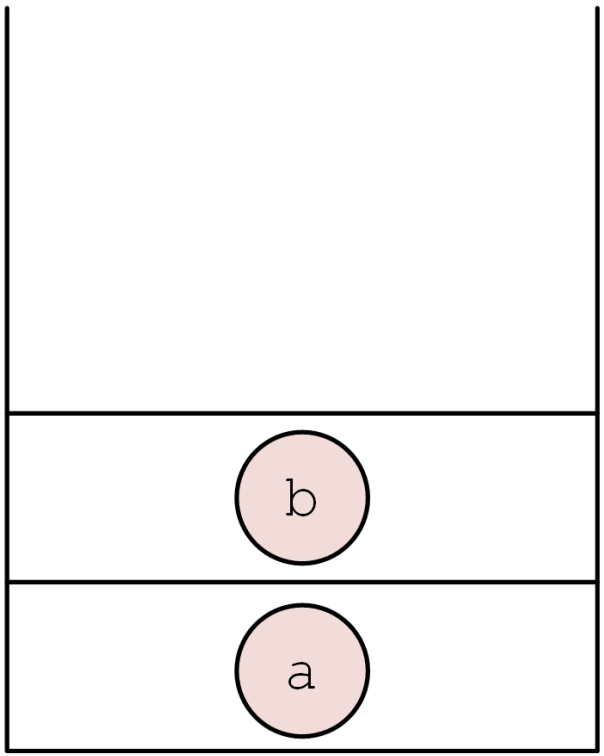


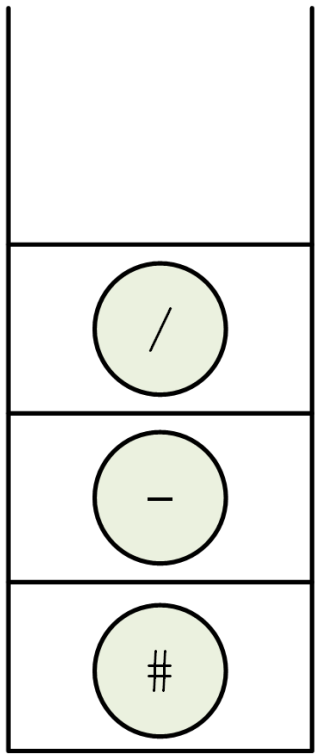
8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

```
a - b / c + d * e =
```



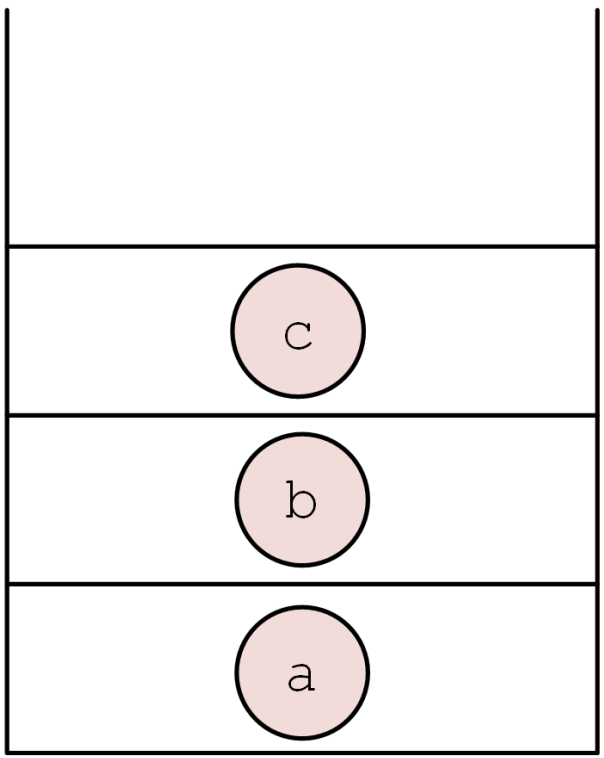


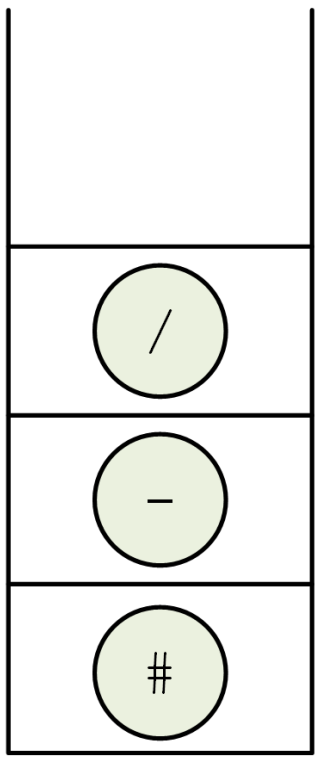
8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

```
a - b / c + d * e =
```



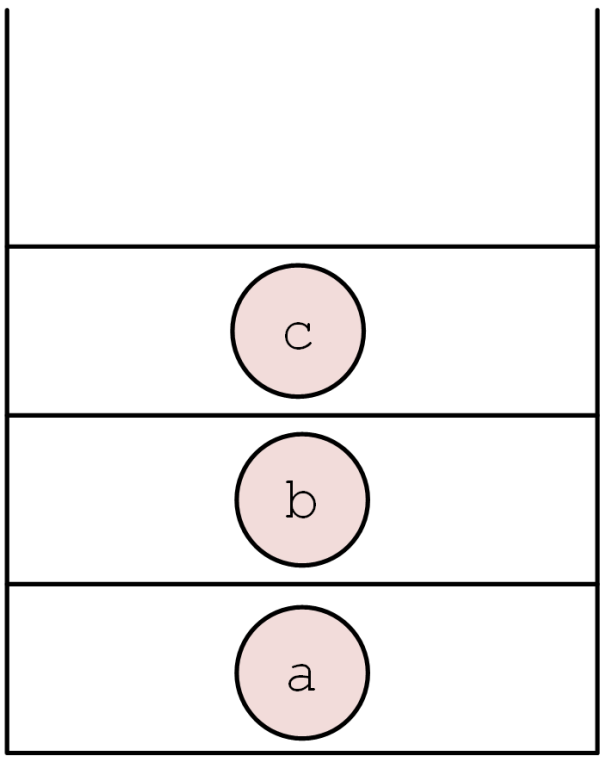


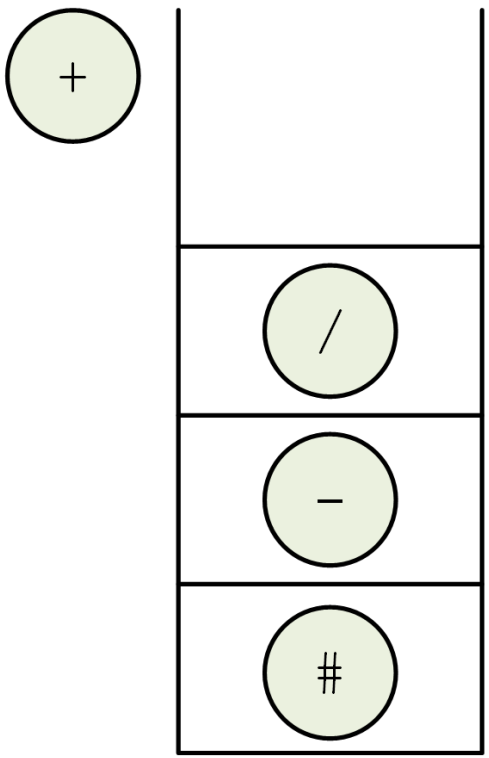
8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

```
a - b / c + d * e =
```



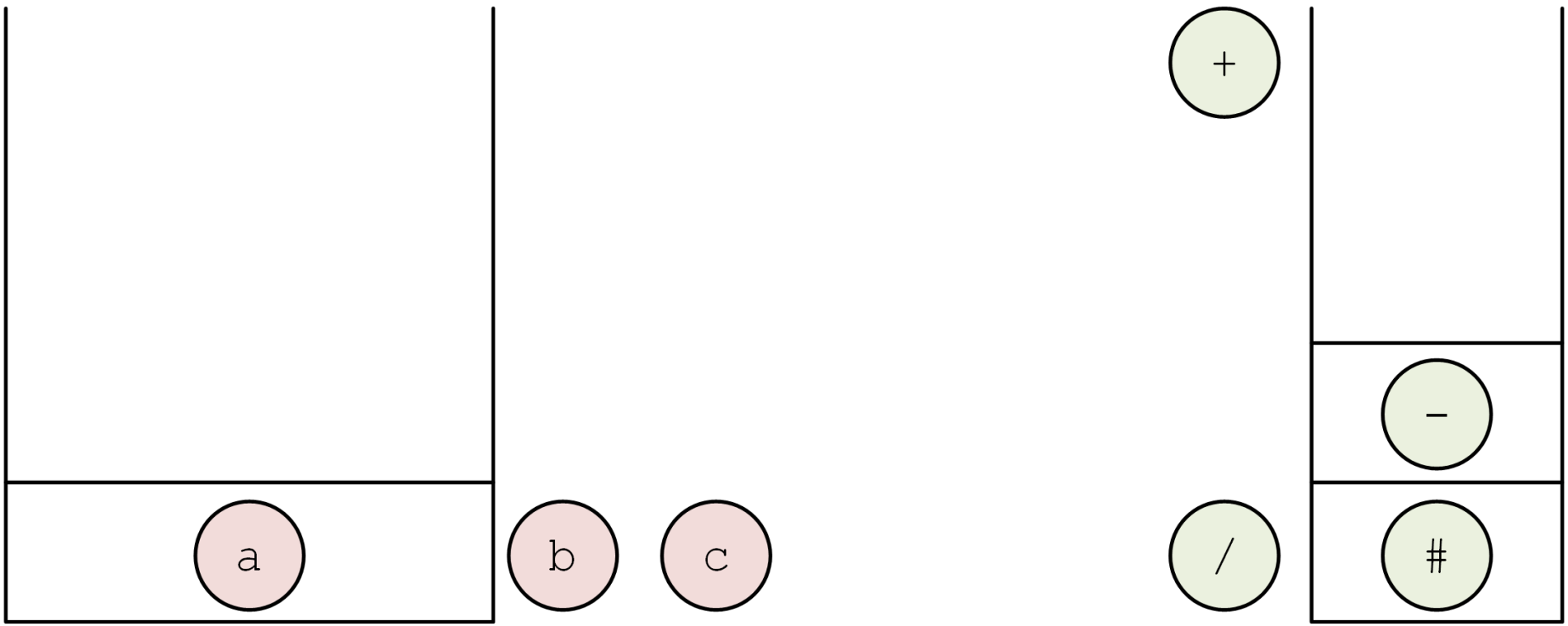


8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

a - b / c + d * e =

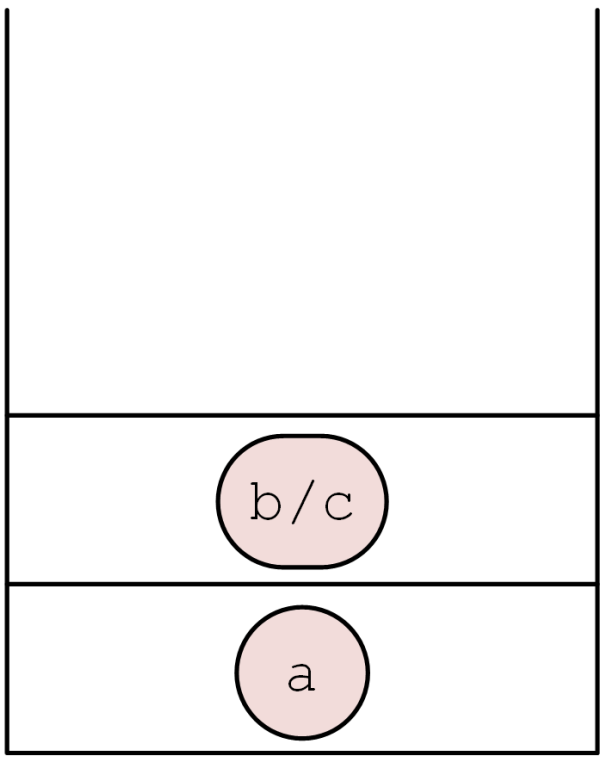


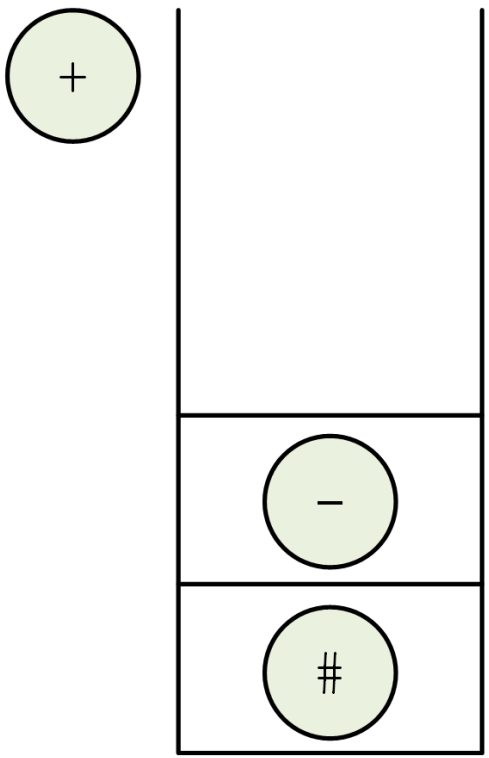
8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

```
a - b / c + d * e =
```



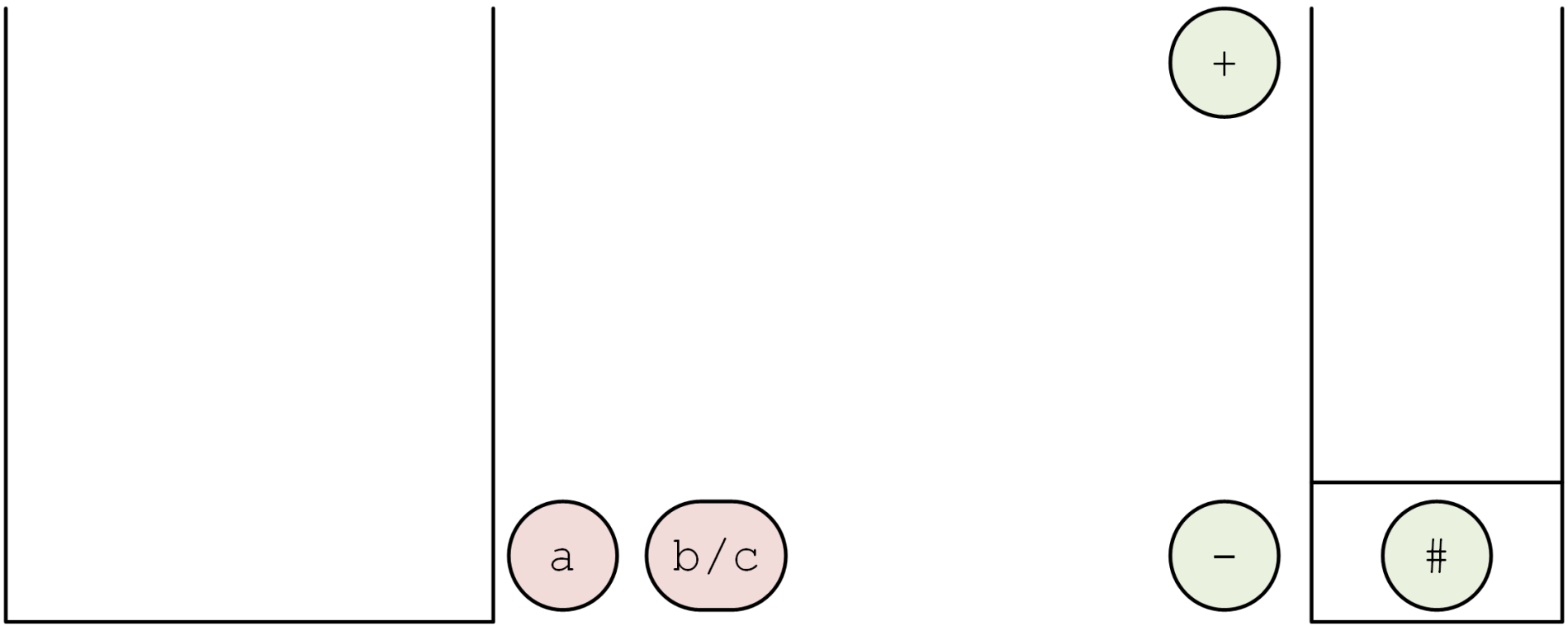


8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

a - b / c + d * e =



8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

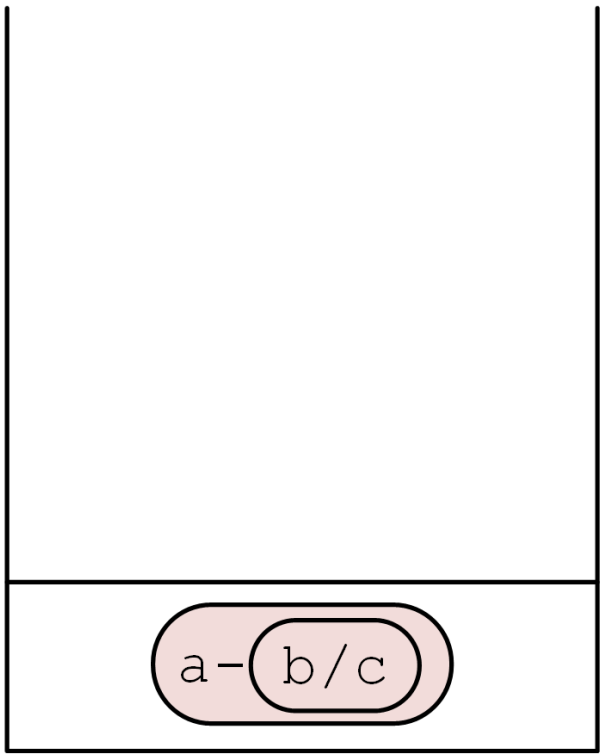
```
a - b / c + d * e =
```

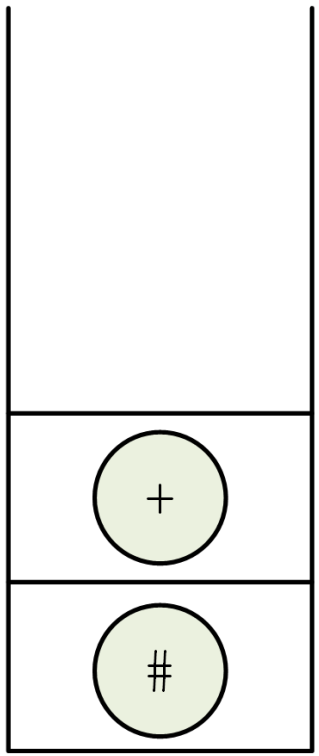
8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

```
a - b / c + d * e =
```



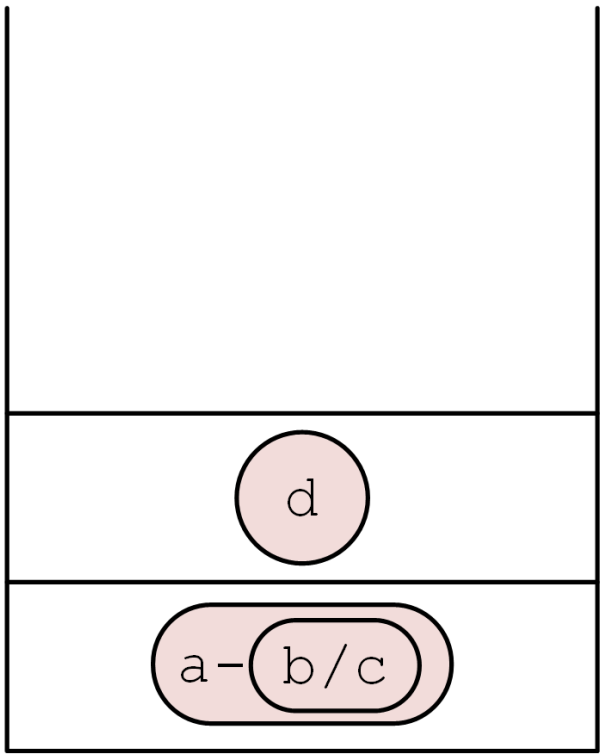


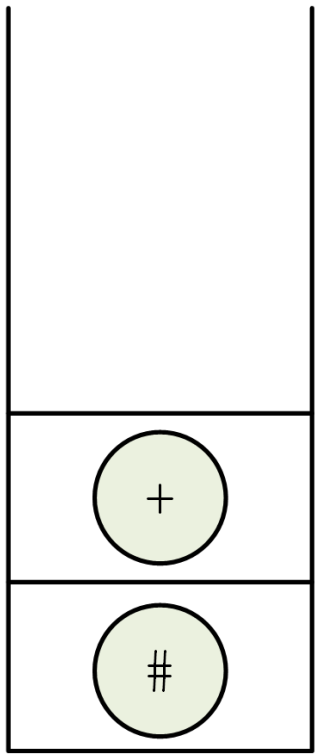
8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

a - b / c + d * e =



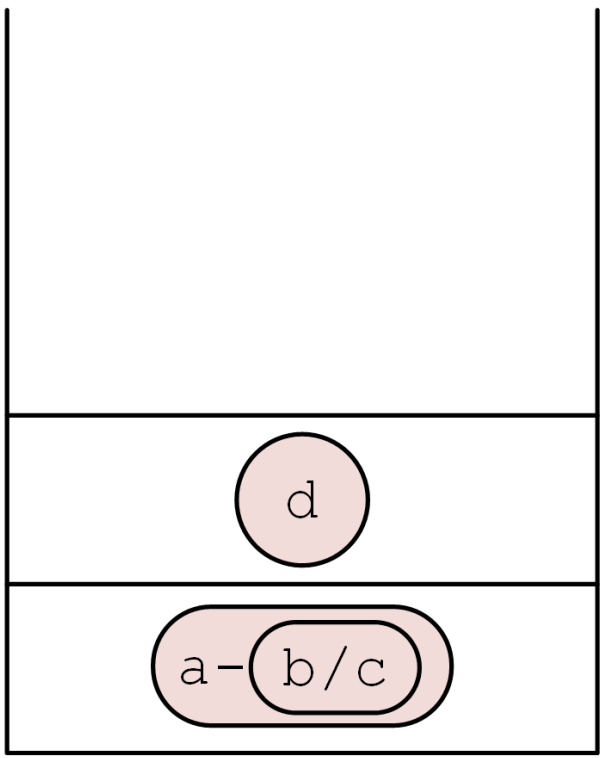


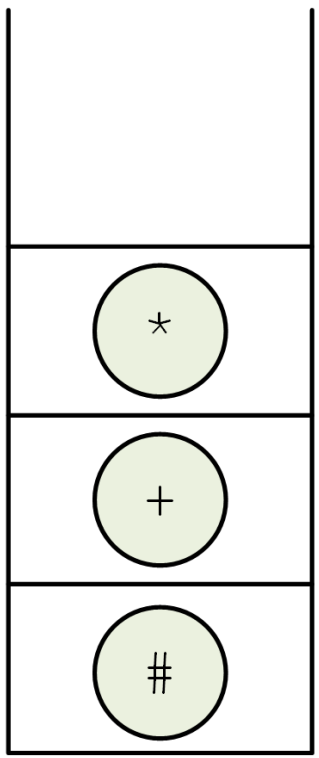
8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

```
a - b / c + d * e =
```



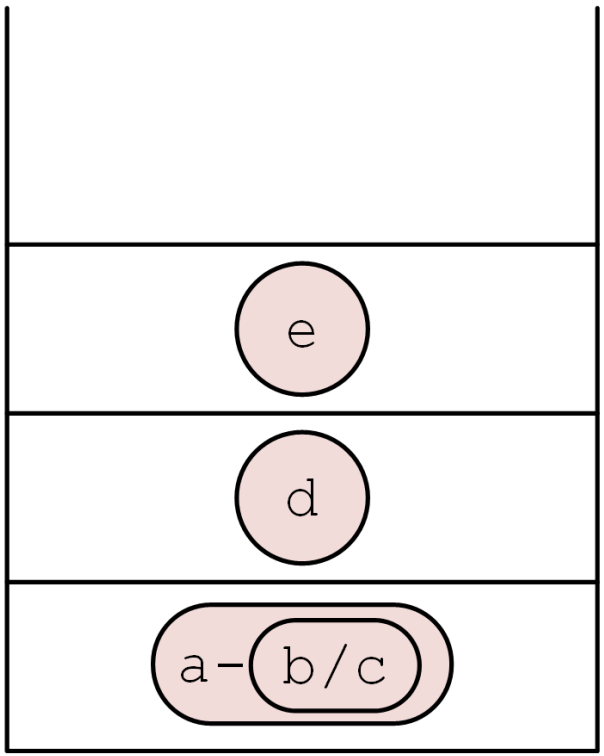


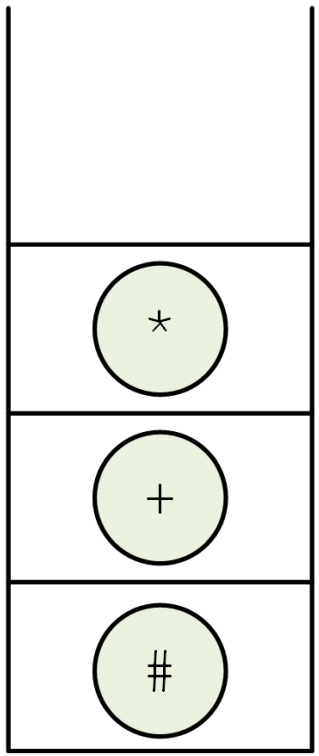
8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

```
a - b / c + d * e =
```



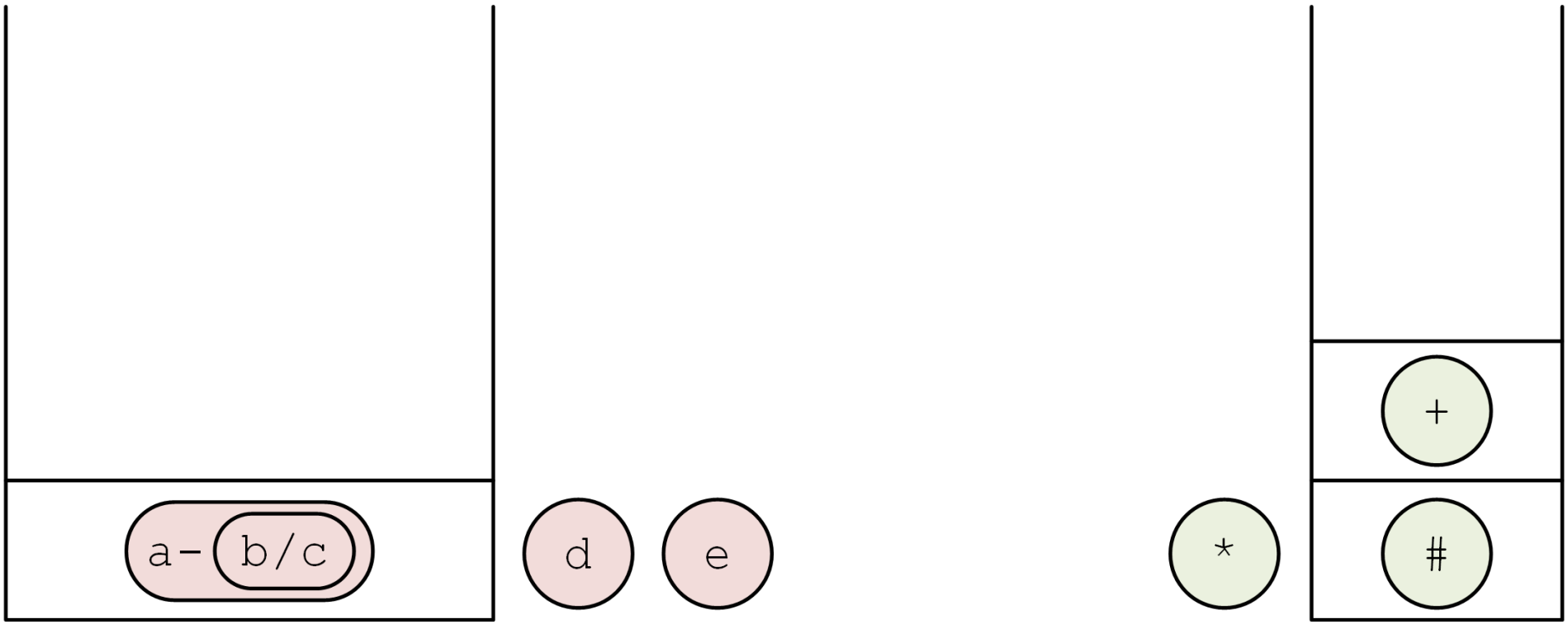


8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

a - b / c + d * e =

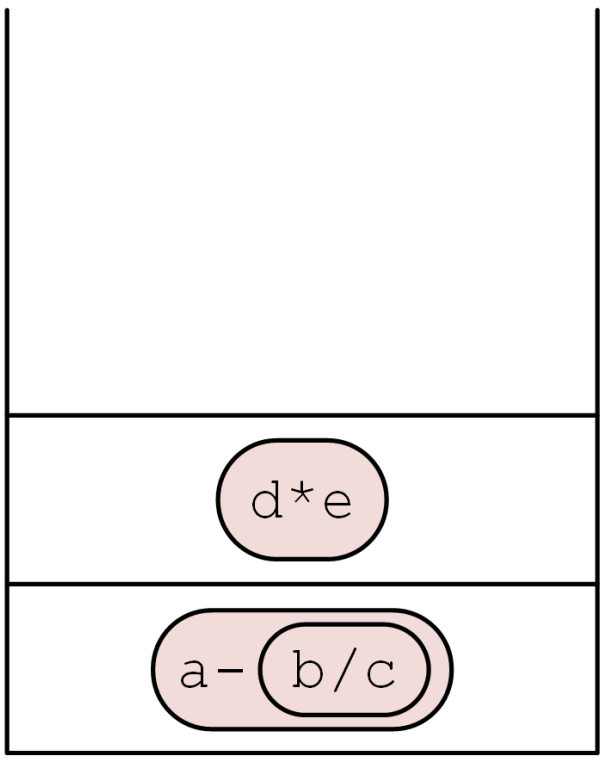


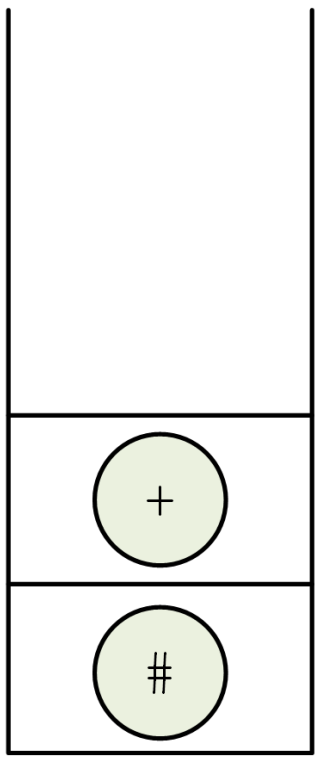
8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

```
a - b / c + d * e =
```



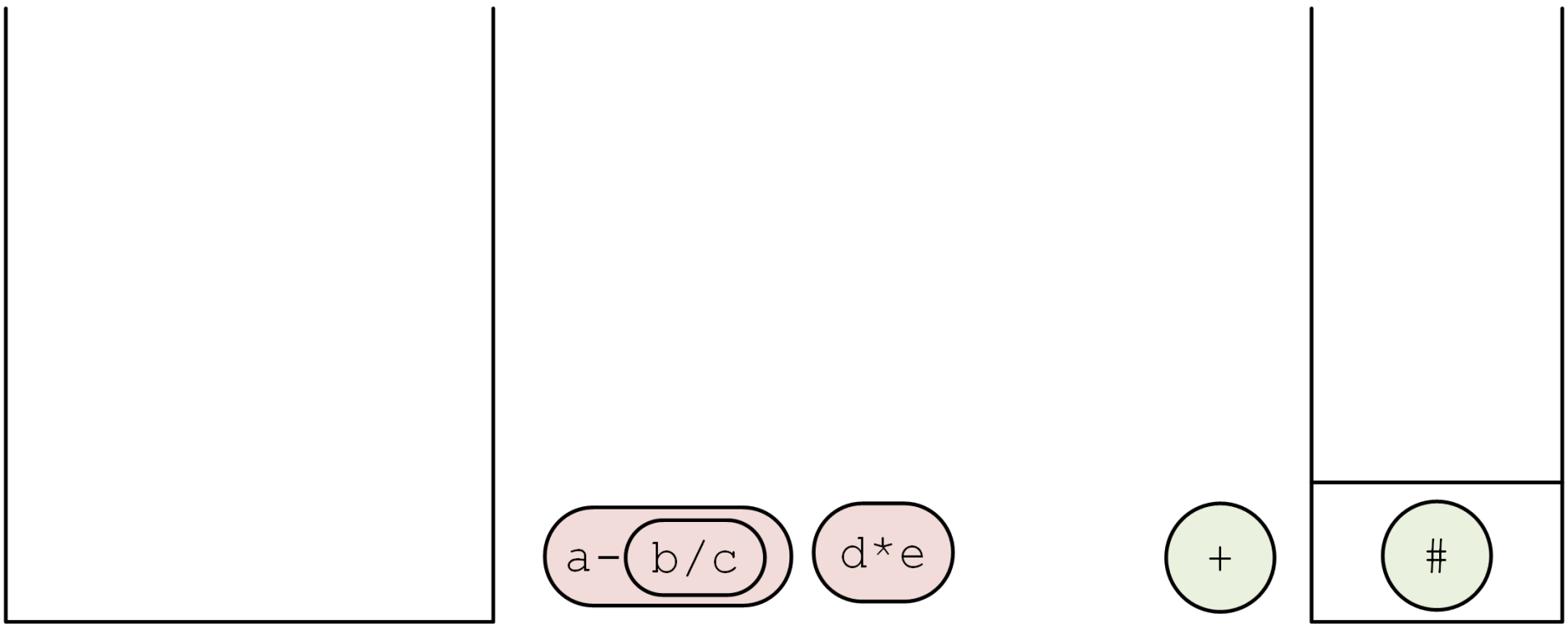


8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

a - b / c + d * e =

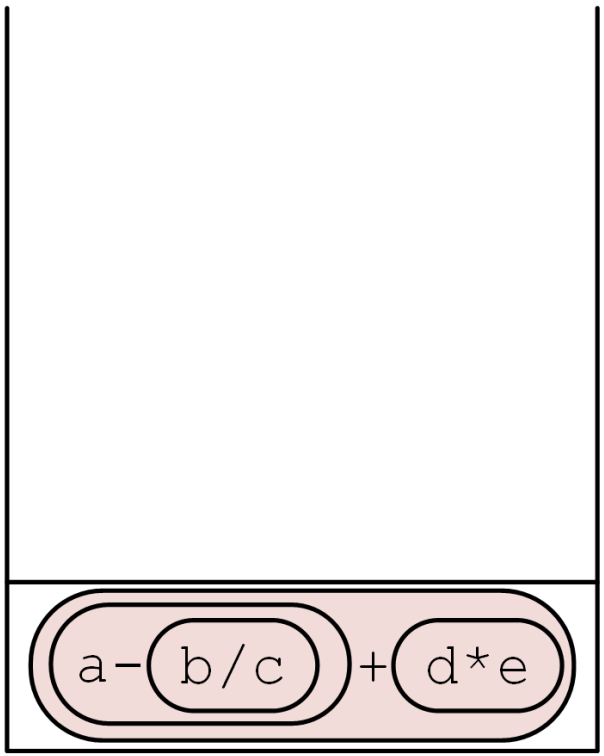


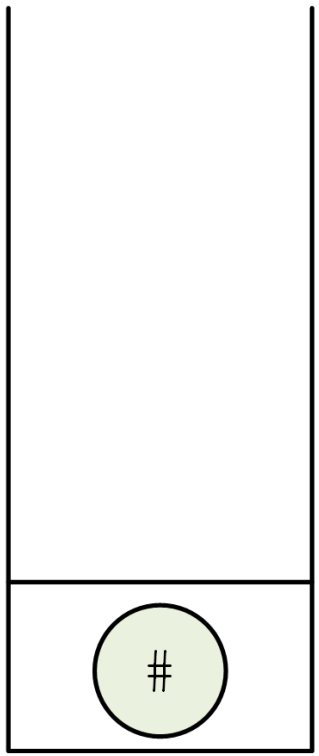
8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

a - b / c + d * e =



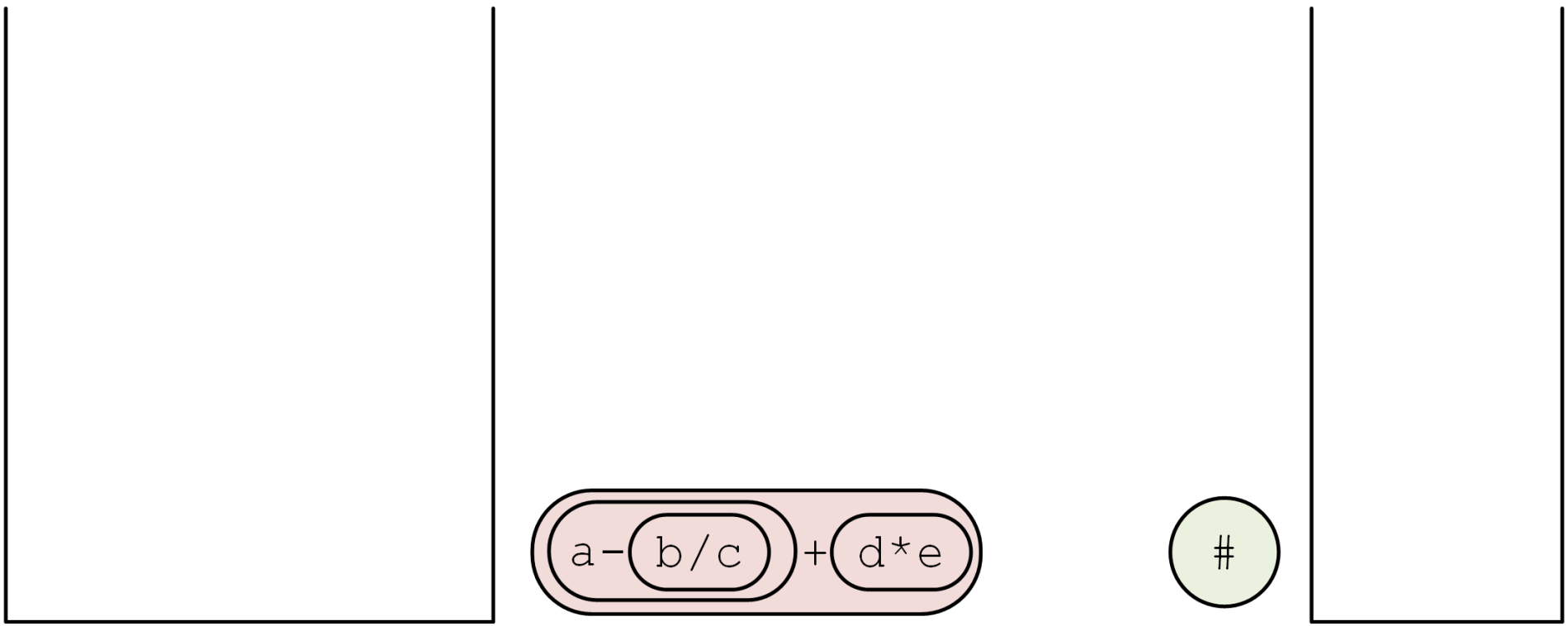


8.4.2 简单计算器

表达式求值是栈的重要应用之一

假设有如下算术表达式：

a - b / c + d * e =



8.4.2 简单计算器

简单四则运算的类代码清单如下：

Calculator 类定义部分一

```
1  class Calculator {
2  private:
3      Stack<double> m_num;                          //操作数栈
4      Stack<char> m_opr;                            //运算符栈
5      int precedence(const char &s ) const;  //获取运算符优先级
6      double readNum(string::const_iterator &it); //读取操作数
7      void calculate();                             //取出运算符和操作数进行计算
8                                                    //内联函数，判断是否为数字
9      bool isNum(string::const_iterator &c) const {
10         return* c >= '0' && * c <= '9'  || * c == '.';
11     }
12 public:
13     Calculator(){ m_opr.push('#'); }       //运算符栈初始化
14     double doIt(const string &exp);        //表达式求值
15 };
```

8.4.2 简单计算器

简单四则运算的类代码清单如下：

Calculator 类定义部分二

```
17  int Calculator::precedence(const char & s) const{
18      switch (s) {
19          case '=' : return 0;
20          case '#' : return 1;
21          case '+' : case '-' : return 2;
22          case '*' : case '/' : return 3;
23      }
24  }
25  double Calculator::readNum(string::const_iterator &it){
26      string t;
27      while (isNum(it))
28      t += *it++;                 //继续扫描，直到遇到运算符
29      return stod(t);             //将数字字符串转换为double类型（C++11新特性）
30  }
```

8.4.2 简单计算器

简单四则运算的类代码清单如下：

Calculator 类定义部分三

```
31  void Calculator::calculate(){
32      double b = m_num.top();          //取出右操作数
33      m_num.pop();                     //右操作数出栈
34      double a = m_num.top();          //取出左操作数
35      m_num.pop();                     //左操作数出栈
36      if (m_opr.top() == '+')
37          m_num.push(a + b);           //将计算结果压栈，下面三个运算与此操作相同
38      else if (m_opr.top() == '-')
39          m_num.push(a - b);
40      else if (m_opr.top() == '*')
41          m_num.push(a*b);
42      else if (m_opr.top() == '/')
43          m_num.push(a / b);
44      m_opr.pop();                     //当前运算结束，运算符出栈
45  }
```

8.4.2 简单计算器

简单四则运算的类代码清单如下：

Calculator 类定义部分四

```
46  double Calculator::doIt(const string & exp){
47      m_num.clear();                  //保证同一个对象再次调用doIt时数据栈为空
48      for (auto it = exp.begin(); it != exp.end();) {
49          if (isNum(it))              //遇到操作数
50              m_num.push(readNum(it));//操作数入栈
51          else{                       //遇到运算符，下面while循环条件中不能忽略优先级相
              同的情况
52              while (precedence( * it) <= precedence(m_opr.top())){
53                  if (m_opr.top() == ' #' )
54                      break;          //如果运算符栈只剩下#，则计算完毕
55                  calculate();        //执行栈顶运算符计算
56              }
57              if ( * it != ' =' )
58                  m_opr.push( * it); //运算符入栈
59              ++it;                   //继续扫描
60          }
61      }
62      return m_num.top();             //返回计算结果，注意数据栈此时非空
63  }
```

8.4.2 简单计算器

测试 Calculator：

使用 Calculator 类对象

```
string exp;
Calculator cal;
while (getline(cin, exp) ) //获取一行表达式
    cout << exp << cal.doIt(exp) << endl;
```

输入 9-4/2+2.5*2=

输出结果为：

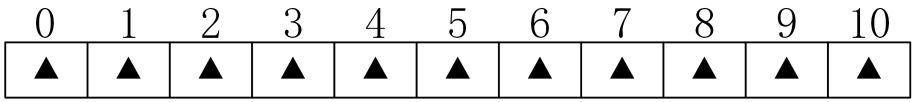
9-4/2+2.5*2=12

8.5 二叉树

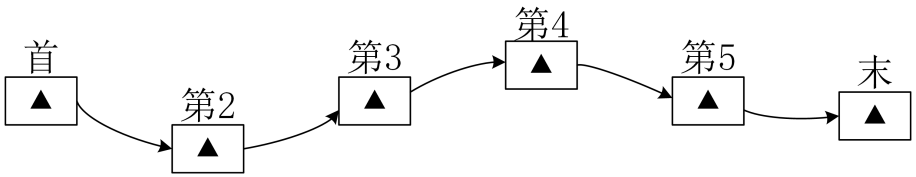
线性结构

每个结点只有一个后继

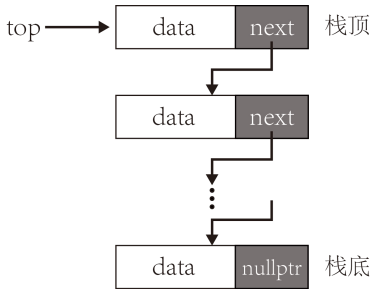
数组：



线性链表：



链栈：

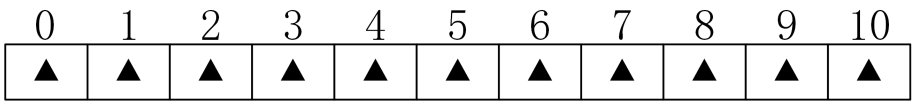


8.5 二叉树

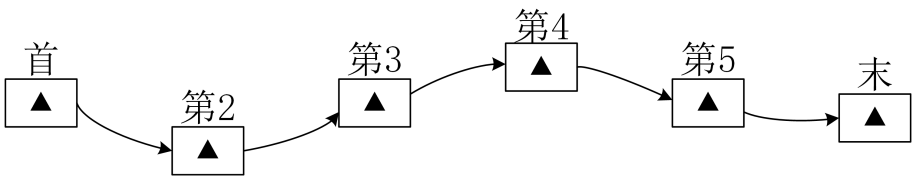
线性结构

每个结点只有一个后继

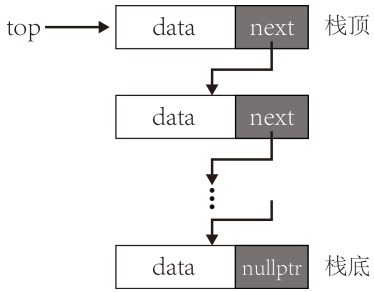
数组：



线性链表：



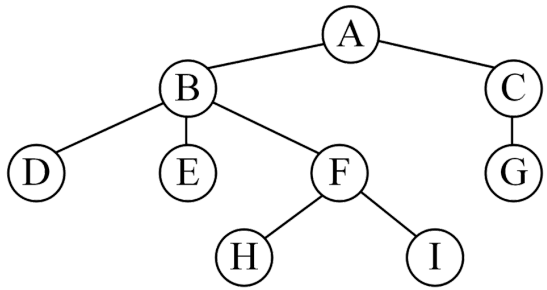
链栈：



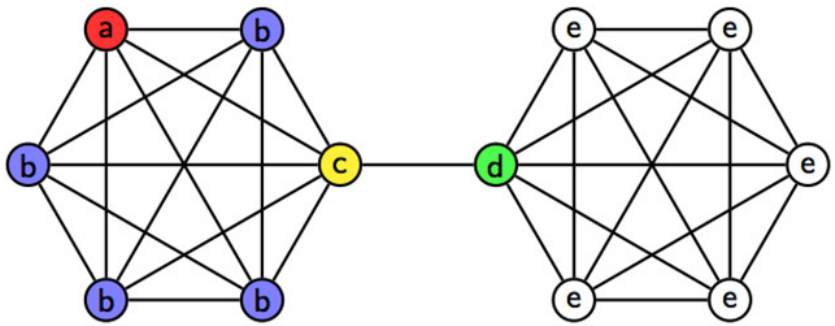
非线性结构

一个结点可能有多个后继多个前驱

树：

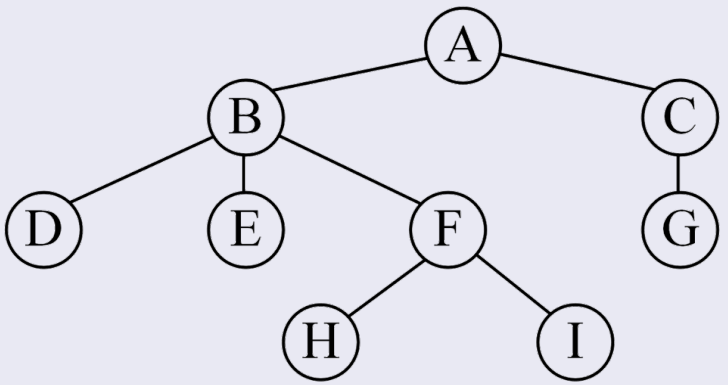


图：



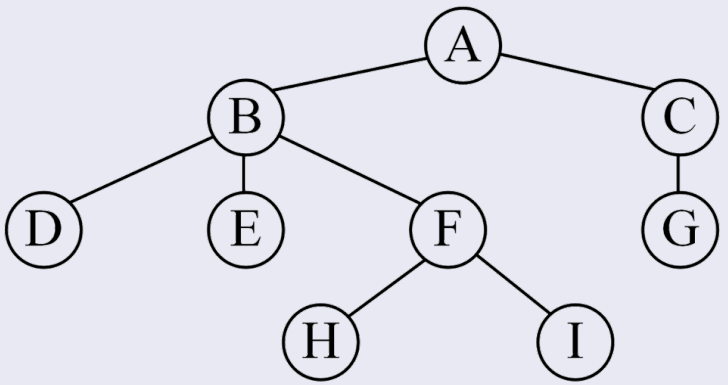
8.5 二叉树

树



8.5 二叉树

树

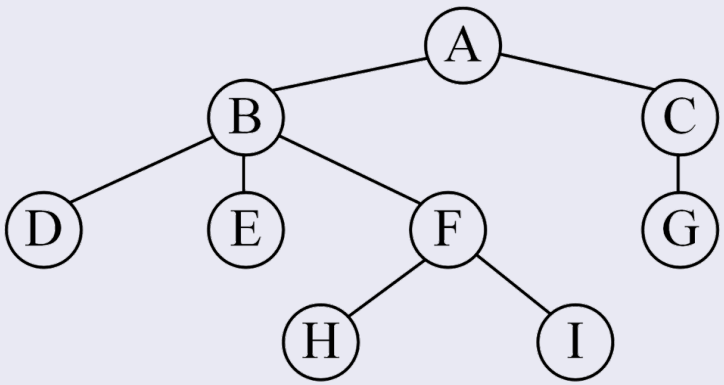


根结点

一棵非空树有且仅有一个根结点

8.5 二叉树

树



根结点

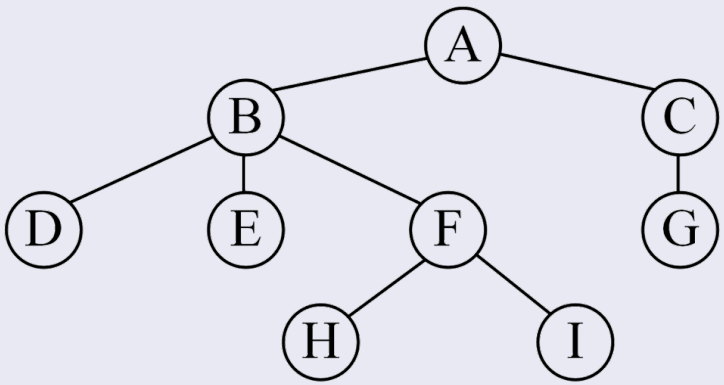
一棵非空树有且仅有一个根结点

子树

除了根结点外，每个集合互不相交的结点集称为根的子树

8.5 二叉树

树



根结点

一棵非空树有且仅有一个根结点

子树

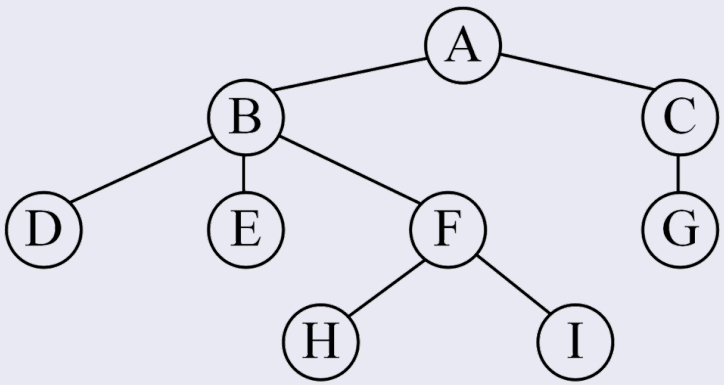
除了根结点外，每个集合互不相交的结点集称为根的子树

度

每个结点的子树的数量为该结点的度

8.5 二叉树

树



根结点

一棵非空树有且仅有一个根结点

子树

除了根结点外，每个集合互不相交的结点集称为根的子树

度

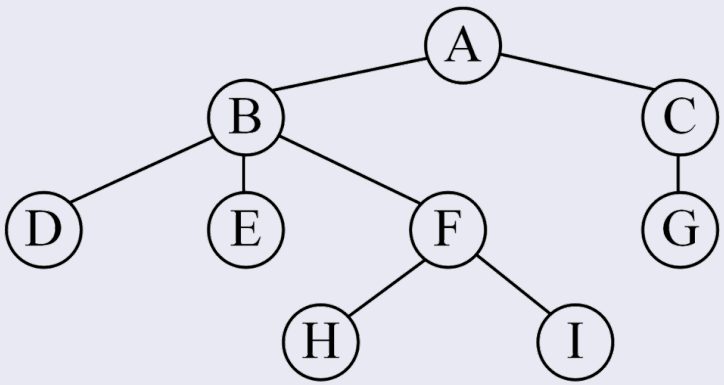
每个结点的子树的数量为该结点的度

叶子结点

度为 0 的结点称为叶子结点

8.5 二叉树

树



根结点

一棵非空树有且仅有一个根结点

子树

除了根结点外，每个集合互不相交的结点集称为根的子树

度

每个结点的子树的数量为该结点的度

叶子结点

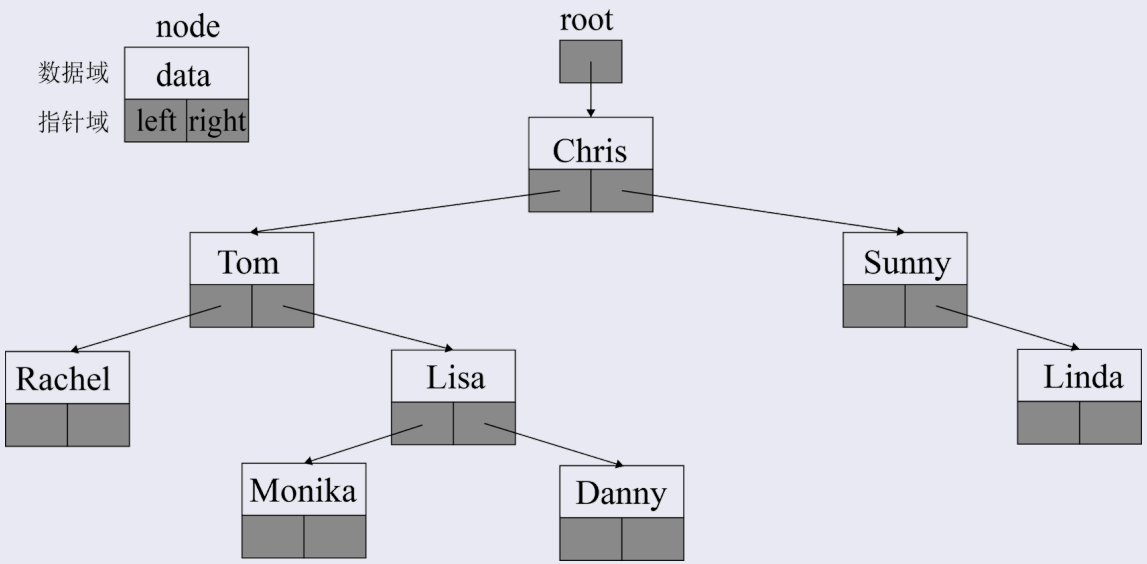
度为 0 的结点称为叶子结点

子结点

每个结点的子树的根结点称为该结点的子结点

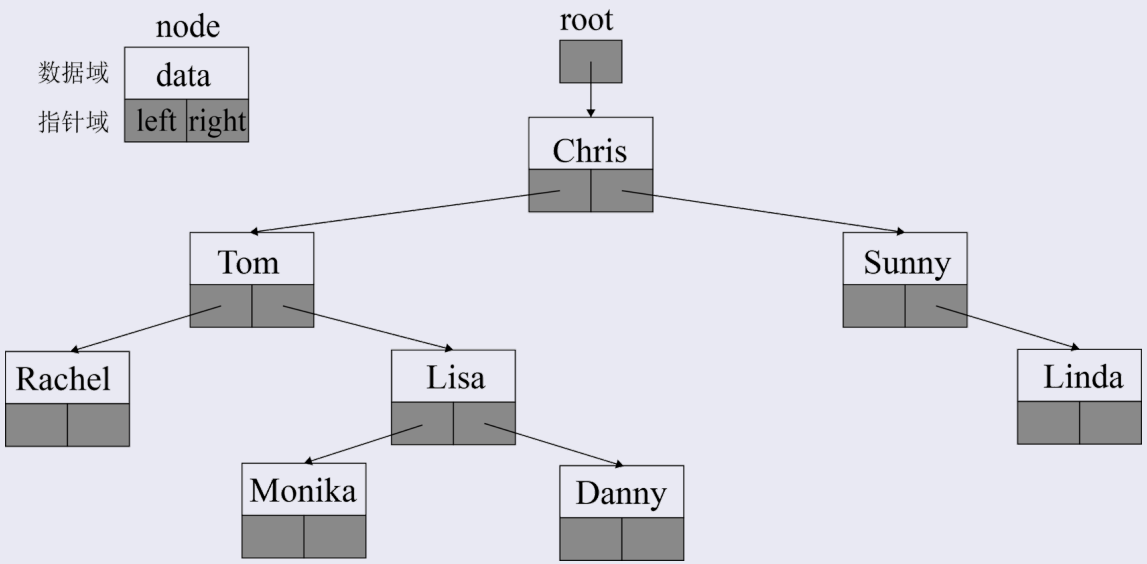
8.5.1 二叉树的概念和表示

二叉树



8.5.1 二叉树的概念和表示

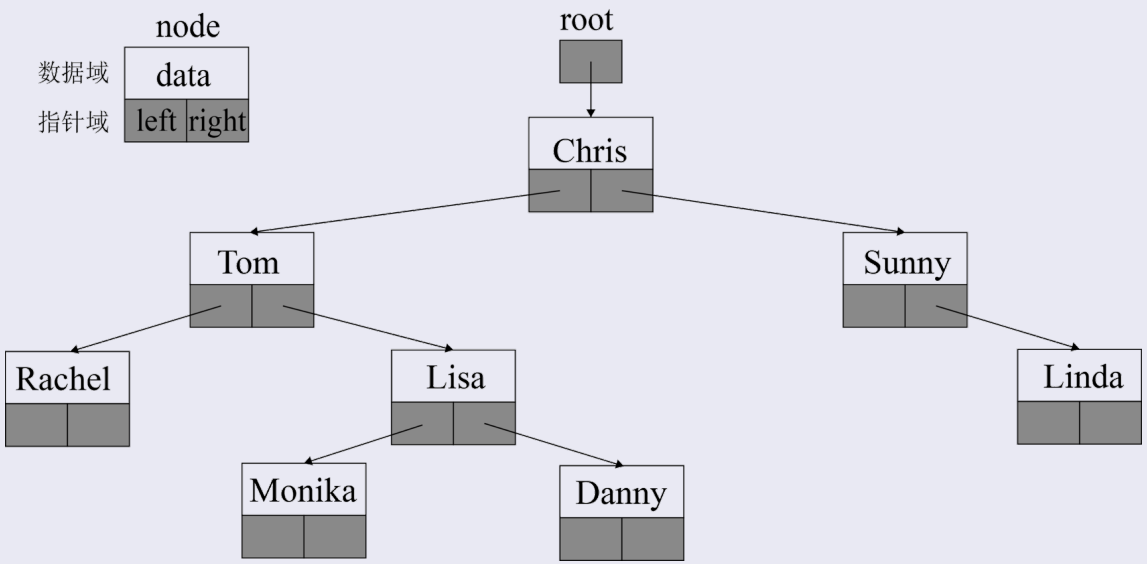
二叉树



每个结点的子结点数目不超过 2

8.5.1 二叉树的概念和表示

二叉树



每个结点的子结点数目不超过 2

每个结点的两个子树也称为该结点的左子树和右子树

8.5.1 二叉树的概念和表示

二叉树结点的定义如下：

二叉树结点类模板 Node 定义

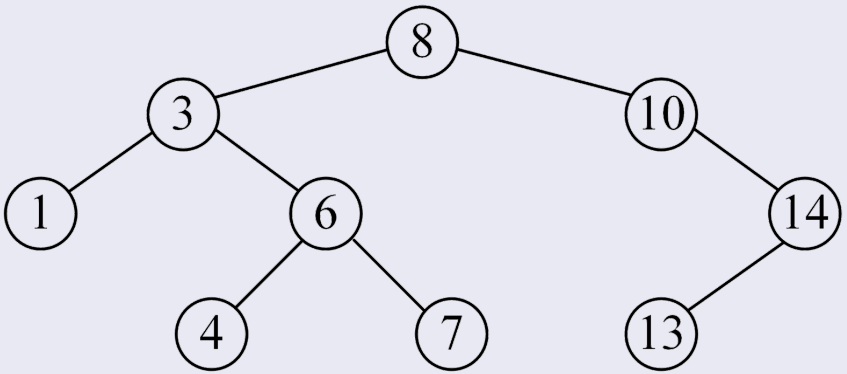
```
template<typename T>
class Node {
private:
    T m_data;
    Node *m_left = nullptr, *m_right = nullptr;
public:
    Node(const T &data):m_data(data){}
    T& data() { return m_data; }
    const T& data() const{ return m_data; }
    Node* left() { return m_left; }
    Node* right() { return m_right; }
};
```

说明

- 数据成员 m_data 表示一个结点的数据域
- 成员指针 m_left 和 m_right 分别为结点的左子树和右子树的根结点的指针

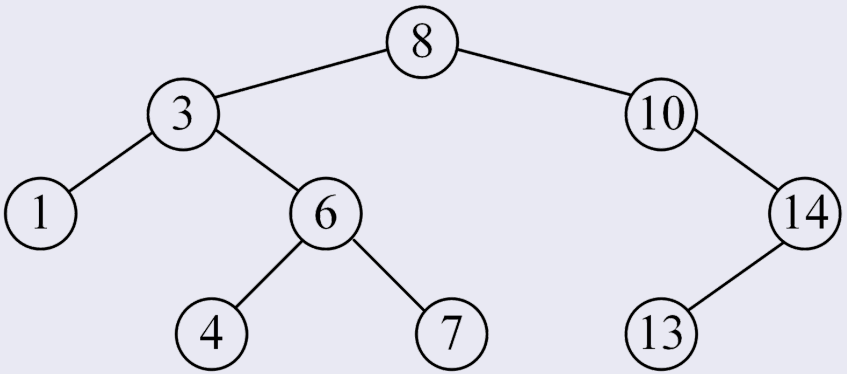
8.5.1 二叉树的概念和表示

二叉搜索树



8.5.1 二叉树的概念和表示

二叉搜索树



任意一个结点的左子树中的数据值都小于该结点的数据值，右子树的数据值都大于或等于该结点的数据值

8.5.1 二叉树的概念和表示

定义一个二叉搜索树类，包含插入结点、遍历、查找、销毁子树等操作：

BinaryTree 类模板定义

```
template<typename T>
class BinaryTree{
public:
    ~BinaryTree() { destroy(m_root); }
    Node<T>* root() const { return m_root; }
    Node<T>* insert(const T &value){
        return insert_(m_root, value);//返回新建结点指针
    }
    Node<T>* search(const T &value)const{
        return search_(m_root, value);
    }
    void inOrder(Node<T> *p,void (*visit)(Node<T>&));
private:
    Node<T>* search_(Node<T> *p, const T &value) const;
    Node<T>* insert_(Node<T> * &p, const T &value);
    void destroy(Node<T> *p);
private:
    Node<T> *m_root = nullptr;
};
```

说明

- insert 和析构函数都分别调用私有成员函数 insert_ 和 destory，分别递归进行插入和销毁子树操作
- inorder 函数执行中序遍历操作，其第二个参数为遍历时对元素进行操作的函数
- search 函数调用 search_，从根结点开始进行二分搜索

8.5.2 创建二叉搜索树

通过逐个插入元素来创建二叉搜索树：

创建二叉搜索树

```
int keys[] = { 8,3,10,1,6,14,4,7,13 };
BinaryTree<int> bstree;
    for (auto i:keys)
        if (Node<int> *n=bstree.insert(i) )
            cout << n->data() << "␣";
```

说明

插入新结点成功则返回结点指针，然后打印数据

8.5.2 创建二叉搜索树

通过逐个插入元素来创建二叉搜索树：

创建二叉搜索树

```
int keys[] = { 8,3,10,1,6,14,4,7,13 };
BinaryTree<int> bstree;
    for (auto i:keys)
        if (Node<int> *n=bstree.insert(i) )
            cout << n->data() << "␣";
```

8,3,10,1,6,14,4,7,13

说明

插入新结点成功则返回结点指针，然后打印数据

说明

从根结点开始，若待插入结点小于根结点，则在左子树上继续查找插入位置；否则在右子树中查找

8.5.2 创建二叉搜索树

通过逐个插入元素来创建二叉搜索树：

创建二叉搜索树

```
int keys[] = { 8,3,10,1,6,14,4,7,13 };
BinaryTree<int> bstree;
    for (auto i:keys)
        if (Node<int> *n=bstree.insert(i) )
            cout << n->data() << "␣";
```

8,3,10,1,6,14,4,7,13

说明

插入新结点成功则返回结点指针，然后打印数据

说明

从根结点开始，若待插入结点小于根结点，则在左子树上继续查找插入位置；否则在右子树中查找

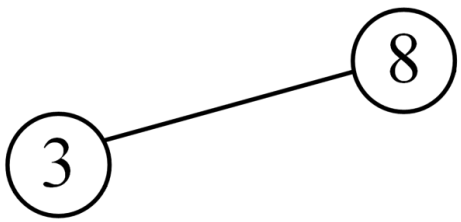
8.5.2 创建二叉搜索树

通过逐个插入元素来创建二叉搜索树：

创建二叉搜索树

```
int keys[] = { 8,3,10,1,6,14,4,7,13 };
BinaryTree<int> bstree;
    for (auto i:keys)
        if (Node<int> *n=bstree.insert(i) )
            cout << n->data() << "␣";
```

8,3,10,1,6,14,4,7,13



说明

插入新结点成功则返回结点指针，然后打印数据

说明

从根结点开始，若待插入结点小于根结点，则在左子树上继续查找插入位置；否则在右子树中查找

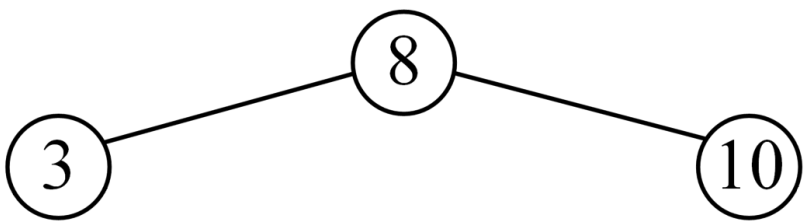
8.5.2 创建二叉搜索树

通过逐个插入元素来创建二叉搜索树：

创建二叉搜索树

```
int keys[] = { 8,3,10,1,6,14,4,7,13 };
BinaryTree<int> bstree;
    for (auto i:keys)
        if (Node<int> *n=bstree.insert(i) )
            cout << n->data() << "␣";
```

8,3,10,1,6,14,4,7,13



说明

插入新结点成功则返回结点指针，然后打印数据

说明

从根结点开始，若待插入结点小于根结点，则在左子树上继续查找插入位置；否则在右子树中查找

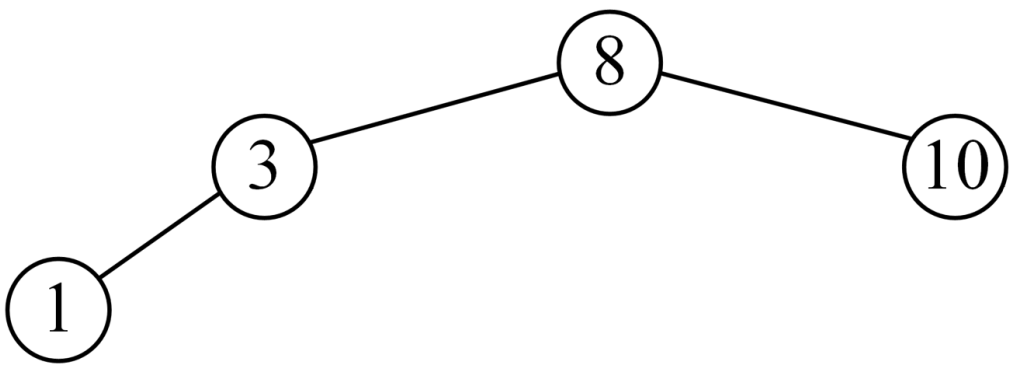
8.5.2 创建二叉搜索树

通过逐个插入元素来创建二叉搜索树：

创建二叉搜索树

```
int keys[] = { 8,3,10,1,6,14,4,7,13 };
BinaryTree<int> bstree;
    for (auto i:keys)
        if (Node<int> *n=bstree.insert(i) )
            cout << n->data() << "␣";
```

8,3,10,1,6,14,4,7,13



说明

插入新结点成功则返回结点指针，然后打印数据

说明

从根结点开始，若待插入结点小于根结点，则在左子树上继续查找插入位置；否则在右子树中查找

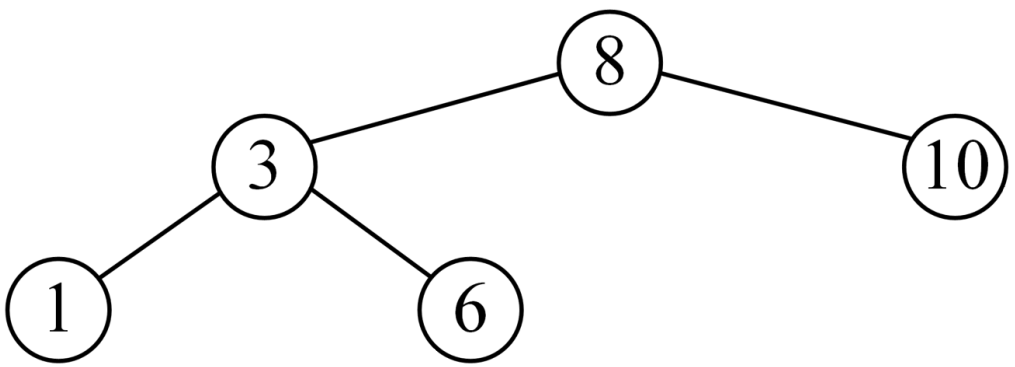
8.5.2 创建二叉搜索树

通过逐个插入元素来创建二叉搜索树：

创建二叉搜索树

```
int keys[] = { 8,3,10,1,6,14,4,7,13 };
BinaryTree<int> bstree;
    for (auto i:keys)
        if (Node<int> *n=bstree.insert(i) )
            cout << n->data() << "␣";
```

8,3,10,1,6,14,4,7,13



说明

插入新结点成功则返回结点指针，然后打印数据

说明

从根结点开始，若待插入结点小于根结点，则在左子树上继续查找插入位置；否则在右子树中查找

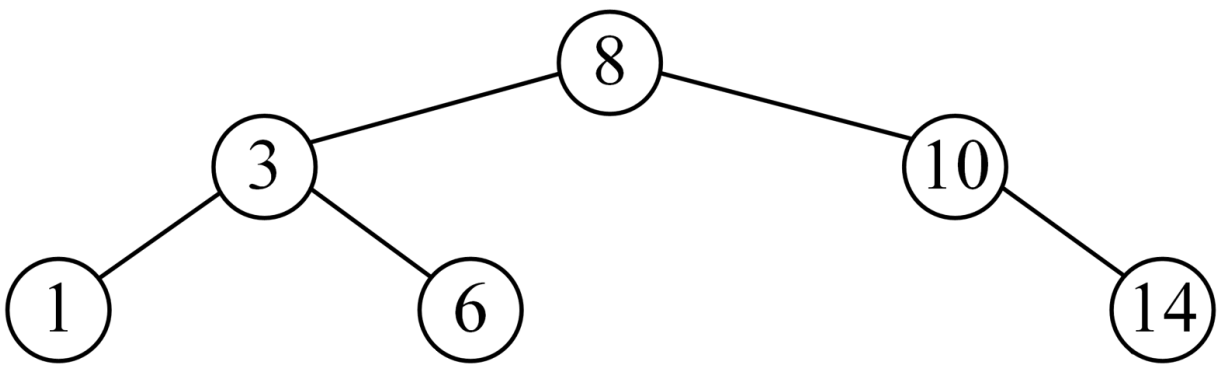
8.5.2 创建二叉搜索树

通过逐个插入元素来创建二叉搜索树：

创建二叉搜索树

```
int keys[] = { 8,3,10,1,6,14,4,7,13 };
BinaryTree<int> bstree;
    for (auto i:keys)
        if (Node<int> *n=bstree.insert(i) )
            cout << n->data() << "␣";
```

8,3,10,1,6,14,4,7,13



说明

插入新结点成功则返回结点指针，然后打印数据

说明

从根结点开始，若待插入结点小于根结点，则在左子树上继续查找插入位置；否则在右子树中查找

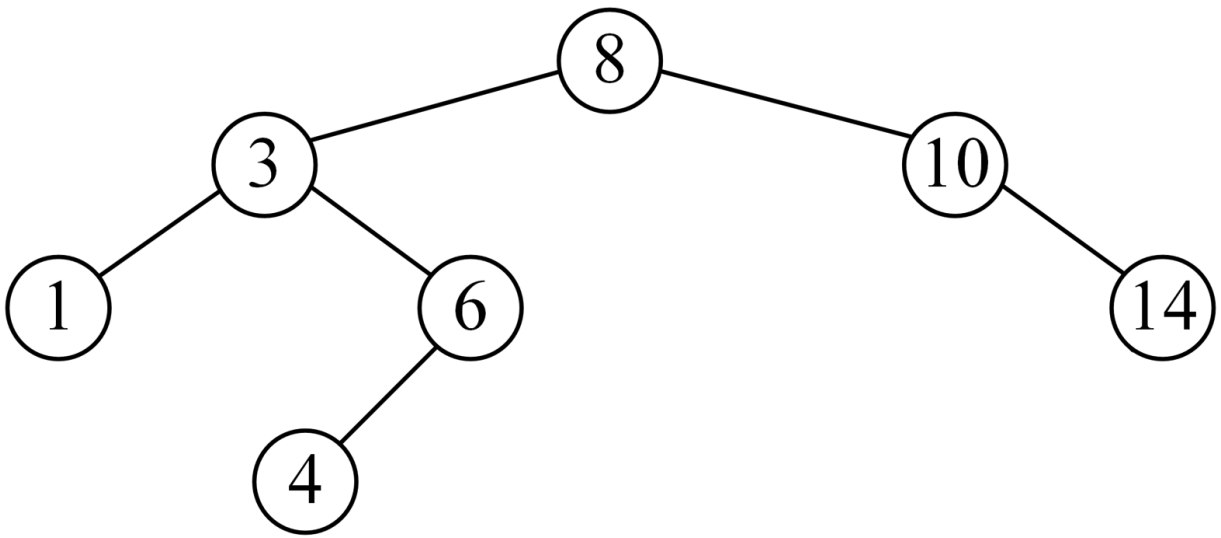
8.5.2 创建二叉搜索树

通过逐个插入元素来创建二叉搜索树：

创建二叉搜索树

```
int keys[] = { 8,3,10,1,6,14,4,7,13 };
BinaryTree<int> bstree;
    for (auto i:keys)
        if (Node<int> *n=bstree.insert(i) )
            cout << n->data() << "␣";
```

8,3,10,1,6,14,4,7,13



说明

插入新结点成功则返回结点指针，然后打印数据

说明

从根结点开始，若待插入结点小于根结点，则在左子树上继续查找插入位置；否则在右子树中查找

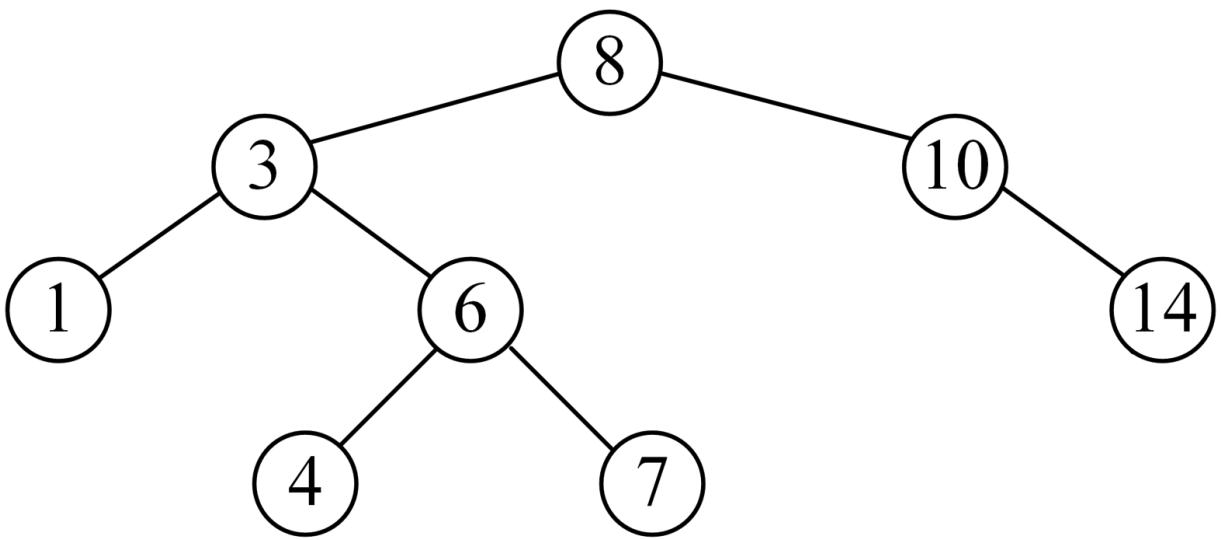
8.5.2 创建二叉搜索树

通过逐个插入元素来创建二叉搜索树：

创建二叉搜索树

```
int keys[] = { 8,3,10,1,6,14,4,7,13 };
BinaryTree<int> bstree;
    for (auto i:keys)
        if (Node<int> *n=bstree.insert(i) )
            cout << n->data() << "␣";
```

8,3,10,1,6,14,4,7,13



说明

插入新结点成功则返回结点指针，然后打印数据

说明

从根结点开始，若待插入结点小于根结点，则在左子树上继续查找插入位置；否则在右子树中查找

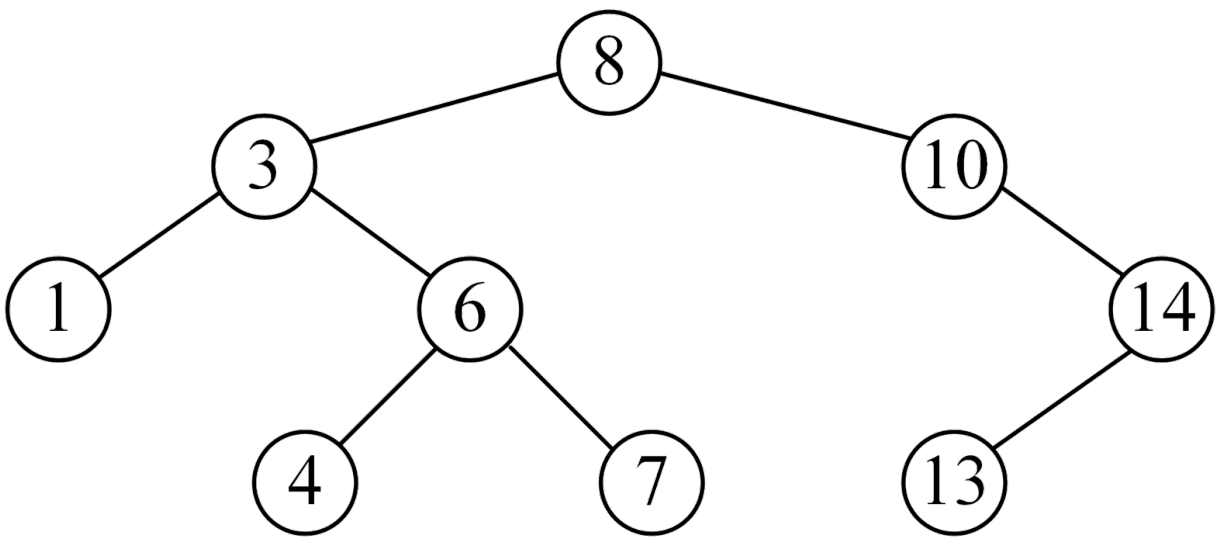
8.5.2 创建二叉搜索树

通过逐个插入元素来创建二叉搜索树：

创建二叉搜索树

```
int keys[] = { 8,3,10,1,6,14,4,7,13 };
BinaryTree<int> bstree;
    for (auto i:keys)
        if (Node<int> *n=bstree.insert(i) )
            cout << n->data() << "␣";
```

8,3,10,1,6,14,4,7,13



说明

插入新结点成功则返回结点指针，然后打印数据

说明

从根结点开始，若待插入结点小于根结点，则在左子树上继续查找插入位置；否则在右子树中查找

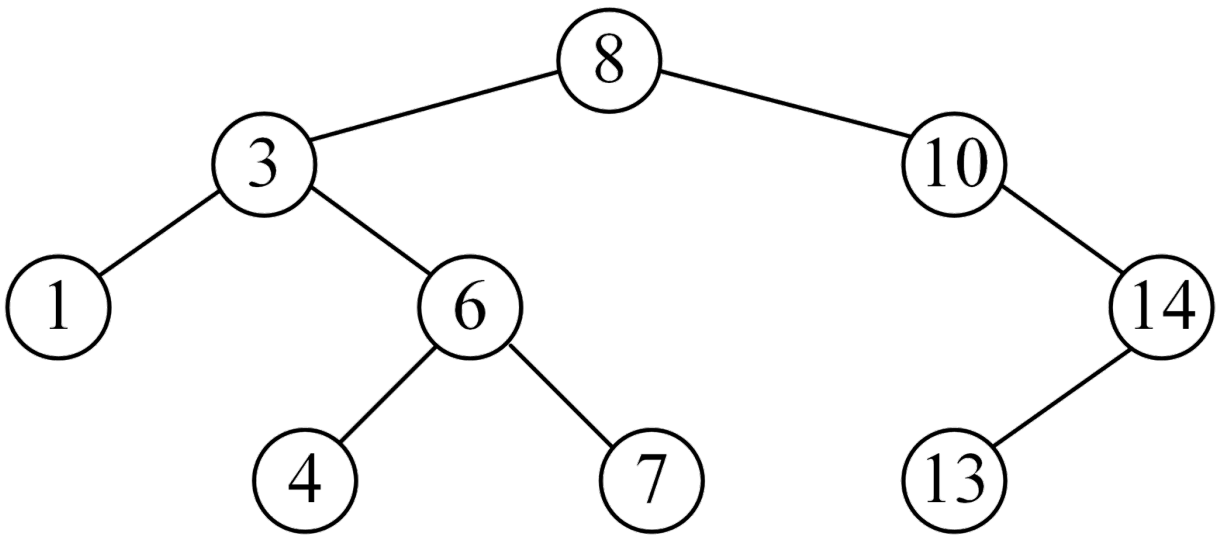
8.5.2 创建二叉搜索树

通过逐个插入元素来创建二叉搜索树：

创建二叉搜索树

```
int keys[] = { 8,3,10,1,6,14,4,7,13 };
BinaryTree<int> bstree;
    for (auto i:keys)
        if (Node<int> *n=bstree.insert(i) )
            cout << n->data() << "␣";
```

8,3,10,1,6,14,4,7,13



说明

插入新结点成功则返回结点指针，然后打印数据

说明

从根结点开始，若待插入结点小于根结点，则在左子树上继续查找插入位置；否则在右子树中查找

问题

如果改变插入顺序，如将 1 调到第一个插入，树的结构会有多大变化？

8.5.2 创建二叉搜索树

成员函数 insert_ 的实现如下：

创建二叉搜索树

```
template<typename T>
Node<T> * BinaryTree<T>::insert_(Node<T>* &p, const
  T &value){
    if (p == nullptr) //找到插入位置，创建新结点
        return p = new (std::nothrow) Node<T>(value);
    else if (value < p->m_data) //在左子树中查找
        return insert_(p->m_left, value);
    else                        //在右子树中查找
        return insert_(p->m_right, value);
}
```

说明

如果 new 运算失败，std::nothrow 保证返回空指针

8.5.2 创建二叉搜索树

成员函数 insert_ 的实现如下：

创建二叉搜索树

```
template<typename T>
Node<T> * BinaryTree<T>::insert_(Node<T>* &p, const
  T &value){
    if (p == nullptr) //找到插入位置，创建新结点
        return p = new (std::nothrow) Node<T>(value);
    else if (value < p->m_data) //在左子树中查找
        return insert_(p->m_left, value);
    else                        //在右子树中查找
        return insert_(p->m_right, value);
}
```

说明

如果 new 运算失败，std::nothrow 保证返回空指针

问题

第一个形参必须为 Node 类型的指针的引用，而不是指针，为什么？

8.5.2 创建二叉搜索树

成员函数 insert_ 的实现如下：

创建二叉搜索树

```
template<typename T>
Node<T> * BinaryTree<T>::insert_(Node<T>* &p, const
  T &value){
    if (p == nullptr) //找到插入位置，创建新结点
        return p = new (std::nothrow) Node<T>(value);
    else if (value < p->m_data) //在左子树中查找
        return insert_(p->m_left, value);
    else                        //在右子树中查找
        return insert_(p->m_right, value);
}
```

说明

如果 new 运算失败，std::nothrow 保证返回空指针

问题

第一个形参必须为 Node 类型的指针的引用，而不是指针，为什么？

答案

否则创建新结点时，只有局部对象 p 被改指向新的动态内存地址，真正的实参的值还是 nullptr

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构

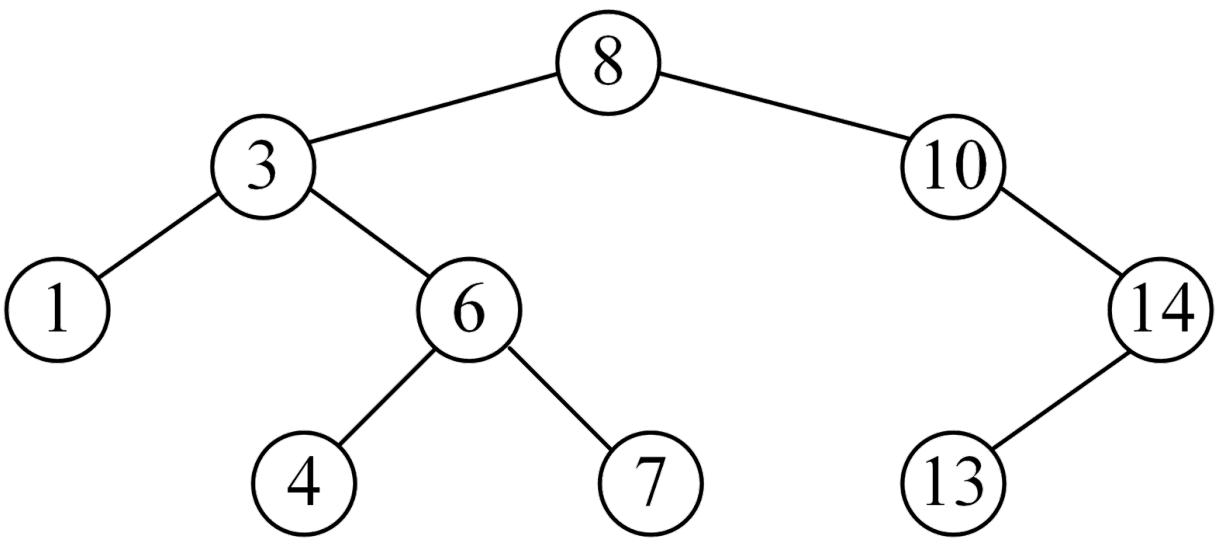
8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构

先序遍历

根结点–> 左子树–> 右子树



8.5.3 遍历操作

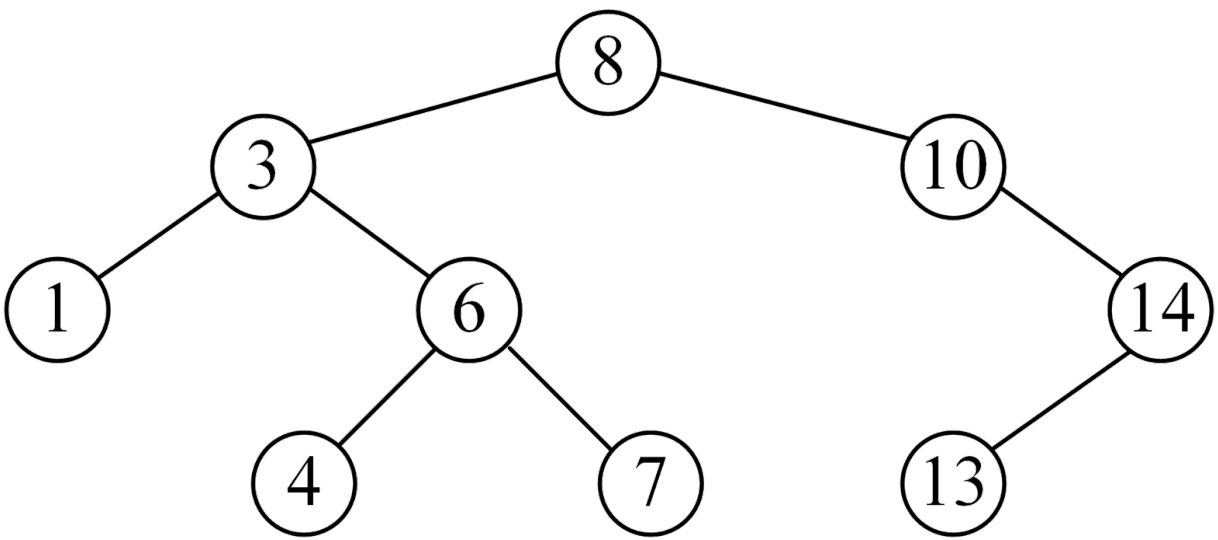
二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构

先序遍历

根结点–> 左子树–> 右子树

8



8.5.3 遍历操作

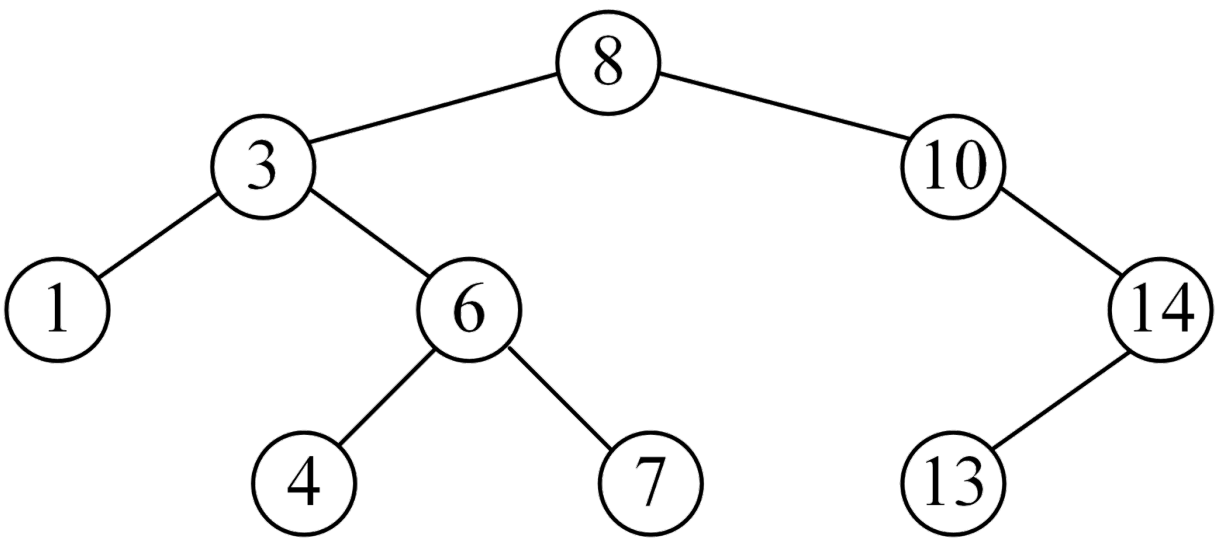
二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构

先序遍历

根结点–> 左子树–> 右子树

8 3



8.5.3 遍历操作

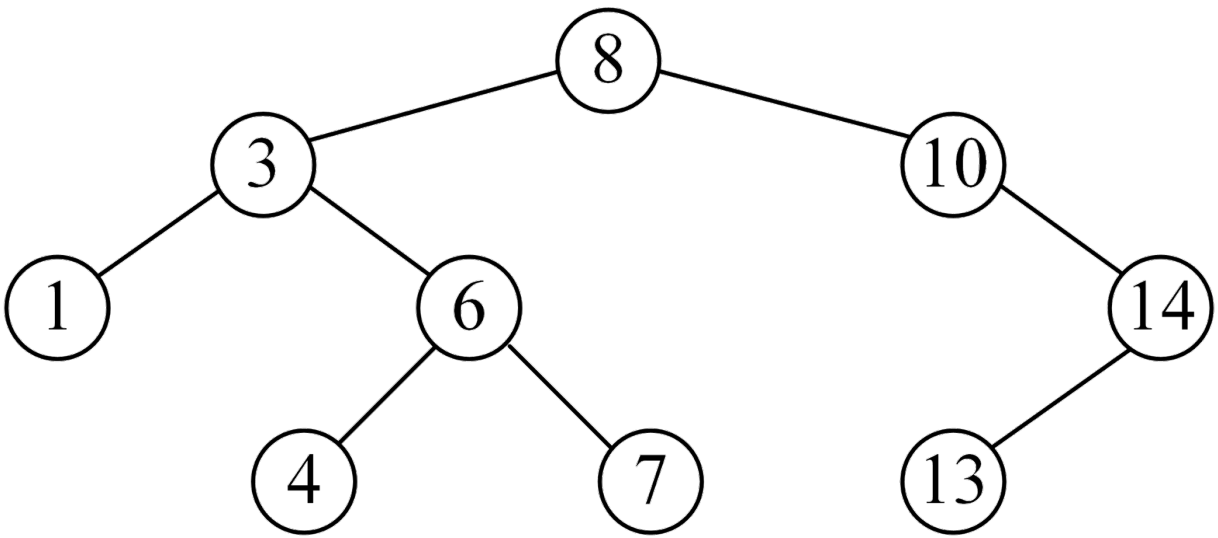
二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构

先序遍历

根结点–> 左子树–> 右子树

8 3 1



8.5.3 遍历操作

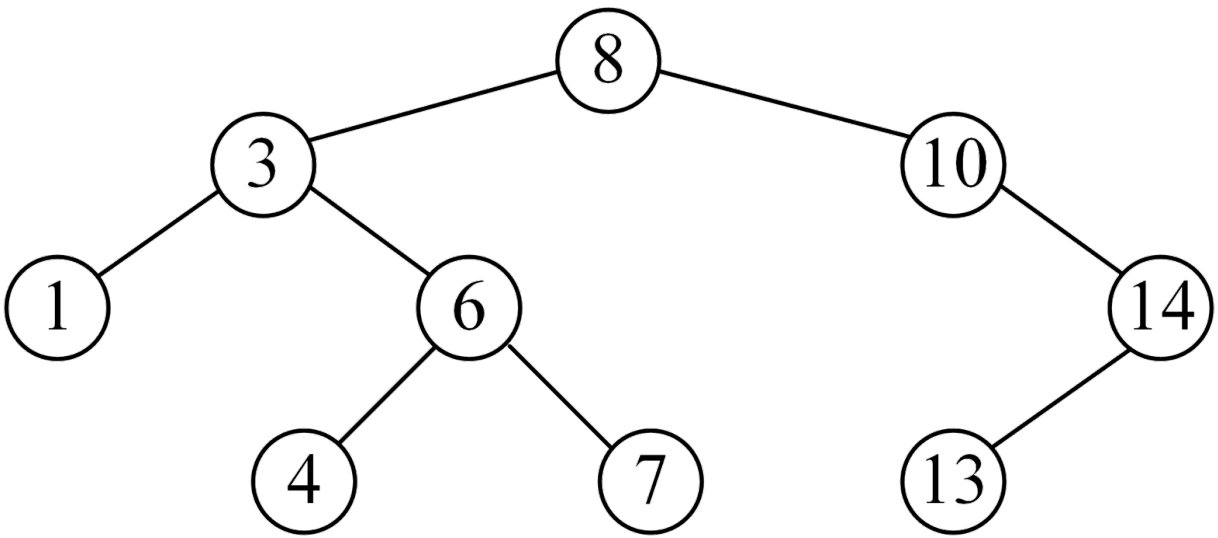
二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构

先序遍历

根结点–> 左子树–> 右子树

8 3 1 6



8.5.3 遍历操作

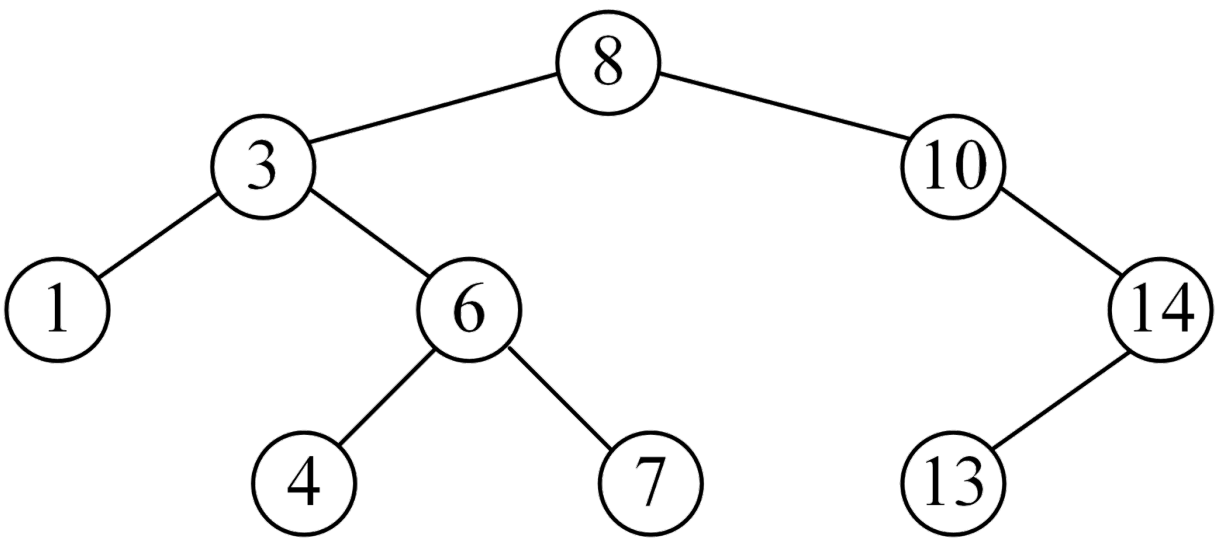
二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构

先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4



8.5.3 遍历操作

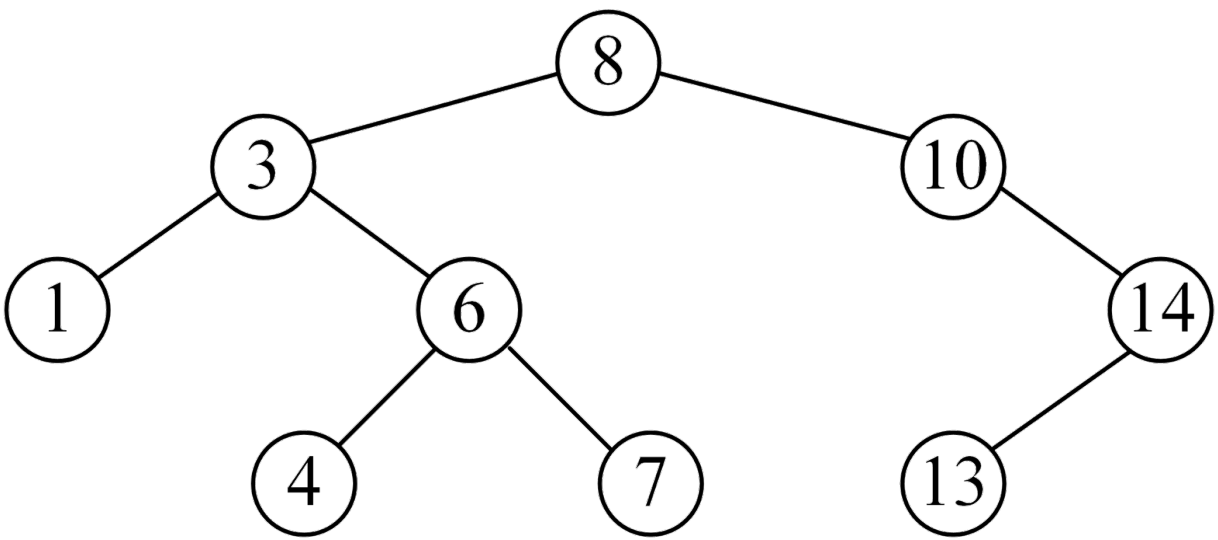
二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构

先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7



8.5.3 遍历操作

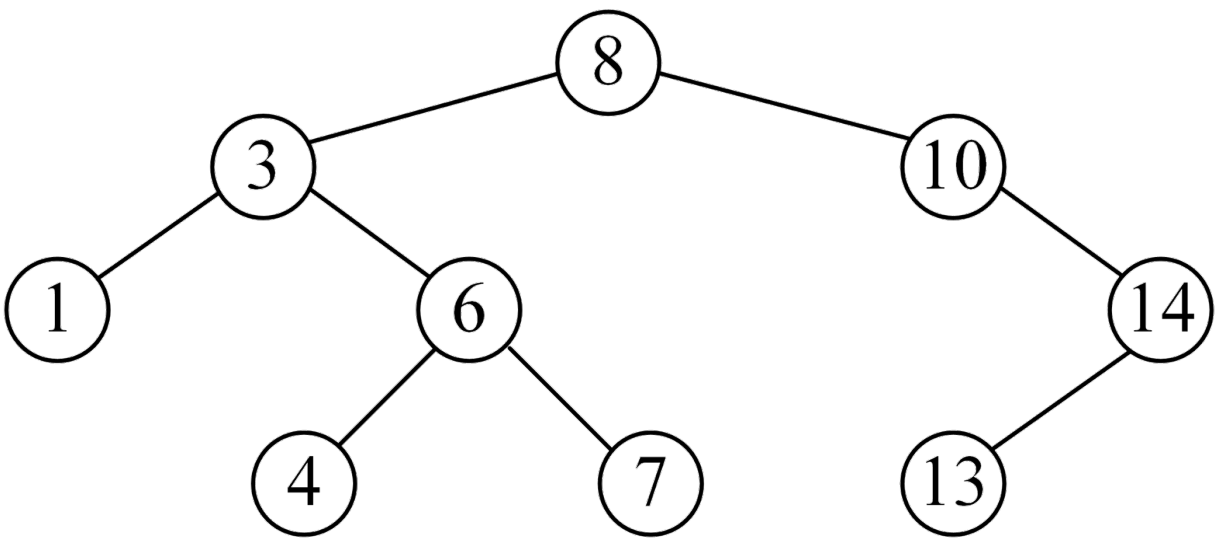
二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构

先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10



8.5.3 遍历操作

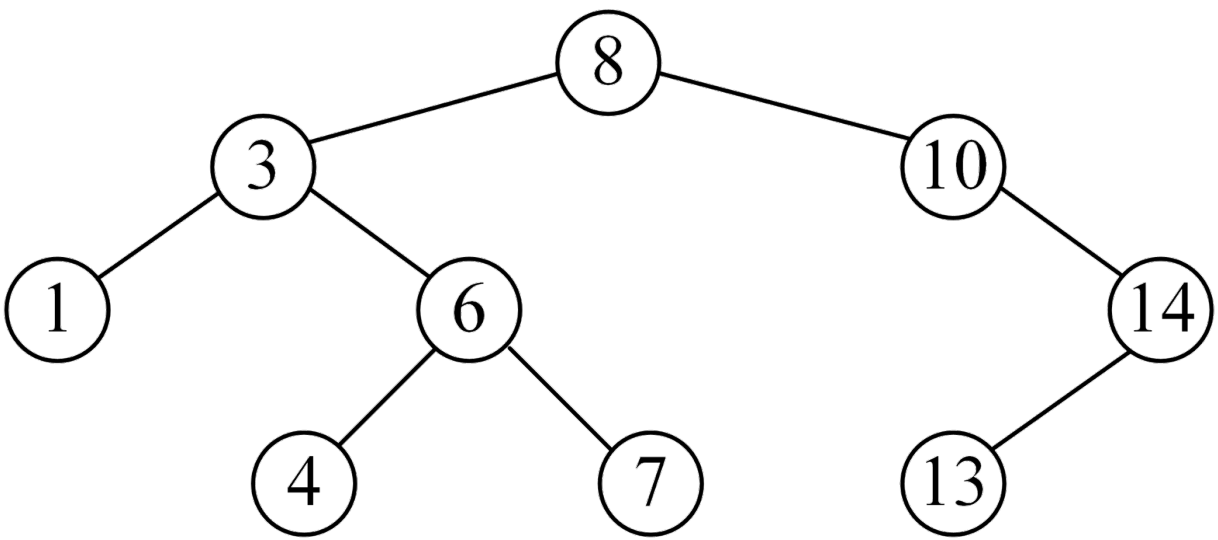
二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构

先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10 14



8.5.3 遍历操作

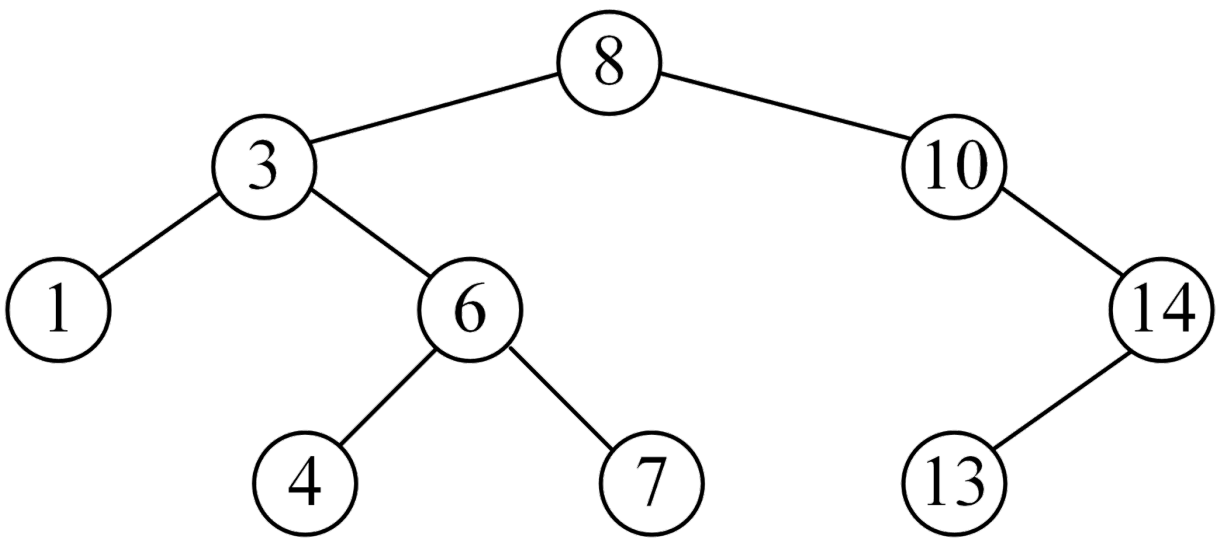
二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构

先序遍历

根结点–> 左子树–> 右子树

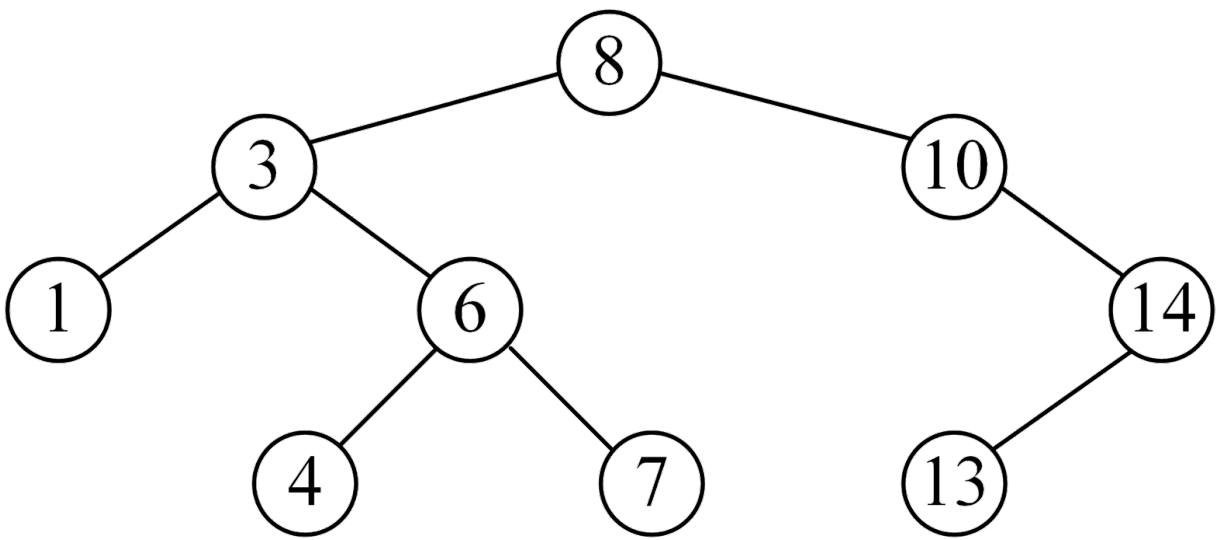
8 3 1 6 4 7 10 14 13



8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10 14 13

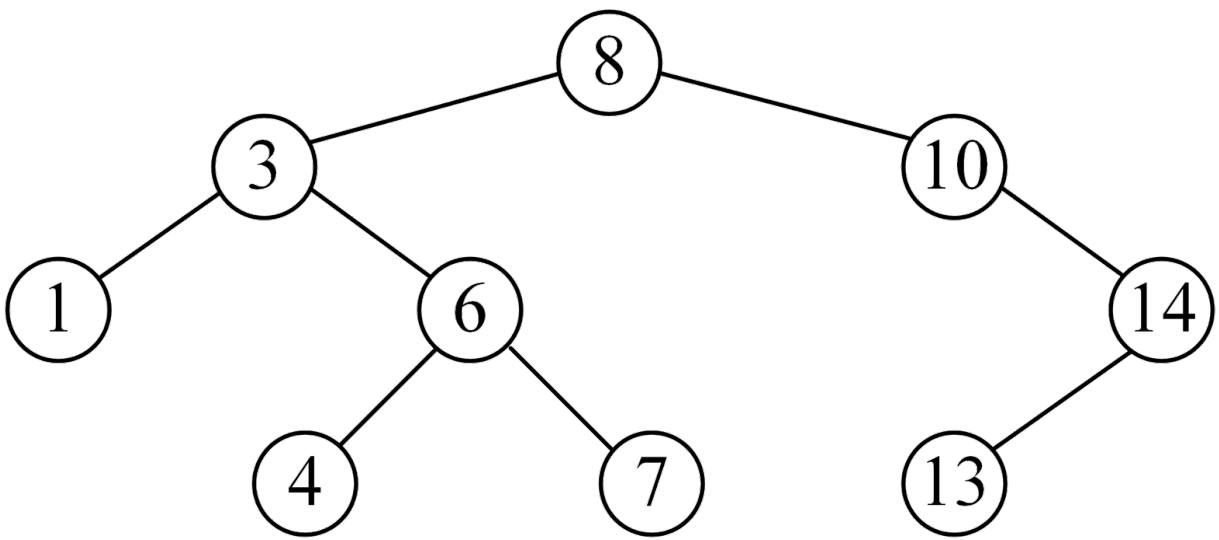
中序遍历

左子树–>根结点–> 右子树

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10 14 13

中序遍历

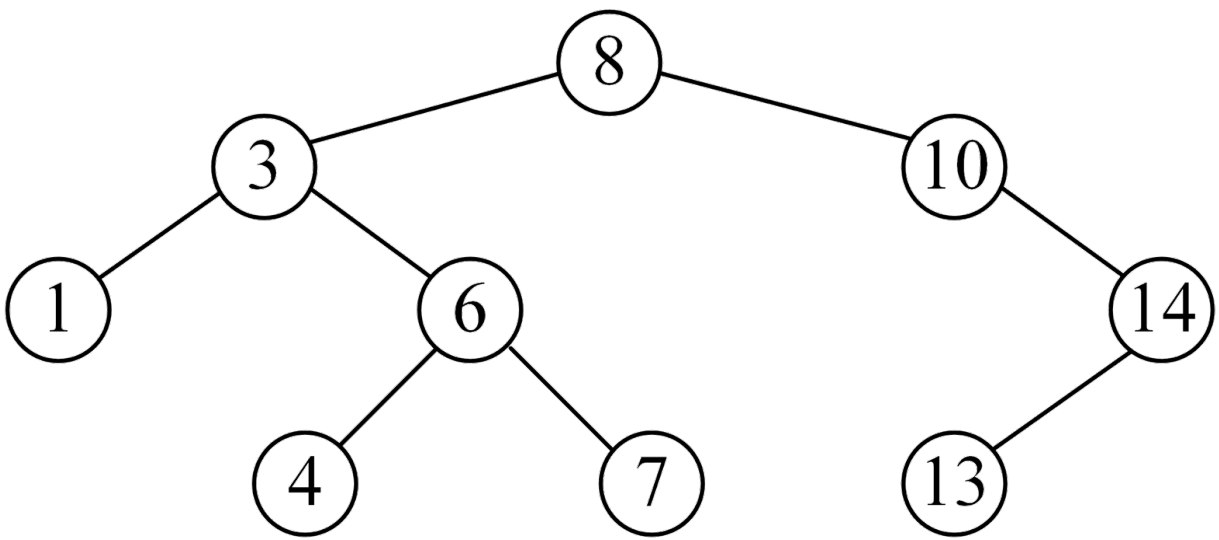
左子树–>根结点–> 右子树

1

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10 14 13

中序遍历

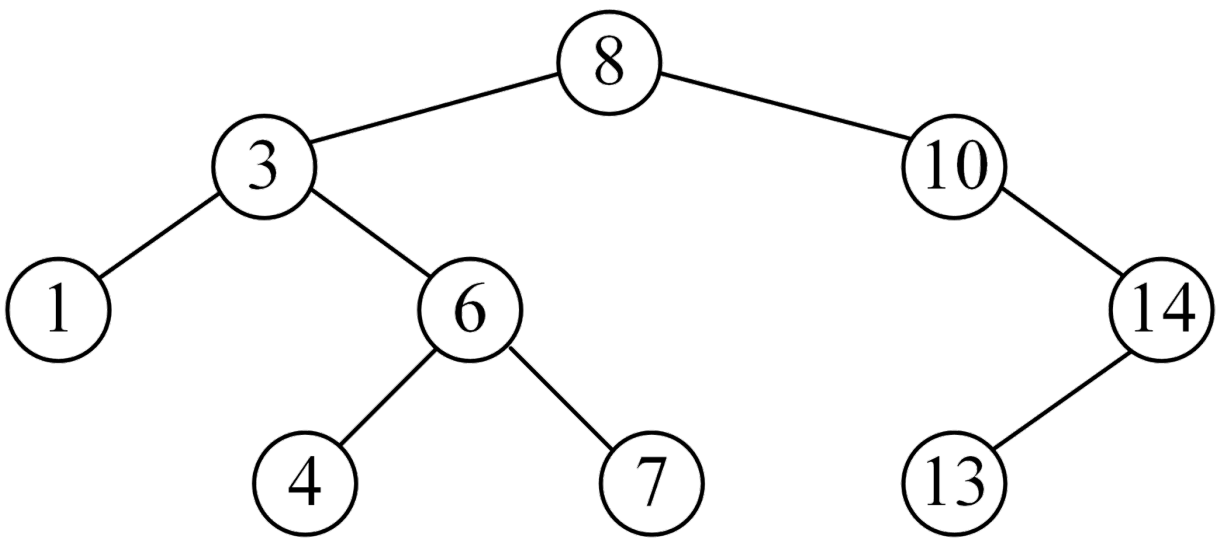
左子树–>根结点–> 右子树

1 3

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点-> 左子树-> 右子树

8 3 1 6 4 7 10 14 13

中序遍历

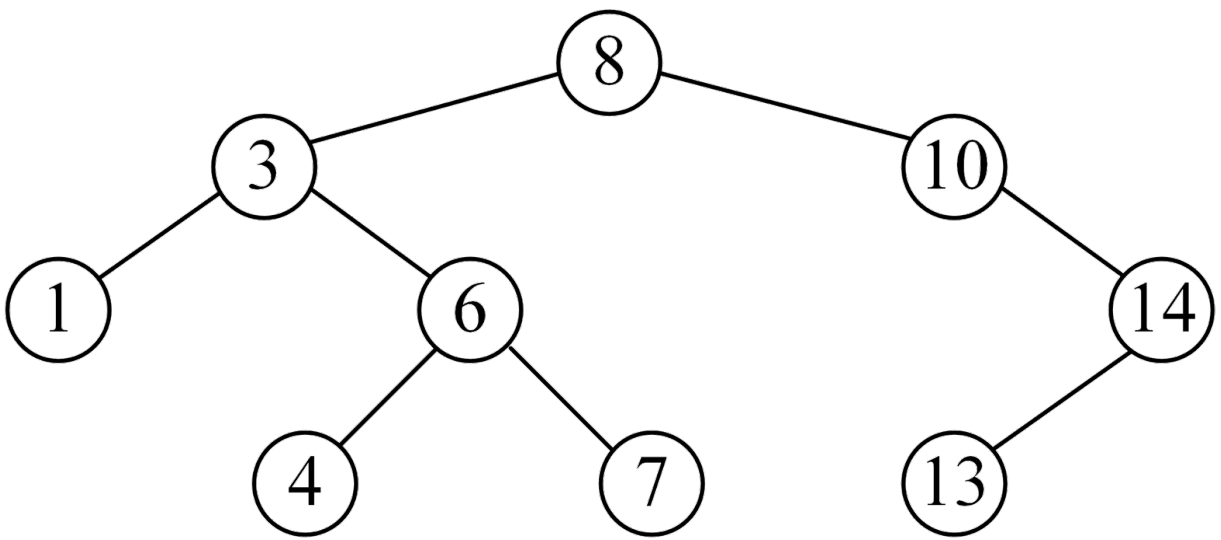
左子树->根结点-> 右子树

1 3 4

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点-> 左子树-> 右子树

8 3 1 6 4 7 10 14 13

中序遍历

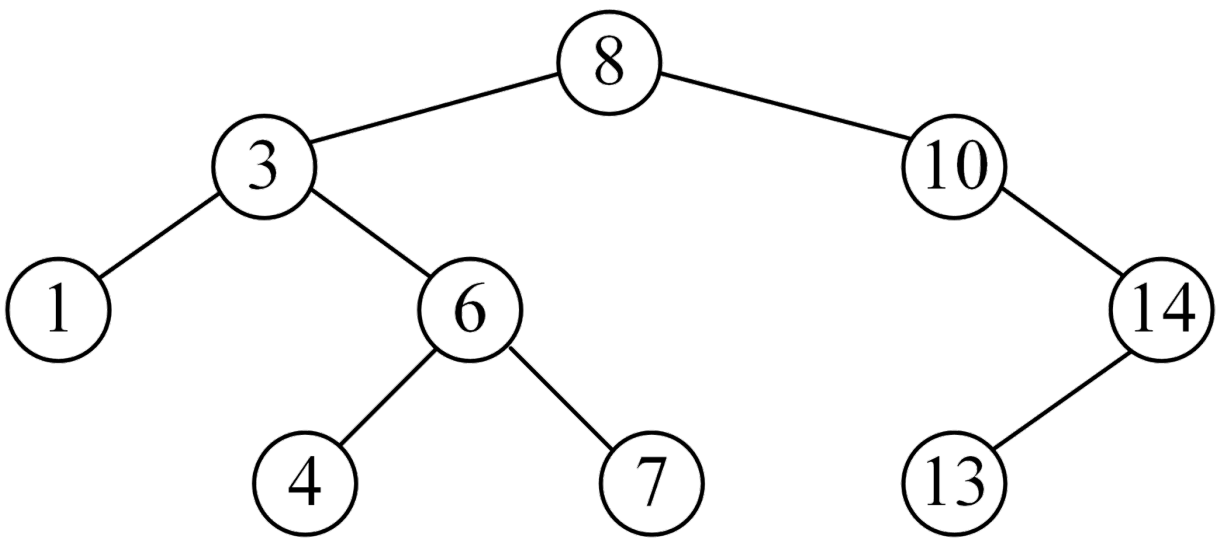
左子树->根结点-> 右子树

1 3 4 6

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10 14 13

中序遍历

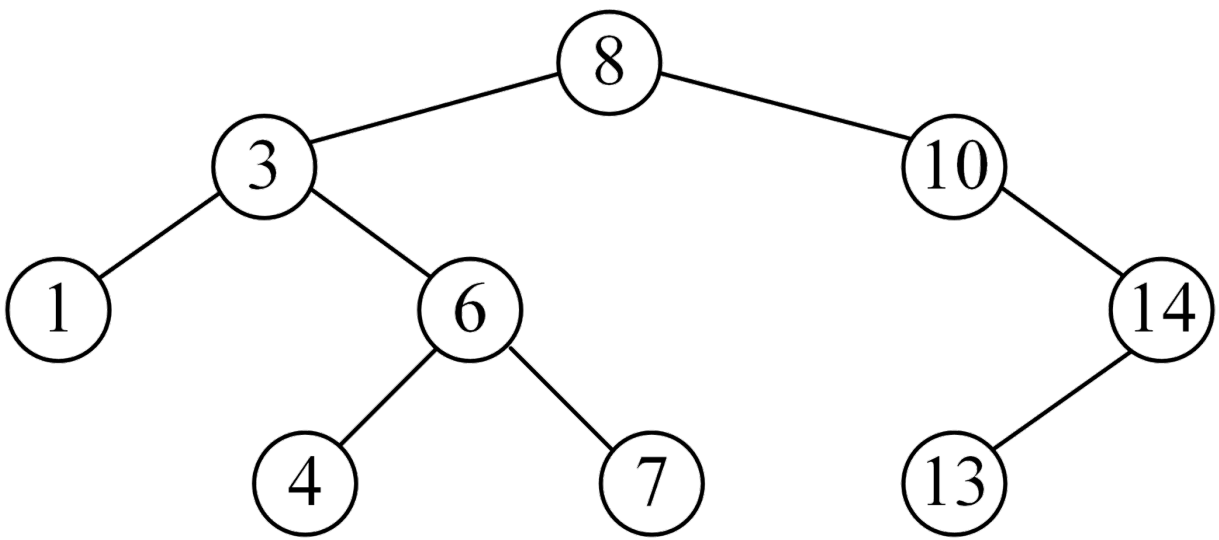
左子树–>根结点–> 右子树

1 3 4 6 7

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10 14 13

中序遍历

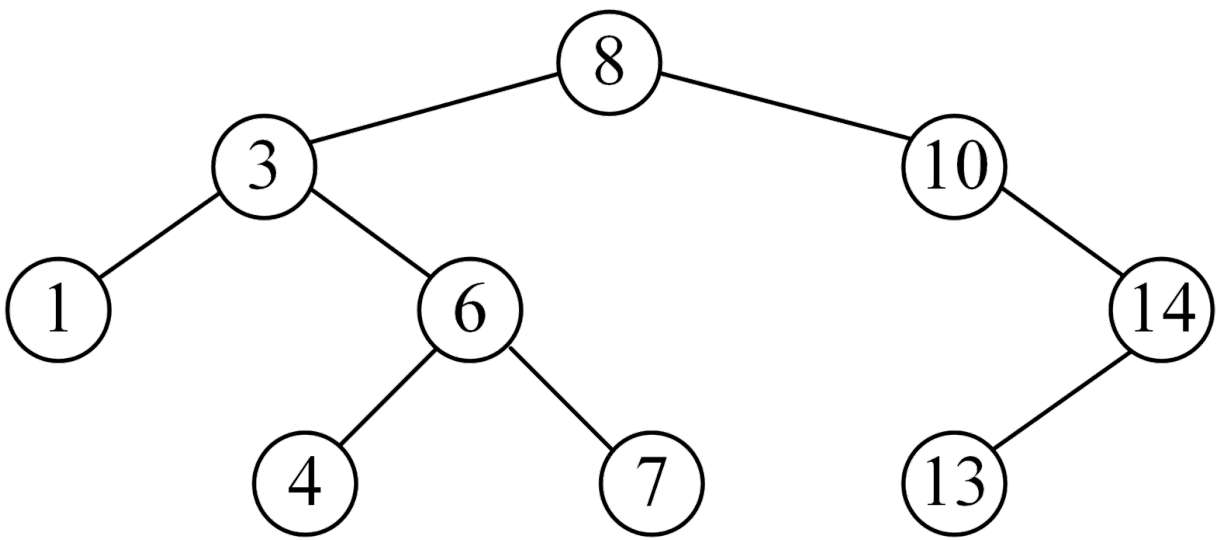
左子树–>根结点–> 右子树

1 3 4 6 7 8

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10 14 13

中序遍历

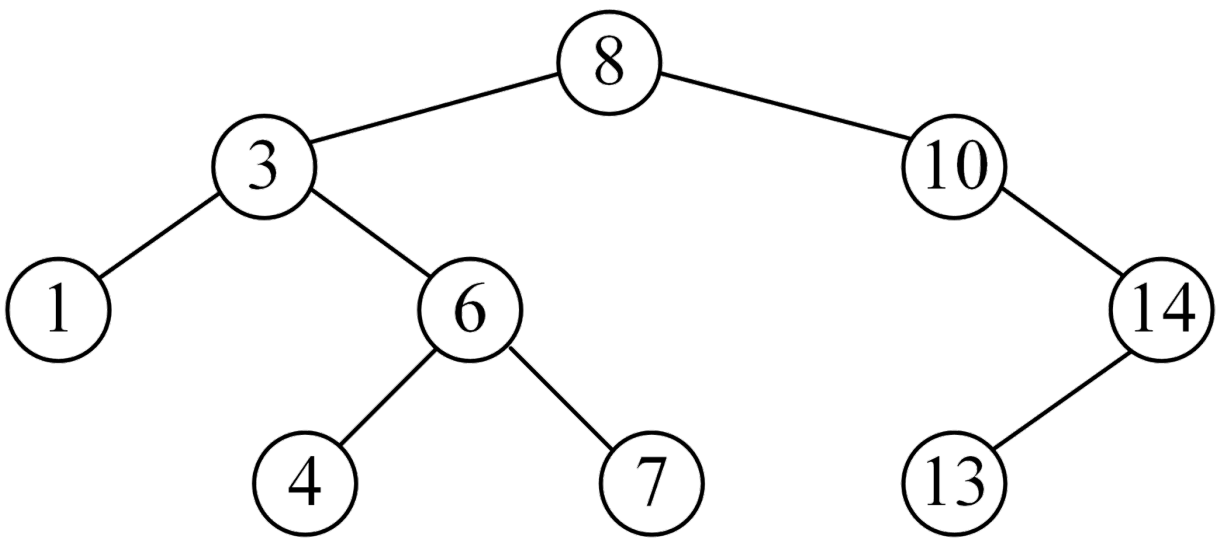
左子树–>根结点–> 右子树

1 3 4 6 7 8 10

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10 14 13

中序遍历

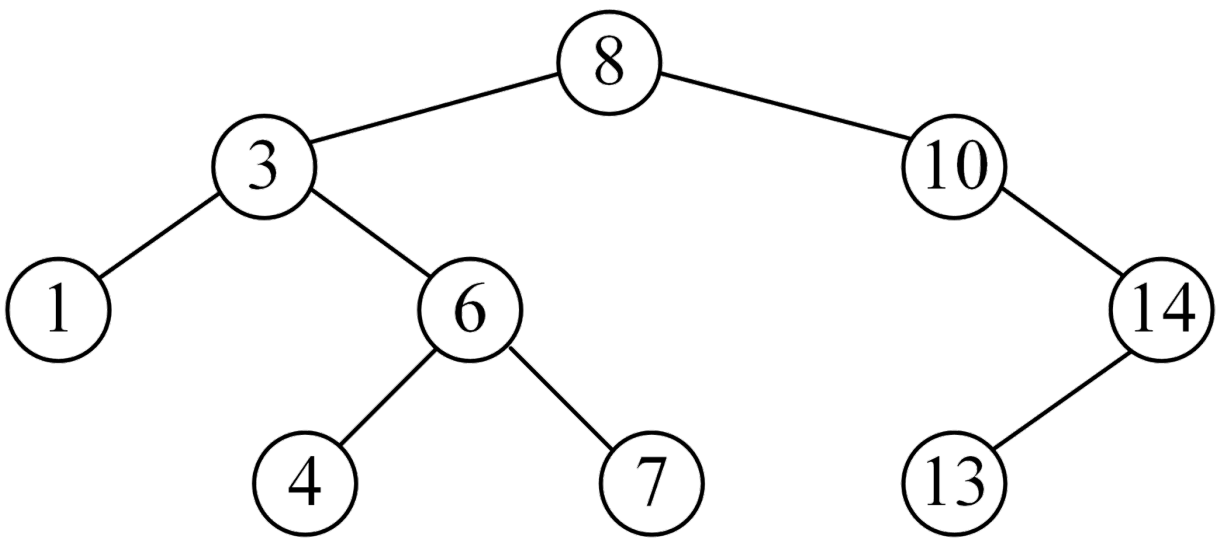
左子树–>根结点–> 右子树

1 3 4 6 7 8 10 13

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10 14 13

中序遍历

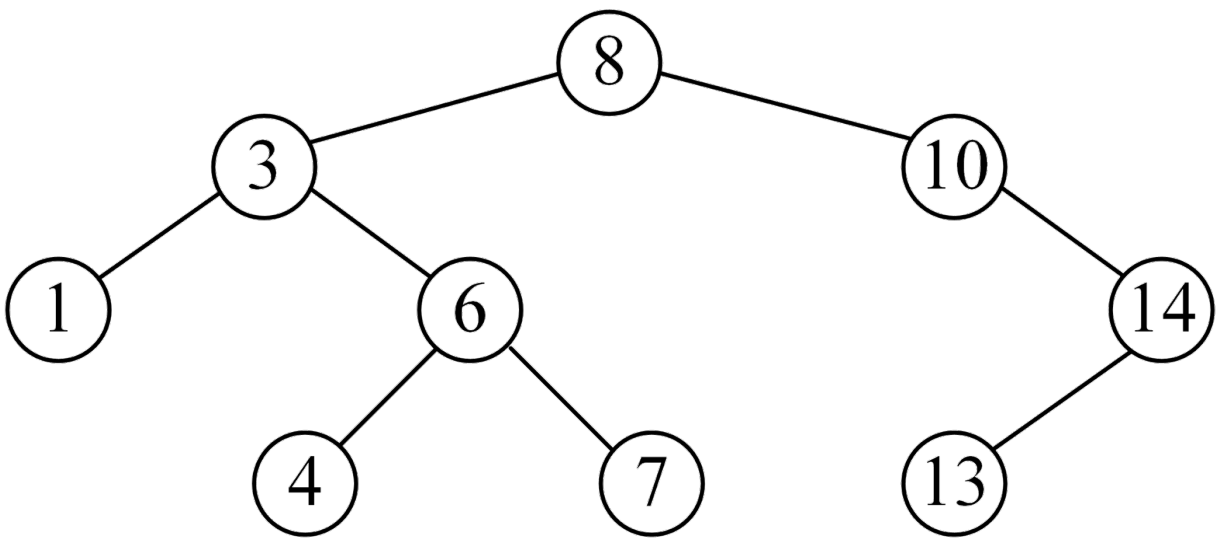
左子树–>根结点–> 右子树

1 3 4 6 7 8 10 13 14

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10 14 13

中序遍历

左子树–>根结点–> 右子树

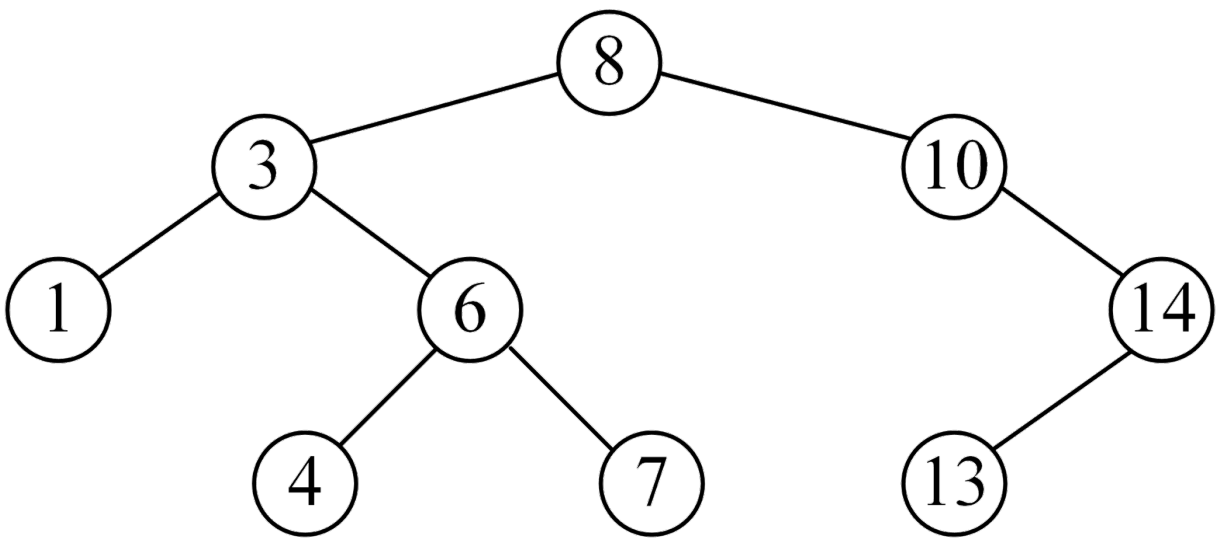
1 3 4 6 7 8 10 13 14

有序序列

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10 14 13

中序遍历

左子树–>根结点–> 右子树

1 3 4 6 7 8 10 13 14

有序序列

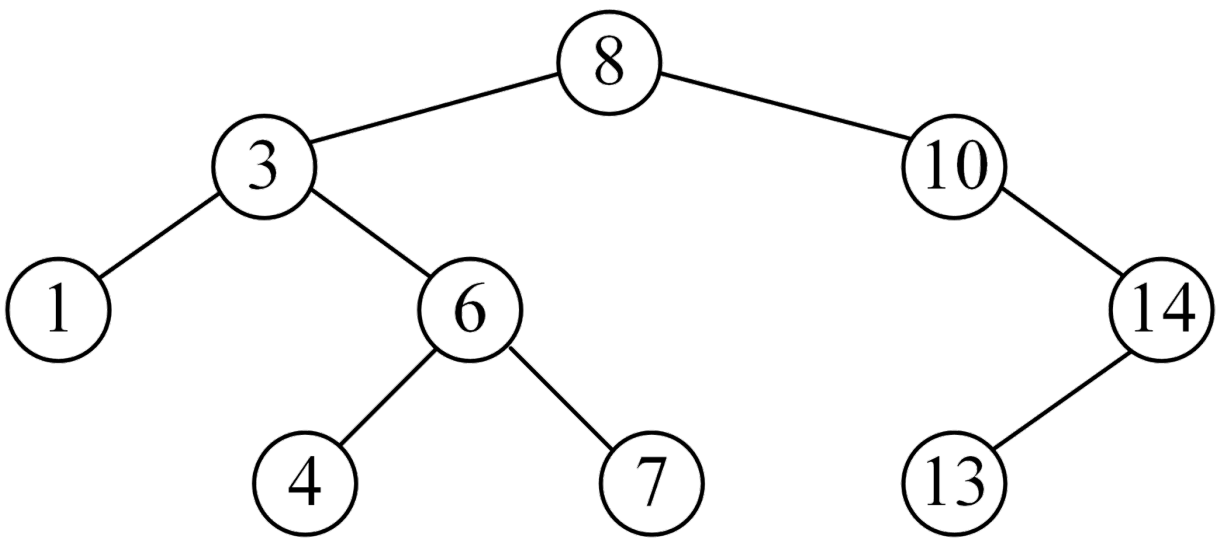
后序遍历

左子树–> 右子树–>根结点

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10 14 13

中序遍历

左子树–>根结点–> 右子树

1 3 4 6 7 8 10 13 14

有序序列

后序遍历

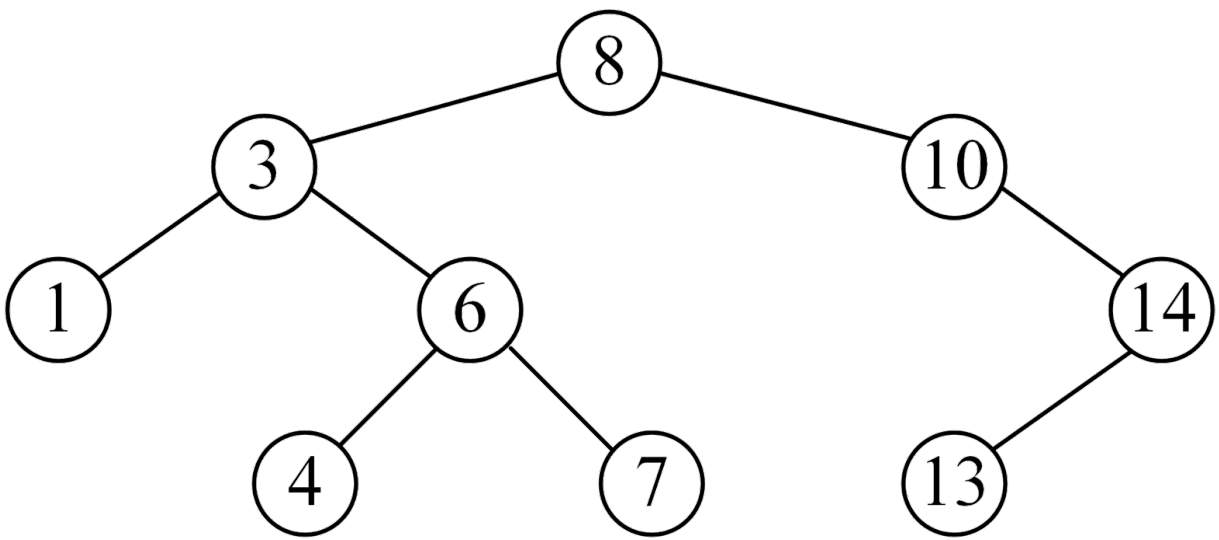
左子树–> 右子树–>根结点

1

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10 14 13

中序遍历

左子树–>根结点–> 右子树

1 3 4 6 7 8 10 13 14

有序序列

后序遍历

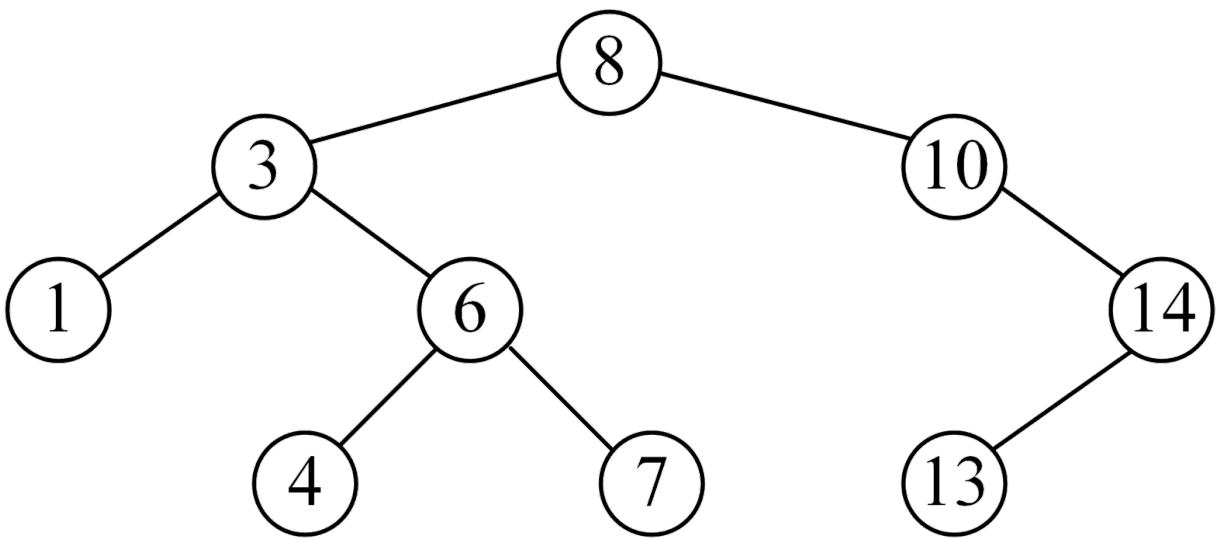
左子树–> 右子树–>根结点

1 4

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10 14 13

中序遍历

左子树–>根结点–> 右子树

1 3 4 6 7 8 10 13 14

有序序列

后序遍历

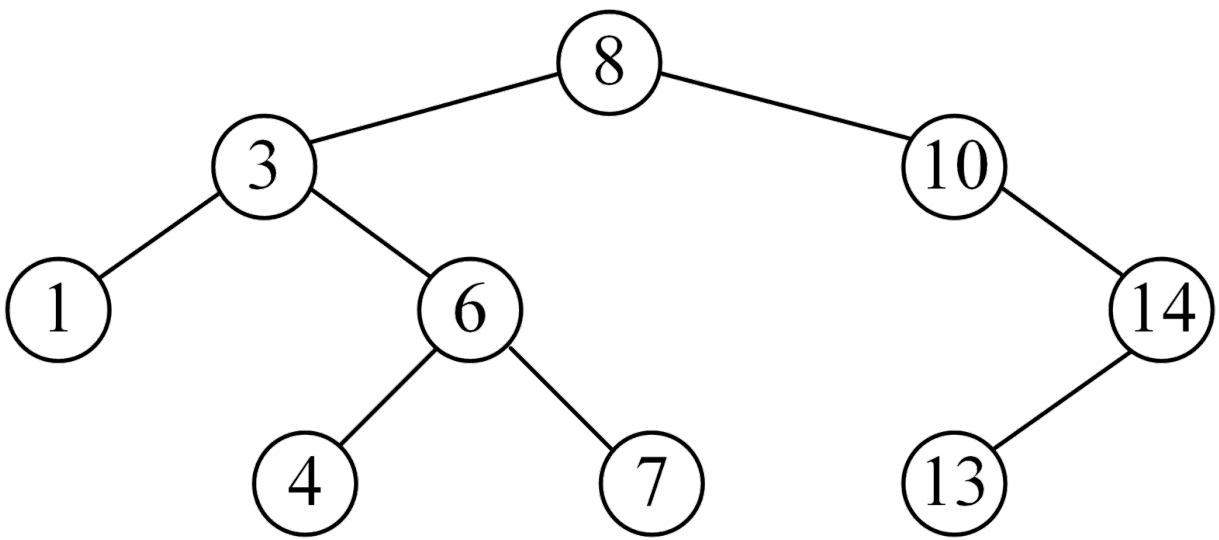
左子树–> 右子树–>根结点

1 4 7

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10 14 13

中序遍历

左子树–>根结点–> 右子树

1 3 4 6 7 8 10 13 14

有序序列

后序遍历

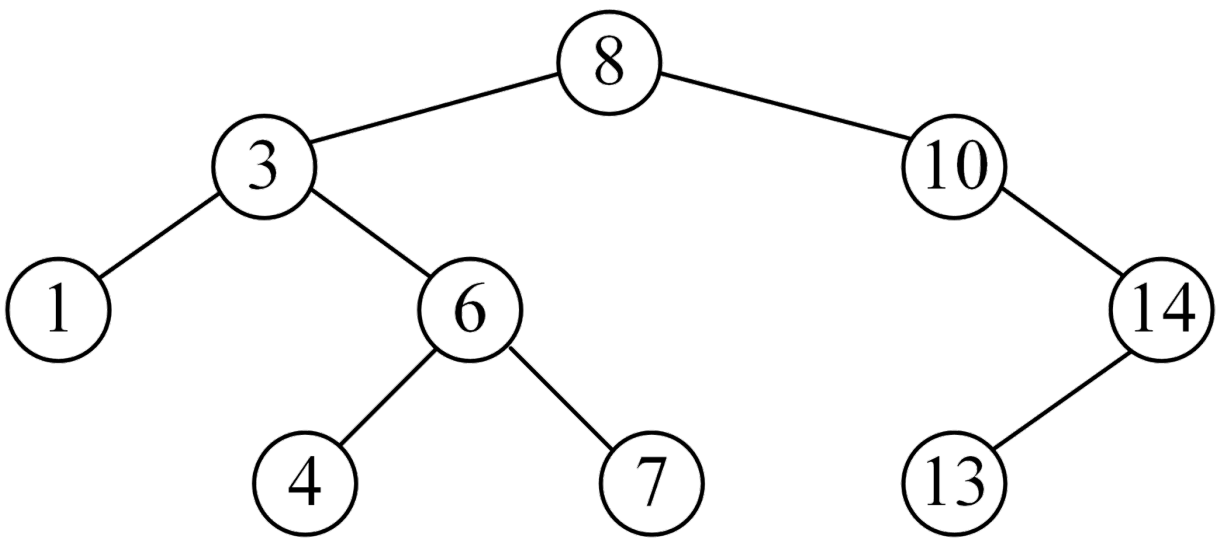
左子树–> 右子树–>根结点

1 4 7 6

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10 14 13

中序遍历

左子树–>根结点–> 右子树

1 3 4 6 7 8 10 13 14

有序序列

后序遍历

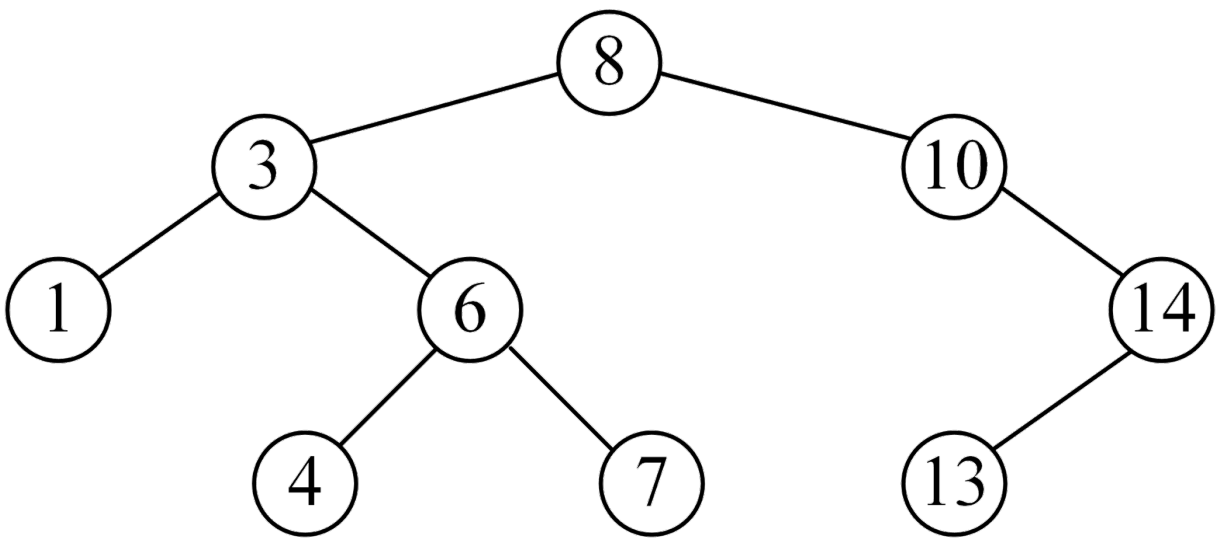
左子树–> 右子树–>根结点

1 4 7 6 3

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10 14 13

中序遍历

左子树–>根结点–> 右子树

1 3 4 6 7 8 10 13 14

有序序列

后序遍历

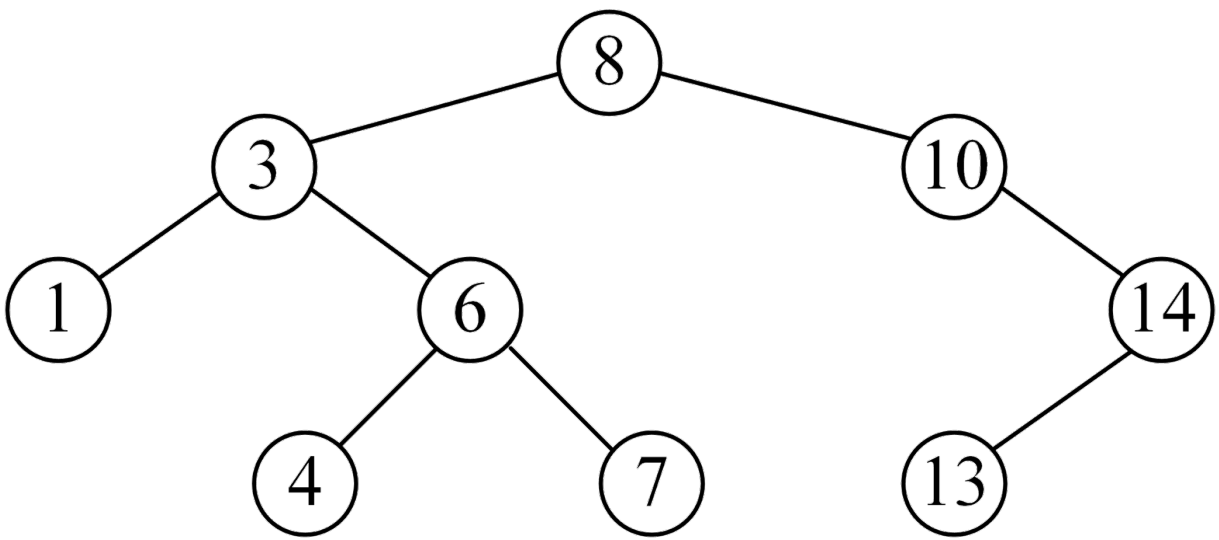
左子树–> 右子树–>根结点

1 4 7 6 3 13

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10 14 13

中序遍历

左子树–>根结点–> 右子树

1 3 4 6 7 8 10 13 14

有序序列

后序遍历

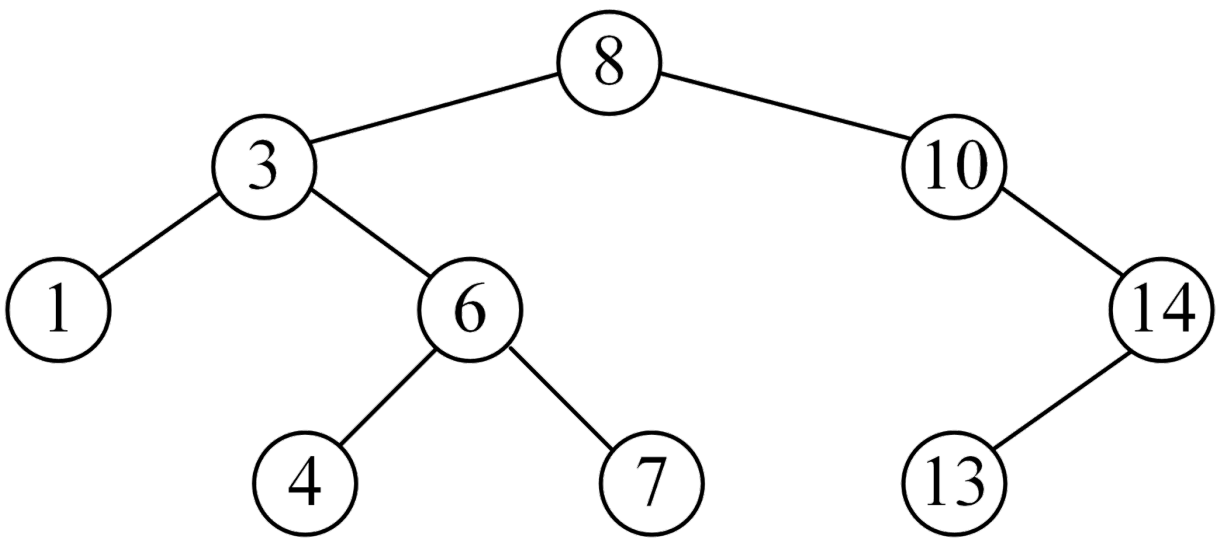
左子树–> 右子树–>根结点

1 4 7 6 3 13 14

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10 14 13

中序遍历

左子树–>根结点–> 右子树

1 3 4 6 7 8 10 13 14

有序序列

后序遍历

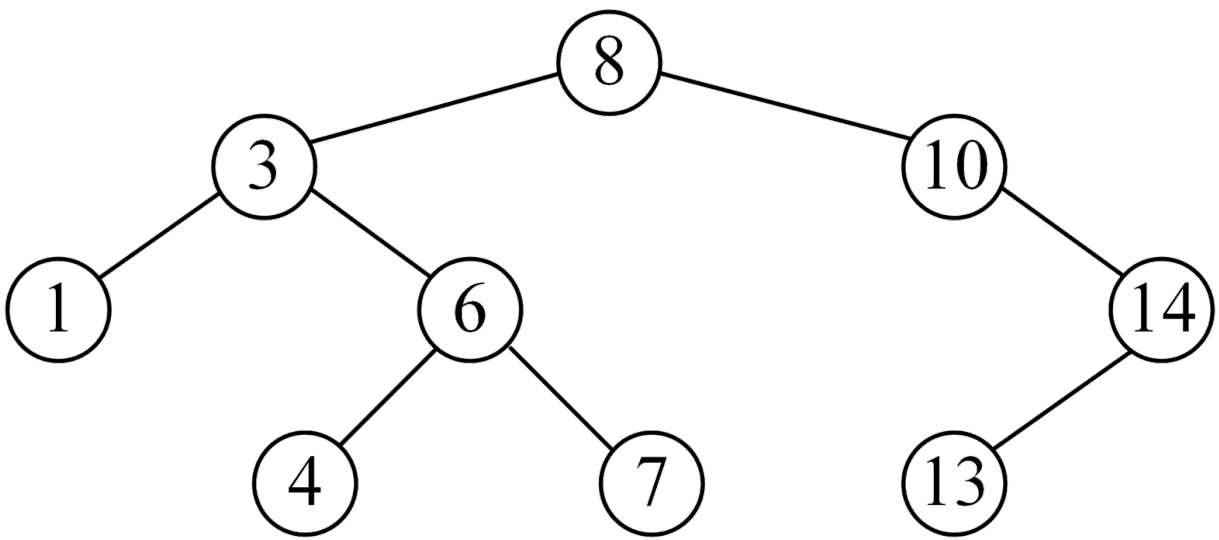
左子树–> 右子树–>根结点

1 4 7 6 3 13 14 10

8.5.3 遍历操作

二叉树的遍历

- 根据某种次序访问树中每个结点一次且仅一次
- 访问的过程中可以根据需要对结点的数据进行不同的处理操作，但不能改变原来的结构



先序遍历

根结点–> 左子树–> 右子树

8 3 1 6 4 7 10 14 13

中序遍历

左子树–>根结点–> 右子树

1 3 4 6 7 8 10 13 14

有序序列

后序遍历

左子树–> 右子树–>根结点

1 4 7 6 3 13 14 10 8

8.5.3 遍历操作

以中序遍历的实现为例：

示例代码

```
template<typename T>
void BinaryTree<T>::inOrder(Node<T> *p,void (*visit)(T&)){
    if (p != nullptr){
        inOrder(p->m_left, visit); //遍历左子树
        visit(p->m_data); //用户自定义访问函数
        inOrder(p->m_right, visit); //遍历右子树
    }
}
```

说明

第二个形参为一个返回值为空、包含一个 T& 类型形参的函数指针，指向用户自定义的访问处理函数

8.5.3 遍历操作

以中序遍历的实现为例：

示例代码

```
template<typename T>
void BinaryTree<T>::inOrder(Node<T> *p,void (*visit)(T&)){
    if (p != nullptr){
        inOrder(p->m_left, visit); //遍历左子树
        visit(p->m_data); //用户自定义访问函数
        inOrder(p->m_right, visit); //遍历右子树
    }
}
```

说明

第二个形参为一个返回值为空、包含一个 T& 类型形参的函数指针，指向用户自定义的访问处理函数

定义一个简单的访问函数模板：

visit 函数模板定义

```
template<typename T>
void visit(T &value) { cout << value << "␣"; }
```

说明

打印结点的数据

8.5.3 遍历操作

以中序遍历的实现为例：

示例代码

```
template<typename T>
void BinaryTree<T>::inOrder(Node<T> *p,void (*visit)(T&)){
    if (p != nullptr){
        inOrder(p->m_left, visit); //遍历左子树
        visit(p->m_data); //用户自定义访问函数
        inOrder(p->m_right, visit); //遍历右子树
    }
}
```

说明

第二个形参为一个返回值为空、包含一个 T& 类型形参的函数指针，指向用户自定义的访问处理函数

定义一个简单的访问函数模板：

visit 函数模板定义

```
template<typename T>
void visit(T &value) { cout << value << "␣"; }
```

说明

打印结点的数据

中序遍历之前创建的二叉搜索树：

```
bstree.inOrder(bstree.root(), visit<int>);
```

输出结果为：1 3 4 6 7 8 10 13 14

8.5.4 搜索操作

根据二叉排序树的性质，可以采用二分法来实现快速搜索：

成员函数 search_ 定义

```
template<typename T>
Node<T>* BinaryTree<T>::search_(Node<T> *p, const T &value
  )const{
    while (p != nullptr && p->m_data != value){
        if (value < p->m_data)
            p = p->m_left;
        else
            p = p->m_right;
    }
    return p;
}
```

说明

将返回第一个数据值为 value 的结点的指针

8.5.5 销毁操作

采用后序方式逐个释放每个结点的内存：

成员函数 destroy 定义

```
template<typename T>
void BinaryTree<T>::destroy(Node<T> *p){
    if (p != nullptr){
        destroy(p->m_left);//销毁左子树
        destroy(p->m_right);//销毁右子树
        delete p;//释放根结点内存
    }
}
```

说明

- 释放给定结点及其左右子树的内存
- 访问权限声明为私有，是析构函数的实现

8.5.5 销毁操作

采用后序方式逐个释放每个结点的内存：

成员函数 destroy 定义

```
template<typename T>
void BinaryTree<T>::destroy(Node<T> *p){
    if (p != nullptr){
        destroy(p->m_left);//销毁左子树
        destroy(p->m_right);//销毁右子树
        delete p;//释放根结点内存
    }
}
```

说明

- 释放给定结点及其左右子树的内存
- 访问权限声明为私有，是析构函数的实现

注意

若在它处执行此函数后，必须把给定结点的父结点（如有）指向此结点的指针成员置空，否则成为空悬指针

8.5.6 拷贝控制及友元声明

类似于单链表，二叉树中的结点以及二叉树本身不允许执行默认的拷贝成员，因此将它们声明为 delete

BianryTree 和 Node 类模板的拷贝控制及友元声明

```
template<typename T> class BinaryTree;//前向声明
template<typename T>
class Node{
friend class BinaryTree<T>;
public:
    Node(const Node&) = delete;
    Node& operator=(const Node&) = delete;
    //其它成员保持不变
};

template<typename T>
class BinaryTree{
public:
    BinaryTree() = default; //使用默认的构造函数
    BinaryTree(const BinaryTree &) = delete;
    BinaryTree& operator=(const BinaryTree &) = delete;
    //其它成员保持不变
};
```

说明

同时将 BinaryTree 类模板声明为 Node 类模板的友元

本章结束