第八章 动态内存与数据结构

动态数据

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

动态数据

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。支持内存管理使得 C/C++ 语言有了出色的性能。

目录

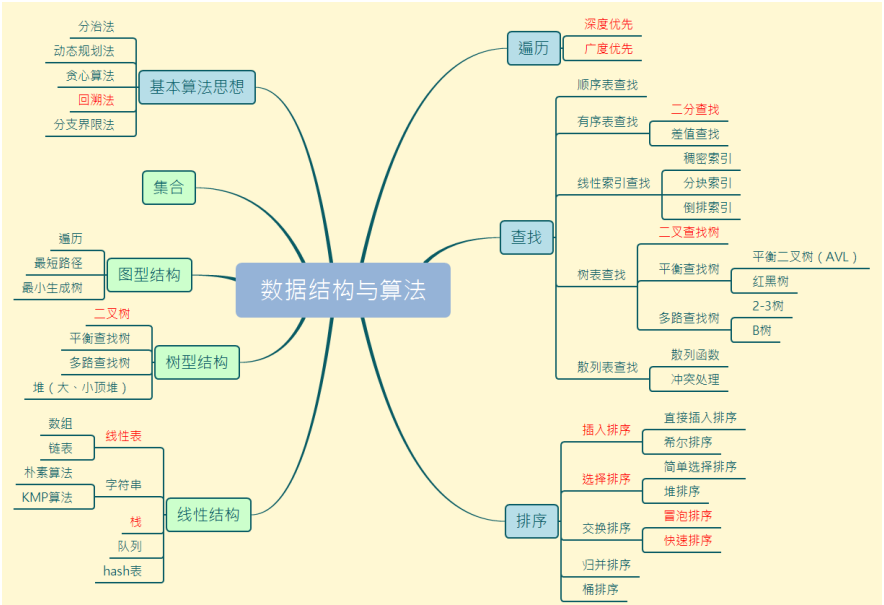
1. 线性链表
   - 链表表示
   - 插入操作
   - 删除操作
   - 清空链表
   - 打印链表
   - 拷贝控制与友元声明
2. 链栈
   - 链栈表示与操作
   - 简单计算器
3. 二叉树
   - 二叉树的概念和表示
   - 创建二叉搜索树
   - 遍历操作
   - 搜索操作

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。

如何在计算机中组织数据和高效处理数据（如添加、查询、修改等)？

数据结构与算法


数据结构与算法
基本算法思想
分治法
动态规划法
贪心算法
回溯法
分支界限法
集合
图型结构
遍历
最短路径
最小生成树
树型结构
二叉树
平衡查找树
多路查找树
堆（大、小顶堆）
线性结构
数组
链表
线性表
朴素算法
KMP算法
字符串
栈
队列
hash表
遍历
深度优先
广度优先
查找
顺序表查找
有序表查找
二分查找
差值查找
线性索引查找
稠密索引
分块索引
倒排索引
树表查找
二叉查找树
平衡查找树
平衡二叉树（AVL）
红黑树
多路查找树
2-3树
B树
散列表查找
散列函数
冲突处理
排序
插入排序
直接插入排序
希尔排序
选择排序
简单选择排序
堆排序
交换排序
冒泡排序
快速排序
归并排序
桶排序

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。

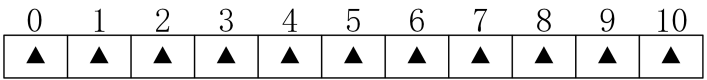
数组是一种线性结构，在逻辑结构上相邻的元素在物理结构上也相邻：

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：

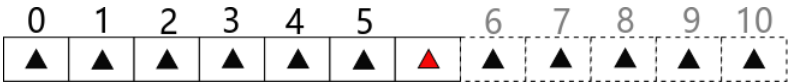


8.3 线性链表

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：

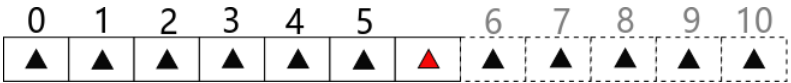


8.3 线性链表

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：



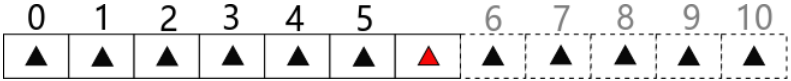
线性链表为链式结构，在逻辑结构上相邻的元素在物理结构上不要求相邻：

8.3 线性链表

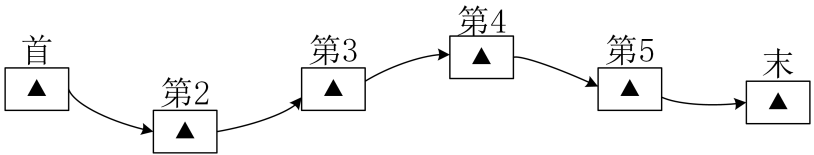
线性链表

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：

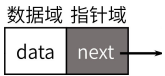


线性链表为链式结构，在逻辑结构上相邻的元素在物理结构上不要求相邻：



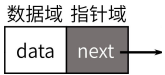
8.3.1 链表表示

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 SList 定义

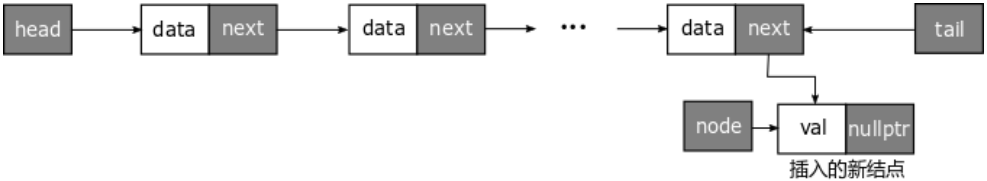
```
template<typename T>
class SList {
    Node<T> *m_head= nullptr, *m_tail= nullptr;
public:
    SList()= default; // 使用默认构造函数
    ~SList();
    void clear();
    void push_back(const T &val);
    Node<T>* insert(Node<T> *pos, const T &val);
    void erase(const T &val);
    Node<T>* find(const T &val);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

说明

- clear 函数清空所有元素
- 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：

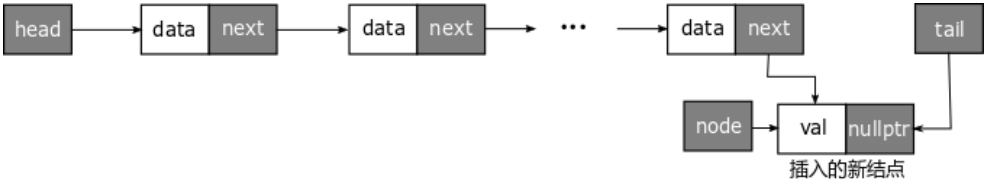


尾插操作 push_back 函数定义

```
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;
    }
}
```

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 push_back 函数定义

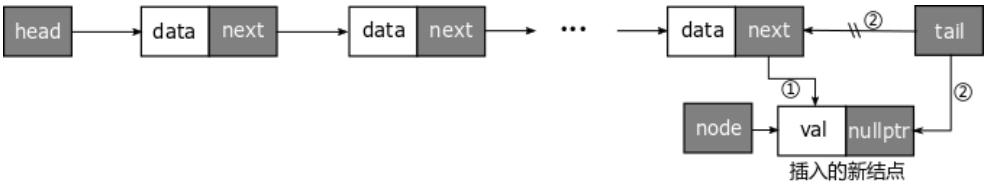
```
template<typename T>
void SList<T>::push_back(const T &val) {
     Node<T> *node = new Node<T>(val);// 创建新结点
    if (m_head == nullptr)
        m_head = m_tail= node;
    else {
         m_tail->m_next = node;
         m_tail = node;
    }
}
```

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 — 尾插

尾插操作将新结点插入到链表的表尾：



尾插操作 push_back 函数定义

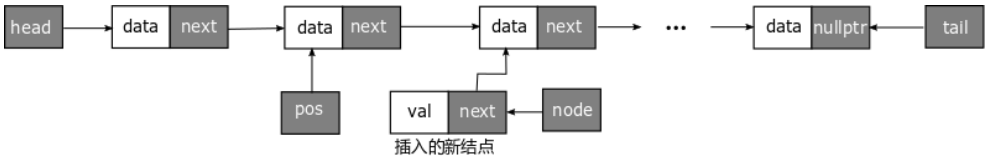
```
template<typename T>
void SList<T>::push_back(const T &val) {
     Node<T> *node = new Node<T>(val);// 创建新结点
    if (m_head == nullptr)
        m_head = m_tail= node;
    else {
         m_tail->m_next = node;
         m_tail = node;
    }
}
```

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：

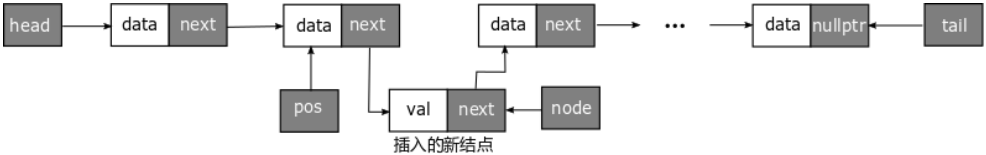


插入操作 insert 函数定义

```
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;
}
```

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 insert 函数定义

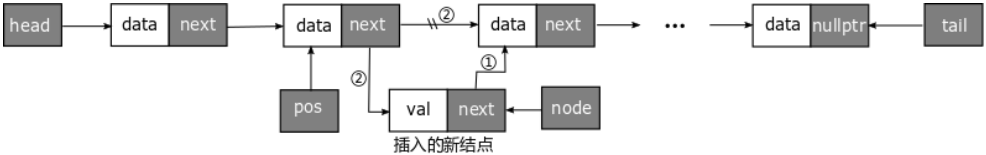
```
template<typename T>
Node<T>* SList<T>::insert(Node<T> *pos, const T &val) {
    Node<T> *node = new Node<T>(val); // 创建新结点
    node->m_next = pos->m_next;
    pos->m_next = node;
    if (pos == m_tail) // 判断pos是否为尾结点
        m_tail = node;
    return node;
}
```

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 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

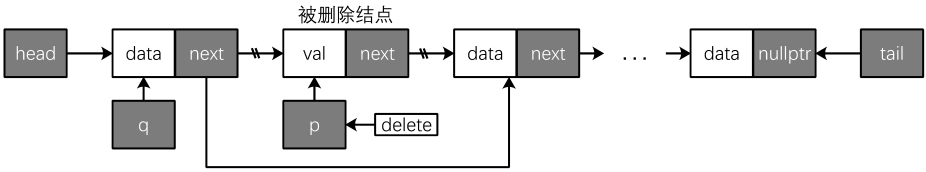
```
template<typename T>
Node<T>* SList<T>::find(const T &val) {
    Node<T> *p = m_head;
    while (p != nullptr && p->m_data != val)
        p = p->m_next;
    return p;
}
```

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

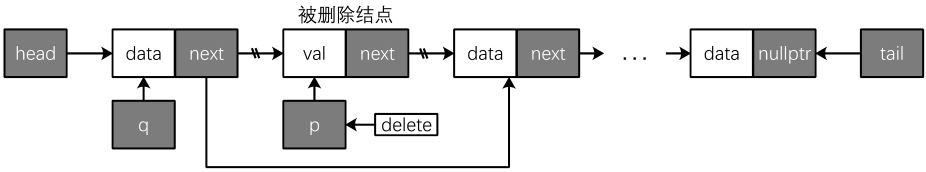
8.3.3 删除操作

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;          p = p->m_next;
    }
    if (p) q->m_next = p->m_next;
    if (p == m_tail) m_tail = q;
    if (p == m_head && p) m_head = p->m_next;
    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

8.3.4 清空链表

clear 函数表头开始，逐个移除每个结点并释放其内存

```
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

clear 函数表头开始，逐个移除每个结点并释放其内存

```
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
}
```

SList 析构函数调用 clear 函数释放链表的内存空间

```
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、友元声明和定义

```
template<typename T> ostream& operator<<(ostream&,const SList<T>&);
template<typename T>
class SList {
    friend ostream& operator<< <T>(ostream&,const SList<T>&);
    //...
};
template<typename T>
ostream& operator<<(ostream &os, const SList<T>& list) {
    Node<T> *p = list.m_head;
    while (p != nullptr) {
        os << p->data() << " "; // 类型T支持<<运算符
        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、友元声明和定义

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

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

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问题?

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问题?

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问题?

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

利用 delete 关键字禁止 Node 类型实例的复制与赋值

```
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

不允许 SList 类型实例的复制与赋值

```
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

不允许 SList 类型实例的复制与赋值

```
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

不允许 SList 类型实例的复制与赋值

```
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

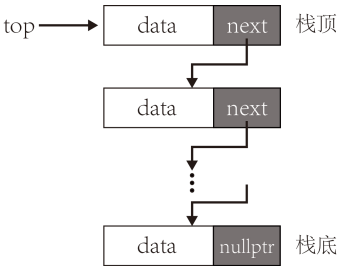
10 20 30

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(Last In First Out,LIFO) 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。

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-push 函数

```
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-push 函数

```
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-pop 函数

```
template<typename T>
void Stack<T>::pop() {
    if (empty()) return;
    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-push 函数

```
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-push 函数

```
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;
    }
}
```

释放内存操作-Stack 析构函数定义

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

说明

利用出栈的操作，逐个释放每个元素的内存空间

8.4.2 简单计算器

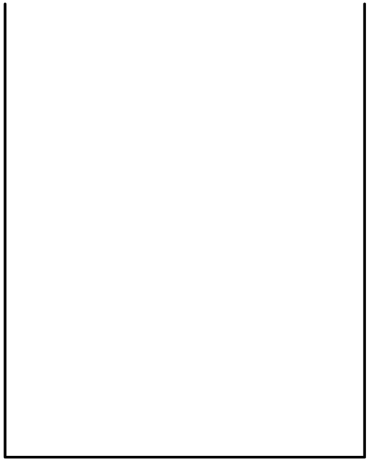
表达式求值是栈的重要应用之一，假设有如下算术表达式：

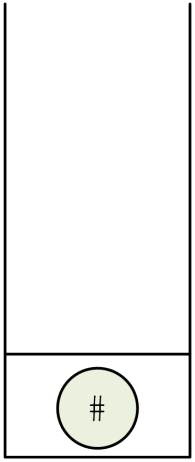
```
a - b / c + d * e =
```

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 =

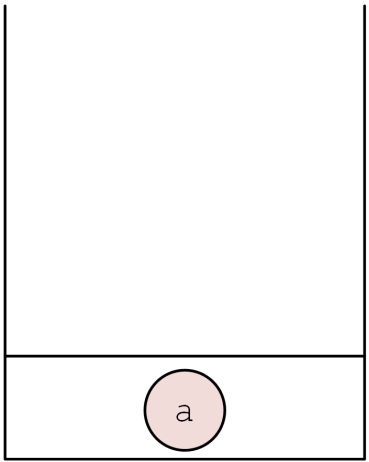


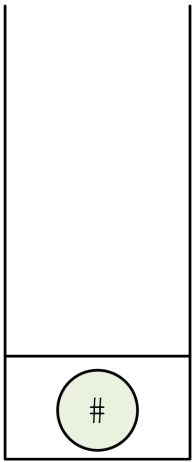


8.4.2 简单计算器

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 =

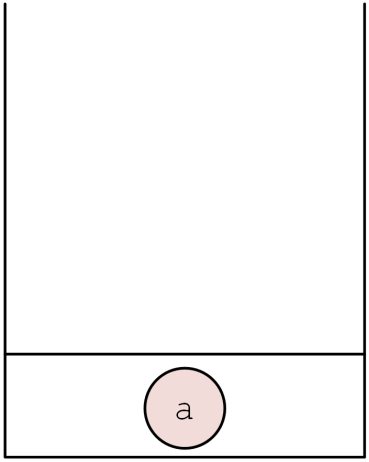


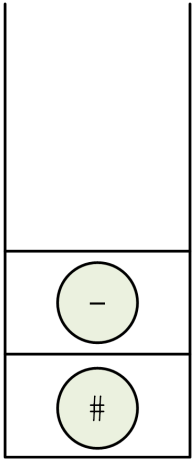


8.4.2 简单计算器

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 =

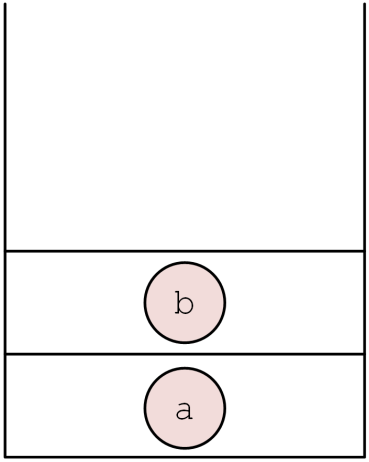


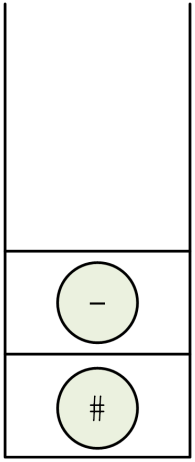


8.4.2 简单计算器

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 =

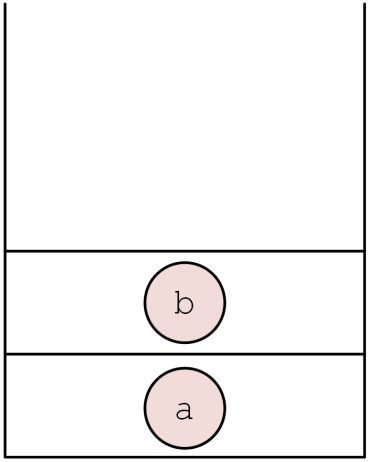


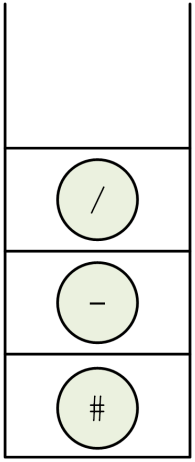


8.4.2 简单计算器

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 =

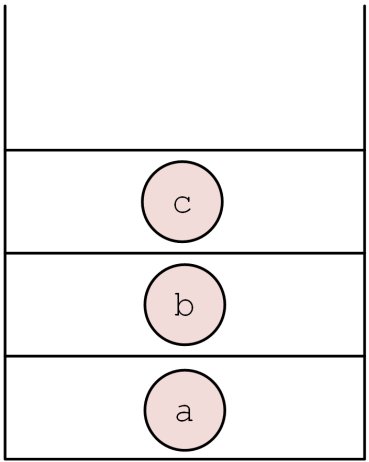


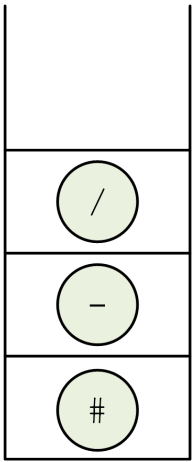


8.4.2 简单计算器

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 =

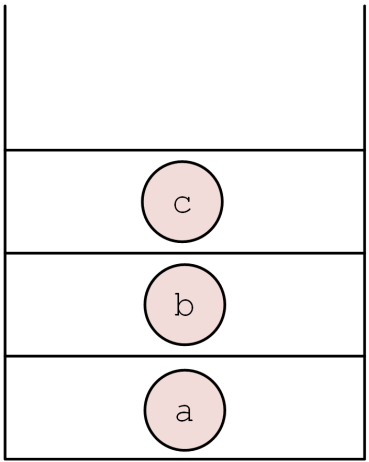


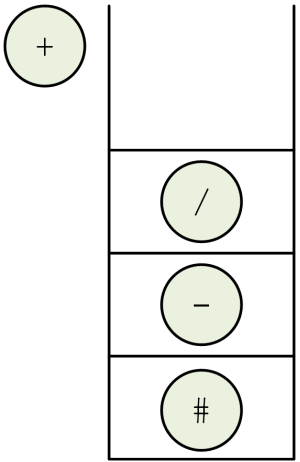


8.4.2 简单计算器

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 =

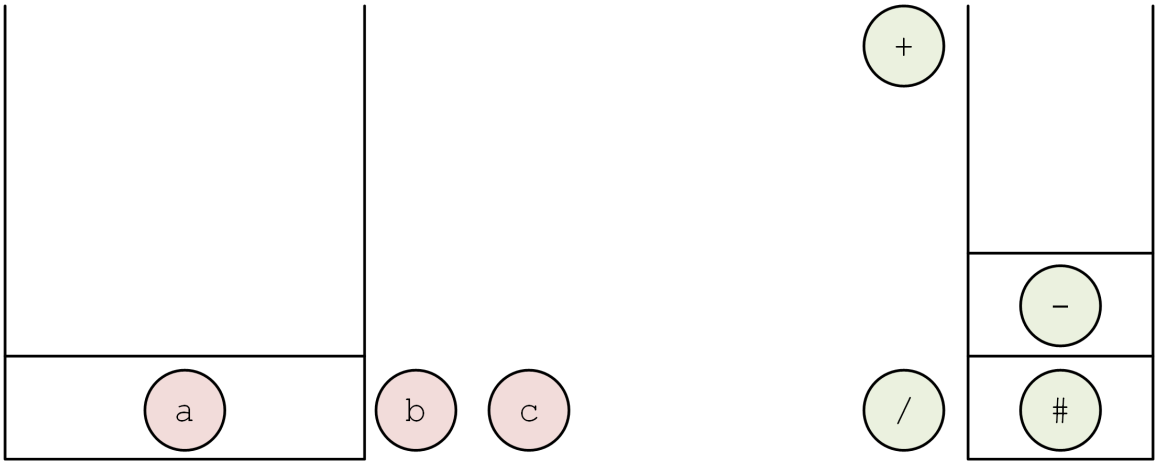




8.4.2 简单计算器

表达式求值是栈的重要应用之一，假设有如下算术表达式：

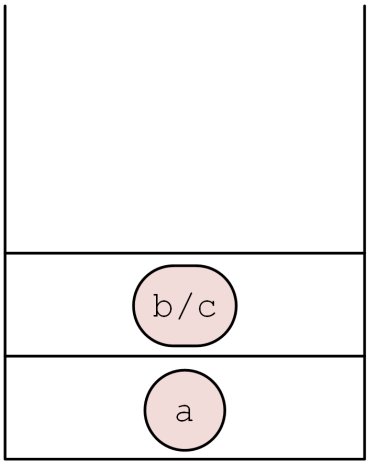
a - b / c + d * e =

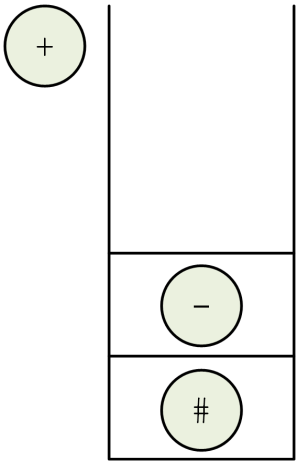


8.4.2 简单计算器

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 =

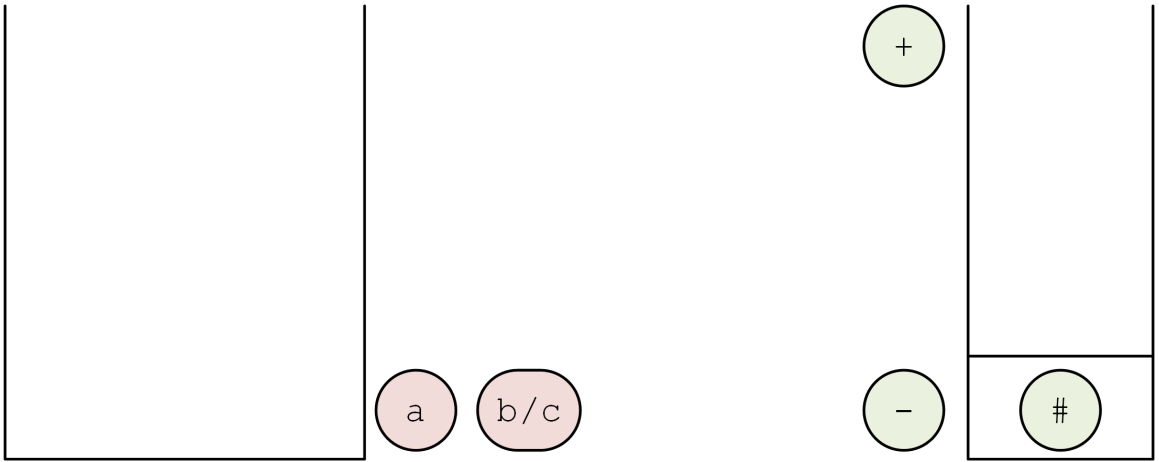




8.4.2 简单计算器

表达式求值是栈的重要应用之一，假设有如下算术表达式：

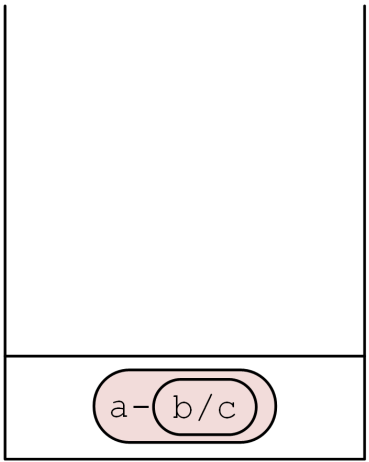
a - b / c + d * e =

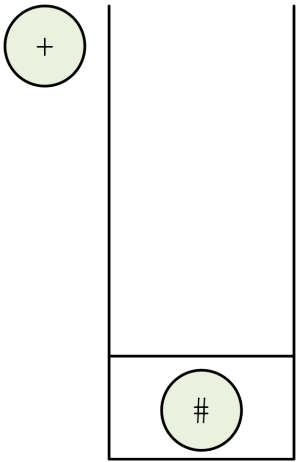


8.4.2 简单计算器

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 =

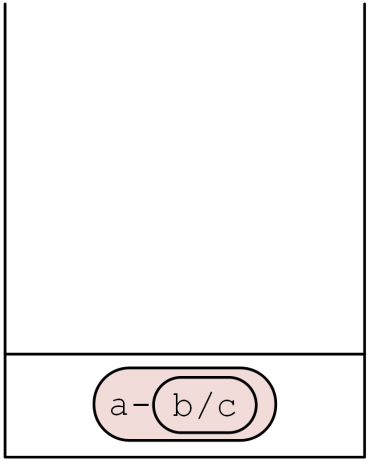


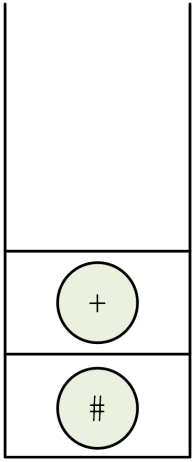


8.4.2 简单计算器

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 =

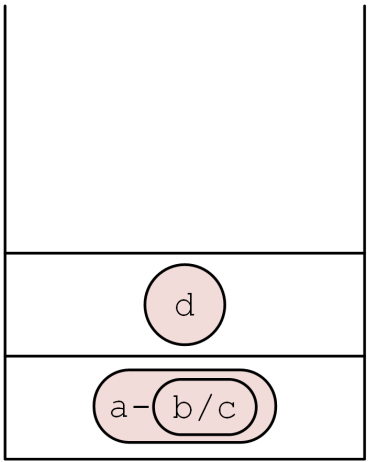


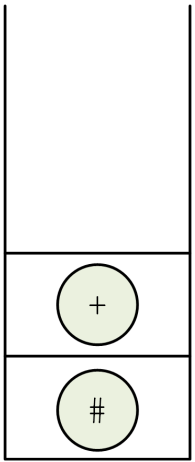


8.4.2 简单计算器

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 =

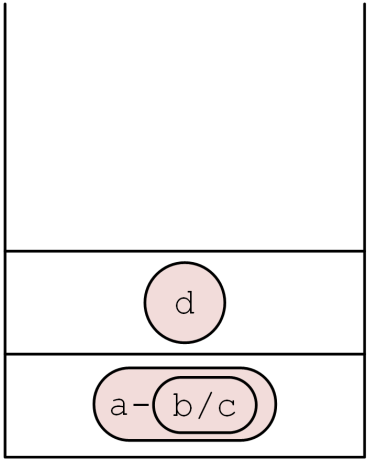


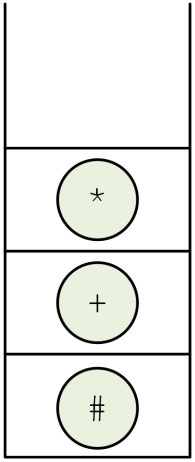


8.4.2 简单计算器

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 =

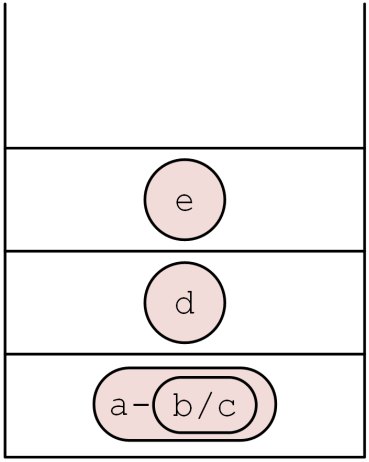


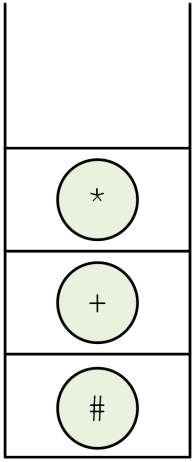


8.4.2 简单计算器

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 =

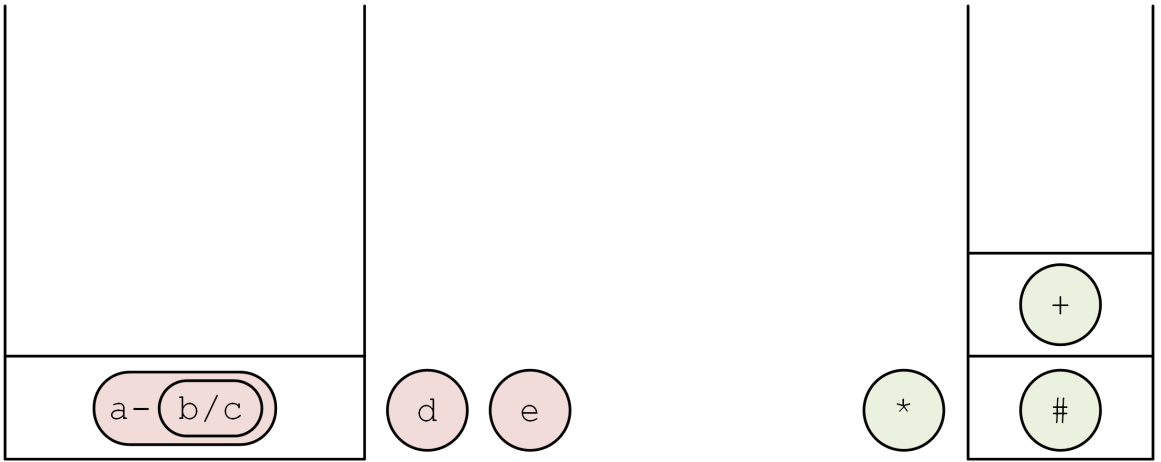




8.4.2 简单计算器

表达式求值是栈的重要应用之一，假设有如下算术表达式：

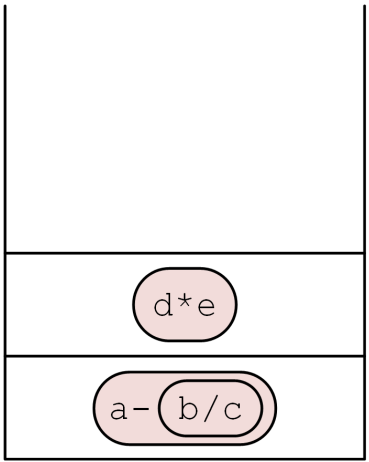
a - b / c + d * e =

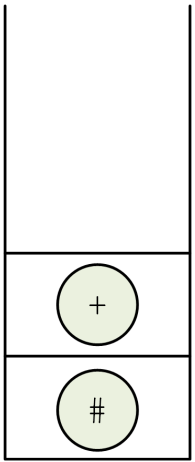


8.4.2 简单计算器

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 =

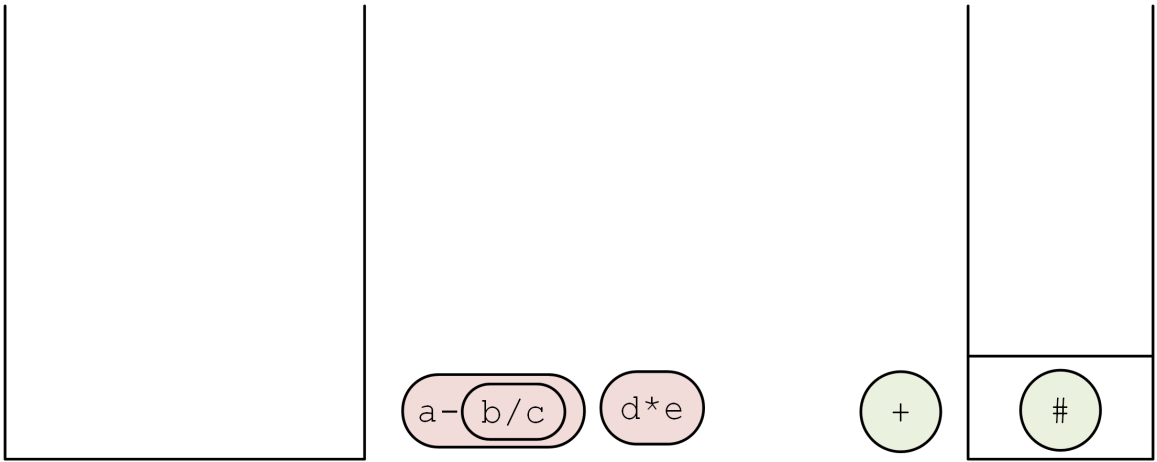




8.4.2 简单计算器

表达式求值是栈的重要应用之一，假设有如下算术表达式：

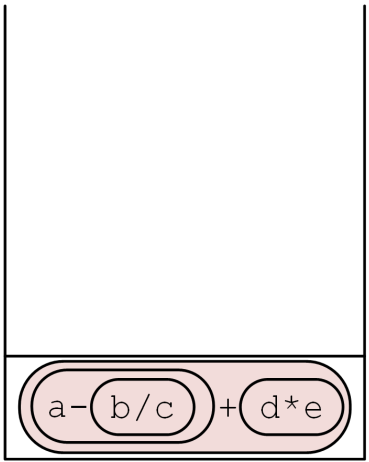
a - b / c + d * e =

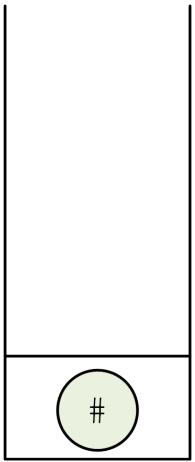


8.4.2 简单计算器

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 =
```

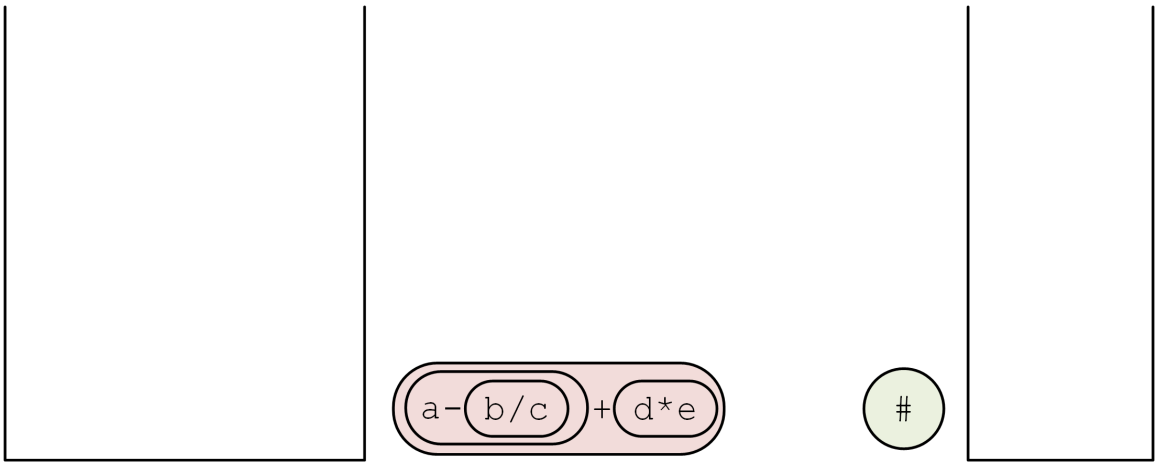




8.4.2 简单计算器

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      bool isNum(string::const_iterator &c) const {  //内联函数，判断是否为数字
9          return* c >= ' 0' && * c <= ' 9'  || * c == ' .' ;
10     }
11 public:
12     Calculator(){ m_opr.push(' #' ); }             //运算符栈初始化
13     double doIt(const string &exp);               //表达式求值
14 };
```

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，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() == ' #' ) break;     //如果运算符栈只剩下#，则计算完毕
54                  calculate();                        //执行栈顶运算符计算
55              }
56              if ( *it != ' =' ) m_opr.push( *it);    //运算符入栈
57              ++it;                                   //继续扫描
58          }
59      }
60      return m_num.top();                         //返回计算结果，注意数据栈此时非空
61  }
```

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=

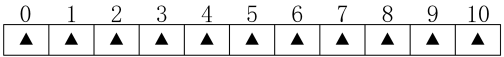
输出结果为：

9-4/2+2.5*2=12

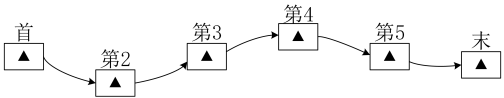
线性结构

每个结点只有一个后继

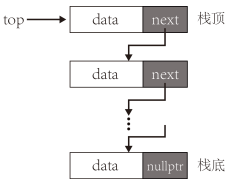
数组：



线性链表：



链栈：

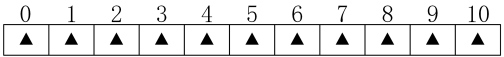


8.5 二叉树

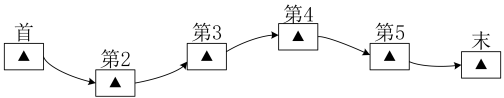
线性结构

每个结点只有一个后继

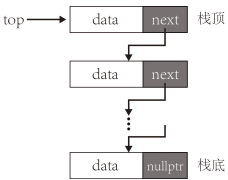
数组：



线性链表：



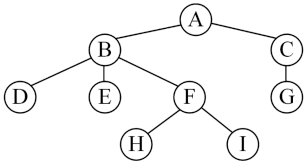
链栈：



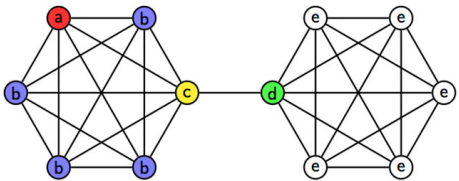
非线性结构

一个结点可能有多个后继多个前驱

树：

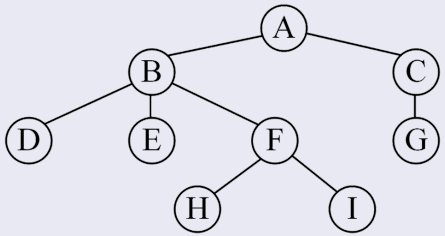


图



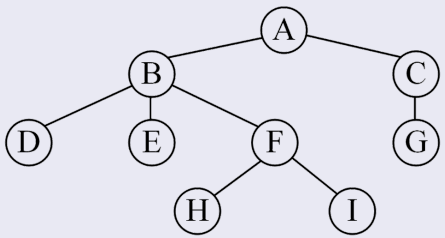
8.5 二叉树

树



8.5 二叉树

树

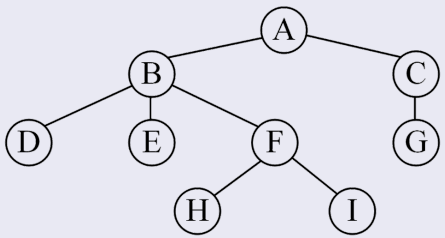


根结点

一棵非空树有且仅有一个根结点

8.5 二叉树

树



根结点

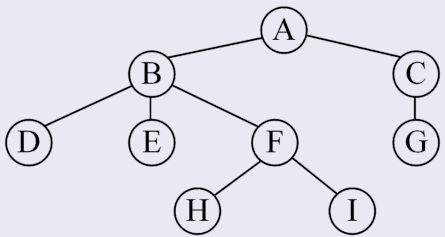
一棵非空树有且仅有一个根结点

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

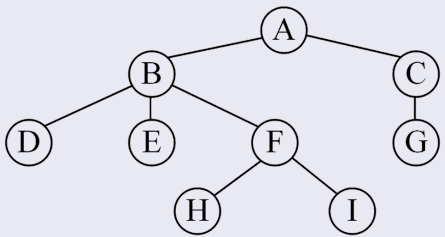
除了根结点外，每个集合互不相交的结点集称为根的子树

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

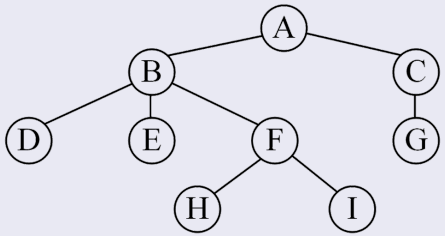
每个结点的子树的数量为该结点的度

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

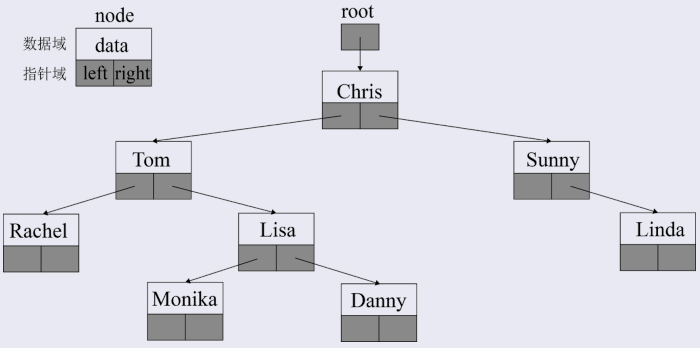
度为 0 的结点称为叶子结点

子结点

每个结点的子树的根结点称为该结点的子结点

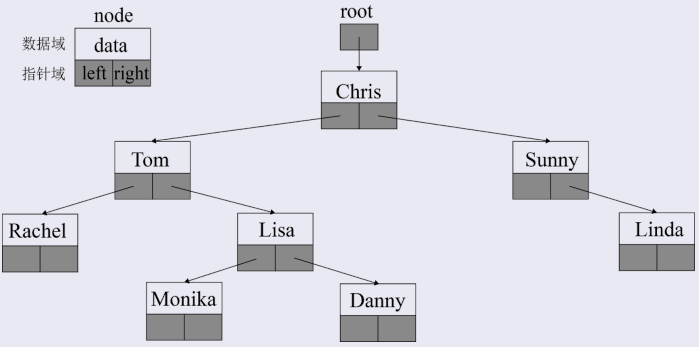
8.5.1 二叉树的概念和表示

二叉树



8.5.1 二叉树的概念和表示

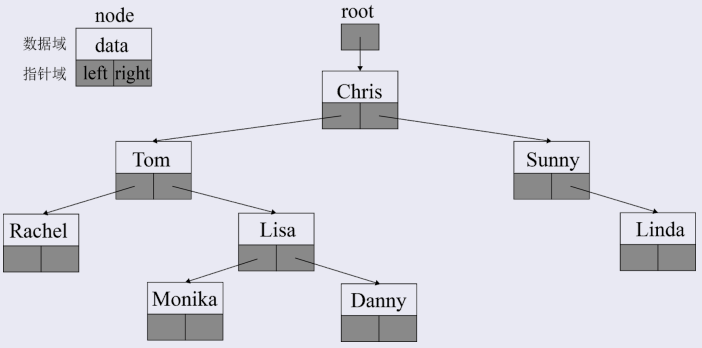
二叉树



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

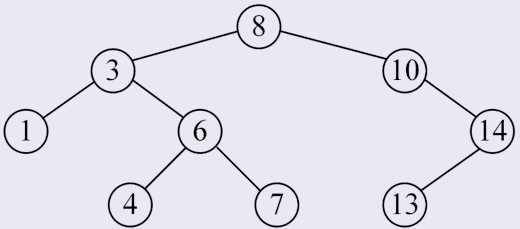
```
template<typename T>
class Node {
private:
    T m_data;
    Node *m_left = nullptr, *m_right = nullptr;
public:
    Node(const T &data):m_data(data){}
    T& data() { return m_data; }
    const T& data() const{ return m_data; }
    Node* left() { return m_left; }
    Node* right() { return m_right;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

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

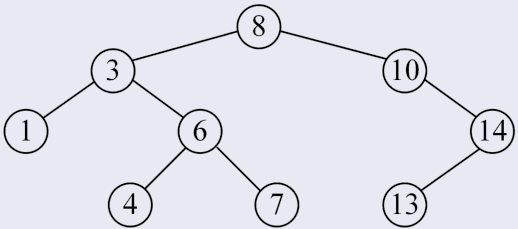
8.5.1 二叉树的概念和表示

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

二叉搜索树类模板，包含插入、遍历、查找、销毁子树等操作

```
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

逐个插入元素来创建二叉搜索树

```
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

逐个插入元素来创建二叉搜索树

```
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

逐个插入元素来创建二叉搜索树

```
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

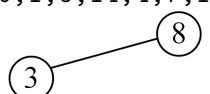
从根结点开始，若待插入结点小于根结点，则在左子树上继续查找插入位置；否则在右子树中查找

8.5.2 创建二叉搜索树

逐个插入元素来创建二叉搜索树

```
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

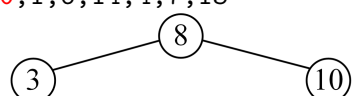
从根结点开始，若待插入结点小于根结点，则在左子树上继续查找插入位置；否则在右子树中查找

8.5.2 创建二叉搜索树

逐个插入元素来创建二叉搜索树

```
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

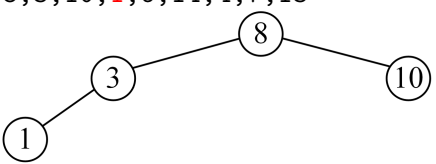
从根结点开始，若待插入结点小于根结点，则在左子树上继续查找插入位置；否则在右子树中查找

8.5.2 创建二叉搜索树

逐个插入元素来创建二叉搜索树

```
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

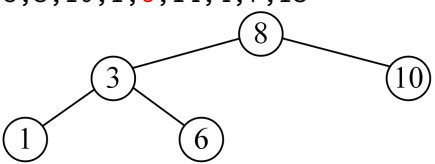
从根结点开始，若待插入结点小于根结点，则在左子树上继续查找插入位置；否则在右子树中查找

8.5.2 创建二叉搜索树

逐个插入元素来创建二叉搜索树

```
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

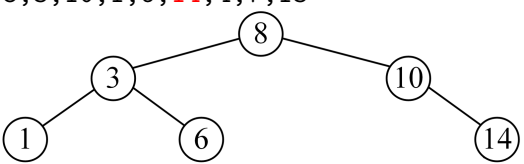
从根结点开始，若待插入结点小于根结点，则在左子树上继续查找插入位置；否则在右子树中查找

8.5.2 创建二叉搜索树

逐个插入元素来创建二叉搜索树

```
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

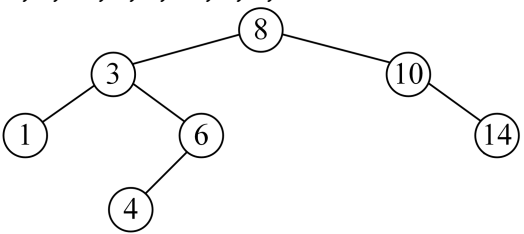
从根结点开始，若待插入结点小于根结点，则在左子树上继续查找插入位置；否则在右子树中查找

8.5.2 创建二叉搜索树

逐个插入元素来创建二叉搜索树

```
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

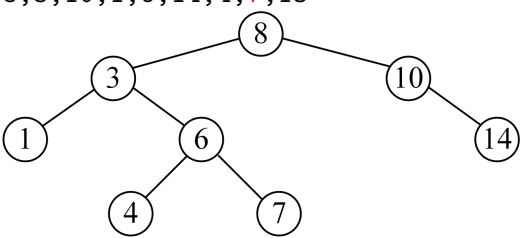
从根结点开始，若待插入结点小于根结点，则在左子树上继续查找插入位置；否则在右子树中查找

8.5.2 创建二叉搜索树

逐个插入元素来创建二叉搜索树

```
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

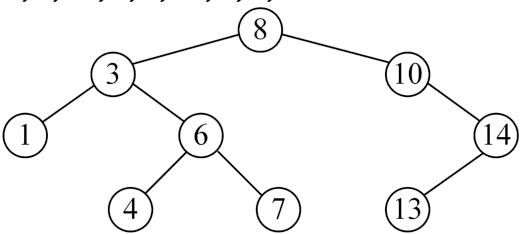
从根结点开始，若待插入结点小于根结点，则在左子树上继续查找插入位置；否则在右子树中查找

8.5.2 创建二叉搜索树

逐个插入元素来创建二叉搜索树

```
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

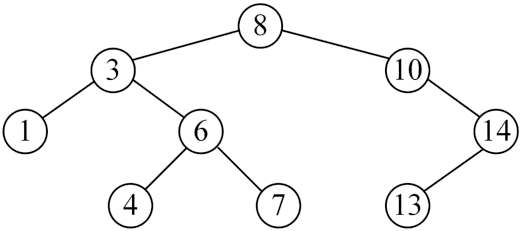
从根结点开始，若待插入结点小于根结点，则在左子树上继续查找插入位置；否则在右子树中查找

8.5.2 创建二叉搜索树

逐个插入元素来创建二叉搜索树

```
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_

```
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)
        return insert_(p->m_left, value); //在左子树中查找
    else
        return insert_(p->m_right, value); //在右子树中查找
}
```

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_

```
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)
        return insert_(p->m_left, value); //在左子树中查找
    else
        return insert_(p->m_right, value); //在右子树中查找
}
```

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_

```
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)
        return insert_(p->m_left, value); //在左子树中查找
    else
        return insert_(p->m_right, value); //在右子树中查找
}
```

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

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

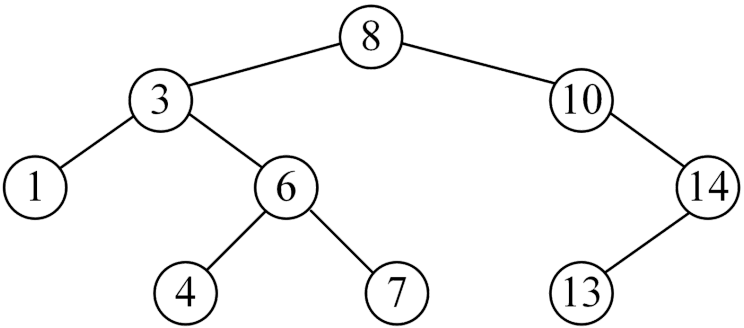
8.5.3 遍历操作

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

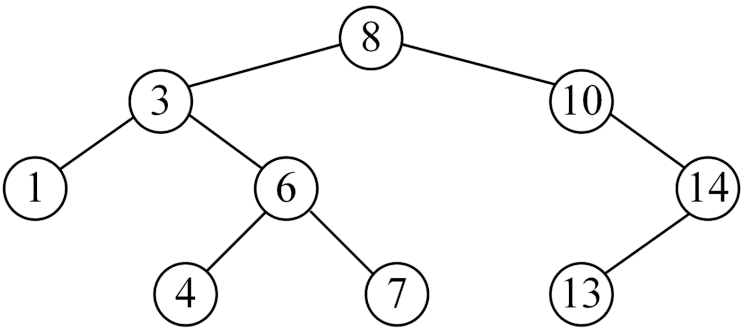
根结点–> 左子树–> 右子树



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

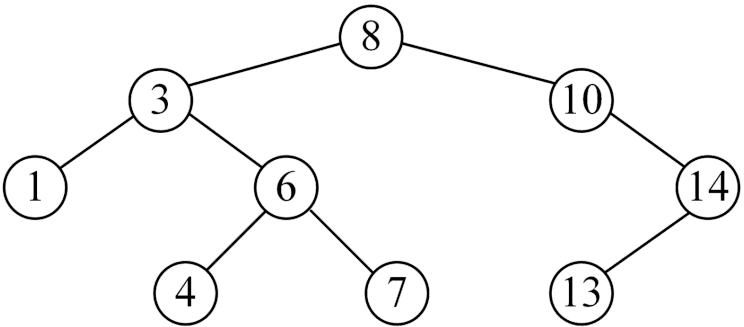
根结点–> 左子树–> 右子树

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

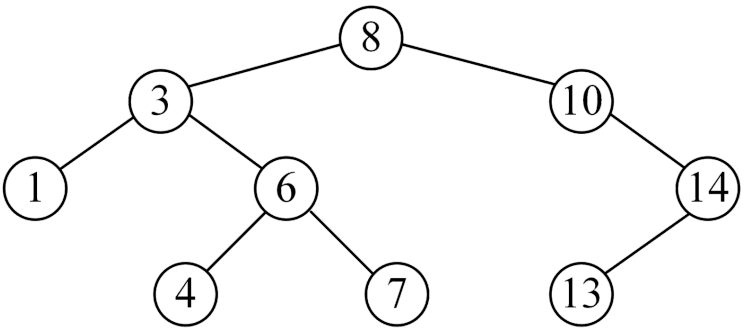
根结点–> 左子树–> 右子树

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

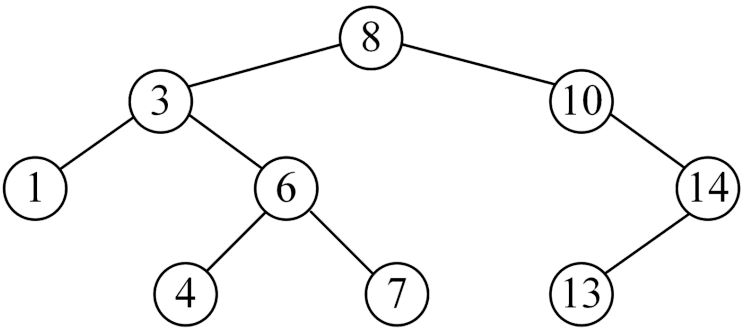
根结点–> 左子树–> 右子树

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

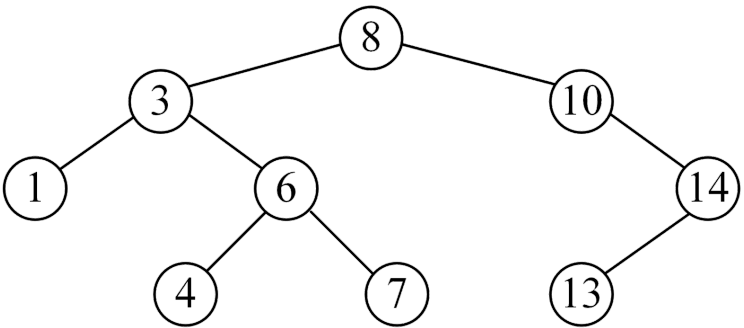
根结点–> 左子树–> 右子树

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

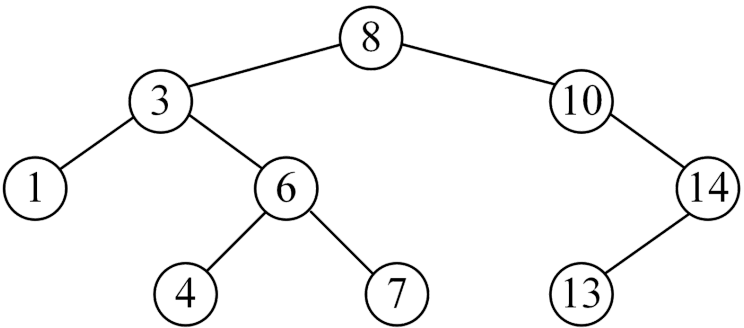
根结点–> 左子树–> 右子树

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

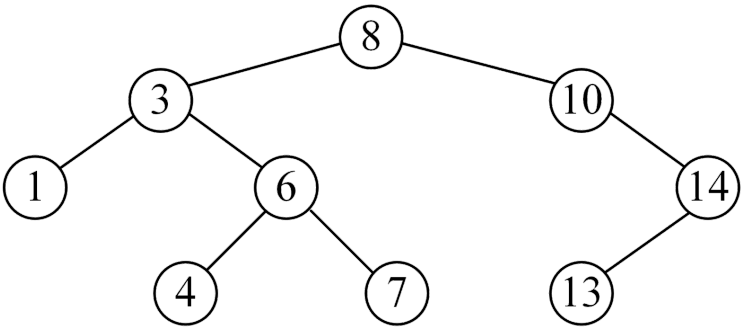
根结点–> 左子树–> 右子树

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

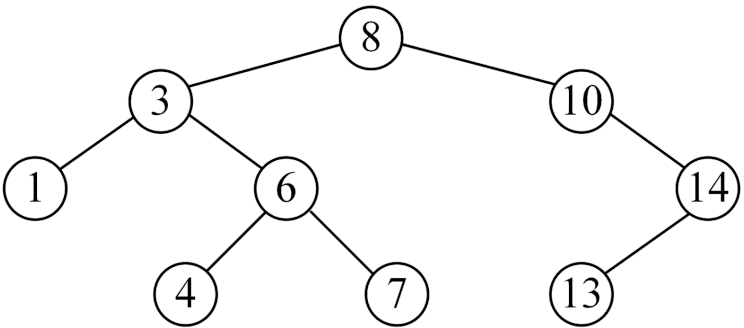
根结点–> 左子树–> 右子树

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

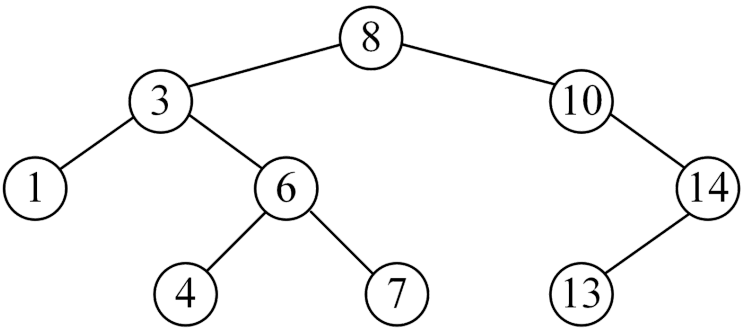
根结点–> 左子树–> 右子树

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

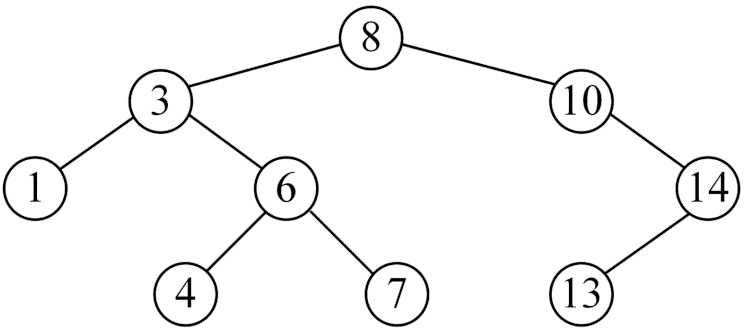
根结点–> 左子树–> 右子树

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

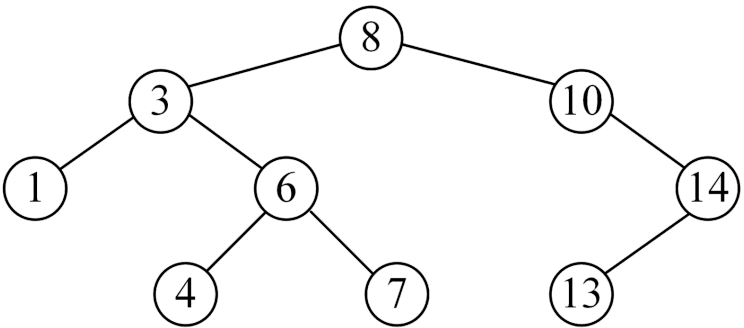
中序遍历

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

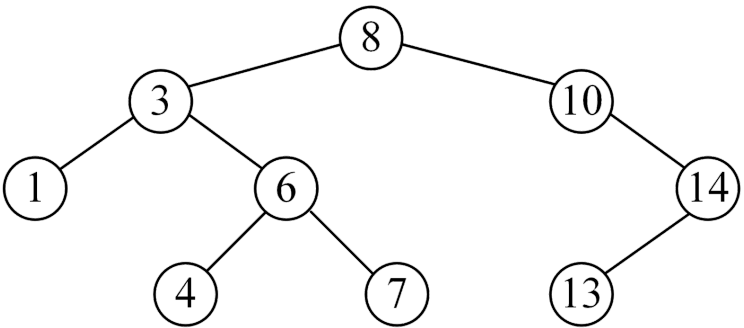
左子树->根结点-> 右子树

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

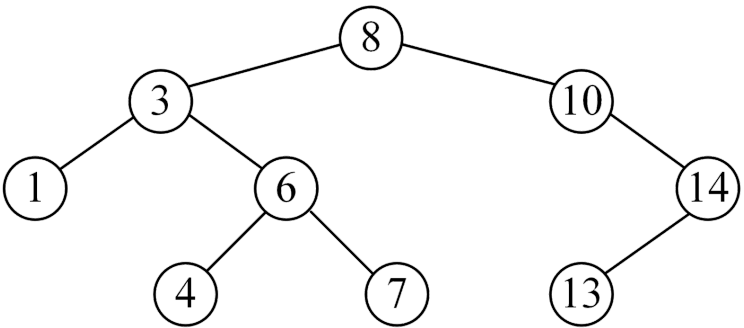
左子树->根结点-> 右子树

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

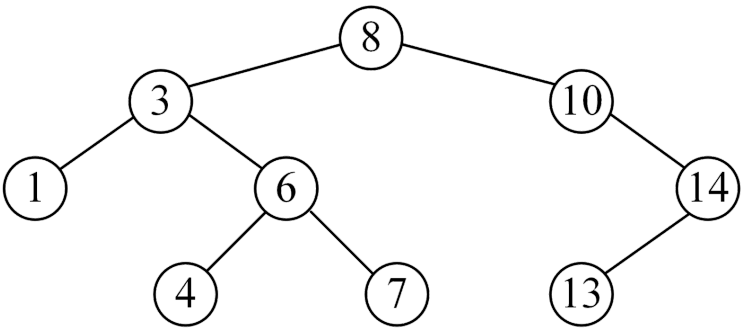
左子树->根结点-> 右子树

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

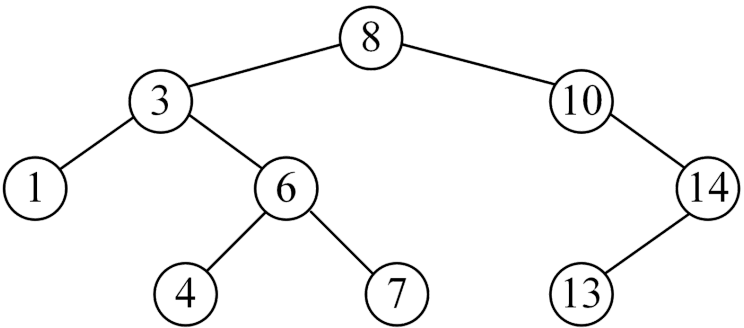
左子树->根结点-> 右子树

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

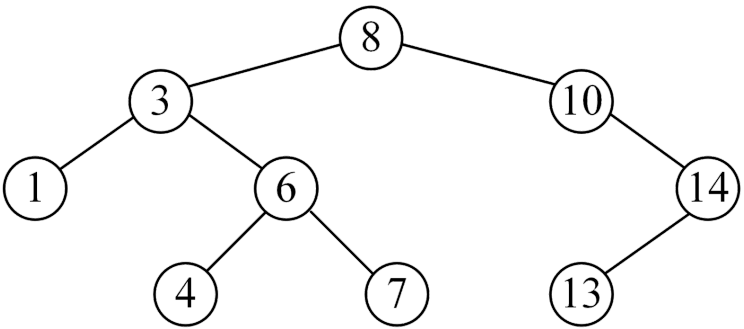
左子树->根结点-> 右子树

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

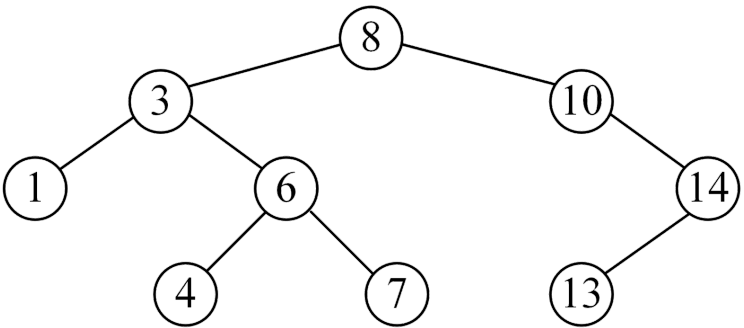
左子树->根结点-> 右子树

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

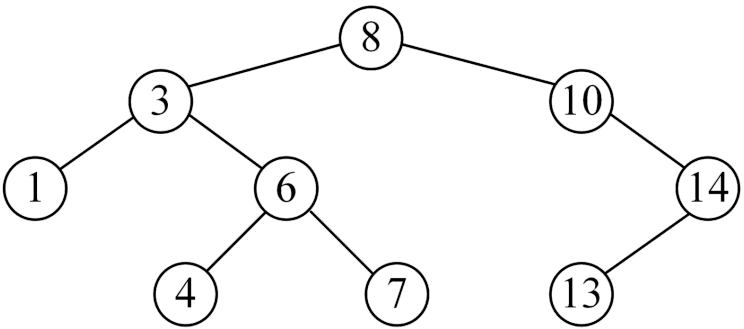
左子树->根结点-> 右子树

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

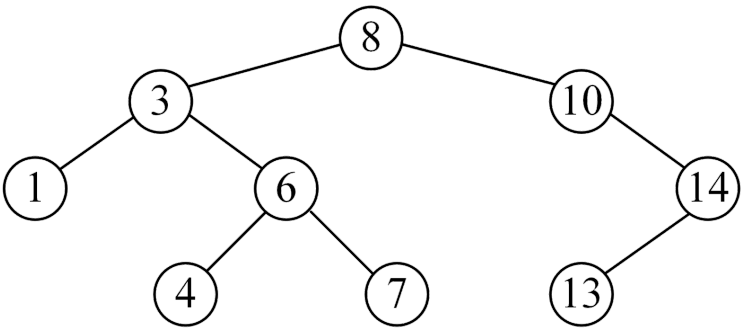
左子树->根结点-> 右子树

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

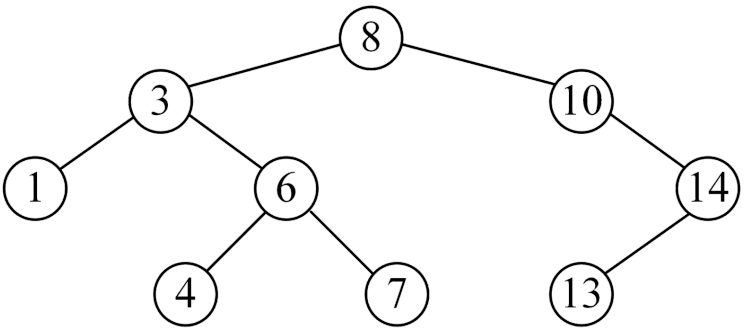
左子树->根结点-> 右子树

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

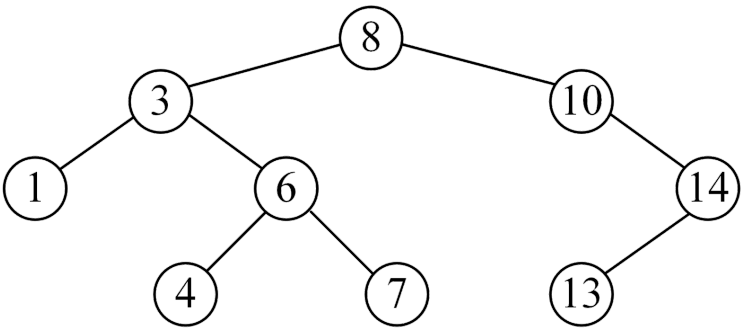
1 3 4 6 7 8 10 13 14

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

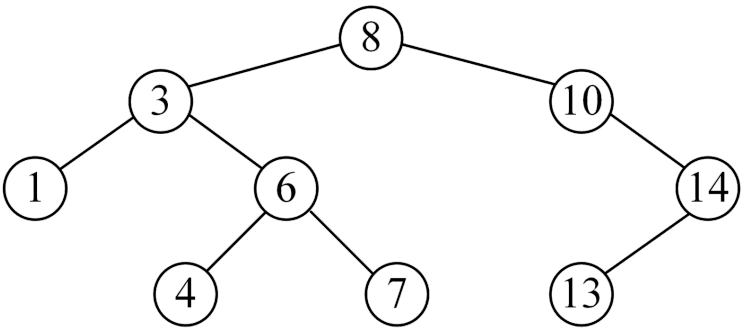
后序遍历

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

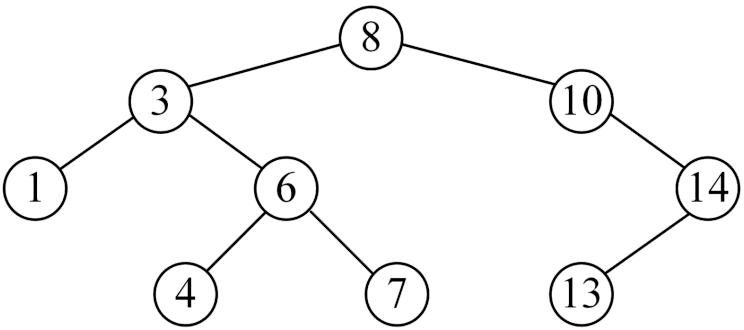
左子树-> 右子树->根结点

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

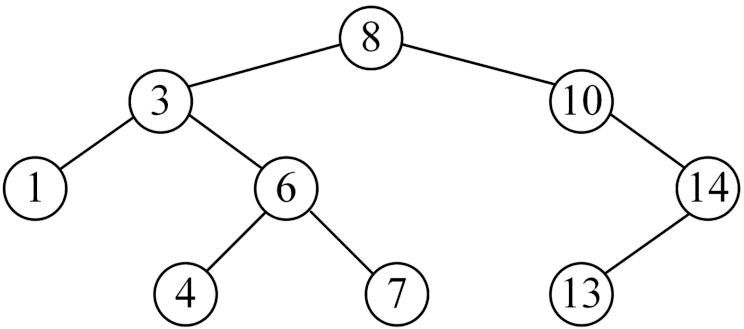
左子树-> 右子树->根结点

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

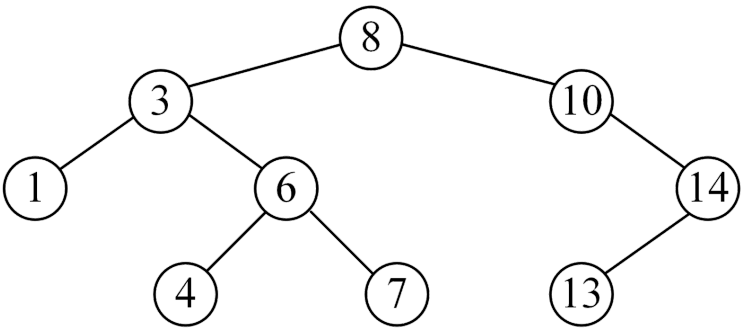
左子树-> 右子树->根结点

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

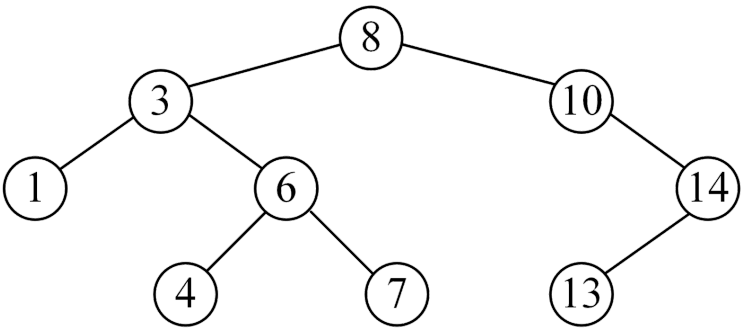
左子树-> 右子树->根结点

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

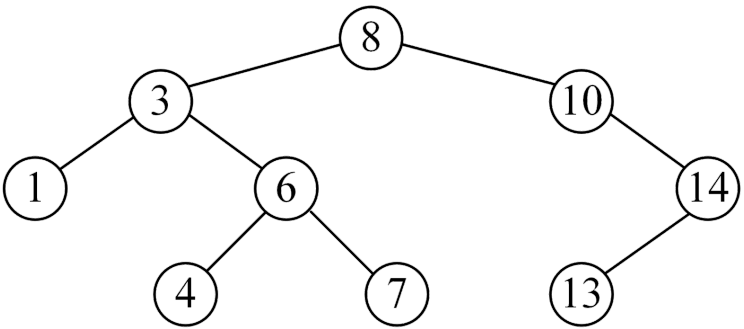
左子树-> 右子树->根结点

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

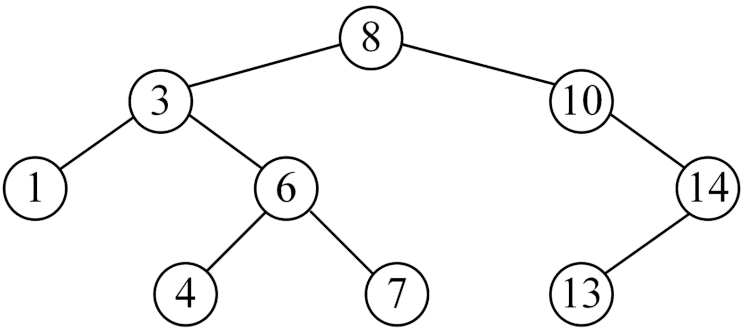
左子树-> 右子树->根结点

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

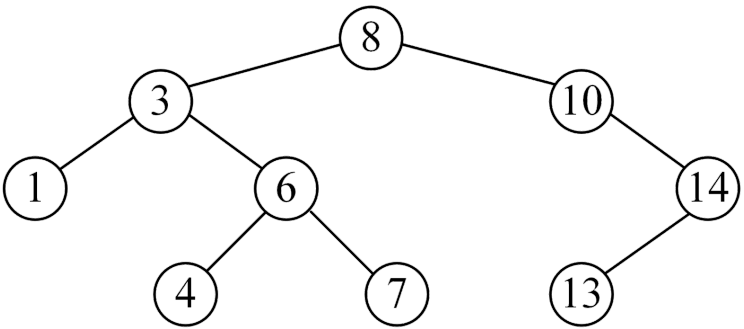
左子树-> 右子树->根结点

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

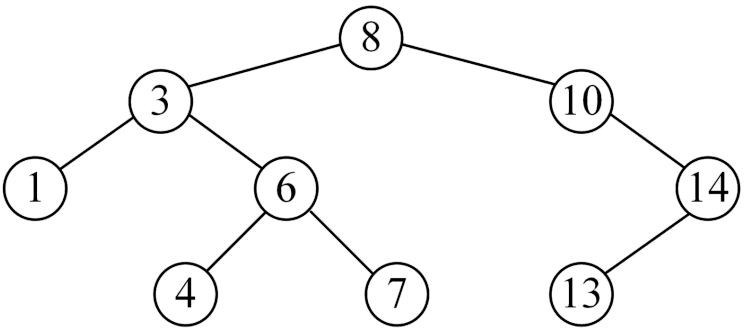
左子树-> 右子树->根结点

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

```
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

```
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

```
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

```
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

```
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;
```

说明

同时将 BinaryTree 类模板声明为 Node 类模板的友元

本章结束

上机作业

1. 实验指导书：第八章
2. 检查日期：待定
3. 地点：与助教协商
4. 特别说明：第三道题目二叉树的应用：哈夫曼编码