**数据：**数据是能输入到计算机中并由计算机程序识别和处理的符号。

数据元素：数据元素是数据的基本单位，在计算机程序中通常作为一个整体进行处理和考虑。数据元素的最小单位是数据项，并且数据元素通常具有相同个数和类型的数据项。

数据结构：数据结构是指相互之间存在一定关系的数据元素的集合。

数据的逻辑结构：数据的逻辑结构是指数据元素以及数据元素之间的逻辑关系，是从实际问题中抽象出的数据模型。

**逻辑结构包括：**

集合结构：数据元素之间就是“属于同一个集合”，除此以外没有其他关系。

线性结构：数据元素之间存在着一对一的线性结

树结构：数据元素之间存在着一对多的层次结构

图结构：数据元素之间存在着多对多的任意结构

数据的存储结构：数据的存储结构也称数据的物理结构，是数据元素的逻辑结构在计算机中的表现。

存储结构包括：顺序存储结构、链式存储结构，循环存储结构

抽象数据模型：抽象数据模型是一个数据模型以及定义在该模型上的一组操作的总称。

**算法：**算法是对特定问题求解步骤的一种描述，是指令的有限序列。

有穷性：一个算法总是（对任何合法的输入）在执行有穷步之后结束，且每一步都在有穷时间内完成。

确定性：算法中的每一条指令必须有确切的含义，不存在二义性，并且，在任何条件下，对于相同的输入只能得到相同的输出。

可行性：描述算法的每一条指令可以转换为某种程序设计语言对应的语句，并在计算机上可以执行。

**好的算法应该满足的条件**

正确性：算法能满足具体问题的需求，即对于任何合法的输入，算法都会得到正确的结果。

健壮性：算法对非法输入的抵抗能力，即对于错误的输入，算法应能识别并作出处理，而不是产生错误动作或陷入瘫痪。

可理解性：算法容易理解和实现，算法首先是为了人的阅读和交流，其次是为了程序的实现，因此，算法要易于被人理解、易于转换为程序。

抽象分级：算法是由人来阅读、理解、使用和修改，研究发现，对大多数人来说，认识限度是7±2，如果算法涉及的步骤太多，人就会糊涂，因此必须用抽象分级来组织算法表达的思想。

高效性：算法的效率包括时间效率和空间效率，时间效率显示了算法运行的有多快，而空间效率则显示了算法需要多少额外的存储空间。不言而喻，一个好的算法应该具有较短的执行时间且占用较少的辅助空间。

**算法分析**

时间复杂度：算法中基本语句的执行次数在渐进意义下的阶，称作算法的渐进时间复杂度，简称时间复杂度，通常用大O记号表示。

常见的时间复杂度如下：O(log₂n)<O(n)<O(nlog₂n)<O(n²)<O(n³)<……<O(2ⁿ)<O(n!)

算法的空间复杂都是指算法在执行过程中需要的辅助空间数量，也就是除算法本身和输入输出数据所占空间外，算法临时开辟的存储空间。

递归算法：对递归算法时间复杂度的分析，关键是根据递归过程建立递推关系式，然后求解这个递推关系式。通常用扩展递归技术将递推关系式中的中等式右边的项根据递推式进行替换，这成为扩展，扩展后的项被再次扩展，依此下去，就会得到一个求和表达式。

最好最快和平均情况：对于某些算法，即使问题规模相同，如果输入数据不同，其时间开销也不同。此时，需要分析最好罪化以及平均情况的时间性能。

**线性表：**线性表简称表，是n(n≥0)个数据元素的有限序列，线性表中数据元素的个数称为线性表的长度。

线性表的数据元素：线性表的数据元素具有抽象（即不确定）的数据类型，在实际问题中，数据元素的抽象类型将被具体的数据类型所取代。

**随机存储结构：**只要确定了存储顺序表的起始地址（即基地址），计算任意一个元素的存储地址的时间是相等的，具有这一特点的存储结构称为随机存取结构。

**顺序表的实现**

const int MaxSize = 100;

template <typename DataType>

class SeqList

{

public:

SeqList( );

SeqList(DataType a[ ], int n);

~SeqList( ); int Length( );

DataType Get(int i);

int Locate(DataType x );

void Insert(int i, DataType x);

DtaType Delete(int i);

int Empty( );

int Empty( );

void PrintList( );

private:

DataType data[MaxSize];

int length;

};

无参构造函数

SeqList :: SeqList( )

{

Length = 0;

}

有参构造函数

SeqList :: SeqList (DataType a[ ], int n)

{

if (n > MaxSize) throw “参数非法”;

for (int i = 0; i < n; i++)

data[ i] = a[ i];

length = n;

}

判空

int SeqList :: Empty( )

{

if (length == 0)

return 1;

else

return 0;

}

长度

int SeqList :: Length( )

{

return length ;

}

按位查找

template <typename DataType>

SeqList :: Get(int i)

{

if (i < 1 && i > length) throw "查找位置非法";

else

return data[i - 1];

}

按值查找

template <typename DataType>

int SeqList :: Locate(DataType x)

{

for (int i = 0; i < length; i++)

if (data[i] == x) return i+1; //返回其序号i+1

return 0; //退出循环，说明查找失败

}

插入

template <typename DataType>

void SeqList :: Insert(int i, DataType x)

{

if (length == MaxSize) throw "上溢";

if (i < 1 || i > length + 1) throw "插入位置错误";

for (int j = length; j >= i; j--)

data[j] = data[j - 1]; //第j个元素存在数组下标为j-1处

data[i - 1] = x;

length++;

}

删除

template <typename DataType>

void SeqList :: Insert(int i, DataType x)

{

if (length == MaxSize) throw "下溢";

if (i < 1 || i > length + 1) throw "删除位置错误";

x=data[i-1]

for (int j = i ; j >= i; j--)

data[j - 1] = data[j]; //第j个元素存在数组下标为j-1处

length--;

return x;

}

最好情况（ i = n+1）：执行0次，时间复杂度为O(1)

最坏情况（ i = 1）：执行n+1次，时间复杂度为O(n)

平均情况（1 ≤ i ≤ n+1）：时间复杂度为O(n)

单链表：单链表是一组用任意的存储单元存放线性表的元素，这组存住单元可以连续也可以不连续，甚至可以零散分布在内存的任意位置。单链表通过每个结点的指针域将线性表的数据元素按其逻辑次序链接在一起，由于每个节点只有一个指针域，故称为单链表。

头指针：第一个元素无前驱，所以设头指针指向第一个元素所在结点

尾标志：由于最后一个元素无后继，故最后一个元素所在节点的指针域为空，这个空指针称为尾标志。

头结点：通常在单链表的开始结点之前附设一个类型相同的节点，称为头结点

单链表的实现：

template <typename DataType>

struct Node

{

DataType data; //数据域

Node<DataType> \*next; //指针域

};

template <typename DataType>

class LinkList

{

public:

LinkList( ); //无参构造函数，建立只有头结点的空链表

LinkList(DataType a[ ], int n); //有参构造函数，建立有n个元素的单链表

~LinkList( ); //析构函数

int Length( ); //求单链表的长度

int Empety();

DataType Get(int i); //按位查找。查找第i个结点的元素值

int Locate(DataType x); //按值查找。查找值为x的元素序号

void Insert(int i, DataType x); //插入操作，第i个位置插入值为x的结点

DataType Delete(int i); //删除操作，删除第i个结点

void PrintList( ); //遍历操作，按序号依次输出各元素

private:

Node<DataType> \*first; //单链表的头指针

};

无参构造函数

template <typename DataType>

LinkList<DataType> :: LinkList( )

{

first = new Node<DataType>; //生成头结点

first->next = nullptr; //头结点的指针域置空

}

判空

template <typename DataType>

int LinkList<DataType> :: Empety()

{

if(first->next == nullptr)

return 1;

else

return 0;

}

遍历

template <typename DataType>

void LinkList<DataType> :: PrintList( )

{

Node<DataType> \*p = first->next; //工作指针p初始化

while (p != nullptr)

{

cout << p->data << "\t";

p = p->next; //工作指针p后移，注意不能写作p++

}

}

求长度

template <typename DataType>

int LinkList<DataType> :: Length( )

{

Node<DataType> \*p = first->next; //工作指针p初始化为开始接点

int count = 0; //累加器count初始化

while (p != nullptr)

{

p = p->next;

count++;

}

return count; //注意count的初始化和返回值之间的关系

}

按位查找

template <typename DataType>

DataType LinkList<DataType> :: Get(int i)

{

Node<DataType> \*p = first->next; //工作指针p初始化

int count = 1; //累加器count初始化

while (p != nullptr && count < i)

{

p = p->next; //工作指针p后移

count++;

}

if (p == nullptr) throw "位置";

else return p->data;

}

按值查找

template <typename DataType>

int LinkList<DataType> :: Locate(DataType x)

{

Node<DataType> \*p = first->next; //工作指针p初始化

int count = 1; //累加器count初始化

while (p != nullptr)

{

if (p->data == x) return count; //查找成功，结束函数并返回序号

p = p->next;

count++;

}

return 0; //退出循环表明查找失败

}

插入操作

template <typename DataType>

void LinkList<DataType> :: Insert(int i, DataType x)

{

Node<DataType> \*p = first, \*s = nullptr ; //工作指针p初始化

int count = 0;

while (p != nullptr && count < i - 1) //查找第i – 1个结点

{

p = p->next; //工作指针p后移

count++;

}

if (p == nullptr) throw "位置"; //没有找到第i – 1个结点

else {

s = new Node<DataType>; s->data = x; //申请结点s，数据域为x

s->next = p->next; p->next = s; //将结点s插入到结点p之后

}

}

头插构造法（构造函数）

template <typename DataType>

LinkList<DataType> :: LinkList(DataType a[ ], int n)

{

first = new Node<DataType>; first->next = nullptr; //初始化一个空链表

for (int i = 0; i < n; i++)

{

Node<DataType> \*s;

s = new Node<DataType>; s->data = a[i];

s->next = first->next; first->next = s; //将结点s插入到头结点之后

}

}

尾插构造法（构造函数）

template <typename DataType>

LinkList<DataType> :: LinkList(DataType a[ ], int n)

{

first = new Node<DataType>; //生成头结点

Node<DataType> \*r = first, \*s = nullptr; //尾指针初始化

for (int i = 0; i < n; i++)

{

s = new Node<DataType>; s->data = a[i];

r->next = s; r = s; //将结点s插入到终端结点之后

}

r->next = nullptr; //单链表建立完毕，将终端结点的指针域置空

}

删除

template <typename DataType>

DataType LinkList<DataType> :: Delete(int i)

{

DataType x;

Node<DataType> \*p = first, \*q = nullptr; //工作指针p指向头结点

int count = 0;

while (p != nullptr && count < i - 1) //查找第i-1个结点

{

p = p->next;

count++;

}

if (p == nullptr || p->next == nullptr) //结点p不存在或p的后继结点不存在

throw "位置";

else {

q = p->next; x = q->data; //暂存被删结点

p->next = q->next; //摘链

delete q;

return x;

}

}

析构函数

template <class DataType>

LinkList<DataType> :: ~LinkList( )

{

Node<DataType> \*q = NULL;

while (first != NULL) //释放单链表的每一个结点的存储空间

{

q = first; //暂存被释放结点

first = first->next; // first指向被释放结点的下一个结点

delete q;

}

}

双链表

插入

template <typename DataType>

void LinkList<DataType> :: Insert(int i, DataType x)

{

Node<DataType> \*p = first, \*s = nullptr ; //工作指针p初始化

int count = 0;

while (p != nullptr && count < i - 1) //查找第i – 1个结点

{

p = p->next; //工作指针p后移

count++;

}

if (p == nullptr) throw "位置"; //没有找到第i – 1个结点

else {

s = new Node<DataType>; s->prior = p;

s->next = p->next;

p->next ->prior = s;

p->next =s;

}

删除

template <typename DataType>

DataType LinkList<DataType> :: Delete(int i)

{

DataType x;

Node<DataType> \*p = first, \*q = nullptr; //工作指针p指向头结点

int count = 0;

while (p != nullptr && count < i - 1) //查找第i-1个结点

{

p = p->next;

count++;

}

if (p == nullptr || p->next == nullptr) //结点p不存在或p的后继结点不存在

throw "位置";

else {

(p->prior)->next=p->next;

(p->next)->prior=p->prior;

delete q;

return x;

}

}

循环单链表：单链表中，如果将终端结点的指针由空指针改为指向头结点，就是整个链表形成一个环，这种头尾相接的单链表称为循环单链表。

**顺序表和链表的比较**

从空间上讲，若线性表中元素个数变化较大或者未知，最好使用链表实现；如果用户事先知道线性表的大致长度，使用顺序表的空间效率会更高

从时间上讲，若线性表频繁查找却很少进行插入和删除操作，或其操作和元素在表中的位置密切相关时，宜采用顺序表作为存储结构；若线性表需频繁插入和删除时，则宜采用链表做存储结构

**栈：**栈是限定仅在表的一端进行插入和删除造作的线性表，允许插入和删除的一端称为栈顶，另一端称为栈底，不含任何数据元素的栈称为空栈。栈中元素除了具有线性关系外，还具有后进先出的特性。

顺序栈的实现：

const int StackSize = 10; //10是示例性的数据，根据实际问题具体定义

template <typename DataType> //定义模板类SeqStack

class SeqStack

{

public:

SeqStack( ); //构造函数，初始化一个空栈

~SeqStack( ); //析构函数

void Push( DataType x ); //入栈操作，将元素x入栈

DataType Pop( ); //出栈操作，将栈顶元素弹出

DataType GetTop( ); //取栈顶元素（并不删除）

int Empty( ); //判断栈是否为空

private:

DataType data[StackSize]; //存放栈元素的数组

int top; //游标，栈顶指针，为栈顶元素在数组中的下标

};

构造函数

template <typename DataType>

SeqStack<DataType> :: SeqStack()

{

top = -1;

}

析构函数

template <typename DataType>

SeqStack<DataType> :: ~SeqStack()

{

}

入栈

template <typename DataType>

void SeqStack<DataType> :: Push(DataType x)

{

if (top == StackSize - 1) throw "上溢";

data[++top] = x;

}

出栈

template <typename DataType>

DataType SeqStack<DataType> :: Pop( )

{

DataType x;

if (top == -1) throw "下溢";

x = data[top--];

return x;

}

判空

template <typename DataType>

int SeqStack<DataType> :: Empty()

{

if(top == -1)

return 1;

else

return 0;

}

取栈顶元素

template <typename DataType>

DataType SeqStack<DataType> :: GetTop( )

{

if(top == -1)

throw "下溢异常";

else

return data[top];

}

template <typename DataType>

struct Node

{

DataType data; //数据域

Node<DataType> \*next; //指针域

};

链栈的实现

template <typename DataType>

class LinkStack

{

public:

LinkStack( ); //构造函数，初始化一个空链栈

~LinkStack( ); //析构函数，释放链栈各结点的存储空间

void Push(DataType x); //入栈操作，将元素x入栈

DataType Pop( ); //出栈操作，将栈顶元素出栈

DataType GetTop( ); //取栈顶元素（并不删除）

int Empty( ); //判空操作，判断链栈是否为空栈

private:

Node<DataType> \*top; //栈顶指针即链栈的头指针

};

构造函数

template <typename DataType>

LinkStack<DataType> :: LinkStack()

{

top = new Node<DataType>; //生成头结点

top->next = nullptr; //头结点的指针域置空

}

析构函数

template <typename DataType>

LinkStack<DataType> :: ~LinkStack()

{

Node<DataType> \*q = nullptr;

while (top != nullptr) //释放链栈的每一个结点的存储空间

{

q = top; //暂存被释放结点

top = top->next; // top指向被释放结点的下一个结点

delete q;

}

}

入栈

template <typename DataType>

void LinkStack<DataType> :: Push(DataType x)

{

Node<DataType> \*s = nullptr;

s = new Node<DataType>; s->data = x; //申请结点s数据域为x

s->next = top; top = s; //将结点s插在栈顶

}

出栈

template <typename DataType>

DataType LinkStack<DataType> :: Pop( )

{

Node<DataType> \*p = nullptr;

DataType x;

if (top == nullptr) throw "下溢";

x = top->data; p = top; //暂存栈顶元素

top = top->next; //将栈顶结点摘链

delete p;

return x;

}

取栈顶元素

template <typename DataType>

DataType LinkStack<DataType> :: GetTop()

{

if(top == nullptr)

throw "下溢异常";

else

return top->data;

}

判空

template <typename DataType>

int LinkStack<DataType> :: Empty()

{

if(top == nullptr)

throw "下溢异常";

else

return 0;

}

对列：队列是指允许在一端进行插入操作，在另一端进行删除操作的线性表，允许插入（也称入队，进队）的一端称为队尾，允许删除的一端称为队头。队列中的元素除了具有线性关系外，还具有先进先出的特性。

**循环队列的存储结构**

当元素被插入到数组中下表最大的位置之后，数组空间就用尽了。尽管此时数组的低端还有空闲空间，这种现象叫作假溢出。解决假溢出的方法就是将存储队列的数组看成是头尾相接的循环结构，即允许队列直接从数组中下标最大的位置延续到下标最小的位置。队列的这种头尾相接的顺序存储结构称为循环队列。

循环队列的实现

const int QueueSize = 100; //100是示例性数据，根据需要重新定义

template <typename DataType> //定义模板类CirQueue

class CirQueue

{

public:

CirQueue( ); //构造函数，初始化空队列

~ CirQueue( ); //析构函数

void EnQueue(DataType x); //入队操作，将元素x入队

DataType DeQueue( ); //出队操作，将队头元素出队

DataType GetQueue( ); //取队头元素（并不删除）

int Empty( ); //判断队列是否为空

private:

DataType data[QueueSize]; //存放队列元素的数组

int front, rear; //游标，队头和队尾指针

};

构造函数

template <typename DataType>

CirQueue<DataType> :: CirQueue()

{

rear = front = QueueSize - 1;

}

析构函数

template <typename DataType>

CirQueue<DataType> :: ~CirQueue()

{

}

入队

template <typename DataType>

void CirQueue<DataType> :: EnQueue(DataType x)

{

if ((rear + 1) % QueueSize == front)

throw "上溢";

rear = (rear + 1) % QueueSize; //队尾指针在循环意义下加1

data[rear] = x; //在队尾处插入元素

}

出队

template <typename DataType>

DataType CirQueue<DataType> :: DeQueue( )

{

if (rear == front) throw "下溢";

front = (front + 1) % QueueSize; //队头指针在循环意义下加1

return data[front]; //读取并返回出队前的队头元素

}

取队头元素

template <typename DataType>

DataType CirQueue<DataType> :: GetQueue( )

{

if (rear == front) throw "下溢";

return data[(front + 1) % QueueSize]; //注意不修改队头指针

}

判空

template <typename DataType>

int CirQueue<DataType> :: Empty()

{

if(front == rear)

return 1;

else

return 0;

}

链队列的实现

template <typename DataType>

struct Node

{

DataType data; //数据域

Node<DataType> \*next; //指针域

};

template <typename DataType>

class LinkQueue

{

public:

LinkQueue( ); //构造函数，初始化一个空的链队列

~LinkQueue( ); //析构函数，释放链队列各结点的存储空间

void EnQueue(DataType x); //入队操作，将元素x入队

DataType DeQueue( ); //出队操作，将队头元素出队

DataType GetQueue( ); //取链队列的队头元素

int Empty( ); //判断链队列是否为空

private:

Node<DataType> \*front, \*rear; //队头和队尾指针，分别指向头结点和终端结点

};

构造函数

template <typename DataType>

LinkQueue<DataType> :: LinkQueue( )

{

Node<DataType> \*s = nullptr;

s = new Node<DataType>; s->next = nullptr; //创建头结点s

front = rear = s; //将队头指针和队尾指针都指向头结点s

}

析构函数

template <typename DataType>

LinkQueue<DataType> :: ~LinkQueue()

{

Node<DataType> \*q = nullptr;

while (front != nullptr) //释放单链表的每一个结点的存储空间

{

q = front; //暂存被释放结点

front = front->next; // front指向被释放结点的下一个结点

delete q;

}

}

入队

template <typename DataType>

void LinkQueue<DataType> :: EnQueue(DataType x)

{

Node<DataType> \*s = nullptr;

s = new Node<DataType>; //申请结点s

s->data = x; s->next = nullptr;

rear->next = s; rear = s; //将结点s插入到队尾

}

出队

template <typename DataType>

DataType LinkQueue<DataType> :: DeQueue( )

{

DataType x;

Node<DataType> \*p = nullptr;

if (rear == front) throw "下溢";

p = front->next; x = p->data; //暂存队头元素

front->next = p->next; //将队头元素所在结点摘链

if (p->next == nullptr) rear = front; //判断出队前队列长度是否为1

delete p;

return x;

}

取队头元素

template <typename DataType>

DataType LinkQueue<DataType> :: GetQueue()

{

if(front == rear)

throw "下溢异常";

else

return front->next->data;

}

判空

template <typename DataType>

int LinkQueue<DataType> :: Empty()

{

if(front == rear)

return 1;

else

return 0;

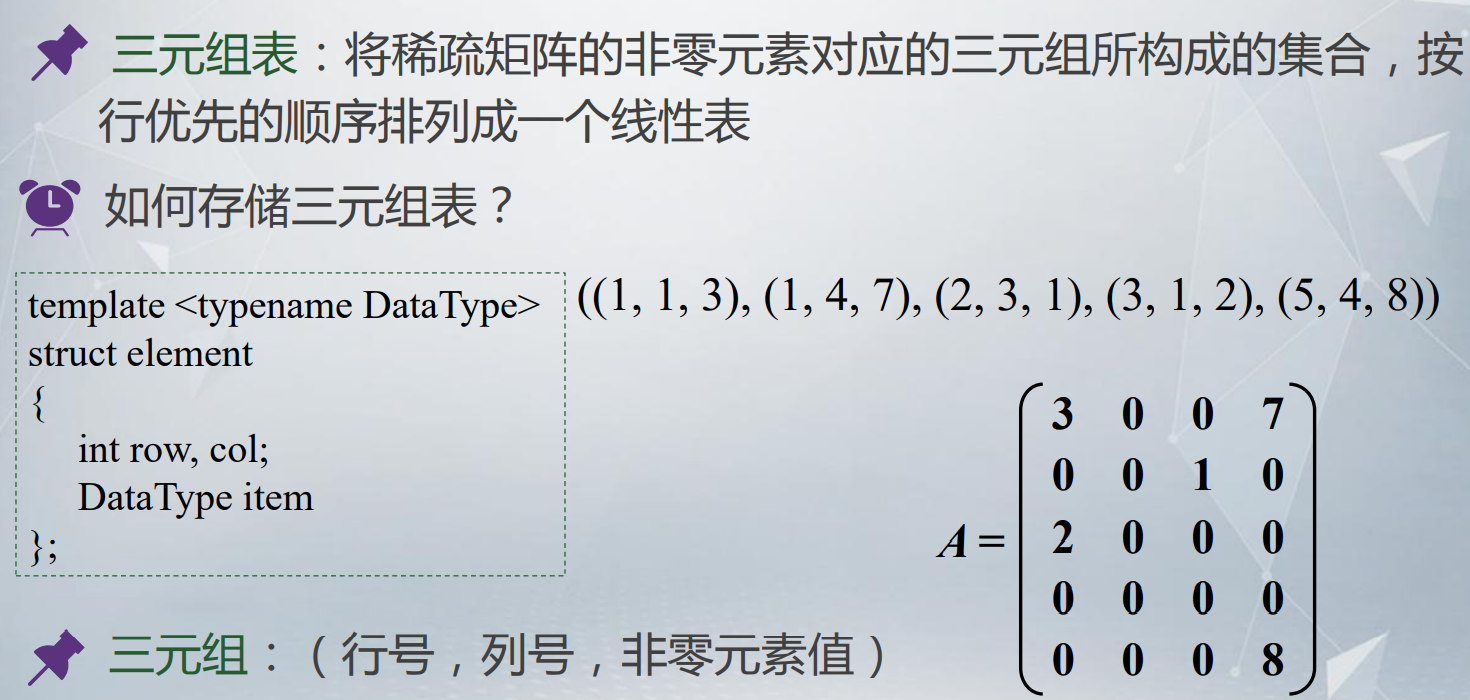
}

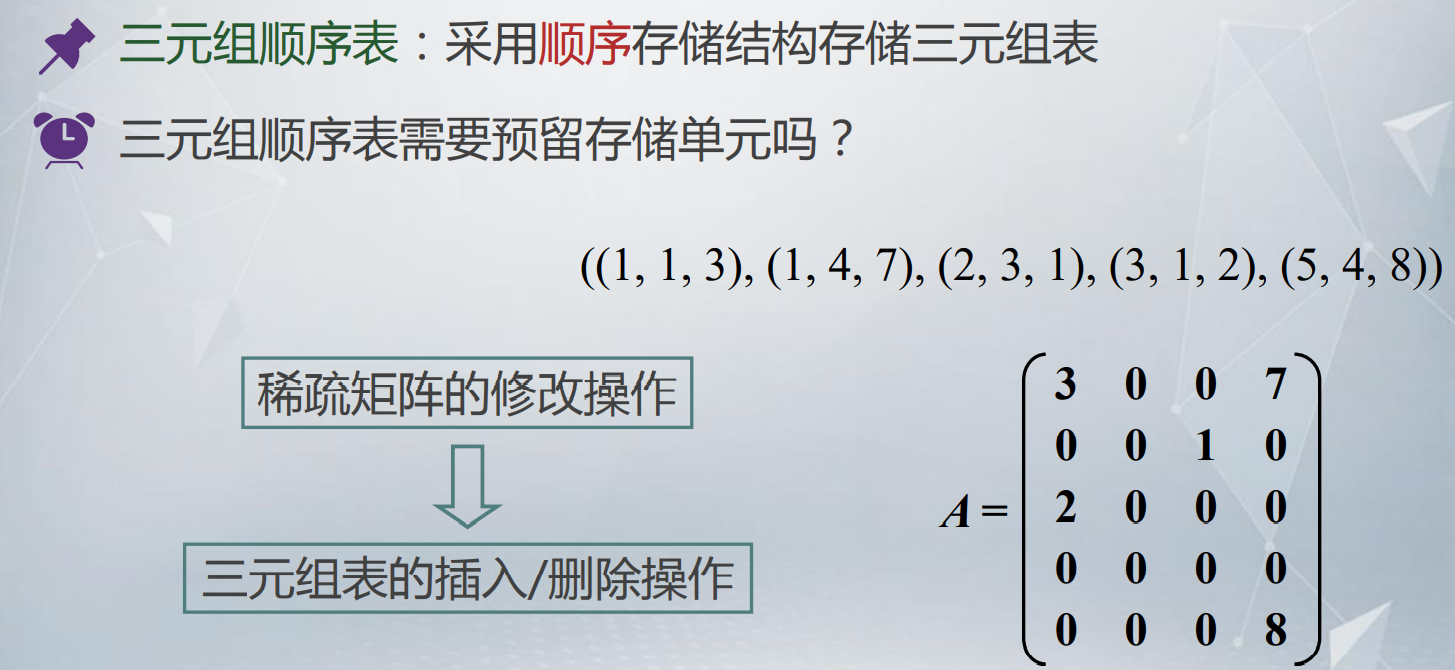
**字符串：**字符串是n(n≥0)个字符组成的有限序列，串中所包含的字符个数称为称为串的长度，字符串中任意个连续的字符组成的子序列称为该串的**子串**。相应地，包含子串的串称为**主串**。子串的第一个字符在主串中的序号称为子串在主串中的**位置**。

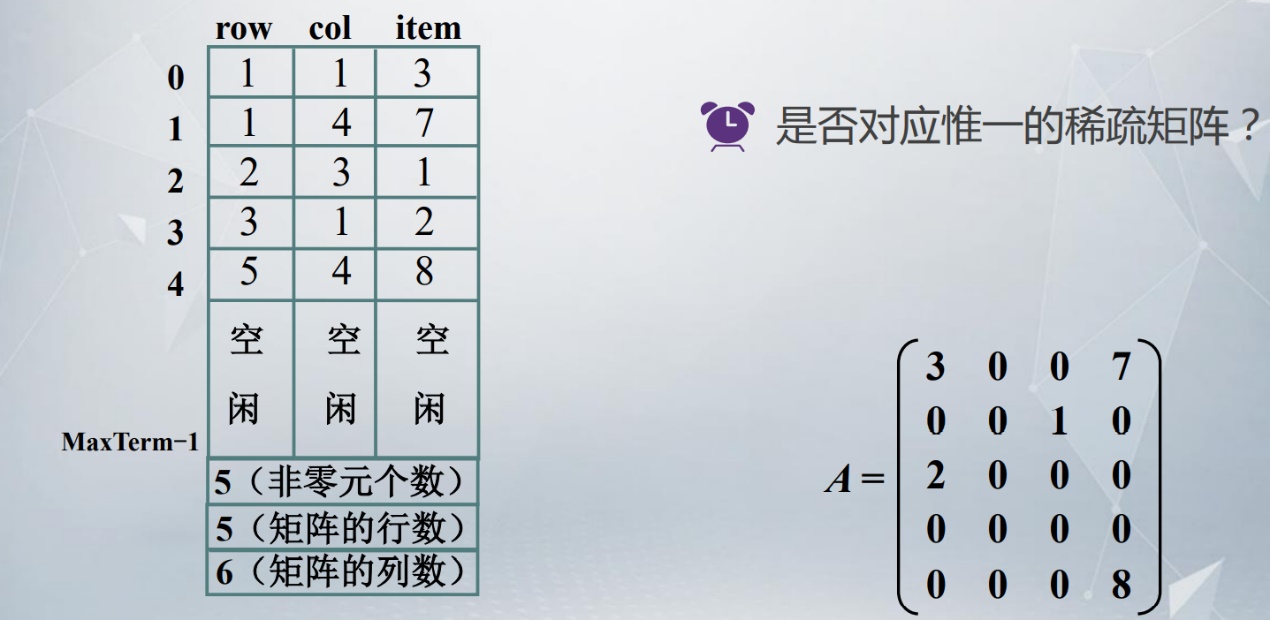
**数组：**数组是由类型相同的数据元素构成的有序集合，每个数据元素称为一个数据元素（简称“元素”），每个元素受n（n≥1）个线性关系的约束，每个元素在n个线性关系中的序号i1,i2……in称为该元素的下标，并称该数组为n维数组。

**稀疏矩阵：**同时在矩阵中有很多值相同的元素并且它们的分布有一定的规律，称为**特殊矩阵**，或者矩阵中有很多零元素——称为**稀疏矩阵**

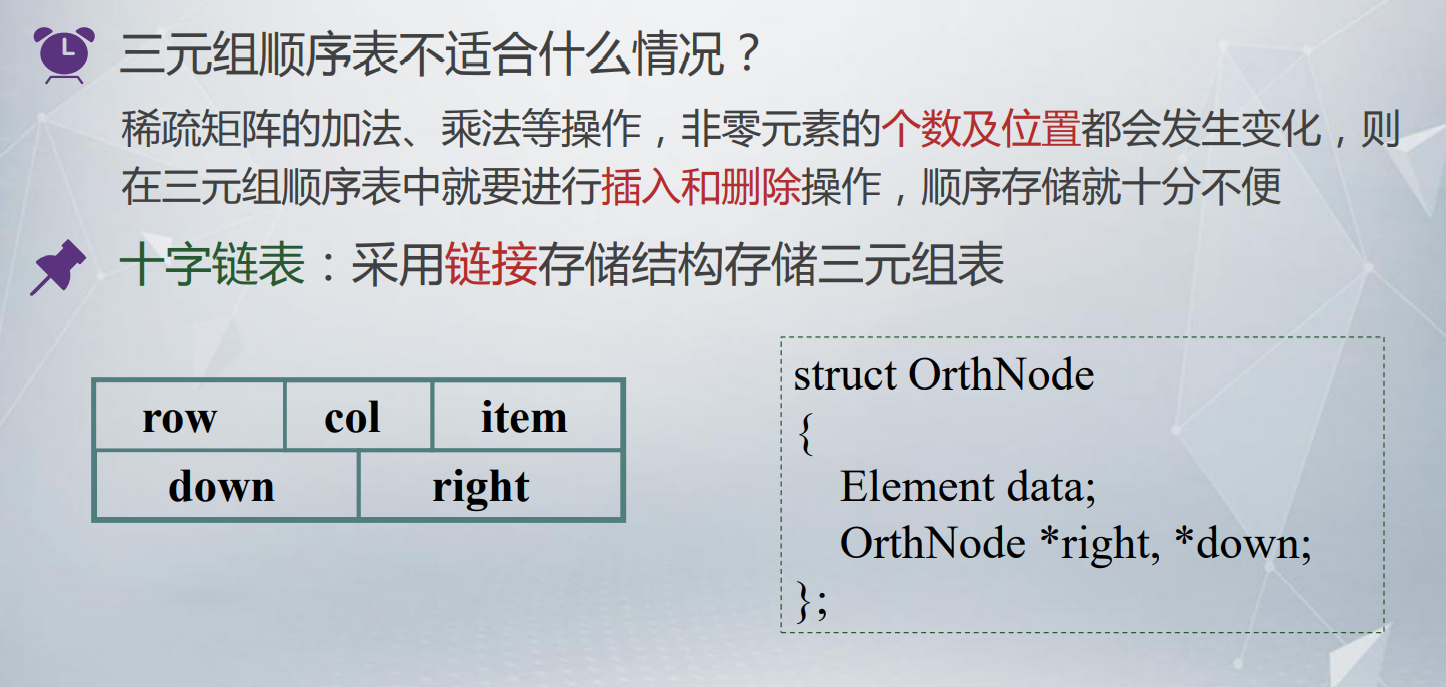
**稀疏矩阵的压缩存储**：

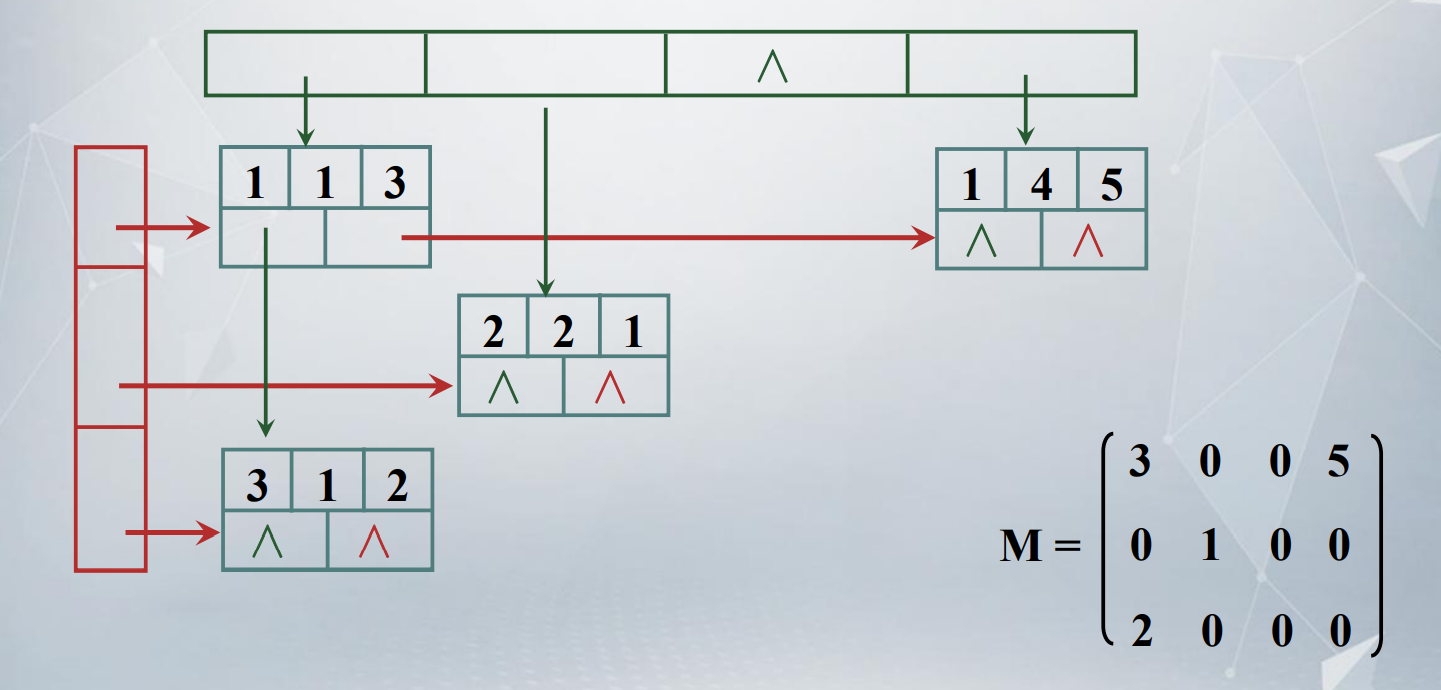




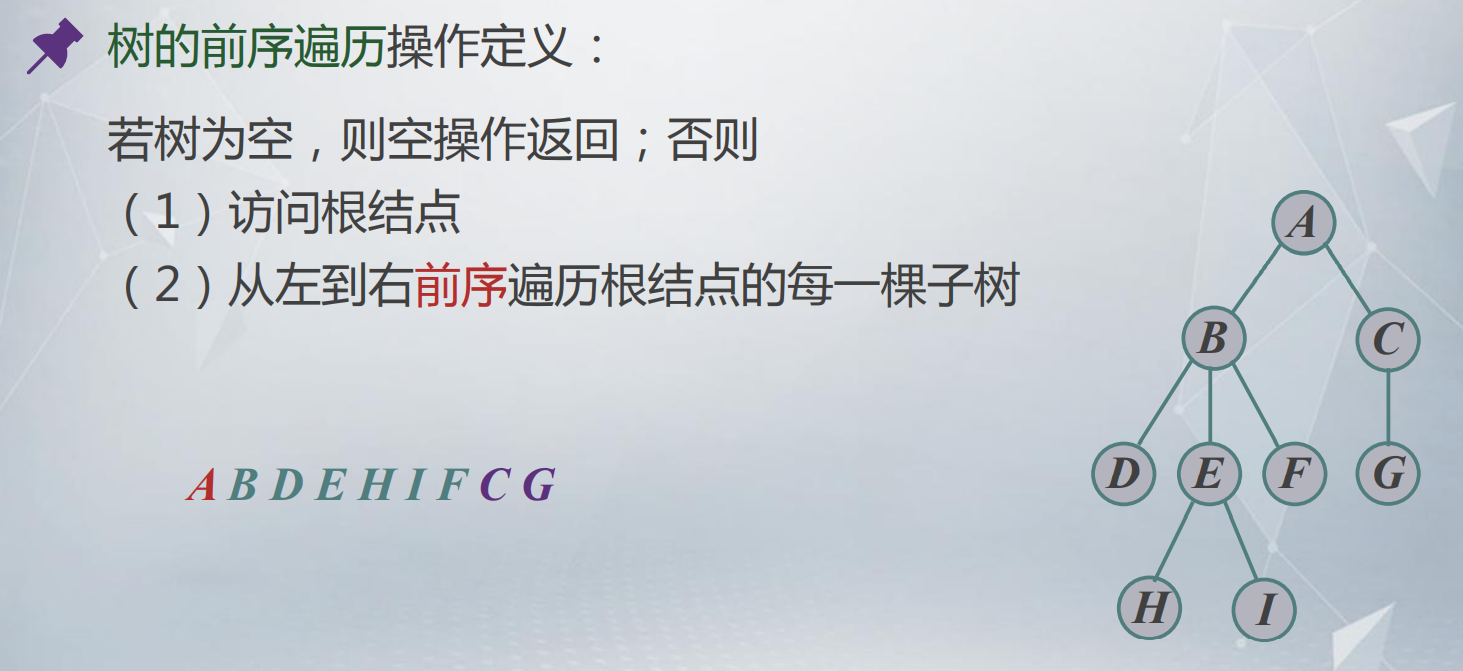


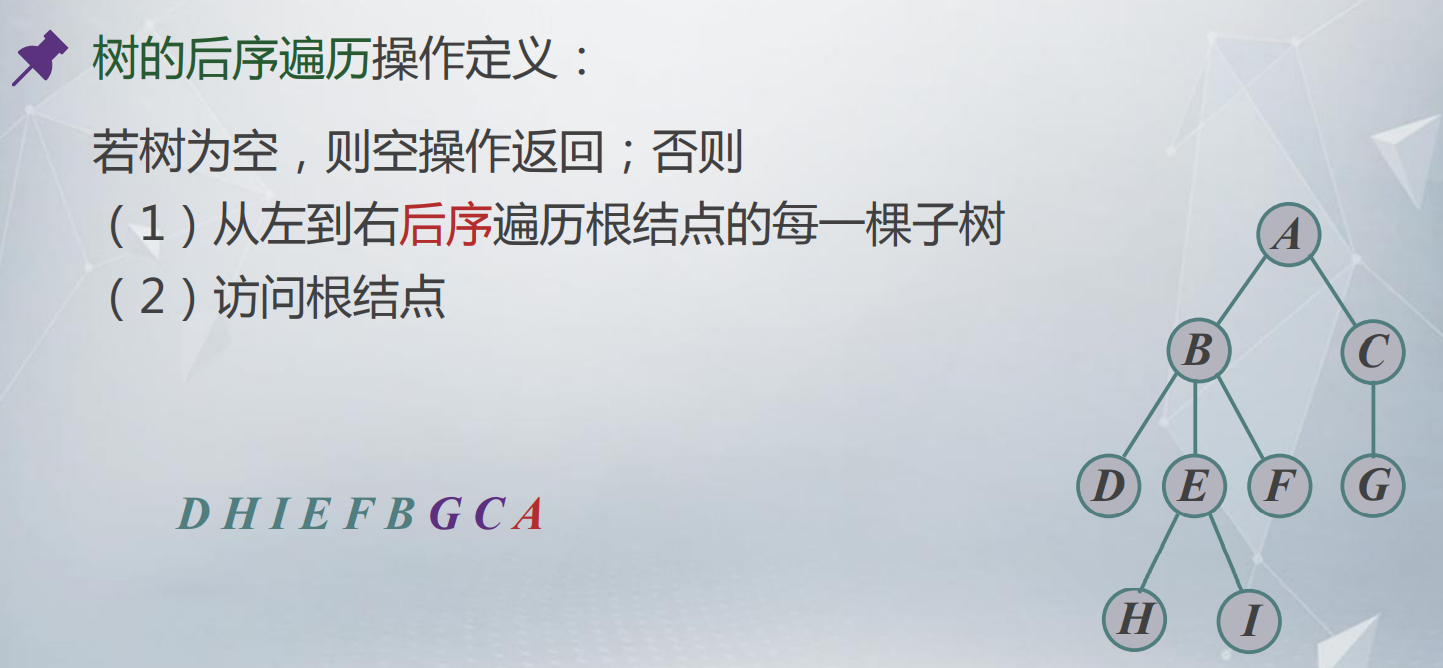
**十字链表：**

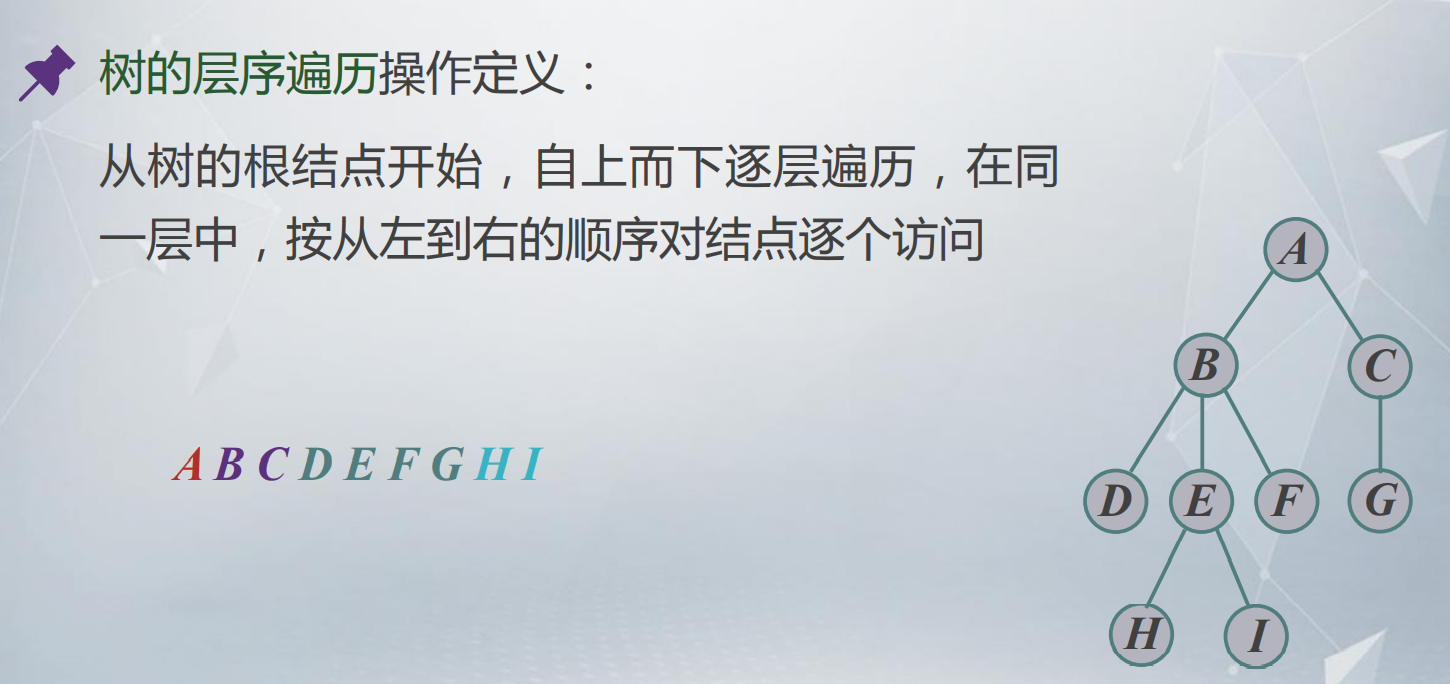




**树的遍历操作**

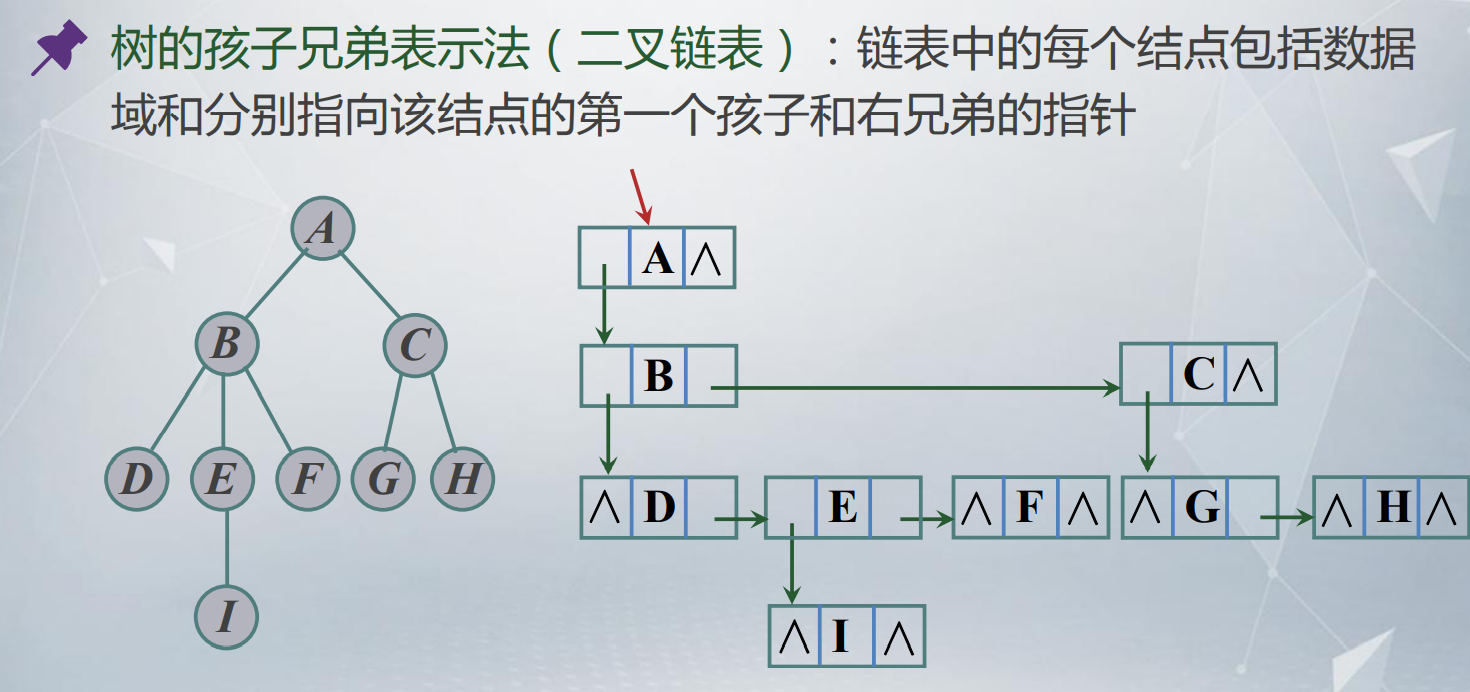


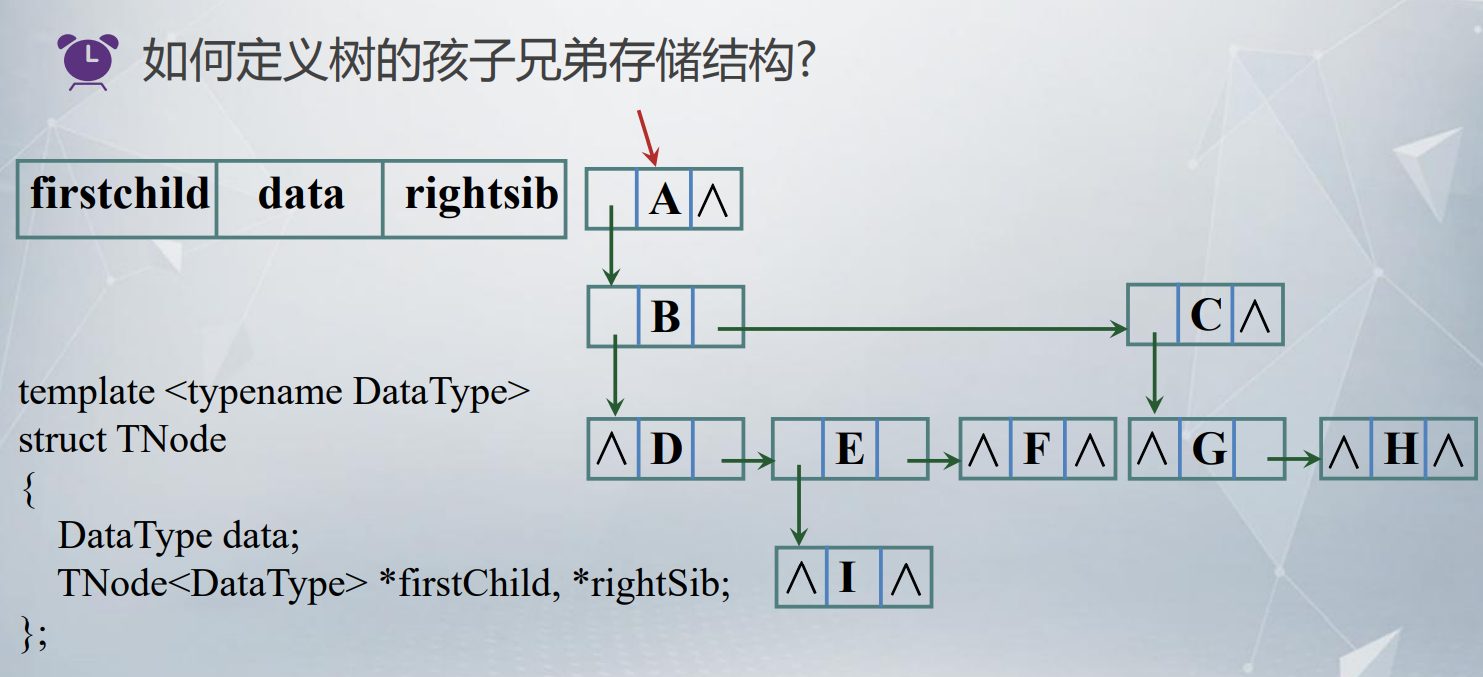




**树的存储结构：**

**（孩子兄弟表示法）**



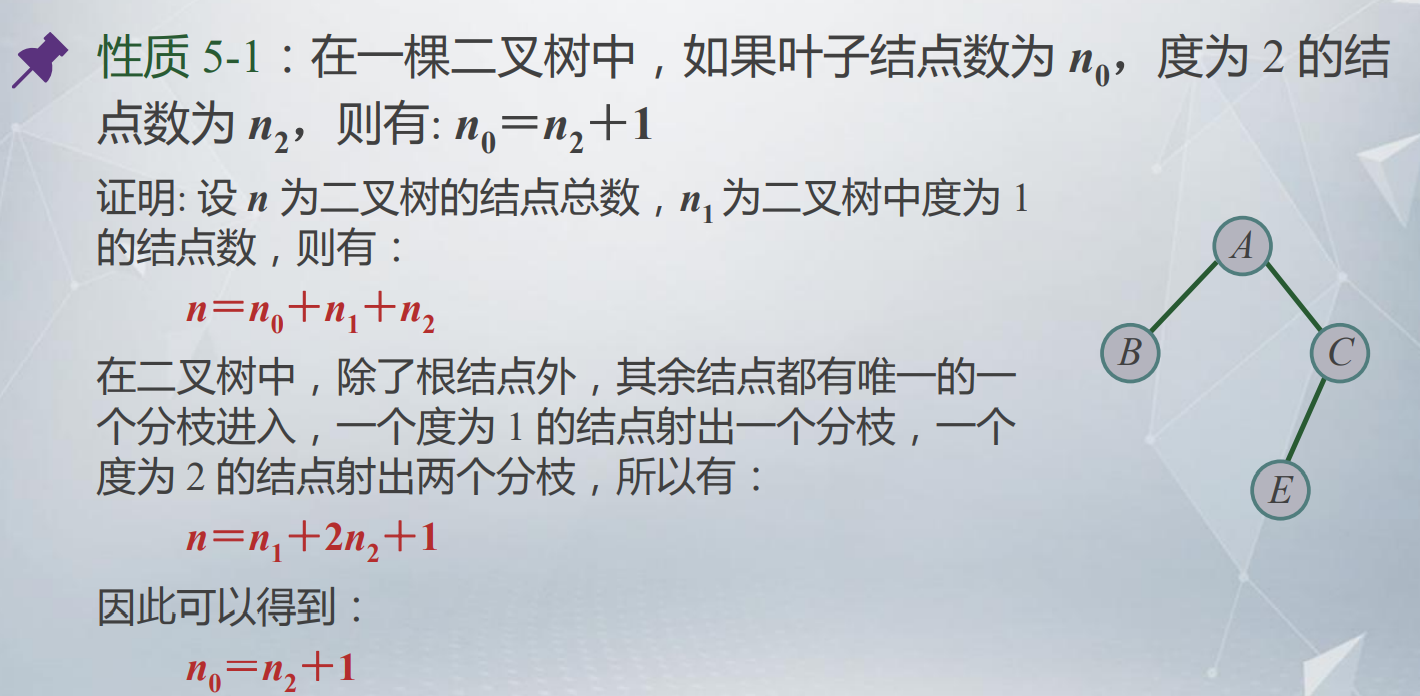


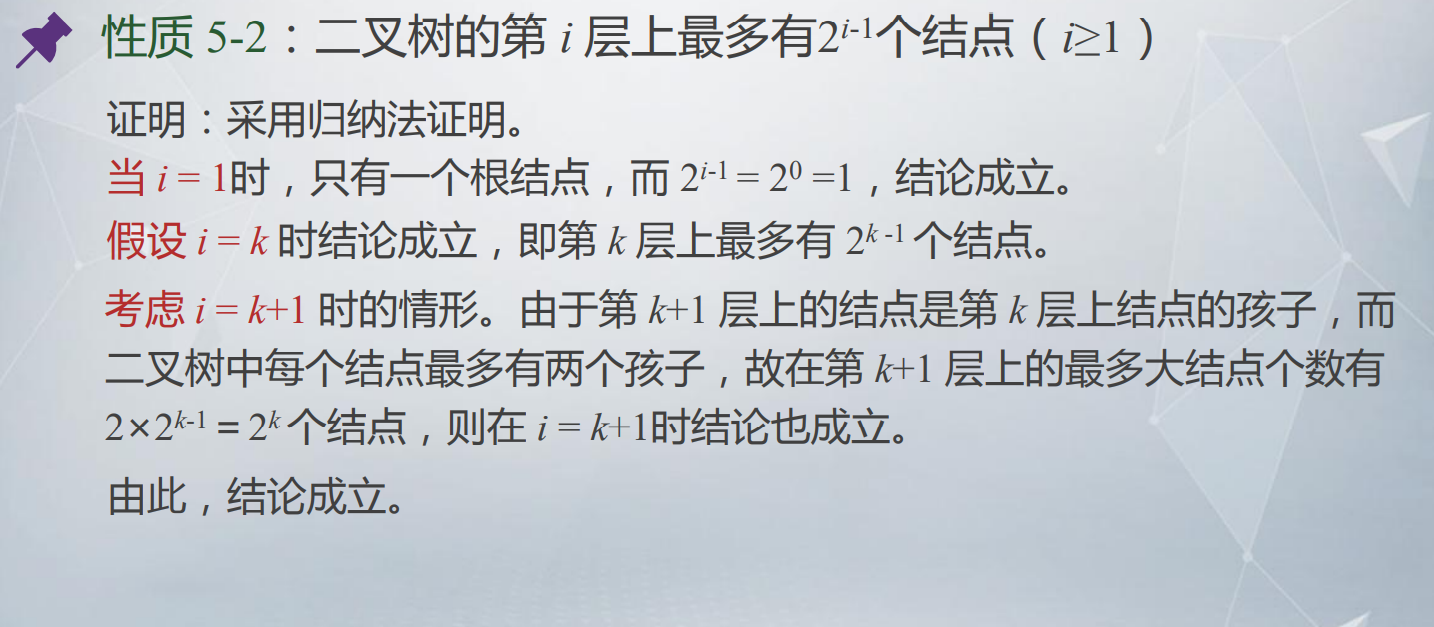
**二叉树的逻辑结构**

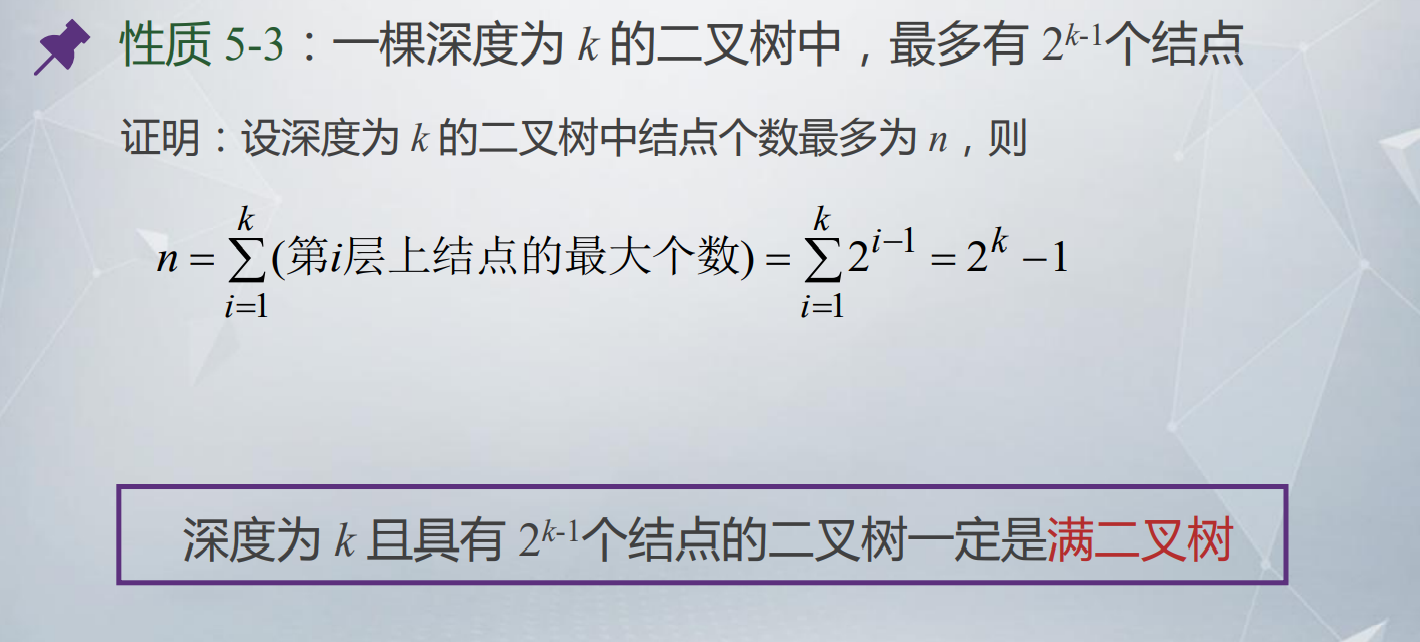
**二叉树：**二叉树是n（n≥0）个结点的有限集合，该集合或者空集（称为空二叉树），或者由一个根节点和两棵互不相交的，分别称为根结点的左子树和右子树的二叉树组成。

满二叉树：在一棵二叉树中，如果所有分支结点都存在左子树和右子树，并且所有叶子都在同一层上，这样的二叉树称为满二叉树。

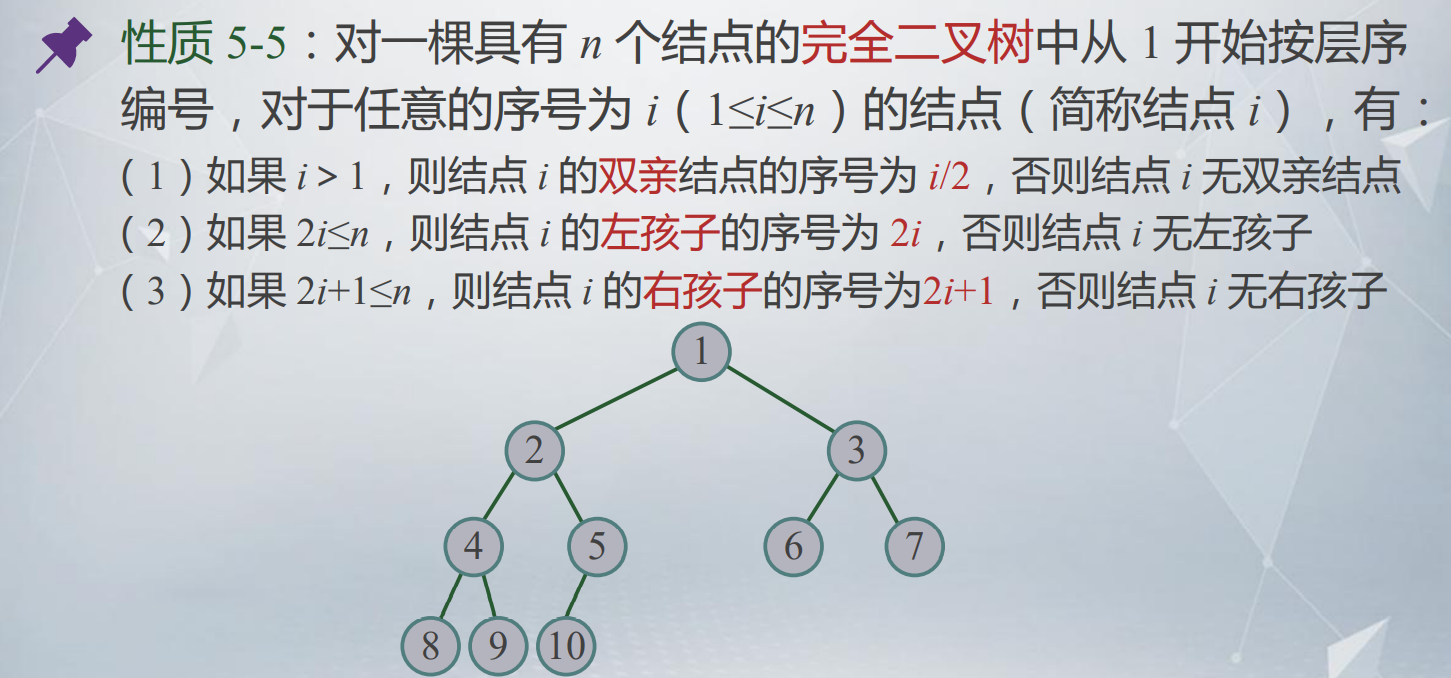
**二叉树的基本性质**





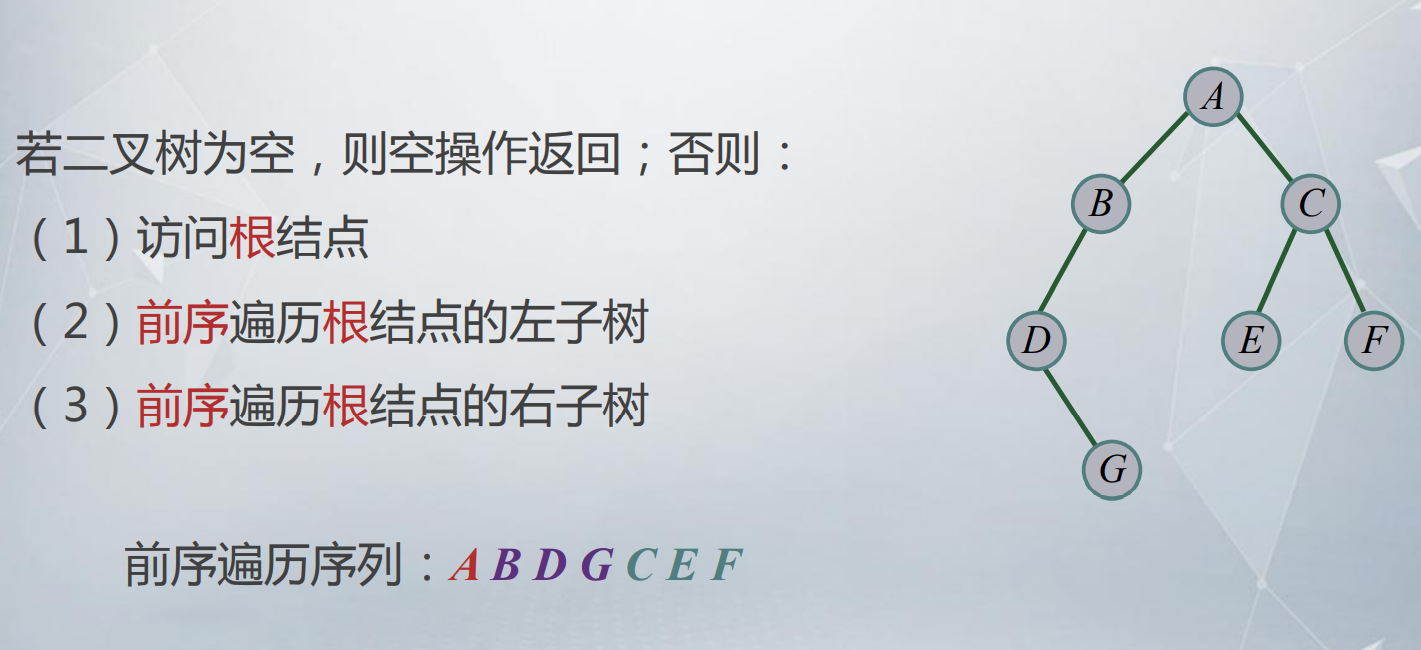




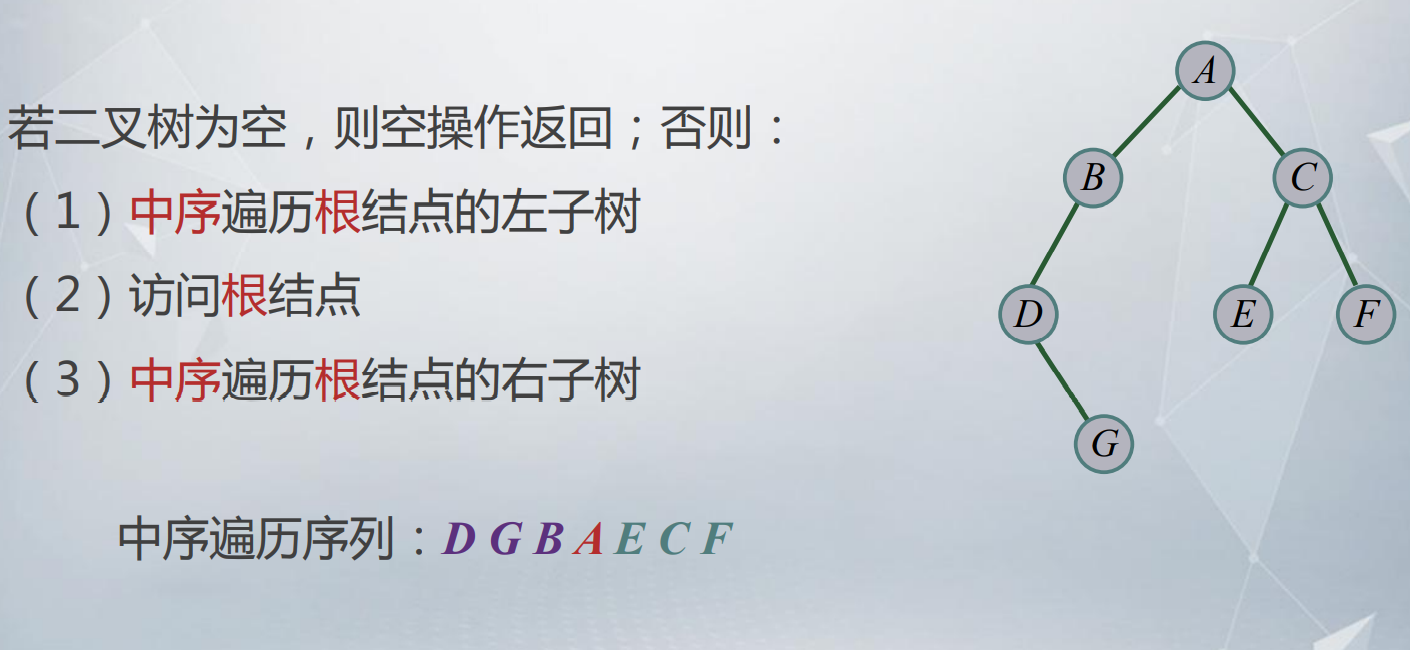


二叉树遍历

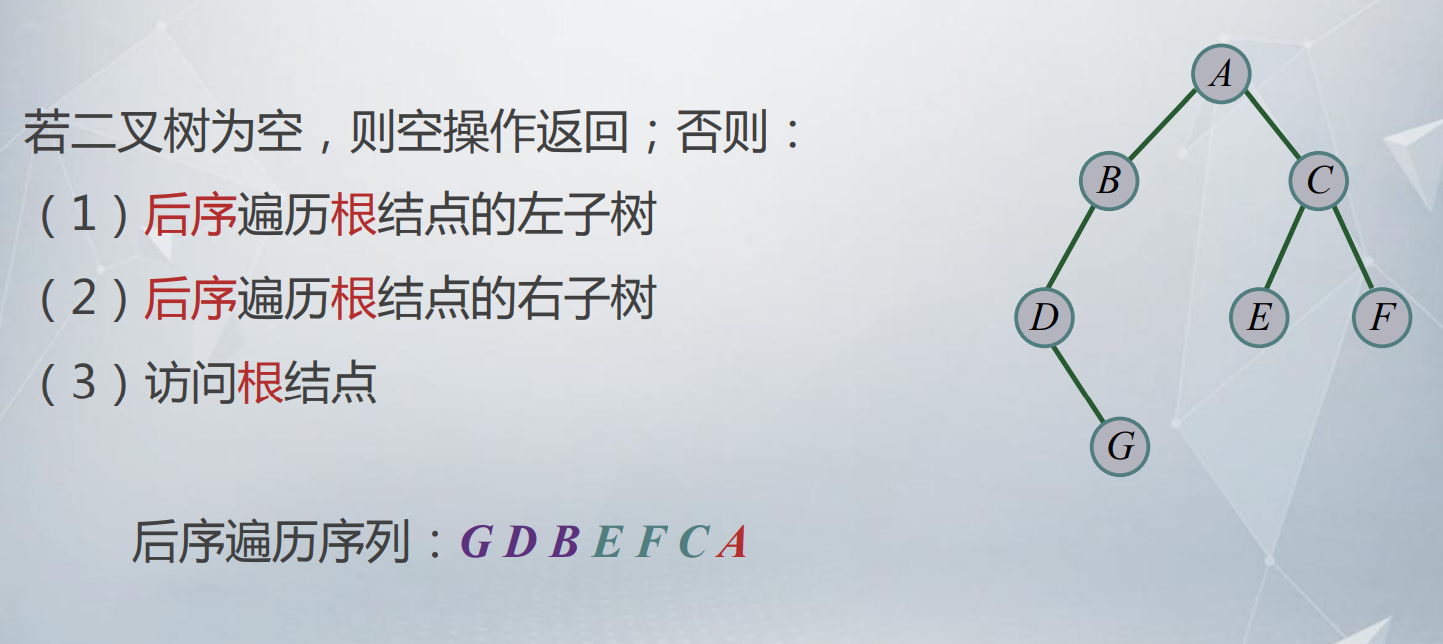
前序



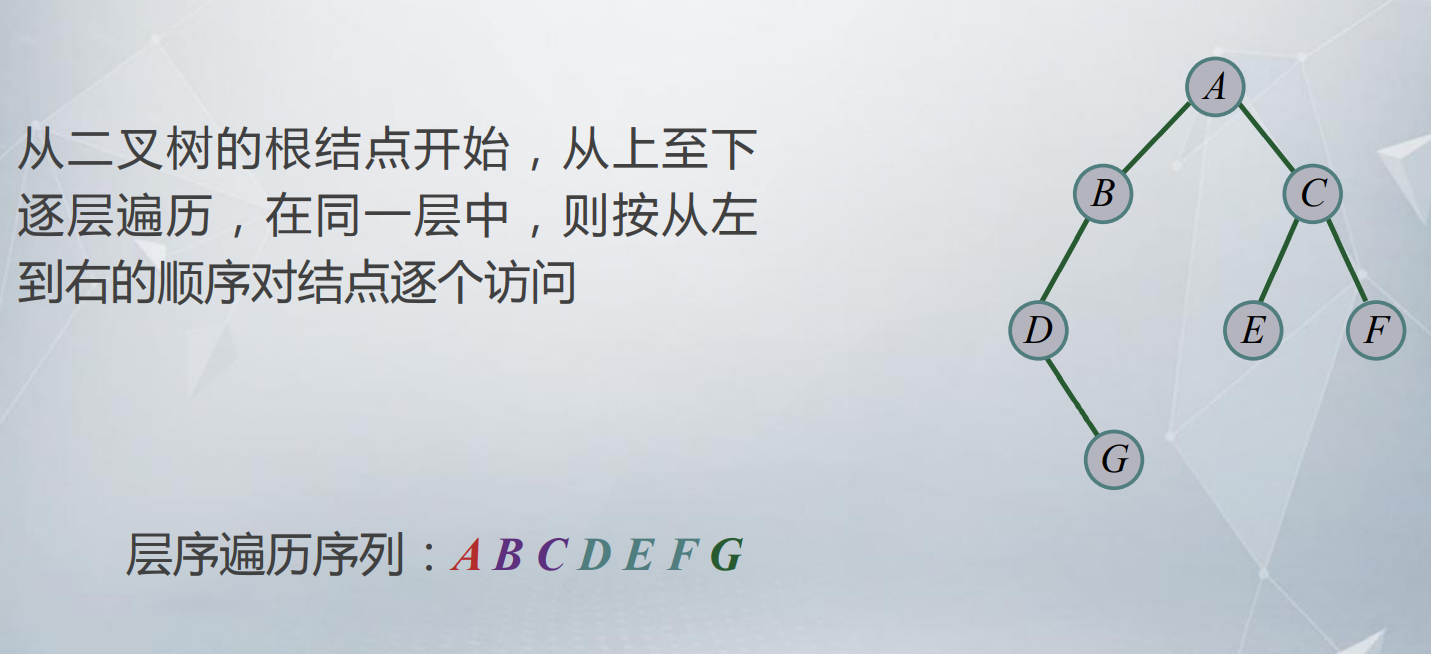
中序

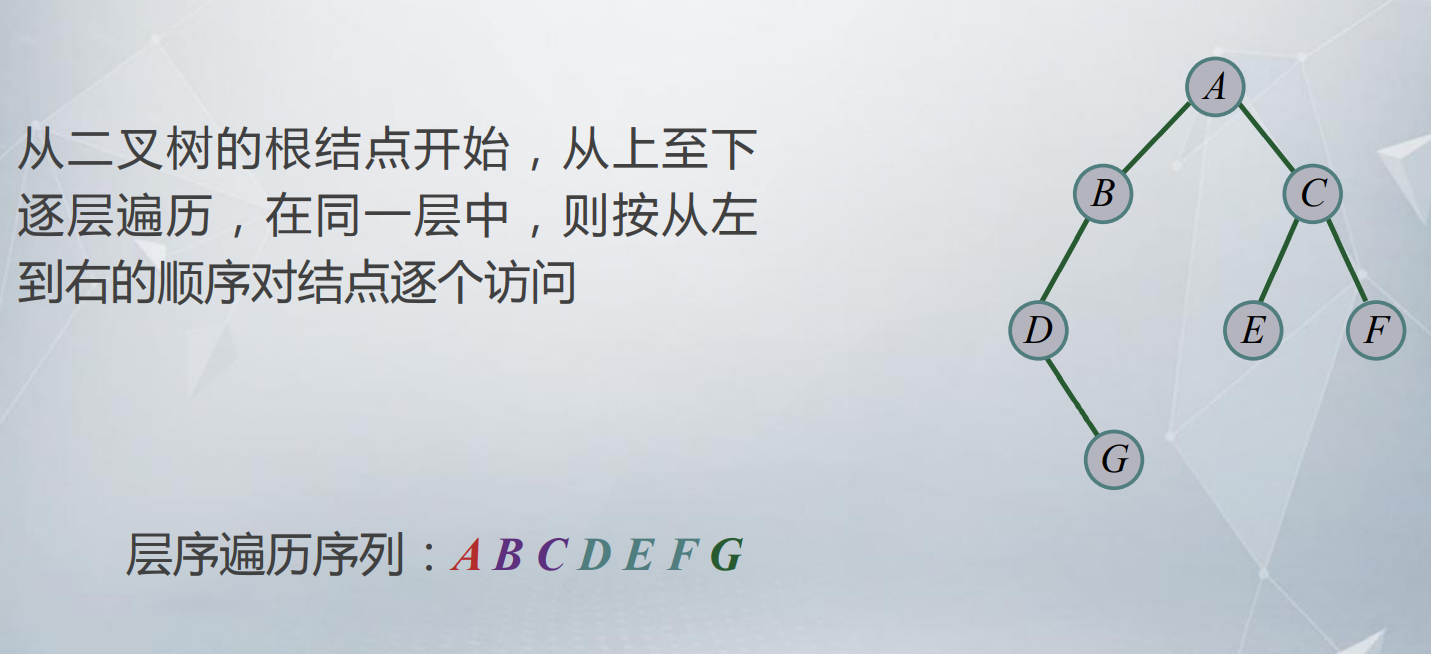


后序



层序





**二叉链表的实现**

template<typename DataType>

class BiTree

{

public:

BiTree(){root=Create();}

~BiTree(){Release(root);}

void PreOrder(){PreOrder(root);}

void InOrder(){InOrder(root);}

void PostOrder(){PostOrder(root);}

void LevelOrder();

private:

BiNode<DataType>\*Creat();

void Release(BiNode<DataType>\*bt);

void PreOrder(BiNode<DataType>\*bt);

void InOrder(BiNode<DataType>\*bt);

void PostOrder(BiNode<DataType>\*bt);

BiNode<DataType>\*root;

};

前序遍历

template <class DataType>

void BiTree<DataType> :: PreOrder(BiNode<DataType> \*bt)

{

if (bt == nullptr) return; //递归调用的结束条件

else {

cout << bt->data; //访问根结点bt的数据域

PreOrder(bt->lchild); //前序递归遍历bt的左子树

PreOrder(bt->rchild); //前序递归遍历bt的右子树

}

}

中序遍历

template <class DataType>

void BiTree<DataType> :: InOrder(BiNode<DataType> \*bt)

{

if (bt == nullptr) return; //递归调用的结束条件

else {

InOrder(bt->lchild); //前序递归遍历bt的左子树

cout << bt->data; //访问根结点bt的数据域

InOrder(bt->rchild); //前序递归遍历bt的右子树

}

}

后序遍历

template <class DataType>

void BiTree<DataType> :: PostOrder(BiNode<DataType> \*bt)

{

if (bt == nullptr) return; //递归调用的结束条件

else {

InOrder(bt->lchild); //前序递归遍历bt的左子树

InOrder(bt->rchild); //前序递归遍历bt的右子树

cout << bt->data; //访问根结点bt的数据域

}

}

层序遍历

template <class DataType>

void BiTree<DataType> :: LeverOrder( )

{

BiNode<DataType> \*Q[100], \*q = nullptr; //顺序队列最多100个结点

int front = -1, rear = -1; //队列初始化

if (root == nullptr) return; //二叉树为空，算法结束

Q[++rear] = root; //根指针入队

while (front != rear) //当队列非空时

{

q = Q[++front]; //出队

cout << q->data;

if (q->lchild != nullptr) Q[++rear] = q->lchild;

if (q->rchild != nullptr) Q[++rear] = q->rchild;

}

}

构造函数

template <class DataType>

BiNode<DataType> \*BiTree<DataType> :: Creat(BiNode<DataType> \*bt)

{

char ch;

cout << "请输入扩展二叉树的前序遍历序列，每次输入一个字符:";

cin >> ch; //输入结点的数据信息，假设为字符

if (ch == '#') bt = nullptr; //建立一棵空树

else {

bt = new BiNode<DataType>; bt->data = ch;

bt->lchild = Creat(bt->lchild); //递归建立左子树

bt->rchild = Creat(bt->rchild); //递归建立右子树

}

return bt;

}

析构函数

template <class DataType>

void BiTree<DataType> :: Release(BiNode<DataType> \*bt)

{

if (bt == nullptr) return;

else{

Release(bt->lchild); //释放左子树

Release(bt->rchild); //释放右子树

delete bt; //释放根结点

}

}