[数论 2](#_Toc481928121)

[1.十进制数转换为任意进制数 2](#_Toc481928122)

[2.整数二进制展开中数位1总数统计 2](#_Toc481928123)

[3.计算n个整数的总和 3](#_Toc481928124)

[4.幂函数算法 4](#_Toc481928125)

[5.Fibonacci数列第n项计算 4](#_Toc481928126)

[功能工具 5](#_Toc481928127)

[1.数组倒置 5](#_Toc481928128)

[2.括号匹配 5](#_Toc481928129)

[3.表达式求值及RPN转换 6](#_Toc481928130)

[排序 7](#_Toc481928131)

[1.起泡排序 7](#_Toc481928132)

[2.归并排序 7](#_Toc481928133)

[3.插入排序 8](#_Toc481928134)

[4.选择排序 8](#_Toc481928135)

[向量 8](#_Toc481928136)

[1.复制构造 8](#_Toc481928137)

[2.重载向量赋值操作符 9](#_Toc481928138)

[3. 扩容 9](#_Toc481928139)

[4. 缩容 9](#_Toc481928140)

[5.重载下标操作符[] 9](#_Toc481928141)

[6.置乱算法 9](#_Toc481928142)

[7.无序向量的顺序查找接口find() 9](#_Toc481928143)

[8.向量元素插入接口insert() 10](#_Toc481928144)

[9.向量元素删除或区间删除 10](#_Toc481928145)

[10.无序向量唯一化或清除重复元素或去重接口 10](#_Toc481928146)

[11.遍历接口 10](#_Toc481928147)

[12.有序性甄别 10](#_Toc481928148)

[13.有序向量唯一化或去重 11](#_Toc481928149)

[14.有序向量的查找 11](#_Toc481928150)

[列表 12](#_Toc481928151)

[1.默认构造方法 12](#_Toc481928152)

[2.重载列表类下标操作符 12](#_Toc481928153)

[3.无序列表查找 12](#_Toc481928154)

[4.列表节点插入接口 12](#_Toc481928155)

[5.基于复制的构造 13](#_Toc481928156)

[6.删除 13](#_Toc481928157)

[7.析构 14](#_Toc481928158)

[8.无序列表的唯一化或去重接口 14](#_Toc481928159)

[9.遍历 14](#_Toc481928160)

[10.有序列表唯一化 14](#_Toc481928161)

[11.有序列表查找 14](#_Toc481928162)

[12.列表最大节点/最大值定位 15](#_Toc481928163)

[二叉树 15](#_Toc481928164)

[1.孩子节点插入接口 15](#_Toc481928165)

[2.二叉树节点的高度更新 15](#_Toc481928166)

[3.树根节点和节点插入 15](#_Toc481928167)

[4.子树接入 15](#_Toc481928168)

[5.子树删除 16](#_Toc481928169)

[6.子树分离 16](#_Toc481928170)

[7.先序遍历 16](#_Toc481928171)

[8.后序遍历 17](#_Toc481928172)

[9.中序遍历 17](#_Toc481928173)

[10.层次遍历 19](#_Toc481928174)

常用技巧

①整数的翻倍或减半，使用位操作比使用乘除法效率更高

# 数论

## 1.十进制数转换为任意进制数

|  |
| --- |
| 1 **void** convert(Stack<char>& S, **\_\_int64** n, **int** base) { //十进制数n到base进制的转换（递归版） |
| 2 **static char** digit[] //0 < n, 1 < base <= 16，新进制下的数位符号，可视base取值范围适当扩充 |
| 3 = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' }; |
| 4 **if** (0 < n) { //在尚有余数之前，不断 |
| 5 convert(S, n / base, base); //通过递归得到所有更高位 |
| 6 S.push(digit[n % base]); //输出低位 |
| 7 } |
| 8 } //新进制下由高到低的各数位，自顶而下保存于栈S中 |
| 1 **void** convert(Stack<char>& S, **\_\_int64** n, **int** base) { //十进制数n到base进制的转换（迭代版） |
| 2 **static char** digit[] //0 < n, 1 < base <= 16，新进制下的数位符号，可视base取值范围适当扩充 |
| 3 = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' }; |
| 4 **while** (n > 0) { //由低到高，逐一计算出新进制下的各数位 |
| 5 **int** remainder = (int) (n % base); S.push(digit[remainder]); //余数（当前位）入栈 |
| 6 n /= base; //n更新为其对base的除商 |
| 7 } |
| 8 } //新进制下由高到低的各数位，自顶而下保存于栈S中 |

## 2.整数二进制展开中数位1总数统计

|  |
| --- |
| 1 **int** countOnes(unsigned **int** n) { //统计整数n的二进制展开中数位1的总数：O(logn) |
| 2 **int** ones = 0; //计数器复位 |
| 3 **while** (0 < n) { //在n缩减至0之前，反复地 |
| 4 ones += (1 & n); //检查最低位，若为1则计数 |
| 5 n >>= 1; //右移一位 |
| 6 } |
| 7 **return** ones; //返回计数 |
| 8 } //等效于glibc的内置函数int \_\_builtin\_popcount (unsigned int n) |
| 1 **int** countOnes1(unsigned **int** n) { //统计整数n的二进制展开中数位1的总数：O(ones)正比于数位1的总数 |
| 2 **int** ones = 0; //计数器复位 |
| 3 **while** (0 < n) { //在n缩减至0之前，反复地 |
| 4 ones++; //计数（至少有一位为1） |
| 5 n &= n - 1; //清除当前最靠右的1 |
| 6 } |
| 7 **return** ones; //返回计数 |
| 8 } //等效于glibc的内置函数int \_\_builtin\_popcount (unsigned int n) |
| 1 **#define** POW(c) (1 << (c)) //2^c |
| 2 **#define** MASK(c) (((unsigned **long)** -1) / (POW(POW(c)) + 1)) //以2^c位为单位分组，相间地全0和全1 |
| 3 // MASK(0) = 55555555(h) = 01010101010101010101010101010101(b) |
| 4 // MASK(1) = 33333333(h) = 00110011001100110011001100110011(b) |
| 5 // MASK(2) = 0f0f0f0f(h) = 00001111000011110000111100001111(b) |
| 6 // MASK(3) = 00ff00ff(h) = 00000000111111110000000011111111(b) |
| 7 // MASK(4) = 0000ffff(h) = 00000000000000001111111111111111(b) |
| 8 |
| 9 //输入：n的二进制展开中，以2^c位为单位分组，各组数值已经分别等于原先这2^c位中1的数目 |
| 10 **#define** ROUND(n, c) (((n) & MASK(c)) + ((n) >> POW(c) & MASK(c))) //运算优先级：先右移，再位与 |
| 11 //过程：以2^c位为单位分组，相邻的组两两捉对累加，累加值用原2^(c + 1)位就地记录 |
| 12 //输出：n的二进制展开中，以2^(c + 1)位为单位分组，各组数值已经分别等于原先这2^(c + 1)位中1的数目 |
| 13 |
| 14 **int** countOnes2(unsigned **int** n) { //统计整数n的二进制展开中数位1的总数 |
| 15 n = ROUND(n, 0); //以02位为单位分组，各组内前01位与后01位累加，得到原先这02位中1的数目 |
| 16 n = ROUND(n, 1); //以04位为单位分组，各组内前02位与后02位累加，得到原先这04位中1的数目 |
| 17 n = ROUND(n, 2); //以08位为单位分组，各组内前04位与后04位累加，得到原先这08位中1的数目 |
| 18 n = ROUND(n, 3); //以16位为单位分组，各组内前08位与后08位累加，得到原先这16位中1的数目 |
| 19 n = ROUND(n, 4); //以32位为单位分组，各组内前16位与后16位累加，得到原先这32位中1的数目 |
| 20 **return** n; //返回统计结果 |
| 21 } //32位字长时，O(log\_2(32)) = O(5) = O(1)，复杂度为O(logr)，r=O(log2n) |

## 3.计算n个整数的总和

|  |
| --- |
| 1 **int** sumI(int A[], **int** n) { //数组求和算法（迭代版） |
| 2 **int** sum = 0; //初始化累计器，O(1) |
| 3 **for** (int i = 0; i < n; i++) //对全部共O(n)个元素，逐一 |
| 4 sum += A[i]; //累计，O(1) |
| 5 **return** sum; //返回累计值，O(1) |
| 6 } //O(1) + O(n)\*O(1) + O(1) = O(n+2) = O(n) |
| 1 **int** sum(int A[], **int** n) { //数组求和算法（线性递归版） |
| 2 **if** (1 > n) //平凡情况，递归基 |
| 3 **return** 0; //直接（非递归式）计算 |
| 4 **else** //一般情况 |
| 5 **return** sum(A, n - 1) + A[n - 1]; //递归：前n - 1项之和，再累计第n - 1顷 |
| 6 } //O(1)\*递归深度 = O(1)\*(n + 1) = O(n) |
| 1 **int** sum(int A[], **int** lo, **int** hi) { //数组求和算法（二分递归版，入口为sum(A, 0, n - 1)） |
| 2 **if** (lo == hi) //如遇递归基（区间长度已降至1），则 |
| 3 **return** A[lo]; //直接返回该元素 |
| 4 **else** { //否则（一般情况下lo < hi），则 |
| 5 **int** mi = (lo + hi) >> 1; //以居中单元为界，将原区间一分为二 |
| 6 **return** sum(A, lo, mi) + sum(A, mi + 1, hi); //递归对各子数组求和，然后合计 |
| 7 } |
| 8 } //O(hi - lo + 1)，线性正比于区间长度 |

## 4.幂函数算法

|  |
| --- |
| 1 **\_\_int64** power2BF\_I(int n) { //幂函数2^n算法（蛮力迭代版），n >= 0 |
| 2 **\_\_int64** pow = 1; //O(1)：累积器初始化为2^0 |
| 3 **while** (0 < n --) //O(n)：迭代n轮，每轮都 |
| 4 pow <<= 1; //O(1)：将累积器翻倍 |
| 5 **return** pow; //O(1)：返回累积器 |
| 6 } //O(n) = O(2^r)，r为输入指数n的比特位数 |
| 1 **inline \_\_int64** sqr(\_\_int64 a) { **return** a \* a; } |
| 2 **\_\_int64** power2(int n) { //幂函数2^n算法（优化递归版），n >= 0 |
| 3 **if** (0 == n) **return** 1; //递归基 |
| 4 **return** (n & 1) ? sqr(power2(n >> 1)) << 1 : sqr(power2(n >> 1)); //视n的奇偶分别递归 |
| 5 } //O(logn) = O(r)，r为输入指数n的比特位数 |
| 1 **\_\_int64** power(\_int64 a, int n) { //幂函数a^n算法（优化迭代版），n >= 0 |
| 2 **\_\_int64** pow = 1; //O(1)：累积器初始化为a^0 |
| 3 **\_\_int64** p = a; //O(1)：累乘项初始化为a |
| 4 **while** (0 < n) { //O(logn)：迭代log(n)轮，每轮都 |
| 5 **if** (n & 1) //O(1)：根据当前比特位是否为1，决定是否 |
| 6 pow \*= p; //O(1)：将当前累乘项计入累积器 |
| 7 n >>= 1; //O(1)：指数减半 |
| 8 p \*= p; //O(1)：累乘项自乘 |
| 9 } |
| 10 **return** pow; //O(1)：返回累积器 |
| 11 } //O(logn) = O(r)，r为输入指数n的比特位数 |

## 5.Fibonacci数列第n项计算

|  |
| --- |
| 1 **\_\_int64** fib(int n) { //计算Fibonacci数列的第n项（二分递归版）：O(2^n) |
| 2 **return** (2 > n) ? |
| 3 (\_\_int64)n //若到达递归基，直接取值 |
| 4 : fib(n - 1) + fib(n - 2); //否则，递归计算前两项，其和即为正解 |
| 5 } |
| 1 **\_\_int64** fib(int n, **\_\_int64&** prev) { //计算Fibonacci数列第n项（线性递归版）：入口形式fib(n, prev) |
| 2 **if** (0 == n) //若到达递归基，则 |
| 3 { prev = 1; **return** 0; } //直接取值：fib(-1) = 1, fib(0) = 0 |
| 4 **else** { //否则 |
| 5 **\_\_int64** prevPrev; prev = fib(n - 1, prevPrev); //递归计算前两项 |
| 6 **return** prevPrev + prev; //其和即为正解 |
| 7 } |
| 8 } //用辅助变量记录前一项，返回数列的当前项，O(n) |
| 1 **\_\_int64** fibI(int n) { //计算Fibonacci数列的第n项（迭代版）：O(n) |
| 2 **\_\_int64** f = 0, g = 1; //初始化：fib(0) = 0, fib(1) = 1 |
| 3 **while** (0 < n--) { g += f; f = g - f; } //依据原始定义，通过n次加法和减法计算fib(n) |
| 4 **return** f; //返回 |
| 5 } |

# 功能工具

## 1.数组倒置

|  |
| --- |
| 1 **void** reverse(**int\*** A, **int** lo, **int** hi) { //数组倒置（多递归基递归版） |
| 2 **if** (lo < hi) { |
| 3 swap(A[lo], A[hi]); //交换A[lo]和A[hi] |
| 4 reverse(A, lo + 1, hi - 1); //递归倒置A(lo, hi) |
| 5 } //else隐含了两种递归基 |
| 6 } //O(hi - lo + 1) |
| 1 **void** reverse(int\* A, **int** lo, **int** hi) { //数组倒置（迭代版） |
| 2 **while** (lo < hi) //用while替换跳转标志和if，完全等效 |
| 3 swap(A[lo++], A[hi--]); //交换A[lo]和A[hi]，收缩待倒置区间 |
| 4 } //O(hi - lo + 1) |

## 2.括号匹配

|  |
| --- |
| 1 **void** trim(**const char** exp[], **int&** lo, **int&** hi) { //删除表达式exp[lo, hi]不含括号的最长前缀、后缀 |
| 2 **while** ((lo <= hi) && (exp[lo] != '(') && (exp[lo] != ')')) lo++; //查找第一个和 |
| 3 **while** ((lo <= hi) && (exp[hi] != '(') && (exp[hi] != ')')) hi--; //最后一个括号 |
| 4 } |
| 5 |
| 6 **int** divide(**const char** exp[], **int** lo, **int** hi) { //切分表达式exp[lo, hi]，使exp匹配仅当它们匹配 |
| 7 **int** mi = lo; **int** crc = 1; //crc为[lo, mi]范围内左、右括号数目之差 |
| 8 **while** ((0 < crc) && (++mi < hi)) //逐个检查各字符，直到左、右括号数目相等，或者越界 |
| 9 { **if** (exp[mi] == ')') crc--; **if** (exp[mi] == '(') crc++; } //左、右括号分别计数 |
| 10 **return** mi; //若mi <= hi，则为合法切分点；否则，意味着局部不可能匹配 |
| 11 } |
| 12 |
| 13 **bool** paren(**const char** exp[], **int** lo, **int** hi) { //检查表达式exp[lo, hi]是否括号匹配（递归版） |
| 14 trim(exp, lo, hi); **if** (lo > hi) **return true;** //清除不含括号的前缀、后缀 |
| 15 **if** (exp[lo] != '(') **return false;** //首字符非左括号，则必不匹配 |
| 16 **if** (exp[hi] != ')') **return false;** //末字符非右括号，则必不匹配 |
| 17 **int** mi = divide(exp, lo, hi); //确定适当的切分点 |
| 18 **if** (mi > hi) **return false;** //切分点不合法，意味着局部以至整体不匹配 |
| 19 **return** paren(exp, lo + 1, mi - 1) && paren(exp, mi + 1, hi); //分别检查左、右子表达式 |
| 20 } //复杂度O(n^2) |
| 1 **bool** paren(**const char** exp[], **int** lo, **int** hi) { //表达式括号匹配检查，可兼顾三种括号（迭代版） |
| 2 Stack<char> S; //使用栈记录已发现但尚未匹配的左括号 |
| 3 **for** (int i = 0; exp[i]; i++) /\* 逐一检查当前字符 \*/ |
| 4 **switch** (exp[i]) { //左括号直接进栈；右括号若与栈顶失配，则表达式必不匹配 |
| **5 case** '(': **case** '[': **case** '{': S.push(exp[i]); **break;** |
| **6 case** ')': **if** ((S.empty()) || ('(' != S.pop())) **return false; break;** |
| **7 case** ']': **if** ((S.empty()) || ('[' != S.pop())) **return false; break;** |
| **8 case** '}': **if** ((S.empty()) || ('{' != S.pop())) **return false; break;** |
| 9 **default: break;** //非括号字符一律忽略 |
| 10 } |
| 11 **return** S.empty(); //整个表达式扫描过后，栈中若仍残留（左）括号，则不匹配；否则（栈空）匹配 |
| 12 } |

## 3.表达式求值及RPN转换

|  |
| --- |
| 1 **float** evaluate(char\* S, **char\*&** RPN) { //对（已剔除白空格的）表达式S求值，并转换为逆波兰式RPN |
| 2 Stack<float> opnd; Stack<char> optr; //运算数栈、运算符栈 |
| 3 optr.push('\0'); //尾哨兵'\0'也作为头哨兵首先入栈 |
| 4 **while** (!optr.empty()) { //在运算符栈非空之前，逐个处理表达式中各字符 |
| 5 **if** (isdigit(\*S)) { //若当前字符为操作数，则 |
| 6 readNumber(S, opnd); append(RPN, opnd.top()); //读入操作数，并将其接至RPN末尾 |
| 7 } **else** //若当前字符为运算符，则 |
| 8 **switch(orderBetween(optr.top(),** \*S)) { //视其与栈顶运算符之间优先级高低分别处理 |
| 9 **case** '<': //栈顶运算符优先级更低时 |
| 10 optr.push(\*S); S++; //计算推迟，当前运算符进栈 |
| 11 **break;** |
| 12 **case** '=': //优先级相等（当前运算符为右括号或者尾部哨兵'\0'）时 |
| 13 optr.pop(); S++; //脱括号并接收下一个字符 |
| 14 **break;** |
| 15 **case** '>': { //栈顶运算符优先级更高时，可实施相应的计算，并将结果重新入栈 |
| 16 **char** op = optr.pop(); append(RPN, op); //栈顶运算符出栈并续接至RPN末尾 |
| 17 **if** ('!' == op) { //若属于一元运算符 |
| 18 **float** pOpnd = opnd.pop(); //只需取出一个操作数，并 |
| 19 opnd.push(calcu(op, pOpnd)); //实施一元计算，结果入栈 |
| 20 } **else** { //对于其它（二元）运算符 |
| 21 **float** pOpnd2 = opnd.pop(), pOpnd1 = opnd.pop(); //取出后、前操作数 |
| 22 opnd.push(calcu(pOpnd1, op, pOpnd2)); //实施二元计算，结果入栈 |
| 23 } |
| 24 **break;** |
| 25 } |
| 26 **default** : **exit(-1);** //逢语法错误，不做处理直接退出 |
| 27 }//switch |
| 28 }//while |
| 29 **return** opnd.pop(); //弹出并返回最后的计算结果 |
| 30 } |

# 排序

## 1.起泡排序

|  |
| --- |
| 1 **template** <**typename** T> //向量的起泡排序 |
| 2 **void** Vector<T>::bubbleSort(Rank lo, Rank hi) //assert: 0 <= lo < hi <= size |
| 3 { **while** (!bubble(lo, hi--)); } //逐趟做扫描交换，直至全序 |
| 1 **template** <**typename** T> **bool** Vector<T>::bubble(Rank lo, Rank hi) { //一趟扫描交换，前后总时间复杂度O(n^2) |
| 2 **bool** sorted = **true;** //整体有序标志 |
| 3 **while** (++lo < hi) //自左向右，逐一检查各对相邻元素 |
| 4 **if** (\_elem[lo - 1] > \_elem[lo]) { //若逆序，则 |
| 5 sorted = **false;** //意味着尚未整体有序，并需要 |
| 6 swap(\_elem[lo - 1], \_elem[lo]); //通过交换使局部有序 |
| 7 } |
| 8 **return** sorted; //返回有序标志 |
| 9 } |

## 2.归并排序

|  |
| --- |
| 1 **template** <**typename** T> //向量归并排序 总时间复杂度Θ(nlogn) |
| 2 **void** Vector<T>::mergeSort(Rank lo, Rank hi) { //0 <= lo < hi <= size |
| 3 **if** (hi - lo < 2) **return;** //单元素区间自然有序，否则... |
| 4 **int** mi = (lo + hi) >> 1; //以中点为界 |
| 5 mergeSort(lo, mi); mergeSort(mi, hi); merge(lo, mi, hi); //分别对前、后半段排序，然后归并 |
| 6 } |
| 1 **template** <**typename** T> //有序向量的归并 时间复杂度Θ(n) |
| 2 **void** Vector<T>::merge(Rank lo, Rank mi, Rank hi) { //以mi为界、各自有序的子向量[lo, mi)和[mi, hi) |
| 3 T\* A = \_elem + lo; //合并后的向量A[0, hi - lo) = \_elem[lo, hi) |
| 4 **int** lb = mi - lo; T\* B = **new** T[lb]; //前子向量B[0, lb) = \_elem[lo, mi) |
| 5 **for** (Rank i = 0; i < lb; B[i] = A[i++]); //复制前子向量 |
| 6 **int** lc = hi - mi; T\* C = \_elem + mi; //后子向量C[0, lc) = \_elem[mi, hi) |
| 7 **for** (Rank i = 0, j = 0, k = 0; (j < lb) || (k < lc); ) { //将B[j]和C[k]中的小者续至A末尾 |
| 8 **if** ( (j < lb) && ( !(k < lc) || (B[j] <= C[k]) ) ) A[i++] = B[j++]; |
| 9 **if** ( (k < lc) && ( !(j < lb) || (C[k] < B[j]) ) ) A[i++] = C[k++]; |
| 10 } |
| 11 **delete** [] B; //释放临时空间B |
| 12 } //归并后得到完整的有序向量[lo, hi) |
| 1 **template** <**typename** T> //列表的归并排序算法：对起始于位置p的n个元素排序，复杂度O(nlogn) |
| 2 **void** List<T>::mergeSort(ListNodePosi(T)& p, **int** n) { //valid(p) && rank(p) + n <= size |
| 3 **if** (n < 2) **return;** //若待排序范围已足够小，则直接返回；否则... |
| 4 **int** m = n >> 1; //以中点为界 |
| 5 ListNodePosi(T) q = p; **for** (int i = 0; i < m; i++) q = q->succ; //均分列表 |
| 6 mergeSort(p, m); mergeSort(q, n - m); //对前、后子列表分别排序 |
| 7 merge(p, m, \*this, q, n - m); //归并 |
| 8 } //注意：排序后，p依然指向归并后区间的（新）起点 |
| 1 **template** <**typename** T> //有序列表的归并：当前列表中自p起的n个元素，与列表L中自q起的m个元素归并 |
| 2 **void** List<T>::merge(ListNodePosi(T)& p, **int** n, List<T>& L, ListNodePosi(T) q, **int** m) { |
| 3 // assert: this.valid(p) && rank(p) + n <= size && this.sorted(p, n) |
| 4 // L.valid(q) && rank(q) + m <= L.\_size && L.sorted(q, m) |
| 5 // 注意：在归并排序之类的场合，有可能 this == L && rank(p) + n = rank(q) |
| 6 ListNodePosi(T) pp = p->pred; //借助前驱（可能是header），以便返回前 ... |
| 7 **while** (0 < m) //在q尚未移出区间之前 |
| 8 **if** ((0 < n) && (p->data <= q->data)) //若p仍在区间内且v(p) <= v(q)，则 |
| 9 { **if** (q == (p = p->succ)) **break;** n--; } //将p替换为其直接后继（等效于将p归入合并的列表） |
| 10 **else** //若p已超出右界或v(q) < v(p)，则 |
| 11 { insertBefore(p, L.remove((q = q->succ)->pred)); m--; } //将q转移至p之前 |
| 12 p = pp->succ; //确定归并后区间的（新）起点 |
| 13 } //复杂度O(n+m) |

## 3.插入排序

|  |
| --- |
| 1 **template** <**typename** T> //列表的插入排序算法：对起始于位置p的n个元素排序 |
| 2 **void** List<T>::insertionSort(ListNodePosi(T) p, **int** n) { //valid(p) && rank(p) + n <= size |
| 3 **for** (int r = 0; r < n; r++) { //逐一为各节点 |
| 4 insertAfter(search(p->data, r, p), p->data); //查找适当的位置并插入 |
| 5 p = p->succ; remove(p->pred); //转向下一节点 |
| 6 } |
| 7 } //复杂度O(n^2) |

## 4.选择排序

|  |
| --- |
| 1 **template** <**typename** T> //列表的选择排序算法：对起始于位置p的n个元素排序 |
| 2 **void** List<T>::selectionSort(ListNodePosi(T) p, **int** n) { //valid(p) && rank(p) + n <= size |
| 3 ListNodePosi(T) head = p->pred; ListNodePosi(T) tail = p; |
| 4 **for** (int i = 0; i < n; i++) tail = tail->succ; //待排序区间为(head, tail) |
| 5 **while** (1 < n) { //在至少还剩两个节点之前，在待排序区间内 |
| 6 ListNodePosi(T) max = selectMax(head->succ, n); //找出最大者（歧义时后者优先） |
| 7 insertBefore(tail, remove(max)); //将其移至无序区间末尾（作为有序区间新的首元素） |
| 8 tail = tail->pred; n--; |
| 9 } |
| 10 } //复杂度Θ(n^2) |

# 向量

## 1.复制构造

|  |
| --- |
| 1 **template** <**typename** T> //元素类型 |
| 2 **void** Vector<T>::copyFrom(T **const\*** A, Rank lo, Rank hi) { //以数组区间A[lo, hi)为蓝本复制向量 |
| 3 \_elem = **new** T[\_capacity = 2 \* (hi - lo)]; \_size = 0; //分配空间，规模清零 |
| 4 **while** (lo < hi) //A[lo, hi)内的元素逐一 |
| 5 \_elem[\_size++] = A[lo++]; //复制至\_elem[0, hi - lo) |
| 6 } //复杂度O(n)或O(\_size) |

## 2.重载向量赋值操作符

|  |
| --- |
| 1 **template** <**typename** T> Vector<T>& Vector<T>::operator=(Vector<T> **const&** V ) { //重载赋值操作符 |
| 2 **if** (\_elem) **delete** [] \_elem; //释放原有内容 |
| 3 copyFrom(V.\_elem, 0, V.size()); //整体复制 |
| 4 **return** \*this; //返回当前对象的引用，以便链式赋值 |
| 5 } |

## 3. 扩容

|  |
| --- |
| 1 **template** <**typename** T> **void** Vector<T>::expand() { //向量空间不足时扩容 |
| 2 **if** (\_size < \_capacity) **return;** //尚未满员时，不必扩容 |
| 3 **if** (\_capacity < DEFAULT\_CAPACITY) \_capacity = DEFAULT\_CAPACITY; //不低于最小容量 |
| 4 T\* oldElem = \_elem; \_elem = **new** T[\_capacity <<= 1]; //容量加倍 |
| 5 **for** (int i = 0; i < \_size; i++) |
| 6 \_elem[i] = oldElem[i]; //复制原向量内容（T为基本类型，或已重载赋值操作符'='） |
| 7 **delete** [] oldElem; //释放原空间 |
| 8 } //单次复杂度O(n)，分摊复杂度O(1) |

## 4. 缩容

|  |
| --- |
| 1 **template** <**typename** T> **void** Vector<T>::shrink() { //装填因子过小时压缩向量所占空间 |
| 2 **if** (\_capacity < DEFAULT\_CAPACITY << 1) **return;** //不致收缩到DEFAULT\_CAPACITY以下 |
| 3 **if** (\_size << 2 > \_capacity) **return;** //以25%为界 |
| 4 T\* oldElem = \_elem; \_elem = **new** T[\_capacity >>= 1]; //容量减半 |
| 5 **for** (int i = 0; i < \_size; i++) \_elem[i] = oldElem[i]; //复制原向量内容 |
| 6 **delete** [] oldElem; //释放原空间 |
| 7 } //单次复杂度O(n)，分摊复杂度O(1) |

## 5.重载下标操作符[]

|  |
| --- |
| 1 **template** <**typename** T> T& Vector<T>::operator[](Rank r) **const** //重载下标操作符 |
| 2 { **return** \_elem[r]; } // assert: 0 <= r < \_size |

## 6.置乱算法

|  |
| --- |
| 1 **template** <**typename** T> **void** permute(Vector<T>& V) { //随机置乱向量，使各元素等概率出现于每一位置 |
| 2 **for** (int i = V.size(); i > 0; i--) //自后向前 |
| 3 swap(V[i - 1], V[rand() % i]); //V[i - 1]与V[0, i)中某一随机元素交换 |
| 4 } |
| 1 **template** <**typename** T> **void** Vector<T>::unsort(Rank lo, Rank hi) { //等概率随机置乱向量区间[lo, hi) |
| 2 T\* V = \_elem + lo; //将子向量\_elem[lo, hi)视作另一向量V[0, hi - lo) |
| 3 **for** (Rank i = hi - lo; i > 0; i--) //自后向前 |
| 4 swap(V[i - 1], V[rand() % i]); //将V[i - 1]与V[0, i)中某一元素随机交换 |
| 5 } //复杂度O(n) |

## 7.无序向量的顺序查找接口find()

|  |
| --- |
| 1 **template** <**typename** T> //无序向量的顺序查找：返回最后一个元素e的位置；失败时，返回lo - 1 |
| 2 Rank Vector<T>::find(T **const&** e, Rank lo, Rank hi) **const** { //assert: 0 <= lo < hi <= \_size |
| 3 **while** ((lo < hi--) && (e != \_elem[hi])); //从后向前，顺序查找 |
| 4 **return** hi; //若hi < lo，则意味着失败；否则hi即命中元素的秩 |
| 5 } //复杂度O(n) |

## 8.向量元素插入接口insert()

|  |
| --- |
| 1 **template** <**typename** T> //将e作为秩为r元素插入 |
| 2 Rank Vector<T>::insert(Rank r, T **const&** e) { //assert: 0 <= r <= size |
| 3 expand(); //若有必要，扩容 |
| 4 **for** (int i = \_size; i > r; i--) \_elem[i] = \_elem[i-1]; //自后向前，后继元素顺次后移一个单元 |
| 5 \_elem[r] = e; \_size++; //置入新元素并更新容量 |
| 6 **return** r; //返回秩 |
| 7 } //复杂度O(n) |

## 9.向量元素删除或区间删除

|  |
| --- |
| 1 **template** <**typename** T> **int** Vector<T>::remove(Rank lo, Rank hi) { //删除区间[lo, hi) |
| 2 **if** (lo == hi) **return** 0; //出于效率考虑，单独处理退化情况，比如remove(0, 0) |
| 3 **while** (hi < \_size) \_elem[lo++] = \_elem[hi++]; //[hi, \_size)顺次前移hi - lo个单元 |
| 4 \_size = lo; //更新规模，直接丢弃尾部[lo, \_size = hi)区间 |
| 5 shrink(); //若有必要，则缩容 |
| 6 **return** hi - lo; //返回被删除元素的数目 |
| 7 } //复杂度O(n) |
| 1 **template** <**typename** T> T Vector<T>::remove(Rank r) { //删除向量中秩为r的元素，0 <= r < size |
| 2 T e = \_elem[r]; //备份被删除元素 |
| 3 remove(r, r + 1); //调用区间删除算法，等效于对区间[r, r + 1)的删除 |
| 4 **return** e; //返回被删除元素 |
| 5 } |

## 10.无序向量唯一化或清除重复元素或去重接口

|  |
| --- |
| 1 **template** <**typename** T> **int** Vector<T>::deduplicate() { //删除无序向量中重复元素（高效版） |
| 2 **int** oldSize = \_size; //记录原规模 |
| 3 Rank i = 1; //从\_elem[1]开始 |
| 4 **while** (i < \_size) //自前向后逐一考查各元素\_elem[i] |
| 5 (find(\_elem[i], 0, i) < 0) ? //在其前缀中寻找与之雷同者（至多一个） |
| 6 i++ : remove(i); //若无雷同则向后调查下一个元素，否则删除雷同者 |
| 7 **return** oldSize - \_size; //向量规模变化量，即被删除元素总数 |
| 8 } //总体复杂度O(n^2)；迭代(n-1)次，每次用于find和remove的时间相加正好为O(n) |

## 11.遍历接口

|  |
| --- |
| 1 **template** <**typename** T> **void** Vector<T>::traverse(void (\*visit)(T&)) //利用函数指针机制的遍历 |
| 2 { **for** (int i = 0; i < \_size; i++) visit(\_elem[i]); } |
| 3 |
| 4 **template** <**typename** T> **template** <**typename** VST> //元素类型、操作器 |
| 5 **void** Vector<T>::traverse(VST& visit) //利用函数对象机制的遍历，或称仿函数 |
| 6 { **for** (int i = 0; i < \_size; i++) visit(\_elem[i]); } //使用仿函数的效果更好，复杂度O(n) |

## 12.有序性甄别

|  |
| --- |
| 1 **template** <**typename** T> **int** Vector<T>::disordered() **const** { //返回向量中逆序相邻元素对的总数 |
| 2 **int** n = 0; //计数器 |
| 3 **for** (int i = 1; i < \_size; i++) //逐一检查\_size - 1对相邻元素 |
| 4 **if** (\_elem[i - 1] > \_elem[i]) n++; //逆序则计数 |
| 5 **return** n; //向量有序当且仅当n = 0 |
| 6 } |

## 13.有序向量唯一化或去重

|  |
| --- |
| 1 **template** <**typename** T> **int** Vector<T>::uniquify() { //有序向量重复元素剔除算法（低效版） |
| 2 **int** oldSize = \_size; **int** i = 1; //当前比对元素的秩，起始于首元素 |
| 3 **while** (i < \_size) //从前向后，逐一比对各对相邻元素 |
| 4 \_elem[i - 1] == \_elem[i] ? remove(i) : i++; //若雷同，则删除后者；否则，转至后一元素 |
| 5 return oldSize - \_size; //向量规模变化量，即被删除元素总数 |
| 6 } //复杂度O(n^2) |
| 1 **template** <**typename** T> **int** Vector<T>::uniquify() { //有序向量重复元素剔除算法（高效版） |
| 2 Rank i = 0, j = 0; //各对互异“相邻”元素的秩 |
| 3 **while** (++j < \_size) //逐一扫描，直至末元素 |
| 4 **if** (\_elem[i] != \_elem[j]) //跳过雷同者 |
| 5 \_elem[++i] = \_elem[j]; //发现不同元素时，向前移至紧邻于前者右侧 |
| 6 \_size = ++i; shrink(); //直接截除尾部多余元素 |
| 7 **return** j - i; //向量规模变化量，即被删除元素总数 |
| 8 } //复杂度O(n) |

## 14.有序向量的查找

|  |
| --- |
| 1 // 二分查找算法（版本A）：在有序向量的区间[lo, hi)内查找元素e，0 <= lo <= hi <= \_size |
| 2 **template** <**typename** T> **static** Rank binSearch(T\* A, T **const&** e, Rank lo, Rank hi) { |
| 3 **while** (lo < hi) { //每步迭代可能要做两次比较判断，有三个分支 |
| 4 Rank mi = (lo + hi) >> 1; //以中点为轴点 |
| 5 **if** (e < A[mi]) hi = mi; //深入前半段[lo, mi)继续查找 |
| 6 **else if** (A[mi] < e) lo = mi + 1; //深入后半段(mi, hi)继续查找 |
| 7 **else return** mi; //在mi处命中 |
| 8 } //成功查找可以提前终止 |
| 9 **return** -1; //查找失败 |
| 10 } //有多个命中元素时，不能保证返回秩最大者；查找失败时，简单地返回-1，而不能指示失败的位置，复杂度O(logn)  11 //成功和失败查找长度均为O(1.5log2n) |
| 1 **#include** "..\fibonacci\Fib.h" //引入Fib数列类 |
| 2 // Fibonacci查找算法（版本A）：在有序向量的区间[lo, hi)内查找元素e，0 <= lo <= hi <= \_size |
| 3 **template** <**typename** T> **static** Rank fibSearch(T\* A, T **const&** e, Rank lo, Rank hi) { |
| 4 Fib fib(hi - lo); //用O(log\_phi(n = hi - lo))时间创建Fib数列 |
| 5 **while** (lo < hi) { //每步迭代可能要做两次比较判断，有三个分支 |
| 6 **while** (hi - lo < fib.get()) fib.prev(); //通过向前顺序查找（分摊O(1)）——至多迭代几次？ |
| 7 Rank mi = lo + fib.get() - 1; //确定形如Fib(k) - 1的轴点 |
| 8 **if** (e < A[mi]) hi = mi; //深入前半段[lo, mi)继续查找 |
| 9 **else if** (A[mi] < e) lo = mi + 1; //深入后半段(mi, hi)继续查找 |
| 10 **else return** mi; //在mi处命中 |
| 11 } //成功查找可以提前终止 |
| 12 **return** -1; //查找失败 |
| 13 } //有多个命中元素时，不能保证返回秩最大者；失败时，简单地返回-1，而不能指示失败的位置  14 //成功查找长度为O(1.44log2n) |
| 1 // 二分查找算法（版本B）：在有序向量的区间[lo, hi)内查找元素e，0 <= lo <= hi <= \_size |
| 2 **template** <**typename** T> **static** Rank binSearch(T\* A, T **const&** e, Rank lo, Rank hi) { |
| 3 **while** (1 < hi - lo) { //每步迭代仅需做一次比较判断，有两个分支；成功查找不能提前终止 |
| 4 Rank mi = (lo + hi) >> 1; //以中点为轴点 |
| 5 (e < A[mi]) ? hi = mi : lo = mi; //经比较后确定深入[lo, mi)或[mi, hi) |
| 6 } //出口时hi = lo + 1，查找区间仅含一个元素A[lo] |
| 7 **return** (e == A[lo]) ? lo : -1 ; //查找成功时返回对应的秩；否则统一返回-1 |
| 8 } //有多个命中元素时，不能保证返回秩最大者；查找失败时，简单地返回-1，而不能指示失败的位置 |
| 1 // 二分查找算法（版本C）：在有序向量的区间[lo, hi)内查找元素e，0 <= lo <= hi <= \_size |
| 2 **template** <**typename** T> **static** Rank binSearch(T\* A, T **const&** e, Rank lo, Rank hi) { |
| 3 **while** (lo < hi) { //每步迭代仅需做一次比较判断，有两个分支 |
| 4 Rank mi = (lo + hi) >> 1; //以中点为轴点 |
| 5 (e < A[mi]) ? hi = mi : lo = mi + 1; //经比较后确定深入[lo, mi)或(mi, hi) |
| 6 } //成功查找不能提前终止 |
| 7 **return** --lo; //循环结束时，lo为大于e的元素的最小秩，故lo - 1即不大于e的元素的最大秩 |
| 8 } //有多个命中元素时，总能保证返回秩最大者；查找失败时，能够返回失败的位置，复杂度同版本B |

# 列表

## 1.默认构造方法

|  |
| --- |
| 1 **template** <**typename** T> **void** List<T>::init() { //列表初始化，在创建列表对象时统一调用 |
| 2 header = **new** ListNode<T>; //创建头哨兵节点 |
| 3 trailer = **new** ListNode<T>; //创建尾哨兵节点 |
| 4 header->succ = trailer; header->pred = NULL; |
| 5 trailer->pred = header; trailer->succ = NULL; |
| 6 \_size = 0; //记录规模 |
| 7 } //复杂度O(1) |

## 2.重载列表类下标操作符

|  |
| --- |
| 1 **template** <**typename** T> //重载下标操作符，以通过秩直接访问列表节点（虽方便，效率低，需慎用） |
| 2 T& List<T>::operator[](Rank r) **const** { //assert: 0 <= r < size |
| 3 ListNodePosi(T) p = first(); //从首节点出发 |
| 4 **while** (0 < r--) p = p->succ; //顺数第r个节点即是 |
| 5 **return** p->data; //目标节点，返回其中所存元素 |
| 6 } //复杂度O(r) |

## 3.无序列表查找

|  |
| --- |
| 1 **template** <**typename** T> //在无序列表内节点p（可能是trailer）的n个（真）前驱中，找到等于e的最后者 |
| 2 ListNodePosi(T) List<T>::find(T **const&** e, **int** n, ListNodePosi(T) p) **const** { //0<=n<=rank(p)<\_size |
| 3 **while** (0 < n--) //对于p的最近的n个前驱，从右向左 |
| 4 **if** (e == (p = p->pred)->data) **return** p; //逐个比对，直至命中或范围越界 |
| 5 **return** NULL; //p越出左边界意味着区间内不含e，查找失败 |
| 6 } //失败时，返回NULL，复杂度O(n) |

## 4.列表节点插入接口

|  |
| --- |
| 1 **template** <**typename** T> ListNodePosi(T) List<T>::insertAsFirst(T **const&** e) |
| 2 { \_size++; **return** header->insertAsSucc(e); } //e当作首节点插入 |
| 3 |
| 4 **template** <**typename** T> ListNodePosi(T) List<T>::insertAsLast(T **const&** e) |
| 5 { \_size++; **return** trailer->insertAsPred(e); } //e当作末节点插入 |
| 6 |
| 7 **template** <**typename** T> ListNodePosi(T) List<T>::insertBefore(ListNodePosi(T) p, T **const&** e) |
| 8 { \_size++; **return** p->insertAsPred(e); } //e当作p的前驱插入 |
| 9 |
| 10 **template** <**typename** T> ListNodePosi(T) List<T>::insertAfter(ListNodePosi(T) p, T **const&** e) |
| 11 { \_size++; **return** p->insertAsSucc(e); } //e当作p的后继插入 |
| 1 **template** <**typename** T> //将e紧靠当前节点之前插入于当前节点所属列表（设有哨兵头节点header） |
| 2 ListNodePosi(T) ListNode<T>::insertAsPred(T **const&** e) { |
| 3 ListNodePosi(T) x = **new** ListNode(e, pred, **this);** //创建新节点 |
| 4 pred->succ = x; pred = x; //设置正向链接 |
| 5 **return** x; //返回新节点的位置 |
| 6 } |
| 1 **template** <**typename** T> //将e紧随当前节点之后插入于当前节点所属列表（设有哨兵尾节点trailer） |
| 2 ListNodePosi(T) ListNode<T>::insertAsSucc(T **const&** e) { |
| 3 ListNodePosi(T) x = **new** ListNode(e, **this,** succ); //创建新节点 |
| 4 succ->pred = x; succ = x; //设置逆向链接 |
| 5 **return** x; //返回新节点的位置 |
| 6 } //单纯的插入操作复杂度均为O(1) |

## 5.基于复制的构造

|  |
| --- |
| 1 **template** <**typename** T> //列表内部方法：复制列表中自位置p起的n项 |
| 2 **void** List<T>::copyNodes(ListNodePosi(T) p, **int** n) { //p合法，且至少有n-1个真后继节点 |
| 3 init(); //创建头、尾哨兵节点并做初始化 |
| 4 **while(n--)** { insertAsLast(p->data); p = p->succ; } //将起自p的n项依次作为末节点插入 |
| 5 } //复杂度O(n) |
| 1 **template** <**typename** T> //assert: p为合法位置，且至少有n-1个后继节点 |
| 2 List<T>::List(ListNodePosi(T) p, **int** n) { copyNodes(p, n); } //复制列表中自位置p起的n顷 |
| 3 |
| 4 **template** <**typename** T> |
| 5 List<T>::List(List<T> **const&** L) { copyNodes(L.first(), L.\_size); } //整体复制列表L |
| 6 |
| 7 **template** <**typename** T> //assert: r+n <= L.\_size |
| 8 List<T>::List(List<T> **const&** L, **int** r, **int** n) { copyNodes(L[r], n); } //复制L中自第r项起的n项 复杂度O(r+n+1) |

## 6.删除

|  |
| --- |
| 1 **template** <**typename** T> T List<T>::remove(ListNodePosi(T) p) { //删除合法位置p处节点，返回其数值 |
| 2 T e = p->data; //备份待删除节点的数值（假定T类型可直接赋值） |
| 3 p->pred->succ = p->succ; p->succ->pred = p->pred; //后继、前驱 |
| 4 **delete** p; \_size--; //释放节点，更新规模 |
| 5 **return** e; //返回备份的数值 |
| 6 } //复杂度O(1) |

## 7.析构

|  |
| --- |
| 1 **template** <**typename** T> List<T>::~List() //列表析构器 |
| 2 { clear(); **delete** header; **delete** trailer; } //清空列表，释放头、尾哨兵节点 |
| 1 **template** <**typename** T> **int** List<T>::clear() { //清空列表 |
| 2 **int** oldSize = \_size; |
| 3 **while** (0 < \_size) remove(header->succ); //反复删除首节点，直至列表变空 |
| 4 **return** oldSize; |
| 5 } //复杂度O(n) |

## 8.无序列表的唯一化或去重接口

|  |
| --- |
| 1 **template** <**typename** T> **int** List<T>::deduplicate() { //剔除无序列表中的重复节点 |
| 2 **if** (\_size < 2) **return** 0; //平凡列表自然无重复 |
| 3 **int** oldSize = \_size; //记录原规模 |
| 4 ListNodePosi(T) p = header; Rank r = 0; //p从首节点开始 |
| 5 **while** (trailer != (p = p->succ)) { //依次直到末节点 |
| 6 ListNodePosi(T) q = find(p->data, r, p); //在p的r个（真）前驱中查找雷同者 |
| 7 q ? remove(q) : r++; //若的确存在，则删除之；否则秩加一 |
| 8 } //assert: 循环过程中的任意时刻，p的所有前驱互不相同 |
| 9 **return** oldSize - \_size; //列表规模变化量，即被删除元素总数 |
| 10 } //复杂度O(n^2) |

## 9.遍历

|  |
| --- |
| 1 **template** <**typename** T> **void** List<T>::traverse(void (\*visit)(T&)) //利用函数指针机制的遍历 |
| 2 { **for** (ListNodePosi(T) p = header->succ; p != trailer; p = p->succ) visit(p->data); } |
| 3 |
| 4 **template** <**typename** T> **template** <**typename** VST> //元素类型、操作器 |
| 5 **void** List<T>::traverse(VST& visit) //利用函数对象机制的遍历 |
| 6 { **for** (ListNodePosi(T) p = header->succ; p != trailer; p = p->succ) visit(p->data); } |

## 10.有序列表唯一化

|  |
| --- |
| 1 **template** <**typename** T> **int** List<T>::uniquify() { //成批剔除重复元素，效率更高 |
| 2 **if** (\_size < 2) **return** 0; //平凡列表自然无重复 |
| 3 **int** oldSize = \_size; //记录原规模 |
| 4 ListNodePosi(T) p; ListNodePosi(T) q; //依次指向紧邻的各对节点 |
| 5 **for** (p = header, q = p->succ; trailer != q; p = q, q = q->succ) //从自左向右扫描 |
| 6 **if** (p->data == q->data) { remove(q); q = p; } //若p和q雷同，则删除后者 |
| 7 **return** oldSize - \_size; //列表规模变化量，即被删除元素总数 |
| 8 } |

## 11.有序列表查找

|  |
| --- |
| 1 **template** <**typename** T> //在有序列表内节点p（可能是trailer）的n个（真）前驱中，找到不大于e的最后者 |
| 2 ListNodePosi(T) List<T>::search(T **const&** e, **int** n, ListNodePosi(T) p) **const** { |
| 3 // assert: 0 <= n <= rank(p) < \_size |
| 4 **while** (0 <= n--) //对于p的最近的n个前驱，从右向左逐个比较 |
| 5 **if** (((p = p->pred)->data) <= e) **break;** //直至命中、数值越界或范围越界 |
| 6 // assert: 至此位置p必符合输出语义约定——尽管此前最后一次关键码比较可能没有意义（等效于与-inf比较） |
| 7 **return** p; //返回查找终止的位置 |
| 8 } //失败时，返回区间左边界的前驱（可能是header）——调用者可通过valid()判断成功与否 复杂度O(n) |

## 12.列表最大节点/最大值定位

|  |
| --- |
| 1 **template** <**typename** T> //从起始于位置p的n个元素中选出最大者 |
| 2 ListNodePosi(T) List<T>::selectMax(ListNodePosi(T) p, **int** n) { |
| 3 ListNodePosi(T) max = p; //最大者暂定为首节点p |
| 4 **for** (ListNodePosi(T) cur = p; 1 < n; n--) //从首节点p出发，将后续节点逐一与max比较 |
| 5 **if** (!lt((cur = cur->succ)->data, max->data)) //若当前元素不小于max，则 |
| 6 max = cur; //更新最大元素位置记录 |
| 7 **return** max; //返回最大节点位置 |
| 8 } //复杂度O(n) |

# 二叉树

## 1.孩子节点插入接口

|  |
| --- |
| 1 **template** <**typename** T> //将e作为当前节点的左孩子插入二叉树 |
| 2 BinNodePosi(T) BinNode<T>::insertAsLC(T **const&** e) { **return** lChild = **new** BinNode(e, **this);** } |
| 3 |
| 4 **template** <**typename** T> //将e作为当前节点的右孩子插入二叉树 |
| 5 BinNodePosi(T) BinNode<T>::insertAsRC(T **const&** e) { **return** rChild = **new** BinNode(e, **this);** } |

## 2.二叉树节点的高度更新

|  |
| --- |
| 1 **template** <**typename** T> **int** BinTree<T>::updateHeight(BinNodePosi(T) x) //更新节点x高度 |
| 2 { **return** x->height = 1 + max(stature(x->lChild), stature(x->rChild)); } //具体规则因树不同而异 |
| 3 |
| 4 **template** <**typename** T> **void** BinTree<T>::updateHeightAbove(BinNodePosi(T) x) //更新v及祖先的高度 |
| 5 { **while** (x) { updateHeight(x); x = x->parent; } } //可优化：一旦高度未变，即可终止 |

## 3.树根节点和节点插入

|  |
| --- |
| 1 **template** <**typename** T> BinNodePosi(T) BinTree<T>::insertAsRoot(T **const&** e) |
| 2 { \_size = 1; **return** \_root = **new** BinNode<T>(e); } //将e当作根节点插入空的二叉树 |
| 3 |
| 4 **template** <**typename** T> BinNodePosi(T) BinTree<T>::insertAsLC(BinNodePosi(T) x, T **const&** e) |
| 5 { \_size++; x->insertAsLC(e); updateHeightAbove(x); **return** x->lChild; } //e插入为x的左孩子 |
| 6 |
| 7 **template** <**typename** T> BinNodePosi(T) BinTree<T>::insertAsRC(BinNodePosi(T) x, T **const&** e) |
| 8 { \_size++; x->insertAsRC(e); updateHeightAbove(x); **return** x->rChild; } //e插入为x的右孩子 |

## 4.子树接入

|  |
| --- |
| 1 **template** <**typename** T> //二叉树子树接入算法：将S当作节点x的左子树接入，S本身置空 |
| 2 BinNodePosi(T) BinTree<T>::attachAsLC(BinNodePosi(T) x, BinTree<T>\* &S) { //x->lChild == NULL |
| 3 **if** (x->lChild = S->\_root) x->lChild->parent = x; //接入 |
| 4 \_size += S->\_size; updateHeightAbove(x); //更新全树规模与x所有祖先的高度 |
| 5 S->\_root = NULL; S->\_size = 0; release(S); S = NULL; **return** x; //释放原树，返回接入位置 |
| 6 } |
| 7 |
| 8 **template** <**typename** T> //二叉树子树接入算法：将S当作节点x的右子树接入，S本身置空 |
| 9 BinNodePosi(T) BinTree<T>::attachAsRC(BinNodePosi(T) x, BinTree<T>\* &S) { //x->rChild == NULL |
| 10 **if** (x->rChild = S->\_root) x->rChild->parent = x; //接入 |
| 11 \_size += S->\_size; updateHeightAbove(x); //更新全树规模与x所有祖先的高度 |
| 12 S->\_root = NULL; S->\_size = 0; release(S); S = NULL; **return** x; //释放原树，返回接入位置 |
| 13 } |

## 5.子树删除

|  |
| --- |
| 1 **template** <**typename** T> //删除二叉树中位置x处的节点及其后代，返回被删除节点的数值 |
| 2 **int** BinTree<T>::remove(BinNodePosi(T) x) { //assert: x为二叉树中的合法位置 |
| 3 FromParentTo(\*x) = NULL; //切断来自父节点的指针 |
| 4 updateHeightAbove(x->parent); //更新祖先高度 |
| 5 **int** n = removeAt(x); \_size -= n; **return** n; //删除子树x，更新规模，返回删除节点总数 |
| 6 } |
| 7 |
| 8 **template** <**typename** T> //删除二叉树中位置x处的节点及其后代，返回被删除节点的数值 |
| 9 **static int** removeAt(BinNodePosi(T) x) { //assert: x为二叉树中的合法位置 |
| 10 **if** (!x) **return** 0; //递归基：空树 |
| 11 **int** n = 1 + removeAt(x->lChild) + removeAt(x->rChild); //递归释放左、右子树 |
| 12 release(x->data); release(x); **return** n; //释放被摘除节点，并返回删除节点总数 |
| 13 } |

## 6.子树分离

|  |
| --- |
| 1 **template** <**typename** T> //二叉树子树分离算法：将子树x从当前树中摘除，将其封装为一棵独立子树返回 |
| 2 BinTree<T>\* BinTree<T>::secede(BinNodePosi(T) x) { //assert: x为二叉树中的合法位置 |
| 3 FromParentTo(\*x) = NULL; //切断来自父节点的指针 |
| 4 updateHeightAbove(x->parent); //更新原树中所有祖先的高度 |
| 5 BinTree<T>\* S = **new** BinTree<T>; S->\_root = x; x->parent = NULL; //新树以x为根 |
| 6 S->\_size = x->size(); \_size -= S->\_size; **return** S; //更新规模，返回分离出来的子树 |
| 7 } //除了更新祖先高度和释放节点等操作，只需常数时间 |

## 7.先序遍历

|  |
| --- |
| 1 **template** <**typename** T, **typename** VST> //元素类型、操作器 |
| 2 **void** travPre\_R(BinNodePosi(T) x, VST& visit) { //二叉树先序遍历算法（递归版） |
| **3 if** (!x) **return;** |
| 4 visit(x->data); |
| 5 travPre\_R(x->lChild, visit); |
| 6 travPre\_R(x->rChild, visit); |
| 7 } |
| 1 //从当前节点出发，沿左分支不断深入，直至没有左分支的节点；沿途节点遇到后立即访问 |
| 2 **template** <**typename** T, **typename** VST> //元素类型、操作器 |
| 3 **static void** visitAlongLeftBranch(BinNodePosi(T) x, VST& visit, Stack<BinNodePosi(T)>& S) { |
| 4 **while** (x) { |
| 5 visit(x->data); //访问当前节点 |
| 6 S.push(x->rChild); //右孩子入栈暂存（可优化：通过判断，避免空的右孩子入栈） |
| 7 x = x->lChild; //沿左分支深入一层 |
| 8 } |
| 9 } |
| 10 |
| 11 **template** <**typename** T, **typename** VST> //元素类型、操作器 |
| 12 **void** travPre\_I2(BinNodePosi(T) x, VST& visit) { //二叉树先序遍历算法（迭代版#2） |
| 13 Stack<BinNodePosi(T)> S; //辅助栈 |
| 14 **while** (true) { |
| 15 visitAlongLeftBranch(x, visit, S); //从当前节点出发，逐批访问 |
| 16 **if** (S.empty()) **break;** //直到栈空 |
| 17 x = S.pop(); //弹出下一批的起点 |
| 18 } |
| 19 } |

## 8.后序遍历

|  |
| --- |
| 1 **template** <**typename** T, **typename** VST> //元素类型、操作器 |
| 2 **void** travPost\_R(BinNodePosi(T) x, VST& visit) { //二叉树后序遍历算法（递归版） |
| **3 if** (!x) **return;** |
| 4 travPost\_R(x->lChild, visit); |
| 5 travPost\_R(x->rChild, visit); |
| 6 visit(x->data); |
| 7 } |
| 1 **template** <**typename** T> //在以S栈顶节点为根的子树中，找到最高左侧可见叶节点 |
| 2 **static void** gotoHLVFL(Stack<BinNodePosi(T)>& S) { //沿途所遇节点依次入栈 |
| 3 **while** (BinNodePosi(T) x = S.top()) //自顶而下，反复检查当前节点（即栈顶） |
| 4 **if** (HasLChild(\*x)) { //尽可能向左 |
| 5 **if** (HasRChild(\*x)) S.push(x->rChild); //若有右孩子，优先入栈 |
| 6 S.push(x->lChild); //然后才转至左孩子 |
| 7 } **else** //实不得已 |
| 8 S.push(x->rChild); //才向右 |
| 9 S.pop(); //返回之前，弹出栈顶的空节点 |
| 10 } |
| 11 |
| 12 **template** <**typename** T, **typename** VST> |
| 13 **void** travPost\_I(BinNodePosi(T) x, VST& visit) { //二叉树的后序遍历（迭代版） |
| 14 Stack<BinNodePosi(T)> S; //辅助栈 |
| 15 **if** (x) S.push(x); //根节点入栈 |
| 16 **while** (!S.empty()) { |
| 17 **if** (S.top() != x->parent) //若栈顶非当前节点之父（则必为其右兄），此时需 |
| 18 gotoHLVFL(S); //在以其右兄为根之子树中，找到HLVFL（相当于递归深入其中） |
| 19 x = S.pop(); visit(x->data); //弹出栈顶（即前一节点之后继），并访问之 |
| 20 } |
| 21 } |

## 9.中序遍历

|  |
| --- |
| 1 **template** <**typename** T, **typename** VST> //元素类型、操作器 |
| 2 **void** travIn\_R(BinNodePosi(T) x, VST& visit) { //二叉树中序遍历算法（递归版） |
| **3 if** (!x) **return;** |
| 4 travIn\_R(x->lChild, visit); |
| 5 visit(x->data); |
| 6 travIn\_R(x->rChild, visit); |
| 7 } |
| 1 **template** <**typename** T> //从当前节点出发，沿左分支不断深入，直至没有左分支的节点 |
| 2 **static void** goAlongLeftBranch(BinNodePosi(T) x, Stack<BinNodePosi(T)>& S) { |
| 3 **while** (x) { S.push(x); x = x->lChild; } //当前节点入栈后随即向左侧分支深入，迭代直到无左孩子 |
| 4 } |
| 5 |
| 6 **template** <**typename** T, **typename** VST> //元素类型、操作器 |
| 7 **void** travIn\_I1(BinNodePosi(T) x, VST& visit) { //二叉树中序遍历算法（迭代版#1） |
| 8 Stack<BinNodePosi(T)> S; //辅助栈 |
| 9 **while** (true) { |
| 10 goAlongLeftBranch(x, S); //从当前节点出发，逐批入栈 |
| 11 **if** (S.empty()) **break;** //直至所有节点处理完毕 |
| 12 x = S.pop(); visit(x->data); //弹出栈顶节点并访问之 |
| 13 x = x->rChild; //转向右子树 |
| 14 } |
| 15 } |
| 1 **template** <**typename** T, **typename** VST> //元素类型、操作器 |
| 2 **void** travIn\_I2(BinNodePosi(T) x, VST& visit) { //二叉树中序遍历算法（迭代版#2） |
| 3 Stack<BinNodePosi(T)> S; //辅助栈 |
| 4 **while** (true) |
| 5 **if** (x) { |
| 6 S.push(x); //根节点进栈 |
| 7 x = x->lChild; //深入遍历左子树 |
| 8 } **else if** (!S.empty()) { |
| 9 x = S.pop(); //尚未访问的最低祖先节点退栈 |
| 10 visit(x->data); //访问该祖先节点 |
| 11 x = x->rChild; //遍历祖先的右子树 |
| **12** } **else** |
| 13 **break;** //遍历完成 |
| 14 } |
| 1 **template** <**typename** T> BinNodePosi(T) BinNode<T>::succ() { //定位节点v的直接后继 |
| 2 BinNodePosi(T) s = **this;** //记录后继的临时发量 |
| 3 **if** (rChild) { //若有右孩子，则直接后继必在右子树中，具体地就是 |
| 4 s = rChild; //右子树中 |
| 5 **while** (HasLChild(\*s)) s = s->lChild; //最靠左（最小）的节点 |
| 6 } **else** { //否则，直接后继应是“将当前节点包含于其左子树中的最低祖先”，具体地就是 |
| 7 **while** (IsRChild(\*s)) s = s->parent; //逆向地沿右向分支，不断朝左上方移动 |
| 8 s = s->parent; //最后再朝右上方移动一步，即抵达直接后继（如果存在） |
| 9 } |
| 10 **return** s; |
| 11 } |
| 1 **template** <**typename** T, **typename** VST> //元素类型、操作器 |
| 2 **void** travIn\_I3(BinNodePosi(T) x, VST& visit) { //二叉树中序遍历算法（迭代版#3，无需辅助栈） |
| 3 **bool** backtrack = **false;** //前一步是否刚从右子树回溯——省去栈，仅O(1)辅助空间 |
| 4 **while** (true) |
| 5 **if** (!backtrack && HasLChild(\*x)) //若有左子树且不是刚刚回溯，则 |
| 6 x = x->lChild; //深入遍历左子树 |
| 7 **else** { //否则——无左子树或刚刚回溯（相当于无左子树） |
| 8 visit(x->data); //访问该节点 |
| 9 **if** (HasRChild(\*x)) { //若其右子树非空，则 |
| 10 x = x->rChild; //深入右子树继续遍历 |
| 11 backtrack = **false;** //并关闭回溯标志 |
| 12 } **else** { //若右子树空，则 |
| 13 **if** (!(x = x->succ())) **break;** //回溯（含抵达末节点时的退出返回） |
| 14 backtrack = **true;** //并设置回溯标志 |
| 15 } |
| 16 } |
| 17 } |

## 10.层次遍历

|  |
| --- |
| 1 **template** <**typename** T> **template** <**typename** VST> //元素类型、操作器 |
| 2 **void** BinNode<T>::travLevel(VST& visit) { //二叉树层次遍历算法 |
| 3 Queue<BinNodePosi(T)> Q; //辅助队列 |
| 4 Q.enqueue(this); //根节点入队 |
| 5 **while** (!Q.empty()) { //在队列再次变空之前，反复迭代 |
| 6 BinNodePosi(T) x = Q.dequeue(); visit(x->data); //取出队首节点并访问之 |
| 7 **if** (HasLChild(\*x)) Q.enqueue(x->lChild); //左孩子入队 |
| 8 **if** (HasRChild(\*x)) Q.enqueue(x->rChild); //右孩子入队 |
| 9 } |
| 10 } |