## 时间复杂度与空间复杂度

时间复杂度：O(1)

Int i = 1; //执行一次

Int n = 1; //执行一次

T(n) = 1 + 1 = 2 = O(1)

时间复杂度：O(n)

Int i = 1; //执行1次

While(i <= n) //执行n次

{

I++; //执行n次

}

T(n) = 1 + n + n = 2n + 1 = O(n)

时间复杂度：O(logn) log:以2为底的。

int i = 1;

While(i <= n)//执行logn次

{

i = i \* 2; //执行logn次

Printf(“sada”); //执行logn次

}

T(n) = logn+logn + logn= 3logn = O(logn)

时间复杂度: O(nlogn)

Int num1, num2; //各执行1次

For (int i = 0; i < n; i++) // int i = 0;执行n次 i < n;执行n次 i++执行n次

{

Num1+=1; //执行n次

For (int j = 1; j < n; j\*=2) //int j =1执行n次, j<n; 执行n\*logn次 j\*=2 执行n\*logn次

{

Num 2 += num1; //执行n\*logn次

}

}

T(n) = 1 + 1 + n + n + n + n + n\*logn + n\*logn +nl\*ogn = 2 + 4n + 3n\*logn = O(nlogn)

## 链表

### 单向链表

介绍:

应用范围:

代码示例:

template <class T>

class ChainNode

{

public:

T m\_data;

ChainNode<T> \*m\_link;

};

template <class T>

class Chain

{

public:

Chain(){m\_first = 0;}

~Chain();

bool IsEmpty() const {return m\_first == 0;}

bool Find(int nPos, T& data)const;

int Length() const;

int Search(const T data)const;

bool Delete(int nPos, T& data);

void InsertSort(const T data);

bool Insert(int nPos, const T data);

void Output(ostream& out)const;

void Reverse(void);

private:

ChainNode<T> \*m\_first; //指向第一个节点的指针

};

template<class T>

Chain<T>::~Chain()

{

ChainNode<T> \*next; //下一个节点

while(m\_first)

{

next = m\_first->m\_link; //保存下一个节点

delete m\_first; //删除第一个节点

m\_first = next; //把下一个节点当成第一个节点

}

}

template<class T>

int Chain<T>::Length() const

{

ChainNode<T>\* current = m\_first;

int nLen = 0;

while(current)

{

nLen++;

current = current->m\_link;

}

return nLen;

}

template <class T>

int Chain<T>::Search(const T data)const

{

ChainNode<T>\* current = m\_first;

int nIndex = 1; //current的索引

while(current && (current->m\_data != data) )

{

current = current->m\_link;

nIndex++;

}

if (current)

return nIndex;

return 0;

}

template<class T>

bool Chain<T>::Find(int nPos, T& data)const

{

if (nPos < 1)

{

return false;

}

ChainNode<T> \*current = m\_first;

while(nPos-1 && current)

{

current = current->m\_link;

nPos--;

}

if (current)

{

data = current->m\_data;

return true;

}

return false; //不存在该元素

}

//在第nPos之后插入,若nPos = 0;则每次在头插入

template<class T>

bool Chain<T>::Insert(int nPos, const T data)

{

if (nPos < 0)

{

return false;

}

ChainNode<T> \*p = m\_first;

for (int nIndex = 1; nIndex < nPos && p; nIndex++)

{

p = p->m\_link;

}

if (NULL == p && nPos > 0)

{

return false; //没有第nPos位置

}

ChainNode<T>\* newNode = new ChainNode<T>;

newNode->m\_data = data;

newNode->m\_link = NULL;

if(nPos)

{

newNode->m\_link = p->m\_link;

p->m\_link = newNode;

}

else

{

newNode->m\_link = m\_first;

m\_first = newNode;

}

return true;

}

template<class T>

bool Chain<T>::Delete(int nPos, T& data)

{

if(nPos < 1 || !m\_first)

{

return false;

}

ChainNode<T> \*pDelete = NULL;

if (1 == nPos)

{

m\_first = m\_first->m\_link;

}

else

{

ChainNode<T> \*q = m\_first;

for (int nIndex = 1; nIndex < nPos-1 && q; nIndex++)

{

q = q->m\_link; //遍历到要删除节点的前一个节点

}

if (NULL == q || NULL == q->m\_link)

{

return false; //不存在

}

pDelete = q->m\_link;

q->m\_link = pDelete->m\_link;

data = pDelete->m\_data;

delete pDelete;

}

return true;

}

template<class T>

void Chain<T>::Reverse(void)

{

ChainNode<T> \*pBefor = m\_first; //颠倒前的头指针

ChainNode<T> \*pAfter = NULL; //颠倒后的头指针

ChainNode<T> \*pTemp = NULL;

while(NULL != pBefor)

{

pTemp = pBefor->m\_link;

pBefor->m\_link = pAfter; //颠倒往前插入

pAfter = pBefor;

pBefor = pTemp;

}

m\_first = pAfter;

}

template<class T>

void Chain<T>::InsertSort(const T data) //插入排序(升序)

{

ChainNode<T>\* pNewNode = new ChainNode<T>;

ChainNode<T>\* pPos = NULL; //要插入该位置的后面

ChainNode<T>\* pHead = m\_first;

pNewNode->m\_data = data;

pNewNode->m\_link = NULL;

for (pHead; pHead && (pHead->m\_data < data); pHead = pHead->m\_link)

{

pPos = pHead;

}

if (pHead == m\_first)

{

m\_first = pNewNode;

}

else

{

pPos->m\_link = pNewNode;

}

pNewNode->m\_link = pHead;

}

template<class T>

void Chain<T>::BinSort(int nSortMax) //箱子排序,参数是排序中最大的数

{

int nNumber = 0; //箱子序号

ChainNode<T>\*\* ppBottom = new ChainNode<T>\*[nSortMax+1];

ChainNode<T>\*\* ppTop = new ChainNode<T>\*[nSortMax+1];

//初始化箱子

for (nNumber = 0; nNumber < nSortMax + 1; nNumber++)

{

ppBottom[nNumber] = NULL;

}

//将数据相同的节点放入同一箱子,每个箱子是一个链表

for (; m\_first; m\_first = m\_first->m\_link)

{

nNumber = m\_first->m\_data;

if (ppBottom[nNumber])

{

ppTop[nNumber]->m\_link = m\_first;

ppTop[nNumber] = m\_first;

}

else

{

ppBottom[nNumber] = ppTop[nNumber] = m\_first;

}

}

//将所有箱子进行排序

ChainNode<T> \* TempChest = NULL;

for (nNumber = 0; nNumber < nSortMax + 1; nNumber++)

{

if (ppBottom[nNumber])

{

if (TempChest)

{

TempChest->m\_link = ppBottom[nNumber];

}

else

{

m\_first = ppBottom[nNumber]; //根据for循环从小到大排序

}

TempChest = ppTop[nNumber];

}

}

delete[] ppBottom;

delete[] ppTop;

}

template<class T>

void Chain<T>::Output(ostream& out)const

{

ChainNode<T> \*current;

for (current = m\_first; current; current = current->m\_link)

{

out << current->m\_data << " ";

}

}

### 双向链表

介绍:

应用范围:

代码示例:

class List

{

public: // 构造函数中初始化为空链表

List (void) : m\_head (NULL), m\_tail (NULL) {}

// 析构函数中销毁剩余的节点

~List (void)

{

for (Node\* next; m\_head; m\_head = next)

{

next = m\_head -> m\_next;

delete m\_head;

}

}

// 追加

void append (int data)

{

m\_tail = new Node (data, m\_tail);

if (m\_tail -> m\_prev)

m\_tail -> m\_prev -> m\_next = m\_tail;

else

m\_head = m\_tail;

}

// 插入

void insert (size\_t index, int data)

{

for (Node\* find = m\_head; find;find = find -> m\_next)

if (index-- == 0)

{

Node\* node = new Node (data,find -> m\_prev, find);

if (node -> m\_prev)

node -> m\_prev -> m\_next = node;

else

m\_head = node;

node -> m\_next -> m\_prev = node;

return;

}

throw OverBound ();

}

// 删除

void remove (size\_t index)

{

for (Node\* find = m\_head; find; find = find -> m\_next)

if (index-- == 0)

{

if (find -> m\_prev)

find -> m\_prev -> m\_next =find -> m\_next;

else //删除的第一个

m\_head = find -> m\_next;

if (find -> m\_next)

find -> m\_next -> m\_prev =find -> m\_prev;

else//删除的最后一个

m\_tail = find -> m\_prev;

delete find;

return;

}

throw OverBound ();

}

// 正向遍历

void forward (void) const

{

for (Node\* node = m\_head; node; node = node -> m\_next)

cout << node -> m\_data << ' ';

cout << endl;

}

// 反向遍历

void backward (void) const

{

for (Node\* node = m\_tail; node;node = node -> m\_prev)

cout << node -> m\_data << ' ';

cout << endl;

}

// 伪随机访问

int& operator[] (size\_t index)

{

for (Node\* find = m\_head; find;

find = find -> m\_next)

if (index-- == 0)

return find -> m\_data;

throw OverBound ();

}

const int& operator[] (size\_t index) const

{

for (Node\* find = m\_head; find; find = find -> m\_next)

if (index-- == 0)

return find -> m\_data;

throw OverBound ();

}

private:

// 越界异常

class OverBound :

public exception

{

public:

const char\* what (void) const throw ()

{

return "链表越界";

}

};

// 节点类

class Node

{

public:

Node (int data = 0, Node\* prev = NULL,Node\* next = NULL) : m\_data (data),m\_prev (prev), m\_next (next) {}

int m\_data; // 节点数据

Node\* m\_prev; // 前向指针

Node\* m\_next; // 后向指针

};

Node\* m\_head;

// 头节点

Node\* m\_tail; // 尾节点

};

int main (void)

{

try

{

List list;

list.append (10);

list.append (30);

list.append (50);

list.append (60);

list.append (80);

list.forward ();

list.insert (1, 20);

list.insert (3, 40);

list.insert (6, 70);

list.backward ();

list.remove (7);

list.remove (6);

list.remove (5);

list.forward ();

list[0]++;

list[1] = 100;

list[2] += 30;

list.forward ();

const List& r = list;

cout << r[3] << endl;//

r[3]--;

}

catch (exception& ex)

{

cout << ex.what () << endl;

return -1;

}

return 0;

}

## 树

## 图

## 排序

### 冒泡排序

(升序)思路:两两比较相邻(与交换排序的区别)。{5,2,3,9,1}->{5,2,3,1,9}->{5,2,1,3,9}->{5,1,2,3,9}->{1,5,2,3,9}

效率:已经排好序时，时间复杂度为O(n)，但需在增加个变量判断来退出第一层循环。 逆序为O(n2);

代码示例:

void bubble (int data[], int size)

{

for (int i = 0; i < size - 1; i++) 需要对size-1个数进行两两相邻比较,例如3个数则需要比较2次。

{

bool ordered = true;

for (int j = 0; j < size - 1 - i; j++) //正向循环将最大值排到最后位置。

if (data[j+1] < data[j])

{

int temp = data[j];

data[j] = data[j+1];

data[j+1] = temp;

ordered = false;

}

if (ordered)

break;

}

}

### 选择排序

(升序)思路: 每次循环比较选出剩下数据中最小值的key，本轮循环结束再将当前值与选出来key的最小值进行交换。

效率:任何情况下时间复杂度都是O(n2)， 比较次数n2, 交换次数n-1; 由于交换次数少，略优于冒泡排序。

代码示例:

void select (int data[], int size)

{

for (int i = 0; i < size - 1; i++)

{

int min = i;

for (int j = i + 1; j < size; j++)

if (data[j] < data[min])

min = j;

if (min != i)

{

int temp = data[i];

data[i] = data[min];

data[min] = temp;

}

}

}

### 交换排序

(升序)思路:第一个与后面每一个进行比较， 小则交换， 第一轮下来最小肯定第一个。

效率:最简单且效率最低的排序,任何情况下时间复杂度都是O(n2);

### 插入排序

(升序)思路: {5,2,1,7,4,6}假设第1个数字为已排好序{5}, 将{2,1,7,4,6}这5个数字循环依次插入{5}序列中。

效率:已经是升序情况下， 已经已排好序的数字与变量的比较则不会进入循环中， 则时间复杂度为O(n)。逆序时则是O(n2),性能优于冒泡及选择。 适合较小数组排序。

代码示例:

void insert (int data[], int size)

{

for (int i = 1; i < size; i++) //从要排序的第2个数字插入到第1个数已排好的序中。

{

int temp = data[i]; //保存插入的数据，防止移动位置覆盖后找不到。

int j;

for (j = i; j > 0 && temp < data[j-1]; j--) //从后往前遍历已排好序的数组为了方便移动，若从前往后则需要一个变量保存移动前的后位置数据，否则就会被覆盖。Temp插入的数据最大则不会进入此循环，则效率最高。

data[j] = data[j-1];

if (j != i)

data[j] = temp; //找到位置进行插入

}

}

### 希尔排序

原理：通过跳跃循环（根据设定的增量）多做几次插入排序，使其元素基本有序， 当增量递减为1时做最后一次插入排序使其更少的进入内循环(也就是移动乱序的位置)，从而提升直接插入排序的效率。增量递减一般设置为“increment = increment/3+1”。

效率：跳跃循环(根据设定的增量)比较两个数的有序的几率大于顺序循环（增量1）,因此希尔排序效率高于直接插入排序。由于增量的选取不同，因此希尔排序是非稳定排序。

(初始增量为1的希尔排序就是直接插入排序,也就是稳定排序。)

最坏情况：每个子序列都需要进行移动后在插入。

最好情况：根据要排序的元素设定一个适合的增量。

时间复杂度:为nlogn。

### 快速排序

快速排序即为冒泡排序的升级版。

(升序)思路: 选定一个枢轴(一般为第一个)，进行一轮排序过后， 枢轴左边的数字都比枢轴小，右边的都比枢轴大，然后再对枢轴左边与右边分别进行相同的递归操作。每轮排序规则是从最后一个数与枢轴进行比较，若大于枢轴，则最后一个数的(高端)下标开始递减在与枢轴进行比较, 若小于枢轴则进行交换(优化后在循环中是赋值，即每轮排序就交换一次)；并且改为被交换的低端下标准备开始递增（比枢轴小）。若低端小标与高端下标重合则本轮排序结束;

效率:升序或逆序时的时间复杂度为O(n2)。 每轮排序后，所选定的枢轴都将排序的数字一分为二(即枢轴正好是该轮排序的中间值)，则时间复杂度为O(nlogn)。适合较大数组排序。

代码示例:

//int left:序列左边第一个位置

//int right:序列右边最后一个位置

void quick (int data[], int left, int right)

{

int p = (left + right) / 2;

int pivot = data[p];

for (int i = left, j = right; i < j;)

{

while (! (i >= p || pivot < data[i]))

i++;

if (i < p)

{

data[p] = data[i];

p = i;

}

while (! (j <= p || data[j] < pivot))

j--;

if (j > p)

{

data[p] = data[j];

p = j;

}

}

data[p] = pivot;

if (p - left > 1)

quick (data, left, p - 1);

if (right - p > 1)

quick (data, p + 1, right);

}

### 箱子排序

介绍:

将数据相同的内容放入同一个箱子， 然后对箱子排序。每个箱子里的内容顺序保持不变(稳定排序)。

使用场景:

代码示例:

## 相关练习题

### 二叉查找树根据某个值找路径

题目: 二叉查找树根据某个值找路径

解析:

代码:

template<class T>

class CNode

{

public:

CNode(T data) : m\_data(data), m\_pLeft(NULL), m\_pRight(NULL){}

~CNode(){}

T m\_data;

CNode<T>\* m\_pLeft;

CNode<T>\* m\_pRight;

};

template<class T>

class CBTree

{

public:

CBTree() : m\_pRoot(NULL){}

~CBTree(){}

void AddTreeNode(T data){ InserNode(m\_pRoot, data);}

void ShowPath(T data){GetNodePath(m\_pRoot, data);}

private:

void GetNodePath(CNode<T>\* root, T data);

void InserNode(CNode<T>\* &root, T data);

private:

CNode<T>\* m\_pRoot;

vector<T> m\_vecPath;

};

template<class T> void CBTree<T>::InserNode(CNode<T>\* &root,T data)

{

if (root == NULL)

{

root = new CNode<T>(data);

}

else

{

if (root->m\_data > data)

{

InserNode(root->m\_pLeft, data);

}

else

{

InserNode(root->m\_pRight, data);

}

}

}

template<class T> void CBTree<T>::GetNodePath(CNode<T>\* root, T data)

{

if (root == NULL)

{

return;

}

if (root->m\_data > data)

{

return;

}

if (root->m\_data <= data)

{

m\_vecPath.push\_back(root->m\_data);

}

if (root->m\_data == data)

{

for (size\_t i = 0; i < m\_vecPath.size(); i++)

cout << m\_vecPath[i] << " ";

cout << endl;

}

GetNodePath(root->m\_pLeft, data- root->m\_data);

GetNodePath(root->m\_pRight, data - root->m\_data);

m\_vecPath.pop\_back();

}

### 二叉查找树转换双向排序链表

题目: 输入一棵二元查找树，将该二元查找树转换成一个排序的双向链表。

要求不能创建任何新的结点，只调整指针的指向。输入一棵二元查找树，将该二元查找树转换成一个排序的双向链表。

要求不能创建任何新的结点，只调整指针的指向。

解析:

1.创建一个二元查找树。

2.对二元查找树做中序遍历。从最小或最大元素开始调整指针。

代码:

### 最小值的栈

题目: 设计包含min 函数的栈。

定义栈的数据结构，要求添加一个min 函数，能够得到栈的最小元素。

要求函数min、push 以及pop 的时间复杂度都是O(1)。

解析: 用一个变量记录最小元素。

代码:

template<class T>

struct data

{

T value;

T minValue;

};

template<class T, size\_t STACKSIZE>

class MyStack

{

public:

MyStack():m\_topPos(0), m\_stackSize(STACKSIZE){}

~MyStack(){}

T Min();

T Pop();

void Push(T value);

void Show();

private:

size\_t m\_topPos;

size\_t m\_stackSize;

data<T> m\_stack[STACKSIZE];

};

template<class T, size\_t STACKSIZE> T MyStack<T, STACKSIZE>::Min()

{

return m\_stack[m\_topPos-1].minValue;

}

template<class T, size\_t STACKSIZE> T MyStack<T, STACKSIZE>::Pop()

{

T tempValue;

if (m\_stackSize != 0)

{

tempValue = m\_stack[--m\_topPos].value;

}

return tempValue;

}

template<class T, size\_t STACKSIZE> void MyStack<T, STACKSIZE>::Push(T value)

{

T minValue;

if (m\_stackSize != 0)

{

minValue = m\_stack[m\_topPos-1].minValue < value ? m\_stack[m\_topPos-1].minValue : value;

}

else

{

minValue = value;

}

m\_stack[m\_topPos].minValue = minValue;

m\_stack[m\_topPos].value = value;

m\_topPos++;

}

template<class T, size\_t STACKSIZE> void MyStack<T, STACKSIZE>::Show()

{

for (size\_t i = 0; i < m\_topPos; i++)

{

cout << m\_stack[i].value << endl;

}

}

### 检测单链表是否存在环

题目:

写一算法检测单向链表中是否存在环(whether there is a loop in a link list),

要求算法复杂度(Algorithm's complexity是O(n)) 并只使用常数空间(space is O(c)).

注意，你只知道一个指向单向链表头的指针。链表的长度是不定的，而且环出现的地方也是不定的，环有可能在头，有可能在中间。而且要求是检测, 不能破坏环的结构.

解析:

方法一: 用两个步长分别为1和2的指针步进，相遇则证明有环 O(N)

方法二: 占用内存的思想，用一个指针前进，指针不断步进，但是链表占用的内存空间不再上升 O(N)

详细：已有的内存地址存储起来，哈希到一个bit上，假定cpu位数32，总共需要2^32的内存。

方法三: 将链表逆序，如果逆序后的头指针即为原来的头指针，则有环 O(N)

代码:

针对方法一:

bool check(const node\* head)

{

if(head==NULL) return false;

node \*low=head, \*fast=head->next;

while(fast!=NULL && fast->next!=NULL)

{

low=low->next;

fast=fast->next->next;

if(low==fast) return true;

}

return false;

}

题目:

解析:

代码: