**重言式判断实验报告**

1. **需求分析**

一个逻辑表达式如果对于其变元的任一种取值都为真，则称重言式；反之，如果对与其变元的任一种取值都为假，则称矛盾式；其他情形称为可满足式。编写一个程序，通过真值表判断一个逻辑表达式属于上述中的哪种情形。

逻辑运算符包括： '~', '&', '^', '|', '(', ')' 六种。

（1）逻辑表达式从终端输入，长度不超过一行，逻辑变元为大写字母。

（2）若是重言式或矛盾式，显示“True Forever”或“False Forever”，否则显示“Satisfiable”。

（3）本程序先使用辅助工作栈将逻辑二叉表达式的变量进行存储，然后将栈中的元素作为二叉树的结点结构，根据逻辑运算符的优先级顺序读取逻辑表达式以建立逻辑二叉表达式树，最后根据逻辑二叉表达式树对包括逻辑变量的逻辑表达式进行重言式的判别。

1. **概要设计**

1、为实现上述功能，需要定义两个抽象数据类型，如下：

（1）使用栈识别逻辑二叉表达式树

ADT：Sen\_Stack

基本操作：

void Init(int n);

操作结果：初始化一个空（操作）栈

void clear();

操作结果：清空栈中的所有元素

void Destroy();

操作结果：摧毁操作栈

int length();

操作结果：获取当前栈顶指针位置（当前栈内容量）

T Get\_top();

初始条件：存在一个非空的操作栈

操作结果：获取栈顶元素的值

bool push(T x)

初始条件：存在一个栈内空间仍有余量的操作栈

操作结果：插入元素x作为新的栈顶元素。

bool pop()

初始条件：存在一个非空的操作栈

操作结果：删除操作栈的栈顶元素

bool Sen\_Stack\_Empty();

操作结果：判断操作栈是否非空

受保护的成员：

T \*base;

int \_top;

int size;

} ADT Sen\_Stack

（2）二叉树的定义

ADT BTree

{数据对象D：D={ai | ai∈ElemSet, i=1,2, …,n, n≥0}

数据关系R：若D为空集，则称为空二叉树。

若D仅含一个数据元素，则R为空集，否则R＝｛H｝，H满足关系：

(1)T中**存在唯一的一个结点**，它没有前驱，称为树的根，*用root表示*，在集合D中用a1表示；

(2)若D中元素个数大于1，对于任意的数据元素aj∈D且j≥2，存在唯一的数据元素ai∈D，有<ai, aj>∈H；

(3)若D中元素个数大于1，对于任意的数据元素ai∈D，仅存在**不多于2个数据元素**aj,ak∈D且j, k≥i，有< ai, aj > , < ai, ak >∈H，其中，若j<k，则称aj为ai的左孩子节点，ak为ai的右孩子节点。

(4)对于任意的数据元素aj∈D且j≥2，存在D1∈D，D1={D|Dk∈ElemSet, k=1,2, …,m, m≥0}，有唯一的Pj={< , >, < , >, …, < , >, < , >}，这个集合Pj表示从根结点到结点aj的一条路径。

基本操作：

void InitBTree(TNode\* &T);

操作结果：初始化二叉树的准备工作

void DestroyBTree(TNode\* &T);

初始条件：二叉树已存在

操作结果：摧毁二叉树

bool Judge\_Punc(char &c);

操作结果：判断输入的是否是运算符，是返回true，否则返回false

bool Judge\_Compare(char &c1, char &c2);

操作结果：判断新进运算符与栈顶运算符之间的优先级，保持新进运算符的优先级比栈顶运算符的优先级高，弹出栈顶运算符就返回0（栈顶运算符优先级低），弹出新进运算符就返回1（栈顶运算符优先级高）

int Operator(int a, char punctutation, int b);

操作结果：计算特定表达式的值（a&b、a|b等）

void visit(TNode\* &T, char c);

操作结果：用于递归遍历二叉树访问结点的data

bool Pre\_Calc(TNode\* &T, int \*a);

操作结果：\*a本质是一个a[]数组，用于有序存放所有变元的0/1赋值结果，其中共有num个变元，所以a[]中共有2^num组0/1数据。bool Pre\_Calc(TNode\* &T, int \*a)函数计算表达式的真值情况，返回0/1，0代表假，1代表真。

void Calculation(TNode\* &T, int num);

操作结果：将表达式所有变元的0/1赋值结果都存入res int型变量中。如果是永真式，那res == 2^num；如果是永假式，那res == 0；否则就是可满足式。

TNode\* New\_Node(char c);

操作结果：在二叉树中新建一个结点

bool Build\_Tree(char \*s);

操作结果：构建逻辑表达式二叉树。先新建两个栈：用于存放操作数和操作符。再根据栈顶运算符是否是左括号，新进运算符是不是右括号（那就在二叉树中新建一个结点），新进运算符与栈顶运算符之间的优先级，新进输入的是操作数还是操作符，特殊处理取反符号以及判断栈空的条件，构建逻辑表达式二叉树。

void PreorderTraverse(TNode\* &T);

初始条件：二叉树存在

操作结果：先序递归遍历二叉树

void InorderTraverse(TNode\* &T);

初始条件：二叉树存在

操作结果：中序递归遍历二叉树

void PostorderTraverse(TNode\* &T);

初始条件：二叉树存在

操作结果：后序递归遍历二叉树

公有成员：

struct TNode{

TNode \*lchild;

TNode \*rchild;

TElemtype data;

}; //储存结点信息（左孩子、右孩子、）

char letter[200]; //用于储存表达式中的字母

int lettertag[260]; //变元映射(A->1, B->2, C->3)

int num; //变元个数:遍历逻辑二叉表达式树共需要判断 2^num 次

TNode\* root; //逻辑表达式二叉树根节点，存储完整的逻辑表达式

}ADT BTree

2.本程序主要分为六个模块：

（1）主程序模块

void main (){

初始化

While{

接受命令

处理命令}

}

（2）对操作栈的基本底层实现模块。

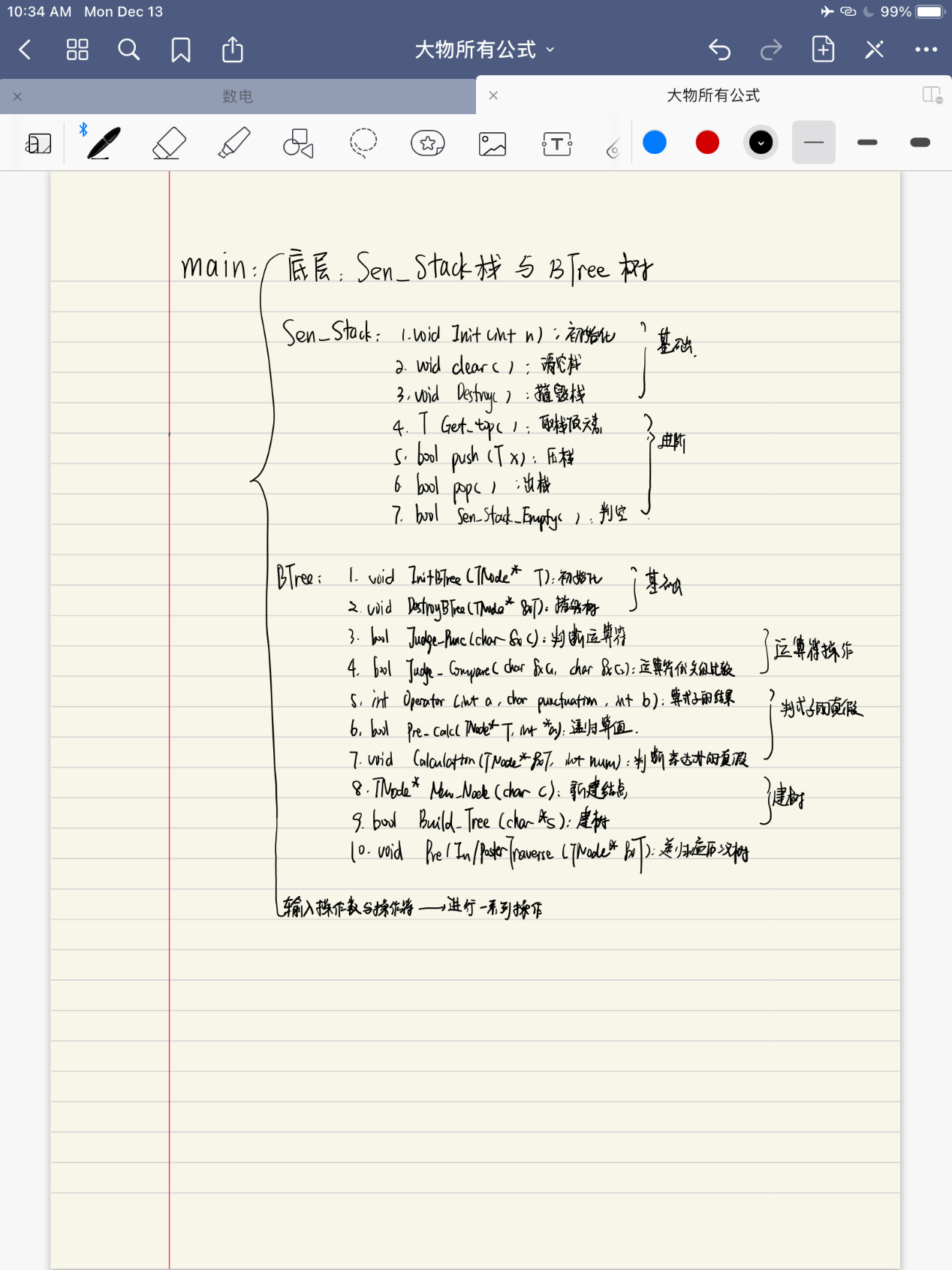
（3）逻辑表达式二叉树的建立模块。

（4）逻辑运算符的判断模块与优先级判别模块。

（5）逻辑表达式求真值模块。

（6）递归遍历二叉树模块。

3.各个模块之间的调用关系具体如下图：



1. **详细设计**

**1、数据储存结构**

**操作栈：**

T\* base;

int \_top; //栈顶指针的位置

int size; //栈的最大容量

**二叉树：**

struct TNode{

TNode \*lchild;

TNode \*rchild;

TElemtype data;

};

char letter[200]; //用于储存表达式中的字母

int lettertag[260];

int num; //变元个数:遍历逻辑二叉表达式树共需要判断 2^num 次

TNode\* root;

**2、逻辑表达式计算功能的实现**

**1、识别逻辑表达式并建立二叉树：**

TNode\* New\_Node(char c){

TNode\* p = new TNode;

p->data = c;

p->lchild = p->rchild = NULL;

return p;

}

bool Build\_Tree(char \*s){

int length = strlen(s);

Sen\_Stack<TNode\* > letterStack;

letterStack.Init(length);

Sen\_Stack<TNode\* > puncStack;

puncStack.Init(length);

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

if(Judge\_Punc(s[i])){

if(s[i]==')'){

while(!puncStack.Sen\_Stack\_Empty() && puncStack.Get\_top()->data != '('){

//栈顶不是左括号：结合一组操作数和操作符 把其作为操作数传入letterStack中

if(letterStack.length() < 2) return false;

//防止栈溢出

TNode \*p = puncStack.Get\_top();

puncStack.pop();

TNode \*p1 = letterStack.Get\_top();

letterStack.pop();

TNode \*p2 = letterStack.Get\_top();

letterStack.pop();

p->lchild = p2;

p->rchild = p1;

///此时p结点是一个combined letter

letterStack.push(p);

}

puncStack.pop(); //弹出左括号

}else{

//如果s[i]不是右括号 那就新建一个'根'结点

TNode \*q = New\_Node(s[i]);

if(s[i]=='('){

puncStack.push(q);

continue;

}

while(!puncStack.Sen\_Stack\_Empty() && Judge\_Compare(puncStack.Get\_top()->data, q->data)){

//比较新进运算符与栈顶运算符的优先级

//保持puncStack栈中的运算符按照优先级升幂排列

if(letterStack.length() < 2) return false;

TNode \*p = puncStack.Get\_top();

puncStack.pop();

TNode \*p1 = letterStack.Get\_top();

letterStack.pop();

TNode \*p2 = letterStack.Get\_top();

letterStack.pop();

p->lchild = p2;

p->rchild = p1;

letterStack.push(p);

}

puncStack.push(q); //把新运算符压进栈

continue;

}

}else{

//新传入的不是运算符 是变元

if(lettertag[s[i]] < 0){

lettertag[s[i]] = num++;

letter[num-1] = s[i];

//传入新的逻辑变量

}

TNode \*p = New\_Node(s[i]);

letterStack.push(p);

}

while(!puncStack.Sen\_Stack\_Empty() && puncStack.Get\_top()->data == '~'){

//处理特殊情况:取反符号

TNode \*p = puncStack.Get\_top();

puncStack.pop();

TNode \*p1 = letterStack.Get\_top();

letterStack.pop();

p->lchild = p1;

letterStack.push(p);

}

}

while(!puncStack.Sen\_Stack\_Empty()){

if(letterStack.length() < 2) return false;

TNode \*p = puncStack.Get\_top();

puncStack.pop();

TNode \*p1 = letterStack.Get\_top();

letterStack.pop();

TNode \*p2 = letterStack.Get\_top();

letterStack.pop();

p->lchild = p2;

p->rchild = p1;

letterStack.push(p);

}

root = letterStack.Get\_top();

return true;

//root现在储存的就是重言表达式

}

**2、判断逻辑表达式的真值**

int Operator(int a, char punctutation, int b){

int c;

switch(punctutation){

case'|':

c = a | b;

break;

case'&':

c = a & b;

break;

}

return c;

}

bool Pre\_Calc(TNode\* &T, int \*a){

if(~lettertag[T->data]) return a[lettertag[T->data]];

if(T){

if(T->data == '~') return !Pre\_Calc(T->lchild, a);

return Operator(Pre\_Calc(T->lchild, a), T->data, Pre\_Calc(T->rchild, a)); //递归

}

}

//用后序遍历的方式求二叉表达式树所述的表达式值

void Calculation(TNode\* &T, int num){

int a[40];

int res = 0;

for(int n = 0; n < 1 << num; n++){

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

a[i] = n >> i & 1;

res += Pre\_Calc(T, a);

}

if(res == (1<<num))

cout<<"True"<<endl;

else if(res == 0)

cout<<"False"<<endl;

else cout<<"Satisfactible"<<endl;

}

**3、递归遍历二叉树**

//先序递归遍历二叉树

void PreorderTraverse(TNode\* &T){

if(T){

if(T->data)

visit(T,T->data);

PreorderTraverse(T->lchild);

PreorderTraverse(T->rchild);

}

}

//中序递归遍历二叉树

void InorderTraverse(TNode\* &T){

if(T){

InorderTraverse(T->lchild);

if(T->data)

visit(T, T->data);

InorderTraverse(T->rchild);

}

}

//后序递归遍历二叉树

void PostorderTraverse(TNode\* &T){

if(T){

PostorderTraverse(T->lchild);

PostorderTraverse(T->rchild);

if(T->data)

visit(T, T->data);

}

**4、判断运算符及其优先级**

bool Judge\_Punc(char &c){

switch(c){

case'&':

case'|':

case'~':

case'(':

case')':

case'=': return true;

default: return false;

}

}

bool Judge\_Compare(char &c1, char &c2){

//char &c2 > char &c1

switch (c2){

case '|':

if(c1 == '(')

return 0;

else

return 1;

break;

case '^':

if(c1 == '(' || c1 == '|')

return 0;

else

return 1;

break;

case '&':

if(c1 == '(' || c1 == '|' || c1 == '^')

return 0;

else

return 1;

break;

case '~':

if(c1 == '(' || c1 == '|' || c1 =='^' || c1 == '&')

return 0;

else

return 1;

break;

case '(':

return 1;

break;

case ')':

return 0;

break;

}

}

**3、主程序**

cout << "---此程序以二叉链表为数据结构，实现逻辑表达式求值并判断类型---" << endl <<endl;

cout << "逻辑表达式中运算符为 ~、^、&、|、(、)，逻辑变量为大写字母。" << endl << endl;

char s[200];

while (scanf("%s", s) != EOF){

TreeList T;

T.InitBTree(T.root);

T.Build\_Tree(s);

int n = T.num;

T.Calculation(T.root, n);

T.DestroyBTree(T.root);}

1. **编程要点及调试分析**

1、建立逻辑表达式二叉树后，遍历所有变元逐个进行0/1赋值，最后通过真值判断即可实现重言式的判别。

2、逻辑符号的优先级判别：一定是新进的逻辑符优于栈顶逻辑符。

3、模块设计包括结构体模块、操作算法模块和主函数模块，其中操作算法模块主要包括了调用栈和建树的操作函数。

4、算法的时空分析：

本程序的主要算法为逻辑表达式二叉树的建立。由于二叉树的结点即栈中的元素，二叉树建立及栈的时间和空间复杂度都等同于其规模，即变量个数num，为O(num)。

5、本程序先使用栈将逻辑表达式的变量进行存储，然后将栈中的元素作为二叉树的结点结构，再根据优先级读取逻辑表达式建立二叉树，并通过对所有变元逐个进行0/1赋值以判断逻辑表达式的真值。调试代码顺利，测试结果符合预期。

1. **测试数据以及用户手册**

程序经Dev C++及Visual Studio Code等编译器编译，运行环境为Windows操作系统，进入程序运行后即交互显示文本方式的用户界面，用户使用过程可参照提示进行。

在进行测试时,执行main.cpp文件，代入测试数据后，程序的运行结果如下：

**测试数据：**

(1) (A|~A)&(B|~B)

(2) (A&~A)&C

(3) A|B|C|D|E|~A

(4) A&B&C&~B

(5) (A|B)&(A|~B)

(6) A&~B|~A&B

**输出结果:**

True

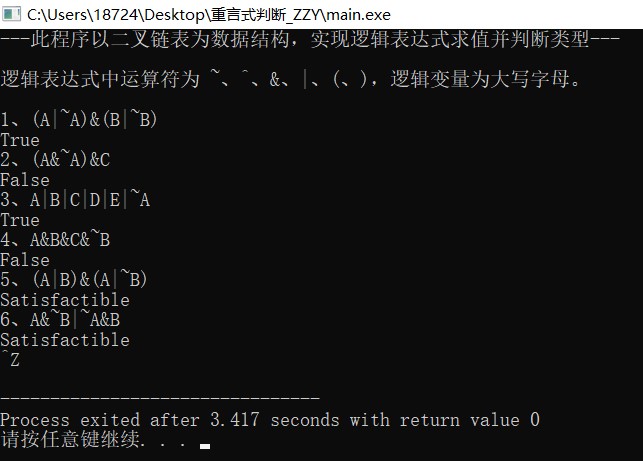
False

True

False

Satisfactible

Satisfactible



经检验，测试结果正确。

//补充：重言式判断的底层结构是用栈实现的，附件中提供了栈底层结构的实现代码。