DSA ASSIGNMENT 1

BT22CSH011

Devashish Athawale

Problem Statement:

Applications of Linked List - Polynomial representation and arithmetic operations

Q. Design and build a linked list allocation system to represent and manipulate polynomials. Use circular linked list with header nodes. Each term of the polynomial will be represented in a linked list node

structure as shown:

Write and test the following functions:

(a) Pread(): Read a polynomial and convert it to its circular representation. Return a pointer to the header

node of this polynomial.

(b) Pwrite(): Output the polynomial in its mathematical form.

(c) Padd(): Compute c=a+b. Do not change polynomials a and b.

(d) Psub(): Compute c=a-b. Do not change polynomials a and b.

(e) Pmult(): Compute c=a*b. Do not change polynomials a and b.

(f) Peval(): Evaluate the polynomials at some point a, where a is a floating point constant.

(g) Perase(): Erase a certain term of the polynomial. (circular linked list helps in this operation)**.**

Node class:

```
class polynomialNode
{
public:
unsigned int exponent;
float coefficient;
polynomialNode* nextLoc;
polynomialNode()
{
exponent = 0;
coefficient = 0.0f;
nextLoc = NULL;
}
polynomialNode(unsigned int INexponent, float
INcoefficient)
{
exponent = INexponent;
coefficient = INcoefficient;
nextLoc = NULL;
}
polynomialNode(unsigned int INexponent, float
INcoefficient, polynomialNode* nextLocation,
polynomialNode* prevLocation)
{
exponent = INexponent;
coefficient = INcoefficient;
nextLoc = nextLocation;
}
};
```

Polynomial circular linked list class:

```
class polynomialListCirc
{
public:
polynomialNode* TAIL;

polynomialListCirc()
{
TAIL = NULL;
}

//copies data of input polynomial into itself. Does not affect input polynomial
polynomialListCirc(polynomialListCirc* INpolynomial)
{
TAIL = NULL;
if (INpolynomial->TAIL)
{
polynomialNode* TEMP = INpolynomial->TAIL->nextLoc;
do
{
insertElement(TEMP->exponent, TEMP->coefficient);
TEMP = TEMP->nextLoc;
} while (TEMP != INpolynomial->TAIL->nextLoc);
}
}
```

Polynomial circular linked list class:

```
//accepts input from user as a polynomial
void pRead()
{
unsigned int numberOfElements;
std::cout << "Enter number of elements in list: ";
std::cin >> numberOfElements;
std::string polynomial;
float coefficient;
unsigned int exponent, minExp = -1;
std::string temp;
for (int i = 0; i < numberOfElements; i++)
{
std::cout << i + 1 << ": \nEnter coefficient: ";
std::cin >> coefficient;
std::cout << "Enter exponent: ";
std::cin >> exponent;

if (coefficient < 0)
temp = "-";
else
temp = "+";

if (minExp == -1)
{
polynomial.append(std::to_string(coefficient)).append("x^").append(std::to_string(exponent));
}
else
{
if (exponent >= minExp)
{
std::cout << "Invalid: Enter polynomial in descending order of exponents || No repition of exponents" << std::endl;
goto end;
}
else
{
polynomial.append(temp).append(std::to_string(abs(coefficient))).append("x^").append(std::to_string(exponent));
}
}
minExp = exponent;
}
//std::cout << polynomial << std::endl;
pRead(polynomial);

end:;
}
```

Polynomial circular linked list class:

```
//accepts a string for conversion into a polynomial
list. Example format : 321x^5+22x^2-3x^0
void pRead(std::string polynomial)
{
size_t first_index = 0;
unsigned int exp;
float coeff;

while (!polynomial.empty())
{
first_index =
polynomial.find_first_not_of("0123456789+-.");
coeff = atof(polynomial.substr(0,
first_index).c_str());
polynomial.erase(0, first_index);
polynomial.erase(0, 2);

first_index =
polynomial.find_first_not_of("0123456789");
exp = atoi(polynomial.substr(0, first_index).c_str());
polynomial.erase(0, first_index);
if(!TAIL)
insertElement(exp, coeff);
else
{
if (exp >= TAIL->exponent)
{
std::cout << "Invalid: Enter polynomial in descending
order of exponents || No repition of exponents" <<
std::endl;
}
else
{
insertElement(exp, coeff);
}
}
}
}
```

Polynomial circular linked list class:

```
//prints contents of polynomial. Example : +321x^2+5x-3
void pWrite()
{
if (TAIL)
{
polynomialNode* CURRENT = TAIL->nextLoc;
do
{
if (CURRENT->coefficient > 0)
std::cout << '+';
else if (CURRENT->coefficient < 0)
std::cout << "";
else
{
CURRENT = CURRENT->nextLoc;
continue;
}

std::cout << CURRENT->coefficient;

if (CURRENT->exponent == 0)
{

}
else if (CURRENT->exponent == 1)
{
std::cout << 'x';
}
else
{
std::cout << "x^" << CURRENT->exponent;
}

CURRENT = CURRENT->nextLoc;
} while (CURRENT != TAIL->nextLoc);
std::cout << std::endl;
}
else
{
std::cout << "Empty polynomial while writing" << std::endl;
}

}
```

Polynomial circular linked list class:

```
//accepts two polynomialListCirc object pointers and stores their sum as a polynomialListCirc. Erases current data in object, does not affect input polynomials
void pAdd(polynomialListCirc* polynomialA, polynomialListCirc* polynomialB)
{
deleteList();

polynomialNode* CURA;
polynomialNode* CURB;
bool firstTailA = true;
bool firstTailB = true;

if (polynomialA->TAIL == NULL)
{
polynomialListCirc TEMP = new polynomialListCirc(polynomialB);
TAIL = TEMP.TAIL;
goto end;
}
else if (polynomialB->TAIL == NULL)
{
polynomialListCirc TEMP = new polynomialListCirc(polynomialA);
TAIL = TEMP.TAIL;
goto end;
}

CURA = polynomialA->TAIL->nextLoc;
CURB = polynomialB->TAIL->nextLoc;

while (((CURA != polynomialA->TAIL->nextLoc) && (CURB != polynomialB->TAIL->nextLoc)) || (firstTailA || firstTailB))
{
if (CURA->exponent == CURB->exponent)
{
insertElement(CURA->exponent, CURA->coefficient + CURB->coefficient);
CURA = CURA->nextLoc;
CURB = CURB->nextLoc;
firstTailA = firstTailB = false;
}
else if (CURA->exponent > CURB->exponent)
{
insertElement(CURA->exponent, CURA->coefficient);
CURA = CURA->nextLoc;
firstTailA = false;
}
else if (CURA->exponent < CURB->exponent)
{
insertElement(CURB->exponent, CURB->coefficient);
CURB = CURB->nextLoc;
firstTailB = false;
}
}
if ((CURA != polynomialA->TAIL->nextLoc) || firstTailA)
{
while ((CURA != polynomialA->TAIL->nextLoc) || firstTailA)
{
insertElement(CURA->exponent, CURA->coefficient);
CURA = CURA->nextLoc;
firstTailA = false;
}
}
if ((CURB != polynomialB->TAIL->nextLoc) || firstTailB)
{
while ((CURB != polynomialB->TAIL->nextLoc) || firstTailB)
{
insertElement(CURB->exponent, CURB->coefficient);
CURB = CURB->nextLoc;
firstTailB = false;
}
}

end:;
}
```

Polynomial circular linked list class:

```
//accepts two polynomialListCirc object pointers and stores their sum as a polynomialListCirc which is
returned as a pointer. Does not affect input polynomials
static polynomialListCirc* p_Add(polynomialListCirc* polynomialA, polynomialListCirc* polynomialB)
{
polynomialListCirc final;

polynomialNode* CURA;
polynomialNode* CURB;
bool firstTailA = true;
bool firstTailB = true;

if (polynomialA->TAIL == NULL)
{
final = new polynomialListCirc(polynomialB);
goto end;
}
else if (polynomialB->TAIL == NULL)
{
final = new polynomialListCirc(polynomialA);
goto end;
}

CURA = polynomialA->TAIL->nextLoc;
CURB = polynomialB->TAIL->nextLoc;

while (((CURA != polynomialA->TAIL->nextLoc) && (CURB != polynomialB->TAIL->nextLoc)) || (firstTailA ||
firstTailB))
{
if (CURA->exponent == CURB->exponent)
{
final.insertElement(CURA->exponent, CURA->coefficient + CURB->coefficient);
CURA = CURA->nextLoc;
CURB = CURB->nextLoc;
firstTailA = firstTailB = false;
}
else if (CURA->exponent > CURB->exponent)
{
final.insertElement(CURA->exponent, CURA->coefficient);
CURA = CURA->nextLoc;
firstTailA = false;
}
else if(CURA->exponent < CURB->exponent)
{
final.insertElement(CURB->exponent, CURB->coefficient);
CURB = CURB->nextLoc;
firstTailB = false;
}
}
if ((CURA != polynomialA->TAIL->nextLoc) || firstTailA)
{
while ((CURA != polynomialA->TAIL->nextLoc) || firstTailA)
{
final.insertElement(CURA->exponent, CURA->coefficient);
CURA = CURA->nextLoc;
firstTailA = false;
}
}
if ((CURB != polynomialB->TAIL->nextLoc) || firstTailB)
{
while ((CURB != polynomialB->TAIL->nextLoc) || firstTailB)
{
final.insertElement(CURB->exponent, CURB->coefficient);
CURB = CURB->nextLoc;
firstTailB = false;
}
}

end:
return &final;
}
```

Polynomial circular linked list class:

```
//accepts two polynomialListCirc object pointers and stores their difference as a polynomialListCirc.
Erases current data in object, does not affect input polynomials
void pSub(polynomialListCirc* polynomialA, polynomialListCirc* polynomialB)
{
deleteList();

polynomialNode* CURA;
polynomialNode* CURB;
bool firstTailA = true;
bool firstTailB = true;

if (polynomialA->TAIL == NULL)
{
std::cout << "Polynomial A is empty" << std::endl;
TAIL = NULL;
goto end;
}
else if (polynomialB->TAIL == NULL)
{
polynomialListCirc TEMP = new polynomialListCirc(polynomialA);
TAIL = TEMP.TAIL;
goto end;
}

CURA = polynomialA->TAIL->nextLoc;
CURB = polynomialB->TAIL->nextLoc;

while (((CURA != polynomialA->TAIL->nextLoc) && (CURB != polynomialB->TAIL->nextLoc)) || (firstTailA ||
firstTailB))
{
if (CURA->exponent == CURB->exponent)
{
insertElement(CURA->exponent, CURA->coefficient - CURB->coefficient);
CURA = CURA->nextLoc;
CURB = CURB->nextLoc;
firstTailA = firstTailB = false;
}
else if (CURA->exponent > CURB->exponent)
{
insertElement(CURA->exponent, CURA->coefficient);
CURA = CURA->nextLoc;
firstTailA = false;
}
else if (CURA->exponent < CURB->exponent)
{
insertElement(CURB->exponent, -CURB->coefficient);
CURB = CURB->nextLoc;
firstTailB = false;
}
}
if ((CURA != polynomialA->TAIL->nextLoc) || firstTailA)
{
while ((CURA != polynomialA->TAIL->nextLoc) || firstTailA)
{
insertElement(CURA->exponent, CURA->coefficient);
CURA = CURA->nextLoc;
firstTailA = false;
}
}
if ((CURB != polynomialB->TAIL->nextLoc) || firstTailB)
{
while ((CURB != polynomialB->TAIL->nextLoc) || firstTailB)
{
insertElement(CURB->exponent, -CURB->coefficient);
CURB = CURB->nextLoc;
firstTailB = false;
}
}
end:;
}
```

Polynomial circular linked list class:

```
//accepts two polynomialListCirc object pointers and stores their difference as a polynomialListCirc which is returned as a pointer. Does not affect input polynomials
static polynomialListCirc* p_Sub(polynomialListCirc* polynomialA, polynomialListCirc* polynomialB)
{
polynomialListCirc final;
polynomialNode* CURA = polynomialA->TAIL->nextLoc;
polynomialNode* CURB = polynomialB->TAIL->nextLoc;

bool firstTailA = true;
bool firstTailB = true;


if (polynomialA->TAIL == NULL)
{
std::cout << "Polynomial A is empty" << std::endl;
final = NULL;
goto end;
}
else if (polynomialB->TAIL == NULL)
{
final= new polynomialListCirc(polynomialA);
goto end;
}

while (((CURA != polynomialA->TAIL->nextLoc) && (CURB != polynomialB->TAIL->nextLoc)) || (firstTailA || firstTailB))
{
if (CURA->exponent == CURB->exponent)
{
final.insertElement(CURA->exponent, CURA->coefficient - CURB->coefficient);
CURA = CURA->nextLoc;
CURB = CURB->nextLoc;
firstTailA = firstTailB = false;
}
else if (CURA->exponent > CURB->exponent)
{
final.insertElement(CURA->exponent, CURA->coefficient);
CURA = CURA->nextLoc;
firstTailA = false;
}
else if (CURA->exponent < CURB->exponent)
{
final.insertElement(CURB->exponent, CURB->coefficient);
CURB = CURB->nextLoc;
firstTailB = false;
}
}
if ((CURA != polynomialA->TAIL->nextLoc) || firstTailA)
{
while ((CURA != polynomialA->TAIL->nextLoc) || firstTailA)
{
final.insertElement(CURA->exponent, CURA->coefficient);
CURA = CURA->nextLoc;
firstTailA = false;
}
}
if ((CURB != polynomialB->TAIL->nextLoc) || firstTailB)
{
while ((CURB != polynomialB->TAIL->nextLoc) || firstTailB)
{
final.insertElement(CURB->exponent, CURB->coefficient);
CURB = CURB->nextLoc;
firstTailB = false;
}
}

end:
return &final;
}
```

Polynomial circular linked list class:

```
//accepts two polynomialListCirc object pointers and stores their product as a
polynomialListCirc. Erases current data in object, does not affect input polynomials
void pMult(polynomialListCirc* polynomialA, polynomialListCirc* polynomialB)
{
deleteList();

polynomialListCirc resultPolynomial;
polynomialNode* CURRENT;

if ((polynomialA->TAIL == NULL) || (polynomialB->TAIL == NULL))
{
std::cout << "Empty polynomial while multiplying" << std::endl;
goto end;
}
CURRENT = polynomialA->TAIL->nextLoc;

do
{
polynomialListCirc tempPolynomial = *new polynomialListCirc(polynomialB);
tempPolynomial.pMultSingle(CURRENT->exponent, CURRENT->coefficient);

resultPolynomial = polynomialListCirc::p_Add(&resultPolynomial, &tempPolynomial);
CURRENT = CURRENT->nextLoc;
} while (CURRENT != polynomialA->TAIL->nextLoc);

TAIL = resultPolynomial.TAIL;
end:;
}

//accepts two polynomialListCirc object pointers and stores their product as a
polynomialListCirc which is returned as a pointer. Does not affect input polynomials
static polynomialListCirc* p_Mult(polynomialListCirc* polynomialA,
polynomialListCirc* polynomialB)
{
polynomialListCirc resultPolynomial;
polynomialNode* CURRENT;

if ((polynomialA->TAIL == NULL) || (polynomialB->TAIL == NULL))
{
std::cout << "Empty polynomial while multiplying" << std::endl;
goto end;
}
CURRENT = polynomialA->TAIL->nextLoc;

do
{
polynomialListCirc tempPolynomial = *new polynomialListCirc(polynomialB);
tempPolynomial.pMultSingle(CURRENT->exponent, CURRENT->coefficient);

resultPolynomial = polynomialListCirc::p_Add(&resultPolynomial, &tempPolynomial);
CURRENT = CURRENT->nextLoc;
} while (CURRENT != polynomialA->TAIL->nextLoc);

end:;
return &resultPolynomial;
}
```

Polynomial circular linked list class:

```
//multiplies entire polynomial with a polynomial of type
coefficient*x^exponent (helper function)
void pMultSingle(unsigned int exponent, float coefficient)
{
if (TAIL != NULL)
{
polynomialNode* CURRENT = TAIL->nextLoc;
do
{
CURRENT->coefficient *= coefficient;
CURRENT->exponent += exponent;
CURRENT = CURRENT->nextLoc;
} while (CURRENT != TAIL->nextLoc);
}
else
{
std::cout << "Empty polynomial while multiplying singly" << std::endl;
}
}


//evaluates the polynomial at a point A which is taken as input and
returns the evaluated value
float pEvaluate(const float pointA)
{
if (TAIL)
{
float result = 0.0f;
polynomialNode* CURRENT = TAIL->nextLoc;
do
{
result += (CURRENT->coefficient * pow(pointA, CURRENT->exponent));
CURRENT = CURRENT->nextLoc;
} while (CURRENT != TAIL->nextLoc);
}
else
{
return 0.0f;
}
}
```

Polynomial circular linked list class:

```
//erases a term of an input exponent value from the polynomial
void pEraseTerm(unsigned int exponent)
{

if (TAIL)
{
if (TAIL->nextLoc == TAIL)
{
if (TAIL->exponent == exponent)
{
polynomialNode* TEMP = TAIL;
TAIL = NULL;
delete TEMP;
}
else
{
std::cout << "Term not present in polynomial" << std::endl;
}
}
else if (TAIL->exponent == exponent)
{
polynomialNode* CURRENT = TAIL->nextLoc;
while (CURRENT->nextLoc != TAIL)
{
CURRENT = CURRENT->nextLoc;
}
polynomialNode* TEMP = CURRENT->nextLoc;
CURRENT->nextLoc = TEMP->nextLoc;
delete TEMP;
TAIL = CURRENT;
}
else
{
polynomialNode* CURRENT = TAIL;
bool found = false;
do
{
if (CURRENT->nextLoc->exponent == exponent)
{
polynomialNode* TEMP = CURRENT->nextLoc;
CURRENT->nextLoc = CURRENT->nextLoc->nextLoc;
delete TEMP;
found = true;
}
CURRENT = CURRENT->nextLoc;
}while (CURRENT != TAIL);

if (!found)
{
std::cout << "Term not present in polynomial" << std::endl;
}
}
}
else
{
std::cout << "Empty polynomial while erasing term" << std::endl;
}

}
```

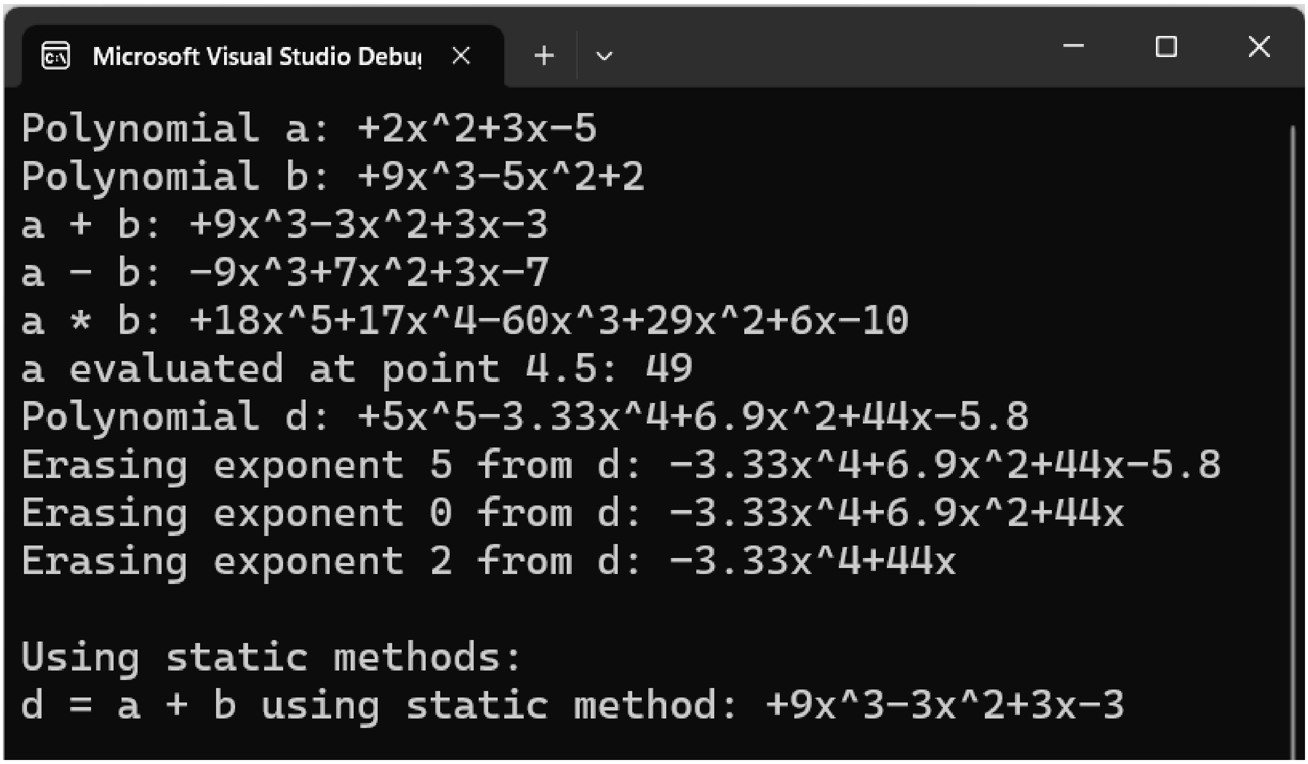
Polynomial circular linked list class:

```
private:
void insertElement(unsigned int exponent, float coefficient)
{
if (TAIL)
{
polynomialNode* TEMP = new polynomialNode(exponent,
coefficient);
TEMP->nextLoc = TAIL->nextLoc;
TAIL->nextLoc = TEMP;
TAIL = TEMP;
}
else
{
TAIL = new polynomialNode(exponent, coefficient);
TAIL->nextLoc = TAIL;
}
}

void deleteList()
{
if (TAIL)
{
polynomialNode* TEMP;
while (TAIL->nextLoc != TAIL)
{
TEMP = TAIL->nextLoc->nextLoc;
delete TAIL->nextLoc;
TAIL->nextLoc = TEMP;
}
delete TAIL;
TAIL = NULL;
}
}
};
```

Main function for demonstration:

```cpp
void main()
{
polynomialListCirc a, b, c;
a.pRead("2x^2+3x^1-5x^0");
std::cout << "Polynomial a: ";
a.pWrite();
b.pRead("9x^3-5x^2+2x^0");
std::cout << "Polynomial b: ";
b.pWrite();

std::cout << "a + b: ";
c.pAdd(&a, &b);
c.pWrite();

std::cout << "a - b: ";
c.pSub(&a, &b);
c.pWrite();

std::cout << "a * b: ";
c.pMult(&a, &b);
c.pWrite();

std::cout << "a evaluated at point 4.5: ";
std::cout << a.pEvaluate(4.5f) << std::endl;

polynomialListCirc d;
d.pRead("5x^5-3.33x^4+6.9x^2+44x^1-5.8x^0");
std::cout << "Polynomial d: ";
d.pWrite();
std::cout << "Erasing exponent 5 from d: ";
d.pEraseTerm(5);
d.pWrite();
std::cout << "Erasing exponent 0 from d: ";
d.pEraseTerm(0);
d.pWrite();
std::cout << "Erasing exponent 2 from d: ";
d.pEraseTerm(2);
d.pWrite();

//using static methods
std::cout << "\nUsing static methods: " << std::endl;
d = polynomialListCirc::p_Add(&a, &b);
std::cout << "d = a + b using static method: ";
d.pWrite();
}
```

Output of program:



```
Polynomial a: +2x^2+3x-5
Polynomial b: +9x^3-5x^2+2
a + b: +9x^3-3x^2+3x-3
a - b: -9x^3+7x^2+3x-7
a * b: +18x^5+17x^4-60x^3+29x^2+6x-10
a evaluated at point 4.5: 49
Polynomial d: +5x^5-3.33x^4+6.9x^2+44x-5.8
Erasing exponent 5 from d: -3.33x^4+6.9x^2+44x-5.8
Erasing exponent 0 from d: -3.33x^4+6.9x^2+44x
Erasing exponent 2 from d: -3.33x^4+44x

Using static methods:
d = a + b using static method: +9x^3-3x^2+3x-3
```