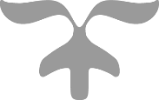
19AIE111 : DATA STRUCTURES AND ALGORITHM





**POLYNOMIAL ADT**

USING HEAP AND HASH CODE



DONE BY   
CB.EN.U4AIE19003 – **ADHITHAN**

CB.EN.U4AIE19023 – **MITHIL**

CB.EN.U4AIE19024 – **DHARSHAN KUMAR**

CB.EN.U4AIE19033 – **KABILAN**

CB.EN.U4AIE19061 – **SIVAMARAN**

BTECH CSE-AI

CEN DEPARTMENT

AMRITA SCHOOL OF ENGINEERING

**Abstract:**

We will be solving the given problem statement. We have done the implementation by storing the power of the polynomial in heap and using a hash map to store the power and the coefficient as key-value pair. we have also implemented the heap and hash map.

By using our java program, the user can add, subtract, multiply, integrate, differentiate, find root, get degree and also compute the polynomial expressions.

We had made this as an abstraction, that is, we had hide the background details and showed only the essential features to the users.

Abstract data type used(ADT) : Heap, Hash

Concrete data type used(CDT) : Array, Linked list

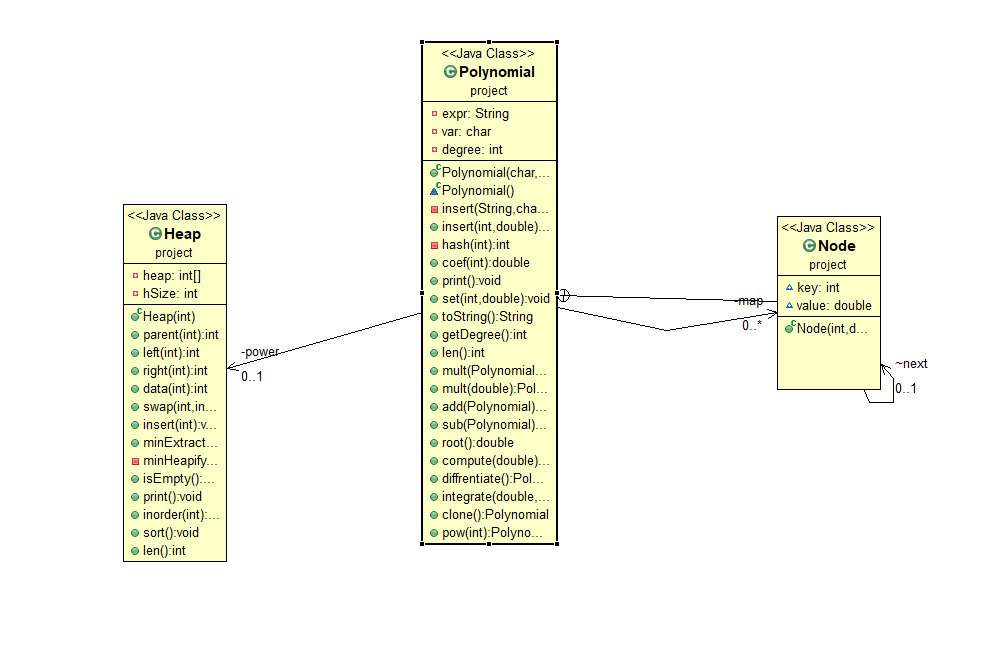
**UML Diagram:**

Table of Contents

[**Introduction**: 5](#_Toc43720130)

[**Problem statement:** 5](#_Toc43720131)

[**ADT min-heap:** 6](#_Toc43720132)

[Implement of min-heap 6](#_Toc43720133)

[Instance variables: 6](#_Toc43720134)

[left(int i): 7](#_Toc43720135)

[right(int i) 7](#_Toc43720136)

[parent(int i) 7](#_Toc43720137)

[data(int i) 7](#_Toc43720138)

[insert(int e): 7](#_Toc43720139)

[Swap(int a,int b): 8](#_Toc43720140)

[minHeapify(int n): 9](#_Toc43720141)

[minExtract(): 9](#_Toc43720142)

[Sort(): 10](#_Toc43720143)

[len(): 10](#_Toc43720144)

[isEmpty(): 10](#_Toc43720145)

[inorder(): 10](#_Toc43720146)

[Clone(): 10](#_Toc43720147)

[print() 10](#_Toc43720148)

[**Complexity using O(n):** 11](#_Toc43720149)

[**Polynomial ADT:** 11](#_Toc43720150)

[Instance variables: 11](#_Toc43720151)

[toString(): 12](#_Toc43720152)

[insert(String s, char v): 13](#_Toc43720153)

[insert(int key, double value) 14](#_Toc43720154)

[int hash(int n): 15](#_Toc43720155)

[double coef(int pow): 16](#_Toc43720156)

[clone(): 16](#_Toc43720157)

[print(): 16](#_Toc43720158)

[set(int pow, double cof): 16](#_Toc43720159)

[getDegree(): 17](#_Toc43720160)

[int len(): 17](#_Toc43720161)

[compute(double n): 17](#_Toc43720162)

[add(Polynomial p1): 18](#_Toc43720163)

[subtract(Polynomial p2): 18](#_Toc43720164)

[mult(Polynomial that): 19](#_Toc43720165)

[mult(double d): 19](#_Toc43720166)

[Polynomial diffrentiate(): 19](#_Toc43720167)

[integrate(double a, double b): 20](#_Toc43720168)

[Polynomial pow(int n): 20](#_Toc43720169)

[root() 21](#_Toc43720170)

[**Big-O notation for Polynomial ADT:** 21](#_Toc43720171)

[**Java code:** 22](#_Toc43720172)

[**Work spilt up:** 30](#_Toc43720172)

# Introduction:

In mathematics, a polynomial is an expression consisting of variables (also called indeterminates) and coefficients, that involves only the operations of addition, subtraction, multiplication, and non-negative integer exponents of variables. An example of a polynomial of a single indeterminate, x, is x2 − 4x + 7.

An Abstract Data Type (ADT) is a set of operations. Abstract data types are mathematical abstractions; nowhere in an ADT's definition is there any mention of how the set of operations is implemented. This can be viewed as an extension of modular design.

Objects such as lists, sets, and graphs, along with their operations, can be viewed as abstract data types, just as integers, float, and boolean are data types. Integers, reals, and boolean have operations associated with them, and so do abstract data types. A polynomial ADT is required to represent a polynomial. For the Polynomial ADT, there are various operations as addition, subtraction, multiplication, append etc.

The basic idea is that the implementation of these operations is written once in the program, and any other part of the program that needs to perform an operation on the ADT can do so by calling the appropriate function. If for some reason implementation details need to change, it should be easy to do so by merely changing the routines that perform the ADT operations. This change, in a perfect world, would be completely transparent to the rest of the program.

# Problem statement:

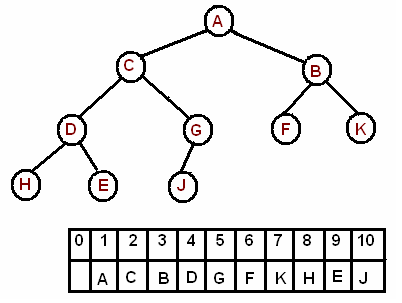
Representing a single variable polynomial of any degree in a class using minheap to store powers. The polynomial class should be able to do operations such as **multiply, add** , **subtract** via methods. Also, one must be able to **square** and **append** the polynomial (modify the coefficients of a certain power term or add more terms. A **‘getDegree ‘**method must be automatically computed. The **‘toString’** method should convert the polynomial into a String with increasing order of degrees.

# ADT min-heap:

A binary heap is a complete binary tree which satisfies the heap ordering property. The ordering of min-heap is such that the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root. A heap is useful data structure when you need to remove the object with the highest (or lowest) priority. A common use of a heap is to implement a priority queue.

Implement of min-heap:

A complete binary tree can be uniquely represented by storing its level order traversal in an array.



## Instance variables:

Private Int []heap- to store the elements of the heap

private int hSize – to update the size of the heap

The root is the first element in the array. Consider i-th element of the array, the left child is located at 2\*i index

its right child is located at 2\*i+1. index

its parent is located at i/2 index

## left(int i):

left child is located at 2\*i index

## right(int i)

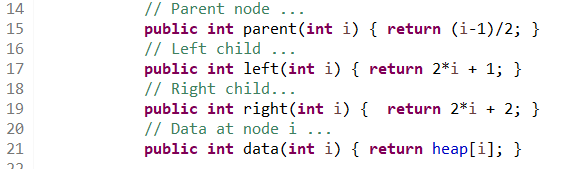
its right child is located at 2\*i+1. Index

## parent(int i)

its parent is located at i/2 index

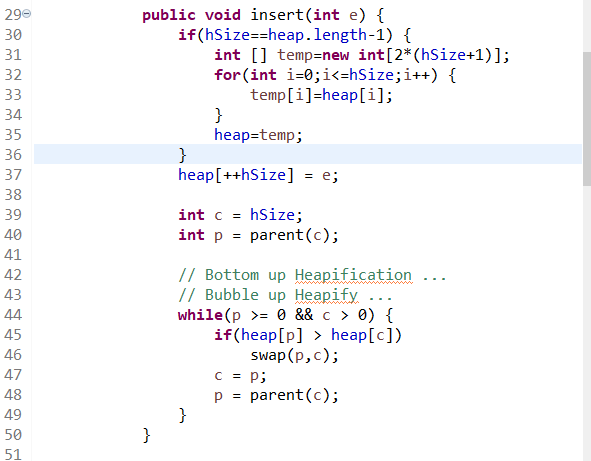
## data(int i)

to get the element at the location of i.



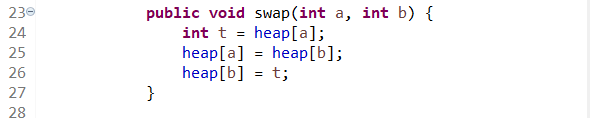
## insert(int e):

The new element is initially appended to the end of the heap (as the last element of the array). The heap property is maintained by comparing the added element with its parent and moving the added element up a level (swapping positions with the parent). This process is called "percolation up”, it is bottom to top heapification. The comparison is repeated until the parent is larger than or equal to the percolating element.

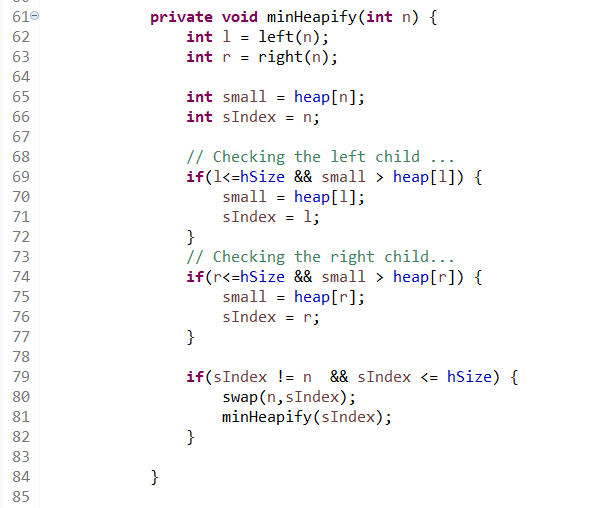


## Swap(int a,int b):

To interchange to elements of the heap at two positions a and b.

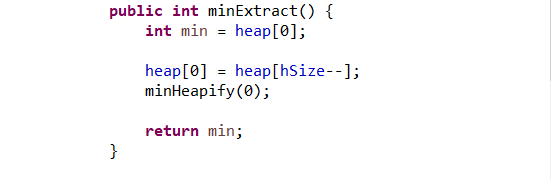


## minHeapify(int n):

To restore the heap property by percolating down (top to bottom heapification). 

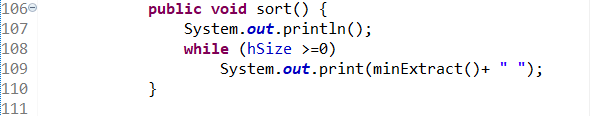
## minExtract():

The minimum element is always the root of the tree, which is the first element of the array. We remove the root and replace it with the last element of the heap and then restore the heap property by percolating down (top to bottom heapification) by calling minHeapify method. Similar to insertion, the worst-case runtime is O{log n).



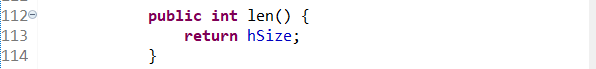
## Sort():

By calling minExtract, the least element is printed at each iteration. The running time of the algorithm is O(n log n).



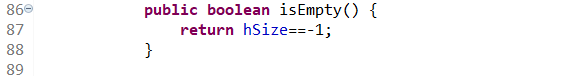
## len():

keeps track of the size of the heap that is filled.



## isEmpty():

To check if the heap is empty.



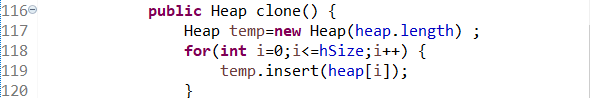
## inorder():

To print the inorder traversal of the heap.



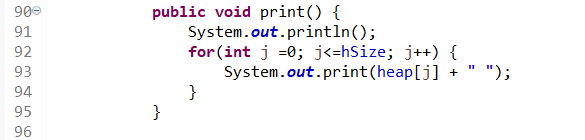
## Clone():

To create a copy of the heap with a different reference.



## print()

printing the elements of the tree by traversing the array.

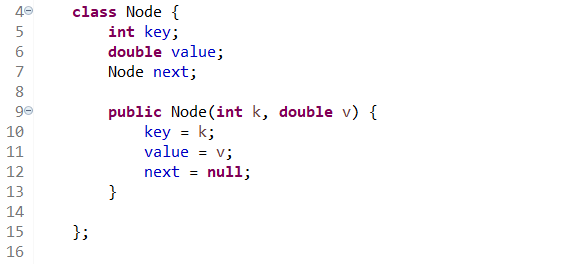


# Complexity using O(n):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | insert | minExtract | findMin | sort |
| Binary heap | O(log n) | O(log n) | O(1) | O(n\*logn) |

# Polynomial ADT:

ADT node: to store the key-value pair with a linked list structure.



## Instance variables:

private String expr: to store the polynomial expression given as string in the constructor.

private char var: to store the variable of the polynomial (eg.in x2+x+1 x is the variable)

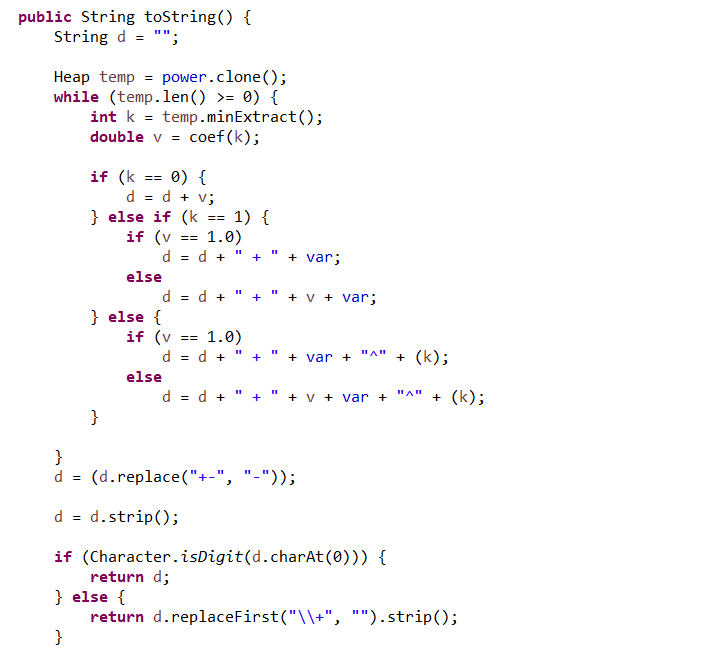
private Node[] map : To store the power and coeffient of a term key-value pair through hash map.

private Heap power: To populate the heap with power as key

private int degree: To store the degree of the polynomial

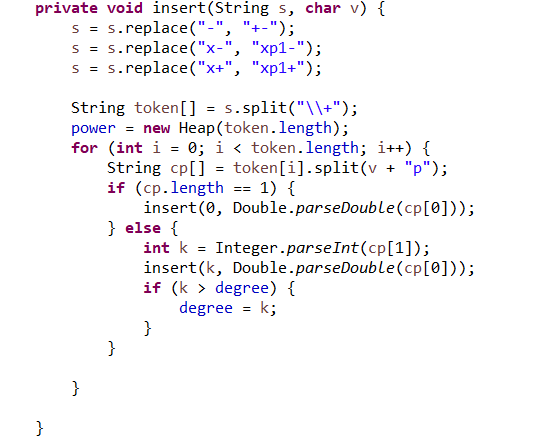
## toString():

to convert the polynomial into a String with increasing order of degrees.



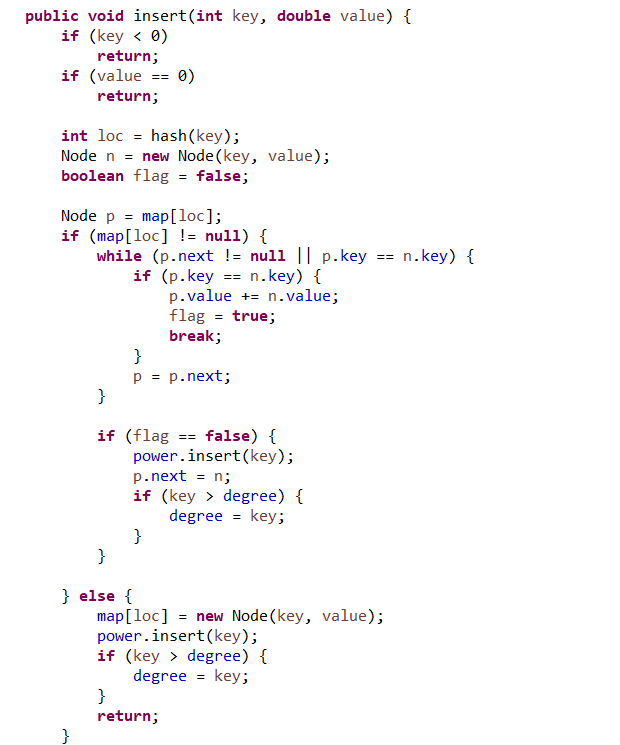
## insert(String s, char v):

To tokenize the string s and store it in the polynomial ADT.



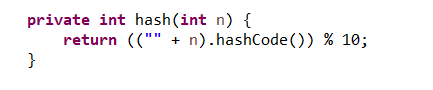
## insert(int key, double value)

To insert additional terms to the Polynomial ADT.



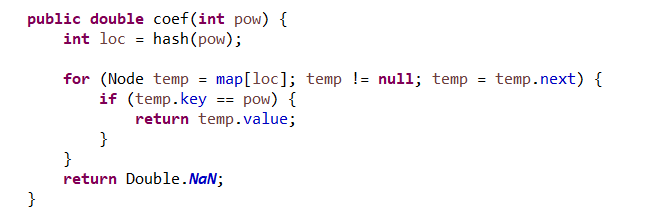
## int hash(int n):

Each polynomial term has power which is the key for the hash table. The key is hashed using the built-in hashCode() such that it is mapped to locations from 0-9 in array map.



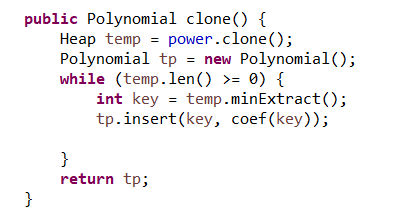
## double coef(int pow):

to return the corresponding coeffient for the given power of the term (key for the hash map).



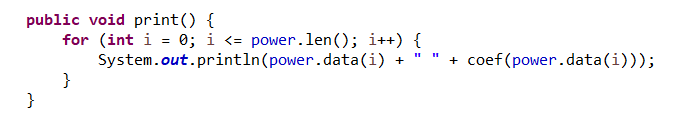
## clone():

to create a duplicate of the polynomial such that it has a different reference. This is to avoid any side effect in the program.



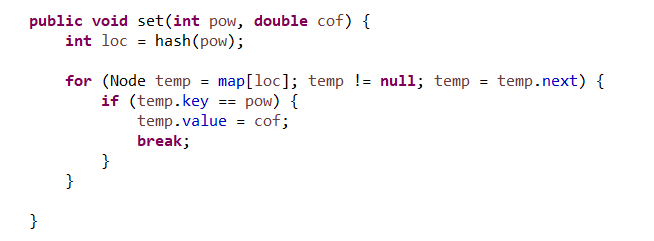
## print():

to print the power and the corresponding coefficient of all terms of the polynomial ADT.



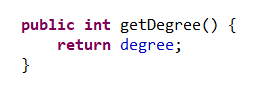
## set(int pow, double cof):

to modify the coefficient of a term ,given the power of term(key).



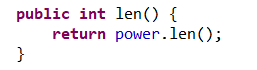
## getDegree():

to return the degree of the polynomial.



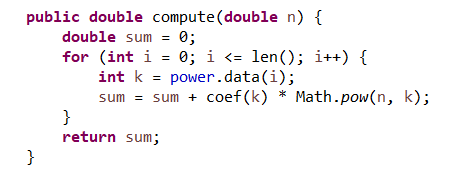
## int len():

to keep tract of the number of terms(size of the heap).



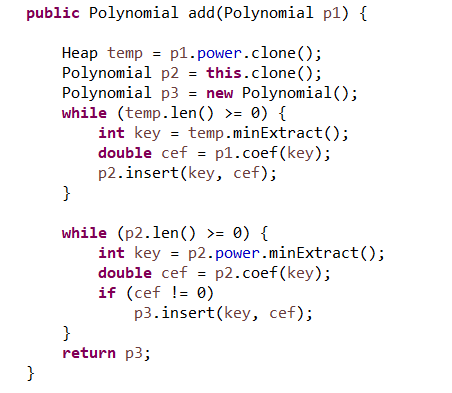
## compute(double n):

To compute the function value of the polynomial at point n.



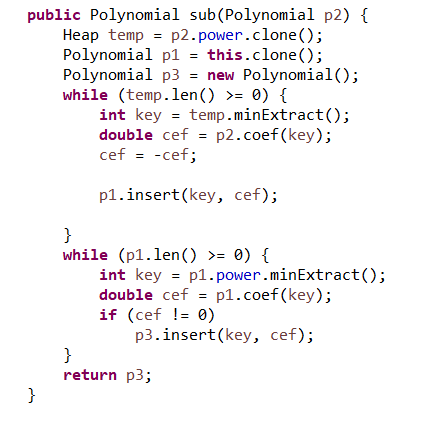
## add(Polynomial p1):

To add two polynomials.



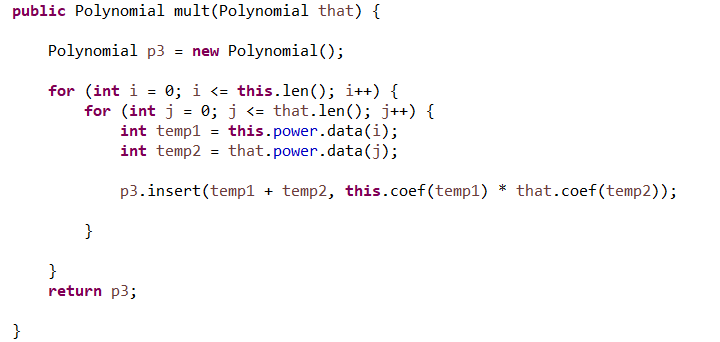
## subtract(Polynomial p2):

To subtract two polynomials.



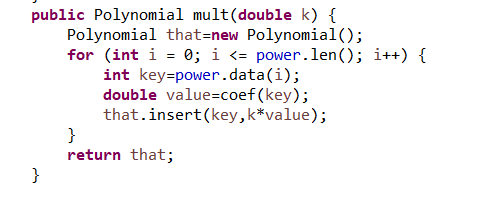
## mult(Polynomial that):

To multiply two polynomials.



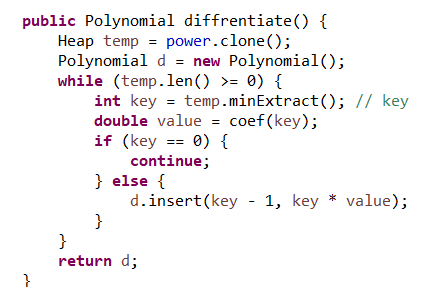
## mult(double d):

To multiply the polynomials with a constant.



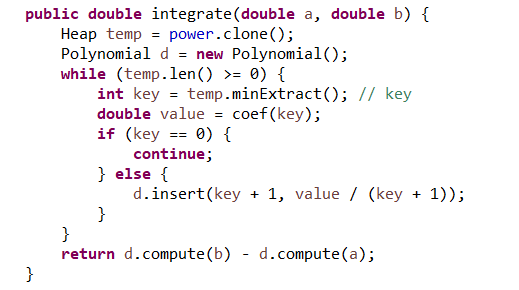
## Polynomial diffrentiate():

To differentiate a the polynomial with respect to the variable.



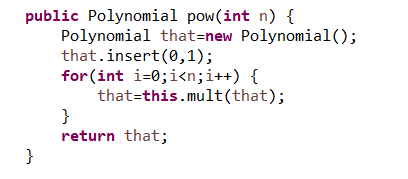
## integrate(double a, double b):

To integrate a the polynomial with respect to the variable from a to b.



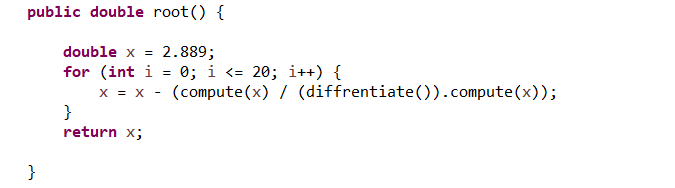
## Polynomial pow(int n):

To return the polynomial raised to the power n(eg x2+x+1 return (x2+x+1)n).



## root()

To compute a root of the polynomial.



# Big-O notation for Polynomial ADT:

|  |  |
| --- | --- |
| Polynomial method | O notation |
| insert() | O(1) |
| hash() | O(1) |
| Coef() | O(1) |
| print() | O(n) |
| set() | O(n) |
| toString() | O(nlog(n)) |
| getDegree() | O(1) |
| len() | O(1) |
| add() | O(nlog(n)) |
| sub() | O(nlog(n)) |
| Mult(Polynomial) | O(n2) |
| Mult(constant) | O(n) |
| compute() | O(n) |
| differentiate() | O(nlog(n)) |
| integrate() | O(nlog(n)) |
| clone() | O(nlog(n)) |
| root() | O(nlog(n)) |
| pow() | O(n3) |

# Java code:

**package** project;

**public** **class** Heap {

// Array the implements the CBT...

**private** **int**[] heap;

// number of values in the array

**private** **int** hSize;

**public** Heap(**int** size) {

heap = **new** **int**[size];

hSize = -1;

}

// Parent node ...

**public** **int** parent(**int** i) { **return** (i-1)/2; }

// Left child ...

**public** **int** left(**int** i) { **return** 2\*i + 1; }

// Right child...

**public** **int** right(**int** i) { **return** 2\*i + 2; }

// Data at node i ...

**public** **int** data(**int** i) { **return** heap[i]; }

**public** **void** swap(**int** a, **int** b) {

**int** t = heap[a];

heap[a] = heap[b];

heap[b] = t;

}

**public** **void** insert(**int** e) {

**if**(hSize==heap.length-1) {

**int** [] temp=**new** **int**[2\*(hSize+1)];

**for**(**int** i=0;i<=hSize;i++) {

temp[i]=heap[i];

}

heap=temp;

}

heap[++hSize] = e;

**int** c = hSize;

**int** p = parent(c);

// Bottom up Heapification ...

// Bubble up Heapify ...

**while**(p >= 0 && c > 0) {

**if**(heap[p] > heap[c])

swap(p,c);

c = p;

p = parent(c);

}

}

**public** **int** minExtract() {

**int** min = heap[0];

heap[0] = heap[hSize--];

minHeapify(0);

**return** min;

}

**private** **void** minHeapify(**int** n) {

**int** l = left(n);

**int** r = right(n);

**int** small = heap[n];

**int** sIndex = n;

// Checking the left child ...

**if**(l<=hSize && small > heap[l]) {

small = heap[l];

sIndex = l;

}

// Checking the right child...

**if**(r<=hSize && small > heap[r]) {

small = heap[r];

sIndex = r;

}

**if**(sIndex != n && sIndex <= hSize) {

swap(n,sIndex);

minHeapify(sIndex);

}

}

**public** **boolean** isEmpty() {

**return** hSize==-1;

}

**public** **void** print() {

System.***out***.println();

**for**(**int** j =0; j<=hSize; j++) {

System.***out***.print(heap[j] + " ");

}

}

**public** **void** inorder(**int** root) {

**if**(root > hSize) **return**;

inorder(left(root));

System.***out***.print(data(root) +" ");

inorder(right(root));

}

**public** **void** sort() {

System.***out***.println();

**while** (hSize >=0)

System.***out***.print(minExtract()+ " ");

}

**public** **int** len() {

**return** hSize;

}

**public** Heap clone() {

Heap temp=**new** Heap(heap.length) ;

**for**(**int** i=0;i<=hSize;i++) {

temp.insert(heap[i]);

}

**return** temp;

}

}

**package** project;

**public** **class** Polynomial {

**class** Node {

**int** key;

**double** value;

Node next;

**public** Node(**int** k, **double** v) {

key = k;

value = v;

next = **null**;

}

};

**private** String expr;//Store the polynomial expression in string

**private** **char** var;// Polynomial variable as char

**private** Node[] map;// to map power with coefficient through hashing

**private** Heap power;//storing the power in a minheap

**private** **int** degree;//degree of the polynomial

**public** Polynomial(**char** v, String e) {

var = v;

expr = e;

map = **new** Node[10];

degree = 0;

insert(expr, var);

}

Polynomial() {

map = **new** Node[10];

power = **new** Heap(10);

degree = 0;

var = 'x';

}

**private** **void** insert(String s, **char** v) {

s = s.replace("-", "+-");

s = s.replace("x-", "xp1-");

s = s.replace("x+", "xp1+");

String token[] = s.split("\\+");//tokenizing the regex with '+' as delimiter

power = **new** Heap(token.length);

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

String cp[] = token[i].split(v + "p");//tokenizing the power and coefficient

**if** (cp.length == 1) {

insert(0, Double.*parseDouble*(cp[0]));

} **else** {

**int** k = Integer.*parseInt*(cp[1]);

insert(k, Double.*parseDouble*(cp[0]));//insert the key-value in the map and populate heap

**if** (k > degree) {

degree = k;

}

}

}

}

**public** **void** insert(**int** key, **double** value) {

**if** (key < 0)//neglet negative power

**return**;

**if** (value == 0)//neglet zeroed terms

**return**;

**int** loc = hash(key);//location of the map

Node n = **new** Node(key, value);

**boolean** flag = **false**;

Node p = map[loc];

**if** (map[loc] != **null**) {

**while** (p.next != **null** || p.key == n.key) {

**if** (p.key == n.key) {

p.value += n.value;//adding the coef when they power is same

flag = **true**;

**break**;

}

p = p.next;

}

**if** (flag == **false**) {

power.insert(key);

p.next = n;

**if** (key > degree) {

degree = key;

}

}

} **else** {

map[loc] = **new** Node(key, value);//create new node

power.insert(key);

**if** (key > degree) {

degree = key;

}

**return**;

}

}

**private** **int** hash(**int** n) {

**return** (("" + n).hashCode()) % 10;

}

**public** **double** coef(**int** pow) {

**int** loc = hash(pow);

// traverse through the bucket

**for** (Node temp = map[loc]; temp != **null**; temp = temp.next) {

**if** (temp.key == pow) {

**return** temp.value;

}

}

**return** Double.***NaN***;

}

**public** **void** print() {

**for** (**int** i = 0; i <= power.len(); i++) {

System.***out***.println(power.data(i) + " " + coef(power.data(i)));

}

}

**public** **void** set(**int** pow, **double** cof) {

**int** loc = hash(pow);

**for** (Node temp = map[loc]; temp != **null**; temp = temp.next) {

**if** (temp.key == pow) {

temp.value = cof;

**break**;

}

}

}

**public** String toString() {

String d = "";

Heap temp = power.clone();

**while** (temp.len() >= 0) {

**int** k = temp.minExtract();

**double** v = coef(k);

**if** (k == 0) {

d = d + v;

} **else** **if** (k == 1) {

**if** (v == 1.0)

d = d + " + " + var;

**else**

d = d + " + " + v + var;

} **else** {

**if** (v == 1.0)

d = d + " + " + var + "^" + (k);

**else**

d = d + " + " + v + var + "^" + (k);

}

}

d = (d.replace("+-", "-"));

d = d.strip();

**if** (Character.*isDigit*(d.charAt(0))) {

**return** d;

} **else** {

**return** d.replaceFirst("\\+", "").strip();

}

}

**public** **int** getDegree() {

**return** degree;

}

**public** **int** len() {

**return** power.len();

}

**public** Polynomial mult(Polynomial that) {

Polynomial p3 = **new** Polynomial();

**for** (**int** i = 0; i <= **this**.len(); i++) {

**for** (**int** j = 0; j <= that.len(); j++) {

**int** temp1 = **this**.power.data(i);

**int** temp2 = that.power.data(j);

p3.insert(temp1 + temp2, **this**.coef(temp1) \* that.coef(temp2));

}

}

**return** p3;

}

**public** Polynomial mult(**double** k) {

Polynomial that=**new** Polynomial();

**for** (**int** i = 0; i <= power.len(); i++) {

**int** key=power.data(i);

**double** value=coef(key);

that.insert(key,k\*value);

}

**return** that;

}

**public** Polynomial add(Polynomial p1) {

Heap temp = p1.power.clone();

Polynomial p2 = **this**.clone();

Polynomial p3 = **new** Polynomial();

**while** (temp.len() >= 0) {

**int** key = temp.minExtract();

**double** cef = p1.coef(key);

p2.insert(key, cef);

}

**while** (p2.len() >= 0) {

**int** key = p2.power.minExtract();

**double** cef = p2.coef(key);

**if** (cef != 0)

p3.insert(key, cef);

}

**return** p3;

}

**public** Polynomial sub(Polynomial p2) {

Heap temp = p2.power.clone();

Polynomial p1 = **this**.clone();

Polynomial p3 = **new** Polynomial();

**while** (temp.len() >= 0) {

**int** key = temp.minExtract();

**double** cef = p2.coef(key);

cef = -cef;

p1.insert(key, cef);

}

**while** (p1.len() >= 0) {

**int** key = p1.power.minExtract();

**double** cef = p1.coef(key);

**if** (cef != 0)

p3.insert(key, cef);

}

**return** p3;

}

**public** **double** root() {

**double** x = 2.889;

**for** (**int** i = 0; i <= 20; i++) {

x = x - (compute(x) / (diffrentiate()).compute(x));

}

**return** x;

}

**public** **double** compute(**double** n) {

**double** sum = 0;

**for** (**int** i = 0; i <= len(); i++) {

**int** k = power.data(i);

sum = sum + coef(k) \* Math.*pow*(n, k);

}

**return** sum;

}

**public** Polynomial diffrentiate() {

Heap temp = power.clone();

Polynomial d = **new** Polynomial();

**while** (temp.len() >= 0) {

**int** key = temp.minExtract(); // key

**double** value = coef(key);

**if** (key == 0) {

**continue**;

} **else** {

d.insert(key - 1, key \* value);

}

}

**return** d;

}

**public** **double** integrate(**double** a, **double** b) {

Heap temp = power.clone();

Polynomial d = **new** Polynomial();

**while** (temp.len() >= 0) {

**int** key = temp.minExtract(); // key

**double** value = coef(key);

**if** (key == 0) {

**continue**;

} **else** {

d.insert(key + 1, value / (key + 1));

}

}

**return** d.compute(b) - d.compute(a);

}

**public** Polynomial clone() {

Heap temp = power.clone();

Polynomial tp = **new** Polynomial();

**while** (temp.len() >= 0) {

**int** key = temp.minExtract();

tp.insert(key, coef(key));

}

**return** tp;

}

**public** Polynomial pow(**int** n) {

Polynomial that=**new** Polynomial();

that.insert(0,1);

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

that=**this**.mult(that);

}

**return** that;

}

**public** **static** **void** main(String args[]) {

Polynomial p = **new** Polynomial('x',"1x+1");

Polynomial p1 = p.pow(3);

System.***out***.println(p1.root());

}

}

# Conclusion:

By using our java program, the user can add, subtract, multiply, integrate, differentiate, find root, get degree and also compute the polynomial expressions.

We had made this as an abstraction, that is, we had hide the background details and showed only the essential features to the users.

Abstract data type used(ADT) : Heap, Hash

Concrete data type used(CDT) : Array, Linked list

# Work spilt up:

**Adhithan :**

Multiplication method, power method, compute function and root function

**Mithil :**

Min Heap part of java code, getDegree function and print function

**Dharshan Kumar :**

Clone method, set function, differentiate method and integrate method

**Kabilan :**

Hash function, Coefficient function, add method and subtract method

**Sivamaran :**

Insert method, ToString method and Implementation Idea of project