ՀԱՅԱՍՏԱՆԻ ՀԱՆՐԱՊԵՏՈՒԹՅԱՆ ԿՐԹՈՒԹՅԱՆ ԵՎ ԳԻՏՈՒԹՅԱՆ ՆԱԽԱՐԱՐՈՒԹՅՈՒՆ

ՀԱՅԱՍՏԱՆԻ ԱԶԳԱՅԻՆ ՊՈԼԻՏԵԽՆԻԿԱԿԱՆ ՀԱՄԱԼՍԱՐԱՆ (ՀԻՄՆԱԴՐԱՄ)

ՄԱԳԻՍՏՐՈՍԱԿԱՆ ԿՐԹԱԿԱՆ ԾՐԱԳԻՐ

**ԹԵՄԱ՝ Ինքնաուսուցման համակարգերի կիրառմամբ ինտեգրալ սխեմաների ջերմային տեղաբաշխման ծրագրային միջոցի մշակումը և հետազոտումը**

Թովմասյան Գրիգոր Կառլենի

«Էլեկտրոնային նախագծման ավտոմատացում» մասնագիտությամբ ճարտարագիտության մագիստրոսի որակավորման աստիճան հայցելու ատենախոսություն

ԵՐԵՎԱՆ 2019

ՀԱՍՏԱՏՄԱՆ ԹԵՐԹ

**ԹԵՄԱ՝ Ինքնաուսուցման համակարգերի կիրառմամբ ինտեգրալ սխեմաների ջերմային տեղաբաշխման ծրագրային միջոցի մշակումը և հետազոտումը**

Թովմասյան Գրիգոր Կառլենի

|  |  |
| --- | --- |
| Ատենախոսության ղեկավար՝ | Տ. Ա. Գասպարյան  ֆ.-մ. գ. թ. |
| Մագիստրանտ՝ | Գ. Կ. Թովմասյան  Բակալավր |
| Գրախոս՝ | Վ. Շ. Մելիքյան  տ. գ. դ., պրոֆեսոր |
| Ամբիոնի վարիչ՝ | Վ. Շ. Մելիքյան  տ. գ. դ., պրոֆեսոր |

ԿԵՆՍԱԳՐԱԿԱՆ ՏՎՅԱԼՆԵՐ

|  |  |
| --- | --- |
| Մագիստրանտ՝ | Թովմասյան Գրիգոր Կառլենի |
| Աստիճանը՝ | «Էլեկտրոնային նախագծման ավտոմատացում» մասնագիտության ճարտարագիտության մագիստրանտ |
| Տարեթիվը՝ | 2017 |
| Ծննդյան տարեթիվը՝ | 1995 |
| Մինչ մագիստրոսական որակավորումը՝ | Ճարտարագիտության բակալավրի աստիճան |
| Մասնագիտությունը՝ | Ինֆորմատիկա և հաշվողական տեխնիկա |
| Հրատարակված աշխատանքները՝ | Չկան |

**ՀԱՄԱՌՈՏԱԳԻՐ**

**ԹԵՄԱ՝ Ինքնաուսուցման համակարգերի կիրառմամբ ինտեգրալ սխեմաների ջերմային տեղաբաշխման ծրագրային միջոցի մշակումը և հետազոտումը**

Թովմասյան Գրիգոր Կառլենի

Սույն մագիստրոսական ատենախոսության շրջանակներում ուսումնասիրվել են ջերմային տեղաբաշխում իրականացնող մինչև այժմ հայտնի տեխնոլոգիանները: Դրանց համատեղման արդյունքում մշակվել է նոր ծրագրավորման գրադարան, որը հնարավորություն է տալիս, հաշվել ածանցյալի արժեքը փոփոխականների ցանկացած առժեքների դեպքում, ածանցելով մուտքային ֆունկցիան մայն մեկ անգամ, ի տարբերություն այժմ գոյություն ունեցող այլ գրադարաննեի: Մշակված գրադարանի առավելությունը կայանում է նրանում, որ օգտատերը պարտավոր չէ կոդավորել մուտքային ֆունկցին ծրագրավորման համար հարմար եղանակով, այսինք պարտավոր չէ ստեղծել նոր ֆունկցիաներ գրված C/C++ լեզուններով, այլ պետք է պարզապես տողային տեսքով ստանա ցանկալի ֆունկցիայի տեսքը և փոխանցի գրադարանին: Ատենախոսության ընթացքում հետազոտվել է մշակված գրադարանի օգտագործած ալգորիթմների ծախսած ժամանակները կախված մուտքային ալգորիթմի բարդություննից:

Բովանդակությունը

[Նկարների Ցանկ 7](#_Toc419368042)

[Աղյուսակների Ցանկ 8](#_Toc419368043)

[Ներածություն 9](#_Toc419368044)

[ԳՐԱԿԱՆՈՒԹՅԱՆ ԱԿՆԱՐԿ 12](#_Toc419368045)

[2.1 Ընդհանուր տեղեկություններ (ԱԱԾ) 12](#_Toc419368046)

[2.2 ADOL-C գրադարան 13](#_Toc419368047)

[2.2.1 ADOL-C օգտագործման օրինակներ և անհրաժեշտ կոդերի ձևափոխություններ 14](#_Toc419368048)

[2.3 Adept գրադարան 21](#_Toc419368049)

[2.3.1 Adept գրադարանի օգտագործման օրինակներ 21](#_Toc419368050)

[2.3.2 Adept գրադարանի հիշողության կառուցվածքը 23](#_Toc419368051)

[2.4 Matlab ծրագրային միջոցը (գրադարանը) 24](#_Toc419368052)

[2.4.1 Matlab – Symbolic Math ToolBoxTMծրագիր 25](#_Toc419368053)

[2.4.1.1 Symbolic Math ToolBoxTMծրագիր- մի քանի փոփոխականներով արտահայտության ածանցումը 26](#_Toc419368054)

[ԽՆԴՐԻ ԴՐՎԱԾՔԸ 29](#_Toc419368055)

[ՏԵՍԱԿԱՆ ԱՌՆՉՈՒԹՅՈՒՆՆԵՐ 31](#_Toc419368056)

[4.1 Անտառի ներկայացումը երկուական ծառի տեսքով 31](#_Toc419368057)

[4.2 Երկուական ծառի շրջանցման եղանակները 34](#_Toc419368058)

[4.3 Կարված երկուական ծառ 36](#_Toc419368059)

[4.4 Լեհական գրառում 38](#_Toc419368060)

[4.5 Ֆունկցիոնալ ծառ 38](#_Toc419368061)

[4.6 Ֆունկցիոնալ ծառ – կարված ֆունկցիոնալ ծառ ձևափոխություն 40](#_Toc419368062)

[4.7 Կարված ֆունկցիոնալ ծառը և ածանցյալը 41](#_Toc419368063)

[4.8 Մշակված գրադարանը 45](#_Toc419368064)

[4.8.1 Մշակված գրադարանի տիպերը և դրանց հնարավորությունները 45](#_Toc419368065)

[4.8.1.1 Parser տիպ 45](#_Toc419368066)

[4.8.1.2 function\_tree տիպ 49](#_Toc419368067)

[ՓՈՐՁԱՐԱՐԱԿԱՆ ՏԵԽՆԻԿԱ 66](#_Toc419368068)

[ՓՈՐՁՆԱԿԱՆ ՀԵՏԱԶՈՏՈՒԹՅՈՒՆՆԵՐ 68](#_Toc419368069)

[Եզրակացություն 71](#_Toc419368070)

[ԳՐԱԿԱՆՈՒԹՅԱՆ ՑԱՆԿ 72](#_Toc419368071)

|  |
| --- |
| Նկարների Ցանկ |
| Նկ 2.1 Հաշվարկային գրաֆի օրինակ …..…..…..…..…..…..…................................... 13 |
| Նկ 4.1.1 Անտառի օրինակ................................................................................................. 32 |
| Նկ 4.1.2 Անտառ – երկուական ծառ ձևափոխություն.................................................. 33 |
| Նկ 4.3 կարված երկուական ծառի գագաթների միջև կապերը.................................. 37 |
| Նկ 4.3.1 Երկուական ծառ, կարված երկուական ծառ ձևափոխություն................... 37 |
| Նկ. 4.5 Ֆունկցիոնալ ծառի օրինակ ................................................................................ 39 |
| Նկ. 4.6 Ֆ.Ծ. – Կ.Ֆ.Ծ ձևափոխություն.............................................................................. 40 |
| Նկ. 6 Հետազոտության գրաֆիկ 1.................................................................................... 68 |
| Նկ. 6.1 Հետազոտության գրաֆիկ 2................................................................................. 69 |

|  |
| --- |
| Աղյուսակների Ցանկ |
| Աղ 2.3.1 Adept գրադարանի օգտագործման օրինակ.............................................. 22 |
| Աղ 2.3.2.1 Հայտարարություններ ստեկ .................................................................... 24 |
| Աղ 2.3.2.1 Գործողությունների ստեկ ........................................................................ 24 |
| Աղ 4.3 Կարված և սովորական երկուական ծառերի դաշտերի արժեքները....... 36 |

ԳԼՈՒԽ 1

Ներածություն

**ՆԵՐԱԾՈՒԹՅՈՒՆ**

Հանրահաշվական արտահայտությունների վերլուծության և վերլուծման արդյունքում ստացված ֆունկցիաի ածանցման միջոցով լուծվող խնդիրների մի մեծ դաս է կազմում միջարկման խնդիրը և իրենից բխող դիֆերենցման և ինտեգրման խնդիրնեը:

Հանրահաշվական արտահայտությունների վերլուծության արդիական է, քանի որ ժամանակակից ինտեգրալ սխեմաների նախագծման համար մշակվող ծրագրային միջոցների առջև խնդիր է դրվում լուծել միջարման խնդիրը և դրան հարակից ածանցման և ինտեգրման խնդիրները: Հանրահաշվական արտահայտությունների վերլուծության և ֆունկցիաների հետազոտման բարդությունները հետևյալն են.

* տողային տեսքի վերլուծությունը
* կառուցվածքային ծառի կառուցումը
* ֆունկցիայի ածանցումը

Տողային տեսքի վերլուծության բարդությունը կայանում է նրանում, որ մարդու համար ընդունելի բանաձևրի տեսքը գործնականում բավականին բարդացնում են իրենց հետ աշխատելու եղանակները համակարգչի միջոցով: Դրա համար անհրաժեշտ-ություն է առաջանում կատարել լրացուցիչ ձևափոխություններ որոնք ավելի հասկանա-լի տեսքով են ներկայացնում մուտքային արտահայտությունը:

Տողային տեսքի վերլուծությունից հետո անհրաժեշտ է կառուցել կառուցված-քային ծառը:

Կառուցվածքային ծառի կառուցումից հետո, հնարավոր է դառնում կատարել մուտքային ֆունկցիայի հետազոտություններ, որոնք պետք է սկսվեն ֆունկցիան ածանցելուց:

ԳԼՈՒԽ 2

**Գրականության ակնարկ**

ԳՐԱԿԱՆՈՒԹՅԱՆ ԱԿՆԱՐԿ

# 2.1 Ընդհանուր տեղեկություններ (ԱԱԾ)

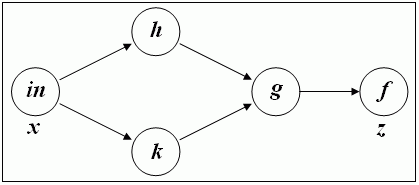
Ծրագրավորման մեջ, ֆունկցիաների ածանցման մոտեցումներից, գրականությունից հայտնի են մի շարք մետեցումներ: Դրանցից մեկն է Արագ Ավտոմատ Ածանցման (ԱԱԾ) մոտեցումը[5]:

ԱԱԾ մետեցումը հիմնված է ծրագրավորման լեզուների ճկունությունների օգտագործման վրա: Հիմնականում օգտագործվում է պոլիմորֆիզմի գաղափարը. մի տեսք, տարբեր իրականացուներ (single interface different implementations): Բացի պոլիմորֆիզմից օգտագործվում է նաև ածանցման համար հայտնի բանաձևերը: Դիտարկենք այդ բանաձևերը և դրանց մոտեցումները: Ենթադրենք ունենք բարդ ֆունկցիա y(x) = f(g(x)) և g(x) ու f(t) ածանցելի են x = x0 կոտում, ապա y(x) նույնպես ածանցելի x = x0 կետում և ածանցյալը որոշվում է dy/dx = df/dt\*dg/dx, որտեղ dg/dx հաշվարկում է ըստ x = x0 կոտում, իսկ df/dt t = g(x0) կետում: Իսկ եթե g(x) իր հերթին հանդիսանում է բարդ ֆունկցիա, ապա անհրաժեշտ է կրկնել ասյ գործողությունների այքան ժամանակ միչև հերթական g(x) չլինի պարզ ֆունկցիա:

Այս օրենքը կարելի է ընդհանրացնել մի քանի փոփոխականների ֆունկցիաների համար:

Մյուս կոնցեպտը որը պետք է իմանալ ԱԱԾ-ն օգտագործելու համար դա հաշվարկային գրաֆն է: Այն իրենից ներկայացնում է մուտաքյին ֆունկցիան պահելու համար նախատեսված գրաֆային կառուցվածք: Որի գագաթները իրենցից ներկայացնում են տարրական գործողությունները, իսկ կեղերը կապում են այդ գործողությունները իրենց արգումենտների հետ:

Օրինակ՝ z(x) = f(g(h(x), k(x))): Այս ֆունկցիայի հաշվարկային գրաֆը կունենա հետևյալ տեսքը՝



Նկ 2.1 Հաշվարկային գրաֆի օրինակ[5]:

# 2.2 ADOL-C գրադարան

ADOL-C (**A**utomatic **D**ifferentiation by **O**ver**L**oading in **C**++)[4] գրադարանը ինենից ներկայացնում ԱԱԾ եղանակով մշակված, որը հնարավոր է կապակցել ցանկացած ծրագրի հետ, որը կարող է կապակցվել C լեզվի հետ, օրինակ՝ Fortran, C/C++ և այլն:

ADOL-C օգտագործում է օպերատորների գերբեռնման սկզբույքը, որպեսի կարողանա հաշվարկել ուղղիղ կամ հակառակ ուղղությամբ ավտոմատ ածանցուման մեթոդով հետևյալ մեծությունները.

* ցանկածած աստիճանի ածանցյալը
* միակողմանի ածանցյալը ոչ պարզ դեպքերի համար (օրինակ fabs` մոդուլի ածանցյալը)

այն ֆունկցիայի համար, որը ներկայացված է C/C++ լեզվով: Օգտագործողը պետք է կատարի մի շարք փոփոխություններ իր կոդի մեջ, որպեսի կարողանա օգտվել այս գրադարանի հնարավորություններից:

Այդ փոփոխություններն են.

* վերասահմանել այն փոփոխականները որոնք օգտագործվելու են ֆունկցիայի մեջ, սահմանել դրանք որպես adouble նոր տիպի:
* նշել հաշվարկվող տիրույթը
* սահմանել կախված և անկախ փոփոխականները և տալ դրանց արժեքները
* նորից կոմպիլիացիայի ենթարկել և կապակցել ADOL-C գրադարանի հետ

## 2.2.1 ADOL-C օգտագործման օրինակներ և անհրաժեշտ կոդերի ձևափոխություններ

Ենթադրենք տրված է Կորտեսյան կոորդիանատային համակարգից ձևափոխություն դեպի գնդաձև բևեռային կոորդինատային համակարգ հետևյալ տեսքով[3]՝

F: R3 R3, y = F(x)

Օրինակ՝

y1 = , y2= arctan(/x3, y3 = arctan(x2/x1)

Համապատասխամ կոդը որը հաշվում է այս համակարգը հետևյալն է՝

#include <iostream>

unsing namespace std;

int main()

{

double x[3], y[3];

for (int i = 0; i < 3; ++i) {

... // initialize xi

}

y[0] = sqrt(x[0]\*x[0] + x[1]\*x[1] + x[2]\*x[2]);

y[1] = atan(sqrt(x[0]\*x[0] + x[1]\*x[1])/x[2]);

y[2] = atan(x[1]/x[0]);

cout<<''y1=''<<y[0]<<'' , y2=''<< y[1] <<'' , y3 =''<< y[2]<<endl;

return 0;

}

Այս կոդը ADOL-C-ով ներկայացնելուց առաջ անհրաժեշտ է ուսումնասիրել adtl::adouble տիպը:

adtl::adouble տիպը գտնվում է adouble.h գլխագրային (header) ֆայլում և adtl անուների տիրույթում (namespace): Այն հնարավոր է սահմանել երկու եղանակով.

Օրինակ առաջին՝

adouble x1(2, 1), x2(4, 0), y;

Այս դեպքում ստեղծվում են երեք փոփոխականներ x1, x2 և y, x1-ում պահվում է փոփոխականանի ընթացիկ 2 առժեքը և ածանցյալի 1 առժեքը, x2-ում պահվում 4 և 0 առժեքները համապատասխանաբար, իսկ y-ի առժեքները անորոշ են:

Օրինակ երկրորդ՝

adouble x1 = 2, x2 = 4, y;

…

x1.setADValue(1);

x2.setADValue(0);

Այս դեպքում փոփոխականները սկզբնավորվում են նույն արժեքներով, ինչ որ նախորդ օրինակում էին, միակ տարբերությունը այն է, որ այստեղ օգտագործվում է setADValue(double) ֆունկցիան, որը ինիցիալիզացնում է ածանցյալների արժեքները:

Ածանցյալի արժեքը ցանկացած պահին հասանելի է getADValue() ֆունկցիայի միջոցով, օրինակ՝

adouble y;

....

std::cout<<y.getADValue()<<std::endl;

Այժմ կարող ենք դիտարկել վերունշյալ կոդի տեսքը ADOL-C գրադարանի օգտագործման դեպքում.

#include<iostream>

using namespace std;

#dene ADOLC\_TAPELESS

#include<adouble.h>

typedef adtl::adouble adouble;

int main()

f

adouble x[3], y[3];

for (int i=0; i<3; ++i) // Initialize xi

...

x[0].setADValue(1); // derivative of f with respect to x1

y[0] = sqrt(x[0]\*x[0]+x[1]\*x[1]+x[2]\*x[2]);

y[1] = atan(sqrt(x[0]\*x[0]+x[1]\*x[1])/x[2]);

y[2] = atan(x[1]/x[0]);

cout<<"y1="<<y[0].getValue()<<" , y2="<<y[1].getValue ... ;

cout<<"dy2/dx1 = "<<y[1].getADValue()<<endl;

return 0;

}

Վեկտորների օգտագործման դեպքում կունենք հետևյալ տեսքը.

#include<iostream>

using namespace std;

#dene ADOLC\_TAPELESS

#dene NUMBER\_DIRECTIONS 3

#include<adouble.h>

typedef adtl::adouble adouble;

ADOLC\_TAPELESS\_UNIQUE\_INTERNALS;

int main()

{

adouble x[3], y[3];

for (int i=0; i<3; ++i) {

... // Initialize xi

for (int j=0; j<3; ++j) if (i==j) x[i].setADValue(j,1);

}

y[0] = sqrt(x[0]\*x[0]+x[1]\*x[1]+x[2]\*x[2]);

y[1] = atan(sqrt(x[0]\*x[0]+x[1]\*x[1])/x[2]);

y[2] = atan(x[1]/x[0]);

cout<<"y1="<<y[0].getValue()<<" , y2="<<y[1].getValue ... ;

cout<<"jacobian : "<<endl;

for (int i=0; i<3; ++i) {

for (int j=0; j<3; ++j)

cout<<y[i].getADValue(j)<<" ";

cout<<endl;

}

return 0;

}

Այժմ դիտարկենք ցանկացած i-երորդ ածանացյալի հաշվելու օրինակ, որպես ֆունկցիայի օրինակ դիտարկեն աստիճան բարձրացնող ֆունկցիան.

#include <adolc/adolc.h> // use of ALL ADOL-C interfaces

adouble power(adouble x, int n)

{

adouble z=1;

if (n>0) { // recursion and branches

int nh =n/2; // that do not depend on

z = power(x,nh); // adoubles are fine !!!!

z \*= z;

if (2\*nh != n)

z \*= x;

return z;

} // end if

else {

if (n==0) // the local adouble z dies

return z; // as it goes out of scope.

else

return 1/power(x,-n);

} // end else

} // end power

Այս ֆունկցիան սովորական աստիճան բարձրանցնող ֆունկցիայից տարբերվում միայն նրանով, որ double տիպի փոփոխականների փոխարեն օգտագործվում է adouble տիպի փոփոխականներ:

Իսկ ամբողջական ծրագիրը կունենա հետևյալ տեսքը.

#include ... // as above

int main()

{

int i,n,tag=1;

cout <<"COMPUTATION OF N-TH POWER (ADOL-C Documented Example)\n\n";

cout<<"monomial degree=? \n"; // input the desired degree

cin >> n;

// allocations and initializations

double\* Y[1];

\*Y = new double[n+2];

double\* X[1]; // allocate passive variables with

\*X = new double[n+4]; // extra dimension for derivatives

X[0][0] = 0.5; // function value = 0. coefficient

X[0][1] = 1.0; // first derivative = 1. coefficient

for(i=0;i<n+2;i++)

X[0][i+2]=0; // further coefficients

double\* Z[1]; // used for checking consistency

\*Z = new double[n+2]; // between forward and reverse

adouble y,x; // declare active variables

// beginning of active section

trace\_on(1); // tag = 1 and keep = 0

x <<= X[0][0]; // only one independent var

y = power(x,n); // actual function call

y >>= Y[0][0]; // only one dependent adouble

trace\_off(); // no global adouble has died

// end of active section

double u[1]; // weighting vector

u[0]=1; // for reverse call

for(i=0;i<n+2;i++) { // note that keep = i+1 in call

forward(tag,1,1,i,i+1,X,Y); // evaluate the i-the derivative

if (i==0)

cout << Y[0][i] << " - " << y.value() << " = " << Y[0][i]-y.value()

<< " (should be 0)\n";

else

cout << Y[0][i] << " - " << Z[0][i] << " = " << Y[0][i]-Z[0][i]

<< " (should be 0)\n";

reverse(tag,1,1,i,u,Z); // evaluate the (i+1)-st derivative

Z[0][i+1]=Z[0][i]/(i+1); } // scale derivative to Taylorcoeff.

return 1;

} // end main

# 2.3 Adept գրադարան

Adept(Automatic Differentiation using Expression Templates)[2] գրադարանը նույպես հիմնված է ԱԱԾ-ի վրա: Adept գրադարանը ավելի խնայողաբար է օգտագործում հիշողությունը և ավելի արագագործ է քան ADOL-C գրադարանը:

Adept գրադարանի հնարավորություններն են.

* Հաշվել ամբողջական Յակոբյան մատրիցը (Տրված է ոչ գծային ֆունկցիա, որը կոդավորված է C կամ C++ լեզվով, Adept կհաշվի դրա H=dy/x, որտեղ I տողի և j սյան հատման կետում գտնվող տարրը որոշվում է Hi,j = dyi/dxj: Յակոբյան մատրիցը օգտագործվում է Նյուտոն-Կոտեսի և Լեվենբերք-Մարքարթիի մինիմիզացման ալգորիթմներում)
* Հակառակ ռեժիմում դիֆերենցում (Սա կարևոր կեմպենենտ է օպտիմիզացման խնդիրներում, որտեղ ոչ գծային ֆունկցիան անհրաժեշտ է մինիմիզացնել, բայց վիճակների վեկտորը շատ տարրեր է պարունակում, որպեսի հնարավոր լինի հաշվել Յակոբյան մատրիցը)
* Ուղիղ ռեժիմում դիֆերենցում (Տրված է ոչ գծային ֆունկցիա y = f(x) և փոխազդեցություններ վեկտոր dx-ը, Adept-ը կհաշվի համապատասխան dy, որը կբխի f ֆունկցիայի գծայնացումից: Այս մեթոդում Adept-ը օպտիմզացումնում իր կոդը, հնարավոր է, որ արագագործ չլինի և օգտագործի մեծ հիշողություններ)

## 2.3.1 Adept գրադարանի օգտագործման օրինակներ

Մեզ հիմնականում հետաքրքրում է տարրական ֆունկցիաներով կազմված բազմանդամների ածանցյալները, հետևաբար դիտարկենք այդպիսի օինակ[3]:

|  |  |
| --- | --- |
| Ֆունկցիա | Ածանցյալ |
| y = 4; | dy = 0; |
| s = 2x0 + 3x12; | ds = 2dx0 + 6x1dx1; |
| y = y\*sin(s); | dy = sin(s)dy + y\*cos(s)ds |

Աղ 2.3.1 Adept գրադարանի օգտագործման օրինակ[1]

Ձախ մասում ներկայացված է հիմնական ֆունկցիան իր հաշվման քայլերով, իսկ աջում յուրաքանչյուր քայլին համապատասխանող ֆունկցիայի դիֆերենցիալը:

Այս ֆունկցիան հաշվող ծրագիը ունի շատ պարզ տեսք.

double algorithm(const double x[2])

{

double y = 4.0;

double s = 2.0\*x[0] + 3.0\*x[1]\*x[1];

y \*= sin(s);

return y;

}

Adept-ի միջոցով այս ֆունկցիան ածանցող ֆունկցիա գրելը բավականին նման է ADOL-C գրադարանի մոջոցով գրվող ֆունկցիաներին.

double algorithm\_ad(const double x\_val[2], // Input values

double\* Y\_ad, // Input-output adjoint

double x\_ad[2])

{ // Output adjoint

using namespace adept; // Import Stack and adouble

// from adept

Stack stack; // Where differential

// information is stored

adouble x[2] = {x\_val[0], x\_val[1]}; // Initialize adouble inputs

stack.new\_recording(); // Start recording derivatives

adouble Y = algorithm(x); // Version overloaded for

// adouble args

Y.set\_gradient(\*Y\_ad); // Load the input-output

stack.reverse(); // Run the adjoint algorithm

x\_ad[0] = x[0].get\_gradient(); // Extract the output adjoint x\_ad[1] = x[1].get\_gradient();

\*Y\_ad = Y.get\_gradient();

return Y.value(); // Return result of simple computation

}

## 2.3.2 Adept գրադարանի հիշողության կառուցվածքը

Այս բաժնում կդիտարկենք Adept գրադարանի հիշողության կառավարման եղանակները[1]:

Անհնար է ցանկացած adouble տիպի փոփոխականի համար առանձին պահել իր գրադիենտները, քանի որ այդպիսի շատ օբյեկտներ լոկալ են առանձին ցիկլերում կամ բլոկերում և կդառնային անորոշ այդ ցիկլերից կամ բլոկերից հետո: Դրա համար ստեղծվել է մի ընդհանուր հիշողության տիրույթ՝ ստեկ, որի մեջ իրենց գտնվելու ինդեքսներն են պահում առանձին փոփոխականները: Երբ adouble տիպի օբյեկտ է ստեղծվում, ապա նրան հատկացվում ստեկում հերթական ազատ ինդեքսը, իսկ ջնջվելուց հետո այդ ինեքսը վերադարձրվում է ստեկին:

Որպեսի ավելի հասկանալի լինի ստեկի օգտագործման եղանակները դիտարկեն Աղ 2.3.1 –ում բերված ֆունկցիայի օրինակը: Այս օրինակում մենք ունենք երեք հայտարարումներ, որոնցից յուրաքանչյուրին համապատասխանում է մի արտահայտություն, որը բաղկացած է զրո կամ ավելի գործողություններից: Այս ինֆորմացիան պահվում է հայտարարությունների ստեկում (statements stack) և գործողությունների ստեկում (operation stack).

|  |  |  |
| --- | --- | --- |
|  | Հայտարարություննեի ստեկ | |
| Գրառումներ  համար | Գրադիենտի ինդեքս (LHS) | Առաջին գործ. ին. |
| 0 | 3(dy) | 0 |
| 1 | 4(ds) | 0 |
| 2 | 3(dy) | 3 |
| 3 | 2(dy) | 5 |

|  |  |  |
| --- | --- | --- |
|  | Գործողություննեի ստեկ | |
| Գրառումներ  համար | Բազմապատիկ | Գրադիենտի ինդեքս (LHS) |
| 0 | 2.0 | 0(dx0) |
| 1 | 3.0x1 | 1(dx1) |
| 2 | 3.0x1 | 1(dx1) |
| 3 | sin(s) | 3(dy) |
| 4 | y\*cos(s) | 4(ds) |
| 5 | 1.0 | 3(dy) |

Աղ 2.3.2.1 Հայտարարություններ ստեկ[1]

Աղ 2.3.2.1 Գործողությունների ստեկ[1]

# 2.4 Matlab ծրագրային միջոցը (գրադարանը)

Matlab-ը բարձր մակարդակի լեզու է և ինտերակտիվ միջավայր թվային հաշվարկների, վիզուալիզացիայի և ծրագրավորման համար: Matlab-ը կարող է օգտագործվել, որպեսի հետազոտվեն մուտքային տվյալները, մշակված ալգորիթմները և ստեղծմեն մոդուլներ և ծրագրեր: Այս լեզուն կամ գործիքները և ներդրված մաթեմատիկական ֆունկցիաները լուծելու խնդիները շատ ավոլի արագ քան ավանդական ծրագրավորման լեզուները, ինչպիսին են C/C++ կամ Java-ն[6]:

Matlab-ի հնարավորությունները մեզ հետաքրքրող բաժնում՝ մաթեմատիկայի (Mathematics), հետևյալն են.

* գծային հանրահաշիվ
* հիմնական վիճակագրություն
* դիֆերենցում և ինտեգրում
* Ֆուրյեի ձևափոխություն
* և այլն

## 2.4.1 Matlab – Symbolic Math ToolBoxTMծրագիր

Այս բաժնում կդիարկենք թե ինչպես ածանցել ֆունկցիաները Synmbolic Math ToolBoxTM ծրագրային գործիքի միջոցով[6]: Սկզբի համար պետք է ստեղծել սիմվոլային արտահայտություն.

syms x

f = sin(5\*x)

այնուհետև կանչել diff(f) ֆունկցիան (f-ի ածանցում x-ի նկատմամբ).

ans = 5\*cos(5\*x)

Մեկ այլ օրինակ.

g = exp(x)\*cos(x);

y = diff(g) // իրական արդյունքը կլինի. y = exp(x)\*cos(x)-exp(x)\*sin(x)

Որպեսի գտնենք երկրորդ աստիճանի ածանցյալը անհրաժեշտ է կանչել հետևյալ ֆունկցիան.

diff(g, 2)

Նույն արդյունքը կստանանք, եթե երկու անգամ կանչենք diff ֆունկցիան.

diff(diff(g))

Որպեսի հաստատուն մեծության ածանցյալը հաշվենք, ահրաժեշտ սկզիբ այն հայտատրարել որպես սիմբոլանի արտահայտություն.

c = sym(‘5’)

diff(c)

ans = 0

եթե կանչենք միայն diff(5), ապա Matlab-ը կվերադարձնի ans = [], քանի որ 5 սիմբոլային արտահայտություն չէ:

### 2.4.1.1 Symbolic Math ToolBoxTMծրագիր- մի քանի փոփոխականներով արտահայտության ածանցումը

Որպեսի ածանցենք, մեկից ավելի սիբոլային փոփոխական պարունակող արտահայտություն անհրաժեշտ է նշել, թե փոփոխականի հանդեպ է ածանցվելու արտահայտությունը: Այնուհետև diff հրամանը կհաշվի մուտքային ֆունկցիայի մասնակի ածանցյալը ըստ նշված փոփոխականի:

Օրինակ, տրված է՝

syms s t

f = sin(s\*t)

diff(f, t) հրամանը կհաշվի df/dt ածանցյալը.

ans = s\*cos(s\*t)

Նմանապես diff(f, s).

ans = t\*cos(s\*t)

Եթե չնշվի, թե որ փոփոխականի նկատմաբ է անհրաժեշտ ածանցել, ապա Matlab-ը կվերցնի այն փոփոխականը, որը ավելի մոտ է գտնվում այբուբենում x-ին:

Մի քանի փոփոխականի ֆունկցիայի երկրորդ աստիճանի ածանցյալը ահշվելու համար ահնրաժեշտ է կանչել diff(f, t, 2), ans = -s^2\*sin(s\*t)

ԳԼՈՒԽ 3

**Խնդրի դրվածք**

ԽՆԴՐԻ ԴՐՎԱԾՔԸ

* Հանրահաշվական արտահայտությունների կառուցվածքային ծառի կառուցումը
* Հանրահաշվական արտահայտությունները ածանցող ալգորիթմի մշակումը
* Մշակված ալգորիթմի արագագործության՝ արտահայտության բարդությունից կախվածության հետազոտումը
* Մշակված ալգորիթմի հիմման վրա ծրագրավորման դինամիկ գրադարանի ստեղծումը

ԳԼՈՒԽ 4

**Տեսական առնչություններ**

ՏԵՍԱԿԱՆ ԱՌՆՉՈՒԹՅՈՒՆՆԵՐ

Ինչպես ասվեց ներածություն բաժնում հանրահաշվական արտահայտությունների վերլուծության և ֆունկցիաների հետազոտման բարդությունները հետևյալն են.

* տողային տեսքի վերլուծությունը
* կառուցվածքային ծառի կառուցումը
* ֆունկցիայի ածանցումը

Մինչ այս կետերին անրադառնալը, դիտարկեն մի քանի տվյալների կառուցվածքներ և տողային տեսքի ձրափոխություններ և ցույց տանք, թե ինչպես են դրանք կապված միմիանց:

# 4.1 Անտառի ներկայացումը երկուական ծառի տեսքով

Դիցուկ V={ v1,v2,v3,…,vn }-ն որևէ վերջավոր բազմություն է, իսկ F2(V)= { {u,v}/ u ∊ V , v ∊ V}-ն այդ բազմության երկու տարր պարունակող ենթաբազմությունների բազմությունն է և E ⊆ F2(V):

G=(V,E) կարգավորված զույգը անվանենք գրաֆ, V բազմության տարրերը՝ գրաֆի գագաթներ, իսկ E բազմության տարրերը՝ գրաֆի կողեր:

Եթե x= { u, v} ∊ E , ապա կասենք, որ x կողը կից է u և v գագաթներին: Այդ դեպքում կասենք նաև, որ u և v գագաթները կից են x կողին: Միևնույն կողին կից գագաթները անվանենք միմյանց հարևան գագաթներ:

Ենթադրենք տրված է G=(V,E) գրաֆը: Նրա գագաթների u1,u2,…,uk-1,uk հաջորդականությունը կանվանենք u1-ից uk ճանապարհ, եթե { u1,u2}, { u2,u3},…, { uk-1,uk} –ն G գրաֆի միմյանցից տարբեր կողեր են: u1-ից uk ճանապարհը կանվանենք ցիկլ, եթե u1 = uk :

G=(V,E) գրաֆը անվանենք կապակցված գրաֆ, եթե նրա ցանկացած u և v գագաթների համար գոյություն ունի u-ից v ճանապարհ:

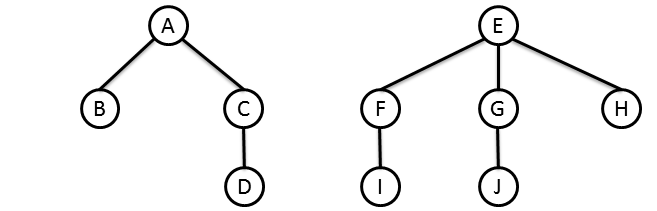
G=(V,E) գրաֆը անվանենք ծառ, եթե նա կապակցված է և ցիկլ չի պարունակում:

G=(V,E) ծառը կանվանենք երկուական ծառ, եթե նրա յուրաքանչյուր գագաթից դուրս է գալիս երկու կող:

Ծառերի բազմությունը կանվանենք անտառ: Ծառի գագաթներից յուրաքանչյուրի ենթածառերը ձևավորում են անտառ:

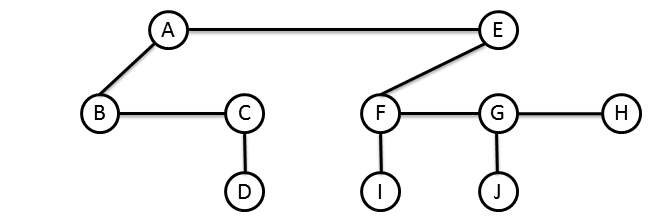
Այժմ դիտարկենք մի ալգորիթմ, որը հնարավորություն է տալիս ձևափոխել անտառը երկուական ծառի:

Գոյություն ունի բնական եղանակ ցանկացած անտառի ներկայացման երկուական ծառի տեսքով: Դիտարկենք հետևյալ անտառև որը կազմված է երկու ծառերից.



Նկ 4.1.1 Անտառի օրինակ

Երկուական ծառ կստանանք, եթե կապակցենք նույն ծնողին պատկանող գագաթները և հեռացնենք բոլոր կապերը ծնեղների հետ բացառությամբ առաջին կապի:



Նկ 4.1.2 Անտառ – երկուական ծառ ձևափոխություն

Այժ, եթե թեքենք այս ծառը 45Օ և թեթևակի տեղափոխենք գագաթները, ապա կարող ենք համոզվել, որ սա իրականում համապատասխանում է երկուական ծառի սահմանմանը:

Եվ ընդհակառակը, եթե կատարենք այս գործողությունները հակառակ հերթականությամբ, ապա ցանկացած երկուական ծառի համապատասխանում է միայն մեկ անտառ:

Նկ 4.1.2 –ում ներկայացված տեսքը ունի շատ կաևոր նշանակություն և կոչվում է բնական համապատասխանություն (natural correspondence) անտառի և երկուական ծառի միջև:

Այս ձևափոխության ընդհանուր մետեցումը կձևակերպենք ստորև.

Տրված է F = (T1, T2 , . . ., Tn) որոշակի անտառը: Այս անտառին համապատասխանող երկուական ծառը B(F) կարելի է խիստ որոշել՝ հետևյալ եղանակով.

* Եթե n = 0, ապա B(F) դատարկ է:
* Եթե n > 0, ապա B(F) –ի արմատը (root) հանդիսանում է T1,իսկ B(T11, T12, . . . , T1m)

հանդիսանում է ձախ ենթածառ B(F) ծառի, որտեղ T-երը T1 –ի ենթածառեր, իսկ B(T2,. . . , Tn) հանդիսանում է B(F) ծառի աջ ենթածառ:

Այս ձևափոխությունը միանշանակ արտապատկերում Նկ 4.1.1 ներկայացված ծատը Նկ 4.2.2 տեսքի ծառի:

# 4.2 Երկուական ծառի շրջանցման եղանակները

Գոյություն ունեն բավականին մեծ թվով ալգորիթմներ, որոնք աշխատում են ծառանման կառուցվածքների հետ: Այդ ալգորիթմներից ամենաշատը հանդիպում է ծառի շրջանցում (travesing) գաղափարը: Ծառի այսպիսի ուսումնասիրման եղանակներով, ծառի յուրաքանչյուր գագաթ դիտարկվում է միայն մեկ անգամ, իսկ ծառի աբեղջական շրջանցումը տալսի է գագթների գծային դասակարգում, որը հնարավորություն է տալիս հեշտացնելու ալգորիթմները, քանի որ այդ դեպքում կարելի է լինիում օգտագործել հաջորդ գագաթ կամ հանգույց գաղափարը:

Երկուական ծառի շրջանցման համար կարելի է օգտագործել երեք հիմնորեն տարբեր ալգորիթմներ.

* ուղիղ հերթականությամբ (preorder)
* կենտրոնացված հերթականությամբ (inorder)
* հակառակ հերթականությամբ (postorder)

Այս երեք եղենակները որոշվում ռեկուրսորեն: Եթե երկուական ծառը դատարկ է, ապա շրջանցման համար ոչ մի բան ահնրաժեշտ չէ անել, հակառակ դեպքում շրջանցումը կատարվում է հետևյալ փուլերով.

Ուղիղ հերականությամբ շրջանցում՝

* հասնել (դիտարկել) արմատիը (root)
* անցնել ձախ ենթածառը
* անցնել աջ ենթածառը

Կենտրոնացված հերթականությամբ շրջանցում՝

* անցնել ձախ ենթածառը
* դիտարկել արմատը
* անցնել աջ ենթածառը

Հակառակ հերթականությամբ շրջանցում

* անցնել ձախ ենթածառը
* անցնել աջ ենթածառը
* դիտարկել արմատը

Դիտարկենք ոչ ռեկուրսիվ եղանակով ծառի շրջանցում սիմետրիկ եղանակով, որը օգտագործում է ստեկ:

Ենթադրենք T ցուցիչ է երկուական ծառի վրա, իսկ A՝ օգնական ստեկի.

T1. Դատարկել A ստեկը և տեղադրել գագթի P ցուցիչը T վրա P = T:

T2. Եթե P դատարկ է անցնել T4 քայիլն:

T3. A մեջ ավելացնել P, այնուհետև P = LLINK(P) և վերադառնալ T2 քայլին:

T4. Եթե A դատարկ է, ապա դաթարացնել ալգորիթմի աշխատանքը, հակառակ դեպքում P վերագրել A առաջին առժեքը:

T4. Դիտարկել NODE(P), այնուհետև P = RLINK(P) և վերադառնալ T2 քայլին:

Կատարենք մի քանի նշանակումներ, որոնք հետագայում կօգտագործենք այլ ալգորիթմների նկարագրության ժամանակ:

P$ նշանակենք սիմետրիկ շրջանցման ժամանակ NODE(P) հաջորդող գագաթի հասցեն:

$P նշանակենք սիմետրիկ շրջանցման ժամանակ NODE(P) նախորդող գագաթի հասցեն:

Դիտարկենք ծառային մի կառուցվածք, որը խնայում է ժամանկը արդեն և օգտագործում է հատկացված հիշողությունը ամբողջությամբ, չթողնելով որևէ անորոշ դաշտ:

# 4.3 Կարված երկուական ծառ

Դիտարկենք երկուական ծառ, որի գագաթները ունեն հետևյալ դաշտերը.

* left
* right
* value

որտեղ left-ը ցուցիչ է դեպի տվյալ գագաթին կից ձախ գագաթը, իսկ right-ը ցուցիչ է դեպի տվյալ գագաթին կից աջ գագաթը, value տվյալ գագաթում պահվող տվյալն է:

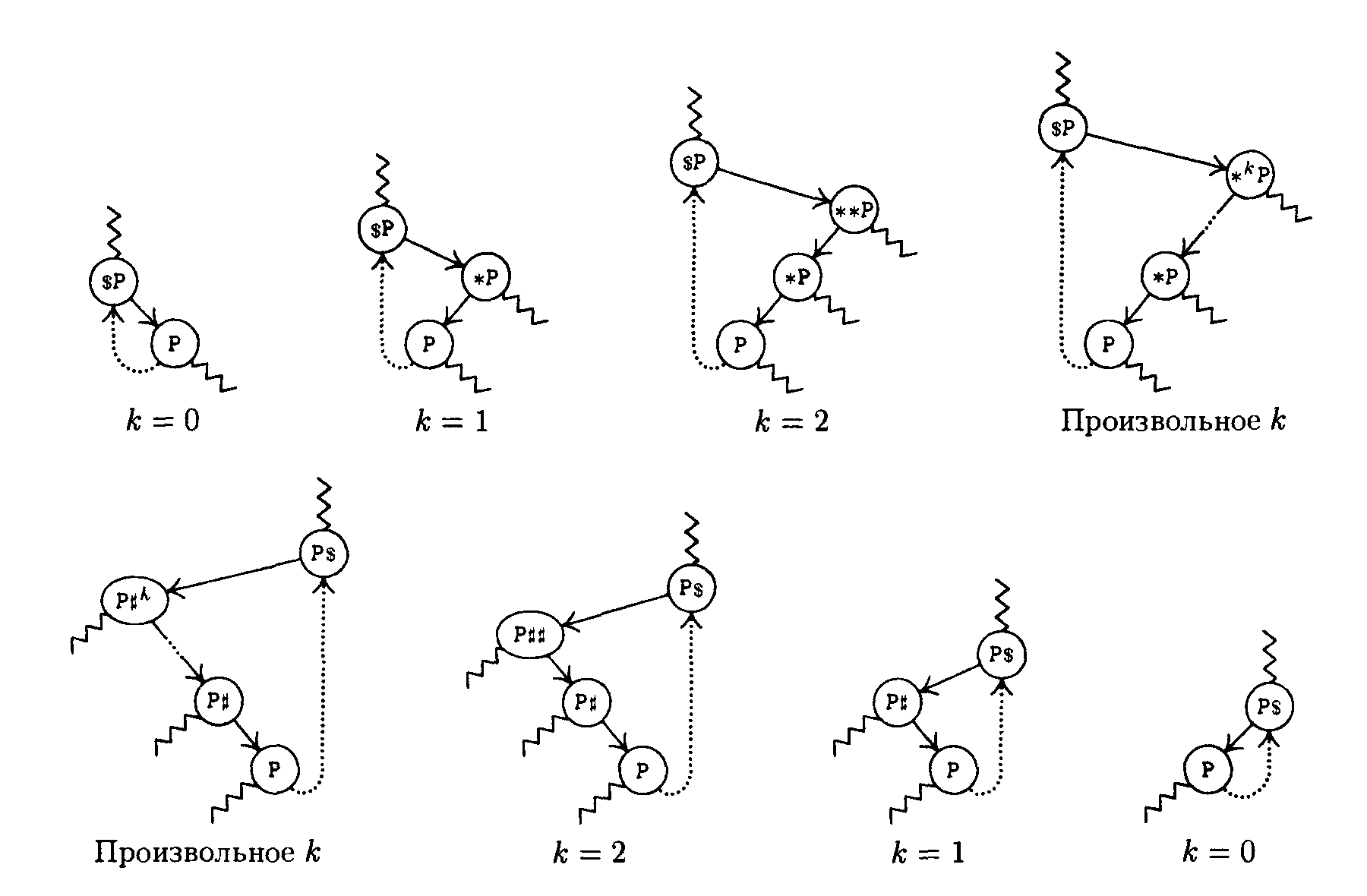
Շատ դեպքեր են լինում, որտեղ left կամ right դաշտերի համար հատկացում է հիշողություն բայց, այդ դաշտերից մեկը, իսկ տերև գագաթների դեպքում երկուսը, օգտակար տվյալ չեն պարունակում կախված տվյալ ծառի կառուցվածքից:

Այս հիշողությունների օգտակար օգտագործման եղանակներ են առաջարկել Ա. Պերլիսը և Չ. Տորնտոնը, որոնք ստեղծել են կարված եկուական ծառ գաղափարը: Յուրաքանչյուր գագաթում ավելացնելով երկու լրացուցիչ դաշտեր (rtag և ltag), որոնք կասեն արդյոք left կամ right իրական կողեն են կարեր, կարելի կլինի սահամանել կարված երկուական ծառի գագաթնեի դաշտերի առժեքները հետևյալ կերպ.

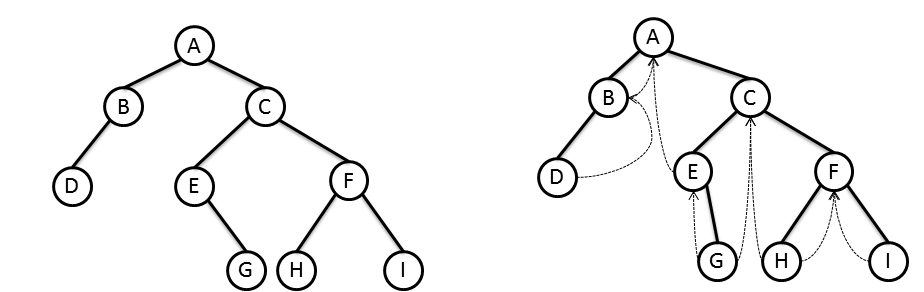
|  |  |
| --- | --- |
| Սովորական ներկայացում | Կարված ներկայացում |
| LLINK(P) = Λ | LTAG(P) = 1, LLINK(P) =$P |
| LLINK(P) = Q != Λ | LTAG(P) = 0, LLINK(P) = Q |
| RLINK(P) = Λ | RTAG(P) = 1, RLINK(P) =P$ |
| RLINK(P) = Q != Λ | RTAG(P) = 0, RLINK(P) = Q |

Աղ 4.3 Կարված և սովորական երկուական ծառերի դաշտերի արժեքները

Նկ 4.3 ցույց է տրված կարված երկուական ծառի գագաթների միջև կապերը



Նկ 4.3 կարված երկուական ծառի գագաթների միջև կապերը[7]



Նկ 4.3.1 Երկուական ծառ, կարված երկուական ծառ ձևափոխություն

# 4.4 Լեհական գրառում

Լեհական գրառումը, որը հայտնի է որպես պրեֆիքս լեհական գրառում կամ պրեֆիքս գրառում, եղանակ է տրամաբանական, մաթեմատիկական և հանրահաշվական արտահայտութոյւնների գրառման: Իր տարբերող հատկանիշն այն է, որ գործողությունները գրառման մեջ գրվում են իրենց արգումենտներից առաջ[9]:

Երփեմն լեհական գրառում ասելով նաև հասկանում են պոստֆիքս գրառում, որտեղ գործողության արգումենտները գրվում են ավելի շուտ քան գործողությունները:

Որպես օրինակ դիտարկենք հետևյալ հանրահաշվական արտահայտությունը և այն ձևափոխենք և՛ պրեֆիքս, և՛ պոստֆիքս գրառման:

y = 3ln(x+1) – a/

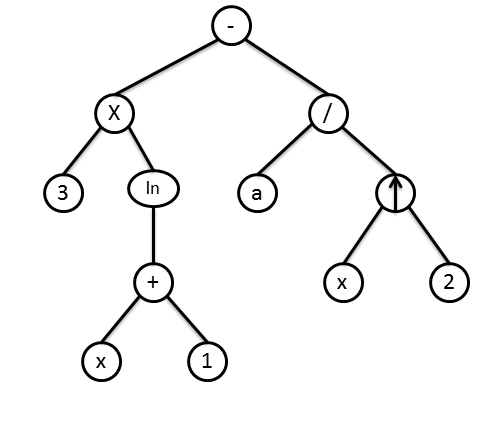
y = - \* 3 ln + x 1 / a ↑ x 2 պրեֆիքս գրառում (գործողություն, արգումենտներ)

y = 3 x 1 + ln \* a x 2 ↑ / - պոստֆիքս գրառում (արգումենտներ, գործողություն)

# 4.5 Ֆունկցիոնալ ծառ

Ֆունկցիոնալ ծառ կանվանենք այն երկուական ծառը, որի գագաթուներում գտնվող գործողությունների արգումմենտները կհանդիսանան այդ գագթին կից և դրա երեխա հանդիսացող գագաթներում գրված տվյալները:

Այդպիսի ծառի օրինակ է հետևյալ ծառը.



Նկ. 4.5 Ֆունկցիոնալ ծառի օրինակ

Նկ. 4.5 ներկայացված ծառը հանդիսանում է հետևյալ արտահայտության ֆունկցիոնալ ծառը. y = 3ln(x+1) – a/:

Այժ դիտարկենք այն ագոլիթմը, որի մի միջոցով հնարավոր է եղել տեղծել այս ծառային կառուցվածքը սովորական տեղային տեսքով գրված հանրահաշվական արտահայտությունից:

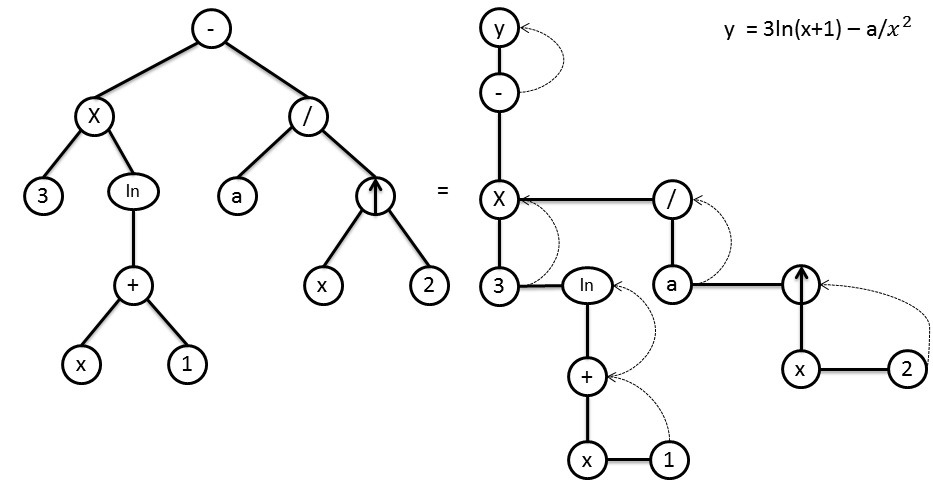
Մինչ բուն ծառի կառուցմանը անցնելը անհրաժեշտ է տողային տեսքի արտահայտությունը ձևափոխել և գրել լեհական գրառման պոստֆիքս եղանակով:

Պոստֆիքս եղանակով գրված տողի վրայով անցնելիս, յուրաքանչյուր հանդիպած հաստատուն մեծություն կամ փոփոխական ավելացվում է ծառի գագթին համապատասխքնող կառուցվածքի մեջ և այդ գագաթը ավելացվում է ստեկի մեջ, այնուհետը, եթե ընթացիկ սիմբոլը գործողություն է կամ ֆունկցիա այն ավելացվում ծառի գագաթի մեջ, ստեկից հանվում են այնքան տարրեր որքան այդ գործողության կամ ֆունկցիայի արգումենտների քանակն է և ավելացվում այդ գագաթին որպես երեխաներ և այդ գագաթը ավելացվում է ստեկի մեջ, եթե մինչը այդ գագաթի ավելացնելը ստեկի գագաթին գտնվող տարրը նույնպես գործողություն է պարունակում կամ ֆունկցիա, ապա նրա հանդեպ նույնպես կիրառվում են վերոնշյալ քայլերը:

Այս գործողությունը կատարվում այքան ժամանակ մինչև ստեկը դատարկվի:

Եթե ֆունկցիոնալ ծառը շրջանցվի հակառակ հերթականությամբ և գագաթների մեջի տվյալները տպվեն, ապա կնկատենք, որ այդ տողը իրենից ներկայացնում է մուտքային ֆունկցիայի պոստֆիքս գրառումը:

# 4.6 Ֆունկցիոնալ ծառ – կարված ֆունկցիոնալ ծառ ձևափոխություն

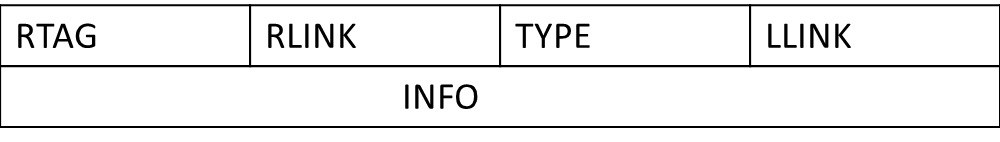
Այժմ կարող ենք ֆունկցիոնալ ծառը ձևափոխել և ստանալ կարված ֆունկցիոնալ ծառ: Դիտարկենք մեզ հայտնի օրինակը.

Նկ. 4.6 Ֆ.Ծ. – Կ.Ֆ.Ծ ձևափոխություն

# 4.7 Կարված ֆունկցիոնալ ծառը և ածանցյալը

Այժմ դիտարկենք ալգորիթմ, որի մուտքին գալիս է կարված ֆունկցիոնալ ծառ, և ելքում ստացում այդ ծառի մեջ զետեղված ֆունկցիային համապատասխանող ածանցյալ ֆունկցիայի կարված ֆունկցիոնալ ծառը[7]:

Կատարենք մի շարք նշանակումներ որոնք հետագայում կհեշտացնեն ալգորիթմի ներկայացումը[7]:



* TYPE
  + 0 – հաստատուն մեծություն
  + 1 – փոփոխական մեծություն (սիմբոլ)
  + 2> – գործողություններ
* RTAG – ձախ թեգ
* RTAG – ձախ հղում
* LLINK – աջ հղում
* INFO – TYPE տիպի պահվող մեծություն

Ալգորիթմը կներկայացնենք պսեֆդո կոդի տեսքով.

* Ինիցիալիզացիա
  + P = Y$
* Ածանցում
  + P1 = LLINK(P)
  + if (P1 != 0) Q1 = RLINK(P1)
  + DIFF(TYPE(P)) (Q = ածանցման արդյունքում ստացված ծառի արմատը)
* Կապերի վերականգնում
  + If (TYPE(p) == BinOp) RLINK(P1) = P2
* Անցում դեպի P$
  + P2 = P; P = P$; if (RTAG(P2) == 0) RLINK(P2) = Q
* Անցման ավարտի ստուգում
  + If (P != Y) goto stage 2; else LLINK(DY) = Q; RLINK(Q) = DY; RTAG(Q) = 1

DIFF ֆունկցիայի իրականացումը կներկայացնենք հետագայում, մինչ այդ վերհիշենք ածանցման մի քանի կանոն.

D(x) = 1

D(a) = 0

D(ln(u)) = D(u)/u

D(-u) = -D(u)

D(u + v) = D(u) + D(v)

D(u - v) = D(u) – D(v)

D(u \* v) = D(u) \* v + u \* D(v)

D(u / v) = D(u) / v – (u \* D(v)) / (v ↑ 2)

D(u ↑ v) = D(u)\* (v\* (u ↑ (v - 1))) + ((ln (u) \* D(v)) \* (u ↑ v)

Դիտարկենք մի քանի ֆունկցիաներ որոնք կօգտագործեքն DIFF ֆունկցիայում.

TREE(x, u, v) – ստեղծում է նոր ծառ x արմատով և u և v ենթածառերով.

W = AVAIL, INFO(W) = x, LLINK(W) = u, RLINK(u) = v, RTAG(u) = 0, RLINK(v) = W, RTAG(v) = 1

TREE(x, u) – ստեղծում է նոր ծառ x արմատով և u ենթածառերով.

W = AVAIL, INFO(W) = x, LLINK(W) = u, RLINK(u) = W, RTAG(u) = 1

TREE(x) - ստեղծում է նոր ծառ x արմատով, որը հանդիսանում է ծառի տերև.

W = AVAIL, INFO(W) = x, LLINK(W) = Λ

Այժմ կարող ենք դիտարկել DIFF ֆունկցիայի իրականացումը[7].

Զրոյական օպերտորներ (հաստատուններ և փոփոխականներ).

* DIFF[0] (NODE(P) հաստատուն է) Q = TREE(0)
* DIFF[1] (NODE(P) փոփոխական է) IF INFO(P) == x THEN Q = TREE(1) ELSE TREE = TREE(0)

Ունար օպերատորներ.

* DIFF[2] (NODE(P) == ln) IF INFO(Q) != 0, THEN Q = TREE(''/'', Q, COPY(P1))
* DIFF[3] (NODE(P) == neg) IF INFO(Q) != 0, THEN Q = TREE(''neg'', Q)

Երկուական օպերատորներ.

* DIFF[4] (+ օպեչատոր) IF INFO(Q1) == 0 THEN AVAIL = Q ELSE IF INFO(Q) == 0, AVAIL = Q; Q = Q1 ELSE Q = TREE(''+'', Q1, Q)
* DIFF[5] (- օպերատոր) IF INFO(Q) == 0 THEN AVAIL = Q1; Q =Q1 ELSE IF INFO(Q1) == 0 THEN AVAIL = Q1; Q = TREE(''neg'', Q) ELSE Q = TREE(''-'', Q1, Q)
* DIFF[6] (\* օպերատոր) IF INFO(Q1) != 0 THEN Q1 = MUL(Q1, COPY(P2)); IF INFO(Q) != 0 THEN Q =MULL(COPY(P1), Q); GOTO DIFF[4]
* DIFF[7] (/ օպերատոր) IF INFO(Q1) != 0 THEN Q1 = TREE (''/'', Q1, COPY(P2)); IF INFO(Q) != 0 THEN Q = TREE(''/'', MULL(COPY(P1), Q), TREE(''↑'', COPY(P2), TREE(2)))
* DIFF[8] (↑ օպերատոր) IF INFO(Q1) != 0 THEN R = COPY(P1) IF TYPE(P2) != 2 THEN R = TREE(''↑'', TREE(INFO(P2) - 1)); IF TYPE(P2) != 0 THEN R = TREE(''↑'', R, TREE(''-'', COPY(P2), TREE(1))); Q1 = MULL(Q1, MULL(COPY(P2), R));; IF INFO(Q) != 0 THEN Q = TREE(''\*'', MULT(TREE(''ln'', COPY(P1)), Q), TREE(''↑'', COPY(P1), COPY(P2))) GOTO DIFF[4]

COPY ֆունկցիան պարզապես կրկնորինակում է տրված ենթածառը, իսկ MULL ֆունկցիան իրականացված է հետևյալ կերպ[7].

MULL(U, V)

IF INFO(U) == 1 && TYPE (U) == 0 THEN AVAIL = U RETURN U;

IF INFO(V) == 1 && TYPE (V) == 0 THEN AVAIL = V RETURN V;

RETURN TREE(''\*'', U, V)

Այս, ալգորթմի աշխատանքի բարդությունը գնահատելու համար առանձին կգնահատենք նրա մուտքին եկած ծառի կառուցման բարդությունը և բուն ալգորիթմի բարդությունը:

Լեհական գրառում ստանալու համար օգտագործված ագլորիթմի բարդությունը գնահատվում է Θ(n), որտեղ n-ը մուտքային արտահայտության տրամաբանական տարրեի քանակն է:

Լեհական գրառումից ֆունկցինալ ծառի կառուցման համար օգտագործված ալգորիթմի բարդությունը նույնպես Θ(n) է:

Այս երկու դեպքում էլ օգտագործվում է ստեկ, որը Օ(n) կարգի հիշողություն է պահանջում:

Քանի որ բուն ալգորիթը իրենից ներկայացնում է ֆունկցիոնալ ծառի շրջանցում սիմետրիկ եղանակով և յուրաքանչյուր քայլում միջինում Θ(1), իսկ վատագույն դեպքում O(k) բարդությամբ գործողություն, որտեղ k պատճենվող ենթածաոի տարրեի քանակն է, ապա ածանցման ալգորիթմի բարդությունը վատագույն դեպքում կլինի Θ(n)\* O(k), հիմնականում n >> k և բարդությունը կարելի է գրել O(n), միայն եթե n տարրերն էլ իրենցից ներկայացնեն աստիճան բարձացնելու գործողություններ (որը վատագույն դեպքն է), ապա բարդությունը կլինի Օ(n2): Միջին դեպքում նույնպես բարդութույնը Օ(n2) է:

# 4.8 Մշակված գրադարանը

Վերոնշյալ ալգորիթմների և տվյալնեի կառուցվածքի հիմման վրա մշակվել և ստեղծվել է ծրագրավորման մի դինամիկ գրադարան, որը հնարավորություն է տալիս.

* մշակել տողային տեսքով ներկայացված արտահայտություննրը
* ստանալ այդ արտահայտությունների Լեհական գրառումը
* հաշվել արտահայտությունների արժեքները փոփոխականների և պարամետրերի որոշակի արժեքների դեպքում
* ածանցել այդ արտահայտությունները
* հաշվել ածանցյալի արժեքը, տրված կետերում
* այլ ալգորիթմների մուտքին հարմար տեսքով ֆունկցինալ ծառ կառուցել և ստանալ գրդարանում ներառված տիպերի միջոցով:

## 4.8.1 Մշակված գրադարանի տիպերը և դրանց հնարավորությունները

### 4.8.1.1 Parser տիպ

Parser տիպը պատասխանատվություն է կրում հանրահաշվական արտահայտության.

* վերլուծության
* ճշտության ստուգման
* Լեհական գրառման տեսքի ստացման
* Ֆունկցիոնալ ծառի կառուցման համար hարմար տեսքի բերման համար:

Parser տիպի ինտերֆեյսն է.

class parser

{

private:

parser();

parser(const parser&);

parser& operator=(const parser&);

public:

static parser\* get\_instance();

public:

typedef std::list<fobject\*> function;

static function parse(const std::string&);

public:

void add\_variable(char v);

public:

std::string to\_postfix(const std::string&);

public:

std::string to\_prefix(const std::string&);

private:

typedef std::list<std::string> f\_string;

public:

bool is\_action(char a);

bool is\_variable(char v);

bool is\_function(const std::string&, std::string&);

};

Այս տիպը մշակվել է singleton եղանակով, այսինք այս տիպի օբյեկտը միակն է ամբողջ ծրագրում:

Այժմ կդիտարկենք parser տիպի օգտագործման օրինակներ.

Օրինակ 1՝

#include <parser.h>

#include <cassert>

int main()

{

parser\* p = parser::get\_instance();

assert(p != 0);

std::string str;

std::cin>>str;

std::cout<<p->to\_postfix(str)<<std::endl;

return 0;

}

Օրինակ 2՝

#include <parser.h>

#include <cassert>

int main()

{

parser\* p = parser::get\_instance();

assert(p != 0);

std::string str;

std::cin>>str;

std::cout<<p->to\_prefix(str)<<std::endl;

return 0;

}

Օրինակ 3՝

#include <parser.h>

#include <cassert>

int main()

{

parser\* p = parser::get\_instance();

assert(p != 0);

char v;

std::cin>>v;

std::string str;

std::cin>>str;

p->add\_vaiable(v);

parser::function f = p->parse(str);

return 0;

}

### 4.8.1.2 function\_tree տիպ

Այս տիպը պատասխանատու է ֆունկցիոնալ ծառի կառուցման և այդ ծառի միջոցով.

* ֆունկցիայի առժեքի հաշվման
* ֆունկցիայի ածանցման
* ֆունկցիայի ածանցյալի առժեքի հաշվան՝ փոփոխականների կոնկրետ առժեքների դեպքում:

function\_tree տիպի ինտերֆեյսն է.

class function\_tree : public tree<fobject\*>

{

public:

typedef std::list<fobject\*> function;

public:

function\_tree(const function& f);

public:

typedef std::list<std::string> f\_string;

function\_tree(const f\_string& function);

public:

void set\_var\_value(char, double);

void set\_par\_value(char, double);

double count\_value();

public:

function\_tree diff();

private:

typedef node<fobject\*> f\_node;

void diff(f\_node\* p, f\_node\*& p1, f\_node\*& p2, f\_node\*& q1, f\_node\*& q);

void create\_tree(f\_node\* x, f\_node\* u, f\_node\* v);

void create\_tree(f\_node \*x, f\_node \*u);

f\_node\* copy(f\_node\* u);

f\_node\* mult(f\_node\* u, f\_node\* v);

private:

void save\_iterators();

typedef std::map<f\_node\*, iterator> node\_map;

node\_map m\_nodes\_to\_next;

};

Քանի որ, այս տիպը ժառանգված է tree տիպից կարիք կա նաև ներկայացնելու tree տիպի ինտերֆեյսը նույնպես, որպեսի հասկանալի լինի function\_tree բոլոր հնարավորությունները:

template <typename T>

class tree

{

protected:

template <typename T1> class node;

public:

template <typename T2 = T>

class iterator\_in\_order

{

public:

/// Prefix

iterator\_in\_order& operator++()

{

next\_node();

return \*this;

}

/// Postfix

iterator\_in\_order& operator++(int i)

{

iterator\_in\_order old = \*this;

int j = 0;

while (j++ < i) {

next\_node();

}

return old;

}

iterator\_in\_order& operator--()

{

prev\_node();

return \*this;

}

iterator\_in\_order& operator--(int i)

{

iterator\_in\_order old = \*this;

int j = 0;

while (j++ < i) {

prev\_node();

}

return old;

}

bool operator ==(const iterator\_in\_order& i)

{

return m\_tree == i.m\_tree && m\_node == i.m\_node;

}

bool operator !=(const iterator\_in\_order& i)

{

return !operator ==(i);

}

T2& operator\*()

{

return m\_node->m\_value;

}

private:

void next\_node()

{

if (m\_node == 0) {

return;

}

node<T2>\* p;

if (m\_node->m\_right != 0 && m\_node->m\_rtag == 0)

{

m\_node = m\_node->m\_right;

while (m\_node->m\_left != 0) {

m\_node = m\_node->m\_left;

}

}

else

{

p = m\_node->m\_parent;

while (p != 0 && (m\_node == p->m\_right && p->m\_rtag == 0))

{

m\_node = p;

p = p->m\_parent;

}

m\_node = p;

}

if (m\_node == m\_tree->m\_root) {

m\_node = 0;

}

}

void prev\_node()

{

if(m\_node == 0) {

return;

}

if ((m\_node->m\_parent != 0) && (m\_node->m\_parent->m\_right == m\_node)) {

m\_node = m\_node->m\_parent;

return;

}

while((m\_node->m\_parent != 0) && (m\_node->m\_parent->m\_left == m\_node)) {

m\_node = m\_node->m\_parent;

}

m\_node = m\_node->m\_parent;

}

private:

iterator\_in\_order(const tree<T2>\* t, node<T2>\* n = 0)

: m\_node(n), m\_tree(t)

{

}

public:

iterator\_in\_order(const iterator\_in\_order<T2>& i)

{

m\_node = i.m\_node;

m\_tree = i.m\_tree;

}

public:

iterator\_in\_order& operator=(const iterator\_in\_order<T2>& i)

{

if (&i == this) {

return \*this;

}

m\_node = i.m\_node;

m\_tree = i.m\_tree;

return \*this;

}

iterator\_in\_order()

: m\_node(0), m\_tree(0)

{

}

private:

node<T2>\* m\_node;

const tree<T2>\* m\_tree;

friend class tree<T2>;

friend class function\_tree;

};

protected:

template <typename T1 = T>

class node

{

public:

node(const T1& v, node<T1>\* p = 0)

: m\_value(v) , m\_left(0)

, m\_right(0) , m\_parent(p)

, m\_ltag(false) , m\_rtag(false)

{}

node()

: m\_left(0)

, m\_right(0) , m\_parent(0)

, m\_ltag(false) , m\_rtag(false)

{}

private:

T1 m\_value;

private:

node<T1>\* m\_left;

node<T1>\* m\_right;

node<T1>\* m\_parent;

private:

bool m\_ltag;

bool m\_rtag;

friend class tree<T1>;

friend class iterator\_in\_order<T1>;

friend class function\_tree;

};

public:

tree()

{

m\_root = new node<T>();

}

public:

enum position {left, right};

public:

iterator\_in\_order<T> insert(const T& v, iterator\_in\_order<T> i = begin(), position p = left)

{

if (i == end()) {

return end();

}

node<T>\* n = new node<T>(v, i.m\_node);

if (p == left) {

i.m\_node->m\_left = n;

} else {

i.m\_node->m\_right = n;

}

return iterator\_in\_order<T>(this, n);

}

public:

typedef iterator\_in\_order<T> iterator;

public:

iterator begin()

{

node<T>\* p = m\_root;

while(p != 0 && p->m\_left != 0) {

p = p->m\_left;

}

return iterator(this, p);

}

iterator end()

{

return iterator(this);

}

public:

void convert\_to\_levels()

{

std::queue<node<T>\*> q1, q2;

q1.push(m\_root);

while(!q1.empty()) {

node<T>\* on = 0;

while(!q1.empty()) {

node<T>\* n = q1.front();

if (n->m\_left != 0) {

q2.push(n->m\_left);

}

if (n->m\_right != 0) {

q2.push(n->m\_right);

}

if (on != 0) {

if (n->m\_parent == on->m\_parent && n->m\_parent->m\_left != n) {

if (n->m\_parent->m\_right == n) {

n->m\_parent->m\_right = 0;

}

on->m\_right = n;

n->m\_parent = on;

}

}

on = n;

q1.pop();

}

if (on != 0) {

on->m\_right = 0;

}

std::swap(q1, q2);

std::queue<node<T>\*> empty;

std::swap(q2, empty);

}

}

protected:

node<T>\* get\_root()

{

return m\_root;

}

protected:

node<T>\* m\_root;

friend class iterator\_in\_order<T>;

};

Tree տիպի հանրավորությունները կախված չեն գրադարանի մնացած մասնագիտացված մասերից, և այն կարեղ է օգտագործվել որպես լիովին առանձին ծառային կառուցվածք:

Այժմ կդիտարկենք օրինակ, որը ամբողջությամբ օգտագործում է գրդարանի հանարավորությունները.

#include <tree.h>

#include "parser.h"

#include <function\_tree.h>

#include <iostream>

#define L tree<int>::left

#define R tree<int>::right

int main()

{

tree<int> t;

typedef tree<int>::iterator I;

tree<int>::iterator i = t.insert(4, t.begin());

std::cout<<"\*i "<<\*i<<std::endl;

tree<int>::iterator p = t.insert(2, i, tree<int>::left);

I j = t.insert(12, i, tree<int>::right);

i = p;

t.insert(1, i, L);

t.insert(3, i, R);

i = t.insert(8, j, L);

j = t.insert(14, j, R);

I m = t.insert(6, i, L);

I k = t.insert(10, i, R);

t.insert(5, m, L);

t.insert(7, m, R);

t.insert(9, k, L);

t.insert(11, k, R);

t.insert(13, j);

j = t.insert(16, j, R);

t.insert(15, j, L);

t.insert(17, j, R);

for (tree<int>::iterator k = t.begin(); k != t.end(); ++k) {

std::cout<<\*k<<std::endl;

}

t.convert\_to\_levels();

tree<int>::iterator c = t.begin();

std::cout<<\*c<<std::endl;

std::cout<<\*++c<<std::endl;

std::cout<<\*++c<<std::endl;

std::cout<<\*++c<<std::endl;

std::cout<<\*++c<<std::endl;

std::cout<<\*++c<<std::endl;

std::cout<<\*++c<<std::endl;

std::cout<<\*++c<<std::endl;

std::cout<<\*++c<<std::endl;

std::cout<<\*++c<<std::endl;

std::cout<<\*++c<<std::endl;

std::cout<<\*++c<<std::endl;

std::cout<<\*++c<<std::endl;

for (tree<int>::iterator k = t.begin(); k != t.end(); ++k) {

std::cout<<\*k<<std::endl;

}

std::string str;

std::cin>>str;

parser::get\_instance()->add\_variable('x');

std::cout<<parser::get\_instance()->to\_postfix(str)<<std::endl;;

function\_tree ft(parser::parse(str));

for (function\_tree::iterator k = ft.begin(); k != ft.end(); ++k) {

(\*k)->print();

}

Function\_tree diff\_tree = ft.diff();

std::cout<<"aaaa"<<std::endl;

for (function\_tree::iterator k = ft.begin(); k != ft.end(); ++k) {

(\*k)->print();

}

diff\_tree.set\_var\_value(‘x’, 5);

std::cout<<diff\_tree.count\_value()<<std::endl;

return 0;

}

ԳԼՈՒԽ 5

**Փորձարարական տեխնիկա**

ՓՈՐՁԱՐԱՐԱԿԱՆ ՏԵԽՆԻԿԱ

Նախագծման համար օգտագործված մեքենայի տեխնիկական տվյալները՝

* Intel(R) Core(TM) i7-3540M CPU @ 3.00ԳՀց 3.00 ԳՀց
* 4.00 ԳԲ (3.89 ԳԲ օգտագործմամբ) Օպերատիվ հիշողություն
* Windows 7 Enterprise 64-բիթ օպերացիոն համակարգ

Տողային տեսքի հետ աշխատելու համար օգտագործվել է QT 5.4.1 գրդարանի core ենթագրադարանը:

Գրադարանը մշակվել է C++ ծրագարվորման լեզվով:

ԳԼՈՒԽ 6

**Փորձնական հետազոտություններ**

ՓՈՐՁՆԱԿԱՆ ՀԵՏԱԶՈՏՈՒԹՅՈՒՆՆԵՐ

Մշակված գրադարանի հիմման վրա հետազետվել է, կիրառված ածանցման ալգերիթմի արագագործության, արտահայտության բարդությունից կախվածությունը:

Արտահայտության բարդություն ասելով կհասկանանք ներդրված ֆունկցիաների քանկը: Այսպիսով ներկայացված գրաֆիկում երևում է, որ ներդրված ֆունկցիաների քանակից կախված ածանցման գործողությունը բավականին ծախսատար է և բնութագրվում է քառակուսային բանաձևով(O(n2)):

Նկ. 6 Հետազոտության գրաֆիկ 1

Նկ. 6.1 Հետազոտության գրաֆիկ 2

ԳԼՈՒԽ 7

**Եզրակացություն**

Եզրակացություն

1. Աշխատանքում հետազոտվել են այժմ գոյություն ունեցող ծրագրերը և գրադարանները, որոնք ածանցում են հանրահաշվական արտահայտությունները:
2. Հետազոտվել են այժմ գոյություն ունեցեղ ալգորիթմեր, որոնք օգտագործվում ծառային կառուցվածքների վրա ձևափոխություններ անելու համար:
3. Հետազոտված ալգորիթմների հիմման վրա մշակվել է ալգորիթմ, որը ածանցում է տողային տեսքով մուտքին եկած արտահայտությունը:
4. Մշակված ալոգրիթմը հետազոտվել է տարբեր աստիճանի բարդություն ունեցող արտահայտությունների ածանցման համար:
5. Մշակված ալգորիթմի հիմման վրա նախագծվել է ծրագարվորման դինամիկ գրադարան

Այսպիսով՝ կատարվել են խնդրի դրվածքում առաջադրված պահանջները:

ԳՐԱԿԱՆՈՒԹՅԱՆ ՑԱՆԿ

1. Robin J. Hogan. 2014. Fast reverse-mode automatic differentiation using expression templates in C++. ACM Trans. Math. Softw. 40, 4, Article 26 (June 2014), 16 pages.
2. <http://dx.doi.org/10.1145/2560359>
3. Walther und A. Griewank: Getting started with ADOL-C. In U. Naumann und O. Schenk, Combinatorial Scientific Computing, Chapman-Hall CRC Computational Science, pp. 181-202 (2012).
4. <https://projects.coin-or.org/ADOL-C/wiki>
5. <http://www.codeproject.com/Articles/15432/Fast-Automatic-Differentiation-in-C>
6. <http://www.mathworks.com/help/matlab/index.html>
7. Donald E. K. The Art of Computer Programming -2005, -Vol1, -P. 320-346.
8. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill- 2009 [1990]. -P. 287-355
9. <http://en.wikipedia.org/wiki/Polish_notation>