پیاده‌سازی ساختار داده‌ها د ر C++

مهندس محمدباقر دستغیب
عضو هیات علمی گروه پژوهشی زبان‌شناسی رایانه‌ای

مرکز منطقه‌ای اطلاع رسانی علوم و فناوری
تابستان ۱۳۹۰

به نام خدا

نظر به اهمیت نقش اطلاع رسانی در زمینه‌های مختلف علوم و تکنولوژی، به منظور پاسخ‌گویی به نیازهای محققان و کارشناسان، و ارائه تازه‌ترین اطلاعات علمی وفنی در کوتاه‌ترین زمان، **مرکز منطقه‌ای اطلاع رسانی علوم وفناوری** براساس تفاهم‌نامه منعقده میان *وزارت علوم، تحقیقات و فناوری و فرهنگستان علوم جهان سوم* در سال ۱۳۷۰ دایر و لایحه تاسیس آن در سال ۱۳۷۵ از تصویب مجلس شورای اسلامی و شورای نگهبان گذشت. این مرکز در چهارچوب ضوابط و مقررات آموزش عالی جمهوری اسلامی ایران فعالیت می نماید.

مرکز منطقه‌ای اطلاع رسانی علوم و فناوری از طریق توزیع آخرین اطلاعات علمی و فنی و نیز کمک به تامین منابع علمی مورد نیاز دانشگاهها، سازمان‌ها و نهادهای تحقیقاتی و متخصصان داخلی و منطقه‌ای برای ارتقاء سطح علمی جمهوری اسلامی ایران و سایر کشورهای منطقه فعالیت می‌نماید.

کتاب «*پیاده‌سازی ساختار داده‌ها در C++*» در راستای تحقق اهداف فوق تهیه شده و مرکز منطقه‌ای اطلاع‌رسانی علوم و فناوری افتخار دارد که سی‌وهشتمین اثر خود را که حاصل تلاش فکر محققان و کارشناسان این سازمان است به زیور طبع بیاراید و تقدیم جامعه علمی و فنی بنماید.

امید است اقداماتی از این قبیل موجبات رضایت هرچه بیشتر جامعه علمی و فنی ایران را فراهم آورده و در راستای هدف ما که نشر و اشاعه اطلاعات علمی و فنی است موثر واقع گردد.

دکتر جعفر مهراد

رئیس مرکز منطقه‌ای اطلاع رسانی علوم و فناوری

سرپرست پایگاه استنادی علوم جهان اسلام (ISC)

فهرست مطالب

فصل اول

تعاریف

۱-۱- تعاريف

۱-۱-۱- كپسوله سازي داده (پنهان سازي داده)[1] عبارت از پنهان كردن جزئيات پياده سازي و داده‌ها (اطلاعات) از محيط بيرون شيء است.

۱-۱-۲- داده‌هاي انتزاعي[2]: جدا سازي توصيف داده‌ها از پياده سازي آنها، انتزاعي كردن داده‌ها است. به عبارت ديگر، در اين روش تعريف داده‌ها جدا از جزئيات پياده‌سازي، تعريف مي‌شود.

۱-۱-۳- تايپ داده: مجموعة داده‌ها (اشياء) و عمگرهاي مجاز براي آنها، تايب داده ناميده مي‌شود. (خواص و متدهاي قابل اجرا)

۱-۱-٤- تايپ داده انتزاعي **(ADT)**[3]: يك تايپ داده است، كه داده‌هايش به شكل انتزاعي تعريف شده‌اند. به عبارت ديگر، تعاريف داده‌ها و پياده سازي از يكديگر جدا شده‌اند.

از اين پس، ابزار پياده سازي تايپهاي انتزاعي، C++ انتخاب مي‌گردد. از توابع چند ريختي[4]، سربارگذاري عملگرها، تايپ قالب[5] و ارث بري براي پياده سازي كلاس‌هاي مختلف استفاده مي‌شود.

۱-۲- مقدمه‌اي بر C++

در اين فصل مقدمه‌اي بر C++ و روش تعريف توابع گفته شده است، به خواننده توصيه مي‌گردد، براي آشنايي بيشتر به كتابهاي آموزشي C++ مراجعه نمايد.

۱-۱-۲- تعريف داده‌ها: الف: براي تعريف مجموعه‌ها از كلمة كليدي enum استفاده مي‌شود، براي مثال مجموعة منطقي (Boolean) در زير تعريف شده است.

```
enum BOOL {FALSE,TRUE};
```

ب: تعريف متغيرها (اشياء): روش كلي تعريف بصورت زير است:

```
DataType    VarName(Constructor Parameters), ….;
```

مثال ۱-۱-۲: تعريف اشياء

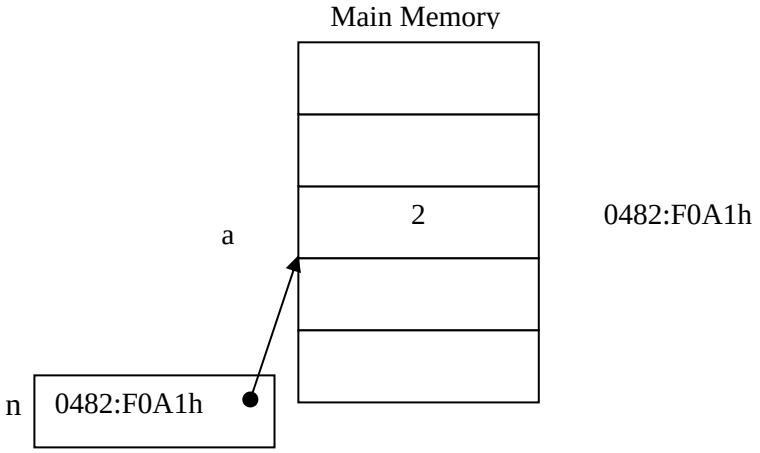
```
int y=2;
int a=2,b=7;
int *n=&a;//(آدرس)نگهداري مرجع
int &j=a;
```

j نام ديگر a است.

---

1 Information Hiding
2 Data Abstraction
3 Abstract Data Type
4 Polymorphism
5 Template

در زير نمودار يك اشاره گر، كه آدرس متغير ديگري را نگهداري مي‌كند نشان داده شده است:



۲-۲-۱- ورودي و خروجي: عملگر جريان ورودي << و جريان خروجـي >> اسـت، كه براي هر شيء، بايد (براي هر كلاس) تعريف شود.

cin >> Var_name;//خواندن از ورودي

cout << Var_name; // نوشتن در خروجي

۳-۲-۱- توابع در **C++**: توابع بـا شـكل ظـاهري اش (نـام و پارامترهـايش) شناسـايي مي‌گردد، بنابراين توابع مي‌تواند به شكل‌هاي گوناگون وجود داشته باشد، به عبارت ديگر وجه تمايز توابع با نام يكسان، پارامترهاي تابع است.

مثال ۲-۲-۱- تابع مينيمم و نشان دادن توابع چند ريختي

```
int min(int a, int b)
{    if(a<b)
          return a;
     else
          return b;
}
1- void f(int a)
     {    cout<< a; }
2- void f (int a, int b)
     {
          cout << a+b;
     }
3- void f(float a)
     { cout << a; }
     void main( )
```

```
{
    int a=1,b=2;
    float c=1.2;
    f(a);// Calling 1
    f(a,b); // Calling 2
    f(c);//Calling 3
}
```

پارامتر توابع مي‌تواند به دو صورت تعريف شود:

۱- پارامترهاي مقدار **(Value Parameters)**

در اين حالت يك كپي، از شيء به تابع داده مي‌شود پس تغيير شيء در تابع، پارامتر واقعي متأثر نمي‌گردد. براي مثال توابع تعريف شده در مثال ۲-۲ را نگاه كنيد. در اين حالت سازندة كپي فراخواني شده و از شيء كپي مي‌گيرد. براي شيء مورد نظر، بايد سازندة كپي تعريف شده باشد.

۲- پارامترهاي مرجع **(Reference Parameters)**

در اين حالت، آدرس متغير به تابع فرستاده مي‌شود، اگر پارامتر ظاهري تغيير كند، پارامتر واقعي هم تغيير خواهد كرد.

- پارامترهاي مرجع با اشاره گرها و يا متغيرهاي مرجع تعريف مي‌شود. به مثال ۲-۳-۱ توجه كنيد.

مثال ۳-۲-۱-پارامترهاي مقدار و مرجع

```
void f1(int a)
{ a=1; }
void f2(int * a)
{ *a=1; }
void f3(int &a)
{ a=1; }
void main( )
{
    int b=7;
    f1(b);
    cout << b;// b is 7
    f2( &b);
    cout<<b;// b is 1
    b=8;// b is 8
    f3(b);
    cout << b;// b is 1
}
```

٤-٢-١- تايپ قالب: نوع تايب قالب، در زمان استفاده، مشخص مي‌شود و در زمان تعريف، نوع آن مشخص نمي‌گردد.

با كمك تايپ داده قالب، داده‌ها به صورت انتزاعي تعريف مي‌شود.

مثال ٤-٢-١-تعريف تايپ قالب

```
template <class T>
void swap( T&a , T&b)
{
    T temp=a;
    a=b;
    b=temp;
}
void main( )
{
    int a=1,b=2;
    float k=7.2, j=6.7;
    swap(a,b);//template is int here
    swap(k,j);//template is float here
}
```

٥-٢-١- تابع بازگشتي: تابعي است، كه در تعريف اش، خودش را فراخواني مي‌كند.

در زمان پياده سازي تابع بازگشتي به اين نكات توجه كنيد:

١- در ابتداي تابع شرط خروج را كنترل كنيد.

٢- با توجه به تعريف بازگشتي، مجدداً، تابع را فراخواني شود.

تابع بازگشتي در هر بار فراخواني، محتواي متغيرهاي محلي، آدرس دستور بعدي و پارامترها را درون ساختاري به نام پشته[٦] قرار مي‌دهد، و پس از بازگشت، عنصر بالاي پشته بر مي‌دارد. (به قسمت پشته مراجعه شود) بنابراين معمولاً مدل بازگشتي از غير بازگشتي هزينه بيشتري دارد.

مثال ٢-٥-١-تابع بازگشتي فاكتوريل

```
long fact(unsigned int i)
{
    if( i = = 0 || i = = 1)//شرط خروج
        return 1;
    else
        return i*fact(i-1);//شرط بازگشتي
}
```

---

6 Stack

٦-٢-١-تعريف شيء (تايپ داده)در C++

```
class NEWTYPE
{
```

public: متغيرها و عملگرهاي عمومي اين قسمت توسط توابع غير عضو در دسترس است.

private: اين قسمت خصوصي است، فقط توسط توابع عضو كلاس و دوست كلاس و عضو كلاس دوست در دسترس است.

protected: مانند قسمت خصوصي ولي به ارث نيز مي‌رسد

```
};
```

مثال ٦-٢-١- تعريف تايپ دادة نقطه

```
class point
{    public:
        point(int x1=0,int y1=0)//سازندة كلاس
        {setxy(x1,y1);}
        ~point( ){ }
        void setxy(int x1,int y1){x=x1; y=y1;}
        int getx( ) {return x;}
        int gety( ) {return y;}
    protected: int x,y; };
```

٣-١- پيچيدگي زماني الگوريتم‌ها **Time Complexity**

زمان T(p)، براي برنامة p مجموع زمان كامپايل و اجرا است.

پيچيدگي زماني به پارامترهاي زيادي بستگي دارد، براي بدست آوردن معياري براي مقايسة پيچيدگي زماني الگوريتم‌ها بدون اجرا، زمان اجراي دستورات را مساوي فرض مي‌كنيم. و تعداد دستورات را مي‌شماريم. آنرا با نماد O (بيگ او بخوانيد) نشان مي‌دهيم.

اين پارامتر نشان ميدهد بر اساس پارامتر تاثير گذار در مساله (n)، تعداد تكرار چگونه تغيير ميكند؟ (مثلاً تغييرات خطي، غير خطي، يا بدون تاثير به پارامتر ورودي است).

هر ترتيب از دستورات غير وابسته به پارامتر ورودي طبق فرض، داراي پيچيدگي O(1) است.

مثال : O(1)

```
void f (int n)
{
    cout << n;
}
```

تعريف : حلقه‌هاي تكرار (با n بار تكرار) داراي پيچيدگي O(n) هستند.

مثال O(n)

```
void f( int n )
{
    for(int i=0;i<n;i++)
        cout << i;
}
```

در محاسبات بزرگترين حالت پيچيدگي را در نظر مي‌گيريم

به عنوان مثال

$O(1)+O(n) = O(n)$

جدول۱-۳-۱- پيچيدگي بر اساس فاكتور **n** (به صورت صعودي براساس رشد زمان)

O(1)
O(log n)
O(n)
O(n log n)
$O(n^2)$
$O(n^3)$

مثال : $O(n^2)$

```
for(int i=0;i<n;i++)
    for(int j=0;j<n;j++){ ….O(1)…. }
```

فصل دوم

آرایه ها

تعريف آرايه: آرايه تعدادي خانة هم نوع است، كه هر خانه، توسط يـك انـديس عـددي منحصر به فرد مشخص مي‌شود.

مثال :آرايه A

int A[MAX];

Int	Int	Int	Int	Int	…	Int	Int
0	1	2	3	4			MAX-1

توجه: در C++، نام آرايه، آدرس خانة اول آرايـه اسـت. در C++، مـي‌تـوان بـا كمـك اشاره‌گرها و تخصيص حافظة پويا، آرايه‌هاي پوياايجاد كرد كه تعداد عناصرش، پويا باشد.

مثال:

```
int *a;
a=new int[Array_Length];
assert(a!=0);
استفاده از آرایه//
delete [] a;
```

۱-۲-پياده سازي ساختار مجموعه با استفاده از آرايه

مي‌خواهيم يك مجموعه را در يك آرايه ذخيره كنيم.

مجموعه داراي عملگرهاي زير است:

۱- درج در مجموعه

۲- حذف از مجموعه

۳- عضو بودن، و بدست آوردن دادة عنصر عضو

٤- آيا مجموعه خالي از عضو است؟ (مجموعه تهي)

```
template < class T >
class ArrayEnum
{
    public:
        ArrayEnum(int maxsize=100);
        ~ ArrayEnum( ){delete [] Array;}
        void Insert(T mydata);
        void Delete(T mydata);
        bool IsInEnum(T sample);
        bool IsEmpty( );
    private:
        void AddSpace(unsigned=1);

    protected:
        T *Array;//آراية پويا
        unsigned max;
```

```
        int end;//آخرین عضو درج شده
};

template<class T>
ArrayEnum<T> :: ArrayEnum(int maxsize)
{
    max=maxsize;
    Array = new T[max];
    end=-1;
    assert(Array != 0);
}
template<class T>
void ArrayEnum<T> :: Insert(T mydata)
{
    if(IsInEnum(mydata))//موجود است
        return;
    if(end+1 == max-1)
        AddNewSpace(1);//اضافه کردن خانه به آرایة پویا
    Array[++end]=mydata;
}
template<class T>
void ArrayEnum<T> :: IsEmpty( )
{   if(end == -1)
        return TRUE;
    else
        return FALSE;
}
template<class T>
bool ArrayEnum<T> :: IsInEnum(T mydata)
//عملگر تساوي == در کلاس قالب باید تعریف شود
{
    if( IsEmpty( ) ) return FALSE;
    for(int i=0;i<=end;i++)
        if(Array[i] == mydata)
            return TRUE;
    return FALSE;
}
template<class T>
void ArrayEnum<T> :: Delete(T mydata)
{   if(!IsInEnum(mydata)) return ;
    for(int i=0;i<=end;i++)
        if(Array[i]==mydata)  break;
    for(int j=i;j<end;j++)//Shift To Delete
        Array[i]=array[i+1];
    end--;
}
```

```
template<class T>
void ArrayEnum<T> :: AddSpace(unsigned n)
{
    if(n==0) retuen;
    T *temp= new T[n+max];
    assert(temp);
        max+=n;
        for(int i=0;i<=end;i++)
         temp[i]=Array[i];
        delete [] Array;
        Array=temp;}
```

دراين پياده سازي، بخصوص اگر ترتيب عناصر نيز مهم باشد، (مجموعه مرتب) احتياج به شيفت دادن عناصر است.

بعداً ساختارهايي مطرح خواهد شد كه هزينة كمتري را درجستجو و درج خواهد داشت.

دراين روش، درج، حذف، جستجو از درجة O(n) است. (n تعداد عناصر موجود در مجموعه است)

۲-۲- پياده سازي چند جمله‌اي‌ها با آرايه

چند جمله‌اي درجه n از نظر رياضي به صورت زير تعريف مي‌شود و توان جملات اعداد صحيح مثبت هستند، و ضرايب اعداد حقيقي هستند.

$f(x) = a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_0$

x نيز عدد حقيقي است، و كلاً n+1 جمله موجود است.

روش‌هاي پيشنهادي براي ذخيره سازي چند جمله‌اي در حافظه

۱- براي هر ضريب يك خانة آرايه در نظر گرفته شود، بنابراين يك آرايـة n+1 عضوي مورد نياز است، انديس خانه‌ها توان را نشان مي‌دهد و محل ذخيره ساري هر ضريب مشخص است.

عيب‌اين روش، اتلاف فضاي حافظه است، زيرا براي ضرايب صفر هم محلي در نظر گرفته شده است.

مثال : ذخيره كردن $f(x) = 4.2x^{12}+2x+1$

1.0	2	0	0	0	0	0	0	0	0	0	0	4.2
0	1	2	3	4	5	6	7	8	9	10	11	12

۲- دراين روش از آرايه‌اي مشترك براي چندين، چند جمله‌اي استفاده مي‌شود. دراين روش فضاي حافظة كمتري مصرف مي‌شود ولي در زمان درج و حذف ضرايب نياز به شيفت

دادن است.

در روش قبل درج و حذف با پيچيدگي O(1) انجـام مـي‌شـود ولـي درايـن روش O(n) است. (n تعداد جمـله‌هاي چند جمله‌اي است)

٣- فقط جمله‌هايي با ضرايب غير صفر را ذخيره كنيم و هر چند جملـه‌اي در يـك آرايـه ذخيره گردد.

تعريف ADT براي چندجمله‌اي

```
class polynomial
{
    friend ostream &operator<<(ostream &,const polynomial &);
    friend istream &operator>>(istream &,polynomial &);
    public:
        polynomial(unsigned Degree);
        ~polynomial( );
        polynomial( const polynomial &);// سازندة كپي
        unsigned Degree( );//درجة چند جمله‌اي
        float coefficient(unsigned n);//nضريب جمله با توان
        polynomial operator+(const polynomial &);//جمع دوچندجمله‌اي
        polynomial operator-(const polynomial &);//تفريق دوچندجمله‌اي
        polynomial operator*(const polynomial &);//ضرب دوچندجمله‌اي
        polynomial operator/(const polynomial &);//تقسيم دوچندجمله‌اي
        polynomial operator%(const polynomial &);
                //تقسيم دوچندجمله‌اي باقي مانده
                float operator( )(float x);//x محاسبة چند جمله‌اي در نقطة
        void savenode(unsigned e,float c);//ذخيرة هر جمله
    private:
    //پياده سازي بروش دلخواه
};
```

مثال: پياده سازي بروش دوم (ذخيرة ضرايب غير صفر)

پياده سازي ساختار هر نود چند جمله‌اي

```
class term
{   friend class polynomial;
    public:
        term( int e=0,float c=0.0 );
        ~term( ){ }
    private:
        float coef;
        int exp;
};
```

```
term :: term( int e, float c )
{
      coef=c;
      exp=e;
}
```

پیاده سازي قسمت Private

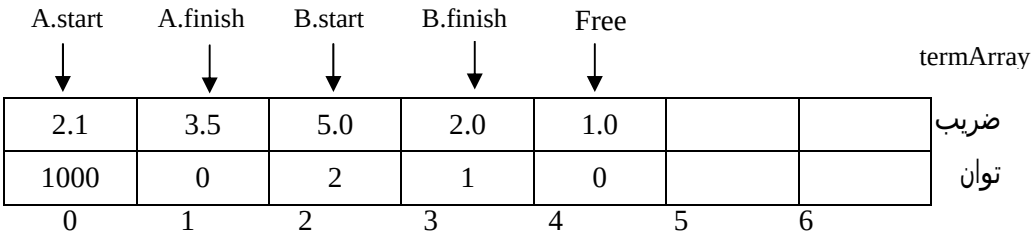
```
static term termArray[MAX];
static int free;
int start,finish;
```

مقدار اولية free

```
int polynomial :: free=0;
polynomial :: polynomial( )
{
      start=0;
      finish=-1;
}
```

مثالي از نماي حافظه در پياده سازي

$A(x) = 2.1x^{1000} + 3.5$ $B(x) = 5x^2 + 2x + 1$



جمع دو چند جمله‌اي دراين روش:

```
polynomial polynomial :: operator+(const polynomial &p)
{
      // (*this) + p
      polynomial c(max(this->Degree( ) , p.Degree( ));
      int a=strat, b=p.start,c.start=free;
      while( a<=finish && b<=p.finish )
      {
            if(termArray[a].exp == termArray[b].exp)
            {
                  c.savenode(termArray[a].exp,termArray[a].coef+
                                    termArray[b].coef);
                  a++;
                  b++;
            }
            else if( termArray[a].exp < termArray[b].exp )
```

```
            {
                c.savenode(termArray[b].exp,termArray[b].coef);
                b++;
            }
            else
            {
                c.savenode(termArray[a].exp,termArray[a].coef);
                a++;
            }

        }
        for( ; a<=finish;a++)
            c.savenode(termArray[a].exp,termArray[a].coef);
        for( ; b<=p.finish;b++)
            c.savenode(termArray[b].exp,termArray[b].coef);
           c.finish=free-1;
           return c; // سازندة كپي فعال مي‌گردد
}
void polynomial :: savenode(unsigned e,float c)
{//فقط ضرايب غير صفر ذخيره مي‌شود
    if(c==0.0) return;
    int a=this->start;
    int b=this->finish;
    for(int i=1;i<=b;i++)
        if(termArray[i].exp == e)
            break;
    if(i<=b)//توان پيدا شده است
    {
        if(coef+termArray[i].coef == 0)
        {
            for(int j=i;j<b;j++)
                termArray[j]=termArray[j+1];
        }
        else
            termArray[i].coef+=c;
    }
    else
    {
        //جابجايي عناصر براي اضافه كردن فضا
        b=shift(start,finish);
        termArray[b].exp=e;
        termArray[b].coef=c;
    }
}
```

جمع دو چند جمله‌اي اگر يکـي n جملـه و ديگـري m جملـه داشـته باشـد، از درجـة (O(n)) O(m+n) است. يکي از مشکلات‌اين روش آنست که با تغيير Start و Finish يکي بقيه نيز بايد تغيير کنند.

مشکل‌اين پياده سازي با در نظر گرفتن آرايه‌هاي جداگانه براي هر چندجملـه‌اي برطـرف مي‌شود.

```
private:
    term *termArray;
    unsigned max;
    int end;
```

تعريف خانة آزاد[1]: خانه‌اي از آرايه است، که قسمت ضـريب آن صـفر اسـت. زيـرا ضـرايب غيرصفر را ذخيره مي‌کنيم.

بنابراين عدد صفر در قسمت ضريب نشان دهندة آن است، که خانه مورد نظـر اسـتفاده نـشده است.

```
polynomial :: polynomial( )
{
    max=10;
    end=0;
    termArray=new term[max];
    assert(termArray != 0);
    //سازندة کلاس تمام خانه‌ها را مقداردهي اوليه مي‌کند
}
```

پياده سازي ضرب دو چندجمله‌اي

```
polynomial polynomial :: operator*(const polynomial & p)
{
    polynomial temp;
    for(int i=0;i<end;i++)
        for(int j=0;j<p.end;j++)
        {
            int e=termArray[i].exp+p.termArray[j].exp;
            float c=termArray[i].coef*p.termArray[j].coef;
            temp.savenode(e,c);
        }
        return c;
}

void polynomial :: savenode(unsigned e,float c)
{
    for(int i=0;i<end;i++)
        if(termArray[i].exp == e)    break;
```

1 Free Home

```
    if(i==end)
    {
        termArray[end].coef=c;
        termArray[end].exp=e;
    }
    else termArray[i].coef+=c; }
polynomial polynomial :: operator/(const polynomial &p)
```

دراين تابع پس از تقسيم this بر p، this باقي مانده مي‌شود و خارج قسمت را برمي گرداند.

```
{
    polynomial *temp=new polynomial( );
    assert(termp !=0);
        if(degree( ) <p.degree( ))
         return *temp;
        bool doloop=TRUE;
    while(doloop)
    {
        int i=this->degree( );
        int j=p.degree( );
        float f=coefficient(i);
        float k=p.coefficient(j);
        temp->savenode(-f/k,i-j);
        polynomial rem=(*this) * (*temp); //Multiply this polynomial to
polynomial pointed by temp
        *this=(*this)+rem;
        id(this->degree( ) < p.degree( ))
            doloop=FALSE;
    }
    return *temp;
}
```

۲-۲- ماتريس پراكنده **Sparse matrices**

تعريف: ماتريسي است كه اغلب درايه هاي آن يكسان باشد (براي مثال صفر باشد) اين ماتريس، ماتريس پراكنده (خلوت) ناميده مي‌شود.

مثالي ازكاربردهاي ماتريس پراكنده:

مثال۱ : صفحه Desktop رادرسيستم عامل windows 98 درنظربگيريد، اگربراي هر نقطه عدد رنگ آنرا ذخيره كنيد، متوجه خواهيد شدكه اكثرنقاط داراي رنگ يكسان هستند، بنابراين ماتريس بوجود آمده يك ماتريس پراكنده است .

مثال ۲ : براي پياده سازي ارتباطات ميان يك شبكه بزرگ (Graph) مي‌توان ازيك ماتريس استفاده كرد كه ارتباطات را پياده سازي كند،اين ماتريس يك ماتريس پراكنده است زيرا ممكن است هر نود به تمام نودهاي مجود در شبكه، ارتباط نداشته باشد.

براي بهينه سازي مصرف حافظه، عناصر غير يكسان (غير صفر) را ذخيره مي‌كنيم.

مثال ۳ : ماتريس پراكنده

$$A=\begin{bmatrix} 5 & 5 & 5 & 5 & 5 \\ 5 & 5 & 5 & 5 & 5 \\ 5 & 1 & 2.7 & 5 & 5 \\ -1.2 & 5 & 5 & 5 & 5 \end{bmatrix}_{4\times5} \qquad B=\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 2 & 0 \end{bmatrix}_{4\times4}$$

حال فرض كنيديك ماتريس ۲۵۰۰۰=۵۰۰۰*۵۰۰۰ رامي خواهيم ذخيره كنيم، اگر هر خانه عدد صحيح (int) باشد، بنابراين فضاي لازم byte ۵۰۰۰۰۰۰۰=۲*۲۵۰۰۰۰۰۰خواهد بود.

حال آنكه ممكن است ۱۰/۰۰۰ مورد، عناصر غير يكسان موجود باشد، بنابراين مي‌توان ۲۰/۰۰۰=۲*۱۰/۰۰۰ byte حافظه را استفاده كرد فضاي زيادي نيز صرفه جويي شده است.

مثال٤ : يك ماتريس پراكنده ونمونه Sparse پراكنده آن.

ROW COL VAL

$$SPARSE(A)=\begin{bmatrix} 1 & 2 & 1 \\ 2 & 2 & 5 \\ 3 & 3 & 7 \end{bmatrix} \qquad A=\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

دراين حالت عنصر غالب (MOD) صفر است ، و حجم ماتريس اصلي ۸۰ بايت و ماتريس پراكنده ۱۸ بايت است. در مثال فوق Row نشان‌دهنده سطر، Col نمايانگر ستون و VAL نمايانگر مقدار ماتريس است.

پياده سازي ماتريس پراكنده:

```
class  matrixterm
{     friend     class          sparsematrix ;
      public :
           matrixterm ( )     { row = col = -1; }
       ~matrixterim( ){ }
private :
          int    row , col;
          float     value;
};
```

با در نظر گرفتن نوع float براي value هر نوع عدد مثبت ، منفي يا اعشاري قابل ذخيره كردن است.

هر خانه براي ذخيره يك درايه ماتريس بصورت شكل زير است.

Row	Col	Value

ادامه پیاده سازي ماتریس پراکنده:

```
class     sparsematrix
{
    friend     ostream &  operator << (   ostream  & , const   sparsematrix & );
   friend     istream  &  operator >> ( istream & , sparsematrix & );

   public:
      sparsematrix(int   m=5 , int   n=5 );                    // پیش فرض ماتریس 5 *5  است
      ~sparsematrix( );
      float operator ( )(int , int);                    بدست آوردن درایه  ( i , j)

void     saveterm ( int , int  , float );                          ذخیره درایة  ( i , j )
         sparsematrix   operator  + ( const       sparsematrix & ) ;        جمع دو ماتریس
        sparsematrix   operator   * (const       sparsematrix  & ) ;        ضرب دو ماتریس
                  :
                  :
private:
           termmatrix     * Array;
           unsigned          maxterm;
           int     m , n;
           float               midterm;
 };

sparsematrix::sparsematrix ( int  m , int   n , float    mt)
{       maxterm = 10;
        this->m = (m > 0 ? m : 5 ) ;
        this ->n = (n > 0  ? n : 5 ) ;
        modterm = mt;
        Array  = new     termmatrix [maxterm];
        assert ( Array ! = 0 );
}
sparsematrix :: ~ sparsematrix ( )
{   delete [ ]  Array; }

float  sparsematrix :: operator( )(int   i, int  j )
{       if (i > =0 &&  i<m &&  j>=0&&  j<n)
        {for (int   k=0;k<maxterm;k+ + )
                if (Array [k].row = =i  && Array[k].col = = j)
                                return   Array [k].value;
      return  modterm;
        }
        else
        {   cerr << `` out of range ``;
            return   0.0;
         }
}
```

اين تابع يك تابع دسترسي براي درايه هاي ماتريس است . اگـر سـطر و سـتون در آرايـه عناصر، موجود باشد، يعني اين عنصر، غير غالب (صفر) بوده اسـت، بنـابراين ذخيـره شـده و مقدارش رابر مي‌گرداند، و اگر ذخيره نشده باشد، پس حتماًعنصر غالب (صفر) بوده است.

براي ذخيرة عناصر نيز احتياج به يـك set method اسـت كـه بـا گـرفتن انـديس سـطر وستون و مقدار درايه آنرا ذخيره كنداين تابع، اگـر مقـدار داده باشـد بـا modterm (عنـصر غالب) برابر نباشد، آنرا ذخيره مي‌كند. در ضمن اگر نياز باشد ميـزان حافظـه آرايـه را افـزايش مي‌دهد.

```
void  sparsematrix::saveterm(int  i,int  j ,float term )
 {      if (i>0 &&  i<m   &&  j>=0 &&   j<n  )
        if (term = = modterm )
        {  for (int   k=0;k<maxterm;k+ + )
        if(Array[k].row = =i&&Array[k].col = = j)  Array [k].row=Array[k].row=-1;
           return;//don't  save
        }
       else
       {   int empty = 0;
           for (int   k=0;k<maxterm;k++)
         {        if (Array [empty] .col ! = -1)
                          empty + +;
                  if (Array [k].col = = j  && Array [k] .row = = i)
                  {          Array [k] .value = term;
                           return;
                 }
    }
        if ( empty = =max term )                          //   خانه آزاد نبوده است
   {     term   matrix   *  temp = new  termmatrix [maxterm + 10];
         maxterm + = 1 0;
         for (int  k= 0;k <max term –1 0;k + + )
             temp [k] = Array [k];
        delete  [ ] Array;
       Array = temp;
    }
    Array [empty] . row = i;
    Array [empty]. Col = j;
    Array [empty] . value = term;
     Return;
   }// else
   else
   cerr <<`` out  of   range ``;
}

sparsematrix  sparse matrix::operator +(const  sparsematrix  &b)
{        if ( m! = b.m | | n! = b.n )  return  * this;
         sparsematrix   *p = new   sparsematrix(m,n , modterm);
```

```
        for( int  i= 0;i<n;i+ +)
              for(int  j= 0;j<m;j++)
                    p->saveterm(i,j ,(* this)(i,j)+ b(i,j ));
        return  *p;
}
sparsematrix   sparsematrix::operator *(const   sparsematrix    & b)
{       if (n! = b.m )
        {    cerr << `` con`t  mui``;
             return     * this;
        }
        sparsematrix  *p = new   sparsematrix (m, b.n  ,0 );
        for (int  i=0;i<m;i ++)
                for(int  j=0;j<n;j++)
                        for (int   k=0;k< b.n;k++)
                              P->saveterm(i, k  ,(*this) (i,j) *  b(j,k));//O(n)

       return   *p;
}
```

پیچیدگي الگوریتم فوق از $O(m.n.k.MAX)$ یا $O(n^4)$ است.

```
ostream  & operator << ( ostream  & output , const  sparsematrix  &M)
{
        for(int   i=0;i<M.m;i++)
        {           for (int   j= 0; j< M.m;j+ +)
                                output << M(i , j );
                    out put  << `` \n ``;
         }
     return      output;
}

istream   & operator >>( istream  & input , sparsematrix  &n)
{           float      temp;
            for (int   i=0;i< n.m;i++)
               for (int  j=0;j< n.n; j++)
               {
                   cout << `` enter  for  row =`` << i << `` and  col = ``<<j;
                     in put >>temp;
                     n . saveterm (i,j  ,temp );
               }
        return     input;
}
```

تمرین ۱ : تابع det رابراي محاسبة دترمینان سربارگذاري کنید.

تمرین ۲ : تابع inv رابراي محاسبة وارون ماتریس سربارگذاري کنید.

فصل سوم

لیست پیوندی

تعريف: مجموعه‌اي است كه عناصرش بطور خطي پشت سرهم قرار گرفته‌اند. به عبارت ديگر ترتيبي از عناصر اســت، كه هر عنصربراي ايجاد دسـترسي، به عنصر همـسايه‌اش، به آن اشاره مي‌كند.

هرگاه بخواهيم ترتيبي از عناصر ايجاد كنيم و عمليات درج، حـذف و ويرايـــش را انجام دهيم مي‌توان اين عمل را توسط ليست پيوندي انجام داد .

به عنوان مثال اگر بخواهيم يك ليست از اسامي دانشجويان بـه ترتيـب ورود ايجـاد كنيم مي‌توان آنرا توسط ليست پيوندي پياده سازي كرد .

ليست پيوندي با آرايه و حافظة پويا پياده سازي مي‌شود، اين ساختار (Cursor) نـام دارد، ولي در پياده سازي با آرايه ترتيب فيزيكـي عناصـر نـشان دهنـدة ترتيـب منطقـي نيست، وهر عنصر قسمتي بنام NEXT (خانة بعدي) دارد كه آدرس خانة بعد يا انـديس خانة بعد را ذخيره مي‌كند.

ساختار هر عنصر بصورت روبرو است:

Data	Next

عملياتي كه در هر ليست پيوندي مجاز است عبارتنداز:

۱. درج عنصر در ابتدا
۲. درج عنصر در انتها
۳. درج عنصر بعداز عنصر k
٤. حذف عنصر دلخواه
۵. كنترل وضعيت ليست (خالي بودن ليست)

مثال ۱- مثالي از اسامي به ترتيب زير: (ali, ahmad, kamran, reza)

	داده	اندیس بعدي
	data	Next
0	``ahmad``	3
1	``reza``	-1
2	``ali``	0
3	``kamran``	1
4	0	0
5	:	

خانة آخر، خانه‌اي است كه قسمت Next آن ۱- است.

در پياده سازي با آرايه اندیس خانه بعد را ذخيره مي‌كنيم.

start

2

پياده سازي ليست پيوندي با آرايه پويا

```
class Arraylinkedlist;
class HomeArray
{  fried  class Arraylinkedlist;
  publish:
              HomeArray ( );
           ~ HomeArray ( );
  private:
         int Next ;
         char   data[10];
};
class Arraylinklist
   { friend  ostream &operator<<(ostream &, const  Arraylinkedlist & )
    public :
              Arraylinkedlist( );
            ~ Arraylinkedlist( );
              void  Insertlast(char * );
             void  Insert first(char *);
              void  InsertAfter(char * , char * );
              bool  search (char *);
              void  Delete(char *);
private :
          HomeArray  *Array;
          int start;
          unsigned      max;
}

Arraylinkedlist :: Arraylinkedlist( )
{    Max = 100;
     Array = new   HomeArray [max];
     assert (Array ! = 0);
      start = -1;
}

Arraylinkedlist :: ~ Arraylinkedlist( )
{
      delete  [ ]  Array;
}
```

```
HomeArray :: HomeArray( )
{    data[0] = `\0`;
       Next = -1; }

                                  HomeArray :: ~
HomeArray( )
  {
  }
```

چاپ كل ليست

با توجه به سازنده، ليست تهي، ليستي است كه start براي آن ١- باشد.

تابع اضافه كردن به ابتداي ليست: Insert first

الگوريتم:

الف: اگر ليست تهـي باشـد (start = = -1) ، آنگـاه كـافي اسـت start انـديس

خانه‌اي كه داده را در آن مي‌گذاريم، ذخيره كند.

ب: اگر ليست تهي نباشد، پس بايد مقدار Next يك خانه خالي را بـرا بـر start قرار دهيم، داده را درون آن خانه ذخيره كنيم، و start برابر انديس آن خانه قـرار گيـرد. اگر خانه آزاد موجود نباشد مي‌توان حافظه بيشتري تخصيص داد.

```
void Array linkedlist :: Insert first (char *d)
{    if (starlen (d) > 9 )   return;
     if ( start = = -1)
     {
          strcpy ( Array [0].data,d );
          start = 0;
     }
   else
   {  for (int  empty  =0; empty <max; empty + + )
          if (Array [empty ].data [0] = =0 )
                    break ;
          if (empty = =max ) //      خانة خالي موجود نبوده
          {   max + = 100;
              HomeArray  *temp = new   HomeArray [max];
              if(! temp )    return;
              for( int  i=0;i<max-100 ;i ++)
              {       temp [i] .Next = Array [i] . Next ;
                      strcpy (temp[i] .data , Array[i].data );
               }
              delete  [ ]  Array;
              Array  =  temp;
          }
          Array [empty ] .Next = start;
          start = empty;
          strcpy  (Array [empty].data , d );
   }
}
```

اضافه كردن به انتهاي ليست:

الف- اگر ليست تهي است، كافي است يك خانه آزاد در نظر بگيريم و داده را درون آن بگذاريم، سپس start انديس آن خانه را نگهداري كند.

ب- اگر ليست تهي نيست بايد خانه آخر ليست، (خانـه اي كـه قـسمت Next آن ۱- است و با شروع كردن از start به دست مي‌آيد) را بيابيم، سـپس يـك خانـه آزاد را پيدا كنيم، داده را درون آن بگذاريم، سپس قسمت Next خانه آخر برابـر انـديس خانـه آزاد قبلي قرار بگيرد .

```
void  Arraylinkedlist :: Insertlast (char  *p)
{       if ( start ! = -1)
            for (int  i = start ;Array[i].Next ! = -1 ; )
```

```
                    i= Array [i].Next;
        else
        {       start = 0;
                  strcpy (Array[0].data ,p);
                 Array [0].Next = -1;
               return;
        }
        for (int  empty = 0;empty <max;empty ++)
              if ( Array[empty ].data[0] = = Null)
                                  break;
        if ( empty = =max )
        {
          //              اضافه كردن تعداد خانه هاي آرايۀ پويا
         AddSpace(10);
        }
        strcpy (Array [empty ].data ,p);
        Array [empty ] .Next =-1;
        Array[i].Next = empty;
}

void  Arraylinkedlist :: InsertAfter(char  * p1 , char  *p2)
{       int   I= start;
    while (I ! = -1 )
    {
      if (!strcmp (Array [i],data , p1))
                  break;
        I = Array [i].Next;
     }
     if (I = = -1 )      return i   // not found
     for (int  empty = 0;empty <max;empty + +)
           if(Array[empty ].data [0] = =`\0`)
                        break;
      if (empty = =max)
      {
           //              اضافه كردن تعداد خانه هاي آرايۀ پويا
          AddSpace(10);
       }
       strcpy (Array [empty].data,p2);
       Array [empty].Next = Array[I].Next;
       Array [I].Next = empty;
}
```

عيب اين روش آنست كه براي پيدا كردن يك خانۀ آزاد بايد يك جستجوي خطي انجام دهيم و پيچيدگي از O(N) است . روش ديگر يك جستجوي حلقوي و ذخيره سازي انديس اولين نشانه آزاد بعدي است – تا در هر بار اضافه كردن زمان كمتري شود.

جستجو: براي جستجوي ليست، از ابتداي ليست شروع مي‌كنيم و تا زماني كـه داده پيدا نشده، حداكثر، تا انتهاي ليست جلو مي‌رويم و جـستجوي خطـي مـي‌كنـيم. حالـت ميانگين $O(\frac{n+1}{2})$ است يا O(n) است.

```
bool  Arraylinkedlist :: search (char  * p )
   {    int   i=start;
        while ( i  != -1 )
        {   if ( ! strcmp (Array[i].data ,p ))
                          break;
          i = Array[i].Next;
        }

        if (i = = -1)    return  false;
         else      return     true;
       }
```

سربار گذاري >> براي چاپ ليست:

اين تابع بايد به صورت دوست كلاس سربارگذاري گردد.

```
ostream  & operator<<(ostream   & output , const  Arraylinkedlist   &l)
 {              for (int    i= l.start; i! = -1;)
                {        output <<l.Array[i].data;
                           i= l.Array[i].Next;
                 }
                 return   output;

}
```

حذف ليست: براي حذف از ليست بايد سه حالت در نظر بگيريم :

۱- حذف از ابتدا: در اين صورت start بايد به عنصر دوم اشاره كند و خانه مورد نظر خالي گردد.

۲- حذف از انتها: در اين صورت عنصر ماقبل از آخر، بايد جديد شود، وبه عنصر بعدي اشاره كند.

۳- حذف عناصر مابين: در اين صورت عنصر ماقبل از عنصري كه حذف مي‌شود، بايد جديد شود، و به عنصر بعدي اشاره كند.

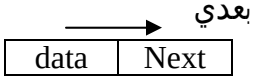
```
void   Arraylinkedlist :: delete ( char   *p)
 {        if ( start = = -1)  return;
          if (! Strcmp(Array [start] .data,p))       // عنصر اول
     {
            Array [start].data[0]=`\0`;
            Start = Array[start].Next;
            return;
```

```
        }
        else // is not the first node of list
       {
              int  i=Array [start].Next;
            int    j=start;
        while (i != -1 )
       {
          if (! strcmp(Array[i].data,p))
              break;
              j=i;
              i= Array[i].Next;
       }
       if (i = = -1 )       return;   // پيدا نشد
       Array[j].Next = Array [i] .Next;
       Array[i].data [0] =`\0`;
}
```

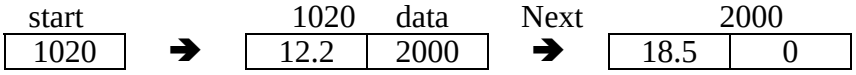
۳-۲ پياده سازي ليست پيوندي با حافظه پويا:

در اين روش بازاي هر نود، حافظه اختصاص مي‌يابد و خانه هاي حافظه بـراي هـر نود با ديگري تفاوت دارد و ممكن است پشت سر هم نيـز نباشـد، قـسمت Next يـك اشاره گر است و آدرس خانه بعدي را نگه مي‌دارد.

اشاره گر به خانه بعدي



مثال: نماي حافظه از يك ليست پيوندي با دو نود.



۳-۲-۱ پياده سازي كلاس قالب براي ليست پيوندي

الف- پياده سازي كلاس هر نود كلاس

```
class  linkedlist;
 template < class  datatype >
 class    listnode
 {  friend   class  linkedlist;
                 listnode (datatype );
                 listnode ( ) {  Next = 0;}
                ~listnode ( ) { }
     private:
                 listnode<datatype>     * Next;
                 datatype     data;
};
```

```
template <class  datatype>
listnode<datatype>::listnode(datatype  d )
   {          data = d ;          //    باید سر بار گذاري شده باشد  template در کلاس =
              Next = 0;
  }
```

ب- پياده سازي ليست پيوندي

```
template<class   datatype>
class  linkedlist
{  friend    ostream   & operator << (ostream  &,const  linkedlist <datatype
>&);
    publish :
              linkedlist ( );
             ~ linkedlist ( );
               void   Insertlast(datatype);
               void  Insertfirst (datatype);
               void  Delete ( datatype);
               bool  search (datatype );
             datatype  operator [ ](int);          //  ام  i  بدست آوردن ديتاي نود
              private:
                       listnode <datatype>  *start;
};
```

کلاس فوق ساختار تايپ دادة انتزاعي را براي ليست پيوندي نشان مي‌دهد.

```
template <class  datatype>
linkedlist <datatype > :: linkedlist ( )
{         start =0;
}
template < class  datatype>
linkedlist <datatype>:: ~ linkedlist ( ) //خذف حافظة كلية نودها
      {        listnode <datatype>   * i=start;
               while (i ! = Null )
               {       list node <datatype >   *j;
               j=i;
               i = i->Next;
                      delete  j;
            }
}
```

اضافه كردن به انتها:

الگوريتم: ابتدا انتهاي ليست را پيدا مي‌كنيم (از ابتداي ليست شروع مي‌كنيم، با كمك آدرسهاي حافظه جلو مي‌رويم، تا به انتهاي ليست برسيم يعني خانه‌اي كه next براي آن صفر است) سپس حافظه اختصاص مي‌دهيم، و next خانة آخر، آدرس خانـة جديـد را ذخيره مي‌كند.

```
template < class  datatype>
void  linkedlist <datatype > :: Insertlast(datatype  p)
```

```
{
      listnode <datatype >   *i = start , *j = new  listnode (p);
      assert (j!=0 );
      if (!i)  // list  is  empty
        start =j;
      else  // adding   to end  of   list
      {     while (i->Next)
               i = i ->Next;
       i->Next = j;
       }
}
```

پيچيدگي اين الگوريتم بدليل اينكه بايد انتهاي ليست را پيدا كند از درجة o(n) است.

اضافه كردن به ابتدا:

در اين حالت start هميشه عنصر اول را نشان مي‌دهد، و اضافه كردن بـا پيچيـدگي O(1) قابل انجام است.

```
template < class  datatype >
void    linkedlist <datatype> :: insertfirst (datatype   p)
{         listnode < datatype>   * i = new   listnode (p);
          assert(i ! =0);
          if ( ! start)
              start= i;
         else  //adding  to first  of  list
         {        i->Next = start;
                  start =i;
          }
}
```

حذف يك نود از ليست:

الف: عنصر اول: start به عنصر دوم اشاره مي‌كند و حافظة عنصر اول آزاد مي‌گردد. (پيچيدگي O(1))

ب: اگر عناصر مياني حذف شود - آنگاه ابتدا بايد محل آنرا پيـدا كنـيم (جـستجوي خطي) و سپس، عنصر ماقبل، قسمت Next به عنصر مابعد اشاره كنـد. و حافظـة عنـصر آزاد گردد.

(پيچيدگي O(n))

ج: اگر عنصر آخر ليست باشد، آنگاه قسمت Next عنصر ماقبل صـفر مـي‌گـردد و حافظه آزاد مي‌گردد . (پيچيدگي O(n))

```
template <class  datatype>
```

```
void  linkedlist <datatype > :: delete(datatype  p)
// الف- اگر اولين عنصر ليست است
{
list node <datatype >   *i = start;
if (!i )  return;
if (i->data = = p)      //  the first  node
{      start = i->Next;
      delete    i;
      return;
}
else
{      listnode < datatype > *j=i;
       while (i && i->data ! =p)
      {          j=i;
                 i= i->Next;
       }
              if (! i)    //  not  found  !
                return;
              j->Next=i->Next;
             delete  i;
  }
}
```

جستجوي خطي ليست پيوندي

از ابتداي ليست شروع مي‌كنيم، پيمايش ليست تا يافتن عنصر دلخواه ادامه مي‌يابد. پيچيدگي از درجة O(n) است.

```
templete <class  datatype >
 bool   linkedlist <datatype > :: search (datatype   p)
{          list node <datatype >     *i = start;
           if (!i)
                 return  false;
           while (i && i->data ! = p)
                    i = i-> Next;
           if (! i)  return  false;
           else   return   true;
}
```

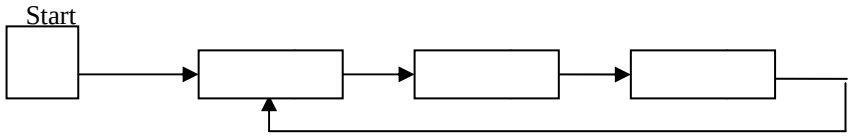
مثال: سربارگزاري عملگر [] براي بدست آوردن دادة عنصر I ام:

```
template< class  datatype>
datatype  linkedlist <datatype>::operator [] (int  i)
 {
       if (! start )
       {
      cerr<<"list is empty!";
```

```
    assert (0);
      }
     if (i < 0)     assert(0);
     int   c= 0;
     listnode <datatype>   *j=start;
     while (j&&  i!=c)
     {          j=j->Next;
                c++;
      }
      if ( !j)
      {
    datatype temp;
    cerr<<"out of range!";
    return temp;
       }
      else
      return  j->data;
}
```

٣-٣ ليست پيوند حلقوي: **(circular linked list)**

ليست حلقوي، يك ليست پيوندي است كه Nextعنصر انتهايي آن، به ابتداي ليست اشاره مي‌كند در اين صورت اگر يك اشاره گر به هر محل از ليست داشته باشيم، به تمامي نودهاي ليست دسترسي داريم.



نكته: در ليست حلقوي قسمت Next خانة آخر Null نيست و به عنصر اول، (شروع ليست) اشاره مي‌كند.

تعريف **ADT** براي ليست حلقوي

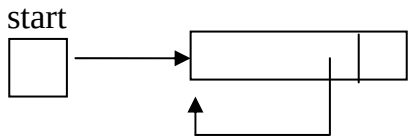
```
template < class  datatype >
class clist
  {  friend  ostream  & operator << (ostream &, const clist<datatype> &);
public :
          clist ( );
          ~ clist ( );
          void  Insertfirst (datatype);
          void  Insertlast (datatype);
          bool search(datatype p);
          void delete(datatype);
          bool isempty( );
```

```
        private:
                listnode<datatype> *start;

};
```

ليست تهي: ليستي كه اشاره‌گر start آن Null باشد.

نمايش ليست حلقوي با يك عنصر



اضافه كردن به ابتدا

```
template  <class    datatype>
void clist <datatype > :: Insertfirst (datatype   p)
{               listnode <datatype >  *  i=new    listnode < datatype > (p);
                  assert (i! = 0 );
                   if ( !  start )
                   {       start = i;
                    i->Next = start;
                   }
                   else
                    {       listnode <datatype >   * j= start;
                             i->Next  = start;
                              While (j->Next  ! = start )
                                        j=j->Next;
                             start = i;
                             j->Next = start;
                     }   // else
}  // function
```

در زمان اضافه كردن يك نود به ابتداي ليست، بايد قسمت Next عنصر آخر ليست نيز به روز درآيد بنابراين پيچيدگي از درجة O(n) است.

```
template < class  datatype >
 void   clist < datatype > :: insertlast  (datatype     p)
{             listnode< datatype > * i=new listnode <datatype >(p);
              assert (i!=0);
              listnode <datatype>    *j= start;
              if (! start)
              {        start  =i;
                        i->Next = start;
               }
               else
               {        while (j->Next !=start )  // finding   last node
                              j = j->Next;
```

```
            j->Next = i;
            i->Next = start;
          }
}
```

چاپ كلية عناصر ليست .

```
template <class datatype>
ostream & operator <<(ostream & out , const clist <datatype> & p)
{       lisnode <datatype> * i = p.start;
        if (! i)  return  out;
        while (i->Next !=p.start)
        {       out << i->data;
                i= i->Next;
        }
        return out;
}
```

٣-٤- ليست دو طرفه: 2- way linked list

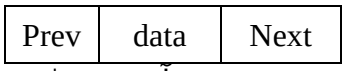
در اين نوع ليست پيوندي علاوه بر اشاره گر به عنصري بعدي، اشاره گر به عنصر قبلي نيز ذخيره مي‌گردد و اشاره گر به عنصر بعد و قبل را داريم .

ساختار هر نود در اين نوع ليست پيوندي بصورت زير است.

١) Next اشاره گر به عنصر بعدي

٢) Prev اشاره گر به عنصر قبلي

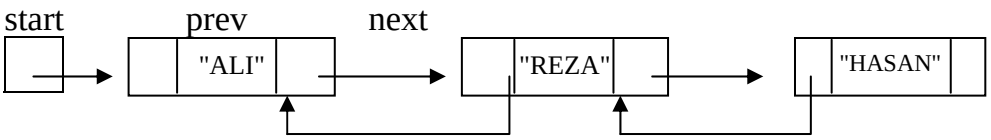
٣) Data دادة هر نود.



نكته ١: در عنصر اول ليست، قسمت prev آن Null است.

نكته ٢: عنصر آخر ليست، قسمت Next آن Null است.

مثالي از يك ليست پيوندي دو طرفه



پياده سازي كلاس هر نود.

```
template < class datatype>
class node
{       friend class twlinklist<datatype>;
        public :
```

```
                node (datatype );
                ~ node ( ) { }
                private :
                        node <datatype>     * Next ,    * prev;
                        datatype  data;
};
template < class datatype>
node < datatype > :: node(datatype   p)
{
        data = p;
        Next = prev = Null;
}
templata < class  datatype >
class   twlinklist
{
public:
        twlinklist ( );
        ~ twlink list ( );
        void     Insertlast ( datatype);
        void  Insertfirst( datatype);
        void  delete( datatype);
private:
        node <datatype>    * start;

};
```

درج در ابتدا:

درج در ابتدا باعث تغيير، start مي‌گردد و پيچيدگي بدليل مشخص بودن ابتدا و انتها، از درجة O(1) است.

```
template      <class datatype >
void    twlinklist <datatype >:: Insertfirst(datatype   p)
 {      node  <datatype >   *i=new   node <datatype >(p);
        assert(i! = 0 );
         if  (! start )
         {          start =i;
                    return;
      }
      else
      {           i->Next = start;
                    start = i;
      }
}
```

درج در انتهاي ليست:

در اين حالت بايد، نود آخر ليست را پيدا كنيم و نـود جديـد را درج كنـيم در ايـن حالت پيچيدگي از درجة O(n) است.

```
template  < class  datatype >
voide    twlinklist <datatype> :: Insertlast (datatype   p)
 {        node <datatype > *i = new node <datatype >(p);
          assert( i! = 0);
           if (! start )
           {        start = i;
                     return;
            }
           else
           {      node <datatype >   *j= start;
                 while ( j->Next)
                       j = j->Next;
                 j->Next = i;
           }
}
```

حذف يك نود

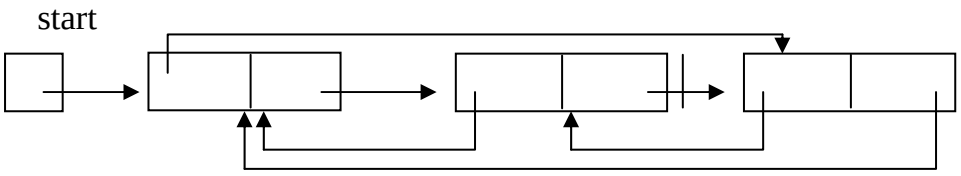
```
template      <class  datatype >
void      twlinklist<datatype >:: Delete(datatype     p)
 {        if (! start )     return;
          node <datatype >   *i = start;
          if ( i->data = = p)   //  first  node
          {          start = i->Next;
                     i->Next->prev = Null;
                      delete    i;
                       return;
     }
          i=i->Next;
          while (i && i->data ! = p)
                i = i->Next;
        if (!i)
       {      cerr <<``con.`t  find  your  data``;
               return;
        }
        else    if (i->Next)
        {
              i->Next->prev  = i->prev;
              i->prev->Next = i->Next;
              delete   i;
          }
          else
```

```
        {     i->prev->Next = Null;
              delete   i;
         }
  }
```

ليست دو طرفه حلقوي: **circular 2 way link list**

در اين ليست اشاره گر به نود قبل و نود بعد موجود است و قسمت Next عنصر آخر به نود اول اشاره مي‌كند و قسمت prev عنصر اول به عنصر آخر اشاره مي‌كند.

مثال:



تمرين: كلاس ليست دو طرفة حلقوي را پياده سازي كنيد. توابع – Delete insertfirst و insertlast را بنويسيد.

۳-۵- ليست مرتب **Sorted list**

اين نوع ليست بر اساس ديتاي نودها مرتب است، مرتب سازي بصورت صعودي يا نزولي امكان پذير است .

مثال: پياده سازي و بررسي ليست پيوندي مرتب صعودي.

```
template < class datatype >
class  sortedlinklist
 {        public:
               Sortedlinklist ( );
              ~ sortedlinklist ( );
               void Insert(datatype  p);

          private :
                    listnode <datatype>   *start;
};

template <class datatype >
sortedlinklist <datatype > :: sortedlinklist ( )
{start = Null;}

 template < class  datatype >
 sortedlinklist  <datatype >:: ~ sortedlinklist( )
{     listnode < datatype >  * i= start,  * j;
      while (i)
           {      j= i;
```

```
                i=i->Next;
            delete  j;
          }
}
template <class datatype >
void   sortedlinklist < datatype > :: Insert (datatype  p)
{               listnode <datatype >  *i =new  listnode<datatype>(p);
                assert (i! = 0);
                if (! start)
                      start = i;
                listnode<datatype> *j ,*l=0;
                j= start;
                while (j && j->data <=p)
                {          l = j ;
                            j=j->Next;
                }
            if(!l)
            {   i->Next=start;
                start=i;
            }
            else
            {
                i->Next = j;
                l->Next = i;
            }
 }
```

مثال: عملكرد + را براي تركيب (merge) دو ليست مرتب صعودي، در حالت دوست سربارگذاري كنيد .

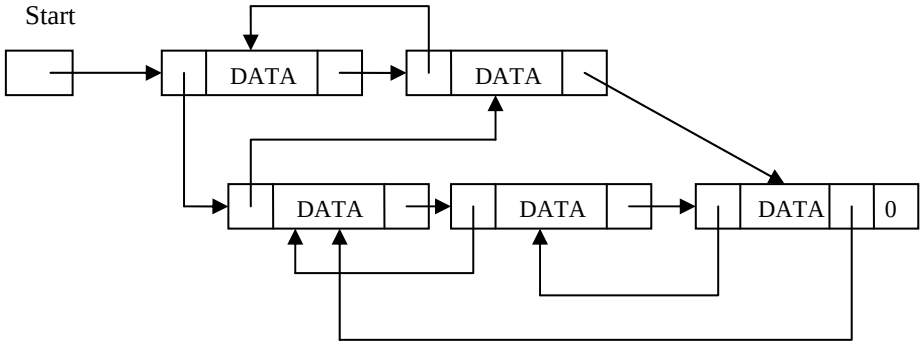
```
template <class  datatype >
sortedlinklist < datatype > operator+(const sortedlinklist <datatype >
&a, const sortedlinklist <datatype> & b)
{
    listnode < datatype > *i = a.start;
    listnode <datatype>   *j=b.start;
    sortedlinklist <datatype>  * temp = new  sortedlinklist <datatype >(
        );
    while (i && j)
    {
         if (i->data < j->data)
        {
              temp->Insertlast(i->data);
              i = i->Next;
         }
        else if ( j->data < i->data)
        {          temp->Insertlast( j->data );
             j = j->Next;
```

```
    }
        else
         {      temp->Insertlast (j->data );
                i = i->Next;
                j = j->Next;
          }
       while(i)
       {
            temp->Insertlast(i->data);
                i = i->Next;
           }
       while(j)
       {
            temp->Insertlast(j->data);
                j = j->Next;
           }
           return *temp;
}
```

٣-٦- ليست‌هاي چند پيوندي **Multi Linked Lists**

در اين حالت از هر نود چند پيوند به نودهاي ديگر، خارج مي‌شود.

شكل زير مثالي از يك ليست چند پيوندي است.



از جمله كاربردهاي اين نوع ليست، در پياده سازي درختها و گرافها است.

مثالي كاربردهاي ليست پيوندي:

١- پياده سازي ماتريس پراكنده با ليست پيوندي:

همانطور كه قبلاً بيان شد، مي‌توان فقط عناصر غير MOD را ذخيره كرد، ليست پيوندي بدليل اينكه در زمان استفاده، حافظه اختصاص مي‌دهد، مدلي بهينه از نظر حافظه دارد.

قسمت ديتا براي هر نود مي‌تواند بصورت زير باشد:

Row	Col	Value

۲- پياده سازي چند جمله اي با ليست پيوندي:

چند جمله‌اي‌ها را نيز ميتوان با ليست پيوندي پياده سازي كرد.

در اين صورت ساختار قسمت ديتا براي بصورت زير است:

ضريب	توان

```
template <class T>
class mydata
{    friend class linkedlist<mydata>;
       friend ostream&operator<<(ostream&,const mydata &);
    public:
        mydata(float,int);
              ~mydata( );
    private:
        float coef;
        int exp;
};
class polynomial
{
    public:
        polynomial(unsigned degree);
        ~ polynomial( );
        void savepoly(float,int);
        polynomial operator+(const polynomial &b);
        .
        .
        .
    private:
        linkedlist<mydata> mylist;
        unsigned degree;
};
```

در اين حالت حذف و اضافه كردن جملات، نودهايي به ليـست پيونـدي را اضـافه، حذف مي‌كند يا تغيير مي‌دهد.

تمرين: سربارگذاري توابع مورد نياز براي كلاس چند جمله اي را بنويسيد.

فصل چهارم

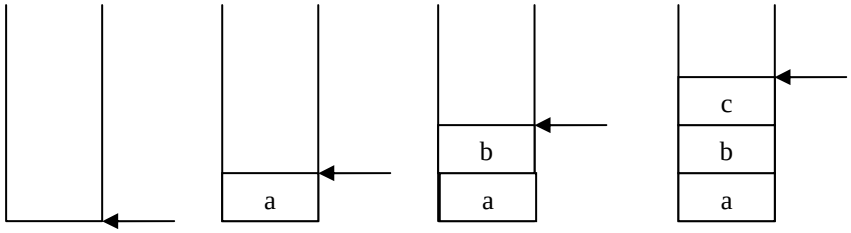
پشته

تعريف: پشته يك ليست ترتيب‌دار است، كه درج و حذف عناصر در محل انتهايي پشته، كه TOP (بالاي پشته) نام دارد، انجام مي‌شود. پشته را مي‌توان به صورت S=(a0,a1,a2,…an) تصور كرد كه a0 عنصر پائين پشته و an عنصر بالاي پشته است.

تعريف ADT براي پشته:

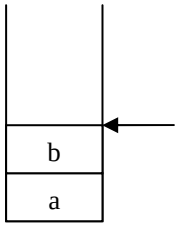
- داراي عملگري بنام PUSH است، كه داده را در روي داده هاي موجود (بالاي پشته) قرار مي‌دهد.
- داراي عملگري بنام POP است، كه داده بالاي پشته، (آخرين عنصر وارد شده) را از درون پشته خارج مي‌كند.
- داراي عملگري بنام TOP است، كه داده بالاي پشته را برمي گرداند ولي آنرا از پشته خارج نمي كند.
- داراي عملگري بنام ISEMPTY است، كه مشخص مي‌كند، آيا پشته خالي است يا خير؟

مثال: نمودار پشته بعد از چند عمليات.



Empty Stack PUSH(a) PUSH(b) PUSH(c)

علامت فلش نشان دهندة TOP در پشته است.



POP

حال ساختار ADT بصورت كلاس Template در C++ پياده سازي مي‌كنيم:

پياده سازي پشته با كمك آرايه، و يا ليست پيوندي امكان پذير است.

الف) پیاده سازي پشته با آرایه:

```
template <class T>
class stack
{
      public:
            stack(int maxsize=100);
            ~stack( );
            T TOP( );
            void PUSH(T mydata);
            T POP( );
            bool ISEMPTY( );
      private:
            T *Array;
            const unsigned size;
            int topstack;
};

template <class T>
stack<T> :: stack(unsigned maxsize)
      : size(mazsize)
{
      Array = new T[maxsize];
      assert(Array!=0);
      topstack=-1;
}

template <class T>
stack<T> :: ~stack( )
{
      delete [] Array;
}

template <class T>
bool stack<T> :: ISEMPTY( )
{
      if( topstack ! = -1 )
            return FALSE;
      else
            return TRUE;
}

template <class T>
T stack<T> :: TOP( )
{
      if( ! ISEMPTY( ) )
            return Array[topstack];
      else
      {
            T temp;
```

```
            cerr << "STACK IS EMPTY!";
            return temp;
        }
}

template <class T>
T stack<T> :: POP( )
{
        if( ! ISEMPTY( ) )
            return Array[topstack--];
        else
        {
            T temp;
            cerr << "STACK IS EMPTY!";
            return temp;
        }
}

template <class T>
void stack<T> :: PUSH(T mydata )
{
        if( topstack = = maxsize)
        {
            cerr << "STACK IS FULL ! ";
            return;
        }
        else
        {
            Array[++topstack] = mydata;
        }
}
```

نكته: در ساختار پشته، نمي‌توان عنصر ديگري به غير از عنصر روي پشته را برداشت.

ب)پياده سازي پشته با ليست پيوندي

```
template <class T>
class stack
{
        public:
            stack( ){ }
            ~stack( ){ }
            T TOP( );
            void PUSH(T mydata);
            T POP( );
            bool ISEMPTY( );
        private:
            linkedlist<T> stacklist;
};

template <class T>
```

```
bool stack<T> :: ISEMPTY( )
{
    return stacklist.ISEMPTY( );//خالي بودن پشته عبارت از خالي بودن ليست است
}

template <class T>
T stack<T> :: TOP( )
{
    if( ! ISEMPTY( ) )
        return stacklist.start->data;
}

template <class T>
T stack<T> :: POP( )
{
    if( ! ISEMPTY( ) )
    {
        T temp = stacklist.start->data;
        stacklist.deletefirst( );// حذف عنصر اول ليست
        return temp;
    }
}

template <class T>
T stack<T> :: PUSH(T mydata)
{
    stacklist.insertfirst(mydata);// درج در ابتداي ليست پيوندي
}
```

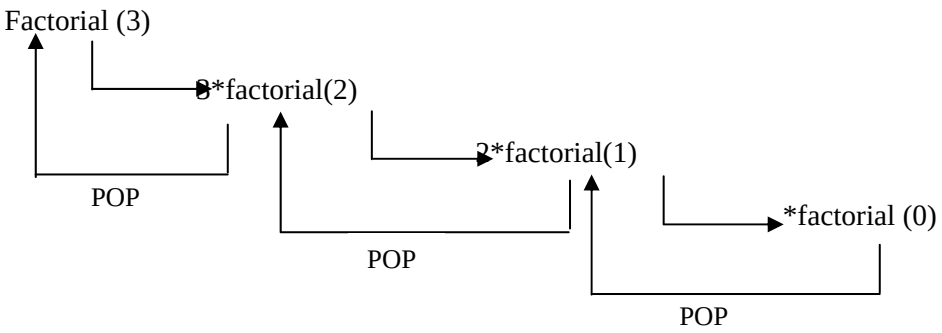
كاربردهاي پشته:

١- توابع بازگشتي

در زمان صدا زدن يك تابع، آدرس برگشت (دستور بعد از صدا زدن تابع) و متغيرهاي محلي در درون پشته قرار داده مي‌شود، و بارسيدن به return (بازگشت از تابع) عنصر روي پشته برداشته مي‌شود (POP) و اجرا ادامه مي‌يابد.

مثال: نمودار پشته را براي محاسبهْ !3 با توجه به تابع زير بنويسيد.

```
int factorial(int i)
{
    if(i==0)
        return 1;
    else
        return i*factorial(i-1);
}
```



٢- عبارات رياضي و تبديل آنها بهم

- عبارت infix:

در عبارات رياضي infix عملگر مابين دو عملوندش، قرار مي‌گيرد.

مثال:

```
1 + 25 * 345
```

- عبارت postfix:

در اين حالت عملگر بعد از عملوندهايش، قرار مي‌گيرد.

مثال:

```
Infix: a+b*c
Postfix: abc*+

Infix: a*b/c
Postfix: ab*c/
```

- عبارت prefix:

در اين نوع عبارات عملگر، قبل از عملوندهايش قرار مي‌گيرد.

مثال:

```
Infix: a+b*c
Postfix: +a*bc

Infix: a*b/c
Postfix: /*abc
```

مزيت عبارتهاي postfix و prefix نسبت به infix اين است كه، احتياج به گذاشتن پرانتز براي اولويت عملگرها نيست. به مثال زير توجه كنيد:

```
Infix: (a+b)/(c+d)
Postfix: ab+cd+/
Prefix: /+ab+cd
```

نكته: وجود پرانتز در عبارت‌هاي postfix و prefix غلط است.

الگوريتم تبديل عبارت **INFIX** به **POSTFIX**:

۱- رشتة INFIX ورودي را از ابتدا شروع به پردازش كنيد تا به انتها (NULL) برسيد.

۲- بادیدن هر عملوند، آنرا به انتهاي رشتة خروجي اضافه كنيد.

۳- بادیدن هر عملگر، با درنظر گرفتن شروط زير آن‌را در پشته قرار دهيد. (PUSH)

الف) اگر پشته خالي است، آنگاه عملگر را PUSH كنيد.

ب) اگر پشته خالي نيست و اولويت TOP كمتر از اولويت عملگري است، كه درون پشته است آنگاه عنصر PUSH مي‌شود. در غيراين‌صورت تا وقتي اولويت عملگر TOP پشته بيشتر مساوي عنصر ورودي است، از پشته، POP مي‌كنيم و در انتهاي رشتة ورودي قرار مي‌دهيم.

٤- با رسيدن به انتهاي رشته ورودي تا وقتي، پشته خالي شود عناصر آنرا به انتهاي رشتة خروجي اضافه مي‌كنيم.

۵- پرانتز داراي بالاترين اولويت است، ولي هر عملگري مي‌تواند روي پرانتز باز PUSH شود و با رسيدن به پرانتز بسته آنقدر از پشته بر مي‌داريم و به انتهاي رشتة خروجي اضافه مي‌كنيم، تا به پرانتز باز در پشته برسيم. پرانتزها را نيز حذف مي‌كنيم.

مثال: نمودار پشته را براي تبديل عبارت a+b/c به POST FIX ترسيم كنيد.

نماد ورودي	خروجي	پشته
A	a	-
+	a	+
B	ab	+
/	ab	+/
C	abc	+/
NULL	abc/+	-

مثال نمودار پشته را براي تبديل (a+b)/(c+d)*k ترسيم كنيد.

نماد ورودي	خروجي	پشته
(	-	(
a	a	(
+	a	(+
b	ab	(+
)	ab+	-
/	ab+	/
(	ab+	/(
c	ab+c	/(
+	ab+c	/(+
d	ab+cd	/(+
)	ab+cd+	/
*	ab+cd+/	*
k	ab+cd+/k	*
NULL	ab+cd+/k*	-

٣- كنترل پرانتز - براكت و آكولادهاي تو در تو در عبارات رياضي:

الگوريتم:

١- باديدن پرانتز باز - آكولاد باز - براكت باز ،در رشتة ورودي آنرا در پشته قرار دهيد.

٢- باديدن پرانتز بسته - آكولاد بسته - براكت بسته ،در رشتة ورودي، TOP پشته را با آن مقايسه كنيد اگر هم جنس نبود – خطاست و اگر پشته خالي باشد نيز خطاست.

٣- اگر در انتهاي رشته ورودي پشته خالي نشده باشد نيز خطاست. (كمبود) يا] يا })

مثال: نمودار پشته در كنترل عبارت (a+b)-[(a+c)*d-{c+d}]*b

ورودي	پشته
(	(
a	(
+	(
b	(
)	-
[	[
(	[(
a	[(
+	[(
c	[(
)	[
*	[
d	[
-	[

POP & COMPARE

POP & COMPARE

{	[{
c	[{
+	[{
d	[{
}	[
]	-
NULL	-

POP & COMPARE
POP & COMPARE

با توجه به نمودار پشته در انتهاي عمليات، پشته خالي است، پس عبارت درست است.

٤- محاسبه عبارت‌هاي POSTFIX

الگوريتم:

ورودي: رشتة POSTFIX

خروجي: نتيجة محاسبة عبارت

١-با ديدن هر عملوند، ارزش آنرا درون پشته قرار دهيد.

٢-با ديدن هر عملگر، دو بار POP كنيد و عملگر را انجام دهيد، و مجدداً نتيجه را درون پشته قرار دهيد.

٣-در انتهاي رشته، فقط يك مقدار كه نتيجة عبارت است، بايـد درون پـشته بـاقي مانـده باشد.

مثال: نمودار پشته براي محاسبة عبارت ab+c/ را رسم كنيد.(a=1,b=2,c=3)

نماد ورودي	پشته	عمليات
a	<1>	PUSH
b	<1><2>	PUSH
+	<3>	POP+POP PUSH
c	<3><3>	PUSH
/	<1>	POP/POP PUSH
NULL	<1>	TOP

فصل پنجم

صف

تعريف: ساختاري از عناصر داراي ترتيب (مرتب) است، كه عمليات درج در انتهاي صف، و عمليات حذف، از ابتداي صف انجام مي‌شود. بنابراين عنصري كه زودتر وارد صف شده، زودتر خارج خواهد شد. به اين دليل به ساختار صف FIRST IN FIRST OUT (FIFO) نيز مي‌گويند.

تعريف **ADT** براي صف:

با توجه به تعريف فوق داراي عملگرهاي زير نيز مي‌باشد :

- صف داراي عملگري بنام Front است، كه عنصر جلو صف (سر صف) را برمي گرداند ولي آنرا از صف خارج نمي كند.
- صف داراي عملگري بنام Add است، كه عنصري را در انتهاي صف درج مي‌كند.
- صف داراي عملگري بنام Delete است، كه عنصر جلو صف را حذف مي‌كند.
- صف داراي عملگري بنام ISEmpty است، كه مشخص ميكند آيا صف خالي است يا خير؟

صف را با آراية حلقوي (Circular Array)، و يا با ليست پيوندي پياده سازي مي‌كنند.

الف) پياده سازي صف با آراية حلقوي

بدليل آنكه ممكن است، سرعت ورود و خروج اطلاعات از صف يكسان نباشد، بايد ساختار صف را با آراية حلقوي كه ابتدا و انتهاي آنرا به هم متصل كرده‌ايم پياده سازي كنيم كه نيازي به شيفت دادن (جابجايي) عناصر نباشد.

```
template <class T>
class Queue
{
    public:
        Queue(int maxsize=100);
        ~Queue( ){delete  [ ] CArray;}
        void Add(T mydata);
        T Front( );
        T Delete( );
        bool ISEmpty( );
        bool ISFull( );
    private:
        T *CArray;//آراية حلقوي پويا
        const int size;//سايز صف
        int front,rear;//ابتدا و انتهاي صف
        int add(int i);// افزايش چرخشي
};
```

براي آنكه بين صف خالي و پر (FULL) تفاوت حاصل شود، بين front و rear يك خانه بدون استفاده در نظر مي‌گيريم.

شرط صف پر (FULL): اگر به rear به صورت حلقوي اضافه كنيم و مساوي front شود صف پر است.

شرط صف خالي: اگر front = = rear باشد، آنگاه صف خالي است.

```
template <class T>
Queue<T>::Queue(int maxsize)
    : size(maxsize)
{
    CArray = new T[size];
    assert(array!=0);
    front=rear=0;
}

template <class T>
bool Queue<T> :: ISEmpty( )
{
    if(front = = rear)
        reurn TRUE;
    else
        return FALSE;
}

template <class T>
bool Queue<T> :: ISFull( )
{
    if(front = = add(rear))
        reurn TRUE;
    else
        return FALSE;
}

template <class T>
int Queue<T> :: add(int i)
{
    return (i+1) % size;
}

template <class T>
void Queue<T> :: Add(T mydata)
{
    if( ! ISFull( ) )
    {
        CArray[rear]=mydata;
        rear=add(rear);
    }
```

```
    else
        cerr << "Queue Is Full !";
}

template <class T>
T Queue<T> :: Front( )
{   T temp=CArray[front];
    if( ! ISEmpty( ) )
        return temp;
    else
    {
        cerr << "QUEUE IS EMPTY";
        return temp;//باید مقداري را برگردانیم
    }}
template <class T>
T Queue<T> :: Delete( )
{   T temp=CArray[front];
    if( ! ISEmpty( ) )
    {
        front=add(front);
        return temp;
    }
    else
    {
        cerr << "QUEUE IS EMPTY";
        return temp;//باید مقداري را برگردانیم
    }
}
```

ب) پیاده سازي صف با لیست پیوندي:

در این حالت درج معادل با درج در انتهاي لیست است و حذف عبارت از حذف عنـصر اول لیست پیوندي است.

```
template <class T>
class Queue
{
    public:
        Queue( ){ list.start=0};
        ~Queue( ){ }
        void Add(T mydata){ list.InsertLast(mydata);}
        T Front( );
        T Delete( );
        bool ISEmpty( ){return list.ISEmpty( );}
        bool ISFull( ){return FALSE;}
    private:
        linkedlist<T> list;
};
```

```
template <class T>
T Queue<T> :: Front( )
{
     if( ! ISEmpty( ) )
          return list.start->data;
}

template <class T>
T Queue<T> :: Delete( )
{
     if( ! ISEmpty( ) )
     {
          T  temp= list.start->data;
          list.DeleteFrist( );
          return temp;}}
```

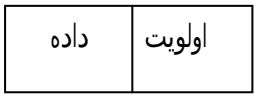
۵-۲- صف اولويت **PRIORITY QUEUE**

تعريف: صف اولويت، يك صف معمولي است كه ترتيب پشت سرهم قرار گرفتن عناصر، ترتيب ورود (زمان ورود) نيست، بلكه اولويت آنها است، و عناصر در درون صف بر اساس اولويتشان مرتب مي‌شوند.

اگر اولويت را برابر زمان ورود عناصر قرار دهيم، آنگاه صف اولويت، به صف معمولي تبديل مي‌شود.

پياده سازي با كمك آراية حلقوي:

ابتدا ساختار هر عنصر درون صف را تعريف مي‌كنيم.




```
template<class T>
class QueueNode
{
     friend class PQueue;
friend void swap(QueueNode<T> &,QueueNode<T> &);
          public:
          QueueNode( ) { }
          ~QueueNode( ) { }
     private:
          T data;
          int priority; // فرض ميكنيم عدد صحيح بزگتر نشان دهندة اولويت بالاتر است
};

template<class T>
class PQueue
{    friend void swap(QueueNode<T> &,QueueNode<T> &);
     public:
          Queue(int maxsize=100);
```

```
        ~Queue( ){delete  [ ] CArray;}
        void Add(T mydata, int Prr=0);
        T Front( );
        T Delete( );
        bool ISEmpty( );
        bool ISFull( );
    private:
        QueueNode<T> *CArray;//آراية حلقوي پويا
        const int size;//سايز صف
        int front,rear;//ابتدا و انتهاي صف
        int add(int i);// افزايش چرخشي
        int pred(int i); // كاهش چرخشي  }

template <class T>
PQueue<T>::PQueue(int maxsize)
    : size(maxsize)
{
    CArray = new QueueNode<T>[size];
    assert(array!=0);
    front=rear=0;
}

template <class T>
bool PQueue<T> :: ISEmpty( )
{
    if(front = = rear)
        reurn TRUE;
    else
        return FALSE;
}

template <class T>
bool PQueue<T> :: ISFull( )
{
    if(front = = add(rear))
        reurn TRUE;
    else
        return FALSE;
}

template <class T>
int PQueue<T> :: add(int i)
{
    return (i+1) % size;
}
```

```
template <class T>
int PQueue<T> :: pred(int i)
{
      if( i!=0)
            return (i-1);
      else
            return size-1;
}

template <class T>
void PQueue<T> :: Add(T mydata, int Prr)
{
      if( ! ISFull( ) )
      {
            CArray[rear].data=mydata;
            CArray[rear].Priority=Prr;
            rear=add(rear);
            int i=rear;
            while(i!=front && CArray[i].Priority > CArray[pred( i )].Priority)
            {
                  swap(CArray[i],CArray[pred( i )]);
                  i=pred( i );
            }
      }
      else
            cerr << "Queue Is Full !";
}

template <class T>
T PQueue<T> :: Front( )
{     T temp=CArray[front].data;
      if( ! ISEmpty( ) )
            return temp;
      else
      {
            cerr << "QUEUE IS EMPTY";
            return temp;//باید مقداري را برگردانیم
      }
}

template <class T>
T PQueue<T> :: Delete( )
{     T temp=CArray[front].data;
      if( ! ISEmpty( ) )
      {
            front=add(front);
            return temp;
      }
      else
```

```
    {
        cerr << "QUEUE IS EMPTY";
        return temp;//بايد مقداري را برگردانيم
    }
}
```

تمرين: يك پمپ بنزين را كه اتومبيل‌ها با سه اولويت مختلف وارد مي‌شوند، T ثانيه منتظر سوخت گيري شده و خارج مي‌شوند . شبيه سازي كرده و برنامه را براي ١٠٠ دادة تصادفي تست كنيد.

فصل ششم

درخت‌ها

تعريف: يك درخت يك مجموعة متناهي از يك يا چند نود (گره) است كه:

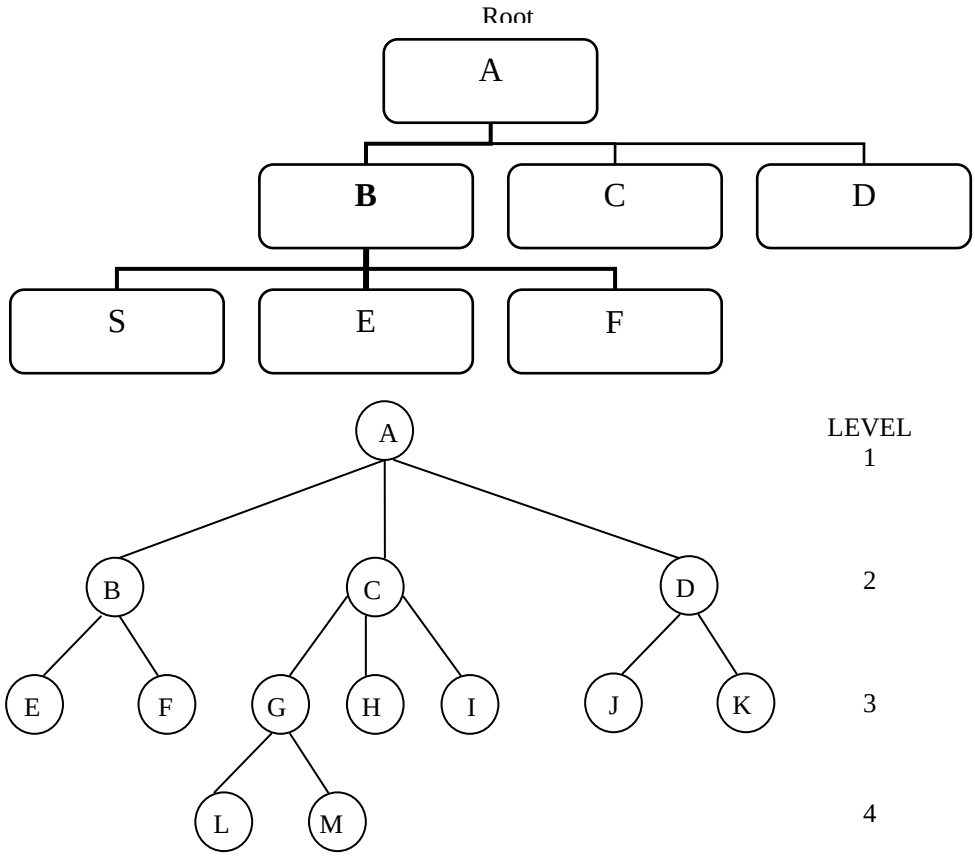
١- يك نود با طراحي خاص، وجود دارد كه ريشه (ROOT) ناميده مي‌شود.

٢- بقية نودها به $n \geq 0$ نود مجموعة مجزا، $(T_1,T_2,\ldots,T_n)$ تقسيم مي‌شوند، كه هر كدام از اين نودها خود يك درخت است.هر كدام از $(T_1,T_2,\ldots,T_n)$ زيردرخت‌هاي ريشه ناميده مي‌شود.

توجه كنيد كه اين تعريف، يك تعريف بازگشتي است.

درخت‌ها جهت ذخيرة سلسله مراتبي اطلاعات بسيار مفيد است. و با كمك آن مي‌توان اطلاعات را دسته بندي نمود.

مثال نمونه‌اي از يك درخت



٦-١- درخت K تايي يا جنگل (K-Tree or FOREST)

در اين نوع درخت K يك عدد ثابت و مثبت است، كه حداكثر تعداد فرزندان هـر نـود را مشـخص مي‌كند، در اين درخت هر نود مي‌تواند حداكثر K فرزند داشته باشد.

تعريف ADT براي درخت K تايي

```
template <class T>
class KTree
{
    public:
        KTree( );
        ~KTree( );
        //سمت چپ ترين فرزند يك نود
        const KNode<T> *LeftMostChild(const KNode<T> *);
        //برادر سمت راست (نود مجاور يك نود)
        const KNode<T> *RightSibling(const KNode<T> *);
        //پدر يك نود (نود سطح بالاتر يك نود)
        const KNode<T> *Parent(const KNode<T> *);
        // ايجاد ريشة درخت
        void AddRoot(T mydata);
        // ايجاد فرزندان براي يك نود
        void AddChild(const Knode<T> *p,T mydata);
        //inorder پيمايش
        void inorder( );
        // preorder پيمايش
        void preorder( );
        // postorder پيمايش
        void postorder( );
        // level order پيمايش
        void levelorder( );
        //ارتفاع درخت
        int height( );
        //سطح يك نود
        int level(const KNode<T> *);
    protected:
        Knode<T> *root;//ريشة درخت
        unsigned degree;//  (K) درجة درخت
};
template <class T>
const int maxK =  10;
class KNode //كلاس ساختار هر نود درخت
{   public:
```

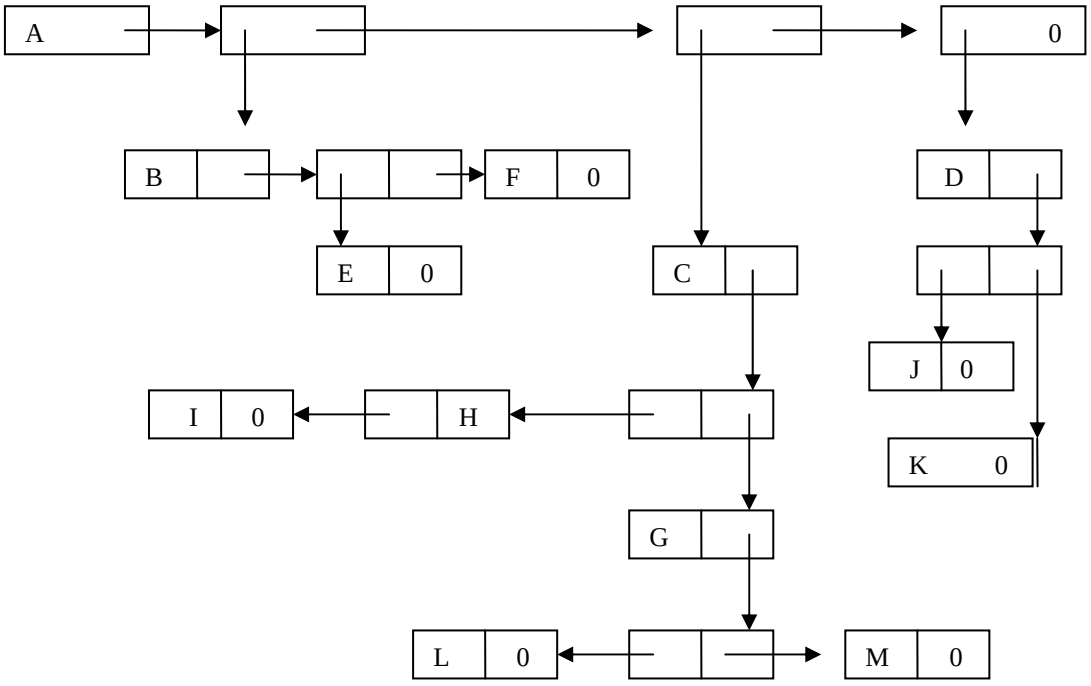
ساختار هر نود

DATA	CH[0]	CH[1]	…	CH[n]

```
        KNode(T mydata);
        ~KNode( ){ }
    private:
        T data;
        KNode<T> * childs[maxK]; }
```

درخت تهي: درختي است كه اشاره گر root براي آن NULL باشد.

براي پياده سازي درخت مي‌توان از ليست پيوندي استفاده كرد، در اين حالت بعضي از نودهاي ليست جنبة ارتباطي دارند.



نمايش درخت **k** تايي با ليست پيوندي

```
template <class T>
KNode<T>:: KNode(T mydata)
{
    for(int i=0;i<maxK;i++)
        childs[i]=0;
    data=mydata;
}

template <class T>
KTree<T>:: KTree( )
{
    root = 0;
}
```

```
template <class T>
KTree<T>:: ~KTree( )
{
    //آزادسازي حافظة تك تك نودهاي درخت}
```

سمت چپ ترين فرزند: در اين حالت بايد سمت چپ ترين اشاره گر فرزندان را برگردانيم، و اين به آن معني است كه فرزندان از چپ براست بوجود مي‌آيند.

```
template <class T>
const KNode<T> *KTree<T>:: LeftMostChild(const KNode<T> *i)
{   if( !i ) return 0;
    return i->childs[0];
}
template <class T>
const KNode<T> *KTree<T>:: RightSibling(const KNode<T> *i)
{   if( !i ) return 0;
    KNode<T> *j=parent( i );
    for(int k=0;k<maxK&&j->childs[k]!=i;k++);
    if(k=maxK)  return 0;
    else return j->childs[k+1];
}

template <class T>
const KNode<T> *KTree<T>:: parent(const KNode<T> *i)
{
    if( !i ) return 0;
    if( !root ) return 0;
    if( i = = root) return 0;
    Queue<KNode<T> *> Q;// ايجاد صف
    KNode<T> *j=root;
    while(j)
    {
        for(int k=0;k<maxK;k++)
            if(j->childs[k] = = i)
                return j;
            else if(j->childs[k])
                Q.Add(j->childs[k]);
        if(! Q.ISEmpty( ))
        {
            j=Q.Front( );
            Q.Delete( );
        }
        else
            j=0;
    }
    return 0;
}
```

پياده سازي مخرب درخت

```
template <class T>
KTree<T>:: ~KTree( )
{
    //آزادسازي حافظة تك تك نودهاي درخت
    KNode<T> *i=root;
    if(!i) return;
    Queue<KNode<T> *> Q;
    while(i)
    {
        for(int j=0;j<maxK;j++)
            if(i->childs[j])
                Q.Add(i->childs[j]);
            delete i; //آزاد سازي حافظه
            if(Q.ISEmpty( ))
              i=0;
            else
            {
              i=Q.Front( );
              Q.Delete( );
            }
    }
}
```

٦-٢- پيمايش درخت **Tree Traversal**

منظور از پيمايش درخت، روشي است كه در آن تمام نودهـاي درخـت ديـده (VISIT) شود، تفاوت ميان روش‌هاي مختلف، در ترتيب ديده شده نودها است.

منظور از VISIT انجام عمليات دلخواه بروي نود است.

پيمايش درخت به چهار روش امكان پذير است.

١-INORDER

در اين روش پيمايش طبق الگوريتم زير انجام مي‌شود:

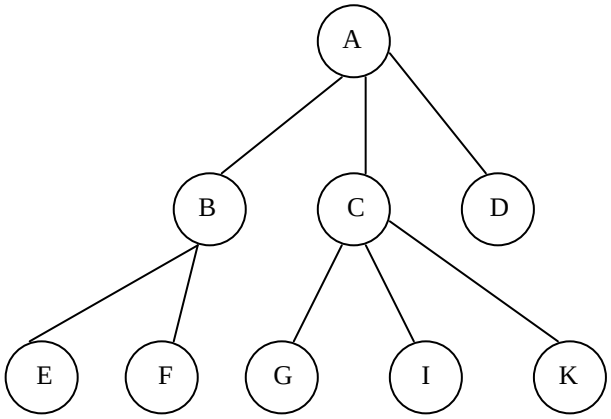
الف) از ريشه شروع مي‌كنيم،

ب) سمت چپ ترين فرزند را INORDER پيمايش مي‌كنيم.

ج)ريشه را VISIT مي‌كنيم.

د)مابقي فرزندان را INORDER پيمايش مي‌كنيم.

مثال: درخت روبرو را درنظر بگیرید نتیجة پیمایش INORDER را بنویسید.



نتیجة پیمایش: EBFAGCIKD

POSTORDER-۲

طبق الگوریتم زیر عمل کنید:

الف) از ریشه شروع کنید.

ب) ابتدا سمت چپ ترین فرزند را POSTORDER پیمایش مي‌شود.

ج) بقیة فرزندان را POSTORDER پیمایش کنید.

د) ریشه را VISIT کنید

مثال: نتیجة پیمایش درخت در حالت POSTORDER

EFBGIKCD**A**

نکته: در این حالت ریشه، آخرین نود خواهد بود.

PREORDER-۳

طبق الگوریتم زیر عمل کنید:

الف) از ریشه شروع کنید و آنرا VISIT کنید.

ب) ابتدا سمت چپ ترین فرزند را PREORDER پیمایش مي‌شود.

ج) بقیة فرزندان نیز PREORDER پیمایش مي‌گردد.

مثال: نتیجة پیمایش درخت:

ABEFCGIKD

نکته: در این حالت ریشه، اولین نود خواهد بود.

٤- LEVEL ORDER

در اين حالت نودهاي درخت، سطح به سطح، از چپ براست، VISIT مي‌شود.

مثال: نتيجة پيمايش سطحي درخت:

ABCDEFGIK

پيمايش INORDER درخت K تايي

```
tamplate <class T>
void KTree<T>:: inorder( )
{
     inorder(root);//call another type of this function
}

tamplate <class T>
void KTree<T>:: inorder(KNode<T> *i)
{
     if( !i ) return;
     inorder(LeftMostChild(i));
     cout << i->data;
     for(int j=1;j<maxK;j++)
          inorder(i->childs[j]);
}
```

پيمايش POSTORDER درخت K تايي

```
tamplate <class T>
void KTree<T>:: postorder( )
{
     postorder(root);//call another type of this function
}

tamplate <class T>
void KTree<T>:: postorder(KNode<T> *i)
{
     if( !i ) return;
     postorder(LeftMostChild(i));
     for(int j=1;j<maxK;j++)
          postorder(i->childs[j]);
     cout << i->data;
}
```

پيمايش PREORDER درخت K تايي

```
tamplate <class T>
void KTree<T>:: preorder( )
{
     preorder(root);//call another type of this function
}
```

```
tamplate <class T>
void KTree<T>:: preorder(KNode<T> *i)
{
    if( !i ) return;
    cout << i->data;
    preorder(LeftMostChild(i));
    for(int j=1;j<maxK;j++)
        preorder(i->childs[j]);
}
```

پيمايش هاي فوق داراي پيچيدگي از درجة ((O(n log(n است.

پيمايش LEVEL ORDER

```
template <class T>
void KTree<T>:: levelorder( )
{
    if( !root ) return;
    Queue<KNode<T> *> Q;// ايجاد صف
    KNode<T> *j=root;
    while(j)
    {   cout << j->data;
        for(int k=0;k<maxK;k++)
            if(j->childs[k])
                Q.Add(j->childs[k]);
        if(! Q.ISEmpty( ))
        {
            j=Q.Front( );
            Q.Delete( );
        }
        else
            j=0;
    }
    return;
}
```

مثال: يك درخت K تايي مفروض است، تابعي براي جستجوي درخت بنويسيد.

الگـــوريتم: مـــي‌تـــوان درخـــت را بـــه روش‌هـــاي PREORDER، POSTORDER، INORDER يا LEVELORDER پيمايش كرد و نود را جستجو كرد.

```
template <class T>
KNode<T> *KTree<T>:: search(T sample)
{
    if( !root ) return 0;
    Queue<KNode<T> *> Q;// ايجاد صف
    KNode<T> *j=root;
    while(j)
    {   if(j->data = = sample)
            return j;
```

```
            for(int k=0;k<maxK;k++)
                if(j->childs[k])
                    Q.Add(j->childs[k]);
            if(! Q.ISEmpty( ))
            {
                j=Q.Front( );
                Q.Delete( );
            }
            else
                j=0;
        }
        return 0;
}
```

٦-٣- ارتفاع درخت: **Height**

ارتفاع درخت ماكزيمم، LEVEL تمامي نودهاي درخت است. براي آنكه بتوان ارتفاع را بدست آورد كافي است درخت را پيمايش كنيم، و ماكزيمم سطح نودهايي كه پيمايش مي‌شود، بدست مي‌آوريم.

بدست آوردن سطح يك نود:

براي بدست آوردن سطح يك نود كافي است، آنقدر PARENT را صدا بزنيم تا به ريشة درخت برسيم، كافي است اين مراحل را بشماريم، نتيجه سطح نود خواهد بود.

```
template <class T>
int KTree<T>:: level(const KNode<T> *i)
{
    if( ! i ) return 0;
    int l=0;
    while( i )
    {
        l++;
        i=parent( i );
       }
       return l;
}
template <class T>
int KTree<T>:: height( )
{
    if( !root ) return 0;
    int max=1;
    Queue<KNode<T> *> Q;// ايجاد صف
    KNode<T> *j=root;
    while(j)
    {   if(level(j)>max)
            max=level( j );
        for(int k=0;k<maxK;k++)
            if(j->childs[k])
```

```
                Q.Add(j->childs[k]);
        if(! Q.ISEmpty( ))
        {
            j=Q.Front( );
            Q.Delete( );
        }
        else
            j=0;
    }
    return max;
}
```

تمرين:

١- تابع پيمايش درخت K تايي را بـروش INORDER، غيربازگـشتي بـه كمـك پـشته بنويسيد.

٢- تابع پيمايش درخت K تايي را بروش PREORDER، غيربازگشتي به كمـك پـشته بنويسيد.

٣- تابع پيمايش درخت K تايي را بروش POSTORDER، غيربازگشتي به كمك پشته بنويسيد.

٤- تابعي بنويسيد كه از درخت K تايي كپي بگيرد.(COPY CONSTRUCTORE)

٥- تابعي براي محاسبة تعداد تكرار يك داده در درخت k – تايي بنويسيد.

فصل هفتم

درخت دو دویی

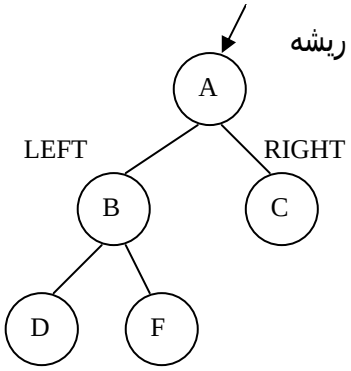
تعريف:

درخت دودويي مجموعه‌اي است متناهي از تعدادي نود كه:

- نودي با طراحي خاص بنام ريشه وجود دارد، كه ممكن است، NULL باشد.
- ريشه دو فرزند چپ و راست (زيردرخت چپ و راست) دارد، كه خود درخت دودويي هستند.

توجه: تعريف درخت دودويي به صورت بازگشتي بيان شده است.

مثال: درخت دودويي



تعريف **ADT** درخت دودويي

درخت دودويي داراي عملگرهاي زير است:

١- اضافه كردن فرزند سمت چپ AddLeftChild

٢- اضافه كردن فرزند سمت راست AddRightChild

٣- ايجاد ريشه AddRoot

٤- بدست آوردن ارتفاع درخت Height

٥- بدست آوردن سطح يك نود Level

٦- بدست آوردن والد يك نود Parent

٧- پيمايش Inorder درخت

٨- پيمايش Postorder درخت

٩- پيمايش Preorder درخت

١٠- پيمايش Levelorder درخت

```
template <class T>
class BNode
{    friend class BTree;
     public:
          BNode(T mydata){data=mydata;  left=right=0;}
          ~BNode( ) { }
```

```
    protected:
        T data;
        BNode<T> *left, *right;
};

template <class T>
class BTree
{
    public:
        BTree( ){root=0;}
        ~BTree( );
        void AddRoot(T mydata);
        void AddLeftChild(T mydata, BNode<T> *i);
        void AddRightChild(T mydata, BNode<T> *i);
        void Inorder( );
        void Postorder( );
        void Preorder( );
        void Levelorder( );
        int Level(const BNode<T> *);
        const BNode<T> *parent(const BNode<T> *);
        int Height( );
    protected:
        BNode<T> *root;
        void Inorder(const BNode<T> *);
        void Postorder(const BNode<T> *);
        void Preorder(const BNode<T> *);
};

template<class T>
BTree<T>:: ~BTree( )
{
    //آزادسازي حافظة تك تك نودهاي درخت
}

template<class T>
void BTree<T>:: AddRoot(T mydata)
{
    BNode<T> *t=new BNode<T>(mydata);
    assert( t!=0);
    root=t;
}

template<class T>
void BTree<T>:: AddLeftChild(T mydata, BNode<T> *i)
{   if( i->left ) return;
    BNode<T> *t=new BNode<T>(mydata);
    assert( t!=0);
    i->left=t;
```

```
}

template <class T>
void BTree<T>:: AddRightChild(T mydata, BNode<T> *i)
{     if( i->right ) return;
      BNode<T> *t=new BNode<T>(mydata);
      assert( t!=0);
      i->right=t;
}

template<class T>
void BTree<T>:: Inorder( )
{
      Inorder(root);
}

template<class T>
void BTree<T>:: Inorder(const BNode<T> *i)
{
      if( !i ) return;
      Inorder(i->left);
      cout<<i->data;
      Inorder(i->right);
}

template<class T>
void BTree<T>:: Preorder( )
{
      Preorder(root);
}

template<class T>
void BTree<T>:: Preorder(const BNode<T> *i)
{
      if( !i ) return;
      cout<<i->data;
      Preorder(i->left);
      Preorder(i->right);
}

template<class T>
void BTree<T>:: Postorder( )
{
      Postorder(root);
}

template<class T>
void BTree<T>:: Postorder(const BNode<T> *i)
{
```

```
	if( !i ) return;
	Postorder(i->left);
	Postorder(i->right);
	cout<<i->data;
}

template<class T>
void BTree<T>:: Levelorder( )
{
	BNode<T> *i=root;
	if( !i ) return;
	Queue<BNode<T>*> Q;
	while( i )
	{
		cout<<i->data;
		if(i->left)
			Q.Add(i->left);
		if(i->right)
			Q.Add(i->right);
		if(Q.ISEmpty( ))
			i=0;
		else
		{
			i=Q.Front( );
			Q.Delete( );
		}
	}
}

template <class T>
int BTree<T>:: Level(const BNode<T> *i)
{
	BNode <T> *j=i;
	if( !i ) return 0;
	int c=0;
	while(j)
	{
		c++;
		j=Parent( j );
	}
}

template<class T>
int BTree<T>:: Height( )
{
	int max=1;
	BNode<T> *i=root;
	if( !i ) return 0;
	Queue<BNode<T>*> Q;
```

```
    while( i )
    {
        if( Level(i)>max )
            max=Level(i);
        if(i->left)
            Q.Add(i->left);
        if(i->right)
            Q.Add(i->right);
        if(Q.ISEmpty( ))
            i=0;
        else
        {
            i=Q.Front( );
            Q.Delete( );
        }
    }
    return max;
}

template<class T>
const Bnode<T> *BTree<T>:: Parent(const BNode<T> *j)
{
    BNode<T> *i=root;
    if( !i ) return 0;
    if( !j ) return 0;
    if (j = =root) return 0;
    Queue<BNode<T>*> Q;
    while( i )
    {
        if(i->left = = j || i->right = = j)
            return i;
        if(i->left)
            Q.Add(i->left);
        if(i->right)
            Q.Add(i->right);
        if(Q.ISEmpty( ))
            i=0;
        else
        {
            i=Q.Front( );
            Q.Delete( );
        }
    }
    return 0;
}
```

مثال: درخت دودويي را غيربازگشتي بصورت INORDER پيمايش كـرده و نودهـاي ديـده شده را چاپ كنيد.

حل: تمامي توابع بازگشتي به كمك پشته به صورت تكراري نوشته مي‌شوند.

```
template<class T>
void BTree<T>:: nonrecInorder( )
{
    Stack<BNode<T> *> S;
    BNode<T> *i=root;
    if( !i ) return;

    while(1)
    {
        while(i)
        {
            S.PUSH(i);
            i=i->left;
        }
        if( ! S.ISEmpty( ) )
        {
            i=S.TOP( );
            S.POP( );
            cout << i->data;
            i=i->right;
        }
        else
            break;
    }
}
```

مثال: سازندة كپي را براي درخت دودويي بنويسيد.

```
template<class T>
BTree<T>:: BTree( const BTree<T> &b)
{
    this->root = copy(b.root);
}

template<class T>
const BNode<T> *BTree<T>:: copy(BNode<T> *j)
{
    if( j )
    {
        BNode<T> *newnode= new BNode<T>(j->data);
        newnode->left=copy(j->left);
        newnode->right=copy(j->right);
        return newnode;
    }
    else
        return 0;
}
```

تمرين: ۱- تابعي براي محاسبة تعداد برگ‌هاي درخت دودويي بنويسيد.

۲- تابعي براي محاسبة تعداد تكرار يك داده در درخت دودويي بنويسيد.

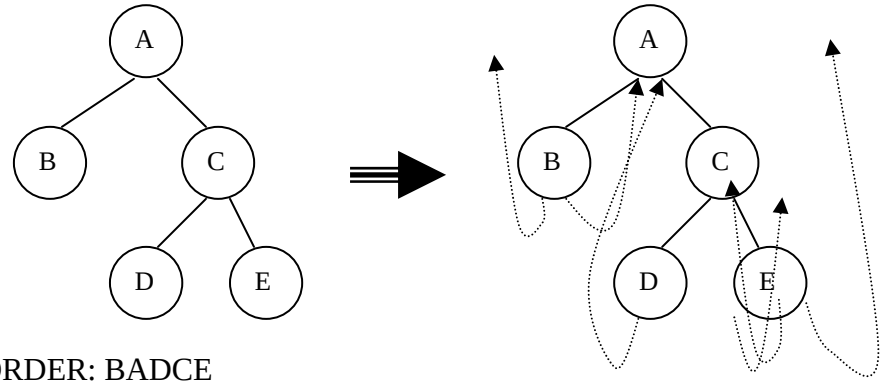
۷-۲- درخت دودويي نخ كشي شده **Threaded Binary Tree**

اگر به ساختار درخت دودويي دقت كنيم، متوجه مي‌شويم كه اگر N نود در درخت وجود داشته با شد، تعداد كل اشاره گرها 2N است، و تعداد اشاره گرهاي N+1،NULL است. براي آنكه پيمايش درخت با پيچيدگي كمتري قابل نجام باشد مي‌توان از اشاره گرهاي NULL استفاده كرد و نودهاي قبل و بعد از هر نود را در پيمايش درخت مشخص نمود. اين درخت، درخت نخ كشي شده نام دارد.

تعريف نخ چپ: اشاره گر فرزند چپ است (Left) كه قبلاً صفر بوده، و با اشاره گري به عنصر ماقبل در پيمايش INORDER جايگزين شده است.

تعريف نخ راست: اشاره گر فرزند راست است (Right) كه قبلاً صفر بوده، و با اشاره گري به عنصر بعدي در پيمايش INORDER جايگزين شده است.

مثال: درخت دودويي و نخ كشي شدة آن:



INORDER: BADCE

نخ‌ها بصورت نقطه چين نمايش داده شده و نخ‌هايي كه به هيچ نودي اشاره نمي كنند مربوط به اولين و آخرين نود در پيمايش مي‌باشند.

براي مشخص كردن تفاوت نخ و ارتباط (LINK)، از يك بيت نشانه (TAG) استفاده مي‌كنيم.

ساختار هر نود در درخت دودويي نخ كشي شده بصورت زير است.

LT	Left	Data	Right	RT

LT و RT: BOOLEAN هستند و نشان دهندة نخ يا ارتباط بودن اشاره گرهاي LEFT و يا Right هستند.

```
template<class T>
class TNode
{
    friend class TTree;
    public:
        TNode(T mydata)
        {   LT=RT=FALSE;
            Left=Right=0;
            data=mydata;
        }
        ~TNode( ){ }
    private:
        bool LT,RT;
        T data;
        TNode<T> *Left,*Right;
};
```

اشاره‌گر Left عنصر اول در پیمایش INORDER، NULL و LT آن TRUE است.

اشاره‌گر Right عنصر اول در پیمایش INORDER، NULL و RT آن TRUE است.

```
template<class T>
class TTree
{
    public:
        TTree( ) {root=0;}
        ~TTree( );
        void Inorder( );
        void InsertRightChild(TNode<T> *,T);
    private:
        TNode<T> *root;
        TNode<T> *Next(TNode<T> *current);// بدست آوردن نود بعدي
};
```

پیچیدگي پیمایش درخت، در درخت نخ کشي شده از درجة O(n) است، که از روش بازگشتي و غیر بازگشتي آن در درخت نخ کشي نشده، کمتر است.

```
template<class T>
TNode<T> *TTree<T>:: Next(TNode<T> *current)
{
    if( !current ) return 0;
    TNode<T> *temp=current->Right;
    if( !current->RT )// اگر سمت راست ارتباط نیست
    {
        while( ! temp->LT)
            temp=temp->Left;
        return temp;
    }
    else return current->Right;
}
```

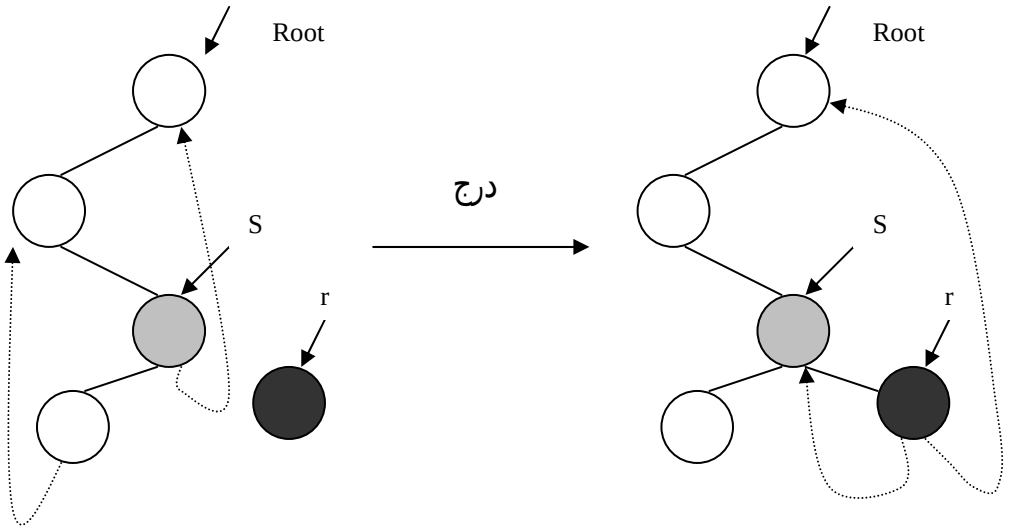
```
template<class T>
void  TTree<T>:: Inorder( )
{
     TNode<T> *i=root;
     if( !i ) return;
     while( ! i->LT )
          i=i->Left;
     cout << i->data;
     while(i)
     {
          i=Next(i);
          cout << i->data;
     }
}
```

درج نود سمت راست، به يك نود و تنظيم مجدد نخ‌ها:

براي بدست اوردن الگوريتم دو حالت را در نظر مي‌گيريم:

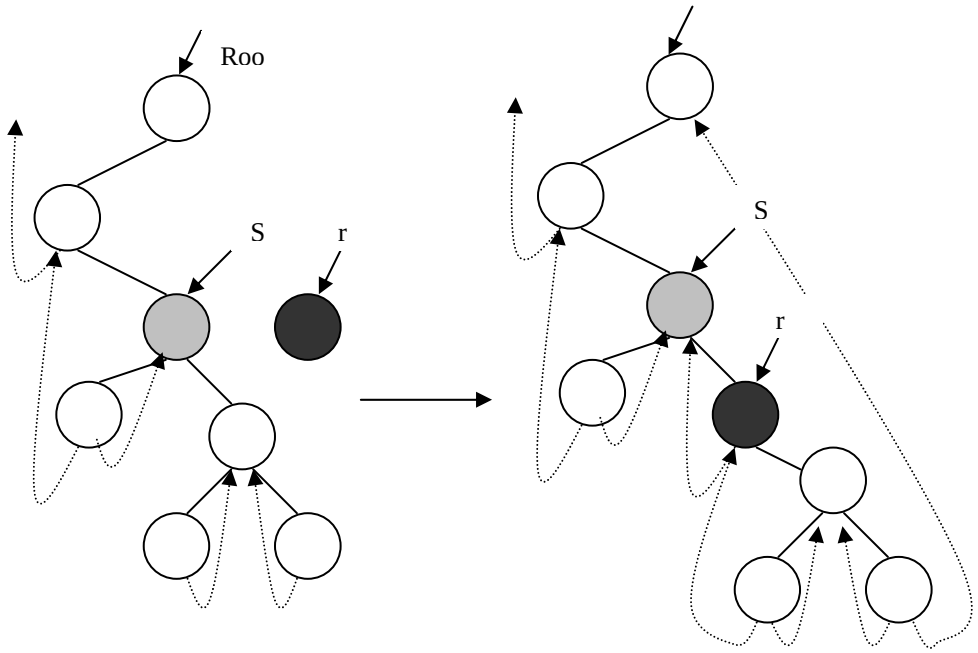
الف) سمت راست، يك نخ است.

در اين حالت عمليات بسيار ساده است، و در شكل زير نشان داده شده است.



ب) سمت راست، ارتباط، باشد (نخ نباشد)

زير درخت سمت راست، بايد زير درخت نود جديد، شود و نخ‌ها نيز بايستي بـه روز در آيد. در اين حالت نود r عضو قبلي در پيمايش INORDER براي زير درخت است، بنـابراين نخها به روز رساني مي‌شود و به r اشاره خواهد كرد.




```
template<class T>
void TTree<T>:: InsertRightChild(TNode<T> *s ,T mydata)
{
    TNode<T> *r = new TNode<T>(mydata);
    assert( r!=0 );
    r->Right = s->Right;
    r->RT = s->RT;
    r->Left=s;
    r->LT=TRUE; // سمت چپ نخ است
    s->Right = r; //اتصال به نود
    s->RT=FALSE;
    if( !r->RT) // اگرسمت راست نخ نيست
    {
        TNode<T> *temp=Next( r );
        temp->Left = r;
    }
}
```

تمرين:

١- تابعي براي حذف يك نود از درخت نخ كشي شده بنويسيد.

٢- تابعي براي درج فرزند چپ به يك نود بنويسيد.

۳- پيمـايش POSTORDER و PREORDER و LEVEL ORDER را در درخـت نخ كشي شده بنويسيد.

۷-۳- پياده سازي درخت دودويي با كمك آرايه

Array Representation of Binary Tree

يكي از روش‌هاي پياده سازي درخت دودويي، پياده سازي با آرايه است. اين پياده سازي بسيار ساده است و نسبت به پياده سازي با اشاره گرها به فـضاي بيـشتري نيـاز دارد و نيـز در برخي موارد (مثل حذف و درج بين دو نود) نياز به شيفت عناصر است.

در اين حالت جنس هر خانه آرايه از جنس DataType است. بـراي آنكـه پيـاده سـازي راحت باشد، از خانه اول (انديس صفر) استفاده نمي كنيم.

ريشه هميشه در خانه با انديس يك است.

فرزند سمت چپ نودي كه در خانه i ام آرايـه اسـت، در صـورت وجـود در خانـة 2*i است.

فرزند سمت راست نودي كه در خانه i ام آرايه است، در صورت وجود در خانـة 2*i+1 است.

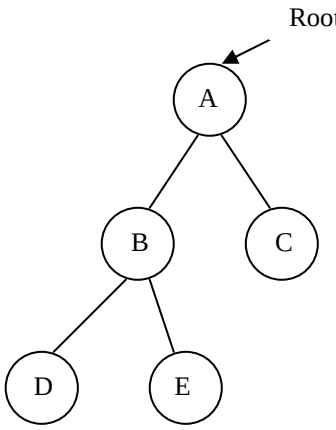
والد يا Parent نودي كه در خانة i ام است، در خانة [i/2] (جزء صحيح i/2) قرار دارد.

```
template <class T>
class ArrayBtree
{
    public:
        ArrayBtree(int Maxsize=100);
        ~ ArrayBtree( ){ delete [] array;}
        void Inorder( );
        void Postorder( );
        void Preorder( );
        void Levelorder( );
        void InsertLeftChild(int node, T mydata);
        void InsertRightChild(int node, T mydata);
        void AddRoot(T mydata){array[1]=mydata;}
        int leftchild(int i){if 2*i<max return 2*i; else return 0;}
        int rightchild(int i){ if 2*i+1<max return 2*i+1; else return 0;}
        int parent(int i){return i/2;}
    private:
        T *array;
        int max;
};
```

```
template <class T>
ArrayBtree<T>:: ArrayBtree(int maxsize)
{
    max=maxsize;
    array=new T[max];
    assert( array!=0);
}
```

مثال: پیاده سازي درخت دودويي با آرايه

Unused	A	B	C	D	E	0	0	0
0	1	2	3	4	5	6	7	8




```
template <class T>
void ArrayBtree<T>:: InsertLeftChild(int i, T mydata)
{
    if(i<0 || 2*i>max) return;
    array[2*i]=mydata;
}
template <class T>
void ArrayBtree<T>:: InsertRightChild(int i, T mydata)
{
    if(i<0 || 2*+1i>max) return;
    array[2*i+1]=mydata;
}
template <class T>
void ArrayBtree<T>:: Inorder( )
{
    Inorder(1);//شروع پیمایش از ریشه
}
```

```
template <class T>
void ArrayBtree<T>:: Inorder(int i)
{
     if( i == 0 || i>=max || array[i] == T:: EMPTYHOME( ) )
          return;
     inorder(leftchild(i));
     cout << array[i];
     inorder(rightchild(i));
}

template <class T>
void ArrayBtree<T>:: Preorder( )
{
     Preorder(1);//شروع پیمایش از ریشه
}

template <class T>
void ArrayBtree<T>:: Preorder(int i)
{
     if( i == 0 || i>=max || array[i] == T:: EMPTYHOME( ) )
          return;
     cout << array[i];
     inorder(leftchild(i));
     inorder(rightchild(i));
}

template <class T>
void ArrayBtree<T>:: Postorder( )
{
     Postorder(1);//شروع پیمایش از ریشه
}
template <class T>
void ArrayBtree<T>:: Postorder(int i)
{
     if( i == 0 || i>=max || array[i] == T:: EMPTYHOME( ) )
          return;
     inorder(leftchild(i));
     inorder(rightchild(i));
     cout << array[i];
}
template <class T>
void ArrayBtree<T>:: Levelorder( )
{
     Queue<int> Q;
          int i=1;
     while( i < max && i>0)
     {
```

```
            cout << array[i];
            if( leftchild(i) && array[leftchild(i)] != T:: EMPTYHOME( ) )
                Q.Add(leftchild(i));
            if( rightchild(i) && array[rightchild(i)] != T:: EMPTYHOME( ) )
                Q.Add(rightchild(i));
            if( !Q.ISEmpty( ) )
            {
                i=Q.Front( );
                Q.Delete( );
            }
            else
                i=0;
        }

}
```

تمرين:

۱- پيمايش غير بازگشتي inorder را براي درخت دودويي با آرايه بنويسيد.

۲- درخت نخ كشي شده را با آرايه پياده سازي كنيد.

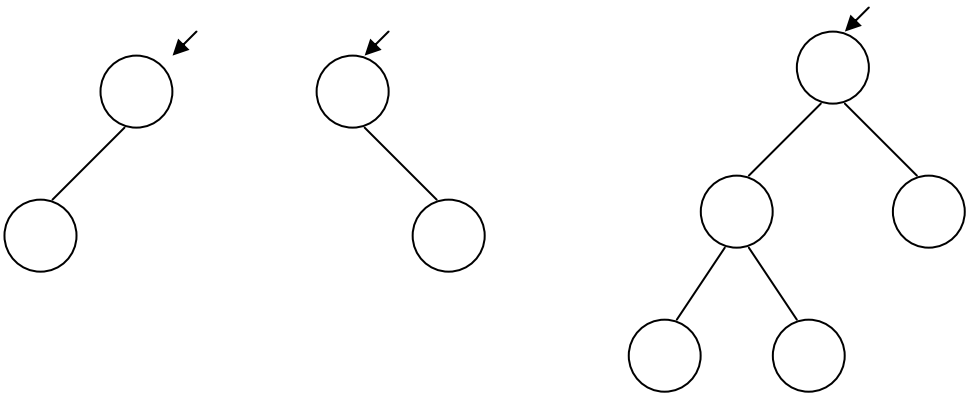
۳- تابع بدست آوردن ارتفاع درخت دودويي را بنويسيد.

٤- تابعي بنويسيد كه تعداد برگهاي درخت دودويي را چاپ كند.

درخت دودويي كامل **(Complete Binary Tree)**

يك درخت دودويي است كه قدرمطلق تفاضل ارتفاع زير درخت چپ و راست براي همة نودها، صفر يا يك است.(نداشتن فرزند در سطح يكي مانده به آخر مجاز است)

مثال: درخت دودويي كامل



درخت دودويي پر (Fully Complete Binary Tree)

يك درخت دودويي است كه نودها يا فرزندي ندارند، يا دو فرزند دارند. تعداد نودهـا در اين درخت، اگر ارتفاع برابر با l باشد 2^l-1 است و برعكس (شرط لازم و كافي).

۷-٤- ساختار Heap (درخت Heap)

اين ساختار در حقيقت بهترين پياده سازي بـراي Priority Queue يـا صـف اولويـت است. كه در دو حالت max – Heap ويا min- Heap پياده سازي مي‌شود.

تعريف Heap

يك درخت دودويي كامل يا (complete) است كه ريشه از تمامي نودهاي دو زير درخت چپ و راست خود كمتر نيست و تمامي نودهاي درخت، شرط فوق را دارند (max- Heap)

يك درخت دودويي complete است كه ريشه از تمامي نودهاي دو زير درخت خود بيشتر نيست و تمامي نودهاي درخت، شرط فوق را دارند (min- Heap)

در درخت (max- Heap)، ريشه هميشه بيشترين كليد درخت را داراست.

در درخت (min- Heap)، ريشه هميشه كمترين كليد درخت را داراست.

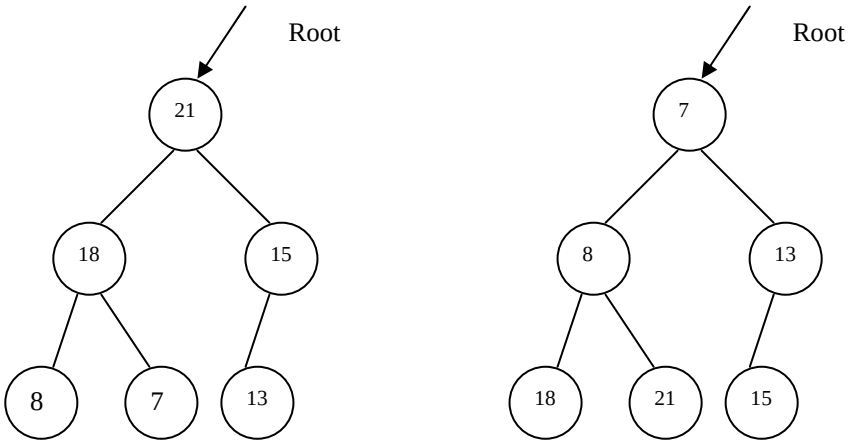
تعريف **ADT** براي **Heap** :

اين ساختار يك صف اولويت است.

- درج نود با توجه به اولويت در محل خودش انجام مي‌شود. Insert (درج در مجموعه)
- حذف نود براي ماكزيمم اولويت (max- Heap) ويا مينيمم اولويت (min-Heap) قابـل انجام است و هميشه ريشه درخت حذف مي‌شود. Delete (حذف از مجموعه)
- عملگر isempty كه مشخص مي‌كند Heap ، تهي است يا خير ؟

مثال: درخت min – Heap , max – Heap

در اين مثال فقط كليدها نمايش داده شده



MAX HEAP MIN HEAP

به دليل اينكه ساختار كامل (complete) است بنابراين درج در محل خاصي انجام خواهد شد وسپس كليدهاي درخت مرتب خواهند شد

```
template <class  datatype >
class  node
{        friend  class  Heap;
     public:
            node (int   pr = 0);
            ~ node ( ){ }
     private:
             datatype  data;
             int   priority;
             bool    isempty;
};
template < class  data type >
node <data type >:: node (data type  d1 , int  pr)
{        isempty = true;
         priority = pr;
}
```

در اين حالت ساختار هر نود درخت به صورت زير است و با آرايه پياده سازي مي‌شود

خالي بودن Isempty	اولويت priority
data	داده

```
template <class data type >
class  Heap
{        public:
                Heap (unsigned  m= 100);

           ~ Heap ( ) { delete [] Array;}
         void Insert ( data type d ,int pr);
```

```
        data type Delete ( );
        bool  isempty ( );
    private:
        unsigned   max;
        node <data type>    * Array ;
      int  lastnode; };
```

سازنده کلاس

```
template <class  datatype>
Heap <datatype >:: Heap(int  m)
{     max = m ;
       Array = new   node <datatype > (max);
       assert  (Array ; = 0);
       lastnode = 0;
}
```

الگوريتم درج: هميشه بايد بصورتي انجام شود كه

۱- درخت complete باقي بماند

۲- درخت در حالت Heap باقي بماند

متغير lastnode همیشه محل درج نود بعد را مشخص مي‌كند، نود را در درخت، به صورت درخت كامل درج مي‌كنيم و سپس آن درخت را به حالت Heap تبديل مي‌كنيم. اين برنامه براي حالت max Heap نوشته شده است. براي تبديل درخت به حالت Heap، نود درج شده با پدرش مقايسه مي‌شود اگر اولويت بيشتر باشد، با نود پدرش تعويض مي‌شود.

```
 template <class  datatype>
 void Heap<data type>:: Insert (datatype  d , int  pr)
{
lastnode ++ ;
        Array [last node]. isempty = false;
Array [last node ]. Data = d ;
Array [last node ]. priority = pr;
int  i= lastnode ;
while (i ! = 0) && (i != 1))
{
                if (Array[i].priority >Array [i/2].priority)
                        swap(Array[i] , Array [i/2]);
                i= i/2 ;
              }
       }

template <class  datatype>
bool   Heap <data type >:: isempty ( )
    {        return  Array[1].isempty ;   }
```

حذف نود

همیشه نود با بالاترین اولویت (max Heap) یا با کمترین اولویت (min- Heap)، حذف مي‌شود (ریشه درخت) پس همیشه باید ریشه را حذف کنیم، در این حالت آخرین نود در level آخر، جایگزین ریشه مي‌شود و درخت دوباره به حالت Heap تبدیل مي‌گردد. (يعني تا رسيدن به level آخر اگر اولويت ريشه از يكي از فرزندانش كوچكتر است، با ماكزيمم آنها عوض مي‌شود)

```
template <class  datatype>
datatype Heap<datatype>:: Delete( )
{   datatype temp = Array[1].data;
swap (Array [1],Arry [last node];
Array [last node --].isempty = true;
int  i= 1 ;
while ( i<= last node   &.& (Array [i]. Priority <
max (Array [2*i],Array [2*i+1])
{          int  i = max indexchild(i) ;
swap(Array [I],Array [j] ) ;
 i = j ;
}
return    temp ;
}
```

تمرين: تابعي براي تبديل MAX-HEAP به MIN-HEAP بنويسيد.

٧-٥- درخت دودويي جستجو **Binary Search Tree**

در روش‌هاي جستجوي اطلاعات، ساده ترين روش جستجوي خطي است. كه در اين روش اطلاعات به ترتيب جستجو مي‌شود و حداكثر پيچيدگي الگوريتم جستجو O(N) است. جستجوي دودويي در مورد اطلاعاتي كه بر اساس كليد مرتب باشند قابل استفاده است و بسيار سريع جستجو را انجام مي‌دهد و حد اكثر تعداد مقايسه Log n است.

تعريف كليد جستجو: معيار مقايسه در جستجو و مرتب‌سازي اطلاعات است.

تعريف درخت دودويي جستجو: درخت دودويي جستجو يك درخت دودويي است كه نودي دارد بنام ريشه، كه كليد واقع در ريشه از تمامي كليدهاي واقع در شاخة چپ بيشتر واز تمامي كليدهاي شاخة راست كمتر است. و زير شاخة چپ و راست (فرزند چپ و راست) خود درخت دودويي جستجو هستند.

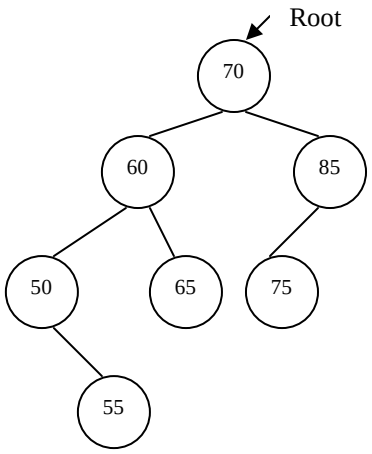
ساختار درخت دودويي در حقيقت يك مجموعه با امكان جستجوي سريع اطلاعات را در اختيار ما قرار مي‌دهد.

تعريف **ADT**

```
template <class  T>
class  binarysearchtree
{
     public:
binarysearchtree( );
     ~ binarysearchtree( );
void  insert( T  mydata , ink  key) ;// درج نود در مجموعه
void  Delete(ink  key) ;// حذف كليد دلخواه از مجموعه
Bsnode<T>  *search(ink  key) ;// جسنجوي كليد دلخواه
     private:
Bsnode<T>   *Root ;// ريشة درخت

};
template <class   T>
class  Bsnode
{     friend  class   binarysearchtree<T>;
       public:
Bsnode (T  d , ink   k)
{     data = d ;
key  =k ;
}
~ Bsnde ( ) { }
private:
ink  key ;
T  data ; };
```

مثالي از يك درخت دودويي جستجو (در اين مثال فقط كليد يا نشان داده شده)



الگوريتم جستجوي دودويي درخت دودويي جستجو:

جستجو را از ريشه درخت شروع كنيد. اگر كليد برابر ريشه است، ريـشه جـواب اسـت. اگر كليد از ريشه كمتر است زير شاخه چپ را جستجوي دودويي كنيـد. اگـر كليـد از ريـشه بيشتر است زير شاخة راست را جستجوي دودويي كنيد.

```
template <class  T>
Bsnode<T>   * binarysearchtree <T>:: search(ink  key1)
{
      Bsnode<T>   *I = Root ;
       if  (! I)  return  0 ;                          درخت تهي را نمي‌توان جستجو كرد
      while ( I)
{
if ( I -> key = = key1)
            return  I ;
               else  if (I -> key >key 1)              جواب بايد در شاخة چپ باشد
                   I = I -> left ;
                         else   //          جواب در شاخة راست است
                         I = I -> Right ;
}
              return   0 ;  .پيدا نشده است
}
```

الگوريتم فوق بصورت باز گشتي نيز نوشته مي‌شود.

تمرين:

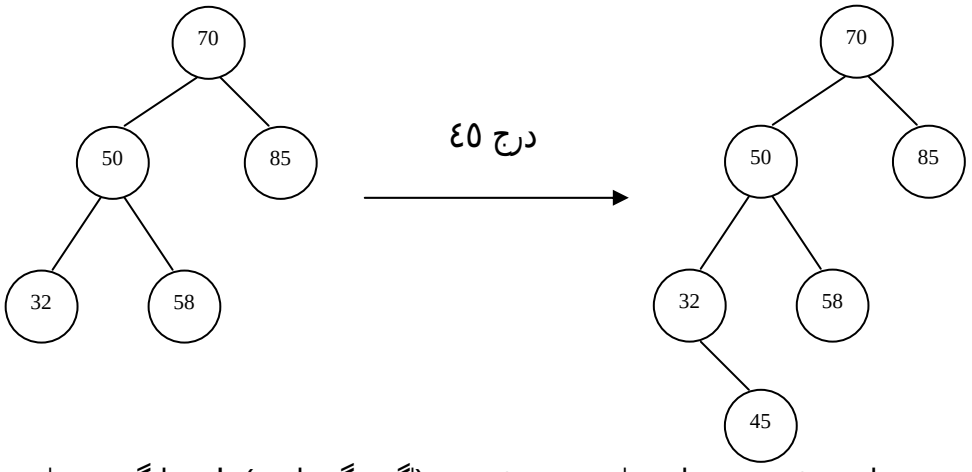
الگوريتم جستجو را بصورت باز گشتي بنويسيد.

درج نود در درخت دودويي جستجو:

براي درج نود – ابتدا بايد محل درج نود را بيابيم و آنرا درج كنيم در زمان درج درخت بايد BST باقي بماند.

```
template <class  T>
void  binarysearchtree<T>:: insert (t  data , ink  key1)
{     Bsnode <T>  *i = Root , j = i ;
while (i)
{     if ( i -> key = = key 1)  // exists
            return;
else  if ( i -> key < key1 )
{     j = i ;
i = i -> Right ;
}
else
{     j= i ;
i = i -> left ;
```

```
}
}
Bsnode <T>  * temp = new  Bsnode <T> ( data , key1);
assert  ( temp ! = 0);
if ( j -> key > key 1)
j -> left = temp ;
else
j -> Right = temp ;
}
```



در زمان حذف نود نيز بايد براي نود حذف شده (اگر برگ نباشد) يك جايگزين پيدا شود. كه معمولاً از بزرگترين نود شاخة سمت چپ يا كوچكترين نود شاخه راست استفاده مي‌شود.

هزينه حذف در بدترين حالت O (n log n) است.

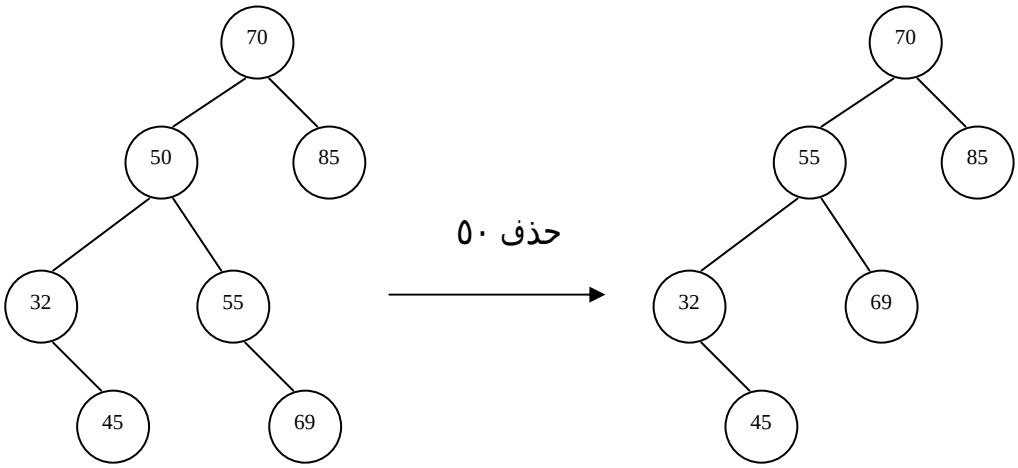
تمرين:

١- عملگر >> را سربار گذاري كنيد كه درخت دودويي را بصورت صعودي براساس كليد چاپ كند.

٢- تابع min را براي يافتن كليد مينيمم در درخت دودويي جستجو سربار گذاري كنيد.

٣- تابع max را براي يافتن كليد ماكزيمم در درخت دودويي جستجو سربار گذاري كنيد.

٤- كلاس BST را با كمك درخت دودويي بكمك آرايه بنويسيد.



بهترين روش درج كوچكترين فرزند زير شاخة راست و يا بزرگترين فرزند زير شاخة چپ بجاي نودي است كه مي‌خواهد حذف شود.

تمرين: تابع Delete را براي حذف نود در درخت دودويي جستجو بنويسيد.

فصل هشتم

گراف

تعريف گراف: گراف G از دو مجموعه تشكيل شده است، مجموعة V شامل تعدادي گره (لبه يا نود)، اين مجموعه متناهي و غيرتهي است. و مجموعة E شامل تعـدادي يـال (اتـصال) كـه بصورت زوج مرتب نشان داده مي‌شود.

انواع گراف:

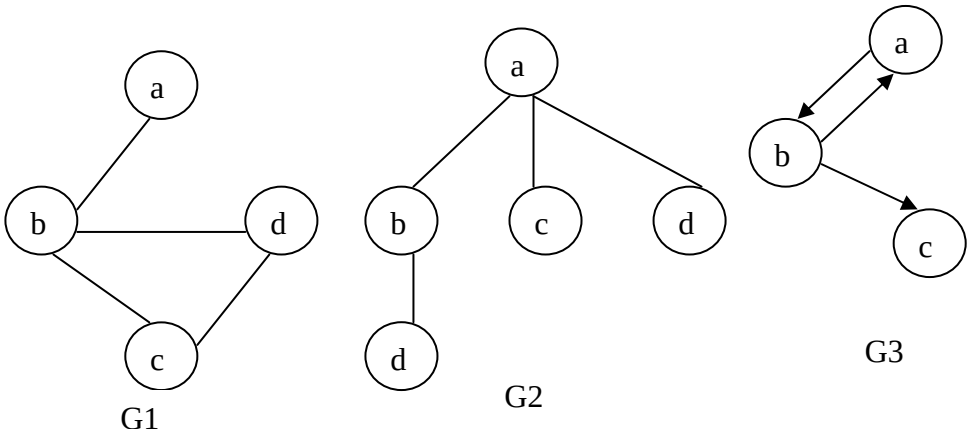
۱-گراف جهت دار (Directed Graph): در اين گراف يـال‌هـا داراي جهـت هـستند و جهت حركت در هر يال مشخص است.

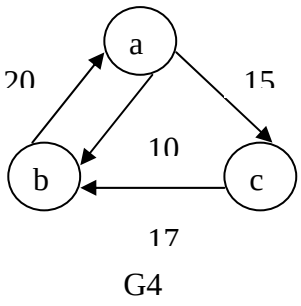
۲-گراف بدون جهت (Undirected Graph): در اين گراف يال‌ها، داراي جهت نيستند.

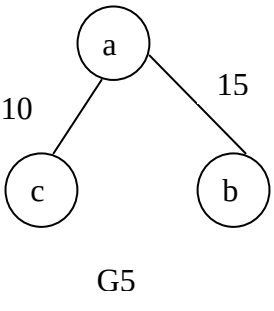
۳-گراف وزن دار: گرافي است كه هر يال، داراي يك عـدد وزن يـا هزينـه اسـت وزنهـا ممكن است مثبت و يا منفي باشند و در گراف جهت دار در هـر دو جهـت عـدد وزن وجـود دارد.

٤-گراف بدون وزن: براي حركت در يال‌ها وزن مشخص نمي‌گردد.

مثال: نمايش انواع گراف







G1: گراف بدون جهت - بدون وزن

G2: گراف بدون جهت - بدون وزن

G3: گراف جهت دار - بدون وزن G4: گراف جهت دار - وزن دار
G5: گراف بدون جهت - وزن دار

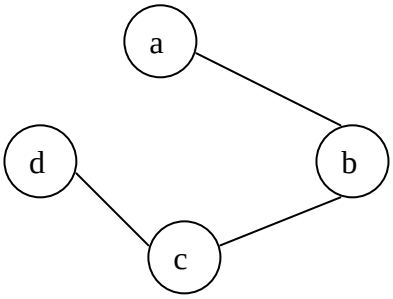
تعريف گراف پيوسته (Connected)

گراف G، پيوسته است، اگر از هر نود شروع كنيم، بتوان به تمام نودهـاي ديگـر حركـت كرد. (دسترسي به همة نودهاي ديگر وجود داشته باشد)

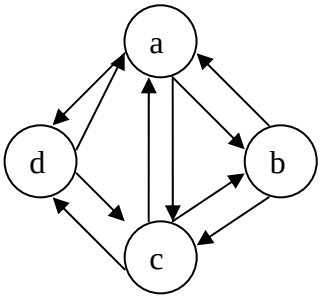
تعريف درجة يك گره: تعداد يال‌هايي است كه به آن نود متصل شده است، براي گـراف جهت دار درجة ورودي و خروجي براي تعداد يال‌هايي كه وارد يا خـارج مـي‌شـوند تعريـف مي‌شود.

مثال: درگراف G4 درجة ورودي نود b، برابر ۲ است و درجة خروجي آن برابر ۱ است.

مثال: گراف پيوسته



G1



G2

G1: گراف پيوستة بدون جهت
G2: گراف پيوستة جهت دار

تعريف ADT براي گراف

```
class Graph
{
    public:
        Graph( );
        ~Graph( );
        void InsertNode(char v);//درج نود
        void InsertLink(char a, char b, int w=1);//درج يال
        void DeleteNode(char v);//حذف نود
        char *Adjaycent(char a);//همسايه هاي يك نود
        int Link(char a, char b);
        int **path( );//ماتريس مسيري
```

```
        private:
            int **adj;
            int NodeNo;
            char *data;
}
Graph:: Graph( ) { NodeNo=0;  data=0;  adj=0; }

void Graph:: InsertNode(char a)
{//این تابع ماتریس همسایگی و آرایه نودهای گراف را اضافه می‌کند
    NodeNo++;
    char *temp= new char[NodeNo+1];
    assert(temp!=0);
    strcpy(temp,data);
    temp[NodeNo-1]=a;
    delete [ ] data;
    data = temp;
    int **adjnew = new (int *)[NodeNo];
    assert(adjnew);
    for(int i=0;i<NodeNo;i++)
    {
        adjnew[i]=new int[NodeNo];
        assert(adjnew[i]!=0);
    }
    for(int i=0;i<NodeNo-1;i++)
        for(int j=0;j<NodeNo-1;j++)
            adjnew[i][j] = adj[i][j];
    delete [] adj;
    adj=adjnew;
}
void Graph:: InsertLink(char a, char b, int w)
{
    for(int i=0;i<NodeNo;i++)
        if(data[i] == a)    break;
    for(int j=0;j<NodeNo;j++)
        if(data[j] == a)    break;
    if(i == NodeNo || j==NodeNo)  return; //خطای یافتن نود
    adj[i][j]=w;//درج یال
}

int Graph:: Link(char a, char b)
{
    for(int i=0;i<NodeNo;i++)
        if(data[i] == a)    break;
    for(int j=0;j<NodeNo;j++)
        if(data[j] == a)    break;
    if(i == NodeNo || j==NodeNo)  return 0; //خطای یافتن نود
    return adj[i][j];
```

```
}
int **Graph:: pathN(int N) // ماتريس مسيرهايي به طول داده شده
```

براي بدست اوردن اين كار ماتريس همسايگي را به توان N ميرسانيم.

```
{
    int **p=new (int *)[NodeNo];
    assert(p!=0);
    for(int i=0;i<NodeNo;i++)
    {
        p[i]=new int[NodeNo];
        assert(p[i]!=0);
    }
    for(int i=0;i<NodeNo;i++)
        for(int j=0;j<NodeNo;j++)
            p[i][j] = adj[i][j];

    for(int i=0;i<N;i++)
        for(int j=0;j<NodeNo;j++)
            for(int l=0;l<NodeNo;l++)
                for(int k=0;k<NodeNo;k++)
                    p[j][k] += p[j][k]*adj[j][l]
    return p;
}
int **Graph:: path( )
{
    int **p=new (int *)[NodeNo];
    assert(p!=0);
    for(int i=0;i<NodeNo;i++)
    {
        p[i]=new int[NodeNo];
        assert(p[i]!=0);
    }
    for(int i=1;i<NodeNo-1;i++)
    {
        int **temp=pathN(i);
        for(int k=0;k<NodeNo;k++)
            for(int l=0;l<NodeNo;l++)
                if(temp[k][l] !=0)
                    p[k][l]=1;
    }
    return p;
}
```

بايد مسيرهايي بطول يك الي N-1 را كنترل كنيم.

مثال: تابعي بنويسيد كه كنترل كند آيا گراف پيوسته است يا خير؟

بايد ماتريس مسيري را محاسبه كنيم و تمام درايه ها به غير از اعضاي قطر اصلي يك باشد.

```
bool Graph:: isconnected( )
{
     int **p=path( );
     bool isc=True;
     for(int i=0;i<NodeNo;i++)
          for(int j=0;j<NodeNo;j++)
               if(i!=j && p[i][j]==0)
                    return False;
     return isc;
}
```

پیمایش گراف:

پیمایش گراف به روش‌هاي زیر قابل انجام است، در این روش‌ها تمامي نودهـاي گـراف مشاهده خواهـد شـد و فقـط ترتیـب مـشاهده‌ي نودهـا متفـاوت اسـت. پیمـایش معمـولاً در گراف‌هاي پیوسته کاربرد دارد.

الف: پیمایش عمقي **(Depth First Search)**

پیمایش عمقي از نود v شروع مي‌شود (v، مشاهده مي‌گـردد)، سـپس همـسایه پیمـایش نشدهٔ w که همسایهٔ v است، پیمایش عمقي مي‌گردد، این الگوریتم بازگشتي است.

```
void Graph:: DFS( )
{
     bool *visited = new bool[NodeNo];
     for(int i=0;i<NodeNo;i++)
          visited[i]=False;
     DFS(0,visited);
     delete [ ] visited;
}
void Graph:: DFS(int v, bool * visited)
{    visited[v]=True;
     cout << data[i];//visit the node
     for(int i=0;i<NodeNo;i++)
          if(!visited[j] && adjmat[i][v] !=0)
                    DFS(i);
}
```

کارآیي الگوریتم: پیمایش تمامي نودهاي همسایه، براي هر نود داراي پیچیـدگي $O(n)$، است. (n تعداد نودهاي گراف است)، پیمایش کل گراف از درجهٔ $O(n^2)$ است.

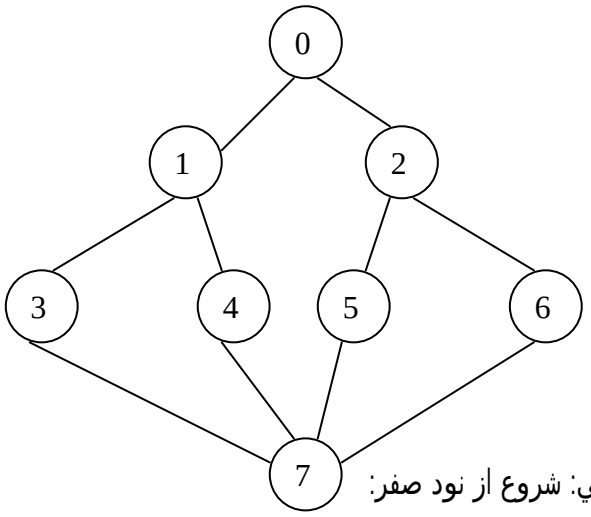
ب:پیمایش سطحي گراف **(Breath First Search)**

پیمایش از نود v شروع مي‌گردد (v،مشاهده مي‌گردد)، سـپس تمـام نودهـاي همـسایهٔ v پیمایش نشده‌اند، پیمایش مي‌گردند، و سپس همسایه هاي آنها نیز مشاهده مي‌شود.

```
void Graph:: BFS(int v)
{
    bool *visited = new bool[NodeNo];
    for(int i=0;i<NodeNo;i++)
        visited[i]=False;
    Queue<int> q;
    q.insert(v);
    while( !q.isempty( ) )
    {
        v=q.Front( );
        q.Delete( );
        for(int i=0;i<NodeNo;i++)
            if( adj[v][i] && !visited[i] )
            {
                q.Add(i);
                cout << data[i];
                visited[i]=True;
            }
    }
    delete [ ] visited;
}
```

کارآيي الگوريتم: پيمايش سطحي تمامي نودهاي همسايهٔ v، از درجهٔ O(n) است، و پيمايش کل گراف از درجهٔ $O(n^2)$ است.

مثال: پيمايش گراف



پيمايش عمقي: شروع از نود صفر:

Visited = 0,1,3,7,4,5,2,6

پيمايش سطحي گراف: شروع از نود صفر

Visited =0,1,2,3,4,5,6,7

۸-٤- درخت‌هاي پوشا **(Spanning Trees)**

اگر گراف G پيوسته (Connected) باشد، آنگاه پيمايش عمقي يا سطحي گـراف، از هـر نودي كه شروع شود، تمامي نودهاي گراف را مي‌پيمايد. در اين صورت يـال‌هـا بـه دو دسـتة (يال‌هاي تشكيل دهندة درخت) و (يال‌هاي غير درخت) تقسيم مي‌شود. كه يال‌هـاي درخـت، يال‌هايي است كه در زمان پيمايش (عمقي يا سطحي) ديده مي‌شوند.

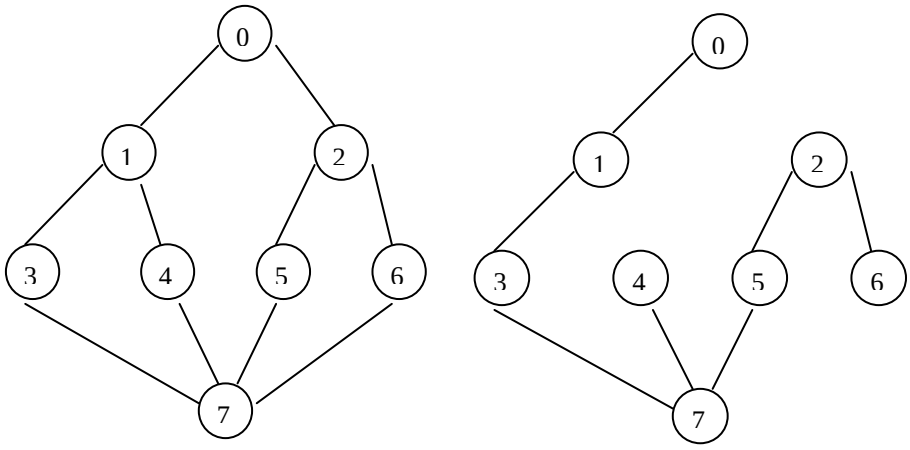
به عبارت ديگر هر زير گراف از G كه با كمك زير مجموعه اي از يال‌ها به تمامي نودهـا دستيابي داشته باشد (پيوسته باشد)، درخت پوشا ناميده مي‌شود.

مثال: شكل زير درختهاي پوشاي گراف G با چهار نود است.

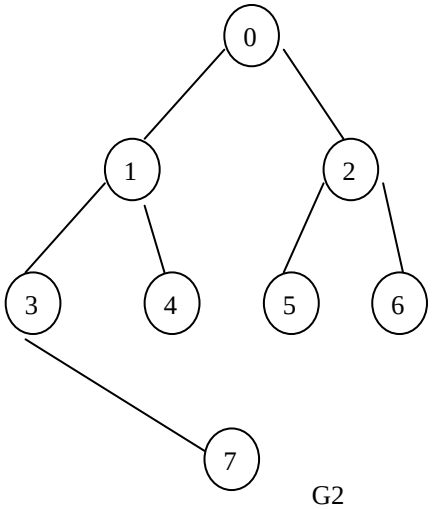
G G G G

درخت پوشايي كه از پيمايش عمقي گراف بدست آيد، درخت پوشاي عمقـي و درخـت پوشايي كه از پيمايش سطحي بدست آيد، درخت پوشاي سطحي ناميده مي‌شود.

مثال: گراف پيوسته و دو درخت پوشاي عمقي (G1) و سطحي (G2)



G G1



G2

نكته: براي جستجوي گراف، اگر گراف پيوسته باشد، ميتوان يكي از دو روش عمقي يا سطحي را بكار برد. در هر دو حالت، پيچيدگي الگوريتم در بدترين حالت $O(N^2)$ است.

۸-۵- يافتن كم هزينه‌ترين درخت پوشا **(Minimum Cost Spanning Tree)**

در كاربردهاي گراف (مثلاً در سيستم عامل يا در شبكه ها) يافتن كم هزينه ترين درخت پوشا بسيار كاربرد دارد. كم هزينه ترين درخت پوشا، كم هزينه‌ترين درخت، از ميان درخت‌هاي پوشاي ممكن يك گراف است. براي يافتن اين درخت الگوريتم هاي متنوعي وجود دارد.

تمامي اين الگوريتم ها بايد نكات ذيل را رعايت كنند:

۱- فقط از يال‌هاي موجود در گراف استفاده شود.

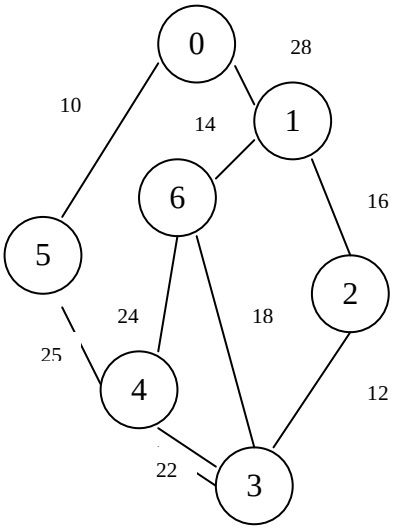
۲- گراف نتيجه بايد دقيقاً N-1 يال داشته باشد. (N تعداد نودهاي گراف است)

۳- از يال‌هايي كه ايجاد حلقه (Cycle) مي‌كنند، استفاده نمي‌شود.

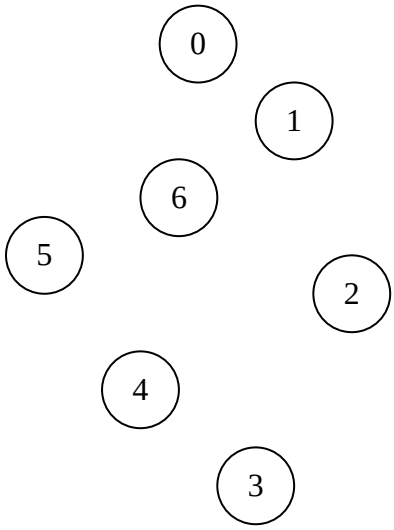
الگوريتم **Kruskal**

اين الگوريتم درخت پوشاي T را مي‌سازد و يال‌ها را از گراف اصلي به ترتيب غير نزولي هزينه انتخاب مي‌كند، هر يال كه به گراف نتيجه اضافه مي‌شود نبايد ايجاد حلقه نمايد (در اين صورت بايد كنار گذاشته شود) اين عمل آنقدر ادامه مي‌يابد تا دقيقاً N-1 يال انتخاب شود.

مثال: مراحل تشکیل درخت پوشا با کمترین هزینه طبق الگوریتم Kruskals



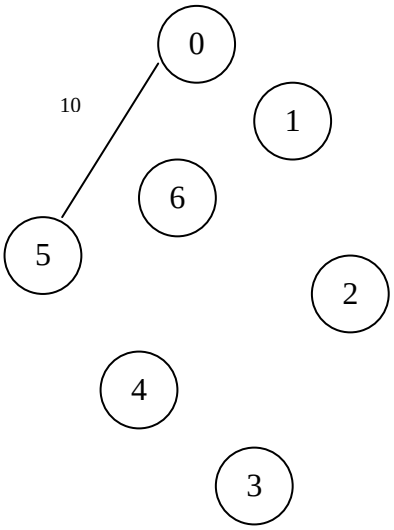
G



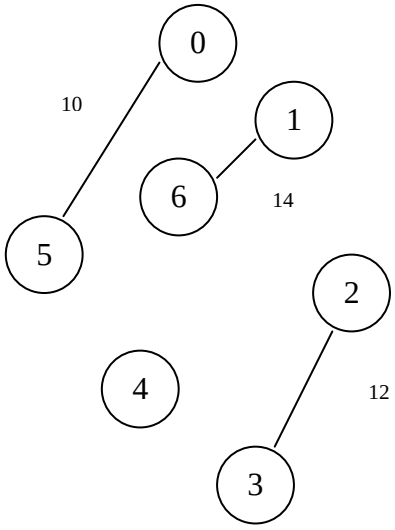
a

در مرحلة اول تمامي نودهاي موجود نوشته مي‌شود و يال‌ها به ترتيب غير نزولـي هزينـه در نظر گرفته مي‌شود.

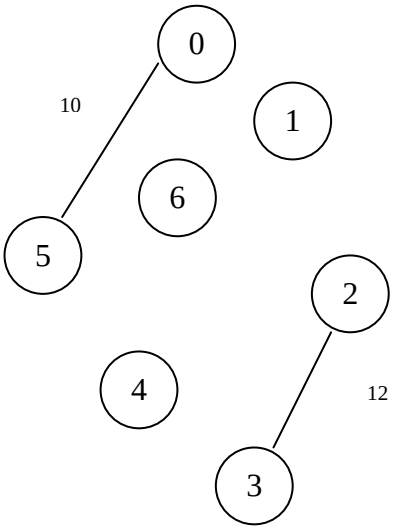
انتخاب اولين يال به ترتيب غير نزولي (مينيمم هزينه موجود در گراف G)



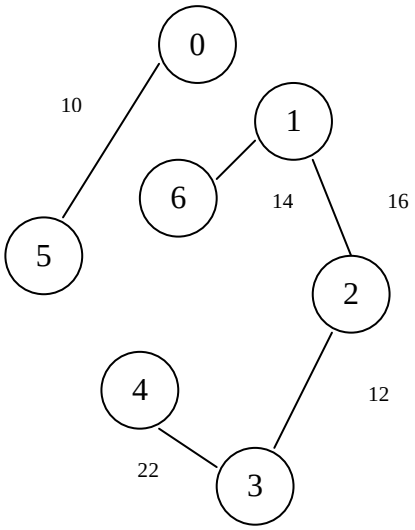
b



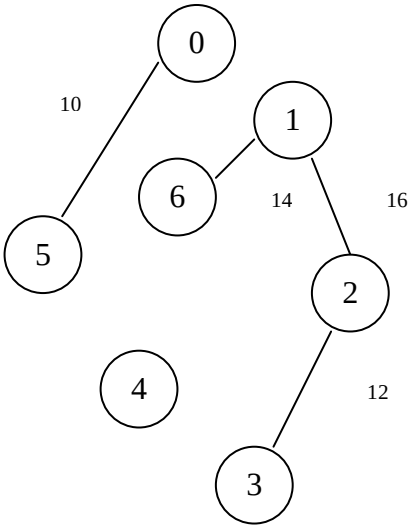
d



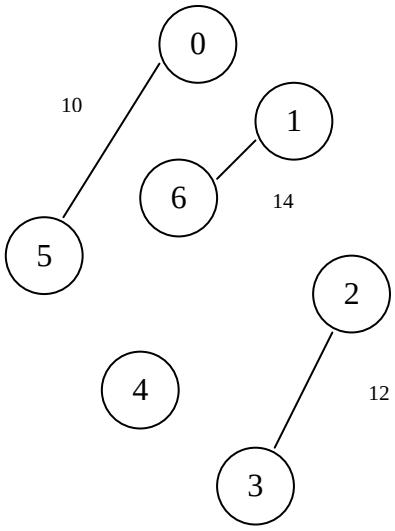
c



f

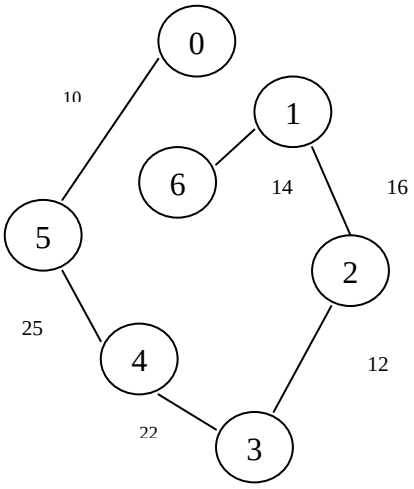


e



e

انتخاب ۱۸ در مرحلهٔ f ایجاد حلقه مي‌نمايد، پس از آن صرف نظـر شـده اسـت. در ايـن مرحله تعداد يال‌ها به N-1 رسـيده است، پـس پايـان الگـوريتم اسـت و گـراف روبـرو کـم هزينه‌ترين گراف ممکن است.



g

پیچیدگي الگوریتم: بدترین حالت الگوریتم، اگـر e تعـداد یـال‌هـا باشـد. O(e log e) است، و اگر N تعداد نودها باشد پیچیدگي $O(N^2)$ است.

الگوریتم **Prim**

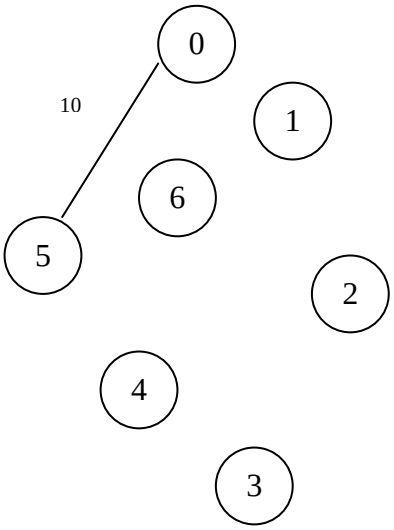
این الگوریتم نیز کم هزینه ترین درخت پوشا را مي‌سازد. اگر درخت پوشا T باشد، آنگاه از نود u شروع کرده و کم هزینه ترین یال بین دو نود (u,v) را استفاده مي‌کنیم، بـصورتي کـه حلقه نیز ایجاد نشود، سپس الگوریتم از v ادامه مي‌یابد، تا تعداد یال‌ها به n-1 برسد.

به الگوریتم زیر توجه کنید.

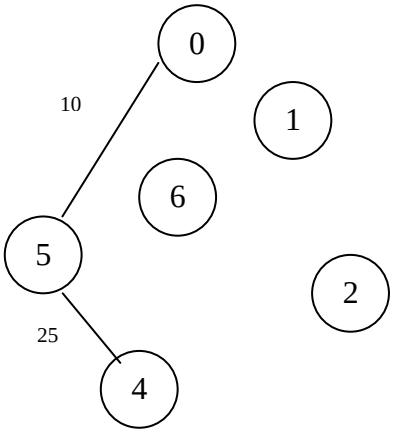
```
TV={0};//شروع از نود صفر
For(T=∅;edges of T is fewer than N-1;add (u,v) to T)
{
    let (u,v) be the least_cost edge such that u∈TV and v∉TV;// تمام همسایه هـا کنتـرل مي‌گردد
    if(there is no such edge) break;
        add v to TV;
}
if (T contains less than N-1 edges ) cout<<"can't build spanning tree";
```

پیچیدگي این الگوریتم نیز ازدرجة $O(N^2)$ است.

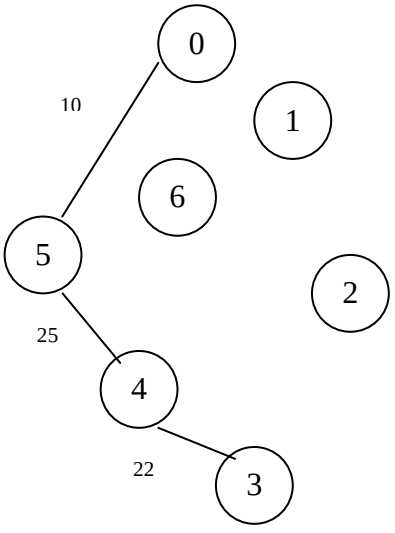
مثال: گراف G و مراحل شکل گرفتن الگوریتم Prim در مرحلة اول از نود صفر شروع کـرده و یال مینیمم ۱۰ انتخاب شده است.



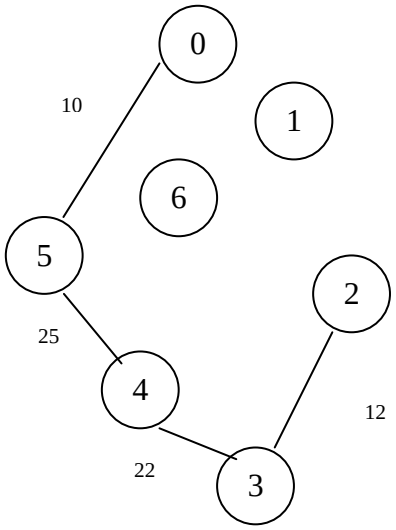
a



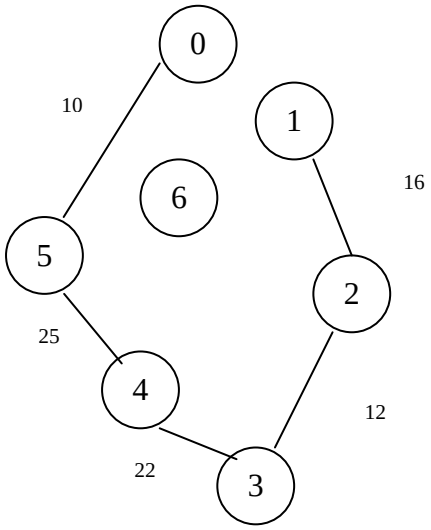
b



c

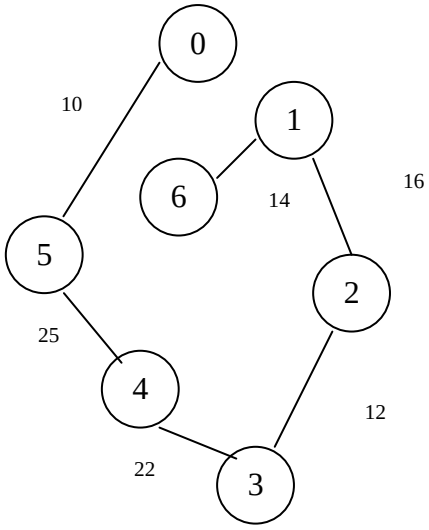


d



e

در هر مرحله اگر چند انتخاب وجود دارد، کم هزينه ترين انتخاب مي‌گردد.

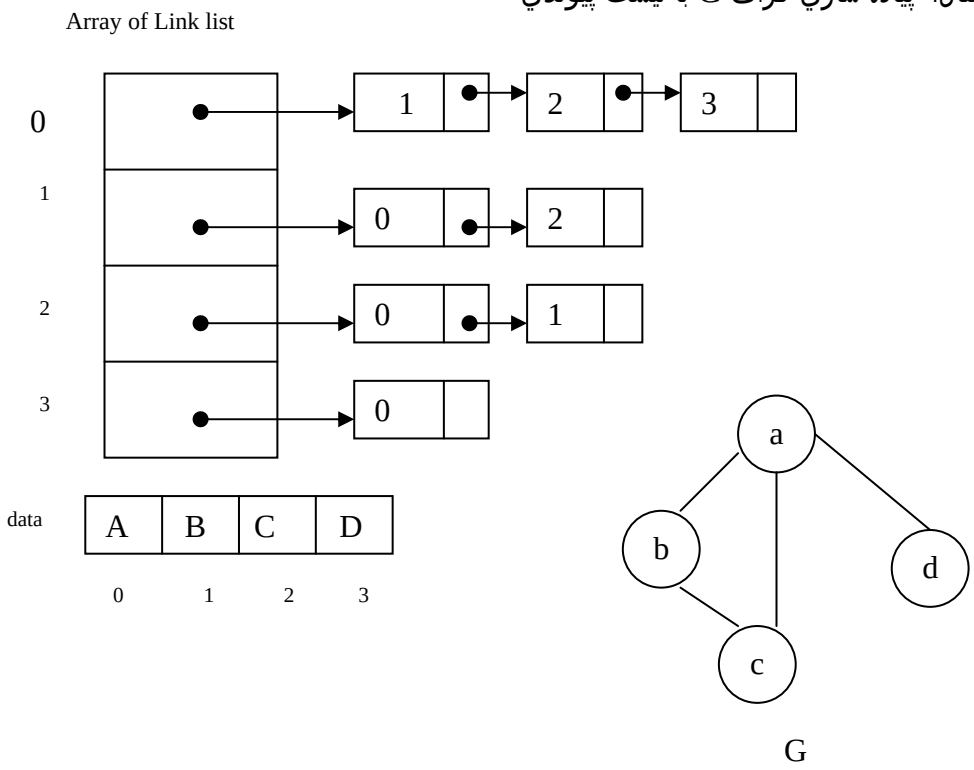


f

۸-٦-پياده سازي گراف با ليست پيوندي

در اين حالت بجاي ذخيره كردن ماتريس همسايگي از يك آرايه به تعداد نودها، از ليست‌هاي پيوندي استفاده مي‌كنيم.

مثال: پياده سازي گراف G با ليست پيوندي



G

پس براي پياده سازي بايد آراية پويا ازجنس ليست پيوندي ايجاد كنيم.

```
Private:
    Unsigned int nodeno;
    Char *data; // آراية پويا براي داده ها
    Linkedlist<int> *Array;
```

در اين حالت براي محاسبة ماتريس مسيري و ... بايد ابتدا مـاتريس همـسايگي از روي ليست پيوندي محاسبه گردد و سپس ضرب ماتريس‌ها انجام شود.

اين روش اتلاف حافظة كمتري براي گراف‌هاي بزرگ با تعداد يال كم دارد.

تمرين:

١- در روش پياده سازي ماتريس با ليست پيوندي عملگر ( ) را براي بدست آوردن يال بين دو نود سربارگذاري كنيد.

٢- در روش پياده سازي ماتريس با ليست پيوندي، تابع PathMat را براي بدست آوردن ماتريس مسيري بنويسيد.

فصل نهم

مرتب سازی

در اين فصل ما بروي ليست‌هايي از ركوردها كار مي‌كنيم، هر ركورد داراي فيلدي بنام كليد است و عنصركليد معيار مقايسه ركوردها با يكديگر است. كليد ليست‌هاي‌مختلف ممكن است متفاوت باشد و عملگرهاي > و < و == و =! در كلاس مربوطه سربارگذاري شده است.
در كلاس هر ركورد، يك تابع دسترسي عمومي (Public) به نام KEY وجود دارد كـه مقـدار كليد ركورد را بر مي‌گرداند.

۹-۱- جستجوي خطي: **(Sequential Search)**

در اين روش ليست از ابتدا الي انتها به دنبال كليد دلخواه جستجو مي‌گردد و در صـورت يافتن كليد، عمليات متوقف مي‌گردد. (فرض بر يكتا بودن كليد است).

پس پيچيدگي الگوريتم جستجو در بدترين حالت O(N) است.

```
Int sequential_search(element *mylist, const int N,KeyType myKey)
{
    //Nتعداد عناصر
    //mylist آرايه از جنس ركورد
    //myKey كليدي كه به دنبال آن جستجو مي‌كنيم
    for(int I=0;I<N;I++)
        if(list[I].key( ) == myKey)
            return I;
    return -1;//.پيدا نشده است
}
```

مقايسة دو ليست نامرتب و چاپ مواردي از ليست اول كه در ليست دوم وجود ندارد:

در اين حالت اعضاي ليست اول در ليست دوم جستجوي خطي مي‌شود، و موارد متناقض چـاپ مي‌گردد، پيچيدگي اين الگوريتم از درجة O(m n) است اگر n تعـداد اعـضاي ليـست اول و m تعداد اعضاي لسيت دوم باشد.

```
Void verify_unsorted_lists(element *l1, element *l2,const int n, const int m)
{
    for(int I=0;I<n;I++)
        if(sequential_search(l2,m,l1[I].key( )) = = -1)
            cout << “key value=”<<l1[I].key<<”Not Found\n”;

}
```

الگوريتم مقايسة دو ليست مرتب:

در اين حالت از ابتداي هر دو ليست (با فرض صعودي) شروع مي‌كنيم، داده ها را مقايسه مي‌كنيم، آنكه كوچكتر است، متغير انديسش را به خانة بعد مي‌بريم، اگر هر دو مساوي باشـند، دو انديس جلو مي‌رود، اين عمل آنقدر ادامه مي‌يابد تا يكي حداقل يكي از ليست‌ها تمام شود.

پیچیدگي این الگوریتم با محاسبة زمان مرتب سازي دو لیست میتواند باشد. O(max(nlogn, mlogm))

```
Void fast_verify(element *l1, element *l2,const int n, const int m)
{
    sort(l1,n);
    sort(l2,m);
    int I=0, j=0;// شروع از ابتداي دو ليست
    while( (I<m) && (j<n) )
    {
        if(l1[I].key = = l2[j].key)//حالت تساوي
        {
            I++;
            J++;
        }
        else if (l1[I].key>l2[j].key)
        {
            cout << “the key value”<<l2[j].key<<”is not in first list\n”;
            j++;
        }
        else // l1[I].key<l2[j].key
        {
            cout << “the key value”<<l1[I].key<<”is not in second list\n”;
            I++;
        }

    }
```

۹-۲- مرتب سازي بروش درج (INSERTION SORT)

تئوري این روش بر درج عنصر در محل صحیح خودش بصورت مرتب است. پس ما تمامي اعضاي ليست نامرتب را در يك ليست جديد كه مرتب است درج مي‌كنيم.

```
Void insert(element *list, const element &e, int N)
// درج عنصردر محل صحيح خودش
{
        while(e.key( ) <list[N].key( ) )
        {
         list[N+1]=list[N];
         N--;
        }
        list[N+1]=e;
}

void InsertionSort(element *list,const int n)
// درج اعضا در ليست جديد بصورت مرتب
{
```

فرض مي‌كنيم در خانة نخست كمترين كليد ممكن است.//;(MINKEY).setkey[0]list

```
    For(int j=2;j<=n;j++)
        insert(list,list[j],j-1);
}
```

پيچيدگي اين الگوريتم، در حالت ميانگين، از درجة $O(N^2)$ است.

مثال: مراحل مرتب سازي ليست با كليدهاي 5,4,3,2,1 بصورت صعودي (بدترين حالت)

J	[1]	[2]	[3]	[4]	[5]
-	**5**	4	3	2	1
2	**4**	**5**	3	2	1
3	**3**	**4**	**5**	2	1
4	**2**	**3**	**4**	**5**	1
5	**1**	**2**	**3**	**4**	**5**

قسمتهاي پر رنگ نشان دهندة عناصر SHIFT داده شده و درج شده در هر مرحله است.

۹-۳- مرتب سازي سريع **(Quick Sort)**

در مرتب سازي درجي، عنصر K_I محور محل درج فعلي است، تفاوت اين الگويتم با روش مرتب سازي سريع در محور است. اين تفاوت در اين است، كه محور محل درج نسبت به كل ليست در نظر گرفته مي‌شود. بنابراين اگر K_I در موقعيت S(I) قرار بگيرد آنگاه، $K_j \le K_{S(I)}$ براي $J<S(I)$ و $K_j \ge K_{S(I)}$ براي $J>S(I)$. بنابراين بعد از جاگذاري ليست به دو قسمت تقيسم مي‌شود، $R1,R2,R3,\ldots,R_{S(I)-1}$ و $R_{S(I)+1},\ldots,R_n$ كه تمام قسمت اول سمت چپ S(I) و تمام قسمت دوم سمت راست S(I) در ليست واقعي قرار مي‌گيرند.

پيچيدگي الگوريتم QuickSort در بدترين حالت از درجة $O(N^2)$ است.

```
Void QuickSort(element *list,const int left,const int right)
{
    if (left < right)
    {   int I=left, j=right+1;
        KeyType pivot = list[left].key( );
        do {
            do I++; while(list[I].key( ) < pivot);
            do j -- ; while(list[j].key( ) > pivot);
            if(I<j) listswap(list,I,j);
        }while(I<j);
        listswap(list,left,j);
        QuickSort(list,left,j-1);
        QuickSort(list,j+1,right);
    }
}
```

```
void listswap(element *list,int a,int b)
{
        element temp=list[a];
        list[a]=list[b];
        list[b]=temp;
}
```

مثال : مرتب سازي ليست بروش مرتب سازي سريع

R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	left	right
[26	5	37	1	61	11	59	15	48	19]	1	10
[11	5	19	1	15]	26	[59	61	48	37]	1	5
[1	5]	11	[19	15]	26	[59	61	48	37]	1	2
1	5	11	[19	15]	26	[59	61	48	37]	4	5
1	5	11	15	19	26	[59	61	48	37]	7	10
1	5	11	15	19	26	[48	37]	59	[61]	7	8
1	5	11	15	19	26	37	48	59	[61]	10	10
1	5	11	15	19	26	37	48	59	61		

سوال : حداكثر با چه سرعتي مي‌توان مرتب سازي يك ليست n عضوي را انجام داد؟ مي‌توان ثابت كرد كه حداكثر، سرعت مرتب سازي O(N log N) است.

۹-٤- مرتب سازي تركيبي **(Merge Sort)**

قبل از آنكه الگوريتم Merge Sort را بررسي كنيم ، روشي براي تركيب دو ليست مرتب، بررسي خواهد شد. در اين روش، از ابتداي هر دو ليست شروع مي‌كنيم، عنصري كه كليدش كوچكتر است به ليست نتيجه اضافه مي‌كنيم، و انديسش جلو مي‌رود، اگر عناصر با كليدهاي مساوي موجود بود، يكي را اضافه مي‌كنيم و انديس هر دو ليست جلو مي‌رود. اين عمل آنقدر ادامه مي‌يابد، تا يكي از دو ليست به پايان برسد، آنگاه ليست باقي مانده در انتهاي ليست نتيجه اضافه مي‌شود.

اين روش تركيب دوگانه (2 way merge) نام دارد، اين الگوريتم براي تركيب k ليست نيز به همين روش قابل انجام است و K Way Merge نام دارد.

```
element *merge(element *list1, element *list2,const int m, const int n)
{
    element *temp=new element[m+n];
    assert(temp!=0);
    int I=0,j=0,k=0;
    while(I<m && j<n)
        {
         if(list1[I].key( ) <list2.[j].key( ) )
```

```
        {
            temp[k++]=list1[I];
            I++;
        }
        else if(list1[I].key( ) > list2.[j].key( ) )
        {
            temp[k++]=list2[j];
            j++;
        }
        else //حالت تساوي
        {
            temp[k++]=list2[j];
            j++;
            I++;
        }
       }
       while(I<m)
        temp[k++]=list1[I++];
       while(j<n)
        temp[k++]=list2[j++];
       return temp;
}
```

پيچيدگي اين الگوريتم از درجة O(n) است.

مرتب سازي ليست بروش **Merge Sort**

اين الگوريتم در ابتدا با N ليست مرتب يك عضوي كار خود را شروع مي‌كند.در اين روش عناصر دو بدو بصورت مرتب با هم تركيب مي‌شوند، پس از يك مرحله تعدادي ليست دو عنصري وجود دارد كه بايد با هم تركيب، شوند تا ليست نهايي بدست آيد.

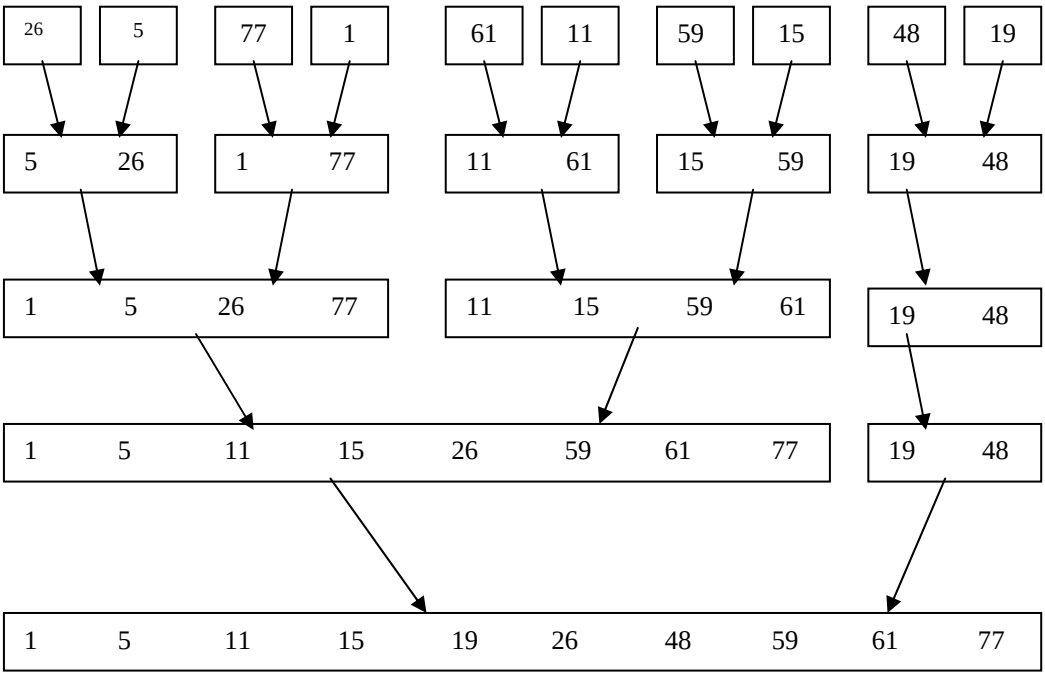
```
void mergepass(element *orginallist,element *resultlist, const int n, const int l)
{//يك مرحله از الگوريتم مرتب سازي تركيبي است،عناصر همسايه دو ليست با هم تركيب مي‌شوند
for(int I=1;I<=n-2*l+1;I+=2*l)// l شرط ادامة حلقه تا اتمام دو ليست با زير ليست‌هايي بطول
        merge(orginallist,resultlist,I,I+l-1,I+2*l-1);
    if( (I+l-1)<n) merge(orginallist,resultlist,I,I+l-1,n);//تركيب باقي ماندة دو ليست
    else for(int t=I;t<=n;t++) resultlist[t]=orginallist[t];
}

void mergeSort(element *list,const int n)
{
    element *templist=new element[n+1];
    for(int l=1;l<=n;l*=2)//.l طول زير ليست‌هايي است كه با هم تركيب مي‌شود
    {
        MergePass(list,templist,n,l);
        l*=2;
    MergePass(templist,list,n,l);جابجايي عناصر از ليست موقتي به ليست واقعي
```

```
    }
delete [ ]  templist;
}
```

پيچيدگي اين الگوريتم، از درجة O(N log N) است.

مثال : نمايش مراحل مرتب سازي تركيبي



۹-۵- مرتب سازي **Heap Sort**

در اين روش، يك ساختار Max Heap با داده هاي ليست ايجاد مي‌گردد. توابع حذف و درج در ساختار Heap از درجة O(N log N) است.

```
void adjust(element *tree, const int root, const int n)
//تابع تنظيم درخت
{
    element e=tree[root];
    KeyType k=e.key( );
        for(int j=2*root;j<=n;j*=2)
        {//ابتدا ماكزيمم فرزند چپ و راست را پيدا مي‌كنيم
         if(j<n)
              if(tree[j].key( ) < tree[j+1].key( )) j++;
```

ماكزيمم فرزندان را با k مقايسه مي‌كنيم اگر k بزرگتر باشد تمام است

```
        if(k>=tree[j].key( )) break;
        tree[j/2]=tree[j];//تعويض با والد
       }
      tree[j/2]=e;
}

void HreapSort(element *list, const int n)
{
    for(int I=n/2;I>=1;I--)
        adjust(list,I,n);//تبديل ليست به هيپ
    for(I=n-1;I>=1;I--)//مرتب سازي ليست
    {
        element t=list[I+1];
        list[I+1]=list[1];
        list[1]=t;
        adjust(list,1,I);//مرتب سازي مجدد هيپ
    }
}
```

فصل دهم

دیکشنری

جدول نماد، در كاربردهاي گوناگون، استفاده مي‌شود، بنابراين هدف جدول نمادي (ديكشنري) سريع، با قابليت‌هاي ذيل است:

- مشخص كند كه يك نام خاص (كليد) در جدول است يا خير؟
- اگر نام در جدول موجود است خواص (داده‌ها) آنرا بازيابي كند.
- تغيير خصوصيات يك نام خاص.
- درج نام (كليد) جديد در جدول و ذخيرة خصوصيات آن
- حذف يك نام و خصوصياتش از جدول

كلاس ADT براي ساختار Hashing به صورت زير تعريف مي‌شود:

فرض مي‌كنيم نام (كليد) رشته‌اي باشد:

```
template <class Attribute>
class SymbolTable
{
    public:
        SymbolTable(int size=200); //ايجاد جدول نماد تهي
        ~SymbolTable( );//مخرب كلاس
    bool IsInTable(const char *name);//جستجوي جدول به دنبال كليد خاص
    const Attribute  &Find(char *name);//بدست آوردن خصوصيات يك كليد
    void insert(const char *name,const Attribute &attr);// درج
    void Delete(const char *name);//حذف يك كليد و خصوصياتش
};
```

در اين ساختار (STATIC HASHING) محل ذخيره شدن هر عنصر مشخص است بنابراين جستجوي آن داراي پيچيدگي زماني O(1) است. اگر در حالت عملي حداكثر K تداخل وجود داشته باشد، آنگاه پيچيدگي O(K) است.

مثال زير يك جدول نماد را در حالت ايستا نشان مي‌دهد.

0	خصوصيت ۰
1	خصوصيت ۱
2	…
0	
0	
N	

در اين ساختار تـابعي بنـام HASH FUNCTION وجـود دارد، كـه كليـد را دريافـت مي‌كند و خروجي آن محلي است كه عنصر در جدول وجود دارد.(و يا بايد وجود داشته باشد)

اين تابع در حالت ايده آل براي هر كليد، يك جـواب عـددي منحـصربفرد بـين ۰ الـي n ميدهد. ولي در حالت عملي، ممكن است براي دو كليـد نامـساوي، جوابهـاي يكـسان داشـته باشيم، اين حالت تداخل (Collision) ناميده مي‌شود.

روشهاي متنوعي براي بر طرف كردن تداخل وجـود دارد، يكـي از ايـن روشـها، در نظـر گرفتن چندين ستون، براي داده‌هاست، تابتوان بيش از يك داده را در هر سطر ذخيره كرد.

روش ديگر استفاده از يك ليست پيوندي در هر سـطر اسـت. البتـه حتـي در حـالتي كـه براحتي بتوان تداخل را برطرف كرد، بايد تابعي براي Hash انتخاب شود كه كمترين تداخل را داشته باشد. و تعداد تداخل‌ها بايد نسبت به نوع مساله قابل قبول باشد.

از نظر رياضي توابع Hash، بررسي مي‌شود، تا كمترين تداخل را داشـته باشـد . در زيـر چند نمونه از توابع معروف آورده شده است:

Mid Square-۱

اين تابع با به توان رساندن شناسه، و انتخاب بيتهايي از ميان شناسة به توان رسـيده سـعي در توليد اعداد متمايز در محدودة داده شده مي‌نمايد.بيت‌هاي مياني شناسـة مجـذور شـده، بـه تمامي بيتهاي شناسه مرتبط است.

Division-۲

در اين روش از عملگر باقي مانده (%) استفاده مي‌شود. اين عملگر از شناسه يك مولفـة عددي استخراج مي‌شود و باقي ماندة آن بر M كه يـك تـوان از ۲ اسـت محاسـبه مـي‌گـردد. جواب بين ۰ الي M-1 خواهد بود.

Folding-۳

در اين روش، شناسه به چند قسمت تقسيم مي‌شود، قسمتهاي مختلف شناسـه ارزشـيابي عددي مي‌شوند و سپس يك عملگر رياضي مانند مجموع بروي آنها اعمال مي‌شود.

حال با كمك آرايه پويا، جدول نماد را پياده سازي مي‌كنيم:

```
Private:
    element **Array;
    Unsigned max;
    Int hash(char *x);// تابع مشخص كنندة محل كليد به يكي از روشهاي فوق

template <class Attribute>
class element
{   friend class SymbolTable;
```

```
    private:
        Attribute attr;
        bool isempty;
        char *key;

    public:
        element( ) {isempty=true;key=NULL;}
};

template <class Attribute>
SymbolTable<Attribute>::SymbolTable(int size)
{
    max=size;
    Array=(element<Attribute> **)(new element<Attribute>[max][3]);
    Assert(Array!=0);
}

template <class Attribute>
bool SymbolTable<Attribute>::IsInTable(char *name)
{
    int I=hash(name);
    if(squential_srearch(Array[I],name,3) = = -1)
        return FALSE;
    else
        return TRUE;
}

template <class Attribute>
const Attribute &SymbolTable<Attribute>::Find(char *name)
{
    int I=hash(name);
    int t;
    if((t = squential_srearch(Array[I],name,3) )= = -1)
        assert(0);
    else
        return Array[I][t];
}

template <class Attribute>
void SymbolTable<Attribute>::Delete(char *name)
{
    if(IsInTable(name))
    {
        int I=hash(name);
        int k=sequential_sreach(Array[I],name,3);
        Array[I][k].isempty=TRUE;
        delete Array[I][k].key;
    }
}
```

```
template <class Attribute>
void SymbolTable<Attribute>::Insert(char *name,const Attribute &attr)
{
    int I=hash(name);
    int empty=0;
    for(;empty<3&& ! Array[I][empty].isempty;empty++);
    if(empty = = 3)
    {
         cerr<< “cant Add Collision size is FULL”;
         return;
        }
        Array[I][empty].key=new char(strlen(name)+1);
        assert(Array[I][empty].key!=0);
    strcpy(Array[I][empty].key,name);
    Array[I][empty].isempty=FALSE;
    Array[I][empty].attr = attr;
}
```

اگر بخواهيم، اين ساختار را با ليست پيوندي پياده سازي كنيم، بايد يك ليست پيوندي كه داده‌هايش از جنس كلاس element باشد بوجود بياوريم.

```
private:
    linkedlist<element<attribute>> *Array;
    unsigned Max;
```

در اين حالت، حذف داده، حذف نود از ليست مربوطه است.

اضافه كردن داده، اضافه كردن به انتهاي ليست مربوطه است.

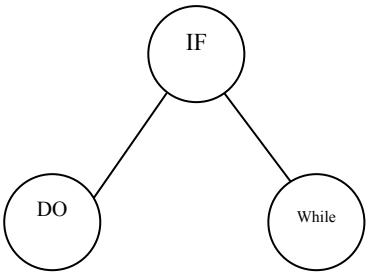
جستجو عبارت از جستجوي خطي، ليست مربوطه است.

فصل یازدهم

ساختارهای درخت جستجوی بهینه

در قسمتهاي قبل درخت دودويي جستجو، مورد بررسي قرار گرفت.

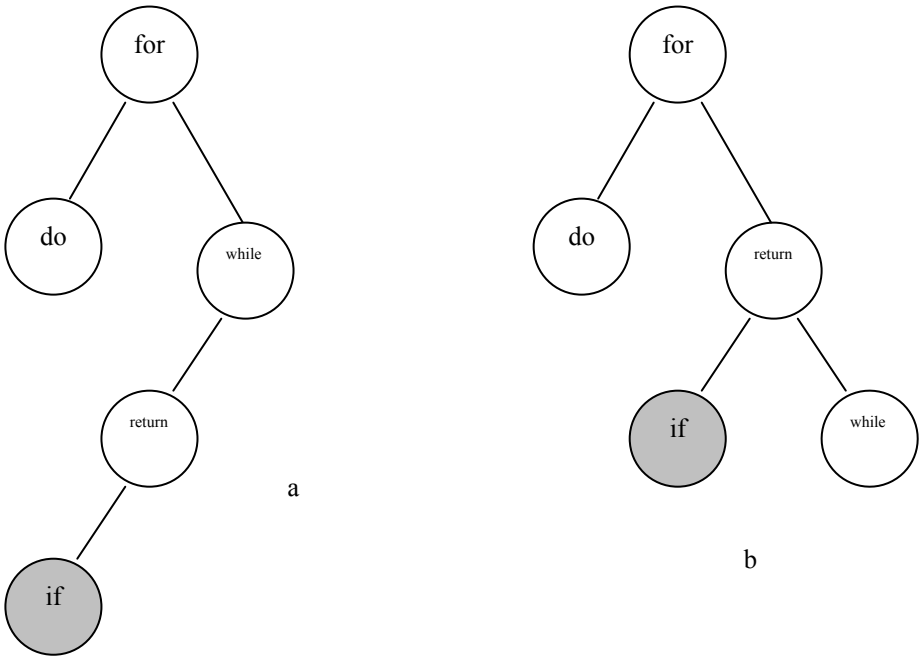
به درخت دودويي جستجو، شكل زير دقت كنيد، اين درخت حالت بهينـه نـدارد و يـك درخت دودويي كامل است، زيرا اگر فرمول‌هاي محاسبة احتمال را بكاربريم، براي متغيرهـاي مختلف با توجه به ارزش آن مقادير متفاوتي بدست مي‌آيد.



درخت دودويي جستجوي كامل

براي يافتن درخت جستجوي بهينه، براي فايل ايستاي دلخواه ابتدا بايد هزينه اي را بـراي جستجوي درخت، مشخص كنيم. وقتي به دنبال يك مشخصه در سطح K جـستجو مـي‌كنـيم، حلقه K بار تكرار مي‌شود، اين نكته هزينه را مشخص مي‌كند.

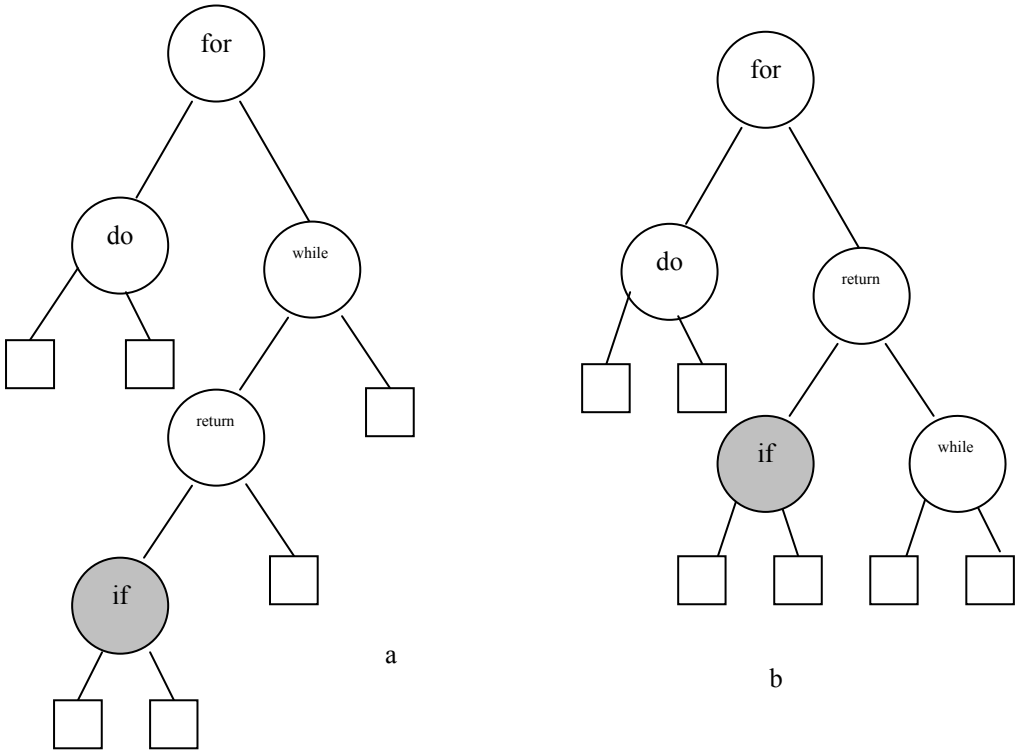
به دو درخت زير توجه نمائيد، درخت a به حداكثر ٤ مقايسه براي يافتن كليد if نيازمنـد است و درخت b همين كليد را در ۳ مقايسه بدست مي‌آورد.



مقايسة دو درخت دودويي جستجو

بنابراين در بدترين حالت، درخت b نسبت به حالت a ارجح است. براي محاسبة درخت دودويي جستجو، اشاره‌گرهاي Null را با نودهاي مربعي جايگزين مي‌كنيم. توجه كنيد در يك درخت دودويي جستجو با n نود تعداد اشاره گرهاي Null، n+1 است. اين نودهاي مربعي را نودهاي خارجي (External node) ناميده مي‌شود زيرا نودهاي واقعي درخت اصلي نيستند، و نودهـاي غير مربعي را نودهاي داخلي (Internal node) ناميده مي‌شود. هرگاه كه يك درخت دودويي جستجو به دنبال يك كليد، كه در درخت موجود نيست، جـستجو شـود، جـستجو در نودهاي خارجي پايان مي‌پذيرد. بنابراين به نودهاي خارجي، نودهـاي جـستجوي نـاموفق نيـز گفته مي‌شود.

يك درخت دودويي جستجو كه نودهاي خارجي بـه آن اضـافه شـده، درخـت دودويـي توسعه يافته ناميده مي‌شود. به شكل زير توجه كنيد.



درخت دودويي توسعه يافته

تعريف- طول مسيرخارجي **(External Path Length)**

طول مسيرخارجي براي يك درخت دودويي، مجموع طول مسير تمام نودهاي خارجي از

ريشه است.

تعريف- طول مسير داخلي **(Internal Path Length)**

طول مسير داخلي براي يك درخت دودويي، مجموع طول مسير تمام نودهـاي داخلـي از ريشه است.

مثال : به شكل درخت توسعه يافتة a توجه كنيد

I = 0+1+1+2+3=7
E = 2+2+4+4+3+2=17

I طول مسيرداخلي و E طول مسير خارجي است.

اگر I طول مسيرداخلي و E طول مسير خارجي و n تعداد نود درخت باشد، آنگاه را بطة زير برقرار خواهد بود: $E = I + 2n$

با توجه به فرمول، متوجه مي‌شويم كه طول مسير داخلي و خارجي به هم وابـسته اسـت. ماكزيمم I زماني است كه درخت با n نود، عمق n داشته باشيم (مانند ليست) در اين حالـت I از فرمول زير محاسبه مي‌گردد:

$$I = n(n-1)/2$$

حال مي‌خواهيم مقدار كمينه را براي I مشخص كنيم، حداكثر دو نود با طول يك مي‌توان داشت، چهار نود با طول دو و .... پس خواهيم داشت:

$$0+2*1+4*2+8*3+\ldots+$$

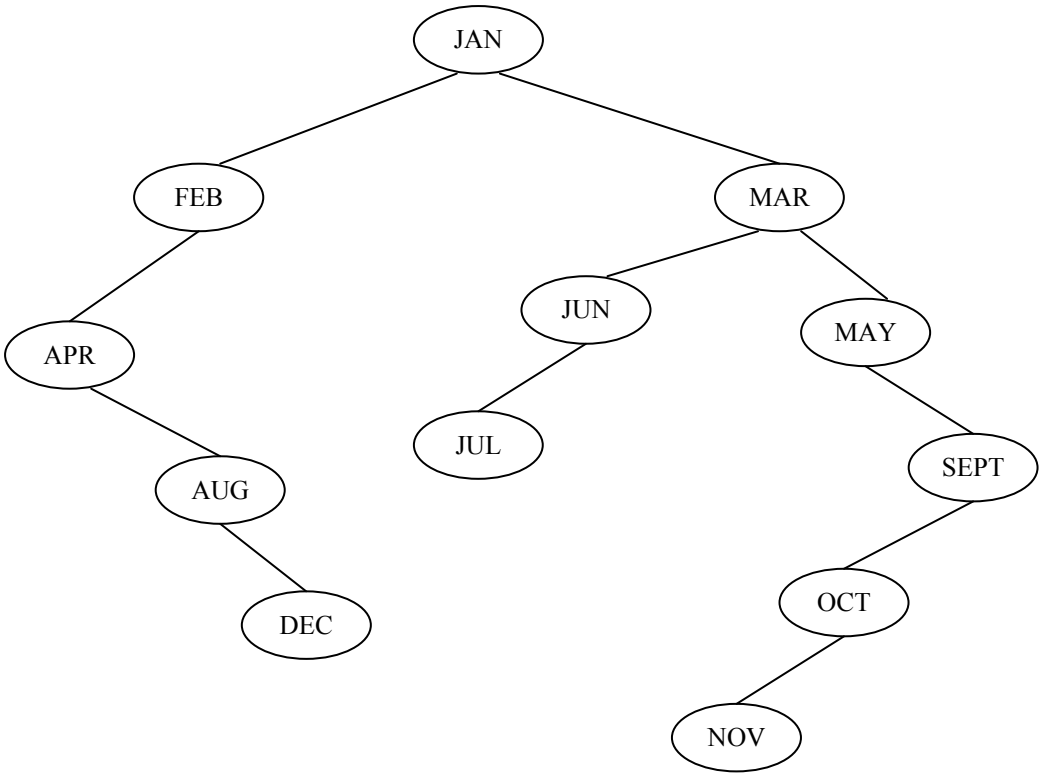
حالت كمينه براي طول مسير داخلـي بـراي يـك درخـت دودويـي كامـل (Complete) حاصل مي‌گردد. اگر نودها را از ۱ الي n شماره گذاري كنيم آنگـاه، فاصـلة نـود i ام از ريـشه $[\log_2 i]$ است.

بنابراين با n نود كمترين مقدار ممكن از $O(n \log n)$ است.

بنابراين درخت دودويي جستجو كه – كامل نيز باشد ساختار بهينه اي دارد.

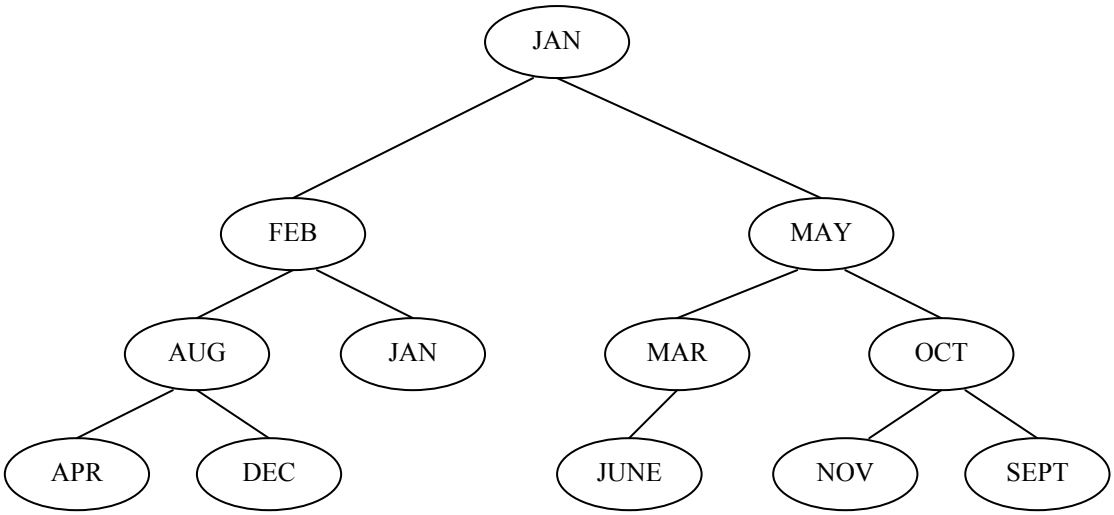
۱۱-۲- درخت AVL

جداول پويا، معمولاً توسط درخت دودويي جستجو مدل مـي‌شـوند. در قـسمتهاي قبـل، روش درج و حذف در درخت دودويي جستجو بررسي گرديد. شـكل زيـر درخـت دودويـي ساخته شده از ماه هاي ميلادي را نشان مي‌دهد.



درخت دودويي جستجو از ماه هاي ميلادي

به درخت زير توجه كنيد، ايـن همـان درخـت دودويـي قبـل اسـت، كـه كـاملاً *بـالانس* (Balance) شده است. در اين حالت طول مسير تمامي برگها تا ريشه برابر است و هيچ يـك بيش از ديگري نيست.



درخت دودويي جستجوي بالانس از ماه هاي ميلادي

بدترين حالت زماني خواهد بود كه n نود تشكيل درختي به عمق n بدهد، در اين حالت جستجو در بدترين حالت n مقايسه لازم دارد و حالت ميانگين O(log n) است.

تعريف درخت دودويي **AVL** : يك درخت تهي، كاملاً موازنه (بالانس)، است. اگر درخت دودويي T غير تهي است، كه T_L و T_R، زير درخت چپ و راست باشد، آنگاه T كاملاً موازنه (بالانس) است، اگر و فقط اگر T_L و T_R كاملاً موازنه باشد و $|h_L-h_R|\leq 1$ باشد، كه h_L و h_R، ارتفاع زير درخت‌هاي T_L و T_R است.

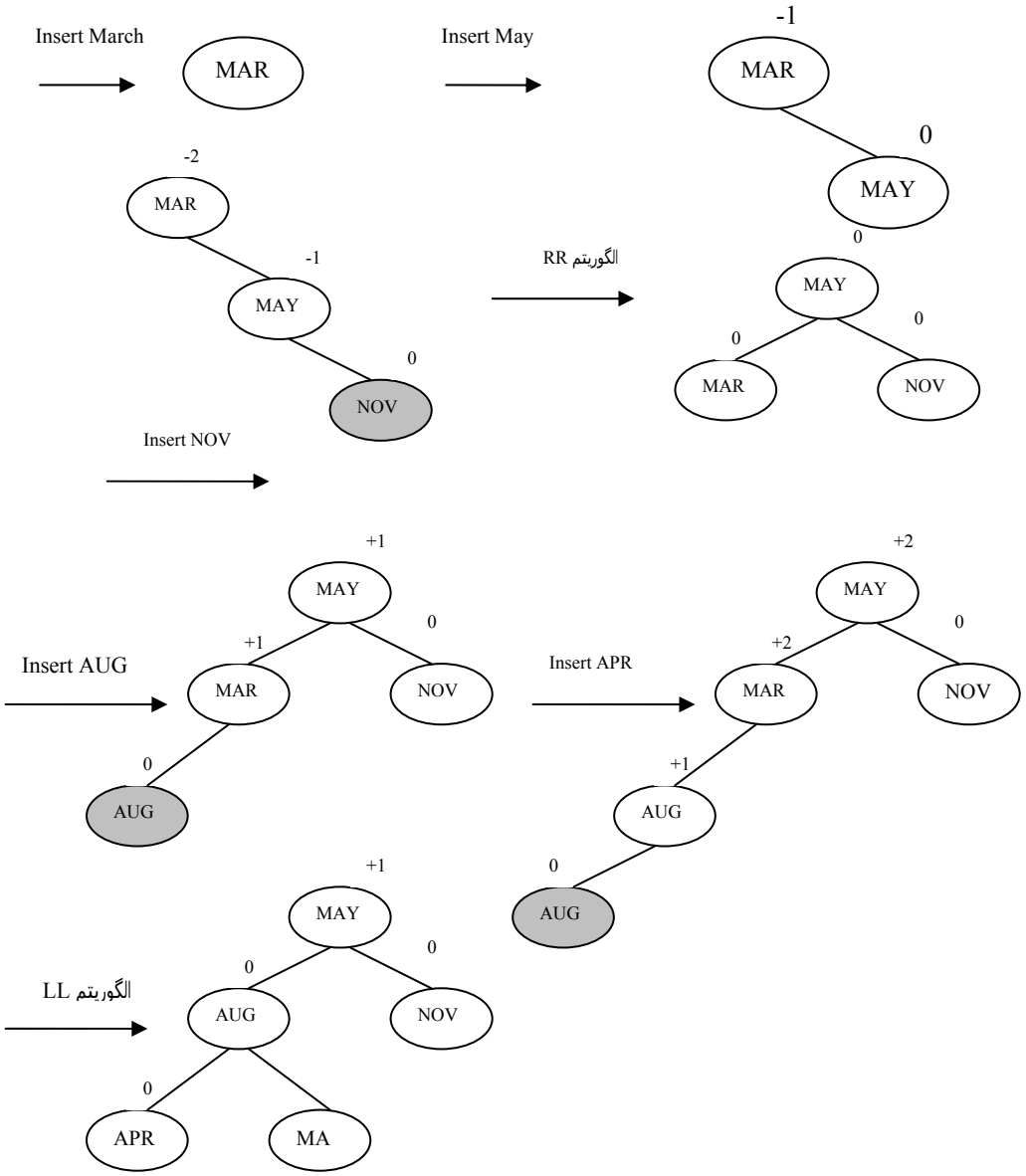
دقت: تعريف فوق بازگشتي است.

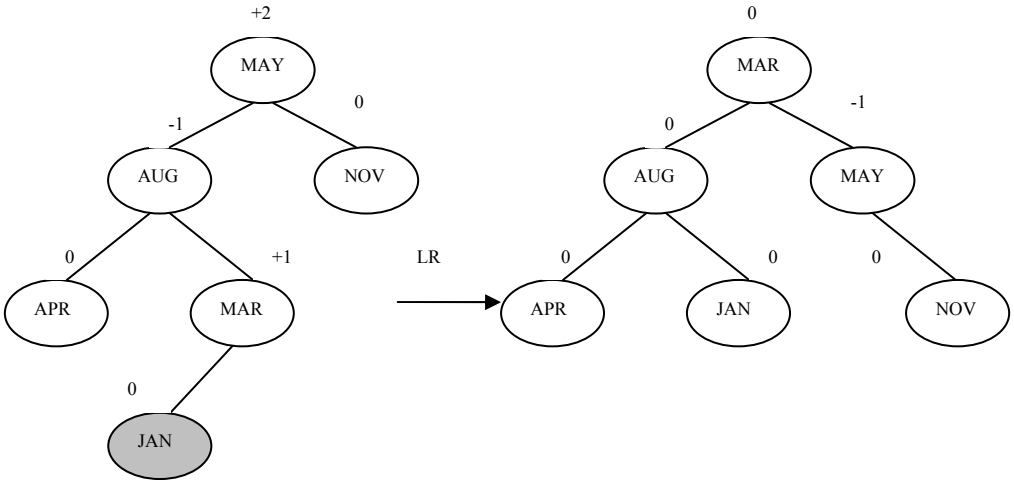
تعريف فاكتور موازنه **BF(T) (Balance Factor)**

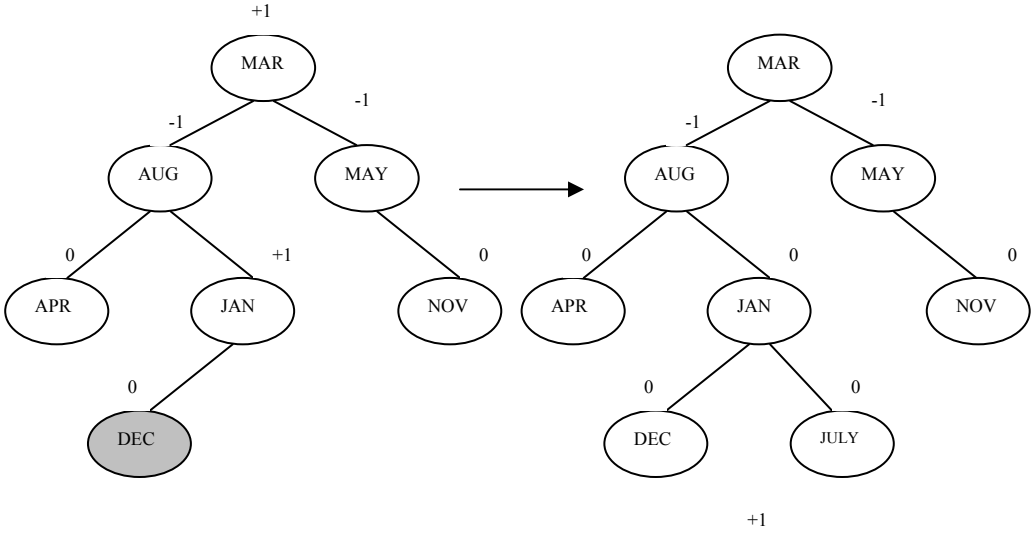
BF(T)، فاكتور موازنه، براي نود T، در درخت دودويي، برابر ارتفاع زير درخت چپ منهاي ارتفاع زير درخت راست است. در درخت دودويي AVL، BF(T)، براي هر نود عدد ۰ يا ۱+ يا ۱- است.

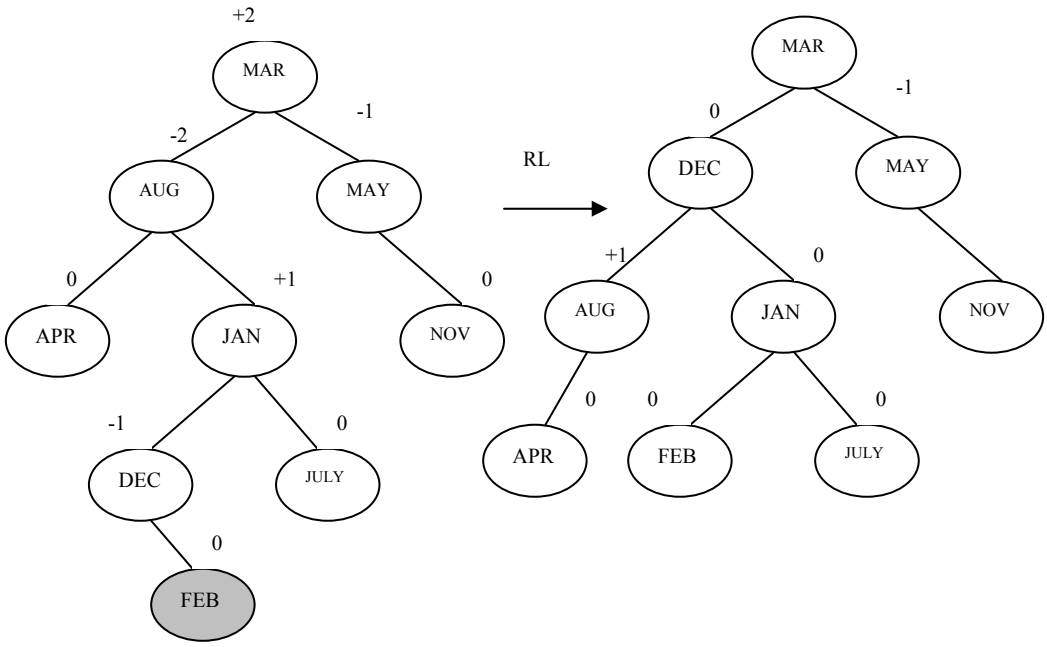
پس از هر درج يا حذف، بايد مجدد درخت AVL را موازنه كنيم.

مثال: درج نود در درخت AVL




+2
MAY
-1
0
AUG
NOV
0
+1
APR
MAR
0
JAN
LR
0
MAR
0
-1
AUG
MAY
0
0
0
APR
JAN
NOV


+1
MAR
-1
-1
AUG
MAY
0
+1
0
APR
JAN
NOV
0
DEC
MAR
-1
-1
AUG
MAY
0
0
0
APR
JAN
NOV
0
0
DEC
JULY
+1



با توجه به اشكال موجود در صفحات قبل، بعد از درج نود Y، بـراي نـودي كـه فـاكتور موازنة آن ٢± است، يكي از حالات زير اتفاق مي‌افتد،

LL : نود جديد Y در سمت چپ، زير درخت چپ A، درج شده است.

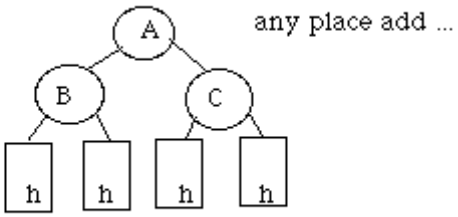
LR : نود جديد Y در سمت راست زير درخت چپ A، درج شده است.

RR : نود جديد Y در سمت راست، زير درخت راست A، درج شده است.

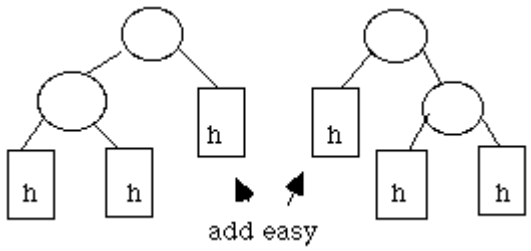
RL : نود جديد Y در سمت چپ، زير درخت راست A، درج شده است.

به شكلهاي زير توجه نمائيد، محلهاي مناسب جهت درج و يا بازسازي مشخص شده است:

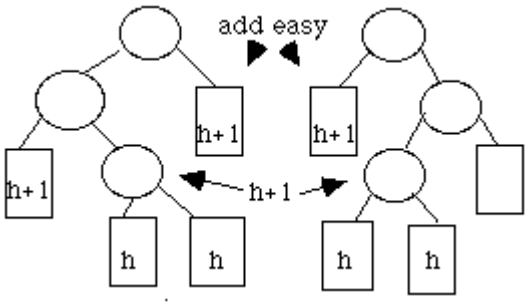
درج در هر محل احتياج به بازسازي درخت ندارد.(Fully Compelete Tree)



در محل مشخص شده درخت، پس از درج، AVL مي‌ماند.



در محل مشخص شده درخت، پس از درج، AVL مي‌ماند.



ساختار هر نود در درخت AVL

```
template <class T> class AVL;
template <class T>
class AVLnode
{
    friend class AVL<T>;
   private:
    T data;// دادة هر نود
    AVLnode<T> *left,*right;//اشاره گربه فرزند چپ و راست
    int bf;//فاکتور موازنه };
template <class T>
class AVL
{
    public:
        AVL(const AVLnode<T> *myroot) :root(myroot) {}
        bool Insert(const T &mydata);
        bool Delete(const T &mydata);
        const AVLnode *Search(const T &sample);
    private:
        AVLnode<T>  *root;
};
```

در درخت AVL، درج، حذف، درج عنصر K ام، جستجو داراي پيچيدگي از درجة O(log n) است.

تمرين:

١- درج يك نود و بازسازي درخت AVL را در تابع Insert بنويسيد.

٢- حذف يك نود را با كمك تابع جستجو، و بازسازي درخت را بنويسيد.

١١-٣- درخت‌هاي ٢-٣ (2-3 Trees)

با در نظر گرفتن درخت‌هايي با درجة بيش از ٢، مي‌توانيم به درخت‌هايي برسيم كه درج و حذف در آنها از درخت دودويي ساده‌تر باشد. اين الگوريتم‌ها در اين درخت‌ها با پيچيدگي O(log n) انجام مي‌شود.

درخت ٢-٣ يك حالت خاص از درخت‌هاي B (B-Trees) است. هر نود داخلي در درخت ٢-٣ داراي درجة ٢ و يا ٣ است. يك نود با درجة ٢، 2-node و نود با درجة ٣، 3-node ناميده مي‌شود.

تعريف: يك درخت ٢-٣، يك درخت جستجو است، كه يا تهي است و داراي خواص زير است:

١- هر نود داخلي، 2-node يا 3-node است. يك 2-node داراي يك المان، و 3-node داراي دو المان است.

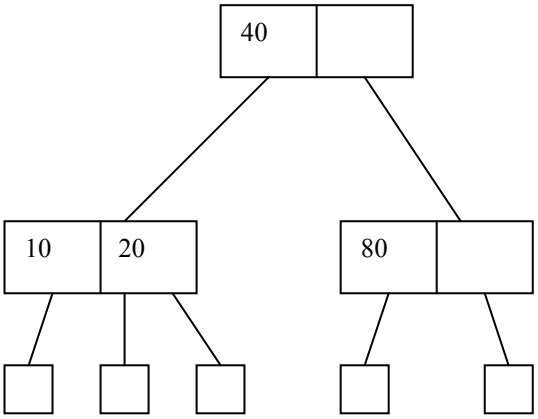
٢- فرزندان نود 2-node را، فرزند چپ (Left) و فرزند مياني (Middle) مي‌ناميم. و dataL، المان (كليد و دادة) درون، اين نود است، تمامي نودهاي زيرشاخة چپ، داراي كليدي كمتر از كليد واقع در 2-node دارند. و تمامي نودهاي واقع در زير درخت مياني، داراي كليدي بيشتر از كليد واقع در ريشة خودشان دارند.

٣-فرزندان 3-node را فرزند چپ (Left)، فرزند مياني (Middle) و فرزند راست (Right) مي‌ناميم. المانهاي 3-node را dataL و dataR مي‌ناميم. (با شرط dataL<dataR)

تمامي نودهاي زير درخت چپ داراي كليدي كمتر از كليد dataL، هستند. و تمامي نودهاي واقع در زير درخت مياني، داراي كليدي كمتر از كليد dataR، و بيشتر از كليد dataL هستند. تمامي نودهاي واقع در زير درخت راست، داراي كليدي بيشتر از كليد dataR هستند.

٤- تمامي نودهاي خارجي (External) در يك سطح هستند.

مثال: درخت ۲-۳



تعداد نودهاي درخت ۲-۳، با ارتفاع h، (نودهاي خارجي در ســطح h+1 هستند)، بين 2^h-1 و 3^h-1 است.

ارتفاع يك درخت ۲-۳، با n نود، بين $[\log_3(n+1)]$ و $[\log_2(n+1)]$ است.

تعريف ADT درخت ۲-۳

```
template <class T> Two3;
template <class T>
class Two3Node
{
    friend class Two3<T>;
    private:
        T dataL,dataR;
        Two3Node<T> *left,*middle,*right;
};

template <class T>
class Two3
{
    public:
        Two3(KEYTYPE max, const Two3Node<T> *myroot=0)
                        :MAXKEY(max), root(myroot) { }
        bool Insert(const T &element);
        bool Delete(const T &element);
        const Two3Node<T> *Search(const T &sample);
    private:
        Two3Node<T> *root;//ريشة درخت
        KEYTYPE MAXKEY;//ماكزيمم كليد درخت
};
```

جستجوي درخت ٢-٣

```
template <class T>
const Two3Node<T> *Two3 :: Search(const T &x)
{
    Two3Node<T> *p=root;
    while( p )
    {
        if( p->compare(x) )
        {
            case 1: p=p->left;       break; //dataL کلید کوچکتر از
            case 2: p=p->middle;  break; //dataR کلید کوچکتر از
                                    // dataL کلید بزرگتر از
            case 3: p=p->right;      break;// dataR کلید بزرگتر از
            case 4: return p; //.x در نود پیدا شده است
        }
    }
    return p;
}
```

جستجو در درخت ٢-٣، داراي پيچيدگي از درجة O(log n) است.

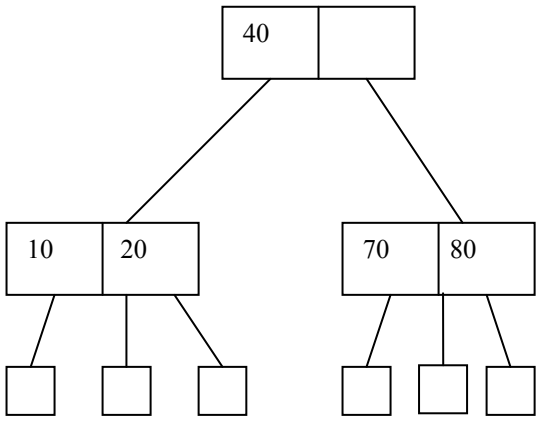
درج در درخت ٢-٣:

درج در درخت ٢-٣ بسيار ساده است، شكل زير درج نودي با كليد ٧٠ را نشان مي‌دهـد. درخت اوليه، درخت مثال قبل است.

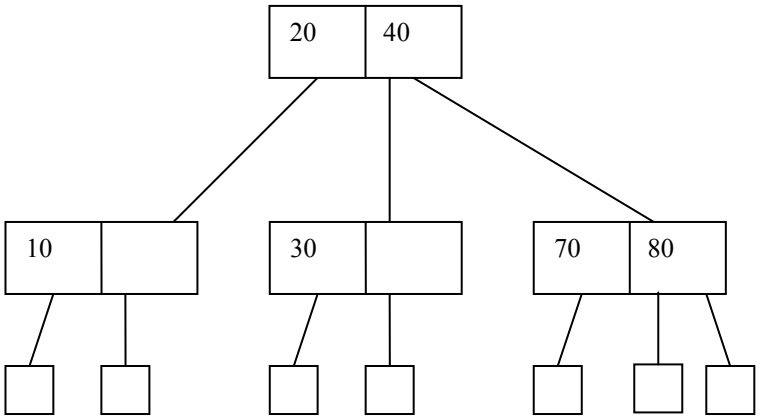
الگوريتم:

ابتدا كليد را جستجو مي‌كنيم، اگر موجود باشد، درج انجام نمي شود.

سپس نود محل درج نود را پيدا مي‌كنيم، و آنرا در نود درج مي‌كنيم. (دراين حالـت يـك 2-node به 3-node تبديل شده است)

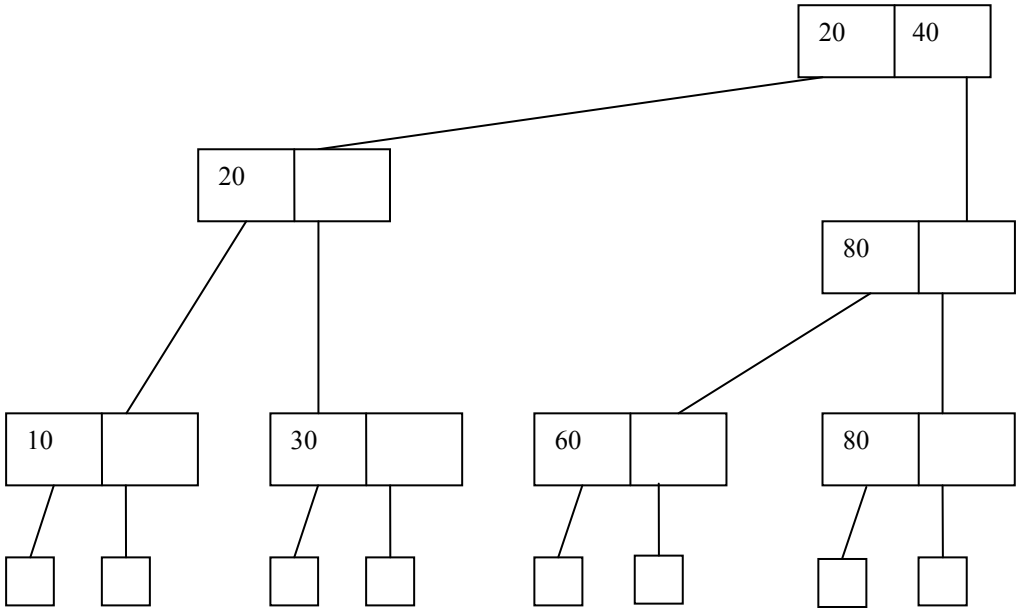


درج نود با کلید ۳۰



در این حالت ریشه از 2-node به 3-node تبدیل شده است، و نود سمت چپ به 2-node تبدیل شده است . (به دلیل محل درج، در ضمن درخت باید ۲-۳ باقي بماند)

حال درخت را براي درج نودي با کلید ٦٠ رسم مي‌کنیم. در این حالت، در این حالت تعداد سطوح درخت اضافه مي‌گردد.



توابع کمکي مورد نیاز براي درج در درخت ۲-۳

```
void Two3 :: NewRoot(const T &x,Two3Node<T> *a);
```

اين تابع وقتي فراخواني مي‌گردد، كه قرار است ريشة درخت ٢-٣ تغيير كند، المـان x در ريشة جديد درج مي‌گردد، و a فرزند مياني ريشه مي‌گردد. ريشة قديمي، فرزنـد چـپ ريـشة جديد مي‌گردد.

```
Two3Node<T> *Two3<T> :: FindNode(const T &x);
```

اين تابع نسخة تغيير يافتة تابع Search است، اين تابع درخت غير تهي را به دنبال المان x جستجو مي‌كند، اگر نود پيدا نشود، 0 برميگرداند. در غير اينصورت يك اشاره گر بـه بـرگ (p)، محل درج نود بر مي‌گرداند. امكان بازگشت از p به سمت ريشة درخت نيـز وجـود دارد (با ليست كردن مسير نودها از ريشه تا p)

```
void InsertionError( )
```

اين تابع پيام خطا را براي زماني كه نود در درخت موجود است، چاپ مي‌كند.

```
void Two3Node<T> :: PutIn(const T &x, Two3Node<T> *a)
```

اين تابع براي درج المان x در نود (this) است. زير درخت a به عنـوان سـمت راسـت مستقيم x قرار خواهد گرفت. بنابراين اگر x، dataL شود، بنابراين a فرزنـد ميـاني خواهـد بود، و مقدار قبلي dataL و فرزند مياني قبلي، به dataR و فرزند راست منتقل مي‌گـردد.اگـر x، dataR شود، آنگاه a فرزند راست خواهد بود.

```
T &Two3<T> :: Split(Two3Node<T> *p, T &x, Two3Node<T> *olda,
Two3Node<T> *a)
```

اين تابع بروي نود p يك در ابتدا داراي دو المان است كـار مـي‌كنـد، نـود جديـد توليـد مي‌گردد، و با بزگترين كليد دو المان اولية موجود در p و x مقدار دهي مـي‌گـردد و توسـط a مشخص مي‌گردد.

```
template <class T>
bool Two3<T> :: Insert(const T &y)
{
    Two3Node<T> *p;
    T x = y;
    if( x.key( ) > MAXKEY )  return FALSE; //كليد اشتباه
    if( !root ) { NewRoot(x,0);  return TRUE; } //درج در درخت تهي
    if( !(p=FindNode(x))) //نود وجود است؟
    {
        InsertionError( );//چاپ پيام خطا
        return FALSE;
    }
    for( Two3Node<T> *a=0 ; ;)
        if( p->dataR == MAXKEY ) // است 2-node - p
        {
            p->PutIn(x,0);
```

```
            return TRUE;
        }
        else { // است 3-node – p
            Two3Node<T> *olda=a;
            a = new(Two3Node<T>);
            assert(a!=0);
            x=Split(p,x,olda,a);
            if( root == p){ NewRoot(x,a);  return TRUE; } //ريشه شكسته مي‌شود
            else p=p->parent( );
    }//for
}//function
```

درج يك نود در درخت ٢-٣ با n نود داراي پيچيدگي از درجة O(log n) است.

حذف نود در درخت ٢-٣

حذف نود مشكل تر از درج آن نيست، اگر الماني در نود غير برگ حذف گردد، ما آنرا به حذف المان از برگ تبديل خواهيم كرد، اين عمل با كمك، جايگزيني المان حذف شده با المان مناسبي در يك برگ انجام خواهيم داد. در حالت كلي جايگزيني با بزرگترين المان زير درخت چپ، ويا با كوچكترين المان زير درخت راست انجام خواهد شد. بنابراين مي‌توان حالتي را در نظر گرفت كه فقط نود از يك برگ حذف مي‌گردد. (بقية حالات به اين حالت تبديل مي‌گردد)

الگوريتم حذف نود از يك برگ مي‌تواند بصورت زير باشد :

١-نود p را تغيير دهيد، كه وضعيت آن به كل دلخواه در آيد.

٢- تاوقتي p المان صفر دارد، و p برابر ريشه نيست :

{ r را برابر، پدر p قرار دهيد. و q را برابر همسايهٔ چپ يا راست p قرار دهيد.

اگر q، يك 3-node است : چرخش انجام دهيد.

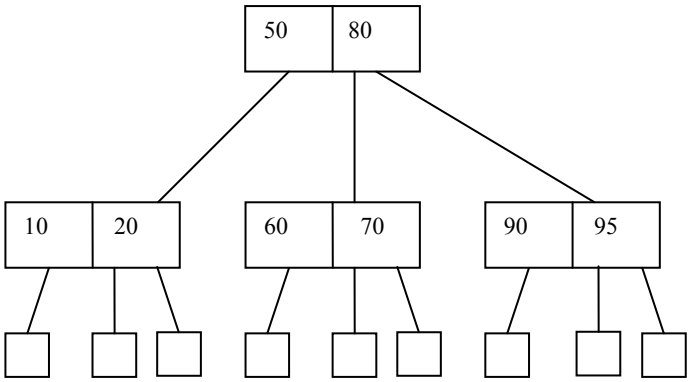
وگرنه تركيب انجام دهيد.

p را برابر r قرار دهيد و مرحلهٔ ٢ را ادامه دهيد

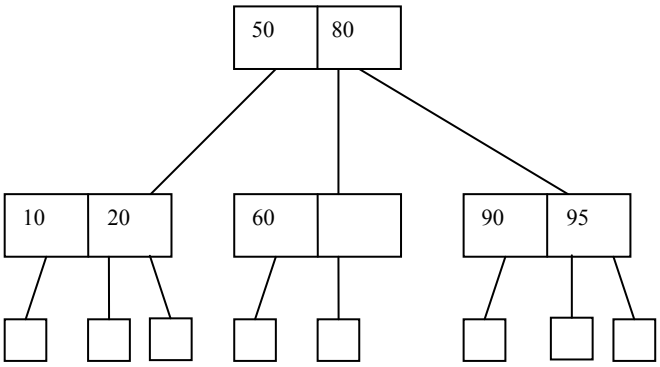
}

٣-اگر P داراي المان صفر است، p بايد ريشهٔ درخت باشد، فرزند چپ p، ريشهٔ جديد مي‌گردد و نود p حذف مي‌گردد.

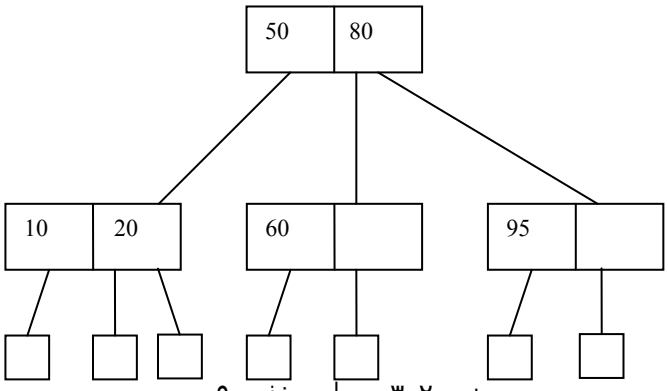
مثال : حذف کلید ۷۰ از درخت ۲-۳



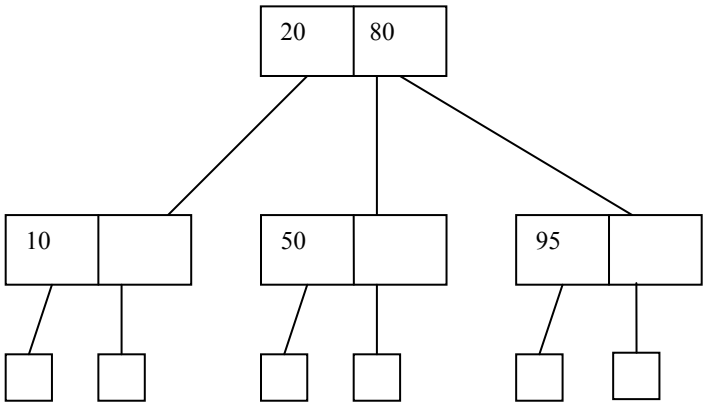
درخت ۲-۳ قبل از حذف



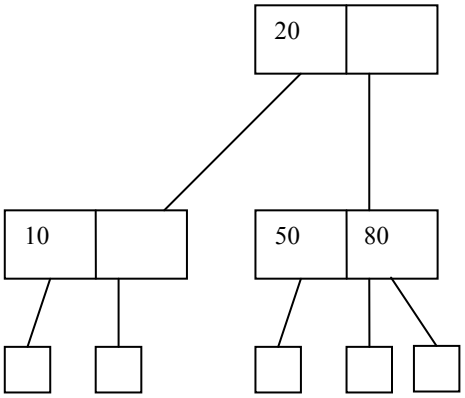
درخت ۲-۳ بعد از حذف کلید ۷۰



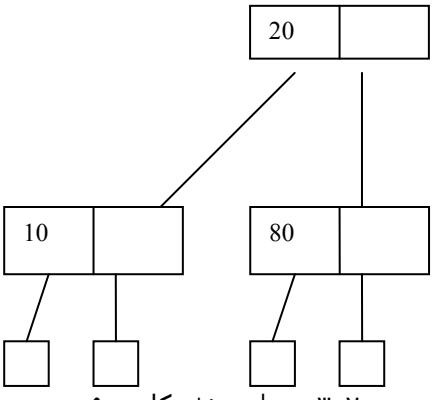
درخت ۲-۳ بعد از حذف ۹۰



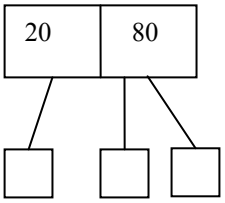
درخت ۲-۳ بعد از حذف کلید ٦۰



درخت ۲-۳ بعد از حذف ۹۵



درخت ۲-۳ بعد از حذف کلید ۵۰



درخت ۲-۳ بعد از حذف ۱۰

چرخش و ترکیب با پیچیدگي O(1) قابل انجام است، با توجه به تعداد چرخش و ترکیبهاي مورد نیاز، حذف از یك درخت ۲-۳ . با n نود از درجة (O(log n است.

۱۱-٤-درخت **M-Way Search Tree** و **BTree**

درخت‌هاي جستجوي بهینه که در قسمتهاي قبل مرور گردید، (درخت AVL و 2-3)، جستجو و درج و حذف را با پیچیدگي O(log N) انجام مي‌دهند. (N تعداد عناصر است)

اگر جدول به اندازة کافي کوچك باشد، مي‌توان آنرا در حافظة داخلي RAM نگهداري کرد، ولي اگر جدول بزرگ باشد، باید آنرا روي دیسك ذخیره نمود، ولي ساختارهایي که تابه حال بررسي گردید، مناسب جهت ذخیره سازي روي دیسك نیست. در زمان بازیابي، باید مسیر ریشه را تا نود دلخواه طي کنیم، اگر نود در ارتفاع h باشد، پیچیدگي جستجو از O(h) است.براي مثال اگر n=1000 باشد، آنگاه h=10 خواهد بود، بنابراین به ۱۰ دسترسي به دیسك نیاز خواهد بود.

مي خواهیم ساختاري بهینه براي ذخیره سازي روي دیسك بدست آوریم. معمولاً شاخص هاي فایل بصورت جدول نماد، ذخیره مي‌گردد و ورودیهاي جدول ممکن است، بسیار زیاد باشد و جدول بزرگ باشد.

۱۱-٤-۱- تعریف درخت جستجو با **m** مسیر **m-way search tree**

۱- ریشه داراي حداکثر m زیر درخت است و داراي ساختار زیر است:

$$n,A_0,(K_1,A_1),(K_2,A_2),\ldots,(K_n,A_n)$$

که A_i، $0\le i\le n<m$، اشاره گر به زیر درخت است و K_i کلید است. $(0\le i\le n<m)$

۲- $K_i<K_{i+1}$ براي $1\le i<n$

۳- تمام کلیدهاي زیر درخت A_i کمتر از کلید K_{i+1} و بیشتر از کلید K_i هستند. این شرط براي تمام نودهاي درخت صادق است.

٤- تمام کلیدهاي زیر درخت A_n بیشتر از K_n است، و تمام کلیدهاي A_0 کمتر از K_1 است.

۵- زير درخت A_i براي $(0\le i\le n)$ يك m-way search tree است.

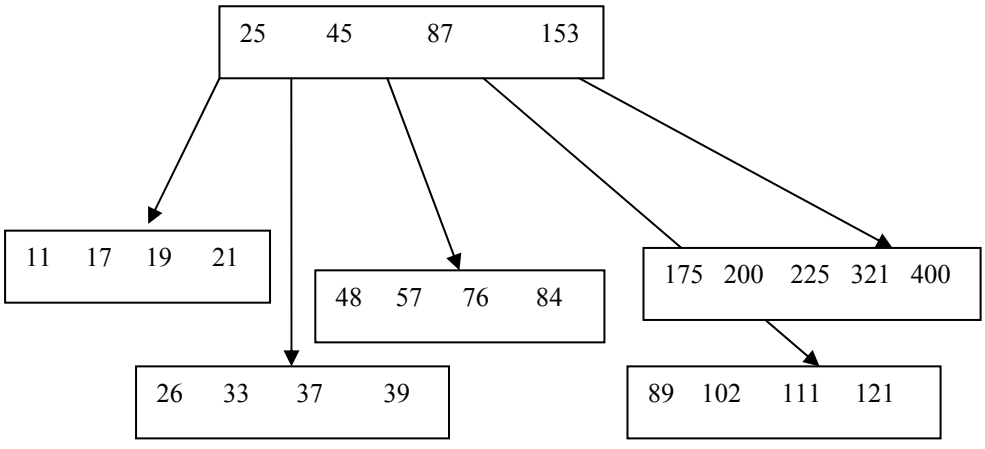
* دقت كنيد كه تعريف فوق بازگشتي است.

```
template <class Keytype>
class Mtree
{
    public:
        Mtree( );
        ~Mtree( );
        MNode<Keytype> *search(const Keytype &k);
    protected:
        MNode<Keytype> *Root;
}
```

الگوريتم جستجو در **m-way search tree**

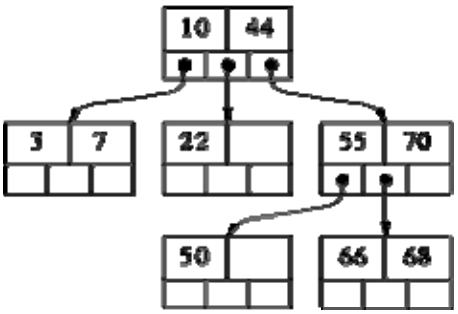
فرض كنيم مي‌خواهيم درخت T را براي كليد x جستجو كنيم، و فرض كنيد درخت T روي ديسك ذخيره شده باشد. جستجو با بازيابي ريشه از روي ديسك شروع مي‌شود و فرض كنيد $K_0=-\infty$ و $K_{n+1}=+\infty$ باشد، با جستجوي كليدهاي واقع در ريشه i بصورتي مشخص مي‌گردد كه:

$K_i\le x<K_{i+1}$ باشد، اگر $x=K_i$ باشد، پايان الگوريتم است. و اين نود جواب است. در غير اينصورت $x\neq K_i$، جستجو از زير درخت A_i ادامه مي‌يابد، اين عمل آنقدر ادامه مي‌يابد، تا نودي پيدا شود كه x در آن موجود باشد و يا x در درخت موجود نباشد.



مثالي از درخت m-way search tree

مثال : درخت m-way search tree با m=3



در درختي با ارتفاع h و درجة m، حداكثر تعداد نودها از فرمول زير محاسبه مي‌گردد :

$$\sum_{0\le i\le h-1} m^i = \frac{m^h - 1}{m-1}$$

در هر نود حداكثر m-1 كليد موجود خواهد بود و حداكثر تعداد كليدها برابر m^h-1 خواهد بود.

براي كارآيي بهينه، درخت حاصل از ساختار M-way search tree بايد موازنه باشد.

```
template <class Keytype>
MNode<Keytype> *Mtree<Keytype> :: search(const Keytype &k)
{
    K0 = -MAXKEY;
    for(MNode<Keytype> *p=Root, q=0;p;q=p,p=Ai)
    {
        get Node located at p on disk;//خواندن نود از ديسك
        Kn+1 = MAXKEY;
        obtain i such that Ki≤x<Ki+1 ;//i را چنان بيابيد كه
        if (x == Ki) return (p,i,1);//يافتن كليد در نود
    }
    return (q,i,0);//كليد داده شده در درخت موجود نيست
}
```

۱۱-٤-۲- ساختار **BTree**

تعريف: يك BTree از درجة m، يك m-way search tree است، كه يا تهي است و يا داراي شرايط زير است:

۱- ريشه داراي حداقل دو فرزند است.

۲- تمام نودها بجز ريشه و برگها داراي حداقل [m/2]، فرزند هستند.

۳- تمام برگ‌ها در يك سطح قرار دارند.

٤- تمامي نودها بغير از ريشه داراي حداقل [m/2] كليد و حداكثر m-1 كليد هستند. و ريشه بين ۱ الي m-1 كليد مي‌تواند داشته باشد.

```
template <class Keytype>
class BTree : public Mtree<Keytype>
{
    public:
        //جستجو از كلاس سطح بالاتر به ارث مي‌رسد
        bool insert(const Keytype &x);
        bool delete(const Keytype &x);
}
```

توجه كنيد كه درخت 2-3 يك درخت Btree با درجة ۳ است.

تعداد كليدها در **BTree**

يك Btree با درجة m، مه برگ‌هايش در سطح L+1 هستند، داراي حداكثر m^L-1 كليد مي‌باشند و حداقل تعداد كليد ها $2[m/2]^{L-1}$ است.

درج در **BTree**

الگوريتم درج در Btree، الگوريتم توسعه يافتة درج در درخت‌هاي 2-3، است. براي درج ابتدا درخت را جستجو مي‌كنيم و نود برگي، مثل P را پيدا مي‌كنيم كه كليد بايد در آن درج گردد.

اگر نود داراي m كليد باشد، آنگاه نود بايد تجزيه شود، (Split) در غير اينصورت P، روي ديسك نوشته مي‌شود و درج كامل است.

براي تجزية نود فرض كنيد نود P بصورت زير است:

$m,A_0,(K_1,A_1), (K_2,A_2), \ldots,(K_m,A_m)\ , K_i<K_{i+1}\ , 1\le i<m$

اين نود به دو نود P و Q بصورت زير تجزيه خواهد شد.

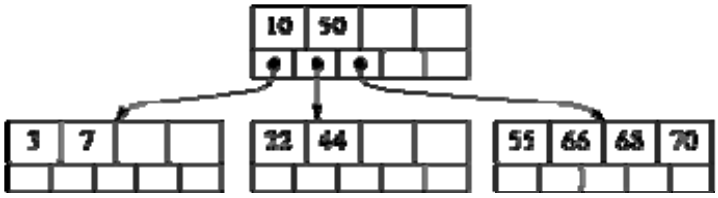
NODE P = $[m/2]-1\ , A_0,(K_1,A_1), (K_2,A_2), \ldots,(K_{[m/2]-1},A_{[m/2]-1})$

NODE Q= $m-[m/2]\ , A_{[m/2]},(K_{[m/2]+1},A_{[m/2]+1}),\ldots, (K_m,A_m)$

كليد باقي ماندة $K_{[m/2]}$ و يك اشاره گر به نود Q، $(K_{[m/2]},Q)$، در نود والد P درج مي‌گردد، و سپس P و Q روي ديسك ذخيره مي‌گردد.

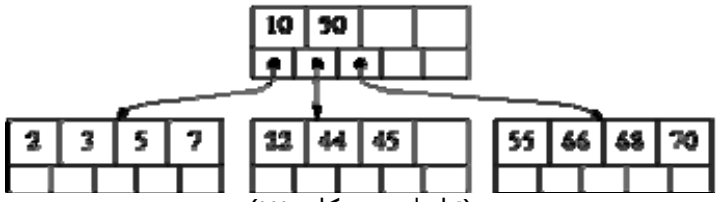
اين عمل ممكن است تا سطح ريشة درخت در درخت 2-3، ادامه يابد، و اگر ريشة درخت شكسته شود، آنگاه ارتفاع درخت يك واحد اضافه مي‌شود.

مثال : Btree از درجة ٥

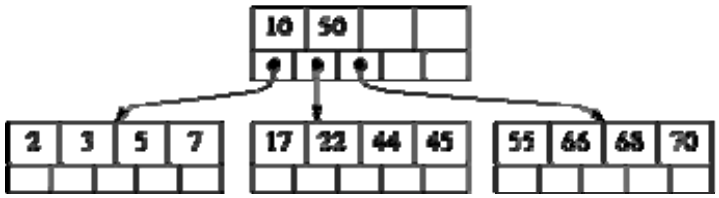



```
template <class Keytype>
bool Btree<Keytype> :: insert(const Keytype &x)
{
   درج نود در محل خودش//
   بازسازي درخت طبق الگوريتم //
}
```

مثال: درج نود در درخت Btree از درجة ٥



(قبل از درج کلید ١٧)

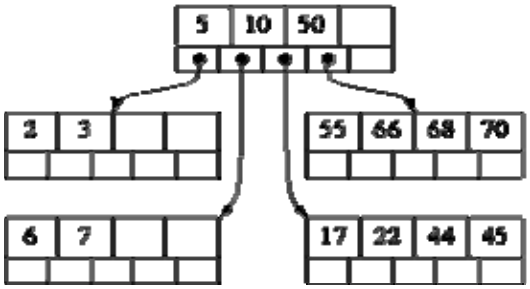


(بعد از درج کلید ١٧)

بدليل آنكه نود مياني، پس از درج داراي سرريز نيست، عمل شكستن لازم نيـست و درج كامل است.

سپس به درخت فوق كليد ٦ را درج مي‌كنـيم، نـود سـمت چـپ داراي سـرزيز شـده و شكسته مي‌شود.

- Left = [2 3]
- Middle = 5
- Right = [6 7]

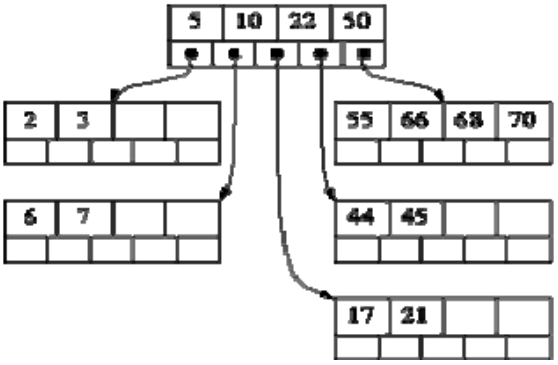


قسمت‌هاي چپ و راست خود تشكيل نود جديد مي‌دهند و عنصر مياني به والد فرستاده مي‌شود.

به دليل اينكه ريشه پس از پذيرفتن ۵، داراي سرريز نيست، درج كامل است.

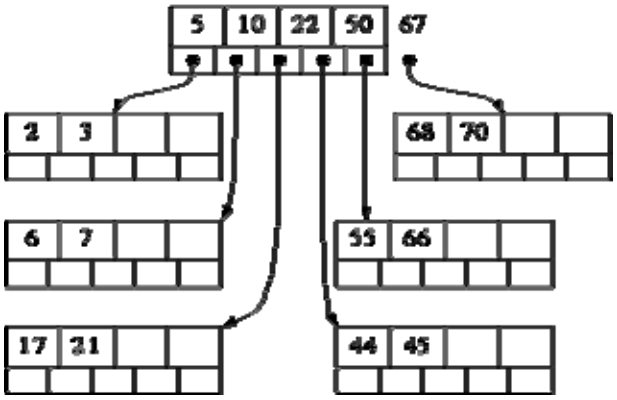
حال فرض كنيد مي‌خواهيم كليد ۲۱ را در درخت درج كنيم:

در اين حالت نيز ريشه سرريز نشده.



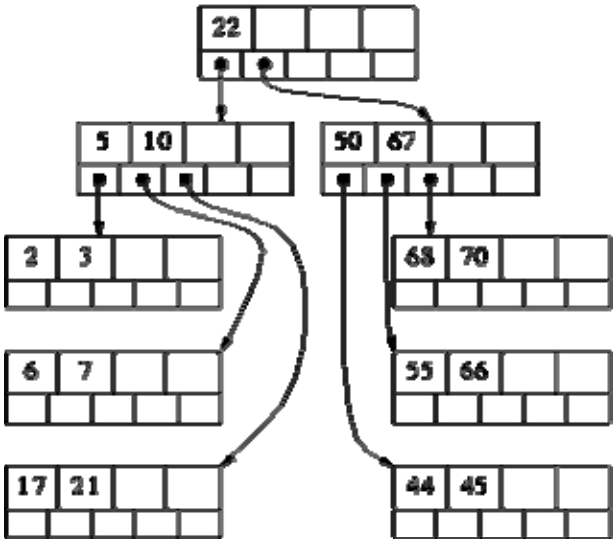
درج كليد ٦٧:

اين درج باعث تجزيه شده سمت راست ترين نود مي‌شود و سپس ريشه نيز داراي سرريز مي‌گردد مراحل تجزيه و درج بصورت زير است:



حال بايد ريشه نيز شكسته شود:

- Left = [5 10] (along with their children)
- Middle = 22
- Right = [50 67] (along with their children)



حذف از **Btree**

الگوريتم حذف نيز، توسعة الگوريتم حذف، در درخت‌هاي 3-2 است.

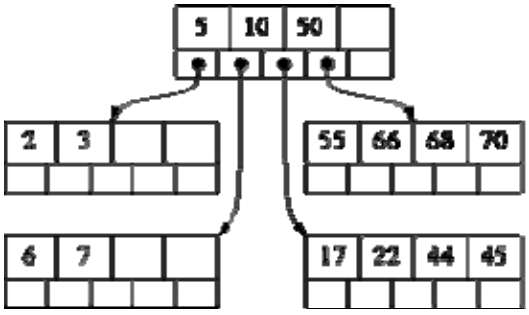
ابتدا كليد x، در درخت جستجو مي‌گردد، اگر x در نودي غير برگ مثل z است، محل كليد x با يك كليد از نودهاي برگ جايگزين مي‌گردد، فرض كنيد، x كليد i ام در نود z باشد، (K_i=x)، سپس x مي‌تواند توسط كوچكترين كليد در زير درخت A_i، و يا با بزرگترين كليد در A_{i-1}، جايگزين گردد، در هر دو حالت، حذف به حذف از يك برگ تبديل مي‌گردد.

حالات مختلف حذف كليد از برگ:

١- اگر P ريشة درخت باشد و حداقل يك كليد براي آن باقي مانده باشد، آنگاه آنرا در ديسك ذخيه مي‌كنيم و گرنه يك درخت تهي تشكيل مي‌گردد.

٢- اگر P ريشه درخت نباشد حذف را به حذف از برگ تبديل مي‌كنيم و اگر P، برگ است آنگاه نبايد كليدهايش از حداقل مقدار ممكن كمتر شود. (يادآوري: ريشه حداقل بايد يك كليد و بقية نودها حداقل (m-1)/2 كليد داشته باشند)، اگر كليد ها از اين مقدار كمتر شود گوييم پاريز اتفاق افتاده است و بايد نود اصلاح گردد.

مثال : حذف كليد ٦ از درخت زير با درجة ٥



با حذف كليد ٦، فقط كليد ٧ در نود باقي مي‌ماند و نود پاريز مي‌شود، پس بايـد اصـلاح شود، استراتژي اصلاح نود در زمان پاريز، قرض كردن كليد از نودهاي مجاور اسـت (تركيـب نودها)

نود را با همسايهٔ پر جمعيت تر تركيب مي‌كنيم و كليد بين ايـن دو زيـر درخـت در نـود والد نيز با اين كليدها تركيب مي‌شود با توجه به شكل فوق كليدهاي [7,10,17,22,44,45] بدست مي‌آيد.

حالاتي كه ممكن است پس از تركيب نود بوجود آيد بصورت زير است:

الف) نود والد يك هميشه يك كليد را قرض مي‌دهد.

ب) نودي كه پاريز شده بود دقيقـاً داراي (m-1)/2-1 كليـد اسـت كـه در تركيـب نـود شركت مي‌دهد.

ج) نود همسايه ممكن است تعداد كليدهايش بين (m-1)/2 و m-1 باشد.

الگوريتم داراي دو حالت بر اساس اينكه همسايه دقيقـاً (m-1)/2 كليـد يـا بيـشتر دارد، مي‌باشد.

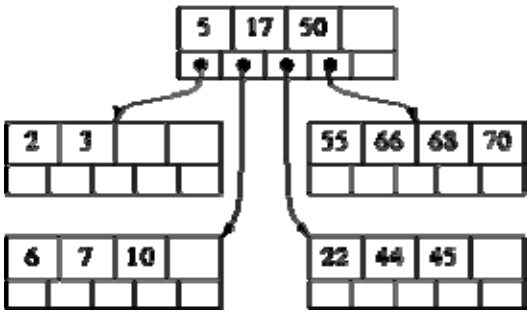
١-فرض كنيد همسايه بيش از (m-1)/2 كليد دارد:

بنابراين پس از تركيب كليد ها تعداد كليد ها حتمـاً بيـشتر از ((m-1)/2 - 1) + 1 + (1)/2) و قطعاً بيش از m-1 نيز خواهد بود. بنابراين m عدد كليد و يا بيشتر خواهد داشت.

نود تركيب شده را به سه قسمت چپ، مياني و راست تقسيم خواهد شد، بدليل آنكه نود تركيبي بيش از m كليد دارد، تضمين خواهد شد كه نـود چـپ و راسـت حـداقل (m-1)/2 كليد داشته باشند، بنابراين اين نودها درست هستند و نيازي بـه اصـلاح آنهـا نيـست. در ايـن حالت مقداري كه از والد قرض گرفته‌ايم، با مقدار مياني جايگزين مي‌كنيم و نودهـاي چـپ و راست را فرزندان آن قرار مي‌دهيم، بنابراين سايز ريشه عوض نمي شود و اين پايان الگـوريتم

است.

در مثال فوق ۱۷ كليد مياني نود تركيبي و [6,7,10] كليدهاي نود چپ و [22,44,45] كليدهاي نود راست خواهد بود، پس ۱۰ با ۱۷ جايگزين مي‌گردد و درخت زير بدست مي‌آيد:



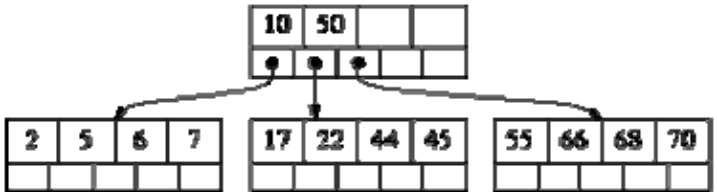
۲- فرض كنيد، نود همسايه دقيقاً (m-1)/2 كليد، داشته باشد:

بنابراين تعداد كليدها در نود تركيب شده برابر خواهد بود با

$$1 + ((m-1)/2 - 1) + ((m-1)/2) = (m-1)$$

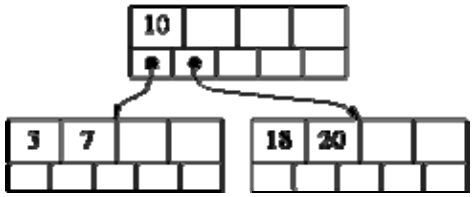
اين تعداد كليد تشكيل يك نود صحيح ميدهد، بنابراين، كليدي كه از طرف ريشه (والـد) قرض داده شده بود از ريشه حذف مي‌گردد.

براي مثال فرض كنيد در درخت اوليه كليد ۳ حذف گـردد، آنگـاه پـاريز اتفـاق افتـاده و تركيب انجام مي‌شود ([7 6 5 2]) كه دقيقاً داراي m-1 كليد است و نتيجه درختـي بـصورت زير است:

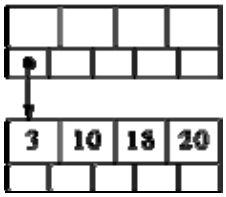


بسيار مهم است كه ريشه بيش از يك كليد داشته باشد، فرض كنيد ۵ تنها كليـد واقـع در ريشه بود، بنابراين پاريز اتفاق مي‌افتد و عمليات تركيب نهايتاً تا ريشة درخت جلو مي‌رود.

حال فرض كنيد ريشة درخت فقط يك كليد داشته باشد و پاريز اتفاق افتد :



در درخت فوق كليد ٧، را حذف مي‌كنيم، پس از حذف پاريز اتفاق مي‌افتد و تركيب انجام شده و نود [20 18 10 3]، بدست مي‌آيد كه دقيقاً داراي ٤ كليد است، و يك نود صحيح مي‌باشد ولي ريشه ديگر داراي كليدي نيست. (به شكل زير توجه كنيد)



طبيعي است، كه در چنين حالتي، ريشة تهي قبلي حذف شده و نود جديد ريشة جديد درخت خواهد بود.

منابع

منابع

A. Habed. (2001). Data Structures and Algorithms, University of Windor.

D. Mehta. (1994). The C Data Structures.

J. Morris. (1998). Data Structures and Algorithms.

Robert C. Holte (2002). Data Structure Course, MacDonald Hall, University of Ottawa.

S. Sahni. (1995). Fundamentals of Data Structures in C++, E. Horowitz.