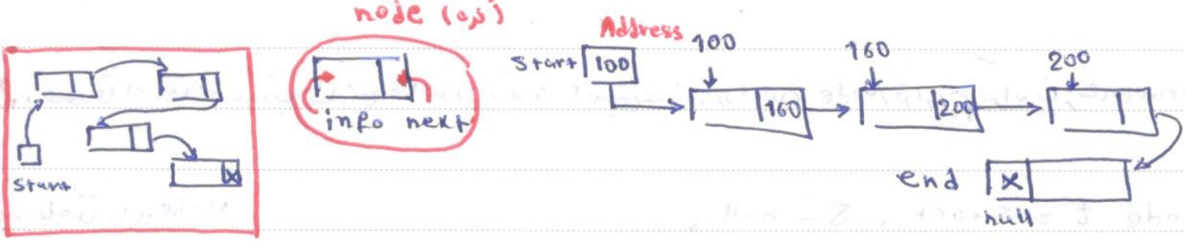
Subject: ساختمان داده — جلسه چهارم

Year. ۹۵ Month. ۱۲ Date. ۱۹

link list ها :

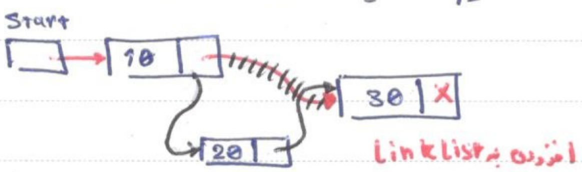
مجموعه‌ای از عناصر که در حافظه پشت سر هم نیستند و بنابراین علاوه بر ذخیره عنصر، کمکی برای ذخیره آدرس آن نیز داریم

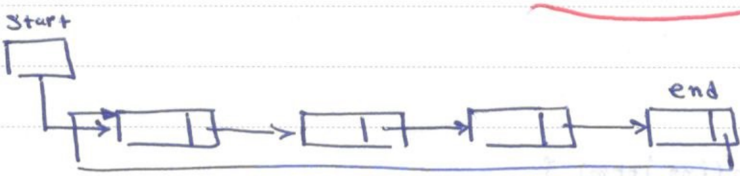


لینک لیست یک طرفه متصل می‌شوند (به null می‌رسند) ★

linklist : مجموعه از عناصر که یک ساختار دارند الزاما پشت سر هم نیستند و سرعت دسترسی به آنها یکسان نیستند و تعداد آنها ثابت نیست.

هنگامی لینک لیست استفاده می‌شود که ندانیم چه تعداد عنصر داریم. در لینک لیست ما می‌توانیم به راحتی عنصر اضافه کنیم بدون آنکه نیاز باشد عناصر را به عقب یا جلو shift (حرکت) دهیم. بنابراین اگر نیاز به حذف یا اضافه عناصر شده یا عنصری کم کنیم مجبور می‌شویم که از لینک لیست‌ها استفاده کنیم.





ساختار لینک لیست مدور

انتها به ابتدا اشاره می‌کند

این ساختار در اجرای هم زمان برنامه‌ها در سیستم عامل‌ها اجرا می‌شود.

ایجاد linklist ها :

۱) تعریف Node : ایجاد یک کلاس

```
Class Node {
    public int info ;
    public Node Next ;
}
```

جلسه چهارم

1. دریافت Node از refrence

```
Node p = new Node();
if (p == null) { return false; }
p.info = item
```

چنانچه حافظه پر باشد p برابر null برگردد. // item مقداردهی

در صورتی که نیاز به مرتب سازی عناصر نباشد با بستن عنصر را اولین Node قرار دهیم چرا نیاز به traverce نداریم.

2. یافتن جای Node (traverce)

```
Node t = Start , S = null ;
while (t != null && t.info < item) {
    S = t ;
    t = t.next ;
}
```

3. افزودن Node

```
p.next = t ;
if ( S == null) { Start = p ; }
else { S.next = p ; }
```

4. تکمیل کلاس

```
class LinkedList {
  0.
  Node Start = null ;
  Public bool AddItem(int item) {
    1.
    2.
    3.
    return true;
  }
}
```

2, 3. درصورتی که بخواهیم linkedlist مرتب شده باشد

3'. آن sort شده نخواهیم نیازی به مرحله 2 نبوده و

```
p.next = Start ;
Start = p ;
```

Delete item

2. پیمایش عنصر (traverce)
4. آیا عنصر پیدا شد ــ در صورتی که عنصر پیدا نشود

```
if ( t == null || t.info > item ) { return false; }
```

5. حذف عناصر وسط

```
if (S != null) { S.next = t.next; }
else { Start = t.next; }
```

برای حذف اولین عنصر

6. تکمیل کد

```
Public bool DeleteItem (int item) {
    2.
    4.
    5.
    return true;
}
```

linkedlist

Find item

```
public bool Find item ( int item) {
    2.
    4.
    return true;
}
```

```
return ( t != null && t.info == item );
```

می‌توان جهت حرفه‌ای تر شدن برنامه این را بکار برد !

print item

```
public void Print ( ) {
    Node t = Start;
    while (t != null) {
        System.out.println (t.info);
        t = t.next;
    }
}
```

* جلسه چهارم *

تمارین *

۱) Delete , Find , Print برای linked list پیاده شده انجام دهد.

۲) linkedlist عدد بپذیرید.

تمارین linkedlist ۲۲

۱) برنامه ای بنویسید که به عنوان ورودی شروع یک linkedlist را گرفته و آنرا معکوس نماید.

۲) برنامه ای بنویسید که شروع یک linked list را بگیرد و Node های که مقدارشان زوج است را حذف کند.

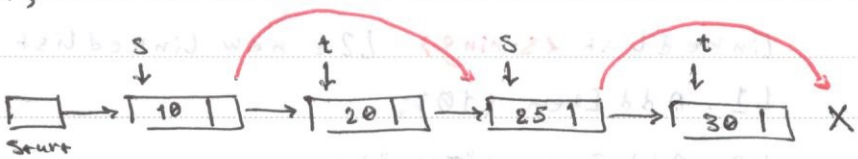
۳) برنامه ای بنویسید که به عنوان ورودی شروع یک linkedlist را گرفته و Node های پوزیشن زوج دارند را حذف کند.

۴) تابعی بنویسید که اشاره گری به شروع یک linkedlist مرتب شده را بگیرد و آنرا با الگوریتم bubblesort مرتب کند.

پاسخ ۳)

```
public void DelNode (Node Start) {
    Node S = Start, t;
    if ( S != Null ) {
        t = S.next;
    }
    while (t!= null && S!=null) {
        S.next = t.next;
        S = S.next;
        t = S.next;
    }
}
```

```
Node S = Start, t;
while( S != null && S.next
!= null ) {
    t = S.next;
    S.next = t.next;
    S = S.next;
}
```



ایجاد حالت General برای کلاس ها (Generic) حذف از استثنا

با این حالت دیگر لازم نیست کلاس فقط نوع خاصی از داده مثل int بگیرد بلکه می‌تواند هر نوع دیگر را دریافت کند.

بدین منظور کافی است در جلوی نام کلاس <T> و به جای نوع متغیرها T را قرار دهیم.

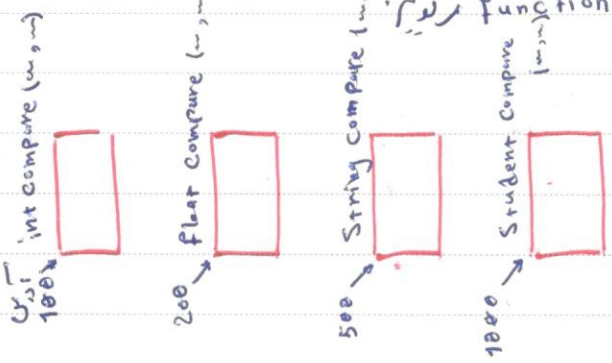
```
Class linkedlist <T>
    Class Node {
        public T info;
        public Node Next;
    }
    Node Start = NULL;
    public bool AddItem (T item) {
        ---
    }
}
public Static void main () {
    linkedlist <int> l1 = new linkedlist <int> ();
    linkedlist <string> l2 = new linkedlist <string> ();
    l1.AddItem (10);
    l2.Add Item ("This");
}
```

Delegate چند مثال

برای حل مشکل نوع داده، DataType ها می‌توانیم به جای نوع متغیرها، T قرار دهیم.

```
T []A = new T [n];
for(int i = 0 ; i < A.length -1 ; i++){
    for( int j = 0 ; j < A.length - i -1 ; j++){
        if( A[i] > A[j] ){
            T Temp = A[i];
            A[i] = A[j];
            A[i+1] = Temp;
        }
    }
}
```

در این ساختار می‌توانیم توابع کلی با prototype مانند الگوریتم اما با ای متفاوت داشته باشیم و در جای مورد نیاز جایگذاری کنیم. در C++ به آن function pointer می‌گویند.

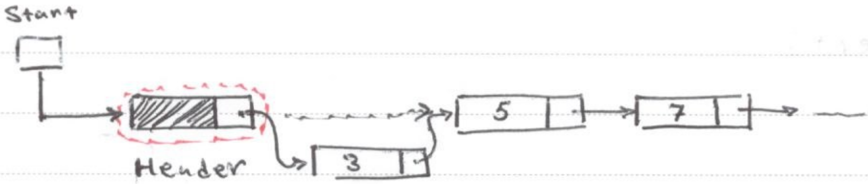


چنین ساختاری در زبان Java توسط interface قابل پیاده سازی است.

```
interface Icompare {
    public int Compare (Student S1 , Student S2 ) {
        ---
    }
}
```

```
class X : Icompare {
    ---
    int Compare ( student S1 , student S2 ) {
        return ( S1.id > S2.id );
    }
}
class Y : Icompare {
    ---
    int compare ( student S1 , student S2 ) {
        return ( S1.Name > S2.Name );
    }
}
```

برای کاهش if ها در linkedlist ها یک Node اضافی می‌سازیم و Node های دیگر را در ادامه آن قرار داده که به این Node اولیه Node Header یا سرلیست گفته می‌شود. در آن data ای نمی‌ریزیم اما به ما این امکان را می‌دهد تا if ها را حذف کنیم.



برای داشتن linked list هدردار به صورت مرتب عمل می‌کنیم،

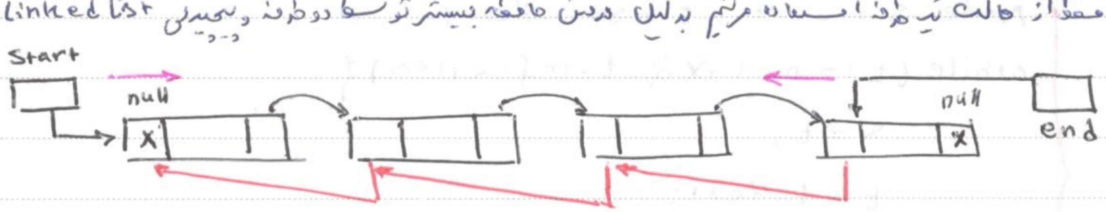
```
class linkedlist {
    class Node {
        public int info;
        public Node next;
    }
    Node Start = new Node( );
    public boolean Additem(int item) {
1       Node p = new Node( );
        if (p == null) {
            return false;
        }
        p.info = item;
2       Node s = Start , t = Start.Next;
        while (t != null && t.info < item) {
                s = t;
                t = t.Next;
        }
3       p.Next = t;
        s.Next = p;
4       return true;
    }
```

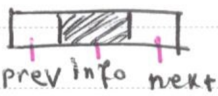
(In the original, "boolean" is written above a struck-through "bool".)

```
public boolean Delitem (int item) {
    if (t == null || t.info > item)
        return false;
    S.Next = t.Next;
    return true;
}
public void Print () {
    Node t = Start.Next;
    while (t != null) {
        Console.Writeline(t.info);
        t = t.Next;
    }
}
```

« linkedlist دوطرفه »

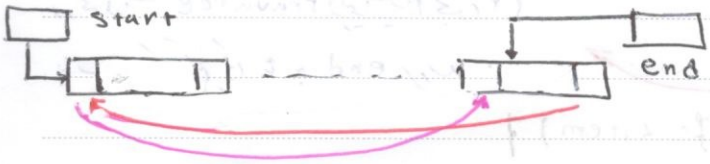
عملیات travarce در linkedlist یک طرفه فقط از یک سو انجام میگیرد. گاهی وقت ها نیاز است تراورس دو طرفه داشته باشیم مثل زمانی که میخواهیم یک editor متن داشته باشیم. معمولا در editor داریم که تعداد آنها مشخص نیست بنابراین باید از linked list ها استفاده کنیم. در editor ها با اینتر به خط بعدی میرویم که میتوانیم درآن متنی را وارد کنیم. (linked list های از رشته ها) اگر از linkedlist یک طرفه استفاده کنیم فقط میتوانیم به خط بعدی برویم و امکان ویرایش خطوط قبلی وجود نخواهد داشت. ساختار به این صورت است. linked list دو طرفه فقط زمانی بکار میرود که بخواهیم travarce از دو طرف داشته باشیم در حالت عادی معمولا حالت یک طرفه استفاده میکنیم بدلیل گرفتن حافظه بیشتر توسط دو طرفه و پیچیدگی linked list دو طرفه



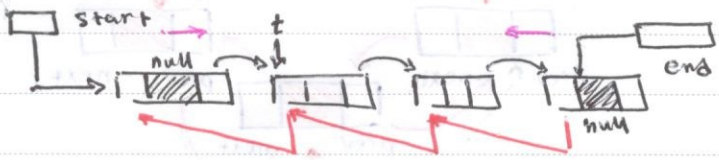


جلسه ششم

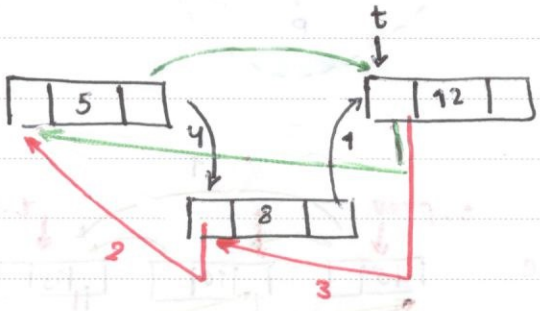
مرتوال همانند قبل linked list هست داشت و با بستی آخرین next به اولین prev و اولین prev به آخرین prev اشاره کند.



linked list دوطرفه حرکت تکلیف دار



افزودن یک عنصر در لیست دوطرفه حرکت دار



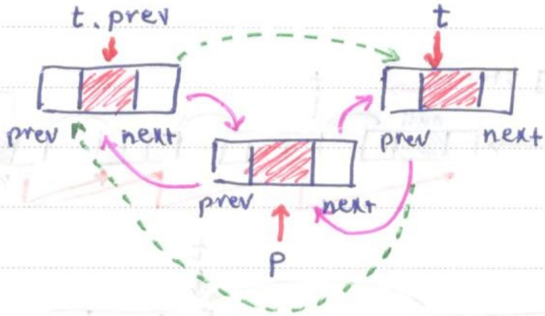
1. p.next = t;
2. p.prev = t.prev;
3. t.prev.next = p;
4. t.prev = p

ساختار لیست دوطرفه حرکت دار

```
Class Dlinked {
    class Node {
        public int info;
        public Node prev;   ← اضافه شد
        public Node next;
    }
    Node start, end;
    public Dlinked() {
        start = new Node();
        end = new Node();
        start.next = end;
        end.prev = start;
    }
```

لینک لیست دوطرفه به null نرسد بنابراین در عملیات travarce (پیمایش) (بخش ۲) بایستی چک کنیم که t به end نرسد.

```
public bool Additem (int item) {
1 { p.info = item;
  { Node t = start.next;
2 { while (t != end && t.info < item)
  {     t = t.next;
  { p.next = t;
3 { p.prev = t.prev;
  { t.prev.next = p;
  { t.prev = p;
    return true;
}
```



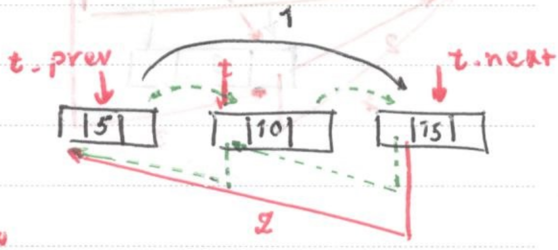

```
public bool Delitem (int item) {
```

② travarce یکسری لینک لیست دوطرفه هم رو تیتر بدار

```
4 { if (t == null || t.info > item)
  {     return false;
```

یعنی item پیدا نشد.

```
5 { t.prev.next = t.next;     ①
  { t.next.prev = t.prev;     ②
    return true
}
```



اگر true بود از start به سمت end
اگر false بود از end به سمت start

```
public void print (bool Dir = true) {
    if (Dir == true) {
        Node t = start.next;
        while (t != end) {
            Console.WriteLine (t.info);
            t = t.next;
        }
}
```

```
        else {
            Node t = end.prev;
            while (t != Start) {
                Console.WriteLine(t.info);
                t = t.prev;
            }
        }
    }
```

تمرین

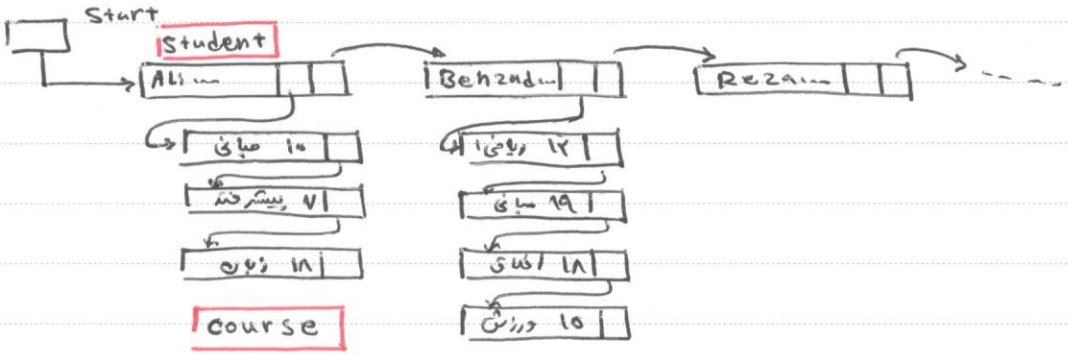
۱. linkedlist دو طرفه بدون هدر و تیل را بنویسید.

۲. linkedlist دو طرفه هدر را بنویسید.

multi linked

ساختار Node را همیشه به یک شکل استاندارد در می‌آوریم، حتی اگر linkedlist ما در بعضی وقت‌ها لازم است.

مثل ساختار دانشجویان با واحدهای درسی:

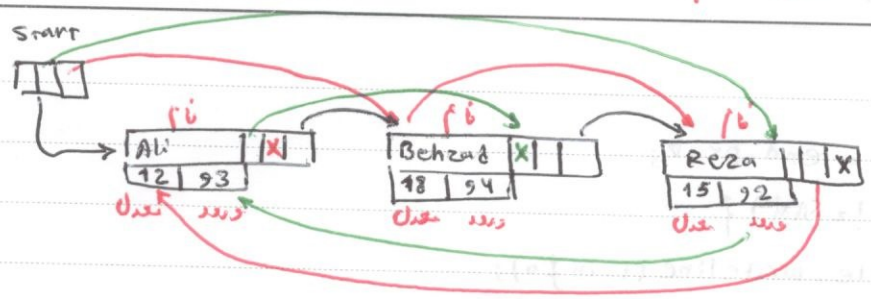


در multi linked یک Node می‌تواند به بیش از یک Node اشاره کند. همچنین multi linked با یک ساختار Node

نیز می‌تواند داشته باشیم.

۸

حتی Node ها می‌توانند از ساختارهای متفاوت باشند



- مرتب براساس نام
- مرتب براساس تعدل
- مرتب براساس سال ورود

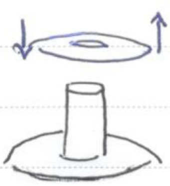
در sort ها از تکنیک جدیدی استفاده شده است. بنابراین به تعداد next ها sort خواهیم داشت.

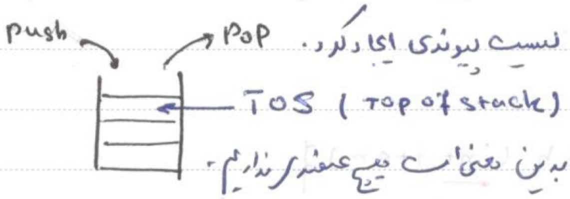
```
class Node {
 public Student info;
 public Node nextName;
 public Node nextAvg;
 public Node nextYear;
}
```

« Queue & stack » پشته و صف

stack ساختار داده‌ای است که عناصر فقط از یک سو می‌توانند به آن اضافه و تنها از همان سمت می‌توانند خارج شوند. مثل یک دسته CD. به اضافه شدن push و خارج شدن (حذف شدن) pop می‌گویند.

این ساختارها را می‌توان با آرایه یا لیست پیوندی ایجاد کرد.





Last In First out

LIFO

در ابتدا TOS را ‎-1‎ می‌گذاریم که به این معنی است هیچ عنصری نداریم.

در ادامه آن را ++ می‌کنیم.

اگر به انتهای آرایه رسیدیم

```
push :  if (TOS == Array.Length-1)
            return false;
        TOS++;
        Array[TOS] = item ;
        return True;
```

5	10	3		...
0	1	2		

```
pop :   if (TOS == -1)        →  اگر پشته ما خالی بود
            return false;
        item = Array[TOS];
        TOS--;
        return True;
```

فضای آرایه محدود است و TOS تا مقدار محدودی می‌تواند افزایش یابد و در صورت افزایش بیش از حد overflow رخ می‌دهد.

بنابراین قبل از انجام عملیات چک می‌کنیم.

⚠ در ساختار stack آخرین عنصر وارد شده اولین عنصر خارج شده می‌باشد.

```
class stack {
    int [ ] Array;
    int TOS;
    public stack(int size){
        Array = new int [size];
        TOS = -1;
    }
    public bool push (int item){
        if (TOS == Array.Length - 1)
            return false;
        TOS ++;
        Array [TOS] = item;
        rerurn true;
    }
    public bool pop (ref int item) {
        if (TOS == -1)
            return false;
        item = Array [TOS];
        TOS --;
        return true;
    }
}
```

دیتا تایپ قابلیت تعریف به هر چند دلخواه را دارد.

ساختار Stack با آرایه

تنظیم سایز آرایه //

چک کردن امکان آرایه //

pass by refrence

نحوه ی فرستادن بازگردادن در C#

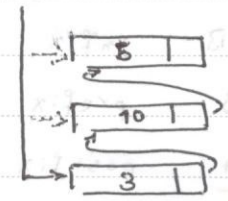
در Java به جای یک object بفرستیم

چک کردن استثنا آرایه //

ساختار Stack با استفاده از آرایه

بدلیل کمبودن آرایه، مرتبا با لینک لیست نیز ساختار stack داشت.

در linkedlist هنگام push یک عنصر درج شده و بنابراین اضافه می‌شود.



این ساختار زمانی overflow می‌شود که نتوانیم حافظه بگیریم.

در هنگام شروع کار TOS را برابر null می‌گذاریم.

ساختار Stack و linked list

```
Class Stack ( ) {
    Class Node ( ) {
        public int info;
        public Node next;
    }
    Node TOS = null;
    public bool push (int item) {
        Node p = new Node ();
        if (p == null)
            return false;
        p.info = item;
        p.next = TOS;
        TOS = p;
        return True;
    }
    public bool pop (ref int item) {
        if (TOS == null)
            return false;
        item = TOS.info;
        TOS = TOS.next;
        return True;
    }
}
```

جلسه هفتم

یکی از موارد کاربرد Stack و ساختارهای سبانی و خروجی‌ها.

(A + B) * C / (D - F) * g / h)

A+B infix

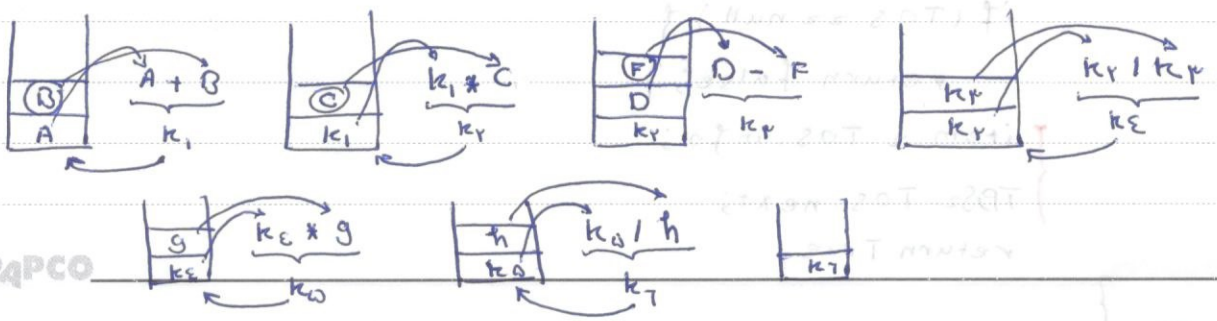
+ AB prefix

AB+ postfix

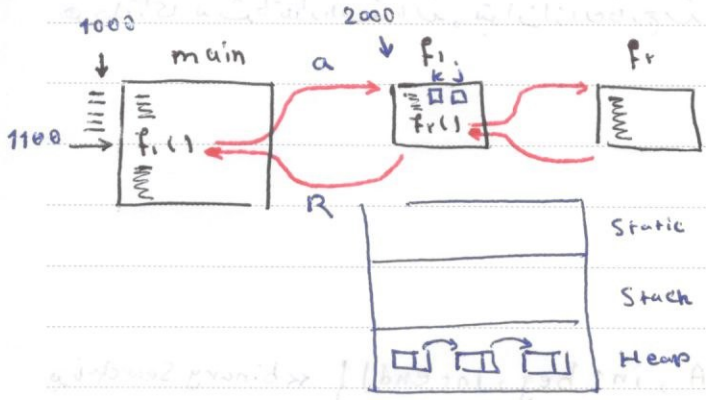
Token	Stack	postfix
(	(	
A	((	A
+	((+	
B		AB
)	(	AB+
*	(*	
C		AB+C
/	(/	AB+C*
(	(/(	
D		AB+C*D
-	(/(-	
f		AB+C*DF
)	(/	AB+C*DF-
*	(*	AB+C*DF-/
g		AB+C*DF-/g
/	(/	AB+C*DF-/g*
h		AB+C*DF-/g*h
)		AB+C*DF-/g*h/

در صورتی که به "(" برسیم و تا پرانتز باز مربوط به آن همه چیز میرود.

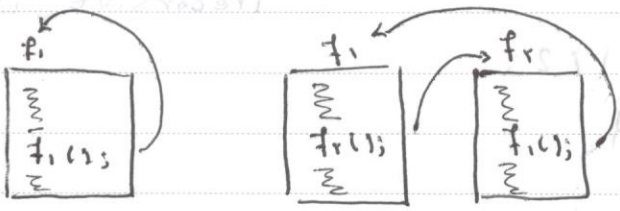
در صورتی که به علامت برسیم، چنانچه اولویت علامت کنونی بیشتر یا مساوی علامت بالای مانده در Stack باشد، میرود به جای آن در نشیند.



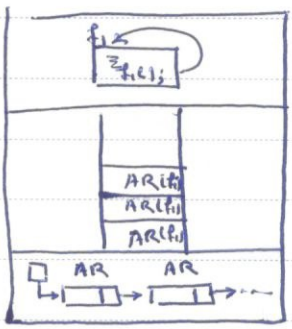
یکی دیگر از کاربردهای Stack در function call ها، عملیات back tracking (سیرهای بازگشتی) است.



در توابع بازگشتی:



تابع بازگشتی فاکتوریل




```
int fact (int n) {
    if (n < 2)
        return 1;
    return n * fact(n-1);
}
```

سوال انتهای درسی: a^b را بصورت جمع بنویسید (با تابع بازگشتی، میتوان با تابع بیانی نیز نوشت.)

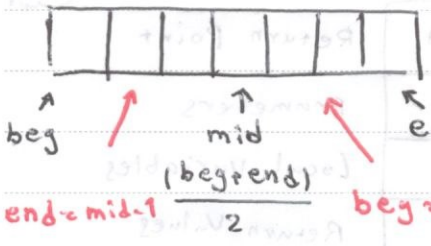
$a^b = \underbrace{a \times a \times a \times \dots \times a}_{b}$

$\downarrow$

$\underbrace{a + a + a + \dots + a}_{a}$

الگوریتم Binary Search

هر حلقه‌ای که شرط آن دارای شمارنده باشد، می‌توان آن را به صورت recursive نیز نوشت.




```
int bsearch(int item, int []A, int beg, int end){
    while (beg <= end){
        int mid = (beg+end)/2;
        if (item == A[mid])
            return mid;
        else{
            if (item < A[mid])
                end = mid-1;
            else
                beg = mid+1;
        }
    }
    return -1;
}
```

برنامه binary search به صورت recursive (بدون while)

برنامه binary search به صورت recursive

```
int bsearch(int item, int []A, int beg, int end){
    if (beg > end)
        return -1;
    int mid = (beg+end)/2;
```

```
    if (item == A[mid])
        return mid;
    else {
        if (item < A[mid])
            bsearch(item, A, beg, mid-1);
        else
            bsearch(item, A, mid+1, end);
    }
```

الگوریتم quick sort

current

7, 3, 5, 11, 2, 8, 6 ← right, left

6, 3, 5, 11, 2, 8, 7 ← right, left

6, 3, 5, 7, 2, 8, 11

6, 3, 5, 2, 7, 8, 11

سمت راست تمام شد چرا که اعداد قبلی کوچکتر و بعدی بزرگتر هستند

quick sort برنامه

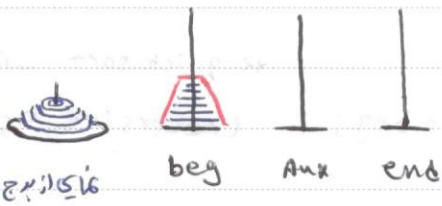
(recorsive بصورت)

```
Void qSort(int[] A, int beg, int end) {
    int left = beg, right = end, current = beg;
    while (left <= right) {
        while(current < right && A[current] <= A[right])
            right--;
        if (current < right) {
            int temp = A[current];
            A[current] = A[right];
            A[right] = temp;
            current = right;
        }
```

```
while(current > left+ && A[current+] >= A[left+])
        left++;
if(current > left+){
        int temp = A[current+];
        A[current+] = A[left+];
        A[left+] = temp;
        current = left+;
    }
}
if(beg < current+ - 1)
    qsort(A, beg, current+ - 1);
if(current+ +1 < end)
    qSort(A, current+ +1, end);
}
```

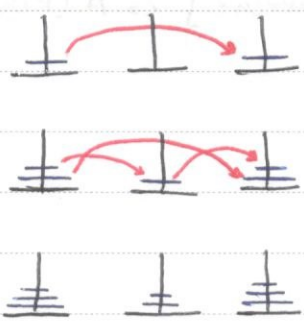
الگوریتم برج هانوی

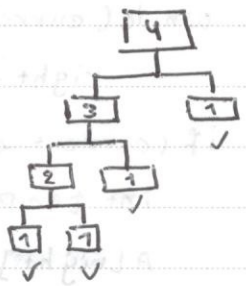
این الگوریتم با افزایش تعداد دیسک ها که باعث سخت شدن الگوریتم میشود و ما میتوانیم به الگوریتم های دیگر کمک کند تقسیم کنیم.



برج هانوی

$n \rightarrow 2^n - 1$





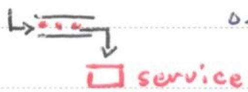
جلسه هشتم

برنامه برج هانوی recorsive :

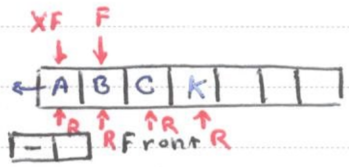
```
void tower(int n, char beg, char aux, char end){
    if (n==1){
        console.writeline(beg + "→" + end);
        return;
    }
    tower(n-1, beg, end, aux);
    console.writeline(beg + "→" + end);
    tower(n-1, aux, beg, end);
}
```

این الگوریتم هانوی recorsive آسان‌تر و مختصرتر در حالت عادی بسیار طولانی خواهد بود.

صف (Queue) :



ساختار داده‌ای که عناصر از یک طرف وارد و از طرف دیگر خارج می‌شوند. یک service دهنده همانند server نیز در انتها وجود دارد که این روند را کنترل و سرویس دهی کند. این ساختار هم با لینک لیست و هم با آرایه قابل پیاده سازی است.



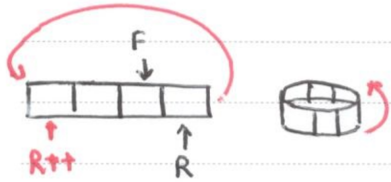
جهت افزودن عنصر به لیست Rear را یک عدد به جلو می‌بریم. در واقع انتهای صف با rear مشخص می‌گردد و ابتدای صف با Front. درصورتی که بخواهیم عنصر A را حذف کنیم Front را روی B قرار می‌دهیم.

صف دارای حالات خاص بسیاری است:

I) Rear و Front برابر 1- : هیچ عنصری در صف نیست.

II) Rear و Front برابر 0 : تنها یک عنصر در صف است.

III) overflow رخ دهد: درصورتی که شرایط مطابق روبرو باشد.



همچنین در حالت حذف نیز دارای چنین حالات خاصی هستیم.

جلسه هشتم

پیاده سازی با آرایه ها

Front : ابتدای صف (خروج از ابتدا)

Rear : انتهای صف (ورود به انتها)

```
class Queue {
    char [] Q;
    int Front, Rear;
    public Queue(int size) {
        Q = new char [size];   // تعریف آرایه به سایز دلخواه
        Front = Rear = -1;     // ابتدای کار هیچ عضوی در صف نداریم
    }
    public bool Additem( char item) {
        if ((Front == 0 && Rear == Q.length -1)) ||
           (Rear + 1 == Front))     وقتی صف پر باشد
           return false;            Front, Rear به هم رسیده باشند
        if (Rear == -1)              // اگر عضوی نداشتیم
           Front = Rear = 0;
        else
           Rear = (Rear + 1) % Q.Lenth;
        Q [Rear] = item;
        return True;
    }
    public bool Delitem ( ref char item) {
        if (Front == -1)    // اگر هیچ عضوی نداشتیم
           return false;
        item = Q[Front];
        if (Front == Rear)    // اگر یک عضو در صف داشتیم
           Front = Rear = -1;
        else
           Front = (Front + 1) % Q.length;
        return True;
    }
}
```

(حالت غیر معتبر — margin note beside the first condition in Additem and beside the check in Delitem)

پیاده سازی با لیست پیوندی

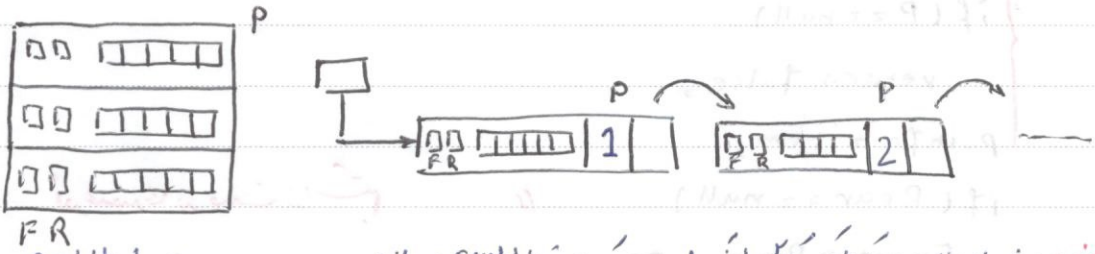
```
class Queue {
    class Node {
        public char info;
        public Node next;
    }
    Node Front = null, Rear = null;
    public bool AddItem (char item) {
        Node P = new Node();
        if (P == null)
            return false;
        P.info = item;
        if (Rear == null)          // اگر عنصری در صف نداشتیم
            Front = Rear = P;
        else {
            Rear.next = P;
            Rear = P;
        }
        return true;
    }
    public bool DelItem (ref char item) {
        if (Front == null)         // اگر عنصری در صف نداشتیم
            return false;
        if (Front == Rear)         // اگر یک عنصر در صف داشتیم
            Front = Rear = null;
        else
            Front = Front.next;
        return true;
    }
}
```

صف های مختلف

علاوه بر این صف های ساده که دیدیم ما صف های دیگری نیز می توانیم داشته باشیم. یک نمونه صف های ممدود هستند. عناصر بعد از دریافت سرویس کجا به انتهای صف می روند تا کجا سرویس بگیرند. نوع دیگر صف دوسره یا دوطرفه هستند. که می توانند دو ورودی و دو خروجی داشته باشند.

صف ممدود صف دوسره

نوع دیگر صف اولویت است. که عملا چند صف موازی هم داریم.



صف با اولویت

سوال امتحانی: برنامه ای بنویسید که با یک کلاک لیست یک صف با اولویت بسازید.

صف با اولویت با آرایه

```
class PQ {
    Queue [ ] QA;
    Public PQ (int size) {
        QA = new Queue[size];
        for (int i=0; i<size; i++)
            QA [i] = new Queue();
    }
    public bool AddItem (int P, char item) {
        if (P>=0 && P<= QA.length-1)
            return QA[P].AddItem(item);
        return false;
    }
    public bool DelItem (int P, ref char item) {
```

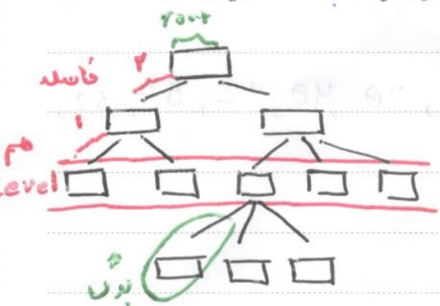
جلسه هشتم و نهم

```
if (P >= 0 && P <= QA.length-1)
    return QA[P].DelItem(ref item);
return false;
}
}
```

تمرین: یک کلاس صف لینک لیست با طول صف دلخواه بنویسید.

درخت ها:

درخت از ساختارهای چندضلعی است که قبل از هر عنصر بیش از دو عنصر قرار گرفته باشد. در درخت ها حالت سلسله مراتبی داریم و خبری از حلقه و loop نیست.



root: عنصری که پدر ندارد و پدر تمام شاخه ها را root گوییم.

برگ: عنصری که فرزندی ندارد و انتها/شاخه است را برگ گوییم.

فاصله: تعداد مراتب هر عنصر تا ریشه را فاصله گوییم.

هم level: عناصری که فاصله آنها تا ریشه برابر باشد را هم خانواده یا هم level گوییم.

ارتفاع یا عمق: بیشترین فاصله تا ریشه را ارتفاع یا عمق درخت گوییم (دورترین عنصر تا ریشه)

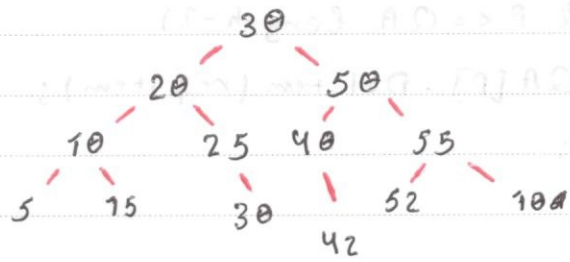
درخت جستجوی دودویی: BST

درخت ها در نرم افزارهای جستجو، فشرده سازی کاربرد دارند. یکی از معروف ترین ها Binary Search Tree
این درخت جهت جستجو به صورت پیشنهاد/عناصر دودویی استفاده می گردد. زمانی که تعداد عناصر نامعلوم است و نمی خواهیم ابتدا sort و سپس جستجو را انجام دهیم از جستجو دودویی استفاده می کنیم که خیلی راحت می توان عنصر اضافه کرد و جستجو را انجام داد. Binary درختی است که هر عنصر آن حداکثر دو فرزند خواهند داشت.

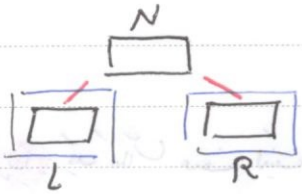
30, 20, 50, 10, 25, 40, 55, 5, 100   30, 15, 42, 52

با اولین عدد مقایسه کرده و اگر کوچکتر بود سمت چپ و اگر بزرگتر بود سمت راست

جلسه نهم



پیمایش درخت ها

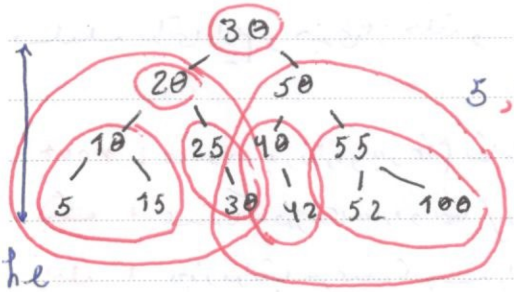


LNR → infix روش

NLR → prefix روش

LRN → postfix روش

روش LNR عناصر درخت را مرتب شده در اختیار قرار می دهد و پرکاربردترین است.

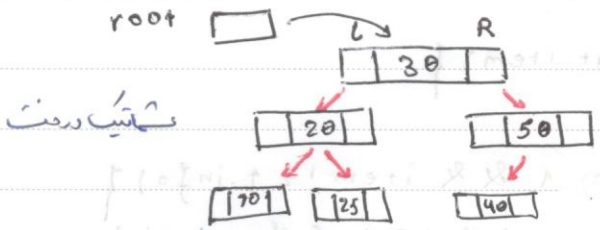


5, 10, 15, 20, 25, 30, 30, 40, 42, 50, 52, 55, 100

این روش سرعت را با حذف تعدادی از درخت هم در جستجو و هم در افزودن عناصر افزایش می دهد. درخت جستجو باینری محدودیتی در افزودن عنصر همانند آرایه ندارد و همزمان در حال sort کردن عناصر است بنابراین جستجو و افزودن عنصر در آن بسیار ساده و سریع است.

این نوع درخت محدودیتی دارد که بایستی دیتا ان معبوت random و پراکنده باشد، چنانچه دیتا ها معبوت sort شده باشند درخت به شکل یک خط خواهد بود. از طریق فرمول $|h_L - h_R| \le 1$ می توانیم آن را به بالانس بودن درخت ببریم. h_e ارتفاع آخرین node نسبت به root است.

چنانچه بخواهیم این درخت را بسازیم از یک لینک لیستی استفاده می کنیم.



شماتیک درخت

اعمال انجام شده بر روی درخت ها

- Create BST
- Add item
- Del item
- Find
- Print LNR

```
class BST {
    Class Node {
        public Node left;
        public int info;
        public Node right;
    }
    Node root = null;
```

تعریف متغیر اشاره‌گر به ریشه با نام root

```
    public bool AddItem (int item) {
        Node p = New Node ();
(1)     if (p == null) { return false; }
        p.info = item;
        if (root == null) {
            root = p;
            return true;
        }
```

اگر درخت خالی بود

چنانچه هیچ عنصری در درخت نداشتیم p را به root می‌دهیم و کار تمام خواهد بود در غیر اینصورت نیاز به travarce داریم

```
        Node S = null, t = root;
        while (t != null) {
(2)         S = t;
            t = (item < t.info) ? t.left : t.right;
        }
        if (item < S.info) { S.left = p; }
        else { S.right = p; }
        return true;
    }
```

آدرس ریشه درخت داریم

پیمایش

عملیات travarce

tips and trick
Art of programming
تکنیک‌های برنامه‌نویسی جالب

```
public bool Find(int item){
    Node t = root;
    While (t != null && item != t.info)
        t = (item < t.info) ? t.left : t.right;
    return (t != null);
}
```

مقدار true یا false را به این صورت نوشت !! که (t != null) کار if را می‌کند.

```
public void LNR(){
    if (root != null)
        LNR(root);
}
private void LNR(Node p){
        if (p.left != null)          L
            LNR(p.left);
        console.writeline(p.info);   N
        if (p.Right != null)         R
            LNR(p.Right);
}
```

NLR — LRN

اگر فرزند سمت چپ وجود داشت LNR مربوط به آن بایستی انجام گردد.

اگر فرزند سمت راست وجود داشت LNR مربوط به آن بایستی انجام شود.

جهت Delete کردن عناصر سه حالت ممکن داریم:

۱) عنصر مورد نظر برگ باشد که ساده‌ترین حالت است و به راحتی می‌توانیم اشاره‌گر والد را null بگذاریم.

۲) عنصر مورد نظر یک فرزند داشته باشد که فرزند را بایستی به جای آن جایگذاری کنیم.

۳) عنصر مورد نظر دو فرزند داشته باشد آنگاه چپ‌ترین عنصر سمت راست (کوچکترین عنصر سمت راست) یا راست‌ترین عنصر سمت چپ (بزرگترین عنصر سمت چپ) را می‌توانیم به جای آن جایگذاری کنیم.

```
public bool Delete item (int item) {
    Node t = root, S = null;
    while ( t != null && t.info != item) { S = t;
travarce        t = (item < t.info) ? t.left : t.right; }
    if(t == null)
      return false;
    if (t.left != null && t.right != null) {          دو فرزند //
      Node suc = t.right, P_suc = null;
      while (suc.left != null) {
          P_suc = suc;
          suc = suc.left;
      }
      if (P_suc == null)
          t.right = suc.right;
      else
          P_suc.left = suc.right;
      t.info = suc.info;
    }
    else {                                            یک یا بدون فرزند //
      Node child = t.left;
      if (t.right != null)
          child = t.right;
      if (S != null) {
          if (t == S.left)
              S.left = child;
          else
              S.right = child;
      }
      else
          root = child;
      return true;
    }
```

مشخص کردن فرزند کدر صورت عدم وجود null جایگزین میشود

والد در ریشه نیست

جلسه پنجم

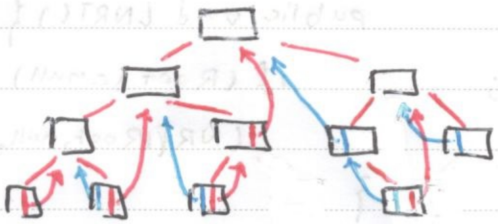
تمرین ۱: تابعی بنویسید که به کلاس BST اضافه شود و با گرفتن ریشه بصورت recursive عمق درخت را بازگرداند (دورترین عضو نسبت به ۰)

تمرین ۲: بررسی کنید آیا درخت بالانس هست یا خیر. بایستی عمق سمت چپ و راست هر گره تفاضل آنها کمتر یا مساوی ۱ باشد.

تمرین ۳: تابعی بنویسید که برای هر گره یک درخت را با گرفتن ریشه بازگرداند. Node ها با یک فرزند را بازگرداند. با گرفتن ریشه تعداد کل Node ها را بازگرداند.

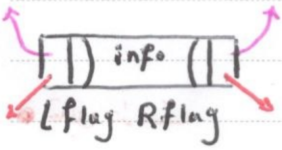
تمرین ۴: Additem بصورت recursive بنویسید.

ریسمان کشیدن یک طرفه LNR



Node های خالی را به عنصری که به روش LNR بعدی می‌آید می‌برد اشاره می‌دهیم در این روش Node های خالی تکمیل خواهند بود

اگر بخواهیم درخت را thread کنیم یا به ریسمان ببندیم هر Node دو تا thread دو طرفه می‌تواند به عنصر بعدی



اشاره کند یا به فرزند بدین منظور می‌توان از یک flag بولین استفاده کرد.

در LNR همواره به سمت چپ برویم عنصر بعدی پدر است اگر به سمت راست برویم عنصر بعدی برابر با عنصر بعدی پدر است.

برای عنصر قبلی درست برعکس همواره به سمت چپ برویم عنصر قبلی در واقع عنصر قبلی پدر و به سمت راست برویم عنصر قبلی پدر است.

ریسمان کشیده دو طرفه با پیمایش LNR

• اگر بخواهیم یک طرفه بخواهیم قسمت های آبی رنگ حذف می‌شوند.

```
private
Void LNRT(Node P, Node prev, Node next){
    if(P.left != null){
        p.leftchildflag = true;
        LNRT(P.left, prev, P);
    }
    else{
        p.leftchildflag = false;
        p.left = prev;
    }
    if(P.Right != null){
        p.Rightchildflag = true;
        LNRT(P.Right, P, next);
    }
```

```
else {
    p.RightchildFlag = false;
    p.Right = next;
}
}
```

```
public void LNRT() {
    if (Root != null)
        LNRT(Root, null, null)
}
```

این راه حل صادق است که با پشته تعریف recursive انجام شود بنابراین ،
راه حل بهتر از همان ابتدا در ساخت درخت عناصر را thread شده وارد کنیم.

```
if (item <= S.info) {
    S.left = P;              → P.left = S.left;
    S.leftchildflag = true;
    p.Right = S;
    p.RightChildFlag = false;   ✗
    p.leftchildflag = false;    ✗
}
else {
    p.Right = S.Right;
    S.Right = P;
    S.RightChildflag = true;
    p.left = S;
    p.RightchildFlag = false;   ✗
    p.leftchildflag = false;    ✗
}
```

● قابل حذف

با اینکار تغییراتی در traverse خواهیم داشت که هیچ‌گاه به null نمی‌رسیم بنابراین flag فرزندان را چک می‌کنیم:

```
Node t = Root , s = null;
while (t != null) {
    S = t;
    if (item <= t.info)
        t = (t.leftchildflag == true) ? t.left : null;
```

```
else
        t = (t.RightChildflag == true)? t.Right : null;
}
```

جهت انجام بصورت recursive بایستی ببینیم کدام تکه در برنامه بصورت loop انجام شود یعنی در یک حلقه باشد.

سوال امتحانی:

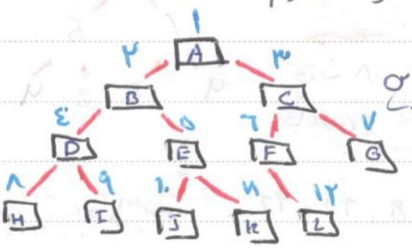
LNR: D, B, F, E, A, G, C, H

NLR: A, B, D, E, F, C, G, H

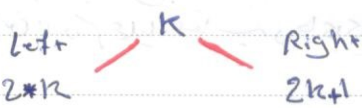
(Root)

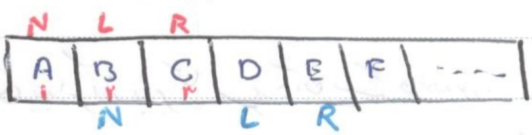
درخت دودویی کامل (Complete tree) **

این درخت همیشه به سمت چپ و سپس به سمت راست را پر میشود. اگر به هر جا بخواهیم فرزندی اضافه کنیم ابتدا به فرزند سمت چپ فرزند میدهیم تا کامل گردد. نظم سلسله مراتبی داخل درخت وجود دارد و شماره میزنیم.



میتوانیم برای ذخیره، کار از آرایه استفاده کنیم، و رابطه جالبی بین فرزندان هست:



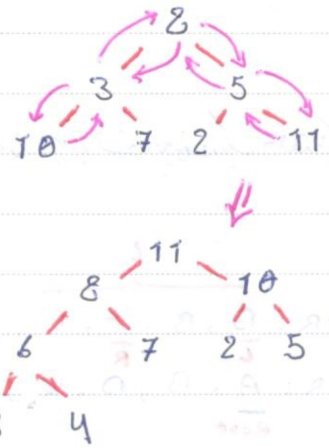


** Heap Sort

سریع ترین نوع sort است برای آشنایی با این الگوریتم باید با Heap tree آشنا شویم که دو نوع maxHeap tree و Min Heap Tree میباشد. Max Heap Tree درختی است که هر عنصر آن از سایر عناصر بزرگتر است و Min Heap Tree برعکس آن. بایستی به ترتیب دیتاها را دلیت نماییم.

هر عنصری که اضافه می‌شود ابتدا درخت باید شبیه درخت کامل باشد و بعد باید مقایسه شود تا جای آن مشخص گردد.

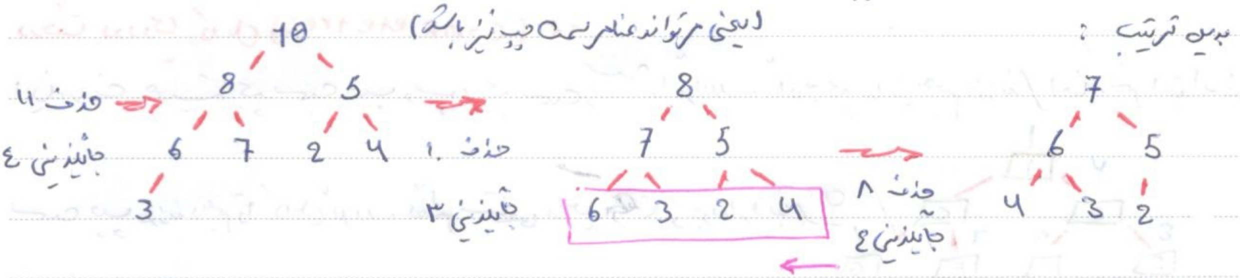
8, 3, 5, 10, 7, 2, 11, 6, 4



در نهایت چنین چیزی خواهیم داشت: Max Heap Tree

حالا عناصر را از آخر به اول حذف می‌کنیم و حالا در اول آخرین عنصر را اضافه می‌کنیم و بعد جای آن را پیدا می‌کنیم. بدین ترتیب:

(یعنی می‌تواند عناصر سمت چپ نیز باشد)



حذف ۷ → جایگزینی ۲

بدین ترتیب خواهیم داشت: 2, 3, 4, 5, 6, 7, 8, 10, 11

در این الگوریتم عملیات مقایسه و جابجایی کاهش پیدا کرده بنابراین سرعت آن بیشتر خواهد بود.

(برنامه در کانال)

8	3	5	10	7	2	11	6	4

Left = 2; $p = k/2 = 4/2 = 2$

Right = 3; $k = p = 2$

$p = k/2 = 2/2 = 1$; $p = k/2 = 1$

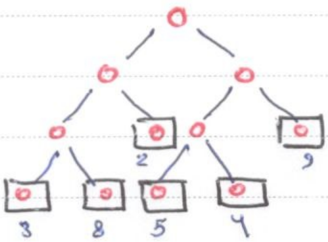
$p = 3/2 = 1$;

$k = 4$

```
Public void AddItem (int [] A, int item, int n) {
    n++;
    int Temp = A[n];
    int k = n, p = n/2;
    while ( p >= 1 ) {
        if ( A[P] < Temp)
            A[k] = A[P];
        else {
            A[k] = Temp;
            return;
        }
        k=P;
        p = k/2 ;
    }
    A[P] = Temp;
}
```

درخت هافمن — وزن دار

درخت هافمن یک دو درختی است یعنی هیچ عنصری تا یک فرزند وجود ندارد. آنهایی که فرزند ندارند برگ نامیده می‌شوند که به آن‌ها node های خارجی می‌گویند و باقی را node داخلی می‌نامند.



$N_E = N_I + 1$

طول داخلی $l_I = 0 + 1 + 2 + 1 + 2 = 6$

$l_E = 3 + 3 + 2 + 3 + 3 + 2 = 16$

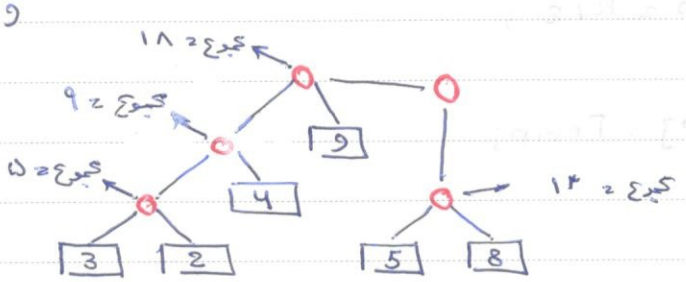
وزن × فاصله $P = (3 \times 3) + (8 \times 3) + (2 \times 2) + \dots$

وزن دلخواهی را برای هر برگ (node های خارجی) در نظر می‌گیریم و در فاصله ضرب می‌کنیم.

درخت هافمن دو درختی است که مجموع وزن دار node های خارجی آن کوچکترین عدد ممکن باشد.

ساخت درخت هافمن: اعضای کوچکتر را دو تا دو تا می‌گیریم.

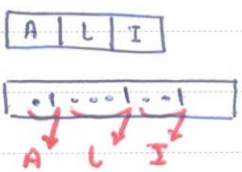
3, 8, 2, 5, 4, 9
8, 5, 5, 4, 9
۲،۳ جمع 8, 9, 9
13, 9, 9
13, 18



کاربرد این درخت در فشرده سازی‌ها و رمزنگاری است. در این درخت چپ و راست بودن اهمیت ندارد و تنها به دنبال فاصله و طول هستیم.

جهت رمزنگاری ابتدا بنا بر فراوانی حروف به آنها وزن اختصاص می‌دهیم به عنوان مثال و بعد درخت هافمن متناظر با آن را رسم و در این مرحله تمام سمت راستی‌ها را صفر و تمام سمت چپی‌ها را یک قرار دهیم آنگاه خواهیم داشت.

A	100	01
O	98	11
U	90	
e	95	10
I	92	001
z	18	
l		0001



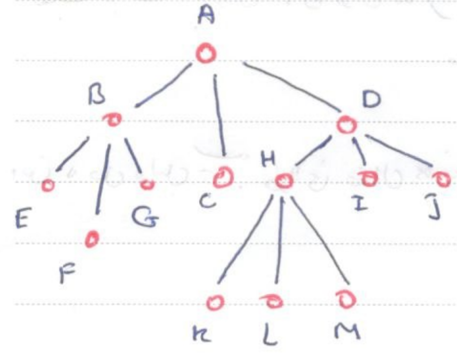
24 bit → 9 bit

از آنجا که درخت هافمن است و برگ‌ها فرزندی ندارند به راحتی می‌توانیم رمز را دیکود کنیم.

جلسه یازدهم

درخت هافمن را در اول با min Heap نوشت. (در امتحان نیست)

درخت عمومی »



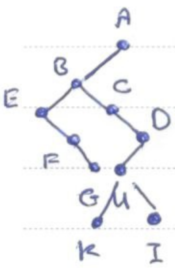
درختی است که محدودیت تعداد فرزند را ندارد. برای آن آرایه ای به تعداد node ها تعیین می کنیم که هر خانه آن سه بخش داشته باشد.

Node left sibl → برادر سمت راست

فرزند سمت چپ

C	-	D
I	-	J
A	B	-
B	E	C

چینش هر گونه اتفاقی باشد ←



چنانچه sibl را به right یعنی فرزند سمت راست تبدیل کنیم درخت دودویی خواهیم داشت.

گراف »

درختی که بین عناصر آن چندین رابطه برقرار و وجود داشته باشد. روابط یک طرفه یا دو طرفه. گراف های مورد نظر ما گراف های هم بند است و عنصر آزاد نداریم. گراف هر یال آن وزن دار باشد.

برای نمایش گراف در حافظه دو ساختار داریم آرایه ماتریسی و لینک لیست. اگر تعداد node ها ثابت باشد برنامه از آرایه استفاده کنیم. برای نمایش گراف به صورت ماتریسی باید ماتریسی با سطر و ستون به تعداد گره ها داشته باشیم.

(1) A ↓

این گراف طول مسیر را مشخص می کند.

ماتریس مجاورت:

	A	B	C	D
A	1	2	0	0
B	0	0	1	0
C	1	0	0	1
D	0	1	0	0

درایه های ماتریس تعداد مسیر بین دو عنصر را مشخص می کند.

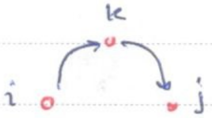
ماتریس مسیر ماتریسی باینری است که نشان می‌دهد بین دو عنصر مسیری وجود دارد یا خیر و مقادیر آن صفر و یک است.
جهت بدست آوردن آن توان های ماتریس مجاورت گراف را با هم جمع می‌کنیم و آنرا باینری می‌کنیم.

$A + A^2 + A^3 + \dots + A^{n} = B$ → Binary → ماتریس مسیر

(n → تعداد عناصر)

این راه حل بهینه نیست. فرمول‌های جدا بهینه نیست بنابراین از الگوریتم وارشال استفاده می‌کنیم.

$P[i,j] = 1$

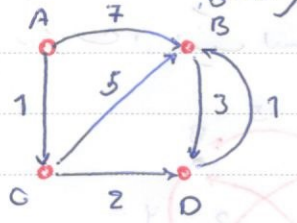


$P[i,j] = 0$

$P[i,j] = P[i,j] \vee (P[i,k] \wedge P[k,j])$

```
public bool[,] Varshal(int[,] A, int n) {
        bool[,] P = new bool[n,n];
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                P[i,j] = (A[i,j] > 0) ? true : false;
            }
        }
        for(int k =  ; k < n; k++) {
            for(int i = 0; i < n; i++) {
                for(int j = 0; j < n; j++) {
                    P[i,j] = P[i,j] || P[i,k] && P[k,j];
                }
            }
        }
        return P;
}
```

الگوریتم فلوید وارشال



$W =$

	A	B	C	D
A	0	7	1	0
B	0	0	0	3
C	0	5	0	2
D	0	1	0	0

$\xrightarrow{Q}$

$Q_0 =$

	A	B	C	D
A	∞	7	1	∞
B	∞	∞	∞	3
C	∞	5	∞	2
D	∞	1	∞	∞

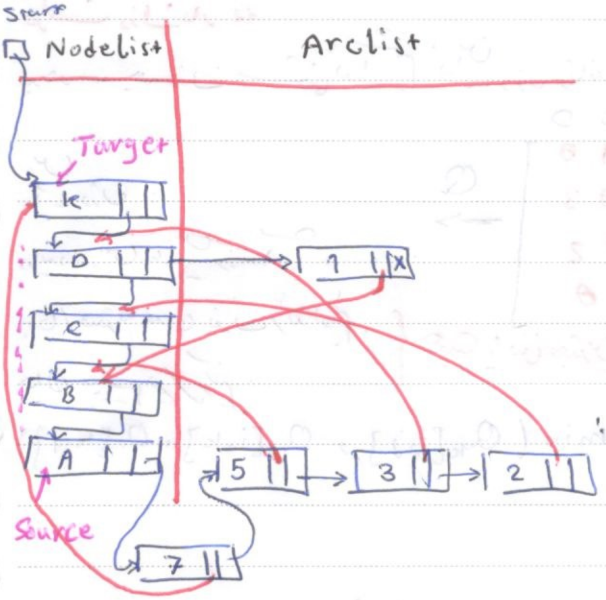
هدف یافتن بهترین مسیر

$$Q^{k+1}[i,j] = \min\left(Q_k[i,j],\ Q_k[i,k] + Q[k,j]\right)$$

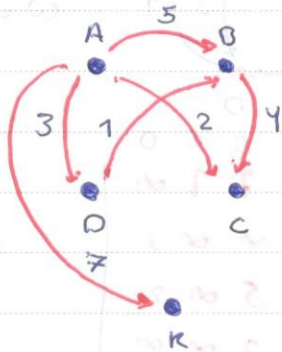
```
public int [,] BestPath (int[,] A, int n) {
    int [,] Q = new int [n,n];
    for(int i=0 ; i<n; i++){
        for(int j=0; j<n; j++){
            Q[i,j] = (A[i,j] > 0) ? Q[i,j] : 10000;
        }
    }
    for(int k=0; k<n ; k++){
        for(int i=0; i<n; i++){
            for(int j=0; j<n; j++){
                int temp = Q[i,k] + Q[k,j];
                Q[i,j] = (Q[i,j] <= temp) ? Q[i,j] :temp;
            }
        }
    }
    return 0;
}
```

گراف با Linkedlist

چنانچه عناصر در گراف محدود نباشد و بخواهیم به آن اضافه کنیم مجبوریم از لینک لیست ما کمک بگیریم.



جای چیدن یالها باشند:



Add Node ("P") باید ابتدا چک کنیم این Node جدید از قبل وجود نداشته باشد سپس Node را وارد کنیم.

SName ↑ TName ↑

Add Ark ("A", "k", 7) سپس Arc را باید اضافه کنیم که باید بگوییم از کجا به کجا و چه وزنی آن را باید قرار دهیم.

s t w
source target weight

★ k را می‌خواهیم اضافه کنیم.

```
Class Nodelist {
    public String Name;
    public Nodelist Next;
    public Arclist StartofArcs;
}
class Arclist {
    public int Arclist;
    public Nodelist target;
    public Arclist Next;
}
```

```
Nodelist Target = null , Source = null;
Nodelist t = Start;
while (t != null && (target == null || source == null)) {
      if (t.Name == TName)
            target = t;
      if (t.Name == SName)
            Source = t;
      t = t.next;
}
if (Source == null || target == null)
      return false;
```

پیدا کردن target و source

در صورتی که target و source را داشتیم ، از Arclist یک object درست و آن را به بعد از source اضافه می‌کنیم و به object بعدی مرتبط می‌کنیم (طبق شکل عدد ۷)

Delete Arc("A", "B", 5) :

برای delete می‌توانیم از همان کد بالا استفاده کنیم تا اگر خروجی false نبود آنگاه همانند لینک لیست یک طرفه که t و s داشت آن را حذف کنیم.

Delete Node("B") : می‌توانیم الگوریتم دیگری استفاده کنیم که کمی کمتر است. و آن حذف Node است.

بایستی تمام Arcهای ورودی و خروجی به Node را حذف کنیم.

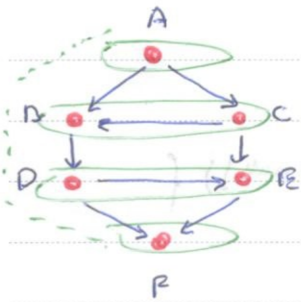
ابتدا بایستی بررسی کنیم Node B وجود دارد یا خیر. اگر وجود داشت آن را از لینک لیست خارج کنیم.

سپس تک تک عناصر Arclist را چک target را با B مقایسه کنیم ، در صورت یکسان بودن آن را حذف کنیم.

بایستی تمام Nodeها را چک کنیم و تنها به حذف اولین بسنده نکنیم.

به دلیل وجود Garbage collector در زبان‌های Java و C# نیازی نیست که خروجی‌ها را B را حذف کنیم. با حذف B آنها نیز اتوماتیک وار حذف می‌گردند.

(کد نوشته شده در کانال - سوال امتحانی ★)



پیمایش گراف

پیمایش بر دو نوع عرضی و عمقی است. پیمایش یعنی سراغ تک تک عناصر یک بار و فقط یک بار برویم. در این گراف مثلا Node B ارش هم از A در دسترس است و هم از C بنابراین برای یک بار پیمایش آن باید Status ای را برای Node ها در نظر بگیریم.

۱) پیمایش نشده

۲) آماده جهت پردازش

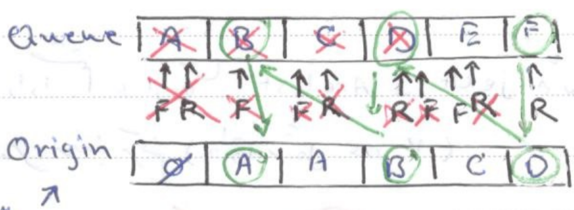
۳) پردازش شده

Node	Adjustment Nodes
A	B , C
B	D
C	B , E
D	E , F
E	F
F	—

بنابراین لیستی به نام لیست مجاورت در نظر میگیریم (Adj Node).

در پیمایش عرضی جهت یافتن کوتاه ترین مسیر استفاده می شود از صف (Queue) استفاده می شود.

A → F



به ما می گوید از کجا به Node رسیدیم.

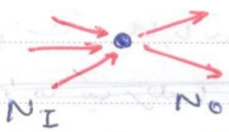
برای پیمایش عمقی داخل stack قرار میدهیم و از آن خارج می کنیم. اگر بخواهیم کوتاه ترین مسیر را بیابیم از پیمایش عرضی استفاده می شود. اگر بخواهیم بدانیم چه Node هایی از Node اول مثل A قابل دسترسی اند از پیمایش عمقی استفاده می شود.

A , C , E , F , B , D → از A این Node ها قابل دسترسی اند.

E , F

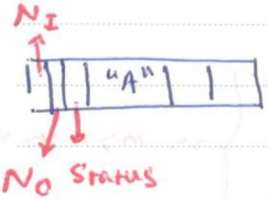
ممکن است از ما خواسته شود Node ای را پیدا کنیم که دارای بیشترین تعداد Node وصل شدگی باشد و می‌توان پیمایش عمقی را از آن آغاز کرد.

باید Node ای را پیدا کنیم که کمترین تعداد N_I (Node ورودی) را دارد.



ممکن است دو عنصر با N_I یکسان داشته باشیم که آنگاه عنصری را انتخاب می‌کنیم که N_O بیشتری داشته باشد.

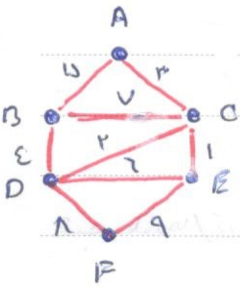
باید به هر Node گراف علاوه بر Node های قبلی سه فیلد جدید جهت پیمایش اضافه کنیم.

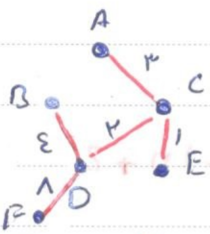


تمرین: پیمایش عرضی و عمقی از Node دلخواه انجام دهید.

N_O و N_I با اضافه شدن یا کم شدن از گراف تغییر می‌کنند.

درخت پوشای کمینه (کمینه)





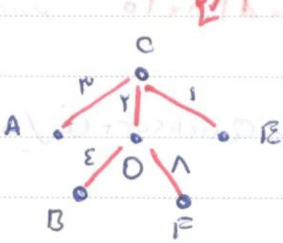
از یک گراف می‌خواهیم درختی بسازیم تا Arc های آن کمترین باشد.

یک راس دلخواه انتخاب می‌کنیم مثلا C. کمترین وزن از آن را خارج می‌کنیم

سپس وزن بعدی را خارج می‌کنیم به شرطی که حلقه تشکیل نشود.

این کار را به ترتیب ادامه می‌دهیم تا تمام رئوس حذف گردند.

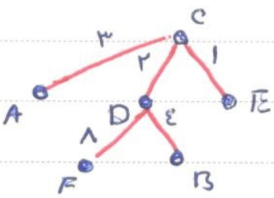
★ روش اول (پرایم)



(گراف همبند)

روش زیر بدون نیاز به رئوس است که به لبه‌ها کار دارد. لبه‌های کوچکتر انتخاب می‌شوند به شرطی که لبه‌ها شامل حلقه نشوند.

★ روش دوم (کراسکال)



برای یک گراف تنها یک درخت پوشای کمینه وجود دارد.

در نهایت درخت بایستی همبند باشد.

الگوریتم های خارج از امتحان))

الگوریتمی بهتر است که سریعتر عمل نماید. برای مقایسه سرعت الگوریتم ها روش های مختلفی داریم: یکی از روش ها روش تخمینی است که معیار ما تعداد دستورالعمل ها است. هر Assignmute یک مرحله if else ها هر کدام که بیشترین دستورالعمل دارد به تعداد آن. switch ها برای هر case که بیشترین تعداد دستورالعمل دارد. برای for ها و حلقه ها تعداد اجرای آن ضربدر دستورات

```
if ( ) {
  ——        m
}
else {                  → m > n ⇒ m
  ——        n
}

switch ( — ) {
  case 1:
    ——      m
  case 2:                 ⇒ m > n ⇒ m
    ——      n
}

for (int i = 0 ; i < 10 ; i++) {      (n+1 , n)
  ——  → m                             (m+2)n+2
}
```

همه حلقه ها تو در تو $mn^2 + (m+2)n$

از آنجا که الگوریتم های شکل Quicksort سریعتر هستند چرا که معمولاً log عمل مرتب‌سازی می‌کنند $\log n < n$
$\Rightarrow n \log n < n^2$