ساختمان های داده

دکتر اسدی

تحلیل
- زمانی → تابع پیچیدگی زمانی F(n)
- حافظه ای

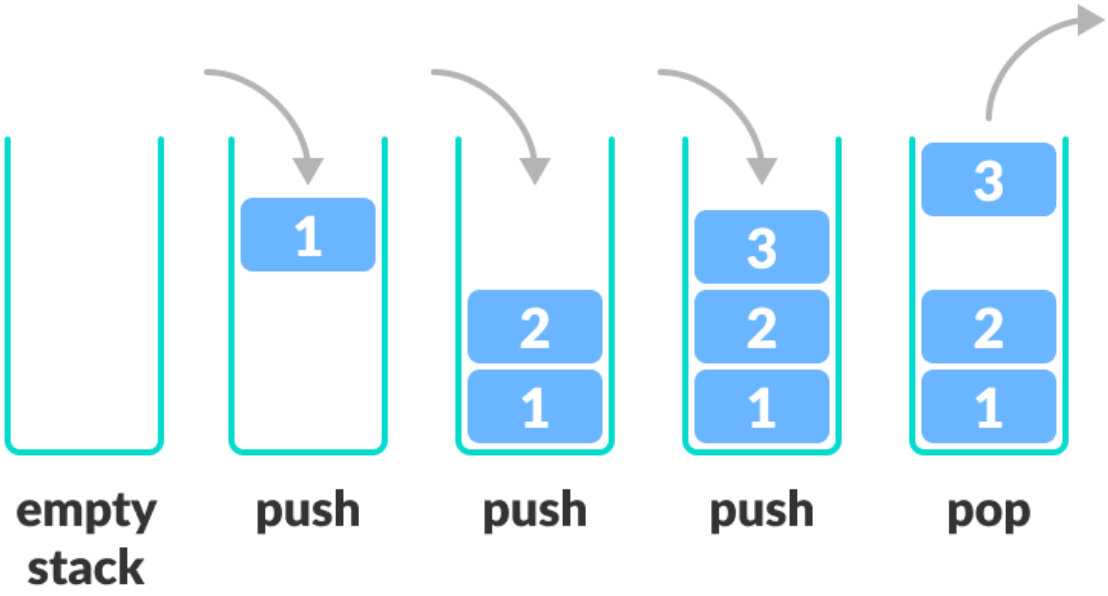
```
for i in range(n):
    for j in range(n):
        print(i*j)
        print(i)
    print(j)
print(i+j)
```

$F(n) = 2n^2 + n = 1$

O (حد بالای پیچیدگی زمانی) : بزرگتر یا مساوی بزرگترین توان F(n)

Ω (حد پایین پیچیدگی زمانی) : کوچکتر یا مساوی بزرگترین توان F(n)

θ (حد وسط پیچیدگی زمانی) : مساوی بزرگترین توان F(n)



پشته (Stack) :

ساختمان داده‌ای است که داده ها بر اساس منطق در (Last In First Out) LIFO در آن قرار میگیرند.

```
class Stack ():
    def __init__(self, limit = 100):
        self.stack = []
        self.limit = limit
```

تابع push (اضافه کردن) :

شرط : پر نبودن پشته

```
class Stack ():
    def __init__(self, limit = 100):
        self.stack = []
        self.limit = limit

    def push (self, data):
        if len(self.stack) >= self.limit: #شرط پر نبودن
            return -1
        else:
            self.stack.append(data)
```

تابع pop (حذف آخرین المان) :

شرط : خالی نبودن پشته

```
def pop (self):
    if len(self.stack) <= 0: #شرط خالی نبودن
        return -1
    else:
        return self.stack.pop()
```

تابع peek (برگرداندن آخرین المان) :

شرط : خالی نبودن پشته

```
def peek (self):
    if len(self.stack) <= 0: #شرط خالی نبودن
        return -1
    else:
        return self.stack[-1]
```

تابع find (پیدا کردن یک المان) :

شرط : خالی نبودن پشته

```
def find (self, x):
    if len(self.stack) <= 0:
        return -1
    else:
        for i in range(len(self.stack)):
            if self.stack[i] == x :
                return i
        return -1
```

```python
class Stack:
    def __init__(self, limit=100):
        self.stack = []
        self.limit = limit

    def peek(self):
        if len(self.stack)<=0:
            return -1
        else:
            return self.stack[len(self.stack)-1]

    def push(self, data):
        if len(self.stack) >= self.limit:
            return -1
        else:
            self.stack.append(data)

    def pop(self):
        if len(self.stack)>=0:
            return self.stack.pop()
        else :
            return -1

    def show(self):
        for i in range(len(self.stack)-1, -1, -1):
            print(self.stack[i], end=' ')

    def display(self):
        for i in range(len(self.stack)):
            print(self.stack[i])

    def replace(self, old_value, new_value):
        if old_value in self.stack:
            index = self.stack.index(old_value)
            self.stack[index] = new_value
            return self.stack
        else:
            return -1
```

نحوه نوشتن عبارات ریاضی

: Infix A+B

: postfix AB+

: Prefix +AB

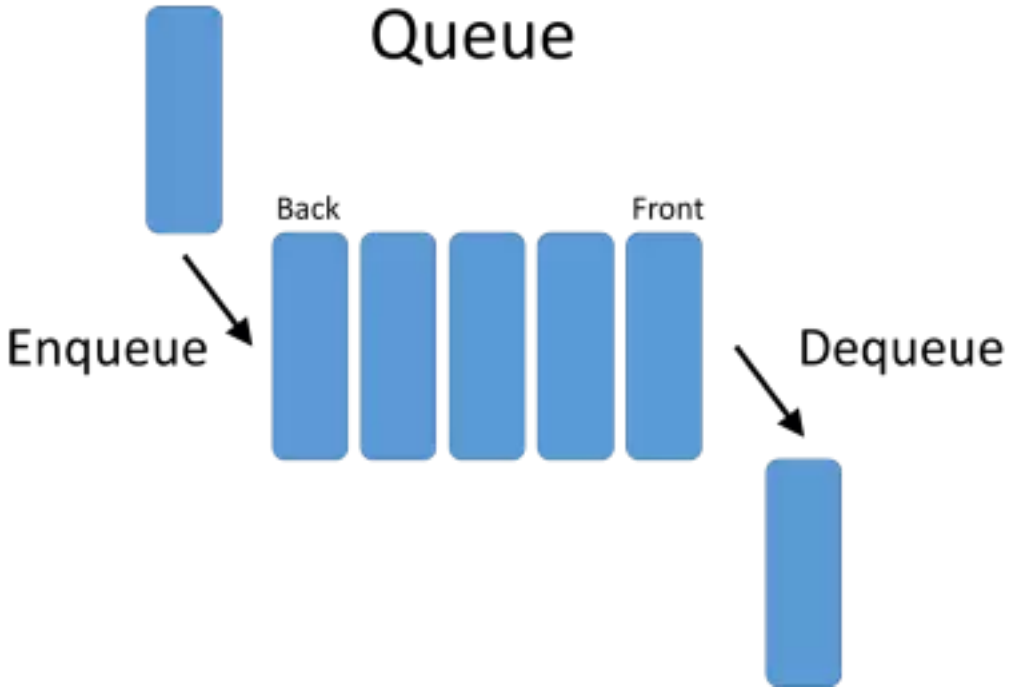
جابجا کردن ← پرانتز گذاری ← تبدیل infix به postfix , prefix

مثال :

: Infix (A+(B*C))

: postfix ABC*+

: Prefix +A*BC



صف (Queue) :

ساختمان داده‌ای است که داده ها بر اساس منطق در (First In First Out) FIFO در آن قرار میگیرند.

```
class Queue :
    def __init__(self, limit):
        self.Q = [None] * limit
        self.limit = limit
        self.front = -1 #ابتدای صف
        self.rear = -1 #انتهای صف
```

تابع insert (اضافه کردن به صف) :

شرط : پر نبودن صف

```
class Queue :
    def __init__(self, limit):
        self.Q = [None] * limit
        self.limit = limit
        self.front = -1 #ابتدای صف
        self.rear = -1 #انتهای صف

    def insQueue (self, data):
        if self.rear >= self.limit-1: #پر نبودن
            return -1
        else:
            self.Q[self.rear+1] = data
            self.rear += 1
            if self.front == -1:
                self.front = 0
```

تابع delete (حذف اولین المان) :

شرط : خالی نبودن صف _ آکبند نبودن صف

```
def delQueue (self):
    if self.front == -1: #آکبند نبودن صف
        return -1
    elif self.front > self.rear: #خالی نبودن صف
        return -1
    else:
        self.front += 1
        return self.Q[self.front-1]
```

تابع is_full (چک کردن پر بودن صف) :

```
def is_full (self):
    return self.rear >= self.limit-1
```

تابع is_empty (چک کردن خالی بودن صف) :

روش اول :

```
def is_empty (self):
    return self.front == -1
```

روش دوم :

```
def is_empty (self):
    return self.front > self.rear
```

تابع show (نشان دادن المان های صف) :

شرط : آکبند نبودن صف

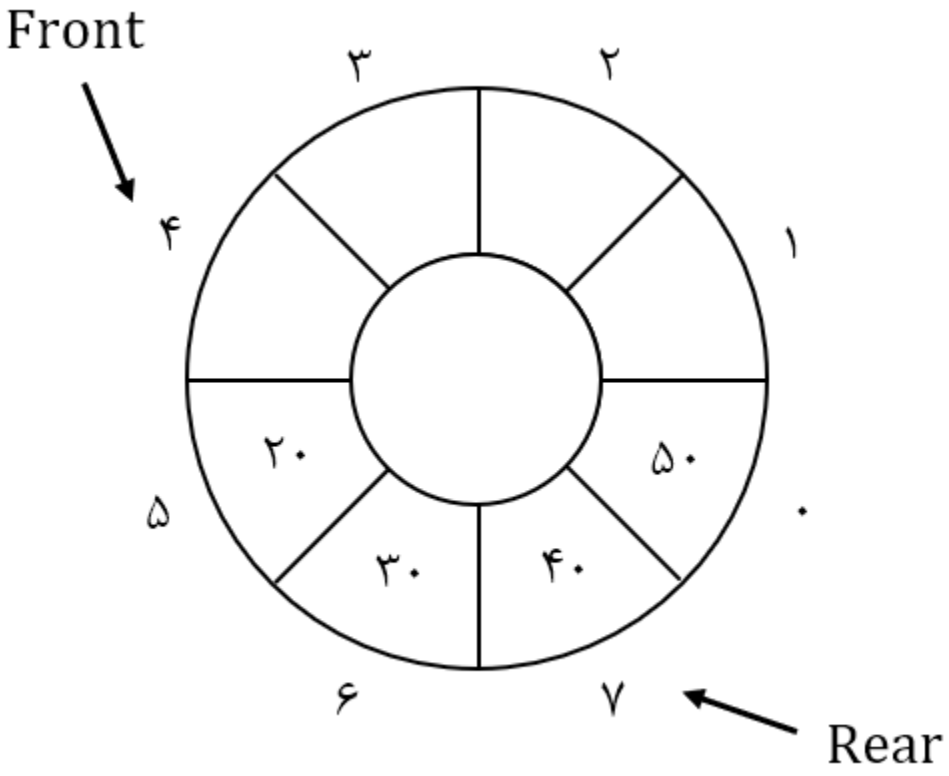
```
def showQueue (self):
    if self.front == -1: #آکبند نبودن صف
        return -1
    else:
        for i in range(self.front, self.rear+1):
            print(self.Q[i])
```

تابع revstack (برعکس کردن پشته با استفاده از پشته) :

مثال : تابعی بنویسید که بوسیله ساختمان داده پشته داده های درون پشته ورودی را معکوس کند.

```
def revstack(L):
    S1 = Stack(10)
    S2 = Stack(10)
    while len(L.Stack) > 0:
        S1.push(L.pop())
    while len(S1.Stack) > 0:
        S2.push(S1.pop())
    while len(S2.Stack) > 0:
        L.push(S2.pop())
```

```python
class Queue:
    def __init__(self,limit):
        self.Q=[None]*limit
        self.limit=limit
        self.front=-1
        self.rear=-1

    def insQueue(self,data):
        if self.rear>=self.limit-1:
            return -1
        elif self.front==-1:
            self.front=0
            self.rear=0
            self.Q[0]=data
        else:
            self.rear+=1
            self.Q[self.rear]=data

    def delQueue(self):
        if self.front==-1:
            return -1
        elif self.rear<self.front:
            return -1
        else:
            self.front+=1

    def show(self):
        for i in range(self.front,self.rear+1):
            print(self.Q[i])
```



صف حلقوی :

ساختمان داده‌ای است که داده ها بر اساس منطق در (First In First Out) FIFO در آن قرار میگیرند.

1. خالی بودن صف حلقوی — front = rear = -1
2. پر بودن صف حلقوی — front = (rear + 1) % limit
3. تک عنصری بودن صف حلقوی — front = rear

ساختار :

```
class CQueue :
    def __init__(self, limit = 100):
        self.Q = [None] * limit
        self.limit = limit
        self.front = -1 #ابتدای صف
        self.rear = -1 #انتهای صف
```

تابع insert :

حالات اضافه کردن به صف حلقوی :

1. وقتی صف پر است : اضافه نشود.
2. وقتی صف خالی است : هم به ابتدا و هم به انتهای صف اضافه شود.
3. سایر موارد : فقط به انتهای صف اضافه گردد.

```
def insQueue (self, data):
    if self.front == (self.rear+1) % self.limit: #پر نبودن
        print("full")
        return -1
    elif self.front == -1: #خالی بودن صف
        self.front += 1
        self.rear += 1
        self.Q[0] = data
    else:
        self.rear += 1
        self.Q[self.rear] = data
```

تابع delete :

حالات حذف کردن از صف حلقوی :

1. خالی بودن : حذف نکند.
2. تک عنصری بودن : front = rear = -1
3. سایر موارد : front += 1

```
def delQueue (self):
    if self.front == -1: #خالی بودن
        print("empty")
        return -1
    elif self.front == self.rear: #تک عنصری
        x = self.Q[self.rear]
        self.front = -1
        self.rear = -1
        return x
    else:
        x = self.Q[self.front]
        self.front = (self.front + 1) % self.limit
        return x
```

تابع display (نمایش المان ها) :

حالات نمایش یک صف حلقوی :

1. خالی بودن : نمایش "empty"
2. اگر front <= rear باشد : از حلقه for بصورت (front, rear + 1) استفاده میکنیم.
3. اگر front > rear باشد : از دو حلقه for بصورت (front, limit) و (0, rear + 1) استفاده میکنیم.

نمونه زیر را ببینید :

```
def display (self):
    if self.front == -1: #خالی بودن
        print("empty")
        return -1
    elif self.front <= self.rear:
        for i in range(self.front, self.rear + 1):
            print(self.Q[i], end="")
    else:
        for i in range(self.front, self.limit):
            print(self.Q[i], end="")
        for i in range(0, self.rear + 1):
            print(self.Q[i], end="")
```

```python
class CQueue:
    def __init__(self,limit):
        self.Q=[None]*limit
        self.limit=limit
        self.front=-1
        self.rear=-1

    def insCQueue(self,x):
        if (self.rear+1)%self.limit==self.front:
            print('Full')
            return -1
        elif self.front==-1:
            self.front+=1
            self.rear+=1
            self.Q[0]=x
        else:
            self.rear+=1
            self.Q[self.rear]=x

    def delCQueue(self):
        if self.front==-1:
            print('empty')
            return -1
        elif self.rear==self.front:
            temp = self.Q[self.front]
            self.front=-1
            self.rear=-1
            return temp
        else:
            temp = self.Q[self.front]
            self.front+=1
        return temp

    def displayCQueue(self):
        if self.front==-1:
            print('empty')
            return
        elif self.front<=self.rear:
            for i in range(self.front,self.rear+1):
                print(self.Q[i],end=' ')
        else:
            for i in range(self.front,self.limit):
                print(self.Q[i])
            for i in range(0,self.rear+1):
                print(self.Q[i])
```

```python
def factorial(n):
    if n == 1:
        return 1
    return factorial(n-1) * n

print(factorial(6))
```

توابع بازگشتی :

- شرط بازگشت
- مقدار بازگشت
- رابطه بازگشتی

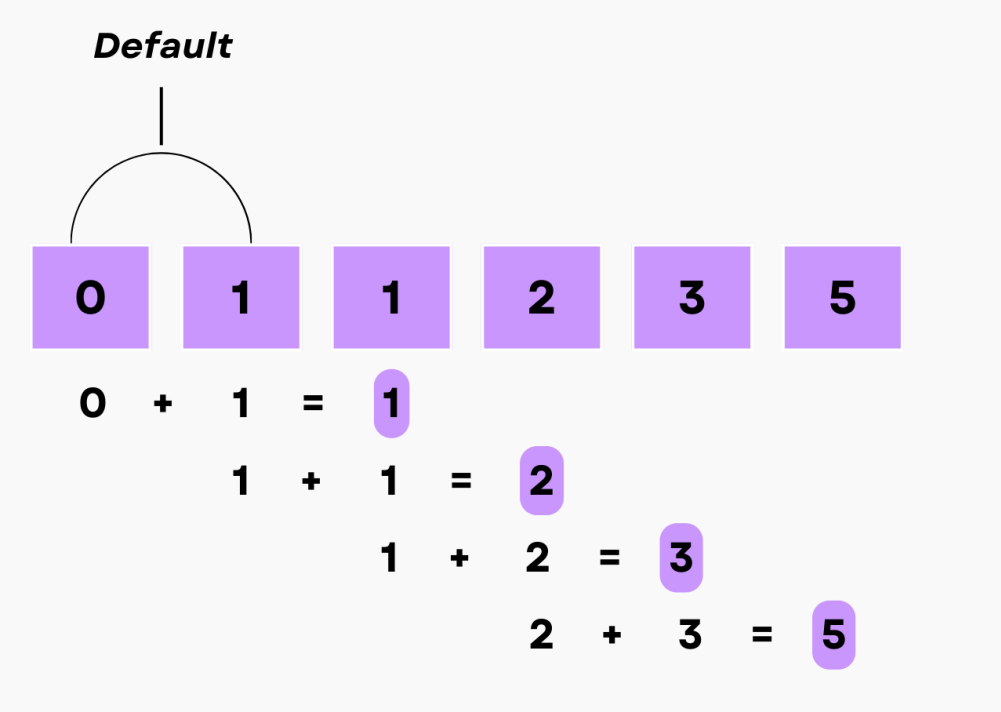
```
def rec (...):
    if شرط بازگشت :
        return مقدار بازگشت
    else :
        return رابطه بازگشتی
```

تابع بازگشتی محاسبه فاکتوریل (n!) :

- شرط بازگشت : n == 1
- مقدار بازگشت : 1
- رابطه بازگشتی : n! = n * (n – 1)

```
def fact (n):
    if n == 1 :
        return 1
    else:
        return n * fact( n-1 )
```

تابع بازگشتی محاسبه جمله n ام سری فیبوناچی :



- شرط بازگشت : $n < 2$
- مقدار بازگشت : n
- رابطه بازگشتی : $f(n) = f(n-1) + f(n-2)$

```python
def fib (n):
    if n < 2 :
        return n
    else:
        return fib( n-1 ) + fib( n-2 )
```

تابع بازگشتی حاصل جمع a و b : ($b > 0$ و a)

- شرط بازگشت : $b == 0$
- مقدار بازگشت : a
- رابطه بازگشتی : $S(a, b) = S(a, b-1) + 1$

```python
def sum (a , b):
    if b == 0 :
        return a
    else:
        return sum( a , b-1 ) + 1
```

تابع بازگشتی حاصل ضرب a و b : (a و b > 0)

- شرط بازگشت : b == 0
- مقدار بازگشت : 0
- رابطه بازگشتی : M (a , b) = M (a , b -1) + a

```
def mul (a, b):
    if b == 0 :
        return 0
    else:
        return a + mul( a , b-1 )
```

تابع بازگشتی حاصل تقسیم a و b : (a و b > 0)

- شرط بازگشت : a < b
- مقدار بازگشت : 0
- رابطه بازگشتی : D (a , b) = D ((a – b) , b) +1

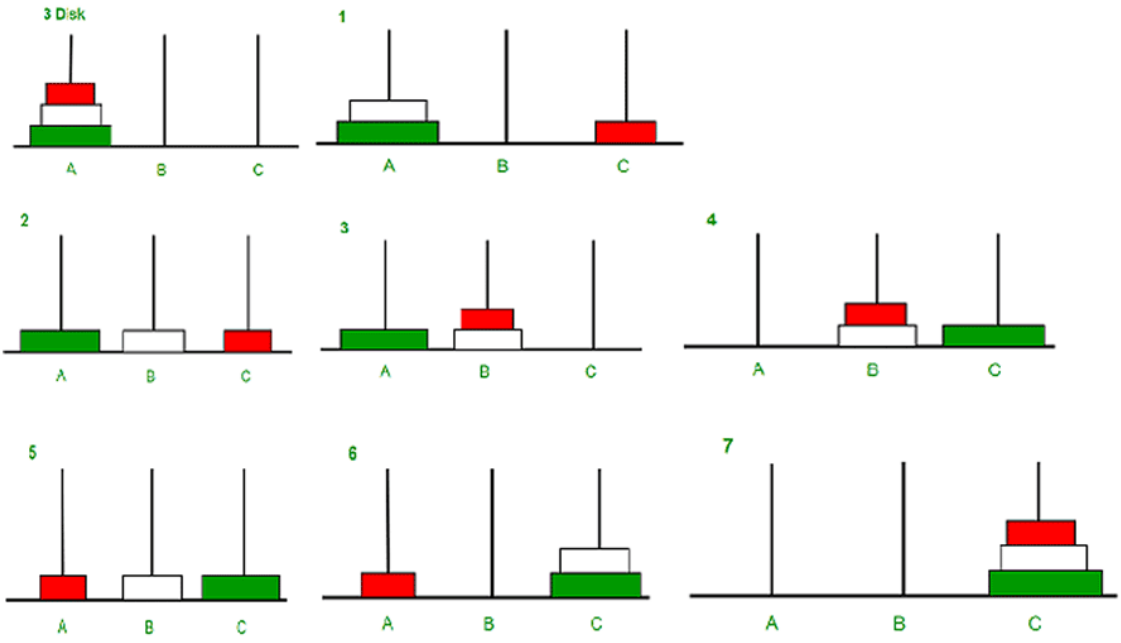
```
def div (a , b):
    if a < b :
        return 0
    else:
        return 1 + div( a-b , b )
```

تابع بازگشتی رشته تشخیص رشته یا پالیندروم : (رشته ای که از ابتدا و انتها به یک شکل خوانده میشود)

```
def pal (S):
    if len(S) <= 1 :
        return True
    if S[0] == S[-1] :
        return pal( S[ 1 : -1 ])
    return False
```

برج هانوی :

- در هر مرحله فقط یک دیسک جابجا شود.
- هیچوقت دیسک بزرگ بر روی دیسک کوچکتر قرار نگیرد.



n = تعداد

A = میله اول (مبدا)

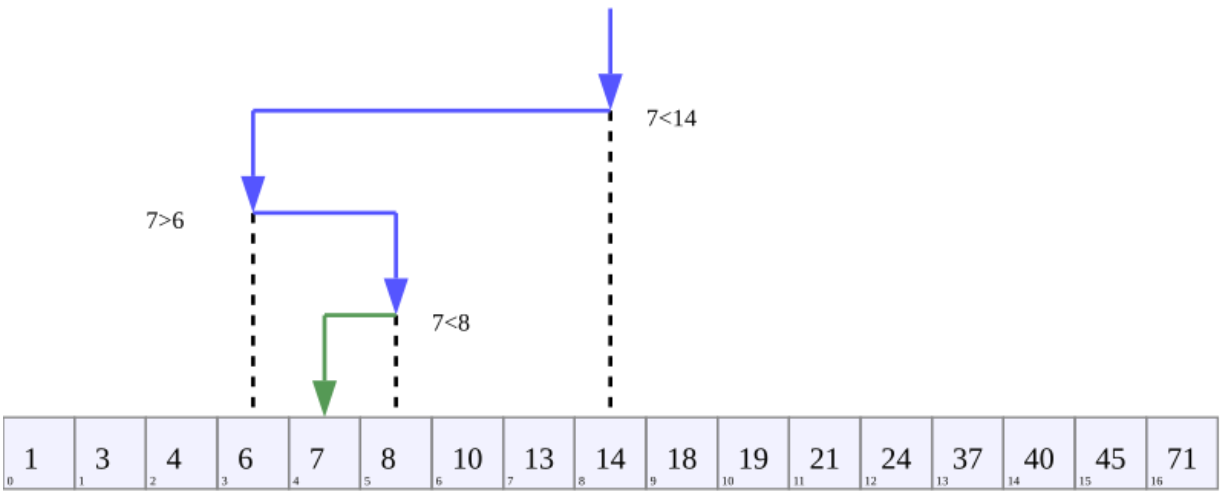
B = میله دوم (کمکی)

C = میله سوم (مقصد)

```
def Hanoi (n, A, B, C):
    if n == 1 :
        print(A, "to", C)
    else:
        Hanoi( n - 1, A, C, B )
        print( A, "to", C)
        Hanoi( n - 1, B, A, C )
```

تابع بازگشتی دودویی :

1. آرایه از کوچک به بزرگ مرتب است.
2. هر بار آرایه نصف میشود.



- شرط بازگشت : if first > last
- مقدار بازگشت : False

- رابطه بازگشتی :

x > Arr [mid]

last = last

first = mid + 1

x < Arr [mid]

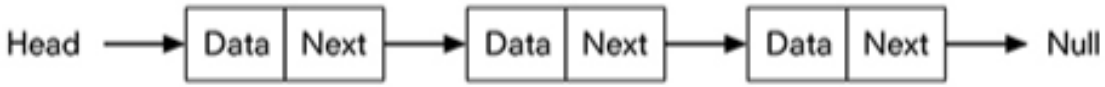
first = first

last = mid – 1

x == Arr [mid]

return mid

```
def Bsearch (arr, first, last, x):
    mid = (first + last) // 2
    if first > last :
        return False
    elif x > arr[mid] :
        return Bsearch(arr, mid + 1, last, x)
    elif x < arr[mid] :
        return Bsearch(arr, first, mid - 1, x)
    else:
        return mid
```



لیست پیوندی یک طرفه :

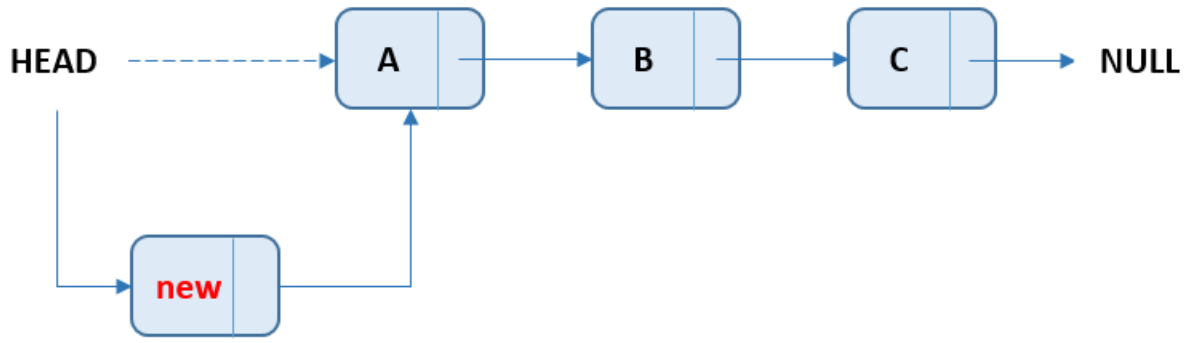
- مجموعه نود های متصل به یکدیگر با آدرس ابتدای head

ابتدا کلاس نود را میسازیم:

```
class Node :
    def __init__(self , x):
        self.data = x
        self.next = None
```

سپس به سراغ کلاس لیست پیوندی میرویم و تابع **add first** را مینویسیم: (اضافه کردن به ابتدای لیست)

```
class LinkedList():

    def __init__(self):
        self.head = None

    def add_first(self , d):
        a = Node(d)
        a.next = self.head
        self.head = a
```



تابع add last : (اضافه کردن به انتهای لیست)

```
def add_last(self , d):
    a = Node(d)
    if self.head is None: # if empty
        self.head = a
        return
    temp = self.head
    while temp.next:
        temp = temp.next
    temp.next = a
```

تابع add after x : (اضافه کردن بعد از عنصر x)

```
def add_after(self ,d ,x):
    if self.head is None:
        return
    temp = self.head
    while temp.data != x:
        if temp.next is None:
            return
        temp = temp.next
    a = Node(d)
    a.next = temp.next
    temp.next = a
```

تابع del first : (حذف اولین عنصر لیست)

```
def del_first(self):
    if self.head is None:
        return
    self.head = self.head.next
```

تابع del last : (حذف آخرین عنصر لیست)

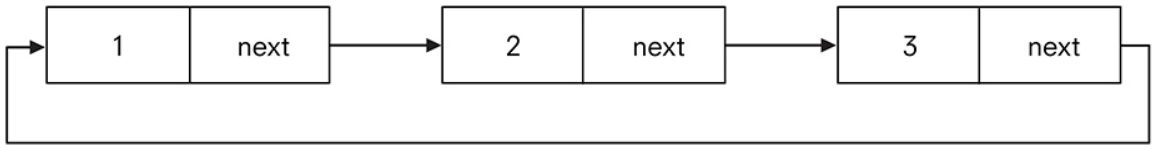
```
def del_last(self):
    if self.head is None:
        return
    if self.head.next is None:
        self.head = None
        return
    temp = self.head
    while temp.next.next:
        temp = temp.next
    temp.next = None
```

تابع del after x : (حذف عنصر بعد از x)

```
def del_after(self, x):
    if self.head is None:
        return
    temp = self.head
    while temp.data != x:
        if temp.next is None:
            return
        temp = temp.next
    if temp.next:
        temp.next = temp.next.next
```

تابع delete : (حذف عنصر x)

```
def delete(self, x):
    if self.head is None:
        return
    if self.head.data == x:
        self.head = self.head.next
        return
    if self.head.next is None:
        return
    temp = self.head
    while temp.next.data == x:
        if temp.next.next is None:
            return
        temp = temp.next
    temp.next = temp.next.next
```



لیست پیوندی یک طرفه حلقوی :

```
class cLinked_List():
    def __init__(self):
        self.head = None
```

تابع add first : (اضافه کردن به ابتدای لیست)

```
def ins_first(self,d):
    a = node(d)
    if self.head is None: # استثنا خالی بودن
        self.head = a
        a.next = a
        return
    a.next = self.head
    temp = self.head
    while temp.next != self.head:
        temp = temp.next
    temp.next = a
    self.head = a
```

تابع add last : (اضافه کردن به انتهای لیست)

```
def ins_last(self,d):
    a = node(d)
    if self.head is None:    # استثنا خالی بودن
        self.head = a
        a.next = a
        return
    a.next = self.head
    temp = self.head
    while temp.next != self.head:
        temp = temp.next
    temp.next = a
```

تابع add after x : (اضافه کردن بعد از عنصر x)

```
def ins_after(self,x,d):
    if self.head == None:              # استثنا خالی بودن
        return
    temp = self.head
    while temp.data != x:
        if temp.next == self.head:    #استثنا عدم وجود X
            return
        temp = temp.next
    a = node(d)
    a.next = temp.next
    temp.next = a
```

تابع add mid : (اضافه کردن به میانه لیست)

```
def ins_mid(self,d):
    if self.head == None:           # استثنا خالی بودن
        return
    if self.head.next == self.head:  # استثنا تک عضوی
        self.ins_first(d)
    else:                           # قرار دادن تعداد اعضا در count
        count = 1
        temp = self.head
        while temp.next != self.head:
            temp = temp.next
            count += 1
    for i in range( count // 2 ):    # تمپ به عنصری اشاره میکند که پس از ان اضافه میشود
        temp = temp.next
    self.ins_after(self,temp.data,d) # اضافه کردن به پس از عنصر تمپ
```

تابع del first : (حذف اولین عنصر لیست)

```
def del_first(self):
    if self.head is None:             # استثنا خالی بودن
        return
    if self.head.next == self.head:  # استثنا تک عضوی
        del(self.head)
        self.head = None
        return
    temp = self.head
    while temp.next != self.head:
        temp = temp.next
    temp.next = self.head.next
    del (self.head)
    self.head = temp.next
```

تابع del last : (حذف آخرین عنصر لیست)

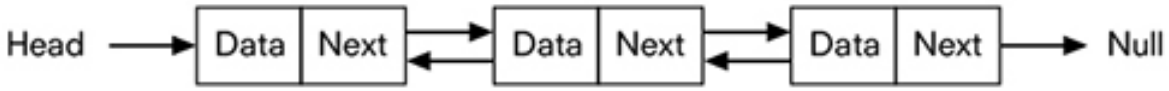
```
def del_last(self):
    if self.head is None:           # استثنا خالی بودن
        return
    if self.head.next == self.head:  # استثنا تک عضوی
        del(self.head)
        self.head = None
        return
    temp = self.head                # کپی از هد
    while temp.next.next != self.head:   #قرار دادن تمپ بر روی یک عنصر قبل از اخر
        temp = temp.next
    del(temp.next)                  # حذف عنصر اخر
    temp.next = self.head           # اتصال به ابتدای لیست
```

تابع del after x : (حذف عنصر بعد از x)

```
def del_after(self,x):
    if self.head is None:           # استثنا خالی بودن
        return
    if self.head.next == self.head:  # استثنا تک عضوی
        return
    temp = self.head                # کپی از هد
    while temp.data != x:           # حرکت تا رسیدن به عنصر X
        if temp.next == self.head:   # استثنا عدم وجود X
            return
        temp = temp.next
    t = temp.next                   # کپی عنصر بعد از X
    temp.next = t.next              # پریدن از روی X
    del(t)                          # پاک کردن X
```

تابع delete : (حذف عنصر x)

```
def delete(self,x):
    if self.head is None:                # استثنا خالی بودن
        return
    if self.head.next == self.head:
        if self.head.data == x:
            del(self.head)
            self.head = None
        return
    temp = self.head
    while temp.next.data != x:
        if temp.next.next == self.head:
            return
        temp = temp.next
    t = temp.next
    temp.next = t.next
    del(t)
```



ابتدا کلاس **نود** دو طرفه تعریف میکنیم :

```
class dnode:
    def __init__(self,data):
        self.back = None
        self.data = data
        self.next = None
```

سپس کلاس **لیست پیوندی** دو طرفه را میسازیم :

```
class dLinkedList:
    def __init__(self):
        self.head = None
```

تابع add first : (اضافه کردن به ابتدای لیست)

```
def ins_first(self,d):
    a = dnode(d)
    if self.head is None:  # استثنا خالی بودن
        self.head = a
        return
    a.next = self.head
    self.head.back = a
    self.head = a
```

تابع add last : (اضافه کردن به انتهای لیست)

```
def ins_last(self,d):
    a = dnode(d)
    if self.head is None:  # استثنا خالی بودن
        self.head = a
        return
    temp = self.head
    while temp.next:
        temp = temp.next
    temp.next = a
    a.back = temp
```

تابع add after x : (اضافه کردن بعد از عنصر x)

```
def ins_after(self,x,d):
    if self.head is None:  # استثنا خالی بودن
        return
    temp = self.head
    while temp.data != x:
        if temp.next is None:
            return
        temp = temp.next
    a = dnode(d)
    if temp.next:
        a.next = temp.next
        temp.next = a
        a.next.back = a
        a.back = temp
    else:
        temp.next = a
        a.back = temp
```

تابع del first : (حذف اولین عنصر لیست)

```
def del_first(self):
    if self.head is None:       # استثنا خالی بودن
        return
    if self.head.next is None:  # استثنا تک عضوی
        del(self.head)
        self.head = None
        return
    a = self.head
    self.head = self.head.next
    self.head.back = None
    del(a)
```

تابع del last : (حذف آخرین عنصر لیست)

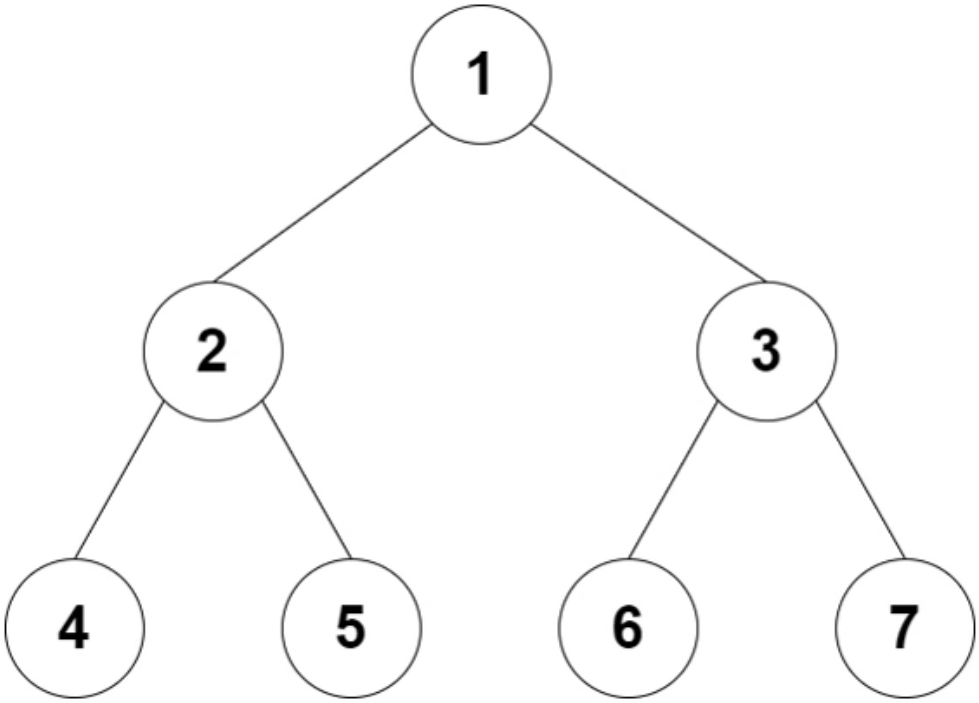
```
def del_last(self):
    if self.head is None:       # استثنا خالی بودن
        return
    if self.head.next is None:  # استثنا تک عضوی
        del(self.head)
        self.head = None
        return
    temp = self.head
    while temp.next:
        temp = temp.next
    temp.back.next = None
    del(temp)
```

تابع del after x : (حذف عنصر بعد از x)

```
def del_after(self,x):
    if self.head is None:
        return
    if self.head.next is None:
        return
    temp = self.head
    while temp.data != x:
        if temp.next is None:
            return
        temp = temp.next
    if temp.next:
        temp.next = temp.next.next
        a = temp.next.back
        temp.next.back = temp
        del(a)
```

تابع delete : (حذف عنصر x)

```
def delete(self,x):
    if self.head is None: # استثنا خالی بودن
        return
    if self.head.data == x:
        a = self.head
        self.head = a.next
        if a.next:
            a.next.back = None
        del(a)
        return
    temp = self.head
    while temp.data != x:
        if temp.next is None:
            return
        temp = temp.next
        if temp.next:
            temp.back.next = temp.next
            temp.next.back = temp.back
            del(temp)
        else:
            temp.back.next = None
            del(temp)
```



ساختمان داده درخت :

ساختمان داده ای است که یک ریشه دارد و بقیه اجزا آن نیز درخت میباشد.

اجزای درخت :

- node (گره)
- یالها

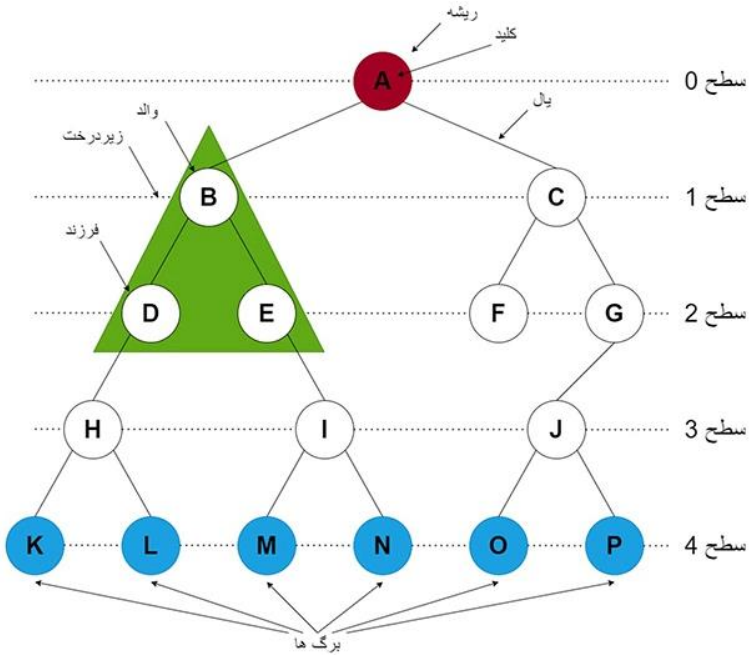
درجه نود : تعداد فرزندان نود

درجه درخت : بیشترین درجه نودهایش

درخت باینری : درختی با درجه 2

ارتفاع درخت : فاصله بین پایین ترین گره تا ریشه

سطح درخت : فاصله بین ریشه تا پایین ترین گره



درخت باینری پر (full binary tree) : درختی است که تمامی گره های آن یا برگ هستند یا درجه دو

درخت پر با ارتفاع h

حداقل : 2n + 1

برگ : n + 1

درجه 2 : n

حداکثر : $2^{n+1} - 1$

برگ : 2^n

درجه 2 : $2^n - 1$

درخت باینری کامل (complete binary tree) : درختی پر است که تمامی گره های آن از چپ به راست پر باشند.

درخت پر و کامل با ارتفاع h

حداقل : $2^n + 1$

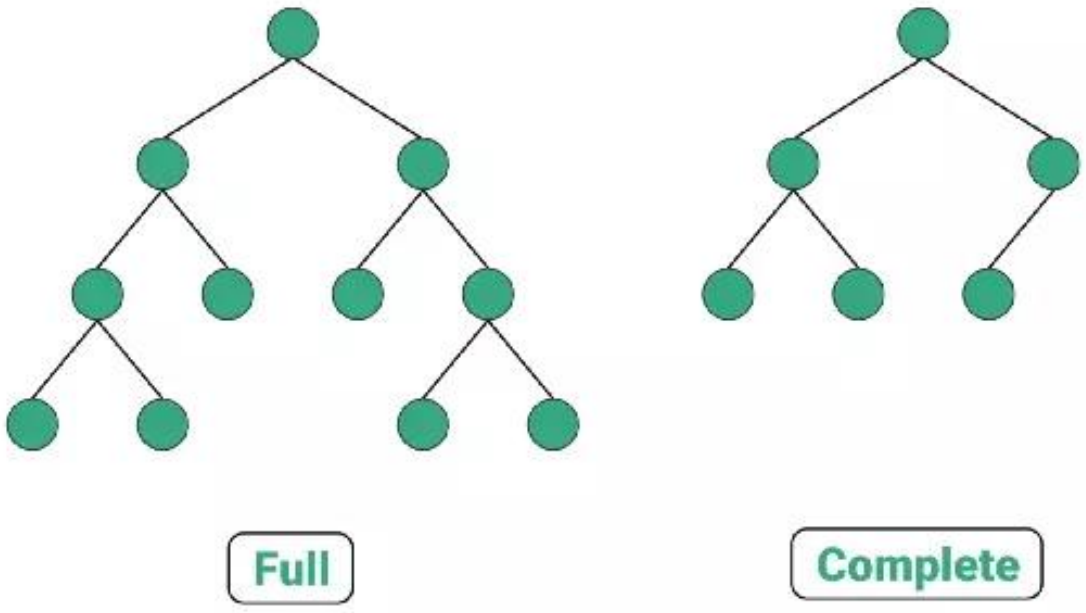
برگ : $2^{n-1} + 1$

درجه 2 : 2^{n-1}

حداکثر : $2^{n+1} - 1$

برگ : 2^n

درجه 2 : $2^n - 1$



تعریف کلاس باینری نود :

```
class bnode:
    def __init__(self,d):
        self.Lchild = None
        self.data = d
        self.Rchild = None
```

تعریف کلاس درخت باینری و متد افزودن به چپ :

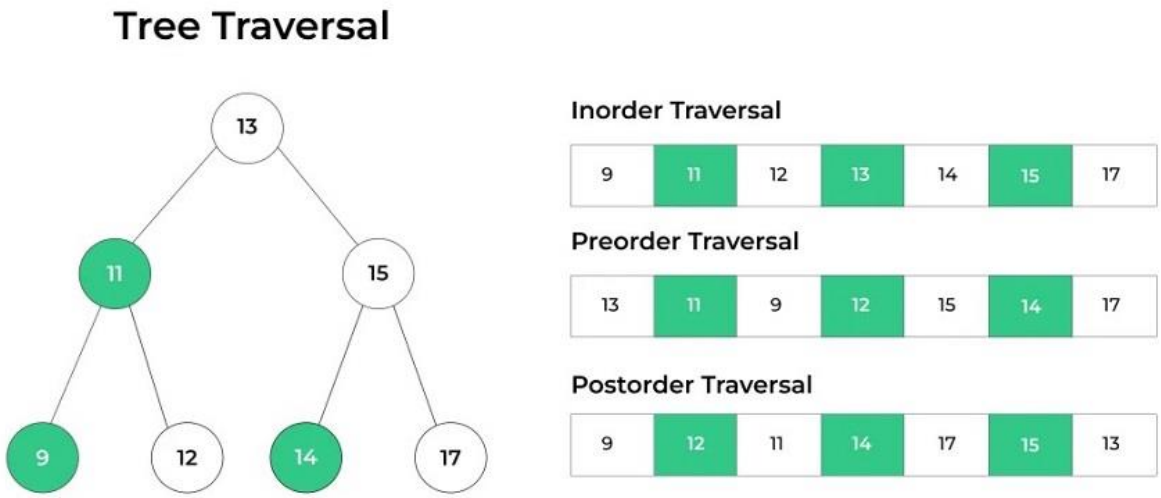
```
class btree:
    def __init__(self):
        self.root = None

    def insLeft(self,d):
        if self.root is None:  #شرط خالی بودن
            self.root = bnode(d) #در نظر گرفتن به عنوان ریشه در صورت خالی بودن
        else:
            temp = self.root # کپی از ریشه
            while temp.Lchild: #حرکت تا چپ ترین خانه
                temp = temp.Lchild
            temp.Lchild = bnode(d) # اضافه کردن فرزند چپ به چپ ترین خانه
```

متد افزودن به راست :

```
def insRight(self,d):
    if self.root: #شرط خالی نبودن
        temp = self.root
        while temp.Rchild: #حرکت تا راست ترین خانه
            temp = temp.Rchild
        temp.Rchild = bnode(d) # اضافه کردن فرزند راست به راست ترین خانه
    else:
        self.root = bnode(d) #در نظر گرفتن به عنوان ریشه در صورت خالی بودن
```

چهار نوع از الگوریتم های پیمایش درخت :

1. پیمایش میان‌ترتیب (Inorder Traversal) : در ابتدا همه گره‌های درون زیردرخت سمت چپ را بازدید می‌کنیم و سپس گره فعلی را نیز قبل از بازدید هر گره‌ای در زیردرخت‌های سمت راست بازدید می‌کنیم.
2. پیمایش پیش‌ترتیب (Preorder Traversal) : قبل از اینکه هر گره درون زیردرخت‌های سمت راست یا چپ را بازدید کنیم باید گره فعلی را بررسی کنیم.
3. پیمایش پس‌ترتیب (Postorder Traversal) : بعد از اینکه همه گره‌های درون زیردرخت‌های چپ و راست را بازدید کردیم، گره فعلی را بازدید می‌کنیم.
4. پیمایش سطح‌ترتیب :(Level order Traversal) همه گره‌ها را به صورت سطح به سطح بازدید می‌کنیم. در هر سطح بازدید گره‌ها را از سمت چپ به راست انجام می‌دهیم.



متد نمایش NLR :

```
def displayNLR(self): # Node-Left-Right نمایش با ترتیب
    self.showNLR(self.root) #صدا زدن متد کمکی
def showNLR(self,root):
    if root: #شرط خالی نبودن
        print(root.data,end=" ") #چاپ ریشه

        self.showLNR(root.Lchild) #تابع بازگشتی با در نظر گرفتن
                                  #فرزند چپ به عنوان ریشه

        self.showLNR(root.Rchild) #تابع بازگشتی با در نظر گرفتن
                                  #فرزند راست به عنوان ریشه
```

متد نمایش LNR :

```
def displayLNR(self): #Left-Node-Right نمایش با ترتیب
    self.showLNR(self.root) #صدا زدن متد کمکی
def showLNR(self,root): #متد کمکی
    if root: #شرط خالی نبودن
        self.showLNR(root.Lchild) #تابع بازگشتی با در نظر گرفتن
                                  #فرزند چپ به عنوان ریشه

        print(root.data,end=" ") #چاپ ریشه

        self.showLNR(root.Rchild) #تابع بازگشتی با در نظر گرفتن
                                  #فرزند راست به عنوان ریشه
```

متد نمایش LRN :

```
def displayLRN(self): #Left-Right_Node نمایش با ترتیب
    self.showLRN(self.root) #صدا زدن متد کمکی
def showLRN(self,node): #متد کمکی
    if node: #شرط خالی نبودن
        self.showLRN(node.Lchild) #تابع بازگشتی با در نظر گرفتن
                                  #فرزند چپ به عنوان ریشه

        self.showLRN(node.Rchild) #تابع بازگشتی با در نظر گرفتن
                                  #فرزند راست به عنوان ریشه
        print(node.data,end = ' ') #چاپ ریشه
```

متد نمایش level order :

```
def levelOrder(self): #Level نمایش با ترتیب
    list = [] # ایجاد لیست
    temp = self.root # کپی از ریشه
    if temp is None: # شرط خالی بودن
        return
    list.append(temp) #اضافه کردن ریشه به لیست
    while list: #شرط خالی نبودن لیست
        k = list.pop(0) #برداشتن اولین ایتم لیست
        print(k.data,end = ' ') #چاپ ایتم

        if k.Lchild: #شرط وجود داشتن فرزند چپ
            list.append(k.Lchild) #اضافه کردن فرزند چپ به لیست
        if k.Rchild: #شرط وجود داشتن فرزند راست
            list.append(k.Rchild) #اضافه کردن فرزند راست به لیست
```

افزودن فرزند چپ به عنصر مورد نظر :

```
def insAfterL(self,x,d): #افزودن فرزند چپ به نود مورد نظر
    self.pinsAfterL(self.root,x,d) # صدا زدن متد کمکی
def pinsAfterL(self,node,x,d): #متد کمکی
    if node: #شرط خالی نبودن
        if node.data == x: #تطابق نود فعلی با نود مورد نظر
            temp = node.Lchild #کپی از فرزند چپ فعلی
            node.Lchild = bnode(d) #قرار دادن داده جدید به عنوان فرزند چپ جدید
            node.Lchild.Lchild = temp # قرار دادن فرزند چپ قبلی به عنوان نوه چپ
        self.pinsAfterL(node.Lchild,x,d) #جستجوی در فرزند های چپ برای رسیدن به نود مورد نظر
        self.pinsAfterL(node.Rchild,x,d) #جستجوی در فرزند های راست برای رسیدن به نود مورد نظر
```

افزودن فرزند راست به عنصر مورد نظر :

```
def insAfterR(self,x,d): #افزودن فرزند راست به نود مورد نظر
    self.pinsAfterR(self.root,x,d) # صدا زدن متد کمکی
def pinsAfterR(self,node,x,d): #متد کمکی
    if node: #شرط خالی نبودن
        if node.data == x: #تطابق نود فعلی با نود مورد نظر
            temp = node.Rchild #کپی از فرزند راست فعلی
            node.Rchild = bnode(d) #قرار دادن داده جدید به عنوان فرزند راست جدید
            node.Rchild.Rchild = temp # قرار دادن فرزند راست قبلی به عنوان نوه راست
        self.pinsAfterR(node.Rchild,x,d) #جستجوی در فرزند های راست برای رسیدن به نود مورد نظر
        self.pinsAfterR(node.Lchild,x,d) #جستجوی در فرزند های راست برای رسیدن به نود مورد نظر
```

حذف چپ ترین نود :

```
def delLeft(self): #حذف چپ ترین نود
    if self.root is None: #شرط خالی بودن
        return print("empty")
    if self.root.Lchild is None: #شرط عدم وجود فرزند چپ
        temp = self.root.Rchild # کپی از فرزند راست
        del self.root #حذف ریشه
        self.root = temp #قرار دادن فرزند راست به عنوان ریشه جدید
        return
    temp = self.root #کپی از ریشه
    while temp.Lchild.Lchild: #حرکت تا یک خانه قبل از چپ ترین نود
        temp = temp.Lchild
    temp1 = temp.Lchild #کپی از چپ ترین نود
    temp.Lchild = None #حذف چپ ترین نود
    del temp1
```

حذف راست ترین نود :

```
def delRight(self): #حذف راست ترین نود
    if self.root is None: #شرط خالی بودن
        return print("empty")
    if self.root.Rchild is None: #شرط عدم وجود فرزند راست
        temp = self.root.Lchild # کپی از فرزند چپ
        del self.root #حذف ریشه
        self.root = temp #قرار دادن فرزند راست به عنوان ریشه جدید
        return
    temp = self.root #کپی از ریشه
    while temp.Rchild.Rchild: #حرکت تا یک خانه قبل از راست ترین نود
        temp = temp.Rchild
    temp1 = temp.Rchild #کپی از راست ترین نود
    temp.Rchild = None #حذف راست ترین نود
    del temp1
```

حذف عنصر x از درخت :

```
def delete_x(self, x):
    if self.root is None:
        print('empty')
        return None
    else:
        self.pdelete(self.root, x)
def pdelete(self, node, x):
    if node != None:
        if node.Lchild:
            if node.Lchild.data == x:
                del(node.Lchild)
                node.Lchild = None
                return None
            self.pdelete(node.Lchild, x)
            self.pdelete(node.Rchild, x)
        if node.Rchild:
            if node.Rchild.data == x:
                del(node.Rchild)
                node.Rchild = None
                return None
            self.pdelete(node.Lchild, x)
            self.pdelete(node.Rchild, x)
        if node.data == x:
            node = None
            return
```

درخت باینری سرچ (binary search tree) :

کاربرد : جستجو در دیتاست های بزرگ

- مقدار گره پدر از فرزندان چپ بیشتر و از فرزندان راست کمتر است.

الگوریتم اضافه کردن :

1. مقدار مساوی در گره ها وجود نداشته باشد.
2. اضافه کردن فقط به جاهای خالی اتفاق می افتد یعنی یا باید به برگها یا به گره های درجه یک اضافه گردند. (اضافه کردن به گره درجه 2 ممنوع است)

الگوریتم پاک کردن :

1. پاک کردن برگ
2. پاک کردن گره درجه یک
3. پاک کردن گره درجه دو

1. پاک کردن برگ :
    - جستجو
    - یافتن پدر
    - قطع ارتباط بین پدر و فرزند
    - پاک کردن فرزند

2. پاک کردن گره درجه یک :
    - جستجو
    - یافتن پدر
    - کپی از نوه
    - قطع ارتباط بین پدر و فرزند
    - ارتباط بین پدر و نوه
    - حذف گره

3. پاک کردن گره درجه دو :
    - جستجو
    - یافتن پدر
    - کپی کوچکترین نوه راست (بزرگترین نوه چپ)
    - قطع ارتباط بین پدر و نوه برگدار
    - حذف گره

الگوریتم تبدیل لیست به BST :

- ساده ترین : اولین عنصر به عنوان ریشه درخت انتخاب شود. سپس بقیه عناصر لیست به ترتیب بر اساس الگوریتم اضافه کردن به درخت متصل شوند.

تعریف کلاس BST و متد افزودن :

```
class BST:
    def __init__(self):
        self.root = None
        self.list = []

    def add(self , x):
        if self.root is None: #empty
            self.root = bnode(x)
            self.list.append(x)
        else:
            self.padd(self.root , x) #صدا زدن متد کمکی
    def padd(self , root , x):
        if x > root: #اگر دیتای جدید از ریشه بزرگتر باشد
            if root.Rchild == None: #چک کردن فرزند راست
                root.Rchild = bnode(x)
                self.list.append(x)

            else:
                self.padd(root.Rchild,x) #اگر پدر فرزند راست داشته باشد بوسیله
                                         #تابع بازگشتی فرزند راستش به عنوان ریشه در
                                         # نظر گرفته میشود و شروط دوباره چک میشوند

        if x < root: #اگر دیتای جدید از ریشه کوچکتر باشد
            if root.Lchild is None: #چک کردن فرزند راست
                root.Lchild = bnode(x)
                self.list.append(x)

            else:
                self.padd(root.Lchild,x)#اگر پدر فرزند چپ داشته باشد بوسیله
                                        #تابع بازگشتی فرزند چپش به عنوان ریشه در
                                        # نظر گرفته میشود و شروط دوباره چک میشوند
        return
```

متد نشان دادن :

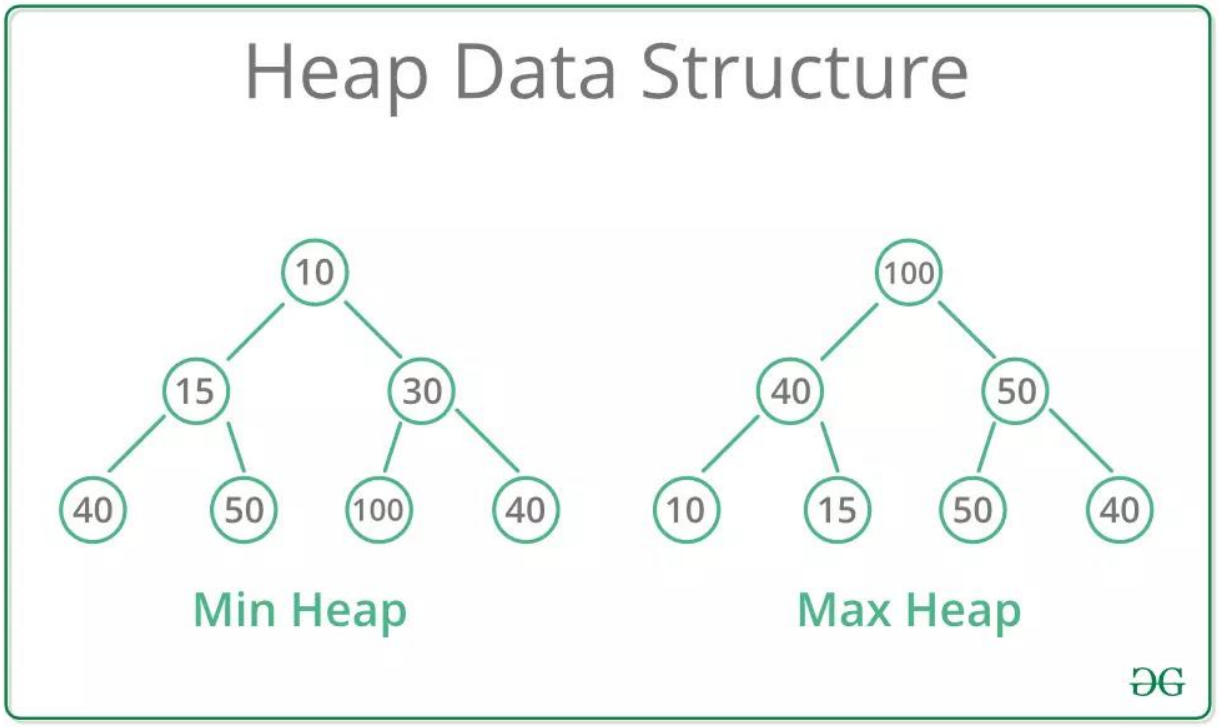
```
def show(self):
    return self.pshow(self.root) #صدا زدن متد کمکی
def pshow(self,root):
    if root: #شرط خالی نبودن
        self.pshow(root.Rchild) #تابع بازگشتی با ورودی فرزند راست به عنوان ریشه
        print(root.data, end=' ')#چاپ ریشه فعلی
        self.pshow(root.Lchild) #تابع بازگشتی با ورودی فرزند چپ به عنوان ریشه
    return
```

مثال : تابعی بنویسید که یک BST را از ورودی گرفته و لیست مربوط به آن را بازگرداند.

```
def CreateList(Tree):
    print(Tree.list)
```

مثال : برعکس تابع قبل.

```
def createTree(A):
    Tree = BST()
    for i in A:
        Tree.add(i)
    return Tree
```



درخت heap :

Max heap : مقدار پدر از مقدار فرزندان بیشتر باشد.

Min heap : مقدار پدر از مقدار فرزندان کمتر باشد.

الگوریتم اضافه کردن در درخت Heap :

1. اضافه کردن عنصر به انتهای لیست
2. Heapify کردن عنصر جدید

مراحل heapifydu کردن :

1. بدست آوردن محل قرارگیری پدر
2. مقایسه با مقدار پدر؛ اگر بزرگتر بود:
    1. جابجا میکنیم
    2. مکان پدر را heapify میکنیم

مراحل heapifyud کردن :

3. بدست آوردن محل قرارگیری فرزندان چپ و راست
4. اگر فرزندان چپ و راست وجود داشت :
    1. بزرگترین عنصر را پیدا کند و با پدر جابجا کنیم
    2. فرزند جابجا شده را heapify کنیم

تعریف کلاس max heap و متد افزودن :

```
class MaxHeap:
    def __init__(self):
        self.list = [] # ایجاد یک لیست برای ذخیره عناصر هیپ

    def insert(self,x): # افزودن عنصر به هیپ
        self.list.append(x) # افزودن عنصر به انتهای لیست
        self.Heapifydu(len(self.list)-1) # متعادل کردن هیپ به سمت بالا
```

متد heapifydu :

```
def Heapifydu(self,n): # متعادل کردن به سمت بالا
    p = (n-1) // 2 # محاسبه اندیس والد
    if p != n: # چک کردن اینکه والد با خود عنصر متفاوت باشد
        if self.list[n] > self.list[p]: # اگر مقدار عنصر بزرگتر از والد باشد
            self.list[n],self.list[p] = self.list[p],self.list[n] # جابجایی عنصر با والد
            self.Heapifydu(p) # ادامه متعادل‌سازی از موقعیت والد
```

متد heapifyud :

```
def Heapifyud(self,n): # متعادل کردن به سمت پایین
    e = 2*n + 1 # محاسبه اندیس فرزند چپ
    r = 2*n + 2 # محاسبه اندیس فرزند راست
    if e < len(self.list): # بررسی اینکه فرزند چپ در محدوده باشد
        if r < len(self.list): # بررسی اینکه فرزند راست در محدوده باشد
            if self.list[n] < self.list[e] and self.list[r] < self.list[e]: # چک کردن بزرگترین فرزند
                self.list[e],self.list[n] = self.list[n],self.list[e] # جابجایی با فرزند چپ
                self.Heapifyud(e) # ادامه متعادلسازی از موقعیت فرزند چپ
            elif self.list[n] < self.list[r] and self.list[r] > self.list[e]: # بررسی فرزند راست
                self.list[r],self.list[n] = self.list[n],self.list[r] # جابجایی با فرزند راست
                self.Heapifyud(r) # ادامه متعادلسازی از موقعیت فرزند راست
            else:
                return # اگر نیازی به جابجایی نبود
        else:
            if self.list[e] > self.list[n]: # بررسی تنها فرزند چپ
                self.list[n],self.list[e] = self.list[e],self.list[n] # جابجایی با فرزند چپ
                return
            else:
                return
```

الگوریتم حذف کردن :

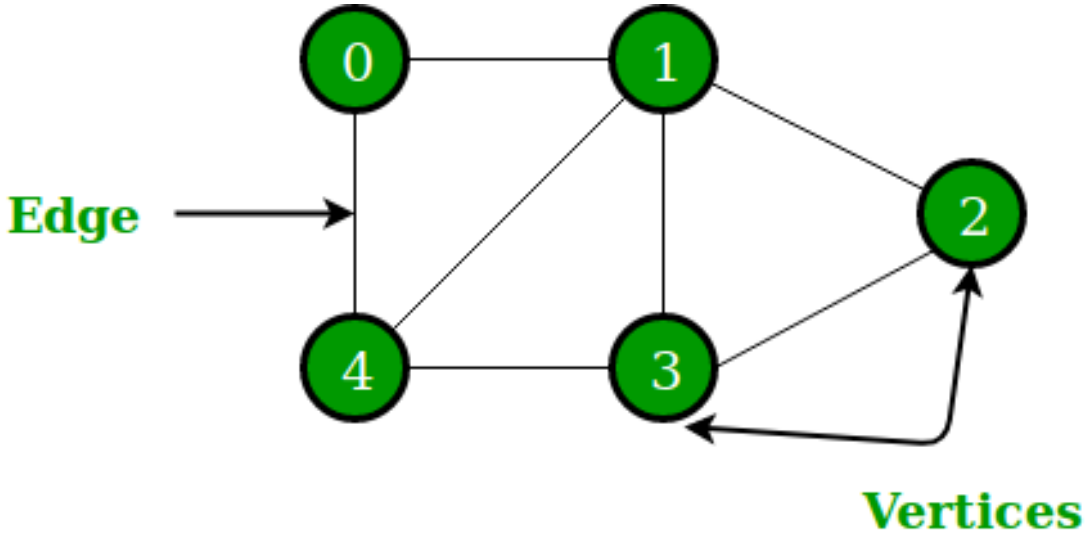
حذف ریشه :

1. جابجایی آخرین عنصر با اولین عنصر
2. حذف کردن آخرین عنصر
3. انجام Heapifyud برای عنصر اول

```
def delRoot(self): # حذف عنصر ریشه
    if len(self.list) == 0: # اگر هیپ خالی باشد
        return
    if len(self.list) == 1: # اگر هیپ تنها یک عنصر داشته باشد
        self.list.pop() # حذف تنها عنصر موجود
        return
    e = len(self.list) -1 # محاسبه اندیس آخرین عنصر
    self.list[e] , self.list[0] = self.list[0] , self.list[e] # جابجایی ریشه با آخرین عنصر
    self.list.pop() # حذف آخرین عنصر که اکنون ریشه است
    self.Heapifyud(0) # متعادلسازی هیپ به سمت پایین از ریشه
```

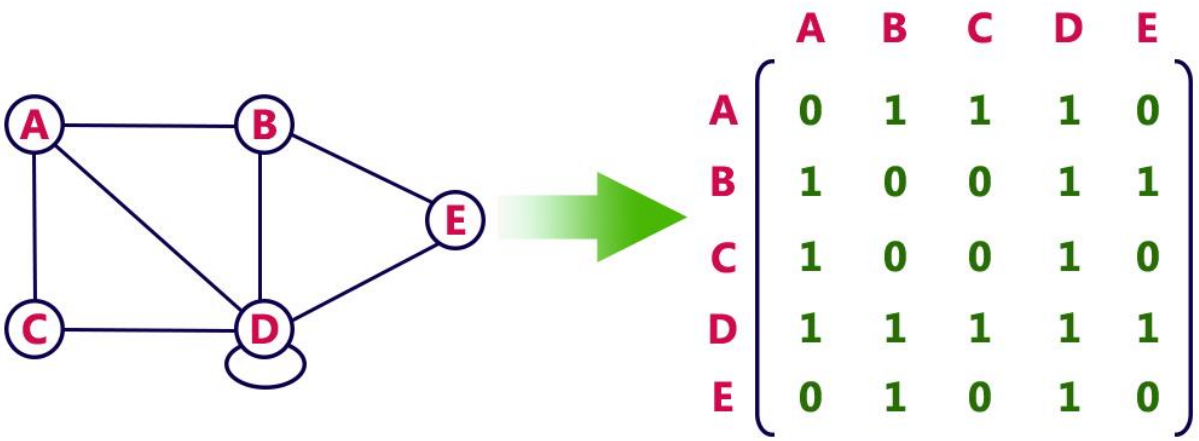
ساختمان داده گراف (Graph) :

G (V (مجموعه راس ها) , E (مجموعه یالها))



گراف بدون جهت و بدون وزن :

نمایش بوسیله ماتریس مجاورت : ماتریس مربعی است که به تعداد راس ها سطر و ستون دارد.
بین هر راس که یال موجود باشد 1 و بقیه جا ها 0 قرار دارد.



درجه هر راس = تعداد یک های سطر آن راس

تعداد یالها = مجموع درجات / 2 یا تعداد یالها = تعداد یک های ماتریس / 2

تعریف کلاس گراف و متد افزودن یال :

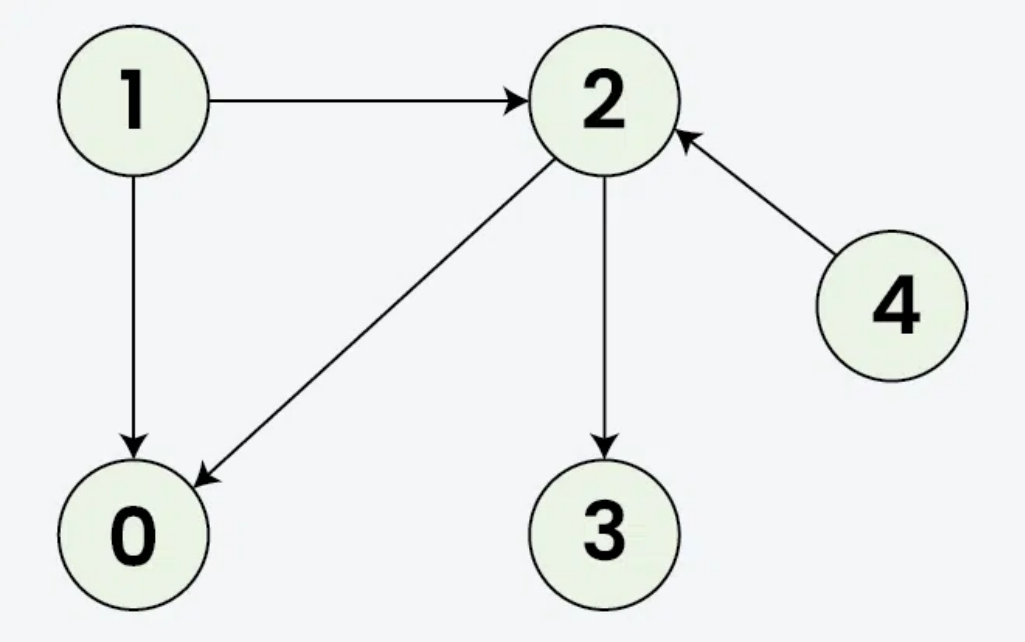
```
class Graph:
    def __init__(self,n):
        self.M = [[0]*n for _ in range(n)] # n*n ایجاد ماتریس

    def insertEdge(self,s,t): # افزودن یال
        if s<len(self.M[0]) and t<len(self.M[0]): #چک کردن محدوده مبدا و مقصد ورودی
            self.M[s][t] = 1 #ایجاد یال در هردو جهت
            self.M[t][s] = 1
```

متد حذف یال :

```
def delEdge(self,s,t): #حذف یال
    if s<len(self.M[0]) and t<len(self.M[0]): #چک کردن محدوده مبدا و مقصد ورودی
        self.M[s][t] = 0
        self.M[t][s] = 0
```

متد شمارش یال :

```
def countEdge(self):
    c = 0
    for i in range(len(self.M[0])):
        c = sum(self.M[i]) + c
    c = c/2
    if c/2 != 0:
        c += 0.5
    return print(c)
```



گراف جهت دار :

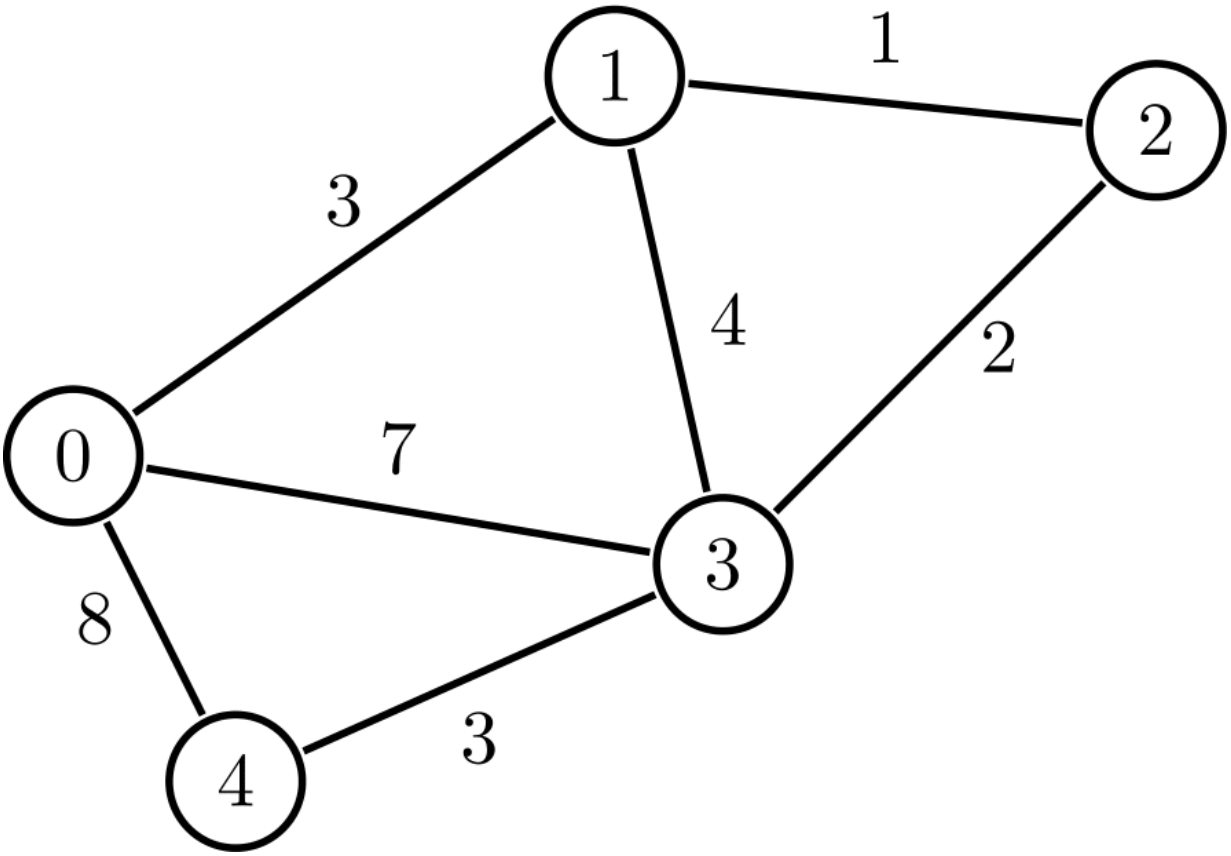
- چاه : راسی است که خروجی ندارد.
- چشمه : راسی است که ورودی ندارد.
- تعداد یالها = تعداد یک های ماتریس
- درجه هر راس در ماتریس مجاورت = تعداد یک های ستون آن + تعداد یک های سطر های آن

تعریف کلاس گراف جهت دار و متد افزودن یال :

```
class GraphD: #گراف جهت دار
    def __init__(self,n):
        self.M = [[0]*n for _ in range(n)] # n*n ایجاد ماتریس

    def insertEdge(self,s,t): # افزودن یال
        if s<len(self.M[0]) and t<len(self.M[0]): #چک کردن محدوده مبدا و مقصد ورودی
            self.M[s][t] = 1 #ایجاد یال با یک جهت
```

متد حذف یال :

```
def delEdge(self,s,t): #حذف یال
    if s<len(self.M[0]) and t<len(self.M[0]): #چک کردن محدوده مبدا و مقصد ورودی
        self.M[s][t] = 0 #حذف یک جهت یال
```



گراف وزن دار :

- بجای یک در ماتریس مقدار وزن را قرار میدهیم.
- بر روی هر یال وزن آن را قرار میدهیم.

تعریف کلاس گراف وزن دار و متد افزودن یال :

```
class GraphW: #گراف وزن دار
    def __init__(self,n):
        self.M = [[0]*n for _ in range(n)] # n*n ایجاد ماتریس

    def insertEdge(self,s,t,w): # افزودن یال وزن دار
        if s<len(self.M[0]) and t<len(self.M[0]):
            self.M[s][t] = w #قرار دادن وزن در خانه مورد نظر
```

متد حذف یال :

```
def delEdge(self,s,t): #حذف یال
    if s<len(self.M[0]) and t<len(self.M[0]): #چک کردن محدوده مبدا و مقصد ورودی
        self.M[s][t] = 0
        self.M[t][s] = 0
```

گراف کامل : (بدون وزن یا وزن دار)

گراف بدون جهت که از هر راس به تمامی راس ها غیر از خودش یال دارد.

اگر گراف کاملی n راس داشته باشد :

- درجه هر راس = n – 1
- مجموع درجات = n (n – 1)
- تعداد یال ها = n (n -1) / 2
- فقط قطر اصلی صفر است.

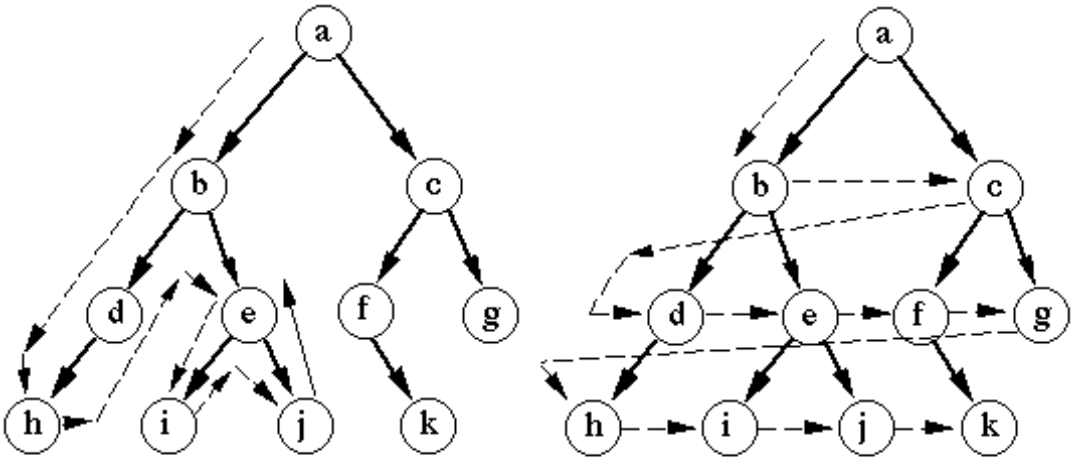
پیمایش گراف
- سطحی (صف) ← BFS
- عمقی (پشته) ← DFS

الگوریتم‌های جست‌وجوی عمق‌اول : (Depth-First Search |DFS)

در این نوع از الگوریتم‌ها عمل پیمایش را از گره اول شروع می‌کنیم. در ابتدا قبل از عقب نشینی از هر شاخه، همه گره‌های آن شاخه را به ترتیب تا عمیق‌ترین سطح ممکن بازدید می‌کنیم. سپس گره‌های همه شاخه‌های دیگر نیز طبق همین روش باید بازدید شوند.

الگوریتم جست‌وجوی سطح‌اول : (Breadth-First Search |BFS)

این الگوریتم نیز از ریشه شروع به کار می‌کند. روش کار به این صورت است که قبل از رفتن به هر سطحی در درخت، در ابتدا همه گره‌های سطح بالاتر بررسی می‌شود.



شما میتوانید با مراجعه به لینک زیر به تمامی کد های سر کلاس، تمرینات و تکالیف بر روی گیت هاب دسترسی داشته باشید.

کلیک کنید

با تشکر از همراهی شما