جزوه ساختمان داده

Data Structure

استاد: دکتر میرروشندل

نویسنده: امیرمحمد عزتی – سید امیرکسائی

**انواع مرتب سازی ها (Sort)**

1. Insertion Sort

InsertionSort (array) :

    for i in range (2, len(array) ) :

        Key = array[i]

        j = i - 1

*while*(j > 0 and array[j] > key):

            swap(array[j], array[j + 1])

            j--

1. Merge Sort

mergeSort(array, begin, end):

*if* begin > end:

*return* null

    mid = (begin + end)/2

    mergeSort(array, begin, mid)

    mergeSort(array, mid, end)

    merge(array, begin, end)

merge(array, begin, end):

    tmp = []

    i = begin

    mid = (begin + end)/2

    j = mid + 1

    k = 0

*while*(i < mid and j < end):

*if* array[i] <= array[j]:

            tmp[k++] = array[i++]

*else*:

            tmp[k++] = array[j++]

*while*(j <= end):

        tmp[k++] = array[j++]

*while*(i <= mid):

        tmp[k++] = array[i++]

    array = tmp

1. Bubble Sort

bubbleSort(array):

*for* i *in* range(1, n):

*for* j *from* n to i+1:

*if* array[j-1] > array[j]:

                swap(array[j], array[j-1])

Hanoi Tower

hanoiTower(n, start, goal, help):

*if* n==1:

        print(start + "-->" + goal)

*return*

    hanoiTower(n-1, start, help, goal)

    print(start + "-->" + goal)

    hanoiTower(n-1, help, start, goal)

**Order**

**Master theorem**

**Elementary data structures**

1. Linked list:

*class* LinkedList:

    DataType data

    LinkedList next

Insert last

Linkedlist insertLast(Linkedlist l, int newData):

    Linkedlist temp = new Linkedlist

    temp.data = newData

    temp2 = l

*while* temp2.next != null:

        temp2 = temp2.next

    temp.next = temp

*return* l

Insert first

Linkedlist insertFirst(Linkedlist l, int newData):

    tmp = LinkedList()

    tmp.data = newData

    tmp.next = l

*return* temp

Get Length

*def* getLength(LinkedList l):

    length =  0

*while* l is not null:

        l = l.next

        length++

*return* length

*def* getLength(LinkedList l):

*if* l is null:

*return* 0

*return* 1 + getLength(l.next)

Delete

LinkedList delete(LinkedList l, data):

*if* l is null:

*return* l

*if* l.data = data:

*return* l.next

    LinkedList tmp = l

*while* tmp.next is not null:

*if* tmp.next.data == data:

            tmp.next = tmp.next.next

        tmp = tmp.next

*return*

Copy

LinkedList copy(LinkedList l):

*if*  l is null:

*return* null

    tmp = LinkedList()

    tmp.data = l.data

    tmp.next = copy(l.next)

*return* tmp

Merging two sorted linked list

LinkedList merge(LinkedList l1, LinkedList l2):

*if* l1 is null:

*return* l2

*if* l2 is null:

*return* l1

*if* l1.data <= l2.data:

        l1.next = merge(l1.next, l2)

*return* l1

*else*:

        l2.next = merge(l1, l2.next)

*return* l2

Reverse

linkedList reverse(linkedList l):

*if* l is null:

*return* null

*if* l.next is null:

*return* l

    tmp = reverse(l.next)

    l.next.next = l

    l.next = null

*return* tmp

Purge1

linkedList purge(linkedList l):*# removes duplicate data*

    i, j = linkedList()

    i = l

*while* i.next is not null:

        j = i.next

*while* j.next is not null:

*if* i.data == j.data:

                delete(j)

*else*:

                j = j.next

        i = i.next

*return* l

Purge2

linkedList purge(linkedList l):

    meregSort(l)

    tmp = l

*while* tmp.next is not null:

*while* tmp == tmp.next:

            delete(tmp.next)

        tmp = tmp.next

*return* l

1. Queue

Is Empty

isEmpty(queue q):

*if* q.head == q.tail + 1:

*return* True

*return* False

Enqueue and Dequeue

enqueue(q, data):

    q.tail = q.tail + 1

    q[q.tail] = data

dequeue(q):

    return q[head++]

1. Rounded Queue

Next

next(int k):

*return* (k+1)%max

Is Empty

isEmpty(q):

*return* next(q.tail) == q.head

Is Full

isFull(q):

*return* next(next(q.tail)) == q.head

Enqueue

enqueue(data):

*if* isFull():

*return* -1

    q.tail = next(q.tail)

    q[q.tail] = data

Dequeue

dequeue():

*if* isEmpty():

*return* "ERROR"

    tmp = q[q.head]

    q.head = next(q.head)

*return* tmp

1. General linked List

Get general List from expression

getList(input):

    l = new List

    t = l

*if* data is null:

*return* null

*while* input.data is not ")":

*if* input.data is "(":

            t.next.down = getList(input.next)

*else* *if* input.data is not ",":

            t.next.data = input.data

        input = input.next

        t = t.next

*return* l

Print expression of general linked list

printList(gerenalLinkedList l):

    print("(")

*if* l is null:

    print(")")

*return*

    l = l.next

*while* l is not null:

*if* l.down is null:

*if* l.down.next is null:

                print(l.data)

*else*:

                print(l.data + ",")

*else*:

            printList(l.down)

        l = l.next

    print(")")

1. Stack

Train station example

data = []

train = []

tempTrain = []

trainStation = []

possible = True

n = int(input("Enter number of elements : "))

print("Please enter data (press enter after each element): ")

*for* i *in* range(n):

    x = int(input())

    data.append(x)

    train.append(i + 1)

    tempTrain.append(i + 1)

*def* check(index, pr):

*if* index == n:

*return*

*if* data[index] in train:

*while* True:

*if* pr:

                print("push " + str(train[0]))

            trainStation.append(train[0])

*if* train[0] == data[index]:

                train.pop(0)

*break*

            train.pop(0)

*if* pr:

            print("pop " + str(trainStation[-1]))

        trainStation.pop(-1)

*elif* data[index] == trainStation[-1]:

*if* pr:

            print("pop " + str(trainStation[-1]))

        trainStation.pop(-1)

*else*:

        global possible

        possible = False

*return*

    check(index + 1, pr)

check(0, False)

*if* possible:

    train = tempTrain

    check(0, True)

*else*:

    print("impossible")

1. Tree

درخت گراف همبندی است که نه دور دارد نه حلقه.

class node:

    data

    children = [node]

class Tree:

    root = node()

getHeight(binaryTree):

    if t is null:

        return -1

    return 1 + max(getHeight(t.left), getHeight(t.right))

*اثبات: به روش استقرا*

***ریشه:*** *گره ای است که یال ورودی ندارد. پدر ندارد!*

***برگ:*** *گره ای که یال خروجی ندارد. )بچه ندارد(*

***ارتفاع یک گره:*** *طولانی ترین مسیر از آن گره به یکی از برگ ها*

***ارتفاع درخت:*** *ارتفاع ریشه*

***عمق یک گره:*** *طول مسیر از آن گره به ریشه*

***درخت K تایی:*** *درختی که هر گره حداکثر K فرزند داشته باشد.*

***درخت K کامل:*** *درختی که هر گره آن دقیقا k فرزند داشته باشد یا فرزند نداشته باشد.*

***درخت متوازن:*** *درختی که عمق برگ های آن حداکثر یک واحد اختلاف داشته باشد.*

***درخت کاملا متوازن:*** *درختی که عمق برگ هایش با هم برابر باشد.*

***درخت مرتب(ordered Tree):*** *درختی که بین فرزندان آن ترتیب قائل شویم.*

***درخت دودویی(binary):*** *درخت دوتایی مرتب.*

*سوال: تعداد برگ های درخت k تایی کامل با n گره را محاسبه کنید.*

*ترتیب پیمایش(ملاقات) گره ها:*

1. *پیش ترتیب preorder :*

*اول ریشه سپس فرزندان*

1. *پس ترتیب postorder:*

*اول فرزندان سپس ریشه*

1. *میان ترتیب inoerder:*

*اول فرزند چپ سپس ریشه سپس بقیه فرزندان*

1. *ترتیب سطحی levelorder*

*به ترتیب سطح*

levelOrder(Tree t):

    q.empty()

    q.enqueue(t)

    while(q is not empty):

        v = q.dequeue()

        print(v)

        q.enqueueAll(v.child)

1. *درخت دودویی معادل:*

*درختی است که گره هایش همان گره های درخت اولیه هستند! فرزند سمت چپ در درخت دودویی معادل سمت چپ ترین فرزند درخت k تایی است و فرزند سمت راست درخت دودویی برادر سمت راستی گره مورد نظر است.*

class node:

    data

    left = node()

    right = node()

class binaryTree:

    root = node()

Pre Order

preOrder(binaryTree t):

    if t is null:

        return

    print(t.data)

    preOrder(t.left)

    preOrder(t.right)

Post Order

postOrder(binaryTree t):

    if t is null:

        return

    postOrder(t.left)

    postOrder(t.right)

    print(t.data)

In Order

inOrder(binaryTree t):

    if t is null:

        return

    inOrder(t.left)

    print(t.data)

    inOrder(t.right)

1. *درخت انبوه Heap Tree*
2. *درخت دودویی*
3. *درخت متوازن*
4. *برگ هایی عمقشان بیشتر است در سمت چپ برگ های با عمق کمتر ظاهر خواهند شد. (برگها از سمت چپ پر می شود)*
5. *ارزش هر گره از فرزندان آن کمتر (min Heap) یا بیشتر (max Heap) است.*

*Min-Heap*

*اگر نود های درخت باینری را شماره گذاری کنیم، فرزند نود k ام به ترتیب نود ام و ام است و پدرش نود ام است.*

1. *Insert*

*Order:*

1. *Delete*

*Order:*

*کاربرد اول:*

*Heap Sort*

*Order:*

1. *یک min heap تهی می سازیم*
2. *باید n بار insert heap کنیم*
3. *باید n بار delete min یا delete root کنیم*

*کاربرد دوم: صف الویت*

*Priority queue*

1. *Binary Search Tree*

*فرزند چپ کوچک تر مساوی ریشه است و فرزند راست بزرگتر مساوی ریشه است.*

*BST Search*

BSTSearch(BST t, target):

*if* is null:

*return* null

*if* target == t.data:

*return* t

*if* target < t.data:

*return* BSTSearch(t.left, target)

*return* BSTSearch(t.right, target)

*نکته: روش BST Search سریعترین جستجوی ممکن است.*

*تعداد BST های ممکن با n گره:*

*نکته: inorder درخت BST برابر آرایه مرتب (sort) شده آن است.*

*BST-Min*

BST-Min(BST t):

*if* t is null:

*return* null

*if* t.left is null;

*return* t

    BST-Min(t.left)

BST-Min(Tree t):

*if* t is null:

*return* null

    tmp = t

*while* tmp.left is not null:

        tmp = tmp.left

*return* tmp

*BST-Successor*

BST-Successor(BST t):

*if* t is null:

*return* null

*if* t.right is not null:

*return* BST-Min(t.right)

        # BST-Min has been defined in *class*

*else*:

*if* t.parent is null:

*return* null

*else*:

*if* t.parent.left is equal t:

*return* t.parent

*else* *if* t.parent.right is equal t:

                t = t.parent

                t.right = null

*return* BST-Successor(t)

Insert BST

insertBST(value, BST t, BST parent):

*if* t is null:

        tmp = BST(value)

*if* parent is not null:

*if* value <= parent.data:

                parent.left = tmp

*else*:

             parent.right = tmp

*return*

*if* value <= t.data:

        insertBST(value, t.left, t)

*else*:

     insertBST(value, t.right, t)

*BST-Delete*

BST-Delete(BST t, int value):

*if* t is null:

*return* null

    target = BST-Search(t, value)

    # BST-Search & BST-Min has been defined in *class*

    tmp = BST-Min(target.right)

    target.data = tmp.data

*if* tmp.right is null:

        tmp = null

*else*:

        tmp = tmp.right

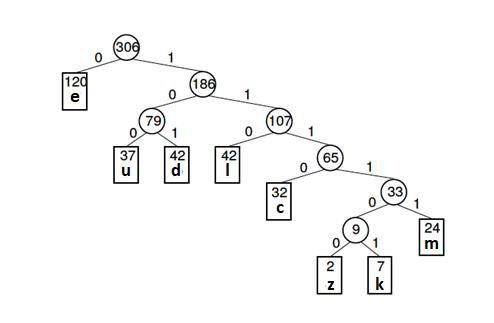
*return* t

*نکته: بعد از delete های متعدد در BST توازن بهم می خورد برای تشکیل یک درخت متوازن، inorder درخت را یافته و از روی آن یک BST متوازن جدید می سازیم.*

1. Huffman Tree

| **Letter** | **Freq** | **Code** | **Bits** |
| --- | --- | --- | --- |
| e | 120 | 0 | 1 |
| d | 42 | 101 | 3 |
| l | 42 | 110 | 3 |
| u | 37 | 100 | 3 |
| c | 32 | 1110 | 4 |
| m | 24 | 11111 | 5 |
| k | 7 | 111101 | 6 |
| z | 2 | 111100 | 6 |

The Huffman tree (for the above example) is given below -



1. *Expression Tree*
2. *درخت دودویی*
3. *درختی است که برگ ها عملوند و غیر برگ ها(گره های داخلی) عملگر*
4. *الویت عملگر ها از ریشه به سمت برگ ها افزایش پیدا می کند(ریشه کمترین الویت را دارد)!*

به inorder درخت عبارت، infix گفته می شود که همان عبارتی است که در حالت پرانتری معادل غیر پرانتزی با الویت باشد.

به نتیجه preOrder درخت prefix می گوییم.

به نتیجه postOrder درخت postfix می گوییم.

*Calculate Expression*

calculateExpTree(Tree t):

*if* isOprand(t.data):

*return* t.data

  case t.data:

    '+':

*return* calculateExpTree(t.left) + calculateExpTree(t.right)

    '-':

*return* calculateExpTree(t.left) + calculateExpTree(t.right)

    '-1':

*return* - calculateExpTree(t.right)

Infix to Postfix

inFixToPostFix(inFixEXP):

s1, s2 = stack()

*for* any token *in* inFixEXP:

*if* isOprand(x):

        s1.push(x)

*else*:

        y = s2.top

*if* y is null:

            s2.push(x)

*elif* priority(x) > priority(y):

            s2.push(x)

*else*:

*if* y is binary:

                operator = s2.pop()

                opr2 = s1.pop()

                opr1 = s1.pop()

                s1.push("opr1 opr2 operator")

*else*:

                operator = s2.pop()

                opr1 = s1.pop()

                s1.push("opr1 operator")

s2.push(x)

*return* s1.pop()

Postfix to Infix

postFixToInFix(postFixEXP):

    Stack s

*for* any token x *in* postFixEXP:

*if* isOprand(x):

            s.push(x)

*else* *if* isSingleOperator(x):

            y = s.pop()

            k = x  y

            s.push(k)

*else*:

            z = s.pop()

            y = s.pop()

            k = y x z

            s.push(k)

*return* s.pop()

Postfix to Prefix

postFixToPreFix(postFixEXP):

    Stack s

*for* any token x *in* postFixEXP:

*if* isOprand(x):

            s.push(x)

*else* *if* isSingleOperator(x):

            y = s.pop()

            k = x  y

            s.push(k)

*else*:

            z = s.pop()

            y = s.pop()

            k = x y z

            s.push(k)

*return* s.pop()

Prefix to Infix

preFixToInFix(preFixEXP):

    Stack s

    Queue q

*for* any token x *in* preFixEXP:

*if* isOprand(x) or isSingleOperator(x):

            y = s.pop()

*if* y is null:

                s.push(x)

*else*:

                z = q.Dequeue()

*if* z is null:

                    k = y x

*else* *if* isSingleOperator(y):

                    k = y x z

*else*:

                    k = y z x

                s.push(k)

*else* *if* isBinaryOperator(x):

            q.Enqueue(x)

    retrun s.pop()

Prefix to EXP Tree

index = 0

Tree preToTree(preFix):

    tmp = Tree()

*if* isOprand(preFix.getToken(index)):

        tmp.data = preFix.getToken(index)

        index++

*return* tmp

*if* isBinaryOperator(preFix.getToken(index)):

        tmp.data = preFix.getToken(index)

        index++

        tmp.left = preToTree(preFix)

        tmp.right = preToTree(preFix)

        retrun tmp

    tmp.data = preFix.getToken(index)

    index++

    tmp.right = preFixToTree(prefix)

*return* tmp

*Sort Orders*

1. *Merge Sort*
2. *Insertion Sort*
3. *Heap Sort*
4. *Bubble Sort*
5. *Selection Sort*
6. *Quick Sort*

*الگوریتم ها دارای سه تقسیم بندی هستند:*

1. *داخلی (internal) یا خارجی (external) بودن*
2. *متعادل (stable) یا نامتعادل (unstable) بودن:*

*الگوریتم متعادل الگوریتمی است که ترتیب نسبی عناصری که مقدار کلیدی شان یکسان است را حفظ می کند.*

1. *مبتی بر مقایسه و غیر مبتنی بر مقایسه بودن*

*Radix Sort*

*Count Sort*

*Bucket Sort*

1. *Count Sort*

Internal

Stable

Independent of comparison

*n تا عدد در بازه 0 تا m-1 می دهد. بر اساس تعداد تکرار اعداد را از اخر sort می کنیم.*

countSort(A, n, m):

    C = [0]\*m

    B = []\*n

*for* i *in* range(n):

        C[A[i]]++

*for* i *in* range(1,m):

        C[i] = C[i] + c[i-1]

*for* i *in* range(n-1, 0, -1):

        B[C[A[i]]-1] = A[i]

        C[A[i]]--

*return* B

1. *Radix Sort*

Stable

Independent of Comparison

*از رقم کم ارزش تا پر ارزش ترین count Sort می زنیم*

*با فرض اینکه k رقم داشته باشیم*

1. *Bucket Sort*

*Stable*

*Independent of Comparison*

**

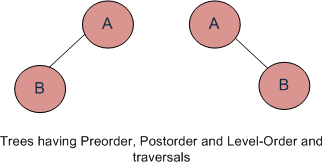
*نکته: merge sort را می توان stable طراحی کرد.*

*سوال: آیا با داشتن level-Order و Post-Order می توان یک درخت یکتا درست کرد؟*

*آیا با داشتن level-Order و In-Order می توان یک درخت یکتا درست کرد؟*

*هزینه زمانی آن چقدر است؟*

It depends on what traversals are given. If one of the traversal methods is Inorder then the tree can be constructed, otherwise not.

  
**Therefore, following combination can uniquely identify a tree.**

Inorder and Preorder.  
Inorder and Postorder.  
Inorder and Level-order.

**And following do not.**  
Postorder and Preorder.  
Preorder and Level-order.  
Postorder and Level-order.

For example, Preorder, Level-order and Postorder traversals are same for the trees given in above diagram.

Preorder Traversal = AB  
Postorder Traversal = BA  
Level-Order Traversal = AB

So, even if three of them (Pre, Post and Level) are given, the tree can not be constructed.

1. Quick Sort

Stable

Dependent of Comparison

partition(A, begin, end):

    x = A[b]

    i = begin

    j = end

*while* i < j:

*while* A[i] <= x and i < j:

            i++

*while* A[j] >= x and i < j:

            j--

*if* i < j:

            swap(A[i], A[j])

    swap(A[b], A[i-1])

*return* i-1

quickSort(A, begin, end):

*if* begin >= end:

*return*

    r = partition(A, b, e)

    quickSort(A, b, r - 1)

    quickSort(A, r + 1, end)

برای پیدا کردن min یک آرایه با n عنصر حداقل و حداکثر چند مقایسه بطور دقیق انجام می دهیم؟

برای محاسبه همزمان min و max اردر زمانی چقدر است؟

i-Select

i امین عنصر از نظر بزرگی یا کوچکی

iSelect(A, begin, end, i):

*if* begin = end:

*return* A[p]

    r = partition(A,b,e)

*if* r == i:

*return* A[r]

*if* r < i:

*return* iSelect(A, r + 1, end, i - r)

*if* r > i:

*return* iSelect(A, b, r - 1, i)

median: میانه

order:

نکته: برای یافتن میانه را محاسبه میکنیم.

1. External merge Sort

برای n های بزرگ که داخل رم جا نمی شود.

تعداد دسترسی به حافظه مهم است.

Order:

اگر k تا sort انجام شود برابر

1. Multi way External merge Sort

شکستن و تبدیل به m گروه

Order:

**Hashing**

1. Directed Hashing

داده i ام در خانه i ام آرایه ذخیره می شود.

1. Chaining Hashing

با استفاده از آرایه ای از linked list ها و روش mod گیری

به عنوان مثال اگر مود m بگیریم، آنگاه طول آرایه m-1 خواهد شد.

Order:

Simple Uniform Hashing

درهم سازی یکنواخت ساده

بهتر است m توانی از 2 نباشد.

بهتر است m یک عدد اول باشد که نزدیک به توانی از 2 نباشد.

روش ضرب:

*درهم سازی کلی Universal*

*نوع درهم سازی مشخص نبوده و قابل سوء استفاده نیست.*

*آدرس دهی باز Open Addressing*

*وارسی خطی linear hashing*

*وارسی درجه دو*

*درهم سازی دوگانه Double Hashing*

*ضرب دو عدد به روش تقسیم و حل divide and conquer*

X = Xl\*2n/2 + Xr

[Xl and Xr contain leftmost and rightmost n/2 bits of X]

Y = Yl\*2n/2 + Yr

[Yl and Yr contain leftmost and rightmost n/2 bits of Y]

XY = 2n XlYl + 2n/2 \* [(Xl + Xr) (Yl + Yr) - XlYl – XrYr] + XrYr

روش هورنر

تعداد نواحی حاصل از تقاطع n خط:

خط n ام با n-1 خط برخورد دارد، n ناحیه جدید:

تعداد نواحی حاصل از تقاطع n زاویه:

**سوال:** فرض کنید x1 تا xn اعداد صحیح مثبت باشند و الگوریتمی با ارائه دهید که این مجموعه از اعداد را به دو مجموعه با مجموع یکسان تبدیل کند.

*def* findSplitPoint(arr, n) :

    leftSum = 0

*for* i *in* range(0, n) :

        leftSum += arr[i]

    rightSum = 0

*for* i *in* range(n-1, -1, -1) :

        rightSum += arr[i]

        leftSum -= arr[i]

*if* (rightSum == leftSum) :

*return* i

*return* -1

*def* printTwoParts(arr, n) :

    splitPoint = findSplitPoint(arr, n)

*if* (splitPoint == -1 or splitPoint == n ) :

        print ("Not Possible")

*return*

*for* i *in* range (0, n) :

*if*(splitPoint == i) :

            print ("")

        print (arr[i], end = " ")

arr = [1, 2, 3, 4, 5, 5]

n = len(arr)

printTwoParts(arr, n)

شبکه مرتب ساز sorting gate

* Concurrent programing

**Dynamic Programing (DP)**

ممکن است همه جا قابل اعمال نباشد، اما اگر کار کند، مرتبه نمایی را به مرتبه زمانی چند جمله ای تبدیل می کند.

مختص مسائل بهینه سازی

استفاده از زیر مسئله های بهینه

حل زیر مسئله ها از پایین به بالا (روش memorization از بالا به پایین عمل می کند.)

ذخیره نتایج زیر مسئله در حافظه

حل مسئله اصلی از روی نتایج زیر مسئله ها

Combination

dynamicComb(n, m):

*for* i *in* range(0, n):

        c[i][0] = 1

*for* i *in* range(0, m):

        c[0][i] = 1

*for* i *in* range(1, m):

*for* j *in* range(2, max((i=m), n)):

            c[j][i] = c[j-1][i] + c[j-1][i-1]

*return* c[n][m]

*Matrix chain*

*تعداد ضرب ها در ضرب ماتریس ها*

*عدد کاتالان*

matrixMult(i, j): *# without dynamic programming*

*if* i >= j:

*return* 0

    minMult = +infinity

*for* k *in* range(i+1, j):

        mult = matrixMult(i, k-1) + matrixMult(k, j) +

        firstDimension(M(i))

\*firstDimension(M(k))

\*secondDimension(M(j))

*if* minMult > mult:

            minMult = mult

*return* minMult

print(matrixMult(1, n))

matrixMinMult(n): *# with dynamic programming*

    mult = [][0]\*(n+1)[]\*2

*for* len in range(1, n-1):

*for* i *in* range(1, n-len):

            j = i + len

            minMult = +infinity

*for* k *in* range(i+1, j):

                mult = mult[i][k-1][0] + mult[k][j][0] +

                        firstDimension(M(i))\*firstDimension(M(k))

\*secondDimension(M(j))

*if* minMult > mult:

                    minMult = mult

            mult[i][j][0] = minMult

*return* mult[1][n][0]

**Memoization**

ممکن است همه جا قابل اعمال نباشد، اما اگر کار کند، مرتبه نمایی را به مرتبه زمانی چند جمله ای تبدیل می کند.

مختص مسائل بهینه سازی

استفاده از زیر مسئله های بهینه

روش memoization از بالا به پایین عمل می کند

ذخیره نتایج زیر مسئله در حافظه

*Matrix Chain*

*# with memoization*

mult = [[]\*(n+1)]\*(n+1)

*for* i *in* in range(n):

*for* j *in* range(n):

*if* i == j:

            mult[i][j] = 0

*else*:

            mult[i][j] = +infinity

matrixMinMult(i, j):

*if* i >= j:

*return* 0

*if* mult[i][j] < +infinity:

*return* mult[i][j]

    minMult = +infinity

*for* k *in* range(i+1, j):

        mult = matrixMinMult(i, k-1) + matrixMinMult(k, j) +

                firstDimension(M(i))\*firstDimension(M(k))

\*secondDimension(M(j))

*if* minMult > mult:

            minMult = mult

    mult[i][j] = minMult

*return* minMult

print(matrixMult(1, n))

***گراف ها Graph***

1. *Directed جهت دار*

*Max*

1. *Indirect بی جهت*

*Max*

1. *Sparse Graph خلوت*
2. *Dense Graph پُر*

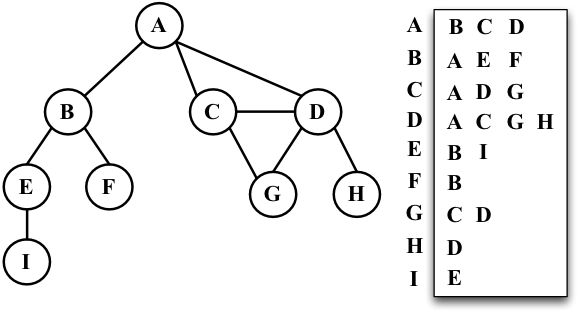
***روش های نمایش گراف:***

1. *ماتریس مجاورت مناسب برای Dense*

*مقدار حافظه لازم*

*-برای گراف وزن دار:*

1. *لیست مجاورت مناسب برای Sparse*

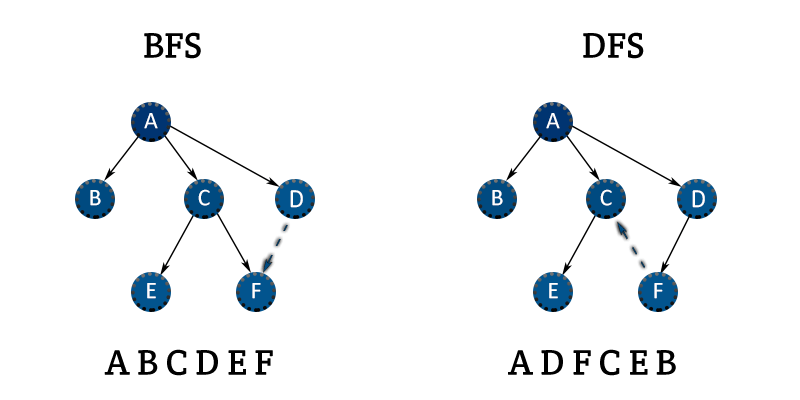
**

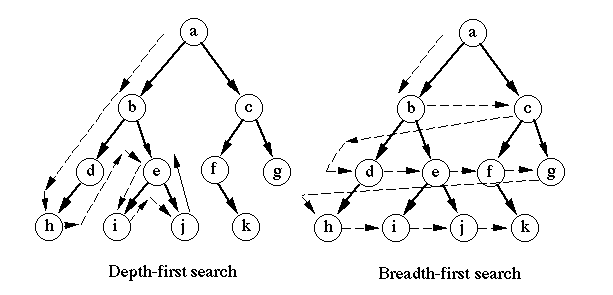
*مقدار حافظه لازم:*

*نکته: لیست مجاورت برای گراف Dense مناسب نیست چون هزینه حافظه یکسانی با ماتریس مجاورت دارد ولی سرعت کمتری بدلیل سرچ دارد.*

*نکته: لیست مجاورت برای گراف sparse مناسب تر است چون با سرعت یکسان حافظه کمتری استفاده می کند .*

*انواع جستجو search در گراف ها*

**

**

1. *BFS*

*Breath first search جستجوی-سطح اول*

BFS(Graph g):

    q = Queue()

    q.empty()

*for* v *in* g:

        visited[v] = False

    q.enqueue(source(g)) *# source is the start point of search*

    visited[source(g)] = True

*while* not q.isEmpty():

*# finish time*

        u = q.dequeue()

        visited[u] = True

*for* v *in* adjacent(u):

*if* not visited[v]:

*# discovery time*

                q.enqueue(v)

                father[v] = u

                visited[v] = True

*Order:*

1. *DFS*

*Depth first search جستجوی-عمق اول*

DFS(Graph g):

    ss = Stack()

    ss.empty()

    for v in g:

        visited[v] = False

    ss.push(source(g)) # source is the start point of search

    visited[source(g)] = True

    while not ss.isEmpty():

        # finish time

        u = ss.pop()

        visited[u] = True

        for v in adjacent(u):

            if not visited[v]:

                # discovery time

                ss.push(v)

                father[v] = u

                visited[v] = True

*Order:*

*Recursive function for DFS*

DFS(Graph g):

    visited[s] = True

*# discovery time(s)*

*for* v *in* adjacent(s):

*if* not visited[v]:

            father[v] = s

            DFS(v)

*# finish time(s)*

*Print Path (for both search methods)*

path(node):

*if* node == s:

        print(s)

*return*

    path(father(node))

    print(node)

*نکته: تعداد گره های قابل دسترسی توسط یک گره خاص:*

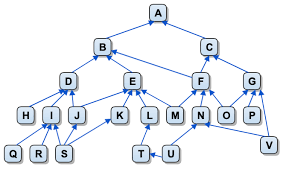
: finish time

: discovery time

*نکته: order های زمانی DFS و BFS باهم برابر است. ولی DFS اردر حافظه کمتری دارد.*

*DAG )Directed acyclic graph(*

*گراف جهت دار بدون دور(مثل دروس پیش نیاز و هم نیاز)*



*Topological ordering*

*ترتیبی از بیان رئوس DAG که ترتیب پیش نیاز در آن حفظ شود.*

*برای این کار روی finish time هر راس sort نزولی میزنیم.*

*Shortest path کوتاه ترین مسیر*

*1.single source shortest path*

*from A to destination (shortest path)*

*2.all pairs shortest path*

*find distance of all pairs of sources*

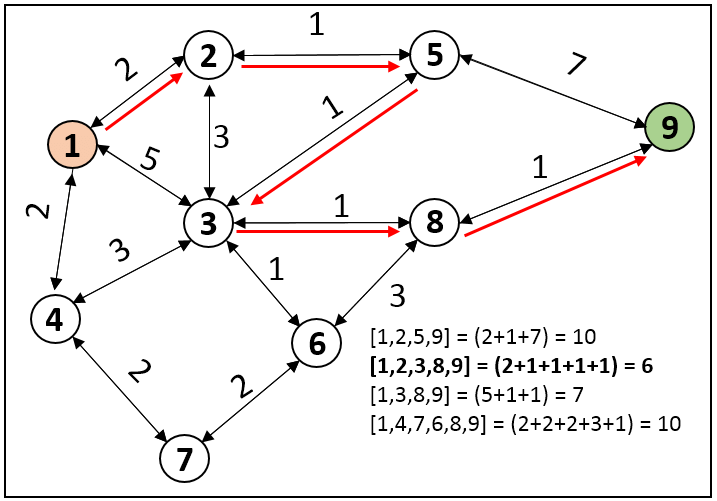
*در این الگوریتم ها می توان یال منفی داشت ولی دور منفی نمیتوان داشت.(چون در حلقه بی انتها می افتیم)*

*در گراف بدون وزن*

*Dijkstra Algorithm: این الگوریتم یال منفی نیز نمی تواند داشته باشد*

*for single source shortest path*

*Order:*

**

Dijkstra(Graph s):

    solved = [s]

*while* solved != v: *# O(V)*

        (i,j) = min{w(u,v)| u in solved and v not in solved} *# O(v^2)*

        d[j] = w[i, j]

        solved = solved union {j}

*for* any k | (j,k) *in* E and k not in solved: *# O(E)*

            w[j,k] = w[i,j] + w[j,k]

*return* d

***یافتن کوتاه ترین مسیر از* i *به* j *با حداکثر* k *یال***

* *Floyd warshall* k*با گذشتن از راس های 1 تا*

*Red-Black Tree*

1. *هر گره یا قرمز است یا سیاه.*
2. *هر برگ null و سیاه است.*
3. *دو فرزند یک گره قرمز سیاه هستند.*
4. *هر مسیر ساده از گره به فرزند(نه لزوما فرزند مستقیم) شامل تعداد یکسانی گره سیاه است.*
5. *ریشه درخت سیاه است (این شرط اصلی نیست)*
6. *حتما خاصیت BST را داراست.*

حداکثر ارتفاع یک درخت قرمز سیاه که دارای n گره داخلی است برابر

زیر درخت با ریشه دلخواه x حداقل دارای گره داخلی است.