


دانشکده‌ی علوم ریاضی

داده‌ساختارها و الگوریتم‌ها ۱۳ بهمن ۱۳۹۶

گزارش پروژه دوم

مدرّس: دکتر شهرام خزائی

نگارنده: علی عمادی – ۹۵۱۰۰۱۹۱

۱ مزرعه بی‌خاصیت

۱.۱ توضیح الگوریتم

ابتدا m ماشین را با توجه به مکان اولیه آنها توسط $quicksort$ مرتب می‌کنیم و آنها را در یک $linkedlist$ دوطرفه نگه‌داری می‌کنیم.اطلاعاتی که برای یک ماشین در $linkedlist$ نگه‌داری می‌شوند :

۱. شماره ماشین

۲. مکان

۳. سرعت

۴. زمان برخورد

۵. شماره ماشینی که با آن برخورد می‌کند.

سپس زمان برخورد هر ماشین با ماشین بعدی خود را (فقط در صورتی که زمان مثبت باشد و کمتر از مثبت بی‌نهایت) در یک درخت دودویی جست‌وجو می‌ریزیم.برای ماشین اول زمان خارج شدن از سمت چپ جاده و برای ماشین آخر زمان خارج شدن از سمت راست جاده را در نظر می‌گیریم.ویژگی هایی که هر گره درخت در خود ذخیره می‌کند عبارتند از:

۱. فرزند چپ

۲. فرزند راست

۳. ماشین اول که باعث این برخورد در زمان ذکر شده می‌شود(پوینتر به راس موردنظر در $linkedlist$)

۴. ماشین دوم که یاعث این برخورد در زمان ذکر شده می‌شود(پوینتر به راس موردنظر در $linkedlist$)

۵. زمان برخورد

داده ها در درخت براساس زمان مرتب شده‌اند،بنابراین برای دسترسی به داده مینیمم به زمان $O(m)$ نیاز داریم.داده مینیمم از درخت را پیدا می‌کنیم و زمان مورد نظر را به ماشین های اول و دوم نسبت می‌دهیم.بعد شماره ماشین مقابل را برای هر ماشین قرار می‌دهیم.یعنی شماره ماشین دوم را برای ماشین اول و شماره ماشین اول را برای ماشین دوم قرار می‌دهیم.سپس داده مینیمم از درخت و ماشین اول و دوم را از $linkedlist$ حذف می‌کنیم.با استفاده از $linkedlist$ زمان برخورد ماشین قبل ماشین اول در صورت وجود و ماشین بعدی ماشین دوم را محاسبه و به درخت اضافه می‌کنیم.سپس همین روند را تا خالی شدن درخت ادامه می‌دهیم.
بنابراین برای هر ماشین زمان برخورد و شماره ماشینی که با آن برخورد می‌کند را خواهیم داشت.این داده ها را چاپ می‌کنیم.

جدول ۱: زمان برحسب طول ورودی سوال ۱

m	time
10	0.006
100	0.032
500	0.057
1000	0.094
5000	0.168
10000	0.227
50000	0.812
100000	1.312

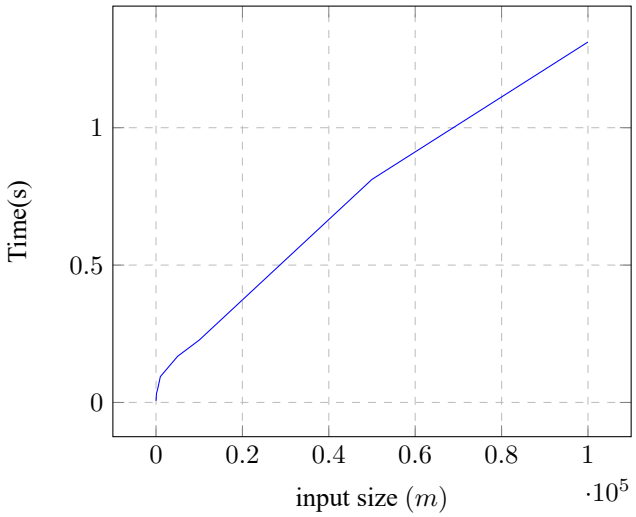
۲.۱ شبه‌کد

فرض کنید L ، لیست توضیح داده شده باشد.

```
function Q1(L, cars[0...m − 1])
    while !tree.isEmpty() do
        node = tree.minumum()
        first = node.firstCar
        second = node.secondCar
        cars[first.ip].time = node.time
        cars[first.ip].VSip = second.ip
        cars[second.ip].time = node.time
        cars[second.ip].VSip = first.ip
        L.delete(first)
        L.delete(second)
        first = first.prev
        second = second.next
        v_i = first.speed
        x_i = first.X
        v_j = second.speed
        x_j = second.X
        time = (x_j − x_i)/(v_i − v_j)
        if time > 0 and time !=INFINITY then
            tree.insert(time, first, second)
```

۳.۱ کارآرایی کد

از آنجایی که زمان این الگوریتم به مقدار n وابسته نیست بنابراین فقط تست‌هایی با m های مختلف تولید شده است.



۲ به یاد عطا

۱.۲ توضیح الگوریتم

برای این سوال فقط کافی است درخت قرمز سیاه متمایل به چپ را پیاده‌سازی می‌کنیم. تمام توابعی که نامشان در توضیح ذکر شده،در قسمت شبه‌کد،شبه‌کد آنها آمده است.

درج:

به صورت بازگشتی پیاده‌سازی شده‌است.مانند درخت دودویی تا جایی جلو می‌رویم که به یک راس خالی برسیم،وقتی به راس خالی رسیدیم یعنی جای درست راس پیدا شده،راس را اضافه می‌کنیم و به طور بازگشتی از بالا به پایین ویژگی های درخت قرمز سیاه متمایل به چپ را چک می‌کنیم و اگر جایی این ویژگی‌ها رعایت نشده بود با اعمال مناسب درخت را قرمز سیاه متمایل به چپ می‌کنیم.

ویژگی هایی که به طور بازگشتی چک می‌شوند:

۱. وقتی فرزند سمت راست راسی که روی آن هستیم قرمز و راس چپ آن سیاه است: از تابع $rotateLeft$ استفاده می‌کنیم.

۲. وقتی فرزند سمت چپ و فرزند چپ فرزند چپ راسی که روی آن هستیم قرمزند: از تابع $rotateRight$ استفاده می‌کنیم.

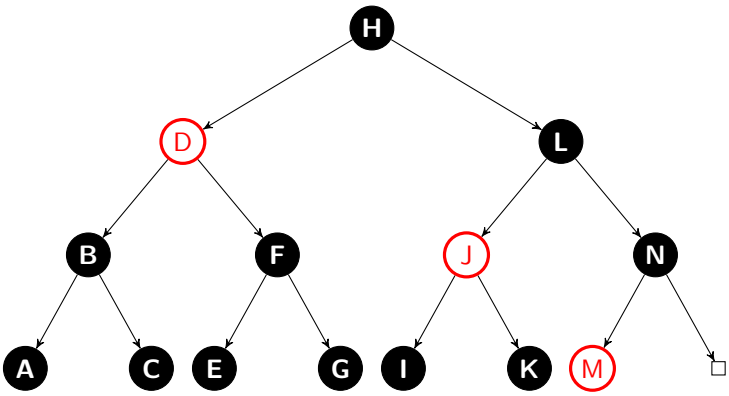
۳. وقتی فرزند راست و چپ راسی که روی آن هستند قرمزند: از تابع $flipColors$ استفاده می‌کنیم.

حذف:

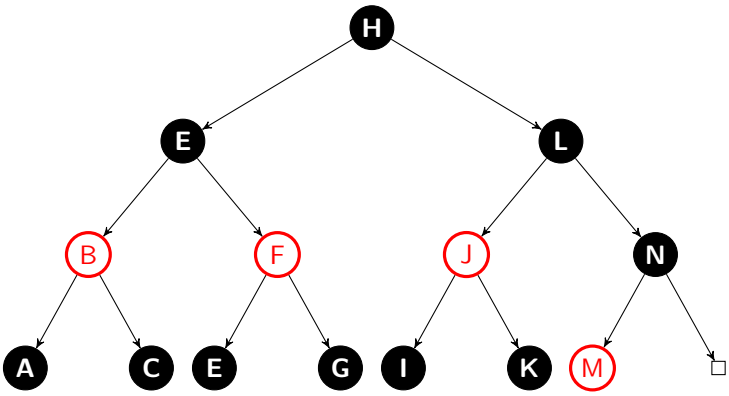
برای حذف هم مانند درج عمل می‌کنیم.اگر راسی که می‌خواهیم حذف کنیم یک برگ قرمز باشد مشکلی برای حذف آن نداریم بنابراین همیشه می‌خواهیم راسی که باید حذف شود را تبدیل به برگ قرمز کنیم تا حذف راحت انجام شود.در حین تبدیل آن راس به برگ قرمز ممکن است ویژگی‌هایی از درخت قرمز سیاه متمایل به چپ از بین برود اما به طور بازگشتی این ویژگی‌ها را دوباره در درخت پیاده‌سازی می‌کنیم.

برای تبدیل راس مورد نظر به برگ قرمز از یک ناوردایی استفاده می‌کنیم این ناوردایی آن است که خود راسی که روی آن هستیم و یا یکی از فرزندانش قرمز است(در واقع خودمان این ناوردایی ها را ایجاد می‌کنیم).تمام مراحل حذف هم به‌صورت بازگشتی پیاده‌سازی شده‌است.

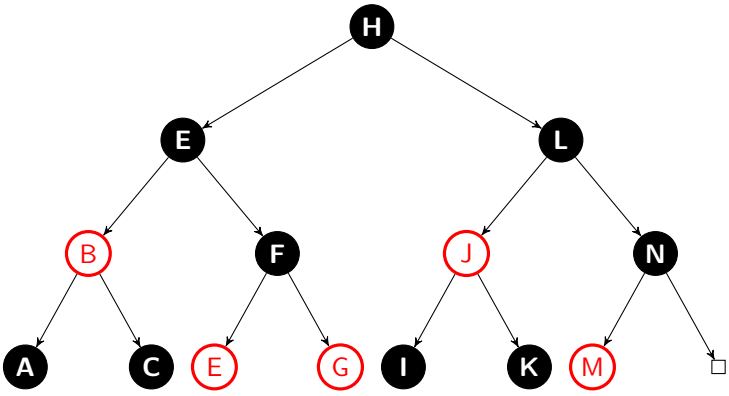
به عنوان مثال فرض کنید راس D را می‌خواهیم از درخت حذف کنیم.



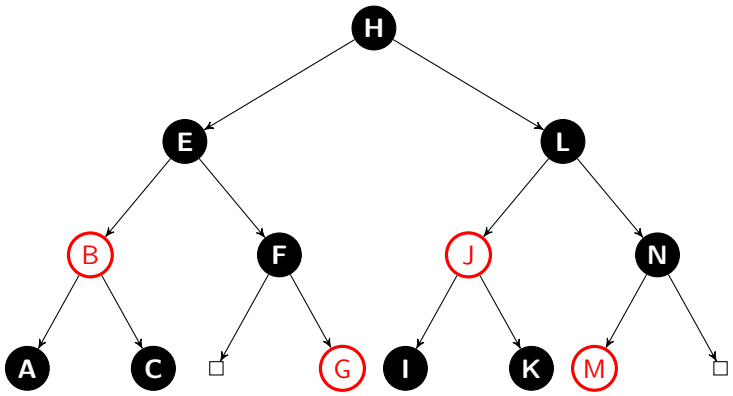
ابتدا از ریشه حرکت می‌کنیم.چون فرزند چپ ریشه قرمز است بنابراین ناوردایی برقرار است و نیازی به تغییر در درخت نیست.چون D کوچکتر از H است پس باید به سمت چپ بیاییم.به D می‌رسیم.می‌بینیم که نه Dو نه هیچ‌کدام از فرزندانش قرمز نیستند،همچنین فرزند چپ فرزند راست D هم سیاه است بنابراین با استفاده از تابع $moveRedRight$ ناوردایی را ایجاد می‌کنیم.(در اینجا چون فرزند چپ فرزند چپ D قرمز نیست $moveRedRight$ مانند $flipColors$ عمل می‌کند.چون D فرزند راست دارد کافی است جای کوچکترین مقدار زیردرخت با ریشه D را با D عوض می‌کنیم و ادامه می‌دهیم.درخت به درخت زیر تبدیل می‌شود:



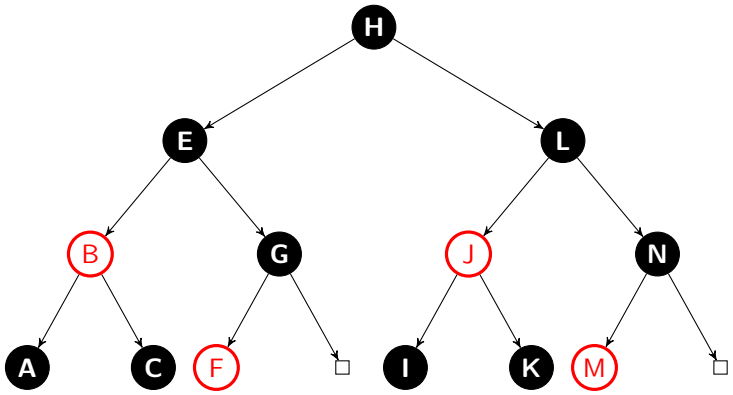
حال کافی است که E(برگ) را از درخت حذف کنیم.برای این کار از تابع $deleteMin$ که روی D صدازده شده استفاده می‌کنیم.با حرکت به سمت راست به راس F می‌رسیم.چون نه فرزند چپ و نه فرزند چپ فرزند چپ F قرمز نیستند،تابع $moveRedLeft$ روی F صدازده می‌شود.درخت به درخت زیر تبدیل می‌شود:



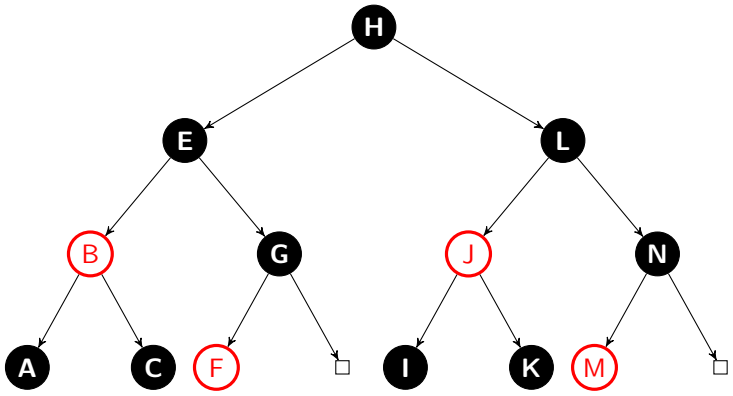
به چپ می‌رویم و به راسی می‌رسیم که می‌خواهیم پاک کنیم.از آنجایی که برگ قرمز است به راحتی آن را پاک می‌کنیم.حال به طور بازگشتی درخت را تبدیل به درخت قرمز سیاه متمایل به چپ می‌کنیم.



دوباره به راس F برمی‌گردیم،از آنجایی که فرزند راست آن قرمز است از تابع $rotateLeft$ استفاده می کنیم و به پدر F یعنی E می‌رویم.



E و فرزندانش هیچ‌کدام از ویژگی های درخت قرمز سیاه متمایل به چپ را نقض نکرده‌اند بنابراین به پدر E یعنی ریشه درخت می‌رویم.این راس و فرزندانش هم تمام ویژگی‌های درخت قرمز سیاه متمایل به چپ را حفظ کرده‌اند،بنابراین حذف تمام شده است و به درخت زیر پس از حذف راس D از درخت اولیه رسیدیم.



۲.۲ شبه‌کد

Algorithm 1 FLIP THE COLORS OF A NODE AND ITS TWO CHILDREN

function FLIPCOLORS(h)
$h.color = !h.color$
$h.left.color = !h.left.color$
$h.right.color = !h.right.color$
$root.color = BLACK$

Algorithm 2 MAKE A LEFT-LEANING LINK LEAN TO THE RIGHT

function ROTATERIGHT(h)
$x = h.left$
$h.left = x.right$
$x.right = h$
$x.color = x.right.color$
$x.right.color = RED$
return x

Algorithm 3 MAKE A RIGHT-LEANING LINK LEAN TO THE LEFT

function ROTATELEFT(h)
$x = h.right$
$h.right = x.left$
$x.left = h$
$x.color = x.left.color$
$x.left.color = RED$
return x

Algorithm 4 ASSUMING THAT H IS RED AND BOTH H.RIGHT AND H.RIGHT.LEFT ARE BLACK, MAKE H.RIGHT OR ONE OF ITS CHILDREN RED

function MOVEREDRIGHT(h)
$flipColors(h)$
if $isRed(h.left.left)$ **then**
$h = rotateRight(h)$
$flipColors(h)$
return h

Algorithm 5 ASSUMING THAT H IS RED AND BOTH H.LEFT AND H.LEFT.LEFT ARE BLACK, MAKE H.LEFT OR ONE OF ITS CHILDREN RED

function MOVEREDLEFT(h)
$flipColors(h)$
if $isRed(h.right.left)$ **then**
$h.right = rotateRight(h.right)$
$h = rotateLeft(h)$
$flipColors(h)$
return h

Algorithm 6 RESTORE RED-BLACK TREE INVARIANT

```
function BALANCE(h)
    if isRed(h.right) then
        h = rotateLeft(h)
    if isRed(h.left) and isRed(h.left.left) then
        h = rotateRight(h)
    if isRed(h.left) and isRed(h.right) then
        flipColors(h)
    return h
```

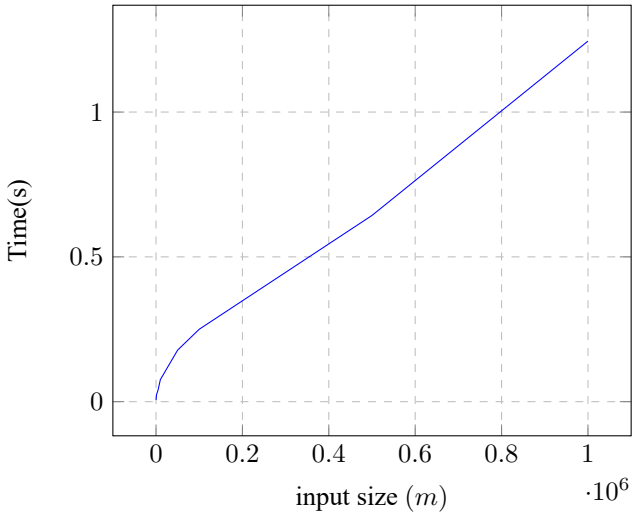
```
function INSERT(h, key)
    if h == null then
        return newNode(key, RED)
    if key < h.key then
        h.left = insert(h.left, key)
    else
        if key > h.key then
            h.right = insert(h.right, key)
    if isRed(h.right) and !isRed(h.left) then
        h = rotateLeft(h)
    if isRed(h.left) and isRed(h.left.left) then
        h = rotateRight(h)
    if isRed(h.left) and isRed(h.right) then
        flipColors(h)
```

```
function DELETE(h, key)
    if key < h.key then
        if !isRed(h.left) and !isRed(h.left.left) then
            h = moveRedLeft(h)
        h.left = delete(h.left, key)
    else
        if isRed(h.left) then
            h = rotateRight(h)
        if key == h.key and h.right == null then
            return null
        if !isRed(h.right) and !isRed(h.right.left) then
            h = moveRedRight(h)
        if key == h.key then
            x = min(h.right)
            h.key = x.key
            h.right = deleteMin(h.right)
        else
            h.right = delete(h.right, key)
    return balance(h)
```

جدول ۲: سوال ۲ – زمان برحسب طول ورودی

m	time
100	0.006
1000	0.023
5000	0.042
10000	0.076
50000	0.178
100000	0.250
500000	0.643
1000000	1.245

۳.۲ کارآرایی کد



۳ دوستان اجتماعی

۱.۳ توضیح الگوریتم

ابتدا m جمله امیرعلی را با استفاده از تابع $hash$ که در قسمت شبه‌کد آمده است،درهم‌سازی می‌کنیم.سپس این m عدد را در یک آرایه می‌ریزیم.n جمله مشاخ را در نظر بگیرید،برای هر جمله هر زیر رشته را با استفاده از تابع درهم‌سازی گفته شده،تبدیل به عدد و همه‌ی این اعداد (شامل اعداد حاصل درهم‌سازی هر زیررشته از n جمله مشاخ) را در آرایه‌ی دیگری می‌ریزیم و این آرایه را مرتب می‌کنیم.حال با استفاده از $binarysearch$ اعداد موجود در آرایه اول را در آرایه دوم جست‌وجو می‌کنیم،اگر مثلاً عدد حاصل درهم‌سازی جمله‌ی iام امیرعلی در آرایه دوم وجود داشت،یعنی اینکه این جمله امیرعلی در اول یکی از جمله‌های مشاخ آمده است.تعداد این اعداد جواب مسئله ماست.

جدول ۳: زمان برحسب طول ورودی سوال ۳

n	m	time
100000	100	0.781
100000	1000	0.792
100000	10000	0.804
100000	100000	0.875
10000	100000	0.151
1000	100000	0.071
100	100000	0.042
10	100000	0.032

۲.۳ شبه‌کد

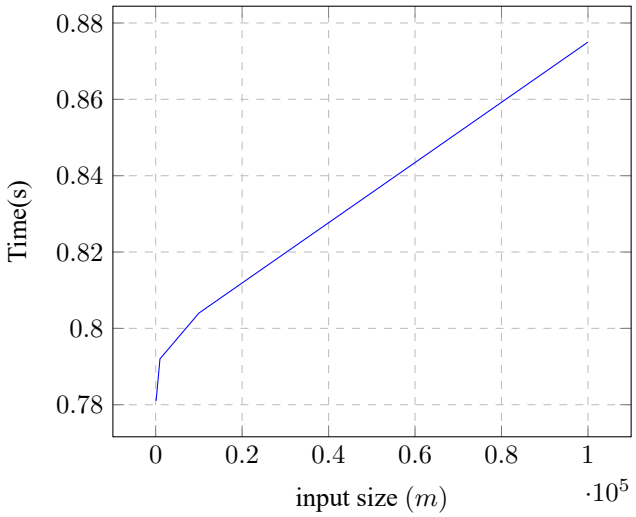
```
function HASH(string)
    hash = 0
    prime = 16777619
    for i = 0 to string.length() do
        hash = hash * prime
        hash = hash XOR string.charAt(i)
    return hash
```

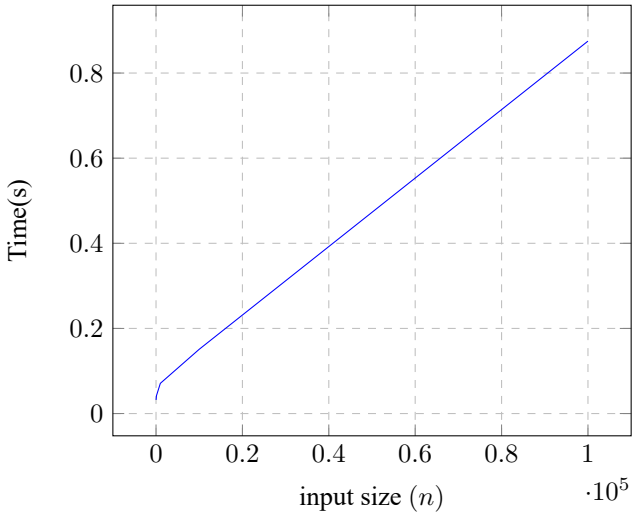
فرض کنید آرایه$perfix$به طول $n * ۶۰$ باشد.

```
function Q3(AmirAli[0...m − 1], Mashakh[0...n − 1])
    result = 0
    for i = 0 to n do
        for j = 0 to Mashakh[i].length() do
            prefix[i * 60 + j] = hash(Mashakh[i].substring(0, j + 1)
    sort(prefix)
    for i = 0 to m do
        hashCode = hash(AmirAli[i])
        if binarySearch(prefix, hashCode) >= 0 then
            result = result + 1
    return result
```

۳.۳ کارآرایی کد

برای این سوال یک بار نمودار زمان برحسب n و یک بار برحسب m کشیده شده است.


0.88
0.86
0.84
0.82
0.8
0.78
Time(s)
0
0.2
0.4
0.6
0.8
1
input size (m)
$\cdot 10^5$


0.8
0.6
0.4
0.2
0
Time(s)
0
0.2
0.4
0.6
0.8
1
input size (n)
$\cdot 10^5$