Data Structures and algorithms
Instructor : **Dr.Amin Eskandari**
Head-Ta's : Hamid Namjoo manesh , Amir Hossein Hemati , Ali Mojahed
Ta's : Amir Mohammad Asadjoo , Padina razmjooi , Armin Janfaza , Alireza Heydari , Mahdi Torabi ,
Maryam Ghorbani , Fatemeh Keshavarz , MohammadReza Fasli , Abolfazl Tadrisi

Quiz 04 Answers

پاسخنامه آزمون شماره ۰۴

پاسخ سوال اول :

برای پاسخ دادن به این سوال ها شما باید یک تاریخچه کوتاه از Multilevel Queue داشته باشید و تقریبا با Deque یا همون Double end queue و priority queue (این یکی زیاد نیست یکم بدونید چی به چیه کافیه) آشنا باشید.

- پاسخ قسمت ۱ :

انتخاب ما Multilevel Queue می‌باشد و اگر از کلمه feedback در این قسمت استفاده شده باشه یا از نام های دیگه نشونه استفاده از Ai می باشد.

- پاسخ قسمت ۲ :

اول از هرچیزی استفاده از کلمه هایی مثل CPU Scheduler نشانه بودن در مسیر درست است ولی اگر نبود فرقی ندارد.

این الگوریتم در واقع تسک ها و پردازش های مارا بر اساس مموری و سرعت اجرا و این‌گونه موارد دسته بندی می کند. توجه کنید که این الگوریتم کارش با Context switching فرق دارد . چرا این دسته بندی انجام می شود؟ به این دلیل که تسک های بزرگ جلو تسک های کوچک را نگیرند. به عنوان مثال یک تسک کوچک مثل کلیک کردن یا باز کردن برنامه یا وصل بودن کیبورد، تسک های حیاطی هستند که اگر به موقع انجام نشوند باعث بهینه نبودن می شوند.

همچنین این الگوریتم از deque درون deque ساخته شده یعنی ما یک صف دو طرفه درون یک صف دو طرفه دیگر داریم که برای هر کدام از deque های داخلی میتوانیم یک سیاست اعمال کنیم و در

نتیجه امروزه میتوانیم 20 بار رفرش کنیم یا برنامه هامون رو باز کنیم ، وقتی که کروم دارد باز می شود نه اینکه منتظر باشیم تا کروم باز شود تا پس از آن کار هارا انجام دهیم.

- پاسخ قسمت ۳ :

Multi-Level Feedback Queue در این قسمت کلمه feedbackمجاز میشود.

- پاسخ قسمت ۴ :

قبل از دونستن بیاین بدونیم که ما دو نوع برنامه داریم یکی Interactive و CPU-bound این الگوریتم همان الگوریتم بالایی است اما با این تفاوت که میتواند بین صف های مختلفمون حرکت کنه و هدفش اینه که سیستم عامل بدون اینکه بدون نوع برنامه چیه اونو دسته بندی کنه برای اجرا

مثلا اگر ما چهار صف داشته بشیم که queue 0 مربوط به پردازش های زیر 5ms باشه اگر برنامه بیشتر از این زمان طول کشید تا پروسس کنه یا به quantum CPUرسید اونو به صف بعدی میفرسته که الویتش کمتر و این الگوریتم چند خصوصیات دارد:

یکی برگردوندن همه تسک ها به اولین صف با بالاترین الویت که تسک های اخر همیشه همون اخر نمونن.

یکی اینکه اگر تسک جدیدی وارد شد اونو در صفی با بالاترین الویت میزاره چون ممکه کوتاه باشه

یکی دیگه هم اینکه بعد از انجام هر تسک اونو خارج میکنه از صف.

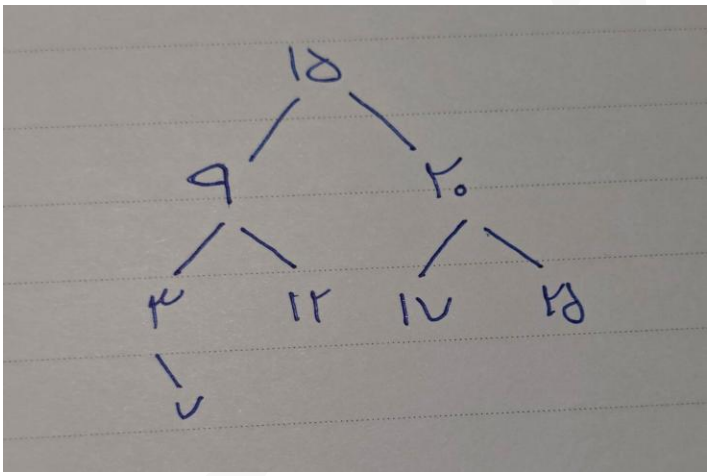
عمیق شدن و جواب کامل به این سوال ها کم کم نیاز به صفحه های زیادی پاسخ و درک زیادی از نحوه کار CPU و سیستم عامل که در همین حد کافیه.

پاسخ سوال دوم :

بخش الف:
ساخت BST با درج ترتیبی [15,9,3,12,7,20,17,25]

مراحل درج:
- ۱۵ریشه می‌شود.
- ۹ < ۱۵ ← فرزند چپ ۱۵.
- ۳ < ۱۵ ← چپ؛ ۳ < ۹ ← فرزند چپ ۹.
- ۱۲ < ۱۵ ← چپ؛ ۱۲ > ۹ ← فرزند راست ۹.
- ۷ < ۱۵ ← چپ؛ ۷ > ۹ ← چپ؛ ۷ > ۳ ← فرزند راست ۳.
- ۲۰ > ۱۵ ← فرزند راست ۱۵.
- ۱۷ < ۱۵ ← راست؛ ۱۷ < ۲۰ ← فرزند چپ ۲۰.
- ۲۵> ۱۵ ← راست؛ ۲۵ > ۲۰ ← فرزند راست ۲۰.

ساختار نهایی :



پیمایش inorder :
[3,7,9,12,15,17,20,25]

LCA دو گره ۷ و ۱۷ :

شروع از ریشه = ۱۵.
- ۷ < ۱۵ و ۱۷ > ۱۵ ← یکی در چپ و دیگری در راست ← LCA = 15.

بخش ب:
تبدیل آرایه به Max-Heap با روش Bubble-Down (Heapify)

آرایه اولیه:
[15, 9, 3, 12, 7, 20, 17, 25]
مرحله ۱: تعیین آخرین گره‌ی غیربرگ
برای آرایه‌ای با طول n = 8:

$$\text{آخرین والد} = \left\lfloor \frac{n-2}{2} \right\rfloor = 3$$

بنابراین عملیات bubble-down از اندیس‌های 3، 2، 1، 0 انجام می‌شود.

مرحله ۲: اجرای Bubble-Down
۱) اندیس 3
مقدار: 12
فرزند چپ: 25
فرزند راست: ندارد
← بزرگ‌ترین فرزند: 25
← جابه‌جایی
آرایه جدید:
[15, 9, 3, 25, 7, 20, 17, 12]

۲) اندیس 2
مقدار: 3
فرزندان: 20 و 17
← بزرگ‌ترین: 20
← جابه‌جایی
آرایه جدید:
[15, 9, 20, 25, 7, 3, 17, 12]
اندیس 5 فرزند ندارد ← توقف

۳) اندیس 1
مقدار: 9
فرزندان: 25 و 7
← بزرگ‌ترین: 25
← جابه‌جایی

آرایه جدید:
[15, 25, 20, 9, 7, 3, 17, 12]
ادامه روی اندیس 3 (مقدار 9):
فرزند چپ: 12
← جابه‌جایی
آرایه جدید:
[15, 25, 20, 12, 7, 3, 17, 9]

۴) اندیس 0
مقدار: 15
فرزندان: 25 و 20
←بزرگ‌ترین: 25
← جابه‌جایی
آرایه جدید:
[25, 15, 20, 12, 7, 3, 17, 9]
اندیس 1 فرزندانی کوچک‌تر از خودش دارد ← توقف

نتیجه نهایی Max-Heap

$$[25، 15، 20، 12، 7، 3، 17، 9]$$

پیچیدگی زمانی ساخت Max-Heap از یک آرایه n عضوی با روش Bubble-Down (Heapify) برابر است با:

$$O(n)$$

در الگوریتم heapify عمل Bubble-Down تنها برای گره‌های غیربرگ انجام می‌شود.
گره‌های سطوح پایین تعداد بیشتری دارند اما ارتفاع کمتری دارند، در حالی که گره‌هایی که bubble-down طولانی دارند تعدادشان بسیار کم است.
بنابراین ساخت Max-Heap از یک آرایه **زمان خطی** دارد، نه $O(n\log\ n)$.

بخش ج:

بازسازی منطق change_order

ریشه = اولین عنصر preorder = 15
موقعیت 15 در اندیسinorder = 4.
- inorder چپ: [3,7,9,12] ← اندازه ۴
- inorder راست [17,20 , 25] ← اندازه ۳
- preorder چپ: [9,3,7,12] (۴ عنصر بعد از ریشه)
- preorder راست: [20,17,25]

زیردرخت چپ: [3,7,9,12] = in, pre= [9,3,7,12]
ریشه = 9، اندیس آن در inorder =2
- چپ: pre= [3,7] و in =[3,7] ← ریشه ۳، راستش ۷ ← postorder این بخش = [3 ,7]
- راست: ← in= [12] , pre= [12] ← postorder = [12]
- postorder کل این زیردرخت = [7, 3, 12, 9]

زیردرخت راست: in=[17,20,25] , pre= [20,17,25],
ریشه = ۲۰ , چپش ۱۷، راستش ۲۵.
- postorder = [17, 25, 20]

postorder نهایی:
ابتدا چپ، سپس راست، سپس ریشه:
[7,3,12,9,17,25,20,15]

پاسخ سوال سوم :

تاکتیک اول : ***لیست پیوندی ساده***

ساختار نهایی : یک لیست پیوندی شامل تمام n زندانی.

تحلیل زمان کل : در دور اول اتحاد، $n/2$ عملیات داریم که هر کدام هزینه $O(1)$دارند (اتصال لیست ۱ عضوی به ۱ عضوی). در دور دوم، $n/4$ عملیات داریم که هر کدام هزینه $O(2)$ دارند (اتصال لیست ۲ عضوی به ۲ عضوی). این الگو ادامه دارد. هزینه کل $Union$ها برابر است با:

$$\sum_{i=1}^{\log n} \frac{n}{2^i} \cdot 2^{i-1} = \sum_{i=1}^{\log n} \frac{n}{2} = O(n \log n)$$

هزینه کل find ها نیز برابر است با :

$$n \times O(1) = O(n)$$

پیچیدگی کل : O(nlogn)

تاکتیک دوم : ***لیست پیوندی با ادغام وزن‌دار***

ساختار نهایی :یک لیست پیوندی شامل تمام nزندانی.

تحلیل زمان کل :در سناریوی ما، همیشه دو گروه با **اندازه یکسان** با هم متحد می‌شوند. بنابراین، راهکار ادغام وزن‌دار هیچ مزیتی ایجاد نمی‌کند و رفتار آن دقیقاً مشابه حالت ساده خواهد بود. هزینه $Union$ ها $O(n\log\ n)$ و هزینه $Find$ ها $O(n)$ است.

پیچیدگی کل : O(nlogn)

تاتیک سوم : ***درخت ساده***

ساختار نهایی :به دلیل "**قانون اتحاد**" (رهبر با شماره کوچکتر به بزرگتر می‌پیوندد)، یک درخت کاملاً متوازن از نظر تعداد نود در هر زیرشاخه ایجاد می‌شود. ارتفاع این درخت دقیقاً $\boldsymbol{k = \log\ n}$ خواهد بود .

زمان کل :هزینه $Union$ ها $O(n)$ است. هزینه $Find$ ها به عمق گره‌ها بستگی دارد. مجموع عمق تمام گره‌ها در یک درخت دودویی کامل $O(n\log\ n)$ است. بنابراین هزینه کل مرحله بازجویی $O(n\log\ n)$ خواهد بود .

پیچیدگی کل : O(nlogn)

تاکتیک چهارم : ***درخت با ادغام بر اساس رتبه و فشرده‌سازی مسیر***

ساختار نهایی :یک درخت بسیار کم‌عمق, ادغام بر اساس رتبه به تنهایی درختی با ارتفاع $O(\log\ n)$می‌سازد (مشابه حالت قبل). اما فشرده‌سازی مسیر در "**مرحله بازجویی نهایی**" همه چیز را تغییر می‌دهد. با اجرای $Find$ برای تمام گره‌ها، تقریباً تمام گره‌ها مستقیماً به ریشه متصل می‌شوند .

زمان کل :این سناریو دقیقاً جایی است که قدرت کامل این دو بهینه‌سازی با هم مشخص می‌شود. هزینه سرشکن شده برای $O(n)$ عملیات $Union$ و $O(n)$ عملیات $Find$ تحلیل می‌شود .

پیچیدگی کل :O(n·α(n)) که به $O(n)$بسیار نزدیک است. قیام با کارآمدترین روش ممکن به سرانجام می‌رسد.

پاسخ سوال چهارم :

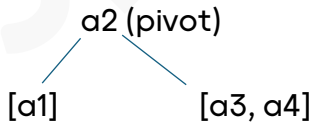
بخش الف. :

1. انتخاب Pivot : Median برای (n = 4)

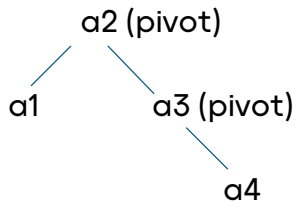
در این حالت ، فرض می کنیم Pivot همواره میانگین (median) عناصر متمایز انتخاب می شود. برای n=4 با فرض عناصر مرتب شده (a_1<a_2<a_3<a_4) ،median برابر با (a_2) یا (a_3) خواهد بود. برای سادگی فرض می کنیم (a_2) به عنوان pivot انتخاب می شود.

ساخت درخت تصمیم برای 4 عنصر :

فرض کنید عناصر ما (a_1, a_2, a_3, a_4) باشند. اگر (a_2) به عنوان pivot انتخاب شود :



- گره ریشه : (a_2) به عنوان pivot انتخاب می شود.
- (a_1) به زیردرخت چپ می رود چون (a_1 < a_2)
- (a_3) و (a_4) به زیر درخت راست می روند چون (a_3 > a_2) و (a_4 > a_2)
- زیردرخت چپ : شامل (a_1) . این زیر مسئله مرتب شده است.
- زیردرخت راست : شامل (a_3, a_4) . برای این زیر مسئله ،median انتخاب می شود. اگر (a_3) median باشد :
  - (a_3) به عنوان pivot انتخاب می شود.
  - (a_4) به زیردرخت راست خود می رود.



انتخاب median به عنوان pivot ، QuickSort را به سمت یک الگوریتم مرتب‌سازی متوازن سوق می دهد. در این حالت ، نه تنها درخت تصمیم ، بلکه تقسیم‌بندی آرایه در هر مرحله نیز به بهترین شکل ممکن انجام می شود و عملا بهترین حالت QuickSort را شبیه‌سازی می‌کند.

- ارتفاع درخت : ارتفاع درخت تصمیم به تعداد لبه ها از ریشه تا عمیق ترین برگ گفته می شود. در این مثال، عمیق ترین برگ (a_4) است.
  - ریشه (a_2) به گره (a_3) . (1 لبه)
  - گره (a_3) به برگ (a_4) . (1 لبه)
  - بنابراین،ارتفاع درخت 2 است.
- تعداد مقایسه ها :
  - در ریشه (a_2) : 3 مقایسه برای یافتن (a_2) (مثلا با (a_1) ، (a_3) ، (a_4)).
  - در زیردرخت راست (a_3) : 1 مقایسه برای مرتب کردن (a_4) (با (a_3)).
  - مجموع مقایسه ها : (4 = 3 + 1) مقایسه.

2. انتخاب Pivot : کوچکترین عنصر (بدترین حالت برای n = 4)

در این حالت ، فرض می کنیم pivot همواره کوچک‌ترین عنصر موجود در زیرآرایه انتخاب می‌شود.

شکل کلی درخت تصمیم(یک زنجیره راست / degenerate) :

فرض کنید عناصر مرتب شده (a_1 < a_2 < a_3 < a_4) باشند.

1. مرحله اول : (a_1) به عنوان pivot انتخاب می‌شود.
   - (a_1) در جای خود قرار می‌گیرد.

- تمام عناصر دیگر ((a_4) , (a_3) , (a_2)) بزرگتر از (a_1) هستند و به زیردرخت راست می‌روند.

a 1 (pivot)

[a2, a3, a4]

2. مرحله دوم : در زیردرخت راست ، کوچک‌ترین عنصر یعنی (a_2) به عنوان pivot انتخاب می‌شود.
   - (a_2) در جای خود قرار می‌گیرد.
   - عناصر باقی‌مانده ((a_4) , (a_3)) بزرگتر از (a_2) هستند و به زیردرخت راست می‌روند.

a1 (pivot)

a2 (pivot)

[a3, a4]

3. مرحله سوم : در زیردرخت راست، کوچک‌ترین عنصر یعنی (a_3) به عنوان pivot انتخاب می‌شود.
   - (a_3) در جای خود قرار می‌گیرد.
   - عنصر باقی‌مانده (a_4) بزرگتر از (a_3) است و به زیردرخت راست می‌رود.

a1 (pivot)

a2 (pivot)

a3 (pivot)

[a4]

این ساختار ، یک درخت تصمیم کاملا نامتوازن یا زنجیره راست را تشکیل می‌دهد.

- تعداد مقایسه برای n=4
  - در مرحله اول (ریشه (a_1)) : (n – 1 = 3) مقایسه برای جدا کردن عناصر کوچکتر و بزرگتر از (a_1)
  - در مرحله دوم (گره (a_2)) : (n – 2 = 2) مقایسه.
  - در مرحله سوم (گره (a_3)) : (n – 3 = 1) مقایسه.

- مجموع مقایسه ها : (6 = 3 + 2 + 1) مقایسه.

- مرتبه زمانی بر حسب (n) (در بدترین حالت) :
  - در بدترین حالت ، انتخاب pivot همواره کوچک‌ترین یا بزرگ‌ترین عنصر است. این منجر به یک درخت تصمیم با عمق (n) می‌شود. مجموع مقایسه‌ها در این حالت به صورت زیر است $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$ این مقدار برابر با $O(n^2)$ است. بنابراین، مرتبه زمانی QuickSort در بدترین حالت $\theta(n^2)$ می‌باشد.

بخش ب. Pivot تصادفی یکنواخت

1. QuickSort متناظر با یک درخت تصمیم دودویی تصادفی

وقتی pivot به صورت تصادفی یکنواخت از آرایه انتخاب می‌شود، ترتیب انتخاب pivot ها و نحوه تقسیم‌بندی آرایه در هر مرحله، به یک فرآیند تصادفی تبدیل می‌شود. هر اجرای QuickSort روی یک ورودی معین، می‌تواند منجر به یک درخت تصمیم متفاوت شود. این درخت ، نمایانگر توالی عملیات مقایسه و پارتیشن بندی است.

- نحوه ایجاد تصادفی بودن : در هر فراخوانی بازگشتی ، pivot از زیر آرایه فعلی به طور تصادفی انتخاب می شود. این انتخاب تصادفی، تضمین می کند که با احتمال بالا، pivot انتخابی عنصری نزدیک به میانه زیرآرایه باشد. این امر از وقوع بدترین حالت که در آن pivot همواره کوچک‌ترین یا بزرگ‌ترین عنصر است ، جلوگیری می کند.
- درخت تصمیم دودویی تصادفی : نتیجه این فرآیند ، ساختاری است که مشابه یک درخت جستجوی دودویی تصادفی است.. در این درخت ، گره های داخلی نمایانگر pivot های انتخاب شده در هر مرحله و زیر درخت های چپ و راست نمایانگر عناصری هستند که کوچکتر یا بزرگتر از pivot هستند.

2. عمق یک عنصر تصادفی در درخت بازگشتی : شهود O(log n)

فرض کنید (x) یک عنصر تصادفی در آرایه ورودی (A) با اندازه (n) باشد. ما به دنبال امید ریاضی عمق (x) در درخت بازگشتی QuickSort هستیم.

- شهود : در هر مرحله ، QuickSort آرایه را به دو زیر آرایه تقسیم می‌کند. اگر pivot به طور تصادفی انتخاب شود، انتظار داریم که pivot نزدیکه به میانه باشد. این بدان معناست که هر دو زیر آرایه حاصل ، تقریبا نیمی از عناصر را شامل می شوند.
  - در اولین مرحله ، (x) در یکی از زیر آرایه هایی ک قرار می گیرد که اندازه اش تقریبا (n/2) است.
  - در مرحله بعد ،(x) در زیر آرایه ای قرار می گیرد که اندازه اش تقریبا (n/4) است.
  - این روند ادامه می یابد تا زمانی که (x) به عنوان pivotانتخاب شود یا در یک زیر آرایه با اندازه 1 قرار گیرد.

تعداد مراحلی که طول می کشد تا اندازه زیر آرایه به 1 برسد، با حل معادله $\frac{n}{2^k} = 1$ به دست می آید. بنابراین، انتظار داریم که عمق یک عنصر تصادفی حدود $O(\log n)$ باشد. این شهود بر این اساس استوار است که هر بار ، اندازه مسئله به طور قابل توجهی کاهش می یابد، مشابه عملیات جستجو در یک درخت جستجوی دودویی متوازن.

3. ارتباط با ساختار درخت جستجوی دودویی تصادفی : عمق مورد انتظار گره ها O(logn) و مشابهت با QuickSort

ساختار درخت بازگشتی QuickSort با pivot تصادفی ، شباهت زیادی به ساختار یک درخت جستجوی دودویی تصادفی دارد.

- درخت جستجوی دودویی تصادفی : در RBST ، عناصری که وارد درخت می شوند ، به طور تصادفی به عنوان گره ریشه انتخاب می شوند. سپس عناصر بعدی به صورت بازگشتی در زیردرخت چپ یا راست قرار می گیرند. این فرآیند تضمین می کند که با احتمال بالا، درخت حاصل متوازن خواهد بود.
- شباهت با QuickSort
  - انتخاب تصادفی : هر دو الگوریتم بر پایه انتخاب تصادفی (چه pivot و چه ریشه) بنا شده‌اند.
  - ساختار درختی : هر دو منجر به ساختاری درختی می شوند که نمایانگر سلسله مراتب داده هاست.

- عمق مورد انتظار : عمق مورد انتظار گره ها در یک RBST برابر با O(logn) است. مشابه این ، عمق مورد انتظار یک عنصر (گره) در درخت بازگشتی QuickSort با pivot تصادفی است. این بدان معناست که در حالت متوسط، QuickSort عملکردی بسیار کارآمد دارد و مرتب سازی را در زمان O(log n) انجام می دهد.

این ارتباط نشان می دهد که انتخاب تصادفی QuickSort ، pivot را از بدترین حالت $\theta(n^2)$ دور کرده و به سمت حالت متوسط $\theta(n \log n)$ سوق می دهد، که این نتیجه با تحلیل درخت جستجوی دودویی تصادفی مطابقت دارد.