جزوه جلسه هشتم داده ساختارها و الگوریتم

۲۵ مهر ۱۴۰۰

فهرست مطالب

۱ درخت دودویی جست و جو یا Binary Search Tree

مبحث را با یک مثال معروف شروع میکنیم:

۱.۱ مسئله رزرو باند فرودگاه

در یک فرودگاه، مسئول برج مراقبت باید عملیات زیر را انجام دهد:
۱. به مدت زمان t قبل و بعد فرود هر هواپیما، نباید هیچ هواپیمایی در باند فرودگاه باشد.
۲. در صورت ارضا شدن مورد اول، هواپیما به لیست رزرو اضافه میشود.
۳. بعد از فرود موفقیت آمیز، هواپیما از لیست انتظار خط میخورد.

عملیات فوق، دقیقا معادل عملیات تعریف شده در واسط مجموعه ای مرتب پویا هستند؛ به نحوی که مورد اول همان find _next(A) و find _prev(A)، مورد دوم معادل insert(A) و مورد سوم نیز delete(A) میباشد.

۲.۱ مقایسه پیچیدگی زمانی مسئله برای داده ساختارهای مختلف

اگر مسئله را با یک آرایه معمولی حل کنیم اردر های زمانی به شکل زیر خواهند بود:
۱. insert(A): $O(1)$
۲. delete(A): $O(n)$
۳. find(A): $O(n)$
۴. find _next(A)/ find _prev(A): $O(n)$

برای پیاده سازی با یک آرایه مرتب، مرتبه های زمانی به شکل زیر تغییر میکنند:
۱. insert(A): $O(n)$
۲. delete(A): $O(n)$
۳. find(A): $O(logn)$
۴. find _next(A)/ find _prev(A): $O(logn)$

در گام آخر اگر مسئله با یک درخت دودویی جست و جو مدل شود، مرتبه های زمانی زیر را خواهیم داشت:
۱. insert(A): $O(logn)$
۲. delete(A): $O(logn)$
۳. find(A): $O(logn)$
۴. find _next(A)/ find _prev(A): $O(logn)$
در اصل، مرتبه زمانی تمامی عملیات به اندازه $O(height())$ یا ارتفاع درخت است و

حداکثر ارتفاع نیز میتواند n-1 باشد ولی به صورت متوازن، به اندازه $logn$ میباشد.
حال به تعریف درخت دودویی جست و جو برگردیم:

۳.۱ تعریف داده ساختار Binary Search Tree

درخت دودویی جست و جو، درختی دودویی است که ترتیب ندارد و کامل نیست. همچنین هر راس آن، بزرگتر مساوی زیر درخت چپ خود و همچنین کوچکتر مساوی زیر درخت راست خود میباشد.
توجه شود که خاصیت فوق را نمیتوان به صورت راسی (هر راس، بزرگتر مساوی فرزند چپ خود و کوچکتر مساوی فرزند راست خود) برای هر راس بیان کرد.

۴.۱ متد های مختلف پیمایش یک درخت دودویی جست و جو

در هر شیوه، از ریشه شروع کرده و به ترتیب ذکر شده شروع به پیمایش میکنیم:

pre-order: ابتدا خود راس، سپس درخت چپ و بعد از آن، درخت راست راس پیمایش میشود.
post-order: در این متد ترتیب پیمایش به ترتیب درخت چپ، درخت راست و خود راس میباشد.
in-order: در هر راسی که هستیم، ابتدا درخت چپ، سپس خود راس و درنهایت درخت سمت چپ پیمایش میشود.
در این متد، درخت به صورت مرتب و صعودی پیمایش میشود و میتوان ادعا کرد برای هر درخت دودویی، آن درخت BST است اگر و تنها اگر پیمایش in-order آن مرتب شده باشد.

۵.۱ تعریف عملیات تعریف شده برای Binary Search Tree

۱.۵.۱ find(node, k)

این عملیات مانند جست و جوی دودویی (Binary Search) میباشد. از ریشه شروع به پیمایش میکنیم. اگر k بزرگتر از ریشه بود جست و جو را در زیر درخت راست و در صورت کوچک بودن در زیر درخت چپ انجام میدهیم.

```
def find(node, k):
    if node.key == k:
        return node
    elif k < node.key and node.left != None:
        return find(node.left, k)
```

```
    elif k > node.key and node.right != None:
        return find(node.right, k)
    return None
```

همانگونه ک بیان شد، تابع find(node, k) دقیقا عملیات Binary Search را در آرایه in-order انجام میدهد.

۲.۵.۱ insert(node, k)

```
def insert(node, k):
    if k <= node:
        if node.left != None:
            insert(node.left, k)
        else:
            node.left = new _node(key = k, parent = node, left = None, right = None)
    else:
        #Same on right side
```

۳.۵.۱ delete(node)

```
def find _min(node):
    if node.left != None:
        return find _min(node.left)
    return node
```

```
def delete(node):
    if node.left != Node and node.right != None:
        succ = find _min (node.right)
        node.key = succ.key
        delete(succ)
    elif node.left != None:
        # replace left child
    elif node.right != None:
        # replace right child
    else:
        # replace parent
```

۴.۵.۱ find _prev(node)

```
def if _right _child(node):
    return node.parent
```

```
def successor(node):
    if node.right != None:
        return find _min(node.right)
    while is _rigth _child(node) != None:
        node = node.right
    return node.parent
```

کد تابع find _next(node) نیز مانند کد بالا نوشته میشود.