حوری دهش 9821413

به نام خدا

تمرین تئوری سری دوم ساختمان داده

==========

سوال 1:

الف) اگر فقط یک اشاره گر به اول لیست داشته باشیم درست است چون در بدترین حالت عنصر جدید به اخر لیست می خواهد اضافه شود و تنها یه اشاره گره head به ابتدای لیست داریم پس باید به اندازه n تا پیمایش کنیم که بتوانیم اشاره گره head رو به به اخر لیست ببریم و در اخر عنصر جدید رو اضافه کنیم ولی اگر دو اشاره گر داشتیم که یکیش به اول لیست و یکیش به اخر لیست اشاره می کرد اون موقع این عبارت غلط می شد چون در این حالت ما O(1) داشتیم.

ب) غلط است چون اگر بخواهیم یک عنصر را در موقعیت i درج کنیم باید به موقعیت عنصر قبلی اش دسترسی داشته باشیم یعنی اشاره گر عنصر قبلیشو پیدا کنیم پس باید به عقب برگردیم که این کار از O(1) نیست.

ج) غلط است چون لیست ما یک سویه است پس به عنصر یکی مونده به اخر دسترسی نداریم در نهایت باید به اندازه n-1 تا پیمایش کنیم که اشاره گر f ما به عنصر یکی مونده به اخر برسد در نهایت می توانیم عنصر اخر رو حذف کنیم و اشاره گر r در جایی که f اشاره می کند، قرار می گیرد وr->next =null می شود پس کلا این کار از O(n) است.

د) درست است چون اگر min را در یک متغییر ذخیره کنیم هر بار که در پشته پوش و پاپ انجام می شود این متغییر را باید اپدیت کنیم ولی از یه طرف فقط به top دسترسی داریم پس انجام این کار امکان پذیر نیست.

ی) درست است چون اشاره گر lastNode به انتهای لیست اشاره می کند اگر بخواهیم به ابتدای لیست نود اضافه کنیم --> newNode->link=last->link و last->link=newNod که این کار از مرتبه یک است و اگر بخواهیم به اخر لیست نود اضافه کنیم newNode->link=last->link و last->link=newNode و last=newNode که این هم باز از مرتبه یک است.

حوری دهش 9821413

سوال 2 :

فرض می کنیم که لینکد لیست های مرتب شده ما به صورت صعودی مرتب شده اند و تنها یک اشاره گر به اول لیست داریم.

	Unsorted,singly Linked List	sorted,singly Linked List	Unsorted,Doubly Linked	sorted,Doubly Linked
Search(L,K)	O(n)	O(n)	O(n)	O(n)
Insert(L,K)	O(1)	O(n)	O(1)	O(n)
Delete(L,K)	O(n)	O(n)	O(n)	O(n)
Successor(L,K)	O(n)	O(1)	O(n)	O(1)
Predecessor(L,K)	O(n)	O(n)	O(n)	O(1)
Minimum(L)	O(n)	O(1)	O(n)	O(1)
Maximum(L)	O(n)	O(n)	O(n)	O(1)

سوال 3 :

```
Reverse ( L )
        if L = null
                then return L
        r ← Reverse ( next [L] )
        next [ next [ L ] ] ← L
        next [ L ] ← null
       return r
```

این الگوریتم به دو علت غلط است :

1) زمانی که L=null است ما به L->next دسترسی نداریم یعنی در واقع برنامه می خواهد به next نال دسترسی پیدا کند که این موضوع غلط است پس برنامه به ارور می خورد.

2) زمانی که L=null است یعنی به نود اخر رسیدیم و برنامه return null می کند در این حالت هم r=null می شود و هر بار برنامه return r می کند یعنی در واقع هر بار برنامه return null می کند پس برنامه ما همیشه null بر می گرداند --> پس اگر تا یکی قبل از نود اخر پیش برویم مشکل حل می شود.

حوری دهش 9821413

سوال 4 :

الف) ساخت یک پشته با دوصف :

عملیات push یا push (a element) :

فرض می کنیم می خواهیم عنصر a را پوش کنیم و دو صف queue1 , queue2 داریم --> دو حالت داریم:

1) اگر queue1 خالی باشد a را در queue1 پوش می کنیم <-- Time Complexity : O(1)

2) اگر queue1 خالی نباشد و queue2 خالی باشد همه عنصرهای queue1 را در queue2 می ریزیم تا queue1 خالی شود سپس عنصر a را در queue1 پوش می کنیم و بعد از ان همه عنصر ها را از queue2 به queue1 بر می گردانیم --> Time Complexity : O(n)

عملیات pop :

یک عنصر را از queue1 پاپ می کنیم چون اولین عنصری که اول از همه وارد صف می شود می تواند اول از همه از صف خارج شود (طبق قاعده FIFO) --> Time Complexity : O(1)

ب) ساخت یک صف با دو پشته :

عملیات push یا push (a element) :

فرض می کنیم می خواهیم عنصر a را پوش کنیم و دو پشته stack1 , stack2 داریم --> دو حالت داریم:

1) اگر stack1 خالی باشد a را در stack1 پوش می کنیم <-- Time Complexity : O(1)

2) اگر stack1 خالی نباشد و stack2 خالی باشد همه عنصرهای stack1 را در stack2 می ریزیم تا stack1 خالی شود سپس عنصر a را در stack1 پوش می کنیم و بعد از ان همه عنصر ها را از stack2 به stack1 بر می گردانیم --> Time Complexity : O(n)

عملیات pop :

یک عنصر را از stack1 پاپ می کنیم چون اولین عنصری که اول از همه وارد استک می شود اخر از همه می تواند خارج شود (طبق قاعده LIFO) --> Time Complexity : O(n)

حوری دهش 9821413

سوال 5 :

این مسئله مانند مسئله biggest advertising banner حل می شود.

ابتدا همه اعدادمان را به صورت میله در نظر می گیریم و عرض همه میله ها را برای سادگی 1 در نظر می گیریم یعنی مثلا در دنباله {6, 2, 5, 4, 5, 1, 6} ما میله ای به ارتفاع 6 و عرض 1 داریم.

برای حل این مسئله برای هر میله "x"، مساحت را با "x" به عنوان کوچکترین میله در مستطیل محاسبه می کنیم. اگر چنین مساحتی را برای هر میله 'x' محاسبه کنیم و حداکثر همه مساحت را پیدا کنیم، کار ما انجام می شود.

پس ما همه میله ها را از چپ به راست پیمایش می کنیم، استکی از میله ها ایجاد می کنیم به این صورت که هر میله یک بار به استک push می شود. وقتی میله بعدی که می خواهد در استک قرار گیرد، اگر ارتفاعش از میله قبلی درون استک کمتر یا مساوی باشد انقدر از استک pop می کنیم که میله با ارتفاع کمتر بتواند در استک قرار گیرد.

هنگامی که یک میله pop می شود ما مساحت را با میله های pop شده همانند میله های کوچکتر حساب می کنیم. چگونه ما index های چپ و راست میله های pop شده را می گیریم؟ index کنونی ما یعنی i به ما index راست را می ده و index ایتم قبلی در استک index چپ را به ما می ده.

الگوریتم ما به این صورت کار می کند:

1) یک پشته خالی ایجاد می کنیم.

2) شروع از اولین میله و دنبال کردن همه میله ها (hist [i]) از جایی که i بین 0 تا n-1 است که تعداد میله ها = n است :

الف) اگر استک خالی است یا hist [i] > میله بالای استک باشد انگاه i را به استک push می کنیم.

ب) اگر این میله کوچکتر از میله بالای استک باشد انگاه میله بالای استک را pop می کنیم تا اینکه میله مورد نظر ما از میله کنونی دورن استک بزرگتر شود در این حالت میله ما بالای استک قرار می گیرد --> میله حذف شده را با hist [tp] در نظر می گیریم و مساحت مستطیل را با hist [tp] همانند کوچکترین میله محاسبه می کنیم. برای hist [tp] <-- index چپش، ایتم قبلی ما در استک است و index راستش i می باشد.

3) اگر استک خالی نباشد انگاه یک به یک همه میله ها را از استک pop می کنیم و قسمت دوم بخش 2 را یعنی ب-2 برای هر میله ی pop شده انجام می دهیم.

Code:

```
// C++ program to find maximum rectangular area in
// linear time
#include<bits/stdc++.h>

using namespace std;

// The main function to find the maximum rectangular
// area under given histogram with n bars
int getMaxArea(int hist[], int n) {
    // Create an empty stack. The stack holds indexes
    // of hist[] array. The bars stored in stack are
    // always in increasing order of their heights.
    stack<int> s;

    int max_area = 0; // Initialize max area
    int tp; // To store top of stack
    int area_with_top; // To store area with top bar
    // as the smallest bar
```

```
    // Run through all bars of given histogram
    int i = 0;
    while (i < n) {
        // If this bar is higher than the bar on top
        // stack, push it to stack
        if (s.empty() || hist[s.top()] <= hist[i])
            s.push(i++);

            // If this bar is lower than top of stack,
            // then calculate area of rectangle with stack
            // top as the smallest (or minimum height) bar.
            // 'i' is 'right index' for the top and element
            // before top in stack is 'left index'
        else {
            tp = s.top(); // store the top index
            s.pop(); // pop the top

            // Calculate the area with hist[tp] stack
            // as smallest bar
            area_with_top = hist[tp] * (s.empty() ? i : i - s.top() - 1);

            // update max area, if needed
            if (max_area < area_with_top)
                max_area = area_with_top;
        }
    }

    // Now pop the remaining bars from stack and calculate
    // area with every popped bar as the smallest bar
    while (s.empty() == false) {
        tp = s.top();
        s.pop();
        area_with_top = hist[tp] * (s.empty() ? i : i - s.top() - 1);
        if (max_area < area_with_top)
            max_area = area_with_top;
    }
    return max_area;
}

// Driver program to test above function
int main() {
    int hist[] = {6, 1, 5, 4, 5, 2, 6};
    int n = sizeof(hist) / sizeof(hist[0]);
    cout << "Maximum area is " << getMaxArea(hist, n);
    return 0;
}
```

حوری دهش 9821413

سوال 6 :

سه پوینتر current->head و next->null و prev->null ایجاد می کنیم.

Time Complexity : O(n)
Space Complexity : O(1)

Code:

```
While (current != null){
    next = current->next;
    current->next = prev;
    prev = current;
    current = next;
}
*head_ref = prev;
```

برای اینکه الگوریتم ما O(1) شود می توانیم یک ارایه از پوینترها به تمامی نودهای لیست ذخیره کنیم و ان را برگردانیم در این حالت با داشتن ادرس تمامی نودها ما می توانیم به O(1) دست پیدا کنیم.

سوال 7 :

الف) [4, 3 , 2, 8, 7, 6, 9, 10 , 5 , 1] ؟؟

از پارکینگ قطارها استفاده می کنیم به این صورت که در ابتدا می گوییم 4 باید در خروجی ما باشد پس هرچه قبل از 4 است درون استک پوش می کنیم پس:

1) 1 را پوش می کنیم --> 2 را پوش می کنیم --> 3 را پوش می کنیم.

2) 3 را پاپ می کنیم --> 2 را پاپ می کنیم.

3) 5 را پوش می کنیم --> 6 را پوش می کنیم --> 7 را پوش می کنیم --> 8 را پوش می کنیم.

4) 8 را پاپ می کنیم --> 7 را پاپ می کنیم --> 6 را پاپ می کنیم.

5) 9 را پوش می کنیم.

6) 9 را پاپ می کنیم.

7) 10 را پوش می کنیم.

8) 10 را پاپ می کنیم --> 5 را پاپ می کنیم --> و در نهایت 1 را پاپ می کنیم.

پس جایگشت مورد نظر ما ساخته می شود.

ب) [4, 3 , 8 , 7, 9 , 5 , 10 , 6 , 2 , 1] ؟؟

در ابتدا می گوییم 4 باید در خروجی ما باشد پس هرچه قبل از 4 است درون استک پوش می کنیم پس:

1) 1 را پوش می کنیم --> 2 را پوش می کنیم --> 3 را پوش می کنیم.

2) 3 را پاپ می کنیم.

3) 5 را پوش می کنیم --> 6 را پوش می کنیم --> 7 را پوش می کنیم --> 8 را پوش می کنیم.

4) 8 را پاپ می کنیم --> 7 را پاپ می کنیم.

5) 9 را پوش می کنیم.

6) 9 را پاپ می کنیم.

7) در این مرحله اگر بخواهد 5 در خروجی باشد اول باید 6 را پاپ کنیم که در این حالت جایگشت مورد نظرما ساخته نمی شود

ج) ؟؟

سوال 8 :

برای حل این سوال از مسئله برج هانوی کمک می گیریم که در این مسئله ما دو تا شرط داریم :

1) نمی توانیم دیسک بزرگتر را روی دیسک کوچکتر قرار بدهیم.

2) فقط یک دیسک را می توانیم در یک زمان جابه جا کنیم.

که این دو شرط همانند شرط های این مسئله هستند در این مسئله ما 4 تا میله داریم --> یکی زمین و سه تا قفسه، که از این الگوریتم می رویم :

Time Complexity : $O(2^n - 1) = O(2^n)$

Code:

```
# Recursive program for Tower of Hanoi
# Recursive function to solve Tower
# of Hanoi puzzle

def towerOfHanoi(n, from_rod, to_rod, aux_rod1, aux_rod2):
    if (n == 0):
        return
    if (n == 1):
        print('Move disk', n, 'from rod', from_rod, 'to rod', to_rod)
        return

    towerOfHanoi(n - 2, from_rod, aux_rod1, aux_rod2, to_rod)

    print('Move disk', n - 1, 'from rod', from_rod, 'to rod', aux_rod2)

    print('Move disk', n, 'from rod', from_rod, 'to rod', to_rod)

    print('Move disk', n - 1, 'from rod', aux_rod2, 'to rod', to_rod)

    towerOfHanoi(n - 2, aux_rod1, to_rod, from_rod, aux_rod2)


# driver program
n = 4  # Number of disks
```

```
# A, B, C and D are names of rods
towerOfHanoi(n, 'A', 'D', 'B', 'C')
```

سوال 9 :

الگوریتم ما به این صورت است که در ابتدای کار رشته مورد نظر را می گیرد و کاراکتر های رشته را از اول تا اخر دورن استک و صف می ریزد و تا زمانی که استک ما خالی نشده باشد و s.top() == q.front() باشد می تواند یکی از استک pop کند و یکی از صف که در این حالت هر بار مقدار top و front اپدیت می شود و باز این عمل تکرار می شود تا زمانی که یا :

1) استک ما خالی شود که در این حالت متوجه می شویم رشته ما متقارن است.

2) و یا اینکه s.top() != q.front() است پس متوجه می شویم که رشته متقارن نیست.

Code:

```
#include <iostream>
#include <stack>
#include <queue>
#include <string>

int main() {
    while (true) {
        std::string letters;
        std::cout << "Please enter a string (Enter - exit): ";
        std::getline(std::cin, letters);

        if (letters.empty()) break;

        std::stack<char>
                s(std::stack<char>::container_type(letters.begin(), letters.end()));
        std::queue<char>
                q(std::queue<char>::container_type(letters.begin(), letters.end()));

        while (!s.empty() && s.top() == q.front()) {
            s.pop();
            q.pop();
        }

        if (s.empty()) std::cout << "The string is a palindrome" << std::endl;
        else std::cout << "The string is not a palindrome" << std::endl;
    }

    return 0;
}
```

حوری دهش 9821413

سوال 10 :

الگوریتم:

اسم الگوریتم ما الگوریتم تشخیص حلقه فلوید است و روند ما در این الگوریتم این است که ما دو اشاره گره fast , slow داریم که اشاره گر slow یک پرش دارد و مطابق با هر پرشی که اشاره گر slow انجام می دهد، اشاره گر fast دو پرش انجام می دهد. اگر یک حلقه وجود داشته باشد، هر دو اشاره گر fast , slow دقیقا به همان نود خواهند رسید. اگر هیچ حلقه ای در لینکد لیست وجود نداشته باشد، اشاره گر fast قبل از اینکه اشاره گر slow به پایان یا null برسد، به انتهای لینکد لیست می رسد.

Time Complexity : O(N)
Space Complexity : O(1)

Where N is number of Nodes in Linked-List.

Code:

```
bool detectCycle(Node *head) {
    if (head == NULL || head->next == NULL) {
        return false;
    }

    //  Slow Pointer - This will be incremented by 1 Nodes.
    Node *slow = head;
    //  Fast Pointer  - This will be incremented by 2 Nodes.
    Node *fast = head->next;

    while (slow != fast) {
        //  We reached the end of the List and haven't found any Cycle.
        if (fast == NULL || fast->next == NULL) {
            return false;
        }
        slow = slow->next;
        fast = fast->next->next;
    }

    //  We found a Cycle.
    return true;
}
```