Hw3 report
BST

MARCH 29

Amirkabir university
Authored by: mohammad reza babaei mosleh

:BST

در این گزارش به بررسی تمرین شماره سوم میپردازیم
فایل های ارسالی شامل دو بخش .h و.cpp. بوده ابتدا به بررسی فایل .h و نکات و چالش های ان میپردازیم سپس در فایل .cpp به بررسی و توضیحی مختصر از هر method میپردازیم و به طور کامل به چالش های موجود در سیر نوشتن تمرین میپردازیم

بررسی فایل ها:

در ابتدا فایل های include شده را توضیح میدهیم
initializer_list: برای نوشتن بخش challange از ان استفاده شده که در ادامه بیشتر توضیح داه میشود.
compare : برای استفاده از عملگر های سه طرفه نیاز است از ان استفاده شود.
در ادامه به تعریف BST پرداخته ایم با توجه به اینکه Node عضوی از BST است پس باید داخل ان تعریف شود پس در نتیجه در اولین خط به تعریف Node در داخل ان پرداخته ایم سپس تعریف ان را در خارج از بدنه کلاس می اوریم.

در فایل .h نکته دیگری وجود ندارد به غیر از انکه مشاهده میشود دو خط زیر هر دو تعریف بیرون بدنه دو کلاس اورده شده اند دلیل ان این است که این operator مربوط به هیچ کدام از اون دو نیست و مربوط به ostream میباشد دو راه برای تعریف ان وجود دارد یکی friend کردن ان ها با هر کلاس و دیگری تعریف ان ها به شکل زیر و بردن بدنه در داخل .cpp.

```
std::ostream& operator<<(std::ostream& os, BST::Node node);

std::ostream& operator<<(std::ostream& os, BST bst);
```

در ادامه به بررسی فایل cpp. می پردازیم :(با توجه به در خواست استاد همه توابع توضیح داده نشده و تنها بخش های چالشی توضیح کامل داده میشوند.

```
void BST::bfs(std::function<void(Node*& node)> func)
{
    //defining lambda func for calculate depth(height) of tree
    std::function<size_t(Node*& node)> depth = [&](Node* node)->size_t
    {

        if(node == nullptr)
        {
            return 0;
        }
        else
        {
            size_t left_depth{ depth(node->left) };
            size_t right_depth{ depth(node->right) };

            if(left_depth > right_depth)
            {
                return left_depth + 1 ;
            }
            else
            {
                return right_depth + 1 ;
            }
        }

    };

    //defining lambda function for applying func to every node
    std::function<void(size_t current_depth, Node*& node)> apply = [&](size_t
current_depth , Node* node)->void
    {

        if(node == nullptr)
        {
            return;
        }

        if(current_depth == 1)
        {
            func(node);
        }
        else if(current_depth > 1 )
```

```
        {
            apply(current_depth - 1 , node->left );
            apply(current_depth - 1 , node->right);
        }

    };

    size_t tree_depth{ depth(root) };

    if(tree_depth == 0)
    {
        return;
    }
    else
    {
        for(size_t current_depth{ 1 } ; current_depth <= tree_depth ; current_depth++)
        {
            apply(current_depth, root);
        }
    }

}
```

bsf : در ابتدا مطلب جدید این method استفاده از lambda function میباشد این توابع که به صورت درجا تعریف میشوند و میتوانند انواع ورودی را بگیرند توجه شود که در ارگمان ان ها [&] مشاهده میشود که ب معنی این است که متغیر ها و همچنین خود تابع از نوع رفرنسی بوده اند در تابع depth به صورت بازگشتی اعضا از هر طرف فراخوانده میشوند و هر عمق بیشتر در از هر طرف در هر مرحله باز گردانده میشود تا در نهایت عمق درخت به دست اید.
پس از به دست اوردن عمق درخت و با استفاده از ان تابع دیگری ساخته میشود که مطابق با الگوریتمی که مشاهده میکنید تابع ورودی func را روی تمام node ها به ترتیب صدا میزند

```
size_t BST::length()
{

    size_t counter{};
    std::function<void(size_t& count, Node*& root)> length_cal = [&](size_t count,
Node* root)->void
    {
        if(root != nullptr)
```

```
        {
            counter += 1;
        }
        else
        {
            return;
        }

        length_cal(count, root->left);
        length_cal(count, root->right);

    };
    length_cal(counter, root);

    return counter;
}
```

length : تابع بالا با الگوریتمی متفاوت و به صورت نا منظم همه نود ها را بررسی میکند و با رسیدن به هر **Node** ان را میشمارد تا تعداد **Node** ها مشخص شود.

```
bool BST::add_node(int value)
{

    Node* node{ new Node{ value } };

    if(root == nullptr) //empty tree
    {
        root = node;
        root->value = value;
        return true;
    }

    std::function<bool(int value, Node*& root)> adder = [&](int value, Node* root)-
>bool
    {

        if(root->value == value)
        {
            std::cout << "this value already exists in the tree" << std::endl;
            return false;
        }
        else if(root->value > value)
```

```
        {

            if(root->left == nullptr)
            {

                root->left = node;
                return true;
            }
            else
            {
                adder(value, root->left);
            }

        }
        else if(root->value < value)
        {

            if(root->right == nullptr)
            {
                root->right = node ;
                return true;
            }
            else
            {
                adder(value, root->right);
            }

        }

        return false;

    };

    if(adder(value, root))
    {
        return true;
    }
    else
    {
        return false;
    }

}
```

add_node : این تابع چند نکته مهم دارد که در ادامه بحث میشود نکته اول ان است که با توجه به اینکه قرار است یک **Node** در داخل **Method** تعریف و به **Node** اضافه شود و همچنین رفرس ان برگردانده شود نیاز است که این **Node** از را با استفاده از **dynamic variable** ها تعریف کنیم که بعد از تمام شدن **method** حافظه مربوط به ان از دست نرود و متغیری **local** نباشد و بشود بعدا به ان هم دسترسی داشت طبیعتا این کار طبعاتی دارد که در ادامه باید ان ها را حل کنیم.

در ادامه **Node** ساخته شده را با استفاده از خواص **BST** به محل درست خود برده و **pointer** ان را منصوب میکنیم تا درخت ما ساخته شود.

توجه شود که اگر درخت خالی باشد اولین **Node** به عنوان **root** در نظر گرفته شده و درخت حول ان ساخته میشود.

```
BST::Node** BST::find_parrent(int value)
{

    if(find_node(value) == nullptr)
    {
        return nullptr;
    }

    std::function<Node**(int value, Node** root)> parrent_finder = [&](int value,
Node** root)->Node**
    {

        if((*root) == nullptr)
        {
            return nullptr;
        }

        if((*root)->value == value)
        {
            std::cout << "given valuue is the root and has no parrents" << std::endl;
            return nullptr;
        }
        else if(((*root)->left) !=nullptr && ((*root)->left)->value == value)
        {
            std::cout << "parrent is   " << (*root)->value << std::endl;
            return root;
        }
        else if(((*root)->right) !=nullptr && ((*root)->right)->value == value)
```

```
        {
            std::cout << "parrent is    " << (*root)->value << std::endl;
            return root;
        }
        else if((*root)->value > value)
        {
            return parrent_finder(value, &((*root)->left));
        }
        else if((*root)->value < value)
        {
            return parrent_finder(value, &((*root)->left));
        }

        return nullptr;

    };

    Node** parrent{ parrent_finder(value, &root) };
    return parrent;

}
```

find_parrent : این تابع دارای نکته خاصی نیست و عملکرد ان تا حد زیادی شبیه **find_node** است تنها نکته قابل توجه راجب ان این است که در این الگوریتم و الگورتیم های بعدی که خروجی ان ها به صورت **Node** است (مانند **add_node**) دیگر نیازی به تعریف **dynamic variable** نیست زیرا در **add_node** این کار انجام شده است و ررنس های پیدا شده دیگر از دست نمیروند.

```
bool BST::delete_node(int value)
{
    Node** target_node{ find_node(value) };

    if(target_node == nullptr)
    {
        return false;
    }

    if((*target_node)->left == nullptr && (*target_node)->right == nullptr)
    {
        (*target_node) = nullptr;
        return true;
```

```
    }

    if((*target_node)->left == nullptr && (*target_node)->right != nullptr)
    {
        *target_node = (*target_node)->right;
        return true;
    }

    if((*target_node)->left != nullptr && (*target_node)->right == nullptr)
    {
        *target_node = (*target_node)->left;
        return true;
    }

    Node** successor{ find_successor(value) };
    (*target_node)->value = (*successor)->value;
    *successor = nullptr;
    return true;

}
```

: delete_node

این تابع نیز دارای نکته خاصی نیست تنها نکته الگوریتم کار ان است که در هر مرحله **successor** را برای **Node** مورد نظر پیدا میکند و با جایگذاری ان به جای **Node** مساله حل شده است.

```
std::ostream& operator<<(std::ostream& os, BST bst)
{
    os << std::string(80, '*') << std::endl;

    std::function<void(BST::Node*& node)> print = [&](BST::Node* node)->void
    {
        os << node << "\t" ;
        os << "=> value: " << node->value << "\t";
        os << "left :"<< std::left << std::setw(15) << node->left << "\t" << "right: "
<< node->right << std::endl;
    };

    bst.bfs(print);

    os << "binary search tree size: " << bst.length() << std::endl;
```

```
    os << std::string(80, '*') << std::endl;
    return os;
}
```

: operator <<

این عملگر همانطور که ثلب تر گفته شد متعلق به هیچ کدام از کلاس ها نیست پس بیرون ان ها تعریف میشود تنهیا نکته قابل توجه ان این است که خروجی ان باید مانند ورودی یعنی **os** باشد تا عملگر **<<** بتواند دوباره روی ان فراخوانده شود.

```
BST& BST::operator++()
{
    //ussing lambda function just for using bfs
    std::function<void(Node*& node)> plus_1 = [&](Node* node)->void
    {
        node->value = node->value + 1;
    };

    bfs(plus_1);

    return *this;
}
```

: operator ++(left)

این عملگر بسیار کارکرد ساده ای دارد تنها نکته مهم در مورد ان این است که خروجی ان باید به صورت ***this** باشد تا این عملگر بتوان در معادلات و همراه با عملگر های دیگر نیز استفاده شود.

```
BST BST::operator++(int)
{
    BST tmp{ *this };
    ++(*this);

    return tmp;
}
```

operator++(right) :

تنها نکته درباره این عملگر ان است که با توجه به نحوه کارکرد ان یعنی اولویت کمتر ان و انجام ان در انتهیا محاسبات نیاز است که **bst** اصلی در یک متغیر موقت ذخیره شود و ان متغیر به عنوان خروجی برگردانده شود.

```
BST::~BST()
{

    std::vector<Node*> nodes;
    bfs([&nodes](BST::Node*& node){nodes.push_back(node);});
    for(auto& node: nodes)
    {
        delete node;
    }
}
```

distructor :

میدانیم وقتی از متغیر های دینامیک استفاده میکنیم باید خودمان هم ان ها را پاک کنیم این کار به شکل بالا در انتهای طول عمر **bst** انجام میشود .

```
BST::BST(const BST& bst) // copy constructor
{

    std::function<void(Node*const& root_org, Node** root)> filler = [&](Node*const
root_org, Node** root)->void
    {

        if(root_org != nullptr)
        {
            *root = new Node{ (root_org->value) };
        }


        if(root_org->left != nullptr)
        {
            filler(root_org->left, &((*root)->left));
        }

        if(root_org->right != nullptr)
        {
            filler(root_org->right, &((*root)->right));
        }

        return;

    };

    filler(bst.root, &root);
}
```

:copy constructor

نکته مهم درباره ان این است که با توجه به اینکه ما داریم به **reference** ها کار میکنیم پس اگر برای کپی کردن یک متغیر از حالت **default** استفاده کنیم به مشکل بر میخوریم زیرا در این حالت کپی از درخت ما درست نشده و عینا در درخت دیگر قرار میگیرد و دو درخت مستقل نخواهیم داشت برای این کار باید برای هر **Node** از درخت جدید نیز یک **new Node** انجام دهیم تا رفرنس ها از یکدیگر جدا شده و یکسال نباشد و تنها **value** ها برابر باشد.

```
BST& BST::operator=(const BST& bst)
{
    if(this == &bst) // bst = bst
    {
        return *this;
    }

    std::vector<Node*> nodes;
    bfs([&nodes](BST::Node*& node){nodes.push_back(node);});
    for(auto& node: nodes)
    {
        delete node;
    }

    std::function<void(Node*const& root_org, Node** root)> filler = [&](Node*const
root_org, Node** root)->void
    {

        if(root_org != nullptr)
        {
            *root = new Node{ (root_org->value) };
        }


        if(root_org->left != nullptr)
        {
            filler(root_org->left, &((*root)->left));
        }

        if(root_org->right != nullptr)
        {
            filler(root_org->right, &((*root)->right));
        }

        return;

    };

    filler(bst.root, &root);

    return *this;

}
```

: operator =

نکته مهم در مورد ان این است که **bst** که در سمت چپ قرار میگیرد و عملگر روی ان فراخوانی میشود نیاز است که به صورت کامل پاک شود این کار را به روشی که پیشتر اشاره شد انجام میدهیم و سپس دستور **copy constructor** را تکرار میکنیم.

```
BST& BST::operator=(BST&& bst)
{
    // deleting old nodes
    std::vector<Node*> nodes;
    bfs([&nodes](BST::Node*& node){nodes.push_back(node);});
    for(auto& node: nodes)
    {
        delete node;
    }

    // stealing the root pointer
    root = bst.root;
    bst.root = nullptr;

    return *this;

}


BST::BST(BST&& bst)
{
    root = new Node{};
    root = bst.root;
    bst.root = nullptr;
}
```

Constructor move version & operator = move version

برای اینکه کد **optimize** تر باشد از این دو استفاده میشود نکته جالب در مورد ان ها این است که دیگر نیاز نیست از هر **Node** یک کپی تهیه شود و با توجه به **rvalue** بودن ورودی میتوان مقادیر ان را برداشت(دزدید😂) نکته مهم نیز در همینجاست با توجه به اینکه ورودی **rvalue** است و بلافاصله پاک میشود پس **distructor**ان فراخوانی میشود و مقادیر مورد نیاز ما را نیز پاک میکند برای جلوگیری از این کار مقدار **root** ان را حذف میکنیم تا دسترسی **distructor** به زیشه و در نتیجه تمامی **Node** ها از دست برود.

: challenge

در این بخش ما باید کار میکردیم که **constructor** بتواند مقادیر نا مشخص با تعداد نامشخص را به عنوان ورودی دریافت کرده و درخت متناظر را تشکیل دهد.

```
BST::BST(std::initializer_list<int> args)
    : root{ nullptr }
{
    for(auto i : args)
    {
        add_node(i);
    }
}
```

این کار را با استفاده از کتابخانه **initializer_list** و دستورات بالا انجام میدهیم و سپس با استفاده از **raged based for loop** مقادیر را اضافه میکنیم نکته قابل توجه ان است که با توجه به شروط تایین شده در **add_node** باید مقدار اولین **root** را برابر با **nullptr** قرار دهیم وگرنه به **segmentaion error** بر میخوریم

پایان

Github link: https://github.com/ghostoftime111/hw3.git