Oblig 2 ADS101

Adam Aske

7. oktober 2021

Innhold

1 Inorder Traversal

Inorder traversal vil printe venstre node, root og høyre node. Dette skjer ved at vi går helt ned i venstre gren, når det da ikke er flere venstre noder å besøke, besøker den rooten og besøker høyre node. Dette gjentar til det ikke er flere noder i treet. Først skjekker vi om "current" har en GetLeft(), hvis den ikke har det printer vi denne noden vi er på og fortsetter til høyre node. Dersom vi finner en venstre node lagrer vi denne i "previous". Vi går da gjennom alle høyre noder den har og setter det som ny "previous" peker. Hvis "previous" ikke har noe høyre node, setter vi dens høyre til current for å lagre pekeren som vi kan returnere til senere, så gjentar stegene. Dersom den har en høyre ndoe, printer vi denne og setter den nye current til denne høyre noden og gjentar stegene.

Listing 1: Oblig2.cpp

```
void Inorder(BinaryNode* root) {
    BinaryNode* current, *previous;
    current = root;

    while(current != nullptr) {
        //Goes down the left, until there is no more to the left, then goes down right until
        if (current->GetLeft() != nullptr) {
            previous = current->GetLeft();
            //Get the bottom right and stores it in previous
            while (previous->GetRight() != nullptr && previous->GetRight() != current) {
                previous = previous->GetRight();
            }
            //Here it has gotten the bottom right it can, without a left
            //Store this in previous
            if (previous->GetRight() == nullptr) {
                //Store the current as previous->Right
                previous->right = current;
                //Then go down the left as long as it goes
                current = current->GetLeft();
            }
            else {
                //If the previous has no right, then print it
                std::cout << current->GetData() << " ";
                //Get the new current as this ones right
                current = current->GetRight();
                //Then returns to check if it has a left, then it prints it
            }
        }
        else {
            //Get the right instead of the left
            std::cout << current->GetData() << " ";
            current = current->GetRight();
        }
    }
}
```

2 Postorde Traversal

Postorder besøker rooten først, så besøker den vesntre og høyre node. Jeg bruker en stack til å utføre denne traversalen. Først pusher jeg root til stacken, den vil da være Top() i stacken. Så lenge stacken ikke er tom, vil disse stegene gjenta seg. I loppen lagrer jeg en peker til Top() fra stacken. Så printer vi verdien

noden har, vi er da ferdige med denne og popper den ut av stacken. Så pusher jeg venstre og høyre node hvis rooten har det. Så gjentar jeg stegene til stacken er tom, alle vil da være besøkt.

Listing 2: Oblig2.cpp

```
void Postorder(BinaryNode* root)
{
    Stack<BinaryNode*> stack;
    stack.Push(root);

    while (!stack.Empty()) {
        BinaryNode* current = stack.Top();
        stack.Pop();

        std::cout << current->GetData() << " ";

        if (current->GetLeft()) {
            stack.Push(current->GetLeft());
        }

        if (current->GetRight()) {
            stack.Push(current->GetRight());
        }
    }
}
```

3 Amount of Nodes

For å finne antall noder i treet, gjør jeg nesten det samme som en Postorder traversal. Jeg lager en int som heter count, som da blir plusset på 1 hver gang den har besøkt en node. Stegene jeg gjør er nøyaktig de samme som Postorder utenom at jeg sier "count'++" istedet for å printe verdien til noden.

Listing 3: Oblig2.cpp

```
int AmountOfNodes(BinaryNode* root)
{
        Stack<BinaryNode*> stack;
        stack.Push(root);

    int count = 0;

    while (!stack.Empty())
    {
                BinaryNode* current = stack.Top();
                stack.Pop();

                count++;

                if (current->GetLeft()) {
                        stack.Push(current->GetLeft());
                }

                if (current->GetRight()) {
                        stack.Push(current->GetRight());
                }
    }
    return count;
}
```

4 Amount of Levels

Denne funksjonen fungerer veldig likt som Postorder og Amount of Nodes. For å bare telle nivåer istedenfor noder så laget jeg en bool som heter hasCounted. Denne holder telling på om det er blitt talt dette nivået. Hvergang den pusher en ny node til stacken sjekker den om den har allerede telt dette nivået, hvis ikke så tell og pluss på count. Den da returnerer count som antall nivåer i treet.

Listing 4: Oblig2.cpp

```
int AmountOfLevels(BinaryNode* root)
{
        Stack<BinaryNode*> stack;
        stack.Push(root);

        int count = 0;
        bool hasCounted = false;
        while (!stack.Empty()) {
                BinaryNode* current = stack.Top();
                stack.Pop();
        hasCounted = false;

                //Only add count if it hasnt done it already
                if (current->GetLeft()) {
                        stack.Push(current->GetLeft());
                        count++;
                        hasCounted = true;
                }

                if (current->GetRight()) {
                        stack.Push(current->GetRight());
                        if (!hasCounted) {
                        count++;
                        hasCounted = true;
                        }
                }
        }
    return count;
}
```

5 Balance Check

Height() funksjonen finner hvor mange nivåer det er nedover fra seg selv. BalancedCheck() sjekker forskjellen mellom høyde fra høyre og vesntre node.

Listing 5: Oblig2.cpp

```
bool BalancedCheck(BinaryNode* root)
{
    //Return true if this is null
    if (!root) {
        return true;
    }
    //Find how many levels there is down from here for both left and right
    int leftHeight = Height(root->GetLeft());
    int rightHeight = Height(root->GetRight());
    //The heights must be either 0, then check this for every level of the tree
    //if any level is not balanced the whole loop returns false
    if ((leftHeight - rightHeight) <= 0 && BalancedCheck(root->GetLeft()) && BalancedCheck(ro
        return true;
```

```
    }
    else {
                return false;
    }

}
```

Listing 6: Oblig2.cpp

```
int Height(BinaryNode* root) {
    if (!root) {
        return 0;
    }
    else {
        //Goes down the
        int a = Height(root->GetLeft());
        int b = Height(root->GetRight());

        //returns the highest of the two
        if (a > b) {
            return 1 + a;
        }
        else {
            return 1 + b;
        }
    }
}
```

6 Resultater for binary tree

Bilde av binary treet som blir brukt:
Inorder Traversal resultat: 1, 2, 4, 7, 8, 10, 16
Postorder Traversal resultat: 7, 10, 16, 8, 2, 4, 1
Amount of Nodes: 7
Amount of Levels: 3
Balanced Check: 1 (0 = Unlbalanced, 1 = Balanced)

7 Lage quadtree med blader på forskjellige nivåer

MakeQuadTree() er en funksjon som returnerer en QuadTree* uten noen noder. Jeg subdivider den en gang, så får jeg 4 blader i den, så subdivider jeg den ene noden og får 4 nye blader der. For å lage de nye kordinatene til de nye quadene så finner jeg forskjellige punkter til å bli brukt i constructorene, kommentert i koden.

Listing 7: Oblig2.cpp

```
int main(){
        QuadTree* tree = MakeQuadTree();
        tree->Subdivide(1);
        tree->q_ne->Subdivide(1);
}
void QuadTree::Subdivide(int amount)
{
        if (amount > 0) {
```

```
                std::cout << "Subdivided: " << amount << std::endl;
                //Gets point between south west and south east
                Vector2D a = (v_sw + v_se) / 2;
                //Gets point between south east and nort heast
                Vector2D b = (v_se + v_ne) / 2;
                //Gets point between noerth east and north weast
                Vector2D c = (v_ne + v_nw) / 2;
                //Gets between north west and sout hwest
                Vector2D d = (v_nw + v_sw) / 2;
                //Gets middle
                Vector2D m = (v_ne + v_sw) / 2;

                //new North East Corner (10,10), (0,10), (10,0), (0,0)
                q_ne = new QuadTree(v_ne,  c, b, m);
                q_ne->Subdivide(amount - 1);
                //new North West Corner (0,10), (-10,10), (0,0), (-10,0)
                q_nw = new QuadTree(c, v_nw, m, d);
                q_nw->Subdivide(amount - 1);
                //new South East Corner (10,0), (0,0), (10,-10), (0,-10)
                q_se = new QuadTree(b, m, v_se, d);
                q_se->Subdivide(amount - 1);
                //new South West Corner (0,0), (-10, 0), (0,-10), (-10,-10)
                q_sw = new QuadTree(m, d, a, v_sw);
                q_sw->Subdivide(amount - 1);
        }
}
```

8 Finne alle blader i et quadtree

Hvis denne noden er et leaf, så bør alle dens noder være nullptr'ere. IsLeaf() returnerer true hvis det er sant. FindAllLeafes() gjør dette med rekursjon på alle sine nodes hvis IsLead() er usann. Alle bladene printer sine kordinater.

Listing 8: Oblig2.cpp

```
bool QuadTree::IsLeaf()
{
        //If its a leaf, all QuadTree pointers should be null
        if (q_ne == nullptr && q_nw == nullptr && q_se == nullptr && q_sw == nullptr) {
                return true;
        }
        else {
                return false;
        }
}
QuadTree* QuadTree::FindAllLeafes()
{
        if (IsLeaf()) {
                std::cout << "NE:(" << v_ne.x << ", " << v_ne.y << "), " << "NW:(" << v_nw.x
                        << "SE:(" << v_se.x << ", " << v_se.y << "), " << "SW:(" << v_sw.x <<
                std::cout << "I have " << data.size() << " elements" << std::endl;
                return this;
        }
        if (q_ne) {
                q_ne->FindAllLeafes();
        }
        if (q_nw) {
                q_nw->FindAllLeafes();
        }
        if (q_se) {
```

```
                q_se->FindAllLeafes();
        }
        if (q_sw) {
                q_sw->FindAllLeafes();
        }
        return nullptr;
}
```

9 Resultat av FindAllLeafes

Funksjon printer kordinatene til quadsa som er blader. Den første quaden er startet kordinatene (100, 100), (0, 100), (100, 0), (0,0).
Resultat:
NE:(100, 100), NW:(75, 100), SE:(100, 75), SW:(75, 75)
NE:(75, 100), NW:(50, 100), SE:(75, 75), SW:(50, 75)
NE:(100, 75), NW:(75, 75), SE:(100, 50), SW:(50, 75)
NE:(75, 75), NW:(50, 75), SE:(75, 50), SW:(50, 50)
NE:(50, 100), NW:(0, 100), SE:(50, 50), SW:(0, 50)
NE:(100, 50), NW:(75, 50), SE:(100, 25), SW:(50, 50)
NE:(75, 50), NW:(50, 50), SE:(50, 50), SW:(25, 50)
NE:(100, 25), NW:(50, 50), SE:(100, 0), SW:(25, 50)
NE:(50, 50), NW:(25, 50), SE:(50, 25), SW:(0, 50)
NE:(50, 50), NW:(0, 50), SE:(50, 0), SW:(0, 0)