

Algoritmer og Datastrukturer - prep sheet

Binary Trees

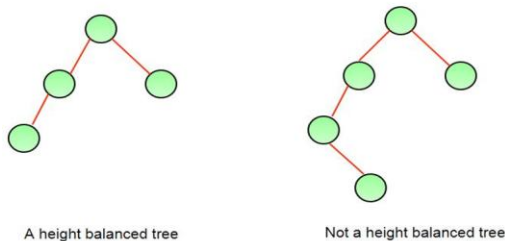
Generelt

- Hver node har 0-2 børn
- Følger formen: venstre barn < noden, højre barn > noden

Karakteristikker

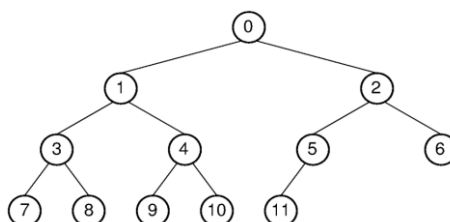
- Højde
 - Længste distance fra noden til blad
 - Optimal height:
 - $hmin = \lceil \log_2(n + 1) \rceil - 1$
- Dybde
 - Afstand fra noden til roden

Balanceret binært træ



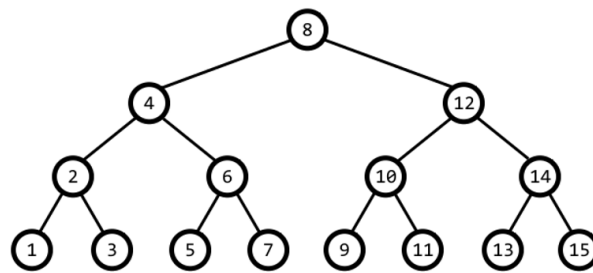
- Højdeforskellen mellem to noder må ikke være mere end 1

Komplet binært træ



- Alle niveauer skal være fyldt op fra venstre mod højre (men sidste niveau behøver ikke være færdigt)
- Er også et balanceret træ

Perfekt binært træ



- Alle niveauer skal være fyldt op fra venstre mod højre (skal være helt fyldt)
- Antallet af noder: $N = (2^{h+1}) - 1$
- Er også både et komplet og balanceret træ

Traversal

Summary

Summary Table

Traversal Type	Order	Category
Inorder	$L \rightarrow \text{Root} \rightarrow R$	DFS
Preorder	$\text{Root} \rightarrow L \rightarrow R$	DFS
Postorder	$L \rightarrow R \rightarrow \text{Root}$	DFS
Level-Order	Level by level	BFS
Morris	Inorder / Preorder	DFS ($O(1)$ space)

Inorder traversal code

```
static void InOrderTraversal(TreeNode node)
{
    if (node == null)
        return;

    InOrderTraversal(node.leftChild);
    Console.WriteLine(node.nodeValue);
    InOrderTraversal(node.rightChild);
}
```

Preorder traversal code

```
static void PreorderTraversal(TreeNode node)
{
    if (node == null)
        return;

    Console.WriteLine(node.nodeValue);
    PreorderTraversal(node.leftChild);
    PreorderTraversal(node.rightChild);
}
```

Postorder traversal code

```
static void PostorderTraversal(TreeNode node)
{
    if (node == null)
        return;

    PostorderTraversal(node.leftChild);
    PostorderTraversal(node.rightChild);
    Console.WriteLine(node.nodeValue);
}
```

Level-Order traversal code

```
static void levelOrderTraversal(TreeNode node)
{
    if (node == null)
        return;

    Queue<TreeNode> queue = new Queue<TreeNode>();
    queue.Enqueue(node);

    while (queue.Count > 0)
    {
        TreeNode active = queue.Dequeue();
        Console.WriteLine(active.nodeValue);

        if (active.leftChild != null)
        {
            queue.Enqueue(active.leftChild);
        }

        if (active.rightChild != null)
        {
            queue.Enqueue(active.rightChild);
        }
    }
}
```

Avl tree

- Et binært træ, som er balanceret efter balance factor
- Vil automatisk være et balanceret træ
 - $Balance\ factor = H_L - H_R$
 - Balance factor regnes med højden og IKKE antal børn
 - Balance factor må maxs være mellem -1 og 1
 - Ved større balance factor skal der laves rotation

Avl rotation

- Identificer den ubalancerede node med den højeste position i træet og roter derfra

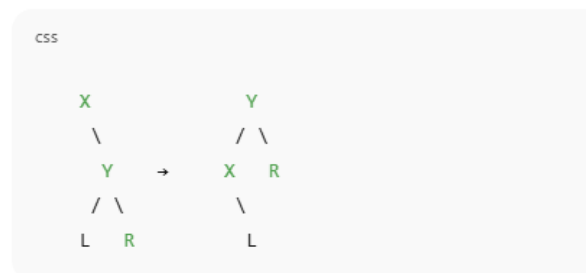
Left rotation (single rotation)

What a left rotation does (conceptually)

A **left rotation** is used when:

- A node is **right-heavy**
- Its right child becomes the new root of the subtree

For a node **x** with right child **y**:



Key rules:

- **y** replaces **x**
- **y.left** becomes **x.right**
- **x** becomes **y.left**



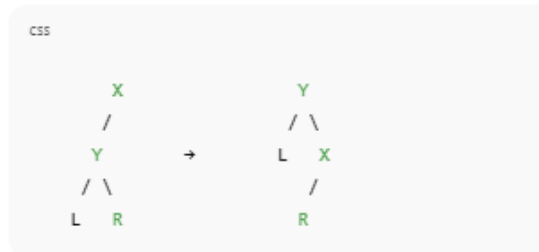
Right rotation (single rotation)

1 Right Rotation (conceptually)

A right rotation is used when:

- A node is left-heavy
- Its left child becomes the new root of the subtree

For a node X with left child Y:



Key rules:

- Y replaces X
 - Y.right becomes X.left
 - X becomes Y.right
-

Left - Right rotation (double rotation)

2 Left-Right (LR) Rotation (conceptually)

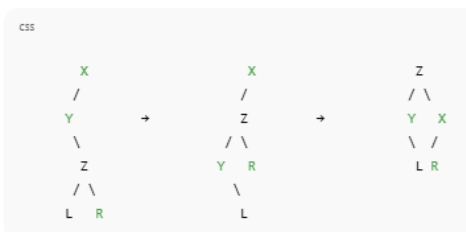
This is a double rotation:

1. Left rotation on the left child
2. Right rotation on the node

A Left-Right rotation is used when:

- A node is left-heavy
- Its left child is right-heavy

For a node X:



Key rules:

- First left-rotate Y
 - Then right-rotate X
 - Z becomes the new root of the subtree
-

Right - Left rotation (double rotation)

3 Right-Left (RL) Rotation (conceptually)

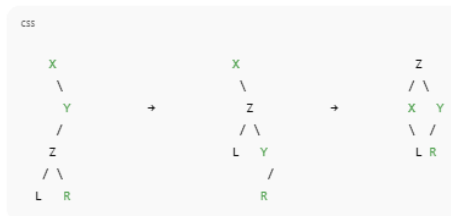
This is the mirror image of LR:

1. Right rotation on the right child
2. Left rotation on the node

A Right-Left rotation is used when:

- A node is right-heavy
- Its right child is left-heavy

For a node X:



Key rules:

- First right-rotate Y
- Then left-rotate X
- Z becomes the new root of the subtree

Binary Tree Code Examples

Opsætning af træ som klasse med konstruktor

```
class TreeNode //Node Class
{
    public int nodeValue;
    public TreeNode leftChild;
    public TreeNode rightChild;

    public TreeNode(int nodevalue) //Constructor for a node
    {
        nodeValue = nodevalue;
        leftChild = null;
        rightChild = null;
    }
}
```

Konstruktion af træet

```
public class BinaryTree
{
    private static TreeNode root;

    public static void Run()
    {
        root = new TreeNode(7);

        //root children
        root.leftChild = new TreeNode(4);
        root.rightChild = new TreeNode(28);

        //4 children
        root.leftChild.leftChild = new TreeNode(3);
        root.leftChild.rightChild = new TreeNode(20);

        //28 children
        root.rightChild.leftChild = new TreeNode(24);
        root.rightChild.rightChild = new TreeNode(55);

        //55 children
        root.rightChild.rightChild.leftChild = new TreeNode(51);
        root.rightChild.rightChild.rightChild = new TreeNode(60);
    }
}
```


Find route to node

```
static void findRoute(TreeNode root, int locate)
{
    if (root == null)
        return;

    if (root.nodeValue > locate)
    {
        Console.WriteLine(root.nodeValue);
        findRoute(root.leftChild, locate);
    }

    if (root.nodeValue < locate)
    {
        Console.WriteLine(root.nodeValue);
        findRoute(root.rightChild, locate);
    }

    if (root.nodeValue == locate)
    {
        Console.WriteLine(root.nodeValue);
    }
}
```

Check if node is a leaf

```
static bool isLeaf(TreeNode node)
{
    if (node == null)
        return false;

    if (node.leftChild == null && node.rightChild == null)
    {
        return true;
    }

    return false;
}
```

Find only child of node

```
static TreeNode getOnlyChild(TreeNode node)
{
    if (node == null)
        return null;

    if (node.leftChild != null && node.rightChild == null)
    {
        return node.leftChild;
    }
    if (node.leftChild == null && node.rightChild != null)
    {
        return node.rightChild;
    }
    return null;
}
```

Find number of branches

```
static int numberofbranch(TreeNode node)
{
    if (node == null)
    {
        return 0;
    }

    int branchnumber = 0;

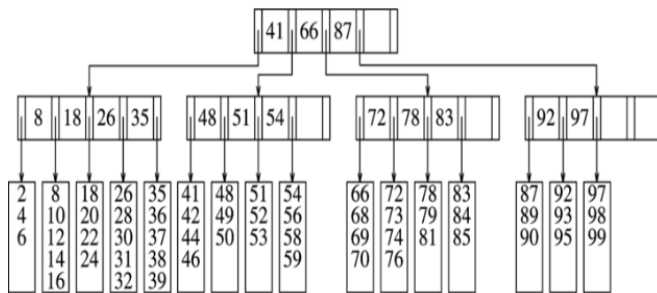
    if (getOnlyChild(node) != null)
    {
        TreeNode child = getOnlyChild(node);

        if (getOnlyChild(child) != null)
        {
            TreeNode grandChild = getOnlyChild(child);

            if (getOnlyChild(grandChild) != null)
            {
                TreeNode greatGransChild = getOnlyChild(grandChild);
                if (greatGransChild.leftChild == null &&
greatGransChild.rightChild == null)
                {
                    branchnumber++;
                }
            }
        }
    }

    return branchnumber + numberofbranch(node.leftChild) +
numberofbranch(node.rightChild);
}
```

5 ordens træ



- IKKE et binært træ

Graphs

Characteristics

Weight

- Vægt på edges

Direction

- Direction på edges

Cycles

- Er der loops mellem vertices
 - Cyclic = med loops
 - Acyclic = uden loops
- Der kan være loops i både directed og undirected graphs
- DAG – Directed Acyclic Graph

Density

- “No self-loops” betyder at en edge i må gå direkte tilbage til den node den lige er kommet fra (ikke noget med cyclic at gøre)

Formal definition

For a graph with:

- V vertices
- E edges

The maximum number of edges is:

- Undirected (no self-loops):

$$\frac{V(V-1)}{2}$$

- Directed (no self-loops):

$$V(V-1)$$

A graph is considered **dense** if:

$$E \approx \text{maximum possible edges}$$

And **sparse** if:

$$E \ll \text{maximum possible edges}$$

Connections

4. Directed Graph: Important distinction

For **directed graphs**, “connected” splits into different concepts:

(a) Strongly connected

For every pair u, v :

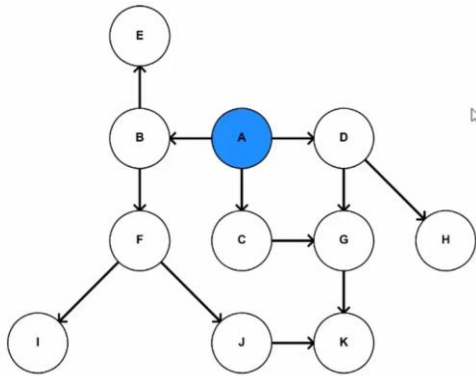
- $u \rightarrow v$ and
- $v \rightarrow u$ paths exist

(b) Weakly connected

If you **ignore edge directions**, the graph is connected

Topological Sort

- Krav
 - Acyclic
 - Directed
 - Måde både være vægtet og non-vægtet
- Anvendelse
 - Start ved en node som ikke er afhængig af andre noder
 - Gå til node som ikke har andre afhænginger af sig
 - Bevæg tilbage mod start noden og stack
- Brug
 - Prioritetskøer, hvilket element skal være kørt før andre kan køre



Dijkstras

- Krav
 - Directed
 - Weighted
- Anvendelse
 - Vælg en startnode, hvis ikke tildelt
 - Tag korteste vej og skriv previous vertice and samlet afstand fra startnode
 - Hvis kortere vej fra start til node findes opdater tabel
 - Opdag løbende noderne (Kender ikke alt på forhånd)
- Brug
 - Find den korteste vej til alle noder fra startnoden

<u>v</u>	<u>Known</u>	<u>dv</u>	<u>pv</u>
A	T	Inf	0
B			
C			
D			
E			
F			
G			
H			
I			
J			

Prim

- Krav
 - Undirected
 - Weighted
- Anvendelse
 - Vælg vilkårlig startnode
 - Tag laveste edgeværdi først
 - Besøg alle noder
 - Ingen loops
- Brug
 - Lav minimum spanning tree

Step	Node pair	Vægt
1		
2		
3		
4		
5		
6		
7		
8		
SUM	-	

Kruskal

- Krav
 - Undirected
 - Weighted
- Anvendelse
 - Start med laveste edge
 - Fortsæt med laveste edge
 - Stop når alle nodes er nået og alle components er connected
 - Ingen loops
- Brug
 - Lave minimum spanning tree

Step	Node pair	Vægt
1		
2		
3		
4		
5		
6		
7		
8		
SUM	-	

Prioritetskøer

Prioritetskø

- Behøver ikke være sorteret efter størrelse
- Index må ikke være gentaget
- Values må gerne være gentaget

Heaps

- Skal være sorteret efter størrelse
- Max heap
 - Størst(root) til mindst(leaf) – Value for en node er større end eller lig med value for dens børn
- Min heap
 - Mindst(root) til størst(leaf) - Value for en node er mindre end eller lig med value for dens børn

Time Complexity

Tommelfinger regler

- Halvering af problem $O(\log N)$
- En for-løkke: $O(N)$
- To for-løkker i sekvens: $O(N)$
- for-løkke med indb. halvering: $O(N \log N)$
- To nestede for-løkker: $O(N^2)$
- Tre nestede for-løkker: $O(N^3)$
- Tjek alle kombinationer: $O(2^N)$

- $O(N^{1,5}) = O(N^{3/2}) = O(N\sqrt{N})$
- $O(\sqrt{N}) = O(N^{1/2})$

Eksempler med kode

$O(1)$ – Konstant

```
for (int i = 0; i < 3; i++)  
{  
}
```

- For loop kører en konstant tid uafhængigt af N (her 3 gange)

$O(N)$

```
for (int i = 0; i < N; i++)  
{  
}
```

- For loop kører N gange

```
for (int i = 0; i < N - 3; i++)  
{  
}
```

- For loop kører N gange (konstanten -3 bliver ignoreret)

```
for (int i = 0; i < N * 3; i++)  
{  
}
```

- For loop kører N gange (konstanten *3 bliver ignoreret)

```
for (int i = 0; i < N / 3; i++)  
{  
}
```

- For loop kører N gange (konstanten /3 bliver ignoreret)

$O(\sqrt{N})$ || $O(N^{0,5})$

```
for (int i = 0; i < Math.sqrt(N); i++)  
{  
}
```

- For loop kører kvadratrods af N gange

$O(N^2)$

```
for (int i = 0; i < Math.pow(N, 2); i++)  
{  
}
```

- For loop kører N i anden potens gange

$O(\log N)$

```
for (int i = 0; i < Math.log(N); i++)  
{  
}
```

- For loop kører logaritisk N gange

```
for (int i = 0; i < N; i++)  
{  
    i *= 2;  
}
```

- $i *= 2$ vil dominere over N, så tidskompleksiteten vil være $\log(N)$

$O(I / J)$

```
for (int k = 1; k <= j; k += i) //  $O(j/i)$   
{  
    x++;  
}
```

Min og Maks Heap kode

- Min og maks heaps har samme tidskompleksitet

How to Calculate in Exams (Simple Rule)

Ask yourself:

- ◆ Does the operation move up or down the tree?
☑ Yes → $O(\log n)$
- ◆ Does it only look at the root?
☑ Yes → $O(1)$
- ◆ Does it scan elements?
☑ Yes → $O(n)$

Insert

1 Insert (Min-Heap / Max-Heap)

Steps:

1. Insert element at the end
2. Perform **heapify-up** (bubble up)

Worst case: element moves from leaf → root

Time complexity:

$O(\log n)$

Delete

❑ Delete (Extract Min / Extract Max)

- Min-Heap → delete minimum
- Max-Heap → delete maximum

Steps:

1. Remove root
2. Move last element to root
3. Perform **heapify-down**

Worst case: root moves to leaf

Time complexity:

$$O(\log n)$$

Peek

3 Peek (Get Min / Get Max)

- Just look at the root
- No restructuring

Time complexity:

$O(1)$

Build Heap

4 Build Heap (from an array)

This is a common exam trick ⚠️

✗ NOT $O(n \log n)$

✓ Correct complexity:

$O(n)$

Reason:

- Bottom-up heap construction
- Most nodes are near leaves (small height)

Search

5 Search (Find an element)

- Heap is **not** fully sorted
- You may have to check many nodes

Worst case:

$O(n)$

Quadratic Probing

- Start med en value som hasher til en plads X (key)
- Sæt value ind på plads X
 - Hvis X er optaget så prøv plads: **index + x % table size**

```
public static int calcCollisionQP(int collisionNumber, int tableSize, int hashIndex)
{
    int indexAfterQP = (hashIndex+collisionNumber^2) % hashIndex;

    Console.WriteLine(indexAfterQP);
    return indexAfterQP;
}
```

1^2	1
2^2	4
3^2	9
4^2	16
5^2	25
6^2	36
7^2	49
8^2	64
9^2	81
10^2	100

Load factor

- Antal elementer i tabellen
- Hvis load factor er mere end 0,5 skal tablet rehashes

```
public static int calcLoadfactor(int nubmerOfEntries, int tableSize)
{
    int loadFactor = nubmerOfEntries / tableSize;

    Console.WriteLine(loadFactor);
    return loadFactor;
}
```

Andet

ASCII – Værdier

- Tal har en “character value” kaldet ASCII
- For tal er ASCII: Tallet + 48

Funktioner

- Modulus %
 - $x \% y$ giver resten af x / y .
 - Eks. $4 \% 3 = 1$

Math.

- Math.Pow(x, y)
 - Tag potensen X af tallet Y
 - Eks: Math.Pow(2, 6) = 2^6
- Math.Log2(x)
 - Finder base 2 logaritmen til x. Aka. Hvad tal skal 2 opløftes i for at få x?
 - Ex. Math.Pow2(64) = 6 Fordi $2^6 = 64$
- Math.Round(x)
 - Afrunder til nærmeste int
 - C# afrunder selv mod 0, hvis man arbejder med ints.

Nummereret Alfabet

A	1
B	2
C	3
D	4
E	5
F	6
G	7
H	8
I	9
J	10
K	11
L	12
M	13
N	14
O	15
P	16
Q	17
R	18
S	19
T	20
U	21
V	22
W	23
X	24
Y	25
Z	26
Æ	27
Ø	28
Å	29

Ekstra kode

Majority Vote Algoritme

Oles kode:

```
int majorityElementNaive(int A[], int n)
{
    // check whether 'A[i]' is a majority element or not
    for (int i = 0; i <= n / 2; i++)
    {
        int count = 1;
        for (int j = i + 1; j < n; j++) {
            if (A[j] == A[i]) {
                count++;
            }
        }
        if (count > n / 2) {
            return A[i];
        }
    }
    return -1;
}
```

Vores opgave:

```
public static string Task1(string Text) //Majority
{
    Text = Text.ToLower();
    string[] words = Text.Split(new[] {',', ' ', '.'},
StringSplitOptions.RemoveEmptyEntries);
    string CommonWord = "";
    int highCount = 0;
    List<string> holdList = new List<string>();

    for(int i = 0; i < words.Length; i++)
    {
        if (!holdList.Contains(words[i]))
        {
            holdList.Add(words[i]);
            int count = 1;
            for(int j = i + 1; j < words.Length; j++)
            {
                if(words[i] == words[j])
                {
                    count++;
                }
            }

            if(count > highCount)
            {
                CommonWord = words[i];
                highCount = count;
            }
        }
    }

    //Time complexity = O(N^2)
    return CommonWord;
}
```

Sum of digits

```
public static int sumOfDigits(int n)
{
    int sum = 0;
    if (n == 0)
        return 0;

    return (sum + (n % 10)) + sumOfDigits((n / 10));
}
```

Find most occurrences

```
public static int Occurence(String word, char letter, int index) //Find
most occurrences
{
    int number = 0;
    if (index >= word.Length)
    {
        return 0;
    }

    if (word[index] == letter)
    {
        number++;
    }
    return number + Occurence(word, letter, index + 1);
}
```

Rotate first in array

```
public static void rotateFirst(int[] arr, int k) //First index to last k
times
{
    for (int i = 0; i < k; i++)
    {
        int first = arr[0];
        for (int j = 0; j < arr.Length - 1; j++)
        {
            arr[j] = arr[j + 1];
        }
        arr[arr.Length - 1] = first;
    }

    Console.WriteLine("Rotate First array: ");
    foreach (int i in arr)
    {
        Console.Write(i);
    }
}
```

Rotate last in array

```
public static void rotateLast(int[] arr, int k) //last index to first k  
times  
{  
    for (int i = 0; i < k; i++)  
    {  
        int last = arr[arr.Length - 1];  
        for (int j = arr.Length - 1; j > 0; j--)  
        {  
            arr[j] = arr[j - 1];  
        }  
        arr[0] = last;  
    }  
    Console.WriteLine("Rotate Last array: ");  
    foreach (int i in arr)  
    {  
        Console.Write(i);  
    }  
}
```