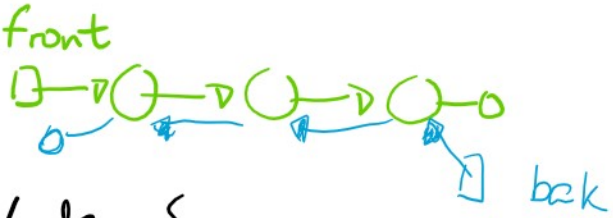
Lineært:



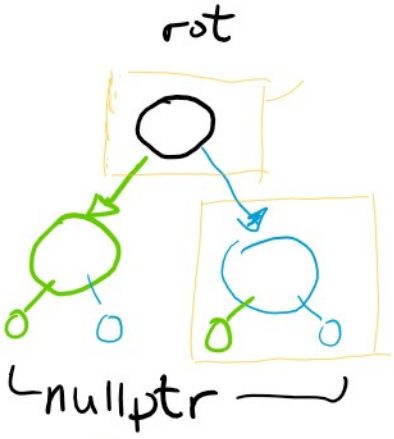

```
class CharNode {

    CharNode* m_neste;   // rekursiv struktur
    CharNode* m_forrige;

};
```

```
class BinaryTree {

    BinaryTree* m_left;
    BinaryTree* m_right;
};
```



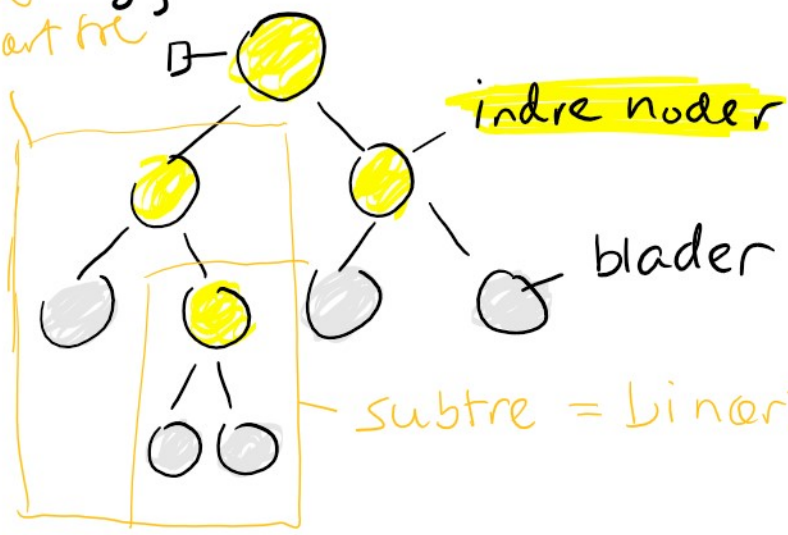
nivå: 0 $2^0 = 1$

1 $2^1 = 2$

2 $2^2 = 4$

3 $2^3 = 8$

subtre = binært tre



Søk lineært $O(n)$

Binært søk: kortere vei til blad
- hvis vi kan klare oss med en sammenligning

- hvis vi kan klare oss med en sammenligning på hvert nivå, blir det raskere søk.
Hvor mye raskere?

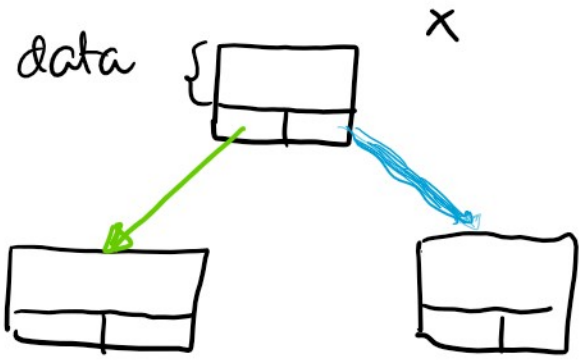
$\log_2 8 = 3 \quad (2^3 = 8)$

$\log_2 4 = 2$

Binært tre kan altså brukes til søking.

Binært søketre

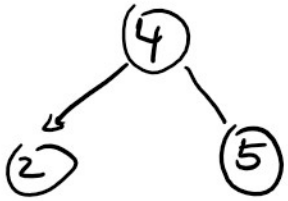
- krever: litt jobb når vi setter inn noder
- definisjon: For enhver node i treet gjelder:



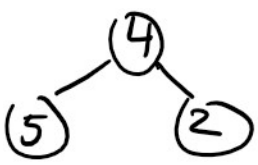
x → left → data < x → data

x → right → data ≥ x - data

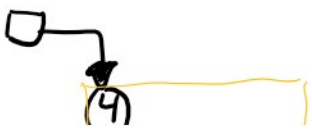
Kun > hvis vi ikke tillater duplikater



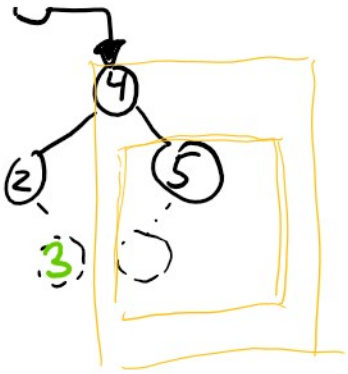
✓ binært søketre



✗ ikke binært søketre

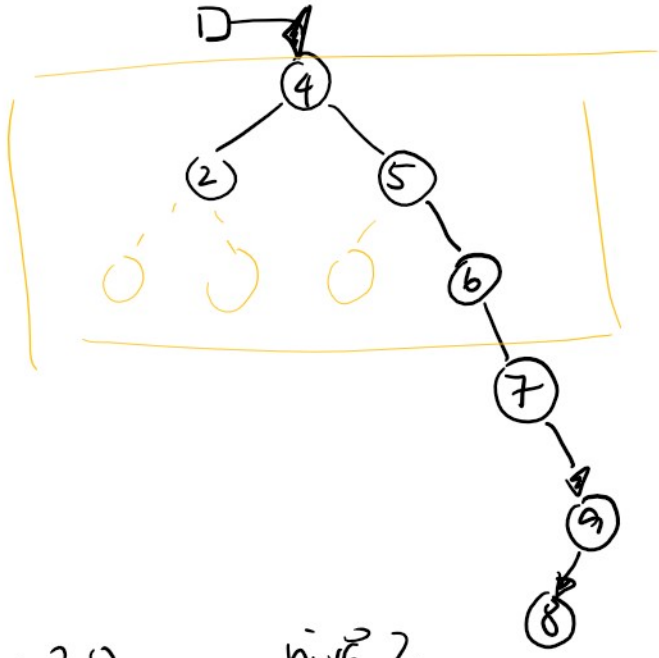


✓ Hvordan plassere en ny node



V Hvordan plassere en ny node 3 slik at vi fortsatt har et V binært søketre

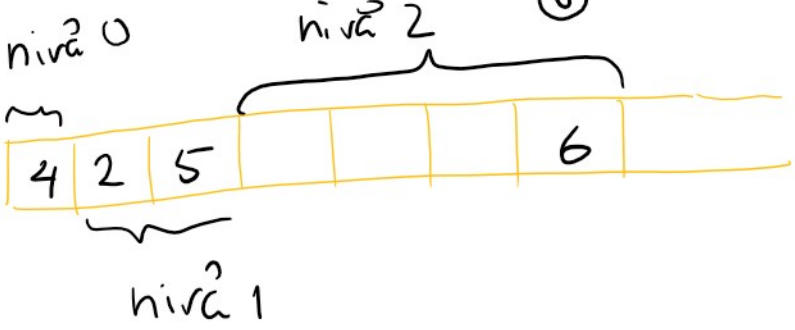
Sette inn 6, 7, 9, 8 i tur og orden. Tegn!



ganske sorterte innlete

⇓

ganske lineært



etc.

jfr. figur 4.1.

Søking, innsetting, traversering.

Gitt søkedata (heltall, char < >)

overlastes

Eksempel: Søke etter 8 på fig. 4.2

Eksempel: Søke etter 8 på fig. 4.2

Sml 8 med rot (5)

```
BinaryNode* node = rot;
data        node->m_data
if (data < node->data)
{
    node = node->m_left;
    //Sml på nytt
}
else // data >= node->data
{
    node = node->m_right;
    // Sml på nytt
}
```

rekursjon

m_left->find()

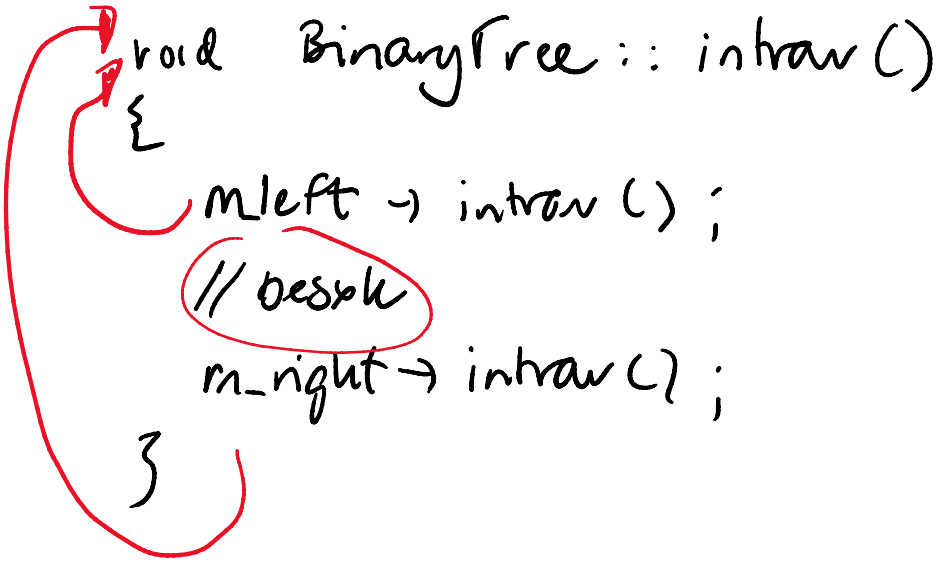
m_right->find()

```
BinaryNode* BinaryNode :: find (int data)
{
    // se s. 30
}
```

Binært søk er O(log n)

Traversering

- inorder
- preorder
- postorder



inorder traversering ⇒
stigende sortert rekkefølge

innsetting: Les/studer