UiT Norges arktiske universitet

DTE-2602
Introduksjon til maskinlæring og AI

Binære søketrær

Asbjørn Danielsen
Førsteamanuensis
Rom: D2260
Epost: asbjorn.danielsen@uit.no

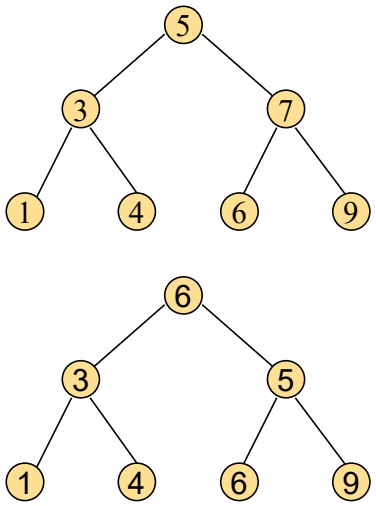
Gjennomgang av …

- Grunnleggende ideer bak binære søketrær
  - Datastrukturen, Operasjoner, og Algoritmer
  - Analysering av operasjonene
- Balanserte binære søketrær
  - AVL-Tre
  - Red-Black-Tre
- Balanserte (og avanserte) trær
  - AA-trær
  - B^+-Trær

Binære trær

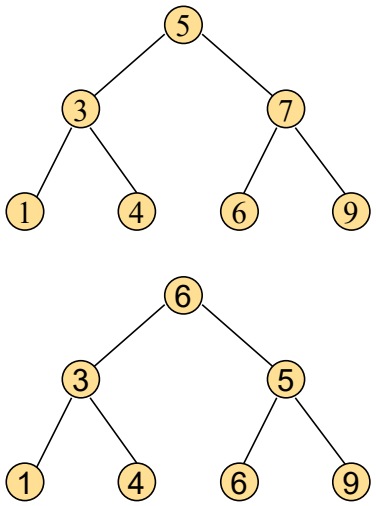
- Hvordan ser de ut?

  Prinsipp:
  - Hver node i treet har maksimalt to under-noder
  - Av denne grunn er treet binært
  - Vi skiller mellom binære trær og binære søketrær



Binære søketrær

- Definisjon: Binært søketre
  - Venstre node er alltid ***mindre*** enn mor-noden eller ikke definert
  - Høyre node er alltid ***større*** enn mor-noden eller ikke definert
  - Alle noder i venstre gren er mindre enn mor-noden og alle i høyre gren er større enn mor-noden
- Hvilket av de to trærne er et binært søketre?



Dette regnes som den ”formelle definisjonen”

Binære søketrær

- Konsekvens av definisjon:
  - Man må kunne identifisere om en node er mindre enn eller større enn en annen node
  - Binære søketrær tillater ikke duplikater (like verdier)
  - Nøkkel som identifserer verdien
  - Metoder for å sjekke verdier mot hverandre.

```
#
# Python har operator overloading
#

# self < other
def __lt__(self, other):

# self <= other
def __le__(self, other):

# self > other
def __gt__(self, other):

# self >= other
def __ge__(self, other):

# self != other
def __ne__(self, other):

# self == other
def __eq__(self, other):
```

Dette er en sannhet med modifikasjoner. Binære søketrær KAN HA duplikater, men de ivaretas da enten vha. en teller (for antall duplikater), eller i form av en lenket liste (med duplikater).

I kurset er det dog slik at vi antar at man IKKE HAR DUPLIKATER i et binært søketre. Dette av hensyn til enkeltheten.

Implementasjon (1)

- Behov for å finne ut om innholdet i en node sett opp mot en annen node

- Løsning:

Implementer operator overloading i Python

```
def __eq__(self, other):   # self == other
    if other != None:
        return self.value == other.value
    elif other == None and self.value == None:
        return True
    return False

def __ne__(self, other):   # self != other
    if other != None:
        if self.value == None:
            return False
        else:
            return not self.value != other.value
    return False

def __lt__(self, other):    #  self < other
    if other != None:
        return self.value < other.value
    elif other == None and self.value == None:
        return False
    return False

def __le__(self, other):    # self <= other
    if other != None:
        return self.value <= other.value
    elif other == None and self.value == None:
        return False
    return False

def __gt__(self, other):    # self > other
    if other != None:
        return self.value > other.value
    elif other == None and self.value == None:
        return False
    return False

def __ge__(self, other):    # self >= other
    if other != None:
        return self.value >= other.value
    elif other == None and self.value == None:
        return False
    return False
```

Vi skal nå over og se på noen mulige implementasjoner:

Ingen av implementasjonene som presenteres er lik de som finnes i læreboka.
Dog er det slik at funksjonaliteten er identisk lik.

Implementasjon (2) : BinaryTreeNode

```
class BinaryTreeNode:
    def __init__(self, value,
                 lefttree = None,
                 righttree = None):
        self.value = value
        self.left = lefttree
        self.right = righttree

    # Bruker setter/getters
    @property
    def value(self):
        return self.__value

    @value.setter
    def value(self, value):
        self.__value = value

    @property
    def left(self):
        return self.__left

    @left.setter
    def left(self, lefttree):
        self.__left = lefttree

    @property
    def right(self):
        return self.__right

    @right.setter
    def right(self, righttree):
        self.__right = righttree
```

```
    @right.setter
    def right(self, righttree):
        self.__right = righttree

    def __str__(self):
        return self.value

    def hasRight(self):
        if self.right == None:
            return False
        return True

    def hasLeft(self):
        if self.left == None:
            return False
        return True

    def prefixOrder(self):
        print(self, ' ', end = '')
        if self.hasLeft():
            self.left.prefixOrder()
        if self.hasRight():
            self.right.prefixOrder()

    def infixOrder(self):
        if self.hasLeft():
            self.left.infixOrder()
        print(self, ' ', end = '')
        if self.hasRight():
            self.right.infixOrder()

    def postfixOrder(self):
        if self.hasLeft():
            self.left.postfixOrder()
        if self.hasRight():
            self.right.postfixOrder()
        print(self, ' ', end = '')
```

```
    def levelOrder(self):
        from queue import SimpleQueue
        FIFOQueue = SimpleQueue()
        FIFOQueue.put(self)
        self.levelOrderEntry(FIFOQueue)
        while not FIFOQueue.empty():
            node = FIFOQueue.get()
            print(node, " ", end='')

    def levelOrderEntry(self, queue):
        if queue.empty():
            return
        node = queue.get()
        print(node, " ", end='')
        if node.hasLeft():
            queue.put(node.left)
        if node.hasRight():
            queue.put(node.right)
        if node.hasLeft() or node.hasRight:
            self.levelOrderEntry(queue)

    def __eq__(self, other):   # self == other
        if other != None:
            return self.value == other.value
        elif other == None and self.value == None:
            return True
        return False

    def __ne__(self, other):   # self != other
        if other != None:
            if self.value == None:
                return False
            else:
                return not self.value != other.value
        return False
```

```
    def __lt__(self, other):    #  self < other
        if other != None:
            return self.value < other.value
        elif other == None and self.value == None:
            return False
        return False

    def __le__(self, other):    # self <= other
        if other != None:
            return self.value <= other.value
        elif other == None and self.value == None:
            return False
        return False

    def __gt__(self, other):    # self > other
        if other != None:
            return self.value > other.value
        elif other == None and self.value == None:
            return False
        return False

    def __ge__(self, other):    # self >= other
        if other != None:
            return self.value >= other.value
        elif other == None and self.value == None:
            return False
        return False
```

Vi tar for oss en vilkårlig node i et tre og ser litt på hvordan denne ser ut …

Hver node har en verdi, og et venstre og et høyre subtree

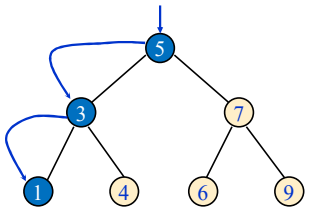
In-order traversering

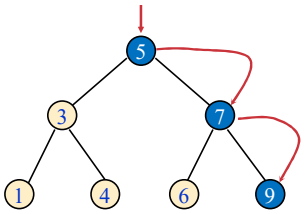
I dette eksempelet har jeg valgt å bruke getters og setters i Python via property- og setter-annoteringene.
Samtidig har jeg valgt å gjøre de indre delene av noden slik at de ikke kan endres på annen måte enn via getter-settere

Bør kommentere operatorene spesielt …

Noen enkle operasjoner/algoritmer

- Søk etter minste verdi i et tre - `findMin()`
  - Ta utgangspunkt i roten til treet
  - Følg venstre gren inntil venstre gren er `null`
  - Returnerer verdi
- Søk etter største verdi i et tre - `findMax()`
  - Ta utgangspunkt i roten til treet
  - Følg høyre gren inntil høyre gren er `null`
  - Returnerer verdi






```python
class BinaryTree:
    def __init__(self, data = None):
        self._root = None
        if isinstance(data, BinaryTreeNode):
            self._root = data

    def findLeftMost(self, treenode):
        left = treenode.left
        if left == None:
            return treenode
        return self.findLeftMost(left)

    def findMin(self):
        return self.findLeftMost(self._root)

    def findRightMost(self, treenode):
        right = treenode.right
        if right == None:
            return treenode
        return self.findRightMost(right)

    def findMax(self):
        return self.findRightMost(self._root)
```

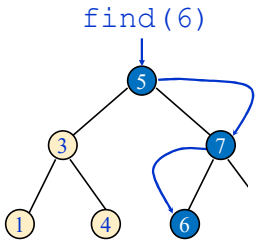
Vi ser nærmere på et Binært Tre – hva inneholder det

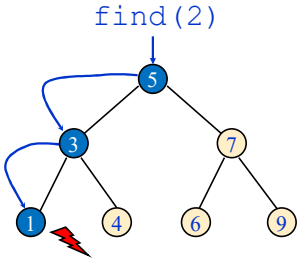
Her kunne man ha konstruert findMin() og findMax som egne metoder som

Implementasjon er ofte basert på rekursivitet, men dette behøver ikke være slik

Noen enkle operasjoner/algoritmer

- Finn en node - `find(N)`
  - Ta utgangspunkt i roten på treet
  - Leter etter noe som er mindre, traverser mot venstre
    - Hvis venstre node = = `None`, gi KeyError: «Key not found»
  - Leter etter noe som er større, traverser mot høyre
    - Hvis høyre node = = `None`, gi KeyError: «Key not found»
  - Hvis funnet returner Noden

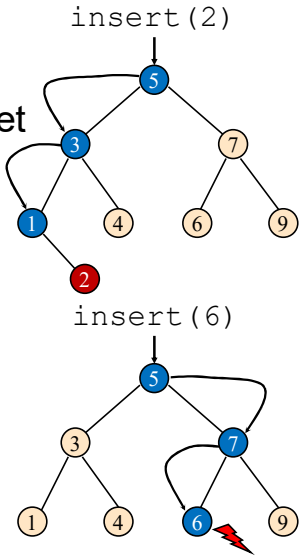





```
def find(self, key, treenode = None):
    if treenode == None:
        treenode = self._root
    if treenode.value > key:
        if treenode.left:
            return self.find(key, treenode.left)
    elif treenode.value < key:
        if treenode.right:
            return self.find(key, treenode.right)
    elif treenode.value == key:
        return treenode
    else:
        raise KeyError("Key not found")
```

Noen enkle operasjoner/algoritmer

- Legg inn ny node - `insert(N)`
  - Ta utgangspunkt i roten på treet
  - Hvis key < current, traverser mot venstre
    - Hvis venstre node = = `None`, legg inn ny node (N) og returner
  - Hvis key > current, traverser mot høyre
    - Hvis høyre node = = `None`, legg inn ny node (N) og returner
  - Hvis key == current, Exception: DuplicateKey



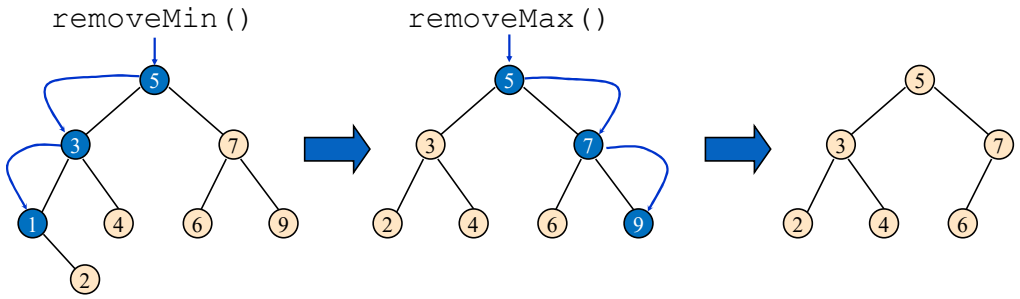

```python
def insert(self, current = None, treenode = None, value = None):
    if current == None:
        current = self._root
    # Checking consistency ...
    current, treenode = self._getnodes(current, treenode, value)
    if current != None:
        if treenode.value < current.value:
            if current.left is None:
                current.left = treenode
            else:
                self.insert(current.left, treenode)
        elif treenode.value > current.value:
            if current.right is None:
                current.right = treenode
            else:
                self.insert(current.right, treenode)
        else:
            if self._root == None:
                self._root = treenode
            else:
                raise Exception("Duplicate key")
    else: # If empty tree, the first node entered is the root
        self._root = treenode
```

Hva med sletting?

- Sletting er den mest komplekse av de vanlige, enkle operasjonene på binære trær.
- Vi kan tenke oss fire varianter
  - Sletting av den minste noden i treet - `removeMin()`
  - Sletting av den største noden i treet - `removeMax()`
  - Sletting av en node inne i treet - `remove(Node N)`
  - Sletting av roten i treet - (spesialtilfelle)

Kompleksitet

- `removeMin()` og `removeMax()` er ikke komplekse siden operasjonene kun påvirker noder som enten utgjør løv-noder eller noder som kun har ett barn



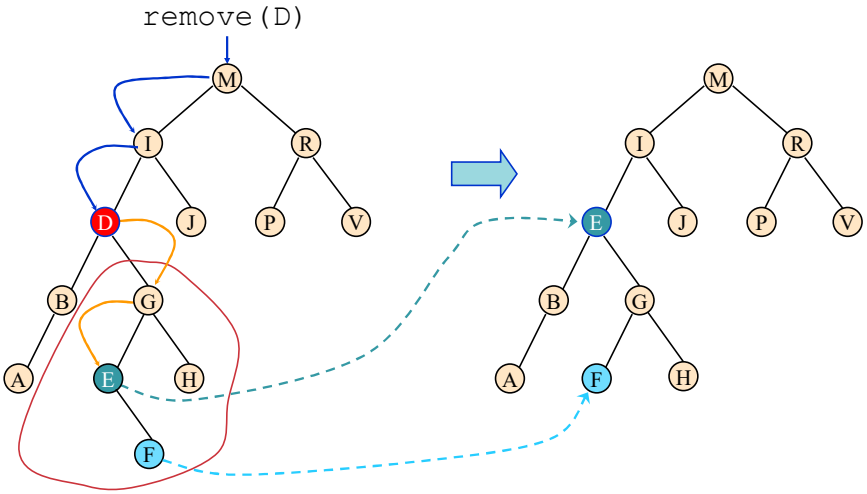
Vi får plutselig et behov for å holde orden på forrige node:

Hvis vi sletter node 1, så skal 1's høyre node (2), bli 1's mors venstre node

… og motsatt for removeMax

Kompleksitet

- Hva hvis vi sletter en node midt i treet?



Vanlig strategi: (OBS!! DETTE MÅ FORKLARES NØYE)

Vi tar noden som skal slettes (D), og følger dens ben ut til høyre (G). I denne noden (G) finner vi minste verdi (E). Denne noden (E) blir satt inn i treet i posisjonen til noden som skulle fjernes (D).

Noden (E) hadde ikke noe venstre barn (og vil aldri ha det siden den er minst i det aktuelle subtre), slik at det venstre subtre'et til noden som skulle slettes (D), dvs. (B), blir venstre subtre til (E).

Noden som flyttes (E), kan ha et høyre subtre (F), og denne noden rykker opp i (E)'s posisjon som følge av *deleteMin( )*.

Noden som slettes (D) forutsettets å ha et høyre subtre (G), og dette subtreet blir i så tilfelle høyre subtree av noden som erstatter (D), dvs. (E)

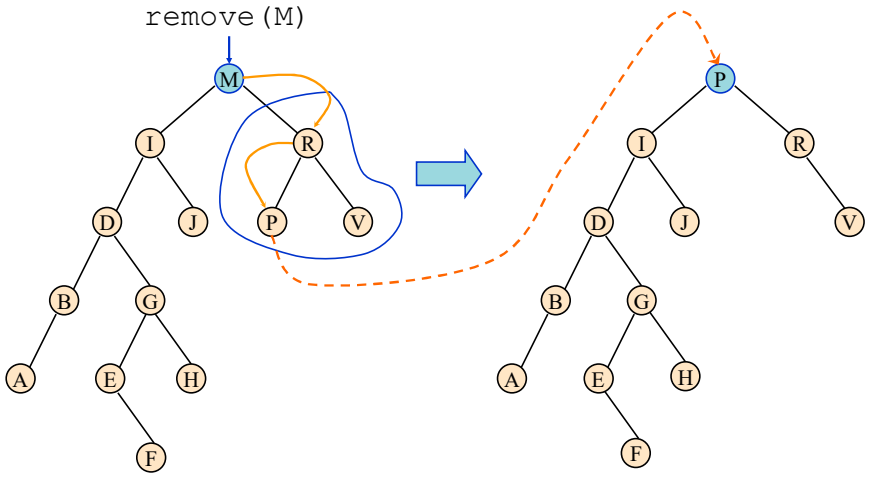
Hva skjer hvis noden som slettes er rota?

Spesialtilfelle: Hvis noden som slettes ikke har et høyre tre, blir det venstre tre som "rykker opp" til å erstatte noden som slettes.

Se strategien opp mot egenutviklet eksempelkode, og deretter opp mot boka.

Kompleksitet

- Hva hvis vi sletter roten?



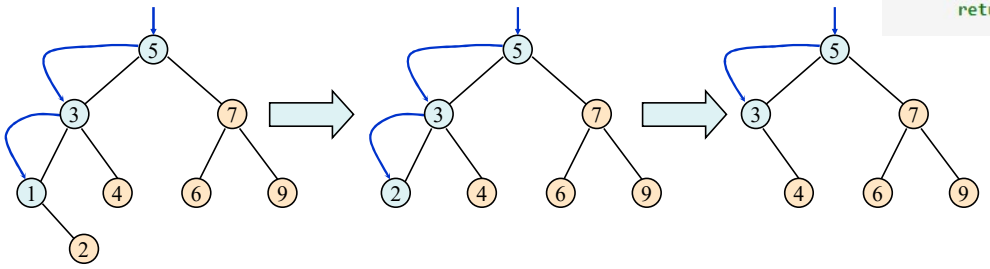
Oppskrift på tavla (se foil 17)

1. Finn Noden som skal slettes
2. Hvis noden ikke har barn, sett noden til ”null”
3. Hvis noden har en høyre gren:
    A. Hent ut minste node i den høyre grenen
    B. Sett noden som ble hentet ut i forrige punkt (A) inn i det sted hvor noden som skal slettes ligger (1).
    C. Den noden som ble flyttet i forrige punkt’s høyre ben rykker opp og erstatter noden som ble flyttet
4. Hvis noden som skal slettes kun har en venstre gren rykker denne opp.

Algoritmer for sletting av minste verdi

- Ta utgangspunkt i roten
  - Følg venstre gren inntil venstre gren er `None`
  - Hvis noden har en høyre gren != `None`, sett noden lik noden i høyre gren
  - Hvis noden ikke har en høyre gren, sett noden til `None`

```python
def deleteMin(self):
    if self._root.left == None:
        return self._deleteRoot()
    parent = self._root
    while True:
        current = parent.left
        if current.left == None:
            if current.right != None:
                parent.left = current.right
                break
            else:
                parent.left = None
                break
        else:
            parent = current
    return
```



Igjen rekursiv kode, må ses opp mot eksempelkoden:

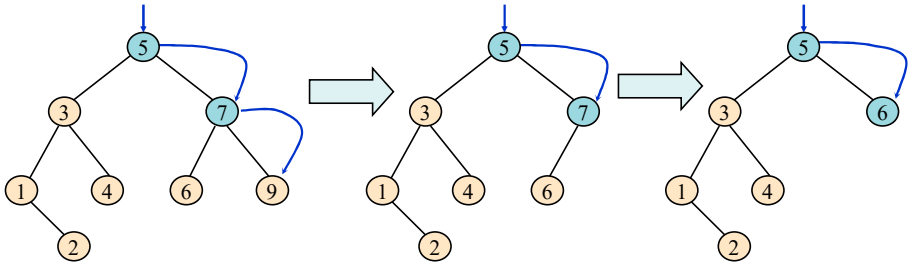
```
// Rekursiv implementasjon (i BinNode)
public BinNode removeMin(BinNode N) throws NotFoundException {
  if (N == null) throw new NotFoundException();
  else if (N.getLeft() != null)
    N.setLeft(removeMin(N.getLeft());
  else
    N = N.getRight();
  return N
}

// I søketreet, i tillegg:

public void removeMin() {
  // Siden root-noden kan endres
  root = root.removeMin(root);
}
```

Algoritmer for sletting av største verdi

- Ta utgangspunkt i roten
  - Følg høyre gren inntil høyre gren er `None`
  - Hvis noden har en venstre gren != `None`, sett noden lik noden i venstre gren
  - Hvis noden ikke har en venstre gren, sett noden til `None`




```python
def deleteMax(self):
    if self._root.left == None:
        return self._deleteRoot()
    parent = self._root
    while True:
        current = parent.right
        if current.right == None:
            if current.left != None:
                parent.right = current.left
                break
            else:
                parent.right = None
                break
        else:
            parent = current
    return
```

Igjen rekursiv kode, må ses opp mot eksempelkoden:

```
// Rekursiv implementasjon (i BinNode)
public BinNode removeMax(BinNode N) throws NotFoundException {
  if (N == null) throw new NotFoundException();
  else if (N.getRight() != null)
    N.setRight(removeMax(N.getRight());
  else
    N = N.getLeft();
  return N
}

// I søketreet, i tillegg:

public void removeMax() {
  // Siden root-noden kan endres
  root = root.removeMax(root);
}
```

Algoritmer for sletting - remove(Node N)

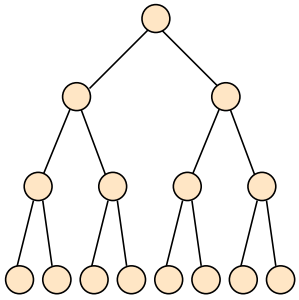
- `delete(N)`
  - Kjør algoritmen for `find(N)`
  - Hvis noden (N) ikke har barn, sett noden = `None`
  - Hvis noden har en høyre gren
    - Kjør `deleteMin()` i den høyre grenen til noden N
    - Sett innholdet i N (datene) lik den noden som ble slettet ved `deleteMin()`
  - Hvis noden kun har venstre gren, sett noden lik noden i venstre gren

```
def delete(self, key):
    node = self.find(key)
    if not node.left and not node.right:
        node = None
    elif node.right:
        temptree = BinaryTree(node.right)
        mintempnode = temptree.deleteMin()
        node.value = mintempnode.value
    elif node.left:
        node = node.left
```

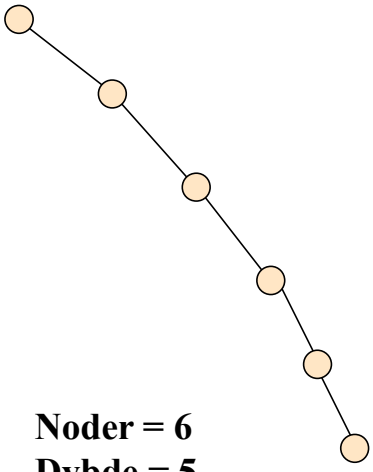
Analyse av operasjoner

- Kostanden for en rekke operasjoner (find, remove, insert) er nært knyttet til dybden til den siste noden som aksesseres.
- Kostnaden er logaritmisk for et (tilnærmet) balansert tre
- For et ubalansert tre kan kostnaden i værste fall være linær i forhold til antall noder i treet

Eksempler



Noder = 15
Dybde = 3
O(log N)



Noder = 6
Dybde = 5
O(N)

Hvordan ”måler” vi kostanden?

- Vi har altså to ytterligheter
  - *O(N)* for det maksimalt ubalanserte tre
  - *O(log N)* for 100% balansert tre
- Det store spørsmålet blir da:

  Er de fleste trær mer balansert enn ubalansert, eller finnes det et slags konstant gjennomsnitt?
- Det finnes et slags konstant gjennomsnitt
  - Gjennomsnittet er ca. 38% verre enn ”best-case”

Hvordan måler vi kostnaden?

- Konsekvensen av dette (mht. målinger) blir da:
- Treet må vite hvor mange noder som treet inneholder for å kunne beregne kostnaden
- Hver node må kjenne sin egen dybde i treet
- Den interne path-lengden i en binært tre er lik summen av dybden av alle noder

  $BS_{IPL} = \sum_{i=0}^{i=N-1} D(N_i)$ hvor $D(N_i)$ = Dybden i N_i

- BS_{IPL} brukes til å måle kostnaden ved et søk

BS_{IPL} = Binary SearchTree Internal Path Length

Eksempel

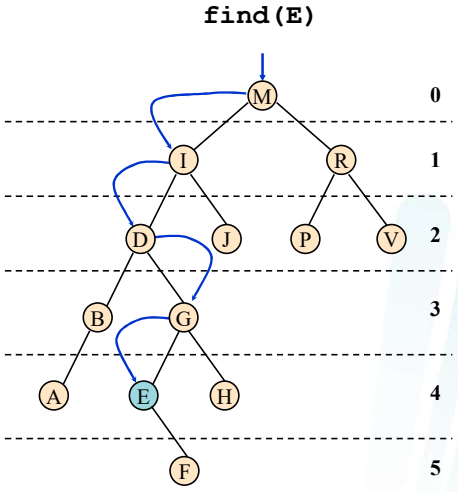
- Søk etter noden E

$$BS_{IPL} = \sum_{i=0}^{i=13-1} D(N_i)$$

$= D(N_M) + D(N_I) + D(N_R) +$
$D(N_D) + D(N_J) + D(N_P) + D(N_V) +$
$D(N_B) + D(N_G) +$
$D(N_A) + D(N_E) + D(N_H) +$
$D(N_F)$
$= 0 + 2 + 8 + 6 + 12 + 5 = 33$

E_{IPL} $= D(N_M) + D(N_I) + D(N_D) +$
$D(N_G) + D(N_E)$
$= 10$

- Gjennomsnittlig path-lengde
  $AVG_{IPL} = BS_{IPL}/N = 33/13 = 2.538$



BS_{IPL} = Binary SearchTree Internal Path Length

E_{IPL} = Noden E's Internal Path Length

AVG_{IPL} = Average Internal Path Length