aufgabe 16

gruppe Nofal
Nikolic
Hattinger

Counting sort ist stabil

Beweis :- (indirekt)

annahme Algorithmus ist nicht stabil

O.B.d.A:

$X = a = b$ Kommen an Position $i, j$ vor

also $a = A[i] \wedge b = A[j] \wedge (i < j)$

$\Rightarrow a = B[n] \wedge b = B[m] \wedge (m > n)$

Abarbeitung von Hinten [weil die schleife mit length[A] bignnt]

also $A[j]$ vor $A[i]$

$\Rightarrow P_j := C[A[j]] \Longrightarrow B[P_j] := b$

$C[P_j] := C[P_j] - 1$

$P_i := C[A[i]] = P_j - 1$

$P_j - 1 < P_j$

$\Rightarrow$ a ist vor b $\Rightarrow n > m$

widerspruch zur annahme.

$\Rightarrow$ Counting sort ist stabil

aufgabe 17

Nein es kann keinen vergleich Algorithmus geben der aus einem gegebnen array von n elementen einen binär suchbaum in zeit o(n log n) erzeugt

Beweis (indirekt)

annahme: ~~es~~ es gäbe so einen algorithmus mit o(n log n)

Nach def von o: heisst dass alle funktionen (g) g ~~w~~ müssen langsamer wachsen als n log n bzw (n log) ist eine obere schranke für solche funktionen aber gemäss der vorlesung besitzen alle vergleich funktionen $\Omega(n \log n)$ bzw (n log n) ist eine unterschranke für die vergleich funktionen

⟹ widerspruch zur annahme

⟹ es gibt keinen algorithmus (vergleich) mit o(n log n) aufgabe17

SortierAlgorithmus mit BinärsuchBaum

Tree-Sort(A)

for i ← 1 to n
    do Tree-insert(T, A[i])

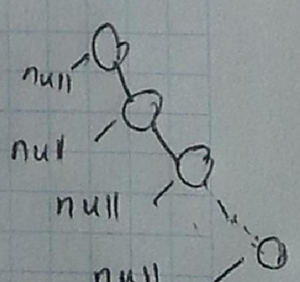
Inorder-Tree-Walk(root[T])

das bedeutet die array elemente von A werden in ein BinärsuchBaum hinzugefügt; danach werden die Knoten mit inorder-Tree-Walk besucht bzw Traversiert Beginned von root

#die laufzeit, da gibt es 2 falle

fall $\boxed{1}$ wenn der Baum Balanciert ist
einfügen von einem Knoten Kostet $O(\log(n))$
einfügen von "n" Knoten Kostet $O(n \log n)$

fall $\boxed{2}$ Wenn der Baum Ausbalanciert ist
Konnte der Baum so aussiehen
eigentlich wie ein Linked list
wei überall das Linke Kind = null




einfugen (und search auch) von einem Knoten Kostet $O(n)$
einfügen (und search auch) von einem "n" Knoten Kostet $O(n \cdot n) = O(n^2)$,
Rekursenz gleichung ist $T(n) = T(n-1) + Cn$.

aufgabe 18

Damit die geforderte Operation in O(1) ausgeführt werden kann, bekommt jede verzweigung einen zusätzlichen Parameter: int Weight

```
Weight(v)
if v = nil
      return 0
else
      return weight[v]

insert(T, z)
  y ← nil
  x ← root[T]
  while x ≠ nil do
        y ← x
        if Key[z] < Key[x]
              x ← leftchild[x]
        else
              x ← rightchild[x]
  p[z] ← y
  if y = nil
        root[T] = z
  else
        if Key[z] < Key[y]
              leftchild[y] = z
              Weight[z] = 1
        else
              rightchild[y] ← z
              Weight[z] = 1
```

```
20- while y ≠ nil do
21-       weight[y] ← (rightchild[y] + 1)
22-       if weight[y] ≤ leftchild[y]
23-            weight[y] ← (leftchild[y] + 1)
24-       y ← p[y]
```

in zeile "13" wurde ein Algorithmus mit $\Theta(\log n)$ hinzugefügt, damit die gesamte laufzeit auf das doppelte $2 * \Theta(\log(n))$, wobei die eigentliche laufzeit noch in $\Theta(\log(n))$ liegt

```
delete(T, z)
if (leftchild[z] = nil or rightchild[z] = nil)
        y ← z
else
        x ← searchnext(z)
if lc[y] ≠ nil
        x ← leftchild[y]
else
        x ← rightchild[y]
if x ≠ nil
        p[x] = p[y]
        if [y] = nil
              root[T] ← x
        else if  y = leftchild[p[y]]
                 leftchild[p[y]] ← x
        else
                 rightchild[p[y]] ← x
if y ≠ z
```

Key[z] ← Key[y]
Weight[z] ← (rightchild[z]+1)

if weight(z) ≤ leftchild[z]
weight[z] ← (leftchild[z]+1)
return y

die operation search ist nicht betroffen, da sie nichts an der Baumstruktur ändert.

Bei der operation insert muss bei jedem passierten Knoten weight um 1 erhöht werden. Bei der operation delete muss, falls der Knoten im Baum ist, bei jedem Knoten auf dem Weg zu dem zu löschenden Knoten weight um eines verringert werden.
da jeder Knoten bei dem weight nun modifiziert werden muss bei diesen beiden operationen. bisher sowieso schon besucht wird ~~er~~ und das modifizieren selber in O(1) geht, ändern sich die laufzeiten dieser operationen nicht. operation search ist überhaupt nicht betroffen da sie nichts an der Baumstruktur ändert. Das letzte was überprüft werden muss, sind die rotationen, diese geschehen bisher in O(1), was sich nicht ändert, auch wenn hier weight modifiziert werden muss, da pro Ratation nur 2 Additionen durchgeführt werden müssen, was in Konstanter zeit O(1) geschehen Kann. die operation Baumgrösse(x) ist

Baumgroesse(x)
return Weight[x]