**Serie 5**

Theoretische Aufgaben

**1. Reihenfolge umkehren**

Nicht rekursiv, Zeit Θ(n), einfach verkettet, n Elemente

LIST-REVERSE(L)

point := NIL //head wird zu tail werden und auf NIL zeigen current := L.head //aktuelle Stelle, Beginn bei Head

WHILE current != nil

next := current.point //Zwischenspeicher Zeiger von akt. El.

current.point := point //Zeiger überschrieben (1. NIL)

point := current // Zeiger von nächstem El. (akt. El.)

current := next // auf nächstes Element

L.head = point //Kopf = letztes El. Auf das gezeigt wurde

**2. Warteschlange**

e.next ist ein Element oder NIL, e.key ist ein Wert

Nil.next und NIL.key geht nicht

Enqueue(Queue Q, Element e)

e.next := NIL //neues Element zeigt auf NIL

If Q.head = NIL //e ist erstes Element

Q.head := e

Else

Q.tail.next := e //aktueller tail zeigt auf e

Q.tail := e //e wird tail

Dequeue(Queue Q)

If Q.head = NIL //Q.head.key geht nicht wenn Q.head = NIL

Return NIL

ELSE //Queue nicht leer

e := Q.head.key //Zwischenspeicher Ausgabe (e kein Element)

If Q.head.next = NIL //letztes Element

Q.tail := NIL //tail = NIL

Q.head = Q.head.next //Element oder NIL

return e

**Ein Bild, das Text, Tisch enthält.

Automatisch generierte Beschreibung**

**3. Schlüssel aller Knoten eines gerichteten Baums: rekursiv**

GET-KEYS(n)

IF n == NIL

RETURN

ELSE

PRINT(n.key)

GET-KEYS (n.left-child)

GET-KEYS (n.right-sibling)

**4. Schlüssel aller Knoten eines gerichteten Baums: nicht rekursiv**

S = new Stack();

root = Tree.root

S.push(root)

TRAVERSE\_TREE(Node: root)

while S is not empty

current = Node

while current is not null

S.push(Node.left\_child)

current = current.left\_child

end

last\_node = S.pop()

Print(last\_node.key)

end

**5. Merge**

LIST\_MERGE(Node A, Node B)

Node first = New Node;

Node list\_end = first;

While(true)

If(A == NULL)

List\_end.next = B;

Break;

If(B == NULL)

list\_end.next = A;

break;

if(A.key <= B.key)

list\_end.next = A;

A = A.next;

else

tail.next = B;

B = B.next;

List\_end = list\_end.next;

Return first.next;

**5. Merge**

Seien L (left) und R (right) die zwei Listen die zusammengefügt (und sortiert) in die neue Liste M geschrieben werden.

MERGE (L, R, M)

M.tail := NIL

WHILE L.head != NIL | R.head != NIL

IF (L.head.key <= R.head.key) | (R.head = NIL && L.head != NIL)

M.tail := L.head

M.tail := M.tail.next

L.head := L.head.next

ELSE

M.tail := R.head

M.tail := M.tail.next

R.head := R.head.next

M.tail.next := NIL

Return M

Wenn M auch zyklisch sein muss wird es noch etwas komplizierter:

MERGE (L, R, M)

M.head := NIL

IF L.head != NIL | R.head != NIL

IF (L.head.key <= R.head.key) | (R.head = NIL && L.head != NIL)

M.head := L.head

L.head := L.head.next

ELSE

M.head := R.head

R.head := R.head.next

M.tail := M.head.next

WHILE L.head != NIL | R.head != NIL

IF (L.head.key <= R.head.key) | (R.head = NIL && L.head != NIL)

M.tail := L.head

M.tail := M.tail.next

L.head := L.head.next

ELSE

M.tail := R.head

M.tail := M.tail.next

R.head := R.head.next

M.tail.next := M.head //zyklisch

Return M