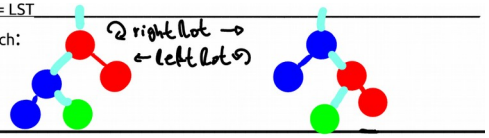
RBT LST = left subtree, isRchild(n): $n = n \to p \to r$

RightRotate(node):
parent = node→p
LST = node→l
node→l = LST→r
LST→p = parent
if (parent == NIL) { root = LST }
else
  if (isRChild(node)) { parent→r = LST }
  else { parent→l = LST }
LST→r = node
parent = LST

Graphisch:



Insert

RB-INSERT(T, z)

```
1  y = T.nil
2  x = T.root
3  while x ≠ T.nil
4        y = x
5        if z.key < x.key
6              x = x.left
7        else x = x.right
8  z.p = y
9  if y == T.nil
10       T.root = z
11 elseif z.key < y.key
12       y.left = z
13 else y.right = z
14 z.left = T.nil
15 z.right = T.nil
16 z.color = RED
17 RB-INSERT-FIXUP(T, z)
```

1) "BST"-Insert
2) color node RED
3) fixup
uncle is in opposite tree half...

case 0: node is root → color it BLACK

case 1: uncle's RED
→ parent.col = uncle.col = BLACK
grandpa.col = red [propagate coloring uppards, g might now violate RBT property]

case 2 (3eck): uncle's BLACK, node is right child
→ Lrot(parent)
→ now ge get case 3

case 3 (Linie): uncle's BLACK, node is left child
→ parent = BLACK
grandpa = RED
Rrot(grandpa)
→ tree is now balanced

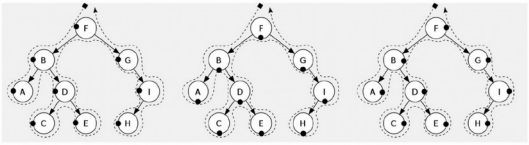
RB-INSERT-FIXUP(T, z)

```
1  while z.p.color == RED
2        if z.p == z.p.p.left
3              y = z.p.p.right
4              if y.color == RED
5                    z.p.color = BLACK                  // case 1
6                    y.color = BLACK                    // case 1
7                    z.p.p.color = RED                  // case 1
8                    z = z.p.p                          // case 1
9              else if z == z.p.right
10                   z = z.p                            // case 2
11                   LEFT-ROTATE(T, z)                  // case 2
12             z.p.color = BLACK                        // case 3
13             z.p.p.color = RED                        // case 3
14             RIGHT-ROTATE(T, z.p.p)                   // case 3
15       else (same as then clause
               with "right" and "left" exchanged)
16 T.root.color = BLACK
```

RB-TRANSPLANT(T, u, v)

```
1  if u.p == T.nil
2        T.root = v
3  elseif u == u.p.left
4        u.p.left = v
5  else u.p.right = v
6  v.p = u.p
```

Tree-Traversals:
inorder: leftroot-right [print]
preorder: rootleft-right [root/l/l/l...r/r, copy tree]
postorder: leftright-root [delete tree]



Delete

RB-DELETE(T, z)

```
1  y = z
2  y-original-color = y.color
3  if z.left == T.nil
4        x = z.right
5        RB-TRANSPLANT(T, z, z.right)
6  elseif z.right == T.nil
7        x = z.left
8        RB-TRANSPLANT(T, z, z.left)
9  else y = TREE-MINIMUM(z.right)
10       y-original-color = y.color
11       x = y.right
12       if y.p == z
13             x.p = y
14       else RB-TRANSPLANT(T, y, y.right)
15             y.right = z.right
16             y.right.p = y
17       RB-TRANSPLANT(T, z, y)
18       y.left = z.left
19       y.left.p = y
20       y.color = z.color
21 if y-original-color == BLACK
22       RB-DELETE-FIXUP(T, x)
```

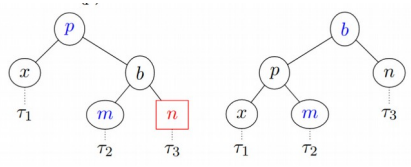
case 1: bro's RED
→ bro = BLACk
parent = RED
Lrot(parent)
→ we get case 2/3 or 4

case 2: bro's BLACK, both children of node are BLACK
→ bro = RED
set node = parent
if (parent=RED) parent = black

case 3: bro's BLACK, leftChild of bro is RED, rightChild of bro is BLACK
→ bro→l = BLACK
bro = RED
Rrot(bro)
set bro = bro→l
→ we get case 4

case 4: bro's BLACK, rightChild of bro is RED
→ bro.col = parent.col
parent.col = bro→r.col = BLACK
→ tree is now balanced

case 4 illustrated:



RB-DELETE-FIXUP(T, x)

```
1  while x ≠ T.root and x.color == BLACK
2        if x == x.p.left
3              w = x.p.right
4              if w.color == RED
5                    w.color = BLACK                    // case 1
6                    x.p.color = RED                    // case 1
7                    LEFT-ROTATE(T, x.p)                // case 1
8                    w = x.p.right                      // case 1
9              if w.left.color == BLACK and w.right.color == BLACK
10                   w.color = RED                      // case 2
11                   x = x.p                            // case 2
12             else if w.right.color == BLACK
13                         w.left.color = BLACK         // case 3
14                   w.color = RED                      // case 3
15                   RIGHT-ROTATE(T, w)                 // case 3
16                   w = x.p.right                      // case 3
17             w.color = x.p.color                      // case 4
18             x.p.color = BLACK                        // case 4
19             w.right.color = BLACK                    // case 4
20             LEFT-ROTATE(T, x.p)                      // case 4
21             x = T.root                               // case 4
22       else (same as then clause with "right" and "left" exchanged)
23 x.color = BLACK
```

Heap → O(nlog(n))

→issa complete tree: fill up from left to right (MaxHeap: parent > children)

heapify(A,n,i):
max = l;lChild = 2i+1;rChild = 2i+2
if (A[l] > A[max] && l<n) **then:** { max = l }
if (A[r] > A[max] && r<n) **then:** { max = r }
if (max != l) **then:**
  swap(A[i], A[max])
  heapify(A,n,max)

builMaxHeap(A,n): for(i=n/2; i>=0; i—) do: { heapify(A,n,i) }

HEAPSORT(A)

```
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3        exchange A[1] with A[i]
4        A.heap-size = A.heap-size − 1
5        MAX-HEAPIFY(A, 1)
```

BST

print in 'reversed order' → O(n)
printTree(root):
if root != NIL then:
  printTree(root→right) // print biggest values
  print(root→key)
  printTree(root→left) // print other values

sibling(idx):
if (idx == root) return NULL
if (idx%2 = 1) { return idx-1 }
if (idx=n) { return -1}
else{ return idx+1 }

TREE-SUCCESSOR(x)

```
1  if x.right ≠ NIL
2        return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5        x = y
6        y = y.p
7  return y
```

Insertion

Tree-Insert(T, node):
back = NIL; temp = T→root
// find correct place to insert new node
while (temp != NIL) do:
  back = temp
  if (node→key < temp→key): { x = x→left }
  else { x = x→right }
// insert node
node→parent = back
if (back == NIL): { T→root = node } // tree was empty
else if (node→key < back→key):
{ back→left = node }
else { back→right = node }

Deletion

TREE-DELETE(T, z)

```
1  if z.left == NIL
2        TRANSPLANT(T, z, z.right)
3  elseif z.right == NIL
4        TRANSPLANT(T, z, z.left)
5  else y = TREE-MINIMUM(z.right)
6        if y.p ≠ z
7              TRANSPLANT(T, y, y.right)
8              y.right = z.right
9              y.right.p = y
10       TRANSPLANT(T, z, y)
11       y.left = z.left
12       y.left.p = y
```

Graphs

Kante = Paar v. Knoten | *gerichteter* Graph: Kante (u,v) [!= {u,v}]
vollständiger G.: Jeder Knoten hat ne Verbindung zu allen anderen Knoten

DFS →Θ(|V| + |E|)

DFS(G)

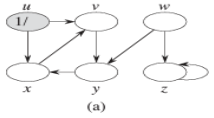
```
1  for each vertex u ∈ G.V
2        u.color = WHITE
3        u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6        if u.color == WHITE
7              DFS-VISIT(G, u)
```

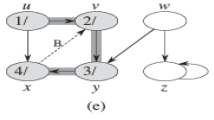
DFS-VISIT(G, u)

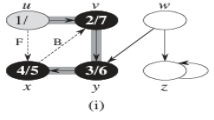
```
1  time = time + 1                  // white vertex u has just been discovered
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]            // explore edge (u, v)
5        if v.color == WHITE
6              v.π = u
7              DFS-VISIT(G, v)
8  u.color = BLACK                  // blacken u; it is finished
9  time = time + 1
10 u.f = time
```

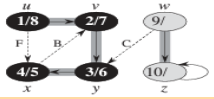
TRANSPLANT(T, u, v)

```
1  if u.p == NIL
2        T.root = v
3  elseif u == u.p.left
4        u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7        v.p = u.p
```









→Alle Knoten in G werden besucht

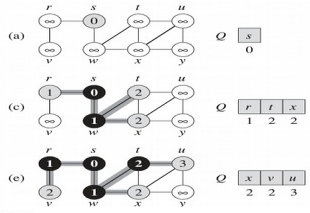
DFS-Visit(G,v):
// markier v als besucht
for each (v,w) € E(dges) do:
  if ("w noch nicht besucht") then:
  DFS-Visit(G,w)

BFS → good for shortest path, social networks →vlt nicht alle Knoten besucht

BFS(G, s)

```
1  for each vertex u ∈ G.V − {s}
2        u.color = WHITE
3        u.d = ∞
4        u.π = NIL
5  s.color = GRAY
6  s.d = 0
7  s.π = NIL
8  Q = ∅
9  ENQUEUE(Q, s)
10 while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each v ∈ G.Adj[u]
13             if v.color == WHITE
14                   v.color = GRAY
15                   v.d = u.d + 1
16                   v.π = u
17                   ENQUEUE(Q, v)
18       u.color = BLACK
```

1) init nodes (ex make'em white)
2) init ADT (ex init(Q))
3) bearbeite Knoten in ADT



The following procedure prints out the vertices on a shortest path from s to v, assuming that BFS has already computed a breadth-first tree:

PRINT-PATH(G, s, v)

```
1  if v == s
2        print s
3  elseif v.π == NIL
4        print "no path from" s "to" v "exi
5  else PRINT-PATH(G, s, v.π)
6        print v
```

Other Graph algos.

TOPOLOGICAL-SORT(G)

```
1  call DFS(G) to compute finishing times v.f for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

GENERIC-MST(G, w)

```
1  A = ∅
2  while A does not form a spanning tree
3        find an edge (u, v) that is safe for A
4        A = A ∪ {(u, v)}
5  return A
```

Topo-s: gut um zu wissen, **in welcher Reihengole Aufgaben abgearbeitet** werden müssen
Bsp: kleider anziehen
Shirt→Tie→Jacket
Put in LL:
←Jacket←Tiey-Shirt
LL: 1.)Shirt2.)Tie 3.) Jacket

DAG-SHORTEST-PATHS(G, w, s)

```
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4        for each vertex v ∈ G.Adj[u]
5              RELAX(u, v, w)
```

MST-PRIM(G, w, r)

```
1  for each u ∈ G.V
2        u.key = ∞
3        u.π = NIL
4  r.key = 0
5  Q = G.V
6  while Q ≠ ∅
7        u = EXTRACT-MIN(Q)
8        for each v ∈ G.Adj[u]
9              if v ∈ Q and w(u, v) < v.key
10                   v.π = u
11                   v.key = w(u, v)
```

DIJKSTRA(G, w, s)

```
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5        u = EXTRACT-MIN(Q)
6        S = S ∪ {u}
7        for each vertex v ∈ G.Adj[u]
8              RELAX(u, v, w)
```

INITIALIZE-SINGLE-SOURCE(G, s)

```
1  for i = 1 to |G.V| − 1
2        for each edge (u, v) ∈ G.E
3              RELAX(u, v, w)
4  for each edge (u, v) ∈ G.E
5        if v.d > u.d + w(u, v)
6              return FALSE
7  return TRUE
```

Adj. Matrix: Θ(|V|²)
Adj. List: Θ(|V| + |E|)

Recurrences

Master-Meth: T(n) = aT(n/b) + f(n) [n^(log_b(a) = x]
case 1: f(n) < x → O(x)
case 2: = → O(x*log(n))
case 3: f(n) > x → =(f(n)

Binary search(A, l, r): → O(log(n)), da Baum..! | worst/insert/del: O(n)

```
if ( l < r): return NIL
mid = (low + high)
if(A[mid] > value): return BS(A, value, l, mid-1)
else if(A[mid] < value): return BS(A, value, mid+1, r)
else: return mid
```

Quicksort →O(nlg(n)) | worst O(n²)

```
QS(Q, p, r):
if( p < r ) then:
  q = partition(A, p, r)
  QS(A, p, q-1)   // linke partition
  QS(A, q+1, p)
```

```
Hoarce(A, p, r):
i = p-1; j = r+1; p = A[l]
while(true) do:
  while(A[j] > A[p]) {j--}
  while(A[i] < A[p]) {i++}
  if( l < j ) { swap(A[j], A[i]) }
  else { return j } // position of new pivot
```

```
Lomuto(A, p, r):
for( j=l to r-1) do:
  if( A[j] <= p) then:
    i++
    swap(A[i], A[j])
  swap(A[i+1], A[r])
return i+1
```

Mergesort(): →O(nlg(n))

```
if l<r then
    m = ⌊(l+r)/2⌋;
    MergeSort(A,l,m);
    MergeSort(A,m+1,r);
    Merge(A,l,r,m);
```

Merge():

```
for i = l to m do B[i] = A[i];
for i = m+1 to r do B[r+m-i+1] = A[i];
i = l; j= r;
for k = l to r do
    if B[i]<B[j] then A[k] = B[i]; i = i+1;
    else A[k] = B[j]; j = j-1;
```

DP

Vorgehen:
1) **Tabelle** aufstellen
-Dimensionen? Bedeutung eines Slots?
2) Wie berechnet man nen **Slot**?
3) Berechnungs-**Reihenfolge**
welche Einträge müssen vorhanden sein für current slot?
4) wie **Lösung** aus Tab. Herauslesen?

Matrix

Für 1 slot - FORMEL:
  cost = m[i,k]+m[k+1,j]+dim_(i-1)*dim_k*dim_j
I = leftmost index, j = rightmost index, k = inbetween index of parenthesis

MatrixChainDP(d): → O(n³) ->3x for loops

```
from i=1 to n do: m[i,i] = 0      //init Diagonale zu 0
from "currLen"=1 to n (tot.length) do:
  from left=1 to currLen do:
    right = left + 1
    m[l,r] = 'infinity'
    from k=left to right do:
      cost = FORMEL
      if (cost < m[l,r] ) then: // if cheaper version found
        m[l,r] = cost
        klammern[l,r] = k // Klammer setzen bei k
return (m(Konstentabelle), klammern)
```

Longest common increasing subsequence

Algo: D_PROG(A[1..n])

```
for i = 1 to n do
    S[i] = 1;
    for j = 1 to i − 1 do
        if A[i] > 2A[j] and S[i] < S[j] + 1 then
            S[i] = S[j] + 1;
return max_{0≤i≤n}(S[i]);
```

Fibonacci

base case: f_1=1, f_2=1 →if (n<=2) return 1
Formel: f_n = f_(n-1) + f_(n-2) → f_n = fibo(n-1) + fibo(n-2)

```
fiboRec(n):
if (n<=2): return1
return fibo(n-1)
+ fibo(n-2)
```

```
fiboMemo(n):
[top-down]
// check if
already stored
if (memo[n] !=
NIL): return
memo[n]
// suscht
berächne
if (n<=2):
memo[n] = 1
else: memo[n] =
fiboMemo(n-1) +
fiboMemo(n-2)
return memo[n]
```

```
fiboMemo(n):
[bottom-up]
memo[1] = 1;
memo[2] = 1;
from i=3 to n do:
memo[i] =
memo[i-1] +
memo[i-2]
return memo
```

→ O(n), Platz: acuh n!

Longest common subsequence

Algo: LCSdyn(X_n,Y_m)

```
for i = 1 to n do c[i,0] = 0;
for j = 0 to m do c[0,j] = 0
for i = 1 to n do
    for j = 1 to m do
        if x_i == y_j then
            c[i,j] = c[i-1,j-1] + 1
        else
            if c[i-1,j]≥c[i,j-1] then
                c[i,j] = c[i-1,j]
            else
                c[i,j] = c[i,j-1]
return c;
```

- The conditions in the problem restrict the subproblems
  - If $x_i = y_j$, one considers the subproblem of finding the LCS of X_{i-1} and Y_{j-1}
  - If $x_i \neq y_j$, one considers the subproblems of finding the LCS of X_{i-1} and Y_j and of X_i and Y_{j-1}

Wine profit

Algo: WINEPROFITMEMOIZED(price, n, begin, end)

```
if begin > end then
    return 0;
if m[begin][end] > -1 then
    return m[begin][end];
year = n - (end-begin+1) + 1;
m[begin][end] = max(wineprofitMemoized(price, n, begin+1, end) + year *
  price[begin], wineprofitMemoized(price, n, begin, end-1) + year * price[end]);
return m[begin][end];
```

Algo: WINEPROFITDYNAMIC(price, n)

```
for i = 0 to n do
    m[i][i] = price[i] * n;
for j = 1 to n do
    for i = 0 to n − j do
        begin = i;
        end = i + j;
        year = n - (end - begin);
        m[begin][end] = max(m[begin + 1][end] + year * price[begin],
          m[begin][end - 1] + year * price[end]);
return m[0][n - 1];
```

Knapsack

```
KSRec( tot.itelms(n), capa(city) ):
// if no items or no capa (remaining) // base case
if (n <= 0 || capa <= 0) then: result = 0
// if weight exceeds capa
else if (weights[n] > capa) then:
   result = KSRec(n-1, capa) // go to next item in the array
else:
   // what's better? To put item in KS or nah?
   x1 = KSRec(n-1, capa)
   x2 = values[n] + 1KSRec(n-1, capa – weights[n])
   result = max{ x1, x2 }
return result
```

```
KSMemo(n, capa):
// base case: same as KSRec()
// check if we've already got that optimum
if ( !(results[n-1, capa]) ):
   results[n-1, capa] = calculate(n-1, capa)
// item doesn't fit in the bag
if (w[n] > capa):
   results[n, capa] = results[n-1, capa] // take previous optimum
else:
   // calc costs with current item
   if ( !( results[n-1, capa-w[n]] ):
      results[n-1, capa-w[n]] = calculate it()
   // result = max of with/without current item
   results[n, capa] =
      max{ r[n-1, capa], r[n-1, capa-w[n] + values[n]) }
return results[n, capa]
```

```
KSBottomUp(n, capa, cost, profit):
results[n, capa] "R"
// init base cases
for( i=0 to n ): R[i][0] = 0
for( c = 0 to capa ): R[0][c] = 0
// fill up table
for( i = 0 to n ):
   for( c = 0 to capa ):
      if( cost[i-1] > capa ): R[i, c] = R[i-1, c]
      else:
         R[i, c] =
         max{ R[i-1, c], R[i-1, (capa-cost[i-1]) + profit[i-1] }
return R(esults)
```

Recursive definition

f(x, capa "c") =

0	*if x = 0 or capa = 0*
f(x-1, c)	*if cost[x] > capa*
max{ f(x-1, c), f(x-1, c-cost[x])+profit[x] }	*else*

HT → sichern konstanter Zugriff auf Elemente

Hash functions

Division: h(k) = k mod m
Multiplication: h(k) = m*(kA mod 1) | abgerundet,
A=((sqrt(5)-1)/2

Collision resolution

-Chaining: use linked list
-lin.probing: h(k,i) = (h'(k)+ic) mod m
c = constant (= 1 unless stated otherwise)
i gets incremeted every time there's a collision
-()² probing: h(k,i) = (h'(k) + c_1(i) + c_2(i)²) mod m
-double hashing: h(k,i) = (h_1(k) + ih_2(k)) mod m

Algo: LinearProbingInsert(k)

```
if table is full then return error;
probe = h(k);
while HT[probe] is used do
    probe = (probe+1) mod m
table[probe] = k;
```

Algo: DoubleHashingInsert(HT,k)

```
i = 0;
repeat
    probe = (h1(k) + i*h2(k)) mod m;
    i = i+1;
until (HT[probe] is free ∨ i>m);
if i>m then Error: hash table overflow;
else HT[probe] = k;
```

```
1 i = h(k); j = 0;
2 if HT[i]==empty ∨ HT[i]==k then stop search;
3 j = (j+1) mod m; i = (i+j) mod m; goto 2;
```

Show that this schema is a quadratic probing

HASHING WITH OPEN ADDRESSING

Algo: HTinsert(HT,k)

```
i = -1;
repeat
    i++; probe = h(k,i);
until i≥m ∨ HT[probe].status≠OCC;
if i≥m then return -1;
HT[probe].status = 0;
HT[probe].key = k;
return probe
```

Algo: HTdelete(HT,k)

```
i = -1;
repeat
    i++; probe = h(k,i);
until i≥m ∨
      HT[probe].status==EMP ∨
      HT[probe].status==OCC ∧ HT[probe].key==k;
if i≥m ∨ HT[probe].status==EMP then return -1;
HT[probe].status = DEL;
return probe
```