Aufgabe 1

Wir unterscheiden im Folgenden nicht zwischen Knoten und ihrem Inhalt.

(a)

Der *inorder*-Durchlauf ergibt: *abcdefg*
Der *postorder*-Durchlauf ergibt: *badfgec*
Und der *preorder*-Durchlauf: *cabedgf*

(b)

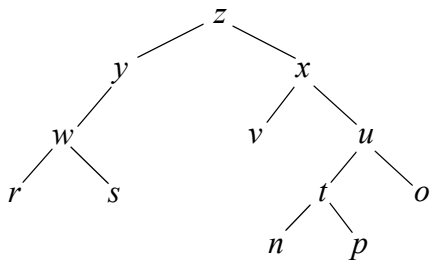
Um den Binärbaum zu rekonstruieren, machen wir uns zunutze, dass bei einem *preorder*-Durchlauf stets zuerst der Inhalt der Wurzel ausgegeben wird und dass bei einem *inorder*-Durchlauf der linke Teilbaum, gefolgt von der Wurzel, und anschließend der rechte Teilbaum betrachtet wird.

Also ist *z* die Wurzel des Binärbaums, der linke Unterbaum besteht aus den Knoten *y*, *w*, *r* und *s*, während die Knoten *x*, *v*, *u*, *t*, *o*, *n* und *p* zum rechten Unterbaum gehören.

Betrachten wir nun den linken Unterbaum und dort die Position der zugehörigen Knoten in der *preorder*-Ausgabe: ***z*ywrs***xvutnpo*

Der erste Knoten, nämlich *y*, muss Wurzel dieses Unterbaums sein. Aus der Lage von *y* in der *inorder*-Ausgabe entnehmen wir, dassmuss die Knoten *w*, *r* und *s* zum linken Unterbaum von *y* gehören.

Dieses Verfahren setzen wir fort, bis der gesamte Baum eindeutig rekonstruiert ist. Er hat die Form



(c)

Man erhält die Anzahl der Knoten in einem nichtleeren Binärbaum mit Wurzel *root*, indem man die Anzahl der Knoten des linken Unterbaums von *root*, die Anzahl der Knoten des rechten Unterbaums von *root* und 1 addiert.

```
algorithm No_Of_Nodes(root)
{   Input: rootnode of (part-)tree.
    Output: Number of nodes in the tree}
  if (root = NIL)then
       return 0
```

```
  else
     return No_Of_Nodes(left(root)) + No_Of_Nodes(right(root)) + 1
  end if
end algorithm
```

(d)

Ein leerer Baum enthält kein Blatt und ein Baum, der nur aus der Wurzel besteht, enthält ein Blatt. Sonst erhält man die Anzahl der Blätter in einem Binärbaum mit Wurzel *root* offensichtlich genau dann, wenn man die Blätter des linken Unterbaums von *root* zu den Blättern des rechten Unterbaums addiert.

```
algorithm No_Of_Leaves(root)
{    Input: rootnode of (part-)tree.
     Output: Number of leavenodes}
  if (root = NIL)then
        return 0
  else
     if (left(root) = NIL and right(root)= NIL) then
        return 1
     else
        return No_Of_Leaves(left(root)) + No_Of_Leaves(right(root))
     end if
  end if
end algorithm
```

(e)

Die Anzahl der inneren Knoten eines Binärbaums erhält man, wenn man von der Gesamtzahl der Knoten die Anzahl der Blätter subtrahiert.

```
algorithm No_Of_Inner_Nodes(root)
{    Input: rootnode of (part-)tree.
     Output: Number of inner nodes of the tree}
  return No_Of_Nodes(root) - No_Of_Leaves(root)
end algorithm
```

(f)

Existiert ein Knoten *k*, dessen Tiefe gesucht wird, nicht im Baum, so kann keine Tiefe berechnet werden und es wird -1 zurückgegeben. Ist *k* die Wurzel des Baums, dann hat *k* die Tiefe 0. Sonst erhält man die Tiefe von *k*, indem man 1 zu der Tiefe dieses Knotens im linken bzw. rechten Unterbaum addiert.

```
algorithm Depth_Of_Node(root, k, found)
{    Input: rootnode of (part-)tree, elem to search k, boolean flag found.
     Output: Depth of node k if k can be found in the tree, else -1.}
  depth := 0
  if (not found) then
     if (root = NIL) then
        return -1
```

```
    else
      if (key(root) = k) then
        found = TRUE
        return 0
      else
        if (left(root) <> NIL) then
          depth = Depth_Of_Node(left(root), k, found)
          if (found) then
            return depth + 1
          end if
        end if
        if (right(root) <> NIL) then
          depth = Depth_Of_Node(right(root), k, found)
          if (found) then
            return depth + 1
          end if
        end if
        return -1
      end if
    end if
  end if
end algorithm
```

Der Algorithmus wird mit der Wurzel des Baums, dem Wert des zu suchenden Elements und "*found* = FALSE" aufgerufen.

Aufgabe 2

(a)

Der Algorithmus besteht aus drei Teilen: Zunächst räumen wir Gleis *H*, indem wir die Waggons für *A* nach *I* und die Waggons für *B* nach *G* schieben. Danach räumen wir Gleis *G*, indem wir die Waggons für *A* ebenfalls nach *I* schieben und die Waggons für *B* auf das nun freie Gleis *H* schieben. Schließlich schieben wir noch alle Waggons (für *A*) von Gleis *I* nach Gleis *G*.

```
algorithm rangieren
  while not isEmpty(H) do
    x := top(H);
    pop(H);
    if x = A then push(I, x)
    else push(G, x)
    end if
  end while;

  while not isEmpty(G) do
    x := top(G);
    pop(G);
    if x = A then push(I, x)
```

```
    else push(H, x)
    end if
  end while;

  while not isEmpty(I) do
    x := top(I);
    pop(I);
    push(G, x);
  end while
end rangieren.
```

(b)

Wir zählen die Anzahl der bewegten Waggons in jeder **while**-Schleife und geben dazu die Anzahl der Waggons für A und B auf dem jeweils zu räumenden Gleis an:

Schleife	zu räumendes Gleis	Waggons auf dem Gleis
1	H	$a_H + b_H$
2	G	$a_G + b_G + b_H$
3	I	$a_H + a_G$
		$2a_G + b_G + 2a_H + 2b_H$

Wenn also n die Anzahl aller Waggons ist, so benötigt der Algorithmus $2n\text{-}b_G$ Verschiebe-Operationen

Aufgabe 3

(a)

Der worst case tritt ein, wenn nach einer Verdoppelung nur noch ein Element aufgenommen werden muss, also falls die Anzahl der Elemente dargestellt werden kann als $N = 2^k m + 1$. In diesem Fall müssen $k + 1$ Verdopplungen der ursprünglichen Arraygröße m durchgeführt werden. Außerdem bleiben $2^k m - 1$ Plätze des Arrays ungenutzt. Die Zuweisung der Elemente an das interne Array verursacht die Kosten $N{\cdot}S$, damit erhält man insgesamt die Kosten

$$\begin{aligned} N{\cdot}S + (1 + ... + 2^k)\ 2\cdot m{\cdot}S &= N{\cdot}S + (2^{k+1} - 1)\ 2\cdot m{\cdot}S \\ &= N{\cdot}S + (2^k + 2^k)\ 2\cdot m{\cdot}S - 2\cdot m{\cdot}S + (4{-}4)S \\ &= 5N{\cdot}S - (m + 2)\ 2\cdot S \end{aligned}$$

Somit ist das aufeinanderfolgende Einfügen von N Elementen in $O(N)$ realisierbar.

(b)

Der Speicherplatzverbrauch läßt sich reduzieren, wenn intern ein Array A der Größe n von Zeigern auf Arrays der festen Größe g gespeichert werden (entspricht einem Array von Arrays in Java). Zusätzlich entfällt dadurch das mehrfache Kopieren der darin gespeicherten Daten, denn nun muss nur noch das Array mit den Zeigern vergrößert und kopiert werden.

Das Auffinden eines Elementes an der Position p geschieht dann durch Ermitteln des Indexes für den Zeiger über die Formel $i = p$ div g, das Element liegt dann auf Position

$(A[i]\uparrow)[p \bmod g]$.

Der worst case tritt bei dieser Implementierung auf, wenn nach dem Verdoppeln des Zeiger-Arrays nur noch ein Element aufgenommen werden muss. Die Kosten für das Einfügen von $N = 2^k n \cdot g + 1$ Elementen führt nun also zu $k + 1$ Verdoppelungen von A. Es bleiben dann jedoch lediglich $(2^k n - 1)$ Plätze im Array A und $g - 1$ Plätze im Array $A[2^k n]\uparrow$ ungenutzt. Die Kosten für das Einfügen betragen nun $N \cdot S + N \cdot s/g$ (Array-Elemente + Zeiger), wobei $s < S$ die Kosten für das Einfügen bzw. Kopieren eines Zeigers angibt. Insgesamt erhält man nun die Kosten

$$\begin{aligned} N \cdot S + N \cdot s/g + (1 + \ldots + 2^k) n \cdot s &= N \cdot S + N \cdot s/g + (2^{k+1} - 1) n \cdot s \\ &= N \cdot S + N \cdot s/g + (2^k + 2^k) n \cdot g \cdot s/g - n \cdot s + (2-2) \cdot s/g \\ &= N \cdot S + 3N \cdot s/g - n \cdot s - 2 \cdot s/g \\ &= N \cdot S + 3N \cdot s/g - (n \cdot g + 2) \cdot s/g \end{aligned}$$

Wählt man nun z.B. $n = 1$ und $g = m$, so daß N denselben Wert wie in Aufgabenteil (a) hat, so erhält man

$$N \cdot S + 3N \cdot s/m - (m + 2) \cdot s/m$$

und es läßt sich leicht ablesen (wenn $m << N$ und $s << S$), daß die Kosten dieser Implementierung für das aufeinanderfolgende Einfügen von N Elementen deutlich niedriger ist. Dafür sind aber die Kosten für den direkten Zugriff höher, da die Position berechnet, und ein Zeiger dereferenziert werden muss.

Alternativ könnte man statt der Arrays der Größe g auch direkt Speicher für jedes Element allokieren und dieses dorthin kopieren, so dass nur noch Zeiger auf Elemente verwaltet werden müssen, dies entspräche dem Fall $g = 1$.