Datenstrukturen und Algorithmen

Vorlesung 11

Überblick

- Vorige Woche:
  - Hashtabellen:
    - Offene Adressierung
    - Perfektes Hashing
    - Cuckoo Hashing
    - Verkettete Hashtabellen

- Heute betrachten wir:
  - Hashtabellen: Hopscotch hashing
  - Bäume - allgemeines
  - Binärbäume
    - Einführung
    - ADT Binary Tree
    - Traversierungen

Kollisionsauflösung:
Hopscotch Hashing

Offene Adressierung – Wiederholung

- Bei der offenen Adressierung werden die Elemente in der Hashtabelle gespeichert
- Für lineares Sondieren ist die Hashfunktion:

$$h(k, i) = (h'(k) + i) \; mod \; m, \forall i = 0, \ldots, m-1$$

- wobei h’(k) eine einfache Hashfunktion ist (z.B. h’(k) = k mod m)

- die Sondierungssequenz für lineares Sondieren ist:

  <h’(k), h’(k)+1,h’(k)+2, ..., m-1, 0, 1, ..., h’(k)-1>

Offene Adressierung – Wiederholung

- Nachteil des linearen Sondierens ist *primäres (und sekundäres) Clustern* – es bilden sich lange Folgen besetzter Slots, wodurch sich die Suchzeiten erhöhen

- Cluster sind ein Problem, da eine erfolglose Suchoperation viele Positionen durchsuchen muss bis man eine leere Position erreicht

- Ein Vorteil des linearen Sondierens ist, dass man aufeinanderfolgende Positionen durchsucht

Hopscotch Hashing

- *Hopscotch Hashing* versucht die Komplexität des Suchalgorithmus bei dem linearen Sondieren zu verbessern
- Hopscotch Hashing vereint Cuckoo-Hashing und lineares Sondieren

- Die Hauptidee bei *Hopscotch Hashing* ist eine **Umgebung/Nachbarschaft von Größe *H*** (*H* ist eine Konstante) für jede Position zu definieren und sicherzustellen, dass jedes Element, das zu einer Position *i* gehasht wird, in der Umgebung der Position *i* gespeichert wird
- Folglich muss man ein Element nur in der Umgebung der Position, wo dieses gehasht wird, suchen

Hopscotch Hashing

- Jede Position aus der Hashtabelle hat eine entsprechende ***Hop-Info*** – meistens ein **Bitmap von Größe *H***

- Wenn der **Bit *p*** in der ***Hop-Info* der Position *i*** den **Wert 1** hat, dann wird das Element von der **Position *i + p* auf die Position *i* gehasht**

- Wenn der **Bit *p*** in der *Hop-Info* der Position *i* den **Wert 0** hat, dann ist die **Position *i + p* entweder leer** oder es enthält ein Element, das auf eine Position **unterschiedlich von *i* gehasht** wird

Hopscotch Hashing – Beispiel

0		0000
1	67	1010
2	13	1000
3	45	0010
4	92	1010
5	25	0000
6	15	0001
7	51	1000
8	19	1000
9	72	0000
10		0000

h(92, 0) = 4
h(13, 0) = 2
h(51, 0) = 7
h(67, 0) = 1
h(19,0) = 8
h(45, 0) = 1
h(45, 1) = 2
h(45, 2) = 3
h(25, 0) = 3
h(25, 1) = 4
h(25, 2) = 5
h(15, 0) = 4
h(15, 1) = 5
h(25, 2) = 6
h(72, 0) = 6
h(72, 1) = 7
h(72, 2) = 8
h(72, 3) = 9

m = 11
h(k) = k % m
H = 4

- Schwarze Linien zeigen die endgültige Position der Elemente
- Fett gedruckte Linien zeigen die Positionen wo jedes Element gehasht wird (wird benutzt um Hop-Info zu erstellen)

Hopscotch Hashing – Einfügeoperation

- Wenn man ein neues Element *k* einfügen muss:
  - Man berechnet die anfängliche Position, $p = h(k,0)$
  - Falls *p* leer ist, dann speichert man *k* auf die Position *p*
  - Falls *p* nicht leer ist, dann muss man eine leere Position finden, *pe,* genau wie beim linearen Sondieren
  - Falls *pe* in der Umgebung von *p* ist (zwischen *p* und $p + H - 1$), dann speichert man *k* auf die Position *pe*
  - Falls ***pe* zu weit** ist, dann versucht man eine Position näher an *p* zu finden: man überprüft ob es ein Element (näher an *p*) gibt, das auf die Position *pe* verschoben werden kann.
  - Falls man ein solches Element findet, dann verschiebt man dieses auf die Position *pe.* Man wiederholt diesen Teil bis es eine leere Position in der Umgebung von *p* gibt.

Hopscotch Hashing – Einfügeoperation Beispiel

- Füge das Element 90 ein
- $p = h(90,0) = 2$ - besetzt
- *pe* = erste leere Position = 10 - zu weit (Umgebung H = 4)
- Man versucht ein Element auf die Position 10 zu verschieben. Kandidaten dafür sind Elemente, die auf die Positionen 7, 8 und 9 gehasht werden (diese enthalten die Position 10 in der Umgebung)
- Wir schauen uns die Hop-Info der Position 7 an. Falls die Hop-Info einen Wert 1 enthält, dann verschiebt man dieses Element auf die Position 10, damit diese neue Position frei wird.

Hopscotch Hashing – Einfügeoperation Beispiel

0		0000
1	67	1010
2	13	1000
3	45	0010
4	92	1010
5	25	0000
6	15	0001
7		0001
8	19	1000
9	72	0000
10	51	0000

h(92, 0) = 4
h(13, 0) = 2
h(51, 0) = 7
h(67, 0) = 1
h(19,0) = 8
h(45, 0) = 1
h(45, 1) = 2
h(45, 2) = 3
h(25, 0) = 3
h(25, 1) = 4
h(25, 2) = 5
h(15, 0) = 4
h(15, 1) = 5
h(25, 2) = 6
h(72, 0) = 6
h(72, 1) = 7
h(72, 2) = 8
h(72, 3) = 9

m = 11
h(k) = k % m
H = 4

Hopscotch Hashing – Einfügeoperation Beispiel

- Jetzt ist die Position 7 frei, aber diese ist immer noch zu weit von der Position 2 entfernt, wo das Element 90 gehasht wird
- Man versucht ein Element auf die Position 7 zu verschieben. Mögliche Kandidaten sind die Positionen: 4, 5, 6.

Hopscotch Hashing – Einfügeoperation Beispiel

0		0000
1	67	1010
2	13	1000
3	45	0010
4		0011
5	25	0000
6	15	0001
7	92	0001
8	19	1000
9	72	0000
10	51	0000

h(92, 0) = 4
h(13, 0) = 2
h(51, 0) = 7
h(67, 0) = 1
h(19,0) = 8
h(45, 0) = 1
h(45, 1) = 2
h(45, 2) = 3
h(25, 0) = 3
h(25, 1) = 4
h(25, 2) = 5
h(15, 0) = 4
h(15, 1) = 5
h(25, 2) = 6
h(72, 0) = 6
h(72, 1) = 7
h(72, 2) = 8
h(72, 3) = 9

m = 11
h(k) = k % m
H = 4

Hopscotch Hashing – Einfügeoperation Beispiel

- Position 4 ist jetzt frei und befindet sich in der Umgebung der Position 2, wo 90 gehasht wird

0		0000
1	67	1010
2	13	1010
3	45	0010
4	90	0011
5	25	0000
6	15	0001
7	92	0001
8	19	1000
9	72	0000
10	51	0000

h(92, 0) = 4
h(13, 0) = 2
h(51, 0) = 7
h(67, 0) = 1
h(19,0) = 8
h(45, 0) = 1
h(45, 1) = 2
h(45, 2) = 3
h(25, 0) = 3
h(25, 1) = 4
h(25, 2) = 5
h(15, 0) = 4
h(15, 1) = 5
h(25, 2) = 6
h(72, 0) = 6
h(72, 1) = 7
h(72, 2) = 8
h(72, 3) = 9

m = 11
h(k) = k % m
H = 4

Hopscotch Hashing – Einfügeoperation

- Falls kein Element verschoben wird, um eine nähere Position zu befreien, dann kann das Element nicht in die Hashtabelle eingefügt werden

- In dieser Situation muss die Hashtabelle vergrößert werden und alle Elemente müssen wieder eingefügt werden (resize, rehash)

- In der Praxis funktioniert Hopscotch Hashing gut mit einem Belegungsfaktor von bis zu 0.9

Hopscotch Hashing – Suchoperation und Löschoperation

- Wenn man ein Element sucht, dann berechnet man den Hashwert für das Element, *p*, und dann überprüft man die Positionen entsprechend zu Werten von 1 in dem Hop-Info von *p*

- Man muss höchstens *H* Positionen überprüfen (H ist eine Konstante)

- Wenn man ein Element löschen muss, dann muss man das Element erstmal suchen und falls das Element gefunden wird, dann setzt man die Position als leer (man muss keine anderen Elemente verschieben)

Hashtabellen – Anwendungen

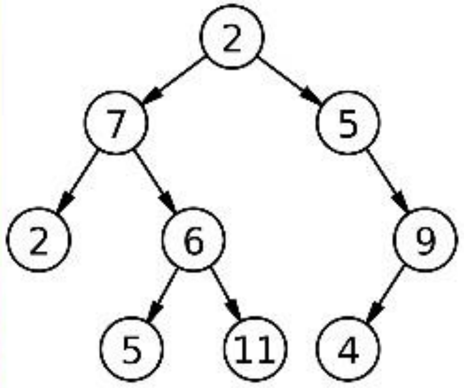
- Typische Implementierungen für ADT Map und Set (warum?)
- Symboltabellen im Compiler / Interpreter
- Indexieren / Suchen in Datenbanken (Tendenz ist aber, eher Bäume zu benutzen)
- Datenschutz (man benutzt besondere Hashfunktionen)

Bäume

- Bäume sind eine der wichtigsten Datenstrukturen der Informatik und Datenverarbeitung, da diese eine effiziente Methode zur Datenspeicherung bieten

- Ein Baum ist in der Graphentheorie ein spezieller Typ von Graph, der zusammenhängend ist und keine geschlossenen Pfade enthält

- Meistens redet man von Bäumen, die genau einen ausgezeichneten Knoten besitzen, der als **Wurzel** bezeichnet wird



Baum – Definition

- Eine Datenstruktur *B* = (N, V) heißt **Baum**, wenn *B* die folgenden Eigenschaften besitzt:
  - Es gibt genau einen Knoten, der **keinen Vorgänger** besitzt. Diesen Knoten bezeichnen wir als die **Wurzel** von *B*
  - Alle Knoten von *B,* außer der Wurzel besitzen **genau einen Vorgängerknoten**

Baum – Definition

- Ein Baum ist eine endliche Menge *T* von 0 oder mehreren Elementen, die Knoten heißen, mit folgenden Eigenschaften:
  - Falls *T* leer ist, dann ist der Baum leer
  - Falls *T* nicht leer ist, dann:
    - Gibt es genau einen ausgezeichneten Knoten, der als *Wurzel des Baumes* bezeichnet wird
    - Alle anderen Knoten werden in disjunkten Bäumen zerlegt T_1, T_2,..., T_k, wobei die Wurzel des Baumes durch eine Kante mit der Wurzel dieser Bäume verbunden ist. Die Bäume T_1, T_2,..., T_k heißen *Unterbäume* von R, und R heißt *Vater* der Unterbäume

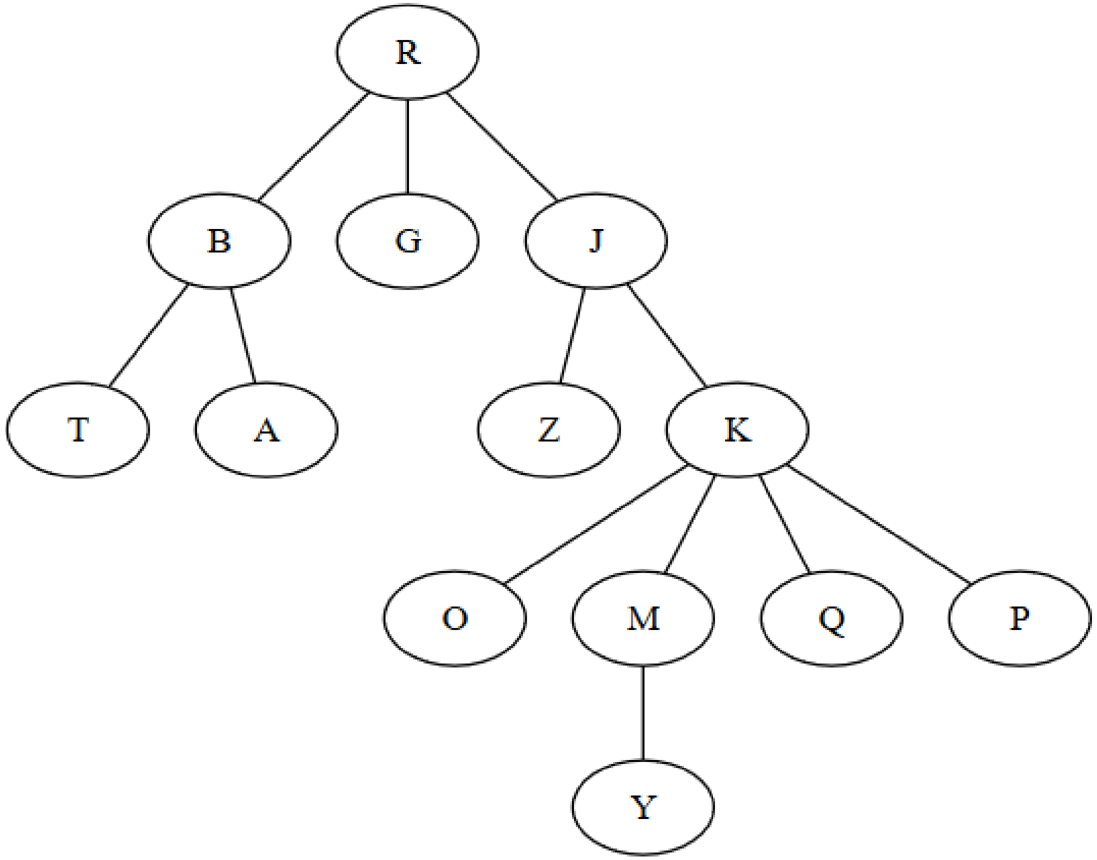
Baum – Begriffe

- **Geordneter Baum** – Ein Baum heißt geordnet, wenn die Kinder eines jeden Knotens eine vollständig geordnete Menge bilden (man kann eine **Ordnung auf die Knoten** definieren, die eine Rolle spielt). Sie lassen sich dann z.B. gemäß dieser Ordnung linear aufsteigend verketten
- Unter dem ***Rang*** eines Knotens verstehen wir die **Anzahl der Kinder/Söhne des Knotens.**
- Der **maximale Rang** eines Knoten im Baum heißt ***Ordnung*** des Baumes
- Als ***Blätter*** bezeichnen wir diejenigen Knoten, die keinen Nachfolger besitzen (Rang 0). Alle anderen Knoten heißen ***innere Knoten*** von *B*

Baum – Begriffe

- Ein ***Pfad*** in einem Baum ist eine Folge von unterschiedlichen Knoten, in der die aufeinander folgenden Knoten durch Kanten miteinander verbunden sind
- ***Die Tiefe eines Knotens*** ist sein Abstand von der Wurzel, d.h. die Länge des Pfades (Anzahl der Kanten) von der Wurzel zu ihm
- Die Wurzel hat die Tiefe 0
- Als ***Höhe/Niveau eines Knotens*** bezeichnen wir die größtmögliche Pfadlänge von dem Knoten zu einem Blatt
- Als ***Höhe eines Baumes*** bezeichnen wir die größtmögliche Pfadlänge in ihm, d.h. die Länge des längsten Weges zwischen der Wurzel und einem Blatt (i.e. die Höhe der Wurzel)

Baum – Begriffe (Beispiel)



- Wurzel des Baumes: R
- Kinder von R: B, G, J
- Vater von M: K
- Blätterknoten: T, A, G, Z, O, Y, Q, P
- innere Knoten: R, B, J, K, M
- Tiefe des Knotens K: 2 (Pfad R-J-K)
- Höhe des Knotens K: 2 (Pfad K-M-Y)
- Höhe des Baumes: 4 (Höhe des Knotens R)
- Knoten auf dem Niveau 2: T, A, Z, K

k-närer Baum

- Wenn die Anzahl der Kinder vorgegeben ist (höchstens k Kinder), dann spricht man von einem k-nären Baum.
- Wie kann man einen k-nären Baum darstellen?
- Eine Möglichkeit ist, die Struktur des Knoten folgendermaßen zu definieren:
  - Information des Knotens
  - Adresse des Vaterknotens (kann auch fehlen)
  - *k* Felder für die k Kinderknoten

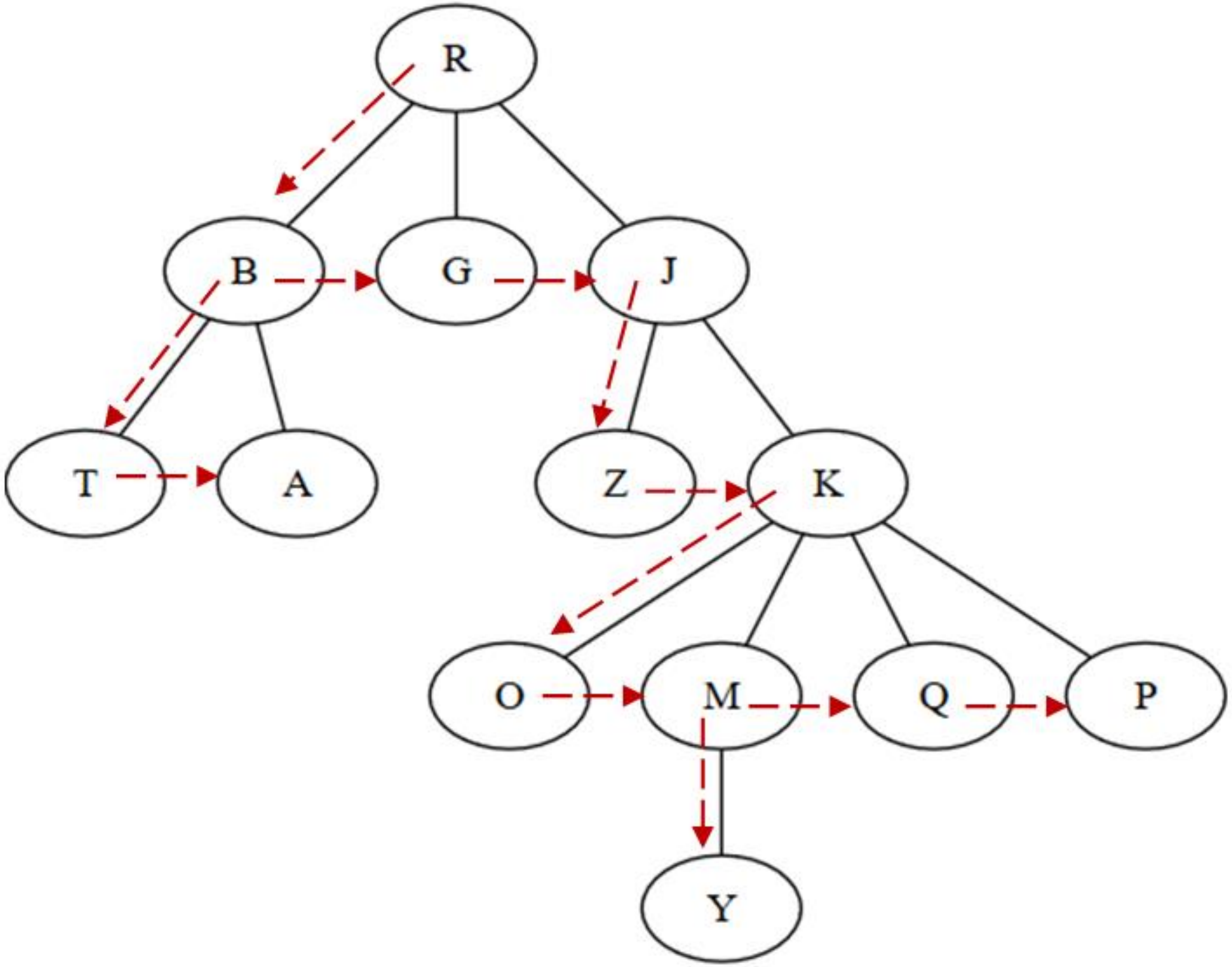
k-närer Baum

- Eine andere Möglichkeit wäre, die Struktur des Knotens wie folgt zu definieren:
  - Information des Knotens
  - Adresse des Vaterknotens (kann auch fehlen)
  - Ein Array von Größe k, in welchem man die Adressen der Kinderknoten speichert
  - die Anzahl der Kinder (Anzahl der besetzen Positionen in dem Array)
- Nachteil dieser zwei Methoden ist, dass man Speicherplatz für k Kinder besetzt, auch wenn es weniger als k Kinder für einen Knoten gibt

k-närer Baum

- Eine dritte Möglichkeit ist die *linkes-Kind rechter-Bruder* Darstellung (s. binomial Heap), wobei die Struktur eines Knotens folgende Felder enthält:
  - Information des Knotens
  - Adresse des Vaterknotens (kann auch fehlen)
  - Adresse des Kindes ganz links
  - Adresse des rechten Bruders des Knotens (der nächste Knoten auf demselben Niveau)

linkes-Kind rechter-Bruder Darstellung – Beispiel



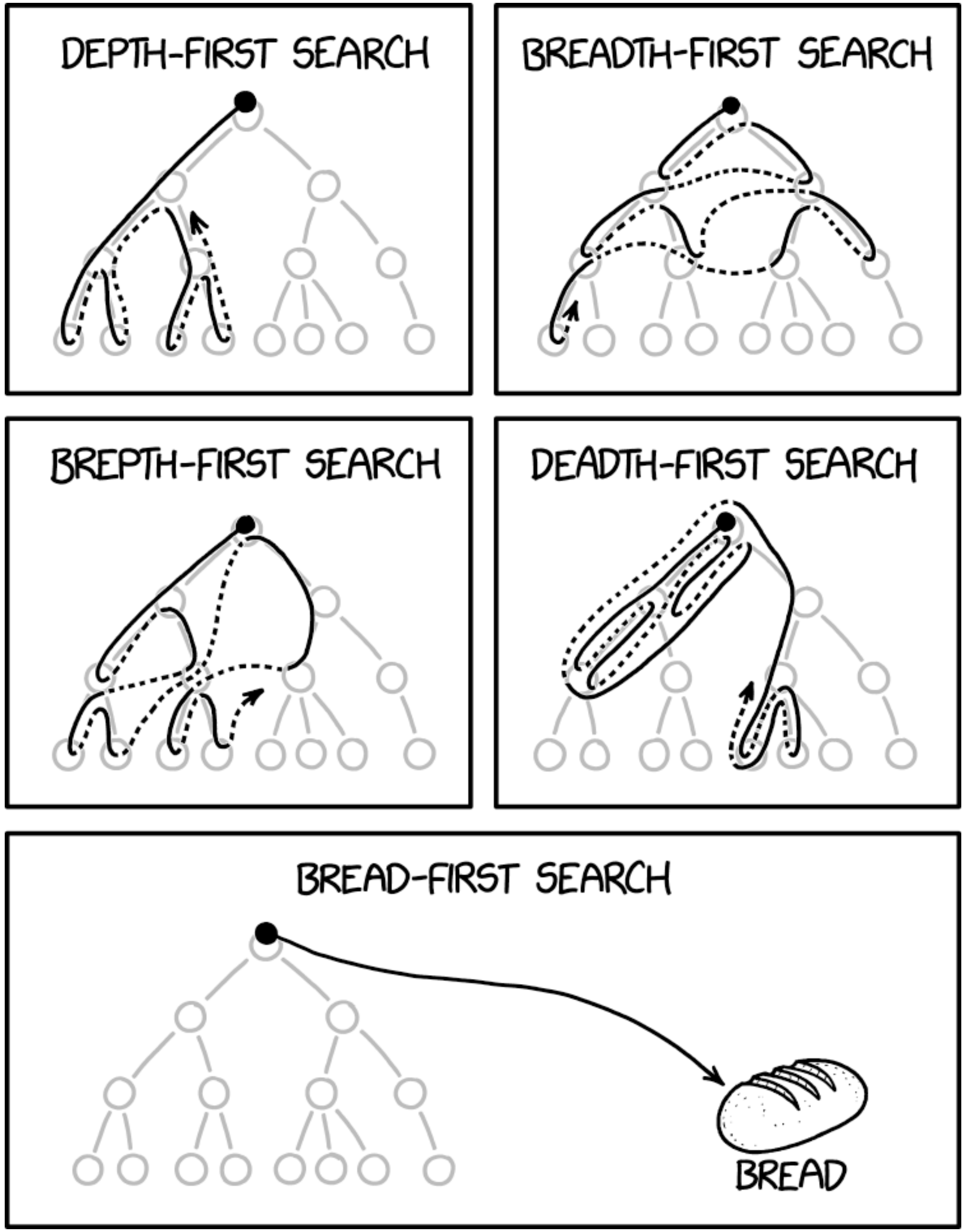
Baum – Traversierungen

- Mit dem Begriff “Traversieren” bezeichnet man das Durchlaufen sämtlicher Knoten eines Baumes in einer bestimmten Reihenfolge.
- In der Regel wird mit dem Traversieren die Bearbeitung vieler oder aller Knoten verbunden sein.

Baum – Traversierungen

- Für die Traversierung eines k-nären Baumes gibt es nur zwei mögliche Traversierungen:

  - **Level-Ordnung** (breadth first)
  - **In die Tiefe** (depth first)

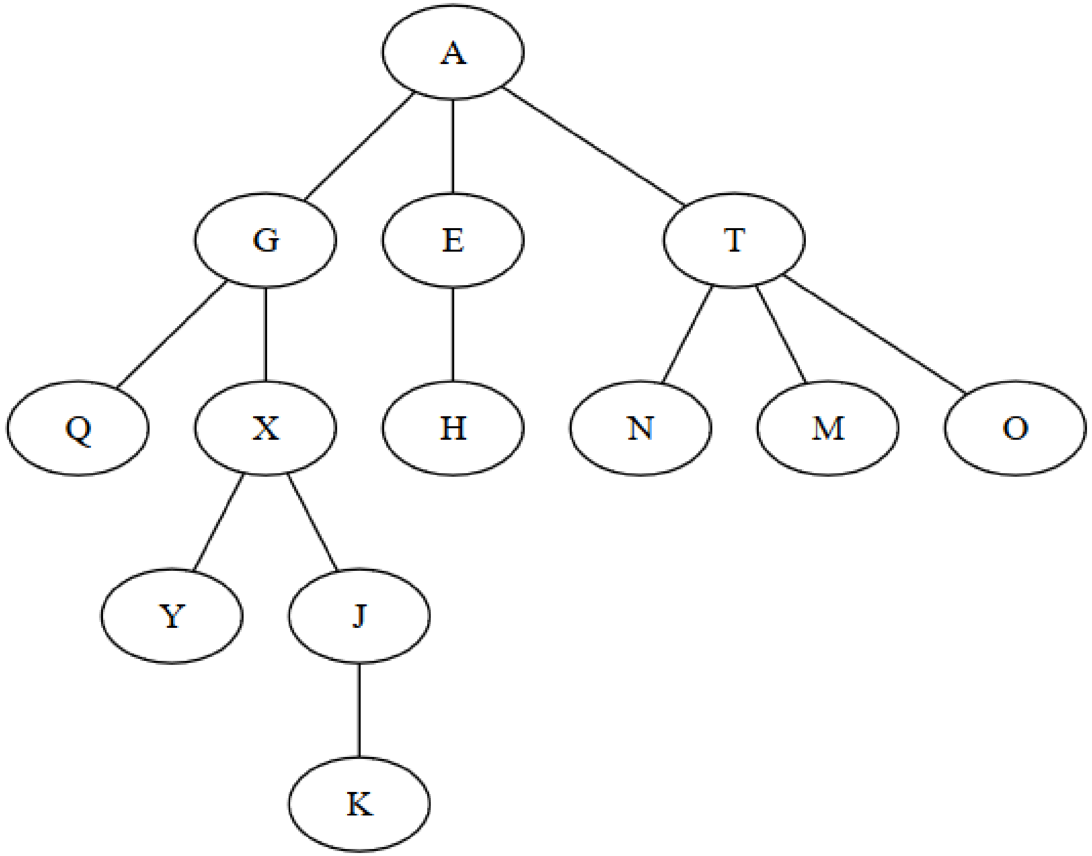


"A death-first search is when you lose your keys and travel to the depths of hell to find them, and then if they're not there you start checking your coat pockets."

Baum – Level-Ordnung Traversierung

- Traversierung fängt von der Wurzel an
- Man besucht alle Kinder der Wurzel und erst nachher geht man zu der nächsten Ebene (die Kinder der Kinder), usw.
- Man geht zu der nächsten Ebene nur nachdem alle Knoten einer Ebene besucht wurden.
- Man benutzt eine Hilfs-Schlange um die Knoten zu speichern, die besucht werden müssen.

Baum – Level-Ordnung Traversierung - Beispiel



Queue: A

Besuche A, Queue: G, E, T

Besuche G, Queue: E, T, Q, X

Besuche E, Queue: T, Q, X, H

Besuche T, Queue: Q, X, H, N, M, O

Besuche Q, Queue: X, H, N, M, O

Besuche X, Queue: H, N, M, O, Y, J

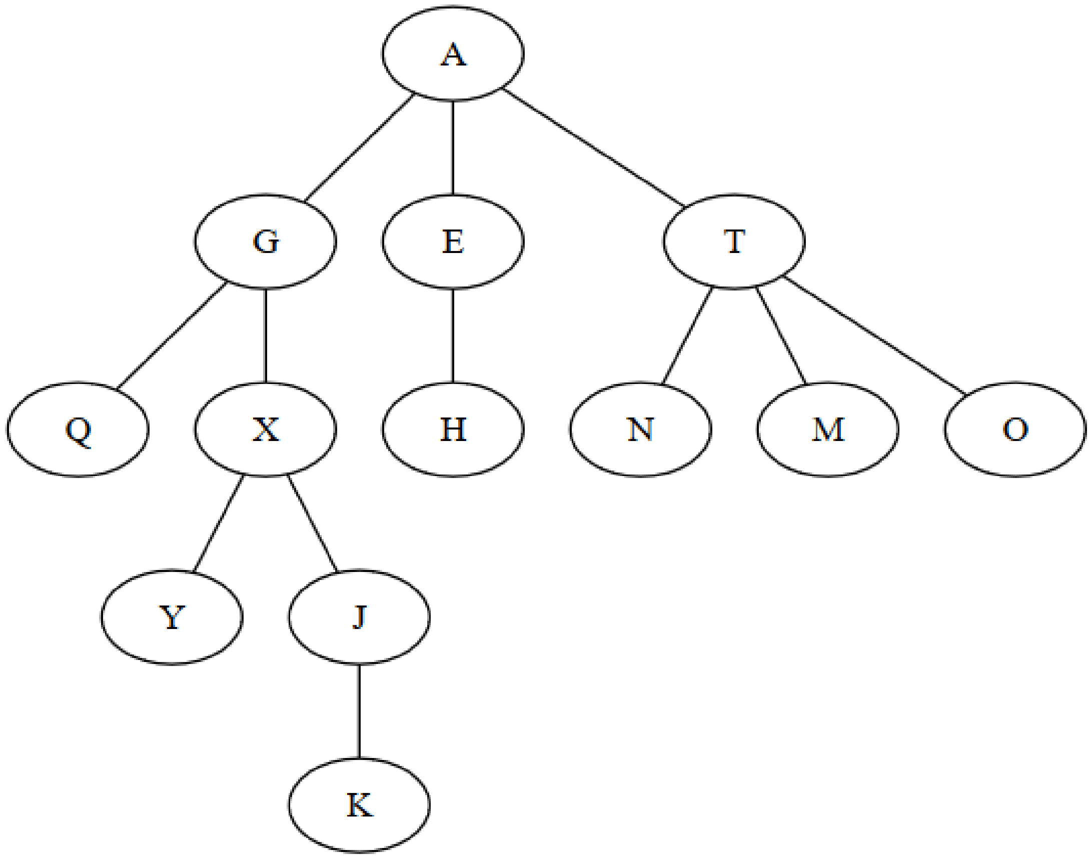
Usw.

Level-Ordnung (breadth first): A, G, E, T, Q, X, H, N, M, O, Y, J, K

Baum – Traversierung in die Tiefe

- Traversierung fängt von der Wurzel an
- Von der Wurzel aus besucht man eines der Kinder, dann ein Kind von diesem Kind, usw.
- Man geht so tief wie möglich in dem Baum, und erst nachdem der ganze Teilbaum (alle Nachkommen des ersten Kindes) besucht wurde, geht man zu dem nächsten Kind weiter.
- Man benutzt einen zusätzlichen Hilfs-Stack um die Knoten zu speichern, die besucht werden müssen.

In die Tiefe Traversierung - Beispiel



- Stack: A
- Besuche A, Stack: T, E, G
- Besuche G, Stack: T, E, X, Q
- Besuche Q, Stack: T, E, X
- Besuche X, Stack: T, E, J, Y
- Besuche Y, Stack: T, E, J
- Besuche J, Stack: T, E, K
- usw.

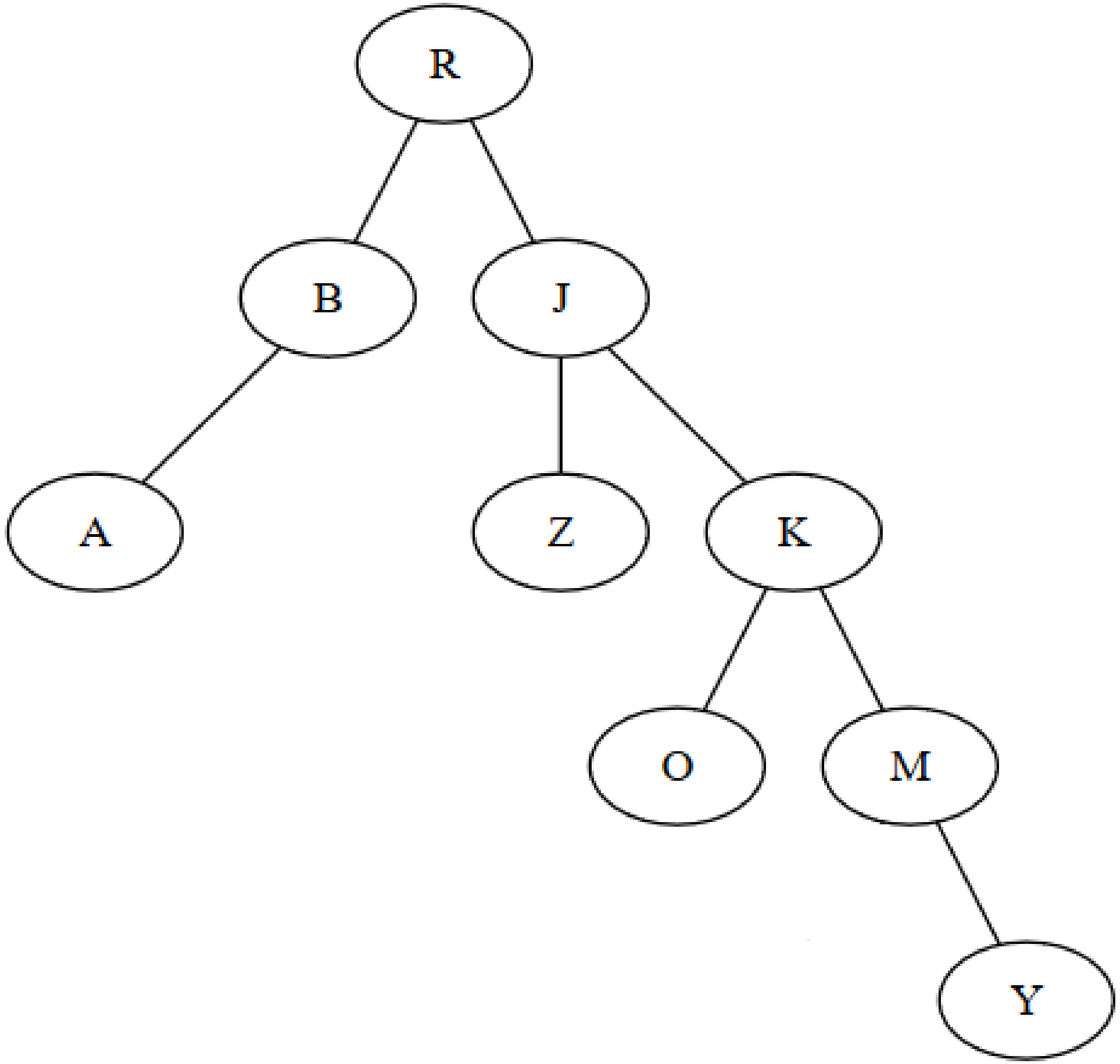
In die Tiefe (depth first): A, G, Q, X, Y, J, K, E, H, T, N, M, O

Binärbäume

- Ein Binärbaum ist ein Baum, in welchem jeder Knoten höchstens zwei Kinder hat
- Ein Binärbaum ist ein spezieller k-närer Baum, nämlich ein k-närer Baum der Ordnung $k = 2$.

- Alle Kinder werden als linkes Kind oder rechtes Kind bezeichnet.
- Auch wenn ein Knoten nur ein Kind hat, muss man wissen ob dieses das linke oder das rechte Kind ist.

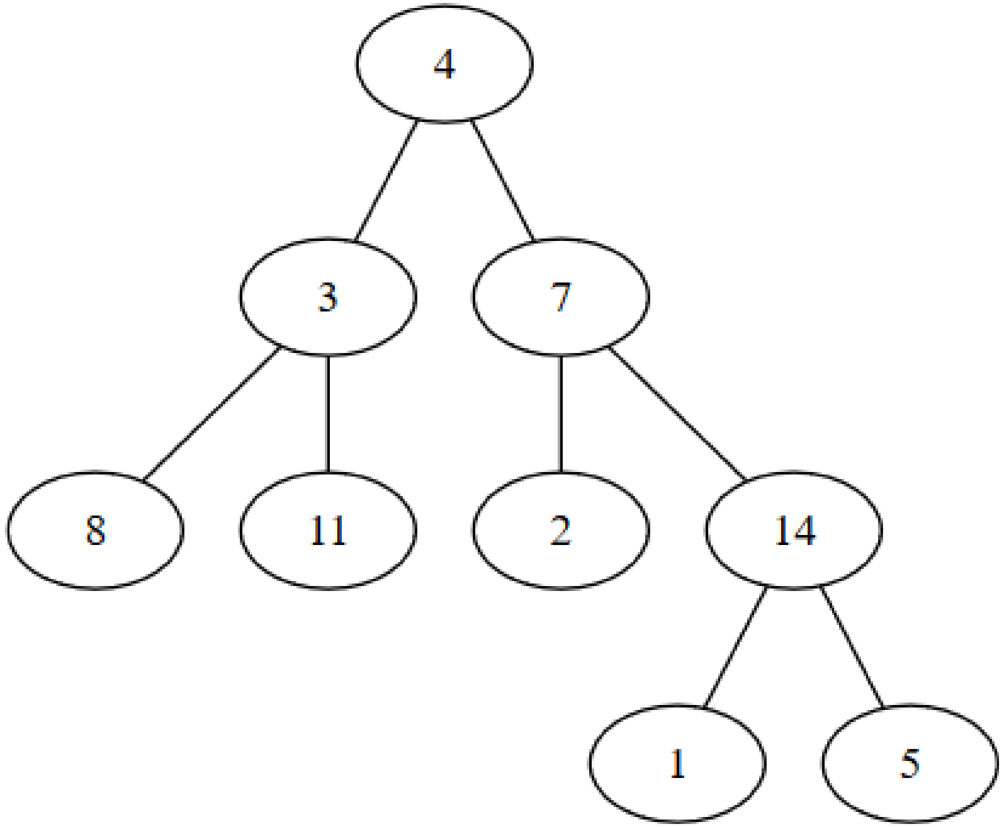
Binärbaum – Beispiel



- A ist das linke Kind von B
- Y ist das rechte Kind von M

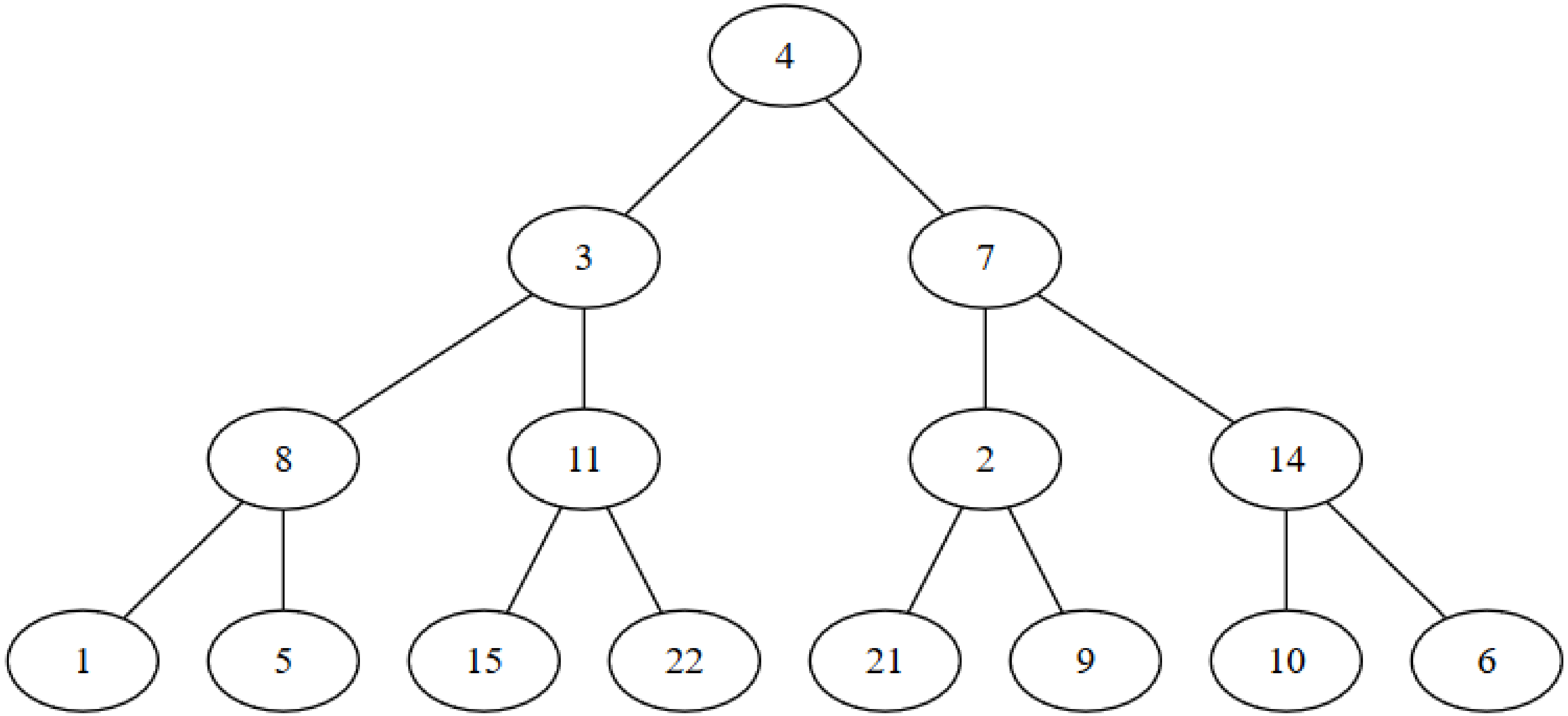
Binärbäume

- Ein Binärbaum heißt **voll** (full), wenn jeder **innere Knoten genau zwei Kinder** hat



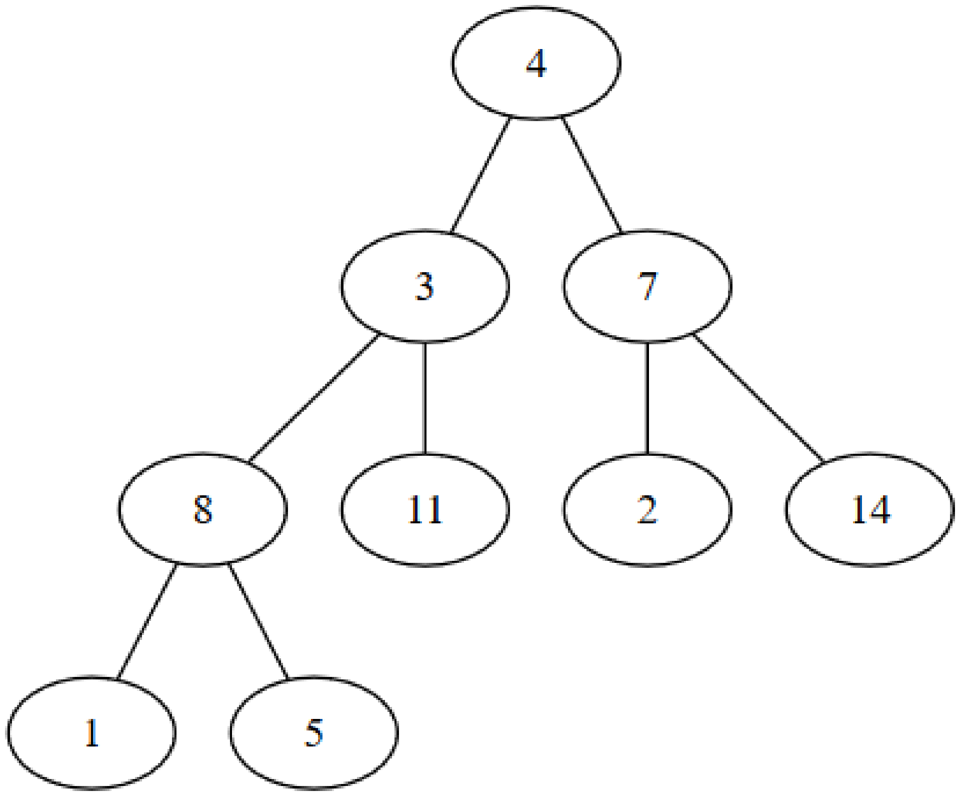
Binärbäume

- Ein Binärbaum heißt ***vollständig/komplett*** (complete), wenn **jede Ebene die maximale Anzahl von Knoten** enthält



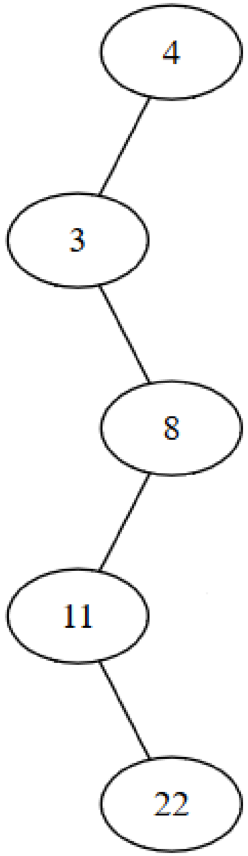
Binärbäume

- Ein Binärbaum heißt ***fast vollständig*** (almost complete), wenn **jede Ebene außer der letzten die maximale Anzahl von Knoten** enthält, und auf der letzten Ebene die Knoten **von links nach rechts ausgefüllt sind** (die Struktur eines binären Heap)



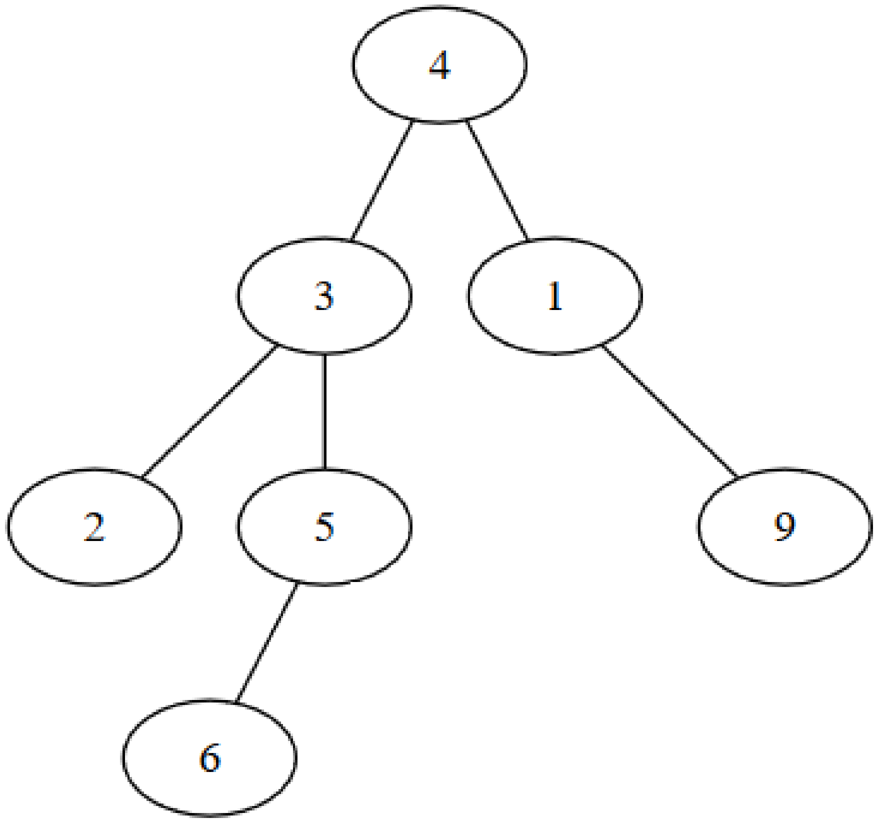
Binärbäume

- Ein Binärbaum heißt ***degeneriert***, wenn jeder innere Knoten genau ein Kind hat (eigentlich eine Verkettung von Knoten)



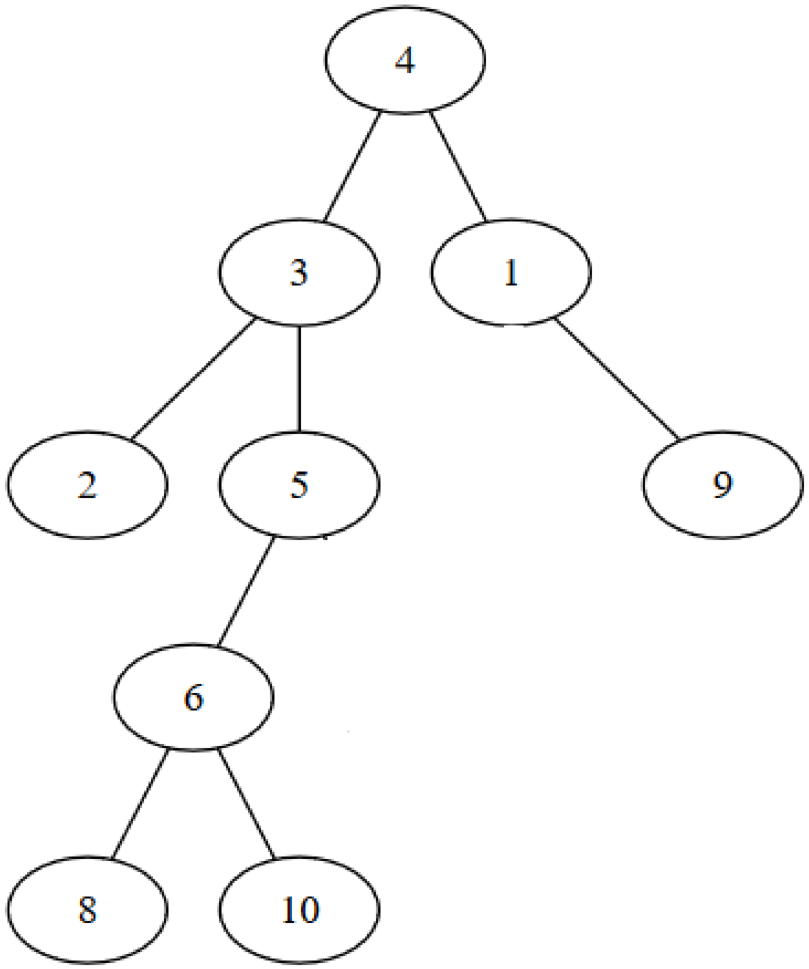
Binärbäume

- Ein Binärbaum heißt ***balanciert***, wenn für jeden Knoten **die Höhen der zwei Teilbäume mit höchstens 1 voneinander abweichen**



Binärbäume

- Es gibt binäre Bäume, die keine der bisherigen Eigenschaften besitzen, zum Beispiel:



Binärbäume – Eigenschaften

- Ein Binärbaum mit *n* Knoten hat genau *n-1* Kanten (stimmt für jeden Baum, nicht nur für binären Bäume)
- Ein vollständiger Binärbaum der Höhe *N* hat $2^{N+1} - 1$ Knoten ($1 + 2 + 4 + 8 + ... + 2^N$), davon 2^N Blätter – **von hier kommt die logarithmische Komplexität der meisten Algorithmen auf Binärbäumen**
- Die maximale Anzahl von Knoten in einem Binärbaum der Höhe *N* ist $2^{N+1} - 1$ – falls der Baum vollständig ist
- Die minimale Anzahl von Knoten in einem Binärbaum der Höhe *N* ist *N+1* – falls der Baum degeneriert ist
- Ein Binärbaum mit *N* Knoten hat eine Höhe zwischen $log_2 N$ und *N-1*

ADT Binary Tree/Binärbaum

- Domäne für ADT Binary Tree/Binärbaum:

$\mathcal{BT}$ = {*bt* | *bt* ist ein Binärbaum mit Knoten, die Informationen vom Typ TElem enthalten}

ADT Binary Tree

- init(bt)
  - **descr:** erstellt einen neuen, leeren Binärbaum
  - **pre:** wahr
  - **post**: $bt \in \mathcal{BT}$, *bt* ist ein leerer Binärbaum

- initLeaf(bt, e)
  - **descr**: erstellt einen neuen Binärbaum, der nur eine Wurzel mit dem gegebenen Wert enthält
  - **pre**: $e \in TElem$
  - **post**: $bt \in \mathcal{BT}$, *bt* ist ein Binärbaum mit einem einzigen Knoten (die Wurzel), der den Wert *e* enthält

ADT Binary Tree

- initTree(bt, left, e, right)
  - **descr:** erstellt einen neuen Binärbaum, mit dem gegebenen Wert in der Wurzel und mit den zwei gegebenen Binärbäume als Unterbäume
  - **pre:** $left \in \mathcal{BT}$, $right \in \mathcal{BT}$, $e \in TElem$
  - **post**: $bt \in \mathcal{BT}$, *bt* ist ein Binärbaum, deren Wurzel den Wert *e* enthält und deren linken Kind *left* und rechten Kind *right* ist

ADT Binary Tree

- insertLeftSubtree(bt, left)
  - **descr:** setzt den linken Unterbaum des gegebenen Binärbaumes mit dem gebenen Wert (falls der Binärbaum einen linken Unterbaum besitzt wird dieser ersetzt)
  - **pre:** $bt \in \mathcal{BT}$, $left \in \mathcal{BT}$
  - **post**: $bt' \in \mathcal{BT}$, der linke Unterbaum von bt' ist gleich mit *left*

- insertRightSubtree(bt, right)
  - **descr:** setzt den rechten Unterbaum des gegebenen Binärbaumes mit dem gebenen Wert (falls der Binärbaum einen rechten Unterbaum besitzt wird dieser ersetzt)
  - **pre:** $bt \in \mathcal{BT}$, $right \in \mathcal{BT}$
  - **post**: $bt' \in \mathcal{BT}$, der rechte Unterbaum von bt' ist gleich mit *right*

ADT Binary Tree

- root(bt)
  - **descr:** gibt den Wert aus der Wurzel des Binärbaumes zurück
  - **pre:** $bt \in \mathcal{BT}, bt \neq \emptyset$
  - **post**: $root \leftarrow e$, $e \in TElem$, e ist die Information aus der Wurzel von *bt*
  - **throws:** Exception falls *bt* leer ist

- left(bt)
  - **descr:** gibt den linken Unterbaum des Binärbaumes zurück
  - **pre:** $bt \in \mathcal{BT}, bt \neq \emptyset$
  - **post**: $left \leftarrow l$, $l \in \mathcal{BT}$, *l* ist der linke Unterbaum von *bt*
  - **throws:** Exception falls *bt* leer ist

ADT Binary Tree

- right(bt)
  - **descr:** gibt den rechten Unterbaum des Binärbaumes zurück
  - **pre:** $bt \in \mathcal{BT}, bt \neq \emptyset$
  - **post**: *right* $\leftarrow l$, *l* $\in \mathcal{BT}$, *l* ist der rechte Unterbaum von *bt*
  - **throws:** Exception falls *bt* leer ist

- isEmpty(bt)
  - **descr:** überprüft ob der Binärbaum leer ist
  - **pre:** $bt \in \mathcal{BT}$
  - **post**: *isEmpty* $\leftarrow \begin{cases} wahr, & falls\ bt \neq \emptyset \\ falsch, & ansonsten \end{cases}$

ADT Binary Tree

- iterator(bt, traversal, i)
  - **descr:** gibt einen Iterator zurück für den Binärbaum
  - **pre:** *bt* $\in \mathcal{BT}$ *, traversal* stellt die Reigenfolge dar, in welcher die Elemente des Binärbaumes durchlaufen werden
  - **post**: *i* $\in \mathcal{I}$, *i* ist ein Iterator für *bt*, der die Elemente in der Reihenfolge bestimmt von der gegebenen Traversierung *traversal* durchläuft

- destroy(bt)
  - **descr:** zerstört einen Binärbaum
  - **pre:** *bt* $\in \mathcal{BT}$
  - **post**: *bt* wurde zerstört

ADT Binary Tree

- Andere mögliche Operationen:
  - Die Information in der Wurzel des Binärbaumes ändern
  - Einen Unterbaum (linker oder rechter) löschen
  - Ein Element in dem Binärbaum suchen
  - Die Anzahl der Elemente in dem Binärbaum zurückgeben

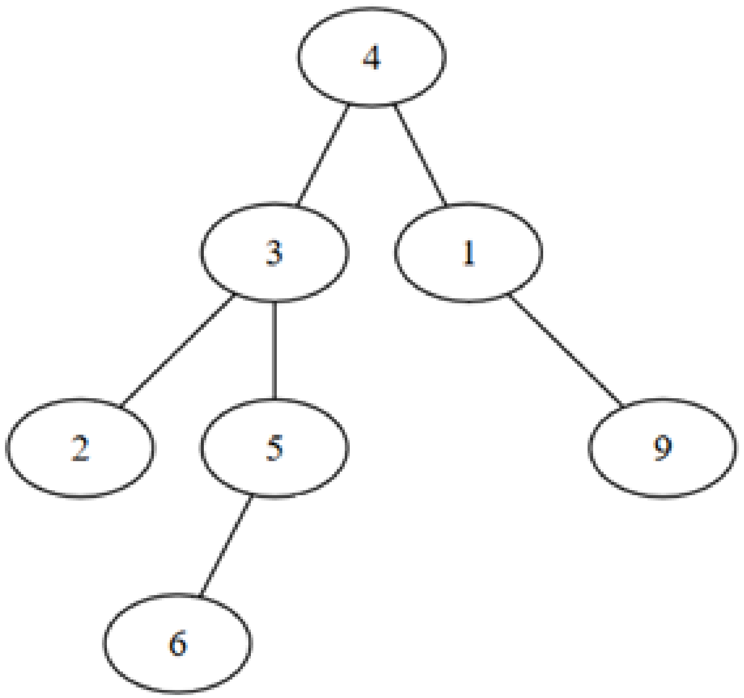
ADT Binary Tree - Repräsentierungen

- Es gibt unterschiedliche Möglichkeiten einen Binärbaum zu repräsentieren:
  - Mithilfe eines Arrays (ähnlich wie bei dem binären Heap)
  - Mit verketteten Listen:
    - Mit dynamischer Allokation
    - Auf einem Array

ADT Binary Tree – Repräsentierung I

- Repräsentierung mithilfe eines Arrays:
  - Man speichert die Elemente in einem Array folgendermaßen:
    - Wurzel an der ersten Position
    - Für einen Knoten an Position i:
      - Vater-Knoten an Position $\lfloor i/2 \rfloor$
      - Linkes Kind an Position 2 * i
      - Rechtes Kind an Position 2 * i + 1
    - Man braucht spezielle Werte um eine leere Position zu markieren

ADT Binary Tree - Repräsentierung mithilfe eines Arrays



Pos	Elem
1	4
2	3
3	1
4	2
5	5
6	-1
7	9
8	-1
9	-1
10	6
11	-1
12	-1
13	-1
…	…

- Nachteil: es hängt von der Form des Baumes ab, aber man kann viel Speicherplatz verschwenden

ADT Binary Tree – Repräsentierung II

- **Verkettete Repräsentierung mit dynamischer Allokation**:
  - Man braucht eine Struktur für den Knoten, welche Folgendes enthält:
    - Information des Knotens,
    - die Adresse des linken Kindes und
    - die Adresse des rechten Kindes
    - (möglicherweise auch die Adresse des Vaterknotens)

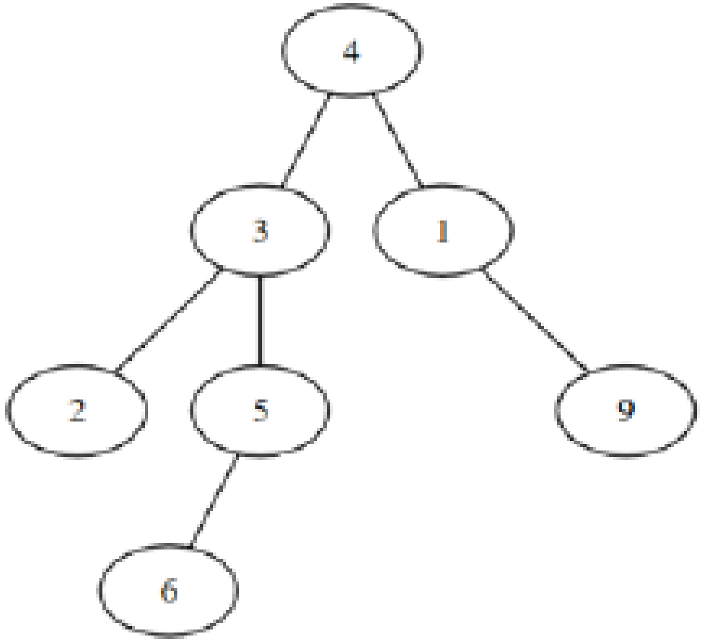
  - Ein leerer Baum enthält den Wert NIL in der Wurzel

  - Es gibt einen Knoten für jedes Element in dem Baum

ADT Binary Tree – Repräsentierung III

- **Verkettete Repräsentierung auf Arrays**:
  - Die Information aus dem Knoten wird in einem Array gespeichert.
  - Die Adresse des linken und rechten Kindes sind die **Indexe** wo sich diese Elemente befinden.
  - Man kann einen separaten Array für die Vaterknoten haben

ADT Binary Tree – Verkettete Repräsentierung auf Array



Pos	1	2	3	4	5	6	7	8
Info	4	3	2	5	6	1	9	
Left	2	3	-1	5	-1	-1	-1	
Right	6	4	-1	-1	-1	7	-1	
Parent	-1	1	2	2	4	1	6	

root = 1

cap = 8

firstEmpty = 8

- Man muss wissen, dass sich die Wurzel an der ersten Position befindet (es kann auch eine andere Position sein)
- Falls der Array voll ist, kann man diesen vergrößern
- Man kann eine verkettete Liste mit den leeren Positionen speichern, damit es einfacher ist einen neuen Knoten einzufügen

ADT Binary Tree – Verkettete Repräsentierung auf Array

Pos	1	2	3	4	5	6	7	8
Info								
Left	2	3	4	5	6	7	8	-1
Right								
Parent								

firstEmpty = 1

root = -1

cap = 8

- Die Liste der leeren Positionen wird am Anfang erstellt (s. SLLA)
- Auch wenn Binärbäume keine linearen Strukturen sind, kann man die *left* und/oder *right* Arrays benutzen um eine einfache oder doppelt verkettete Liste von Positionen zu erstellen

Traversieren von Binärbäumen

- Für einen Binärbaum eignen sich vier Traversierungs-Algorithmen:
  - Traversieren in Präordnung (preorder)
  - Traversieren in Inordnung/symmetrischer Reihenfolge (inorder)
  - Traversieren in Postordnung (postorder)
  - Traversieren in Level-Ordnung (Breadth-first)

Traversieren von Binärbäumen

- Für Binärbäume eignen sich insbesondere rekursive Traversierungs-Algorithmen.
- Die ersten drei Vorgehensweisen beruhen auf der Aufteilung eines Binärbaumes in drei Teile:
  - Wurzel
  - Den linken Teilbaum
  - Den rechten Teilbaum
- Mit dem erzeugten Teilbaum wird in gleicher Weise verfahren

Binärbaum – Repräsentierung

- Für die Traversierungs-Algorithmen benutzen wir folgenden Repräsentierung des Binärbaumes:

<u>BTNode:</u>
info: TElem
left: ↑ BTNode
right: ↑ BTNode

<u>BinaryTree:</u>
root: ↑ BTNode

Präordnung, Inordnung, Postordnung

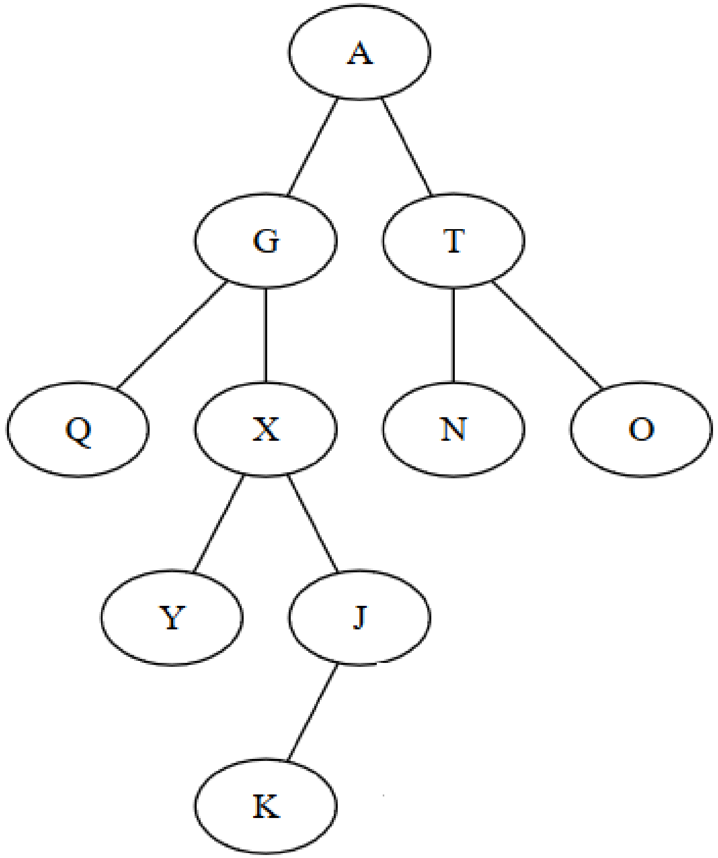
- Wie kann man sich am einfachsten der Unterschied zwischen den Traversierungs-Methoden merken?
  - Der linke Teilbaum wird immer vor dem rechten besucht
  - Die Wurzel wird aber in unterschiedlicher Reihenfolge besucht:
    - **Prä**ordnung – besuche die Wurzel **vor** dem linken und rechten Teilbaum
    - **In**ordnung – besuche die Wurzel **zwischen** dem linken und rechten Teilbaum
    - **Post**ordnung – besuche die Wurzel **nach** dem linken und rechten Teilbaum

Traversieren in Präordnung (preorder)

- Präordnung Reihenfolge:
  - Besuche Wurzel
  - Durchlaufe linken Teilbaum (falls vorhanden)
  - Durchlaufe rechten Teilbaum (falls vorhanden)

- Bei der Traversierung der Teilbäume gilt dieselbe Reihenfolge

Traversieren in Präordnung - Beispiel



- Traversieren in Präordnung: A, G, Q, X, Y, J, K, T, N, O

Traversieren in Präordnung – rekursive Implementierung

```
subalgorithm preorder_recursive(node) is:
//pre: node ist ein ↑ BTNode
    if node ≠ NIL then
        @visit [node].info
        preorder_recursive([node].left)
        preorder_recursive([node].right)
    end-if
end-subalgorithm
```

Traversieren in Präordnung – rekursive Implementierung

- Der *preorder_recursive* Algorithmus hat einen Pointer zu einem Knoten als Parameter, man braucht also ein Wrapper Algorithmus, der ein *BinaryTree* als Eingabeparameter hat:

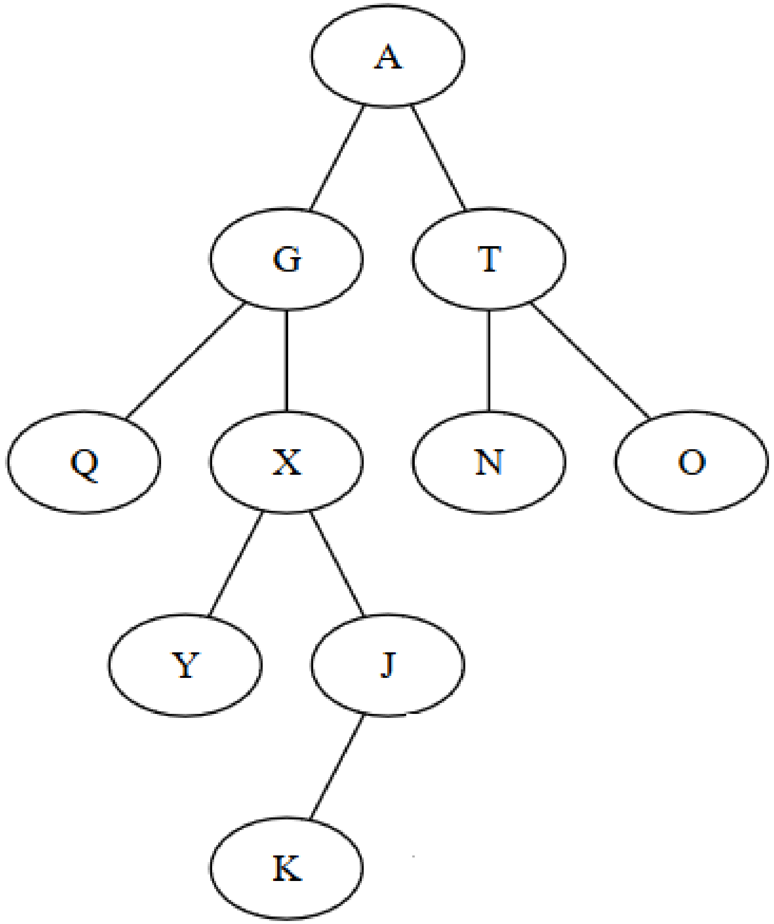
```
subalgorithm preorderRec(tree) is:
//pre: tree ist ein BinaryTree
    preorder_recursive(tree.root)
end-subalgorithm
```

- Wenn man annimmt, dass ein Knoten in konstanter Zeit besucht wird (z.B. die Information aus dem Knoten auszudrucken), dann braucht man für die Traversierung eines Binärbaumes mit n Knoten $\Theta(n)$ Zeit

Traversieren in Präordnung – nicht-rekursive Implementierung

- Man kann den Traversierungs-Algorithmus in Präordnung auch ohne Rekursion implementieren, mithilfe eines zusätzlichen **Stacks** wo man die Knoten speichert
  - Man fängt mit einem leeren Stack an
  - Füge die Wurzel des Baumes zu dem Stack ein
  - Solange der Stack nicht leer ist:
    - Pop einen Knoten und besuche ihn
    - Füge das rechte Kind des Knotens in den Stack ein
    - Füge das linke Kind des Knotens in den Stack ein

Traversieren in Präordnung – nicht-rekursive Implementierung - Beispiel



- **Stack: A**
- Besuche A, push das rechte und das linke Kind (**Stack: T G**)
- Besuche G, push das rechte und das linke Kind (**Stack: T X Q**)
- Besuche Q, push das rechte und das linke Kind (**Stack: T X**)
- Besuche X, push das rechte und das linke Kind (**Stack: T J Y**)
- Besuche Y, push das rechte und das linke Kind (**Stack: T J**)
- Besuche J, push das rechte und das linke Kind (**Stack: T K**)
- Besuche K, push das rechte und das linke Kind (**Stack: T**)
- Besuche T, push das rechte und das linke Kind (**Stack: O N**)
- Besuche N, push das rechte und das linke Kind (**Stack: O**)
- Besuche O, push das rechte und das linke Kind (**Stack:**)
- Stack ist leer

Traversieren in Präordnung – nicht-rekursive Implementierung

subalgorithm preorder(tree) **is**:
//pre: tree ist ein Binärbaum
    s: Stack //s ist ein Hilfs-Stack
    if tree.root ≠ NIL **then**
        push(s, tree.root)
    end-if
    while not isEmpty(s) **execute**
        currentNode ← pop(s)
        @visit currentNode
        if [currentNode].right ≠ NIL **then**
            push(s, [currentNode].right)
        end-if
        if [currentNode].left ≠ NIL **then**
            push(s, [currentNode].left)
        end-if
    end-while
end-subalgorithm

Traversieren in Präordnung – nicht-rekursive Implementierung

- Die Zeitkomplexität der non-rekursiven Implementierung für die Traversierung in Präordnung ist $\Theta(n)$
- Man braucht O(n) zusätzlicher Speicherplatz für den Stack

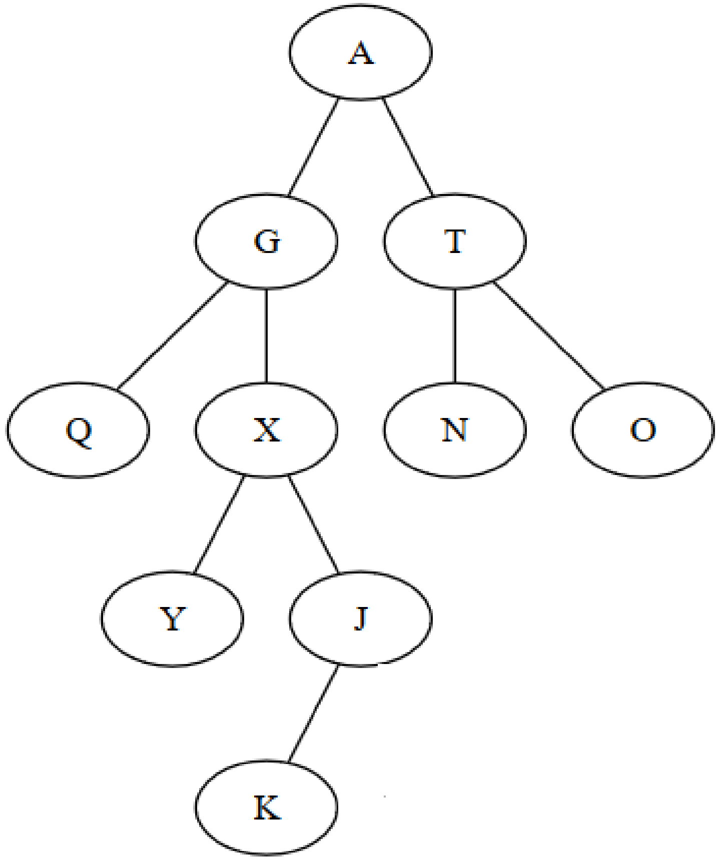
- Bem. Traversieren in Präordnung ist genau dassselbe wie Traversieren in die Tiefe mit der Bemerkung, dass wir hier darauf achten müssen, zuerst das das rechte Kind in dem Stack einzufügen und dann das linke (im Fall von Traversieren in die Tiefe ist die Reihenfolge, in der wir die Kinder einfügen, nicht so wichtig)

Traversieren in symmetrischer Ordnung (inorder)

- Inorder Reihenfolge:
  - Durchlaufe linken Teilbaum
  - Besuche Wurzel
  - Durchlaufe rechten Teilbaum

- Bei der Traversierung der Teilbäume gilt dieselbe Reihenfolge

Traversieren in symmetrischer Ordnung (inorder) - Beispiel



- Traversieren in Inorder: Q, G, Y, X, K, J, A, N, T, O

Traversieren in symmetrischer Ordnung (inorder) – rekursive Implementierung

subalgorithm inorder_recursive(node) **is**:

//pre: node ist ein ↑ BTNode

    if node ≠ NIL **then**

        inorder_recursive([node].left)

        @visit [node].info

        inorder_recursive([node].right)

    end-if

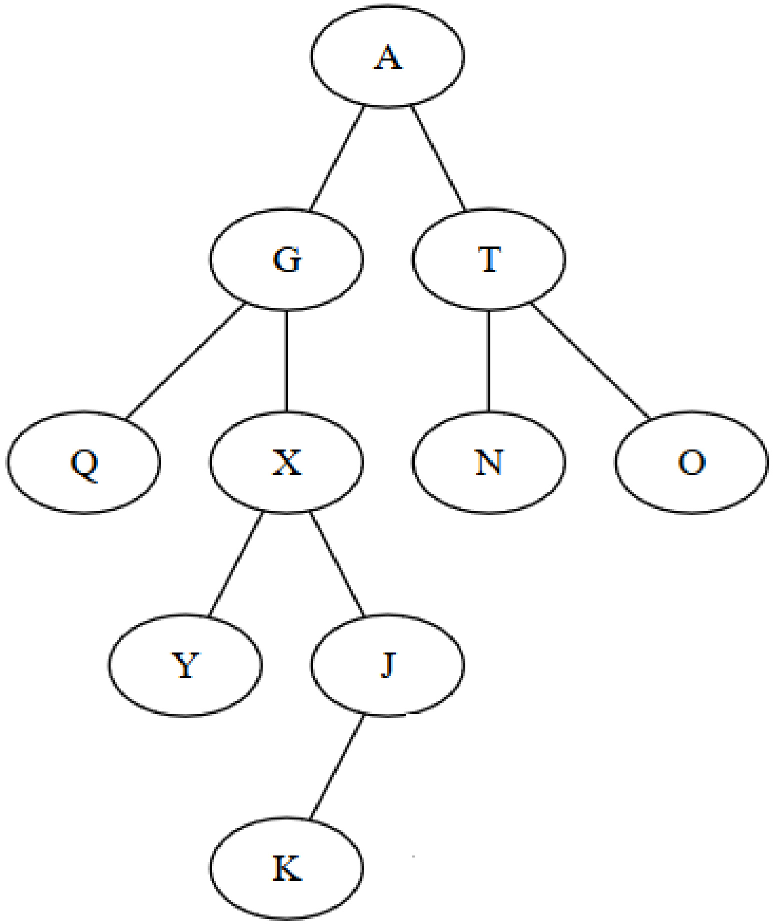
end-subalgorithm

- Man braucht wieder einen Wrapper-Algorithmus
- Die Traversierung braucht $\Theta(n)$ Zeit für einen Binärbaum mit n Knoten

Traversieren in symmetrischer Ordnung (inorder) – nicht-rekursive Implementierung

- Man kann den Traversierungs-Algorithmus in Inorder auch ohne Rekursion implementieren, mithilfe eines zusätzlichen **Stacks** wo man die Knoten speichert
  - Man fängt mit einem leeren Stack und einem *currentNode* an
  - *currentNode* wird mit der Wurzel initialisiert
  - Solange *currentNode* nicht NIL ist, füge *currentNode* zu dem Stack ein und setze es auf das linke Kind
  - Solange der Stack nicht leer ist:
    - Pop einen Knoten und besuche ihn
    - Setze *currentNode* auf das rechte Kind
    - Solange *currentNode* nicht NIL ist, füge ihn zu dem Stack ein und setze ihn auf das linke Kind

Traversieren in Inorder – nicht-rekursive Implementierung - Beispiel



- CurrentNode: A, Stack:
- CurrentNode: NIL, Stack: A G Q
- Besuche Q, currentNode: NIL, Stack: A G
- Besuche G, currentNode: X, Stack: A
- CurrentNode: NIL, Stack: A, X, Y
- Besuche Y, CurrentNode: NIL, Stack: A X
- Besuche X, CurrentNode: J, Stack: A
- CurrentNode: NIL, Stack: A J K
- Besuche K, CurrentNode: NIL, Stack: A J
- Besuche J, CurrentNode: NIL, Stack: A
- Besuche A, CurrentNode: T, Stack:
- CurrentNode: NIL, Stack: T N
- ...

Solange *currentNode* nicht NIL ist, füge *currentNode* zu dem Stack ein und setze es auf das linke Kind

Solange der Stack nicht leer ist:

- Pop Knoten und besuche
- Setze *currentNode* auf das rechte Kind
- Solange *currentNode* nicht NIL ist, füge ihn zu dem Stack ein und setze ihn auf das linke Kind

Traversieren in Inordnung – nicht-rekursive Implementierung

```
subalgorithm inorder(tree) is:
//pre: tree ist ein Binärbaum
    s: Stack //s ist ein Hilfs-Stack
    currentNode ← tree.root
    while currentNode ≠ NIL execute
        push(s, currentNode)
        currentNode ← [currentNode].left
    end-while
    while not isEmpty(s) execute
        currentNode ← pop(s)
        @visit currentNode
        currentNode ← [currentNode].right
        while currentNode ≠ NIL execute
            push(s, currentNode)
            currentNode ← [currentNode].left
        end-while
    end-while
end-subalgorithm
```

Traversieren in Inordnung – nicht-rekursive Implementierung

- Zeitkomplexität: $\Theta(n)$
- Speicherplatzkomplexität: $O(n)$

Denke nach

- Wenn es einen Binärbaum gibt, aber man kennt nichts darüber außer der Traversierungen in Präordnung und Inordnung, kann man den Binärbaum mithilfe der zwei Traversierungs-Methoden aufbauen?
- Zum Beispiel:
  - Präordnung: A B F G H E L M
  - Inordnung: B G F H A L E M
- Kann man den Binärbaum mithilfe der Postordnung und Inordnung Traversierungen aufbauen?
- Kann man den Binärbaum mithilfe der Präordnung und Postordnung Traversierungen aufbauen?