universitätbonn

Algorithmen auf Listen und Bäumen

Anwendungen von Zeigerstrukturen

Einfach verkettete Liste

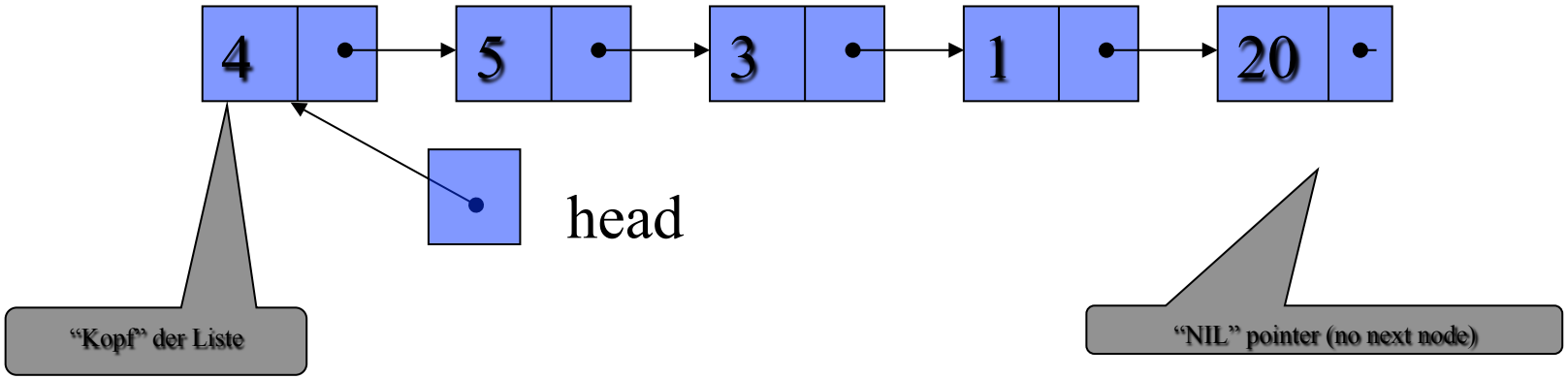
- Zur Erinnerung
  - Einfach verkettete Liste einfachster Fall einer „rekursiven“ Datenstruktur
  - Repräsentieren im folgenden Liste durch Zeigervariable auf den Kopf (head) der Liste

- Liste von Integer in C

```
typedef struct IntNode {
 struct IntNode *next;
 int data;
 } IntNode;

IntNode *head;
```

Schematisches Beispiel:



Einfach verkettete Listen

- Funktionen und Prozeduren auf Listen erhalten daher Zeigervariable auf den Kopf der Liste als Parameter
  - Diese verändern im Allgemeinen die Liste, auch wenn der Kopf der Liste nicht verändert wird
  - Verzeigerte Datenstrukturen sind mächtiges aber auch komplexes Programmierparadigma
    - ð Und finden sich in (fast) allen imperativen Programmiersprachen
      - – Und auch in objektorientierten Sprachen

Einfach verkettete Listen: Einfügen am Anfang

- Funktion „insertFirst“
  - Verändert in jedem Fall den Kopf der Liste
  - Benötigt nur sehr wenige Operationen
    - ð Konstante Anzahl
      - – Unabhängig von der Länge der Liste
  - Realisieren insertFirst als Funktion
    - ð Zeiger auf Anfang der neuen Liste wird zurückgegeben
    - ð „Alte Liste“ wird nicht verändert

Einfach verkettete Listen: Einfügen am Anfang

- Programmcode

```
/*We insert a node with data d at the first position of
 list given by head and return the reference to the new list.
 The old list is not modified.
 Code is correct for special case of empty list.*/

 IntNode * insertFirst( IntNode *head, int d){
 IntNode *tmp;
  tmp= malloc(sizeof(IntNode));
   tmp->data = d;
   tmp->next = head;
   return tmp;
}
```

Ausgeben der Inhalte einer Liste

- Benutzen zur Ausgabe einer Liste die „LISP-Konvention“
  - Listen werden durch Klammernpaar beschränkt
  - Elementе werden durch Leerzeichen („Blank“, ein „white space“-Character getrennt)
    - ð Leere Liste: ( )

Ausgeben der Inhalte einer Liste

```
/*printList prints the content of the list to output. The list is enclosed in
 brackets and the elements are  seperated by a blank. */

void printList(IntNode *head){
IntNode *cursor;
printf ("(");
cursor=head;
  while (cursor != NULL)
  {
     printf("%d",cursor->data);
     printf(" ");
     cursor=cursor->next;
  }
printf(")");
}
```

Einfügen am Ende: insertLast

- Prozedur zum Einfügen am Ende „inhaltlich“ ähnlich zum Einfügen am Anfang
- Bei der Realisierung treten aber wichtige (und subtile) Unterschiede zu Tage
  - Ursprungsliste wird immer verändert
    - ð Hat also „Seiteneffekte“
  - Aufwand abhängig von der Länge der Liste
    - ð Man muss sich erst einmal ans „Ende hangeln“

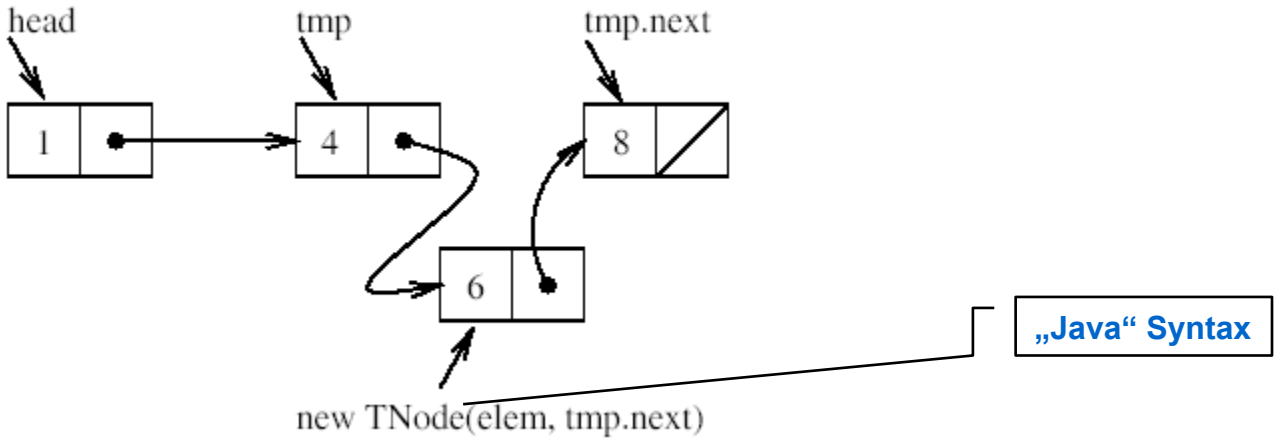
Einfügen am Ende: insertLast

```
/*insertLast inserts a node with data d after the last position of
 list given by head. The given list is modified.  The reference to the
 head of the modified list is returned.*/

IntNode * insertLast(IntNode *head, int d)
 {
   IntNode *tmp, *cursor;
    //trivial case: empty list
    if (head==NULL) {return insertFirst(head,d);}
    else{
    //general case: goto end
      cursor=head;
      while (cursor->next != NULL){cursor =cursor->next;}
      tmp= malloc(sizeof(IntNode));
      tmp->data = d;
      tmp->next = NULL;
      cursor->next=tmp;
      return head;
 }
}
```

Geordnetes Einfügen: insertSorted

- Annahme: Liste ist sortiert
  - Bei der integer-Liste gemäß der natürlichen Ordnung auf den Integern
  - In Listen über allgemeinen Daten bräuchte man noch eine spezielle Vergleichsoperation
    - ð Oder ein „Schlüsselfeld“ vom Typ integer
  - Einfügen eines neuen Datenfeldes in die erste mögliche Position, so dass Liste sortiert bleibt



Geordnetes Einfügen: insertSorted

```
/*insertSorted inserts a node with data d before the first node with
data field being bigger (in integer order). head can be changed
(if d is inserted in first position) */
IntNode * insertSorted(IntNode *head, int d){

 IntNode *cursor, *tmp;
//trivial case: empty list or insertion before first element
  if (head==NULL){head=insertFirst(head,d);}
  else if (head->data > d){ head=insertFirst(head,d);}
  else{
    //general case: goto right postion.
```

Geordnetes Einfügen: insertSorted (Forts.)

```
   //general case: goto right postion
     cursor=head;
     while (cursor->next != NULL && (cursor->next)->data <= d)
           cursor=cursor->next;
     //insert new node
     // works also if new node is inserted  after last node
     tmp= malloc(sizeof(IntNode));
     tmp->data = d;
     tmp->next = cursor->next;
     cursor->next=tmp;
    }
  return head;
}
```

Geordnetes Einfügen: insertSorted rekursiv

- Struktur
  - Vergleiche das erste Element der Liste (head->data) mit dem einzufügenden Element d
  - Falls die Liste leer ist oder das neue Element kleiner ist, dann füge das neue Element als erstes in die Liste ein
    - ð Ist dann ein insertFirst
  - Andernfalls füge das neue Element sortiert in eine neue, um den Kopf verkürzte Liste head->next ein (rekursiver Aufruf)
    - ð und gebe den Kopf der Liste zurück

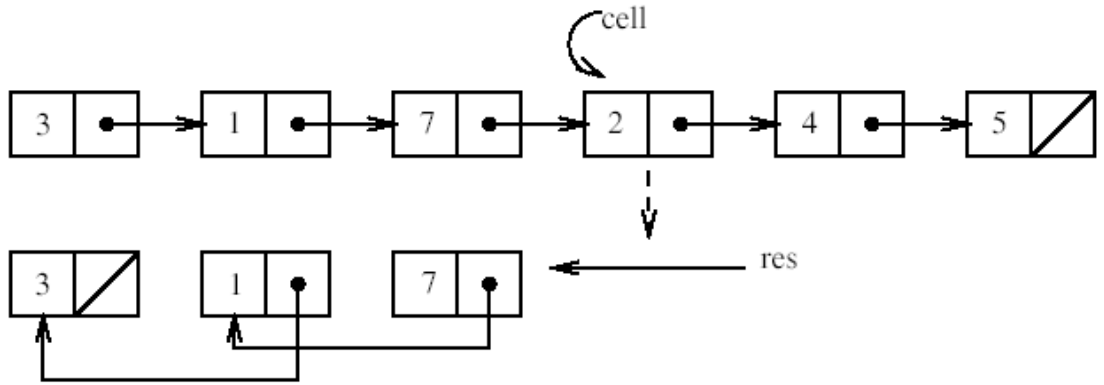
Geordnetes Einfügen: insertSorted rekursiv

```
/*insertSortedRec inserts a node with data d before the first node
withdata field being bigger (in integer order). returns head of new list
and changes old list. Recursvie implementation.*/

IntNode *insertSortedRec(IntNode *head, int d) {
   //trivial case: empty list or d is smallest element
   IntNode *ret;
   if (head==NULL){ret=insertFirst(head,d);}
   else if ((head->data)>d) {ret=insertFirst(head,d);}
   else{
      head->next=insertSortedRec(head->next,d);
      ret=head;
         }
 return ret;
}
```

Konstruktives Invertieren einer Liste

- Konstruktives Invertieren:
  - Alte Liste bleibt erhalten
  - Die invertierte Liste ist eine modifizierte Kopie der alten Liste

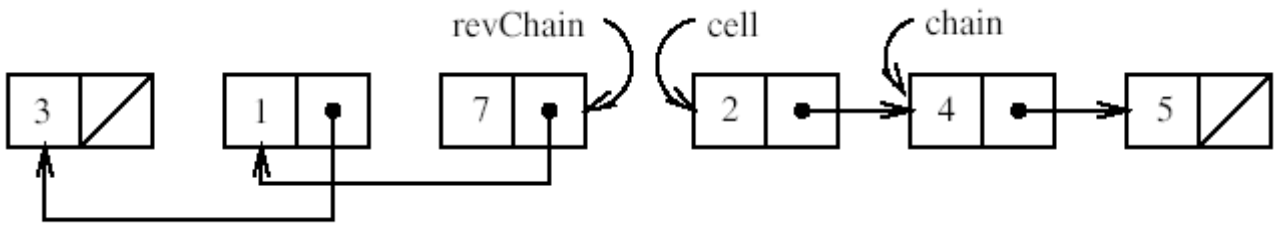


Konstruktives Invertieren einer Liste

```
/*reverseListCon returns reference to the head of
a constructively inverted copy of argument list.*/
IntNode *reverseListCon(IntNode *head){
IntNode *res, *tmp, *cell;
cell=head; res=NULL;
   while (cell != NULL) {
    tmp= malloc(sizeof(IntNode));
       tmp->data=cell->data;
       tmp->next=res;
       res=tmp;
       cell=cell->next;
}
return res;
}
```

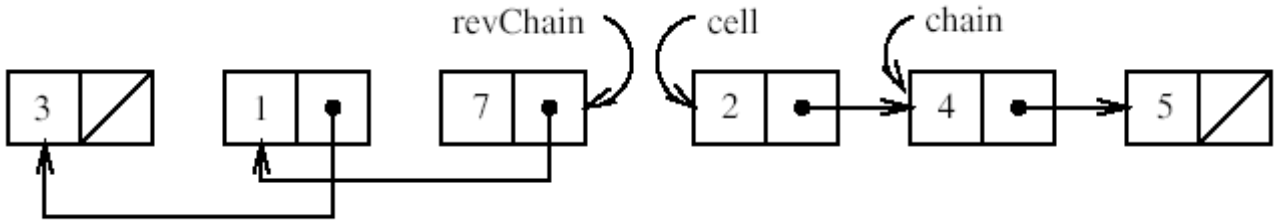
Destruktives Invertieren einer Liste

- Liste wird umgebaut
- Verwenden daher Referenzparameter auf Listenanfang
  - Der nach Aufruf von reverseList auf den Anfang der invertierten Liste zeigt



Destruktives Invertieren einer Liste

```
//reverseList destructively inverts list.
void reverseList(IntNode * head){
IntNode *revChain, *chain, *cell;
cell=head;
  revChain=NULL;
   while (cell != NULL) {
       //set chain to remaining node chain
       chain=cell->next;
       //reverse next pointer in cell
       cell->next=revChain;
       //add cell to revChain
        revChain=cell;
       //set cell to next cell in chain
       cell=chain;
    }
  //finalize
  head=revChain;
}
```



Löschen einer Liste

- Cerfordert explizite Freigabe des Speichers von verketteten Strukturen
  - Ebenso wie dies in C++ und PASCAL notwendig ist
  - Java (und funktionale Sprachen):
    - ð Automatisches reklamieren nicht mehr verwendeter Strukturen
      - – Garbage collection

Löschen einer Liste

- Folgender einfacher Code ist sehr gefährlich
  - Wenn auf Teillisten mehrfach verwiesen wird
    - ð Wie dies bei komplexen Programmen häufig vorkommt

```
/*deleteList deletes the list starting with head and frees the memory.  After completion head is nil. Procedure
crashes if head starts not a proper list, i.e. if there are cycles.  Also references to substructures are invalidated*/
void deleteList(IntNode * head){
IntNode *cursor, *last;
  //init
  cursor=head;
  while (cursor != NULL){
     last=cursor;
     //go to next before deletion
     cursor = cursor->next;
     free(last);
   }
  //finalize
  head=NULL;
}
```

Referenz auf Anfang der Liste wird geändert

Löschen einer Liste

- Beispiel für Problem

```
list1=NULL;
list1=insertFirst(list1, 3);
list1=insertFirst(list1, 5);
list2=insertFirst(list1,7);
deleteList(list1);

//now list2 is also invalid
printList(list2); //crashes or, even worse, returns nonsense!
```

Löschen einer Liste

- Lösung für Problem
  - Verwende „garbage collection“
    - ð Java und funktonale Sprache (LISP; Haskell, …) unterstützen solche
    - ð Kann vom Anwendungsprogrammierer nicht in vollständige sicherere Art und Weise erzwungen werden
      - – „Garbage collection“ libraries in C/C++ nicht vollständig sicher

Löschen einer Liste

- Lösung für Problem
  - Verwende „Reference Counts“
    - Jeder Listenknoten enthält noch einen counter, der die Zugriffe auf diesen Knoten von verschiedenen Startreferenzen aus zählt
    - Erst bei Referenz 0 wird wirklich ein dispose durchgeführt
    - Erfordert Reimplimentierung von InsertFirst (und copy, …)
      - Statt nur eine Startreferenz zu kopieren, muss immer die ganze Liste besucht werden, um die Counts hochzusetzen
    - Wird trotzdem häufig verwendet
      - Zum Beispiel in dem in C geschriebenen Kernel verschiedener Computeralgebra-Systeme
        - Mathematica, MuPad

Verkettete Listen vs. Arrays

- Listen
  - Vorteile
    - Einfügen neuer Elemente leicht möglich
    - Löschen leicht möglich
  - Nachteile
    - „Durchhangeln“ durch viele Elemente der Liste, um ein spezielles zu erreichen
      - Etwa das „Fünfte in der Liste“
    - Zusätzlicher Speicherbedarf
      - Eine Referenz pro Listenelement
- Arrays
  - Nachteile
    - Einfügen neuer Elemente erfordert „umkopieren“
    - Löschen von Elementen erfordert ebenfalls ein „umkopieren“
  - Vorteile
    - Wahlfreier Zugriff
    - Speicherbedarf nur für Daten
      - Nur insgesamt pro Array noch Speicher-Bedarf für ein length-Feld

universitätbonn

Binärbäume

Binärbäume

- Die Generalisierung von verketteten Listen mit einem Nachfolger next auf zwei Nachfolger führt zu einer der wichtigsten Datenstrukturen der Informatik
  - Vielleicht sogar zu „der wichtigsten“
  - Binärbäume
    - ð Da Aufgrund der zwei Nachfolger zu vielen (Such-)Algorithmen Datenstrukturen aufgebaut werden können, die 2^n mal „schnellere“ Suchen erlauben als bei Verwendung von Listen oder Arrrays

Binärbäume

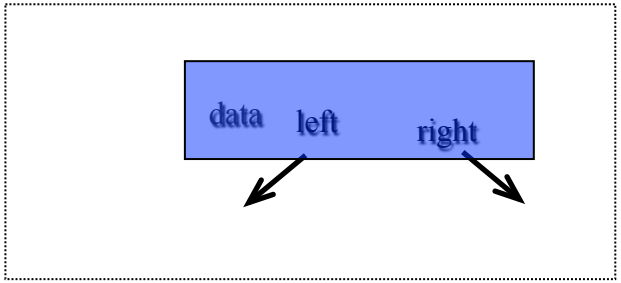
- Sprechweise etwas verschieden von verketteten Listen
  - Statt head spricht man von „Wurzel“ (root)
  - Und die zwei Nachfolger werden im Allgemeinen als linkes und rechts Kind bezeichnet
  - Und statt vom (implizit) gegebenen Vorgänger vom „Elternknoten“ (parent)

Binärbäume

- Datentyp eines Binärbaums (über integer) in C

```
typedef struct IntTreeNode {
 struct IntTreeNode *left, *right ;
 int data;
 } IntTreeNode;

IntTreeNode *iroot;
```



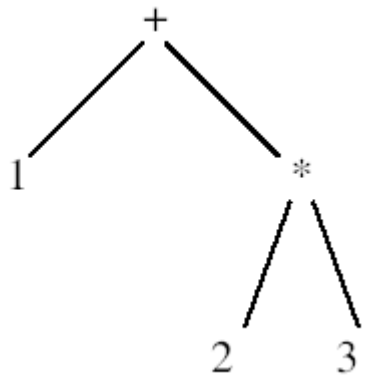
Datentyp eines Binärbaums (über char) in C

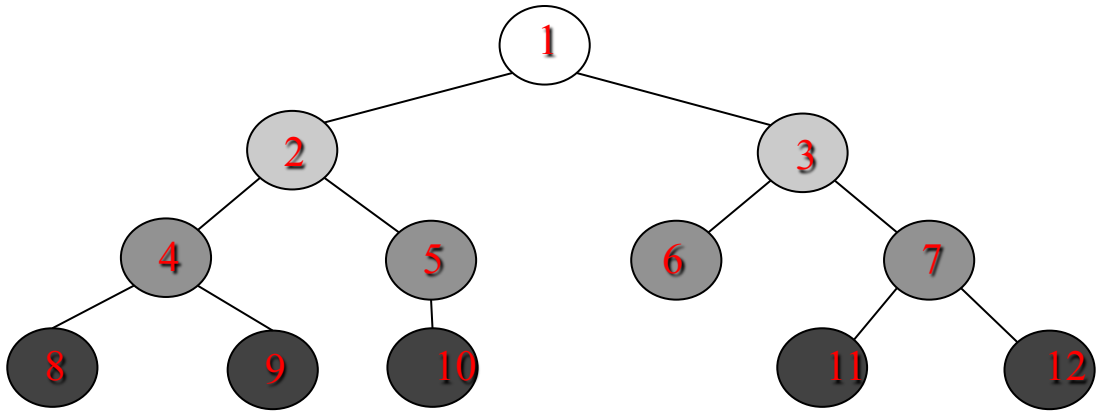
```
typedef struct CharTreeNode {
 struct CharTreeNode *left, *right ;
 int data;
 } CharTreeNode;

CharTreeNode *croot;
```

Binärbäume

- Beispiele für Binärbäume
  - Über integer und char



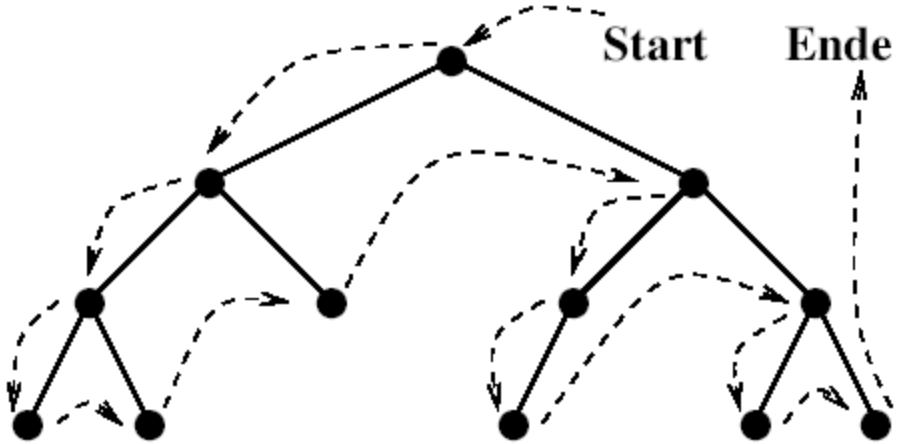


Binärbäume

- Baumdurchläufe
  - Bei Listen gibt es nur eine „natürliche“ Möglichkeit des Durchlaufs
    - ð Beginne bei head und hangle Dich von einem Element zum nächsten
  - Bei Bäumen gibt es hier schon sehr viel mehr Möglichkeiten
    - ð Die je nach Anforderungen des Algorithmus, in denen sie eingesetzt werden, auch alle vorkommen
    - ð Wollen uns im folgenden auf Durchläufe mit „Tiefensuche“ beschränken
    - ð Die sehr elegant rekursiv realisiert werden können

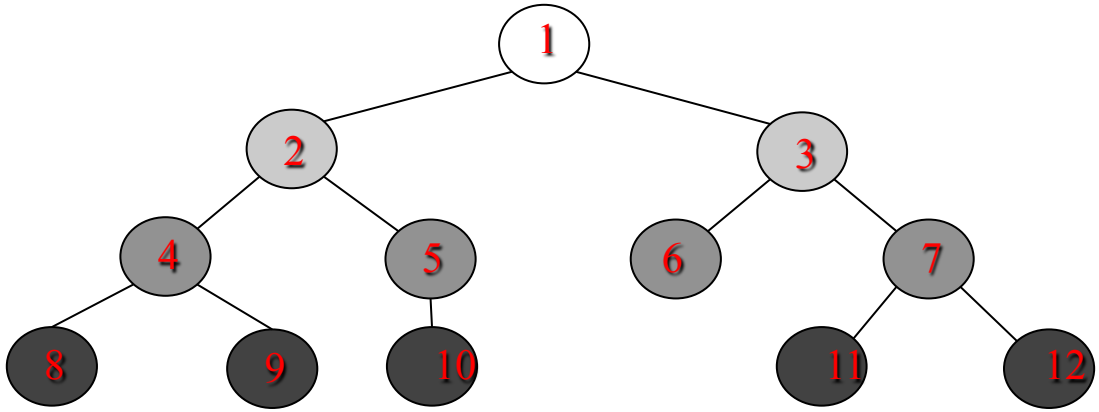
Baumdurchläufe: Präorder

- Das abstrakte Verfahren zum Durchlauf in Präorder lautet folgendermaßen:
  1. Betrachte die Wurzel des Baums (und führe eine Operation auf ihr aus)
  2. Durchlaufe den linken Teilbaum
  3. Durchlaufe den rechten Teilbaum



Tree Traversals: Preorder

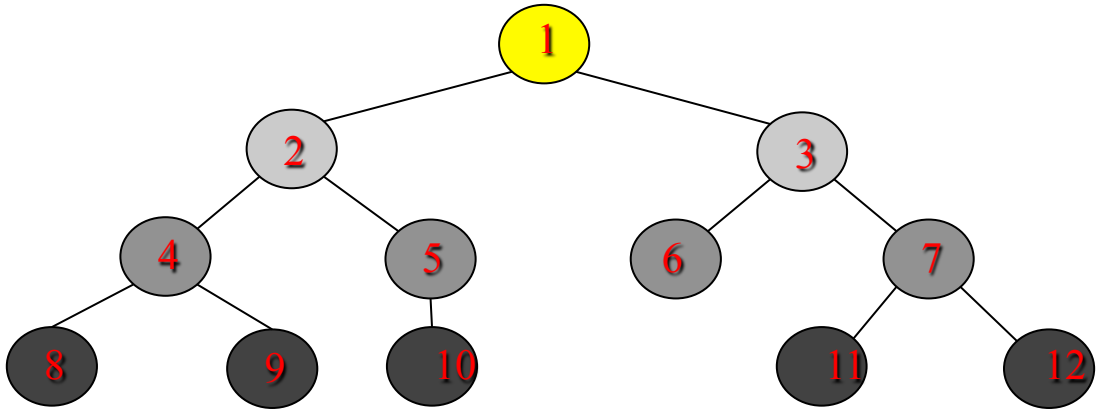
- Example: printing the node numbers in preorder



  - Result:

Tree Traversals: Preorder

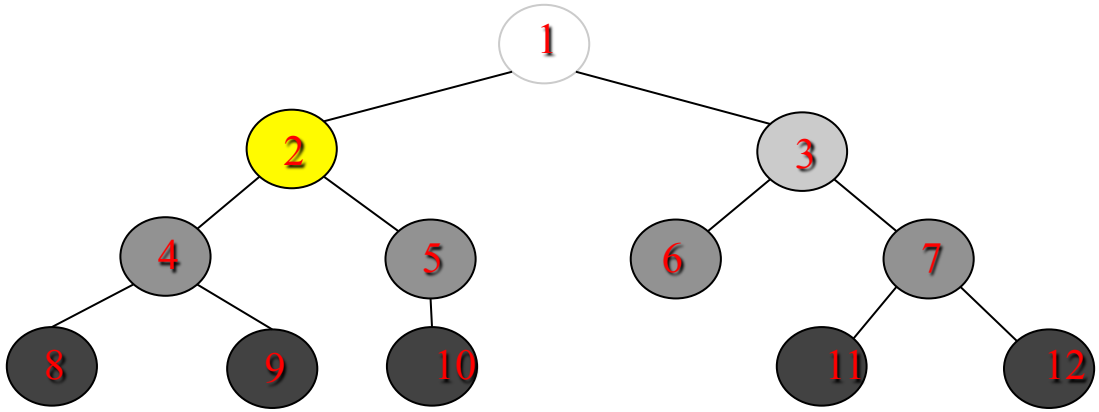
- Example: printing the node numbers in preorder



  - Result: 1

Tree Traversals: Preorder

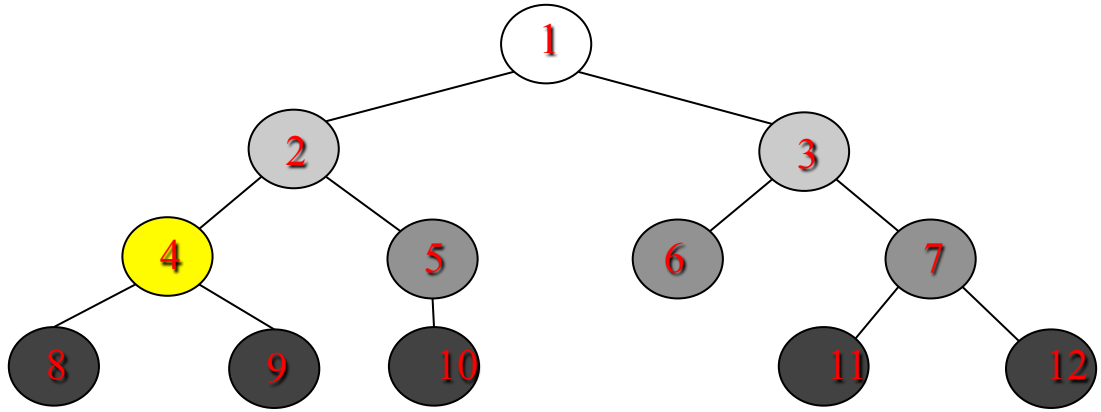
- Example: printing the node numbers in preorder



  - Result: 1,2

Tree Traversals: Preorder

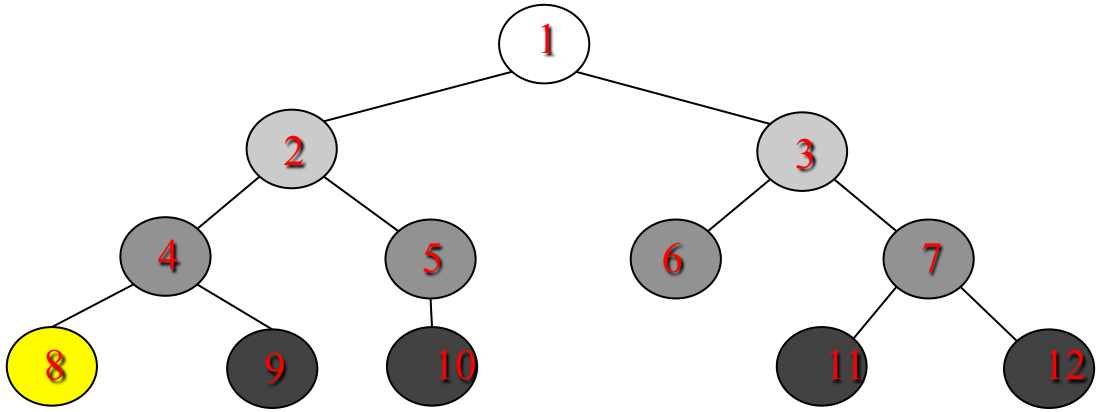
- Example: printing the node numbers in preorder



  - Result: 1,2,4

Tree Traversals: Preorder

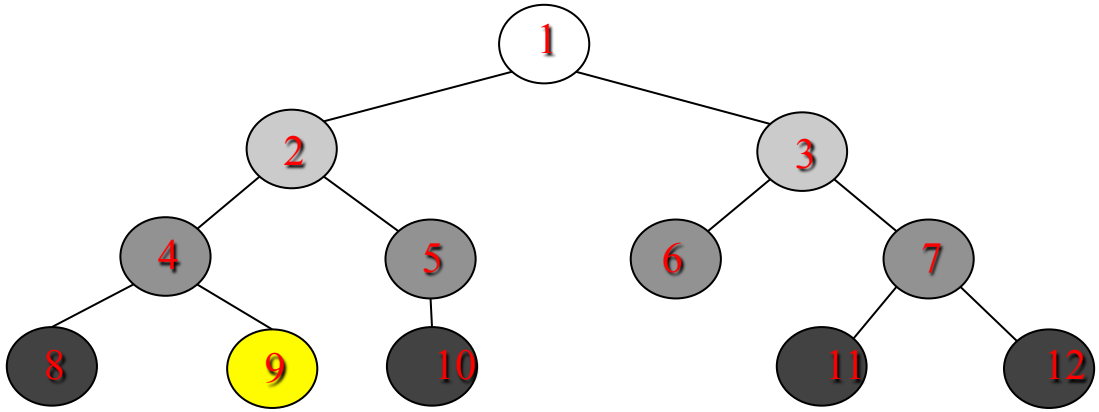
- Example: printing the node numbers in preorder



  - Result: 1,2,4,8

Tree Traversals: Preorder

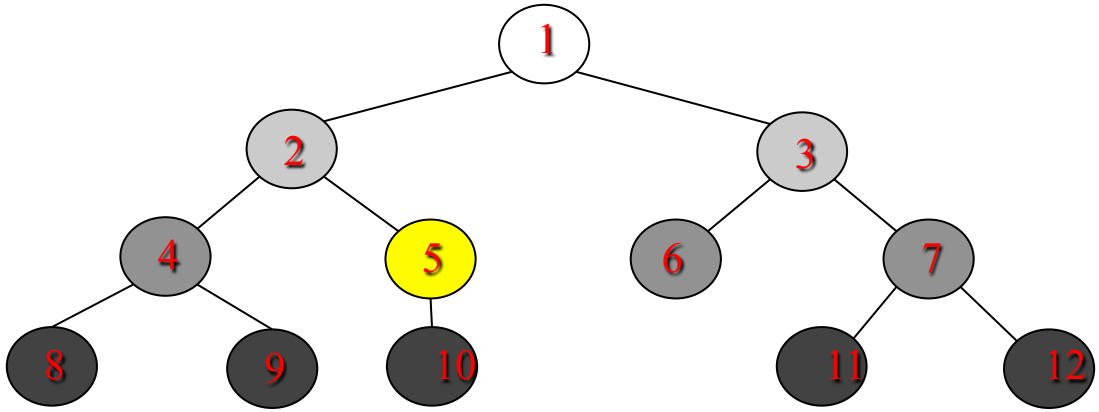
- Example: printing the node numbers in preorder



  - Result: 1,2,4,8,9

Tree Traversals: Preorder

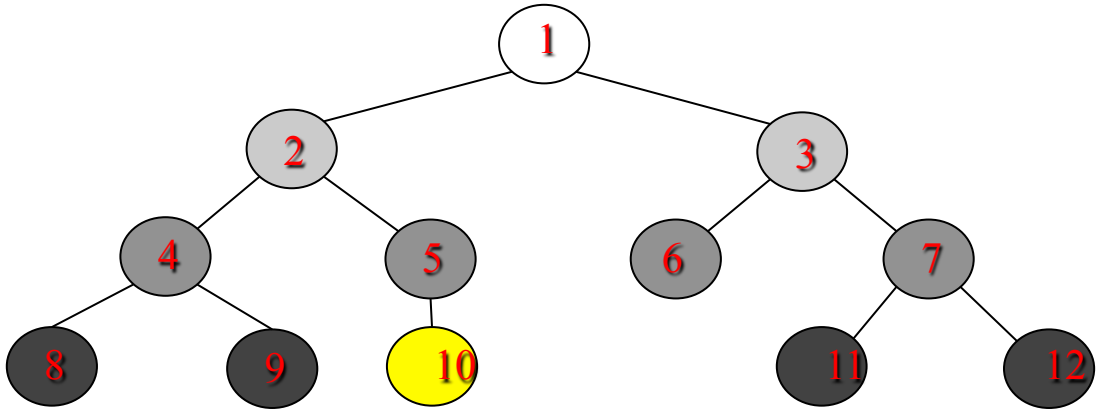
- Example: printing the node numbers in preorder



  - Result: 1,2,4,8,9,5

Tree Traversals: Preorder

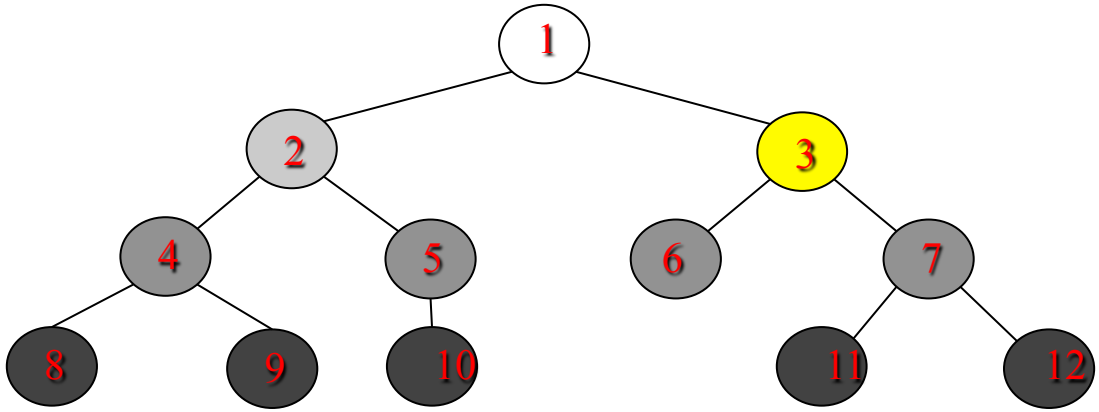
- Example: printing the node numbers in preorder



  - Result: 1,2,4,8,9,5,10

Tree Traversals: Preorder

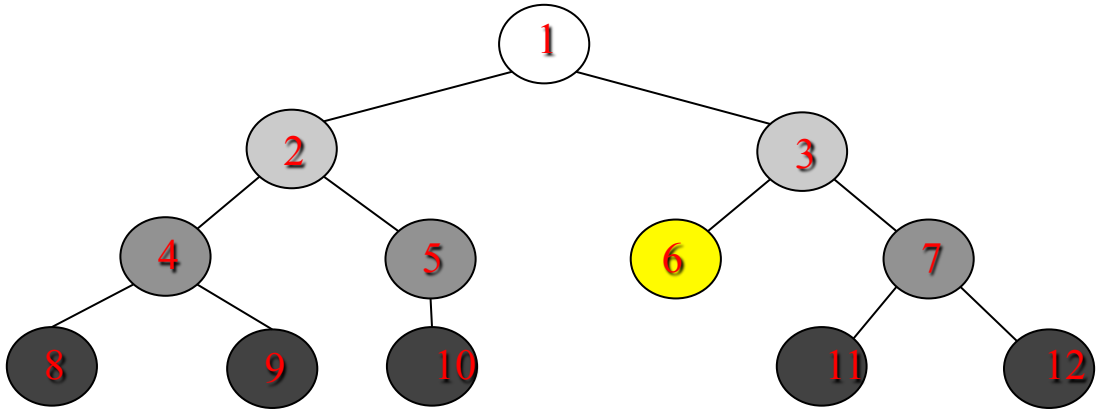
- Example: printing the node numbers in preorder



  - Result: 1,2,4,8,9,5,10,3

Tree Traversals: Preorder

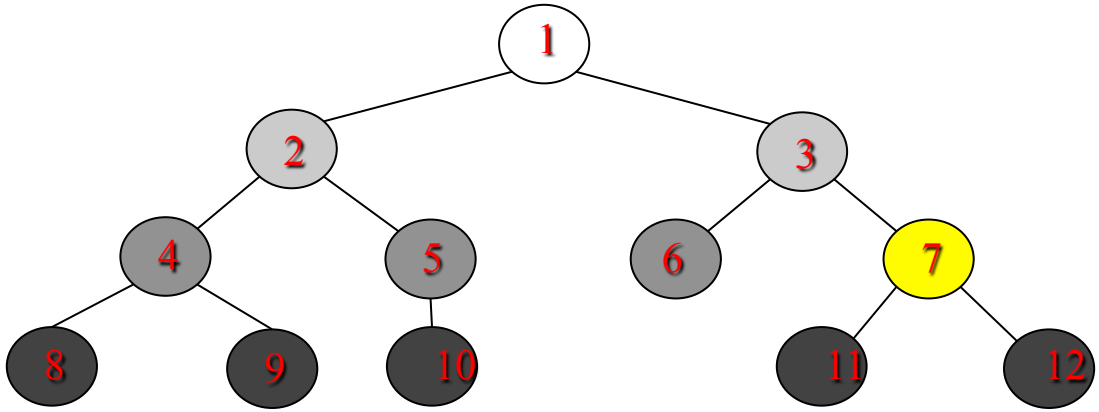
- Example: printing the node numbers in preorder



  - Result: 1,2,4,8,9,5,10,3,6

Tree Traversals: Preorder

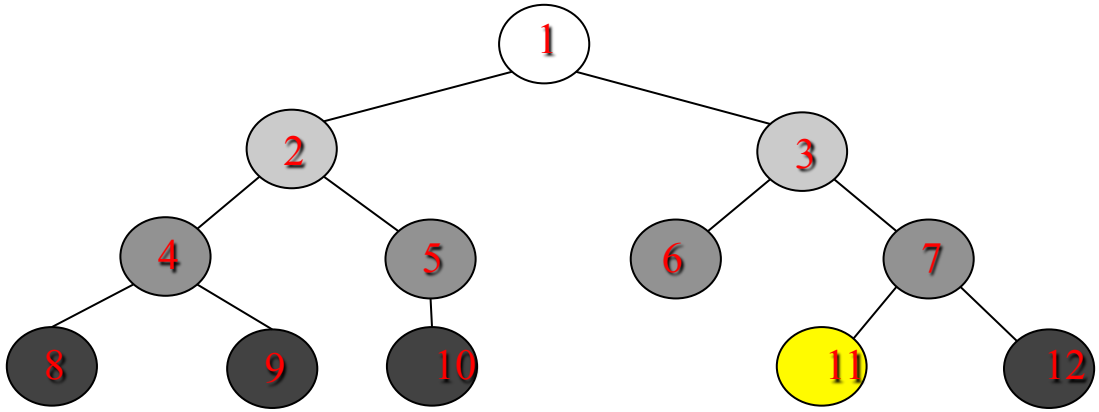
- Example: printing the node numbers in preorder



  - Result: 1,2,4,8,9,5,10,3,6,7

Tree Traversals: Preorder

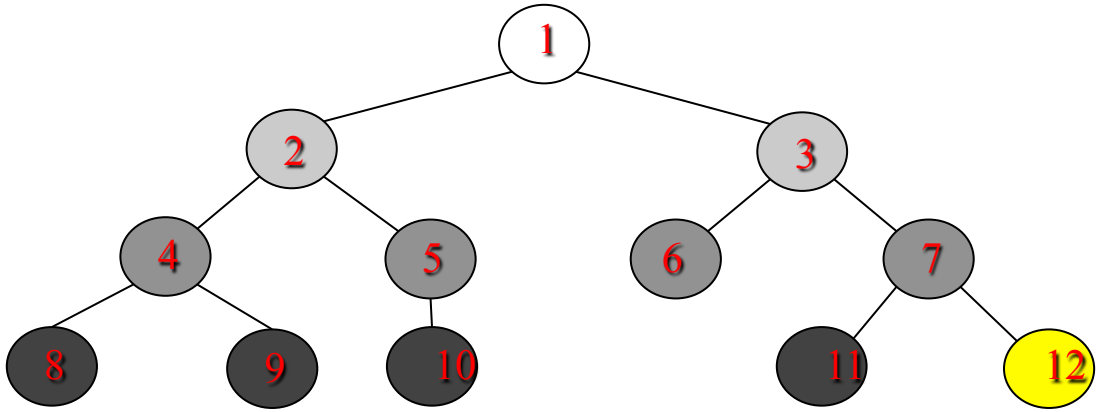
- Example: printing the node numbers in preorder



  - Result: 1,2,4,8,9,5,10,3,6,7,11

Tree Traversals: Preorder

- Example: printing the node numbers in preorder



  - Result: 1,2,4,8,9,5,10,3,6,7,11,12

Baumdurchläufe: Präorder

```
void printPreorder(CharTreeNode *root) {
  if (root != NULL) {
    printf("%c", root->data);  printf(" ");
    printPreorder(root->left);
    printPreorder(root->right);
  }
}
```

Ein Baumdurchlauf für den oben gegebenen Strukturbaum (zu 1+2*3) in Präorder mit der Operation *Drucke Symbol* erzeugt folgende Ausgabe:

+ 1 * 2 3

Die Wiedergabe eines Strukturbaums für einen Ausdruck mit Präorder entspricht der **polnischen Notation** (Polish notation) für Ausdrücke.

Baumdurchläufe: Postorder

Ein Baumdurchlauf für den oben gegebenen Strukturbaum in Postorder mit der Operation *Drucke Symbol* erzeugt folgende Ausgabe:

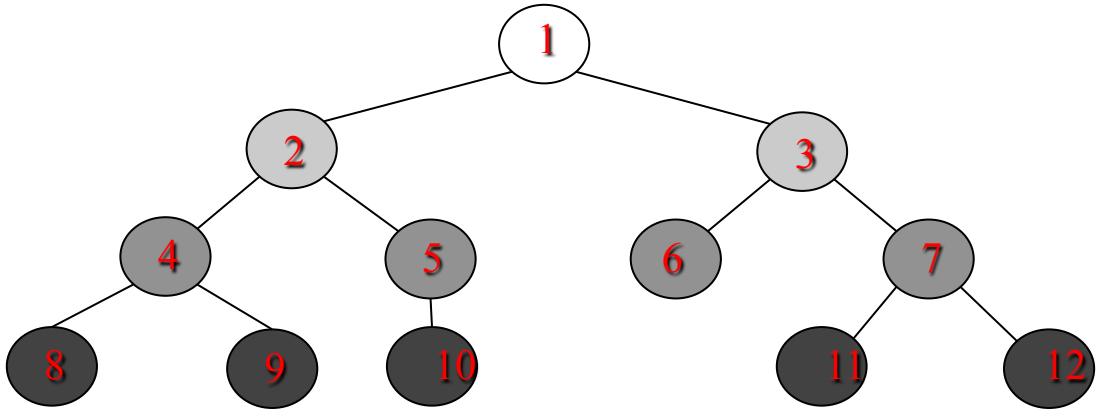
1 2 3 * +

Die Wiedergabe eines Strukturbaums für einen Ausdruck mit Postorder entspricht der **umgekehrten polnischen Notation** (reverse Polish notation) für Ausdrücke.

```
void printPostorder(CharTreeNode *root){
 if (root != NULL){
    printPostorder(root->left);
    printPostorder(root->right);
    printf("%c", root->data);  printf(" ");
  }
}
```

Tree Traversals: Postorder

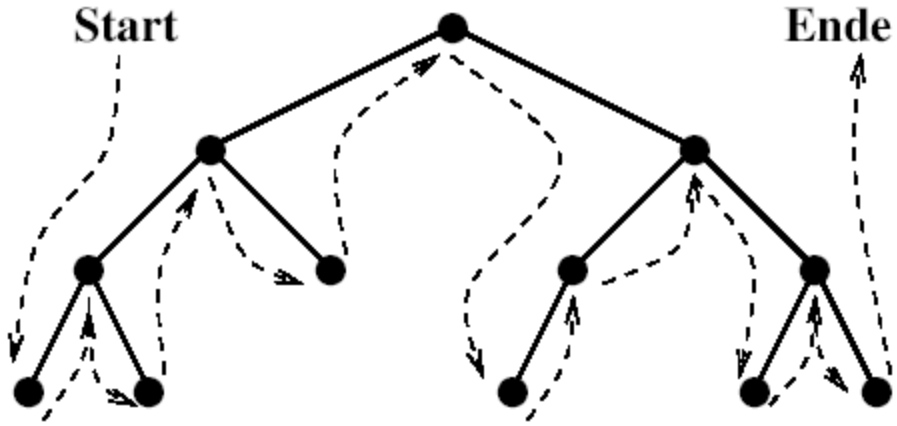
- Example:printing the node numbers in postorder



  - Result: 8,9,4,10,5,2,6,11,12,7,3,1

Baumdurchläufe: Inorder

- Das abstrakte Verfahren zum Durchlauf in Inorder lautet folgendermaßen:
  1. Durchlaufe den linken Teilbaum
  2. Betrachte die Wurzel des Baums (und führe eine Operation auf ihr aus)
  3. Durchlaufe den rechten Teilbaum



Baumdurchläufe: Inorder

```
void printInorder(CharTreeNode *root){
if (root != NULL){
//optional: print brackets for unique readability
    printf("(");
    printInorder(root->left);
    printf(" ");
    printf("%c", root->data);  printf(" ");
    printInorder(root->right);
    printf(")");
  }
}
```

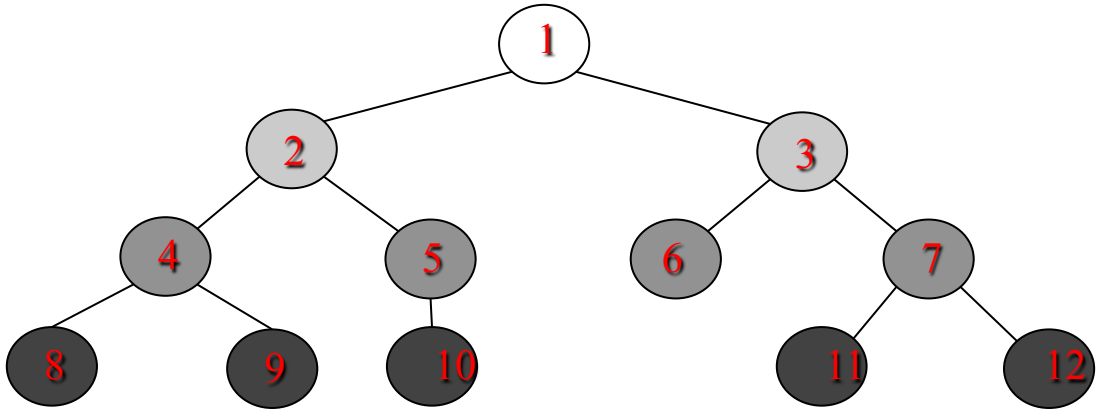
Ein Baumdurchlauf für den oben gegebenen Strukturbaum in Inorder mit der Operation *Drucke Symbol* erzeugt folgende Ausgabe:

1 + 2 * 3

Die Wiedergabe eines Strukturbaums für einen Ausdruck mit Inorder entspricht der normalen Operator-Schreibweise (wenn zusätzlich die Subausdrücke noch geklammert werden)
(1 + (2 * 3)) oder gar
((1) + ((2) * (3)))
[Ausgabe der Prozedur links]

Tree Traversals: Inorder

- Example:printing the node numbers in inorder



  - Result: 8,4,9,2,10,5,1,6,3,11,7,12

Binäre Suchbäume

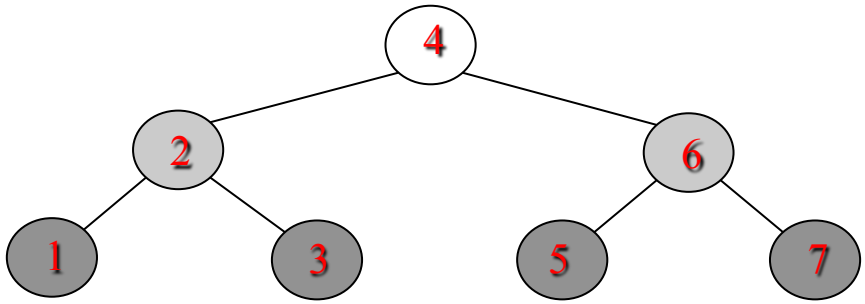
- Schnelle Suche nach Elementen in einer Folge *f* kann durch binäre Suchbäume realisiert wrden
  - Räpresentiere *f* durch Baum, in dessen Knoten die Elemente gespeichert werden, und der folgende Bedingung erfüllt
    - ð Für jeden Knoten
      - – Linker Subbaum enthält nur kleinere Werte
      - – Rechter Subbaum enthält nur größere Werte

Binäre Suchbäume

- Test, ob ein Wert *x* in f vorkommt dann wie folgt
  - Vergleiche mit Wert *y* des momentanen Knotens
    - Wenn *x=y* beende Suche (und antworte “ja”)
    - Traversiere linken Subbaum falls *x<y*
    - Traversiere rechten Subbaum falls *y<x*
  - Wenn kein Subbaum vorhanden, antworte “nein”
- Es handelt sich also um eine Präorder-Traversierung

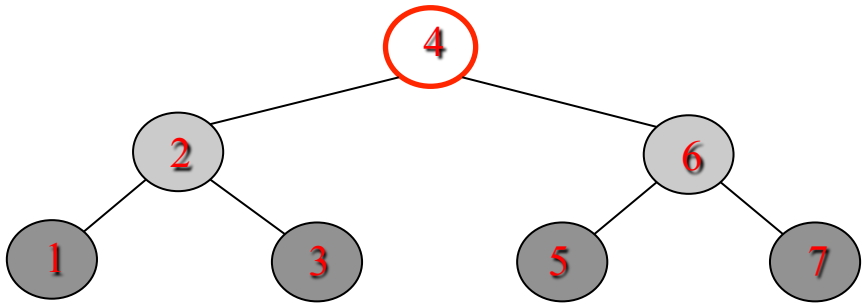
Binäre Suchbäume

- Beispiel:
  - Folge *{1,2,3,4,5,6,7}*
  - Suche nach 3



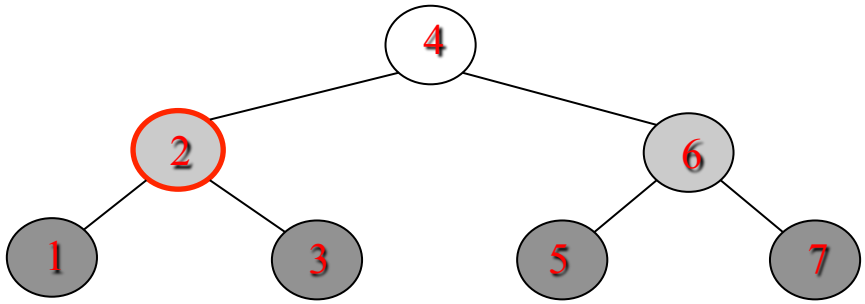
Binäre Suchbäume

- Beispiel:
  - Folge *{1,2,3,4,5,6,7}*
  - Suche nach 3



Binäre Suchbäume

- Beispiel:
  - Folge *{1,2,3,4,5,6,7}*
  - Suche nach 3



Binäre Suchbäume

- Beispiel:
  - Folge *{1,2,3,4,5,6,7}*
  - Suche nach 3