ILV Datenstrukturen und Algorithmen

08: Bäume, Rekursion

Stefan Huber

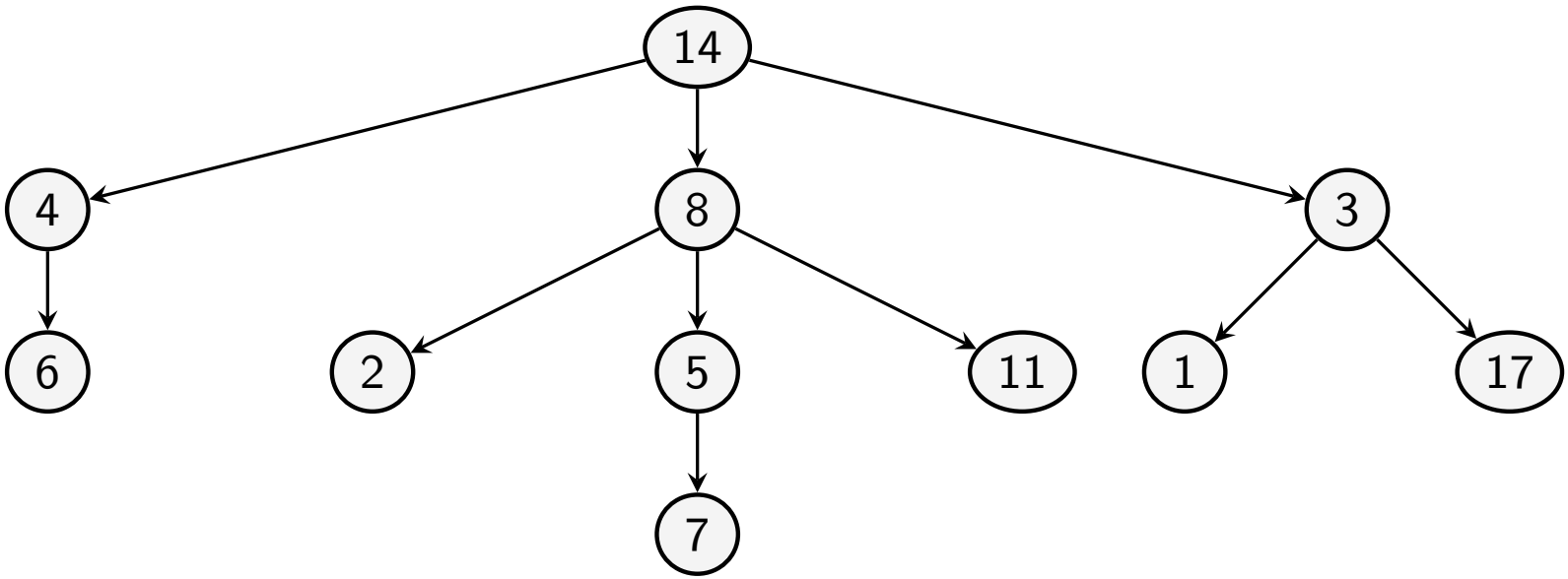
FH Salzburg, Studiengang MMT / 2012

Kapitel

Bäume

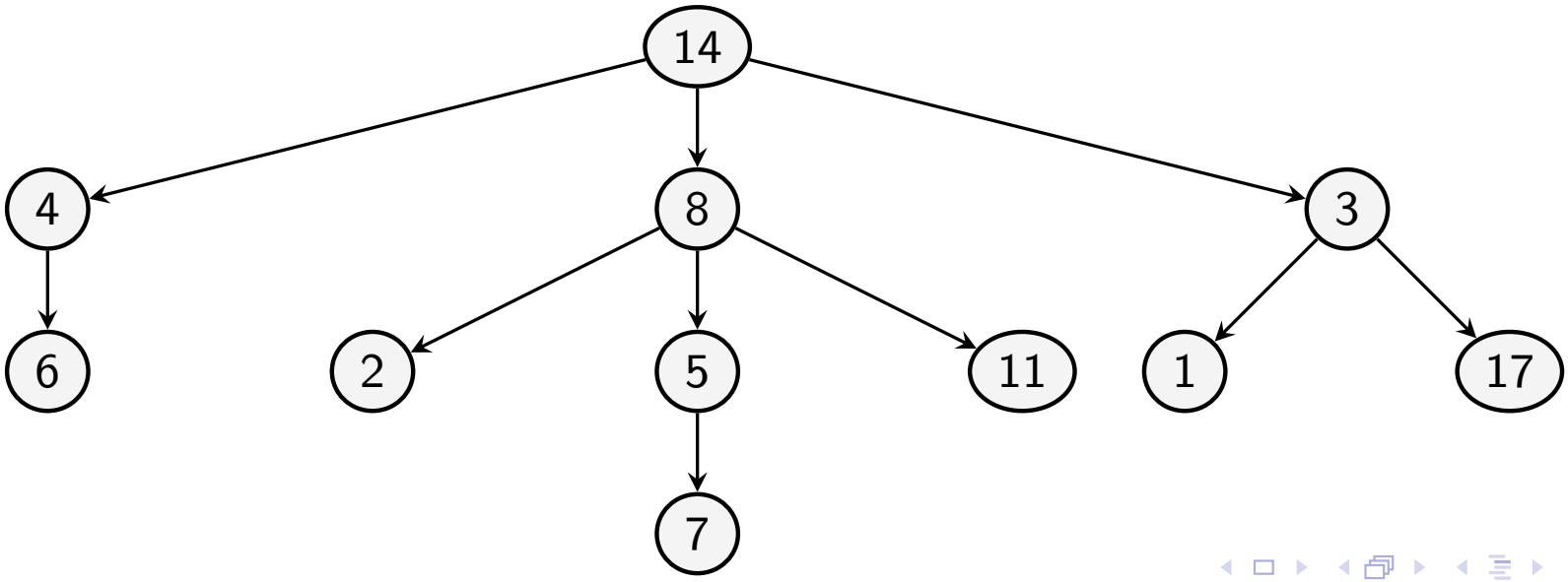
Definitionen

- Ein Baum ist eine hierarchische Datenstruktur:
  - Ein Knoten kann mehrere Nachfolger (*Kinder, children*) haben.
  - Kinderlose Knoten heißen *Blätter (leaves)*: 6, 2, 7, 11, 1, 17
    - Nicht-Blätter heißen *interne (internal) Knoten*: 14, 4, 3, 8, 5
  - Genau ein Knoten, die *Wurzel (root)*, ist selbst kein Kind: 14.
  - Knoten sind durch *Kanten (edges)* verbunden.
  - Ein *Pfad (path)* ist eine Folge von aneinanderhängenden Kanten.
    - Jeder Knoten ist mit der Wurzel durch genau einen Pfad verbunden.



Definitionen

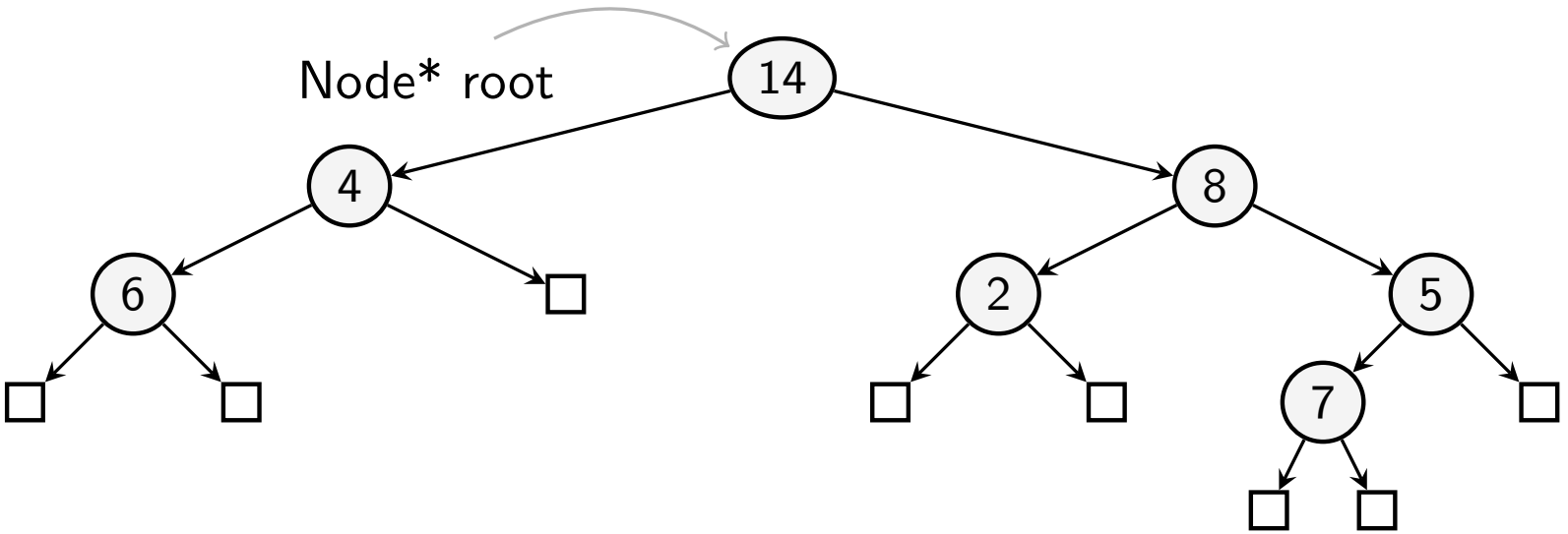
- Die *Höhe* eines Baumes ist die Länge des längsten Pfades.
- Der Knoten p ist der Elternknoten von n, wenn n ein Kind von p ist.
  - Jeder Knoten, bis auf die Wurzel, hat genau einen *Elternknoten (parent)*.
- Knoten mit gleichem Elternknoten sind *Geschwister (siblings)*: 1, 17
- Das *Level* k besteht aus jenen Knoten, die durch Pfade der Länge k von der Wurzel aus erreicht werden: Level 2 besteht aus 6, 2, 5, 11, 1, 17.
- Ein Knoten n ist *unter* einem Knoten p, wenn ein Pfad von p nach n (ggf. der Länge 0) existiert.
  - Alle Knoten unter einem Knoten bilden einen *Teilbaum (subtree)*.



Binärbaum

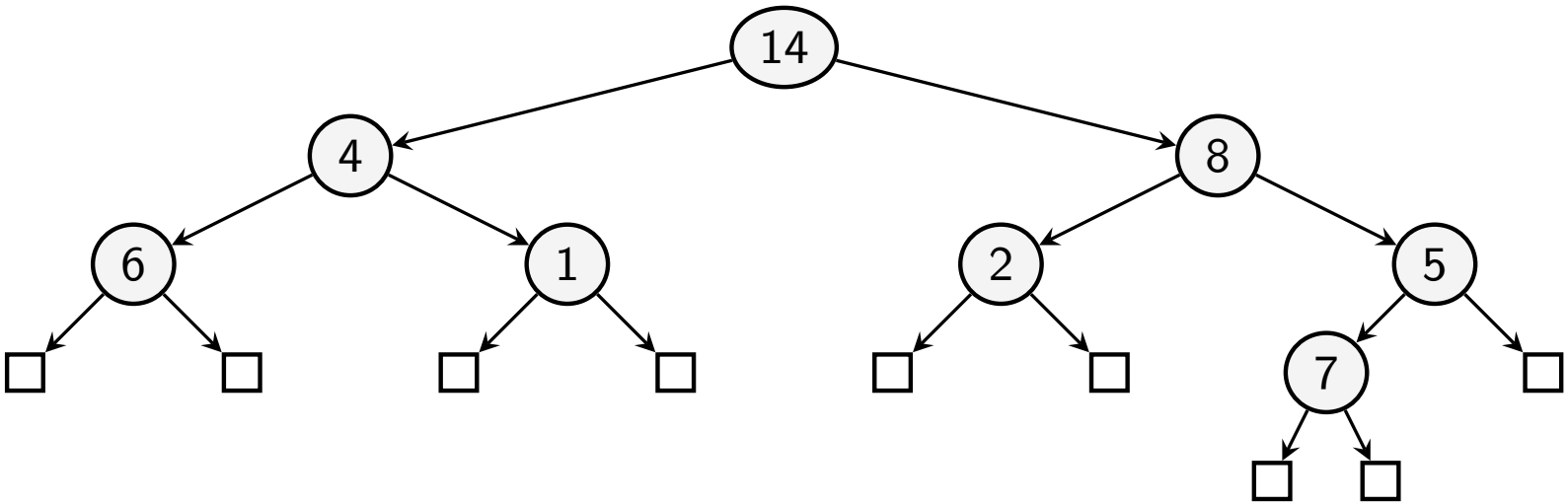
- Ein *Binärbaum* ist ein Baum, bei dem jeder Knoten maximal zwei Kinder hat.
  - Es wird zwischen linkem und rechten Kind unterschieden: Der Knoten 5 hat ein linkes Kind, aber kein rechtes.

```
1 struct Node
2 {
3   int value;
4   Node* left;
5   Node* right;
6 };
```



Binärbaum

- Ein Binärbaum ist *voll (full)*, wenn jedes Level (ggf. bis auf das letzte) vollständig gefüllt ist.
- Ein voller Binärbaum ist *vollständig (complete)*, wenn das letzte Level von links her gefüllt ist.
  - Ein Heap ist stets vollständig.



Eigenschaften

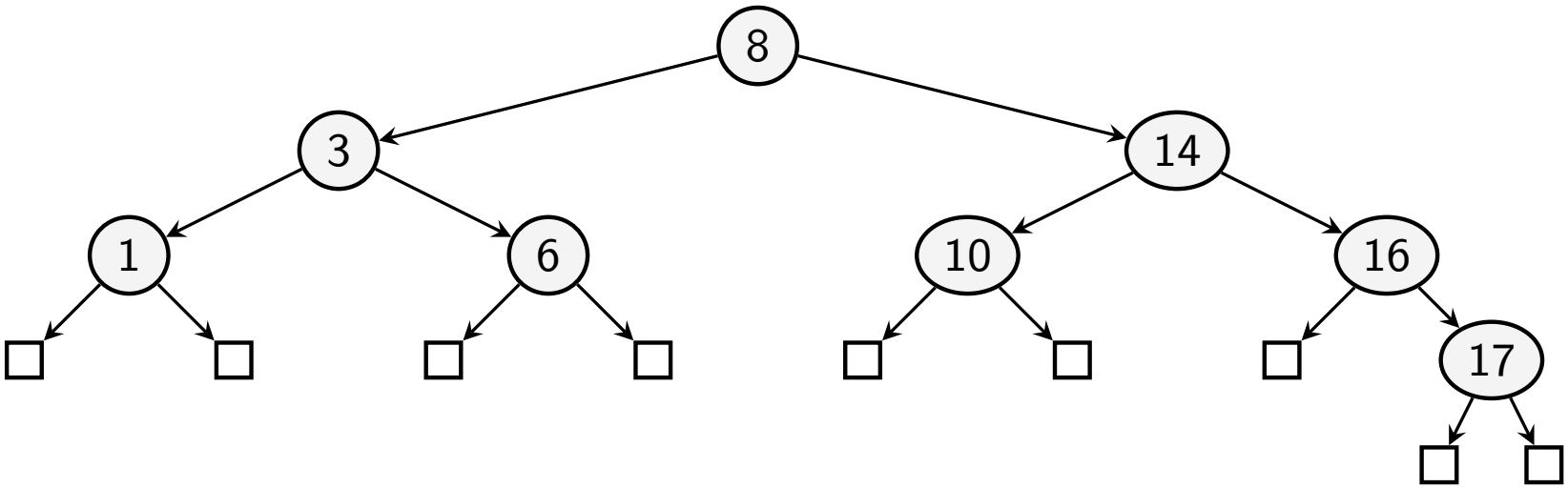
- Ein Baum mit n Knoten enthält $n-1$ Kanten.
- Ein voller Binärbaum der Höhe h, dessen letztes Level ganz gefüllt ist, besitzt $2^{h+1}-1$ Knoten.

$$1+2+4+\cdots+2^h = \sum_{i=0}^{h} 2^i = 2^{h+1}-1$$

- Daher: Ein voller Binärbaum der Höhe h hat mindestens 2^h und maximal $2^{h+1}-1$ Knoten. (Das letzte Level enthält zwischen 1 und 2^h Knoten.)
- Weiters: ein Binärbaum mit n Knoten hat mindestens die Höhe $\log_2(n+1)-1$ und maximal die Höhe $n-1$.

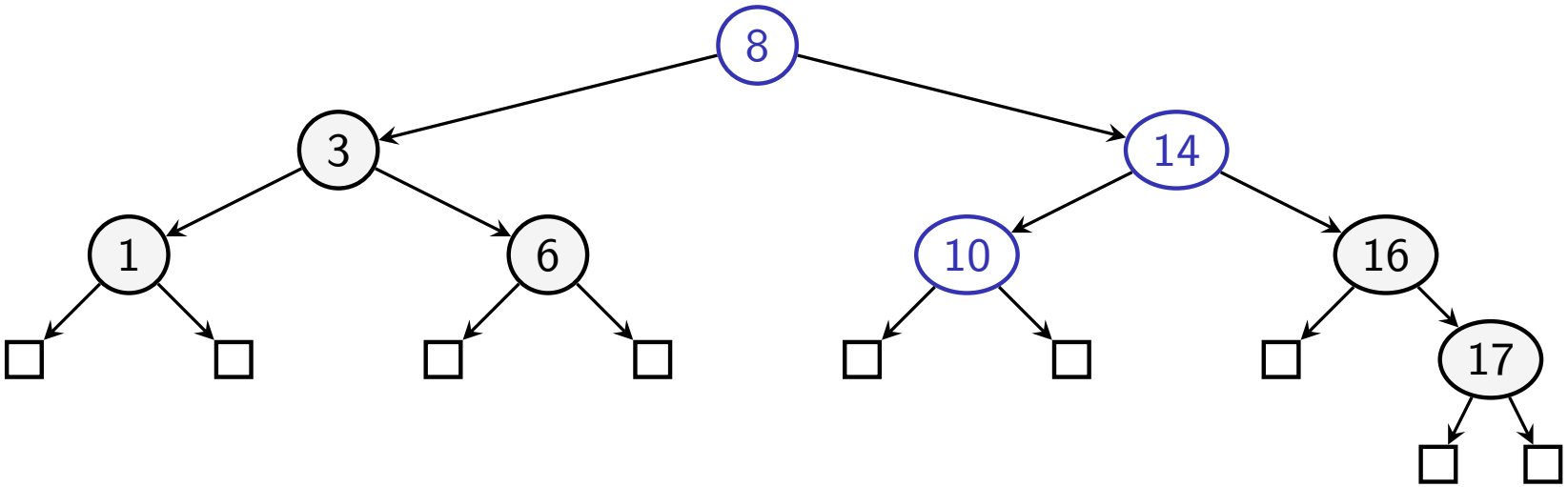
Binärer Suchbaum

- Ein *binärer Suchbaum* ist ein Binärbaum, falls für jeden Knoten N gilt:
  - Die Knoten im linken Teilbaum von N enthalten Werte $\leq$ dem Knoten N.
  - Die Knoten im rechten Teilbaum von N enthalten Werte $\geq$ dem Knoten N.
- Ein binärer Suchbaum ordnet also die Kinder gemäß deren Größe.



Binärer Suchbaum: doesContain

- Wie wollen den Knoten mit dem Wert k, z.B. 10, suchen.
  - Beginne bei der Wurzel und wiederhole:
  - Enthält der aktuelle Knoten den Wert k?
  - Wenn nein, dann gehe nach links oder rechts, je nachdem ob k kleiner oder größer als der Wert im aktuellen Knoten ist.



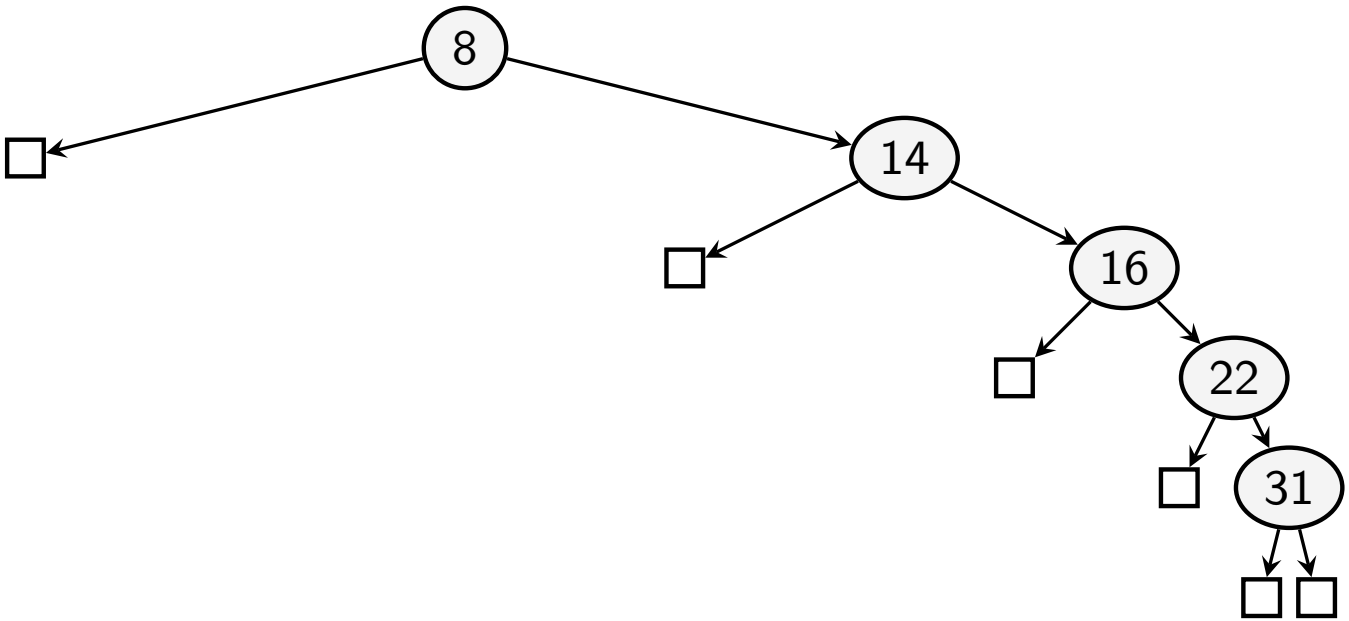
Binärbaum: doesContain

```
1  bool doesContain(int value)
2  {
3    Node* n = getRootNode();
4    while( n != 0 )
5    {
6      if( n->value == value )
7        return true;
8      if( value < n->value )
9        n = n->left;
10     else
11       n = n->right;
12   }
13   return false;
14 }
```

Wie schnell ist doesContain?

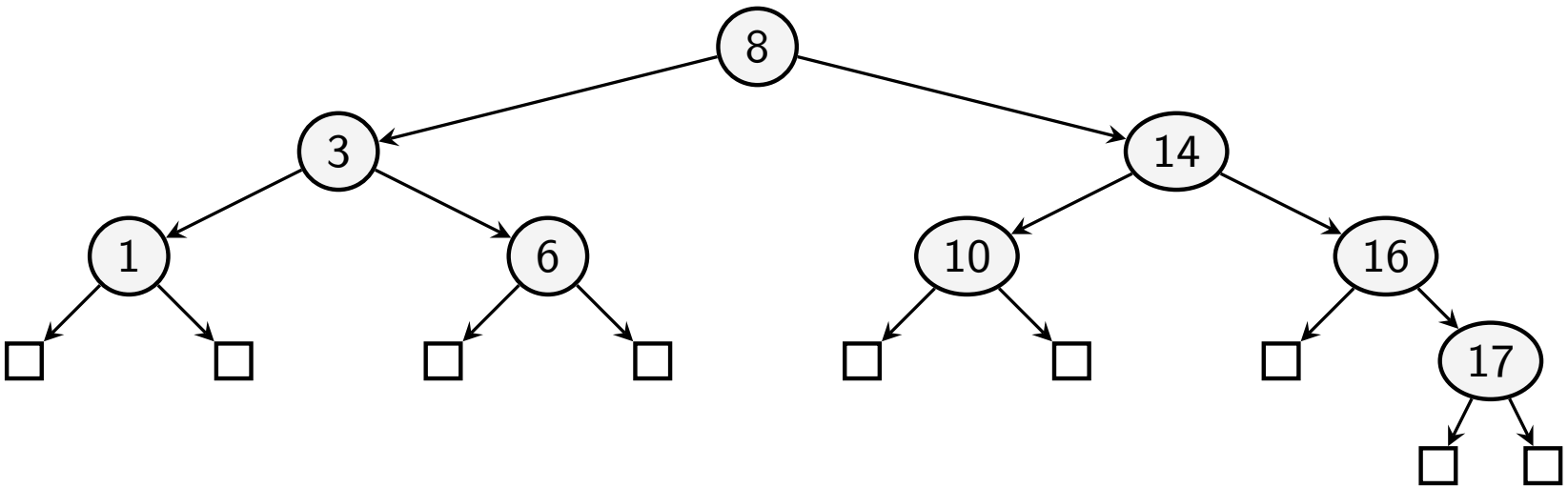
Der Baum enthält n Knoten.

- Im schlimmsten Fall machen wir genau soviele Schritte, wie der Baum hoch ist. (Was wissen wir über die Höhe?)
  - Ist der Baum sehr ausgeglichen, dann ungefähr $\log_2 n$ Schritte.
  - Ist der Baum sehr unausgeglichen, dann ungefähr n Schritte.
- Im allerschlimmsten Fall ist der Binärbaum zu einer Liste *degeneriert* und wir suchen das letzte Element.



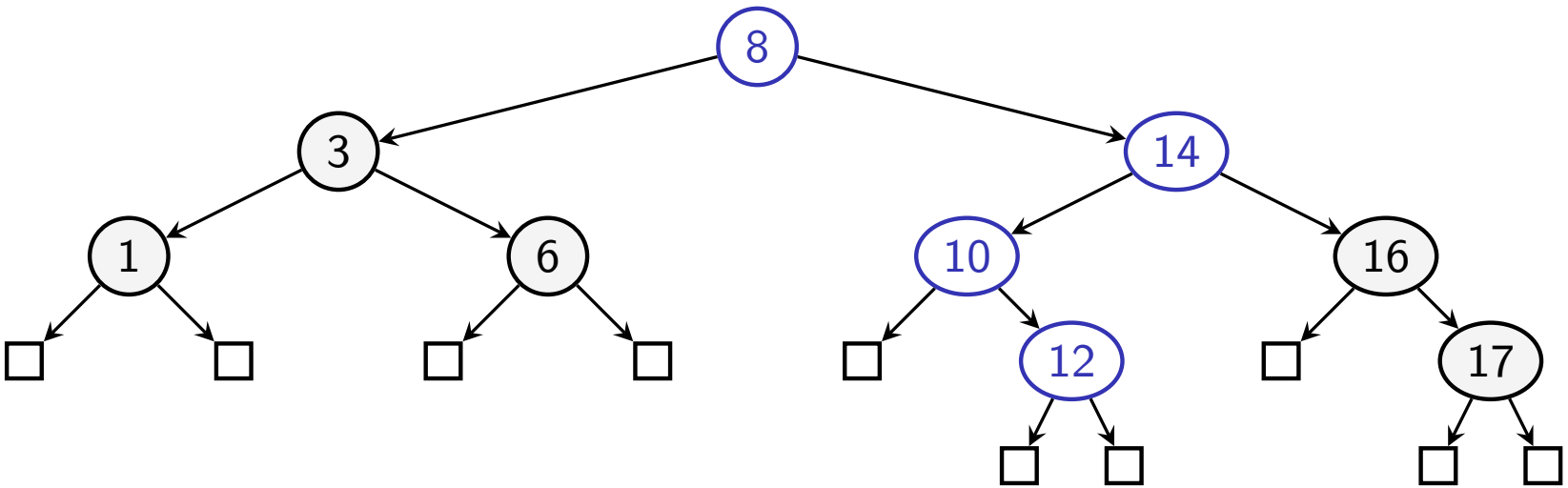
Binärbaum: Einfügen

- Wir wollen die Zahl 12 einfügen.



Binärbaum: Einfügen

- Wir wollen die Zahl 12 einfügen.
- Von der Wurzel beginnend absteigen:
  - Ist 12 kleiner: gehe nach links.
  - Ist 12 größer: gehe nach rechts.
- Gibt es kein weiteres Kind, dann füge dort den neuen Knoten ein.
- Wie viele Schritte sind notwendig? Im schlimmsten Fall: Höhe des Baums.

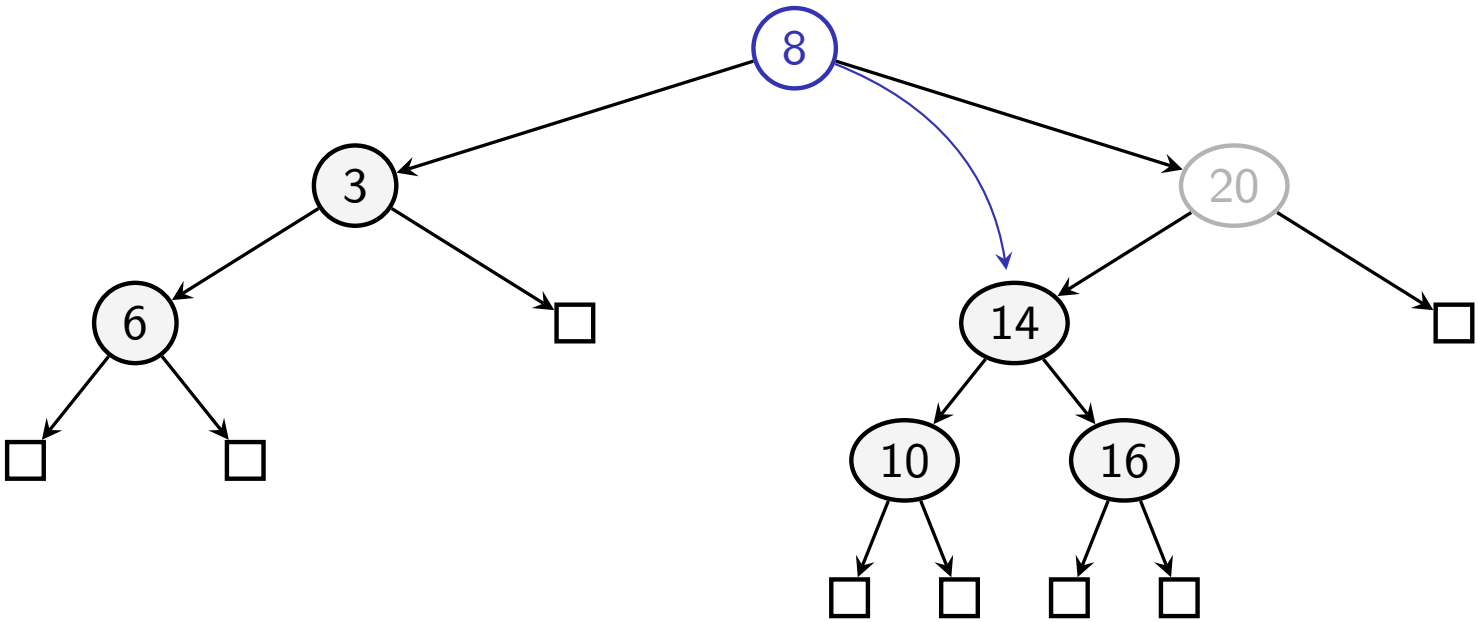


Binärbaum: Einfügen

```
1  void insert(int value)
2  {
3      Node* n = new Node;
4      n->left = n->right = 0; n->value = value;
5      if( getRootNode() == 0 ) {       // Tree is empty
6          root = n;
7          return;
8      }
9      Node* p  = getRootNode();
10     while(1) {
11         if( value >= p->value ) {     // Need to go right
12             if( p->right==0 ) {       // Insert new node
13                 p->right = n;
14                 return;
15             }
16             p = p->right;
17         } else {                      // Need to go left
18             if( p->left==0 ) {        // Insert new node
19                 p->left = n;
20                 return;
21             }
22             p = p->left;
23         }
24     }
```

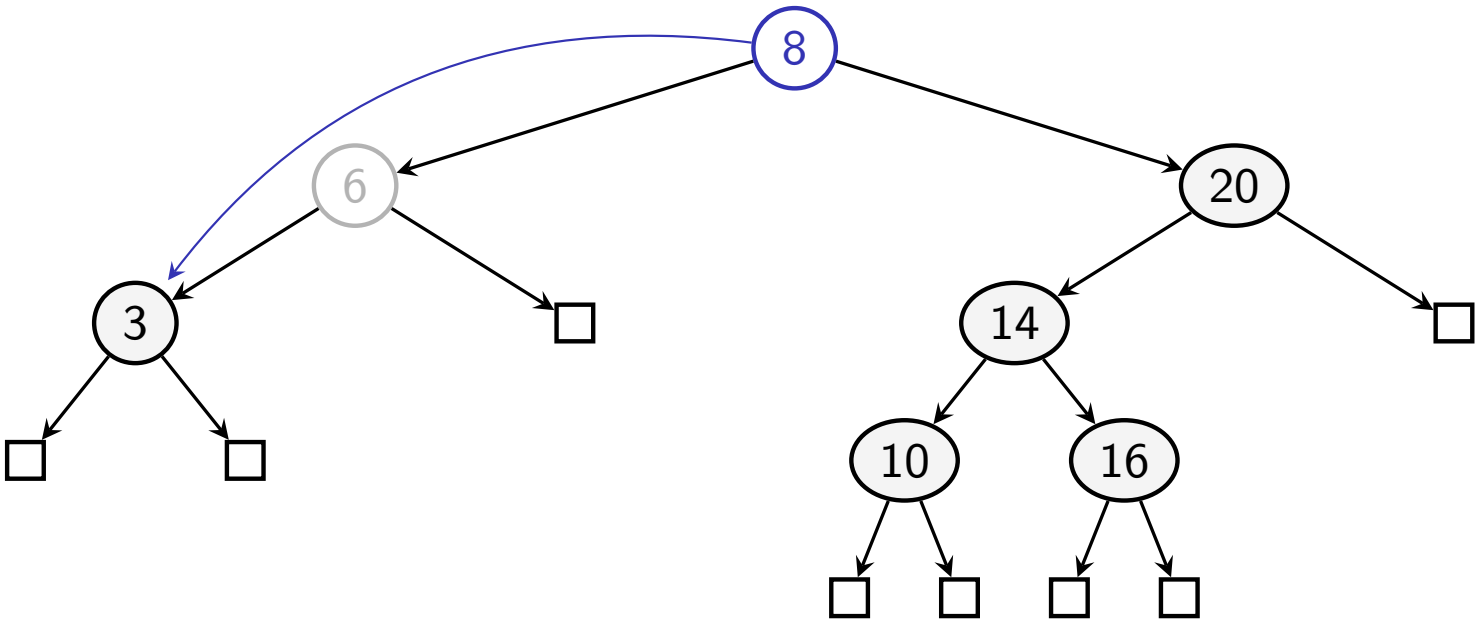
Binärbaum: Entfernen (Fall 1)

- Der zu löschende Knoten hat kein rechtes Kind, z.B. 20.
- Knoten entfernen und Pointer des Elternknoten updaten.
  - Spezialfall: zu löschender Knoten ist Wurzel.



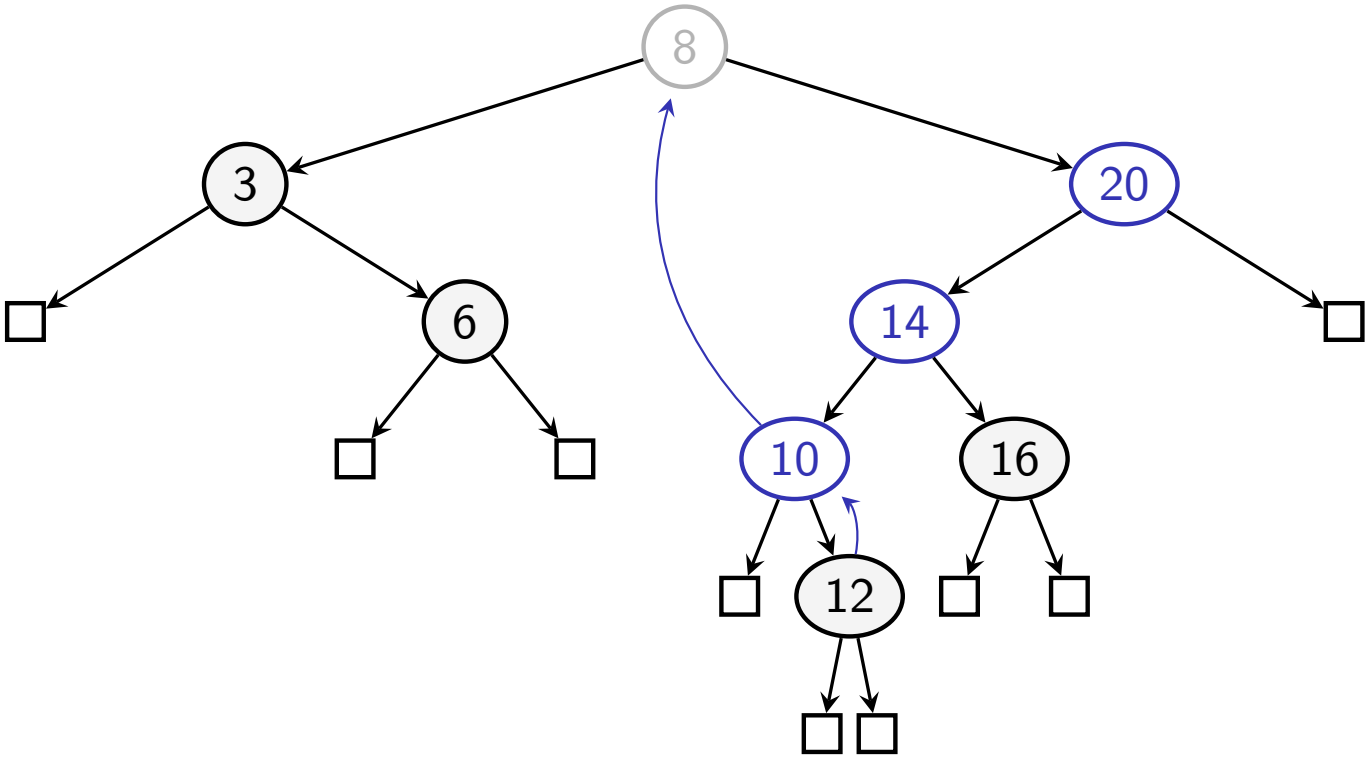
Binärbaum: Entfernen (Fall 1)

- Der zu löschende Knoten hat kein rechtes Kind, z.B. 3.
- Knoten entfernen und Pointer des Elternknoten updaten.
  - Spezialfall: zu löschender Knoten ist Wurzel.



Binärbaum: Entfernen (Fall 2)

- Zu löschender Knoten hat ein rechtes Kind, z.B. 8
  - Wähle im rechten Teilbaum den kleinsten (linkesten) Nachkommen: 10
    - Verschiebe diesen Knoten an die Stelle des entfernten Knoten.
  - Hätte 10 ein rechtes Kind, so rückt dieses an die Stelle von 10.
    - Achtung: 20 hätte bereits kein linkes Kind haben können.
- Wie viele Schritte sind notwendig? Maximal Höhe des Baums.



Binärbaum: Entfernen (Fall 2b)

- Das rechte Kind (20) vom zu löschenden Knoten (8) hat kein linkes Kind.
  - Gib die 20 an die Stelle der 8 und den lösche den Knoten, der die 20 enthielt.

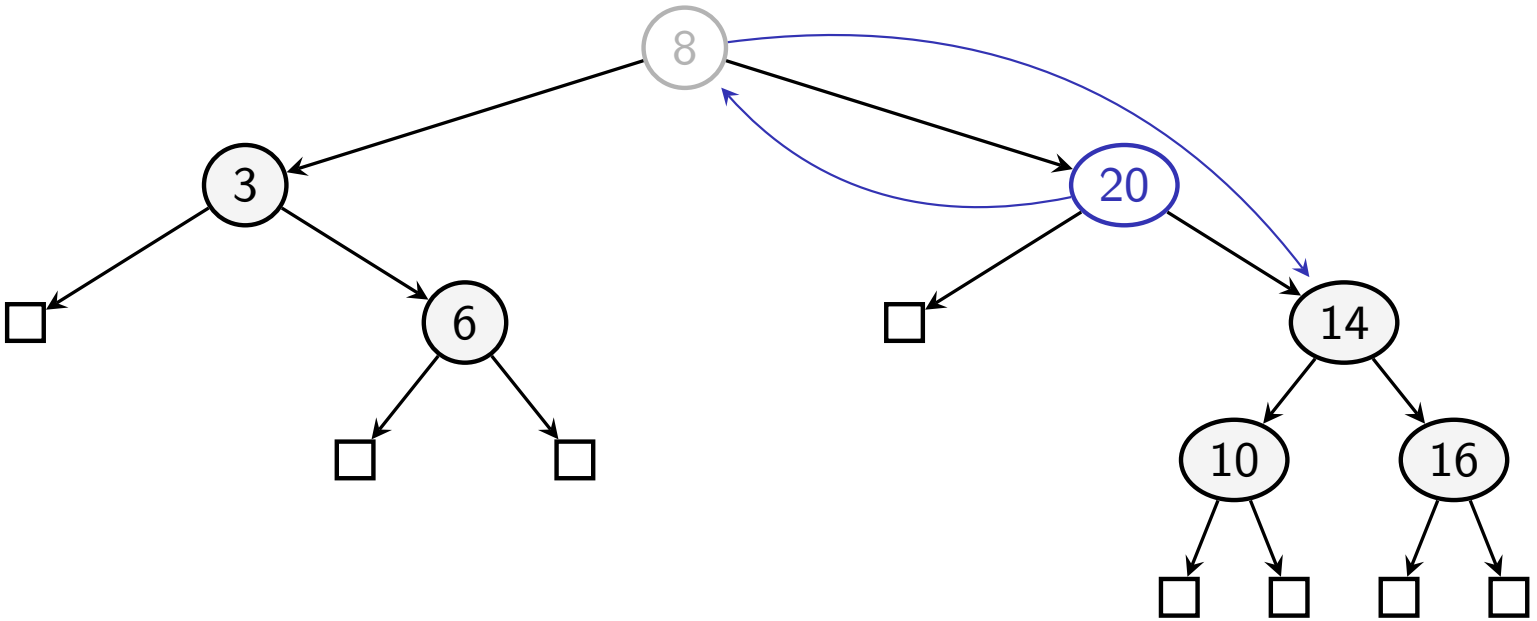
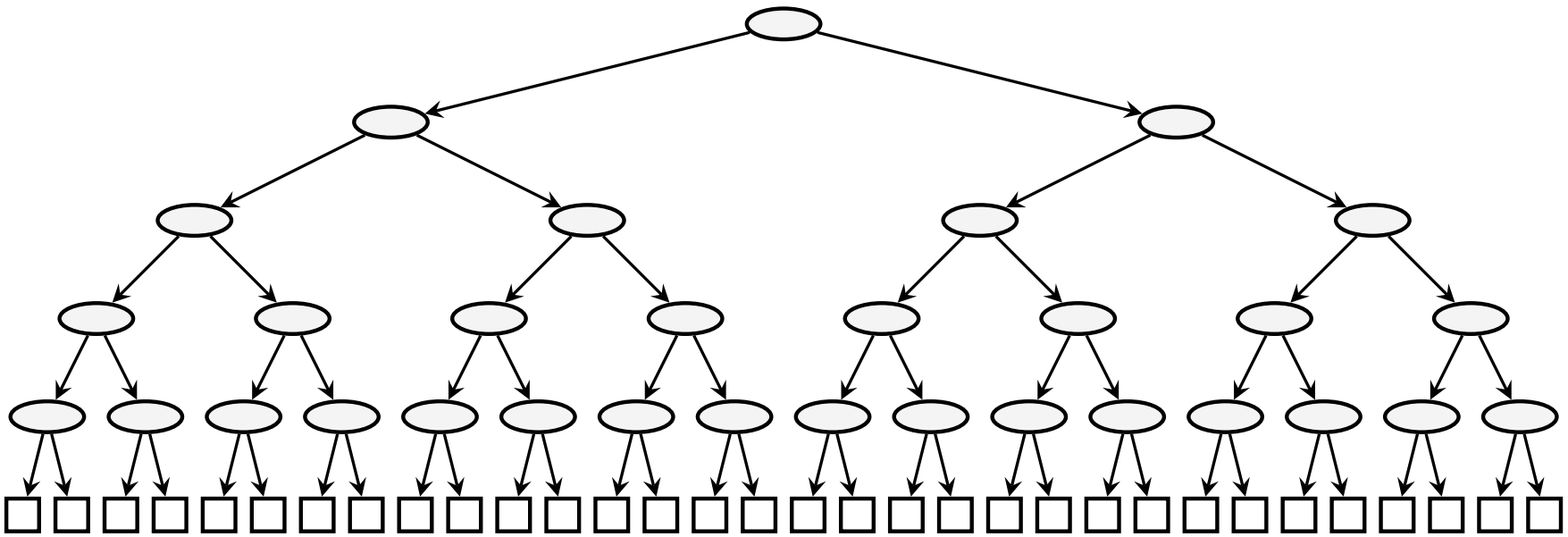


Illustration der Höhe

- Die Höhe des Baums ist nur 4.
- Im unteren Level sind schon 16 Knoten.
- Bei einer Höhe von 10 sind es schon 1024 Knoten.
- Bei einer Höhe von 20 sind es schon 1048576 Knoten.



Binärer Suchbaum: Fazit

- Suchen, Einfügen, Löschen: man benötigt soviele Schritte, wie der Baum hoch ist.
- Quintessenz: **Der Baum soll breit sein, nicht hoch!**
- Ist der Baum relativ ausgeglichen, so ist Suchen, Einfügen und Löschen sehr schnell.
  - Bei ca. 10^6 Elementen brauchen wir ungefähr 20 Schritte für doesContain, insert, remove.

Weiterführendes Thema: Balancierte Binärbaume (balanced binary search trees):

- Idee: Achte bei insert/remove darauf, dass der Baum ausgeglichen bleibt.
- Beispiele für ausgeglichene Binärbäume:
  - AVL Bäume
  - Rot-Schwarz Baum (red-black trees)
- Beispiele für ausgeglichene Bäume mit mehr Kinder:
  - 2-3-4 Bäume
  - Bayer Bäume (B tree)

Mengen (set)

- Eine Menge (set) unterstützt zumindest folgende Operationen:
  - `void insert(value)`
  - `void remove(value)`
  - `bool doesContain(value)`
- Herkömmliche Implementierungsmöglichkeiten:
  - Verkettete Liste: `doesContain` ist langsam.
  - Array: `doesContain` ist langsam.
  - Array (sortiert): `insert, remove` ist langsam.
- Mit einem Binärbaum können alle drei Operationen sehr schnell sein.

Assoziatives Array (map)

- Idee: Eine Datenstruktur, die Schlüssel-Werte Paare (key-value pairs) abspeichert.
  - Wir wollen zu einem bestimmten Schlüssel einen Wert abspeichern.
  - Über den Schlüssel kann auf den abgespeicherten Wert wieder zugegriffen werden.
- Firmiert unter vielen Namen: dictionary, associative array, map, associative memory, ...

```
1 mailaddr.insert(key, value);
2 mailaddr.insert("Alice", "alice@kernel.org");
3 mailaddr.insert("Bob", "bob@kernel.org");
4 mailaddr.insert("Mallory", "mallory@google.com");
5 String addr = mailaddr.get("Bob");
6 mailaddr.remove("Mallory");
```

- Wir wollen schnell einfügen, löschen, suchen.
- Ähnliche zu Menge, aber mit key-value Paaren.
- Idee:
  - Nehme binären Suchbaum und speichere in Knoten Schlüssel und Wert.
  - Für die Ordnung im Baum vergleiche mit den Schlüsseln.

Assoziatives Array (map)

```
1  template<typename Key, typename Value>
2  class BintreeMap
3  {
4    public:
5      struct Node
6      {
7        Key k;
8        Value v;
9        Node* left;
10       Node* right;
11     };
12   private:
13     Node* root;
14
15   public:
16     void insert(const Key& k, const Value& v);
17     void remove(const Key& k);
18     Value& get(const Key& k);
19     bool doesContain(const Key& k) const;
20 };
```

Kapitel

Rekursion

Rekursion in der Mathematik

- Eine rekursive Funktion in der Mathematik ist eine Funktion, die durch sich selbst definiert ist.
  - Beispiel: Fakultät von n

$$0! := 1$$
$$(n+1)! := (n+1) \cdot n! \quad \text{für } n \in \mathbb{N}_0$$

    - $4! = 4 \cdot 3 \cdot 2 \cdot 1$
  - Eine rekursive Definition braucht einen **Rekursionsanfang**!
  - Beispiel: Summe

$$\sum_{i=0}^{0} a_i := a_0$$
$$\sum_{i=0}^{n+1} a_i := a_{n+1} + \sum_{i=0}^{n} a_i \quad \text{für } n \in \mathbb{N}_0$$

Rekursion in der Informatik

- Eine Funktion/Prozedur in einer Programmiersprache heißt rekursiv, wenn sie sich selbst aufruft.

```
1 unsigned factorial(unsigned n)
2 {
3   if( n==0 )
4     return 1;
5   return n*factorial(n-1);
6 }
```

- `factorial(4) = 4*factorial(3) = 4*3*factorial(2) = 4*3*2*factorial(1) = 4*3*2*1*factorial(1) = 4*3*2*1*1 = 24`
- Laufzeit: $\Theta(n)$

Fibonacci Zahlen

- Die Fibinacci-Zahlen F_n sind wie folgt definiert:

$$F_n := F_{n-1} + F_{n-2} \quad \text{für } n \geq 2$$
$$F_1 := 1$$
$$F_0 := 1$$

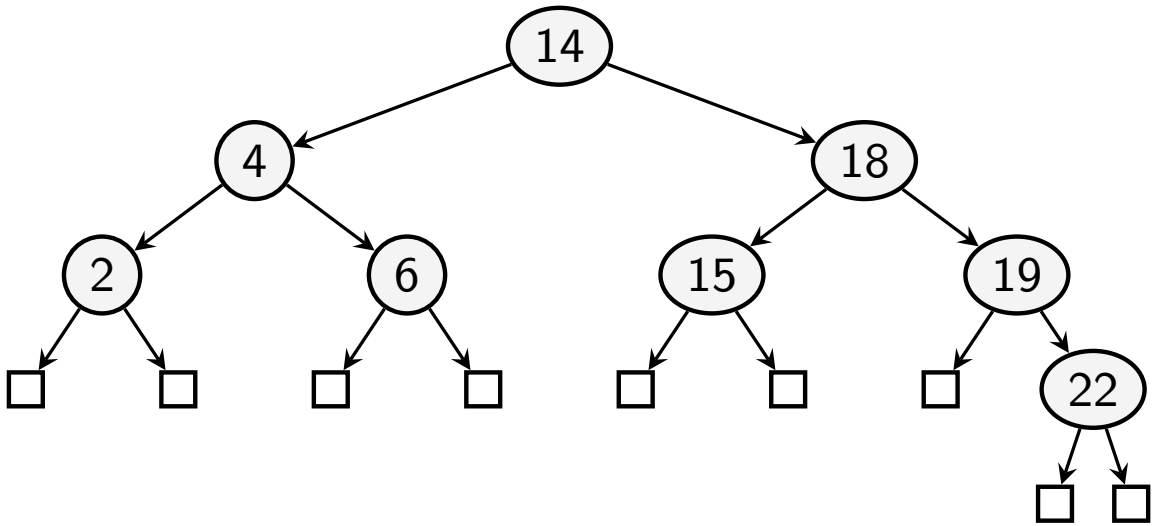
- Die Fibonacci-Folge $F_0, F_1, F_2, F_3, \ldots$ lautet daher $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, \ldots$.

Fibonacci Zahlen

```
1  unsigned fib(unsigned n)
2  {
3    unsigned a = 1;
4    unsigned b = 1;
5    for(unsigned i=2; i<=n; i++)
6    {
7      const unsigned c = a+b;
8      a = b;
9      b = c;
10   }
11   return b;
12 }
13 unsigned fib_rec(unsigned n)
14 {
15   if( n <= 1 )
16     return 1;
17   return fib_rec(n-1) + fib_rec(n-2);
18 }
```

Warum Rekursion?

- Oft sehr einfache Lösungen für zunächst schwierige Probleme.
- Beispiel: preorder Traversierung von Binärbäumen
  - Traversierung: Besuchen aller Knoten eines Baums.
  - preorder: Für jeden Knoten N:
    1. Gib N aus.
    2. Gib die Knoten des linken Teilbaums pre-order aus.
    3. Gib die Knoten des rechten Teilbaums pre-order aus.



  - 14, 4, 2, 6, 18, 15, 19, 22

Preorder Traversierung: nicht rekursiv

```
1  void print_preorder(Node* root)
2  {
3    if( root == 0 )
4      return;
5
6    Stack<Node*> s;
7    s.push(root);
8
9    while( !s.isempty() )
10   {
11     Node* n = s.top();
12     s.pop();
13     cout << n->value << endl;
14     // Push right before left to process left before right
15     if( n->right != 0 )
16       s.push(n->right);
17     if( n->left != 0 )
18       s.push(n->left);
19   }
20 }
```

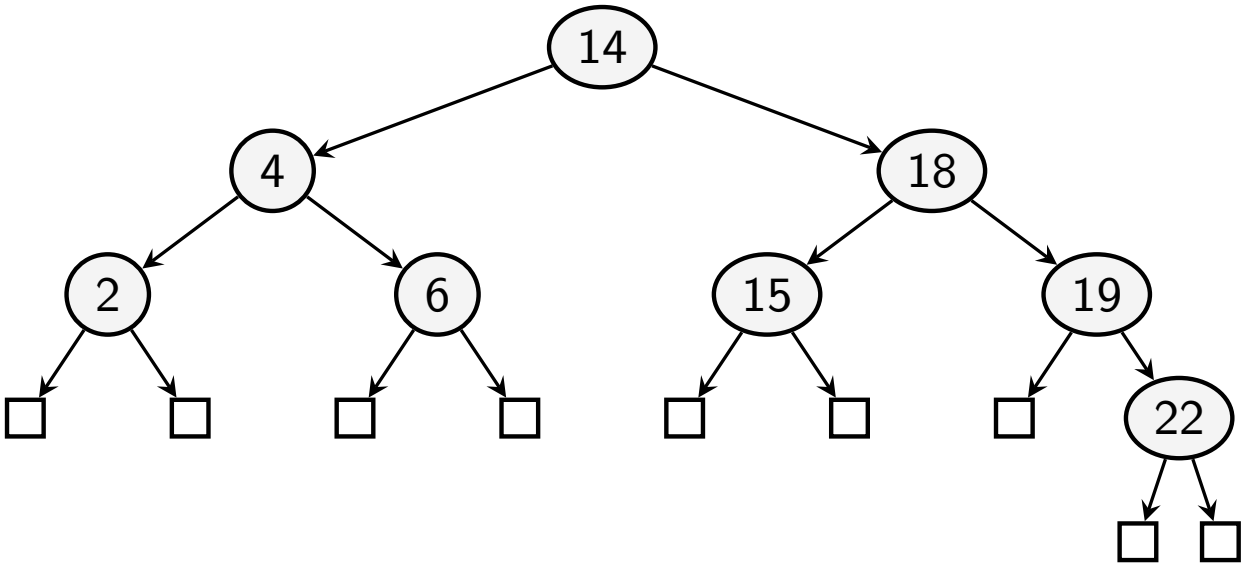
Preorder Traversierung: rekursiv

```
1  void print_preorder(Node* root)
2  {
3    // Termination condition (Abbruchbedingung)
4    if( root==0 )
5      return;
6
7    cout << root->value << endl;
8    print_preorder(root->left);
9    print_preorder(root->right);
10 }
```

- Der call stack übernimmt die Rolle des „händischen“ Stack von vorhin.
- Code ist lesbarer und genauso effizient.
- Achtung: der call stack darf nicht beliebig groß werden.
  - Hängt vom Betriebssystem ab.
  - Wenn die Rekursionstiefe zu groß wird: Segmentation fault.

Inorder Traversierung

- Bei jedem Knoten N:
  1. gib die Knoten im linken Teilbaum aus,
  2. gib N aus,
  3. gib die Knoten im rechten Teilbaum aus.



- 2, 4, 6, 14, 15, 18, 19, 22

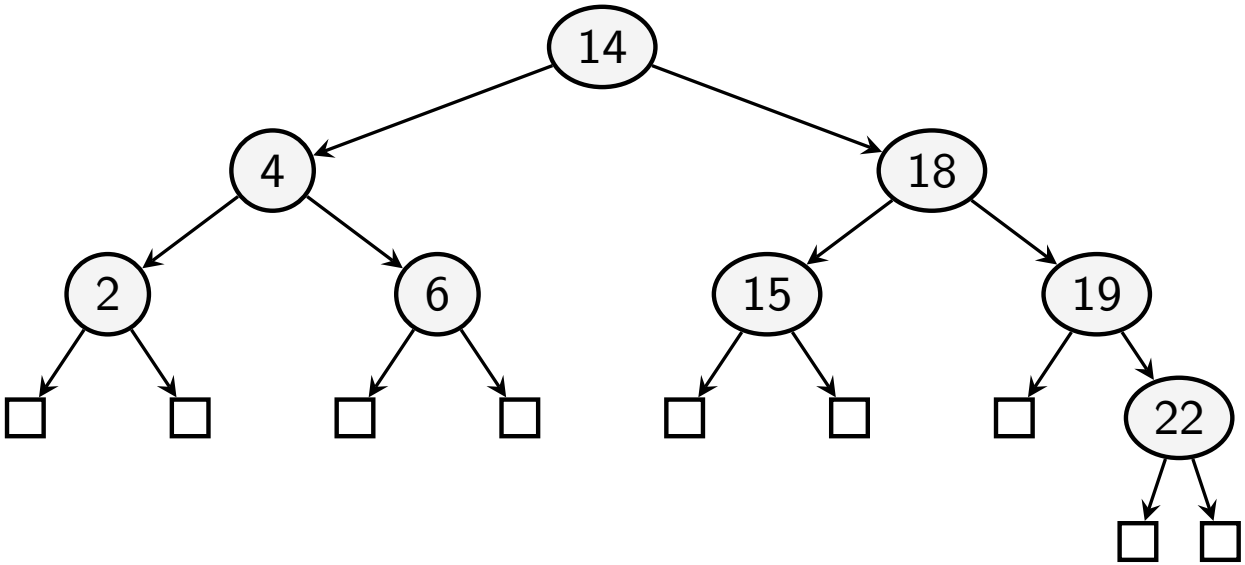
Inorder Traversierung

```
1  void print_inorder(Node* root)
2  {
3    // Termination condition (Abbruchbedingung)
4    if( root==0 )
5      return;
6
7    print_inorder(root->left);
8    cout << root->value << endl;
9    print_inorder(root->right);
10 }
```

- Wie macht man das ohne Rekursion?

Level-order Traversierung

- Gib die Knoten level für level aus.



- 14, 4, 18, 2, 6, 15, 19, 22

Level-order Traversierung

- Motto bei Rekursion: Behandle die Teilbäume unabhängig von einander.
  - Nicht bei level-order Traversierung!

```
1  void print_levelorder(Node* root)
2  {
3    if( root == 0 )
4      return;
5    Queue<Node*> q;
6    q.enqueue(root);
7    while( !q.isempty() )
8    {
9      Node* n = q.dequeue();
10     cout << n->value << endl;
11     if( n->left != 0 )
12       q.enqueue(n->left);
13     if( n->right != 0 )
14       q.enqueue(n->right);
15   }
16 }
```

- Gleich wie preorder Traversierung, aber mit Queue statt Stack!