Technische Universität München
Fakultät für Informatik
Lehrstuhl für Effiziente Algorithmen
Dr. Hanjo Täubig
Tobias Lieber

Sommersemester 2011
Übungsblatt 5
19. September 2011

Grundlagen: Algorithmen und Datenstrukturen

Abgabetermin: 19. September 2011

Hausaufgabe 1

Implementieren Sie in der Klasse `UIbiHeap` einen Binomial-Heap. Hierzu muss auch ein BinomialBaum in der Klasse `binomialTree` implementiert werden.
Verwenden Sie für Ihre Implementierung die auf der Übungswebseite bereitgestellten Klassen und verändern Sie für Ihre Implementierung *ausschließlich* die Klasse `UIbiHeap` und `binomialTree`.

Lösungsvorschlag

Siehe Übungswebseite.

Hausaufgabe 2

Implementieren Sie in der Klasse `IbinaryTree` einen binären Baum, der neben den Operationen `insert, remove, find` auch die Operationen `preOrder` und `postOrder` implementiert.
Die Funktion `preOrder` druckt ein Element (anfangs die Wurzel) und steigt dann rekursiv zuerst in den linken und dann in den rechten Teilbaum ab (In einer ersten Version des Blattes war links und rechts vertauscht). Die Funktion `postOrder` arbeitet umgekehrt. Sie führt zuerst die rekursiven Aufrufe aus und druckt dann das Element (anfangs die Wurzel).
Verwenden Sie für Ihre Implementierung die auf der Übungswebseite bereitgestellten Klassen und verändern Sie für Ihre Implementierung *ausschließlich* die bereitgestellten Klassen aber nicht deren Interfaces.

Lösungsvorschlag

Siehe Übungswebseite.

Hausaufgabe 3

Implementieren Sie in der Klasse `Graph` eine Repräsentation eines Graphen durch eine Adjazenzliste. Stellen Sie die Funktionen wie im Interface angegeben bereit.
Verwenden Sie für Ihre Implementierung die auf der Übungswebseite bereitgestellten Klassen und verändern Sie für Ihre Implementierung *ausschließlich* die bereitgestellten Klassen aber nicht deren Interfaces.

Lösungsvorschlag

Siehe Übungswebseite.

Hausaufgabe 4

Implementieren Sie in der Klasse `DFS` eine Funktion, die auf der eben implementierten `Graph`-Klasse zu einem angegebenen Knoten s eine Tiefensuche durchführt und die Knoten ausgibt nachdem alle Kinder abgearbeitet sind. Achten Sie darauf, dass Ihre Implementierung im Gegensatz zu dem in der Vorlesung vorgestellten Ansatz nicht rekursiv ist.
Verwenden Sie für Ihre Implementierung die auf der Übungswebseite bereitgestellten Klassen und verändern Sie für Ihre Implementierung *ausschließlich* die bereitgestellten Klassen aber nicht deren Interfaces.

Lösungsvorschlag

Siehe Übungswebseite.

Aufgabe 1

Geben Sie einen Algorithmus für eine Build-Funktion an, die n Elemente $e_1, \ldots, e_n$ als Eingabe erhält und diese in einem Binomial-Heap speichert und die in $\mathcal{O}(n)$ Zeit läuft. Beweisen Sie die Laufzeit Ihres Algorithmus.

Lösungsvorschlag

Ähnlich zum linearen Build-Algorithmus von Binär-Heaps wählen wir einen Bottom-Up-Ansatz: Anfangs haben wir n Bäume mit Rang 0. Davon werden jeweils zwei zu einem Binomialbaum mit Rang 1 zusammengefügt. Kann kein Baum mehr mit Rang 1 erzeugt werden, werden Bäume mit Rang 1 zu Binomialbäumen mit Rang 2 zusammengefügt. Dieses Verfahren wird so lange wiederholt bis kein Baum mehr mit größerem Rang entstehen kann. Am Ende muss nur noch der Zeiger auf das Minimum gesetzt werden.
Die Laufzeit beträgt dann also:

$$T(n) \leq \sum_{j=0}^{\lfloor \log n \rfloor} \frac{n}{2^j} = n \sum_{j=0}^{\lfloor \log n \rfloor} \frac{1}{2^j} = n \frac{1 - \frac{1}{2}^{\lfloor \log n \rfloor + 1}}{1 - \frac{1}{2}} \leq 2n$$

Alternativ kann man auch jedes Zusammenlinken von zwei Bäumen als einen internen Knoten eines Binärbaumes darstellen. Somit erhält man höchstens einen vollständigen binären Baum. Da die Zahl der internen Knoten eines vollständigen binären Baumes mit n Blättern gerade $n-1$ ist, sind somit weniger als $2n$ Verknüpfungsoperationen durchgeführt worden.
Somit ist die Laufzeit also in $\mathcal{O}(n)$.

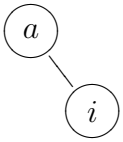
Aufgabe 2

Fügen Sie die Schlüssel $a, b, \ldots, i$ in der Reihenfolge $(a, i, e, g, h, f, c, b, d)$ in einen anfänglich leeren AVL-Baum ein. Entfernen Sie anschließend den Schlüssel i. Zeichnen Sie jeweils den resultierenden Baum (einschließlich notwendiger Rotationen).

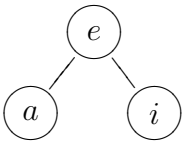
Lösungsvorschlag

Lösungsvorschlag:
Einfügen des Schlüssels 1:

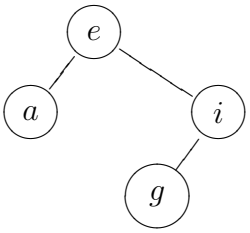
Einfügen des Schlüssels i:



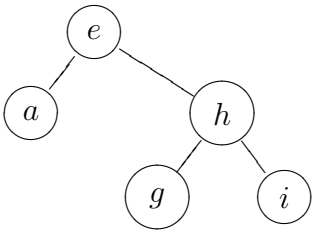
Einfügen des Schlüssels e mit Doppelrotation:



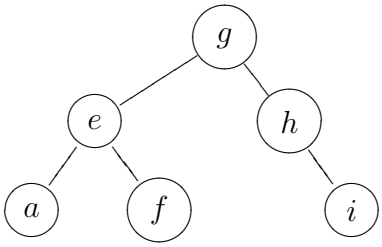
Einfügen des Schlüssels g:



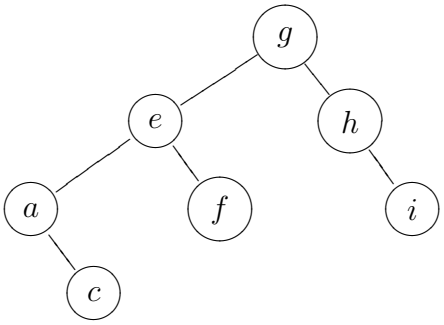
Einfgen des Schlüssels h mit Doppelrotation:



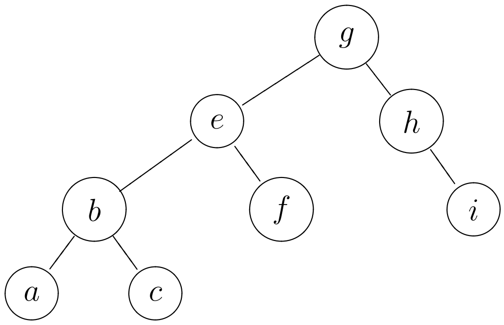
Einfügen des Schlüssels f mit Doppelrotation:



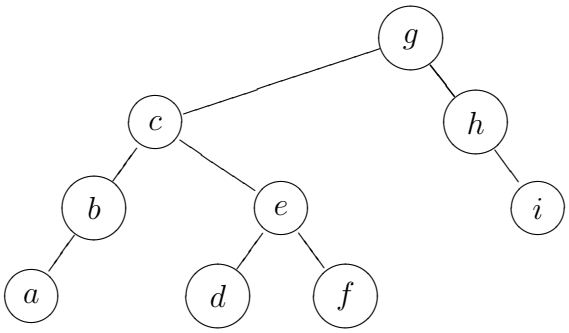
Einfügen des Schlüssels c:



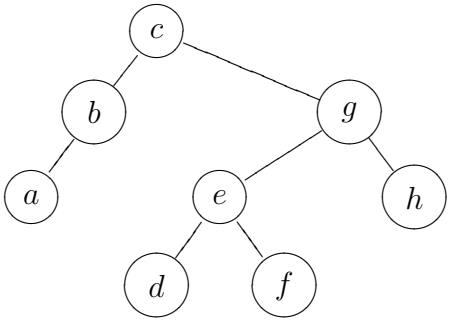
Einfügen des Schlüssels b mit Doppelrotation:


g
e
h
b
f
i
a
c

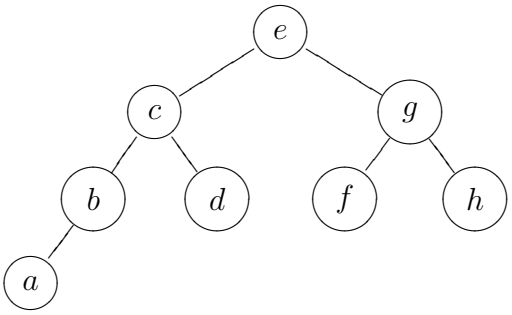
Einfügen des Schlüssels d mit Doppelrotation:



Löschen des Schlüssels i mit Einzelrotation:



Alternativ wäre auch folgender AVL-Baum möglich (mit Doppelrotation):

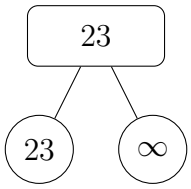


Aufgabe 3

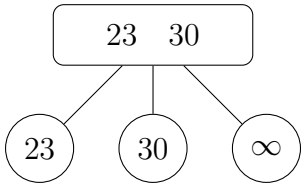
Führen Sie auf einem anfangs leeren $(2, 4)$-Baum folgende Operationen aus und zeichnen Sie die Zwischenergebnisse: `insert(23)`, `insert(30)`, `insert(13)`, `insert(6)`, `insert(40)`, `insert(80)`, `insert(62)`, `insert(75)`, `insert(28)`, `insert(21)`, `insert(29)`, `remove(62)`, `remove(75)`, `remove(13)`.

Lösungsvorschlag

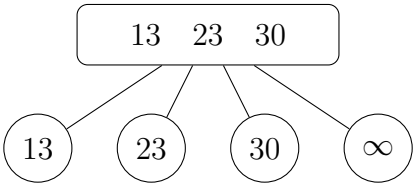
`insert(23)`:



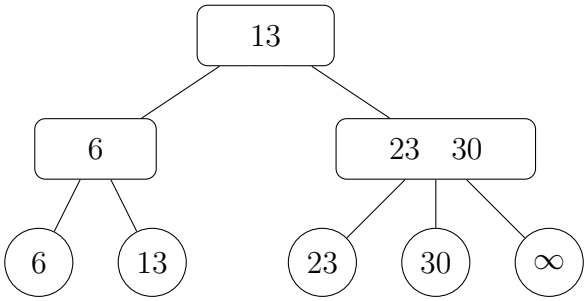
`insert(30)`:



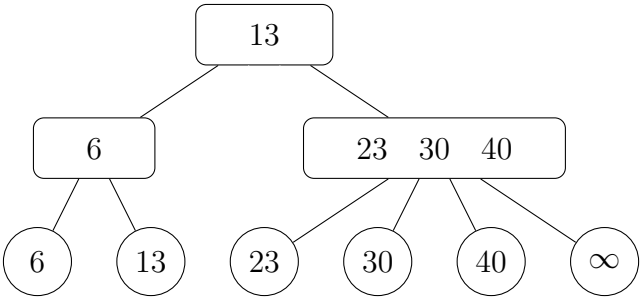
`insert(13)`:



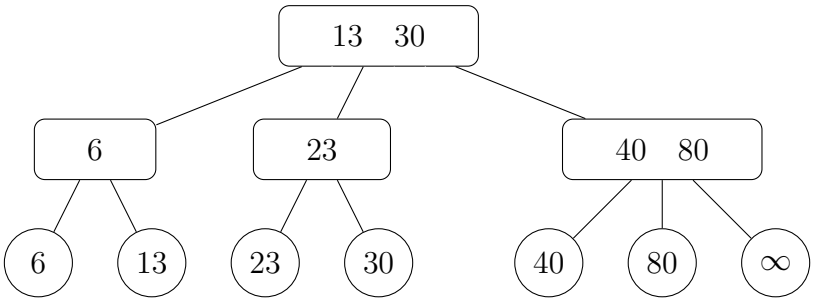
`insert(6)`:



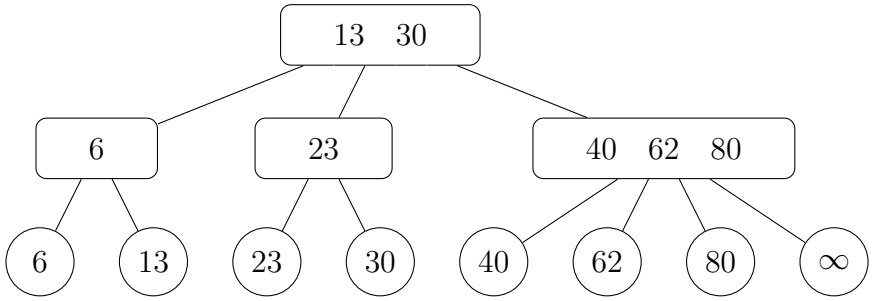
`insert(40)`:



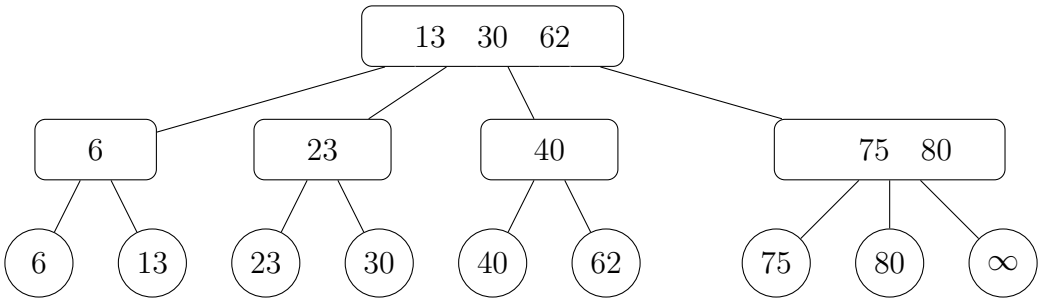
`insert(80)`:



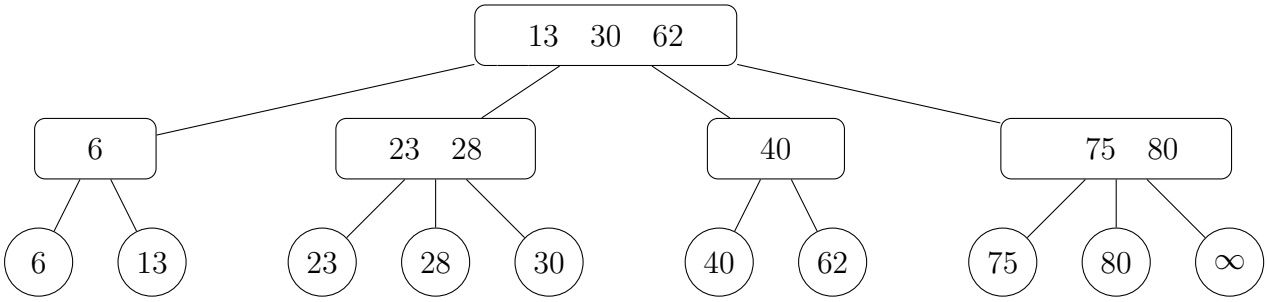
`insert(62)`:



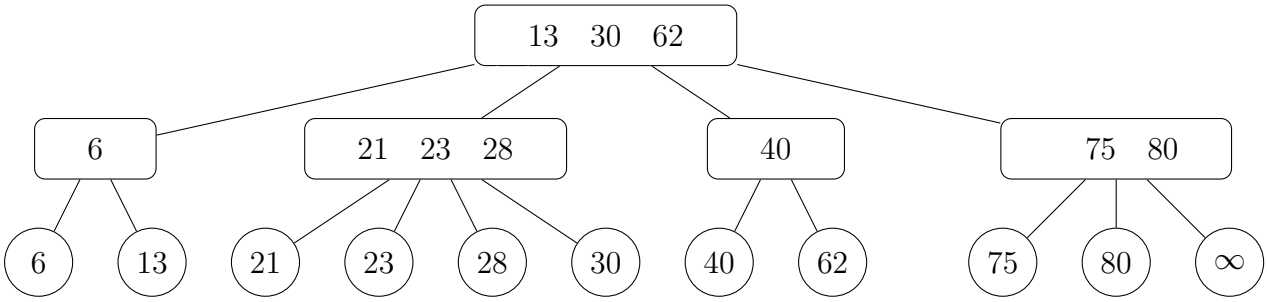
`insert(75)`:



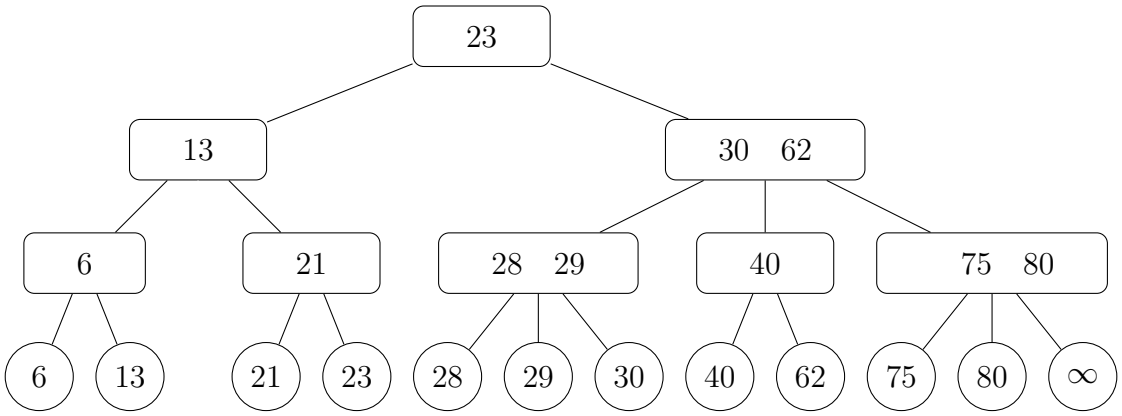
`insert(28)`:



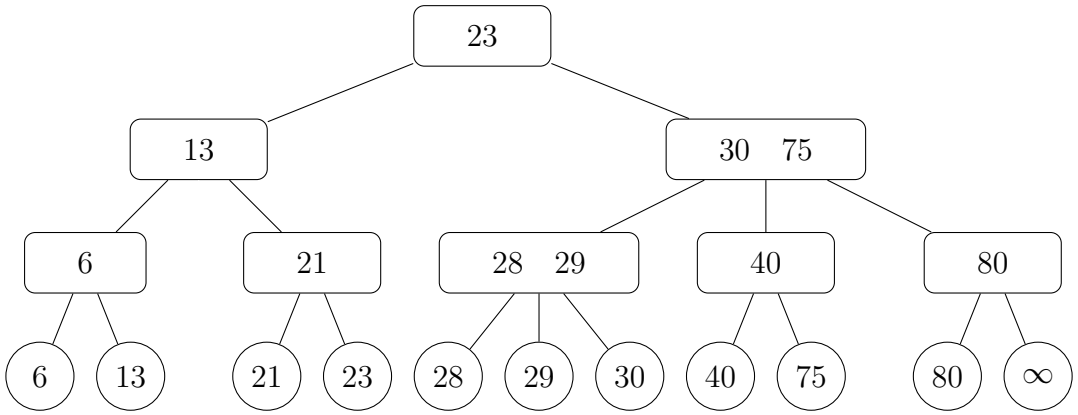
`insert(21)`:



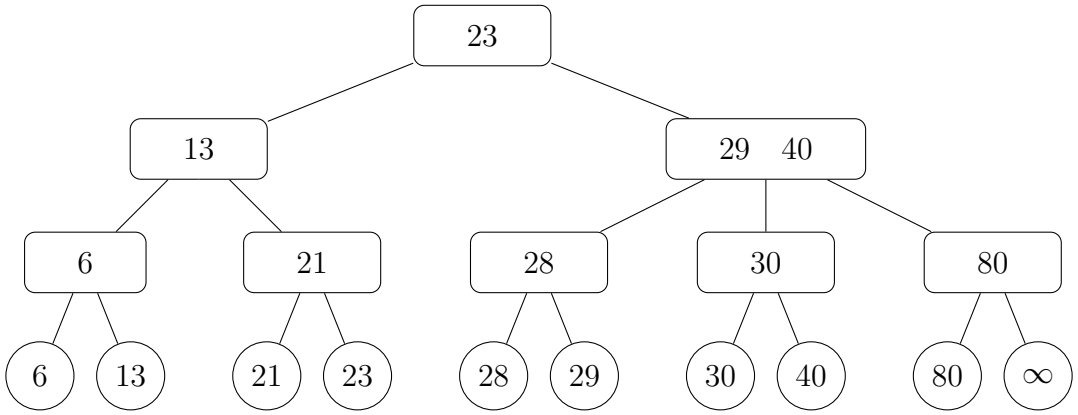
textttinsert(29):



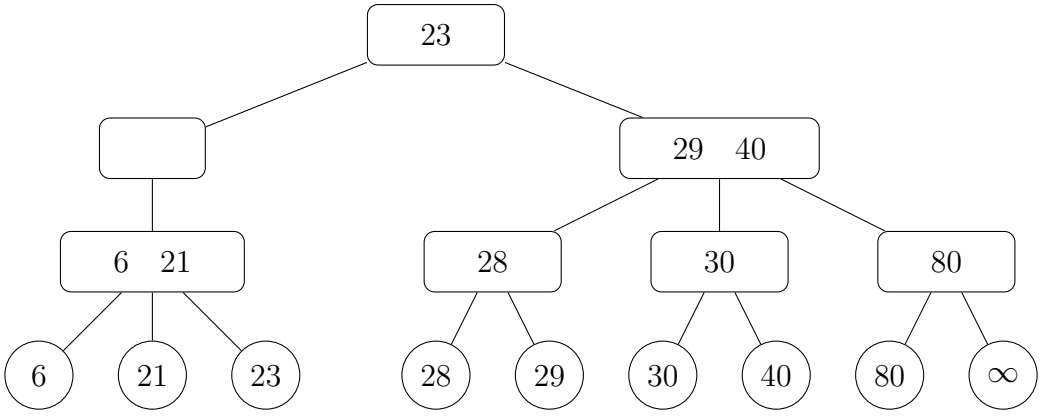
`remove(62)`:



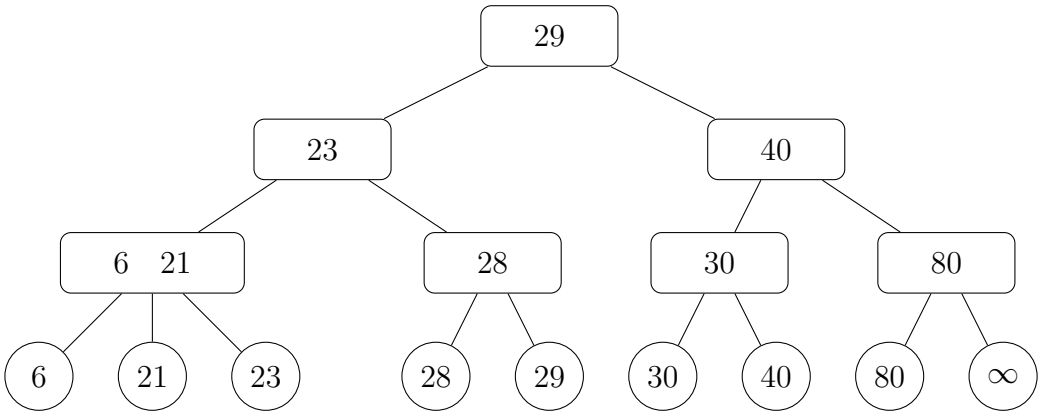
`remove(75)`:



`remove(13)` Teil a):



`remove(13)` Teil b):



Aufgabe 4

Beweisen Sie folgende Aussage:
Für einen $(2,3)$-Baum gibt es eine Folge von n `insert` bzw. `remove`-Operationen, so dass die Anzahl der nötigen Aufspaltungen und Vereinigungen von internen Knoten in $\Omega(n \log n)$ ist.

Lösungsvorschlag

Zunächst sorgen wir dafür, dass der (anfangs leere) $(2,3)$-Baum eine gewisse Anzahl von Blättern enthält. Wir berechnen dazu die größte Zahl so dass der $(2,3)$-Baum eine Zweierpotenz an Blättern hat, die kleiner oder gleich $n/2$ ist. Diese Größe ist gerade $n' := 2^{\lfloor \log_2(n/2) \rfloor} - 1$ und damit gilt $n' \geq \frac{1}{4}n - 1$. Daher ist $n' \in \Omega(n)$. Wir fügen die Schlüssel von 1 bis n' in den $(2,3)$-Baum ein und erhalten einen vollständigen Binärbaum. Die restlichen $n - n'$ Operationen verwenden wir, um abwechselnd immer wieder denselben Schlüssel mit `remove` zu entfernen und mit `insert` wieder einzufügen. Die Anzahl dieser Operationen ist natürlich auch mindestens halb so groß wie n, also natürlich auch in $\Omega(n)$. Bei jeder `remove`-Operation wird ein Blatt entfernt, so dass der entsprechende Vater-Knoten nur noch ein Kind hat. Dieser Knoten muss mit einem anderen Knoten vereinigt werden. Der gemeinsame Vater-Knoten hatte vorher jedoch auch nur diese zwei Kinder und muss nun ebenfalls mit seinem einzigen Bruder verschmolzen werden usw. Dieser Prozess endet erst an der Wurzel. Umgekehrt wird bei jeder darauffolgenden `insert`-Operation ein Knoten mit drei Kindern in zwei Knoten mit je zwei Kindern aufgespalten,

was sich auch wieder bis zur Wurzel fortsetzt. Da die Höhe des Baums mit $n' \in \Omega(n)$ Blättern gleich $\lfloor \log_2(n') \rfloor \in \Omega(\log_2 n)$ ist, beträgt der Gesamtaufwand $\Omega(n \log_2 n)$.

Aufgabe 5

In dieser Aufgabe modifizieren wir die gestellten Bedingungen für einen (a, b)-Baum so dass wir einen B^*-Baum erhalten.
Wir ändern die Grad-Invariante wie folgt: Ein Knoten darf höchstens Grad b haben. Jeder Knoten außer der Wurzel hat mindestens $\frac{2b-1}{3}$ Kinder. Die Wurzel hat mindestens 2 Kinder und höchstens $2\lfloor \frac{2b-2}{3} \rfloor + 1$ Kinder.

- Wie müssen die `insert` und `delete`-Operationen des (a, b)-Baumes modifiziert werden, so dass die Grad-Invariante immer gegeben ist?
- Welche Vorteile und Nachteile ergeben sich aus praktischer und theoretischer Sicht für die Speicherausnutzung und die Laufzeit der `insert`-Operationen?

Lösungsvorschlag

- Modifikationen bei `insert`: Hat ein Knoten v, zu dem ein Kind eingefügt wird, Grad b, so überprüfen wir ob wir das o.B.d.A. rechtesteste Kind beim rechten Nachbarn einfügen können und somit genau b Elemente in Knoten v erhalten. Geht dies ohne Probleme, so sind wir fertig. Anderenfalls hat der rechte Nachbar r auch b Knoten. Dann können wir aber v und r in 3 Knoten mit $\lfloor (2b-2)/3 \rfloor + 1, \lfloor (2b-1)/3 \rfloor + 1$ bzw. $\lfloor 2b/3 \rfloor + 1$ Kinder aufspalten (also grob $\frac{2}{3}b$ Kinder pro Knoten). Danach müssen die darüberliegenden Level auf die Grad-Invariante überprüft werden.

  Modifikationen für `delete`: Hat Knoten v gerade $\lceil \frac{2b-1}{3} \rceil$ Elemente so wird überprüft ob von den o.B.d.A. beiden rechten Nachbarn Kinder transferiert werden können, so dass alle drei Knoten die Gradbedingung erfüllen. Geht dies nicht, da alle drei Knoten zusammen zu wenige Kinder haben, so werden die drei Knoten zu 2 Knoten zusammengefügt.

- Da jeder Knoten mindestens Grad $\frac{2b-1}{3}$ hat, folgt, dass die Höhe des Baumes nicht mehr durch $1 + \log_a \frac{n+1}{2} = 1 + \log_{\lceil \frac{b}{2} \rceil} \frac{n+1}{2}$ sondern durch $1 + \log_{\lceil \frac{2(b-1)}{3} \rceil} \frac{n+1}{2}$ beschränkt ist. Dieser Wert ist kleiner als der vorherige, da die Basis größer ist. Das wirkt sich direkt auf die Laufzeit für beispielsweise `find` aus, die dadurch sinkt.

  Allerdings steigt der Aufwand für `insert`-Operationen da wir ähnlich wie beim Löschen die Nachbarknoten betrachten und im Bedarsfall neu aufsplitten müßen. Die Speicherplatzausnutzung ist in den B^*-Bäumen deutlich höher, da ja jeder Knoten mit mindestens $\frac{2b-1}{3} - 1$ Navigationselementen befüllt sein muss (Dies ist ein Vorteil!).

Aufgabe 6

Geben Sie Algorithmen `postNext(v)` und `preNext(v)` an, die zu einem Knoten v in einem Binärbaum den in der PreOrder bzw. PostOrder folgenden Knoten w berechnet. Analysieren Sie die asymptotische Worst-Case Laufzeit Ihres Pseudocodes.
Berechnen Sie außerdem die asymptotische Laufzeit wenn mittels der Operationen `postNext(v)` und `preNext(v)` die vollständige PreOrder bzw. PostOrder berechnet wird (also n-maliges Anwenden der Funktion).

Lösungsvorschlag

- `preorderNext(v)`: Zunächst prüfen wir, ob v ein linkes Kind hat und geben dieses gegebenenfalls zurück. Hat v kein linkes aber ein rechtes Kind, dann geben wir das rechte Kind zurück. Ansonsten gehen wir solange in Richtung Wurzel bis es einen rechten Teilbaum gibt, in dem wir bisher noch nicht waren. Von diesem Teilbaum geben wir die Wurzel zurück. Gibt es einen solchen Teilbaum nicht, dann sind wir fertig und geben `null` zurück.

```
1  public TreePosition preNext(TreePosition v){
2    if (hasleftChild(v))
3    return leftChild(v);
4    else if (hasrightChild(v))
5    return rightChild(v);
6    else{
7    TreePosition p = v;
8    TreePosition q;
9    while (!isRoot(p)){
10     q = p;
11     p = parent(p);
12     if (hasrightChild(p) && rightChild(p)!=q)
13       return rightChild(p);
14   }
15   return null;
16   }
17 }
```

- `postorderNext(v)`: Zunächst prüfen wir, ob v nicht die Wurzel ist, da wir anderenfalls `null` zurückgeben. Ansonsten betrachten wir den Elternknoten p von v. Ist v das rechte Kind von p, dann ist der rechte Teilbaum von p abgearbeitet und wir geben p zurück. Sonst müssen wir im rechten Teilbaum das "linkeste" Blatt zurückgeben. Dieses Blatt erreichen wir, indem wir ausgehend vom rechten Kind von p immer zu einem linken Kind gehen. Nur für den Fall, dass es kein linkes Kind gibt, gehen wir zum rechten Kind bis wir schließlich ein Blatt erreichen.

```
1  public TreePosition postNext(TreePosition v){
2    if (!isRoot(v)){
3    TreePosition p = parent(v);
4    if (hasrightChild(p)){
5      if (v==rightChild(p))
6        return p;
7      else{
8        TreePosition c = rightChild(p);
9        while (isInternal(c)){
10         if (hasleftChild(c))
11           c = leftChild(c);
12         else
13           c = rightChild(c);
```

```
14          }
15          return c;
16        }
17      }else return p;
18      }else return null;
19  }
```

Für beide Methoden gilt, dass wir in den `while`-Schleifen nur entlang eines Wurzel-Blatt-Pfades laufen. Somit ist die Laufzeit in allen drei Fällen $O(h(T))$, wobei $h(T)$ die Höhe des Baumes T ist. Da wir allgemeine binäre Bäume betrachten, ist die Höhe nicht beschränkt und wir erhalten im worst-case eine Laufzeit von $O(n)$.
Da jede Kante des Binärbaums nur höchstens zweimal traversiert wird, (wir vernachlässigen die Parent bzw. rightChild-Vergleiche) ist die Anzahl der Vergleiche durch die Anzahl der Kanten in einem Binärbaum beschränkt. Da jeder Knoten durch einen eindeutigen Vater angebunden ist, ist die Anzahl der Kanten also durch die Anzahl der Knoten im Baum beschränkt, und die Laufzeit liegt damit in $\mathcal{O}(n)$.
kt, und die Laufzeit liegt damit in $\mathcal{O}(n)$.

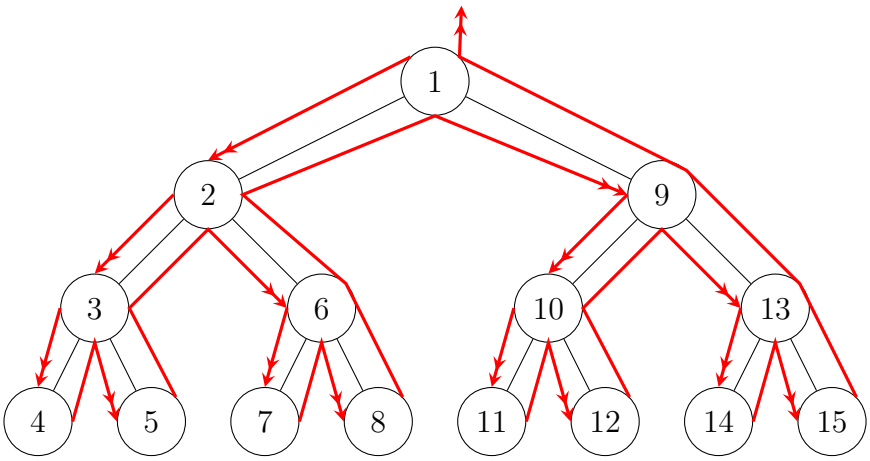


Abbildung 1: Veranschaulichung der traversierten Kanten bei n `preNext`-Aufrufen.