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

Sommersemester 2011
Übungsblatt 8
30. Juni 2011

Grundlagen: Algorithmen und Datenstrukturen

Abgabetermin: In der jeweiligen Tutorübung

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`.
Achten Sie bei der Abgabe Ihrer Aufgabe darauf, dass Ihre Klassen `UIbiHeap` und `binomialTree` heißen und auf den Rechnern der Rayhalle (`rayhalle.informatik.tu-muenchen.de`) mit der bereitgestellten Datei `main_bi` kompiliert werden können. Anderenfalls kann eine Korrektur nicht garantiert werden. Achten Sie darauf, dass Ihr Quelltext ausreichend kommentiert ist.
Schicken Sie die Lösung per Email mit dem Betreff `[GAD] Gruppe <Gruppennummer>` an Ihren Tutor.

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.
Achten Sie bei der Abgabe Ihrer Aufgabe darauf, dass Ihre Klasse auf den Rechnern der Rayhalle (`rayhalle.informatik.tu-muenchen.de`) mit der bereitgestellten Datei `main_bt` kompiliert werden kann. Anderenfalls kann eine Korrektur nicht garantiert werden. Achten Sie darauf, dass Ihr Quelltext ausreichend kommentiert ist.
Schicken Sie die Lösung per Email mit dem Betreff `[GAD] Gruppe <Gruppennummer>` an Ihren Tutor.

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 $\mathcal{O}(n)$.

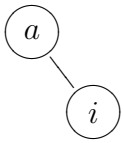
Aufgabe 2

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).

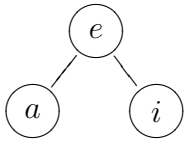
Lösungsvorschlag

Lösungsvorschlag:
Einfügen des Schlüssels 1:

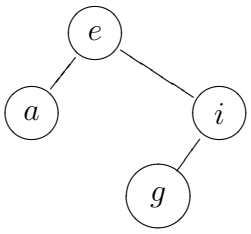
Einfügen des Schlüssels i:



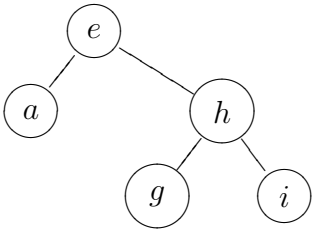
Einfügen des Schlüssels e mit Doppelrotation:



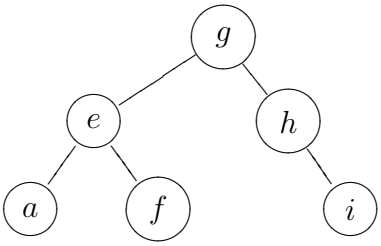
Einfügen des Schlüssels g:



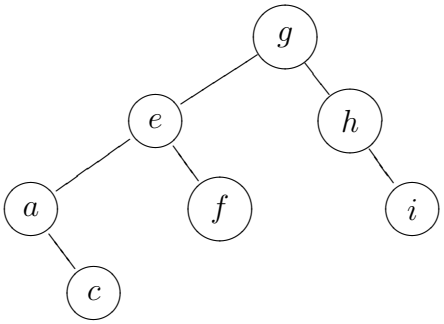
Einfgen des Schlüssels h mit Doppelrotation:



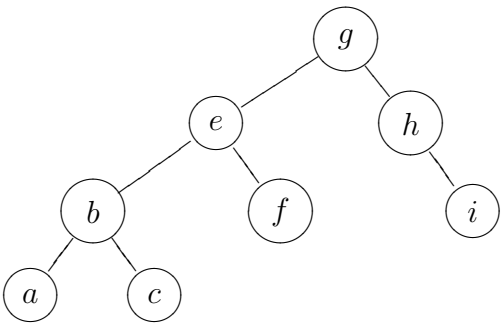
Einfügen des Schlüssels f mit Doppelrotation:
smallskip



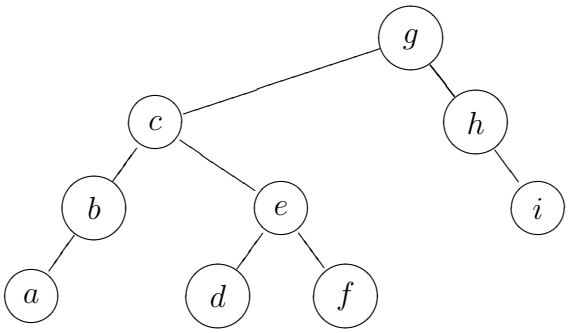
Einfügen des Schlüssels c:



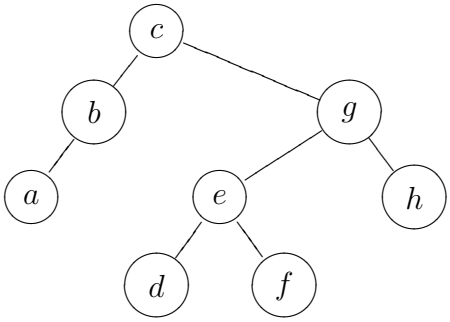
Einfügen des Schlüssels b mit Doppelrotation:



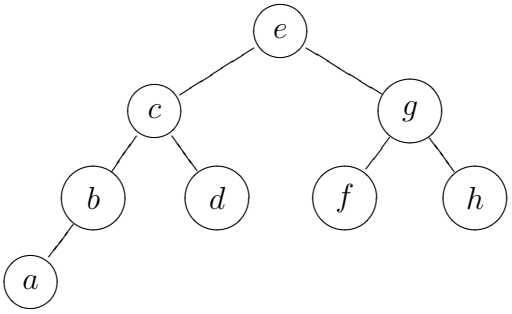
Einfügen des Schlüssels d mit Doppelrotation:



Löschen des Schlüssels i mit Einzelrotation:



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

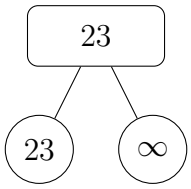


Aufgabe 3

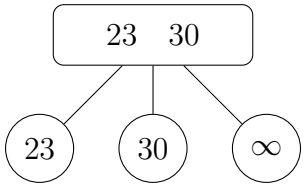
Führen Sie auf einem anfangs leeren $(2, 4)$-Baum folgende Operationen aus und zeichnen Sie die Zwischenergebnisse: `insert(23)`, `insert(30)`, `insert(13)`, `insert(6)`, `insert(40)`, `insert(80)`, `insert(62)`, `insert(75)`, `insert(28)`, `insert(21)`, `insert(29)`, `remove(62)`, `remove(75)`, `remove(13)`.

Lösungsvorschlag

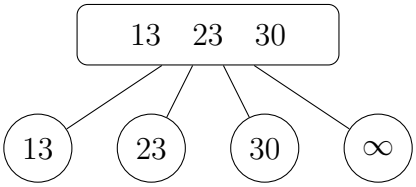
`insert(23)`:



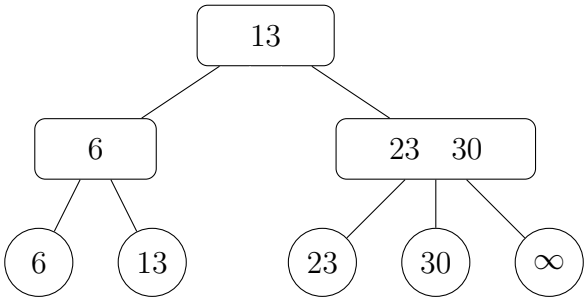
`insert(30)`:



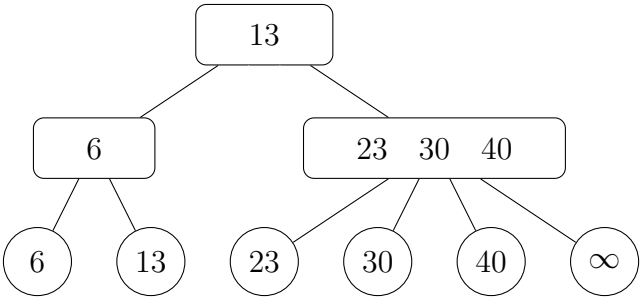
`insert(13)`:



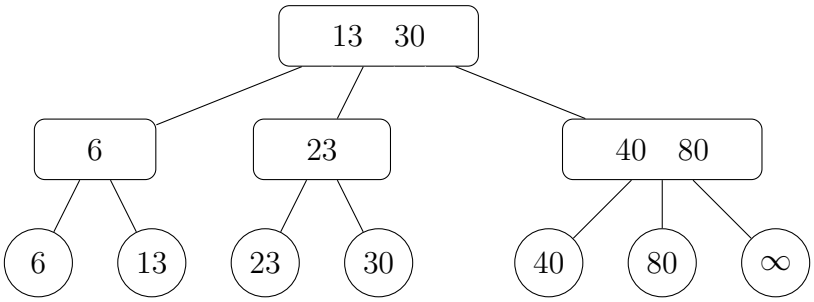
`insert(6)`:



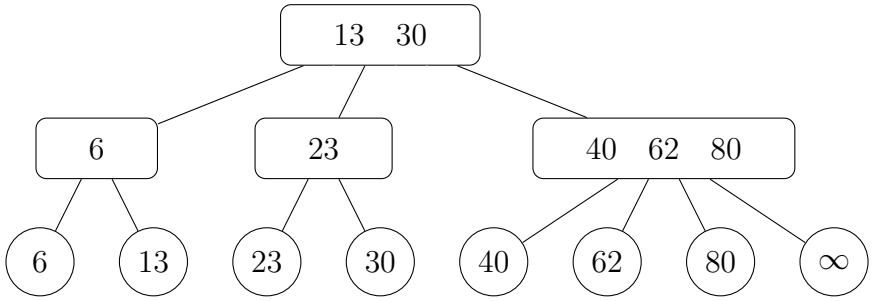
`insert(40)`:



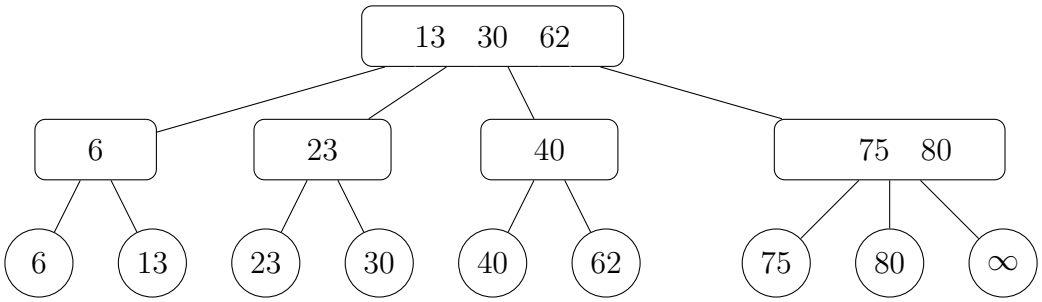
`insert(80)`:



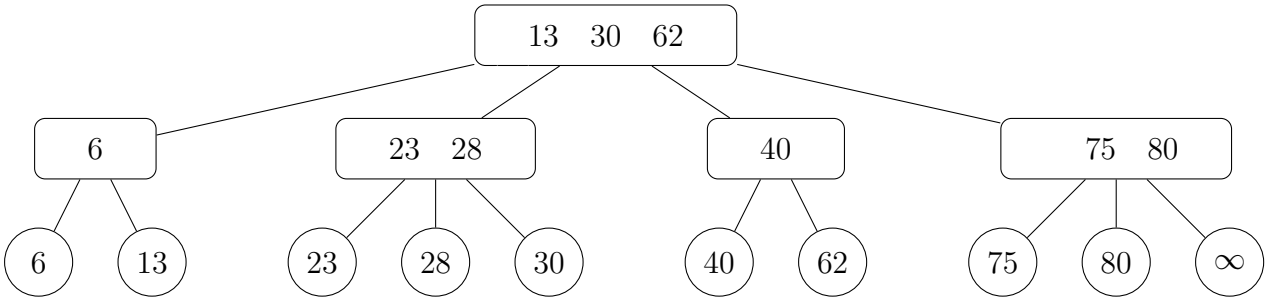
`insert(62)`:



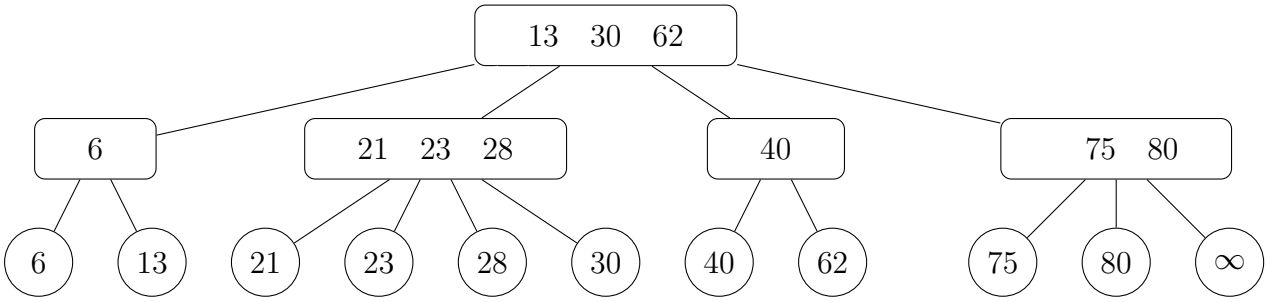
`insert(75)`:



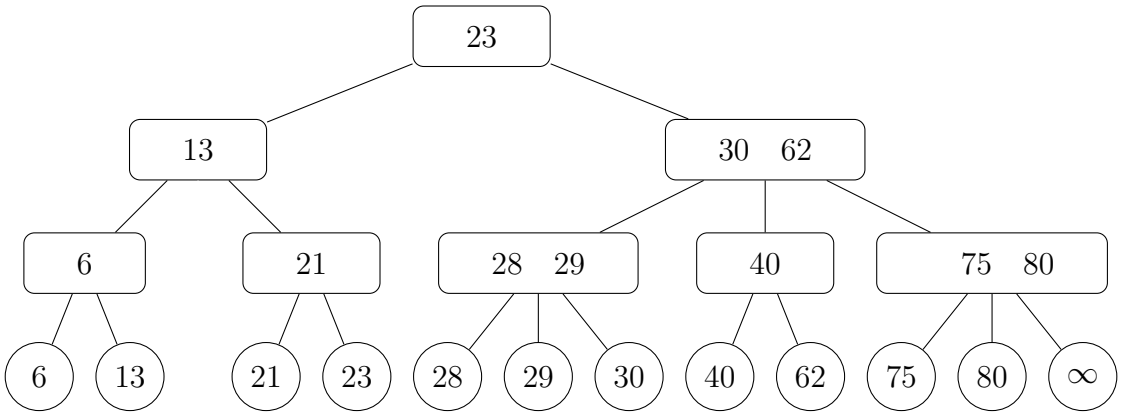
`insert(28)`:



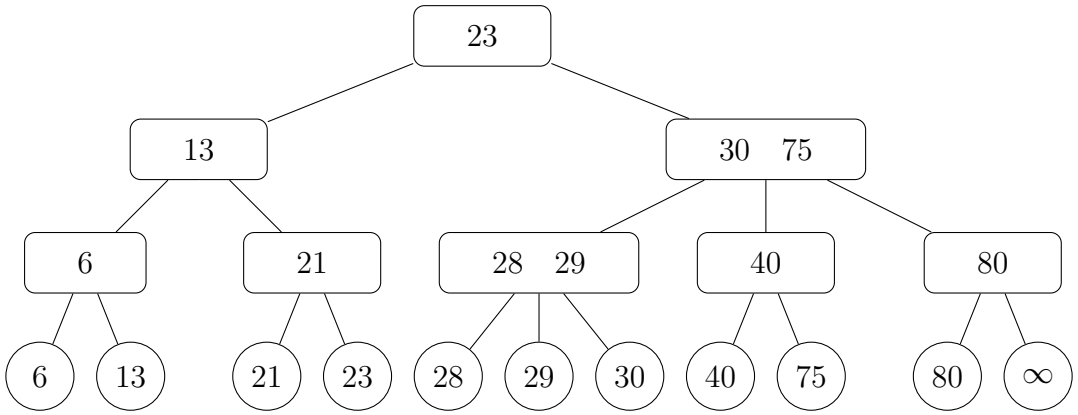
`insert(21)`:



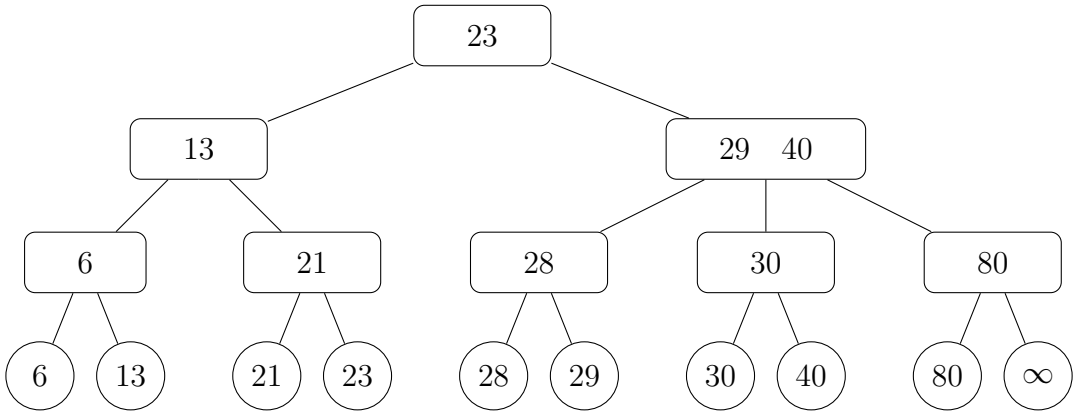
textttinsert(29):



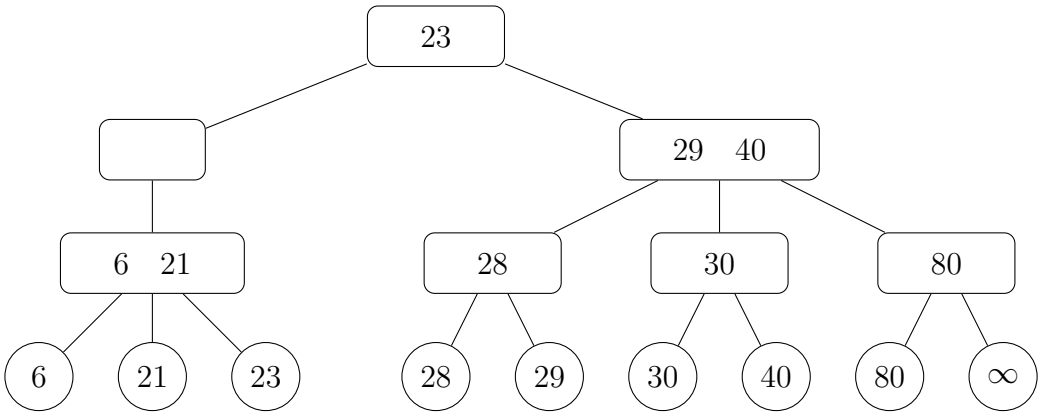
`remove(62)`:



`remove(75)`:



`remove(13)` Teil a):



`remove(13)` Teil b):