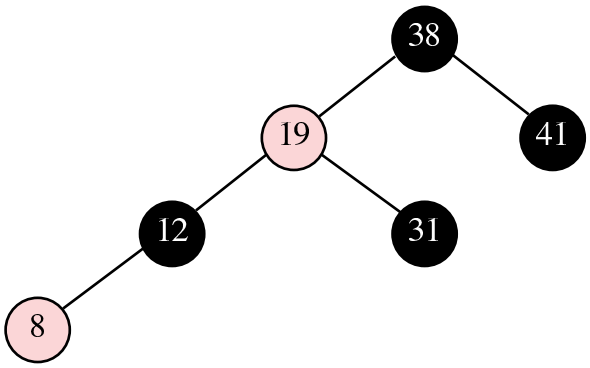
Universität des Saarlandes
FR 6.2 Informatik
Prof. Dr. Hans-Peter Lenhof
Dipl. Inform. Andreas Hildebrandt

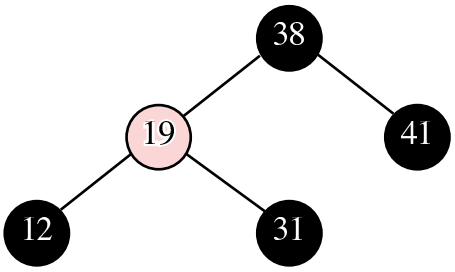
Programmierung II, SS 2003
Musterlösung zur 9. Übung

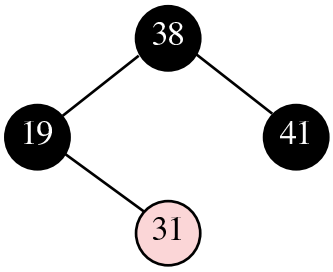
Aufgabe 1: Binäre Suchbäume (10 Punkte)

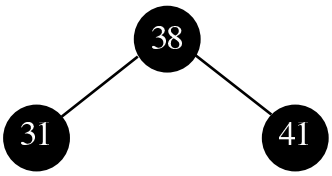
- Da die in der Vorlesung vorgestellte Implementierung in diesem Fall eine lineare Kette, also einen maximal unbalancierten Baum erzeugt, und jeweils unten an den letzten Knoten angehängt wird, muss vor jedem *einfügen* die gesamte bisherige Kette abgelaufen werden. Also läuft die i-te *einfüge*-Operation in $\mathcal{O}(i)$. Damit haben wir als Gesamtkosten für die Sequenz: $\sum_{i=1}^{n} ci = \frac{1}{2}n(n+1) \in \mathcal{O}(n^2)$.
- In diesem Fall erzeugen wir offenbar einen balancierten binären Baum! Im i-ten *einfüge*-Aufruf kann dieser dann maximal die Höhe $\log_2(i)$, besitzen, und wir erhalten als Gesamtkosten der Sequenz: $\sum_{i=1}^{n} \log_2(i) = \log_2(\Pi_{i=1}^{n} i) = \log_2(n!)$
- In diesem Fall erhalten wir einen Baum mit genau einem Knoten. Da wir z.B. vorne an die Liste in $\mathcal{O}(1)$ anhängen können, erhalten wir eine Gesamtlaufzeit von $\mathcal{O}(n)$.
- Im worst-case erhalten wir genau das Verhalten, das wir aus dem Algorithmus der Vorlesung kennen, d.h. wir erzeugen in $\mathcal{O}(n^2)$ eine lineare Kette.
  Wie sieht es nun im Durchschnitt aus? Ganz informell gilt: der Baum wächst an allen Enden etwa gleich schnell. Daher erwarten wir einen balancierten Baum und damit das Verhalten wie im 2. Fall $\Rightarrow \in \mathcal{O}(n \log_2(n))$

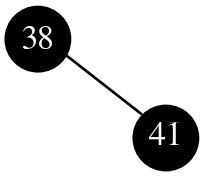
Aufgabe 2: Rot-Schwarz-Bäume, Teil 1 (10 Punkte)


38
19
41
12
31
8


38
19
41
12
31


38
19
41
31


38
31
41


38
41


41

Aufgabe 3: Erwartungswerte (10 Punkte)
Für den Erwartungswert benötigen wir die Wahrscheinlichkeit $p(X = i)$, die wir hier offenbar nicht zur Verfügung haben. Wie können wir sie uns beschaffen? Da X nur ganzzahlige Werte annehmen kann, gilt offenbar $p(X = i) = p(X \geq i) - p(X \geq i + 1)$.

Damit können wir nun den Erwartungswert $E[X]$ bestimmen:

$$
\begin{aligned}
E[X] &= \sum_{i=0}^{\infty} p(X = i)i \\
&= \sum_{i=0}^{\infty} i(p(X \geq i) - p(X \geq i + 1)) \\
&= \sum_{i=1}^{\infty} i(\frac{1}{i}\sum_{k=1}^{i}\frac{1}{k^2} - \frac{1}{i+1}\sum_{k=1}^{i+1}\frac{1}{k^2}) \\
&= \sum_{i=1}^{\infty}(\underbrace{\sum_{k=1}^{i}\frac{1}{k^2}}_{:=a_i} - (\sum_{k=1}^{i+1}\frac{1}{k^2} - \frac{1}{i+1}\sum_{k=1}^{i+1}\frac{1}{k^2})) \\
&= \sum_{i=1}^{\infty}(a_i - a_{i+1}) + \sum_{i=1}^{\infty}\frac{1}{i+1}\sum_{k=1}^{i+1}\frac{1}{k^2} \\
&= \sum_{i=1}^{\infty}\frac{1}{i+1}\sum_{k=1}^{i+1}\frac{1}{k^2} - \sum_{i=1}^{\infty}(a_{i+1} - a_i) \\
&= \sum_{i=1}^{\infty}\frac{1}{i+1}\sum_{k=1}^{i+1}\frac{1}{k^2} - \lim_{b\to\infty}\left\{\sum_{i=1}^{b}(a_{i+1} - a_i)\right\} \\
&= \sum_{i=1}^{\infty}\frac{1}{i+1}\sum_{k=1}^{i+1}\frac{1}{k^2} - \lim_{b\to\infty}(a_{b+1} - a_1) \\
&= \sum_{i=1}^{\infty}\frac{1}{i+1}\underbrace{\sum_{k=1}^{i+1}\frac{1}{k^2}}_{\geq 1} - \underbrace{\sum_{k=1}^{\infty}\frac{1}{k^2}}_{=\frac{\pi^2}{6}} + \sum_{k=1}^{1}\frac{1}{k^2} \\
&\geq \sum_{i=2}^{\infty}\frac{1}{i} + 1 - \frac{\pi^2}{6} \\
&= \sum_{i=1}^{\infty}\frac{1}{i} - \frac{\pi^2}{6} \\
&= \infty\blacksquare
\end{aligned}
$$

Der Erwartungswert ist also unendlich! Kann das sein? Natürlich. Wenn die Gewinnwahrscheinlichkeit nur hinreichend gering ist, dann benötigt man natürlich unendlich viele Versuche, bis man erfolgreich ist!

Aufgabe 4: 2–4 Bäume (10 Punkte)

(a) Sehen wir uns einfach die Extremfälle an: wenn jeder Knoten im Baum immer genau 4 Kinder hat, dann hat der Baum offenbar eine Höhe von $\lceil \log_4(n) \rceil$. Wenn aber jeder Knoten nur genau 2 Kinder hat dann erhalten wir einen binären Baum, für dessen Höhe bekanntlich gilt: $h(n) \leq \lfloor \log_2(n) \rfloor$.

$$\Rightarrow \lceil \log_4(n) \rceil \leq h(n) \leq \lfloor \log_2(n) \rfloor$$

(i) Suche: wir hangeln uns von der Wurzel bis zu den Blättern, indem wir in jedem Schritt das linkeste Kind des aktuellen Knoten besuchen, welches einen Schlüssel größer oder gleich dem gesuchten Element enthält, bis wir schließlich bei den Blättern angekommen sind. Da die Anzahl von Kindern jedes Knotens in $\Theta(1)$ liegt, benötigen wir auf jeder Ebene des Baumes $\Theta(1)$ Operationen. Damit ist die Laufzeit offenbar linear in der Höhe des Baumes, und damit gilt: $T(n) \in \Omega(\log_4(n))$ und $T(n) \in \mathcal{O}(\log_2(n))$, und da alle Logarithmen gleich schnell wachsen haben wir schließlich $T(n) \in \Theta(\log_2(n))$.

(ii) Einfügen: wir führen zuerst eine Suche nach dem Element durch fügen anschliessend das einzufügende Element an den zuletzt betrachteten inneren Knoten an.
Löschen: wir führen wieder eine Suche nach dem Element durch und streichen es aus der Kindliste des letzten betrachteten inneren Knotens falls es im Baum enthalten ist.
Beide Operationen benötigen offenbar $T_{\text{suche}} + \Theta(1)$ Zeit $\Rightarrow T(n) \in \Theta(\log_2(n))$.

(iii) Falls wir die 2–4 - Eigenschaft zerstört haben sollten hat mindestens ein Knoten entweder mehr als 4 oder weniger als 2 Kinder. Da wir jeweils nur entweder ein Blatt hinzugefügt oder ein Blatt entfernt haben, existiert nun schlimmstenfalls genau ein Knoten mit nur einem Kind oder genau ein Knoten mit 5 Kindern.
Beim *Spalten* ersetzen wir einen 5-er Knoten durch einen 3er Knoten und einen 2er Knoten. Dadurch erhält der Vaterknoten ein weiteres Kind. Falls er dadurch zum 5er Knoten wird, spalten wir nun ihn auf usw., bis wir an der Wurzel ankommen. Falls wir die Wurzel spalten müssen setzen wir noch eine weitere Wurzel darüber und sind fertig.
Was machen wir mit 1er - Knoten?
Falls wir einen 3er oder 4er Geschwisterknoten haben, nehmen wir diesem ein Kind weg und stecken es in unseren Knoten. Damit ist die 2–4 - Eigenschaft wieder hergestellt.

Sind aber alle Geschwister 2er Knoten, so verschmelzen wir eins dieser Geschwister mit dem 1er Knoten zu einem 3er - Knoten. Dadurch verliert der Elternknoten ein Kind, und wir müssen u.U. wieder verschmelzen. Dies kann sich wieder bis zur Wurzel hinziehen. Falls wir eine Wurzel vom Grad eins erhalten, können wir diese einfach streichen und erhalten einen intakten 2–4 - Baum.

Da wir im worst-case diese Balancierungen bis hin zur Wurzel ziehen müssen, erhalten wir schlimmstenfalls $\mathcal{O}(\log_2(n))$ Rebalancierungsoperationen.

Aufgabe 5. (Binäre Suchbäume, Implementierung Bonuspunkte)
10 Den folgenden Code finden Sie in der Datei `BinSuchbaum.h`:

```
#include <iostream>

using namespace std;

// Unsere Bin"arbaumklasse
template <typename T>
class BinSuchbaum
{
  // Eine Hilfsklasse zum Speichern der Baumelemente. Diese brauchen wir
  // nicht nach aussen sichtbar zu machen, da wir sie nur intern verwenden
  class Knoten
  {
    public:
      // Wir brauchen einen Defaultkonstruktor
      Knoten() : vater(), kind_links(), kind_rechts(), schluessel() {};

      // Ein praktischer Alternativkonstruktor
      Knoten(Knoten* vaterKnoten, const T& myKey) : vater(vaterKnoten),
          kind_links(), kind_rechts(), schluessel(myKey) {};

      // Der Destrurktor muss nichts tun.
      ~Knoten() {};

      // Ein Zeiger auf unseren Vaterknoten
      Knoten *vater;

      // Und auf die Kinder
      Knoten *kind_links, *kind_rechts;

      // Schliesslich noch der gespeicherte Schluessel
      T schluessel;
  };

    public:
      // Der Konstruktor unserer Baumklasse. Der uebergebene Schluessel wird
      // in der Wurzel gespeichert.
      BinSuchbaum(const T& schluessel);

      // Der Destruktor unserer Baumklasse
      ~BinSuchbaum();
```

```
      // Einfuegen eines Knotens mit Schluessel s
      void einfuegen(const T& s);

      // Loeschen eines Knotens mit Schluessel s
      void loeschen(const T& s);

      // Diese Hilfsfunktion ermoeglicht die Ausgabe auf cout
      void rek_print(Knoten *x) const;

      // "Syntaktischer Zucker"
      void print() const;

      // Rekursive Suche
      Knoten* rek_suche(Knoten* x, const T& s) const;

      // "Syntaktischer Zucker"
      bool suche(const T& s) const;

      // Nachfolgerknoten
      Knoten* nachfolger(Knoten *x);

      // Minimum im Baum
      Knoten *rek_minimum(Knoten* x);

      // Maximum im Baum
      Knoten *rek_maximum(Knoten* x);

      // Syntaktischer Zucker
      T maximum();
      T minimum();

    protected:
      // Die Wurzel des Baumes
      Knoten* wurzel_;

      // Eine Hilfsfunktion zum Loeschen des Baumes
      void rek_delete(Knoten* k);
};

template<typename T>
BinSuchbaum<T>::BinSuchbaum(const T& s)
{
  // Wir benoetigen natuerlich einen Wurzelknoten
  wurzel_ = new Knoten(NULL, s);
}

template<typename T>
BinSuchbaum<T>::~BinSuchbaum()
```

```
{
  // Wir haben Speicher von Hand allokiert, und diesen muessen wir
  // nun freigeben. Dazu muessen wir den Baum so traversieren, dass
  // immer zuerst die Kinder und dann erst die Wurzel geloescht werden.
  rek_delete(wurzel_);
}

template<typename T>
void BinSuchbaum<T>::rek_delete(Knoten *k)
{
  // Diese Funktion fuehrt einen Postorder - Walk ueber den Baum aus
  // und loescht dabei die Knoten
  if (k != NULL)
  {
    rek_delete(k->kind_links);
    rek_delete(k->kind_rechts);

    if (k->vater != NULL)
    {
      if (k->vater->kind_links == k)
      {
        k->vater->kind_links = NULL;
      }
      else
      {
        k->vater->kind_rechts = NULL;
      }
    }
    delete k;
  }
}

template<typename T>
void BinSuchbaum<T>::einfuegen(const T& s)
{
  Knoten *z = new Knoten(NULL, s);

  // Wir brauchen jeweils einen aktuellen Knoten und dessen Vater
  Knoten *x = wurzel_;
  Knoten *p = NULL;

  while (x != NULL)
  {
    p = x;
    if (x->schluessel < z->schluessel)
    {
      x = x->kind_rechts; // Suche geht rechts weiter
    }
```

```
    else
    {
      x = x->kind_links; // Suche geht links weiter
    }
  }

  z->vater = p;

  if (p==NULL) // In diesem Fall ist z die neue Wurzel
  {
    wurzel_ = z;
  }
  else
  {
    if (z->schluessel < p->schluessel)
    {
      p->kind_links = z; // links einfuegen
    }
    else
    {
      p->kind_rechts = z; // rechts einfuegen
    }
  }
}

template <typename T>
void BinSuchbaum<T>::rek_print(Knoten *x) const
{
  if (x != NULL)
  {
    rek_print(x->kind_links);
    cout << x->schluessel << endl;
    rek_print(x->kind_rechts);
  }
}

template <typename T>
void BinSuchbaum<T>::print() const
{
  rek_print(wurzel_);
}

template <typename T>
typename BinSuchbaum<T>::Knoten* BinSuchbaum<T>::rek_suche(Knoten *x, const T& s) con
{
  // Diese Hilfsfunktion durchsucht rekursiv den Teilbaum
  // der von x aufgespannt wird
  Knoten *ergebnis = NULL;
```

```
  if (x!=NULL)
  {
    if (x->schluessel == s)
    {
      ergebnis = x;
    }
    else
    {
      if (s < x->schluessel)
      {
        ergebnis = rek_suche(x->kind_links, s);
      }
      else
      {
        ergebnis = rek_suche(x->kind_rechts, s);
      }
    }
  }

  return ergebnis;
}

// "Syntaktischer Zucker"
template <typename T>
bool BinSuchbaum<T>::suche(const T& s) const
{
  return ( rek_suche(wurzel_, s) != NULL );
}

// Loeschen eines Knotens mit Schluessel s
template <typename T>
void BinSuchbaum<T>::loeschen(const T& s)
{
  // erst mal muessen wir den Knoten mit Schluessel s finden
  Knoten *z = rek_suche(wurzel_, s);
  Knoten *x, *y;

  if (z == NULL) // Der Schluessel ist gar nicht enthalten
  {
    return;
  }

  if ((z->kind_links == NULL) || (z->kind_rechts == NULL))
  {
    // Nur ein Kind -> "Splice out z"
    y = z;
  }
```

```
  else
  {
    // Zwei Kinder -> "Splice out successor"
    y = nachfolger(z);
  }

  // Weiter im Text
  if (y->kind_links != NULL)
  {
    x = y->kind_links;
  }
  else
  {
    x = y->kind_rechts;
  }

  if (x != NULL)
  {
    x->vater = y->vater;
  }

  if (y->vater == NULL)
  {
    // Wir haben eine neue Wurzel
    wurzel_ = x;
  }
  else
  {
    if (y == y->vater->kind_links)
    {
      // y ist ein linkes Kind
      y->vater->kind_links = x;
    }
    else
    {
      y->vater->kind_rechts = x;
    }
  }

  if ( y != z )
  {
    z->schluessel = y->schluessel;
  }

  delete y;
}

// Nachfolger im Baum
```

```
template <typename T>
typename BinSuchbaum<T>::Knoten* BinSuchbaum<T>::nachfolger(Knoten *x)
{
  if (x->kind_rechts != NULL)
  {
    return rek_minimum(x->kind_rechts);
  }

  Knoten* y = x->vater;

  while ((y!=NULL) && (x == y->kind_rechts))
  {
    x = y;
    y = x->vater;
  }

  return y;
}

// Minimum im Teilbaum
template <typename T>
typename BinSuchbaum<T>::Knoten* BinSuchbaum<T>::rek_minimum(Knoten *x)
{
  while (x->kind_links != NULL)
  {
    x = x->kind_links;
  }

  return x;
}

// Maximum im Teilbaum
template <typename T>
typename BinSuchbaum<T>::Knoten* BinSuchbaum<T>::rek_maximum(Knoten *x)
{
  while (x->kind_rechts != NULL)
  {
    x = x->kind_rechts;
  }

  return x;
}

// Syntaktischer Zucker
template <typename T>
T BinSuchbaum<T>::minimum()
{
return rek_minimum(wurzel_)->schluessel;
```

```
}

// Syntaktischer Zucker
template <typename T>
T BinSuchbaum<T>::maximum()
{
return rek_maximum(wurzel_)->schluessel;
}
```

Und ein paar Tests in `BinSuchbaum.cpp`

```
#include "BinSuchbaum.h"

int main(int argc, char **argv)
{
  BinSuchbaum<float> b(0.);

  srand48(42);

  for (int i=0; i<10; i++)
  {
    b.einfuegen(drand48());
  }

  b.print();

  b.einfuegen(4711.);

  if (b.suche(4711) && !b.suche(42.))
  {
    cout << "4711 ist drin, 42 nicht!" << endl;
  }
  else
  {
    cout << "Mist!" << endl;

    return 1;
  }

  b.loeschen(4711);

  if (!b.suche(4711))
  {
    b.print();
    cout << "4711 erfolgreich entfernt!" << endl;
  }
  else
  {
    cout << "Mist!" << endl;
```

```
    return 1;
  }

  return 0;

}
```
