Inhaltsverzeichnis

1. OOP - Teil 1

1.1. Objekt „Bruch“

```
#include <stdio.h>

class c_bruch
{
   private:
   int zaehler;
   int nenner;
   int ggT (int zahl1, int zahl2);
   void kuerzen (void);

   public:
   c_bruch (void);
   c_bruch (int i_zaehler, int i_nenner);
   c_bruch (int i_wert);
   int get_zaehler (void);
   int get_nenner  (void);
   void print (void);
   void add   (const c_bruch &summand);
   void subtr (const c_bruch &subtrahend);
   void mult  (const c_bruch &faktor);
   void div   (const c_bruch &divisor);
};

int c_bruch::ggT (int zahl1, int zahl2)
{
   if (zahl2 == 0)
      return zahl1;
   else
      return ggT (zahl2, zahl1 % zahl2);
}

void c_bruch::kuerzen (void)
{
   int max_teiler = ggT (zaehler, nenner);

   zaehler = zaehler / max_teiler;
   nenner  = nenner  / max_teiler;
}

c_bruch::c_bruch (void)
{
   zaehler = 0;
   nenner  = 1;
}

c_bruch::c_bruch (int i_zaehler, int i_nenner)
{
   zaehler = i_zaehler;
   nenner  = i_nenner;
}

c_bruch::c_bruch (int wert)
{
   zaehler = wert;
   nenner  = 1;
}

int c_bruch::get_zaehler (void)
{
   return zaehler;
}

int c_bruch::get_nenner  (void)
{
   return nenner;
}

void c_bruch::print (void)
{
   printf ("%d/%d\n", zaehler, nenner);
}

void c_bruch::add (const c_bruch &summand)
{
   zaehler = zaehler * summand.nenner + nenner * summand.zaehler;
   nenner  = nenner  * summand.nenner;
```

```
   kuerzen();
}

void c_bruch::subtr (const c_bruch &subtrahend)
{
   zaehler = zaehler * subtrahend.nenner - nenner * subtrahend.zaehler;
   nenner  = nenner  * subtrahend.nenner;
   kuerzen();
}

void c_bruch::mult (const c_bruch &faktor)
{
   zaehler = zaehler * faktor.zaehler;
   nenner  = nenner  * faktor.nenner;
   kuerzen();
}

void c_bruch::div (const c_bruch &divisor)
{
   zaehler = zaehler * divisor.nenner;
   nenner  = nenner  * divisor.zaehler;
   kuerzen();
}

void main (void)
{
   c_bruch bruch1 (1,4);
   c_bruch bruch2 (2,3);

   bruch1.print();
   bruch1.add   (bruch2);
   bruch1.print();
   bruch1.subtr (bruch2);
   bruch1.print();
   bruch1.mult  (bruch2);
   bruch1.print();
   bruch1.div   (bruch2);
   bruch1.print();
}
```

1.2. Objekt „komplexe Zahl“

```
#include <stdio.h>

class c_komplex
{
   private:
   int real;
   int imag;

   public:
   c_komplex (void);
   c_komplex (int i_real, int i_imag);
   c_komplex (int i_wert);
   int get_real (void);
   int get_imag (void);
   void print (void);
   void add   (const c_komplex &summand);
   void subtr (const c_komplex &subtrahend);
   void mult  (const c_komplex &faktor);
};

c_komplex::c_komplex (void)
{
   real = 0;
   imag = 0;
}

c_komplex::c_komplex (int i_real, int i_imag)
{
   real = i_real;
   imag = i_imag;
}

c_komplex::c_komplex (int i_wert)
{
   real = i_wert;
   imag = 0;
}

int c_komplex::get_real (void)
```

```
{
   return real;
}

int c_komplex::get_imag (void)
{
   return imag;
}

void c_komplex::print (void)
{
   printf ("%d + %di\n", real, imag);
}

void c_komplex::add (const c_komplex &summand)
{
   real = real + summand.real;
   imag = imag + summand.imag;
}

void c_komplex::subtr (const c_komplex &subtrahend)
{
   real = real - subtrahend.real;
   imag = imag - subtrahend.imag;
}

void c_komplex::mult (const c_komplex &faktor)
{
   int new_real = real * faktor.real - imag * faktor.imag;
   int new_imag = real * faktor.imag + imag * faktor.real;

   real = new_real;
   imag = new_imag;
}

void main (void)
{
   c_komplex komplex1 (1,4);
   c_komplex komplex2 (2);

   komplex1.print();
   komplex1.add   (komplex2);
   komplex1.print();
   komplex1.subtr (komplex2);
   komplex1.print();
   komplex1.mult  (komplex2);
   komplex1.print();
}
```

1.3. Objekt „komplexer Bruch“

```
#include <stdio.h>

class c_bruch
{
   private:
   int zaehler;
   int nenner;
   int ggT (int zahl1, int zahl2);
   void kuerzen (void);

   public:
   c_bruch (void);
   c_bruch (int i_zaehler, int i_nenner);
   c_bruch (int i_wert);
   int get_zaehler (void);
   int get_nenner  (void);
   void print (void);
   void add   (c_bruch summand);
   void subtr (c_bruch subtrahend);
   void mult  (c_bruch faktor);
   void div   (c_bruch divisor);
   c_bruch operator+ (c_bruch summand);
   c_bruch operator- (c_bruch subtrahend);
   c_bruch operator* (c_bruch faktor);
   c_bruch operator/ (c_bruch divisor);
};

class c_komplex
{
   private:
```

```
   c_bruch real;
   c_bruch imag;

   public:
   c_komplex (void);
   c_komplex (c_bruch i_real, c_bruch i_imag);
   c_komplex (c_bruch i_wert);
   c_bruch get_real (void);
   c_bruch get_imag (void);
   void print (void);
   void add   (c_komplex summand);
   void subtr (c_komplex subtrahend);
   void mult  (c_komplex faktor);
};

int c_bruch::ggT (int zahl1, int zahl2)
{
   if (zahl2 == 0)
      return zahl1;
   else
      return ggT (zahl2, zahl1 % zahl2);
}

void c_bruch::kuerzen (void)
{
   int max_teiler = ggT (zaehler, nenner);

   zaehler = zaehler / max_teiler;
   nenner  = nenner  / max_teiler;
}

c_bruch::c_bruch (void)
{
   zaehler = 0;
   nenner  = 1;
}

c_bruch::c_bruch (int i_zaehler, int i_nenner)
{
   zaehler = i_zaehler;
   nenner  = i_nenner;
}

c_bruch::c_bruch (int wert)
{
   zaehler = wert;
   nenner  = 1;
}

int c_bruch::get_zaehler (void)
{
   return zaehler;
}

int c_bruch::get_nenner  (void)
{
   return nenner;
}

void c_bruch::print (void)
{
   printf ("%d/%d", zaehler, nenner);
}

void c_bruch::add (c_bruch summand)
{
   zaehler = zaehler * summand.nenner + nenner * summand.zaehler;
   nenner  = nenner  * summand.nenner;
   kuerzen();
}

void c_bruch::subtr (c_bruch subtrahend)
{
   zaehler = zaehler * subtrahend.nenner - nenner * subtrahend.zaehler;
   nenner  = nenner  * subtrahend.nenner;
   kuerzen();
}

void c_bruch::mult (c_bruch faktor)
{
```

```
   zaehler = zaehler * faktor.zaehler;
   nenner  = nenner  * faktor.nenner;
   kuerzen();
}

void c_bruch::div (c_bruch divisor)
{
   zaehler = zaehler * divisor.nenner;
   nenner  = nenner  * divisor.zaehler;
   kuerzen();
}

c_bruch c_bruch::operator+ (c_bruch summand)
{
   c_bruch tmp = *this;

   tmp.add (summand);
   return tmp;
}

c_bruch c_bruch::operator- (c_bruch subtrahend)
{
   c_bruch tmp = *this;

   tmp.subtr (subtrahend);
   return tmp;
}

c_bruch c_bruch::operator* (c_bruch faktor)
{
   c_bruch tmp = *this;

   tmp.mult (faktor);
   return tmp;
}

c_bruch c_bruch::operator/ (c_bruch divisor)
{
   c_bruch tmp = *this;

   tmp.div (divisor);
   return tmp;
}

c_komplex::c_komplex (void)
{
}

c_komplex::c_komplex (c_bruch i_real, c_bruch i_imag)
{
   real = i_real;
   imag = i_imag;
}

c_komplex::c_komplex (c_bruch i_wert)
{
   real = i_wert;
   imag = 0;
}

c_bruch c_komplex::get_real (void)
{
   return real;
}

c_bruch c_komplex::get_imag (void)
{
   return imag;
}

void c_komplex::print (void)
{
   real.print();
   printf ("+i");
   imag.print();
   printf ("\n");
}

void c_komplex::add (c_komplex summand)
{
```

```
   real.add (summand.real);
   imag.add (summand.imag);
}

void c_komplex::subtr (c_komplex subtrahend)
{
   real.subtr (subtrahend.real);
   imag.subtr (subtrahend.imag);
}

void c_komplex::mult (c_komplex faktor)
{
   c_bruch new_real = real * faktor.real - imag * faktor.imag;
   c_bruch new_imag = real * faktor.imag + imag * faktor.real;

   real = new_real;
   imag = new_imag;
}

void main (void)
{
   c_komplex komplex1 (1,4);
   c_komplex komplex2 (2);

   komplex1.print();
   komplex1.add   (komplex2);
   komplex1.print();
   komplex1.subtr (komplex2);
   komplex1.print();
   komplex1.mult  (komplex2);
   komplex1.print();
}
```

1.4. Objekt „Datum“

```
#include <stdio.h>

class c_datum
{
   private:
   int tag;
   int monat;
   int jahr;
   int get_max_days (int i_jahr, int i_monat);

   public:
   c_datum (void);
   c_datum (int i_tag, int i_monat, int i_jahr);
   void days_forw (int i_days);
   void days_back (int i_days);
   void display (void);
};

int c_datum::get_max_days (int i_jahr, int i_monat)
{
   switch (i_monat)
   {
      case 2:  if (((i_jahr % 4 == 0) && (i_jahr % 100 != 0)) ||
               (i_jahr % 400 == 0))
                return 29;
              else
                return 28;
      case 4:
      case 6:
      case 9:
      case 11: return 30;
   };
   return 31;
}

c_datum::c_datum (void)
{
   tag   = 1;
   monat = 1;
   jahr  = 1900;
}

c_datum::c_datum (int i_tag, int i_monat, int i_jahr)
{
   jahr = i_jahr;
   if ((i_monat >= 1) && (i_monat <= 12))
```

```
      monat = i_monat;
   else
      monat = 1;

   if ((i_tag >= 1) && (i_tag <= get_max_days (jahr, monat)))
      tag   = i_tag;
   else
      tag = 1;
}

void c_datum::days_forw (int i_days)
{
   while (tag + i_days > get_max_days (jahr, monat))
   {
      i_days -= (get_max_days (jahr, monat) - tag +1);
      tag = 1;
      monat++;
      if (monat > 12)
      {
         monat = 1;
         jahr++;
      }
   }

   tag += i_days;
}

void c_datum::days_back (int i_days)
{
   while (i_days > tag)
   {
      i_days -= tag;
      monat--;
      if (monat == 0)
      {
         monat = 12;
         jahr --;
      }
      tag = get_max_days (jahr, monat);
   }
   tag -= i_days;
}

void c_datum::display (void)
{
   if (tag < 10)
      printf ("0");
   printf ("%d.", tag);

   if (monat < 10)
      printf ("0");
   printf ("%d.", monat);

   if (jahr < 1000)
      printf ("0");
   if (jahr < 100)
      printf ("0");
   if (jahr < 10)
      printf ("0");
   printf ("%d\n", jahr);
}

void main (void)
{
   c_datum datum1 (1,1,2002);
   c_datum datum2 (1,1, 10);

   datum1.display ();
   datum1.days_forw (150);
   datum1.display ();
   datum1.days_back (160);
   datum1.display ();

   datum2.display ();
   datum2.days_back (2);
   datum2.display ();
}
```

1.5. Objekt „Liste“

```
#include <stdio.h>
```

```
#include <stdlib.h>

class c_liste
{
   private:
   int *element;
   int anzahl;
   int belegt;

   public:
   c_liste (int i_anzahl);
   ~c_liste ();
   void append (int neu);
   void remove (int index);
   int  search (int wert);
   int  get_element (int index);
};

c_liste::c_liste (int i_anzahl)
{
   anzahl = i_anzahl;
   belegt = 0;
   element = (int *) malloc (anzahl * sizeof (int));
   if (element == NULL)
      anzahl = 0;
}

c_liste::~c_liste ()
{
   if (element != NULL)
      free (element);
}

void c_liste::append (int neu)
{
   if (belegt < anzahl)
   {
      element [belegt] = neu;
      belegt ++;
   }
}

void c_liste::remove (int index)
{
   int i;

   if ((index >= 0) && (index < belegt))
   {
      for (i=index; i<belegt-1; i++)
         element[i] = element [i+1];
      belegt --;
   }
}

int  c_liste::search (int wert)
{
   int i;
   for (i=0; i<belegt; i++)
      if (element[i] == wert)
         return i;

   return -1;
}

int  c_liste::get_element (int index)
{
   if ((index >= 0) && (index < belegt))
      return element [index];
   else
      return -1;
}

void main (void)
{
   c_liste liste (10);

   liste.append (3);
   liste.append (42);
   liste.append (97);
```

```
   printf ("Element 42 steht an Position %d\n", liste.search (42)+1);

   liste.remove (0);

   printf ("Element 97 steht an Position %d\n", liste.search (97)+1);
}
```

2. OOP - Vererbung

2.1. Objekt „Liste“

```
#include <stdio.h>
#include <stdlib.h>

class c_liste
{
   protected:
   int *element;
   int anzahl;
   int belegt;

   public:
   c_liste (int i_anzahl);
   ~c_liste ();
   void append (int neu);
   void remove (int index);
   int  search (int wert);
   int  get_element (int index);
   void display (void);
};

class c_sort_list : public c_liste
{
   public:
   c_sort_list (int i_anzahl);
   void append (int neu);
   int  search (int wert);
};

c_liste::c_liste (int i_anzahl)
{
   anzahl = i_anzahl;
   belegt = 0;
   element = (int *) malloc (anzahl * sizeof (int));
   if (element == NULL)
      anzahl = 0;
}

c_liste::~c_liste ()
{
   if (element != NULL)
      free (element);
}

void c_liste::append (int neu)
{
   if (belegt < anzahl)
   {
      element [belegt] = neu;
      belegt ++;
   }
}

void c_liste::remove (int index)
{
   int i;

   if ((index >= 0) && (index < belegt))
   {
      for (i=index; i<belegt-1; i++)
         element[i] = element [i+1];
      belegt --;
   }
}

int  c_liste::search (int wert)
{
   int i;
   for (i=0; i<belegt; i++)
      if (element[i] == wert)
         return i;

   return -1;
}

int  c_liste::get_element (int index)
{
```

```
   if ((index >= 0) && (index < belegt))
      return element [index];
   else
      return -1;
}

void c_liste::display (void)
{
   int i;

   for (i=0; i<belegt; i++)
      printf ("%d  ", element[i]);
   printf ("\n");
}

c_sort_list::c_sort_list (int i_anzahl) : c_liste (i_anzahl)
{
}

void c_sort_list::append (int neu)
{
   int i=0,j;

   if (belegt < anzahl)
   {
      while ((i < belegt) && (element[i] <= neu))
         i++;

      for (j=belegt; j>i; j--)
         element[j] = element[j-1];

      element[i] = neu;
      belegt ++;
   }
}

int  c_sort_list::search (int wert)
{
   int i=0;

   while ((i < belegt) && (element[i] < wert))
      i++;

   if (element[i] == wert)
      return i;
   else
      return -1;
}

void main (void)
{
   c_liste     liste (10);
   c_sort_list sort_list (10);

   liste.append (97);
   liste.append (42);
   liste.append (3);
   liste.display ();

   sort_list.append (97);
   sort_list.append (3);
   sort_list.append (42);
   sort_list.display ();
}
```

2.2. Objekt „Fahrzeug“

```
#include <stdio.h>

class c_fahrzeug
{
   protected:
   int speed;
   int max_speed;

   public:
   c_fahrzeug (int i_max_speed);
   virtual void speed_up   (void) = 0;
   virtual void speed_down (void) = 0;
   virtual void show_speed (void) = 0;
};
```

```
class c_auto : public c_fahrzeug
{
   protected:
   int beschleunigung;

   public:
   c_auto (int i_max_speed, int i_beschleunigung);
   virtual void speed_up   (void);
   virtual void speed_down (void);
   virtual void show_speed (void);
};

class c_fahrrad : public c_fahrzeug
{
   protected:
   int gang;
   int max_gang;

   public:
   c_fahrrad (int i_max_speed, int i_max_gang);
   virtual void speed_up   (void);
   virtual void speed_down (void);
   virtual void show_speed (void);
};

c_fahrzeug::c_fahrzeug (int i_max_speed)
{
   max_speed = i_max_speed;
   speed     = 0;
}

c_auto::c_auto (int i_max_speed, int i_beschleunigung) : c_fahrzeug (i_max_speed)
{
   beschleunigung = i_beschleunigung;
}

void c_auto::speed_up (void)
{
   if (speed + beschleunigung < max_speed)
      speed += beschleunigung;
   else
      speed = max_speed;
}

void c_auto::speed_down (void)
{
   if (speed - beschleunigung > 0)
      speed -= beschleunigung;
   else
      speed = 0;
}

void c_auto::show_speed (void)
{
   printf ("Geschwindigkeit Auto = %d\n", speed);
}

c_fahrrad::c_fahrrad (int i_max_speed, int i_max_gang) : c_fahrzeug (i_max_speed)
{
   max_gang = i_max_gang;
   gang     = 1;
}

void c_fahrrad::speed_up (void)
{
   if ((speed == max_speed) && (gang < max_gang))
   {
      speed = 0;
      gang ++;
   }

   if (speed < max_speed)
      speed ++;
}

void c_fahrrad::speed_down (void)
{
   if ((speed == 1) && (gang > 1))
   {
```

```
      speed = max_speed + 1;
      gang --;
   }

   if (speed > 0)
      speed --;
}

void c_fahrrad::show_speed (void)
{
   printf ("Geschwindigkeit Fahrrad = %d im %d. Gang\n", speed, gang);
}

void main (void)
{
   c_auto    pkw (150, 10);
   c_fahrrad rad (5, 3);
   int       i;
   char      dummy;

   for (i=0; i<20; i++)
   {
      pkw.speed_up ();
      pkw.show_speed ();
   }

   scanf ("%c", &dummy);

   for (i=0; i<20; i++)
   {
      pkw.speed_down ();
      pkw.show_speed ();
   }

   scanf ("%c", &dummy);

   for (i=0; i<20; i++)
   {
      rad.speed_up ();
      rad.show_speed ();
   }

   scanf ("%c", &dummy);

   for (i=0; i<20; i++)
   {
      rad.speed_down ();
      rad.show_speed ();
   }
}
```

2.3. Objekt „Mitarbeiter“

```
#include <stdio.h>
#include <string.h>

class c_mitarbeiter
{
   protected:
   char name [20];
   int  pers_nr;

   public:
   c_mitarbeiter (char *i_name, int i_pers_nr);
   virtual void visitenkarte (void);
};

class c_leiter : public c_mitarbeiter
{
   protected:
   char abteilung[20];
   int mitarbeiter;

   public:
   c_leiter (char *i_name, int i_pers_nr, char *i_abteilung,
             int i_mitarbeiter);
   virtual void visitenkarte (void);
};

c_mitarbeiter::c_mitarbeiter (char *i_name, int i_pers_nr)
{
```

```
   strcpy (name, i_name);
   pers_nr = i_pers_nr;
}

void c_mitarbeiter::visitenkarte (void)
{
   printf ("Name: %s, Pers-Nr.: %d\n", name, pers_nr);
}

c_leiter::c_leiter (char *i_name, int i_pers_nr, char *i_abteilung,
                    int i_mitarbeiter) : c_mitarbeiter (i_name, i_pers_nr)
{
   strcpy (abteilung, i_abteilung);
   mitarbeiter = i_mitarbeiter;
}

void c_leiter::visitenkarte (void)
{
   c_mitarbeiter::visitenkarte ();
   printf ("Leiter %s, %d Mitarbeiter\n", abteilung, mitarbeiter);
}

void main (void)
{
   c_mitarbeiter mitarbeiter ("Schmidt", 123);
   c_leiter      leiter      ("Müller", 007, "Entwicklung", 3);

   mitarbeiter.visitenkarte ();
   leiter.visitenkarte ();
}
```

3. dynamische Speicherverwaltung

3.1. Objekt „Stack“

```
#include <stdio.h>
#include <stdlib.h>

class c_stack
{
   private:
   int *element;
   int groesse;
   int belegt;

   public:
   c_stack (int i_groesse);
   ~c_stack ();
   void push (int i_wert);
   int pop (void);
};

c_stack::c_stack (int i_groesse)
{
   groesse = 0;
   belegt  = 0;
   element = (int *) malloc (i_groesse * sizeof (int));
   if (element != NULL)
      groesse = i_groesse;
}

c_stack::~c_stack ()
{
   if (element != NULL)
      free (element);
}

void c_stack::push (int i_wert)
{
   if (belegt < groesse)
   {
      element [belegt] = i_wert;
      belegt ++;
   }
}

int c_stack::pop (void)
{
   if (belegt > 0)
   {
      belegt --;
      return element [belegt];
   }
   return -1;
}

void main (void)
{
   c_stack stack (10);

   stack.push (3);
   stack.push (4);
   stack.push (stack.pop () + stack.pop ());
   printf ("3+4=%d\n", stack.pop ());
}
```

3.2. Feld mit Brüchen

```
#include <stdio.h>

class c_bruch
{
   private:
   int zaehler;
   int nenner;
   int ggT (int zahl1, int zahl2);
   void kuerzen (void);

   public:
   c_bruch (void);
   c_bruch (int i_zaehler, int i_nenner);
   c_bruch (int i_wert);
   void init (int i_zaehler, int i_nenner);
```

```
   int get_zaehler (void);
   int get_nenner  (void);
   void print (void);
   void add   (const c_bruch &summand);
   void subtr (const c_bruch &subtrahend);
   void mult  (const c_bruch &faktor);
   void div   (const c_bruch &divisor);
};

int c_bruch::ggT (int zahl1, int zahl2)
{
   if (zahl2 == 0)
      return zahl1;
   else
      return ggT (zahl2, zahl1 % zahl2);
}

void c_bruch::kuerzen (void)
{
   int max_teiler = ggT (zaehler, nenner);

   zaehler = zaehler / max_teiler;
   nenner  = nenner  / max_teiler;
}

c_bruch::c_bruch (void)
{
   init (0, 1);
}

c_bruch::c_bruch (int i_zaehler, int i_nenner)
{
   init (i_zaehler, i_nenner);
}

c_bruch::c_bruch (int wert)
{
   init (wert, 1);
}

void c_bruch::init (int i_zaehler, int i_nenner)
{
   zaehler = i_zaehler;
   nenner  = i_nenner;
}

int c_bruch::get_zaehler (void)
{
   return zaehler;
}

int c_bruch::get_nenner  (void)
{
   return nenner;
}

void c_bruch::print (void)
{
   printf ("%d/%d\n", zaehler, nenner);
}

void c_bruch::add (const c_bruch &summand)
{
   zaehler = zaehler * summand.nenner + nenner * summand.zaehler;
   nenner  = nenner  * summand.nenner;
   kuerzen();
}

void c_bruch::subtr (const c_bruch &subtrahend)
{
   zaehler = zaehler * subtrahend.nenner - nenner * subtrahend.zaehler;
   nenner  = nenner  * subtrahend.nenner;
   kuerzen();
}

void c_bruch::mult (const c_bruch &faktor)
{
   zaehler = zaehler * faktor.zaehler;
   nenner  = nenner  * faktor.nenner;
   kuerzen();
```

```
}

void c_bruch::div (const c_bruch &divisor)
{
   zaehler = zaehler * divisor.nenner;
   nenner  = nenner  * divisor.zaehler;
   kuerzen();
}

void main (void)
{
   c_bruch liste [10];

   liste [0].init (1,4);
   liste [1].init (2,3);
   liste [2].init (4,5);
   liste [3].init (1,7);
   liste [4].init (2,5);
   liste [5].init (5,3);
   liste [6].init (7,4);
   liste [7].init (8,3);
   liste [8].init (5,2);
   liste [9].init (3,1);

   liste [0].add (liste [1]);
   liste [2].subtr (liste [3]);
   liste [4].mult (liste [5]);
   liste [6].div (liste [7]);
   liste [8].add (liste [9]);

   liste [0].print ();
   liste [2].print ();
   liste [4].print ();
   liste [6].print ();
   liste [8].print ();
}
```

3.3. Feld mit komplexen Zahlen

```
#include <stdio.h>

class c_komplex
{
   private:
   int real;
   int imag;

   public:
   c_komplex (void);
   c_komplex (int i_real, int i_imag);
   c_komplex (int i_wert);
   void init (int i_real, int i_imag);
   int get_real (void);
   int get_imag (void);
   void print (void);
   void add   (const c_komplex &summand);
   void subtr (const c_komplex &subtrahend);
   void mult  (const c_komplex &faktor);
};

c_komplex::c_komplex (void)
{
   init (0, 0);
}

c_komplex::c_komplex (int i_real, int i_imag)
{
   init (i_real, i_imag);
}

c_komplex::c_komplex (int i_wert)
{
   init (i_wert, 0);
}

void c_komplex::init (int i_real, int i_imag)
{
   real = i_real;
   imag = i_imag;
}
```

```
int c_komplex::get_real (void)
{
   return real;
}

int c_komplex::get_imag (void)
{
   return imag;
}

void c_komplex::print (void)
{
   printf ("%d + %di\n", real, imag);
}

void c_komplex::add (const c_komplex &summand)
{
   real = real + summand.real;
   imag = imag + summand.imag;
}

void c_komplex::subtr (const c_komplex &subtrahend)
{
   real = real - subtrahend.real;
   imag = imag - subtrahend.imag;
}

void c_komplex::mult (const c_komplex &faktor)
{
   int new_real = real * faktor.real - imag * faktor.imag;
   int new_imag = real * faktor.imag + imag * faktor.real;

   real = new_real;
   imag = new_imag;
}

void main (void)
{
   c_komplex liste [10];

   liste [0].init (1,4);
   liste [1].init (2,3);
   liste [2].init (4,5);
   liste [3].init (1,7);
   liste [4].init (2,5);
   liste [5].init (5,3);
   liste [6].init (7,4);
   liste [7].init (8,3);
   liste [8].init (5,2);
   liste [9].init (3,1);

   liste [0].add (liste [1]);
   liste [2].subtr (liste [3]);
   liste [4].mult (liste [5]);
   liste [6].add (liste [7]);
   liste [8].subtr (liste [9]);

   liste [0].print ();
   liste [2].print ();
   liste [4].print ();
   liste [6].print ();
   liste [8].print ();
}
```

3.4. Liste mit veränderlicher Größe

```
#include <stdio.h>
#include <stdlib.h>

class c_liste
{
   private:
   int *element;
   int anzahl;
   int belegt;
   int delta;

   public:
   c_liste (int i_anzahl, int i_delta);
   ~c_liste ();
   void append (int neu);
```

```
   void remove (int index);
   void display (void);
};

c_liste::c_liste (int i_anzahl, int i_delta)
{
   anzahl = i_anzahl;
   belegt = 0;
   element = (int *) malloc (anzahl * sizeof (int));
   if (element == NULL)
      anzahl = 0;
   delta = i_delta;
}

c_liste::~c_liste ()
{
   if (element != NULL)
      free (element);
}

void c_liste::append (int neu)
{
   int *new_element = NULL;

   if (belegt == anzahl)
   {
      new_element = (int *) realloc (element, (anzahl + delta) * sizeof (int));
      if (new_element != NULL)
      {
         element = new_element;
         anzahl += delta;
      }
   }

   if (belegt < anzahl)
   {
      element [belegt] = neu;
      belegt ++;
   }
}

void c_liste::remove (int index)
{
   int i;

   if ((index >= 0) && (index < belegt))
   {
      for (i=index; i<belegt-1; i++)
         element[i] = element [i+1];
      belegt --;
   }
}

void c_liste::display (void)
{
   int i;

   printf ("%d von %d Elementen belegt\n", belegt, anzahl);
   for (i=0; i<belegt; i++)
      printf ("%d ", element [i]);
   printf ("\n");
}

void main (void)
{
   c_liste liste (3, 2);

   liste.display ();
   liste.append (3);
   liste.display ();
   liste.append (12);
   liste.display ();
   liste.append (7);
   liste.display ();
   liste.append (8);
   liste.display ();
   liste.append (13);
   liste.display ();
   liste.append (42);
   liste.display ();
```

```
}
```

4. Listen

4.1. Einfach verkettete int-Liste

```
#include <stdio.h>
#include <stdlib.h>

struct t_liste
{
   int      wert;
   t_liste *next;
};

void main (void)
{
   t_liste *wurzel  = NULL;
   t_liste *last    = NULL;
   t_liste *current = NULL;
   int      zahl;
   int      summe = 0;

   printf ("Bitte Summanden eingeben (0=Ende)");

   do
   {
      scanf ("%d", &zahl);
      if (zahl != 0)
      {
         if (last == NULL)
         {
            wurzel = (t_liste *) malloc (sizeof (t_liste));
            last = wurzel;
         }
         else
         {
            last->next = (t_liste *) malloc (sizeof (t_liste));
            last = last->next;
         }
         last->wert = zahl;
         last->next = NULL;
      }
   }
   while (zahl != 0);

   current = wurzel;
   while (current != NULL)
   {
      summe += current->wert;
      current = current->next;
   }

   printf ("Summe = %d\n", summe);

   while (wurzel != NULL)
   {
      current = wurzel;
      wurzel = wurzel->next;
      free (current);
   }
}
```

4.2. Einfach verkettete Bruch-Liste

```
#include <stdio.h>
#include <stdlib.h>

class c_bruch
{
   private:
   int zaehler;
   int nenner;
   int ggT (int zahl1, int zahl2);
   void kuerzen (void);

   public:
   c_bruch (void);
   c_bruch (int i_zaehler, int i_nenner);
   c_bruch (int i_wert);
   void init (int i_zaehler, int i_nenner);
   int get_zaehler (void);
   int get_nenner  (void);
   void print (void);
```

```
   void add   (const c_bruch &summand);
   void subtr (const c_bruch &subtrahend);
   void mult  (const c_bruch &faktor);
   void div   (const c_bruch &divisor);
};

int c_bruch::ggT (int zahl1, int zahl2)
{
   if (zahl2 == 0)
      return zahl1;
   else
      return ggT (zahl2, zahl1 % zahl2);
}

void c_bruch::kuerzen (void)
{
   int max_teiler = ggT (zaehler, nenner);

   zaehler = zaehler / max_teiler;
   nenner  = nenner  / max_teiler;
}

c_bruch::c_bruch (void)
{
   init (0, 1);
}

c_bruch::c_bruch (int i_zaehler, int i_nenner)
{
   init (i_zaehler, i_nenner);
}

c_bruch::c_bruch (int wert)
{
   init (wert, 1);
}

void c_bruch::init (int i_zaehler, int i_nenner)
{
   zaehler = i_zaehler;
   nenner  = i_nenner;
}

int c_bruch::get_zaehler (void)
{
   return zaehler;
}

int c_bruch::get_nenner  (void)
{
   return nenner;
}

void c_bruch::print (void)
{
   printf ("%d/%d\n", zaehler, nenner);
}

void c_bruch::add (const c_bruch &summand)
{
   zaehler = zaehler * summand.nenner + nenner * summand.zaehler;
   nenner  = nenner  * summand.nenner;
   kuerzen();
}

void c_bruch::subtr (const c_bruch &subtrahend)
{
   zaehler = zaehler * subtrahend.nenner - nenner * subtrahend.zaehler;
   nenner  = nenner  * subtrahend.nenner;
   kuerzen();
}

void c_bruch::mult (const c_bruch &faktor)
{
   zaehler = zaehler * faktor.zaehler;
   nenner  = nenner  * faktor.nenner;
   kuerzen();
}

void c_bruch::div (const c_bruch &divisor)
```

```
{
   zaehler = zaehler * divisor.nenner;
   nenner  = nenner  * divisor.zaehler;
   kuerzen();
}

struct t_liste
{
   c_bruch  wert;
   t_liste *next;
};

void main (void)
{
   t_liste *wurzel  = NULL;
   t_liste *last    = NULL;
   t_liste *current = NULL;
   int      zaehler, nenner;
   c_bruch  summe = 0;

   printf ("Bitte Zaehler und Nenner der Summanden eingeben (0=Ende)");

   do
   {
      scanf ("%d%d", &zaehler, &nenner);
      if (zaehler != 0)
      {
         if (last == NULL)
         {
            wurzel = (t_liste *) malloc (sizeof (t_liste));
            last = wurzel;
         }
         else
         {
            last->next = (t_liste *) malloc (sizeof (t_liste));
            last = last->next;
         }
         last->wert.init (zaehler, nenner);
         last->next = NULL;
      }
   }
   while (zaehler != 0);

   current = wurzel;
   while (current != NULL)
   {
      summe.add (current->wert);
      current = current->next;
   }

   printf ("Summe = ");
   summe.print ();

   while (wurzel != NULL)
   {
      current = wurzel;
      wurzel = wurzel->next;
      free (current);
   }
}
```

4.3. Stack als einfach verkettete Liste

```
#include <stdio.h>
#include <stdlib.h>

struct t_daten
{
   int      wert;
   t_daten *next;
};

class c_stack
{
   private:
   t_daten *wurzel;

   public:
   c_stack ();
   ~c_stack ();
   void push (int i_wert);
```

```
   int pop (void);
};

c_stack::c_stack ()
{
   wurzel = NULL;
}

c_stack::~c_stack ()
{
   t_daten *current;

   while (wurzel != NULL)
   {
      current = wurzel;
      wurzel = wurzel->next;
      free (current);
   }
}

void c_stack::push (int i_wert)
{
   t_daten *neu = (t_daten *) malloc (sizeof (t_daten));

   if (neu != NULL)
   {
      neu->wert = i_wert;
      neu->next = wurzel;
      wurzel = neu;
   }
}

int c_stack::pop (void)
{
   int      result  = -1;
   t_daten *current = wurzel;

   if (wurzel != NULL)
   {
      result = wurzel->wert;
      wurzel = wurzel->next;
      free (current);
   }

   return result;
}

void main (void)
{
   c_stack stack;

   stack.push (3);
   stack.push (4);
   stack.push (stack.pop () + stack.pop ());
   printf ("3+4=%d\n", stack.pop ());
}
```

4.4. Queue als einfach verkettete Liste

```
#include <stdio.h>
#include <stdlib.h>

struct t_daten
{
   int      wert;
   t_daten *next;
};

class c_queue
{
   private:
   t_daten *wurzel;
   t_daten *last;

   public:
   c_queue ();
   ~c_queue ();
   void push (int i_wert);
   int pop (void);
};
```

```
c_queue::c_queue ()
{
   wurzel = NULL;
   last   = NULL;
}

c_queue::~c_queue ()
{
   t_daten *current;

   while (wurzel != NULL)
   {
      current = wurzel;
      wurzel = wurzel->next;
      free (current);
   }
}

void c_queue::push (int i_wert)
{
   if (last == NULL)
   {
      wurzel = (t_daten *) malloc (sizeof (t_daten));
      last = wurzel;
   }
   else
   {
      last->next = (t_daten *) malloc (sizeof (t_daten));
      last = last->next;
   }

   last->wert = i_wert;
   last->next = NULL;
}

int c_queue::pop (void)
{
   int      result  = -1;
   t_daten *current = wurzel;

   if (wurzel != NULL)
   {
      result = wurzel->wert;
      wurzel = wurzel->next;
      free (current);
      if (wurzel == NULL)
         last = NULL;
   }

   return result;
}

void main (void)
{
   c_queue queue;

   queue.push (3);
   queue.push (4);
   queue.push (6);
   queue.push (11);
   printf ("%d ", queue.pop ());
   printf ("%d ", queue.pop ());
   queue.push (14);
   printf ("%d ", queue.pop ());
   printf ("%d ", queue.pop ());
   printf ("%d ", queue.pop ());
}
```

4.5. Einfach verkettete Liste beliebiger Elemente

```
#include <stdio.h>
#include <stdlib.h>

struct t_daten
{
   void    *wert;
   t_daten *next;
};

class c_liste
{
```

```
   private:
   t_daten *wurzel;
   t_daten *last;
   t_daten *get_current (int index);

   public:
   c_liste (void);
   ~c_liste ();
   void append (void *i_wert);
   int   get_as_int   (int index);
   char  get_as_char  (int index);
   float get_as_float (int index);
};

t_daten *c_liste::get_current (int index)
{
   t_daten *current=wurzel;

   while (index > 0)
   {
      if (current != NULL)
         current = current->next;
      index --;
   }

   return current;
}

c_liste::c_liste (void)
{
   wurzel = NULL;
   last   = NULL;
}

c_liste::~c_liste ()
{
   t_daten *current;

   while (wurzel != NULL)
   {
      current = wurzel;
      wurzel = wurzel->next;
      if (current->wert != NULL)
         free (current->wert);
      free (current);
   }
}

void c_liste::append (void *i_wert)
{
   if (last == NULL)
   {
      wurzel = (t_daten *) malloc (sizeof (t_daten));
      last = wurzel;
   }
   else
   {
      last->next = (t_daten *) malloc (sizeof (t_daten));
      last = last->next;
   }

   last->wert = i_wert;
   last->next = NULL;
}

int c_liste::get_as_int (int index)
{
   t_daten *current=get_current (index);

   if (current != NULL)
      return *((int *) current->wert);
   else
      return -1;
}

char c_liste::get_as_char (int index)
{
   t_daten *current=get_current (index);

   if (current != NULL)
```

```
      return *((char *) current->wert);
   else
      return 0;
}

float c_liste::get_as_float (int index)
{
   t_daten *current=get_current (index);

   if (current != NULL)
      return *((float *) current->wert);
   else
      return -1.0;
}

void main (void)
{
   c_liste liste;
   int   *zahl;
   char  *zeichen;
   float *real;

   zahl = (int *) malloc (sizeof (int));
   *zahl = 42;

   zeichen = (char *) malloc (sizeof (char));
   *zeichen = 'A';

   real = (float *) malloc (sizeof (float));
   *real = 16.23;

   liste.append (zahl);
   liste.append (zeichen);
   liste.append (real);

   printf ("1. Eintrag: %d\n", liste.get_as_int (0));
   printf ("2. Eintrag: %c\n", liste.get_as_char (1));
   printf ("3. Eintrag: %f\n", liste.get_as_float (2));
}
```

4.6. Einfach verkettete Liste von Instanzen

```
#include <stdio.h>
#include <stdlib.h>

class c_element
{
   friend class c_liste;

   private:
   int        daten;
   c_element *next;

   c_element (int i_daten, c_element *i_next);
   ~c_element ();
   int get_daten (void);
   c_element *get_next (void);
   void set_next (c_element *new_next);
   void print (void);
};

class c_liste
{
   private:
   c_element *anfang;

   public:
   c_liste (void);
   ~c_liste ();
   void append (int neu);
   void print (void);
};

c_element::c_element (int i_daten, c_element *i_next)
{
   daten = i_daten;
   next  = i_next;
}

c_element::~c_element ()
{
```

```
}

int c_element::get_daten(void)
{
   return daten;
}

c_element *c_element::get_next(void)
{
   return next;
}

void c_element::set_next (c_element *new_next)
{
   next = new_next;
}

void c_element::print (void)
{
   printf ("%d ", daten);
}

c_liste::c_liste (void)
{
   anfang = new (c_element) (-1, NULL);
}

c_liste::~c_liste()
{
   c_element *element;

   while (anfang != NULL)
   {
      element = anfang;
      anfang = anfang->get_next();
      delete (element);
   }
}

void c_liste::append(int neu)
{
   c_element *element = anfang;
   c_element *new_elm;

   while (element->get_next() != NULL)
      element = element->get_next();

   new_elm = new (c_element) (neu, NULL);
   element->set_next (new_elm);
}

void c_liste::print (void)
{
   c_element *element;

   if (anfang == NULL)
   {
      printf ("Fehler beim Anlegen der Liste!!!\n");
   }
   else
   {
      element = anfang->get_next();
      printf ("Inhalt der Liste:\n");
      while (element != NULL)
      {
         element->print();
         element = element->get_next();
      }
      printf ("\n");
   }
}

void main (void)
{
   c_liste liste;

   liste.append (3);
   liste.append (5);
   liste.append (2);
   liste.append (4);
```

```
   liste.print ();
}
```

5. Sortierverfahren

5.1. Insertion Sort

```
#include <stdio.h>

void insertion_sort (int *liste, int anzahl)
{
   int next_pos;
   int next_element;
   int ins_pos;

   // Nimm das nächste Element, das in den sortierten Bereich
   // eingefügt werden soll
   for (next_pos=1; next_pos<anzahl; next_pos++)
   {
      next_element = liste [next_pos];
      // Suche die richtige Stelle an der das Element einzufügen ist.
      // Dazu werden alle größeren Elemente um eine Stelle nach hinten
      // verschoben.
      ins_pos = next_pos;
      while ((ins_pos > 0) && (liste[ins_pos -1] > next_element))
      {
         liste [ins_pos] = liste [ins_pos -1];
         ins_pos--;
      };
      // Füge das Element an der richtigen Stelle ein.
      liste [ins_pos] = next_element;
   }
}

void ausgabe_liste (int *liste, int anzahl)
{
   int i;

   for (i=0; i<anzahl; i++)
      printf ("%d ", liste [i]);
   printf ("\n");
}

void main (void)
{
   int liste [10] = {12,3,9,11,18,25,13,6,42,37};

   ausgabe_liste (liste, 10);
   insertion_sort (liste, 10);
   ausgabe_liste (liste, 10);
}
```

5.2. Selection Sort

```
#include <stdio.h>

void tauschen (int &zahl1, int &zahl2)
// vertauscht die beiden angegebenen Zahlen. Durch die
// Verwendung der Referenzparameter wird das Ergebnis
// des Tauschs zurückgeliefert
{
   int temp = zahl1;

   zahl1 = zahl2;
   zahl2 = temp;
}

void selection_sort (int *liste, int anzahl)
{
   int ins_pos;
   int such_pos;
   int min_pos;

   // Suche das nächste Element, das in den sortierten Bereich
   // eingefügt werden muß
   for (ins_pos=0; ins_pos < anzahl-1; ins_pos++)
   {
      min_pos = ins_pos;
      // Suche dazu das kleinste Element im verbliebenen Rest
      for (such_pos = ins_pos +1; such_pos < anzahl; such_pos++)
         if (liste [such_pos] < liste [min_pos])
            min_pos = such_pos;
      // Vertausche das gefundene Minimum mit dem Element, das
      // sich am Anfang des unsortierten Teils befindet.
      tauschen (liste [ins_pos], liste [min_pos]);
```

```
   }
}

void ausgabe_liste (int *liste, int anzahl)
{
   int i;

   for (i=0; i<anzahl; i++)
      printf ("%d ", liste [i]);
   printf ("\n");
}

void main (void)
{
   int liste [10] = {12,3,9,11,18,25,13,6,42,37};

   ausgabe_liste (liste, 10);
   selection_sort (liste, 10);
   ausgabe_liste (liste, 10);
}
```

5.3. Bubble Sort

```
#include <stdio.h>

void tauschen (int &zahl1, int &zahl2)
// vertauscht die beiden angegebenen Zahlen. Durch die
// Verwendung der Referenzparameter wird das Ergebnis
// des Tauschs zurückgeliefert
{
   int temp = zahl1;

   zahl1 = zahl2;
   zahl2 = temp;
}

void bubble_sort (int *liste, int anzahl)
{
   int bubble_pos;
   int tausch;

   do
   {
      tausch = 0;
      // Prüfe, ob in der Liste Element-Paare vorhanden sind, bei denen
      // die Sortierung nicht stimmt
      for (bubble_pos = 0; bubble_pos < anzahl -1; bubble_pos++)
      {
         // Ein solches Paar wird vertauscht und der Tausch wird vermerkt
         if (liste [bubble_pos] > liste [bubble_pos +1])
         {
            tauschen (liste [bubble_pos], liste [bubble_pos +1]);
            tausch++;
         }
      }
   }
   // Solange Element-Paare vertauscht wurden, muß die Liste nochmals
   // geprüft werden
   while (tausch > 0);
}

void ausgabe_liste (int *liste, int anzahl)
{
   int i;

   for (i=0; i<anzahl; i++)
      printf ("%d ", liste [i]);
   printf ("\n");
}

void main (void)
{
   int liste [10] = {12,3,9,11,18,25,13,6,42,37};

   ausgabe_liste (liste, 10);
   bubble_sort (liste, 10);
   ausgabe_liste (liste, 10);
}
```

5.4. Quicksort

```
#include <stdio.h>
```

```
void tauschen (int &zahl1, int &zahl2)
// vertauscht die beiden angegebenen Zahlen. Durch die
// Verwendung der Referenzparameter wird das Ergebnis
// des Tauschs zurückgeliefert
{
   int temp = zahl1;

   zahl1 = zahl2;
   zahl2 = temp;
}

void quick_sort (int *liste, int von, int bis)
{
   int teilelement;
   int min_pos;
   int max_pos;

   if (von < bis)
   {
      // Sortiere solange Elemente um, bis links vom Teilungselement
      // keine größeren und rechts davon keine kleineren Elemente stehen
      teilelement = liste [bis];
      max_pos = von;
      min_pos = bis -1;
      do
      {
         // Suche dazu von links das erste Element, das kleiner als
         // das Teilungselement ...
         while (liste [max_pos] < teilelement)
            max_pos++;
         // ... und von rechts das erste Element, das größer ist.
         while ((min_pos >= von) && (liste [min_pos] > teilelement))
            min_pos--;
         // Wenn diese nicht in der richtigen Reihenfolge stehen,
         // werden sie vertauscht
         if (min_pos > max_pos)
            tauschen (liste [min_pos], liste [max_pos]);
      }
      // Dies wird solange durchgeführt, bis die grobe Vorsortierung
      // abgeschlossen ist
      while (min_pos > max_pos);
      tauschen (liste [max_pos], liste [bis]);
      // Sortiere danach die linke und rechte "Hälfte" per Rekursion
      quick_sort (liste, von, max_pos -1);
      quick_sort (liste, max_pos +1, bis);
   }
}

void ausgabe_liste (int *liste, int anzahl)
{
   int i;

   for (i=0; i<anzahl; i++)
      printf ("%d ", liste [i]);
   printf ("\n");
}

void main (void)
{
   int liste [10] = {12,3,9,11,18,25,13,6,42,37};

   ausgabe_liste (liste, 10);
   quick_sort (liste, 0, 9);
   ausgabe_liste (liste, 10);
}
```

5.5. Heapsort

```
#include <stdio.h>

void ausgabe_liste (int *liste, int anzahl)
{
   int i;

   for (i=0; i<anzahl; i++)
      printf ("%d ", liste [i]);
   printf ("\n");
}

void down_heap (int *heap, int anzahl, int pos_vater)
// Überprüft, ob das Element an der Stelle "pos_vater" wirklich
```

```
// größer ist als seine beiden Söhne, wenn nicht, wird dieses
// Element an die richtige Stelle im Heap verschoben.
{
   int wurzel = heap[pos_vater];
   int max_sohn;

   // Prüfe, ob der Knoten überhaupt Söhne besitzt
   if (2 * pos_vater +1 < anzahl)
   {
      do
      {
         // Ermittle den größeren der Söhne
         max_sohn = 2 * pos_vater +1;
         if ((max_sohn < anzahl -1) &&
            (heap [max_sohn +1] > heap [max_sohn]))
            max_sohn++;
         // Wenn das Element kleiner als der größere Sohn ist,
         // dann muß das Element im Heap nach unten verschoben
         // werden
         if (wurzel < heap [max_sohn])
         {
            heap [pos_vater] = heap [max_sohn];
            pos_vater = max_sohn;
         }
      }
      // Dies erfolgt solange, bis das Element am unteren Ende des
      // Baums angekommen ist, oder bis es größer als beide Söhne ist
      while ((2*pos_vater +1 < anzahl) && (wurzel < heap [max_sohn]));
      heap [pos_vater] = wurzel;
   }
}

int remove_heap (int *heap, int &anzahl)
// Entfernt das größte Element (=Wurzel) aus dem Heap
{
   int ret = heap[0];

   // Wurzel durch letztes Element ersetzen
   heap[0] = heap[anzahl -1];
   anzahl--;
   // Dieses Element auf korrekte Position überpüfen
   down_heap (heap, anzahl, 0);
   return ret;
}

void heap_sort (int *liste, int anzahl)
{
   int heap_anzahl=anzahl;
   int heap_wurzel;

   // Aus der unsortierten Liste schrittweise einen Heap aufbauen
   for (heap_wurzel = anzahl/2 -1; heap_wurzel >=0; heap_wurzel--)
      down_heap (liste, heap_anzahl, heap_wurzel);
   ausgabe_liste (liste, anzahl);
   // Aus dem Heap schrittweise eine sortierte Liste aufbauen
   for (heap_wurzel = anzahl -1; heap_wurzel >=0; heap_wurzel--)
      liste [heap_wurzel] = remove_heap (liste, heap_anzahl);
}

void main (void)
{
   int liste [10] = {12,3,9,11,18,25,13,6,42,37};

   ausgabe_liste (liste, 10);
   heap_sort (liste, 10);
   ausgabe_liste (liste, 10);
}
```

5.6. Abgeleitete Klassen

```
#include <stdio.h>
#include <stdlib.h>

// Gemeinsame Basisklasse
class t_liste
{
   protected:
   int element [40];
   int anzahl;
   void vertauschen (int i, int j);
```

```
   public:
   t_liste (void);
   ~t_liste ();
   int einfuegen (int daten);
   virtual void sort (void)=0;
   void ausgabe (void);
};

// Liste mit Bubble-Sort
class t_bubble_liste : public t_liste
{
   public:
   virtual void sort (void);
};

// Liste mit Selection-Sort
class t_selection_liste : public t_liste
{
   public:
   virtual void sort (void);
};

// Liste mit Insertion-Sort
class t_insertion_liste : public t_liste
{
   public:
   virtual void sort (void);
};

// Liste mit Quick-Sort
class t_quick_liste : public t_liste
{
   private:
   void quicksort (int links, int rechts);

   public:
   virtual void sort (void);
};

// Liste mit Heap-Sort
class t_heap_liste : public t_liste
{
   private:
   int heap_anzahl;
   void downheap (int pos_vater);
   int removeheap (void);

   public:
   t_heap_liste (void);
   virtual void sort (void);
};

t_liste::t_liste (void) : anzahl (0)
{
}

t_liste::~t_liste ()
{
}

void t_liste::vertauschen (int i, int j)
{
   // Dreiertausch
   int temp = element [i];
   element [i] = element [j];
   element [j] = temp;
}

int t_liste::einfuegen (int daten)
{
   // Wenn möglich, das neue Element hinten anfügen
   if (anzahl < 40)
   {
      element [anzahl] = daten;
      anzahl ++;
   }
   return anzahl;
}

void t_liste::ausgabe (void)
```

```
{
   int i;

   // Alle Listenelemente ausgeben
   for (i=0; i<anzahl; i++)
      printf ("%d ", element [i]);
   printf ("\n");
}

void t_bubble_liste::sort (void)
{
   int bubble_pos;
   int getauscht;

   do
   {
      getauscht = 0;
      // Prüfe, ob in der Liste Element-Paare vorhanden sind, bei denen
      // die Sortierung nicht stimmt
      for (bubble_pos = 0; bubble_pos < anzahl -1; bubble_pos++)
      {
         // Ein solches Paar wird vertauscht und der Tausch wird vermerkt
         if (element [bubble_pos] > element [bubble_pos +1])
         {
            vertauschen (bubble_pos, bubble_pos +1);
            getauscht = 1;
         }
      }
   }
   // Solange Element-Paare vertauscht wurden, muß die Liste nochmals
   // geprüft werden
   while (getauscht != 0);
}

void t_selection_liste::sort (void)
{
   int min_pos;
   int ins_pos;
   int such_pos;

   // Suche das nächste Element, das in den sortierten Bereich
   // eingefügt werden muß
   for (ins_pos=0; ins_pos < anzahl-1; ins_pos++)
   {
      min_pos = ins_pos;
      // Suche dazu das kleinste Element im verbliebenen Rest
      for (such_pos = ins_pos +1; such_pos < anzahl; such_pos++)
         if (element [such_pos] < element [min_pos])
            min_pos = such_pos;
      // Vertausche das gefundene Minimum mit dem Element, das
      // sich am Anfang des unsortierten Teils befindet.
      vertauschen (ins_pos, min_pos);
   }
}

void t_insertion_liste::sort (void)
{
   int next_pos;
   int next_element;
   int ins_pos;

   // Nimm das nächste Element, das in den sortierten Bereich
   // eingefügt werden soll
   for (next_pos=1; next_pos<anzahl; next_pos++)
   {
      next_element = element [next_pos];
      // Suche die richtige Stelle an der das Element einzufügen ist.
      // Dazu werden alle größeren Elemente um eine Stelle nach hinten
      // verschoben.
      ins_pos = next_pos;
      while ((ins_pos > 0) && (element[ins_pos -1] > next_element))
      {
         element [ins_pos] = element [ins_pos -1];
         ins_pos--;
      };
      // Füge das Element an der richtigen Stelle ein.
      element [ins_pos] = next_element;
   }
}
```

```
void t_quick_liste::quicksort (int links, int rechts)
{
   int teilelement;
   int min_pos;
   int max_pos;

   if (links < rechts)
   {
      // Sortiere solange Elemente um, bis links vom Teilungselement
      // keine größeren und rechts davon keine kleineren Elemente stehen
      teilelement = element [rechts];
      max_pos = links;
      min_pos = rechts -1;
      do
      {
         // Suche dazu von links das erste Element, das kleiner als
         // das Teilungselement ...
         while (element [max_pos] < teilelement)
            max_pos++;
         // ... und von rechts das erste Element, das größer ist.
         while ((min_pos >= links) && (element [min_pos] > teilelement))
            min_pos--;
         // Wenn diese nicht in der richtigen Reihenfolge stehen,
         // werden sie vertauscht
         if (min_pos > max_pos)
            vertauschen (min_pos, max_pos);
      }
      // Dies wird solange durchgeführt, bis die grobe Vorsortierung
      // abgeschlossen ist
      while (min_pos > max_pos);
      vertauschen (max_pos, rechts);
      // Sortiere danach die linke und rechte "Hälfte" per Rekursion
      quicksort (links, max_pos -1);
      quicksort (max_pos +1, rechts);
   }
}

void t_quick_liste::sort (void)
{
   quicksort (0, anzahl -1);
}

t_heap_liste::t_heap_liste (void) : t_liste(), heap_anzahl (0)
{
}

void t_heap_liste::downheap (int pos_vater)
// Überprüft, ob das Element an der Stelle "pos_vater" wirklich
// das größer ist als seine beiden Söhne, wenn nicht, wird dieses
// Element an die richtige Stelle im Heap verschoben.
{
   int wurzel = element[pos_vater];
   int max_sohn;

   // Prüfe, ob der Knoten überhaupt Söhne besitzt
   if (2 * pos_vater +1 < heap_anzahl)
   {
      do
      {
         // Ermittle den größeren der Söhne
         max_sohn = 2 * pos_vater +1;
         if ((max_sohn < heap_anzahl -1) &&
            (element [max_sohn +1] > element [max_sohn]))
            max_sohn++;
         // Wenn das Element kleiner als der größere Sohn ist,
         // dann muß das Element im Heap nach unten verschoben
         // werden
         if (wurzel < element [max_sohn])
         {
            element [pos_vater] = element [max_sohn];
            pos_vater = max_sohn;
         }
      }
      // Dies erfolgt solange, bis das Element am unteren Ende des
      // Baums angekommen ist, oder bis es größer als beide Söhne ist
      while ((2*pos_vater +1 < heap_anzahl) && (wurzel < element [max_sohn]));
      element [pos_vater] = wurzel;
   }
}
```

```
int t_heap_liste::removeheap (void)
// Entfernt das größte Element (=Wurzel) aus dem Heap
{
   int ret = element[0];

   // Wurzel durch letztes Element ersetzen
   element[0] = element[heap_anzahl -1];
   heap_anzahl--;
   // Dieses Element auf korrekte Position überpüfen
   downheap (0);
   return ret;
}

void t_heap_liste::sort (void)
{
   int heap_wurzel;

   heap_anzahl=anzahl;
   // Aus der unsortierten Liste schrittweise einen Heap aufbauen
   for (heap_wurzel = anzahl/2 -1; heap_wurzel >=0; heap_wurzel--)
      downheap (heap_wurzel);
   // Aus dem Heap schrittweise eine sortierte Liste aufbauen
   for (heap_wurzel = anzahl -1; heap_wurzel >=0; heap_wurzel--)
      element [heap_wurzel] = removeheap ();
}

void main (void)
{
   t_liste *liste;
   int wahl, i;

   printf ("Mit welchem Verfahren möchten Sie die Liste sortieren?\n");
   printf ("(1) Bubble-Sort\n");
   printf ("(2) Selection-Sort\n");
   printf ("(3) Insertion-Sort\n");
   printf ("(4) Quick-Sort\n");
   printf ("(5) Heap-Sort\n");
   printf ("Ihre Wahl:");
   scanf ("%d", &wahl);
   switch (wahl)
   {
      case 1: liste = new (t_bubble_liste); break;
      case 2: liste = new (t_selection_liste); break;
      case 3: liste = new (t_insertion_liste); break;
      case 4: liste = new (t_quick_liste); break;
      case 5: liste = new (t_heap_liste); break;
      default: liste = NULL;
   }
   if (liste != NULL)
   {
      for (i=0; i<20; i++)
         liste->einfuegen (i*i -13*i +5);
      liste->ausgabe();
      liste->sort();
      liste->ausgabe();
      delete (liste);
   }
}
```

6. Bäume

6.1. Binärbaum ohne Duplikate

```
#include <stdio.h>
#include <stdlib.h>

struct t_baum
{
   int daten;
   t_baum *links;
   t_baum *rechts;
};

struct t_queue
{
   t_baum **daten;
   int max;
   int anzahl;
   int out_pos;
};

//
// Algorithmen für Queue (wird für Level-Order benötigt)
//

void init_queue (t_queue &queue, int max_anzahl)
// Queue erzeugen. Die Queue kann maximal die übergebene Anzahl
// von Elementen aufnehmen
{
   // Queue initialisieren
   queue.daten = (t_baum **) malloc (max_anzahl * sizeof (t_baum *));
   if (queue.daten != NULL)
      queue.max = max_anzahl;
   else
      queue.max = 0;
   queue.anzahl = 0;
   queue.out_pos = 0;
}

void push_queue (t_queue &queue, t_baum *knoten)
// Ein neues Element -wenn möglich- in die Queue stellen
{
   if (queue.anzahl < queue.max)
   {
      queue.daten[(queue.out_pos + queue.anzahl) % queue.max] = knoten;
      queue.anzahl++;
   }
}

t_baum *pop_queue (t_queue &queue)
// Ein Element -wenn vorhanden- aus der Queue entnehmen
// Falls kein Element mehr vorhanden ist, wird NULL geliefert
{
   t_baum *daten = NULL;

   if (queue.anzahl > 0)
   {
      daten = queue.daten [queue.out_pos];
      queue.out_pos = (queue.out_pos +1) % queue.max;
      queue.anzahl--;
   }
   return daten;
}

void delete_queue (t_queue &queue)
// Gibt den von der Queue reservierten Speicher frei
{
   if (queue.daten != NULL)
   {
      free (queue.daten);
      queue.anzahl = 0;
   }
}

//
// Algorithmen für Einfügen und Entfernen einzelner Elemente im Baum
//

t_baum *einfuegen (t_baum *wurzel, int neudaten)
```

```
// Fügt in den Baum einen neuen Knoten ein, der die angegebenen
// Daten enthält.
{
   // Wenn der Teilbaum nicht existiert, wird er angelegt und
   // dort werden die neuen Daten eingetragen
   if (wurzel == NULL)
   {
      wurzel = (t_baum *) malloc (sizeof (t_baum));
      if (wurzel != NULL)
      {
         wurzel->daten  = neudaten;
         wurzel->links  = NULL;
         wurzel->rechts = NULL;
      }
   }
   // Andernfalls wird zum linken oder rechten Teilbaum
   // weitergegangen
   else
   {
      if (neudaten < wurzel->daten)
         wurzel->links = einfuegen (wurzel->links, neudaten);
      else
         wurzel->rechts = einfuegen (wurzel->rechts, neudaten);
   }
   // Als Ergebnis wird die Wurzel des Teilbaums zurückgeliefert
   return wurzel;
}

t_baum *maximum (t_baum *wurzel)
// Liefert einen Zeiger auf das größte Element des Baums
{
   // Gehe zum größten Element
   if (wurzel != NULL)
   {
      while (wurzel->rechts != NULL)
         wurzel = wurzel->rechts;
   }
   // Gib dessen Adresse zurück
   return wurzel;
}

t_baum *entfernen (t_baum *wurzel, int deldaten)
// Entfernt die angegebenen Daten aus dem Baum
{
   t_baum *maxknoten;
   t_baum *delknoten = wurzel;

   // Wenn die Daten im Baum noch nicht gefunden wurde, ...
   if ((wurzel != NULL) && (wurzel->daten != deldaten))
   {
      // ... dann suche im linken, bzw. rechten Teilbaum danach
      if (deldaten < wurzel->daten)
         wurzel->links = entfernen (wurzel->links, deldaten);
      else
         wurzel->rechts = entfernen (wurzel->rechts, deldaten);
   }
   // Andernfalls entferne den gefundenen Knoten
   else if (wurzel != NULL)
   {
      // Prüfe, ob ein einfacher Spezialfall vorliegt
      if (wurzel->links == NULL)
      {
         wurzel = wurzel->rechts;
         free (delknoten);
      }
      else if (wurzel->rechts == NULL)
      {
         wurzel = wurzel->links;
         free (delknoten);
      }
      // Andernfalls ermittle das größte Element, das kleiner als
      // das zu löschende Element ist und übertrage dessen Daten
      // in das zu löschende Element. Danach lösche stattdessen
      // dieses Element
      else
      {
         maxknoten = maximum (wurzel->links);
         wurzel->daten = maxknoten->daten;
         wurzel->links = entfernen (wurzel->links, wurzel->daten);
      }
```

```
   }
   // Als Ergebnis wird die Wurzel des Teilbaums zurückgegeben
   return wurzel;
}

//
// Algorithmen, mit denen der Baum durchlaufen wird
//

void ausgabe (t_baum *wurzel, int tiefe)
{
   // Durchlaufe den Baum und gib alle Knoten entsprechend ihrer Tiefe aus
   // Der Baum wird um 90° nach links gedreht ausgegeben
   if (wurzel != NULL)
   {
      ausgabe (wurzel->rechts, tiefe +1);
      printf ("%*c%d\n", tiefe *4, ' ', wurzel->daten);
      ausgabe (wurzel->links,  tiefe +1);
   }
}

int anzahl (t_baum *wurzel)
// Liefert die Anzahl der Elemente im Baum
{
   if (wurzel != NULL)
      return 1 + anzahl (wurzel->links) + anzahl (wurzel->rechts);
   else
      return 0;
}

void preorder (t_baum *wurzel, int *&liste)
// Durchläuft den Baum in PreOrder und speichert alle Elemente
// in einem Feld.
{
   if (wurzel != NULL)
   {
      // Durchlaufe Baum in PreOrder
      *liste = wurzel->daten;
      liste++;
      preorder (wurzel->links,  liste);
      preorder (wurzel->rechts, liste);
   }
}

void inorder (t_baum *wurzel, int *&liste)
// Durchläuft den Baum in InOrder und speichert alle Elemente
// in einem Feld.
{
   if (wurzel != NULL)
   {
      // Durchlaufe Baum in InOrder
      inorder (wurzel->links, liste);
      *liste = wurzel->daten;
      liste++;
      inorder (wurzel->rechts, liste);
   }
}

void postorder (t_baum *wurzel, int *&liste)
// Durchläuft den Baum in PostOrder und speichert alle Elemente
// in einem Feld.
{
   if (wurzel != NULL)
   {
      // Durchlaufe Baum in PostOrder
      postorder (wurzel->links,  liste);
      postorder (wurzel->rechts, liste);
      *liste = wurzel->daten;
      liste++;
   }
}

void levelorder (t_baum *wurzel, int *liste)
// Durchläuft den Baum in LevelOrder und speichert alle Elemente
// in einem Feld. Zum Durchlaufen wird eine Queue verwendet, die
// die einzelnen Knoten des Baums speichert
{
   t_baum *knoten;
   t_queue queue;
   int anz = anzahl (wurzel);
```

```
   if (wurzel != NULL)
   {
      init_queue (queue, anz);
      // Durchlaufe Baum in LevelOrder
      push_queue (queue, wurzel);
      while ((knoten = pop_queue (queue)) != NULL)
      {
         // Speichere die Daten eines Knoten ...
         *liste = knoten->daten;
         liste++;
         // ... und stelle -wenn vorhanden- dessen Söhne in die Queue
         if (knoten->links != NULL)
            push_queue (queue, knoten->links);
         if (knoten->rechts != NULL)
            push_queue (queue, knoten->rechts);
      }
   }
}

void invers (t_baum *wurzel, int *&liste)
// Durchläuft den Baum in absteigender Reihenfolge und speichert
// alle Elemente in einem Feld
{
   if (wurzel != NULL)
   {
      // Durchlaufe Baum in absteigender Reihenfolge
      invers (wurzel->rechts, liste);
      *liste = wurzel->daten;
      liste++;
      invers (wurzel->links,  liste);
   }
}

int *durchlaufen (t_baum *wurzel, int reihenfolge)
// Durchläuft den Baum und speichert alle Elemente in einem Feld.
// Die Adresse dieser Liste wird als Funktionswert zurückgegeben.
// Die Reihenfolge für das Durchlaufen wird über den Parameter
// "reihenfolge" gesteuert.
{
   int anz = anzahl (wurzel);
   int *liste = (int *) malloc (anz * sizeof (int));
   int *anfang = liste;

   // Wenn ein Feld mit Platz für die Elemente des Baums
   // angelegt werden konnte, dann durchlaufe den Baum
   if (liste != NULL)
   {
      switch (reihenfolge)
      {
         case 1: preorder   (wurzel, liste); break;
         case 2: inorder    (wurzel, liste); break;
         case 3: postorder  (wurzel, liste); break;
         case 4: levelorder (wurzel, liste); break;
         case 5: invers     (wurzel, liste); break;
      }
   }
   return anfang;
}

t_baum *umsortieren (t_baum *wurzel, int reihenfolge)
// Erstellt aus dem übergebenen Baum einen neuen Baum.
// Die Daten werden dabei in der angegebenen Reihenfolge
// eingefügt
{
   t_baum *neubaum = NULL;
   int *liste = durchlaufen (wurzel, reihenfolge);
   int anz = anzahl (wurzel);
   int i;

   if (liste != NULL)
   {
      for (i=0; i<anz; i++)
         neubaum = einfuegen (neubaum, liste[i]);
      free (liste);
   }
   return neubaum;
}

//
```

```
// Baum vollständig freigeben
//

void loeschen (t_baum *wurzel)
// Gibt den gesamten Speicherplatt frei, der vom Baum belegt wird
{
   if (wurzel != NULL)
   {
      // Entferne linken und rechten Teilbaum aus dem Speicher
      loeschen (wurzel->links);
      loeschen (wurzel->rechts);
      // Anschließend gib den von der Wurzel belegten Speicher frei
      free (wurzel);
   }
}

void main (void)
{
   t_baum *wurzel = NULL;
   t_baum *duplikat = NULL;

   // Baum erstellen
   wurzel = einfuegen (wurzel,  3);
   wurzel = einfuegen (wurzel,  8);
   wurzel = einfuegen (wurzel,  1);
   wurzel = einfuegen (wurzel, 12);
   wurzel = einfuegen (wurzel, 16);
   wurzel = einfuegen (wurzel,  7);
   ausgabe (wurzel, 0);
   // neuen Baum (PreOrder) erstellen
   duplikat = umsortieren (wurzel, 1);
   ausgabe (duplikat, 0);
   loeschen (duplikat);
   // neuen Baum (InOrder) erstellen
   duplikat = umsortieren (wurzel, 2);
   ausgabe (duplikat, 0);
   loeschen (duplikat);
   // neuen Baum (PostOrder) erstellen
   duplikat = umsortieren (wurzel, 3);
   ausgabe (duplikat, 0);
   loeschen (duplikat);
   // neuen Baum (LevelOrder) erstellen
   duplikat = umsortieren (wurzel, 4);
   ausgabe (duplikat, 0);
   loeschen (duplikat);
   // neuen Baum (Invers) erstellen
   duplikat = umsortieren (wurzel, 5);
   ausgabe (duplikat, 0);
   loeschen (duplikat);
   // Einzelne Elemente entfernen
   wurzel = entfernen (wurzel, 8);
   ausgabe (wurzel, 0);
   wurzel = entfernen (wurzel, 3);
   ausgabe (wurzel, 0);
   // Baum loeschen
   loeschen (wurzel);
}
```

6.2. Binärbaum mit Duplikaten

```
#include <stdio.h>
#include <stdlib.h>

struct t_liste
{
   int daten;
   t_liste *next;
};

struct t_baum
{
   t_liste *anfang;
   t_baum *links;
   t_baum *rechts;
};

void anfuegen_liste (t_liste *&anfang, int neudaten)
// Fügt am Anfang einer einfach verketteten Liste ein neues
// Element an
{
   t_liste *neuelement = (t_liste *) malloc (sizeof (t_liste));
```

```
   if (neuelement != NULL)
   {
      // Hänge die bisherige Liste an das neue Element an
      neuelement->daten = neudaten;
      neuelement->next  = anfang;
      // und gib als Anfang der Liste das neue Element zurück
      anfang = neuelement;
   }
}

void loesche_liste (t_liste *anfang)
// Gibt den gesamten von der Liste belegten Speicher frei
{
   if (anfang != NULL)
   {
      loesche_liste (anfang->next);
      free (anfang);
   }
}

t_baum *einfuegen (t_baum *wurzel, int neudaten)
// Fügt in den Baum einen neuen Knoten ein, der die angegebenen
// Daten enthält.
{
   // Wenn der Teilbaum nicht existiert, wird er angelegt und
   // dort werden die neuen Daten eingetragen
   if (wurzel == NULL)
   {
      wurzel = (t_baum *) malloc (sizeof (t_baum));
      if (wurzel != NULL)
      {
         wurzel->anfang = (t_liste *) malloc (sizeof (t_liste));
         // Wenn eine neue Liste angelegt werden konnte, wird
         // dort das neue Element eingefügt
         if (wurzel->anfang != NULL)
         {
            wurzel->anfang->daten = neudaten;
            wurzel->anfang->next  = NULL;
            wurzel->links  = NULL;
            wurzel->rechts = NULL;
         }
         else
         // Andernfalls wird der Knoten wieder freigegeben. Damit
         // wird sichergestellt, daß kein Knoten im Baum eine leere
         // Liste enthält
         {
            free (wurzel);
            wurzel = NULL;
         }
      }
   }
   // Andernfalls wird zum linken oder rechten Teilbaum
   // weitergegangen
   else
   {
      if (neudaten < wurzel->anfang->daten)
         wurzel->links = einfuegen (wurzel->links, neudaten);
      else if (neudaten > wurzel->anfang->daten)
         wurzel->rechts = einfuegen (wurzel->rechts, neudaten);
      else
         anfuegen_liste (wurzel->anfang, neudaten);
   }
   // Als Ergebnis wird die Wurzel des Teilbaums zurückgeliefert
   return wurzel;
}

void ausgabe (t_baum *wurzel, int tiefe)
{
   // Durchlaufe den Baum und gib alle Knoten entsprechend ihrer Tiefe aus
   // Der Baum wird um 90° nach links gedreht ausgegeben
   if (wurzel != NULL)
   {
      ausgabe (wurzel->rechts, tiefe +1);
      printf ("%*c%d\n", tiefe *4, ' ', wurzel->anfang->daten);
      ausgabe (wurzel->links,  tiefe +1);
   }
}

void loeschen (t_baum *wurzel)
```

```
// Gibt den gesamten Speicherplatt frei, der vom Baum belegt wird
{
   if (wurzel != NULL)
   {
      // Entferne linken und rechten Teilbaum aus dem Speicher
      loeschen (wurzel->links);
      loeschen (wurzel->rechts);
      // Anschließend gib den von der Wurzel belegten Speicher frei
      loesche_liste (wurzel->anfang);
      free (wurzel);
   }
}

void main (void)
{
   t_baum *wurzel = NULL;

   wurzel = einfuegen (wurzel,  3);
   wurzel = einfuegen (wurzel,  8);
   wurzel = einfuegen (wurzel,  1);
   wurzel = einfuegen (wurzel, 12);
   wurzel = einfuegen (wurzel, 16);
   wurzel = einfuegen (wurzel,  7);
   ausgabe (wurzel, 0);
   loeschen (wurzel);
}
```

6.3. AVL-Baum

```
#include <stdio.h>
#include <stdlib.h>

struct t_baum
{
   int daten;
   int balance;
   t_baum *links;
   t_baum *rechts;
};

void rotieren (t_baum *&wurzel)
{
   int bal_sohn;
   t_baum *altewurzel = wurzel;

   // ermittle Balance-Faktor beim entsprechenden Sohn
   if (wurzel->balance == -2)
      bal_sohn = wurzel->rechts->balance;
   else
      bal_sohn = wurzel->links->balance;
   // Betrachte die vier verschiedenen Möglichkeiten der Rotation
   switch (wurzel->balance + bal_sohn)
   {
      // einfache Linksrotation
      case -3:
      case -2: wurzel = altewurzel->rechts;
             altewurzel->rechts = wurzel->links;
             wurzel->links = altewurzel;
             altewurzel->balance =  -wurzel->balance -1;
             wurzel->balance = wurzel->balance +1;
             break;
      // Doppelrotation rechts-links
      case -1: wurzel = altewurzel->rechts->links;
             altewurzel->rechts->links = wurzel->rechts;
             wurzel->rechts = altewurzel->rechts;
             altewurzel->rechts = wurzel->links;
             wurzel->links = altewurzel;
             wurzel->links->balance  = (wurzel->balance ==  1) ? -1 : 0;
             wurzel->rechts->balance = (wurzel->balance == -1) ?  1 : 0;
             wurzel->balance = 0;
             break;
      // Doppelrotation links-rechts
      case  1: wurzel = altewurzel->links->rechts;
             altewurzel->links->rechts = wurzel->links;
             wurzel->links = altewurzel->links;
             altewurzel->links = wurzel->rechts;
             wurzel->rechts = altewurzel;
             wurzel->links->balance  = (wurzel->balance == -1) ?  1 : 0;
             wurzel->rechts->balance = (wurzel->balance ==  1) ? -1 : 0;
             wurzel->balance = 0;
             break;
```

```
      // einfache Rechtsrotation
      case  2:
      case  3: wurzel = altewurzel->links;
             altewurzel->links = wurzel->rechts;
             wurzel->rechts = altewurzel;
             wurzel->balance = 0;
             altewurzel->balance = 0;
             break;
   }
}

int einfuegen (t_baum *&wurzel, int neudaten)
{
   int check_bal;

   if (wurzel==NULL)
   {
      // Erstelle einen neuen Knoten und trage dort die übergebenen
      // Daten ein. Dabei ändert sich die Höhe dieses Teilbaums
      wurzel = (t_baum *) malloc (sizeof (t_baum));
      if (wurzel != NULL)
      {
         wurzel->daten   = neudaten;
         wurzel->balance = 0;
         wurzel->links   = NULL;
         wurzel->rechts  = NULL;
         return 1;
      }
      else
         return 0;
   }
   else
   {
      // Suche im passenden Teilbaum nach der richtigen Einfügeposition
      // Der rekursive Aufurf liefert TRUE zurück, wenn sich beim Einfügen
      // die Höhe des Teilbaums geändert hat. Dann muß der Balance-Faktor
      // der Wurzel überprüft werden
      if (neudaten < wurzel->daten)
         check_bal = einfuegen (wurzel->links, neudaten);
      else
         check_bal = einfuegen (wurzel->rechts, neudaten);
      // Wenn der Balance-Faktor geprüft werden muß, ...
      if (check_bal)
      {
         // ... korrigiere den Balance-Faktor der Wurzel
         if (neudaten < wurzel->daten)
            wurzel->balance ++;
         else
            wurzel->balance --;
         // Falls der Balance-Faktor dabei die Regeln des AVL-Baums verletzt,
         // muß rotiert werden
         if ((wurzel->balance == -2) || (wurzel->balance == 2))
            rotieren (wurzel);
         // Wenn der Balance-Faktor der Wurzel am Schluß ungleich 0 ist, dann
         // hat sich die Höhe des Teilbaums geändert
         return (wurzel->balance != 0);
      }
      return 0;
   }
}

t_baum *maximum (t_baum *wurzel)
// Liefert einen Zeiger auf das größte Element des Baums
{
   // Gehe zum größten Element
   if (wurzel != NULL)
   {
      while (wurzel->rechts != NULL)
         wurzel = wurzel->rechts;
   }
   // Gib dessen Adresse zurück
   return wurzel;
}

int entfernen (t_baum *&wurzel, int deldaten)
{
   t_baum *delknoten = wurzel;
   t_baum *vorgaenger;
   int check_bal;
```

```
   if (wurzel != NULL)
   {
      // Prüfe, wo sich das zu löschende Element befindet
      if (deldaten < wurzel->daten)
         check_bal = entfernen (wurzel->links, deldaten);
      else if (deldaten > wurzel->daten)
         check_bal = entfernen (wurzel->rechts, deldaten);
      else
      {
         // Zu löschendes Element gefunden. Prüfe, ob das Element
         // direkt durch einen Sohn ersetzt werden kann
         if (wurzel->rechts == NULL)
         {
            wurzel = wurzel->links;
            free (delknoten);
            return 1;
         }
         else if (wurzel->links == NULL)
         {
            wurzel = wurzel->rechts;
            free (delknoten);
            return 1;
         }
         else
         {
            // Wenn nicht, dann ermittle den symmetrischen Vorgänger der
            // Wurzel und ersetze die Daten der Wurzel durch die Daten des
            // Vorgänger. Danach muß nur noch der Vorgänger gelöscht werden
            vorgaenger = maximum (wurzel->links);
            wurzel->daten = vorgaenger->daten;
            check_bal = entfernen (wurzel->links, wurzel->daten);
         }
      }
      if (check_bal)
      {
         // Wenn der Teilbaum in der Höhe gefallen ist, dann korrigiere
         // den Balance-Faktor der Wurzel
         if (deldaten < wurzel->daten)
            wurzel->balance --;
         else
            wurzel->balance ++;
         // Wird dabei die Regel für AVL-Bäume verletzt, muß rotiert werden
         if ((wurzel->balance == -2) || (wurzel->balance == 2))
            rotieren (wurzel);
         // Die Höhe des Teilbaums ist genau dann um eins gefallen, wenn der
         // Balance-Faktor der Wurzel 0 wurde
         return (wurzel->balance == 0);
      }
   }
   return 0;
}

void ausgabe (t_baum *wurzel, int tiefe)
{
   // Durchlaufe den Baum und gib alle Knoten entsprechend ihrer Tiefe aus
   if (wurzel != NULL)
   {
      ausgabe (wurzel->rechts, tiefe +1);
      printf ("%*c%d(%d)\n", tiefe *8, ' ', wurzel->daten, wurzel->balance);
      ausgabe (wurzel->links,  tiefe +1);
   }
}

void loeschen (t_baum *wurzel)
// Gibt den gesamten Speicherplatt frei, der vom Baum belegt wird
{
   if (wurzel != NULL)
   {
      // Entferne linken und rechten Teilbaum aus dem Speicher
      loeschen (wurzel->links);
      loeschen (wurzel->rechts);
      // Anschließend gib den von der Wurzel belegten Speicher frei
      free (wurzel);
   }
}

void main (void)
{
   t_baum *wurzel = NULL;
```

```
   einfuegen (wurzel, 4);
   einfuegen (wurzel, 2);
   einfuegen (wurzel, 8);
   einfuegen (wurzel, 1);
   einfuegen (wurzel, 3);
   einfuegen (wurzel, 9);
   einfuegen (wurzel, 6);
   einfuegen (wurzel, 7);
   einfuegen (wurzel, 5);
   ausgabe   (wurzel, 0);
   entfernen (wurzel, 3);
   ausgabe   (wurzel, 0);
   entfernen (wurzel, 1);
   ausgabe   (wurzel, 0);
   loeschen (wurzel);
}
```