```
/*****************************************************************************/
/* Diese Headerdatei beinhaltet alle Deklarationen und Definitionen des      */
/* Hauptprogramms                                                            */
/*****************************************************************************/
#include <getopt.h>

#define FALSE 0
#define TRUE  1

#define UNDEF '\0'
#define DIR_SEP '/'

/* Das Zeichen, mit dem die Längen im Parse_String getrennt werden           */
#define PARSE_IFS ','


/* Folgende Struktur beschreibt die Eigenschaften eines Index, welche        */
/* zusätzlich zum Baum als sozusagen Meta-Informationen in der Indexdatei    */
/* gespeichert werden müssen                                                 */
typedef struct index_tag {
  unsigned int order;;        /* Ordnung, in der der Index-Baum aufgebaut ist */
  unsigned int anz_key; ;             /* Anzahl in Index enthaltene Schlüssel */
   /* Anzahl der folgenden Zeichen, die durch parse_string eingenommen werden */
  unsigned int cnt;
  char *parse_string; ;              /* Zeigt auf parse_string der Indexdatei */
} index_prop;
#define SIZEOF_PROP (3*sizeof(int))

/* enums zur Verbesserung der Lesbarkeit des Codes                           */
typedef enum file_actions_tag {
  CREATE, OPEN
} file_actions;
typedef enum verboseness_tag {
  QUIET, VERBOSE, MORE_VERBOSE
} verboseness;

/* Eine Instanz der Struktur option, die dem kontextsensitiven Parsing der   */
/* Parameter dient                                                           */
struct option long_options[]={
  {"help",    no_argument,       NULL,'h'},
  {"create",  no_argument,       NULL,'c'},
  {"add",     no_argument,       NULL,'a'},
  {"delete",  no_argument,       NULL,'d'},
  {"list",    no_argument,       NULL,'l'},
  {"file",    required_argument, NULL,'f'},
  {"parse",   required_argument, NULL,'p'},
  {"range",   no_argument,       NULL,'r'},
  {"order",   required_argument, NULL,'o'},
  {"newline", required_argument, NULL,'n'},
  {"verbose", optional_argument, NULL,'v'}
};

/* Eine initialisierte, instantiierte Struktur, die die dem Programm         */
/* übergebenen Parameter aufnimmt                                            */
struct param_tag {
  char action;
  unsigned int order;
  char *parse;
  char *filename;
  FILE *src;
  unsigned int crbytes;
  verboseness verbose;
} params={UNDEF, 16, NULL, NULL, NULL, 0, QUIET};
```

```c
/*****************************************************************************/
/* Dieses Programm stellt ein Tool zur Indexverwaltung dar                   */
/*****************************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "btree.h"

#include "index.h"

/* lokale Strukturen, Variablen */
struct key_lens_tag {
  unsigned int len_value;
  unsigned int anz_key;
  unsigned int *len_key;
  unsigned int len_keyall;
} keys;

index_prop idx_prop;
btree_param tree_param;
unsigned int counter;

/*****************************************************************************/
/* Diese Funktion dient dem Parsen des Formatstrings für die Eingabe         */
/*****************************************************************************/
int parse_param(char *s)
{
  unsigned int i=0, ok;
  char *p, *t;
  p=s;
  while((p=strchr(p, PARSE_IFS))!=NULL)
  {
    keys.anz_key++;
    p++;
  }
  if(keys.anz_key==0) return 1;
  keys.len_key=malloc(keys.anz_key*sizeof(int));

  t=malloc(strlen(s));
  while((p=strchr(s, PARSE_IFS))!=NULL)
  {
    t[0]='\0';
    strncpy(t, s, p-s);
    t[p-s]='\0';
    keys.len_key[i]=atoi(t);
    i++;
    s=p+1;
  }
  keys.len_value=atoi(s);
  ok=0;
  keys.len_keyall=0;
  free(t);

  for(i=0; i<keys.anz_key; i++)
  {
    if(*keys.len_key>0) ok=1;
    keys.len_keyall+=keys.len_key[i];
```

```
  }
  keys.len_keyall+=keys.len_value;
  if(ok) return 0;
  return ok;
}

/*****************************************************************************/
/* Diese Funktion implementiert einen einfachen Stringvergleich              */
/*****************************************************************************/
int compar(const void *key1, const void *key2)
{
  unsigned int i;
  int cmp;
  for(i=0; i<keys.len_key[0]; i++)
  {
    cmp=((char *)key1)[i]-
        ((char *)key2)[i];
    if(cmp!=0) return cmp;
  }
  return cmp;
}

/*****************************************************************************/
/* Diese Funktion wird für jeden Schlüssel der Abfragemenge aufgerufen       */
/*****************************************************************************/
void iterkey(const char *key)
{
  int i;
  /* Schlüssel ausgeben */
/*  for(i=0; i<keys.len_keyall-keys.len_value; i++)
    printf("%c", key[i]);*/
  /* Wert ausgeben */
  for(i=keys.len_keyall-keys.len_value; i<keys.len_keyall; i++)
    printf("%c", key[i]);
  printf("\n");
  counter++;
}

/*****************************************************************************/
/* Diese Funktion gibt den Hilfetext des Programms aus                       */
/*****************************************************************************/
void do_help(char *progname)
{
  char *p;
  if((p=strrchr(progname, DIR_SEP))!=NULL)
    progname=p+1;
  printf(
"%s - Verwaltung eines Index zu einer Datenbasis\n\n"
"Das Programm arbeitet auf Basis von Stringvergleichen zwischen den
Schlüsseln\n"
"Dadurch ist es möglich nach Strings, Werten sowie binären Werten zu
sortieren.\n\n"
"Syntax:\n"
"%s <action> [options]\n\n"
"Aktionen:\n"
"-h, --help      Gibt eine kurze Beschreibung des Programms aus\n"
"-c, --create    Erstellt initial einen Index aus Schlüssel-Werte-Paaren an
der\n"
```

```
"                Standardeingabe\n"
"-a, --add       Fügt dem Index Schlüssel-Werte-Paare an der Standardeingabe\n"
"                hinzu\n"
"-d, --delete    Löscht Schlüssel aus dem Index, die über die Standardeingabe\n"
"                übergeben werden. Hier dürfen nur die Schlüssel übergeben
werden"
"-r, --range     Führt eine Bereichsabfrage über den Index aus. Dazu werden\n"
"                zwei Schlüssel der Standardeingabe entnommen.\n"
"-l, --list      Gibt die Eigenschaften des Index aus. Dazu gehören die\n"
"                Definition der Schlüssel- und Wertlängen, die Ordnung und
die\n"
"                Anzahl der Schlüssel.\n\n"
"Optionen:\n"
"-f, --file      Gibt die Datei des zu verwaltenden Indexes an\n"
"-p, --parse     Über diese Option werden die Schlüssel- und Wertelängen\n"
"                angegeben, die über die Standardeingabe eingelesen werden\n"
"                sollen. Diese Option wird nur für -c benötigt, da bei einem\n"
"                einmal erstellten Index der Aufbau der Schlüsseln nicht mehr\n"
"                geändert werden kann.\n"
"                z.B. -p 5,10 für 5 Byte Schlüssel und 10 Byte Wert\n"
"-o, --order     Gibt die Ordnung an, in der der Indexbaum aufgebaut werden
soll\n"
"                Diese Option hat ebenfalls nur beim Erstellen des Index eine\n"
"                Wirkung. Default: 16\n"
"-n, --newline   Gibt an, wieviel Byte zwischen den Schlüssel-Werte-Paaren im\n"
"                Eingabestrom überlesen werden sollen. Default: 0\n"
"-v, --verbose   Ausführliche Ausgabe (Abschließende Statusmeldung)\n"
"-vv, --verbosev Ausführlichere Ausgabe (zusätzlich Ausgabe des akt.
Schlüssels)\n"
  , progname, progname);
}

/*****************************************************************************/
/* Diese Funktion öffnet oder erstellt eine neue Indexdatei                  */
/*****************************************************************************/
void openfile(char *filename, file_actions action, btree *tree)
{
  FILE *f;
  if(action==CREATE)
  {
    f=fopen(filename, "w+b");
    if(f==NULL)
    {
      fprintf(stderr, "indexfile %s could not be opened\n", filename);
      exit(1);
    }
    idx_prop.order=params.order;
    idx_prop.cnt=strlen(params.parse)+1;
    idx_prop.parse_string=params.parse;
    idx_prop.anz_key=0;
    fseek(f, 0, SEEK_SET);
    fwrite(&idx_prop, SIZEOF_PROP, 1, f);
    fwrite(idx_prop.parse_string, sizeof(char), idx_prop.cnt, f);
  }
  else
  {
    f=fopen(filename, "r+b");
    if(f==NULL)
```

```
      /* Datei existiert noch nicht */
      f=fopen(filename, "w+b");
    if(f==NULL)
    {
      fprintf(stderr, "indexfile %s could not be opened\n", filename);
      exit(1);
    }
    fseek(f, 0, SEEK_SET);
    fread(&idx_prop, SIZEOF_PROP, 1, f);
    idx_prop.parse_string=malloc(idx_prop.cnt*sizeof(char));
    if(idx_prop.parse_string==NULL)
    {
      printf("Bad indexfile!\n");
      exit(1);
    }
    fread(idx_prop.parse_string, sizeof(char), idx_prop.cnt, f);
  }
  tree_param.f=f;
  tree_param.ofs=SIZEOF_PROP+idx_prop.cnt*sizeof(char);

  tree->order=idx_prop.order;
  tree->param=&tree_param;
  tree->keylen=keys.len_keyall;
  tree->root=NULL;
  btree_init(tree);
  tree->compar=&compar;
}
/*****************************************************************************/
/* Diese Funktion schließt eine bereits geöffnete Indexdatei                 */
/*****************************************************************************/
void closefile(btree *tree)
{
  fseek(((btree_param *)(tree->param))->f, 0, SEEK_SET);
  fwrite(&idx_prop, SIZEOF_PROP, 1, ((btree_param *)(tree->param))->f);
  fwrite(idx_prop.parse_string, sizeof(char), idx_prop.cnt, ((btree_param
*)(tree->param))->f);
  btree_exit(tree);
  fclose(((btree_param *)(tree->param))->f);
}

/*****************************************************************************/
/* Diese Funktionen kapseln ihre jeweilige Funktionalität                    */
/*****************************************************************************/
void do_create(btree *tree)
{
  char *buf;
  int i;
  unsigned int cnt=0;
  char c;
  openfile(params.filename, CREATE, tree);
  buf=malloc(keys.len_keyall+params.crbytes);

  while(fread(buf, keys.len_keyall+params.crbytes, 1, params.src)==1)
  {
    if(!btree_addkey(buf, tree))
    {
      if(params.verbose==MORE_VERBOSE)
      {
```

```
        printf("adding: ");
        for(i=0; i<keys.len_key[0]; i++)
          printf("%c", buf[i]);
        printf("\n");
      }
      cnt++;
    }
    else
      if(params.verbose==MORE_VERBOSE)
      {
        printf("could not add: ");
        for(i=0; i<keys.len_key[0]; i++)
          printf("%c", buf[i]);
        printf("\n");
      }
  }

  idx_prop.anz_key+=cnt;
  if(params.verbose!=QUIET&&cnt>0)
    printf("%u Schlüssel erfolgreich hinzugefügt.\n", cnt);
  free(buf);
  closefile(tree);
}

void do_add(btree *tree)
{
  char *buf;
  unsigned int i,cnt=0;
  openfile(params.filename, OPEN, tree);
  parse_param(idx_prop.parse_string);
  buf=malloc(keys.len_keyall+params.crbytes);

  while(fread(buf, keys.len_keyall+params.crbytes, 1, params.src)==1)
  {
    if(!btree_addkey(buf, tree))
    {
      if(params.verbose==MORE_VERBOSE)
      {
        printf("adding: ");
        for(i=0; i<keys.len_key[0]; i++)
          printf("%c", buf[i]);
        printf("\n");
      }
      cnt++;
    }
    else
      if(params.verbose==MORE_VERBOSE)
      {
        printf("could not add: ");
        for(i=0; i<keys.len_key[0]; i++)
          printf("%c", buf[i]);
        printf("\n");
      }
  }
  idx_prop.anz_key+=cnt;
  if(params.verbose!=QUIET&&cnt>0)
    printf("%u Schlüssel erfolgreich hinzugefügt.\n", cnt);
  free(buf);
```

```
  closefile(tree);
}

void do_delete(btree *tree)
{
  char *buf;
  unsigned int i, cnt=0;
  openfile(params.filename, OPEN, tree);
  parse_param(idx_prop.parse_string);
  buf=malloc(keys.len_keyall-keys.len_value+params.crbytes);

  while(fread(buf, keys.len_keyall-keys.len_value+params.crbytes, 1,
params.src)==1)
  {
    if(!btree_delkey(buf, tree))
    {
      if(params.verbose==MORE_VERBOSE)
      {
        printf("delete: ");
        for(i=0; i<keys.len_key[0]; i++)
          printf("%c", buf[i]);
        printf("\n");
      }
      cnt++;
    }
    else
      if(params.verbose!=QUIET)
      {
        printf("could not delete: ");
        for(i=0; i<keys.len_key[0]; i++)
          printf("%c", buf[i]);
        printf("\n");
      }
  }
  idx_prop.anz_key-=cnt;
  if(params.verbose!=QUIET&&cnt>0)
    printf("%u Schlüssel erfolgreich gelöscht.\n", cnt);
  free(buf);
  closefile(tree);
}

void do_list(btree *tree)
{
  openfile(params.filename, OPEN, tree);
  printf("This index has the following properties:\n"
         "order              : %u\n"
         "key-,value-length  : %s\n"
         "keys               : %u\n"
         , idx_prop.order, idx_prop.parse_string, idx_prop.anz_key);
  closefile(tree);
}

void do_rangequery(btree *tree)
{
  char *from, *to;
  openfile(params.filename, OPEN, tree);
  parse_param(idx_prop.parse_string);
  from=malloc(keys.len_keyall-keys.len_value+params.crbytes);
```

```
  to=malloc(keys.len_keyall-keys.len_value+params.crbytes);

  counter=0;
  if(fread(from, keys.len_keyall-keys.len_value+params.crbytes, 1,
params.src)!=1)
    printf("begin of range could not be read!");
  else
  if(fread(to, keys.len_keyall-keys.len_value+params.crbytes, 1, params.src)!=1)
    printf("end of range could not be read!");
  else
    btree_rangeiterate(from, to, &iterkey, tree);
  if(params.verbose!=QUIET)
    printf("%u Schlüssel abgefragt.\n", counter);
  free(from);
  free(to);
  closefile(tree);
}

/* Diese Funktion (Makro) ist quasi-inline, wie aus C++ bekannt */
#define invalidaction() \
{\
  fprintf(stderr, "Only one action possible!\n");\
  exit(1);\
}

int main(int argc, char *argv[])
{
  int c;
  int digit_optind=0;
  int option_index;
  /* Wird die Option o nicht angegeben wird Baum mit Ordnung 16 benutzt */
  btree tree={16, 0, NULL, -1, NULL, 0 };
  FILE *f;
  /* Gibt die Anzahl der Bytes ab Anfang des Streams an, die für ihn */
  /* reserviert bleiben                                              */
  unsigned int ofs;
  tree.param=&tree_param;
  params.src=stdin;
  /* Parameter einlesen */
  while((c=getopt_long(argc, argv, "cadrlhf:p:o:n:v::",
                       long_options, &option_index))!=':'&&c!='?'&&c!=EOF)
    switch(c)
    {
      case 'h' :
        if(!params.action) params.action=c;
        else invalidaction();
        break;
      case 'c' :
        if(!params.action) params.action=c;
        else invalidaction();
        break;
      case 'a' :
        if(!params.action) params.action=c;
        else invalidaction();
        break;
      case 'd' :
        if(!params.action) params.action=c;
        else invalidaction();
```

```
      break;
    case 't' :
      if(!params.action) params.action=c;
      else invalidaction();
      break;
    case 'l' :
      if(!params.action) params.action=c;
      else invalidaction();
      break;
    case 'r' :
      if(!params.action) params.action=c;
      else invalidaction();
      break;
    case 'v' :
      if(optarg)
      {
        if(!strcmp(optarg, "v"))
          params.verbose=MORE_VERBOSE;
        else
        {
          printf("Unknown verboseness level!\n");
          exit(1);
        }
      }
      else
        params.verbose=VERBOSE;
      break;
    case 'f' :
      if(!optarg)
      {
        printf("No argument specified for filename.\n");
        exit(1);
      }
      params.filename=optarg;
      break;
    case 'o' :
      if(optarg) params.order=atoi(optarg);
      break;
    case 'n' :
      if(optarg) params.crbytes=atoi(optarg);
      break;
    case 'p' :
      if(!optarg)
      {
        printf("No argument specified for parsestring.\n");
        exit(1);
      }
      params.parse=optarg;
      break;
  }
/* Parse-Ende überprüfen */
switch(c)
{
  case ':' : /* fehlendes Argument */
    fprintf(stderr, "missing argument for %s\n", argv[optind+1]);
    exit(1);
    break;
  case '?' : /* unbekannte Option */
```

```
      exit(1);
      break;
    case EOF : /* Optionen ok */
    default  :
  }
  /* Validitätsprüfung und Ausführung */
  if(params.action==UNDEF)
  {
    fprintf(stderr, "No action specified! - help assumed\n\n");
    do_help(argv[0]);
    exit(1);
  }
  switch(params.action)
  {
    case 'h' : /* Hilfeausgabe */
      do_help(argv[0]);
      break;
    default:
      if(params.filename==NULL)
      {
        fprintf(stderr, "No indexfile specified!\n");
        exit(1);
      }
      switch(params.action)
      {
        case 'c' : /* Index erstellen */
          if(params.parse==NULL)
          {
            fprintf(stderr, "No parsestring specified!\n");
            exit(1);
          }
          if(parse_param(params.parse))
          {
            fprintf(stderr, "Bad formatted parsestring: %s\n", params.parse);
            exit(1);
          }
          do_create(&tree);
          break;
        case 'l' : /* Indexeigenschaften ausgeben */
          do_list(&tree);
          break;
        case 'a' : /* Schlüssel hinzufügen */
          do_add(&tree);
          break;
        case 'd' : /* Schlüssel entfernen */
          do_delete(&tree);
          break;
        case 'r' : /* Bereichsabfrage */
          do_rangequery(&tree);
          break;
        default : /* sollte nicht auftreten können */
      }
  }
  free(keys.len_key);
  return 0;
}
```

```
#if !defined(_BTREE_H)
#define _BTREE_H
/*****************************************************************************/
/* Diese Headerdatei beinhaltet alle Deklarationen und Prototypen für        */
/* einen B-Baum                                                              */
/*****************************************************************************/

#if defined(_NODEFILE)
  #include "nodefile.h"
#else
  #include "nodemem.h"
#endif

/* Diese Struktur beschreibt einen B-Baum, sie wird als Parameter            */
/* von den btree-Funktionen erwartet und dient als Einstiegspunkt zum Baum   */
typedef struct btree_tag {
  unsigned int order;                              /* die Ordnung des Baumes */
  unsigned int keylen;  /* Länge in Byte für die Speicherung eines Schlüssels */
  int (*compar)(const void *, const void *); /* Zeiger auf Vergleichsfunktion */
  int height;               /* bei Operationen aktualisierte Höhe des Baumes */
  NODE root;             /* Zeiger auf die Wurzel des Baumes, Einstiegspunkt */
  void *param; /*Zeiger auf weitere Infos, von denen der Baum noch nichts weiß*/
} btree;

/* Dieser "Schalter" dient der zusätzlichen evtl. störenden Ausgabe von      */
/* Debug-Informationen auf dem Standardfehlerkanal                           */
/* Einfach eine der beiden folgenden Zeilen auskommentieren                  */
#define _DEBUG
#undef _DEBUG

/* Zum Suchen der richtigen Position innerhalb eines Knoten sind verschiedene */
/* Suchverfahren verwendbar.                                                 */
/* Wenn _SEARCH_ALG_LIN definiert ist, wird Linearsuche genommen (für kleine  */
/* Ordnungen/linke Schlüssel suchen günstig), sonst BinarySearch             */
/* Einfach eine der beiden folgenden Zeilen auskommentieren                  */
#define _SEARCH_ALG_LIN
#undef _SEARCH_ALG_LIN

/* Error-Codes                                                               */
#define ERR_ALLOC -1
#define ERR_KEY_NOT_FOUND -2

/*****************************************************************************/
/* Prototypen                                                                */
/*****************************************************************************/

/* Diese Funktion gibt dem Baum die Chance sich zu initialisieren            */
int btree_init(btree *tree);
/* Diese Funktion gibt dem Baum die Chance z.B. Betriebsmittel freizugeben   */
void btree_exit(btree *tree);
/* Diese Funktion fügt einen Schlüssel dem B-Baum hinzu                      */
int btree_addkey(const void *key, btree *tree);
/* Diese Funktion entfernt einen Schlüssel aus dem B-Baum                    */
int btree_delkey(const void *key, btree *tree);
/* Diese Funktion implementiert eine Bereichsabfrage nach Schlüsseln         */
unsigned int btree_rangeiterate(void *keyfrom, void *keyto,
                                void (*iter)(const void *key), btree *tree);
/* Dieses Makro implementiert eine Schlüsselabfrage                          */
```

```
#define btree_keyiterate(key, iter, tree) \
        btree_rangeiterate(key, key, iter, tree)
/* Diese Funktion implementiert eine Bereichsabfrage zur Bestimmung des       */
/* Umfanges der Abfragemenge                                                  */
unsigned int btree_rangecount(void *keyfrom, void *keyto, btree *tree);
/* Diese Funktion ermittelt den kleinsten im Baum gespeicherten Schlüssel     */
int btree_getkeylowest(void *dest, NODE n, btree *tree);
/* Diese Funktion ermittelt den größten im Baum gespeicherten Schlüssel       */
int btree_getkeyhighest(void *dest, NODE n, btree *tree);
/* Diese Funktion führt eine Tiefensuche nach einer bestimmten Tiefe          */
/* ab einem bestimmten Knoten durch und                                       */
/* ruft für jeden Schlüssel dieser Tiefe eine Funktion auf                    */
void btree_iteratenodedepth(NODE n, unsigned int depth,
       void (*iter)(unsigned int depth, NODE node, void *key, btree *tree),
       btree *tree);
/* Dieses Makro führt eine Tiefensuche nach einer bestimmten Tiefe            */
/* im Baum durch                                                              */
#define btree_iteratedepth(depth, iter, tree) \
        btree_iteratenodedepth(tree->root, depth, iter, tree)
/* Diese Funktion ruft für alle Schlüssel in allen Ebenen des Baumes          */
/* nacheinander eine Funktion auf                                             */
void btree_iteratedepths(
       void (*iter)(unsigned int depth, NODE node, void *key, btree *tree),
       btree *tree);

#endif
```

```
/****************************************************************************/
/* Diese Quelldatei beinhaltet alle Routinen zum Handling von B-Bäumen      */
/****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <setjmp.h>

#include "btree.h"
#include "node.h"

/* interne Konstanten                                                       */
#define FALSE 0
#define TRUE 1
#define ANYTHING ((void*)1)

/****************************************************************************/
/* Diese Funktion gibt den Index des richtigen Links für den erstmöglichen  */
/* Abstieg bzw. die Position, an der das Element im Blatt bei LIFO eingefügt */
/* werden muß, zurück                                                       */
/****************************************************************************/
/* Parameter:                                                               */
/*   n    - Zeigt auf den Knoten, in dem nach key gesucht werden soll       */
/*   key  - Zeigt auf den gesuchten Schlüssel                               */
/*   tree - Zeigt auf den B-Baum                                            */
/* Ergebnis:                                                                */
/*   Position in der key im Knoten einzufügen ist                           */
/****************************************************************************/
static unsigned int btree_getposfirst(NODE n, void *key, btree *tree)
{
  /* Hier sind effektivere Implementierungen möglich:
     Linearsuche, bsearch, directShot, etc. */
#if defined(_SEARCH_ALG_LIN)
  /* Linearsuche: */
  unsigned int i, cnt;
  int cmp;
  void *k;
  k=malloc(tree->keylen);
  cnt=node_getcount(n, tree);
  for(i=0; i<cnt; i++)
  {
    node_getkey(k, n, i, tree);
    cmp=(*tree->compar)(key, k);
    if(cmp<=0)
    {
      /* key ist kleiner oder gleich akt. Schlüssel */
      break;
    }
  }
  free(k);
  return i;
#else
  /* Binärsuche: */
  unsigned int l=0, m, r;
  int cmp;
  void *k;
  k=malloc(tree->keylen);
  r=node_getcount(n, tree)-1;
```

```
  node_getkey(k, n, 0, tree);
  if((*tree->compar)(key, k)<=0)
  {
    free(k);
    return 0;
  }
  while(l<r)
  {
    m=(l+r+1)/2;
    node_getkey(k, n, m, tree);
    cmp=(*tree->compar)(key, k);
    if(cmp<=0)
      /* key ist kleiner als akt. Schlüssel */
      r=m-1;
    else
    if(cmp>0)
      /* key ist größer als akt. Schlüssel */
      l=m;
  }
  free(k);
  return l+1;
#endif
}

/*****************************************************************************/
/* Diese Funktion gibt den Index des richtigen Links für den letztmöglichen  */
/* Abstieg bzw. die Position, an der das Element im Blatt bei FIFO eingefügt */
/* werden muß, zurück                                                        */
/*****************************************************************************/
/* Parameter:                                                                */
/*   n    - Zeigt auf den Knoten, in dem nach key gesucht werden soll        */
/*   key  - Zeigt auf den gesuchten Schlüssel                                */
/*   tree - Zeigt auf den B-Baum                                             */
/* Ergebnis:                                                                 */
/*   Position in der key im Knoten einzufügen ist                            */
/*****************************************************************************/
static unsigned int btree_getposlast(const NODE n, const void *key, btree *tree)
{
  /* Hier sind effektivere Implementierungen möglich:
     Linearsuche, bsearch, directShot, etc. */
#if defined(_SEARCH_ALG_LIN)
  /* Linearsuche: */
  unsigned int i, cnt;
  int cmp;
  void *k;
  k=malloc(tree->keylen);
  cnt=node_getcount(n, tree);
  for(i=0; i<cnt; i++)
  {
    node_getkey(k, n, i, tree);
    cmp=(*tree->compar)(key, k);
    if(cmp<0)
    {
      /* key ist kleiner akt. Schlüssel */
      break;
    }
  }
  free(k);
```

```
  return i;
#else
  /* Binärsuche: */
  unsigned int l=0, m, r;
  int cmp;
  void *k;
  k=malloc(tree->keylen);
  r=node_getcount(n, tree)-1;
  node_getkey(k, n, 0, tree);
  if((*tree->compar)(key, k)<0)
  {
    free(k);
    return 0;
  }
  while(l<r)
  {
    m=(l+r+1)/2;
    node_getkey(k, n, m, tree);
    cmp=(*tree->compar)(key, k);
    if(cmp<0)
      /* key ist kleiner als akt. Schlüssel */
      r=m-1;
    else
    if(cmp>=0)
      /* key ist größer als akt. Schlüssel */
      l=m;
  }
  free(k);
  return l+1;
#endif
}

/*****************************************************************************/
/* Diese Funktion wandert rekursiv im Baum hinab, fügt dort den Schlüssel ein */
/* und restauriert den Baum hinaufgehend bis zur Wurzel                      */
/*****************************************************************************/
/* Parameter:                                                                */
/*   n    - Zeigt auf den Knoten, ab dem nach key gesucht werden soll        */
/*   key  - Zeigt auf den gesuchten Schlüssel                                */
/*   tree - Zeigt auf den B-Baum                                             */
/* Ergebnis:                                                                 */
/*   Zeiger auf evtl. neu erstelltes rechtes Kind                            */
/*****************************************************************************/
static NODE btree_descend(NODE n, const void *key, btree *tree)
{
  NODE rnode;
  NODE t;
  NODE tnode;
  NODE result;
  unsigned int pos;
  void *k;
  /* Baum existiert bereits */

  /* Einfügen nach FIFO */
  pos=btree_getposlast(n, key, tree);

  t=node_getlnk(n, pos, tree);
  if(t!=NULL)
```

```
  {
    /* aktueller Knoten hat Kinder */
    rnode=btree_descend(t, key, tree);
    if(rnode==NULL||(int)rnode==ERR_ALLOC)
      return rnode;
    else
    {
      if(node_getcount(n, tree)<tree->order*2)
      {
        /* Schlüssel paßt in Knoten */
        k=malloc(tree->keylen);
        if(node_getkey(k, t, tree->order, tree))
          node_insertkey(n, NULL, rnode, pos, tree);
        else
          node_insertkey(n, k, rnode, pos, tree);
        free(k);
        return NULL;
      }
      else
      {
        /* Knoten ist schon voll */
        tnode=node_alloc(tree);
        if(tnode==NULL)
        {
          fprintf(stderr, "Kein Speicher allozierbar!");
          return (NODE)ERR_ALLOC;
        }
        k=malloc(tree->keylen);
        if(node_getkey(k, t, tree->order, tree))
          result=node_insertkey_split(n, tnode, NULL, rnode, pos, tree);
        else
          result=node_insertkey_split(n, tnode, k, rnode, pos, tree);
        free(k);
        return result;
      }
    }
  }
  else
  {
  /* Blatt erreicht */
    if(node_getcount(n, tree)<tree->order*2)
    {
      /* Schlüssel paßt in Blatt */
      node_insertkey(n, key, NULL, pos, tree);
      return NULL;
    }
    else
    {
      /* Blatt ist schon voll */
      tnode=node_alloc(tree);
      if(tnode==NULL)
      {
        fprintf(stderr, "Kein Speicher allozierbar!");
        return (NODE)ERR_ALLOC;
      }
      return node_insertkey_split(n, tnode, key, NULL, pos, tree);
    }
  }
```

```
}

/*****************************************************************************/
/* Diese Funktion implementiert eine rekursive Suche nach dem letzten        */
/* Auftreten eines Schlüssels ab einem bestimmten Knoten                     */
/*****************************************************************************/
/* Parameter:                                                                */
/*   key_struct - Zeiger auf eine Struktur, die das Ergebnis aufnehmen soll  */
/*   n          - Knoten, ab dem gesucht werden soll                         */
/*   key        - Zeigt auf den niedrigsten gewünschten Schlüssel            */
/*   tree       - Zeigt auf den B-Baum                                       */
/* Ergebnis:                                                                 */
/*   TRUE       - es wurde kein Schlüssel gefunden,                          */
/*                ansonsten FALSE                                            */
/*****************************************************************************/
typedef struct key_tag {
  NODE node;
  unsigned int pos;
} key;
static int btree_getkeylast(key *key_struct, NODE n, void *key, btree *tree)
{
  unsigned int i, pos;
  NODE t;

  /* Abbruchbedingung: Blatt gefunden */
  if(n==NULL)
    return TRUE;
  pos=btree_getposlast(n, key, tree);

  t=node_getlnk(n, pos, tree);
  if(btree_getkeylast(key_struct, t, key, tree)==TRUE)
  {
    /* der Anfang wurde bereits rekursiv tiefer gefunden */
    if(pos==0)
      return TRUE;
    else
    {
      /* letzter Schlüssel befindet sich noch im aktuellen Knoten */
      key_struct->node=n;
      key_struct->pos=pos-1;
      return FALSE;
    }
  }
  return FALSE;
}

/*****************************************************************************/
/* Diese Funktion implementiert eine rekursive Bereichsabfrage ab einem      */
/* bestimmten Knoten im Baum                                                 */
/*****************************************************************************/
/* Parameter:                                                                */
/*   anz     - Zeigt auf ein int, das das Ergebnis aufnehmen soll            */
/*   n       - Knoten, ab dem gesucht werden soll                            */
/*   keyfrom - Zeigt auf den niedrigsten gewünschten Schlüssel               */
/*   keyto   - Zeigt auf eine Struktur, die die Position des höchsten        */
/*             gewünschten Schlüssels enthält                                */
/*   iter    - Zeigt auf eine Funktion, die für jeden gefundenen Schlüssel   */
/*             aufgerufen wird                                               */
```

```
/*   tree    - Zeigt auf den B-Baum                                        */
/* Ergebnis:                                                              */
/*   TRUE    - aktueller Knoten gehört zur Auswahlmenge,                  */
/*             ansonsten FALSE                                            */
/************************************************************************/
static int btree_rangeiteratered(unsigned int *anz, NODE n, void *keyfrom, key
*keyto,
                                 void (*iter)(const void *key), btree *tree)
{
  unsigned int i, pos, cnt;
  NODE t;
  void *k;

  /* Abbruchbedingung: Blatt gefunden */
  if(n==NULL)
    return TRUE;
  pos=btree_getposfirst(n, keyfrom, tree);

  t=node_getlnk(n, pos, tree);
  if(btree_rangeiteratered(anz, t, keyfrom, keyto, iter, tree)==TRUE)
  {
    /* der Anfang wurde bereits rekursiv tiefer gefunden */
    cnt=node_getcount(n, tree);
    for(i=pos; i<cnt; i++)
    {
      if(!(n==keyto->node&&i>=keyto->pos))
      {
        k=malloc(tree->keylen);
        node_getkey(k, n, i, tree);
        (*iter)(k);
        free(k);
        (*anz)++;
        /* Solange nicht der letzte gefunden ist, Links folgen */
        t=node_getlnk(n, i+1, tree);
        if(btree_rangeiteratered(anz, t, keyfrom, keyto, iter, tree)!=TRUE)
          return FALSE;
      }
      else
        return FALSE;
    }
    return TRUE;
  }
  return FALSE;
}

/************************************************************************/
/* Diese Funktion implementiert eine rekursive Bereichsabfrage zur Bestimmung */
/* der Größe der Abfragemenge                                             */
/************************************************************************/
/* Parameter:                                                             */
/*   anz     - Zeigt auf ein int, das das Ergebnis aufnehmen soll         */
/*   n       - Knoten, ab dem gesucht werden soll                         */
/*   keyfrom - Zeigt auf den niedrigsten gewünschten Schlüssel            */
/*   keyto   - Zeigt auf eine Struktur, die den höchsten gewünschten      */
/*             Schlüssel referenziert                                     */
/*   tree    - Zeigt auf den B-Baum                                       */
/* Ergebnis:                                                              */
/*   TRUE    - aktueller Knoten gehört zur Auswahlmenge,                  */
```

```
/*             ansonsten FALSE                                              */
/*****************************************************************************/
static int btree_rangecountrec(unsigned int *anz, NODE n,
                               void *keyfrom, key *keyto, btree *tree)
{
  unsigned int i, pos, cnt;
  NODE t;

  /* Abbruchbedingung: Blatt gefunden */
  if(n==NULL)
    return TRUE;
  pos=btree_getposfirst(n, keyfrom, tree);

  t=node_getlnk(n, pos, tree);
  if(btree_rangecountrec(anz, t, keyfrom, keyto, tree)==TRUE)
  {
    /* der Anfang wurde bereits rekursiv tiefer gefunden */
    cnt=node_getcount(n, tree);
    for(i=pos; i<cnt; i++)
    {
      (*anz)++;
      if(n!=keyto->node||i!=keyto->pos)
      {
        /* Solange nicht der letzte gefunden ist, Links folgen */
        t=node_getlnk(n, i+1, tree);
        if(btree_rangecountrec(anz, t, keyfrom, keyto, tree)!=TRUE)
          return FALSE;
      }
      else
        return FALSE;
    }
    return TRUE;
  }
  return FALSE;
}

/*****************************************************************************/
/* Diese Funktion wandert rekursiv rechts im Baum hinab, holt sich den       */
/* Schlüssel und restrukturiert den Baum hinaufwandernd                      */
/* und restauriert den Baum hinaufgehend bis zur Wurzel                      */
/*****************************************************************************/
/* Parameter:                                                                */
/*   dest - Zeiger auf Speicher, in den der rechteste Schlüssel soll         */
/*   n    - Zeigt auf den Vaterschlüssel                                     */
/*   tree - Zeigt auf den B-Baum                                             */
/* Ergebnis:                                                                 */
/*   NULL bei abgeschlossenem Entfernen                                      */
/*****************************************************************************/
static void *btree_del_borrow(void *dest, NODE n, btree *tree)
{
  unsigned int cnt;
  NODE l;
  NODE r;
  void *k;

  cnt=node_getcount(n, tree);
  r=node_getlnk(n, cnt, tree);
  if(r==NULL)
```

```
  {
    /* Knoten ist Blatt -> letzten Schlüssel entfernen */
    node_getkey(dest, n, cnt-1, tree);
//    memcpy(result, k, tree->keylen);
/*    result->node=n;
    result->pos=cnt-1;*/
    node_delkey(n, cnt-1, tree);
    if(cnt>tree->order)
      /* Schlüssel entfernen störte nicht Integrität */
      return NULL;
    else
      return ANYTHING;
  }

  if(btree_del_borrow(dest, r, tree)==NULL)
    return NULL;
  else
  {
    /* r hat zu wenig Schlüssel: Restaurierung! */
    l=node_getlnk(n, cnt-1, tree);
    if(node_getcount(l, tree)>tree->order)
    {
      /* l und r ausgleichen */
      k=malloc(tree->keylen);
      node_getkey(k, n, cnt-1, tree);
      node_rotateright(k, l, r, tree);
      node_setkey(k, n, cnt-1, tree);
      free(k);
      return NULL;
    }
    else
    {
      /* l,n,r wird neuer Knoten -> letzter Schlüssel von n wird gelöscht */
      k=malloc(tree->keylen);
      node_getkey(k, n, cnt-1, tree);
      node_merge(k, l, r, tree);
      free(k);
      node_free(r);
      node_delkey(n, cnt-1, tree);
      if(node_getcount(n, tree)<tree->order)
        return ANYTHING;
      else
        return NULL;
    }
  }
}

/*****************************************************************************/
/* Diese Funktion wandert rekursiv rechts im Baum hinab, holt sich den       */
/* Schlüssel und restrukturiert den Baum hinaufwandernd                      */
/* und restauriert den Baum hinaufgehend bis zur Wurzel                      */
/*****************************************************************************/
/* Parameter:                                                                */
/*   n       - Zeiger auf Schlüssel                                          */
/*   destkey - Zeiger auf Ergebnisstruktur                                   */
/*   jmp_buf - Abbruch-Sprungposition                                        */
/*   tree - Zeigt auf den B-Baum                                             */
/* Ergebnis:                                                                 */
```

```
/*   TRUE  - Einfügeposition, aber nicht Schlüssel rekursiv tiefer gefunden   */
/*   FALSE - Schlüssel tiefer gefunden, Restaurierung                         */
/****************************************************************************/
static int btree_del_descend(NODE n, const void *destkey, jmp_buf buf,
                             btree*tree)
{
  unsigned int i, pos, res;
  NODE t;
  NODE l;
  NODE r;
  void *result;
  key *borrow;
  void *k;
  int cmp;

  /* Abbruchbedingung: Blatt gefunden */
  if(n==NULL)
    return TRUE;
  /* Suchen von key nach LIFO */
  pos=btree_getposlast(n, destkey, tree);

  t=node_getlnk(n, pos, tree);
  res=btree_del_descend(t, destkey, buf, tree);
  if(res==TRUE)
  {
    /* der Anfang wurde bereits rekursiv tiefer gefunden */
    if(pos==0)
      return TRUE;
    else
    {
      /* letzter Schlüssel befindet sich noch im aktuellen Knoten */
      k=malloc(tree->keylen);
      node_getkey(k, n, pos-1, tree);
      cmp=(*tree->compar)(destkey, k);
      free(k);
      if(cmp==0)
      {
        /* zu löschender Schlüssel gefunden */

        /* hinabsteigen und versuchen einen Schlüssel zu borgen */
        l=node_getlnk(n, pos-1, tree);
        if(l==NULL)
        {
          /* zu löschender Schlüssel ist in Blatt */
          node_delkey(n, pos-1, tree);
          if(node_getcount(n, tree)<tree->order)
            return FALSE;
          else
            longjmp(buf, 2);
        }
        else
        {
          result=malloc(tree->keylen);
          borrow=btree_del_borrow(result, l, tree);
          node_setkey(result, n, pos-1, tree);
          free(result);
          if(borrow==NULL)
          {
```

```
            /* Linker Teilbaum korrekt */
            /* Restaurierung fertig */
            longjmp(buf, 2);
          }
          else
          {
            /* Linker Sohn hat zu wenig Schlüssel */
            r=node_getlnk(n, pos, tree);
            if(node_getcount(r, tree)>tree->order)
            {
              /* l und r ausgleichen */
              k=malloc(tree->keylen);
              node_getkey(k, n, pos-1, tree);
              node_rotateleft(k, l, r, tree);
              node_setkey(k, n, pos-1, tree);
              free(k);
              /* Restaurierung fertig */
              longjmp(buf, 2);
            }
            else
            {
              /* l,n,r wird neuer Knoten */
              k=malloc(tree->keylen);
              node_getkey(k, n, pos-1, tree);
              node_merge(k, l, r, tree);
              free(k);
              node_free(r);
              node_delkey(n, pos-1, tree);
              /* Restaurierung nach oben fortsetzen */
              if(node_getcount(n, tree)<tree->order)
                return FALSE;
              else
                longjmp(buf, 2);
            }
          }
        }
      }
      else
      {
        /* zu löschender Schlüssel existiert nicht im Baum */
        longjmp(buf, 1);
      }
    }
  }
  else
  {
    /* Restaurierung muß weiter bis nach oben fortgesetzt werden */
    if(pos==0)
    {
      /* mit rechtem Kind ausgleichen */
      l=t;
      r=node_getlnk(n, 1, tree);

      if(node_getcount(r, tree)>tree->order)
      {
        /* l und r ausgleichen */
        k=malloc(tree->keylen);
        node_getkey(k, n, 0, tree);
```

```
        node_rotateleft(k, l, r, tree);
        node_setkey(k, n, 0, tree);
        free(k);
        /* Restaurierung fertig */
        longjmp(buf, 2);
      }
      else
      {
        /* l,n,r wird neuer Knoten -> Schlüssel von n wird gelöscht */
        k=malloc(tree->keylen);
        node_getkey(k, n, pos, tree);
        node_merge(k, l, r, tree);
        free(k);
        node_free(r);
        node_delkey(n, pos, tree);
        if(node_getcount(n, tree)<tree->order)
          return FALSE;
        else
          longjmp(buf, 2);
      }
    }
    else
    {
      /* mit linkem Kind ausgleichen */
      l=node_getlnk(n, pos-1, tree);
      r=t;

      if(node_getcount(l, tree)>tree->order)
      {
        /* l und r ausgleichen */
        k=malloc(tree->keylen);
        node_getkey(k, n, pos-1, tree);
        node_rotateright(k, l, r, tree);
        node_setkey(k, n, pos-1, tree);
        free(k);
        /* Restaurierung fertig */
        longjmp(buf, 2);
      }
      else
      {
        /* l,n,r wird neuer Knoten -> letzter Schlüssel von n wird gelöscht */
        k=malloc(tree->keylen);
        node_getkey(k, n, pos-1, tree);
        node_merge(k, l, r, tree);
        free(k);
        node_free(r);
        node_delkey(n, pos-1, tree);
        if(node_getcount(n, tree)<tree->order)
          return FALSE;
        else
          longjmp(buf, 2);
      }
    }
  }
  /* Hier sollte Ausführung niemals hingelangen, aber damit Compiler Ruhe gibt
*/
  return FALSE;
}
```

```
/*****************************************************************************/
/* Diese Funktion führt eine Tiefensuche nach einer bestimmten Tiefe durch   */
/* und ruft für jeden Schlüssel dieser Tiefe eine Funktion auf               */
/*****************************************************************************/
/* Parameter:                                                                */
/*   n     - Zeiger auf Knoten, der die Tiefe 0 hat                          */
/*   level - Tiefe des Knoten                                                */
/*   depth - gesuchte Tiefe                                                  */
/*   iter  - Zeiger auf Funktion, die für jeden Schlüssel der gesuchten Tiefe */
/*           aufgerufen wird                                                 */
/*   tree  - Zeiger auf den Baum                                             */
/*****************************************************************************/
static void btree_iteratedepthrec(NODE n, unsigned int level, unsigned int
depth,
              void (*iter)(unsigned int depth, NODE node, void *key, btree
*tree),
              btree *tree)
{
  int i, cnt;
  void *k;
  if(n==NULL)
    return;
  cnt=node_getcount(n, tree);
  if(depth==level)
  {
    /* gesuchte Zieltiefe erreicht */
    k=malloc(tree->keylen);
    for(i=0; i<cnt; i++)
    {
      node_getkey(k, n, i, tree);
      (*iter)(depth, n, k, tree);
    }
    free(k);
  }
  else
  {
    for(i=0; i<=cnt; i++)
      btree_iteratedepthrec(node_getlnk(n, i, tree),
                            level+1, depth, iter, tree);
  }
}

/*****************************************************************************/
/*****************************************************************************/
/**                                                                         **/
/** Öffentliche Funktionen                                                  **/
/**                                                                         **/
/*****************************************************************************/
/*****************************************************************************/

/*****************************************************************************/
/* Diese Funktion gibt dem Baum die Chance sich zu initialisieren            */
/*****************************************************************************/
/* Parameter:                                                                */
/*   tree - Zeigt auf den B-Baum                                             */
/* Ergebnis:                                                                 */
/*   0 bei erfolgreicher Initialisierung                                     */
```

```
/*****************************************************************************/
int btree_init(btree *tree)
{
  tree->height=-1;
  return node_init(tree);
}

/*****************************************************************************/
/* Diese Funktion gibt dem Baum die Chance z.B. Betriebsmittel freizugeben   */
/*****************************************************************************/
/* Parameter:                                                                */
/*   tree - Zeigt auf den B-Baum                                             */
/*****************************************************************************/
void btree_exit(btree *tree)
{
  node_exit(tree);
}

/*****************************************************************************/
/* Diese Funktion fügt einen Schlüssel dem B-Baum hinzu                      */
/*****************************************************************************/
/* Parameter:                                                                */
/*   key - Zeigt auf den neu einzufügenden Schlüssel                         */
/*   tree - Zeigt auf den B-Baum                                             */
/* Ergebnis:                                                                 */
/*   0 bei erfolgreicher Ausführung, sonst ERR-Code                          */
/*****************************************************************************/
int btree_addkey(const void *key, btree *tree)
{
  NODE n;
  NODE rnode;
  void *k;

  if(tree->root==NULL)
  {
    /* Baum ist noch leer */
    tree->root=node_calloc(key, NULL, NULL, tree);
    if(tree->root==NULL)
      return ERR_ALLOC;
    tree->height=0;
    return 0;
  }

  /* zum Blatt hinabsteigen, in das key gehört und wieder aufsteigen */
  rnode=btree_descend(tree->root, key, tree);
  if(rnode!=NULL)
  {
    /* Eine neue Wurzel muß her */
    k=malloc(tree->keylen);
    node_getkey(k, tree->root, tree->order, tree);
    n=node_calloc(k, tree->root, rnode, tree);
    free(k);
    if(n==NULL)
      return ERR_ALLOC;
    tree->root=n;
    tree->height++;
  }
  return 0;
```

```
}

/*****************************************************************************/
/* Diese Funktion entfernt einen Schlüssel aus dem B-Baum                    */
/*****************************************************************************/
/* Parameter:                                                                */
/*   key - Zeigt auf den zu löschenden Schlüssel                             */
/*   tree - Zeigt auf den B-Baum                                             */
/* Ergebnis:                                                                 */
/*   0 bei erfolgreicher Ausführung, sonst ERR-Code                          */
/*****************************************************************************/
int btree_delkey(const void *key, btree *tree)
{
  NODE n;
  jmp_buf buf;

  if(tree->root==NULL)
    return ERR_KEY_NOT_FOUND;

  switch(setjmp(buf))
  {
    case 0:
      /* zum zu löschenden Knoten hinabsteigen */
      if(btree_del_descend(tree->root, key, buf, tree)==TRUE)
        return ERR_KEY_NOT_FOUND;
      break;
    case 1:
      /* zurück aus Rekursion -> Fehler */
      return ERR_KEY_NOT_FOUND;
    case 2:
      /* zurück aus Rekursion -> fertig */
      return 0;
  }
  if(node_getcount(tree->root, tree)==0)
  {
    /* Wurzel leer -> löschen */
    n=tree->root;
    tree->root=node_getlnk(n, 0, tree);
    node_free(n);
    tree->height--;
  }
  return 0;
}

/*****************************************************************************/
/* Diese Funktion implementiert eine Bereichsabfrage nach Schlüsseln         */
/*****************************************************************************/
/* Parameter:                                                                */
/*   keyfrom - Zeigt auf den niedrigsten gewünschten Schlüssel               */
/*   keyto   - Zeigt auf den höchsten gewünschten Schlüssel                  */
/*   iter    - Zeigt auf eine Funktion, die für jeden gefundenen Schlüssel   */
/*             aufgerufen wird                                               */
/*   tree    - Zeigt auf den B-Baum                                          */
/* Ergebnis:                                                                 */
/*   Umfang der Auswahlmenge                                                 */
/*****************************************************************************/
unsigned int btree_rangeiterate(void *keyfrom, void *keyto,
                                void (*iter)(const void *key), btree *tree)
```

```
{
  unsigned int anz=0;
  key k;
  if(tree->root!=NULL)
  {
    if((*tree->compar)(keyfrom, keyto)>0)
    {
      /* keyto ist kleiner als keyfrom */
      return 0;
    }
    if(btree_getkeylast(&k, tree->root, keyto, tree)==TRUE)
      /* kein Schlüssel gefunden -> leere Menge */
      return 0;
    else
      btree_rangeiteratered(&anz, tree->root, keyfrom, &k, iter, tree);
  }
  return anz;
}

/*****************************************************************************/
/* Diese Funktion implementiert eine Bereichsabfrage zur Bestimmung des      */
/* Umfanges der Abfragemenge                                                 */
/*****************************************************************************/
/* Parameter:                                                                */
/*   keyfrom - Zeigt auf den niedrigsten gewünschten Schlüssel               */
/*   keyto   - Zeigt auf den höchsten gewünschten Schlüssel                  */
/*   tree    - Zeigt auf den B-Baum                                          */
/* Ergebnis:                                                                 */
/*   Umfang der Abfragemenge                                                 */
/*****************************************************************************/
unsigned int btree_rangecount(void *keyfrom, void *keyto, btree *tree)
{
  unsigned int anz=0;
  key k;
  if(tree->root!=NULL)
  {
    if((*tree->compar)(keyfrom, keyto)>0)
    {
      /* keyto ist kleiner als keyfrom */
      return 0;
    }
    if(btree_getkeylast(&k, tree->root, keyto, tree)==TRUE)
      /* kein Schlüssel gefunden -> leere Menge */
      return 0;
    else
      btree_rangecountred(&anz, tree->root, keyfrom, &k, tree);
  }
  return anz;
}

/*****************************************************************************/
/* Diese Funktion ermittelt den kleinsten im Baum gespeicherten Schlüssel    */
/*****************************************************************************/
/* Parameter:                                                                */
/*   dest - Zeiger auf Speicher, der den Schlüssel aufnehmen soll            */
/*   n    - Zeiger auf Knoten, ab dem gesucht werden soll                    */
/*   tree - Zeiger auf den B-Baum                                            */
/* Ergebnis:                                                                 */
```

```
/*   TRUE - es wurde kein Schlüssel gefunden,                                */
/*          ansonsten FALSE                                                  */
/***************************************************************************/
int btree_getkeylowest(void *dest, NODE n, btree *tree)
{
  NODE t;
  if(n==NULL)
    return TRUE;
  do
  {
    t=n;
    n=node_getlnk(n, 0, tree);
  }
  while(n!=NULL);
  /* Blatt gefunden */
  node_getkey(dest, t, 0, tree);
  return FALSE;
}

/***************************************************************************/
/* Diese Funktion ermittelt den größten im Baum gespeicherten Schlüssel    */
/***************************************************************************/
/* Parameter:                                                              */
/*   dest - Zeiger auf Speicher, der den Schlüssel aufnehmen soll          */
/*   n    - Zeiger auf Knoten, ab dem gesucht werden soll                  */
/*   tree - Zeiger auf den B-Baum                                          */
/* Ergebnis:                                                               */
/*   TRUE       - es wurde kein Schlüssel gefunden,                        */
/*                ansonsten FALSE                                          */
/***************************************************************************/
int btree_getkeyhighest(void *dest, NODE n, btree *tree)
{
  NODE t;
  if(n==NULL)
    return TRUE;
  do
  {
    t=n;
    n=node_getlnk(n, node_getcount(n, tree), tree);
  }
  while(n!=NULL);
  /* Blatt gefunden */
  node_getkey(dest, t, node_getcount(t, tree)-1, tree);
  return FALSE;
}

/***************************************************************************/
/* Diese Funktion führt eine Tiefensuche nach einer bestimmten Tiefe       */
/* ab einem bestimmten Knoten durch und                                    */
/* ruft für jeden Schlüssel dieser Tiefe eine Funktion auf                 */
/***************************************************************************/
/* Parameter:                                                              */
/*   n     - Zeiger auf Knoten, der die Tiefe 0 hat                        */
/*   depth - gesuchte Tiefe                                                */
/*   iter  - Zeiger auf Funktion, die für jeden Schlüssel der gesuchten Tiefe */
/*           aufgerufen wird                                               */
/*   tree  - Zeiger auf den Baum                                           */
/***************************************************************************/
```

```
void btree_iteratenodedepth(NODE n, unsigned int depth,
       void (*iter)(unsigned int depth, NODE node, void *key, btree *tree),
       btree *tree)
{
  /* Dies ist der Einstiegspunkt für die Rekursion */
  btree_iteratedepthrec(n, 0, depth, iter, tree);
}

/*****************************************************************************/
/* Diese Funktion ruft für alle Schlüssel in allen Ebenen des Baumes         */
/* nacheinander eine Funktion auf                                            */
/*****************************************************************************/
/* Parameter:                                                                */
/*   iter  - Zeiger auf Funktion, die für jeden Schlüssel aufgerufen wird    */
/*   tree  - Zeiger auf den Baum                                             */
/*****************************************************************************/
void btree_iteratedepths(
       void (*iter)(unsigned int depth, NODE node, void *key, btree *tree),
       btree *tree)
{
  int h, i=1;
  NODE n=tree->root;
  /* hier wird keine Breitensuche durchgeführt, da die dafür notwendige Queue */
  /* Platz für die Hälfte aller Schlüssel bereitstellen müßte.                */
  /* Deshalb lieber für jede Tiefe eine separate Tiefensuche machen,          */
  for(h=0; h<=tree->height; h++)
    btree_iteratedepth(h, iter, tree);
}
```

```
/*****************************************************************************/
/* Diese Headerdatei beinhaltet alle Prototypen für die Funktionen, die alle  */
/* Implementationen implementieren                                            */
/*****************************************************************************/

/*****************************************************************************/
/* Prototypen                                                                 */
/*****************************************************************************/

/* Diese Funktion initialisiert das verwendete physikalische Medium bzw.      */
/* Betriebsmittel, in dem die Knoten des B-Baumes ihre Repräsentation finden  */
int node_init(btree *tree);
/* Diese Funktion gibt das verwendete physikalische Medium bzw.               */
/* Betriebsmittel, in dem die Knoten des B-Baumes ihre Repräsentation finden, */
/* wieder frei                                                                */
void node_exit(btree *tree);

/* Diese Funktion erstellt eine physikalische Repräsentation eines leeren     */
/* uninitialisierten B-Baum-Knotens                                           */
NODE node_alloc(btree *tree);
/* Diese Funktion erstellt eine physikalische Repräsentation eines mit einem  */
/* Schlüssel vorinitialisierten B-Baum-Knotens                                */
NODE node_calloc(const void *key, NODE lchild, NODE rchild, btree *tree);
/* Diese Funktion löscht die physikalische Repräsentation eines Knoten        */
void node_free(NODE n);

/* Diese Funktion gibt die Anzahl der Schlüssel in einem Knoten zurück        */
unsigned long node_getcount(const NODE n, btree *tree);
/* Diese Funktion gibt einen Zeiger auf ein Kind zurück                       */
NODE node_getlnk(NODE n, unsigned int pos, btree *tree);
/* Diese Funktion gibt einen Schlüssel eines Knoten zurück                    */
int node_getkey(void *dest, const NODE n, unsigned int pos, btree *tree);
/* Diese Funktion setzt den Schlüssel eines Knoten                            */
void node_setkey(void *key, NODE n, unsigned int pos, btree *tree);

/* Diese Funktion fügt einen Schlüssel in einen noch nicht vollen Knoten ein  */
void node_insertkey(NODE n, const void *key, NODE rchild, unsigned int pos,
                    btree *tree);
/* Diese Funktion fügt durch Aufteilen der Schlüssel in 2 übergebene Knoten   */
/* einen Schlüssel in einen vollen Knoten ein                                 */
NODE node_insertkey_split(NODE n, NODE fork, const void *key, NODE rchild,
                          unsigned int pos, btree *tree);
/* Diese Funktion löscht einen Schlüssel aus einem Knoten                     */
void node_delkey(NODE n, unsigned int pos, btree *tree);
/* Diese Funktion rotiert einen Schlüssel über seinen Vaterschlüssel um eine  */
/* Stelle nach links                                                          */
void node_rotateleft(void *key, NODE l, NODE r, btree *tree);
/* Diese Funktion rotiert einen Schlüssel über seinen Vaterschlüssel um eine  */
/* Stelle nach rechts                                                         */
void node_rotateright(void *key, NODE l, NODE r, btree *tree);
/* Diese Funktion vereinigt zwei Knoten mit seinem Vaterschlüssel             */
void node_merge(void *key, NODE l, NODE r, btree *tree);
```

```
#if !defined(_NODEFILE_H)
#define _NODEFILE_H
/*****************************************************************************/
/* Diese Headerdatei beinhaltet alle Deklarationen und Prototypen für den    */
/* Umgang mit Knoten eines B-Baumes als Stream                               */
/*****************************************************************************/

/*****************************************************************************/
/* Diese Struktur beinhaltet nur den Inhaltszähler count und                 */
/* die Zeiger auf die Nachfahren. Dahinter würden die Schlüssel kommen       */
/* Sie wird nur für den einfachen Zugriff auf die wegen ihrer abstrakten     */
/* Schlüssel abstrakt gewordenen Knoten verwendet:                           */
/* casts und Pointerberechnungen entfallen dadurch weitgehend                */
/*****************************************************************************/

typedef struct node_tag {
  unsigned int count;
  unsigned long lnk[1];
} node;

typedef struct tree_param_tag {
  /* Zeigt auf die offene Indexdatei */
  FILE *f;
  /* Gibt die Anzahl der Bytes ab Anfang des Streams an, die für ihn */
  /* reserviert bleiben                                              */
  unsigned int ofs;
} btree_param;

#define NODE unsigned int
#define NULL 0 /* definiert die Bedeutung von NULL für diese Implementierung */

#define LNK_LEN sizeof(unsigned int)

#define OFS_LNK sizeof(unsigned int)
#define OFS_KEY (OFS_LNK+(tree->order*2+1)*LNK_LEN)
#define NODE_LEN (OFS_KEY+tree->order*2*tree->keylen)

#define FILE_OFS (((btree_param *)(tree->param))->ofs+sizeof(btree))
#define STREAM ((btree_param *)(tree->param))->f

#endif
```

```c
/*****************************************************************************/
/* Diese Quelldatei beinhaltet alle Funktionen zum Umgang mit den Knoten     */
/* eines B-Baumes.                                                           */
/* Somit kann durch Ändern/Auswechseln dieser Implementierung die            */
/* physikalische Repräsentation des Baumes geändert werden                   */
/*                                                                           */
/* Diese Implementierung benutzt einen Stream                                */
/* Besonderheiten:                                                           */
/*   Hinter tree->param wird eine PARAM-Struktur erwartet.                   */
/*   Diese beinhaltet eine FILE-Struktur, über die binär gelesen und         */
/*   geschrieben werden kann                                                 */
/*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "nodefile.h"
#include "btree.h"

/*****************************************************************************/
/* Diese Funktion initialisiert das verwendete physikalische Medium bzw.     */
/* Betriebsmittel, in dem die Knoten des B-Baumes ihre Repräsentation finden */
/*****************************************************************************/
/* Parameter:                                                                */
/*   tree - Zeiger auf den B-Baum                                            */
/* Ergebnis:                                                                 */
/*   0 bei erfolgreicher Initialisierung                                     */
/*****************************************************************************/
int node_init(btree *tree)
{
  void *t, *c;
#if defined(_DEBUG)
  fprintf(stderr, "init\n");
#endif
  /* Betriebsmittel Datei muß speziell initialisiert zu werden */
  fseek(STREAM, 0, SEEK_END);
  if(ftell(STREAM)==((btree_param *)(tree->param))->ofs)
  {
    fwrite(tree, sizeof(btree), 1, STREAM);
  }
  fseek(STREAM, ((btree_param *)(tree->param))->ofs, SEEK_SET);
  t=tree->param;
  c=tree->compar;
  fread(tree, sizeof(btree), 1, STREAM);
  tree->compar=c;
  tree->param=t;
  return 0;
}

/*****************************************************************************/
/* Diese Funktion gibt das verwendete physikalische Medium bzw.              */
/* Betriebsmittel, in dem die Knoten des B-Baumes ihre Repräsentation finden, */
/* wieder frei                                                               */
/*****************************************************************************/
/* Parameter:                                                                */
/*   tree - Zeiger auf den B-Baum                                            */
/*****************************************************************************/
void node_exit(btree *tree)
```

```
{
#if defined(_DEBUG)
  fprintf(stderr, "exit\n");
#endif
  /* Betriebsmittel Datei muß speziell geschlossen werden */
  fseek(STREAM, ((btree_param *)(tree->param))->ofs, SEEK_SET);
  fwrite(tree, sizeof(btree), 1, STREAM);
}

/*****************************************************************************/
/* Diese Funktion erstellt eine physikalische Repräsentation eines leeren    */
/* uninitialisierten B-Baum-Knotens                                          */
/*****************************************************************************/
/* Parameter:                                                                */
/*   tree - Zeiger auf den B-Baum                                            */
/* Ergebnis:                                                                 */
/*   abstrakter Zeiger auf neuen Knoten, sonst NULL                          */
/*****************************************************************************/
NODE node_alloc(btree *tree)
{
  node *n;
  long pos;
#if defined(_DEBUG)
  fprintf(stderr, "alloc\n");
#endif
  /* Datei auf Ende positionieren */
  fseek(STREAM, 0, SEEK_END);
  pos=ftell(STREAM);
  /* Speicherrepräsentation erstellen */
  n=(node *)malloc(NODE_LEN);
  n->count=0;
  fwrite(n, NODE_LEN, 1, STREAM);
  free(n);
  return (pos-FILE_OFS)/NODE_LEN+1;
}

/*****************************************************************************/
/* Diese Funktion erstellt eine physikalische Repräsentation eines mit einem */
/* Schlüssel vorinitialisierten B-Baum-Knotens                               */
/*****************************************************************************/
/* Parameter:                                                                */
/*   key    - Zeiger auf neuen Schlüssel                                     */
/*   lchild - Zeiger auf das linke Kind des Schlüssels                       */
/*   rchild - Zeiger auf das rechte Kind des Schlüssels                      */
/*   tree   - Zeiger auf den B-Baum                                          */
/* Ergebnis:                                                                 */
/*   abstrakter Zeiger auf neuen Knoten, sonst NULL                          */
/*****************************************************************************/
NODE node_calloc(const void *key, NODE lchild, NODE rchild, btree *tree)
{
  node *n;
  long pos;
  /* Datei auf Ende positionieren */
  fseek(STREAM, 0, SEEK_END);
  pos=ftell(STREAM);
  /* Speicherrepräsentation erstellen */
  n=(node *)malloc(NODE_LEN);
  n->count=1;
```

```
  n->lnk[0]=lchild;
  n->lnk[1]=rchild;
  /* schreiben */
  fwrite(n, OFS_KEY, 1, STREAM);
  fwrite(key, tree->keylen, 1, STREAM);
  fwrite((char *)n+OFS_KEY+tree->keylen, NODE_LEN-OFS_KEY-tree->keylen, 1,
STREAM);
  free(n);
#if defined(_DEBUG)
  fprintf(stderr, "calloc: %i\n", pos);
#endif
  return (pos-FILE_OFS)/NODE_LEN+1;
}

/*****************************************************************************/
/* Diese Funktion löscht die physikalische Repräsentation eines Knoten       */
/*****************************************************************************/
/* Parameter:                                                                */
/*   n - Zeiger auf Knoten, der gelöscht werden soll                         */
/*****************************************************************************/
void node_free(NODE n)
{
#if defined(_DEBUG)
  fprintf(stderr, "free: %u\n", n);
#endif
  /* Keine Freigabe -> Datei-Fragmentation */
  /* Spätere Implementierung einer Datei-Bitmap, First-(/Best-)Fit-Routinen */
  /* bleibt möglich */
}

/*****************************************************************************/
/* Diese Funktion gibt die Anzahl der Schlüssel in einem Knoten zurück       */
/*****************************************************************************/
/* Parameter:                                                                */
/*   n - Zeiger auf den Knoten                                               */
/* Ergebnis:                                                                 */
/*   Anzahl der Schlüssel im Knoten                                          */
/*****************************************************************************/
NODE node_getcount(const NODE n, btree *tree)
{
  NODE cnt;
  /* Datei auf Knoten positionieren */
  fseek(STREAM, FILE_OFS+(n-1)*NODE_LEN, SEEK_SET);
  fread(&cnt, sizeof(cnt), 1, STREAM);
#if defined(_DEBUG)
/*  fprintf(stderr, "getcount: %u hat %i\n", n, cnt);*/
#endif
  return cnt;
}

/*****************************************************************************/
/* Diese Funktion setzt die Anzahl der Schlüssel in einem Knoten             */
/*****************************************************************************/
/* Parameter:                                                                */
/*   n   - Zeiger auf den Knoten                                             */
/*   count - Anzahl der Schlüssel                                            */
/*****************************************************************************/
static void node_setcount(NODE n, unsigned int count, btree *tree)
```

```
{
  /* Datei auf Knoten positionieren */
  fseek(STREAM, FILE_OFS+(n-1)*NODE_LEN, SEEK_SET);
  fwrite(&count, sizeof(count), 1, STREAM);
#if defined(_DEBUG)
  fprintf(stderr, "setcount: %u zu %u\n", n, count);
#endif
}

/*****************************************************************************/
/* Diese Funktion gibt einen Zeiger auf ein Kind zurück                      */
/*****************************************************************************/
/* Parameter:                                                                */
/*   n   - Zeiger auf Knoten, der auf das Kind verweist                      */
/*   pos - Position des Verweises auf das Kind                               */
/*****************************************************************************/
/* Ergebnis:                                                                 */
/*   Zeiger auf Kindknoten                                                   */
/*****************************************************************************/
NODE node_getlnk(NODE n, unsigned int pos, btree *tree)
{
  NODE lnk;
  /* Datei auf Link des Knoten positionieren */
  fseek(STREAM, FILE_OFS+(n-1)*NODE_LEN+OFS_LNK+pos*LNK_LEN, SEEK_SET);
  fread(&lnk, sizeof(lnk), 1, STREAM);
#if defined(_DEBUG)
/*  fprintf(stderr, "getlnk: %u an Pos %u steht %u\n", n, pos, lnk);*/
#endif
  return lnk;
}

/*****************************************************************************/
/* Diese Funktion setzt den Zeiger auf ein Kind                              */
/*****************************************************************************/
/* Parameter:                                                                */
/*   n   - Zeiger auf Knoten, der den Zeiger beinhaltet                      */
/*   pos - Position des Verweises auf das Kind                               */
/*   lnk - Zeiger auf Kind                                                   */
/*****************************************************************************/
static void node_setlnk(NODE n, unsigned int pos, NODE lnk, btree *tree)
{
#if defined(_DEBUG)
  fprintf(stderr, "setlnk: %u an Pos %u zu %u\n", n, pos, lnk);
#endif
  /* Datei auf Link des Knoten positionieren */
  fseek(STREAM, FILE_OFS+n*NODE_LEN+OFS_LNK+pos*LNK_LEN, SEEK_SET);
  fwrite(&lnk, sizeof(lnk), 1, STREAM);
}

/*****************************************************************************/
/* Diese Funktion gibt einen Schlüssel eines Knoten zurück                   */
/*****************************************************************************/
/* Parameter:                                                                */
/*   dest - Zeiger auf Speicher, der den Schlüssel aufnehmen soll            */
/*   n    - Zeiger auf Knoten, der den Schlüssel besitzt                     */
/*   pos  - Position des Schlüssels                                          */
/*   tree - Zeiger auf den B-Baum                                            */
/* Ergebnis:                                                                 */
```

```
/*   0 - Schlüssel wurde erfolgreich gelesen                                */
/*************************************************************************/
int node_getkey(void *dest, const NODE n, unsigned int pos, btree *tree)
{
  /* Datei auf Schlüssel des Knoten positionieren */
  fseek(STREAM, FILE_OFS+(n-1)*NODE_LEN+OFS_KEY+pos*tree->keylen, SEEK_SET);
  if(fread(dest, tree->keylen, 1, STREAM)!=1)
    /* Schlüssel nicht vorhanden */
    return 1;
#if defined(_DEBUG)
/*  fprintf(stderr, "getkey: %u an Pos %u steht %u\n", n, pos, *(unsigned
int*)k);*/
#endif
  return 0;
}

/*************************************************************************/
/* Diese Funktion setzt den Schlüssel eines Knoten                       */
/*************************************************************************/
/* Parameter:                                                            */
/*   key  - Zeiger auf den zu setzenden Schlüssel                        */
/*   n    - Zeiger auf Knoten, der den zu überschreibenden Schlüssel besitzt */
/*   pos  - Position des zu überschreibenden Schlüssels                  */
/*   tree - Zeiger auf den B-Baum                                        */
/*************************************************************************/
void node_setkey(void *key, NODE n, unsigned int pos, btree *tree)
{
#if defined(_DEBUG)
/*  fprintf(stderr, "setkey: %u an Pos %u zu %u\n", n, pos, *(int*)key);*/
#endif
  /* Datei auf Schlüssel des Knoten positionieren */
  fseek(STREAM, FILE_OFS+(n-1)*NODE_LEN+OFS_KEY+pos*tree->keylen, SEEK_SET);
  fwrite(key, tree->keylen, 1, STREAM);
}

/*************************************************************************/
/* Diese Funktion fügt einen Schlüssel in einen noch nicht vollen Knoten ein */
/*************************************************************************/
/* Parameter:                                                            */
/*   n      - Zeiger auf den Knoten, in den eingefügt werden soll        */
/*   key    - Zeiger auf den Schlüssel, der eingefügt werden soll        */
/*   rchild - Zeiger auf das rechte Kind des einzufügenden Schlüssels    */
/*   pos    - Position, an der Schlüssel eingefügt werden soll           */
/*   tree   - Zeiger auf den B-Baum                                      */
/*************************************************************************/
void node_insertkey(NODE n, const void *key, NODE rchild, unsigned int pos,
btree *tree)
{
  void *pkey, *plnk;
  unsigned int cnt;
#if defined(_DEBUG)
  fprintf(stderr, "insertkey: %i\n", *(int*)key);
#endif
  /* Schlüssel paßt in Knoten: Lediglich umordnen der Schlüssel und Einfügen */
  cnt=node_getcount(n, tree);
  node_setcount(n, cnt+1, tree);
  n--;
  if(pos<cnt)
```

```
  {
    pkey=malloc((cnt-pos)*tree->keylen);
    plnk=malloc((cnt-pos)*LNK_LEN);
    fseek(STREAM, FILE_OFS+n*NODE_LEN+OFS_LNK+(pos+1)*LNK_LEN, SEEK_SET);
    fread(plnk, LNK_LEN, cnt-pos, STREAM);
    fseek(STREAM, FILE_OFS+n*NODE_LEN+OFS_KEY+pos*tree->keylen, SEEK_SET);
    fread(pkey, tree->keylen, cnt-pos, STREAM);
  }
  fseek(STREAM, FILE_OFS+n*NODE_LEN+OFS_KEY+pos*tree->keylen, SEEK_SET);
  fwrite(key, tree->keylen, 1, STREAM);
  if(pos<cnt)
  {
    fwrite(pkey, tree->keylen, cnt-pos, STREAM);
  }
  fseek(STREAM, FILE_OFS+n*NODE_LEN+OFS_LNK+(pos+1)*LNK_LEN, SEEK_SET);
  fwrite(&rchild, LNK_LEN, 1, STREAM);
  if(pos<cnt)
  {
    fwrite(plnk, LNK_LEN, cnt-pos, STREAM);
    free(pkey);
    free(plnk);
  }
}

/*****************************************************************************/
/* Diese Funktion fügt durch Aufteilen der Schlüssel in 2 übergebene Knoten  */
/* einen Schlüssel in einen vollen Knoten ein                                */
/*****************************************************************************/
/* Parameter:                                                                */
/*   n      - Zeiger auf den Knoten, in den eingefügt werden soll            */
/*   fork   - Zeiger auf eine leere Instanz des 2. Knoten                    */
/*   key    - Zeiger auf den Schlüssel, der eingefügt werden soll            */
/*   rchild - Zeiger auf das rechte Kind des einzufügenden Schlüssels        */
/*   pos    - Position, an der Schlüssel eingefügt werden soll               */
/*   tree   - Zeiger auf den B-Baum                                          */
/* Ergebnis:                                                                 */
/*   Zeiger auf den nun nicht mehr leeren 2. Knoten fork                     */
/*****************************************************************************/
NODE node_insertkey_split(NODE n, NODE fork,
                          const void *key, NODE rchild,
                                unsigned int pos, btree *tree)
{
  char *bufkey, *buflnk;
#if defined(_DEBUG)
  fprintf(stderr, "insertkey_split: %i\n", *(int*)key);
#endif
  /* Blatt ist schon voll, Splitten und Aufsteiger zurückliefern! */
  node_setcount(n, tree->order, tree);
  n--;
  if(pos>tree->order)
  {
    /* Der einzufügende Schlüssel landet in fork */
    bufkey=malloc((tree->order-1)*tree->keylen);
    buflnk=malloc(tree->order*LNK_LEN);

    fseek(STREAM, FILE_OFS+n*NODE_LEN+OFS_LNK+(tree->order+1)*LNK_LEN,
SEEK_SET);
    fread(buflnk, LNK_LEN, tree->order, STREAM);
```

```
    fseek(STREAM, FILE_OFS+n*NODE_LEN+OFS_KEY+(tree->order+1)*tree->keylen,
SEEK_SET);
    fread(bufkey, tree->keylen, tree->order-1, STREAM);

    node_setcount(fork, tree->order, tree);
    fork--;
    fseek(STREAM, FILE_OFS+fork*NODE_LEN+OFS_LNK, SEEK_SET);
    fwrite(buflnk, LNK_LEN, pos-tree->order, STREAM);
    fwrite(&rchild, LNK_LEN, 1, STREAM);
    fwrite(buflnk+(pos-tree->order)*LNK_LEN, LNK_LEN, 2*tree->order-pos,
STREAM);

    fseek(STREAM, FILE_OFS+fork*NODE_LEN+OFS_KEY, SEEK_SET);
    fwrite(bufkey, tree->keylen, pos-tree->order-1, STREAM);
    fwrite(key, tree->keylen, 1, STREAM);
    fwrite(bufkey+(pos-tree->order-1)*tree->keylen, tree->keylen,
2*tree->order-pos, STREAM);

    free(bufkey);
    free(buflnk);
  }
  else
  {
    /* Der einzufügende Schlüssel landet in n */
    bufkey=malloc((2*tree->order-pos)*tree->keylen);
    buflnk=malloc((2*tree->order-pos)*LNK_LEN);

    fseek(STREAM, FILE_OFS+n*NODE_LEN+OFS_LNK+(pos+1)*LNK_LEN, SEEK_SET);
    fread(buflnk, LNK_LEN, 2*tree->order-pos, STREAM);
    fseek(STREAM, FILE_OFS+n*NODE_LEN+OFS_KEY+pos*tree->keylen, SEEK_SET);
    fread(bufkey, tree->keylen, 2*tree->order-pos, STREAM);

    node_setcount(fork, tree->order, tree);
    fork--;
    fseek(STREAM, FILE_OFS+fork*NODE_LEN+OFS_LNK, SEEK_SET);
    if(pos==tree->order)
      fwrite(&rchild, LNK_LEN, 1, STREAM);
    else
      fwrite(buflnk+(tree->order-pos-1)*LNK_LEN, LNK_LEN, 1, STREAM);
    fwrite(buflnk+(tree->order-pos)*LNK_LEN, LNK_LEN, tree->order, STREAM);
    fseek(STREAM, FILE_OFS+fork*NODE_LEN+OFS_KEY, SEEK_SET);
    fwrite(bufkey+(tree->order-pos)*tree->keylen, tree->keylen, tree->order,
STREAM);

    fseek(STREAM, FILE_OFS+n*NODE_LEN+OFS_LNK+(pos+1)*LNK_LEN, SEEK_SET);
    fwrite(&rchild, LNK_LEN, 1, STREAM);
    if(pos!=tree->order)
      fwrite(buflnk, LNK_LEN, tree->order-pos-1, STREAM);
    fseek(STREAM, FILE_OFS+n*NODE_LEN+OFS_KEY+pos*tree->keylen, SEEK_SET);
    fwrite(key, tree->keylen, 1, STREAM);
    fwrite(bufkey, tree->keylen, tree->order-pos, STREAM);

    free(bufkey);
    free(buflnk);
  }
  return fork+1;
}
```

```c
/*****************************************************************************/
/* Diese Funktion löscht einen Schlüssel aus einem Knoten                    */
/*****************************************************************************/
/* Parameter:                                                                */
/*   n      - Zeiger auf den Knoten, aus dem gelöscht werden soll            */
/*   pos    - Position, an der der zu löschende Schlüssel steht              */
/*   tree   - Zeiger auf den B-Baum                                          */
/*****************************************************************************/
void node_delkey(NODE n, unsigned int pos, btree *tree)
{
  char *bufkey, *buflnk;
  unsigned int cnt;
#if defined(_DEBUG)
  int *k;
  k=malloc(tree->keylen);
  node_getkey(k, n, pos, tree);
  fprintf(stderr, "delkey: %i löschen\n", *k);
  free(k);
#endif
  cnt=node_getcount(n, tree);
  node_setcount(n, cnt-1, tree);
  n--;
  if(pos<cnt-1)
  {
    bufkey=malloc((cnt-pos)*tree->keylen);
    buflnk=malloc((cnt-pos)*LNK_LEN);

    fseek(STREAM, FILE_OFS+n*NODE_LEN+OFS_LNK+(pos+1)*tree->keylen, SEEK_SET);
    fread(buflnk, LNK_LEN, cnt-pos, STREAM);
    fseek(STREAM, FILE_OFS+n*NODE_LEN+OFS_KEY+pos*tree->keylen, SEEK_SET);
    fread(bufkey, tree->keylen, cnt-pos, STREAM);

    fseek(STREAM, FILE_OFS+n*NODE_LEN+OFS_LNK+(pos+1)*tree->keylen, SEEK_SET);
    fwrite(buflnk+LNK_LEN, LNK_LEN, cnt-pos-1, STREAM);
    fwrite(buflnk, LNK_LEN, 1, STREAM);
    fseek(STREAM, FILE_OFS+n*NODE_LEN+OFS_KEY+pos*tree->keylen, SEEK_SET);
    fwrite(bufkey+tree->keylen, tree->keylen, cnt-pos-1, STREAM);
    fwrite(bufkey, tree->keylen, 1, STREAM);

    free(bufkey);
    free(buflnk);
  }
}

/*****************************************************************************/
/* Diese Funktion rotiert einen Schlüssel über seinen Vaterschlüssel um eine  */
/* Stelle nach links                                                         */
/*****************************************************************************/
/* Parameter:                                                                */
/*   key  - Zeiger auf den Vaterschlüssel                                    */
/*   l    - Zeiger auf den linken Knoten                                     */
/*   r    - Zeiger auf den rechten Knoten                                    */
/*   tree - Zeiger auf den B-Baum                                            */
/*****************************************************************************/
void node_rotateleft(void *key, NODE l, NODE r, btree *tree)
{
  char *bufkey, *buflnk;
  unsigned int rcnt;
```

```
#if defined(_DEBUG)
  fprintf(stderr, "rotateleft: %i\n", *(int*)key);
#endif
  rcnt=node_getcount(r, tree);
  node_setcount(r, rcnt-1, tree);
  r--;
  bufkey=malloc(rcnt*tree->keylen);
  buflnk=malloc((rcnt+1)*LNK_LEN);

  fseek(STREAM, FILE_OFS+r*NODE_LEN+OFS_LNK, SEEK_SET);
  fread(buflnk, LNK_LEN, rcnt+1, STREAM);
  fseek(STREAM, FILE_OFS+r*NODE_LEN+OFS_KEY, SEEK_SET);
  fread(bufkey, tree->keylen, rcnt, STREAM);

  fseek(STREAM, FILE_OFS+r*NODE_LEN+OFS_LNK, SEEK_SET);
  fwrite(buflnk+LNK_LEN, LNK_LEN, rcnt, STREAM);
  fseek(STREAM, FILE_OFS+r*NODE_LEN+OFS_KEY, SEEK_SET);
  fwrite(bufkey+tree->keylen, tree->keylen, rcnt-1, STREAM);

  node_setcount(l, tree->order, tree);
  l--;
  fseek(STREAM, FILE_OFS+l*NODE_LEN+OFS_LNK+tree->order*LNK_LEN, SEEK_SET);
  fwrite(buflnk, LNK_LEN, 1, STREAM);
  fseek(STREAM, FILE_OFS+l*NODE_LEN+OFS_KEY+(tree->order-1)*tree->keylen,
SEEK_SET);
  fwrite(key, tree->keylen, 1, STREAM);

  memcpy(key, bufkey, tree->keylen);

  free(bufkey);
  free(buflnk);
}

/*****************************************************************************/
/* Diese Funktion rotiert einen Schlüssel über seinen Vaterschlüssel um eine  */
/* Stelle nach rechts                                                        */
/*****************************************************************************/
/* Parameter:                                                                */
/*   key  - Zeiger auf den Vaterschlüssel                                    */
/*   l    - Zeiger auf den linken Knoten                                     */
/*   r    - Zeiger auf den rechten Knoten                                    */
/*   tree - Zeiger auf den B-Baum                                            */
/*****************************************************************************/
void node_rotateright(void *key, NODE l, NODE r, btree *tree)
{
  char *bufkey, *buflnk;
  unsigned int lcnt;
#if defined(_DEBUG)
  fprintf(stderr, "rotateright: %i\n", *(int*)key);
#endif
  lcnt=node_getcount(l, tree);
  node_setcount(l, lcnt-1, tree);
  l--;
  bufkey=malloc(tree->order*tree->keylen);
  buflnk=malloc((tree->order+1)*LNK_LEN);

  fseek(STREAM, FILE_OFS+l*NODE_LEN+OFS_LNK+lcnt*tree->keylen, SEEK_SET);
  fread(buflnk, LNK_LEN, 1, STREAM);
```

```
  fseek(STREAM, FILE_OFS+l*NODE_LEN+OFS_KEY+(lcnt-1)*tree->keylen, SEEK_SET);
  fread(bufkey, tree->keylen, 1, STREAM);

  node_setcount(r, tree->order, tree);
  r--;
  fseek(STREAM, FILE_OFS+r*NODE_LEN+OFS_LNK, SEEK_SET);
  fread(buflnk+LNK_LEN, LNK_LEN, tree->order, STREAM);
  fseek(STREAM, FILE_OFS+r*NODE_LEN+OFS_KEY, SEEK_SET);
  fread(bufkey+tree->keylen, tree->keylen, tree->order-1, STREAM);

  fseek(STREAM, FILE_OFS+r*NODE_LEN+OFS_LNK, SEEK_SET);
  fwrite(buflnk, LNK_LEN, tree->order+1, STREAM);
  fseek(STREAM, FILE_OFS+r*NODE_LEN+OFS_KEY, SEEK_SET);
  fwrite(key, tree->keylen, 1, STREAM);
  fwrite(bufkey+tree->keylen, tree->keylen, tree->order-1, STREAM);

  memcpy(key, bufkey, tree->keylen);

  free(bufkey);
  free(buflnk);
}

/*****************************************************************************/
/* Diese Funktion vereinigt zwei Knoten mit seinem Vaterschlüssel             */
/*****************************************************************************/
/* Parameter:                                                                */
/*   key  - Zeiger auf den Vaterschlüssel                                    */
/*   l    - Zeiger auf den linken Knoten                                     */
/*   r    - Zeiger auf den rechten Knoten                                    */
/*   tree - Zeiger auf den B-Baum                                            */
/*****************************************************************************/
void node_merge(void *key, NODE l, NODE r, btree *tree)
{
  char *bufkey, *buflnk;
  unsigned int lcnt, rcnt;
#if defined(_DEBUG)
  fprintf(stderr, "merge: %i und rechtes Kind wird zu linkem\n", *(int*)key);
#endif
  rcnt=node_getcount(r, tree);
  r--;
  bufkey=malloc(rcnt*tree->keylen);
  buflnk=malloc((rcnt+1)*LNK_LEN);

  fseek(STREAM, FILE_OFS+r*NODE_LEN+OFS_LNK, SEEK_SET);
  fread(buflnk, LNK_LEN, rcnt+1, STREAM);
  fseek(STREAM, FILE_OFS+r*NODE_LEN+OFS_KEY, SEEK_SET);
  fread(bufkey, tree->keylen, rcnt, STREAM);

  lcnt=node_getcount(l, tree);
  node_setcount(l, 2*tree->order, tree);
  l--;
  fseek(STREAM, FILE_OFS+l*NODE_LEN+OFS_LNK+(lcnt+1)*LNK_LEN, SEEK_SET);
  fwrite(buflnk, LNK_LEN, rcnt+1, STREAM);
  fseek(STREAM, FILE_OFS+l*NODE_LEN+OFS_KEY+lcnt*tree->keylen, SEEK_SET);
  fwrite(key, tree->keylen, 1, STREAM);
  fwrite(bufkey, tree->keylen, rcnt, STREAM);

  free(bufkey);
```

```
  free(buflnk);
}
```

```
#if !defined(_NODEMEM_H)
#define _NODEMEM_H
/*****************************************************************************/
/* Diese Headerdatei beinhaltet alle Deklarationen und Prototypen für den    */
/* Umgang mit Knoten eines B-Baumes als dynamische Speicherallokierung       */
/*****************************************************************************/

/*****************************************************************************/
/* Diese Struktur beinhaltet nur den Inhaltszähler count und                 */
/* die Zeiger auf die Nachfahren. Dahinter würden die Schlüssel kommen       */
/* Sie wird nur für den einfachen Zugriff auf die wegen ihrer abstrakten     */
/* Schlüssel abstrakt gewordenen Knoten verwendet:                           */
/* casts und Pointerberechnungen entfallen dadurch weitgehend                */
/*****************************************************************************/
typedef struct node_tag {
  unsigned int count;
  struct node_tag *lnk[1];
} node;

#define NODE node *
/* definiert die Bedeutung von NULL für diese Implementierung                */
/*#define NULL NULL*/

#define LNK_LEN sizeof(void *)

#define OFS_LNK sizeof(unsigned int)
#define OFS_KEY (OFS_LNK+(tree->order*2+1)*LNK_LEN)
#define NODE_LEN (OFS_KEY+tree->order*2*tree->keylen)

#endif
```

```c
/*****************************************************************************/
/* Diese Quelldatei beinhaltet alle Funktionen zum Umgang mit den Knoten     */
/* eines B-Baumes.                                                           */
/* Somit kann durch Ändern/Auswechseln dieser Implementierung die            */
/* physikalische Repräsentation des Baumes geändert werden                   */
/*                                                                           */
/* Diese Implementierung benutzt dynamische Speicherallokierung              */
/*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "nodemem.h"
#include "btree.h"

/*****************************************************************************/
/* Diese Funktion initialisiert das verwendete physikalische Medium bzw.     */
/* Betriebsmittel, in dem die Knoten des B-Baumes ihre Repräsentation finden */
/*****************************************************************************/
/* Parameter:                                                                */
/*   tree - Zeiger auf den B-Baum                                            */
/* Ergebnis:                                                                 */
/*   0 bei erfolgreicher Initialisierung                                     */
/*****************************************************************************/
int node_init(btree *tree)
{
  /* Betriebsmittel Speicher braucht nicht initialisiert zu werden */
  return 0;
}

/*****************************************************************************/
/* Diese Funktion gibt das verwendete physikalische Medium bzw.              */
/* Betriebsmittel, in dem die Knoten des B-Baumes ihre Repräsentation finden, */
/* wieder frei                                                               */
/*****************************************************************************/
/* Parameter:                                                                */
/*   tree - Zeiger auf den B-Baum                                            */
/*****************************************************************************/
void node_exit(btree *tree)
{
  /* Betriebsmittel Speicher braucht nicht freigegeben zu werden */
}

/*****************************************************************************/
/* Diese Funktion erstellt eine physikalische Repräsentation eines leeren    */
/* uninitialisierten B-Baum-Knotens                                          */
/*****************************************************************************/
/* Parameter:                                                                */
/*   tree - Zeiger auf den B-Baum                                            */
/* Ergebnis:                                                                 */
/*   abstrakter Zeiger auf neuen Knoten, sonst NULL                          */
/*****************************************************************************/
NODE node_alloc(btree *tree)
{
  node *n;
  n=(node *)malloc(NODE_LEN);
  n->count=0;
  return n;
```

```
}

/*****************************************************************************/
/* Diese Funktion erstellt eine physikalische Repräsentation eines mit einem  */
/* Schlüssel vorinitialisierten B-Baum-Knotens                                */
/*****************************************************************************/
/* Parameter:                                                                 */
/*   key    - Zeiger auf neuen Schlüssel                                      */
/*   lchild - Zeiger auf das linke Kind des Schlüssels                        */
/*   rchild - Zeiger auf das rechte Kind des Schlüssels                       */
/*   tree   - Zeiger auf den B-Baum                                           */
/* Ergebnis:                                                                  */
/*   abstrakter Zeiger auf neuen Knoten, sonst NULL                           */
/*****************************************************************************/
NODE node_calloc(const void *key, NODE lchild, NODE rchild, btree *tree)
{
  node *n;
  n=(node *)malloc(NODE_LEN);
  if(n!=NULL)
  {
    n->count=1;
    n->lnk[0]=lchild;
    n->lnk[1]=rchild;
    memcpy((char *)n+OFS_KEY,
           key,
           tree->keylen);
  }
  return n;
}

/*****************************************************************************/
/* Diese Funktion löscht die physikalische Repräsentation eines Knoten       */
/*****************************************************************************/
/* Parameter:                                                                 */
/*   n - Zeiger auf Knoten, der gelöscht werden soll                          */
/*****************************************************************************/
void node_free(NODE n)
{
  free(n);
}

/*****************************************************************************/
/* Diese Funktion gibt die Anzahl der Schlüssel in einem Knoten zurück       */
/*****************************************************************************/
/* Parameter:                                                                 */
/*   n - Zeiger auf den Knoten                                                */
/* Ergebnis:                                                                  */
/*   Anzahl der Schlüssel im Knoten                                           */
/*****************************************************************************/
unsigned long node_getcount(const NODE n, btree *tree)
{
  return n->count;
}

/*****************************************************************************/
/* Diese Funktion gibt einen Zeiger auf ein Kind zurück                      */
/*****************************************************************************/
/* Parameter:                                                                 */
```

```
/*   n   - Zeiger auf Knoten, der auf das Kind verweist                    */
/*   pos - Position des Verweises auf das Kind                             */
/*************************************************************************/
/* Ergebnis:                                                             */
/*   Zeiger auf Kindknoten                                               */
/*************************************************************************/
NODE node_getlnk(NODE n, unsigned int pos, btree *tree)
{
  return n->lnk[pos];
}

/*************************************************************************/
/* Diese Funktion gibt einen Schlüssel eines Knoten zurück               */
/*************************************************************************/
/* Parameter:                                                            */
/*   dest - Zeiger auf Speicher, der den Schlüssel aufnehmen soll        */
/*   n    - Zeiger auf Knoten, der den Schlüssel besitzt                 */
/*   pos  - Position des Schlüssels                                      */
/*   tree - Zeiger auf den B-Baum                                        */
/* Ergebnis:                                                             */
/*   0 - Schlüssel wurde erfolgreich gelesen                             */
/*************************************************************************/
int node_getkey(void *dest, const NODE n, unsigned int pos, btree *tree)
{
  memcpy(dest, (char *)n+OFS_KEY+pos*tree->keylen, tree->keylen);
  return 0;
}

/*************************************************************************/
/* Diese Funktion setzt den Schlüssel eines Knoten                       */
/*************************************************************************/
/* Parameter:                                                            */
/*   key  - Zeiger auf den zu setzenden Schlüssel                        */
/*   n    - Zeiger auf Knoten, der den zu überschreibenden Schlüssel besitzt  */
/*   pos  - Position des zu überschreibenden Schlüssels                  */
/*   tree - Zeiger auf den B-Baum                                        */
/*************************************************************************/
void node_setkey(void *key, NODE n, unsigned int pos, btree *tree)
{
#if defined(_DEBUG)
  void *k;
  k=malloc(tree->keylen);
  node_getkey(k, n, pos, tree);
  fprintf(stderr, "setkey: %u zu %u\n", *(int*)k,
                                        *(int*)key);
  free(k);
#endif
  memcpy((char *)n+OFS_KEY+pos*tree->keylen,
         key,
         tree->keylen);
}

/*************************************************************************/
/* Diese Funktion fügt einen Schlüssel in einen noch nicht vollen Knoten ein  */
/*************************************************************************/
/* Parameter:                                                            */
/*   n      - Zeiger auf den Knoten, in den eingefügt werden soll        */
/*   key    - Zeiger auf den Schlüssel, der eingefügt werden soll        */
```

```
/*   rchild - Zeiger auf das rechte Kind des einzufügenden Schlüssels         */
/*   pos    - Position, an der Schlüssel eingefügt werden soll                */
/*   tree   - Zeiger auf den B-Baum                                           */
/****************************************************************************/
void node_insertkey(NODE n, const void *key, NODE rchild, unsigned int pos,
btree *tree)
{
#if defined(_DEBUG)
  fprintf(stderr, "insertkey: %i\n", *(int*)key);
#endif
  /* Schlüssel paßt in Knoten: Lediglich umordnen der Schlüssel und Einfügen */
  if(pos<n->count)
  {
    memmove((char *)n+OFS_KEY+(pos+1)*tree->keylen,
            (char *)n+OFS_KEY+pos*tree->keylen,
            (n->count-pos)*tree->keylen);
    memmove(&n->lnk[pos+2],
            &n->lnk[pos+1],
            (n->count-pos)*LNK_LEN);
  }
  memcpy((char *)n+OFS_KEY+pos*tree->keylen,
         key,
         tree->keylen);
  n->lnk[pos+1]=rchild;
  n->count++;
}

/****************************************************************************/
/* Diese Funktion fügt durch Aufteilen der Schlüssel in 2 übergebene Knoten */
/* einen Schlüssel in einen vollen Knoten ein                               */
/****************************************************************************/
/* Parameter:                                                               */
/*   n      - Zeiger auf den Knoten, in den eingefügt werden soll           */
/*   fork   - Zeiger auf eine leere Instanz des 2. Knoten                   */
/*   key    - Zeiger auf den Schlüssel, der eingefügt werden soll           */
/*   rchild - Zeiger auf das rechte Kind des einzufügenden Schlüssels       */
/*   pos    - Position, an der Schlüssel eingefügt werden soll              */
/*   tree   - Zeiger auf den B-Baum                                         */
/* Ergebnis:                                                                */
/*   Zeiger auf den nun nicht mehr leeren 2. Knoten fork                    */
/****************************************************************************/
NODE node_insertkey_split(NODE n, NODE fork, const void *key, NODE rchild,
                          unsigned int pos, btree *tree)
{
#if defined(_DEBUG)
  fprintf(stderr, "insertkey_split: %i\n", *(int*)key);
#endif
  /* Blatt ist schon voll, Splitten und Aufsteiger zurückliefern! */
  fork->count=tree->order;
  if(pos>tree->order)
  {
    /* Der einzufügende Schlüssel landet in fork */
    memcpy((char *)fork+OFS_KEY,
           (char *)n+OFS_KEY+(tree->order+1)*tree->keylen,
           (pos-tree->order-1)*tree->keylen);
    memcpy(&fork->lnk[0],
           &n->lnk[tree->order+1],
           (pos-tree->order)*LNK_LEN);
```

```
    memcpy((char *)fork+OFS_KEY+(pos-tree->order-1)*tree->keylen,
           key,
           tree->keylen);
    fork->lnk[pos-tree->order]=rchild;
    memcpy((char *)fork+OFS_KEY+(pos-tree->order)*tree->keylen,
           (char *)n+OFS_KEY+pos*tree->keylen,
           (tree->order*2-pos)*tree->keylen);
    memcpy(&fork->lnk[pos-tree->order+1],
           &n->lnk[pos+1],
           (tree->order*2-pos)*LNK_LEN);
  }
  else
  {
    /* Der einzufügende Schlüssel landet in n */
    memcpy((char *)fork+OFS_KEY,
           (char *)n+OFS_KEY+tree->order*tree->keylen,
           tree->order*tree->keylen);
    memcpy(&fork->lnk[1],
           &n->lnk[tree->order+1],
           tree->order*LNK_LEN);
    if(pos==tree->order)
      fork->lnk[0]=rchild;
    else
    {
      fork->lnk[0]=n->lnk[tree->order];
      memmove(&n->lnk[pos+2],
              &n->lnk[pos+1],
              (tree->order-pos-1)*LNK_LEN);
    }
    memmove((char *)n+OFS_KEY+(pos+1)*tree->keylen,
            (char *)n+OFS_KEY+pos*tree->keylen,
            (tree->order-pos)*tree->keylen);
    memcpy((char *)n+OFS_KEY+pos*tree->keylen,
           key,
           tree->keylen);
    n->lnk[pos+1]=rchild;
  }
  n->count=tree->order;
  return fork;
}

/*****************************************************************************/
/* Diese Funktion löscht einen Schlüssel aus einem Knoten                    */
/*****************************************************************************/
/* Parameter:                                                                */
/*   n      - Zeiger auf den Knoten, aus dem gelöscht werden soll            */
/*   pos    - Position, an der der zu löschende Schlüssel steht              */
/*   tree   - Zeiger auf den B-Baum                                          */
/*****************************************************************************/
void node_delkey(NODE n, unsigned int pos, btree *tree)
{
  void *key, *lnk;
#if defined(_DEBUG)
  void *k;
  k=malloc(tree->keylen);
  node_getkey(k,n, pos, tree);
  fprintf(stderr, "delkey: %i löschen\n", *(int*)k);
  free(k);
```

```
#endif
  if(pos<n->count-1)
  {
    key=malloc(tree->keylen);
    lnk=malloc(LNK_LEN);
    memcpy(key,
           (char *)n+OFS_KEY+pos*tree->keylen,
           tree->keylen);
    memcpy(lnk,
           (char *)n+OFS_LNK+pos*tree->keylen,
           LNK_LEN);
    memmove((char *)n+OFS_KEY+pos*tree->keylen,
            (char *)n+OFS_KEY+(pos+1)*tree->keylen,
            (n->count-pos-1)*tree->keylen);
    memmove(&n->lnk[pos+1],
            &n->lnk[pos+2],
            (n->count-pos)*LNK_LEN);
    memcpy((char *)n+OFS_KEY+(n->count-1)*tree->keylen,
           key,
           tree->keylen);
    memcpy((char *)n+OFS_LNK+n->count*tree->keylen,
           lnk,
           LNK_LEN);
    free(key);
    free(lnk);
  }
  n->count--;
}

/*****************************************************************************/
/* Diese Funktion rotiert einen Schlüssel über seinen Vaterschlüssel um eine  */
/* Stelle nach links                                                         */
/*****************************************************************************/
/* Parameter:                                                                */
/*   key  - Zeiger auf den Vaterschlüssel                                    */
/*   l    - Zeiger auf den linken Knoten                                     */
/*   r    - Zeiger auf den rechten Knoten                                    */
/*   tree - Zeiger auf den B-Baum                                            */
/*****************************************************************************/
void node_rotateleft(void *key, NODE l, NODE r, btree *tree)
{
#if defined(_DEBUG)
  fprintf(stderr, "rotateleft: %i\n", *(int*)key);
#endif
  memcpy((char *)l+OFS_KEY+l->count*tree->keylen,
         key,
         tree->keylen);
  memcpy((char *)l+OFS_LNK+(l->count+1)*LNK_LEN,
         (char *)r+OFS_LNK,
         LNK_LEN);
  l->count++;
  memcpy(key,
         (char *)r+OFS_KEY,
         tree->keylen);
  memmove((char *)r+OFS_KEY,
          (char *)r+OFS_KEY+tree->keylen,
          (r->count-1)*tree->keylen);
  memmove((char *)r+OFS_LNK,
```

```
          (char *)r+OFS_LNK+LNK_LEN,
          (r->count)*LNK_LEN);
  r->count--;
}

/*****************************************************************************/
/* Diese Funktion rotiert einen Schlüssel über seinen Vaterschlüssel um eine  */
/* Stelle nach rechts                                                        */
/*****************************************************************************/
/* Parameter:                                                                */
/*   key  - Zeiger auf den Vaterschlüssel                                    */
/*   l    - Zeiger auf den linken Knoten                                     */
/*   r    - Zeiger auf den rechten Knoten                                    */
/*   tree - Zeiger auf den B-Baum                                            */
/*****************************************************************************/
void node_rotateright(void *key, NODE l, NODE r, btree *tree)
{
#if defined(_DEBUG)
  fprintf(stderr, "rotateright: %i\n", *(int*)key);
#endif
  memmove((char *)r+OFS_KEY+tree->keylen,
          (char *)r+OFS_KEY,
          r->count*tree->keylen);
  memmove((char *)r+OFS_LNK+LNK_LEN,
          (char *)r+OFS_LNK,
          (r->count+1)*LNK_LEN);
  memcpy((char *)r+OFS_KEY,
         key,
         tree->keylen);
  memcpy((char *)r+OFS_LNK,
         (char *)l+OFS_LNK+l->count*LNK_LEN,
         LNK_LEN);
  r->count++;
  memcpy(key,
         (char *)l+OFS_KEY+(l->count-1)*tree->keylen,
         tree->keylen);
  l->count--;
}

/*****************************************************************************/
/* Diese Funktion vereinigt zwei Knoten mit seinem Vaterschlüssel            */
/*****************************************************************************/
/* Parameter:                                                                */
/*   key  - Zeiger auf den Vaterschlüssel                                    */
/*   l    - Zeiger auf den linken Knoten                                     */
/*   r    - Zeiger auf den rechten Knoten                                    */
/*   tree - Zeiger auf den B-Baum                                            */
/*****************************************************************************/
void node_merge(void *key, NODE l, NODE r, btree *tree)
{
#if defined(_DEBUG)
  fprintf(stderr, "merge: %i und rechtes Kind wird zu linkem\n", *(int*)key);
#endif
  memcpy((char *)l+OFS_KEY+l->count*tree->keylen,
         key,
         tree->keylen);
  memcpy((char *)l+OFS_KEY+(l->count+1)*tree->keylen,
         (char *)r+OFS_KEY,
```

```
          r->count*tree->keylen);

  memcpy((char *)l+OFS_LNK+(l->count+1)*LNK_LEN,
          (char *)r+OFS_LNK,
          (r->count+1)*LNK_LEN);
  l->count=tree->order*2;
}
```

```c
/*****************************************************************************/
/* Test-Programm des B-Baumes für wiederholtes manuelles, inkrementelles     */
/* Hinzufügen und Löschen von ausgewählten Schlüsseln (int)                  */
/*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

/* Ist die Konstante _NODEFILE definiert, dann läuft das Programm auf Basis  */
/* einer Indexdatei.                                                         */
/* Das Setzen dieser Konstante übernimmt hier der Compiler                   */
/*#define _NODEFILE*/
/* Einfach die folgende Zeile auskommentieren und schon arbeitet das Programm */
/* mit einer physikalischen Repräsentation des B-Baumes im Speicher          */
/*#undef _NODEFILE*/

#include "btree.h"

#define ORDER 1
#define ANZ_WIDTH 3
#define ANZ_WIDTH_MEM 1 /*(int)(log10(ANZ_WIDTH)+1)*/
#define ANZ_SPACES 1

static unsigned int lastdepth=-1;
static void *lastnode=NULL;

int compar(const void *int1, const void *int2)
{
  return *(int *)int1-*(int *)int2;
}

/* Gibt die nötige Anzahl Zeichen zur Ausgabe einer bestimmten Höhe zurück ->
einfache Zentrierung */
int getwidthbyheight(int height, btree *tree)
{
  unsigned int w, anz, o=tree->order;
  anz=pow(2*o+1, height);
  w=2*o*ANZ_WIDTH+(2*o+1)*1;
  return anz*w+(anz-1)*ANZ_SPACES;
}

/* Gibt die nötige Anzahl erste Zeichen zur Ausgabe einer bestimmten Höhe zurück
*/
int getindentbyheight(int height, btree *tree)
{
  unsigned int w, anz, o=tree->order;
  anz=pow(2*o+1, height);
  w=2*o*ANZ_WIDTH+(2*o+1)*1;
  return anz*w+(anz-1)*pow(2*o*(w+ANZ_SPACES), tree->height-height);
}

unsigned int cnt_keys=0;
unsigned int cnt_node=0;
unsigned int oldidxqueue=0;
unsigned int idxqueue=0;
unsigned int *queue=NULL;
```

```
void iter(unsigned int depth, void *node, void *key, btree *tree)
{
  int i, anz;
  char s[ANZ_WIDTH_MEM+4]={'%','0','\0'};
  sprintf((char *)(&s)+2, "%u", ANZ_WIDTH);
  strcat(&s, "u");

  if(lastnode!=node&&lastdepth!=-1)
  {
    while(cnt_keys<2*tree->order)
    {
      for(i=0; i<ANZ_WIDTH; i++) printf("_");
      cnt_keys++;
      if(depth==tree->height)
        printf("-");
      else if(cnt_keys<tree->order) printf("/");
      else if(cnt_keys==tree->order) printf("|");
      else printf("\\");
    }
    for(i=0;
i<pow(2*tree->order*(2*tree->order*ANZ_WIDTH+(2*tree->order+1)*1+ANZ_SPACES),
tree->height-depth); i++) printf(" ");
    /* Hier muß noch die Implementierung einer Queue hinein, die die Anzahlen
der Schlüssel
       aller Knoten einer Ebene speichert und sie benutzt, um Aussparungen
zwischen den Knoten
       zu generieren */
  }
  if(lastdepth!=depth)
  {
    printf("\nTiefe %u: ", depth);
    anz=(getindentbyheight(tree->height, tree)-getindentbyheight(depth,
tree))/2;
    for(i=0; i<anz; i++) printf(" ");
    lastdepth=depth;
  }
  if(lastnode!=node)
  {
    cnt_keys=0;

    if(depth==tree->height)
      printf("-");
    else if(cnt_keys<tree->order) printf("/");
    else if(cnt_keys==tree->order) printf("|");
    else printf("\\");
    lastnode=node;
  }
  printf(&s, *(unsigned int *)key);
  cnt_keys++;
  if(depth==tree->height)
    printf("-");
  else
    if(cnt_keys<tree->order) printf("/");
    else if(cnt_keys==tree->order) printf("|");
    else printf("\\");
}
void dothelast(btree *tree)
{
```

```
  int i;
  while(cnt_keys<2*tree->order)
  {
    for(i=0; i<ANZ_WIDTH; i++) printf("_");
    cnt_keys++;
    if(lastdepth==tree->height)
      printf("-");
    else if(cnt_keys<tree->order) printf("/");
    else if(cnt_keys==tree->order) printf("|");
    else printf("\\");
  }
}

void iterkey(NODE n)
{
  printf(" %u", *(unsigned int *)n);
}

int main(int argc, char *argv[])
{
  unsigned int i, key, to;
  btree tree={ORDER, sizeof(key), &compar, -1, NULL };
  char action;
  char buf[11];
/*  Dieses Array diente dem automatischem initialen Hinzufügen */
/*  unsigned int a[]={5,11,1,14,27,10,12,0,28,27};*/
#if defined(_NODEFILE)
  FILE *f;
  char *fname;
  btree_param param={NULL, 0};
#endif

#if defined(_NODEFILE)
  if(argc>1)
    fname=argv[1];
  else
    fname="testidx.idx";
#endif
  printf("Test-Programm für den B-Baum\n");
  printf("---------------------------\n");
  printf("zum Einfügen a, zum Löschen d, zum Abfragen q den Werten
voranstellen:\n");
  printf("Ende mit Wert -1.\n");
#if defined(_NODEFILE)
  f=fopen(fname, "w+b");
  if(f==NULL)
  {
    fprintf(stderr, "Die Ausgabedatei %s konnte nicht geöffnet werden\n",
fname);
    exit(1);
  }
  param.f=f;
  tree.param=&param;
  tree.root=NULL;
  btree_init(&tree);
  tree.compar=&compar;
#endif
/*  for(i=0; i<sizeof(a)/sizeof(int); i++)
```

```
    btree_addkey(&a[i], &tree);
  btree_iteratedepths(&iter, &tree);
  dothelast(&tree);*/
  do
  {
    lastdepth=-1;
    lastnode=NULL;
    cnt_keys=0;
    printf("\nGeben Sie die Aktion und einen Schlüssel ein: ");
    scanf("%10s", &buf);
    key=atoi(buf+1);
    if(buf[0]!='-'&&key!=-1)
    {
      switch(buf[0])
      {
        case 'q': case 'Q':
          printf("Von Wert %i bis: ", key);
          scanf("%i", &to);
          printf("Schlüssel in Reihenfolge:");
          printf("\nAnzahl: %u", btree_rangeiterate(&key, &to, &iterkey ,
&tree));
          printf("\nseparat ermittelte Anzahl: %u", btree_rangecount(&key, &to,
&tree));
          break;
        case 'd': case 'D':
          if(btree_delkey(&key, &tree)!=0)
            printf("Schlüssel nicht gefunden!\n");
          btree_iteratedepths(&iter, &tree);
          dothelast(&tree);
          break;
        case 'a': case 'A': default:
          btree_addkey(&key, &tree);
          btree_iteratedepths(&iter, &tree);
          dothelast(&tree);
        break;
      }
    }
  }
  while(key!=-1&&buf[0]!='-');
#if defined(_NODEFILE)
  btree_exit(&tree);
  if(fclose(f))
    printf("Datei konnte nicht geschlossen werden.\n");
#endif
  return 0;
}
```

```c
/*****************************************************************************/
/* Test-Programm des B-Baumes für wiederholte Hinzufügen und Löschen von     */
/* zufällig ausgewählten Schlüsseln (int)                                    */
/*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* Ist die Konstante _NODEFILE definiert, dann läuft das Programm auf Basis  */
/* einer Indexdatei.                                                         */
/* Das Setzen dieser Konstante übernimmt hier der Compiler                   */
/*#define _NODEFILE*/
/* Einfach die folgende Zeile auskommentieren und schon arbeitet das Programm */
/* mit einer physikalischen Repräsentation des B-Baumes im Speicher          */
/*#undef _NODEFILE*/

#include "btree.h"

#define _DEBUG
#undef _DEBUG

#define HASH_CNT 80

unsigned int lastdepth=-1;
void *lastnode=NULL;

int compar(const void *int1, const void *int2)
{
  return *(unsigned int *)int1-*(unsigned int *)int2;
}

void iter(unsigned int depth, void *node, void *key, btree *tree)
{
  if(lastdepth!=depth)
  {
    printf("\nTiefe %u:", depth);
    lastdepth=depth;
  }
  if(lastnode!=node)
  {
    printf(" ");
    if(depth==tree->height)
      printf("N");
    else
      printf("L");
    lastnode=node;
  }
  printf("%u", *(unsigned int *)key);
  if(depth==tree->height)
    printf("N");
  else
    printf("L");
}

void swapints(int *a, int *b)
{
  int t;
  t=*a;
```

```
  *a=*b;
  *b=t;
}

int main(void)
{
  unsigned int i, j, anz, key, hash;
  btree tree={1, sizeof(key), &compar, -1, 0};
  unsigned int *field=NULL;
#if defined(_NODEFILE)
  FILE *f;
  char *fname="output.idx";
  btree_param param={NULL, 0};
#endif
  printf("Test-Programm für die Funktion des B-Baumes\n"
         "-------------------------------------------\n"
         "\n"
         "Im folgenden werden dem Baum:\n"
         "  1. eine bestimmte Anzahl zufälliger Werte hinzugefügt\n"
         "  2. dieselben Werte in zufälliger Reihenfolge wieder entfernt\n");
  do
  {
    lastdepth=-1;
    lastnode=NULL;
    printf("\nIn welcher Ordnung soll der Baum aufgebaut werden (Ende mit 0)?:
");
    scanf(" %i", &tree.order);
    if(tree.order==0)
      break;
#if defined(_NODEFILE)
    f=fopen(fname, "w+b");
    if(f==NULL)
    {
      fprintf(stderr, "Die Ausgabedatei %s konnte nicht geöffnet werden\n",
fname);
      exit(1);
    }
    param.f=f;
    tree.param=&param;
    tree.root=NULL;
    btree_init(&tree);
    tree.compar=&compar;
#endif
    printf("\nMit wievielen Werten möchten Sie testen?: ");
    scanf(" %i", &anz);

    printf("\nNun werden zufällige Werte hinzugefügt (.=1/%u)\n", HASH_CNT);
    field=realloc(field, anz*sizeof(int));
    if(field==NULL)
    {
      printf("Es konnte kein Speicher reserviert werden. Abbruch!\n");
      exit(1);
    }
    srand((unsigned int)time(NULL));
    hash=0;
    for(i=0; i<anz; i++)
    {
      key=rand()%(3*anz);
```

```
      field[i]=key;
      if(btree_addkey(&key, &tree)!=0)
        printf("\nDer Schlüssel %u konnte nicht hinzugefügt werden!\n", key);
#if defined(_DEBUG)
      printf("%u ", key);
#endif
      for(j=hash; j<HASH_CNT*i/anz; j++)
      {
        printf(".");
        fflush(stdout);
      }
      hash=HASH_CNT*i/anz;
    }
    for(j=hash; j<HASH_CNT; j++) printf(".");
#if defined(_DEBUG)
    btree_iteratedepths(&iter, &tree);
#endif
    printf("\nAlle %u Werte wurden erfolgreich hinzugefügt!"
           "\nDer Baum hat nun die Höhe %i"
           "\nNun werden Sie in zufälliger Reihenfolge wieder entfernt
(.=1/%u)\n", anz, tree.height, HASH_CNT);
    srand((unsigned int)time(NULL));
    hash=0;
    for(i=0; i<anz; i++)
    {
      /* Ausgewählter Schlüssel nach vorne */
      swapints(&field[i], &field[i+rand()%(anz-i)]);
      key=field[i];
      if(btree_delkey(&key, &tree)!=0)
        printf("\nEin Schlüssel konnte nicht entfernt werden: %u\n", key);
#if defined(_DEBUG)
      printf("%u ", key);
#endif
      for(j=hash; j<HASH_CNT*i/anz; j++)
      {
        printf(".");
        fflush(stdout);
      }
      hash=HASH_CNT*i/anz;
    }
    for(j=hash; j<HASH_CNT; j++) printf(".");
    if(tree.height==-1)
      printf("\nAlle %u Werte wurden auch wieder erfolgreich entfernt!"
             "\nDer Baum hat nun die Höhe %i\n", anz, tree.height);
    else
      printf("\nDie %u Werte wurden zwar entfernt, aber"
             "\nDer Baum hat nun die Höhe %i", anz, tree.height);
    /* Hier immer Baum anzeigen, da er ja eigentlich leer sein müßte */
    btree_iteratedepths(&iter, &tree);
#if defined(_NODEFILE)
    btree_exit(&tree);
    if(fclose(f))
      printf("Datei konnte nicht geschlossen werden.\n");
#endif
  }
  while(1);
  free(field);
  return 0;
```

```
}
```