1. Bret's Hash Table Functions. These hash table functions are tailored to a very specific purpose, namely storing meteorological METAR reports using the 4-character station name as a key. The design was influenced by the GNU DBM functions, which have some severe limitations where our usage is concerned. In particular, a GDBM file which has been opened for writing is not available to be read until the writer has closed the file. The LDM's DBFILE option does not allow us to close the file after an update, which means that the GDBM file will be unavailable until enough open files force the LDM to start releasing file handles. This could mean that a GDBM file will be inaccessible to a reader for days, which defeats the purpose.

Instead, these functions will allow a file to be opened by one writer at a time, while allowing an unlimited number of readers to have access to the data in the file. In addition, some care is taken to preserve access to the file across minor architecture differences so that variations in native word size and endianness may still be accessed (though there may be a performance penalty). Of course, we have our own constraints, as well:

- All keys are of a fixed size, though the size is selectable at creation time.
- Keys must be unique. However, data may be added to the value of the key.
- The size of the hash table is fixed at creation time.

The basic container elements:

```
#define TRUE (1 ≡ 1)
#define FALSE (¬TRUE)
  ⟨Includes 2⟩
  ⟨Structure Definitions 5⟩
  ⟨Local Includes 4⟩
  ⟨Function Definitions 6⟩
```

2. ⟨Includes 2⟩ ≡

```
#include <stdint.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <stdarg.h>
```

This code is used in section 1.

3. We'll create a "bhtf.h" file for user-accessible definitions and to provide some data hiding.

⟨bhtf.h 3⟩ ≡

```
#ifndef _BHTF_H_
#define _BHTF_H_
#include <stdint.h>    /* Forward function defs need this */
#ifdef __cplusplus
  extern "C"
  {
#endif
    ⟨User Definitions 16⟩
#ifdef __cplusplus
  }
#endif
#endif
```

4. We need some of the user definitions here, but we also need the real structures to be defined first. So we separate the the local include file out to be inserted after the structure defs. This will also allow us to verify that the forward function definitions in the include file jibe with the actual definitions.

⟨ Local Includes 4 ⟩ ≡

```
#include "bhtf.h"
```

This code is used in section 1.

5. ⟨ Structure Definitions 5 ⟩ ≡

⟨ Small Structures 14 ⟩
⟨ Primary Structures 15 ⟩

This code is used in section 1.

6. ⟨ Function Definitions 6 ⟩ ≡

⟨ Forward Declarations 67 ⟩
⟨ Utility Functions 7 ⟩
⟨ File Functions 19 ⟩
⟨ Buffer Functions 45 ⟩
⟨ Message Functions 53 ⟩
⟨ User Functions 82 ⟩

This code is used in section 1.

7. Basic I/O. We do a lot of memory allocation and de-allocation. To help with debugging, we create our own wrapper functions here.

⟨ Utility Functions 7 ⟩ ≡

```
static void *bhtmalloc(char *fname, size_t size)
{
  uint8_t *ptr ← malloc(size);
  if (¬ptr) bht_log("Function␣%s:␣Couldn't␣allocate␣%d␣bytes\n", fname, size);
  return (void *) ptr;
}
```

See also sections 8, 9, 11, 12, 13, 21, 22, 23, 24, 25, 26, 27, 28, 29, 33, 34, 37, 38, 64, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, and 93.

This code is used in section 6.

8. ⟨ Utility Functions 7 ⟩ +≡

```
static void *bhtcalloc(char *fname, size_t els, size_t size)
{
  uint8_t *ptr ← calloc(els, size);
  if (¬ptr)
    bht_log("Function␣%s:␣Couldn't␣allocate␣%d␣elements␣of␣%d␣bytes\n", fname, els, size);
  return (void *) ptr;
}
```

9. To implement *free*( ) with resetting values to Λ, we need a macro.

```
#define bhtfree(str, x) _bhtfree(str, (void **) &(x))
```

⟨ Utility Functions 7 ⟩ +≡

```
static void _bhtfree(char *fname, void **ptr)
{
  if (*ptr ≠ Λ) {
    free(*ptr);
    *ptr ← Λ;
  } else bht_log("Function␣%s:␣Attempting␣to␣free()␣already␣freed␣space.\n", fname);
}
```

10. Next for some simple file operations. These functions are mostly wrappers around the system operations, but they perform a little error-checking. They will return `BHT_OK` on success, and `BHT_ERR` on failure. We could (should?) eventually do more sophisticated error-handling here, so that we can take appropriate action should things fail.

11. Set the file location for the next read/write operation.

⟨ Utility Functions 7 ⟩ +≡

```
static int setloc(int fd, off_t loc)
{
  off_t retval;
  retval ← lseek(fd, loc, SEEK_SET);
  if (retval < 0) {
    bht_log("Couldn't␣seek:␣err=%d\n", errno);
    return BHT_ERR;
  } else return BHT_OK;
}
```

12. When reading from a file, there are several things that may not go completely right. The *read*( ) call may have been interrupted by a signal, in which case we just need to try again. Or there may have been some other error, in which case we must abort the read.

If we get a positive return value but the number of bytes read is less than expected, we should try to continue to read until we have a complete buffer. But we don't try forever: `MAXTRIES` is the most attempts we'll make before calling it quits on the buffer.

```
#define MAXTRIES 5
```

⟨ Utility Functions 7 ⟩ +≡

```
static int disk_to_buff(int fd, uint8_t *d, int size)
{
  int tries, bytes, retval;
  bytes ← tries ← 0;
  while ((tries++ < MAXTRIES) ∧ (bytes < size)) {
    retval ← read(fd, d + bytes, size − bytes);
    if (retval < 0) {
      if (errno ≡ EINTR) continue;
      else {
        bht_log("Buffer␣read␣failed:␣%d\n", errno);
        break;
      }
    } else bytes += retval;
  }
  if (bytes ≡ size) return BHT_OK;
  else return BHT_ERR;
}
```

13. The writing process is similar to the reading process.

⟨ Utility Functions 7 ⟩ +≡

```
static int buff_to_disk(int fd, uint8_t *d, int size)
{
  int tries, bytes, retval;
  bytes ← tries ← 0;
  while ((tries++ < MAXTRIES) ∧ (bytes < size)) {
    retval ← write(fd, d + bytes, size − bytes);
    if (retval < 0) {
      if (errno ≡ EINTR) continue;
      else {
        bht_log("Buffer␣write␣failed:␣%d\n", errno);
        break;
      }
    } else bytes += retval;
  }
  if (bytes ≡ size) return BHT_OK;
  else return BHT_ERR;
}
```

14. Principal Structures. This is what a hashtable element looks like. When we're reading the hash table from disk, we'll set up pointers to the key strings and fill in the data offsets.

⟨ Small Structures 14 ⟩ ≡

```
struct ht {
  uint8_t *key;
  uint32_t offset;
};
```

See also sections 36 and 65.

This code is used in section 5.

15. We have to keep track of a few variables per file handle. This is the structure returned when opening a file.

⟨ Primary Structures 15 ⟩ ≡

```
struct bht_conf {
  int fd, writer, seekkey;
  int32_t serial, keysize, tabsize, keycount, maxnode;
  int32_t first_char, last_char;
  uint32_t *lvals;
  struct ht **htable;
  uint32_t HMult, HInit;
  uint32_t scram_off, ht_off, data_off, next_off;
  struct buff *y, *o;
};
typedef struct bht_conf *BHTFile;
```

This code is used in section 5.

16. We provide an opaque type for the user, but we leave ourselves with the real structure.

```
#define _BHTF_SRC_ TRUE
```

⟨ User Definitions 16 ⟩ ≡

```
#ifndef _BHTF_SRC_
  struct bht_conf;
  typedef struct bht_conf *BHTFile;
#endif
```

See also sections 80, 81, 91, and 97.

This code is used in section 3.

17. A file is laid out as follows: First, the parameter block, which contains pointers and configuration information about the file itself. Following the parameter block, the randomized letter-conversion data are written. Following the letter conversions, the hash table data are written, followed by the data associated with the hash table keys. Hash table data should aligned on `BUFFSIZE` boundaries so that a buffer read/write is less likely to straddle file-system blocks.

```
#define PARAM_BLOCK_SZ 512
#define PARAM_HEADER "BHTF"
#define PARAM_VMAJOR 1
#define PARAM_VMINOR 0
#define PARAM_VSUBMIN 1
```

18. Here we establish offsets in the parameter block for locations of the variables. These offsets are used as keys to retrieve or place data. Changing these offsets or their order will mean that older files may no longer be readable, so `VMAJOR` should be incremented so that older files won't be scrozzled by the current version of these routines.

```
#define SPHEADER 0
#define SPVERSION (SPHEADER + 4)
#define SPSERIAL (SPVERSION + 4)
#define SPKEYSIZE (SPSERIAL + 4)
#define SPTABSIZE (SPKEYSIZE + 4)
#define SPKEYCOUNT (SPTABSIZE + 4)
#define SPHMULT (SPKEYCOUNT + 4)
#define SPHINIT (SPHMULT + 4)
#define SPSCRMFC (SPHINIT + 4)
#define SPSCRMLC (SPSCRMFC + 4)
#define SPSCRMOFF (SPSCRMLC + 4)
#define SPHTOFF (SPSCRMOFF + 4)
#define SPDTOFF (SPHTOFF + 4)
#define SPNXTOFF (SPDTOFF + 4)
#define SPMAXND (SPNXTOFF + 4)
```

19. ⟨ File Functions 19 ⟩ ≡

```
static void flush_parameters(BHTFile c)
{
  uint8_t *p ← bhtmalloc("flush_parameters", PARAM_BLOCK_SZ);
  int retval;
  if (¬p) return;
  putstr(p, SPHEADER, (uint8_t *) PARAM_HEADER);
  putver(p, SPVERSION, PARAM_VMAJOR, PARAM_VMINOR, PARAM_VSUBMIN);
  putint(p, SPSERIAL, c→serial);
  putint(p, SPKEYSIZE, c→keysize);
  putint(p, SPTABSIZE, c→tabsize);
  putint(p, SPKEYCOUNT, c→keycount);
  putuint(p, SPHMULT, c→HMult);
  putuint(p, SPHINIT, c→HInit);
  putint(p, SPSCRMFC, c→first_char);
  putint(p, SPSCRMLC, c→last_char);
  putuint(p, SPSCRMOFF, c→scram_off);
  putuint(p, SPHTOFF, c→ht_off);
  putuint(p, SPDTOFF, c→data_off);
  putuint(p, SPNXTOFF, c→next_off);
  putint(p, SPMAXND, c→maxnode);
  if ((retval ← setloc(c→fd, 0)) ≡ BHT_OK) retval ← buff_to_disk(c→fd, p, PARAM_BLOCK_SZ);
  write_scramble(c, p, PARAM_BLOCK_SZ);
  bhtfree("flush_parameters", p);
}
```

See also sections 20, 30, 35, 39, 42, 66, 68, 88, 89, and 90.

This code is used in section 6.

20. Set the **BHTFile** structure up according to the information contained in the file. The only assumption we have at the beginning of this function is that c has already been allocated, and that $c\to fd$ contains a valid open file descriptor.

```
⟨ File Functions 19 ⟩ +≡
  static int ingest_parameters(BHTFile c)
  {
    uint8_t *p ← bhtmalloc("ingest_parameters", PARAM_BLOCK_SZ);
    int retval;
    if (¬p) return BHT_ERR;
    if ((retval ← setloc(c→fd, 0)) ≡ BHT_OK) retval ← disk_to_buff(c→fd, p, PARAM_BLOCK_SZ);
    if ((retval ≠ BHT_OK) ∨ ¬verify_headver(p, SPHEADER, SPVERSION)) {
      bhtfree("ingest_parameters", p);
      return BHT_ERR;
    }
    c→serial ← getint(p, SPSERIAL);
    c→keysize ← getint(p, SPKEYSIZE);
    c→tabsize ← getint(p, SPTABSIZE);
    c→keycount ← getint(p, SPKEYCOUNT);
    c→HMult ← getuint(p, SPHMULT);
    c→HInit ← getuint(p, SPHINIT);
    c→first_char ← getint(p, SPSCRMFC);
    c→last_char ← getint(p, SPSCRMLC);
    c→scram_off ← getuint(p, SPSCRMOFF);
    c→ht_off ← getuint(p, SPHTOFF);
    c→data_off ← getuint(p, SPDTOFF);
    c→next_off ← getuint(p, SPNXTOFF);
    c→maxnode ← getint(p, SPMAXND);
    read_scramble(c, p, PARAM_BLOCK_SZ);
    bhtfree("ingest_parameters", p);
    return BHT_OK;
  }
```

21. We need some function to assist in reading/writing quantities from/to disk.

```
⟨ Utility Functions 7 ⟩ +≡
  inline static void putint(uint8_t *p, uint32_t off, int32_t val)
  {
    int i ← sizeof(int32_t);
    p += off + 3;
    while (i−−) {
      *p−− ← val & #ff;
      val ≫= 8;
    }
  }
```

22. ⟨ Utility Functions 7 ⟩ +≡

```
inline static void putuint(uint8_t *p, uint32_t off, uint32_t val)
{
  int i ← sizeof(uint32_t);
  p += off + 3;
  while (i--) {
    *p-- ← val & #ff;
    val >>= 8;
  }
}
```

23. ⟨ Utility Functions 7 ⟩ +≡

```
inline static void putstr(uint8_t *p, uint32_t off, uint8_t *s)
{
  p += off;
  while (*s) *p++ ← *s++;
}
```

24. ⟨ Utility Functions 7 ⟩ +≡

```
inline static void putver(uint8_t *p, uint32_t off, uint32_t maj, uint32_t min, uint32_t submin)
{
  p += off;
  *p++ ← 'v';
  *p++ ← maj & #ff;
  *p++ ← min & #ff;
  *p ← submin & #ff;
}
```

25. ⟨ Utility Functions 7 ⟩ +≡

```
inline static int32_t getint(uint8_t *p, uint32_t off)
{
  int i ← sizeof(int32_t);
  int32_t val ← 0;
  p += off;
  while (i--) {
    val <<= 8;
    val |= *p++;
  }
  return val;
}
```

26. ⟨ Utility Functions 7 ⟩ +≡

```
inline static uint32_t getuint(uint8_t *p, uint32_t off)
{
  int i ← sizeof(uint32_t);
  uint32_t val ← 0;
  p += off;
  while (i--) {
    val <<= 8;
    val |= *p++;
  }
  return val;
}
```

27. Compare the file header with the library header string, returning `TRUE` if the file has a compatible header and version string, or `FALSE` otherwise. The `PARAM_VMAJOR` parameters must match, but the file `PARAM_VMINOR` and `PARAM_VSUBMIN` need only be less than or equal the values contained here.

⟨ Utility Functions 7 ⟩ +≡

```
inline static int verify_headver(uint8_t *p, uint32_t hoff, uint32_t voff)
{
  uint8_t *ph;
  uint32_t maj, min, submin;
  ph ← p + hoff;
  if (strncmp((char *) ph, PARAM_HEADER, 4) ≠ 0) return FALSE;
  ph ← p + voff;
  if (*ph++ ≠ 'v') return FALSE;
  maj ← *ph++;
  min ← *ph++;
  submin ← *ph;
  if (maj ≠ PARAM_VMAJOR) return FALSE;
  if ((min < PARAM_VMINOR) ∨ ((min ≡ PARAM_VMINOR) ∧ (submin ≤ PARAM_VSUBMIN))) return TRUE;
  else return FALSE;
}
```

28. Write the character scramble table to disk. The table offset begins right after the main paramater block. The table is just a list of unsigned integers, and the length of the table is defined by the first and last character as specified in the parameter block. As the scramble table is written along with the parameters, we ask the calling function to pass us a usable buffer rather than re-allocating our own here.

```
⟨ Utility Functions 7 ⟩ +≡
  static void write_scramble(BHTFile c, uint8_t *p, int buffsz)
  {
    int retval, counter, i, numchars, maxinbuff;
    if ((retval ← setloc(c→fd, c→scram_off)) ≠ BHT_OK) return;
    counter ← 0;
    numchars ← c→last_char − c→first_char + 1;
    maxinbuff ← buffsz/sizeof(uint32_t);
    for (i ← 0; i < numchars; i++) {
      putuint(p, (sizeof(uint32_t) * counter++), c→lvals[i]);
      if (counter ≡ maxinbuff) {
        if ((retval ← buff_to_disk(c→fd, p, buffsz)) ≠ BHT_OK) return;
        counter ← 0;
      }
    }
    if (counter > 0) retval ← buff_to_disk(c→fd, p, buffsz);
  }
```

29. Read the character scramble table from disk, allocating space for the scramble table.

```
⟨ Utility Functions 7 ⟩ +≡
  static void read_scramble(BHTFile c, uint8_t *p, int buffsz)
  {
    int retval, counter, i, numchars, maxinbuff;
    if ((retval ← setloc(c→fd, c→scram_off)) ≠ BHT_OK) return;
    numchars ← c→last_char − c→first_char + 1;
    maxinbuff ← buffsz/sizeof(uint32_t);
    c→lvals ← bhtcalloc("read_scramble", numchars, sizeof(uint32_t));
    if (¬c→lvals) return;
    counter ← 0;
    if ((retval ← disk_to_buff(c→fd, p, buffsz)) ≠ BHT_OK) {
      bhtfree("read_scramble", c→lvals);
      c→lvals ← Λ;
      return;
    }
    for (i ← 0; i < numchars; i++) {
      c→lvals[i] ← getuint(p, (sizeof(uint32_t) * counter++));
      if (counter ≡ maxinbuff) {
        if ((retval ← disk_to_buff(c→fd, p, buffsz)) ≠ BHT_OK) {
          bhtfree("read_scramble", c→lvals);
          c→lvals ← Λ;
          return;
        }
        counter ← 0;
      }
    }
  }
```

30. Create and initialize a **BHTFile** structure. We call this function if we're creating a new file from scratch. (This implies that we have write permission already, or what's the point?)

```
#define HASHPRIME 5507
#define FNV1_32_INIT ((unsigned int) #811c9dc5)
#define FNV_32_PRIME ((unsigned int) #01000193)
```

⟨ File Functions 19 ⟩ +≡

```
static BHTFile bht_init(int keysize, int tablesize, int nbuffs)
{
  BHTFile c ← bhtmalloc("bht_init", sizeof(struct bht_conf));
  if (¬c) return Λ;
  ⟨ Initialize fixed parameters 31 ⟩
  set_char_scramble(c, (int) '␣', (int) '~');
  if (¬c→lvals) {
    bhtfree("bht_init", c);
    return Λ;
  }
  if (¬(c→htable ← bhtcalloc("bht_init", c→tabsize, sizeof(struct ht)))) {
    bhtfree("bht_init", c→lvals);
    bhtfree("bht_init", c);
    return Λ;
  }
  create_buffs(c, nbuffs);
  ⟨ Calculate file offsets 32 ⟩
  return c;
}
```

31. ⟨ Initialize fixed parameters 31 ⟩ ≡

```
c→fd ← −1;
c→writer ← TRUE;
c→serial ← 0;
c→seekkey ← −1;
c→keysize ← keysize;
c→HMult ← FNV_32_PRIME;
c→HInit ← FNV1_32_INIT;
c→maxnode ← −1;
c→keycount ← 0;
if (tablesize ≤ 0) c→tabsize ← HASHPRIME;
else c→tabsize ← tablesize;
```

This code is used in section 30.

32. Here we calculate the disk offsets of certain portions of a BHT file. The first part of the file contains basic parameters and the character scramble table used in the hash function calculation. The parameter block is of a fixed size `PARAM_BLOCK_SZ`, and the scramble table is an array of 4-byte integers which follows immediately.

We are assuming that a filesystem block is 4K bytes, but unless we have a huge character scramble table, we aren't close to that yet. Rather than waste most of a filesystem block, we allow the hash table to start at a 2K boundary. (Maybe this is foolish.) However, we do force the data area to begin at a filesystem block boundary, rounding the data offset up to the nearest 4K-multiple past the hash table.

Finally, we initialize *next_off* to point to the beginning of the data area. This value changes as keys are added to the file.

```
⟨ Calculate file offsets 32 ⟩ ≡
  c→scram_off ← PARAM_BLOCK_SZ;
  c→ht_off ← (c→scram_off + (c→last_char − c→first_char + 1) ∗ sizeof(uint32_t) + THASHBUFFSZ − 1) &
      (∼(THASHBUFFSZ − 1));
  c→data_off ← (c→ht_off + c→tabsize ∗ (c→keysize + BHT_MSG_HDRSZ) + BUFFSIZE − 1) & (∼(BUFFAMOD));
  c→next_off ← c→data_off;
```

This code is used in section 30.

33. Return a 32-bit pseudo-random number.

```
⟨ Utility Functions 7 ⟩ +≡
  static uint32_t rand32( )
  {
    return random( ) & #ffffffff;
  }
```

34. Set up a character scramble table for the hash function. So that the hash values may be recovered, this "scramble" will also need to be written to the database file.

```
⟨ Utility Functions 7 ⟩ +≡
  static void set_char_scramble(BHTFile c, int first, int last)
  {
    int i, numchars;
    c→first_char ← first;
    c→last_char ← last;
    numchars ← last − first + 1;
    c→lvals ← bhtcalloc("set_char_scramble", numchars, sizeof(uint32_t));
    if (¬c→lvals) {
      bht_log("Couldn't␣allocate␣space␣for␣character␣scramble␣table\n");
      return;
    }
    for (i ← 0; i < numchars; i++) c→lvals[i] ← rand32( );
  }
```

35. Open an existing file, and set up a **BHTFile** structure from the existing parameters.

```
⟨ File Functions 19 ⟩ +≡
  static BHTFile bht_read(int fd, int writer, int nbuffs)
  {
    BHTFile c ← bhtmalloc("bht_read", sizeof(struct bht_conf));
    if (¬c) return Λ;
    c→fd ← fd;
    c→writer ← writer;
    c→seekkey ← −1;
    if (ingest_parameters(c) ≠ BHT_OK) {
      bhtfree("bht_read", c);
      return Λ;
    }
    if (¬(c→htable ← bhtcalloc("bht_read", c→tabsize, sizeof(struct ht)))) {
      bhtfree("bht_read", c→lvals);
      bhtfree("bht_read", c);
      return Λ;
    }
    read_hash(c);
    create_buffs(c, nbuffs);
    return c;
  }
```

36. Buffer Management. We will keep the hash table in RAM, but the data values associated with each key will be kept in buffers which will need to be written out to disk occasionally (when dirty). We'll need a minimum of two buffers, in case we need to copy data between key values which may be on different buffer "pages". We'll probably allow for more, however. The buffer size must be a power of two since we'll be using bit-shifts and bit-wise logical ANDs of file offsets to determine the buffer page number and buffer offsets.

The buffer slots will be kept in a doubly-linked list so that we can more easily manage a Least-Recently-Used algorithm to recycle buffers as we need them. The y pointer will point back in the list to the next "younger" slot, and the o pointer will point forward to the next "older" slot.

```
#define BUFFSHFT (12)      /* size of bit shifts */
#define BUFFSIZE (1 ≪ BUFFSHFT)      /* 2^12 = 4096 */
#define BUFFAMOD (BUFFSIZE − 1)      /* #fff */
⟨ Small Structures 14 ⟩ +≡
  struct buff {
    int buffnum;
    int dirty;
    uint8_t *data_area;
    struct buff *y, *o;
  };
```

37. Create the buffers. This entails creating the buffer slots and the doubly-linked list structure, and allocating space for the buffers themselves. The most-recently-used and least-recently-used pointers are also set. Buffers are numbered starting at 0, so any *buffnum* < 0 indicates an unused slot. LRU management and copying data between buffers both require that we have a minimum of two buffers.

```
#define MINBUFFS 2

⟨ Utility Functions 7 ⟩ +≡
  static void create_buffs(BHTFile c, int numbuffs)
  {
    struct buff *slot, *prevslot ← Λ;
    if (numbuffs < MINBUFFS) numbuffs ← MINBUFFS;
    while (numbuffs−−) {
      slot ← bhtmalloc("create_buffs", sizeof(struct buff));
      if (prevslot) prevslot→o ← slot;
      else c→y ← slot;
      slot→buffnum ← −1;
      slot→dirty ← FALSE;
      slot→data_area ← bhtmalloc("create_buffs", BUFFSIZE);
      slot→y ← prevslot;
      prevslot ← slot;
    }
    slot→o ← Λ;
    c→o ← slot;
  }
```

38. Make a buffer slot s the most recently used, possibly reordering the buffer slot list. The slots are in a doubly-linked list, with $c\to y$ pointing to the most-recently-used (head), and $c\to o$ pointing to the least-recently-used (tail). If s is already at the head of the list, then there's nothing more to do. If s is in the middle of the list, then it needs to be unlinked from its present position and moved to the front, and the surrounding pointers must be updated. If s is in the LRU position, then the LRU pointer will also need to be updated to point to the second least-recently-used slot.

```
⟨ Utility Functions 7 ⟩ +≡
  static void touch_buffer(BHTFile c, struct buff *s)
  {
    if (¬s→y) return;    /* already at head of list, nothing to do */
    s→y→o ← s→o;
    if (s→o) s→o→y ← s→y;    /* Not at tail of list */
    else c→o ← s→y;    /* in LRU slot */
    s→y ← Λ;
    s→o ← c→y;
    c→y→y ← s;
    c→y ← s;
  }
```

39. Read a buffer-full of data from the file. The buffer number indicates some multiples of BUFFSIZE from the beginning of the file's data area. The data area follows the key hash table area, and it's given by *c*→*data_off*.

```
⟨ File Functions 19 ⟩ +≡
  static int getbuff(BHTFile c, struct buff *dbuff, int buffnum)
  {
    off_t foffset;
    int retval;
    ⟨ Basic parameter and buffer slot error checking 40 ⟩
    foffset ← c→data_off + (buffnum ≪ BUFFSHFT);
    if ((retval ← setloc(c→fd, foffset)) ≡ BHT_OK) retval ← disk_to_buff(c→fd, dbuff→data_area, BUFFSIZE);
    ⟨ Update the file state on completion 41 ⟩
  }
```

40. We assume that the buffer is already clean (i.e., the buffer has been flushed to disk already if it needed to be updated).

```
⟨ Basic parameter and buffer slot error checking 40 ⟩ ≡
  if (¬c ∨ ¬dbuff ∨ ¬dbuff→data_area ∨ buffnum < 0) {
    bht_log("Invalid␣values␣trying␣to␣read␣a␣buffer\n");
    return BHT_ERR;
  }
  if (c→fd < 0) {
    bht_log("Bad␣file␣descriptor␣trying␣to␣read␣a␣buffer\n");
    return BHT_ERR;
  }
```

This code is used in section 39.

41. At this point, either the buffer read was successfully or the read failed. In either case, we consider the buffer slot clean. If the read did not succeed, however, we may have partial data or none at all, so we mark the slot "unused" by setting *dbuff*→*buffnum* ← −1.

```
⟨ Update the file state on completion 41 ⟩ ≡
  dbuff→dirty ← FALSE;
  if (retval ≡ BHT_OK) {
    dbuff→buffnum ← buffnum;
    return BHT_OK;
  } else {
    dbuff→buffnum ← −1;
    return BHT_ERR;
  }
```

This code is used in section 39.

42. Write a buffer-full of data to the file. Update the file's maximum node number (*c→maxnode*) if this is a new node. We use the slot's *buffnum* parameter to determine which buffer chunk we are writing to disk, and we calculate the absolute offset *foffset* from that.

```
⟨ File Functions 19 ⟩ +≡
  static int putbuff(BHTFile c, struct buff *dbuff)
  {
    off_t foffset;
    int retval;
    ⟨ Check the parameters for writing 43 ⟩
    foffset ← c→data_off + (dbuff→buffnum ≪ BUFFSHFT);
    if ((retval ← setloc(c→fd, foffset)) ≡ BHT_OK) retval ← buff_to_disk(c→fd, dbuff→data_area, BUFFSIZE);
    ⟨ Update the state of the file after writing 44 ⟩
  }
```

43. This is not a user-callable function, so some of these checks are redundant and/or excessive. However, we should probably verify that the buffer is actually dirty before writing.

```
⟨ Check the parameters for writing 43 ⟩ ≡
  if (¬c ∨ ¬dbuff ∨ dbuff→buffnum < 0 ∨ c→fd < 0) {
    bht_log("Parameter␣errors␣in␣trying␣to␣write␣a␣buffer\n");
    return BHT_ERR;
  }
  if (¬dbuff→dirty) return BHT_ERR;
```

This code is used in section 42.

44. If we have successfully written data to the file, we can reset the buffer slot's *dirty* flag. If we have extended the file's data area (*dbuff→buffnum* > *c→maxnode*), then we update *c→maxnode* as well.

```
⟨ Update the state of the file after writing 44 ⟩ ≡
  if (retval ≡ BHT_OK) {
    if (dbuff→buffnum > c→maxnode) c→maxnode ← dbuff→buffnum;
    dbuff→dirty ← FALSE;
    return BHT_OK;
  } else return BHT_ERR;
```

This code is used in section 42.

45. Flush any dirty buffers to disk, starting from the least-recently-used buffer, working toward the most-recently-used buffer.

```
⟨ Buffer Functions 45 ⟩ ≡
  static void flush_buffers(BHTFile c)
  {
    struct buff *slot;
    slot ← c→o;
    while (slot) {
      if (slot→dirty) putbuff(c, slot);
      slot ← slot→y;
    }
  }
```

See also sections 46, 49, 50, and 51.

This code is used in section 6.

46. Find a buffer containing the desired file offset. If there is no current buffer with the appropriate data, then find the least-recently-used buffer and load it with the desired data from the file. If the buffer is dirty, its current data must first be flushed to disk.

If the flag *markdirty* ≡ TRUE, we will mark the selected slot dirty before returning the buffer's address. Otherwise, the slot's dirty status will remain unchanged. (Since this function does not actually alter any buffer contents, it seems inappropriate to manage buffer-dirtiness here. However, since this function returns an address to the data buffer and not to the buffer slot, this is the only opportunity a calling function has to update the *dirty* status.)

TODO: check the return values of *putbuff* ( ) and *getbuff* ( ) and take appropriate action of either of these have failed.

```
⟨ Buffer Functions 45 ⟩ +≡
  static uint8_t *find_buffer(BHTFile c, uint32_t offset, int markdirty)
  {
    struct buff *slot;
    int buffnum;
    ⟨ Seek a slot containing offset, and return if found 47 ⟩
    ⟨ Retrieve the LRU slot and load it with the buffnum 48 ⟩
    if (markdirty) slot→dirty ← TRUE;
    return slot→data_area;
  }
```

47. The file's data area is kept in BUFFSIZE chunks, with the first chunk located at *c→data_off*. We first calculate the chunk number *buffnum* from the absolute file *offset*. Then we search through the buffer slots to see if the chunk is already loaded in memory, starting with the most-recently-used slot. If we find the buffer already loaded, we can return the address immediately, after updating the least-recently-used status of the buffer slot and the *dirty* flag (if necessary).

```
⟨ Seek a slot containing offset, and return if found 47 ⟩ ≡
  buffnum ← (offset − c→data_off) ≫ BUFFSHFT;
  slot ← c→y;
  while (slot) {
    if (slot→buffnum ≡ buffnum) {
      touch_buffer(c, slot);
      if (markdirty) slot→dirty ← TRUE;
      return slot→data_area;
    }
    slot ← slot→o;
  }
```

This code is used in section 46.

48. The search for an existing buffer containing *buffnum* failed, so the backup plan is to grab the LRU slot, flush the buffer to disk if it's dirty, and load the required *buffnum* from disk if it exists. However, if *buffnum* > *c*→*maxnode*, then we are extending the file's data area, so we'll set up an empty buffer instead, and we won't attempt to read anything from the file.

⟨ Retrieve the LRU slot and load it with the *buffnum* 48 ⟩ ≡

```
slot ← c→o;
touch_buffer(c, slot);
if (slot→dirty) putbuff(c, slot);
if (buffnum > c→maxnode) {
  bzero(slot→data_area, BUFFSIZE);
  slot→buffnum ← buffnum;
} else getbuff(c, slot, buffnum);
```

This code is used in section 46.

49. Write *data* of *length* to the file beginning at *offset* using the buffer system.

⟨ Buffer Functions 45 ⟩ +≡

```
static int wr_to_buffs(BHTFile c, uint8_t *data, int length, uint32_t offset)
{
  uint8_t *mbuff, *p, *maxp;
  int w ← 0;
  if (¬data ∨ length ≤ 0 ∨ ¬offset) return BHT_OK;
  while (w < length) {
    mbuff ← find_buffer(c, offset + w, TRUE);
    p ← &(mbuff[(offset − c→data_off + w) & BUFFAMOD]);
    maxp ← mbuff + BUFFSIZE;
    while ((w < length) ∧ (p < maxp)) {
      *p++ ← *data++;
      w++;
    }
  }
  return BHT_OK;
}
```

50. Read *data* of *length* from the file beginning at *offset* using the buffer system. Pointer *data* must point to valid space, as nothing is *alloc*( )'d by this function.

```
⟨ Buffer Functions 45 ⟩ +≡
  static int rd_from_buffs(BHTFile c, uint8_t *data, int length, uint32_t offset)
  {
    uint8_t *mbuff, *p, *maxp;
    int w ← 0;
    if (¬data ∨ length ≤ 0 ∨ ¬offset) return BHT_OK;
    while (w < length) {
      mbuff ← find_buffer(c, offset + w, FALSE);
      p ← &(mbuff[(offset − c→data_off + w) & BUFFAMOD]);
      maxp ← mbuff + BUFFSIZE;
      while ((w < length) ∧ (p < maxp)) {
        *data++ ← *p++;
        w++;
      }
    }
    return BHT_OK;
  }
```

51. Copy data of *length* from *src_offset* to *dst_offset* through the buffers. The source and destination locations should not overlap. At least two buffers must have been created, or this function will not always work correctly.

When we have reached the end of either the source or destination buffer, we recalculate pointers and limits for both buffers. Since we will be reloading at least one buffer, by re-accessing both buffer slots using *find_buffer*( ), we ensure that we have the correct address for the source and destination pointers.

```
⟨ Buffer Functions 45 ⟩ +≡
  static int cp_buff_data(BHTFile c, uint32_t src_offset, uint32_t dst_offset, int length)
  {
    uint8_t *sbuff, *dbuff, *sp, *dp, *maxs, *maxd;
    int w ← 0;
    while (w < length) {
      sbuff ← find_buffer(c, src_offset + w, FALSE);
      sp ← &(sbuff[(src_offset − c→data_off + w) & BUFFAMOD]);
      maxs ← sbuff + BUFFSIZE;
      dbuff ← find_buffer(c, dst_offset + w, TRUE);
      dp ← &(dbuff[(dst_offset − c→data_off + w) & BUFFAMOD]);
      maxd ← dbuff + BUFFSIZE;
      while ((w < length) ∧ (sp < maxs) ∧ (dp < maxd)) {
        *dp++ ← *sp++;
        w++;
      }
    }
    return BHT_OK;
  }
```

52. Message Handling. A message is a simple construct: it consists of a short header, followed by the message data. Message headers consist of 4 bytes. The first byte will be either `'a'` or `'d'`, indicating that the message slot is either "active" or "deactivated", respectively. The next three bytes are the length of the message, not counting the header bytes. The maximum message length is therefore $2^{24} - 1$ bytes.

The message location is an absolute file offset stored along with its key in the hash table. While the message length is contained in the message header, the data space allocated to the message may actually be larger. This extra space at the end of the message data should be zero-filled. Right now, a message that is smaller than its allocated data space will only be smaller if a new, shorter message replaced the older, longer message data. This may become more common if a free-list of old message space is implemented.

```
#define BHT_MSG_HDRSZ 4
```

53. Given a file *offset*, return the message length at that location. A negative return length indicates an inactive data slot. A 0 return value indicates an invalid entry.

```
⟨ Message Functions 53 ⟩ ≡
  static int get_msg_length(BHTFile c, uint32_t offset)
  {
    uint8_t b[BHT_MSG_HDRSZ];
    int length;
    rd_from_buffs(c, b, BHT_MSG_HDRSZ, offset);
    length ← (b[1] ≪ 16) | (b[2] ≪ 8) | b[3];
    if (b[0] ≡ 'a') return length;
    else if (b[0] ≡ 'd') return −length;
    else return 0;
  }
```

See also sections 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, and 79.

This code is used in section 6.

54. Set the message *length* at a given file *offset*. A negative *length* indicates that the data slot of −*length* should be marked inactive. On success, length is returned.

```
⟨ Message Functions 53 ⟩ +≡
  static int set_msg_length(BHTFile c, uint32_t offset, int length)
  {
    uint8_t b[BHT_MSG_HDRSZ];
    int mylen;
    if (length < 0) {
      mylen ← −length;
      b[0] ← 'd';
    } else if (length > 0) {
      mylen ← length;
      b[0] ← 'a';
    } else return 0;
    if ((mylen & #ffffff) ≠ mylen)      /* too big */
      return 0;
    b[1] ← (mylen ≫ 16) & #ff;
    b[2] ← (mylen ≫ 8) & #ff;
    b[3] ← mylen & #ff;
    wr_to_buffs(c, b, BHT_MSG_HDRSZ, offset);
    return mylen;
  }
```

55. Write message *data* of *length* to the message *offset*. The actual data will be written just past the message header, which is not included in the message data *length* count.

```
⟨ Message Functions 53 ⟩ +≡
  static void write_msg_data(BHTFile c, uint32_t offset, uint8_t *data, int length)
  {
    wr_to_buffs(c, data, length, offset + BHT_MSG_HDRSZ);
  }
```

56. Copy a message from *src_offset* to *dst_offset*.

```
⟨ Message Functions 53 ⟩ +≡
  static void copy_msg_data(BHTFile c, uint32_t src_offset, uint32_t dst_offset, int length)
  {
    if (length ≤ 0) return;
    cp_buff_data(c, src_offset + BHT_MSG_HDRSZ, dst_offset + BHT_MSG_HDRSZ, length);
  }
```

57. Given a longer old message length and a newer shorter message length, clear the data at the end of the new message. If *newlength* ≥ *oldlength*, the function does nothing. Otherwise, it creates a zeroed string of the proper length to copy to the buffer(s).

```
⟨ Message Functions 53 ⟩ +≡
  static void clear_msg_data(BHTFile c, uint32_t offset, int oldlength, int newlength)
  {
    uint8_t *clrstr;

    if (oldlength − newlength ≤ 0) return;
    clrstr ← bhtmalloc("clear_msg_data", oldlength − newlength);
    bzero(clrstr, oldlength − newlength);
    wr_to_buffs(c, clrstr, oldlength − newlength, offset + BHT_MSG_HDRSZ + newlength);
    bhtfree("clear_msg_data", clrstr);
  }
```

58. Retrieve message data of *length* from *offset*, copying data to the supplied *data* pointer.

```
⟨ Message Functions 53 ⟩ +≡
  static void read_msg_data(BHTFile c, uint32_t offset, uint8_t *data, int length)
  {
    rd_from_buffs(c, data, length, offset + BHT_MSG_HDRSZ);
  }
```

59. Find and return a new file *offset* which can contain a message of *length*. (The *length* parameter should not include the size of the header: this extra info is handled internally by the functions.)

Eventually, this function might include some message free-list management, but for now it will just create a new data offset.

```
⟨ Message Functions 53 ⟩ +≡
  static uint32_t find_newoffset(BHTFile c, int length)
  {
    uint32_t offset;

    offset ← c→next_off;
    c→next_off += length + BHT_MSG_HDRSZ;
    return offset;
  }
```

60. Create a brand new data message and store it, updating the reference in the hash table.

```
⟨ Message Functions 53 ⟩ +≡
  static void msgnew(BHTFile c, struct ht *hp, uint8_t *d, int length)
  {
    if (length ≤ 0) {
      hp→offset ← 0;
      return;
    }
    hp→offset ← find_newoffset(c, length);
    set_msg_length(c, hp→offset, length);
    write_msg_data(c, hp→offset, d, length);
  }
```

61. Replace a current message with new data, updating the message offset in the hash table if necessary. We can place a new message in the same slot as the older message only if the new message *length* ≤ *olength*. Otherwise, we allocate new space for the new message, and deactivate the old data slot. If the new message *length* ≤ 0, we deactivate any message which may currently be stored with the key.

```
⟨ Message Functions 53 ⟩ +≡
  static void msgreplace(BHTFile c, struct ht *hp, uint8_t *d, int length)
  {
    int olength;
    olength ← get_msg_length(c, hp→offset);
    if (length ≤ 0) {
      if (olength > 0)    /* Deactivate old */
        set_msg_length(c, hp→offset, −olength);
      return;
    }
    if (olength < 0) olength ← −olength;
    if (olength < length) {
      if (olength > 0) set_msg_length(c, hp→offset, −olength);    /* deactivate current */
      hp→offset ← find_newoffset(c, length);    /* create new */
    }
    set_msg_length(c, hp→offset, length);
    write_msg_data(c, hp→offset, d, length);
    if (olength > length) clear_msg_data(c, hp→offset, olength, length);
  }
```

62. Create a new message space, copy the old message to it, and append the new message.

```
⟨ Message Functions 53 ⟩ +≡
  static void msgappend(BHTFile c, struct ht *hp, uint8_t *d, int length)
  {
    int olength, nlength;
    uint32_t noffset;
    if (length ≤ 0) return;
    olength ← get_msg_length(c, hp→offset);
    if (olength > 0) {
      nlength ← olength + length;
      noffset ← find_newoffset(c, nlength);
      set_msg_length(c, hp→offset, −olength);    /* deactivate old space */
      copy_msg_data(c, hp→offset, noffset, olength);    /* copy current */
      write_msg_data(c, noffset + olength, d, length);    /* add new */
      set_msg_length(c, noffset, nlength);
      hp→offset ← noffset;
    }
    else msgnew(c, hp, d, length);
  }
```

63. Retrieve a message, allocating space for it. If the message data no longer exists at the current location, then return Λ. Otherwise return a pointer to the message, and set *mlength* to the message length. This data area should eventually be *free*( )'d. The message is null-terminated for convenience, though this is not strictly required.

```
⟨ Message Functions 53 ⟩ +≡
  static uint8_t *msgget(BHTFile c, struct ht *hp, int *mlength)
  {
    int length;
    uint8_t *p;
    length ← get_msg_length(c, hp→offset);
    if (length ≤ 0) return Λ;
    p ← bhtmalloc("msgget", length + 1);
    read_msg_data(c, hp→offset, p, length);
    p[length] ← '\0';
    if (mlength) *mlength ← length;
    return p;
  }
```

64. Hash and Key Functions. This hash function is a slight modification of the so-called FNV-1 (Fowler-Noll-Vo) hash function for 32 bits. The values in *HInit* and *HMult* are magic numbers determined by the algorithm authors. Rather than XORing the value of an 8-bit character directly into the hash value (as in FNV-1), this function looks up a 32-bit random number for the character which is XORed into the hash function. For this application, it provides slightly better hash dispersion than the plain FNV-1 algorithm.

A character $*p < first\_char$ or $*p > last\_char$ does not contribute to the hash value, though the multiply still occurs.

```
⟨ Utility Functions 7 ⟩ +≡
  static unsigned int mkhash(BHTFile c, uint8_t *p)
  {
    uint32_t hash ← c→HInit, i, thec;
    for (i ← 0; i < c→keysize; i++) {
      hash *= c→HMult;
      thec ← *p++;
      if ((thec ≥ c→first_char) ∧ (thec ≤ c→last_char)) hash ⊕= c→lvals[thec];
    }
    return hash % c→tabsize;
  }
```

65. We introduce this little structure to assist us in reading and writing the hash table to/from disk.

```
⟨ Small Structures 14 ⟩ +≡
  struct htbuff {
    uint8_t *barr, *ptr, *maxp;
    int count;
  };
```

66. Write the current hash table to disk. We must make sure that the hash table offsets are written in a non-platform-specific way. To do this, we allocate a temporary buffer of `THASHBUFFSZ`, then we pack data into the buffer from the hash table in memory, rearranging the memory-to-disk offset values in a prescribed manner, and then we write the hash table to the file one buffer at a time as it fills.

```
#define THASHBUFFSZ (1 ≪ 11)    /* Should be power of 2: 2^11 = 2048 */
⟨ File Functions 19 ⟩ +≡
  static void flush_hash(BHTFile c)
  {
    struct htbuff *hb;
    int i;
    hb ← new_htbuff();
    for (i ← 0; i < c→tabsize; i++) {
      if (c→htable[i]) {
        packstr(c, hb, c→htable[i]→key, c→keysize);
        packu32(c, hb, c→htable[i]→offset);
      } else packnull(c, hb, c→keysize);
    }
    cleanup_htbuff(c, hb, TRUE);
  }
```

67.
```
⟨ Forward Declarations 67 ⟩ ≡
  static void read_hash(BHTFile c);
  static void bht_log(char *fmt, ...);
```

This code is used in section 6.

68. We also need to be able to read in a hash table from an existing file on disk, so here's the writing process inverted.

```
⟨ File Functions 19 ⟩ +≡
  static void read_hash(BHTFile c)
  {
    struct htbuff *hb;
    uint8_t *kbuf ← bhtmalloc("read_hash", c→keysize);
    int i;

    hb ← new_htbuff();
    rdhtbuff_next(c, hb);
    for (i ← 0; i < c→tabsize; i++) c→htable[i] ← unpackkey(c, hb, kbuf, c→keysize);
    cleanup_htbuff(c, hb, FALSE);
    bhtfree("read_hash", kbuf);
  }
```

69. Allocate a new **struct htbuff** to organize the hash table for file input or output.

```
⟨ Utility Functions 7 ⟩ +≡
  static struct htbuff *new_htbuff(void)
  {
    struct htbuff *hb;

    hb ← bhtmalloc("new_htbuff", sizeof(struct htbuff));
    hb→barr ← bhtmalloc("new_htbuff", THASHBUFFSZ);
    hb→ptr ← hb→barr;
    hb→maxp ← hb→barr + THASHBUFFSZ;
    hb→count ← 0;
    return hb;
  }
```

70. Write a buffer full of converted hash table data out to disk.

```
⟨ Utility Functions 7 ⟩ +≡
  static void wrhtbuff_next(BHTFile c, struct htbuff *hb)
  {
    int retval;

    if ((retval ← setloc(c→fd, c→ht_off + (THASHBUFFSZ * hb→count))) ≡ BHT_OK)
      retval ← buff_to_disk(c→fd, hb→barr, THASHBUFFSZ);
    if (retval ≡ BHT_OK) {
      hb→count++;
      hb→ptr ← hb→barr;
    }
    else bht_log("Couldn't␣read␣hashtable␣buffer:␣err=%d\n", errno);
  }
```

71. Read a buffer-size chunk of the hash table from disk.

```
⟨ Utility Functions 7 ⟩ +≡
  static void rdhtbuff_next(BHTFile c, struct htbuff *hb)
  {
    int retval;
    if ((retval ← setloc(c→fd, c→ht_off + (THASHBUFFSZ * hb→count))) ≡ BHT_OK)
      retval ← disk_to_buff(c→fd, hb→barr, THASHBUFFSZ);
    if (retval ≡ BHT_OK) {
      hb→count++;
      hb→ptr ← hb→barr;
    }
    else bht_log("Couldn't␣read␣hashtable␣buffer:␣err=%d\n", errno);
  }
```

72. If we are writing the hash table, and if there are any data left in the buffer to be written out to disk, zero-fill the rest of the buffer and write it. (This step is not necessary if we are reading the hash table.) Then free the memory that we had allocated.

```
⟨ Utility Functions 7 ⟩ +≡
  static void cleanup_htbuff(BHTFile c, struct htbuff *hb, int writing)
  {
    if (writing ∧ (hb→ptr ≠ hb→barr)) {
      while (hb→ptr < hb→maxp) *(hb→ptr++) ← '\0';
      wrhtbuff_next(c, hb);
    }
    bhtfree("cleanup_htbuff", hb→barr);
    bhtfree("cleanup_htbuff", hb);
  }
```

73. Return TRUE if the string s of *length* is all Λs, or return FALSE otherwise.

```
⟨ Utility Functions 7 ⟩ +≡
  static inline int nullstr(const uint8_t *s, int len)
  {
    while (len--)
      if (*s++) return FALSE;
    return TRUE;
  }
```

74. Pack a string s of length *len* into the hash table buffer structure *hb*. If the buffer becomes full, flush it to disk, resetting the pointers to continue with the string packing.

```
⟨ Utility Functions 7 ⟩ +≡
  static inline void packstr(BHTFile c, struct htbuff *hb, uint8_t *s, int len)
  {
    while (len--) {
      *(hb→ptr++) ← *s++;
      if (hb→ptr ≥ hb→maxp) wrhtbuff_next(c, hb);
    }
  }
```

75. Convert a *val* to a 4-byte buffer for packing into a **struct htbuff**.

```
⟨ Utility Functions 7 ⟩ +≡
  static inline void packu32(BHTFile c, struct htbuff *hb, uint32_t val)
  {
    uint8_t b[4];
    b[0] ← (val ≫ 24) & #ff;
    b[1] ← (val ≫ 16) & #ff;
    b[2] ← (val ≫ 8) & #ff;
    b[3] ← val & #ff;
    packstr(c, hb, b, 4);
  }
```

76. Pack an empty key and offset into the buffer for writing.

```
⟨ Utility Functions 7 ⟩ +≡
  static inline void packnull(BHTFile c, struct htbuff *hb, int klen)
  {
    while (klen−−) {
      *(hb→ptr++) ← '\0';
      if (hb→ptr ≥ hb→maxp) wrhtbuff_next(c, hb);
    }
    packu32(c, hb, 0);
  }
```

77. Read a string from the **struct htbuff**.

```
⟨ Utility Functions 7 ⟩ +≡
  static inline void unpackstr(BHTFile c, struct htbuff *hb, uint8_t *sbuff, int slen)
  {
    while (slen−−) {
      *sbuff++ ← *(hb→ptr++);
      if (hb→ptr ≥ hb→maxp) rdhtbuff_next(c, hb);
    }
  }
```

78. Unpack a key from the buffer, and return a hash table slot surrounding it. If the key is empty, return Λ instead.

```
⟨ Utility Functions 7 ⟩ +≡
  static struct ht *unpackkey(BHTFile c, struct htbuff *hb, uint8_t *kbuf, int klen)
  {
    struct ht *nht;
    uint8_t b[4], *s, *d;
    uint32_t offset;

    unpackstr(c, hb, kbuf, klen);
    unpackstr(c, hb, b, 4);
    if (nullstr(kbuf, klen)) return Λ;
    offset ← (b[0] ≪ 24) | (b[1] ≪ 16) | (b[2] ≪ 8) | b[3];
    nht ← bhtmalloc("unpackkey", sizeof(struct ht));
    d ← nht→key ← bhtmalloc("unpackkey", klen + 1);
    s ← kbuf;
    while (klen −−) *d++ ← *s++;
    *d ← '\0';
    nht→offset ← offset;
    return nht;
  }
```

79. Create a new **struct ht**, and add the key information and data offset to it. (Because of *strndup*( ), it is assumed that the *key* does not contain Λs. This should probably be fixed.)

```
⟨ Message Functions 53 ⟩ +≡
  static struct ht *newhp(BHTFile c, uint8_t *key, uint8_t *data, int dlen)
  {
    struct ht *hp ← bhtmalloc("newhp", sizeof(struct ht));

    hp→key ← (uint8_t *) strndup((char *) key, c→keysize);
    msgnew(c, hp, data, dlen);
    return hp;
  }
```

80. User-accessible Functions. We'll start with some error codes as we begin to describe the user interface. As with system function calls, a 0 return generally indicates success. Negative return codes indicate some sort of error.

```
⟨ User Definitions 16 ⟩ +≡
#define BHT_ENOKEY (−6)
#define BHT_ENOACC (−5)
#define BHT_EUNSPEC (−4)
#define BHT_EFULL (−3)
#define BHT_EEXISTS (−2)
#define BHT_ERR (−1)
#define BHT_OK 0
```

81. In addition to error return codes, we have a few parameter options which a user can specify. The following are for the *bht_addkey*( ) function.

```
⟨ User Definitions 16 ⟩ +≡
#define BHT_EXISTERR 1
#define BHT_VALREPLACE 2
#define BHT_VALAPPEND 3
```

82. Write a key and its data to the file. The *flag* determines what should happen if the key already exists and has a assigned a value. If *flag* ≡ `BHT_EXISTERR`, then an error is returned if the key is already assigned. If *flag* ≡ `BHT_VALREPLACE`, the key's data is replaced with the new value. If *flag* ≡ `BHT_VALAPPEND`, then the data is added to the end of the current data string.

```
⟨User Functions 82⟩ ≡
  int bht_addkey(BHTFile c, uint8_t *key, uint8_t *data, int datalen, int flag)
  {
    uint32_t hv ← mkhash(c, key);
    int inc ← 0;
    if (¬c) return BHT_ERR;
    if (¬c→writer) return BHT_ENOACC;
    while (inc < c→tabsize ∧ c→htable[hv]) {    /* Hash slot has data */
      if (strncmp((char *) c→htable[hv]→key, (char *) key, c→keysize) ≡ 0) {    /* key exists */
        if (flag ≡ BHT_EXISTERR) return BHT_EEXISTS;
        else if (flag ≡ BHT_VALREPLACE) {
          msgreplace(c, c→htable[hv], data, datalen);
          return BHT_OK;
        } else if (flag ≡ BHT_VALAPPEND) {
          msgappend(c, c→htable[hv], data, datalen);
          return BHT_OK;
        } else return BHT_EUNSPEC;
      }
      hv++;
      inc++;
      if (hv ≥ c→tabsize) hv ← 0;
    }    /* Hash slot is empty or out of slots */
    if (inc ≥ c→tabsize) return BHT_EFULL;
    c→htable[hv] ← newhp(c, key, data, datalen);
    c→keycount++;
    return BHT_OK;
  }
```

See also sections 83, 84, 85, 86, 92, 94, 95, and 96.

This code is used in section 6.

83. Retrieve a key and its data from the file. If the key does not exist in the table, then a Λ pointer is returned, and *mlength* ← 0.

```
⟨ User Functions 82 ⟩ +≡
  uint8_t *bht_getkey(BHTFile c, uint8_t *key, int *mlength)
  {
    uint32_t hv;
    int inc ← 0;
    if (¬c) return Λ;
    hv ← mkhash(c, key);
    while (inc < c→tabsize ∧ c→htable[hv]) {      /* Hash slot has data */
      if (strncmp((char *) c→htable[hv]→key, (char *) key, c→keysize) ≡ 0)      /* key exists */
        return msgget(c, c→htable[hv], mlength);
      hv++;
      inc++;
      if (hv ≥ c→tabsize) hv ← 0;
    }
    if (mlength) *mlength ← 0;
    return Λ;
  }
```

84. We allow the user to step through all the keys in the hash table. This function returns the key value in a *malloc*( )'d string. The user should use *bht_getkey*( ) if s/he wants the data with it. The keys will not be returned in any discernable order. When the end of the table is reached, the function will return Λ one time, and then it will reset to retrieve the first key again.

```
⟨ User Functions 82 ⟩ +≡
  uint8_t *bht_nextkey(BHTFile c)
  {
    if (¬c) return Λ;
    if (c→seekkey < 0) c→seekkey ← 0;
    if (c→seekkey ≥ c→tabsize) {
      c→seekkey ← −1;
      return Λ;
    }
    while (c→htable[c→seekkey] ≡ Λ) {
      c→seekkey++;
      if (c→seekkey ≥ c→tabsize) {
        c→seekkey ← −1;
        return Λ;
      }
    }
    return (uint8_t *) strndup((char *) c→htable[c→seekkey++]→key, c→keysize);
  }
```

85. Reset the key-seeking mechanism so that the first key will be retrieved by the *bht_nextkey*( ) function.

```
⟨ User Functions 82 ⟩ +≡
  void bht_resetkey(BHTFile c)
  {
    if (c) c→seekkey ← −1;
  }
```

86. Delete a key from the hash file. This function is not compiled because this method of deleting a key will break subsequent searches for look-aside hash values. Though a delete function is necessary for completeness and general use, it is not immediately necessary for the problem we are trying to solve.

Problem description: keys are hashed to an index, and if that index does not already contain an element, the key is inserted at that index. However, if an element already exists in that slot, then a linear search for an empty slot begins at the next higher index value and continues (with wrapping) until an empty slot is found (unless the table is already full). For searches, we calculate a hash value and begin searching for keys at that index, halting our search when we encounter an empty slot. If we allow that keys may be deleted, then a linear search may fail pre-maturely, since an empty slot may have been created between a key's original hash value and its actual resting index.

I have considered two solutions to this problem. The first solution is to keep the deleted key in the table, but mark it inactive. In this way, a search for that particular key will fail, but linear searches for look-aside keys would continue normally. The active/inactive flag would need to be preserved to disk as well, unless the entire hash table were to be rebuilt.

Another solution is to change the key storage strategy so that keys that hash to the same index would be kept in a list structure with a root at the hash index. Two different disk-storage strategies then come to mind. At present, the entire hash table including empty slots is written to disk. Instead, one strategy could be that the hash index is stored along with the key. Another strategy could be that no hash index is stored at all, and new hash values are calculated each time the disk file is read. The advantage of this second idea is that the size of the hash table could change between file reads, with the disadvantage that reading the hash table requires significantly more overhead.

While dealing with a dynamically-sized hash table is appealing at some levels, there are implications for data area storage. Should the hash table grow too large, then the data area region of the file would need to be moved, which could impose a substantial penalty in disk I/O. Not only that, but offsets are currently stored as absolute values, not relative values. Every file offset would need to be recalculated if the data area had to be moved. (This certainly argues in favor of relative file offsets for the data area, however.)

```
⟨User Functions 82⟩ +≡
#if 0
  int bht_delkey(BHTFile c, uint8_t *key)
  {
    uint32_t hv;
    int inc ← 0, mlen;
    if (¬c) return BHT_ERR;
    if (¬c→writer) return BHT_ENOACC;
    hv ← mkhash(c, key);
    while (inc < c→tabsize ∧ c→htable[hv]) {      /* Hash slot has data */
      if (strncmp(c→htable[hv]→key, key, c→keysize) ≡ 0) {      /* key exists */
        ⟨Release key resources 87⟩
        return BHT_OK;
      }
      hv++;
      inc++;
      if (hv ≥ c→tabsize) hv ← 0;
    }
    return BHT_ENOKEY;
  }
#endif
```

87. Deactivate the selected key's message data, and then remove the key information from the slot.

⟨ Release key resources 87 ⟩ ≡

```
mlen ← get_msg_length(c, c→htable[hv]→offset);
if (mlen > 0) set_msg_length(c, c→htable[hv]→offset, −mlen);
bhtfree("bht_delkey", c→htable[hv]→key);
bhtfree("bht_delkey", c→htable[hv]);
c→htable[hv] ← Λ;
```

This code is used in section 86.

88. File-handling Functions. Create a function for writing log output in a standard way.

⟨ File Functions 19 ⟩ +≡

```
static void bht_log(char *fmt, ...)
{
  va_list ap;
  fprintf(stderr, "BHT:␣");
  va_start(ap, fmt);
  vfprintf(stderr, fmt, ap);
  va_end(ap);
}
```

89. Open a file for writing and lock the file as well, or return an error if the file could not be opened *and* locked. This only applies to files that are being opened for writing. We hope that read locks are not required.

⟨ File Functions 19 ⟩ +≡

```
static int open_and_lock(const char *fn, int flags, int perm)
{
  int fd, retval;
  struct flock fl;
  fd ← open(fn, flags, perm);
  if (fd < 0) return fd;
  fl.l_type ← F_WRLCK;
  fl.l_whence ← SEEK_SET;
  fl.l_start ← 0;
  fl.l_len ← 0;
  retval ← fcntl(fd, F_SETLK, &fl);
  if (retval < 0) {
    close(fd);
    return BHT_ERR;
  }
  return fd;
}
```

90. To create a BHT file, we first open it and fill in the structure information based on the parameters passed to us. Once we've built the basic components, we flush the parameters and the empty hash table out to disk.

```
⟨ File Functions 19 ⟩ +≡
  static BHTFile bht_create(const char *fn, int keysize, int tablesize, int nbuffs)
  {
    BHTFile c;
    int fd;
    fd ← open_and_lock(fn, O_RDWR | O_CREAT, °666);
    if (fd < 0) return Λ;
    c ← bht_init(keysize, tablesize, nbuffs);
    if (¬c) return Λ;
    c→fd ← fd;
    flush_parameters(c);
    flush_hash(c);
    fdatasync(fd);
    return c;
  }
```

91. The following values are flags for the *bht_open*( ) function. Semantically, BHT_READ and BHT_WRITE are mutually exclusive states. If both are specified, then BHT_WRITE will be assumed. BHT_CREAT and BHT_EXCL may be bitwise-OR'd with BHT_WRITE.

```
⟨ User Definitions 16 ⟩ +≡
#define BHT_READ #0001
#define BHT_WRITE #0002
#define BHT_CREAT #0004
#define BHT_EXCL #0008
```

92. Open (or create) a **BHTFile** for processing. The variable *flags* will contain one of `BHT_READ` or `BHT_WRITE`. Only one process may hold `BHT_WRITE` on a file, but a `BHT_READ` process may have a file open while a writing process is operational. For this reason, an advisory write lock (exclusive lock) is placed on the file when opened for writing, but if opened for reading, no lock checking takes place. If `BHT_WRITE` is specified, then it may be logically OR'd with `BHT_CREAT` and `BHT_EXCL`. If the file does not exist and `BHT_CREAT` is specified, then a new file will be created. If the file does exist and `BHT_EXCL` and `BHT_CREAT` have been specified, then the open will not succeed and Λ is returned. Any other error on open will return Λ as well. On successful open or create, a **BHTFile** structure is returned.

```
⟨ User Functions 82 ⟩ +≡
  BHTFile bht_open(const char *fn, int flags, int keysize, int tablesize, int nbuffs)
  {
    struct stat sbuf;
    int fd, retval;
    retval ← stat(fn, &sbuf);
    if (retval < 0) {
      if (errno ≡ ENOENT) {    /* doesn't exist */
        if ((flags & BHT_WRITE) ∧ (flags & BHT_CREAT)) return bht_create(fn, keysize, tablesize, nbuffs);
        else return Λ;    /* not instructed to create */
      } else return Λ;    /* Some other error: no recovery */
    } else {    /* exists */
      if ((flags & BHT_WRITE) ∧ (flags & BHT_CREAT) ∧ (flags & BHT_EXCL)) return Λ;
          /* but we wanted to create a new file */
    }
    if (flags & BHT_WRITE) fd ← open_and_lock(fn, O_RDWR, 0);
    else fd ← open(fn, O_RDONLY);
    if (fd < 0) return Λ;
    return bht_read(fd, (flags & BHT_WRITE) ? TRUE : FALSE, nbuffs);
  }
```

93. Cleanup the storage allocated for the **BHTFile** structure.

```
⟨ Utility Functions 7 ⟩ +≡
  static void bht_cleanup(BHTFile c)
  {
    struct buff *b, *ob;
    int i;    /* Free the Hash table entries */
    for (i ← 0; i < c→tabsize; i++)
      if (c→htable[i]) {
        bhtfree("bht_cleanup(1)", c→htable[i]→key);
        bhtfree("bht_cleanup(2)", c→htable[i]);
      }    /* Free the hash table */
    bhtfree("bht_cleanup(3)", c→htable);    /* Free the character scramble */
    bhtfree("bht_cleanup(4)", c→lvals);    /* Free the buffers */
    b ← c→y;
    while (b) {
      bhtfree("bht_cleanup(5)", b→data_area);
      ob ← b;
      b ← b→o;
      bhtfree("bht_cleanup(6)", ob);
    }    /* Free the BHTFile structure */
    bhtfree("bht_cleanup(7)", c);
  }
```

94. Flush the current hash table and data out to the file.

⟨ User Functions 82 ⟩ +≡

```
void bht_flush(BHTFile c)
{
  if (¬c ∨ ¬c→writer) return;
  flush_buffers(c);
  flush_hash(c);
  flush_parameters(c);
  fdatasync(c→fd);
}
```

95. Close an open file.

⟨ User Functions 82 ⟩ +≡

```
void bht_close(BHTFile c)
{
  if (¬c) return;
  if (c→writer) bht_flush(c);
  close(c→fd);
  c→fd ← −1;
  c→writer ← FALSE;
  bht_cleanup(c);
}
```

96. Return the quantity of active keys currently stored in the hash table.

⟨ User Functions 82 ⟩ +≡

```
int bht_keycount(BHTFile c)
{
  if (¬c) return BHT_ERR;
  else return c→keycount;
}
```

97. We provide forward declarations of the user-accessible functions.

⟨ User Definitions 16 ⟩ +≡

```
extern BHTFile bht_open(const char *path, int flags, int keysz, int tabsz, int nbuffs);
extern void bht_flush(BHTFile c);
extern void bht_close(BHTFile c);
extern int bht_addkey(BHTFile c, uint8_t *key, uint8_t *data, int dlen, int flag);
extern uint8_t *bht_getkey(BHTFile c, uint8_t *key, int *len);
extern uint8_t *bht_nextkey(BHTFile c);
extern void bht_resetkey(BHTFile c);
extern int bht_keycount(BHTFile c);
```

98. Testing. Create 4-character keys and and populate the file until the hash table is full. Continue to add data to the existing keys after that. Then extract all the keys and verify that the data have not been corrupted. Start with the important include files.

⟨ bhtftest.c 98 ⟩ ≡

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "bhtf.h"
```

See also sections 99, 100, and 101.

99. Test Parameters.

```
⟨ bhtftest.c 98 ⟩ +≡
#define TEST_FILE "testfile.bht"
#define REPEAT_COUNT 5
#define TABLE_SIZE 5033
#define NBUFFS 4
```

100. When we add data to the file, we request that the test string be appended to the current data. When we read keys back for verification purposes, there should be an integral number of copies of the test string. We verify that here.

```
⟨ bhtftest.c 98 ⟩ +≡
  int verify_data(char *s, char *v, int len, int count)
  {
    int i, j, vlen ← strlen(v);
    char *st;
    for (j ← 0; j < count; j++) {
      st ← &(s[j * vlen]);
      for (i ← 0; i < vlen; i++)
        if (st[i] ≠ v[i]) {
          fprintf(stderr, "At␣%d␣expected␣'%c',␣found␣'%c'\n", j * vlen + i, v[i], st[i]);
          return 1;
        }
    }
    return 0;
  }
```

101. Main Program.

```
⟨ bhtftest.c 98 ⟩ +≡
  main(int argc, char *argv[])
  {
    BHTFile hf;
    char *m, key[4], a, b, c, teststr[256];
    int len, retval, i, repcount ← REPEAT_COUNT;
    hf ← bht_open(TEST_FILE, BHT_WRITE | BHT_CREAT, 4, TABLE_SIZE, NBUFFS);
    if (¬hf) {
      fprintf(stderr, "Couldn't␣open␣BHT␣file␣%s\n", TEST_FILE);
      exit(1);
    }
    for (i ← 0; i ≤ 186; i++) teststr[i] ← (i & #1) ? '␣' : (char)(33 + (i ≫ 1));
    teststr[i++] ← '\n';
    teststr[i] ← '\0';
    for (i ← 0; i < repcount; i++) {
      for (a ← 'A'; a ≤ 'Z'; a++) {
        key[0] ← key[1] ← a;
        for (b ← 'A'; b ≤ 'Z'; b++) {
          key[2] ← b;
          for (c ← 'A'; c ≤ 'Z'; c++) {
            key[3] ← c;
            retval ← bht_addkey(hf, key, teststr, strlen(teststr), BHT_VALAPPEND);
            if (retval ≠ BHT_OK) {
              fprintf(stderr, "\nKeys␣added␣=␣%d\n", bht_keycount(hf));
              a ← b ← c ← 'Z';      /* terminate all loops */
            }
          }
        }
      }
    }
    for (a ← 'A'; a ≤ 'Z'; a++) {
      key[0] ← key[1] ← a;
      for (b ← 'A'; b ≤ 'Z'; b++) {
        key[2] ← b;
        for (c ← 'A'; c ≤ 'Z'; c++) {
          key[3] ← c;
          m ← bht_getkey(hf, key, &len);
          if (m) {
            if (verify_data(m, teststr, len, repcount) ≠ 0) {
              fprintf(stderr, "key␣%c%c%c%c␣has␣bad␣data:\n%s\n", a, a, b, c, m);      /* exit(1); */
            }
            free(m);
          } else {
            fprintf(stderr, "Couldn't␣retrieve␣key␣%c%c%c%c\n", a, a, b, c);
            a ← b ← c ← 'Z';      /* terminate all loops */
          }
        }
      }
    }
    bht_close(hf);
  }
```

102. Index.

Bret's Hash Table Functions

(Version 0.2)

Bret D. Whissel