COS 217 Midterm Exam

Princeton University, Spring 2005

March 9, 2005

This exam is **closed book, closed notes**. You have **50 minutes**.

When writing program code, don't bother with comments.
Use the "paranoid" style of assertions.

Pages 6 and 7 are a copy of the one-line Emacs program.

	Score	Possible
1		5
2		9
3		10
4		10
5		6
6		5
7		5
TOTAL		50

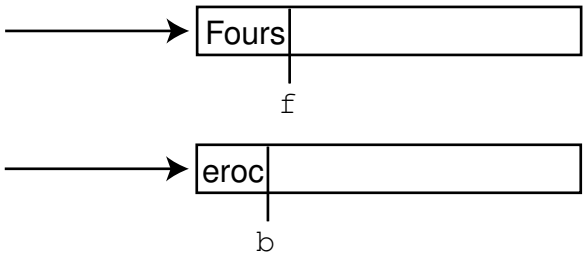
Your name:

Unix login:

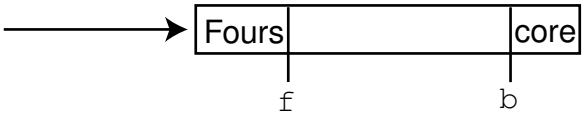
Precept: Monday Afternoon Tuesday Noon Tuesday Afternoon

Write and sign the honor pledge:

In lecture I discussed a data structure for Emacs buffers that looks like this:



A more efficient data structure (using half as much memory to accomplish the same thing) would like like this:



1) Write the definition of `struct buffer` for the new data structure.

2) Write the definition of the INVARIANTS for the new data structure.

```
#define INVARIANTS(buf)
```

3) Write the implementation of `Buffer_forward` for the new data structure.

```
void Buffer_forward(Buffer_T b) {




}
```

4) Write a function for the new data structure that builds a new buffer with the contents of an (already-opened) text file.

5) Suppose we wish make a new SymTable module whose keys are not strings, but buffers. The new buffer-SymTable should work as much like string-SymTable as possible, but it should be a client of the Buffer module.

Two buffers should be considered equivalent if they contain the same text, regardless of the cursor position. Assume that, during the time a buffer is in the SymTable as a key, the client may perform "forward" and "back" operations on that buffer but will not do "insert" and "delete."

Indicate what changes are necessary in the interface by marking up the old interface:

```
SymTable_T SymTable_new(void);


void SymTable_free(SymTable_T oSymTable);


unsigned int SymTable_getLength(SymTable_T oSymTable);


int SymTable_put(SymTable_T oSymTable, const char *pcKey, const void *pvValue);


int SymTable_remove(SymTable_T oSymTable, const char *pcKey);


int SymTable_contains(SymTable_T oSymTable, const char *pcKey);


void *SymTable_get(SymTable_T oSymTable, const char *pcKey);


void SymTable_map(SymTable_T oSymTable,
   void (*pfApply)(const char *pcKey, void *pvValue, void *pvExtra),
   const void *pvExtra);
```

6) Recommend and justify a memory-management "ownership" policy for buffers.

7) Modify this hash-function implementation as necessary for the implementation of buffer symbol tables.

```
#define HASH_MULTIPLIER 65599


static unsigned int hash(const char *key)  {


   int i;


   unsigned int h = 0;


   for (i = 0; key[i] != '\0'; i++)


      h = h * HASH_MULTIPLIER + key[i];


   return h;


}
```

The one-line Emacs program

buffer.h

```
typedef struct buffer *Buffer_T;

Buffer_T Buffer_new(void);

void Buffer_insert(Buffer_T b, char c);
void Buffer_delete(Buffer_T b);

void Buffer_forward(Buffer_T b);
void Buffer_back(Buffer_T b);

int  Buffer_size(Buffer_T b);
int  Buffer_pos(Buffer_T b);
char Buffer_getchar(Buffer_T b, int index);
```

client.c

```
#include <stdio.h>
#include <assert.h>
#include <ctype.h>
#include "buffer.h"

#define CONTROL(x) ((x) & 31)

void display(Buffer_T buf) {
  int min = Buffer_pos(buf) < 20 ?
              0 : Buffer_pos(buf)-20;
  int max = Buffer_size(buf)-Buffer_pos(buf) < 20
             ? Buffer_size(buf)
             : Buffer_pos(buf)+20;
  int i;
  for (i=min; i<max; i++) {
    if (i==Buffer_pos(buf))
      putchar('|');
    putchar(Buffer_getchar(buf,i));
  }
  if (i==Buffer_pos(buf))
    putchar('|');
  putchar('\n');
}

void run(void) {
  Buffer_T buf = Buffer_new();
  for(;;) {
    char c = getchar();
    switch(c) {
    case CONTROL('f'):
      Buffer_forward(buf);
      break;
    case CONTROL('b'):
      Buffer_back(buf);
      break;
    case CONTROL('d'):
      Buffer_delete(buf);
      break;
    case CONTROL('h'):
      Buffer_back(buf);
      Buffer_delete(buf);
      break;
    case CONTROL('c'):
      return;
    default:
      if (isprint(c))
        Buffer_insert(buf,c);
    }
    display(buf);
  }
}

int main(int argc, char **argv) {
  et cetera ...
}
```

buffer2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "buffer.h"

#define CAPACITY  1000

struct buffer {
  int frontsize, backsize;
  char *front, *back;
  int f, b;
};

Buffer_T Buffer_new(void) {
  Buffer_T buf = (Buffer_T) malloc (sizeof *buf);
  assert(buf);
  buf->frontsize = buf->backsize = CAPACITY;
  buf->front = (char *) malloc (buf->frontsize);
  assert(buf->front);
  buf->f = buf->b = 0;
  buf->back = (char *) malloc (buf->backsize);
  assert(buf->back);
  return buf;
}

void Buffer_insert(Buffer_T buf, char c) {
  assert(buf);
  assert (buf->f + 1 < buf->frontsize);
  buf->front[buf->f++] = c;
}

void Buffer_delete(Buffer_T buf) {
  assert(buf);
  assert (buf->b > 0);
  --(buf->b);
}

void Buffer_forward(Buffer_T buf) {
  assert(buf);
  assert(buf->f + 1 < buf->frontsize);
  assert(buf->b > 0);
  buf->front[buf->f++] = buf->back[--(buf->b)];
}

void Buffer_back(Buffer_T buf) {
  assert(buf);
  assert(buf->f > 0);
  assert(buf->b + 1 < buf->backsize);
  buf->back[buf->b++] = buf->front[--(buf->f)];
}

int  Buffer_size(Buffer_T buf) {
  assert(buf);
  return (buf->f + buf->b);
}

char Buffer_getchar(Buffer_T buf, int index) {
  assert(buf);
  assert(index >= 0 && index < buf->f + buf->b);
  if (index < buf->f)
    return (buf->front[index]);
  else return (buf->back[buf->f + buf->b - 1 - index]);
}

int  Buffer_pos(Buffer_T buf) {
  assert(buf);
  return (buf->f);
}
```