COS 217 Midterm Exam

Princeton University, Fall 2003

October 23, 2003

This exam is **open-book, open notes** (but no computers!).
You have **two hours**.

	Score	Possible
1		30
2		25
3		25
4		20
Total		100

Your name:

Unix login:

Precept: Monday Morning   Monday Afternoon   Tuesday Afternoon

Write and sign the honor pledge:

1 Find the bugs

This program reads a list of strings from its standard input, inserts each one into a linked list (kept in sorted order), and then prints out the sorted list.

There are 6 bugs in this program. Correct the bugs.

(Note: gcc doesn't notice any problem with this program; it's syntactically correct.)

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>

#define MAX_STRING_LENGTH 10

struct node {
  char *str;
  struct node *next;
};

/* Read a line of stdin into output; return NULL if end-of-file */
char * read_string() {
  size_t n = 0;
  int c;
  char str[MAX_STRING_LENGTH];
  char *output;

  /* read the input into a temporary buffer */
  for (n = 0; n < MAX_STRING_LENGTH; n++) {
    c = getchar();
    if (c == '\n' || c == EOF) break;
    str[n] = (char) c;
  }

  if (c==EOF && n==0) return NULL;

  /* Allocate the output buffer, and copy the string there */
  output = malloc(strlen(str) + 1);
  assert(output != NULL);
  strcpy(output, str);

  /* Discard any remaining characters on the line. */
  while ((c != '\n') && (c != EOF)) c = getchar();

  return output;
}
```

```c
int main(int argc, char **argv) {
  char *line;
  struct node *first_node = NULL, *pn;

  for(line=read_string(); line; line=read_string()) {
    struct node *new_node = malloc(sizeof(struct node *));
    assert(new_node != NULL);

    new_node->str = line;

    if (first_node == NULL) {
      /* case 1, this is the first node */
      new_node->next = NULL;
      first_node = new_node;
    } else if (strcmp(new_node->str, first_node->str) == 0) {
      /* case 2: new_node should be the first string in the list */
      new_node->next = first_node;
      first_node = new_node;
    } else { /* all other cases */
      for (pn = first_node; ; pn = pn->next) {
        if (strcmp(new_node->str, pn->next->str) <= 0 || pn->next == NULL) {
          new_node->next = pn->next;
          pn->next = new_node;
          break;
        }
      }
    }
  }

  /* Now print all strings so far in sorted order. */
  for (pn = first_node; pn; pn = pn->next) {
    printf("%s\n", pn->str);
    free(pn->str);
    free(pn);
  }

  return 0;
}
```

2 Interface design

One way to improve alpha-beta search is to keep track of positions you've evaluated before. If you come across such a position, then instead of doing a full search to depth K, you can just use the value that you computed before. Here's how the alpha-beta function would be organized. Except for the parts in *italics*, it's just like what's in the lecture notes.

search(board, α, β, depth) {
    Move moves[ ]; Delta deltas[ ];
    If we've seen the board looking just like this before, and
    at that time we computed an evaluation v for it, return v.
    if (gameOver() || depth > LIMIT) return eval(board)
    if N is a Min node then
        genMoves(board, moves)
        for each m in moves
            expandMove(board, m, deltas)
            for each d in deltas do applyDelta(board, d);
            $\beta \leftarrow$ Min(β, search(board, α, β, depth+1));
            for each d in deltas do unApplyDelta(board, d);
            if $\alpha \geq \beta$ then {$\beta \leftarrow \alpha$; break;}
        Record the fact that this state of the game board evaluates to β.
        return β;
    else (N is a Max node)
        do similar stuff...
}

A. Design the header file of an abstract data type that can keep track of the previously seen game states. Include not only the two functions that the search function would need to call, but other functions that would almost certainly be necessary. Don't go overboard with extra frills.

B. Show the C code that the search function would use to implement the following two operations:

1. If we've seen the board looking just like this before, and at that time we computed an evaluation v for it, return v.

2. Record the fact that this state of the game board evaluates to β.

C. Inside the .c file of an *efficient* implementation of the ADT there will be some data-structure declarations and some function implementations. Show the data-structure declarations in C code. *Don't show the functions!*

3   Abstract Data Types

This question explores design alternatives for a SymTable ADT. *Be brief. Answer each part in 20 words or less. Complete sentences are **not** necessary.*

(a) Describe what it means for a SymTable object to "own" its keys.

(b) State an advantage of designing a SymTable ADT so a SymTable object owns its keys.

(c) To illustrate your answer to (b), show a fragment of client code that is problematic if a SymTable object does not own its keys.

(d) State an advantage of designing a SymTable ADT so a SymTable object *does not* own its keys.

(e) Under what circumstances could one design a SymTable ADT so a SymTable object owns its *values*?

(f) Suppose you wish to generalize the *linked list* version of your SymTable ADT so its keys are not necessarily strings, but instead could be data of any type. Show the declarations of the SymTable_new and SymTable_get functions that would appear in the generalized ADT's interface (.h) file. *Hint: What function should the SymTable_get function (and other SymTable functions) call to compare keys? Who would provide it?*

(g) Suppose you wish to generalize the *hash table* version of your SymTable ADT so its keys are not necessarily strings. Could that SymTable ADT use the same interface as the linked list version that you (partially) wrote in (f)? Explain.

4 Incremental Evaluation

This question is worth less than the others and might take a lot of time. Don't work on it until you're happy with your answers to the other questions.

Below you will find part of a program to play Checkers. Checkers is played on a board of 8x8 squares. The RED pieces start on rows 0–2 and try to get to row 7. To make the exam easier, there will be no Kings, just ordinary pieces. The BLACK pieces start on rows 5–7 and try to get to row 0. Pieces can be captured, and you lose the game by losing all your pieces.

I won't explain the rules of Checkers here, because you don't need to know them all. Instead, I will explain some simple board-evaluation heuristics:

1. More pieces is better.

2. A piece with "protection" is better. Protection means either that a piece is on the extreme left or right side of the board, or that there is a piece of the same color diagonally adjacent.

3. It's good to block the opponent from getting into his last row, which is your first row, so you get a bonus proportional to the the square of the number of your own pieces occupying your own first row.

These heuristics are implemented in the `eval` function below.

A `Delta` is represented as "before,after;" that is, what was in the square before, and what goes in after.

Your task is to implement the eval function in an incremental way. That is, move some functionality from `eval` to `applyDelta` so that `eval` doesn't have to examine every square of the board every time it's called. Your new implementation should compute exactly the same results as mine.

There is also a function `unApplyDelta`. Presumably, whatever adjustements you make to `applyDelta` would have to be made to `unApplyDelta` as well, but I'm not asking you to show these.

As a general guideline, your revised implementation should still call the `protection` function, which should be unchanged.

Note: By multiplying the value of each heuristic by a piece color (e.g., `v += p * protection(...)`), we count things positive for RED and negative for BLACK, so the `eval` function shows how good the position is from Red's point of view.

Hint: There are three separate evaluation heuristics (each piece worth a point, protection, base-row). Clearly divide your applyDelta function into these three parts, and it'll be easier for us to give you partial credit for the ones you get right.

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

typedef enum {NONE=0, RED=1, BLACK= -1} Piece;

typedef struct checkerboard {
 Piece square [8][8];
} * Board;

double protection(Board b, Piece p, int row, int col) {
  /* You don't need to read the body of this function! */
  if (col==0 || col==7)
    return 0.4;
  return 0.2 * (b->square[row-1][col-1]==p || b->square[row+1][col+1]==p)
       + 0.2 * (b->square[row+1][col-1]==p || b->square[row-1][col+1]==p);
}

void applyDelta(Board b, int row, int col, Piece before, Piece after) {
  assert (b->square[row][col]==before);
  b->square[row][col]==after;
}

void unApplyDelta(Board b, int row, int col, Piece before, Piece after) {
  assert (b->square[row][col]==after);
  b->square[row][col]==before;
}

double eval (Board b) {
  int row, col;
  double v=0.0;
  int red_base=0, black_base=0;
  for (row=0; row<8; row++)
    for (col=0; col<8; col++) {
      Piece p = b->square[row][col];
      if (p != NONE) {
        v += p;
        v += p * protection(b,p,row,col);
      }
    }
  for(col=0;col<8;col++) {
    red_base   += (b->square[0][col]==RED);
    black_base += (b->square[7][col]==BLACK);
  }
  v += 0.01 * RED   * red_base   * red_base;
  v += 0.01 * BLACK * black_base * black_base;
  return v;
}
```

```
typedef struct {
 Piece square [8][8];




} * Board;

void applyDelta(Board b, int row, int col, Piece before, Piece after) {
  assert (b->square[row][col]==before);




  b->square[row][col]==after;




}
```

```
double eval (Board b) {



}
```