Homework 3 Answers

Due Date: Monday, May 13, 2002, at 11:59PM
Points: 100

UNIX System

1. (10 points) What is the parent directory of /?
   Answer: If you change to the / directory, and go to “..”, you will find yourself back in /. Hence / is its own parent.

2. (15 points) The file /usr/share/dict/words contains a list of English words and abbreviations, one per line. How many words and abbreviations does it have? List all words in it with the trigram "gry" in them.
   Answer: The command is
   
   grep gry$ /usr/share/dict/words
   
   and the words are “angry” and “hungry”.

C Programming

3. (30 points) Write a program that reads words from the standard input and prints them in sorted order (use ASCII ordering). Use a linked list to do this. The structure of a node in the linked list is to be:

   struct lnode {
   char *word;      /* pointer to word */
   struct lnode *nxt;    /* pointer to next entry in linked list */
   }

   You will need to use the function malloc(3) to allocate both space for the word and space for the nodes. If a word occurs more than once, list each occurrence separately. A “word” is a maximal sequence of letters and digits.

   Your program should print one word per line. For example:

   Sample stdin  Corresponding stdout
   Hello, there, my old friend!    Goodbye
   How are you today?    Hello
   I am very well, thank you!    How
   Goodbye ...

   Answer:

   /*
   * A Word Listing Program
   *
   * Problem:
   * Given input, print all the words, one per line, sorted.
   * A "word" is a maximal sequence of alphanumeric characters.
   */
/*
* Version 1.0* Matt Bishop (bishop@cs.ucdavis.edu)
*/
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

/*
* macros
*/
#define SZBUFFER 1024 /* max length of an input line */

/*
* structure for the list
*/
struct word {
    char *data; /* data field (the word to be sorted) */
    struct word *next; /* points to next element in linked list */
    /* (NULL pointer if no next element) */
};

/*
* pointer to the first element (the head) of the list
* NULL if there's nothing in the list
*/
struct word *head = NULL;

/*
* forward declarations (prototypes)
*/
struct word *mknode(char *); /* create new node in linked list */
struct word *linsert(struct word *); /* insert new node in linked list */
void lprint(struct word *); /* print the tree */

/*
* this is the main routine
* arguments: ignored
* return: exits with 0
* function: read from stdin, break each line into words,
* add words to linked list, and when input ends
* print the words
* exceptions: none
*/
int main(void)
{
    char line[SZBUFFER]; /* input line */
    char word[SZBUFFER]; /* word */
    register char *p; /* current line position */
    register char *w; /* used to load line */
    struct word *n; /* points to new node for word */

    /*
* read the file a line at a time
*/
while(fgets(line, SZBUFFER, stdin) != NULL){
    for(p = line; *p; ){
        while (*p && !isalnum(*p))    p++;
        *w++ = *p++;
        *w = '0';
        if (word[0]){ if((n = mknode(word)) == NULL) perror(word); else head = linsert(n); }
    }
}

struct word *mknode(char *n) { struct word *p; /* pointer to new space */

    /* sanity check */
    if (n == NULL)
        return(NULL);

    /* create a new node
     * and initialize the two fields
     */
    return(p = malloc(sizeof(struct word)));

    p->word = (char *) malloc(sizeof(n) + 1);
    strcpy(p->word, n);
    p->count = 1;
    p->left = p->right = NULL;

    return(p);

    /* print the tree
     */
    lprint(head);

    /* say goodbye nicely
     */
    return(0);
/* create the element, reporting errors */
if ((p = malloc(sizeof(struct word))) == NULL)
    return(NULL);

/* create the word, reporting errors */
if ((p->data = malloc(strlen(n)+1)) == NULL)
    return(NULL);

/* initialize the element */
(void) strcpy(p->data, n);
p->next = NULL;

/* return a pointer to the new entity */
return(p);
}

/*
 * insert the element that new points to into the linked list,
 * and return a pointer to the (possibly new) head of the list
 */
struct word *linsert(struct word *new)
{
    struct word *prev, *temp; /* previous, current nodes */

    /* empty list: put head at the front */
    if (head == NULL)
        return(new);

    /* it goes before the first element */
    if (strcmp(head->data, new->data) > 0){
        new->next = head;
        return(new);
    }

    /* now walk the list
     * from here on in, prev->next == temp
     * we'll insert after prev and before temp
     */
    prev = head;
temp = head->next;
while(temp != NULL && strcmp(temp->data, new->data) < 0){
    /* advance prev and temp */
    prev = temp;
temp = temp->next;
}

    /* here's the insertion
     * make prev->next the new element
     * and new->next the one temp points to
     */
    new->next = temp;
    prev->next = new;
4. (30 points) Please make two modifications to the program you wrote for part 3:

   a. Change the program so the user can name one or more files on the command line, and the program will take input from those files. If no files are named, the program should read from the standard input.

   b. Add a command-line option \(-r\) that causes the words to be printed in reverse order. (Hint: use recursion.)

Answer: The answers to a and b are combined in the following program:

   /*
   * A Word Listing Program
   *
   * Problem:
   *  Given input, print all the words, one per line, sorted.
   *  A "word" is a maximal sequence of alphanumeric characters.
   *
   * Version 1.0* Matt Bishop (bishop@cs.ucdavis.edu)
   * Version 1.1* take command line args and the -r (reverse) option
   */
   #include <stdio.h>
   #include <ctype.h>
   #include <stdlib.h>

   /*
   * macros
   */
   #define SZBUFFER 1024    /* max length of an input line */

   /*
   * structure for the list
   */
struct word {
    char *data; /* data field (the word to be sorted) */
    struct word *next; /* points to next element in linked list */
    /* (NULL pointer if no next element) */
};

/*
 * pointer to the first element (the head) of the list
 * NULL if there's nothing in the list
 */
struct word *head = NULL;

/*
 * set to 1 if printing in reverse order, 0 if not
 */
int reverse = 0;

/*
 * forward declarations (prototypes)
 */
void dofile(FILE *); /* process a file */
struct word *mknode(char *); /* create new node in linked list */
struct word *linsert(struct word *); /* insert new node in linked list */
void lprint(struct word *); /* print the tree */

/*
 * this is the main routine
 *
 * arguments: option -r to reverse, rest are file names
 * return: exits with 0
 * function: set order of sort, open files and hand them to dofile
 *           for processing, print linked list at end
 * exceptions: none
 */
int main(int argc, char *argv[])
{
    int curarg = 1; /* number of current argument */
    FILE *fp; /* input file pointer */

    /*
    * check for -r option
    */
    if (argc > 1 && strcmp(argv[curarg], "-r") == 0){
        /* yep -- reverse list and skip this arg now */
        reverse = 1;
        curarg++;
        argc--;
    }

    /*
    * figure out where to get input:
    * if no files, stdin; else, files
    */
if (argc == 1) {
    /* from standard input */
    dofile(stdin);
}
else{
    /* from the argument list; walk it */
    for( ; argv[curarg] != NULL; curarg++){
        /* open file and report error */
        if ((fp = fopen(argv[curarg], "r")) == NULL)
            perror(argv[curarg]);
        else{
            /* worked -- process file */
            dofile(fp);
            (void) fclose(fp);
        }
    }
}

/*
 * print the tree
 */
lprint(head);
return(0);
}

/*
 * function:    dofile
 * arguments:   ignored
 * return:      ignored
 * function:    read from file pointer, break each line into words,
 *               add words to linked list
 * exceptions:  none
 */
void dofile(FILE *fp) {
    char line[SZBUFFER];  /* input line */
    char word[SZBUFFER];  /* word */
    register char *p;    /* current line position */
    register char *w;    /* used to load line */
    struct word *n;      /* points to new node for word */

    /*
     * read the file a line at a time
     */
    while(fgets(line, SZBUFFER, fp) != NULL){
        /*
         * begin at the beginning of a new line
         * and loop until end of line
         */
        for(p = line; *p; ){
            /*
             * skip leading non-word stuff
             */
            while (*p && !isalnum(*p))
p++;  
/*
 * stopped at word; put it into word[]
 * /
 w = word;
 while(isalnum(*p))
   *w++ = *p++;
 *w = '\0';
/*
 * insert it into tree
 */
 if (word[0]){
   if ((n = mknode(word)) == NULL)
     perror(word);
   else
     head = linsert(n);
 }
}
}  
/*
 * function:   mknode
 * arguments:  char *n     word to be entered into new node
 * return:     pointer to new node
 * function:   create a new node containing the word n
 * exceptions: running out of memory; return NULL
 */
 struct word *mknode(char *n)
 {  
 struct word *p;  /* pointer to new space */
    
 /* sanity check */
 if (n == NULL)
   return(NULL);
    
 /* create the element, reporting errors */
 if ((p = malloc(sizeof(struct word))) == NULL)
   return(NULL);
    
 /* create the word, reporting errors */
 if ((p->data = malloc(strlen(n)+1)) == NULL)
   return(NULL);
    
 /* initialize the element */
 (void) strcpy(p->data, n);
 p->next = NULL;
    
 /* return a pointer to the new entity */
 return(p);
 }
}
struct word *linsert(struct word *new)
{
    struct word *prev, *temp; /* previous, current nodes */

    /* empty list: put head at the front */
    if (head == NULL)
        return(new);

    /* it goes before the first element */
    if (strcmp(head->data, new->data) > 0){
        new->next = head;
        return(new);
    }

    /* now walk the list */
    /* from here on in, prev->next == temp */
    /* we'll insert after prev and before temp */
    prev = head;
    temp = head->next;
    while(temp != NULL && strcmp(temp->data, new->data) < 0){
        /* advance prev and temp */
        prev = temp;
        temp = temp->next;
    }

    /* here's the insertion */
    /* make prev->next the new element */
    /* and new->next the one temp points to */
    new->next = temp;
    prev->next = new;

    /* return the pointer to the head of the list */
    return(head);
}

void lprint(struct word *h)
/*
 * base case: empty list
 */
if (h == NULL)
    return;

/*
 * reverse order?
 */
if (reverse)
    /* recurse */
    lprint(h->next);
    /* print current contents */
    printf("%s\n", h->data);
}
else{
    /* print current contents */
    printf("%s\n", h->data);
    /* recurse */
    lprint(h->next);
}
}

Debugging

5. (15 points) The program getbit.c (available on the class website) reads in two numbers, $n$ and $b$. It returns the $b$th bit of integer $n$, where the smallest (rightmost) bit is bit number 0. Rather, it is supposed to. But it doesn’t work. Please debug it.

Answer:
/*
 * getbit -- a program for obtaining a bit from an integer
 *
 * History
 * 1.0   Matt Bishop; original version
 */
#include <stdio.h>

/*
 * the main routine -- does it all
 * Entered by:   no parameters; takes input from stdin
 * Exits with:   nothing
 * Exceptions:  bit position must be between 0 and 32"
 *               -- invalid bit position entered
 */
int main(void)
{
    int n;          /* number to get bits from */
    int bpos;       /* bit position we want */
    register int i; /* number of numbers read */
    */
* loop reading numbers and printing bit positions

```c
while(printf("integer bitnumber> "),
    /*** BUG FIX #1: scanf takes addresses ***/
    (i = scanf("%d %d", &n, &bpos)) != -1){
    /* first check the numbers input for errors */
    if (i == 0){
        /* nothing entered -- reprompt */
        continue;
    } else if (i == 1){
        /* number entered -- go for bit number */
        printf("bit number>> ");
        /* on eof, exit */
        /*** BUG FIX #2: scanf takes addresses (again) ***/
        if ((i = scanf("%d", &bpos)) == -1)
            return(0);
        /* if just a return, ask for number again */
        if (i == 0)
            continue;
    }
    /* is bit position too big? */
    if (bpos > (sizeof(n) * 8 - 1)){
        printf("bit position must be between 0 and %d\n",
                sizeof(n) * 8 - 1);
        continue;
    }
    /* print the result */
    /*** BUG FIX #3: print the last bit ONLY ***/
    printf("%d\n", (n>>bpos)&01);
}
```

/*
 * put out a newline, so shell prompt is on the next line
 */
putchar(\n');

/*
 * fare thee well!
 */
return(0);
}

**Extra Credit**

6. *(10 points)* In your program for problem 3, change the way you handle repeated words as follows. Suppose the word hello occurs 3 times. Instead of printing it 3 times, print it as follows:

```
    hello (3)
```

*Hint:* You will need to change the structure of the node to include a counter.

*Answer:*

```c
/*
 * A Word Listing Program
 */
```
Problem:
Given input, print all the words, one per line, sorted.
A "word" is a maximal sequence of alphanumeric characters.

Version 1.0  Matt Bishop (bishop@cs.ucdavis.edu)
Version 1.1  take command line args and the -r (reverse) option
Version 1.2  count duplicate words rather than re-listing them

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

#define SZBUFFER 1024  /* max length of an input line */

struct word {
    char *data;       /* data field (the word to be sorted) */
    int count;        /* how many times this word appeared */
    struct word *next;/* points to next element in linked list */
};

struct word *head = NULL;

int reverse = 0;

void dofile(FILE *);  /* process a file */
struct word *mknode(char *);  /* create new node in linked list */
struct word *linsert(struct word *);  /* insert new node in linked list */
void lprint(struct word *);  /* print the tree */

/*
 * this is the main routine
 *
 * arguments: option -r to reverse, rest are file names
 * return: exits with 0
 * function: set order of sort, open files and hand them to dofile
 * for processing, print linked list at end
 */
int main(int argc, char *argv[])
{
    int curarg = 1;    /* number of current argument */
    FILE *fp;        /* input file pointer */

    /* check for -r option */
    if (argc > 1 && strcmp(argv[curarg], "-r") == 0){
        /* yep -- reverse list and skip this arg now */
        reverse = 1;
        curarg++;
        argc--;
    }

    /* figure out where to get input:
    * if no files, stdin; else, files */
    if (argc == 1){
        /* from standard input */
        dofile(stdin);
    } else {
        /* from the argument list; walk it */
        for( ; argv[curarg] != NULL; curarg++) {
            /* open file and report error */
            if ((fp = fopen(argv[curarg], "r")) == NULL)
                perror(argv[curarg]);
            else {
                /* worked -- process file */
                dofile(fp);
                (void) fclose(fp);
            }
        }
    }

    /* print the tree */
    lprint(head);
    return(0);
}

/*
 * function:    dofile
 * arguments:   ignored
 * return:      ignored
 * function:    read from file pointer, break each line into words,
 *               add words to linked list
 * exceptions:  none
*/
```c
void dofile(FILE *fp)
{
    char line[SZBUFFER]; /* input line */
    char word[SZBUFFER]; /* word */
    register char *p; /* current line position */
    register char *w; /* used to load line */
    struct word *n; /* points to new node for word */

    /*
    * read the file a line at a time
    *
    * while(fgets(line, SZBUFFER, fp) != NULL){
    */
    for(p = line; *p; ){  
      /*
      * skip leading non-word stuff
      */
        while (*p && !isalnum(*p)) 
            p++;
        /*
        * stopped at word; put it into word[]
        */
        w = word;
        while(isalnum(*p))
            *w++ = *p++;
        *w = '\0';
        /*
        * insert it into tree
        */
        if (word[0]){  
            if ((n = mknode(word)) == NULL)
                perror(word);
            else
                head = linsert(n);
        }
    }
    }

    /*
    * function:    mknode
    * arguments:   char *n    word to be entered into new node
    * return:      pointer to new node
    * function:    create a new node containing the word n
    * exceptions:  running out of memory; return NULL
    */
    struct word *mknode(char *n)
    {
        struct word *p; /* pointer to new space */
```
/* sanity check */
if (n == NULL)
    return(NULL);

/* create the element, reporting errors */
if ((p = malloc(sizeof(struct word))) == NULL)
    return(NULL);

/* create the word, reporting errors */
if ((p->data = malloc(strlen(n)+1)) == NULL)
    return(NULL);

/* initialize the element */
(void) strcpy(p->data, n);
p->count = 1;
p->next = NULL;

/* return a pointer to the new entity */
return(p);
}

/*
 * function: frnode
 * arguments: struct word *nnode to be released
 * return: nothing
 * function: release storage for node
 * exceptions: none
 */
void frnode(struct word *n)
{
    /* sanity check */
    if (n == NULL)
        return;

    /* another sanity check; release the string */
    if (n->data != NULL)
        (void) free(n->data);

    /* release the node */
    (void) free(n);
}

/*
 * function: linsert
 * arguments: struct word *new pointer to node to be inserted
 * return: pointer to head of list
 * function: insert node into linked list in order
 * exceptions: none
 * external: head; points to head of linked list
 */
struct word *linsert(struct word *new)
{
    struct word *prev, *temp; /* previous, current nodes */
/* empty list: put head at the front */
if (head == NULL)
    return(new);

/* it goes before the first element */
if (strcmp(head->data, new->data) > 0){
    new->next = head;
    return(new);
}

/* now walk the list
   * from here on in, prev->next == temp
   * we'll insert after prev and before temp
   */
prev = head;
temp = head->next;
while(temp != NULL && strcmp(temp->data, new->data) < 0){
    /* advance prev and temp */
    prev = temp;
    temp = temp->next;
}

/* here's the insertion
   * if the current node:
   *   increment counter
   * if it is not the current node:
   *   make prev->next the new element
   *   and new->next the one temp points to
   */
if (temp != NULL && strcmp(new->data, temp->data) == 0){
    /* bump count */
    temp->count++;
    /* release new node */
    frnode(new);
}
else{
    new->next = temp;
    prev->next = new;
}

/* return the pointer to the head of the list */
return(head);

/* function: lprint
 * arguments: struct word *h     points to head of list to be printed
 * return: nothing
 * function: print linked list
 * exceptions: none
 * external: reverse; indicates order of printing
`void lprint(struct word *h) {
    /*
     * base case: empty list
     */
    if (h == NULL)
        return;

    /*
     * reverse order?
     */
    if (reverse) {
        /* recurse */
        lprint(h->next);
        /* print current contents */
        printf("%s (%d)\n", h->data, h->count);
    }
    else {
        /* print current contents */
        printf("%s (%d)\n", h->data, h->count);
        /* recurse */
        lprint(h->next);
    }
}