Problem Set 6

Part 1: Pointers to pointers. Multidimensional arrays. Stacks and queues.
Part 2: Function pointers, hash table

Problem 6.1

In this problem, we will implement a simple "four-function" calculator using stacks and queues. This calculator takes as input a space-delimited infix expression (e.g. 3 + 4 * 7), which you will convert to postfix notation and evaluate. There are four (binary) operators your calculator must handle: addition (+), subtraction (-), multiplication (*), and division (/). In addition, your calculator must handle the unary negation operator (also -). The usual order of operations is in effect:

- **the unary negation operator - has higher precedence than the binary operators, and is evaluated right-to-left (right-associative)**
- *** and / have higher precedence than + and -**
- **all binary operators are evaluated left-to-right (left-associative)**

To start, we will not consider parentheses in our expressions. The code is started for you in prob1.c, which is available for download from Stellar. Read over the file, paying special attention to the data structures used for tokens, the stack, and queue, as well as the functions you will complete.

(a) We have provided code to translate the string to a queue of tokens, arranged in infix (natural) order. You must: – fill in the infix to postfix() function to construct a queue of tokens arranged in postfix order (the infix queue should be empty when you're done) – complete the evaluate postfix() function to evaluate the expression stored in the postfix queue and return the answer You may assume the input is a valid infix expression, but it is good practice to make your
code robust by handling possible errors (e.g. not enough tokens) . Turn in a printout of your code, along with a printout showing the output from your program for a few test cases (infix expressions of your choosing) to demonstrate it works properly.

OUTPUT:

```
 struct token_queue infix_to_postfix(struct token_queue *pqueue_infix)
{
  struct token_queue queue_postfix;
  queue_postfix.front = queue_postfix.back = NULL;
  struct token *ptoken;
  struct expr_token_stack top = NULL;

  for (ptoken = dequeue(pqueue_infix); ptoken; ptoken = dequeue(pqueue_infix))
    {
```

```
            switch (ptoken->type) {
                case OPERAND: {
                    enqueue(&queue_postfix, ptoken);
                     break;  }
                case OPERATOR: {
                     while (top && (operator_precedences[top->value.opcode]>
operator_precedences[ptoken->value.opcode] ||
                         (operator_precedences[top->value.opcode] ==
operator_precedences[ptoken->value.opcode] &&
                          operator_associativity[operator_precedences[ptoken->value.opcode]] ==
LEFT))) {
                         enqueue(&queue_postfix, pop(&top));  }
                         push(&top, ptoken);
                         break;  }
                   default: {
                        free(ptoken);
                         break;  } } }
       while (top) {
       enqueue(&queue_postfix, pop(&top));
   }
   return queue_postfix;
}
double evaluate_postfix(struct token_queue *pqueue_postfix) {
   double ans = 0.0;
   struct expr_token_stack values = NULL;
   struct token *ptoken, *pvalue;
   double operands[2];
   union token_value value;
   int i;

   while ((ptoken = dequeue(pqueue_postfix))) {
      switch (ptoken->type) {
         case OPERAND: {
            push(&values, ptoken);
            break;
         }
         case OPERATOR: {
            for (i = 0; i < operator_operands[ptoken->value.opcode]; i++) {
               if ((pvalue = pop(&values))) {
                  operands[i] = pvalue->value.operand;
                  free(pvalue);
               } else {
                  goto error;
               }
```

```
            }
            switch (ptoken->value.opcode) {
                case ADD: {
                    value.operand = operands[1] + operands[0];
                    break;
                }
                case SUBTRACT: {
                    value.operand = operands[1] - operands[0];
                    break;
                }
                case MULTIPLY: {
                    value.operand = operands[1] * operands[0];
                    break;
                }
                case DIVIDE: {
                    value.operand = operands[1] / operands[0];
                    break;
                }
                case NEGATE: {
                    value.operand = -operands[0];
                }
            }
            push(&values, new_token(OPERAND, value));
        }
        default: {
            free(ptoken);
            break;
        }
    }
}
if (values) {
    ans = values->value.operand;
}
cleanup: {
    while ((ptoken = dequeue(pqueue_postfix))) {
        free(ptoken);
    }
    while ((pvalue = pop(&values))) {
        free(pvalue);
    }
    return ans;
}
error: {
    fputs("Error evaluating the expression.\n", stderr);
```

```
        goto cleanup;
    }
}
```

(b) Now, an infix calculator really is not complete if parentheses are not allowed. So, in this part, update the function infix to postfix() to handle parentheses as we discussed in class. Note: your postfix queue should contain no parentheses tokens. Turn in a printout of your code, along with a printout showing the output from your program for a few test cases
utilizing parentheses

```
struct token_queue infix_to_postfix(struct token queue *p_queue_infix) {
    struct token stack_top = NULL, p_token;
    struct token queue_postfix;
    queue_postfix.front = queue_postfix.back = NULL;

    for (p_token = dequeue(p_queue_infix); p_token; p_token = dequeue(p_queue_infix)) {
        switch (p_token->type) {
            case OPERAND:
                enqueue(&queue_postfix, p_token);
                break;
            case OPERATOR:
                while (stack_top && stack_top->type == OPERATOR &&
                      (op_precedences[stack_top->value.op_code] >
                       op_precedences[p_token->value.op_code] ||
                       (op_precedences[stack_top->value.op_code] ==
                        op_precedences[p_token->value.op_code] &&
                        op_associativity[op_precedences[p_token->value.op_code]] == LEFT)))
                    enqueue(&queue_postfix, pop(&stack_top));
                push(&stack_top, p_token);
                break;
            case LPARENS:
                push(&stack_top, p_token);
                break;
            case RPARENS:
                free(p_token);
                while ((p_token = pop(&stack_top))) {
                    if (p_token->type == LPARENS) {
                        free(p_token);
                        break;
                    }
                    enqueue(&queue_postfix, p_token);
                }
        }
```

```
    }

    while (stack_top)
        enqueue(&queue_postfix, pop(&stack_top));

    return queue_postfix;
}
```

OUTPUT:

Problem 6.2

A useful data structure for storing lots of strings is the "trie." This tree structure has the special property that a node's key is a prefix of the keys of its children. For instance, if we associate a node with the string "a," that node may have a child node with the key "an," which in turn may have a child node "any." When many strings share a common prefix, this structure is a very inexpensive way to store them. Another consequence of this storage method is that the trie supports very fast searching – the complexity of finding a string with m characters is O(m).

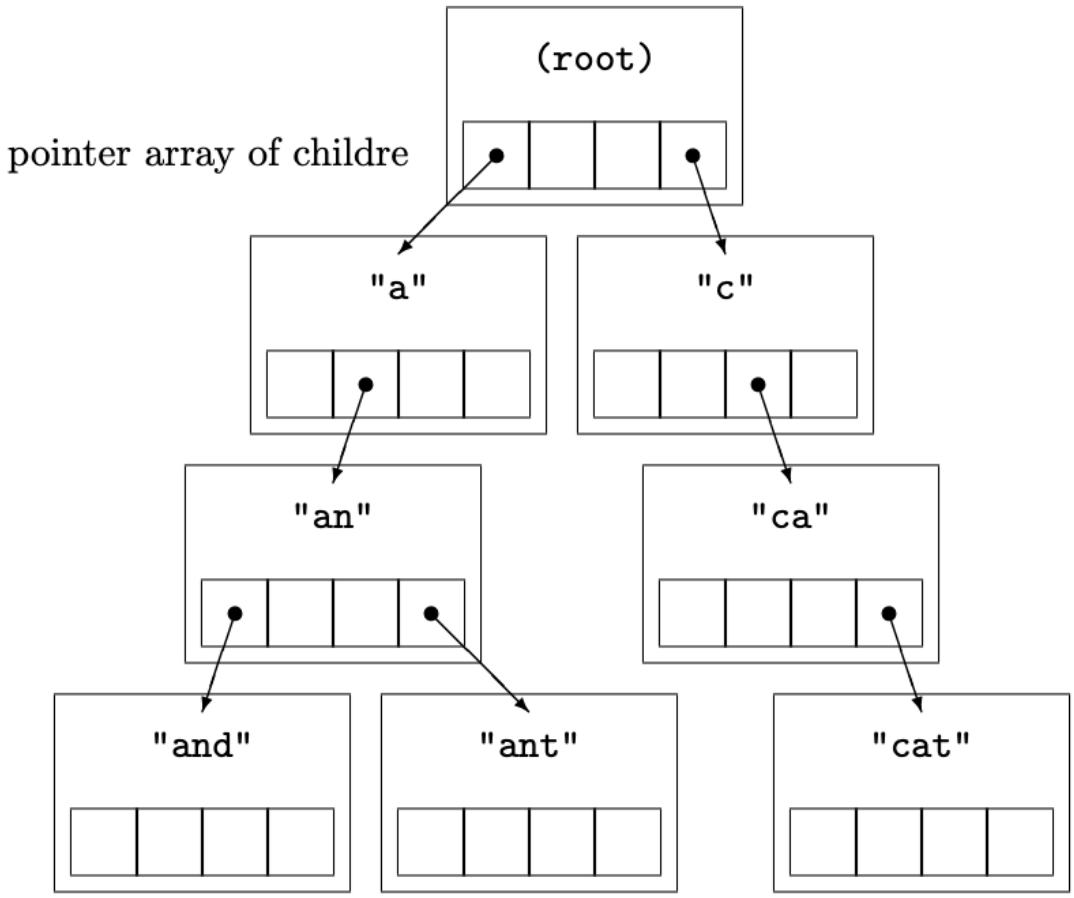

Figure 6.2-1: Trie structure (translations not shown). For each node, its key (e.g. "and") is not explicitly stored in the node; instead, the key is defined by the node's position in the tree: the key of its parent node + its index in that parent's pointer array of children.

In this problem, you will utilize a trie structure and to implement a simple one-way Englishto-French dictionary. The trie structure, shown in Figure 6.2-1, consists of nodes, each of which contains a string for storing translations for the word specified at that node and an array of pointers to child nodes. Each node, by virtue of its position in the trie, is associated with a string; in the dictionary context, this string is the word (or part of a word) in the dictionary. The dictionary may contain multiple translations for the same word; in this case, words should be separated by commas. For example: the word like, which has two meanings, could translate as comme, a preposition, or as aimer, a verb. Thus, the translation string should be "comme,aimer." To get you started, we've provided code in prob2.c, which can be downloaded from Stellar. You will need to:

• fill in the helper functions new node(), delete node() • complete the function add word(), which adds a word to the trie • complete the function lookup word(), which searches the trie for a word and returns its translation(s) Once your code is working, run a few test cases.

ANSWER

```
struct strienode *new_node(void)
{
      struct strienode *pnode = (struct strienode *)malloc(sizeof(struct strienode));
       int i;
      pnode->translation = NULL;
      for (i = 0; i < UCHARMAX + 1; i++)
         pnode->children[i] = NULL;
      return pnode;
 }

 void delete_node(struct strienode *pnode)
 {
 int i;
 if (pnode->translation)
 free(pnode->translation);
 for (i = 0; i < UCHARMAX + 1; i++)
   if (pnode->children[i])
       delete_node(pnode->children[i]);
 free(pnode);
}

int add_word(const char *word, char *translation)
  {
struct strienode *pnode = proot;
int i, len = strlen(word), inew = 0;
unsigned char j;
for (i = 0; i < len; i++)
     {
          j = word[i];
          if ((inew = !pnode->children[j]))
              pnode->children[j] = new_node();
          pnode = pnode->children[j];
 }

if (pnode->translation) {
    char *old_translation = pnode->translation;
    int oldlen = strlen(old_translation),
     newlen = strlen(translation);
    pnode->translation = malloc(oldlen + newlen + 2);
    strcpy(pnode->translation, old_translation);
    strcpy(pnode->translation + oldlen, ",");
```

```
        strcpy(pnode−>translation + oldlen + 1, translation);
        free(old_translation);
    }
 Else
        pnode−>translation = strcpy(malloc(strlen(translation) + 1), translation);
      return inew;
}

char ∗lookup_word(const char ∗word)
{
    struct strienode ∗pnode = proot;
    int i, len = strlen(word);
    unsigned char j;
    for (i = 0; i < len; i++)
      {
        j = word[i];
        if (!pnode−>children[j])
            return NULL;
        pnode = pnode−>children[j];
    }
    return pnode−>translation;
}
```

Problem 6.3

In this problem, we will use and create function that utilize function pointers. The file 'callback.c' contains an array of records consisting of a fictitious class of celebrities. Each record consists of the firstname, lastname and age of the student. Write code to do the following: • Sort the records based on first name. To achieve this, you will be using the qsort() function provided by the standard library: void qsort(void∗ arr,int num,int size,int (∗fp)(void∗ pa,void∗pb)). The function takes a pointer to the start of the array 'arr', the number of elements 'num' and size of each element. In addition it takes a function pointer 'fp' that takes two arguments. The function fp is used to compare two elements within the array. Similar to strcmp(), it is required to return a negative quantity,zero or a positive quantity depending on whether element pointed to by 'pa' is "less" than, equal to or "greater" the element pointed to by 'pb'. You are required to write the appropriate callback function.

• Now sort the records based on last name. Write the appropriate callback function.

• The function void apply (...) iterates through the elements of the array calling a function for each element of the array. Write a function isolder() that prints the record if the age of the student is greater 20 and does nothing otherwise.

Problem 6.4

A useful data structure for doing lookups is a hash table. In this problem, you will be implementing a hash table with chaining to store the frequency of words in a file. The hash table is implemented as an array of linked lists. The hash function specifies the index of the linked list to follow for a given word. The word can be found by following this linked list. Optionally, the word can be appended to this linked list. You will require the code file 'hash.c' and data file 'book.txt' for this problem. You are required to do the following

• The function lookup() returns a pointer to the record having the required string. If not found it returns NULL or optionally creates a new record at the correct location. Please complete the rest of the code.

• Complete the function cleartable() to reclaim memory. Make sure each call to malloc() is matched with a free()

SOLUTION:

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

#define MAX_BUCKETS 1000
#define MULTIPLIER 31
#define MAX_LEN 100

struct wordrec {
    char *word;
    unsigned long count;
    struct wordrec *next;
};

struct wordrec *allocate(const char *str) {
    struct wordrec *p = (struct wordrec *)malloc(sizeof(struct wordrec));
    if (p != NULL) {
        p->count = 0;
        p->word = strdup(str);
        p->next = NULL;
    }
    return p;
```

```
}

struct wordrec *table[MAX_LEN];
unsigned long hash_string(const char *str) {
    unsigned long hash = 0;
    while (*str) {
        hash = hash * MULTIPLIER + *str;
        str++;
    }
    return hash % MAX_BUCKETS;
}

struct wordrec *lookup(const char *str, int create) {
    unsigned long hash = hash_string(str);
    struct wordrec *wp = table[hash];
    struct wordrec *curr = NULL;
    for (curr = wp; curr != NULL; curr = curr->next) {
        if (strcmp(curr->word, str) == 0) {
            return curr;
        }
    }
    if (create) {
        curr = (struct wordrec *)malloc(sizeof(struct wordrec));
        curr->word = strdup(str);
        curr->count = 0;
        curr->next = table[hash];
        table[hash] = curr;
    }
    return curr;
}

void clear_table() {
    struct wordrec *wp = NULL, *p = NULL;
    int i = 0;
    for (i = 0; i < MAX_BUCKETS; i++) {
        wp = table[i];
        while (wp) {
            p = wp;
            wp = wp->next;
            free(p->word);
            free(p);
        }
    }
}
```

```
int main(int argc, char *argv[]) {
    FILE *fp = fopen("book.txt", "r");
    char word[1024];
    struct wordrec *wp = NULL;
    int i = 0;

    memset(table, 0, sizeof(table));
    while (1) {
        if (fscanf(fp, "%s", word) != 1)
            break;
        wp = lookup(word, 1);
        wp->count++;
    }
    fclose(fp);

    for (i = 0; i < MAX_BUCKETS; i++) {
        for (wp = table[i]; wp != NULL; wp = wp->next) {
            if (wp->count > 1000) {
                printf("%s-->%ld\n", wp->word, wp->count)
}   }   }
    clear_table();
    return 0;
}
```