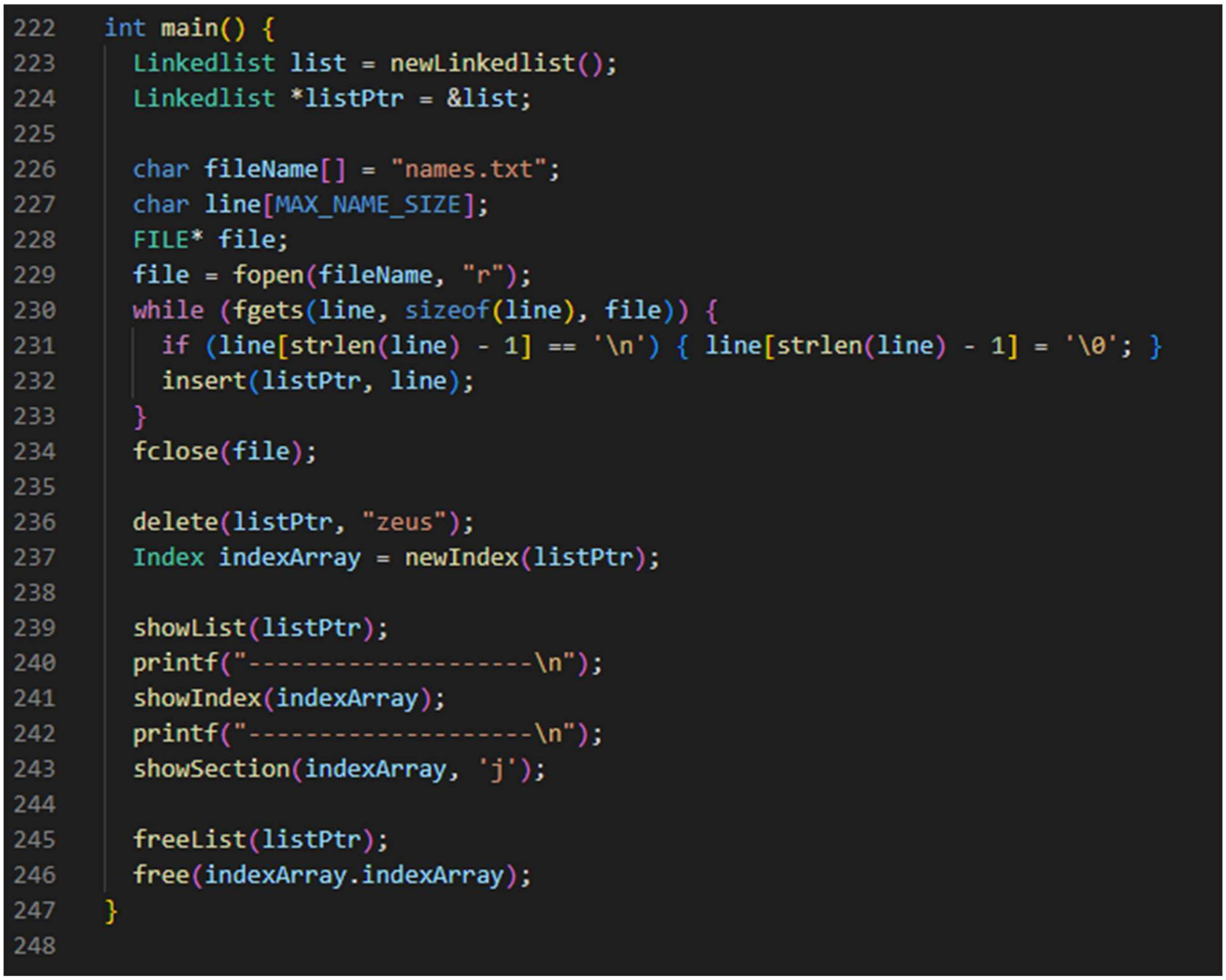
Jacob Tucker

CMPS 390

3/19/23

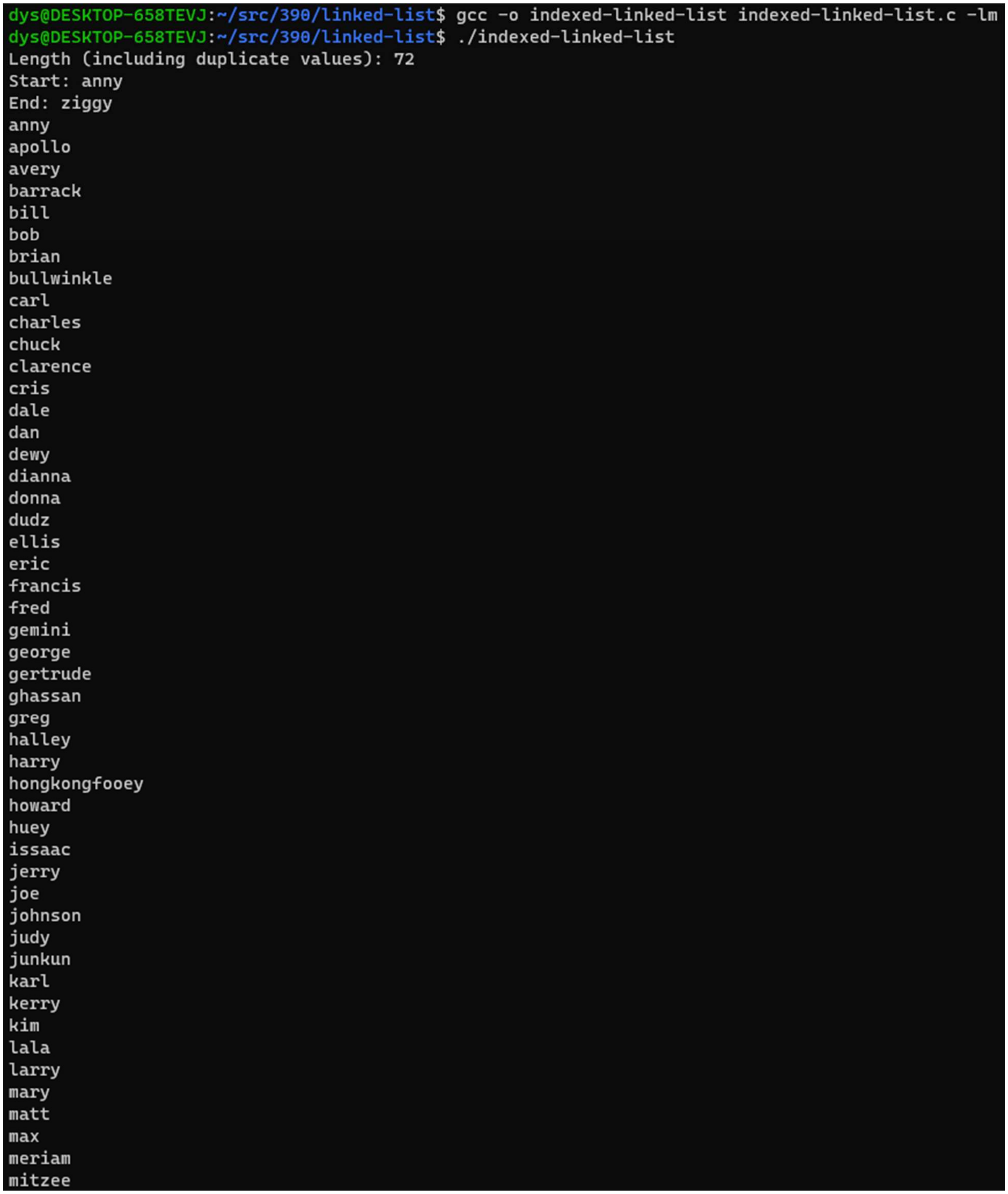
# Indexed Linked List

This program is similar to the Simple Linked List program except that indexing was added. The extra features include reading input from a file, working with strings and storing them in alphabetical order, showing the count for all strings beginning with a specified character, and showing the section of strings that begin with a specified letter. These features are demonstrated by calling the relevant functions in *main()* as shown in these screenshots:

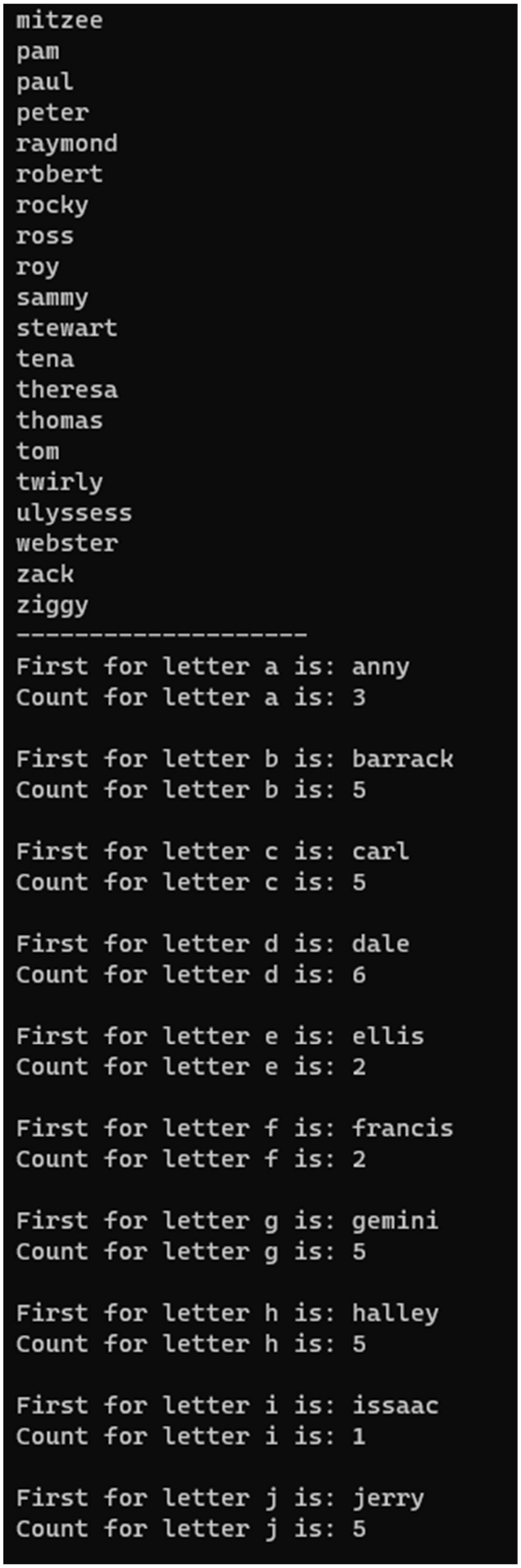


```c
int main() {
  Linkedlist list = newLinkedlist();
  Linkedlist *listPtr = &list;

  char fileName[] = "names.txt";
  char line[MAX_NAME_SIZE];
  FILE* file;
  file = fopen(fileName, "r");
  while (fgets(line, sizeof(line), file)) {
    if (line[strlen(line) - 1] == '\n') { line[strlen(line) - 1] = '\0'; }
    insert(listPtr, line);
  }
  fclose(file);

  delete(listPtr, "zeus");
  Index indexArray = newIndex(listPtr);

  showList(listPtr);
  printf("--------------------\n");
  showIndex(indexArray);
  printf("--------------------\n");
  showSection(indexArray, 'j');

  freeList(listPtr);
  free(indexArray.indexArray);
}

```

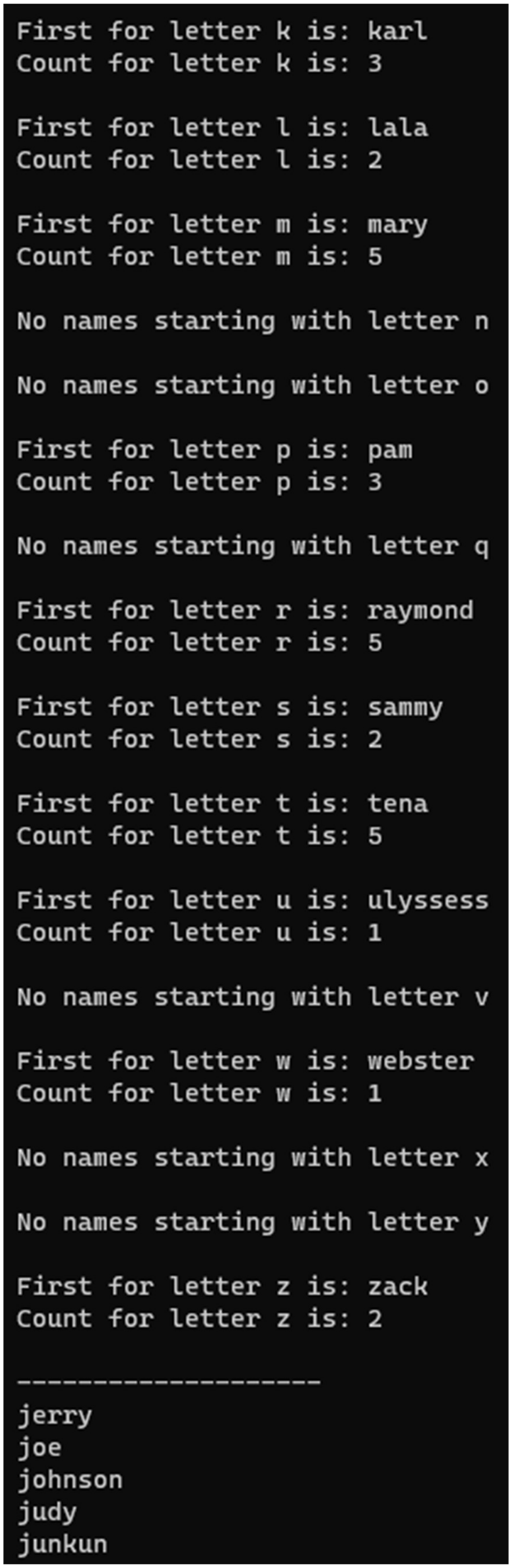
We create a linked list, fill it with strings from the “names.txt” file, delete one of the names, create the index for the array, print the list, print the index, print the section for names starting with the letter “j”, and finally free all manually allocated memory. The output is as follows:



```
dys@DESKTOP-658TEVJ:~/src/390/linked-list$ gcc -o indexed-linked-list indexed-linked-list.c -lm
dys@DESKTOP-658TEVJ:~/src/390/linked-list$ ./indexed-linked-list
Length (including duplicate values): 72
Start: anny
End: ziggy
anny
apollo
avery
barrack
bill
bob
brian
bullwinkle
carl
charles
chuck
clarence
cris
dale
dan
dewy
dianna
donna
dudz
ellis
eric
francis
fred
gemini
george
gertrude
ghassan
greg
halley
harry
hongkongfooey
howard
huey
issaac
jerry
joe
johnson
judy
junkun
karl
kerry
kim
lala
larry
mary
matt
max
meriam
mitzee
```

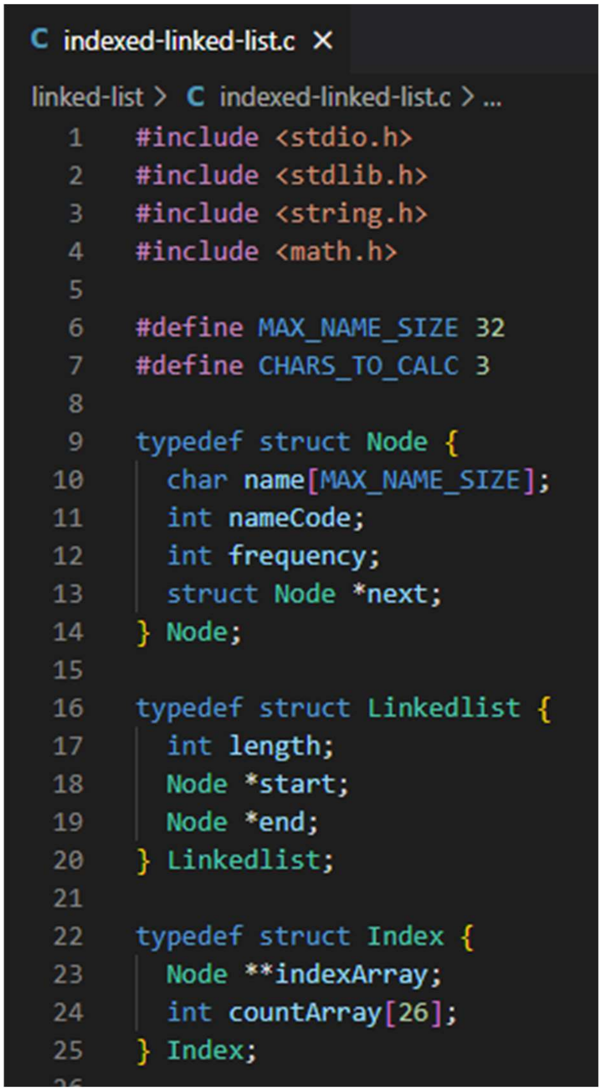


```
mitzee
pam
paul
peter
raymond
robert
rocky
ross
roy
sammy
stewart
tena
theresa
thomas
tom
twirly
ulyssess
webster
zack
ziggy
--------------------
First for letter a is: anny
Count for letter a is: 3

First for letter b is: barrack
Count for letter b is: 5

First for letter c is: carl
Count for letter c is: 5

First for letter d is: dale
Count for letter d is: 6

First for letter e is: ellis
Count for letter e is: 2

First for letter f is: francis
Count for letter f is: 2

First for letter g is: gemini
Count for letter g is: 5

First for letter h is: halley
Count for letter h is: 5

First for letter i is: issaac
Count for letter i is: 1

First for letter j is: jerry
Count for letter j is: 5
```



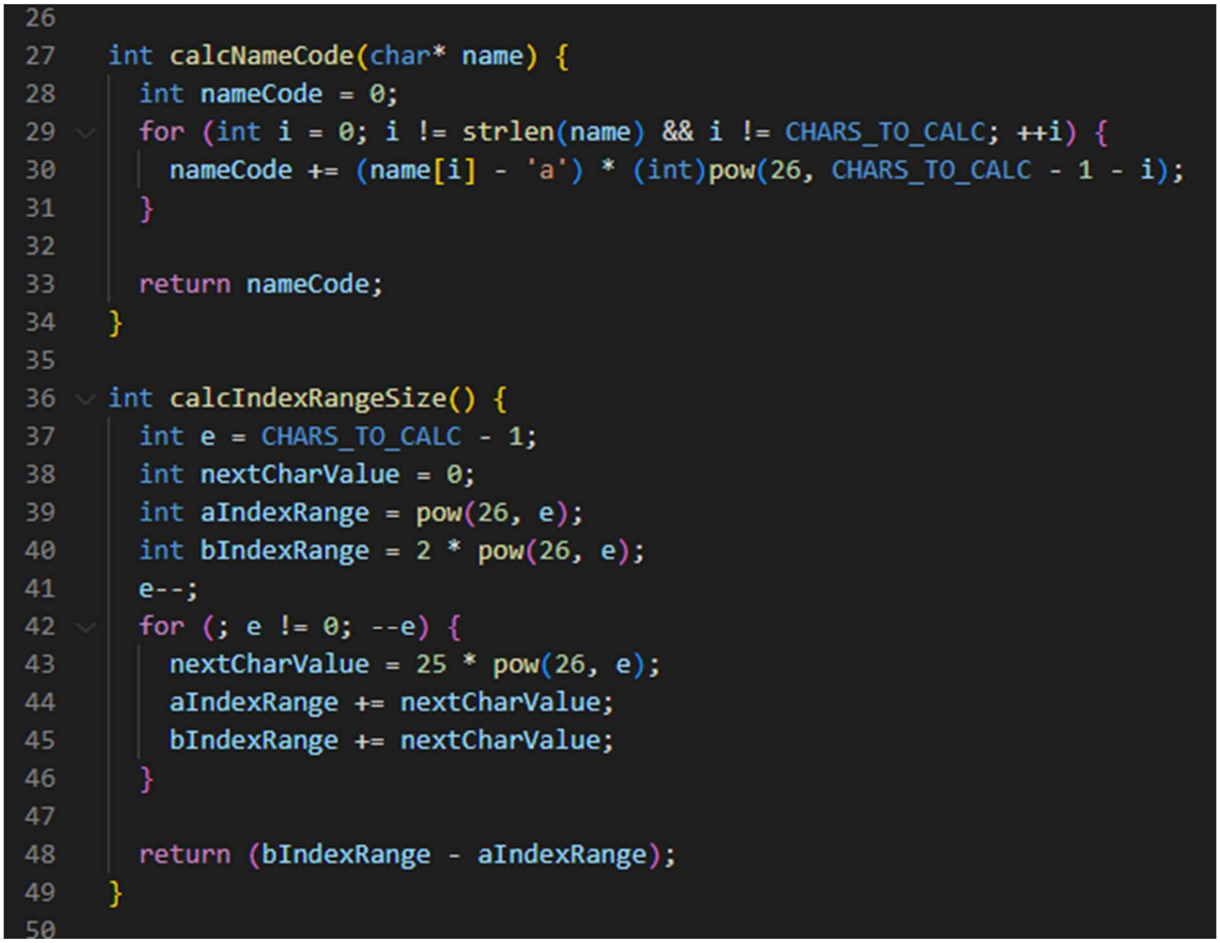
```
First for letter k is: karl
Count for letter k is: 3

First for letter l is: lala
Count for letter l is: 2

First for letter m is: mary
Count for letter m is: 5

No names starting with letter n

No names starting with letter o

First for letter p is: pam
Count for letter p is: 3

No names starting with letter q

First for letter r is: raymond
Count for letter r is: 5

First for letter s is: sammy
Count for letter s is: 2

First for letter t is: tena
Count for letter t is: 5

First for letter u is: ulyssess
Count for letter u is: 1

No names starting with letter v

First for letter w is: webster
Count for letter w is: 1

No names starting with letter x

No names starting with letter y

First for letter z is: zack
Count for letter z is: 2

--------------------
jerry
joe
johnson
judy
junkun
```

Since the Simple Linked List code was used as a template for this program, for the sake of brevity I will just highlight the parts that are different.



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

#define MAX_NAME_SIZE 32
#define CHARS_TO_CALC 3

typedef struct Node {
  char name[MAX_NAME_SIZE];
  int nameCode;
  int frequency;
  struct Node *next;
} Node;

typedef struct Linkedlist {
  int length;
  Node *start;
  Node *end;
} Linkedlist;

typedef struct Index {
  Node **indexArray;
  int countArray[26];
} Index;
```

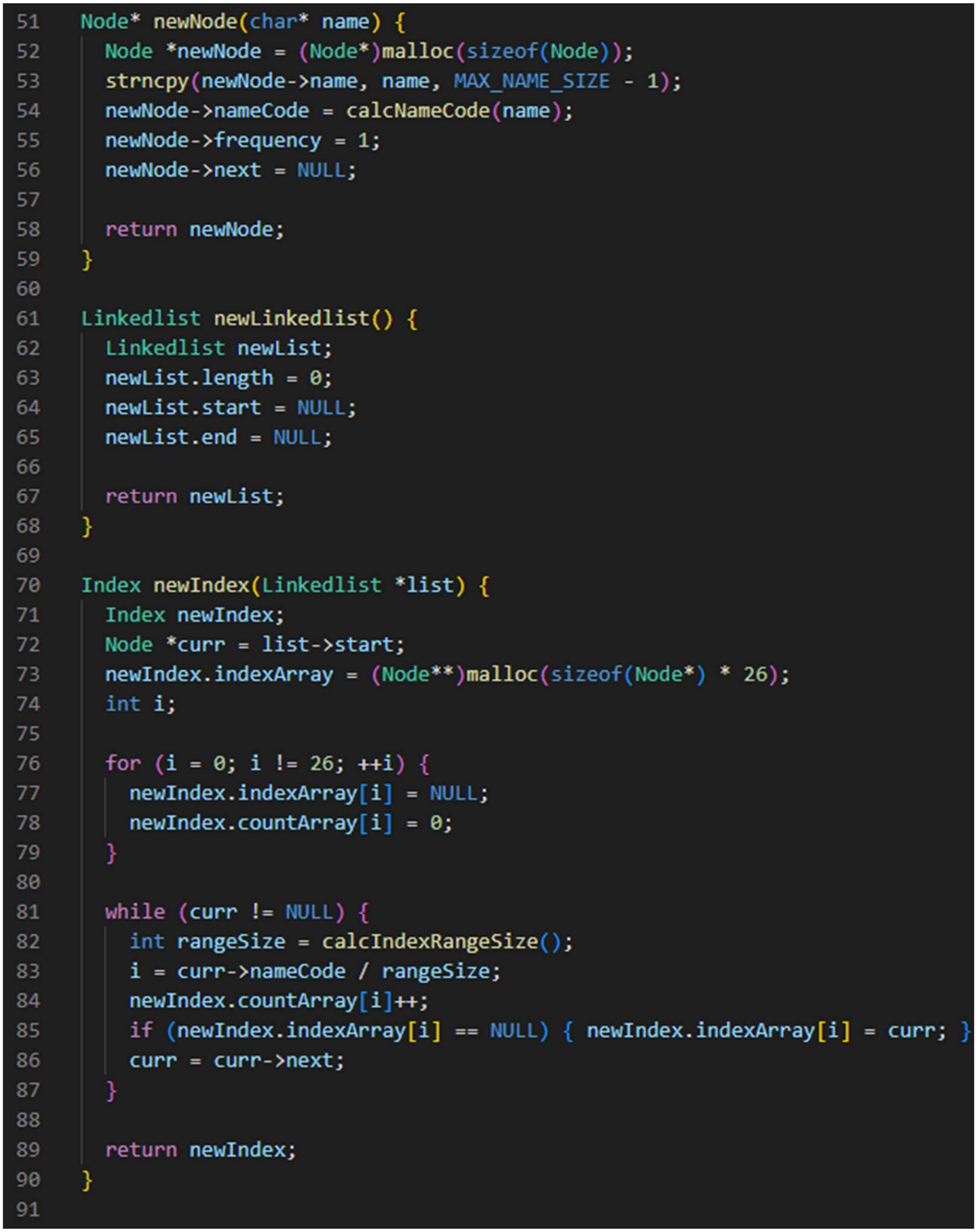
The *Node* struct was given an additional field for storing *name* string. An additional structure *Index* was also added, which has two fields. They are both arrays, with one being an array of integers to keep track of the count of names for each letter and the other being an array of pointers to the first *Node* in a *Linkedlist* for each letter. These will be used as parallel arrays for convenient data storage and retrieval.



```
int calcNameCode(char* name) {
  int nameCode = 0;
  for (int i = 0; i != strlen(name) && i != CHARS_TO_CALC; ++i) {
    nameCode += (name[i] - 'a') * (int)pow(26, CHARS_TO_CALC - 1 - i);
  }

  return nameCode;
}

int calcIndexRangeSize() {
  int e = CHARS_TO_CALC - 1;
  int nextCharValue = 0;
  int aIndexRange = pow(26, e);
  int bIndexRange = 2 * pow(26, e);
  e--;
  for (; e != 0; --e) {
    nextCharValue = 25 * pow(26, e);
    aIndexRange += nextCharValue;
    bIndexRange += nextCharValue;
  }

  return (bIndexRange - aIndexRange);
}
```

The *calcNameCode()* function takes a name as an argument and calculates its name code, using a constant *CHARS_TO_CALC* in the formula so that it will alphabetize the names based on more than just the first 3 letters if desired.

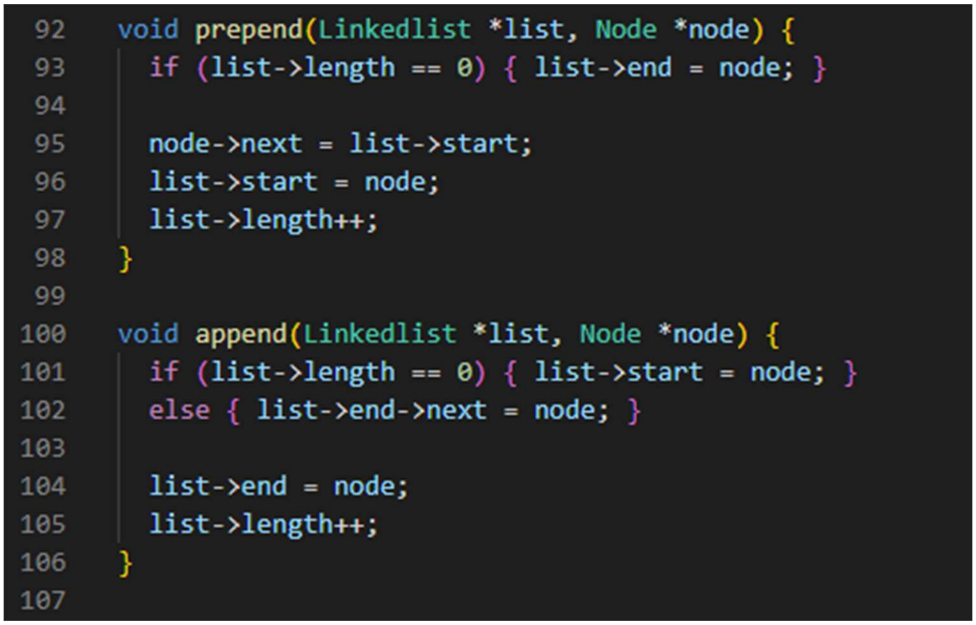
The *calcIndexRangeSize()* function uses the *CHARS_TO_CALC* constant to determine the highest *nameCode* value a word beginning with letter “a” will have and subtracts it from the highest value a word beginning with letter “b” will have. This gives us the size of the range each index will have based on the number of characters we want to calculate. E.g., for 3 characters it will return 676. This is used later to easily assign words to the appropriate indices.



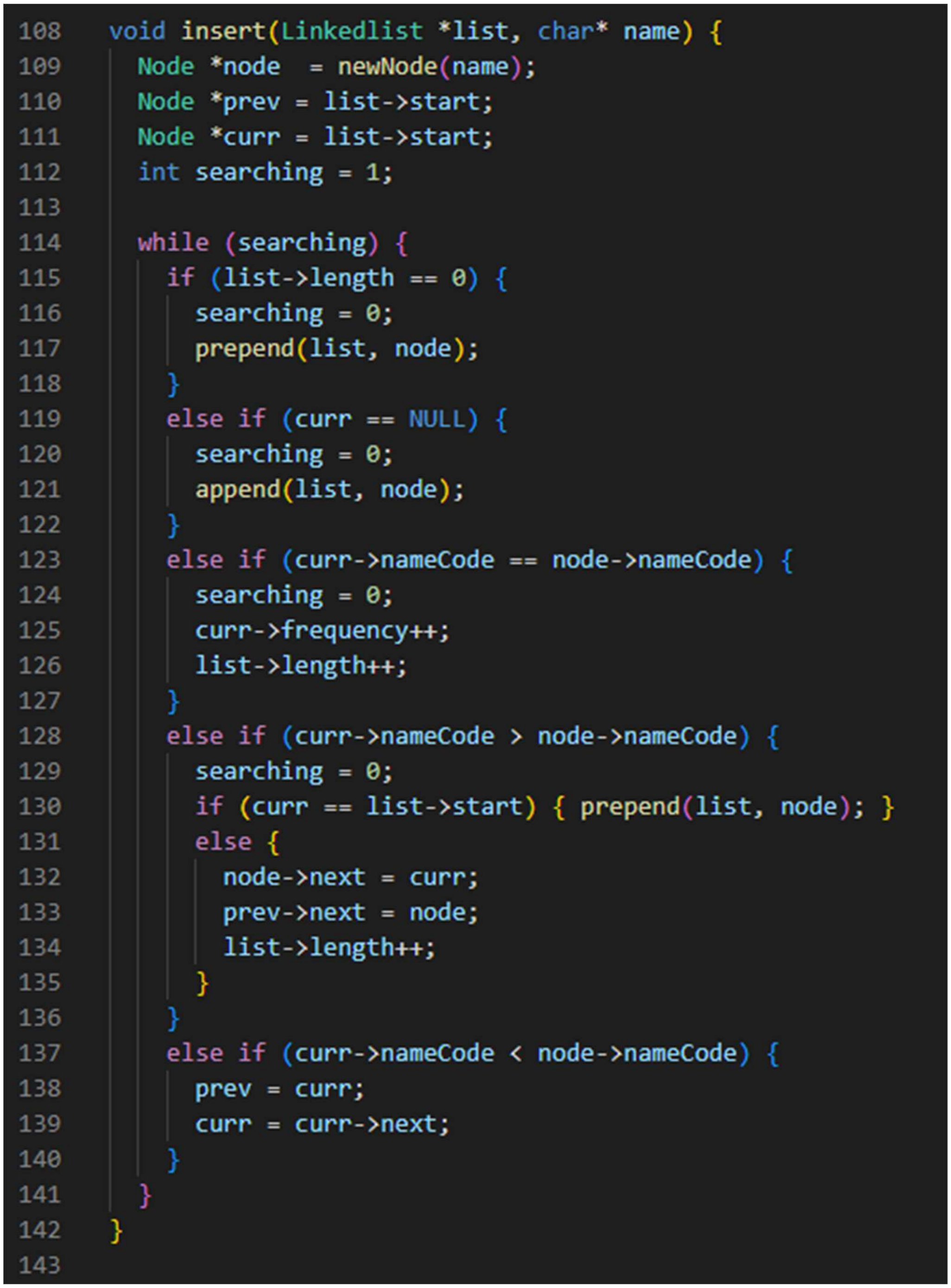
```c
Node* newNode(char* name) {
  Node *newNode = (Node*)malloc(sizeof(Node));
  strncpy(newNode->name, name, MAX_NAME_SIZE - 1);
  newNode->nameCode = calcNameCode(name);
  newNode->frequency = 1;
  newNode->next = NULL;

  return newNode;
}

Linkedlist newLinkedlist() {
  Linkedlist newList;
  newList.length = 0;
  newList.start = NULL;
  newList.end = NULL;

  return newList;
}

Index newIndex(Linkedlist *list) {
  Index newIndex;
  Node *curr = list->start;
  newIndex.indexArray = (Node**)malloc(sizeof(Node*) * 26);
  int i;

  for (i = 0; i != 26; ++i) {
    newIndex.indexArray[i] = NULL;
    newIndex.countArray[i] = 0;
  }

  while (curr != NULL) {
    int rangeSize = calcIndexRangeSize();
    i = curr->nameCode / rangeSize;
    newIndex.countArray[i]++;
    if (newIndex.indexArray[i] == NULL) { newIndex.indexArray[i] = curr; }
    curr = curr->next;
  }

  return newIndex;
}
```

The *newNode()* function is mostly the same, but now also stores the *name* it is given and gets its *nameCode* to store as well. The *newIndex()* function allocates memory for its array of pointers to *Node*s and then initializes all of the elements to NULL. It also initializes all the elements of its count tracking array to 0. It then loops through the *Linkedlist* it is passed, calculating the index each string should belong to with integer division. It divides the current *nameCode* by the size of each index range and stores the integer portion of this quotient to be used as the index. This mathematically maps each *nameCode* to an number from 0 – 25 no matter how many letters we wish to use in the calculation, with every word beginning with the letter “a” being mapped to 0, “b” to 1, etc. The parallel array tracking the count for each index is incremented, and if there is currently no name being pointed to by the array of indices then the pointer at that index is changed to point to the current *Node*. This ensures only the first *Node* with that letter is pointed to, but all *Node*s with that letter will increment the count.

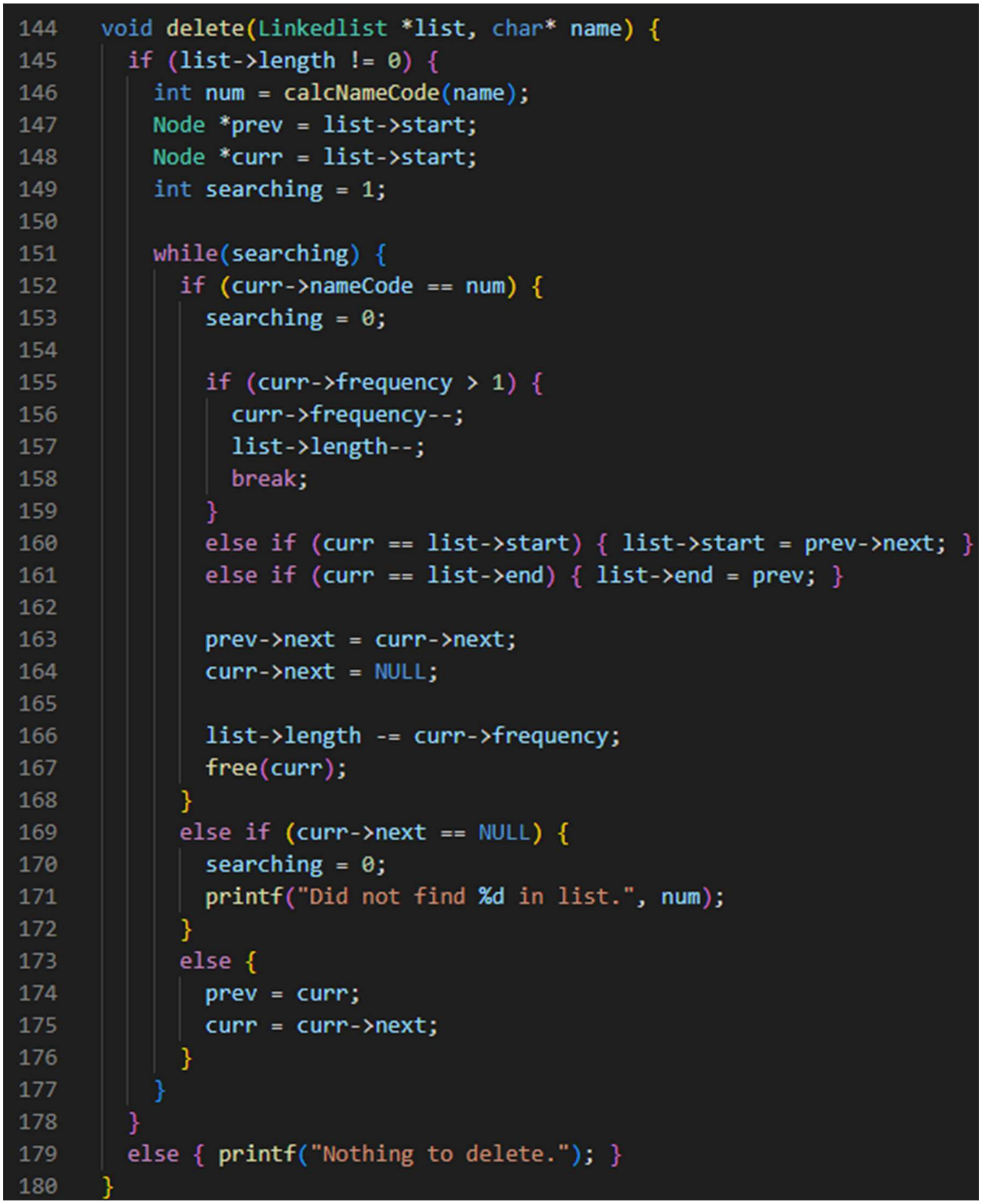
The following code has no changes form the simple linked list program:



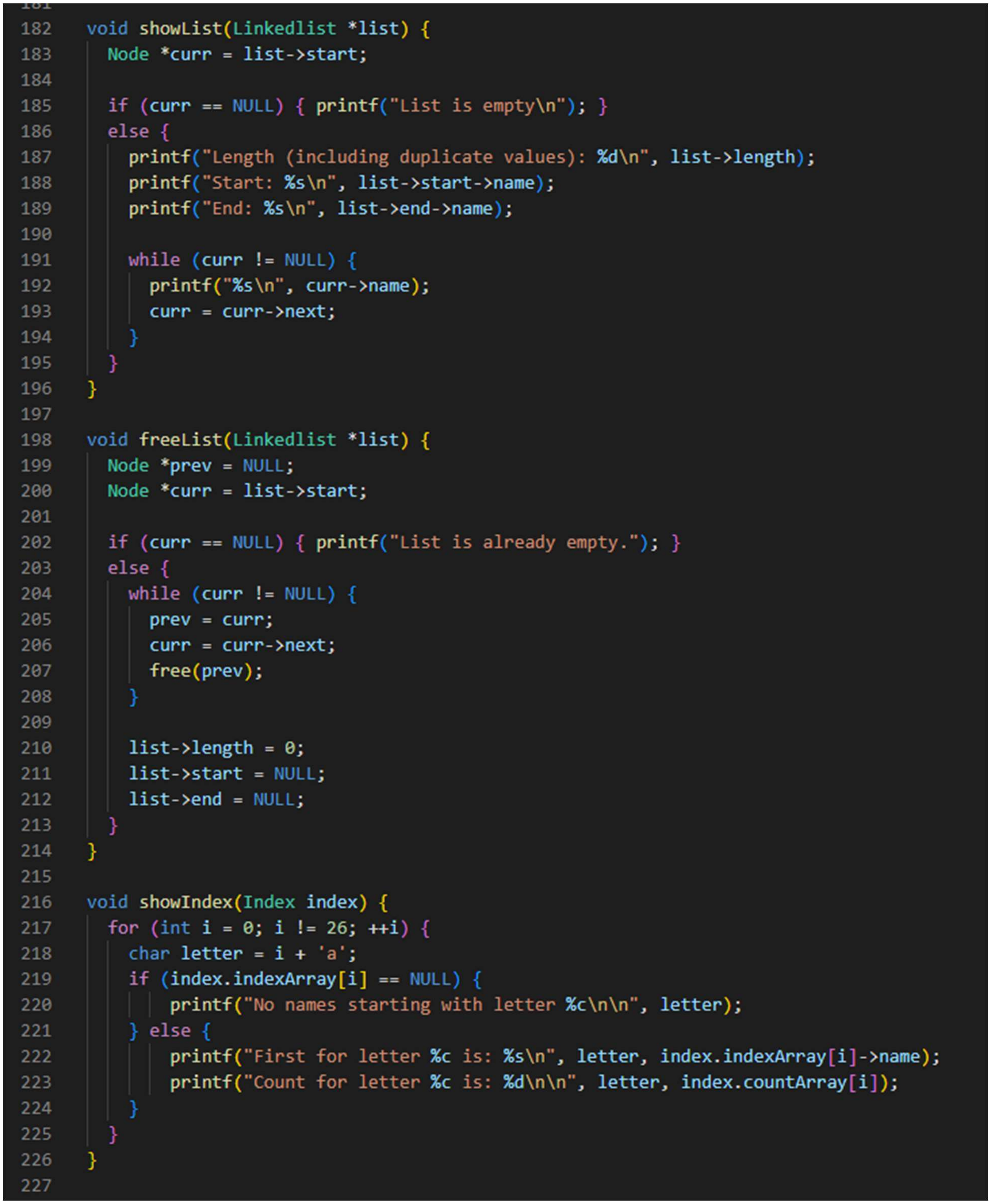
```c
void prepend(Linkedlist *list, Node *node) {
  if (list->length == 0) { list->end = node; }

  node->next = list->start;
  list->start = node;
  list->length++;
}

void append(Linkedlist *list, Node *node) {
  if (list->length == 0) { list->start = node; }
  else { list->end->next = node; }

  list->end = node;
  list->length++;
}
```



```c
void insert(Linkedlist *list, char* name) {
  Node *node  = newNode(name);
  Node *prev = list->start;
  Node *curr = list->start;
  int searching = 1;

  while (searching) {
    if (list->length == 0) {
      searching = 0;
      prepend(list, node);
    }
    else if (curr == NULL) {
      searching = 0;
      append(list, node);
    }
    else if (curr->nameCode == node->nameCode) {
      searching = 0;
      curr->frequency++;
      list->length++;
    }
    else if (curr->nameCode > node->nameCode) {
      searching = 0;
      if (curr == list->start) { prepend(list, node); }
      else {
        node->next = curr;
        prev->next = node;
        list->length++;
      }
    }
    else if (curr->nameCode < node->nameCode) {
      prev = curr;
      curr = curr->next;
    }
  }
}

```

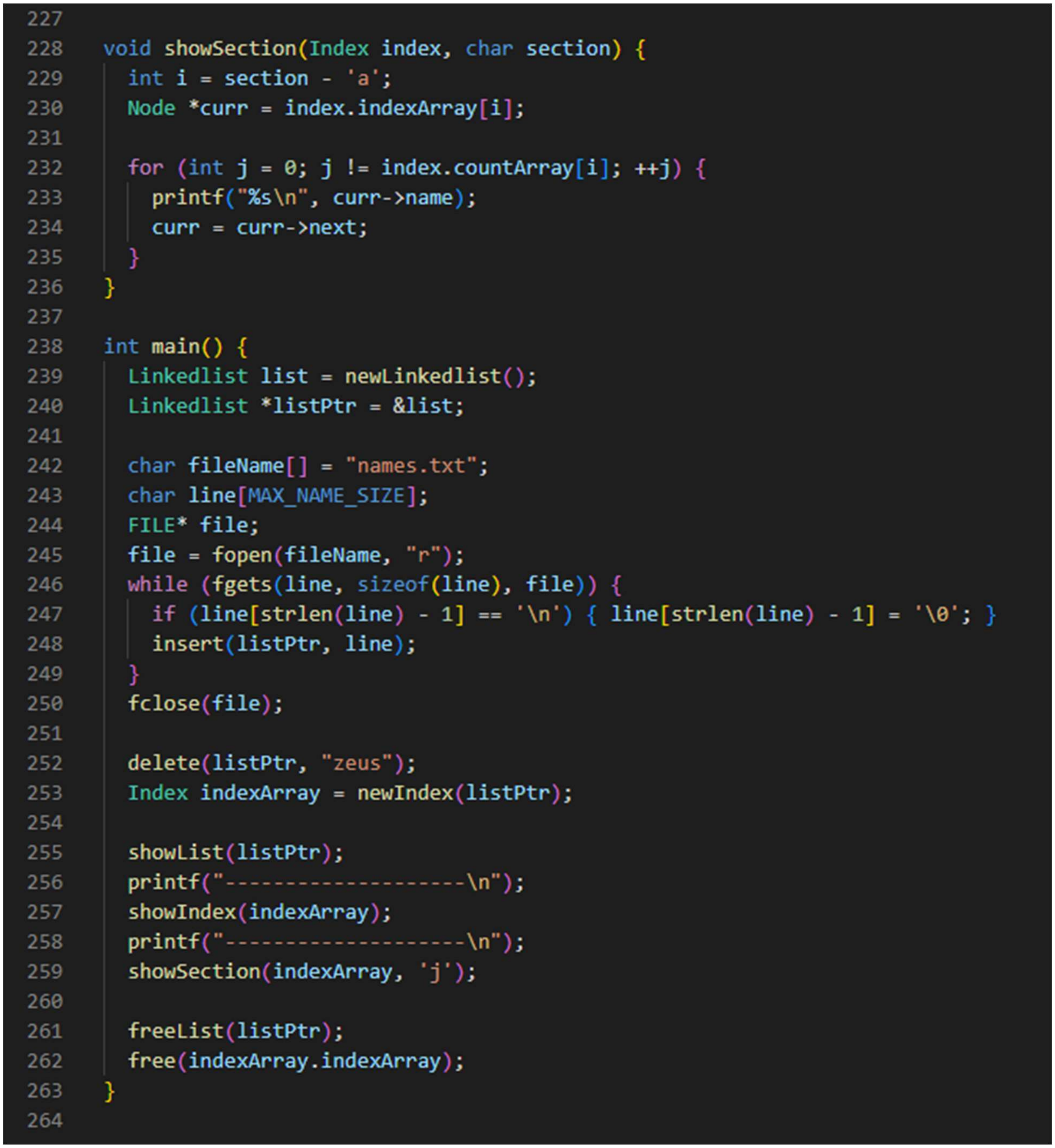


```
void delete(Linkedlist *list, char* name) {
  if (list->length != 0) {
    int num = calcNameCode(name);
    Node *prev = list->start;
    Node *curr = list->start;
    int searching = 1;

    while(searching) {
      if (curr->nameCode == num) {
        searching = 0;

        if (curr->frequency > 1) {
          curr->frequency--;
          list->length--;
          break;
        }
        else if (curr == list->start) { list->start = prev->next; }
        else if (curr == list->end) { list->end = prev; }

        prev->next = curr->next;
        curr->next = NULL;

        list->length -= curr->frequency;
        free(curr);
      }
      else if (curr->next == NULL) {
        searching = 0;
        printf("Did not find %d in list.", num);
      }
      else {
        prev = curr;
        curr = curr->next;
      }
    }
  }
  else { printf("Nothing to delete."); }
}
```



```c
void showList(Linkedlist *list) {
  Node *curr = list->start;

  if (curr == NULL) { printf("List is empty\n"); }
  else {
    printf("Length (including duplicate values): %d\n", list->length);
    printf("Start: %s\n", list->start->name);
    printf("End: %s\n", list->end->name);

    while (curr != NULL) {
      printf("%s\n", curr->name);
      curr = curr->next;
    }
  }
}

void freeList(Linkedlist *list) {
  Node *prev = NULL;
  Node *curr = list->start;

  if (curr == NULL) { printf("List is already empty."); }
  else {
    while (curr != NULL) {
      prev = curr;
      curr = curr->next;
      free(prev);
    }

    list->length = 0;
    list->start = NULL;
    list->end = NULL;
  }
}

void showIndex(Index index) {
  for (int i = 0; i != 26; ++i) {
    char letter = i + 'a';
    if (index.indexArray[i] == NULL) {
        printf("No names starting with letter %c\n\n", letter);
    } else {
        printf("First for letter %c is: %s\n", letter, index.indexArray[i]->name);
        printf("Count for letter %c is: %d\n\n", letter, index.countArray[i]);
    }
  }
}
```

The *showList()* function was updated to print the names of each *Node*. The *showIndex()* function was added, which simply iterates over the *Index* provided and prints the contents of its parallel arrays for each letter if they exist, otherwise it says there were no names added for the letter.



```c

void showSection(Index index, char section) {
  int i = section - 'a';
  Node *curr = index.indexArray[i];

  for (int j = 0; j != index.countArray[i]; ++j) {
    printf("%s\n", curr->name);
    curr = curr->next;
  }
}

int main() {
  Linkedlist list = newLinkedlist();
  Linkedlist *listPtr = &list;

  char fileName[] = "names.txt";
  char line[MAX_NAME_SIZE];
  FILE* file;
  file = fopen(fileName, "r");
  while (fgets(line, sizeof(line), file)) {
    if (line[strlen(line) - 1] == '\n') { line[strlen(line) - 1] = '\0'; }
    insert(listPtr, line);
  }
  fclose(file);

  delete(listPtr, "zeus");
  Index indexArray = newIndex(listPtr);

  showList(listPtr);
  printf("--------------------\n");
  showIndex(indexArray);
  printf("--------------------\n");
  showSection(indexArray, 'j');

  freeList(listPtr);
  free(indexArray.indexArray);
}

```

The last function added is *showSection()*, which uses the *Index* to quickly go to the first *Node* with a name beginning with the given letter, and then prints off each name and incrementing the pointer down the list until it has gone a number of times equal to the count for that index.