Comparing and Contrasting Searching Algorithms and their Specifications

By: Benjamin D. Miller
McMaster University
April 25th, 2017

Contents

Abstract

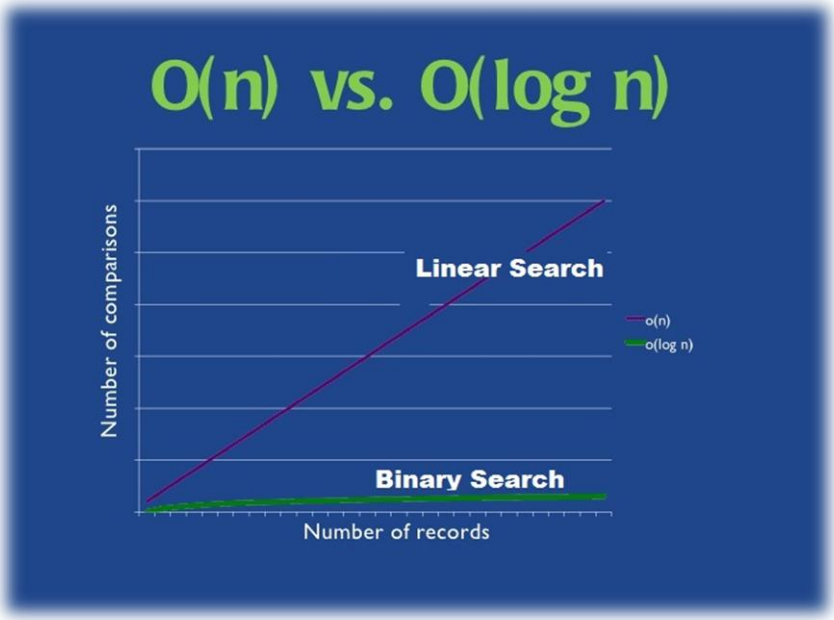
This report aims to identify the key differences between the specifications and correctness implementations of two of today's most well-known searching algorithms; Linear Search, and Binary Search. This report will provide an implementation of these algorithms in C code as well as a formal proof of their correctness using the most recent version of the source-code analysis software for C programs known as Frama-C. Their design and specification properties will be contrasted in their usage in both the array and linked-list data structures. Finally, the report will touch on the problems encountered in building and testing these algorithms in the C language as well as issues proving their correctness in Frama-C. All corresponding code for this paper can be found in the code appendix.

Introduction

Linear search is an intuitive search algorithm that works by incrementing through each element of the data structure that it is being run on until either the element being searched for is found, or the end of the list has been reached. Thus, it is easy to prove that this searching algorithm will have a run time complexity of $O(n)$[1]. Where n is the number of elements in the data structure. The worst case is when the entire list has been scanned, and this is how we can show that the run time complexity is O(n). This may seem practical for a small number of elements but what about when there are millions of elements to be searched for?

Binary search is a popular search algorithm that is commonly implemented with a time complexity of $O(\log n)$[2] with n being the total amount of elements in the data structure. Binary search works by splitting the list of elements in half at each iteration, which can save a lot of time when there are many elements to be searched through. However for most practical purposes the elements must be sorted, which can add complexity.

The importance of these two algorithms and their comparisons will be explored in this paper, as well as their application within the real world and proof of their correctness using Frama-C preconditions, post conditions, and invariants.



The difference in time complexity of the two searching algorithms.
Source: www.Slideshare.com/indexingqueryoptimizations

Linear Search

Linear Search on the surface may appear incredibly intuitive and easy to prove, although it may be intuitive to program, it is not as clear on how to prove that the algorithm executes correctly. I implemented my version of Linear Search with the linked list data structure which brought along its own difficulties when trying to prove its correctness using the Frama-C software verification tool.

Essentially, the program that I wrote keeps track of the head of the linked list, and iterates the head through the entire list until either the element is found or the end of the list has been reached (there are no more numbers to be searched), and then returns either the found number, or a message informing the user that the number has not been found.

Linear search works sequentially, and knowing this, a Frama-C program can be designed with loop invariants such that:

```
/* loop invariant \forall integer i;

0 <= i < n */
```

Where i is the current iteration and n is the length of the linked list. If the iteration is greater than the length of the list, then we are searching outside of the list and will get a segmentation fault.

By ensuring that this property is never violated (making it an invariant and verifying that it is correct in Frama-C) we can ensure that the linear search will always execute, and will always finish (in theory). An outlier could be if an infinitely large linked list was inputted as an argument to be searched through; however, this algorithm would finish given enough time to complete the calculation.

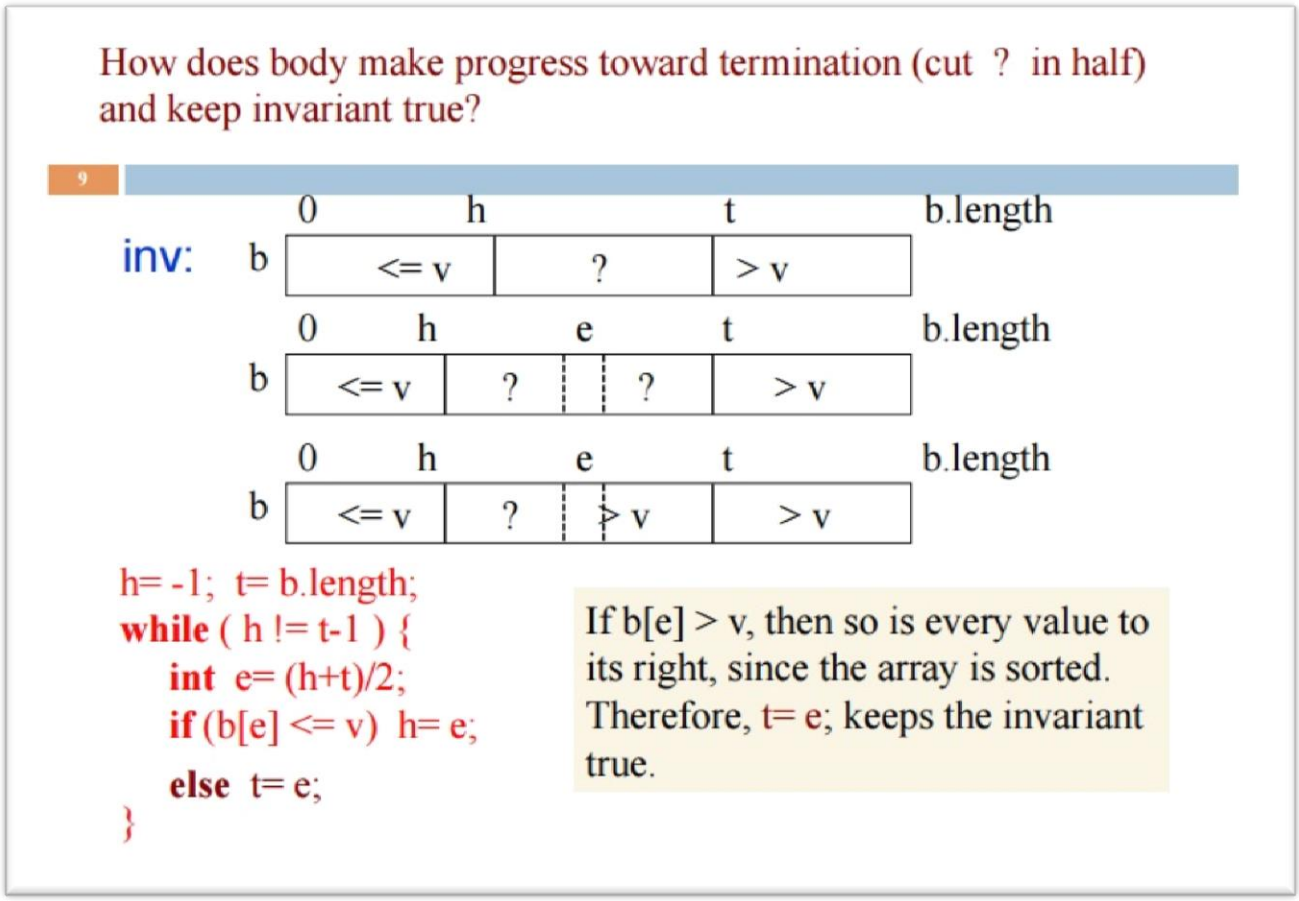
Linear search has practical application uses in embedded systems where every bit of memory counts because it is easy to implement and does not require much memory, as well it works best on an unsorted linked list or array, as it will always return the correct result of whether or not a value is within in a data structure because the entire structure is guaranteed to have been searched.

Binary Search

Despite their popularity among the computing and software community, there are many common misconceptions in regards to binary search, while if asked, most will believe that they have a firm understanding of the algorithm. One of the biggest misconceptions is the idea that in order for a binary search to work properly, the list must be sorted. While in practical use this may be the case, however we can still satisfy the correctness of Binary Search on a list with all duplicate elements.

In order to prove a program correct we need to satisfy two fundamental properties: convergence (the algorithm will stop), and partial correctness (the algorithm will produce the intended result). In my implementation of Binary Search, I proved convergence using the loop invariant of having the first <= middle < last variables of the currently being search subset of the original binary search. Binary search's convergence can be proven by the following loop

invariants:



[4] The following invariants show that as binary search increments, it's remaining list of elements to be searched decreases in size until the while loop condition ends, and the convergence is proven.

General Comparison

A naïve comparison of the two searching algorithms could lead one to believe that linear search has no practical application due to the fact that it has an exponentially higher time complexity in most scenarios. While binary search is generally the accepted searching algorithms in most cases, linear search still has a large impact today and does have its own practical uses.

With more in depth approach we can see that binary search needs more memory allocated to it in order for it to search through its elements, especially if the data structure is a linked list, as the binary search will need to store even more pointers in order to perform the same function. Binary search needs to store the low, middle, and higher values of the array that it is searching, and these values change throughout each iteration of the search. This is why in embedded systems and for smaller list or array sizes, using a variant of linear search is actually more

practical and efficient. Linear search also performs better on an unsorted data structure as it will sort through the entire list, while an improper implementation of binary search may miss the value all together even though it is inside the array or list.

Binary search requires ordering comparisons, while linear search only requires equality comparisons. Binary search also requires essentially random access to the data, while linear search only requires sequential access to data in order to operate.

Binary Search vs. Linear Search in Arrays

The array data structure is unique in that the size of the array must be known ahead of time in order for the array to be valid. This can cause a lot of problems when trying to formally prove a program correct and are constructing an array's size based on user input. The array data structure requires less memory and is less complex than the linked list data structure.

Shuffling an array is complicated and can require more memory than a linked list, and since binary search requires shuffling of index positions at each iteration, it is often not a good choice to implement when using an array. Although it will work perfectly fine, it is better to implement binary search with a linked list. In contrast, due to its low memory requirements and sequential order of comparisons, linear search is the perfect match for searching through the contents of an array data structure.

A binary search algorithm would require a sorted array, and sorting an array is often more difficult to do than sorting a linked list because there are not as many pointers to the data within the array and sorting often has to be done somewhat sequentially. Linear search has no preference for any order and can be done on any random assortment of elements. When all things are considered, linear search is better suited with the array data structure than binary search.

Binary Search vs. Linear Search in Linked Lists

Linked lists are often used when the amount of data is unknown, and they have the ability to grow dynamically[3]. This is done by changing the values of the pointers that point to the beginning and the ending of the list. Due to this nature, shuffling the elements of a linked list requires less memory and can be done much more easily than using an array. This unique property favours the binary search algorithm because in binary search, the "list" that is being searched is constantly being split in half due to the nature of binary search reducing the size of the elements it is searching for.

A linked list is more complex than an array, likewise as binary search is more complex than a simple and intuitive implementation of linear search. This is why for large scale and in

most industry applications the linked list data structure is used in conjunction with binary search in order to provide a fast and reliable way to search through incredibly large amounts of data.

Problems Encountered and Challenges

The most difficult problem that I encountered when attempting to verify these algorithms was sanitizing and formally verifying user inputted variables. In fact, I was never truly able to get Frama-C to recognize user inputted variables, or variables that were inputted via the command line to be proven absolutely correct.

I attempted to solve the above by verifying that the types were of the correct type, I verified that the correct numbers of input arguments were added, and I even attempted to ensure that the user inputted variables were in sequential order for the binary search algorithm. However, despite these efforts Frama-C still did not accept my C programs as valid, and thus cannot be considered truly correct according to my preconditions, post conditions, and invariants that were written in the ACSL language.

Conclusion

In conclusion, linear search and binary search both have very many practically uses in today's society and are the two most well-known searching algorithms for both arrays and linked lists within the software engineering and computer science community. This paper has proven the correctness of both of these algorithms and their implementations within the C language, using the Frama-C software verification tool to prove this formally.

Sources

1 … http://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/

2 … http://www.cs.cornell.edu/courses/cs211/2005sp/Sections/S2/binsearch.pdf

3 … http://stackoverflow.com/questions/166884/array-versus-linked-list

4 … http://www.cs.cornell.edu/courses/cs2110/2014sp/L12-Sorting/L12cs2110BSearchInvariants.pdf

https://www.quora.com/Whats-the-point-of-search-algorithms-e-g-binary-search-in-coding-when-there-are-database-queries-that-you-can-use

Code Appendix

BinarySearchArray.c

```
/* Binary Search on Linked Lists Implementation
 * ========================================
 *
 * This program searches an array using the Binary Search algorithm.
 * This is used to search through a user
 * provided array and value.
 *
 *  USAGE: ./binarysearch [Array Length] [Value to find]
 *  where [Array Length] and [Value] are both integers.
 *
 * By: Benjamin D. Miller
 *  millebd
 *  001416516
 */
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>

bool valid_num(char num[]);

/*@ predicate sorted{L}(long *vals, integer a, integer b) =
  @    \forall integer i,j; a <= i <= j <= b ==> vals[i] <= vals[j];
  @*/

/*@ requires n >= 0 && \valid_range(vals,0,n-1);
  @ ensures -1 <= \result < n;
  @ behavior success:
  @   ensures \result >= 0 ==> vals[\result] == middle;
  @ behavior failure:
  @   assumes sorted(vals,0,n-1);
  @   ensures \result == -1 ==>
  @     \forall integer k; 0 <= k < n ==> vals[k] != middle;
  @*/
int main(int argc, char* argv[]){

    if(argc != 3){
        fprintf("ERROR. Incorrect number of arguments.\n");
        fprintf(stderr,"USAGE: ./binarysearch [Length of list] [Value to
find].\n");

        exit(1); // Return an error code indicating the user input error.
    }

    int n = atoi(argv[1]);
    int val = atoi(argv[2]);

    // If either commandline input isn't an integer, then we must exit
    if(!isNumber(n) || !isNumber(val)){
        frprintf("Error. The arguments were not valid integers.")
        fprintf(stderr,"USAGE: ./binarysearch [Length of list] [Value to
find].\n");

        exit(1); // Exit the program because an invalid integer was entered.
    }
    int vals[n];

    // Retrieve the values of the array
    int i = 0;
```

```
    for(i; i < n; i++){
        printf("Please enter the value at index %d (numbers must be sorted!): ",
i);

        // Validating user input as an integer.
        char term;
        if(scanf("%d%c", &vals[i], &term) != 2 || term != '\n'){
                printf("An invalid integer was entered.\nExiting.\n");
            exit(1)
            }
        else{
                if(!isNumber(vals[i])){
                frprintf("An invalid integer was entered. \nExiting.\n");
                exit(1);
            }
        }

        printf("\n");

        // Verify that the array numbers are entered in order!
        if(i > 0){
            if(vals[i] < vals[i-1]){
                printf("The array is out of order! Try again!\n");
                return -1;
            }
        }

    }

    int first = 0;
    int last = n - 1;
    int middle = (first+last)/2;

    /*@ loop invariant
      @   0 <= first && last <= n-1;
      @ for failure:
      @   loop invariant
      @   \forall integer k; 0 <= k < n && vals[k] == middle ==> first <= k <=
last;
      @ loop variant last-first;
      @*/
    while (first <= last) {
        if (vals[middle] < val){first = middle + 1;}
        else if (vals[middle] == val) {
```

```
            printf("The value %d is in the array at index position %d!\n", val,
middle);
            break;
            }
        else{last = middle - 1;}

        middle = (first + last)/2;

        if (first > last){
            printf("The value %d is not in this array.\n", val);
        }
    }

    return 0;
}

bool valid_num(char num[]){
    //Function to validate integers
        int i = 0;

        //checking for negative numbers
        if (num[0] == '-'){i = 1;}

        for (i; num[i] != 0; i++){
            //if (num[i] > '9' || num[i] < '0')
            if (!isdigit(num[i])){return false;}
        }

    return true;
}
```

LinearSearchLinkedList.c

```c
/* Implementation of Linear Search on Linked Lists
 * ===============================================
 *
 *  USAGE: ./linearsearch [List Length] [Value to find]
 *  where [List Length] and [Value] are both integers.
 *
 *  This program will ask you to enter the values of the list.
 *  Enter them individually, separated by the ENTER key.
 *
 * By: Benjamin D. Miller
 *  millebd
 *  001416516
 */
#include <stdio.h>
#include <stdlib.h>

// Need node structure in order to define Linked Lists
struct node{
    int a;
    struct node *next;
};

int vals[100]; // 100 as a buffer.

void create_list(struct node **, int);
void linear_search(struct node *, int);
void delete(struct node **);


/*@ requires argc > 0;
  @ requries \valid argc;
  @ assigns \nothing;
  @ behavior inlist:
  @     assumes \forall size_t l; 0 <= l < len;
  @ behavior notinlist:
  @     ensures \result == 0;
 */
int main(int argc, char *argv[]){

    if(argc != 3){
        printf("ERROR. Incorrect number of arguments.\n");
        printf("USAGE: ./linearsearch [Length of list] [Value to find].\n");
```

```
        return -1; // Return an error code indicating the user input error.
    }

    struct node *head = NULL;

    int l_length = atoi(argv[1]);
    int val = atoi(argv[2]);

    // Retrieve the values of the array
    int i = 0;
    for(i; i < l_length; i++){
        printf("Please enter the value at index %d: ", i);
        scanf("%d", &vals[i]);
        printf("\n");
    }


        create_list(&head, l_length);

        linear_search(head, val); // Search through the linked list.
        delete(&head); // Clean up and delete the linked list from memory.

    return 0;
}

void create_list(struct node **head, int l_length){
    // Creates a linked list from an array.
        struct node *temp;

    int i = 0;
        for (i; i < l_length; i++){
            temp = (struct node *)malloc(sizeof(struct node));
            temp->a = vals[i]; // Insert the values from the vals array into the
linked list.

        if (*head == NULL){
                    *head = temp;
                    temp->next = NULL;
            }
            else{
                    temp->next = *head;
                    *head = temp;
            }
    }
}
```

```
/*@ requires n >= 0 && \valid_range(vals,0,n-1);
  @ ensures -1 <= \result < n;
*/
void linear_search(struct node *head, int value){
    // Iterate through the list until the value is found or the end of the list
has been reached.

    /* loop invariant \forall integer i;
        0 <= i < n */
    while (head != NULL){
            if (head->a == value){
                    printf("The value %d has been found in the list!\n", value);

            return;
        }

        head = head->next;
        }

        printf("The value %d is not in this list.\n",value);

    return;
}

void delete(struct node **head){
    struct node *temp;

        while (*head != NULL){
            temp = *head;
            *head = (*head)->next;
            free(temp);
        }
}
```