Assignment Two – Searching & Hashing

Ryan Munger
Ryan.Munger1@marist.edu

October 14, 2024

1 Introduction

1.1 Goals

Assignment 2 instructed us to implement sequential/linear search, binary search, and a hash table. We took a random sample of magic items from the full list to demonstrate searching and hash look-ups.

1.2 Write-up Format

In this report I will describe the logic being presented. Below the text explanation, relevant code will follow in both C++ and Ada. I have learned a lot about Ada! I have also found that AI models such as ChatGPT and Gemini are abysmal at writing it, as not even the most rudimentary examples I asked for would even compile. I called upon a grand wizard (my father) to help my with my Ada magic items. I did not use Alire this time, just opting for GNATmake.

1.3 Limerick of Luck

After years of searching
Interviewees found an idol perching.
    The ultimate guess for any question,
    Answering "Hash Table" is rarely a transgression,
Bolstering job-seekers with no besmirching.

Why did they close Hash Street? - *It had too many collisions...*

---

2 Searching

2.1 Sequential/Linear Search

Selection/Linear search is a very simple algorithm. We simply iterate through the array looking for the target. As soon as we find it, we return the index we found it at. If we don't find it, we can return -1 to indicate this. This algorithm is $O(n)$ time as we need to iterate through the length of the array to find our target. On average, it will take us $\frac{n}{2}$ comparisons since sometimes we will find our target early, and sometimes late. So, for 666 magic items, it should take us an average of 333 comparisons.

```cpp
// returns first index of the item if it is in the array
// we don't need a comparison counter as it is 1 more than the index the item is found at (since indexing starts at 0).
template <typename T>
int sequentialSearch(const vector<T>& arr, const T& target) {
    for(size_t i = 0; i < arr.size(); ++i) {
        if(arr[i] == target){return static_cast<int>(i);}
    }
    return -1;
};
```

```ada
   function Sequential_Search -- loop through each item and see if we found it
      (Arr : Vector; Target : Unbounded_String) return Integer is
   begin
      for I in 1 .. Integer(Arr.Elements.Length) loop  -- INDEXING STARTS AT 1!?!?!!
         if Arr.Elements(I) = Target then
            return I;
         end if;
      end loop;
      return -1;
   end Sequential_Search;
```

Notice I did not count comparisons directly in this function. This is because since we are linearly searching the array item by item, the number of comparisons is exactly the index we found the item at + 1 (Since indexing starts at 0 - we need 1 comparison even if it is element 0). If we did not find it, then the amount of comparisons is the length of the array!

2.2 Binary Search

Binary search is much more efficient than sequential search. For this algorithm, the trade-off is that the array must be sorted. Using the fact that the array is sorted to our advantage, we know that a splice of the array after a certain value contains only values greater than it and one before a certain value only contains values less than it. In each iteration of binary search, the search space is halved, resulting in a logarithmic reduction in the number of comparisons. We check half of the half of the half... This leaves binary search at $O(log(n))$. For our array of 666 magic items, we can expect it to take $\log_2(666)$ comparisons, or 9.38. It is base 2 as we are halving.

1. Find the midpoint of the array.
2. If the midpoint is the target, return the midpoint index.
3. If the target is greater than the midpoint, search the half after the midpoint.
4. If the target is less than the midpoint, search the half before the midpoint.
5. Continue calculating midpoints on the smaller halves until we find the element, or our left and right indexes hit each other.

```
// returns first index of the item if it is in the array
int binarySearch(const std::vector<string>& arr, const string& target, int& comparisons) {
    int left = 0;
    int right = arr.size() - 1;

    // if left > right we did not find it
    while (left <= right) {
        int mid = left + (right - left) / 2;

        comparisons++;
        if (toLowerCase(arr[mid]) == toLowerCase(target)) {
            return mid;
        }
        else if (toLowerCase(arr[mid]) < toLowerCase(target)) {
            left = mid + 1; // search right half
        }
        else {
            right = mid - 1; // search left half
        }
    }
    return -1;
};
```

```
   function Binary_Search -- use sorting to break search area in half
     (Arr : Vector; Target : Unbounded_String;
      Comparisons : out Integer) return Integer
   is
      Left : Integer := 0;
```

```
      Right : Integer := Integer (Arr.Elements.Length) - 1;
      Mid : Integer;
      Target_Lower : constant Unbounded_String := To_Lower_Case (Target);
   begin
      Comparisons := 0;
      while Left <= Right loop
         Mid := Left + (Right - Left) / 2;
         Comparisons := Comparisons + 1;

         if To_Lower_Case (Arr.Elements(Mid)) = Target_Lower then
            return Mid;
         elsif To_Lower_Case (Arr.Elements(Mid)) < Target_Lower then
            Left := Mid + 1;  -- search right half
         else
            Right := Mid - 1;  -- search left half
         end if;
      end loop;
      return -1;
   end Binary_Search;
```

I considered implementing **EVIL** Binary Search like we saw in class - *"Yeah man, its in the array..."*

---

3 Hash Table

3.1 How it Works

Hash tables are a very efficient method of storing and looking up data - no sorting required! To begin, we choose the size of our hash table (this can be less than the amount of data we wish to store). Next, we *hash* the value to be stored. This will turn our value into a hash code that indicates where in the hash table array it should be stored. For my hash function, I simply totaled up the ASCII values of the characters of each string item (lowercase), multiplied it by a prime number (to better distribute the hashes), and then took the modulus of the result and the hash table size (so that the index we get is in the array).

What if two values produce the same hash code? This is called a *collision*. There are several ways to remedy this issue such as chaining and probing. I implemented chaining in my hash table. Lets suppose that our value has a hash code of 3. We would place this item at index 3 of our hash table array. If a collision occurred, there is already a value in its spot! We fix this by storing a linked list at each array index. By making each value into a node, we can leave a pointed to the start of the linked list in our hash table array. Now when we need to find a value, we can simply search the (hopefully) small linked list of collided values for a given hash. I will now bombard you with both of my implementations.

```
template <typename T>
struct Node {
    T value;
    Node<T>* next;
};

// utilizing chaining (linked lists) to handle collisions
class HashTable {
    private:
        Node<string>* hashTable[HASH_TABLE_SIZE];

    public:
        // constructor
        HashTable() {
            // initialize the start of table chains to null
            for (int i = 0; i < HASH_TABLE_SIZE; i++) {
                hashTable[i] = nullptr;
            }
        };

        // destructor
        ~HashTable() {
            for (int i = 0; i < HASH_TABLE_SIZE; i++) {
                Node<string>* current = hashTable[i];
                while (current != nullptr) {
                    Node<string>* temp = current;
                    current = current->next;
                    delete temp;  // deallocate
                }
            }
        };

        void put(string str) {
            int hashCode = makeHash(str);
            Node<string>* newItem = new Node<string>;
            newItem->value = str;
            if (hashTable[hashCode] == nullptr) { // start a chain if there isn't one
                newItem->next = nullptr;
                hashTable[hashCode] = newItem;
            } else { // add new element to front of respective chain
                Node<string>* oldFront = hashTable[hashCode];
                newItem->next = oldFront;
                hashTable[hashCode] = newItem;
            };
        };

        // return count of comparisons or -1 if not found
        int get(string str) {
            int hashCode = makeHash(str);
            if (hashTable[hashCode] == nullptr) { // if the pigeonhole doesn't have anything
                return -1;
            } else { // search the pigeonhole's pigeons for value (chain)
```

```cpp
                int getComps = 1; // get is one compare plus chain iterations
                Node<string>* currentNode = hashTable[hashCode];
                while (currentNode != nullptr) {
                    getComps++;
                    if (currentNode->value == str) { return getComps; }
                    currentNode = currentNode->next;
                }
                return -1;
            };
        };

        // use ascii values and a prime to distribute values across table
        int makeHash(string value) {
            value = toLowerCase(value);
            int asciiTotal = 0;
            for(char letter : value) {
                asciiTotal += int(letter); // sorry Alan
                // cout << letter << "    .    " << int(letter) << endl;
            }
            int hashCode = (asciiTotal * 1031) % HASH_TABLE_SIZE; // using a prime, 1031 in honor of halloween!
            return hashCode;
        };

        // use asterisks to visualize the table's population
        void generateHistogram() {
            int pigeons;
            int pigeonHoles = HASH_TABLE_SIZE; // i just wanted to say it

            for (int i = 0; i < pigeonHoles; i++) {
                pigeons = 0;
                Node<string>* currentNode = hashTable[i];
                while (currentNode != nullptr) {
                    pigeons++;
                    currentNode = currentNode->next;
                }
                cout << setw(15) << "Pigeonhole " << i << ": " << string(pigeons, '*') << endl;
            }
        };
};
```

```ada
    -- must declare node to make access to it and then later go back and use its access...
    type Node;
    type Node_Ptr is access all Node;

    type Node is record
        Value : Unbounded_String;
        Next  : Node_Ptr := null;
    end record;

    -- chained hash table so each array element holds node ptr
```

```
    type Hash_Table_Array is array (0 .. HASH_TABLE_SIZE - 1) of Node_Ptr;
    Hash_Table : Hash_Table_Array := (others => null); -- init to null values

    -- need to do a translate with a map as the string are unbounded
    function To_Lower_Case (S : Unbounded_String) return Unbounded_String is
    begin
        return Ada.Strings.Unbounded.Translate(S, Ada.Strings.Maps.Constants.Lower_Case_Map);
    end To_Lower_Case;

    -- total up ascii values to create hash
    function Make_Hash (Value : Unbounded_String) return Natural is
        Ascii_Total : Natural := 0;
    begin
        for C of To_String (Value) loop
            Ascii_Total := Ascii_Total + Character'Pos (C);
        end loop;
        return (Ascii_Total * 1031) mod HASH_TABLE_SIZE; -- mult by prime to spread data
    end Make_Hash;

    -- find where the hash code goes and either start a chain or add to it
    procedure Put (Str : Unbounded_String) is
        Hash_Code : constant Natural := Make_Hash (To_Lower_Case (Str));
        New_Node  : Node_Ptr := new Node'(Value => Str, Next => Hash_Table(Hash_Code));
    begin
        Hash_Table(Hash_Code) := New_Node;
    end Put;

   -- return amount of get+ it took to find a value or -1
   function Get (Str : Unbounded_String) return Integer is
      Hash_Code   : constant Natural := Make_Hash (To_Lower_Case (Str));
      Current     : Node_Ptr := Hash_Table(Hash_Code);
      Comparisons : Integer := 1;
   begin
      while Current /= null loop -- watch for end of chain
         if Current.Value = Str then
            return Comparisons;
         end if;
         Current := Current.Next;
         Comparisons := Comparisons + 1;
      end loop;
      return -1;
   end Get;

   -- visualize the buckets and how full they are
   procedure Generate_Histogram is
        Pigeons : Natural;
        Pigeon_Holes : constant Natural := HASH_TABLE_SIZE;
```

```
    begin
        for I in 0 .. Pigeon_Holes - 1 loop
            Pigeons := 0;
            declare
                Current_Node : Node_Ptr := Hash_Table(I);
            begin
                while Current_Node /= null loop
                    Pigeons := Pigeons + 1;
                    Current_Node := Current_Node.Next;
                end loop;
            end;

            Put(Head("Pigeonhole " & Integer'Image(I) & ": ", 15));
            Put_Line((Pigeons * '*'));
        end loop;
    end Generate_Histogram;
```

Phew! You made it... To visualize how the data structure is formed, I implemented a histogram function. This function simply displays each bucket/pigeonhole and the amount of pigeons stored in it (represented by asterisks). The time complexity for a hash table that has enough space to hold all of the required data with no collisions is constant time $O(1)$! In reality, we will need to iterate over the chain at the given index of the hash table array. This brings us up to $O(1 + \alpha)$ where α is the average chain length. Since we are storing 666 magic items in a hash table of size 250, each chain on average has 2.664 items in it. This leaves our expected comparison count at $1 + (\frac{666}{250})$, or 3.66.

3.2 Testing the Hash Table

Code:

```
    cout << "\nHash Table Testing: " << endl;
    HashTable* hashy = new HashTable;
    hashy->put("Hello!");
    // it will add again, not worth traversing chain just to avoid duplicate
    // a classic space vs time
    hashy->put("Hello!");
    hashy->put("test!");
    hashy->put("Something with a different hash");
    cout << "\"Hello!\" was found in the table with " << hashy->get("Hello!") << " comparisons." << endl;
    if (hashy->get("Not in table...") == -1) {
        cout << "\"Not in table...\" was not found in the table." << endl;
    }
```

Output:

```
Hash Table Testing:
"Hello!" was found in the table with 2 comparisons.
"Not in table..." was not found in the table.
```

Code:

```
   Magic_Table : Hash_Table_Array;
```

```
   New_Line;
   Put_Line ("Hash Table Testing:");
   Put (To_Unbounded_String("Hello!"));
   Put (To_Unbounded_String("Hello!"));
   Put (To_Unbounded_String("test!"));
   Put (To_Unbounded_String("Something with a different hash"));

   -- quick test case without declaring this variable before the main area
   -- pretty neat
   declare
      Comparisons : Integer := Get (To_Unbounded_String ("Hello!"));
   begin
      if Comparisons /= -1 then
         Put_Line ("""Hello!"" was found in the table with " & Integer'Image(Comparisons) & " comparisons.");
      end if;
   end;

   if Get (To_Unbounded_String ("Not in table...")) = -1 then
      Put_Line ("""Not in table..."" was not found in the table.");
   end if;
```

Output:

```
Hash Table Testing:
"Hello!" was found in the table with  1 comparisons.
"Not in table..." was not found in the table.
```

4 Searches in Action

4.1 Random Sample

To test our searching algorithms, we are taking a sample of 42 unique items from the list of Magic Items (length 666). I re-used code from assignment1 to read in the items and sort the array (using my merge sort if you were curious). I then used the c++ sampler to create my 42 item subarray. This was tricky to do, but after seeing how concise this function was (available in c++ 17 or later) on stack overflow, I had to understand and implement it. Interestingly, even though it samples the elements uniquely & randomly, it places them into the new subarray in the relative order they originally were in! Another good way to take the sample would be to shuffle the array and take the first 42 elements, but shuffling the entire array for just 42 items is not as efficient.
I did not muster up anything quite as elegant in Ada but I still wanted to avoid the shuffle. I instead just kept track of the indices I had already used and employed a random number generator. Classic space for time trade!

```cpp
    vector<string> magicItems = getMagicItems(MAGICITEMS_PATH);
    mergeSort(magicItems, 0, magicItems.size() - 1);

    /* taken from stack overflow. Used ChatGPT to find out I needed
     c++ version 17.
    sample() takes range of elements and number to select, selects
    randomly without repetition.
    selected elements are added to the randomSample container with
    back_inserter().
    Rand generator (mt19937) ensures sample is random. After some
    ChatGPT and google, I found out that
    the reason the new selection is sorted is because sample()
    maintains the relative ordering of the elements it selected. */
    vector<string> randomSample;
    const int sample_size = 42;
    sample(magicItems.begin(), magicItems.end(), back_inserter(
    randomSample), sample_size, mt19937{random_device{}()});
```

```ada
   Sample_Size : constant := 42;
   List_Length : constant := 666;
   -- fandom number generator for sampling
   -- taken from stack overflow, I'm not 100% confident I really
    know how it works
   subtype Random_Range is Positive range 1 .. List_Length;
   package Random_Integer is new Ada.Numerics.Discrete_Random (
    Random_Range);
   Gen : Random_Integer.Generator;

    -- get a UNIQUE random sample from our magic items
    -- we do this by keeping track of the elements we've used
   function Get_Random_Sample (Items : Vector; Sample_Size :
    Positive) return Vector is
      Sample : Vector;
      -- default them to false as in not used
      Used_Indices : array (1 .. Natural(Items.Elements.Length)) of
     Boolean := (others => False);
      Random_Index : Positive;
   begin
      Random_Integer.Reset (Gen);  -- initialize the random number
    generator
      while Natural(Sample.Elements.Length) < Sample_Size loop
         Random_Index := Random_Integer.Random (Gen) mod Natural(
    Items.Elements.Length) + 1;
         if not Used_Indices(Random_Index) then
            Sample.Elements.Append (Items.Elements(Random_Index));
            Used_Indices(Random_Index) := True;
         end if;
      end loop;
      return Sample;
   end Get_Random_Sample;
```

4.2 Testing the Searches

Using the sample, we found each of the 42 elements in the original array of 666 items using sequential search, binary search, and our hash table. I then took an average of the trials (to two decimal places).
Since the sample is in order - it is fun to watch the number of comparisons needed by sequential sort gradually increase!
To begin, we need to first load our hash table with all of the magic items.

```
    HashTable* magicTable = new HashTable;
    for (string item : magicItems) {
        magicTable->put(item);
    }
    cout << "\nMagic Items Table Visualization: " << endl;
    magicTable->generateHistogram();
```

```
   -- read in magic items, apparently its best practice to put a space before the ()
   Magic_Items := Get_Magic_Items (MAGICITEMS_PATH);
   Selection_Sort (Magic_Items);

   Random_Sample := Get_Random_Sample (Magic_Items, Sample_Size);

   -- load up our hash table
   for Item of Magic_Items.Elements loop
      Put (Item);
   end loop;
   -- make sure its filled up and distributed
   New_Line;
   Put_Line ("Magic Items Table Visualization:");
   Generate_Histogram;
```

```
Magic Items Table Visualization:
    Pigeonhole 0:   ****
    Pigeonhole 1:   **
    Pigeonhole 2:   *****
    ...
    Pigeonhole 248: ***
    Pigeonhole 249: ***
```

Now, let's go through each sampled item and find it using each method! Prepare for another code barrage...

```
    cout << "\nSearching for a random sample:\n" << endl;
    // since the random sample is in order (relative to the sorted array),
    // as we progress through the sample, comparisons will always increase for seq search!
    int totalSeqComps = 0, totalBinComps = 0, totalHashGet = 0;
    int binaryComps = 0, hashGet = 0;
    int foundIdx;
    for (string item : randomSample) {
        foundIdx = sequentialSearch(magicItems, item);
        if(foundIdx != -1){
            cout << "\"" << item << "\" found with Sequential Search at idx: " << foundIdx
```

```
                << ". It took " << foundIdx + 1 << " Comparisons." << endl;
        } else {
            cout << "\"" << item << "\" was not found in magicItems. Comparisons: " << magicItems.size() << endl;
            foundIdx = magicItems.size() - 1; // since we are adding one later
        }
        totalSeqComps += foundIdx + 1;

        foundIdx = binarySearch(magicItems, item, binaryComps);
        if(foundIdx != -1){
            cout << "\"" << item << "\" found with Binary search at idx: " << foundIdx
                << ". It took " << binaryComps << " Comparisons." << endl;
        } else {
            cout << "\"" << item << "\" was not found in magicItems. Comparisons: " << binaryComps << endl;
        }
        totalBinComps += binaryComps;
        binaryComps = 0;

        hashGet = magicTable->get(item);
        if(hashGet != -1){
            cout << "\"" << item << "\" found with Hash Table with " << hashGet
                << " get+ comparisons." << endl;
        } else {
            cout << "\"" << item << "\" was not found in magicItems w/ hash table." << endl;
        }
        totalHashGet += hashGet;
    }
    cout << "\nSequential/Linear search took an average of "
        << fixed << setprecision(2) // Set fixed-point notation and precision
        << static_cast<double>(totalSeqComps) / randomSample.size() // cast double so we don't lose our decimal accuracy
        << " comparisons to find each element." << endl;

    cout << "\nBinary search took an average of "
        << fixed << setprecision(2)
        << static_cast<double>(totalBinComps) / randomSample.size()
        << " comparisons to find each element." << endl;

    cout << "\nHash Table took an average of "
        << fixed << setprecision(2)
        << static_cast<double>(totalHashGet) / randomSample.size()
        << " comparisons to find each element." << endl;
```

```
   New_Line;
   Put_Line ("Generating and searching for a random sample:");
   for Item of Random_Sample.Elements loop
      -- sequential search
      Found_Idx := Sequential_Search (Magic_Items, Item);
      if Found_Idx /= -1 then
        -- once again, i do not like that "" is escaped "
```

```
            Put_Line ("""" & To_String(Item) & """ found with Sequential Search at idx: " &
                      Integer'Image(Found_Idx) & ". It took " & Integer'Image(Found_Idx + 1) & " Comparisons.");
            Total_Seq_Comps := Total_Seq_Comps + Found_Idx + 1;
         else
            Put_Line ("""" & To_String(Item) & """ was not found in magicItems. Comparisons: " &
                      Integer'Image(Natural(Magic_Items.Elements.Length)));
            Total_Seq_Comps := Total_Seq_Comps + Natural(Magic_Items.Elements.Length);
         end if;

         -- Binary Search
         Found_Idx := Binary_Search (Magic_Items, Item, Binary_Comps);
         if Found_Idx /= -1 then
            Put_Line ("""" & To_String(Item) & """ found with Binary search at idx: " &
                      Integer'Image(Found_Idx) & ". It took " & Integer'Image(Binary_Comps) & " Comparisons.");
         else
            Put_Line ("""" & To_String(Item) & """ was not found in magicItems. Comparisons: " &
                      Integer'Image(Binary_Comps));
         end if;
         Total_Bin_Comps := Total_Bin_Comps + Binary_Comps;

         -- hash table lookups
         Hash_Get := Get (Item);
         if Hash_Get /= -1 then
            Put_Line ("""" & To_String(Item) & """ found with Hash Table with " &
                      Integer'Image(Hash_Get) & " get+ comparisons.");
         else
            Put_Line ("""" & To_String(Item) & """ was not found in magicItems w/ hash table.");
         end if;
         Total_Hash_Get := Total_Hash_Get + Hash_Get;
      end loop;

      New_Line;
      -- print averages.. i know the output looks like garbage
      -- I read lots of stack overflow about printing these and solutions either
      -- gave me a compiler error or wanted me to use many lines of put. I choose my ugly put line.
      Put_Line ("Sequential/Linear search took an average of " &
                Float'Image(Float(Total_Seq_Comps) / Float(Sample_Size)) & " comparisons to find each element.");
      Put_Line ("Binary search took an average of " &
                Float'Image(Float(Total_Bin_Comps) / Float(Sample_Size)) & " comparisons to find each element.");
      Put_Line ("Hash Table took an average of " &
                Float'Image(Float(Total_Hash_Get) / Float(Sample_Size)) & " comparisons to find each element.");
```

Output:

```
Searching for a random sample:

"Arachnid tome" found with Sequential Search at idx: 28. It took 29 Comparisons.
"Arachnid tome" found with Binary search at idx: 28. It took 10 Comparisons.
"Arachnid tome" found with Hash Table with 6 get+ comparisons.
"Armatha's long sword" found with Sequential Search at idx: 30. It took 31 Comparisons.
"Armatha's long sword" found with Binary search at idx: 30. It took 9 Comparisons.
"Armatha's long sword" found with Hash Table with 7 get+ comparisons.
...
"Vambraces of Unarmed Prowess" found with Sequential Search at idx: 633. It took 634 Comparisons.
"Vambraces of Unarmed Prowess" found with Binary search at idx: 633. It took 10 Comparisons.
"Vambraces of Unarmed Prowess" found with Hash Table with 2 get+ comparisons.
"Vaporizer" found with Sequential Search at idx: 635. It took 636 Comparisons.
"Vaporizer" found with Binary search at idx: 635. It took 9 Comparisons.
"Vaporizer" found with Hash Table with 2 get+ comparisons.

Sequential/Linear search took an average of 388.43 comparisons to find each element.
Binary search took an average of 8.55 comparisons to find each element.
Hash Table took an average of 3.12 comparisons to find each element.
```

4.3 Comparing the Searches

Even though the code gave us the average of the 42 lookups, I went ahead and ran it 20 times, averaging the averages.

Search	Time Complexity	Expected	Actual
Sequential Search	$O(n)$	333	338.13
Binary Search	$O(log(n))$	9.38	8.77
Hash Table	$O(1+\alpha)$	3.66	3.17

Our searching algorithms are performing just as we expected! There is a tradeoff and use case for each algorithm. Binary search required sorted data but is incredibly fast. Sequential search is fantastic if data is randomly ordered and perhaps we only need to do a lookup or two. Hashing is an all around solution, with the tradeoff being that a hash table must be created and hashes computed. If array index of the item is required, this may not be the best solution.

Not exactly a joke - but when I file things I use binary search to find them!
