CSCI 104
Searching and Sorted Lists
Mark Redekopp
David Kempe
Linear Search

• Search a list (array) for a specific value, k, and return the location
  • Sequential Search
    – Start at first item, check if it is equal to k, repeat for second, third, fourth item, etc.
  • O( ___ )
  • O(n)
Binary Search

- Sequential search does not take advantage of the ordered (a.k.a. sorted) nature of the list
  - Would work the same (equally well) on an ordered or unordered list
- Binary Search
  - Take advantage of ordered list by comparing $k$ with middle element and based on the result, rule out all numbers greater or smaller, repeat with middle element of remaining list, etc.
Binary Search

• Search an ordered list (array) for a specific value, k, and return the location

• Binary Search
  – Compare k with middle element of list and if not equal, rule out ½ of the list and repeat on the other half
  – "Range" Implementations in most languages are [start, end)
  – Start is inclusive, end is non-inclusive (i.e. end will always point to 1 beyond true ending index to make arithmetic work out correctly)

```cpp
int bsearch(vector<int> mylist, 
            int k, 
            int start, int end)
{
    // range is empty when start == end
    while(start < end){
        int mid = (start + end)/2;
        if(k == mylist[mid])
            return mid;
        else if(k < mylist[mid])
            end = mid;
        else
            start = mid+1;
    }
    return -1;
}
```
Binary Search

```cpp
int bsearch(vector<int> mylist, int k, int start, int end)
{
    // range is empty when start == end
    while(start < end){
        int mid = (start + end)/2;
        if(k == mylist[mid])
            return mid;
        else if(k < mylist[mid])
            end = mid;
        else
            start = mid+1;
    }
    return -1;
}
```
Prove Time Complexity

• \( T(n) = \)
Search Comparison

- Linear search = $O(______)$
- Precondition: None
- Works on (ArrayList / LinkedList)

- Binary Search = $O(______)$
- Precondition: List is sorted
- Works on (ArrayList / LinkedList)

```cpp
int search(vector<int> mylist, int k) {
    int i;
    for(i=0; i < mylist.size(); i++){
        if(mylist[i] == k) {
            return i;
        }
    }
    return -1;
}
```

```cpp
int bsearch(vector<int> mylist, int k, int start, int end) {
    int i;
    // range is empty when start == end
    while(start < end){
        int mid = (start + end)/2;
        if(k == mylist[mid])
            return mid;
        else if(k < mylist[mid])
            end = mid;
        else {
            start = mid + 1;
        }
    }
    return -1;
}
```
Search Comparison

• Linear search = $O(n)$
• Precondition: None
• Works on ArrayList or LinkedList

• Binary Search = $O(\log(n))$
• Precondition: List is sorted
• Works on ArrayList only

```cpp
int search(vector<int> mylist, int k) {
    int i;
    for(i=0; i < mylist.size(); i++){
        if(mylist[i] == k)
            return i;
    }
    return -1;
}
```

```cpp
int bsearch(vector<int> mylist, int k, int start, int end) {
    int i;
    // range is empty when start == end
    while(start < end) {
        int mid = (start + end)/2;
        if(k == mylist[mid])
            return mid;
        else if(k < mylist[mid])
            end = mid;
        else {
            start = mid+1;
        }
    }
    return -1;
}
```
Introduction to Interpolation Search

• Given a dictionary, if I say look for the word 'banana' would you really do a binary search and start in the middle of the dictionary?

• Assume a uniform distribution of 100 random numbers between [0 and 999]
  – [679 372 554 ... ]

• Now sort them
  – [002 009 015 ... ]

• At what index would you start looking for key=130
Linear Interpolation

- If I have a range of 100 numbers where the first is 400 and the last is 900, at what index would I expect 532 (my target) to be?

\[
\frac{\text{EndIdx} - \text{StartIdx} + 1}{\text{EndVal} - \text{StartVal}} = \frac{\text{desiredIdx} - \text{StartIdx}}{\text{target} - \text{StartVal}}
\]

\[
\frac{(\text{EndIdx} - \text{StartIdx} + 1)}{\text{EndVal} - \text{StartVal}} \times \frac{(\text{target} - \text{StartVal})}{\text{EndVal} - \text{StartVal}} + \text{StartIdx} = \text{desiredIdx}
\]

\[
(532 - 400) \times \frac{100}{500} + 0 = \text{desiredIdx}
\]

\[
132 \times 0.2 = \text{desiredIdx}
\]

\[
26.4 = \text{desiredIdx}
\]

\[
\text{floor}(26.4) = 26 = \text{desiredIdx}
\]
Interpolation Search

• Similar to binary search but rather than taking the middle value we compute the interpolated index

```c
int bin_search(vector<int> mylist, int k, int start, int end)
{
    // range is empty when start == end
    while(start < end){
        int mid = (start + end)/2;
        if(k == mylist[mid])
            return mid;
        else if(k < mylist[mid])
            end = mid;
        else
            start = mid+1;
    }
    return -1;
}

int interp_search(vector<int> mylist, int k, int start, int end)
{
    // range is empty when start > end
    while(start <= end){
        int loc =
            interp(mylist, start, end, k);
        if(k == mylist[loc])
            return loc;
        else if(k < mylist[loc])
            end = loc;
        else
            start = loc+1;
    }
    return -1;
}
```
Another Example

• Suppose we have 1000 doubles in the range 0-1
• Find if 0.7 exists in the list and where
• Use interpolation search
  – First look at location: 0.7 * 1000 = 700
  – But when you pick up List[700] you find 0.68
  – We know 0.7 would have to be between location 700 and 100 so we narrow our search to those 300
• Interpolate again for where 0.7 would be in a list of 300 items that start with 0.68 and max value of 1
  – (0.7-0.68)/(1-0.68) = 0.0675
  – Interpolated index = floor( 700 + 300*0.0675 ) = 720
  – You find List[720] = 0.71 so you narrow your search to 700-720
• Interpolate again
  – (0.7-0.68)/(0.71-0.68) = 0.6667
  – Interpolated index = floor( 700 + 20*0.6667 ) = 713

Interpolation Search Summary

• Requires a sorted list
  – An array list not a linked list (in most cases)
• Binary search = \(O(\log(n))\)
• Interpolation search = \(O(\log(\log(n)))\)
  – If \(n = 1000\), \(O(\log(n)) = 10\), \(O(\log(\log(n))) = 3.332\)
  – If \(n = 256,000\), \(O(\log(n)) = 18\), \(O(\log(\log(n))) = 4.097\)
• Makes an assumption that data is uniformly (linearly) distributed
  – If data is "poorly" distributed (e.g. exponentially, etc.), interpolation search will break down to \(O(\log(n))\) or even \(O(n)\)
  – Notice interpolation search uses actual values (target, startVal, endVal) to determine search index
  – Binary search only uses indices (i.e. is data agnostic)
• Assumes some 'distance' metric exists for the data type
  – If we store Webpage what's the distance between two webpages?
SORTED LISTS
Overview

• If we need to support fast searching we need sorted data

• Two Options:
  – Sort the unordered list (and keep sorting when we modify it)
  – Keep the list ordered as we modify it

• Now when we insert a value into the list, we'll insert it into the required location to keep the data sorted.

• See example

```
push(7)  0  1  2  3
push(3)  0  1  3  7
push(8)  0  1  2  3  7  8
push(6)  0  1  2  3  6  7  8
```
**Sorted Input Class**

- **insert()** puts the value into its correct ordered location
  - Backed by array: $O(\ )$
  - Backed by LinkedList: $O(\ )$
- **find()** returns the index of the given value
  - Backed by array: $O(\ )$
  - Backed by LinkedList: $O(\ )$

```cpp
class SortedIntList
{
    public:
        bool empty() const;
        int size() const;
        void insert(const int& new_val);
        void remove(int loc);

        // can use binary or interp. search
        int find(int val);

        int& get(int i);
        int const & get(int i) const;
    private:
        ???
};
```
Sorted Input Class

• Assume an array based approach, implement insert()

```cpp
class SortedIntList
{
    public:

    private:
        int* data; int size; int cap;
};

void SortedIntList::insert(const int& new_val) {
}
```
XKCD #724

Courtesy of Randall Munroe
@ http://xkcd.com