CS 261-020
Data Structures

Lecture 8
Binary Search
Midterm Review
1/27/22, Thursday
Odds and Ends

• Assignment 2 due Sunday midnight
• Quiz 4 unlocked after today’s lecture

• Assignment 1 demo due Friday (1/28)

• Midterm:
  • Tuesday (2/1) during lecture time, same classroom
  • Review today
Lecture Topics:

• Binary Search
• Midterm Review
Binary Search

• Important to be able to search through a collection of elements
  • i.e., find the index of an element
  • Determine that element does not exist

• How to do this using linear data structures we’ve seen?
  • By iterating through elements one by one until we find the element or the end of the collection (doesn’t exist)
  • This is called linear search
  • Runtime Complexity: $O(n)$ where $n$ is number of elements of the collection

• Can we improve this?
**Binary Search**

- Can you perform a search for 65 in the following array:

| 3 | 8 | 12 | 14 | 23 | 40 | 48 | 51 | 63 | 65 | 71 | 79 | 83 | 89 | 90 | 100 |

- What do you notice?
  - The array is sorted
  - Each iteration, eliminate half of the remaining array

- Binary Search: iterate through an ordered (sorted) array, and repeatedly divide the search interval in half

- Run Complexity: $O(\log n)$
Binary Search vs. Linear Search

• Searching in an array of size $n = 1,000,000$
  • Linear Search: $O(n) = 1,000,000$ comparisons, on average
  • Binary Search: $O(\log n) \approx 20$ comparisons, on average

• Searching in an array of size $n = 4,000,000,000$
  • Linear Search: $O(n) = 4,000,000,000$ comparisons, on average
  • Binary Search: $O(\log n) \approx 32$ comparisons, on average

→ Binary Search is a lot faster, especially for large values of $n$
How does Binary Search work?

• At each iteration:
  • Compare the query value (the value it’s searching for) to the value at the midpoint of the array
  • If they matches, break and return (i.e., index)
  • Otherwise …
    • If query value < array’s midpoint value, repeat only on the “lower” half of the array
    • If query value > array’s midpoint value, repeat only on the “upper” half of the array
  • If the array under consideration has size 0, break and return. The query value does not exist. (i.e., -1 or where it should be inserted to maintain a sorted array)
How does Binary Search work?

• Iteration:

```
int binary_search(int q, int* array, int n) {
    int mid, low = 0, high = n - 1;
    while (low <= high) {
        mid = (low + high) / 2;
        if (array[mid] == q)
            return mid;
        else if (array[mid] < q)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return low;
}
```

• low – the first index of the sub-array
• high – the last index of the sub-array
• mid – the index of the midpoint of the sub-array
How does Binary Search work?

• Recursion:

```c
int binary_search(int q, int* array, int low, int high) {
    while (low <= high) {
        int mid = (low + high) / 2;
        if (array[mid] == q)
            return mid;
        else if (array[mid] < q)
            return binary_search(q, array, mid+1, high);
        else
            return binary_search(q, array, low, mid-1);
    }
    return low;
}
```

• low – the first index of the sub-array
• high – the last index of the sub-array
• mid – the index of the midpoint of the sub-array
Ordered Array

• Note: Binary Search can only work within an ordered (sorted) array
  • The assumption that allows binary search to eliminate half of the array at each iteration

• Make sure the array is sorted before using binary search!
  • Using a sorting algorithm
  • Using binary search
Ordering Array using a sorting algorithm

• Using a sorting algorithm to order an array

• Runtime complexity of the best general-purpose sorting algorithm
  • $O \left( n \log n \right)$
  • Best if we limit the number of times to “sort”

• Examples:
  • Look up in a phone book
  • Look up a word in a dictionary

• What if we expected new elements to be inserted frequently?
Ordering Array using Binary Search

• If data is frequently changing (i.e., insertion), run binary search after each insertion to maintain an ordered array
  • Recall: binary_search() may return the index where an element should be inserted
  • Thus, the cost of each insertion:
    • O(log n) to identify the index to be inserted using binary search
    • O(n) to shift the subsequent elements back one spot
    • Since O(n) dominates O(log n), the cost of each insertion is O(n)

• The total cost of n insertion is O(n^2)
Midterm

• 2/1 Tuesday during lecture time (2:00 – 3:20)
• Same classroom
• Close book, close notes
• No calculator allowed
• Question types: multiple choices, T/F, short answer
  • Similar to your weekly quizzes
• Bring pencil/pen, and your photo ID (student ID/driver license/passport)
• Scratch paper will be provided if needed
Midterm

• Topics: Week 1-4 (lecture 1-8):
  • C Basics
    • scanf()/printf()
    • Conditionals and loops
    • Struct
  • Pointers
    • void*
    • Stack vs. heap
    • malloc(), free()
  • C strings
  • Function pointers
Midterm

• Topics: Week 1-4 (lecture 1-8):
  • Dynamic Arrays
    • Struct: data, size, capacity
    • Basic operations:
      • get()
      • set()
      • insert()
        • When to resize?
      • remove()
  • Linked List
    • Struct: val, next pointer
    • Basic operations:
      • Insert()
      • Remove()
Midterm

• Topics: Week 1-4 (lecture 1-8):
  • Complexity Analysis
    • Big O
    • Compute Runtime complexity
      • Dominant Components
    • Best, worst, and average cases
    • Dynamic Array insertion
    • Linked list insertion
  • Stack
    • LIFO
    • Basic Operations:
      • Push()
      • Pop()
    • Implement stack using linked list vs. dynamic array
      • Complexity
Midterm

• Topics: Week 1-4 (lecture 1-8):
  • Queue
    • FIFO
    • Basic Operations
      • Enqueue()
      • Dequeue()
    • Implement queue using linked list vs. dynamic array
      • Complexity
      • Circular buffer: logical index vs. physical index
  • Deque
    • Basic operations:
      • Add front
      • Add back
      • Remove front
      • Remove back
Midterm

• Topics: Week 1-4 (lecture 1-8):
  • Deque
    • Implement deque using doubly linked list
      • Sentinels
      • Complexity
  • Encapsulation
  • Iterator
    • Next()
    • Has_next()
  • Binary Search
    • collection must be sorted
    • Complexity