Lec_36.md

Lecture 36

Dec. 09/2020

Final Lecture! :O

Big-O Notation and Time Complexity

Big-O notation is useful because it provides a means for comparing algorithms

- Can use BigO to compare effectiveness of two algorithms by measuring run-time

However, it has shortcomings:

1. Imprecise

- **Big-O** notation loses exact time
- `O(n^2)` can be equal to `3n^2` or `1500n^2`

2. Only works for *large* n

- Compare a runtime of `10log(n)` versus `2n`
  - For small `n` (for example `n<100`), `10log(n)` is actually *significantly* slower

3. Can be misused/misinterpreted

- `T(n)=O(g(n))`
- Looking for the slowest growing `g(n)` that bounds `T(n)` from above
  - Worst-case analysis
- But what above best-case or average-case analysis?
  - Look at `omega(g(n))` and `theta(g(n))`
  - Requires extra computations

The Search Problem

Given a collection of `n` items, each associated with a *unique* key, and a key `k`

- Does there exist an item `J` in the collection s.t. `key(J)=k` ?
- Ubiquitous problem!
  - Think about online authentication
  - Database searching

So given `n` items, how do we organize them to make search more efficient?

- Look at several different organizations
  - Use **Big-O** notation to determine which one is more efficient
- Examine efficiency of searching for an item with key `k`

Unsorted Array

`n` items in no particular order, and in array format

Searching

1. Search Algorithm

- Suppose linear search

2. Time Complexity

- Worst Case: `O(n)`
- Best Case: `O(1)`
- Average: `O(n)`
  - On average will look through `n/2` elements

Inserting

1. Insertion Algorithm

- Suppose insertion search

2. Time Complexity

- `O(1)`
  - Just insert anywhere (unsorted)

Sorted Array

`n` items sorted, and in array format

Searching

1. Search Algorithm

- Binary search
  - Look left, look right

2. Time Complexity

- Worst Case: `O(log(n))`
- Best Case: `O(1)`
- Average: `O(log(n))`

Inserting

1. Insertion Algorithm

- Suppose insertion search

2. Time Complexity

- Worst Case: `O(n)`
  - Need to expand size of array (shift elements to make space for inserting element)
- Best Case: `O(n)`

Unsorted Linked Lists

`n` items in no particular order, and in a linked list

Searching

1. Search Algorithm

- Linear search
  - Its a linked list

2. Time Complexity

- Worst Case: `O(n)`
- Best Case: `O(1)`
- Average: `O(n)`

Inserting

1. Insertion Algorithm

- Suppose insertion search

2. Time Complexity

- `O(1)`
  - It's unsorted - just insert anywhere

Sorted Linked Lists

`n` items sorted, and in a linked list

Searching

1. Search Algorithm

- Linear search
  - Its a linked list

2. Time Complexity

- Worst Case: `O(n)`
- Best Case: `O(1)`
- Average: `O(n)`

Inserting

1. Insertion Algorithm

- Suppose insertion search

2. Time Complexity

- Worst Case: `O(n)`
  - Scan through linked list and insert `O(1)` at end
- Best Case: `O(1)`
  - If insert at head, can just modify pointers

Binary Search Tree

`n` items in a Binary Search Tree

- Tree sorted by definition

Searching

1. Search Algorithm

- Linear search
    - Its a linked list

2. Time Complexity

- Worst Case: `O(log(n))`
    - But could be `O(n)` for unbalanced trees
- Best Case: `O(1)`
- Average: `O(log(n))`

Inserting

1. Insertion Algorithm

- Suppose insertion search

2. Time Complexity

- Worst Case: `O(n)`
- Best Case: `O(log(n))`

Beauty of **Binary Search Trees**:

- Get the search time of **binary search** with a *dynamic* data type