CS321 Spring 2021

Lecture 2

Jan 13 2021

Admin

- A1 Due next Saturday Jan 23rd – 11:59PM

Course in 4 Sections

- Section I: Basics and Sorting
- Section II: Hash Tables and Basic Data Structs
- Section III: Binary Search Trees
- Section IV: Graphs

Section I

- Sorting methods and Data Structures
- Introduction to Heaps and Heap Sort

What is Big O notation?

- A way to approximately count algorithm operations.
- A way to describe the worst case running time of algorithms.
- A tool to help improve algorithm performance.
- Can be used to measure complexity and memory usage.

Bounds on Operations

- An algorithm takes some number of ops to complete:
- a + b is a single operation, takes 1 op.
- Adding up N numbers takes N-1 ops.
- O(1) means 'on order of 1' operation.
- O(c) means 'on order of constant'.
- O(n) means ' on order of N steps'.
- O(n^2) means ' on order of N*N steps'.

How Does $O(k) = O(1)$

$O(n) = c * n$ for some c where $c*n$ is always greater than n for some c.

$O(k) = c*k$
$O(1) = cc * 1$

let $ccc = c*k$
$c*k = c*k* 1$ therefore $O(k) = c * k * 1 = ccc *1 = O(1)$

O(n) times for sorting algorithms.

Technique	O(n) operations	O(n) memory use
Insertion Sort	$O(N^2)$	O(1)
Bubble Sort	$O(N^2)$	O(1)
Merge Sort	N * log(N)	O(1)
Heap Sort	N * log(N)	O(1)
Quicksort	$O(N^2)$	O(logN)

Memory is in terms of EXTRA memory

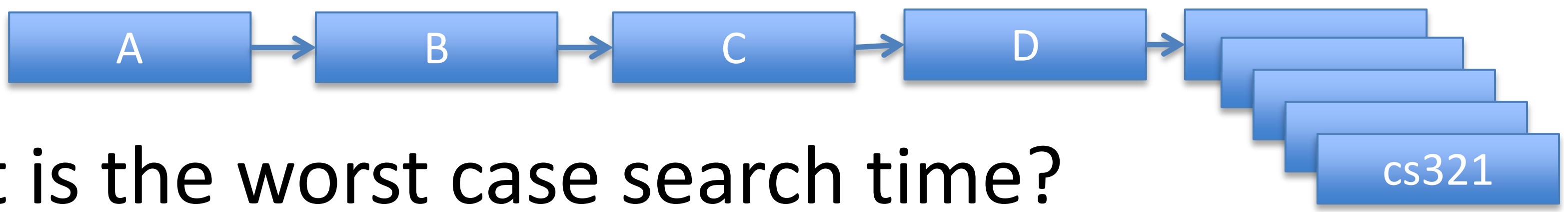
Primary Notation Types

- $O(n)$ = Asymptotic upper bound. Longest
- $\Omega(n)$ = Asymptotic lower bound. Quickest.
- $\Theta(n)$ = Both lower and upper bound.

- *side note these are capital greek letters, hence ‘Big O’.

Find Item in Linked List

Have a list of N strings. Want to see if the string 'cs321' is in the list.



What is the worst case search time?

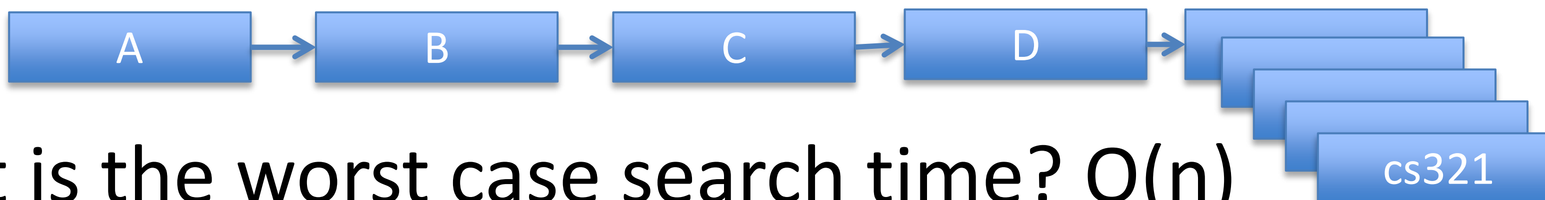
What is the best case search time?

What is the average case search time?

Is there a Θ bound?

Find Item in Linked List

Have a list of N strings. Want to see if the string 'cs321' is in the list.



What is the worst case search time? O(n)

What is the best case search time?O(1)

What is the average case search time?O(n/2)

Is there a Θ bound?No

Linked List Cache

- Basic Algorithm:
- Initialize data structures – Cache, temp variables.
- Read in text file as a single string.
- For each word in text file:
  - Is word in cache? // a cache method.
  - Yes – continue for loop.
  - No – addWordToCache. //cache method
- Print Cache Statistics – // a cache method.

Word In Cache?

- inCache(w)
- {
  - Increment cache 1 reference count.
  - Search Linked List cache1 for w // Big O time?
  - If w in cache1 {increment cache1.hit, Move item to front of list. return true}
  - Else if no cache2 return false.
  - Increment cache2 reference
  - Search linked list cache2 for w // Big O time?
  - If w in cache2 {increment cache2.hit, move to front of Cache 2 and add to cache 1, return true}
- }

addCacheWord

- addWord(w) // add word to cache.
- // not counting this as a cache reference, why?
- If(cache1 full){ remove last item in list}.//cost?
- Add w to front of cache1.
- If (cache2) {
  - If (cache2 full) { remove last item in list // cost?
  - Add w to front of cache2
- }

What Causes Cache1 to be different from Cache2?

Basic Computing Problem

- **Sort a list of numbers in the quickest time**
- A=[1,9,2,8,4,5,0,3,6,7]

```
For i = 0 to 9:
    for j = i to 9:
        if (A[j] < A[i] ) swap(A,i,j);
```

What is the runtime of this?

Answer O(n^2)

- Very simple sorting algorithm takes O(n^2).

- Can we do better?
- If the world were magic what is the best we could do?
  - O(n)
- So should be able to do better than O(n^2)

Heapsort

- A sorting algorithm that uses a specific type of data structure: **Max-Heap**
- Has a worst case and best case performance of $\Theta(n*\log(n))$.

- Point1: Choice of data structure critical for algorithm performance.
- Point2: Additional example of Big-O analysis.

Components.

- Input: list of N numbers stored in an array. Do not know the order of the numbers.
- Desired Output: the numbers sorted smallest to largest.

- Data structure: Max-Heap
- Algorithm: Heapsort.

Max-Heap

– Definition:

To be a binary max-heap, two conditions need to be satisfied.

1. It should be a complete binary tree (all levels, except the last level, must be full and all nodes in the last level need to be as far left as possible).
2. The value of a node should be greater than or equal to its children.

```
Ex:      91                    52                   91
        /  \                  /  \                 /  \
     77    46              77    46              77    46
    /  \      \           / \    /              / \    /
  69    3      11       69   3  11            69   3  11
  not a max-heap       not a max-heap          a max-heap
```

Heap Properties

- The height h of a heap with n nodes: $h = \Theta(\log n)$.

  Since a heap with height h will have the minimum and maximum of nodes as follows.

  Minimum of $n = 1 + 2 + 2^2 + \ldots + 2^{h-1} + 1 = 2^h$

  Maximum of $n = 1 + 2 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$

  From the above two equations, we can derive $h = \Theta(\log n)$.

Max-Heap in an Array

– Array representation for a max-heap:

Assume array index starts at 1. Let `heap-size[A]` stands for the number of elements in the heap stored in the array A.

That is, A[1...heap-size[A]] stores the heap and the root of the heap is stored in A[1].

The `parent-child` relationship between two nodes are represented by the following formulas.

Given a node at array index i, $\text{Parent}(i) = \lfloor i/2 \rfloor$

$$\text{Left}(i) = 2i$$

$$\text{Right}(i) = 2\text{i} + 1$$

The example max-heap in this page can be represented in an array as

91	77	46	69	3	11

Heapsort

```
Heapsort(A)
1. Build-Max-Heap(A)
2. for i <-- length[A] downto 2
3.      do exchange A[1] <--> A[i]
4.         heap-size--
5.         Max-Heapify(A, 1)
```

– Running time analysis of Heapsort(A):

Heapsort call Build-Max-Heap(A) once and call Max-Heapify(A) $n-1$ times.

Thus, the running time is $O(n \log n)$.

Maintain Heap Method

```
Max-Heapify(A, i) // heapification downward
   Pre-condition: Both the left and right subtrees of node i are max-heaps
                  and i is less than or equal to heap-size[A]
   Post-condition: The subtree rooted at node i is a max-heap
l <-- Left(i)     2i
r <-- Right(i)    2i + 1
largest <-- i
if l <= heap-size[A] and A[l] > A[i]
   then largest <-- l
if r <= heap-size[A] and A[r] > A[largest]
   then largest <-- r
if largest != i
   then exchange A[i] <--> A[largest]
        Max-Heapify(A, largest)
```

Maintain Heap Example

```
Ex:         38                                                      38
           /   \                                                   /   \
         29      22                                              65      22
        /  \    /  \   --> call Max-Heapify(A, 2) -->           /  \    /  \
      52    65 12   9                                         52    31 12   9
     / \    /                                                / \    /
   26   7  31                                              26   7  29
```

Build a Max-Heap

```
Build-Max-Heap(A)

1. heap-size <-- length[A]

2. for i <-- length[A]/2  // integer division

3.     do Max-Heapify(A, i)
```

```
Ex:      46                46                46                91
       /    \            /    \            /    \            /    \
     11      77  -->   11      91  -->   69      91  -->   69      77
    /  \    /         /  \    /         /  \    /         /  \    /
  69    3  91       69    3  77       11    3  77       11    3  46
```

Run time Max Heapify A[i]

Let n be the number of nodes in the subtree rooted at node i.

Step 1 to Step 9 take $O(1)$ time.

Step 10 is a subproblem to Max-Heapify node i's subtree (either left or right subtree).

Since the size of a subtree of node i is at most $2n/3$ (occurs when the last row of the tree is half full). Check the figure below.

```
         x
       /     \          There are k+1 nodes in the last level
     x          x       Thus, n = 1 + k + k + k + 1 = 3k+2
    / \        / \      This implies k = (n-2)/3 and
   / k \      / k \     The size of the left subtree is 2k+1 = (2n-1)/3
  /_____\    /_____\    Taking out the constant, it is about 2n/3
  xxxxxxxxx
```

O(9) + O(max-heapify (2n/3))

Runtime Max Heapify

$O(9) + O(\text{max-heapify } (2n/3))$

$O(9) + O(9) + O(\text{max-heapify } (4n/9))$

$O(9) + O(9) + O(9) + O(\text{max-heapify}(8n/27))$

So, how many times can we divide N by 2: $N = 2^h$, $h = \log(N)$.

So, run time for Max Heapify = $\log(N) * O(9) = O(\log N)$