CS321 Spring 2021

Lecture 3

Jan 20 2021

Admin

- A1 Due Sat Jan 23rd – 11:59PM
- Zoom recordings are up on blackboard.
- Slides will be cross-posted on blackboard and web site.
- Web: https://cs.boisestate.edu/~scutchin/cs321

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 Θ(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
```

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 $O(n \log n)$.

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
```

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
1. l <-- Left(i)     2i
2. r <-- Right(i)   2i + 1
3. largest <-- i
4. if l <= heap-size[A] and A[l] > A[i]
5.    then largest <-- l
6. if r <= heap-size[A] and A[r] > A[largest]
7.    then largest <-- r
8. if largest != i
9.    then exchange A[i] <--> A[largest]
10.        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
```

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 $h = \Theta(\log n)$.

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(max-heapify (2n/3))

O(9) + O(9) + O(max-heapify (4n/9))

O(9) + O(9) + O(9) + O(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(logN)