CS 561, Lecture 5

Jared Saia
University of New Mexico

Today's Outline

- Annihilator Wrap-up
- Loop Invariants
- Binary Heaps

Limitations

- Our method does not work on $T(n) = T(n-1) + \frac{1}{n}$ or $T(n) = T(n-1) + \lg n$
- The problem is that $\frac{1}{n}$ and $\lg n$ do not have annihilators.
- Our tool, as it stands, is limited.
- Key idea for strengthening it is *transformations*

Transformations Idea

- Consider the recurrence giving the run time of mergesort $T(n) = 2T(n/2) + kn$ (for some constant k), $T(1) = 1$
- How do we solve this?
- We have no technique for annihilating terms like $T(n/2)$
- However, we can *transform* the recurrence into one with which we can work

Transformation

- Let $n = 2^i$ and rewrite $T(n)$:
- $T(2^0) = 1$ and $T(2^i) = 2T(\frac{2^i}{2}) + k2^i = 2T(2^{i-1}) + k2^i$
- Now define a new sequence t as follows: $t(i) = T(2^i)$
- Then $t(0) = 1$, $t(i) = 2t(i-1) + k2^i$

Now Solve

- We've got a new recurrence: $t(0) = 1$, $t(i) = 2t(i-1) + k2^i$
- We can easily find the annihilator for this recurrence
- $(\mathbf{L}-2)$ annihilates the homogeneous part, $(\mathbf{L}-2)$ annihilates the non-homogeneous part, So $(\mathbf{L}-2)(\mathbf{L}-2)$ annihilates $t(i)$
- Thus $t(i) = (c_1 i + c_2)2^i$

Reverse Transformation

- We've got a solution for $t(i)$ and we want to transform this into a solution for $T(n)$
- Recall that $t(i) = T(2^i)$ and $2^i = n$

$$\begin{aligned} t(i) &= (c_1 i + c_2)2^i && (1) \\ T(2^i) &= (c_1 i + c_2)2^i && (2) \\ T(n) &= (c_1 \lg n + c_2)n && (3) \\ &= c_1 n \lg n + c_2 n && (4) \\ &= O(n \lg n) && (5) \end{aligned}$$

Success!

Let's recap what just happened:

- We could not find the annihilator of $T(n)$ so:
- We did a *transformation* to a recurrence we could solve, $t(i)$ (we let $n = 2^i$ and $t(i) = T(2^i)$)
- We found the annihilator for $t(i)$, and solved the recurrence for $t(i)$
- We *reverse transformed* the solution for $t(i)$ back to a solution for $T(n)$

Another Example

- Consider the recurrence $T(n) = 9T(\frac{n}{3}) + kn$, where $T(1) = 1$ and k is some constant
- Let $n = 3^i$ and rewrite $T(n)$:
- $T(3^0) = 1$ and $T(3^i) = 9T(3^{i-1}) + k3^i$
- Now define a sequence t as follows $t(i) = T(3^i)$
- Then $t(0) = 1$, $t(i) = 9t(i-1) + k3^i$

Now Solve

- $t(0) = 1$, $t(i) = 9t(i-1) + k3^i$
- This is annihilated by $(\mathbf{L} - 9)(\mathbf{L} - 3)$
- So $t(i)$ is of the form $t(i) = c_1 9^i + c_2 3^i$

Reverse Transformation

- $t(i) = c_1 9^i + c_2 3^i$
- Recall: $t(i) = T(3^i)$ and $3^i = n$

$$\begin{aligned} t(i) &= c_1 9^i + c_2 3^i \\ T(3^i) &= c_1 9^i + c_2 3^i \\ T(n) &= c_1 (3^i)^2 + c_2 3^i \\ &= c_1 n^2 + c_2 n \\ &= O(n^2) \end{aligned}$$

In Class Exercise

Consider the recurrence $T(n) = 2T(n/4) + kn$, where $T(1) = 1$, and k is some constant

- Q1: What is the transformed recurrence $t(i)$? How do we rewrite n and $T(n)$ to get this sequence?
- Q2: What is the annihilator of $t(i)$? What is the solution for the recurrence $t(i)$?
- Q3: What is the solution for $T(n)$? (i.e. do the reverse transformation)

A Final Example

Not always obvious what sort of transformation to do:

- Consider $T(n) = 2T(\sqrt{n}) + \log n$
- Let $n = 2^i$ and rewrite $T(n)$:
- $T(2^i) = 2T(2^{i/2}) + i$
- Define $t(i) = T(2^i)$:
- $t(i) = 2t(i/2) + i$

A Final Example

- This final recurrence is something we know how to solve!
- $t(i) = O(i \log i)$
- The reverse transform gives:

$$\begin{aligned} t(i) &= O(i \log i) && (6) \\ T(2^i) &= O(i \log i) && (7) \\ T(n) &= O(\log n \log \log n) && (8) \end{aligned}$$

Correctness of Algorithms

- The most important aspect of algorithms is their correctness
- An algorithm by definition *always* gives the right answer to the problem
- A procedure which doesn't always give the right answer is a *heuristic*
- All things being equal, we prefer an algorithm to a heuristic
- How do we prove an algorithm is really correct?

Loop Invariants

A useful tool for proving correctness is loop invariants. Three things must be shown about a loop invariant

- **Initialization:** Invariant is true before first iteration of loop
- **Maintenance:** If invariant is true before iteration i, it is also true before iteration $i + 1$ (for any i)
- **Termination:** When the loop terminates, the invariant gives a property which can be used to show the algorithm is correct

Example Loop Invariant

- We'll prove the correctness of a simple algorithm which solves the following interview question:
- *Find the middle of a linked list, while only going through the list once*
- The basic idea is to keep two pointers into the list, one of the pointers moves twice as fast as the other
- (Call the head of the list the 0-th elem, and the tail of the list the $(n-1)$-st element, assume that $n-1$ is an even number)

Example Algorithm

```
  GetMiddle (List l){
    pSlow = pFast = l;
    while ((pFast->next)&&(pFast->next->next)){
      pFast = pFast->next->next
      pSlow = pSlow->next
    }
    return pSlow
}
```

Example Loop Invariant

- *Invariant: At the start of the i-th iteration of the while loop, pSlow points to the i-th element in the list and pFast points to the $2i$-th element*
- **Initialization:** True when $i = 0$ since both pointers are at the head
- **Maintenance:** if pSlow, pFast are at positions i and $2i$ respectively before i-th iteration, they will be at positions $i+1$, $2(i+1)$ respectively before the $i+1$-st iteration
- **Termination:** When the loop terminates, pFast is at element $n-1$. Then by the loop invariant, pSlow is at element $(n-1)/2$. Thus pSlow points to the middle of the list

Challenge

- Figure out how to use a similar idea to determine if there is a loop in a linked list *without marking nodes!*

What is a Heap

- “A heap data structure is an array that can be viewed as a nearly complete binary tree”
- Each element of the array corresponds to a value stored at some node of the tree
- The tree is completely filled at all levels except for possibly the last which is filled from left to right

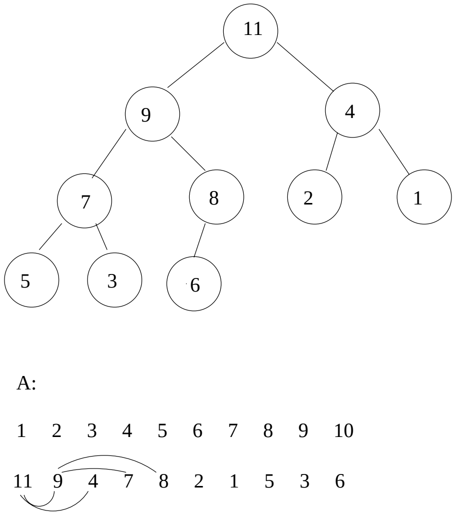
heap-size (A)

- An array A that represents a heap has two attributes
  - length (A) which is the number of elements in the array
  - heap-size (A) which is the number of elems in the heap stored within the array
- I.e. only the elements in A[1..heap-size (A)] are elements of the heap

Tree Structure

- A[1] is the root of the tree
- For all i, $1 < i <$ heap-size (A)
  - Parent (i) $= \lfloor i/2 \rfloor$
  - Left (i) $= 2i$
  - Right (i) $= 2i + 1$
- If Left (i) $>$ heap-size (A), there is no left child of i
- If Right (i) $>$ heap-size (A), there is no right child of i
- If Parent (i) < 0, there is no parent of i

Example



A:

1	2	3	4	5	6	7	8	9	10
11	9	4	7	8	2	1	5	3	6

Max-Heap Property

- For every node i other than the root, A[Parent (i)] $\geq$ A[i]

Max-Heap Property

- For every node i other than the root, A[Parent (i)] $\geq$ A[i]
- Parent is always at least as large as its children
- Largest element is at the root

(A Min-heap is organized the opposite way)

Height of Heap

- Height of a node in a heap is the number of edges in the longest simple downward path from the node to a leaf
- Height of a heap of n elements is $\Theta(\log n)$. Why?

Maintaining Heaps

- Q: How to maintain the heap property?
- A: *Max-Heapify* is given an array and an index i. Assumes that the binary trees rooted at $Left(i)$ and $Right(i)$ are max-heaps, but $A[i]$ may be smaller than its children.
- *Max-Heapify* ensures that after its call, the subtree rooted at i is a Max-Heap

Max-Heapify

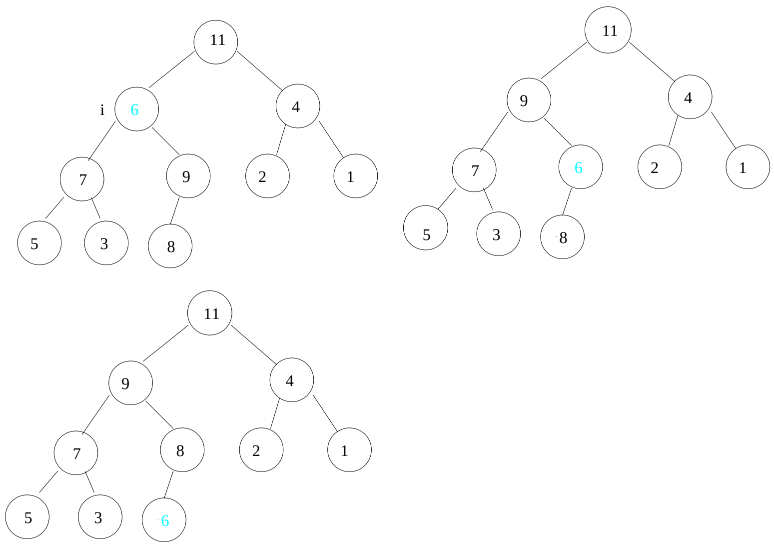
- Main idea of the Max-Heapify algorithm is that it percolates down the element that start at $A[i]$ to the point where the subtree rooted at i is a max-heap
- To do this, it repeatedly swaps $A[i]$ with its largest child until $A[i]$ is bigger than both its children
- For simplicity, the algorithm is described recursively.

Max-Heapify

```
Max-Heapify (A,i)

1. $l = Left(i)$
2. $r = Right(i)$
3. $largest = i$
4. if ($l \leq$ heap-size($A$) and $A[l] > A[i]$) then $largest = l$
5. if ($r \leq$ heap-size($A$) and $A[r] > A[largest]$) then $largest = r$
6. if $largest \neq i$ then
   (a) exchange A[i] and A[largest]
   (b) Max-Heapify (A,largest)
```

Example



Analysis

- Let $T(h)$ be the runtime of max-heapify on a subtree of height h
- Then $T(1) = \Theta(1)$, $T(h) = T(h-1) + 1$
- Solution to this recurrence is $T(h) = \Theta(h)$
- Thus if we let $T(n)$ be the runtime of max-heapify on a subtree of *size* n, $T(n) = O(\log n)$, since $\log n$ is the maximum height of heap of size n

Build-Max-Heap

- Q: How can we convert an arbitrary array into a max-heap?
- A: Use Max-Heapify in a bottom-up manner
- Note: The elements A[$\lfloor n/2 \rfloor + 1$],..,A[n] are all leaf nodes of the tree, so each is a 1 element heap to begin with

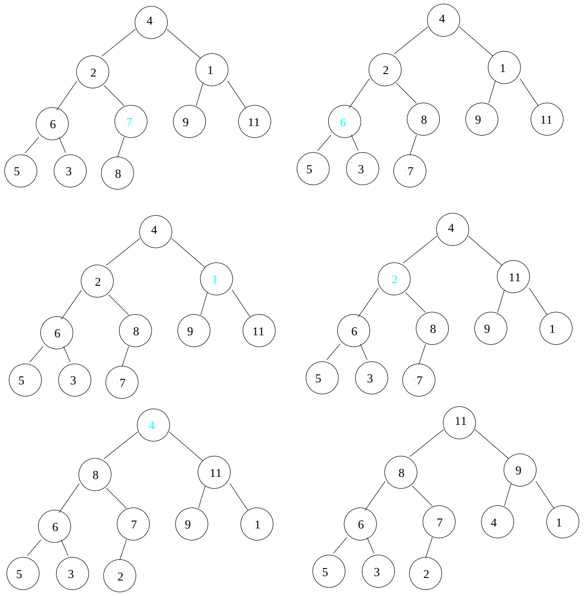
Build-Max-Heap

```
Build-Max-Heap (A)

1. heap-size (A) = length (A)
2. for ($i = \lfloor length(A)/2 \rfloor$;$i > 0$;$i--$)
   (a) do Max-Heapify (A,i)
```

Example

A = 4 2 1 6 7 9 11 5 3 8



Loop Invariant

- Loop Invariant: "At the start of each iteration of the for loop, each node $i+1, i+2, \ldots n$ is the root of a max-heap"

Correctness

- **Initialization:** $i = \lfloor n/2 \rfloor$ prior to first iteration. But each node $\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf so is the root of a trivial max-heap
- **Termination:** At termination, $i = 0$, so each node $1, \ldots, n$ is the root of a max-heap. In particular, node 1 is the root of a max heap.

Maintenance

- **Maintenance:** First note that if the nodes $i+1, \ldots n$ are the roots of max-heaps before the call to Max-Heapify (A,i), then they will be the roots of max-heaps after the call. Further note that the children of node i are numbered higher than i and thus by the loop invariant are both roots of max heaps. Thus after the call to Max-Heapify (A,i), the node i is the root of a max-heap. Hence, when we decrement i in the for loop, the loop invariant is established.

Time Analysis

(Naive) Analysis:

- Max-Heapify takes $O(\log n)$ time per call
- There are $O(n)$ calls to Max-Heapify
- Thus, the running time is $O(n \log n)$

Time Analysis

Better Analysis. Note that:

- An n element heap has height no more than $\log n$
- There are at most $n/2^h$ nodes of any height h (to see this, consider the min number of nodes in a heap of height h)
- Time required by Max-Heapify when called on a node of height h is $O(h)$.
- Thus total time is: $\sum_{h=0}^{\log n} \frac{n}{2^h} O(h)$

Analysis

$$\sum_{h=0}^{\log n} \frac{n}{2^h} O(h) = O\left(n \sum_{h=0}^{\log n} \frac{h}{2^h}\right) \quad (9)$$

$$= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \quad (10)$$

$$= O(n) \quad (11)$$

Analysis

The last step follows since for all $|x| < 1$,

$$\sum_{i=0}^{\infty} i x^i = \frac{x}{(1-x)^2} \quad (12)$$

Can get this equality by recalling that for all $|x| < 1$,

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x},$$

and taking the derivative of both sides!

Heap-Sort

```
Heap-Sort (A)

1. Build-Max-Heap (A)
2. for (i=length (A);$i > 1$;$i--$)
   (a) do exchange A[1] and A[i]
   (b) heap-size (A) = heap-size (A) - 1
   (c) Max-Heapify (A,1)
```

Analysis

- Build-Max-Heap takes $O(n)$, and each of the $O(n)$ calls to Max-Heapify take $O(\log n)$, so Heap-Sort takes $O(n \log n)$
- Correctness???