CS 561, Lecture 4

Jared Saia
University of New Mexico
Outline

• Loop Invariants
• Heaps
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.
- Loop Invariants are essentially proof by induction.
Loop Invariants

Three things must be shown about a loop invariant

- **Initialization**: Invariant is true before first iteration of loop (Base Case)
- **Maintenance**: If invariant is true before iteration \( i \), it is also true before iteration \( i + 1 \) (Inductive Step)
- **Termination**: When the loop terminates, the invariant gives a property which can be used to show the algorithm is correct (Wrapup)
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 \textit{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
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 elements in the heap stored within the array

I.e. only the elements in $A[1..\text{heap-size (A)}]$ are elements of the heap
Tree Structure

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

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

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, $A[\text{Parent}(i)] \geq A[i]$
Max-Heap Property

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

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

---

```
Example

11
4
217
35
11
4
21
9
7
35
11
4
21
9
7
3
8
5
6
6
6
9
8
8
```

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

1. heap-size (A) = length (A)
2. for (i = ⌊length(A)/2⌋; i > 0; i −−)
   (a) do Max-Heapify (A,i)
Example

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

- Diagram 1
- Diagram 2
- Diagram 3
- Diagram 4
- Diagram 5
- Diagram 6

23
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, ..., $n$ is the root of a max-heap. In particular, node 1 is the root of a max heap.
• **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.
(Naive) Analysis:

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

\[ \log n \sum_{h=0}^{n} \frac{n}{2^h} O(h) = O \left( n \sum_{h=0}^{\log n} \frac{h}{2^h} \right) \]  \hspace{1cm} (1)

\[ = O \left( n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \]  \hspace{1cm} (2)

\[ = O(n) \]  \hspace{1cm} (3)
The last step follows since for all $|x| < 1$,

$$\sum_{i=0}^{\infty} ix^i = \frac{x}{(1 - x)^2}$$  \hspace{1cm} (4)

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???