S.KIMMEL

Groups

Christian	Joonwoo	Kai	Henry
Zeke	Teal	Zach	Alderik
Bruce	Tenzin	Lillie	Chris
Scott	Caroline	Peter	Bryan
Dylan	Nadani	Annika	Jacqueline
Michael Ca	Ben	Jackson	Tommaso
Graham	Eliza	Nick	
Pierce	Alex	Will	
Angel	Michael Cz		

Reflection: Use group outside class. Look up master metho?

Programming Assignment!

Goals

- Create + analyze loop invariants
- Analyze QuickSort

S.KIMMEL

Loop Invariants: Prove loops are correct

```
setup
while (condition) {
      stuff
}
Great output
```

Induction tailored to loops

Parts of Loop Invariant Proof

1. State Invariant: thing(s) that is true before & after each loop iteration

2. Base Case: Show invariant is true before loop starts.

3. Maintenance: Show if invariant is true before an iteration, it is true after an iteration

4. Termination: argue loop ends. Given status of invariant after final loop, argue great output.

Input : Array A of integers of length n

Output: Array containing sorted elements of A

for $k = 1$ **to** $n - 1$ **do**
    for $j = n$ **to** $k + 1$ **do**
        if $A[j] < A[j - 1]$ **then**
            Swap $A[j]$ and $A[j - 1]$;
        end
    end
end
return A;

Algorithm 2: $\texttt{BubbleSort}(A)$

S. KIMMEL

Bubble Sort:

1. Inner Loop Invariant:

   - $A[j]$ is the smallest element of $A[j:n]$
   - The elements of A are same as input array.

   Base case: $j=n$, $A[n]$ is smallest of $A[n:n]$

   Maintenance: Since $A[j]$ is smallest of $A[j:n]$, it is smaller than all in $A[j-1:n]$ except perhaps $A[j-1]$, but we compare $A[j-1]$ and $A[j]$ and swap smaller to $A[j-1]$. This preserves elements of A and ensures after loop, $A[j-1]$ is smallest element of $A[j-1:n]$.

   Termination: The loop terminates at $j=k$, so we have
   - $A[k]$ is smallest element of $A[k:n]$
   - Elements of A preserved.

2. Outer loop invariant:

   - $A[1:k-1]$ is sorted
   - $A[1:k-1]$ contains the smallest $k-1$ elements of array
   - Elements of A same as input

   Base case: $k=0$, no elements in $A[1:k]$, A same as input

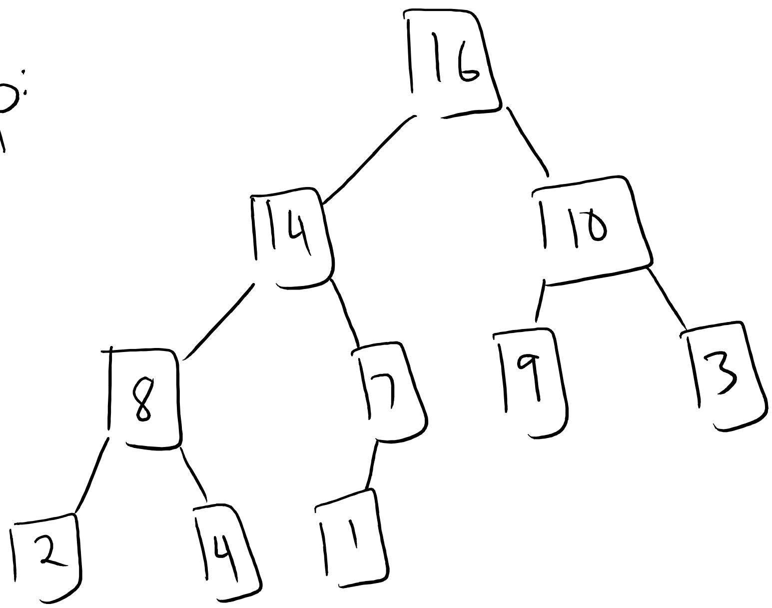
   Maintenance: Start of loop, $A[1:k-1]$ is sorted + contains $k-1$ smallest elements of A. Then inner loop moves smallest of remaining $A[k:n]$ to $A[k]$ while preserving elements, so now $A[1:k]$ contains smallest k elements of A, sorted.

   Termination: At $k=n$, so $A[1:n-1]$ is sorted smallest elements, but there is only one remaining element in $A[n]$, which must be the largest element. Elements are same as input, so output is sorted array.

S.KIMMEL

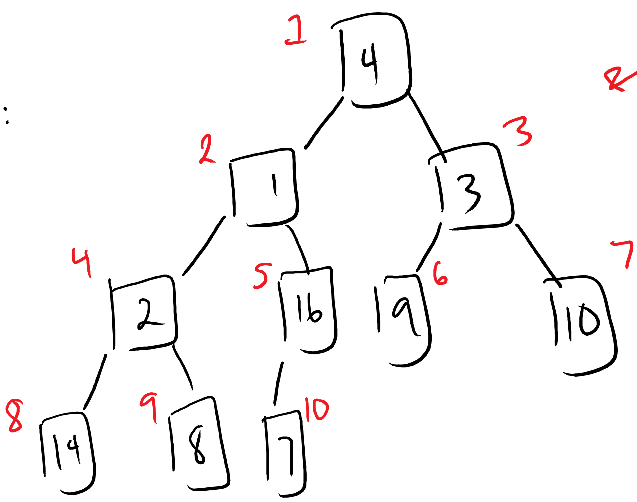
Loop Invariant for Heapify

Max Heap:



The key of each node is larger than all of its descendants

Before creating a heap:



In red: indeces of array where heap is stored:

1	2	3	4	5	...
4	1	3	2	16	...

Build-Max-Heap

for i = $\lfloor A.length/2 \rfloor$ to 1

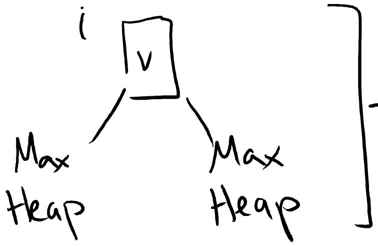
    Max-Heapify(A, i)

All indeces > $\lfloor A.length/2 \rfloor$ are leaves

S.KIMMEL

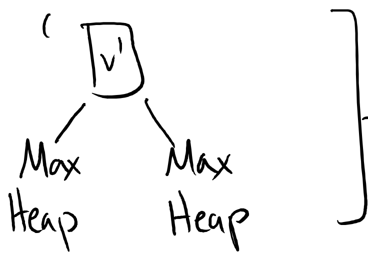
Max Heapify(A, i)

Input:



→ Full tree NOT max heap

Output:



→ Max heap

Prove Build-Max-Heap works correctly:

Invariant:

Initialization

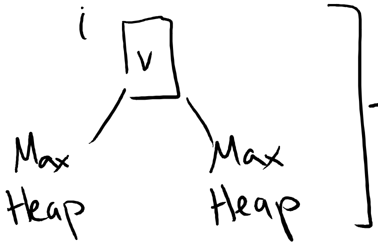
Maintenance

Termination

S. KIMMEL

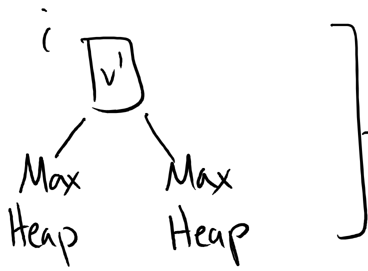
Max Heapify(A, i)

Input :



→ Full tree NOT max heap

Output:



→ Max heap

Prove Build-Max-Heap works correctly:

Invariant: Indeces $i+1, i+2, \ldots, n$ are roots of max heaps

Initialization

Indeces $\lfloor A.length/2 \rfloor, \ldots, n$ are leaves, and trees with one node are max heaps.

Maintenance

By our invariant, the children of i are roots of heaps, so Max-Heapify creates heap at i. Now $i, i+1, \cdots, n$ are roots of heaps.

Termination

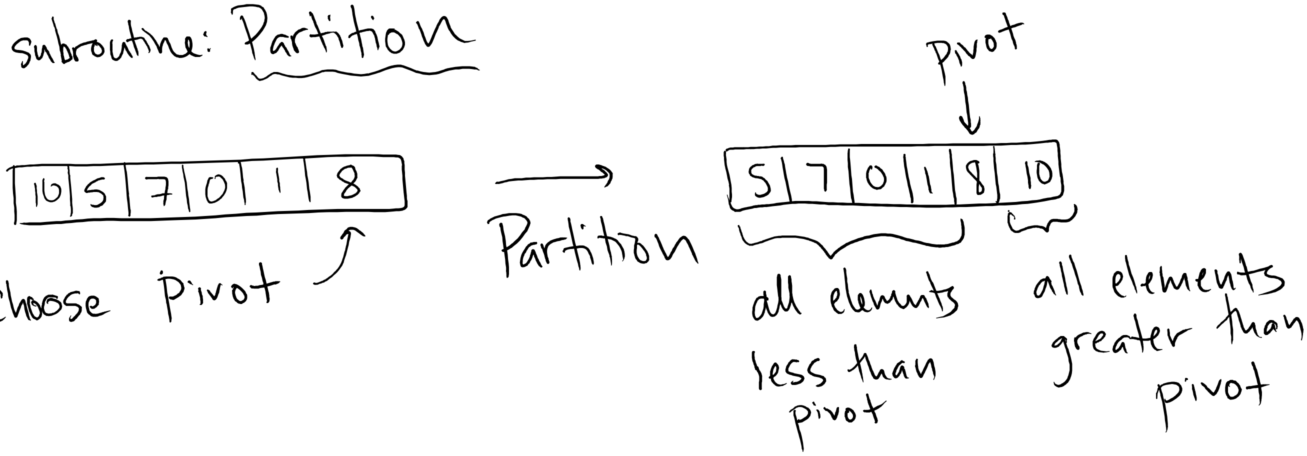
Loop ends at $i=1$, so indeces $1, 2, \cdots, n$ are roots of heaps but in particular, 1 is a root of a heap, so the whole tree is a heap.

S. KIMMEL

Randomization in Recursive Algorithms

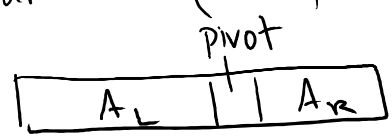
QuickSort Review

Key subroutine: Partition



Input: Array A of length n, no repeated elements
Output: Array with sorted elements

QuickSort(array A)

1. If $|A| = 1$: return A
2. pivot = ChoosePivot(A)
3. Partition(A, pivot)
4. 

5. QuickSort(A_L)
6. QuickSort(A_R)

Conquer

S.KIMMEL

Partition(A, p)

A:

10	5	12	0	1	8

p = value of pivot

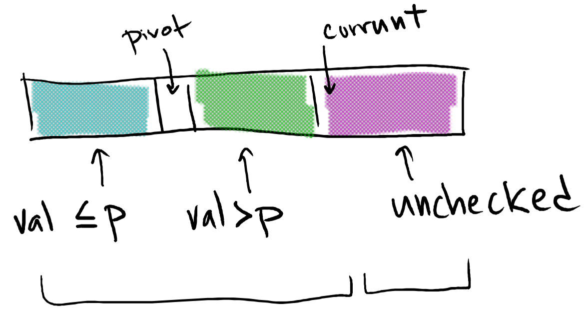
(0) If size = 1, return

(1) Move pivot to start

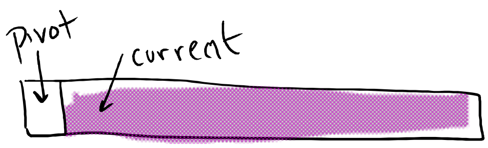
8	5	12	0	1	10

(2) Loop invariant:

Array looks like



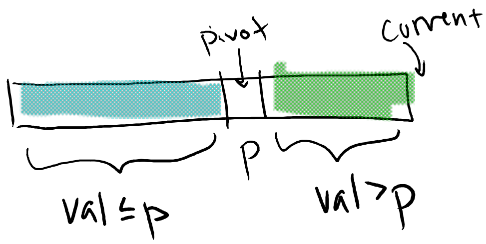
Base Case:



Each step:

- compares current to pivot
- does swaps to maintain invariant ✓
- increases current

Termination:



S. KIMMEL

Runtime of Quicksort is $O(\#\text{ of comparisons})$

Pf: Partition does most of the work, and runtime of Partition is $O(\#\text{ of comparisons.})$

Q: How many comparisons are done by Partition on input array of size n?

A: $O(\sqrt{n})$ B: $O(n)$ C: $O(n\log n)$ D: $O(n^2)$

Q: What is the runtime of QuickSort when the pivot is always chosen to be $\left(\frac{n}{2}\right)^{th}$ largest element of array?

A: $O(\sqrt{n})$ B: $O(n)$ C: $O(n\log n)$ D: $O(n^2)$