ADS Part 3 - Slides

1 Sorting

1.1 General lower bound for comparison based sorting

For any comparison based sorting algorithm $\mathcal{A}$ and any $n \in \mathbb{N}$ large enough there exists an input of length n that requires $\mathcal{A}$ to perform $\Omega(n \log n)$ comparisons

1.2 Bucket Sort

- How/why does it work
  - Puts elements with key i into the ith bucket, then empties one bucket after another
- What is the running time
  - Running time is $O(n + k)$ so if k is small the running time is $o(n \log n)$
- What are the assumptions?
- Do they "beat" the lower bound
  - Yes as bucket sort doesn't do any comparisons

1.3 Radix Sort

- How/why does it work
  - Like bucket sort, but looks at values "one level below", reduces number of buckets needed
- What is the running time
  - $\Theta(d \cdot n)$ or $\Theta(n \cdot \log K)$
- What are the assumptions
- Does it "beat" the lower bound
  - Yes, doesn't do comparisons

2 Binary Search

- What does it do
  - Finds an element in a sorted list
- How does it work
  - Looks at the median, if the element is smaller, look at the left sublist, if bigger the right, recursively call on sublists until the median is the element you want
- How long does it take
  - $O(\log n)$
- How is it analysed
  - $T(n) = T(n/2) + O(1) = O(\log n)$

3 Selection

3.1 QuickSelect

- What does it do
  - Finds an element in an unsorted list
- How does it work
  - Same as quicksort, choosing a pivot
  - Sublists discounted if they are known to not include the element to be found (either bigger/smaller than the pivot)
- How long does it take
  - In worst case $\Theta(n^2)$
  - Choose a random pivot and it will have the expectation of taking $O(n)$
- How is it analysed
  - $T(n) = T(n-1) + O(n) \Rightarrow T(n) = \Theta(n^2)$

3.2 Median-of-Medians

- What does it do
  - Search for an element in an unsorted list
- How does it work
  0. If length(A)$\leqslant$ 5 then sort and return ith smallest
  1. Divide n elements into $\lfloor n/5 \rfloor$ groups of 5 elements each, plus at most one group containing the remaining $n \mod 5$ elements
  2. Find median of each of the $\lceil n/5 \rceil$ groups by sorting each one, and then picking median from sorted group elements
  3. Call select recursively on set of $\lceil n/5 \rceil$ medians found above, giving median-of-medians, x
  4. Partition entire input around x. Let k be # of elements on low side plus one (simply count after partitioning)
     - x is the kth smallest element
     - there are n-k elements on high side of partition
  5. if i=k, return x. Otherwise use **select** recursively to find ith smallest element on low side if $i < k$, or (i-k)th smallest on high side if $i > k$
- How long does it take
  - $O(n)$
- How is is analysed
  - This is horrible

3.3 Comparison

- QuickSelect is singly recursive, so less work in each iteration than Median of Medians
- QuickSelect can have more iterations than Median of Medians
- QuickSelect used when bad behaviour is tolerable if undesirable
- Median of Medians is used when guaranteed good behaviour is needed

4 Binary Search Trees

- What are they
  - No node has more than two children
  - A tree
- Properties
  - All elements in left sub tree "smaller" than v
  - All elements in right sub tree "bigger" than v
- Standard Operations
  - Insertion
    - Insert at the root
    - Keep going down the tree until the correct location has been found
  - Search
    - Call at the root
    - If want smaller, go left
    - Otherwise, go right
  - Deletion
    - If a leaf, remove it
    - If it has one child remove and replace with the child
    - If two children
      - Find smallest node v that's bigger
      - Copy v's data into u
      - Delete v
- Problems

5 RedBlack Trees

- What's the point
  - Ensure that a BST is balanced
  - All BST operations are $O(h)$, where h is the height of the tree, so want to keep that as small as possible
- What are they
  - Every node is either red or black
  - The root is black
  - Every leaf (`NULL`) is black
  - Red nodes have black children
  - For all nodes, all paths from node to descendant leaves contain the same number of black nodes
- Key property
  - A red-black tree with n internal nodes has height at most $2\log(n+1)$

6 Heaps

- What are they
  - Trees typically assumed to be stored in a flat array
- Min-heap vs max-heap
  - MaxHeap - For all nodes v in the tree `v.parent.data >= v.data`
  - MinHeap - For all nodes v in the tree `v.parent.data <= v.data`
- Heap Property

- A[v.parent.index]¿= A[v.index]

- Representation tree vs array
  - The root is in A[1]
  - parent(i)=A[i/2] - integer division, rounds down
  - left(i)=A[2i]
  - right(i)=A[2i+1]

- Heapify (why? how? how long?)
  - Maintains heap property
  - Starting at the root, identify largest current node v and its children
  - Suppose largest element is in w
  - if $w \neq v$
    * Swap A[w] and A[v]
    * Recurse into w (contains now what root contained previously)
  - Runs linear in height of tree $O(\log n)$

- BuildHeap (why? how? how long?)
  - Initially build heap
  - Call heapify on all nodes
  - Runs in $O(n)$

- HeapSort (why? how? how long?)
  - Call buildheap on unsorted data
  - Repeatedly call HeapExtractMin until empty
  - Runs in time $O(n) + n \cdot O(\log n) = O(n \log n)$

7 Lower bounds

- What's the point
  - Can know when an algorithm optimally solves a problem

- Decision trees
  - What are they?
    * A full binary tree
    * Represents comparisons between elements performed by particular algorithm run on particular (size of) input
  - Proof for comparison based sorting
    * Sufficient to determine minimum height of a decision tree in which each permutation appears as a leaf
    * Consider decision tree of height h with ℓ leaves corresponding to a comparison sort on n elements
    * Each of the $n!$ permutations of input appears as some leaf: $\ell \geqslant n!$
    * Binary tree of height h has at most 2^h leaves $\ell \leqslant 2^h$
    * Together $n! \leqslant \ell \leqslant 2^h$

- Adversaries
  - What are they
    * A second algorithm intercepting access to input
    * Gives answers so that there's always a consistent input
    * Tries to make original algorithm delay a decision by dynamically constructing a bad input for it

    * Doesn't know what original algorithm will do in the future, must work for any original algorithm
- Proof for finding max
    * After $\leqslant n-2$ comparisons, $\geqslant 2$ elements never lost (a comp)
        · Adversary can make any of them max and be consistent
        · Not enough information for algorithm to make a decision
    * Hence algorithm needs to make at least n-1 comparisons