CS062
DATA STRUCTURES AND ADVANCED PROGRAMMING

16: Quicksort, Binary Trees and Heaps

Alexandra Papoutsaki
she/her/hers

Tom Yeh
he/him/his
Lecture 16: Quicksort, Binary Trees and Heaps

- Quicksort
QUICKSORT

Algorithm sketch:

- **Shuffle** the array.
- **Partition** so that, for some pivot $j$:
  - Entry $a[j]$ is in place.
  - There is no larger entry to the left of $j$.
  - No smaller entry to the right of $j$.
- **Sort** each subarray recursively.
private static void sort(Comparable[] a, int lo, int hi) {
    if (hi <= lo) return;
    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
Great algorithms are better than good ones

- Your laptop executes $10^8$ comparisons per second
- A supercomputer executes $10^{12}$ comparisons per second

<table>
<thead>
<tr>
<th>Computer</th>
<th>Insertion sort</th>
<th>Mergesort</th>
<th>Quicksort</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Thousand inputs</td>
<td>Million inputs</td>
<td>Billion inputs</td>
</tr>
<tr>
<td>Home</td>
<td>Instant</td>
<td>2 hours</td>
<td>300 years</td>
</tr>
<tr>
<td>Supercomputer</td>
<td>Instant</td>
<td>1 second</td>
<td>1 week</td>
</tr>
</tbody>
</table>
Quicksort analysis: best case

- Quicksort divides everything exactly in half.
- Similar to merge sort.
- Number of compares is $\sim n \log n$. 
Quicksort analysis: worst case

- Data are already sorted or we pick the smallest or largest key as pivot.
- Number of compares is \( \sim n^2 \) - quadratic!
- Extremely unlikely (less likely than the probably that your computer is struck by lightning) if we shuffle and our shuffling is not broken.
Things to remember about quick sort

- $O(n \log n)$ average, $O(n^2)$ worst, in practice faster than mergesort.

- 39% more compares than merge sort but in practice it is faster because it does not move data much.
  - Compare and increment pointer
  - Mergesort moves items into and out of aux array

- Random shuffle = probabilistic guarantee against worst case

- In-place sorting.
- Not stable.
Quicksort practical improvements

- Use insertion sort for small subarrays.
- Best choice of pivot is the median of a small sample.

- For years, Java used quicksort for collections of primitives and mergesort for collections of objects due to stability.

- Has moved to dual-pivot quick sort (Yaroslavskiy, Bentley, and Bloch, 2009) and timsort (Peters, 1993), respectively.
## Sorting: the story so far

<table>
<thead>
<tr>
<th>Which Sort</th>
<th>In place</th>
<th>Stable</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>X</td>
<td></td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$n$ exchanges</td>
</tr>
<tr>
<td>Insertion</td>
<td>X</td>
<td>X</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>Use for small arrays or partially ordered</td>
</tr>
<tr>
<td>Merge</td>
<td>X</td>
<td></td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>Guaranteed performance; stable</td>
</tr>
<tr>
<td>Quick</td>
<td>X</td>
<td></td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$n \log n$ probabilistic guarantee; fastest in practice</td>
</tr>
</tbody>
</table>
Sorting based on comparisons

- All sorting algorithms we have seen so far and we will see in this class are compare-based.
- No compare-based sorting algorithm can sort $n$ elements in less than $O(n \log n)$ time in the worst case.
- Proof and proper notation in CS140.
Readings:

- **Textbook:**
  - Chapter 2.3 (Pages 288-296)

- **Website:**

Practice Problems:

- 2.3.1-2.3.4
Basic data structures

- Arrays,
- Resizing arrays or arraylists,
- Linked Lists,
- Queues, and
- Stacks.

- Runtime and memory analysis for each one.
THE STORY SO FAR

Sorting

- Selection sort,
- Insertion sort,
- Mergesort, and
- Quicksort.

- Runtime (comparisons and exchanges), stability, in-place for each one.
- Comparators: How to sort a data structure with objects of any class.
- Iterators: How to traverse a data structure.
Lecture 16: Binary Trees and Heaps

- Binary Trees
- Tree traversals
- Binary Heaps
Trees in Computer Science

- Abstract data types that store elements hierarchically rather than linearly.

- Examples of hierarchical structures:
  - Organization charts for
    - Companies (CEO at the top followed by CFO, CMO, COO, CTO, etc).
    - Universities (Board of Trustees at the top, followed by President, then by VPs, etc).
  - Sitemaps (home page links to About, Products, etc. They link to other pages).
  - Computer file systems (user at top followed by Documents, Downloads, Music, etc. Each folder can hold more folders.).
Trees in Computer Science

- Hierarchical: Each element in a tree has a \textbf{single} parent (immediate ancestor) and zero or more children (immediate descendants).

\textbf{CS tree root and leaves}

\textbf{Real tree root and leaves}
Definition of a tree

- A tree $T$ is a set of nodes that store elements based on a parent-child relationship:
  - If $T$ is non-empty, it has a node called the root of $T$, that has no parent.
    - Here, the root is A.
  - Each node $v$, other than the root, has a unique parent node $u$. Every node with parent $u$ is a child of $u$.
    - E.g., E’s parent is C and F has two children, H and I.
Tree Terminology

- **Edge**: a pair of nodes s.t. one is the parent of the other, e.g., (K,C).

- **Parent** node is directly above **child** node, e.g., K is parent of C and N.

- **Sibling** nodes have same parent, e.g., A and F.

- K is **ancestor** of B.

- B is **descendant** of K.

- Node plus all **descendants** gives subtree.

- Nodes without descendants are called **leaves** or **external**. The rest are called **internal**.

- A set of trees is called a **forest**.
More Terminology

- **Simple path**: a series of distinct nodes s.t. there are edges between successive nodes, e.g., K-N-V-U.
- **Path length**: number of edges in path, e.g., path K-C-A has length 2.
- **Height of node**: length of longest path from the node to a leaf.
- **Height of tree**: length of longest path from the root to a leaf.
- **Degree of node**: number of its children.
- **Degree of tree (arity)**: max degree of any of its nodes.
- **Binary tree**: a tree with arity of 2.
Even More Terminology

- Level/depth of node defined recursively:
  - Root is at level 0.
  - Level of any other node is equal to level of parent + 1.
  - It is also known as the length of path from root or number of ancestors excluding itself.

- Height of node defined recursively:
  - If leaf, height is 0.
  - Else, height is max height of child + 1.
But wait there’s more!

- **Full (or proper)**: a binary tree whose every node has 0 or 2 children.
- **Complete**: a binary tree with minimal height. Any holes in tree would appear at last level to right, i.e., all nodes of last level are as left as possible.
Neither complete nor full

Complete but not full

Full but not complete

Complete and full

http://code.cloudkaksha.org/binary-tree/types-binary-tree
Practice Time: This tree is

- A: Full
- B: Complete
- C: Full and Complete
- D: Neither Full nor Complete
Answer

- A: Full
- **B: Complete**
- C: Full and Complete
- D: Neither Full nor Complete
Practice Time: This tree is

- A: Full
- B: Complete
- C: Full and Complete
- D: Neither Full nor Complete
Answer

- A: Full
- B: Complete
- C: Full and Complete
- D: Neither Full nor Complete
Counting in binary trees

- **Lemma**: if $T$ is a binary tree, then at level $k$, $T$ has $\leq 2^k$ nodes.
  - E.g., at level 2, at most 4 nodes (A, F, M, V)

- **Theorem**: If $T$ has height $h$, then # of nodes $n$ in $T$ satisfy:
  $$h + 1 \leq n \leq 2^{h+1} - 1.$$
  - Equivalently, if $T$ has $n$ nodes, then $\log(n + 1) - 1 \leq h \leq n - 1$.

- **Worst case**: When $h = n - 1$ or $O(n)$, the tree looks like a left or right-leaning “stick”.

- **Best case**: When a tree is as compact as possible (e.g., complete) it has $O(\log n)$ height.
public class BinaryTree<Item> {
    private Node root;

    /**
     * A node subclass which contains various recursive methods
     * @param <Item> The type of the contents of nodes
     */
    private class Node {
        private Item item;
        private Node left;
        private Node right;

        /**
         * Node constructor with subtrees
         * @param left   the left node child
         * @param right   the right node child
         * @param item   the item contained in the node
         */
        public Node(Node left, Node right, Item item) {
            this.left = left;
            this.right = right;
            this.item = item;
        }
    }
}
Lecture 16: Binary Trees and Heaps

- Binary Trees
- Tree traversals
- Binary Heaps
Pre-order traversal

- Preorder(Tree)
  - Mark root as visited
  - Preorder(Left Subtree)
  - Preorder(Right Subtree)
- K C A B F D H N M V U
In-order traversal

- Inorder(Tree)
  - Inorder(Left Subtree)
  - Mark root as visited
  - Inorder(Right Subtree)
- A B C D F H K M N U V
Post-order traversal

- Postorder(Tree)
  - Postorder(Left Subtree)
  - Postorder(Right Subtree)
- Mark root as visited
- B A D H F C M U V N K
Level-order traversal

- From left to right, mark nodes of level $i$ as visited before nodes in level $i + 1$. Start at level 0.

- K C N A F M V B D H U
Practice Time

- List the nodes in pre-order, in-order, post-order, and level order:
Answer

- Pre-order: 8, 5, 9, 7, 1, 12, 2, 4, 11, 3
- In-order: 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
- Post-order: 9, 1, 2, 12, 7, 5, 3, 11, 4, 8
- Level-order: 8, 5, 4, 9, 7, 11, 1, 12, 3, 2
Lecture 16: Binary Trees and Heaps

- Binary Trees
- Tree traversals
- Binary Heaps
Heap-ordered binary trees

- A binary tree is heap-ordered if the key in each node is larger than or equal to the keys in that node’s two children (if any).

- Equivalently, the key in each node of a heap-ordered binary tree is smaller than or equal to the key in that node’s parent (if any).

- No assumption of which child is smaller.

- Moving up from any node, we get a non-decreasing sequence of keys.

- Moving down from any node we get a non-increasing sequence of keys.
Heap-ordered binary trees

- The largest key in a heap-ordered binary tree is found at the root!
Binary heap representation

- We could use a linked representation but we would need three links for every node (one for parent, one for left subtree, one for right subtree).

- If we use complete binary trees, we can use instead an array.
  - Compact arrays vs explicit links means memory savings!
Binary heaps

- **Binary heap**: the array representation of a complete heap-ordered binary tree.

- Items are stored in an array such that each key is guaranteed to be larger (or equal to) than the keys at two other specific positions (children).

- Max-heap but there are min-heaps, too.
Array representation of heaps

- Nothing is placed at index 0.
- Root is placed at index 1.
- Rest of nodes are placed in level order.
- No unnecessary indices and no wasted space because it’s complete.
Reuniting immediate family members.

- For every node at index $k$, its parent is at index $\lfloor k/2 \rfloor$.
- Its two children are at indices $2k$ and $2k + 1$.
- We can travel up and down the heap by using this simple arithmetic on array indices.
Swim/promote/percolate up/bottom up reheapify

- Scenario: a key becomes larger than its parent therefore it violates the heap-ordered property.
- To eliminate the violation:
  - Exchange key in child with key in parent.
  - Repeat until heap order restored.
Swim/promote/percolate up

```java
private void swim(int k) {
    while (k > 1 && less(k/2, k)) {
        exch(k, k/2);
        k = k/2;
    }
}
```
Binary heap: insertion

- **Insert**: Add node at end in bottom level, then swim it up.
- **Cost**: At most $\log n + 1$ compares.

```java
public void insert(Key x) {
    pq[++n] = x;
    swim(n);
}
```
Practice Time

- Insert 47 in this binary heap.
Answer
Sink/demote/top down heapify

- Scenario: a key becomes smaller than one (or both) of its children’s keys.
- To eliminate the violation:
  - Exchange key in parent with key in \textbf{larger} child.
  - Repeat until heap order is restored.
Sink/demote/top down heapify

private void sink(int k) {
    while (2 * k <= n) {
        int j = 2 * k;
        if (j < n && less(j, j + 1))
            j++;
        if (!less(k, j))
            break;
        exch(k, j);
        k = j;
    }
}
Practice Time

- Sink 7 to its appropriate place in this binary heap.
Answer
Binary heap: return (and delete) the maximum

- **Delete max**: Exchange root with node at end. Return it and delete it. Sink the new root down.

- **Cost**: At most $2 \log n$ compares.

```java
public Key delMax() {
    Key max = pq[1];
    exch(1, n--);
    sink(1);
    pq[n+1] = null;
    return max;
}
```
Binary heap: delete and return maximum
Practice Time

- Delete max (and return it!)
Answer
Things to remember about runtime complexity of heaps

- Insertion is $O(\log n)$.
- Delete max is $O(\log n)$.
- Space efficiency is $O(n)$. 
2.4 Binary Heap Demo
Lecture 16: Binary Trees and Heaps

- Binary Trees
- Tree traversals
- Binary Heaps
Readings:

- Textbook:
  - Chapter 2.4 (Pages 308-327)

- Website:
  - Priority Queues: [https://algs4.cs.princeton.edu/24pq/](https://algs4.cs.princeton.edu/24pq/)

- Visualization:
  - Insert and ExtractMax: [https://visualgo.net/en/heap](https://visualgo.net/en/heap)

Practice Problems:

- Practice with traversals of trees and insertions and deletions in binary heaps