CS 5321: Advanced Algorithms – Sorting

Ali Ebnesasir
Department of Computer Science
Michigan Technological University

Acknowledgement

• Eric Torng
• Moon Jung Chung
• Charles Ofria

Outline

• Heapsort
• Quick review of basic sorting methods
• Lower bounds for comparison-based methods
• Non-comparison based sorting
Why don't CS profs ever stop talking about sorting?!

- Computers spend more time sorting than anything else, historically 25% on mainframes.
- Sorting is the best studied problem in computer science, with a variety of different algorithms known.
- Most of the interesting ideas we encounter in the course are taught in the context of sorting, such as divide-and-conquer, randomized algorithms, and lower bounds.

You should have seen most of the algorithms - we will concentrate on the analysis.

Example Problems

a. You are given a pile of thousands of telephone bills and thousands of checks sent in to pay the bills. Find out who did not pay.

b. You are given all the book checkout cards used in the campus library during the past year, each of which contains the name of the person who took out the book. Determine how many distinct people checked out at least one book.

Heaps and Heapsort

- Definition
- Operations and uses in heap construction
  - Insertion
  - Heapify
  - Delete Max
- Heapsort
Definition
A binary heap is defined to be a binary tree with a key in each node such that:
1: All leaves are on, at most, two adjacent levels.
2: All leaves on the lowest level occur to the left, and all levels except the lowest one are completely filled.
3: The key in root is greater than all its children, and the left and right subtrees are again binary heaps.
Conditions 1 and 2 specify shape of the tree, and condition 3 the labeling of the tree.

Heap Property
• Max-Heap property
  \[ A[parent(i)] \geq A[i] \]
• Min-Heap property
  \[ A[parent(i)] \leq A[i] \]

Which of these are heaps?
Partial Order Property

The ancestor relation in a heap defines a partial order on its elements, which means it is reflexive, anti-symmetric, and transitive.

Reflexive: \( x \) is an ancestor of itself.

Anti-symmetric: if \( x \) is an ancestor of \( y \) and \( y \) is an ancestor of \( x \), then \( x = y \).

Transitive: if \( x \) is an ancestor of \( y \) and \( y \) is an ancestor of \( z \), \( x \) is an ancestor of \( z \).

Partial orders can be used to model hierarchies with incomplete information or equal-valued elements.

Questions

• What are the minimum and maximum number of elements in a heap of height \( h \)?
  – 1 node heap has height 0
• What is the height of a heap with \( n \) elements?
• Where in a heap might the smallest node reside?

Array Implementation

• Root stored in index 1
• \( \text{Left}(x) = 2x \), and \( \text{Right}(x) = 2x + 1 \)
• \( \text{Parent}(x) = \text{floor}(x/2) \)

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>41</td>
<td>29</td>
<td>23</td>
<td>37</td>
<td>17</td>
<td>19</td>
<td>3</td>
<td>7</td>
<td>31</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Insertion Operation

- Place item to be inserted into leftmost open array slot
- If item is greater than parent, then swap and recurse
- Number of comparisons in the worst case?

```
1  2  3  4  5  6
43 41 29 23 37 33
```

Insert Algorithm

- Root has the maximum

```
Insert (X) {
    N = N+1;
    A[N] = X;
    i = N;
    while ((i > 1) && (A[Parent(i)] < X)) {
        i = floor(i/2);
    }
    A[i] = X;
}
```

Heap Construction By Insertion

- Suppose we did heap construction of an n element heap by sequentially inserting n items
- Let T(n) denote the number of comparisons needed in the worst-case to build a heap of n items
- Define a recurrence relation for T(n)
  - T(n) = T(n-1) + 1
- Solve your recurrence relation to derive the worst-case time to build a heap in this manner.
Heapify Operation

- Suppose you have a heap EXCEPT a specific value may violate the heap condition
- Fix by 3-way comparison working DOWN the heap
- WC # of comparisons?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>24</td>
<td>41</td>
<td>29</td>
<td>23</td>
<td>37</td>
<td>17</td>
<td>19</td>
</tr>
</tbody>
</table>

Heapify

- A: an array
- i: an index to an element that may violate the Max-Heap property

```
MaxHeapify(A, i) {
    if (2i > N) return; // the node “i” is a leaf node;
        j = 2i+1;
    else j = 2i;
    if (A[j] > A[i]) {
        swap (A[i], A[j]);
        MaxHeapify(A, j);
    }
}
```

DeleteMax Operation

- Copy root value to be returned
- Move rightmost entry to root
- Perform heapify to fix up heap
- WC running time?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>41</td>
<td>29</td>
<td>23</td>
<td>37</td>
<td>17</td>
<td>19</td>
<td>3</td>
<td>7</td>
<td>31</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>
DeleteMax

deleteMax (A) {
    ans = A[1];
    N = N-1;
    MaxHeapify (A,1);
    return (ans);
}

- Recursive deleteMin is much easier

Heap Construction By Heapify

- How can we construct a heap from n numbers by using the heapify operation?
- Example:
  - 5, 3, 17, 10, 84, 19, 6, 22, 9

Build Heap

- Approach 1: insert (A[i]), i=0,1,...,n.
  - complexity O(nlogn)
- Approach 2: bottom up construction
  - Key idea:
    Suppose that a tree with root T, and left subtree T1 and right subtree T2 is a heap.
    How to adjust them and make a heap?
  - Answer: Use percolate down to create heap-ordered tree
  - Code
    Buildheap () {
        for i=N/2 down to 1 {MaxHeapify (i); }
Analysis: Heap Construction By Heapify

- There is a direct analysis in the textbook. Here we discuss a recurrence relation analysis.
- Let $T(n)$ denote the number of comparisons needed in the worst-case to build a heap of $n$ items.
- Define a recurrence relation for $T(n)$:
  - $T(n) =$
  - $T(1) =$
- Solve your recurrence relation to derive the worst-case time to build a heap in this manner.

Heap Sort

- How can we use a heap and heap operations to solve the sorting problem?
- Do we need all three operations studied?
  - Insertion, Heapify, Delete Max
- What is the running time?

Sorting Algorithm Review I

- $\Theta(n^2)$ worst-case methods:
  - Insertion Sort
  - Selection Sort
  - Bubble Sort
- What is the idea behind each method?
- What are advantages/disadvantages of each method?
Sorting Algorithm Review II

- Faster methods
  - Merge Sort
  - Quicksort
  - Heapsort
- What is the idea behind merge sort?
- What are advantages/disadvantages of each method?

Quicksort Optimizations

- Quicksort is regarded as the fastest sort algorithm in most cases
- Some optimization possibilities
  - Randomized pivot selection: guarantees never to never have worst-case time due to bad data.
  - Median of three pivot selection: Can be slightly faster than randomization for somewhat sorted data.
  - Leave small sub-arrays for insertion sort: Insertion sort can be faster, in practice, for small values of $n$.
  - Do the smaller partition first: minimize runtime memory.

Possible reasons for not choosing quicksort

- Is the data already partially sorted?
- Do we know the distribution of the keys?
- Is the range of possible keys very small?
Lower Bounds

• Any comparison-based sorting program can be thought of as defining a decision tree of possible executions.

Example Decision Tree

• Consider the decision tree $T$ for any comparison-based algorithm. $T$ must have at least $n!$ leaves. Why?
• Given that there are $n!$ leaves, what must the height of the decision tree be?
• What does this imply about the running time of any comparison-based algorithm?

Analysis of Decision Tree

• Consider the decision tree $T$ for any comparison-based algorithm. $T$ must have at least $n!$ leaves. Why?
• Given that there are $n!$ leaves, what must the height of the decision tree be?
• What does this imply about the running time of any comparison-based algorithm?
Linear Time Sorting

• Algorithms exist for sorting n items in $\Theta(n)$ time IF we can make some assumptions about the input data

• These algorithms do not sort solely by comparisons, thus avoiding the $\Omega(n \log n)$ lower bound on comparison-based sorting algorithms
  – Counting sort
  – Radix Sort
  – Bucket Sort

Counting Sort

• Assumption: Keys to be sorted have a limited finite range, say $[0 .. k-1]$
• Method:
  – Count number of items with value exactly i
  – Compute number of items with value at most i
  – Use counts for placement of each item in final array
  – Full details in book
• Running time: $\Theta(n+k)$
• Key observation: Counting sort is stable

Counting Sort Algorithm

A: input arrays of length n
B: output array of length n
C: counting array of length k
1 .. k: range of input numbers

Counting-Sort(A, B, k)
{
  1) For i = 1 to k  C[i] = 0;
  2) For j = 1 to n C[A[j]] = C[A[j]] + 1;
  3) For i = 1 to k  C[i] = C[i] + C[i-1];
  4) For j = n downto 1  B[C[A[j]]] = A[j];
                     C[A[j]] = C[A[j]] – 1;
}

Radix Sort

- Assumption: Keys to be sorted have $d$ digits
- Method:
  - Use counting sort (or any stable sort) to sort numbers starting with least significant digit and ending with most significant digit
- Running time: $\Theta(d(n+k))$ where $k$ is the number of possible values for each digit, i.e., radix

---

Radix Sort - Example

| 110011 | 51 | 000110 | 6 | 110000 | 48 |
| 101001 | 41 | 110000 | 48 | 101001 | 41 |
| 010011 | 19 | 110011 | 51 | 011001 | 25 |
| 000110 | 6 | 101001 | 41 | 000110 | 6 |
| 110000 | 48 | 010011 | 19 | 110011 | 51 |
| 011001 | 25 | 011001 | 25 | 010011 | 19 |
| 010111 | 23 | 010111 | 23 | 010111 | 23 |

How do you describe stability property on this example?

---

Bucket Sort

- Assumption: Keys to be sorted are uniformly distributed over a known range (say 1 to $m$)
- Method:
  - Set up $n$ buckets where each bucket is responsible for an equal portion of the range
  - Sort items in buckets using insertion sort
  - Concatenate sorted lists of items from buckets to get final sorted order
Bucket Sort Analysis

- Key analysis: Let $X$ be a random variable for the \# of comparisons required by insertion sort on items in each bucket
  - Let $n_i$ be the number of items in bucket $i$
  - $E[X] = \sum_{i=1}^{n} O(E[n_i^2])$
  - $E[n_i^2] = 2 - 1/n$ derivation in book
  - Intuition: What is $E[n_i]$?
- Question: Why use insertion sort rather than quicksort?

Bucket Sort Gone Wrong

We can use bucketsort effectively whenever we understand the distribution of the data.
However, bad things happen when we assume the wrong distribution.
Make sure you understand your data, or use a good worst-case or randomized algorithm!