CSE 332 Winter 2024
Lecture 13: Hashing and Sorting

Nathan Brunelle

http://www.cs.uw.edu/332
Collision Resolution: Linear Probing

• When there’s a collision, use the next open space in the table
Linear Probing: Insert Procedure

- To insert $k, v$
  - Calculate $i = h(k) \% \text{arrsize}$
  - If $\text{table}[i]$ is occupied then try $(i + 1)\%\text{arrsize}$
  - If that is occupied try $(i + 2)\%\text{arrsize}$
  - If that is occupied try $(i + 3)\%\text{arrsize}$
  - ...
Linear Probing: Find

• \( i = h(k) \% \text{arrsize} \)
  • If \( i \) has the key or it’s empty, then we’re done
  • Otherwise:
    • Check \((i + 1) \% \text{arrsize}\) if it’s there, done else
    • Check \((i + 2) \% \text{arrsize}\) if it’s there, done else
    • Check \((i + 3) \% \text{arrsize}\)
    • ...
    • Until we hit an empty cell
Linear Probing: Find

• To find key $k$
  • Calculate $i = h(k) \% arrsize$
  • If $table[i]$ is occupied and does not contain $k$ then look at $(i + 1) \% arrsize$
  • If that is occupied and does not contain $k$ then look at $(i + 2) \% arrsize$
  • If that is occupied and does not contain $k$ then look at $(i + 3) \% arrsize$
  • Repeat until you either find $k$ or else you reach an empty cell in the table
Linear Probing: Delete

• Problem: don’t want to leave an empty space when deleting
• Option 1: when we delete, move the “last thing” with a matching hash to that location
• Option 2: “tombstone” deletion. When deleting something, leave a special marker to indicate something used to be there
Linear Probing: Delete

• Option 1: Find the last thing with a matching hash, move that into the spot you deleted from

• Option 2: Called “tombstone” deletion. Leave a special object that indicates an object was deleted from there
  • The tombstone does not act as an open space when finding (so keep looking after its reached)
  • When inserting you can replace a tombstone with a new item
Downsides of Linear Probing

• What happens when $\lambda$ approaches 1?
  • Longer and longer clusters of items
  • Runnings times get longer and longer
Quadratic Probing: Insert Procedure

- To insert \( k, v \):
  - Calculate \( i = h(k) \mod arrsize \)
  - If \( table[i] \) is occupied then try \((i + 1^2) \mod arrsize\)
  - If that is occupied try \((i + 2^2) \mod arrsize\)
  - If that is occupied try \((i + 3^2) \mod arrsize\)
  - If that is occupied try \((i + 4^2) \mod arrsize\)
  - ...

[Diagram of array with some indices marked as occupied]
Quadratic Probing: Example

Insert:
- Insert:
  - 76 → \(10 \times 7 - 6\)
  - 40 → \(2 \times 7 - 5\)
  - 48 → \(3 \times 7 - 6\)
  - 5 → \(4 \times 1 - 2\) → \(5 + 2^2\)
  - 55 → \(5 \times 11 - 2\) → \(6 + 2^2\)
  - 47 → \(6 \times 7 - 2\) → \(5 \times 7^2\) → \(5 + 4^2\)
Using Quadratic Probing

• If you probe \textit{tablesize} times, you start repeating the same indices
• If \textit{tablesize} is prime and \( \lambda < \frac{1}{2} \) then you’re guaranteed to find an open spot in at most \( \textit{tablesize}/2 \) probes

• Helps with the clustering problem of linear probing, but does not help if many things hash to the same value
Double Hashing: Insert Procedure

• Given $h$ and $g$ are both good hash functions

• To insert $k, v$
  • Calculate $i = h(k) \mod \text{size}$
  • If $table[i]$ is occupied then try $(i + g(k)) \mod \text{size}$
  • If that is occupied try $(i + 2 \cdot g(k)) \mod \text{size}$
  • If that is occupied try $(i + 3 \cdot g(k)) \mod \text{size}$
  • If that is occupied try $(i + 4 \cdot g(k)) \mod \text{size}$
  • ...
Rehashing

- If your load factor $\lambda$ gets too large, copy everything over to a larger hash table
  - To do this: make a new array with a new hash function (maybe just a new modulus)
  - Re-insert all items into the new hash table with the new hash function
  - New hash table should be “roughly” double the size (but probably still want it to be prime)

- General Guideline:
  - Separate Chaining: rehash when $\lambda = 2$
  - Open Addressing: rehash when $\lambda = \frac{1}{2}$
Sorting

- Rearrangement of items into some defined sequence
  - Usually: reordering a list from smallest to largest according to some metric

- Why sort things?
  - Makes other things faster
  - Binary search
  - Min/max/i.e. etc.
More Formal Definition

• Input:
  • An array $A$ of items
  • A comparison function for these items
    • Given two items $x$ and $y$, we can determine whether $x < y$, $x > y$, or $x = y$

• Output:
  • A permutation of $A$ such that if $i \leq j$ then $A[i] \leq A[j]$
  • Permutation: a sequence of the same items but perhaps in a different order
Sorting “Landscape”

• There is no singular best algorithm for sorting
• Some are faster, some are slower
• Some use more memory, some use less
• Some are super extra fast if your data matches particular assumptions
• Some have other special properties that make them valuable
• No sorting algorithm can have only all the “best” attributes
“Moving Day” Sorting Algorithm

$\mathcal{O}(n^2)$
Selection Sort

• **Idea:** Find the *next smallest* element, swap it into the *next index* in the array

```plaintext
1 2 3 4 5 6 10 8 7 9 12 11
```

Already In Position

```plaintext
1 2 3 4 5 6 7 8 10 9 12 11
```

Already In Position
Selection Sort

- Swap the thing at index 0 with the smallest thing in the array
- Swap the thing at index 1 with the smallest thing after index 0
- ...
- Swap the thing at index $i$ with the smallest thing after index $i - 1$

```java
for (i=0; i<a.length; i++){
    smallest = i;
    for (j=i; j<a.length; j++){
        if (a[j]<a[smallest]) { smallest=j; }
    }
    temp = a[i];
    a[i] = a[smallest];
    a[smallest] = a[i];
}
```

Running Time:
- Worst Case: $\Theta(\cdot)$
- Best Case: $\Theta(\cdot)$
Insertion Sort

• Idea: Maintain a sorted list prefix, extend that prefix by “inserting” the next element
Insertion Sort

• If the items at index 0 and 1 are out of order, swap them
• Keep swapping the item at index 2 with the thing to its left as long as the left thing is larger
• ...
• Keep swapping the item at index $i$ with the thing to its left as long as the left thing is larger

for (i=1; i<a.length; i++){
    prev = i-1;
    while(a[i] < a[prev] && prev > -1){
        temp = a[i];
        a[i] = a[prev];
        a[prev] = a[i];
        i--;
        prev--;
    }
}

Running Time:
Worst Case: $\Theta(\cdot)$
Best Case: $\Theta(\cdot)$
Aside: Bubble Sort – we won’t cover it

"the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems” –Donald Knuth, The Art of Computer Programming
Heap Sort

- **Idea:** Build a maxHeap, repeatedly delete the max element from the heap to build sorted list Right-to-Left

<table>
<thead>
<tr>
<th>10</th>
<th>9</th>
<th>6</th>
<th>8</th>
<th>7</th>
<th>5</th>
<th>2</th>
<th>4</th>
<th>1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

**Max Heap Property:** Each node is larger than its children
Heap Sort

- **Remove the Max element (i.e. the root) from the Heap:** replace with last element, call `percolateDown(root)`

```
3 9 6 8 7 5 2 4 1
0 1 2 3 4 5 6 7 8 9
```

```
3

9

8 1

4 3 1

7 4

6

5

2

```

**Percolate Down(node):** if `node` satisfies heap property, done. Else swap with largest child and repeat on that subtree
Heap Sort

- Remove the Max element (i.e. the root) from the Heap: replace with last element, call percolateDown(root)

Percolate Down(node): if node satisfies heap property, done. Else swap with largest child and repeat on that subtree.
Heap Sort

- Remove the Max element (i.e. the root) from the Heap: replace with last element, call percolateDown(root)

**Percolate Down(node):** if node satisfies heap property, done. Else swap with largest child and repeat on that subtree
Heap Sort

- Remove the Max element (i.e. the root) from the Heap: replace with last element, call percolateDown(root)

Percolate Down(node): if node satisfies heap property, done. Else swap with largest child and repeat on that subtree
Heap Sort

- Build a heap
- Call deleteMax
- Put that at the end of the array

\[
\text{myHeap} = \text{buildHeap}(a);
\text{for} \ (\text{int} \ i = a.\text{length}-1; \ i>=0; \ i-\text{--})\{
    \text{item} = \text{myHeap}.\text{deleteMax}();
    a[i] = \text{item};
\}
\]

Running Time:
- Worst Case: $\Theta(\cdot)$
- Best Case: $\Theta(\cdot)$
“In Place” Sorting Algorithm

• A sorting algorithm which requires no extra data structures
• Idea: It sorts items just by swapping things in the same array given
• Definition: it only uses $\Theta(1)$ extra space

• Selection sort: In Place!
• Insertion sort: In Place!
• Heap sort: Not In Place!
  • But we can fix that!
In Place Heap Sort

- **Idea:** When “removing” an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter.
Heap Sort

- **Idea**: When “removing” an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter.
Heap Sort

**Idea:** When “removing” an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter
Heap Sort

• **Idea:** When “removing” an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter
Heap Sort

• Idea: When “removing” an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter.
Heap Sort

- **Idea**: When “removing” an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter.
Heap Sort

- Idea: When “removing” an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter.

```
0 1 2 3 4 5 6 7 8 9
```

```
3 8 6 4 7 5 2 8 9 10
```
In Place Heap Sort

• Build a heap using the same array (Floyd’s build heap algorithm works)
• Call deleteMax
• Put that at the end of the array

```
buildHeap(a);
for (int i = a.length-1; i>=0; i--){
    temp=a[i]
    a[i] = a[0];
    a[0] = temp;
    percolateDown(0);
}
```

Running Time:
Worst Case: $\Theta(\cdot)$
Best Case: $\Theta(\cdot)$
Floyd’s buildHeap method

- Working towards the root, one row at a time, percolate down

```java
buildHeap()
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```