Next topic: Hash Tables

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Time to insert</th>
<th>Time to find</th>
<th>Time to delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Array</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Unsorted Linked List</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted Array</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>$\Theta(\text{height})$</td>
<td>$\Theta(\text{height})$</td>
<td>$\Theta(\text{height})$</td>
</tr>
<tr>
<td>AVL Tree</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>Hash Table (Worst case)</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Hash Table (Expected and amortized)</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>
Dictionary (Map) ADT

• Contents:
  • Sets of key+value pairs
    • Keys must be comparable

• Operations:
  • insert(key, value)
    • Adds the (key,value) pair into the dictionary
    • If the key already has a value, overwrite the old value
      • Consequence: Keys cannot be repeated
  • find(key)
    • Returns the value associated with the given key
  • delete(key)
    • Remove the key (and its associated value)
Hash Tables

- Idea:
  - Have a small array to store information
  - Use a **hash function** to convert the key into an index
    - Hash function should “scatter” the keys, behave as if it randomly assigned keys to indices
  - Store key at the index given by the hash function
  - Do something if two keys map to the same place (should be very rare)
    - Collision resolution
Collision Resolution

• A Collision occurs when we want to insert something into an already-occupied position in the hash table

• 2 main strategies:
  • Separate Chaining
    • Use a secondary data structure to contain the items
      • E.g. each index in the hash table is itself a linked list
  • Open Addressing
    • Use a different spot in the table instead
      • Linear Probing
      • Quadratic Probing
      • Double Hashing
Rehashing

• If your load factor $\lambda$ gets too large, copy everything over to a larger hash table
  • To do this: make a new, larger array
  • Re-insert all items into the new hash table by reapplying the hash function
    • We need to reapply the hash function because items should map to a different index
    • New array should be “roughly” double the length (but probably still want it to be prime)

• What does “too large” mean?
  • For separate chaining, typically we want $\lambda < 2$
  • For open addressing, typically we want $\lambda < \frac{1}{2}$
Linear Probing: Insert Procedure

• To insert $k, v$
  • Calculate $i = h(k) \% \text{length}$
  • If $\text{table}[i]$ is occupied then try $(i + 1)\% \text{length}$
  • If that is occupied try $(i + 2)\% \text{length}$
  • If that is occupied try $(i + 3)\% \text{length}$
  • ...$
  • h(k) = k\%10
Linear Probing: Find

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

• To delete key $k$, where $h(k) = i$
  • Assume it is present
• Beginning at index $i$, probe until we find $k$ (call this location index $j$)
• Mark $j$ as empty (e.g. null), then continue probing while doing the following until you find another empty index
  • If you come across a key which hashes to a value $\leq j$ then move that item to index $j$ and update $j$. 

0 1 2 3 4 5 6 7 8 9
Linear Probing: Delete

• Option 1: Fill in with items that hashed to before the empty slot
• Option 2: “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?
  • Get longer and longer contiguous blocks
  • A collision is guaranteed to grow a block
    • Larger blocks experience more collisions
    • Feedback loop!

• What happens when $\lambda$ exceeds 1?
  • Impossible!
  • You can’t insert more stuff
Quadratic Probing: Insert Procedure

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

• Use $i = k \% 7$
• Insert:
  • 76 -> 6; insert at index 6
  • 40 -> 5; insert at index 5
  • 48 -> 6; 6+1^2; insert at index 0
  • 5 -> 5; 5+2^2; insert at index 2
  • 55 -> 6; 6+2^2; insert at index 3
  • 47 -> 5; 5+5^2

<table>
<thead>
<tr>
<th></th>
<th>48</th>
<th>5</th>
<th>55</th>
<th>40</th>
<th>76</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
Using Quadratic Probing

• If you probe \textit{tablesiz}e times, you start repeating the same indices

• If \textit{tablesiz}e is prime and $\lambda < \frac{1}{2}$ then you’re guaranteed to find an open spot in at most $\textit{tablesiz}e/2$ 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 \( (i + 4 \cdot g(k)) \mod \text{size} \)
  • ...
Double Hashing: Insert Procedure

- Given \( h \) and \( g \) are both good hash functions
  - \( h(k) = k \mod 10 \)
  - \( g(k) = k \mod 7 \)
  - \( g(k) \mod \text{len} = 0 \)
- If \( g(k) \mod \text{len} == 0 \) then add 1 to \( g(k) \)

<table>
<thead>
<tr>
<th>0</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>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>23</td>
<td>33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hash tables

• Can use hash functions with “good” properties to make it so a small array has similar performance to a huge one
  • The function should randomly scatter keys
• Because the array is small, it’s possible for collisions to occur
  • Sep. chaining
    • Insert k,v pairs into another DS
  • Open addressing
    • Insert k,v pairs directly into indices
    • Use probing strategy when collisions occur
      • Linear
      • Quad
      • Double hashing
• Need to watch our load factor, and rehash when it gets to big
  • SC: <2
  • OA: <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?
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
\[
\sum_{i=1}^{n} i = \frac{n(n+1)}{2}
\]

"Moving Day" Sorting Algorithm
Selection Sort

• **Idea**: Find the next smallest element, swap it into the next index in the array
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: $\Theta(n^2)$
- Best Case: $\Theta(n^2)$
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] = temp;
        i--;
        prev--;
    }
}

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

<table>
<thead>
<tr>
<th>10</th>
<th>77</th>
<th>5</th>
<th>15</th>
<th>2</th>
<th>22</th>
<th>64</th>
<th>41</th>
<th>18</th>
<th>19</th>
<th>30</th>
<th>21</th>
<th>3</th>
<th>24</th>
<th>23</th>
<th>33</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>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>
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

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)

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

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

```java
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);
    }
}
```