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 $i = h(k) \mod length$
  • If $table[i]$ is occupied then try $(i + 1)\mod length$
  • If that is occupied try $(i + 2)\mod length$
  • If that is occupied try $(i + 3)\mod length$
  • ...
  • $h(k) = k\mod 10$

$$
\begin{array}{|c|c|c|c|c|c|c|c|c|c|}
\hline
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
\end{array}
$$
Linear Probing: Find

- To find key $k$
  - Calculate $i = h(k) \% \text{length}$
  - If $table[i]$ 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 $j$. 
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 \( i = h(k) \mod \text{size} \)
  • If \( table[i] \) is occupied then try \( (i + 1^2) \mod \text{size} \)
  • If that is occupied try \( (i + 2^2) \mod \text{size} \)
  • If that is occupied try \( (i + 3^2) \mod \text{size} \)
  • If that is occupied try \( (i + 4^2) \mod \text{size} \)
  • ...
Quadratic Probing: Example

- Insert:
  - 76
  - 40
  - 48
  - 5
  - 55
  - 47
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 \( \text{table}[i] \) 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 \( (i + 4 \cdot g(k)) \mod \text{size} \)
  • ...

\[ \begin{array}{ccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array} \]
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
“Moving Day” Sorting Algorithm
Selection Sort

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

Already In Position

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

Already In Position

1 2 3 4 5 6 7 8 10 9 12 11
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: $\Theta(\cdot)$

<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>
Insertion Sort

- **Idea**: Maintain a sorted list prefix, extend that prefix by “inserting” the next element.

![Sorted Prefix]

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

Sorted Prefix
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

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

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

<table>
<thead>
<tr>
<th>Max Heap Property: Each node is larger than its children</th>
</tr>
</thead>
</table>

```
10
9
6
8
7
5
2
4
1
3
```

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

```
10
  
9
  
8
  1
  
7
  4
  
4
  
3
  
5
  5
  2
  
2
  
6
```
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

```java
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

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