supB
Heaps and Dictionaries
What's a heap?

- A complete binary tree that is either
  - empty
  - root $\geq$ than both children
  - subtrees are heaps
implementation
Heap<E>

+isEmpty(): bool
+nodeCount(): int
+height(): int
+top(): E
+push(e: const E&): void
+pop(): void
+makeEmpty(): void
Insert
heapInsert(in HeapItemType newItem) {
    items[size] = newItem
    child = size
    parent = (child - 1) / 2

    while (parent >= 0 and
    items[child] > items[parent]) {
        swap items[child] and items[parent]
        child = parent
        parent = (child - 1) / 2
    } // end while
    increment size

} // end heapInsert()
Delete
Disjoint heaps
Semi heap
heapDelete() {
    items[0] = items[size - 1]
    size--
    heapRebuild(items, 0, size)
} // end heapRebuild()
heapRebuild(items: ArrayType, p: int, n: int) {
    if (p is not a leaf) {
        // determine largest child
        max = 2 * p + 1
        if (p has a right child) {
            max = (items[max + 1] > items[max])?
                    max + 1 : max;
        }
        // swap parent with child if need be
        if (items[p] < items[max]) {
            swap items[p] and items[max]
            heapRebuild(items, max, n)
        }
    }
    // end if
    // end heapRebuild()
Uses ...
<table>
<thead>
<tr>
<th>PriorityQueue&lt;E&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>items: E[]</td>
</tr>
<tr>
<td>isEmpty(): bool</td>
</tr>
<tr>
<td>push(e: const E&amp;): void</td>
</tr>
<tr>
<td>pop(): void</td>
</tr>
<tr>
<td>front(): E</td>
</tr>
</tbody>
</table>
• Heap:
  • Insert: $O(\log n)$, Delete: $O(\log n)$

• Sorted list:
  • Insert: $O(n)$, Delete: $O(n)$

• Bst
  • Insert: $O(\log n)$, Delete: $O(\log n)$
HeapSort

(1) Transform to a Heap
(2) Swap root with last element
(3) Rebuilt the heap
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The diagram shows a tree with the root node labeled 7. The nodes labeled 5 and 6 are connected to the root. The table on the right lists the values associated with each node:

- [0]: 7
- [1]: 5
- [2]: 6
- [3]: 8
- [4]: 9
- [5]: 10
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>5</td>
</tr>
<tr>
<td>[1]</td>
<td>6</td>
</tr>
<tr>
<td>[2]</td>
<td>7</td>
</tr>
<tr>
<td>[3]</td>
<td>8</td>
</tr>
<tr>
<td>[4]</td>
<td>9</td>
</tr>
<tr>
<td>[5]</td>
<td>10</td>
</tr>
</tbody>
</table>
heapSort(arr:ArrayType, n:int) {
   // build initial heap
   for (index = n-1 down to 0) {
      heapRebuilt(arr, index, n)
   }// end for

   last = n - 1
   for (step = 1 through n) {
      swap arr[0] and arr[last]
      decrement last

      heapRebuilt(arr, 0, last)
   }// end for
and now Dictionaries
<table>
<thead>
<tr>
<th>City</th>
<th>Country</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buenos Aires</td>
<td>Argentina</td>
<td>13,170,000</td>
</tr>
<tr>
<td>Cairo</td>
<td>Egypt</td>
<td>14,450,000</td>
</tr>
<tr>
<td>Cape Town</td>
<td>South Africa</td>
<td>3,092,000</td>
</tr>
<tr>
<td>London</td>
<td>England</td>
<td>12,875,000</td>
</tr>
<tr>
<td>Madrid</td>
<td>Spain</td>
<td>4,072,000</td>
</tr>
<tr>
<td>Mexico City</td>
<td>Mexico</td>
<td>20,450,000</td>
</tr>
<tr>
<td>Mumbai</td>
<td>India</td>
<td>19,200,000</td>
</tr>
<tr>
<td>New York City</td>
<td>USA</td>
<td>19,750,000</td>
</tr>
<tr>
<td>Paris</td>
<td>France</td>
<td>9,638,000</td>
</tr>
<tr>
<td>Sydney</td>
<td>Australia</td>
<td>3,665,000</td>
</tr>
<tr>
<td>Tokyo</td>
<td>Japan</td>
<td>32,450,000</td>
</tr>
<tr>
<td>Toronto</td>
<td>Canada</td>
<td>4,657,000</td>
</tr>
</tbody>
</table>
• The key (City) dictates efficient searches by key (binary)

• Any other search will require linear searching
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key: K</td>
<td></td>
</tr>
<tr>
<td>value: V</td>
<td></td>
</tr>
<tr>
<td>Entry(key: K, value: V)</td>
<td></td>
</tr>
<tr>
<td>getKey(): K</td>
<td></td>
</tr>
<tr>
<td>getValue(): V</td>
<td></td>
</tr>
<tr>
<td>#setKey(key: K): void</td>
<td></td>
</tr>
<tr>
<td>setValue(value: V): void</td>
<td></td>
</tr>
</tbody>
</table>
Dictionary< Entry<K,V> >

- items: Entry<K, V>[]

+ isEmpty(): bool
+ size(): int
+ push(key: K, value: V): void
+ pop(key: K): void
+ makeEmpty(): void
+ valueForKey(key: K): V
+ contains(key: K): bool
+ traverse(Function visit): void
possible linear implementations
### Array: Unsorted

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Cairo&quot;</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>&quot;London&quot;</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>&quot;Berlin&quot;</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>&quot;Athens&quot;</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
• Insert: at the end - avoids shifting
• Delete: could require shifting
• Find: linear search
Linked list: Unsorted

"Athens" -> "Cairo" -> "London" -> "Berlin"
• Insert: at the beginning or end if tail is used
• Delete: must find first
• Find: linear search
### Array: Sorted

<table>
<thead>
<tr>
<th></th>
<th>&quot;Berlin&quot;</th>
<th>&quot;Cairo&quot;</th>
<th>&quot;London&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;Athens&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• Insert: must find location
• Delete: must find first
• Find: binary search
Linked list: Sorted

- "Athens"
- "Berlin"
- "Cairo"
- "London"
• Insert: must find location
• Delete: must find first
• Find: linear search
hierarchical implementations
• Insert: must find first
• Delete: must find first
• Find: logN
## Putting it all together

<table>
<thead>
<tr>
<th></th>
<th>Insertion</th>
<th>Deletion</th>
<th>Retrieval</th>
<th>Traversal</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsorted Array</strong></td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>Unsorted Linked List</strong></td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>Sorted Array</strong></td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(logn)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>Sorted Linked List</strong></td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>BST</strong></td>
<td>O(logn)</td>
<td>O(logn)</td>
<td>O(logn)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
Hashing
O(1) access

Key → hashing function → index

[key]

[0]...
[1]...
......
[n-2]...
[n-1]...
A hash function must be easy and fast to compute.
It must place items evenly throughout the hash table.
Hash functions use integer arguments
hf #1 - Digit Selection
// Select the 4th and 9th digit

hf(001364825) = 35
hf(113356235) = 35

// They do not evenly distribute the keys
// Best to utilize the entire key
hf #2 - Folding
// Strategy: add all the digits together

hf(001364825) = 0 + 0 + 1 + 3 + 6 + 4 + 8 + 2 + 5
   = 29

// thus, 0 <= hf(k) <= 81 (9 * 9)
// and table would have 82 entries

// adding groups however
hf(001364825) = 001 + 364 + 825 = 1,190

// thus, 0 <= hf(k) <= (3 * 999 = 2,997)
// and table would have 2,998 entries
hf #3 - Modulus
// Strategy: make table size a prime number
// (reduces collisions - spreads items evenly)

hf(001364825) = 001364825 mod 101 = 12
convert text to an int first
ASCII values collide
Convert:
"NOTE" and "TONE"

Use ASCII values:
N = 78, O = 79, T = 84, E = 69

Add digits:
NOTE = 78 + 79 + 84 + 69 = 310
TONE = 84 + 79 + 78 + 69 = 310

Obviously, this results in collisions
Convert:
"NOTE"

Use values 1–26 (letter's alphabet position) and represent as binary:
N = 14 : 01110
O = 15 : 01111
T = 20 : 10100
E = 5 : 00101

Concatenate:
01110 01111 10100 00101 = 474,757

This takes a lot of work to compute, so...
User Horner's rule instead
Horner's simplifies things

Express in base 32 = $2^5$:

N: 14  O: 15  T: 20  E: 5
01110 01111 10100 00101 = 474,757

$32^3$  $32^2$  $32^1$  $32^0$

Collect terms:
$14 \times 32^3 + 15 \times 32^2 + 20 \times 32^1 + 5 \times 32^0$

Apply Horner's rule:
$((14 \times 32 + 15) \times 32 + 20) \times 32 + 5$

Computations can lead to int overflow, so apply hf to each () expression:

hf( hf( hf( 14 \times 32 + 15 ) \times 32 + 20 ) \times 32 + 5 )
Collisions must be resolved
Use these resolutions

- Open addressing using probing (find another entry)
- Restructure the hash table
Open addressing options

• Linear probing
• Quadratic probing
• Double hashing
Linear probing is easy
<table>
<thead>
<tr>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>[22]</td>
<td>7597</td>
</tr>
<tr>
<td>[23]</td>
<td>4567</td>
</tr>
<tr>
<td>[24]</td>
<td>0628</td>
</tr>
<tr>
<td>[25]</td>
<td>3658</td>
</tr>
</tbody>
</table>

| ... | ... |

primary clustering is common

... ...

\[ h(7597) = 22 \]

\[ h(4567) = 22 + 1, h[23] \]

\[ h(0628) = 22 + 2, h[24] \]

\[ h(3658) = 22 + 3, h[25] \]

\[ h(\text{key}) = \text{key mod 101} \]

resolution: \( h + 1 \)

\[ h(7597) = 22, h[22] \]

\[ h(4567) = 22, 22 + 1, h[23] \]

\[ h(0628) = 22, 22 + 2, h[24] \]

\[ h(3658) = 22, 22 + 3, h[25] \]
Quadratic probing eliminates primary clusters
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>7597</td>
<td>h(7597) = 22</td>
</tr>
<tr>
<td>23</td>
<td>4567</td>
<td>h + 1²</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>0628</td>
<td>h + 2²</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>3658</td>
<td>h + 3²</td>
</tr>
</tbody>
</table>

secondary clustering not a big issue

h(key) = key mod 101
resolution: h + i²

h(7597) = 22, ht[22]

h(4567) = 22 + 1², ht[23]

h(0628) = 22 + 2², ht[26]

h(3658) = 22 + 3², ht[31]
Double hashing drastically reduces clustering
$h_1(key) = key \mod 11$

$h_2(key) = 7 - (key \mod 7)$ : probe step

$h_1(58) = 3, \ p = 7 - 58 \mod 7 = 7 - 2 = 5$

$h_1(14) = 3, \text{ collision} \hfill h_2(14) = 3 + 7 = 10$

$h_1(91) = 3, \text{ collision} \hfill h_2(91) = 3 + 7 = 10 + 7 = 17 \mod 11 = 6$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[3]</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>[6]</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>[10]</td>
<td>14</td>
</tr>
</tbody>
</table>
• As the hash table fills, collisions increase
• If hash table is a dynamic array, resize it
  • each old item must be rehashed into new array
  • new size must also use a prime size
Restructuring
Buckets

chains
The size of the bucket is vital

too big: waste

too small: sooner collisions
Dictionary size can be larger than table size

collisions avoided

chains
It is all about the load factor $\alpha$
• $\alpha = \text{number of items / table size}$
• as the table fills, $\alpha$ increases and so do collisions
• efficiency decreases as collisions rise
• Thus, keep $\alpha$ at around .67 or $\frac{2}{3}$