administrivia...
assignment 13 is due tonight

Ryan’s session on Friday, WEB L114
  - 12-12:30: course code review
  - 12:30-1:30: course material review
  - 1:30-2:30: make-up lab

final exam on Monday @ 3:30pm

exam pick-up and regrades on Monday, Dec. 19th in CADE lab starting @ 9am
  - use the TA Queue
today...
- **sorting algorithms**
  - selection, insertion, shell, bubble, merge, quick

- **data structures**
  - linked list, stack, queue, tree, graph, hashmap, binary heap
selection sort
the simplest sorting algorithm
Find (ie. select) the smallest item in the unsorted portion of the array and move to the end of the sorted portion of the array.
selection sort

1) find the minimum item in the unsorted part of the array

2) swap it with the first item in the unsorted part of the array

3) repeat steps 1 and 2 to sort the remainder of the array
selection sort

1) find the minimum item in the unsorted part of the array
2) swap it with the first item in the unsorted part of the array
3) repeat steps 1 and 2 to sort the remainder of the array

what does this look like?
void selectionSort(int[] arr)
{
    for(int i=0; i < arr.length-1; i++)
    {
        min = i;
        for(int j=i+1; j < arr.length; j++)
            if (arr[j] < arr[min])
                min = j;

        temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;
    }
}

insertion sort

good for small $N$
Take the first item in the unsorted portion of the array and *insert* it into the sorted portion of the array.
insertion sort

1) the first array item in the unsorted array is the sorted portion of the array

2) take the second item and insert it in the sorted portion

3) repeat steps 1 and 2 to sort the remainder of the array
insertion sort

1) the first array item in the unsorted array is the sorted portion of the array

2) take the second item and insert it in the sorted portion

3) repeat steps 1 and 2 to sort the remainder of the array

what does this look like?
void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        index = arr[i];
        j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}

shellsort
the simplest subquadratic sorting algorithm
Divide the array (smartly) into subarrays. Do insertion sort on the subarrays. Repeat.

* Take the first item in the unsorted portion of the array and insert it into the sorted portion of the array.
shellsort
insertion sort, with a twist

1) set the gap size to $N/2$

2) consider the subarrays with elements at gap size from each other

3) do insertion sort on each of the subarrays

4) divide the gap size by 2

5) repeat steps 2 — 4 until the is gap size is $<1$
shellsort
insertion sort, with a twist

1) set the gap size to N/2

2) consider the subarrays with elements at gap size from each other

3) do insertion sort on each of the subarrays

4) divide the gap size by 2

5) repeat steps 2 — 4 until the is gap size is <1

what does this look like?
How can we describe insertion sort with respect to shellsort?
void shellSort(int[] arr)
{
    for(gap = arr.length/2; gap > 0; gap /= 2)
    {
        for(i = gap; i < arr.length; i++)
        {
            val = arr[i];
            for(j = i-gap; j >= 0 && arr[j] > val; j -= gap)
                arr[j+gap] = arr[j];
            arr[j+gap] = val;
        }
    }
}
void shellSort(int[] arr) {
    for(gap = arr.length/2; gap > 0; gap /= 2) {
        for(i = gap; i < arr.length; i++) {
            val = arr[i];
            for(j = i-gap; j >= 0 && arr[j] > val; j -= gap) {
                arr[j+gap] = arr[j];
            }
            arr[j+gap] = val;
        }
    }
}
shellsort complexity

-worst case: $O(N^2)$ with Shell’s gaps, $O(N^{3/2})$ with better gaps

-average case: $O(N^{3/2})$ with Shell’s gaps, $O(N^{5/4})$ with better gaps

-proofs of these bounds are complicated
  -the $O(N^{5/4})$ bound is based on simulations only!

-insertion sort performs better the more sorted the array
  -remember, approaches $O(N)$ for a sorted array!
bubble sort
the (usually) most inefficient sorting algorithm
Compare each pair of adjacent items and swap them if necessary. Repeat.
bubble sort

1) for each item, compare it to its next neighbor and swap if necessary

2) repeat step 1 until sorted

what does this look like?
void bubbleSort(int[] arr)
{
    for(int i=0; i < arr.length-1; i++)
    {
        for(int j=0; j < arr.length-2; j++)
            if (arr[j] > arr[j+1])
            {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = arr[j];
            }
    }
}
mergesort
divide and conquer
Merge sorted subarrays together.
first, a bit about merging...

-say we have two sorted lists, how can we efficiently merge them?

-idea: compare the first element in each list to each other, take the smallest and add to the merged list
  -AND… repeat
first, a bit about merging...

-say we have two sorted lists, how can we efficiently merge them?

-idea: compare the first element in each list to each other, take the smallest and add to the merged list

-AND... repeat

what does this look like?
mergesort

1) divide the array in half
2) sort the left half
3) sort the right half
4) merge the two halves together
mergesort

1) divide the array in half

2) sort the left half

3) sort the right half

4) merge the two halves together

what detail are we missing here?
mergesort

1) divide the array in half

2) sort the left half

3) sort the right half

4) merge the two halves together

what detail are we missing here?

how do we sort?
mergesort

1) divide the array in half

2) sort the left half

3) sort the right half

4) merge the two halves together

what detail are we missing here?

how do we sort?

can we avoid sorting?
mergesort

1) divide the array in half
2) sort the left half
3) sort the right half
4) merge the two halves together

2) take the left half, and go back to step 1
3) take the right half, and go back to step 1
mergesort

1) divide the array in half
2) take the left half, and go back to step 1 until???

3) sort the right half
3) take the right half, and go back to step 1 until???

4) merge the two halves together
mergesort

1) divide the array in half
2) sort the left half
3) sort the right half
4) merge the two halves together

what does this look like?

2) take the left half, and go back to step 1 until???
3) take the right half, and go back to step 1 until???
void mergesort(int[] arr, int left, int right)
{
    // arrays of size 1 are already sorted
    if(left >= right)
        return;

    int mid = (left + right) / 2;
    mergesort(arr, left, mid);
    mergesort(arr, mid+1, right);
    merge(arr, left, mid+1, right);
}
quicksort
another divide and conquer
Move all small items to the first subarray, move all large items to the second subarray. Sort each subarray.
quicksort

1) select an item in the array to be the *pivot*

2) *partition* the array so that all items less than the pivot are to the left of the pivot, and all the items greater than the pivot are to the right

3) sort the left half

4) sort the right half
quicksort

1) select an item in the array to be the *pivot*

2) *partition* the array so that all items less than the pivot are to the left of the pivot, and all the items greater than the pivot are to the right

3) sort the left half

4) sort the right half

**NOTE:** after partitioning, the pivot is in its final position!
quicksort

1) select an item in the array to be the pivot

2) *partition* the array so that all items less than the pivot are to the left of the pivot, and all the items greater than the pivot are to the right

3) sort the left half

4) sort the right half

**NOTE:** after partitioning, the pivot is in it’s final position!

*what do you notice?*
quicksort

1) select an item in the array to be the *pivot*

2) *partition* the array so that all items less than the pivot are to the left of the pivot, and all the items greater than the pivot are to the right

3) take the left half, and go back to step 1

4) take the right half, and go back to step 1
quicksort

1) select an item in the array to be the **pivot**

2) *partition* the array so that all items less than the pivot are to the left of the pivot, and all the items greater than the pivot are to the right

3) take the left half, and go back to step 1 *until***?

4) take the right half, and go back to step 1 *until***?
in-place partitioning

1) select an item in the array to be the *pivot*

2) swap the pivot with the last item in the array (*just to get it out of the way*)

3) step from left to right until we find an item > pivot
   - this item needs to be on the *right* of the partition

4) step from right to left until we find an item < pivot
   - this item needs to be on the *left* of the partition

5) swap items

6) continue until left and right stepping cross

7) swap pivot with left stepping item
in-place partitioning

1) select an item in the array to be the *pivot*

2) swap the pivot with the last item in the array (*just to get it out of the way*)

3) step from left to right until we find an item > pivot
   -this item needs to be on the *right* of the partition

4) step from right to left until we find an item < pivot
   -this item needs to be on the *left* of the partition

5) swap items

6) continue until left and right stepping cross

7) swap pivot with left stepping item

*what does this look like?*
find pivot, swap with right_bound;

L = left_bound, R = right_bound - 1;
while (L <= R)
{
    while (arr[L] <= pivot)
    {
        L++; // find next item > pivot
    }

    while (arr[R] >= pivot)
    {
        R--; // find its "swapping partner"
    }

    swap(arr, L, R); // partners found, swap them
    L++; R--;  
}

// point where L met R is the pivot location
swap(arr, L, right_bound); // put pivot back
void quicksort(int[] arr, int left, int right) {
    // arrays of size 1 are already sorted
    if (start >= end)
        return;

    int pivot_index = partition(arr, left, right);
    quicksort(arr, left, pivot_index - 1);
    quicksort(arr, pivot_index + 1, right);
}

linked lists
inserting into an array:

\[
\begin{array}{cccccc}
5 & 9 & 12 & 17 & 25 \\
\end{array}
\]

inserting into a linked list:

\[
\begin{array}{cccccc}
5 & \rightarrow & 9 & \rightarrow & 12 & \rightarrow & 17 & \rightarrow & 25 \\
\end{array}
\]

\[
\begin{array}{cccccc}
5 & \rightarrow & 9 & \rightarrow & 12 & \rightarrow & 17 & \rightarrow & 25 \\
\end{array}
\]
deletion from a linked list:
deletion from a linked list:

9 is now stranded — garbage collector will clean it up
deletion from a linked list:

5 → 9 → 12 → 17 → 25

9 is now stranded — garbage collector will clean it up
doubly-linked list insertion:
doubly-linked list insertion:

```java
newNode = new Node<Character>();
newNode.data = 'n';
```
doubly-linked list insertion:

newNode = new Node<Character>();
newNode.data = 'n';
newNode.prev = current;

head

a  c  k  o  y

current

n

tail
doubly-linked list insertion:

```java
doubly-linked list insertion:

newNode = new Node<Character>();
newNode.data = 'n';

newNode.prev = current;
newNode.next = current.next;
```
doubly-linked list insertion:

```java
newNode = new Node<Character>();
newNode.data = 'n';
newNode.prev = current;
newNode.next = current.next;
newNode.prev.next = newNode;
```

- **head**
  - a
- **current**
  - n
  - k
  - o
- **tail**
  - y
doubly-linked list insertion:

newNode = new Node<Character>();
newNode.data = 'n';

newNode.prev = current;
newNode.next = current.next;
newNode.prev.next = newNode;
newNode.next.prev = newNode;
doubly-linked list deletion:

current.prev.next = current.next;
current.next.prev = current.prev;
doubly-linked list deletion:

current.prev.next = current.next;
current.next.prev = current.prev;

head
a

current
c

ko

n

y
tail
doubly-linked list deletion:

current.prev.next = current.next;
current.next.prev = current.prev;
doubly-linked list deletion:

current.prev.next = current.next;
current.next.prev = current.prev;

```
  head
   a
     c
      k
        n
           o
                y
  tail
```
stacks
- a stack is a data structure in which insertion and removal is restricted to the top (or end) of the list

- also called FIRST-IN, LAST-OUT (FILO)
  - insertion always adds an item to the end
  - deletion always removes an item from the end
important methods

- **push**
  - inserts an item on to the top of the stack

- **pop**
  - removes and returns the item on the top of the stack

- **peek**
  - returns but does not remove the top of the stack

- consecutive calls to **pop** will return items in the reverse order that they were pushed
it is useful to think of stacks as standing upright (like a stack of dishes)
it is useful to think of stacks as standing upright (like a stack of dishes)
pop();
push(5);

it is useful to think of stacks as standing upright (like a stack of dishes)
queue
- A queue is a FIRST-IN, FIRST-OUT data structure
  - FIFO

- Insert on the back, remove from the front

- Operations:
  - enqueue... adds an item to the back of the queue
  - dequeue... removes and returns the item at the front

Terminology avoids confusion with a stack!

- Like a stack, all operations are $O(1)$
trees
-each node has two reference variables
  -one for each of the two children

-if there is no child, the reference is set to `null`
traversal

- **pre-order:**
  use N  // eg. print N
  DFT(N.left);
  DFT(N.right);

- **in-order:**
  DFT(N.left);
  use N  // eg. print N
  DFT(N.right);

- **post-order:**
  DFT(N.left);
  DFT(N.right);
  use N  // eg. print N

**note:** nodes are still traversed in the same order, but “used” (printed) in a different order
binary search trees (BSTs)
a binary search tree is a binary tree with a restriction on the ordering of nodes
- all items in the left subtree of a node are less than the item in the node
- all items in the right subtree of a node are greater than or equal to the item in the node

BSTs allow for fast searching of nodes
looking for h
looking for $h$

is $h$ less than or greater than $i$?
looking for $h$
looking for $h$

is $h$ less than or greater than $d$?
looking for h
looking for $h$

is $h$ less than or greater than £?
looking for h
looking for $h$

$g$ return $true$
we want to insert \( g \)
we want to insert $g$

is $g$ less than or greater than $i$?
we want to insert \( g \)
we want to insert $g$

is $g$ less than or greater than $d$?
we want to insert $g$
we want to insert g

is g less than or greater than £?
we want to insert \( g \)
we want to insert $g$
we want to insert g
-since we must maintain the properties of a tree structure, deletion is more complicated than with an array or linked-list

-there are three different cases:
  1. deleting a leaf node (delete leaf)
  2. deleting a node with one child subtree (set a reference)
  3. deleting a node with two children subtrees (find successor)

-first step of deletion is to find the node to delete
  -just a regular BST search
  -BUT, stop at the parent of the node to be deleted
graphs
- trees are a *subset* of graphs

- a graph is a set of nodes connected by edges
  - an edge is just a link between two nodes
  - nodes don’t have a parent-child relationship
  - links can be bi-directional

- graphs are used EXTENSIVELY throughout CS
what makes this a graph and not a tree?
- a path is a sequence of nodes with a start-point and an end-point such that the end-point can be reached through a series of nodes from the start-point

- in this example, there is a path from SLC to DFW
  - SLC — IAD — ATL — DFW

- there is not a path from DFW to SLC
depth-first search
- Look at the first edge going out of the start node.
- Recursively search from the new node.
- Upon returning, take the next edge.
- If no more edges, return.

- When visiting a node, mark it as visited so we don’t get stuck in a cycle.
  - Skip already visited nodes during traversal.

- For each node visited, save a reference to the node where we came from to reconstruct the path.
breadth-first search
- instead of visiting deeper nodes first, visit shallower nodes first
  - visit nodes closest to the start point first, gradually get further away

- create an empty queue
- put the starting node in the queue
- while the queue is not empty
  - dequeue the current node
  - for each unvisited neighbor of the the current node
    - mark the neighbor as visited
    - put the neighbor into the queue

- notice it is not recursive… it just runs until the queue is empty!
topological sort
topological sort

- consider a graph with no cycles

- a topological sort orders nodes such that...
  - if there is a path from node A to node B, then A appears before B in the sorted order

- example: scheduling tasks
  - represent the tasks in a graph
  - if task A must be completed before task B, then A has an edge to B
- the indegree of a node is the number of edges it has incoming

- this can be saved as part of the Node class, and can be easily computed as the graph is constructed

- any time a node adds another node as a neighbor, increase the neighbor’s indegree
1. step through each node in the graph
   - if any node has indegree 0, add it to a queue

2. while the queue is not empty
   - dequeue the first node in the queue and add to the sorted list
   - visit that node’s neighbors and decrease their indegree by 1
   - if a neighbor’s new indegree is 0, add it to the queue
dijkstra’s algorithm
- Dijkstra’s algorithm finds the *cheapest* path

- keep track of the total path cost from start node to the current node

- cost of path to next node is total cost so far plus weight of edge to next node

- instead of traversing nodes in the order they were encountered, traverse in order of cheapest total cost first
storing graphs
adjacency list

- each vertex stores an array of adjacent vertex indices, usually as a linked list

- confirming the existence of an edge for a vertex is $O(E_V)$*

- iterating through all of the edges is $O(E)$

- space complexity is $O(V + E)$

- used for sparse graphs (common case)

* $E_V$ refers to the edges for a specific vertex $V$
adjacency matrix

- matrix that represents vertices as columns and rows, with entry containing a 1 if an edge exists, and a 0 otherwise

- confirming the existence of an edge for a vertex is $O(c)$

- iterating through all of the edges is $O(V^2)$

- space complexity is $O(V^2)$

- used for dense graphs
hashmaps
<table>
<thead>
<tr>
<th></th>
<th>Access</th>
<th>Insertion</th>
<th>Deletion</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arrays / ArrayLists</strong></td>
<td>$O(c)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>must know size ahead of time</td>
</tr>
<tr>
<td><strong>Linked Lists</strong></td>
<td>$O(N)$</td>
<td>$O(N)$ or $O(c)$ on ends</td>
<td>$O(N)$ or $O(c)$ on ends</td>
<td>can allocate new items on demand</td>
</tr>
<tr>
<td><strong>Binary Search Trees</strong></td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>must be balanced</td>
</tr>
<tr>
<td><strong>Stacks</strong></td>
<td>$O(c)$</td>
<td>$O(c)$</td>
<td>$O(c)$</td>
<td>access limited to top</td>
</tr>
<tr>
<td><strong>Queues</strong></td>
<td>$O(c)$</td>
<td>$O(c)$</td>
<td>$O(c)$</td>
<td>access limited to front / back</td>
</tr>
<tr>
<td></td>
<td>Access</td>
<td>Insertion</td>
<td>Deletion</td>
<td>Notes</td>
</tr>
<tr>
<td>----------------</td>
<td>---------</td>
<td>------------</td>
<td>-----------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td><strong>Arrays / ArrayLists</strong></td>
<td>$O(c)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>must know size ahead of time</td>
</tr>
<tr>
<td><strong>Linked Lists</strong></td>
<td>$O(N)$</td>
<td>$O(N)$ or $O(c)$ on ends</td>
<td>$O(N)$ or $O(c)$ on ends</td>
<td>can allocate new items on demand</td>
</tr>
<tr>
<td><strong>Binary Search Trees</strong></td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>must be balanced</td>
</tr>
<tr>
<td><strong>Stacks</strong></td>
<td>$O(c)$</td>
<td>$O(c)$</td>
<td>$O(c)$</td>
<td>access limited to top</td>
</tr>
<tr>
<td><strong>Queues</strong></td>
<td>$O(c)$</td>
<td>$O(c)$</td>
<td>$O(c)$</td>
<td>access limited to front / back</td>
</tr>
<tr>
<td></td>
<td>Access</td>
<td>Insertion</td>
<td>Deletion</td>
<td>Notes</td>
</tr>
<tr>
<td>------------------</td>
<td>--------</td>
<td>-----------</td>
<td>----------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>Arrays / ArrayLists</td>
<td>O(c)</td>
<td>O(N)</td>
<td>O(N)</td>
<td>must know size ahead of time</td>
</tr>
<tr>
<td>Linked Lists</td>
<td>O(N)</td>
<td>O(c) on ends</td>
<td>O(c) on ends</td>
<td>new items on demand</td>
</tr>
<tr>
<td>Binary Search Trees</td>
<td>O(logN)</td>
<td>O(logN)</td>
<td>O(logN)</td>
<td>must be balanced</td>
</tr>
<tr>
<td>Stacks</td>
<td>O(c)</td>
<td>O(c)</td>
<td>O(c)</td>
<td>access limited to top</td>
</tr>
<tr>
<td>Queues</td>
<td>O(c)</td>
<td>O(c)</td>
<td>O(c)</td>
<td>access limited to front / back</td>
</tr>
</tbody>
</table>

What if we also want constant time access to any item?
<table>
<thead>
<tr>
<th></th>
<th>Access</th>
<th>Insertion</th>
<th>Deletion</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arrays / ArrayLists</strong></td>
<td>O(c)</td>
<td>O(N)</td>
<td>O(N)</td>
<td>must know size ahead of time</td>
</tr>
<tr>
<td><strong>Linked Lists</strong></td>
<td>O(N)</td>
<td>O(c) on ends</td>
<td>O(c) on ends</td>
<td>can allocate new items on demand</td>
</tr>
<tr>
<td><strong>Binary Search Trees</strong></td>
<td>O(logN)</td>
<td>O(logN)</td>
<td>O(logN)</td>
<td>must be balanced</td>
</tr>
<tr>
<td><strong>Stacks</strong></td>
<td>O(c)</td>
<td>O(c)</td>
<td>O(c)</td>
<td>access limited to top</td>
</tr>
<tr>
<td><strong>Queues</strong></td>
<td>O(c)</td>
<td>O(c)</td>
<td>O(c)</td>
<td>access limited to front / back</td>
</tr>
</tbody>
</table>

What if we also want constant time access to **any** item?
-hash functions will give us some \textit{hash value} which we can map to a valid array index using $\%$

-empty spots in the array are set to \texttt{null}

-use any hash value to directly look-up the index of any item

- insertion, deletion, and access: $O(c)$

- \textit{assuming the hash function is $O(c)$!}
-remember: *it is NOT required that two non-equal objects have different hash values*

-because of this, it is possible for two different objects to return the same hash values
  -this is called a **collision**

**insert:**
12, 15, 17, 46, 89, 90

<table>
<thead>
<tr>
<th>array:</th>
<th>90</th>
<th>12</th>
<th>15</th>
<th>46</th>
<th>17</th>
<th>89</th>
</tr>
</thead>
<tbody>
<tr>
<td>index:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
-remember: *it is NOT required that two non-equal objects have different hash values*

- because of this, it is possible for two different objects to return the same hash values
  - this is called a *collision*

insert:
12, 15, 17, 46, 89, 90, 92
remember: it is NOT required that two non-equal objects have different hash values

because of this, it is possible for two different objects to return the same hash values
  - this is called a collision

insert: 12, 15, 17, 46, 89, 90, 92

collision! where can we put 92?

array: 90 12 15 46 17 89
index: 0 1 2 3 4 5 6 7 8 9
insert with linear probing

collisions are resolved on inserts by sequentially scanning the table (with wraparound) until an empty cell is found.
insert with linear probing

array: |   |   |   |   |   |   |   |   |   | 89
index: 0 1 2 3 4 5 6 7 8 9

insert: 89
hash:   9
**insert with linear probing**

<table>
<thead>
<tr>
<th>array:</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>89</th>
</tr>
</thead>
<tbody>
<tr>
<td>index:</td>
<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>

<table>
<thead>
<tr>
<th>array:</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>index:</td>
<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>

insert: 89  
hash: 9

insert: 18  
hash: 8
insert with linear probing

array: __   __   __   __   __   __   __   __   __   89
index: 0 1 2 3 4 5 6 7 8 9

array: __   __   __   __   __   __   __   __   18 89
index: 0 1 2 3 4 5 6 7 8 9

array: 49 __   __   __   __   __   __   18 89
index: 0 1 2 3 4 5 6 7 8 9

insert: 89  hash:  9
insert: 18  hash:  8
insert: 49  hash:  9
insert with linear probing

array: __ __ __ __ __ __ __ __ __ __ 89
index: 0 1 2 3 4 5 6 7 8 9

array: __ __ __ __ __ __ __ __ __ __ 18 89
index: 0 1 2 3 4 5 6 7 8 9

array: 49 __ __ __ __ __ __ __ __ __ __ 18 89
index: 0 1 2 3 4 5 6 7 8 9

array: 49 58 __ __ __ __ __ __ __ __ 18 89
index: 0 1 2 3 4 5 6 7 8 9

insert: 89
hash: 9

insert: 18
hash: 8

insert: 49
hash: 9

insert: 58
hash: 8
insert with linear probing

array: | | | | | | | | | 89
index: 0 1 2 3 4 5 6 7 8 9

array: | | | | 18 89
index: 0 1 2 3 4 5 6 7 8 9

array: 49 | | | | 18 89
index: 0 1 2 3 4 5 6 7 8 9

array: 49 58 | | | | 18 89
index: 0 1 2 3 4 5 6 7 8 9

array: 49 58 9 | | | | 18 89
index: 0 1 2 3 4 5 6 7 8 9

insert: 89
hash: 9

insert: 18
hash: 8

insert: 49
hash: 9

insert: 58
hash: 8

insert: 9
hash: 9
quadratic probing

-probing is how to resolve collisions
  -to address clustering, we will use **quadratic probing**

-if $\text{hash}(\text{key}) = H$, and the cell at index $H$ is occupied:
  -try $H + 1^2$
  -then $H + 2^2$
  -then $H + 3^2$
  -and so on...
  -wrap around to beginning of array if necessary
insert with quadratic probing
insert with quadratic probing

| array: |         |         |         |         |         |         |         |         | 89 |
| index: | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  |

INSERT: 89
HASH: 9
# insert with quadratic probing

<table>
<thead>
<tr>
<th>array:</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>89</th>
</tr>
</thead>
<tbody>
<tr>
<td>index:</td>
<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>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>array:</th>
<th>18</th>
<th>89</th>
</tr>
</thead>
<tbody>
<tr>
<td>index:</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**INSERT:** 89  
**HASH:** 9

<table>
<thead>
<tr>
<th>array:</th>
<th>18</th>
<th>89</th>
</tr>
</thead>
<tbody>
<tr>
<td>index:</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**INSERT:** 18  
**HASH:** 8
insert with quadratic probing

array:

index:

array:

index:

array:

index:
insert with quadratic probing

array:       index: 0 1 2 3 4 5 6 7 8 9
\hline
  \text{insert: 89} & \text{hash: 9} \\
array:       index: 0 1 2 3 4 5 6 7 8 9
\hline
  \text{insert: 18} & \text{hash: 8} \\
array:       index: 0 1 2 3 4 5 6 7 8 9
\hline
  \text{insert: 49} & \text{hash: 9} \\
array:       index: 0 1 2 3 4 5 6 7 8 9
\hline
  \text{insert: 58} & \text{hash: 8}
insert with quadratic probing

array:  
index: 0 1 2 3 4 5 6 7 8 9

array:  
index: 0 1 2 3 4 5 6 7 8 9

array: 49  
index: 0 1 2 3 4 5 6 7 8 9

array: 49 58  
index: 0 1 2 3 4 5 6 7 8 9

array: 49 58 9  
index: 0 1 2 3 4 5 6 7 8 9
separate chaining

- why not make each spot in the array capable of holding more than one item?
  - use an array of linked lists
  - hash function selects index into array
  - called separate chaining

- for insertion, append the item to the end of the list
  - insertion is $O(c)$ if we have what?

- accessing is a linear scan through the list
  - fast if the list is short
binary heaps
- A binary heap is a binary tree with two special properties:
  - **structure**: it is a complete tree
  - **order**: the data in any node is less than or equal to the data of its children

- This is also called a min-heap

- A max-heap would have the opposite property

- Exists to support priority queues
- a complete binary tree has its levels completely filled, with the possible exception of the bottom level

- bottom level is filled from left to right

- each level has twice as many nodes as the previous level
complete trees as an array

-if we are guaranteed that tree is complete, we can implement it as an array instead of a linked structure

-the root goes at index 0, its left child at index 1, its right child at index 2

-for any node at index i, its two children are at index $(i \times 2) + 1$ and $(i \times 2) + 2$
- we must be careful to maintain the two properties when adding to a heap
  - structure and order

- deal with the structure property first... where can the new item go to maintain a complete tree?

- then, *percolate* the item upward until the order property is restored
  - swap upwards until > parent
- the average cases for a PQ implemented with a binary heap:
  - add
    - $O(c)$: percolate up (average of 2.6 compares)
  - findMin
    - $O(c)$: just return the root
  - deleteMin
    - $O(\log N)$: percolate down (rarely terminates before near the bottom of the tree)
min-max heaps
-a min-max heap further extends the heap order property

- for any node E at even depth, E is the minimum element in its subtree
- for any node O at odd depth, O is the maximum element in its subtree

-the root is considered to be at even depth (zero)
where is the smallest item?
where is the largest item?
AGAIN, we must ensure the heap property structure
  - must be a complete tree
  - add an item to the next open leaf node

THEN, restore order with its parent
  - does it belong on a min level or a max level?
  - swap if necessary
  - the new location determines if it is a min or max node

percolate up the appropriate levels
  - if new item is a max node, percolate up max levels
  - else, percolate up min levels
delete max

-max node is one of the two children of the root

-replace max node with the last leaf node in the tree
  -preserve structure property!

-restore order with the new node’s children
  -if any child is larger, swap
  -percolate swapped child down the max levels
  -if no child was larger, percolate the new node down the max levels

-if the node reaches the second to last level of tree, may require one more swap with direct children
delete min

- the min node is always the ___
- replace it with last leaf node
- restore order with direct children
- then, percolate new root down the min levels
delete min

- the min node is always the root
- replace it with last leaf node
- restore order with direct children
- then, percolate new root down the min levels
that’s it! good luck. it’s been fun.