Quiz 12

**A. Hash tables, continued**

Review hashmaps.
Collision resolution

- separate overflow
  - simple storage
  - rehashing
- chaining
- integrated overflow
  - simple linear probing
  - rehashing
  - multiple slots per bucket
    - simple search
    - rehashing
- hybrid

Study linear probing

Trade-offs

O(1) – for perfect hashing
O(N) – for linear probing, almost full

Study the code of linear probing.

Study chaining (since it is in the Zybooks).
A hash table with **linear probing** handles a collision by starting at the key's mapped bucket, and then linearly searches subsequent buckets until an empty bucket is found.

**18.3.1: Hash table with linear probing.**

Start

HashInsert(hashTable, item 124)
HashSearch(hashTable, 124) // Returns item 124
HashSearch(hashTable, 33) // Returns 0

Hash function: key % 10

![Diagram of hash table with linear probing]

**18.3.2: Hash table with linear probing: Insert.**

Given hash function of key % 5, determine the insert location for each item.

<table>
<thead>
<tr>
<th>#</th>
<th>Question</th>
<th>Your answer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HashInsert(numsTable, item 13)</td>
<td>bucket =</td>
</tr>
<tr>
<td></td>
<td>numsTable: 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
B. Graphs

Taxonomy of data structures:

- Directed
  - Cyclic
    - Circular LL
  - Acyclic
    - DAG
- Undirected
  - Tree
    - unary (degenerate)
    - binary
      - ordered
        - BST
      - unordered
        - heap
- Linked List
  - Stack
  - Queue

How do you store a graph where a node can point to an unlimited number of other nodes?

You can't just add more pointer fields!

```python
class GraphNode:
    def __init__(self):
        self.value = ""
        self.child1 = None
        self.child2 = None
        self.child3 = None
        ...
```

You could store a list (or array) of these children pointers.

```python
class GraphNode:
    def __init__(self):
        self.value = ""
        self.children = []
```
But in a system like C or assembler where arrays are fixed in size, that won't work! You would have to set the maximum number of children pointers.

So you could have a linked list of pointers! But the effort of searching through this linked list will be time-consuming.

<picture needed here>

Another solution would be to use a sparse matrix, called an adjacency matrix.

Every graph is comprised of two sets:

\[
\begin{align*}
\text{Node} &= \{ A, B, C, D, E, F, G, H, J, K \} \\
\text{Edges} &= \{ (H, K), (B, F), (A, H), (F, F) \ldots \}
\end{align*}
\]

where Nodes are identified somehow, by a key or a value. Edges are identified by a directed line (arrow) between two nodes.

A non-directed graph.

We can also think of this as a graph where there are two directed edges in opposite directions for each undirected edge above.
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>J</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

In an undirected graph, the transpose of the adjacency matrix is identical to the matrix.

Adjacency matrices are great for speedy lookup, counting # of connections, etc. But they are terrible for storage because so many cells will be empty.
(key, value)

\[ (\text{"Sally"}, \text{"Valentine"}) \]

\[ \xrightarrow{H} 3 \]

\[ \text{a...z} \]
\[ \text{A...Z} \]

MARK
KRAM
111

\[ H = 6 \]

\[ \text{"Mark", "Valentine"} \]

\[ \xrightarrow{H} 3 \]

\[ \text{"Kathy", "Armsworth"} \]

collision

permute
letter
import functools
from Array import *

class RehashMap:
    """ This version of a dictionary uses a fixed-size Array of tuples and hashes to the right spot.
    If the slot is already occupied then it uses linear probing. """

    def __init__(self, size=10):
        self.tuples = Array(size)  # default to holding None values
        self.array_size = size

    def __getitem__(self, key):
        print(">>> Executing: get hashmap[" + str(key) + "]")
        pos = self._make_hash(key)
        print("hashed to ", pos)
        tuple = self.tuples[pos]
        if tuple == None: return None
        if tuple[0] == key:
            return tuple[1]
        else:
            return self._linear_probe_get(key, pos)

    def _find_in_overflow(self, key):
        for pair in overflow:
            if pair[0] == key:
                return pair[1]
        return None

    def __setitem__(self, key, value):
        print(">>> Executing: hashmap[" + str(key) + "] = " + str(value))
        pos = self._make_hash(key)
        print("hashed to ", pos)
        tuple = self.tuples[pos]
        if tuple == None:
            self.tuples[pos] = (key, value)
            return
        if tuple[0] == key:
            self.tuples[pos] = (key, value)
        else:
            # collision
            self._linear_probe_put(key, pos, (key, value))

    def inspect(self):
        print("""NSPECT:"")
        for i in range(self.array_size):
            print("","end="")
            tuple = self.tuples[i]
            if tuple == None:
                print("--None--")
            else:
                print(""," + str(tuple[0]) + "," + str(tuple[1]) + ")"
    print("""END NSPECT"")

    def _make_hash(self, key):
        return functools.reduce((lambda a, b: a + b), [ord(ch) for ch in key]) % self.array_size

    def _remove_key_from_overflow(self, key):
        for tup in self.overflow:
            if tup[0] == key:
                self.overflow.remove(tup)
                break
def _linear_probe_get(self, key, pos):
    for count in range(self.array_size):
        tup = self.tuples[pos]
        if tup != None:
            if tup[0] == key:
                return tup[1]
        pos += 1
    if pos == self.array_size:  # cycle around to the beginning
        pos = 0
    return None  # not found

def _linear_probe_put(self, key, pos, the_tuple):
    for count in range(self.array_size):
        tup = self.tuples[pos]
        if tup == None:  # this spot is empty
            self.tuples[pos] = the_tuple
            return True
        else:
            if tup[0] == key:
                self.tuples[pos] = the_tuple
                return True
        pos += 1
    if pos == self.array_size:  # cycle around to the beginning
        pos = 0
    return None  # not found

x = RehashMap()

x["Mark"] = 57
x["Joanna"] = 25
x["Mary"] = 24
x.inspect()
y = x["Mark"]
print(y)
x["Mark"] = 299
y = x["Mark"]
print(y)
x.inspect()
x["Joanna"] = 6382
x.inspect()
x["Mary"] = 1717
x.inspect()
18.2.2: Hash table with chaining: Inserting items.

Given hash function of key \% 10, type the specified bucket's list after the indicated operation(s). Assume items are inserted at the end of a bucket's list. Type the bucket list as: 5, 7, 9 (or type: Empty).

<table>
<thead>
<tr>
<th>#</th>
<th>Question</th>
<th>Your answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HashInsert(valsTable, item 20) Bucket 0's list:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>HashInsert(valsTable, item 23)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HashInsert(valsTable, item 99)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bucket 3's list:</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>HashRemove(valsTable, 46) Bucket 6's list:</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>HashRemove(valsTable, 218) Bucket 8's list:</td>
<td></td>
</tr>
</tbody>
</table>