Homework 2: Skiplists and Hash Tables

CS 201 Data Structures II
Habib University
Spring 2020

Due: 1830h on Friday, 21 February

Part I
Hash Tables

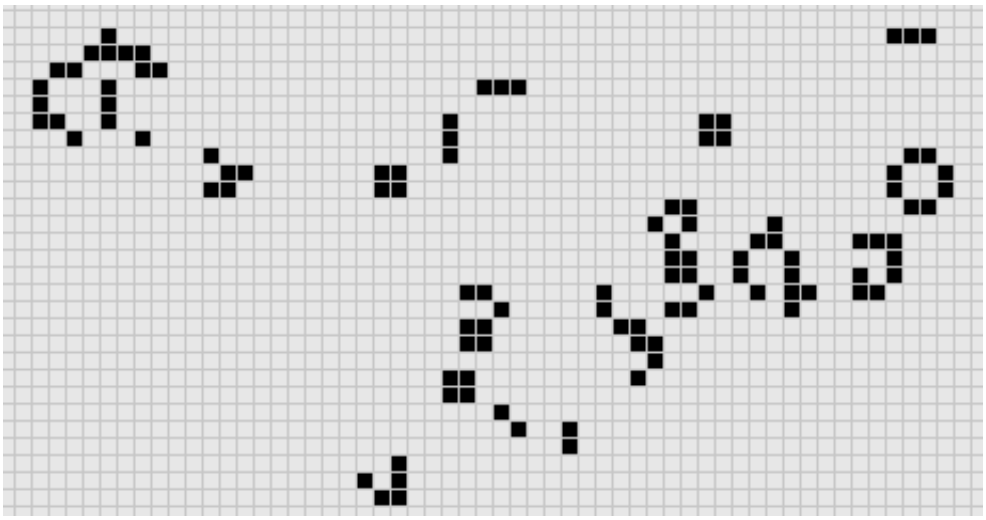

Figure 1: A sample game state in Conway's Game of Life. From Chaos and Fractals: Conway's Game of Life.

In this assignment, we will implement 2 different hash tables which differ in their conflict resolution strategy. We will use the hash table to implement Conway's Game of Life.

1 Game of Life

Conway's Game of Life is a simple simulation with surprisingly complex behavior. We start with a simple 2D grid of cells. Each cell can be either alive or dead. The rules are:

- A live cell stays alive if it has two or three live neighbors.
- A dead cell becomes alive if it has exactly three live neighbors.

We start with a certain set of cell states and then iterate, applying the rules to every cell at each iteration. Despite their simplicity, these rules can produce amazingly complicated behavior.

— A flexible implementation of Conway's Game of Life

The rules above are presented differently from their original formulation but they are nonetheless correct. This formulation is presented because it is more amenable to our eventual implementation.

1.1 Initial Configurations

The game begins with an initial configuration of live cells. As the rules of the game are deterministic, the initial configuration completely determines all subsequent states of the game. Some initial configurations lead to interesting behavior and have led to the identification of certain classes of patterns.

Still Life These patterns are not affected by further iterations of the game and stay unchanged as the game proceeds.

Oscillators These patterns recur after a fixed number of iterations which are called the *period* of the oscillator.

Spaceships These are oscillators that change position or *glide* across the grid.

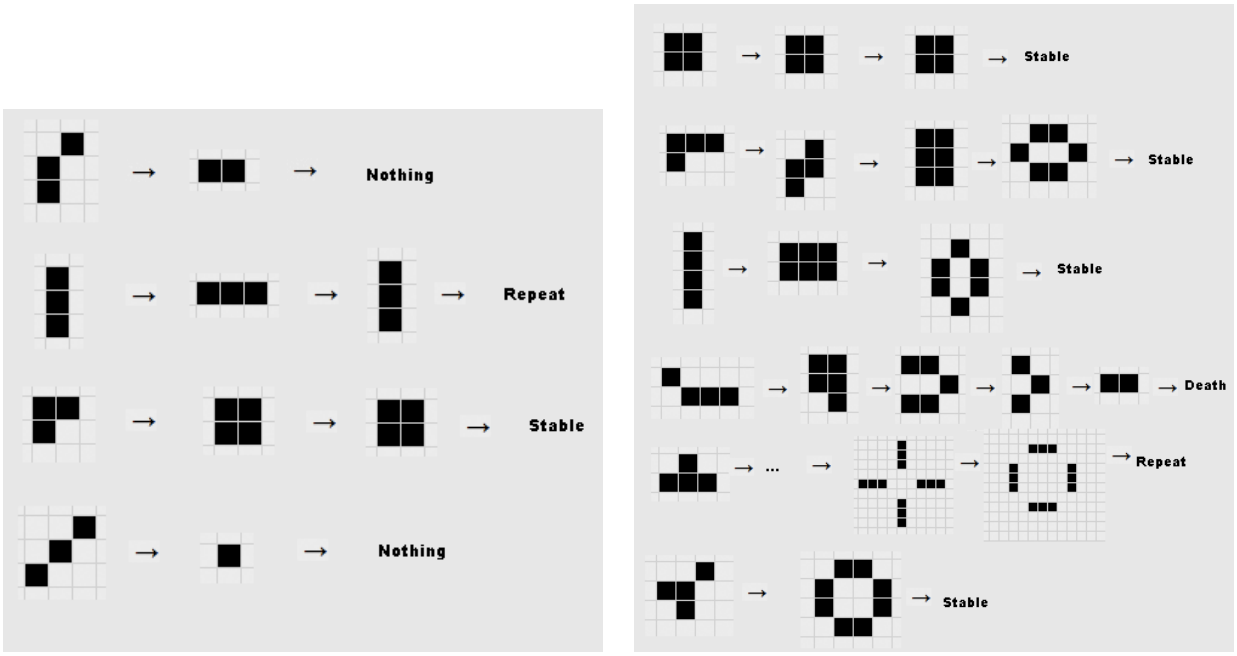

Figure 2: Evolution of the game from some initial states. From Chaos and Fractals: Conway's Game of Life.

1.2 Computation

> It's possible even, to create patterns which emulate logic gates (and, not, or, etc.) and counters. Building up from these, it was proved that the Game of Life is Turing Complete, which means that with a suitable initial pattern, one can do any computation that can be done on any computer. Later, Paul Rendell actually constructed a simple Turing Machine as a proof of concept, which can be found here.
>
> — Chaos and Fractals: Conway's Game of Life

Some links in the above excerpt that are now dead have been updated.

1.3 Visualization

As the Game of Life is a dynamic system, i.e. its stage changes with each iteration, it is best viewed as an animation which this document is unable to present. See the Wikipedia page or any of the other pages linked in this document for helpful animations. Here is a YouTube video.

2 Implementation Details

An implementation of Conway's Game of Life is provided in the classes `Life` and `Game` in the accompanying file `game.py`. The code is in the accompanying file and in the Appendix in this document.

`Life` takes its initial configuration on creation and relies on 2 structures.

Live Cells A hash table containing (x, y) coordinates of live cells.

Neighbor Count A hash table containing key-value pairs. Each key is an (x, y) coordinate of a cell with live neighbors and the value is the number of its live neighbors.

In each iteration, the live cells are used to populate neighbor count which in turn is used to update the live cells.

Do not modify the implementation of this class.

`Game` runs a `Life` instance according to provided configurations with an option for animation. In the animation system used, the origin, $(0, 0)$, is at the top left of the window. This class is provided for your testing. You may modify it as per your needs.

3 Task

Your task is to implement the structures in `Life` in 2 different ways.

Chaining The hash tables use chaining for conflict resolution.

Linear Probing The hash tables use linear probing for conflict resolution.

The hash table to use is indicated through a parameter passed to `Life` at the time of initialization.

Part II
Skiplists

The solutions to the following problems are to be entered inline below. Remove all other parts and sections from this document. Enter your team name for the date in the document's title.

1. Redundant Comparisons

[1] The `find(x)` method in a `SkiplistSet` sometimes performs redundant comparisons; these occur when `x` is compared to the same value more than once. They can occur when, for some node, `u`, `u.next[r] = u.next[r-1]`.

(a) What causes these redundant comparisons to happen?

Solution:

(b) Modify `find(x)` so that redundant comparisons are avoided.

[1] Adapted from Exercise 4.8 from the textbook.

Solution:

2. A Ranked Set

[2] Design, i.e. provide the necessary pseudo code for, a version of a skiplist that implements the `SSet` interface, but also allows fast access to elements by *rank*. That is, it also supports the function `get(i)`, which returns the element whose rank is `i` in $O(\log n)$ expected time. (The rank of an element `x` in an `SSet` is the number of elements in the `SSet` that are less than `x`.)

Solution:

3. Finger Search

[3] A *finger* in a skiplist is an array that stores the sequence of nodes on a search path at which the search path goes down. (The variable `stack` in the `add(x)` code on page 87 is a finger; the shaded nodes in Figure 4.3 show the contents of the finger.) One can think of a finger as pointing out the path to a node in the lowest list, L_0.

A *finger search* implements the find(x) operation using a finger, walking up the list using the finger until reaching a node `u` such that `u.x < x` and `u.next = nil` or `u.next.x > x` and then performing a normal search for `x` starting from `u`. It is possible to prove that the expected number of steps required for a finger search is $O(1 + \log r)$, where r is the number values in L_0 between `x` and the value pointed to by the finger.

Design, i.e. provide the necessary pseudo code for, a version of a skiplist that implements `find(x)` operations using an internal finger. This subclass stores a finger, which is then used so that every `find(x)` operation is implemented as a finger search. During each `find(x)` operation the finger is updated so that each `find(x)` operation uses, as a starting point, a finger that points to the result of the previous `find(x)` operation.

Solution:

4. BONUS: Text Search

[4] Using an `SSet` as your underlying structure, design *and implement* an application that reads a (large) text file and allows you to search, interactively, for any substring contained in the text. As the user types their query, a matching part of the text (if any) should appear as a result.

Hint 1 Every substring is a prefix of some suffix, so it suffices to store all suffixes of the text file.

Hint 2 Any suffix can be represented compactly as a single integer indicating where the suffix begins in the text.

Test your application on some large texts, such as some of the books available at Project Gutenberg. If done correctly, your application will be very responsive; there should be no noticeable lag between typing keystrokes and seeing the results.

Submit your solution as textsearch.py. It is your responsibility to design and implement the necessary interface. Your program should remotely fetch the text to operate on, e.g using the `urllib` module from Homework 1. Any special instructions regarding running and using the program should be indicated clearly in the file.

[2] Adapted from Exercise 4.9 from the textbook.
[3] Adapted from Exercise 4.10 from the textbook.
[4] Adapted from Exercise 4.14 from the textbook.

Part III
Closing

Credits

The Game of Life homework is adapted from A flexible implementation of Conway's Game of Life due to Abdullah Zafar who identified it and adapted the implementation from Racket.

A `Life`

The listing below js based on line numbers from the accompanying file, `game.py`. Modifying the file will change the listing.

```
        for i in self.t:
            if len(i) != 0:
                yield i

    def clear(self):
        self.randomInt = random.randrange(1, 101, 2)
        self.w = 1
        self.d = 0
        self.q = 0
        self.t = [()] * 2**self.w

    def resize(self):
        self.w = int(math.log2(3*self.d))
        self.w += 1
        t = self.t
        self.t = [()] * 2**self.w
        self.q = self.d
        for coord in t:
            if coord != "" and coord != ():
                hashIndex = self.hashFunction(
                    self.randomInt, coord[0], self.w)
                while self.t[hashIndex] != ():
                    hashIndex = (hashIndex + 1) % 2**self.w
                self.t[hashIndex] = coord


class LinearSet:
    def __init__(self, state):
        self.randomInt = random.randrange(1, 101, 2)
        self.w = 1
        self.d = 0
        self.q = 0
        self.t = [None] * 2**self.w
        for i in state:
            self.add(i)

    def hashFunction(self, randomInt, coord, d):
        return ((randomInt*hash(coord)) % 2**(32)) >> (32 - d)

    def add(self, coord):
        if self.find(coord) != None:
            return False
        if 2 * (self.q + 1) > 2**self.w:
            self.resize()
        hashIndex = self.hashFunction(self.randomInt, coord, self.w)
        while self.t[hashIndex] != None and self.t[hashIndex] != "":
            hashIndex = (hashIndex + 1) % 2**self.w
        if self.t[hashIndex] == None:
```

```
            self.q += 1
        self.d += 1
        self.t[hashIndex] = coord
        return True

    def find(self, coord):
        hashIndex = self.hashFunction(self.randomInt, coord, self.w)
        while self.t[hashIndex] != None:
            if self.t[hashIndex] != "" and coord == self.t[hashIndex]:
                return self.t[hashIndex]
            hashIndex = (hashIndex + 1) % 2**self.w

    def discard(self, coord):
        hashIndex = self.hashFunction(self.randomInt, coord, self.w)
        while self.t[hashIndex] != None:
            y = self.t[hashIndex]
            if y != "" and coord == y:
                self.t[hashIndex] = ""
                self.d -= 1
                if 8 * self.d < 2**self.w:
                    self.resize()
                return y
            hashIndex = (hashIndex + 1) % 2**self.w
        return None
```

B Game

The listing below js based on line numbers from the accompanying file, `game.py`. Modifying the file will change the listing.

```
            if i != None and i != "":
                yield i

    def resize(self):
        self.w = int(math.log2(3*self.d))
        self.w += 1
        t = self.t
        self.t = [None] * 2**self.w
        self.q = self.d
        for coord in t:
            if coord != "" and coord != None:
                hashIndex = self.hashFunction(
                    self.randomInt, coord, self.w)
                while self.t[hashIndex] != None:
                    hashIndex = (hashIndex + 1) % 2**self.w
                self.t[hashIndex] = coord


"""A Set implementation that uses hashing with chaining"""


class ChainedSet():
    def __init__(self, state, iterable=[]):
        self.d = 1
        self.t = self._alloc_table((1 << self.d))
        self.z = self._random_odd_int()
        self.n = 0
        for i in state:
            self.add(i)

    def _random_odd_int(self):
        return random.randrange(1 << w) | 1

    def _alloc_table(self, s):
```

```
35          return [[] for _ in range(s)]
36
37      def _resize(self):
38          temp = copy.copy(self.t)
39          self.t = []
40          self.q = self.n
41          if self.n > 0:
42
43              self.size = int(math.log2(3*self.n))
44          if self.size < 2:
```