Data Structures

9.10.1 – Learn Hashes + Heaps

Learning goals

* Understand what a hash map is (key → value) and how collisions are handled at a basic level (separate chaining).
* Understand what a min-heap is (fast access to the smallest value) and how to insert and pop elements.

Part A - Hash Map (separate chaining, tiny version)

Concepts in plain words

* A hash function turns a key into an array index.
* A collision happens when two keys land in the same bucket.
* Separate chaining = each bucket holds a list of (key, value) pairs.

Implement a miniature HashMap with:

* put(key, value)
* get(key) -> value or None
* remove(key) -> bool (True if removed)
* \_\_len\_\_() to count pairs stored

Starter (hash\_map\_starter.py).

You only implement the TODOs. I’ve removed the code, but left commented hints to guide them. These areas are bolded so you see the areas where you should add your code (remove the passes).

# hash\_map\_starter.py

# Tiny educational HashMap using separate chaining.

# Goal: implement put/get/remove with a fixed-size bucket array.

# NOTE: Keep it simple; resizing is not required for this lab.

from typing import Any, List, Tuple, Optional

class HashMap:

def \_\_init\_\_(self, capacity: int = 8):

# Each bucket is a list of (key, value) pairs

self.\_buckets: List[List[Tuple[Any, Any]]] = [[] for \_ in range(capacity)]

self.\_size = 0

def \_\_len\_\_(self) -> int:

return self.\_size

def \_index(self, key: Any) -> int:

# Map a key to bucket index

return hash(key) % len(self.\_buckets)

def put(self, key: Any, value: Any) -> None:

"""Insert or update (key, value)."""

i = self.\_index(key)

bucket = self.\_buckets[i]

**#TODO: If key exists in bucket, update its value. Else append and increase \_size.**

**#Add code here – remove pass**

pass

def get(self, key: Any) -> Optional[Any]:

"""Return value for key or None if not found."""

i = self.\_index(key)

bucket = self.\_buckets[i]

**#TODO: Scan bucket; if key found, return value. Otherwise, return None.**

**#Add code here – remove the pass**

pass

def remove(self, key: Any) -> bool:

"""Remove key if present; return True if removed else False."""

i = self.\_index(key)

bucket = self.\_buckets[i]

**#TODO: Find key, pop it from bucket, decrease \_size, return True; else False.**

**#Add code here – remove the pass**

pass

if \_\_name\_\_ == "\_\_main\_\_":

#demo (prints only)

m = HashMap(capacity=4)

m.put("alice", 10)

m.put("bob", 20)

m.put("alice", 11) # update

print("size:", len(m)) # expect 2

print("alice:", m.get("alice")) # expect 11

print("bob:", m.get("bob")) # expect 20

print("missing:", m.get("zzz")) # expect None

print("removed bob:", m.remove("bob")) # expect True

print("size now:", len(m)) # expect 1

**Add a screen shot of your output here:**

Part B - Min-Heap (priority queue)

Concepts in plain words

* A min-heap is a complete binary tree where the smallest item is always at the root.
* It lives nicely in a list:
  + parent(i) = (i-1)//2, left(i) = 2\*i+1, right(i) = 2\*i+2.
* Two classic moves:
  + sift-up (after insert) to restore order,
  + sift-down (after pop root) to restore order.

Implement a miniature MinHeap with:

* push(x)
* peek()
* pop() (or extract\_min())

Starter (min\_heap\_starter.py) again, code is removed but hinted in comments.

# min\_heap\_starter.py

# Minimal educational Min-Heap (array-backed)

# Implement push (sift-up), pop/extract\_min (sift-down), and peek.

from typing import List

class MinHeap:

def \_\_init\_\_(self):

self.\_a: List[int] = []

def \_\_len\_\_(self) -> int:

return len(self.\_a)

def \_\_bool\_\_(self) -> bool:

return bool(self.\_a)

# Index helpers (0-based)

def \_parent(self, i: int) -> int: return (i - 1) // 2

def \_left(self, i: int) -> int: return 2 \* i + 1

def \_right(self, i: int) -> int: return 2 \* i + 2

def peek(self) -> int:

if not self.\_a:

raise IndexError("peek from empty heap")

return self.\_a[0]

def push(self, x: int) -> None:

**#TODO: append x; then sift-up while parent > x. Swap as needed.**

**#Add code here – remove the pass**

pass

def pop(self) -> int:

"""Remove and return smallest element."""

if not self.\_a:

raise IndexError("pop from empty heap")

**#TODO: store min\_val (root), move last to root, and sift-down.**

**# Add code here – remove the pass**

pass

def \_sift\_down(self, i: int) -> None:

n = len(self.\_a)

while True:

left = self.\_left(i)

right = self.\_right(i)

smallest = i

if left < n and self.\_a[left] < self.\_a[smallest]:

smallest = left

if right < n and self.\_a[right] < self.\_a[smallest]:

smallest = right

if smallest == i:

break

self.\_a[i], self.\_a[smallest] = self.\_a[smallest], self.\_a[i]

i = smallest

if \_\_name\_\_ == "\_\_main\_\_":

#demo sequence

h = MinHeap()

for x in [7, 12, 3, 19, 5]:

h.push(x)

print("peek:", h.peek()) # expect 3

out = []

while h:

out.append(h.pop())

print("popped:", out) # expect sorted order

**Add a screen shot of your output here:**

What to turn in

* hash\_map\_starter.py with working put/get/remove.
* min\_heap\_starter.py with working push/peek/pop.
* Required screen shots added to this document
* Create a video walkthrough of both .py files.