2024_DS_Fall_Homework 2

Notice

- **The deadline is 2024/12/31 23:59**. Homework should be submitted as a c source file, not an executable file. In your homework assignment, read the input from stdin and write your output to stdout.
- You should submit your homework to **140.116.245.97**.
- Please register on the platform before submitting your assignment. Your account should be in the format: **student ID_HW2**, with the student ID in uppercase letters. Example: A12345678_HW2.
- If the account is set incorrectly, please register a new correct account yourself. The teaching assistant will not assist with account transfers.
- If the account name does not meet the requirements, the teaching assistant will be unable to verify your assignment grade, and the grade will be 0. If you inform us after the deadline and need to manually update your registration, the grade will be calculated with a 10% penalty.
- Please pay attention to the system submission deadline and time. The system will automatically close at that time, and there will be no grace period

---

Problem 1: Implementing Basic Operations Using Hash Table (3%)

Description:

In this problem, you are required to design a hash table using a specified hash function to store integer keys. The hash function is defined as **hash(key) = key % n**, where **key** is the input number and **n** is the number of buckets. Each bucket contains **m** slots, which represent positions within the bucket. **Linear probing** should be used to resolve collisions when a bucket's slots are full.

Your hash table should support the following operations:

- **Insert a key**: Add a key to the hash table. If the target bucket has available slots, insert the key starting from the smallest slot. If all slots in the bucket are full, use linear probing to find the next available bucket.
- **Search for a key**: Check if a key exists in the hash table and return the index of the bucket where the key was found.
- **Delete a key**: Remove a key from the hash table. Ensure proper handling of deletion to maintain consistent search behavior.

Input Format:

- The first line contains the number of buckets: bucket <n>, where 1 <= n <= 100 (e.g. bucket 10).
- The second line contains the number of slots per bucket: slot <m>, where 1 <= m <= 20 (e.g., slot 2)
- Each subsequent line contains one of the following commands:
    - insert <key> to insert a key
    - delete <key> to delete a key
    - search <key> to search for a key
- exit to terminate the input sequence and end the program.

Notes:

- In the test cases, **no** duplicate values will be inserted into the Hash Table. Each key will be unique within the Hash table.
- The test cases will **not** contain operations that attempt to delete or search non-existent keys, so you do not need to handle such cases.
- Keys are non-negative integers and satisfy **0 <= key <= 10^4**.
- The total number of commands **q** satisfies **1 <= q <= 100**. Commands are executed sequentially.

Output Format:

- For each search command, output the bucket and slot positions of the key in the format <bucket> <slot>. Print the results of all search commands in the order they appear.

Example:

Sample Input:

```
bucket 10

slot 2

insert 3

insert 13

insert 4

search 13
```

```
delete 3

search 4

exit
```

Sample Output:

```
3 1

4 0
```

Explanation:

In this example, the hash table has 10 buckets, and each bucket contains 2 slots. The operations are executed as follows:

1. **insert 3**: hash(3) = 3 % 10 = 3. Insert 3 into bucket 3, placing it in slot 0 (the first available slot).
2. **insert 13:** hash(13) = 13 % 10 = 3. Bucket 3's slot 0 is already occupied, so 13 is placed in slot 1 (the next available slot in the same bucket).
3. **insert 4:** hash(4) = 4 % 10 = 4. Bucket 4's slot 0 is empty, so 4 is placed in slot 0.
4. **search 13:** hash(13) = 3. Search bucket 3 starting from slot 0. 13 is found in slot 1, so the output is 3 1.
5. **delete 3:** hash(3) = 3. Remove 3 from bucket 3, clearing slot 0.
6. **search 4:** hash(4) = 4. 4 is found in bucket 4, slot 0, so the output is 4 0.
7. **exit**: After all operations are complete, the results of the search commands are printed in the format <bucket> <slot>.

The resulting output is:

```
3 1
4 0
```

This indicates that 13 was found in bucket 3, slot 1, and 4 was found in bucket 4, slot 0.

Problem 2: Implementing Fibonacci Heap (10%)

Description

In this problem, you are required to implement Fibonacci Heap. Your implementation should support **insertion**, **deletion**, **decrease-key**, and **extract-min** operations while maintaining the properties of a Fibonacci Heap. After executing all commands, the program should output:

- The final structure of the Fibonacci Heap, printed in **level-order traversal**, where:
  - Nodes in the same tree and at the same level are separated by a space.
  - Nodes belonging to different trees in the root list are printed on separate lines.

Input Format

The input consists of a series of commands, each on a new line:

- insert <key>: Insert a key into the Fibonacci Heap.
- delete <key>: Delete a key from the Fibonacci Heap.
- decrease <key> <value>: Reduce the value of an existing key by value.
- extract-min: Remove the current minimum key from the heap.
- exit: Terminate the input sequence and end the program.

The commands will be executed sequentially, and the program will stop processing once the exit command is encountered.

The properties of the Fibonacci Heap and the operations can be referenced from the lecture slides.

Notes:

- Keys are positive integers and satisfy **1 <= key <= 10^4**.
- Keys are unique within the Fibonacci Heap.
- The total number of commands **q** satisfies **1 <= q <= 100**. Commands are executed sequentially.
- No operation will attempt to delete or decrease a key that does not exist.
- Operations such as extract-min will only be executed when the heap is non-empty.
- The command before the exit operation must be an extract-min operation.
- The Fibonacci Heap must maintain its properties:
  - The minimum pointer (min) always points to the smallest key.
  - Consolidation is performed after every extract-min and delete operation.
  - During consolidation, when multiple roots have the same degree, the two roots with the smallest key values are merged first and the children of the root of the new tree should be sorted in ascending order(from left to right).

    - Cascading cuts are performed during delete, decrease-key operations, if necessary.

Output Format

After processing all commands, output the final Fibonacci Heap structure in **level-order traversal**:

- For each tree in the root list, the result of the level-order traversal is printed in a single line.
- Start with the tree of a smaller degree (e.g. degree = 0 → degree = 1 → …).

Example 1

Sample Input:

```
insert 10

insert 20

insert 5

insert 30

insert 25

extract-min

decrease 30 22

insert 15

insert 12

extract-min

delete 12

extract-min

exit
```

Sample Output:

```
25
```

```
15 20
```

Explanation:

1. **Input Commands and Heap Evolution:**
   - **insert 10**: Add 10 as the only node.
   - **insert 20**: Add 20. The minimum remains 10. The root list becomes [10, 20].
   - **insert 5**: Add 5. The minimum pointer is updated to 5. The root list becomes [10, 20, 5].
   - **insert 30**: Add 30. The minimum remains 5. The root list becomes [10, 20, 5, 30].
   - **insert 25**: Add 25. The minimum remains 5. The root list becomes [10, 20, 5, 30, 25].

   Struct:

   ```
   10  —  20  —  5  —  30  —  25
   ```

2. **extract-min Operation:**
   - Remove the minimum value 5.
   - No children for node 5, so the root list remains [10, 20, 30, 25].
   - Perform **consolidation (1)**:
     - Merge 10 and 20 (degree 0 trees) → 10 becomes the root, 20 becomes its child.
     - Merge 30 and 25 (degree 0 trees) → 25 becomes the root, 30 becomes its child.
     - Root list becomes [10, 25], with min pointing to 10.
   - Perform **consolidation (2)**:
     - Merge 10 and 25 (degree 1 trees) → 10 becomes the root, 20 , 25 becomes its child.

   Struct:

   ```
      10  —  25                      10
      /      /          →          /    \
   20      30                    20      25
                                         /
                                       30
   ```

3. **decrease 30 22 Operation:**
   - The key 30 is reduced to 8.
   - Since 8 violates the heap order property, it is cut from its parent 25 and moves to the root list.

- Perform cascading cuts (none in this case, as 25 does not lose its second child).
- **Root List:** [10, 8].
- The minimum pointer is updated to 8.

Struct:

```
      10  —  8
     /   \
   20     25
```

4. **insert 15 and insert 12 Operations:**
   - Insert 15 → Root list becomes [10, 8, 15].
   - Insert 12 → Root list becomes [10, 8, 15, 12].

Struct:

```
      10  —  8  —  15  —  12
     /   \
   20     25
```

5. **extract-min Operation:**
   - Removes and outputs the minimum value 8.
   - Perform **consolidation**:
     - Merge 15 and 12 (degree 0 trees) → 12 becomes the root, 15 becomes its child.
     - Root list becomes [10, 12], with min pointing to 10.

Struct:

```
      10  —  12
     /  \     /
   20    25  15
```

6. **delete 12 Operation:**
   - Deletes the node with key 12 and 15 moves to the root list .
   - Root list becomes [10, 15].

Struct:

```
      10  —  15
     /  \
   20    25
```

7. **extract-min Operation:**
   - Removes the minimum value 10.
   - Perform **consolidation**:
     - 20 and 25 move to the root list. Root list becomes [15, 20, 25].

    - Merge 15 and 20 (degree 0 trees) → 15 becomes the root, 20 becomes its child.
    - Root list becomes [15, 25], with min pointing to 15.

   Struct:

```
   15  —  20  —  25                  15  —  25
                            →         /
                                    20
```

8. **The final result of the heap is:**

   (Degree 0):

   25

   (Degree 1):

   15 20

Example 2

Sample Input:

insert 10

insert 20

insert 5

insert 3

insert 7

insert 15

insert 18

insert 22

insert 1

insert 12

extract-min

decrease 20 1

```
insert 25

insert 30

extract-min

decrease 30 10

extract-min

decrease 25 4

insert 8

insert 9

extract-min

insert 1

extract-min

extract-min

exit
```

Sample Output:

```
9

10 12 20 22 15 18 21 19
```
