CSE 332 Summer 18
Section 05

1 Hashing: Mechanical

1. Suppose we have a hash table that uses separate chaining and has an internal capacity of 12 (do NOT worry about resizing for this problem. Assume that each bucket is a linked list where new elements are added to the front of the list.

   Insert the following elements in the EXACT order given using the hash function $h(x) = 4x$:

   0, 4, 7, 1, 2, 3, 6, 11, 16

2. Suppose we have a hash table that uses linear probing and has an internal capacity of 13.

   Insert the following elements in the EXACT order given using the hash function $h(x) = 3x$:

   2, 4, 6, 7, 15, 13, 19

3. Suppose we have a hash table that uses quadratic probing and has an internal capacity of 10.

   Insert the following elements in the EXACT order given using the hash function $h(x) = x$:

   0, 1, 2, 5, 15, 25, 35

4. Consider the following key-value pairs.

   (6, a), (29, b), (41, d). (34, e), (10, f), (64, g), (50, h)

   Suppose each key has a hash function $h(k) = 2k$. So, the key 6 would have a hash code of 12. Insert each key-value pair into the following hash tables and draw what their internal state looks like:

   (a) A hash table that uses separate chaining. The table has an internal capacity of 10. Assume each bucket is a linked list, where new pairs are appended to the end. Do not worry about resizing.

(b) A hash table that uses linear probing, with internal capacity 10. Do not worry about resizing.

(c) A hash table that uses quadratic probing, with internal capacity 10. Do not worry about resizing.

5. Consider the three hash tables in the previous question. What are the load factors of each hash table?

2 Hashing: Conceptual

1. What is the difference between primary clustering and secondary clustering in hash tables?

2. Suppose we implement a hash table using double hashing. Is it possible for this hash table to have clustering?

3. Suppose we have a hash table with an initial capacity of 12. We resize the hash table by doubling the capacity. Suppose we insert integer keys into this table using the hash function $h(x) = 4x$. Why is this choice of hash function and initial capacity suboptimal? How can we fix it?

4. Suppose we have a hash table with an initial capacity of 8 using linear probing. We resize the hash table by doubling the capacity.

   Suppose we insert the integer keys 2^{20}, $2 \cdot 2^{20}$, $3 \cdot 2^{20}$, $4 \cdot 2^{20} \ldots$ using the hash function $h(x) = x$.

   Describe what the runtime of the dictionary operations will over time as you keep adding these keys to the table.

3 Hashing: Code Analysis

For this problem, we will consider a hypothetical hash table that uses linear probing and implements the `Dictionary` interface. Specifically, we will focus on analyzing and testing one potential implementation of the `remove` method.

1. Come up with at least 4 different test cases to test this `remove(...)` method. For each test case, describe what the expected outcome is (assuming the method is implemented correctly).

   Try and construct test cases that check if the `remove(...)` method is correctly using the key's hash code. (You may assume that you can construct custom key objects that let you customize the behavior of the `equals(...)` and `hashCode()` method.)

2. Now, consider the following (buggy) implementation of the `remove(...)` method. List all the bugs you can find.

```
public class LinearProbingDictionary<K, V> implements IDictionary<K, V> {
    // Field invariants:
    //
    // 1. Empty, unused slots are null
    // 2. Slots that are actually being used contain an instance
    //    of a Pair object

    private Pair<K, V>[] array;

    // ...snip...

    public V remove(K key) {
        int index = key.hashCode();

        while ((this.array[index] != null)
                && !this.array[index].key.equals(key)) {
            index = (index + 1) % this.array.length;
        }

        if (this.array[index] == null) {
            throw new NoSuchKeyException();
        }
        V returnValue = this.array[index].value;
        this.array[index] = null;
        return returnValue;
    }
}
```

3. Briefly describe how you would fix these bug(s).

4 B-Trees

1. Insert the following into an empty B-Tree with M = 3 and L = 3: 12, 24, 36, 17, 18, 5, 22, 20.

2. Given the following parameters for a B-Tree with $M = 11$ and $L = 8$:

   - Key Size = 10 bytes
   - Pointer Size = 2 bytes
   - Data Size = 16 bytes per record (includes the key)

   Assuming that M and L were chosen appropriately, what is the likely page size on the machine where this implementation will be deployed? Give a numeric answer based on two equations using the parameter values above. **Hint**: Some equations you might need to use are:

$$M = \lfloor \frac{p+k}{t+k} \rfloor$$
$$L = \lfloor \frac{p}{k+v} \rfloor$$

   where p is the page size in bytes, k is key size in bytes, t is pointer size in bytes, and v is value size in bytes. **Hint**: Think about where these values come from.

3. Give an example of a situation that would be a good job for a B-tree. Furthermore, are there any constraints on the data that B-trees can store?

4. Find a tight upper bound on the worst case runtime of these operations on a B-tree. Your answers should be in terms of L, M, and n.

   (a) Insert a key-value pair

   (b) Look up the value of a key

   (c) Delete a key-value pair

5. Insert and then delete the following from a B-tree:

   (a) Insert the following into an empty B-Tree with $M = 3$ and $L = 3$: 3, 18, 14, 30, 32, 36, 15, 16, 12, 40, 45, 38.

   (b) Delete 45, 14, 15, 36, 32, 18, 38, 40, 12 from the tree in the previous part.

5 Memory

1. What are the two types of memory locality?

2. Does this more benefit arrays or linked lists?

3. What about Java makes it a poor choice for implementing B-trees?

6 Memory and B-Trees

1. Based on your understanding of how computers access and store memory, why might it be faster to access all the elements of an array-based queue than to access all the elements of a linked-list-based queue?

2. Why might `f2` be faster than `f1`?

```
public void f1(String[] strings) {
    for (int i=0; i < strings.length; i++) {
        strings[i] = strings[i].trim(); // omits trailing/leading whitespace
    }
    for (int i=0; i < strings.length; i++) {
        strings[i] = strings[i].toUpperCase();
    }
}
```

```
public void f2(String[] strings) {
    for (int i=0; i < strings.length; i++) {
        strings[i] = strings[i].trim(); // omits trailing/leading whitespace
        strings[i] = strings[i].toUppercase();
    }
}
```

3. Let k be the size of a key, t be the size of a pointer, and v be the size of a value.

   Write an expression (using these variables as well as M and L) representing the size of an internal node and the size of a leaf node.

4. Suppose you are trying to implement a B-tree on a computer where the page size (aka the block size) is $p = 130$ bytes.

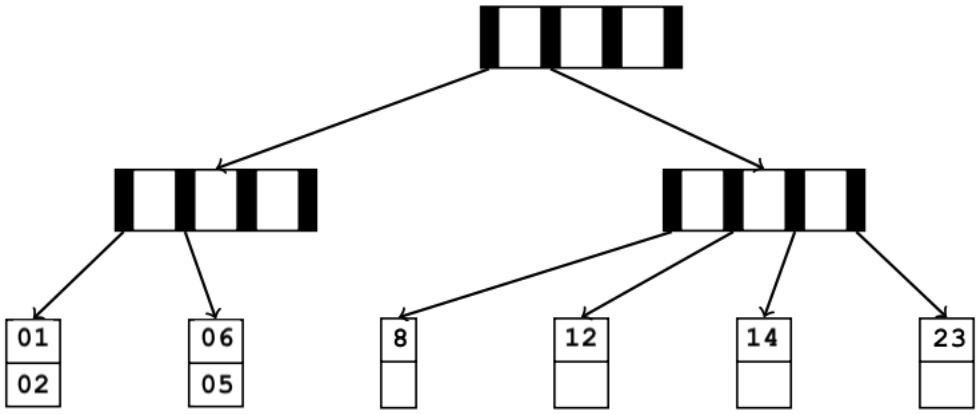
   You know the following facts:

   - Key size $k = 10$ bytes
   - Value size $v = 6$ bytes
   - Pointer size $t = 2$ bytes

   What values of M and L should you pick to make sure that your internal and external nodes (a) fit within a single page and (b) uses as much of that page as possible.

   Be sure to show your work.

5. Consider the following "B-Tree":



(a) What are M and L?

(b) Is there anything wrong with the above B-Tree? If so, what is wrong?

6. Consider the following code:

```
public static int sum(List<Integer> list) {
    int output = 0;
    for (int i = 0; i < 128; i++) {
        // Reminder: foreach loops in Java use the iterator behind-the-scenes
        for (int item : list) {
            output += item;
        }
    }
    return output;
}
```

You try running this method twice: the first time, you pass in an array list, and the second time you pass in a linked list. Both lists are of the same length and contain the exact same values.

You discover that calling `sum` on the array list is consistently 4 to 5 times faster then calling it on the linked list. Why do you suppose that is?

7. Suppose you are trying to use a B-Tree somebody else wrote for your system. You know the following facts:

   - $M = 10$ and $L = 12$
   - The size of each pointer is 16 bytes
   - The size of each key is 14 bytes
   - The size of each value is 11 bytes

   Assuming M and L were chosen wisely, what is most likely the page size on this system?

7 Memory and B-Trees: A Sequel

1. Suppose you've finished writing your `AVLTree` Dictionary. Right? Out of curiosity, you try replacing it with a `SortedArrayDictionary`. You expect this to make no difference since iterating over either dictionary using their iterator takes worst-case $\Theta(n)$ time.

   To your surprise, iterating over `SortedArrayDictionary` is consistently almost 10 times faster!

   Based on your understanding of how computers organize and access memory, why do you suppose that is? Be sure to be descriptive.

2. Excited by your success, you next try comparing the performance of the `get(...)` method. You expected to see the same speedup, but to your surprise, both dictionaries' `get(...)` methods seem to consistently perform about the same.

   Based on your understanding of how computers organize and access memory, why do you suppose that is?

   (Note: assume that the `SortedArrayDictionary`'s `get(...)` method is implemented using binary search.)

3. You want to implement a B-Tree for a computer that has a page or block size of $p = 256$ bytes. Your pointers are $t = 4$ bytes long, your keys are $k = 2$ byte long, and your values are $v = 8$ bytes long. What should you select for M and L in order to maximize the performance of your B-tree? Please show your work.

   Reminder: M and L must selected such that the following two inequalities remain true:

   $$Mt + (M-1)k \leq p \qquad \text{and} \qquad L(k+v) \leq p$$

8 Challenge Problem: Random Hash Functions

In class we talked about various strategies to minimize collisions. In this question we discuss how to use randomness to "spread out" collisions from a small set of very bad inputs into a larger set of almost-always-fine inputs. The last two parts of this problem are beyond the scope of this course, but are interesting nonetheless.

For simplicity, assume our keyspace (the set of possible keys) is the set $\{0, 1, 2, \ldots, 2^{30} - 1\}$. Suppose we have a hashtable of size 2^{10}. Let a be an odd integer less than 2^{30}.

Consider the hash function

$$h_a(x) = \left\lfloor \frac{(ax) \mod 2^{30}}{2^{20}} \right\rfloor$$

Notice that the function changes depending on the value of a we choose, so this is really a set of possible functions.

1. Show that for any a, h_a outputs an integer between 0 and $2^{10} - 1$ (i.e. we can use this as a hash function for our table size)

2. Choose $a = 1$, i.e. the hash function simplifies to

$$h_1(x) = \left\lfloor \frac{x \mod 2^{30}}{2^{20}} \right\rfloor$$

   For this function, find a large set of elements that all hash to 0.

3. Let x, y be any of the two elements you found in the last part. Choose a few thousand values of a, and check whether $h_a(x) = h_a(y)$ for each of them (write code for this part). For what fraction of these hash functions do x, y collide? If the values of the hash function were totally random, how often would you expect collisions?

4. The following statement is true (explaining why is beyond the scope of the course): For any x, y if you choose a at random, the probability that $h_a(x) = h_a(y)$ is at most $2/2^{10}$.

   Use this fact, or your observations in the last part, to Explain why you might decide to choose a random a instead of just choosing $a = 1$ (hint: imagine you know someone is using the hash function with $a = 1$, how can you use the first part to slow their code down? Can you do the same for a random a?)