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

Solution: To make the problem easier for ourselves, we first start by computing the hash values and initial indexes:

key	hash	index (pre probing)
0	0	0
4	16	4
7	28	4
1	4	4
2	8	8
3	12	0
6	24	0
11	44	8
16	64	4

The state of the internal array will be

$6 \to 3 \to 0$	/	/	/	$16 \to 1 \to 7 \to 4$	/	/	/	$11 \to 2$	/	/	/

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

Solution: Again, we start by forming the table:

key	hash	index (before probing)
2	6	6
4	12	12
6	18	5
7	21	8
15	45	6
13	39	0
19	57	5

Next, we insert each element into the internal array, one-by-one using linear probing to resolve collisions. The state of the internal array will be:

13	/	/	/	/	6	2	15	7	19	/	/	4

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

Solution: The state of the internal array will be:

0	1	2	35	/	5	15	/	/	25

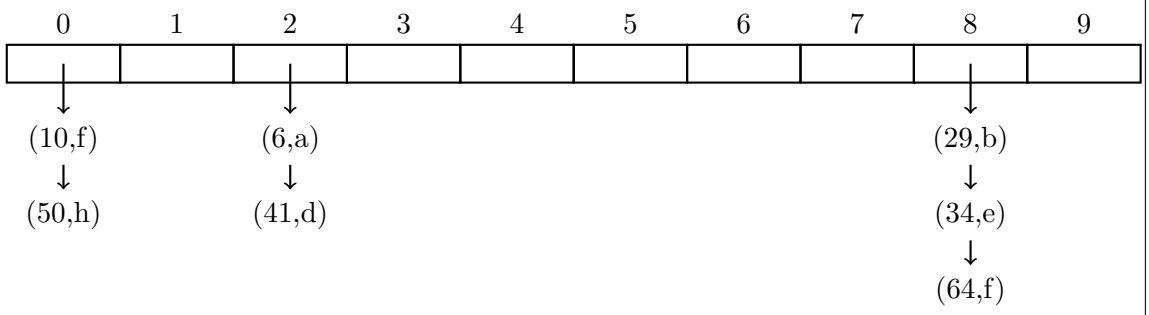
4. Consider the following key-value pairs.

(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.

Solution:



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

Solution:

0	1	2	3	4	5	6	7	8	9
(10, f)	(64, g)	(6, a)	(41, d)	(50, h)				(29, b)	(34, e)

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

Solution:

0	1	2	3	4	5	6	7	8	9
(10,f)	(50, h)	(6,a)	(41,d)				(64, g)	(29,b)	(34,e)

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

Solution:

$$\lambda = \frac{7}{10} = 0.7$$

2 Hashing: Conceptual

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

Solution: Primary clustering occurs after a hash collision causes two of the records in the hash table to hash to the same position, and causes one of the records to be moved to the next location in its probe sequence. Linear probing leads to this type of clustering.

Secondary clustering happens when two records would have the same collision chain if their initial position is the same. Quadratic probing leads to this type of clustering.

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

Solution: Yes, though the clustering is statistically less likely to be as severe as primary or secondary 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?

Solution: Notice that the hash function will initially always cause the keys to be hashed to at most one of three spots: 12 is evenly divided by 4.

This means that the likelyhood of a key colliding with another one dramatically increases, decreasing performance.

This situation does not improve as we resize, since the hash function will continue to map to only a fourth of the available indices.

We can fix this by either picking a new hash function that's relatively prime to 12 (e.g. $h(x) = 5x$), by picking a different initial table capacity, or by resizing the table using a strategy other then doubling (such as picking the next prime that's roughly double the initial size).

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.

Solution: Initially, for the first few keys, the performance of the table will be fairly reasonable.

However, as we insert each key, they will keep colliding with each other: the keys will all initially mod to index 0.

This means that as we keep inserting, each key ends up colliding with every other previously inserted key, causing all of our dictionary operations to take $\mathcal{O}(n)$ time.

However, once we resize enough times, the capacity of our table will be larger than 2^{20}, which means that our keys no longer necessarily map to the same array index. The performance will suddenly improve at that cutoff point then.

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.)

   Solution: Some examples of test cases:

   - If the dictionary contains null keys, or if we pass in a null key, everything should still work correctly.
   - If we try removing a key that doesn't exist, the method should throw an exception.
   - If we pass in a key with a large hash value, it should mod and stay within the array.
   - If we pass in two different keys that happen to have the same hash value, the method should remove the correct key.
   - If we pass in a key where we need to probe in the middle of a cluster, removing that item shouldn't disrupt lookup of anything else in that cluster.

     For example, suppose the table's capacity is 10 and we pass in the integer keys 5, 15, 6, 25, 36 in that order. These keys all collide with each other, forming a primary cluster. If we delete the key 15, we should still successfully be able to look up the values corresponding to the other keys.

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;
    }
}
```

Solution: The bugs:

- We don't mod the key's hash code at the start
- This implementation doesn't correctly handle null keys
- If the hash table is full, the while loop will never end
- This implementation does not correctly handle the "clustering" test case described up above.

  If we insert 5, 15, 6, 25, and 36 then try deleting 15, future lookups to 6, 25, and 36 will all fail.

Note: The first two bugs are, relatively speaking, trivial ones with easy fixes. The middle bug is not trivial, but we have seen many examples of how to fix this. The last bug is the most critical one and will require some thought to detect and fix.

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

Solution:

- Mod the key's hash code with the array length at the start.
- Handle null keys uniquely (extra `if` checks)
- There should be a size field, with `ensureCapacity()` functionality.
- Ultimately, the problem with the "clustering" bug stems from the fact that breaking a primary cluster into two in any way will inevitably end up disrupting future lookups.

  This means that simply setting the element we want to remove to null is not a viable solution. There are a few different ways to solve this, but we'll only discuss one here (you'll discover the rest in your project).

  A common solution would be to use lazy deletion. Rather then trying to "fill" the hole, we instead modify each `Pair` object so it contains a third field named `isDeleted`.

  Now, rather then nulling that array entry, we just set that field to true and modify all of our other methods to ignore pairs that have this special flag set. When rehashing, we don't copy over these "ghost" pairs.

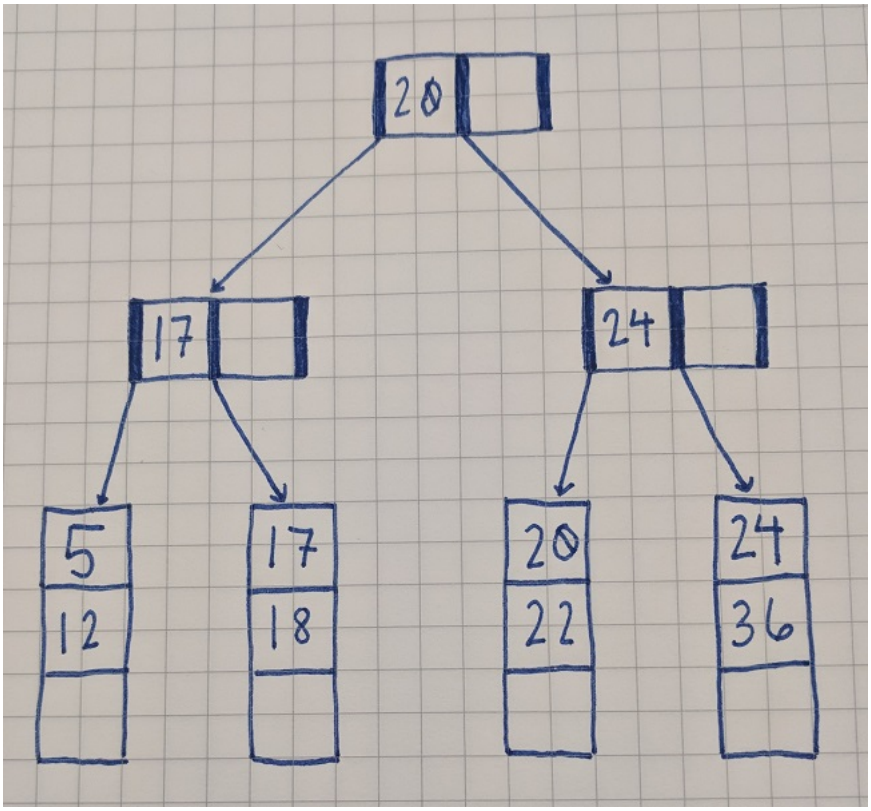
  This helps us keep our delete method relatively efficient, since all we need to do is to toggle a flag.

  However, this approach also does complicate the runtime analysis of our other methods (the load factor is no longer as straightforward, for example).

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.

Solution:



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.

$$\begin{aligned} p &\geq Mt + (M-1)k \\ &\geq (11)(2) + (11-1)(10) \\ &\geq 122 \\ p &\geq L(k+v) \\ &\geq 8(16) \\ &\geq 128 \end{aligned}$$

   Page size must be at least 128 bytes.

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?

   Solution: B-trees are most appropriate for very, very large data stores, like databases, where the majority of the data lives on disk and cannot possibly fit into RAM all at once. B-trees require orderable keys. B-trees are typically not implemented in Java because because what makes them worthwhile is their precise management of memory.

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

   Solution: The steps for insert and delete are similar and have the same worst case runtime.

   i. Find the leaf: $\mathcal{O}(lg(M)log_M(n))$.
   ii. Insert/remove in the leaf – there are L elements, essentially stored in an array: $\mathcal{O}(L)$
   iii. Split a leaf/merge neighbors: $\mathcal{O}(L)$
   iv. Split/merge parents, in the worst case going up to the root: $\mathcal{O}(Mlog_M(n))$

   The total cost is then $lg(M)log_M(n) + 2L + Mlog_M(n)$.
   We can simplify this to a worst-case runtime $\mathcal{O}(L + Mlog_M(n))$ by combining constants and observing that $Mlog_M(n)$ dominates $lg(M)log_M(n)$. Note that in the average case, splits for any reasonably-sized B-tree are rare, so we can amortize the work of splitting over many operations.
   However, if we're using a B-tree, it's because what concerns us the most is the penalty of disk accesses. In that case, we might find it more useful to look at the worst-case number of disk lookup operations in the B-tree, which is $\mathcal{O}(log_M(n))$.

   (b) Look up the value of a key

   Solution:

   i. We must do a binary search on a node containing M pointers, which takes $\mathcal{O}(lg(M))$ time, once at each level of the tree.
   ii. There are $\mathcal{O}(logM(n))$ levels.
   iii. We must do a binary search on a leaf of L elements, which takes $\mathcal{O}(lg(L))$ time.
   iv. Putting it all together, a tight bound on the runtime is $\mathcal{O}(lg(M)log_M(n) + lg(L))$.

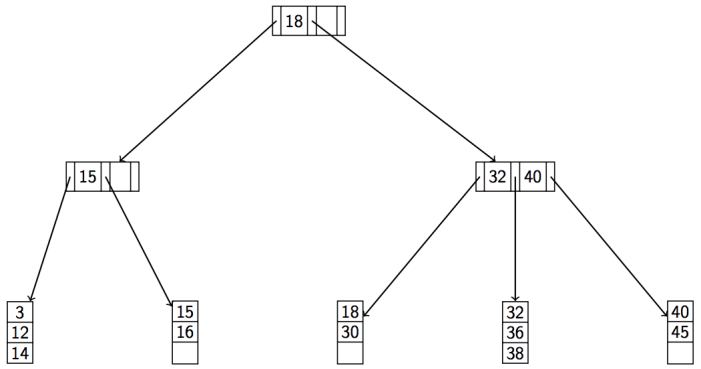
   (c) Delete a key-value pair

   Solution: See solution for inserting 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.

Solution:



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

Solution:

5 Memory

1. What are the two types of memory locality?

Solution: Spatial locality is memory that is physically close together in addresses. Temporal locality is the assumption that pages recently accessed will be accessed again.

2. Does this more benefit arrays or linked lists?

> **Solution:** This typically benefits arrays. In Java, array elements are forced to be stored together, enforcing spatial locality. Because the elements are stored together, arrays also benefit from temporal locality when iterating over them.

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

> **Solution:** Java can't page-align its memory allocation.

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?

> **Solution:** The internal array within the array-based queue is more likely to be contiguous in memory compared to the linked list implementation of an array. This means that when we access each element in the array, the surrounding parts of the array are going to be loaded into cache, speeding up future accesses.
> One thing to note is that the array-based queue won't necessarily automatically be faster then the linked-list-based one, depending on how exactly it's implemented.
> A standard queue implementation doesn't support the `iterator()` operation, and a standard array-list based queue implements either $\mathcal{O}(n)$ enqueue or dequeue.
> In that case, if we're forced to access every element by progressively dequeueing and re-enqueuing each element, iterating over a standard array-based queue would take $\mathcal{O}(n^2)$ time as opposed to the linked-list-based queue's $\mathcal{O}(n)$ time. In that case, the linked-list version is going to be far faster then the array-list version for even relatively smaller values of n.
> The only way we could have the array-based queue be consistently faster is if it supported $\mathcal{O}(1)$ enqueues and dequeues. (Doing this is actually possible, albeit slightly non-trivial.)

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();
    }
}
```

Solution: Temporal Locality. At each iteration, the specific string from the array is already loaded into the cache. When performing the next process `toUppercase()`, the content can just be loaded from cache, instead of disk or RAM.

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.

Solution: An internal node has M children, and therefore $M-1$ keys inside. We need a pointer to each child, so we get $Mt + (M-1)k$ bytes.
A leaf node is defined as having L key-value pairs, so the total size is $L(k+v)$ bytes.
Note: This is all assuming the node objects themselves contain no overhead. Java objects do have a certain amount of overhead associated with them, but many programming languages let you construct objects (or object-like structures) where their size in bytes is exactly the sum of the size of the fields, with no extra overhead. For the sake of simplicity, we will be assuming we are using those kinds of programming languages, and that our B-tree nodes have no extra memory overhead.

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.

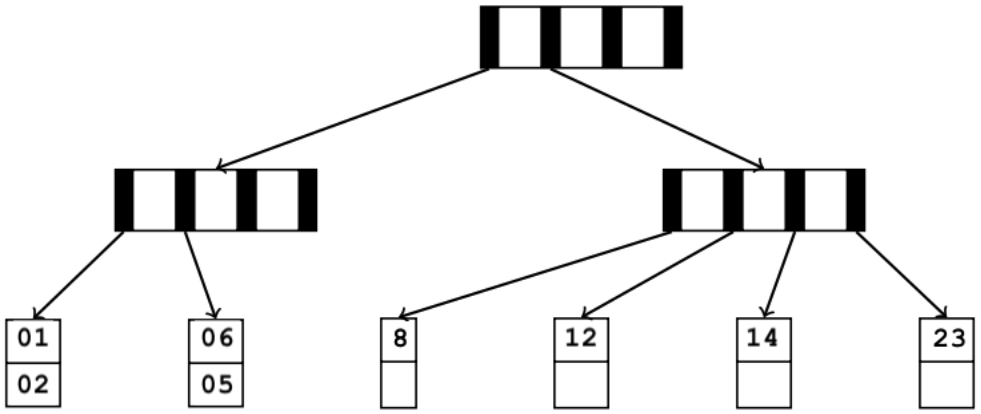
   Be sure to show your work. The equations you derived in the previous part will come in handy here.

   We want to pick the largest M and L that satisfy $Mt + (M-1)k \leq p$ and $L(k+v) \leq p$.
   We can start by computing L, since that's easier. We have $L(10+6)+2 \leq 130$, which simplifies into $L \leq \frac{128}{16}$.
   If we divide 128 by 16, we get about 8.0. L must be a whole number, but this works out anyway, so we know $L = 8$.
   We can do something similar for M. We can rearrange the inequality:

$$\begin{aligned} Mt + (M-1)k &\leq p \\ Mt + Mk - k &\leq p \\ M(t+k) &\leq p + k \\ M &\leq \frac{p+k}{t+k} \end{aligned}$$

   We plug in numbers, and get $M \leq \frac{130+10}{2+10}$. If we divide 140 by 12, we get about 11.66. So, we know $M = 11$.
   Our final answer: $M = 11$ and $L = 8$.

5. Consider the following "B-Tree":



(a) What are M and L?

Solution: $M = 4$; $L = 2$.

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

Solution:

i. 6 and 5 should be swapped in the second leaf

ii. The inner nodes are missing the signposts: left-most inner node should have 5 in the first entry, right-most inner node should have entries 12, 14, and 25.

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?

Solution: This is most likely due to spatial locality. When we iterate through a linked list, accessing the value at one particular index will load the next few elements into the cache, speeding up the overall time needed to access each element.
In contrast, each node in the linked list is likely loaded in a random part of memory – this means we likely must load each node into the cache, which slows down the overall runtime by some constant factor.

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?

Solution: If L is 12, and each key-value pair occupies $14 + 11 = 25$ bytes, we know each leaf node occupies at least $12 \times 25 = 300$ bytes.
If M is 10, we know each each branch node occupies at least $M \times 16 + (M-1) \times 14 = 160 + 126 = 286$ bytes.
This leads us to conclude that the page size is most likely 300 bytes 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.

   Solution: This is almost absolutely because the `SortedArrayDictionary` is implemented with an array, which has much better spatial locality, than how the `AvlTreeDictionary` is most likely implemented, with a series of linked AVL tree nodes. Since we know that iterating over an array is faster than iterating over a linked list, the reasoning behind a faster tree is similar and reasonable.

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.)

   Solution: Spatial locality can only be taken advantage of when iterating sequentially. With that, it's not surprising that because we have to jump from `i=100` to `i=50`, etc. until we find what we're looking for. If the array is so big that it spans over multiple pages, the locality that regular iteration takes advantage of is not available to jumping around with `get()`.

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:

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

   Solution: $M = 43$ and $L = 25$

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)

   Solution: The numerator of the fraction is always a number from 0 to $2^{30} - 1$ (after we do the mod operation). Dividing by 2^{20} moves the number into range 0 to $2^{10} - 1/2^{20}$. When we take the floor, we round the numbers down to the next integer, so the range of possible outputs becomes 0 to $2^{10} - 1$, i.e. exactly the indices for a 0-indexed table of size 2^{10}.

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.

   Solution: The keys $\{0, 1, 2, \ldots, 2^{20} - 1\}$ all hash to 0 (modding by 2^{30} doesn't affect small values, and the floor rounds any number less than 1 to 0). For $a = 1$ this function hashes contiguous sets of 2^{20} numbers into each bin. For other hash functions, it is harder to find this set, but every hash function has this problem: if the key-space is much larger than the size of the table, there must be a large number of values that all collide.

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?

> **Solution:** The exact number of collisions you see will depend on which values of a, x, y you check, but you should see between 0.1% and 0.2% of hash functions causing a collision.
> If the outputs of the hash function were truly random, we would see a collision with probability equal to $\frac{1}{\text{table size}}$, i.e. 1/1024 or about .1% of the time. So the output we're seeing as we change a is nearly as good as really random outputs.

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?)

Solution: A user will have a fixed set of keys they need to use. If these happen to be keys that all hash to the same place, the hash table will have very poor performance, and the user has no hope of fixing this (because they can't really change their keys). On the other hand, if we choose a random hash function, with high probability (say 99.9%) the fixed set of keys won't be a problem (though there is still a small chance of getting the poor performance).
Occasionally, we worry about an attacker intentionally giving us bad data to try to break our code. If an attacker knows your hash function, they can do what we did in the first part, and find a set of inputs that will slow down your hash table. On the other hand, if you're choosing a hash function randomly, there is no single set of inputs that can cause bad performance. Choosing a function randomly lets us "spread out" the bad behavior across inputs. Said a different way:

- With a single, fixed function for most inputs, things work great; but there are some very bad inputs on which things work terribly.
- If you choose a random hash function, every input has a very high probability of good performance, but on many inputs there is a very small chance of bad performance.

There are a lot of mathematical caveats here (e.g. you need a good set of hash functions to choose from, your choice of hash function needs to be random enough that it can't be predicted, etc.) but we don't have time to go into them. See http://jeffe.cs.illinois.edu/teaching/algorithms/notes/12-hashing.pdf for more information on this hashing scheme, and randomized hashing in general, including the formal mathematics.