Week 6 Tutorial Sheet

(Solutions)

Useful advice: The following solutions pertain to the theoretical problems given in the tutorial classes. You are strongly advised to attempt the problems thoroughly before looking at these solutions. Simply reading the solutions without thinking about the problems will rob you of the practice required to be able to solve complicated problems on your own. You will perform poorly on the exam if you simply attempt to memorise solutions to the tutorial problems. Thinking about a problem, even if you do not solve it will greatly increase your understanding of the underlying concepts. Solutions are typically not provided for Python implementation questions. In some cases, psuedocode may be provided where it illustrates a particular useful concept.

Assessed Preparation

Problem 1. A hash table can be used to store different types of elements such as integers, strings, tuples, or even arrays. Assume we want to use a hash table to store arrays of integers (i.e., each element is an array of integers). Consider the following potential hash functions for hashing the arrays of positive integers. Rank them in terms of quality and give a brief explanation of the problems with each of them. Assume that they are all taken mod m, where m is the table size.

- Return the first number in the array (e.g., if `array=[10,5,7,2]`, hash index will be `10%m`)
- Return a random number in the array (e.g., if `array=[10,5,7,2]`, hash index will be a randomly chosen element from the array mod m)
- Return the sum of the numbers in the array (e.g., if `array=[10,5,7,2]`, hash index will be `24%m`)

Give a better hash function than these three and explain why it is an improvement.

Solution

From best to worst:

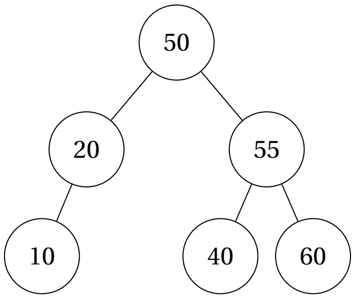
1. **Return the sum of the numbers in the array.** This is a decent hash function, as it accounts for all of the contents of the array. It is not amazing since it will produce identical hashes for permutations of the same elements (or arrays having the same sum), e.g., `array1=[10,5,7,2]`, `array2=[2,5,7,10]` and `array3=[9,6,7,2]` will all collide.
2. **Return the first number in the array** This is not very good since it only accounts for one element of the array, so any two arrays that begin with the same element will collide.
3. **Return a random number in the array**. This is not even a valid hash function since it might return a different value for the same array when used multiple times. Hash functions must be consistent, that is they must always produce the same value for the same key.

A better hash function would be

$$h(A) = A[0] + A[1]x + A[2]x^2 + \ldots + A[n]x^n,$$

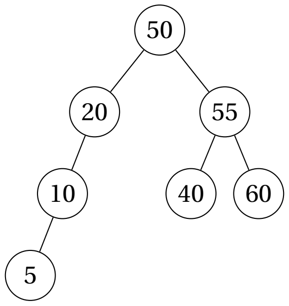
for some value of $x > 1$. This is better since if we use this hash function, all of the array is taken into account, and permuting the order of the elements is likely to change the value of the hash. We can make it even better by selecting $x > \max(A)$ and taking the sum mod p for a suitable prime p, which yields the standard polynomial hash function.

Problem 2. Insert 5 into the following AVL tree and show the rebalancing procedure step by step.

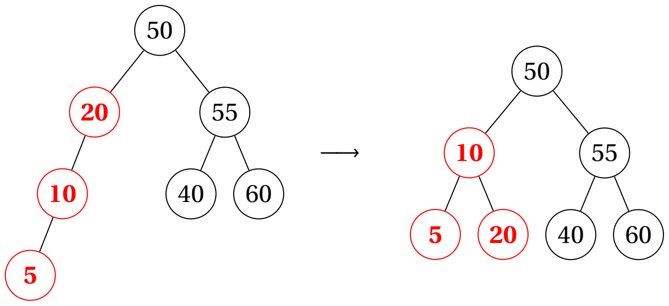


Solution

We insert 5 using the ordinary BST insertion procedure and we obtain

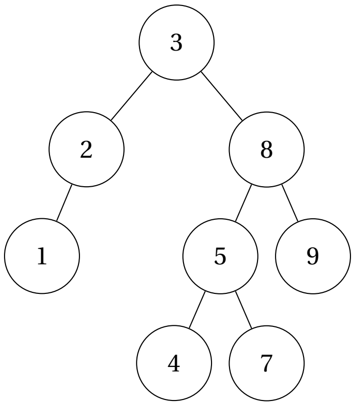


This tree is imbalanced, as the node containing 20 has a height 2 left subtree, and a height 0 right subtree, resulting in a balance factor of 2. To rectify this, we rotate the node 10:



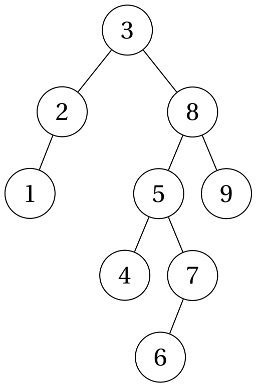
Tutorial Problems

Problem 3. Insert 6 into the following AVL tree and show the rebalancing procedure step by step.



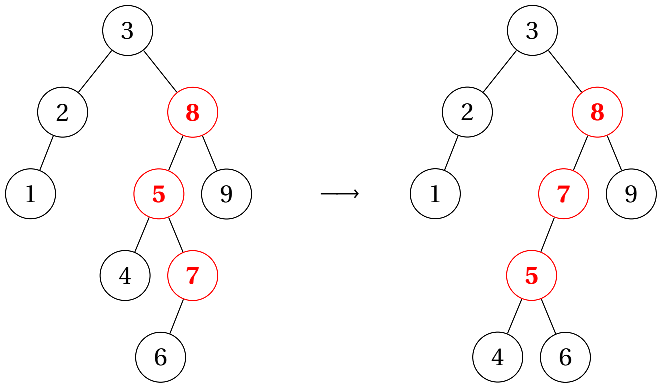
Solution

We insert 6 using the ordinary BST insertion procedure and we obtain the following tree.

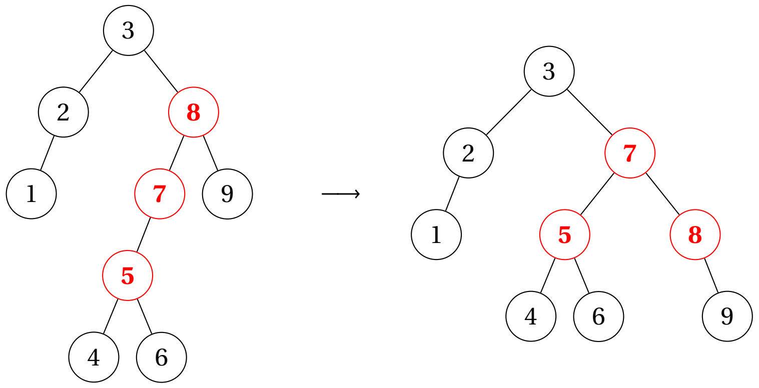


This tree is imbalanced at node 8 since it has a left subtree of height 3 and a right subtree of height 1, leading to a balance factor of 2. Note that the root node 3 is also imbalanced, but we always perform rotations on the lower levels first since this may fix the imbalances at higher levels.

Since node 8 has a taller left subtree and node 5 has a taller right subtree, this is a left-right imbalance so we must perform a double rotation on node 7 to correct it. We perform the first rotation to obtain:



We then perform the second rotation and are left with:



This is a balanced tree, so we are done.

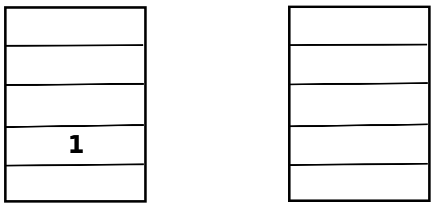
Problem 4. Show the steps taken by Cuckoo hashing when inserting the following sequence of keys using the hash function $f(k) = k \mod m$ for T_1 and $g(k) = ((2k+1) \mod 7) \mod m$ for T_2.

$$1, 3, 6, 8, 2, 7, 0, 5$$

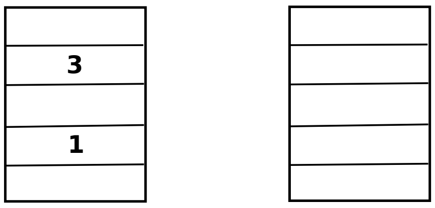
Assume that the table sizes are both $m = 5$.

Solution

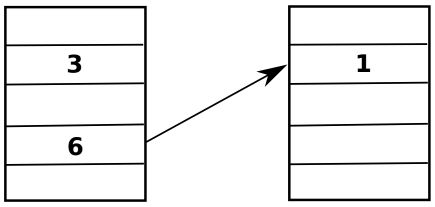
We show the steps of each insertion. Every time an element is kicked out and moved, we draw an arrow to show the location it moves to.



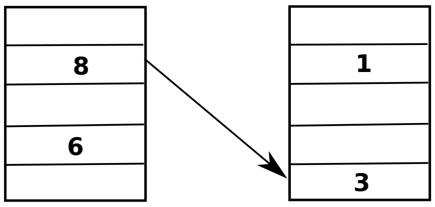
Insert 1



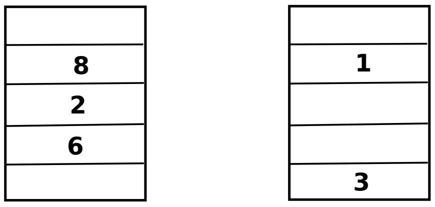
Insert 3



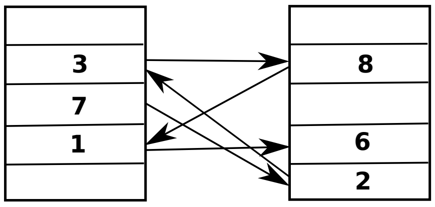
Insert 6



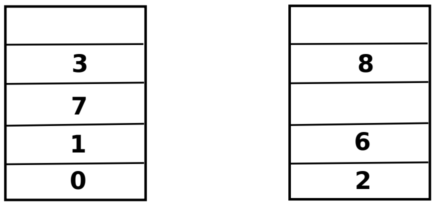
Insert 8



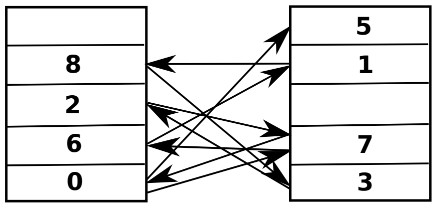
Insert 2



Insert 7



Insert 0



Insert 5

Problem 5. Recall the algorithm for deleting an element from a binary search tree. If that element has no children, we do nothing. If it has one child, we can simply remove it and move its child upwards. Otherwise, if the node has two children, we swap it with its *successor* and then delete it. This algorithm works because of the following fact: The successor of a node with two children has no left child. Prove this fact.

(a) First prove that the successor of a node x with two children must be contained in the right subtree of x

(b) Use this fact to prove that the successor of x can not have a left child

Solution

Recall that the successor of a node is the minimum value node that is greater than x. Consider a node x with two children x_l and x_r. First, we will show that the successor of x lies in the right subtree x_r of x. All elements in the left subtree of x have values $< x$ and hence can not be the successor of x. Suppose that x has a parent p. If $p_x < x$ then x is a right child and hence neither p_x nor its left subtree can contain the successor of x since every node v in the left subtree of p_x satisfies $v < p_x < x$. Otherwise x is a left child of p_x and hence $r_x < p$ and in particular $r_x < v$ for every v in the right subtree of p_x since $r_x < p_x < v$, therefore the successor of x is not a right descendant of p_x. This leaves the only possibility that the successor is in the subtree of r_x.

Suppose that the successor y of x had a left child y_l. By the BST property we have $y_l < y$, but since y_l is contained in the right subtree of x, we also have $x < y_l$. This means that $x < y_l < y$ which implies that y is not in fact the successor of x. This is a contradiction and hence y must have no left child.

Problem 6. Consider the following potential hash functions for hashing integers. Rank them in terms of quality and give a brief explanation. Assume that they are all taken mod m, where m is the table size.

- Return $x \mod 2^p$ for some prime p
- Return $(ax + b)$ for some positive a, b
- Return a random integer

Solution

Ranked from best to worst:

- Return $(ax+b)$ for some positive a, b. This hash is ok. It could be improved by taking the result modulo a prime number.
- Return $x \mod 2^p$ for some prime p. Note that, if p is small, taking modulo powers of two simply keeps the bottom p bits of the integer, which is bad. For p sufficiently large, this is just the same as just taking the value of x as the output of our hash function, which is fine if the keys are random, but not so great if they are not.
- Return a random integer. This does not hash the same key to the same place consistently, and so is not even a valid hash function.

Problem 7. Consider the following probing schemes and for each of them, explain whether they do or do not suffer from primary or secondary clustering.

(a) $h'(k,i) = (h'(k)+5i) \mod m$, where $h'(k)$ is a hash function

(b) $h'(k,i) = (h'(k)+i^{\frac{3}{2}}) \mod m$, where $h'(k)$ is a hash function

(c) $h'(k,i) = (h_1(k) \times (h_2(k)^i) \mod m$, where $h_1(k)$ and $h_2(k)$ are two hash functions

(d) $h'(k,i) = (h'(k)+2)^{i+1}) \mod m$, where $h'(k)$ is a hash function

(e) $h'(k,i) = (h_1(k) \times h_2(k)+i) \mod m$, where $h_1(k)$ and $h_2(k)$ are two hash functions

Solution

(a) This is just linear probing, but in fixed steps of 5 instead of fixed steps of 1. This means it has both primary and secondary clustering.

(b) This is almost quadratic probing, except instead of steps of size i^2 we have steps of size $i^{3/2}$. It will behave in a similar way to quadratic probing, namely that two keys having the same hash will have the same probe chain, since the step size does not depend on the key, but keys with different hashes will have non-overlapping chains, because the step size increases each step. So we do not have primary clustering, but we do have secondary.

(c) This is almost double hashing, but multiplying by $h_2(k)^i$ instead of adding $h_2(k) * i$. It will display neither primary nor secondary clustering.

(d) This hash function probes with ever increasing step sizes, in a similar way to quadratic probing, but the step sizes are increasing powers of a given number. Two items which have the same value for $h'(k,0)$ will have the same probe chains, so this hash displays secondary clustering. It is possible for items with different values for $h'(k,0)$ to have partially overlapping chains, for example when $h'(k_1)+2=3, h'(k_2)+2=9$. Then every second position in the probe chain of k_1 will overlap with an element from the chain from k_2. This is not as bad as primary clustering (and will occur very rarely).

(e) If we define a new hash function, $h_3(k,i) = h_1(k) * h_k(2) + i$, we notice that this is just linear probing. So it has both primary and secondary clustering.

Supplementary Problems

Problem 8. You are given a set of distinct keys $x_1, x_2, ..., x_n$.

(a) Design an algorithm that creates a binary search tree of minimal height containing those keys in $O(n\log(n))$ time.

(b) Prove that your algorithm produces a BST of height at most $\log(n)$.

(c) Prove that $O(n\log(n))$ is the fastest algorithm for this problem in the comparison model

Solution

(a) First we sort the keys, this takes $O(n\log(n))$. We take the median element of the sequence and insert it into a BST, then recursively do the same thing with the resulting left and right sublists. In the below code, we assume we have a *Node* class which contains a key, a left child, and a right child. We must sort the list before calling OPTIMAL_BST(x[1..n]).

```
function OPTIMAL_BST($x[1..n]$)
    if x is empty then return null
    mid = $\lfloor n/2 \rfloor$
    root = Node($x$[mid])
    root.left = OPTIMAL_BST($x$[1..mid-1])
    root.right = OPTIMAL_BST($x$[mid+1..$n$])
    return root
end function
```

(b) To prove that the height of the constructed tree is at most $\log(n)$, we write a recurrence for the height.

$$H(n) = 1 + H\left(\left\lceil \frac{n-1}{2} \right\rceil\right) \le 1 + H\left(\frac{n}{2}\right)$$

Using telescoping,

$$\begin{aligned} H(n) &\le 1 + \left[1 + H\left(\frac{n}{4}\right)\right], \\ &\le 2 + \left[1 + H\left(\frac{n}{8}\right)\right], \\ &\le \ldots, \\ &\le k + H\left(\frac{n}{2^k}\right). \end{aligned}$$

We know that $H(1) = 0$, so setting $k = \log(n)$, we obtain

$$H(n) \le \log(n) + H(1) = \log(n)$$

as required.

(c) If we could construct any BST from n keys in faster than $O(n\log(n))$ time, we could then do an in-order traversal of the resulting BST to obtain the keys in sorted order. This means we could sort faster than $O(n\log(n))$, but $O(n\log(n))$ is a lower bound for comparison-based sorting, so such an algorithm is impossible.

Problem 9. We know that when inserting an element into a binary search tree, there is only one valid place to put that item in the tree. Let's prove this fact rigorously. Let T be a binary search tree and let x be an integer not contained in T. Prove that exactly one of the following statements is true:

- The successor of x has no left child
- The predecessor of x has no right child

That is, prove that it can not be the case that both of these statements are true simultaneously, nor that both of these statements are false simultaneously. This implies that there is a unique insertion point for x since upon insertion x must either become the left child of its successor or the right child of its predecessor.

Solution

Let the predecessor of x be p, and the successor be s. We know that $p < x < s$, and that there is no key k in the tree such that $p < k < x$ or $x < k < s$. We claim that p and s can not have a common ancestor. Suppose for contradiction that p and s have a common ancestor r. Then we know that $p < r < s$. But that means that either $p < r < x < s$ or $p < x < r < s$. But by the definition of successor and predecessor, both of those are impossible. So we have a contradiction, meaning that either s is an ancestor of p (in which case s has a left child, since p will be in its left subtree), or p is an ancestor of s (in which case p has a right child, since s will be in its right subtree). So we know that at least one of the statements is true.

We still need to prove that at most one of the statements is true. Suppose for contradiction that both are true, i.e. that s has a left child s_l and p has as right child p_r. We know that s_l and p_r are between s and p in value. We also know that x is between s and p in value, and that nothing is between s and x, and nothing is between x and p. This is a contradiction, so both statements cannot be true.

Since at least one is true, and both cannot be true, exactly one must be true.

Problem 10. Implement hashtables that use chaining, linear probing, and quadratic probing for collision resolution. You may reuse hashtable code from previous units if you have it. Compare these hashtables on randomly generated integers and compare their performance. Try adjusting the hash function, the table size, and the number of keys inserted and see how this affects the results.

Problem 11. In lectures we claimed that AVL trees are good because the balance property guarantees that the tree always has height $O(\log(n))$. Let's prove this.

(a) Write a recurrence relation for $n(h)$, the **minimum** number of nodes in an AVL tree of height h. [Hint: It should be related to the Fibonacci numbers]

(b) Find an exact solution to this recurrence relationship in terms of the Fibonacci numbers[1]. [Hint: Compare the sequence with the Fibonacci sequence, find a pattern, and then prove that your pattern is right using induction]

(c) Prove using induction that the Fibonacci numbers satisfy $F(n) \geq 1.5^{n-1}$ for all $n \geq 0$

(d) Using (a), (b), and (c), prove that a valid AVL tree with n elements has height at most $O(\log(n))$

Solution

Consider an AVL tree of height h whose root node has the two subtrees L and R. Suppose that L and R have the same height. We could remove the nodes from the lowest level of one of the two subtrees, making them differ in height by one and it would still be a valid AVL tree with the same height. Therefore an AVL tree with the fewest possible nodes must have L and R differing in height by one, hence they must have heights $h-1$ and $h-2$. Lastly, note that L and R must also be AVL trees with the fewest possible nodes. Therefore the minimum number of nodes in an AVL tree satisfies the following recurrence relationship:

$$n(h) = \begin{cases} 1 & \text{if } h = 0, \\ 2 & \text{if } h = 1, \\ 1 + n(h-1) + n(h-2) & \text{if } h > 0. \end{cases}$$

This looks suspiciously similar to the Fibonacci sequence, so lets compare them:

[1]For this problem, index the Fibonacci numbers from zero with the base case $F(0) = F(1) = 1$.

h	0	1	2	3	4	5	6	7	8
$F(h)$	1	1	2	3	5	8	13	21	34
$n(h)$	1	2	4	7	12	20	33	54	88

This table strongly suggests the pattern

$$n(h) = F(h+2) - 1$$

Lets prove this by induction. We have $n(0) = F(2) - 1 = 1$ and $n(1) = F(3) - 1 = 2$ as required. Now suppose $n(h) = F(h+2) - 1$ for all $h \leq k$ for some value of k. We need to show that $n(k+1) = F(k+3) - 1$. From the recurrence, we have

$$n(k+1) = 1 + n(k) + n(k-1).$$

Invoking the inductive hypothesis, we can write

$$\begin{aligned} n(k+1) &= 1 + F(k+2) - 1 + F(k+1) - 1, \\ &= F(k+2) + F(k+1) - 1 \end{aligned}$$

By the definition of Fibonacci numbers, $F(k+2) + F(k+1) = F(k+3)$, hence we have

$$\begin{aligned} n(k+1) &= F(k+2) + F(k+1) - 1, \\ &= F(k+3) - 1, \end{aligned}$$

as required. Hence by strong induction on k, we have $n(h) = F(h+2) - 1$ for all h.

Next, we prove that the Fibonacci numbers have the desired bound. In the base case, we have

$$F(0) = 1 \geq 1.5^{-1}, \qquad F(1) = 1 \geq 1.5^{0}.$$

Now, suppose that $F(n-1) \geq 1.5^{n-2}$ and $F(n) \geq 1.5^{n-1}$. We need to prove that $F(n+1) \geq 1.5^{n}$. From the definition of Fibonacci numbers, we have

$$F(n+1) = F(n) + F(n-1),$$

using using the inductive hypothesis, we can write the bound

$$\begin{aligned} F(n+1) &\geq 1.5^{n-1} + 1.5^{n-2}, \\ &= 1.5 \times 1.5^{n-2} + 1.5^{n-2}, \\ &= (1.5+1)1.5^{n-2}, \\ &\geq 2.25 \times 1.5^{n-2}, \\ &= (1.5)^2 1.5^{n-2}, \\ &= 1.5^{n}. \end{aligned}$$

Therefore by induction on n, we have $F(n) \geq 1.5^{n-1}$.

Finally, combining the above, we have $n(h) = F(h+2) - 1 \geq 1.5^{h+1} - 1$. Rearranging, we find $1.5^{h+1} \leq n(h) + 1$, and hence

$$h + 1 \leq \log_{1.5}(n(h) + 1) \qquad \Longrightarrow \qquad h = O(\log(n)),$$

as required.

Problem 12. (Advanced) Consider a hashtable implementing linear probing, with size $m = 17$ using the hash function

$$h(k) = (7k + 11) \mod m.$$

Give a sequence of keys to insert into the table that will cause its worst-case behaviour.

Solution

We want a sequence of keys that will all hash to the same slot, which will cause linear probing to probe the entire cluster every time. Let's design a set of keys that all hash to slot 0. We want

$$7k + 11 \equiv 0 \mod 17.$$

Adding 6 to both sides, this is equivalent to

$$7k \equiv 6 \mod 17.$$

Now we want to divide out the 7 to get an expression for k. To do so, we need to multiply by some number x such that $7x \equiv 1 \mod 17$ (in other words, we need the modular multiplicative inverse of $7 \mod 17$). We can simply find this by trial and error. Looking at multiples of 7, we find

$$\begin{aligned}
1 \times 7 = 7 \equiv 7 \mod 17, \\
2 \times 7 = 14 \equiv 14 \mod 17, \\
3 \times 7 = 21 \equiv 4 \mod 17, \\
4 \times 7 = 28 \equiv 11 \mod 17, \\
5 \times 7 = 35 \equiv 1 \mod 17.
\end{aligned}$$

Therefore we find that 5 is the inverse we are looking for. We therefore can write

$$5 \times 7k \equiv 5 \times 6 \mod 17,$$

from which we get

$$k \equiv 13 \mod 17.$$

Therefore, a worst-case sequence of keys would be

$$k = 13 + 17i, \qquad i \geq 0,$$

i.e. $k = 13, 30, 47, 64, 81, \ldots.$

Problem 13. (Advanced) Suppose that we insert n keys into a hashtable with m slots using a totally random hash function. What is the expected number of pairs of colliding elements? The pairs (k, k') and (k', k) are considered the same and should not be counted twice.

Solution

First, we define the following.

$$I_{i,j} = \begin{cases} 1 & \text{if } h(k_i) = h(k_j), \\ 0 & \text{otherwise.} \end{cases}$$

We call this an *indicator function* on the condition $h(k_i) = h(k_j)$. For each item k_i, the number of elements it collides with is given by

$$\text{\# collisions with } k_i = \sum_{j=1}^{i-1} I_{i,j}.$$

The total number of collisions is therefore

$$\text{\# collisions} = \sum_{i=1}^{n} \sum_{j=1}^{i-1} I_{i,j}$$

So, by linearity of expectation, the expected number of collisions is

$$\mathrm{E}\left[\sum_{i=1}^{n}\sum_{j=1}^{i-1} I_{i,j}\right] = \sum_{i=1}^{n}\sum_{j=1}^{i-1} \mathrm{E}\big[I_{i,j}\big].$$

We have by the definition of expectation

$$\mathrm{E}\big[I_{i,j}\big] = 1 \times \Pr\big[h(k_i) = h(k_j)\big] + 0 \times \Pr\big[h(k_i) \neq h(k_j)\big] = \Pr\big[h(k_i) = h(k_j)\big].$$

Since the hash is totally random,

$$\mathrm{E}\big[I_{i,j}\big] = \Pr\big[h(k_i) = h(k_j)\big] = \frac{1}{m}.$$

Therefore the expected number of colliding pairs is

$$\sum_{i=1}^{n}\sum_{j=1}^{i-1} \frac{1}{m} = \binom{n}{2}\frac{1}{m} = \frac{n(n-1)}{2m}.$$