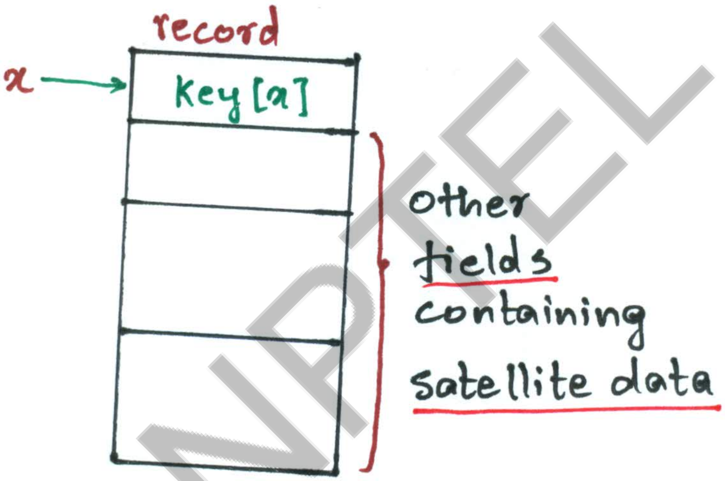
Week 5. Lecture Notes

Topics: Hash Function
Open Addressing
Universal Hashing
Perfect Hashing
Binary Search Tree (BST) Sort

Symbol Table Problem

Symbol table T holding n records:



How should the data structure T be organized?

Operations on T:

- INSERT (T, x)
- DELETE (T, x)
- SEARCH (T, K)

Direct-access table

IDEA: Suppose that the set of keys is $K \subseteq \{0, 1, \ldots, m-1\}$, and the keys are distinct.

Setup an array $T[0, \ldots, m-1]$:

$$T[k] = \begin{cases} x, \text{ if } x \in K \text{ and } key[x] = k; \\ \text{NIL, otherwise.} \end{cases}$$

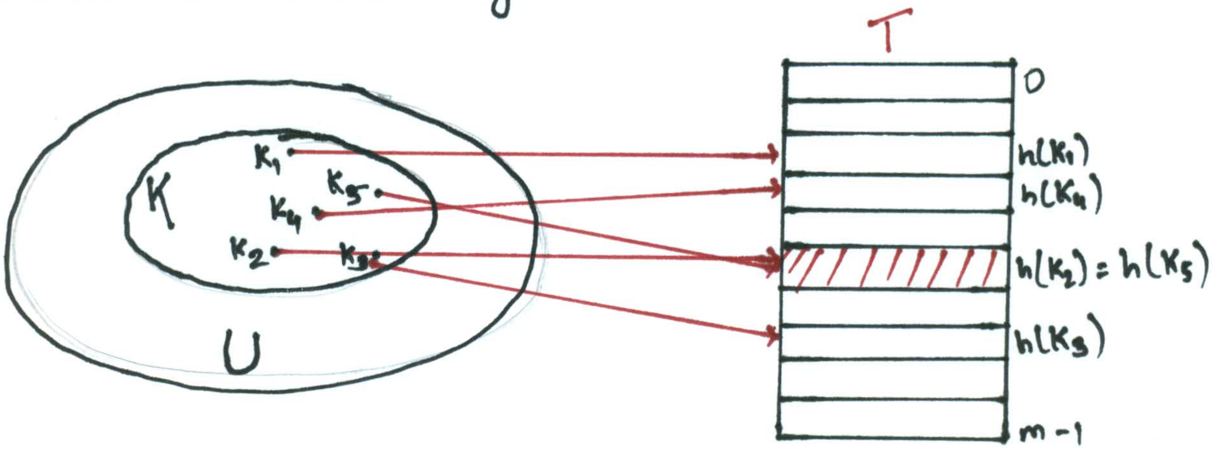
Then the operations take $\Theta(1)$ time.

PROBLEM: The range of keys can be large:

- 64 bit numbers (which represent 64,446,744, 073,709,616 different keys),
- character strings (even larger)

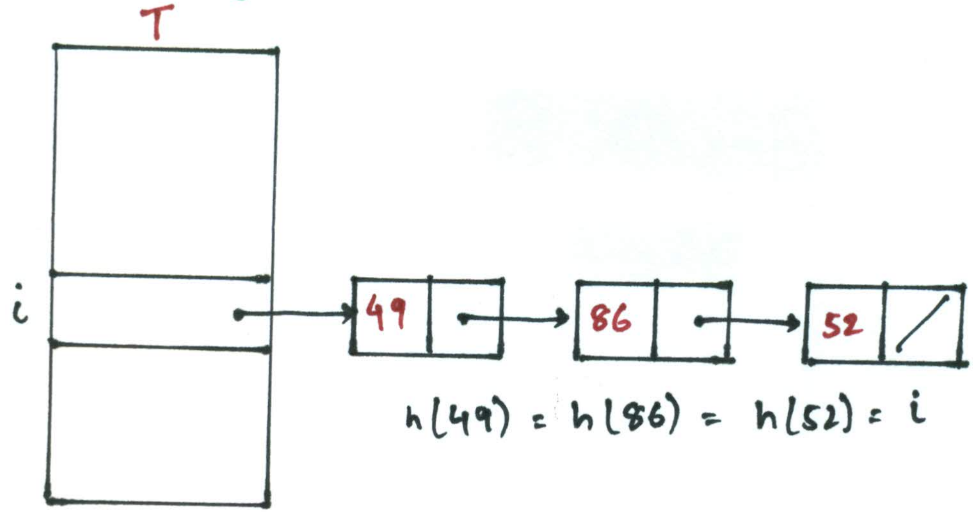
Hash functions

Solution: Use a hash function h to map the universe U of all keys into $\{0, 1, \ldots, m-1\}$:



When a record to be inserted maps to an already occupied slot in T, a collision occurs.

Resolving Collisions by Chaining



$h(49) = h(86) = h(52) = i$

Analysis of Chaining

We make the assumption of simple uniform hashing:

- Each key $k \in K$ of keys is equally likely to be hashed to any slot of table T, independent of where other keys are hashed.

Let n be the number of keys in the table, and let m be the number of slots.

Define the load factor of T to be

$$\alpha = n/m$$

= average number of keys per slot.

Search Cost

Expected time to search for a record with a given key $= \Theta(\underset{\text{apply hash function and access slot}}{1} + \underset{\text{search the list.}}{\alpha})$

- Expected search time $= \Theta(1)$ if $\alpha = O(1)$, or equivalently, if $n = O(m)$

Choosing a hash function

The assumption of simple uniform hashing is hard to guarantee, but several common techniques tend to work well in practice as long as their deficiencies can be avoided.

Desirata:

- A good hash function should distribute the keys uniformly into the slots of the table.
- Regularity in the key distribution should not affect this uniformity.

Division Method

Assume all keys are integers, and define

$$h(k) = k \bmod m$$

Deficiency: Don't pick an m that has a small divisor d. A preponderance of keys that are congruent modulo d can adversely affect uniformity.

Extreme deficiency: If $m = 2^r$, then the hash doesnot even depend on all bits of k:

- If

$$k = 1011000111\underbrace{011010_2}_{h(k)} \text{ and } r = 6$$

then

$$h(k) = 011010_2$$

Division Method (Continued)

$$h(k) = k \bmod m$$

Pick m to be a prime not too close to a power of 2 or 10 and not otherwise used prominently in the computing environment.

Annoyance:

Sometimes, making the table size a prime is inconvenient

But, this method is popular, although the next method we'll see is usually superior

Multiplication Method

Assume that all keys are integers, $m = 2^r$ and our computer has w-bit words. Define

$$h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w-r)$$

where rsh is the "bit-wise right-shift" operator and A is an odd integer in the range $2^{w-1} < A < 2^w$.

- Don't pick A too close to 2^w
- Multiplication modulo 2^w is fast
- The rsh operator is fast.

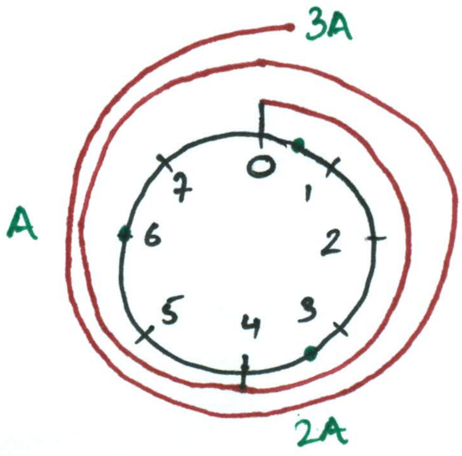
Multiplication method example

$$h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w-r)$$

Suppose that $m = 8 = 2^3$ and that our computer has $w = 7$- bit words.

$$\begin{array}{rl} 1011001 & = A \\ 1101011 & = k \\ \hline 1001010\,\underbrace{011}_{h(k)}\,0011 & \end{array}$$

Modular wheel:



Dot product method

Randomized Strategy:

Let m be prime. Decompose key K into $r+1$ digits, each with value in the set $\{0,1,\ldots,m-1\}$. That is, let $K = \langle K_0, K_1, \ldots, K_{m-1} \rangle$, where $0 \leq K_i < m$.

Pick $a = \langle a_0, a_1, \ldots, a_{m-1} \rangle$ where each a_i is chosen randomly from $\{0,1,\ldots,m-1\}$.

Define: $h_a(K) = \sum_{i=0}^{r} a_i K_i \bmod m$

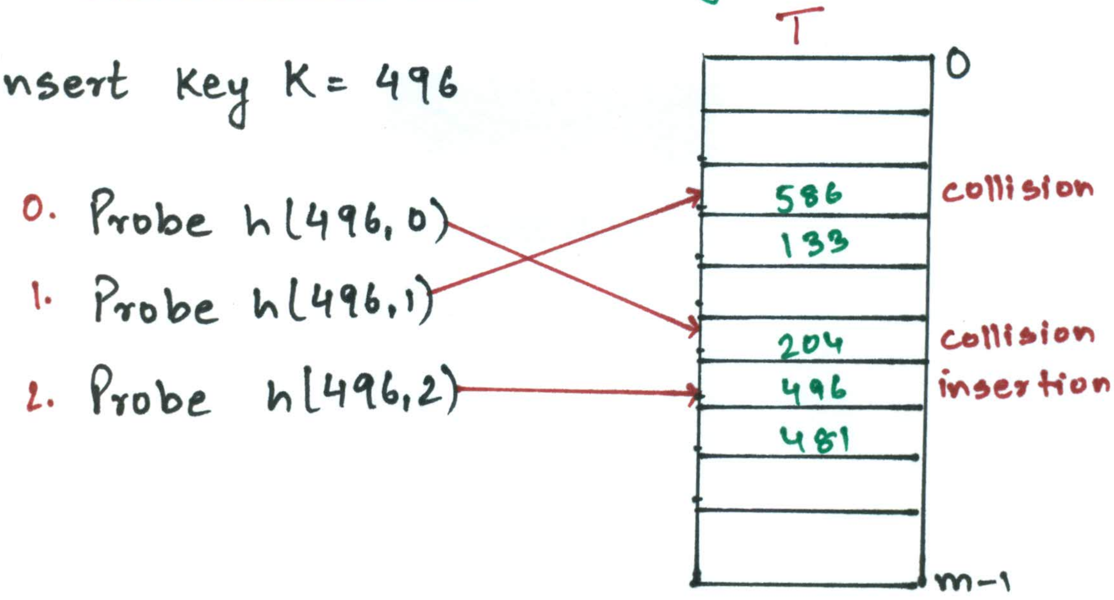
Resolving collisions by open addressing

No storage is used outside the hash table itself.

- Insertion systematically probes the table until an empty slot is found
- The hash function depends on both the key and probe number
  $h: U \times \{0,1,\ldots,m-1\} \rightarrow \{0,1,\ldots,m-\}$.
- The probe sequence $\langle h(K,0), h(K,1), \ldots, h(K,m-1) \rangle$ should be a permutation of $\{0,1,\ldots,m-1\}$.
- The table may fill up, and deletion is difficult (but not impossible)

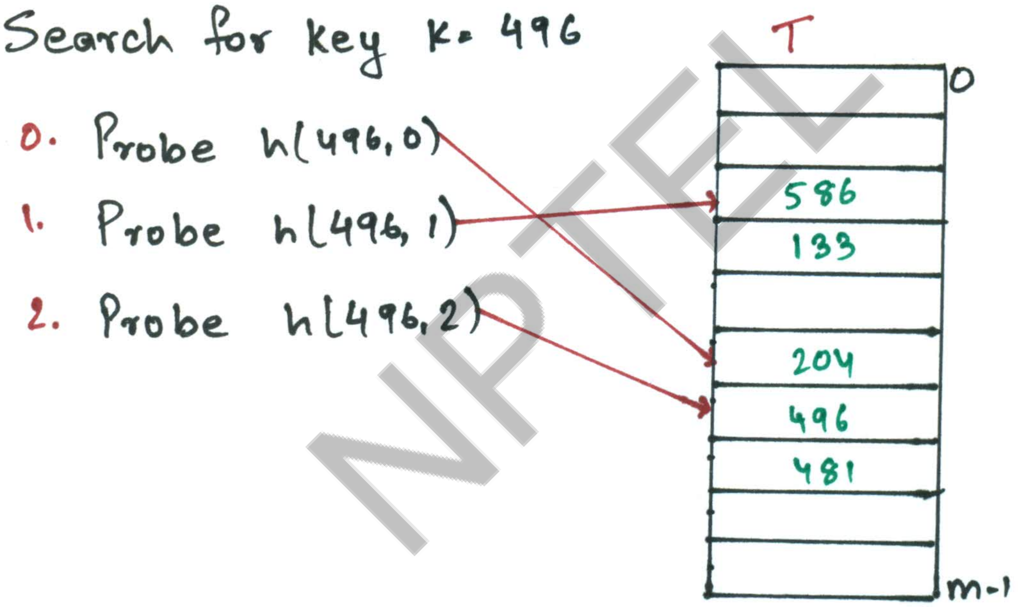
Example of open addressing

- Insert key K = 496



0. Probe h(496,0)
1. Probe h(496,1)
2. Probe h(496,2)

- Search for key K = 496



0. Probe h(496,0)
1. Probe h(496, 1)
2. Probe h(496,2)

Search uses the same probe sequence, terminating successfully if it finds the key an unsuccessfully if it encounters an empty slot.

Probing Strategies

Linear probing:

Given an ordinary hash function $h'(K)$, linear probing uses the hash function

$$h(K, i) = (h'(K) + i) \bmod m$$

This method, though simple, suffers from primary clustering, where long runs of occupied slots build up, clustering the average search time. Moreover, the long runs of occupied slots tends to get longer.

Double hashing

Given two ordinary hash functions $h_1(K)$ and $h_2(K)$, double hashing uses the hash function

$$h(K, i) = (h_1(K) + i h_2(K)) \bmod m$$

This method generally produces excellent results, but $h_2(K)$ must be relatively prime to m. One way is to make m a power of 2 and design $h_2(K)$ to produce only odd numbers.

Analysis of open addressing

We make the assumption of uniform hashing:

Each key is equally likely to have any one of the m! permutations as its probe sequence.

Theorem:

Given an open-addressed hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is atmost $1/(1-\alpha)$

Proof:

- At least one probe is necessary, always.
- With probability n/m, the first probe hits an occupied slot and a second probe is necessary
- With probability $(n-1)/(m-1)$, the second probe hits an occupied slot and a third probe is necessary.
- With probability $(n-2)/(m-2)$, the third probe hits an occupied slot, etc..

Observe that $\frac{n-i}{m-i} < \frac{n}{m} = \alpha$ for $i = 1, 2, \ldots, n$

Therefore, expected number of probes is:

$$1 + \frac{n}{m}\left(1 + \frac{n-1}{m-1}\left(1 + \frac{n-2}{m-1}\left(\cdots\left(1 + \frac{1}{m-n+1}\right)\cdots\right)\right)\right)$$

$$\leq 1 + \alpha\left(1+\alpha\left(1+\alpha\left(\cdots\left(1+\alpha\right)\cdots\right)\right)\right)$$

$$\leq 1 + \alpha + \alpha^2 + \alpha^3 + \ldots$$

$$= \sum_{i=0}^{\infty} \alpha^i$$

$$= \frac{1}{1-\alpha}$$

Implifications of the theorem

- If α is constant, then accessing an open-addressed hash table takes constant time.

- If the table is half-full, then the expected number of probes is

$$1/(1-0.5) = 2$$

- If the table is 90% full, then the expected number of probes is

$$1/(1-0.9) = 10$$

A weakness of hashing

Problem: For any hash function h, a set of keys exists that can cause the average access time of a hash table to skyrocket.

An adversary can pick all keys from $\{K \in U : h(K) = i\}$ for some slot i.

IDEA: Choose the hash function at random, independently of the keys.

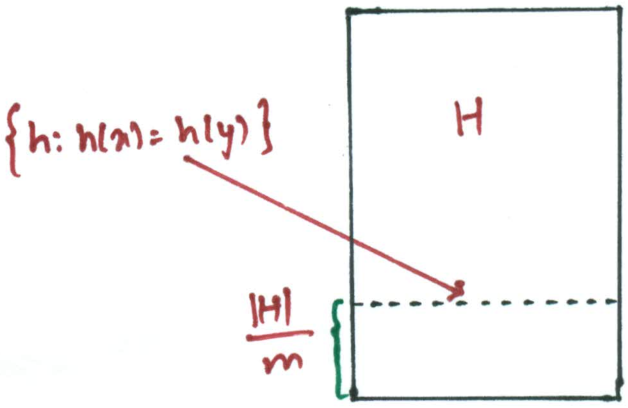
- Even if an adversary can see your code, he or she cannot find a bad set of keys since he or she doesnot know exactly which hash function will be chosen.

Universal Hashing

Definition: Let U be a universe of keys, and let H be a finite collection of hash functions, each mapping U to $\{0, 1, \ldots, m-1\}$. We say H is universal if for all $x, y \in U$ where $x \neq y$, we have

$$|\{h \in H : h(x) = h(y)\}| = |H|/m$$

That is, the chance of a collision between x and y is $1/m$ if we choose h randomly from H



Universality is good

Theorem:

Let h be a hash function chosen (uniformly) at random from a universal set H of hash functions. Suppose h is used to hash n arbitrary keys into the m slots of a table T. Then, for a given key x, we have

$$E[\#\text{collisions with } x] < n/m$$

Proof:

Let C_x be the random variable denoting the total number of collisions of keys in T with x and let

$$C_{xy} = \begin{cases} 1, & \text{if } h(x) = h(y); \\ 0, & \text{otherwise.} \end{cases}$$

Note: $E[C_{xy}] = 1/m$ and $C_x = \sum_{y \in T-\{x\}} C_{xy}$

So,

$$E[C_x] = E\left[\sum_{y \in T-\{x\}} C_{xy}\right] \quad \cdot \text{ Take expectations of both sides}$$

$$= \sum_{y \in T-\{x\}} E[C_{xy}] \quad \cdot \text{ Linearity of expectation}$$

$$= \sum_{y \in T-\{x\}} 1/m \quad \cdot \; E[C_{xy}] = 1/m$$

$$= \frac{n-1}{m}. \quad \cdot \text{ Algebra}$$

Constructing a set of universal hash functions

Let m be prime. Decompose key K into $r+1$ digits, each with value in the set $\{0,1,\ldots,m-1\}$. That is, let $K = \langle k_0, k_1, \ldots, k_r \rangle$, where $0 \le k_i < m$.

Randomized Strategy:

Pick $a = \langle a_0, a_1, \ldots, a_r \rangle$ where each a_i is chosen randomly from $\{0,1,\ldots,m-1\}$

Define $h_a(K) = \sum_{i=0}^{r} a_i k_i \bmod m$ (Dot product modulo m)

How big is $H = \{h_a\}$? $\boxed{|H| = m^{r+1}}$ ← REMEMBER THIS

Universality of dot-product hash functions

Theorem: The set $|H| = \{h_a\}$ is universal.

Proof: Suppose that $x = \langle x_0, x_1, \ldots, x_r \rangle$ and $y = \langle y_0, y_1, \ldots, y_r \rangle$ be distinct keys. Thus, they differ in at least one digit position, say position 0.

For how many $h_a \in H$ do x and y collide?

We must have $h_a(x) = h_a(y)$, which implies that

$$\sum_{i=0}^{r} a_i x_i \equiv \sum_{i=0}^{r} a_i y_i \pmod{m}$$

Equivalently, we have

$$\sum_{i=0}^{r} a_i(x_i-y_i) \equiv 0 \pmod m$$

$$\Rightarrow a_0(x_0-y_0) + \sum_{i=1}^{r} a_i(x_i-y_i) \equiv 0 \pmod m$$

$$\Rightarrow a_0(x_0-y_0) \equiv -\sum_{i=1}^{r} a_i(x_i-y_i) \pmod m$$

A fact from Number Theory

Theorem: Let m be prime. for any $z \in Z_m$ such that $z \neq 0$, there exists a unique $z' \in Z_m$ such that

$$z \cdot z^{-1} \equiv 1 \pmod m$$

Example: $m = 7$

z	1	2	3	4	5	6
z^{-1}	1	4	5	2	3	6

Proof continued:

We have $a_0(x_0-y_0) \equiv -\sum_{i=1}^{r} a_i(x_i-y_i) \pmod m$

Since $x_0 \neq y_0$, an inverse $(x_0-y_0)^{-1}$ must exist, so

$$a_0 \equiv \left(-\sum_{i=1}^{r} a_i(x_i-y_i)\right) \cdot (x_0-y_0)^{-1} \pmod m$$

Thus for any choices of $a_1, a_2, \ldots, a_r$, exactly one choice of a_0 causes x and y to collide.

Proof completed:

Q: How many h_a's cause x and y to collide?

A: There are m choices for each $a_1, a_2, \dots, a_r$, but once these are chosen, exactly one choice for a_0 causes x and y to collide, namely

$$a_0 \equiv \left(\left(-\sum_{i=1}^{r} a_i(x_i-y_i)\right)\cdot(x_0-y_0)^{-1}\right) \bmod m$$

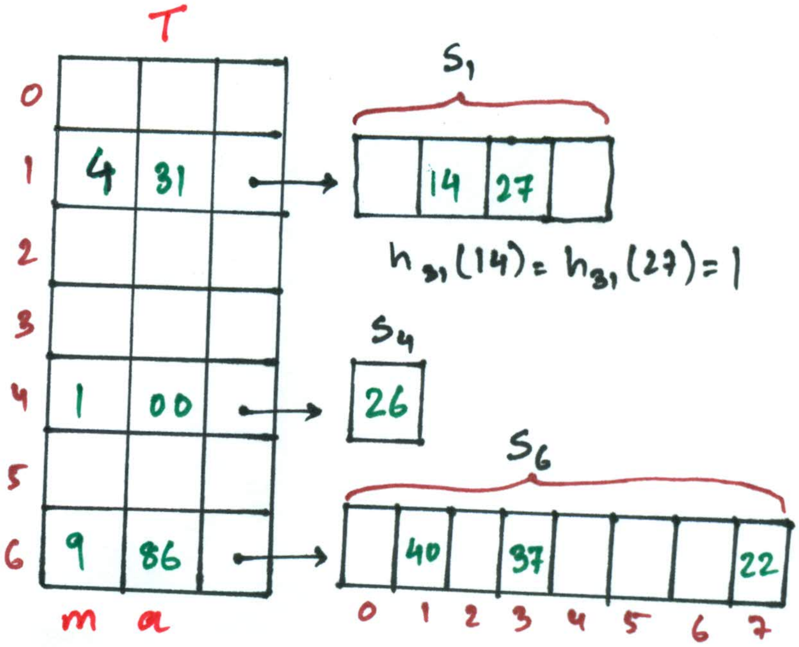
Thus the number of h_a's that cause x and y to collide is $m^r \cdot 1 = m^r = |H|/m$.

Perfect Hashing

Given a set of n keys, construct a static hash table of size $m = O(n)$ such that SEARCH takes $\Theta(1)$ time in the worst case.

IDEA:
Two-level scheme with universal hashing at both levels.

No collisions at level 2!



Collisions at level 2

Theorem:

Let H be a class of universal hash functions for a table of size $m = n^2$. Then if we use a random $h \in H$ to hash n keys into the table, the expected number of collisions is at most $1/2$.

Proof: By the definition of universality, the probability that 2 given keys in the table collide under h is $\frac{1}{m} = \frac{1}{n^2}$. Since there are $\binom{n}{2}$ pairs of keys that can possibly collide, the expected number of collisions is

$$\binom{n}{2} \cdot \frac{1}{n^2} = \frac{n(n-1)}{2} \cdot \frac{1}{n^2} < \frac{1}{2}.$$

No collisions at level 2

Corollary: The probability of no collisions is at least $1/2$.

Proof: Markov's inequality says that for any non-negative random variable X, we have

$$\Pr\{X \geq t\} \leq E[X]/t$$

Applying this inequality with $t = 1$, we find that the probability of 1 or more collisions is at most $1/2$

Thus, just by testing random hash functions in H, we will quickly find one that works.

Analysis of Storage

For the level-1 hash table T, choose $m=n$, and let n_i be random variable for number of keys that hash to slot i in T. By using n_i^2 slots for the level-2 hash table S_i, the expected total storage required for the two-level scheme is therefore

$$E\left[\sum_{i=1}^{m} \Theta(n_i^2)\right] = \Theta(n)$$

Since the analysis is identical to the analysis from recitation of the expected running time of bucket sort.

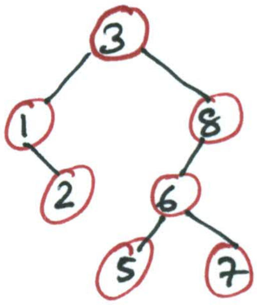
For a probability bound, apply Markov.

Binary-Search-Tree Sort

```
T ← ϕ      ▸ Create an empty BST
for i=1 to n
    do TREE-INSERT (T, A[i])
Perform an inorder tree walk of T.
```

Example:

A = [3 1 8 2 6 7 5]

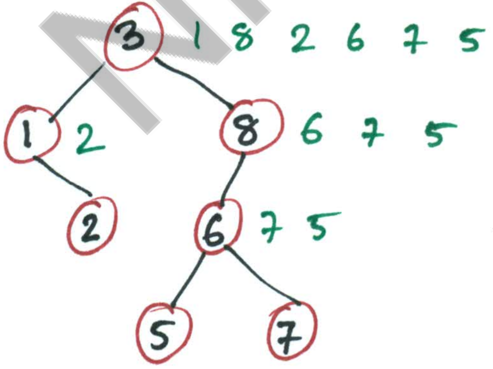


* Tree-walk time = $O(n)$

but how long does it take to build the BST?

Analysis of BST sort

BST sort performs the same comparisions as quicksort but in a different order



The ~~exposed~~ expected time to build the tree is asymptotically the same as the running time of quicksort.

Node depth

The depth of a node = the number of comparisions made during TREE- INSERT

Assuming all input permutations are equally-likely, we have

average node depth

$$= \frac{1}{n} E\left[\sum_{i=1}^{n} (\#\text{ comparisons to insert node } i)\right]$$

$$= \frac{1}{n} O(n\lg n) \qquad \text{(quick sort analysis)}$$

$$= O(\lg n)$$