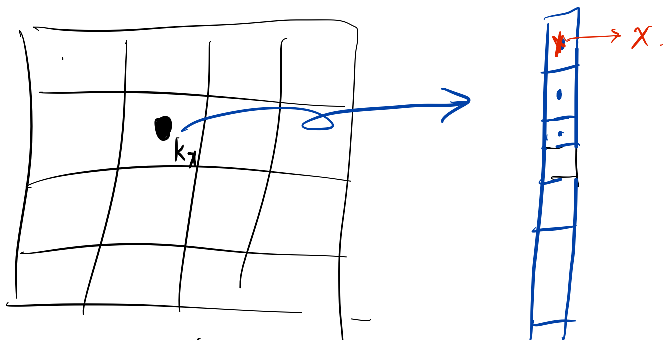
## Insertion Step of Hashing.

Time complexity of the process is usually a constant or linear.

① input $x$ → $\boxed{\text{Keys Generation}}$ → The key $K_x$

②

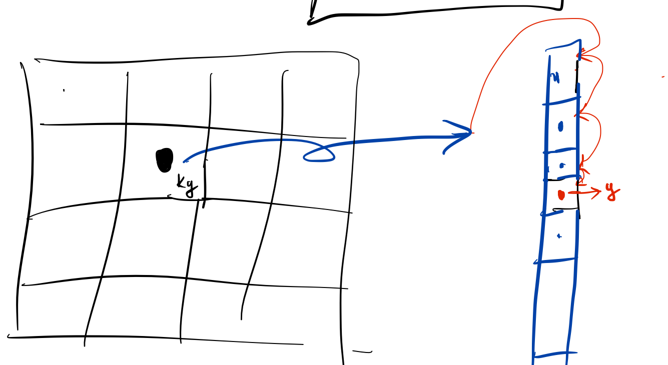



Hash Table

## Searching Step.

① target $y$ → $\boxed{\text{Keys Generation}}$ → The key $K_y$.

②




Hash Table

CSCI 3104: Algorithms
Spring 2022
Recitation #11 - Hashing & Balanced Binary Trees

**SOLUTIONS**

## Problem 1

For this problem, we consider using *balanced binary trees* as dictionaries.

a. What makes a tree a balanced binary tree?

A tree is a balanced binary tree if...

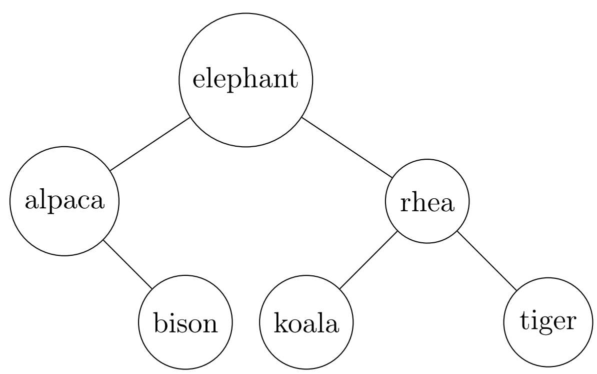
- each vertex has at most 2 children
- each vertex's children are all balanced binary trees
- the difference in height between any vertex's two child subtrees is at most 1

(source: JournalDev)

b. Arrange the following word list into an (alphabetically) sorted balanced binary tree (There are several possibilities, see how many you can find):

bison, tiger, elephant, alpaca, rhea, koala

One possible balanced binary tree is




c. List some of the benefits of using a balanced binary tree. List some of the drawbacks. (with regard to a hash-based dictionary, or in general.)

Benefits:

Intro to Alg, Chapter 11.2 Hash tables

Any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to. We call this the assumption of simple uniform hashing.

Given a hash table $T$ with $m$ slots that stores $n$ elements, we define the ***load factor*** $\alpha$ for $T$ as $n/m$, that is, the average number of elements stored in a chain. Our analysis will be in terms of $\alpha$, which can be less than, equal to, or greater than 1.

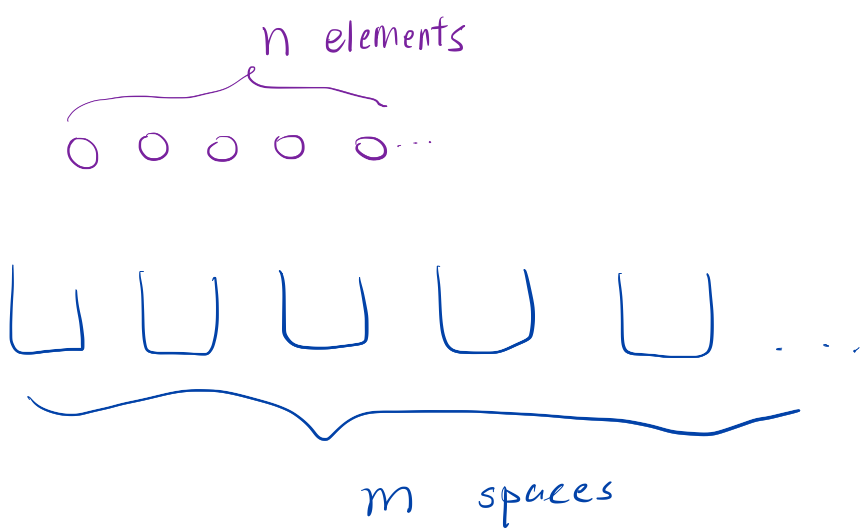
```
CHAINED-HASH-INSERT(T, x)
    insert x at the head of list T[h(x.key)]

CHAINED-HASH-SEARCH(T, k)
    search for an element with key k in list T[h(k)]

CHAINED-HASH-DELETE(T, x)
    delete x from the list T[h(x.key)]
```

***Theorem 11.1***
In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

***Theorem 11.2***
In a hash table in which collisions are resolved by chaining, a successful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.




With simple uniform assumption, if we solve the conflict by chaining,

$E[\#\text{ of elements in a certain space}] = \frac{n}{m}$.

$= \alpha$

the load factor

- Guaranteed $O(\log n)$ lookup time
- Easy to produce a sorted list of elements in the dictionary

Drawbacks:

- High cost for insertions and deletions ($O(\log n)$, even when it seems like it should be simple like deleting a leaf)
- $O(\log n)$ lookup time

## Problem 2

Find the average-case insertion, deletion, and lookup times for a hash table under the Simple Uniform Hashing Assumption, where the table has $m$ buckets:

a. $m = \Theta(n^2)$ buckets.

We first compute the load factor $\alpha = n/m = \Theta(1/n)$, and use the fact that lookup and deletion time are $\mathcal{O}(1 + \alpha)$.

*Insertion.* $\Theta(1)$, assuming we prepend the current element to our linked list at every collision.

*Lookup & Deletion.* $\mathcal{O}(1 + 1/n)$.

(source: Levet Notes)

b. $m = \Theta(\sqrt{n})$ buckets.

We first compute the load factor $\alpha = n/m = \Theta(\sqrt{n})$, and use the fact that lookup and deletion time are $\mathcal{O}(1 + \alpha)$.

*Insertion.* $\Theta(1)$, assuming we prepend the current element to our linked list at every collision.

*Lookup & Deletion.* $\mathcal{O}(1 + \sqrt{n})$.

(source: Levet Notes)

c. $m = \Theta(2^n)$ buckets.

We first compute the load factor $\alpha = n/m = \Theta(n/(2^n))$, and use the fact that lookup and deletion time are $\mathcal{O}(1 + \alpha)$.

*Insertion.* $\Theta(1)$, assuming we prepend the current element to our linked list at every collision.

*Lookup & Deletion.* $\mathcal{O}(1 + n2^{-n})$.

(source: Levet Notes)