

# COS 217: Introduction to Programming Systems

## Data Structures

“Every program depends on algorithms and data structures, but few programs depend on the invention of brand new ones.”

– Kernighan & Pike





# Goals of this Lecture

Help you learn (or refresh your memory) about:

- Common data structures: linked lists and hash tables

Why? Deep motivation:

- Common data structures serve as “high level building blocks”
- A mature programmer:
  - Rarely creates programs from scratch
  - Often creates programs using high level building blocks

Why? Shallow motivation:

- Provide background pertinent to Assignment 3
- ... especially for those who haven't taken COS 226
- ... especially for those who skipped COS 126



# Symbol Table Data Structure

Goal: maintain a collection of key/value pairs

- For now, each key is a **string**; each value is an **int**
- Lookup by key, get value back
- Unknown number of key-value pairs

## Examples

- (student name, class year)
  - (“Andrew Appel”, 81), (“Jen Rexford”, 91), (“JP Singh”, 87)
- (baseball player, number)
  - (“Ruth”, 3), (“Gehrig”, 4), (“Mantle”, 7)
- (variable name, value)
  - (“maxLength”, 2000), (“i”, 7), (“j”, -10)

# Agenda



## **Linked lists**

Hash tables

Hash table issues

Symbol table key ownership

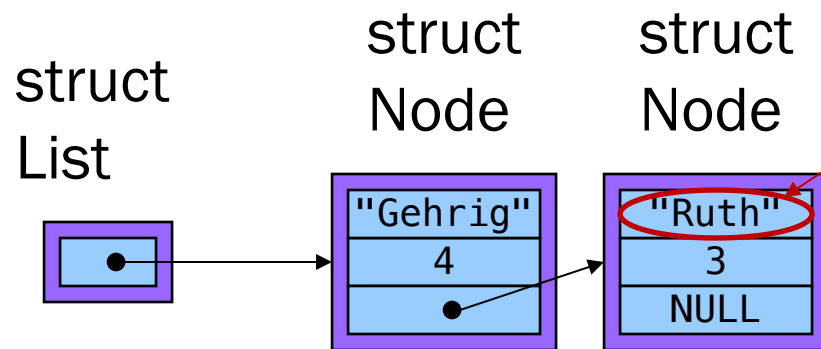


# Linked List Data Structure

```
struct Node {
    const char *key;
    int value;
    struct Node *next;
};

struct List {
    struct Node *first;
};
```

Your Assignment 3  
data structures will  
be more general and  
perhaps more elaborate



Really this is the  
address at which  
a string with  
contents "Ruth"  
resides

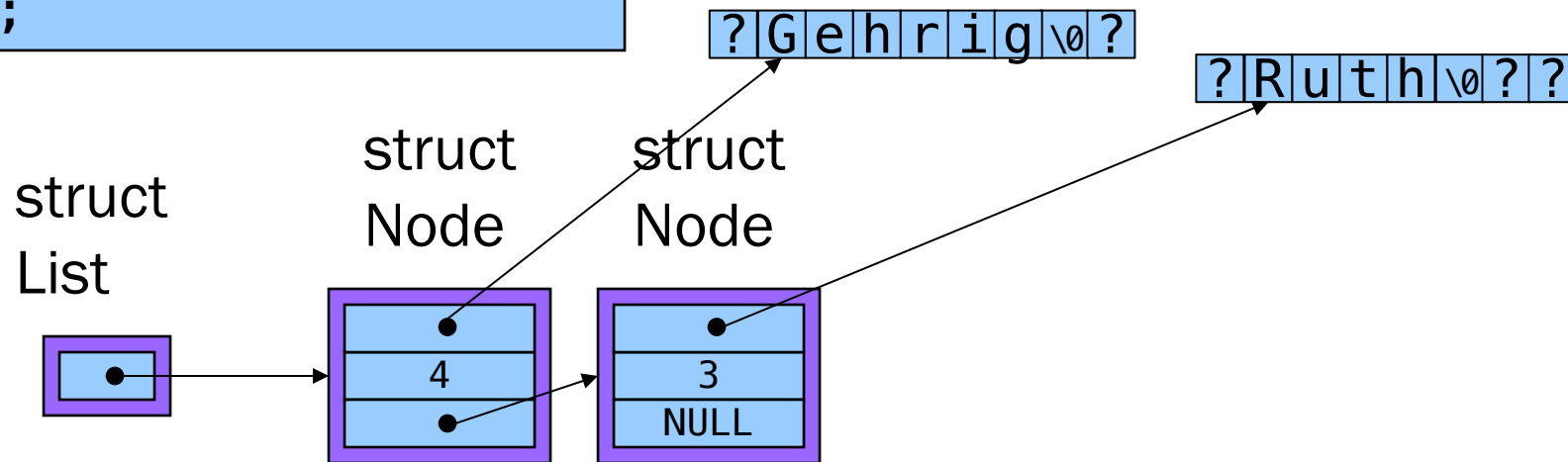


# Linked List Data Structure

```
struct Node {
    const char *key;
    int value;
    struct Node *next;
};

struct List {
    struct Node *first;
};
```

Your Assignment 3  
data structures will  
be more general and  
perhaps more elaborate



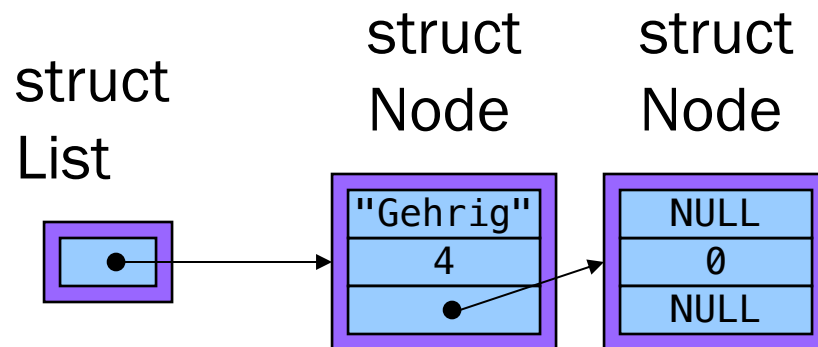


# Accessing a Linked List

```
struct Node {
    const char *key;
    int value;
    struct Node *next;
};

struct List {
    struct Node *first;
};
```

```
struct List lineup;
struct Node g;
struct Node* r =
    calloc(1, sizeof(struct Node));
g.key = "Gehrig";
lineup.first = &g;
(*lineup.first).value = 4;
lineup.first->value = 4;
(*lineup.first).next = r;
lineup.first->next = r;
```





# Linked List Algorithms

## Create

- Allocate `List` structure; set `first` to `NULL`
- Performance:  $O(1) \Rightarrow$  fast

## Add (no check for duplicate key required)

- Insert new node containing key/value pair at front of list
- Performance:  $O(1) \Rightarrow$  fast

## Add (check for duplicate key required)

- Traverse list to check for node with duplicate key
- Insert new node containing key/value pair into list
- Performance:  $O(n) \Rightarrow$  slow





# Linked List Algorithms

## Search

- Traverse the list, looking for given key
- Stop when key found, or reach end
- Performance: ???



# iClicker Question



Q: How fast is searching for a key in a linked list?

- A. Always fast –  $O(1)$
- B. Always slow –  $O(n)$
- C. On average, fast
- D. On average, slow



# Linked List Algorithms

## Search

- Traverse the list, looking for given key
- Stop when key found, or reach end
- Performance:  $O(n) \Rightarrow$  slow

## Free

- Free Node structures while traversing
- Free List structure
- Performance:  $O(n) \Rightarrow$  slow

# Agenda



Linked lists

**Hash tables**

Hash table issues

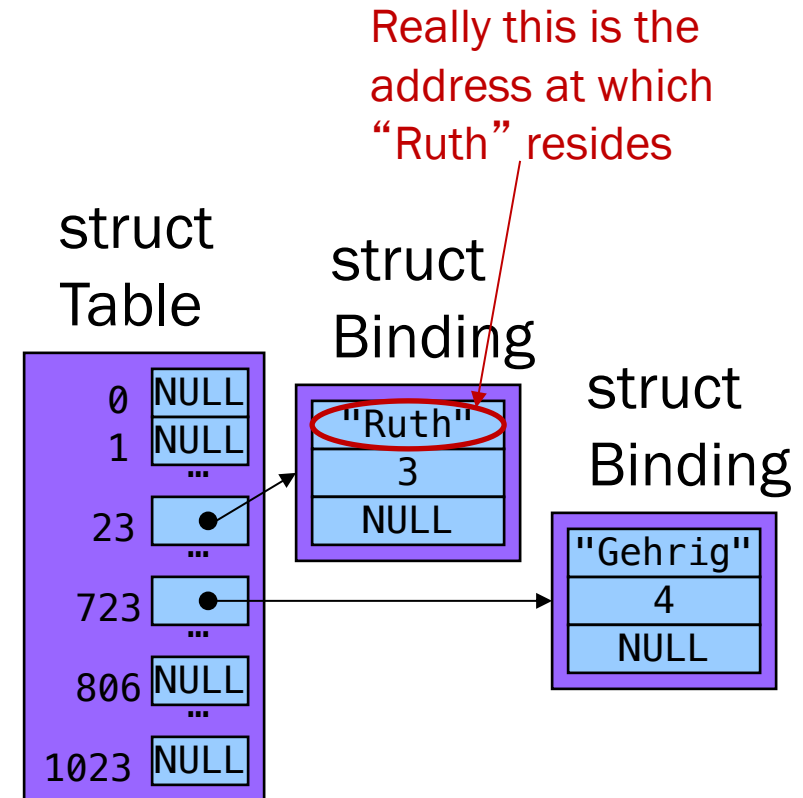
Symbol table key ownership



# Hash Table Data Structure (For COS 226 nerds – hashing with separate chaining)

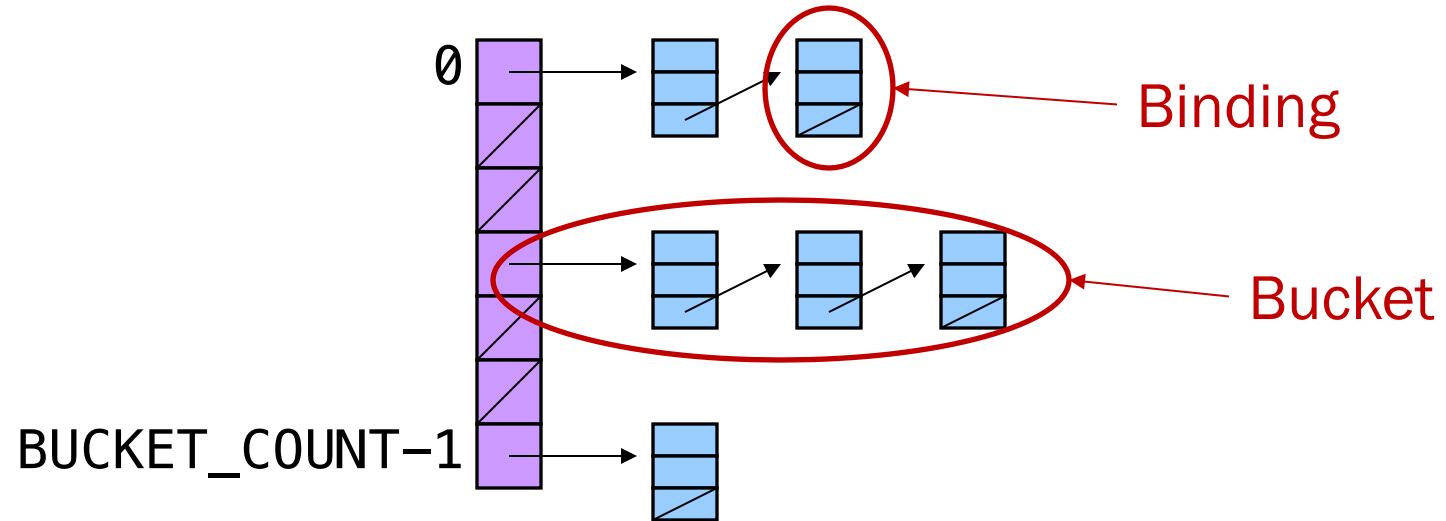
## Array of linked lists

```
enum { BUCKET_COUNT = 1024 };  
  
struct Binding {  
    const char *key;  
    int value;  
    struct Binding *next;  
};  
  
struct Table {  
    struct Binding *buckets[BUCKET_COUNT];  
};
```





# Hash Table Data Structure



**Hash function** maps given key to an integer

Mod integer by **BUCKET\_COUNT** to determine proper bucket



# Hash Table Example

Example: `BUCKET_COUNT = 7`

Add (if not already present) bindings with these keys:

- the, cat, in, the, hat

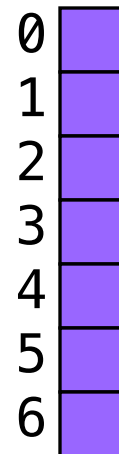


# Hash Table Example (cont.)

First key: “the”

- $\text{hash}(\text{“the”}) = 965156977; 965156977 \% 7 = 1$

Search buckets [1] for binding with key “the”; not found

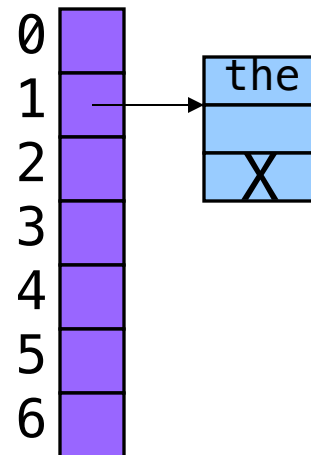






# Hash Table Example (cont.)

Add binding with key “the” and its value to buckets [1]



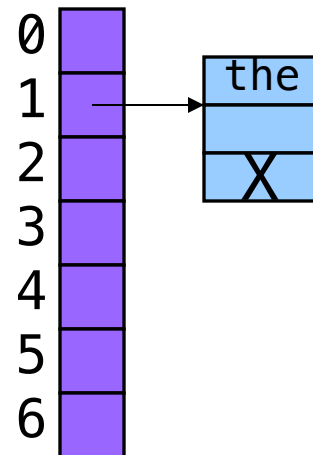


# Hash Table Example (cont.)

Second key: “cat”

- $\text{hash}(\text{“cat”}) = 3895848756; 3895848756 \% 7 = 2$

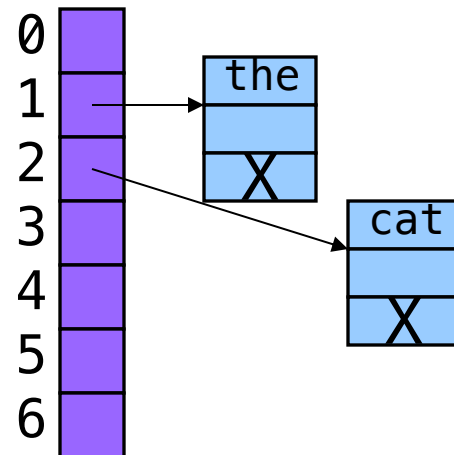
Search buckets [2] for binding with key “cat”; not found





# Hash Table Example (cont.)

Add binding with key “cat” and its value to buckets [2]



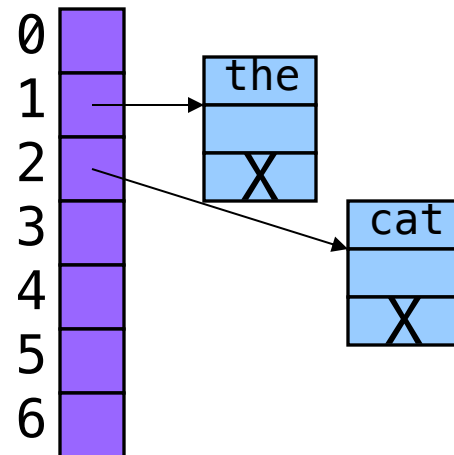


# Hash Table Example (cont.)

Third key: “in”

- $\text{hash}(\text{“in”}) = 6888005; 6888005 \% 7 = 5$

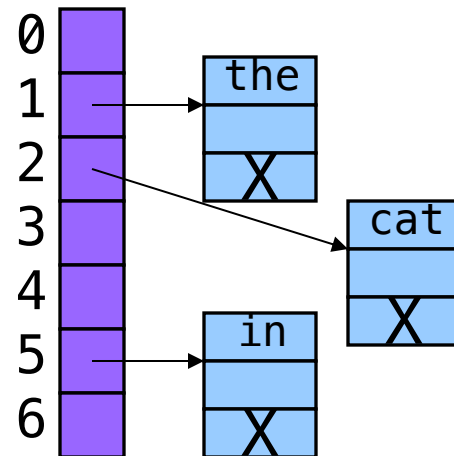
Search buckets [5] for binding with key “in”; not found





# Hash Table Example (cont.)

Add binding with key “in” and its value to buckets [5]





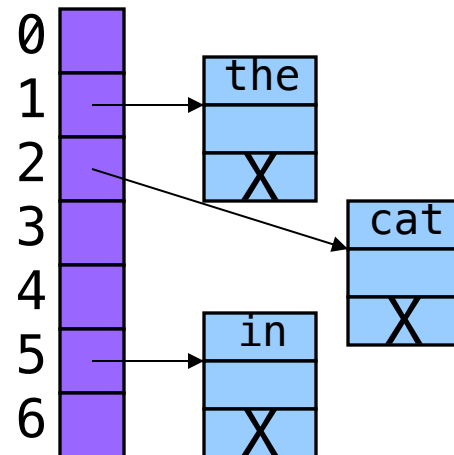
# Hash Table Example (cont.)

Fourth word: “the”

- $\text{hash}(\text{“the”}) = 965156977; 965156977 \% 7 = 1$

Search buckets [1] for binding with key “the”; found it!

- Don’t change hash table



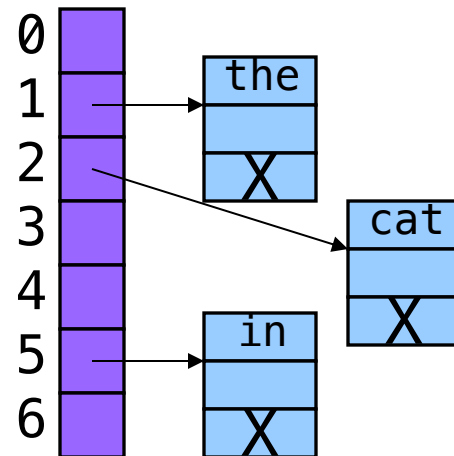


# Hash Table Example (cont.)

Fifth key: “hat”

- $\text{hash}(\text{“hat”}) = 865559739; 865559739 \% 7 = 2$

Search buckets [2] for binding with key “hat”; not found

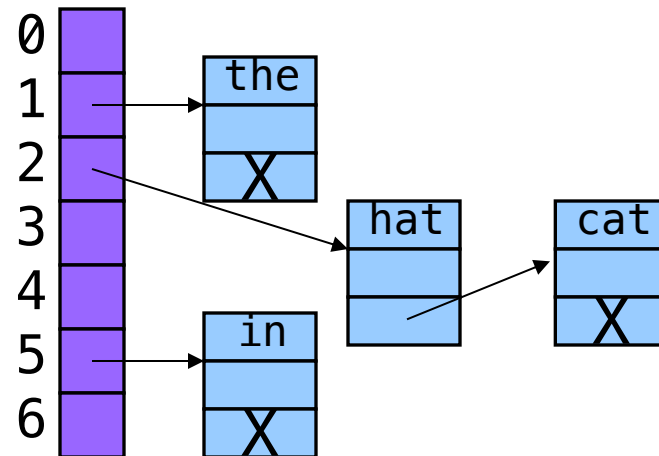




# Hash Table Example (cont.)

Add binding with key “hat” and its value to buckets [2]

- At front or back?







# Hash Table Algorithms

## Create

- Allocate Table structure; set each bucket to NULL
- Performance:  $O(1) \Rightarrow$  fast

## Add

- Hash the given key
- Mod by BUCKET\_COUNT to determine proper bucket
- Traverse proper bucket to make sure no duplicate key
- Insert new binding containing key/value pair into proper bucket
- Performance: ???



# iClicker Question



Q: How fast is adding a key to a hash table?

- A. Always fast
- B. Usually fast, but depends on how many keys are in the table
- C. Usually fast, but depends on how many keys hash to the same bucket
- D. Usually slow
- E. Always slow



# Hash Table Algorithms

## Search

- Hash the given key
- Mod by `BUCKET_COUNT` to determine proper bucket
- Traverse proper bucket, looking for binding with given key
- Stop when key found, or reach end
- Performance: Usually  $O(1) \Rightarrow$  fast

## Free

- Traverse each bucket, freeing bindings
- Free `Table` structure
- Performance:  $O(n) \Rightarrow$  slow

# Agenda



Linked lists

Hash tables

**Hash table issues**

Symbol table key ownership



# How Many Buckets?

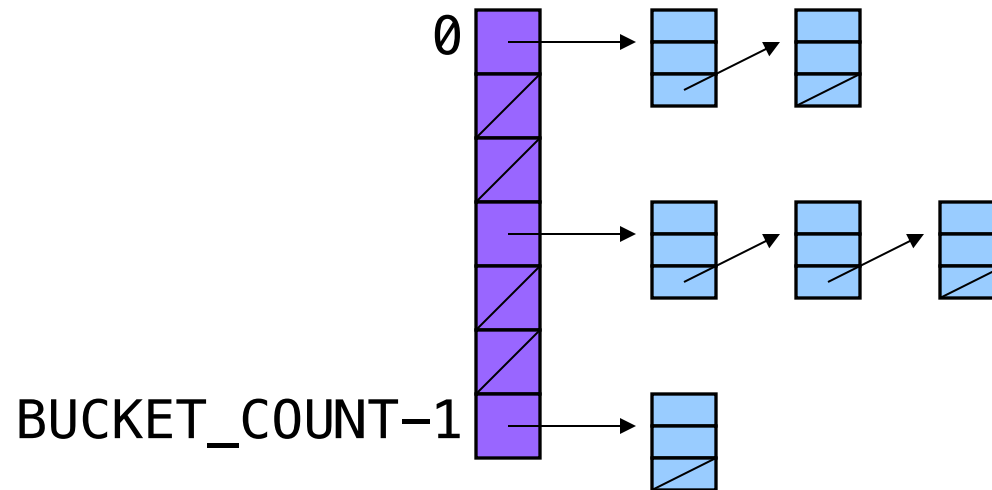
Many!

- Too few  $\Rightarrow$  large buckets  $\Rightarrow$  slow add, slow search

But not too many!

- Too many  $\Rightarrow$  memory is wasted

This is OK:



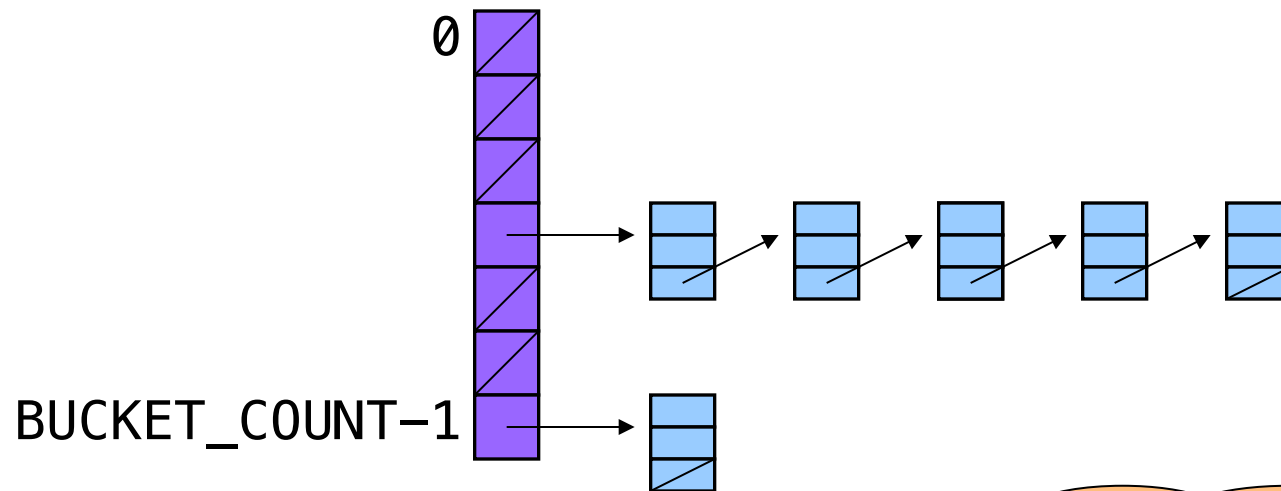


# What Hash Function?

Should distribute bindings across the buckets well

- Distribute bindings over the range  $0, 1, \dots, \text{BUCKET\_COUNT}-1$
- Distribute bindings *evenly* to avoid very long buckets

This is not so good:



What would be the worst possible hash function?



# How to Hash Strings?

Simple hash schemes don't distribute the keys evenly

- Number of characters, mod BUCKET\_COUNT
- Sum the numeric codes of all characters, mod BUCKET\_COUNT
- ...

A reasonably good hash function:

- Weighted sum of characters  $s_i$  in the string  $s$ 
  - $(\sum a^i s_i) \bmod \text{BUCKET\_COUNT}$
- Best if  $a$  and BUCKET\_COUNT are relatively prime
  - e.g.,  $a = 65599$ , BUCKET\_COUNT = 1024



# How to Hash Strings?

A bit of math, and translation to code, yields:

```
size_t hash(const char *s, size_t bucketCount)
{
    enum { HASH_MULT = 65599 };
    size_t i;
    size_t h = 0;
    for (i = 0; s[i] != '\0'; i++)
        h = h * HASH_MULT + (size_t)s[i];
    return h % bucketCount;
}
```



# Agenda



Linked lists

Hash tables

Hash table issues

**Symbol table key ownership**



# How to Protect Keys?

Suppose a hash table function `Table_add()` contains this code:

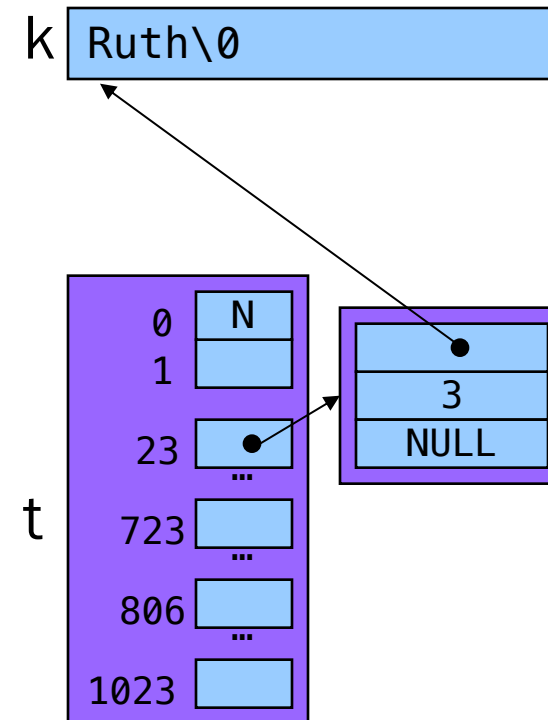
```
void Table_add(struct Table *t, const char *key, int value)
{
    ...
    struct Binding *p =
        (struct Binding*)malloc(sizeof(struct Binding));
    p->key = key;
    ...
}
```



# How to Protect Keys?

Problem: Consider this calling code:

```
struct Table *t;  
char k[100] = "Ruth";  
...  
Table_add(t, k, 3);
```





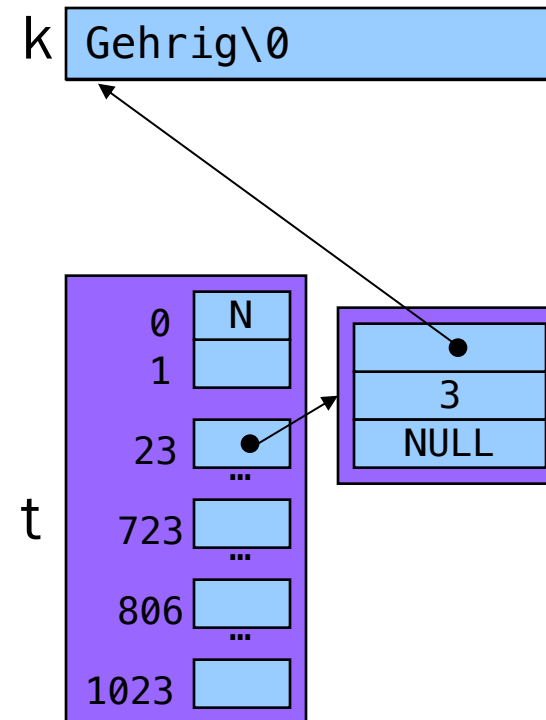
# How to Protect Keys?

Problem: Consider this calling code:

```
struct Table *t;  
char k[100] = "Ruth";  
...  
Table_add(t, k, 3);  
strcpy(k, "Gehrig");
```

**k is REALLY &k[0]!**

What happens if the client searches t for "Ruth"? For Gehrig?





# How to Protect Keys?

Solution: Table\_add() saves a defensive copy of the given key

```
void Table_add(struct Table *t, const char *key, int value)
{
    ...
    struct Binding *p =
        (struct Binding*)malloc(sizeof(struct Binding));
    p->key = (const char*)malloc(strlen(key) + 1);
    strcpy((char*)p->key, key);
    ...
}
```

Why add 1?

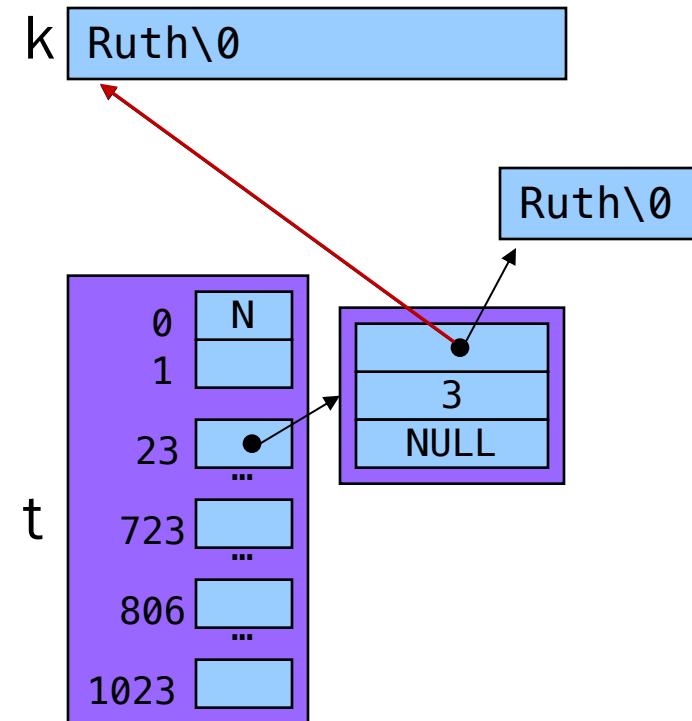
What is missing from this code that you should have in yours?



# How to Protect Keys?

Now consider same calling code:

```
struct Table *t;  
char k[100] = "Ruth";  
...  
Table_add(t, k, 3);
```



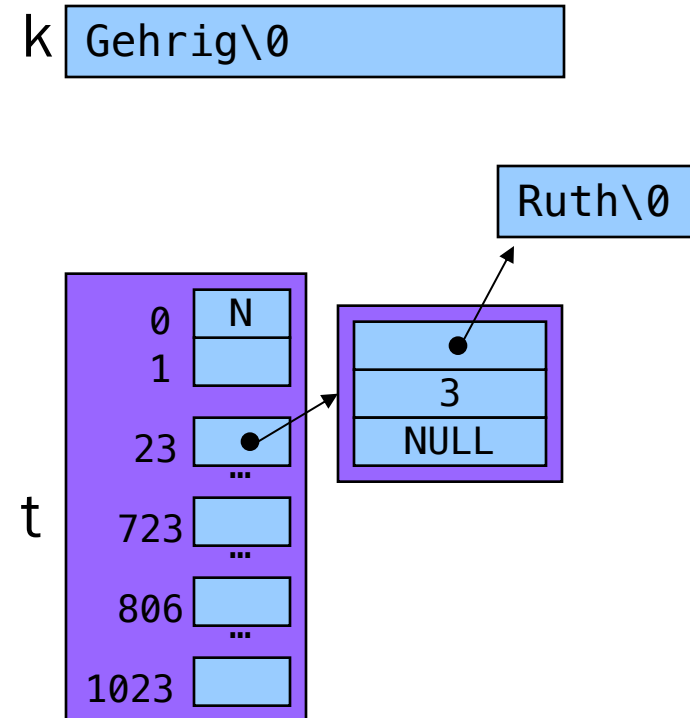


# How to Protect Keys?

Now consider same calling code:

```
struct Table *t;  
char k[100] = "Ruth";  
...  
Table_add(t, k, 3);  
strcpy(k, "Gehrig");
```

Hash table is  
not corrupted!





# Who Owns the Keys?

Then the hash table **owns** its keys

- That is, the hash table allocated the memory in which its keys reside
- `Table_remove()` function must also free the memory in which the key resides, not just its binding





# Summary

## Common data structures and associated algorithms

- Linked list
  - (Maybe) fast add
  - Slow search
- Hash table
  - (Potentially) fast add
  - (Potentially) fast search
  - Very common

## Hash table issues

- (Initial) Bucket array size
- Hashing algorithms

## Symbol table concerns

- Key ownership