Data Structures and Algorithms

The material for this lecture is drawn, in part, from
The Practice of Programming (Kernighan & Pike) Chapter 2

Motivating Quotation

"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 about:
  - Common data structures and algorithms
- Why? Shallow motivation:
  - Provide examples of pointer-related C code
- Why? Deeper motivation:
  - Common data structures and algorithms serve as "high level building blocks"
  - A power programmer:
    - Rarely creates programs from scratch
    - Often creates programs using high level building blocks

A Common Task

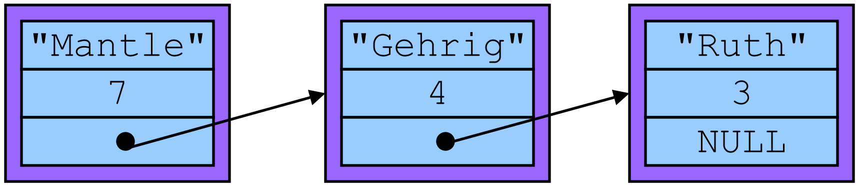
- Maintain a table of key/value pairs
  - Each key is a string; each value is an `int`
  - Unknown number of key-value pairs
  - For simplicity, allow duplicate keys (client responsibility)
    - In Assignment #3, must check for duplicate keys!
- Examples
  - (student name, grade)
    - (“john smith”, 84), (“jane doe”, 93), (“myungbak lee”, 81)
  - (baseball player, number)
    - (“Ruth”, 3), (“Gehrig”, 4), (“Mantle”, 7)
  - (variable name, value)
    - (“maxLength”, 2000), (“i”, 7), (“j”, -10)

Data Structures and Algorithms

- **Data structures**
  - Linked list of key/value pairs
  - Hash table of key/value pairs
- **Algorithms**
  - Create: Create the data structure
  - Add: Add a key/value pair
  - Search: Search for a key/value pair, by key
  - Free: Free the data structure

Data Structure #1: Linked List

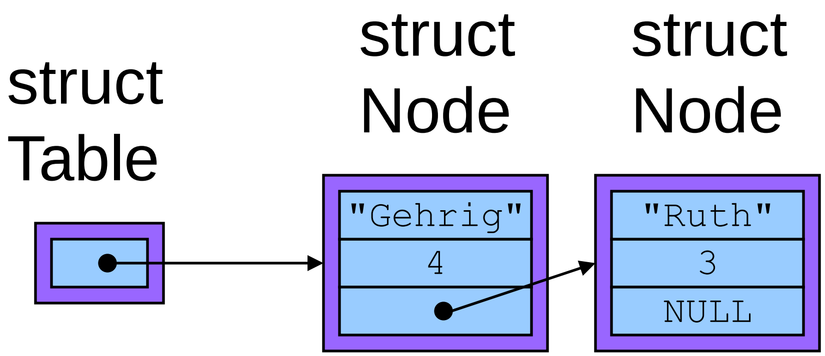
- Data structure: Nodes; each contains key/value pair and pointer to next node



- Algorithms:
  - Create: Allocate Table structure to point to first node
  - Add: Insert new node at front of list
  - Search: Linear search through the list
  - Free: Free nodes while traversing; free Table structure

Linked List: Data Structure

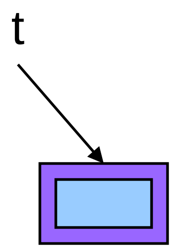
```
struct Node {
   const char *key;
   int value;
   struct Node *next;
};

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



Linked List: Create (1)

```
struct Table *Table_create(void) {
   struct Table *t;
   t = (struct Table *)
      malloc(sizeof(struct Table));
   t->first = NULL;
   return t;
}
```

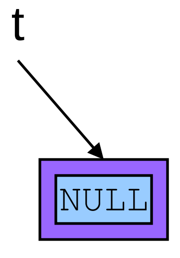
```
struct Table *t;
…
t = Table_create();
…
```



Linked List: Create (2)

```
struct Table *Table_create(void) {
   struct Table *t;
   t = (struct Table *)
      malloc(sizeof(struct Table));
   t->first = NULL;
   return t;
}
```

```
struct Table *t;
…
t = Table_create();
…
```

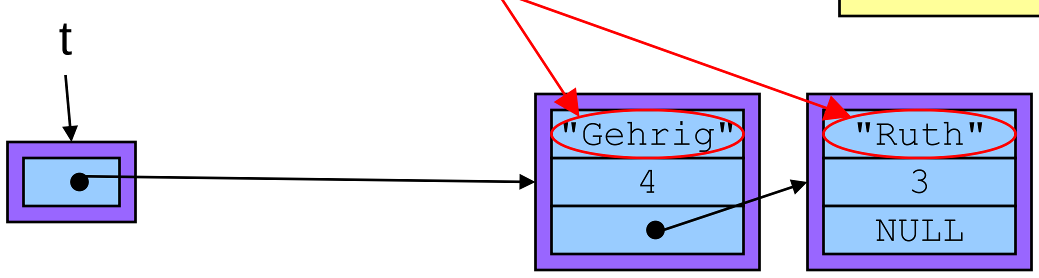


Linked List: Add (1)

```
void Table_add(struct Table *t,const char *key, int value)
{
   struct Node *p = (struct Node*)malloc(sizeof(struct Node));
   /* we omit error checking here (e.g., p == NULL) */
   p->key = key;
   p->value = value;
   p->next = t->first;
   t->first = p;
}
```

```
struct Table *t;
…
Table_add(t, "Ruth", 3);
Table_add(t, "Gehrig", 4);
Table_add(t, "Mantle", 7);
…
```

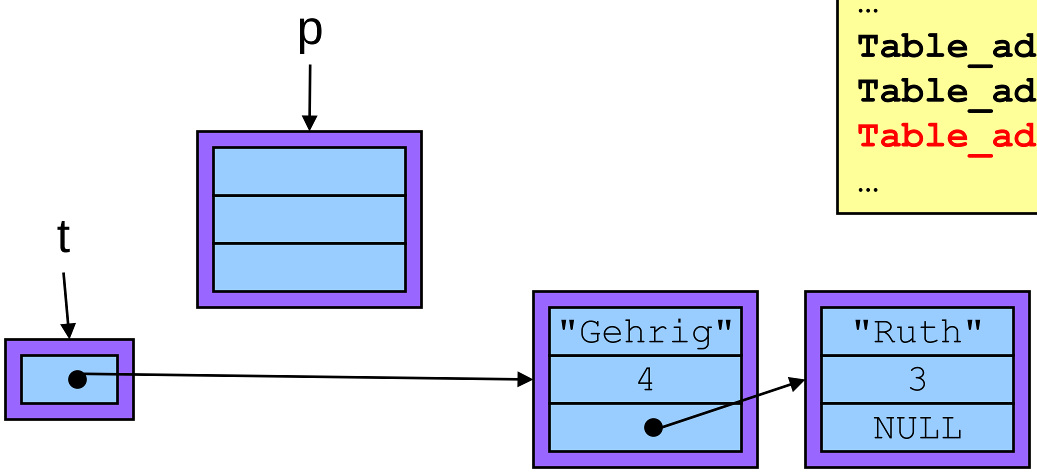
These are pointers to strings that exist in the RODATA section



Linked List: Add (2)

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

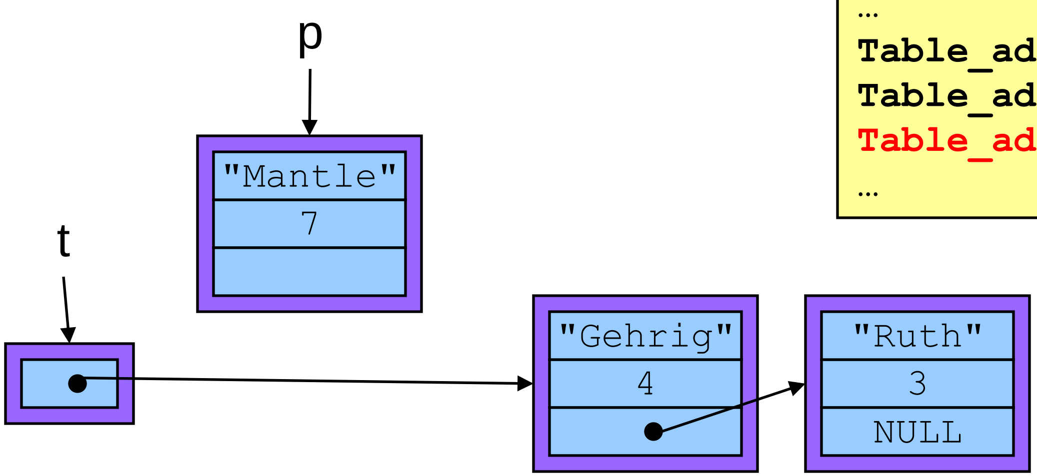
```
struct Table *t;
…
Table_add(t, "Ruth", 3);
Table_add(t, "Gehrig", 4);
Table_add(t, "Mantle", 7);
…
```



Linked List: Add (3)

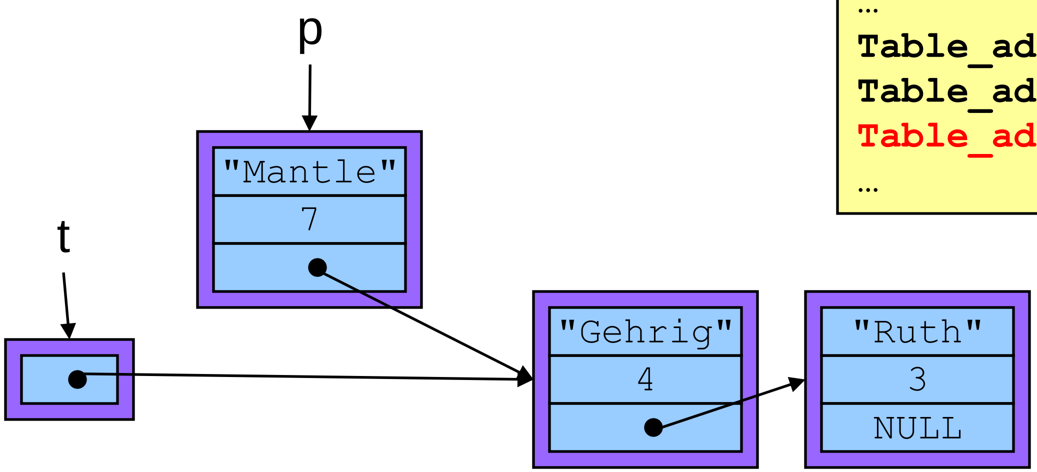
```
void Table_add(struct Table *t,
   const char *key, int value) {
   struct Node *p = (struct Node*)malloc(sizeof(struct Node));
   p->key = key;
   p->value = value;
   p->next = t->first;
   t->first = p;
}
```

```
struct Table *t;
…
Table_add(t, "Ruth", 3);
Table_add(t, "Gehrig", 4);
Table_add(t, "Mantle", 7);
…
```



Linked List: Add (4)

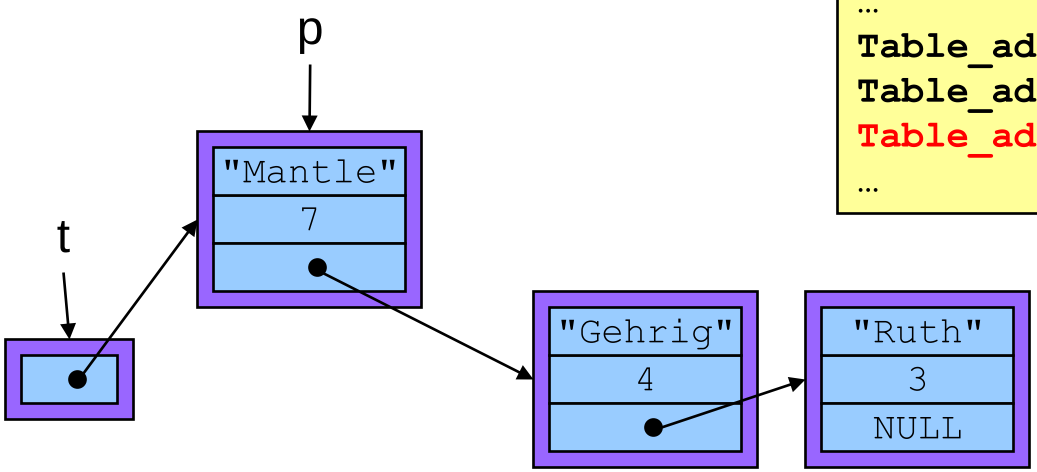
```
void Table_add(struct Table *t,
   const char *key, int value) {
   struct Node *p = (struct Node*)malloc(sizeof(struct Node));
   p->key = key;
   p->value = value;
   p->next = t->first;
   t->first = p;
}
```




```
struct Table *t;
…
Table_add(t, "Ruth", 3);
Table_add(t, "Gehrig", 4);
Table_add(t, "Mantle", 7);
…
```

Linked List: Add (5)

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

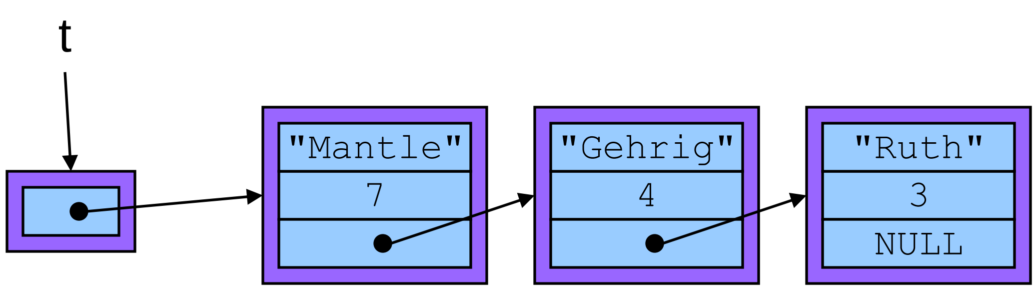



```
struct Table *t;
…
Table_add(t, "Ruth", 3);
Table_add(t, "Gehrig", 4);
Table_add(t, "Mantle", 7);
…
```

Linked List: Search (1)

```
int Table_search(struct Table *t, const char *key, int *value) {
   struct Node *p;
   for (p = t->first; p != NULL; p = p->next)
      if (strcmp(p->key, key) == 0) {
         *value = p->value;
         return 1;
      }
   return 0;
}
```

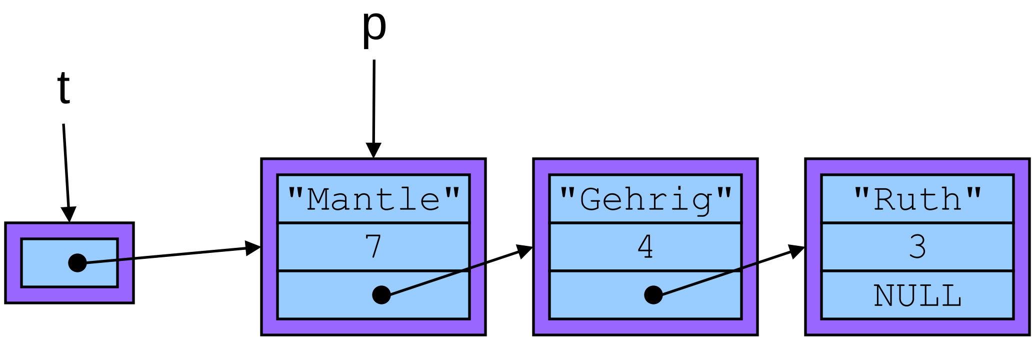
```
struct Table *t;
int value;
int found;
…
found =
   Table_search(t, "Gehrig", &value);
…
```



Linked List: Search (2)

```
int Table_search(struct Table *t, const char *key, int *value)
{
   struct Node *p;
   for (p = t->first; p != NULL; p = p->next)
      if (strcmp(p->key, key) == 0) {
         *value = p->value;
         return 1;
      }
   return 0;
}
```

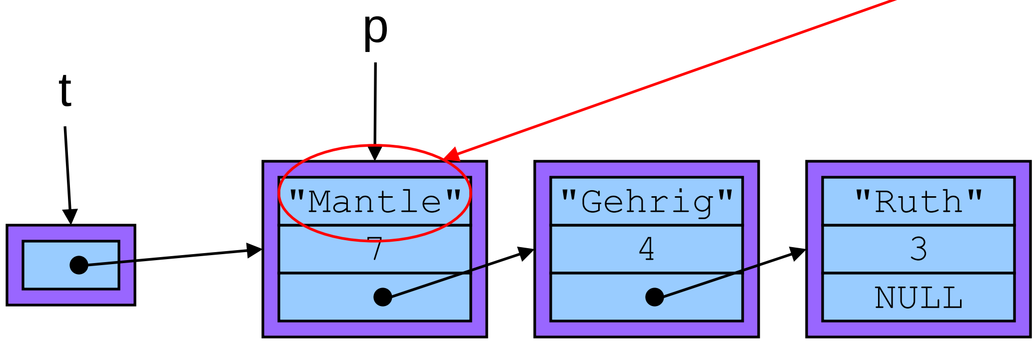
```
struct Table *t;
int value;
int found;
…
found =
   Table_search(t, "Gehrig", &value);
…
```



Linked List: Search (3)

```
int Table_search(struct Table *t, const char *key, int *value)
{
   struct Node *p;
   for (p = t->first; p != NULL; p = p->next)
      if (strcmp(p->key, key) == 0) {
         *value = p->value;
         return 1;
      }
   return 0;
}
```

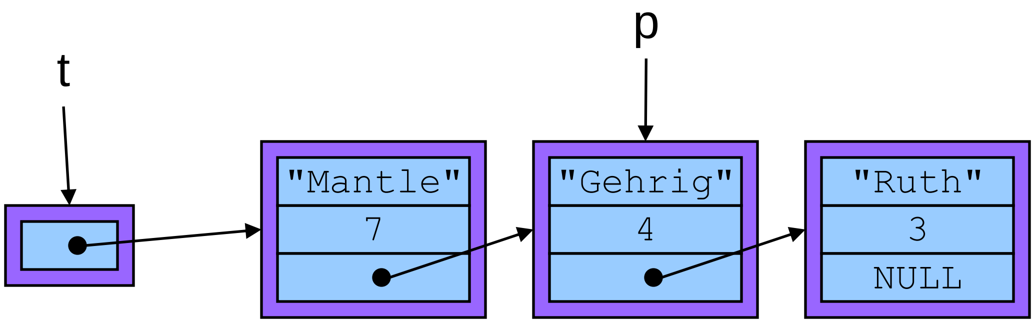
```
struct Table *t;
int value;
int found;
…
found =
   Table_search(t, "Gehrig", &value);
…
```



Linked List: Search (4)

```
int Table_search(struct Table *t, const char *key, int *value)
{
   struct Node *p;
   for (p = t->first; p != NULL; p = p->next)
      if (strcmp(p->key, key) == 0) {
         *value = p->value;
         return 1;
      }
   return 0;
}
```

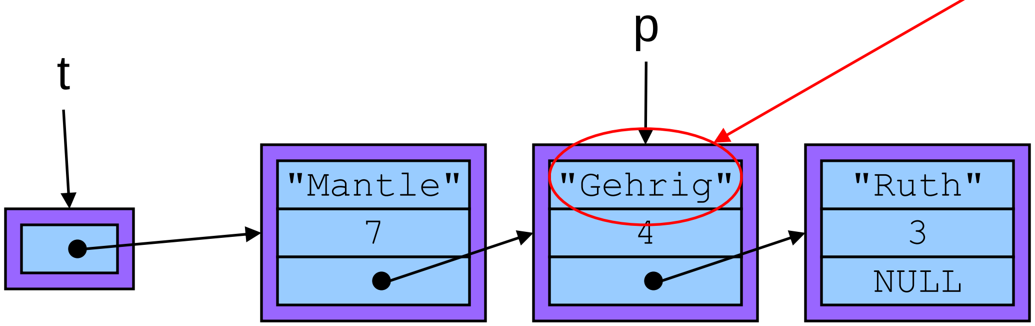
```
struct Table *t;
int value;
int found;
…
found =
   Table_search(t, "Gehrig", &value);
…
```



Linked List: Search (5)

```
int Table_search(struct Table *t, const char *key, int *value)
{
   struct Node *p;
   for (p = t->first; p != NULL; p = p->next)
      if (strcmp(p->key, key) == 0) {
         *value = p->value;
         return 1;
      }
   return 0;
}
```

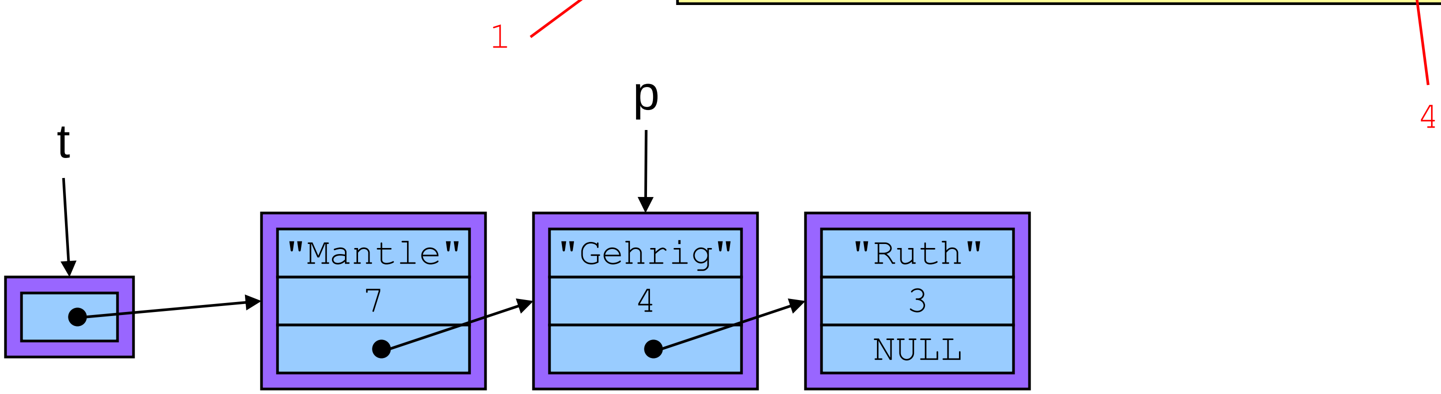
```
struct Table *t;
int value;
int found;
…
found =
   Table_search(t, "Gehrig", &value);
…
```



Linked List: Search (6)

```
int Table_search(struct Table *t, const char *key, int *value)
{
   struct Node *p;
   for (p = t->first; p != NULL; p = p->next)
      if (strcmp(p->key, key) == 0) {
         *value = p->value;
         return 1;
      }
   return 0;
}
```

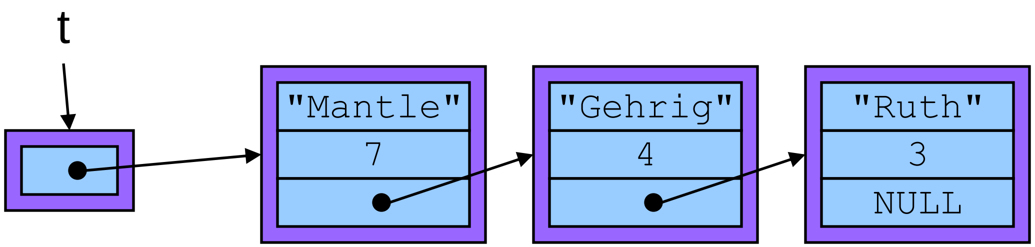
```
struct Table *t;
int value;
int found;
…
found =
   Table_search(t, "Gehrig", &value);
…
```



Linked List: Free (1)

```
void Table_free(struct Table *t) {
   struct Node *p;
   struct Node *nextp;
   for (p = t->first; p != NULL; p = nextp) {
      nextp = p->next;
      free(p);
   }
   free(t);
}
```

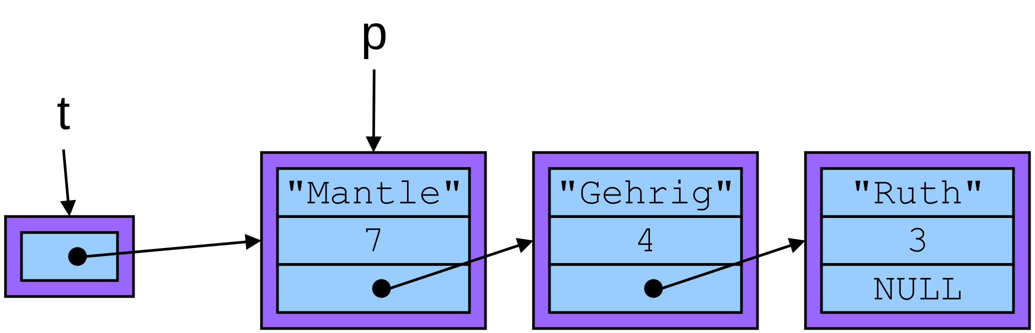
```
struct Table *t;
…
Table_free(t);
…
```



Linked List: Free (2)

```
void Table_free(struct Table *t)
{
   struct Node *p;
   struct Node *nextp;
   for (p = t->first; p != NULL; p = nextp) {
      nextp = p->next;
      free(p);
   }
   free(t);
}
```

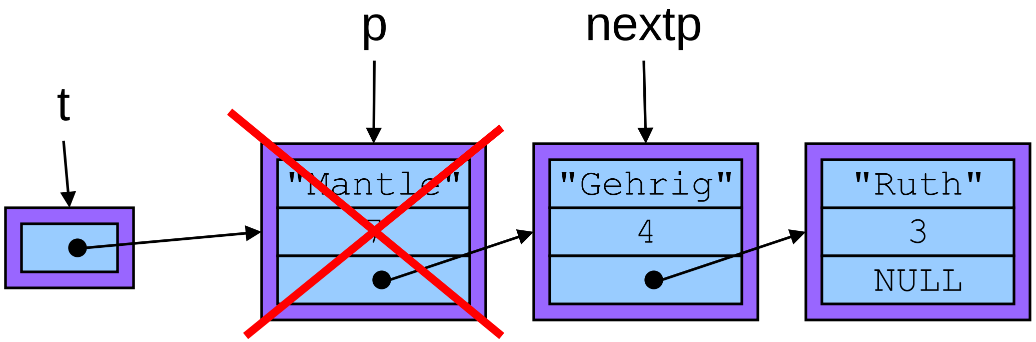
```
struct Table *t;
…
Table_free(t);
…
```



Linked List: Free (3)

```
void Table_free(struct Table *t) {
   struct Node *p;
   struct Node *nextp;
   for (p = t->first; p != NULL; p = nextp) {
      nextp = p->next;
      free(p);
   }
   free(t);
}
```

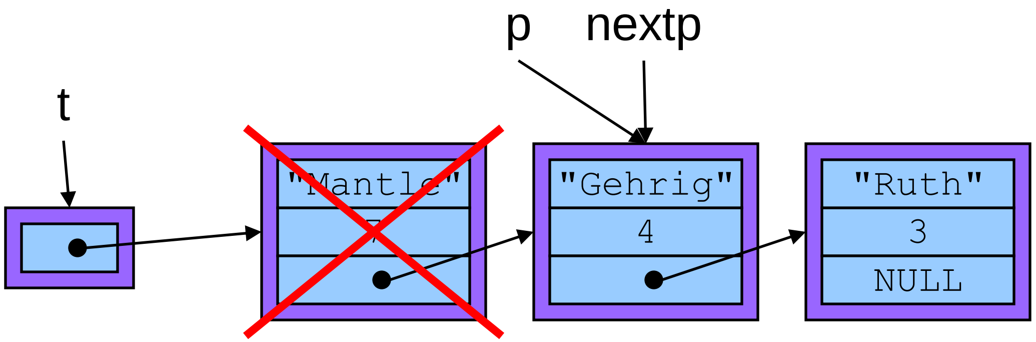
```
struct Table *t;
…
Table_free(t);
…
```



Linked List: Free (4)

```
void Table_free(struct Table *t) {
   struct Node *p;
   struct Node *nextp;
   for (p = t->first; p != NULL; p = nextp) {
      nextp = p->next;
      free(p);
   }
   free(t);
}
```

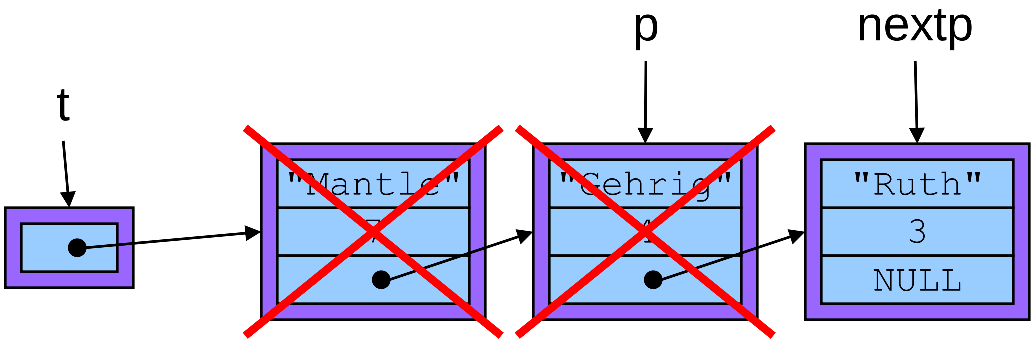
```
struct Table *t;
…
Table_free(t);
…
```



Linked List: Free (5)

```
void Table_free(struct Table *t) {
   struct Node *p;
   struct Node *nextp;
   for (p = t->first; p != NULL; p = nextp) {
      nextp = p->next;
      free(p);
   }
   free(t);
}
```

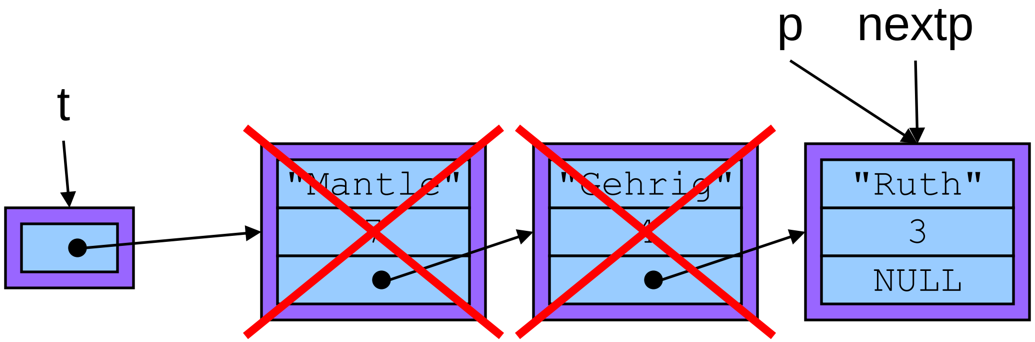
```
struct Table *t;
…
Table_free(t);
…
```



Linked List: Free (6)

```
void Table_free(struct Table *t) {
   struct Node *p;
   struct Node *nextp;
   for (p = t->first; p != NULL; p = nextp) {
      nextp = p->next;
      free(p);
   }
   free(t);
}
```

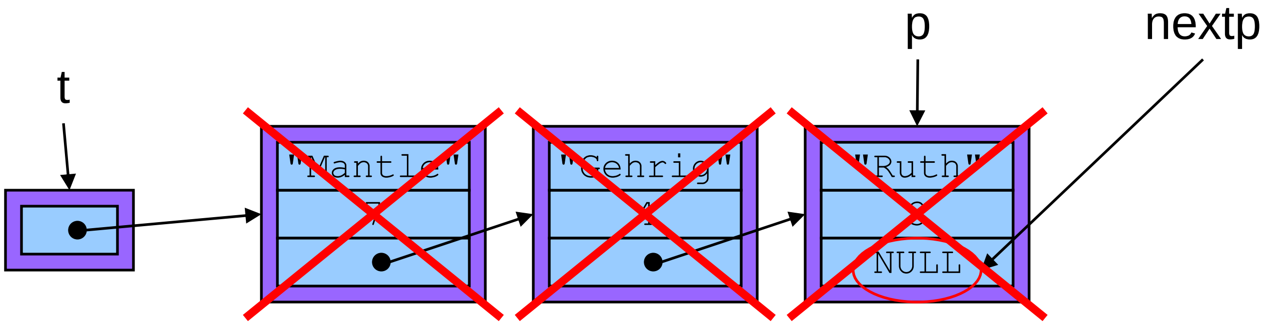
```
struct Table *t;
…
Table_free(t);
…
```



Linked List: Free (7)

```
void Table_free(struct Table *t) {
   struct Node *p;
   struct Node *nextp;
   for (p = t->first; p != NULL; p = nextp) {
      nextp = p->next;
      free(p);
   }
   free(t);
}
```

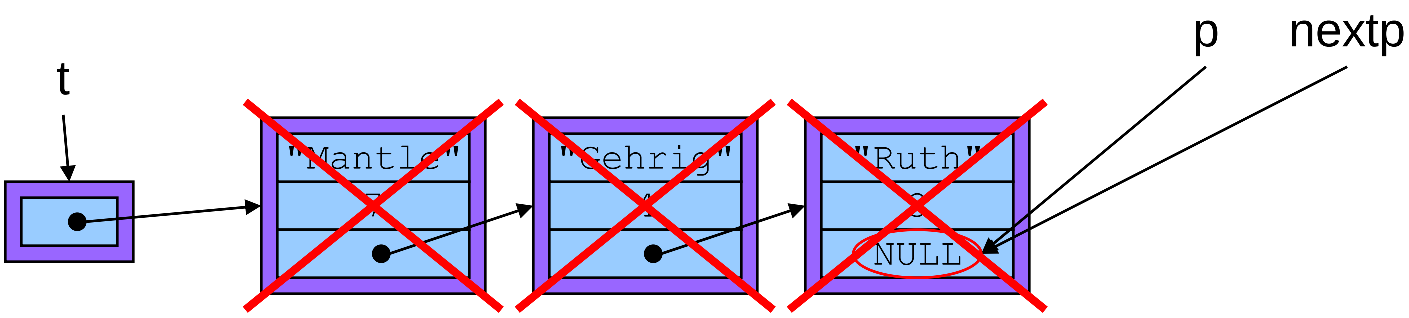
```
struct Table *t;
…
Table_free(t);
…
```



Linked List: Free (8)

```
void Table_free(struct Table *t) {
   struct Node *p;
   struct Node *nextp;
   for (p = t->first; p != NULL; p = nextp) {
      nextp = p->next;
      free(p);
   }
   free(t);
}
```

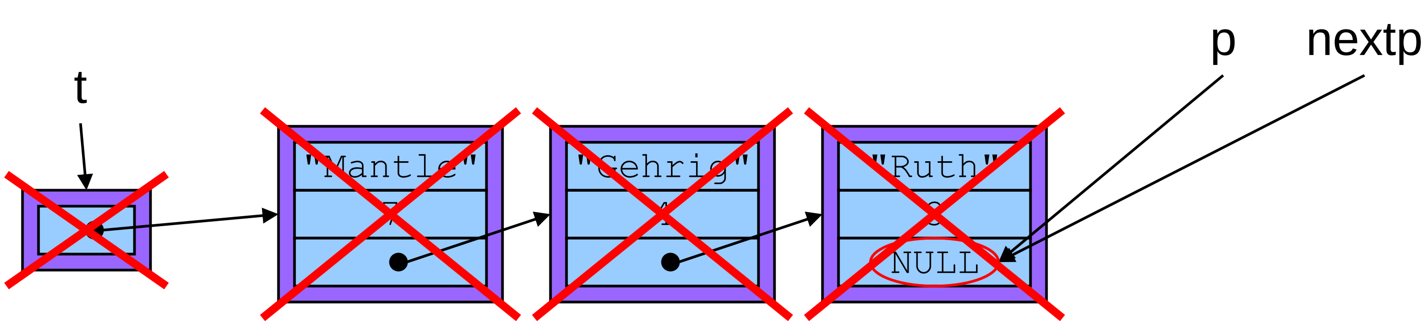
```
struct Table *t;
…
Table_free(t);
…
```



Linked List: Free (9)

```
void Table_free(struct Table *t) {
   struct Node *p;
   struct Node *nextp;
   for (p = t->first; p != NULL; p = nextp) {
      nextp = p->next;
      free(p);
   }
   free(t)
}
```

```
struct Table *t;
…
Table_free(t);
…
```



Linked List Performance

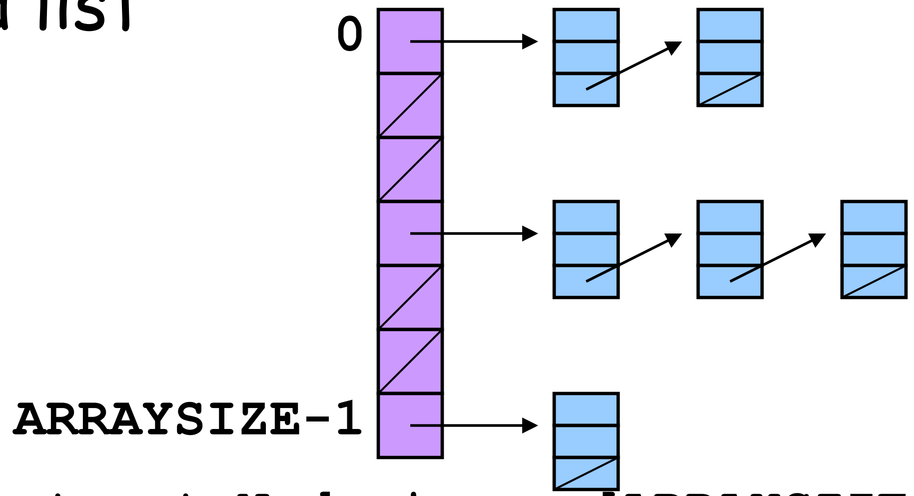
- Create: fast
- Add: fast
- Search: slow
- Free: slow

What are the asymptotic run times (big-oh notation)?

Would it be better to keep the nodes sorted by key?

Data Structure #2: Hash Table

- Fixed-size array where each element points to a linked list

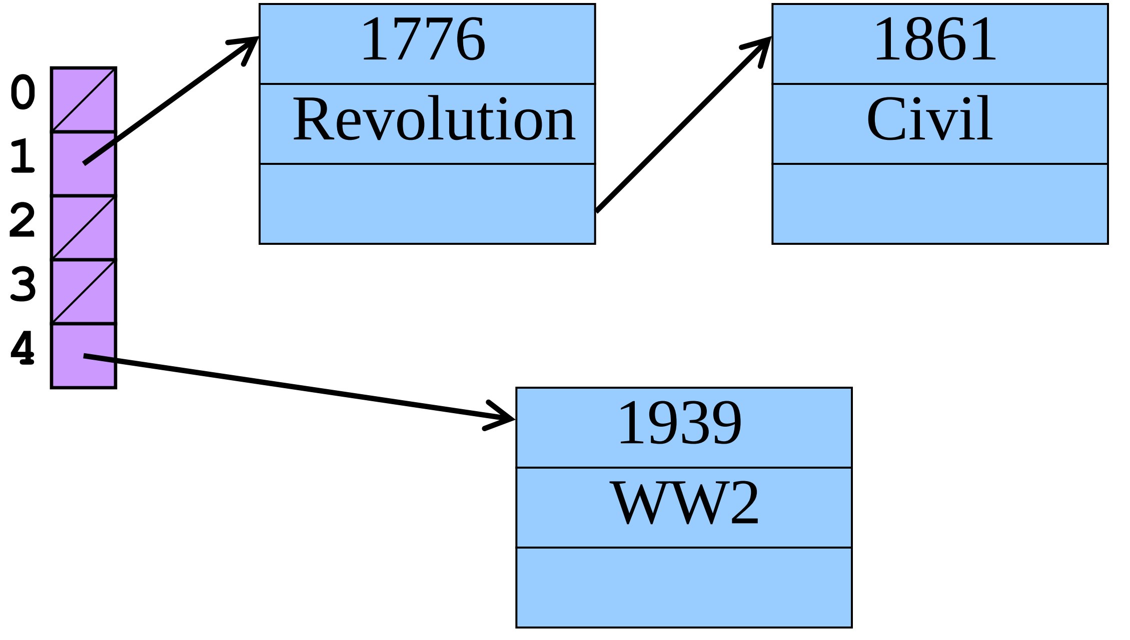



```
struct Node *array[ARRAYSIZE];
```

- Function maps each key to an array index
  - For example, for an integer key **`h`**
    - Hash function: **`i = h % ARRAYSIZE`** (mod function)
  - Go to array element **`i`**, i.e., the linked list **`hashtab[i]`**
    - Search for element, add element, remove element, etc.

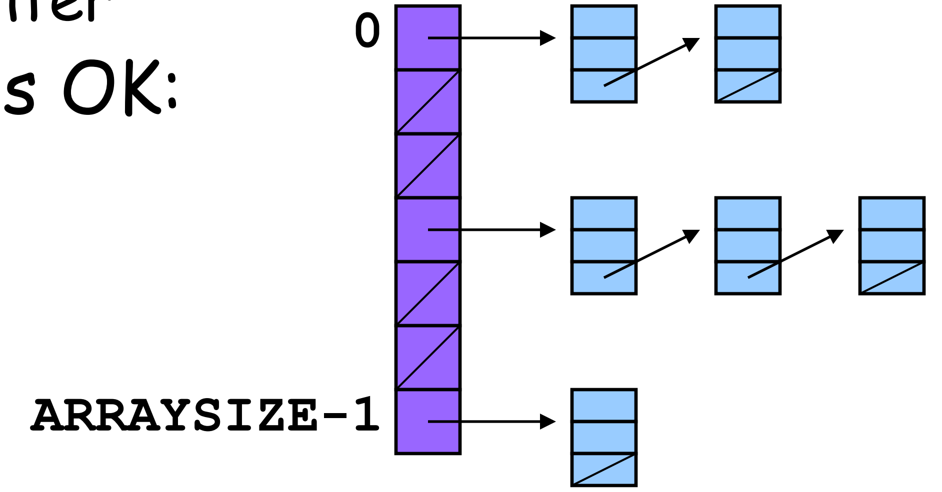
Hash Table Example

- Integer keys, array of size 5 with hash function "h mod 5"
  - "1776 % 5" is 1
  - "1861 % 5" is 1
  - "1939 % 5" is 4



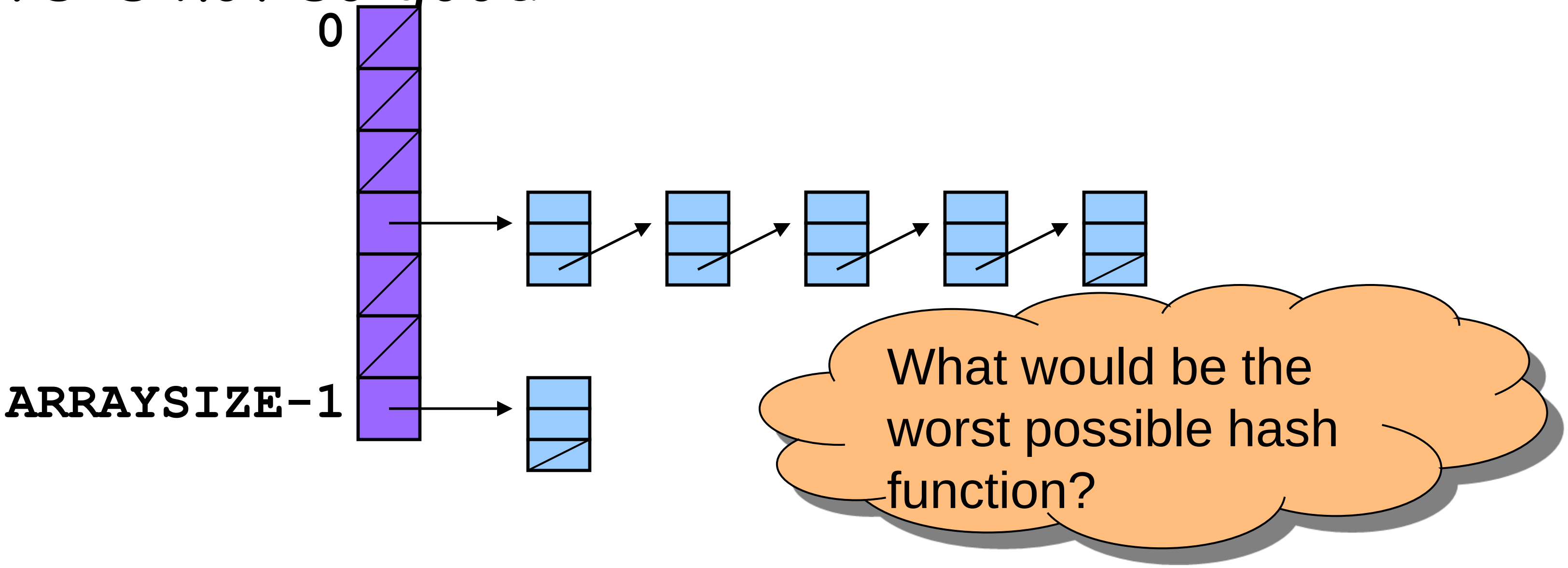
How Large an Array?

- Large enough that average "bucket" size is 1
  - Short buckets mean fast search
  - Long buckets mean slow search
- Small enough to be memory efficient
  - Not an excessive number of elements
  - Fortunately, each array element is just storing a pointer
- This is OK:



What Kind of Hash Function?

- Good at distributing elements across the array
  - Distribute results over the range 0, 1, …, ARRAYSIZE-1
  - Distribute results *evenly* to avoid very long buckets
- This is not so good:



Hashing String Keys to Integers

- Simple schemes don't distribute the keys evenly enough
  - Number of characters, mod ARRAYSIZE
  - Sum the ASCII values of all characters, mod ARRAYSIZE
  - …
- Here's a reasonably good hash function
  - Weighted sum of characters x_i in the string
    - $(\Sigma a^i x_i)$ mod ARRAYSIZE
  - Best if a and ARRAYSIZE are relatively prime
    - E.g., a = 65599, ARRAYSIZE = 1024

Implementing Hash Function

- Potentially expensive to compute a^i for each value of i
  - Computing a^i for each value of I
  - Instead, do (((x[0] * 65599 + x[1]) * 65599 + x[2]) * 65599 + x[3]) * …

```
unsigned int hash(const char *x) {
   int i;
   unsigned int h = 0U;
   for (i=0; x[i]!='\0'; i++)
      h = h * 65599 + (unsigned char)x[i];
   return h % 1024;
}
```

Can be more clever than this for powers of two!
(Described in Appendix)

Hash Table Example

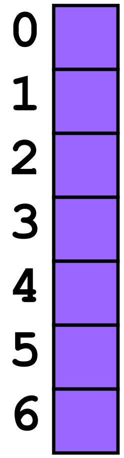
Example: ARRAYSIZE = 7

Lookup (and enter, if not present) these strings: the, cat, in, the, hat

Hash table initially empty.

First word: the. hash("the") = 965156977. 965156977 % 7 = 1.

Search the linked list table[1] for the string "the"; not found.



Hash Table Example (cont.)

Example: ARRAYSIZE = 7

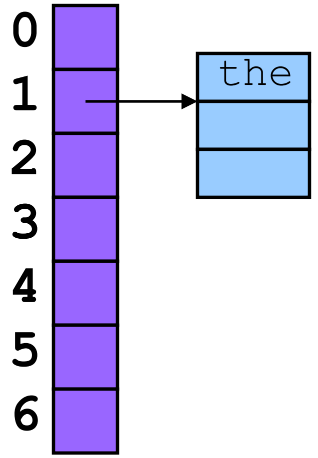
Lookup (and enter, if not present) these strings: the, cat, in, the, hat

Hash table initially empty.

First word: "the". hash("the") = 965156977. 965156977 % 7 = 1.

Search the linked list table[1] for the string "the"; not found

Now: table[1] = makelink(key, value, table[1])

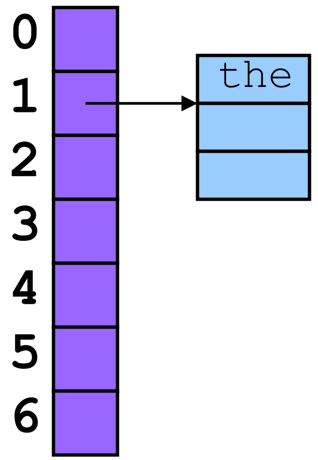


Hash Table Example (cont.)

Second word: "cat". hash("cat") = 3895848756. 3895848756 % 7 = 2.

Search the linked list table[2] for the string "cat"; not found

Now: table[2] = makelink(key, value, table[2])

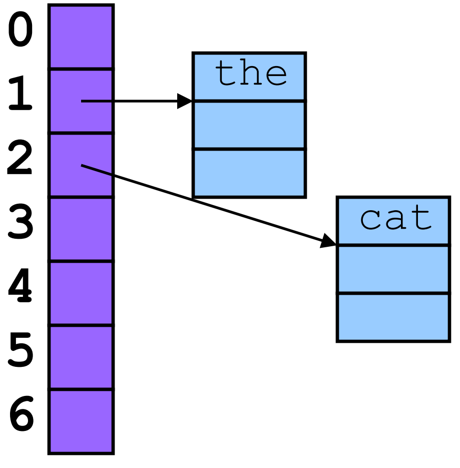


Hash Table Example (cont.)

Third word: "in". hash("in") = 6888005. 6888005% 7 = 5.

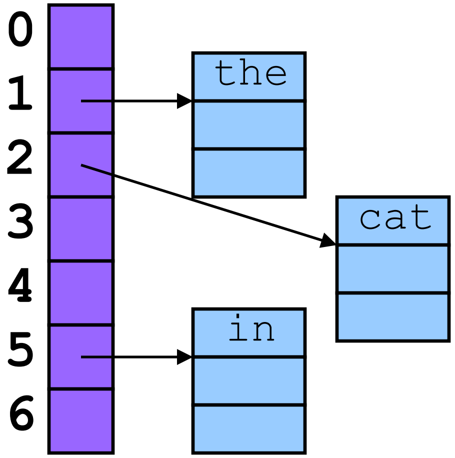
Search the linked list table[5] for the string "in"; not found

Now: table[5] = makelink(key, value, table[5])



Hash Table Example (cont.)

Fourth word: “the”. hash(“the”) = 965156977. 965156977 % 7 = 1.
Search the linked list table[1] for the string “the”; found it!



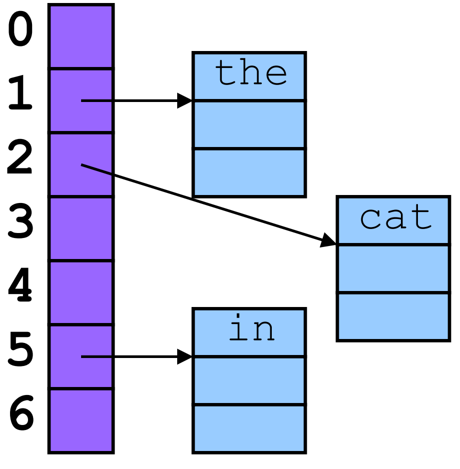
Hash Table Example (cont.)

Fourth word: “hat”. hash(“hat”) = 865559739. 865559739 % 7 = 2.

Search the linked list table[2] for the string “hat”; not found.

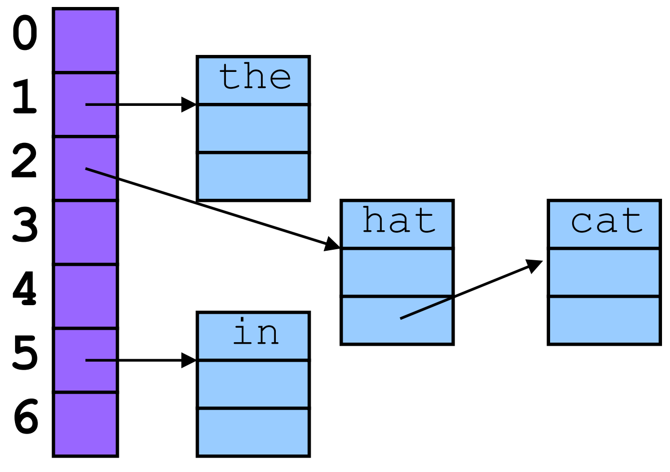
Now, insert “hat” into the linked list table[2].

At beginning or end? Doesn’t matter.



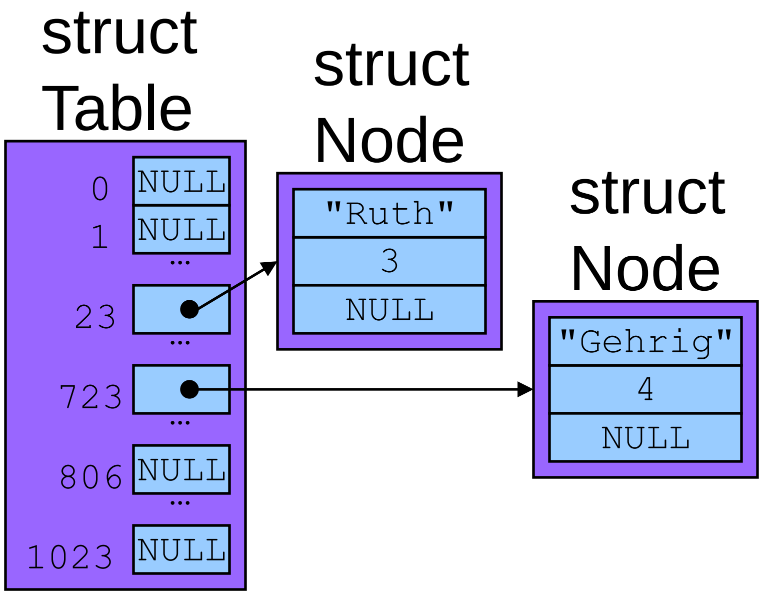
Hash Table Example (cont.)

Inserting at the front is easier, so add "hat" at the front



Hash Table: Data Structure

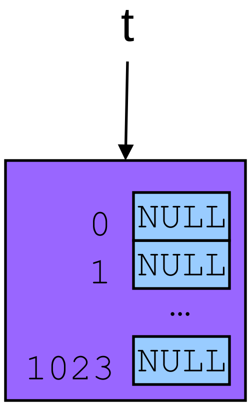
```
enum {BUCKET_COUNT = 1024};

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

struct Table {
   struct Node *array[BUCKET_COUNT];
};
```



Hash Table: Create

```
struct Table *Table_create(void) {
   struct Table *t;
   t = (struct Table*)calloc(1, sizeof(struct Table));
   return t;
}
```

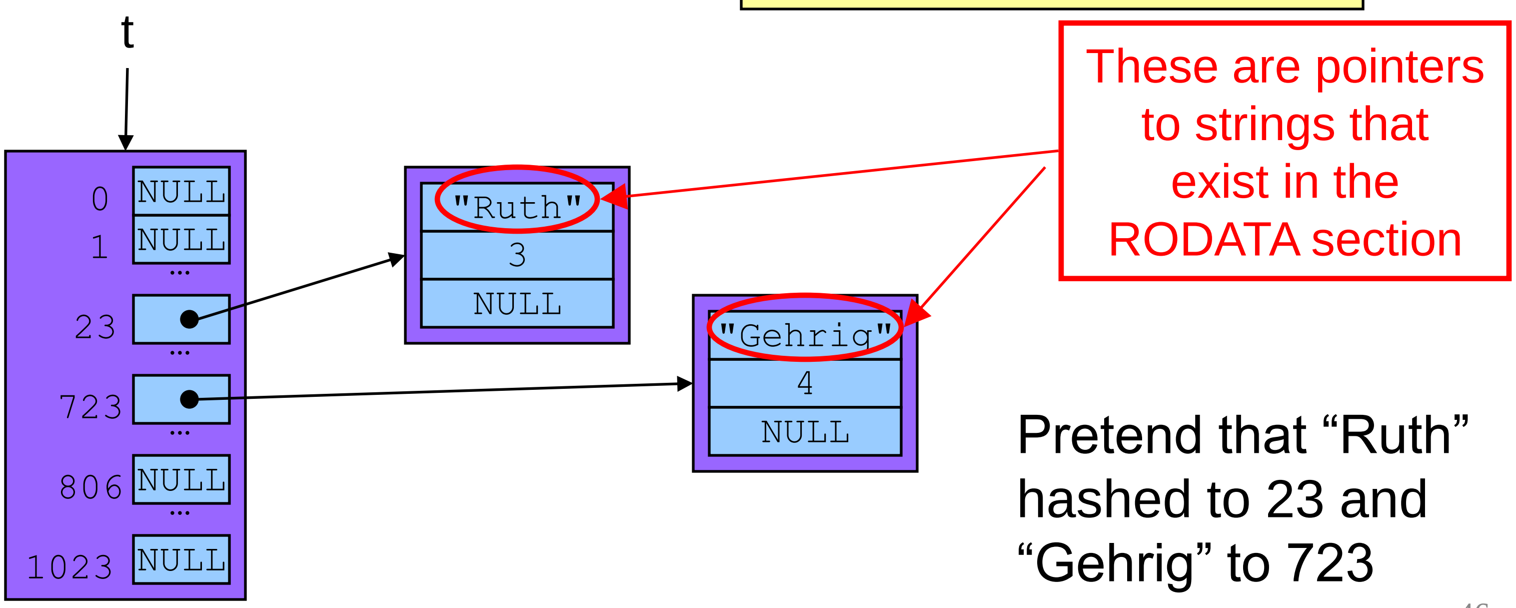
```
struct Table *t;
…
t = Table_create();
…
```



Hash Table: Add (1)

```
void Table_add(struct Table *t,const char *key, int value)
{
   struct Node *p = (struct Node*)malloc(sizeof(struct Node));
   int h = hash(key);
   p->key = key;
   p->value = value;
   p->next = t->array[h];
   t->array[h] = p;
}
```

```
struct Table *t;
…
Table_add(t, "Ruth", 3);
Table_add(t, "Gehrig", 4);
Table_add(t, "Mantle", 7);
…
```

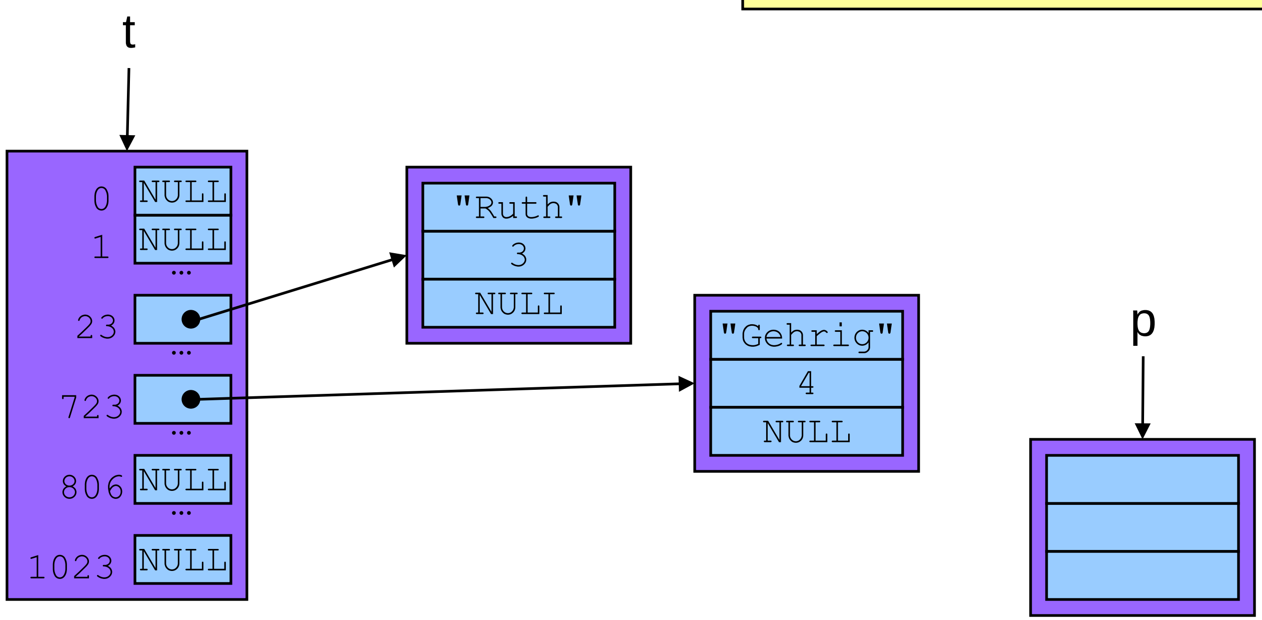


Pretend that “Ruth” hashed to 23 and “Gehrig” to 723

Hash Table: Add (2)

```
void Table_add(struct Table *t, const char *key, int value)
{
   struct Node *p = (struct Node*)malloc(sizeof(struct Node));
   int h = hash(key);
   p->key = key;
   p->value = value;
   p->next = t->array[h];
   t->array[h] = p;
}
```

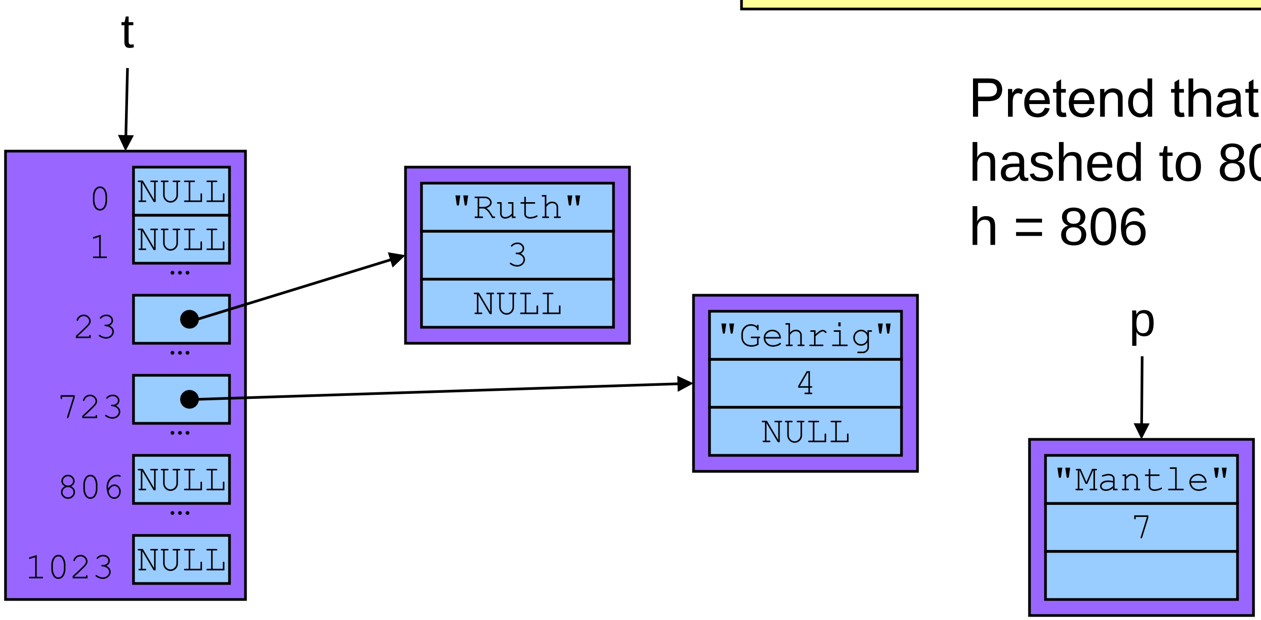
```
struct Table *t;
…
Table_add(t, "Ruth", 3);
Table_add(t, "Gehrig", 4);
Table_add(t, "Mantle", 7);
…
```



Hash Table: Add (3)

```
void Table_add(struct Table *t, const char *key, int value)
{
   struct Node *p = (struct Node*)malloc(sizeof(struct Node));
   int h = hash(key);
   p->key = key;
   p->value = value;
   p->next = t->array[h];
   t->array[h] = p;
}
```

```
struct Table *t;
…
Table_add(t, "Ruth", 3);
Table_add(t, "Gehrig", 4);
Table_add(t, "Mantle", 7);
…
```

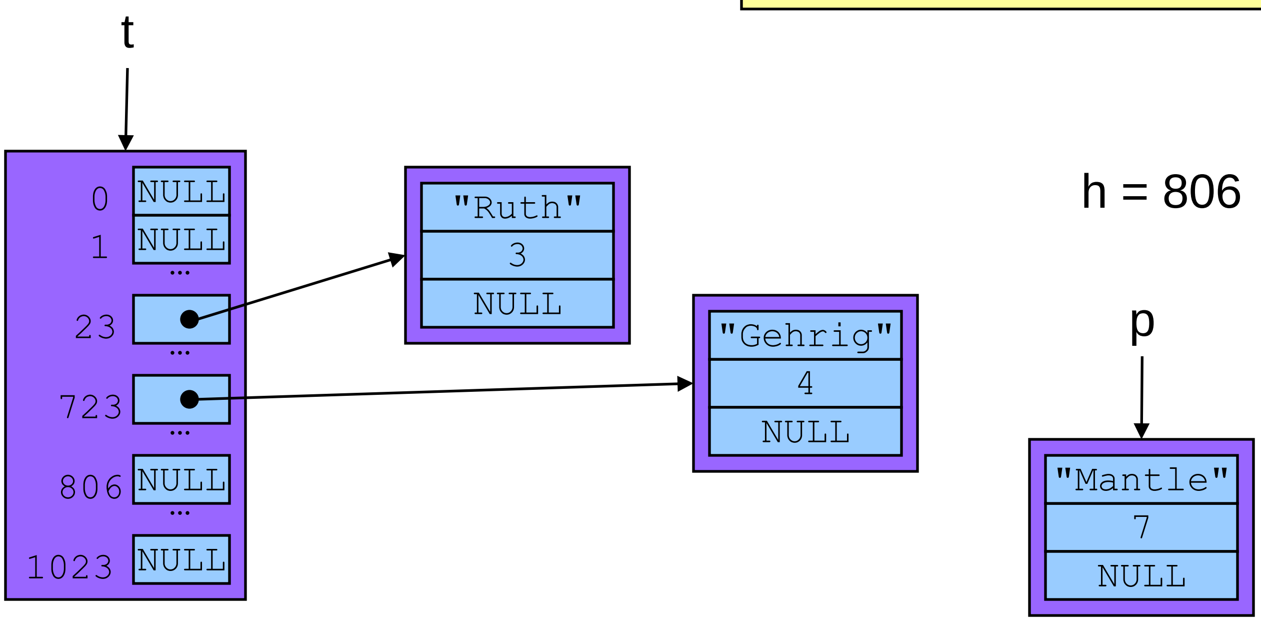


Pretend that “Mantle” hashed to 806, and so h = 806

Hash Table: Add (4)

```
void Table_add(struct Table *t, const char *key, int value)
{
   struct Node *p = (struct Node*)malloc(sizeof(struct Node));
   int h = hash(key);
   p->key = key;
   p->value = value;
   p->next = t->array[h];
   t->array[h] = p;
}
```

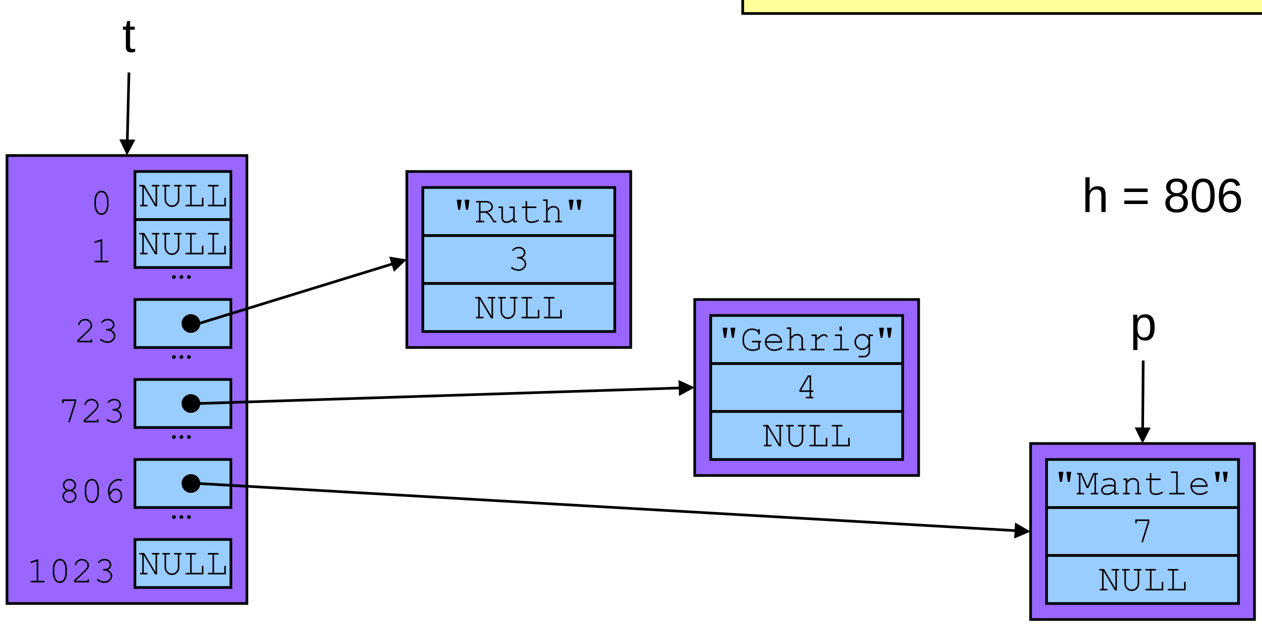
```
struct Table *t;
…
Table_add(t, "Ruth", 3);
Table_add(t, "Gehrig", 4);
Table_add(t, "Mantle", 7);
…
```



Hash Table: Add (5)

```
void Table_add(struct Table *t, const char *key, int value)
{
   struct Node *p = (struct Node*)malloc(sizeof(struct Node));
   int h = hash(key);
   p->key = key;
   p->value = value;
   p->next = t->array[h];
   t->array[h] = p;
}
```

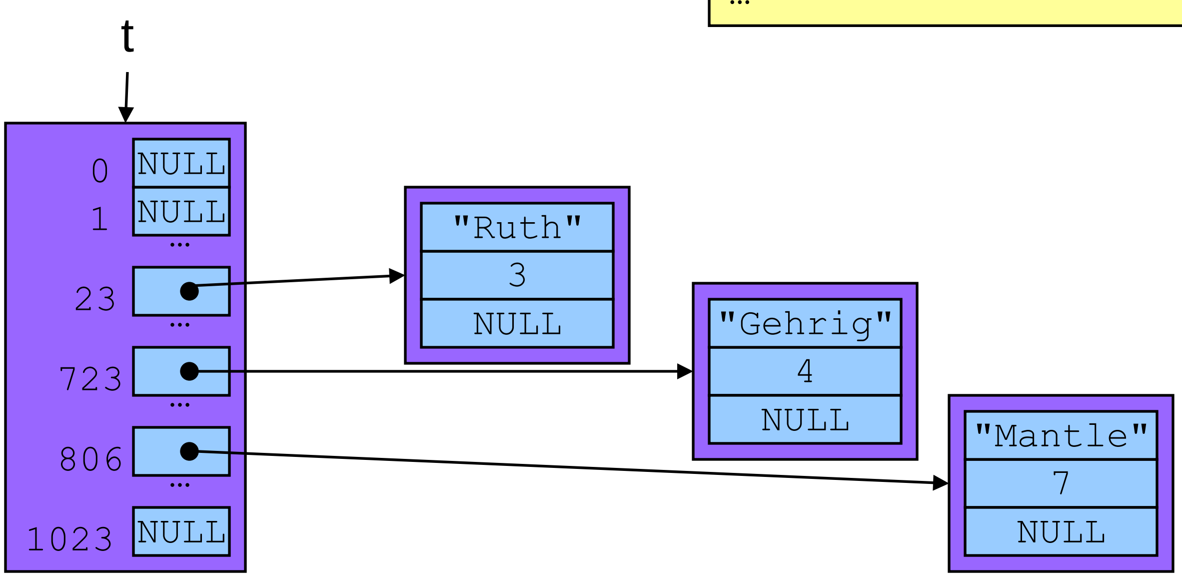
```
struct Table *t;
…
Table_add(t, "Ruth", 3);
Table_add(t, "Gehrig", 4);
Table_add(t, "Mantle", 7);
…
```



Hash Table: Search (1)

```
int Table_search(struct Table *t, const char *key, int *value)
{
   struct Node *p;
   int h = hash(key);
   for (p = t->array[h]; p != NULL; p = p->next)
      if (strcmp(p->key, key) == 0) {
         *value = p->value;
         return 1;
      }
   return 0;
}
```

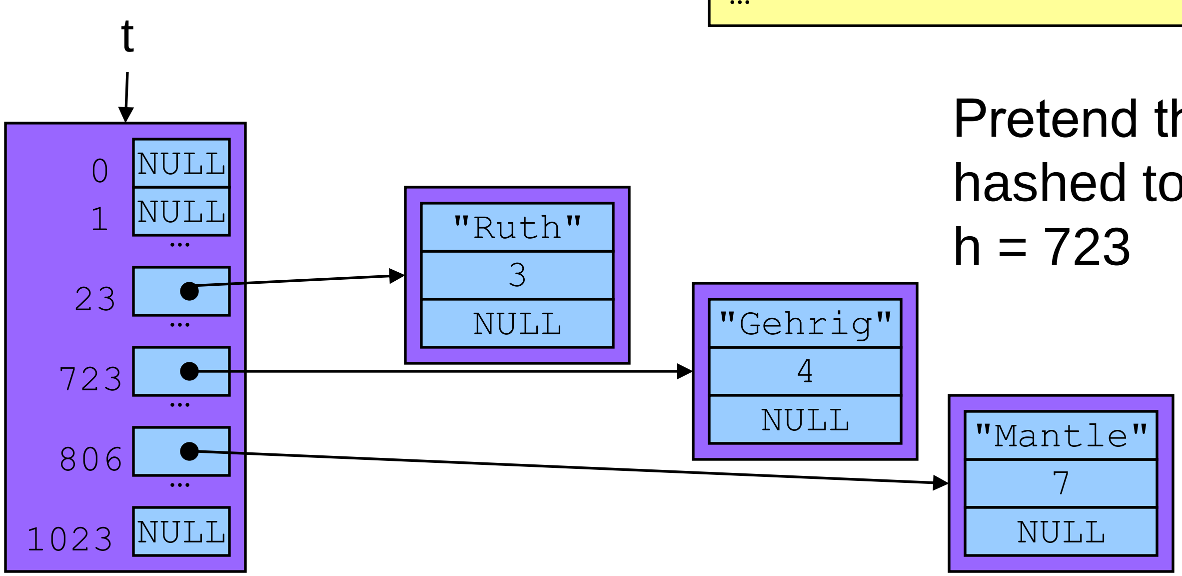
```
struct Table *t;
int value;
int found;
…
found =
   Table_search(t, "Gehrig", &value);
…
```



Hash Table: Search (2)

```
int Table_search(struct Table *t,
   const char *key, int *value) {
   struct Node *p;
   int h = hash(key);
   for (p = t->array[h]; p != NULL; p = p->next)
      if (strcmp(p->key, key) == 0) {
         *value = p->value;
         return 1;
      }
   return 0;
}
```

```
struct Table *t;
int value;
int found;
…
found =
   Table_search(t, "Gehrig", &value);
…
```

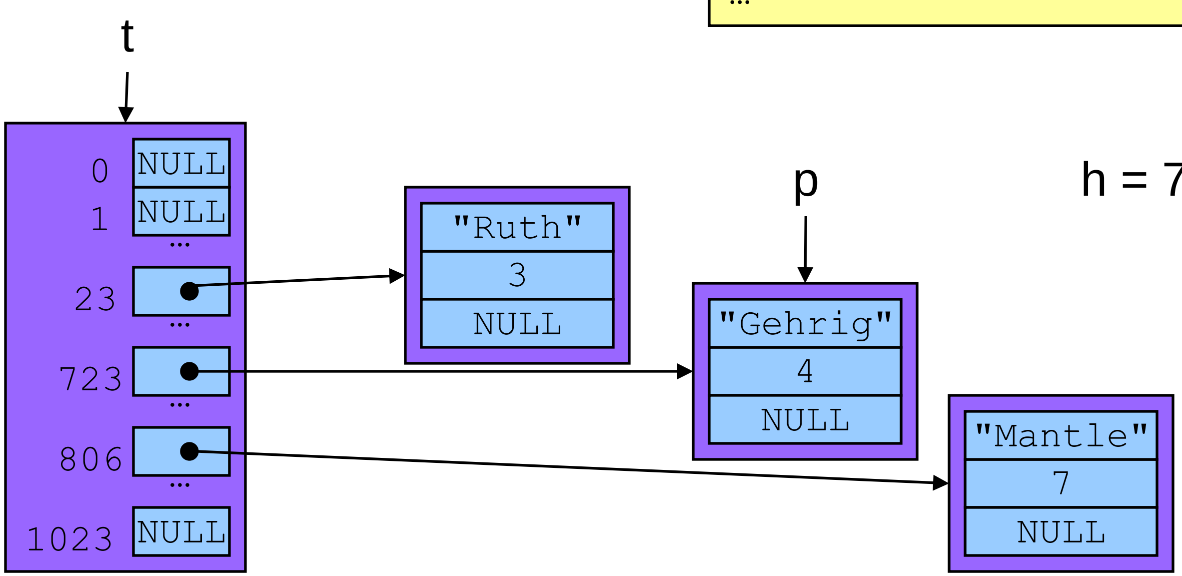


Pretend that “Gehrig” hashed to 723, and so h = 723

Hash Table: Search (3)

```
int Table_search(struct Table *t,
   const char *key, int *value) {
   struct Node *p;
   int h = hash(key);
   for (p = t->array[h]; p != NULL; p = p->next)
      if (strcmp(p->key, key) == 0) {
         *value = p->value;
         return 1;
      }
   return 0;
}
```

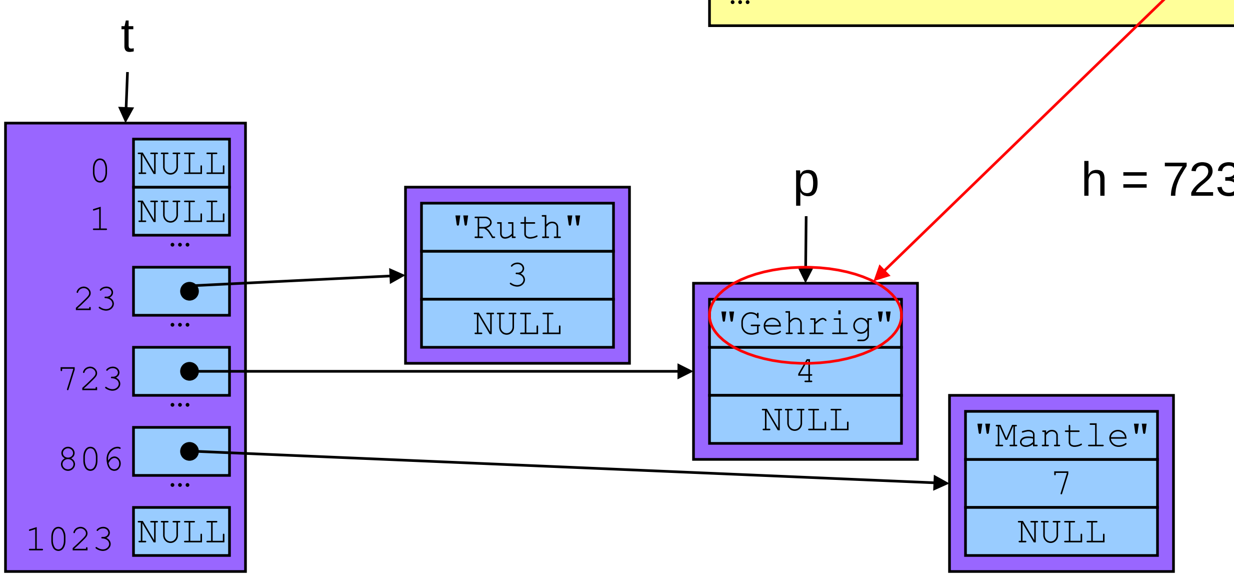
```
struct Table *t;
int value;
int found;
…
found =
   Table_search(t, "Gehrig", &value);
…
```



Hash Table: Search (4)

```
int Table_search(struct Table *t, const char *key, int *value)
{
   struct Node *p;
   int h = hash(key);
   for (p = t->array[h]; p != NULL; p = p->next)
      if (strcmp(p->key, key) == 0) {
         *value = p->value;
         return 1;
      }
   return 0;
}
```

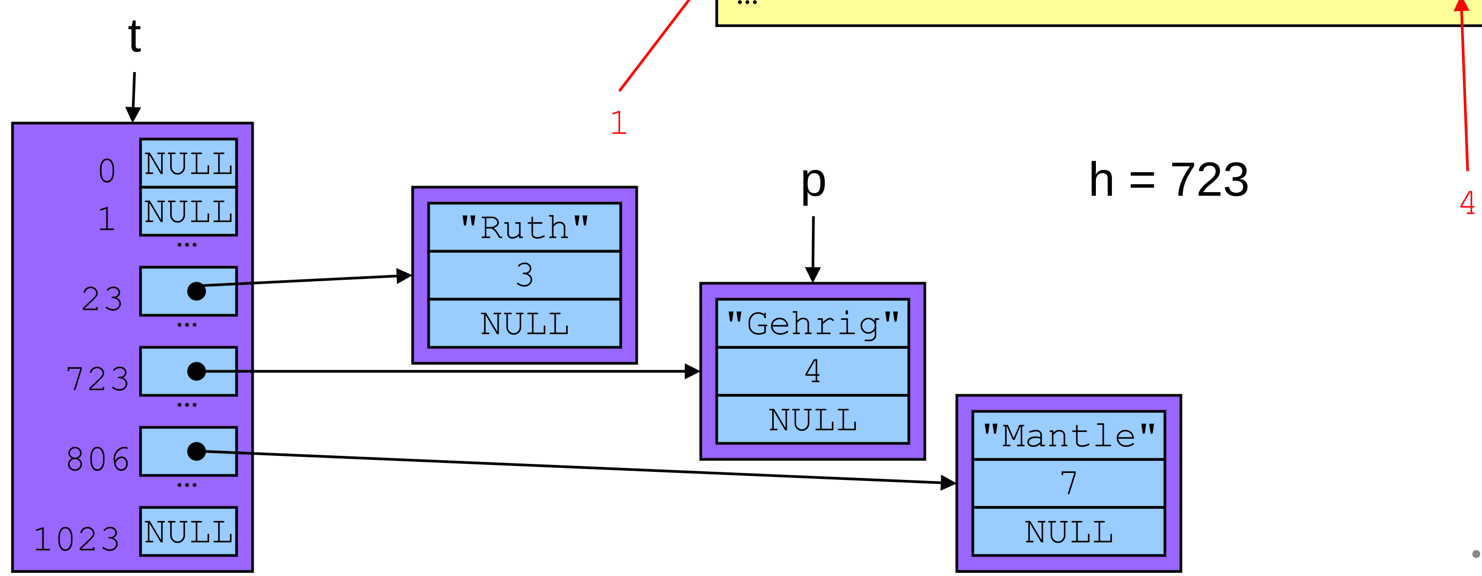
```
struct Table *t;
int value;
int found;
…
found =
   Table_search(t, "Gehrig", &value);
…
```



Hash Table: Search (5)

```
int Table_search(struct Table *t, const char *key, int *value)
{
   struct Node *p;
   int h = hash(key);
   for (p = t->array[h]; p != NULL; p = p->next)
      if (strcmp(p->key, key) == 0) {
         *value = p->value;
         return 1;
      }
   return 0;
}
```

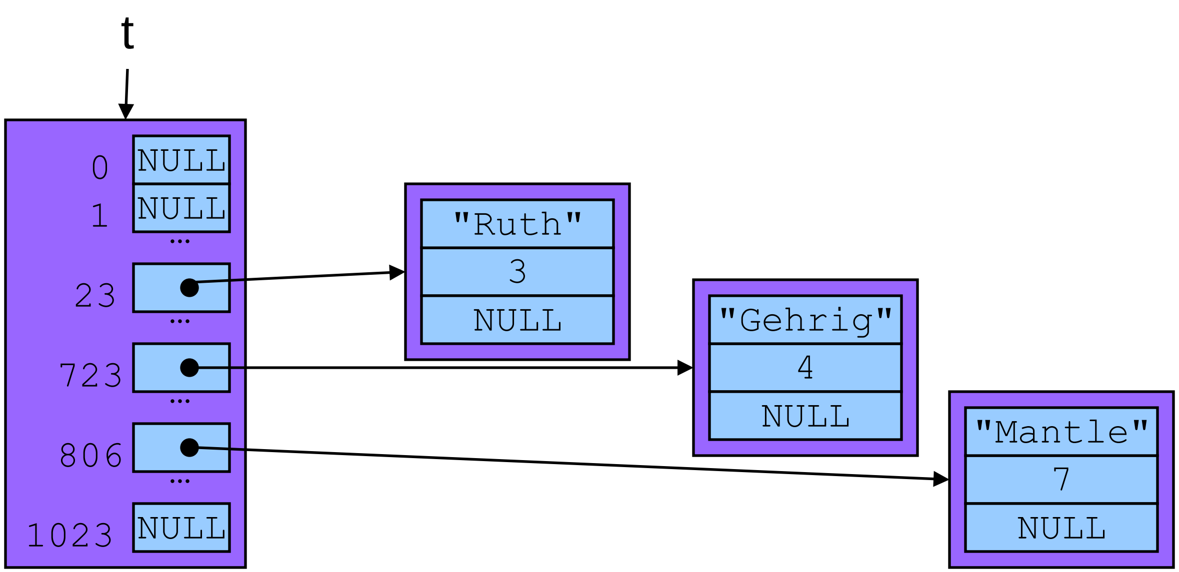
```
struct Table *t;
int value;
int found;
…
found =
   Table_search(t, "Gehrig", &value);
…
```



Hash Table: Free (1)

```
void Table_free(struct Table *t) {
   struct Node *p;
   struct Node *nextp;
   int b;
   for (b = 0; b < BUCKET_COUNT; b++)
      for (p = t->array[b]; p != NULL; p = nextp) {
         nextp = p->next;
         free(p);
      }
   free(t);
}
```

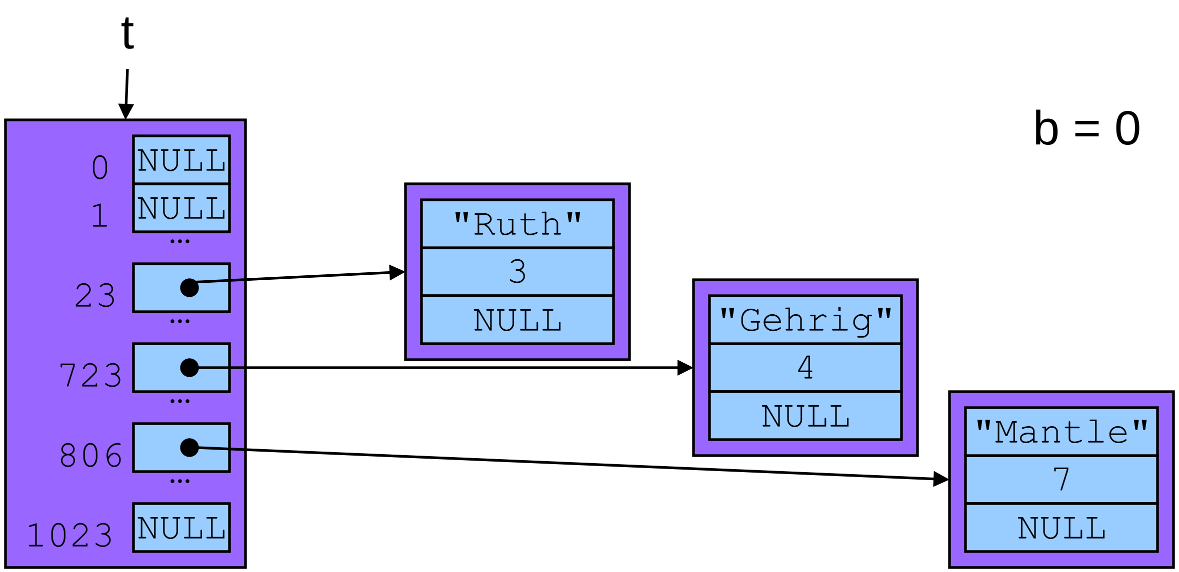
```
struct Table *t;
…
Table_free(t);
…
```



Hash Table: Free (2)

```
void Table_free(struct Table *t) {
   struct Node *p;
   struct Node *nextp;
   int b;
   for (b = 0; b < BUCKET_COUNT; b++)
      for (p = t->array[b]; p != NULL; p = nextp) {
         nextp = p->next;
         free(p);
      }
   free(t);
}
```

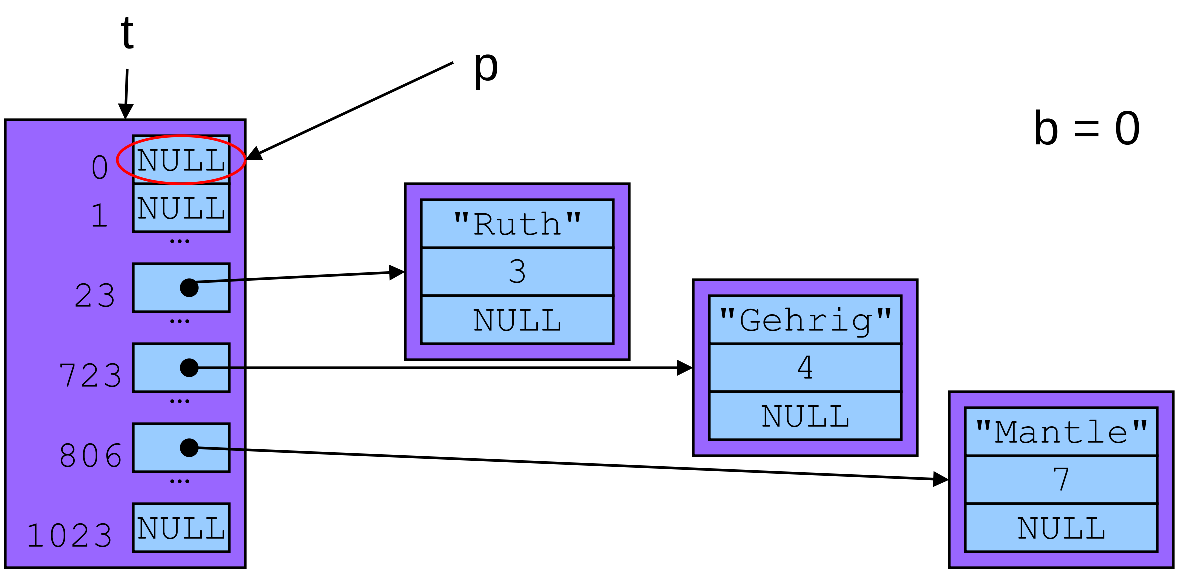
```
struct Table *t;
…
Table_free(t);
…
```



Hash Table: Free (3)

```
void Table_free(struct Table *t) {
   struct Node *p;
   struct Node *nextp;
   int b;
   for (b = 0; b < BUCKET_COUNT; b++)
      for (p = t->array[b]; p != NULL; p = nextp) {
         nextp = p->next;
         free(p);
      }
   free(t);
}
```

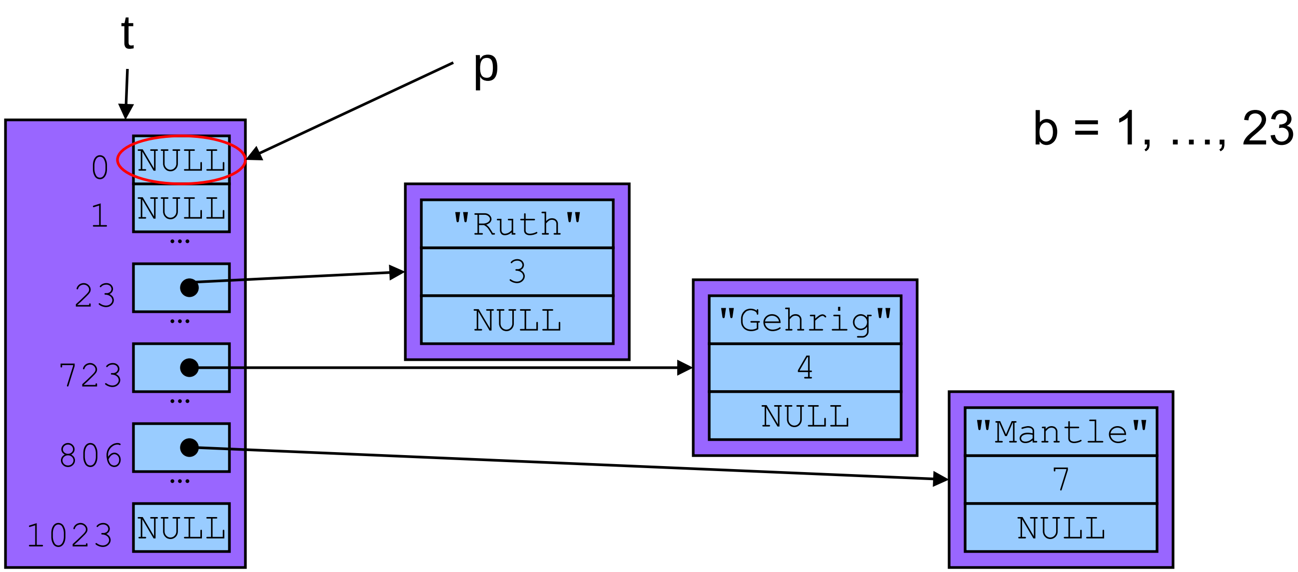
```
struct Table *t;
…
Table_free(t);
…
```



Hash Table: Free (4)

```
void Table_free(struct Table *t) {
   struct Node *p;
   struct Node *nextp;
   int b;
   for (b = 0; b < BUCKET_COUNT; b++)
      for (p = t->array[b]; p != NULL; p = nextp) {
         nextp = p->next;
         free(p);
      }
   free(t);
}
```

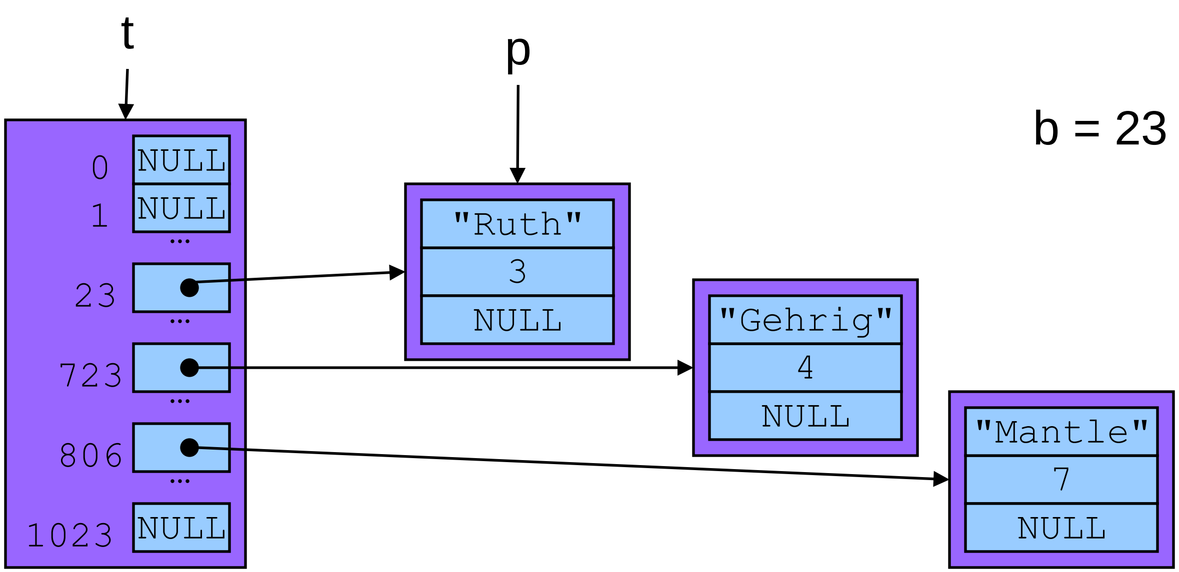
```
struct Table *t;
…
Table_free(t);
…
```



Hash Table: Free (5)

```
void Table_free(struct Table *t) {
   struct Node *p;
   struct Node *nextp;
   int b;
   for (b = 0; b < BUCKET_COUNT; b++)
      for (p = t->array[b]; p != NULL; p = nextp) {
         nextp = p->next;
         free(p);
      }
   free(t);
}
```

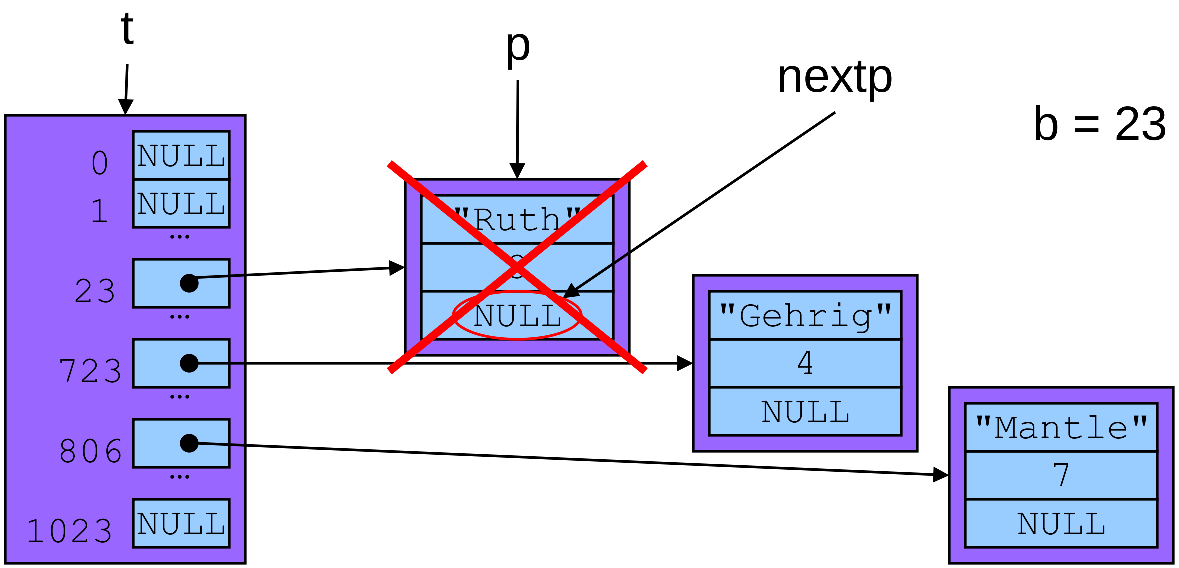
```
struct Table *t;
…
Table_free(t);
…
```



Hash Table: Free (6)

```
void Table_free(struct Table *t) {
   struct Node *p;
   struct Node *nextp;
   int b;
   for (b = 0; b < BUCKET_COUNT; b++)
      for (p = t->array[b]; p != NULL; p = nextp) {
         nextp = p->next;
         free(p);
      }
   free(t);
}
```

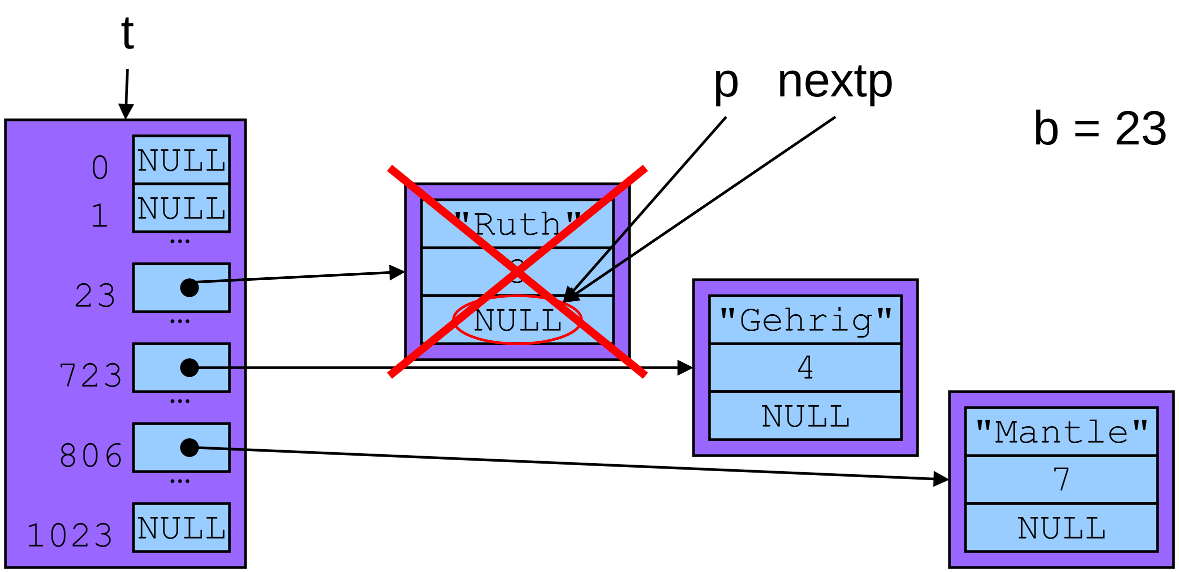
```
struct Table *t;
…
Table_free(t);
…
```



Hash Table: Free (7)

```
void Table_free(struct Table *t) {
   struct Node *p;
   struct Node *nextp;
   int b;
   for (b = 0; b < BUCKET_COUNT; b++)
      for (p = t->array[b]; p != NULL; p = nextp) {
         nextp = p->next;
         free(p);
      }
   free(t);
}
```

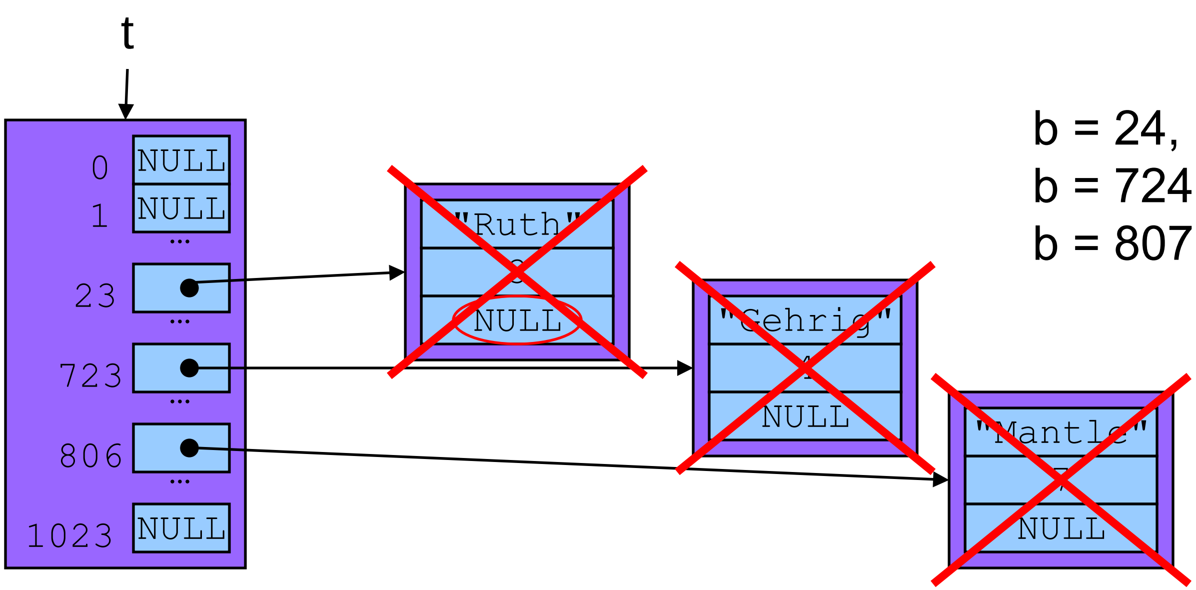
```
struct Table *t;
…
Table_free(t);
…
```



Hash Table: Free (8)

```
void Table_free(struct Table *t) {
   struct Node *p;
   struct Node *nextp;
   int b;
   for (b = 0; b < BUCKET_COUNT; b++)
      for (p = t->array[b]; p != NULL; p = nextp) {
         nextp = p->next;
         free(p);
      }
   free(t);
}
```

```
struct Table *t;
…
Table_free(t);
…
```

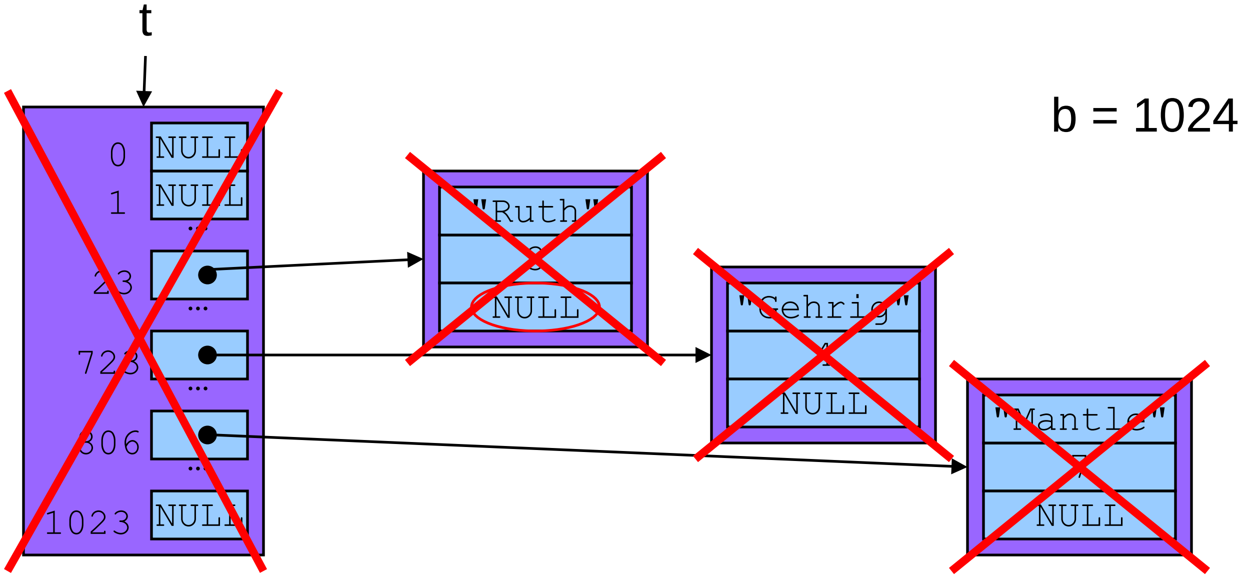


b = 24, …, 723
b = 724, …, 806
b = 807, …, 1024

Hash Table: Free (9)

```
void Table_free(struct Table *t) {
   struct Node *p;
   struct Node *nextp;
   int b;
   for (b = 0; b < BUCKET_COUNT; b++)
      for (p = t->array[b]; p != NULL; p = nextp) {
         nextp = p->next;
         free(p);
      }
   free(t);
}
```

```
struct Table *t;
…
Table_free(t);
…
```



Hash Table Performance

- Create: fast
- Add: fast
- Search: fast
- Free: slow

What are the asymptotic run times (big-oh notation)?

Is hash table search ***always*** fast?

Key Ownership

- Note: Table_add() functions contain this code:

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

  - Caller passes key, which is a pointer to memory where a string resides
  - Table_add() function stores within the table the address where the string resides

Key Ownership (cont.)

- Problem: Consider this calling code:

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

- Via Table_add(), table contains memory address k
- Client changes string at memory address k
- Thus client changes key within table

What happens if the client searches t for “Ruth”?

What happens if the client searches t for “Gehrig”?

Key Ownership (cont.)

- Solution: Table_add() saves **copy** of given key

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

Why add 1?

  – If client changes string at memory add[illegible] structure is not affected
- Then the data structure "owns" the copy, that is:
  – The data structure is responsible for freeing the memory in which the copy resides
  – The Table_free() function must free the copy

Summary

- Common data structures and associated algorithms
  - Linked list
    - fast insert, slow search
  - Hash table
    - Fast insert, (potentially) fast search
    - Invaluable for storing key/value pairs
    - Very common
- Related issues
  - Hashing algorithms
  - Memory ownership

Appendix

- "Stupid programmer tricks" related to hash tables…

Revisiting Hash Functions

- Potentially expensive to compute "mod c"
  - Involves division by c and keeping the remainder
  - Easier when c is a power of 2 (e.g., 16 = 2^4)
- An alternative (by example)
  - 53 = 32 + 16 + 4 + 1

• • •	32	16	8	4	2	1	
0	0	1	1	0	1	0	1

  - 53 % 16 is 5, the last four bits of the number

• • •	32	16	8	4	2	1	
0	0	0	0	0	1	0	1

  - Would like an easy way to isolate the last four bits…

Recall: Bitwise Operators in C

- Bitwise AND (&)

&	0	1
0	0	0
1	0	1

  - Mod on the cheap!
    - E.g., h = 53 & 15;

53	0	0	1	1	0	1	0	1
& 15	0	0	0	0	1	1	1	1
5	0	0	0	0	0	1	0	1

- Bitwise OR (|)

|	0	1
0	0	1
1	1	1

- One's complement (~)
  - Turns 0 to 1, and 1 to 0
  - E.g., set last three bits to 0
    - x = x & ~7;

A Faster Hash Function

```
unsigned int hash(const char *x) {
   int i;
   unsigned int h = 0U;
   for (i=0; x[i]!='\0'; i++)
      h = h * 65599 + (unsigned char)x[i];
   return h % 1024;
}
```

Previous version

```
unsigned int hash(const char *x) {
   int i;
   unsigned int h = 0U;
   for (i=0; x[i]!='\0'; i++)
      h = h * 65599 + (unsigned char)x[i];
   return h & 1023;
}
```

Faster

What happens if you mistakenly write “h & 1024”?

Speeding Up Key Comparisons

- Speeding up key comparisons
  - For any non-trivial value comparison function
  - Trick: store full hash result in structure

```
int Table_search(struct Table *t,
    const char *key, int *value) {
    struct Node *p;
    int h = hash(key); /* No % in hash function */
    for (p = t->array[h%1024]; p != NULL; p = p->next)
        if ((p->hash == h) && strcmp(p->key, key) == 0) {
            *value = p->value;
            return 1;
        }
    return 0;
}
```

Why is this so much faster?