



Hash Tables

Prof. David August
COS 217



Goals of Today's Lecture

- Motivation for hash tables
 - Examples of (key, value) pairs
 - Limitations of using arrays
 - Example using a linked list
 - Inefficiency of using a linked list
- Hash tables
 - Hash table data structure
 - Hash function
 - Example hashing code
 - Who owns the keys?
- Implementing "mod" efficiently
 - Binary representation of numbers
 - Logical bit operators



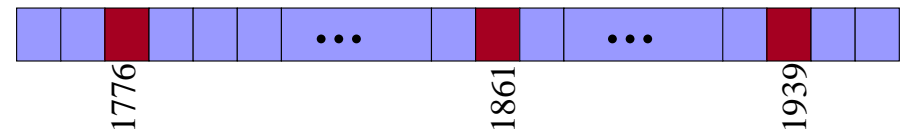
Accessing Data By a Key

- Student grades: (name, grade)
 - E.g., ("john smith", 84), ("jane doe", 93), ("bill clinton", 81)
 - Gradeof("john smith") returns 84
 - Gradeof("joe schmoe") returns NULL
- Wine inventory: (name, #bottles)
 - E.g., ("tapestry", 3), ("latour", 12), ("margeaux", 3)
 - Bottlesof("latour") returns 12
 - Bottlesof("giesen") returns NULL
- Years when a war started: (year, war)
 - E.g., (1776, "Revolutionary"), (1861, "Civil War"), (1939, "WW2")
 - Warstarted(1939) returns "WW2"
 - Warstarted(1984) returns NULL
- Symbol table: (variable name, variable value)
 - E.g., ("MAXARRAY", 2000), ("FOO", 7), ("BAR", -10)



Limitations of Using an Array

- Array stores n values indexed 0, ..., n-1
 - Index is an integer
 - Max size must be known in advance
- But, the key in a (key, value) pair might not be a number
 - Well, could convert it to a number
 - E.g., have a separate number for each possible name
- But, we'd need an extremely large array
 - Large number of possible keys (e.g., all names, all years, etc.)
 - And, the number of unique keys might even be unknown
 - And, most of the array elements would be empty



Could Use an Array of (key, value)



- Alternative way to use an array

- Array element i is a struct that stores key and value

0	1776	Revolutionary
1	1861	Civil
2	1939	WW2

- Managing the array

- Add an elements: add to the end
- Remove an element: find the element, and copy last element over it
- Find an element: search from the beginning of the array

- Problems

- Allocating too little memory: run out of space
- Allocating too much memory: wasteful of space

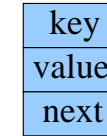
5

Linked List to Adapt Memory Size



- Each element is a struct

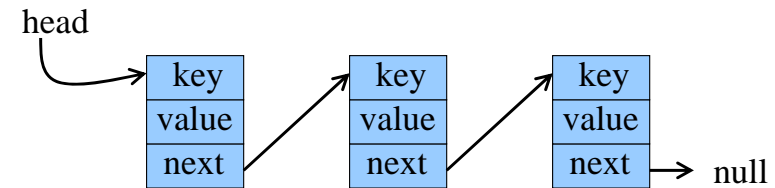
- Key
- Value
- Pointer to next element



```
struct Entry {
    int key;
    char* value;
    struct Entry *next;
};
```

- Linked list

- Pointer to the first element in the list
- Functions for adding and removing elements
- Function for searching for an element with a particular key



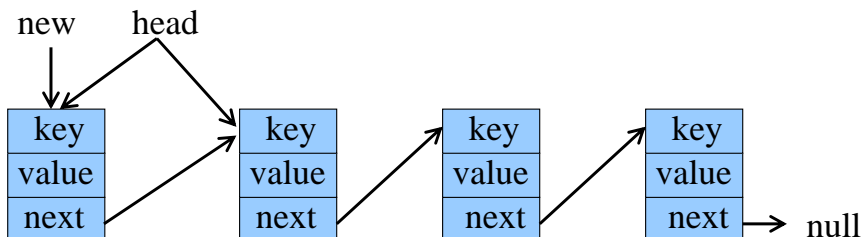
6

Adding Element to a List



- Add new element at front of list

- Make ptr of new element point the current first element
 - `new->next = head;`
- Make the head of the list point to the new element
 - `head = new;`



7

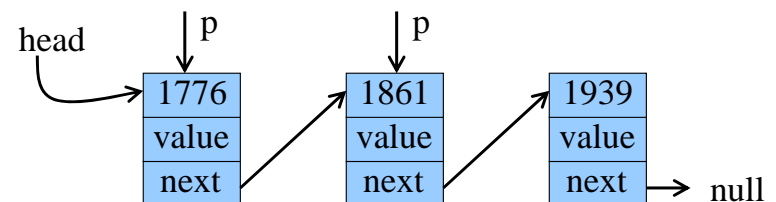
Locating an Element in a List



- Sequence through the list by key value

- Return pointer to the element
- ... or NULL if no element is found

```
for (p = head; p!=NULL; p=p->next) {
    if (p->key == 1861)
        return p;
}
return NULL;
```



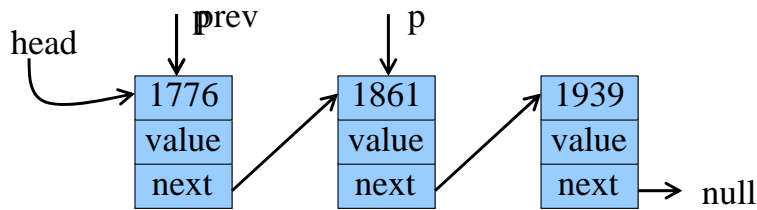
8

Locate and Remove an Element (1)



- Sequence through the list by key value
 - Keep track of the previous element in the list

```
prev = NULL;
for (p = head; p!=NULL; prev=p, p=p->next){
    if (p->key == 1861) {
        delete the element (see next slide!);
        break;
    }
}
```



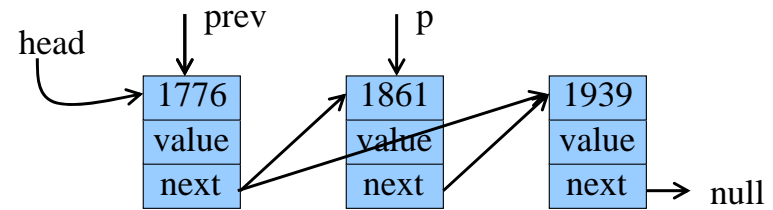
9

Locate and Remove an Element (2)



- Delete the element
 - Head element: make head point to the second element
 - Non-head element: make previous Entry point to next element

```
if (p == head)
    head = head->next;
else
    prev->next = p->next;
```



10

List is Not Good for (key, value)



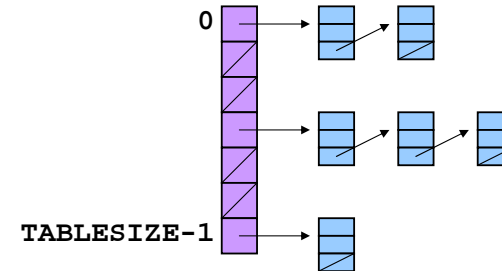
- Good place to start
 - Simple algorithm and data structure
 - Good to allow early start on design and test of client code
- But, testing might show that this is not efficient enough
 - Removing or locating an element
 - Requires walking through the elements in the list
 - Could store elements in sorted order
 - But, keeping them in sorted order is time consuming
 - And, searching by key in the sorted list still takes time
- Ultimately, we need a better approach
 - Memory efficient: adds extra memory as needed
 - Time efficient: finds element by its key instantly (or nearly)

11

Hash Table



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



```
struct Entry *hashtab[TABLESIZE];
```

- Function mapping each key to an array index
 - For example, for an integer key h
 - Hash function: $i = h \% \text{TABLESIZE}$ (mod function)
 - Go to array element i , i.e., the linked list `hashtab[i]`
 - Search for element, add element, remove element, etc.

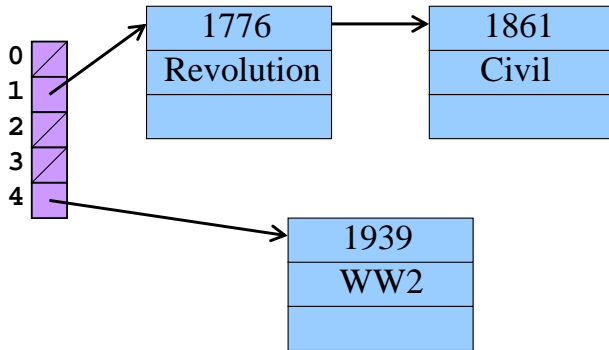
12

Example



- Array of size 5 with hash function “ $h \bmod 5$ ”

- “ $1776 \bmod 5$ ” is 1
- “ $1861 \bmod 5$ ” is 1
- “ $1939 \bmod 5$ ” is 4



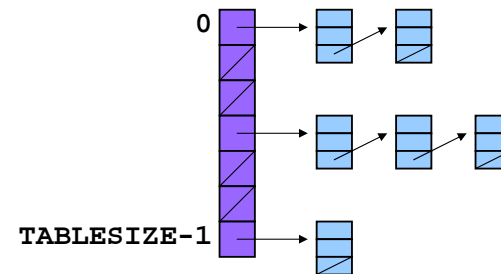
13

How Large an Array?



- Large enough that average “bucket” size is 1
 - Short buckets mean fast look-ups
 - Long buckets mean slow look-ups
- Small enough to be memory efficient
 - Not an excessive number of elements
 - Fortunately, each array element is just storing a pointer

- This is OK:



14

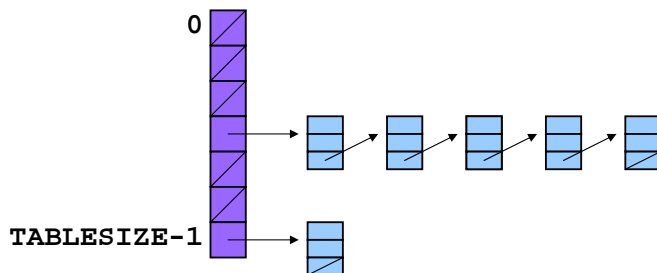
What Kind of Hash Function?



- Good at distributing elements across the array

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

- This is not so good:



15

Hashing String Keys to Integers



- Simple schemes don't distribute the keys evenly enough

- Number of characters, mod TABLESIZE
- Sum the ASCII values of all characters, mod TABLESIZE
- ...

- Here's a reasonably good hash function

- Weighted sum of characters x_i in the string
 - $(\sum a^i x_i) \bmod \text{TABLESIZE}$
- Best if a and TABLESIZE are relatively prime
 - E.g., $a = 65599$, TABLESIZE = 1024

16

Implementing Hash Function



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

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

Can be more clever than this for powers of two!

17

Hash Table Example



Example: TABLESIZE = 7

Lookup (and enter, if not present) these strings: the, cat, in, the, hat
Hash table initially empty.

First word: the. $\text{hash}(\text{"the"}) = 965156977$. $965156977 \% 7 = 1$.

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



18

Hash Table Example



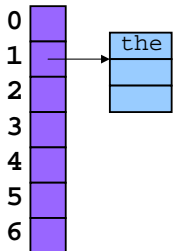
Example: TABLESIZE = 7

Lookup (and enter, if not present) these strings: the, cat, in, the, hat
Hash table initially empty.

First word: "the". $\text{hash}(\text{"the"}) = 965156977$. $965156977 \% 7 = 1$.

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

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



19

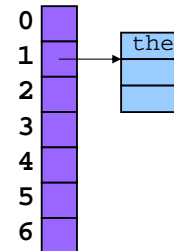
Hash Table Example



Second word: "cat". $\text{hash}(\text{"cat"}) = 3895848756$. $3895848756 \% 7 = 2$.

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

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

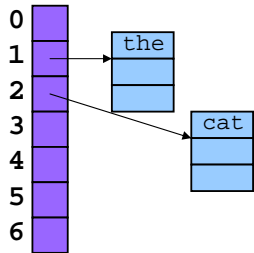


20

Hash Table Example



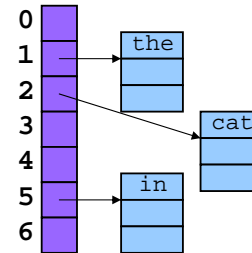
Third word: "in". $\text{hash}(\text{"in"}) = 6888005$. $6888005 \% 7 = 5$.
Search the linked list `table[5]` for the string "in"; not found
Now: `table[5] = makelink(key, value, table[5])`



Hash Table Example



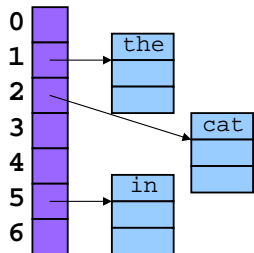
Fourth word: "the". $\text{hash}(\text{"the"}) = 965156977$. $965156977 \% 7 = 1$.
Search the linked list `table[1]` for the string "the"; found it!



Hash Table Example



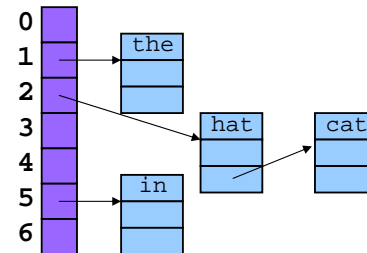
Fourth word: "hat". $\text{hash}(\text{"hat"}) = 865559739$. $865559739 \% 7 = 2$.
Search the linked list `table[2]` for the string "hat"; not found.
Now, insert "hat" into the linked list `table[2]`.
At beginning or end? Doesn't matter.



Hash Table Example



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



Example Hash Table C Code



- Element in the hash table

```
struct Nlist {
    struct Nlist *next;
    char *key;
    char *value;
};
```

- Hash table

- struct Nlist *hashtab[1024];

- Three functions

- Hash function: unsigned hash(char *x)
- Look up with key: struct Nlist *lookup(char *s)
- Install entry: struct Nlist *install(char *key, *value)

25

Lookup Function



- Lookup based on key

- Key is a string *s
- Return pointer to matching hash-table element
- ... or return NULL if no match is found

```
struct Nlist *lookup(char *s) {
    struct Nlist *p;

    for (p = hashtab[hash(s)]; p!=NULL; p=p->next)
        if (strcmp(s, p->key) == 0)
            return p; /* found */
    return NULL; /* not found */
}
```

26

Install an Entry (1)



- Install and (key, value) pair

- Add new Entry if none exists, or overwrite the old value
- Return a pointer to the Entry

```
struct Nlist *install(char *key, char *value) {
    struct Nlist *p;

    if ((p = lookup(name)) == NULL) { /* not found */
        create and add new Entry (see next slide);
    } else /* already there, so discard old value */
        free(p->value);
    p->value = malloc(strlen(value) + 1);
    assert(p->value != NULL);
    strcpy(p->value, value);
    return p;
}
```

27

Install an Entry (2)



- Create and install a new Entry

- Allocate memory for the new struct and the key
- Insert into the appropriate linked list in the hash table

```
p = malloc(sizeof(*p));
assert(p != NULL);
p->key = malloc(strlen(key) + 1);
assert(p->key != NULL);
strcpy(p->key, key);

/* add to front of linked list */
unsigned hashval = hash(key);
p->next = hashtab[hashval];
hashtab[hashval] = p;
```

28

Why Bother Copying the Key?



- In the example, why did I do

```
p->key = malloc(strlen(key) + 1);
strcpy(p->key, key);
```

- Instead of simply

```
p->key = key;
```

- After all, the client passed me **key**, which is a *pointer*
 - So, storage for the key has already been allocated
 - Don't I simply need to copy the *address* where the string is stored?
- I want to preserve the integrity of the hash table
 - Even if the client program ultimately "frees" the memory for key
 - So, the install function makes a copy of the key
- The hash table *owns* the key
 - ... because it is part of the data structure

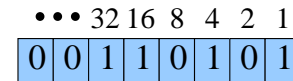
Revisiting Hash Functions



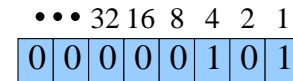
- Potentially expensive to compute "mod c"
 - Involves division by c and keeping the remainder
 - Easier when c is a power of 2 (e.g., $16 = 2^4$)

- Binary (base 2) representation of numbers

- E.g., $53 = 32 + 16 + 4 + 1$



- E.g., $53 \% 16$ is 5, the last four bits of the number



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

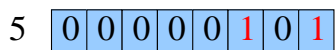
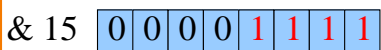
Bitwise Operators in C



- Bitwise AND (&)

&	0	1
0	0	0
1	0	1

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



- 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
 - $x = x \& \sim 7$;

Bitwise Operators in C (Continued)



- Shift left (<<)

- Shift some # of bits to the left, filling the blanks with 0
- E.g., $n \ll 2$ shifts left by 2 bits
 - So, if n is 101_2 (i.e., 5_{10}), then $n \ll 2$ is 10100_2 (i.e., 20_{10})
- Multiplication by powers of two on the cheap!

- Shift right (>>)

- Shift some # of bits to the right
 - For unsigned integer, fill in blanks with 0
 - What about signed integers?
 - Can vary from one machine to another!
- E.g., $n \gg 2$ shifts right by 2 bits
 - So, if n is 10110_2 (i.e., 22_{10}), then $n \gg 2$ is 101_2 (i.e., 5_{10})
- Division by powers of two (dropping remainder) on the cheap!

Stupid Programmer Tricks



- Confusing (val % 1024) with (val & 1024)
 - Drops from 1024 bins to *two* useful bins
 - You really wanted (val & 1023)
- Speeding up compare
 - For any non-trivial value comparison function
 - Trick: store full hash result in structure

```
struct Nlist *lookup(char *s) {
    struct Nlist *p;
    int val = hash(s); /* no % in hash function */

    for (p = hashtab[val%1024]; p!=NULL; p=p->next)
        if (p->hash == val && strcmp(s, p->key) == 0)
            return p;
    return NULL;
}
```

33

Summary of Today's Lecture



- Linked lists
 - A list is always the size it needs to be to store its contents
 - Useful when the number of items may change frequently!
 - A list can be rearranged simply by manipulating pointers
 - When items are added/deleted, other items aren't moved
 - Useful when items are large and, hence, expensive to move!
- Hash tables
 - Invaluable for storing (key, value) pairs
 - Very efficient lookups
 - If the hash function is good and the table size is large enough
- Bit-wise operators in C
 - AND (&) and OR (|) – note: they are different from && and ||
 - One's complement (~) to flip all bits
 - Left shift (<<) and right shift (>>) by some number of bits

34