Hash Tables

COS 217
Goals of Today’s Lecture

• Motivation for hash tables
  o Examples of (key, value) pairs
  o Limitations of using arrays
  o Example using a linked list
  o Inefficiency of using a linked list

• Hash tables
  o Hash table data structure
  o Hash function
  o Example hashing code
  o Who owns the keys?

• Implementing “mod” efficiently
  o Binary representation of numbers
  o 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(“joeschmoe”) 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, \( \ldots, n-1 \)
  - 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

<table>
<thead>
<tr>
<th></th>
<th>1776</th>
<th>Revolutionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1776</td>
<td>Revolutionary</td>
</tr>
<tr>
<td>1</td>
<td>1861</td>
<td>Civil</td>
</tr>
<tr>
<td>2</td>
<td>1939</td>
<td>WW2</td>
</tr>
</tbody>
</table>

- **Managing the array**
  - Add an element: 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
Linked List to Adapt Memory Size

- Each element is a struct
  - Key
  - Value
  - Pointer to next element

- 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

```c
struct entry {
  int key;
  char* value;
  struct entry *next;
};
```

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

```c
for (p = head; p!=NULL; p=p->next) {
    if (p->key == 1861)
        return p;
}
return NULL;
```
Locate and Remove an Element (1)

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

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

```
head  prev  p
     1776 1861 1939
   value value value
   next  next  next
         null
```
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

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

![Diagram showing the deletion of an element in a linked list.](image-url)
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)
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: \( i = h \%\) TABLESIZE (mod function)
  - Go to array element \( i \), i.e., the linked list \( \text{hashtab}[i] \)
    - Search for element, add element, remove element, etc.
Example

• Array of size 5 with hash function “h mod 5”
  o “1776 % 5” is 1
  o “1861 % 5” is 1
  o “1939 % 5” is 4
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:
What Kind of Hash Function?

- Good at distributing elements across the array
  - Distribute results over the range 0, 1, …, TABLESIZE-1
  - Distribute results \textit{evenly} to avoid very long buckets

- This is not so good:
• Simple schemes don’t distribute the keys evenly enough
  o Number of characters, mod TABLESIZE
  o Sum the ASCII values of all characters, mod TABLESIZE
  o …

• Here’s a reasonably good hash function
  o Weighted sum of characters $x_i$ in the string
    – $(\sum a^i x_i) \mod$ TABLESIZE
  o Best if $a$ and TABLESIZE are relatively prime
    – E.g., $a = 65599$, TABLESIZE = 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 $(((x[0] \times 65599 + x[1]) \times 65599 + x[2]) \times 65599 + x[3]) \times \ldots$

```c
unsigned hash(char *x) {
    int i; unsigned 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!
Example: TABLESIZE = 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

Example: TABLESIZE = 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

Second word: “cat”. \( \text{hash(“cat”) = 3895848756.} \quad 3895848756 \mod 7 = 2. \)

Search the linked list \( \text{table[2]} \) for the string “cat”; not found

Now: \( \text{table[2] = makelink(key, value, table[2])} \)
Hash Table Example

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])
Fourth word: “the”. \( \text{hash(“the”)} = 965156977. \) \( 965156977 \mod 7 = 1. \)

Search the linked list \( \text{table}[1] \) for the string “the”; found it!
Fourth word: “hat”. \( \text{hash(“hat”)} = 865559739. \) \( 865559739 \mod 7 = 2. \)

Search the linked list \( \text{table[2]} \) for the string “hat”; not found.

Now, insert “hat” into the linked list \( \text{table[2]} \).

At beginning or end? Doesn’t matter.
Inserting at the front is easier, so add “hat” at the front
Example Hash Table C Code

- Element in the hash table

```c
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)`
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 */
}
```
Install an Entry (1)

• Install and (key, value) pair
  o Add new entry if none exists, or overwrite the old value
  o Return a pointer to the entry

```c
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((void *) p->value);
    if ((p->value = strdup(value)) == NULL)
        return NULL;   /* failure in copying string */
    return p;
}
```
Install an Entry (2)

• Create and install a new entry
  o Allocate memory for the new struct and the key
  o Insert into the appropriate linked list in the hash table

```
p = (struct nlist *) malloc(sizeof(*p));
if ((p == NULL) || (p->key = strdup(key)) == NULL))
    return NULL; /* failure to allocate memory */

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

• In the example, why did I do
  \[
  p->key = strdup(key);
  \]

• Instead of simply
  \[
  p->key = key;
  \]

• After all, the client passed me \texttt{key}, which is a \textit{pointer}
  \begin{itemize}
    \item So, storage for the key has already been allocated
    \item Don’t I simply need to copy the \textit{address} where the string is stored?
  \end{itemize}

• I want to preserve the integrity of the hash table
  \begin{itemize}
    \item Even if the client program ultimately “frees” the memory for key
    \item So, the install function makes a copy of the key
  \end{itemize}

• The hash table \textit{owns} the key
  \begin{itemize}
    \item … because it is part of the data structure
Revisiting Hash Functions

• Potentially expensive to compute “mod c”
  o Involves division by c and keeping the remainder
  o Easier when c is a power of 2 (e.g., 16 = 2⁴)

• Binary (base 2) representation of numbers
  o E.g., \(53 = 32 + 16 + 4 + 1\)

\[
\begin{array}{cccccc}
\cdot & \cdot & \cdot & 32 & 16 & 8 & 4 & 2 & 1 \\
0 & 0 & 1 & 1 & 0 & 1 & 0 & 1
\end{array}
\]

  o E.g., 53 % 16 is 5, the last four bits of the number

\[
\begin{array}{cccccc}
\cdot & \cdot & \cdot & 32 & 16 & 8 & 4 & 2 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1
\end{array}
\]

  o Would like an easy way to isolate the last four bits…
Bitwise Operators in C

• Bitwise AND (&)

\[
\begin{array}{c|cc}
\& & 0 & 1 \\
\hline
0 & 0 & 0 \\
1 & 0 & 1 \\
\end{array}
\]

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

\[
\begin{array}{cc}
53 & 00110101 \\
& 0000011111 \\
\hline
5 & 00000101 \\
\end{array}
\]

• Bitwise OR (|)

\[
\begin{array}{cc}
| & 0 & 1 \\
\hline
0 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\]

• One’s complement (~)

- Turns 0 to 1, and 1 to 0
  - E.g., set last three bits to 0
    - \( x = x \& \sim 7; \)
• Shift left (<<)
  o Shift some # of bits to the left, filling the blanks with 0
  o E.g., \( n << 2 \) shifts left by 2 bits
    – So, if \( n \) is \( 101_2 \) (i.e., \( 5_{10} \)), then \( n << 2 \) is \( 10100_2 \) (i.e., \( 20_{10} \))
  o Multiplication by powers of two on the cheap!

• Shift right (>>)
  o 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!
  o E.g., \( n >> 2 \) shifts right by 2 bits
    – So, if \( n \) is \( 10110_2 \) (i.e., \( 22_{10} \)), then \( n >> 2 \) is \( 101_2 \) (i.e., \( 5_{10} \))
  o Division by powers of two (dropping remainder) on the cheap!
Stupid Programmer Tricks

• Confusing (val % 1024) with (val & 1024)
  o Drops from 1024 bins to two useful bins
  o You really wanted (val & 1023)

• Speeding up compare
  o For any non-trivial value comparison function
  o Trick: store full hash result in structure

```c
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;
}
```
Summary of Today’s Lecture

• Linked lists
  o A list is always the size it needs to be to store its contents
    – Useful when the number of items may change frequently!
  o 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
  o Invaluable for storing (key, value) pairs
  o Very efficient lookups
    – If the hash function is good and the table size is large enough

• Bit-wise operators in C
  o AND (&) and OR (|) – note: they are different from && and ||
  o One’s complement (~) to flip all bits
  o Left shift (<<) and right shift (>>) by some number of bits