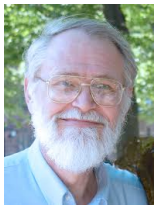
Data Structures

Motivating Quotation

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

– Kernighan & Pike



"Programming in the Large" Steps

Design & Implement
- Program & programming style (done)
- Common data structures and algorithms <-- we are here
- Modularity
- Building techniques & tools (done)

Debug
- Debugging techniques & tools (done)

Test
- Testing techniques (done)

Maintain
- Performance improvement techniques & tools

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 power programmer:
  - Rarely creates programs from scratch
  - Often creates programs using high level building blocks

Why? Shallow motivation:
- Provide background pertinent to Assignment 3
- … esp. for those who have not taken COS 226

Common Task

Maintain a collection of key/value pairs
- Each key is a **string**; each value is an **int**
- Unknown number of key-value pairs

Examples
- (student name, grade)
  - ("john smith", 84), ("jane doe", 93), ("bill clinton", 81)
- (baseball player, number)
  - ("Ruth", 3), ("Gehrig", 4), ("Mantle", 7)
- (variable name, value)
  - ("maxLength", 2000), ("i", 7), ("j", -10)

Agenda

Linked lists

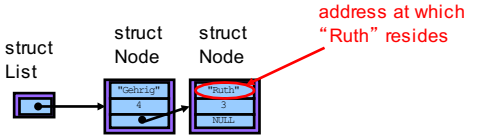
Hash tables

Hash table issues

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 elaborate



7

Linked List Algorithms

Create
- Allocate `List` structure; set `first` to `NULL`
- Performance: O(1) => fast

Add (no check for duplicate key required)
- Insert new node containing key/value pair at front of list
- Performance: O(1) => 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) => slow

8

Linked List Algorithms

Search
- Traverse the list, looking for given key
- Stop when key found, or reach end
- Performance: O(n) => slow

Free
- Free `Node` structures while traversing
- Free `List` structure
- Performance: O(n) => slow

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

9

Agenda

Linked lists

Hash tables

Hash table issues

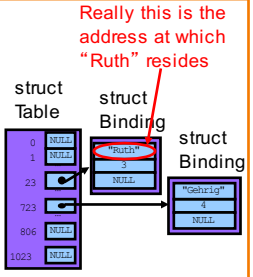
10

Hash Table Data Structure

Array of linked lists

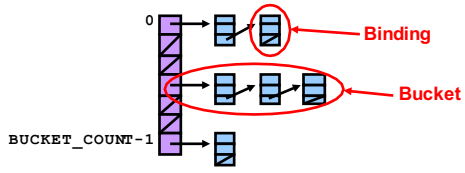
```
enum {BUCKET_COUNT = 1024};

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

struct Table
{  struct Binding *buckets[BUCKET_COUNT];
};
```

Your Assignment 3 data structures will be more elaborate



11

Hash Table Data Structure



Hash function maps given key to an integer

Mod integer by `BUCKET_COUNT` to determine proper bucket

12

Hash Table Example

Example: `BUCKET_COUNT` = 7

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

- the, cat, in, the, hat

13

Hash Table Example (cont.)

First key: "the"

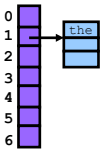
- hash("the") = 965156977; 965156977 % 7 = 1

Search `buckets[1]` for binding with key "the"; not found

14

Hash Table Example (cont.)

Add binding with key "the" and its value to `buckets[1]`



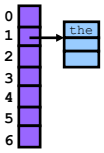
15

Hash Table Example (cont.)

Second key: "cat"

- hash("cat") = 3895848756; 3895848756 % 7 = 2

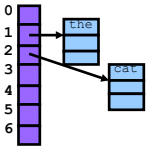
Search `buckets[2]` for binding with key "cat"; not found



16

Hash Table Example (cont.)

Add binding with key "cat" and its value to `buckets[2]`



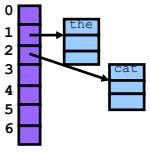
17

Hash Table Example (cont.)

Third key: "in"

- hash("in") = 6888005; 6888005% 7 = 5

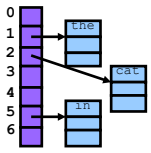
Search `buckets[5]` for binding with key "in"; not found



18

Hash Table Example (cont.)

Add binding with key "in" and its value to `buckets[5]`



19

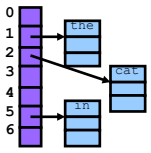
Hash Table Example (cont.)

Fourth word: "the"

- hash("the") = 965156977; 965156977 % 7 = 1

Search `buckets[1]` for binding with key "the"; found it!

- Don't change hash table



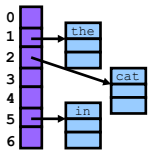
20

Hash Table Example (cont.)

Fifth key: "hat"

- hash("hat") = 865559739; 865559739 % 7 = 2

Search `buckets[2]` for binding with key "hat"; not found

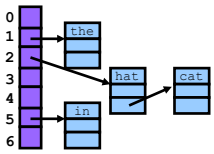


21

Hash Table Example (cont.)

Add binding with key "hat" and its value to `buckets[2]`

- At front or back? Doesn't matter
- Inserting at the front is easier, so add at the front



22

Hash Table Algorithms

Create

- Allocate `Table` structure; set each bucket to `NULL`
- Performance: O(1) => 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: O(1) => fast

Is the add performance always fast?

23

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: O(1) => fast

Is the search performance always fast?

Free

- Traverse each bucket, freeing bindings
- Free `Table` structure
- Performance: O(n) => slow

24

Agenda

Linked lists

Hash tables

Hash table issues

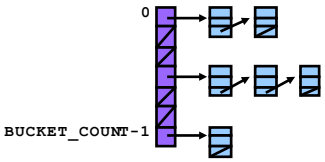
25

How Many Buckets?

Many!
- Too few => large buckets => slow add, slow search

But not too many!
- Too many => memory is wasted

This is OK:

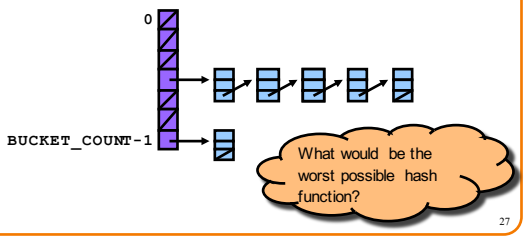


26

What Hash Function?

Should distribute bindings across the buckets well
- Distribute bindings over the range `0, 1, …, BUCKET_COUNT-1`
- Distribute bindings *evenly* to avoid very long buckets

This is not so good:



27

How to Hash Strings?

Simple hash schemes don't distribute the keys evenly enough
- 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`
  - $(\Sigma\ a^i s_i)$ `mod BUCKET_COUNT`
- Best if `a` and `BUCKET_COUNT` are relatively prime
  - E.g., `a` = 65599, `BUCKET_COUNT` = 1024

28

How to Hash Strings?

Potentially expensive to compute $\Sigma\ a^i s_i$

So let's do some algebra
- (by example, for string `s` of length 5, `a`=65599):

$h = \Sigma 65599^i * s_i$

$h = 65599^0 * s_0 + 65599^1 * s_1 + 65599^2 * s_2 + 65599^3 * s_3 + 65599^4 * s_4$

Direction of traversal of s doesn't matter, so…

$h = 65599^0 * s_4 + 65599^1 * s_3 + 65599^2 * s_2 + 65599^3 * s_1 + 65599^4 * s_0$

$h = 65599^4 * s_0 + 65599^3 * s_1 + 65599^2 * s_2 + 65599^1 * s_3 + 65599^0 * s_4$

$h = (((((s_0) * 65599 + s_1) * 65599 + s_2) * 65599 + s_3) * 65599) + s_4$

29

How to Hash Strings?

Yielding this function

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

30

How to Protect Keys?

Suppose `Table_add()` function contains this code:

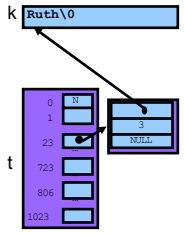
```
void Table_add(struct Table *t, const char *key, int value)
{  …
   struct Binding *p =
      (struct Binding*)malloc(sizeof(struct Binding));
   p->key = key;
   …
}
```

31

How to Protect Keys?

Problem: Consider this calling code:

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

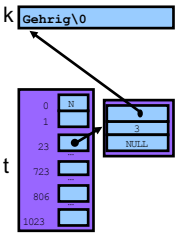


32

How to Protect Keys?

Problem: Consider this calling code:

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



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

33

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);
   …
}
```

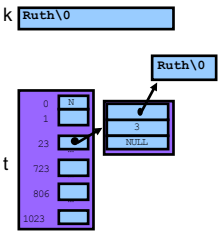
Why add 1?

34

How to Protect Keys?

Now consider same calling code:

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



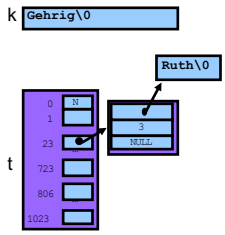
35

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



36

Who Owns the Keys?

Then the hash table **owns** its keys

- That is, the hash table owns the memory in which its keys reside
- `Hash_free()` function must free the memory in which the key resides

37

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

- Hashing algorithms
- Defensive copies
- Key ownership

38