Data Structures

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)
- (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;
};
```

Really this is the address at which "Ruth" resides
**Linked List Data Structure**

- struct List
- struct Node

**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

**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

**Hash Table Data Structure**

- Array of linked lists

```c
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**

- Bucket
- Binding

Hash function maps given key to an integer

Mod integer by $BUCKET\_COUNT$ to determine proper bucket
Hash Table Example (cont.)

First key: "the"
- hash("the") = 96/5167877 % 7 = 1
- Search buckets [1] for binding with key "the": not found

Hash Table Example (cont.)

Second key: "cat"
- hash("cat") = 389564756 % 7 = 2
- Search buckets [2] for binding with key "cat": not found

Hash Table Example (cont.)

Third key: "in"
- hash("in") = 6880205 % 7 = 5
- Search buckets [5] for binding with key "in": not found

Example: BUCKET_COUNT = 7
- Add binding with key "the" and its value to buckets [1]
- the, in, in, the

Hash Table Example (cont.)

Add binding with key "the" and its value to buckets [1]
Hash Table Example (cont.)

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

Fourth word: “the”
- hash("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”
- hash("hat") = 865559739; 865559739 % 7 = 2
Search buckets [2] for binding with key “hat”; not found

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

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

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

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

Is the add performance always fast?
Agenda

Linked lists
Hash tables
Hash table issues

How Many Buckets?

Many!
- Too few ⇒ large buckets ⇒ slow add, slow search
But not too many!
- Too many ⇒ memory is wasted
This is OK:

![Diagram showing bucket distribution]

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:

![Diagram showing hash function]

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

Footnote: I originally designed this homework so that BUCKET\_COUNT is a prime number. In 2016 I wondered, “wouldn’t it work just as well if $a$ and BUCKET\_COUNT are just relatively prime? Measurements show no using a prime number of buckets leads to more even distribution of bucket contents.”

How to Hash Strings?

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

So let’s do some algebra (“Horner’s rule”)
- (by example, for string $s$ of length 5, $a=65599$):

```
\[
\begin{align*}
  h &= \sum a^i s_i \\
  h &= 65599^5 a_5 + 65599^4 a_4 + 65599^3 a_3 + 65599^2 a_2 + 65599 a_1 + a_0 \\
  Direction of traversal of $s$ doesn't matter, so... \\
  h &= 65599^4 a_4 + 65599^3 a_3 + 65599^2 a_2 + 65599 a_1 + a_0 \\
  h &= 65599^3 a_3 + 65599^2 a_2 + 65599 a_1 + a_0 \\
  h &= 65599^2 a_2 + 65599 a_1 + a_0 \\
  h &= 65599 a_1 + a_0 \\
  h &= a_0 \\
\end{align*}
\]
```

Yielding this function

```c
size_t hash(const char *s, size_t bucketCount)
{
    size_t i;
    size_t h = 0;
    for (i=0; s[i]!=\'\0\'; i++)
        h = h * 65599 + (size_t)s[i];
    return h % bucketCount;
}
```
How to Protect Keys?

Suppose `Table_add()` function contains this code:

```c
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:

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

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

```c
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?

How to Protect Keys?

Now consider same calling code:

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

Hash table is not corrupted
**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

**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

---

**Princeton University**

**Computer Science 217: Introduction to Programming Systems**

**Debugging (Part 2)**

---

**"Programming in the Large" Steps**

**Design & Implement**
- Program & programming style (done)
- Common data structures and algorithms
- Modularity
- Building techniques & tools (done)

**Test**
- Testing techniques (done)

**Debug**
- Debugging techniques & tools <-- we are still here

**Maintain**
- Performance improvement techniques & tools

---

**Goals of this Lecture**

Help you learn about:
- Debugging strategies & tools related to dynamic memory management (DMM)

Why?
- Many bugs occur in code that does DMM
- DMM errors can be difficult to find
  - DMM error in one area can manifest itself in a distant area
  - A power programmer knows a wide variety of DMM debugging strategies
  - A power programmer knows about tools that facilitate DMM debugging

* Management of heap memory via `malloc()`, `calloc()`, `realloc()`, and `free()`

---

**Agenda**

(9) Look for common DMM bugs
(10) Diagnose seg faults using gdb
(11) Manually inspect `malloc` calls
(12) Hard-code `malloc` calls
(13) Comment-out free calls
(14) Use Meminfo
(15) Use Valgrind
Look for Common DMM Bugs

Some of our favorites:

```c
int *p; /* value of p undefined */
*p = somevalue;

char *p; /* value of p undefined */
fgets(p, 1024, stdin);
```

```c
int *p;
p = (int*)malloc(sizeof(int));
*p = 5;
free(p);
*p = 6;
```

What are the errors?

```c
int *p;
p = (int*)malloc(sizeof(int));
*p = 5;
p = (int*)malloc(sizeof(int));
*p = 5;
free(p);
free(p);
```

Look for Common DMM Bugs

Some of our favorites:

Agenda

(9) Look for common DMM bugs
(10) Diagnose seg faults using gdb
(11) Manually inspect malloc calls
(12) Hard-code malloc calls
(13) Comment-out free calls
(14) Use Meminfo
(15) Use Valgrind

Diagnose Seg Faults Using GDB

Segmentation fault => make it happen in gdb
- Then issue the gdb where command
- Output will lead you to the line that caused the fault
  - But that line may not be where the error resides!

Manually Inspect Malloc Calls

Manually inspect each call of malloc()
- Make sure it allocates enough memory

Do the same for calloc() and realloc()
Manually Inspect Malloc Calls

Some of our favorites:

```c
char *s1 = "hello, world";
char *s2;
s2 = (char*)malloc(strlen(s1));
strcpy(s2, s1);
```

```c
char *s1 = "Hello";
char *s2;
s2 = (char*)malloc(sizeof(s1));
strcpy(s2, s1);
```

```c
long double *p;
p = (long double*)malloc(sizeof(long double));
```

```c
long double *p;
p = (long double*)malloc(sizeof(p));
```

Agenda

(9) Look for common DMM bugs
(10) Diagnose seg faults using gdb
(11) Manually inspect malloc calls
(12) Hard-code malloc calls
(13) Comment-out free calls
(14) Use Meminfo
(15) Use Valgrind

Hard-Code Malloc Calls

Temporarily change each call of malloc() to request a large number of bytes
- Say, 10000 bytes
- If the error disappears, then at least one of your calls is requesting too few bytes

Then incrementally restore each call of malloc() to its previous form
- When the error reappears, you might have found the culprit

Do the same for calloc() and realloc()

Agenda

(9) Look for common DMM bugs
(10) Diagnose seg faults using gdb
(11) Manually inspect malloc calls
(12) Hard-code malloc calls
(13) Comment-out free calls
(14) Use Meminfo
(15) Use Valgrind

Comment-Out Free Calls

Temporarily comment-out every call of free()
- If the error disappears, then program is
  - Freeing memory too soon, or
  - Freeing memory that already has been freed, or
  - Freeing memory that should not be freed,
  - Etc.

Then incrementally "comment-in" each call of free()
- When the error reappears, you might have found the culprit

Agenda

(9) Look for common DMM bugs
(10) Diagnose seg faults using gdb
(11) Manually inspect malloc calls
(12) Hard-code malloc calls
(13) Comment-out free calls
(14) Use Meminfo
(15) Use Valgrind
Use Meminfo

Use the **Meminfo** tool
- Simple tool
- Initial version written by Dondero
- Current version written by COS 217 alumnus RJ Liljestrom
- Reports errors after program execution
  - Memory leaks
  - Some memory corruption
  - User-friendly output

Appendix 1 provides example buggy programs
Appendix 2 provides Meminfo analyses

Use Valgrind

Use the **Valgrind** tool
- Complex tool
- Written by multiple developers, worldwide
  - See www.valgrind.org
- Reports errors during program execution
  - Memory leaks
  - Multiple frees
  - Dereferences of dangling pointers
  - Memory corruption
- Comprehensive output
  - But not always user-friendly

Appendix 1 provides example buggy programs
Appendix 3 provides Valgrind analyses

Summary

Strategies and tools for debugging the DMM aspects of your code:
- Look for common DMM bugs
- Diagnose seg faults using gdb
- Manually inspect malloc calls
- Hard-code malloc calls
- Comment-out free calls
- Use Meminfo
- Use Valgrind

Appendix 1: Buggy Programs

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. int main(void)
4. { int *pi;
5.     pi = (int*)malloc(sizeof(int));
6.     *pi = 5;
7.     printf("%d\n", *pi);
8.     pi = (int*)malloc(sizeof(int));
9.     *pi = 6;
10.    printf("%d\n", *pi);
11.    free(pi);
12.    return 0;
13. }
```

Memory leak:
Memory allocated at line 5 is leaked
Appendix 1: Buggy Programs

doublefree.c

1. #include <stdio.h>
2. #include <stdlib.h>
3. int main(void)
4. {
5.   int *pi;
6.   pi = (int*)malloc(sizeof(int));
7.   *pi = 5;
8.   printf("%d\n", *pi);
9.   free(pi);
10.  return 0;
11. }

Multiple free:
Memory allocated at line 5 is freed twice

Appendix 1: Buggy Programs
danglingptr.c

1. #include <stdio.h>
2. #include <stdlib.h>
3. int main(void)
4. {
5.   int *pi;
6.   pi = (int*)malloc(sizeof(int));
7.   *pi = 5;
8.   printf("%d\n", *pi);
9.   free(pi);
10.  printf("%d\n", *pi);
11. }

Dereference of dangling pointer:
Memory accessed at line 9 already was freed

Appendix 1: Buggy Programs
toomall.c

1. #include <stdio.h>
2. #include <stdlib.h>
3. int main(void)
4. {
5.   int *pi;
6.   pi = (int*)malloc(1);
7.   *pi = 5;
8.   printf("%d\n", *pi);
9.   free(pi);
10.  return 0;
11. }

Memory corruption:
Too little memory is allocated at line 5
Line 6 corrupts memory

Appendix 2: Meminfo

Meminfo can detect memory leaks:

```bash
$ gcc217m leak.c -o leak
$ leak
1
$ ls
  leak.c  leak  meminfo30462.out
  meminfo30462.out
Errors:
** Unfused bytes (1 block) allocated at leak.c:5
Summary Statistics:
  Maximum bytes allocated at once: 1
  Total number of allocated bytes: 1

Statistics by Line:
  Bytes  Location
  -1  leak.c:11
  4  leak.c:5
  4  leak.c:8
  4  TOTAL

Statistics by Compilation Unit:
  4  leak.c
  4  TOTAL
```

Meminfo caveat:

- Don’t mix .o files built with gcc217 and gcc217m
- meminfo*.out files can be large
- Should delete frequently
- Programs built with gcc217m run slower than those built with gcc217
- Don’t build with gcc217m when doing timing tests
Appendix 3: Valgrind

Valgrind can detect memory leaks:

```c
$ grep leak core -o leak
3 leak
3 leak
```

- `leak`: Memory error detector
- ` Copyright (C) 2002-2012, and GNU GPL/3, by Julian Seward et al.
- `Using Valgrind-3.10.1 and LibVEX; return with -h for copyright info.
- `Command: leak
- `5
- `6
- `leak
- `LEAK SUMMARY:
- `definitely lost: 4 bytes in 1 blocks
- `indirectly lost: 0 bytes in 0 blocks
- `possibly lost: 0 bytes in 0 blocks
- `still reachable: 0 bytes in 0 blocks
- `suppressed: 0 bytes in 0 blocks
- `Return with `--leak-suppress-full` for details of leaked memory
- `For counts of detected and suppressed errors, return with: --v
- `ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 1 from 2)
```

Valgrind can detect multiple frees:

```c
$ grep doublefree core -o doublefree
3 doublefree
3 doublefree
```

- `doublefree`: Memory error detector
- ` Copyright (C) 2002-2012, and GNU GPL/3, by Julian Seward et al.
- `Using Valgrind-3.10.1 and LibVEX; return with -h for copyright info.
- `Command: doublefree
- `5
- `LEAK SUMMARY:
- `invalid free() / delete() / delete[] / realloc()
- `at 0x4e0200f: free (vg_replace_malloc.c:464)
- `by 0xe00000: main (doublefree.c:9)
- `Address 0x2a0540 is 0 bytes inside a block of size 4 freed d
- `at 0x4e0200f: free (vg_replace_malloc.c:464)
- `by 0xe00000: main (doublefree.c:8)
- `5
- `HEAP SUMMARY:
- `in use at exit: 0 bytes in 0 blocks
- `total heap usage: 1 allocs, 2 frees, 0 bytes allocated
- `All heap blocks were freed -- no leaks are possible
- `For counts of detected and suppressed errors, return with: --v
- `ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 1)
```

Valgrind can detect dereferences of dangling pointers:

```c
$ grep danglingptr core -o danglingptr
3 danglingptr
3 danglingptr
```

- `danglingptr`: Memory error detector
- ` Copyright (C) 2002-2012, and GNU GPL/3, by Julian Seward et al.
- `Using Valgrind-3.10.1 and LibVEX; return with -h for copyright info.
- `Command: danglingptr
- `5
- `HEAP SUMMARY:
- `invalid read of size 4
- `in use at exit: 0 bytes in 0 blocks
- `total heap usage: 1 allocs, 1 frees, 4 bytes allocated
- `All heap blocks were freed -- no leaks are possible
- `For counts of detected and suppressed errors, return with: --v
- `ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 1)
```

Valgrind can detect memory corruption:

```c
$ grep toosmall core -o toosmall
3 toosmall
3 toosmall
```

- `toosmall`: Memory error detector
- ` Copyright (C) 2002-2012, and GNU GPL/3, by Julian Seward et al.
- `Using Valgrind-3.10.1 and LibVEX; return with -h for copyright info.
- `Command: toosmall
- `5
- `invalid write of size 4
- `at address 0x2ca78f: 0 bytes inside a block of size 1 allocated
- `at leak.cpp:370: leak (vg_replace_malloc.c:370)
- `by 0xe00000: main (toosmall.c:5)
- `5
- `invalid read of size 4
- `at leak.cpp:370: leak (vg_replace_malloc.c:370)
- `by 0xe00000: main (toosmall.c:7)
- `5
- `Address 0x2ca78f is 0 bytes inside a block of size 1 allocated
- `at leak.cpp:370: leak (vg_replace_malloc.c:370)
- `by 0xe00000: main (toosmall.c:5)
```

Appendix 3: Valgrind

Valgrind can detect memory corruption (cont.):

```c
$ grep toosmall core -o toosmall
3 toosmall
3 toosmall
```

- `toosmall`: Memory error detector
- ` Copyright (C) 2002-2012, and GNU GPL/3, by Julian Seward et al.
- `Using Valgrind-3.10.1 and LibVEX; return with -h for copyright info.
- `Command: toosmall
- `5
- `invalid write of size 4
- `at address 0x2ca78f: 0 bytes inside a block of size 1 allocated
- `at leak.cpp:370: leak (vg_replace_malloc.c:370)
- `by 0xe00000: main (toosmall.c:5)
- `5
- `invalid read of size 4
- `at leak.cpp:370: leak (vg_replace_malloc.c:370)
- `by 0xe00000: main (toosmall.c:7)
- `5
- `Address 0x2ca78f is 0 bytes inside a block of size 1 allocated
- `at leak.cpp:370: leak (vg_replace_malloc.c:370)
- `by 0xe00000: main (toosmall.c:5)
```

Appendix 3: Valgrind

Valgrind can detect memory leaks:
Valgrind caveats:

- Not intended for programmers who are new to C
  - Messages may be cryptic

- Suggestion:
  - Observe line numbers referenced by messages
  - Study code at those lines
  - Infer meanings of messages