**Learn C the hard way**

Part 2

**Linked List**

A Linked list works by nodes having pointers to their next or previous element. A doubly linked list contains pointers to both while a singly linked list only points at the next element.

Anything that involves inserting or deleting an element will be very fast. The main disadvantage is that traversing a linked list involves processing every single pointer along the way, in which case sorting and searching will be slow.

Example – linked list

Typedef struct Node {

Int data;

Struct Node \*next;

} Node;

Int main() {

Node \*head = NULL;

Node \*second = NULL;

Node \*third = NULL;

Head = malloc(sizeof(Node));

Second = malloc(sizeof(Node));

Third = malloc(sizeof(Node));

Head->data = 1;

Head->next = second;

Second->Data = 2;

Second->next = third;

Fprintf(stdout, “%d\n”, head->next->data);

Return 0;

}

**Hash map**

Hash maps (aka hashmaps, hashes, dictionaries) are used for storing key/value data. A hash map works by performing a hashing calculation on the keys to produce an integer, then use that integer to find a *bucket* to get or set the value. A JavaScript object is an example of a hash map.

Example

#define TABLE\_SIZE 20000

typedef struct entry\_t {

char \*key;

char \*value;

struct entry\_t \*next;

} entry\_t;

typedef struct {

entry\_t \*\*entries;

} ht\_t;

unsigned int hash(const char \*key) {

unsigned long int value = 0;

unsigned int i = 0;

unsigned int key\_len = strlen(key);

// do several rounds of multiplication

for (; i < key\_len; ++i) {

value = value \* 37 + key[i];

}

// make sure value is 0 <= value < TABLE\_SIZE

value = value % TABLE\_SIZE;

return value;

}

entry\_t \*ht\_pair(const char \*key, const char \*value) {

// allocate the entry

entry\_t \*entry = malloc(sizeof(entry\_t) \* 1);

entry->key = malloc(strlen(key) + 1);

entry->value = malloc(strlen(value) + 1);

// copy the key and value in place

strcpy(entry->key, key);

strcpy(entry->value, value);

// next starts out null but may be set later on

entry->next = NULL;

return entry;

}

ht\_t \*ht\_create(void) {

// allocate table

ht\_t \*hashtable = malloc(sizeof(ht\_t) \* 1);

// allocate table entries

hashtable->entries = malloc(sizeof(entry\_t\*) \* TABLE\_SIZE);

// set each to null (needed for proper operation)

int i = 0;

for (; i < TABLE\_SIZE; ++i) {

hashtable->entries[i] = NULL;

}

return hashtable;

}

void ht\_set(ht\_t \*hashtable, const char \*key, const char \*value) {

unsigned int slot = hash(key);

// try to look up an entry set

entry\_t \*entry = hashtable->entries[slot];

// no entry means slot empty, insert immediately

if (entry == NULL) {

hashtable->entries[slot] = ht\_pair(key, value);

return;

}

entry\_t \*prev;

// walk through each entry until either the end is

// reached or a matching key is found

while (entry != NULL) {

// check key

if (strcmp(entry->key, key) == 0) {

// match found, replace value

free(entry->value);

entry->value = malloc(strlen(value) + 1);

strcpy(entry->value, value);

return;

}

// walk to next

prev = entry;

entry = prev->next;

}

// end of chain reached without a match, add new

prev->next = ht\_pair(key, value);

}

char \*ht\_get(ht\_t \*hashtable, const char \*key) {

unsigned int slot = hash(key);

// try to find a valid slot

entry\_t \*entry = hashtable->entries[slot];

// no slot means no entry

if (entry == NULL) {

return NULL;

}

// walk through each entry in the slot, which could just be a single thing

while (entry != NULL) {

// return value if found

if (strcmp(entry->key, key) == 0) {

return entry->value;

}

// proceed to next key if available

entry = entry->next;

}

// reaching here means there were >= 1 entries but no key match

return NULL;

}

void ht\_del(ht\_t \*hashtable, const char \*key) {

unsigned int bucket = hash(key);

// try to find a valid bucket

entry\_t \*entry = hashtable->entries[bucket];

// no bucket means no entry

if (entry == NULL) {

return;

}

entry\_t \*prev;

int idx = 0;

// walk through each entry until either the end is reached or a matching key is found

while (entry != NULL) {

// check key

if (strcmp(entry->key, key) == 0) {

// first item and no next entry

if (entry->next == NULL && idx == 0) {

hashtable->entries[bucket] = NULL;

}

// first item with a next entry

if (entry->next != NULL && idx == 0) {

hashtable->entries[bucket] = entry->next;

}

// last item

if (entry->next == NULL && idx != 0) {

prev->next = NULL;

}

// middle item

if (entry->next != NULL && idx != 0) {

prev->next = entry->next;

}

// free the deleted entry

free(entry->key);

free(entry->value);

free(entry);

return;

}

// walk to next

prev = entry;

entry = prev->next;

++idx;

}

}

void ht\_dump(ht\_t \*hashtable) {

for (int i = 0; i < TABLE\_SIZE; ++i) {

entry\_t \*entry = hashtable->entries[i];

if (entry == NULL) {

continue;

}

printf("slot[%4d]: ", i);

for(;;) {

printf("%s=%s ", entry->key, entry->value);

if (entry->next == NULL) {

break;

}

entry = entry->next;

}

printf("\n");

}

}

int main(int argc, char \*\*argv) {

ht\_t \*ht = ht\_create();

ht\_set(ht, "name1", "em");

ht\_set(ht, "name2", "russian");

ht\_set(ht, "name3", "pizza");

ht\_set(ht, "name4", "doge");

ht\_set(ht, "name5", "pyro");

ht\_set(ht, "name6", "joost");

ht\_set(ht, "name7", "kalix");

ht\_dump(ht);

return 0;

}

**Binary tree**

unlike arrays, linked lists, stacks & queues which are linear data structures, trees are hierarchical data structures. The top most node is called root of the tree. The elements that are directly under an element are called its children. The element above something is called its parent. Similar to a filesystem.

The advantages of using a tree structure are; they reflect structural relationships in data, they are used to represent hierarchies, they provide efficient insertion and searching, they are flexible allowing to move sub-trees around with minimal effort.

----

j <-- root

/ \

f k

/ \ \

a h z <-- leaves

Example

typedef struct node {

int data;

struct node \*left;

struct node \*right;

} node;

struct node \*newNode(int data) {

struct node \*node = malloc(sizeof(node));

node->data = data;

node->left = NULL;

node->right = NULL;

return node;

}

int main(int argc, char \*argv[]) {

struct node \*root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

//etc

return 0;

}

**Stacks & Queues**

Stacks & queues are simple data structures that are really variants of the *List* data structure. All they do is use a list with a discipline that says *you always place elements on one end of the list*.

A stack you always push and pop – LIFO

A queue you always shift to the front but pop from the end (FIFO)

Example – stack

Struct Stack {

Int top;

Unsigned capacity;

Int \*array;

};

Struct Stack \*createStack(unsigned capacity) {

Struct Stack \*stack = malloc(sizeof(struct Stack));

Stack->capacity = capacity;

Stack->top = -1;

Stack->array = malloc(stack->capacity \* sizeof(int));

Return stack;

}

Int isFull(struct Stack \*stack) {

Return stack->top == stack->capacity – 1;

}

Int isEmpty(struct Stack \*stack) {

Return stack->top == -1;

}

Void push(struct Stack \*stack, int item) {

If (isFull(stack)) {

Return;

}

Stack->array[++stack->top] = item;

Fprintf(stdout, “%d pushed to stack\n”, item);

}

Int pop(struct Stack \*stack) {

If (isEmpty(stack)) {

Return INT\_MIN; // from <limits.h>

}

Return stack->array[stack->top--];

}

Int peek(struct Stack \*stack) {

If (isEmpty(stack)) {

Return INT\_MIN;

}

Return stack->array[stack->top];

}

Int main() {

Struct Stack \*stack = createStack(100);

Push(stack, 100);

Push(stack, 200);

Push(stack, 500);

Fprintf(stdout, “%d popped from stack\n”, pop(stack));

Return 0;

}

**QUIZ**

1. Add push, pop and unshift functions to linked list
2. Create a doubly linked list
3. Create a queue
4. Add function to stack which will print out the full list
5. Create a hash table
6. Create a binary tree