# 15-122: Principles of Imperative Computation

## Contents

# Things to look out for

- runtime errors
  - div/mod by 0
  - array index out of bounds/not allocated
- undefined behavior
  - dereferencing `NULL` pointer (error in C0)
- common logic errors
  - is your variable for an index or a value at an index?

# Syntax

## Ints

- signed modular arithmetica and fixed-len bit vectors
- C0: 32 bit/8 hex digits/4 bytes
- check if your language truncates int div toward 0. probably.
- bitwise operations
  - (&) and
  - (^) exclusive or
  - (|) or
  - (~) negation
  - (<<) left shift; fills with 0s
  - (>>) right shift; copies highest bit to fill. like division by 2 except truncation toward $-\infty$

## Arrays

- note that the following are equivalent

```
for (int i = 0; i < 100; i++) {
   // ...
}
```

```
int i = 0;
while (i < 100) {
  // ...
  i++;
}
```

- C0 arrays
  - `int[] A = alloc_array(int, 10);`
  - allocates default type (i.e. 0 for `int` arrays)
  - be wary of aliasing

## `typedef`

- used in structs often TODO

- the type of a function describes the number, position, and type of input params and its output type

```
typedef int hash_string_fn(string s);
hash_string_fn* F = &hash_string;
```

## Structs

- aggregates differently-typed data
- struct declarations are thus:

```c
struct img_header {
  pixel_t[] data;
  int width;
  int height;
}
```

  - `data`, `width`, and `height` are called fields
- struct allocations return the allocated pointer

```c
struct img_header* IMG = alloc(struct img_header);
```

- write to fields using arrow notation, a syntactic sugar

```c
IMG->width = 1; /* or */ (*IMG).width = 1;
```

  - `*IMG` is called dereferencing

## Pointers/Memory Allocation

- not just structs: can allocate memory to arbitrary types
- `void *malloc(size_t size)`: reserve `size` bytes of memory, return this pointer

```c
int* int_ptr = malloc(sizeof(int));
```

- `void free(void *p)` your pointers
- `NULL` pointer (`0x00000000`)
  - an "invalid address"
  - cannot be dereferenced

### Generic Pointers

- the void pointer `void*` can point to any address and can be casted to from any address

```c
int* ip = alloc(int);
void* p1 = (void*)ip;
void* p2 = (void*)alloc(string);
void* p3 = (void*)alloc(struct produce);
void* p4 = (void*)alloc(int**);
```

- we may convert them back as well

```c
int* x = (int*)p1;
string x = *(string*)p2;
```

- generic functions may also exist (see `typedef` section)

```c
(*F)("hello")
```

### Address-of

- `&foo` grabs the address of `foo`

## C Preprocessor Language

- common built-in libraries:

```c
#include <stdlib.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
```

- custom libraries use quotation marks instead of angle brackets
- constants are simply substituted; they do not follow order of operations

```
#ifdef DEBUG
printf("debugging time\n");
#else
#define INT_MIN (-1 ^ 0x7FFFFFFF)
#endif
```

- macro functions also exist (out of scope of notes)

# Conceptual Topics

## Big O/Sorting/Searching

- asymptotic complexity measures for algorithms
- $f \in O(g)$ if $\exists c \in \mathbb{R}, n_0 \in \mathbb{N}$ s.t. $\forall n \geq n_0, f(n) \leq c \cdot g(n)$
- merge/quick sort and binsearch are all "divide and conquer" methods
- selection sort

  ```
  /** sorts A in range [lo, hi) */
  void selection_sort(int[] A, int lo, int hi) {
    for (int i = lo; i < hi; i++) {
      int min = find_min_idx(A, i, hi); // search linearly
      swap(A, i, min);
    }
  }
  ```

  - $O(n^2)$
  - in-place
  - not stable
- quicksort

  ```
  int partition(int[] A, int lo, int piv, int hi)
  //@requires 0 <= lo <= pi < hi <= \length(A);
  //@ensures lo <= \result < hi
  //@ensures A[\result] >= all elements from A[lo, \result)
  //@ensures A[\result] <= all elements from A[\result+1, hi)
  ;

  void quicksort(int[] A, int lo, int hi) {
    if (hi - lo <= 1) { return; }

    int piv = ...; // random index
    int mid = partition(A, lo, piv, hi);

    quicksort(A, lo, mid);
    quicksort(A, mid + 1, hi);

    return;
  }
  ```

  - method:
    - pick arbitrary "pivot" element
    - partition array into smaller than pivot elements/larger than pivot elements
    - recursively sort left and right of pivot until partitions are 1 large
  - $O(n \log n)$ because average $\log n$ times partitioned, $n$ copies to do each partition.

    - in-place (needs a clever partition algo to still be efficient)
    - not stable
- mergesort

```
void mergesort(int[] A, int lo, int hi) {
  if (hi - lo <= 1) { return; }
  int mid = lo + (hi - lo) / 2;

  mergesort(A, lo, mid);
  mergesort(A, mid, hi);
  merge(A, lo, mid, hi);

  return;
}
```

    - method:
        - divide array in half
        - sort halves recursively via merging the divided parts
    - $O(n \log n)$ because $\log n$ divisions, merging requires $n$ operations per level
    - not in-place
- binary search

```
int binsearch(int x, int[] A, int n) {
  int lo = 0;
  int hi = n;
  while (lo < hi) {
    int mid = lo + (hi-lo) / 2;
    if (A[mid] == x) { return mid; }
    else if (A[mid] < x) { lo = mid + 1; }
    else { hi = mid; }
  }
  return -1;
}
```

    - $O(\log n)$

### Amortization

- used in unbounded arrays (see the dedicated section)

### C's Memory Model

- TODO see `14-generic.pdf`

## Data Structures

### Linked Lists

- insertion/deletion in $O(1)$
- access in $O(n)$
- basic C0 implementation via a recursive type:

```
typedef struct list_node list;
struct list_node {
  elem data; // arbitrary (pointer to) element type
```

```
  list* next;
};
```

- For the implementation of stacks, we can reuse linked linked lists are often `NULL`-terminated. i.e., we point to the beginning `list_node` and know we have reached the end when we reach `NULL`. so for such a `list* example = ...`, we have:

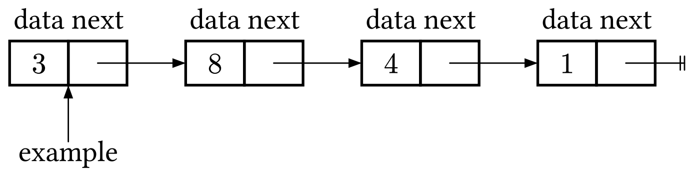


Figure 1: `NULL`-terminated linked list

- this is also called a “concrete type” (as apposed to abstract) because instead of having an implementation/interface divide, there is no abstraction. e.g. we directly use `data` and `next`
- linked lists may also use “dummy nodes” that we keep track of to build different data structures
- note that recursive types must be under pointers (or arrays). if we had `int data`, we have potentially infinite memory to allocate.

- circularity is bad
  - tortoise and hare algorithm: start tortoise (travels one node at a time) at `list[0]`, hare (travels two nodes at a time) at `list[1]`. if hare catches tortoise before it reaches the end, we have a circle.

## Stacks & Queues

- stacks (FIFO) can simply implemented with a `NULL`-terminated linked list
- queues (LIFO) will keep track of the back as well

## Unbounded Arrays

- access in $O(1)$
- insertion/deletion also in $O(1)$, amortized
- implemented via an array that resizes (e.g. doubles when three-fourths full and halves when a fourth full or so)
- see the section on amortization

## Hash Tables

- used to implement dictionaries, aka associative arrays or maps. hash tables will result in
  - $O(1)$ insertion/deletion average (hashing), amortized (bucket resizing)
  - $O(1)$ lookup average (no resizing needed)
- a dictionary will map all its keys to entries or all its keys to values
  - entires include keys, values don’t
- implementation

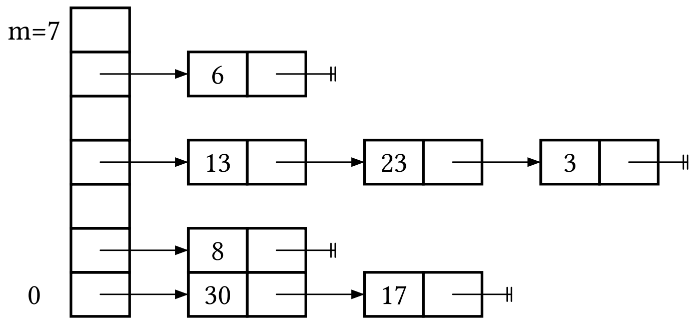


Figure 2: hash table with separate chaining, `hash(i) = i % 10`

- use an array of $m$-many “buckets”
- a hash function converts your key into a int
- store entries at `array[hash(key) % m]`
- if hash collisions occur
  - separate chaining: each array element is either `NULL` or a chain of entries (with the same hash)
  - linear probing: search linearly for unused space in array
  - quadratic probing: search quadratically for unused space in array
- chains will (on average) be $n/m$ long with $n$ entries and $m$ buckets ($n/m$ called the load factor)
- we want a good hash function, possibly tailored specifically to our input data, to preserve even distribution of which keys are placed in bucket
  - linear congruential generator: $((a \cdot x) + c) \mod d$. choose $d = 2^{32}$. choose $c, d$ coprime. eg:

    ```
    /** from c0 random library */
    int rand (rand_t gen) {
      gen->seed = gen->seed * 1664525 + 1013904223;
      return gen->seed;
    }
    ```

    - fine for hashing ints for a hashmap, but bad for cryptography
    - pseudo-random is good for debugging—reproducable hashes if given same seed
- searching through too-long chains will cause time complexity to not be linear again. so when load factor is too large, increase (e.g. double) $m$: preserves constant complexity via amortization

## Trees

## Binary Search Trees

- like hash maps, also used to implement dictionaries
  - $O(h)$ insertion/deletion (not covered)/lookup worst case

    needs to be balanced to be usable
- invariants
  - ordering invariant

## AVL Trees

- $O(\log n)$ insertion/deletion/lookup
- invariants
  - ordering invariant
  - balance aka height invariant: at any node in the tree, the heights of the left/right subtrees differ by at most 1.
    - restore via rotations

## Priority Queues via Heaps

- $O(\log n)$ push/pop, $O(1)$ peek
- built on binary tree. highest priority is at root node
- invariants
  - Min-heap ordering invariant (alternative 1): The key of each node in the tree is less than or equal to all of its children's keys.
  - Min-heap ordering invariant (alternative 2): The key of each node in the tree except for the root is greater than or equal to its parent's key
  - shape invariant: fill tree by level, left to right
- push: place new node via shape invariant. then sift it up to fix ordering invariant
- pop: swap root with last in shape invariant and delete it. then sift new root node down

# Appendix

## Library Examples

### Self-Sorting Array

```
/*
 * An interface to self-sorting arrays
 *
 * 15-122 Principles of Imperative Computation
 */

#use <util>
#use <string>
#use <conio>

/************************************************************************/
/************************* BEGIN IMPLEMENTATION *************************/

// Implementation-side type
struct ssa_header {
  string[] data;  // sorted
  int length;     // = \length(data)
};
typedef struct ssa_header ssa;

// Internal debugging function that prints an SSA without checking contracts
// (useful to debug representation invariant issues)
void ssa_print_unsafe(ssa* A)
{
  int len = A->length;
  printf("SSA length=%d; data=[", len);
  for (int i = 0; i < len; i++)
  //@loop_invariant 0 <= i && i <= len;
  {
    printf("%s", A->data[i]);
    if (i < len-1) printf(", ");
  }
  printf("]");
}


// Auxiliary functions
void swap(string[] D, int i, int j)
//@requires 0 <= i && i < \length(D);
//@requires 0 <= j && j < \length(D);
{
  string tmp = D[i];
  D[i] = D[j];
  D[j] = tmp;
}
```

```
bool is_array_expected_length(string[] A, int length) {
  //@assert \length(A) == length;
  return true;
}

bool ssa_sorted(ssa* A)
//@requires A != NULL && is_array_expected_length(A->data, A->length);
{
  for (int i=0; i < A->length - 1; i++)
    //@loop_invariant 0 <= i;
    if (string_compare(A->data[i], A->data[i+1]) > 0) return false;
  return true;
}

// Representation invariant
bool is_ssa(ssa* A) {
  return A != NULL
      && is_array_expected_length(A->data, A->length)
      && ssa_sorted(A);
}

// Internal debugging function that prints an SSA
// (useful to spot bugs that do not invalidate the representation invariant)
void ssa_print_internal(ssa* A)
//@requires is_ssa(A);
{
  ssa_print_unsafe(A);
}

// Implementation of exported functions
int ssa_len(ssa* A)
//@requires is_ssa(A);
//@ensures \result >= 0;
//@ensures \result == \length(A->data);
{
  return A->length;
}

string ssa_get(ssa* A, int i)
//@requires is_ssa(A);
//@requires 0 <= i && i < ssa_len(A);
{
  return A->data[i];
}

void ssa_set(ssa* A, int i, string x)
//@requires is_ssa(A);
//@requires 0 <= i && i < ssa_len(A);
//@ensures is_ssa(A);
{
  A->data[i] = x;
```

```
  // Move x up the array if needed
  for (int j=i; j < A->length-1 && string_compare(A->data[j],A->data[j+1]) > 0; j++)
    //@loop_invariant i <= j;
    swap(A->data, j, j+1);

  // Move x down the array if needed
  for (int j=i; j > 0 && string_compare(A->data[j],A->data[j-1]) < 0; j--)
    //@loop_invariant 0 <= j && j <= i;
    swap(A->data, j, j-1);
}

ssa* ssa_new(int size)
//@requires 0 <= size;
//@ensures is_ssa(\result);
//@ensures ssa_len(\result) == size;
{
  ssa* A = alloc(ssa);
  A->data = alloc_array(string, size);
  A->length = size;
  return A;
}

// Client-facing print function (does not reveal implementation details)
void ssa_print(ssa* A)
//@requires is_ssa(A);
{
  int len = A->length;
  printf("SSA: [");
  for (int i = 0; i < len; i++)
  //@loop_invariant 0 <= i && i <= len;
  {
    printf("%s", A->data[i]);
    if (i < len-1) printf(", ");
  }
  printf("]");
}

// Client type
typedef ssa* ssa_t;

/************************** END IMPLEMENTATION *************************/
/**********************************************************************/

/**********************************************************************/
/******************************* Interface ******************************/

// typedef ______* ssa_t;

int ssa_len(ssa_t A)
/*@requires A != NULL;   @*/
/*@ensures \result >= 0; @*/ ;
```

```
ssa_t ssa_new(int size)
/*@requires 0 <= size;              @*/
/*@ensures \result != NULL;         @*/
/*@ensures ssa_len(\result) == size; @*/ ;

string ssa_get(ssa_t A, int i)
/*@requires A != NULL;               @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;

void ssa_set(ssa_t A, int i, string x)
/*@requires A != NULL;               @*/
/*@requires 0 <= i && i < ssa_len(A); @*/ ;

// Bonus function
void ssa_print(ssa_t A)
/*@requires A != NULL;               @*/ ;
```

### Pixel

```
/*************************** Implementation ***************************/

typedef int[] pixel;

pixel make_pixel(int alpha, int red, int green, int blue)
//Not including other contracts here as this is part of assignment
//@ensures \length(\result) == 4;
{
    pixel p = alloc_array(int, 4);
    p[0] = alpha;
    p[1] = red;
    p[2] = green;
    p[3] = blue;
    return p;
}

pixel red() { return make_pixel(255, 255, 0, 0); }
pixel green() { return make_pixel(255, 0, 255, 0); }
pixel blue() { return make_pixel(255, 0, 0, 255); }
pixel white() { return make_pixel(255, 255, 255, 255);}

int get_alpha(pixel p)
//@requires \length(p) == 4;
//Not including other contracts here as this is part of assignment
{
    return p[0];
}

int get_red(pixel p)
//@requires \length(p) == 4;
//Not including other contracts here as this is part of assignment
{
    return p[1];
```

```
}

int get_green(pixel p)
//@requires \length(p) == 4;
//Not including other contracts here as this is part of assignment
{
    return p[2];
}

int get_blue(pixel p)
//@requires \length(p) == 4;
//Not including other contracts here as this is part of assignment
{
    return p[3];
}


// Client type
typedef pixel pixel_t;

/**************************** Interface ****************************/

// typedef ____ pixel_t;


pixel_t make_pixel(int alpha, int red, int green, int blue)
// contract omitted -- assignment
;

int get_alpha(pixel_t p)
// contracts omitted -- assignment
;

int get_red(pixel_t p)
// contracts omitted -- assignment
;

int get_blue(pixel_t p)
// contracts omitted -- assignment
;

int get_green(pixel_t p)
// contracts omitted -- assignment
;
```

### Queue

```
/*
 * Queues
 *
 * 15-122 Principles of Imperative Computation */
```

```
#use <conio>

/**************************************************************************/
/************************* BEGIN IMPLEMENTATION *************************/

/* Aux structure of linked lists */
typedef struct list_node list;
struct list_node {
  string data;
  list* next;
};

bool is_acyclic(list* start) {
  if (start == NULL) return true;
  list* h = start->next;         // hare
  list* t = start;               // tortoise
  while (h != t) {
    if (h == NULL || h->next == NULL) return true;
    h = h->next->next;
    //@assert t != NULL; // hare is faster and hits NULL quicker
    t = t->next;
  }
  //@assert h == t;
  return false;
}

/* is_segment(start, end) will diverge if list is circular! */
// Recursive version
bool is_segment(list* start, list* end) {
  if (start == NULL) return false;
  if (start == end) return true;
  return is_segment(start->next, end);
}
// Iterative version using a while loop
bool is_segmentB(list* start, list* end) {
  list* l = start;
  while (l != NULL) {
    if (l == end) return true;
    l = l->next;
  }
  return false;
}
// Iterative version using a for loop
bool is_segmentC(list* start, list* end) {
  for (list* p = start; p != NULL; p = p->next) {
    if (p == end) return true;
  }
  return false;
}

// Will run for ever if the segment is circular
void print_segment(list* start, list* end)
```

```c
//requires start != NULL;
{
  for (list* p = start; p != end; p = p->next) {
    printf("%s", p->data);
    if (p != end) printf("->");
  }
}


/* Queues */

typedef struct queue_header queue;
struct queue_header {
  list* front;
  list* back;
};

void queue_print_unsafe(queue* Q)
{
  printf("[front] ");
  print_segment(Q->front, Q->back);
  printf(" [back]");
}

bool is_queue(queue* Q) {
  return Q != NULL
      && is_acyclic(Q->front)
      && is_segment(Q->front, Q->back);
}

void print_queue_internal(queue* Q)
//@requires is_queue(Q);
{
  queue_print_unsafe(Q);
}

bool queue_empty(queue* Q)
//@requires is_queue(Q);
{
  return Q->front == Q->back;
}

queue* queue_new()
//@ensures is_queue(\result);
//@ensures queue_empty(\result);
{
  queue* Q = alloc(queue);
  list* l = alloc(list);
  Q->front = l;
  Q->back = l;
  return Q;
}
```

```
void enq(queue* Q, string s)
//@requires is_queue(Q);
//@ensures is_queue(Q);
//@ensures !queue_empty(Q);
{
  list* l = alloc(list);
  Q->back->data = s;
  Q->back->next = l;
  Q->back = l;
}

string deq(queue* Q)
//@requires is_queue(Q);
//@requires !queue_empty(Q);
//@ensures is_queue(Q);
{
  string s = Q->front->data;
  Q->front = Q->front->next;
  return s;
}

void queue_print(queue* Q)
//@requires is_queue(Q);
{
  printf("FRONT: ");
  for (list* l = Q->front; l != Q->back; l = l->next) {
    printf("%s", l->data);
    if (l->next != Q->back) printf(" << ");
  }
  printf(" :BACK");
}

// Client type
typedef queue* queue_t;

/************************** END IMPLEMENTATION *************************/
/*********************************************************************/

/*********************************************************************/
/******************************* Interface ******************************/

// typedef ______* queue_t;

bool queue_empty(queue_t Q)       /* O(1) */
/*@requires Q != NULL; @*/ ;

queue_t queue_new()               /* O(1) */
/*@ensures \result != NULL; @*/
/*@ensures queue_empty(\result); @*/ ;

void enq(queue_t Q, string e)     /* O(1) */
```

```
/*@requires Q != NULL; @*/
/*@ensures !queue_empty(Q); @*/ ;

string deq(queue_t Q)             /* O(1) */
/*@requires Q != NULL; @*/
/*@requires !queue_empty(Q); @*/ ;

void queue_print(queue_t Q)       /* O(n) */
/*@requires Q != NULL; @*/ ;
```

**Stack**

```
/*
 * Stacks
 *
 * 15-122 Principles of Imperative Computation */

#use <conio>

/************************************************************************/
/************************* BEGIN IMPLEMENTATION *************************/

/* Aux structure of linked lists */
typedef struct list_node list;
struct list_node {
  string data;
  list* next;
};

bool is_acyclic(list* start) {
  if (start == NULL) return true;
  list* h = start->next;         // hare
  list* t = start;               // tortoise
  while (h != t) {
    if (h == NULL || h->next == NULL) return true;
    h = h->next->next;
    //@assert t != NULL; // hare is faster and hits NULL quicker
    t = t->next;
  }
  //@assert h == t;
  return false;
}

/* is_segment(start, end) will diverge if list is circular! */
bool is_segment(list* start, list* end) {
  if (start == NULL) return false;
  if (start == end) return true;
  return is_segment(start->next, end);
}

// Will run for ever if the segment is circular
void print_segment(list* start, list* end)
```

```
//requires start != NULL;
{
  for (list* p = start; p != end; p = p->next) {
    printf("%s", p->data);
    if (p != end) printf("->");
  }
}


/* Stacks */

typedef struct stack_header stack;
struct stack_header {
  list* top;
  list* floor;
};

void stack_print_unsafe(stack* S)
{
  printf("[top] ");
  print_segment(S->top, S->floor);
}

bool is_stack(stack* S) {
  return S != NULL
      && is_acyclic(S->top)
      && is_segment(S->top, S->floor);
}

void stack_print_internal(stack* S)
//@requires is_stack(S);
{
  stack_print_unsafe(S);
}

bool stack_empty(stack* S)
//@requires is_stack(S);
{
  return S->top == S->floor;
}

stack* stack_new()
//@ensures is_stack(\result);
//@ensures stack_empty(\result);
{
  stack* S = alloc(stack);
  list* l = alloc(list);
  S->top = l;
  S->floor = l;
  return S;
}
```

```
void push(stack* S, string x)
//@requires is_stack(S);
//@ensures is_stack(S);
//@ensures !stack_empty(S);
{
  list* l = alloc(list);
  l->data = x;
  l->next = S->top;
  S->top = l;
}

string pop(stack* S)
//@requires is_stack(S);
//@requires !stack_empty(S);
//@ensures is_stack(S);
{
  string e = S->top->data;
  S->top = S->top->next;
  return e;
}

void stack_print(stack* S)
//@requires is_stack(S);
{
  printf("TOP: ");
  for (list* l = S->top; l != S->floor; l = l->next) {
    printf("%s", l->data);
    if (l->next != S->floor) printf(" | ");
  }
}

// Client type
typedef stack* stack_t;

/************************** END IMPLEMENTATION **************************/
/************************************************************************/
```

```
/************************************************************************/
/******************************* Interface ******************************/

// typedef ______* stack_t;

bool stack_empty(stack_t S)       /* O(1) */
/*@requires S != NULL; @*/ ;

stack_t stack_new()               /* O(1) */
/*@ensures \result != NULL; @*/
/*@ensures stack_empty(\result); @*/ ;

void push(stack_t S, string x)    /* O(1) */
/*@requires S != NULL; @*/
/*@ensures !stack_empty(S); @*/ ;
```

```
string pop(stack_t S)             /* O(1) */
/*@requires S != NULL; @*/
/*@requires !stack_empty(S); @*/ ;

void stack_print(stack_t S)       /* O(n) */
/*@requires S != NULL; @*/ ;
```

### Generic Dictionaries

```
/*
 * Generic dictionaries, implemented with separate chaining
 *
 * 15-122 Principles of Imperative Computation
 */

#use <util>
#use <conio>

/************************************************************************/
/**************************** Client Interface *************************/

typedef void* entry;
typedef void* key;

typedef key entry_key_fn(entry x)           // Supplied by client
          /*@requires x != NULL; @*/ ;
typedef int key_hash_fn(key k);             // Supplied by client
typedef bool key_equiv_fn(key k1, key k2);  // Supplied by client

/************************* End Client Interface ************************/
/************************************************************************/

/************************************************************************/
/************************* BEGIN IMPLEMENTATION ************************/

typedef struct chain_node chain;
struct chain_node {
  entry  data;           // != NULL; contains both key and value
  chain* next;
};

typedef struct hdict_header hdict;
struct hdict_header {
  int size;              // 0 <= size
  chain*[] table;        // \length(table) == capacity
  int capacity;          // 0 < capacity
  entry_key_fn* key;     // != NULL
  key_hash_fn* hash;     // != NULL
  key_equiv_fn* equiv;   // != NULL
};
```

```
bool is_table_expected_length(chain*[] table, int length) {
  //@assert \length(table) == length;
  return true;
}

bool is_hdict(hdict* H) {
  return H != NULL
      && H->capacity > 0
      && H->size >= 0
      && H->key    != NULL
      && H->hash   != NULL
      && H->equiv != NULL
      && is_table_expected_length(H->table, H->capacity);
 /*   && there aren't entries with the same key  */
 /*   && the number of entries matches the size */
 /*   && every entry in H->table[i] hashes to i */
 /*   && ... */
}

int index_of_key(hdict* H, key k)
//@requires is_hdict(H);
//@ensures 0 <= \result && \result < H->capacity;
{
  return abs((*H->hash)(k) % H->capacity);
}

key entry_key(hdict* H, entry x)
//@requires is_hdict(H) && x != NULL;
{
  return (*H->key)(x);
}

bool key_equiv(hdict* H, key k1, key k2)
//@requires is_hdict(H);
{
  return (*H->equiv)(k1, k2);
}

hdict* hdict_new(int capacity,
                 entry_key_fn* entry_key,
                 key_hash_fn* hash,
                 key_equiv_fn* equiv)
//@requires capacity > 0;
//@requires entry_key != NULL && hash != NULL && equiv != NULL;
//@ensures is_hdict(\result);
{
  hdict* H = alloc(hdict);
  H->size = 0;
  H->capacity = capacity;
  H->table = alloc_array(chain*, capacity);
  H->key    = entry_key;
  H->hash   = hash;
```

```
  H->equiv = equiv;
  return H;
}

entry hdict_lookup(hdict* H, key k)
//@requires is_hdict(H);
//@ensures \result == NULL || key_equiv(H, entry_key(H, \result), k);
{
  int i = index_of_key(H, k);
  for (chain* p = H->table[i]; p != NULL; p = p->next) {
    if (key_equiv(H, entry_key(H, p->data), k))
      return p->data;
  }
  return NULL;
}

void hdict_insert(hdict* H, entry x)
//@requires is_hdict(H);
//@requires x != NULL;
//@ensures is_hdict(H);
//@ensures hdict_lookup(H, entry_key(H, x)) == x;
{
  key k = entry_key(H, x);
  int i = index_of_key(H, k);
  for (chain* p = H->table[i]; p != NULL; p = p->next) {
    //@assert p->data != NULL;  // From preconditions -- operational reasoning!
    if (key_equiv(H, entry_key(H, p->data), k)) {
      p->data = x;
      return;
    }
  }

  // prepend new entry
  chain* p = alloc(chain);
  p->data = x;
  p->next = H->table[i];
  H->table[i] = p;
  (H->size)++;
}

// Statistics
int chain_length(chain* p) {
  int i = 0;
  while (p != NULL) {
    i++;
    p = p->next;
  }
  return i;
}

// report the distribution stats for the hashtable
void hdict_stats(hdict* H)
```

```
//@requires is_hdict(H);
{
  int max = 0;
  int[] A = alloc_array(int, 11);
  for(int i = 0; i < H->capacity; i++) {
    int j = chain_length(H->table[i]);
    if (j > 9) A[10]++;
    else A[j]++;
    if (j > max) max = j;
  }

  printf("Hash table distribution: how many chains have size...\n");
  for(int i = 0; i < 10; i++) {
    printf("...%d:   %d\n", i, A[i]);
  }
  printf("...10+: %d\n", A[10]);
  printf("Longest chain: %d\n", max);
}

// Client-side type
typedef hdict* hdict_t;

/************************** END IMPLEMENTATION *************************/
/***********************************************************************/

/***********************************************************************/
/*************************** Library Interface *************************/

// typedef ______* hdict_t;

hdict_t hdict_new(int capacity,
                  entry_key_fn* entry_key,
                  key_hash_fn* hash,
                  key_equiv_fn* equiv)
/*@requires capacity > 0; @*/
/*@requires entry_key != NULL && hash != NULL && equiv != NULL; @*/
/*@ensures \result != NULL; @*/ ;

entry hdict_lookup(hdict_t H, key k)
/*@requires H != NULL; @*/ ;

void hdict_insert(hdict_t H, entry x)
/*@requires H != NULL && x != NULL; @*/ ;
```