Introduction to Hash Table Lab

CMPUT 229

José Nelson Amaral

Lab Requirements

RISC-V

- Function calls and register convention.
- Loading and storing from memory.

General

- Hash tables, key/value stores.
- Hashing.
- Linked Lists.
- Dynamic Memory Allocation.

Overview

“Number of students enrolled in a course”

Key		Value
CMPUT 229	→	154
STAT 151	→	98
CHEM 101	→	248

Functions

- Insert (aka. Add, Append)
- Find (aka. Get)
- Delete (aka. Remove)

Hashing Functions

- Calculates the same hash for each unique input.
- Predictable outputs.
- Hard to reverse the process.
- Multiple inputs can have the same hash.
- Pseudorandom.

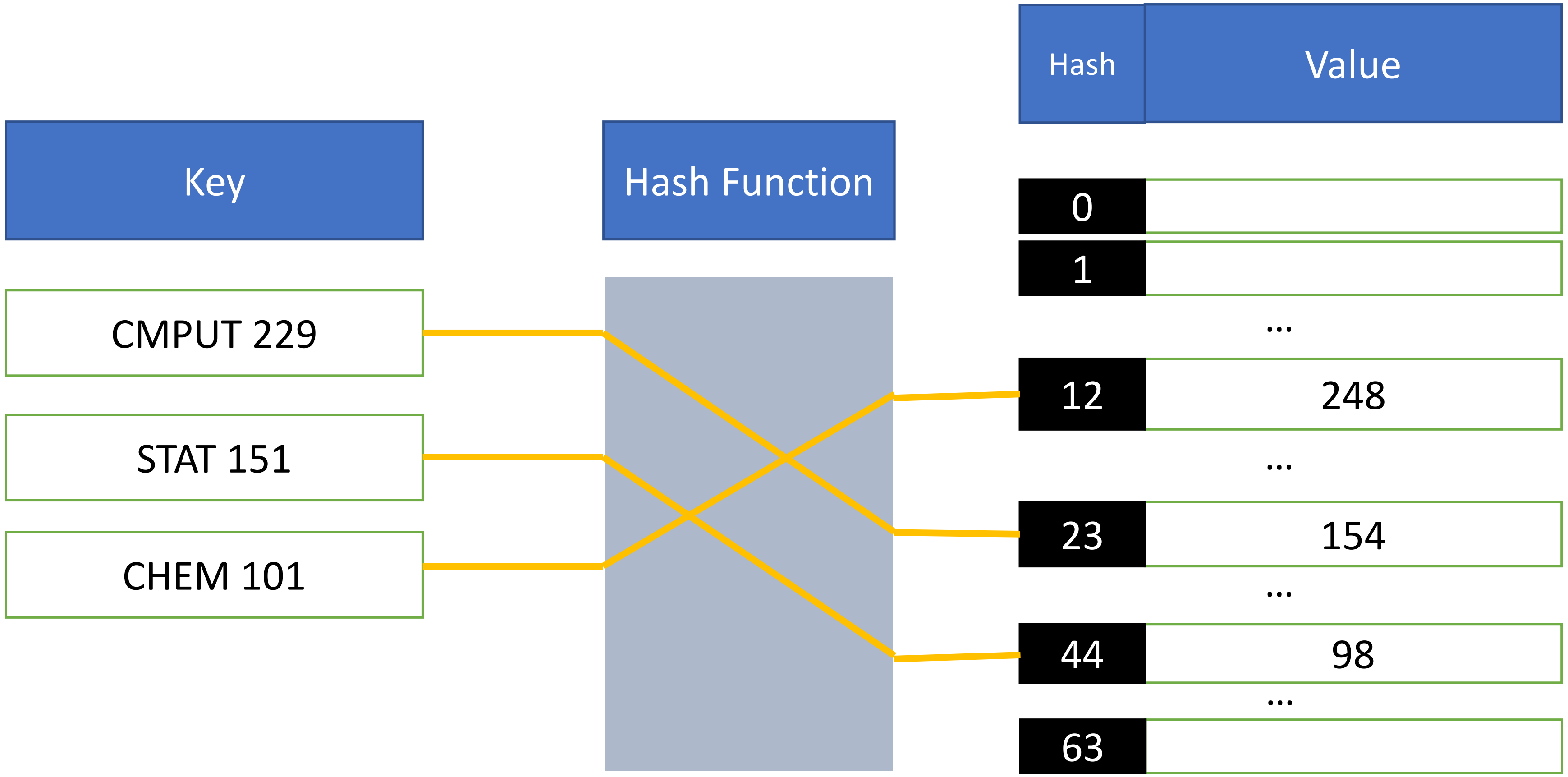
Some popular hashing algorithms

MD5

SHA256

Hash Table

"Number of students enrolled in a course"



(Singly) Linked Lists

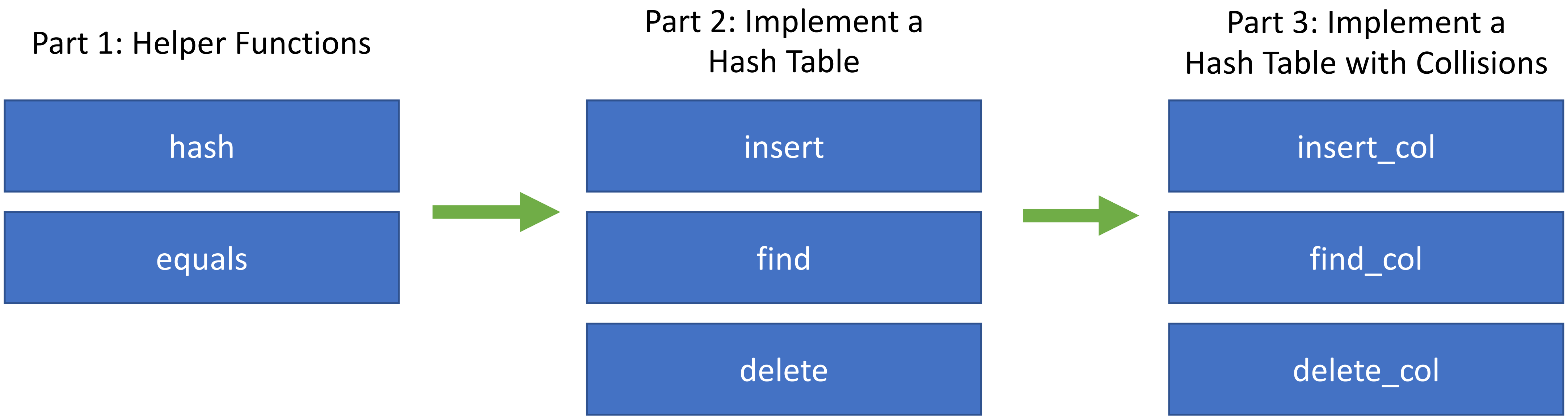
- Two parts: data and pointer to next item.
- Easy to grow and shrink.
- Don't need to know the full size at creation.
- Slower than typical arrays, where data is right next to each other.

Dynamic Memory Allocation

- If you don’t know how much memory you need while you’re writing the program, you can ask the OS to allocate memory for you dynamically, or while the program is running.
- Requires you to manage memory in code (needing to allocate and deallocate/free).
- For part 3 of this lab, you can call the function **alloc** to provide you with an area of memory that you can use to grow the list.

Creating a Hash Table in RISC-V

- This lab is split into 3 parts:



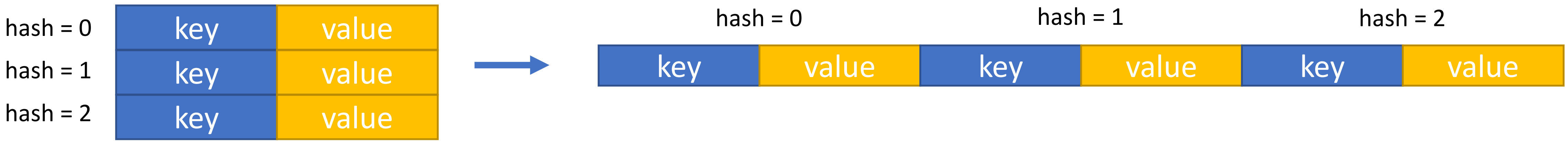
Detailed videos on how each of the algorithms work are embedded on the lab page.

Representing the Entries as an Array

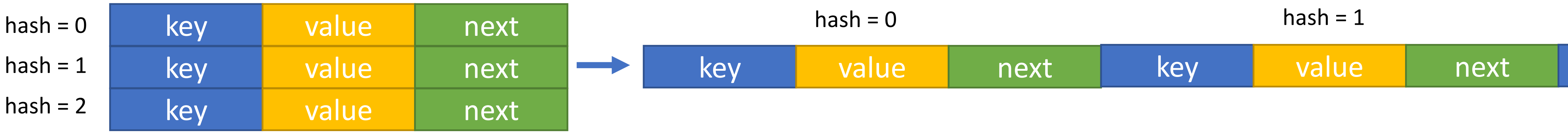
Every block in the diagram is 1 word or 4 bytes long.

- In this lab, you are given an array for storing items in the hash table.
- We have to represent a 2D array in memory, which is one dimension.
- The layout is different in part 2 and part 3, so pay attention!

PART 2



PART 3



Part 1

- You will implement the functions **hash** and **equals**.
- The hash algorithm is a modified version of *djb2*, using the seed ***5381***.
- **equals** checks if 2 strings are equal.

```
hash <- seed

for every character char in string do
  hash <- ((hash * 33) + char) % 22900
end for

hash <- hash % 64
return hash
```

Part 2

- You will implement a hash table (**insert**, **find**, **delete**), with no collisions.
- You must call functions **hash** and **equals**.
- **hash** is used to calculate the index of the array.
- For **find** and **delete**, you must first check that the key in the hash table matches the key that is given in the function using **equals**.
- We will not try to insert 2 items with the same name or the same hash (no collisions).

Part 3

- You will implement a hash table, with collisions
- You must call **hash**, **equals**, and **alloc**.
- Collisions occur when you try to insert two items that have the same hash.
- We will use a linked list to store overflow (extra) entries. You can call **alloc** to allocate room for the overflow entries.
- There are a lot of edge cases in these scenarios, so be vigilant!

Register Convention in RISC-V

- You will be using and possibly writing functions in this lab.
- Review your notes on using the stack pointer and register convention.

Register Usage Conventions

Register	Name	Use	Saver
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller

taken from slides V0C

File Templates

- Write your solution in *hashtable.s.*
- Do not rename this file. We will be marking this file only.
- Function headers and incomplete comment blocks are provided.

```
#-------------------------------------------------------------------------------
# hash
# This function hashes a string using a modified djb2 algorithm.
#
# Args:
#   a0: pointer to string
#
# Returns:
#   a0: hash
#
# Register Usage:
#   --- insert your register usage here ---
#-------------------------------------------------------------------------------
hash:

# --- insert your solution to part 1a here ---

ret


# --- PART 1B ---


#-------------------------------------------------------------------------------
# equals
# This function checks if 2 strings are entirely equal.
#
# Args:
#   a0: pointer to string 1
#   a1: pointer to string 2
#
# Returns:
#   a0: 1 if both strings are entirely equal, 0 if not.
#
# Register Usage:
#   --- insert your register usage here ---
#-------------------------------------------------------------------------------
equals:

# --- insert your solution to part 1b here ---

ret
```

Testing

- Integrated test cases in RARS.
- They do not cover every case!
- playground.s and playground_col.s can be used to test your functions.

```
-- Running tests for part 1a: hash --
:  [X] Great job!     :  [X] Great job!

-- Running tests for part 1b: equals --
:  [ ] Almost there!  :  [ ] Almost there!
```

```
#------------------------------------------------------------------------------
# playground
# This function tests your code and displays a representation of the hash table.
#
# Args:
#       a0: pointer to hash table (storearray)
#------------------------------------------------------------------------------
playground:

    # save registers
    addi sp, sp, -8
    sw ra, 0(sp)
    sw s0, 4(sp)
    mv s0, a0 # the pointer to storearray is stored in s0

    # --- test your code for part 2 (insert, find, delete) here ---

    # --- use the following example to insert a value ---
    # mv a0, s0
    # la a1, input_1
    # li a2, 154
    # jal insert

    # restore registers
    lw ra, 0(sp)
    lw s0, 0(sp)
    addi sp, sp, 8

    ret
```

Final Remarks

- Watch the videos on the instruction page!
- You can do the tasks <u>within</u> each part in any order.
- Write your own test cases in the playground files that handle edge cases.
- Don’t be afraid to ask questions.
- Review the marksheet to see what cases we are testing for.
- Style marks are an easy 20%.

```
                CMPUT 229 Laboratory Assignment - Hash Table
                ============================================

Final Mark:           /100 MARKS TOTAL
                =====

Part 1:       / 15 Implementing hash and equals

Part 2:       / 25 Implementing hash table, without collisions

Part 3:       / 40 Implementing hash table, with collisions

Part 4:       / 20 Style

------------------------------------------------------------------------
Part 1: Implementing hash and equals

              / 15

                      / 10 Hash

                      / 5  Equals
                              / 2  equal strings
                              / 3  non-equal strings

------------------------------------------------------------------------
Part 2: Implementing hash table, without collisions

              / 25

                      / 6  Insert

                      / 9  Find
                              / 4  value exists
                              / 3  value does not exist, same hash not occupied
                              / 2  value does not exist, same hash occupied

                      / 10 Delete
                              / 5  value exists
                              / 3  value does not exist, same hash not occupied
                              / 2  value does not exist, same hash occupied

Part 3: Implementing hash table, with collisions

              / 40

                      / 10 Insert
                              / 2  same hash not occupied
                              / 3  same hash occupied, without overflow
                              / 5  same hash occupied, with overflow

                      / 14 Find
                              / 1  value does not exist, same hash not occupied
                              / 1  value does not exist, same hash occupied, without overflow
                              / 2  value does not exist, same hash occupied, with overflow
                              / 2  value exists, without overflow
                              / 3  value exists, not in overflow
                              / 5  value exists, in overflow

                      / 16 Delete
                              / 1  value does not exist, same hash not occupied
                              / 1  value does not exist, same hash occupied, without overflow
                              / 2  value does not exist, same hash occupied, with overflow
                              / 2  value exists, without overflow
                              / 4  value exists, not in overflow
                              / 6  value exists, in overflow

------------------------------------------------------------------------
Part 4: Style

              / 20

        Some common deductions will include: (8 marks per deduction)
                No subroutine description
                No program header
                No explanation for register usage
                No block comments
                Incorrect submission file name or location in repo
```