CSE 220: Systems Fundamentals I

Stony Brook University

Programming Project #4

Fall 2019

Assignment Due: Friday, December 6, 2019 by 11:59 pm

Updates to the Document:

- 11/24/2019: Expected return values were provided for the example in Part VII on page 16.
- 11/24/2019: Two typos in the algorithm on page 14 have been corrected and highlighted in red.
- 11/23/2019: Ignore the previous update! The diagram as given is correct: byte #8 is the destination address and byte #9 is the source address.

Learning Outcomes

After completion of this programming project you should be able to:

- Implement non-trivial algorithms that require conditional execution and iteration.
- Design and code functions that implement the MIPS assembly register conventions.
- Implement algorithms that process structs and 2D arrays of bytes.
- Implement a priority queue using an array of pointers to structs.

Getting Started

Visit Piazza and download the file `proj4.zip`. Decompress the file and then open `proj4.zip`. Fill in the following information at the top of `proj4.asm`:

1. your first and last name as they appear in Blackboard
2. your Net ID (e.g., jsmith)
3. your Stony Brook ID # (e.g., 111999999)

Having this information at the top of the file helps us locate your work. If you forget to include this information but don't remember until after the deadline has passed, don't worry about it – we will track down your submission.

Inside `proj4.asm` you will find several function stubs that consist simply of `jr $ra` instructions. Your job in this assignment is implement all the functions as specified below. Do not change the function names, as the grading scripts will be looking for functions of the given names. However, you may implement additional helper functions of your own, but they must be saved in `proj4.asm`. Helper functions will not be graded.

If you are having difficulty implementing these functions, write out pseudocode or implement the functions in a higher-level language first. Once you understand the algorithm and what steps to perform, then translate the logic into MIPS assembly code.

Be sure to initialize all of your values (e.g., registers) within your functions. Never assume registers or memory will hold any particular values (e.g., zero). MARS initializes all of the registers and bytes of main memory to zeroes. The grading scripts will fill the registers and/or main memory with random values before calling your functions.

Finally, do not define a `.data` section in your `proj4.asm` file. A submission that contains a `.data` section will probably receive a score of zero.

Important Information about CSE 220 Homework Assignments

- Read the entire homework documents twice before starting. Questions posted on Piazza whose answers are clearly stated in the documents will be given lowest priority by the course staff.
- **You must use the Stony Brook version of MARS posted on Blackboard.** Do not use the version of MARS posted on the official MARS website. The Stony Brook version has a reduced instruction set, added tools, and additional system calls you might need to complete the homework assignments.
- When writing assembly code, try to stay consistent with your formatting and to comment as much as possible. It is much easier for your TAs and the professor to help you if we can quickly figure out what your code does.
- You personally must implement the programming projects in MIPS Assembly language by yourself. You may not write or use a code generator or other tools that write any MIPS code for you. You must manually write all MIPS assembly code you submit as part of the assignments.
- Do not copy or share code. Your submissions will be checked against other submissions from this semester and from previous semesters.
- Do not submit a file with the function/label `main` defined. You are also not permitted to start your label names with two underscores (__). You will obtain a zero for an assignment if you do this.
- Submit your final `.asm` file to Blackboard by the due date and time. Late work will not be accepted or graded. Code that crashes and cannot be graded will earn no credit. No changes to your submission will be permitted once the deadline has passed.

How Your CSE 220 Assignments Will Be Graded

With minor exceptions, all aspects of your homework submissions will be graded entirely through automated means. Grading scripts will execute your code with input values (e.g., command-line arguments, function arguments) and will check for expected results (e.g., print-outs, return values, etc.) For this homework assignment you will be writing *functions* in assembly language. The functions will be tested independently of each other. This is very important to note, as you must take care that no function you write ever has side-effects or requires that other functions be called before the function in question is called. Both of these are generally considered bad practice in programming.

Some other items you should be aware of:

- Each test case must execute in 1,000,000 instructions or fewer. Efficiency is an important aspect of programming. This maximum instruction count will be increased in cases where a complicated algorithm might be necessary, or a large data structure must be traversed. To find the instruction count of your code in MARS, go to the **Tools** menu and select **Instruction Statistics**. Press the button marked **Connect to MIPS**. Then assemble and run your code as normal.

- Any excess output from your program (debugging notes, etc.) might impact grading. Do not leave erroneous print-outs in your code.
- We will provide you with a small set of test cases for each assignment to give you a sense of how your work will be graded. It is your responsibility to test your code thoroughly by creating your own test cases.
- The testing framework we use for grading your work will not be released, but the test cases and expected results used for testing will be released.

Register Conventions

You must follow the register conventions taught in lecture and reviewed in recitation. Failure to follow them will result in loss of credit when we grade your work. Here is a brief summary of the register conventions and how your use of them will impact grading:

- It is the callee's responsibility to save any `$s` registers it overwrites by saving copies of those registers on the stack and restoring them before returning.
- If a function calls a secondary function, the caller must save `$ra` before calling the callee. In addition, if the caller wants a particular `$a`, `$t` or `$v` register's value to be preserved across the secondary function call, the best practice would be to place a copy of that register in an `$s` register before making the function call.
- A function which allocates stack space by adjusting `$sp` must restore `$sp` to its original value before returning.
- Registers `$fp` and `$gp` are treated as preserved registers for the purposes of this course. If a function modifies one or both, the function must restore them before returning to the caller. There really is no reason for your code to touch the `$gp` register, so leave it alone.

The following practices will result in loss of credit:

- "Brute-force" saving of all `$s` registers in a function or otherwise saving `$s` registers that are not overwritten by a function.
- Callee-saving of `$a`, `$t` or `$v` registers as a means of "helping" the caller.
- "Hiding" values in the `$k`, `$f` and `$at` registers or storing values in main memory by way of offsets to `$gp`. This is basically cheating or at best, a form of laziness, so don't do it. We will comment out any such code we find.

How to Test Your Functions

To test your implementated functions, open the provided `main` files in MARS. Next, assemble the `main` file and run it. MARS will include the contents of any `.asm` files referenced with the `.include` directive(s) at the end of the file and then enqueue the contents of your `proj4.asm` file before assembling the program.

Each main file calls a single function with one of the sample test cases and prints any return value(s). You will need to change the arguments passed to the functions to test your functions with the other cases. To test each of your functions thoroughly, create your own test cases in those main files. Your submission will not be graded using the examples provided in this document or using the provided main file(s). Do not submit your main files to Blackboard – we will delete them.

Again, any modifications to the `main` files will not be graded. You will submit only your `proj4.asm` for grading. Make sure that all code required for implementing your functions is included in the `proj4.asm` file. To make sure that your code is self-contained, try assembling your `proj4.asm` file by itself in MARS. If you get any errors (such as a missing label), this means that you need to refactor (reorganize) your code, possibly by moving labels you inadvertently defined in a `main` file (e.g., a helper function) to `proj4.asm`.

A Reminder on How Your Work Will be Graded

It is **imperative** (crucial, essential, necessary, critically important) that you implement the functions below exactly as specified. Do not deviate from the specifications, even if you think you are implementing the program in a better way. Modify the contents of memory only as described in the function specifications!

Network Packet Processing

In this assignment you will be implementing a collection of functions to simulate the "packetizing" and reassembly of a message transmitted through a network. As part of this effort you will implement a priority queue as a binary min heap data structure using a binary tree of pointers to structs. The pointers will be stored in an array, as opposed to a linked structure, to make the algorithms easier to implement.

Data Structures

The assignment involves two main data structures, `Packet`, which represents a network packet for a fictional network protocol, and `PriorityQueue`, which represents a priority queue of `Packet` objects.

In the code below, the notation `:n` indicates that a field consumes `n` bits of memory. Below the definition of `Packet` we provide a graphic of how the bits of a `Packet` are laid out in memory.

```
struct Packet {
    unsigned int total_length : 16;
    unsigned int msg_id : 12;
    unsigned int version : 4;
    unsigned int fragment_offset : 12;
    unsigned int protocol : 10;
    unsigned int flags : 2;
    unsigned int priority : 8;
    unsigned int dest_addr : 8;
    unsigned int src_addr : 8;
    unsigned int checksum : 16;
    char[] payload;
}
```

The first 96 bits (three 32-bit words) comprise the **header** of the packet. The **payload** is appended to the header and contains the actual data carried by the packet.

	byte #3								byte #2								byte #1								byte #0							
	version				message ID#												total length															
bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

	byte #7								byte #6								byte #5								byte #4							
	priority								flags		protocol						fragment offset															
bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

	byte #11								byte #10								byte #9								byte #8							
	checksum																source address								destination address							
bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

	byte #15								byte #14								byte #13								byte #12							
																									payload starts here							
bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The `total_length` field indicates the total size in bytes of the packet, including the 12-byte header and the payload, which follows immediately after the header, beginning at byte #12. The payloads for this assignment will always be strings. In particular, an original message will be packetized into a group of `Packet` objects, with each object's payload carrying part of the original string. The `fragment_offset` field indicates the starting index of the substring from the original message. This offset is needed to reconstruct the original message (identified by the `msg_id` field) by the receiver. For all packets, except for the last packet of a message, `flags` is `0b01`. For the final packet, `flags` is `0b00`. The `checksum` field provides a numerical code that can be used to verify that the bytes of the packet were transmitted correctly.

```
struct PriorityQueue {
    short size;      // # of items in the queue; lower half of a full word
    short max_size;  // max # of items that can be in the queue
                     // upper half of a full word shared with "size"
    Packet*[] array; // array of pointers to Packet structs
                     // consumes 4*max_size bytes of memory
}
```

We see that a `PriorityQueue` struct consumes only $4 + 4 \times$`max_size` bytes of memory.

The values are stored in the minheap in the standard way for representing a complete binary tree, namely, the children (if any) of a node located at index `i` of the array are located at indices `2i+1` and `2i+2`. Likewise, the parent of a node at index `i` is located at index `floor((i-1)/2)`.

Example Packet

Suppose we want to transmit the message given below and use the provided arguments to packetize the message:

```
msg = "Grace Murray Hopper was one of the first computer programmers to work
       on the Harvard Mark I.\0"
payload_size = 24
version =      5
msg_id =       154
priority =     7
protocol =     289
```

```
src_addr =     161
dest_addr =    89
```

The value of `24` for `payload_size` means that the payload of a packet can contain up to `24` characters.

The first packet (of the four) would contain the following values for its fields:

```
Version:         5
Msg ID #:        154
Total Length:    36
Priority:        7
Flags:           0b01
Protocol:        289
Fragment Offset: 0
Checksum:        742
Source Address:  161
Dest Address:    89
Payload:         "Grace Murray Hopper was "
```

Below is shown the packet in hexadecimal, with one word per line. Remember that our MIPS computer uses little-endian, so byte #0 of each word is on the right-hand side (the "little end") of the word:

```
50 9A 00 24
07 52 10 00
02 E6 A1 59
63 61 72 47
75 4D 20 65
79 61 72 72
70 6F 48 20
20 72 65 70
20 73 61 77
```

Let's try to make some sense of this. Consider the first word:

```
50 9A 00 24
```

These four bytes presumably represent the version, message ID# and total length. Let's see how. First write out the values in binary:

```
0101 0000 1001 1010 0000 0000 0010 0100
```

Now remove the spaces:

```
01010000100110100000000000100100
```

Now split these into the fields: 4 bits for the version, 12 bits for the message ID# and 16 bits for the total length:

```
0101 000010011010 0000000000100100
```

Convert these binary values into decimal:

```
5 154 36
```

which are the decimal values of the version, message ID# and total length, as expected. The remaining words of the packet can be "decoded" in a similar way, including the characters of the payload.

In a MIPS `.data` section, the packet would look like this:

```
.align 2
packet:
.byte 0x24 0x00 0x9A 0x50 0x00 0x10 0x52 0x07 0x59 0xA1 0xE6 0x02
.byte 0x47 0x72 0x61 0x63 0x65 0x20 0x4D 0x75 0x72 0x72 0x61 0x79
.byte 0x20 0x48 0x6F 0x70 0x70 0x65 0x72 0x20 0x77 0x61 0x73 0x20
```

The `.align 2` directive ensures that the packet begins in a word-aligned boundary.

The last packet would contain the following. Note that its payload is smaller than the payload of the others, resulting in a smaller total length:

```
Version:         5
Msg ID #:        154
Total Length:    33
Priority:        7
Flags:           0b00
Protocol:        289
Fragment Offset: 72
Checksum:        810
Source Address:  161
Dest Address:    89
Payload:         " the Harvard Mark I.\0"
```

You are strongly encouraged to write your a function to print the contents of a packet struct to help you with development and testing.

Part I: Compute a Packet's Checksum

```
int compute_checksum(Packet* packet)
```

This function takes a pointer to a packet and computes its checksum. The function does not write the checksum, or any other changes, to the packet itself.

The checksum is computed according the following simple formula. Note that the checksum field is unsigned, so all 16 bits of the field can be used to store magnitude.

$$(\text{version} + \text{msg\_id} + \text{total\_length} + \text{priority} + \text{flags} + \text{protocol} + \text{frag\_offset} + \text{src\_addr} + \text{dest\_addr}) \; mod \; 2^{16}$$

The function takes the following arguments, in this order:

- `packet`: a pointer to a `Packet` object.

Returns in `$v0`:

- the checksum of the packet, according to the formula given above

Additional requirements:

- The function must not write any changes to main memory.

Example:

For the first "Grace Murray Hopper" packet from above, the checksum is 742_{10}.

Part II: Compare Two Packets

`int compare_to(Packet* p1, Packet* p2)`

This function compares two packets, determining if they are "equal", or if one is "greater than" or "less than" another. This function will be used later to implement the enqueue and dequeue operations for the priority queue. The algorithm to determine the relationship between two packets is as follows:

```
if p1.msg_id < p2.msg_id then
    return -1
elif p1.msg_id > p2.msg_id then
    return 1
else
    if p1.fragment_offset < p2.fragment_offset then
        return -1
    elif p1.fragment_offset > p2.fragment_offset then
        return 1
    else:
        if p1.src_addr < p2.src_addr then
            return -1
        elif p1.src_addr > p2.src_addr then
            return 1
        else
            return 0
```

Return values have the following meaning:

- `-1`: packet 1 is less than packet 2
- `0`: packet 1 equals packet 2
- `1`: packet 1 is greater than packet 2

The function takes the following arguments, in this order:

- `p1`: a pointer to a `Packet` object
- `p2`: a pointer to a `Packet` object

Returns in `$v0`:

- `-1`, `0` or `1`, according to the algorithm above

Additional requirements:

- The function must not write any changes to main memory.

Example:

Packet #1:

```
Version:         5
Msg ID #:        154
Total Length:    36
Priority:        7
Flags:           0b01
Protocol:        289
Fragment Offset: 0
Checksum:        742
Source Address:  161
Dest Address:    89
Payload:         "Grace Murray Hopper was "
```

Packet #2:

```
Version:         5
Msg ID #:        154
Total Length:    36
Priority:        7
Flags:           0b01
Protocol:        289
Fragment Offset: 24
Checksum:        766
Source Address:  161
Dest Address:    89
Payload:         "one of the first compute"
```

In this case the function would return `-1`.

Part III: Packetize a Message into an Array of Packet Objects

```
int packetize(byte[] packet_data, string msg, int payload_size, int version,
              int msg_id, int priority, int protocol, int src_addr,
              int dest_addr)
```

This function takes a null-terminated string, `msg`, and breaks it into a series of packets, each with a payload of size `payload_size` that contains `payload_size` characters taken from `msg`. Packet #0's payload will contain characters `0..payload_size-1` of `msg`, packet #1 will contain characters

payload_size..2*payload_size-1, packet #2 will contain characters 2*payload_size..3*payload_size-1, and so on. The final packet will contain fewer than payload_size characters in its payload if the length of msg is not evenly divisible by payload_size. The function must *not* pad the payload of the final packet to increase its payload size to payload_size. However, the payload in the last packet of the array must include the null-terminator found at the end of msg.

The function takes the following arguments, in this order:

- packet_data: a word-aligned array of bytes large enough to store the packets. Note: this array will store the packets themselves, laid out sequentially and contiguously in memory. This is not an array of pointers.
- msg: the null-terminated string to packetize
- payload_size: the number of characters to extract from msg and store in each packet's payload
- version: the value of the version field in each packet
- msg_id: the value of the msg_id field in each packet. Available at 0($sp).
- priority: the value of the priority field in each packet. Available at 4($sp).
- protocol: the value of the protocol field in each packet. Available at 8($sp).
- src_addr: the value of the src_addr field in each packet. Available at 12($sp).
- dest_addr: the value of the dest_addr field in each packet. Available at 16($sp).

Note that the function must compute the total_length, flags and fragment_offset fields itself. After assigning values to all fields in a packet (except checksum), the function must call compute_checksum for the packet and write that checksum value into the packet's checksum field.

Returns in $v0:

- the number of packets created by the function

Additional requirements:

- The function must not write any changes to main memory except as specified.
- The function must call compute_checksum.

Example:

Suppose we want to transmit the message given below and use the provided arguments to packetize the message:

```
msg = "Grace Murray Hopper was one of the first computer programmers to work
       on the Harvard Mark I.\0"
payload_size = 12  <-------------- *** Note the payload size ***
version =      5
msg_id =       154
priority =     7
protocol =     289
src_addr =     161
```

```
dest_addr =    89
```

With only some details shown, the 8 packets generated would be:

```
Packet #0:
Total Length:    24
Flags:           0b01
Fragment Offset: 0
Checksum:        730
Payload:         "Grace Murray"

Packet #1:
Total Length:    24
Flags:           0b01
Fragment Offset: 12
Checksum:        742
Payload:         " Hopper was "

Packet #2:
Total Length:    24
Flags:           0b01
Fragment Offset: 24
Checksum:        754
Payload:         "one of the f"

Packet #3:
Total Length:    24
Flags:           0b01
Fragment Offset: 36
Checksum:        766
Payload:         "irst compute"

Packet #4:
Total Length:    24
Flags:           0b01
Fragment Offset: 48
Checksum:        778
Payload:         "r programmer"

Packet #5:
Total Length:    24
Flags:           0b01
Fragment Offset: 60
Checksum:        790
Payload:         "s to work on"

Packet #6:
```

```
Total Length:    24
Flags:           0b01
Fragment Offset: 72
Checksum:        802
Payload:         " the Harvard"

Packet #7:
Total Length:    21
Flags:           0b00
Fragment Offset: 84
Checksum:        810
Payload:         " Mark I.\0"
```

The full contents of the array (given as a MIPS `.data` section) is given in inside the testing main itself for this function.

Part IV: Initialize a Priority Queue Struct

`int clear_queue(PriorityQueue* queue, int max_queue_size)`

This function sets the `size` field of the given `PriorityQueue` to `0`, its `max_size` field to `max_queue_size`, and then assigns `0` to all `max_queue_size` elements of the `array` field. The function can assume that enough memory has been allocated to store the entire array. If `max_queue_size` $\leq$ `0`, the function returns `-1` and writes no changes to memory. Otherwise, the function updates the struct as described and returns `0`.

The function takes the following arguments, in this order:

- `queue`: a pointer to a `PriorityQueue` struct
- `max_queue_size`: the value that should be assigned to the `max_size` field of the `PriorityQueue` struct

Returns in `$v0`:

- `0` if `max_queue_size` is valid, or `-1` if `max_queue_size` ≤ 0

Additional requirements:

- The function must not write any changes to main memory except as specified.

Example:

For a call to `clear_queue` with `max_queue_size = 7` with random garbage initially stored in the 8 words of the `queue` struct, the resulting queue would have the following contents, given here as a group of 8 words in hexadecimal:

```
0x00070000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
```

The return value would be `0`, indicating success.

Part V: Insert (Enqueue) a Packet into a Priority Queue of Packets

`int enqueue(PriorityQueue* queue, Packet* packet)`

This function inserts the given packet into the given priority queue, adding `1` to the size of the queue. If the queue is already at its maximum size, then no insertion takes place. Initially the packet (or rather, the pointer to the packet) is stored at `queue.array[size]`. Then, so as to maintain the minheap property of the data structure and using the standard heapify-up algorithm, the packet is iteratively swapped with its parent while the packet is "less than" its parent. The function returns the size of the queue.

The function takes the following arguments, in this order:

- `queue`: a pointer to a `PriorityQueue` struct
- `packet`: a pointer to a `Packet` struct, which is to be inserted into the given priority queue

Returns in `$v0`:

- the size of the queue after the insertion has taken place. If the function is called on a full queue, the function simply returns `queue.max_size` and makes no changes to memory.

Additional requirements:

- The function must not write any changes to main memory except as specified.
- The function must call `compare_to`.

Example:

Suppose we want to enqueue all the packets given in the `packetize` example in the Part III of the assignment. Also suppose that the packets were enqueued in the order #4, #1, #0, #3, #7, #6, #2, #5. The resulting `queue.array` would contain pointers to the packets in the following order:

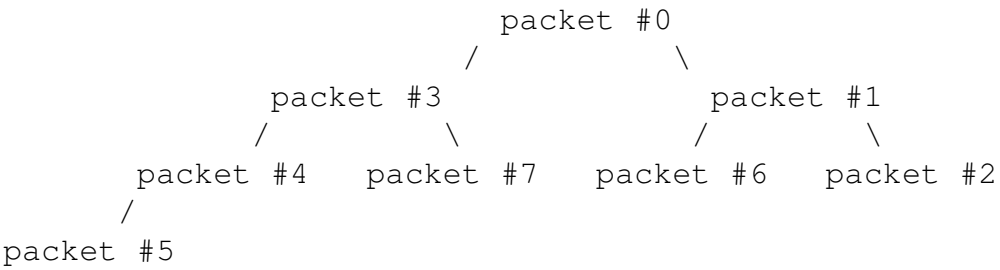
```
array[0]    array[1]    array[2]    array[3]
packet #0   packet #3   packet #1   packet #4

array[4]    array[5]    array[6]    array[7]
packet #7   packet #6   packet #2   packet #5

array[8]    array[9]    array[10]   array[11] ...
null ptr    null ptr    null ptr    null ptr  ...
```

where `null ptr` is short for "null pointer", indicating that the memory address `0x00000000` is stored at that index of the array.

Visualized as a binary tree:



Note that during the heapify-up algorithm we swap a node with its parent if the note is strictly less than its parent, so there is no ambiguity about the resulting binary tree you should build. *You must implement this algorithm in your code.* During grading we will inspect your `array` field to ensure that the heap was generated according to this algorithm.

Part VI: Remove (Dequeue) the Smallest Packet from a Priority Queue of Packets

```
Packet* dequeue(PriorityQueue* queue)
```

This function removes the "smallest" packet from a priority queue of packets (i.e., the root of the underlying binary tree), reheapifying the binary tree according to the regular algorithm for a minheap:

1. Move element `queue.array[queue.size-1]` to `queue.array[0]`.
2. Subtract 1 from `queue.size`.
3. Apply the heapify-down procedure to iteratively swap the moved element with the **smaller** of its children (if it has two children) or with its left child if the left child is **smaller** (if it has one child). If a node is larger than its two children and the two children are equal, the node should be swapped with its left child. The relationship between two packets is determined by the return value of the `compare_to` function.

If the queue is empty when the function is called, the function makes no changes to memory and returns `0`.

The function takes the following arguments, in this order:

- `queue`: a pointer to a `PriorityQueue` struct

Returns in `$v0`:

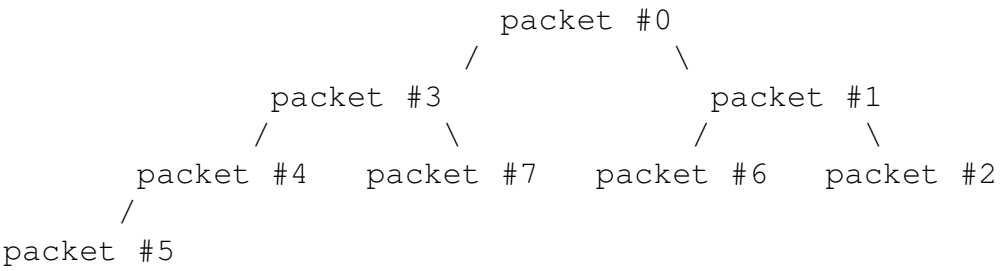
- the address of the dequeued struct, if the queue is not empty. If the function is called on an empty queue, the function simply returns `0`.

Additional requirements:

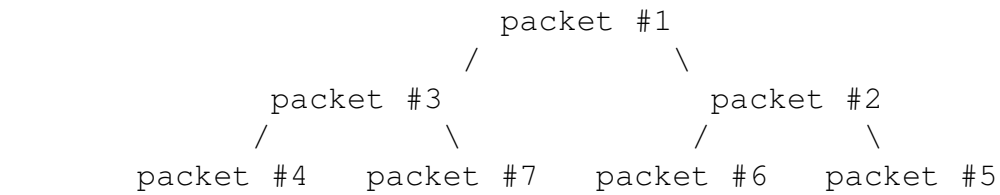
- The function must not write any changes to main memory.
- The function must call `compare_to`.

Example:

Assume the queue is identical to the queue from the example in the previous part of the assignment after Packet #5 has been inserted:



After Packet #0 has been removed, the binary tree will have the following structure:



and the function will have returned the address of Packet #0.

Part VII: Reassemble a Packetized Message

`(int, int) assemble_message(char[] msg, PriorityQueue* queue):`

This function iteratively removes all packets from the given priority queue, writing their payloads into `msg` using the `fragment_offset` and `payload` fields to determine where to write each payload into `msg`. For example, if the `fragment_offset` of a packet is `16` and the packet's `total_length` field is `32`, the function will write the 20 characters of the payload at indices `16..35` of `msg`. (Recall: the `total_length` field includes the header size.)

The function should not assume that every packet will have the same payload size. The provided example in the testing main for `assemble_message` gives an example where the payload sizes vary quite a bit.

As each packet is removed from the queue, the function verifies the packet's checksum. The function returns in `$v1` the number of packets whose computed checksum does not match the packet's `checksum` field.

Note: the function does *not* null-terminate the message. The expectation is that the original message was null-terminated.

The function takes the following arguments, in this order:

- `msg`: an uninitialized memory buffer to save the reassembled message
- `queue`: a pointer to a `PriorityQueue` struct

Returns in `$v0`:

- the number of packets dequeued from the queue

Returns in `$v1`:

- the number of packets whose checksums failed the checksum test

Additional requirements:

- The function must not write any changes to main memory.
- The function must call `compute_checksum`.

Example:

See the testing main for a comprehensive example. For this example, `$v0` should contain `6`, and `$v1` should contain `2`.

Academic Honesty Policy

Academic honesty is taken very seriously in this course. By submitting your work to Blackboard you indicate your understanding of, and agreement with, the following Academic Honesty Statement:

1. I understand that representing another person's work as my own is academically dishonest.
2. I understand that copying, even with modifications, a solution from another source (such as the web or another person) as a part of my answer constitutes plagiarism.
3. I understand that sharing parts of my homework solutions (text write-up, schematics, code, electronic or hard-copy) is academic dishonesty and helps others plagiarize my work.
4. I understand that protecting my work from possible plagiarism is my responsibility. I understand the importance of saving my work such that it is visible only to me.
5. I understand that passing information that is relevant to a homework/exam to others in the course (either lecture or even in the future!) for their private use constitutes academic dishonesty. I will only discuss material that I am willing to openly post on the discussion board.
6. I understand that academic dishonesty is treated very seriously in this course. I understand that the instructor will report any incident of academic dishonesty to the College of Engineering and Applied Sciences.
7. I understand that the penalty for academic dishonesty might not be immediately administered. For instance, cheating in a homework may be discovered and penalized after the grades for that homework have been recorded.
8. I understand that buying or paying another entity for any code, partial or in its entirety, and submitting it as my own work is considered academic dishonesty.
9. I understand that there are no extenuating circumstances for academic dishonesty.

How to Submit Your Work for Grading

To submit your `proj4.asm` file for grading:

1. Login to Blackboard and locate the course account for CSE 220.
2. Click on "Assignments" in the left-hand menu and click the link for this assignment.

3. Click the "Browse My Computer" button and locate the `proj4.asm` file. Submit only that one `.asm` file.
4. Click the "Submit" button to submit your work for grading.

Oops, I messed up and I need to resubmit a file!

No worries! Just follow the steps again. We will grade only your last submission.