Midterm Debrief

- Pick up or view your midterms during office hours today
- 2:30-4:30pm in my office hours
# Midterm Debrief

<table>
<thead>
<tr>
<th></th>
<th>Mean Grade</th>
<th>Median Grade</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>38.40 / 77.00</td>
<td>38.00 / 77.00</td>
<td>15.66</td>
</tr>
</tbody>
</table>

![Bar chart showing distribution of grades]
Studying for the finals: Goals

● **Good** student goals: *at least* be able to
  ○ Redo all exercises in class (e.g. proofs)
  ○ Reproduce all code in lectures on your own
  ○ Reproduce all assignment/midterm solutions on your own

● **Great** student goals: *in addition, at least* be able to
  ○ Teach someone else basic level tasks
  ○ Convert all pseudocode (e.g. in Stack lecture) to code
  ○ Recall details in material (e.g. quicksort worst-case run time)

● There’s nothing a good final can’t fix

● The above works for most courses!
```c
// Pre: arr[first..mid] and arr[mid+1..last] are sorted
// Post: arr[first..last] are sorted
void merge(int arr[], int first, int mid, int last) {
    int len = last - first + 1;  int newArr[len];
    int left = first;  int right = mid + 1;  int newPos = 0;
    while(left <= mid && right <= last) {
        if (arr[left] < arr[right]) {
            newArr[newPos++] = arr[left++];
        } else {
            newArr[newPos++] = arr[right++];
        }
    }
    // Flush non empty piece
    arrCpy(arr + left, newArr + newPos, mid - left + 1);
    arrCpy(arr + right, newArr + newPos, last - right + 1);
    arrCpy(newArr, arr + first, len);
}
```
General tips

- Make a cheat sheet, even though you will not be allowed on one on the final

- Regularly review and study, even if there is no due date
  - Your brain needs time to subconsciously process material

- Ask and answer questions on Piazza

- Reward yourself for studying and learning

- Create practice questions for others or do these questions on Piazza
Code up \texttt{pop}(S)

Q. From which end should you remove an item?

From the tail?

\begin{itemize}
  \item \texttt{return tail->data;}
  \item \texttt{free tail;}
\end{itemize}

\textbf{O(1) steps}

From the head?

\begin{itemize}
  \item \texttt{return head->data;}
  \item \texttt{free(head);} \quad \textbf{O(1) steps}
  \item \texttt{newhead = oldhead->next;}
  \item \texttt{O(N) steps to update tail}
\end{itemize}
Queue ADT

CMPT 125
Mo Chen
SFU Computing Science
6/3/2020
Lecture 22

Today:

- Queue ADT
- An algorithm that uses a Queue
- Implementing a Queue (with a Linked List)
- Information Hiding & Encapsulation - Part 1
Queue ADT (Review)

Queue ADT: A *queue* is a sequence of data, but the insert and remove operations work on opposite ends of the sequence.

- order is first-in-first-out (FIFO)
- like a line-up

Used in simulations and modeling

- to model sequences of work and their processors, e.g., assembly lines

*Operations Research (OR)*

Queue of items  
A processor
Queue-Based Searching (Breadth-First Search)

Problem: Find all locations that are reachable from the start, and compute their distance.

Algorithm:
Create an empty queue Q; enqueue start → Q
Initialize all distances ← -1 (unreachable), except distance(start) ← 0
while Q not empty {
    dequeue from Q → current
    if next is neighbour of current and distance(next) == -1 {
        distance(next) = distance(current) + 1
        enqueue next → Q
    }
}
Problem: Find all locations that are reachable from the start, and compute their distance.

Algorithm:
Create an empty queue Q; enqueue start → Q
Initialize all distances ← -1 (unreachable), except distance(start) ← 0
while Q not empty {
    dequeue from Q → current
    if next is neighbour of current and distance(next) == -1 {
        distance(next) = distance(current) + 1
        enqueue next → Q
    }
}

Rules:
• Numbers represent elevation
• You may only traverse to adjacent grid cells that differ by no more than 2

Sample Map:

Distance:

Q: (0,0)(0,1)(1,0)(1,1)(2,0)(1,2)(2,1)(3,0)(2,2)(4,0)(2,3)(4,1)(5,0)(3,3)(5,1) ...

Dist: 0 1 1 2 2 3 3 3 4 4 5 5 5 5 6 6
Queue Implementation

Queue Interface:

- a sequence of data in FIFO order
- create()
- enqueue(x)
- dequeue()
- isEmpty()

Implement using a Linked List

- create() and isEmpty() are trivial
- for enqueue(x) and dequeue(), only issue is to decide which end of the list
Queue Implementation: Algorithms

create():
    return LLcreate();

isEmpty(Q):
    return (Q->head == NULL);

enqueue(Q, x):
    LLappend(Q, x);

decqueue(Q):
    return LLremoveHead(Q);
typedef LL_t queue_t;

// Creates a pointer to a new empty queue.
// Returns NULL on failure.
queue_t * queue_create(void);

// Recycles a queue
void queue_destroy(queue_t * q);

// Returns 1 iff queue is empty
int queue_isEmpty(queue_t * q);

// Adds element to the back of the queue
void queue_enqueue(queue_t * q, int element);

// Removes element from the front of the queue.
// Undetermined behaviour if queue is empty
int queue_dequeue(queue_t * q);

Information Hiding in C

An invitation for disaster!

Encourages abuse or misuse by calling the linked list functions on the type queue_t *.

Better would be:

typedef struct _queue
queue_t;

which hides all information.

The naming implies that we would or should call these operations only on the type queue_t *. 
Marrying Data and Functions

Encapsulation

- bundle related data and operations together

Forge a language construct that marries data and operations together

- use a struct!
- make the functions part of the data type explicitly
  - called methods
- similar idea to an object in Python

Adds another level of protection against misuse
typedef struct _queue {
    LL_t * intlist;
} queue_t;

queue_t * queue_create(void);
void queue_destroy(queue_t * q);
int queue_isEmpty(queue_t * q);
void queue_enqueue(queue_t * q, int element);
int queue_dequeue(queue_t * q);

typedef struct _queue {
    LL_t * intlist;
    void queue_destroy(queue_t * q);
    int queue_isEmpty(queue_t * q);
    void queue_enqueue(queue_t * q, int element);
    int queue_dequeue(queue_t * q);
} queue_t;

queue_t * queue_create(void);
typedef struct _queue {
    LL_t * intlist;
} queue_t;

queue_t * queue_create(void);

void queue_destroy(queue_t * q);
int queue_isEmpty(queue_t * q);
void queue_enqueue(queue_t * q, int element);
int queue_dequeue(queue_t * q);

typedef struct _queue {
    LL_t * intlist;
    void queue_destroy(struct _queue * q);
    int queue_isEmpty(struct _queue * q);
    void queue_enqueue(struct _queue * q, int element);
    int queue_dequeue(struct _queue * q);
} queue_t;

queue_t * queue_create(void);
typedef struct _queue {
    LL_t * intlist;
} queue_t;

queue_t * queue_create(void);
void queue_destroy(queue_t * q);
int queue_isEmpty(queue_t * q);
void queue_enqueue(queue_t * q, int element);
int queue_dequeue(queue_t * q);

typedef struct _queue {
    LL_t * intlist;
    void (* destroy)(struct _queue * q);
    int (* isEmpty)(struct _queue * q);
    void (* enqueue)(struct _queue * q, int element);
    int (* dequeue)(struct _queue * q);
} queue_t;

queue_t * queue_create(void);

Pointer to a function rather than the function itself

Caller's notation:
Q->enqueue(Q, x);
A Look Ahead to C++

Motivated by these interface issues, C++ evolved out of C.

- formulated by Bjarne Stroustrup in 1978

Provides the syntactic sugar for:

- information hiding
- encapsulation of data and methods
- common code re-use situations

Migrate from `struct` → `class`