BGN: 2191A

P4

Q5

a.

```
#include <memory.h>

#define INT 0
#define DOUBLE 1
struct fifo_entry {
    unsigned int unionType;
    union {
        int i;
        double d;
    } value;
    struct fifo_entry *next;
};

void enqueue_int(struct fifo_entry **head_ptr, struct
fifo_entry **tail_ptr, int val) {
    struct fifo_entry *node = (struct fifo_entry
*)malloc(sizeof(struct fifo_entry));
    node->unionType = INT;
    node->value.i = val;
    node->next = 0;
    if (*head_ptr == 0) {
        *head_ptr = node;
    } else {
        (*head_ptr)->next = node;
    }
    *tail_ptr = node;
    return;
}
```

b.

```
#define INT 0
#define DOUBLE 1
struct fifo_entry {
    unsigned int unionType;
    union {
        int i;
        double d;
    } value;
    struct fifo_entry *next;
};

struct fifo_entry *dequeue(struct fifo_entry **head_ptr,
struct fifo_entry **tail_ptr, struct fifo_entry
**aux_head_ptr, struct fifo_entry **aux_tail_ptr) {
    if (*head_ptr == *tail_ptr) {
        return NULL;
    }
    struct fifo_entry *node = *head_ptr;
    *head_ptr = (*head_ptr)->next;

    node->next = 0;
    if (*aux_head_ptr == 0) {
        *aux_head_ptr = node;
        *aux_tail_ptr = node;
    } else {
        (*aux_tail_ptr)->next = node;
        *aux_tail_ptr = node;
    }

    return node;
}

void enqueue_int(struct fifo_entry **head_ptr, struct
fifo_entry **tail_ptr, struct fifo_entry **aux_head_ptr,
struct fifo_entry **aux_tail_ptr, int val) {
    struct fifo_entry *node;
    if (*aux_head_ptr == 0) {
        node = (struct fifo_entry *)malloc(sizeof(struct
fifo_entry));
    } else {
        node = *aux_head_ptr;
        *aux_head_ptr = (*aux_head_ptr)->next;
    }
    node->unionType = INT;
    node->value.i = val;
    node->next = 0;
    if (*head_ptr == 0) {
        *head_ptr = node;
        *tail_ptr = node;
    } else {
        (*tail_ptr)->next = node;
        *tail_ptr = node;
    }
    return;
}
```

An alternative to this approach would be storing the FIFO entries as a contiguous array list rather than as a linked list. This would allow for the possibility of fast lookups at positions other than the head and tail, and also for efficient computation of the length of the FIFO.

Another alternative would be deallocating nodes as soon as they are popped off of the FIFO. This would result in less memory being used up altogether. This would be appropriate if it is unlikely that lots of new nodes will be created after lots have already been removed.

c.

```
class FIFO {
private:
    fifo_entry *head_ptr;
    fifo_entry *tail_ptr;
    fifo_entry *aux_head_ptr;
    fifo_entry *aux_tail_ptr;

    FIFO(void); // NOTE: this is only private because the question says "It should not be possible to create an instance of class FIFO". I am taking this to mean that FIFO will be extended with factory methods, or will have friend classes to create it.
```

```
public:
    ~FIFO(void);
    void enqueue_int(int);
    void enqueue_double(double);
    bool isempty(void);
    void dequeue(void do_I(int), void do_D(double));
};

FIFO::FIFO(void): head_ptr(nullptr), tail_ptr(nullptr),
aux_head_ptr(nullptr), aux_tail_ptr(nullptr) {}

FIFO::~FIFO(void) {
    if (this->head_ptr != nullptr) {
        fifo_entry *current = this->head_ptr;
        while (current != this->tail_ptr) {
            fifo_entry *next = current->next;
            delete current;
            current = next;
        }
        delete current;
    }

    if (this->aux_head_ptr != nullptr) {
        fifo_entry *current = this->aux_head_ptr;
        while (current != this->aux_tail_ptr) {
            fifo_entry *next = current->next;
            delete current;
            current = next;
        }
        delete current;
    }
}

void FIFO::enqueue_int(int x) {
    fifo_entry *node;
    if (this->aux_head_ptr == nullptr) {
        node = new fifo_entry {INT, x, nullptr};
    } else {
        node = this->aux_head_ptr;
        node->unionType = INT;
        node->value.i = x;
        node->next = nullptr;
        this->aux_head_ptr = this->aux_head_ptr->next;
    }

    if (this->head_ptr == nullptr) {
        this->head_ptr = node;
        this->tail_ptr = node;
    } else {
        this->tail_ptr->next = node;
        this->tail_ptr = node;
    }
}

bool FIFO::isempty() {
    return this->head_ptr == nullptr;
}

void FIFO::dequeue(void do_I(int), void do_D(double)) {
    if (this->isempty()) {
        // Perhaps throw an error here
        return;
    }

    fifo_entry *node = this->head_ptr;
    node->next = nullptr;
    this->head_ptr = this->head_ptr->next;
    if (this->aux_head_ptr == nullptr) {
        this->aux_head_ptr = node;
        this->aux_tail_ptr = node;
    } else {
        this->aux_tail_ptr->next = node;
        this->aux_tail_ptr = node;
    }

    switch (node->unionType) {
        case INT:
            return do_I(node->value.i);
        case DOUBLE:
            return do_D(node->value.d);
        default:
            // Perhaps throw an error here
            return;
    }
}
```

d. Java object types which are supertypes of Integer (including Integer itself), are valid for use in Gen<X>. These include Integer, Double, Float, BigInteger. Note: primitive types such as int, float, double, are not valid.

   Any type which is a supertype of char is valid in Tem<X>. E.g., int, float, double, char. Also, their unsigned equivalents are allowed. X can also be a pointer. Furthermore, X can be any class with an assignment operator from any of these classes defined will be allowed.