04-630
Data Structures and Algorithms for Engineers

Lecture 9: Stack and Queue ADTs

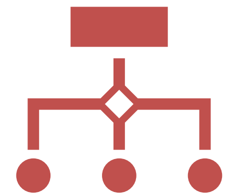
Agenda

Stack ADT

Implementation using List ADT (array and linked-list)

Comparison of order of complexity

Stack applications



Queue ADT

Implementation using List ADT (array and linked-list)

Comparison of order of complexity

Dedicated ADT

Circular queues

Queue applications

Stack ADT

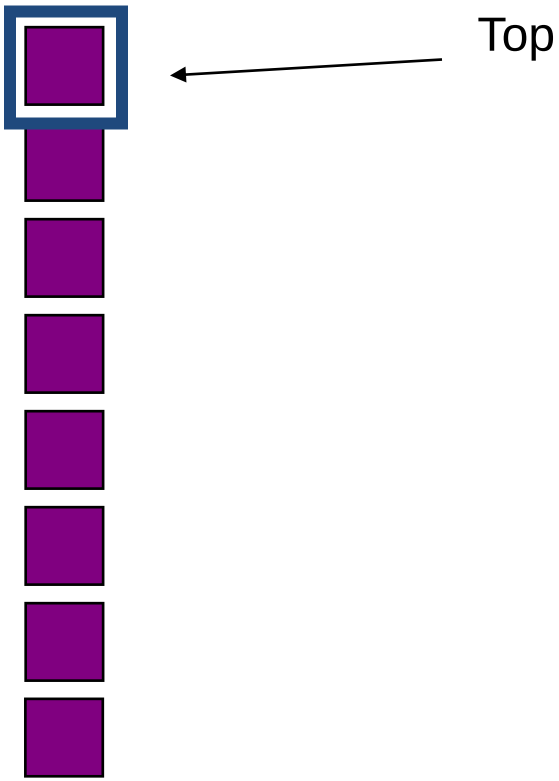
- Implementation using List ADT (array and linked-list)
- Comparison of order of complexity
- Stack applications

Stacks

A stack is a special type of list

- all insertions and deletions take place at one end, called the top
- thus, the last one added is always the first one available for deletion
- also referred to as
  - pushdown stack
  - pushdown list
  - LIFO list (Last In First Out)

Stacks



Stack Operations

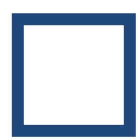
Declare: $\rightarrow$ S :

The function value of Declare(S) is an empty stack

Stack Operations

Empty: $\rightarrow$ S :

The function Empty causes the stack to be emptied and it returns position End(S)



Stack Operations

IsEmpty: **S** $\rightarrow$ **B** :

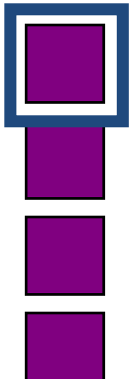
The function value IsEmpty(S) is true if *S* is empty; otherwise it is false

Stack Operations

Top: S $\rightarrow$ E :

The function value Top(S) is the first element in the list;

if the list is empty, the value is undefined

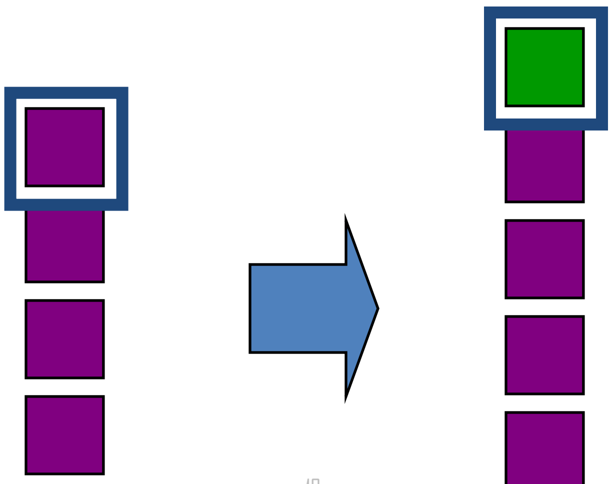


Stack Operations

Push: **E** x **S** $\rightarrow$ **S** :

Push(e, S)

Insert an element e at the top of the stack

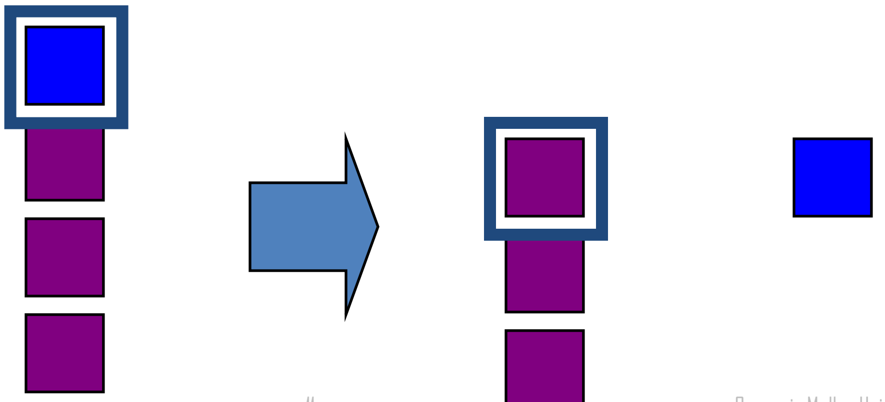


Stack Operations

Pop: $S \rightarrow E$:

Pop(S)

Remove the top element from the stack: i.e. return the top element and delete it from the stack



Stack Operations

- All these operations can be directly implemented using the LIST ADT operations on a List S
- Although it may be more efficient to use a dedicated implementation
- It depends what you want: ***code efficiency*** or ***software re-use (i.e. utilization efficiency)***

Stack Operations

Declare(S)

Empty(S)

Top(S)

  Retrieve(First(S), S)

Push(e, S)

  Insert(e, First(S), S)

Pop(S)

  Retrieve(First(S), S)

  Delete(First(S), S)

Stack Errors

- Stack overflow errors occur when you attempt to Push() an element on a stack that is full
- Stack underflow errors occur when you attempt to Pop() an element off of an empty stack
- Your ADT implementation should provide guards that catch these errors

Stack Implementation

- The List ADT can be implemented
  - As an array
  - As a linked-list

- So, therefore, so can the Stack ADT

- What are the relative advantages and disadvantages of the these two options?

- When would you pick one implementation over the other?

Stack Operations

Declare(S)

Empty(S)

Top(S)

Retrieve(First(S), S)

Push(e, S)

Insert(e, First(S), S)

Pop(S)

Retrieve(First(S), S)

Delete(First(S), S)

Stack Operations

	Array	Linked-List
Declare(S)	$O(1)$	$O(1)$
Empty(S)	$O(1)$	$O(n)$
Top(S) Retrieve(First(S), S)	$O(1)$	$O(1)$
Push(e, S) Insert(e, First(S), S)	$O(n)$ ... why?	$O(1)$
Pop(S) Retrieve(First(S), S) Delete(First(S), S)	$O(\mathrm{n})$	$O(1)$

Stack Operations

	Array	Linked-List
Declare(S)	O(1)	O(1)
Empty(S)	O(1)	O(n)
Top(S) Retrieve(Last(S), S)	O(1)	O(1)
Push(e, S) Insert(e, end(S), S)	O(1)	O(n) … !!!
Pop(S) Retrieve(Last(S), S) Delete(Last(S), S)	O(1)	O(n) … !!!

Stack Implementation

- Reusing the List ADT involves some compromises

- Alternative is to create a new Stack ADT

  – With an implementation that avoids these compromises

Stack Applications

- Reversing the order of a list of items
- Undo sequence (like those in a text editor)
- Page-visited history in a web browser
- Saving local variables when one function calls another, and it calls another, and so on
- Parenthesis (begin-end token) matching

Stack Applications

Saving local variables when one function calls another, and it calls another, and so on

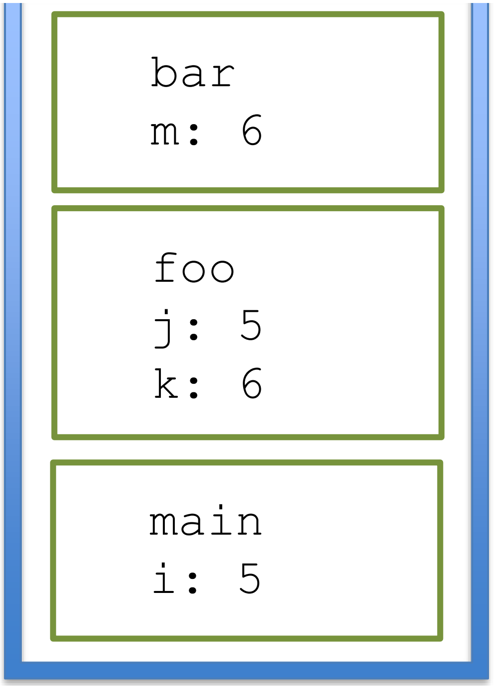
- A typical operating system keeps track of the chain of active functions and local variables with a stack
- When a function is called, the run-time system pushes onto the stack a frame containing local variables and maintains state of program at the point of departure
- When a function returns to the point of departure, the function frame is popped from the stack and control is passed to the code at the point of departure.

Stack Applications

```
int main () {
    int i = 5;
    foo(i);
}

foo(int j) {
    int k;
    k = j+1;
    bar(k);

bar (int m) {
      …
}
```



Stack Applications

Token matching

```
// X is a an array of tokens, e.g. grouping symbol , variable, operator, number

for i=0 to n-1 do {
   if X[i] is an opening grouping symbol {
      S.push(X[i]) }
   else {
      if X[i] is a closing grouping symbol {
         if S.isEmpty() then
            error:: nothing to match with
         if S.pop() is not equal to X[i]
            error:: false {wrong type}
      }
   }
}
if S.isEmpty() then
   return true {every symbol matched}
else
   return false {some symbols were never matched}
```

Stack Applications

Notation of expressions

- Infix notation
- Postfix notation
- Prefix notation

Infix	Postfix	Prefix	Notes
A * B + C / D	A B * C D / +	+ * A B / C D	multiply A and B, divide C by D, add the results
A * (B + C) / D	A B C + * D /	/ * A + B C D	add B and C, multiply by A, divide by D
A * (B + C / D)	A B C D / + *	* A + B / C D	divide C by D, add B, multiply by A

(http://jcsites.juniata.edu/faculty/kruse/cs240/stackapps.htm)

Stack Applications

Evaluation of Postfix Notation Expressions

```
create a new stack
while(input stream is not empty){
   token = getNextToken();
   if(token instanceof operand){
      push(token);
   }
   else if (token instance of operator) {
      op2 = pop();
      op1 = pop();
      result = calc(token, op1, op2);
      push(result);
   }
}
return pop();
```

Stack Applications

Demonstrate with 2 3 4 + * 5 – **(EQ: 2*(3+4)-5)**

The time complexity is O(n) because each operand is scanned once, and each operation is performed once

Stack Applications

Infix transformation to Postfix

- This process also uses a stack
- We have to hold information that's expressed inside parentheses while scanning to find the closing ')'
- We also have to hold information on operations that are of lower precedence on the stack

Stack Applications

Infix transformation to Postfix – Algorithm

1. Create an empty stack and an empty postfix output string/stream

2. Scan the infix input string/stream left to right

3. If the current input token is an operand, append it to the output string

4. If the current input token is an operator, pop off all operators that have equal or higher precedence and append them to the output string; push the operator onto the stack. The order of popping is the order in the output.

5. If the current input token is '(', push it onto the stack

6. If the current input token is ')', pop off all operators and append them to the output string until a '(' is popped; discard the '('.

7. If the end of the input string is found, pop all operators and append them to the output string.

Stack Applications

- Demonstrate with 4*3+8/2*(3+2)->4 3 * 8 2 / 3 2 + * +

Queue ADT

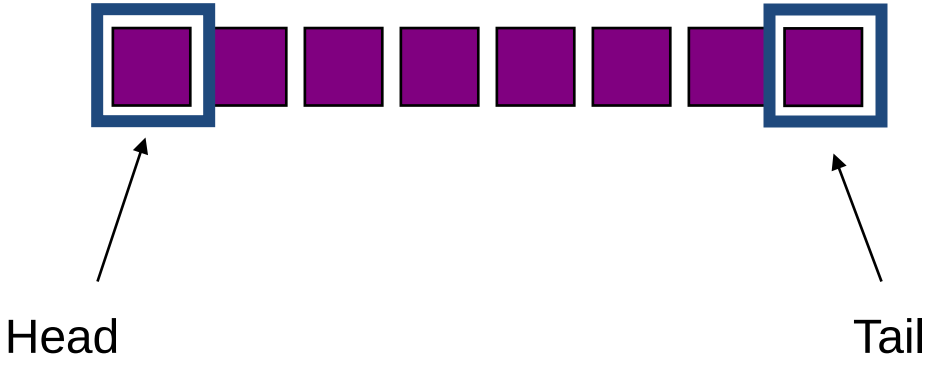
- Implementation using List ADT (array and linked-list)
- Comparison of order of complexity
- Dedicated ADT
- Circular queues
- Queue applications

Queues

A queue is another special type of list

- insertions are made at one end, called the ***tail of the queue***
- deletions take place at the other end, called ***the head***
- thus, the last one added is always the last one available for deletion
- also referred to as
  - FiFO list (First In First Out)

Queues



Queue Operations

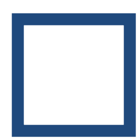
Declare: $\rightarrow$ **Q** :

The function value of Declare(Q) is an empty queue

Queue Operations

Empty: $\rightarrow$ **Q** :

The function Empty causes the queue to be emptied and it returns position End(Q)



Queue Operations

IsEmpty: $\mathbf{Q} \rightarrow \mathbf{B}$:

The function value IsEmpty(Q) is true if *Q* is empty; otherwise it is *false*

Queue Operations

Head: $Q \rightarrow E$:

The function value Head(Q) is the first element in the list;

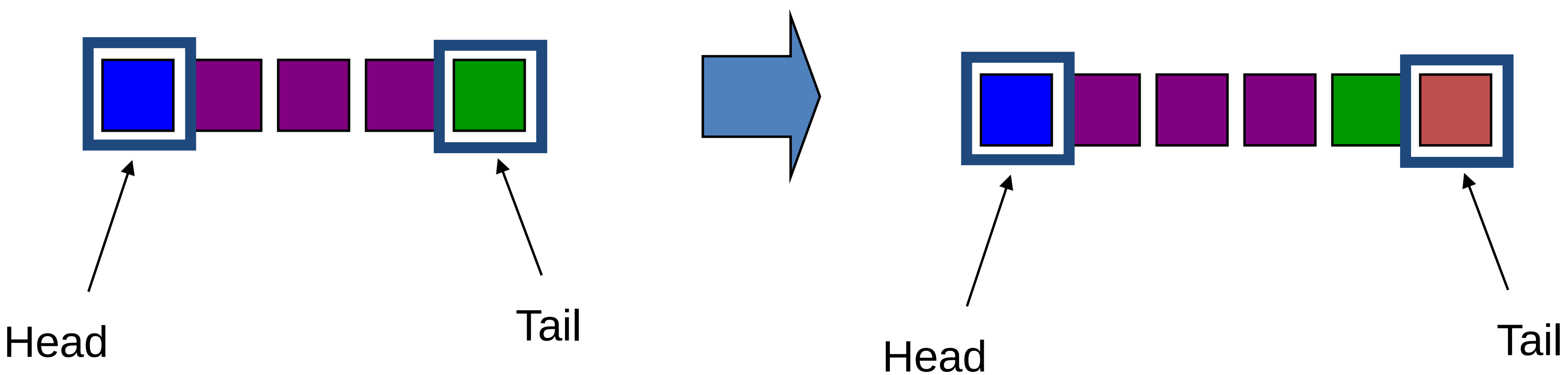
if the queue is empty, the value is undefined

Queue Operations

Enqueue: **E** x **Q** $\rightarrow$ **Q** :

Enqueue(e, Q)

Add an element *e* the the tail of the queue

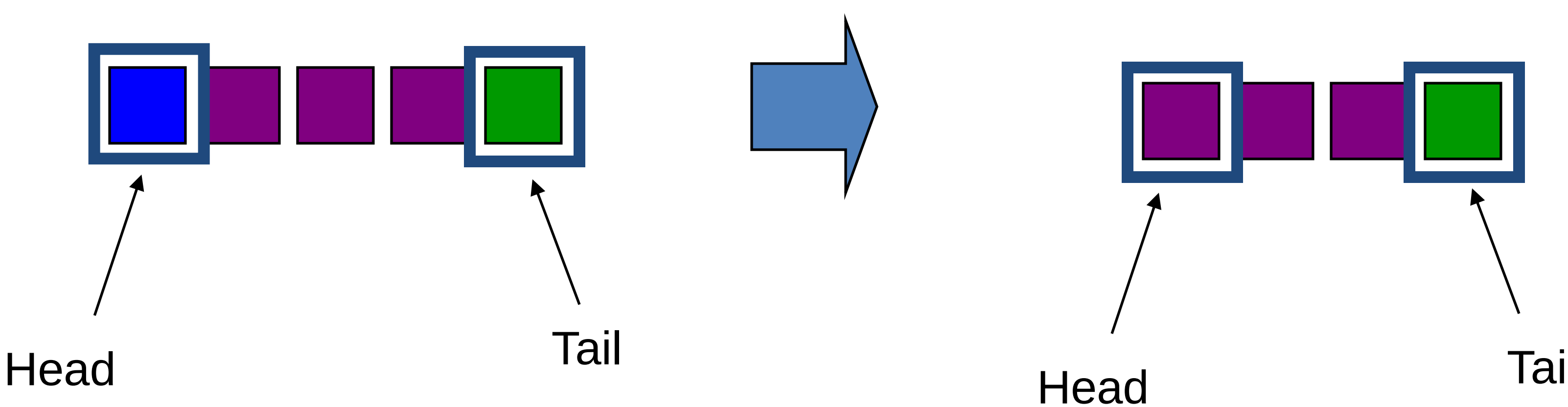


Queue Operations

Dequeue: $Q \rightarrow E$:

Dequeue(Q)

Remove the element from the head of the queue: i.e. return the first element and delete it from the queue



Queue Operations

- All these operations can be directly implemented using the LIST ADT operations on a queue Q
- Again, it may be more efficient to use a dedicated implementation
- And, again, it depends what you want: ***code efficiency or software re-use (i.e. utilization efficiency)***

Queue Operations

Declare(Q)

Empty(Q)

Head(Q)

  Retrieve(First(Q), Q)

Enqueue(e, Q)

  Insert(e, End(Q), Q)

Dequeue(Q)

  Retrieve(First(Q), Q)

  Delete(First(Q), Q)

Queue Errors

- Queue overflow errors occur when you attempt to enqueue() an element in a queue that is full
- Queue underflow errors occur when you attempt to dequeue() an element from an empty queue
- Your ADT implementation should provide guards that catch these errors

Queue Implementation

- The List ADT can be implemented
  - As an array
  - As a linked-list
- So, therefore, so can the Queue ADT
- What are the advantages and disadvantages of the these two options?
- When would you pick one implementation over the other?

Queue Operations

	Array	Linked-List
Declare(Q)	$O(1)$	$O(1)$
Empty(Q)	$O(1)$	$O(n)$
Head(Q) Retrieve(First(Q), Q)	$O(1)$	$O(1)$
Enqueue(e, Q) Insert(e, End(Q), Q)	$O(1)$	$O(n)$ … why?
Dequeque(Q) Retrieve(First(Q), Q) Delete(First(Q), Q)	$O(n)$ … why?	$O(1)$

Queue Implementation

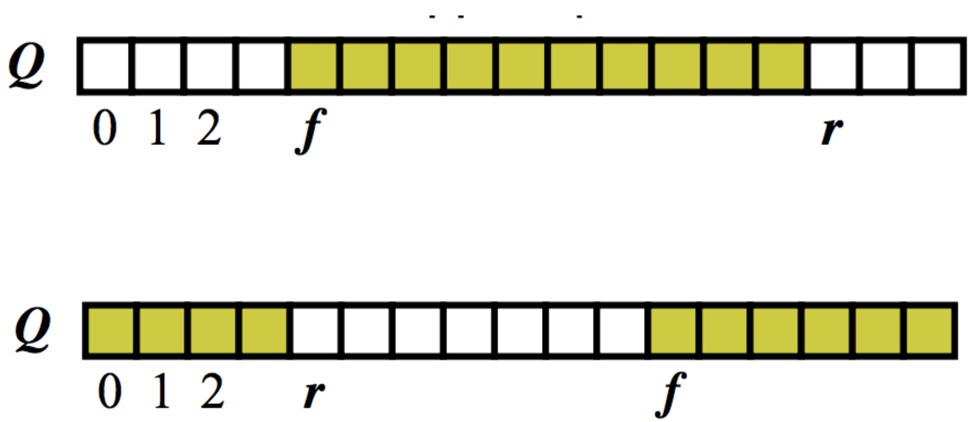
- Reusing the List ADT involves some compromises

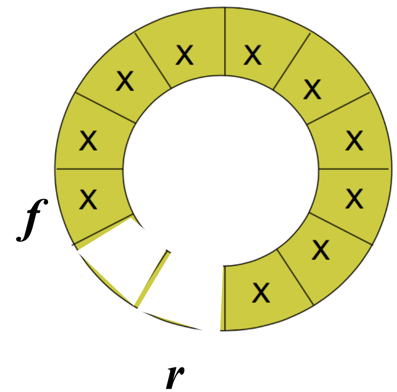
- Alternative is to create a new Queue ADT

  - With an implementation that avoids these compromises

Queue Implementation

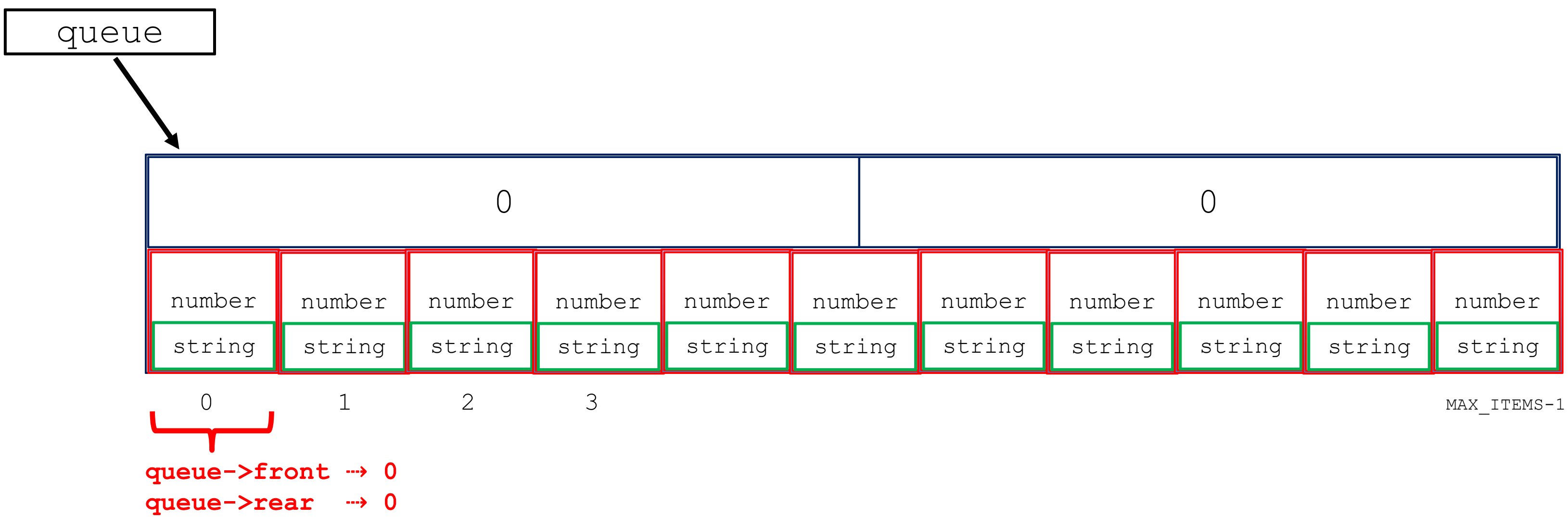
```
typedef struct {
          int front;
          int rear;
          ITEM_TYPE items[MAX_ITEMS];
       } QUEUE_TYPE;
```





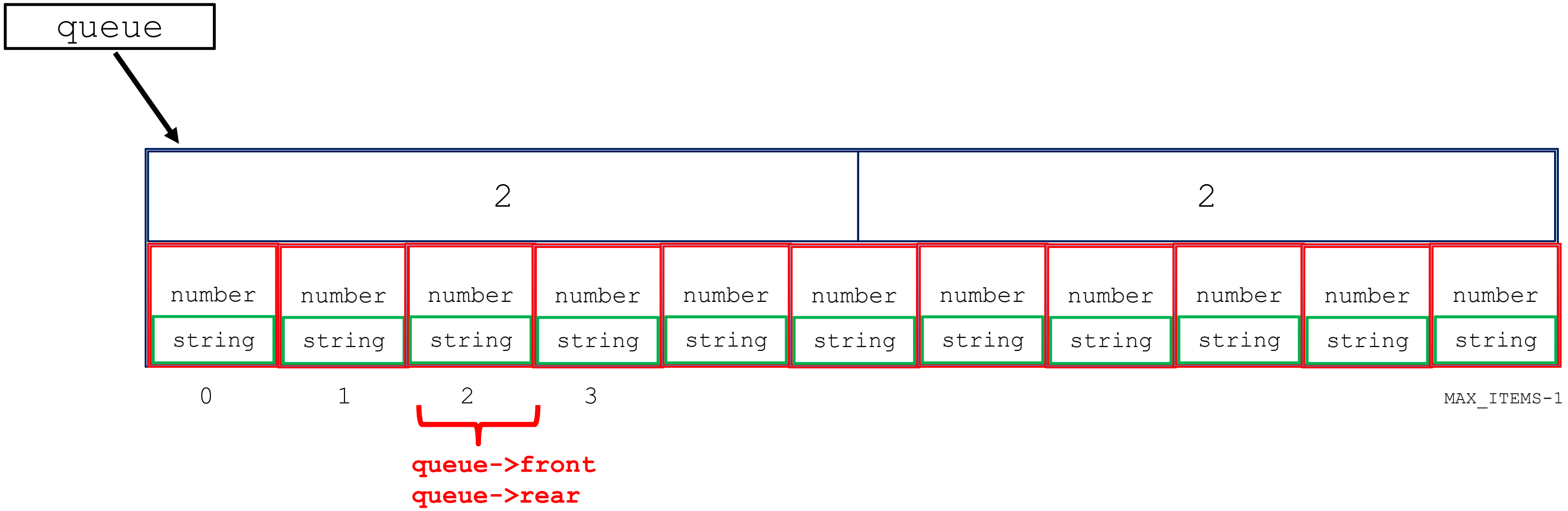
Queue Implementation

```
void empty(QUEUE_TYPE *queue) {
   queue->front = 0;
   queue->rear  = 0;
   return(end(queue));
}
```



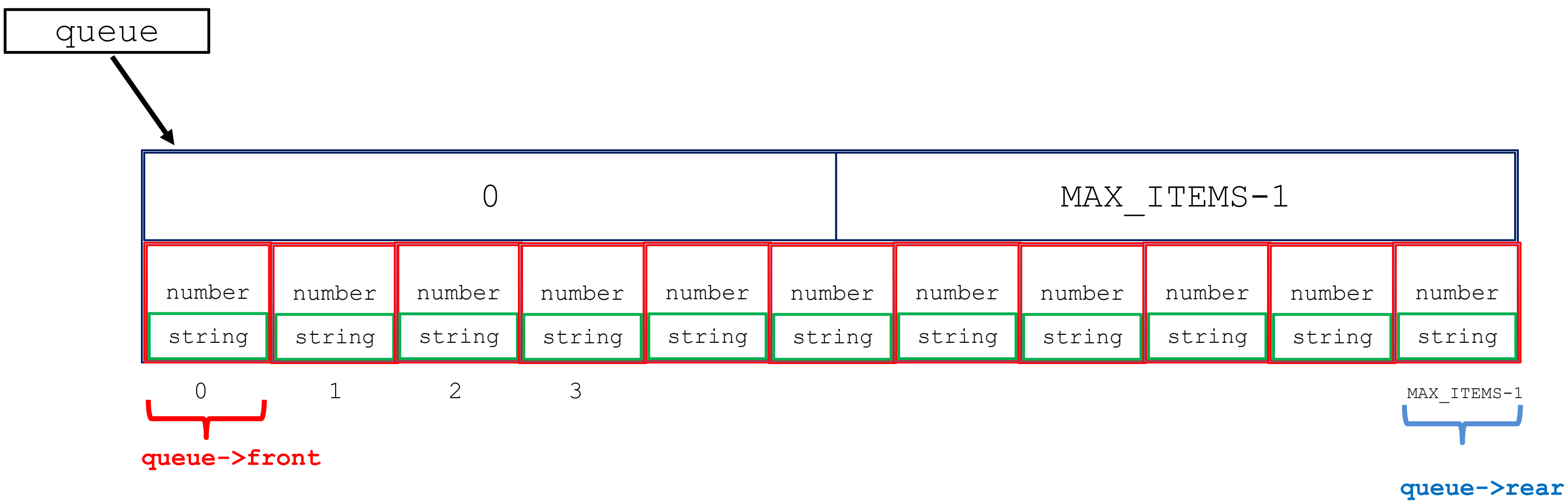
Queue Implementation

```
bool is_empty(QUEUE_TYPE *queue) {
   if (queue->front == queue->rear)
      return(true);
   else
      return(false)
}
```



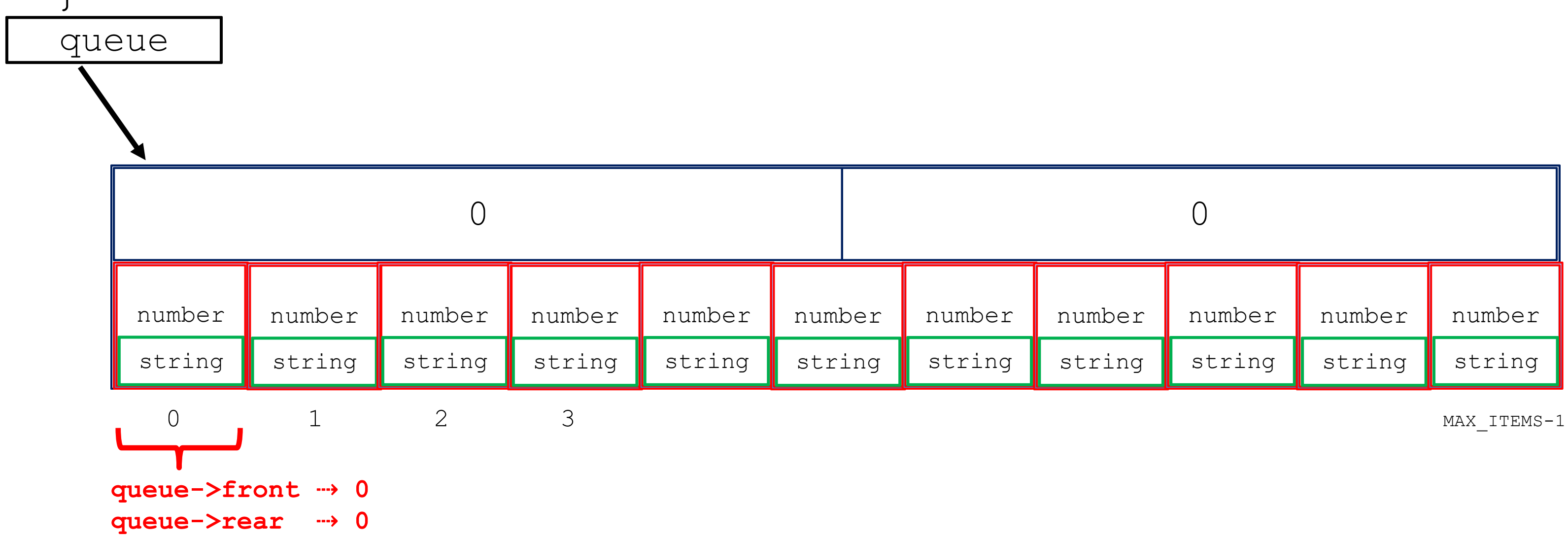
Queue Implementation

```
int is_full(QUEUE_TYPE *queue) {
   if ((queue->rear + 1 ) % MAX_ITEMS == queue->front )
      return(TRUE);
   else
      return(FALSE);
}
```



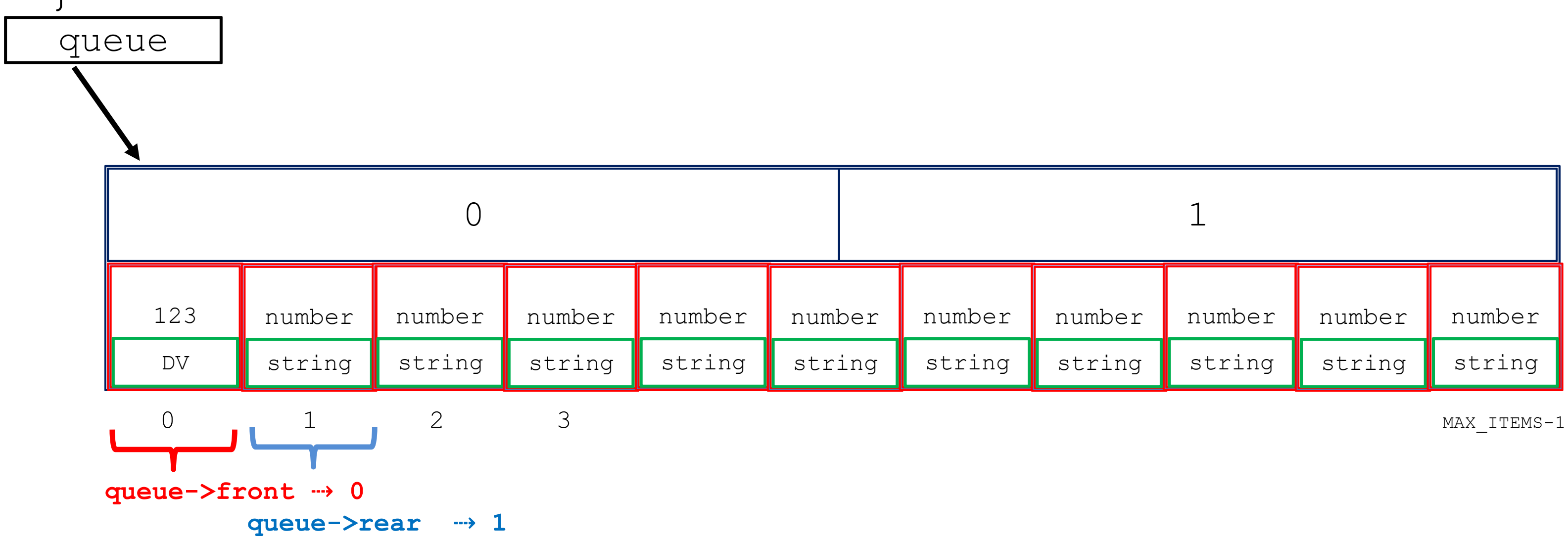
Queue Implementation

```
void enqueue(ITEM_TYPE e, QUEUE_TYPE *queue) {
   if (!is_full(queue)) {
      queue->items[queue->rear] = e;
      queue->rear = (queue->rear +1) % MAX_ITEMS;
   }
   else {
      error("Queue overflow: queue is already full");
   }
}
```



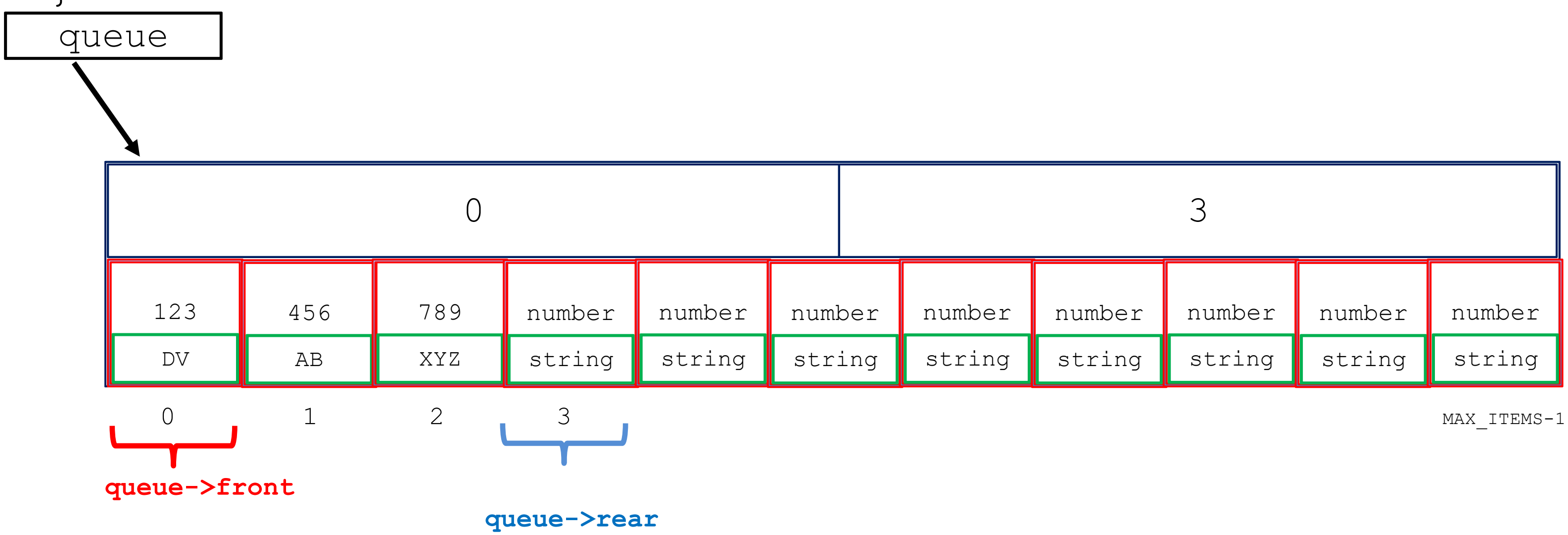
Queue Implementation

```
void enqueue(ITEM_TYPE e, QUEUE_TYPE *queue) {
   if (!is_full(queue)) {
      queue->items[rear] = e;
      queue->rear = (queue->rear +1) % MAX_ITEMS;
   }
   else {
      error(“Queue overflow: queue is already full”);
   }
}
```



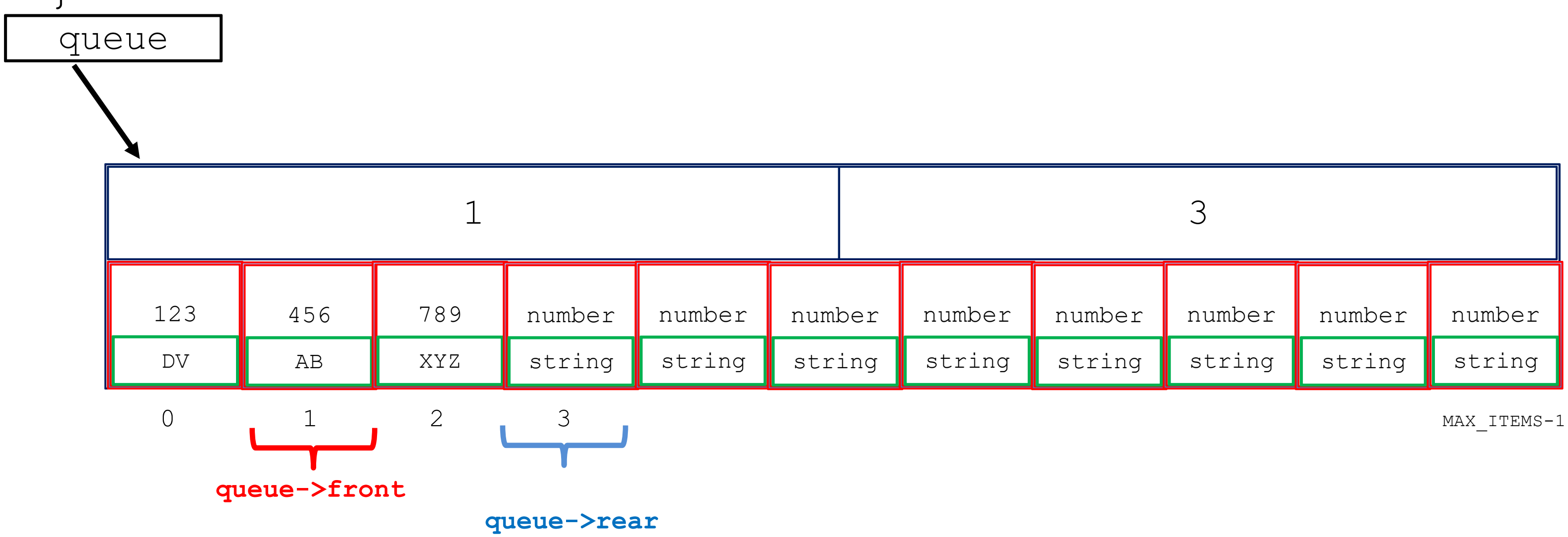
Queue Implementation

```
void dequeue(ITEM_TYPE *e, QUEUE_TYPE *queue) {
   if (!is_empty(queue)) {
      *e = queue->items[queue->front];
      queue->front = (queue->front+1) % MAX_ITEMS;
   }
   else {
      error(“Queue underflow: queue is empty”);
   }
}
```



Queue Implementation

```
void dequeue(ITEM_TYPE *e, QUEUE_TYPE *queue) {
   if (!is_empty(queue)) {
      *e = queue->items[queue->front];
      queue->front = (queue->front+1) % MAX_ITEMS;
   }
   else {
      error(“Queue underflow: queue is empty”);
   }
}
```



Queue Implementation

- Can you see a particular problem with the linked-list implementation?

- How would you fix it?

Queue Operations

	Array	Linked-List
Declare(Q)	O(1)	O(1)
Empty(Q)	O(1)	O(n)
Head(Q) Retrieve(First(Q), Q)	O(1)	O(1)
Enqueue(e, Q) Insert(e, End(Q), Q)	O(1)	O(n) … why?
Dequeque(Q) Retrieve(First(Q), Q) Delete(First(Q), Q)	O(n) … why?	O(1)

Queue Implementation

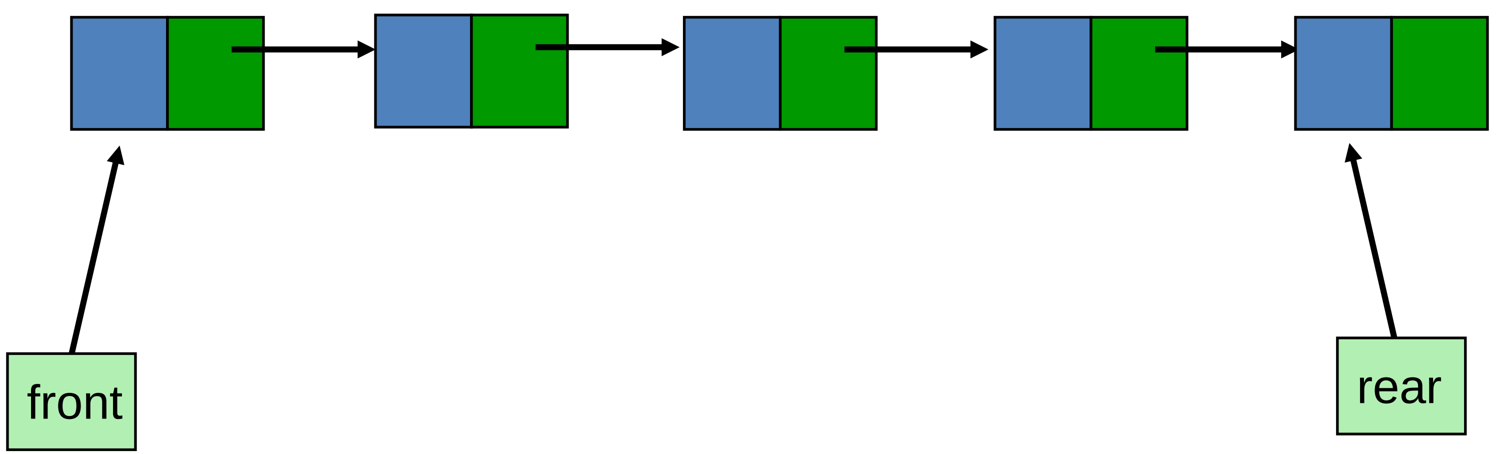
- Can you see a particular problem with the linked-list implementation?

- How would you fix it?

Queue Implementation

- Can you see a particular problem with the linked-list implementation?

- How would you fix it?



Queue Applications

- Scheduling/waiting for system service queues
- Resource queues – provide coordinated access to shared resources
- Message queues (Buffer)
- Multi-queues and priority queues

Acknowledgement

- Adopted and Adapted from Material by:
- David Vernon: vernon@cmu.edu ; www.vernon.eu