Data structures and Algorithms

Lecture 2b: stacks and queues

[GT 2.1]

Dr. Julian Mestre

School of Computer Science

THE UNIVERSITY OF SYDNEY

Stacks and queues

These ADTs are restricted forms of List, where insertion and removal happen only in particular locations:

- stacks follow last-in-first-out (LIFO)
- queues follows first-in-first-out (FIFO)

So why should we care are a less general ADT?

- operations names are part of computing culture
- numerous applications
- simpler/more efficient implementations than Lists

Stack ADT

Main stack operations:

– push(e): inserts an element, e
– pop(): removes and returns the last inserted element

Auxiliary stack operations:

– top(): returns the last inserted element without removing it
– size(): returns the number of elements stored
– isEmpty(): indicates whether no elements are stored

Stack Example

operation	returns	stack
push(5)	-	[5]
push(3)	-	[5, 3]
size()	2	[5, 3]
pop()	3	[5]
isEmpty()	False	[5]
pop()	5	[]
isEmpty()	True	[]
push(7)	-	[7]
push(9)	-	[7, 9]
top()	9	[7, 9]
push(4)	-	[7, 9, 4]
pop()	4	[7, 9]

Stack Applications

Direct applications

- Keep track of a history that allows undoing such as Web browser history or undo sequence in a text editor
- Chain of method calls in a language supporting recursion
- Context-free grammars

Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

Method Stacks

The runtime environment keeps track of the chain of active methods with a stack, thus allowing recursion

When a method is called, the system pushes on the stack a frame containing

- Local variables and return value
- Program counter

When a method ends, we pop its frame and pass control to the method on top

```
main() {
  int i = 5;
  foo(i);
}
foo(int j) {
  int k;
  k = j+1;
  bar(k);
}
bar(int m) {
  …
}
```

bar
PC = 1
m = 6

foo
PC = 3
j = 5
k = 6

main
PC = 2
i = 5

Parentheses Matching

Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”

- correct: ( )(( )){([( )])}
- correct: ((( )(( )){([( )]))}
- incorrect:)(( )){([( )])}
- incorrect: ({[ ])}
- incorrect: (

Scan input string from left to right:

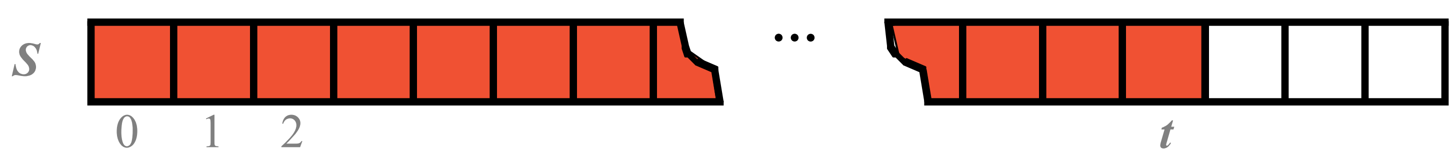
- If we see an opening character, push it to a stack
- If we see a closing character, pop character on stack and check that they match

Stack implementation based on arrays

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

Algorithm *size*()
  return $t + 1$

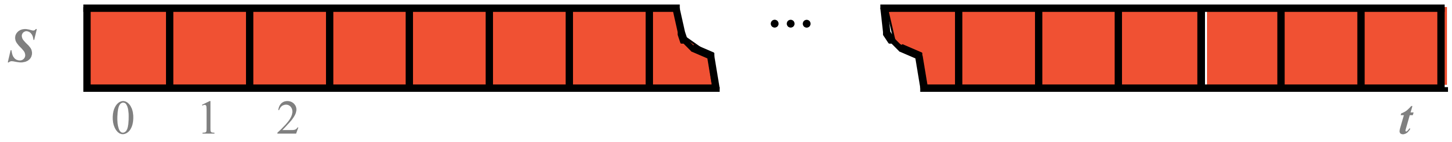
Algorithm *pop*()
  if *isEmpty*() **then**
    return null
  else
    $t \leftarrow t - 1$
    return $S[t + 1]$



Stack implementation based on arrays

- The array storing the stack elements may become full.
- A push operation will then either grow the array or signal an error.

Algorithm *push*(*o*)
  if $t = S.length - 1$ **then**
    signal *stack overflow error*
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$



Stack implementation based on arrays

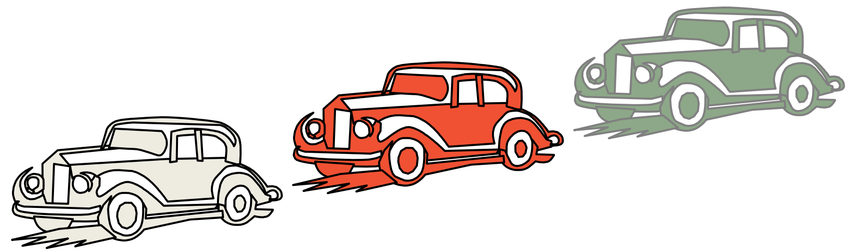
Performance

- Let n be the number of elements in the stack
- The space used is $O(n)$
- Each operation runs in time $O(1)$

Qualifications

- Trying to push a new element into a full stack causes an implementation-specific exception or
- Pushing an item on a full stack causes the underlying array to double in size, which implies each operation runs in $O(1)$ amortized time.

Queue ADT



Main queue operations:

- enqueue(e): inserts an element, e, at the end of the queue
- dequeue(): removes and returns element at the front of the queue

Auxiliary queue operations:

- first(): returns the element at the front without removing it
- size(): returns the number of elements stored
- isEmpty(): indicates whether no elements are stored

Boundary cases:

- Attempting the execution of dequeue or first on an empty queue signals an error or returns null

Queue Example

Operation		*Output* Q
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
first()	*7*	(7)
dequeue()	*7*	()
dequeue()	*null*	()
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	*9*	(7, 3, 5)

Queue applications

Buffering packets in streams, e.g., video or audio

Direct applications

- Waiting lists, bureaucracy
- Access to shared resources (e.g., printer)
- Multiprogramming

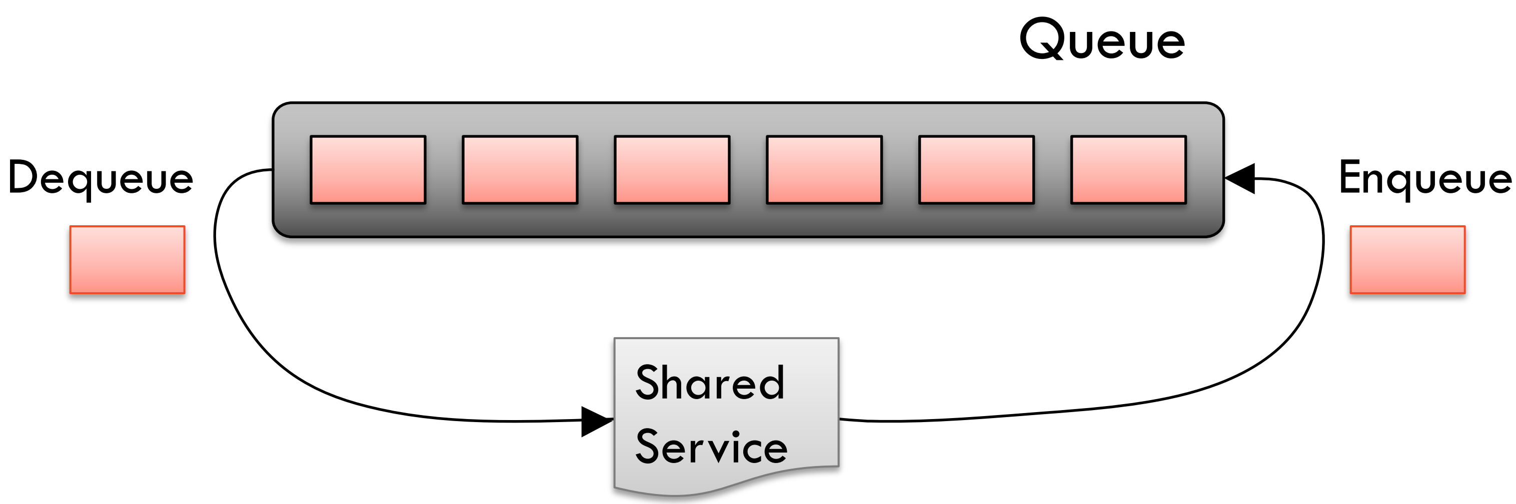
Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

Queue application: Round Robin Schedulers

Implement a round robin scheduler using a queue Q by repeatedly performing the following steps:

1. e = Q.dequeue()
2. Service element e
3. Q.enqueue(e)



Queue implementation based on arrays

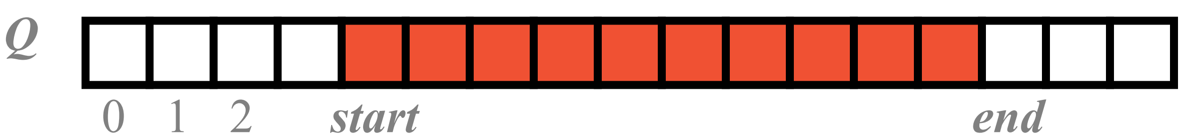
Use an array of size N in a circular fashion

Two variables keep track of the front and size

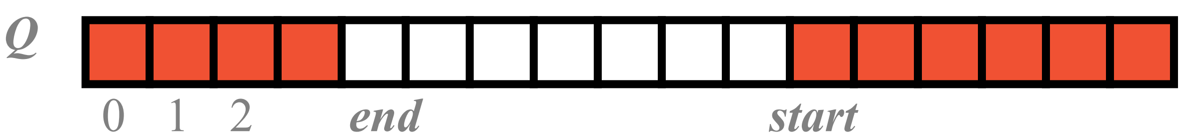
- start : index of the front element
- end : index past the last element
- size : number of stored elements

These are related as follows end = (start + size) mod N, so we only need two

normal configuration



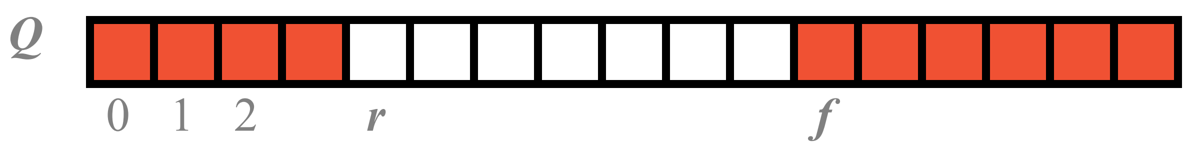
wrapped-around configuration



Queue Operations: Enqueue

Return an error if the array is full. Alternatively, we could grow the underlying array as dynamic arrays do

```
def enqueue(o)
  if size() = N – 1 then
    return "queue full"
  else
    last ← (first + size) mod N
    Q[r] ← o
    size ← (size + 1)
```

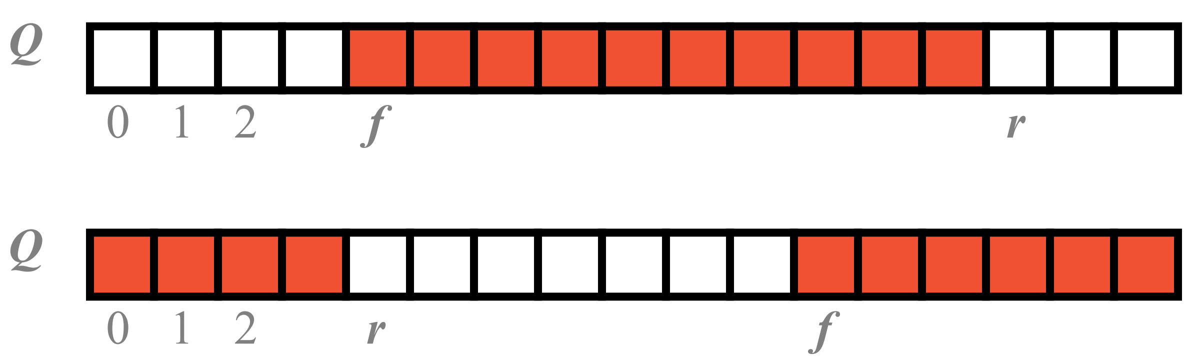


Queue Operations: Dequeue

Note that operation dequeue returns error if the queue is empty

One could alternatively signal an error

```
def dequeue()
  if isEmpty() then
    return "queue empty"
  else
    o ← Q[f]
    f ← (f + 1) mod N
    s ← (s - 1)
    return o
```



Double-ended queues: Deques

– A linear structure that allows insertions and deletions at both ends

Method	Time
size, isEmpty	$O(1)$
getFirst, getLast	$O(1)$
addFirst, addLast	$O(1)$
removeFirst, removeLast	$O(1)$

Table 5.4: Performance of a deque realized by a doubly linked list.

Double-ended queue operations

The deque abstract data type is richer than both the stack and the queue ADTs. The fundamental methods of the deque ADT are as follows:

- addFirst(e): Insert a new element e at the head of the deque.
- addLast(e): Insert a new element e at the tail of the deque.
- removeFirst(): Remove and return the first element of the deque; an error occurs if the deque is empty.
- removeLast(): Remove and return the last element of the deque; an error occurs if the deque is empty.

Additionally, the deque ADT may also include the following support methods:

- getFirst(): Return the first element of the deque; an error occurs if the deque is empty.
- getLast(): Return the last element of the deque; an error occurs if the deque is empty.
- size(): Return the number of elements of the deque.
- isEmpty(): Determine if the deque is empty.