Algorithms
(Stacks, Queues, and Deques)

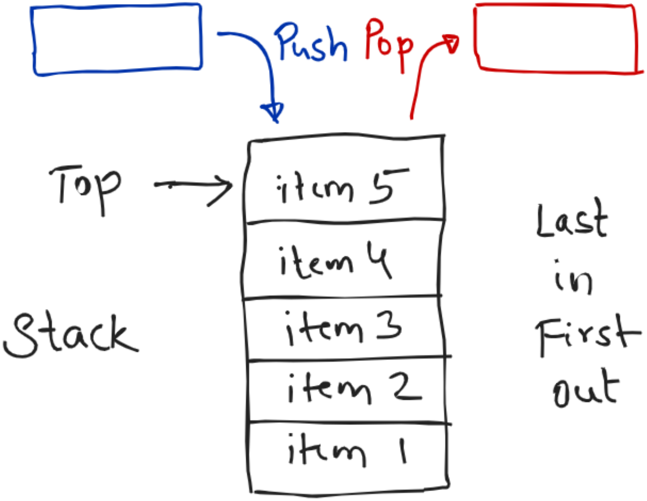
Pramod Ganapathi
Department of Computer Science
State University of New York at Stony Brook

January 3, 2021

Contents

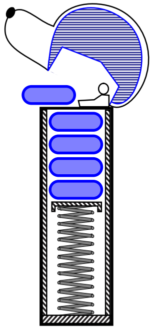
Stacks

Stacks


Push
Pop
Top
item 5
item 4
item 3
item 2
item 1
Stack
Last
in
First
out

Stacks

- A stack is a one-ended linear data structure.
- A stack uses the last in, first out principle.
- A stack has two major operations: push and pop, meaning insert and delete respectively.



A candy dispenser

Applications of stacks

- Function calls in computer programs
  Recursion trace = stack trace
- Undo mechanism in all text editors
- Back button in all web browsers
- Evaluation of arithmetic expressions

Stack ADT

Method	Functionality
`push(e)`	Adds element e to the top of the stack.
`pop()`	Removes and returns the top element from the stack (or null if the stack is empty).
`top()`	Returns the top element of the stack, without removing it (or null if the stack is empty).
`size()`	Returns the number of elements in the stack.
`isEmpty()`	Returns a boolean indicating whether the stack is empty

Operations on a stack

Method	Return value	Stack contents
push(5)	-	(5)
push(3)	-	(5, 3)
size()	2	(5, 3)
pop()	3	(5)
isEmpty()	false	(5)
pop()	5	()
isEmpty()	true	()
pop()	null	()
push(7)	-	(7)
push(9)	-	(7, 9)
top()	9	(7, 9)
push(4)	-	(7, 9, 4)
size()	3	(7, 9, 4)
pop()	4	(7, 9)
push(6)	-	(7, 9, 6)
push(8)	-	(7, 9, 6, 8)
pop()	8	(7, 9, 6)

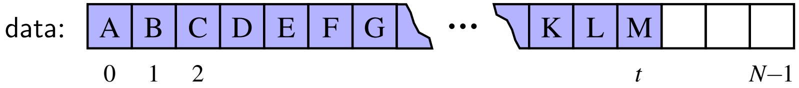
java.util.Stack class

Our Stack ADT	Class java.util.Stack
size()	size()
isEmpty()	empty()
push(e)	push(e)
pop()	pop()
top()	peek()

Stack ADT interface

```java
public interface Stack<E> {

  // Returns the number of elements in the stack.
  int size();

  // Tests whether the stack is empty.
  boolean isEmpty();

  // Inserts an element at the top of the stack.
  void push(E e);

  // Returns, but does not remove, the element at the top of the stack.
  E top();

  // Removes and returns the top element from the stack.
  E pop();
}
```

Stack implemented using array



- Top of the stack $= data[t]$
- Total number of stack elements $= t + 1$

Stack implemented using array

```
public class ArrayStack<E> implements Stack<E> {
  public static final int CAPACITY=1000; // default array capacity
  private E[] data;                      // generic array used for storage
  private int t = -1;                    // index of the top element in stack
  public ArrayStack() { this(CAPACITY); }
  public ArrayStack(int capacity) {      // constructs stack with given capacity
    data = (E[]) new Object[capacity];   // safe cast; compiler may give warning
  }
  public int size() { return (t + 1); }
  public boolean isEmpty() { return (t == -1); }
  public String toString() {...}

  public E top() {...}
  public void push(E e) throws IllegalStateException {...}
  public E pop() {...}

  public static void main(String[] args) {...}
}
```

Stack implemented using array

```java
/** Demonstrates sample usage of a stack. */
public static void main(String[] args) {
  Stack<Integer> S = new ArrayStack<>();  // contents: ()
  S.push(5);                              // contents: (5)
  S.push(3);                              // contents: (5, 3)
  System.out.println(S.size());           // contents: (5, 3)     outputs 2
  System.out.println(S.pop());            // contents: (5)        outputs 3
  System.out.println(S.isEmpty());        // contents: (5)        outputs false
  System.out.println(S.pop());            // contents: ()         outputs 5
  System.out.println(S.isEmpty());        // contents: ()         outputs true
  System.out.println(S.pop());            // contents: ()         outputs null
  S.push(7);                              // contents: (7)
  S.push(9);                              // contents: (7, 9)
  System.out.println(S.top());            // contents: (7, 9)     outputs 9
  S.push(4);                              // contents: (7, 9, 4)
  System.out.println(S.size());           // contents: (7, 9, 4)  outputs 3
  System.out.println(S.pop());            // contents: (7, 9)     outputs 4
  S.push(6);                              // contents: (7, 9, 6)
  S.push(8);                              // contents: (7, 9, 6, 8)
  System.out.println(S.pop());            // contents: (7, 9, 6)  outputs 8
}
```

Stack implemented using array

```java
/** Produces a string representation of the contents of the stack.
(ordered from top to bottom). This exists for debugging purposes only. */
public String toString() {
  StringBuilder sb = new StringBuilder("(");
  for (int j = t; j >= 0; j--) {
    sb.append(data[j]);
    if (j > 0)
      sb.append(", ");
  }
  sb.append(")");
  return sb.toString();
}
```

```java
/** Returns, but does not remove, the element at the top of the stack. */
public E top() {
  if (isEmpty())
    return null;
  return data[t];
}
```

Stack implemented using array

```java
/** Inserts an element at the top of the stack. */
public void push(E e) throws IllegalStateException {
  if (size() == data.length)
    throw new IllegalStateException("Stack is full");
  data[++t] = e;          // increment t before storing new item
}
```

```java
/** Removes and returns the top element from the stack. */
public E pop() {
  if (isEmpty())
    return null;
  E answer = data[t];
  data[t] = null;        // dereference to help garbage collection
  t--;
  return answer;
}
```

Stack implemented using array: Limitation

Problem.

- Stack array is of fixed size.
- If capacity is high, a lot of memory is wasted.
  If capacity is low, the program throws exception when there is overflow.

Solutions.

1. Dynamic array for stack.
2. Singly linked list for stack.

Stack implemented using array: Complexity

Method	Running time
size	$\mathcal{O}(1)$
isEmpty	$\mathcal{O}(1)$
top	$\mathcal{O}(1)$
push	$\mathcal{O}(1)$
pop	$\mathcal{O}(1)$

Stack implemented using linked list

- Which of these should we use: SLL, CLL, DLL?
  SLL is the best choice. Why?
- Where should be the head of SLL: stack bottom or stack top?
  SLL head being stack top is the best choice. Why?

Stack implemented using SLL

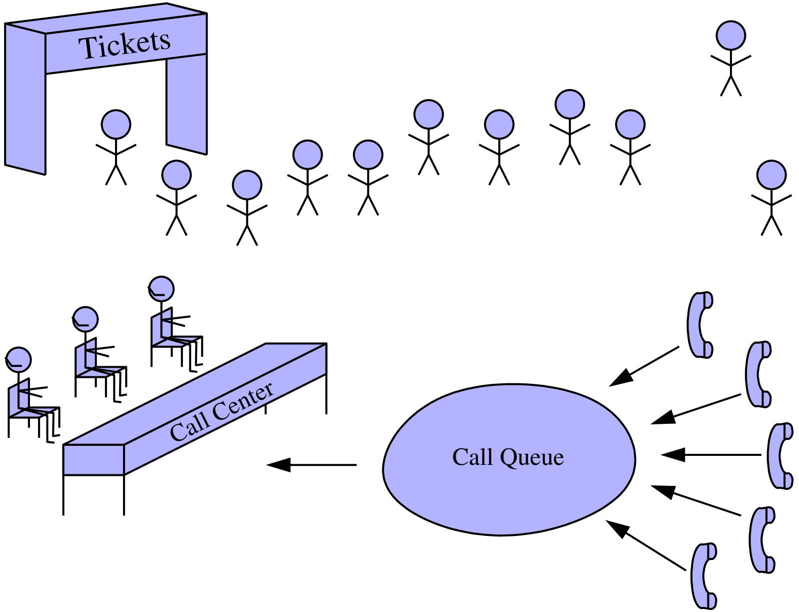
Stack method	SLL method
size()	list.size()
isEmpty()	list.isEmpty()
push(e)	list.addFirst(e)
pop()	list.removeFirst()
top()	list.first()

Stack implemented using SLL

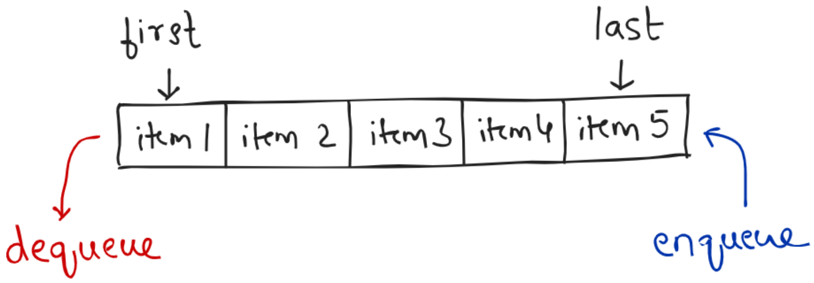
```java
public class LinkedStack<E> implements Stack<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>(); // an empty list

  public LinkedStack() { }      // new stack relies on the initially empty list

  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }

  public E top() { return list.first(); }
  public void push(E element) { list.addFirst(element); }
  public E pop() { return list.removeFirst(); }
}
```

Queues

Queues



Queues



Queues

- A queue is a two-ended linear data structure.
- A queue uses the first in, first out principle.
- A queue has two major operations: enqueue and dequeue, meaning insert and delete respectively.

Applications of queues

- Client/customer requests severed
  Movie tickets, bus tickets, plane tickets, etc
- Process scheduling in operating systems

Queue ADT

Method	Functionality
`enqueue(e)`	Adds element e to the back of queue.
`dequeue()`	Removes and returns the first element from the queue (or null if the queue is empty).
`first()`	Returns the first element of the queue, without removing it (or null if the queue is empty).
`size()`	Returns the number of elements in the queue.
`isEmpty()`	Returns a boolean indicating whether the queue is empty

Queue ADT interface

```java
public interface Queue<E> {
  /** Returns the number of elements in the queue. */
  int size();

  /** Tests whether the queue is empty. */
  boolean isEmpty();

  /** Inserts an element at the rear of the queue. */
  void enqueue(E e);

  /** Returns, but does not remove, the first element of the queue. */
  E first();

  /** Removes and returns the first element of the queue. */
  E dequeue();
}
```

Operations on a queue

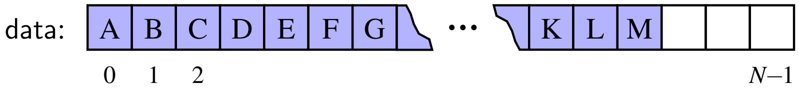
Method	Return value	first ← Q ← last
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
size()	2	(5, 3)
dequeue()	5	(3)
isEmpty()	false	(3)
dequeue()	3	()
isEmpty()	true	()
dequeue()	null	()
enqueue(7)	–	(7)
enqueue(9)	–	(7, 9)
first()	7	(7, 9)
enqueue(4)	–	(7, 9, 4)

java.util.Queue interface

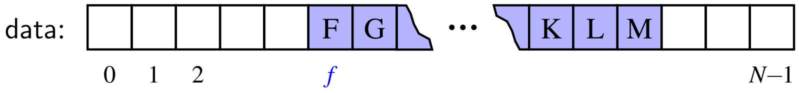
Our Queue ADT	Interface java.util.Queue	
	throws exceptions	returns special value
enqueue(e)	add(e)	offer(e)
dequeue()	remove()	poll()
first()	element()	peek()
size()	size()	
isEmpty()	isEmpty()	

Queues implemented using array

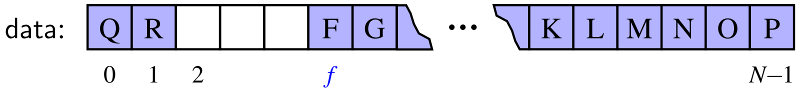
Queue front at index 0 always. Is there a problem?



Queue front at f. Is there a problem?



Queue front at f in circular array.



Queues implemented using array

```java
public class ArrayQueue<E> implements Queue<E> {
  public static final int CAPACITY = 1000;  // default array capacity
  private E[] data;                         // generic array used for storage
  private int f = 0;                        // index of the front element
  private int sz = 0;                       // current number of elements

  public ArrayQueue() {this(CAPACITY);}     // constructs queue with def. cap.
  public ArrayQueue(int capacity) {...}

  public int size() { return sz; }
  public boolean isEmpty() { return (sz == 0); }
  public void enqueue(E e) throws IllegalStateException {...}
  public E first() {...}
  public E dequeue() {...}
  public String toString() {...}
}
```

Queues implemented using array

```java
/** Returns, but does not remove, the first element of the queue. */
public E first() {
  if (isEmpty()) return null;
  return data[f];
}
```

```java
/** Returns a string representation of the queue as a list of elements.
This method runs in O(n) time, where n is the size of the queue. */
public String toString() {
  StringBuilder sb = new StringBuilder("(");
  int k = f;
  for (int j=0; j < sz; j++) {
    if (j > 0)
    sb.append(", ");
    sb.append(data[k]);
    k = (k + 1) % data.length;
  }
  sb.append(")");
  return sb.toString();
}
```

Queues implemented using array

- Enqueuing an element.
  avail = (f + sz) % data.length
  E.g.: When f = 5, sz = 3, data.length = 10,
  at what index the next element will be enqueued?
- Dequeuing an element.
  f = (f + 1) % data.length
  E.g.: When f = 3, data.length = 10,
  at what index the next element will be dequeued?

Queues implemented using array

```java
/** Inserts an element at the rear of the queue. */
public void enqueue(E e) throws IllegalStateException {
  if (sz == data.length) throw new IllegalStateException("Queue is full");
  int avail = (f + sz) % data.length;   // use modular arithmetic
  data[avail] = e;
  sz++;
}
```

```java
/** Removes and returns the first element of the queue. */
public E dequeue() {
  if (isEmpty()) return null;
  E answer = data[f];
  data[f] = null;                // dereference to help garbage collection
  f = (f + 1) % data.length;
  sz--;
  return answer;
}
```

Queues implemented using array: Complexity

Method	Running time
size	$\mathcal{O}(1)$
isEmpty	$\mathcal{O}(1)$
first	$\mathcal{O}(1)$
enqueue	$\mathcal{O}(1)$
dequeue	$\mathcal{O}(1)$

Queues implemented using SLL

```
/** Realization of a FIFO queue as an adaptation of a SinglyLinkedList. */
public class LinkedQueue<E> implements Queue<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>(); // an empty list
  public LinkedQueue() { } // new queue relies on the initially empty list
  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public void enqueue(E element) { list.addLast(element); }
  public E first() { return list.first(); }
  public E dequeue() { return list.removeFirst(); }
}
```

Circular queues implemented using CLL

```java
public interface CircularQueue<E> extends Queue<E> {
  /** Rotates the front element of the queue to the back of the queue.
  This does nothing if the queue is empty. */
  void rotate();
}
```

- Circular queues are useful for multiplayer, turn-based games, or round-robin scheduling of computing processes.
- LinkedCircularQueue class = CircularQueue interface + CircularlyLinkedList class.

Double-Ended Queues

Double-ended queues

- A double-ended queue or deque is a double-ended linear data structure that is more general than stack and queue.
- A deque has four major operations:
  addfirst, addlast, removefirst, and removelast.

Deque ADT

Method	Functionality
addFirst(e)	Insert a new element e at the front of the deque.
addLast(e)	Insert a new element e at the back of the deque.
removeFirst()	Remove and return the first element of the deque (or null if the deque is empty).
removeLast()	Remove and return the last element of the deque (or null if the deque is empty).
first()	Returns the first element of the deque without removing (or null if the deque is empty).
last()	Returns the last element of the deque without removing (or null if the deque is empty).
size()	Returns the number of elements in the deque.
isEmpty()	Returns a boolean indicating whether the deque is empty.

Deque ADT interface

```java
public interface Deque<E> {
  /** Returns the number of elements in the deque. */
  int size();
  /** Tests whether the deque is empty. */
  boolean isEmpty();

  /** Returns (but does not remove) the first element of the deque. */
  E first();
  /** Returns (but does not remove) the last element of the deque. */
  E last();

  /** Inserts an element at the front of the deque. */
  void addFirst(E e);
  /** Inserts an element at the back of the deque. */
  void addLast(E e);

  /** Removes and returns the first element of the deque. */
  E removeFirst();
  /** Removes and returns the last element of the deque. */
  E removeLast();
}
```

Operations on a deque

Method	Return value	Deque
addLast(5)	–	(5)
addFirst(3)	–	(3, 5)
addFirst(7)	–	(7, 3, 5)
first()	7	(7, 3, 5)
removeLast()	5	(7, 3)
size()	2	(7, 3)
removeLast()	3	(7)
removeFirst()	7	()
addFirst(6)	–	(6)
last()	6	(6)
addFirst(8)	–	(8, 6)
isEmpty()	false	(8, 6)
last()	6	(8, 6)

Implementing a deque

- Using circular array.
  removeFirst: f = (f + 1) % data.length
  removeLast: No change to f
  addLast: avail = (f + n) % N
  addFirst: avail = (f − 1 + N) % N
  (Why don't we simply use (f − 1) % N? When f = 0, this feature leads to −1 % N. In Java, −1 % N = −1 when N is large.)
- Using DLL.
  `public class LinkedDeque<E> implements Deque<E>`

Deque via circular array or DLL: Complexity

Method	Running time
size, isEmpty	$\mathcal{O}(1)$
first, last	$\mathcal{O}(1)$
addFirst, addLast	$\mathcal{O}(1)$
removeFirst, removeLast	$\mathcal{O}(1)$

java.util.Deque interface

Our Deque ADT	Interface java.util.Deque	
	throws exceptions	returns special value
first()	getFirst()	peekFirst()
last()	getLast()	peekLast()
addFirst(e)	addFirst(e)	offerFirst(e)
addLast(e)	addLast(e)	offerLast(e)
removeFirst()	removeFirst()	pollFirst()
removeLast()	removeLast()	pollLast()
size()	size()	
isEmpty()	isEmpty()	