Fondamenti di Informatica (Elettronici)

Liste, pile, code in Julia – Lezione 36

13 gennaio 2021

Liste, pile, code in Julia

1. Linked List (Liste collegate)
2. Dequeue
3. Stack (Pila)
4. Queue (Coda)
5. Interfaccia iterator
6. Julia `DataStructures.jl`

Section 1

Linked List (Liste collegate)

Linked list

Il tipo astratto lista consente di rappresentare sequenze di valori detti atomi, tutti dello stesso tipo

Si parla di liste semplici quando i valori della sequenza non siano essi stessi delle liste

Negli esempi di questa lezione i valori degli elementi di una lista saranno valori interi

Quanto diremo si generalizza facilmente a liste di elementi qualsiasi.

Lista come tipo di dato astratto

Un tipo di dato astratto o ADT (Abstract Data Type), è un tipo di dato le cui istanze possono essere manipolate con modalità che dipendono esclusivamente dalla semantica del dato e non dalla sua realizzazione (implementazione).

LISTA — Definizione

Tipo astratto `< S,F,C >`, dove

```
S = < List, Atomo, Boolean >,
F = < cons, car, cdr, null >,
C = < lista_vuota >;
```

con `List` detto dominio di interesse, e dove

$$\texttt{cons} : \texttt{Atomo} \times \texttt{List} \to \texttt{List},$$
$$\texttt{car} : \texttt{List} \to \texttt{Atomo},$$
$$\texttt{cdr} : \texttt{List} \to \texttt{List},$$
$$\texttt{null} : \texttt{List} \to \texttt{Boolean}.$$

Descrizione e semantica delle operazioni

cons prende in ingresso un valore `a` $\in$ `Atomo` e un valore `L` $\in$ `List`; restituisce in uscita la lista semplice data da a seguito da tutti i valori di L;

Descrizione e semantica delle operazioni

cons prende in ingresso un valore `a` $\in$ `Atomo` e un valore `L` $\in$ `List`; restituisce in uscita la lista semplice data da a seguito da tutti i valori di L;

car prende in ingresso un valore `L` $\in$ `List`;
se L e' la lista vuota, allora `car(L)` non e' definita; altrimenti restituisce il primo atomo di `L`;

Descrizione e semantica delle operazioni

cons prende in ingresso un valore `a` $\in$ `Atomo` e un valore `L` $\in$ `List`; restituisce in uscita la lista semplice data da a seguito da tutti i valori di L;

car prende in ingresso un valore `L` $\in$ `List`;
se L e' la lista vuota, allora `car(L)` non e' definita; altrimenti restituisce il primo atomo di `L`;

cdr prende in ingresso un valore `L` $\in$ `List`;
se `L` e' la lista vuota, allora `cdr(L)` non e' definita; altrimenti restituisce la lista semplice costituita da tutti gli atomi di `L` meno il primo;

Descrizione e semantica delle operazioni

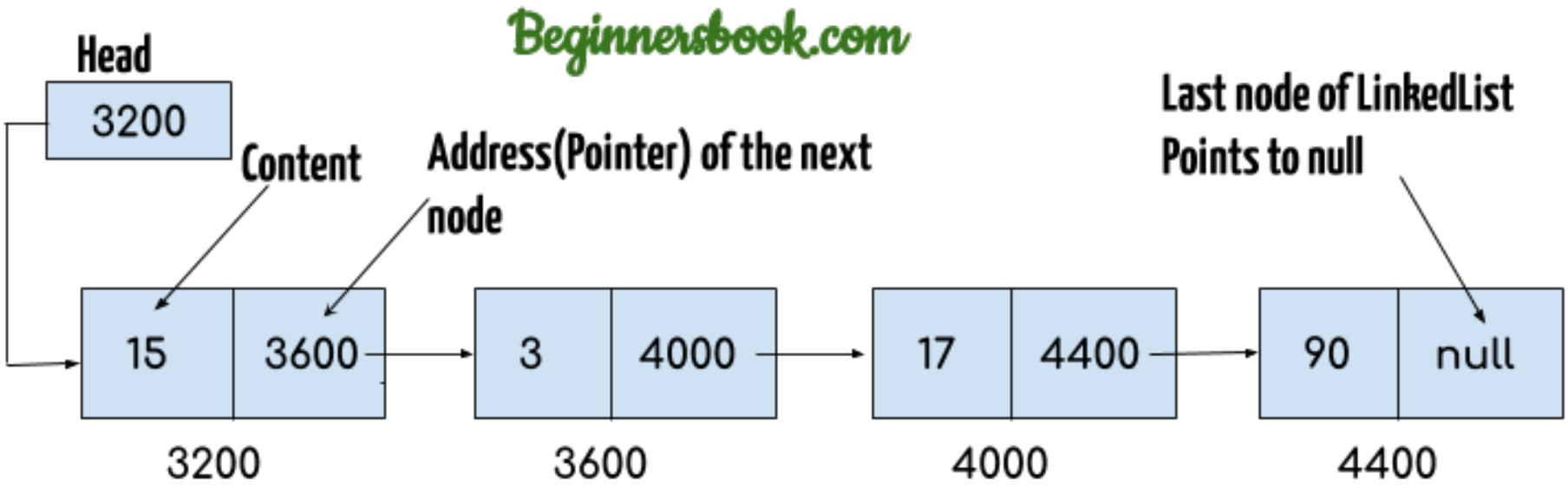
cons prende in ingresso un valore `a` $\in$ `Atomo` e un valore `L` $\in$ `List`; restituisce in uscita la lista semplice data da a seguito da tutti i valori di L;

car prende in ingresso un valore `L` $\in$ `List`;
se L e' la lista vuota, allora `car(L)` non e' definita; altrimenti restituisce il primo atomo di `L`;

cdr prende in ingresso un valore `L` $\in$ `List`;
se `L` e' la lista vuota, allora `cdr(L)` non e' definita; altrimenti restituisce la lista semplice costituita da tutti gli atomi di `L` meno il primo;

null prende in ingresso un valore `L` $\in$ `List`;
restituisce `true` se la lista `L` e' vuota; `false` altrimenti.

Rappresentazione grafica



Esempi di liste collegate

```
julia> l1 = nil()
nil()

julia> l2 = cons(1, l1)
list(1)

julia> l3 = list(2, 3)
list(2, 3)

julia> l4 = cat(l1, l2, l3)
list(1, 2, 3)

julia> l5 = map((x) -> x*2, l4)
list(2, 4, 6)

julia> for i in l5; print(" $i"); end
 2 4 6
julia> collect(l5)
3-element Array{Any,1}:
 2
 4
 6
```

Section 2

Dequeue

Tipo Deque

Il tipo Deque implementa una coda a doppia estremità utilizzando un elenco di blocchi.

- Questa struttura dati supporta l'inserimento / la rimozione in tempo costante di elementi a entrambe le estremità di una sequenza.

Tipo Deque

Il tipo `Deque` implementa una coda a doppia estremità utilizzando un elenco di blocchi.

- Questa struttura dati supporta l'inserimento / la rimozione in tempo costante di elementi a entrambe le estremità di una sequenza.
- Nota: anche il tipo `Vector` di Julia fornisce questa interfaccia e quindi può essere utilizzato come `deque`.

Tipo Deque

Il tipo `Deque` implementa una coda a doppia estremità utilizzando un elenco di blocchi.

- Questa struttura dati supporta l'inserimento / la rimozione in tempo costante di elementi a entrambe le estremità di una sequenza.
- Nota: anche il tipo `Vector` di Julia fornisce questa interfaccia e quindi può essere utilizzato come deque.
- Tuttavia, il tipo `Deque` in questo pacchetto è implementato come un elenco di blocchi contigui (dimensione predefinita = 2K).

Tipo Deque

Il tipo `Deque` implementa una coda a doppia estremità utilizzando un elenco di blocchi.

- Questa struttura dati supporta l'inserimento / la rimozione in tempo costante di elementi a entrambe le estremità di una sequenza.
- Nota: anche il tipo `Vector` di Julia fornisce questa interfaccia e quindi può essere utilizzato come `deque`.
- Tuttavia, il tipo `Deque` in questo pacchetto è implementato come un elenco di blocchi contigui (dimensione predefinita = 2K).
- Man mano che un `deque` cresce, nuovi blocchi possono essere creati e collegati a blocchi esistenti.

Tipo Deque

Il tipo `Deque` implementa una coda a doppia estremità utilizzando un elenco di blocchi.

- Questa struttura dati supporta l'inserimento / la rimozione in tempo costante di elementi a entrambe le estremità di una sequenza.
- Nota: anche il tipo `Vector` di Julia fornisce questa interfaccia e quindi può essere utilizzato come `deque`.
- Tuttavia, il tipo `Deque` in questo pacchetto è implementato come un elenco di blocchi contigui (dimensione predefinita $= 2\mathtt{K}$).
- Man mano che un `deque` cresce, nuovi blocchi possono essere creati e collegati a blocchi esistenti.
- In questo modo si evita la copia durante la crescita di un vettore.

Tipo Deque

Il tipo `Deque` implementa una coda a doppia estremità utilizzando un elenco di blocchi.

- Questa struttura dati supporta l'inserimento / la rimozione in tempo costante di elementi a entrambe le estremità di una sequenza.
- Nota: anche il tipo `Vector` di Julia fornisce questa interfaccia e quindi può essere utilizzato come `deque`.
- Tuttavia, il tipo `Deque` in questo pacchetto è implementato come un elenco di blocchi contigui (dimensione predefinita = `2K`).
- Man mano che un `deque` cresce, nuovi blocchi possono essere creati e collegati a blocchi esistenti.
- In questo modo si evita la copia durante la crescita di un vettore.
- Il benchmark mostra che le prestazioni di `Deque` sono paragonabili a `Vector` su `push!`.

Tipo Deque

Il tipo `Deque` implementa una coda a doppia estremità utilizzando un elenco di blocchi.

- Questa struttura dati supporta l'inserimento / la rimozione in tempo costante di elementi a entrambe le estremità di una sequenza.
- Nota: anche il tipo `Vector` di Julia fornisce questa interfaccia e quindi può essere utilizzato come `deque`.
- Tuttavia, il tipo `Deque` in questo pacchetto è implementato come un elenco di blocchi contigui (dimensione predefinita = `2K`).
- Man mano che un `deque` cresce, nuovi blocchi possono essere creati e collegati a blocchi esistenti.
- In questo modo si evita la copia durante la crescita di un vettore.
- Il benchmark mostra che le prestazioni di `Deque` sono paragonabili a `Vector` su `push!`.
- Ed è notevolmente più veloce su `pushfirst!` (da circa il 30% al 40%).

Uso della struttura dequeue

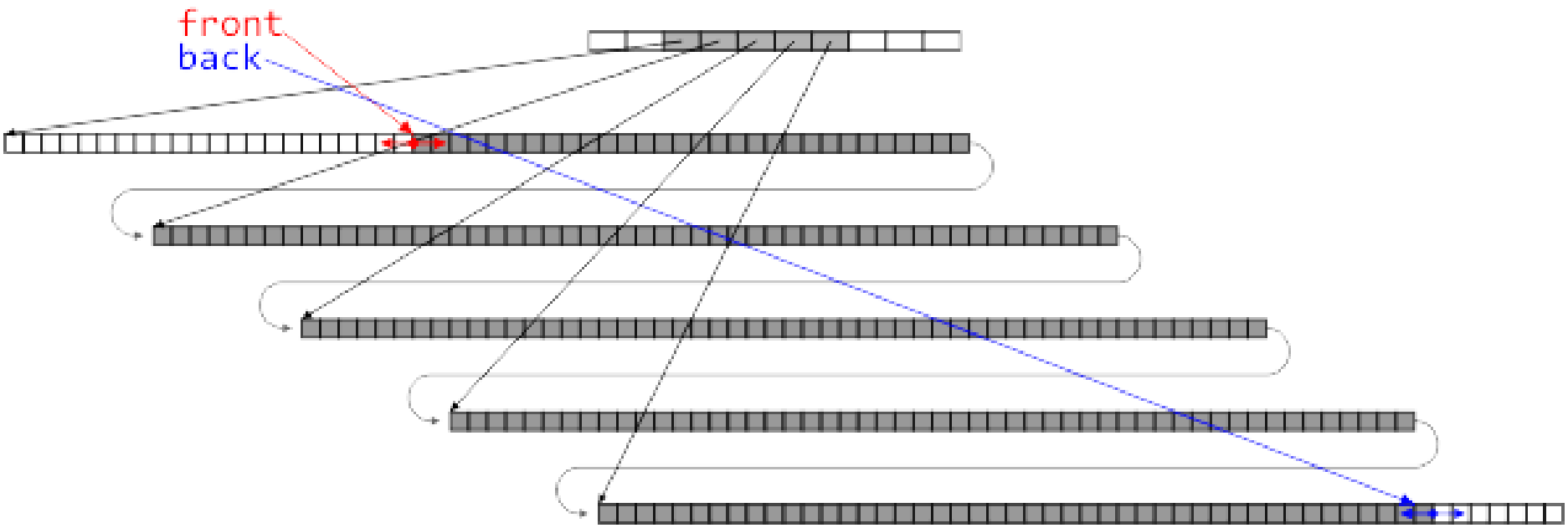
```
julia> a = Deque{Int}()      # creazione di un oggetto Deque
Deque [Int64[]]

julia> typeof(a)             # interrogazione di tipo
Deque{Int64}

julia> isempty(a)            # test whether the dequeue is empty
true

julia> length(a)             # get the number of elements
0

julia> push!(a, 10)          # add an element to the back
Deque [[10]]

julia> pop!(a)               # remove an element from the back
10
```

Uso della struttura dequeue

```
julia> pushfirst!(a, 20)    # add an element to the front
Deque [[20]]

julia> pushfirst!(a, 30)    # add an element to the front
Deque [[30, 20]]

julia> popfirst!(a)         # remove an element from the front
30

julia> first(a)             # get the element at the front
20

julia> b = collect(a)       # convert to Array
1-element Array{Int64,1}:
 20
```

Rappresentazione grafica

La struttura dati deque può essere usata sia come pila che come coda.



Operazioni

Le operazioni di base su un deque sono `enqueue` e `dequeue` su entrambe le estremità.

Inoltre generalmente vengono implementate le operazioni di `peek`, che restituiscono il valore a tale fine senza rimuoverlo dalla coda.

I nomi variano tra i linguaggi; le principali implementazioni includono:

operation	common name(s)	Ada	C++	Java	Perl	PHP	Python	Ruby	Rust	JavaScript
insert element at back	inject, snoc, push	`Append`	`push_back`	`offerLast`	`push`	`array_push`	`append`	`push`	`push_back`	`push`
insert element at front	push, cons	`Prepend`	`push_front`	`offerFirst`	`unshift`	`array_unshift`	`appendleft`	`unshift`	`push_front`	`unshift`
remove last element	eject	`Delete_Last`	`pop_back`	`pollLast`	`pop`	`array_pop`	`pop`	`pop`	`pop_back`	`pop`
remove first element	pop	`Delete_First`	`pop_front`	`pollFirst`	`shift`	`array_shift`	`popleft`	`shift`	`pop_front`	`shift`
examine last element	peek	`Last_Element`	`back`	`peekLast`	`$array[-1]`	`end`	`<obj>[-1]`	`last`	`back`	`<obj>[<obj>.length - 1]`
examine first element		`First_Element`	`front`	`peekFirst`	`$array[0]`	`reset`	`<obj>[0]`	`first`	`front`	`<obj>[0]`

Section 3

Stack (Pila)

Pila e Coda

I tipi Stack e Queue sono un wrapper leggero di tipo deque, che forniscono rispettivamente interfacce per l'accesso LIFO e FIFO.

Pila e Coda

I tipi Stack e Queue sono un wrapper leggero di tipo deque, che forniscono rispettivamente interfacce per l'accesso LIFO e FIFO.

LIFO Stack (tipo di dati astratto), una struttura di dati che fornisce semantica last-in-first-out; chiamata anche coda (protocollo) LIFO

Pila e Coda

I tipi Stack e Queue sono un wrapper leggero di tipo deque, che forniscono rispettivamente interfacce per l'accesso LIFO e FIFO.

LIFO Stack (tipo di dati astratto), una struttura di dati che fornisce semantica last-in-first-out; chiamata anche coda (protocollo) LIFO

Pila e Coda

I tipi Stack e Queue sono un wrapper leggero di tipo deque, che forniscono rispettivamente interfacce per l'accesso LIFO e FIFO.

LIFO Stack (tipo di dati astratto), una struttura di dati che fornisce semantica last-in-first-out; chiamata anche coda (protocollo) LIFO

FIFO Queue (tipo di dati astratto), acronimo per first-in-first-out, è un metodo per la manipolazione di una struttura di dati dove la più vecchia (prima) voce, o 'testa' della coda, viene elaborato per prima.

Uso della struttura dati

```
julia> s = Stack{Int}()                    # create a stack
Stack{Int64}(Deque [Int64[]])

julia> isempty(s)                          # check whether the stack is empty
true

julia> length(s)                           # get the number of elements
0

julia> eltype(s)                           # get the type of elements
Any

julia> push!(s, 1)                         # push back a item
Stack{Int64}(Deque [[1]])
```

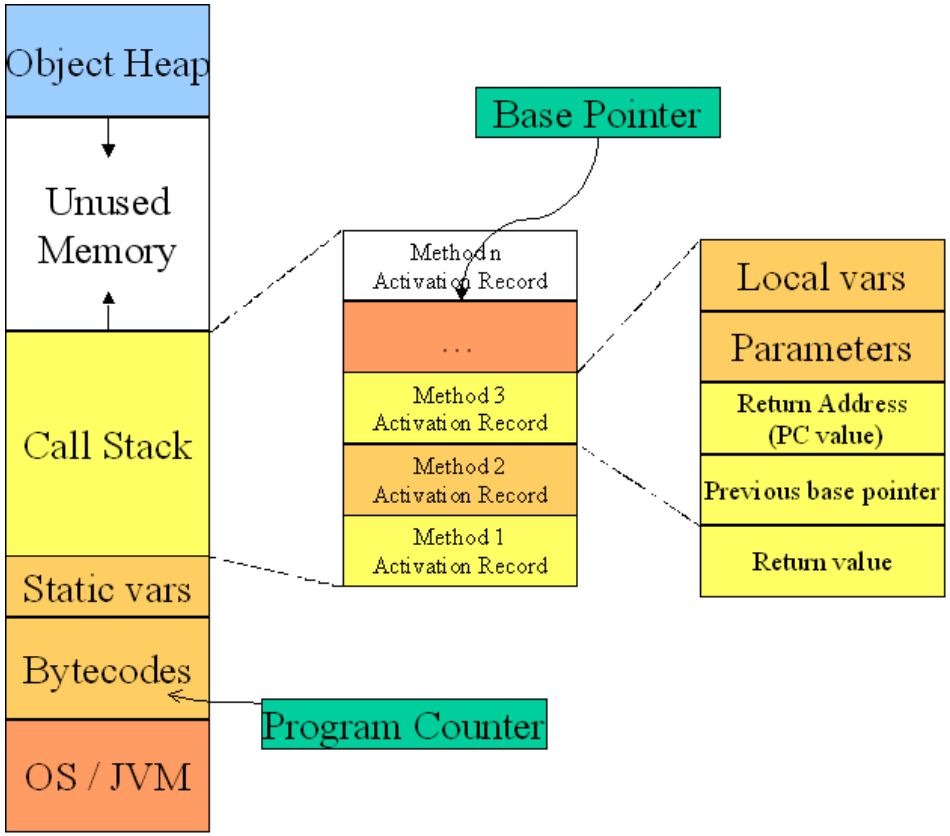
Uso della struttura dati

```
julia> first(s)                          # get an item from the top of stack
1

julia> pop!(s)                           # get and remove a first item
1

julia> empty!(s)                         # make a stack empty
Stack{Int64}(Deque [Int64[]])

julia> iterate(s::Stack)                 # Get a LIFO iterator of a stack

julia> Iterators.reverse(s::Stack)       # Get a FILO iterator of a stack

julia> s1 == s2                          # check whether the two stacks are s
```

Rappresentazione grafica

Tipo di dato astratto che viene usato in diversi contesti per riferirsi a strutture dati, le cui modalità d'accesso ai dati è LIFO.

Esempio tipico: lo stack delle chiamate tra i sottoprogrammi (Java in figura)



Section 4

Queue (Coda)

Section 5

Interfaccia iterator

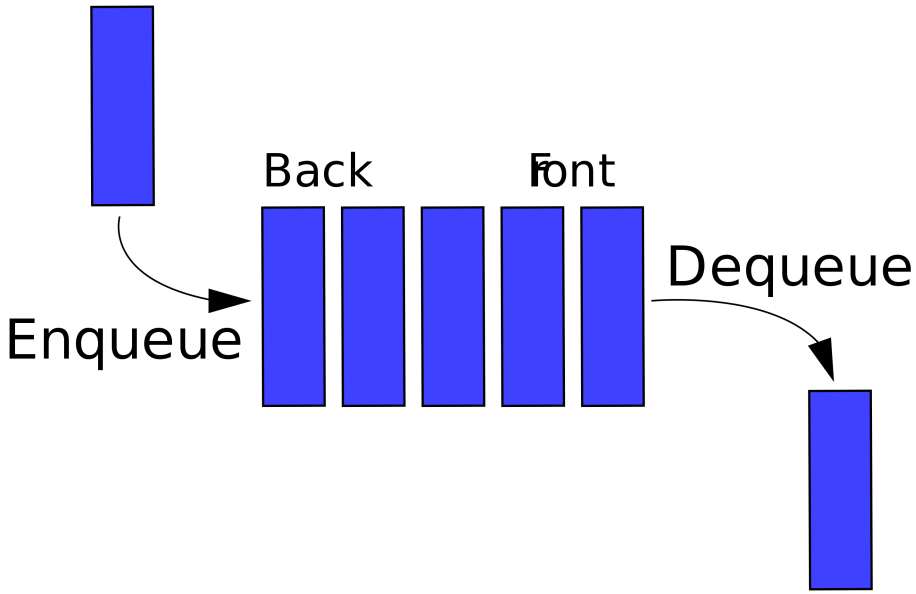
Uso della struttura dati

```
julia> q = Queue{Int}()
Queue{Int64}(Deque [Int64[]])

julia> enqueue!(q, x)
Queue{Int64}(Deque [[100]])

julia> x = first(q)
100

julia> x = last(q)
```

Rappresentazione grafica

L'operazione di aggiunta di un elemento nella parte posteriore della coda è nota come `enqueue` e l'operazione di rimozione di un elemento dalla parte anteriore è nota come `dequeue`.

Possono essere consentite anche altre operazioni, spesso inclusa un'operazione `peek` o `front` che restituisce il valore dell'elemento successivo da rimuovere dalla coda senza rimuoverlo.



Queue

Time complexity in big O notation

Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(n)$	$O(n)$
Insert	$O(1)$	$O(1)$
Delete	$O(1)$	$O(1)$

Code a priorità

Una coda di priorità è un tipo di dati astratto simile a una coda normale ovvero a una struttura a `stack` in cui ogni elemento ha anche una "priorità" associata ad esso.

- In una coda con priorità, un elemento con priorità alta viene servito prima di un elemento con priorità bassa.

Code a priorità

Una coda di priorità è un tipo di dati astratto simile a una coda normale ovvero a una struttura a `stack` in cui ogni elemento ha anche una "priorità" associata ad esso.

- In una coda con priorità, un elemento con priorità alta viene servito prima di un elemento con priorità bassa.
- In alcune implementazioni, se due elementi hanno la stessa priorità, vengono serviti in base all'ordine in cui sono stati accodati, mentre in altre implementazioni, l'ordine degli elementi con la stessa priorità non è definito.

Code a priorità

Una coda di priorità è un tipo di dati astratto simile a una coda normale ovvero a una struttura a `stack` in cui ogni elemento ha anche una "priorità" associata ad esso.

- In una coda con priorità, un elemento con priorità alta viene servito prima di un elemento con priorità bassa.
- In alcune implementazioni, se due elementi hanno la stessa priorità, vengono serviti in base all'ordine in cui sono stati accodati, mentre in altre implementazioni, l'ordine degli elementi con la stessa priorità non è definito.
- Sebbene le code di priorità siano spesso implementate con gli `heap`, sono concettualmente distinte dagli `heap`.

Code a priorità

Una coda di priorità è un tipo di dati astratto simile a una coda normale ovvero a una struttura a `stack` in cui ogni elemento ha anche una "priorità" associata ad esso.

- In una coda con priorità, un elemento con priorità alta viene servito prima di un elemento con priorità bassa.
- In alcune implementazioni, se due elementi hanno la stessa priorità, vengono serviti in base all'ordine in cui sono stati accodati, mentre in altre implementazioni, l'ordine degli elementi con la stessa priorità non è definito.
- Sebbene le code di priorità siano spesso implementate con gli `heap`, sono concettualmente distinte dagli `heap`.
- Una coda di priorità è un concetto come "una lista" o "una mappa"; proprio come un elenco può essere implementato con un elenco collegato o un array, una coda di priorità può essere implementata con un `heap` o una varietà di altri metodi come un array non ordinato.

Code a priorità in `DataStructures`

Il tipo `PriorityQueue` fornisce un'implementazione di base di coda di priorità che consente chiavi arbitrarie e tipi di priorità.

Non sono consentite più chiavi identiche, ma la priorità delle chiavi esistenti può essere modificata in modo efficiente.

Operazioni su PriorityQueue

```
PriorityQueue{K, V}()          # construct a new priority queue with keys of type K
                                 and priorities of type V (forward ordering by default)

PriorityQueue{K, V}(ord)       # construct a new priority queue with the given types
                               and ordering ord (Base.Order.Forward or Base.Order.Revers

enqueue!(pq, k, v)             # insert the key k into pq with priority v

enqueue!(pq, k=>v)             # (same, using Pairs)

dequeue!(pq)                   # remove and return the lowest priority key

dequeue_pair!(pq)              # remove and return the lowest priorty key and value

peek(pq)                       # return the lowest priority key and value without
removing it

delete!(pq, k)                 # delete the mapping for the given key in a priority queue
                                 and return the priority queue.
```

Code a priorità: esempio

```
julia> # Julia code
       pq = PriorityQueue();

julia> # Insert keys with associated priorities
       pq["a"] = 10; pq["b"] = 5; pq["c"] = 15; pq
PriorityQueue{Any,Any,Base.Order.ForwardOrdering} with 3 entries:
  "c" => 15
  "b" => 5
  "a" => 10

julia> # Change the priority of an existing key
       pq["a"] = 0; pq
PriorityQueue{Any,Any,Base.Order.ForwardOrdering} with 3 entries:
  "c" => 15
  "b" => 5
  "a" => 0
```

Rappresentazione grafica

operation	argument	return value	size	contents (unordered)	contents (ordered)
insert	P		1	P	P
insert	Q		2	P Q	P Q
insert	E		3	P Q E	E P Q
remove max		Q	2	P E	E P
insert	X		3	P E X	E P X
insert	A		4	P E X A	A E P X
insert	M		5	P E X A M	A E M P X
remove max		X	4	P E M A	A E M P
insert	P		5	P E M A P	A E M P P
insert	L		6	P E M A P L	A E L M P
insert	E		7	P E M A P L E	A E E L M
remove max		P	6	E E M A P L	A E E L M

A sequence of operations on a priority queue

Section 6

Julia DataStructures.jl

Package DataStructures.jl

https://juliacollections.github.io/DataStructures.jl/latest/

DataStructures.jl

This package implements a variety of data structures, including

- Deque (implemented with an unrolled linked list)
- CircularBuffer
- CircularDeque (based on a circular buffer)
- Stack
- Queue
- Priority Queue
- Fenwick Tree
- Accumulators and Counters (i.e. Multisets / Bags)
- Disjoint Sets
- Binary Heap
- Mutable Binary Heap
- Ordered Dicts and Sets
- RobinDict and OrderedRobinDict (implemented with Robin Hood Hashing)
- SwissDict (inspired from SwissTables)
- Dictionaries with Defaults
- Trie
- Linked List and Mutable Linked List
- Sorted Dict, Sorted Multi-Dict and Sorted Set
- DataStructures.IntSet
- SparseIntSet
- DiBitVector
- Red Black Tree
- AVL Tree
- Splay Tree