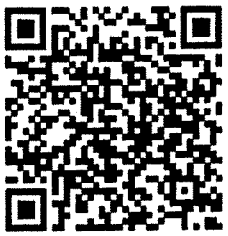
Försättsblad till skriftlig tentamen vid Linköpings universitet



Datum för tentamen	2017-04-06
Sal (1)	SU-salar, IDA(47)
Tid	17-19
Kurskod	TDDC74
Provkod	KTR4
Kursnamn/benämning **Provnamn/benämning**	Programmering - abstraktion och modellering Frivillig datordugga
Institution	IDA
Antal uppgifter som ingår i tentamen	4
Jour/Kursansvarig Ange vem som besöker salen	Jalal Maleki
Telefon under skrivtiden	0706071963
Besöker salen ca klockan	ja
Kursadministratör/kontaktperson (namn + tfnr + mailaddress)	Anna Grabska Eklund, anna.grabska.eklund@liu.se, ankn. 2362
Tillåtna hjälpmedel	Inga
Övrigt	
Antal exemplar i påsen	

TDDC74 Programmering: Abstraktion och modellering
Dugga 2, 2017-04-06, kl 17-19

Läs alla frågorna först och bestäm dig för i vilken ordning du vill lösa uppgifterna. Uppgifterna är inte nödvändigtvis i svårighetsordning.

Använd **väl valda namn** på parametrar och **indentera** din kod. Väl valda namn inkluderar bland annat konsekvent språk. Du behöver inte skriva kodkommentarer, annat än för väldigt svårförklarade hjälpfunktioner (som bör undvikas). Namngivning ska vara tillräcklig (och följa konventioner). Skriv inte onödigt komplicerade lösningar.

Om det är naturligt, definiera gärna hjälpfunktioner! Hjälpfunktioner ska som vanligt lösa tydliga och lättförklarade uppgifter.

Du får använda **alla tillgängliga primitiver** och språkkkonstruktioner i Racket, **om annat inte anges** i uppgiftstexten.

Frågor

Är något otydligt i uppgifterna kan du använda meddelande-funktionen i tentaklienten för att skicka frågor till rättande lärare.

Att lämna in

Skicka in uppgifterna med hjälp av tentaklienten, när du är klar med dem! Du får svar via klienten när din uppgift är rättad. Vänta *inte* på att alla uppgifter är klara med att lämna in. När du har lämnat in en uppgift, fortsätt arbeta på nästa. Du har en inlämning per uppgift (så skicka inte ev a- och b-uppgifter separat). **Följ funktionsnamnen som står i uppgifterna, och testa att alla körexempel i uppgiften fungerar exakt som de är inskrivna!**

Din totalpoäng på duggan ges som summan av lösningar du lämnat in i tid, oavsett om de hinner rättas klart inom skrivtiden eller ej.

Betyg

Det finns två duggor i kursen. På varje dugga kan man få som mest 15p. Totalpoängen avgör slutbetyget i kursen. För trea räcker 50% av det totala poängen.

Ett resultat man inte är nöjd med, kan plussas vid tentamenstillfället i juni.

Lycka till!

Uppgift 1, Omgivningsdiagram (3 poäng)

Skriv dina svar på uppgift a-c som en textkommentar i **uppg1.rkt** och skicka in den.

Följande sekvens av uttryck evalueras (i ordningen de står):

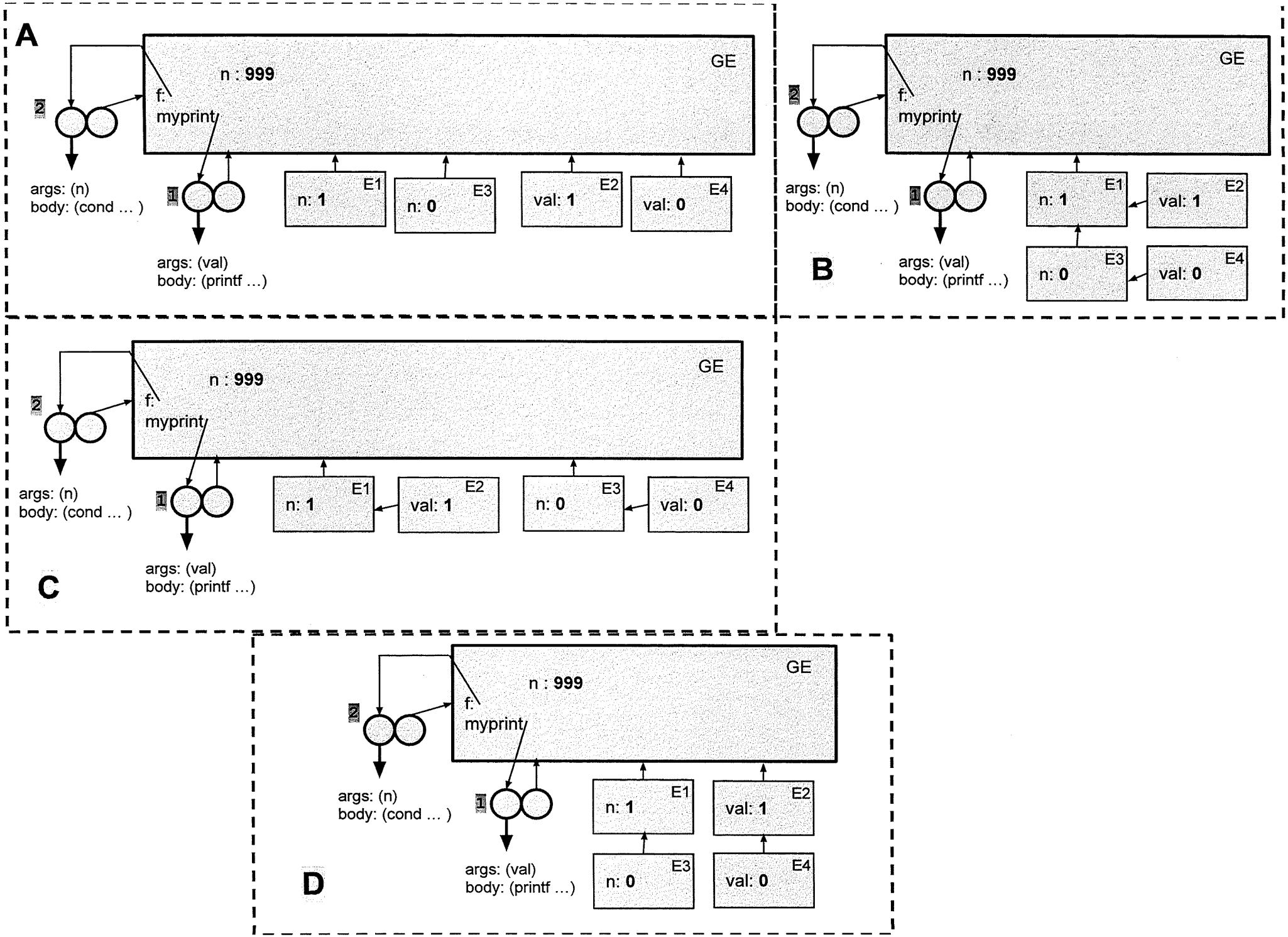
```
(define n 999)

(define (myprint val)
  (printf "~a ~n" val)) ;; printf inbyggd, ger ej ramar

(define (f n)
  (cond
    [(zero? n)
     (myprint n)
     0]
    [else
     (myprint n)
     (+ n (f (- n 1)))]))

(f 1)
```

a) Detta ger upphov till ett av fyra omgivningsdiagram på nästa sida. Vilket av dem? Det räcker att ange en bokstav. För att minska risken för rena gissningar ger ett felaktigt svar på just denna deluppgift (deluppgift a) 0.5p avdrag på uppgiftens totalpoäng[1]. Du behöver inte motivera ditt svar. Du kan svara blankt på a-uppgiften om du vill.

b) Välj ut **exakt ett** av de tre felaktiga omgivningsdiagrammen. Nämn kortfattat (en-två meningar) något som är fel i diagrammet, och *varför* det är fel. Ramarna (CE, E1, ...) och procedurobjekten (1, ...) är märkta. Använd dessa benämningar när du hänvisar till delar av diagrammet.

c) Välj ut **exakt ett** av de tre felaktiga omgivningsdiagrammen (annat än det i b-uppgiften). Nämn kortfattat (en-två meningar) något som är fel i diagrammet, och *varför* det är fel. Ramarna (CE, E1, ...) och procedurobjekten (1, ...) är märkta. Använd dessa benämningar när du hänvisar till delar av diagrammet.

[1]Den kan alltså ge som minst 0p.


A
GE
n : 999
f:
myprint
2
1
args: (n)
body: (cond …)
args: (val)
body: (printf …)
E1
n: 1
E3
n: 0
E2
val: 1
E4
val: 0
B
C
D

Uppgift 2, Dataabstraktion (4 poäng)

I matematiken stöter vi på polynom, som exempelvis $p(x) = 8x^3 + 7x^2 + 6x + 5$. Vi vill nu skapa en abstrakt datatyp `poly` som representerar ett polynom. När vi skapar polynomet anger vi helt enkelt dess lista av koefficienter, där vi börjar med konstanttermen. p ovan skulle alltså ha koefficient-listan (5 6 7 8), och polynomet $q(x) = x^4 + 3x^2 + 7$ skulle ha listan (7 0 3 0 1). Det sista ser vi kanske tydligare om vi skriver ut det som $q(x) = 1x^4 + 0x^3 + 3x^2 + 0x^1 + 7x^0$. Ett polynoms grad (en: degree) är den högsta förekommande exponenten. $p(x)$ har grad 3, $r(x) = 5$ har grad 0.

Lämna in dina svar i filen **uppg2.rkt**. Ha med raden `(provide (all-defined-out))` i filen.

a) Din uppgift är att implementera en datatyp för polynom, genom dessa fyra funktioner:

- En konstruktor `make-poly` som tar en lista av koefficienter och skapar ett polynom. Vi kan anta att användaren anger polynom utan avslutande nollor (så $p(x)$ ovan skapas alltid med listan (5 6 7 8), och inte t ex (5 6 7 8 0 0 0...)).
- Ett predikat `poly?` som kontrollerar om ett objekt är ett polynom[2].
- `poly-degree` tar ett polynom och returnerar dess grad.
- `poly-coeff` tar ett polynom och en exponent k ($k \geq 0$ heltal), och returnerar koefficienten framför termen x^k.

Så här ska det fungera:

```
> (define p (make-poly '(5 6 7 8)))
> (poly? '(5 6 7 8))  ;; var listan ett polynom?
#f
> (poly? p)           ;; blir det vi skapade ett polynom?
#t
> (poly-degree p)
3
> (poly-coeff p 3)  ;; ge mig koefficienten vid x^3
8
> (poly-coeff p 5)  ;; ge mig koefficienten vid x^5
0
```

OBS! Du kan ha hjälp av funktionen `list-ref`, som givet en lista och ett index i (0,...) returnerar elementet på plats i. `(list-ref '(a b c) 1) => b`.

[2]Korrekt utdata från make-poly. Du behöver inte kontrollera att användaren gjort rätt när de anropade make-poly.

b) Vi vill också kunna "skala om" ett polynom. Om vi har ett polynom $p(x)$ och en skalär (en siffra) λ så vill vi kunna skapa det nya polynomet $q(x) = \lambda p(x)$. Skriv en funktion `(poly-scale p l)` som returnerar ett nytt polynom $q(x)$ enligt beskrivningen. Så här ska det fungera:

```
> (define p (make-poly '(7 8))) ;; p(x) = 8x + 7
> (define q (poly-scale p 3)) ;; q(x) = 3p(x) = (3*8)x + (3*7) = 24x + 21
> (eqv? (poly-degree p) (poly-degree q)) ;; samma grad
#t
> (poly-coeff p 0)
7
> (poly-coeff q 0)
21
> (poly-coeff p 1)
8
> (poly-coeff q 1)
24
> (define r (poly-scale p 0)) ;; OBS. r(x) = 0
> (eqv? (poly-degree p) (poly-degree r))
#f
> (poly-degree r)
0
> (poly-coeff r 0)
0
```

OBS! Du måste använda abstraktionen (från a-uppgiften). Du kan inte anta något mer om polynomet än att funktionerna ovan (`make-poly...`) fungerar.

Uppgift 3, Tillståndsmodellering med objekt, closures (4 poäng)

Vi vill (som minimalt exempel på objekt med tillstånd) representera knappar. De kan vara antingen på eller av, och ska hantera hantera kommandona `'flip` (som växlar om knappen är på eller av, och returnerar det nya tillståndet), och `'status` som returnerar tillståndet. Procedurerna ska ta exakt ett argument. Om de får något annat än `'flip` eller `status` ska inget hända. För att skapa en knapp kör vi `make-switch`.

Lämna in dina svar i filen **uppg3.rkt**. Ha med raden `(provide (all-defined-out))` i filen.

Så här ska det fungera:

```
> (define switch-a (make-switch))
> (define switch-b (make-switch))
> (switch-a 'status)
#f
> (switch-b 'status)
#f
> (switch-a 'flip)
#t
> (switch-a 'some-other-arg) ;; inget händer
> (switch-a 'status)
#t
> (switch-b 'status) ;; verifierat oberoende
#f
```

a) Skriv kod för `make-switch`.

b) Vad är det som gör att `switch-a` och `switch-b` är oberoende? Vad *i din kod* är det som gör att de inte råkar nå varandras tillstånd? Förklara kortfattat (max en-två meningar). Svara som en kommentar i koden.

Uppgift 4, Muterbara struturer (4 poäng)

Du får två dataabstraktioner, en för en struktur som heter Triple och en för Deque. Elementen i en deque representeras som en Triple.

Din uppgift är att använda den givna dataabstraktionen för att skriva en procedur `deque-insert-end!` som tar två argument (`deque-insert-end! dq value`) och lägger andra argumentet i ett nytt element som placeras i **slutet** av dequen `dq`. Din procedur ska ändra i själva dequen (snarare än att returnera en ny deque).

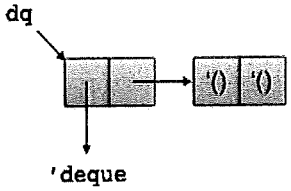
Om du är osäker på hur det hela fungerar, se bifogad skiss över deque-strukturen. Dessutom har tentavakterna kladdpapper om du vill göra egna skisser. Du ska inte lämna in dina papper.

Till din hjälp har du en del kod given, som beskriver hur abstraktionen ska fungera. Du får inte ändra i den. **Koden finns i filen uppgift4-y.rkt på din tentadator**, så du kan kopiera därifrån. Den finns också bifogad sist i tentan. Funktionen `deque->list`, som finns i denna fil, kan användas för att konvertera en deque till en vanlig lista för att se dequens innehåll.

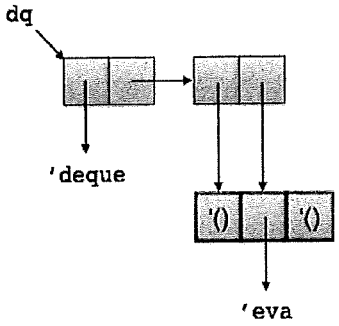
Lämna in dina svar i filen **uppg4.rkt**, där du inkluderar all den givna koden. Ha med raden (`provide (all-defined-out)`) i filen.

Skiss över deque-strukturer

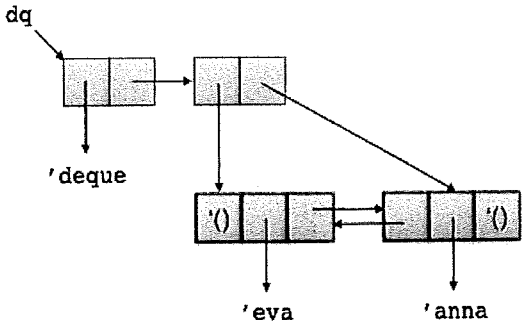
`(define dq (make-deque))` skapar en tom deque:



`(deque-insert-end! dq 'eva)` lägger in ett element i kön:



`(deque-insert-end! dq 'anna)` lägger ytterligare ett element i slutet:



Implementation av deque (exklusive din funktion)

```
#lang racket

;Double ended queue implementation

(print-as-expression #f)

; Elements of a deque will be triples so they can point left
   and right

(define (triple-make left middle right)
  (mcons left (mcons middle right)))

(define (triple-left triple)
  (mcar triple))

(define (triple-middle triple)
  (mcar (mcdr triple)))

(define (triple-right triple)
  (mcdr (mcdr triple)))

(define (triple-set-left! triple element)
  (set-mcar! triple element))

(define (triple-set-middle! triple value)
  (set-mcar! (mcdr triple) value))

(define (triple-set-right! triple element)
  (set-mcdr! (mcdr triple) element))

; Handling double ended queues

(define (make-deque) (mcons 'deque (mcons '() '())))

(define (deque-content dq)
  ;fetch contents of dq
  (mcdr dq))

(define (deque-empty? dq)
  ;deque has no elements?
  (null? (mcar (deque-content  dq))))

(define (deque-singleton? dq)
```

```
  ;deque has only one element?
  (eq? (deque-front dq) (deque-end dq)))

(define (deque-front dq)
  ;fetch first element in the queue
  (mcar (deque-content dq)))

(define (deque-set-front! dq element)
  ;let front of deque point to element
  (set-mcar! (deque-content dq) element))

(define (deque-end dq)
  ;fetch last element in the queue
  (mcdr (deque-content dq)))

(define (deque-set-end! dq element)
  (set-mcdr! (deque-content dq) element))

(define (deque-insert-front! dq value)
  (let ((new-element (triple-make '() value '()))
        (front-element (deque-front dq)))
    (cond ((deque-empty? dq)
           (deque-set-front! dq new-element)
           (deque-set-end! dq new-element))
          (else
           (triple-set-right! new-element front-element)
           (triple-set-left! front-element new-element)
           (deque-set-front! dq new-element)))))

(define (deque-insert-end! dq value)
  ;Din kod som svar till uppgift 4
  )

(define (deque-delete-front! dq)
  ; remove front element and return its value
  (cond ((deque-empty? dq) (error "Deque empty!?"))
        (else
         (let ((value (triple-middle (deque-front dq))))
           (cond ((deque-singleton? dq)
                  (deque-set-front! dq '())
                  (deque-set-end! dq '()))
                 (else
                  (deque-set-front! dq (triple-right (deque-
                     front dq)))
                  (triple-set-left! (deque-front dq) '())))
           value))))
```

```
(define (deque-delete-back! dq)
  ; remove back element and return its value
  (cond ((deque-empty? dq) (error "Deque empty!?"))
        (else
         (let ((value (triple-middle (deque-end dq))))
           (cond ((deque-singleton? dq)
                  (deque-set-front! dq '())
                  (deque-set-end! dq '()))
                 (else
                  (deque-set-end! dq (triple-left (deque-end dq
                     )))
                  (triple-set-right! (deque-end dq) '())))
           value))))

(define (deque->list dq)
  ;convert deque to a list
  (define (iter elements)
    (cond ((null? elements) '())
          (else (cons (triple-middle elements)
                      (iter (triple-right elements))))))
  (if (deque-empty? dq)
      '()
      (iter (deque-front dq))))

;;The following examples show how you use these functions.
;(define deque (make-deque))
;(deque-insert-front! deque 'eva)
;(deque-insert-end! deque 'adam)
;(deque->list deque)                    ; gives (eva adam)
;(deque-insert-front! deque 'helena)
;(deque-insert-end! deque 'carl)
;(deque->list deque)
;(deque-insert-end! deque 'lisa)
;(deque-delete-front! deque)
;(deque-delete-back! deque)
```