Gruppövning 1 — Algoritmkonstruktion och pseudokod

Lösningsförslag

5DV149

2024 LP3

Contents

1 Uppgift 1 — Enkel algoritm

Skatteverket vill ha din hjälp med att skriva en applikation för en person som vill räkna ut sin nettolön. För att beräkna nettolönen dras först skatten bort och sedan eventuell fackföreningsavgift.

Om personen tjänar minst 30000 kr är skatten 45%, i intervallet 20000-–29999 kr är skatten 40%, i intervallet 15000–19999 är skatten 35% och under 15000 är den 30%.

Om personen är fackföreningsansluten ska sedan fackföreningsavgiften på 250 kronor dras bort.

Skriv en algoritm i pseudokod som förklarar hur man går till väga för att räkna ut nettolönen (lönen efter skatt och fackföreningsavgift).

1.1 Lösningsförslag

```
Algorithm salary(gross: Int, is-union-member: Bool)

if gross >= 30000 then
  tax ← 0.45
else if gross >= 20000 then
  tax ← 0.40
else if gross >= 15000 then
  tax ← 0.35
else
  tax ← 0.30

net ← gross * (1 - tax)

if is-union-member then
  net ← net - 250

return net
```

2 Uppgift 2 — Fältalgoritmer

Skriv en algoritm i pseduokod som utformas med hjälp av fältspecifikationen (*array*) i kapitel 5.1 (samma som vi gick igenom på föreläsningen) som hittar **medelvärdet** av talen i ett fält. Ni kan anta att fältet är endimensionellt och har ett tal på varje plats. Ni får bara använda de fältoperationer som ingår i gränsytan till Fält.

```
abstract datatype Array(val, index)
  Create(lo, hi: index) → Array(val, index)
  Low(a: Array(val, index)) → index
  High(a: Array(val, index)) → index
  Set-value(i: index, v: val, a: Array(val, index))
                                  → Array(val, index)
  Has-value(i: index, a: Array(val, index)) → Bool
  Inspect-value(i: index, a: Array(val, index)) → val
  Kill(a: Array(val, index)) → ()
```

2.1 Lösningsförslag

```
Algoritm average(a: Array)

n ← High(a) - Low(a) + 1

sum ← 0
for i from Low(a) to High(a) do
  sum ← sum + Inspect-value(i, a)

return sum / n
```

3 Uppgift 3 — Listalgoritmer

Dessa uppgifter använder definitionerna av datatyperna Lista (`List`) och Riktad lista (`DList`). Gränsytan för dessa är:

```
abstract datatype List(val)
auxiliary pos
  Empty() → List(val)
  Isempty(l: List(val)) → Bool
  First(l: List(val)) → pos
  End(l: List(val)) → pos
  Next(p: pos, l: List(val)) → pos
  Previous(p: pos, l: List(val)) → pos
  Pos-isequal(p1, p2: pos, l: List(val)) → Bool
  Inspect(p: pos, l: List(val)) → val
  Insert(v: val, p: pos, l: List(val))
                                → (List(val), pos)
  Remove(p: pos, l: List(val)) → (List(val), pos)
  Kill(l: List(val)) → ()

abstract datatype DList(val)
auxiliary pos
  Empty() → DList(val)
  Isempty(l: DList(val)) → Bool
  First(l: DList(val)) → pos
  Next(p: pos, l: DList(val)) → pos
  Isend(p: pos, l: DList(val)) → Bool
  Inspect(p: pos, l: DList(val)) → val
  Insert(v: val, p: pos, l: DList(val))
                                → (DList(val), pos)
  Remove(p: pos, l: DList(val)) → (DList(val), pos)
  Kill(l: DList(val)) → ()
```

3.1 Uppgift 3a — Max av Lista

1. Skriv en algoritm i pseduokod som utformas med hjälp av listspecifikationen i kapitel 3.2 (samma som vi gick igenom på föreläsningen) som hittar det största talet i en lista av tal. Ni får bara använda de listoperationer som ingår i gränsytan till lista. Om listan är tom ska värdet `Not-A-Number` returneras. Ni får anta att operationen *mindre-än* (`Is-less-than(a, b)`) är definierad för värdetypen.

   - Lösningsförslag

```
Algorithm list-max(l: List)

if Isempty(l) then
  return Not-A-Number

pos ← First(l)
max ← Inspect(pos,l)

while not (Positions-are-equal(l, pos, End(l)) do
  if Is-less-than(max, Inspect(pos,l)) then
    max ← Inspect(pos,l)
  pos ← Next(pos,l)

return max
```

2. Vad har lösningen för relativ förenklad tidskomplexitet?

   - $O(n)$.

3.2 Uppgift 3b — Kopiera listor

1. Skriv en algoritm som skapar en *rak* kopia av en Lista l, dvs. en lista där ordningen på elementvärdena bevaras. Du får bara använda funktioner i gränsytan till Lista.

   - Lösningsförslag:
     - Traversera in-listan framifrån och sätt in varje värde *sist* i ut-listan.

```
Algorithm List-copy(l: List)
// Create and return a copy of the input list. The order of the elements
// is maintained in the copy.

// Create empty output list.
c ← Empty()

// Use p to traverse l.
p ← First(l)

while not (Positions-are-equal(l, p, End(l)) do
  // Get value to insert.
  v ← Inspect(p, l)

  // Insert the value LAST in the output list. This will maintain the order
  // of the element values.
  // We do not need to returned position, so do not collect it.
  c ← Insert(v, End(c), c)

  // Advance in input list.
  p ← Next(p,l)

// Return the copied list.
return c
```

2. Skriv en algoritm som skapar en *omvänd* kopia av en Lista l, dvs. där elementvärdena kommer i motsatt ordning (”baklänges”). Du får bara använda funktioner i gränsytan till Lista.

   - Lösningsförslag:
     - Traversera in-listan framifrån och sätt in varje värde *först* i ut-listan. Det kommer att vända på ordningen av elementvärdena.

```
Algorithm List-reverse-copy(l: List)
// Create and return a reverse copy of the input list. The order of the elements
// in the copy is reversed compared to the input.

// Create empty output list.
c ← Empty()

// Use p to traverse l.
p ← First(l)

while not (Positions-are-equal(l, p, End(l)) do
  // Get value to insert.
  v ← Inspect(p, l)

  // Insert this value FIRST in the output list. This will reverse the order
  // of the element values.
  // We do not need to returned position, so do not collect it.
  c ← Insert(v, First(c), c)

  // Advance in input list.
  p ← Next(p,l)

// Return the copied list.
return c
```

3. Skriv en algoritm som skapar en *omvänd* kopia av en Riktad lista a. Du får bara använda funktioner i gränsytan till Riktad lista. b. Lösningsförslag:

   - Den här lösningen är nästan identisk med den för Lista. Enda skillnaden är att vi använder `Isend()` is stället för `Pos-isequal`.

```
Algorithm DList-reverse-copy(l: DList)
// Create and return a reverse copy of the input list. The order of the elements
// in the copy is reversed compared to the input.

// Create empty output list.
c ← Empty()

// Use p to traverse l.
p ← First(l)

while not Isend(p, l) do
  // Get value to insert.
  v ← Inspect(p, l)

  // Insert this value FIRST in the output list. This will reverse the order
  // of the element values.
  // We do not need to returned position, so do not collect it.
  c ← Insert(v, First(c), c)

  // Advance in input list.
  p ← Next(p,l)

// Return the copied list.
return c
```

4. Skriv en algoritm som skapar en *rak* kopia av en Riktad lista l. Du får bara använda funktioner i gränsytan till Riktad lista. Not: Denna uppgift är lite svårare än de tidigare.

   - Lösningsförslag:
     - Datatypen Riktad lista saknar funktionen `End()` som vi använde i motsvarande funktion för Lista. Vi måste därför använda en position `q` i ut-listan `c` för att hålla koll på slutet av `c`. Notera följande:
       (a) Om positionen `q` innehåller slutet på listan `c` *före* anropet till `Insert()`, så kommer `q` att vara ogiltig *efter* anropet.
       (b) Om vi tilldelar `q` positionen som returneras av `Insert()` så får vi en giltig position, men den refererar till det nyligen insatta elementet och inte till slutet av listan.
       (c) För att uppdatera `q` till att referera till slutet räcker det med att göra `Next()` efter `Insert()`.

```
Algorithm DList-reverse-copy(l: DList)
// Create and return a copy of the input list. The order of the elements
// is maintained in the copy.

// Create empty output list.
c ← Empty()
// Use q to keep track of where, i.e. the End, to insert elements in c.
q ← First(c)

// Use p to traverse l.
p ← First(l)

while not Isend(p, l) do
  // Get value to insert.
  v ← Inspect(p, l)

  // Insert this value at q in the output list. q will always refer to the end of c,
  // so the order will be maintained. The position q is invalidated by the Insert()
  // operation. However, we can assign q to the positioned returned by Insert().
  (c, q) ← Insert(v, q, c)

  // The returned position refers to the position of the recently inserted element
  // which will be the last element in the list. We want to insert the next element
  // in the position AFTER, so we need to advance q in the output list.
  q ← Next(q, c)

  // Advance in input list.
  p ← Next(p,l)

// Return the copied list.
return c
```

3.3 Uppgift 3c — Snittmängd (tentauppgift 2018-03-08)

Snittmängden s av två mängder s_1 och s_2 är en mängd som innehåller alla elementvärden e som ingår både i s_1 och i s_2.

1. Ge pseudokod för en algoritm som beräknar snittmängden där datatypen mängd är konstruerad som en (osorterad) Riktad lista. Ni får anta att operationen likhet `Is-equal(a, b)` är definierad för värdetypen.

2. Ge pseudokod för en algoritm som beräknar snittmängden där datatypen mängd är konstruerad som en Riktad lista där elementvärdena är **sorterade** i stigande ordning. Tänk på att s ska vara sorterad på samma sätt. För fulla poäng ska algoritmen ha optimal komplexitet. Ni får anta att operationerna *likhet* (`Is-equal(a, b)`) och *mindre-än* (`Is-less-than(a, b)`) är definierade för värdetypen.

3. Vad har algoritmerna för relativ förenklad tidskomplexitet om du antar att mängderna innehåller ungefär lika många element n?

3.3.1 Lösningsförslag

1. Osorterade listor

```
Algorithm Set-intersection(s1, s2: List)

// Start with empty output list
s ← Empty()

// Iterate over all elements in s1
p1 ← First(s1)
while not Isend(p1, s1) do
  v1 ← Inspect(p1, s1)

  // Search for the value v1 in s2 by iterating over all elements
  found ← false
  p2 ← First(s2)
  while not Isend(p2, s2) and not found do
    v2 ← Inspect(p2, s2)
    if Is-equal(v1, v2) then
      // Found it
      found ← true
    p2 ← Next(p2, s2)

  // Insert into output list if we found the element. Inserting first
  // will produce an output list in the opposite order, which is ok
  // since the list is unsorted.
  if found then
    s ← Insert(v1, First(s), s)
  p1 ← Next(p1, s1)

return s
```

   - Komplexitet: $O(n^2)$ eftersom vi måste kolla alla element i `s2` ($\approx n$) för varje element i `s1` (också $\approx n$).

2. Sorterade listor

```
Algorithm Set-intersection(s1, s2: List)

// Start with empty output list
s ← Empty()
// Where to insert the next output element
p ← First(s)

// Iterate in parallel over s1 and s2
p1 ← First(s1)
p2 ← First(s2)

// While we have elements left in both lists
while not (Isend(p1, s1) or Isend(p2, s2)) do
  v1 ← Inspect(p1, s1)
  v2 ← Inspect(p2, s2)
  if Is-equal(v1, v2) then
    // Value is in both sets, insert into s
    (s, p) ← Insert(v1, p, s)
    // The returned position p is where v1 is.
    // Next time we want to insert *after* v1 to keep s sorted.
    p ← Next(p, s)

    // Advance in both input lists since
    p1 ← Next(p1, s1)
    p2 ← Next(p2, s2)

  else if Is-less-than(v1, v2) then
    // v1 was the smallest element, so advance in s1
    p1 ← Next(p1, s1)
  else
    // v2 was the smallest element, so advance in s2
    p2 ← Next(p2, s2)

return s
```

- Komplexitet: $O(n)$ eftersom vi itererar parallellt i s_1 och s_2. Vi kan göra det eftersom listorna är sorterade. Genom att p_1 och p_2 hela tiden refererar till det minsta elementet i **resten** av respektive lista så behöver vi inte söka igenom hela den andra listan; det räcker med att jämföra med det aktuella elementet i den andra listan.

4 Uppgift 4 — Listor

Rita figurer som visar vad som händer vid **insättning** och **borttagning** ur en

1. dubbellänkad lista med huvud,
2. enkellänkad lista.

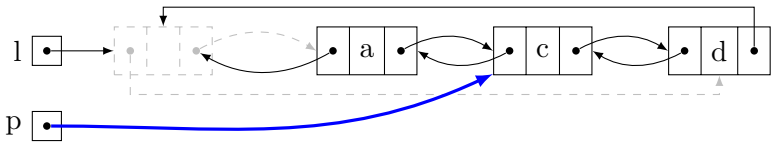
Det är speciellt viktigt att illustrera hur länkar mellan element sätts så att listans integritet bibehålls.

4.1 Lösningsförslag dubbellänkad lista

I exemplen använder jag blå färg för att indikera värden/länkar som **avläses** och röd färg för att indikera värden/länkar som tilldelas.

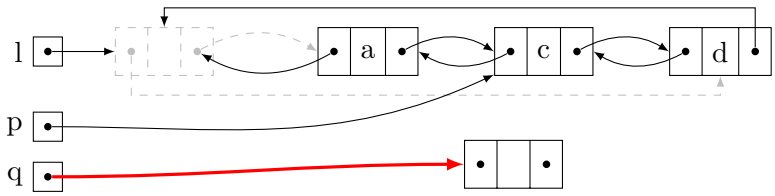
4.1.1 Insättning:

0. Utgångsläge: Vi ska sätta in ett element med värdet 'b' i positionen `p` i listan, dvs. **före** elementet med värdet 'c':

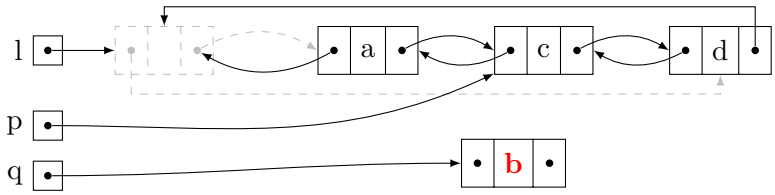


1. Allokera ett nytt element och tilldela värdet:

   (a) `q = malloc(...);`

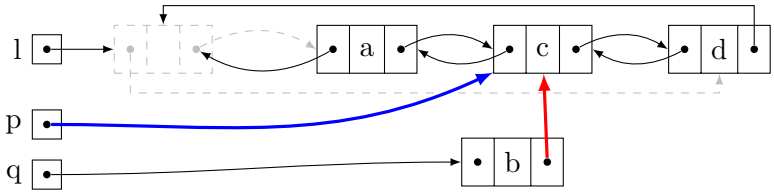


   (b) `q->value = 'b';`

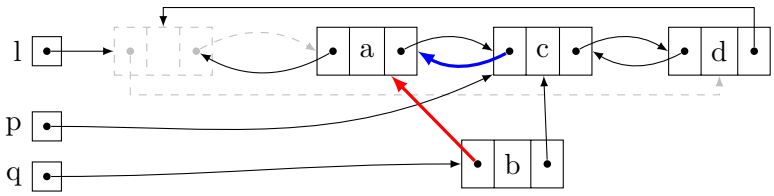


2. Sätt länkarna i det nya elementet:

   (a) `q->next = p;`

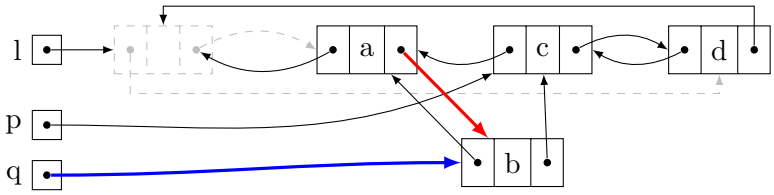


   (b) `q->prev = p->prev;`

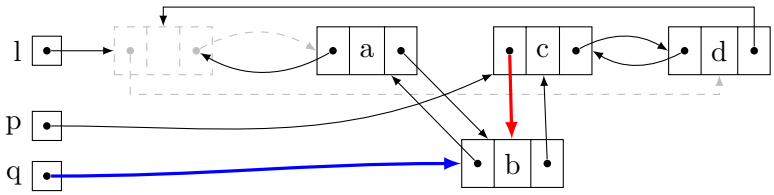


3. Länka in det nya elementet i listan:

   (a) `p->prev->next = q;`

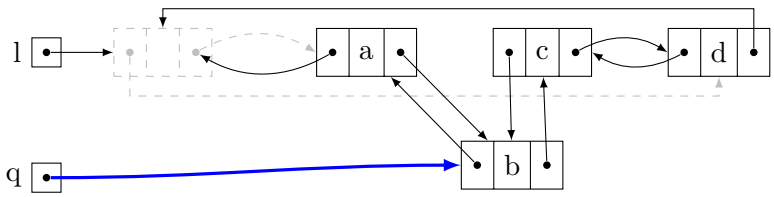


   (b) `p->prev = q;`



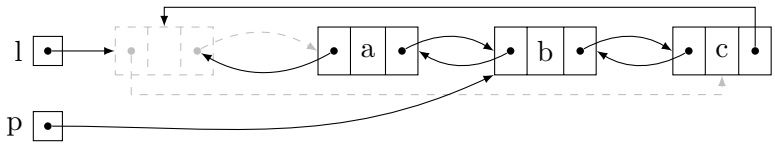
4. Returnera positionen för det ”nya” elementet (med värdet ’b’):

   (a) `return q;`



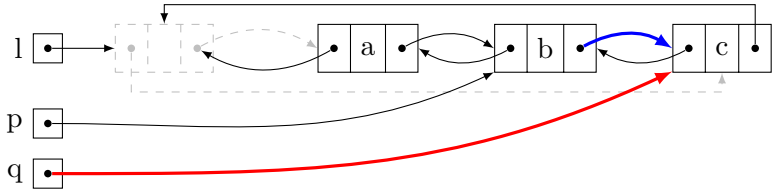
4.1.2 Borttagning:

0. Utgångsläge: Vi ska ta bort elementet på positionen `p`, dvs. elementet med värdet 'b':



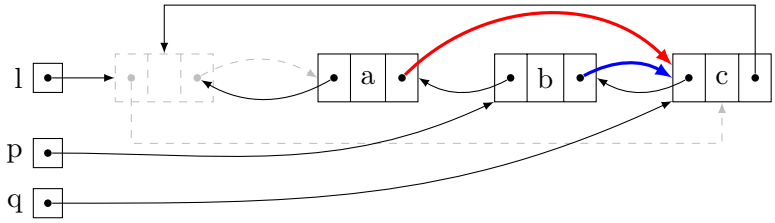
1. Skapa en länk till positionen som ska returneras.

   (a) `q = p->next;`

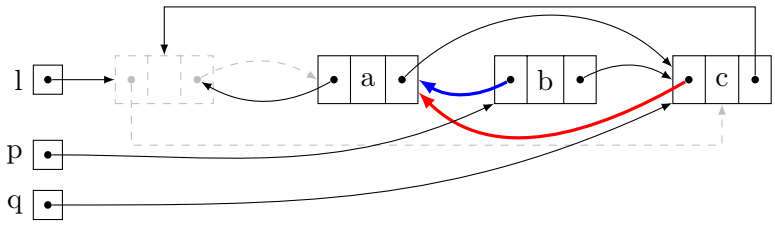


2. Länka förbi elementet som ska tas bort.

   (a) `p->prev->next = p->next;`

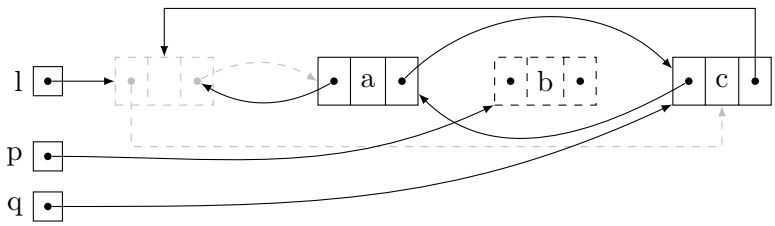


   (b) `p->next->prev = p->prev;`



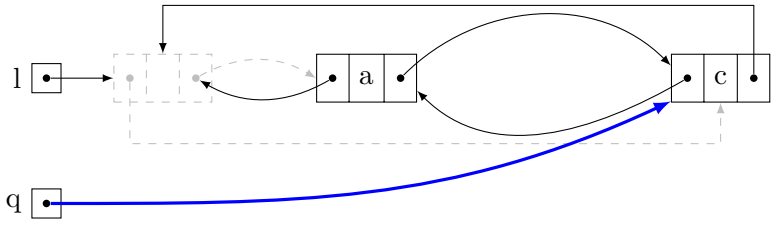
3. Avallokera elementet som ska tas bort.

   (a) `free(p);`



4. Returnera positionen för elementet **efter** det borttagna. (Jämför med utgångsläget.)

   (a) `return q;`

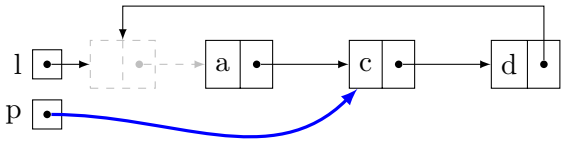


4.2 Lösningsförslag enkellänkad lista, positionen pekar på elementvärdet

Det finns två huvudsakliga sätt att hantera detta. Antingen låter man positionen svara mot en pekare till elementet där värdet **finns**, eller till elementet **innan** det där värdet lagras. Här visar vi exemplet där positionen pekar på elementet **med** värdet.

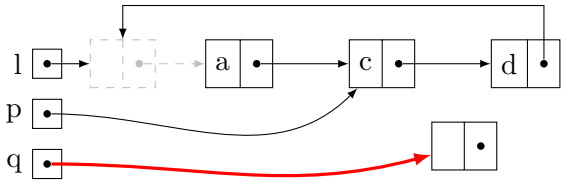
4.2.1 Insättning:

0. Utgångsläge: Vi ska sätta in ett element med värdet 'b' i listan **före** elementet med värdet 'c':

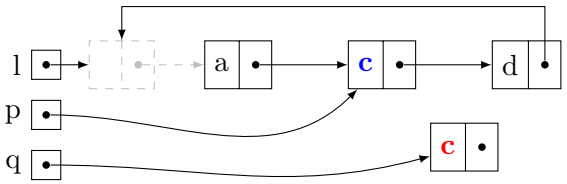


1. Allokering av nytt element, kopiering och tilldelning av värdet:

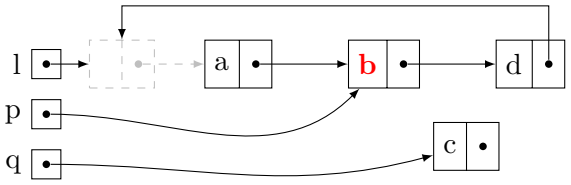
   (a) `q = malloc(...);`



   (b) `q->value = p->value;`

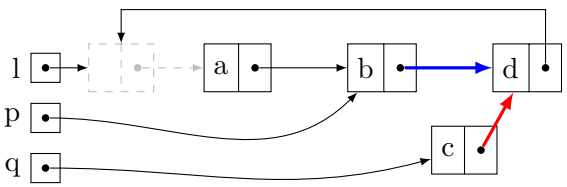


   (c) `p->value = 'b';`



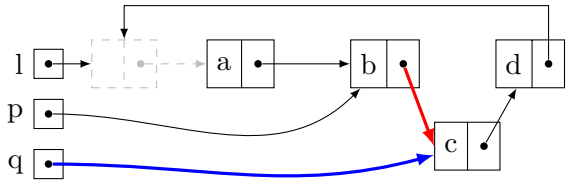
2. Tilldelning av framåtlänk:

   (a) `q->next = p->next;`



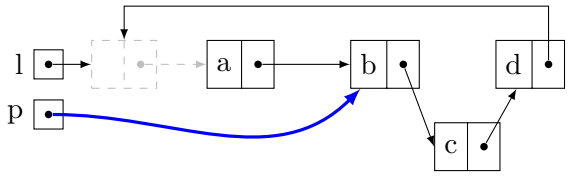
3. Inlänkning i listan:

   (a) `p->next = q;`



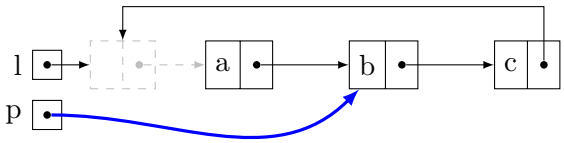
4. Returnera positionen för det ”nya” elementet (med värdet ’b’):

   (a) `return p;`



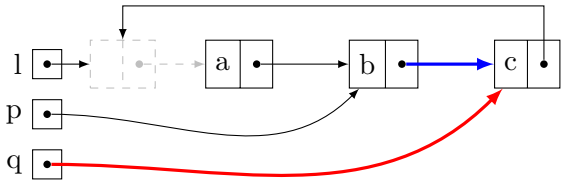
4.2.2 Borttagning:

0. Utgångsläge: Vi ska ta bort elementet på positionen `p`, dvs. elementet med värdet 'b' och returnera positionen för elementet efter det borttagna, dvs. till elementet med värde 'c':

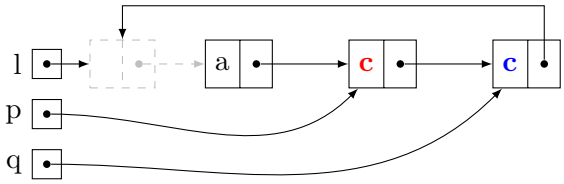


1. Kopiera värdet från nästa element. Skapa länk till elementet efter. Vi behöver den senare.

   (a) `q = p->next;`



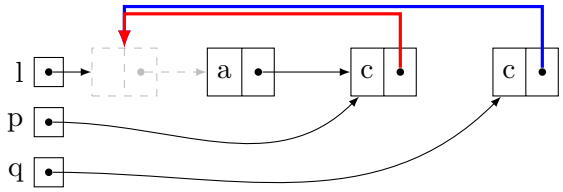
   (b) `p->value = q->value;`



2. Nu när värdena i `p` och `q` är lika kan vi välja vilket element som vi ska ta bort. Vi kommer att ta bort `q`.

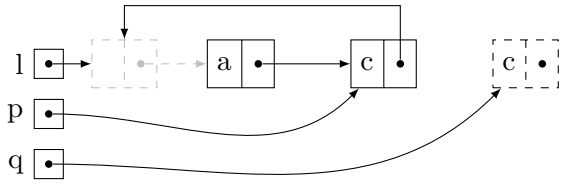
   Länka förbi elementet som ska tas bort.

   (a) `p->next = q->next;`



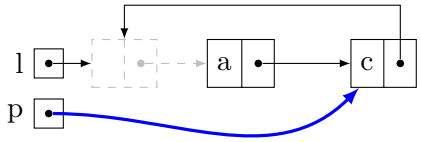
3. Avallokera elementet som ska tas bort.

   (a) `free(q);`



4. Returnera positionen för elementet **efter** det borttagna. (Jämför med utgångsläget.)

   (a) `return p;`

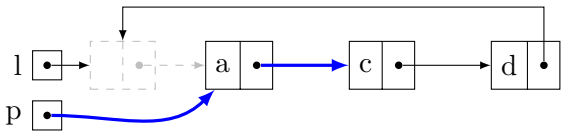


4.3 Lösningsförslag enkellänkad lista, positionen pekar på elementet före

Variant där positionen pekar på elementet **före** elementet med värdet: Obs! Exemplet nedan visar alltså insättning i **samma** position i **samma** lista som i förra exemplet men positionen är implementerad annorlunda!

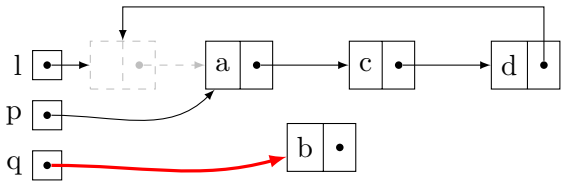
4.3.1 Insättning:

0. Utgångsläge: Vi ska sätta in ett element med värdet 'b' i listan **före** elementet med värdet 'c':

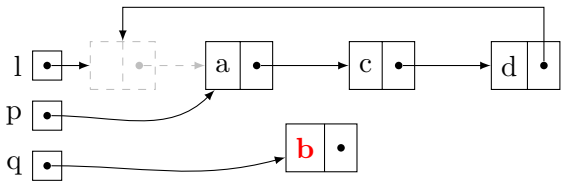


1. Allokering av nytt element och tilldelning av värdet:

   (a) `q = malloc(...);`

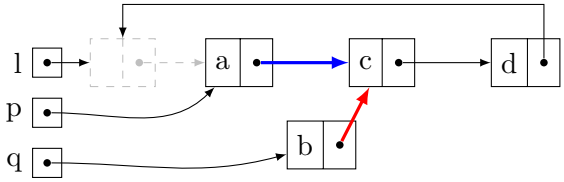


   (b) `q->value = 'b';`



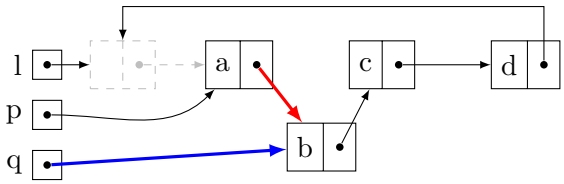
2. Tilldelning av framåtlänk:

   (a) `q->next = p->next;`



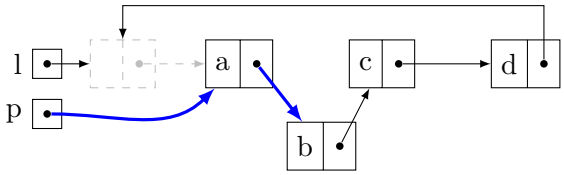
3. Inlänkning i listan:

   (a) `p->next = q;`



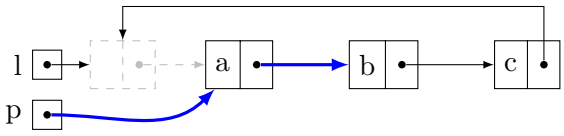
4. Returnera positionen för det ”nya” elementet (med värdet ’b’):

   (a) `return p;`



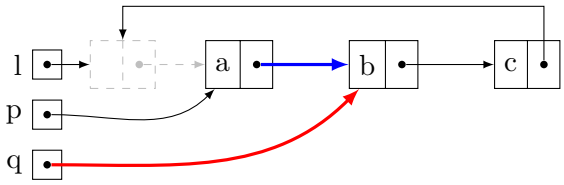
4.3.2 Borttagning:

0. Utgångsläge: Vi ska ta bort elementet på positionen p, dvs. elementet med värdet 'b':



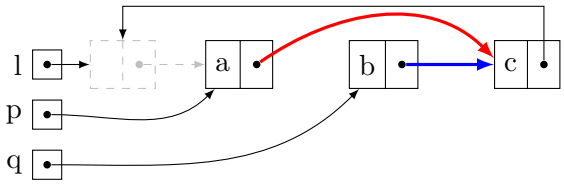
1. Skapa länk till element som ska tas bort. Vi kommer att behöva den senare.

   (a) `q = p->next;`



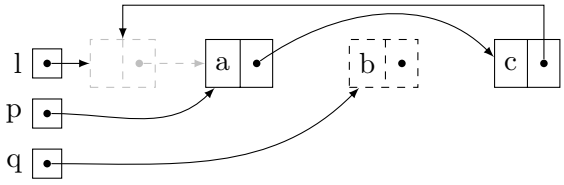
2. Länka runt element som ska tas bort:

   (a) `p->next = q->next;`



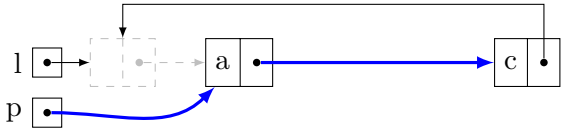
3. Avallokering av borttaget element:

   (a) `free(q);`



4. Returnera positionen för elementet **efter** det borttagna. (Jämför med utgångsläget.)

   (a) `return p;`



5 Uppgift 5 — Konstruera Stack av Lista

Beskriv hur man skulle konstruera en Stack i Lista på så sätt att **sista** elementet i listan är Stackens **topp**. Formulera alla operationerna i gränsytan till Stack med hjälp av gränsytan till Lista. Ange den relativa förenklade komplexiteten för operationerna.

Stack-operation	List-operationer
`Stack-empty()`	`List-empty()`
`Stack-isempty()`	`List-isempty()`
`Stack-push(v, s)`	`List-insert(v, List-end(s), s)`
`Stack-pop(s)`	`List-remove(List-previous(List-end(s)), s)`
`Stack-top(s)`	`List-inspect(List-previous(List-end(s)), s)`

Alla operationer har $O(1)$ i relativ komplexitet.

6 Uppgift 6 — Stack- och Kö-operationer

Hur ser stacken `s` och kön `q` ut efter att följande operationer utförst?

Toppen på stacken är längst till vänster. Front på kön är längst till vänster.

	`s`	`q`
`q <- Queue-empty()`		`[]`
`s <- Stack-empty()`	`[]`	`[]`
`q <- Enqueue(1, q)`	`[]`	`[1]`
`q <- Enqueue(2, q)`	`[]`	`[1, 2]`
`q <- Enqueue(3, q)`	`[]`	`[1, 2, 3]`
`s <- Push(Front(q), s)`	`[1]`	`[1, 2, 3]`
`q <- Dequeue(q)`	`[1]`	`[2, 3]`
`s <- Push(Front(q), s)`	`[2, 1]`	`[2, 3]`
`q <- Dequeue(q)`	`[2, 1]`	`[3]`
`s <- Push(Front(q), s)`	`[3, 2, 1]`	`[3]`
`q <- Dequeue(q)`	`[3, 2, 1]`	`[]`
`q <- Enqueue(Top(s))`	`[3, 2, 1]`	`[3]`
`s <- Pop(s)`	`[2, 1]`	`[3]`
`q <- Enqueue(Top(s))`	`[2, 1]`	`[3, 2]`
`s <- Pop(s)`	`[1]`	`[3, 2]`
`q <- Enqueue(Top(s))`	`[1]`	`[3, 2, 1]`

7 Uppgift 7 — Kö med 1-celler

Antag att du ska konstruera en kö med hjälp av 1-celler. Hur ska riktningen mellan cellerna organiseras, dvs ska fronten på kön på först eller sist i kedjan av celler? Varför?

7.1 Lösningsförslag

Listan är oriktad men är sammansatt med 1-celler. Det betyder att man kan snabbt nå första och sista elementet i listan men att det går mycket fortare att gå ”framåt” i listan än ”bakåt” (eftersom man då måste starta i början och gå fram till elementet före det aktuella.

Det gör fronten på kön bör vara längst fram i listan för då blir komplexiteten för funktionerna bäst.

8 Uppgift 8 — Längd på en Kö

Skriv en algoritm som tar en kö `q` som argument och returnerar **längden** på kön `q`, dvs. antalet element som finns i `q`. Du får bara använda funktioner i gränsytan till Kö. Argumentet `q` skickas *by reference* så när algoritmen är klar så ska `q` se likadan ut som när algoritmen startade.

8.1 Lösningsförslag

Då vi inte kan traversera kön är enda sättet att beräkna antalet element att plocka ut elementen ur kön, ett och ett. Använd en annan kö som mellanlagring och återställ sedan originalkön.

```
Algorithm Queue-length(q: Queue)
// Compute the length of a queue

// This will be a replicate of the queue
r ← Queue-empty()
len ← 0
while not Isempty(q) do
  // Copy first item in queue from q to r
  r ← Enqueue(Front(q), r)
  // Remove from q
  q ← Dequeue(q)
  // Increase count
  len ← len + 1

// Copy all elements from r:
while not Isempty(r) do
  // Copy first item from r to q
  q ← Enqueue(Front(r), q)
  // Remove from r
  r ← Dequeue(r)

return len
```

9 Uppgift 9 — Djupet på en Stack

Skriv en algoritm som tar en stack `s` som argument och returnerar **djupet** på stacken `s`, dvs. antalet element som ligger i `s`. Du får bara använda funktioner i gränsytan till Stack. Argumentet `s` skickas *by reference* så när algoritmen är klar så ska `s` se likadan ut som när algoritmen startade.

9.1 Lösningsförslag

I stort sett identisk algoritm som för kö. En bieffekt av plocka-sönder-sätt-ihop-algoritmen är att ordningen på elementen i stack-kopian blir omvänd. Som ”tur” är så återställs ordningen om vi köra samma plocka-sönder-sätt-ihop-algoritm en gång till.

```
Algorithm Stack-depth(s: Stack)
// Compute the depth of a stack

// This will be a replica of the stack
r ← Stack-empty()
depth ← 0
while not Isempty(s) do
  // Copy first item in stack from s to r
  r ← Push(Top(s), r)
  // Remove from s
  s ← Pop(s)
  // Increase count
  depth ← depth + 1

// Now s is empty. We need to restore it. We can do that from r!
// However, r is upside-down compared to the input. Luckily, running
// the same loop again flips the stack order back again.

// Copy all elements from r:
while not Isempty(r) do
  // Copy first item from r to s
  s ← Push(Top(r), s)
  // Remove from r
  r ← Pop(r)

return depth
```

10 Uppgift 10 — Kopiering av en Kö

Skriv en algoritm som tar en kö `q` som argument och returnerar en **kopia** på `q`. Du får bara använda funktioner i gränsytan till Kö. Argumentet `q` skickas *by reference* så när algoritmen är klar så ska `q` se likadan ut som när algoritmen startade.

10.1 Lösningsförslag

Samma idé som för längden, men nu skapar vi **två** kö-kopior; en som vi kommer att returnera och en som vi använder för att återställa `q`.

```
Algorithm Queue-copy(q: Queue)
// Returns a copy of the input queue

// This will become a replica of the original queue
r ← Queue-empty()
// This will be the proper copy of the queue
c ← Queue-empty()
while not Isempty(q) do
  // Extract and remove first element in the input queue
  v ← Front(q)
  q ← Dequeue(q)
  // Insert element in both queues
  r ← Enqueue(v, r)
  c ← Enqueue(v, c)

// Now q is empty. We need to restore it. We can do that from r!

// Copy all elements from r:
while not Isempty(r) do
  // Copy first item from r to q
  q ← Enqueue(Front(r), q)
  // Remove from r
  r ← Dequeue(r)

// Return the copy of the queue
return c
```

11 Uppgift 11 — Kopiering av en Stack

Skriv en algoritm som tar en Stack `s` som argument och returnerar en **kopia** på `s`. Argumentet `s` skickas *by reference* så när algoritmen är klar så ska `s` se likadan ut som när algoritmen startade.

Återigen i stort sett identisk algoritm som för kö.

```
Algorithm Stack-copy(s: Stack)
// Returns a copy of the input stack

// This will become a replica of the original stack
r ← Stack-empty()
// This will be the proper copy of the stack
c ← Stack-empty()
while not Isempty(s) do
  // Extract and remove top element in the input stack
  v ← Top(s)
  s ← Pop(s)
  // Push element to both stacks
  r ← Push(v, r)
  c ← Push(v, c)

// Now s is empty. We need to restore it. We can do that from r!
// However, r is upside-down compared to the input. Luckily, running
// the same loop again flips the stack order back again.

// Copy all elements from r:
while not Isempty(r) do
  // Copy top value from r to s
  s ← Push(Top(r), s)
  // Remove from r
  r ← Pop(r)

// Return the copy of the stack
return c
```