F02 - Lista, Stack, Testning

5DV149 Datastrukturer och algoritmer
Kapitel 3, 4, 7

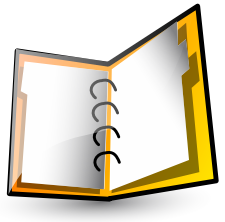
Niclas Börlin
niclas.borlin@cs.umu.se

2024-03-20 Ons

Innehåll

Lista

Lista



- Mental modell: Pärm.
  - Bläddra, läsa, lägga till, ta bort
  - Vi kan lätt ta oss till början eller slutet
  - Vi kan röra oss framåt och bakåt
- Sammansatt datatyp — lagrar element
  - Ändligt antal element
  - Diskret linjärt ordnade
    - Första, sista element
    - Före-efter-relation
  - Homogen datatyp — alla elementvärden är av samma typ
- Generisk datatyp — elementtypen kan vara vad som helst
  - Vi kan t.ex. ha
    - Lista av Heltal,
    - Lista av tecken
    - Lista av (Lista av Heltal)

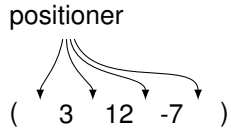
    som alla är abstrakta datatyper
- Dynamisk datatyp
  - Struktur och storlek förändras under datatypens livslängd

Element

- Varje element har två egenskaper
  - Värde Vilket värde som lagras i elementet
  - Position Var i listan som elementet finns
- Exempel:
  - Lista av Heltal (3 12 -7)
    - Första elementet har värdet 3
  - Lista av Tecken ('a' 'x' '!')
    - Första elementet har värdet 'a'
  - Lista av (Lista av Heltal) ((3 1) (9 1 1) (2 0 3 4))
    - Första elementet i har värdet (3 1), som i sin tur har ett första element med värdet 3, osv.
- Listans struktur
  - Oberoende av elementens värden
  - Samtliga exempel ovan är listor med tre element

Position i Lista

- En position är en plats i strukturen
  - Viktigt: För en lista med n element, finns $n+1$ positioner!
  - Den sista positionen i listan är efter det sista elementet!



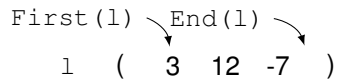
  - En position är bara giltig tillsammans med en lista
    - Den första positionen i en lista har ingen relation till någon position i någon annan lista
- Positionsbeskrivningen förändras när strukturen förändras
  - Positioner som refererar till en lista blir ogiltiga när strukturen förändras (element läggs till eller tas bort)

Gränsyta till Lista (1)

```
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)) → ()
```


Gränsyta till Lista (2)

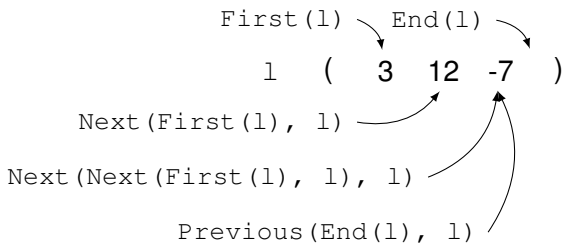
- `Empty()` returnerar en tom lista, dvs en lista utan element
- `Isempty(l)` returnerar `True` om listan `l` är tom
- `First(l)` returnerar den första positionen i listan `l`
- `End(l)` returnerar den sista positionen i listan `l`
- För en icke-tom lista:
  - `First(l)` är positionen för det första elementet
  - `End(l)` är positionen efter det sista elementet



- Specialfall: En tom lista saknar element
  - I en tom lista är den första och sista positionen lika

Gränsyta till Lista (3)

- `Next(p, l)` returnerar positionen som följer efter positionen `p` i listan `l`
- `Next` är odefinierad för positionen `End`
- `Previous(p, l)` returnerar positionen som föregår positionen `p` i listan `l`
- `Previous` är odefinierad för positionen `First`

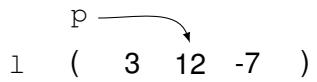


Gränsyta till Lista (4)

- `Pos-isequal(p1, p2, l)` returnerar `True` om positionerna `p1` och `p2` är lika
- Alla jämförelser mellan positioner ska göras med `Pos-isequal`!

Gränsyta till Lista (5)

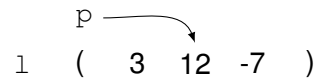
- `Inspect(p, l)` returnerar värdet för elementet med positionen `p` i listan `l`



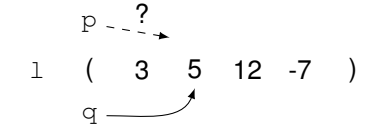
  - `Inspect(p, l)` → 12
- `Inspect` är odefinierad för positionen `End`

Gränsyta till Lista (6)

- `Insert(v, p, l)` sätter in värdet `v` i listan `l` på positionen omedelbart före `p` och returnerar den nya listan samt positionen för det nyinsatta elementet



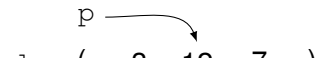
  - `Insert(5, p, l)` → `(l, q)`



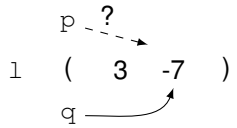
- OBS! Positionen `p` är odefinierad efter anropet

Gränsyta till Lista (7)

- `Remove(p, l)` tar bort elementet på positionen `p` i listan `l` och returnerar den nya listan samt positionen omedelbart efter det borttagna elementet



  - `Remove(p, l)` → `(l, q)`



- Positionen `p` är odefinierad efter anropet
- `Remove` är odefinierad för positionen `End`

Returnera kopia vs modifiera

- Returnerar Insert/Remove en kopia av listan eller modifieras in-listan?

Position kontra Index (1)

- Viktigt: En position är inte detsamma som ett index!
  - Indextypen måste vara diskret linjärt ordnade
  - Positiontypen har inget sådant krav
- Det går att utföra operationer på index utanför ett fält
  - För ett heltalsindex `i` gäller alltid att indexet `j` för elementet närmast efter är
    - `j` ← `i + 1`
    - Vi får ingen varning om `j` blir ett odefinierat index
- En position hör ihop med ”sin” lista
  - För att hitta positionen `j` för nästa element efter `i` i en lista `l` krävs tillgång till listan:
    - `j` ← `Next(i, l)`
    - `Next` är ej definierad om `i` är `End`

Position kontra Index (2)

- Det är lätt att hoppa långt framåt med ett index
  - `j` ← `i + 3` är index för elementet 3 positioner efter `i`
- För en position måste vi beräkna alla mellanliggande positioner
  - `j` ← `Next(Next(Next(i, l), l), l)`

  eller
  - `j` ← `Next(i, l)`
  - `j` ← `Next(j, l)`
  - `j` ← `Next(j, l)`
- Tiden för att komma åt det k:te elementet är alltså
  - oberoende av k för ett Index i ett Fält — det tar samma tid
    - Kallas ibland för *Random access*
  - beroende av k för en Position i en Lista — det tar längre tid för stora k
    - Kallas ibland för *Linear access*

Typ-algoritm Lista (här: List-length)

Pseudokod

```
1  Algorithm Isend(p: Pos, l: List)
2
3  return List-pos-isequal(p, End(l))
4
5
6  Algorithm List-length(l: List)
7
8  len ← 0
9  p ← First(l)
10 while not Isend(p, l) do
11   len ← len + 1
12   p ← Next(p, l)
13
14 return len
```

C-kod

```
bool list_isend(list_pos p, const list *l)
{
  return list_pos_isequal(p, list_end(l), l);
}

int list_length(const list *l)
{
  int len = 0;
  list_pos p = list_first(l);
  while (!list_isend(p, l)) {
    len++;
    p = list_next(l, p);
  }
  return len;
}
```


Konstruktion av lista

- En lista konstrueras vanligen på ett av två olika sätt:
  - Statiskt med hjälp av fält
  - Dynamiskt med hjälp av länkade celler

Men först: Heltal

Gränsyta till ”Heltal”

```
abstract datatype Int
  Create() → Int
  Set-value(i: Int, v: Value) → ()
  Inspect-value(i: Int) → Value
  Kill(i: Int) → ()
```

Informell specifikation av ”Heltal”

- `Create()` — Skapa ett heltal
- `Set-value(i, v)` — Modifiera heltalet `i` genom att sätta det till (heltalsversionen av) värdet `v`
- `Inspect-value(i)` — Läs av värdet i heltalet `i` och returnera en kopia av värdet
- `Kill(i)` — lämnar tillbaka de resurser som heltalet använt

Abstrakt Heltal vs. heltal i C

Pseudokod

```
1  Algorithm main()
2
3  i ← Create()
4  j ← Create()
5
6  Set-value(i, 4)
7  Set-value(j, Inspect-value(i))
8
9  Kill(i)
10 Kill(j)
11 return 0
```

C-kod

```
int main(void)
{
  int i;
  int j;

  i = 4;
  j = i;



  return 0;
}
```


Länk

Länk

- En Länk är ett objekt som refererar till ett annat objekt
  - Kallas ibland pekare eller referens
  - Förekommer som fysiska datatyper i många språk, dvs. inbyggda i språket
- Det är ofta billigare att kopiera länkar än att kopiera objekten själva
- Länkar gör det också möjligt att bygga upp länkade strukturer som listor och träd

Gränsyta till Länk

```
abstract datatype Link(obj)
  Make(x: obj) → Link(obj)
  Nil() → Link(obj)
  Isnil(l: Link(obj)) → Bool
  Follow(l: Link(obj) → obj
  Equal(l1, l2: Link(obj)) → Bool
  Kill(l) → ()
```


Informell specifikation av Länk

- `Make(x)` — Skapa en länk till objektet `x`
- `Nil()` — Returnera den konstanta ”länken till ingenting”
- `Isnil(l)` — Returnera `True` om länken `l` är lika med `Nil()`
- `Follow(l)` — Returnerar objektet som länken `l` refererar till
- `Equal(l1, l2)` — Returnerar `True` om länkarna `l1` och `l2` refererar till samma objekt
- `Kill(a)` — lämnar tillbaka de resurser som länken använt

Abstrakt Länk vs. länk (pekare) i C

Pseudokod

```
1  Algorithm main()
2
3  i ← Int-create()
4  j ← Int-create()
5  p ← Link-make(i)
6
7  Int-set-value(i, 4)
8  Int-set-value(j, Int-inspect-value(Link-follow(p)))
9  Int-set-value(Link-follow(p), 5)
10
11 Link-kill(p)
12 Int-kill(j)
13 Int-kill(i)
14 return 0
```

C-kod

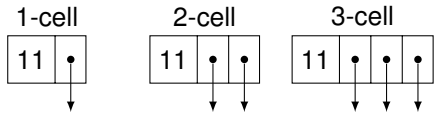
```
int main(void)
{
  int i;
  int j;
  int *p = &i;

  i = 4;
  j = *p;
  *p = 5;



  return 0;
}
```


Blank

Länkade celler

n-Cell

- En *n*-Cell är en Tippel
  - Ett värde (kan vara en länk)
  - *n* stycken länkar



- Vi kommer att använda celler för att bygga länkade strukturer, t.ex. Lista, Träd
- Cellerna göms oftast inuti implementationen av datatypen
- Ibland avslöjar namnet antalet interna länkar
  - ”Dubbel-länkade listor” (*previous*, *next*), ”trippel-länkade träd” (*parent*, *left-child*, *right-child*), osv.

Gränsyta till 1Cell — Cell med en länk

```
abstract datatype 1Cell(val)
  Create() → 1Cell(val)
  Set-value(v: val, c: 1Cell(val)) → 1Cell(val)
  Set-link(l: Link(1Cell(val)), c: 1Cell(val))
                                → 1Cell(val)
  Inspect-value(c: 1Cell(val)) → val
  Inspect-link(c: 1Cell(val)) → Link(1Cell(val))
  Kill(c: 1Cell(val)) → ()
```


Informell specifikation av 1Cell

- `Create()` — Skapa en cell med ett odefinierat värde och en odefinierad länk
- `Set-value(v, c)` — Sätt värdet i cellen `c` till `v`
- `Set-link(l, c)` — Sätt länken i cellen `c` till `l`
- `Inspect-value(c)` — Returnera värdet i cellen `c`
- `Inspect-link(c)` — Returnera länken i cellen `c`
- `Kill(c)` — lämnar tillbaka de resurser som cellen använt

Gränsyta till 2Cell — Cell med två länkar

```
abstract datatype 2Cell(val)
  Create() → 2Cell(val)
  Set-value(v: val, c: 2Cell(val)) → 2Cell(val)
  Set-link1(l: Link(2Cell(val)), c: 2Cell(val))
                                → 2Cell(val)
  Set-link2(l: Link(2Cell(val)), c: 2Cell(val))
                                → 2Cell(val)
  Inspect-value(c: 2Cell(val)) → val
  Inspect-link1(c: 2Cell(val)) → Link(2Cell(val))
  Inspect-link2(c: 2Cell(val)) → Link(2Cell(val))
  Kill(c: 2Cell(val)) → ()
```


Informell specifikation av 2Cell

- Samma som 1Cell men för två länkar
- Ibland kan vi explicit namnge länkarna och funktionerna, t.ex.
  - `Set-next-link()` i stället för `Set-link2()`
  - `Set-previous-link()` i stället för `Set-link1()`
  - `Inspect-next-link()` i stället för `Inspect-link2()`
  - `Inspect-previous-link()` i stället för `Inspect-link1()`

Blank

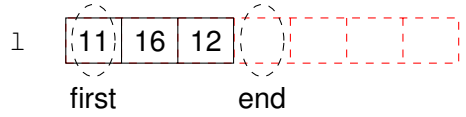
Blank

Konstruktion av Lista (igen)

- En lista konstrueras vanligen på ett av två olika sätt:
  - Statiskt med hjälp av fält
  - Dynamiskt med hjälp av länkade celler
    - Det finns många varianter

Konstruktion av Lista med Fält

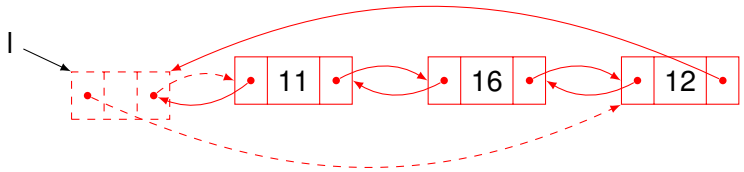
- Lista konstruerad med fält:



  - Antalet upptagna element (här: 3) måste lagras i strukturen
  - De röda elementpositionerna är osynliga för den som använder datatypen
- Indextypen för fältet används som positionstyp för listan
  - `First()` returnerar `Low()`
  - `End()` returnerar `l.last_used_pos + 1`
  - `Next(p, l)` returnerar `p+1`
  - `Previous(p, l)` returnerar `p-1`
  - `Inspect(p, l)` använder `Array-inspect-value(p,l.a)`
- Exempel i kodbasen: `int_list_array`

Konstruktion av Lista med 2-celler (1)

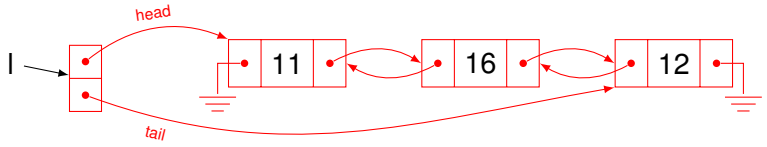
- Lista konstruerad med 2-Cell-huvud och cirkulärt länkade 2-Celler:



  - Allt i rött är oåtkomligt för den som använder datatypen, inkl. cellerna och huvudet
- Positionstypen för `List(val)` är `Link(2Cell(val))`
  - `First()` returnerar `Inspect-next-link(l)`
  - `End()` returnerar `l`
  - `Next(p, l)` returnerar `Inspect-next-link(p)`
  - `Previous(p, l)` returnerar `Inspect-previous-link(p)`
  - `Inspect(p, l)` använder `2Cell-inspect-value(p)`

Konstruktion av Lista med 2-celler (2)

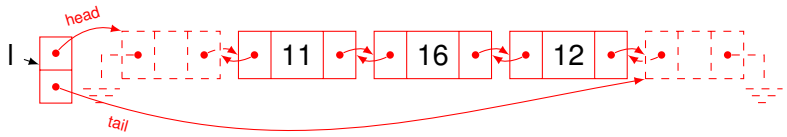
- Lista konstruerad med annan listhuvudstyp och linjärt länkade 2-Celler:



  - Jordsymbolen står för Nil-pekare (`NULL` i C)
- Positionstypen för `List(val)` är `Link(2Cell(val))`
  - `First()` returnerar `l.head`
  - `End()` returnerar `Link-Nil()`
  - `Next(p, l)` returnerar `Inspect-next-link(p)`
  - `Previous(p, l)` returnerar `Inspect-previous-link(p)`
    - `Previous(End(), l)` returnerar `l.tail`
  - `Inspect(p, l)` använder `2Cell-inspect-value(p)`

Konstruktion av Lista med 2-celler (3)

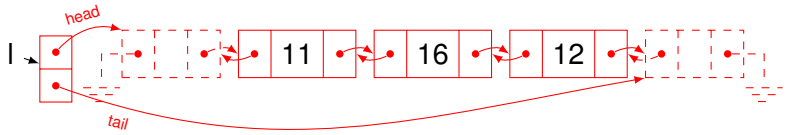
- Lista konstruerad med 2-Cell, annan listhuvudstyp och före-efter-celler:



- Positionstypen för `List(val)` är `Link(2Cell(val))`
  - `First()` returnerar `Inspect-next-link(l.head)`
  - `End()` returnerar `l.tail`
  - `Next(p, l)` returnerar `Inspect-next-link(p)`
  - `Previous(p, l)` returnerar `Inspect-previous-link(p)`
  - `Inspect(p, l)` använder `2Cell-inspect-value(p)`

Lista i kodbasen

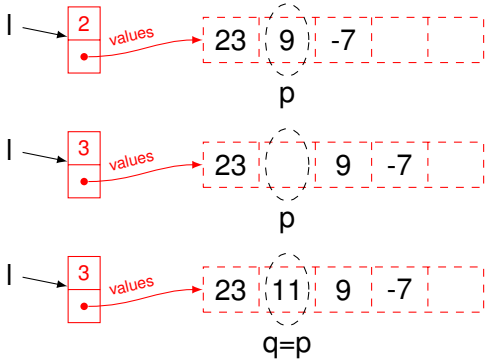
- Implementationerna av (oriktad) Lista i kodbasen använder denna lösning



  - Ovanstående Lista innehåller 3 element
  - Cellerna som representerar head och tail är osynliga för användaren av listan
- `Insert()` sker alltid mellan två celler och saknar därför specialfall

Insert i Lista konstruerad med Fält (1)

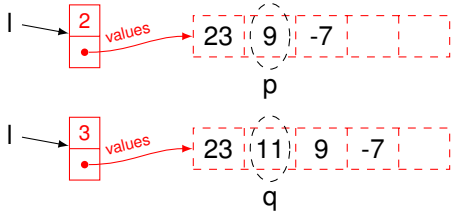
- Vid insättning före positionen `p` (här: 1) måste vi göra plats för det nya elementvärdet genom att flytta alla element efter `p`



- Returnera positionen `q` (här: 1) för det nyligen insatta värdet

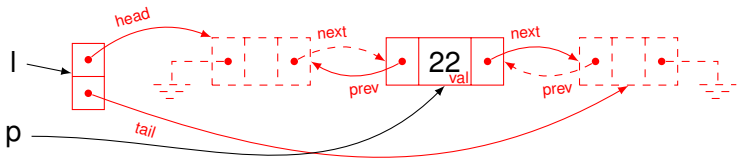
Insert i Lista konstruerad med Fält (2)

- Kom ihåg: Efter insättning är alla gamla positioner ogiltiga!
  - För denna List-konstruktion så refererar positionen `p` efter anropet till `Insert` till samma plats i listan men ett annat elementvärde!



Insert i en länkad struktur (1)

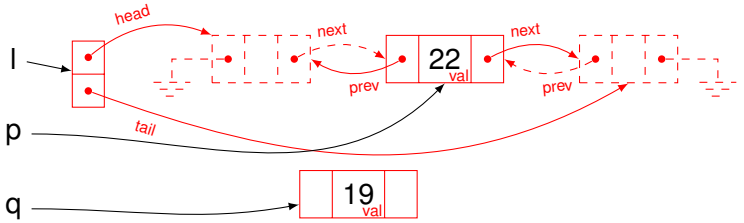
- Listan före insättning



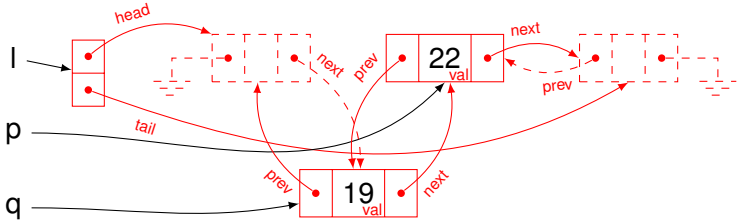
- Vi vill stoppa in värdet `19` före elementet med värde `22`

Insert i en länkad struktur (2)

1. Skapa ett nytt element och sätt värdet till `19`

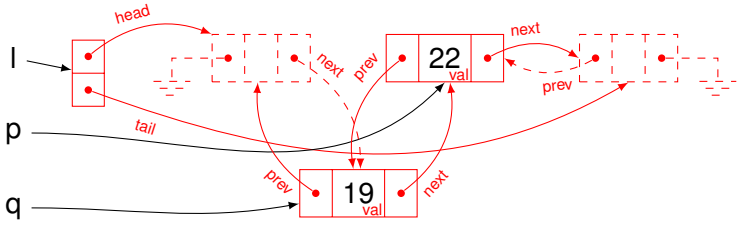


2. Ändra länkarna steg för steg så att det nya elementet hamnar rätt i strukturen



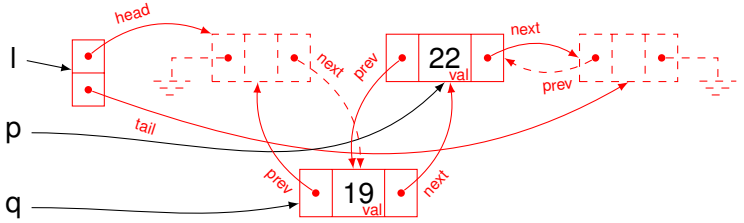
Insert i en länkad struktur (3)

- Returnera positionen `q` för det nyligen insatta värdet



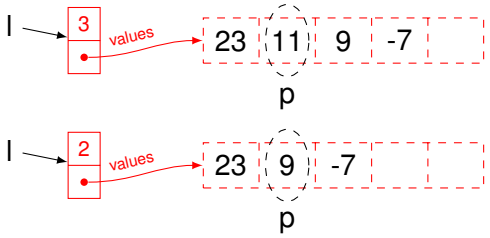
Insert i en länkad struktur (4)

- Jag upprepar: Efter insättning är alla gamla positioner ogiltiga!
  - För denna List-konstruktion så refererar positionen `p` efter anropet till `Insert` till en annan plats i listan men till samma elementvärde!



Remove i Fält-implementation

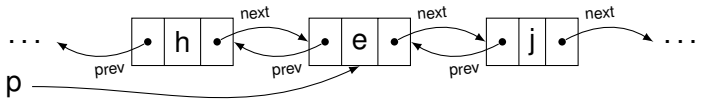
- Vid borttagning av elementet i positionen `p` måste vi flytta efter alla senare element



- Returnera positionen `p` som nu är positionen för elementet omedelbart efter det nyligen raderade elementet

Remove i länkad struktur

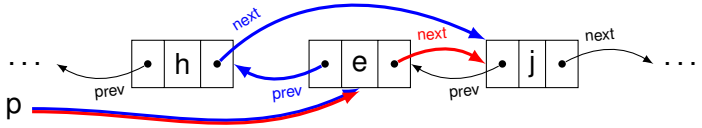
- `Remove` ska ta bort elementet på positionen `p` och returnera positionen omedelbart efter det borttagna elementet



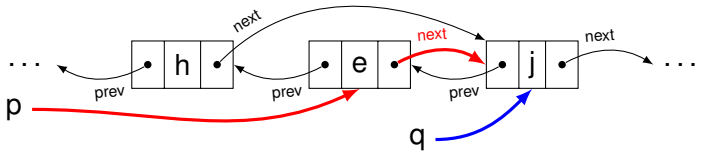
- Vi kommer att använda notationen `p.next` och `p.prev` till att betyda värdet av framåt- resp. bakåt-länken för elementet vars position är `p`
- Hur tar vi bort elementet med värdet *e* utan att tappa bort något i listan?
  - Vi vill att ”h” (korrekt: cellen vars elementvärde är *h*) ska länka framåt till ”j” och att ”j” ska länka bakåt till ”h”

Remove i dubbellänkad lista (1)

- Först fixa ”h”:s framåt-länk:
  `p.prev.next = p.next`

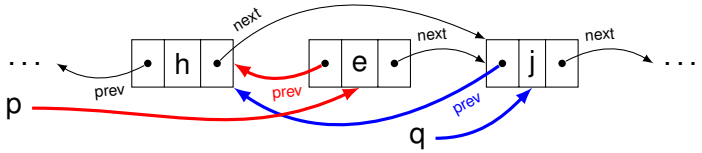


- Skapa en länk som refererar till elementet efter ”e”
  `q = p.next`

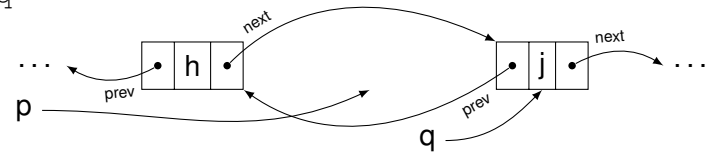


Remove i dubbellänkad lista (2)

- Fixa ”j”:s bakåt-länk
  `q.prev = p.prev`



- Nu är länkarna i listan korrekt och `q` innehåller positionen som ska returneras.
- Vi kan radera elementet ”e” och returnera den nya listan samt `q`



Best practice

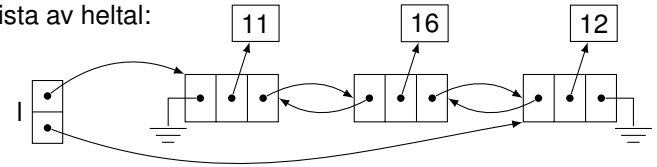
- En strategi som oftast fungerar för att undvika att man använder odefinierade positioner är att
  - använda en positionsvariabel per lista och
  - uppdatera den direkt vid `Insert()` eller `Remove()`:
    - `(p, l)` ← `Insert(v, p, l)`
    - `(p, l)` ← `Remove(p, l)`

Jämförelser

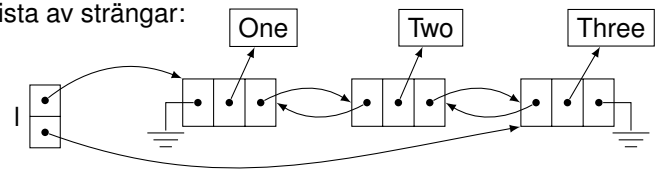
- Lista som Fält
  - Nackdelar:
    - Fast reserverat utrymme
    - Kostsamt sätta in/ta bort element om element måste flyttas
- Lista som länkad struktur
  - Fördelar:
    - Insättning/borttagning går snabbt
    - Minnesutrymmet är proportionellt mot storleken på listan
    - Allokerar minne när det behövs
  - Nackdelar:
    - Länkarna behöver också minnesutrymme

Lista av Länkar (pekare)

- I stället för att lagra värden i cellerna kan vi lagra Länkar till värdena
  - Lista av heltal:



  - Lista av strängar:



- Kodbasen använder denna form för att kunna lagra vilken typ som helst

Konstruktioner Lista med Fält, Fält med Lista, osv.

- Lista kan konstrueras med den abstrakta datatypen Fält
  - Nackdel: Fast utrymme, maxstorlek
- Fält kan konstrueras med den abstrakta datatypen Lista
  - Nackdel: `Inspect-value` måste iterera från början av listan för att hitta till element *k*
- Detta påstående gäller rekursivt
  - Dock måste förstås minst en av Lista och Fält vara implementerad för att datatyperna ska gå att använda

Kodbasen

- Kodbasen innehåller tre olika implementationer av Lista (list)
  - `list` generisk `List(Link(val))`, implementerat med 2-celler i C
  - `int_list` typat för heltal, `List(int)` implementerat med 2-celler i C
  - `int_list_array` typat för heltal, `List(int)` implementerat med fält i C
- Implementationerna `int_list` och `int_list_array` kan vara intressanta att jämföra för att få en grundläggande förståelse av abstrakta datatyper och dynamiskt minne i C
- Kodbasen innehåller flera MWE (*Minimum Working Examples*) för varje datatyp

Blank

Blank

Blank

Abstrakta datatyper — *Stack*

Stack

- Modell: Papperstrave

- Linjärt ordnad struktur
  - Elementen följer en före-efter-relation
- Homogen sammansatt datatyp
- Generisk typ (polytyp)
  - Man kan definiera Stack av heltal, Stack av Lista, osv.
- Kan ses som specialisering av datatypen Lista
  - Begränsningar på operationerna
  - Insättning, borttagning, avläsning alltid i toppen av stacken
  - LIFO — Last In, First Out

Gränsyta för Stack

```
abstract datatype Stack(val)
  Empty() → Stack(val)
  Isempty(s: Stack(val)) → Bool
  Push(v: val, s: Stack(val)) → Stack(val)
  Top(s: Stack(val)) → val
  Pop(s: Stack(val)) → Stack(val)
  Kill(s: Stack(val)) → ()
```


Informell specifikation till Stack

- `Empty()` — skapa en tom stack
- `Isempty(s)` — returnera `True` om stacken `s` är tom
- `Push(v, s)` — lägg ett element med värdet `v` överst på stacken `s` och returnera den modifierade stacken
- `Top(s)` — läs av och returnera värdet på elementet som ligger överst på stacken `s`
- `Pop(s)` — ta bort det översta elementet från stacken `s` och returnera den modifierade stacken
- `Kill(s)` — lämnar tillbaka de resurser som stacken använt

Egenskaper för Stack

- Det går inte att navigera i Stack
  - Det enda elementvärdet vi kan avläsa är det som ligger överst med `Top()`
  - Vill vi avläsa strukturen för Stack, t.ex. hur många element stacken har, så måste vi plocka sönder stacken
    - Vi behöver då troligen sätta ihop den igen

Formell definition

- En uppsättning axiom uttryckta i matematisk logik
- Beskriver relationer mellan typens olika operationer
- Är en mer precis beskrivning av gränsytan
- Axiom kan användas för att göra formella härledningar för datatypen — bevis!
  - Många fall då det inte fungerar i praktiken
- Var tidigare basen för OU1

Formell specifikation till Stack

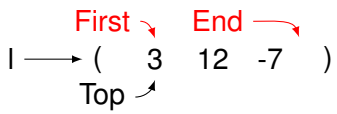
Ax 1	`Isempty(Empty())`	En nyskapad stack är tom
Ax 2	¬ `Isempty(Push(v,s))`	En stack som man lagt ett element på är inte tom
Ax 3	`Pop(Push(v,s)) = s`	Om vi lägger till ett värde på stacken och sen tar bort det översta värdet så blir stacken som innan
Ax 4	`Top(Push(v,s)) = v`	Lägger vi ett värde på stacken så ligger värdet överst på stacken
Ax 5	¬ `Isempty(s)` ⇒ `Push(Top(s),Pop(s)) = s`	Förutsättning: Stacken är inte tom. Om vi tar bort översta elementet och sen lägger tillbaka det så ser stacken ut som innan.

Konstruktioner av Stack

- Stack kan konstrueras med Lista eller Fält
- Vi får olika lösningar beroende på om toppen av stacken ligger först eller sist i listan resp. fältet

Stack konstruerad som Lista

- Toppen av stacken = början av listan



- Uteslutningar och specialiseringar av operationer

Stack-funktionen	konstrueras som
`Stack-Empty()`	`List-Empty()`
`Stack-IsEmpty(s)`	`List-IsEmpty(s)`
`Stack-Top(s)`	`List-Inspect(s, List-First(s))`
`Stack-Pop(s)`	`List-Remove(s, List-First(s))`
`Stack-Push(v,s)`	`List-Insert(s, v, List-First(s))`

- Varje stack-operation kräver alltså 1 eller 2 list-operationer

Komplexitet: Ordobegreppet

- Ett sätt att förenklat beskriva hur mycket tid eller utrymme en algoritm kräver i förhållande till datamängdens storlek
- $O(1)$ innebär att algoritmen tar lika lång tid oavsett antalet element
  - T.ex. att avgöra om en lista är tom eller ej
  - Förenklat: Ingen loop i algoritmen
- $O(n)$ innebär att tiden växer linjärt med antalet element
  - Dubbelt så många element innebär dubbelt så lång tid
  - T.ex. att starta i början av en lista och söka efter största elementet
  - Förenklat: Algoritmer med en loop

Relativ och absolut komplexitet

- Relativ komplexitet
  - Tittar bara på ytan dvs. hur många list- eller fält-operationer som behövs per stack-operation
- Absolut komplexitet
  - Multiplicerar alla relativa komplexiteter ned till fysiska datatyper.
  - Dvs. tittar även på hur listan/fältet är konstruerad och implementerad

Stack konstruerad som Lista

Relativ komplexitet

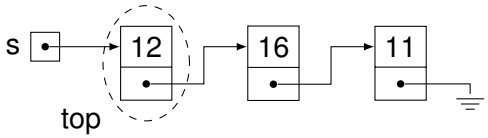
Stack-funktionen	konstrueras som
`Stack-Empty()`	`List-Empty()`
`Stack-IsEmpty(s)`	`List-IsEmpty(s)`
`Stack-Top(s)`	`List-Inspect(s, List-First(s))`
`Stack-Pop(s)`	`List-Remove(s, List-First(s))`
`Stack-Push(v,s)`	`List-Insert(s, v, List-First(s))`

- Varje stack-funktion kräver 1–2 anrop till listan oavsett antal element i stacken
  - Alla stackoperationer har en relativ komplexitet på $O(1)$
- Men vad är komplexiteten hos list-operationerna?
  - Det beror på listans konstruktion och implementation!

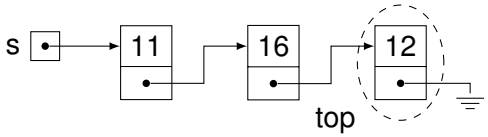
Stack konstruerad som Lista

Absolut komplexitet för Lista konstruerad som Riktad lista med 1-Celler

- Toppen i början av listan: Pop(), Push() och Top() är $O(1)$.



- Toppen i slutet: Pop(), Push(), Top() är $O(n)$
  - Listan är riktad, vi måste traversera alla n elementen för att komma till toppen (slutet)

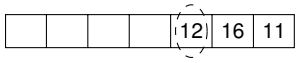


- Toppen i början är bättre: den absoluta komplexiteten blir $O(1)$ för alla stack-operationer!

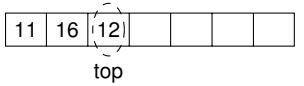
Stack konstruerad som Fält

Absolut komplexitet för Lista konstruerad som Fält

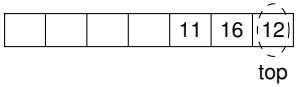
- Fält 1: Botten av stacken i slutet av fältet.



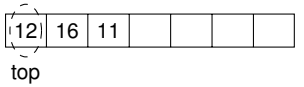
- Fält 2: Botten av stacken i början av fältet.



- Fält 3: Toppen av stacken i slutet av fältet.



- Fält 4: Toppen av stacken i början av fältet.



- Om toppen ligger inåt i fältet (Fält 1+2) så blir alla operationer $O(1)$
  - Vi kan t.o.m. ha två stackar i samma fält
- Om toppen ligger ytterst i fältet (Fält 3+4) kräver Push() och Pop() omflyttningar av data, alltså $O(n)$

Stack, tillämpningar

- Avbryter bearbetning som senare återupptas
- Återspårning (*backtracking*):
  - Till senaste gjorda valet
  - Rekursion (återhoppsadresser)
- Traversera i andra datatyper (grafer och träd).
- Web-läsare: lagra webadresser som nyligen besökts
  - Back => plocka från stacken
- Text-editorers Undo-kommando (Ctrl-Z)
- Evaluering av uttryck

Blank

Modifiera kontra kopiera (1)

- Det finns två viktiga frågor associerade med tolkningen av anropet `t=Push(v,s)`
  - Refererar `t` till samma stack som `s` eller till en kopia av `s`?
  - Modifieras `s` av anropet eller inte?
- Det finns tre vanliga tolkningar:
  - Kopiering:
    - `Push` returnerar en kopia av `s` med elementet `v` tillagt
    - Inparametern `s` är oförändrad efter funktionsanropet
    - I detta fall är det nödvändigt att ta hand om returvärdet från `Push`
  - Modifiering:
    - `Push` modifierar stacken `s`
    - Det som returneras är en referens till det modifierade originalet `s`
    - I detta fall behöver man inte ta hand om returvärdet
  - Det finns också en hybrid-version:
    - `Push` returnerar en kopia och konsumerar samtidigt in-stacken, dvs. in-stacken är oanvändbar efter anropet

Modifiera kontra kopiera (2)

- Notera att ”oförändrad” bara betyder att `s` har samma struktur och innehåll efter anropet som före
  - I en del algoritmer är det ofrånkomligt att `s` modifieras i funktionen och sedan återställs innan funktionen avslutas

Modifiera kontra kopiera (3)

- Skillnaden är viktig redan på abstrakt algoritmnivå
  - Sammansatta datatyper saknar kopieringsoperation
  - Det går att definiera en kopieringsoperation med hjälp av funktionerna i gränsytan
- Ännu viktigare på implementationsnivå
- Ibland tvingar språket fram begränsningar
  - Exempelvis så returnerar den abstrakta formen av `List-Insert(v, p, l)` både den nya listan och en position
  - I C-implementationen i kodbasen så returneras bara positionen
    - Listan modifieras
- Utgå från kopieringstolkningen eller hybriden om inget annat sägs tydligt!
  - Ta för vana att alltid ta hand om returvärdet, även om ni inte tänker använda det!
    - `s = Push(v, s)`
    - `p = Insert(v, p, l)`
  - Flera av labbarna kräver detta!

Kodbasen

- Kodbasen innehåller två olika implementationer av Stack

  `stack` generisk stack `Stack(Link(val))`

  `int_stack` typad stack för heltal `Stack(int)`
- Kodbasen innehåller flera MWE (*Minimum Working Examples*) för varje datatyp

Testning

Testning

- Testning är jätteviktigt!
  - Mordechai (1999), The Bug That Destroyed a Rocket", *Journal of Computer Science Education*, vol. 13, no. 2, pp. 15–16.[1]
- Test går att göra under många olika faser i utvecklingen:
  - Problembeskrivning
  - Systemdesign
  - Mjukvarukomponenter — enhetstester
  - Mjukvarulösning — systemtest
  - Post-release
- På denna kurs kommer vi ge en introduktion till testning av mindre mjukvarukomponenter — enhetstestning

[1] https://www.youtube.com/watch?v=gp_D8r-2hwk

Syftet med testning

- Syftet är att hitta och identifiera fel så tidigt som möjligt i utvecklingsprocessen för att minska kostnaderna
- Testning är framför allt en utmaning av fantasin
  1. A QA engineer walks into a bar.
     - Orders a beer.
     - Orders 0 beers.
     - Orders 999999999 beers.
     - Orders a lizard.
     - Orders -1 beers.
     - Orders a sfdeljknesv.

     The bar's first real customer walks in.
     - Asks where the bathroom is.
     - The bar bursts into flame killing everyone.
  2. Mideval Help desk
     https://www.youtube.com/watch?v=N5mLK4V5P30
  3. PBJ sandwich
     https://www.youtube.com/watch?v=cDA3_5982h8
  4. Star Trek IV: The Voyage Home
     https://www.youtube.com/watch?v=LkqiDu1BQXY

Enhetstestning

- Mål: Testa en mjukvarukomponent (t.ex. en funktion eller datatyp) isolerat från resten av programmet
  - Övertyga sig om att komponenten fungerar korrekt innan man använder sig av den i ett större program
- Enhetstestet fungerar också som ett kontrakt som koden måste följa
  - Ändrar man i implementationen så måste den nya koden också klara enhetstestet
- Görs ofta av samma programmerare som ska skriva koden
  - Kan vara mentalt lättare om en annan person skriver test-koden
- I utvecklingsmodellen *Test-driven Development (TDD)* [2] så skrivs testet först, sedan koden som ska implementera ny funktionalitet

[2] https://en.wikipedia.org/wiki/Test-driven_development

Testning av en datatyp

- Vad ska vi testa?
  - Börja med de enklaste testerna!
  - Testa gränsfall
  - Viktigt att täcka in alla operationer...
  - ... och att täcka in de olika fall som finns i operationerna
- Testa så lite som möjligt i varje test — hjälper till att identifiera vad som går fel
- Skriv varje test i en egen funktion

Testning av Lista

- Här beskriver jag hur man kan tänka när man ska bygga ett testprogram för en datatyp
- Som exempel kommer jag att använda datatypen Lista
- Den är mer komplicerad än Stack som ni ska implementera ett testprogram för, men resonemanget är ungefär detsamma
- Ni ska implementera två stycken testprogram i OU1, en för en typad Stack och en för en generisk
  - Denna beskrivning kommer att följa samma mönster
- Ni får bara använda funktioner i gränssnittet
  - Det kommer att medföra vissa begränsingar
- OK, så vad vet vi? Se den informella specifikationen av Lista (bild 9–15)

Steg 0 - förberedelser

- Innan vi skriver någon testkod så ska vi implementera en jämförelsefunktion
  - `Value-equal(v1, v2: value)`
    - Funktionen returnerar `True` om värdena `v1` och `v2` är lika
    - Alla jämförelser av värden som lagras i listan måste göras med denna funktion
- Kravet att använda funktionen är till för att hjälpa då det gör det troligare att typfel (`int` vs. `int *`) för testet av den generiska stacken fångas upp av kompilatorn

```
/*
 * Function to compare the values stored in the list.
 */
bool value_equal(int v1, int v2)
{
        return v1 == v2;
}
```


Steg 1 - tom lista (1)

- Vi börjar med den typade listan `List(int)` (`int_list` i kodbasen)
- Vilket är det minsta testet, det som vi ska starta med?
- Listan skapas med `Empty()`...
  - Kan vi testa om `Empty()` gör rätt?
  - Hade vi haft tillstånd att titta "under ytan" och se vad `Empty()` faktiskt gör så hade vi kanske det
  - Men vi får bara använda funktioner i gränssnittet
  - Då är det bästa vi kan göra att testa `Empty()` tillsammans med andra funktioner som testar förväntade egenskaper hos det som `Empty()` returnerar
    - På samma sätt kan vi inte testa `Kill()` — det finns inget sätt att kontrollera om resurserna lämnats tillbaka utan att titta under ytan

Steg 1 - tom lista (2)

- Ok, vad vet vi om listan som `Empty()` borde returnera?
  1. `Empty()` bör returnera en lista
  2. Listan borde vara tom
  3. First borde vara lika med End
  4. Vi kan inte testa Next, Previous — förutsätter att det finns minst två positioner
  5. Vi kan inte testa Inspect, Remove — förutsätter att det finns minst ett element
  6. Vi kan testa Insert, vilket skulle ge oss en större lista — vi spar den!

Blank

Steg 1 - tom lista (3)

- `Empty()` bör returnera en lista
  - `Empty()` i kodbasen returnerar en pekare till en `struct`
  - Vi vet inte hur structen ser ut inuti, så det kan vi inte testa, men...
  - ...vi vet att om `Empty()` returnerar `NULL` så har något gått fel!
  - Implementera det i en egen funktion!
  - Ge funktionen ett namn som säger något av vad den testar, t.ex. `empty_returns_non_null`

Steg 1 - tom lista - `empty_returns_non_null`

```
1  /*
2   * empty_returns_non_null() - Test that the list_empty() function returns a non-null pointer.
3   * Precondition: None.
4   */
5  void empty_returns_non_null(void)
6  {
7          // Print a starting message
8          fprintf(stderr,"Starting empty_returns_non_null()...");
9
10         // Create an empty list
11         list *l = list_empty();
12
13         // l should be non-NULL
14         if (l == NULL) {
15                 // Fail with error message
16                 fprintf(stderr, "FAIL: Expected a list pointer, got NULL.\n");
17                 exit(EXIT_FAILURE);
18         }
19
20         // Everything went well, clean up
21         fprintf(stderr,"cleaning up...");
22         list_kill(l);
23         fprintf(stderr,"done.\n");
24 }
```

Steg 1 - tom lista (4)

- Listan borde vara tom
  - `Isempty(Empty())` bör vara True
  - Implementera det i en egen funktion
  - Det är okej att anta att tidigare test klarats av, dvs. att `Empty()` inte returnerar NULL
  - Döp funktionen till t.ex. `empty_is_empty`

Steg 1 - tom lista - `empty_is_empty`

```
1  /*
2   * empty_is_empty() - Test that the list_empty() list is empty.
3   * Precondition: list_empty() returns non-null.
4   */
5  void empty_is_empty(void)
6  {
7          // Print a starting message
8          fprintf(stderr,"Starting empty_is_empty()...");
9
10         // Create an empty list
11         list *l = list_empty();
12
13         // The list returned by empty() should be is_empty()
14         if (!list_is_empty(l)) {
15                 // Fail with error message
16                 fprintf(stderr, "FAIL: is_empty(empty()) == false, expected true\n");
17                 exit(EXIT_FAILURE);
18         }
19
20         // Everything went well, clean up
21         fprintf(stderr,"cleaning up...");
22         list_kill(l);
23         fprintf(stderr,"done.\n");
24 }
```


Steg 1 - tom lista (5)

- Test 3 : First borde vara lika med End
  - `First(l)` borde vara lika med `End(l)` på en tom lista
  - Implementera det i en funktion som Test 3!
  - Döp funktionen till t.ex. `empty_first_end`

Steg 1 - tom lista - `empty_first_end`

```
1  /*
2   * empty_first_end() - Test that first() == end() on an empty list.
3   * Precondition: list_is_empty(l) == false.
4   */
5  void empty_first_end(void)
6  {
7          // Print a starting message
8          fprintf(stderr,"Starting empty_first_end()...");
9
10         // Create an empty list
11         list *l = list_empty();
12
13         // first(l) should be == end(l) for an empty list
14         if (!(list_pos_is_equal(l, list_first(l), list_end(l)))) {
15                 // Fail with error message
16                 fprintf(stderr, "FAIL: expected first(l) == end(l), they are not\n");
17                 exit(EXIT_FAILURE);
18         }
19
20         // Everything went well, clean up
21         fprintf(stderr,"cleaning up...");
22         list_kill(l);
23         fprintf(stderr,"done.\n");
24 }
```

Steg 2a - lista med ett element

- Okej, då antar vi att vi är klara med tester för en tom lista
  - Nästa steg bör vara lista med ett element
- Skapas t.ex. genom (finns bara ett ställe att stoppa in)
  1. `l0 <- Empty()`
  2. `(p, l) <- Insert(v, First(l0), l0)`
- Lite senare insåg jag att jag behövde skriva många tester som använder en ett-elements-lista...
  - ...så jag skrev en hjälpfunktion som skapade en sådan:

```
1  /**
2   * create_one_element_list() - Create a list with one element
3   * @v: Value to insert
4   *
5   * Preconditions: list_empty() and list_first() works.
6   *
7   * Returns: A list with one element with the value v.
8   *
9   * The list is created by inserting an element at the first position
10  * of an empty list.
11  */
12 list *create_one_element_list(int v)
13 {
14         // Create the list
15         list *l = list_empty();
16         // Insert an element at the only position in the empty list
17         list_insert(l, v, list_first(l));
18
19         return l;
20 }
```


Steg 2b - lista med ett element

- Vad vet vi nu om en-elements-listanl?
  1. Listan l borde vara icke-tom (`not Isempty(l)`)
  2. First(l) borde vara != End(l)
  3. Den returnerade positionen `p` borde vara `== First(l)` (INTE `== First(l0)`!)
  4. Inspect(p) borde vara == v
     - Testas med `value_equal`
  5. Nu kan vi testa Next...
     - 5.1 Next(p) borde vara != p
     - 5.2 Next(p) borde vara == End(l)
  6. Vi kan också testa Prev...
     - 6.1 Prev(Next(p)) borde vara == p
     - 6.2 OBS! Vi kan inte testa Next(Prev(p))! Varför?
     - 6.3 Next(Prev(End(l))) borde vara == End(l)
- Vi kan också testa att Remove(p, l) är tom

Steg 3 - lista med två element

- Nästa steg bör vara lista med två element
- Hmm, nu börjar alternativen för insättning att bli intressanta...
- Har vi en lista med ett element så finns det två positioner att sätta in i
  1. `l0 <- Empty()`
  2. `(p, l1) <- Insert(v1, First(l0), l0)`
- Alternativen är
  1. `(p, l2) <- Insert(v2, First(l1), l1)`
  2. `(p, l3) <- Insert(v2, End(l1), l1)`

Många steg senare...

- Det blev många tester innan jag var klar (och jag vet att det saknas minst ett test till)
- Till slut såg mitt huvudprogram ut ungefär så här:

```
int main(void)
{
        printf("%s, %s %s: Test program for the typed int_list datatype.\n",
               __FILE__, VERSION, VERSION_DATE);
        printf("Code base version %s.\n\n", CODE_BASE_VERSION);

        empty_returns_non_null();
        empty_is_empty();
        empty_first_end();
        one_element_list_is_nonempty();
        one_element_list_has_first_neq_end();
        insert_first_returns_correct_pos();
        inserted_element_has_correct_value();
        next_does_something();
        one_element_list_next_eq_end();
        previous_does_something();
        ...

        fprintf(stderr,"\nSUCCESS: Implementation passed all tests. "
                "Normal exit.\n");
        return 0;
}
```

- Ni hittar koden i file `int_list_test.c` i källkodskatalogen för `int_list` i kodbasen

. . . men det är inte ert problem

- Lista är en komplicerad datatyp och de flesta testerna gäller positioner, något som inte är relevant för Stack
- Ni bör landa nånstans mellan fem och tio tester
- Av de funktioner jag skrev så är alla tester som berör positioner ointressant
- De som berör tom/icke-tom, att värden har rätt position, osv. är relevanta
  - `empty_returns_non_null();`
  - `empty_is_empty();`
  - `one_element_list_is_nonempty();`
  - `inserted_element_has_correct_value();`
  - `insert_remove_is_empty();`
  - `insert_and_count_forwards();`
  - `n_element_list_insert_end();`

Testprogram för Lista i kodbasen

- Kodbasen innehåller testprogrammet `int_list_test.c`
  - Ni är välkomna att använda det som inspiration till ert testprogram för `int_stack`
- Kodbasen innehåller också två testprogrm `list_test1.c` och `list_test2.c` för den generiska Listan
- Skillnaden är att `list_test1` använder en s.k. *kill function* som automatiskt tar hand om avallokering av elementvärden som lagts in i listan
  - Ger enklare kod för OU1
- Programmet `list_test2` använder inte en kill function, utan där stannar ansvaret för att avallokera dynamiskt minne hos användaren av listan

OU1 (1)

- Testning av stack
  - Ni ska kunna avgöra om en given stack är ”hel” eller ”trasig”, dvs. korrekt eller felaktigt implementerad
- Utgå från den informella specifikationen av Stack
  - Ni får utgå från axiomen om ni vill, men. . .
    - erfarenheten från tidigare år är tveksam. . .
  - Ni är välkomna att ignorera axiomen helt om ni inte förstår dom
    - Fokusera i stället på hur ni vet att en stack ska fungera!
  - Skriv små testfunktioner
  - Anropa funktionerna i ordning från enklast till mest komplex
  - Var tydlig med vad varje funktion antar
  - Det gör inget om ni skriver många funktioner
    - Att vara paranoid när man skriver testprogram är ingen nackdel

OU1 (2)

- Utmaningar:
  - Ni vet inte på vilket sätt de trasiga stackarna är trasiga!
    - Det är realistiskt
  - Det är svårt att jämföra två stackar
    - Stackarna har samma gränsyta, så om `Top()` läser av fel element i en stack så läser den av fel element i den andra också!

OU1 (3)

- Nytt för i år är att ni ska skriva två testprogram
  1. Ett för en C-implementation av Stack av Heltal (`int_stack`)
     - Stack av Heltal är statiskt konstruerad och använder inget dynamiskt minne
     - Hela stacken kopieras vid anrop/funktionsretur
     - Top() returnerar `int`
  2. Ett för en C-implementation av en generisk Stack (`stack`)
     - Stack använder `void *` som datatyp för att uppnå något som liknar typgeneralitet
     - Element som läggs på stacken måste allokeras dynamisk
     - Top() returnerar `void *`, vilket går att konvertera till `int *` som motsvarar Link(int)
- Syfte: Separera de två utmaningarna testning och att använda dynamiskt minne

OU1 (4)

- Nytt är också att specifikationen kräver att ni gör alla jämförelser mellan värden (heltal) i en funktion
  `bool value_equal(int v1, int v2);`
  - Syftet är att hjälpa er med ett vanligt fel för den generiska stacken — att jämföra `void`-pekare i stället för heltal
  - Om jämförelserna alltid görs i funktionen kommer kompilatorn att fånga upp om ni tänker fel
- Utgå från hybridtolkningen av de strukturförändrande operationerna
  - Utgå från att vid t.ex. `t=Push(s, v)` så är `s` oanvändbar efter anropet!
  - Använd `s=Push(s, v)` så kommer det att fungera
- Använd gärna testprogrammen för list-varianterna i kodbasen som inspiration

OU1 (5)

- Vi kommer att ta ert testprogram och kompilera det med ett tiotal implementationer av stackar
  - Några korrekta
  - De flesta trasiga
- Ert testprogram ska avgöra om implementationen följer specifikationen för Stack eller inte
- De trasiga implementationerna kan innehålla
  - Funktioner som inte gör nånting
  - Funktioner som gör annat än vad dom borde
  - Funktioner som refererar till fel element

OU1 (6)

- TIPS!
  - Utmana era kursare!
    - Skriv egna trasiga stackar som deras testprogram får försöka upptäcka!
    - Skriv hela stackar som skiljer sig från standardimplementationen!
  - Be dina kursare göra detsamma!
  - Dela, sprid, stackimplementationerna mellan varandra...
    - ...men inte testprogrammen — dom är en del av uppgiften som ni ska lösa enskilt och lämna in!