

TDDI16 – Föreläsning 2

Abstrakta datatyper, linjära strukturer

Filip Strömbäck

- 1 Abstrakta datatyper (ADT)
- 2 Länkad lista – `list`
- 3 Enkel arrayimplementation
- 4 Bättre arrayimplementation – `vector`
- 5 Mer algoritmkomplexitet
- 6 `vector` – fortsättning
- 7 Sammanfattning

Vad är en ADT?

- Beskriver **vad** en datatyp kan göra...
- ...men inte **hur** den är implementerad
- Mycket likt publika delar av klassdefinitioner i C++

Exempel från kursen:

- Lista
- Stack
- Kö
- Symboltabell
- Graf

ADT lista

En **ordnad** sekvens av element. Har följande operationer:

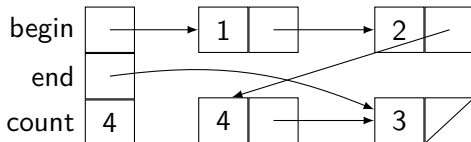
- size() Längden på listan
- empty() Är listan tom?
- elemAt(i) Hämta element på specifik plats
- append(x) Lägg till element sist i listan
- it Någon form av iterator
- insert(it, x) Lägg till element på godtycklig plats
- remove(it) Ta bort ett visst element

Det finns (minst) två implementationer av detta i C++:
vector och list

- 1 Abstrakta datatyper (ADT)
- 2 **Länkad lista – list**
- 3 Enkel arrayimplementation
- 4 Bättre arrayimplementation – vector
- 5 Mer algoritmkomplexitet
- 6 vector – fortsättning
- 7 Sammanfattning

Länkad lista (lik `std::list`, `std::forward_list`)

Idé: Lägg element lite var som helst, spara pekare till nästa element bredvid



Komplexitet:

`elemAt(i)` ?
`append(x)` ?
`insert(it, x)` ?

Notera:

`std::list` är dubbellänkad

`std::forward_list` har bara *begin*

Länkad lista – elemAt

```
void elemAt(int i) {  
    Node *at = begin;  
    for (int x = 0; x < i; x++)  
        at = at->next;  
    return at;  
}
```

Länkad lista – append

```
void append(T x) {  
    Node *n = new Node();  
    n->value = x;  
    n->next = null;  
  
    if (end) {  
        end->next = n;  
        end = n;  
    } else {  
        begin = end = n;  
    }  
}
```

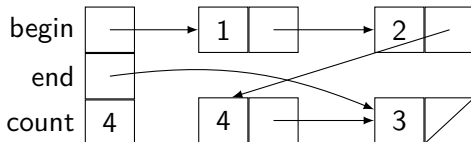

Länkad lista – insert

Notera: Sätter in **efter** element *i*, inte innan.

```
void insert(Node *i, T x) {  
    Node *n = new Node();  
    n->value = x;  
    n->next = i->next;  
  
    i->next = n;  
  
    // div. specialfall  
}
```

Länkad lista (lik `std::list`, `std::forward_list`)

Idé: Lägg element lite var som helst, spara pekare till nästa element bredvid



Komplexitet:

`elemAt(i)` $\mathcal{O}(n)$

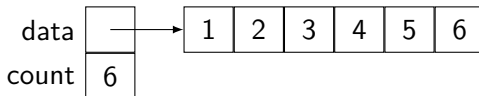
`append(x)` $\mathcal{O}(1)$

`insert(it, x)` $\mathcal{O}(1)$

- 1 Abstrakta datatyper (ADT)
- 2 Länkad lista – `list`
- 3 **Enkel arrayimplementation**
- 4 Bättre arrayimplementation – `vector`
- 5 Mer algoritmkomplexitet
- 6 `vector` – fortsättning
- 7 Sammanfattning

Arrayimplementation

Idé: Lägg elementen efter varandra i minnet



Komplexitet:

elemAt(i) ?

append(x) ?

insert(it, x) ?

Arrayimplementation – elemAt

```
T elemAt(int i) {  
    return data[i];  
}
```

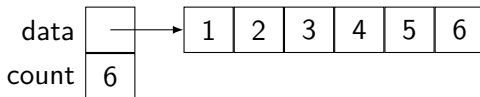
Arrayimplementation – append

```
void append(T x) {  
    T *new_data = new T[count + 1];  
    for (int i = 0; i < count; i++)  
        new_data[i] = data[i];  
  
    new_data[count] = x;  
    delete [] data;  
    data = new_data;  
    count++;  
}
```

Arrayimplementation – insert

```
void insert(int pos, T x) {  
    T *new_data = new T[count + 1];  
    for (int i = 0; i < pos; i++)  
        new_data[i] = data[i];  
  
    new_data[pos] = x;  
  
    for (int i = pos; i < count; i++)  
        new_data[i + 1] = data[i];  
    delete []data;  
    data = new_data; count++;  
}
```

Arrayimplementation – komplexitet



Komplexitet:

elemAt(i) $\mathcal{O}(1)$

append(x) $\mathcal{O}(n)$

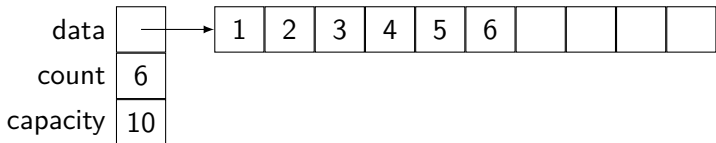
insert(it, x) $\mathcal{O}(n)$

Inte jättebra...

- 1 Abstrakta datatyper (ADT)
- 2 Länkad lista – list
- 3 Enkel arrayimplementation
- 4 **Bättre arrayimplementation – vector**
- 5 Mer algoritmkomplexitet
- 6 vector – fortsättning
- 7 Sammanfattning

Dynamisk array – `std::vector`

Idé: Allokera lite mer minne än vad vi behöver!



Komplexitet:

`elemAt(i)` $\mathcal{O}(1)$

`append(x)` ?

`insert(it, x)` ?

Dynamisk array – append

```
void append(T x) {  
    if (count >= capacity)  
        grow();  
  
    data[count] = x;  
    count++;  
}
```

Dynamisk array – insert

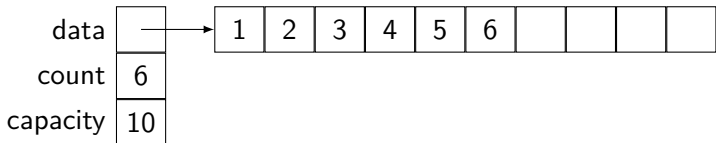
```
void insert(int pos, T x) {  
    if (count >= capacity)  
        grow();  
  
    for (int i = count - 1; i >= pos; i--)  
        data[i + 1] = data[i];  
  
    data[pos] = x;  
    count++;  
}
```

Dynamisk array – grow

```
void grow() {  
    int new_capacity = 2 * capacity;  
    T *new_data = new T[new_capacity];  
  
    for (int i = 0; i < capacity; i++)  
        new_data[i] = data[i];  
  
    data = new_data;  
    capacity = new_capacity;  
}
```

Dynamisk array – `std::vector`

Idé: Allokera lite mer minne än vad vi behöver!



Komplexitet:

`elemAt(i)` $\mathcal{O}(1)$
`append(x)` $\mathcal{O}(n)$
`insert(it, x)` $\mathcal{O}(n)$

Hmmm... inte bättre...

- 1 Abstrakta datatyper (ADT)
- 2 Länkad lista – `list`
- 3 Enkel arrayimplementation
- 4 Bättre arrayimplementation – `vector`
- 5 Mer algoritmkomplexitet**
- 6 `vector` – fortsättning
- 7 Sammanfattning

Ordo, Theta och Omega

Vi kan vara ännu mer specifika:

- $f \in \mathcal{O}(g)$: f växer inte **snabbare** än g
- $f \in \Omega(g)$: f växer inte **långsammare** än g
- $f \in \Theta(g)$: f växer **lika snabbt** som g

Ytterligare gränser

$\mathcal{O}$, Θ och Ω fungerar ungefär som $\leq$, $\approx$ och $\geq$:

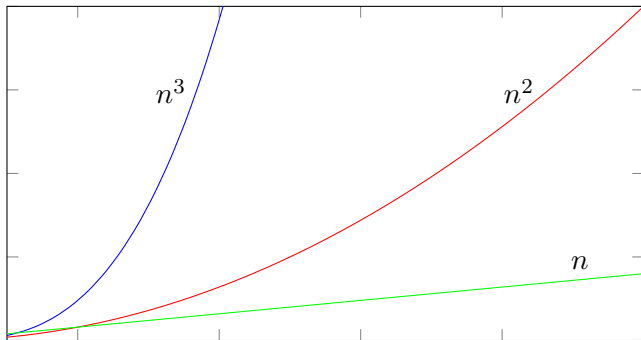
- $f \in \mathcal{O}(g)$, tänk " $f \leq g$ "
- $f \in \Omega(g)$, tänk " $f \geq g$ "

$$f \in \Omega(g) \iff g \in \mathcal{O}(f)$$

- $f \in \Theta(g)$, tänk " $f \approx g$ "

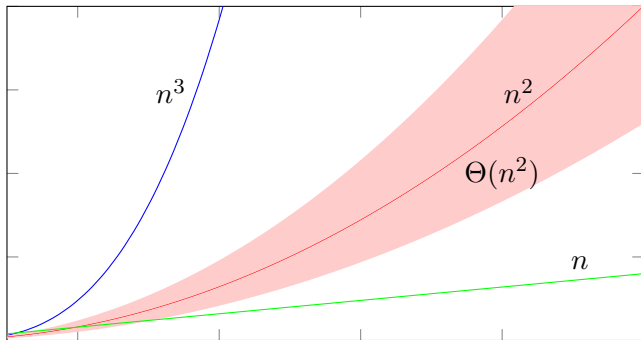
$$\begin{aligned} f \in \Theta(g) &\iff f \in \mathcal{O}(g) \wedge g \in \Omega(f) \\ &\iff f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(f) \end{aligned}$$

Relationer



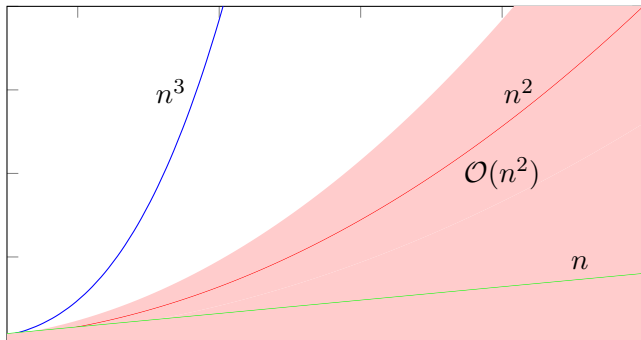
Vi vill beskriva $t(n) = 3n^2 + 2n$

Relationer



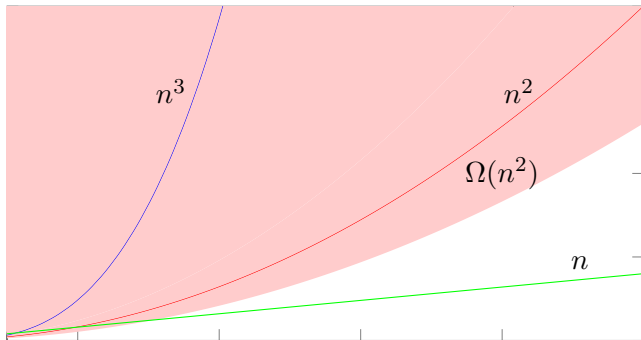
Vi vill beskriva $t(n) = 3n^2 + 2n$

Relationer



Vi vill beskriva $t(n) = 3n^2 + 2n$

Relationer



Vi vill beskriva $t(n) = 3n^2 + 2n$

Bästa- och värstafallsanalys

Vad är tidskomplexiteten för append?

- $\mathcal{O}(n)$ är korrekt, men säger det hela sanningen?
- Många gånger går det snabbare, hur vet vi det?

Bästa- och värstafallsanalys

Vad är tidskomplexiteten för append?

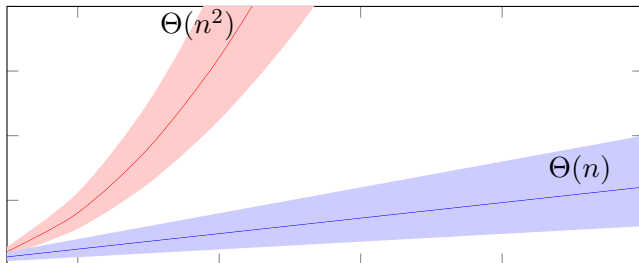
- $\mathcal{O}(n)$ är korrekt, men säger det hela sanningen?
- Många gånger går det snabbare, hur vet vi det?

Separat analys av bästa och värsta fallen:

- Bästa fallet: $\Theta(1)$
- Värsta fallet: $\Theta(n)$

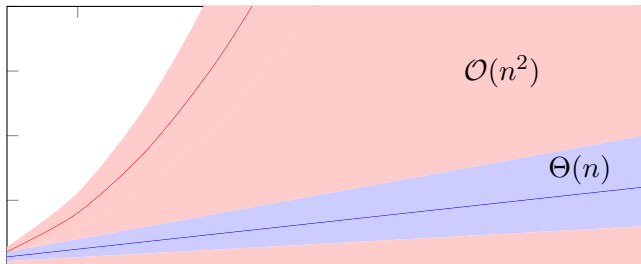
Exempel

Bästa fall: $\Theta(n)$, värsta fall: $\Theta(n^2)$



Exempel

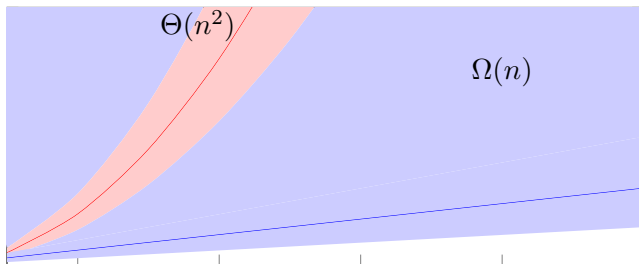
Bästa fall: $\Theta(n)$, värsta fall: $\Theta(n^2)$



Totalt: $\mathcal{O}(n^2)$

Exempel

Bästa fall: $\Theta(n)$, värsta fall: $\Theta(n^2)$



Totalt: $\Omega(n)$

append-funktionen

Bästa fall: $\Theta(1)$

Värsta fall: $\Theta(n)$

Sammantaget: $\mathcal{O}(n)$, $\Omega(1)$

Alla stämmer. Vilka är mest användbara?

Kan vi säga något mer?

Vi testar!

```
void insert(int n) {  
    vector<int> x;  
  
    for (int i = 0; i < n; i++)  
        x.push_back(i);  
}
```

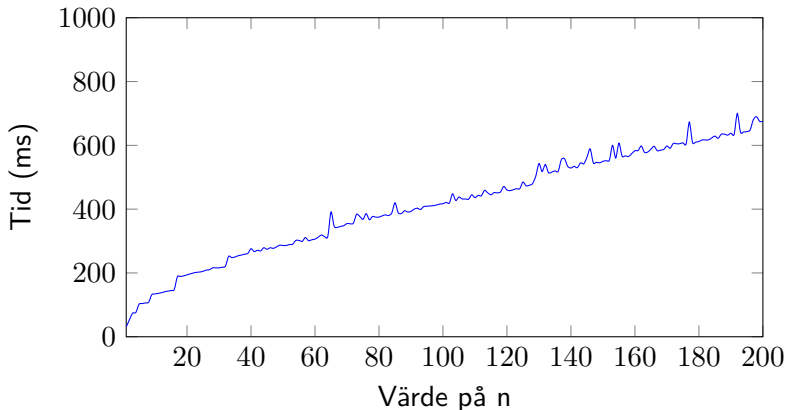
Total komplexitet?

Vi testar!

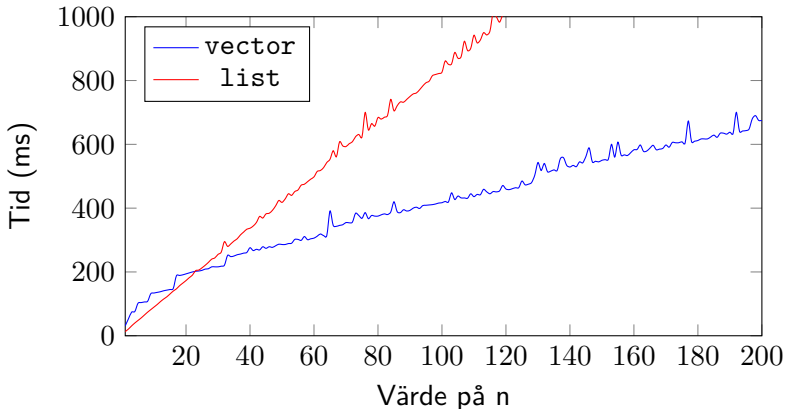
```
void insert(int n) {  
    vector<int> x;  
  
    for (int i = 0; i < n; i++)  
        x.push_back(i);  
}
```

Total komplexitet? $\mathcal{O}(n^2)$?

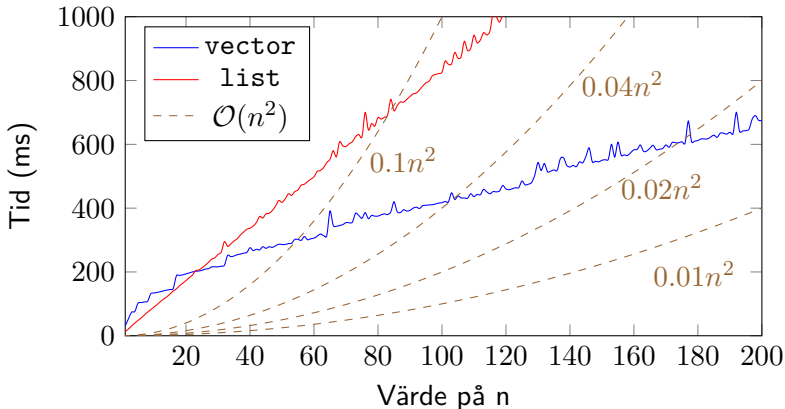
Vi testar! 100 000 körningar per indata



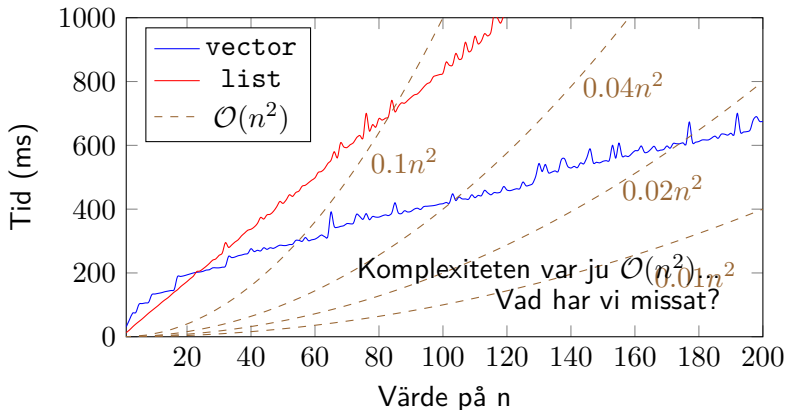
Vi testar! 100 000 körningar per indata



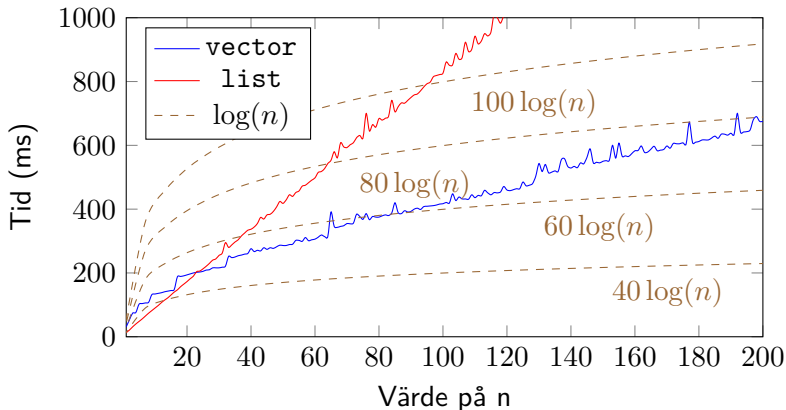
Vi testar! 100 000 körningar per indata



Vi testar! 100 000 körningar per indata



Vi testar! 100 000 körningar per indata



Medelfallsanalys, amorterad analys

Idé: Vi undersöker medelfallet av körtiden hos en algoritm!

- Antag att algoritmen körs n gånger i följd
- Beräkna den totala komplexiteten för körningarna
- Dela uttrycket på n för att få **medelfallet**, eller den **amorterade komplexiteten**

Amorterad analys av append

Kostnad

1										1
1	2									1+1
1	2	3								2+1
1	2	3	4							1
1	2	3	4	5						4+1

Amorterad analys av append

Idé: Låt oss ha en "skuld"

	Kostnad	Betala	Skuld
			0
1	1	1	0
1 2	1+1	2	0
1 2 3	2+1	2	1
1 2 3 4	1	2	0
1 2 3 4 5	4+1	2	3

Amorterad analys av append

Idé: Låt oss ha en "skuld"

Kostnad Betala Skuld

1	2	3	4	5			
---	---	---	---	---	--	--	--

4+1

2

3

1	2	3	4	5	6		
---	---	---	---	---	---	--	--

1

2

2

1	2	3	4	5	6	7	
---	---	---	---	---	---	---	--

1

2

1

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

1

2

0

Mer formell analys av append

$$\begin{aligned} \mathcal{O}(\underbrace{1 + 1 + \dots + 1}_{\text{insättning, } n \text{ gånger}} + \underbrace{1 + 2 + 4 + 8 + \dots + n}_{\text{kopiering, } \log_2(n) \text{ gånger}}) = \\ \mathcal{O}(\underbrace{n}_{\text{förenklat}} + \underbrace{2^{1+\log_2(n)} - 1}_{\text{förenkling av summan, tänk binärt}}) = \\ \mathcal{O}(n + 2 \cdot 2^{\log_2(n)} - 1) = \\ \mathcal{O}(n + 2n - 1) = \mathcal{O}(n) \end{aligned}$$

append-funktionen

Bästa fall: $\Theta(1)$

Värsta fall: $\Theta(n)$

Medelfall: $\Theta(1)$

När kan vi använda medelfallet?

När kan vi använda medelfallet?

A:

```
void insert(int n) {  
    vector<int> x;  
  
    for (int i = 0; i < n; i++)  
        x.push_back(i);  
}
```

När kan vi använda medelfallet?

B:

```
void f(vector<int> &data) {  
    int sum = 0;  
    for (int i = 1; i < data.size(); i += 2) {  
        sum += data[i];  
    }  
    data.push_back(sum);  
}
```

När kan vi använda medelfallet?

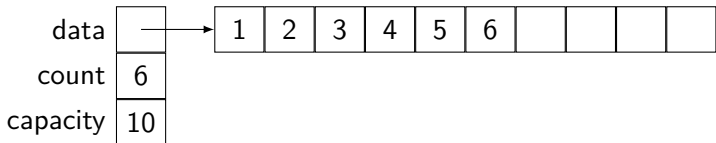
C:

```
void f(vector<int> &data) {  
    int sum = 0;  
    for (int i = data.size() - 1; i > 0; i--) {  
        sum += data[i];  
        data.push_back(sum);  
    }  
}
```

- 1 Abstrakta datatyper (ADT)
- 2 Länkad lista – `list`
- 3 Enkel arrayimplementation
- 4 Bättre arrayimplementation – `vector`
- 5 Mer algoritmkomplexitet
- 6 **`vector` – fortsättning**
- 7 Sammanfattning

Dynamisk array – `std::vector`

Idé: Allokera lite mer minne än vad vi behöver!



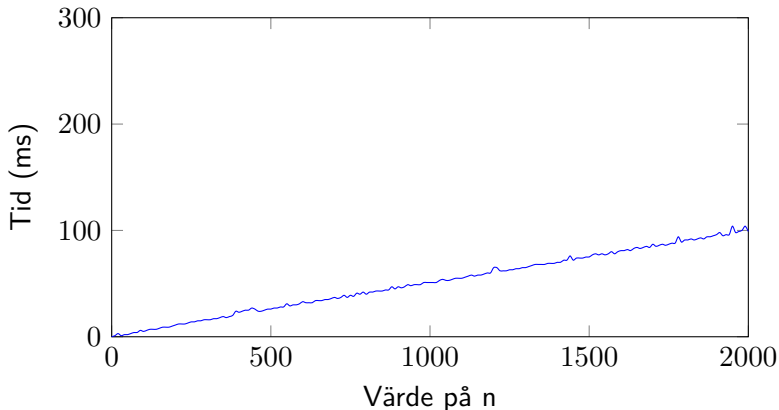
Komplexitet:

`elemAt(i)` $\Theta(1)$

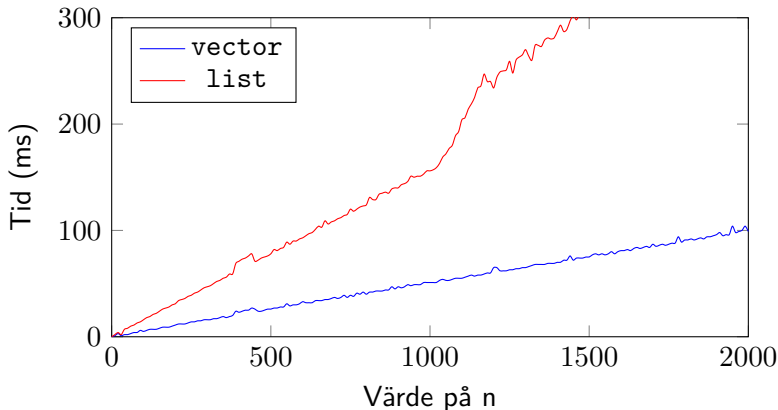
`append(x)` $\Theta(1)$ (amorterad)

`insert(it, x)` $\mathcal{O}(n)$

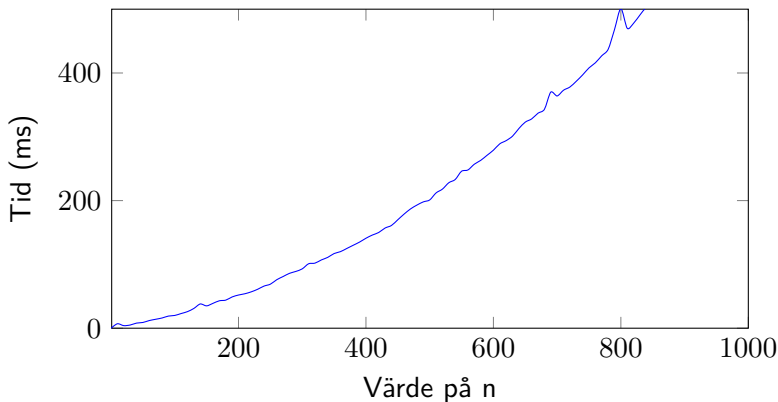
Vi testar iteration! 100 000 körningar per indata



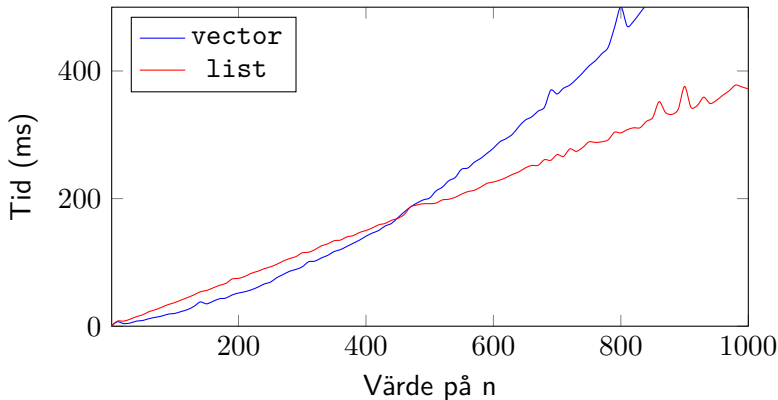
Vi testar iteration! 100 000 körningar per indata



Vi testar insert! 10 000 körningar per indata



Vi testar insert! 10 000 körningar per indata



Är vector eller list bäst?

- Modern hårdvara är väldigt bra på linjär åtkomst i minnet.
- vector vinner oftast...
- ...förutom då insättning i stora listor förekommer ofta, och iteration förekommer sällan

- 1 Abstrakta datatyper (ADT)
- 2 Länkad lista – `list`
- 3 Enkel arrayimplementation
- 4 Bättre arrayimplementation – `vector`
- 5 Mer algoritmkomplexitet
- 6 `vector` – fortsättning
- 7 Sammanfattning

ADT lista

	Länkad lista	Dynamiskt array
elemAt	$\mathcal{O}(n)$	$\Theta(1)$
append	$\Theta(1)$	$\Theta(1)$ (amorterad)
insert	$\Theta(1)$	$\mathcal{O}(n)$

I vilka fall ska vi använda en länkad lista?

I vilka fall ska vi använda ett dynamiskt array?

I kursen framöver

- Föreläsningar i nästa vecka
 - ADT stack, kö, prioritetskö, symboltabell
 - Träd (sökträd, balanserade sökträd)
- Extrauppgifter
 - 11332 (enkel)
Beräkna siffersumman i siffror. Fundera extra på maximal storlek.
 - 414 (svårare)
Se om två "ytor" passar ihop.

Filip Strömbäck

www.liu.se