TDDI16 – Föreläsning 3

Symboltabeller och träd

Filip Strömbäck

LINKÖPING UNIVERSITY

Planering

Vecka	Fö	Lab
36	Komplexitet, Linjära strukturer	----
37	Träd, AVL-träd	1---
38	Hashning, meet-in-the-middle	12--
39	Grafer, graftraversering och kortaste vägen	-2--
40	-	--3-
41	Sortering	--34
42	Repetition	---4

IMPA

Om ni gillar Kattis, testa IMPA!

`https://www.ida.liu.se/projects/impa/new/`

Den 5:e oktober hålls även NCPC

`https://nordic.icpc.io/ncpc2019/`

1 ADT dictionary (symboltabell)
2 Binära sökträd
3 AVL-träd, ett balanserat sökträd
4 Sammanfattning

LINKÖPING UNIVERSITY

Problem

Du håller på att skriva en rapport, och för att se till att göra ett gott intryck vill du se till att du inte har några stavfel i rapporten. Du har hittat en bra ordlista på internet (och i `/usr/share/dict/words`), och tänker att det inte kan vara särskilt svårt att skriva ett program som kontrollerar alla ord i rapporten efter stavfel.

Hur kan vi göra detta på ett bra sätt?

Hur svårt kan det vara?

```
int main() {
  vector<string> d = read_dict();

  int count{};
  string word;
  while (cin >> word) {
    if (find(d.begin(), d.end(), word) == d.end())
      cout << "Wrong:␣" << word << endl;
    count++;
  }
  cout << "Checked␣" << count << "␣words" << endl;
  return 0;
}
```

Testkörning

```
$ time spellcheck < thesis.txt
99171 words in the dictionary
...
Checked 38040 words

real 0m40.807s
user 0m40.700s
sys  0m0.100s
```

Inte jättebra...

Analys – find

```
template <typename Iter, typename Elem>
Iter find(Iter begin, Iter end, Elem elem) {

  for (Iter i = begin; i != end; ++i) {
    if (*i == elem) {
      return i;
    }
  }
  return end;
}
```

Analys – find

```
template <typename Iter, typename Elem>
Iter find(Iter begin, Iter end, Elem elem) {

  for (Iter i = begin; i != end; ++i) {
    if (*i == elem) {
      return i;
    }
  }
  return end;
}
```

Analys – find

```
template <typename Iter, typename Elem>
Iter find(Iter begin, Iter end, Elem elem) {

  for (Iter i = begin; i != end; ++i) {
    if (*i == elem) {
      return i;
    }
  }
  return end;
}
```

Body of `if` statement: $\mathcal{O}(1)$

Whole `for` loop: n gånger

Totalt: $n \cdot \mathcal{O}(1) = \mathcal{O}(n)$ Vad är *bästa* och *värsta* fall?

Bästa och värsta fallet

Vad kan vi säga om bästa och värsta fallet på `find`?

`find`, bästa fall $\Theta(1)$

`find`, värsta fall $\Theta(n)$

Vad är **relevant** att säga? Vad är **tillräckligt** att säga?

Bästa och värsta fallet

Vad kan vi säga om bästa och värsta fallet på `find`?

`find`, bästa fall	$\Theta(1)$	$\mathcal{O}(1)$	$\Omega(1)$
`find`, värsta fall	$\Theta(n)$	$\mathcal{O}(n)$	$\Omega(n)$

Vad är **relevant** att säga? Vad är **tillräckligt** att säga?

Bästa och värsta fallet

Vad kan vi säga om bästa och värsta fallet på `find`?

`find`, bästa fall	$\Theta(1)$	$\mathcal{O}(1)$	$\Omega(1)$
`find`, värsta fall	$\Theta(n)$	$\mathcal{O}(n)$	$\Omega(n)$
`find`	-	$\mathcal{O}(n)$	$\Omega(1)$

Vad är **relevant** att säga? Vad är **tillräckligt** att säga?

Analys

```
while (cin >> word) {
  if (find(...) == d.end())   O(n)
    cout << ...;              O(1)
}
```

$\mathcal{O}(n)$, $\mathcal{O}(1)$ — m gånger

Analys

```
while (cin >> word) {
  if (find(...) == d.end())     O(n)
    cout << ...;                O(1)
}
```

$\mathcal{O}(n)$, $\mathcal{O}(1)$ — m gånger

Totalt: $m(\mathcal{O}(1) + \mathcal{O}(n)) = m \cdot \mathcal{O}(n) = \mathcal{O}(mn)$

Analys

```
while (cin >> word) {
  if (find(...) == d.end())   O(n)  ⎫
    cout << ...;              O(1)  ⎬ m gånger
}                                   ⎭
```

Totalt: $m(\mathcal{O}(1) + \mathcal{O}(n)) = m \cdot \mathcal{O}(n) = \mathcal{O}(mn)$

Om $m \approx n \implies \mathcal{O}(n^2)$: bra eller dåligt?

Analys – vad behövs?

Analys – vad behövs?

	`vector`
Insättning	
Medlemstest	
Stavningskontroll	

Analys – vad behövs?

	`vector`	Önsketänkande
Insättning	$\mathcal{O}(1)$	
Medlemstest	$\mathcal{O}(n)$	
Stavningskontroll	$\mathcal{O}(mn)$	
$\approx$	$\mathcal{O}(n^2)$	

Analys – vad behövs?

	`vector`	Önsketänkande
Insättning	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Medlemstest	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Stavningskontroll	$\mathcal{O}(mn)$	$\mathcal{O}(m)$
$\approx$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$

Kan vi konstruera en sådan datastruktur?

Analys – vad behövs?

	`vector`	Önsketänkande	Mål idag
Insättning	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\log(n))$
Medlemstest	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log(n))$
Stavningskontroll	$\mathcal{O}(mn)$	$\mathcal{O}(m)$	$\mathcal{O}(m \log(n))$
$\approx$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log(n))$

Kan vi konstruera en sådan datastruktur?

ADT dictionary (symboltabell)

En **oordnad** mängd av par, **nyckel-värde**.

Kan vara sorterad efter **nyckel**.

size()	Antal element i datastrukturen
insert(k, v)	Lägg till ett nyckel-värde-par
remove(k)	Ta bort paret med nyckeln **k**
contains(k)	Finns nyckeln **k**?
get(k)	Hämta värdet för nyckeln **k**

Vi testar set...

```
int main() {
  set<string> d = read_dict();

  int count{};
  string word;
  while (cin >> word) {
    if (d.find(word) == d.end())
      cout << "Wrong:␣" << word << endl;
    count++;
  }
  cout << "Checked␣" << count << "␣words" << endl;
  return 0;
}
```

Testkörning

```
$ time spellcheck < thesis.txt
99171 words in the dictionary
...
Checked 38040 words

real 0m0.475s
user 0m0.288s
sys  0m0.016s
```

Nu börjar det likna något!

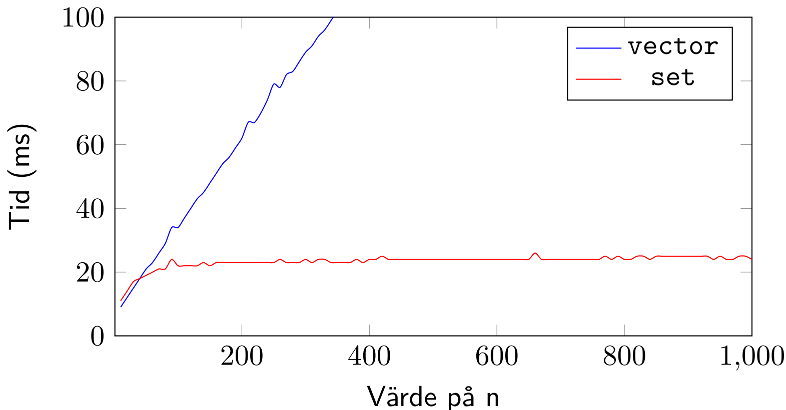
Analys

```
while (cin >> word) {
  if (d.find(...) == d.end())   O(?)
    cout << ...;                O(1)
}
```

$\mathcal{O}(?)$, $\mathcal{O}(1)$ — m gånger

Vad har `find` för komplexitet?

Vi testar! 100 000 körningar per indata

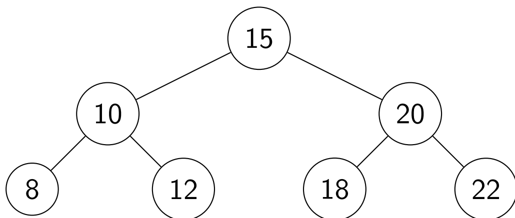


1 ADT dictionary (symboltabell)
2 Binära sökträd
3 AVL-träd, ett balanserat sökträd
4 Sammanfattning

LINKÖPING UNIVERSITY

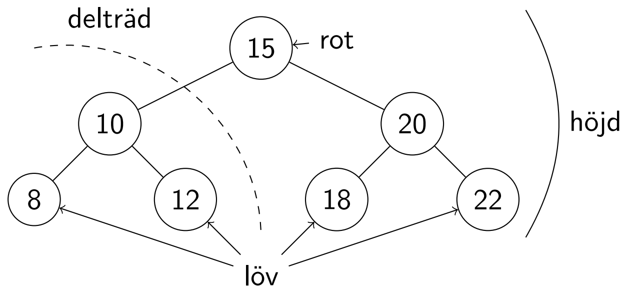
Hur är `set` implementerad?

Idé: Minimera söktid genom att använda ett sökträd!



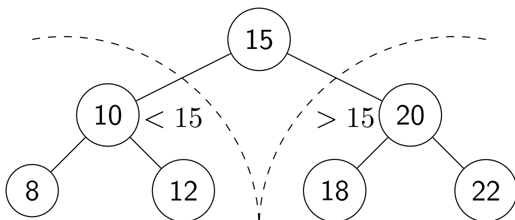
Hur är `set` implementerad?

Idé: Minimera söktid genom att använda ett sökträd!



Hur är `set` implementerad?

Idé: Minimera söktid genom att använda ett sökträd!



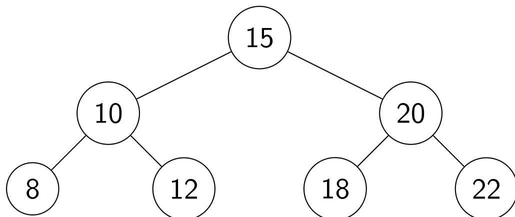
Hitta element i ett sökträd

```
class Node {
public:
  Node(int v, Node *l, Node *r) :
    v{v}, l{l}, r{r} {}

  int v;
  Node *l;
  Node *r;
};
```

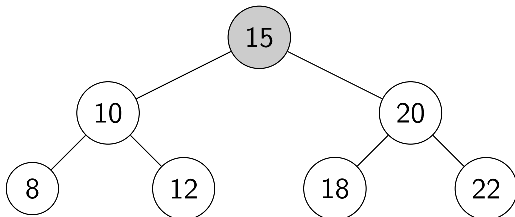
Hitta element i ett sökträd

```
Node *find(Node *root, int value) {
  if (!root)
    return nullptr;

  if (value < root->v) {
    return find(root->l, value);
  } else if (value > root->v) {
    return find(root->r, value);
  } else { // value == root->v
    return root;
  }
}
```

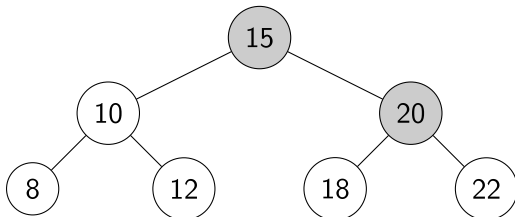
Exempel: Hitta 18



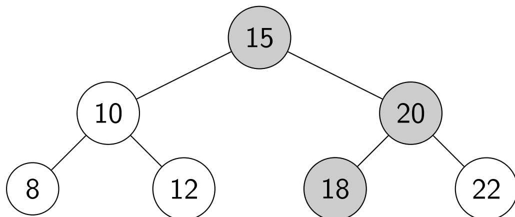
Exempel: Hitta 18



Exempel: Hitta 18



Exempel: Hitta 18



Komplexitet? Värsta och bästa fall?

Insättning

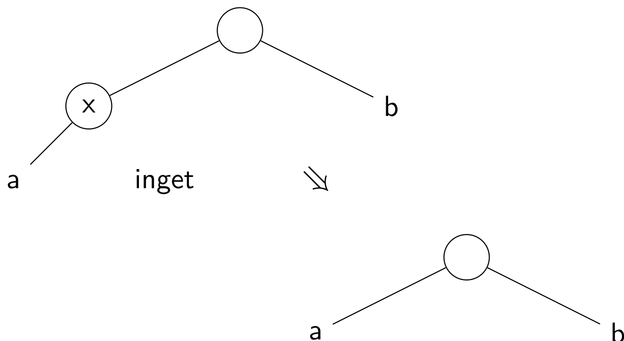
Insättning görs på liknande sätt. Testa exempelvis:

10, 5, 15, 20, 7, 12, 30, 40, 50, ...

Komplexitet?

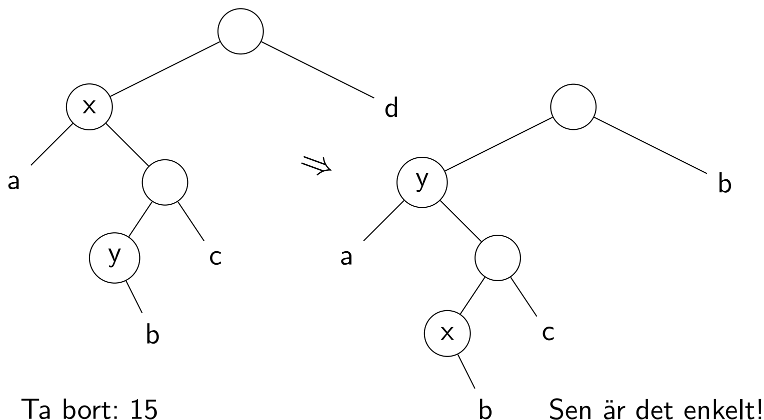
Vad kan vi säga om höjden av trädet?

Borttagning – enkelt fall



Ta bort: 7

Borttagning – generellt fall



Ta bort: 15

Sen är det enkelt!

Höjden av ett sökträd?

h: trädets höjd

n: antal noder i trädet

I värsta fall: $h = n$

I bästa fall: $n = 2^h - 1 \implies h = \log_2(n + 1)$

$\Rightarrow$ Vi vill se till att sökträdet är balanserat!

Tidskomplexitet hos ett BST

	Obalanserat	Balanserat
Insättning	$\mathcal{O}(n)$	$\mathcal{O}(\log_2(n))$[1]
Medlemstest	$\mathcal{O}(n)$	$\mathcal{O}(\log_2(n))$
Stavningskontroll	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log_2(n))$

[1] Om det inte är dyrt att balansera trädet

1 ADT dictionary (symboltabell)
2 Binära sökträd
3 AVL-träd, ett balanserat sökträd
4 Sammanfattning

LINKÖPING UNIVERSITY

Balansering av träd

Sökträdet blir ineffektivt om det inte är balanserat.

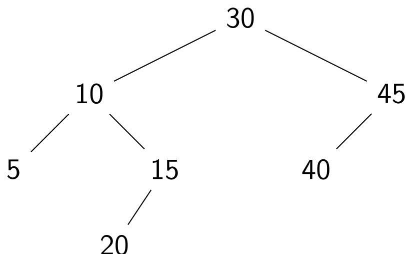
Idé: Håll koll på djupet i varje nod och ”rotera” om det skiljer för mycket.

Viktiga frågor:

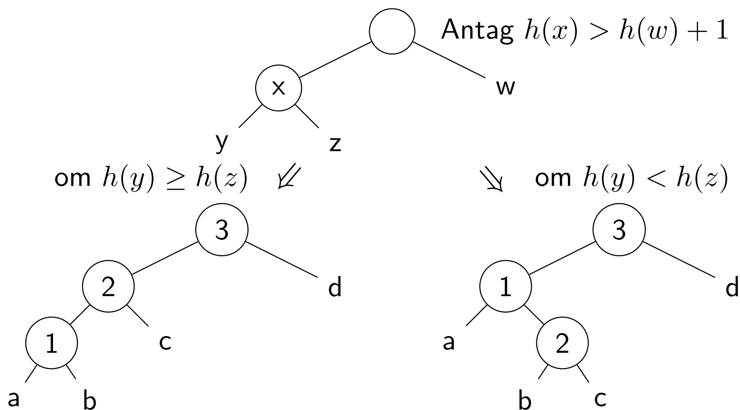
- Kan vi uppdatera höjderna inom tidsramarna?
- Vilka noder kan behöva ”roteras”?
- Hur lång tid tar det?
- Hur stor obalans ska vi tolerera?

AVL-träd – Idé

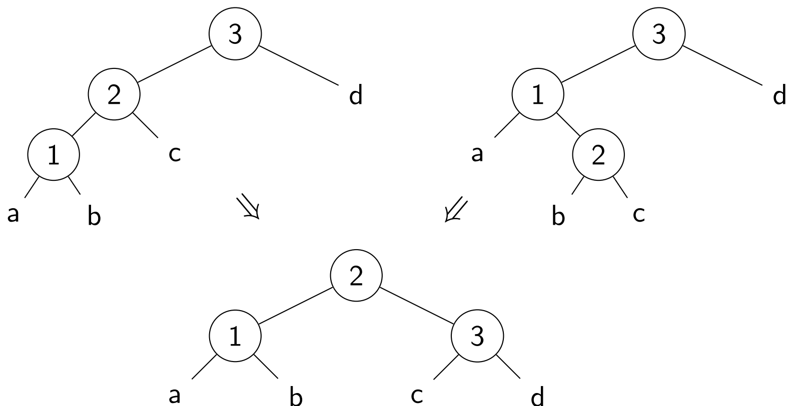
- Kom ihåg hur högt varje delträd är
- Skillnaden i höjd mellan syskon får maximalt vara 1
- *Rotera* noter vid insättning och borttagning



AVL-rotationer



AVL-rotationer



Insättning

Insättning görs som i sökträd, med balansering efteråt

Testa: 10, 5, 15, 20, 7, 30, 40, 35, ...

Komplexitet?

Borttagning

Borttagning görs också som i sökträd, med balansering efteråt

Ta bort: 7, 10, 5, 15

1 ADT dictionary (symboltabell)
2 Binära sökträd
3 AVL-träd, ett balanserat sökträd
4 Sammanfattning

LINKÖPING UNIVERSITY

I kursen framöver

- Nästa föreläsning
  - ADT stack, kö, prioritetskö
  - Trädtraversering, heap, (fenwickträd)
- Uppgifter i Kattis
  - `haypoints` (enkel)
    Användning av en lämplig datatyp.
  - `marblestree` (svårare)
    Övning på att traversera träd. Traversera inte onödigt många gånger!

Filip Strömbäck

www.liu.se

LINKÖPING
UNIVERSITY