

# TDDI16 – Föreläsning 4

Fler abstrakta datatyper

Filip Strömbäck

# Planering

| Vecka | Fö                              | Lab  |
|-------|---------------------------------|------|
| 36    | Komplexitet, Linjära strukturer | ---- |
| 37    | Träd, AVL-träd                  | 1--- |
| 38    | Hashning                        | 12-- |
| 39    | Grafer och kortaste vägen       | 12-- |
| 40    | Fler grafalgoritmer             | -23- |
| 41    | Sortering                       | --3- |
| 42    | Mer sortering, beräkningsbarhet | --34 |
| 43    | Tentaförberedelse               | ---4 |

# IMPA

Om ni gillar extrauppgifterna, testa IMPA!

<https://www.ida.liu.se/projects/impa/new/>

Den 4:e oktober hålls NCPC. Tävlingen sker samtidigt i norden, det kommer gå att delta på LiU. Målet är att alla deltagande lag ska kunna lösa minst 1 problem.

<https://nordic.icpc.io/ncpc2025/>

- 1 Trädtraversering
- 2 Stackar, köer och prioritetsköer
- 3 Prioritetsköer och heapar
- 4 Fenwickträd
- 5 Sammanfattning

# Trädtraversering

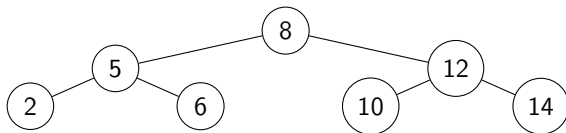
```
class Node {  
public:  
    Node(int v, Node *l, Node *r) :  
        v{v}, l{l}, r{r} {}  
  
    int v;  
    Node *l;  
    Node *r;  
};
```

## Trädtraversering

```
void traverse(Node *n) {  
    if (!n)  
        return;  
  
    // 1  
    traverse(n->l);  
    // 2  
    traverse(n->r);  
    // 3  
}
```

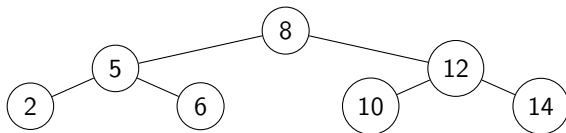
cout << n->v << endl; sätts in vid 1, 2 eller 3

## Preorder – innan rekursiva anrop



```
void traverse(Node *n) {  
    if (!n)  
        return;  
  
    cout << n->v << endl;  
    traverse(n->l);  
    traverse(n->r);  
}
```

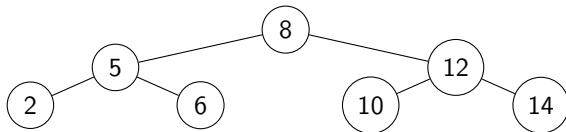
## Preorder – innan rekursiva anrop



```
void traverse(Node *n) {  
    if (!n)  
        return;  
  
    cout << n->v << endl;  
    traverse(n->l);  
    traverse(n->r);  
}
```

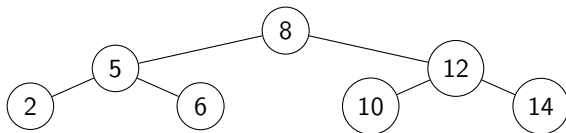
Resultat: 8, 5, 2, 6, 12, 10, 14

## Inorder – mellan rekursiva anrop



```
void traverse(Node *n) {  
    if (!n)  
        return;  
  
    traverse(n->l);  
    cout << n->v << endl;  
    traverse(n->r);  
}
```

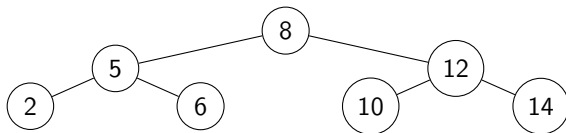
## Inorder – mellan rekursiva anrop



```
void traverse(Node *n) {  
    if (!n)  
        return;  
  
    traverse(n->l);  
    cout << n->v << endl;  
    traverse(n->r);  
}
```

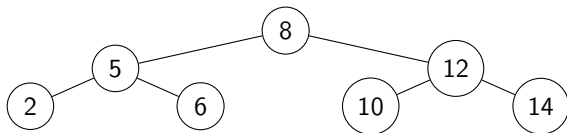
Resultat: 2, 5, 6, 8, 10, 12, 14

## Postorder – efter rekursiva anrop



```
void traverse(Node *n) {  
    if (!n)  
        return;  
  
    traverse(n->l);  
    traverse(n->r);  
    cout << n->v << endl;  
}
```

## Postorder – efter rekursiva anrop

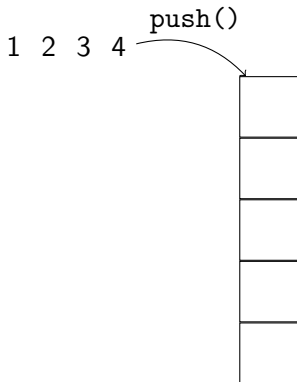


```
void traverse(Node *n) {  
    if (!n)  
        return;  
  
    traverse(n->l);  
    traverse(n->r);  
    cout << n->v << endl;  
}
```

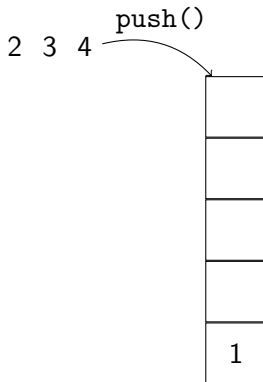
Resultat: 2, 6, 5, 10, 14, 12, 8

- 1 Trädtraversering
- 2 Stackar, köer och prioritetsköer
- 3 Prioritetsköer och heapar
- 4 Fenwickträd
- 5 Sammanfattning

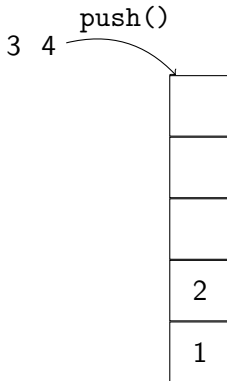
# ADT stack



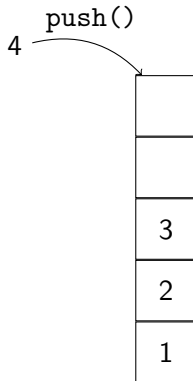
# ADT stack



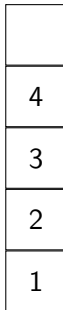
# ADT stack



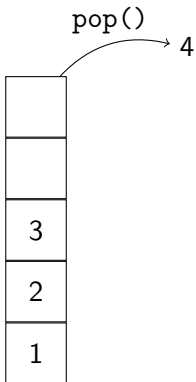
# ADT stack



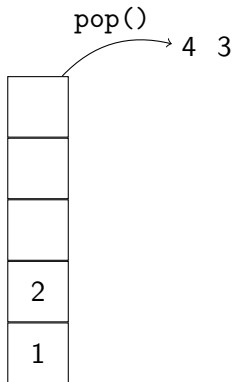
# ADT stack



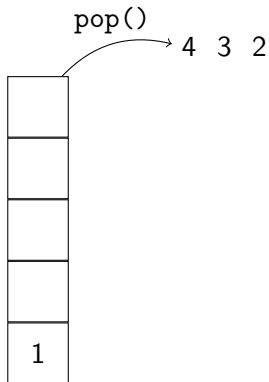
# ADT stack



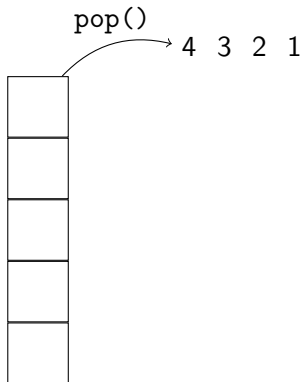
# ADT stack



# ADT stack



# ADT stack



# Implementation

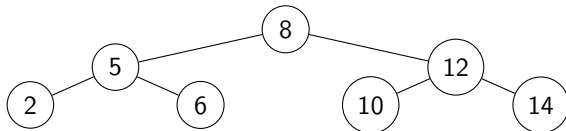
- Länkad lista
  - Kan ge sämre prestanda i och med pekarindirektioner
- Array
  - Måste ibland omallokeras
  - Hur hanterar vi långa sekvenser av `push()` och `pop()`?

## Iterativ trädtraversering

```
void traverse(Node *root) {  
    stack<Node *> s;  
    s.push(root);  
    while (!s.empty()) {  
        Node *n = s.top(); s.pop();  
        cout << n->v << endl;  
        if (n->l) s.push(n->l);  
        if (n->r) s.push(n->r);  
    }  
}
```

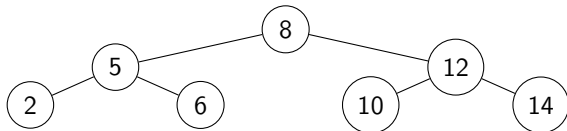
I vilken ordning traverseras trädet?

## Iterativ trädtraversering



```
void traverse(Node *root) {  
    stack<Node *> s;  
    s.push(root);  
    while (!s.empty()) {  
        Node *n = s.top(); s.pop();  
        cout << n->v << endl;  
        if (n->l) s.push(n->l);  
        if (n->r) s.push(n->r);  
    }  
}
```

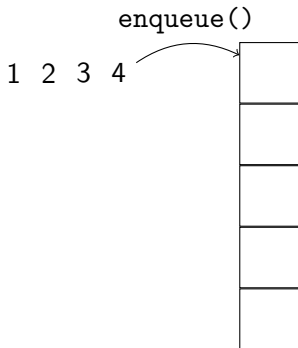
## Iterativ trädtraversering



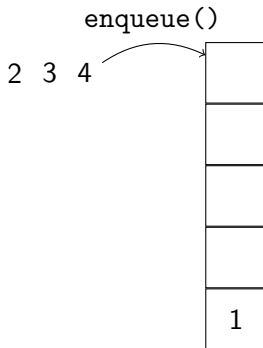
```
void traverse(Node *root) {  
    stack<Node *> s;  
    s.push(root);  
    while (!s.empty()) {  
        Node *n = s.top(); s.pop();  
        cout << n->v << endl;  
        if (n->l) s.push(n->l);  
        if (n->r) s.push(n->r);  
    }  
}
```

Ger: 8, 12, 14, 10, 5, 6, 2

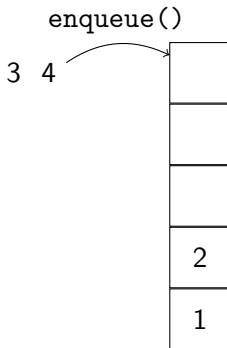
# ADT kö



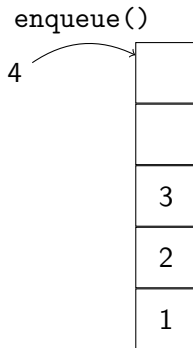
# ADT kö



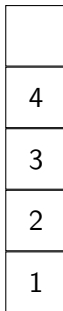
# ADT kö



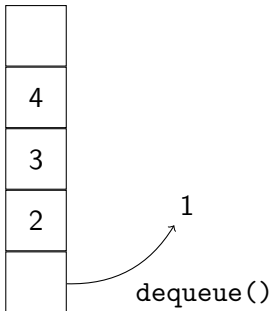
# ADT kö



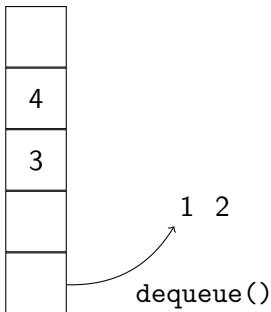
# ADT kö



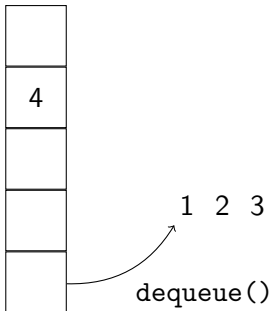
# ADT kö



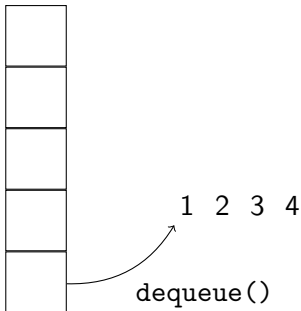
# ADT kö



# ADT kö



# ADT kö



# Implementation

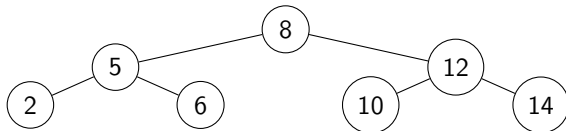
- Länkad lista
  - Kan ge sämre prestanda i och med pekarindirektioner
- Array
  - Måste ibland omallokeras
  - Hur hanterar vi långa sekvenser av `enqueue()` och `dequeue()`?

## Iterativ trädtraversering

```
void traverse(Node *root) {  
    queue<Node *> q;  
    q.push(root);  
    while (!q.empty()) {  
        Node *n = q.front(); q.pop();  
        cout << n->v << endl;  
        if (n->l) q.push(n->l);  
        if (n->r) q.push(n->r);  
    }  
}
```

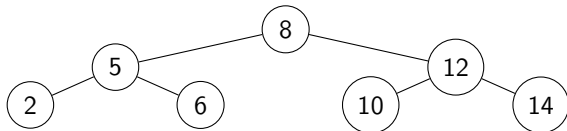
I vilken ordning traverseras trädet?

## Iterativ trädtraversering



```
void traverse(Node *root) {  
    queue<Node *> q;  
    q.push(root);  
    while (!q.empty()) {  
        Node *n = q.front(); q.pop();  
        cout << n->v << endl;  
        if (n->l) q.push(n->l);  
        if (n->r) q.push(n->r);  
    }  
}
```

## Iterativ trädtraversering



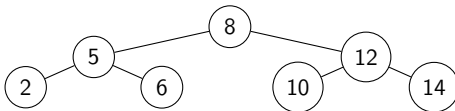
```
void traverse(Node *root) {  
    queue<Node *> q;  
    q.push(root);  
    while (!q.empty()) {  
        Node *n = q.front(); q.pop();  
        cout << n->v << endl;  
        if (n->l) q.push(n->l);  
        if (n->r) q.push(n->r);  
    }  
}
```

Ger: 8, 5, 12, 2, 6, 10, 14

## Iterativ trädtraversering – Mer komplicerat

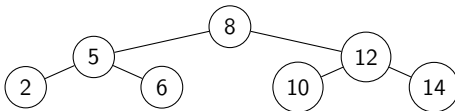
```
void traverse(Node *root) {  
    stack<Node *> s;  
    Node *current = root;  
    while (current || !s.empty()) {  
        while (current) {  
            s.push(current);  
            current = current->l;  
        }  
        current = s.top(); s.pop();  
        cout << current->v << endl;  
        current = current->r;  
    }  
}
```

## Iterativ trädtraversering – Mer komplicerat



```
void traverse(Node *root) {  
    stack<Node *> s;  
    Node *current = root;  
    while (current || !s.empty()) {  
        while (current) {  
            s.push(current);  
            current = current->l;  
        }  
        current = s.top(); s.pop();  
        cout << current->v << endl;  
        current = current->r;  
    }  
}
```

## Iterativ trädtraversering – Mer komplicerat



```
void traverse(Node *root) {  
    stack<Node *> s;  
    Node *current = root;  
    while (current || !s.empty()) {  
        while (current) {  
            s.push(current);  
            current = current->l;  
        }  
        current = s.top(); s.pop();  
        cout << current->v << endl;  
        current = current->r;  
    }  
}
```

Ger: 2, 5, 6, 8, 10, 12, 14

## Problem

Du driver ett nöjesfält. Du har iakttagit att ingen gillar att köa till attraktionerna, och du funderar därför på att introducera ett nytt kösystem.

Idén är enkel: besökarna går till en terminal och scannar sin biljett (som innehåller ett unikt nummer). Systemet kommer sedan ihåg att besökaren vill åka, och visar sedan besökarens namn på en display vid attraktionen.

Hur ska systemet hålla reda på vilka som vill åka, och vems tur det är härnäst? Vilken datastruktur ska vi använda?

## Förbättring(?) av kösystemet

Ditt kösystem har fungerat bra ett antal månader, men du funderar fortfarande på hur det kan förbättras. Du funderar på att låta personer som ogillar att vänta i kö betala för att få en bättre plats i kön.

I stället för att låta personer betala ett fast pris för att få stå i den "snabba" kön tänker du dig ett system liknande en auktion: när attraktionen blir ledig får den som har betalat mest just då gå först i kön.

Hur kan vi implementera detta på ett effektivt sätt?

- 1 Trädtraversering
- 2 Stackar, köer och prioritetsskøer
- 3 Prioritetsskøer och heapar
- 4 Fenwickträd
- 5 Sammanfattning

# ADT prioritetskö

Har följande operationer:

- `enqueue()` - Lägger till ett element
- `max()` - Hittar det största elementet
- `dequeue()` - Tar bort det största elementet

Hur implementerar vi en prioritetskö?

## Olika implementationer

|  | vector | Sökträd |
|--|--------|---------|
|--|--------|---------|

|           |  |  |
|-----------|--|--|
| enqueue() |  |  |
|-----------|--|--|

|       |  |  |
|-------|--|--|
| max() |  |  |
|-------|--|--|

|           |  |  |
|-----------|--|--|
| dequeue() |  |  |
|-----------|--|--|

## Olika implementationer

|           | vector           | Sökträd |
|-----------|------------------|---------|
| enqueue() | $\mathcal{O}(1)$ |         |
| max()     | $\mathcal{O}(n)$ |         |
| dequeue() | $\mathcal{O}(n)$ |         |

## Olika implementationer

|                        | vector           | Sökträd                |
|------------------------|------------------|------------------------|
| <code>enqueue()</code> | $\mathcal{O}(1)$ | $\mathcal{O}(\log(n))$ |
| <code>max()</code>     | $\mathcal{O}(n)$ | $\mathcal{O}(\log(n))$ |
| <code>dequeue()</code> | $\mathcal{O}(n)$ | $\mathcal{O}(\log(n))$ |

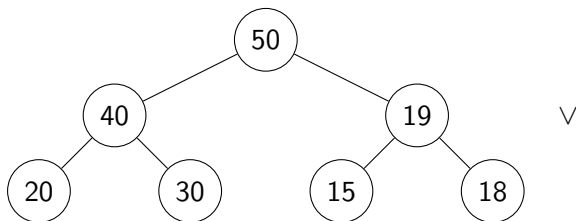
## Olika implementationer

|                        | vector           | Sökträd                | Heap                   |
|------------------------|------------------|------------------------|------------------------|
| <code>enqueue()</code> | $\mathcal{O}(1)$ | $\mathcal{O}(\log(n))$ | $\mathcal{O}(\log(n))$ |
| <code>max()</code>     | $\mathcal{O}(n)$ | $\mathcal{O}(\log(n))$ | $\mathcal{O}(1)$       |
| <code>dequeue()</code> | $\mathcal{O}(n)$ | $\mathcal{O}(\log(n))$ | $\mathcal{O}(\log(n))$ |

Ett sökträd verkar innehålla mer struktur än vi behöver!

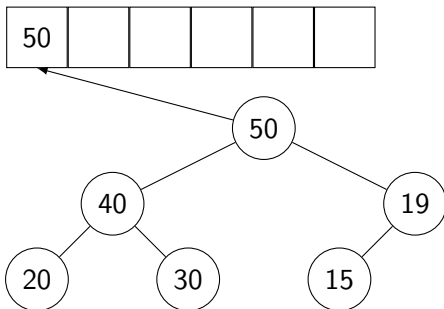
# Heap

Idé: Vi ordnar ett träd från rot till löv i stället för höger till vänster



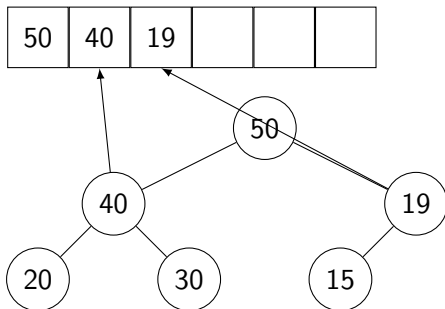
## Lagring av en heap

Vi kan se till att trädet alltid är **komplett**, så vi kan lagra det lager för lager i en array!



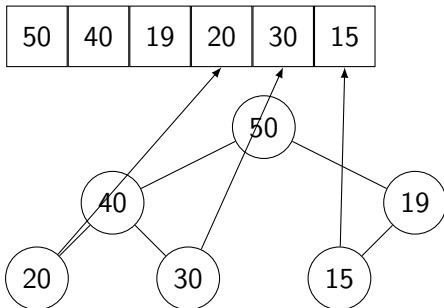
## Lagring av en heap

Vi kan se till att trädet alltid är **komplett**, så vi kan lagra det lager för lager i en array!



## Lagring av en heap

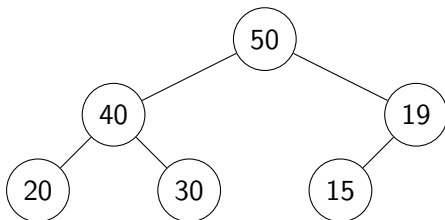
Vi kan se till att trädet alltid är **komplett**, så vi kan lagra det lager för lager i en array!



## Lagring av en heap

Vi kan se till att trädet alltid är **komplett**, så vi kan lagra det lager för lager i en array!

|    |    |    |    |    |    |
|----|----|----|----|----|----|
| 50 | 40 | 19 | 20 | 30 | 15 |
|----|----|----|----|----|----|



# Insättning

Observation: Vi kan bara sätta in saker sist i arrayen (längst ner till höger i trädet), annars är det inte längre komplett.

1. Sätt in elementet sist
2. Flytta det sedan uppåt tills heapegenskapen är uppfyllt (*sift-up*)

Exempel: Sätt in 45

## Borttagning

Observation: Vi kan bara ta bort saker sist i arrayen (längst ner till höger i trädet), annars är det inte längre komplett.

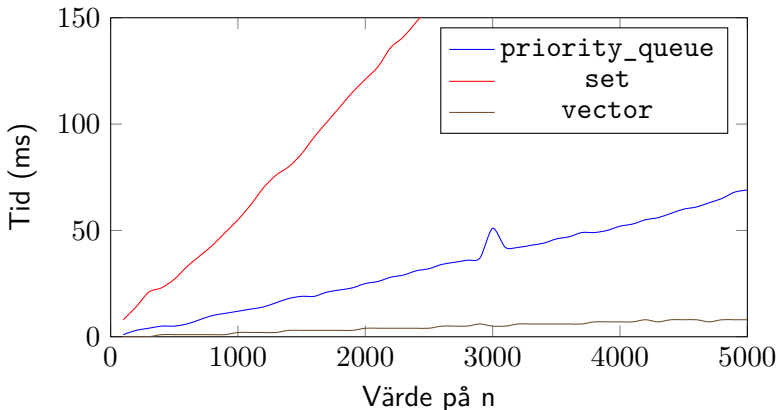
1. Byt plats på rotnoden och det sista elementet
2. Ta bort det sista elementet
3. Flytta det nya rotelementet nedåt tills heapegenskapen är uppfylld (*sift-down*)

## Make-heap

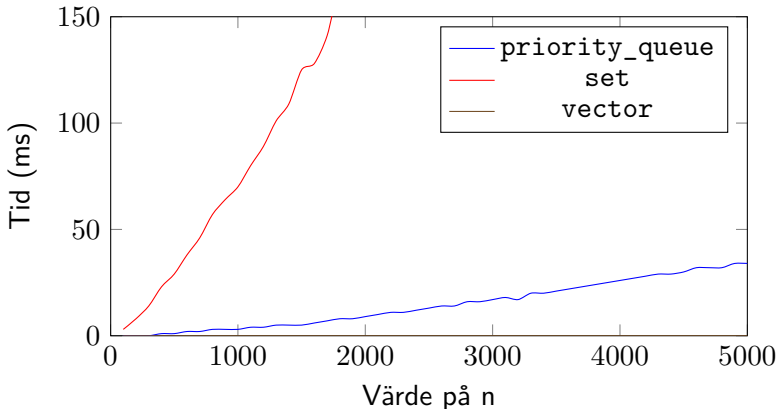
Om vi har en uppsättning element som vi vill skapa en heap av kan vi använda operationen *make-heap*, som kör i  $\mathcal{O}(n)$  tid, i stället för den  $\mathcal{O}(n \log(n))$  tid det tar att sätta in elementen ett och ett.

1. Börja med den första noden som har barn från slutet av arrayen:
2. Kör *sift-down* på den noden
3. Upprepa för alla noder som ligger på lägre index i arrayen

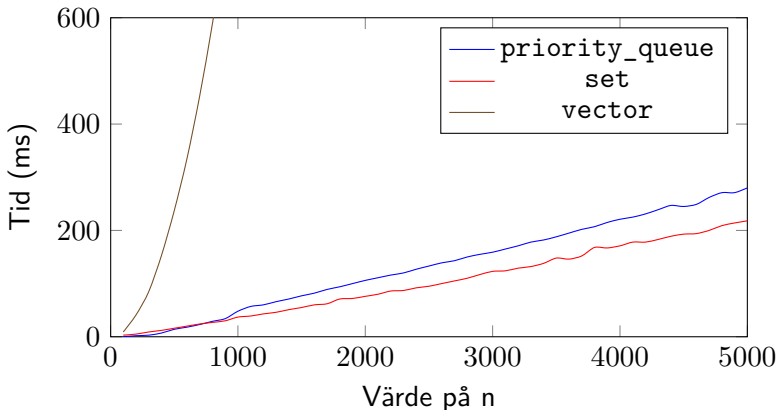
## Test av insättning (1000 körningar per gång)



## Test av make\_heap (1000 körningar per gång)



## Test av borttagning (tar bort $n$ element 1000 gånger)



## Andra heapar

I och med att `make_heap` är bättre än insättningar, kan vi försöka amortera kostnaden för insättning:

- Lägg in element osorterade i slutet.
- Kör `make_heap` när `max` eller `dequeue` anropas.
- Fungerar bra om vi gör många insättningar i en relativt tom heap.

Finns andra, ex.vis Fibonacci-heaps (Binomial heap med lat insättning).

<https://www.cs.usfca.edu/~galles/visualization/FibonacciHeap.html>

- 1 Trädtraversering
- 2 Stackar, köer och prioritetsköer
- 3 Prioritetsköer och heapar
- 4 **Fenwickträd**
- 5 Sammanfattning

## Problem

Du är en stor hyresvärd i staden. Du äger ett stort antal lägenheter, numrerade 1 till  $n$ . Under ditt dagliga arbete är du ofta i behov av att snabbt få svar på frågan: Vad är den totala hyran för ett eller flera sammanhängande kvarter, det vill säga: Vad är hyran för lägenheterna  $a$  till  $b$ ? I och med att du äger så många lägenheter ändras dessutom hyran ofta, så det får inte ta för lång tid att ändra hyran för en viss lägenhet.

Vi kan svara på frågan i  $\mathcal{O}(n)$  tid med en enkel array. Kan det bli bättre än så?

## Fenwickträd – Första idé

Operation: +

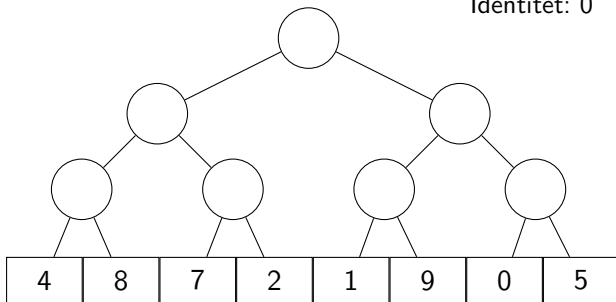
Identitet: 0

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 4 | 8 | 7 | 2 | 1 | 9 | 0 | 5 |
|---|---|---|---|---|---|---|---|

## Fenwickträd – Första idé

Operation: +

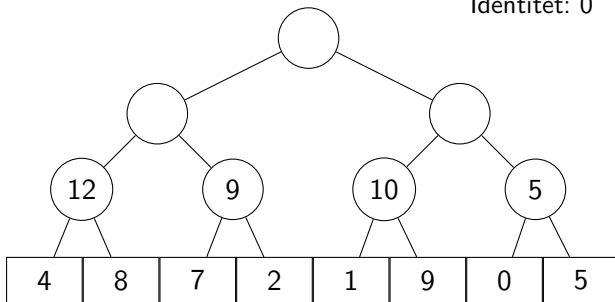
Identitet: 0



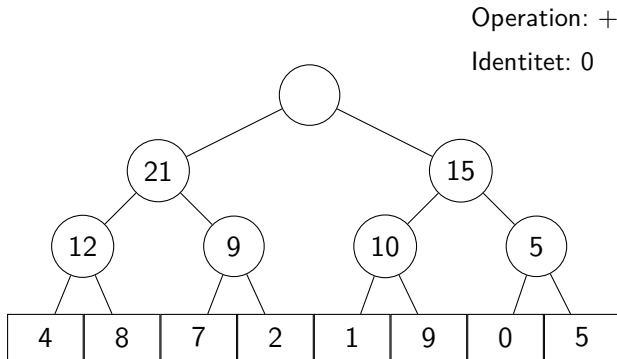
## Fenwickträd – Första idé

Operation: +

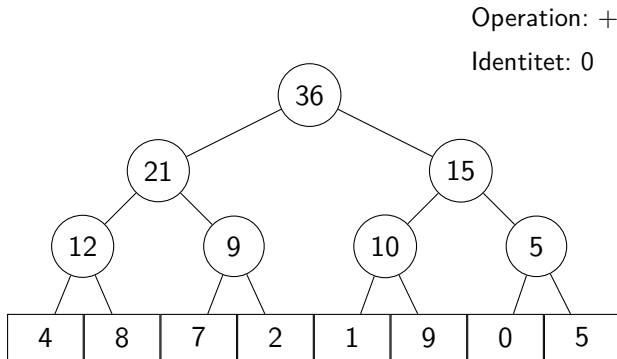
Identitet: 0



## Fenwickträd – Första idé

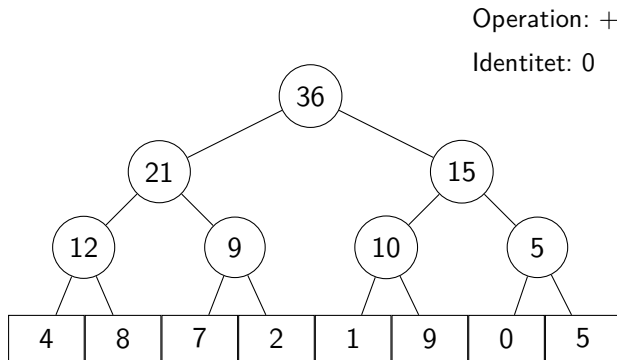


## Fenwickträd – Första idé



Komplett träd  $\Rightarrow$  kan lagras i array!

## Fenwickträd – Första idé



Vi kan hitta  $s(0, n)$  och  $s(a, b)$  (hur?)

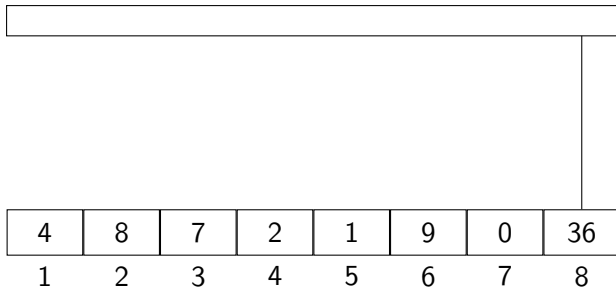
## Fenwickträd

Om operationen är inverterbar:  $s(a, b) = s(0, b) - s(0, a)$

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 4 | 8 | 7 | 2 | 1 | 9 | 0 | 5 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

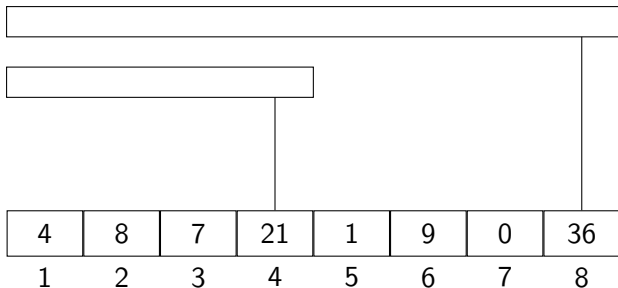
# Fenwickträd

Om operationen är inverterbar:  $s(a, b) = s(0, b) - s(0, a)$



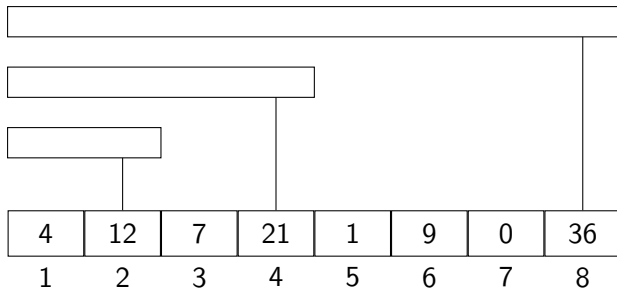
# Fenwickträd

Om operationen är inverterbar:  $s(a, b) = s(0, b) - s(0, a)$



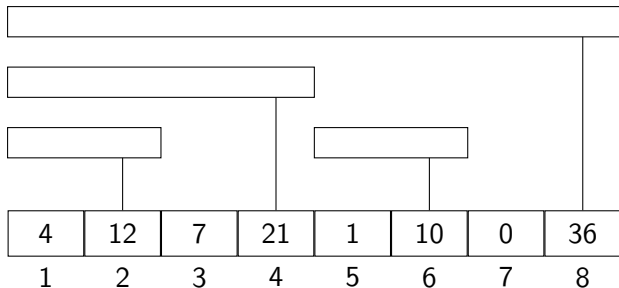
# Fenwickträd

Om operationen är inverterbar:  $s(a, b) = s(0, b) - s(0, a)$



# Fenwickträd

Om operationen är inverterbar:  $s(a, b) = s(0, b) - s(0, a)$



- 1 Trädtraversering
- 2 Stackar, köer och prioritetsköer
- 3 Prioritetsköer och heapar
- 4 Fenwickträd
- 5 Sammanfattning

# I kursen framöver

- Nästa föreläsning
  - Hashning (inför lab 2)
- Extrauppgifter
  - 10954 (enkel)  
Fundera på en allmän regel för att hitta den minimala kostnaden, och implementera den med lämplig datatyp!
  - 1513 (svårare)  
Håll koll på CD-skivor. Tips: ha plats för "för många" CD-skivor från början, och håll koll på vilka platser som är upptagna på lämpligt sätt.

Filip Strömbäck

[www.liu.se](http://www.liu.se)