DVA104 VT19

Datastrukturer och Algoritmer

Stefan Bygde

Dagens föreläsning

- Nyckelordet **`const`**
- `ADTerna stack, kö och set`
  - Möjliga implmentationer av ADTerna

Pekare och Funktioner

Varför skickar vi pekare till en funktion?

Pekare och Funktioner

Varför skickar vi pekare till en funktion?

- För att vi vill ändra värdet på det pekaren pekar på i funktionen
- För att slippa kopiera en struct
- För att arrayer alltid skickas som pekare
- För att en datastruktur representeras av en pekare

Pekare och Funktioner

Varför skickar vi pekare till en funktion?

- För att vi vill ändra värdet på det pekaren pekar på i funktionen
- För att slippa kopiera en struct
- För att arrayer alltid skickas som pekare
- För att en datastruktur representeras av en pekare

Problemet med pekare och funktioner

Varför skickar vi pekare till en funktion?

- För att slippa kopiera en struct
- För att arrayer alltid skickas som pekare
- För att en datastruktur representeras av en pekare

I många av dessa fall vill vi **inte** att funktionen ska kunna ändra på det den pekar på.

Exempel:

```
void printArrayOnScreen(int* arr, int size); // Ska enbart skriva ut arrayen, inte ändra på den
int getInfo(Hugestruct* huge); // Vi vill enbart returnera information från huge, inte ändra den
Data findElement(List list, Data element); // List är en pekare, men funktionen får inte ändra på listan
```

Problemet med pekare och funktioner

- Vi vill i dessa fall lämna programmeraren en **garanti** för att det som pekas på blir ändrat.
- Anledningen är vi vill undvika “överraskningar” eller misstag när vi anropar en funktion.
- Vi kan använda nyckelordet **`const`** före ett typnamn för att kompilatorn ser till att det som pekas på ej blir ändrat.

Problemet med pekare och funktioner

- Nya bättre exempel:

```
/* Kompilatorn ser nu till (genom att signalera fel) att elementen i arr inte kan bli ändrade */
void printArrayOnScreen(const int* arr, int size);

/* Kompilatorn ser till att ingen medlem i *huge kan bli ändrad */
int getInfo(const Hugestruct* huge);

/* List är typedefad som en pekare, listan får ej ändras (detta gäller dock tyvärr bara första elementet)
   Data SKULLE kunna vara typedefad som en pekare, men funktionen får naturligtvis inte ändra det */
Data findElement(const List list, const Data element);
```

Const som Parameter

Man kan använda en väldigt enkel tumregel för när man ska använda **`const`**-parametrar:

Använd en **`const`** parameter om (och endast om):

- Parametern är eller kan vara en pekare
- Det som pekas ut av parametern ska (eller får) inte ändras av funktionen

Const som Parameter

- Exempel på när **`const`** bör användas:

```
Data findElement(const List list, const Data element);
```

  - Listan ska ej ändras (vi ska bara hitta ett element), elementet ska ej ändras (bara hittas).

- Exempel på när **`const`** *ej* bör användas:

```
void addFirst(List* list);
```

  - I detta fall har vi som postcondition att `*list` ska peka på ett nytt element. Dvs, `*list` *ska* ändras av funktionen.

ADTn Stack

- En stack är en abstrakt datatyp (ADT). Vi intresserar oss just nu bara för hur den används.
- En stack kan enkelt konceptualiseras som en hög mynt:
- Vi kan:
  - Bara se framsidan av det översta myntet
  - Lägga ett mynt längst upp på högen
  - Ta bort myntet längst upp på högen

- Mynthögen är själva “stacken”
- Varje mynt är ett *element* i stacken

ADTn Stack

- En stack är en abstrakt datatyp (ADT). Vi intresserar oss just nu bara för hur den används.
- En stack kan enkelt konceptualiseras som en hög mynt:
- Vi kan:
  - Bara se framsidan av det översta myntet (peek)
  - Lägga ett mynt längst upp på högen (push)
  - Ta bort myntet längst upp på högen (pop)

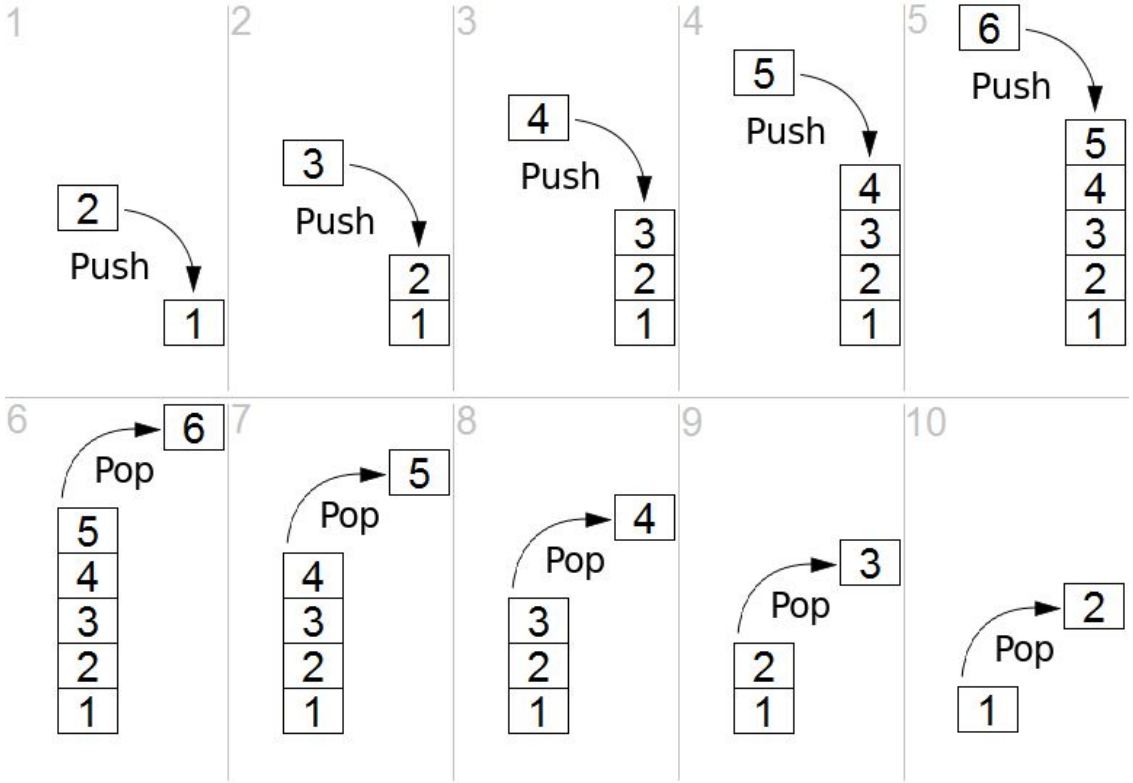
- Mynthögen är själva “stacken”
- Varje mynt är ett *element* i stacken

ADTn Stack

Hur en stack fungerar:

- Push för att lägga ett element överst.
- Pop för att ta bort ett element
- Peek tittar på översta elementet

Elementen i stacken ska kunna ha en godtycklig typ



Stackoperationer

```
Stack createStack(void); // Skapar en tom stack
```

Preconditions:

- Inga

Postconditions:

- Det returnerade värdet är en tom stack

Stackoperationer

```
// Returnerar 1 om stacken är tom 0 annars
int isEmpty(const Stack stack);
```

Preconditions:

- Inga

Postconditions:

- Inga

Stackoperationer

```
void push(Stack* stack, const Data data);
```

Preconditions:

- Stacken är inte full
  - Teoretiskt kan inte en stack bli full. Men i praktiken så finns begränsningar beroende på implementationen.

Postconditions:

- `data` ligger överst i `*stack`

Stackoperationer

```
void pop(Stack* stack);
```

Preconditions:

- Stacken är inte tom (med andra ord: vi får inte “poppa” en tom stack)

Postconditions:

- `Stack` innehåller ett element mindre än tidigare

Stackoperationer

```
Data peek(const Stack stack);
```

Preconditions:

- Stacken är inte tom

Postconditions:

- Inga

Stackoperationer

```
// Returnerar 1 om stacken är full 0 annars
int isFull(const Stack stack);
```

Preconditions:

- Inga

Postconditions:

- Inga

Igen: i teorin kan inte en stack bli full. Men pga datorn har begränsat minne (eller om stacken implementeras som en array) så kan den i praktiken bli full.

ADTn Stack: Interface

```
Stack createStack(void);
void push(Stack* stack, const Data data);
void pop(Stack* stack);
Data peek(const Stack stack);
int isEmpty(const Stack stack);
int isFull(const Stack stack);
```

Stack: Användning (i form av ett testprogram)

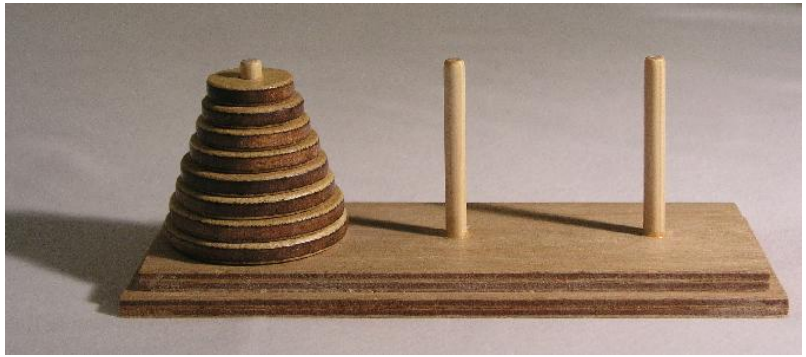
```c
#include “stack.h”

int main(void)
{
    Stack stack = createStack(); // Vi har nu en ny tom stack
    push(&stack, 5);  // Lägg elementet 5 överst på stacken
    assert(peek(stack) == 5); // Denna ska inte aktiveras: 5 ska ligga överst
    pop(&stack);// Ta bort översta elementet (5) från stacken
    assert(isEmpty(stack)); // Stacken ska nu vara tom

    return 0;
}
```

Stack Applikationer

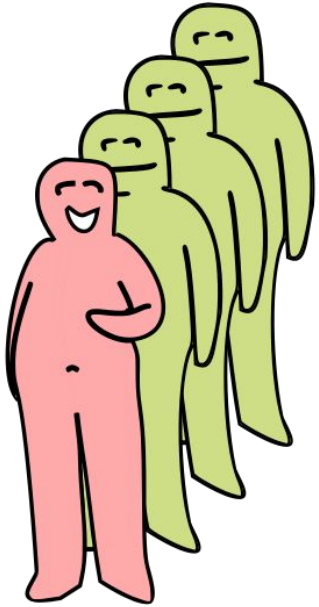
- Minne allokeras som bekant “på stacken”
- Call-stacken är också en stack
- Towers of Hanoi. Varje “pinne” är en stack.
- Backtracking (undo)



En stack kan också kallas för en LIFO-kö.
(Last In First Out)

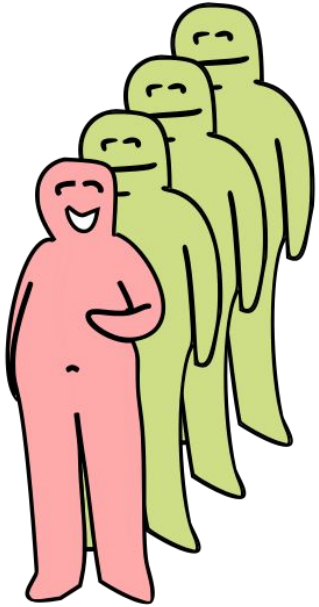
ADTn Kö

- Som en vanlig kassa/toa/bankomat/krogkö fungerar
  - Man kan bara ställa sig sist
  - Den som är först i kön får service
  - Man ser bara hela den som står först i kön (framifrån)
- Personerna i kön blir här “element”



ADTn Kö

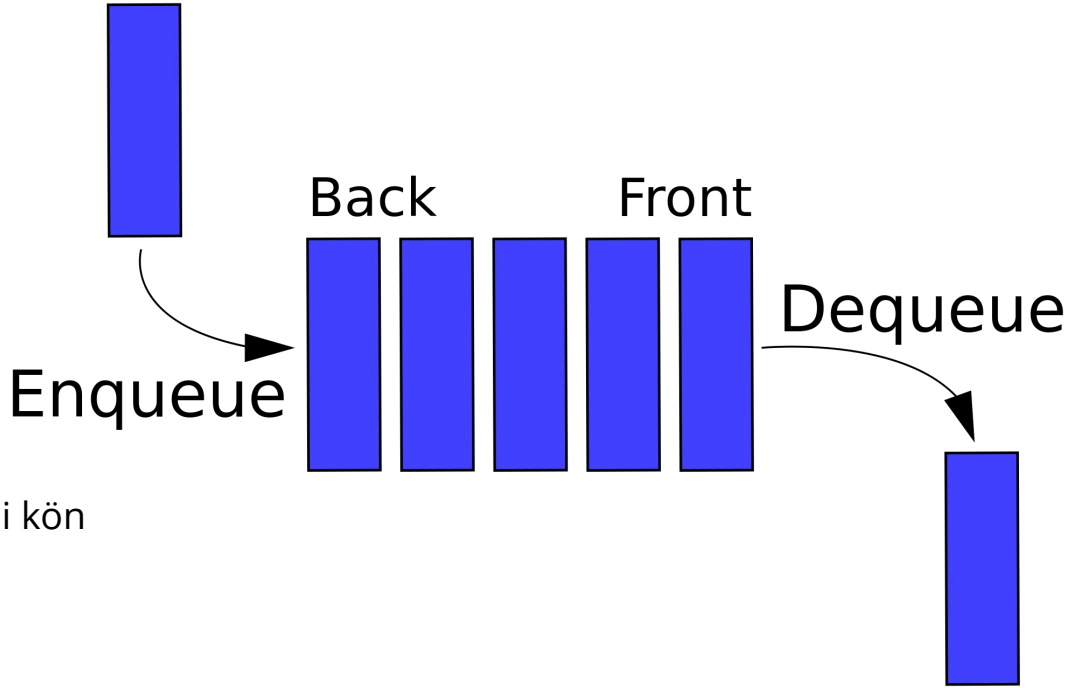
- Som en vanlig kassa/toa/bankomat/krogkö fungerar
  - Man kan bara ställa sig sist (enqueue)
  - Den som är först i kön får service (dequeue)
  - Man ser bara hela den som står först i kön (front)
- Personerna i kön blir här “element”



ADTn Kö

Så funkar en kö:

- Enqueue:
  - Lägg element sist i kön
- Dequeue:
  - Ta bort elementet längst fram i kön
- Front (eller peek):
  - Titta på första elementet i kön



Elementen i kön ska kunna ha godtycklig typ. Precis som för en stack.

Kööperationer

```
Queue createQueue(void); // Skapar en tom kö
```

Preconditions:

- Inga

Postconditions:

- Det returnerade värdet är en tom kö

Köoperationer

```
void enqueue(Queue* queue, const Data data);
```

Pre-conditions:

- Kön är inte full
  - Teoretiskt kan inte en kö bli full. Men i praktiken så finns begränsningar beroende på implementeationen.

Post-conditions:

- `data` ligger sist i `*queue`

Köoperationer

```
void dequeue(Queue* queue);
```

Preconditions:

- Kön är inte tom

Postconditions:

- `queue` innehåller ett element mindre än tidigare

Kööperationer

```
Data front(const Queue queue); // Returnerar elementet först i kön
```

Preconditions:

- Kön är inte tom

Postconditions:

- Inga

Kööperationer

```
// Returnerar 1 om kön är tom 0 annars
int isFull(const Queue queue);
```

Preconditions:

- Inga

Postconditions:

- Inga

Köoperationer

```
// Returnerar 1 om kön är tom 0 annars
int isEmpty(const Queue queue);
```

Preconditions:

- Inga

Postconditions:

- Inga

ADTn Kö: Interface

```
Queue createQueue(void);
void enqueue(Queue* queue, const Data data);
void dequeue(Queue* queue);
Data front(const Queue queue);
int isEmpty(const Queue queue);
int isFull(const Queue queue);
```

ADTn kö: användning

```
#include “queue.h”

int main(void)
{
	Queue queue = createQueue(); // Skapar en ny tom kö
	assert(isEmpty(queue)); // Kontrollera att kön verkligen är tom
	enqueue(&queue, 5); // Lägger till fem sist i kön
	enqueue(&queue, 8); // Lägger till åtta sist i kön
	assert(front(queue) == 5); // Eftersom fem var lagt till först ska det vara först i kön
	dequeue(&queue);// Tar bort elementet först i kön (5)
	assert(front(queue) == 8); // Eftersom 5 blivit borttaget ska 8 vara först i kön
	dequeue(&queue); // Tar bort 8, kön ska nu vara tom
	assert(isEmpty(queue));

	return 0;
}
```

ADTn Set

Ett set är en “mängd”. Ett element antingen tillhör eller inte tillhör en mängd.

- För varje element kan du avgöra om det tillhör mängden eller inte.
- Operationer:
  - Du kan lägga till ett element till mängden (add)
  - Du kan ta bort ett element från mängden (remove)
  - Du kan avgöra om ett element tillhör en mängd eller ej (isInSet)
- I denna datatyp fäster vi alltså ingen vikt vid ordningen elementen ligger i. Vi är bara intresserade av om det är där eller inte.
- Ett element tillhör Set eller inte, ett element kan inte förekomma “flera gånger” i ett Set.

ADTn Set

Exempel på hur `Set` kan användas:

```
add(5);// set innehåller elementet 5
add(8);    // set innehåller elementet 5 och 8

isInSet(3); // svaret på detta är nej eftersom 3 ej finns i mängden
inInSet(8); // svaret på detta är ja eftersom 8 finns i mängden

add(8); // Inget händer: 5 och 8 finns redan i mängden
```

ADTn Set: Applikationer

- `Set` kan användas för att t.ex hålla reda på vilka element som har processats.
- Man kan använda `Set` för att undvika dubbletter.
  - Om vi har ett Set av alla student-idn på MDH så kan vi kolla om ett ID är upptaget eller ej

Setoperationer

`Set createSet(void);` // Skapar ett Set utan element

Preconditions:

- Inga

Postconditions:

- Det returnerade värdet är ett tomt `Set`

Setoperationer

```
void addElement(Set* set, const Data data);
```

Pre-conditions:

- Mängden är inte full
  - Teoretiskt kan inte en mängd bli full. Men i praktiken så finns begränsningar beroende på implementationen.

Post-conditions:

- `data` finns i `*set`

Setoperationer

```
void removeElement(Set* set, const Data data);
```

Pre-conditions:

- Inga
  - Man skulle kunna tro att mängden inte får vara tom. Men istället händer inget om datat inte finns.

Post-conditions:

- `data` finns **inte** i `*set`

Setoperationer

```
// Returnerar 1 om data finns i set, 0 annars
int isInSet(const Set set, const Data data);
```

Pre-conditions:

- Inga

Post-conditions:

- Inga

Setoperationer

```
// Returnerar 1 om set är full, 0 annars
int isFull(const Set set);
```

Pre-conditions:

- Inga

Post-conditions:

- Inga

ADTn Set: Interface

```
Set createSet(void);
void addElement(Set* set, const Data data);
void removeElement(Set* set, const Data data);
int isInSet(const Set set, const Data data);
int isFull(const Set set);
```

ADTn Set: användning

```
int main(void)
{
	Set set = createSet();	// Skapar ett set utan element
	assert(!isInSet(5)); // Då vi inte lagt till något ska inte 5 finnas i mängden
	add(&set, 5);
	add(&set, 8);
	assert(isInSet(5)); // Vi har lagt till 5 så det ska finnas
	add(&set, 5);  // Inget händer om vi lägger till 5 igen
	assert(isInSet(5)); // Inget ska ha förändrats
	remove(&set, 5); // Vi tar bort 5
	assert(!isInSet(5)); // Nu ska inte 5 finnas i mängden längre

	return 0;
}
```

ADTer och preconditions

- När programmet fungerar korrekt ska inga asserts aktiveras
- Detta betyder att en programmerare som använder en ADT måste vi se till att inte anropa funktioner “fel”.

Exempel på felanvänding:

```
Stack stack = createStack(); // Skapa en tom stack
pop(&stack);  // FEL! Precondition är att stacken inte är tom (assert aktiveras)
```

Anropa en tom stack är inte en giltig operation. Det är upp till programmeraren att se till att det inte händer.

ADTer och Preconditions

Kan vi inte bara låta bli att ta bort ett element om vi anropar `pop()` på en tom stack?

- Visst, det skulle undvika att krasha programmet
- Det skulle fortfarande dock inte stämma med postcondition (att stacken innehåller ett element mindre än tidigare).
  - Vi bör se det som en “ogiltig” operation, precis som division med 0 eller att skriva utanför arraygränser

Ett annat exempel:

```
Stack stack = createStack(); // Skapa en tom stack
int x = peek(stack);    // FEL! Och vad skulle peek returnera istället?
// Vi kan inte returnera t.ex -1 (som felvärde) eftersom -1 är ett giltigt element
som skulle kunna ligga på stacken.
```

ADTer och preconditions

För att se till att inte bryta mot preconditions:

- Undvik att anropa funktioner fel och testa programmet ordentligt med hjälp av asserts.
- I vissa fall kan preconditions behöva verifieras innan:

```
if(isEmpty(stack))    // Stacken är potentiellt tom, så vi kontrollerar först
      printf(“Stacken är tom!”);
else
      pop(&stack); // Pop sker enbart om stacken ej är tom
```

Implementation

- Vi ska nu titta på möjliga implementationer för de tre ADTerna stack, kö och set.

Implementationer av ADTer

- Vi har tittat på interfacet till kö, stack och set.
  - Testprogram och hur dessa fungerar är helt oberoende av implementation
- Samtliga av dessa kan implementeras mha olika konkreta datastrukterer tex:
  - En array
  - En länkad lista
  - Set skulle även kunna implementeras mha ett binärt träd (kommer senare)

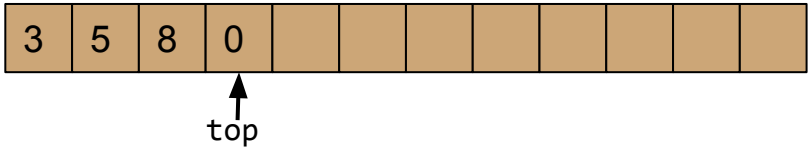
Att implementera en Stack

Vi kan börja med att titta på hur vi skulle kunna implementera en stack som en array.

Vi behöver:

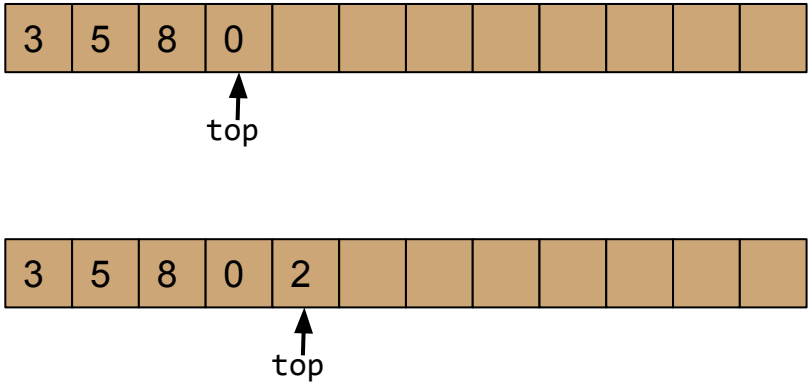
- En array som håller elementen
- En variabel som håller reda på vad som är toppen på stacken

Stack som en Array



- Med hjälp av ett index “top” kan vi hålla reda på vilket element som ligger överst på stacken
- Vi kan alltså representera stacken som en struct innehållande en array och ett index.
- Stacken består av elementen mellan index 0 och index top.
- Vi kan då göra:
  - `typedef struct stackarray Stack;`
  - Namnet “Stack” refererar nu till en struct

Stack som en Array



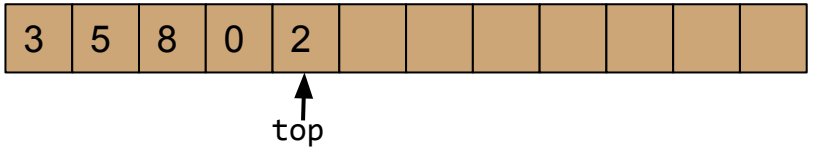
Efter operationen

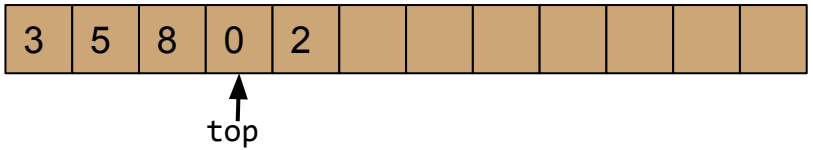
```
push(&stack, 2)
```

Algoritm:

1. Flytta fram `top`
2. Lägg till elementet

Stack som en Array





Efter operationen

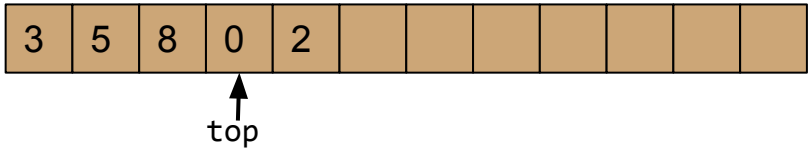
```
pop(&stack)
```

Det gör inget att 2an ligger kvar i arrayen. Bara de under “top” finns i stacken.

Algoritm:

1. Flytta tillbaka top ett steg

Stack som en Array



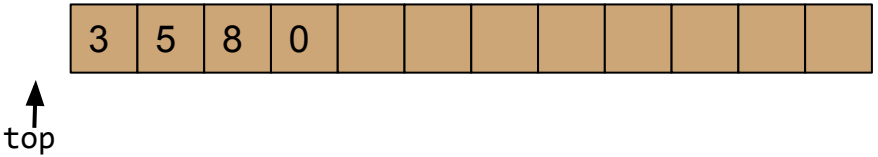
Operationen
`peek(stack)`

Kommer returnera `0`

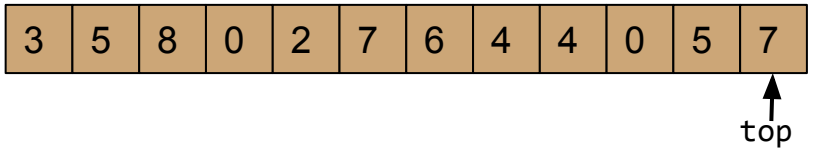
Algoritm:

1. Returnera elementet i arrayen som ligger på index “top”

Stack som en Array



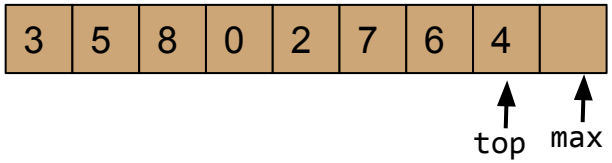
En stack är tom om
`top` är `-1`



En stack är full om
`top` är `MAXLENGTH-1`

Där `MAXLENGTH` är längden
på arrayen

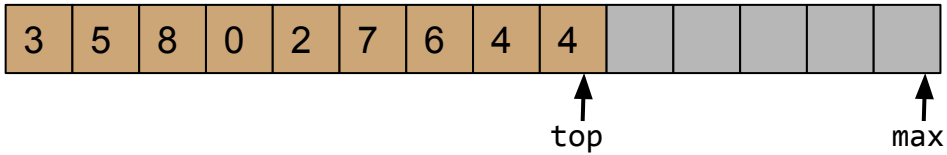
Stack som en Dynamisk Array



Alternativt använder vi en dynamisk array och allokerar fler element när vi kommer till slutet (om det behövs).

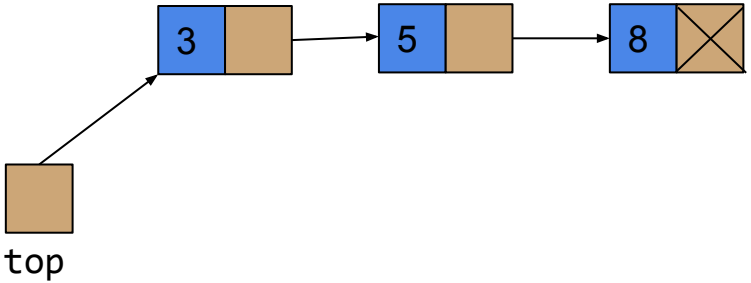
Då behöver vi hålla reda på max.

Stack som en Dynamisk Array



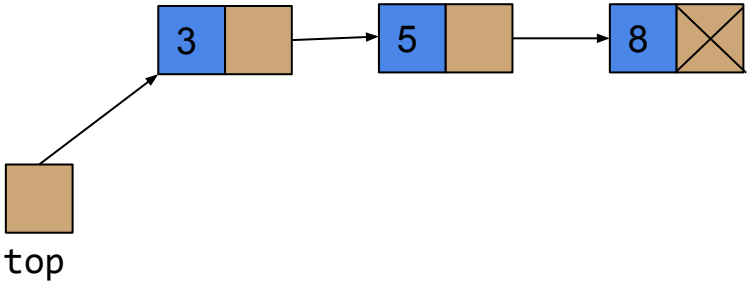
Om vi pushar ytterligare ett element (så att top==max) behöver vi göra en `realloc()` och flytta på `max`.

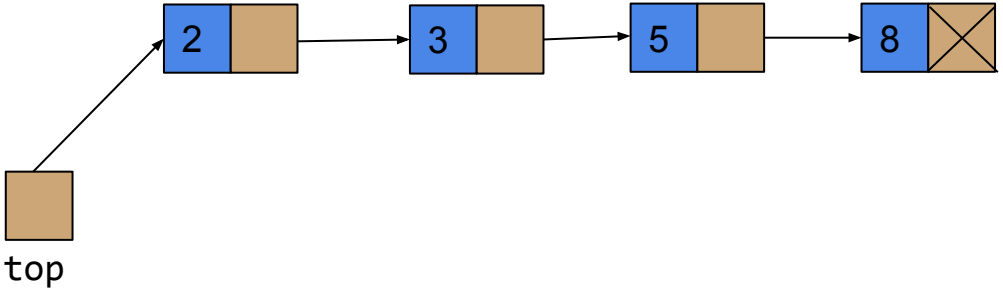
Stack som en länkad lista



- Listhuvudet blir toppen på stacken
- Vi kan skriva:
  - `typedef List Stack;`
  - Nu kommer namnet Stack att referera till en länkad lista

Stack som en länkad lista

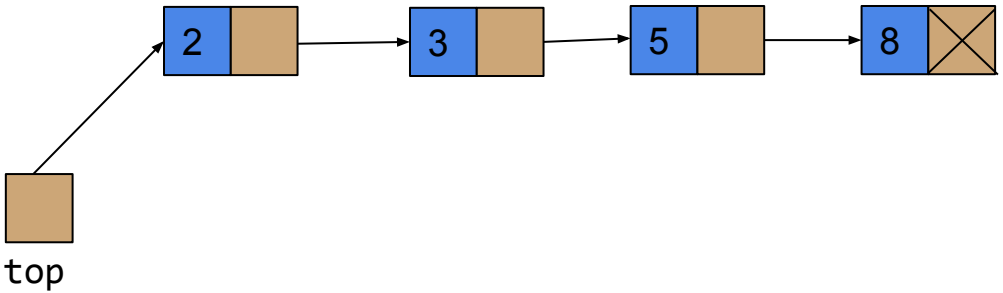


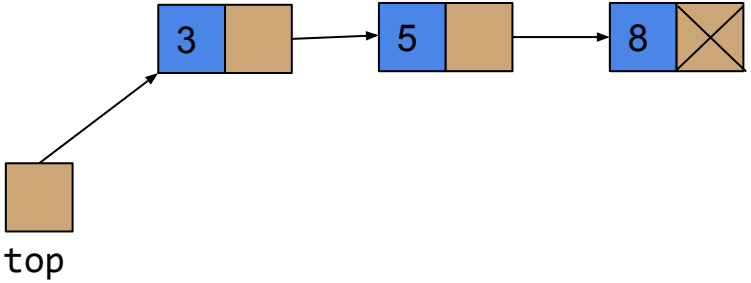


Efter operationen
`push(&stack, 2)`

- Sätt in element först i listan
- `addElementFirst(&stack, 2)`

Stack som en länkad lista

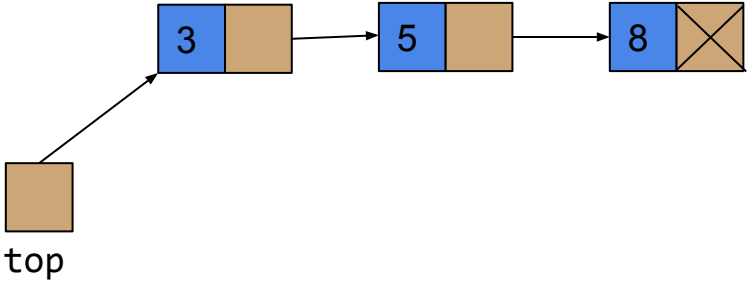




Efter operationen
`pop(&stack)`

- Ta bort första elementet i listan
- `removeFirstElement(&stack)`

Stack som en länkad lista



Efter operationen
`peek(stack)`

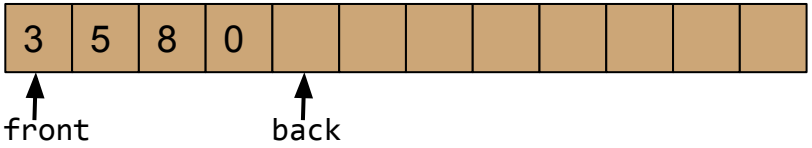
- Returnerar 3 (första elementet)

Stack som en länkad lista

- En tom stack == en tom lista

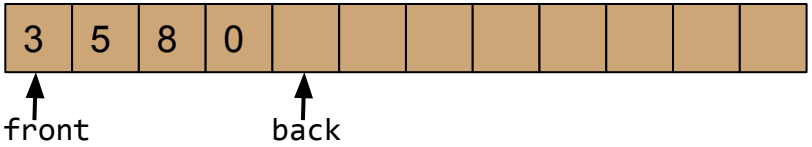
- En stack implementerad som en länkad lista kan inte bli full

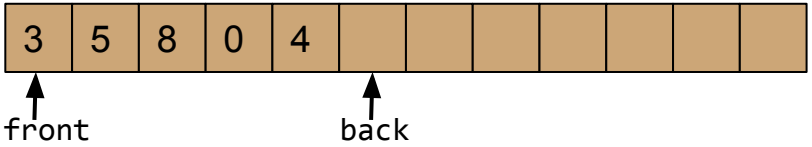
Kö som en Array



- En kö blir lite mer avancerad än en stack: vi behöver hålla reda på både början och slutet av en kö (`front` och `back`)
  - Detta för att element plockas in bakifrån och tas bort framifrån
  - Vi låter back peka på elementet efter sista platsen
- Vi behöver alltså en array och två index för att representera en kö
- Kön består av elementen mellan “front” och fram till elementet innan “back”
- Vi kan då göra:
  - `typedef struct queuearray Queue;`
  - Namnet “Queue” refererar nu till en struct

Kö som en Array





Operationen Enqueue:

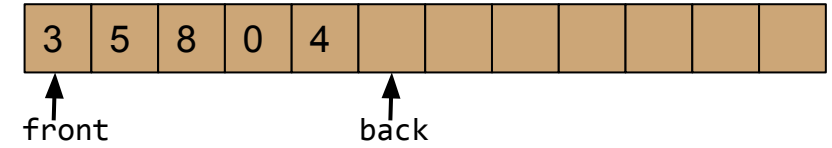
```
enqueue(&queue, 4);
```

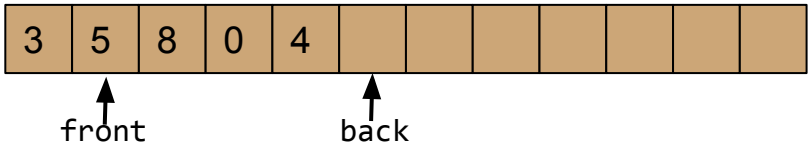
Algoritm:

1. Lägg elementet på plats “back”.
2. Flytta fram back

```
Kön innehåller efter anropet:
3 5 8 0 4
```

Kö som en Array





Operationen Dequeue:

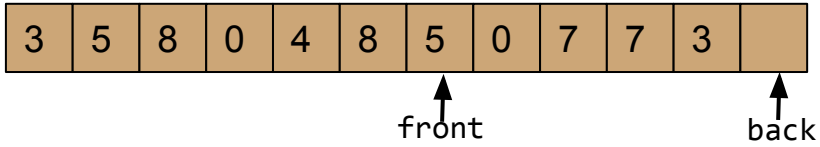
```
dequeue(&queue);
```

Algoritm:

1. Flytta fram “`front`”

Kön innehåller efter anropet:
5 8 0 4

Kö som en Array



Efter vi lagt till och tagit bort några element:

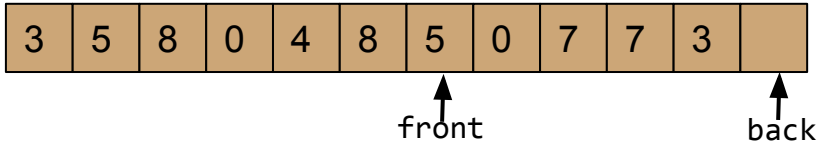
Arrayen är slut!

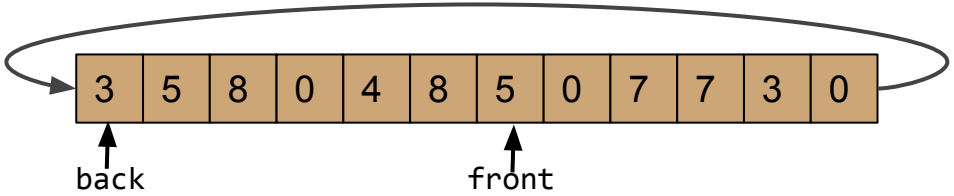
Kön består av:
5 0 7 7 3

Är kön full nu?

Hm.. arrayen har 12 platser, men kön innehåller bara 5 element...

Kö som en Array





Om vi i detta läge skriver:

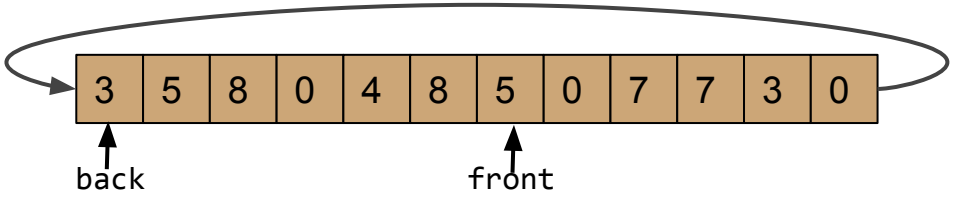
```
enqueue(&queue, 0);
```

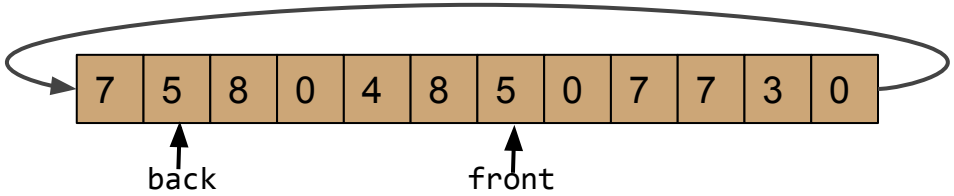
Vi flyttar tillbaka `back` till början!
Detta kallas för en cirkulär kö

Kön går fortfarande mellan `front` och `back`, men hoppar tillbaka när vi kommer till slutet av arrayen.

Kön har nu elementen:
5 0 7 7 3 0

Kö som en Array



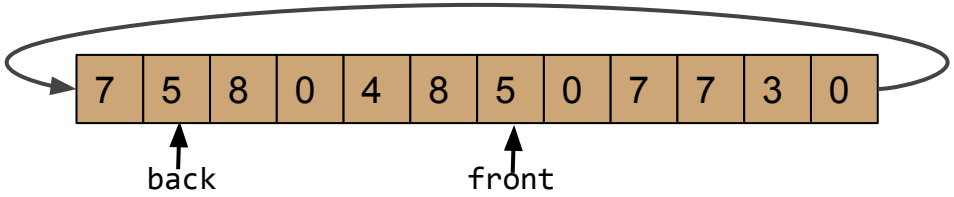


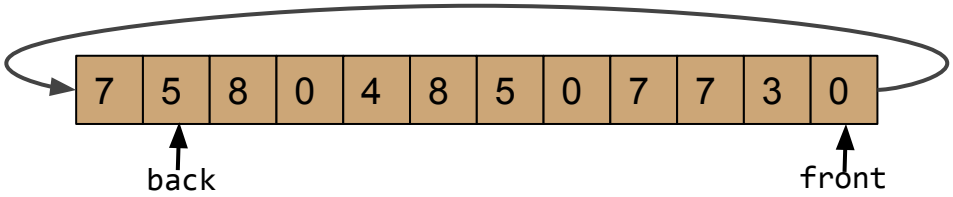
Om vi i detta läge skriver:

```
enqueue(&queue, 7);
```

Kön har nu elementen:
5 0 7 7 3 0 7

Kö som en Array





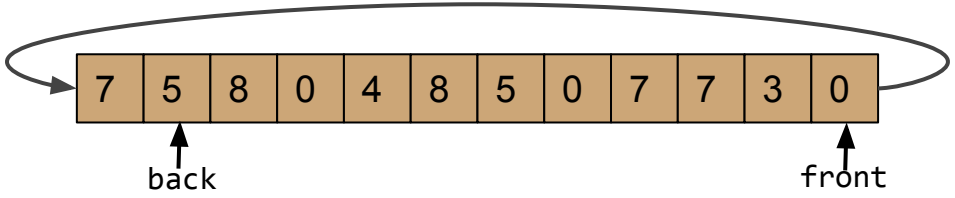
Det samma händer front när den kommer till slutet av arrayen.

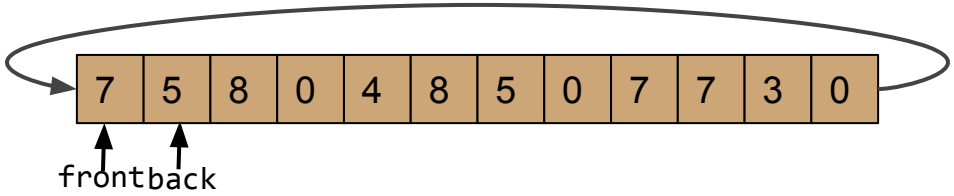
Om vi nu gör

```
for(i=0; i < 5; i++)
      dequeue(&queue);
```

Kön består nu av:
0 7

Kö som en Array





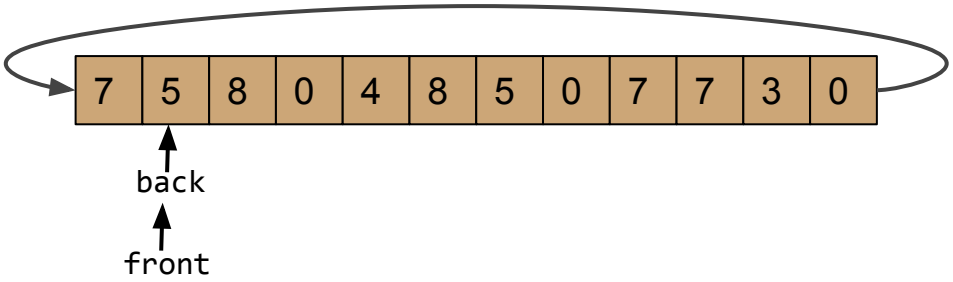
Vi gör tar nu bort ytterligare ett element:

```
dequeue(&queue);
```

Kön består nu av:
7

Även `front` hoppar tillbaka till början.

Kö som en Array



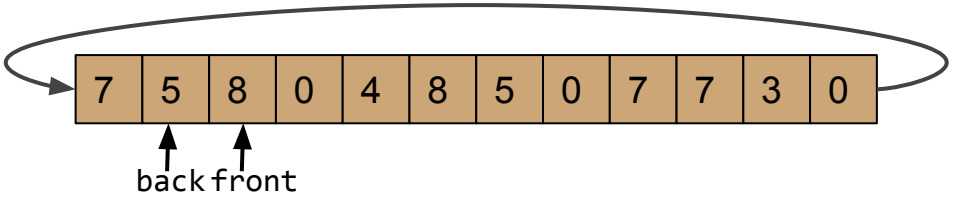
Vi tar nu bort det sista elementet:

```
dequeue(&queue);
```

Kön är nu tom.

Alltså gäller:
`front == back` betyder: kön är tom!

Kö som en Array



En kö är *full* om back är precis bakom front.

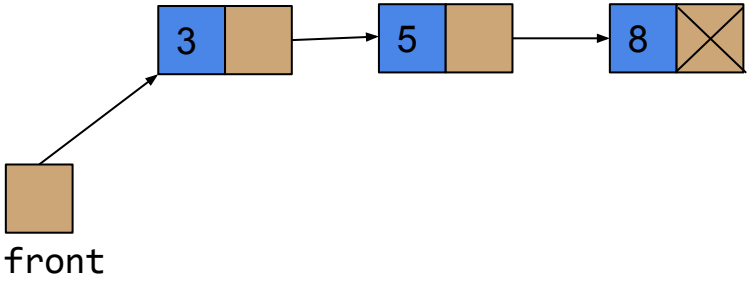
Förklaring:
`back` pekar på elementet bakom sista elementet. Om vi skulle lägga till ett element till så skulle `front == back` vilket betyder tom kö.

Kön är just nu:
8 0 4 8 5 0 7 7 3 0 7

Vi kan inte använda index [1] i arrayen nu för då skulle kön bli “tom”. Detta betyder att om vi har 12 platser i arrayen, kan kön bli maximalt 11 element lång.

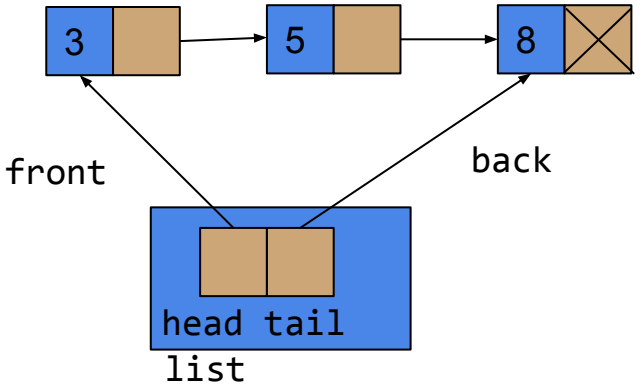
Vi måste alltså arrayen 1 element större än köns maximala längd!

Kö som en länkad lista



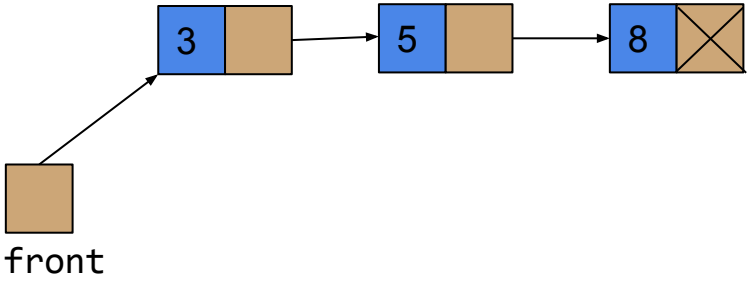
- En kö kan representeras som en länkad lista där första elementet är `front` och sista elementet är `back`.
- Vi kan skriva:
  - `typedef List Queue;`
  - Nu kommer namnet Queue att referera till en länkad lista

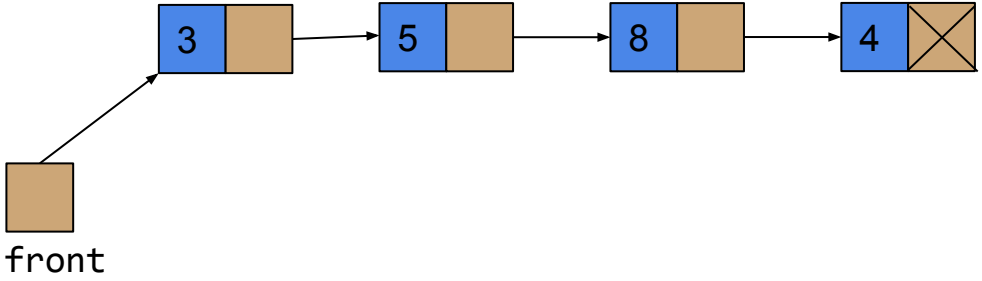
Kö som en länkad lista



- Om vi i vår list-typ använde `head` och `tail`-pekare så mappas de naturligt över som `front` och `back`-pekare.
- Detta är dock inte nödvändigt, men en bonus ifall man implementerat listan så

Kö som en länkad lista





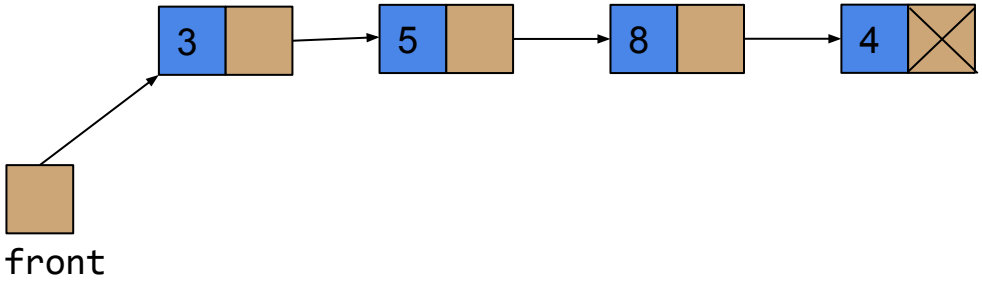
Operationen:

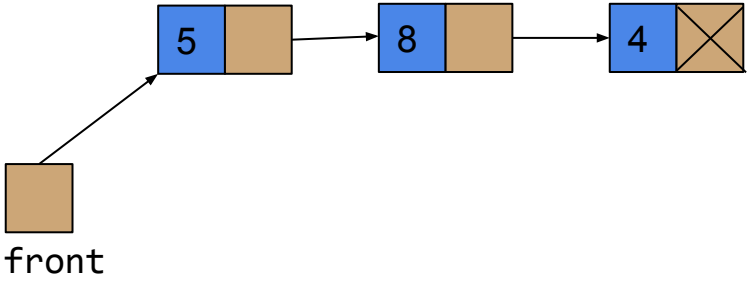
```
enqueue(&queue, 4);
```

Algoritm:

1. Lägg till elementet sist i listan

Kö som en länkad lista





Operationen:
`dequeue(&queue);`

Algoritm:
1. Ta bort det första elementet i listan

Operationen
`front(queue);`

Kommer (efter `dequeue()`) att returnera 5, eftersom 5 ligger först i kön.

Kö som en länkad lista

`front`

En kö är tom om listan är tom

En kö implementerad som en länkad lista kan inte bli full.

Implementation för Set

Precis som de andra ADTerna kan Set implementeras som en array eller en länkad lista. Eftersom positionen för element är irrelevant så behöver vi inte visualisera dessa.

Implementation för Set

```
void addElement(Set* set, const Data data);
```

Algoritm:

1. Kontrollera om data redan finns i listan/arrayen (loopa igenom alla element tills vi hittar det eller kommer till slutet)

Om ja: gör inget
Om nej: lägg till datat någonstans (spelar ingen roll var) i listan/arrayen.

Implementation för Set

```
void removeElement(Set* set, const Data data);
```

Algoritm:

1. Kontrollera om data redan finns i listan/arrayen

Om ja: ta bort datat
Om nej: gör inget.

Implementation för Set

```
int isInSet(const Set set, const Data data);
```

Algoritm:

Löp igenom hela listan/arrayen och returnera 1 ifall data hittades och 0 annars.