DVA104 VT19

Datastrukturer och Algoritmer

Stefan Bygde

Länkad Lista

- Vi ska gå igenom en ny datatyp: länkad lista
- En länkad lista är **inte** en ADT utan en konkret datatyp
- Denna datatyp kan **användas** för att implementera en ADT, som vi senare ska se.
- En länkad lista är ett alternativ till en dynamisk array, med egna för och nackdelar

Arrayer

Fördelar:

- Går väldigt snabbt att komma åt element (pekararitmetik)
- Går snabbt och lätt att allokera och avallokera
- Enkelt att använda

Nackdelar:

- Fast storlek
  - Går att ändra om arrayen är dynamisk: men då mha kopieringar och omallokering av minne, vilket är dyrt
- Kan ej “klämma in” element
  - För att lägga till ett element först eller i mitten måste element “flyttas”
  - Detta leder till många kopieringar, vilket är dyrt

Länkad Lista

Rekommenderad läsning!!
https://en.wikipedia.org/wiki/Linked_list

Fördelar:

- Kan sätta in element var som helst:
  - Först, sist och mitt-emellan. Snabbare än för arrayer.
- Kan växa och krympa fritt, ingen kopiering krävs
- Använder exakt så mycket minne som behövs

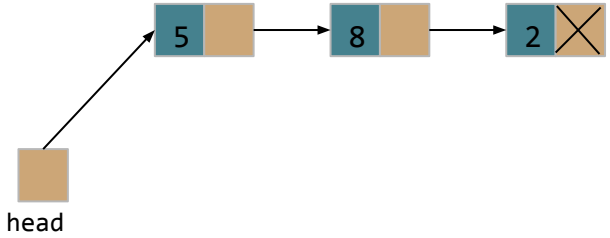
Nackdelar:

- Långsammare åtkomst till element mitt i listan (behöver “loopa”)
- Svårare att implementera
- Kan vara långsammare att lägga till många element i (jmf med arrayer)

Enkellänkad lista

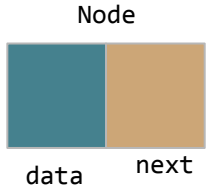
Arrayversion:

```
int arr[] = {5,8,2};
```



Länkad lista:

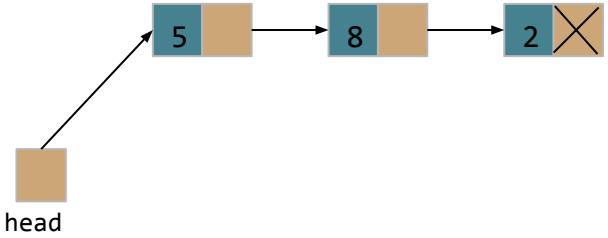
```
Node *head;
```



Enkellänkad lista

En länkad lista består av ett antal “noder” som är sammanlänkade. Varje nod kan innehålla ett värde (data).

Varje nod håller också reda på var “nästa” nod kan hittas.

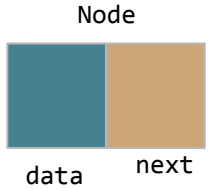


Arrayversion:

```
int arr[] = {5,8,2};
```

Länkad lista:

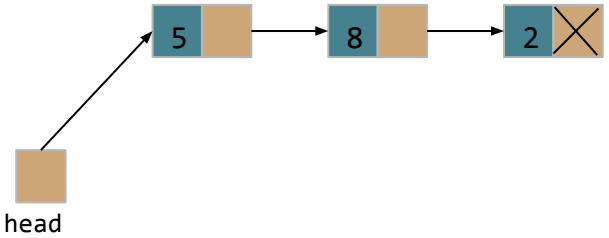
```
Node *head;
```



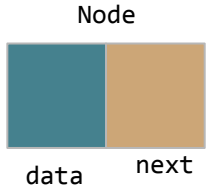
Enkellänkad lista

Vi kan enkelt ta bort eller lägga till noder till listan och “länka om” den så att den fortfarande är sammanhängande.

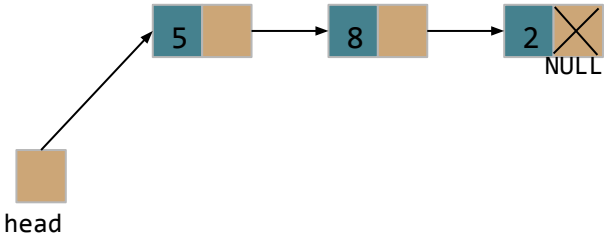
Listan håller vi reda på genom ett “handtag” som brukar kallas för “head”. Detta handtag är en *pekare* till en nod.



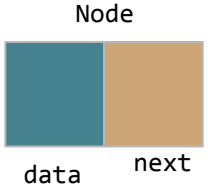
Noder är en *rekursiv* datastruktur: dvs, den definieras i termer av sig själv. En nod har nämligen en pekare som pekar på andra noder.



Enkellänkad lista



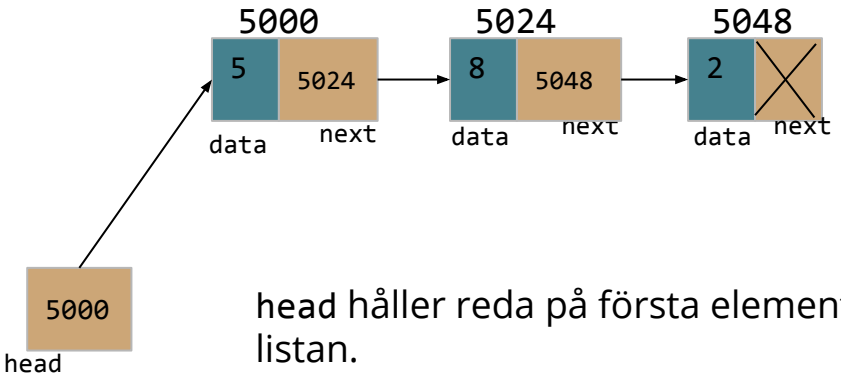
Vi refererar till listan via `head` som har typen `Node*`




```
typedef int Data;
struct node
{
   Data data;
   struct node* next; // Rekursiv definition!
};
typedef struct node Node;
```

Enkelt att ändra vilken typ av data vi sparar: döp om `Data` till något annat än **`int`**.

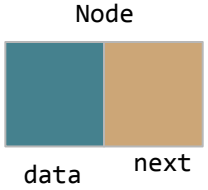
Enkellänkad lista



head håller reda på första elementet i listan.

Varje nod har sedan en “next”-pekare som håller reda på var nästa element är.

Den sista nodens “next”-pekare är NULL.




```
typedef int Data;
struct node
{
   Data data;
   struct node* next; // Rekursiv definition!
};
typedef struct node Node;
```

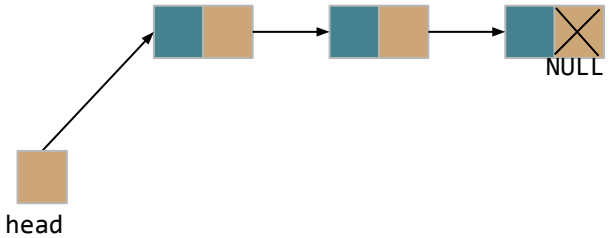
Enkelt att ändra vilken typ av data vi sparar: döp om `Data` till något annat än **`int`**.

Enkellänkad lista

Tom lista

Vi kan definiera en “tom lista” som en `NULL` pekare: En nodpekare som just nu inte pekar på en nod.

```
Node* head = NULL; // Skapar en tom lista
```



Lista med tre element

Det sista elementet i listan är alltid en NULL-pekare. Det är så vi vet att listan är slut.

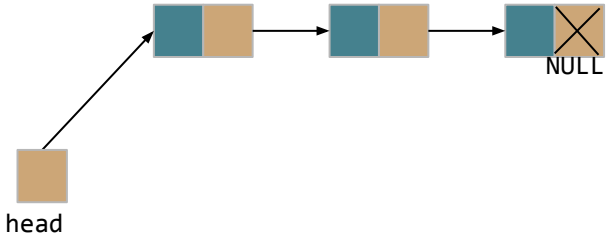
Listor byggs i allmähet från en tom lista som vi sedan lägger till element till, ett efter ett.

Enkellänkad lista

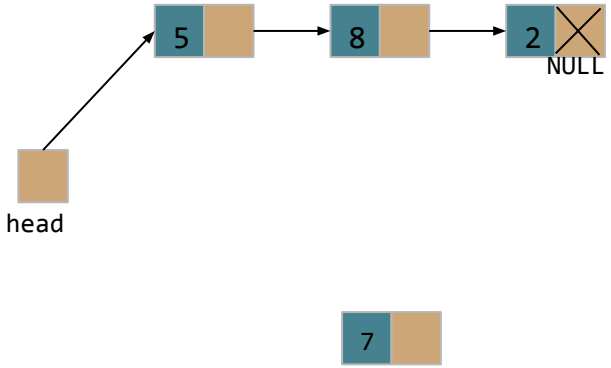
När vi har en lista så kan vi göra följande operationer på den:

- Lägga till en ny nod i början av listan
- Lägga till en ny nod i mitten eller slutet av listan
- Ta bort den första noden i listan
- Ta bort en nod i mitten eller slutet av listan

Notera att vi inte kan göra något av detta (enkelt) med arrayer.
Vi ska titta på exempel på hur vi kan göra alla dessa operationer.



Att lägga till ett element först



Låt oss säga att vi vill lägga till en 7:a i början av vår lista.

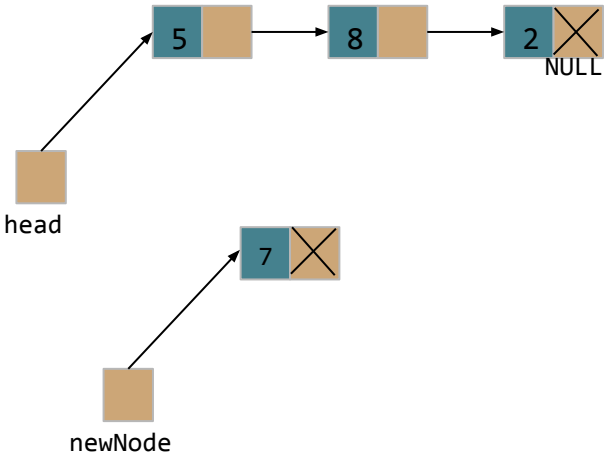
Vi kan göra det genom att:

1) Skapa en ny nod med datat 7
2) “Länka in” 7:an så att den kommer före femman.

Detta betyder att vi kommer att behöva:

1) Länka om head, så att den pekar på den nya noden (7) istället för 5.
2) Länka den nya noden (7) så att den pekar till 5:an.

Att lägga till ett element först



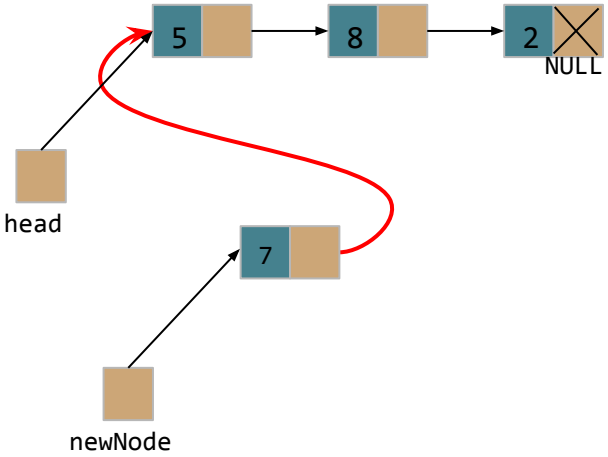
Vi börjar med att skapa en ny nod. Just nu antar vi att det finns en funktion `createNewNode()` som allokerar minne för en nod och returnerar en pekare till den.

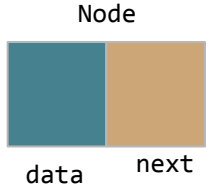
```
Node* newNode = createNewNode(7);
```

Notera att `createNewNode()` måste allokera noden *dynamiskt* (alltså via malloc/calloc) eftersom vi inte vet hur många noder som kommer att läggas till.

Detta betyder att vi kommer att behöva frigöra noderna när vi tar bort dem sen.

Att lägga till ett element först

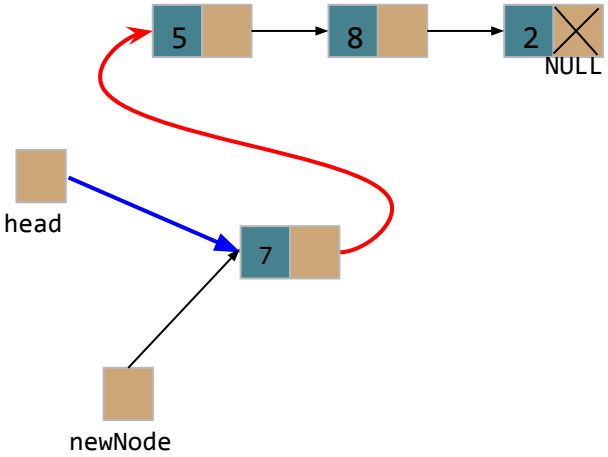




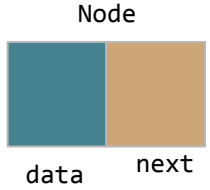

```
newNode->next = head;
```

Nästa steg är att länka denna nya nod så att den pekar på 5an.

Att lägga till ett element först

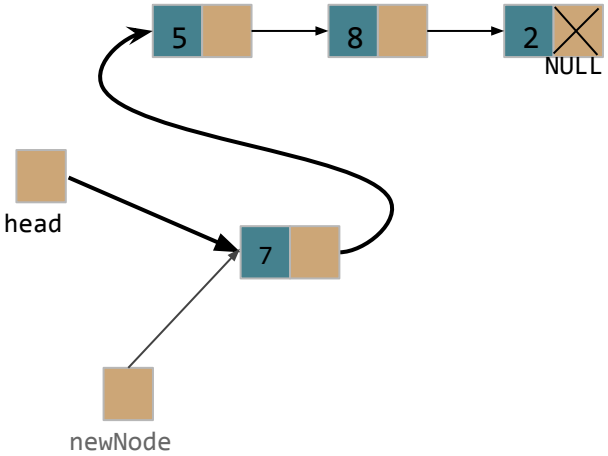



```
newNode->next = head;
head = newNode;
```



Sedan ska “head”, som är handtaget till vår lista peka på den nya noden (eftersom den ska in först).

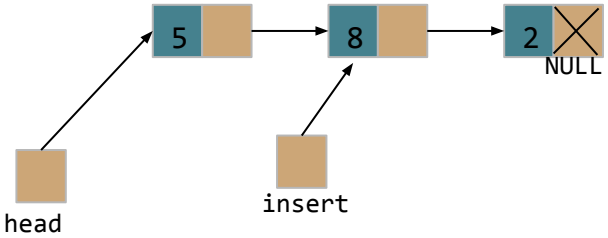
Att lägga till ett element först

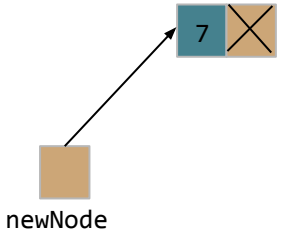



```
newNode->next = head;
head = newNode;
```

Vi är färdiga! Vi har länkat in den nya noden först i listan.

Att lägga till ett element som inte är först





Först behöver vi ha en pekare (`insert`) till den nod som vi vill sätta in den nya noden efter.

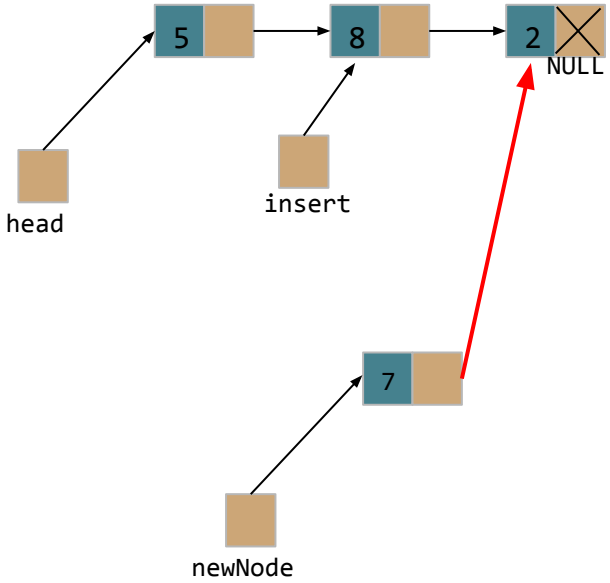
Hur vi hittar den beror på vad vi har för kriterier:

- Sätta in efter nod med data 8?
- Sätta in sist i listan? (hur vet vi att en nod är sist?)

Vi kan då flytta insert till rätt nod, antingen med en loop eller rekursion (mer om rekursion senare).
T.ex

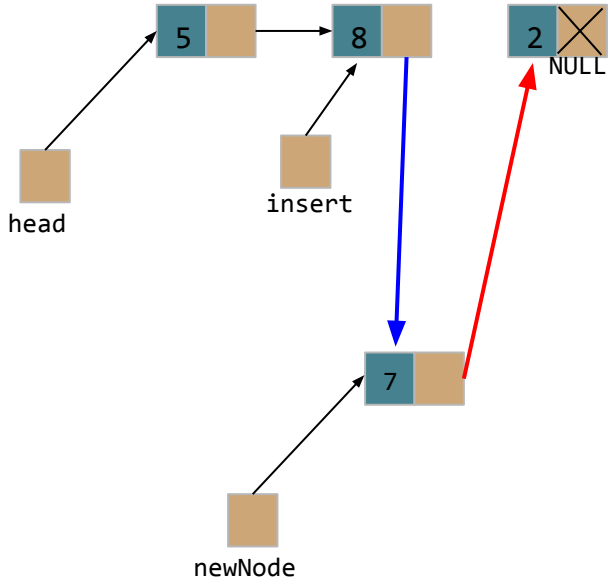
```
while(villkor)
     insert = insert->next;
```

Att lägga till ett element som inte är först



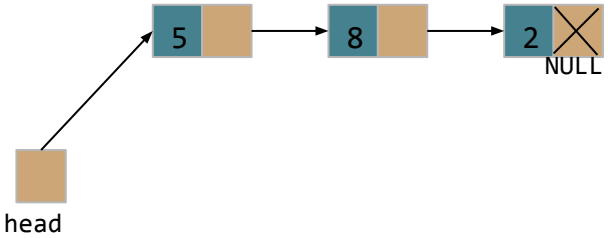

```
newNode->next = insert->next;
```

Att lägga till ett element som inte är först

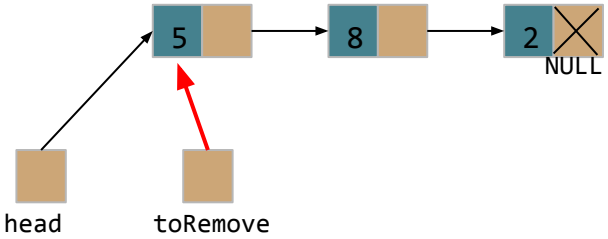



```
newNode->next = insert->next;
insert->next = newNode;
```

Att ta bort det första elementet

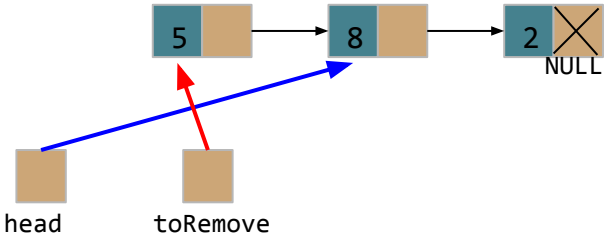


Att ta bort det första elementet




```
/* Vi behöver komma ihåg vilken nod vi ska
frigöra minne för */
Node* toRemove = head;
```

Att ta bort det första elementet

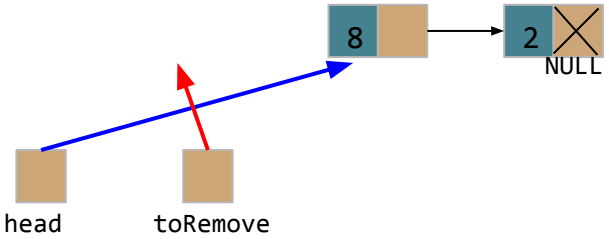



```
/* Vi behöver komma ihåg vilken nod vi ska
frigöra minne för */
Node* toRemove = head;
head = head->next;
```

Den nya början på listan kommer nu istället vara 8an.

Därför behöver vi låta `head` "hoppa över" det första elementet.

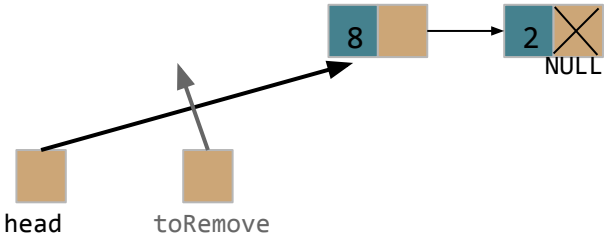
Att ta bort det första elementet




```
/* Vi behöver komma ihåg vilken nod vi ska
frigöra minne för */
Node* toRemove = head;
head = head->next;
free(toRemove);
```

Slutligen plockar vi bort och frigör minne för den första noden.

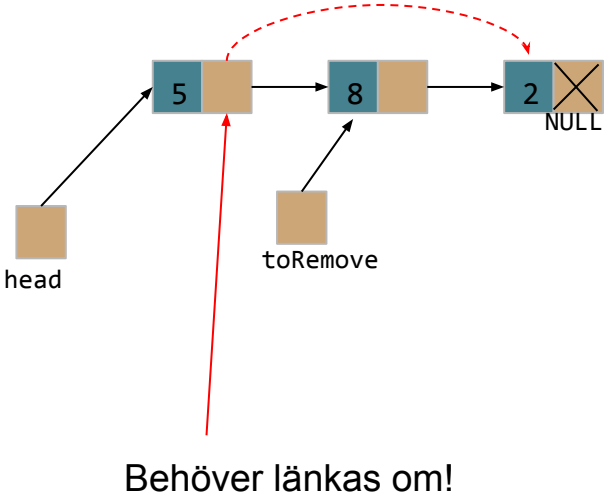
Att ta bort det första elementet




```
/* Vi behöver komma ihåg vilken nod vi ska
frigöra minne för */
Node* toRemove = head;
head = head->next;
free(toRemove);
```

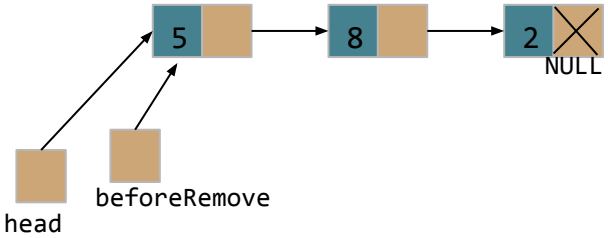
Vi är nu färdiga. Det första elementet har plockats bort.

Att ta bort ett element som inte är först



Om vi vill ta bort noden med element 8 så går det **inte** att leta upp en pekare dit. Eftersom vi behöver länka om noden **innan**.

Att ta bort ett element som inte är först



Vi måste alltså leta upp noden innan elementet vi vill ta bort.

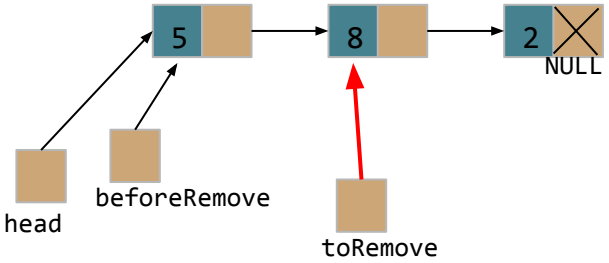
Det kan man göra genom att antingen “titta framåt i listan” eller genom att ha en pekare som ligger steget före.

Exempel:

```
while(beforeRemove->next->data == dataToRemove)
      beforeRemove = beforeRemove->next;
```

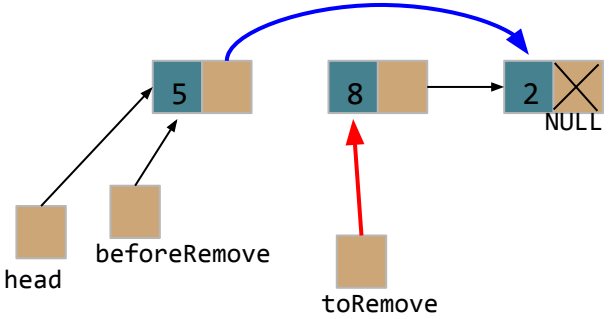
(notera att exemplet inte är fullständigt, vi måste också se till att vi inte har kommit till slutet av listan)

Att ta bort ett element som inte är först



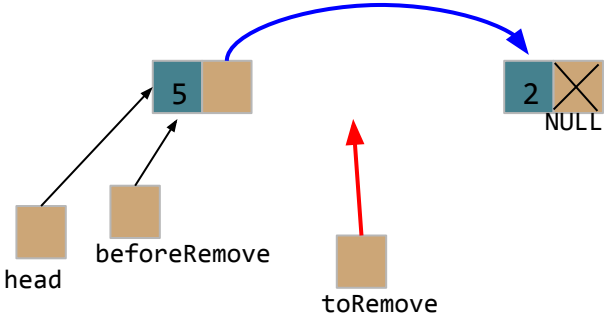

```
Node* toRemove = beforeRemove->next;
```

Att ta bort ett element som inte är först



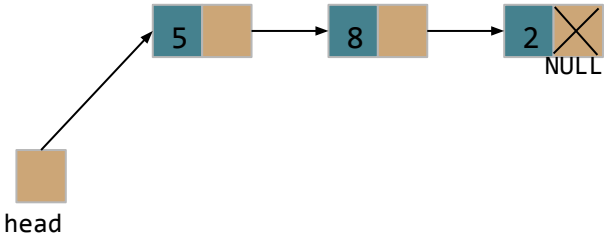

```
Node* toRemove = beforeRemove->next;
beforeRemove->next = toRemove->next;
```

Att ta bort ett element som inte är först




```
Node* toRemove = beforeRemove->next;
beforeRemove->next = toRemove->next;
free(toRemove);
```

Att representera en Lista

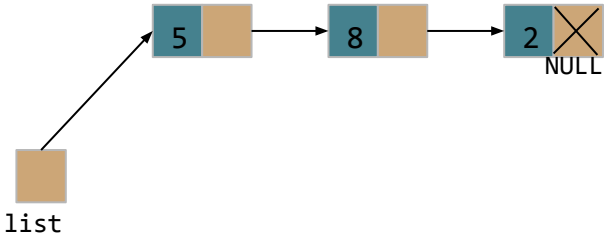


En lista representeras alltså som en pekare till en nod (eller en NULL-pekare om listan är tom).

Vi kan dock välja exakt hur vi representerar en list-typ på lite olika sätt.

På följande slides finns några olika sätt vi kan implementera vår lista.

Att representera en Lista



Representera en lista som en nodpekare

```
typedef Node* List;
```

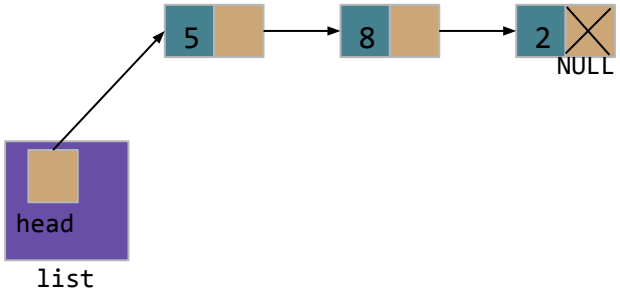
Fördelar:

- Enkel implementation
- Rekursiva egenskaper
  - `list->next` är också en lista

Nackdelar:

- Kräver användandet av dubbelpekare
- Konceptet av nod “exponeras”

Att representera en Lista



Representera en lista som en egen struct

```
struct list
{
    struct Node* head;
};
typedef struct list List;
```

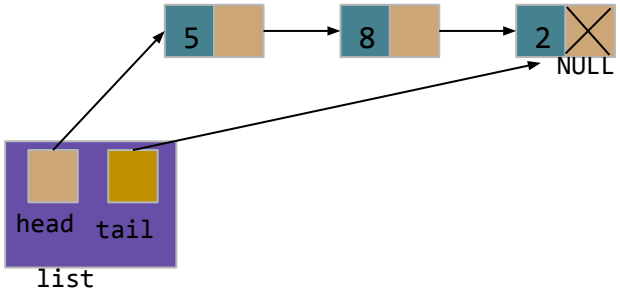
Fördelar:

- Får två olika datatyper för nod och lista, tillåter extra data så som t.ex listans längd
- Slipper dubbelpekare i listfunktionerna

Nackdelar:

- Två datatyper att hålla reda på
- Förlorar rekursiva egenskaper
  - `list.head->next` är inte en lista

Att representera en Lista



Representera en lista som en egen struct som håller reda på början och slutet

```
struct list
{
    struct Node* head;
    struct Node* tail;
};
typedef struct list List;
```

Fördelar:

- Snabbare tillgång till sista elementet

Nackdelar:

- Måste uppdatera två pekare i lägga till/ta bort-operationer

Se en lista som en rekursiv struktur

Om vi använder representationen “en nodpekare är en lista” så får vi följande egenskap.

En lista är antingen:

- Tom (basfall)
- Pekare till ett *data* och *en lista* (rekursivt fall)

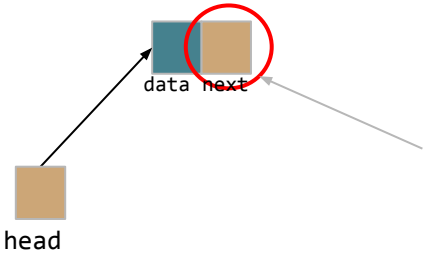
Eftersom varje nod har en nodpekare och “en nodpekare är en lista” så har alltså varje nod en lista!

Se en lista som en rekursiv struktur

En lista är antingen:

- Tom (basfall)

- Pekare till ett *data* och *en lista* (rekursivt fall)



`next` är också en lista!
(som i sin tur kan vara tom eller pekare till data och en lista).

Se en lista som en rekursiv struktur

Vi kan då enkelt göra väldigt korta och enkla rekursiva fall som bara tar hand om dessa två fall.

T.ex:

Låt oss säga att vi vill veta hur många element en lista har, dvs, hur lång den är. Vi vill skriva en funktion:

```
int listLength(List list)
```

(kom ihåg att `List` är en **`typedef`** av `Node*`)

Se en lista som en rekursiv struktur

Om vi ser listan som antingen tom eller som en pekare till ett data och en lista så behöver vi bara ta **hand om två fall**.

```
int listLength(List list)
{
    if(list==NULL) // Listan är tom
        return ???;
    else
        return ???; // Listan pekar en nod
}
```

Se en lista som en rekursiv struktur

Hur lång är listan om den är tom?

```
int listLength(List list)
{
    if(list==NULL) // Listan är tom
        return ???;
    else
        return ???; // Listan pekar en nod
}
```

Se en lista som en rekursiv struktur

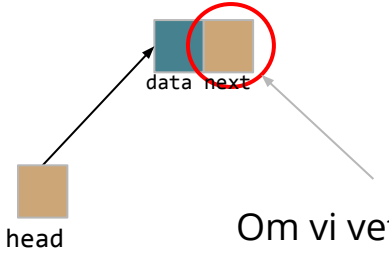
Hur lång är listan om den är tom?

```
int listLength(List list)
{
    if(list==NULL) // Listan är tom
        return 0; // Listan innehåller inga element, längden är 0
    else
        return ???; // Listan pekar en nod
}
```

Se en lista som en rekursiv struktur

Hur lång är listan om den är icke-tom? Hm… listan innehåller ju då minst en nod och har minst längden ett. Hur lång hela listan är beror ju på hur lång listan `next` är. Listan borde ju vara **en** längre än `next`-listan!

```
int listLength(List list)
{
    if(list==NULL) // Listan är tom
        return 0;
    else
        return ???; // Listan pekar en nod
}
```

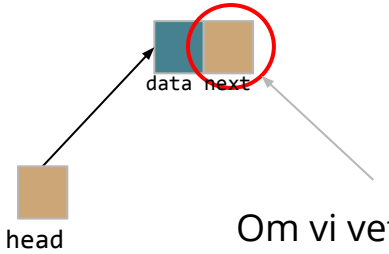


Om vi vet hur lång "next" är, så måste ju "head" vara en nod mer än det.

Se en lista som en rekursiv struktur

Hur lång är listan om den är icke-tom? Hm… listan innehåller ju då minst en nod och har minst längden ett. Hur lång hela listan är beror ju på hur lång listan `next` är. Listan borde ju vara **en** längre än `next`-listan!

```
int listLength(List list)
{
    if(list==NULL) // Listan är tom
        return 0;
    else
        return 1 + listLength(list->next);
}
```



Om vi vet hur lång “next” är, så måste ju “head” vara en nod mer än det.

Se en lista som en rekursiv struktur

Andra funktioner som fungerar väldigt bra att implementera rekursivt:

- Lägg till element efter data/sist i listan
- Ta bort ett element efter data/sist i listan
- Skriv ut listan
- Töm (frigör minne för) listan

För alla dessa finns två fall:

- Listan är tom
- Listan är ett data och en lista: `next`

Se en lista som en rekursiv struktur

- Exempel: lägg till ett element sist.
  - Antag att vi har en funktion `addFirst(List* plist, Data data)`

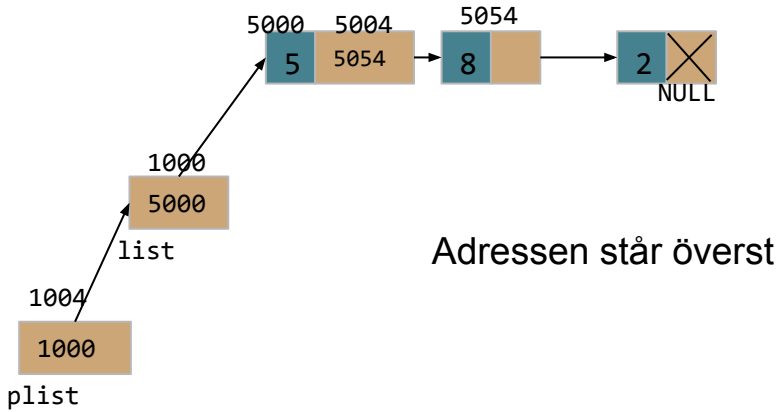
Då kan vi skriva:

Kom ihåg: `List` är en **`typedef`** för `Node*` så `List*` blir en `Node**`.

```
void addLast(List* plist, Data data)
{
	if(*plist == NULL)
		addFirst(list, data); // Basfall: om denna lista är tom, lägg elementet först
	else
		addLast(&(*plist)->next, data); // Rekursivt fall (se mer nästa slide)
}
```

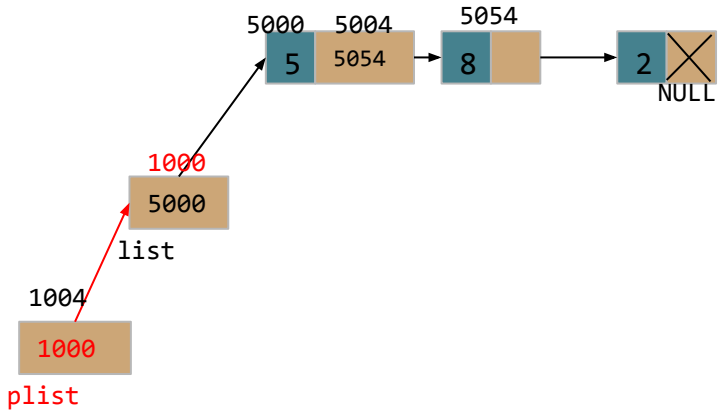
Se en lista som en rekursiv struktur

- Vänta, va?
  - `&(*plist)->next ???`



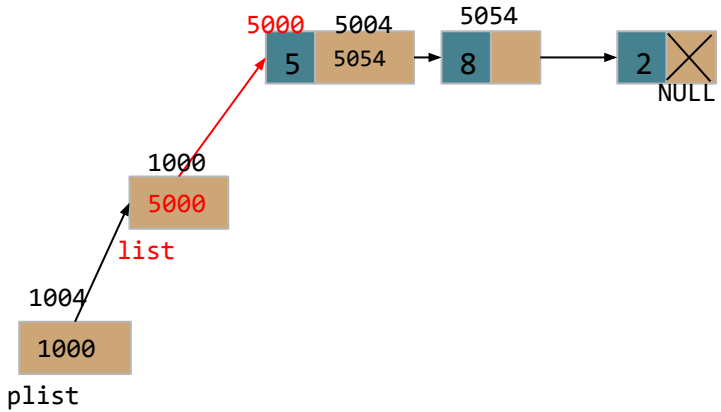
Se en lista som en rekursiv struktur

- Vänta, va?
  - `&(*plist)->next ???`
  - `plist == 1000 (Node**)`



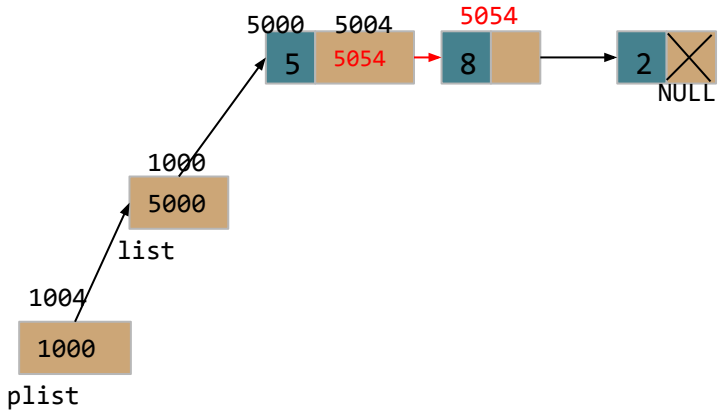
Se en lista som en rekursiv struktur

- Vänta, va?
  - `&(*plist)->next ???`
  - `plist == 1000 (Node**)`
  - `*plist == list == 5000 (Node*)`



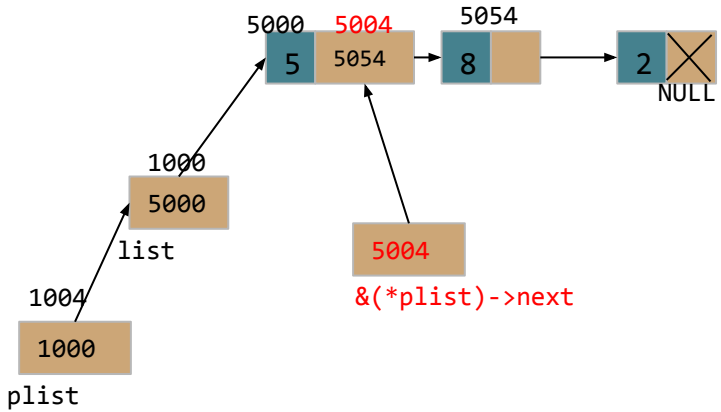
Se en lista som en rekursiv struktur

- Vänta, va?
  - `&(*plist)->next ???`
  - `plist == 1000 (Node**)`
  - `*plist == list == 5000 (Node*)`
  - `(*plist)->next == 5054 (Node*)`



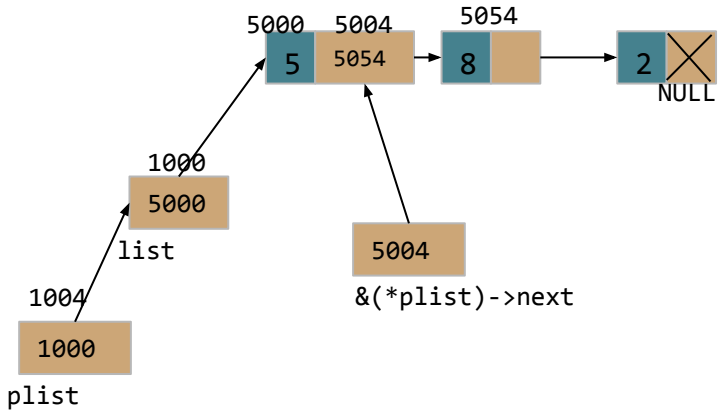
Se en lista som en rekursiv struktur

- Vänta, va?
  - `&(*plist)->next ???`
  - `plist == 1000 (Node**)`
  - `*plist == list == 5000 (Node*)`
  - `(*plist)->next == 5054 (Node*)`
  - `&(*plist)->next == 5004 (Node**)`



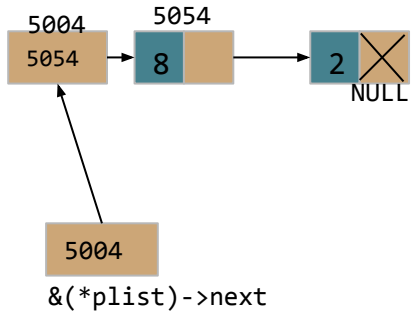
Se en lista som en rekursiv struktur

- Vänta, va?
  - `&(*plist)->next`
  - `plist == 1000 (Node**)`
  - `*plist == list == 5000 (Node*)`
  - `(*plist)->next == 5054 (Node*)`
  - `&(*plist)->next == 5004 (Node**)`



- `plist` är en dubbelpekare till första noden, `&(*plist)->next` är en dubbelpekare till andra noden

Se en lista som en rekursiv struktur



- Vänta, va?
  - `&(*plist)->next`
  - `plist == 1000 (Node**)`
  - `*plist == list == 5000 (Node*)`
  - `(*plist)->next == 5054 (Node*)`
  - `&(*plist)->next == 5004 (Node**)`
- `plist` är en dubbelpekare till första noden, `&(*plist)->next` är en dubbelpekare till andra noden