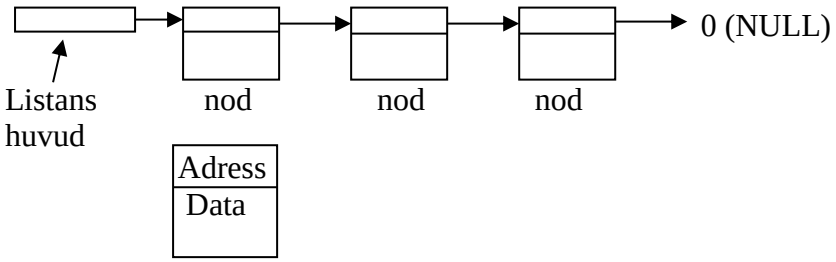
Innehållsförteckning

7 Länkade listor

Vi har tidigare arbetat med listor i form av arrayer. En array har ett bestämt antal element vilket kan vara en nackdel om man på förhand inte vet hur många element som ska läggas in i arrayen. En lösning på detta problem är att man använder en länkad lista. En länkad lista består av en eller flera noder som är länkade till varandra med pekare. I varje nod finns det data, motsvarande det som ligger i arrayens element, och en pekare till nästa nod. Listan börjar med ett huvud som innehåller en pekare till listans första nod. Det kan se ut så här:



Vi ska i detta avsnitt ta upp

- enkellänkad lista
- göra en klass som hanterar en enkellänkad lista
- dubbellänkad lista (i en klass)
- stack med en enkellänkad lista
- kö med en enkellänkad lista

7.1 Enkellänkad lista

7.1.1 Nod

Vi gör en klass, en egen datatyp, för att skapa en nod. Noden ska innehålla data och en pekare till en nod, se figuren ovan. I det här fallet låter vi för enkelhetens skull data vara ett heltal, men data kan vara av godtycklig datatyp.

Klassen får följande utseende

```
class Node
{
  public:
    Node *next;
    int data;
    Node(Node *n, int d) : next(n), data(d) // Konstruktor
    {}
};
```

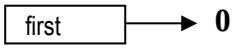
Kommentarer:

- Alla medlemmar görs *public* för att de ska bli enklare nu. Vi åtgärdar detta längre fram.
- Klassen har en datamedlem som ska peka på nästa nod, som ju också har datatypen Node.
- Konstruktorn initierar pekaren och data med en initieringslista.

7.1.2 Skapa en lista

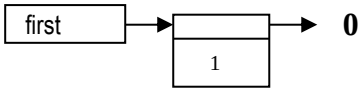
Listans huvud:

```
Node *first = 0;
```



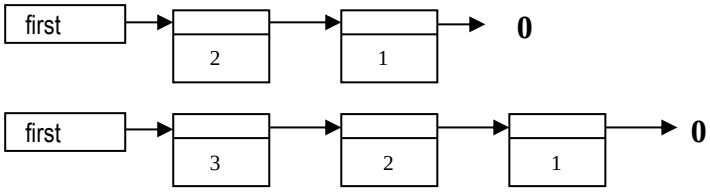
Första elementet:

```
first = new Node(first, 1);
```



Andra och tredje elementen:

```
first = new Node(first,2);
first = new Node(first,3);
```

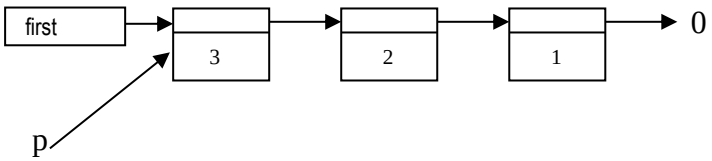


Kommentarer:

- `Node *first = 0` skapar en pekare som senare ska peka på en `Node`.
- `first = new Node(first,1);` skapar
  - plats för en ny Node.
  - tilldelar värdet 1 till datamedlemmen.
  - låter dess pekare få samma värde (adress) som first har, dvs 0 (NULL).
  - ger first adressen till denna nya nod.
- `first = new Node(first,2);` skapar
  - plats för en ny Node.
  - tilldelar värdet 2 till datamedlemmen.
  - låter dess pekare få samma värde som first har, dvs den pekar på noden med data = 1.
  - ger first adressen till denna nya nod.
- Observera att en ny nod läggs **närmast** listhuvudet.

7.1.3 Utskrift av listan

För att skriva ut listan låter vi en hjälppekare (p) stega genom listan nod för nod. För varje nod skrivs nodens data ut. Stegningen avbryts då pekaren får värdet p=0 (NULL), d.v.s. då den pekar "bortom slutet".




```
for(Node *p=first; p ; p=p->next)
    cout << p->data << endl;
```

Kommentarer:

- *p= first gör så att p pekar på första noden i listan
- Fortsättningskriteriet är p! När p kommer till slutet av listan blir p=0, d.v.s. falskt och då avbryts loopen. Alla övriga värden (=adresser) för p är sanna!
- p=p->next medför att p pekar på nästa nod i listan.
- OBS! Det går inte att skriva p++ eftersom listan **inte** är kontinuerlig.

7.1.4 Frigör allokerad plats

När den länkade listan skapas allokerar man plats för noderna med *new*. För att undvika minnesläckage måste man komma ihåg att frigöra denna plats. Ett sätt att göra detta är att göra så:

```
//-----------------------------------------------------------------------------
// Frigör allokerad plats
//-----------------------------------------------------------------------------
  while(first!=0)
  {
    Node *tmp = first;
    first = first->next;
    delete tmp;
 }
```

Kommentarer:

- Låt tmp peka på first.
- Flytta first till nästa node med first=fist->next
- Frigör platsen som tmp pekar på med `delete tmp`
- Fortsätt så länge first!=0

7.1.5 Programexempel med länkad lista

```
// list_010 Version 10
// Enkellänkad lista
// Per Ekeroot 2014-01-13
//-----------------------------------------------------------------------------
#include <iostream>
using namespace std;
//-----------------------------------------------------------------------------
// Definiera en klass som representerar en nod i en enkellänkad lista
//-----------------------------------------------------------------------------
class Node
{
  public:
    Node *next;
    int data;
    Node(Node *n, int d) : next(n), data(d)
    {}
};
//-----------------------------------------------------------------------------
int main()
{
//-----------------------------------------------------------------------------
// Skapa listhuvud
//-----------------------------------------------------------------------------
  Node *first=0;

//-----------------------------------------------------------------------------
// Lägg till tre element i listan
//-----------------------------------------------------------------------------
  first = new Node(first,1);
  first = new Node(first,2);
  first = new Node(first,3);

//-----------------------------------------------------------------------------
// Skriv ut listan
//-----------------------------------------------------------------------------
  for(Node *p=first; p ; p=p->next)
    cout << p->data << endl;
```

```
//--------------------------------------------------------------------------
// Frigör allokerad plats
//--------------------------------------------------------------------------
  while(first!=0)
  {
    Node *tmp = first;
    first = first->next;
    delete tmp;
  }
  return 0;
}
```

7.1.6 Sökning i listan

Sökning i en länkad lista kan göras med följande funktion:

```
Node *search(Node *first, int searchItem)
{
  Node *p;
  for(p=first; p && p->data!=searchItem; p=p->next)
    ;
  return p;
}
```

Kommentarer:

- Se figuren för utskrift av listan på sidan 3.
- **Fortsättnings**villkoret **`p&&p->data!=searchItem`** innebär att p sätts att peka på nästa nod om listan inte är slut **och** sökt data ej har hittats.
- Observera att ordningen är **`p && p->data!= searchItem`**!! Det blir exekveringsfel (allokeringsfel) om ordningen kastas om och sökt data inte finns i listan!
- Observera också att for-loopen saknar den sats som ska utföras för varje varv i loopen. Sökningen görs i for-loopens "huvud" och sökningens resultat finns i pekaren **p**. **p** pekar på den nod i vilken sökt data finns.
- search() returnerar en pekare till sökt data.
- Om sökt data saknas returneras p=0 (NULL).

Vi använder funktionen ovan för att söka i en länkad lista:

```
Node *s = 0;
  s = search(first,3);
  if(s !=0)
    cout << "Sökt data: " << s->data;
  else
    cout << "Data saknas i listan!" << endl;
```

Kommentarer:

- s är en pekare till en Node. s pekar på den nod som innehåller sökt data
- om s == 0 misslyckades sökningen
- använd s->data för att skriva ut funnen data.

7.1.7 Programexempel med sökning i länkad lista

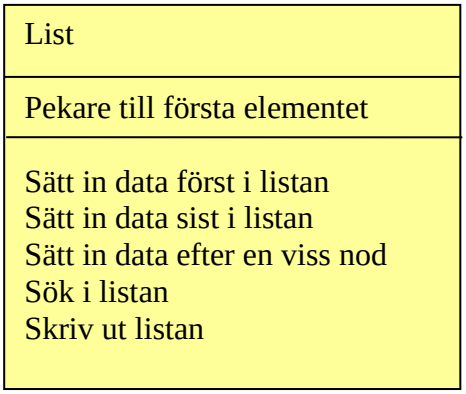
```
// list_020 Version 10
// Sökning i en enkellänkad lista
// Per Ekeroot 2014-01-13
//-----------------------------------------------------------------------------
#include <iostream>
using namespace std;

//-----------------------------------------------------------------------------
// Definiera en klass som representerar en nod i en enkellänkad lista
//-----------------------------------------------------------------------------
class Node
{
  public:
    Node *next;
    int data;
    Node(Node *n, int d) : next(n), data(d)
    {}
};
//-----------------------------------------------------------------------------
// Funktionsprototyp för search()
//-----------------------------------------------------------------------------
Node *search(Node *first, int searchItem);

//-----------------------------------------------------------------------------
// Huvudprogram
//-----------------------------------------------------------------------------
int main()
{
//-----------------------------------------------------------------------------
// Skapa listhuvud
//-----------------------------------------------------------------------------
  Node *first=0;

//-----------------------------------------------------------------------------
// Lägg till tre element i listan
//-----------------------------------------------------------------------------
  first = new Node(first,5);
  first = new Node(first,3);
  first = new Node(first,7);

//-----------------------------------------------------------------------------
// Skriv ut listan
//-----------------------------------------------------------------------------
  for( Node *p=first; p  ; p=p->next)
    cout << p->data << endl;

//-----------------------------------------------------------------------------
// Sök i listan
//-----------------------------------------------------------------------------
  Node *s = 0;
  s = search(first,6);
```

```
//-----------------------------------------------------------------------------
// Skriv sökresultat
//-----------------------------------------------------------------------------
  cout << endl;
  if(s !=0)
    cout << "Found: " << s->data;
  else
    cout << "The number you looked for is not in the list!" ;

  cout << endl << endl;

//-----------------------------------------------------------------------------
// Frigör allokerad plats
//-----------------------------------------------------------------------------
  while(first!=0)
  {
    Node *tmp = first;
    first = first->next;
    delete tmp;
  }
  return 0;
}

//-----------------------------------------------------------------------------
// Funktionsdefinition för search()
//-----------------------------------------------------------------------------
Node *search(Node *first, int searchItem)
{
  Node *p;
  for(p =first;  p && p->data!=searchItem ; p=p->next)
    ;
  return p;
}
```

7.2 Enkellänkad lista i en klass

I föregående avsnitt har vi sett hur man gör för att lägga till element först i listan, skriva ut listan och söka i listan. Vi ska nu göra en abstrakt datatyp av en enkellänkad lista genom att konstruera en klass som hanterar listan.

7.2.1 Klassdefinition



Vi ska använda klassen Node från föregående avsnitt för att definiera listans noder. Förslag på klassdefinition för listan(den finns i list.h):

```
class Node;   // Forward deklaration
class List
{
  private:
    Node *first;
    Node *search(int searchItem) const;

  public:
    List();
    ~List();
    void putFirst(int d);
    void putLast(int d);
    void insAfter(int searchItem, int d);
    void showList() const;
    bool searchFor(int searchItem) const;
};
```

Kommentarer:

- Klassen *List* använder klassen *Node* och klassen *Node* använder klassen *List*, vilket innebär att båda klasserna måste definieras först!! Man löser detta dilemma genom att göra en forward-deklaration av *Node*-klassen. Detta innebär att man talar om för kompilatorn klassen *Node* definieras senare, vilket kompilatorn godtar.
- Klassen *List* innehåller datamedlemmen *Node* *first, dvs en pekare till listan
- Klassen *List* innehåller medlemsfunktioner för att sätta in data först, sist och på önskat ställe, samt sök- och utskrifts funktioner
- Klassens konstruktor skapar ett listhuvud
- Klassens destruktor frigör allokerad plats
- Klassen *List* använder klassen *Node* för att hantera noderna
- Node definieras i implementationsfilen, list.cpp. Detta gör att den inte är åtkomlig utanför klassen *List.*
- search() är en privat medlemsfunktion. Se den som en hjälpfunktion i klassen. Den går inte att använda utanför klassen även om den är **public** eftersom klassen *Node* inte är tillgänglig utanför *List*

Innan vi implementerar medlemsfunktionerna i *List* upprepar vi definitionen av *Node* med ett tillägg som gör att klasserna *List* och *Node* kan kommunicera med varandra.

```
class Node
{
  friend class List;
  Node *next;
  int data;
  Node(Node *n, int d) : next(n), data(d)
  {}
};
```

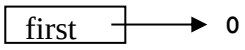
Kommentarer:

- *Node's* medlemmar är **private** eftersom inget annat anges.
- Man låter *List* vara vänklass till *Node* så att *List's* medlemmar kommer åt medlemmarna i *Node*.
- Förutom detta ser *Node* ut som tidigare.

7.2.2 Implementation av medlemsfunktionerna i List

Förvald konstruktor:

```
List::List()
{
  first = 0;
}
```



Kommentarer:

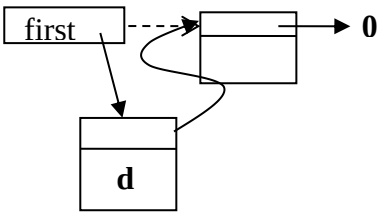
- Listhuvudet skapas genom att pekaren till listan (first) sätts att peka på 0 (NULL)

Sätt in data först i listan:

```
void List::putFirst(int d)
{
  first = new Node(first,d);
}
```

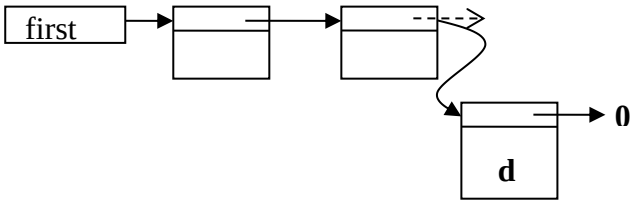
Kommentarer:

- En ny nod skapas närmast listhuvudet
- Plats allokeras på heapen
- Data (d) läggs in



Sätt in data sist i listan:

```
void List::putLast( int d)
{
  if(first==0)
    first = new Node(first,d);
  else
  {
     Node *p;

     for(p=first; p->next; p=p->next) ;
     p->next = new Node(0,d);
  }
}
```



Kommentarer:

- Om listan är tom läggs data närmast listhuvudet annars söks den nod som ligger sist i listan upp, och den nya noden placeras efter denna
- OBS! Fortsättningsvillkoret är **p-> next** och inte p som förut!
- Noden som tidigare låg sist sätts att peka på den nya noden och nya noden pekar på 0 (NULL).

Sök i listan efter noden med data = searchItem:

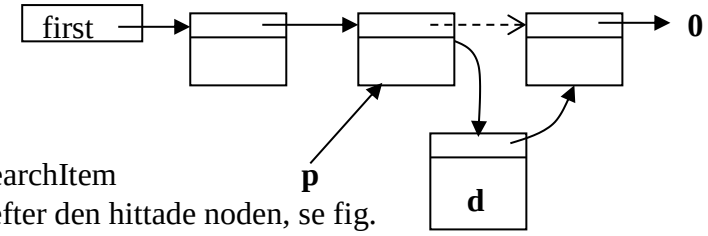
```
Node *List::search(int searchItem) const
{
  Node *p;
  for(p=first; p && p->data != searchItem; p=p->next) ;
  return p;
}
```

Kommentarer:

- Funktionen är en intern hjälpfunktion i klassen List, används i "insAfter()"
- Funktionen returnerar en pekare till funnen nod
- Om sökningen misslyckas returneras 0.

Sätt in data efter nod vars data har värdet 'searchItem':

```
void List::insAfter(int searchItem, int d)
{
  Node *p = search(searchItem);
  if(p!=0)
    p->next = new Node(p->next, d);
}
```



Kommentarer:

- Sök i listan efter data med värdet searchItem
- Om sökt data finns sätts en nod in efter den hittade noden, se fig.

Skriv ut listan:

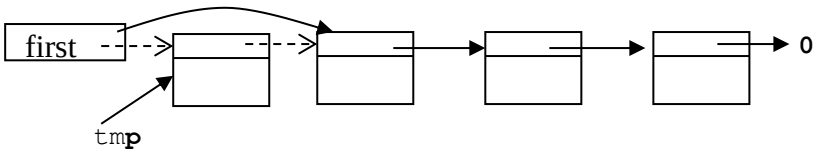
```
void List::showList() const
{
  for(Node *p=first; p  ; p=p->next)
    cout << setw(4) << p->data;
    cout << endl;
}
```

Kommentarer:

- Stega igenom listan och skriv ut data för resp. nod

Destruktor som frigör allokerat minne:

```
List::~List()
{
  while(first !=0)
  {
    Node *tmp= first;
    first = first->next;
    delete tmp;
  }
}
```



Kommentarer:

- Låt tmp peka på first.
- Flytta first till nästa node med first=fist->next
- Frigör platsen som tmp pekar på med `delete tmp`
- Fortsätt så länge first!=0

Sök i listan med data = searchItem:

```
bool List::searchFor(int searchItem) const
{
   Node *p = search(searchItem);
   return p!=NULL;
}
```

Kommentarer:

- Sök efter en nod som innehåller data = searchItem.
- Returnera **true** om data hittas, annars returneras **false.**
- Använd den privata hjälpfunktionen search() för att utföra sökningen. search() göms inne i searchFor() eftersom det inte går att returnera en pekare till en nod utanför klassen. Detta beror på att *Node* enbart är åtkomlig inom *List*.

7.2.3 En applikation som använder klassen List

Skapa en tom lista:

```
  List list;
```

Lägg till värden först i listan och skriv ut:

```
  for(int i=0; i < 4; i++)
    list.putFirst(random(10));
  list.showList();
```

Sätt in ett värde sist i listan:

```
  list.putLast(34);
  list.showList();
```

Sätt in en nod med värdet 13 efter noden med värdet 4:

```
  list.insAfter(4,13);
```

Sök efter värdet 13 listan och skriv ett meddelande som talar om ifall 13 finns i listan eller ej:

```
int searchNum = 13;
  cout << endl << "Talet " << searchNum << " finns ";
  if(!list.searchFor(searchNum))
    cout << "INTE ";
  cout << "i listan!!!";
```

Kommentar:

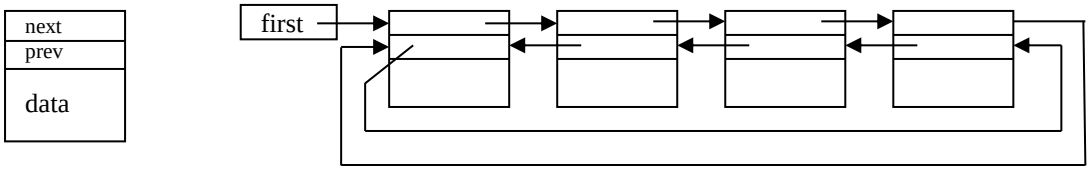
- list.searchFor(searchNum) returnerar true eller false beroende på om searchNum finns eller ej.

Hela programmet på nästa sida:

```
// list_030 Version 10
// Använd en enkellänkad lista definierad i klassen List
// Per Ekeroot 2014-01-13
//--------------------------------------------------------------------------
#include "list.h"
#include <iostream>
using namespace std;

int main()
{
//--------------------------------------------------------------------------
// Skapa en tom lista
//--------------------------------------------------------------------------
  List list;

//--------------------------------------------------------------------------
// Lägg i värden i listan. Skriv sedan ut listan
//--------------------------------------------------------------------------
  for(int i=1; i<=4; i++)
    list.putFirst(12*i);

  list.showList();
//--------------------------------------------------------------------------
// Sätt in värdet '13' efter värdet '24'
//--------------------------------------------------------------------------
  list.insAfter(24,13);
  list.showList();

//--------------------------------------------------------------------------
// Sätt in värdet '134' sist i listan
//--------------------------------------------------------------------------
  list.putLast(134);
  list.showList();

//--------------------------------------------------------------------------
// Sök i listan efter talet searchNum
//--------------------------------------------------------------------------
  int searchNum = 13;
  cout << endl << "The number " << searchNum << " is ";
  if(!list.searchFor(searchNum))
    cout << "NOT ";
  cout << "in the list!!!" << endl << endl;

  return 0;
}
```

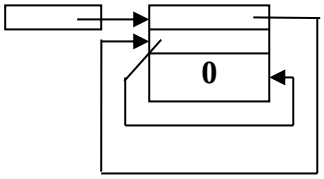
7.3 Dubbellänkad lista

I en dubbellänkad lista har man pekare både till nästa och till föregående nod. Här visas en dubbellänkad cirkulär lista. Att listan är cirkulär ges av att den sista noden pekar på den första noden och inte på null samt att den först noden pekar på den sista.



Listans huvud:

- Innehåller **inte** data
- Pekar på sig själv



7.3.1 Nod

Klassen för noden ser nu ut så här:

```
class DNode
{
  friend class Dlist;
  DNode *next, *prev;
  int data;
  DNode(DNode *n=0, DNode *p=0, int d=0)
    : next(n), prev(p), data(d) {}
};
```

Kommentarer:

- next pekar på nästa nod
- prev pekar på föregående nod (previous)

7.3.2 Klassdefinition

Klassen finns i dlist.h
Klassdefinition för en dubbellänkad lista:

```
class DNode;                        // Forward - deklaration
class Dlist
{
  private:
    DNode *first;
    void putP(DNode *n, int d);           // Hjälpfunktion
    DNode *search(int searchItem) const;   // Hjälpfunktion

public:
    Dlist();                        // Skapa ett listhuvud med ett tomt element
    ~Dlist();                       // Återlämna allokerat utrymme
    void putFirst(int d);
    void putLast (int d);
    bool putPrev (int searchItem, int d);
    bool putAfter(int searchItem, int d);
    void showList() const;
    bool delNode(int searchItem);
};
```

Kommentarer:

- Klassen DList innehåller datamedlemmen DNode *first, dvs en pekare till listan

- Klassen DList innehåller medlemsfunktioner för att sätta in data först, sist, före och efter önskat ställe, ta bort data samt utskriftsfunktioner
- Klassen har en hjälpfunktion (putP) för att sätta in en nod **före** utpekad nod. Denna hjälpfunktion används i de övriga medlemsfunktionerna som sätter in noder.
- Klassens sökfunktion är private-deklarerad eftersom den returnerar en pekare till en nod. Vill man söka efter data i listan får man skriva en särskild sökfunktion för detta.
- Klassens konstruktor skapar ett listhuvud
- Klassens destruktor frigör allokerad plats
- Klassen DList använder klassen DNode för att hantera noderna
- DNode definieras i list.cpp!!

7.3.3 Implementation av medlemsfunktionerna

Här följer implementation av medlemsfunktionerna med kommentarer i anslutning till vare funktion.

Definiera en klass som representerar en nod i en dubbellänkad lista:

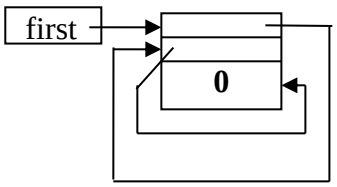
```
class DNode
{
  friend class Dlist;
  DNode *next, *prev;
  int data;
  DNode( DNode *n=0, DNode *p=0, int d=0)
    : next(n), prev(p), data(d) {}
};
```

Förvald konstruktor. Skapa ett listhuvud:

```
Dlist::Dlist()
{
  first =new DNode(0,0,0);
  first->next = first->prev = first;
}
```

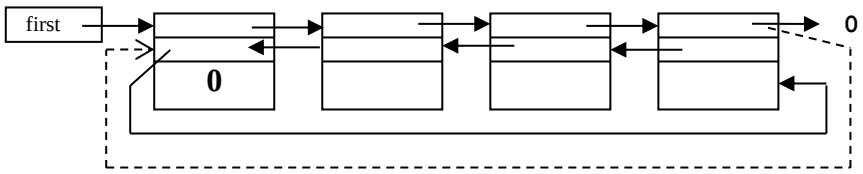
Kommentarer:

- Det allokeras plats för en nod (DNode). first sätts att peka på denna plats
- Båda pekarna i noden pekar på det minnesutrymmet som first pekar på



Destruktor som tar bort allokerad plats:

```
Dlist::~Dlist()
{
  first->prev->next = 0;
  while(first)
  {
    DNode *p = first;
    first = first->next;
    delete p;
  }
}
```

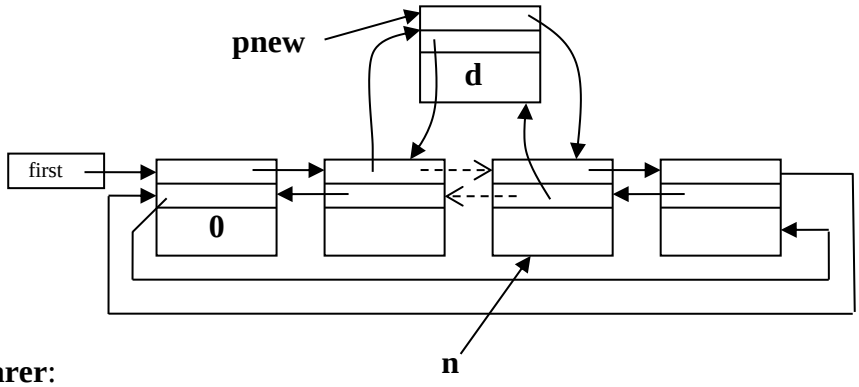


Kommentarer:

- first->prev->next ser till att sista noden pekar på 0 (NULL), sedan fungerar frigörandet av minnet som för en enkellänkad lista

Lägg in en node före noden med adressen n. putP är en hjälpfunktion som används internt i klassen.

```
void Dlist::putP(DNode *n, int d)
{
  DNode *pnew = new DNode(n, n->prev,d);
  n->prev->next = pnew;
  n->prev = pnew;
}
```



Kommentarer:

- putP används för att sätta in en nod före noden med adressen n
- putP är en hjälpfunktion som används av andra medlemsfunktioner t.ex putFirst, putLast, putPrev, putAfter. I dessa används search() (se nedan) för att söka upp rätt nod.

Leta efter noden som har data = searchItem

```
DNode *Dlist::search(int searchItem) const
{
  DNode *p;
  for(p=first->next; p!=first&& p->data!=searchItem; p=p->next)  ;
  return p;
}
```

Kommentarer:

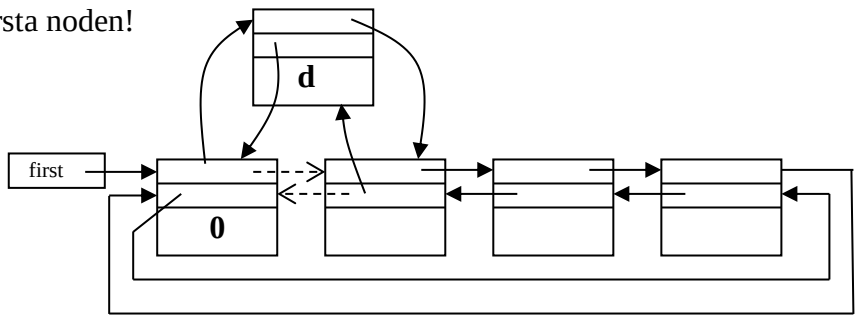
- Fungerar på samma sätt som search() för enkellänkad lista.
- p!= *first* är fortsättningsvillkor eftersom p kommer till *first* efter sista noden. Detta betyder att när p har stegat genom listan och kommit till *first*(= pekar på samma nod som *first*) vet man att p har stegat igenom **hela** listan.

Lägg in data = d, först i listan

```
void Dlist::putFirst(int d)
{
  putP(first->next,d);
}
```

Kommentarer:

- first->next pekar på första noden!

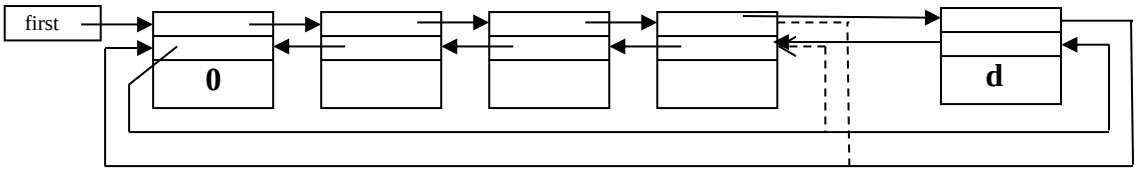


Lägg in data = d sist i listan:

```
void Dlist::putLast (int d)
{
  putP(first,d);
}
```

Kommentarer:

- Sist = före huvudet



Lägg in data = d före noden med data = searchItem

```
bool Dlist::putPrev (int searchItem, int d)
{
  DNode *n = search(searchItem);
  if(n != first)
    putP(n,d);
  return n!= first;
}
```

Kommentarer:

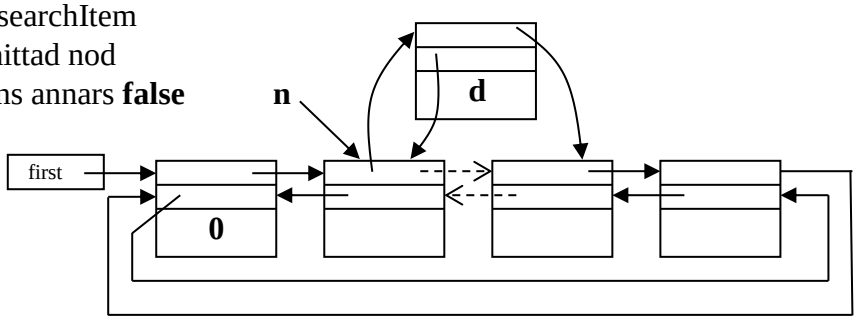
- Endast om searchItem hittas läggs data in
- search() returnerar first om data ej hittas
- Om searchItem är med i listan returnerar putPrev() **true** annars **false**
- Se figuren för putP()!

Lägg in data = d efter noden med data = searchItem:

```
bool Dlist::putAfter(int searchItem, int d)
{
  DNode *n = search(searchItem);
  if(n != first)
    putP(n->next,d);
  return n!=first;
}
```

Kommentarer:

- Sök efter noden med data = searchItem
- Sätt in den nya noden efter hittad nod
- returnera **true** om noden finns annars **false**

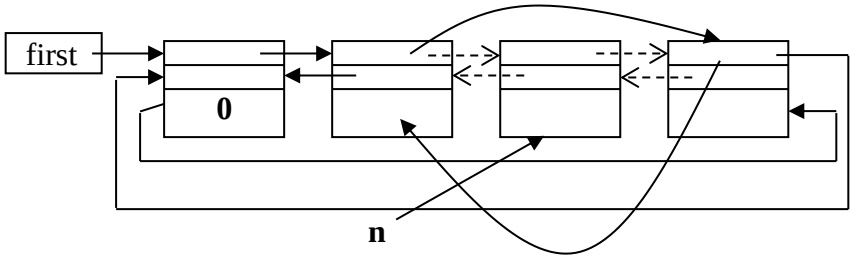


Skriv ut listan:

```
void Dlist::showList() const
{
  DNode *p;
  for(p=first->next; p!=first; p=p->next)
    cout << setw(4) << p->data ;
  cout << endl;
}
```

Tag bort noden med data = searchItem:

```
bool Dlist::delNode(int searchItem)
{
   DNode *n = search(searchItem);
   if(n!=first)
   {
     n->prev->next = n->next;
     n->next->prev = n->prev;
     delete n;
   }
   return n!= first;
}
```



Kommentarer:

- Flytta om pekarna och frigör sedan minnet som allokerats för den noden som ska tas bort.
- n!=first ger true om search() hittar en nod som inte är listhuvudet
- Om searchItem är med i listan returnerar delNode() true annars false

7.3.4 En applikation med en dubbellänkad lista

Skapa en tom lista med

```
  Dlist dlist;
```

Lägg till data sist i listan

```
  dlist.putLast(127);
```

Lägg till data först i listan

```
  dlist.putFirst(11);
```

Skriv ut listan

```
  dlist.showList();
```

Radera ett värde ur listan. Användaren anger värde.

```
int tal;
  cout <<endl<< " Ange värde som du vill ta bort: ";
  cin >> tal;
  if(dlist.delNode(tal))
    cout <<endl<< tal << " raderades ur listan!" <<endl<<endl;
  else
    cout <<endl<< tal << " fins INTE i listan!" <<endl<<endl;
```

Kommentarer:

- dlist.delNode(tal) returnerar **true** om noden med talet *tal* finns i listan så att talet kan tas bort, annars returneras **false**

Och hela programmet:

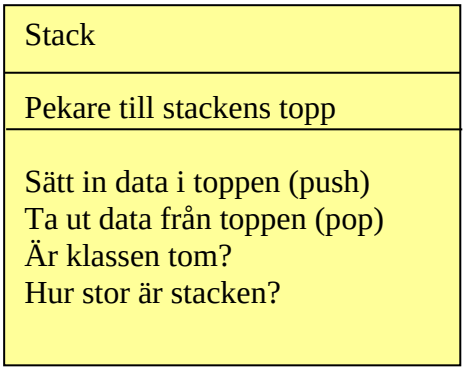
```
// list_040 Version 10
// Använd en dubbellänkad lista definierad i klassen Dlist
// Per Ekeroot 2014-01-13
//-----------------------------------------------------------------------------
#include <iostream>
#include "dlist.h"
using namespace std;

int main()
{
//-----------------------------------------------------------------------------
// Skapa en tom lista
//-----------------------------------------------------------------------------
  Dlist dlist;

//-----------------------------------------------------------------------------
// Lägg in värden sist i listan. Skriv sedan ut listan
//-----------------------------------------------------------------------------
  for(int i=1; i<=4; i++)
    dlist.putLast(12*i);

  dlist.showList();

//-----------------------------------------------------------------------------
// Lägg in ett värde först i listan
//-----------------------------------------------------------------------------
  dlist.putFirst(11);
  dlist.showList();

//-----------------------------------------------------------------------------
// Lägg in ett värde före angivet värde i listan
//-----------------------------------------------------------------------------
  if(dlist.putPrev(254,34))
    dlist.showList();
  else
    cout <<endl<< "No input!" << endl<<endl;

//-----------------------------------------------------------------------------
// Lägg in ett värde efter angivet värde i listan
//-----------------------------------------------------------------------------
  if(dlist.putAfter(24,34))
    dlist.showList();
  else
    cout <<endl<< "No input!" << endl<<endl;

//-----------------------------------------------------------------------------
// Ta bort ett angivet värde ur listan
//-----------------------------------------------------------------------------
  int tal;
  cout <<endl<< " Input value to delete: ";
  cin >> tal;
  if(dlist.delNode(tal))
    cout <<endl<< tal << " deleted from list!" <<endl<<endl;
  else
    cout <<endl<< tal << " is NOT in the list!" <<endl<<endl;

  dlist.showList();
          cout << endl << endl;
  return 0;
}
```

7.4 Stack

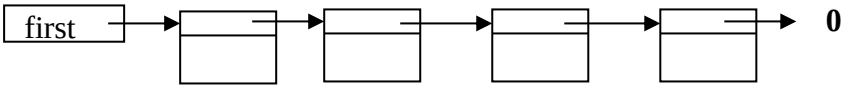
Vi har tidigare i kursen implementerat en stack och då använt en kontinuerlig lista, en array. Nackdelen med denna implementering var att stacken hade en begränsad storlek, eftersom vi måste ange en storlek på arrayen.

Vi ska nu implementera en stack med enkellänkad lista. Fördelen med denna implementering är att vi inte behöver ange någon maximal storlek för stacken.

I en stack lägger man till data först i listan och tar bort data först i listan, LIFO (Last In First Out). Lägga till data brukar heta *push* och ta bort data brukar heta *pop*.



Vi använder den enkellänkad lista på samma sätt som i klassen List. Den länkade listan ser då ut så här:



first pekar på stackens topp.

7.4.1 Klassdefinition

Vi skapar en klass för stacken:

```
typedef int Item;             // Datatyp för elementen i stacken

class Node;

class Stack
{
  private:
    Node *first;
    int size;
  public:
    Stack():first(0), size(0){};
    ~Stack();
    void push(Item pData);
    Item pop();
    bool isEmpty() const;
    int getSize() const;
};
```

Kommentarer:

- Klassens två datamedlemmar är en pekare till en enkellänka lista och antal element på stacken, stackens storlek.
- Observera att klassens konstruktor är en inlinefunktion som skapar en tom stack med en pekare till NULL och nollställer klassens storlek.
- Stacklassen har följande medlemsfunktioner:
  - Lägg till data (push)
  - Ta bort data (pop)
  - Kontrollera om stacken är tom
  - Returnera stackens storlek

Klassen för noden är likadan som nodklassen för den enkellänkade listan ovan:

```
class Node
{
   friend class Stack;
   Node *next;
   Item data;

   Node (Node *n, Item pData) : next(n), data(pData) {}
};
```

Kommentarer:

- Denna kod är placerat i definitionsfilen stack.cpp för att den inte ska vara publikt åtkomlig

7.4.2 Implementation av medlemsfunktionerna i Stack

```
// stackl.cpp Version 10
// Definitionsfil för klassen Stack
// En länkad lista för att hantera en stack
// Per Ekeroot 2014-01-13
//--------------------------------------------------------------------------
#include "stackl.h"
```

Destruktor som frigör allokerat minnet:

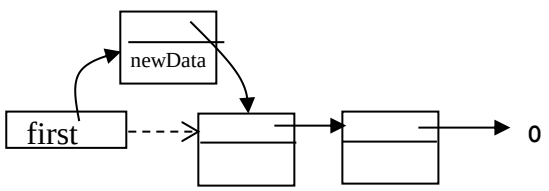
```
Stack::~Stack()
{
  while(!isEmpty())
  {
     Node *tmp= first;
     first = first->next;
     delete tmp;
  }
}
```

Kommentarer:

- Frigör allokerad plats. Se kommentarer för destruktorn i den enkellänkade listan

Lägg till ett värde på stacken:

```
void Stack::push(Item pData)
{
  first = new Node(first, pData);
  size++;
}
```

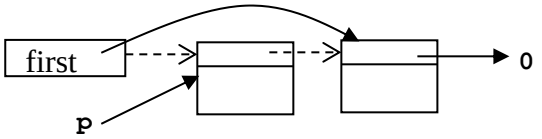


Kommentarer:

- Lägg till data först i listan

Ta bort ett värde från stacken:

```
Item Stack::pop()
{
  Node *p= first;
  Item data = p->data;
  first = first->next;
  delete p;
  size--;
  return data;
}
```



Kommentarer:

- Skapa en temporär pekare p som pekar på första noden
- Tilldela data från första noden till den temporära variabeln data (= returvärde)
- Låt first peka på den andra noden
- Ta bort den först noden (p)
- Returnera data (= ett värde poppas från stacken)

Kontrollera om stacken är tom:

```
bool Stack::isEmpty() const
{
  return first== 0;     //NULL
}
```

Kommentarer:

- Returnera **true** om stacken är tom annars **false**

Returnera stackens storlek:

```
int Stack::getSize() const
{
           return size;
}
```

7.4.3 Ett klientprogram som använder Stack

```
//-----------------------------------------------------------------------------
// list_050 Version 10
// Test av klassen Stack för att hantera en dynamisk stack
// Per Ekeroot 2014-01-13
//-----------------------------------------------------------------------------
#include <iostream>
#include <string>
#include "stackl.h"
#include "..\PEFunkBibl.h"
using namespace std;
//-----------------------------------------------------------------------------
// Funktionsprototyper
//-----------------------------------------------------------------------------
void pushData(Stack &stack);
void popData(Stack &stack);
void printSize(Stack const &stack);

//-----------------------------------------------------------------------------
int main()
{
  // Initiera menyn
  const int ITEMNUM = 4;
  string menuItems[ITEMNUM]= {"Push an integer on the stack",
                              "Pop an integer from the stack",
                              "Number of elements on the stack",
                              "Quit"};
  // Skapa en tom stack
  Stack stack;

  bool go = true;
  do
  {
    system("cls");
    switch(menu( menuItems, ITEMNUM))
    {
       case '1': pushData(stack);    // Lägg data på stacken
                 break;
       case '2': popData(stack);     // Ta data från stacken
                 break;
       case '3': printSize(stack);   // Skriv stackens storlek på skärmen
                 break;
       case '4': go = false;         // Avsluta  programmet
    };
  } while(go);
  return 0;
}
```

Kommentarer:

- Jämför med programmet class_80 i vilket vi hanterade en stack med en kontinuerlig lista array). Programmen är lika, det är bara implementeringen av stacken som skiljer programmen åt!
- Se class_080.cpp för definitionerna av pushData(), popData() och printSize()

7.5 Kö

Vi ska nu titta närmare på en annan ADT, nämligen en kö. I en kö lägger man till data sist i listan och tar bort data först ur listan, FIFO (First In First OUT) . Vi ska implementera kön med hjälp av en enkellänkad lista.

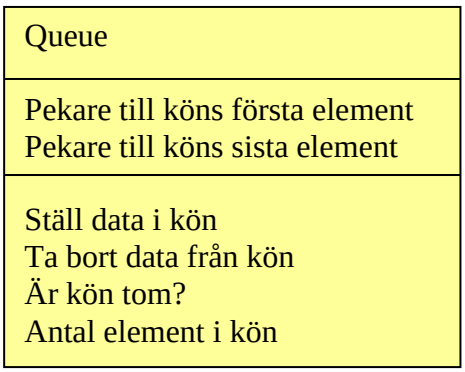
7.5.1 Klassdefinition för en kö

I en kö har man en pekare till första noden och en pekare till sista noden. Pekare till den sista noden används för att lägga till data sist i kön.

Köklassen ska ha medlemsfunktioner för att:

- lägga till data
- ta bort data
- kolla om kön är tom
- ta reda på antal element i kön

Ett klassdiagram för klassen kan se ut så här:



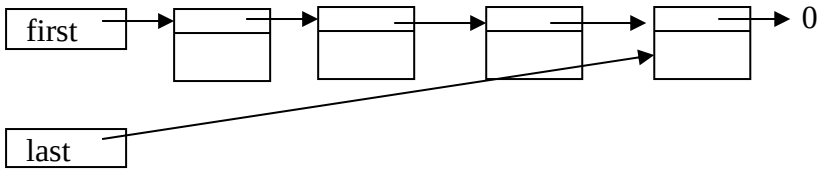
En klassdefinition för köklassen:

```
//queue.h
typedef int Item;            // Datatyp för elementen i kön
class Node;
class Queue
{
  private:
    Node *first, *last;
    int size;

  public:
    Queue() : first(0), last(0), size(0) {}
    ~Queue();
    void enque(Item pData);
    bool deque(Item &pData);
    bool isEmpty()const;
    int getSize() const;
};
```

Kommentarer:

- Klassens datamedlemmar är pekare till köns första respektive sista nod, se figur nedan samt ett heltal som håller reda på antal element i kön.
- Klassen har en konstruktor och en destruktor
  - Konstruktorn initierar pekarna med värdet NULL och antal element till 0.
  - Destruktorn frigör den plats som allokerats av medlemsfunktionen enque().
- Klassen har medlemsfunktionerna
  - enque(): lägger till data sist i kön samtidigt som plats allokeras. Antal element räknas upp med 1.
  - deque(): läser data från köns först nod, tar bort noden och frigör den plats i minnet som allokerats för denna nod samt räknar ner antalet element med 1.
  - isEmpty(): rapporterar **true** om kön är tom annars **false**.
  - getSize() returnerar antal element i kön

Se figur på nästa sida!



7.5.2 Implementation av köns medlemsfunktioner

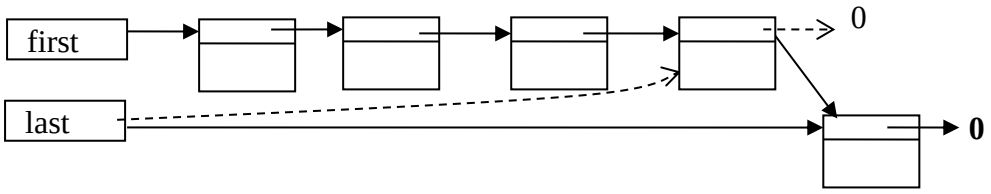
Implementationerna av köns medlemsfunktioner ligger i filen queue.cpp

Destruktor Frigör allokerat minne:

```
Queue::~Queue()
{
   while(!isEmpty())
   {
      Node *tmp = first;
      first = first->next;
      delete tmp;
   }
}
```

Lägg till data sist i kön:

```
void Queue::enque(Item pData)
{
  Node *pNew = new Node(0,pData);
  if(isEmpty())
    first = pNew;
  else
    last->next = pNew;
  last = pNew;

  size++;
}
```

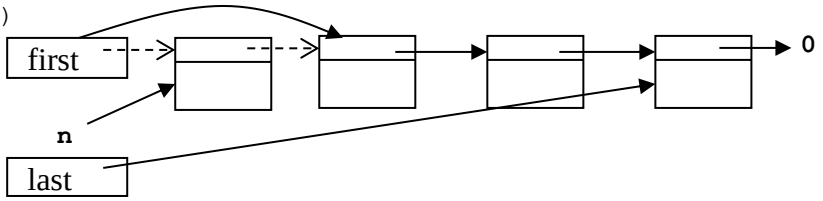


Kommentarer:

- Skapa en ny nod som innehåller pData och pekar på NULL
- Om kön är tom låter man first peka på denna nya nod
- Om kön inte är tom låter man sista noden peka på denna nya nod
- I båda fallen låter man last peka på den nya noden

Tag data från köns första nod:

```
bool Queue::deque(Item &pData)
{
  if(isEmpty())
     return false;

  Node *n = first;
  pData = n->data;
  first = first->next;
  if(isEmpty())  //Om kön BLEV tom
    last =0 ;    //=NULL

  delete n;
  size--;
  return true;
}
```



Kommentarer:

- deque fungerar på samma sätt som pop i en stack
- Kolla först så att kön inte är tom
- Skapa en temporär pekare, som pekar på första noden
- Kopiera data till (referens-) variabeln pData, dvs returnera data från noden
- Låt first peka på andra noden
- Om kön blev tom ska last peka på NULL
- Frigör minnet som n pekar på
- Returnera **true** om en nod togs bort annars **false**

Kontrollera om kön är tom:

```
bool Queue::isEmpty() const
{
   return first== 0;
}
```

7.5.3 En applikation som använder köklassen

Skapa en tom kö:

```
  Queue que;
```

Lägg data sist i kön

```
que.enque(23);
```

Tag data från köns början:

```
  int q;
  if(que.deque(q))
    cout <<endl<< "Data from the queue            :"<< q << "     ";
  else
    cout <<endl<< "The queue is empty!" << endl;
```

Kommentarer:

- Data som tas från kön läggs i variabeln q
- Om det finns data att ta från kön returnerar deque() **true** annars **false**.

Hela programmet:

```
// list_060 Version 10
// Test av klassen Queue för att hantera en dynamisk kö
// Per Ekeroot 2014-01-13
//-----------------------------------------------------------------------
#include <iostream>
#include "queue.h"
using namespace std;
int main()
{
//-----------------------------------------------------------------------
// Skapa en tom kö
//-----------------------------------------------------------------------
  Queue que;

//-----------------------------------------------------------------------
// Lägg till heltal i kön (sist).
//-----------------------------------------------------------------------
  for(int i=1; i < 10; i++)
    que.enque(i);

  cout <<endl<< "Number of integers in the queue: "<<que.getSize()<<endl<< endl;

//-----------------------------------------------------------------------
// Tag bort värden från kön (fån början).
//-----------------------------------------------------------------------
  int q;
  for(int i=0; i < 5; i++)
    if(que.deque(q))
      cout << "Data from the queue            : "<< q << endl;
    else
      cout << "The queue is empty!" << endl;
  cout << endl;

//-----------------------------------------------------------------------
// Lägg till ett värde i kön
//-----------------------------------------------------------------------
  cout << endl << "Add the number 35 to the queue!";
  que.enque(35);
  cout <<endl<<"Number of integers in the queue: "<< que.getSize()<<endl<< endl;

//-----------------------------------------------------------------------
// Töm kön
//-----------------------------------------------------------------------
  for(int i=0; i < 10; i++)
    if(que.deque(q))
      cout << "Delete data from the queue     : "<< q << endl;
    else
      cout << "The queue is empty!" << endl;

  cout << endl;
  cout << "Number of integers in the queue: " << que.getSize() << endl << endl;
  return 0;
}
```

Kommentarer:

- Se klassen Queue som en ADT i vilken vi gömmer alla besvärliga detaljer. Det enda vi behöver hålla reda på är att enque() lägger till data sist i kön och deque() tar bort från köns början.