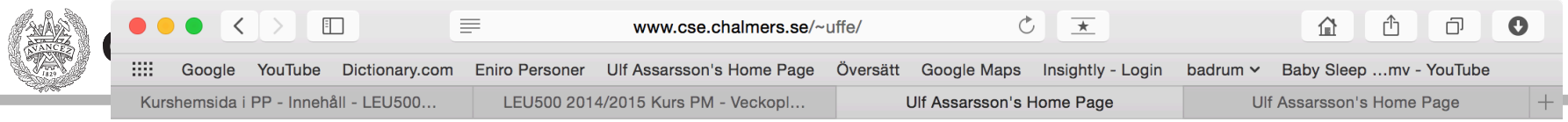
Fortsättning Pekare

Ulf Assarsson

Originalslides av Viktor Kämpe



CHALMERS GÖTEBORG UNIVERSITY

Computer Science and Engineering

- Home
- Research Group
- Publications
- Teaching
- Soft Shadows
- BART
- Bachelor's Theses
- Master's Theses

Teaching

My courses at the Department of Computer Engineering, Chalmers University of Technology:

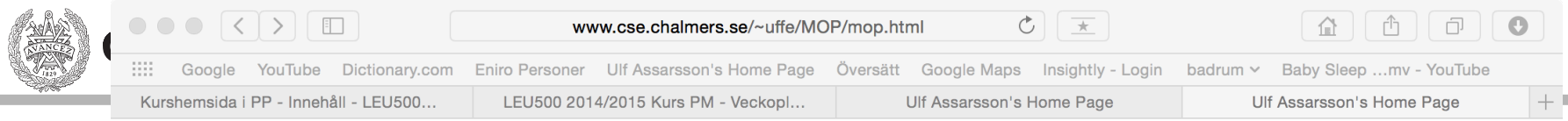
- Seminar Course in Advanced Computer Graphics
- TDA 361 - Computer Graphics
- Maskinorienterad Programmering / Programmering av Inbyggda System
- EDA 425 -Advanced Computer Graphics
- TDA 360 - Computer Graphics

Main research interests: Shadow algorithms - both real-time and non real-time - for hard and soft shadows including shadows in hair, fur, smoke and volumetric participating media (shafts of light), Parallelism, GPGPU-algorithms, GPU-Ray Tracing, Real-time Global Illumination, Real-time rendering algorithms.

My PhD-students:

- Markus Billeter
- Ola Olsson
- Erik Sintorn
- Viktor Kämpe

Previous commitments:
1997-1999: Prosolvia
1998-2001: industrial PhD-student at Computer Engineering, Chalmers



C-kurs

Delkurs i Maskinorienterad Programmering samt Programmering av Inbyggda System. Slides:

- Intro
- Pekare och Fält
- Pekare fortsättning
- Kodningskonventioner

Länk till space-invaderspel i CodeLite och C.

Förra föreläsningen

- Pekare till data
- Arrayer – fix storlek och adress
- Dynamisk minnesallokering
  - `malloc()`
  - `free()`

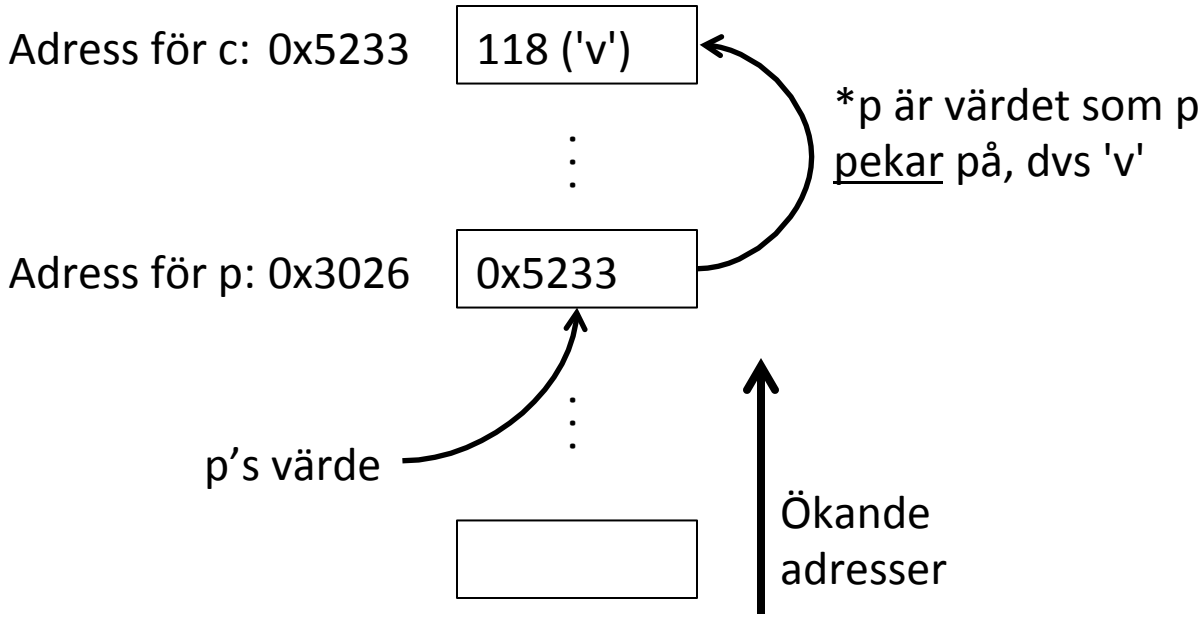
Pekare - sammanfattning

Dvs, om a's värde är en adress, så är *a vad adressen innehåller.

```
&a -> Adress till variabel a. Dvs minnesadress som a är lagrat i.
a  -> variabelns värde (t ex int, float eller en adress om a är pekarvaribel)
*a -> Vad variabel a pekar på (här måste a's värde vara en giltig
      adress och a måste vara av typen pekare). a's värde är en adress till en
      annan variabel eller port. Vi hämtar värdet för den variabeln/porten.
```

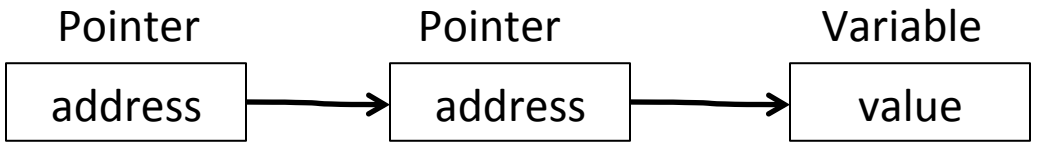
Exempel för pekarvariabel p:

```
char  c = 'v';
…
char* p = &c;

*p == c;
```



Pekare till pekare

```
char *p1, *p2, *p3;
char **pp;

pp = &p1;
```




```
// Annat exempel. Funktion som allokerar minne dynamiskt, t ex för elaka fiender i ett spel
void allocateEnemyArray(struct Enemy **pp, int n)
{
  *pp = (struct Enemy *)malloc(n * sizeof(struct Enemy));
}

int main()
{
  struct Enemy *pEnemies = NULL;
  allocateEnemyArray(&pEnemies, 100);
  // Game logic
  free(pEnemies);
}
```

*pp == pEnemies

Ska funktionen kunna uppdatera argumentet måste vi skicka in adressen för argumentet (istället för värdet på argumentet)

Kul med pekare – vad skrivs ut?

```
#include <stdio.h>
#include <conio.h>
char * s1 = "Emilia"; // variabeln s1 är en variabel som går att ändra, och vid
                      // start tilldelas värdet av adressen till 'E'.
char s2[] = "Roger";  // värdet på s2 känt vid compile time. s2 är konstant, dvs
                      // ingen variabel som går att ändra. Är adressen till 'R'.
int main()
{
          printf("Kul med pekare\n");

          char **pp, *p = s1;
          pp = &p;
          printf("p = %s\n", p);  // p == s1, så ekvivalent med att skicka s1 som argument.
          printf("*pp = %s\n", *pp);

          *pp = s2;         // *pp ekvivalent med p. Ändrar p till s2
          printf("p = %s\n", p);

          *p = 'T';         // (p == s2). *s2 ekv. med s2[0]. Ändrar s2[0] till 'T'
          **pp = 'J';       // *pp ekv med p. p==s2. ->  *s2 = 'J'. Ändrar s2[0] till 'J'
          *(*pp+2) = 'k';   // ändrar s2[2] till 'k'
          printf("%s\n", p);

          (*pp)[0] = 'P';   // *pp ekv med p (== s2). s2[0]. Ändrar s2[0] till 'P'
          printf("%s\n", p);

          getch();
          return 0;
}
```

Array av pekare

```c
#include <stdio.h>

char* fleraNamn[] = {"Emil", "Emilia", "Droopy"};

int main()
{
    printf("%s, %s, %s\n", fleraNamn[2], fleraNamn[1], fleraNamn[0]);

    return 0;
}
```

```
Droopy, Emilia, Emil

sizeof(fleraNamn) = 12; // 3*sizeof(char*) = 3*4 = 12
```

Array av arrayer

```c
#include <stdio.h>

char kortaNamn[][4] = {"Tor", "Ulf", "Per"};

int main()
{
    printf("%s, %s, %s\n", kortaNamn[2], kortaNamn[1], kortaNamn[0]);

    return 0;
}
```

```
Per, Ulf, Tor
```

Array av arrayer

```
#include <stdio.h>

int arrayOfArrays[3][4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };

int main()
{
    int i,j;
    for( i=0; i<3; i++) {
        printf("arrayOfArray[%i] = ", i);
        for ( j=0; j<4; j++)
            printf("%i ", arrayOfArrays[i][j]);
        printf("\n");
    }

    return 0;
}
```

Pekare till portar och funktioner

Denna föreläsning handlar om pekare till

- Portar
- Funktioner
- Sammansatta datatyper (`struct`)

Absolutadressering

- Vid portadressering så kan vi ha en absolut adress (t ex 0x400).

Absolutadressering

```
0x400                          // ett hexadecimalt tal
(unsigned char*)0x400          // en unsigned char pekare som pekar på adress 0x400
*((unsigned char*)0x400)       // dereferens av pekaren

// läser från 0x400
unsigned char value = *((unsigned char*)0x400);

// skriver till 0x600
*((unsigned char*)0x600) = value;
```

Läsbarhet med typedef

```
0x400
(unsigned char*)0x400
*((unsigned char*)0x400)

// läser från 0x400
value = *((unsigned char*)0x400);
```

```
typedef unsigned char* port8ptr;
#define INPORT_ADDR 0x400
#define INPORT *((port8ptr)INPORT_ADDR)

INPORT_ADDR
(port8ptr)INPORT_ADDR
INPORT

// läser från 0x400
value = INPORT;
```

`typedef` förenklar/förkortar uttryck, vilket kan öka läsbarheten.

```
typedef unsigned char* port8ptr;
```

typ alias/typnamn

Volatile qualifier

```
char * inport = (char*)0x400;

void foo(){

    while(*inport != 0)
    {
        // ...
    }
}
```

En kompilator som optimerar kanske bara läser en gång (eller inte alls om vi aldrig skriver till adressen från programmet).

Volatile qualifier

```
volatile char * inport = (char*)0x400;

void foo(){

    while(*inport != 0)
    {
        // ...
    }
}
```

`volatile` hindrar vissa optimeringar (vilket är bra och här nödvändigt!), ty anger att kompilatorn måste anta att innehållet på adressen kan ändras utifrån.

Sammanfattning portar

Inport:

```
typedef volatile unsigned char* port8ptr;
#define INPORT_ADDR 0x400
#define INPORT *((port8ptr)INPORT_ADDR)

// läser från 0x400
value = INPORT;
```

Utport:

```
typedef volatile unsigned char* port8ptr;
#define UTPORT_ADDR 0x401
#define UTPORT *((port8ptr)UTPORT_ADDR)

// skriver till 0x401
UTPORT = value;
```

[portadressering i XCC12]

Funktionspekare

```c
#include <stdio.h>

int square(int x)
{
    return x*x;
}

int main()
{
    int (*fp)(int);

    fp = square;

    printf("fp(5)=%i \n", fp(5));
    return 0;
}
```

En funktionspekare

```
fp(5)=25
```

Funktionspekare

```
int (*fp)(int);
```

Funktionspekarens typ bestäms av:

- Returtyp.
- Antal argument och deras typer.

Funktionspekarens värde är en adress.

Likheter assembler – C

```
utport      EQU         $400

            ORG         $1000
start:
            LDAA        #1

loop_start:
            STAA        utport
            JSR         delay

            LSLA
            BEQ         start
            BRA         loop_start


delay:
            LDAB        #$FF
loop_delay:
            DECB
            BNE         loop_delay
            RTS

var1        RMB         2
```

```
int var1;


void delay()
{
    unsigned char tmp = 0xFF;
    do {
        tmp--;
    } while(tmp);
}
```

Både funktioner och globala variabler har adresser i minnet, men vi använder symboler.

Funktionspekare – exempel

```
int(*print)(const char*,…);

print = printf;
Eller
print = popupWindow;

Objektorientering. Metoder:

struct GfxObject {
 float x,y;
…
 void (*render)(struct GfxObject* this); // renderingsfunktion för att rita
                                         // ut det här objektet på skärmen.
}
struct GfxObject gfxObj[10];
for(int i=0; i<10; i++)
   gfxObj[i].render(&gfxObj[i]);
```

[exempel på funktionspekare]

Sammansatta datatyper

En så kallad `struct` (från eng. *structure*)

- Har en/flera medlemmar.
- Medlemmarna kan vara av
  - bas-typ (t ex `int`, `float`, `char`, `double`)
  - egendefinerad typ (t ex en annan `struct`).
  - Pekare (även till funktioner och samma `struct`)

Användning av `struct`

- Skriv ner en struct som innehåller några medlemmar, inkl en funktionspekare och en annan struct! Checka med grannen, och ta hjälp av grannen om problem.

```
#include <stdio.h>

char* kursnamn = ”Maskinorienterad Programmering";

struct Course {
    char* name;
    float credits;
    int numberOfParticipants;
};

int main()
{
    struct Course mop;

    mop.name = kursnamn;
    mop.credits = 7.5f;
    mop.numberOfParticipants = 110;

    return 0;
}
```

Definition av strukturen

Deklaration av variabeln mop

Access till medlemmar via .-operatorn

Initieringslista

```
struct Course {
    char* name;
    float credits;
    int numberOfParticipants;
};

struct Course kurs1 = {"MOP", 7.5f, 110};
struct Course kurs2 = {"MOP", 7.5f};
```

En `struct` **kan** initieras med en initieringslista. Initieringen sker i samma ordning som deklarationerna, men alla medlemmar måste inte initieras.

Pekare till `struct` (pilnotation "->")

```c
#include <stdio.h>

char* kursnamn = "Maskinorienterad Programmering";

struct Course {
    char* name;
    float credits;
    int numberOfParticipants;
};

int main()
{
    struct Course *pmop;
    pmop = (struct Course*)malloc(sizeof(struct Course));

    (*pmop).name = kursnamn;
    pmop->name   = kursnamn;
    pmop->credits = 7.5f;
    pmop->numberOfParticipants = 110;

    free(pmop);
    return 0;
}
```

Access till medlemmar via -> operatorn

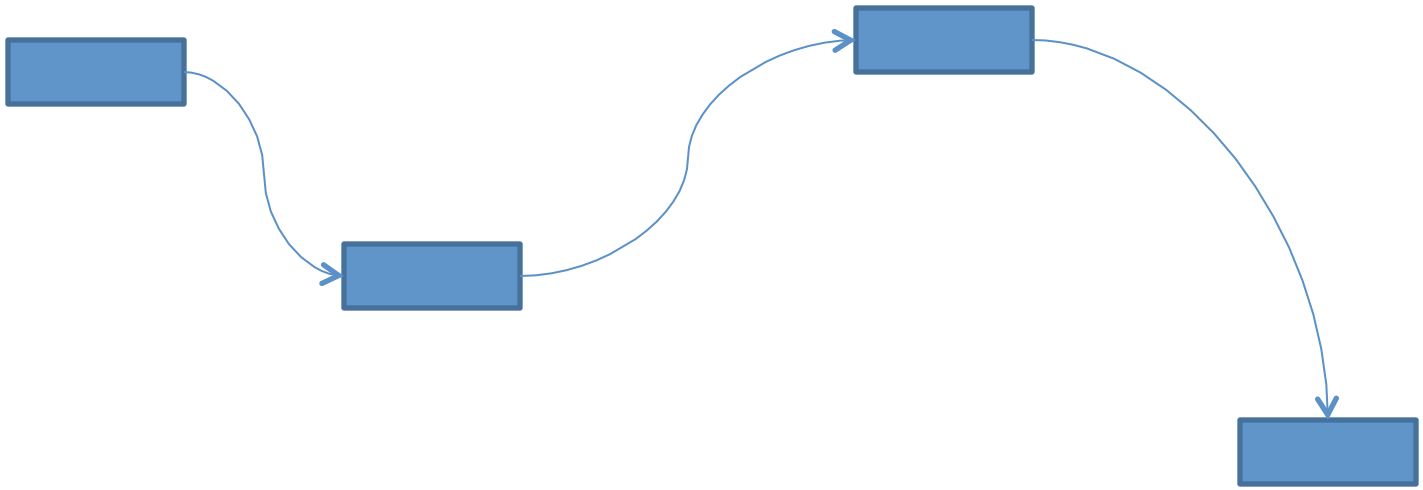
I Java:

```
public class Course {
  String name;
  float credits;
  int numberOfParticipants;
}

Course mop = new Course();
mop.name = …
mop.credits = 7.5;
…
```

[struct-exempel]

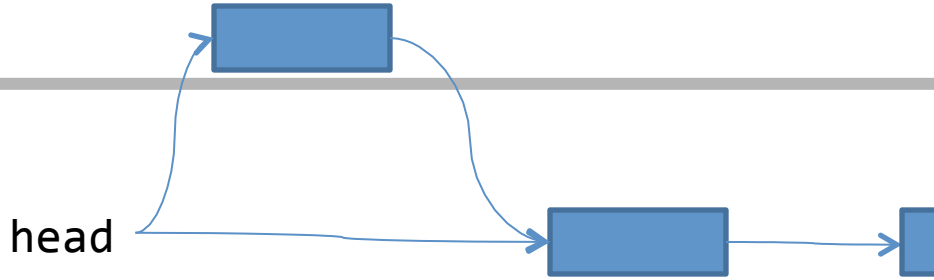
Länkade datastrukturer

- Ex. Länkad lista:



Elementen ligger inte på konsekutiva adresser, utan listans minnesstruktur bestäms av pekare.

Enkellänkad lista

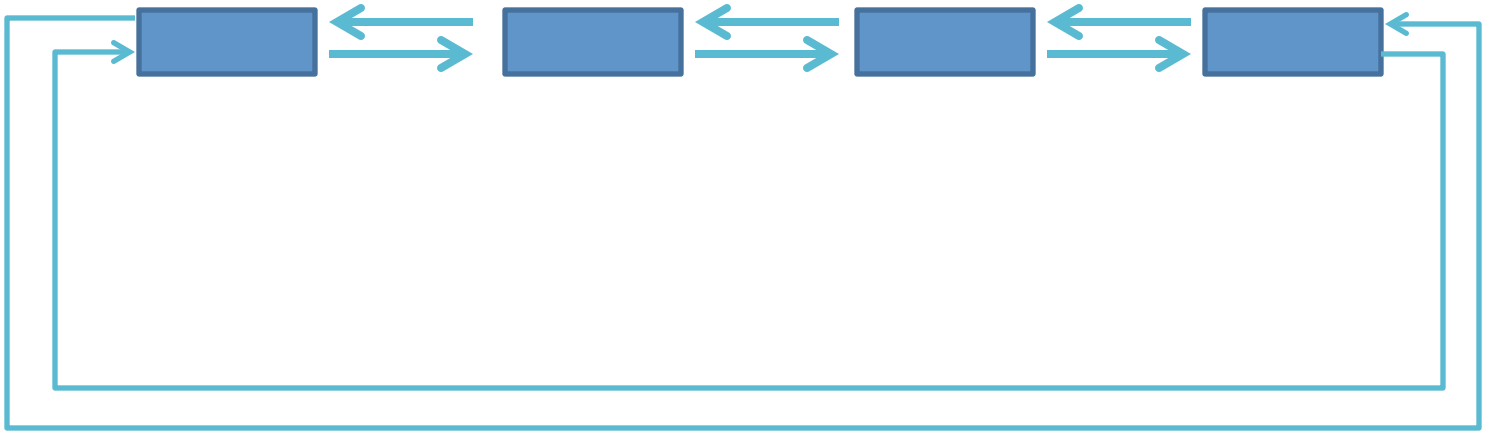


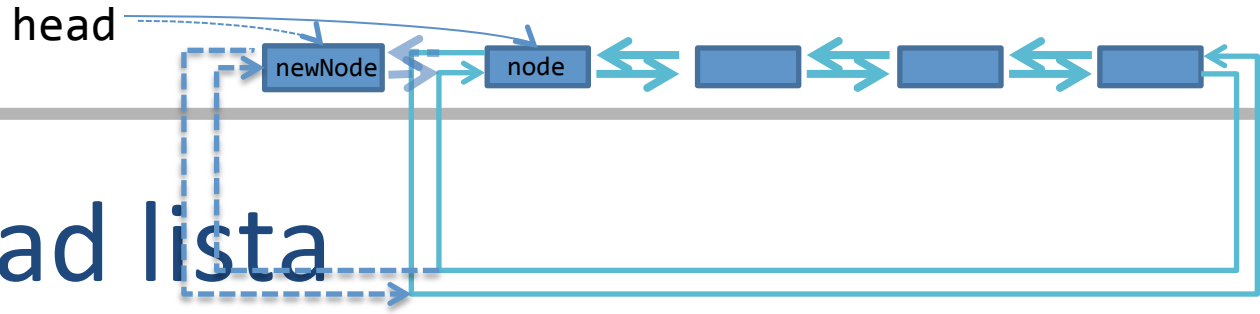

```
struct LinkedList{
    int a;
    struct LinkedList *next;
};

struct LinkedList *head = NULL;

struct LinkedList* insert(
              struct LinkedList *head,
              struct LinkedList *node)
{
    // Stoppa in noden först i listan
    node->next = head;

    // för att uppdatera head
    return node;
};
```

```
// Exempel på insättning av 100 noder i listan.
int main()
{
    struct LinkedList *head = NULL;
    int i;
    for (i=0; i<100; i++)
    {
        // Skapa ny nod
        struct LinkedList* node =
           (struct LinkedList*)
                 malloc(sizeof(struct LinkedList));

        // Initiera noden till något valfritt
        node->a = i;

        // Stoppa in noden i listan och peka head på
        // vår nya nod
        head = insert(head, node);
    }
    return 0;
}
```

Länkade datastrukturer

- Ex. Cyklisk dubbellänkad lista:





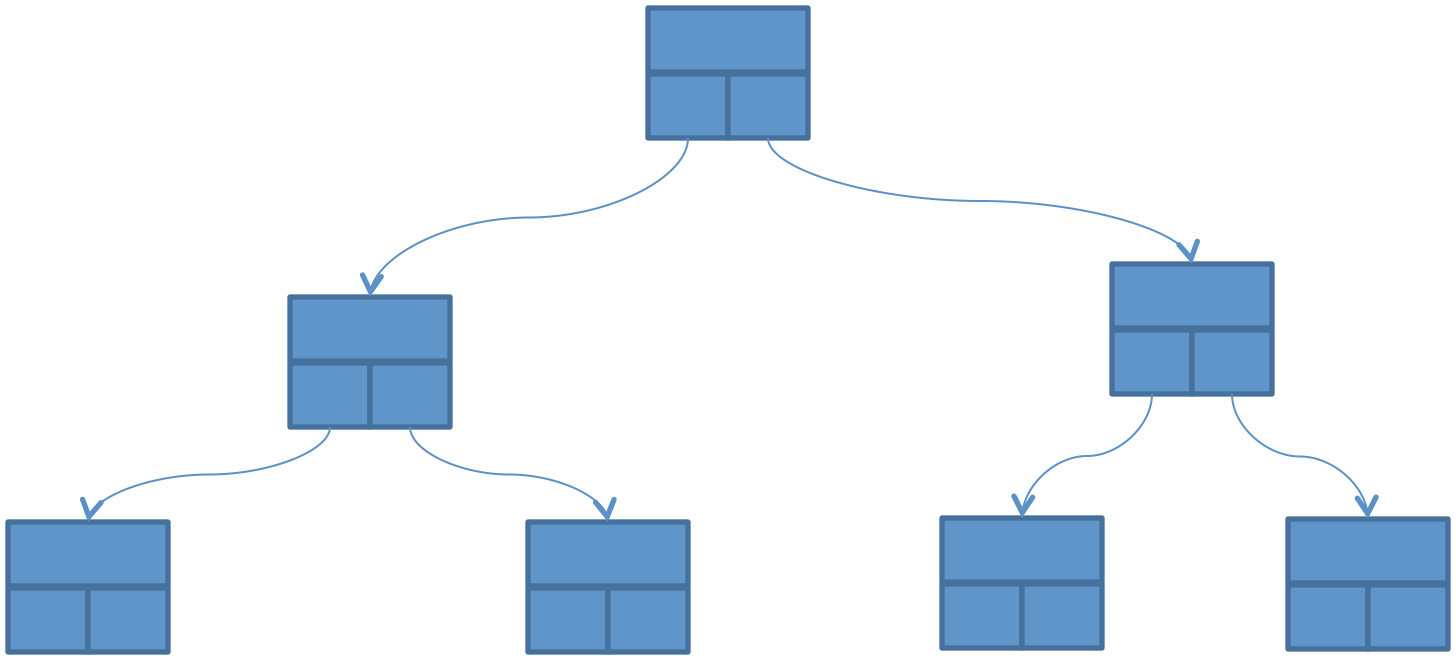
Dubbellänkad lista

```
struct DLinkedList{
    int a;
    struct DLinkedList *next, *prev;
};

struct DLinkedList *head = NULL;

struct DLinkedList* insert(
             struct DLinkedList *node,
             struct DLinkedList *newNode)
{
    // Hantera att node (dvs head) == 0
    if (node == 0) {
        node = newNode;
        node->prev = newNode;
    }
    // Stoppa in vår newNode först i listan
    newNode->next = node;
    newNode->prev = node->prev;
    node->prev->next = newNode;
    node->prev = newNode;
    return newNode;
};
```

```
// Exempel på insättning av 100 noder i listan.
int main()
{
    struct DLinkedList *head = NULL;
    int i;
    for (i=0; i<100; i++) {
        struct DLinkedList *node = (struct DLinkedList*)
                    malloc(sizeof(struct DLinkedList));
        node->a = i;
        head = insert(head, node);
    }
    return 0;
}
```

Länkade datastrukturer

- Ex. träd:



Laboration 3 – enkellänkad lista

- En lista av element som är
  - Dynamiskt allokerade.
  - En `struct`.
- Varje element innehåller
  - En pekare till nästa element.
  - En prioritet.
  - En pekare till data (en sträng i detta fall).

Type "Union"

```
union {
      int d;
      char c;
      float f;
} x;
x.d = 4;
x.c = 'i';
x.f = 3.0;
```

&x.d == &x.c == &c.f
d, c och f delar samma minnesadress. D v s samma minnesadress kan adresseras på tre olika sätt via tre olika variabelnamn.

```
typedef struct {
   union {
      float v[2];
      struct {float x,y;};
   };
} Vec2f;

Vec2f pos;
pos.v[0] = 1; pos.v[1] = 2;
pos.x = 1; pos.y = 2;
```

Exempel: pos.v[0] och pos.x är samma sak. "pos.x" visar tydligt att det är x-koordinaten vi adresserar. "pos.v[i]" är dock användbart om vi vill skriva en loop över x- och y-koordinaten.
Tex:

```
void addVec(Vec2f *a, Vec2f *b)
{
   for(i=0; i<2; i++)
      a->v[i] += b->v[i];
}
```