Föreläsningsöversikt

- Genomgång inlämningsuppgift 2.
- Poster med varierande utseende: `union`–typer.
- Fallanalys: `switch`–satsen.
- Introduktion till inlämningsuppgift 3.
- Funktionspekare, "intelligenta" dataobjekt.

Inlämningsuppgift 2

Skiss för lösningen:

```
#include <stdio.h>
#include <stdlib.h>
struct tree_node { ... };
static void print(struct tree_node *tree) { ... }
struct tree_node *mk_leaf_node(int key) { ... }
static struct tree_node *root = NULL;
void insert_key(int key) { ... }
static void read_and_insert_keys(void) { ... }
int main(void) { ... }
```


Datatyp och print–funktion

```
struct tree_node {
    struct tree_node *left;
    struct tree_node *right;
    int key;
};
static struct tree_node *root = NULL;
static void print(struct tree_node *tree)
{   if (tree != NULL) {
        print(tree->left);
        printf("%d\n", tree->key);
        print(tree->right); /* or loop here */
    }
}
```


Topp–nivån

```
static void read_and_insert_keys(void)
{
    int key;
    while (scanf("%d", &key) == 1)
        insert_key(key);
}
int main(void)
{
    read_and_insert_keys();
    print(root);
    return 0;
}
```

Insättning: rekursiv

```
struct tree_node *insert_recursive(struct tree_node *tree,
                                   int key)
{   if (tree == NULL)
        return mk_leaf_node(key);
    if (key < tree->key)
        tree->left = insert_recursive(tree->left, key);
    else if (key > tree->key)
        tree->right = insert_recursive(tree->right, key);
    return tree;
}
void insert_key_recursive(int key)
{   root = insert_recursive(root, key);
}
```


Insättning: iterativ lösning

Observationer:

- Den rekursiva lösningen gör tilldelningar längs med hela vägen från roten till platsen där det nya lövet ska stoppas in. Endast den sista tilldelningen behövs.
- Det enda som behövs är att vi kommer ihåg den sista trädnoden innan vi kom till platsen för insättningen. Genom att jämföra nycklarna kan vi avgöra om vänstra eller högra delträdet ska tilldelas.

Iterativ insättning: första halvan av koden

```
void insert_key_iterative(int key)
{
    struct tree_node *here = root, *parent = NULL, *leaf;
    while (here != NULL) {
        if (key == here->key)   /* duplicate key */
            return;
        parent = here;
        if (key < here->key)
            here = here->left;
        else
            here = here->right;
    }
```


Iterativ insättning: andra halvan av koden

```
    leaf = mk_leaf_node(key);
    if (parent == NULL) /* special case */
        root = leaf;
    else if (key < parent->key)
        parent->left = leaf;
    else
        parent->right = leaf;
}
```

Iterativ insättning utan specialfall

Observationer:

- Vid avslutningen finns det tre fall att ta hänsyn till: (1) om trädet var tomt, (2) om det vänstra delträdet ska tilldelas, eller (3) om det högra delträdet ska tilldelas.
- Notera att i vart och ett av de tre fallen så tilldelar vi helt enkelt en variabel det nya lövet. Specialfallen talar om vilken variabel som ska tilldelas.
- Antag att vi istället för `parent` har en variabel `prev` som pekar på den variabel som pekar ut den trädnod vi har kommit till. Då räcker det med att tilldela `*prev`, utan några specialfall.


```
void insert_key_iterative_tricky(int key)
{   struct tree_node **prev = &root;
    for(;;) {
        struct tree_node *here = *prev;
        if (here == NULL) {
            *prev = mk_leaf_node(key);
            return;
        }
        if (key < here->key)
            prev = &(here->left);
        else if (key > here->key)
            prev = &(here->right);
        else return;
    }
}
```


Poster med varierande utseende

- I vissa applikationer finns det datatyper där värdena har ska kunna ha olika utseende:
  - Typ “grafiskt objekt”, värden “cirkel”, “rektangel”, osv.
  - Typ “sats”, värden “tilldelning”, “`IF`–sats”, osv.
- Problemet är att finna ett bra sätt att representera dessa typer.
- Varje alternativ är i sig en post–liknande typ, med ett antal fält.

Alternativ 1: “feta” pekare

```
struct circle { double radius; };
struct rect { double height, width; };
enum object_tag { IS_CIRCLE, IS_RECT };
struct graphics_object {
    enum object_tag tag;
    void *pointer; /* Warning! untyped pointer */
};
```

Nackdel: beskrivningen av objektets typ är utanför objektet (kan lättare bli fel), typ–osäker kod eftersom `void*`–pekare måste användas.

Alternativ 2: "feta" structar

```
enum object_tag { IS_CIRCLE, IS_RECT };
struct graphics_object {
    enum object_tag tag;
    struct { double radius; } circle;
    struct { double height, width; } rect;
};
```

Fördel: Pekare till `struct graphics_object` kan bara peka till sådana värden, mindre typosäkert. Nackdel: Storleken i minnet för en sådan post är summan av storlekarna för alla möjliga alternativ, plus tag:en som vi måste ha.

Alternativ 3: unionstyper

```
union graphics_object {
    struct { double radius; } circle;
    struct { double height, width; } rect;
};
enum object_tag { IS_CIRCLE, IS_RECT };
struct graphics_object_with_tag {
    enum object_tag tag;
    union graphics_object go;
};
```

Skillnaden mellan `union` och `struct` är att i en `struct` placeras fälten efter varandra, men i en `union` läggs fälten *ovanpå* varandra så att de delar minne. Storleken på en `union` är lika med storleken på det största alternativet i den.


```
struct graphics_object {
    enum object_tag tag;
    union {
        struct { double radius; } circle;
        struct { double height, width; } rect;
    } u;
};
```

Typiskt så "bakar man in" unionstypen i den struct som den tillhör, eftersom den oftast inte är användbar utan den tag som finnns i structen.

Om alla alternativ har ett eller flera gemensamma fält, så brukar man bryta ut dem och placera dem i början av structen. Exempel: Det finns många olika sätt att skriva uttryck i C, men varje uttryck har en typ.

Unionstyper: fallanalys

```
double area(struct graphics_object *p)
{
    if (p->tag == IS_CIRCLE) {
        double r = p->u.circle.radius;
        return 3.14 * r * r;
    } else if (p->tag == IS_RECT)
        return p->u.rect.height * p->u.rect.width;
    else
        error(...);
}
```

Fallanalys med switch–satser

```
double area(struct graphics_object *p)
{   switch (p->tag) {
    case IS_CIRCLE: {
        double r = p->u.circle.radius;
        return 3.14 * r * r;
    }
    case IS_RECT:
        return p->u.rect.height * p->u.rect.width;
    default: error(...);
    }
}
```

`switch` fungerar ungefär som en serie av `if`–tester, men det som ska jämföras beräknas endast en gång.

Kommentarer om Inlämningsuppgift 3

- Datatypen för uttryck är lämpligen en unionstyp inuti en `struct` med en tag som beskriver vilket alternativ som gäller.
- `print`–funktionen bör vara en rekursiv funktion, ungefär som den för binära träd, som dessutom gör en fallanalys på varje uttryck.
- `eval`–funktionen bör också vara en rekursiv funktion som gör fallanalys på varje uttryck.

Funktionspekare

Pekare till funktioner är vanliga värden och kan därför skickas som argument till funktioner, returneras från funktioner, lagras i variabler, och anropas. Detta är en kraftfull egenskap i språket.

Att ha funktionspekare i datastrukturer utgör grunden för *intelligenta* data: data som själva “vet” hur vissa operationer ska utföras. Så kallad *objekt–orienterad* programmering är baserad på detta paradigm.

Deklaration

`typ (*namn)(argument..)`

deklarerar att **namn** är en variabel som kan peka ut funktioner med angivna argument– och resultattyper. Exempel:

`double (*compute)(double, double)`

Uttryck

Namnet på en funktion konverteras automatiskt till en pekare till funktionen i uttryck. (Analogt med arrayer.)

Uttrycket `funpek(argument..)` anropar den funktion som funktionspekaren pekar på.

Objekt–orienterad programmering

```
struct character_device {
    int (*nextchar)(struct character_device*);
    ... /* state fields, likely in a union */
};

int nextchar(struct character_device *p)
{
    return p->nextchar(p);
}
```



```
int serial_port_nextchar(struct character_device *p)
{
    /* use HW access to the serial port
       and return the character */
}

struct character_device *make_serial_port(void)
{
    struct character_device *p;
    p = malloc(sizeof(struct character_device));
    p->nextchar = serial_port_nextchar;
    ...
    return p;
}
```