

2. LAUSEKIELISEN OHJELMOINNIN PERUSTEIDEN LYHYT KERTAUS

[2.1 Yleistä](#)

[2.2 Peruskäsitteitä](#)

[2.3 Taulukko](#)

[2.4 Linkitetyn listan \(linked list\) toteutus](#)

[2.5 Tapausesimerkki Polynomi](#)

[2.6 Rekursio](#)

2.1 Yleistä

- **Esitiedot:** kyky ymmärtää yksinkertaista lausekielistä ohjelmakoodia sekä alkeistyypeillä toteutettuja yksinkertaisia rakenteita
- **Erityisesti:** lineaaristen tietorakenteiden toteutus
- **Tietorakenne on lineaarinen, jos kaikki sen alkiot on ryhmitetty peräkkäin.**
- **Tällaista rakennetta voidaan käyttää esim.**
 - yksinkertaisena hakurakenteena,
 - tilapäisenä tallennusrakenteena.

2.2 Peruskäsitteitä / Muuttujat

- **Muuttujiin talletetaan dataa**
 - Muuttujiin liittyy *tunnus, osoite ja tyyppi*
 - Muuttujat tulee useimmissa kielissä määritellä, esim.

```
int count;
double table[1000];
```

- **Vakiot** ovat arvoja, joita ei voi muuttaa

```
final int SIZE = 100; (Java)
```

```
#define SIZE 100 (C)
```

2.2 Peruskäsitteitä / Muuttujat

- **Muuttuja voi sisältää myös osoitteen toiseen muuttujaan:**

```
double sum = 0;
double *p;
p = &table[0];
for (int i = 0; i < 1000; i++) {
    sum += *p;
    p++;
}
```

- **Sama kuin**

```
double sum = 0;
for (int i = 0; i < 1000; i++)
    sum += table[i];
```

- **Tässä esimerkissä osoitinten käyttö tarpeetonta, mutta C-kielessä esim. merkkijonoja käsitellään usein osoittimien avulla.**

2.2 Peruskäsitteitä / Muuttujat

- **Osoitinmuuttujien suurin merkitys on mahdollisuus luoda ja käsitellä *dynaamisia tietorakenteita*, esim. linkitettyjä listoja tai puita.**
 - **Rakenteen solmut ovat muuttujia, joihin viitataan osoitinmuuttujien avulla.**
 - **Solmuja voidaan luoda tarpeen mukaan ja niiden keskinäistä järjestystä voidaan vaihtaa.**
 - **C-kielessä osoitinten käsittely on eksplisiittistä.**
 - **Javassa rakenteet toteutetaan luokkien ja olioiden avulla. Luokka-tyyppinen muuttuja on itse asiassa osoitin muistissa olevaan olioon.**
 - **Tärkeä erikoistapaus on tyhjä osoitin NULL.**

5

2.2 Peruskäsitteitä / Tyypit

- **Tyypit jaetaan *alkeistyyppihin* ja *rakenteisiin tyyppihin***
 - **Alkeistyyppinä ovat kokonaisluvut, reaaliluvut, merkkietu ja looginen tieto**
 - **Rakenteista tyyppiä olevat alkiot koostuvat alkeistyypeistä, esimerkiksi taulukot, tietueet, oliot**
 - **Rakenteet voivat olla mielivaltaisen monimutkaisia.**
 - **Taulukot ja tietueet ovat luonteeltaan *staattisia* tietorakenteita.**
 - **Niiden koko on sidottu määrittelyyn**
 - **On olemassa myös dynaamisia taulukoita**
 - **Dynaamiset tietorakenteet luodaan osoitinmuuttujien avulla (C) tai oliorakenteina (Java).**

6

2.2 Peruskäsitteitä / Ohjelman suoritus

- **Ohjelman suorittamat toiminnot määritellään:**
 - **Muuttujalle voidaan *sijoittaa* uusi arvo.**
 - **Lausekkeissa lasketaan jokin arvo**
 - **Lausekkeella voi olla myös *sivuvaikutuksia* muuttujien arvoihin**
 - **Kontrollilauseet ohjaavat suoritusjärjestystä**
 - ***Ehtorakenteet* (if - else, switch)**
 - ***Toistorakenteet* (for, while-do, do-while)**
 - **Isommat kokonaisuudet lasketaan usein *funktioiden, proseduurien* tai *metodien* avulla. Laskentaa ohjataan *parametreilla*.**

7

2.3 Taulukko

- **Kuuluu yleisimpien ohjelmointikielten perusrakenteisiin**
- **Taulukon määrittely C-kielessä:**
 - `< type > < name >[<number of elements>]`
 - **esim.**

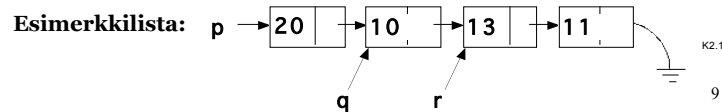
```
int numbers[10];
numbers[5] = 1;
```
 - **indeksointi alkaa nolasta (0 - N-1)**
- **Moniulotteiset taulukot**
- **Dynaaminen taulukko**

8

2.4 Linkitetyn listan (linked list) toteutus

- Lista toteutetaan tyypillisesti osoitinmuuttujien ja dynaamisen muistinvarauksen avulla.
- Voidaan toteuttaa myös pelkän taulukon avulla
- Tyypillinen määrittely C-kielessä:

```
struct node {
    int data;
    struct node * next;
};
typedef struct node * List_t;
List_t p, q, r;
```



- Eräs määrittelytapa Java-kielessä:

```
class ListNode
{
    private Object data;
    private ListNode next;

    ListNode(Object element)
    {
        data = element;
        next = null;
    }
}

Listnode p, q, r;
```

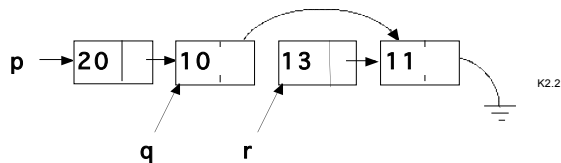
Alkion poistaminen käy helposti, kun on käsillä osoitin poistettavaa alkioita edeltävään alkioon:

(C)

```
r = q->next;
q->next = r->next;
free(x);
```

(Java)

```
r = q.next;
q.next = r.next;
```



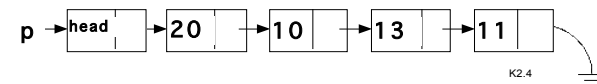
Ensimmäisen alkion poistaminen on erikoistapaus:

```
r = p;
p = p->next;
free(x);
```

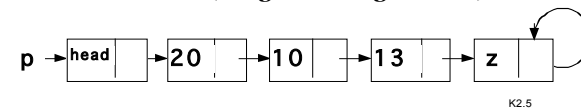
```
r = p;
p = p.next;
```

- Kuinka hoitaa myös ensimmäinen alkio samalla koodinpätkällä kuin muutkin?

=> Määritellään yksi ylimääräinen *header-* alkio, jonka tarkoitus on aloittaa lista



- Samasta syystä (erikoistapausten välttäminen) toisinaan määritellään listan loppuun ylimääräinen alkio z, josta on viittaus itseensä (Sedgwick: Algorithms)



```

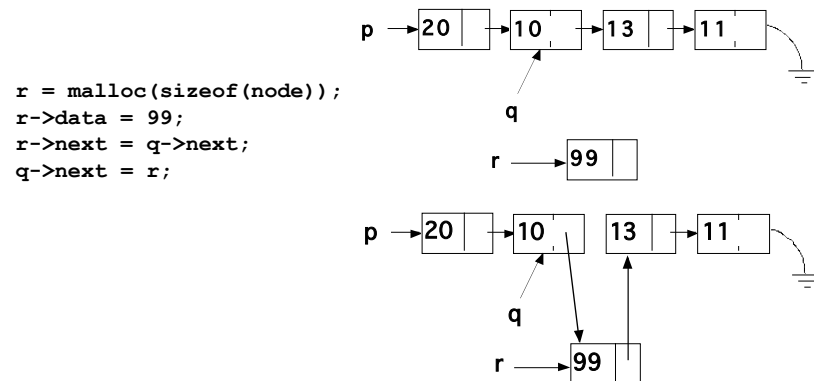
class LinkedList {
    private ListNode header;
    private static ListNode z = new ListNode(null);

    LinkedList ()
    {
        header = new ListNode(null);
        header.next = z;
    }

    public AddFirst(Object element)
    {
        ListNode temp = new Listnode(element);
        temp.next = header.next;
        header.next = temp;
    }
}
    
```

- Useampi lista voi käyttää samaa z-alkiota
- Listan loppumisen tarkastaminen:
 - if (q == q->next) ...; TAI if (q == z) ...

- Alkion lisääminen on helppoa, kun käsillä on osoitin siihen alkioon, jonka jälkeen uusi alkio sijoitetaan



Esimerkkejä listan käsittelyfunktioiden toteuttamisesta (C):

```

TULOSTA: void print_list(List_t head) {
    List_t temp;

    temp = head->next;
    if (temp == z)
        printf("List is empty\n");
    else {
        while (temp != z) {
            printf("%d ", temp->data);
            temp = temp->next;
        }
        printf("\n");
    }
} /* print_list */
    
```

Esimerkkejä listan käsittelyfunktioiden toteuttamisesta (Java):

```

TULOSTA: Public void print_list(LinkedList head) {
    ListNode temp;

    temp = head.next;
    if (temp == z)
        System.out.print("List is empty");
    else {
        while (temp != z) {
            System.out.print(temp.data+" ");
            temp = temp.next;
        }
    }
}

```

17

LISÄÄ ALKIO LISTAN ALKUUN :

```

List_t insert(List_t head, int key)
{
    List_t temp;

    temp = get_node();
    temp->data = key;
    temp->next = head->next;
    head->next = temp;
    return (temp);
} /* insert */

```

18

HAE ALKIO :

```

List_t search(List_t head, int key)
{
    List_t temp;

    z->data = key;
    temp = head->next;

    while (temp->data != key)
        temp = temp->next;

    return (temp);
} /* search */

```

19

POISTA ALKIO :

```

Boolean delete(List_t head, int key){
    List_t temp, previous;
    z->data = key;
    previous = head;
    temp = head->next;

    while (temp->data != key) {
        temp = temp->next;
        previous = previous->next;
    }

    if (temp != z) {
        previous->next = temp->next;
        free (temp);
        return (TRUE);
    } else
        return (FALSE);
} /* delete */

```

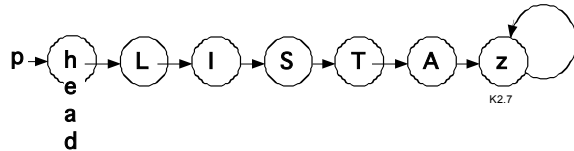
20

- **Lista voidaan toteuttaa myös 2 taulukon avulla (useampi lista voi käyttää samoja taulukoita)**

```

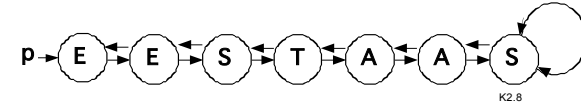
data[0..N]      arvot
next[0..N]     osoittimet
    
```

	data	next
0	head	4
1	z	1
2	T	6
3	I	5
4	L	3
5	S	2
6	A	1



21

- **Kaksisuuntainen linkitetty lista on nopeampi käsitellä, mutta vaatii enemmän muistitilaa.**



```

struct node {
    char data;
    struct node *next, *previous;
};

typedef struct node * Dlist_t;
    
```

22

2.5 Tapausesimerkki POLYNOMI

$$\sum A_i X^i \quad i=0 \dots N$$

- **Jos polynomi on 'tiheä' l. suurin osa kertoimista poikkeaa nolasta**

=> Esitetään kertoimet taulukossa

```
#define MAX 1000
```

```

struct polynomi {
    int kerroin[MAX+1];
    int asteluku;
};
    
```

```
typedef struct polynomi Polynomi;
```

23

- **Jos polynomi on 'harva'**

=> Esitetään kertoimet järjestettynä listana

```

struct polynomi {
    int kerroin, eksponentti;
    struct polynomi * next;
};
    
```

```
typedef struct polynomi * Polynomi;
```

- **Ei rajaa polynomien asteluvulle!**

24

▪ **Toteutetaan polynomeille operaatio:**

```
void A_kertaa_B_on_C(Polynomi a,b,c)
{
    /* Toteutus riippuu valitusta
       tietorakenteesta */
    ...
}
```

=> käsitellään polynomia abstraktiona!

!! Kutsuvan ohjelman ei tarvitse tietää miten kertoimet on talletettu, toteutustapaa voidaan vaihtaa tarpeen mukaan !!

2.6 REKURSIO

▪ **Rekursiivinen ohjelma**

- Kutsuu itseään

▪ **Rekursiivinen funktio**

- On määritelty itsensä avulla

- **Esim. Fibonacci-luvut:**

$$\begin{aligned} X(i) &= X(i-1) + X(i-2), \\ X(0) &= X(1) = 1 \end{aligned}$$

▪ **Rekursio ei voi jatkua loputtomiin**

- Täytyy löytyä päätösehto jonka toteutuminen lopettaa rekursion

- **Esim. Kertoma:**

Rekursio: $N! = N \cdot (N-1)!$

Päätösehto: $0! = 1$

Rekursion neljä kultaista sääntöä:

1. Perustapaukset (ratkaistavissa ilman rekursiota)
2. Edistyminen (liikutaan perustapaukseen kohti)
3. Oletus (kaikki rekursiiviset kutsut toimivat)
4. Vältä turhaa työtä (jokainen tapaus ratkaistaan vain kerran)

▪ **Esim. Kertoma rekursiivisena:**

```
int factorial(int n)
{
    if (n==0)
        return 1;
    else
        return n*factorial(n-1);
}
```

▪ **Esim. Kertoma ei-rekursiivisena:**

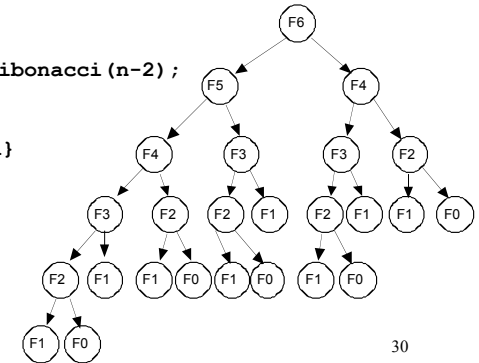
```
int factorial(int n)
{
    int i, fact;
    fact = 1;
    for (i=2; i<=n; i++)
        fact= fact*i;
    return fact;
}
```

Esim. Fibonacci-luvut:

```
int fibonacci(int n)
{
    if (n<=1)
        return 1;
    else
        return
            fibonacci(n-1)+fibonacci(n-2);
}
```

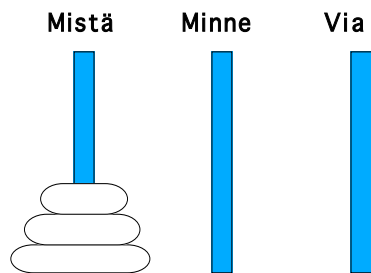
{Rikkoo neljättä sääntöä}

▪ **Parempi toteuttaa ei-rekursiivisena**



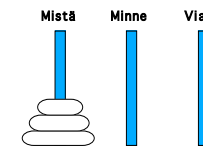
▪ **Esim. Hanoi tornit:**

Siirrä renkaat toiseen tappiin yksi kerrallaan niin, että isompi rengas ei koskaan ole pienemmän päällä:



K2.17

```
void hanoi(int N, int mista, int minne, int via)
{
    if (N==1)
        printf("Siirrä rengas tapista %d
            tappiin %d kiitos\n", mista, minne);
    else {
        hanoi(N-1, mista, via, minne);
        hanoi(1, mista, minne, via); // printf
        hanoi(N-1, via, minne, mista);
    }
}
```



K2.17

▪ **Hajota ja hallitse –menetelmä algoritmisuunnittelussa**

Tehdään kaksi rekursiivista kutsua, toinen datan yhteen puolikkaaseen ja toinen jäljelle jääneeseen osaa.

Eräs hyvin keskeinen periaate algoritmisuunnittelussa.