# Latmannahashning

Hasha bara på dom tre första bokstäverna i söknyckeln. Använd sedan binärsökning.

Lämpligt för sökning med få diskaccesser i stor text när indexet inte kan ligga i primärminnet.

Exempel: Indexera stort svensk-engelskt lexikon.

Givet:
- Stor fil $L$ med hela lexikontexten.
- Index $I$ där varje rad är: sökord position i $L$

Skapa sökindex: Sortera $I$ med sort.
Skapa en indexarray $A[abc]$ som för varje abc anger var i $I$ första ordet som börjar så finns.

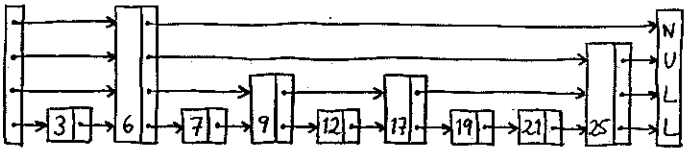
Förberedelse inför sökningar:
Läs in $A$ från fil. Öppna filerna $I$ och $L$.

Sökningsalgoritm(w)=

```
wprefix ← första 3 bokst. i w
i ← A[wprefix]
j ← A[wprefix+1]
WHILE j-i > 1000 DO
    m ← ⌊(i+j)/2⌋
    Gå till position m i I
    Läs in nästa ord från I i s
    IF s≤w THEN i←m
      ELSE j←m
Gå till position i i I
WHILE TRUE DO
    Läs in nästa ord från I i s
    IF s=w THEN
        Läs in positionen i x
        RETURN x
    IF s>w THEN
        RETURN NOTFOUND
```

# Skipplistor

- Probabilistisk datastruktur
- Enkel att implementera
- I allmänhet lika snabb som balanserade träd att söka i och enklare att ändra på.




En elementpost deklareras av typen

```
struct skipnode {
    int key
    struct skipnode *forward[1];
};
```

olika många pekare beroende på nivå

Allokering av en elementpost på nivå k:

```
struct skipnode *e = malloc(sizeof(*e)+
    (k-1) * sizeof(struct skipnode *));
```

# Sökning i skipplistor

Börja i startpostens högsta nivå

Om forward-pekaren pekar på ett för stort element går vi ner en nivå, annars går vi framåt

```
Search(list, searchKey) =
    x ← list.header;
    for i ← list.level downto 1 do
        while x.forward[i].key < searchKey do
            x ← x.forward[i];
    x ← x.forward[1];
    if x.key = searchKey then return x
        else return NOTFOUND;
```

Vid insättning och borttagning av element söker man på samma sätt, men håller reda på alla $x$ i slutet av for-slingan.

När man skapar ett nytt element slumpar man fram dess nivå:

```
RandomLevel() =
    newLevel ← 1;
    while random() < p do
        newLevel ← newLevel + 1;
    return min(newLevel, MAXLEVEL)
```

# Analys av skipplistor

Anta att $n$ = antal element i datastrukturen
$p$ = sannolikheten att ett element på nivå $i$ också finns på nivå $i+1$

**Genomsnittligt antal pekare för ett element:**

$$\frac{\sum_{k=1}^{\infty} \text{antal element på nivå } k}{n} = \frac{\sum_{k=1}^{\infty} n \cdot p^{k-1}}{n} = 1+p+p^2+\ldots = \frac{1}{1-p}$$

**Nivå med bara $\frac{1}{p}$ element:**

$$n \cdot p^{k-1} = \frac{1}{p} \iff n = \left(\frac{1}{p}\right)^k \iff k = \log_{\frac{1}{p}} n$$ — kalla detta $L(n)$

**Förväntat antal jämförelser vid sökning:**

Vi mäter sökstigens längd baklifrån. (från den funna posten åt vänster och uppåt till startposten).
Vid varje rörelse åt vänster eller uppåt görs en jämförelse.

# Analys av sökstigens längd

Låt $C(k)$ = förväntad längd på en sökstig som går upp $k$ nivåer.

Rekursionsekvation: $C(0) = 0$

$$C(k) = (1-p)(1+C(k)) + p(1+C(k-1))$$

(1−p): sht att vi är kvar på samma nivå
p: sht att vi byter nivå

$$\Rightarrow p \cdot C(k) = 1 + p \cdot C(k-1) \Rightarrow C(k) = \frac{1}{p} + C(k-1) \Rightarrow C(k) = \frac{k}{p}$$

Längd av sökstig från nivå 1 till nivå $L(n)$ är $C(L(n)-1) = \frac{L(n)-1}{p}$

På nivå $L(n)$ förväntas $\frac{1}{p}$ element.

Om vi startar på nivå $L(n)$ blir totala längden $\frac{L(n)-1}{p} + \frac{1}{p} = \frac{L(n)}{p}$

Hur mycket längre blir stigen om vi börjar sökningen från översta nivån?

$$Pr[\text{Högsta nivån} > k] = 1 - Pr[\text{Högsta nivån} \leq k] =$$
$$= 1 - (Pr[\text{element i har nivå} \leq k])^n = 1 - (1-p^k)^n \leq n \cdot p^k$$

Låt $X$ = antal nivåer över $L(n)$ som högsta nivån är, givet att vi har $\frac{1}{p}$ element på nivå $L(n)$.

$$E[X] = \frac{1}{p}(p^1 + p^2 + p^3 + \ldots) = 1+p+p^2+\ldots = \frac{1}{1-p}$$

Total sökstigslängd: $\frac{L(n)}{p} + \frac{1}{1-p}$

# Skipplistor i praktiken

Sökstigens längd om $p = \frac{1}{2}$: $2\log_2 n + 2$

$p = \frac{1}{4}$: $4\log_4 n + \frac{4}{3} = \frac{4\log_2 n}{\log_2 4} + \frac{4}{3} = 2\log_2 n + \frac{4}{3}$

Antal pekare om $p = \frac{1}{2}$: $\frac{1}{1-\frac{1}{2}} = 2$

$p = \frac{1}{4}$: $\frac{1}{1-\frac{1}{4}} = \frac{4}{3} = 1.33$

Söktiden är ungefär lika för $p = \frac{1}{2}$ och $p = \frac{1}{4}$,
minnesutrymmet är betydligt mindre för $p = \frac{1}{4}$,
men söktidens varians ökar om $p = \frac{1}{4}$.

Jämförelse med implementation av balanserade träd:

Söktiden för skipplista och balanserat träd är ungefär lika.

Insättnings- och borttagningstiden för balanserat träd är ungefär dubbelt så lång som för skipplista.

Implementation av skipplistor som du kan pröva:
/info/adk02/skiplist