

FÖRELÄSNING 3

ADK

DATASTRUKTURER

REPETITION

HASHFUNKTIONER

FÖRBERÄKNAD FUNKTION

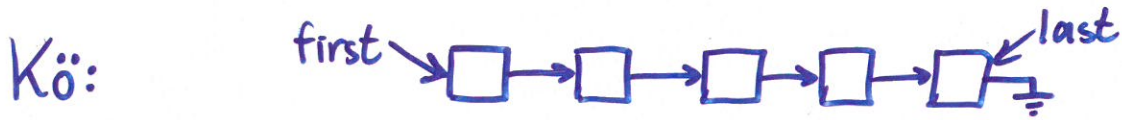
TRIE

LATMANSHASHNING

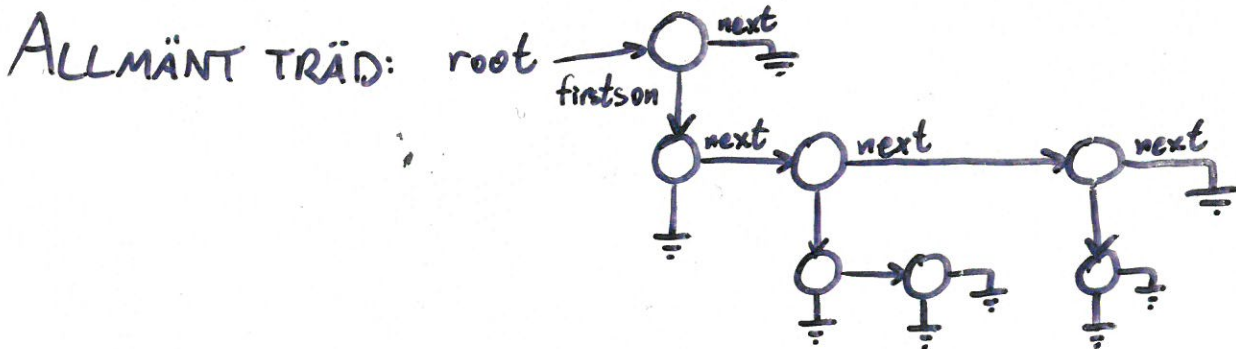
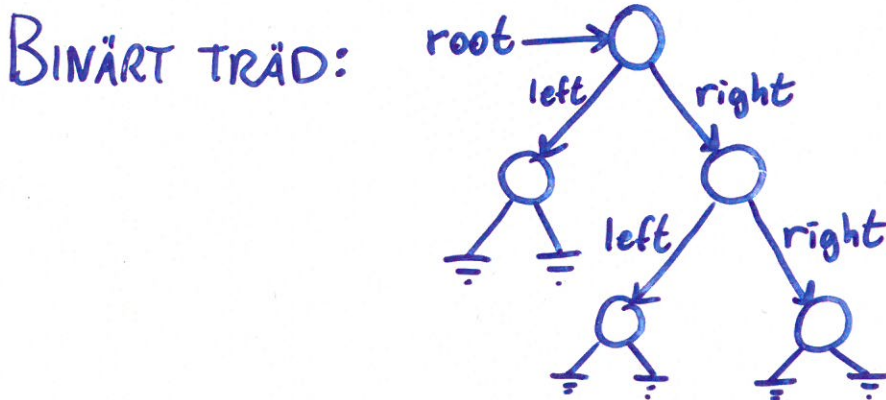
SKIPPLISTA

ÖVERSIKT ÖVER KÄNDA DATASTRUKTURER

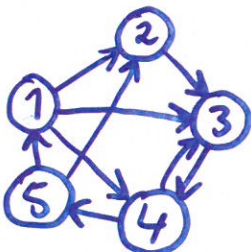
LINJÄRA STRUKTURER:



TRÄD OCH GRAFER:



RIKTAD GRAF:



REPRESENTERAD SOM GRANNMÄTRIS

	1	2	3	4	5
1		1	1	1	
2			1		
3				1	
4			1		1
5	1	1			

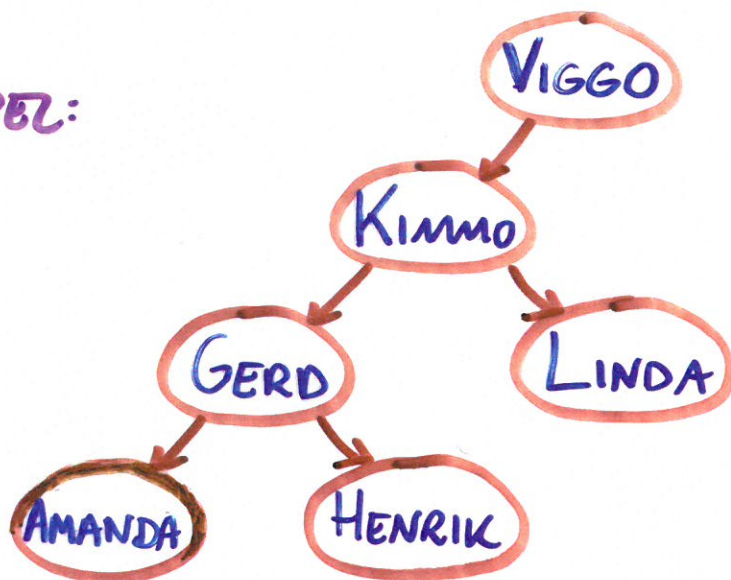
REPRESENTERAD SOM KANTMÄTRIS:

	1	2	3	4	5
1		-1	-1	-1	1
2	1		-1		1
3		1		-1	1
4			1		-1
5				1	-1

BINÄRA SÖKTRÄD

ETT TRÄD SOM ÄR SORTERAT SÅ ATT MINDRE ELEMENT ÄR TILL VÄNSTER KALLAS SÖKTRÄD.

EXEMPEL:



```
TREESearch(ROOT, x) = // SÖKER EFTER x I SÖKTRÄD
IF ROOT = NIL THEN RETURN NIL
IF ROOT.KEY > x THEN
  RETURN TREESearch(ROOT.LEFT, x)
IF ROOT.KEY < x THEN
  RETURN TREESearch(ROOT.RIGHT, x)
RETURN ROOT
```

PRE ROOT ÄR ROT TILL ETT SÖKTRÄD

POST (x FINNS I TRÄDET ⇒ x-NOD I TRÄDET RETURNERAS) ^
(x FINNS INTE I TRÄDET ⇒ NIL RETURNERAS)

KORREKTHETSBEVIS GÖRS MED INDUKTION ÖVER TRÄDSTRUKTUREN (VARJE DELTRÄD ÄR ETT SÖKTRÄD).

BALANSERADE TRÄD

PROBLEM MED BINÄRA SÖKTRÄD:

OBALANSERADE TRÄD TAR TID ATT SÖKA 1.



DÅLIG LÖSNING:

BALANSERA TRÄDET, TAR TID $O(n)$.

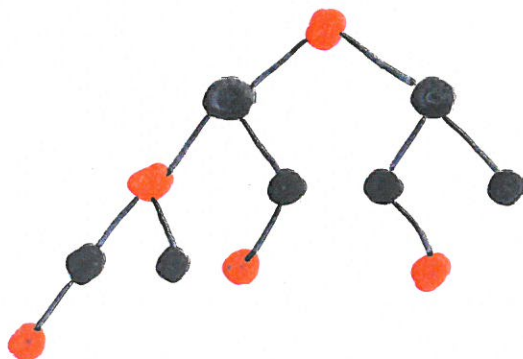


BÄTTRE LÖSNING:

HÅLL TRÄDET BALANSERAT EFTER VARJE INSÄTTNING OCH BORTTAGNING.

RÖD SVARTA TRÄD REALISERAR DETTA GENOM ATT VIDMÅKTHÅLLA FÖLJANDE TRE EGENSKAPER:

1. VARJE TRÄDNOD ÄR ANTINGEN **RÖD** ELLER SVART.
2. EN **RÖD** NOD KAN INTE HA ETT **RÖTT** BARN
3. I VARJE STIG FRÅN EN NOD TILL VILKET LÖV SOM HELST UNDER NODEN FINNS DET LIKA MÅNGA SVARTA NODER.



RÖD SVARTA TRÄD

SÖKNING

ANVÄND VANLIG TRÄDSÖKNINGSALGORITM
SOM TAR TID $O(\text{TRÄDDJUPET})$.

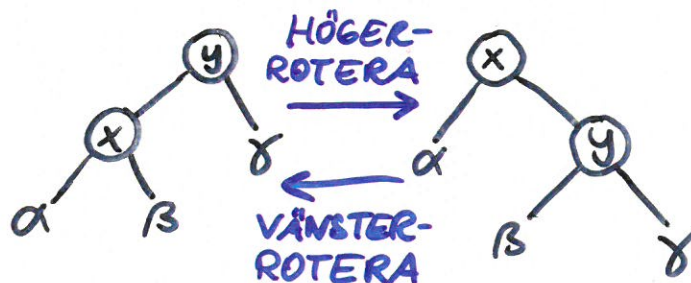
STÖRSTA TRÄDDJUPET I ETT RÖDSVART TRÄD
BEGRÄNSAS AV $2\log_2 n \Rightarrow$ SÖKTID $O(\log n)$

INSÄTTNING

SÄTT IN ELEMENTET MED VANLIG TRÄDINSÄTTNING
SOM ETT RÖDFÄREAT LÖV.

OM NYA ELEMENTETS FAR OCKSÅ VAR RÖD UPPFYLLS INTE
EGENSKAP 2 SÅ VI MÅSTE OMFORMA TRÄDET MED
HJÄLP AV HÖGST $O(\log n)$ FÄRGÄNDRINGAR OCH

ROTATIONER:



\Rightarrow INSÄTTNINGSTID $O(\log n)$

BORTTAGNING

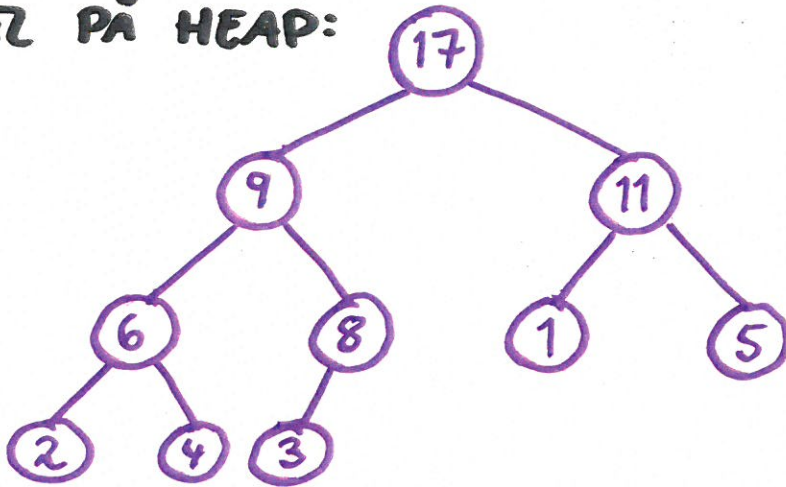
TA BORT NODEN OCH FIXA TILL SOM OVAN \Rightarrow TID $O(\log n)$

HEAPAR — SNABB REPRESENTATION AV PRIORITETSKÖER

KOMPLETT BINÄRTRÄD — ALLA NIVÅER I TRÄDET ÄR FYLDA UTOM DEN SISTA. DEN SISTA SKA FILLAS FRÅN VÄNSTER.
HEAPORDNING: STÖRRE ELEMENT OVANPÅ MINDRE, DVS VARJE NODS BARN ÄR MINDRE ÄN FÖRÄLDERN (ELLER LIKASTORA).

STÖRSTA ELEMENTET FINNS DÄRFÖR ALLTID I ROTEN.

EXEMPEL PÅ HEAP:



EFFEKTIV LAGRING AV KOMPLETT BINÄRTRÄD:

LAGRA I EN ARRAY A SÅ ATT

- ROTEN FINNS I $A[1]$
- BARNEN TILL NODEN $A[i]$ FINNS I $A[2i]$ OCH $A[2i+1]$

EXEMPEL: HEAPEN OVAN LAGRAS SOM

A:	17	9	11	6	8	1	5	2	4	3
	1	2	3	4	5	6	7	8	9	10

PRIORITETSKÖER

DATASTRUKTUR DÄR VARJE POST HAR EN PRIORITET,
DVS ETT TAL SOM ANGER HUR VIKTIG POSTEN ÄR.

OPERATIONER:

INSERT — STOPPAR IN EN NY POST I PRIORITETSKÖN

REMOVE — PLOCKAR UT POSTEN MED HÖGST PRIORITET
UR KÖN OCH RETURNERAR DEN.

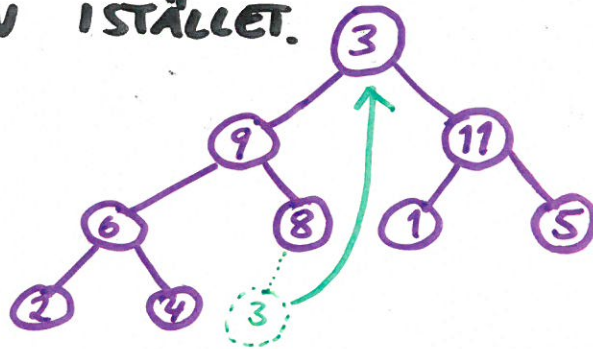
EXEMPEL PÅ ANVÄNDNINGSMOMRÅDEN:

- JOBBHANTERING PÅ FLERANVÄNDAR DATORER
— JOBBET MED HÖGST PRIORITET SKA KÖRAS FÖRST
- SIMULERING AV HÄNDELSE — VARJE HÄNDELSE SKA
INTRÄFFA VID EN VSS TIDPUNKT; HÄNDELSENA SKA
BEARBETAS I TIDSORDNING.
- SORTERING — LÅT SORTERINGSNYCKLARNÄ ANGE
PRIORITETEN
 1. STOPPA IN ALLA POSTER SOM SKA
SORTERAS I PRIORITETSKÖN.
 2. PLOCKA UT EN POST I TAGET. POSTERNA
KOMMER I OMVÄND SORTERINGSORDNING,
DVS STÖRSTA FÖRST OCH MINSTA SIST.

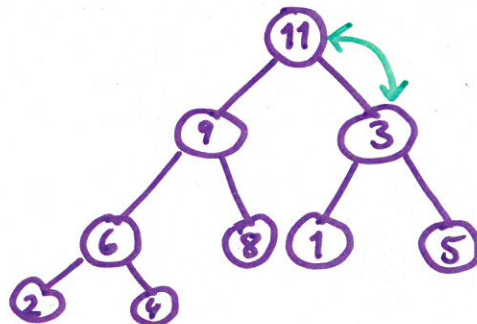
REALISERING AV OPERATIONEN REMOVE I EN HEAP

ELEMENTET SOM SKA TAS BORT LIGGER I ROTEN ($A[1]$)

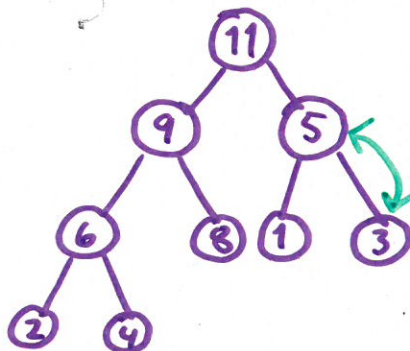
- SPARA UNDAN ELEMENTET SOM LIGGER I ROTEN.
- TA BORT DET SISTA ELEMENTET I HEAPEN OCH LÄGG DET I ROTEN I STÄLLET.



- ÄR ROTELEMENTET MINDRE ÄN DEN STÖRSTA AV SINA SÖNER? BYT I SÅ FALL ROTEN OCH STÖRSTA SONEN.



- ÄR ELEMENTET FORTFARANDE MINDRE ÄN DEN STÖRSTA AV SINA SÖNER? BYT I SÅ FALL OCH GÖR OM DENNA PÅNY



- NU ÄR HEAPEN ORDAD I GEN. (TOG TID $O(\log n)$)

downheap

The standard reference for this is Knuth's "The Art of Computer Science", chapter 6.4. He recommends the hash

```
for (hash=len; len--;)
{
    hash = ((hash<<5) ^ (hash>>27)) ^ *key++;
}
hash = hash % prime;
```

Unfortunately, that hash is only mediocre. The problem is that it doesn't really mix them. Every input bit affects only a few other bits. If two input bits land on the same hash bit, they cancel each other out. This is a problem because the Sparc64 has a 230 bit hash (230 times slower than addition on a Sparc64).

- The keys are unaligned variable-length byte arrays.
- Sometimes keys are several such arrays.
- Sometimes a set of independent hash functions were required.
- Average key lengths ranged from 8 bytes to 200 bytes.
- Keys might be character strings, numbers, bit-arrays, or weirder
- Table sizes could be anything, including powers of 2.
- The hash must be faster than the old one.
- The hash must do a good job.

Most hashes can be modeled like this:

```
initialize(internal state)
for (each text block)
{
    combine(internal state, text block);
    mix(internal state);
}
return postprocess(internal state);
```

```

#define hashsize(n) ((ub4)1<<(n))
#define hashmask(n) (hashsize(n)-1)

/* mix -- mix 3 32-bit values reversibly.
   If mix() is run forward, every bit of c will change betwe
   2/3 of the time.
   mix() takes 36 machine instructions, but only 18 cycles c
   machine (like a Pentium or a Sparc). No faster mixer see
   that's the result of my brute-force search. There were a
   hashes to choose from. I only tested about a billion of
*/

```

```

#define mix(a,b,c) \
{ \
  a -= b; a -= c; a ^= (c>>13); \
  b -= c; b -= a; b ^= (a<<8); \
  c -= a; c -= b; c ^= (b>>13); \
  a -= b; a -= c; a ^= (c>>12); \
  b -= c; b -= a; b ^= (a<<16); \
  c -= a; c -= b; c ^= (b>>5); \
  a -= b; a -= c; a ^= (c>>3); \
  b -= c; b -= a; b ^= (a<<10); \
  c -= a; c -= b; c ^= (b>>15); \
}
/*

```

```

/*----- handle most of the key *
while (len >= 12)
{
    a += (k[0] + ((ub4)k[1]<<8) + ((ub4)k[2]<<16) + ((ub4)k[3]<<24));
    b += (k[4] + ((ub4)k[5]<<8) + ((ub4)k[6]<<16) + ((ub4)k[7]<<24));
    c += (k[8] + ((ub4)k[9]<<8) + ((ub4)k[10]<<16) + ((ub4)k[11]<<24));
    mix(a,b,c);
    k += 12; len -= 12;
}

/*----- handle the last 11 bytes */
c += length;
switch(len)
{
    case 11: c+=((ub4)k[10]<<24);
    case 10: c+=((ub4)k[9]<<16);
    case 9 : c+=((ub4)k[8]<<8);
    /* the first byte of c is reserved for the length */
    case 8 : b+=((ub4)k[7]<<24);
    case 7 : b+=((ub4)k[6]<<16);
    case 6 : b+=((ub4)k[5]<<8);
    case 5 : b+=k[4];
    case 4 : a+=((ub4)k[3]<<24);
    case 3 : a+=((ub4)k[2]<<16);
    case 2 : a+=((ub4)k[1]<<8);
    case 1 : a+=k[0];
    /* case 0: nothing left to add */
}
mix(a,b,c);
/*----- report the result */
return c;

```

FÖRBERÄKNAD FUNKTION

BERÄKNA $f(x)$ FÖR ALLA x OCH LAGRA SOM EN ARRAY $f[x]$.

VARJE ANROP AV f GÅR SEDAN I KONSTANT TID!

EXEMPEL: FUNKTION $u2l(c)$ SOM ÖVERSÄTTER FRÅN STORA BOKSTÄVER TILL SMÅ BOKSTÄVER.

INITIERING:

```
unsigned char u2l[256], *s;  
unsigned char *ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZÄÄÖ";  
int i;  
for (i=0; i<256; i++) u2l[i] = i;  
for (s=ALPHABET; *s; s++)  
    u2l[*s] = *s + 'a' - 'A';
```

ANVÄNDNING AV FUNKTIONEN:

```
unsigned char *s;  
for (s = word; *s; s++)  
    *s = u2l[*s];
```

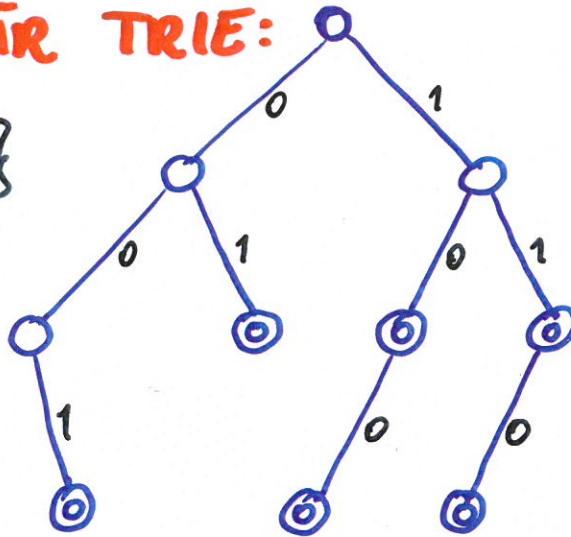
DETTA MÅSTE VARA EN unsigned char, FÖR EN VANLIG char KAN HA NEGATIVA VÄRDEN

TRIE

EN TRIE ÄR EN IMPLEMENTATION AV EN MÄNGD STRÄNGAR SOM ETT TRÄD.

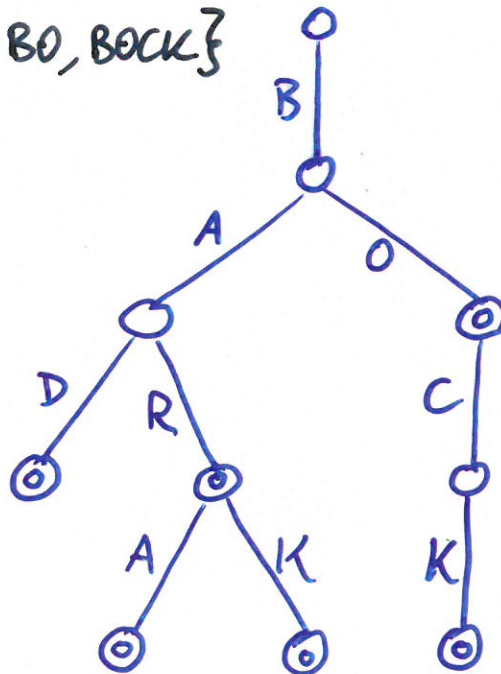
EXEMPEL PÅ BINÄR TRIE:

{001, 01, 10, 100, 11, 110}



EXEMPEL PÅ BOKSTAVSTRIE (AUTOMAT):

{BAD, BAR, BARA, BARK, BO, BOCK}



SVÄRT ATT LAGRA EFFEKTIVT!

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 | 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**.

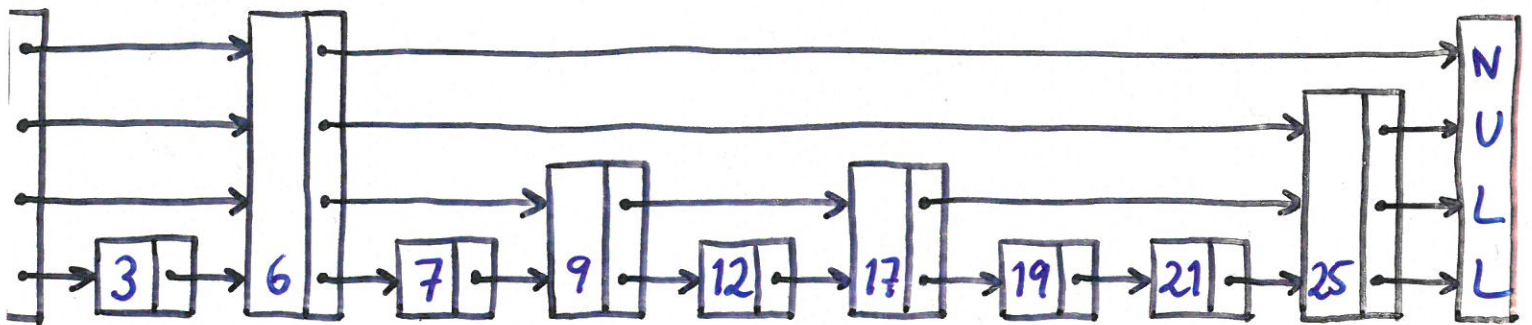
SÖKNINGSALGORITM(w) =

```
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, 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, 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 BERÖENDE 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 down to 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 FÖR-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:

$$\frac{\sum_{k=1}^{\infty} \text{ANTAL ELEMENT PÅ NIVÅ } k}{n} = \frac{\sum_{k=1}^{\infty} n \cdot p^{k-1}}{n} = 1 + p + p^2 + \dots = \frac{1}{1-p}$$

NIVÅ MED BARA $\frac{1}{p}$ ELEMENT:

$$n \cdot p^{k-1} = \frac{1}{p} \Leftrightarrow n = \left(\frac{1}{p}\right)^k \Leftrightarrow k = \log_{\frac{1}{p}} n \quad \text{KALLA DETTA } L(n)$$

FÖRVÄNTAT ANTAL JÄMFÖRELSE RÖRELSER VID SÖKNING:

VI MÄTER SÖKSTIGENS LÄNGD BAKIFRÅN: (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: $C(0) = 0$

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

SHT ATT VI ÄR KVAR PÅ SAMMA NIVÅ

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 $L(n)$.

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

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/adk14/skiplist`

DATASTRUKTURSKOMPLEXITET

	TID FÖR UPPSLAGNING		TID FÖR INSÄTTNING ELLER BORTTAGNING	
	VÄRSTA FALLET	MEDEL	VÄRSTA FALLET	MEDEL
SORTERAD ARRAY				$O(n)$
BINÄRT SÖKTRÄD				$O(\log n)$
HEAP (BARA STÖRSTA, ALTERNATIVT MINSTA, HITTAS ENKELT)				$O(\log n)$
HASHTABELL (NÄSTA/FÖREGÅENDE ELEMENT KAN INTE HITTAS ENKELT)				$O(1)$
RÖDSVART TRÄD				$O(\log n)$