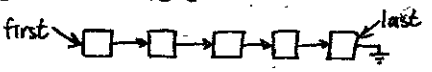
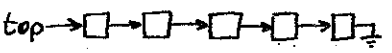
Översikt över kända datastrukturer

Linjära strukturer:

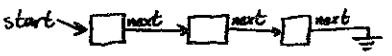
Kö:



Stack:



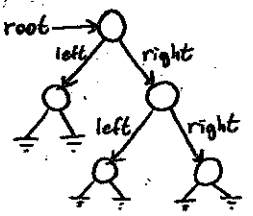
Lista:



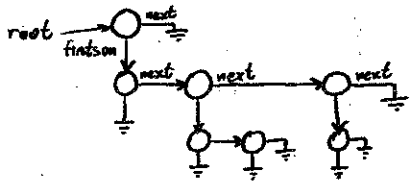
Array/vektor:

Träd och grafer:

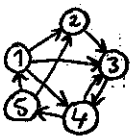
Binärt träd:



Allmänt träd:



Riktad graf:



Representerad som grannmatris

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

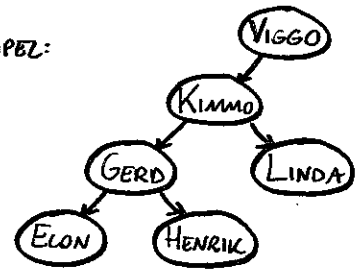
Representerad som kantmatris:

	1	2	3	4	5
1	-1	-1	-1	-1	1
2	1	-1			1
3		1	1	-1	1
4			1	-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 deldträd är ett sökträd).

Balanserade träd

Problem med binära sökträd:

Obalanserade träd tar tid att söka i.

Dålig lösning:

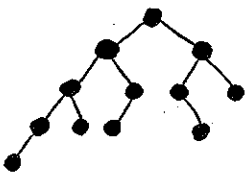
Balansera trädet, tar tid $O(n)$.

Bättre lösning:

Håll trädet balanserat efter varje insättning och borttagning.

Rödsvarta träd realiserar detta genom att vidmakthålla följande tre egenskaper:

1. Varje trädnod är antingen röd eller svart.
2. En röd nod kan inte ha en röd son.
3. I varje stig från en nod till vilket löv som helst under noden finns det lika många svarta noder.



Rödsvarta träd

Sökning

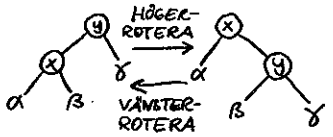
Använd vanlig trädsökningsalgoritm som tar tid $O(\text{träddjupet})$.

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 rödfärgat 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 $O(\log n)$

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 användningsområden:

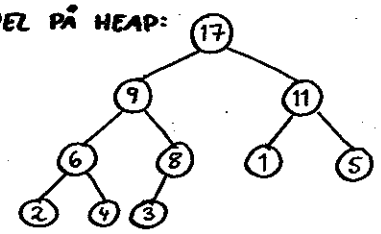
- Jobbhantering på fleranvändardatorer – jobbet med högst prioritet ska köras först
- Simulering av händelser – varje händelse ska inträffa vid en viss tidpunkt; händelserna ska bearbetas i tidsordning.
- Sortering – låt sorteringsnycklarna ange prioriteten
  1. Stoppa in alla poster som ska sorteras i prioritetskön.
  2. Plocka ut en post i taget. Posterna kommer i omvänd sorteringsordning, dvs största först och minsta sist.

Heapar – snabb representation av prioritetsköer

Komplett binärträd – alla nivåer i trädet är fyllda utom den sista. Den sista ska fyllas från vänster.
Heapordning: större element ovanpå mindre, dvs varje nods barn är mindre än fadern (eller lika stora)

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 $A[1]$
- sönerna till noden $A[i]$ finns i $A[2i]$ och $A[2i+1]$

Exempel: Heapen ovan lagras som

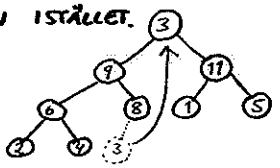
A:

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

Realisering av operationen Remove i en heap

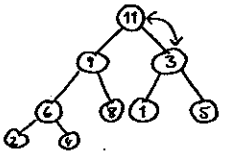
Elementet som ska tas bort ligger i roten (A[1])

- Spara undan elementet som ligger i roten.
- Ta bort det sista elementet i heapen och lägg det i roten istället.

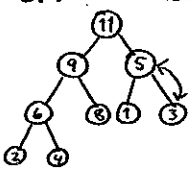


Downheap:

- Är roteelementet 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 punkt.



- Nu är heapen ordnad igen. (tog tid $O(\log n)$)

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: Function u2l(c) som översätter från stora bokstäver till små bokstäver.

Initiering:

```
unsigned char u2l[256], *s;
unsigned char *ALPHABET = "ABCDEFGHIJKLMNOPQ
                           RSTUVWXYZÄÅÖ";
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];
```

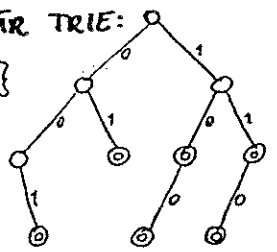
DETTA MÅSTE VARA EN unsigned char, FÖR EN VANLIG char KAN HA NEGATIVA VÄRDEN

Trie

En trie är en implementation av en mängd strängar som ett träd.

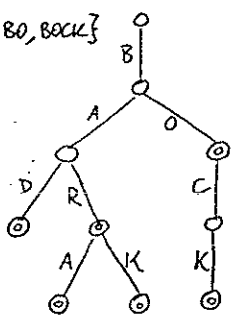
Exempel på binär trie:

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



Exempel på bokstavstrie (automat):

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



Svårt att lagra effektivt!

Hash functions for hash table lookup

Bob Jenkins

A hash function for hash table lookup should be fast, and it should cause as few collisions as possible. If you know the keys you will be hashing before you choose the hash function, it is possible to get zero collisions -- this is called perfect hashing. Otherwise, the best you can do is to map an equal number of keys to each possible hash value and make sure that similar keys are not unusually likely to map to the same value.

The standard reference for this is Knuth's "The Art of Computer Programming", volume 3 "Sorting and Searching", 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 the per-character mixing: it only rotates bits, it doesn't really mix them. Every input bit affects only 1 bit of hash until the final %. If two input bits land on the same hash bit, they cancel each other out. Also, % can be extremely slow (230 times slower than addition on a Sparc).

I offer you a new hash function for hash table lookup that is faster and more thorough than the one you are using now.

Over the past two years I've built a general hash function for hash table lookup. Most of the two dozen old hashes I've replaced have had owners who wouldn't accept a new hash unless it was a plug-in replacement for their old hash, and was demonstrably better than the old hash.

These old hashes defined my requirements:

- 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 things.
- 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);
```

In the new hash, mix() takes 3n of the 6n+35 instructions needed to hash n bytes. Blocks of text are

```
/* /info/adk/xx/hash.c */
/* Bob Jenkins http://ourworld.compuserve.com/homepages/bob_jenkins */
/* From Dr. Dobb's Journal, September 1997 */
typedef  unsigned long  int  ub4;   /* unsigned 4-byte quantities */
typedef  unsigned       char ub1;   /* unsigned 1-byte quantities */

#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 between 1/3 and
  2/3 of the time.
  mix() takes 36 machine instructions, but only 18 cycles on a superscalar
  machine (like a Pentium or a Sparc).  No faster mixer seems to work,
  that's the result of my brute-force search.  There were about 2^^68
  hashes to choose from.  I only tested about a billion of those.
*/
#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); \
}

/*
--------------------------------------------------------------------
hash() -- hash a variable-length key into a 32-bit value
  k       : the key (the unaligned variable-length array of bytes)
  len     : the length of the key, counting by bytes
  initval : can be any 4-byte value
Returns a 32-bit value.  Every bit of the key affects every bit of
the return value.
About 6*len+35 instructions.

The best hash table sizes are powers of 2.  There is no need to do
mod a prime (mod is sooo slow!).  If you need less than 32 bits,
use a bitmask.  For example, if you need only 10 bits, do
  h = (h & hashmask(10));
In which case, the hash table should have hashsize(10) elements.

By Bob Jenkins, 1996.  bob_jenkins@compuserve.com.  You may use this
code any way you wish, private, educational, or commercial.  It's free.

Use for hash table lookup, or anything where one collision in 2^^32 is
acceptable.  Do NOT use for cryptographic purposes.
--------------------------------------------------------------------
*/

ub4 hash( k, length, initval)
register ub1 *k;        /* the key */
register ub4  length;   /* the length of the key */
register ub4  initval;  /* the previous hash, or an arbitrary value */
{
   register ub4 a,b,c,len;

   /* Set up the internal state */
   len = length;
   a = b = 0x9e3779b9;  /* the golden ratio; an arbitrary value */
   c = initval;         /* the previous hash value */

   /*---------------------------------------- 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)              /* all the case statements fall through */
   {
   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;
}
```

combined with the internal state (a,b,c) by addition. This combining step is the rest of the hash function, consuming the remaining 3n instructions. The only postprocessing is to choose c out of (a,b,c) to be the result.

Three tricks promote speed:

1. Mixing is done on three 4-byte registers rather than on a 1-byte quantity.
2. Combining is done on 12-byte blocks, reducing the loop overhead.
3. The final switch statement combines a variable-length block with the registers a,b,c without a loop.

The golden ratio really is an arbitrary value. Its purpose is to avoid mapping all zeros to all zeros.

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.

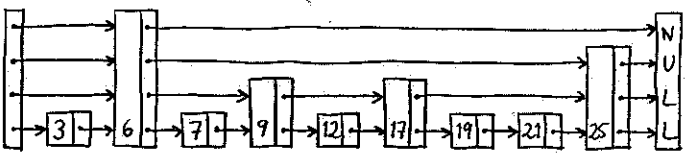
Förberedelse inför sökningar:
Läs in A från fil. Öppna filerna I och L.

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?

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

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/adk[illegible]/skiplist