Machine Translated by Google

Chapter 3

Vectors

Machine Translated by Google

3 vectors

A vector (field, array) manages a fixed number of elements of a uniform type.

- Accessing an element via an integer index (the position in Vector)
- Access effort for all elements is constant

Methods

- Hashing: Datenorganisationsform
- Sorting: Data reorganization methods

Machine Translated by Google

3.1 Dictionary

A *dictionary* is a data structure in which the individual elements each consist of a *key* and an *information part* (info).

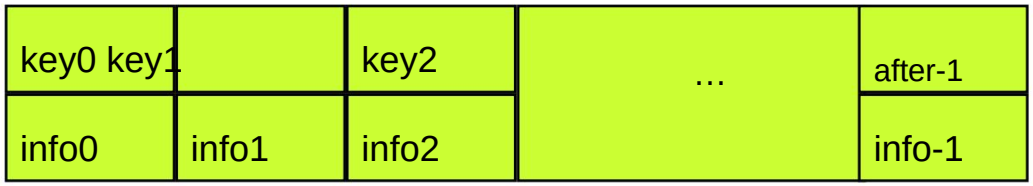
Supports only the simple operations of insert, delete and search provided.

examples are

Dictionaries, name lists, inventory lists, etc.

Vectors are well suited to creating dictionaries

Structure



Machine Translated by Google

Examples

dictionary

(detail)

Key	Information
computable	predictable
computation	calculation
compute	(calculate
computer	calculator

Inventory list

Key	Information
1	CPU
4	Screen
17	Keyboard
25	Maus

Machine Translated by Google

The operation

Construct

Creating an empty dictionary

IsEmpty

Query for empty dictionary

Insert

Inserting an element

Delete

Delete an element

LookUp

Query an element via its key and deliver it
Information

Machine Translated by Google

Klasse Dictionary


```
class Dictionary {
    private:
        node { KeyType Key; InfoType Info; }; node *Dict;

        int NumberElements;
    public:
        Dictionary(int max) {
            Dict = new node[max];
            NumberElements = 0;
        }
        ~Dictionary() { delete[] Dict; } void
        Insert(KeyType k, InfoType I); void Delete(KeyType
        k);
        InfoType LookUp(KeyType k);
};
```

Machine Translated by Google

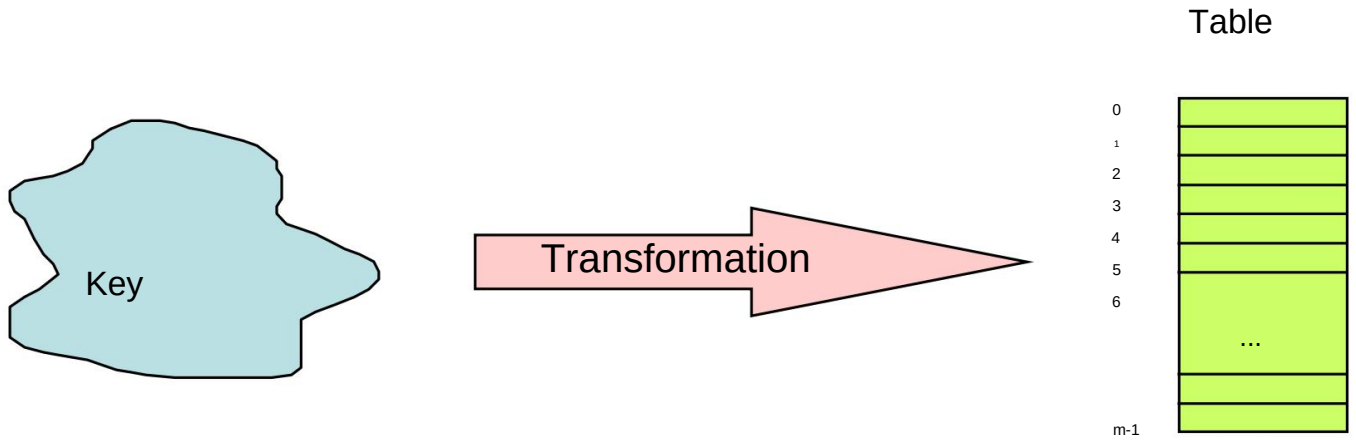
3.2 Hashing

Hashing is a method of directly assigning elements in a table address by replacing their key values with arithmetic

Transformations can be translated directly into table addresses (indexes).

In other words, the index (the position in the table) of a Elements is calculated from the key value of the element itself.

Key ÿ Address (index)



Machine Translated by Google

Hashing, general

Approach

Set K of n keys {k0 , k1 , …, kn-1 }
Hash table T of size m

Boundary condition: n >> m

(Number of possible key values much larger than places in the table)

Transformation

Hash function h

h: K ÿ {0, 1, … m-1}

For each j, the key kj should be stored at location h(kj) in the table. The value h(K) is called the hash value of K.

Choice of hash function (actually) arbitrary!

Machine Translated by Google

Hash function

Desired properties

- Easy and quick to calculate.
- Elements are evenly distributed throughout the table.
- All positions in the table have equal probability calculated.
- Examples
  - Modulo education, key mod m
    - You should make sure that m is a prime number, otherwise it can happen Key transformations (strings) lead to cluster formations (accumulations) through common divisors.
  - Bitstring-Interpretation
    - Parts of the binary representation of the key value are used as a hash value used.
  - Transformation tables

Machine Translated by Google

Example hashing

Inventory list

List size 7

Hash function h(k) = k mod 7

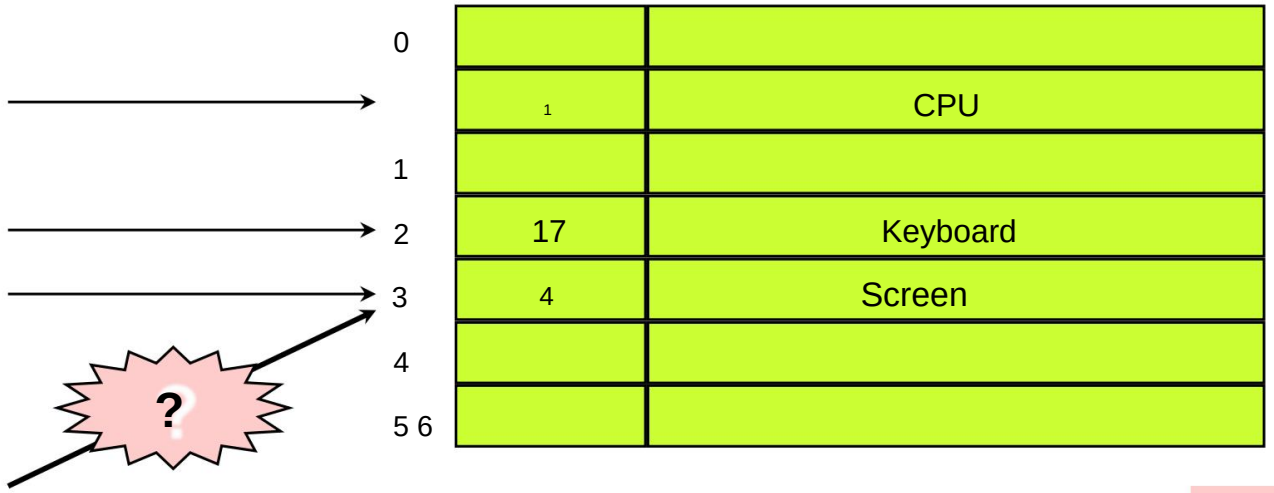
Records

Key	Information
1	CPU
4	Screen
17 25	Keyboard
	Maus

1 against 7 = 1

17 against 7 = 3

4 against 7 = 4



25 mod 7 = **4** ÿ Position in the table already occupied

collision

Machine Translated by Google

collision

A *collision* occurs when two or more key values are mapped *(hashed)* to the same table position .

This means that for 2 keys ki , kj , with ki ÿ kj ,
h(ki) = h(kj).

However, this situation is to be expected since there are many more possible keys than table positions (n >> m as a boundary condition).

The hash function h is generally not injective, i.e. it does not follow from h(x) = h(y). necessarily x=y.

Machine Translated by Google

Collision treatment

A collision triggers necessary *collision handling* .

Ie, the colliding key value must have a
Alternative location can be found.

Collision handling measures are usually quite complex and affect the efficiency of hashing processes.

the collision path is defined by the fallback locations of all elements that have been "hashed" to the same location

We now consider 2 simple collision treatments

Separate Chaining und *Double Hashing*

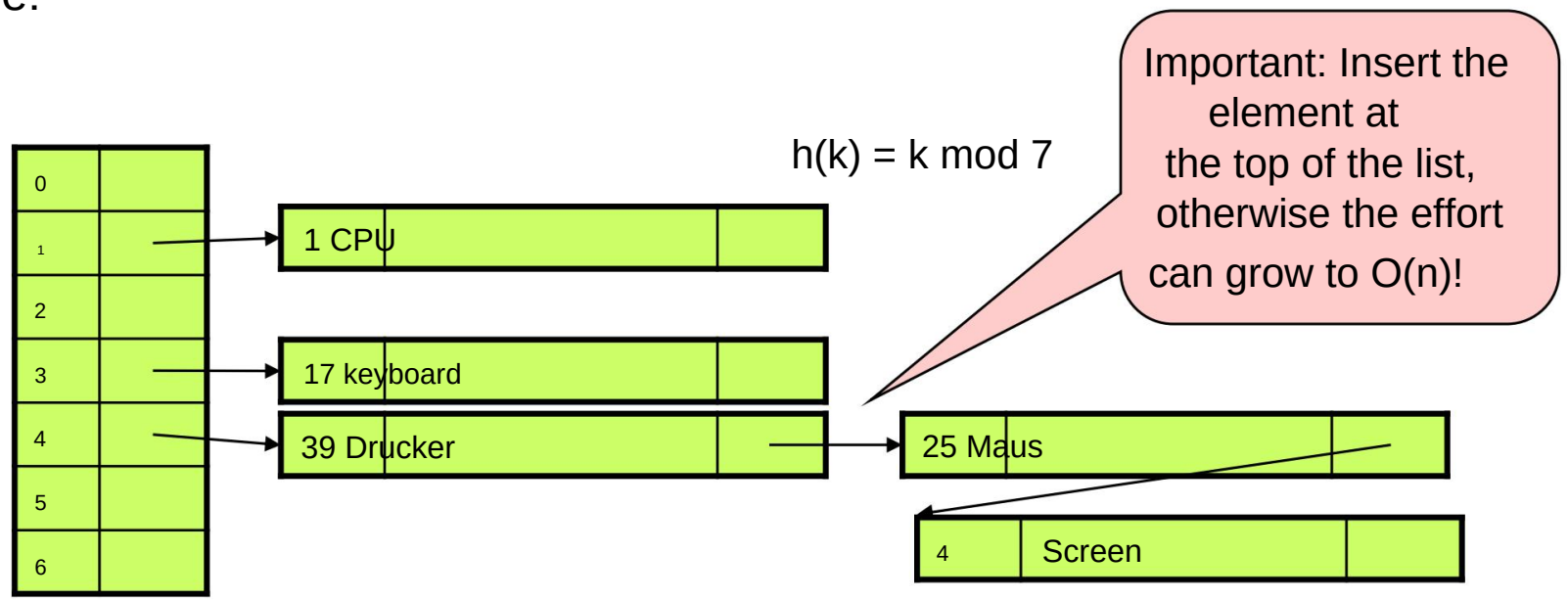
Machine Translated by Google

3.2.1 Separate Chaining

With separate (simple) chaining, collisions are handled accomplished with linear lists.

Colliding key values are stored in a linear overflow chain (list) starting from the original hashed table space.

Example:



The entry of the element (39, printer) leads to an extension of the overflow chain.

Machine Translated by Google

3.2.2 Double Hashing

With *double hashing,* in the event of a collision, a second one of h independent hash function used to determine an alternative position. Overflow to position a0 = h(k)

Determination of an alternative position with collision function g : K ÿ {1, … m-1}, e.g. ai+1 = (ai + g(k)) mod m

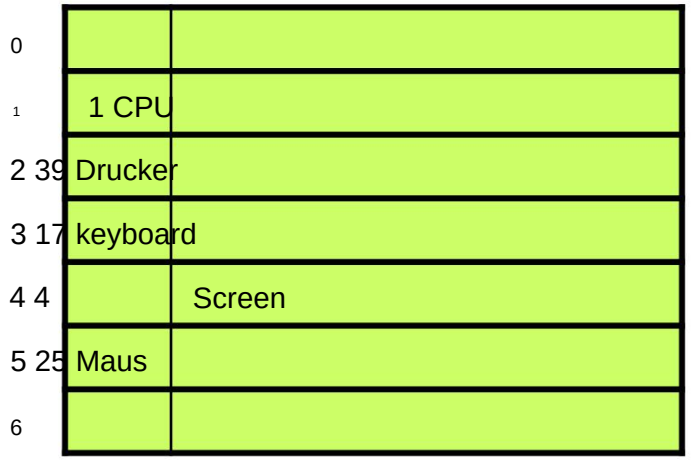
Example

K . Collision

h(k) = k mod 7
g(k) = last digit of k times 3

a0 = 25 mod 7 = 4 K
a1 = (4 + (5*3)) mod 7 = 5

a0 = 39 mod 7 = 4 K
a1 = (4 + (9*3)) mod 7 = 3 K
a2 = (3 + (9*3)) mod 7 = 2



Machine Translated by Google

Linear Probing

Special case of double hashing

Collision handling: The next free space is used

When you reach the end of the table, you look at the beginning of the table further

Hashfuktionen: a0 = h(k), ai+1 = (ai + g(k)) mod m

g(k) = 1

Example

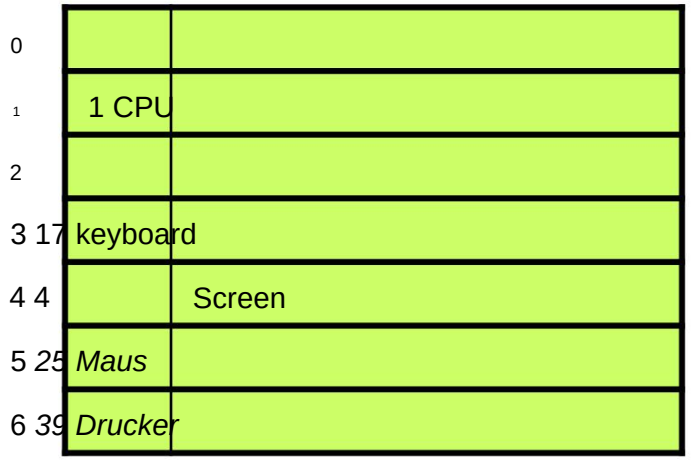
25 against 7 = 4 K

(4+1) against 7 = 5

39 against 7 = 4 K

(4+1) against 7 = 5 K

(5+1) against 7 = 6



Machine Translated by Google

Searching and deleting in hash tables

Seek

As long as a target address is occupied by an element and the searched key value has not yet been found, the search must continue using the collision path.

- Separate Chaining: Tracking the linear list
- Double Hashing: Finding all possible locations of the collision function.

Delete

Separate Chaining: Removing the list item

Double hashing: Positions where an element was deleted may need to be marked with a special value (“free again”) so that the corresponding collision path for the remaining elements is not interrupted.

Machine Translated by Google

Characteristics

Generally for hashing processes

+ Effort when accessing an element is constant in the best case, O(1), simple evaluation of the hash function.

- Collision handling is complex, a full hash table leads to many collisions, so (rule of thumb) never fill more than 70%.

- No access in sort order possible.

Separate chaining +

dynamic data structure, any number of elements.

- Storage space intensive (additional pointer variable).

Double hashing +

optimal storage space utilization.

- Static data structure, table size specified.

- Collision handling complex, “clear again” marking when deleting.

Machine Translated by Google

Analyse Hashing

- Data management
  - Insert and delete supported
- Data volume
  - limited
    - depending on the size of the existing hash table
- Models
  - Main memory oriented
  - Support for simple operations, no range queries, no sort order

Machine Translated by Google

Analyse Hashing (2)

Storage space	O(1)
Constructor	O(1)
access	O(1)
Insert	O(1)
Delete	O(1)

valid only for pure Hash process without collision handling

Please note: The effort involved in hashing depends heavily on the Collision method, often goes towards O(n) in the case of collisions

Machine Translated by Google

3.3 Dynamic hashing methods

Dynamic hashing procedures try in the event of collisions expand the original hash table and create

overflow areas (e.g. linear lists)

Enlargement of the primary area (original table)

Requires hash function modification

Approach: Families of hash functions

h1 : K ÿ addr1

...

hn : K ÿ addrn

Where |addr1 | < |addr2 | < ··· < |addrn

| Goal: Switching from addri to addri+1 requires minimal reorganization of the hash table (i.e. minimizing restoring of data)

Machine Translated by Google

Hash functions scheme

Simple approach

|addr+1| = 2 * |addr |

h(x) is a hash function

Let hi (x) be the i “least significant bits” of h(x)

Therefore applies

hi+1(x) = hi (x) or

hi+1(x) = hi (x) + 2i

Example:

addr2 = 0 .. 3,

addr3 = 0 .. 7,

h(x) = x

4 address changes

Value h(x)		binary	h2 (x)	h3 (x)
7	7	00111 11=3		111=7
8	8	01000 00=0	000=0	
9	9	01001 01=1		001=1
10	10	01010 10=2	010=2	
13	13	01101 01=1		101=5
14	14	01110 10=2	110=6	
23	23	10111 11=3		111=7
26	26	11010 10=2	010=2	

Machine Translated by Google

Dynamic hashing methods

„Two-Disk-Access“ Prinzip

- Finding a key with a maximum of 2 disk accesses

Hash process for external storage media

Linear Hashing

- Directoryless hashing method
- Primary areas, overflow areas

Extendible Hashing

- Directory with binary expansion
- Primary areas

Bounded Index Size Extendible Hashing

- Limited size directory
- Binary expansion of blocks

Machine Translated by Google

3.3.1 Linear Hashing

W. Litwin 1980

Organization of elements in blocks (buckets)

- Analogous to B-trees
- Block size b, ie up to b can be in one address in the table
  - Elements are saved (=block)

Idea

- When a block overflows (= b+1st element of a block
  - inserted) (splitting), the primary and overflow areas are expanded by one block each
- Primary scope by adding a block to the end of the hash table
- Overflow area through linear list for individual blocks
- Round number d stores the number of splits per block, there are
  - simultaneously blocks that were split d times and blocks that were split d+1 times

Machine Translated by Google

Splitting and searching

Splitting is carried out in rounds

A round ends when all the original N blocks (for round R) have been split

Blocks 0 to NextToSplit-1 are already split

Next block to split is defined by index NextToSplit

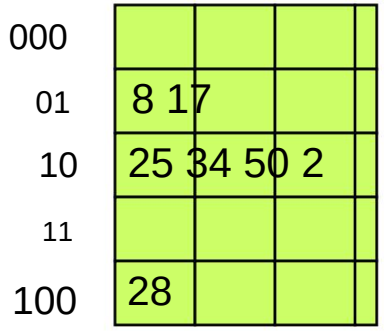
Current lap number is defined by d

Search

Search for value x

a = hd(x); if(a < NextToSplit) a = hd+1(x);

d=2, NextToSplit=1, b=3



2=000010
8=001000
17=010001
25=011001
28=011100
34=100010
50=110010

Machine Translated by Google

Insert

Search block (hd or hd+1)

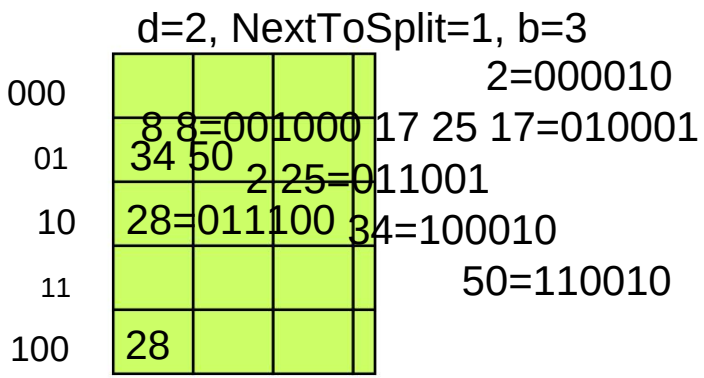
If block full

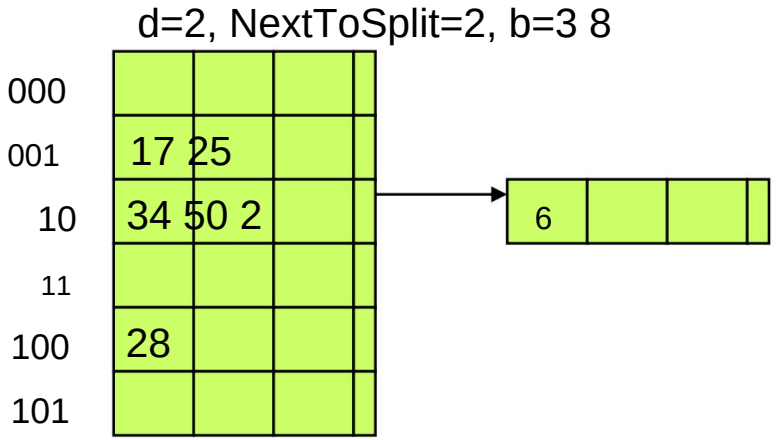
Add element into an overflow block

New element in the linear list

Split block NextToSplit and increase NextToSplit by 1

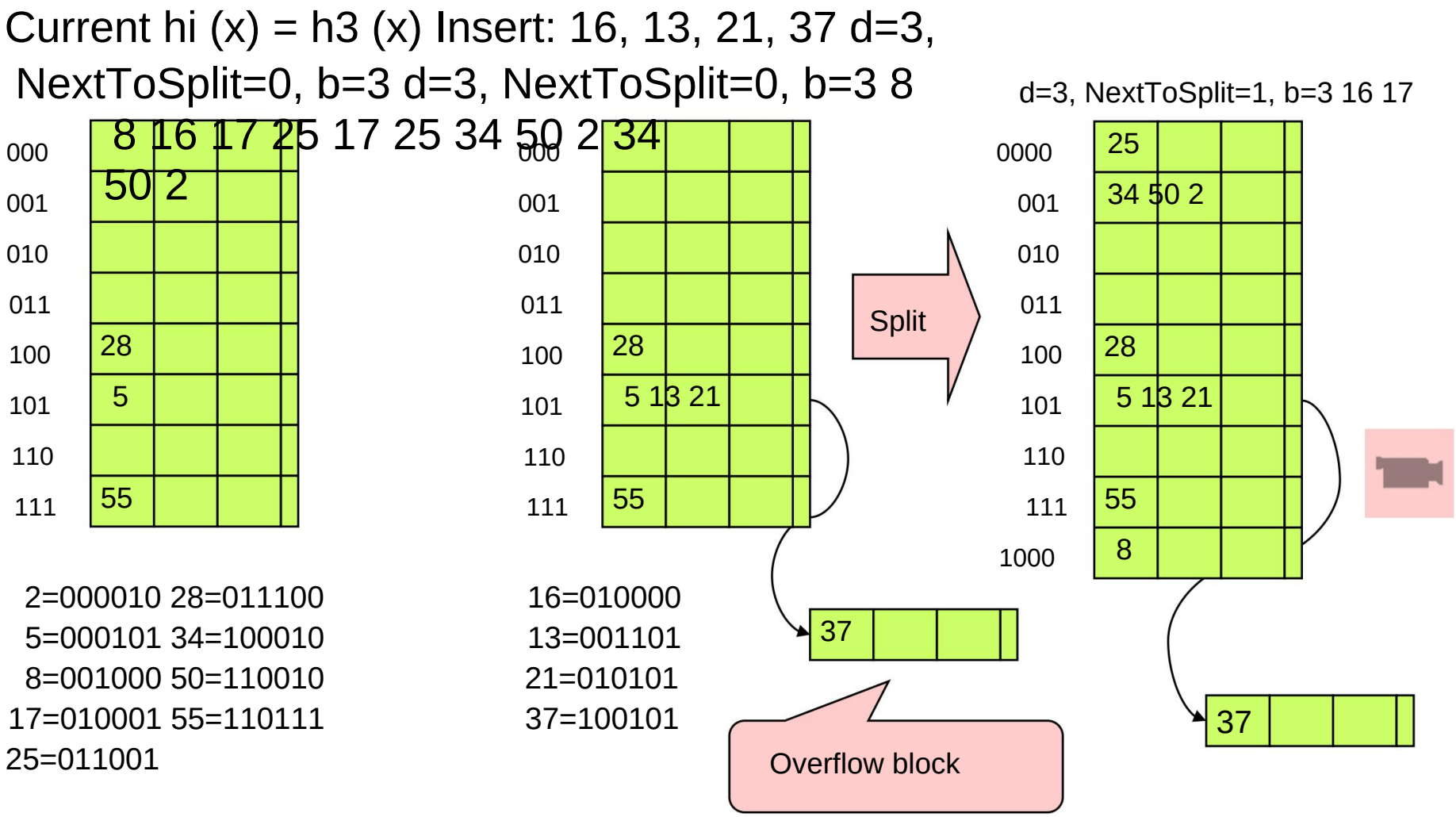
Example: Inserting 6 (=000110)





Machine Translated by Google

Example: linear hashing



Machine Translated by Google

Remarks

The round number d is increased once the splitting is done, ie

if(NextToSplit == 2d) { d++; NextToSplit=0; }

In practice, no linear address spaces but rather splitting them

primary area to multiple hash files per disk

Allocation system via administration blocks

Machine Translated by Google

3.3.2 Extendible Hashing

Hash process with index

R. Fagin, J. Nievergelt, N. Pippenger and HR Strong, 1979

Idea

If primary area becomes too small, enlarge by doubling

Primary area is managed via Index T

If a block overflows, split this block and manage the block by doubling the index

Index is much smaller than file, so doubling it is much cheaper

No overflow areas

Machine Translated by Google

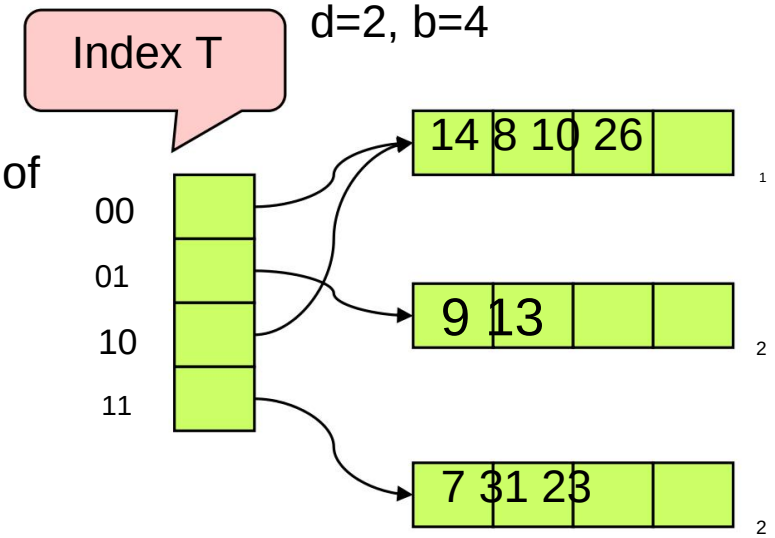
Structure and search

Characteristics

Each index entry references exactly
a data block

Each data block has ÿ b entries and is made up of
exactly 2k with k out of N0
Index entries referenced

The number of index entries that contain the
Referencing block is known locally in the block



Managed by a local depth t ÿ d
(Opposite global depth d for the entire hash table)

Number of referencing the block
Index entries corresponds to 2d-t

7=00111 14=01110
8=01000 23=10111
9=01001 26=11010
10=01010 31=11111
13=01101

Local depth

Search

Element x im Block T[hd (x)]

Machine Translated by Google

Insert

Two cases must be distinguished if a block overflows

t < d: multiple index entries reference this block t = d:

one index entry references this block

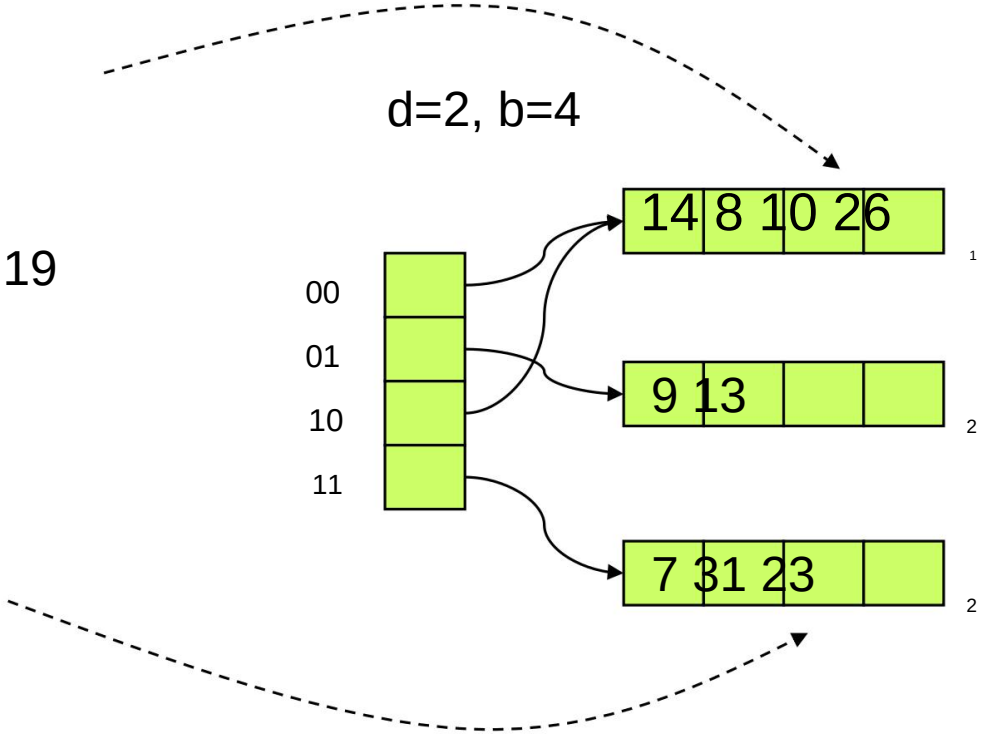
Fall 1: t < d

e.g. B. Insert 6
6=00110

Fall 2: t = d

e.g. B. Insert 15, 19
15=01111

19=10011



7=00111
8=01000
9=01001
10=01010
13=01101
14=01110
23=10111
26=11010
31=11111

Machine Translated by Google

Insert case 1: t < d

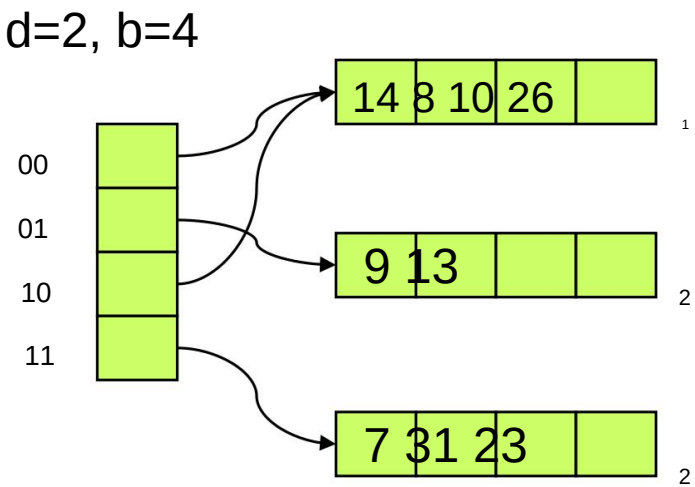
Try to do the split without index expansion

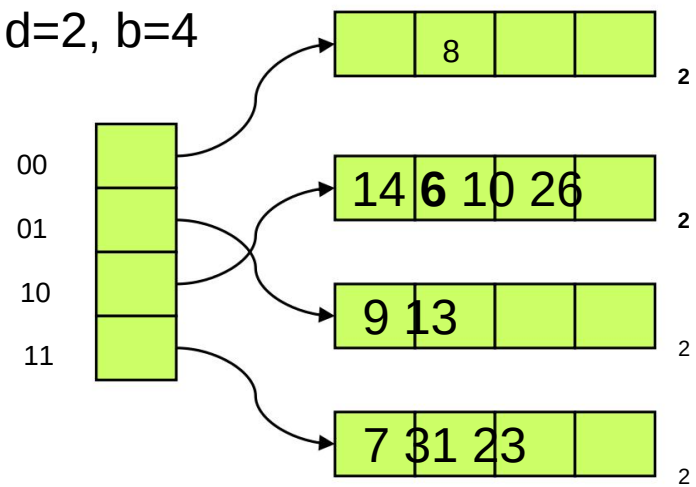
Request new data block

Splitting the data of the overflowing block according to ht+1 t = t + 1 for old and new data block

If split leads to overflow again, repeat

Example: Inserting 6 (=00110)





Machine Translated by Google

Insert case 2: t = d

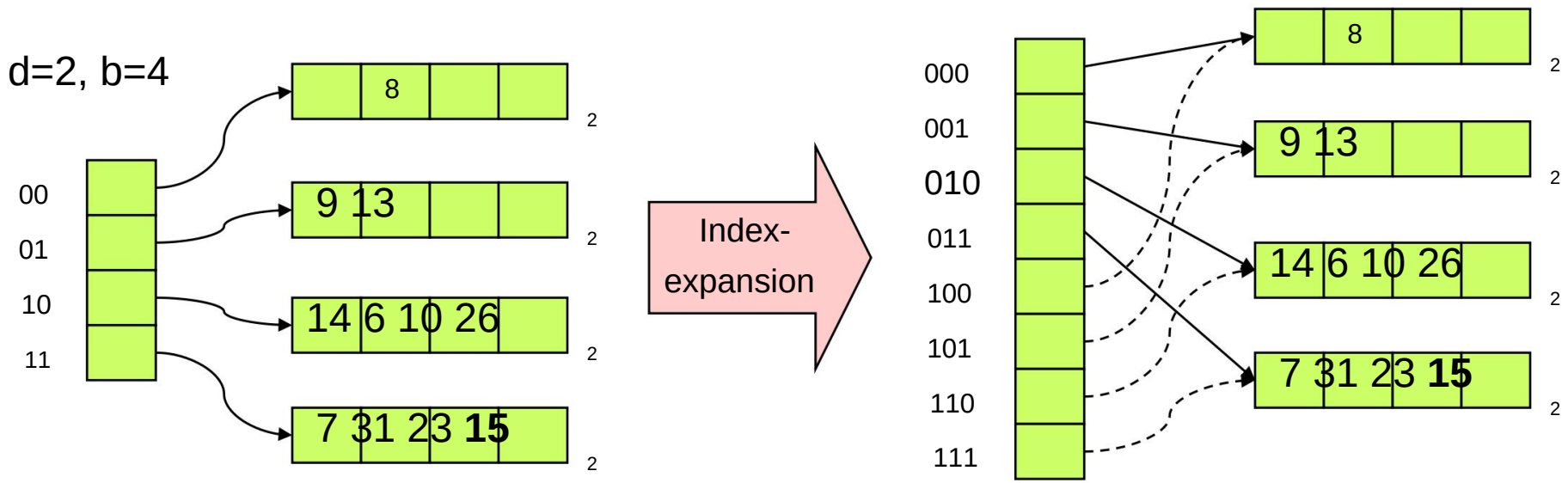
Requires index expansion (doubling the index)

Request new storage area for 2d additional references

For every T[x], for which x ÿ 2 d : Indexeintrag T[x] = T[x-2 d]
= d+1

Then return to case 1

Example: Inserting 15 and 19



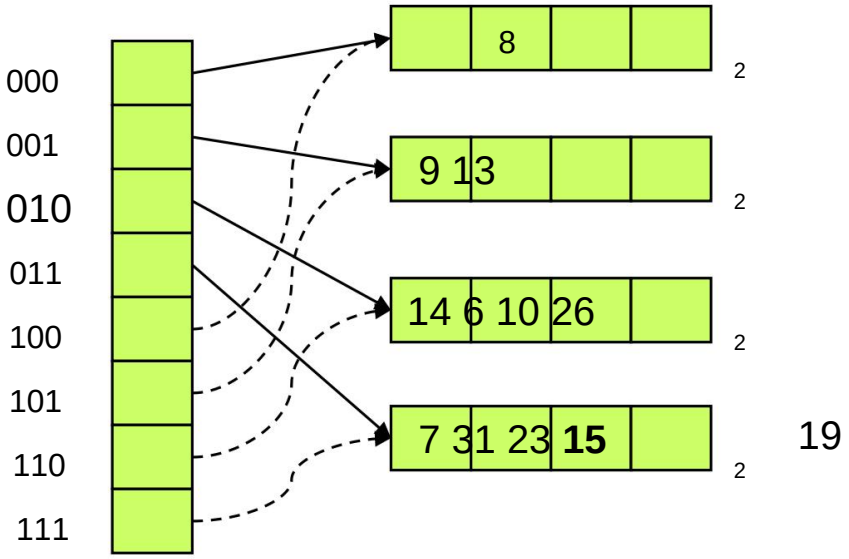
Machine Translated by Google

Insert case 2: t = d (2)

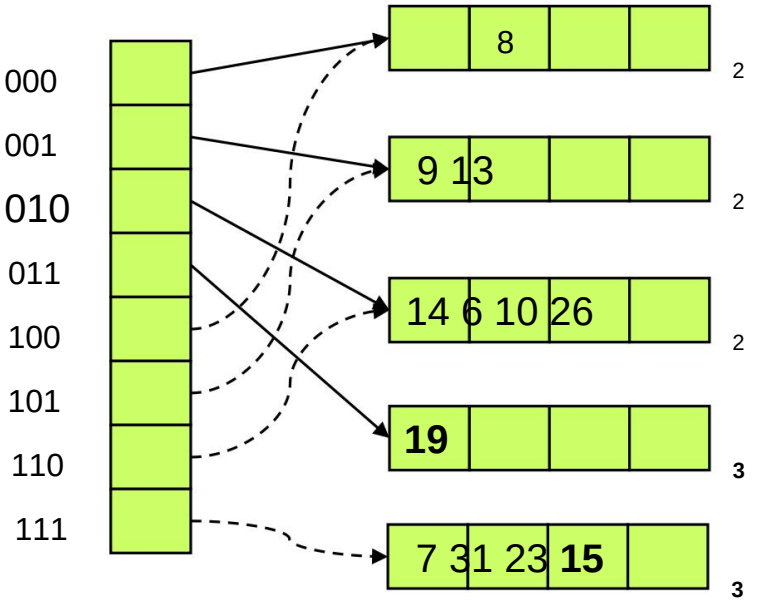
Now case 1

7=00111
15=01111
19=10011
23=10111
31=11111

d=3, b=4



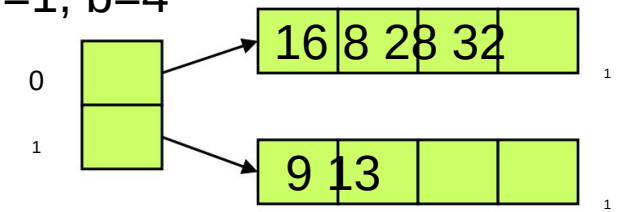
d=3, b=4



Machine Translated by Google

Index expansion can fail, multiple doubling

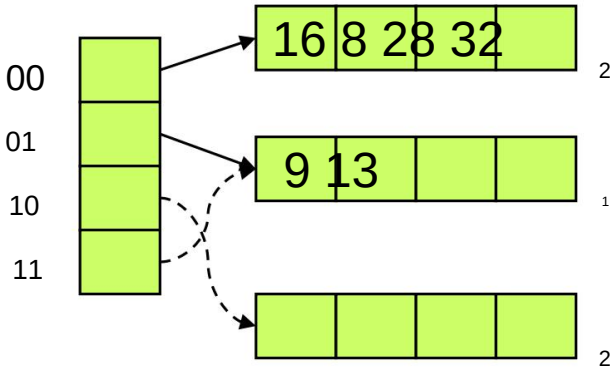
d=1, b=4



8=001000
9=001001
13=001101
16=010000
28=011100
32=100000
56=111000

Insert 56

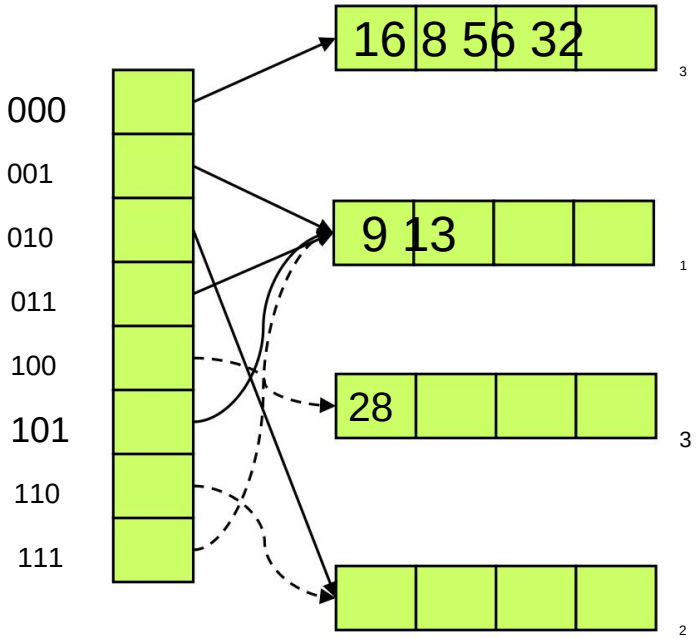
d=2, b=4



56

1. Indexexpansion

d=3, b=4



2. Indexexpansion

Machine Translated by Google

Comments

Deleting empty blocks is not easy

- Merging with their split buddies (“split images”)
- Analogous for index reduction

If index can be kept in main memory, only external access is necessary

In the worst case, exponential index growth

- Typical for clustered data (data is not evenly distributed)
- Therefore storage space requirement for index is not acceptable in the worst case
  - Index usually in main memory

Index blocking problem on disk

No guaranteed minimum occupancy

Machine Translated by Google

Analyse

Analysis complex

- It can be shown that to store a file with n records and a block size of b records, Extendible Hashing requires approximately 1.44*(n/b) blocks
- On average, the directory has n1+1/b/b for uniformly distributed data sets Entries

Big advantage if the directory fits in the main memory, only disk access to one data set is necessary

Machine Translated by Google

3.3.3 Bounded Index Size Extendible Hashing

approach

- Primary area (analogous to Extendible Hashing)
  Limited size index
- Binary expansion of data areas
  - (1 block, 2, 4, ... blocks)

Attempts to address the inability of extendible hashing to accommodate the index on disk with a special index structure

Machine Translated by Google

Structure

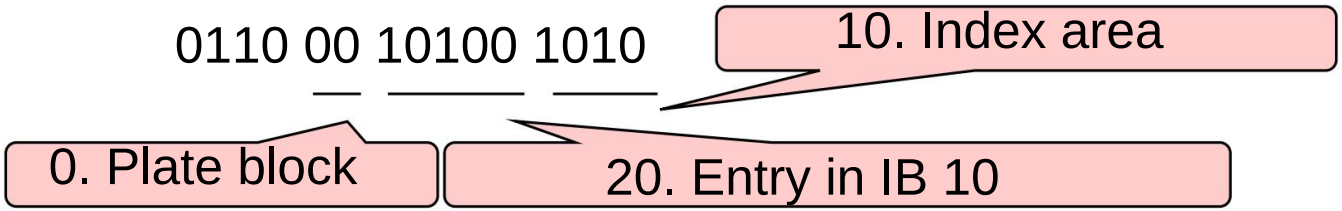
Index consists of x areas

A range contains y entries of the form <z, ptr> where

- ptr is an address on a disk block
- z gives the number of duplications of the corresponding data areas
- i.e. ptr is the address of a data area with 2z disk blocks
- x and y are powers of 2

Hash values are interpreted zoned, e.g

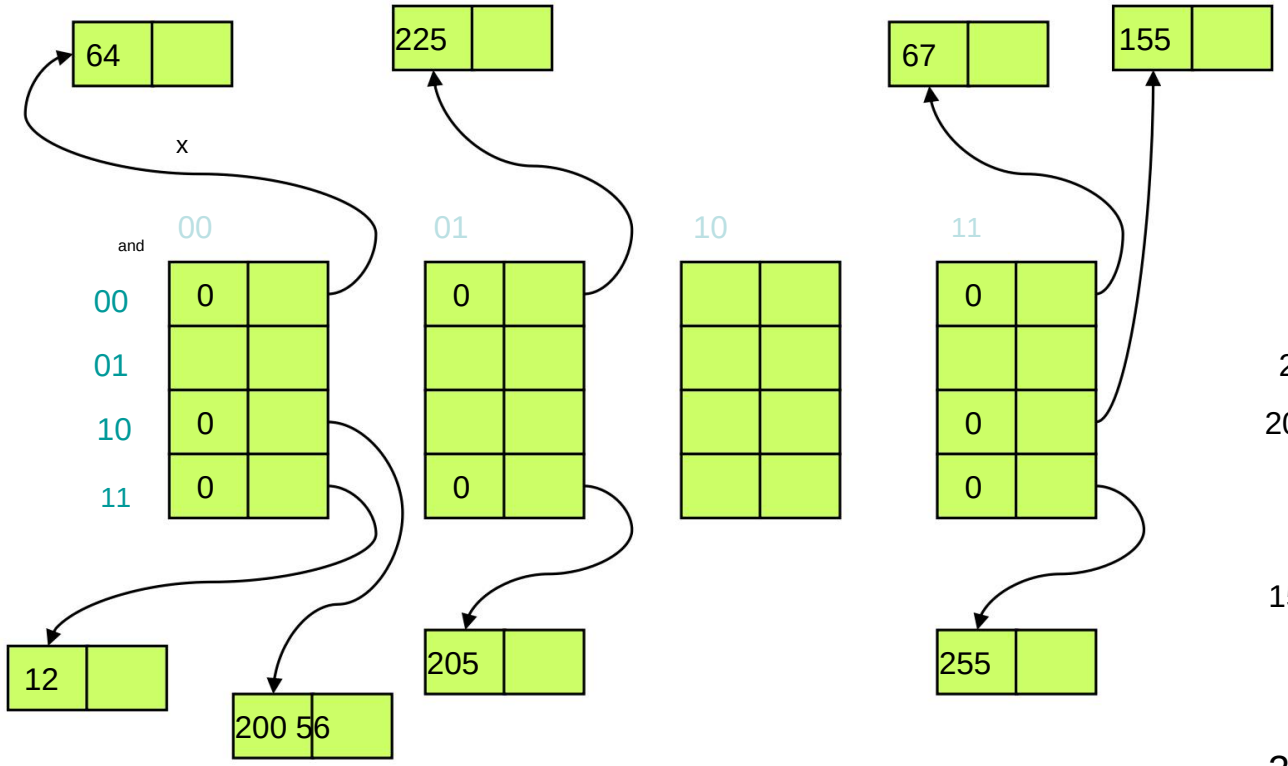
- 16 index areas (x=16), 32 entries per index area (y=32), 4 blocks in each data area
  - Hash signature for the 0th block referenced by the 20th entry in index range 10



Machine Translated by Google

Example BISEH

b=2 (block size), x=4 (index areas), y=4 (entries per IB)



255 = 1111 11 11

200 = 1100 10 00

56 = 0011 10 00

64 = 0100 00 00

155 = 1001 10 11

67 = 0100 00 11

12 = 0000 11 00

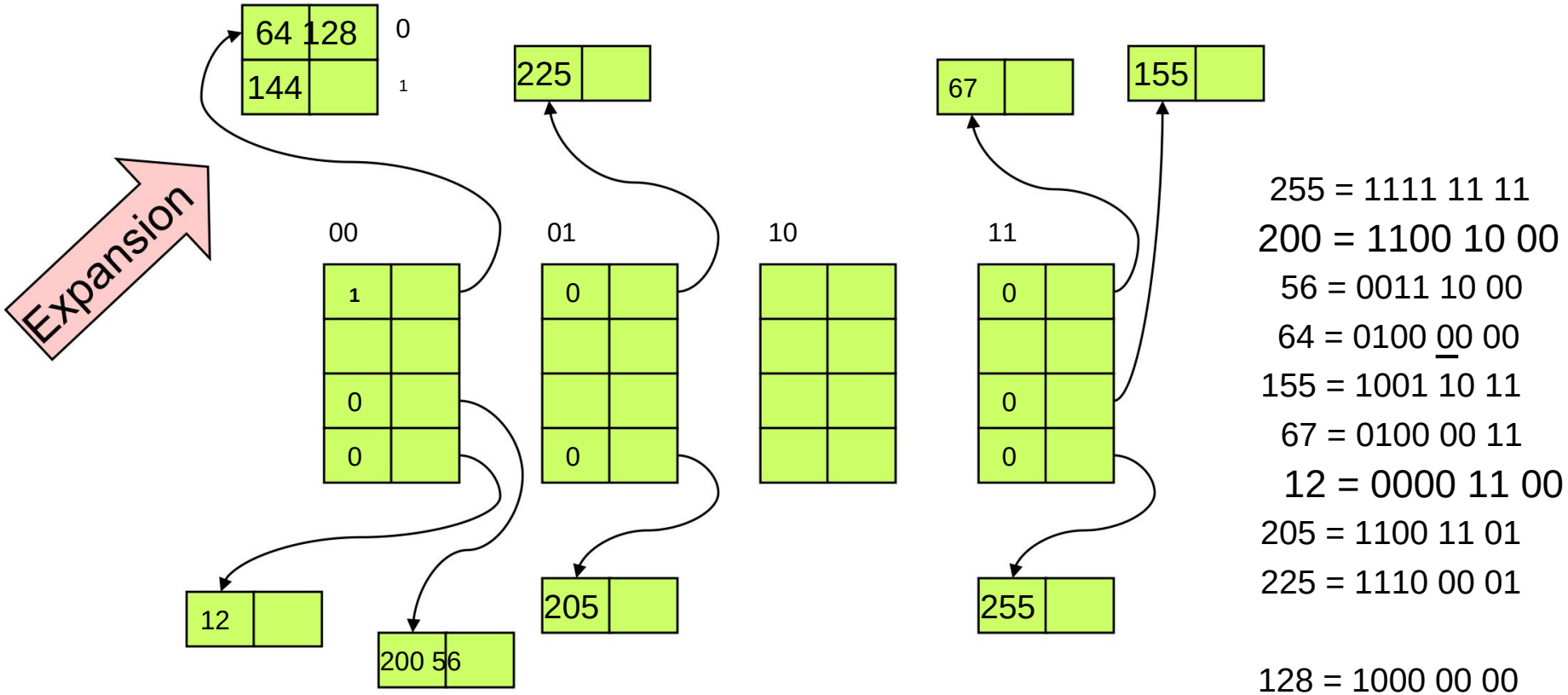
205 = 1100 11 01

225 = 1110 00 01

Machine Translated by Google

Example BISEH (2)

Insertion of 128 and 144, successful in first doubling attempt



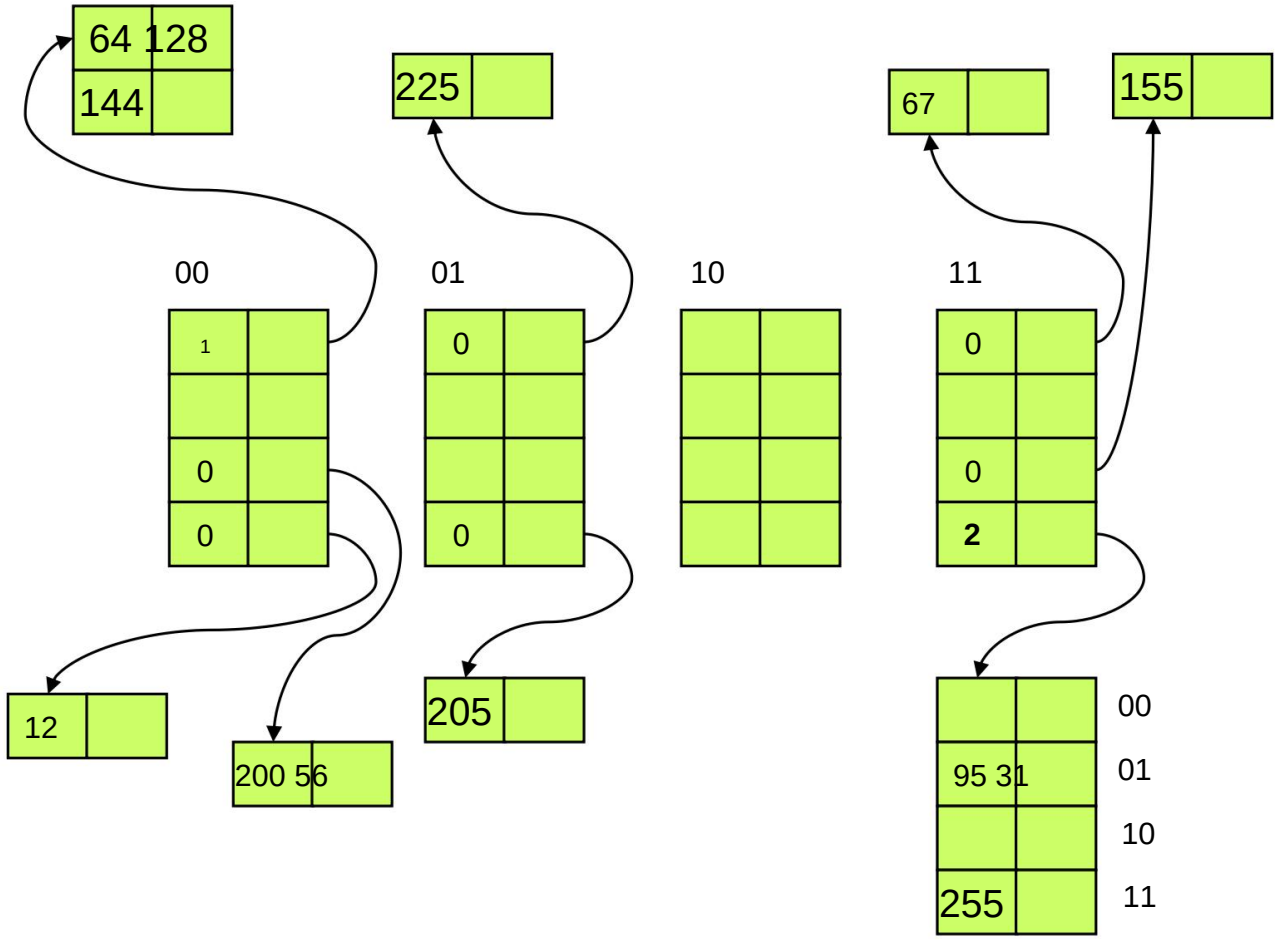
255 = 1111 11 11
200 = 1100 10 00
56 = 0011 10 00
64 = 0100 00 00
155 = 1001 10 11
67 = 0100 00 11
12 = 0000 11 00
205 = 1100 11 01
225 = 1110 00 01

128 = 1000 00 00
144 = 1001 00 00

Machine Translated by Google

Example BISEH (3)

Insertion of 95 and 31, only successful in the second doubling attempt



255 = 1111 11 11
200 = 1100 10 00 56
= 0011 10 00 64 =
0100 00 00 155 =
1001 10 11 67 =
0100 00 11 12 =
0000 11 00 205 =
1100 11 01 225 =
1110 00 01 128 =
1000 00 00 144 =
1001 00 00

95 = 0101 11 11
31 = 0001 11 11

Machine Translated by Google

General comparison of hash to Index procedure

Points to note:

Costs of periodic reorganization

Insertion and deletion frequency

Average Case versus Worst Case Aufwand

Expected query types

- Hashing is generally better for exact key queries
- Index structures (e.g. search trees) are preferable for range queries

Machine Translated by Google

Sorting elements (values, data records, etc.) is one of *the* most important and complex problems in IT practice.

There are hundreds of sorting algorithms in a wide variety of variants for a wide variety of applications.

- Main memory - external memory
- Sequential - Parallel
- Insitu - Exsitu
- Stable - Unstable
- ...

Sorting methods are among the most extensively analyzed and refined algorithms in computer science.

Machine Translated by Google

Criteria for selecting Sorting method

Main memory - external memory

The algorithm can only store data in main memory or files
Sort (files) or tapes on the external storage (disk, tape station)?

Insitu - Exsitu

Does the sorting process make do with the original data area (insitu) or does it need additional storage space (exsitu)?

Worst Case - Average Case

Is the (asymptotic) runtime behavior of the sorting process always the same or can it “degenerate” (become faster or slower) in special cases?

Stable - Unstable

If multiple records have the same key value, will the original order be retained after sorting?

Machine Translated by Google

Sorting method

Task A

vector of size n of the form V[0], V[1], ..., V[n-1] is to be sorted.

Lexicographic order on the elements.

After sorting, the following applies: V[0] ÿ V[1] ÿ … ÿ V[n-1].

Elementary (Simple) Procedures (Selection)

Selection Sort

Bubble Sort

Refined (“smarter”) procedures (selection)

Quicksort

Mergesort

Heapsort

Machine Translated by Google

Class Vector

```
class Vector {
    int *a, size; public:

    Vector(int max) { a = new int[max]; size = max; }
    ~Vector() { delete[] a; } void
    Selectionsort(); void
    Bubblesort(); void
    Quicksort(); void
    Mergesort(); int Length();
    private: void
quicksort(int,
    int); void mergesort(int, int, int*);
    void swap(int&, int&); };
```

Machine Translated by Google

3.4.1 Selection Sort

Selection sort or minimum search algorithm

Find the smallest element in the vector and swap it with the element in the first place. Repeat this process for all unsorted elements.

Swap

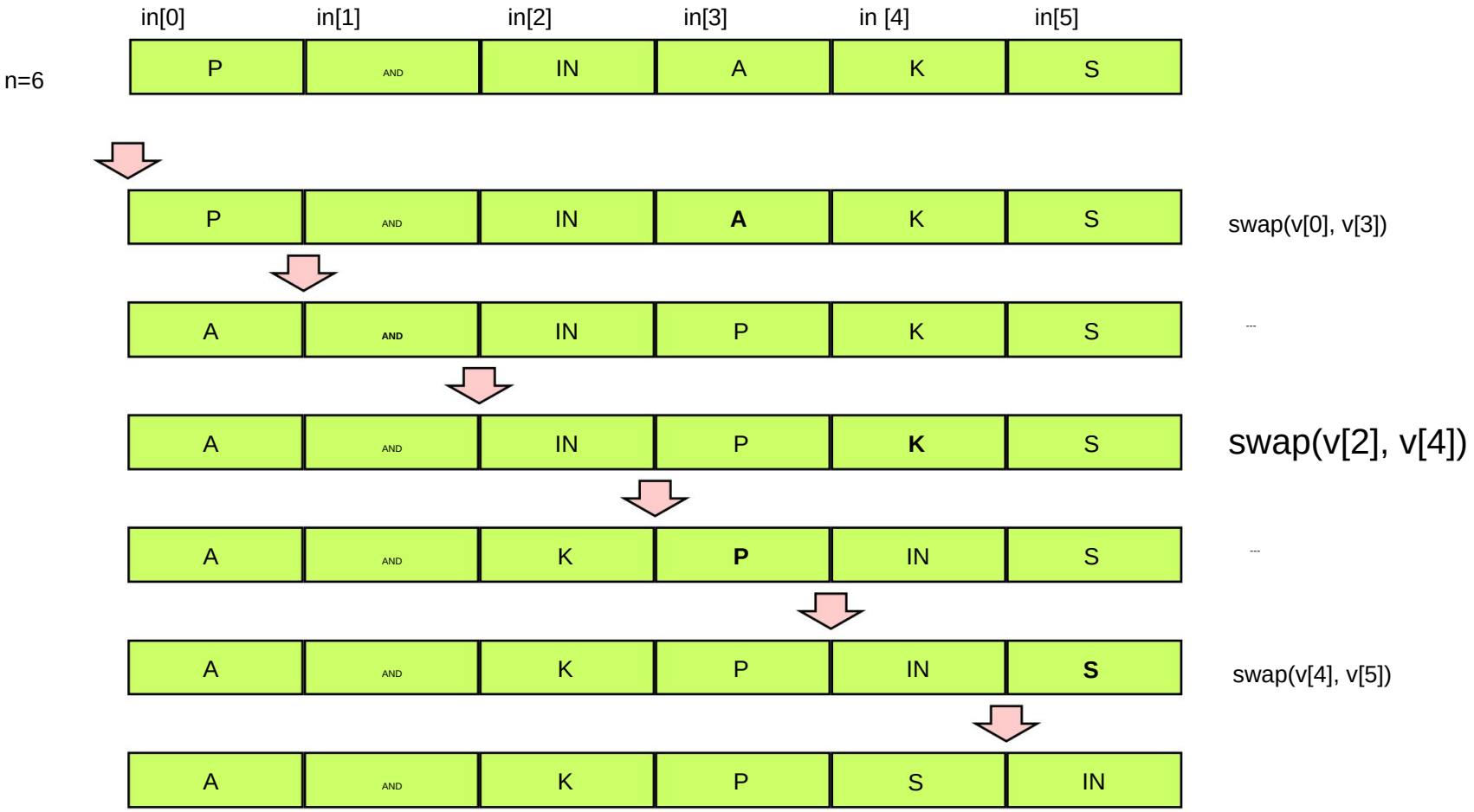
Swapping two elements *(int)* is a central process in sorting.

```
void swap(int &x, int &y) {
    int help = x;
    x = y;
    y = help;
}
```

Machine Translated by Google

Selection sort, example

Example



Machine Translated by Google

Selection sort algorithm

algorithm

```
void Vector::Selectionsort() { int n =
    Length(); int i, j, min;
    for(i = 0; i < n; i++)
    { min = i; for(j = i+1; j < n; j++)


        if(a[j] < a[min]) min = j; swap(a[min],
      a[i]);
    }
}
```

Machine Translated by Google

Selection sort, properties

Characteristics

- Main memory algorithm
- Insitu algorithm
  - sorts in its own data area, only needs a temporary variable, constant disk space consumption
- Expense
  - general O(n2)

Machine Translated by Google

Bubble sort algorithm

Looping through the vector repeatedly.

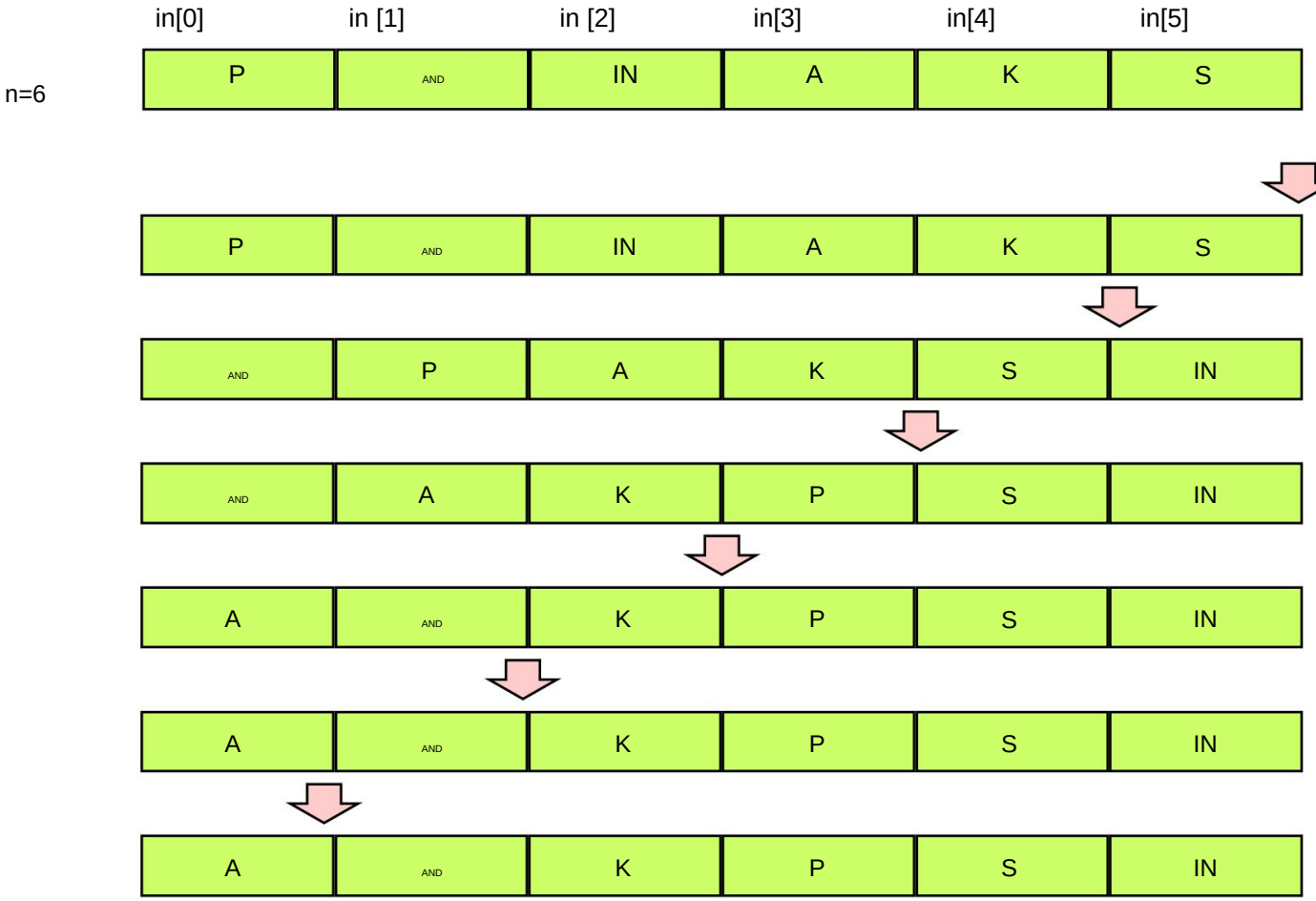
If 2 adjacent elements are out of order (larger before smaller), swap them.

This means that in the first run the largest element is placed last , in the second run the second largest is put in the second to last place, etc. The elements rise like bubbles.

Machine Translated by Google

Bubble sort, example

Example



Machine Translated by Google

Bubble sort, algorithm

algorithm

```
void Vector::Bubblesort() {
    int n = Length(); int i, j;
    for(i = n-1; i
    >= 1; i--)
        for(j = 1; j <= i; j++) if(a[j-1] > a[j])

                swap(a[j-1], a[j]);
}
```

Machine Translated by Google

Bubble Sort, properties

Characteristics

- Main memory algorithm
- Insitu algorithm
  - sorts in its own data area, only needs a temporary variable, constant disk space consumption
- Expense
  - general O(n2)

Machine Translated by Google

3.4.3 Quicksort

Quicksort (Hoare 1962) algorithm

The vector is defined in relation to a freely selectable pivot element

Rearrange the vector elements divided into 2 parts so that all elements to the left of the pivot element are smaller and all elements to the right are larger.

The two partial vectors are recreated independently of each other using quicksort (recursively) sorted.

divide-and-conquer Paradigm

The pivot element is now in its final position. Therefore, only n-1 elements (in the two parts) remain to be sorted (reducing the size of the problem)

Based on the “divide-and-conquer” approach:

- Break down a problem into smaller, non-overlapping sub-problems that are (often) solved recursively. •

Subproblems here: numbers that are smaller than the pivot element and numbers that are larger than the pivot element

Machine Translated by Google

Reorder pivot element

Select any element as pivot element

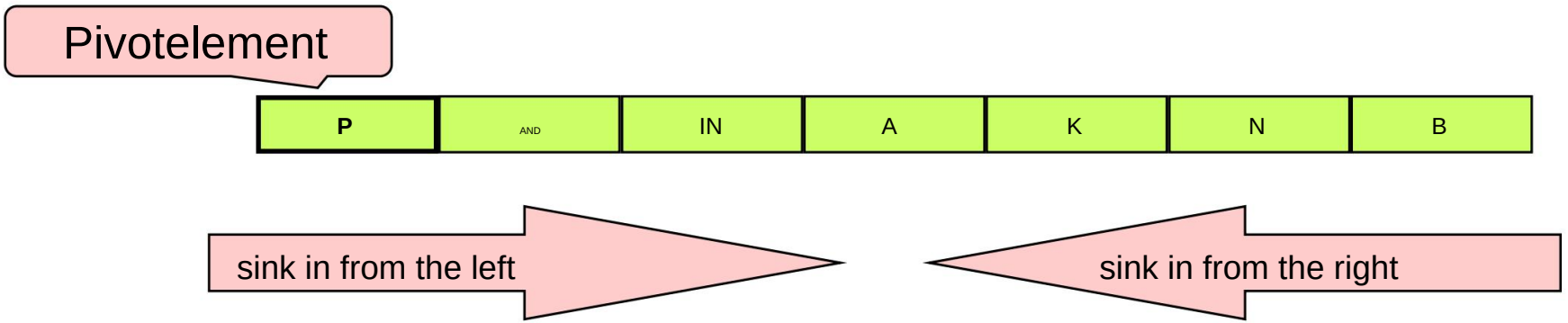
(in example left element in the vector, also random selection or median of 3 random numbers common)

Pivot element 'sink into' the vector from the left or right

i.e. swap

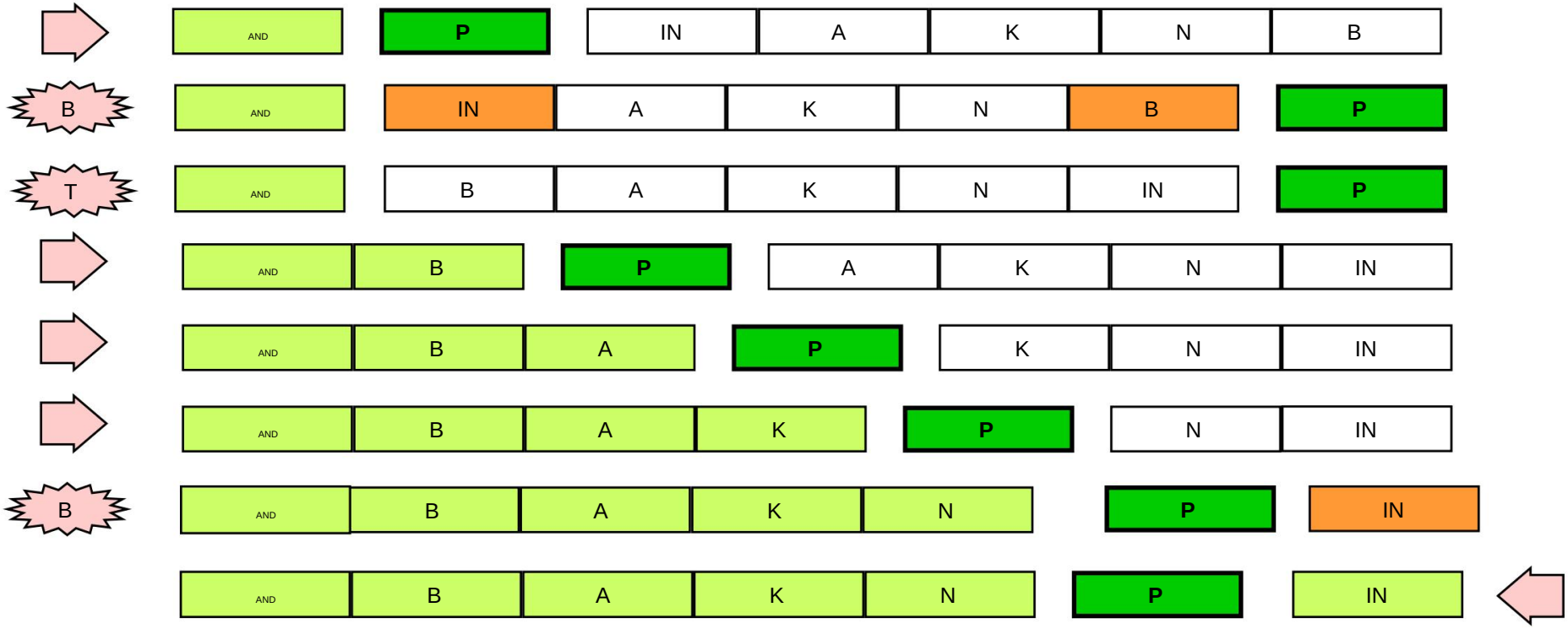
sequentially from the left with all the smaller elements and from the right with all the larger ones.

In case of blocking, ie smaller element on the left and larger element on the right, swap these two elements



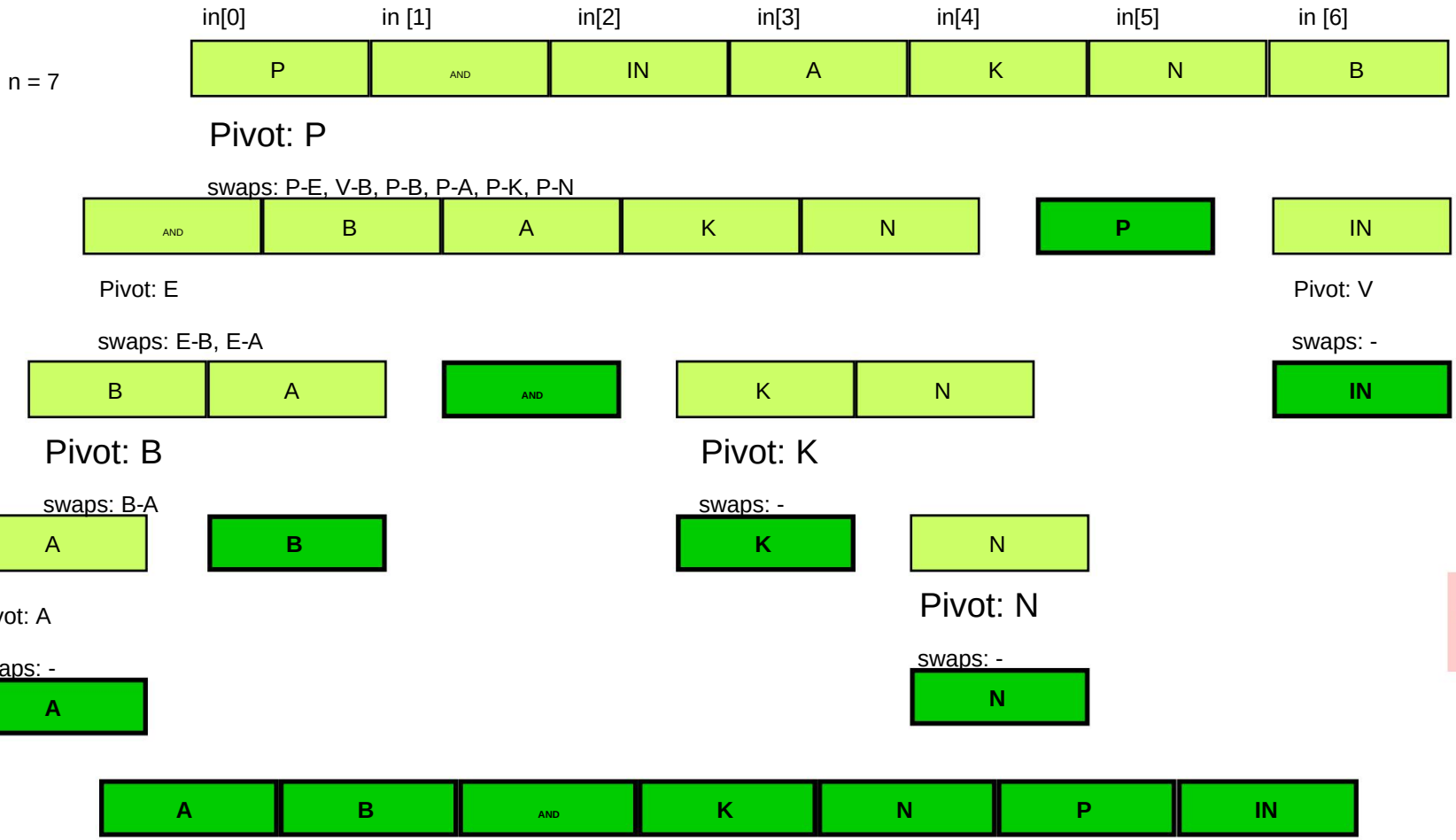
Machine Translated by Google

Reorder pivot element (2)



Machine Translated by Google

Quicksort, recursion



Machine Translated by Google

Quicksort, algorithm


```
void Vector::Quicksort() {
    quicksort(0, Length()-1);

} void Vector::quicksort(int l, int r) {
    int i, j; int pivot; if(r > l) {

        pivot = a[r]; i = l-1; j = r; for(;;) { while(a[+
        +i] < pivot);
            while(a[--j] > pivot) if (j == l)
            break; if(i >= j) break; swap(a[i], a[j]);




        } swap(a[i], a[r]);
        quicksort(l, i-1);
        quicksort(i+1, r);
    }
}
```

Machine Translated by Google


```
void Vector::quicksort(int l, int r) {
  int i, j; int pivot;
  if(r > l) {
    pivot = a[r]; i = l-1; j = r;
    for(;;) {
      while(a[++i] < pivot);
      while(a[--j] > pivot) if (j == l) break;
      if(i >= j) break;
      swap(a[i], a[j]);
    }
    swap(a[i], a[r]);
    quicksort(l, i-1);
    quicksort(i+1, r);
```

3 5 2 6 1 7 4

3 1 2 6 5 7 4

Machine Translated by Google


```
void Vector::quicksort(int l, int r) {
  int i, j; int pivot;
  if(r > l) {
    pivot = a[r]; i = l-1; j = r;
    for(;;) {
      while(a[++i] < pivot);
      while(a[--j] > pivot) if (j == l) break;
      if(i >= j) break;
      swap(a[i], a[j]);
    }
    swap(a[i], a[r]);
    quicksort(l, i-1);
    quicksort(i+1, r);
```

7 6 5 4 3 2 1

Machine Translated by Google

```
void Vector::quicksort(int l, int r) {
  int i, j; int pivot;
  if(r > l) {
    pivot = a[r]; i = l-1; j = r;
    for(;;) {
      while(a[++i] < pivot);
      while(a[--j] > pivot) if (j == l) break;
      if(i >= j) break;
      swap(a[i], a[j]);
    }
    swap(a[i], a[r]);
    quicksort(l, i-1);
    quicksort(i+1, r);
```

5 6 7 3 2 1 4

Machine Translated by Google


```
void Vector::quicksort(int l, int r) {
  int i, j; int pivot;
  if(r > l) {
    pivot = a[r]; i = l-1; j = r;
    for(;;) {
      while(a[++i] < pivot);
      while(a[--j] > pivot) if (j == l) break;
      if(i >= j) break;
      swap(a[i], a[j]);
    }
    swap(a[i], a[r]);
    quicksort(l, i-1);
    quicksort(i+1, r);
```

i can never be larger than r because pivot =a[r]

if (j==l) then have

- all elements at positions k < ir a value > pivot
- i == r

if (i >= j) then have

- all elements at positions k < i have a value >= pivot
- the element at position i has a value >= pivot
- all elements at positions k > i have a value <= pivot

Machine Translated by Google

Quicksort, properties

Main memory algorithm

Insitu algorithm

sorts in its own data area, only needs temporary auxiliary variables

(Stack), constant memory consumption

Expense

On average O(n*log(n))

Can degenerate to O(n2).

Example: Vector is pre-sorted and the first or is always used as the pivot element last vector element selected

Sensitive to careless choice of pivot element

Stability is not easily achieved

Recursive algorithm

Complex to implement if recursion is not available

Machine Translated by Google

3.4.4 Mergesort

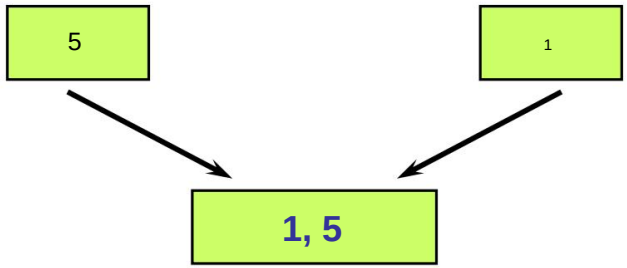
Mergesort (???, 1938) algorithm

- The vector to be sorted is recursively divided into partial vectors of half length until the (trivial, scalar) vectors consist of a single element
- Afterwards, 2 partial vectors each become a vector that is twice as large merged (sorted)
  - When merging, the first two elements of the vectors are successively compared and the smaller one is transferred to the new vector until all Elements in the new vector are stored sorted
  - The merging is repeated until there are 2 vectors in the last phase Length n/2 merge into a sorted vector of length n
- “divide-and-conquer” approach

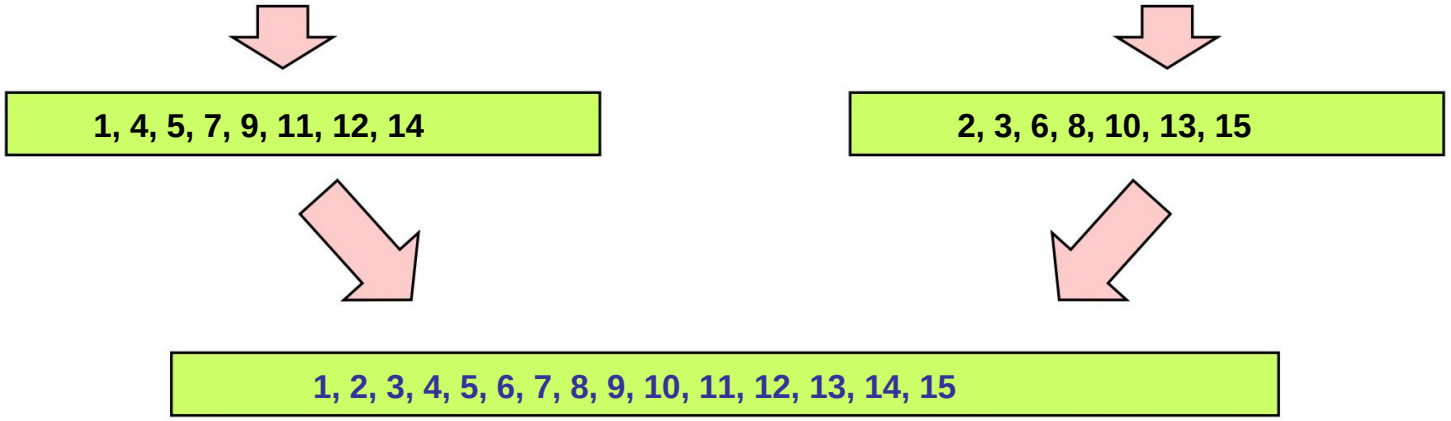
Machine Translated by Google

Merging two vectors

trivialer Fall

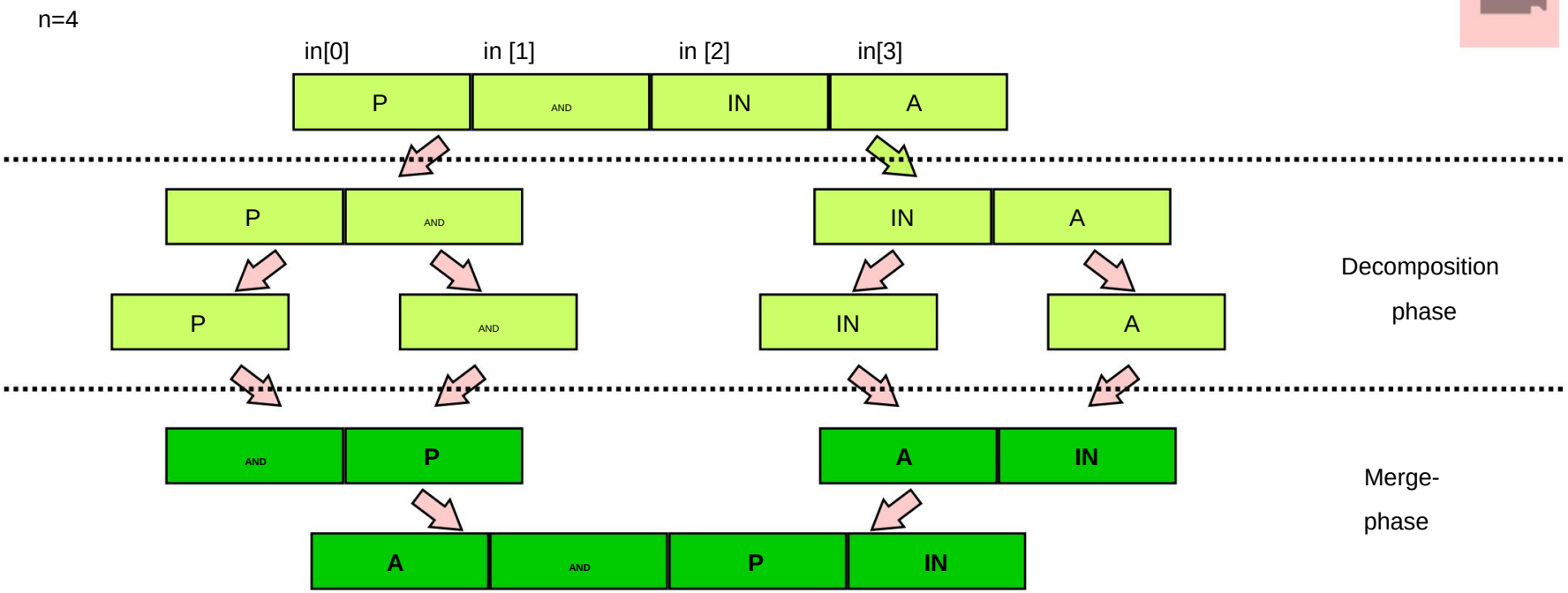


general case



Machine Translated by Google

Mergesort, Recursion



Machine Translated by Google

Class Vector

```
class Vector {
    int *a, size; public:

    Vector(int max) { a = new int[max]; size = max; }
    ~Vector() { delete[] a; } void
    Selectionsort(); void
    Bubblesort(); void
    Quicksort(); void
    Mergesort(); int Length();
    private: void
quicksort(int,
    int); void mergesort(int, int, int*);
    void swap(int&, int&); };
```

Machine Translated by Google

Merge sort, algorithm (naive)

```
void Vector::Mergesort() {
  if (size>1) { int
        m=size/2; Vector
        left(m); Vector
        right(size-m); for (int i=0; i<m;
        ++i) left.a[i]=a[i]; for (int i=0; i<m; ++i) right.a[i]=a[i+m];


        left.Mergesort();
        right.Mergesort();

        int li=0; int
        ri=0; for (int
        i=0; i<size; ++i)
            if (li>=left.size) a[i]=right.a[ri++]; else if (ri>=right.size)
            a[i]=left.a[li++]; else a[i]=(right.a[ri]<left.a[li])?

                                                    right.a[ri++]:left.a[li++];
    }
}
```

covers special cases?

Machine Translated by Google

Merge sort, algorithm (optimized)

```
void Vector::Mergesort() {
    int *buf=new int[Length()]; mergesort(0,
    Length()-1, buf); delete[] buf;

}

void Vector::mergesort(int left, int right, int* buf) {
    if(right > left) { int i, j, k,
        m; m = (right+left)/2;
        mergesort(left, m, buf);
        mergesort(m+1, right, buf);

        for(i=m+1; i>left; --i) buf[i-1] = a[i-1]; for(j=m; j<right; ++j)
        buf[right+m-j] = a[j+1]; // reverse order

        for(k=left; k<=right; ++k)
            a[k]=(buf[i]<buf[j])?buf[i++]:buf[j--];
    }
}
```

buf is temporary vector

// i = left, j = right

Machine Translated by Google

Merge sort, algorithm (optimized)


```
void Vector::Mergesort() {
    int *buf=new int[Length()]; mergesort(0,
    Length()-1, buf); delete[] buf;


}

void Vector::mergesort(int left, int right, int*
     buf) { if(right > left) { int i, j, k,
    m; m = (right+left)/2;
        mergesort(left, m,
        buf); mergesort(m+1, right,
        buf);



        for(i=m+1; i>left; --i) buf[i-1] = a[i-1]; for(j=m; j<right; ++j)
        buf[right+m-j] = a[j+1]; for(k=left; k<=right; ++k)

            a[k]=(buf[i]<buf[j])?buf[i++]:buf[j--];
    }
}
```

```
void Vector::Mergesort() {
 if (size>1) {
    int m=size/2;
    Vector left(m);
    Vector right(size-m);
    for (int i=0; i<m; ++i) left.a[i]=a[i];
    for (int i=0; i<m; ++i) right.a[i]=a[i+m];

    left.Mergesort();
    right.Mergesort();

    int li=0;
    int ri=0;
    for (int i=0; i<size; ++i)
      if (li>=left.size) a[i]=right.a[ri++];
      else if (ri>=right.size) a[i]=left.a[li++];
      else a[i]=(right.a[ri]<left.a[li])?
                                right.a[ri++]:left.a[li++];
  }
}
```

Sonderfälle abdeck

correct?

Machine Translated by Google

Merge sort, algorithm (optimized)

```
void Vector::Mergesort() {
    int *buf=new int[Length()]; mergesort(0,
    Length()-1, buf); delete[] buf;

}

void Vector::mergesort(int left, int right, int* buf) {
    if(right > left) { int i, j, k,
        m; m = (right+left)/2;
        mergesort(left, m, buf);
        mergesort(m+1, right, buf);

        for(i=m+1; i>left; --i) buf[i-1] = a[i-1]; for(j=m; j<right; ++j)
        buf[right+m-j] = a[j+1]; for(k=left; k<=right; ++k)

            a[k]=(buf[i]<buf[j])?buf[i++]:buf[j--];
    }
}
```

possible overflow

Machine Translated by Google

Merge sort, properties

Characteristics

Main memory algorithm, but also as an external memory algorithm usable.

External sorting: instead of partial phase, predetermined data division is often used, or only divided until partial vector (file) can be sorted in main memory.

Stable sorting process

Mergen must be implemented correctly

Exsitu algorithm needs

a second equally large auxiliary vector for restoring

Expense

General O(n*log(n)) Master theorem!

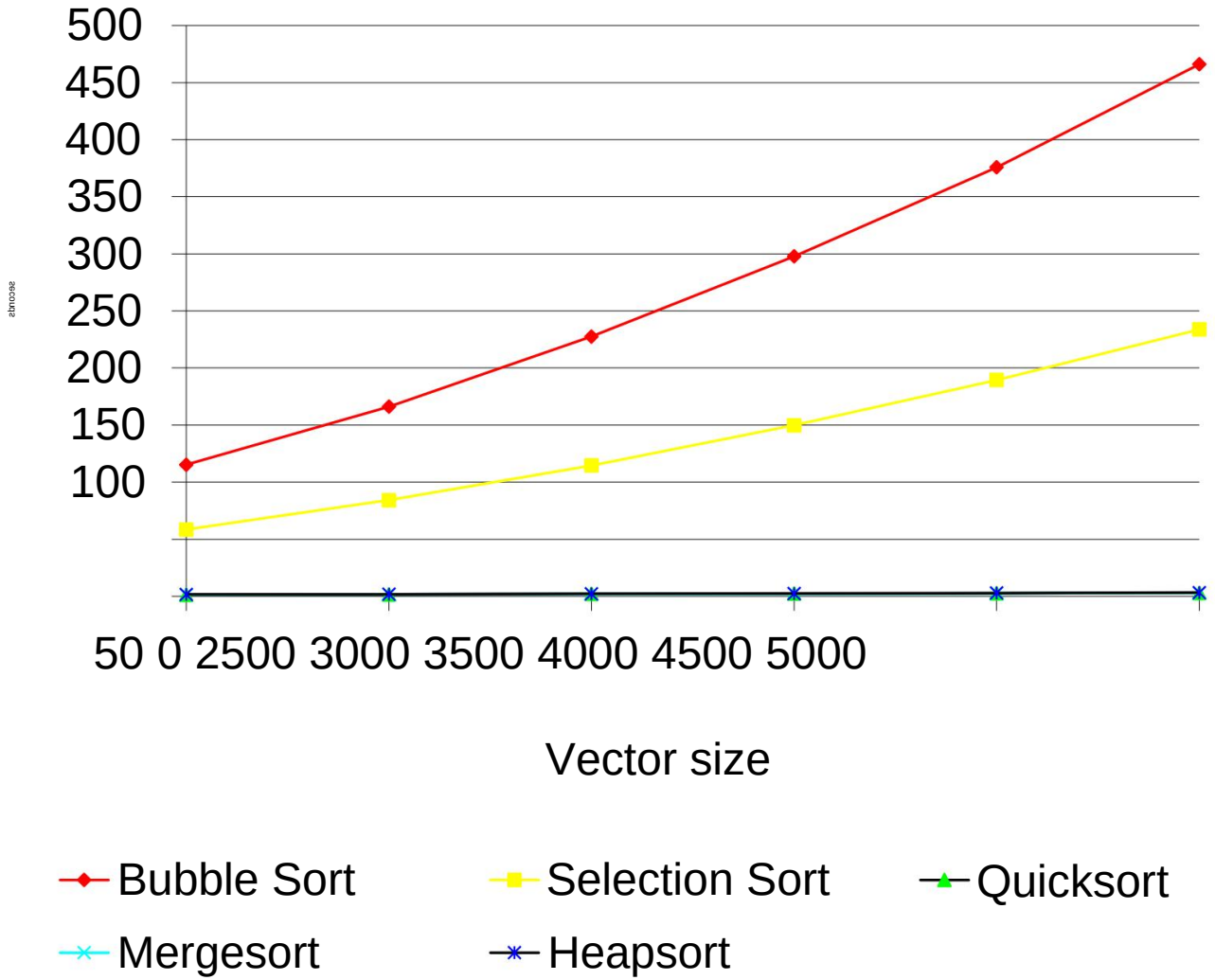
Takes double space Recursive

algorithm

Complex to implement if recursion is not available.

Machine Translated by Google

Runtime comparison of the procedures



Machine Translated by Google

3.4.5 Comparison sort procedure

These well-known procedures are called *comparative Sorting method* (Comparison sort)

The order of the elements is determined by comparing the elements with each other

Only comparisons of the form xi < xj , xi ÿ xj , ... applied

These algorithms show an order of O(n log n) as the cheapest running time , which means that we have so far been able to determine the lower limit as ÿ(n log n).

Conjecture: The problem of sorting by comparing the Elements can be described with ÿ(n log n).

If valid ÿ Algorithmic gap closed since ÿ(P) = O(A)

Machine Translated by Google

Decision tree

Consideration of comparative sorting methods

Decision trees

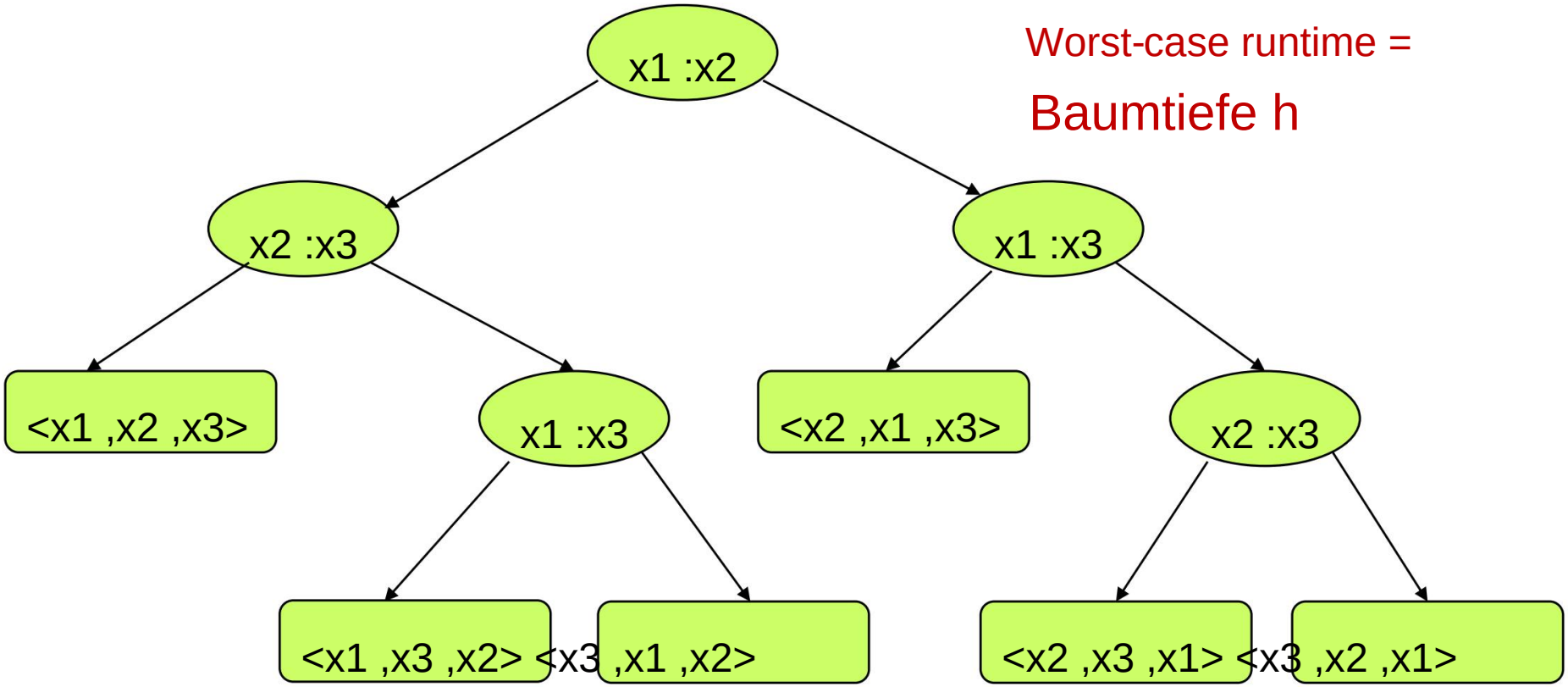
- The decision tree contains all possible comparisons of a
  - Sorting process by any sequence of a given one
    sort length
    - The leaf nodes represent all possible permutations of the input sequence
- The paths from the root to the leaves define the necessary ones
  - Compare to achieve the corresponding permutations

Machine Translated by Google

Example decision tree

Decision tree for 3 elements ie 3! = 6

possible permutations of the input elements < x1 , x2 , x3 > Left successor relation x:y fulfilled, right successor not fulfilled



Worst-case runtime =

Baumtiefe h

Machine Translated by Google

Proof: Lower limit for the worst case

We ignore runtime of swapping, administration operations, etc.

The number of leaves in the decision tree is n!

The number of leaves in a binary tree of depth h is ÿ 2 h

therefore

$$2^{h} \ddot{y} n!$$

$$h \ddot{y} \lg(n!)$$

Stirling approximation $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^{n}\left(1+\ddot{y}\left(\frac{1}{n}\right)\right) \ddot{y} \left(\frac{n}{e}\right)^{n}$

$$h \ddot{y} \lg\left(\frac{n}{e}\right)^{n} = n\lg n - n\lg e$$

$$h = \ddot{y}(n \lg n)$$

i.e. Algorithmic Gap for Comparative Sorting Methods closed

Machine Translated by Google

Sorting methods that do NOT rely on comparing two values can sort an input sequence in linear time

- Reach order O(n)

Examples

- Counting Sort
- Sort Radix
- Bucket Sort

Machine Translated by Google

3.5.1 Counting Sort

1954 First used by Harold H.Seward MIT thesis “Information Sorting in the Application of Electronic Digital Computers to Business Operations” as the basis for Radix Sort.

1956 First published by EH Fried

1960 under the name Mathsort by W. Feurzig.

Machine Translated by Google

Counting Sort

Condition

The n elements to be sorted are integer values in the range 0 to k

Usually k>n applies

If **k is of order n** (k = O(n)) the effort for this is

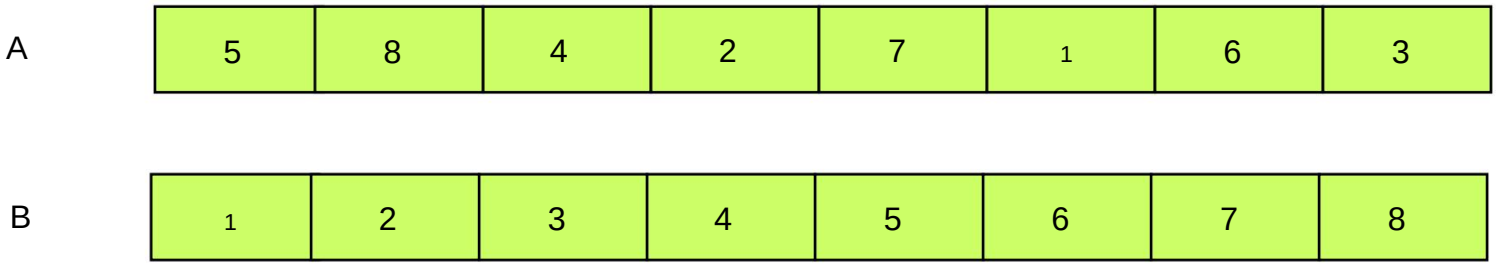
Sortierverfahren Counting Sort O(n)

Approach

An exact permutation is given

Each element is simply assigned to the position of its value in a

Target vector set



Machine Translated by Google

Counting Sort (2)

extension

- The elements come from the range [1..k] and there are no duplicate values
- Question: how can we extend the first approach?

General case

- The elements are from the range [1..k], duplicates are allowed. Idea of counting sort
- For each element x, determine the number of elements smaller than x in the vector, use this information to place the element x at its position in the resulting vector

Machine Translated by Google

Counting sort algorithm

Input vector A, result vector B, auxiliary vector **C** (C has size k)

```
for(i=1; i<=k; i++)
    C[i] = 0;
for(j=1; j<=length(A); j++)
    C[A[j]] = C[A[j]] + 1;
for(i=2; i<=k; i++)
    C[i] = C[i] + C[i-1];
for (j=length(A); j>=1; j--) {
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]] – 1;
}
```

1: C[i] contains the number of elements equal to i, i = 1,2,...,k

2: C[i] contains the number of Elements less than or equal to i

3: Each element is placed in its correct position

If all elements A[j] are different, the correct position is in C[A[j]], since elements can be equal the Value C[A[j]] decreased by 1

Machine Translated by Google

Counting sort example

1:

A 3 6 4 1 3 4 1 4

C 2 0 2 3 0 1

2:

C 2 2 4 7 7 8

3:

B 4

1st pass

C 2 2 4 **6** 7 8

```
for (j=length(A); j>=1; j--)
  B[C[A[j]]] = A[j];
  C[A[j]] = C[A[j]] - 1;
}
```

3:

B 1 4

2. D.

C **1** 2 4 6 7 8

3:

B 1 4 4

3. D.

C 1 2 4 **5** 7 8

3:

End

B 1 1 3 3 4 4 4 6

Machine Translated by Google

Counting Sort Analyse

```
for(i=1; i<=k; i++)                                   OK)
    C[i] = 0;
for(j=1; j<=length(A); j++)                           O(n)
    C[A[j]] = C[A[j]] + 1;
for(i=2; i<=k; i++)                                   OK)
    C[i] = C[i] + C[i-1];
for (j=length(A); j>=1; j--) {                        O(n)
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]] – 1;
}
```

It follows that the total effort is O(n + k).

In practice we very often have k = O(n), which means we have a real overhead of O(n).

- Note that there was no comparison between values, but rather the values of the elements were used for placement (see hashing)

Counting Sort is a stable sorting method

- Guaranteed by final loop working from back to front

No insitu method, 3 vectors of size O(n) necessary

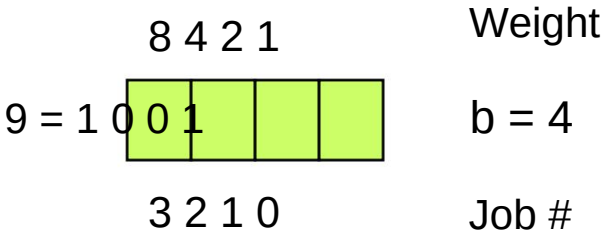
Machine Translated by Google

3.5.2 Radix Sort

Radixsort looks at the structure of the key values

The key values of length b are in a base number system
M represented (M is the radix)

e.g. M=2, the keys are represented in binary, b = 4



Keys are sorted by comparing their individual bits at the same bit position

The principle can also be extended to alphanumeric strings

Machine Translated by Google

Radix sort algorithm

General radix sort algorithm

for(index i runs over all b places) {

sort key values with a (stable)

Sorting procedure regarding location i

}

Direction of index run, stability

Left to right

d.h. **for(int i=b-1; i>=0; i--) { ... }**

Leads to *Binary Quicksort (Radix Exchange Sort)*

Right to left and stable sorting process

d.h. **for(int i=0; i<b; i++) { ... }**

Leads to the *LSD Radix Sort (Straight Radix Sort)*

Machine Translated by Google

Binary quicksort

Very old procedure

In 1929, for the first time for machine sorting of punch cards.

1954 H.H. Seward [MIT R-232 (1954), p25-28]

independent and detailed: P.Hildebrandt J. Schwartz H. Isbitz, H. Rising, [JACM 6

(1959), 156-163]

1 year before Quicksort

Machine Translated by Google

Binary quicksort

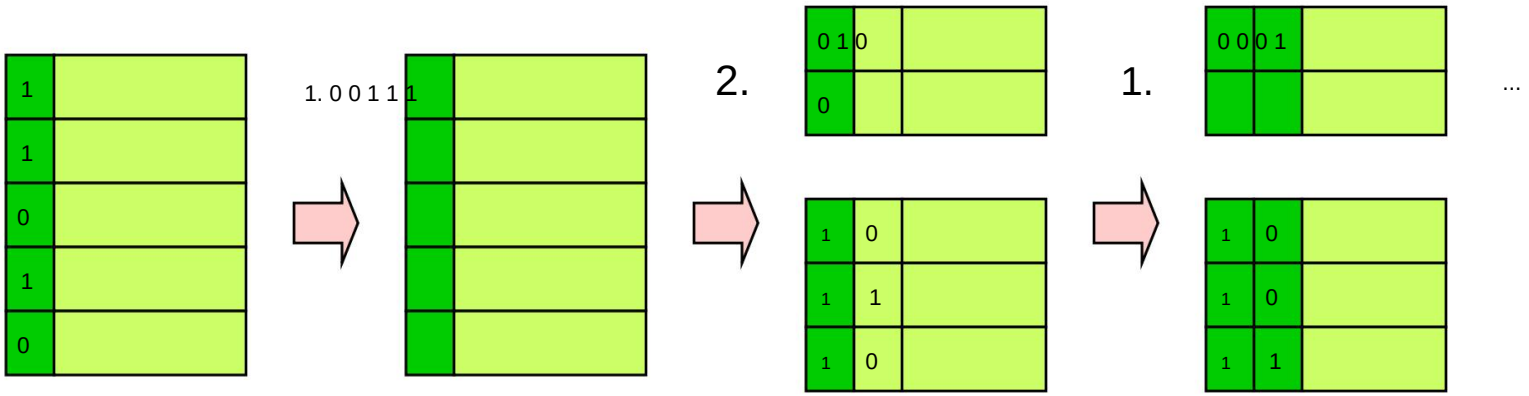
Binary quicksort algorithm

Look at spots from left to right

1. Sort data sets based on the bit under consideration 2. Divide the

data sets into M independent groups, ordered by size, and sort the M groups recursively, ignoring the bits already under consideration

Example



Machine Translated by Google

Binary quicksort partition

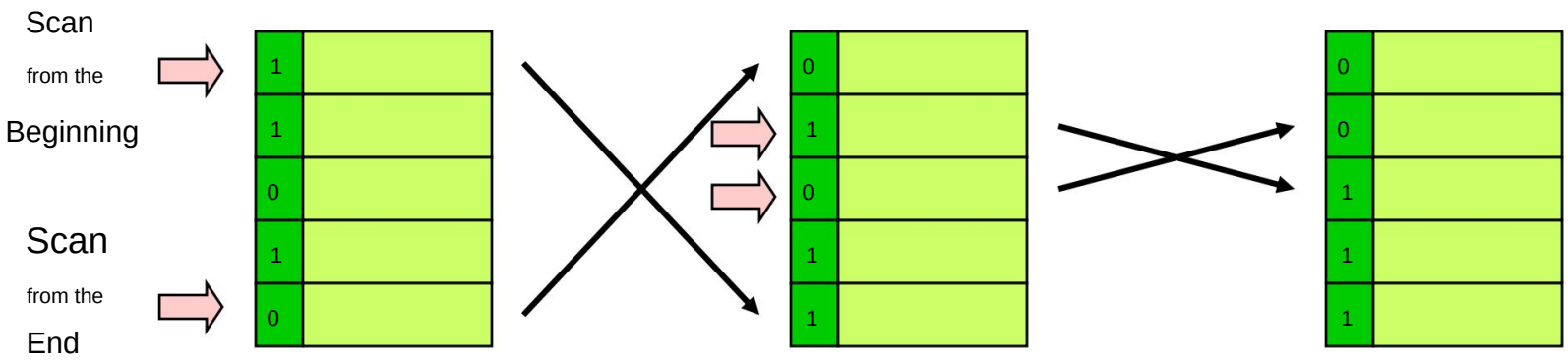
Splitting the data sets using an in-situ approach similar to that of Partitioning in quicksort

do {

scan top-down to find key starting with 1; scan bottom-up to find key starting with 0; exchange keys;

} while (not all indices visited)

Example

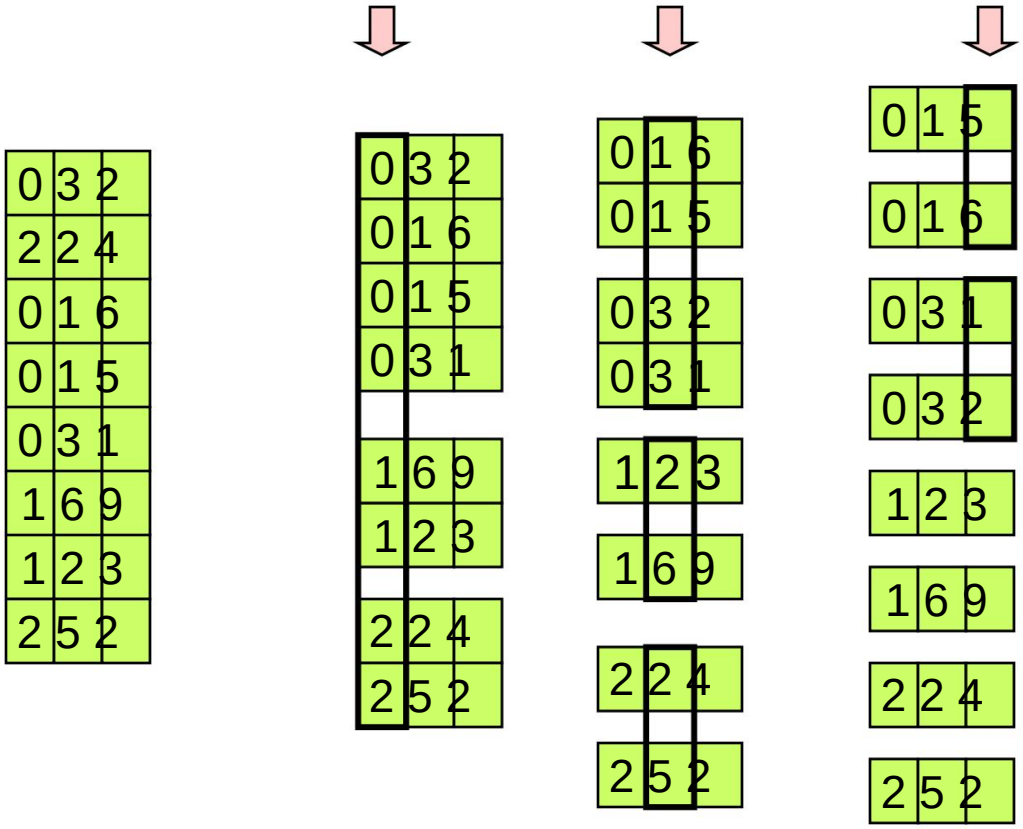


Machine Translated by Google

Analogue: “Decimal” quicksort for Decimal numbers

Number of groups M is 10

Digit values 0 to 9



Machine Translated by Google

Binary quicksort properties

Binary quicksort uses on average approximately N log N

bit comparisons log N (encoding the numerical value) = usually constant factor

N = Number of values to sort

Well suited for small numbers

Requires relatively less memory than other linears Sorting method

It is an unstable sorting method

Similar to quicksort only for positive integers

Machine Translated by Google

universität wien

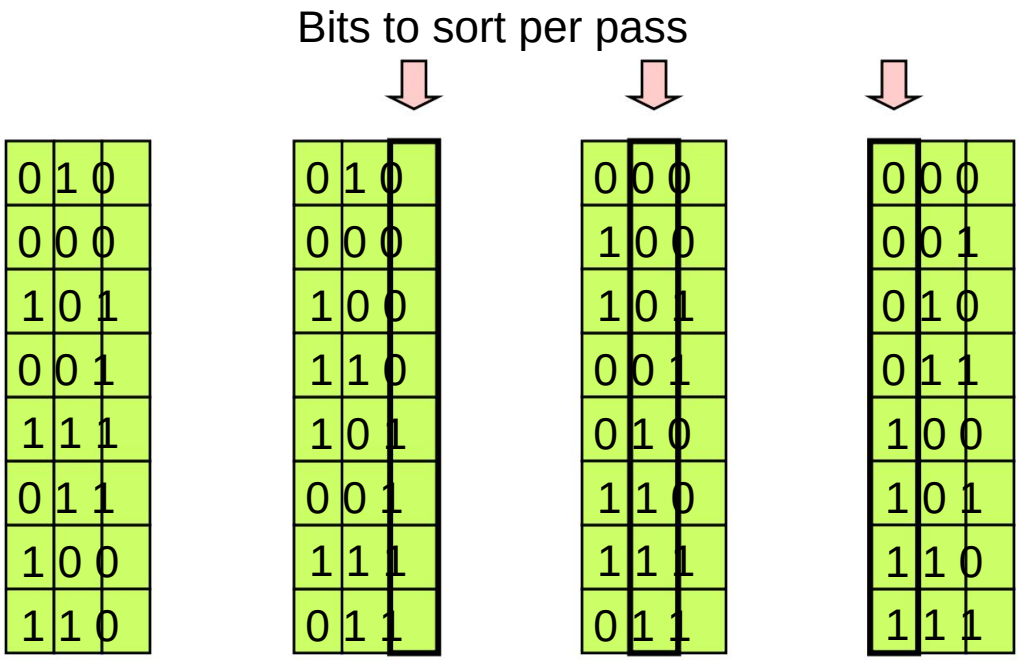
LSD radix sort algorithm

Look at bits from right to left

LSD … “least significant digit”

Sort data sets stably (!) based on the bit under consideration

In binary quicksort from left to right!



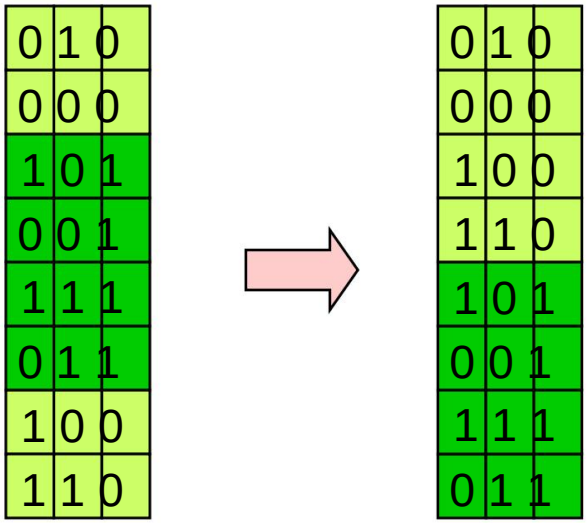
Machine Translated by Google

What does “stable” mean?

With a stable sorting process, the relative order of the same keys (= value under consideration) is not changed

Example

- First pass of last example
- The relative order of the Keys that end with 0 remain unchanged, the same applies to keys that end with 1



Stability guarantees correctness algorithm

- Possible procedure: Counting Sort

Machine Translated by Google

What does “stable” mean?

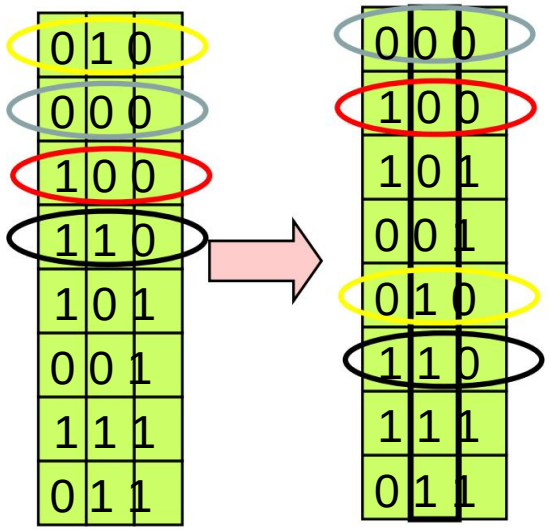
With a stable sorting process, the relative order of identical keys is not changed

Example

- First pass of last example
- The relative order of the Keys that end with 0 remain unchanged, the same applies to keys that end with 1

Stability guarantees correctness algorithm

- Possible procedure: Counting Sort



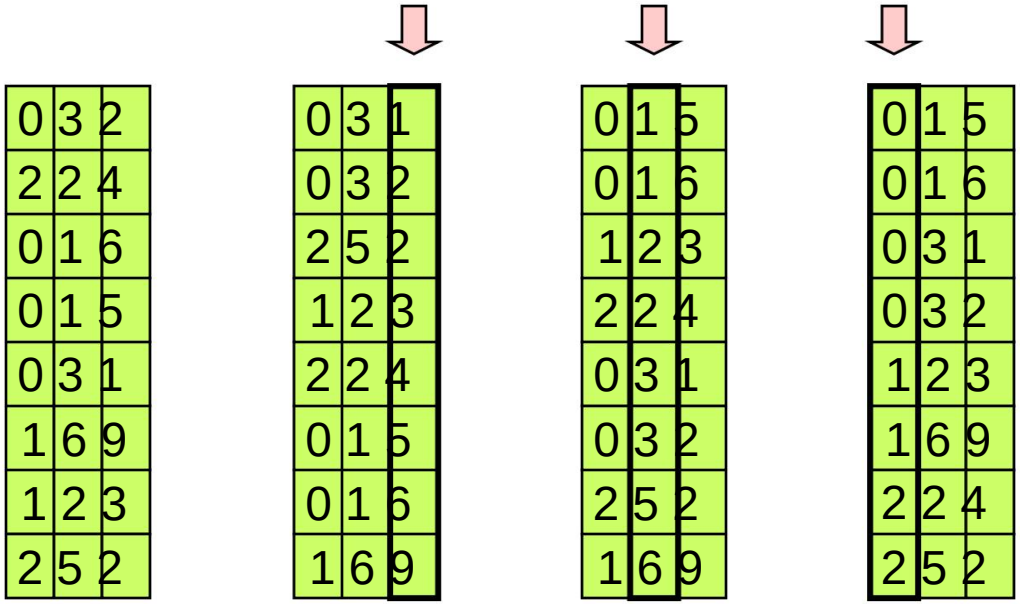
Relative order of blue and red remains unchanged because they have the same value in the 2nd bit

Machine Translated by Google

LSD radix sort for decimal numbers

Number of groups M is 10

Digits 0 to 9



Machine Translated by Google

Analysis of Radix Sort

Sorting n numbers, key length b

```
for(Index i runs over all b positions) {
    sort n key values with a (stable)
        Sorting procedure regarding location i
}
```

When using a sorting method with a runtime of O(n) (such as partitioning in Radix Exchange Sort, Counting Sort in Straight Radix Sort) is therefore the order of Radix Sort O(bn)

If you look at b as a constant you get O(n)

- For a numerical range of m values, representation on the computer with b = log(m)
  - However, since the range of values on the computer is limited, approach b is consistently acceptable

No in-situ procedure, double storage space required (for that stable sorting processes)

- In practice, this leads to the use of O(n log n) methods

Machine Translated by Google

3.5.3 Bucket Sort

Assumption about the n input elements that they are uniform are distributed over the interval [0,1).

Bucket sort algorithm

- Divide the interval [0,1) into n equal-sized sub-intervals (buckets) and distribute the n input elements among the buckets
  - Since the elements are evenly distributed, only a few elements fall into the same bucket
- Sort the items per bucket using a different sorting method (e.g. B. Selection Sort)
- Visit the buckets along the interval and output the sorted items

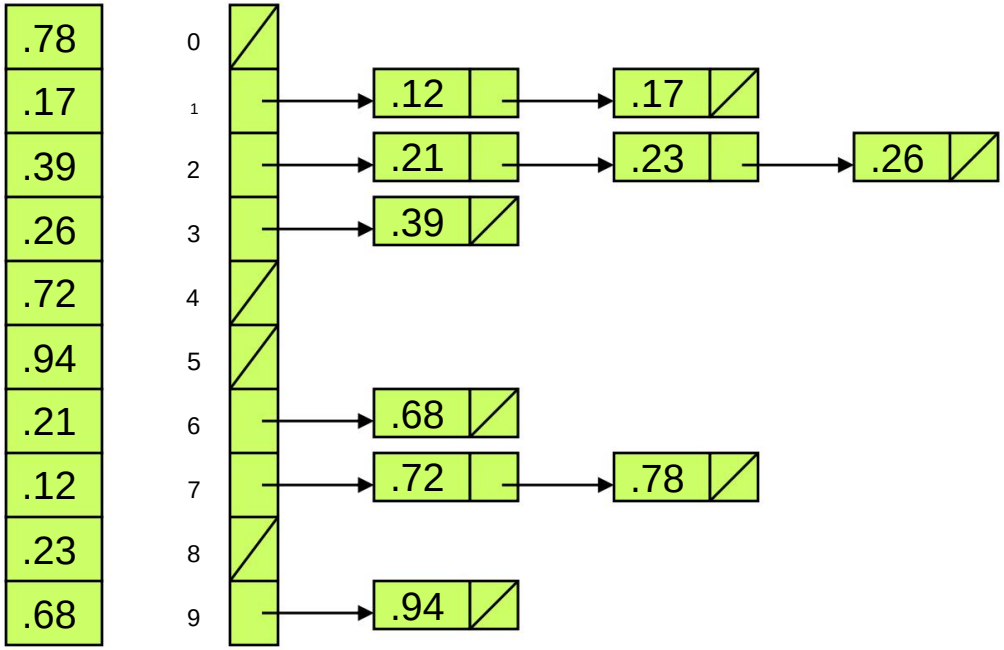
Machine Translated by Google

Bucket sort example

Input vector A of size 10

Bucket Vector B manages elements via linear lists

Bucket B[i] contains the values of the interval [i/10, (i+1)/10)



Machine Translated by Google

Bucket sort algorithm


```
n = length(A);
for(i=1; i<=n; i++) insert A[i]
    into list B[ÿn*A[i]ÿ];
for(i=0; i<n; i++)
    sort list B[i] with insertion sort;
Concatenate the lists B[0], B[1], ..., together in order      B[n-1]
```

Machine Translated by Google

Analyse Bucket Sort

All statements except sort are of order O(n)

Analysis of sorting

ni denotes the random number of elements per bucket

Selection Sort runs in O(n2) time, i.e. the expected time for sorting of a bucket is E[O(n2)] = O(E[n2]), since all buckets have to be sorted

$$O\left(\sum_{i=0}^{n-1} n_i^2\right) = \sum_{i=0}^{n-1} O\left(n_i^2\right)$$

To calculate the expression

Determine distribution of the random variable

ni Assume that the probability of an element falling into a bucket is p = 1/n (e.g. elements in [0,1) equally distributed) ni follows the

binomial distribution E[ni] =

n*p = 1 and Var [ni] = n*p*(1-p) = 1 – 1/n

$E[n_i^2] = Var[n_i] + E2[n_i] = 1 - 1/n + 12 = 2 - 1/n = \ddot{y}(1)$

It follows that the effort for bucket sort is below Assumption ( ) is

Machine Translated by Google

What do we take with us?

Dictionary

Hashing

- Static and dynamic procedures

sort by

- Classic methods O(n2) and O(n log n)
- Linear methods O(n)