

COS 226 Lecture II: External sort/search

HUGE FILE:

- 1970: 1 million bytes
- 2000: 1 trillion bytes

Use high-level abstraction

- local memory
 - random access
 - faster sequential access
- external memory
 - far slower than local memory
 - read/write in large blocks
 - faster sequential access
 - restrictions on access

HUGE FILE:

- thousands of times the size of local memory

Problem: Sorting and Searching for huge files

External sort/search algorithms

SORTING

- N: number of records
- M: size of memory
- $2P$: number of external devices

algorithms: balanced merge, polyphase merge

SEARCHING

- N: number of records
- M: page size

algorithms: indexed sequential, B-trees, extendible hashing

Many ancient algorithms still relevant

Balanced multiway merge

Make multiple passes over the file

- can MERGE huge sorted blocks in memory
- use half the devices for "output"

pass 0:

- sort the file into sorted blocks of size M
on devices $0, 1, 2, \dots, P-1$

pass 1: make blocks of size $M \cdot P$

- P -way merge out to devices $P, P+1, P+2, \dots, 2 \cdot P$

pass 2: make blocks of size $M \cdot P^2$

- P -way merge out to devices $0, 1, 2, \dots, P-1$

...

pass t : make blocks of size $M \cdot P^t$

File is sorted when $M \cdot P^t > N$

- $t = \log_P (N/M)$ passes through the data

Balanced merge analysis

Can actually make initial runs of size $2M$ (use a PQ)

.	PC sort	workstation
. file	1 billion	1 trillion
. memory	1 million	1 billion
. devices	4	10
. passes	$\lg 500 = 9$	$\log_5 500 = 5$

Costs

- number of passes (no. times each datum touched)
- number of devices

Bottom line:

Can sort a file in 5-10 times the amount
of time it takes to read or write it

IF enough devices are available

Balanced merge example

input

T H E Q U I C K B R O W N F X J M P S V L A Z Y D G

sorted runs of size 3

E H T **I Q U** B C K O R W F N X J M P L S V A Y Z D G *

sorted runs of size 9

B C E H I K Q T U **F J M N O P R W X** A D G L S V Y Z *

output

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z *

	sorted runs (lengths) on devices						merge cost
.	0	1	2	3	4	5	
.	-----						
.	3(3)	3(3)	3(3)	0	0	0	
.	0	0	0	1(9)	1(9)	1(9)	27
.	1(27)	0	0	0	0	0	27

Polyphase Merge

Concept:

- cut the number of devices needed in half

MERGE-UNTIL-EMPTY

- one device empty
- different number of runs
on other devices
- merge to empty device
until another device is empty
- iterate

Strategically place runs on devices such that

- last merge empties all but one device to one final run

Polyphase merge run placement

Work backwards to figure placement

Ex: 4 devices

.	1	0	0	0
.	0	1	1	1
.	1	0	2	2
.	3	2	0	4
.	7	6	4	0
.	0	13	11	7
.	13	0	24	20

distribution for 57 runs

General pattern is complicated!

Polyphase merge example

sorted runs of size 3

. 0 0 0 1 1 3 3 3 3
 E H T I Q U B C K O R W F N X J M P L S V A Y Z D G *

result of phase 1

. 0 3 3 2 2
 B C K A Y Z D G * E H J M O P R T W F I L N S Q U V X

result of phase 2

. 3 2 1
 D G * F I L N S Q U V X A B C E H J K M O P R T W Y Z

output

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z *

sorted runs (lengths) on devices

merge cost

.	0	1	2	3	
.	-----				
.	3(3)	2(3)	0	4(3)	
.	1(3)	0	2(9)	2(3)	18
.	0	1(15)	1(9)	1(3)	15
.	1(27)	0	0	0	27

Polyphase merge analysis

Ex. 34 initial sorted runs, 3 devices

.	0	21	13
.	13	8	0
.	5	0	8
.	0	5	3
.	3	2	0
.	1	0	2
.	0	1	1
.	1	0	0

Fibonacci numbers

- $\phi = 1.618\dots$ (golden ratio)
- ϕ^N runs in ϕ phases
- each touching $1/\phi$ of the data

THM: Total number of passes is $\log_{\phi}(N/M)/\phi$

Polyphase merge analysis (continued)

General analysis complicated (generalized Fibonacci numbers)

.	85	56	0	108	90
.	29	0	56	52	44
.	0	29	27	23	15
.	15	14	12	8	0
.	7	6	4	0	8
.	3	2	0	4	4
.	1	0	2	2	2
.	0	1	1	1	1
.	1	0	0	0	0

.		PC sort	workstation
.	file	1 billion	1 trillion
.	memory	1 million	1 billion
.	devices	3	5
.	passes*	8	5

*: effective passes (not all phases touch all data)

Bottom line:

- About half as many devices as balanced merge

Virtual memory

Idea: use quicksort (!)

Paging system "automatically" reads data in blocks

Requirements:

- ability to address huge files
- locality of reference in program

Partitioning on such a system:

- read a block from the left
- read a block from the right
- write small elements on left
- write small elements on right

Paging systems takes care of details

Cost estimate:

- 2 devices (or more with MSD radix sort)
- $2 \ln(N/M)$ passes (or fewer with MSD radix sort)

Basic external searching

N records

stored in pages of size M

Goal:

- find item with given key
- read as few pages as possible

INDEXED SEQUENTIAL ACCESS

Store file sequentially on devices (like encyclopedia)

- build index for each device
- build master index for devices

Problem:

- index is hard to maintain under insertion/deletion
- real goal: DYNAMIC index

B-trees

Generalize 2-3-4 trees: up to M links per node

Split full nodes on the way down

Red-black abstraction still works

- BUT might use binary search instead of internal links

B-trees for external search

- node size = page size
- typical: $M = 1000$

Main advantage: flexibility to do fast insert/delete

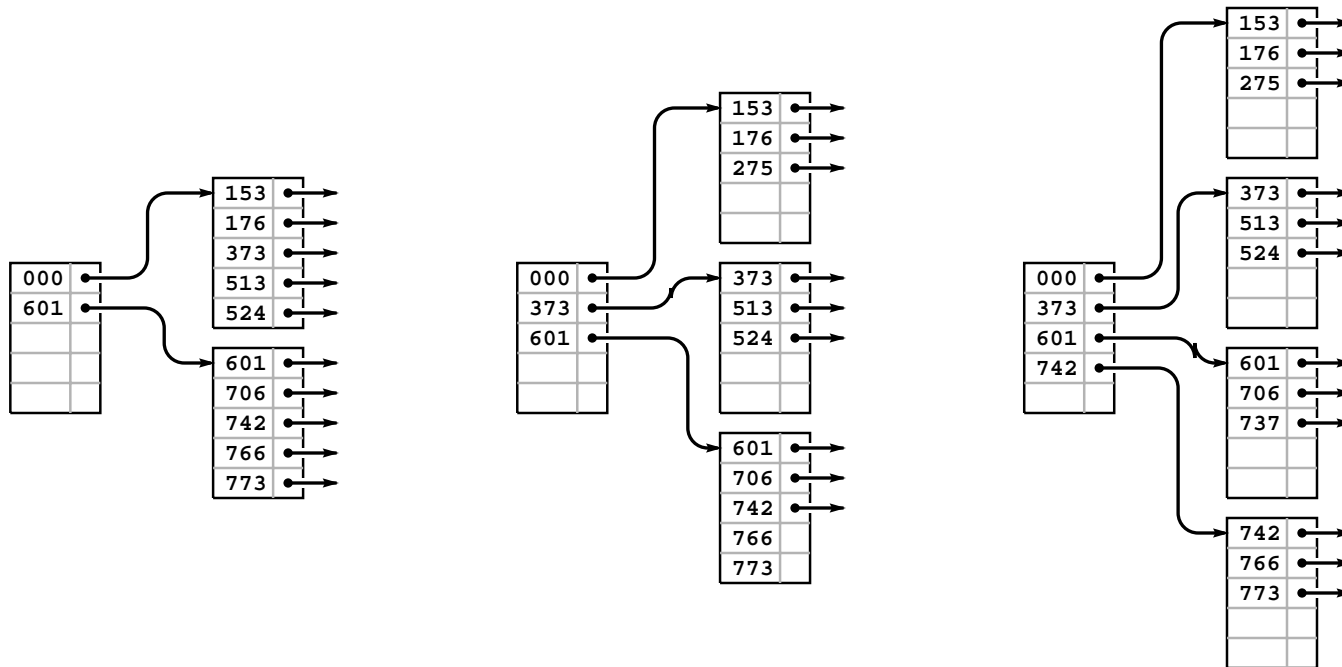
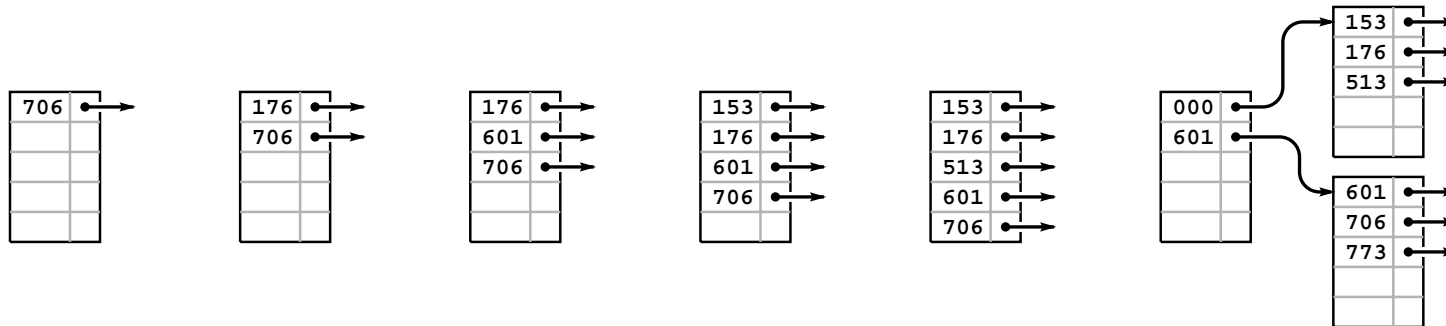
Space-time tradeoff

- M large: only a few levels in tree
- M small: less wasted space

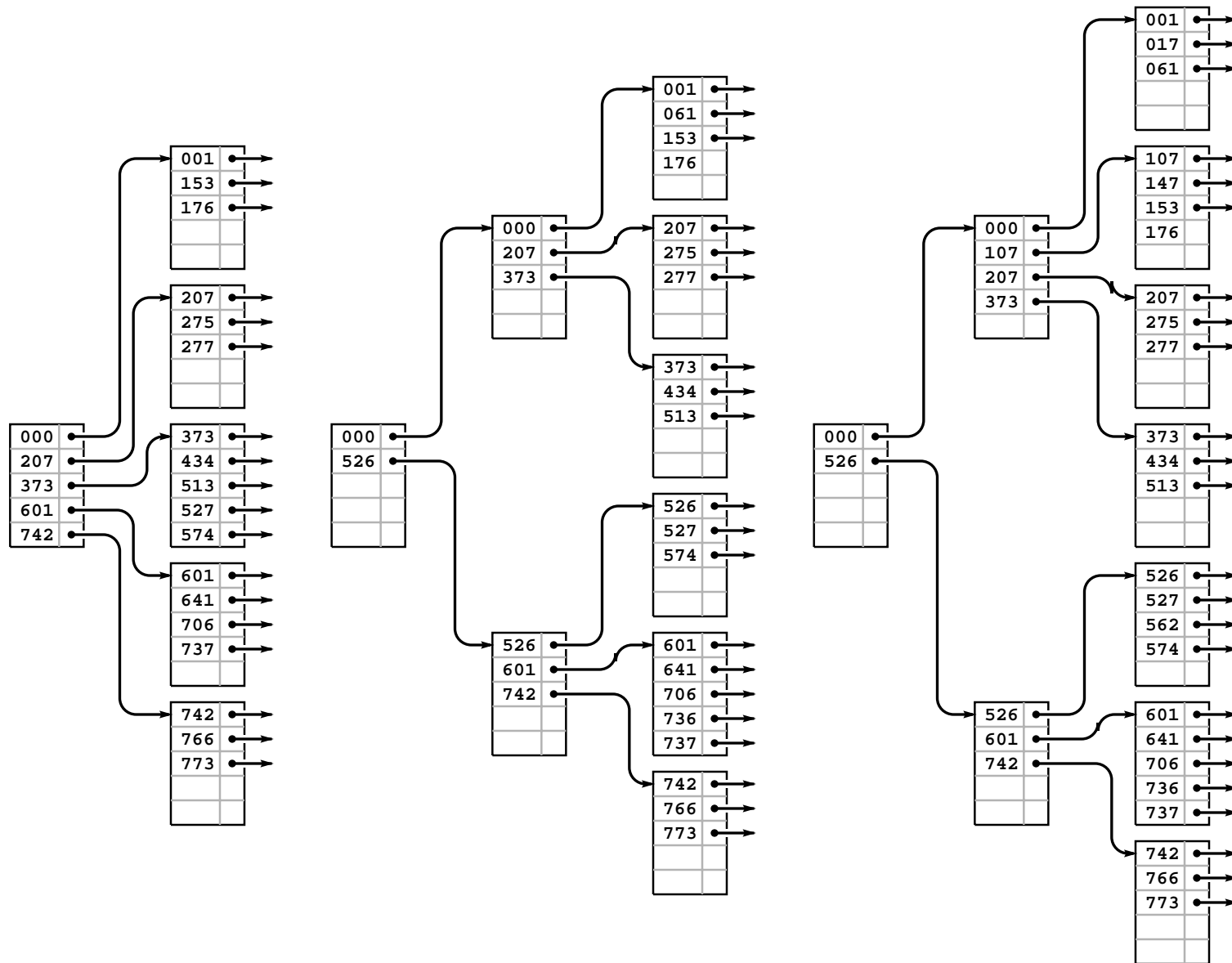
Bottom line:

- $\log_M N$ page accesses (3 or 4 in practice)

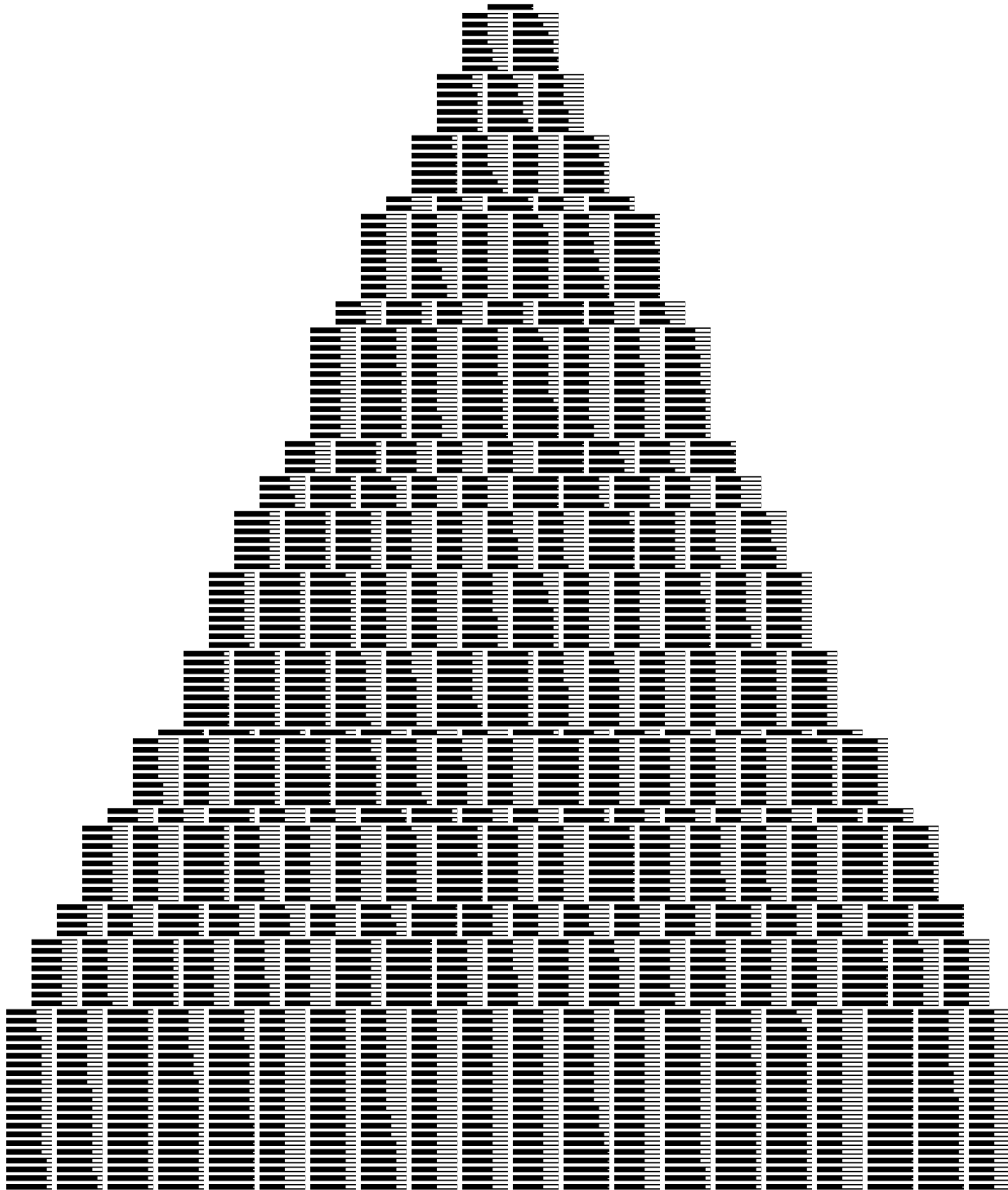
B tree example



B tree example (continued)



B tree growth



Skip Lists

Generalize skip lists

- nodes hold M records
- use sequential search within nodes

INSERT algorithm

- node not full: add record to node
- node full:
 - split into two half-full nodes
 - give new node t links
 - with probability $1/M^t$

Bottom line:

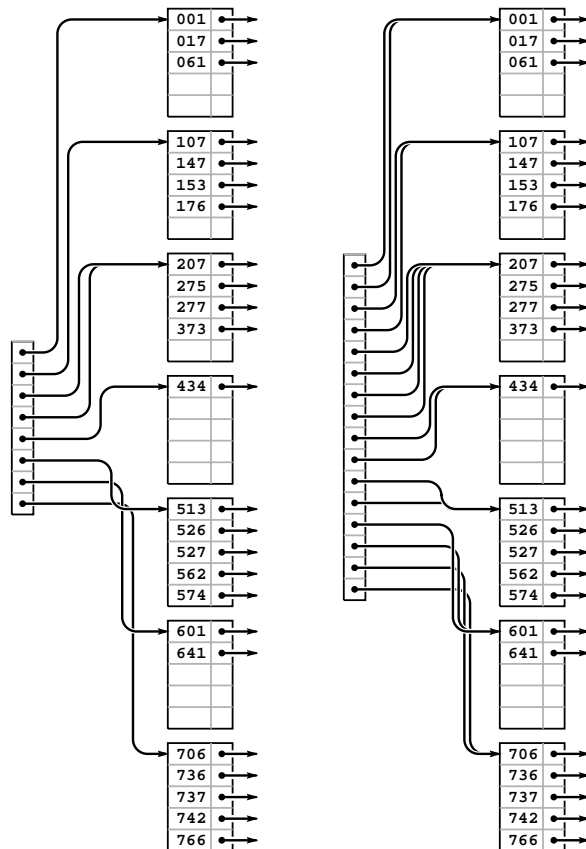
- $\log_M N$ page accesses (3 or 4 in practice)
- randomized algorithm OK?

Key-indexed directory

Use leading t bits of key to find page with record

CAPACITY: $M * 2^t$

To increase capacity, double directory size



Problems when $>M$ keys have same leading bits

Ex: $M+1$ equal keys

Extendible hashing

Generalize trie, hash search

- Hash keys to make them random bits
- Build two-level trie on hashed keys
 - directory size 2^t
 - page size M

SEARCH

- i = first t bits of hashed key
- directory entry i gives page name
- search that page for the key

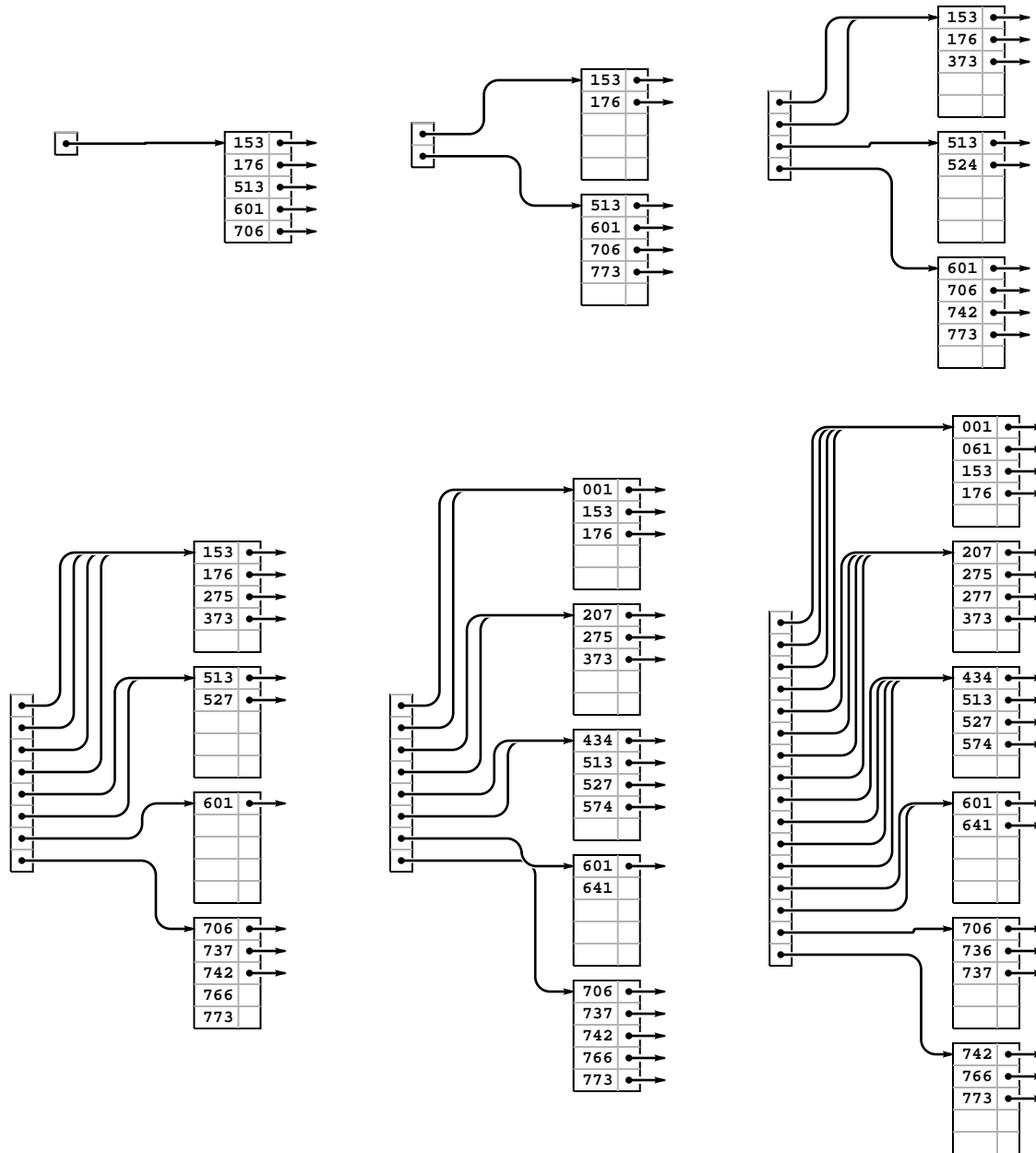
INSERT

- same as search
- put key on page accessed
- if page full, split in two (MAY NEED TO SPLIT DIRECTORY)

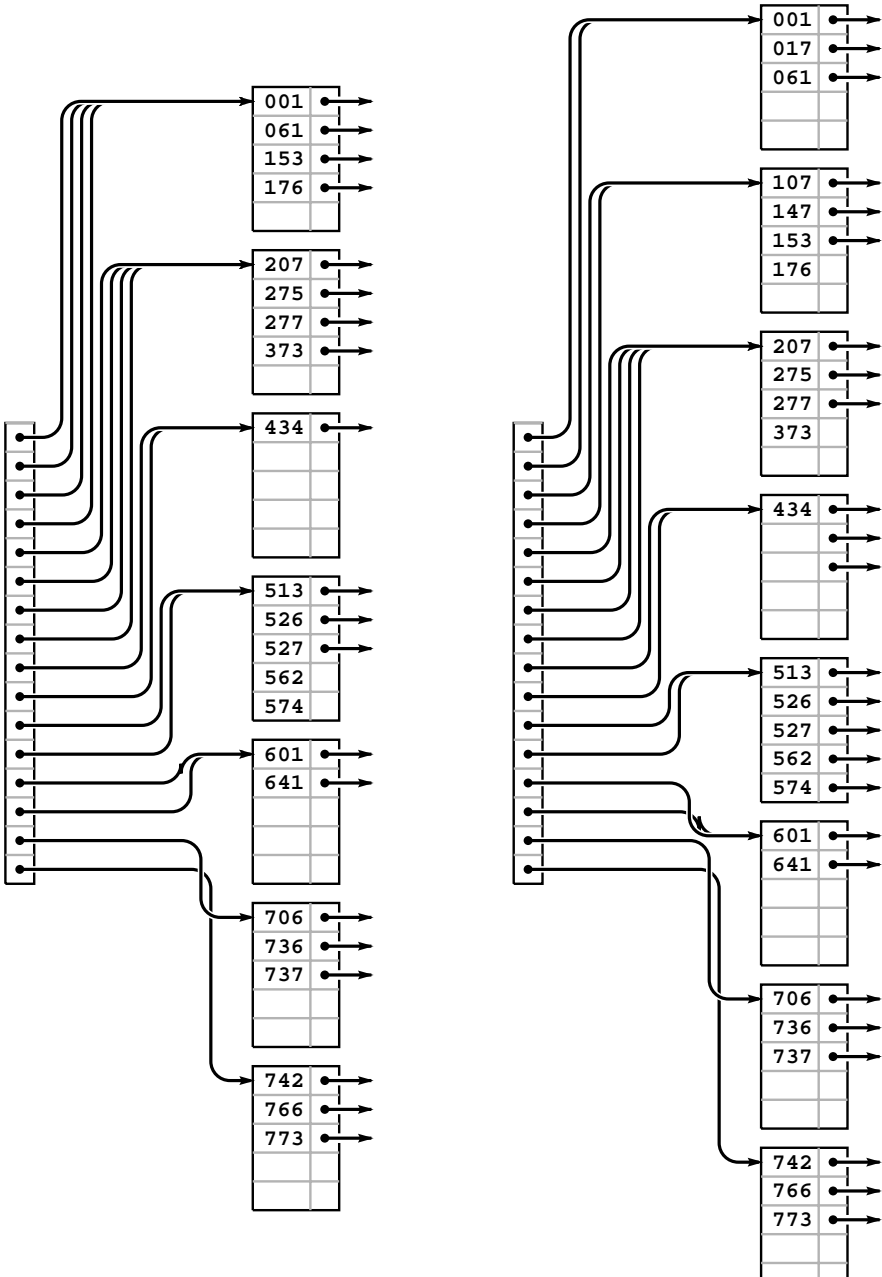
Bottom line:

- 2 page accesses per search, 44 per cent extra space
- randomized algorithm OK?

Extendible hashing example



Extendible hashing example (continued)



Extendible hashing initialization

nodes have M items, directory has D entries

```
typedef struct STnode* link;
struct STnode { Item b[M]; int m; int k; };
static link *dir;
static int d, D, N;
link NEW()
{ link x = malloc(sizeof *x);
  x->m = 0; x->k = 0;
  return x;
}
void STinit(int maxN)
{
  d = 0; N = 0; D = 1;
  dir = malloc(D*(sizeof *dir));
  dir[0] = NEW();
}
```

Extendible hashing search

Use leading d bits of key to index directory
Sequential search on page containing key
(could use binary search)

```
Item search(link h, Key v)
{ int j;
  for (j = 0; j < h->m; j++)
    if (eq(v, key(h->b[j])))
      return h->b[j];
  return NULLitem;
}
Item STsearch(Key v)
{ return search(dir[bits(v, 0, d)], v); }
```

Extendible hashing insertion

Use leading d bits of key to index directory

Insert key ala insertion sort (could live with unordered)

If page full, SPLIT

```
void insert(link h, Item item)
{ int i, j; Key v = key(item);
  for (j = 0; j < h->m; j++)
    if (less(v, key(h->b[j]))) break;
  for (i = (h->m)++; i > j; i--)
    h->b[i] = h->b[i-1];
  h->b[j] = item;
  if (h->m == M) split(h);
}
void STinsert(Item item)
{ insert(dir[bits(key(item), 0, d)], item); }
```

Code simplified by leaving space for key causing split

Extendible hashing node split

Get memory for a new node

Distribute os to old node, is to new node

- tricky point: skip bit if same for all records

Insert new node into directory

```
link split(link h)
{
    int j; link t = NEW();
    while (h->m == M)
    {
        h->m = 0; t->m = 0;
        for (j = 0; j < M; j++)
            if (bits(h->b[j], h->k, 1) == 0)
                h->b[(h->m)++] = h->b[j];
            else t->b[(t->m)++] = h->b[j];
        t->k = ++(h->k);
        if (t->m == M) h = t;
    }
    insertDIR(t, t->k);
}
```

k field: number of bits needed to distinguish keys

Extendible hashing directory split

Double directory

Copy pointers

Stay in loop until k-bit index supported

Insert new pointers

```
void insertDIR(link t, int k)
{ int i, m, x = bits(t->b[0], 0, k);
  while (d < k)
  { link *old = dir;
    d += 1; D += D;
    dir = malloc(D*(sizeof *dir));
    for (i = 0; i < D; i++) dir[i] = old[i/2];
    if (d < k) dir(bits(x, 0, d) ^ 1) = NEW();
  }
  for (m = 1; k < d; k++) m *= 2;
  for (i = 0; i < m; i++) dir[x*m+i] = t;
}
```