

CSCI 362 / V522: Data Structure

Fall, 2005

Part II: File Structures

Chap. 4 Fundamental File Structure Concepts

4.1 File Structure Concepts

- Building file structure → Make data persistent
 - One program can create data in memory and store it in a file and another program can read the file and re-create the data in its memory
- Field and record
 - field: the basic unit of data containing a single data value. It is the smallest logically meaningful unit of information in a file
 - record: an aggregate (list) of different data fields
 - record & fields in files correspond to objects and members in main memory

4.1 File Structure Concepts

– Stream file

- A stream of bytes
- Structureless: it does not maintain the meanings of the data fields

Mary Ames

Alan Mason

123 Maple

90 Eastgate

Stillwater, OK 74075

Ada, OK 74820

AmesMary123 MapleStillwaterOK74075MasonAlan90 EastgateAdaOK74820

– Field structures

- Fix the lengths of fields: allowing the reconstruction of the fields based on byte counting.

Ames Mary 123 Maple Stillwater OK 74075

Mason Alan 90 Eastgate Ada OK 74839

- Begin each field with a length indicator: length based fields.

04Ames**04**Mary**09**123 Maple**10**Stillwater**02**OK**05**74075

05Mason**04**Alan**11**90 Eastgate**03**Ada**02**OK**05**74839

- Separate the fields with delimiters: using a special character(s) that will not appear within a field to separate data fields.

Ames | Mary | 123 Maple | Stillwater | OK74075 |

Mason | Alan | 90 Eastgate | Ada | OK74839 |

- Use a “*keyword = value*” expression to identify fields: the field keyword provides information about itself. (Fig. 4.3)

last=Ames | first=Mary | address=123 Maple | city=Stillwater |
state=OK | zip=74075 |

– Reading a Stream of Fields

```
//person.cpp#include
<iostream.h>#include
<string.h>#include
"person.h"istream & operator >> (istream & stream, Person & p)
{ // read fields from input

stream.getline (p.LastName, 30);
if (strlen(p.LastName)==0) return stream;
    stream.getline(p.FirstName,30);
stream.getline(p.Address,30);
stream.getline(p.City, 30);
stream.getline(p.State,15);
stream.getline(p.ZipCode,10);
return stream;
}
```

– Record structures

- Make records a predictable number of bytes. (Fig. 4.5)

Ames	Mary	123 Maple	Stillwater	OK74075
Mason	Alan	90 Eastgate	Ada	OK74839

Ames | Mary | 123 Maple | Stillwater | OK74075 |-----
Mason | Alan | 90 Eastgate | Ada | OK74839 |-----

- Predictable number of fields.


Ames		Mary		123 Maple		Stillwater		OK74075	
Mason		Alan		90 Eastgate		Ada		OK74839	

- Begin each record with a length indicator
good for variable-length records.

40 Ames | Mary | 123 Maple | Stillwater | OK74075 |
36 Mason | Alan | 90 Eastgate | Ada | OK74839

- Record index addressing
indexing the records based on their byte offsets.

Ames|Mary|123Maple|Stillwater|OK74075|Mason|Alan|90



Index file 00 40

- Record delimiters: separating records by special character(s) (*e.g.* line-breaks). (Fig. 4.6)

– Variable-length records

- Writing variable-length records to file:
we must know the sum of the length of the fields in each record before writing — buffering.
- Reading variable-length records from file:
read the length of the record, move it into a buffer, and break it into fields.

```
// writestr.cpp// write a stream of persons, using fstream.h
#include <fstream.h>
#include <string.h>
#include "readper.cpp"
const int MaxBufferSize = 200;
int WritePerson (ostream & stream, Person & p)
{
    char buffer [MaxBufferSize];
    strcpy(buffer, p.LastName); strcat(buffer, "|");
    strcat(buffer, p.FirstName); strcat(buffer, "|");
    strcat(buffer, p.Address); strcat(buffer, "|");
    strcat(buffer, p.City); strcat(buffer, "|");
    strcat(buffer, p.State); strcat(buffer, "|");
    strcat(buffer, p.ZipCode); strcat(buffer, "|");
    short length=strlen(buffer);
    stream.write ((char *)&length, sizeof(length));
// write length
    stream.write (&buffer, length);}

```

```
int main ()
{
    char filename [20];
    Person p;
    cout << "Enter the file name:"<<flush;
    cin.getline(filename, 19);
    ofstream stream (filename, ios::out);
    if (stream.fail()) {
        cout << "File open failed!" <<endl;
        return 0;
    }
    while (1) {
        // read fields of person
        cin >> p;
        if (strlen(p.LastName)==0) break;
        // write person to output stream
        WritePerson(stream,p);
    }
}
```


Chap. 5 Managing Files of Records

5.1 Record Access (Search)

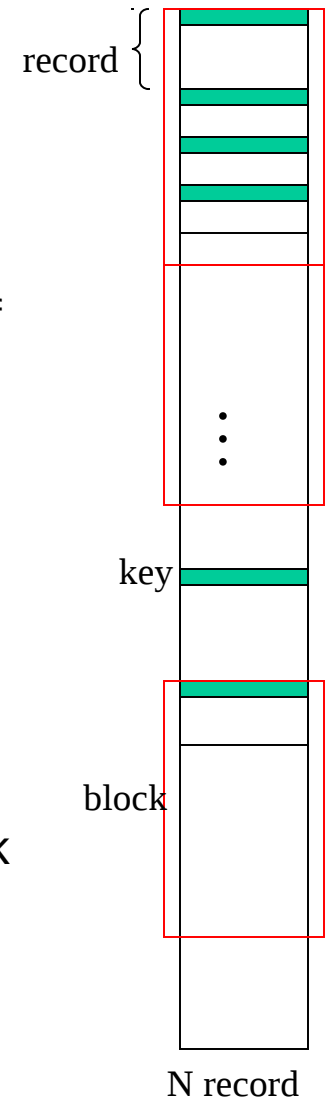
Identify a record with a key based on record's content

– Record Keys

- Primary keys: the primary search key, often not part of the data field.
- Secondary keys: additional search key, often part of the data field.
- Canonical form of keys: a unique representation of keys (*e.g.* All capital letters for string keys).

– Sequential Search

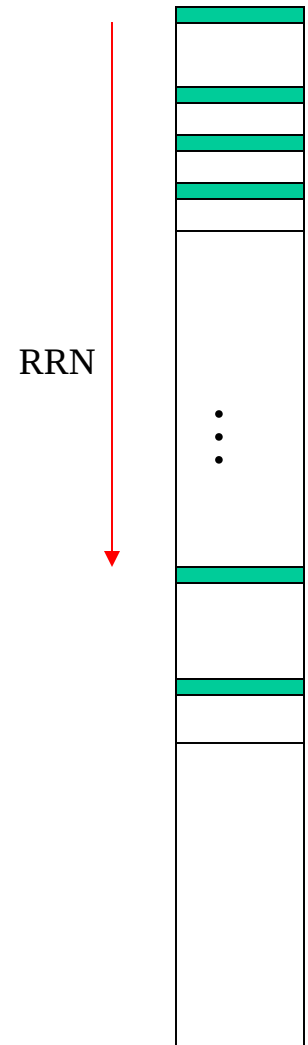
- Worst case: N ; average case: $N/2$
- Record blocking: reducing “seek” time, reading a block at a time and search within a block in memory.
- **Block**: a fixed sized disk space containing a group of records.



- Sequential search is good for: pattern search, short files, and secondary key search.
- Unix tools for sequential search/processing: cat, wc, grep,...

– Direct Access

- Directly seek to the beginning of the record to be accessed.
- Relative Record Number (RRN): a record's position (in term of number of records) relative to the beginning of the file.
- Byte offset: $\text{fixed_record_size} * \text{RRN}$

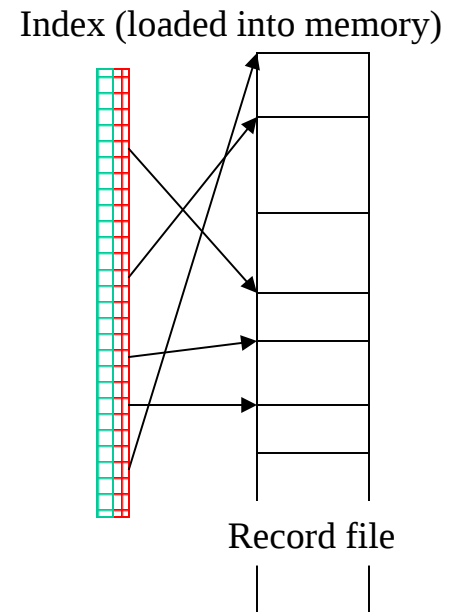


– 5.5 Beyond Record Structures

- Sound, Image, and Document
- Headers and Self-describing Files
 - A name for each field
 - The width of each field
 - Then number of fields per record
- Metadata: describing the primary data in a file
- Image file: TIFF, RAW, SunRaster, ...
 - Compressed image: jpg, gif,

Chap. 7 Indexing

- Index: an array of **keys** and reference fields
 - Impose order on a file without rearranging the file
 - Give us a keyed access to variable-length record files
- A simple linear index for entry-sequenced files
 - Primary key based indexing:
key → **byte offset** (Fig. 7.3)
 - Index file is normally sorted and kept in memory
 - allowing binary search
 - One record per entry in the index file
 - Record file is not sorted.



Example of recordings

Identification number
Title
Composer
Artist or artists
Label(publisher)

Record address	Label	ID number	Title	Composer(s)	Artist(s)
17	LON	2312	Romeo and Juliet	Prokofiev	Maazel
62	RCA	2626	Quartet in C Sharp Minor	Beethoven	Julliard
117	WAR	23699	Touchstone	Corea	Corea
152	ANG	3795	Symphony No. 9	Beethoven	Giulini
196	COL	38358	Nebraska	Springsteen	Springsteen
241	DG	18807	Symphony No. 9	Beethoven	Karajan
285	MER	75016	Coq d'Or Suite	Rimsky-Korsakov	Leinsdorf
338	COL	31809	Symphony No. 9	Dvorak	Bernstein
382	DG	139201	Violin Concerto	Beethoven	Ferras
427	FF	245	Good News	Sweet Honey in the Rock	Sweet Honey in the Rock

Index		Recording file	
Label ID	Key	Reference field	Address of record
ANG3795	152	17	LON 2312 Romeo and Juliet Prokofiev ...
COL31809	338	62	RCA 2626 Quartet in C Sharp Minor Beethoven ...
COL38358	196	117	WAR 23699 Touchstone Corea ...
DG139201	382	152	ANG 3795 Symphony No. 9 Beethoven ...
DG18807	241	196	COL 38358 Nebraska Springsteen ...
FF245	427	241	DG 18807 Symphony No. 9 Beethoven ...
LON2312	17	285	MER 75016 Coq d'Or Suite Rimsky-Korsakov ...
MER75016	285	338	COL 31809 Symphony No. 9 Dvorak ...
RCA2626	62	382	DG 139201 Violin Concerto Beethoven ...
WAR23699	117	427	FF 245 Good News Sweet Honey in the Rock ...

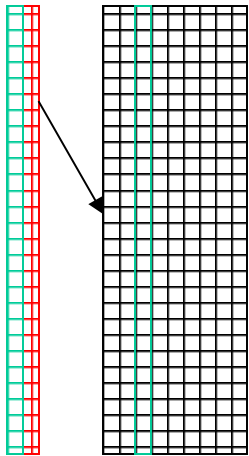
– Indexing Operations (programs provided in Appendix G)

- Creating files: initially empty
- Loading the index into memory
- Rewrite the index file from memory back to disk.

Status flag needs to be defined and set for each index file to indicate out-of-date index file. When index file rewriting fails, it can automatically (by checking the status flag) rebuild the index file from the data record file.

- Record addition: adding record to the end of the record file, and inserting its index entry to the index file (done in memory).
- Record deletion: lazy deletion from the record file (the storage may be reused later) and remove its index entry from the index file.

- Record updating



- a.) Changing part of the key field:

- this involves reordering of the index file, by first deleting the index and then insert the new index

- b.) No change to the key field: index order doesn't change, but the data record may require rearrangement (*e.g.* size of the record changes).

- ⇒ delete/insert approach for record only, and updating the byte offset of its index.

- Large index file

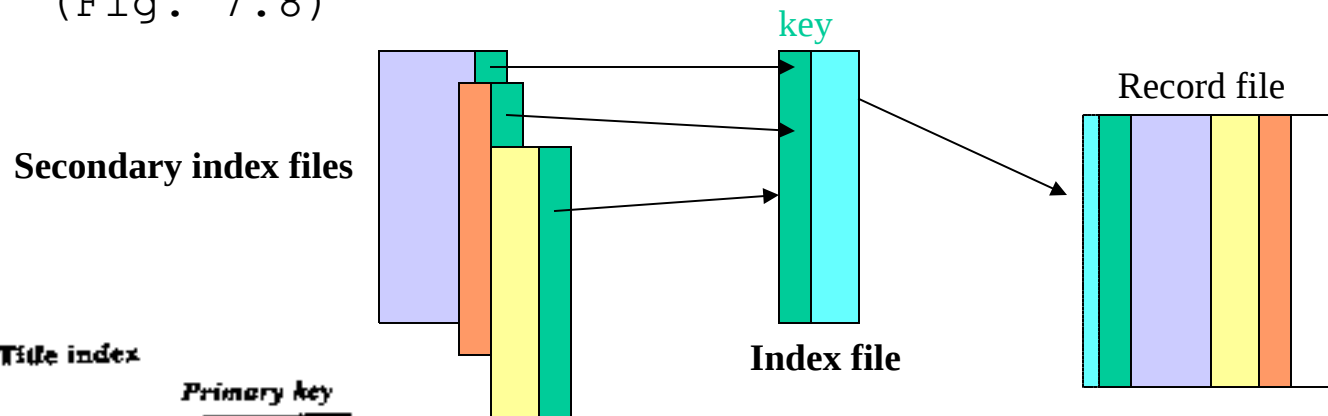
- index file is too large to be kept in memory

- ⇒ Secondary storage index file

- index searching is not significantly faster than direct record searching, though it is still good for variable-length records.
 - Index maintenance is more difficult:
insertion, deletion & updating are expensive
 - Solution: Multilevel indexing & B-tree

– Multiple key indexing

- Content-based searching: using secondary keys for indexing
- Secondary key index file: indexing to the primary keys only.
(Fig. 7.8)



Title index

Secondary key	Primary key
COQ D'OR SUITE	MER75016
GOOD NEWS	FF245
NEBRASKA	COL38358
QUARTET IN C SHARP M	RCA2626
ROMEO AND JULIET	LON2312
SYMPHONY NO. 9	ANG3795
SYMPHONY NO. 9	COL31809
SYMPHONY NO. 9	DG18807
TOUCHSTONE	WAR23699
VIOLIN CONCERTO	DG139201

Index

Key	Reference field	Address of record
ANG3795	152	17
COL31809	338	62
COL38358	196	117
DG139201	382	152
DG18807	241	196
FF245	427	241
LON2312	17	285
MER75016	285	338
RCA2626	62	382
WAR23699	117	427

Recording file

Actual data record
LON 2312 Romeo and Juliet Prokofiev ...
RCA 2626 Quartet in C Sharp Minor Beethoven ...
WAR 23699 Touchstone Corea ...
ANG 3795 Symphony No. 9 Beethoven ...
COL 38358 Nebraska Springsteen ...
DG 18807 Symphony No. 9 Beethoven ...
MER 75016 Coq d'Or Suite Rimsky-Korsakov ...
COL 31809 Symphony No. 9 Dvorak ...
DG 139201 Violin Concerto Beethoven ...
FF 245 Good News Sweet Honey in the Rock ...

- Record addition: inserting entries for both primary & secondary index files. (Note: secondary keys can be duplicated)
- Record deletion: delete key entry in primary and secondary indexes
 - pinned problem
 - delete Primary Index only
- Record updating
 - a.) update changes the secondary key:
rearrangement of secondary index may be necessary
 - b.) update changes the primary key:
need to search the secondary index file, and update all related entries
 - c.) update changes other fields only:
no update necessary in secondary key index file.
- Retrieval using combinations of secondary keys
 - Searching based on boolean combinations of secondary keys.
example: composer = 'BEETHOVEN' AND title = 'SYMPHONY No. 9'
 - Searching for each key individually, and compute the combined result using boolean set operations of the resulting primary keys

- secondary key should be stored in sorted order (by primary keys) to facilitate *boolean* set operations

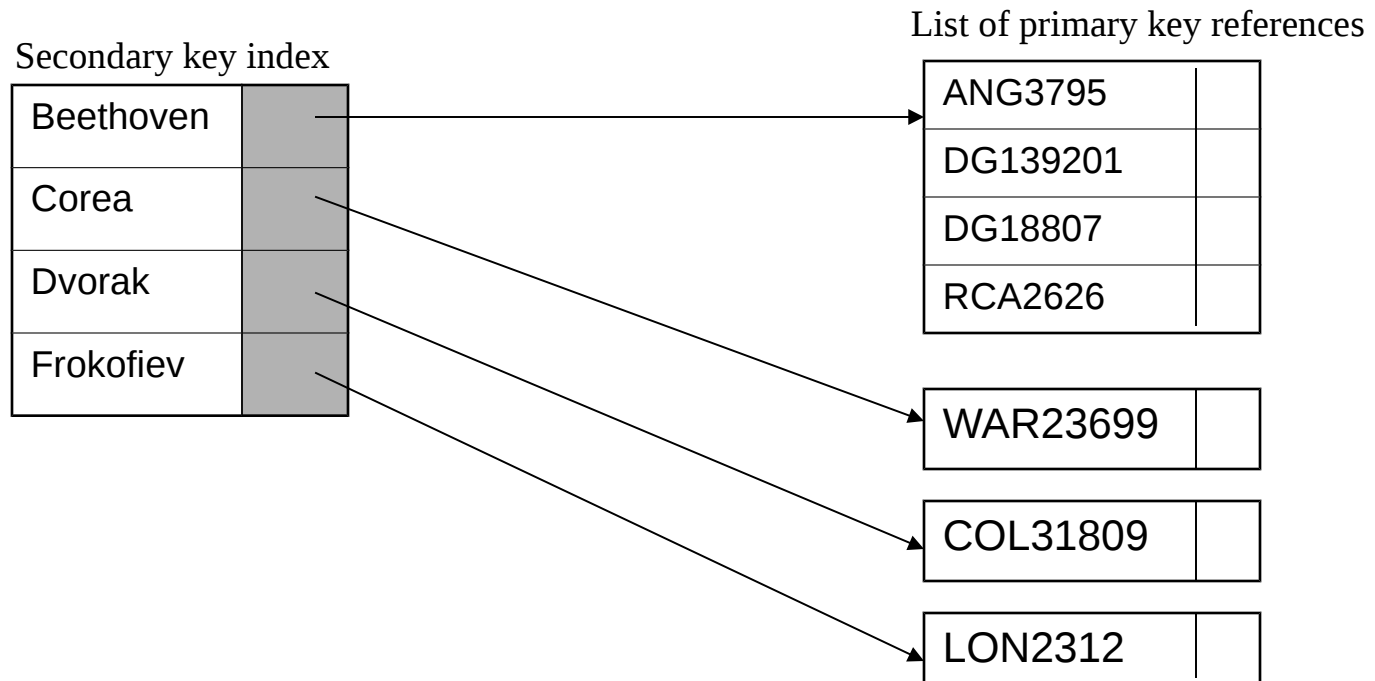
Composers	Titles	Matched list
ANG3795 -----	ANG3795 -----	ANG3795
DG139201	COL31809	----- DG18807
DG18807 -----	DG18807 -----	
RCA2626		

– Inverted Lists

- Improving the secondary index structure: Avoid generating duplicate secondary index entries: using a linked list for each secondary key, containing the primary keys of all records with the same secondary key. (Fig. 7.12 - 7.13)

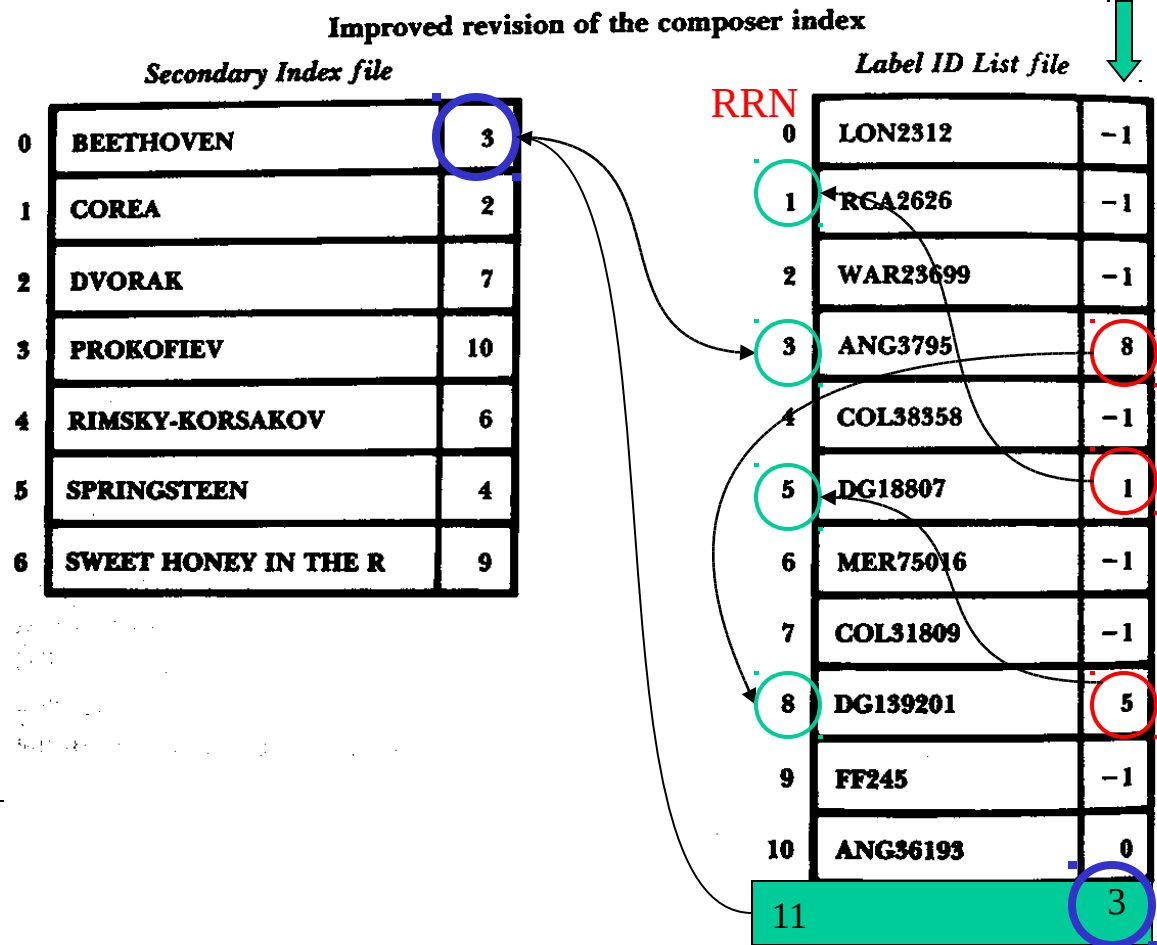
BEETHOVEN	ANG3795	DG139201	DG18807	RCA2626
COREA	WAR23699			
DVORAK	COL31809			
FROKOFIEV	LON2312			
RIMSKY-KORSAKOV	MER75016			
SPRINGSTEEN	COL3835			
SWEET HONEY IN ...	FF245			

- Invert list: a list of primary key references
 - a.) the secondary index file contains only the headers of the linked lists associated with the secondary keys
 - b.) a separate Key File (entry-sequenced) is used to store the contents and pointers (the RRNs) of the linked lists.



- The secondary key index file only needs to be rearranged when a new secondary key is generated
- Adding & deleting records only affect the Key File.
- Secondary key index file is smaller & easier to store & maintain.
- Records with the same secondary key are not stored together (no locality – need a lot of seeking)

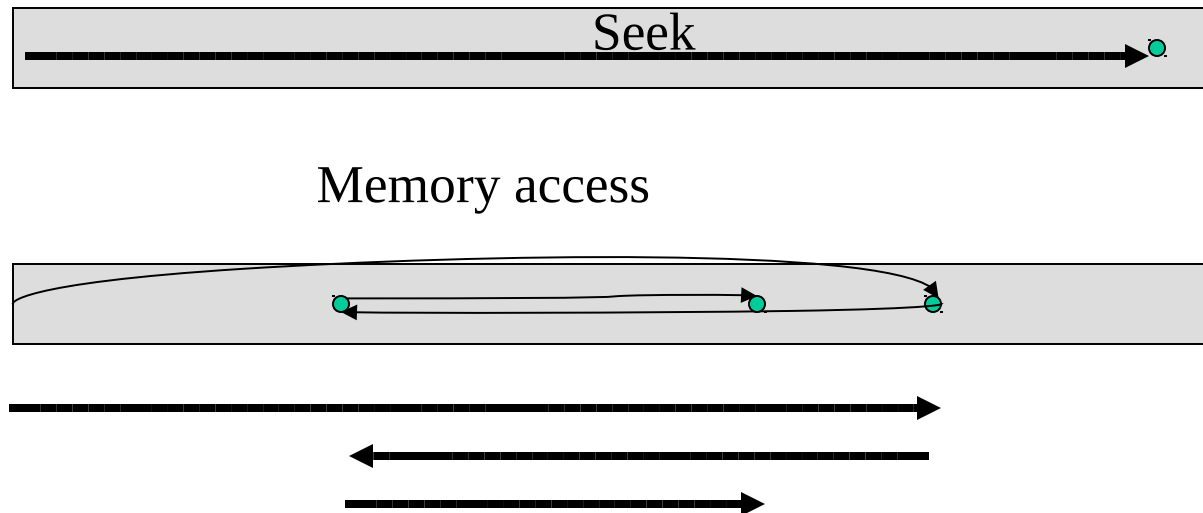
– Linking the List of References



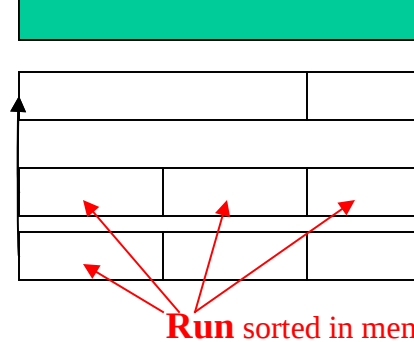
8. Sorting of Large Files

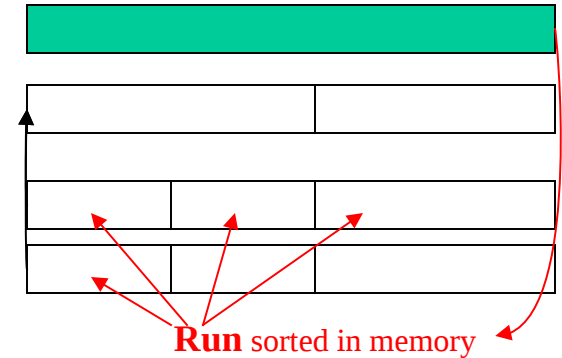
– Disk-access vs. memory access

- each disk access worth $\sim 200,000$ machine instructions.
- reduce the number of disk access to a very small constant, using complicated data structures & algorithms (since machine instructions are virtually free compared to disk-access)



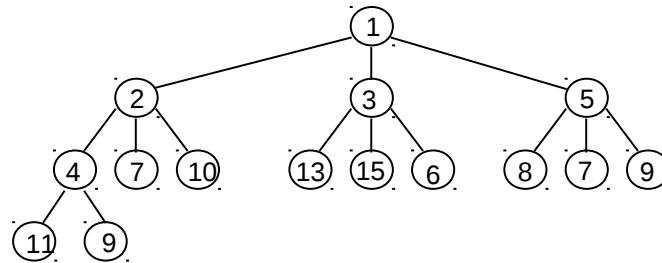
7.11 External Sorting

- Internal Sorting: random access
 - External storage: sequential access
 - Not directly addressable (*e.g.* tape)
 - Slow
 - External sorting is device dependent.
 - External sorting with tapes (A simple merge sort)
 - 1) Four tapes are used: T_{a1} , T_{a2} , T_{b1} , T_{b2} ,
Initial input is on T_{a1}
 - 2) The internal memory can hold and sort M elements at a time. The result of each internal sorting (of M elements) is called a run. Each run is stored on tape T_{b1} or T_{b2} in an alternate fashion.
 - 3) Take a run from both T_{b1} & T_{b2} at a time, merge them and write the result (a run twice as long) to T_{a1} or T_{a2} alternately.
 - 4) Continue this merging process from T_{a1} & T_{a2} to T_{b1} & T_{b2} , until all are sorted.
 - Example: [pp. 290 - 291]
- 



6.5 *d*-Heaps

- An extension of binary heap — a *d*-ary tree structure

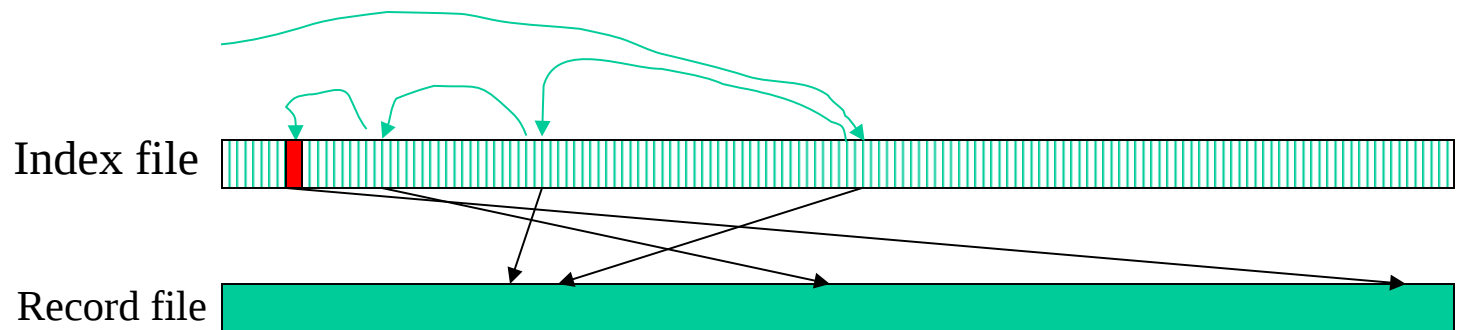


- *insert*: $O(\log_d N)$
- *deleteMin*: $O(d \log_d N) \rightarrow$ compare with d children
- Array implementation is not as efficient:
 - Multiplication & division, in general, cannot be carried out by bit shifting.
- Maybe used for disk storage (similar to *B*-trees)
- Merge operation is hard

Chap. 9 Multilevel Indexing & B-trees

– Sequential index file

- Large index file cannot be kept in main memory
 - search has to be done in secondary storage with sorted sequential index file
- Binary search is not efficient (too many seeks)
- insertion/deletion can involve massive movements of the index file.

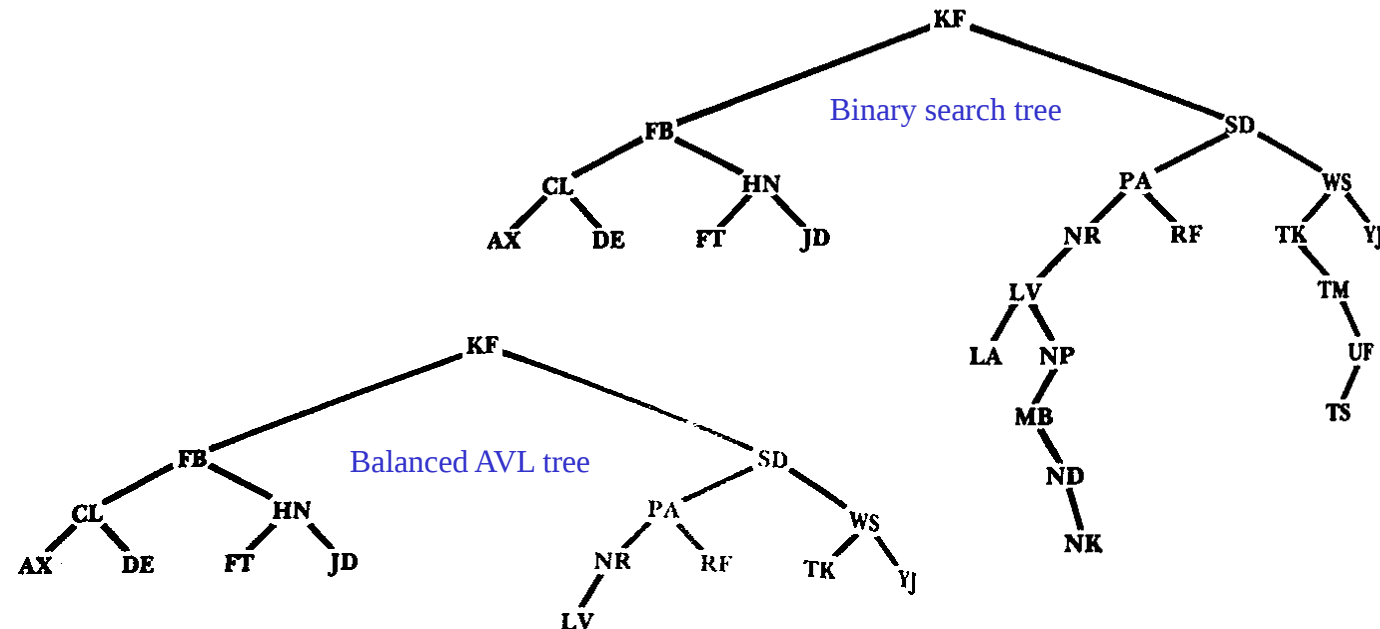
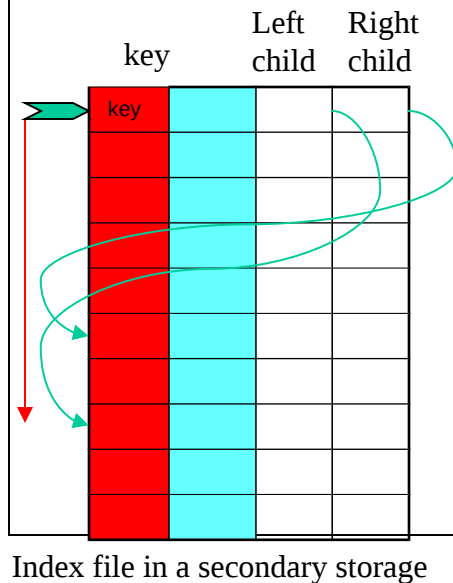


– Statement of the Problem

- Searching the index must be faster than binary searching
- Insertion and deletion must be as fast as search

– Indexing using Binary Search Tree

- Using binary search tree based file structure:
each index node contains the **key** value (with **reference address**) and its left & right child nodes (offsets in the index file)
- AVL & SPLAY tree can also be used to maintain balance.
- Search requires too many seeks, and insertion/deletion are expensive.
- Each reading (at least one page) generates very little useful information.



– Paged binary trees

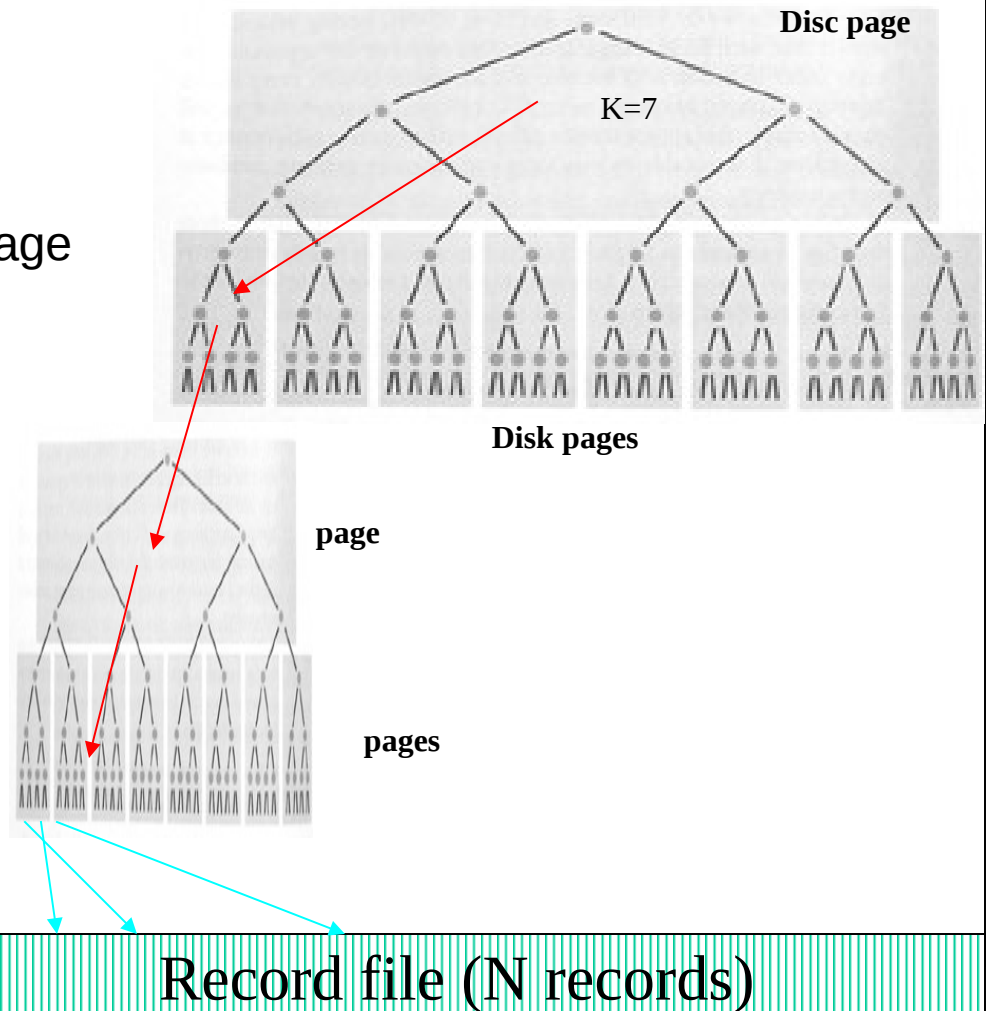
- A solution for more efficient use of disk reading: locating multiple nodes of the binary tree on the same disk page.
- Total number of seeks:

$$Height = \log_{(K+1)}(N+1)$$

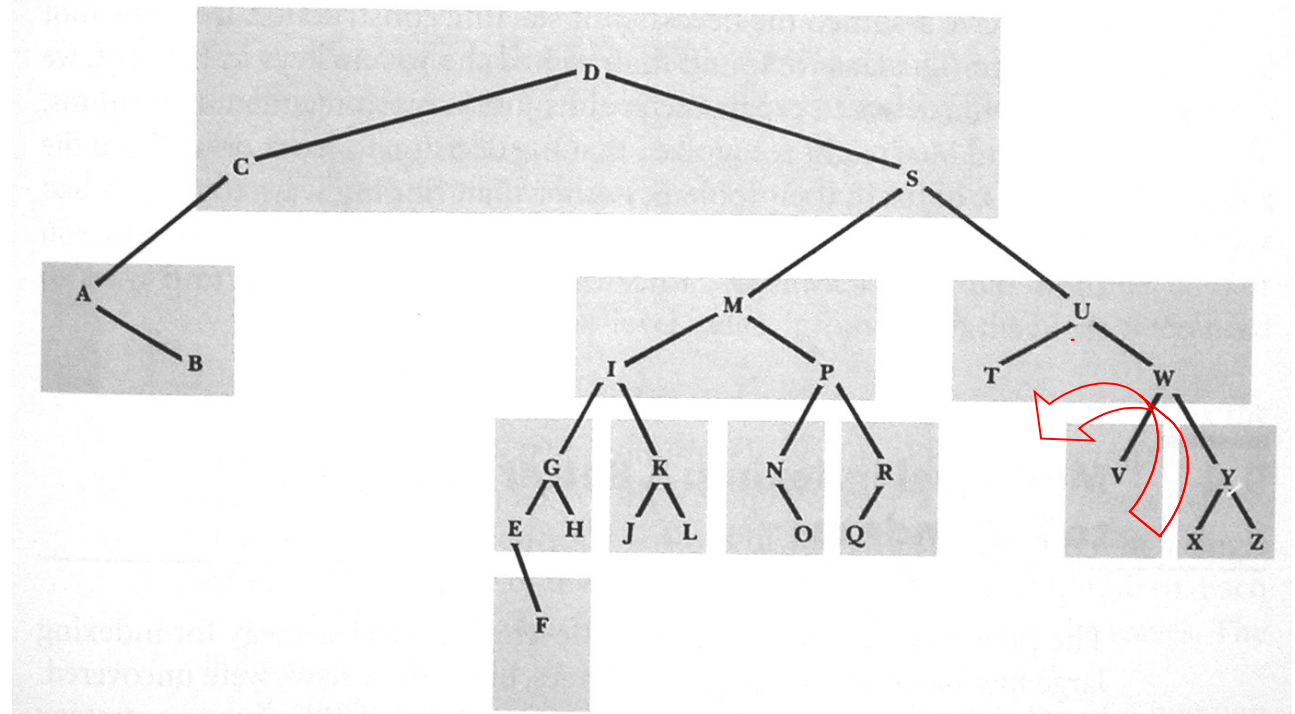
$$\text{Because } (k+1)^{height} = N$$

K : Number of nodes in a page

N : total number of keys



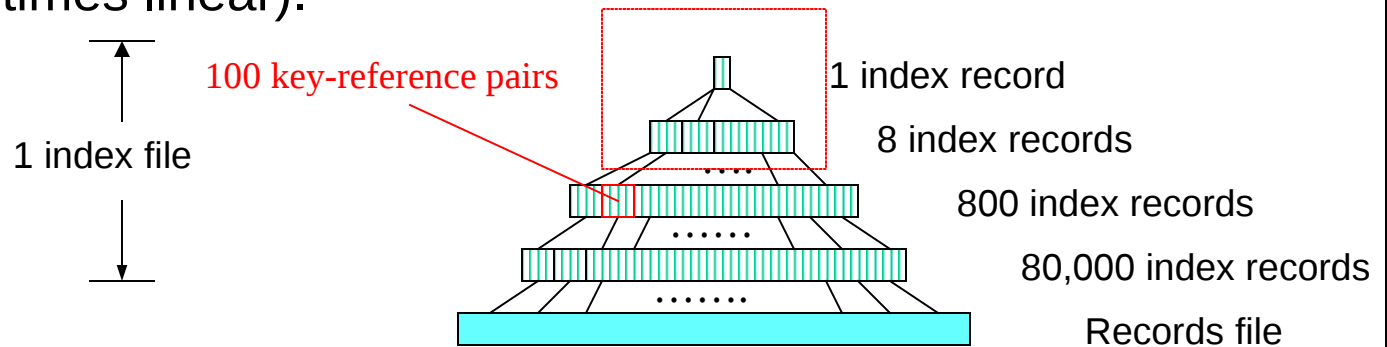
- A problem: the tree structure within a page is not efficient nor necessary.
- Another problem: building a balanced paged tree requires breaking the pages.
- The *3rd* problem: A large number of pages may not be sufficiently full. (Fig. 9.13)



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

– Multilevel indexing

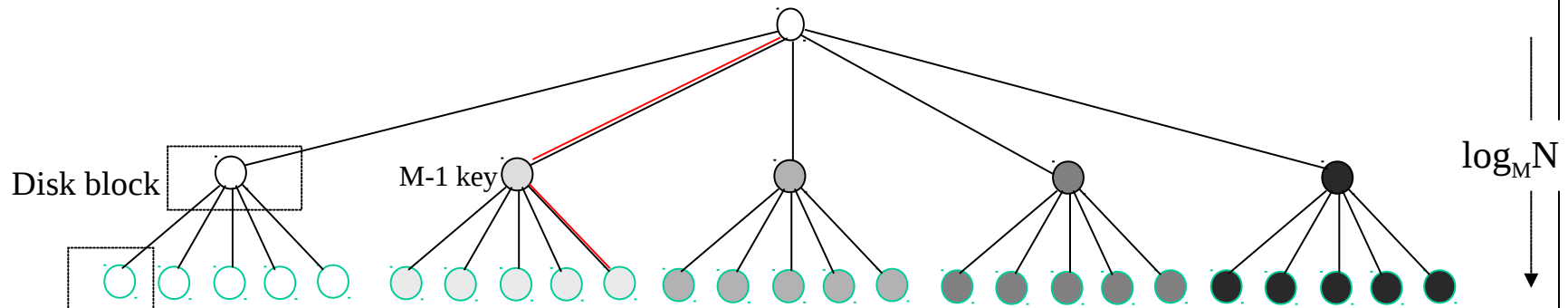
- A hierarchy of index records, the first level index is the index to the data records, the second level index is the index to the *1st* level index record, and so on ...
- An example:
8,000,000 data records with 100 bytes per record, 10 byte key, 100 key-reference pairs per index record → four disc access
- The index records in each level are sorted, and the last key of the record can be used as the key of the index record.
- The index file is entry-sequenced (unsorted).
- Insertion may lead to rearrangement of the index hierarchy (sometimes linear).



4.7 B-trees

- B-tree

- A B-tree of order M is an M -ary tree with conditions:
 - 1). data records are stored at leaves
 - 2). non-leaf nodes store up to $M-1$ keys to guide the searching;
key i represents the smallest key in subtree $i+1$
 - 3). the root is either a leaf or has between two and M children



- 4). all non-leaf nodes (except root) have between $\lceil M/2 \rceil$ and M children.
 - 5). If L is the number of leaf nodes in a block, all leaf nodes are at the same depth and have between $\lceil L/2 \rceil$ and L children.
- Each node represents a dish block, and is guaranteed to be at least half full, which prevents it from degenerating into a simple binary tree.
 - M & L are chosen according to data size, key size, and disk block size.

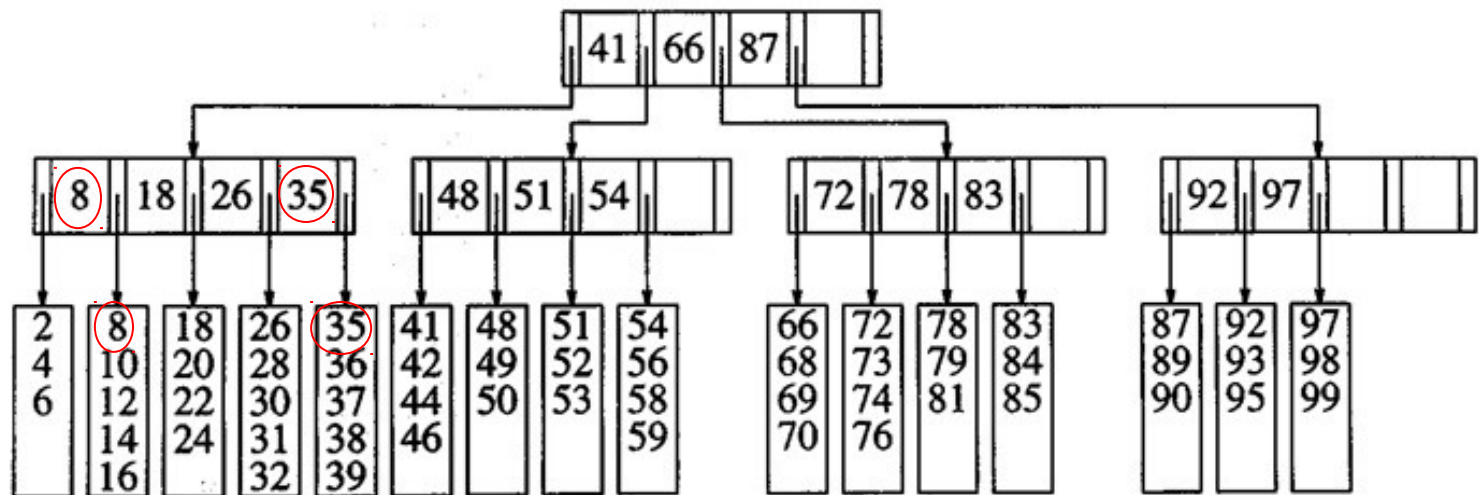


Figure 4.62 B-tree of order 5

10,000,000 data records

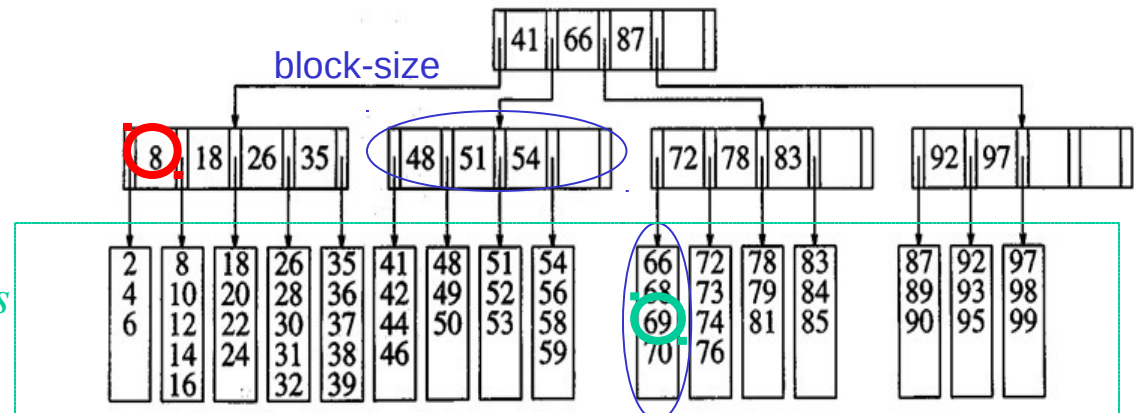


Figure 4.62 B-tree of order 5

A leaf node (block)

- Example:

block-size = 8192 bytes,

key-size = 32 bytes,

data-record-size = 256 bytes

each non-leaf node needs $32(M-1)+4M$ bytes (<8192).

⇒ the largest M is 228

Each data-record needs 256 bytes

⇒ the largest L is 32 ($8192/256$), Each block holds 16~32 records

If we have up to 10,000,000 data records, we can use at most 625,000 ($10,000,000/16$) leaves (blocks).

⇒ leaf nodes will have depth at most 4 (or, in general, worst case $\log_{M/2} N$)

If we store the root and the 1st level in main memory/cache, only 2 disk accesses are needed.

Best case:

2 levels indexing M nodes

3 levels indexing M^2 nodes

4 levels indexing M^3 nodes

Worst case:

2 levels indexing $M/2$

3 levels indexing $(M/2)^2$

4 levels indexing $(M/2)^3$

$(228/2)^3 = 1481,544 > 625,000$

– B-trees

- Each node is an index record.
The order of the index record is the maximum number of key-reference pairs the record can hold.
- Each index record is at least half-full.
- Overflow of a record (by insertion) leads to a split of the record into two (one-to-two split), each half full
- Deletion merging two records into a single record when necessary
- Insertion can lead to the change of keys for index records
 - bottom-up updating: build trees upward from the bottom instead of downward from the top, which involves a lot of adjustment
 - each insertion has one change and a new one created by a split,
→ linear insertion
- The root can have at least two key-reference pairs. (Fig.9.14-9.15)

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

C, S, D, T



A



M, P, I

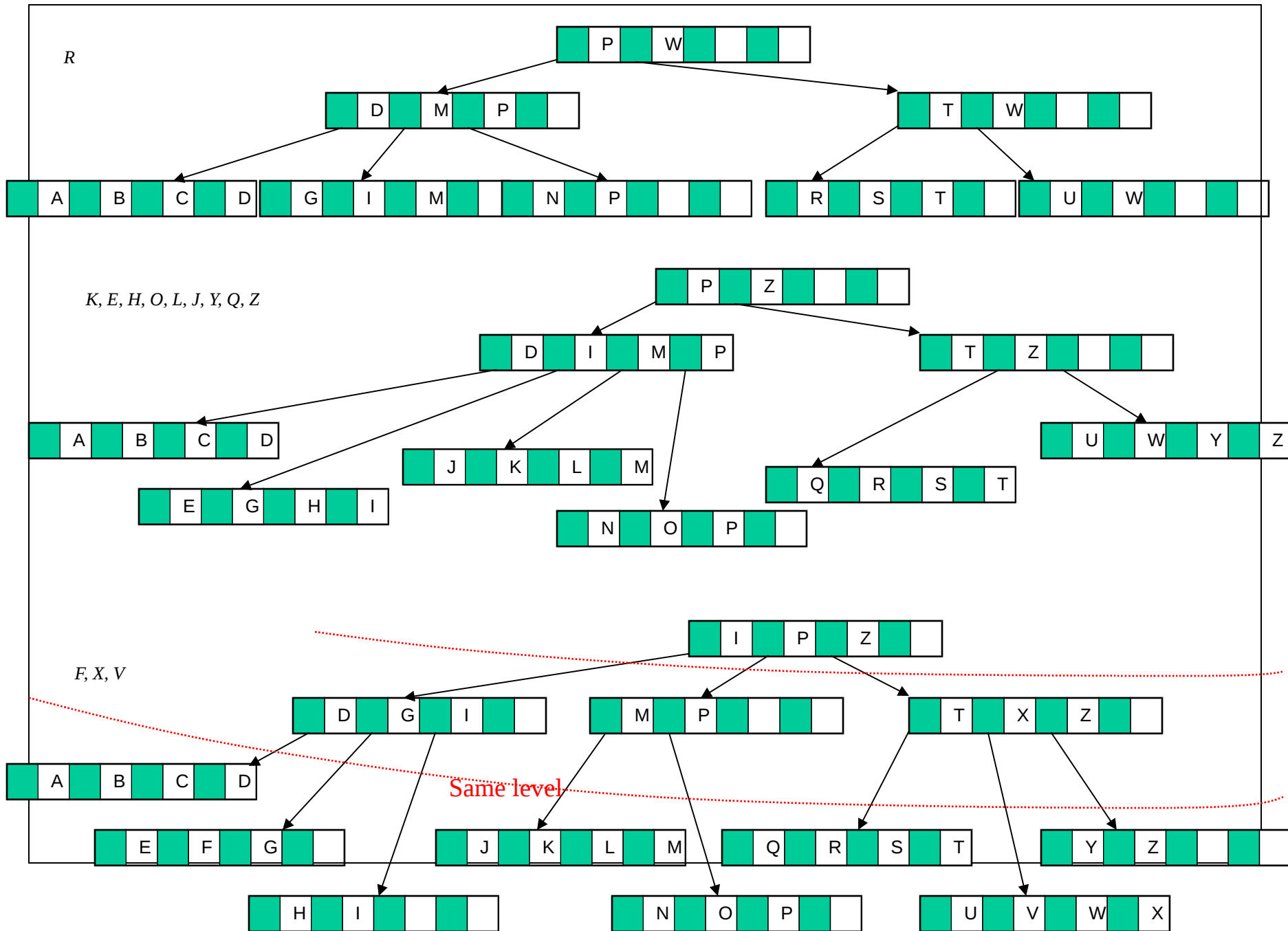


B, W, N, G



U





- B-tree (of order m) properties
 - 1.) every page has a maximum of m children
 - 2.) every page, except for the root and the leaves, has at least $\lceil m/2 \rceil$ children
 - 3.) the root has at least two children
 - 4.) all leaves appear on the same level
 - 5.) the leaf level forms a complete, ordered index of the data file.

– Simple B-tree operations

- Search:

```
int search(KeyType key, int recAddr)
{
    leafNode=findLeaf(key);
    return leafNode->Serach(key, recAddr);
}

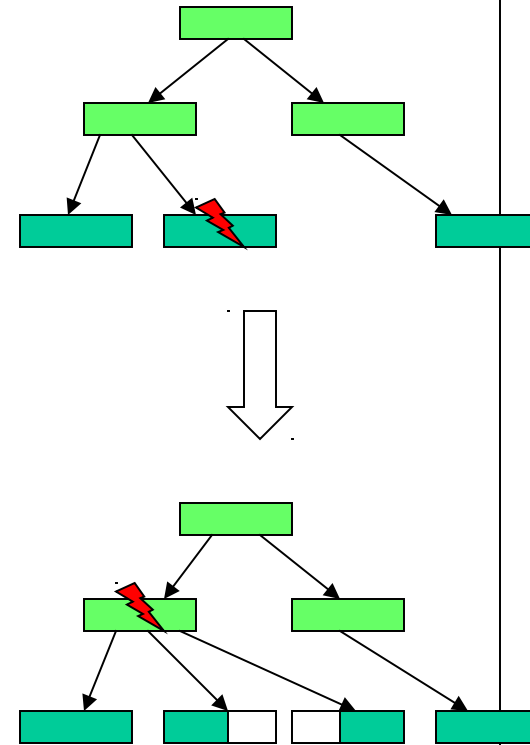
Node *findLeaf(KeyType key) {
    for (i=1; i<tree_depth; i++) {
        recAddr=Nodes[i-1]->search(key, -1);
        Nodes[i]=fetch(recAddr);
    }
    return Nodes[i-1]
}
```

Note: the leaf here is defined as the lowest index node.

- *insertion*

- 1.) search to the leaf level using `findLeaf`
- 2.) insertion, overflow, detection, and splitting on the upward path
- 3.) creation of a new root node, if the current root was split.

- *create, open and close* B-tree files



– worst-case search depth

- maximum depth occurs when each node has its minimum number of children

Level	# of children
1	2
2	$2 \times \lceil m/2 \rceil$
3	$2 \times \lceil m/2 \rceil^2$
\vdots	\vdots
d	$2 \times \lceil m/2 \rceil^{d-1}$

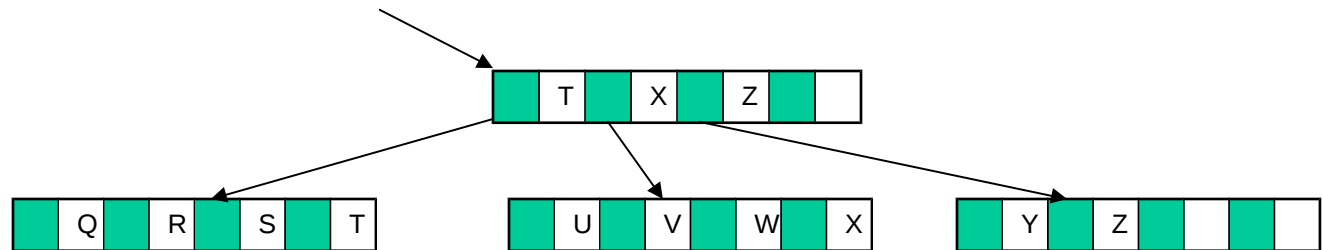
- For N keys in a m -order B-tree:

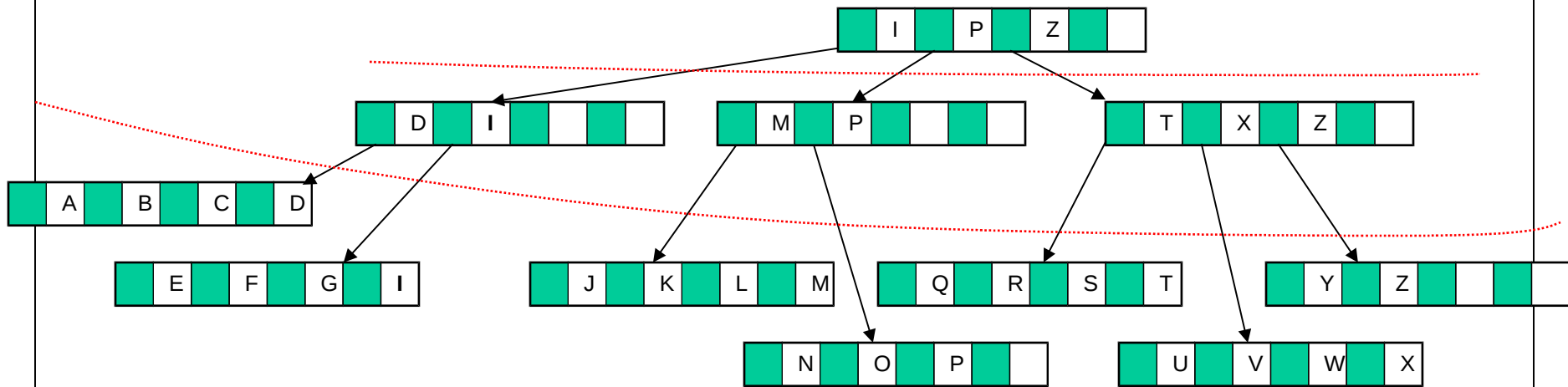
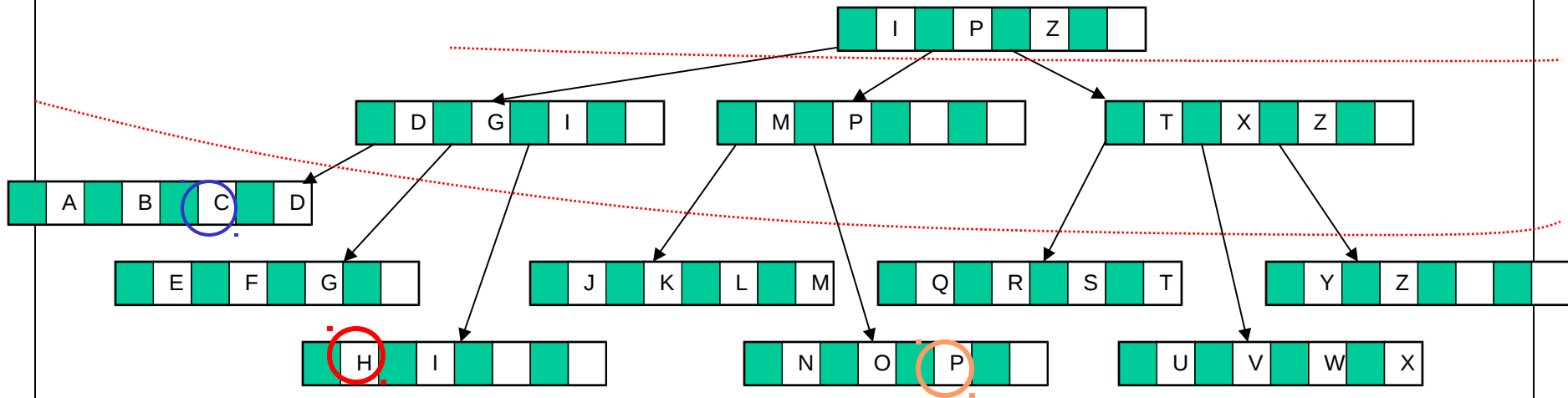
$$N \geq 2 \times \lceil m/2 \rceil^{d-1}, \quad d \leq 1 + \log_{\lceil m/2 \rceil} (N/2)$$

$$\text{if } m = 512, N = 1,000,000 \Rightarrow d \leq 1 + \log_{256} 500,000 \approx 3.37$$

– Deletion, Merging & Redistribution

- deleting a key k from a node n
 - 1) if n has more than the minimum number of keys and k is not the largest in n , simply **delete** k from n .
 - 2) if n has more than the minimum number of keys and k is the largest in n , delete k and **modify** the higher level indexes to reflect the new largest key in n .
 - 3) if n has exactly the minimum number of keys and one of the siblings of n has few enough keys, **merge** n with its sibling and delete a key from the parent node.
 - 4) if n has exactly the minimum number of keys and one of the siblings of n has extra keys, **redistribute** by moving some keys from a sibling to n , and modify the higher level indexes to reflect the new largest keys in the affected nodes. (Fig. 9.21 - 9.22)





– Redistribution

- redistribution does not change the B-tree at the node level
- redistribution is only done between siblings
- redistribution is not unique, normally an even redistribution is desirable
- redistribution may also be used with insertion to avoid or postpone creations of new nodes (as an alternative way to splitting)

– B*-trees

- Using redistribution in B-tree insertion allows two-to-three split, with each new page $2/3$ full.
- B*-tree: the same as B-tree except that each page has at least $\lceil (2m - 1)/3 \rceil$ children.
- The root page is allowed to grow larger than normal page limit such that it can be split into $2/3$ full pages.

– Buffering of pages

- to reduce the number of disk access, some pages can be kept in main memory.
- A simple strategy: keeping the first few levels of the B-tree in memory — this can normally reduce one or two disk access.
- Page buffering: using a fixed-sized buffer in memory to hold pages that are read into the memory. When the buffer is full, some pages in the buffer will be replaced using the LRU replacement rule.
- LRU replacement: The least-recently-used page in the buffer will be the first to be replaced. A “request” order will be assigned to each page, and a priority queue may be used to implement the LRU replacement.
- LRU rule is based on the assumption that we are more likely to access a page that is used recently than a page that has not been used for some time — temporal locality
- Height-weighted replacement: in B-tree, higher level pages are more likely to be accessed than lower level pages.
⇒ Using the B-tree heights of the pages as weights in deciding which page is to be replaced first.

Chap. 10 Indexed Sequential File Access and Prefix B⁺ trees

- Indexed Sequential Access
 - Indexed sequential access: retrieving all records in order by key
 - Indexed sequential access using index file is expensive: N seeks
 - Sorting records is also not feasible
 - The solution has to accommodate both sequential access and random access
- The use of blocks
 - Using block as the basic unit of read/write.
 - The record file is a linked list of blocks that are sorted, but not physically sequential.
 - Records within block are sorted. (Fig. 10.1)
 - Each block holds a fixed number of records, and a block is at least half full.
 - As in B-tree, block split, merging and redistribution are used to support record insertion & deletion.

- Drawbacks:
 - 1.) it uses more memory space due to block fragmentation
 - 2.) no physical locality among blocks
- Block size considerations
 - 1.) the memory buffer should be able to hold several blocks at the same time
 - 2.) the block read/write should take less time than a seek operation

– Indexing Sequence Set

- Using index to access the block in the sequence set.
- A single index file is sufficient if the index file can be entirely kept in memory
 - 1.) binary search is efficient in memory
 - 2.) updating index file can be done in memory due to block splitting, merging & redistribution
- A B-tree like file structure is needed if the index file is too large to be kept in memory.

– Simple prefix B⁺ tree

- Using separators to guide block search.
- Using shortest separator as variable-length keys in the index structure
 - prefix. (Fig. 10.4)

- When a separator is the same as the key, always go to the right child block.
- Simple prefix B⁺-tree:
 - 1.) Separator-based index pages
 - 2.) Each page contains n separator and $n+1$ references to its child blocks.
(Fig. 10.7)

– Simple prefix B⁺-tree maintenance

- record deletion/insertion without changing the number of blocks: No change needed for the index set. (Fig. 10.8)
- insertion involving block splitting: a new separator is needed, and B-tree update may be necessary. (Fig. 10.9)
- Deletion involving block merging: a separator will be removed, and B-tree update may be necessary. (Fig. 10.10)
- Block redistribution will lead to changes of the separator values
- The page size of the index set is often set to be the same as the block size of the sequence set.
 - 1.) physical storage factors
 - 2.) uniform buffering & buffer size
 - 3.) sequence set and index set can be put in a common file.

- Index set block structure: variable order B-tree
 - Number of separators in each index block can be very large, thus requires binary search once read into memory
 - index to separators: it supports variable-length separators & binary search (Fig. 10.11)
 - Reference to child blocks: relative block number (RBN).
 - An index set block contains:
 - 1.) total number of separators, n , and the list of n separators
 - 2.) index to the separators
 - 3.) $n+1$ RBNs associated with the n separators
 - 4.) Total length of the concatenated separators. (Fig. 10.12)
- B⁺-trees
 - B⁺-tree is similar to the simple prefix B⁺-tree except:
 - 1.) the separators are actual copies of the *1st* key in its right child block in a B⁺-tree
 - 2.) separators are fixed-length keys, thus, the index set blocks have fixed number of separators.
 - B⁺-trees are good when:
 - 1.) the overhead in managing variable-length separators outweighs its savings
 - 2.) the prefix-based separators are not very different from the actual keys.

- Comparisons: B-tree, B⁺-tree, Simple prefix B⁺-tree
 - Common properties
 - 1.) all paged index structures
 - 2.) all perfectly balanced trees
 - 3.) button-up growth using splitting, merging & redistribution
 - 4.) all can use two-to-three splitting for more efficient storage utilization
 - 5.) all can use buffering and LRU replacement to reduce number of disk access
 - 6.) variable-length records can be implemented in the sequence set blocks using similar structure used in the simple prefix B⁺ tree.
 - B-tree: simple to implement, but expensive for sequential access
 - B⁺-tree: efficient sequential access, smaller index file (due to the use of record set blocks), but more difficult to implement.
 - Simple prefix B⁺-tree: more efficient index structure (by separator compression), but more difficult to implement than general B⁺-tree (variable-length fields in index set blocks).