Data Structures (CS 1520) Lecture 24 Name:_________________

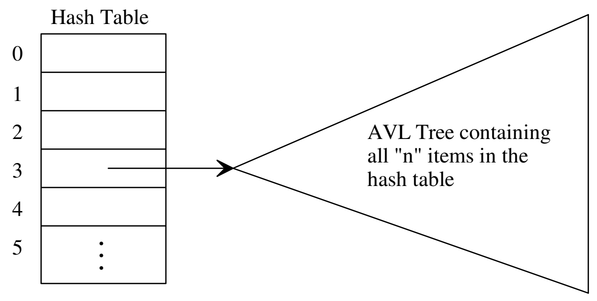
1. BST, AVL trees, and hash tables can all be used to implement a dictionary ADT.

Dictionary Successful Search Comparisons with 10,000 integer items (Time in seconds)						
	Items added in sorted order		Items added in random order		Order did not matter (Hash table sizes 2^{15} = 32K)	
	BST	AVL Tree	BST	AVL Tree	Open Addr. (Quadratic)	Closed Addr. (Chaining)
Total add/put time	47.785	0.205	0.119	0.195	0.064	0.074
Total search time	38.100	0.060	0.079	0.062	0.044	0.039
Height of resulting tree	9,999	13	30	15	NA	NA

a) The `puts` of these 10,000 randomly ordered items into the BST took 0.119 seconds and 0.195 seconds into the AVL tree. Why did the BST `puts` take less time eventhough the final height was 30 vs. a final AVL tree height of 15?

b) With a very, very poor hash function or very, very bad choice of keys all keys could hash to the same home address.

- What would be the worst-case big-oh of open-address hashing with quadratic probing?
- What would be the worst-case big-oh of chaining using a linked list at each home address (i.e., ChainingDict)?
- What would be the worst-case big-oh of chaining using an AVL tree at each home address?



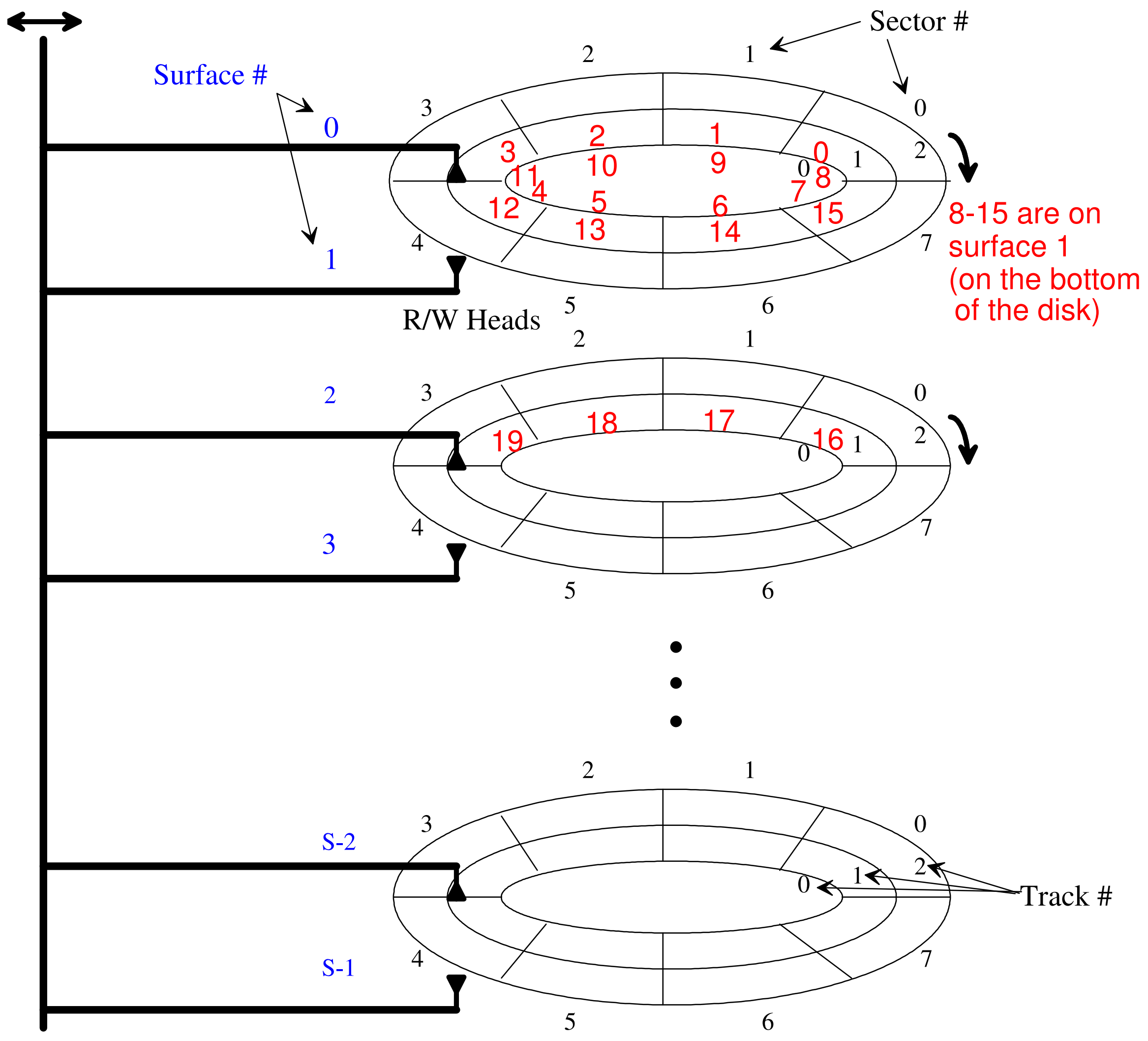
2. The data structures we have discussed so far are all in-memory, i.e., data is stored in main/RAM memory. Data can also be stored on secondary storage in a file (e.g., moiveData.txt file). Currently, most secondary storage consists of hard-disks.

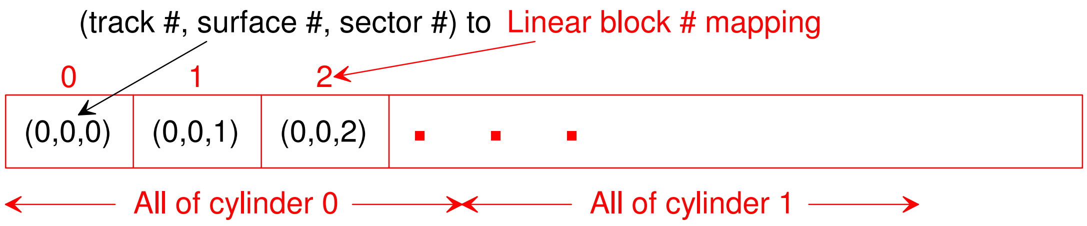
a) Complete the following table comparing main/RAM memory vs. hard-disk:

Criteria	Main/RAM memory	Hard-disk Drive	Solid-State Drive
Size on a typical desktop computer			
Average access time			

b) Which criterion seems to be the most important difference between the main and secondary memories?

Logical View of Disk as Linear Collection of Blocks





Bits of linear block # : | track # | surface # | sector # |

3. Disk-access time = (seek time) + (rotational delay) + (date transfer time). How is each component of the disk-access time effected by increasing the disk's RPMs (revolutions per minute)?

b) If we want fast access to a collection of sectors, where can we place them to minimize seek time and rotational delay?

User Program - HLL programming language make system calls to OS to:

1. open file - establish a link between file variable and file for either reading, writing, or both
2. access file - read or write one piece of data at a time (e.g., char., record, etc.)
3. close file - flush changes to disk

Operating System - manage and control access to secondary storage through its file system which contains information about every file: location on disk, ownership and security/protection

- OS views disk as linear sequence of blocks (block 0, block 1, etc.), but assumes closeness in block # means close with respect to access time.
- OS buffers some blocks in memory to improve efficiency
- OS maintains free disk space "list"

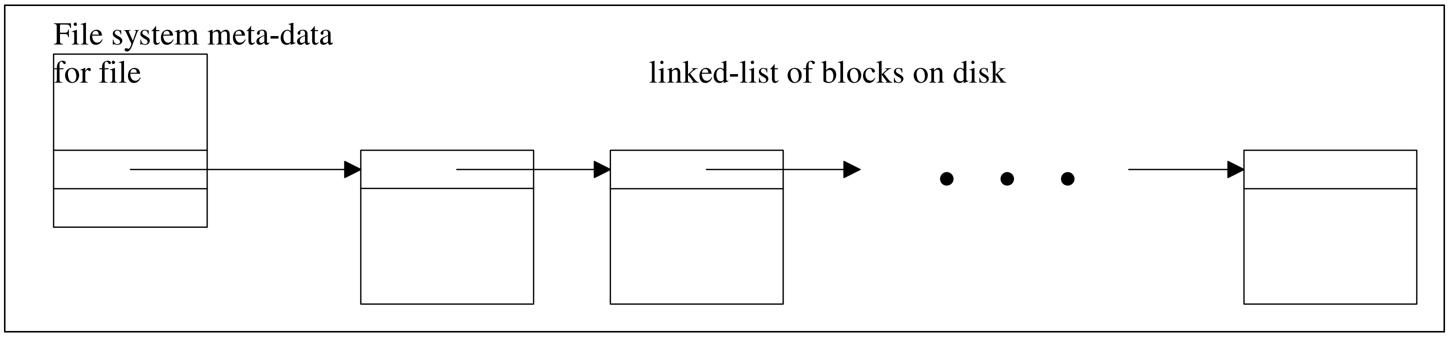
Secondary Storage Device - accepts R/W requests from OS for block# and maps block# to internals physical address (e.g., (track #, surface #, sector #) - more complex than above picture!)

Kinds of File Access:

- serial/sequential files - open at the beginning and read sequentially from beginning to end linearly
- random-access files - "*seek*" to any position by specifying a byte-offset from the beginning of the file, record #, etc.
- random-access of a record by key

Implementation of Files on Disk- how are blocks allocated?

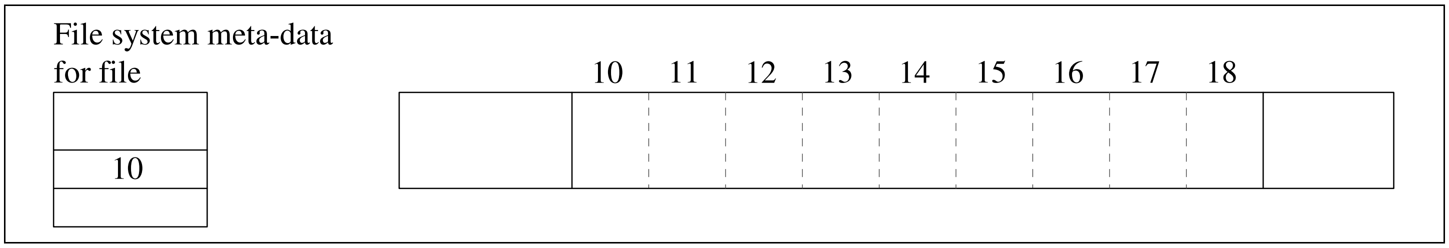
4. non-contiguous - scattered across linear address space of OS and disk



a) What types of file access are supported efficiently?

b) How easy is it for the file to grow in size?

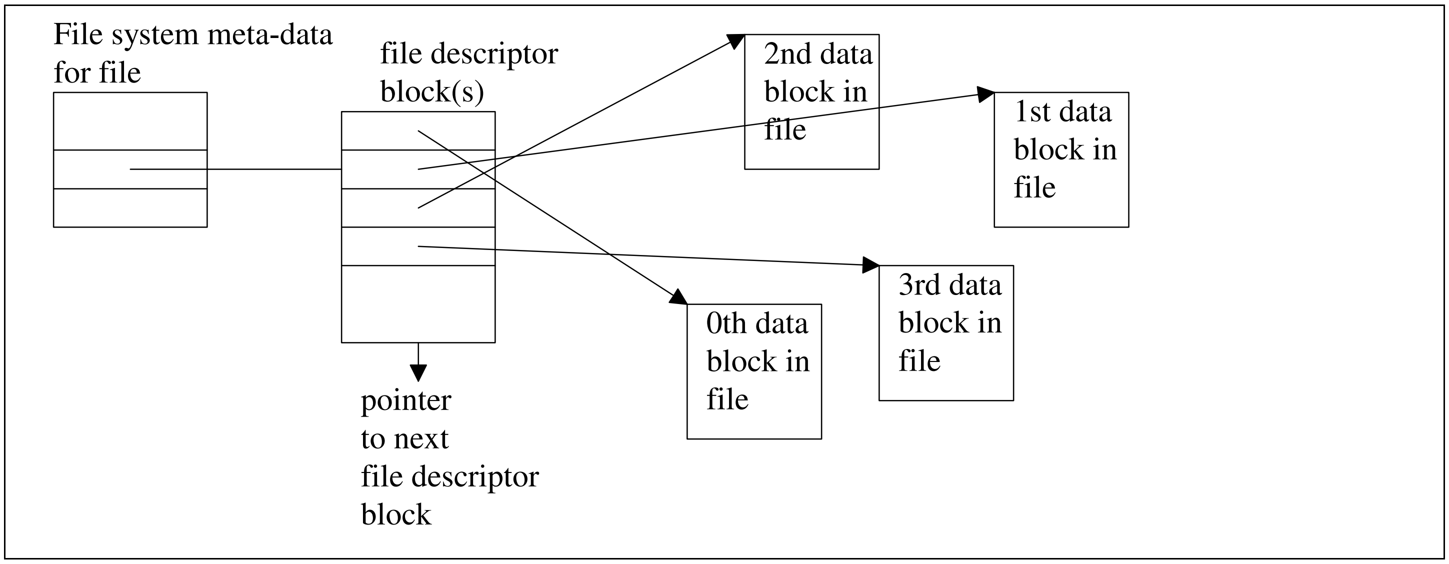
5. contiguous - sequential collection of blocks from OS linear view of disk



a) What types of file access are supported efficiently?

b) How easy is it for the file to grow in size?

6. file descriptor blocks - list of blocks hold the address of the physical location of data blocks



a) What types of file access are supported efficiently?

b) How easy is it for the file to grow in size?

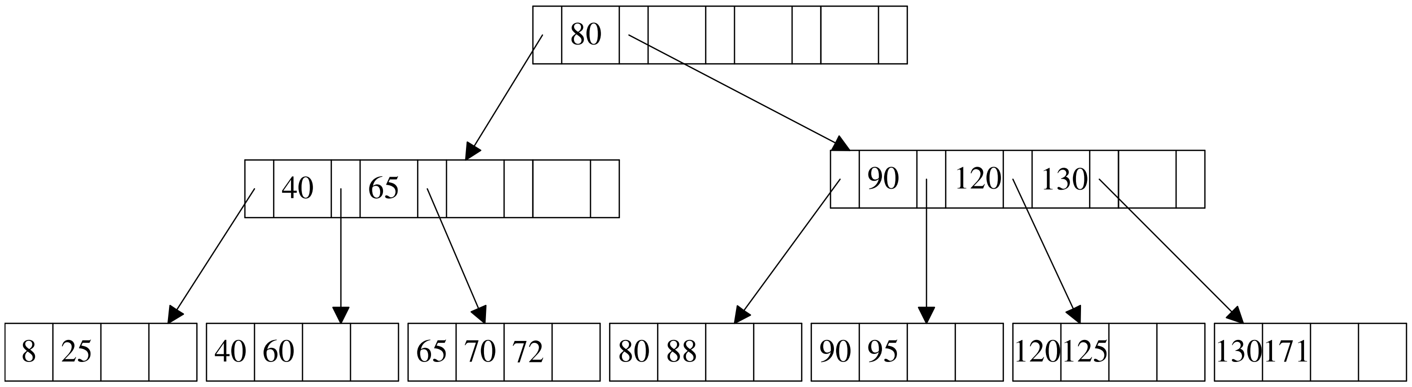
7. To implement "random-access of a record by key" in a file how might we use hashing?

8. To implement "random-access of a record by key" in a file why would an AVL tree not work well?

9. A B+ Tree is a multi-way tree (typically in the order of 100s children per node) used primarily as a file-index structure to allow fast search (as well as insertions and deletions) for a target key on disk. Two types of *pages* (B+ tree "nodes") exist:

- Data pages - which always appear as leaves on the same level of a B+ tree (usually a doubly-linked list too)
- Index pages - the root and other interior nodes above the data page leaves. Index nodes contain some minimum and maximum number of keys and pointers bases on the B+ tree's *branching factor (b)* and *fill factor*. A 50% fill factor would be the minimum for any B+ tree. All index pages must have $\lceil b/2 \rceil \leq$ # child $\leq$ b, except the root which must have at least two children.

Consider an B+ tree example with b = 5.



a) How would you find 88?

b) The insert algorithm for a B+ tree is summarized by the below table. Where would you insert 50, 100, 105, 110, 180, 200, 210?

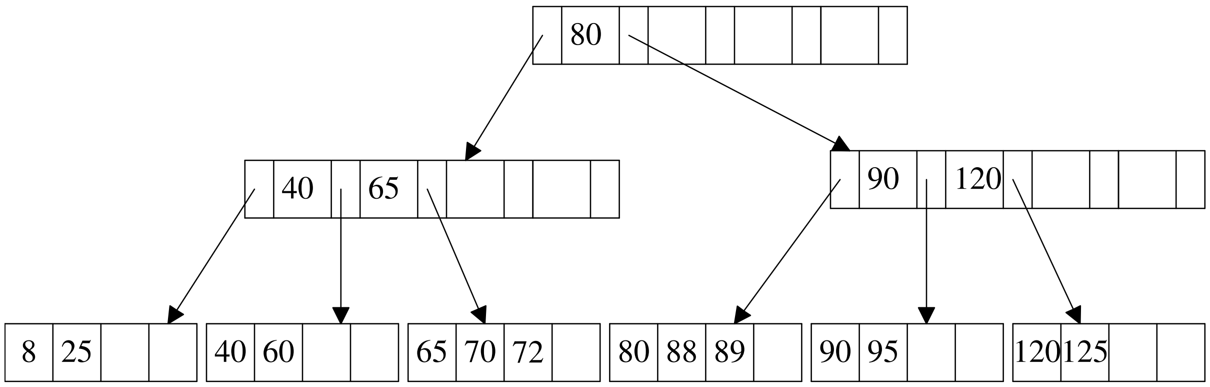
Situation		insertion Algorithm
Data Page Full?	Parent Index Page Full?	
No	No	Place record in sorted position in the appropriate data page.
Yes	No	1. Split data page with records < middle key going in left data page and records ≥ middle key going in right data page. 2. Place middle key in index page in sorted order with the pointer immediately to its left pointing to the left data page and the pointer immediately to its right pointing to the right data page.
Yes	Yes	1. Split data page with records < middle key going in left data page and records ≥ middle key going in right data page. 2. Adding middle key to parent index page causes it to split with keys < middle key going into the left index page, keys > middle key going in right index page, **and** the middle key inserted into the next higher level index page. If the next higher index page is full continue to splitting index pages up the B+ tree as necessary.

c) For a B+ tree with a branch factor 201, what would be the worst case height of the tree if the number of keys was 1,000,000,000,000?

10. The deletion algorithm for a B+ tree is summarized by the below table.

Situation		deletion Algorithm
Data Page Below Fill Factor?	Parent Index Page Below Fill Factor?	
No	No	Delete record from the data page. Shifting records with larger keys to left to fill in the hole. If the deleted key appears in the index page, use the next key to replace it.
Yes	No	1. Combine data page and its sibling. Change the index page to reflect the change.
Yes	Yes	1. Combine data page and its sibling. 2. Adjusting the index page to reflect the change causes it to drop below the fill factor, so combine the index page with its sibling. 3. Continue combining the next higher level index pages until you reach an index page with the correct fill factor or you reach the root index page.

Consider an B+ tree example with b = 5 and 50% fill factor. Delete 89, 65, and 88. What is the resulting B+ tree?


80
40
65
90
120
8
25
40
60
65
70
72
80
88
89
90
95
120
125