10) Lecture 27.5 to 28 |B Tree |

26 September 2023 12:45

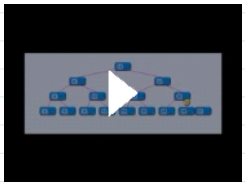
(2,4) tree

① All the leaf are at same level.

② Each node should have atmost three and atleast one key

③ " " " atmost four atleast two childre

Slide Credit : Yael Moses, IDC Herzliya
Animated Video B-Tree example



Motivation

- Large differences between time access to disk, cash memory and core memory
- Minimize expensive access (e.g., disk access)
- B-tree: Dynamic sets that is optimized for disks

B-Trees

A B-tree is an M-way search tree with two properties :

1. It is perfectly balanced: every leaf node is at the same depth
2. Every internal node other than the root, is at least half-full, i.e. $M/2-1 \le \#keys \le M-1$
3. Every internal node with k keys has $k+1$ non-null children

$\frac{M}{2}-1 \le \#key \le M-1$

For simplicity we consider M even and we use $t=M/2$:

2.* Every internal node other than the root is at least half-full, i.e. $t-1 \le \#keys \le 2t-1$, $t \le \#children \le 2t$

$K_1, K_2 \dots K_r$

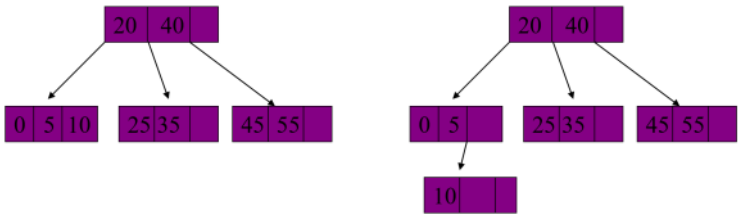
$t-1 \le r \le 2t-1$ #of keys

$t \le$ # children $\le 2t$

Example: a 4-way B-tree

$2t-1=3$

$t=2$



B-tree *4-way tree*

B-tree
1. It is perfectly balanced: every leaf node is at the same depth.
2. Every node, except maybe the root, is at least half-full
   $t-1 \le \#keys \le 2t-1$
3. Every internal node with k keys has $k+1$ non-null children

$K_1 \dots K_r$ — One node

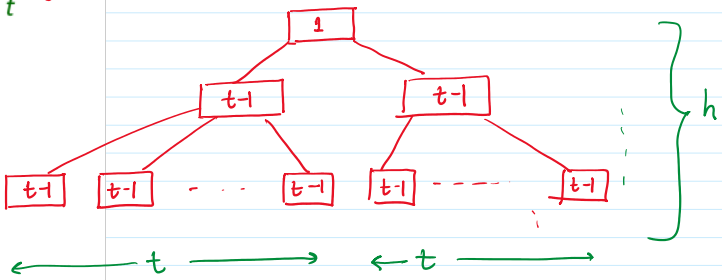
$t-1 \le r \le 2t-1$

B-tree Height

t = #of children for any internal node

Claim: any B-tree with n keys, height h and minimum degree t satisfies:

$$h \le \log_t \frac{n+1}{2}$$



$n \ge 1 + (t-1) \times$ #of nodes

$= 1 + (t-1) \sum_{i=1}^{h} 2t^{i-1}$

Proof:

- The minimum number of KEYS for a tree with height h is obtained when:
  - The root contains one key
  - All other nodes contain $t-1$ keys

$n \ge 1 + (t-1) \times$ # of nodes in B tree

$= 1 + (t-1) \times \sum_{i=1}^{h} 2t^{i-1}$

$$= 1 + (t-1) \times \sum_{i=1}^{h} 2\, t^{i-1}$$

$$n \geq 1 + (t-1) \sum_{i=1}^{h} 2\, t^{i-1}$$

B-Tree: Insert X

1. As in *M-way* tree find the leaf node to which *X* should be added
2. Add *X* to this node in the appropriate place among the values already there
   (there are no subtrees to worry about)
3. Number of values in the node after adding the key:
   - Fewer than *2t-1:* done
   - Equal to *2t: overflowed*
4. Fix overflowed node

$t-1 \leq$ # of keys $\leq 2t-1$

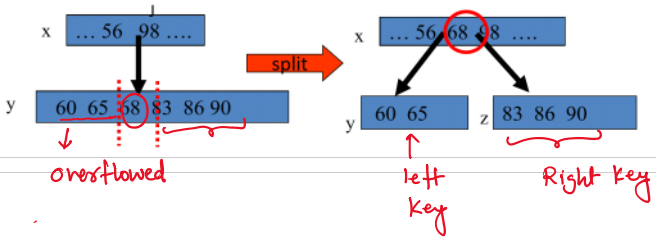
$t \leq$ # of children $\leq 2t$

Fix an Overflowed

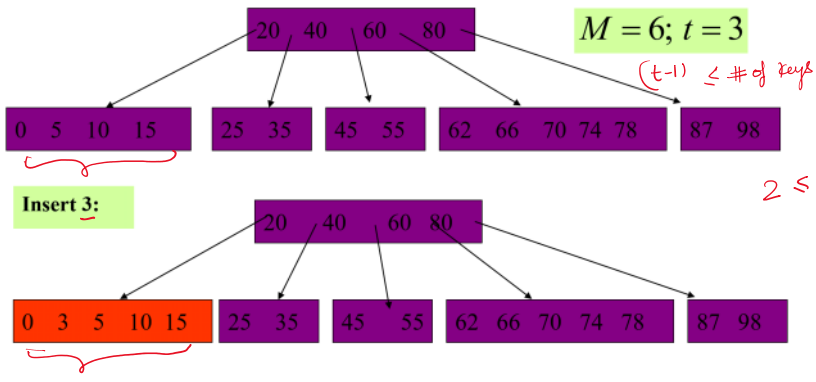
1. Split the node into three parts, *M=2t*:
   - Left: the first *t* values, become a left child node
   - Middle: the middle value at position *t*, goes up to parent
   - Right: the last *t-1* values, become a right child node
2. Continue with the parent:
   1. Until no overflow occurs in the parent
   2. If the root overflows, split it too, and create a new root node

(2t-1) key

2t



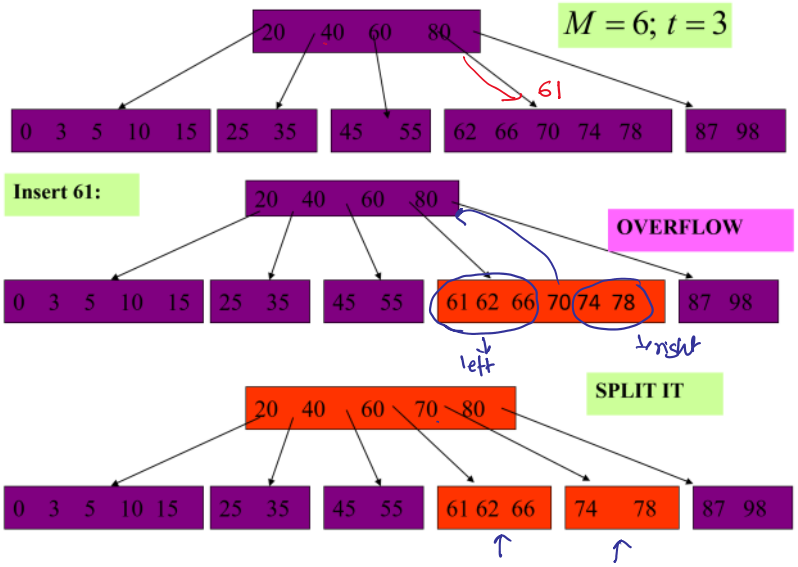
Insert example

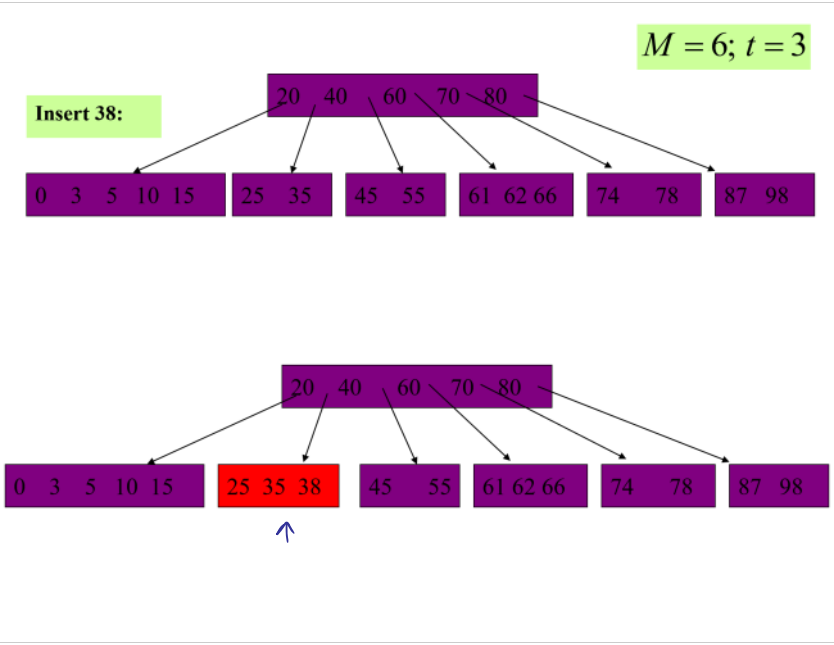


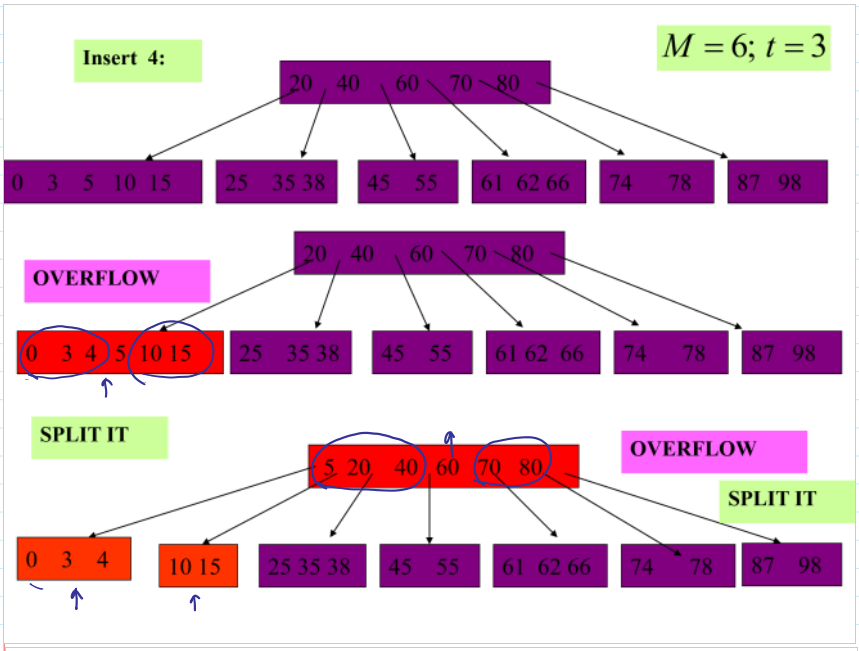
$(t-1) \leq$ # of keys $\leq (2t-1)$

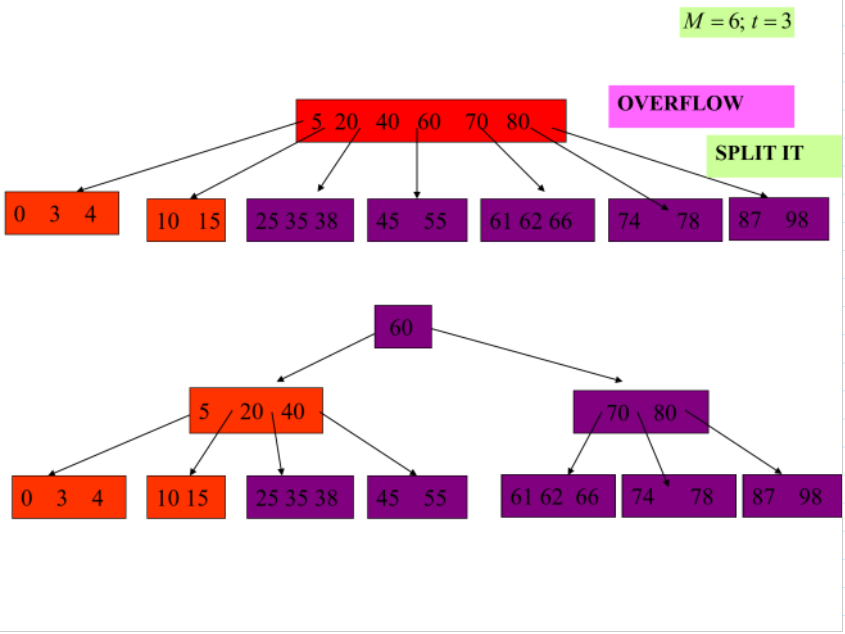
$2 \leq$ # of keys ≤ 5


0 3 5 10 15
25 35
45 55
62 66 70 74 78
87 98


M = 6; t = 3
20 40 60 80
61
0 3 5 10 15
25 35
45 55
62 66 70 74 78
87 98
Insert 61:
OVERFLOW
61 62 66 70 74 78
left
right
SPLIT IT
20 40 60 70 80
61 62 66
74 78


M = 6; t = 3
Insert 38:
20 40 60 70 80
0 3 5 10 15
25 35
45 55
61 62 66
74 78
87 98
25 35 38





Complexity Insert

- Inserting a key into a B-tree of height *h* is done in a single pass down the tree and a single pass up the tree

Complexity: $O(h) = O(\log_t n)$

$t-1 \leq$ # keys $\leq 2t-1$

B-Tree: Delete X

- *Delete as in M-way tree*
- *A problem:*
  - might cause *underflow*: the number of keys remain in a node < t-1

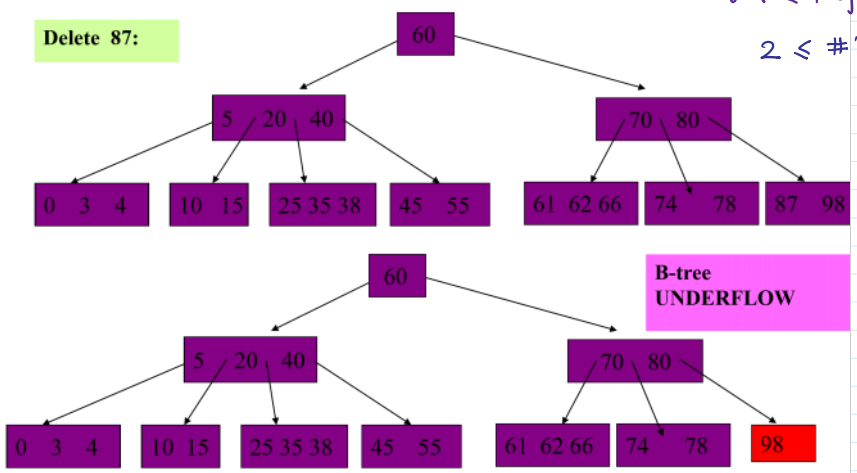
Recall: The root should have at least 1 value in it, and all other nodes should have at least t-1 values in them

$M = 6; t = 3$

Underflow Example

$t-1 \leq$ # of keys $\leq 2t-1$

$2 \leq$ # keys ≤ 5



B-Tree: Delete X,k

- *Delete as in M-way tree*
- *A problem:*
  - might cause *underflow*: the number of keys remain in a node < t-1
- Solution:
  - make sure a node that is visited has at least t instead of t-1 keys.
  - If it doesn't have k
    - (1) either take from sibling via a rotate, or
    - (2) merge with the parent
  - If it does have k
    - See next slides

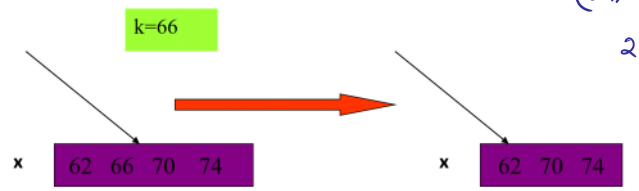
Recall: The root should have at least 1 value in it, and all other nodes should have at least t-1 (at most 2t-1) values in them

B-Tree-Delete(x,k)

1st case: *k* is in *x* and *x* is a *leaf* → delete *k*

$(t-1) \leq$ # keys $\leq (2t-1)$

$2 \leq$ # keys ≤ 5



How many keys are left?

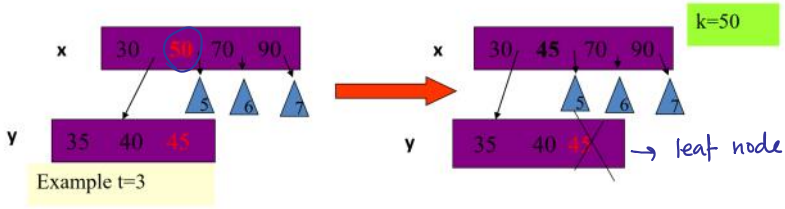
Example t=3

2nd case: *k* in the internal node *x*, *y* and z are the preceding and succeeding nodes of the key k∈ *x*

a. If *y* has at least *t* keys:

- Replace *k* in *x* *k'* ∈ *y*, where *k'* is the predecessor of *k* in *y*
- Delete *k'* recursively

b. Similar check for successor case

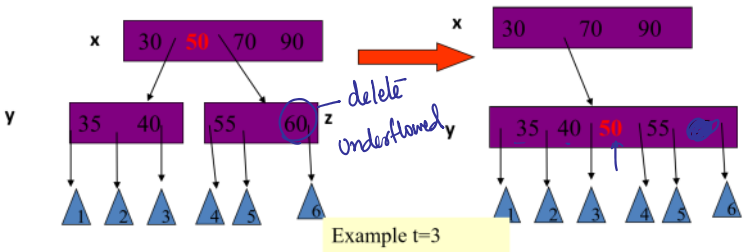


Example t=3

2nd case cont.:

c. Both a and b are not satisfied: y and z have *t-1* keys

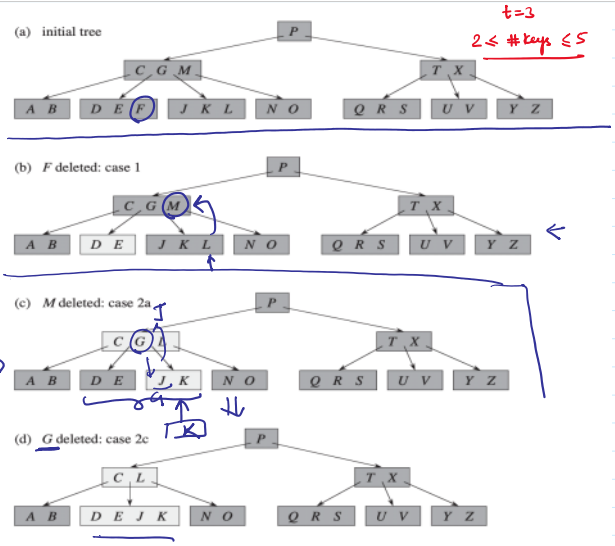
- Merge the two children, y and z
- Recursively delete *k* from the merged cell



Example t=3

Questions

- When does the height of the tree shrink?
- Why do we need the number of keys to be at least *t* and not *t-1* when we proceed down in the tree?



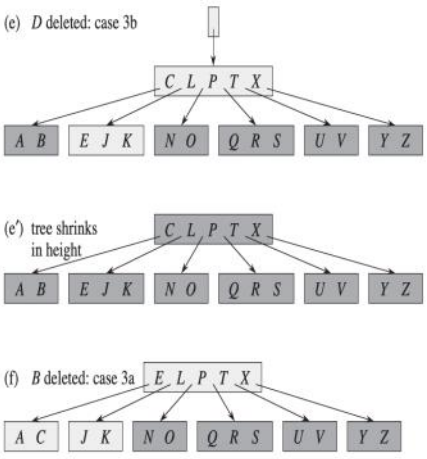


Figure 18.8 Deleting keys from a B-tree. The minimum degree for this B-tree is $t = 3$, so a node (other than the root) cannot have fewer than 2 keys. Nodes that are modified are lightly shaded. **(a)** The B-tree of Figure 18.7(e). **(b)** Deletion of F. This is case 1: simple deletion from a leaf. **(c)** Deletion of M. This is case 2a: the predecessor L of M moves up to take M's position. **(d)** Deletion of G. This is case 2c: we push G down to make node $DEGJK$ and then delete G from this leaf (case 1).

Figure 18.8, continued **(e)** Deletion of D. This is case 3b: the recursion cannot descend to node CL because it has only 2 keys, so we push P down and merge it with CL and TX to form $CLPTX$; then we delete D from a leaf (case 1). **(e′)** After (e), we delete the root and the tree shrinks in height by one. **(f)** Deletion of B. This is case 3a: C moves to fill B's position and E moves to fill C's position.

Delete Complexity

- Basically downward pass:
  - Most of the keys are in the leaves – one downward pass
  - When deleting a key in internal node – may have to go one step up to replace the key with its predecessor or successor

Complexity $O(h) = O(\log_t n)$

Run Time Analysis of B-Tree Operations

- For a B-Tree of order *M=2t*
  - #keys in internal node: *M-1*
  - #children of internal node: between *M/2* and *M*
  - → Depth of B-Tree storing *n* items is $O(\log_{M/2} N)$
- Find run time is:
  - *O(log M)* to binary search which branch to take at each node, since *M* is constant it is *O(1).*
  - Total time to find an item is *O(h*log M) = O(log n)*
- Insert & Delete
  - Similar to find but update a node may take : *O(M)=O(1)*

Note: if M is >32 it worth using binary search at each node

A typical B-Tree

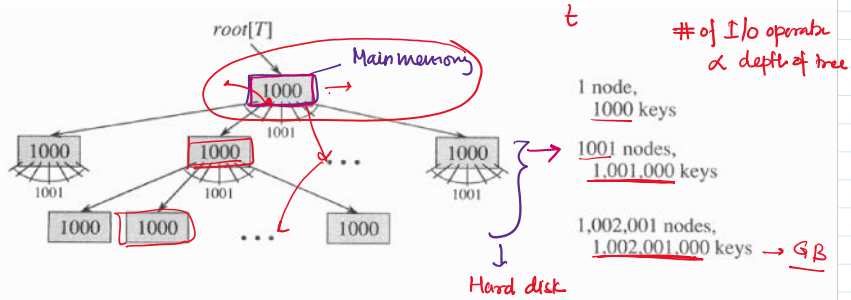


Figure 18.3 A B-tree of height 2 containing over one billion keys. Each internal node and leaf contains 1000 keys. There are 1001 nodes at depth 1 and over one million leaves at depth 2. Shown inside each node x is $n[x]$, the number of keys in x.

Why B-Tree?

- B-trees is an implementation of dynamic sets that is optimized for disks
  - The memory has an hierarchy and there is a tradeoff between size of units/blocks and access time
  - The goal is to optimize the number of times needed to access an "expensive access time memory"
  - The size of a node is determined by characteristics of the disk – block size – page size
  - The number of access is proportional to the tree depth

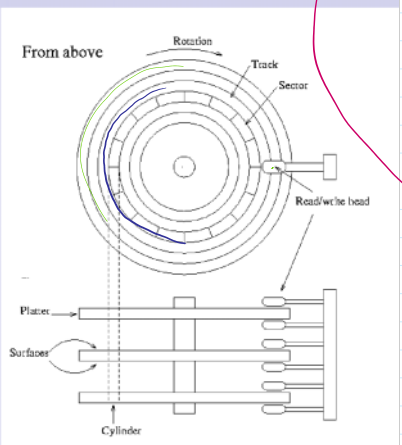
Disk Based Data Structures

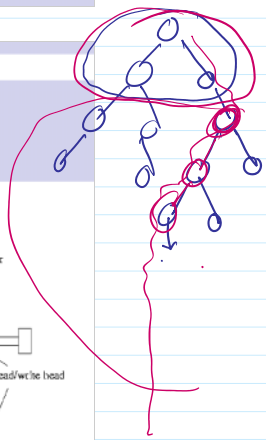
- So far search trees were limited to main memory structures
  - Assumption: the dataset organized in a search tree fits in main memory (including the tree overhead)
- Counter-example: transaction data of a bank > 1 GB per day
  - use secondary storage media (punch cards, hard disks, magnetic tapes, etc.)
- Consequence: make a search tree structure secondary-storage-enabled

Key value

Hard Disks

- **Large amounts of storage, but slow access!**
- Identifying a page takes a long time (seek time plus rotational delay – 5-10ms), reading it is fast
  - It pays off to read or write data in **pages** (or blocks) of 2-16 Kb in size.





Algorithm analysis

- The running time of disk-based algorithms is measured in terms of
  - computing time (CPU)
  - number of disk accesses
    - sequential reads
    - random reads
- Regular main-memory algorithms that work one data element at a time can not be “ported” to secondary storage in a straight-forward way