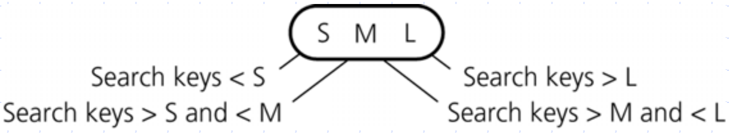
Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

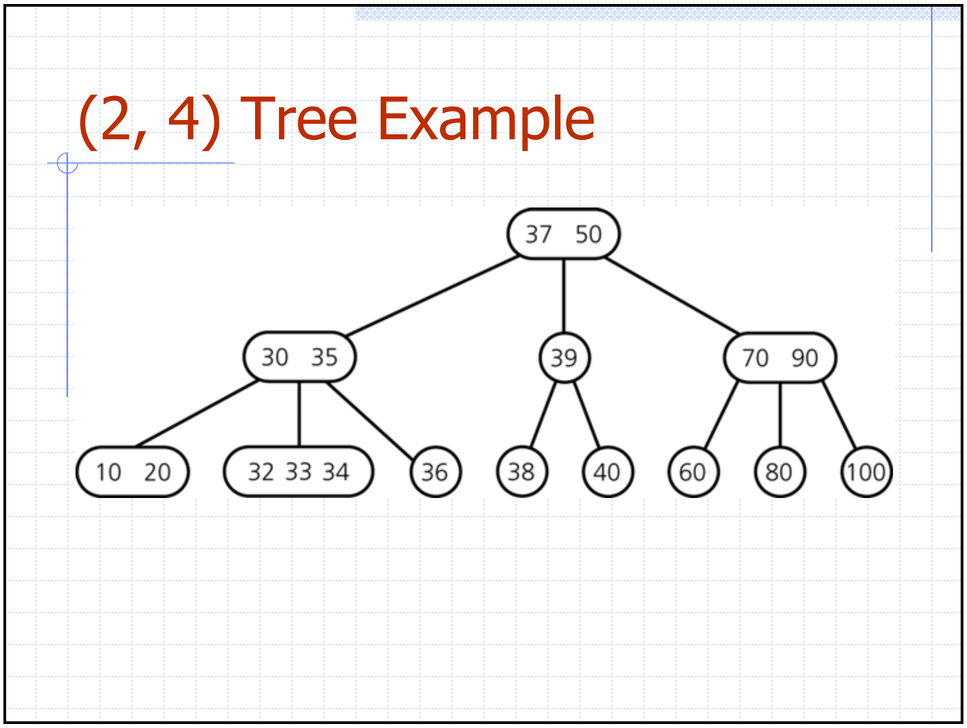
B-Trees and External Memory

Columbia Supercomputer at NASA's Advanced Supercomputing Facility at Ames Research Center, 2006. U.S. government image. Credit: Trower, NASA.

(2, 4) Trees: Generalization of BSTs

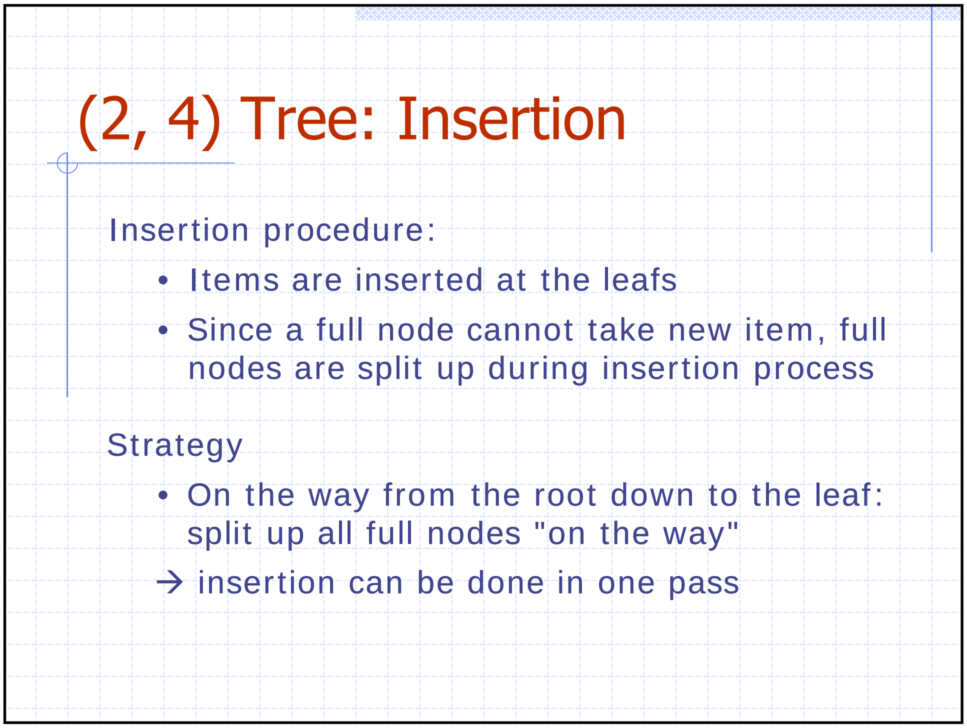
- Each internal node has 2 to 4 children:
  - The number of data items in each internal node is one less than the number of its children.
- Leaves (external nodes) are at the same level and have 1 to 3 data items.
- Nodes with 3 items are called full nodes.
- Items in each node are in ascending order: S < M < L
- It also maintains the following order:





(2, 4) Tree Example

3



(2, 4) Tree: Insertion

Insertion procedure:

- Items are inserted at the leafs
- Since a full node cannot take new item, full nodes are split up during insertion process

Strategy

- On the way from the root down to the leaf: split up all full nodes "on the way"

→ insertion can be done in one pass

4

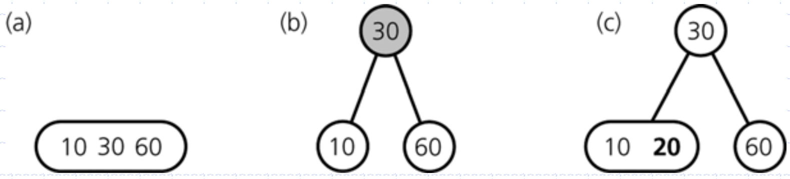
(2, 4) Tree: Insertion

Inserting 60, 30, 10, 20, 50, 40, 70, 80, 15, 90, 100

5

(2, 4) Tree: Insertion

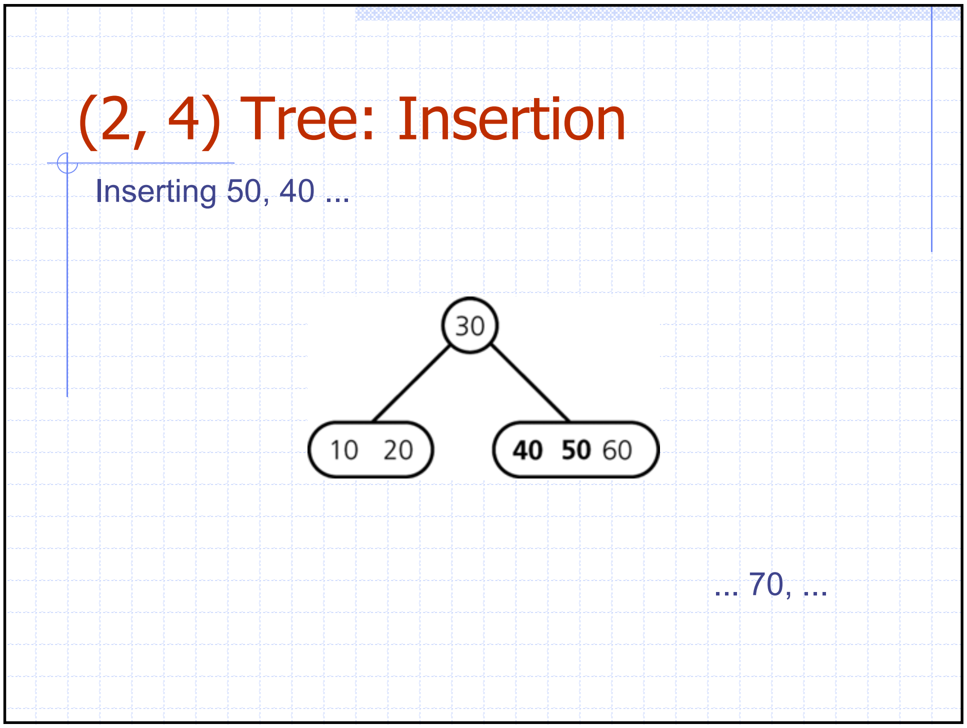
Inserting 60, 30, 10, 20 ...



The root is split and a new root is created.

... 50, 40 ...

6

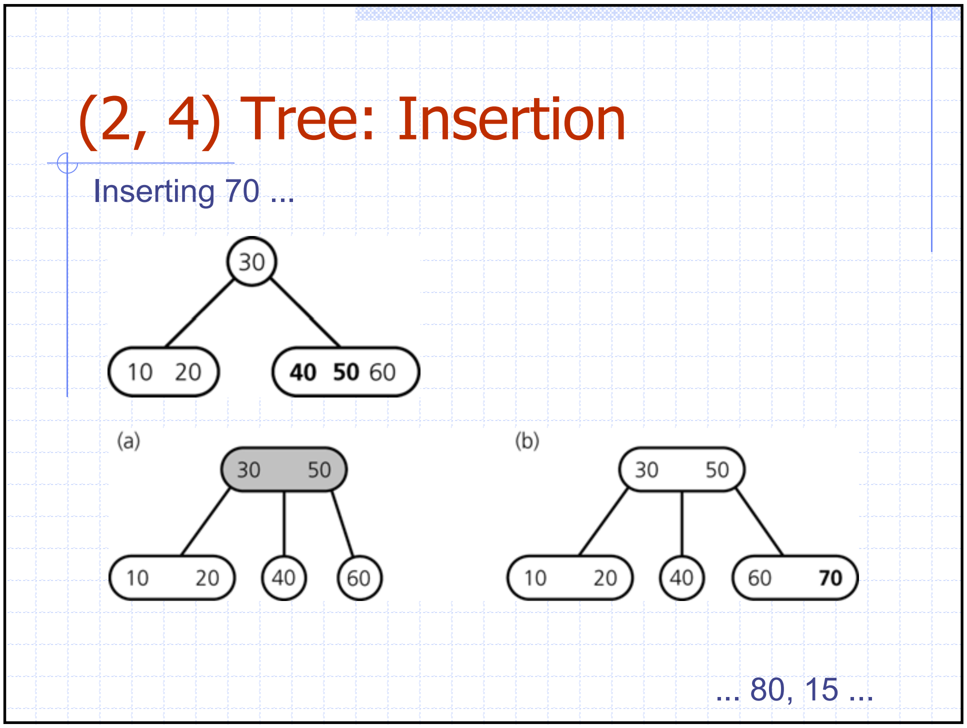


(2, 4) Tree: Insertion

Inserting 50, 40 ...

... 70, ...

7

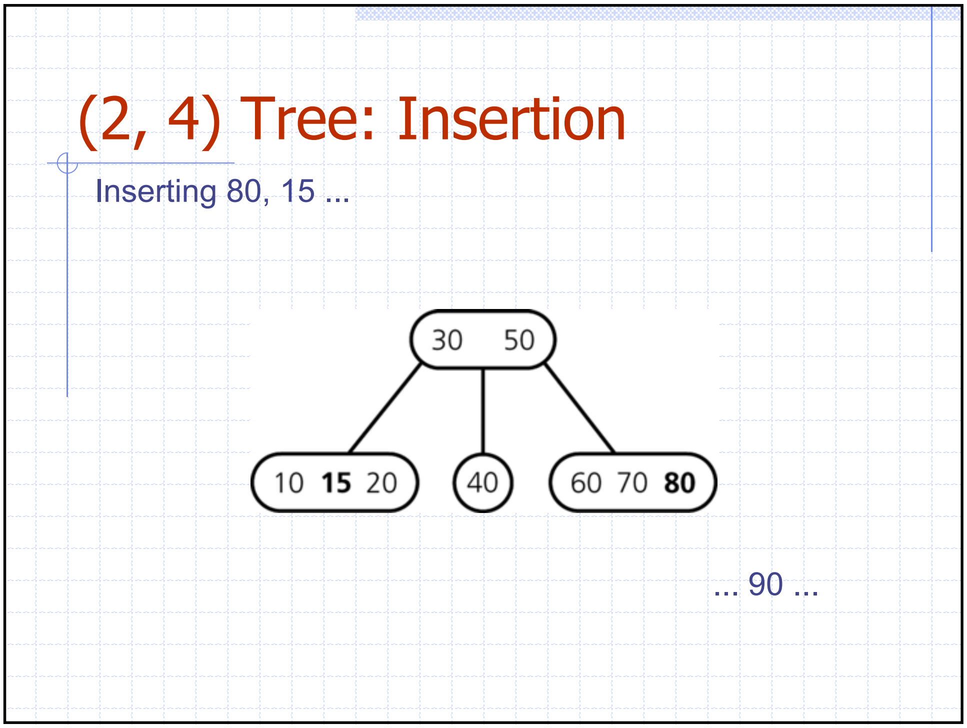


(2, 4) Tree: Insertion

Inserting 70 ...

... 80, 15 ...

8

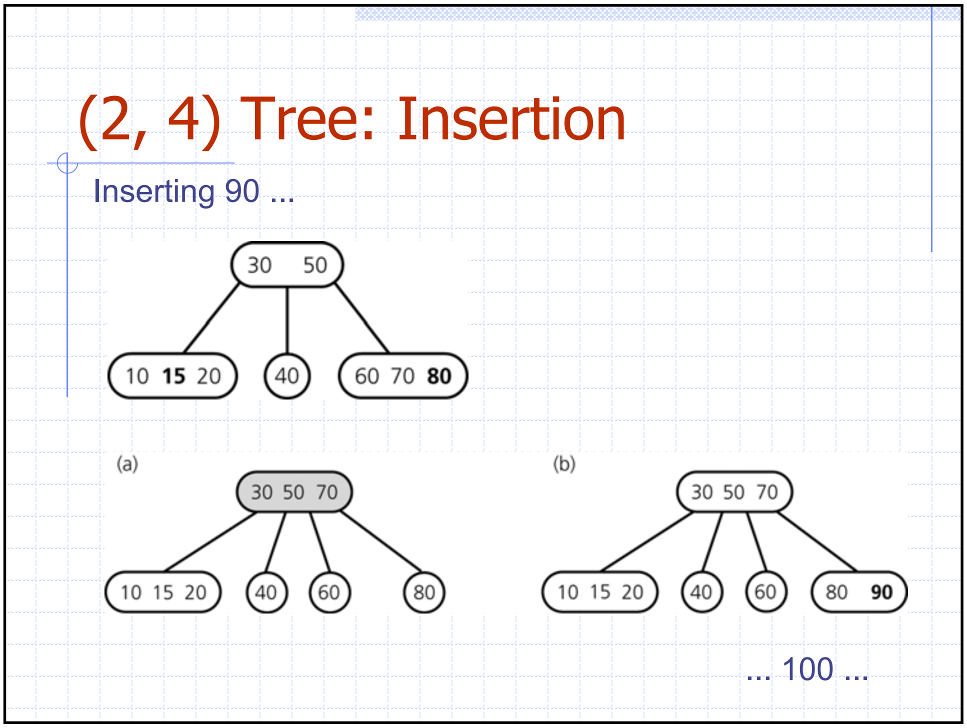


(2, 4) Tree: Insertion

Inserting 80, 15 ...

... 90 ...

9



(2, 4) Tree: Insertion

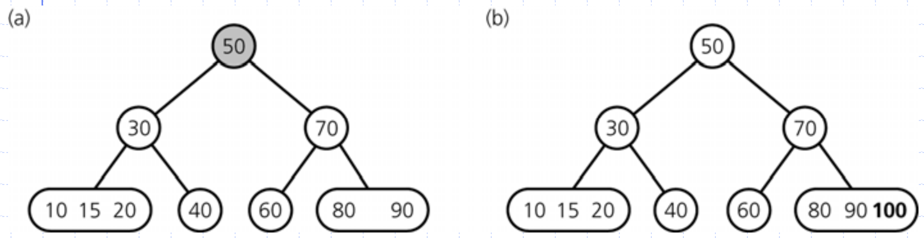
Inserting 90 ...

... 100 ...

10

(2, 4) Tree: Insertion

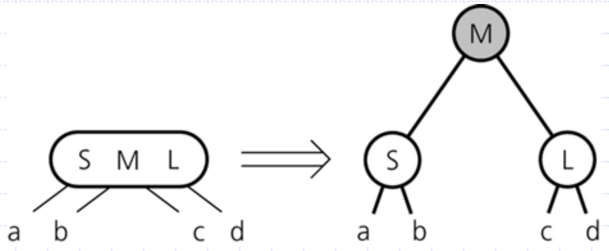
Inserting 100 ...
The root is split and a new root is created.



11

(2, 4) Tree: Insertion Procedure

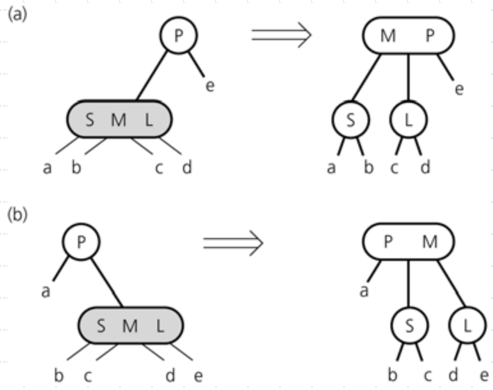
Splitting full nodes during Insertion



12

(2, 4) Tree: Insertion Procedure

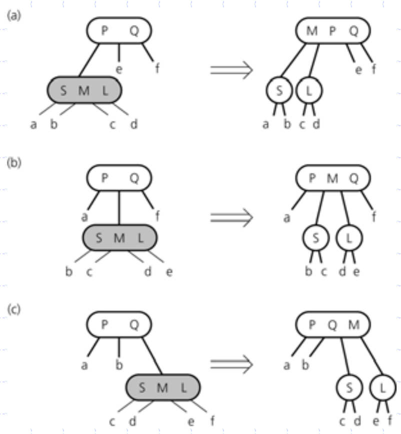
Splitting a full node whose parent is a 2-node during insertion



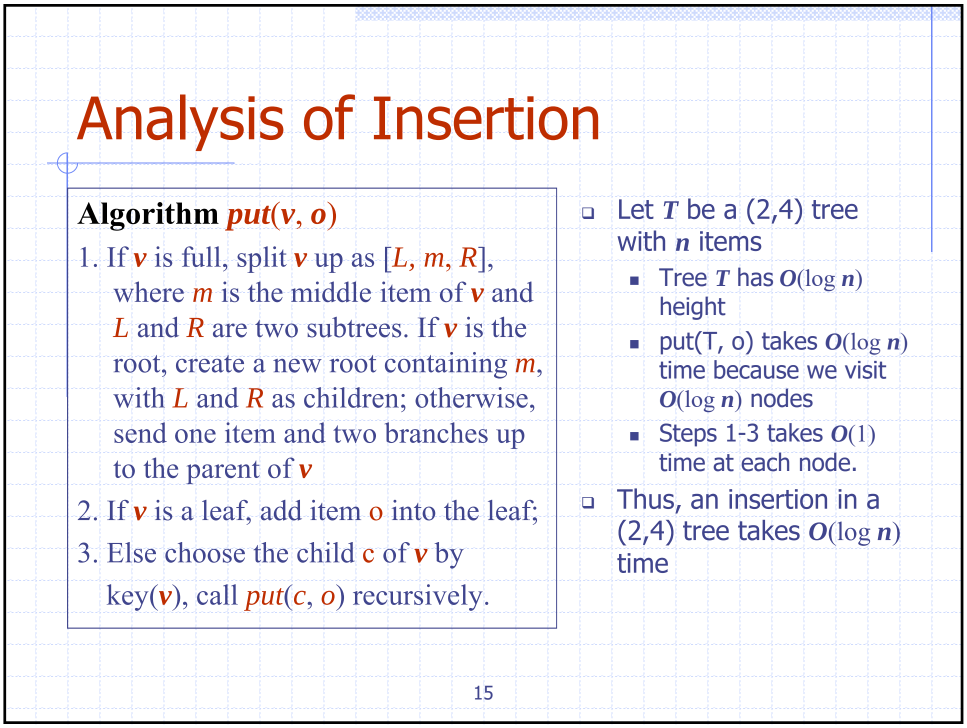
13

(2, 4) Tree: Insertion Procedure

Splitting a full node whose parent has 2 items



14



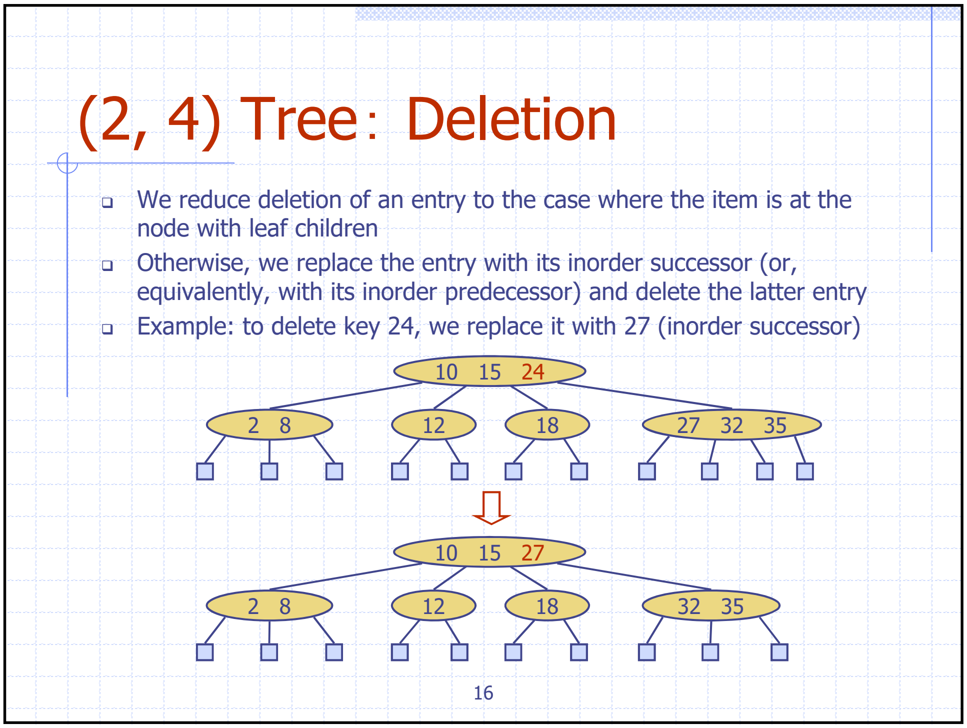
Analysis of Insertion

```
Algorithm put(v, o)
1. If v is full, split v up as [L, m, R],
     where m is the middle item of v and
     L and R are two subtrees. If v is the
     root, create a new root containing m,
     with L and R as children; otherwise,
     send one item and two branches up
     to the parent of v
2. If v is a leaf, add item o into the leaf;
3. Else choose the child c of v by
     key(v), call put(c, o) recursively.
```

- Let T be a (2,4) tree with n items
    - Tree T has $O(\log n)$ height
    - put(T, o) takes $O(\log n)$ time because we visit $O(\log n)$ nodes
    - Steps 1-3 takes $O(1)$ time at each node.
- Thus, an insertion in a (2,4) tree takes $O(\log n)$ time

15

15

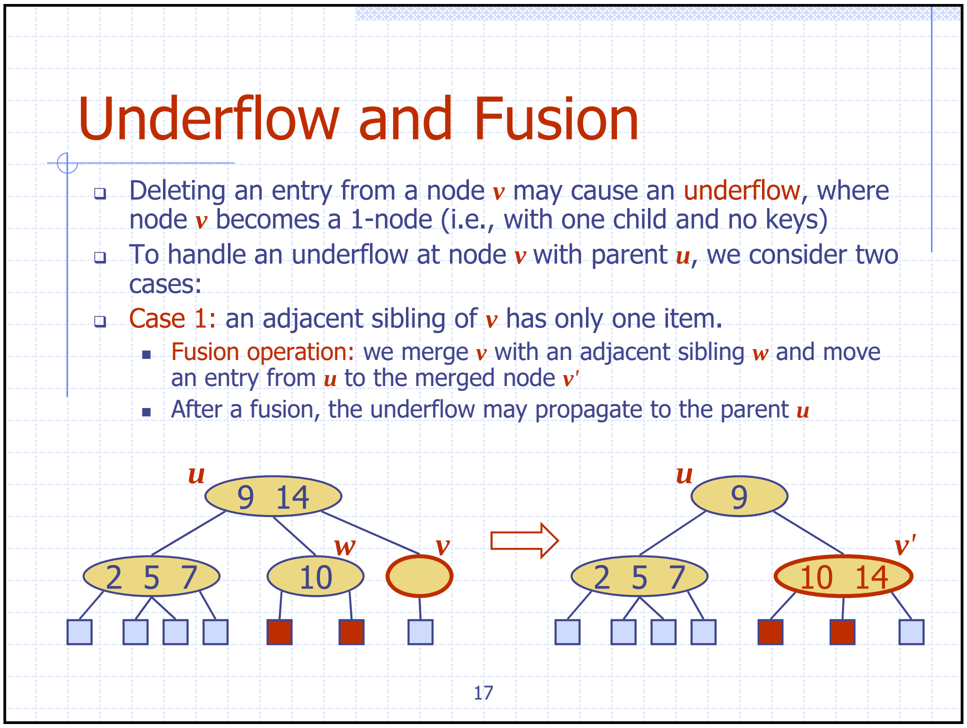


(2, 4) Tree: Deletion

- We reduce deletion of an entry to the case where the item is at the node with leaf children
- Otherwise, we replace the entry with its inorder successor (or, equivalently, with its inorder predecessor) and delete the latter entry
- Example: to delete key 24, we replace it with 27 (inorder successor)

16

16

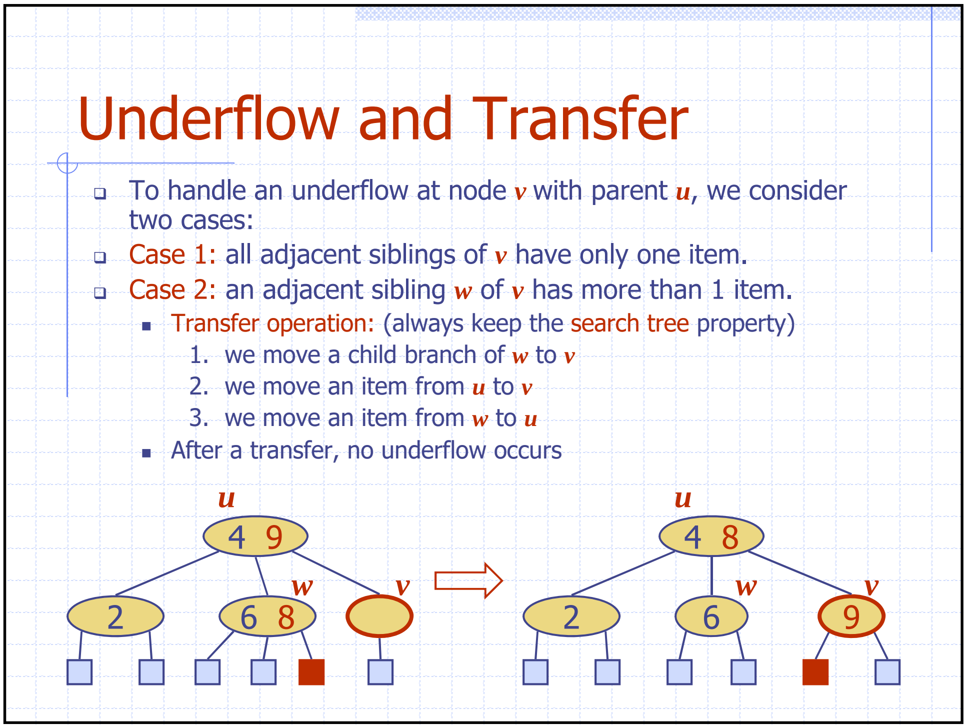


Underflow and Fusion

- Deleting an entry from a node *v* may cause an underflow, where node *v* becomes a 1-node (i.e., with one child and no keys)
- To handle an underflow at node *v* with parent *u*, we consider two cases:
- Case 1: an adjacent sibling of *v* has only one item.
  - Fusion operation: we merge *v* with an adjacent sibling *w* and move an entry from *u* to the merged node *v'*
  - After a fusion, the underflow may propagate to the parent *u*

17

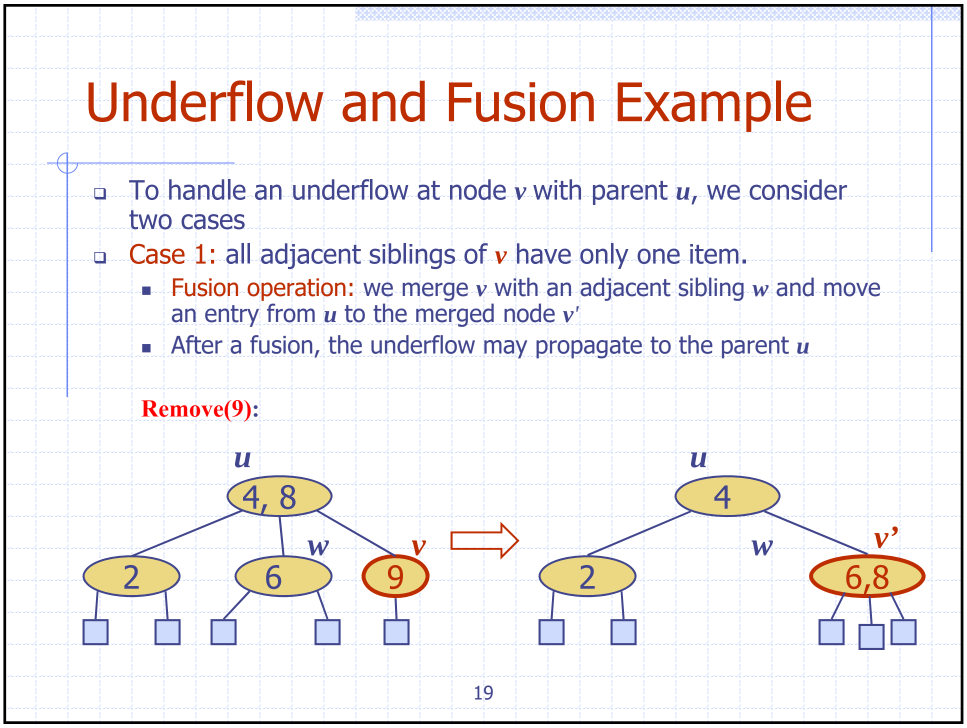
17



Underflow and Transfer

- To handle an underflow at node *v* with parent *u*, we consider two cases:
- Case 1: all adjacent siblings of *v* have only one item.
- Case 2: an adjacent sibling *w* of *v* has more than 1 item.
  - Transfer operation: (always keep the search tree property)
    1. we move a child branch of *w* to *v*
    2. we move an item from *u* to *v*
    3. we move an item from *w* to *u*
  - After a transfer, no underflow occurs

18



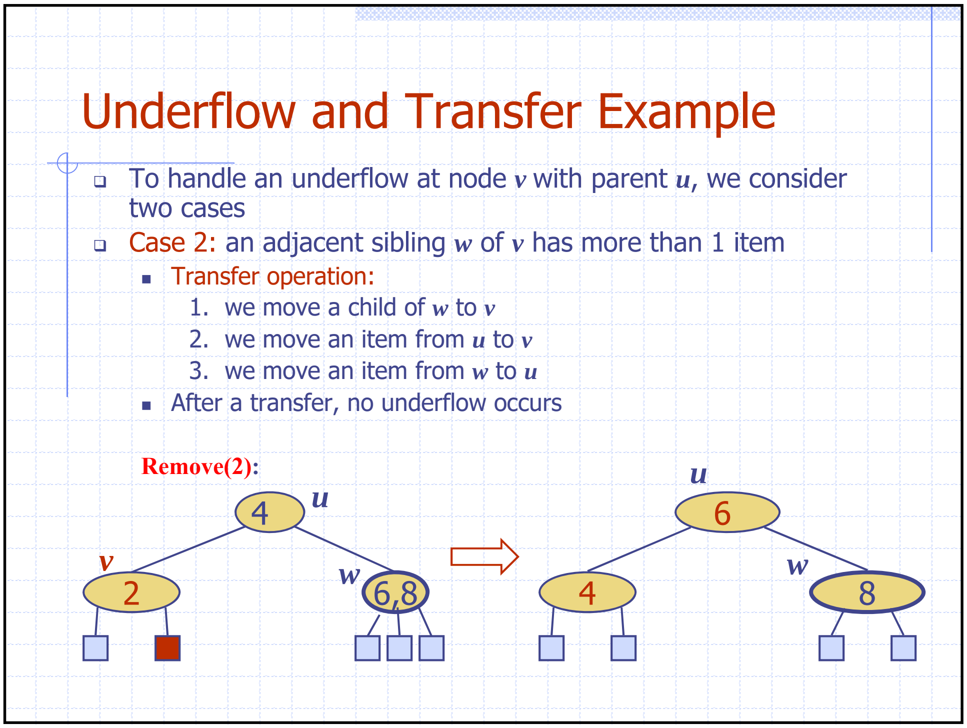
Underflow and Fusion Example

- To handle an underflow at node ***v*** with parent ***u***, we consider two cases
- Case 1: all adjacent siblings of ***v*** have only one item.
  - Fusion operation: we merge ***v*** with an adjacent sibling ***w*** and move an entry from ***u*** to the merged node ***v***′
  - After a fusion, the underflow may propagate to the parent ***u***

Remove(9):

19

19

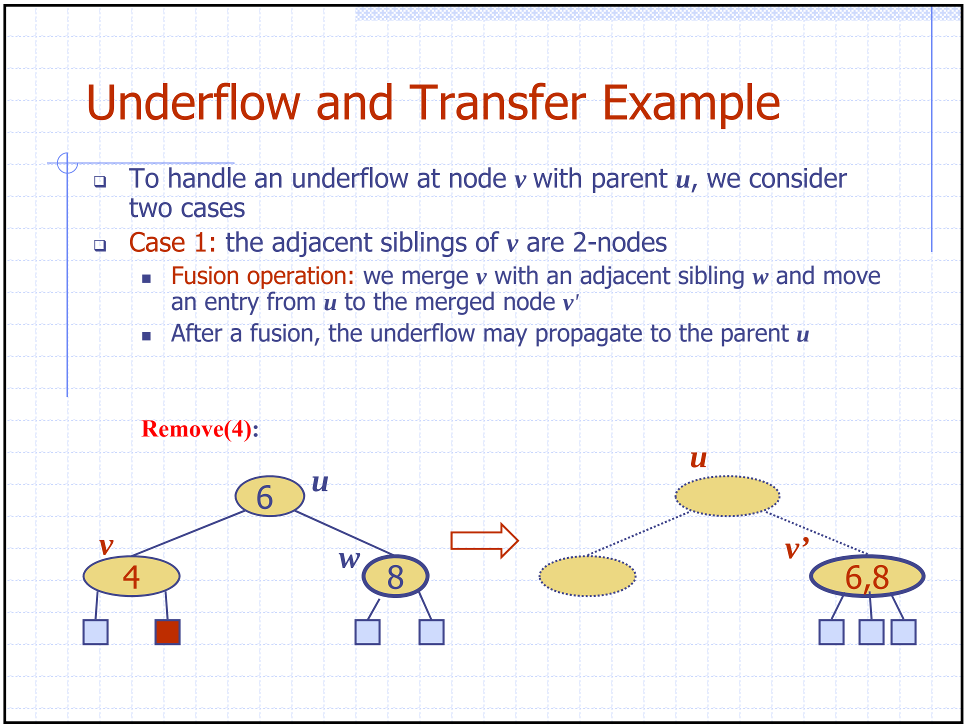


Underflow and Transfer Example

- To handle an underflow at node ***v*** with parent ***u***, we consider two cases
- Case 2: an adjacent sibling ***w*** of ***v*** has more than 1 item
  - Transfer operation:
    1. we move a child of ***w*** to ***v***
    2. we move an item from ***u*** to ***v***
    3. we move an item from ***w*** to ***u***
  - After a transfer, no underflow occurs

Remove(2):

20

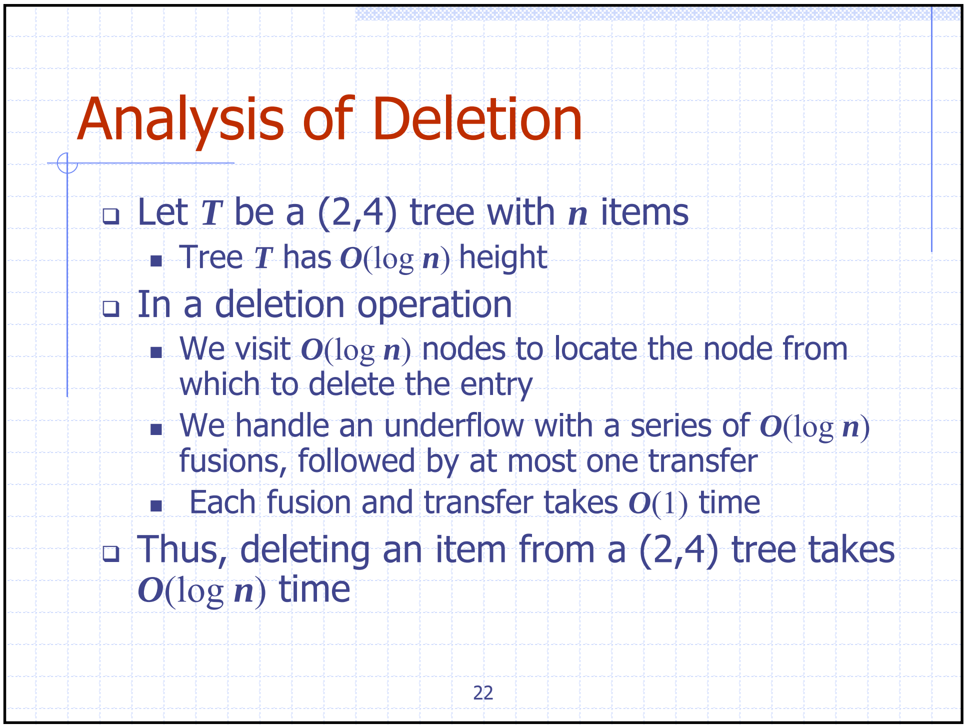


Underflow and Transfer Example

- To handle an underflow at node v with parent u, we consider two cases
- Case 1: the adjacent siblings of v are 2-nodes
  - Fusion operation: we merge v with an adjacent sibling w and move an entry from u to the merged node v'
  - After a fusion, the underflow may propagate to the parent u

Remove(4):

21

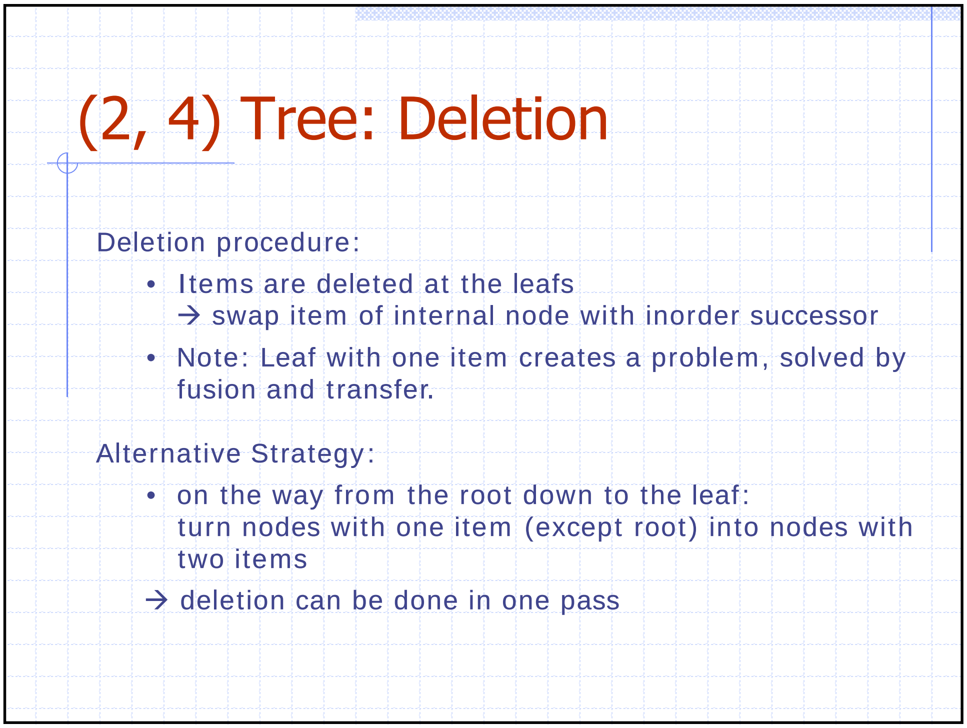


Analysis of Deletion

- Let T be a (2,4) tree with n items
  - Tree T has $O(\log n)$ height
- In a deletion operation
  - We visit $O(\log n)$ nodes to locate the node from which to delete the entry
  - We handle an underflow with a series of $O(\log n)$ fusions, followed by at most one transfer
  - Each fusion and transfer takes $O(1)$ time
- Thus, deleting an item from a (2,4) tree takes $O(\log n)$ time

22

22



(2, 4) Tree: Deletion

Deletion procedure:

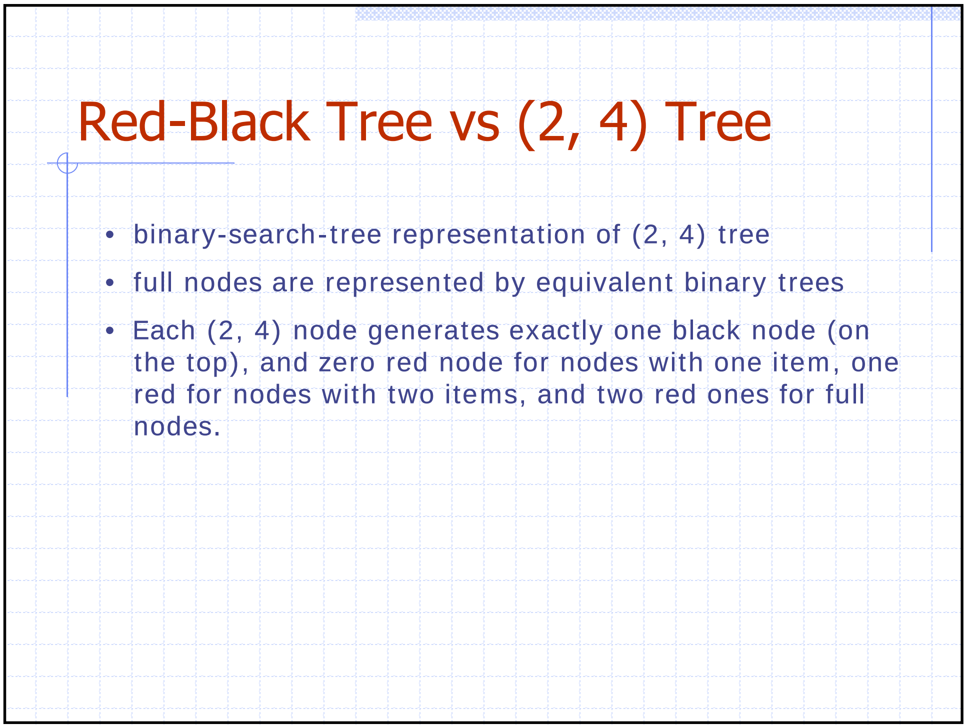
- Items are deleted at the leafs
  → swap item of internal node with inorder successor
- Note: Leaf with one item creates a problem, solved by fusion and transfer.

Alternative Strategy:

- on the way from the root down to the leaf:
  turn nodes with one item (except root) into nodes with two items

→ deletion can be done in one pass

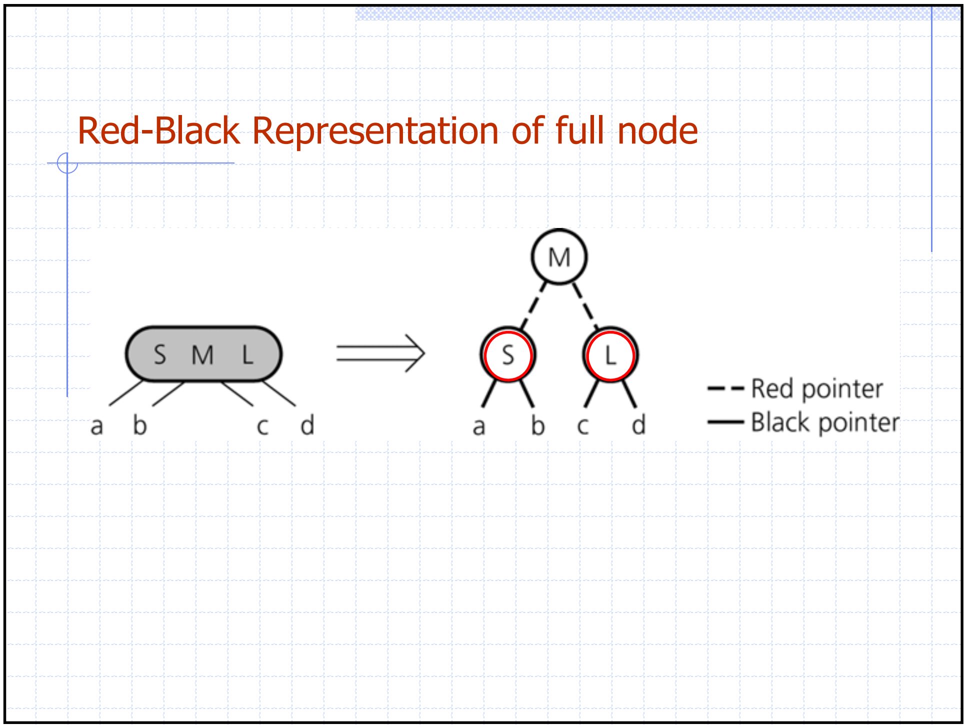
23



Red-Black Tree vs (2, 4) Tree

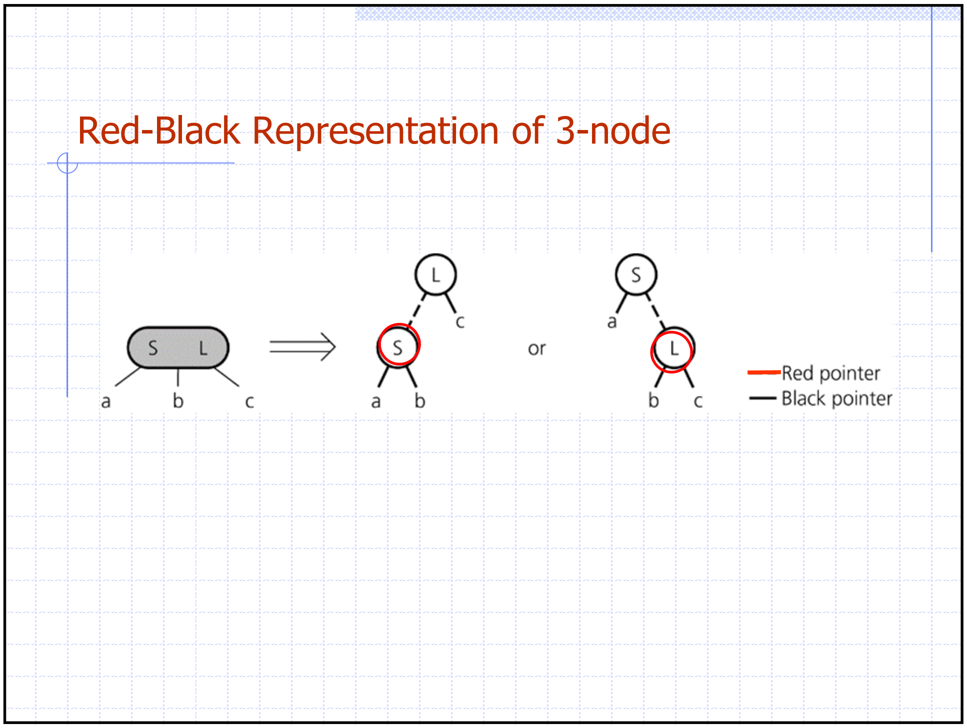
- binary-search-tree representation of (2, 4) tree
- full nodes are represented by equivalent binary trees
- Each (2, 4) node generates exactly one black node (on the top), and zero red node for nodes with one item, one red for nodes with two items, and two red ones for full nodes.

24



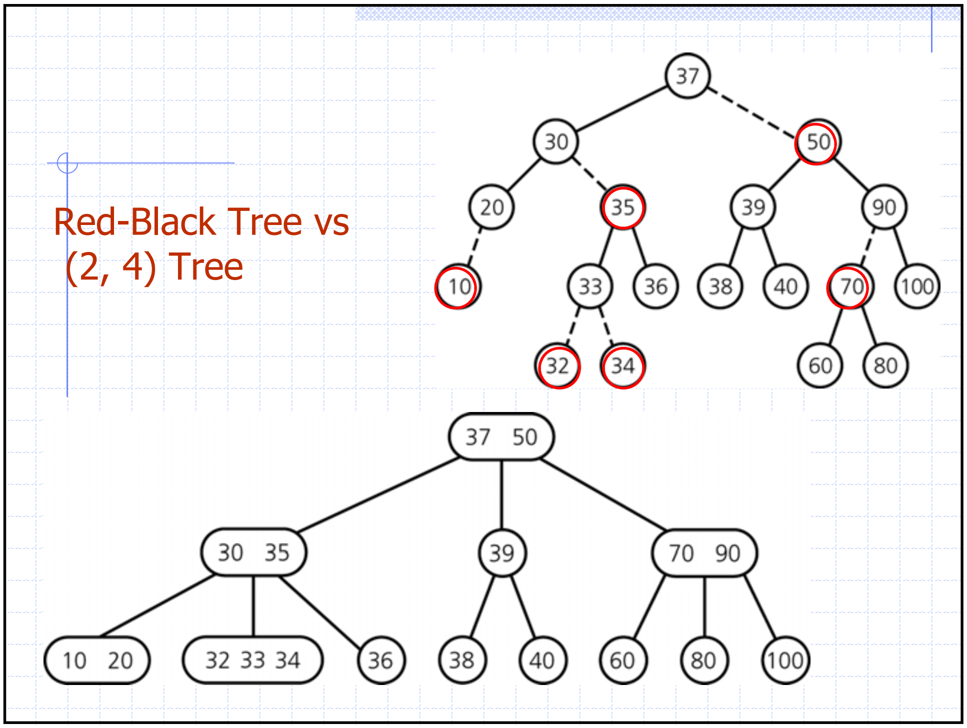
Red-Black Representation of full node

25



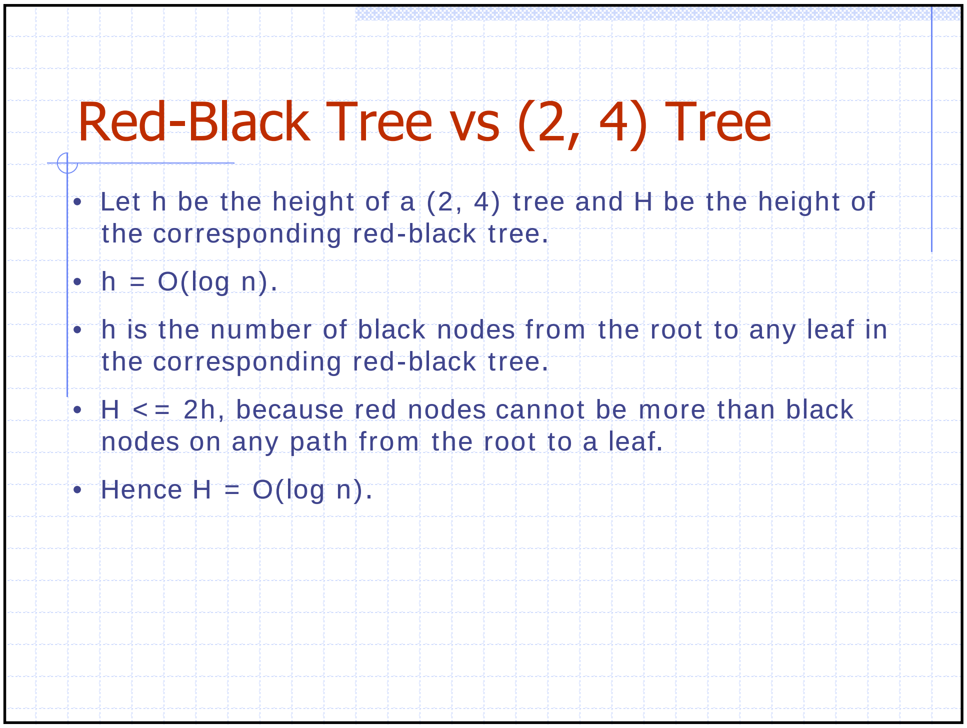
Red-Black Representation of 3-node

26



Red-Black Tree vs (2, 4) Tree

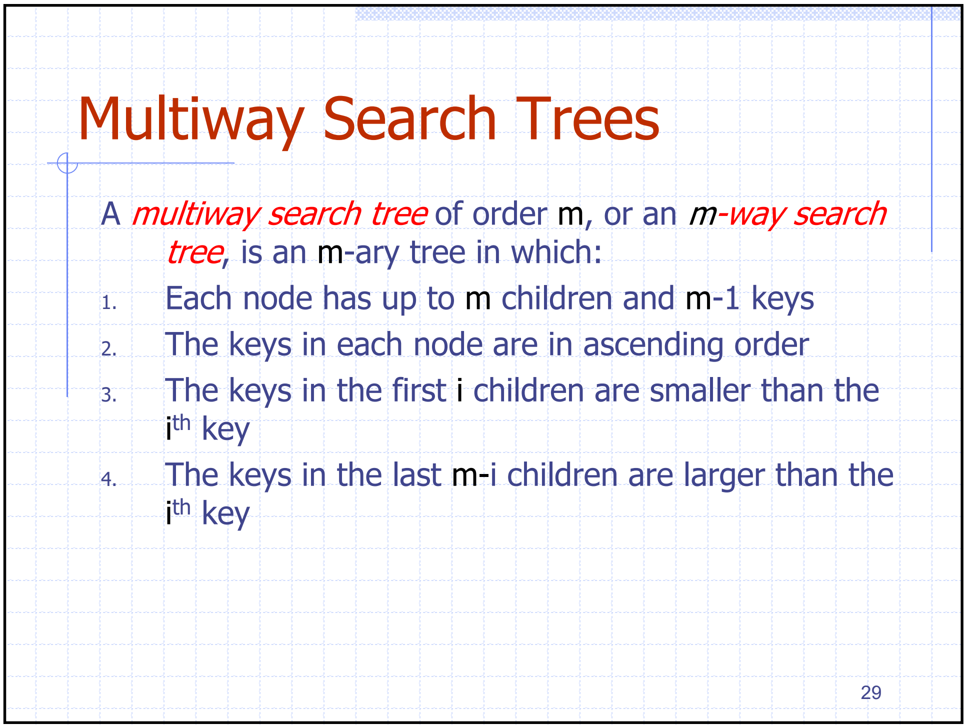
27



Red-Black Tree vs (2, 4) Tree

- Let h be the height of a (2, 4) tree and H be the height of the corresponding red-black tree.
- h = O(log n).
- h is the number of black nodes from the root to any leaf in the corresponding red-black tree.
- H <= 2h, because red nodes cannot be more than black nodes on any path from the root to a leaf.
- Hence H = O(log n).

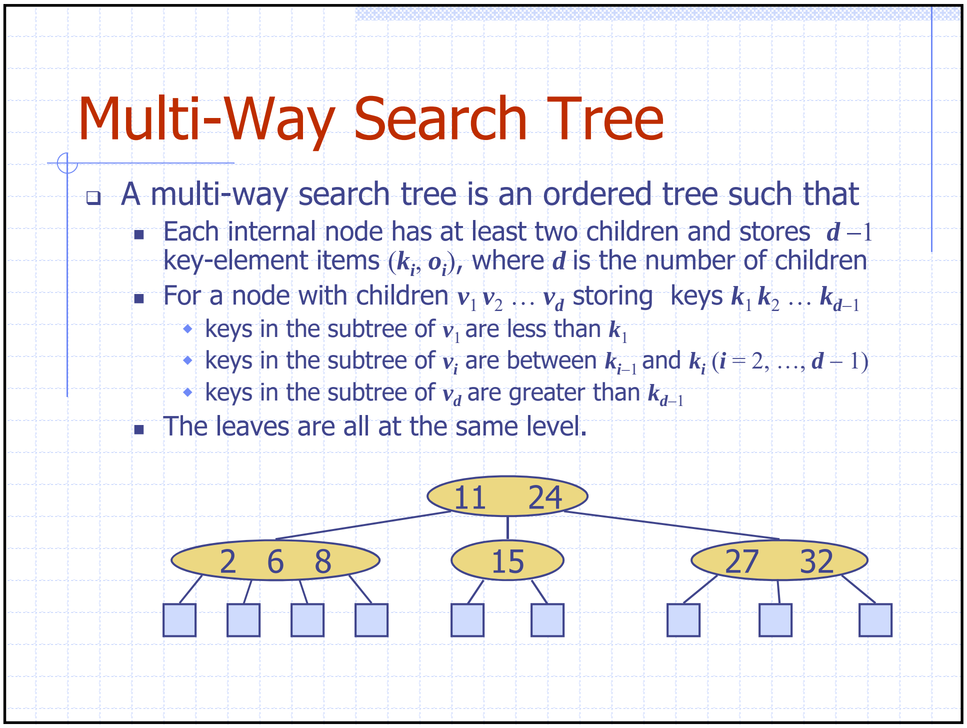
28



Multiway Search Trees

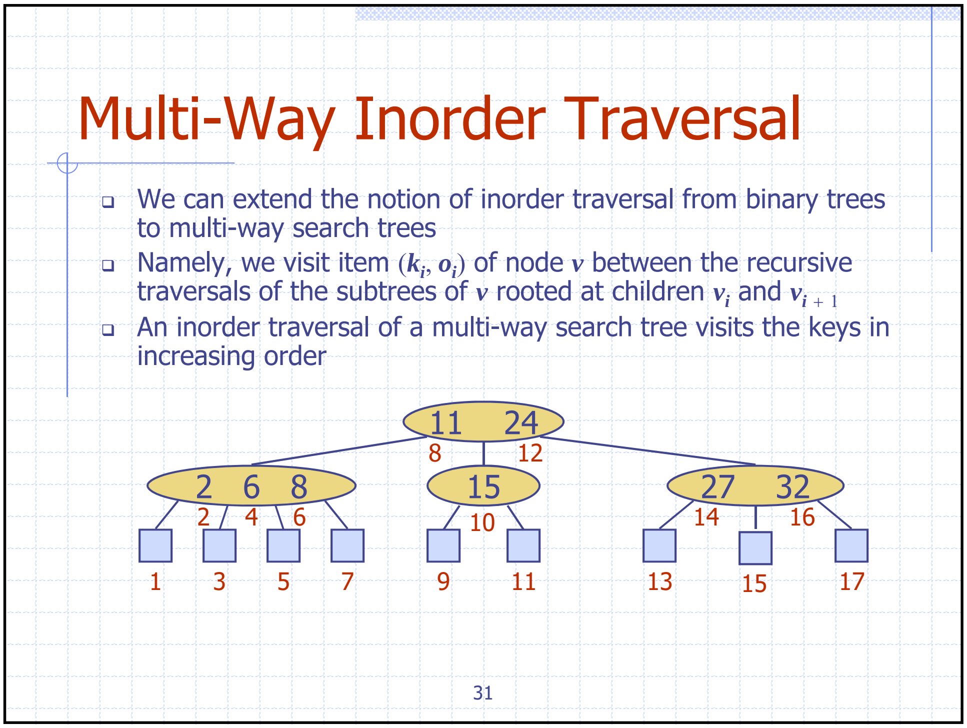
A *multiway search tree* of order m, or an *m-way search tree*, is an m-ary tree in which:

1. Each node has up to m children and m-1 keys
2. The keys in each node are in ascending order
3. The keys in the first i children are smaller than the i^{th} key
4. The keys in the last m-i children are larger than the i^{th} key



Multi-Way Search Tree

- A multi-way search tree is an ordered tree such that
  - Each internal node has at least two children and stores $d-1$ key-element items (k_i, o_i), where d is the number of children
  - For a node with children $v_1\ v_2 \ldots v_d$ storing keys $k_1\ k_2 \ldots k_{d-1}$
    - keys in the subtree of v_1 are less than k_1
    - keys in the subtree of v_i are between k_{i-1} and k_i $(i = 2, \ldots, d-1)$
    - keys in the subtree of v_d are greater than k_{d-1}
  - The leaves are all at the same level.



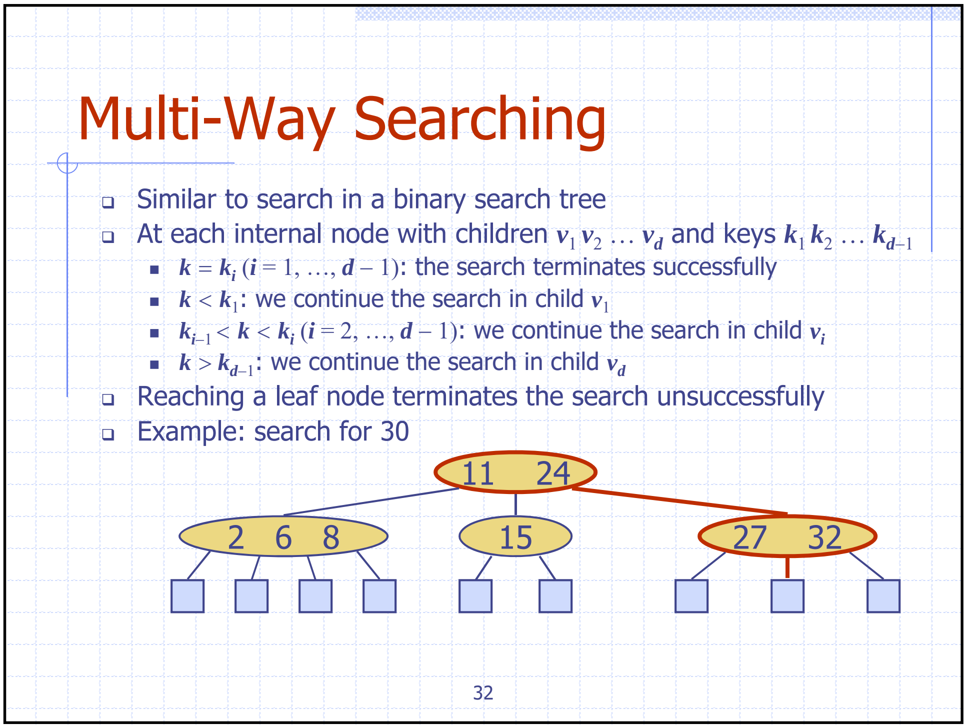
Multi-Way Inorder Traversal

- We can extend the notion of inorder traversal from binary trees to multi-way search trees
- Namely, we visit item (k_i, o_i) of node v between the recursive traversals of the subtrees of v rooted at children v_i and v_{i+1}
- An inorder traversal of a multi-way search tree visits the keys in increasing order

11 24
8
12
2 6 8
2
4
6
15
10
27 32
14
16
1
3
5
7
9
11
13
15
17

31

31

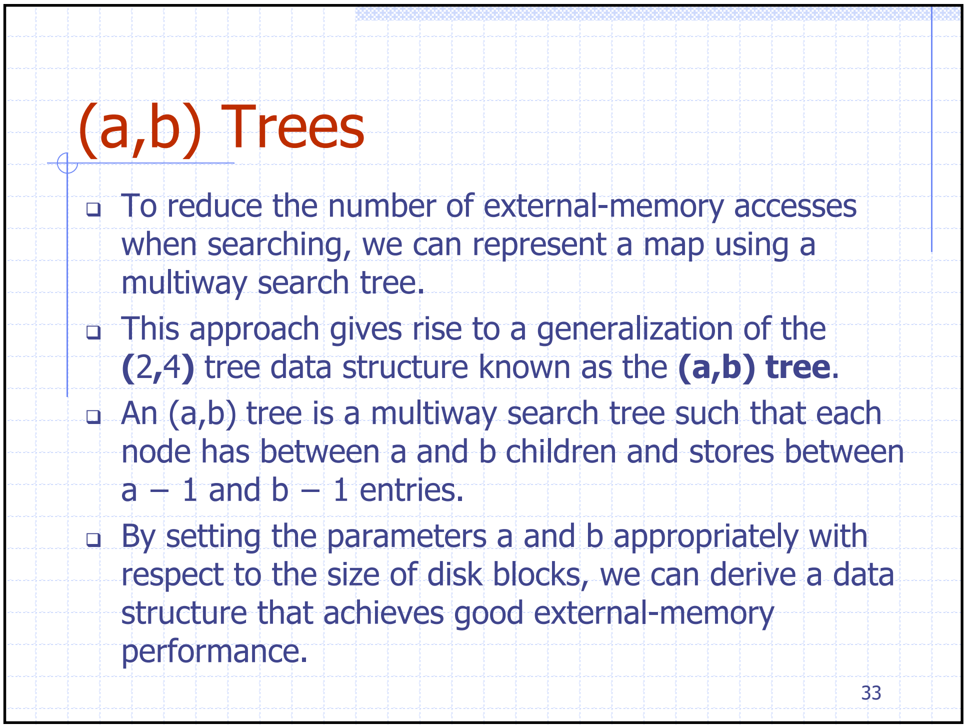


Multi-Way Searching

- Similar to search in a binary search tree
- At each internal node with children $v_1 v_2 \dots v_d$ and keys $k_1 k_2 \dots k_{d-1}$
  - $k = k_i$ $(i = 1, \dots, d-1)$: the search terminates successfully
  - $k < k_1$: we continue the search in child v_1
  - $k_{i-1} < k < k_i$ $(i = 2, \dots, d-1)$: we continue the search in child v_i
  - $k > k_{d-1}$: we continue the search in child v_d
- Reaching a leaf node terminates the search unsuccessfully
- Example: search for 30

32

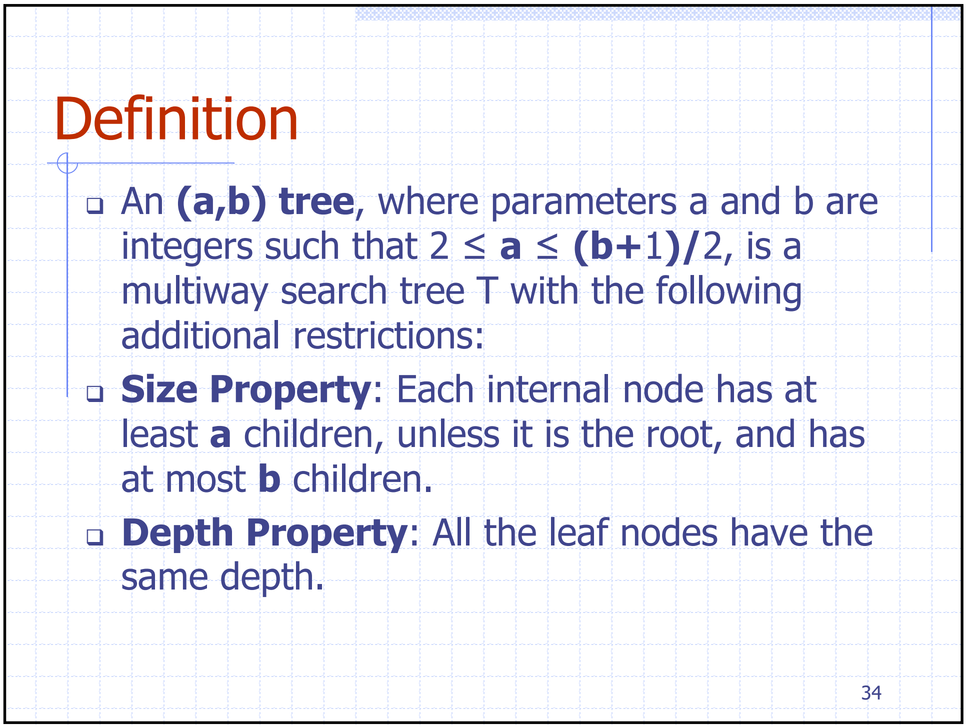
32



(a,b) Trees

- To reduce the number of external-memory accesses when searching, we can represent a map using a multiway search tree.
- This approach gives rise to a generalization of the **(**2,4**)** tree data structure known as the **(a,b) tree**.
- An (a,b) tree is a multiway search tree such that each node has between a and b children and stores between a − 1 and b − 1 entries.
- By setting the parameters a and b appropriately with respect to the size of disk blocks, we can derive a data structure that achieves good external-memory performance.

33

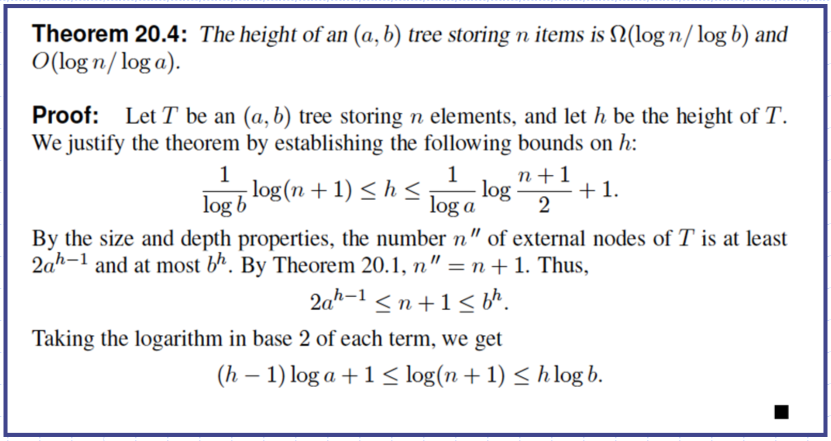


Definition

- An **(a,b) tree**, where parameters a and b are integers such that 2 ≤ **a** ≤ **(b+**1**)/**2, is a multiway search tree T with the following additional restrictions:
- **Size Property**: Each internal node has at least **a** children, unless it is the root, and has at most **b** children.
- **Depth Property**: All the leaf nodes have the same depth.

34

Height of an (a,b) Tree



Theorem 20.4: *The height of an (a, b) tree storing n items is $\Omega(\log n/\log b)$ and $O(\log n/\log a)$.*

Proof: Let T be an (a, b) tree storing n elements, and let h be the height of T. We justify the theorem by establishing the following bounds on h:

$$\frac{1}{\log b}\log(n+1) \le h \le \frac{1}{\log a}\log\frac{n+1}{2}+1.$$

By the size and depth properties, the number n'' of external nodes of T is at least $2a^{h-1}$ and at most b^h. By Theorem 20.1, $n'' = n+1$. Thus,

$$2a^{h-1} \le n+1 \le b^h.$$

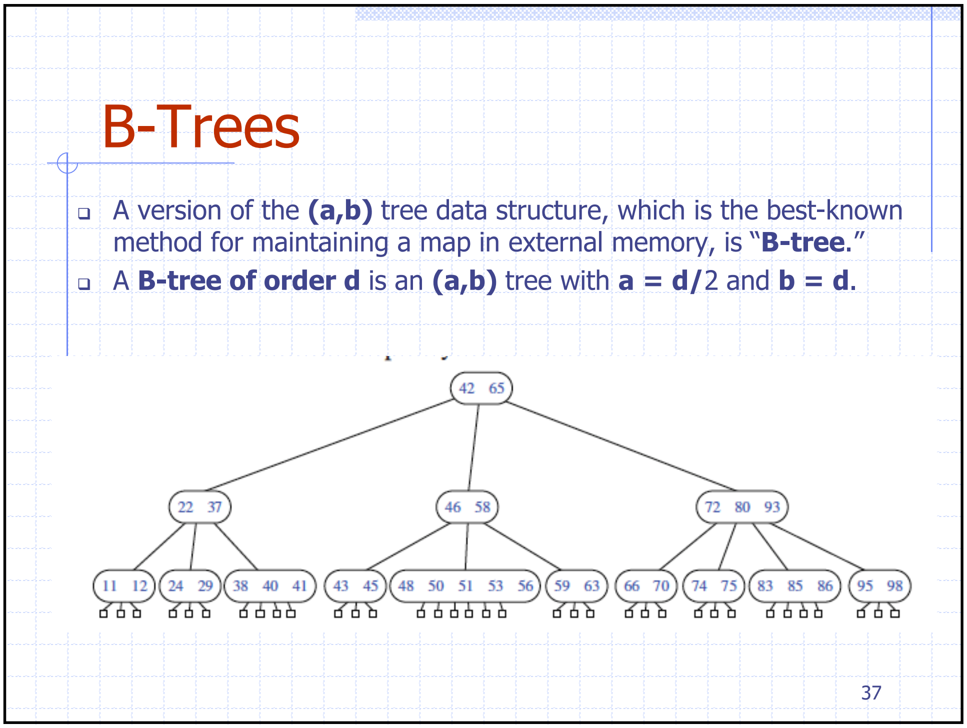
Taking the logarithm in base 2 of each term, we get

$$(h-1)\log a + 1 \le \log(n+1) \le h\log b.$$

■

Searches and Updates

- The search algorithm in an **(a,b)** tree is exactly like the one for multiway search trees.
- The insertion algorithm for an **(a,b)** tree is similar to that for a **(**2,4**)** tree.
  - An overflow occurs when an entry is inserted into a **b**-node **w**, which becomes an illegal **(b+**1**)**-node.
  - To remedy an overflow, we split node w by moving the median entry of w into the parent of w and replacing w with a **(b+**1**)/**2-node w and a **(b+**1**)/**2-node w.
- Removing an entry from an **(a,b)** tree is similar to what was done for **(**2,4**)** trees.
  - An underflow occurs when a key is removed from an **a**-node **w**, distinct from the root, which causes **w** to become an **(a−**1**)**-node.
  - To remedy an underflow, we perform a transfer with a sibling of **w** that is not an **a**-node or we perform a fusion of **w** with a sibling that is an **a**-node.

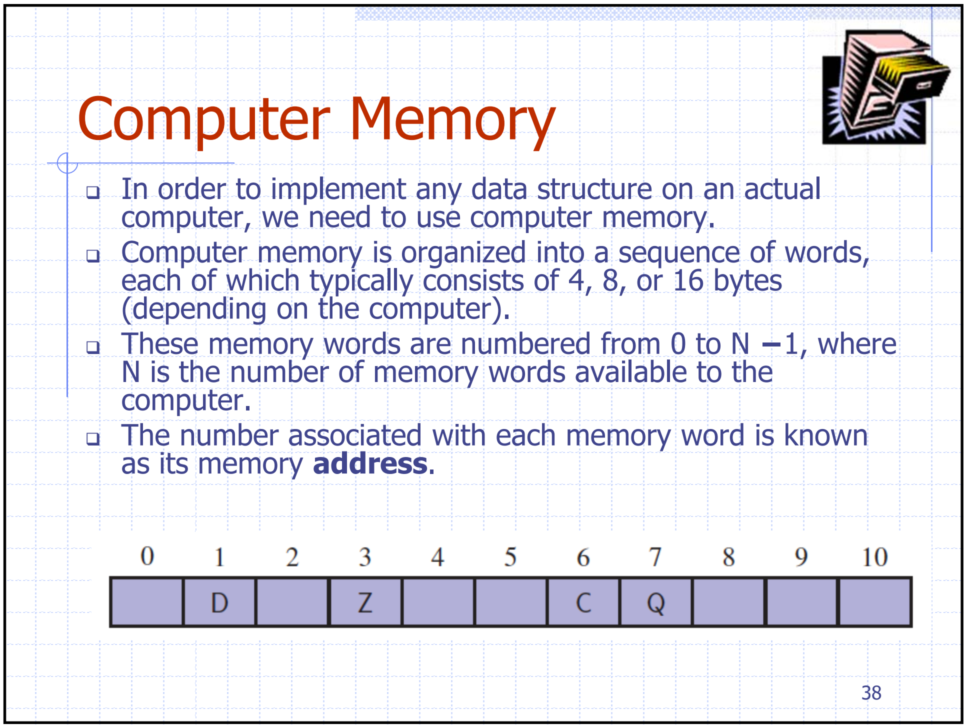


B-Trees

- A version of the **(a,b)** tree data structure, which is the best-known method for maintaining a map in external memory, is "**B-tree.**"
- A **B-tree of order d** is an **(a,b)** tree with **a = d/**2 and **b = d**.

37

37



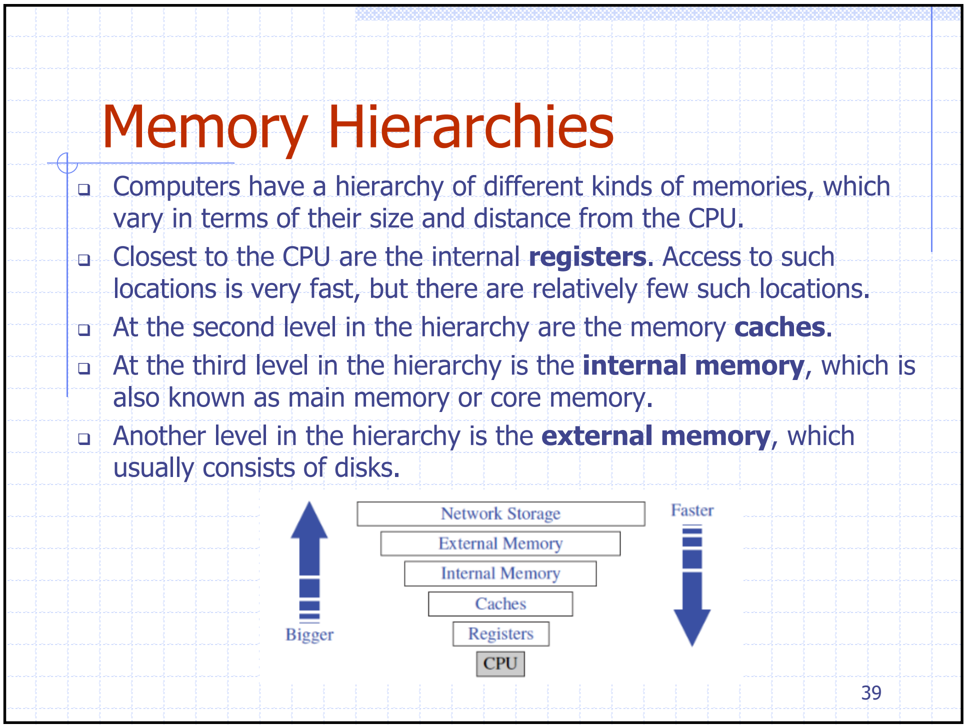
Computer Memory

- In order to implement any data structure on an actual computer, we need to use computer memory.
- Computer memory is organized into a sequence of words, each of which typically consists of 4, 8, or 16 bytes (depending on the computer).
- These memory words are numbered from 0 to N −1, where N is the number of memory words available to the computer.
- The number associated with each memory word is known as its memory **address**.

0	1	2	3	4	5	6	7	8	9	10
	D		Z			C	Q			

38

38



Memory Hierarchies

- Computers have a hierarchy of different kinds of memories, which vary in terms of their size and distance from the CPU.
- Closest to the CPU are the internal **registers**. Access to such locations is very fast, but there are relatively few such locations.
- At the second level in the hierarchy are the memory **caches**.
- At the third level in the hierarchy is the **internal memory**, which is also known as main memory or core memory.
- Another level in the hierarchy is the **external memory**, which usually consists of disks.

Bigger

Network Storage

External Memory

Internal Memory

Caches

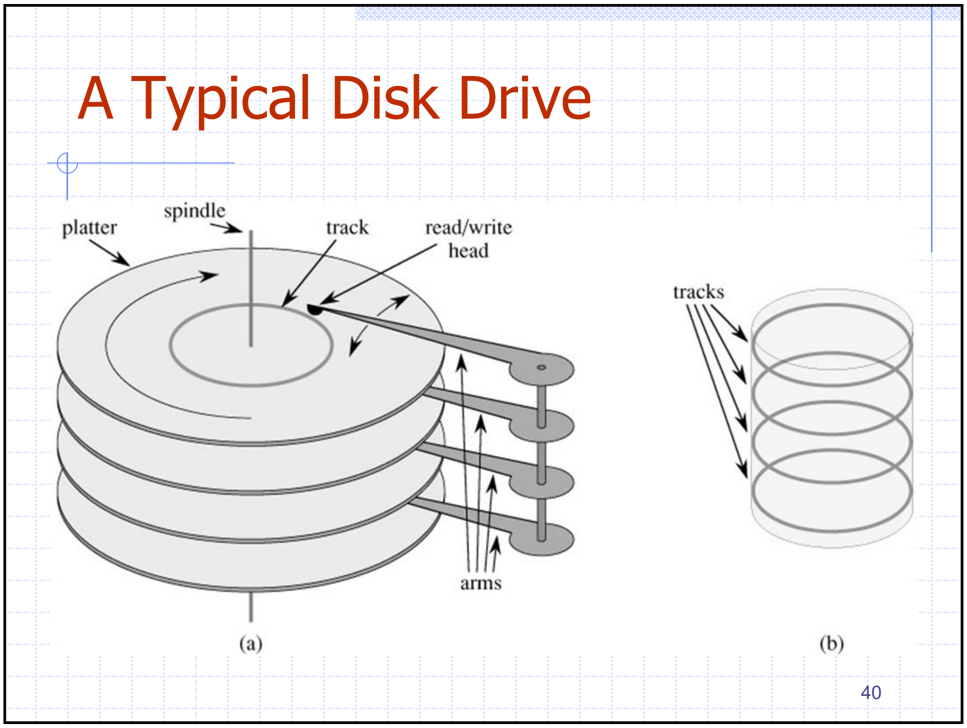
Registers

CPU

Faster

39

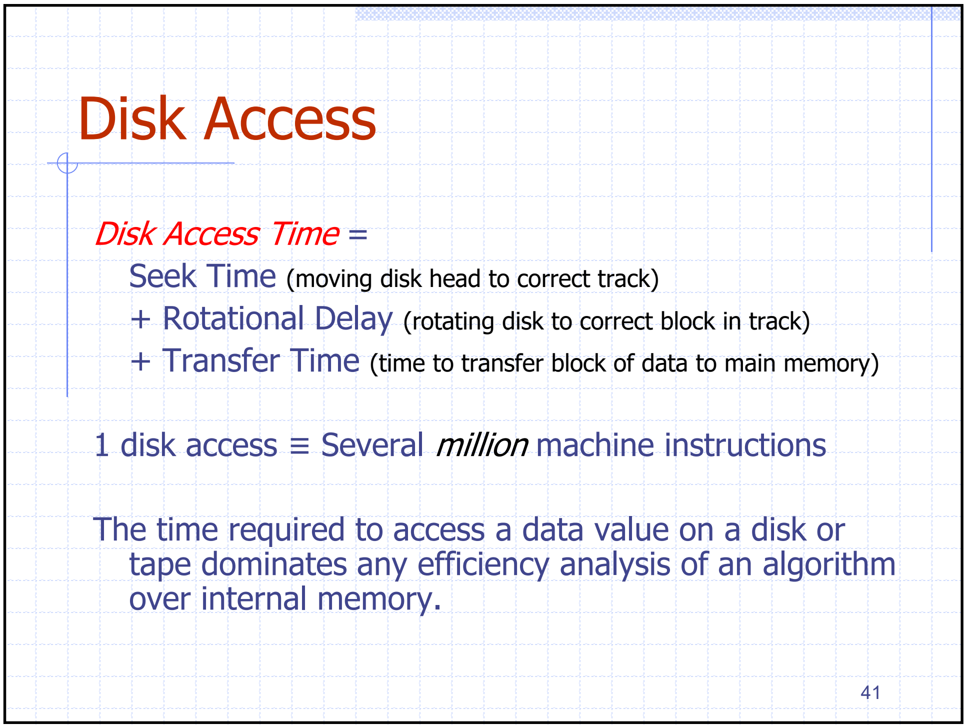
39



A Typical Disk Drive

40

40



Disk Access

Disk Access Time =

Seek Time (moving disk head to correct track)

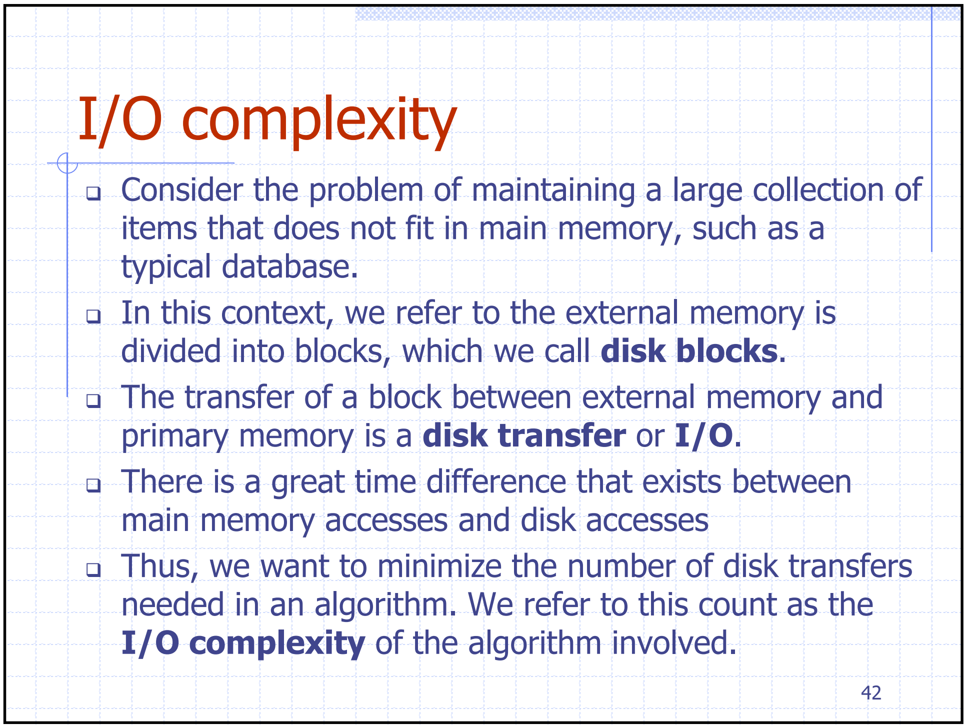
+ Rotational Delay (rotating disk to correct block in track)

+ Transfer Time (time to transfer block of data to main memory)

1 disk access ≡ Several *million* machine instructions

The time required to access a data value on a disk or tape dominates any efficiency analysis of an algorithm over internal memory.

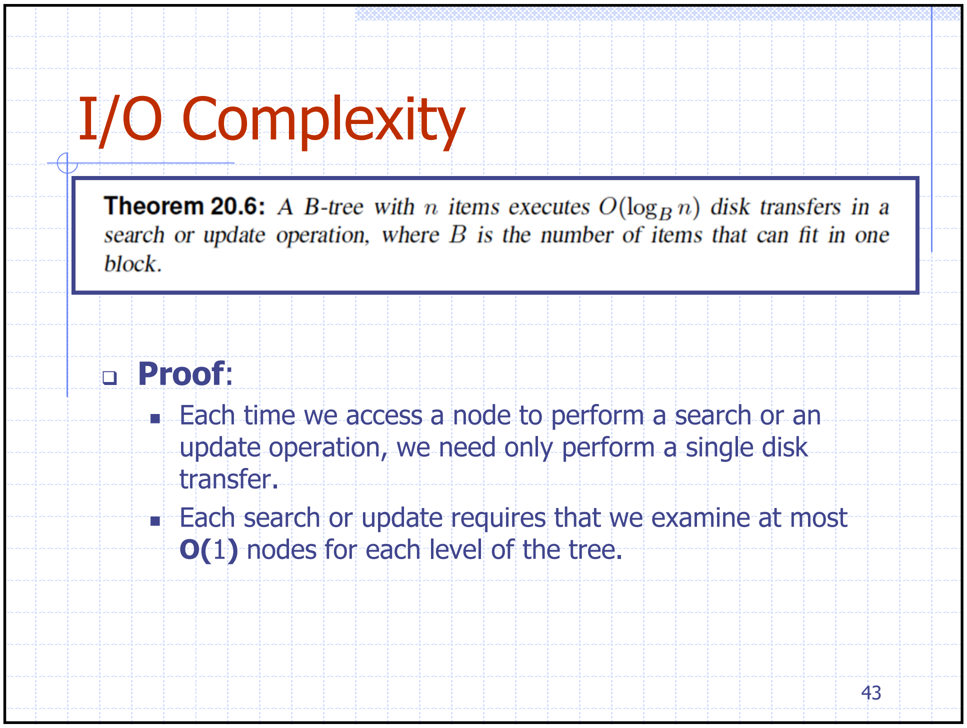
41



I/O complexity

- Consider the problem of maintaining a large collection of items that does not fit in main memory, such as a typical database.
- In this context, we refer to the external memory is divided into blocks, which we call **disk blocks**.
- The transfer of a block between external memory and primary memory is a **disk transfer** or **I/O**.
- There is a great time difference that exists between main memory accesses and disk accesses
- Thus, we want to minimize the number of disk transfers needed in an algorithm. We refer to this count as the **I/O complexity** of the algorithm involved.

42



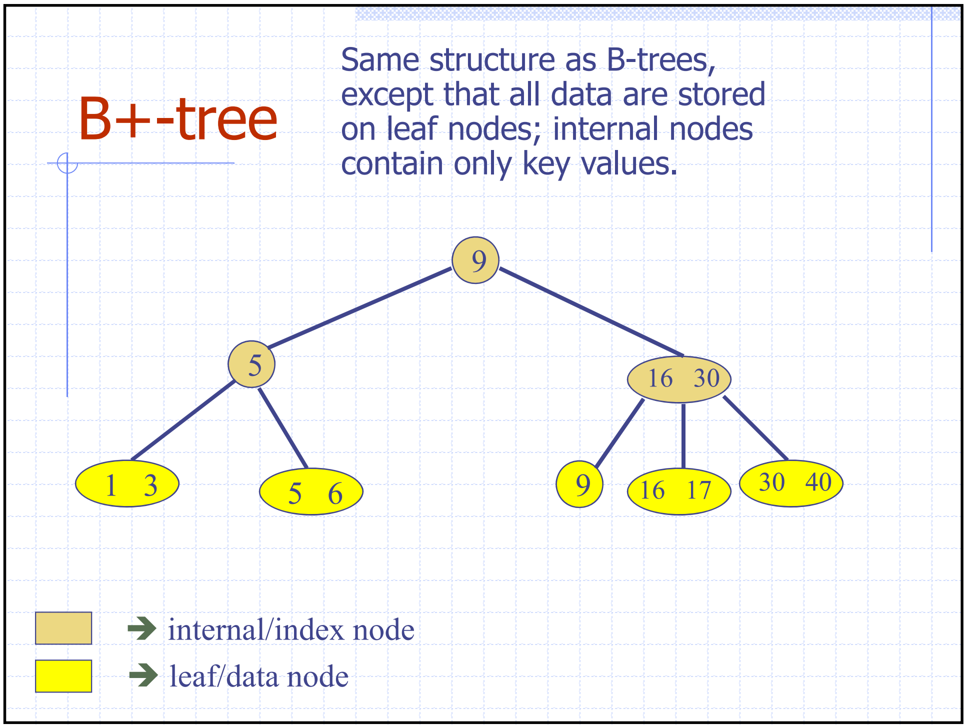
I/O Complexity

Theorem 20.6: *A B-tree with n items executes $O(\log_B n)$ disk transfers in a search or update operation, where B is the number of items that can fit in one block.*

- **Proof**:
  - Each time we access a node to perform a search or an update operation, we need only perform a single disk transfer.
  - Each search or update requires that we examine at most **O(**1**)** nodes for each level of the tree.

43

43



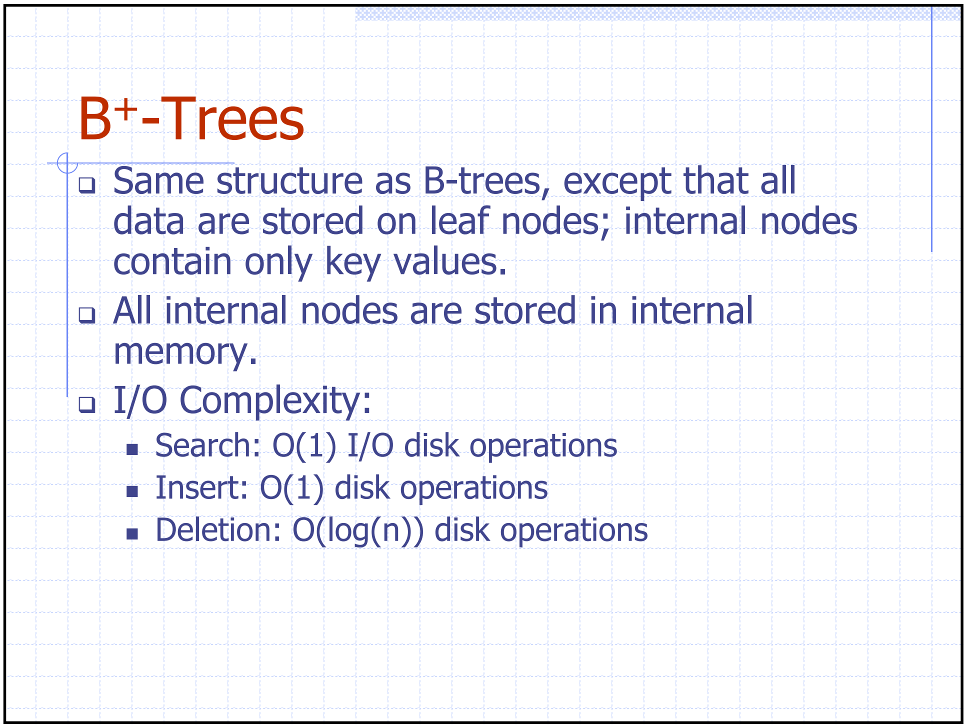
B+-tree

Same structure as B-trees, except that all data are stored on leaf nodes; internal nodes contain only key values.

▨ ➔ internal/index node

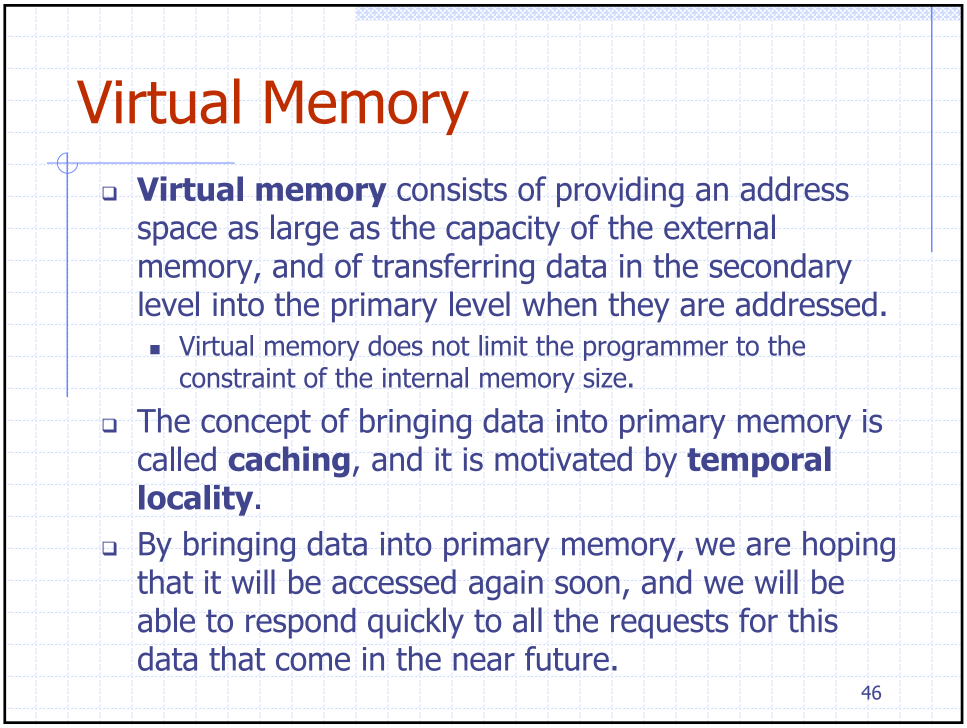
🟨 ➔ leaf/data node

44



B+-Trees

- Same structure as B-trees, except that all data are stored on leaf nodes; internal nodes contain only key values.
- All internal nodes are stored in internal memory.
- I/O Complexity:
  - Search: O(1) I/O disk operations
  - Insert: O(1) disk operations
  - Deletion: O(log(n)) disk operations



Virtual Memory

- **Virtual memory** consists of providing an address space as large as the capacity of the external memory, and of transferring data in the secondary level into the primary level when they are addressed.
  - Virtual memory does not limit the programmer to the constraint of the internal memory size.
- The concept of bringing data into primary memory is called **caching**, and it is motivated by **temporal locality**.
- By bringing data into primary memory, we are hoping that it will be accessed again soon, and we will be able to respond quickly to all the requests for this data that come in the near future.

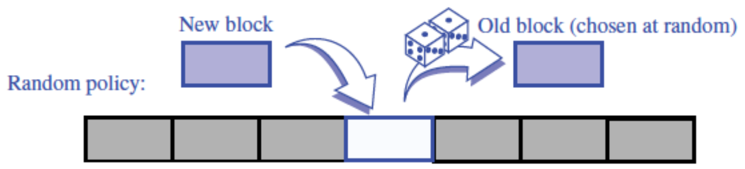
Page Replacement Strategies

- When a new block is referenced and the space for blocks from external memory is full, we must evict an existing block.
- There are several such **page replacement** strategies, including:
  - FIFO
  - LIFO
  - Random

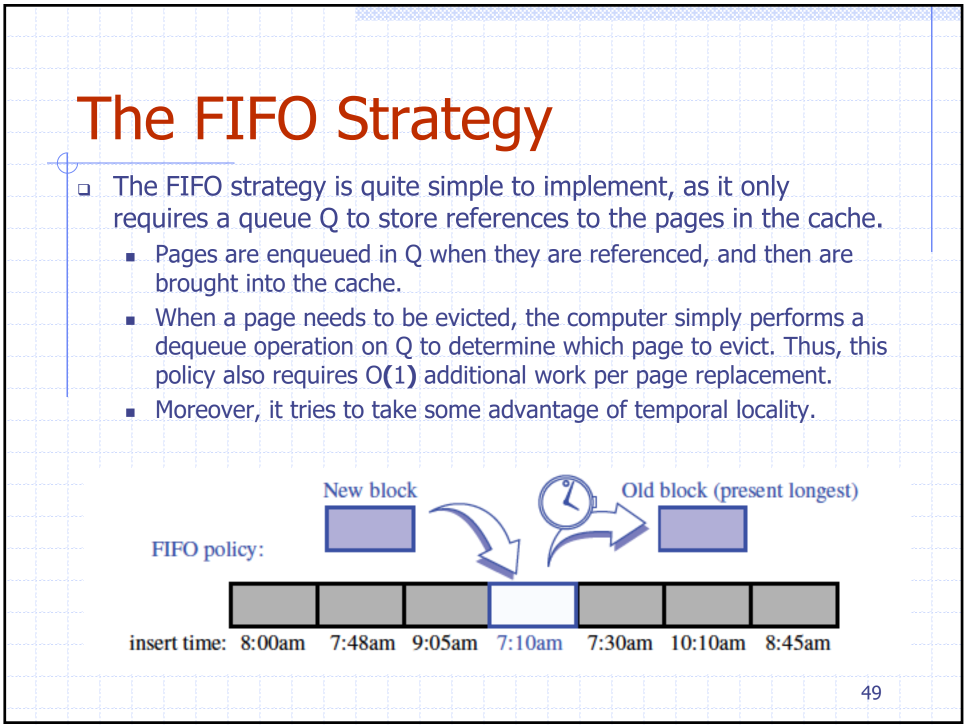
47

The Random Strategy

- Choose a page at random to evict from the cache.
  - The overhead involved in implementing this policy is an **O(**1**)** additional amount of work per page replacement.
  - Still, this policy makes no attempt to take advantage of any temporal locality exhibited by a user's browsing.



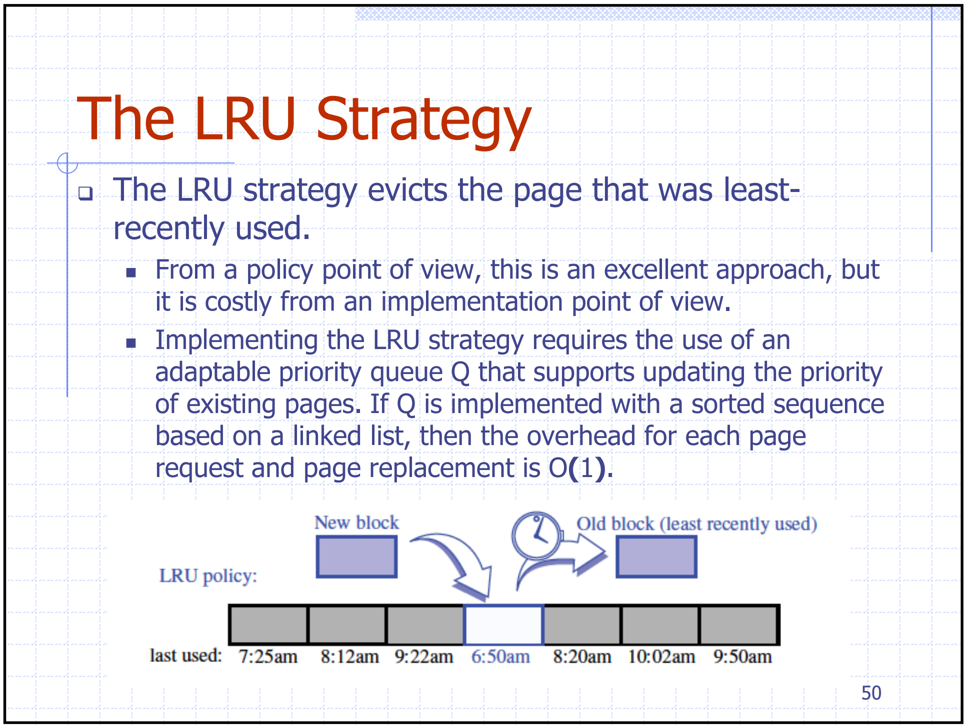
48



The FIFO Strategy

- The FIFO strategy is quite simple to implement, as it only requires a queue Q to store references to the pages in the cache.
  - Pages are enqueued in Q when they are referenced, and then are brought into the cache.
  - When a page needs to be evicted, the computer simply performs a dequeue operation on Q to determine which page to evict. Thus, this policy also requires O(1) additional work per page replacement.
  - Moreover, it tries to take some advantage of temporal locality.

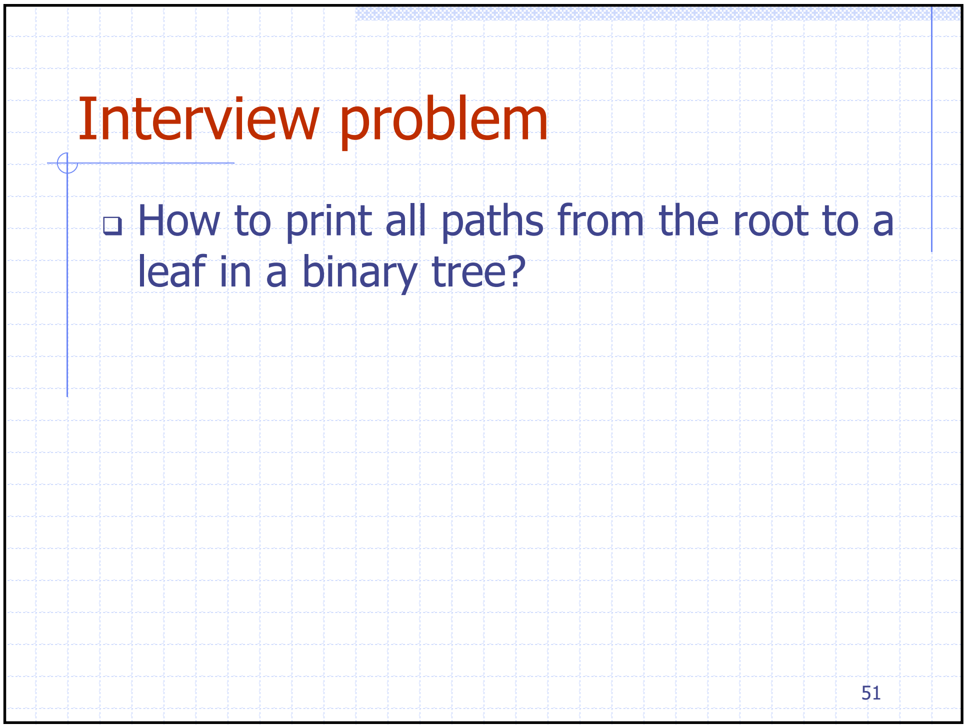
New block
Old block (present longest)
FIFO policy:
insert time: 8:00am 7:48am 9:05am 7:10am 7:30am 10:10am 8:45am



The LRU Strategy

- The LRU strategy evicts the page that was least-recently used.
  - From a policy point of view, this is an excellent approach, but it is costly from an implementation point of view.
  - Implementing the LRU strategy requires the use of an adaptable priority queue Q that supports updating the priority of existing pages. If Q is implemented with a sorted sequence based on a linked list, then the overhead for each page request and page replacement is O(1).

New block
Old block (least recently used)
LRU policy:
last used: 7:25am 8:12am 9:22am 6:50am 8:20am 10:02am 9:50am

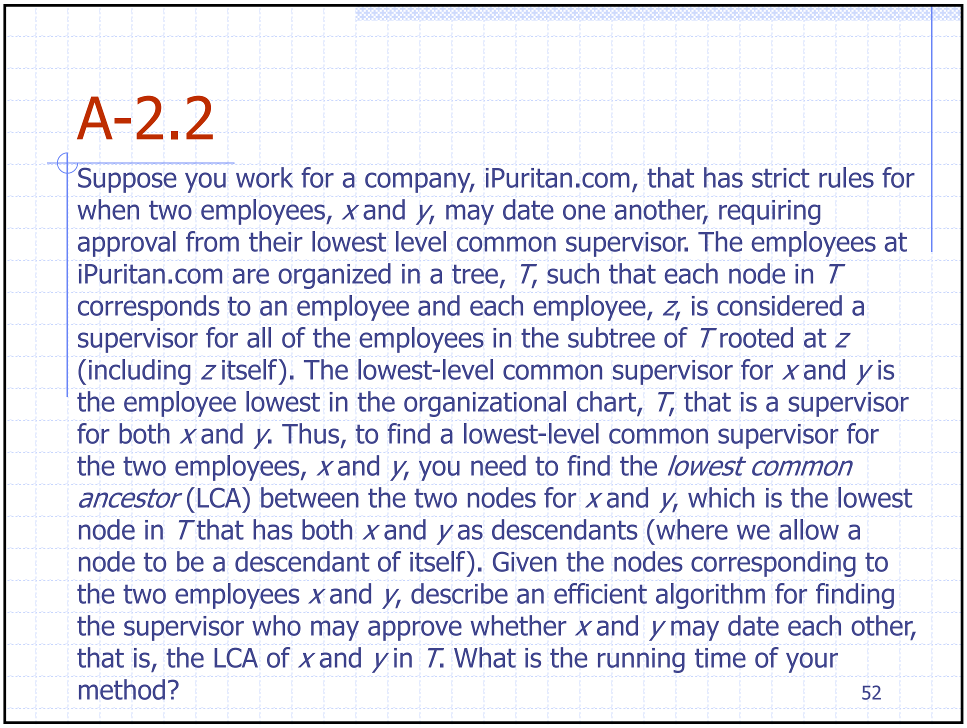


Interview problem

- How to print all paths from the root to a leaf in a binary tree?

51

51

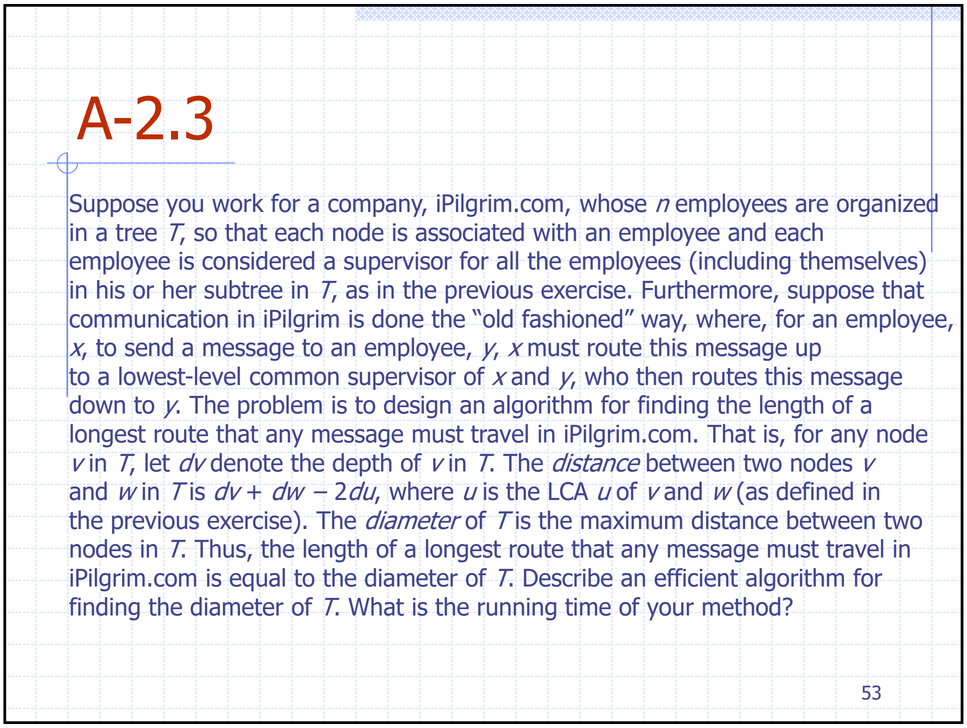


A-2.2

Suppose you work for a company, iPuritan.com, that has strict rules for when two employees, *x* and *y*, may date one another, requiring approval from their lowest level common supervisor. The employees at iPuritan.com are organized in a tree, *T*, such that each node in *T* corresponds to an employee and each employee, *z*, is considered a supervisor for all of the employees in the subtree of *T* rooted at *z* (including *z* itself). The lowest-level common supervisor for *x* and *y* is the employee lowest in the organizational chart, *T*, that is a supervisor for both *x* and *y*. Thus, to find a lowest-level common supervisor for the two employees, *x* and *y*, you need to find the *lowest common ancestor* (LCA) between the two nodes for *x* and *y*, which is the lowest node in *T* that has both *x* and *y* as descendants (where we allow a node to be a descendant of itself). Given the nodes corresponding to the two employees *x* and *y*, describe an efficient algorithm for finding the supervisor who may approve whether *x* and *y* may date each other, that is, the LCA of *x* and *y* in *T*. What is the running time of your method?

52

52

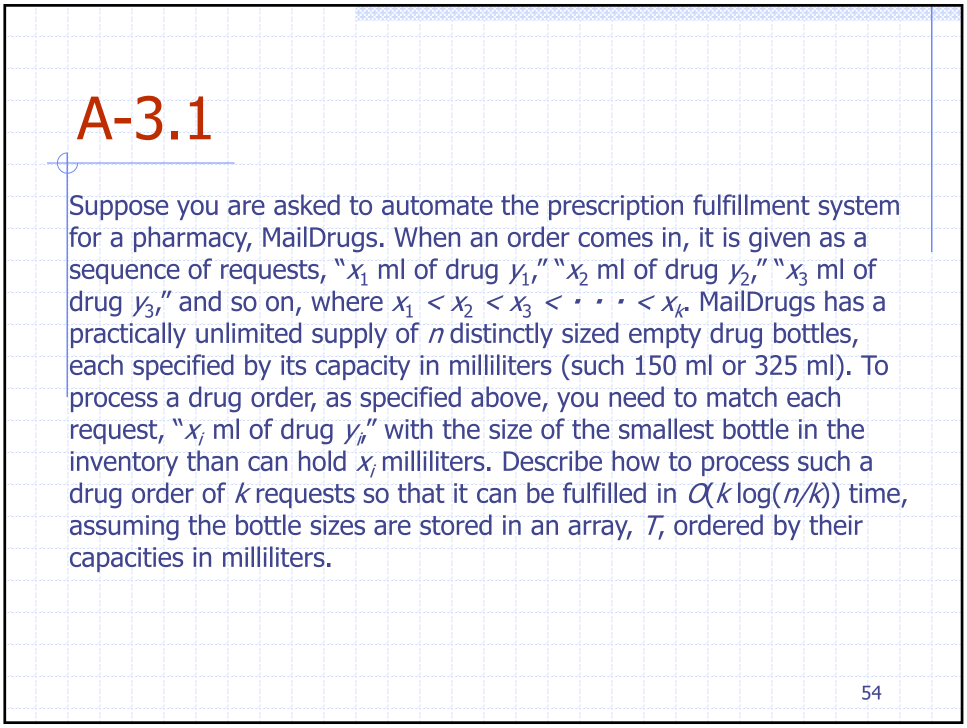


A-2.3

Suppose you work for a company, iPilgrim.com, whose n employees are organized in a tree T, so that each node is associated with an employee and each employee is considered a supervisor for all the employees (including themselves) in his or her subtree in T, as in the previous exercise. Furthermore, suppose that communication in iPilgrim is done the "old fashioned" way, where, for an employee, x, to send a message to an employee, y, x must route this message up to a lowest-level common supervisor of x and y, who then routes this message down to y. The problem is to design an algorithm for finding the length of a longest route that any message must travel in iPilgrim.com. That is, for any node v in T, let dv denote the depth of v in T. The *distance* between two nodes v and w in T is $dv + dw - 2du$, where u is the LCA u of v and w (as defined in the previous exercise). The *diameter* of T is the maximum distance between two nodes in T. Thus, the length of a longest route that any message must travel in iPilgrim.com is equal to the diameter of T. Describe an efficient algorithm for finding the diameter of T. What is the running time of your method?

53

53

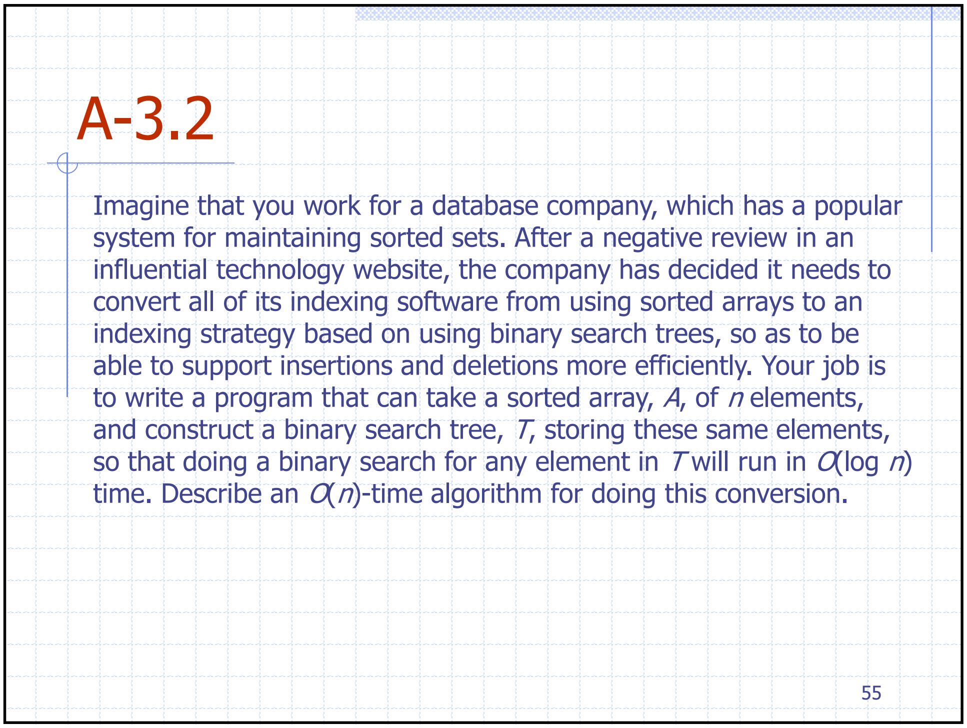


A-3.1

Suppose you are asked to automate the prescription fulfillment system for a pharmacy, MailDrugs. When an order comes in, it is given as a sequence of requests, "x_1 ml of drug y_1," "x_2 ml of drug y_2," "x_3 ml of drug y_3," and so on, where $x_1 < x_2 < x_3 < \cdots < x_k$. MailDrugs has a practically unlimited supply of n distinctly sized empty drug bottles, each specified by its capacity in milliliters (such 150 ml or 325 ml). To process a drug order, as specified above, you need to match each request, "x_i ml of drug y_i," with the size of the smallest bottle in the inventory than can hold x_i milliliters. Describe how to process such a drug order of k requests so that it can be fulfilled in $O(k \log(n/k))$ time, assuming the bottle sizes are stored in an array, T, ordered by their capacities in milliliters.

54

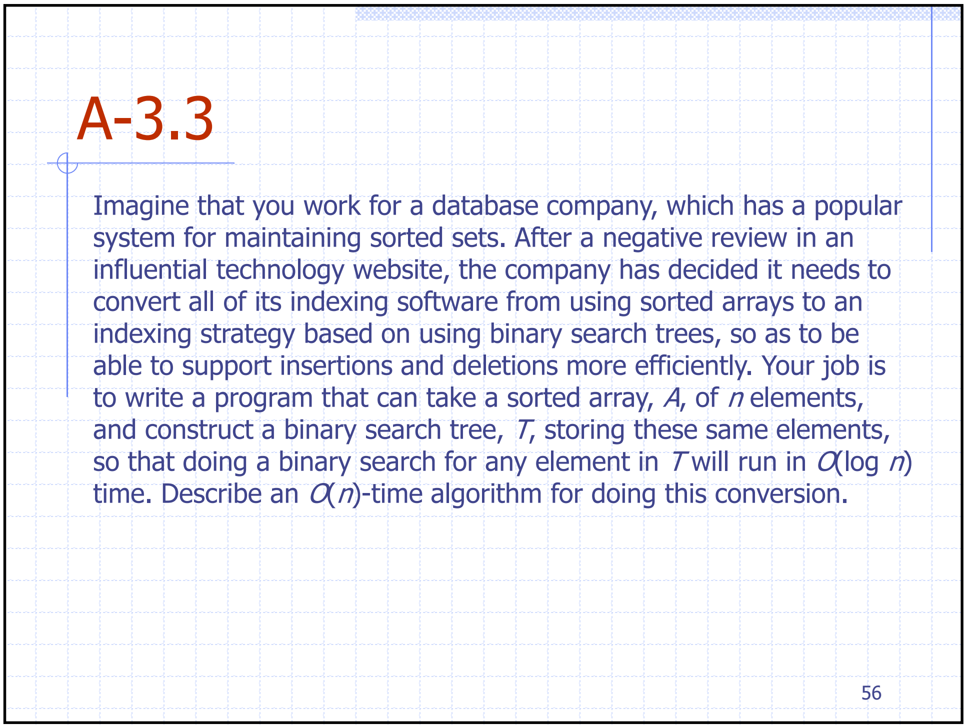
54



A-3.2

Imagine that you work for a database company, which has a popular system for maintaining sorted sets. After a negative review in an influential technology website, the company has decided it needs to convert all of its indexing software from using sorted arrays to an indexing strategy based on using binary search trees, so as to be able to support insertions and deletions more efficiently. Your job is to write a program that can take a sorted array, *A*, of *n* elements, and construct a binary search tree, *T*, storing these same elements, so that doing a binary search for any element in *T* will run in *O*(log *n*) time. Describe an *O*(*n*)-time algorithm for doing this conversion.

55



A-3.3

Imagine that you work for a database company, which has a popular system for maintaining sorted sets. After a negative review in an influential technology website, the company has decided it needs to convert all of its indexing software from using sorted arrays to an indexing strategy based on using binary search trees, so as to be able to support insertions and deletions more efficiently. Your job is to write a program that can take a sorted array, *A*, of *n* elements, and construct a binary search tree, *T*, storing these same elements, so that doing a binary search for any element in *T* will run in *O*(log *n*) time. Describe an *O*(*n*)-time algorithm for doing this conversion.

56