COT 6405
ANALYSIS OF ALGORITHMS

B-Trees

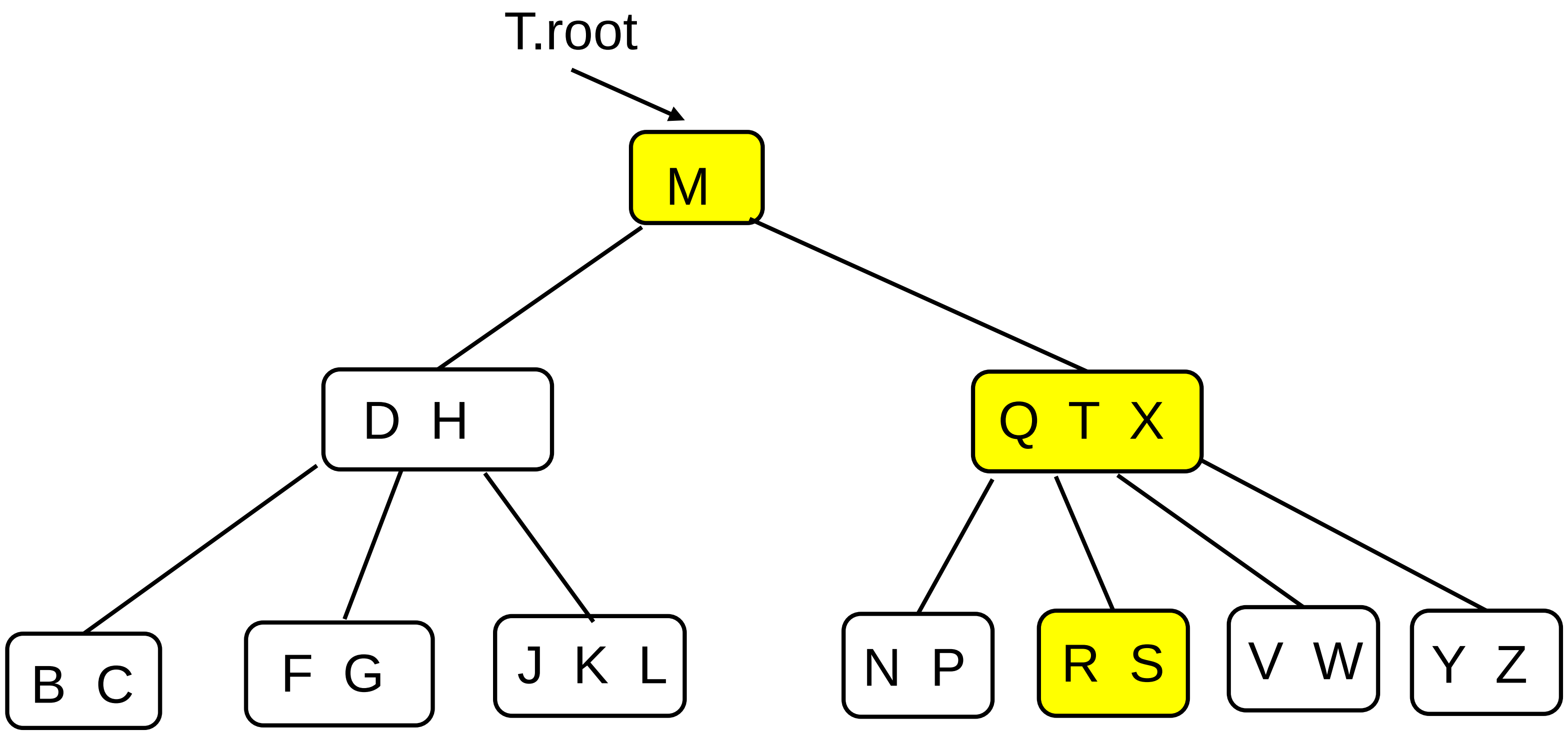
Computer & Electrical Engineering and Computer Science Department
Florida Atlantic University

B-trees

- Balanced search trees
- Work well on disks and other direct-access secondary storage devices
- Many database systems use B-trees or variants of B-trees to store information
- Efficient in minimizing disk I/O operations
- B-tree nodes may have from a few to thousands children
- B-trees have height O(lg n)

Reference: *Introduction to Algorithms*, 3rd edition, by T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, The MIT Press, 2009, chapter 18.

B-tree example



- keys are the consonants of English alphabet
- an internal node with x.n keys has x.n + 1 children
- all leaves have the same depth
- the shaded nodes are examined in search for the letter R

Primary/Secondary storage

- The **primary memory** (main memory) consists of silicon memory chips
- **Secondary storage**: magnetic storage technology such as tapes or disks
- Disks are *cheaper* and have *higher capacity* than the main memory
- Disks are much *slower* than the main memory because they have moving mechanical parts

Primary/Secondary storage

• average access time for commodity disks is ~ 8 – 11 ms
• access time for silicon memory is ~ 50 ns
⇒ access time for disks is over 5 order of magnitude slower!

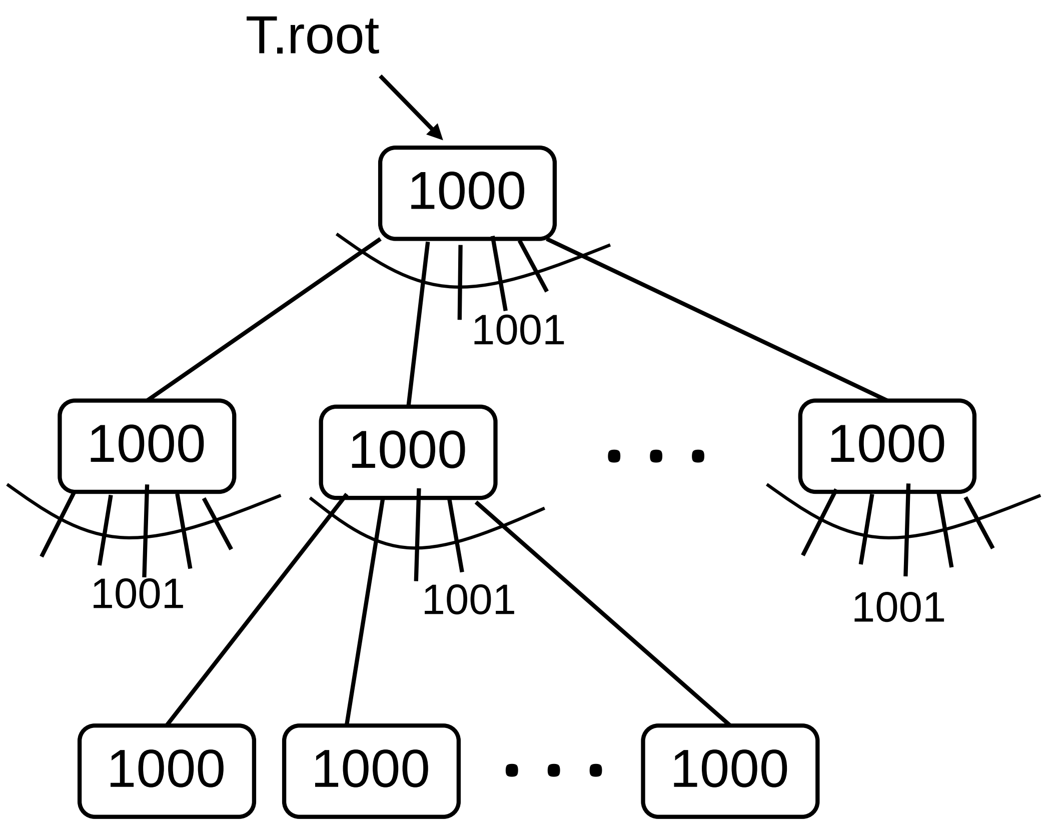
• information divided in **pages** (2^{11} – 2^{14} bytes)
• each disk reads/writes one or more pages

B-trees

- in a typical B-tree application, the whole B-tree does not fit in the main memory
- copy pages from disk to the main memory (MM), then write back onto the disk the pages that have changed
- usually, B-tree algorithms keep only a constant number of nodes in the main memory (MM)

```
x = a pointer to some object
DISK-READ(x)   // read object x in MM; no-op if
               // x already in the MM
operations that access/modify attributes of x
DISK-WRITE(x)  // omitted if no attributes of x changed
other operations that access but do not modify x
```

B-tree example: branching factor = 1001, height=2



1 node,
1000 keys

1001 nodes,
1,001,000 keys

1,002,001 nodes,
1,002,001,000 keys

- B-trees stored on disks, often have branching factors 50 … 2000
- keep the root node permanently in the MM $\Rightarrow$ find any key with at most two disk accesses

B-tree definition

A B-tree T is a rooted tree (where T.root is the root) with the following properties:

1. every node x has the following attributes
   a. $x.n$ – the number of keys currently stored in x
   b. the keys $x.key_1$, $x.key_2$, …, $x.key_{x.n}$ so that

   $$x.key_1 \leq x.key_2 \leq \ldots \leq x.key_{x.n}$$

   c. $x.leaf$ – a boolean value which is TRUE if x is a leaf and FALSE if x is an internal node
2. each internal node x has $x.n+1$ pointers $x.c_1$, $x.c_2$, …, $x.c_{x.n+1}$ to its children; if x is a leaf then its c_i attributes are undefined
3. if k_i is any key stored in the subtree with root $x.c_i$ then:

   $$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \ldots \leq x.key_{x.n} \leq k_{x.n+1}$$

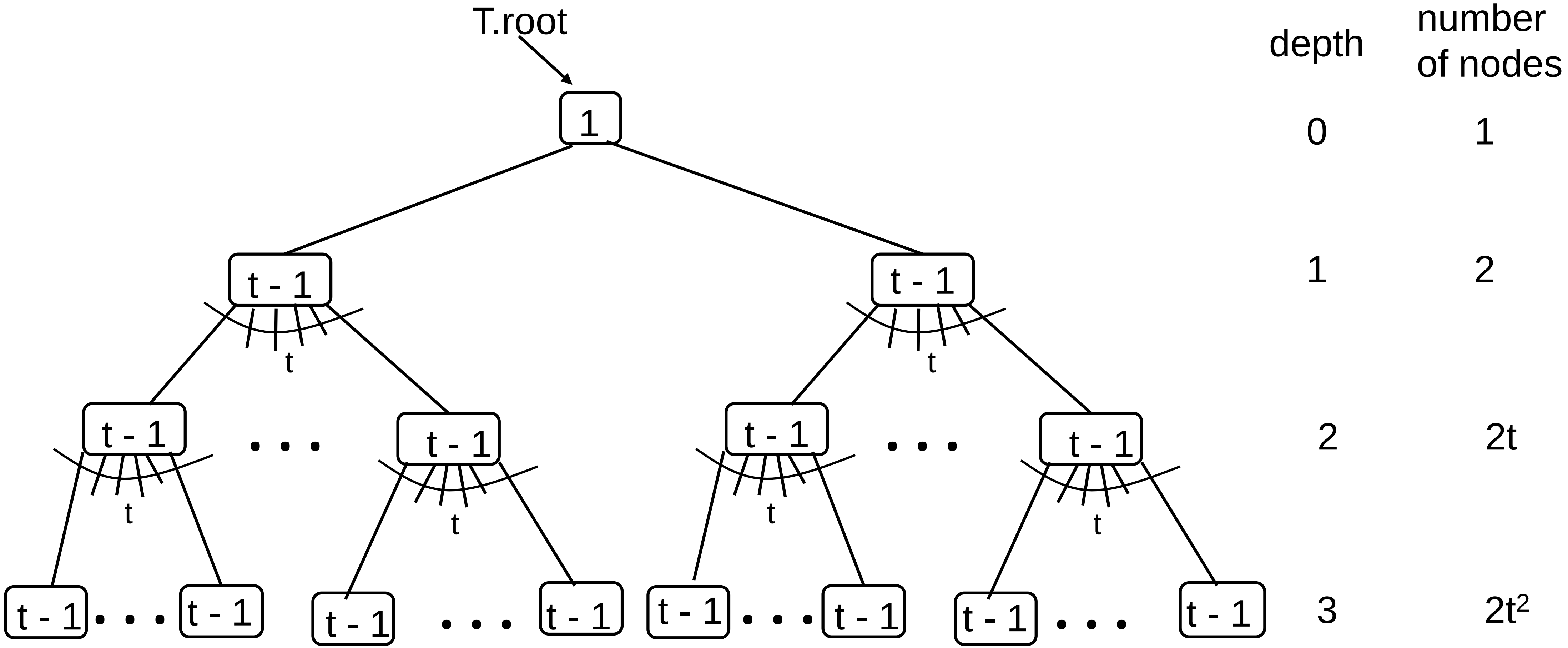
B-tree definition, cont.

4. all the leaves have the same depth, which is the tree high *h*.
5. the B-tree has a **minimum degree** *t* (*t* is an integer $t \geq 2$):
   - every node other than the root must have $\geq t - 1$ keys and $\geq t$ children; if B-tree is nonempty, then the root has at least one key
   - every node has $\leq 2t - 1$ keys and $\leq 2t$ children

   A node is full is it has $2t - 1$ keys.

The height of a B-tree

Theorem: if $n \geq 1$, then for any *n*-key B-tree T of height *h* and minimum degree *t*, $$h \leq \log_t \frac{n+1}{2}$$



The height of a B-tree, cont.

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

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

$$t^h \leq \frac{n+1}{2}$$

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

$h = O(\log_t n)$

$\boxed{h = O(\lg n)}$

Basic operations of B-trees

- B-TREE-SEARCH
- B-TREE-CREATE
- B-TREE-INSERT
- B-TREE-DELETE

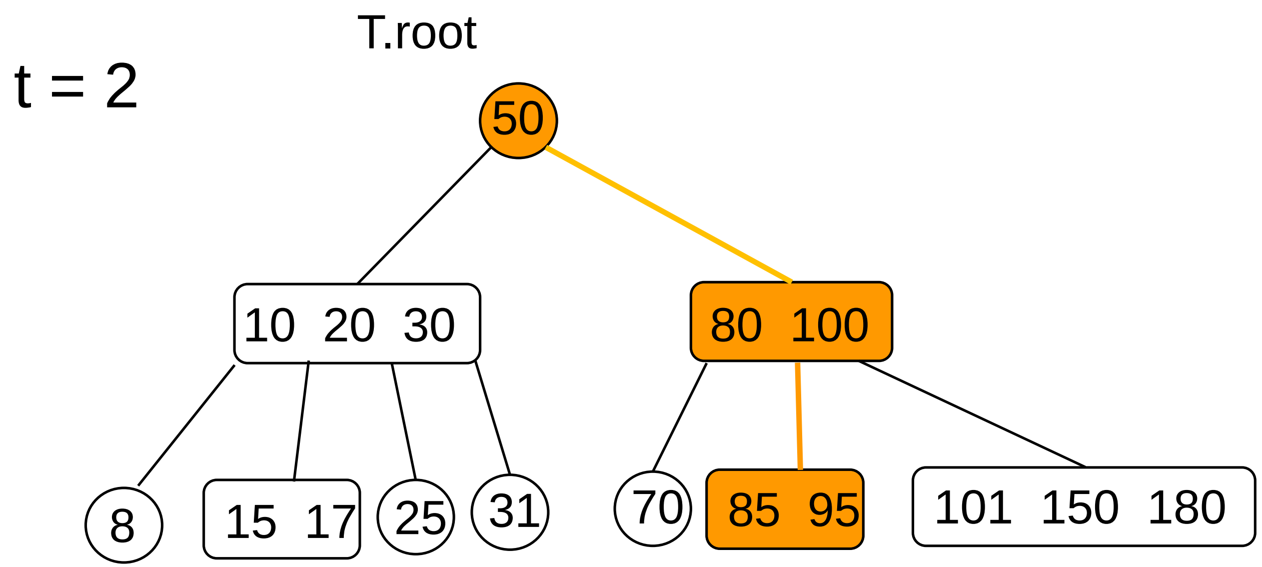
Conventions

- The root of the B-tree is always in the main memory
  - no need to call DISK-READ for the root
  - we do need to call DISK-WRITE when the root is changed
- All nodes passed as parameters must have already had a DISK-READ operation performed on them
- The procedures are **"one-pass" algorithms** that proceed downward from the root, without having to back up

Searching a B-tree

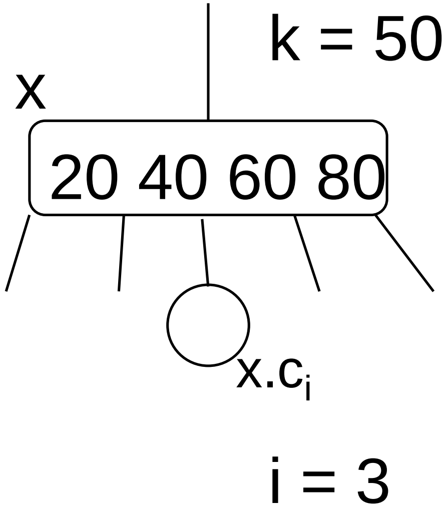
- At each node make a (x.n +1) – way branching decision

B-TREE-SEARCH(T.root, 85)



Search operation

```
B-TREE-SEARCH(x, k)
i = 1
while i ≤ x.n and k > x.key_i
        i = i + 1
if i ≤ x.n and k == x.key_i
        return (x,i)
elseif x.leaf == TRUE
        return NIL
else DISK-READ(x.c_i)
        return B-TREE-SEARCH(x.c_i, k)
```



Initial call: B-TREE-SEARCH(T.root, k)

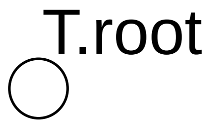
RT = $O(t \cdot \log_t n)$

- while loop takes $O(t)$
- number of recursive calls is $O(h) = O(\log_t n)$

Creating an empty B-tree

- Creates an empty root node

```
B-TREE-CREATE(T)
x = ALLOCATE-NODE()
x.leaf = TRUE
x.n = 0
DISK-WRITE(x)
T.root = x
```



RT = O(1)