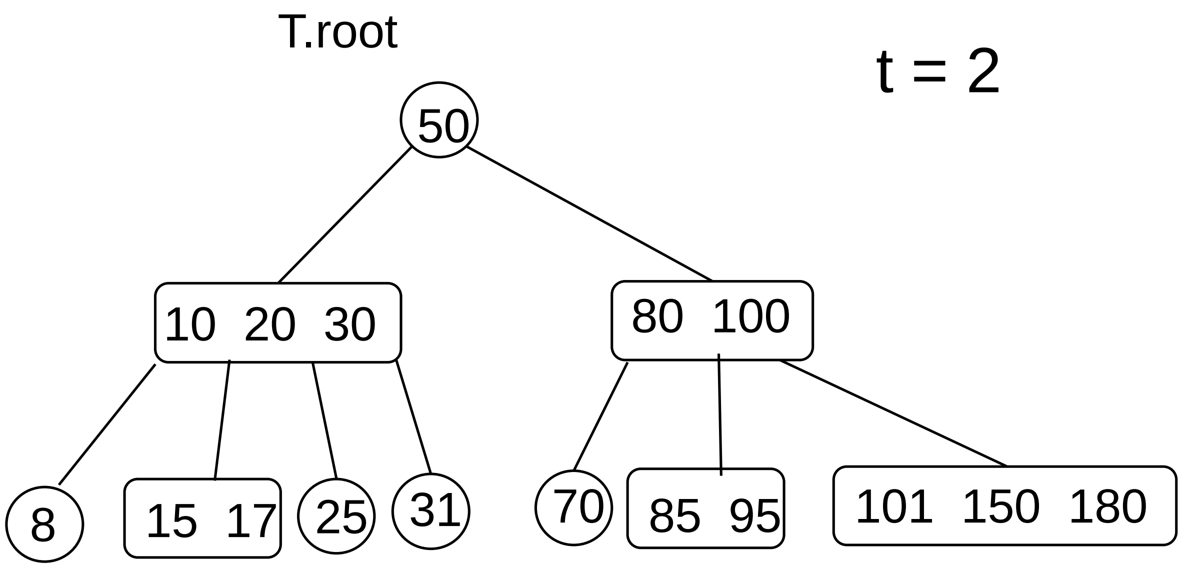
COT 6405
ANALYSIS OF ALGORITHMS

B-Trees

Computer & Electrical Engineering and Computer Science Department
Florida Atlantic University

Delete operation

- When deleting a key from an internal node, rearrange the node's children
- Any node (except the root) cannot have fewer than *t* – 1 keys
- B-TREE-DELETE(*x*,*k*) – deletes the key *k* from the subtree rooted at *x*



Idea : when calling delete on a node x, guarantee that the number of keys in x is ≥ t

- sometimes a key has to be moved to a child before recursion descends to that child

Delete operation

- **Goal**: delete a key from the tree in one downward pass w/o having to “back-up”
- If the root x becomes an internal node with no keys, then delete x and $x.c_1$ (the only child!) becomes the new root of the tree. The height of the tree is decreased by 1.
- Next, we discuss the rules for deleting keys from a B-tree

Rules for deleting a key

Rule 1: If the key $k \in$ to the LEAF node x , then delete the key k from x

Rule 2: If the key $k \in$ to the internal node x :

a. if the child y that precedes k in a node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k' and replace k by k' in x. Find and delete k' in a single downward pass.

b. if y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k' and replace k by k' in x. Find and delete k' in a single downward pass.

c. otherwise, if both *y* and *z* have only *t*-1 keys, merge *k* and all of *z* into *y*, so that *x* loses both *k* and the pointer to *z*, and *y* now contains 2*t*-1 keys. Then free *z* and recursively delete *k* from *y*.

Rules for deleting a key, cont.

Rule 3: If the key $k \notin$ to the internal node x, take $x.c_i$ the root of the subtree that must contain k (if k is in the tree). If $x.c_i$ has only $t - 1$ keys, then use 3a or 3b to guarantee we descend to a node with $\geq t$ keys

a. if $x.c_i$ has an immediate sibling with $\geq t$ keys, then give $x.c_i$ an extra key by:
   - moving a key from x to $x.c_i$,
   - moving a key from $x.c_i$'s immediate left or right sibling up to x,
   - moving the appropriate child pointer from the sibling into $x.c_i$

b. if both $x.c_i$'s immediate siblings have t-1 keys, merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median for that node

Delete operation, RT analysis

- Most of the keys are in the leaves
  - In practice, most often delete keys from the leaves
- One downward pass through the tree, without having to back up
  - Cases 2a & 2b: make a downward pass through the tree. Return to the node where the key was deleted to replace it with the predecessor/successor key.
- O(h) disk operations
  - O(1) calls to DISK-READ and DISK-WRITE between recursive invocations of the procedure.
- RT = O(th), thus RT = $O(t \log_t n)$