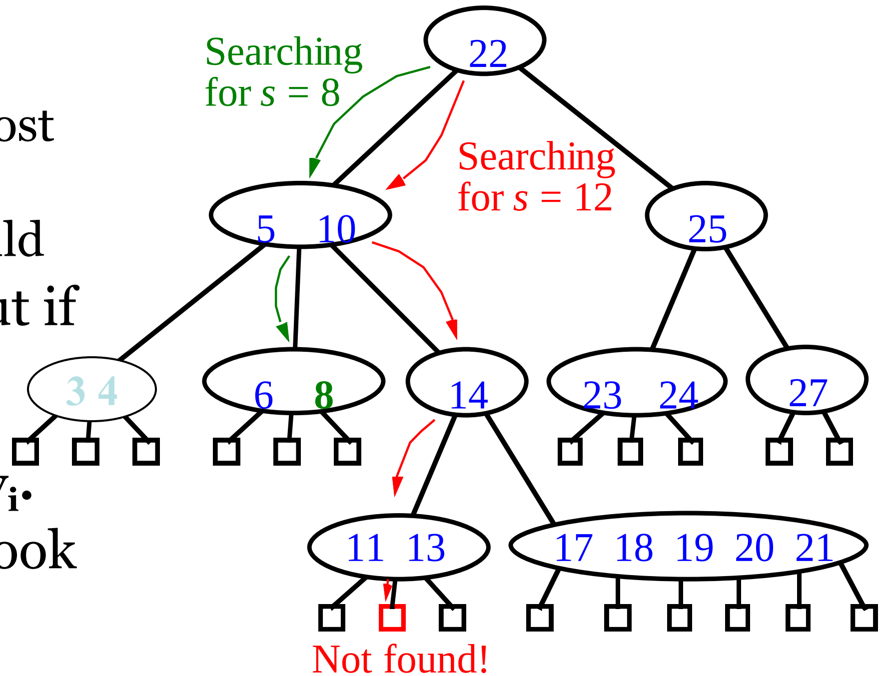
Multi-way Search Trees

- Each internal node of a multi-way search tree T:
  - has at least two children
  - contains d - 1 items, where d is the number of children ➔ d-nodes
  - "contains" 2 pseudo-items: $\mathbf{k_0=-\infty}$, $\mathbf{k_d=\infty}$
- Children of each internal node are "between" items
- all keys in the subtree rooted at the child fall between keys of those items

Multi-way Searching

- Similar to binary searching
  - If search key **s<k₁** search the leftmost child
  - If **s>k_{d-1}** , search the rightmost child
- That's it in a binary tree; what about if **d>2**?
  - Find two keys **k_{i-1}** and **k_i** between which s falls, and search the child **v_i**.
- What would an in-order traversal look like?



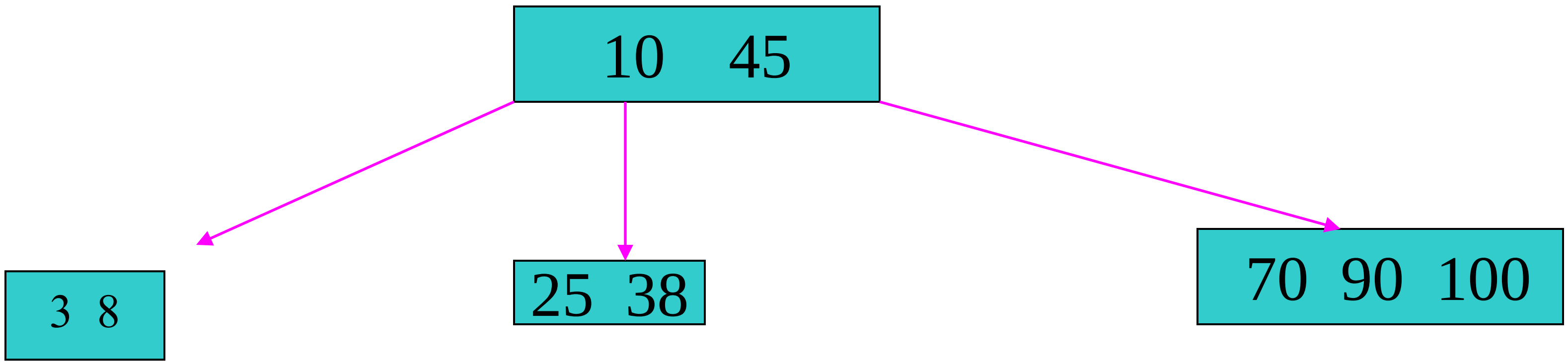
2-4 Trees

a. Nodes may contain 1, 2 or 3 items.

b. A node with k items has k + 1 children

c. All leaves are on same level.

Example

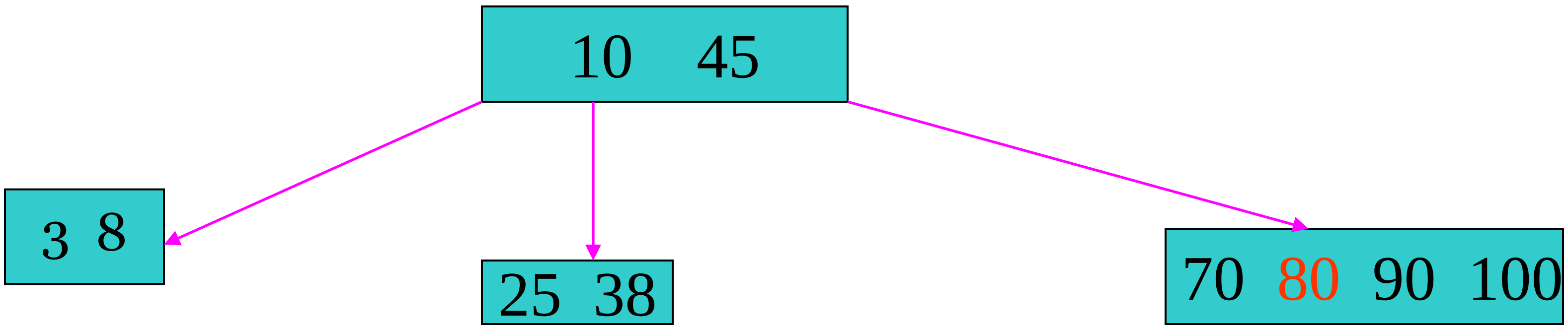


Insertion

- Insertion:
- Find the appropriate leaf. If there is only one or two items, just add to leaf.
- If no room, move middle item to parent and split remaining two items among two children.

Insertion

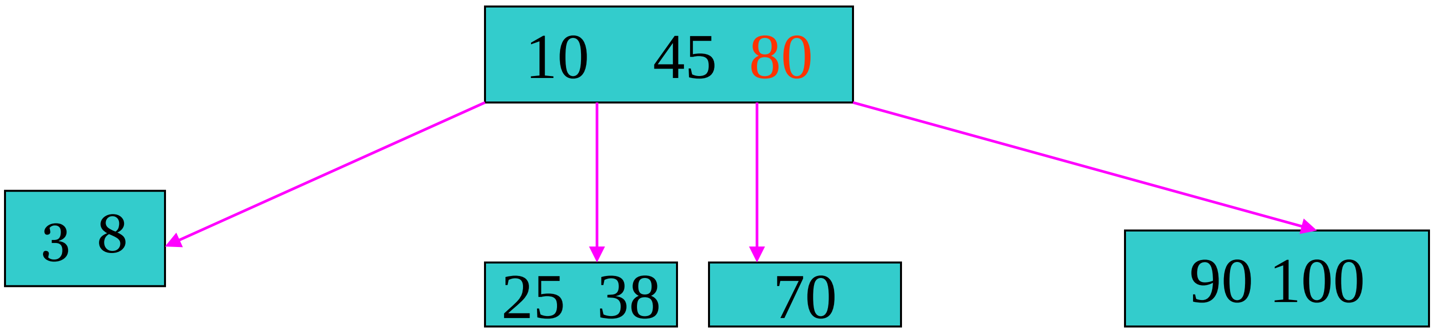
insert 80



Overflow!

Insertion

Split & move middle element to parent



Removal

- First : find the key with a simple multi-way search
- If the item to delete is in an internal node, reduce to the case where item is at the bottom of the tree by:
  - Find item which precedes it in in-order traversal
    - which one?
  - Swap them
  - Remove the item
  - Alternative?

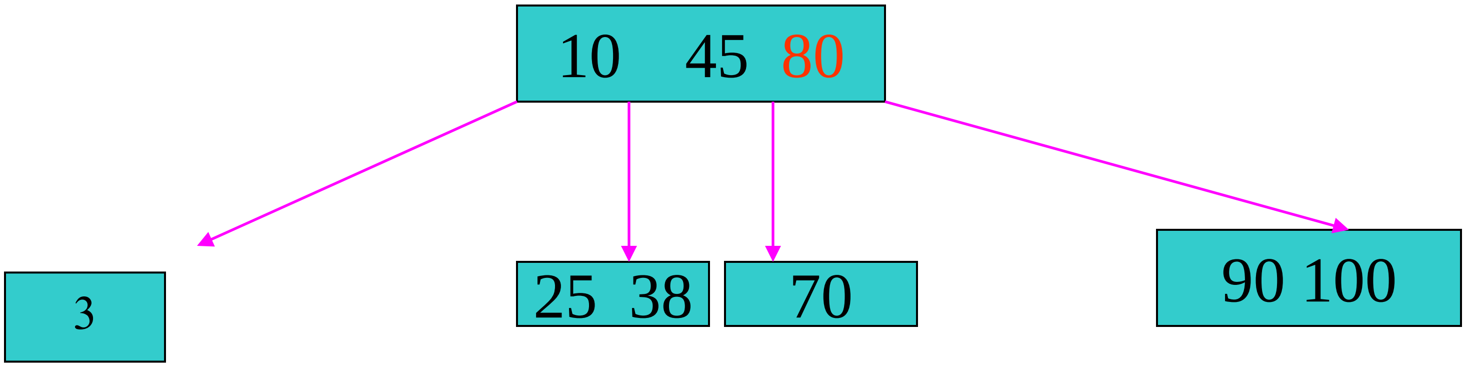
Removal

- Not enough items in the node

  Underflow!
- Pull an item from the parent, replace it with an item from a sibling - transfer
- Still not good enough! What happens if siblings are 2-nodes?
- Could we just pull one item from the parent?

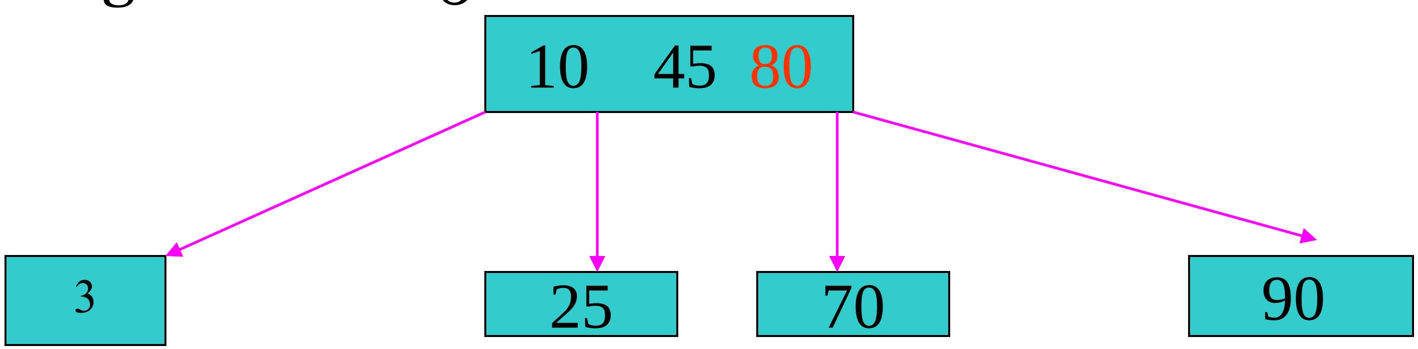
Removal

- Remove 3
  - move 10 into the subtree
  - move 25 into the parent



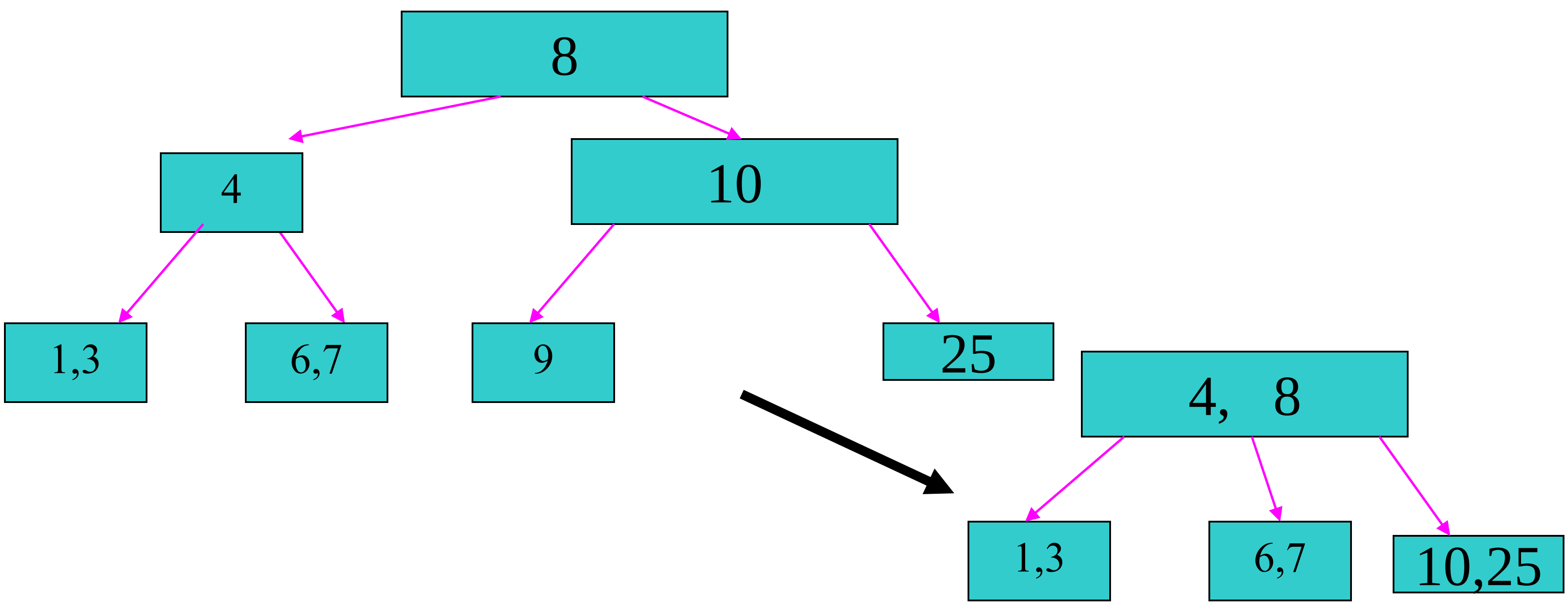
Removal

- If siblings are 2-nodes (i.e. contain only one key)
  - cannot ‘steal’ from them
- Do node merging
- Remove 3
  - move 10 into the subtree
  - merge 10 with 25



2-4 Trees

- More on removal:
  - What if parent is a 2-node?
  - Propagate underflow up the tree
  - Delete 9



2-4 Trees

- 2-4 trees are easy to maintain
- Insertion and deletion take O(log n)
- Balanced trees

B-Trees

- Up to now, all data that has been stored in the tree has been in memory.
- If data gets too big for main memory, what do we do?
- If we keep a pointer to the tree in main memory, we could bring in just the nodes that we need.
- For instance, to do an insert with a BST, if we need the left child, we do a disk access and retrieve the left child.
- If the left child is NIL, then we can do the insert, and store the child node on the disk.
- Not too good for a BST

B-Trees

- The problem with BST: storing the data requires disk accesses, which is expensive, compared to execution of machine instructions.
- If we can reduce the number of disk accesses, then the procedures run faster.
- The only way to reduce the number of disk accesses is to increase the number of keys in a node.
- The BST allows only one key per leaf.

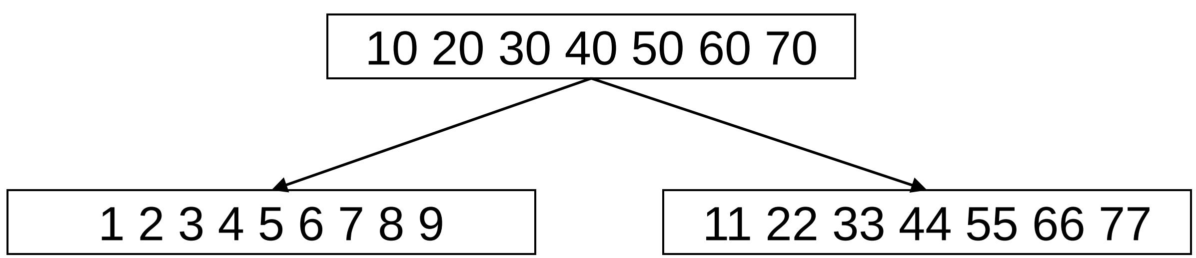
- Very good and often used for Search Engines!
  - (when collection size gets very big → the index does not fit in memory)

B-Trees

- If we increase the number of keys in the nodes, how will we do any tree operations effectively?

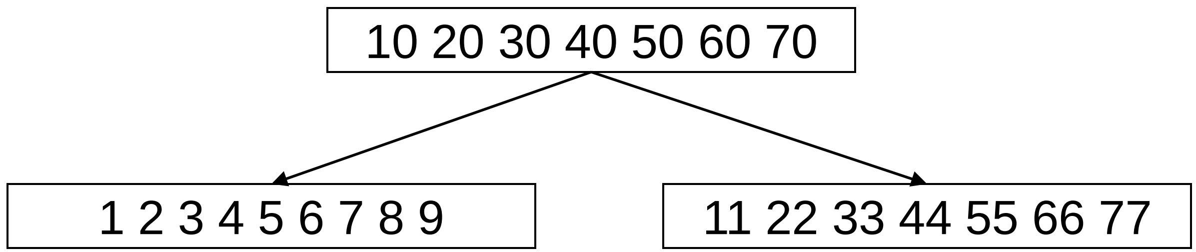
10 20 30 40 50 60 70

- Above is a node with 7 keys. How do we add children?



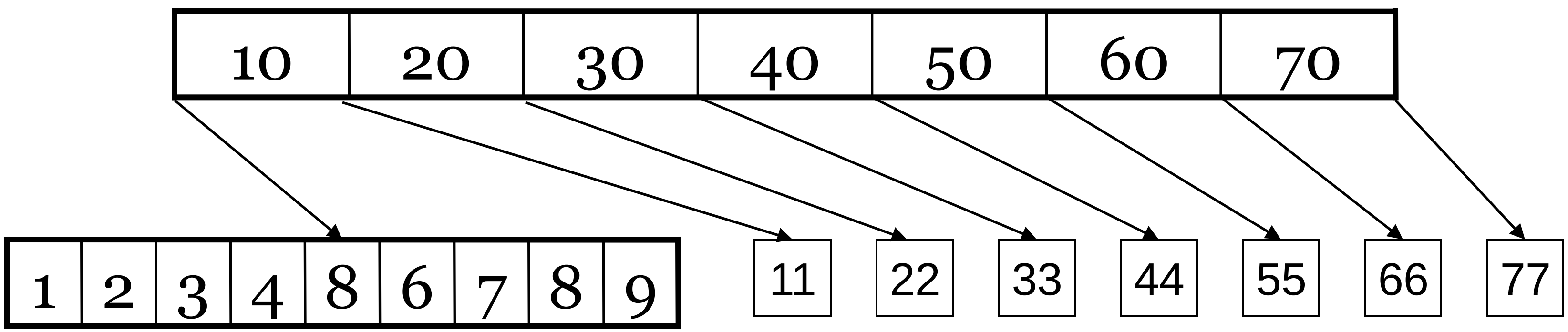
B-Trees

- Clearly, the tree below is useless.



- How many pointers do we need?
- Using the idea of BST, we need to be able to put nodes into the tree that have smaller, same and larger values than the node we are currently examining.

B-Trees: A General Case of Multi-Way Search Trees



- We can easily find any value.
- We need to create operations, which require rules on what makes a tree a B-Tree.
- Clearly, having one key per node would be very bad.
- We need a mechanism to increase the height of the tree (since the number of keys in any node can get very high) so we can shift keys out of a node, making the nodes smaller.

B-Trees: Fields in a Node

A B-Tree is a rooted tree (whose root is root[T])) having the following properties:

1. Every internal node x has the following fields:

n[x]	$key_1[x]$	$key_2[x]$	$key_3[x]$	$key_4[x]$	$key_5[x]$	$key_6[x]$	$key_7[x]$	$key_8[x]$	leaf[x]
$c_1[x]$	$c_2[x]$	$c_3[x]$	$c_4[x]$	$c_5[x]$	$c_6[x]$	$c_7[x]$	$c_8[x]$	$c_9[x]$	

n[x] is the number of keys in the node. n[x] = 8 above.

leaf[x] = false for internal nodes, since x is not a leaf.

The $key_i[x]$ are the values of the keys, where $key_i[x] \leq key_{i+1}[x]$.

$c_i[x]$ are pointers to child nodes. All the keys in $c_i[x]$ have values that are between $key_{i-1}[x]$ and $key_i[x]$.

B-Trees

- Leaf nodes have no child pointers
- leaf[x] = true for leaf nodes.
- All leaf nodes are at the same level

n[x]	key_1[x]	key_2[x]	key_3[x]	key_4[x]	key_5[x]	key_6[x]	key_7[x]	key_8[x]	leaf[x]

B-Trees

- There are lower and upper bounds on the number of keys a node can contain. This depends on the **“minimum degree” t** ≥ 2, which we must specify for any given B-Tree.

a. Every node other than the root must have at least t-1 keys. Every internal node other than the root thus has t children. If the tree is nonempty, the root must have at least one key.
b. Every node can contain at most 2t-1 keys. Therefore, an internal node can have at most 2t children. A node is **full** if it contains exactly 2t-1 keys.

Height of B-Tree

- If $n \geq 1$, then for any n-key B-tree of height h and mimimum degree $t \geq 2$,

  $$\text{height} = h \leq \log_t[(n+1)/2]$$

- The important thing to notice is that the height of the tree is log base t. So, as t increases, the height, for any number of nodes n, will decrease.
- Using the formula $\log_a x = (\log_b x)/(\log_b a)$, we can see that
  - $\log_2 10^6 = (\log_{10} 10^6)/(\log_{10} 2) \approx 6/0.30102999566398 \approx 19$
  - $\log_{10} 10^6 = 6$

So, 13 less disk accesses to get to the leafs!

Basic Operations

- The root of the B-tree is always in main memory, so that a Disk-Read on the root is never required; a Disk-Write of the root is required, however, whenever the root node is changed.

- Any nodes that are passed as parameters must already have had a Disk-Read operation performed on them.

Searching a B-Tree

```
B-TREE-SEARCH(x, k)
i ← 1
while i ≤ n[x] and k > key_i[x]
      do i ← i + 1
if i ≤ n[x] and k = key_i[x]
    then return (x, i)
if leaf[x]
    then return NIL
    else DISK-READ(c_i[x])
         return B-TREE-SEARCH(c_i[x], k)
```

Start at the leftmost key in the node, and go to the right until you go too far.

If it is a leaf node, then you are done, as there is no leaf to inspect

Otherwise, retrieve the child node from the disk, and put it into memory

Inserting into B-trees

- Really very easy. Very similar with (2,4) trees.
- Just keep in mind that you are starting at the root, and then finding the subtree where the key should be inserted, and following the pointer.
- A deletion may eventually occur, and sometimes deletions force keys into their parents. So, if we encounter a full node on our way to the node where the insertion will take place, we must split that node into two.

Inserting into B-trees (cont'd)

```
B-TREE-INSERT(T, k)
r ← root[T]
if n[r] = 2t − 1
   then s ← ALLOCATE-NODE()
        root[T] ← s
        leaf[s] ← FALSE
        n[s] ← 0
        c1[s] ← r
        B-TREE-SPLIT-CHILD(s, 1, r)
        B-TREE-INSERT-NONFULL(s, k)
   else B-TREE-INSERT-NONFULL(r, k)
```

If the node has 2t-1 keys, it can't accept any more keys, so you need to split it into 2 nodes before doing the insert.

Otherwise, call Nonfull()

Deleting Keys from Nodes

1. If the key k is in node x and x is a leaf, delete the key k from x.
2. If the key k is in node x and x is an internal node, do the following.

   a. If the child y that precedes k in 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. (Finding k' and deleting it can be performed in a single downward pass.)

   b. Symmetrically, if the child z that follows k in node x 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. (Finding k' and deleting it can be performed 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 $2t - 1$ keys. Then, free z and recursively delete k from y.

Deleting Keys from Nodes

3. If the key k is not present in internal node x, determine the root $c_i[x]$ of the appropriate subtree that must contain k, if k is in the tree at all. If $c_i[x]$ has only $t-1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then, finish by recursing on the appropriate child of x.

   a. If $c_i[x]$ has only $t-1$ keys but has an immediate sibling with at least t keys, give $c_i[x]$ an extra key by moving a key from x down into $c_i[x]$, moving a key from $c_i[x]$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $c_i[x]$.

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