Lecture 7: B-Trees

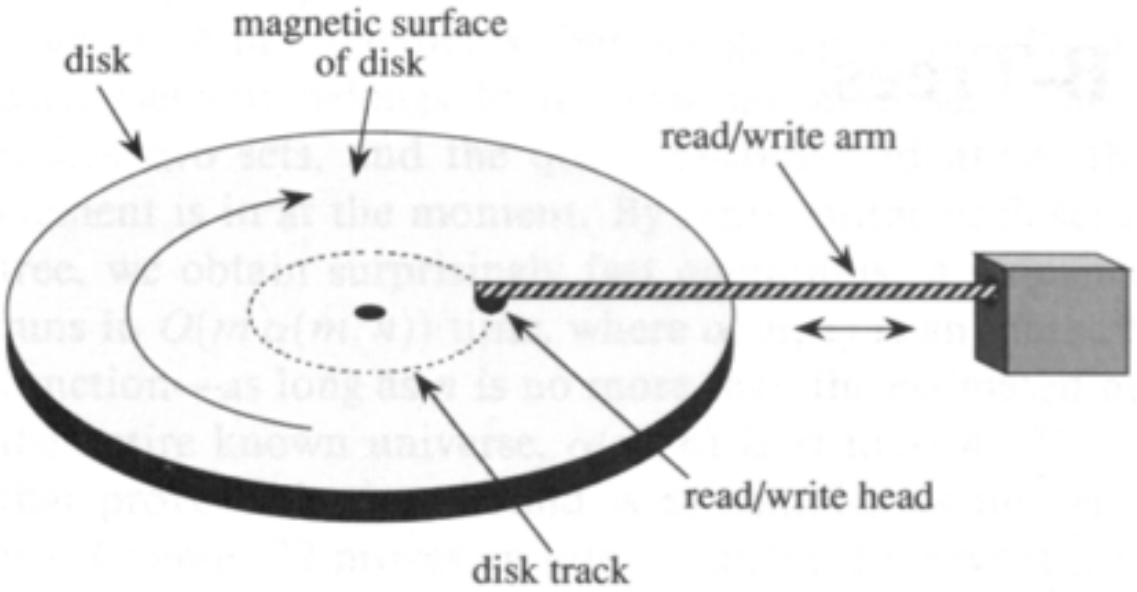
Data structures for secondary storage devices

- The search trees mentioned so far (binary search tree, red-black tree) were limited to main memory structures
  - The dataset organized in a search tree fits in the main memory (including the tree overhead, such as pointers, colors, etc.)
- Counterexample: daily transaction data of a bank $>$ 1GB
  - does not fit in main memory $\Rightarrow$ use secondary storage media: hard disks, magnetic tapes, etc.
- Consequence: enable a search tree structure for secondary storage devices

Types of memory

- Primary memory (or main memory)
  - Typical implementation: RAM memory chips
  - fast R/W operations: $10^3 - 10^4$ times faster than secondary storage
  - expensive: $\approx$ 100 times more expensive than secondary storage
  - volatile: data does not persist without power supply
- Secondary storage
  - Typical implementation: hard disk
  - slower R/W, but cheaper investment
  - Can store large amounts of data (1-2 TB vs. 4-8 GB)
  - persistent data storage

Hard Disks



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

Hard Disk access model

- DISKREAD(x : `pointer_to_node`) reads object x from secondary storage into main memory, and refers to it by x.
  - It x is already in memory, then DISKREAD(x) requires no disk access (it is a "no-op")
- DISKWRITE(x : `pointer_to_node`) saves to disk the object referred to by x.
- ALLOCATENODE() : `pointer_to_node` allocates space for a node.

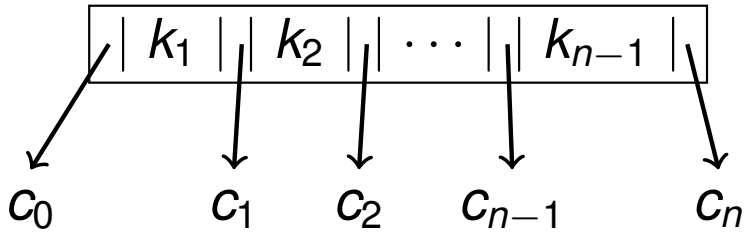
Typical programming pattern

```
...
// x is a pointer to some object
DISKREAD(x)
operations that access and/or modify the fields of x
DISKWRITE(x)    // omitted, if no fields of x were changed
other operations that access but don't modify the fields of x
...
```

B-trees

Balanced search trees for huge datasets read from secondary memory because they do not fit in main memory.

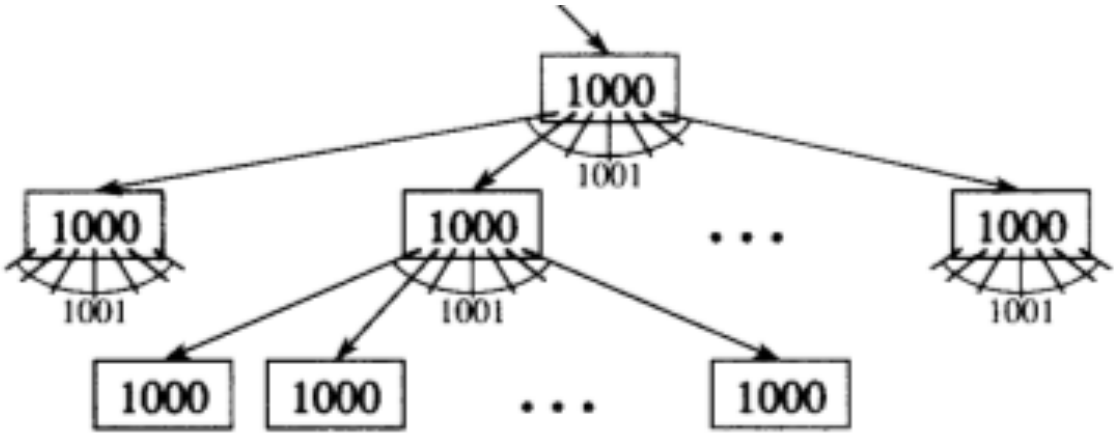
- Pointers in data structures are no longer addresses in main memory, but locations in files
- If x is a pointer to an object, then
  - If x is in main memory, we access it directly.
  - Otherwise, we use DISKREAD(x) to read the object from disk to main memory. (DISKWRITE(x) writes it back to disk.)
  - Typical structure of a node (graphical representation):



where $k_0, k_1, \ldots, k_{n-1}$ are the keys stored in the node, and $c_0, c_1, \ldots, c_n$ are pointers to B-trees, such that: if k'_j is any key stored in a B-tree referred to by c_j $(0 \leq j \leq n)$, then $k'_0 \leq k_0 \leq k'_1 \leq k_1 \leq k'_2 \leq \ldots \leq k'_{n-1} \leq k_{n-1} \leq k'_n$.

Binary trees versus B-trees

- Size of B-tree is determined by the page size.
  One page $\leftrightarrow$ one node.
- A B-tree of height 2 may contain more than 10^6 keys!

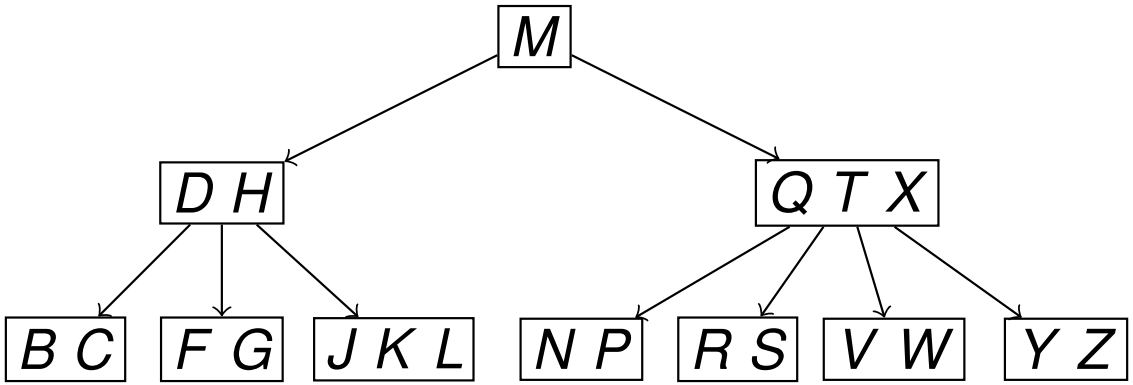


- Heights of binary tree and B-tree are both logarithmic:

  B-tree: $\log_t n$ where t is the branching factor

  Binary tree: $\log_2 n$

The 21 English consonants as keys of a B-tree:



(1) Every internal node x has n keys and $n+1$ children.

(2) All leaves are at the same depth in the tree.

- The tree may *shrink* (by deleting nodes) or *grow* (by adding nodes), but properties (1) and (2) are preserved.
- See here an animated behavior of insertion and deletion in a B-tree.

B-tree formal definition

We assume bounded B-trees, where the bounds of the tree are given by an integer $t \geq 2$.

Definition (B-tree)

A B-tree is a rooted tree T with root $T.root$, and with every node x consisting of **four fields**:

- $x.n$: the number of keys currently stored in node x.
- An array $x.key[0..2t-2]$ of size $2t-1$, where $n < 2t$. Keys are stored in nondecreasing order in $x.key$:

$$x.key[0] \leq x.key[1] \leq \ldots \leq x.key[n-1].$$

- A boolean value $x.leaf$, which is `true` if x is a leaf node, and `false` otherwise.
- An array $x.c[0..2t-1]$ of size $2t$, which contains $n+1$ pointers $x.c[0], \ldots, x.c[n]$ to its children.

The following properties must be satisfied: (*see next slide* $\rightarrow$)

B-tree formal definition (continued)

Definition (B-tree (contd.))

- The keys $x.key[i]$ separate the ranges of keys stored in each subtree: If k_i is any key stored in the subtree pointed to by $x.c[i]$ then

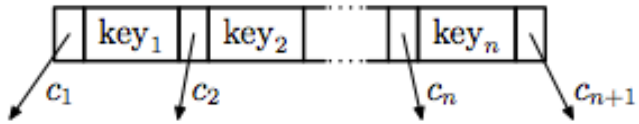
$$k_0 \leq x.key[0] \leq k_1 \leq x.key[1] \leq \ldots \leq x.key[n-1] \leq k_n.$$

- All leaves have the same height, which is the tree's height h.
- $t - 1 \leq n \leq 2t - 1$. The number t is called the minimum degree of the B-tree:

  Lower bound: every node other than the root must have at least $t - 1$ keys
  $\Rightarrow$ at least t children.

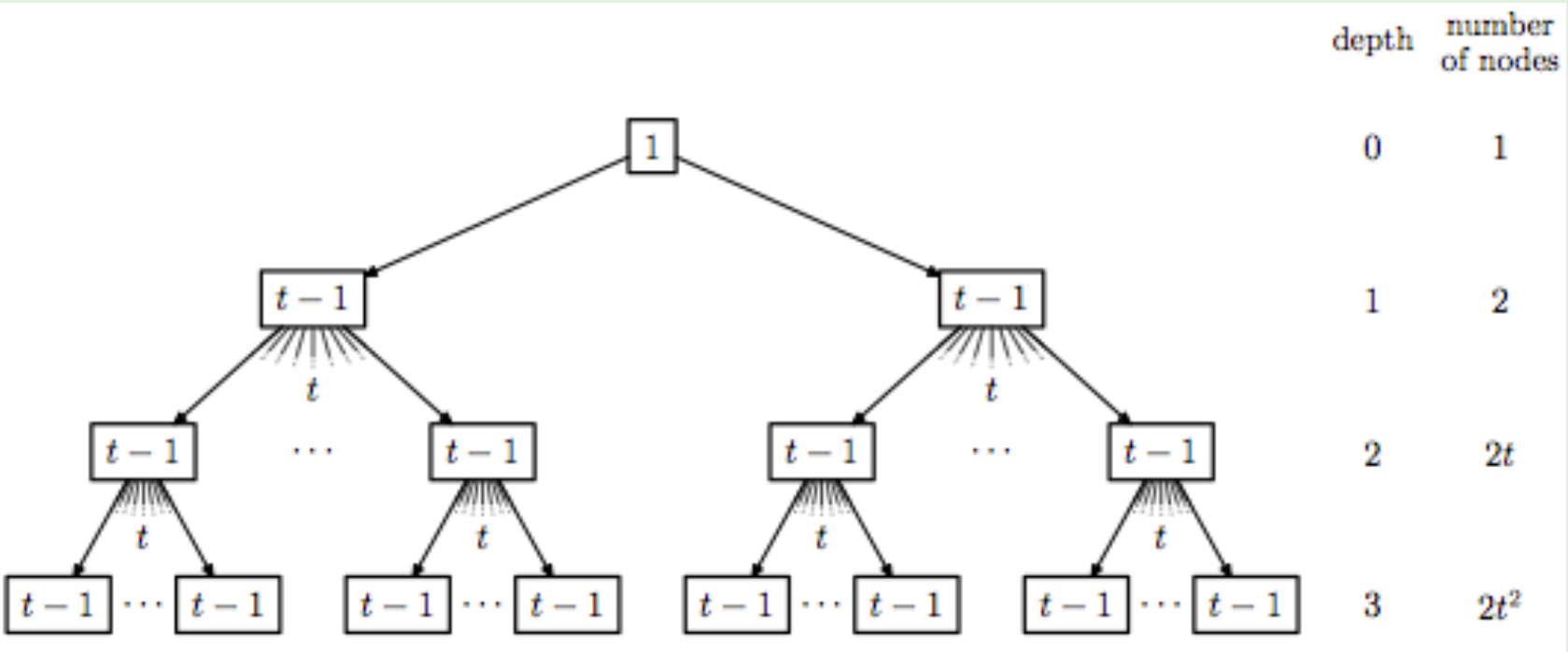
  Upper bound: every node can contain at most $2t - 1$ keys
  $\Rightarrow$ at most $2t$ children.

Representing pointers and keys in a node:



Example (B-tree height: the worst case)

A *B*-tree of height 3 containing a minimum possible number of keys.



Inside each node x, we show the number of keys $x.n$ contained.

The height of a B-tree

- Number of **disk accesses** is proportional to the **height** of the B-tree.
- The worst-case height of a B-tree is

$$h \leq \log_t \frac{n+1}{2} \sim O(\log_t n).$$

- **Main advantage** of B-trees compared to red-black trees: The base of the logarithm, t, can be much larger.

$\Longrightarrow$ B-trees save a factor $\sim \log_t$ over red-black trees in the number of nodes examined in tree operatios

$\Longrightarrow$ Number of disk access operations is substantially reduced.

Basic operations on B-trees

Details of the following operations:

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

Assumptions

- The root of the B-tree is always in main memory (DISKREAD of the root is never required)
- Any node passed as parameter must have had a DISKREAD operation performed on it.

All presented procedures are top-down algorithms starting at the root of the tree.

Searching a B-tree (I)

2 inputs: x, pointer to the root node of a subtree,
k, a key to be searched in that subtree.

function B-TREE-SEARCH(x, k)
returns (y, i) such that $y\text{->}key[i] = k$ or NIL
$i := 0$
while $i < x\text{->}n$ and $k > x\text{->}key[i]$
    $i := i = 1$
if $i < x\text{->}n$ and $k = x\text{->}key[i]$ then return (x, i)
if $x\text{->}leaf$
    return NIL
else DISK-READ($x\text{->}c[i]$)
    return B-TREE-SEARCH($x\text{->}c[i], k$)

Note. At each internal node x we make an $(x.n + 1)$-way branching decision.

Searching a B-tree (II)
Complexity analysis

- Number of disk pages accessed by B-TREE-SEARCH

  $$\Theta(h) = \Theta(\log_t n)$$

- Time of **while** loop within each node is $O(t)$ therefore the total CPU time is

  $$O(t\,h) = O(t\,\log_t n)$$

Creating an empty B-tree

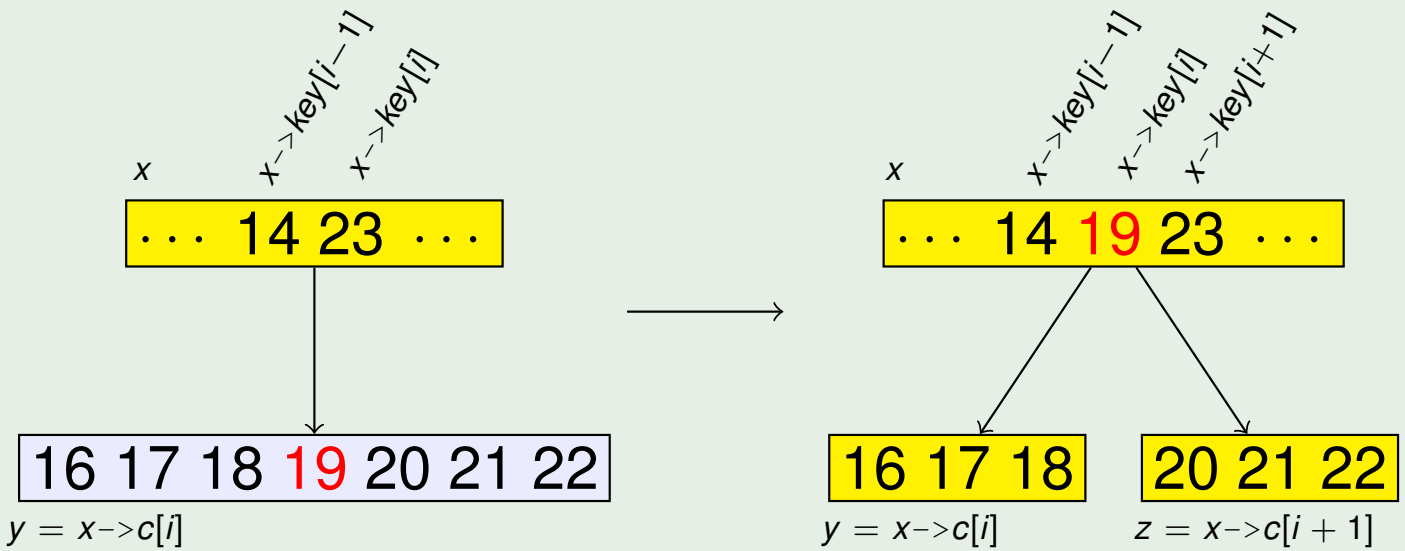
B-TREE-CREATE(T)
  x := ALLOCATENODE()
  x->*leaf* := `true`
  x->n := 0
  DISK-WRITE(x)
  T.*root* := x

- ALLOCATENODE() allocates one disk page to be used as a new node.
- requires $O(1)$ disk operations and $O(1)$ CPU time.

Splitting a node in a B-tree (I)

- Inserting a key into a B-tree is more complicated than inserting in a binary search tree.
  - If the insertion happens in a full node y with $2t - 1$ keys, then the node must be split in two:
    - Splitting is done around the median key $y.key[t - 1]$ of y.
    - The median key moves into the parent of y (which has to be nonfull).
    - If y is the root node, the tree height grows by 1.

Example ($t = 4$)



Splitting a node in a B-tree (II)

3 inputs: x, a nonfull internal node,
i, an index,
y, a node such that $y = x\text{->}c[i]$ is a full child of x.

```
B-TREE-SPLIT-CHILD(x, i, y)
  z := ALLOCATENODE()
  z->leaf := y->leaf
  z->n := t − 1
  for j := 0 to t − 2
      z->key[j] := y->key[j + t]
  if not(y->leaf) then
      for j := 0 to t − 1
          z->c[j] := y->c[j + t]
  y->n := t − 1
  for j := x->n downto i + 1
      x->c[j + 1] := x->c[j]
      x->key[j] := x->key[j − 1]
  x->c[i + 1] := z
  x->key[i] := y->key[t − 1]
  x->n := 1 + x->n
  DISK-WRITE(y); DISKWRITE(z); DISKWRITE(x)
```

Complexity time of B-TREE-SPLIT-CHILD is $\Theta(t)$ due to the loops.

Inserting a node in a B-tree (I)

- The key is always inserted in a leaf node
- Inserting is done in a single pass down the tree
- Requires $O(h) = O(\log_t n)$ disk accesses
- Requires $O(t\,h) = O(t \log_t n)$ CPU time
- Uses B-TREE-SPLIT-CHILD to guarantee that recursion never descends to a full node

Inserting a node in a B-tree (II)

2 inputs: T, the root node,
k, key to insert.

```
B-TREE-INSERT(T, k)
  r := T.root
  if r->n = 2t − 1 then
    s := ALLOCATENODE()
    T.root := s
    s->leaf := false
    s->n := 0
    s->c[0] := r
    B-TREE-SPLIT-CHILD(s, 0, r)
    B-TREE-INSERT-NONFULL(s, k)
  else B-TREE-INSERT-NONFULL(r, k)
```

Uses B-TREE-INSERT-NONFULL to insert key k into nonfull node x.

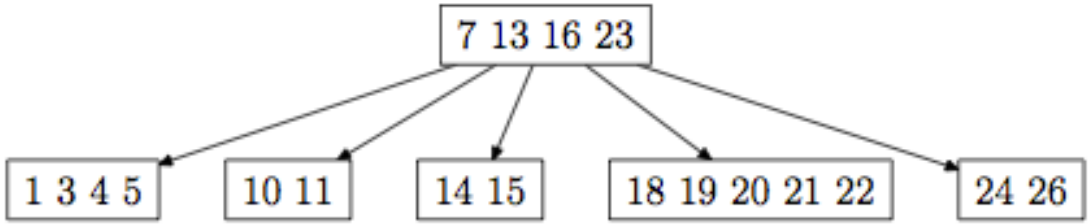
Inserting a node in a nonfull node of a B-tree

```
B-TREE-INSERT-NONFULL(x, k)
i := (x->n) − 1
if x->leaf then
   while i ≥ 0 and k < x->key[i]
         x->key[i + 1] := x->key[i]
         i := i − 1
   x->key[i + 1] := k
   x->n := x->n + 1
   DISKWRITE(x)
else while i ≥ 0 and k < x->key[i]
              i := i − 1
      i := i + 1
      DISKREAD(x->c[i])
      if x->c[i]->n = 2t − 1 then
            B-TREE-SPLIT-CHILD(x, i, x->c[i])
            if k > x->key[i] then i := i + 1
      B-TREE-INSERT-NONFULL(x->c[i], k)
```

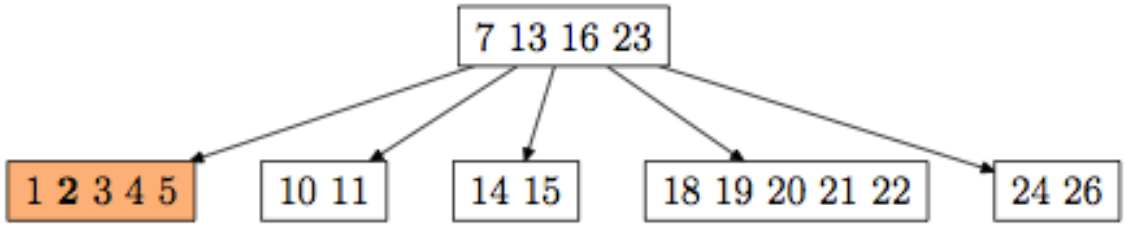
Inserting a node in a B-tree
Examples (I)

Initial tree:

$t = 3$

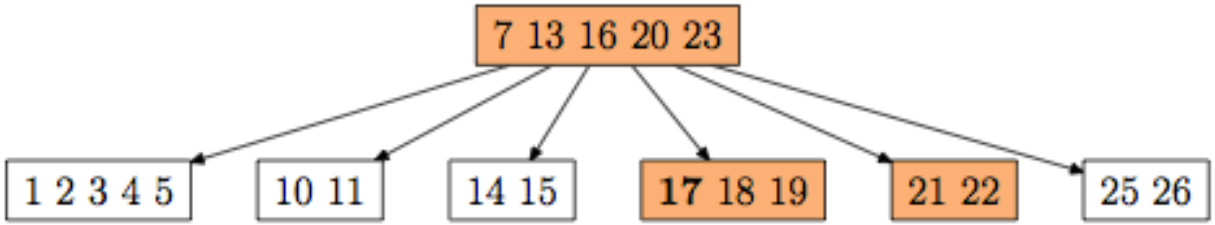


2 inserted:



17 inserted:

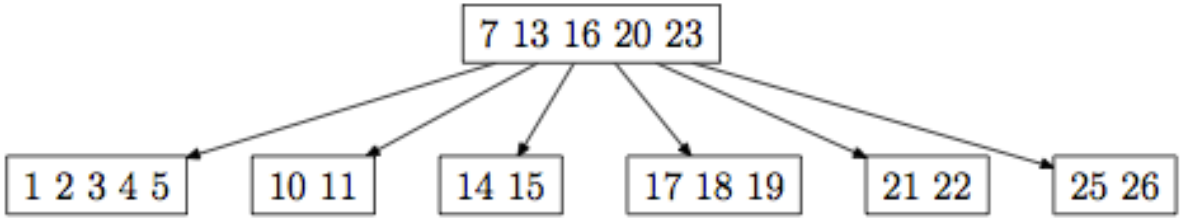
(to the previous one)



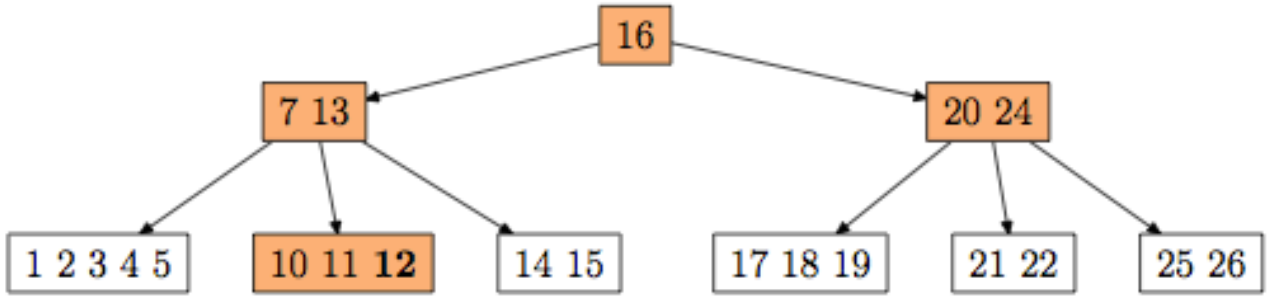
Inserting a node in a B-tree
Examples (II)

Initial tree:

$t = 3$

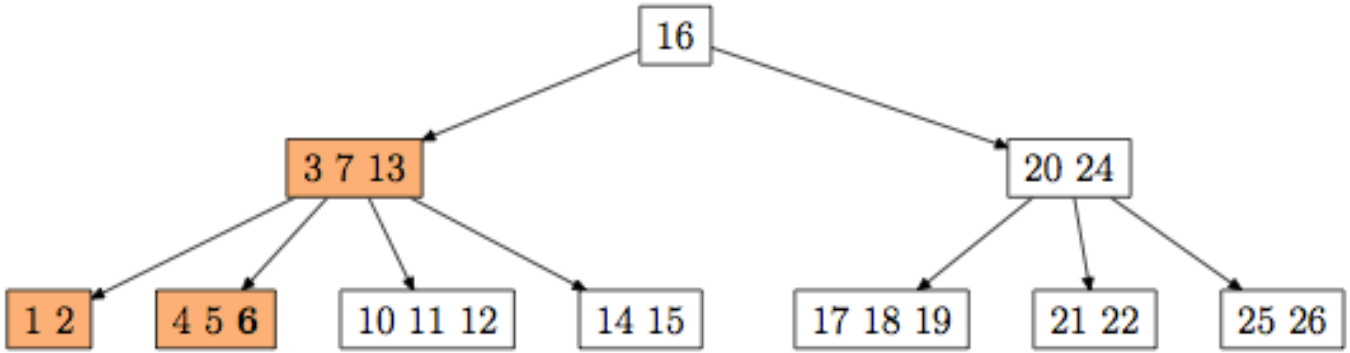


12 inserted:



6 inserted:

(to the previous one)



Deleting a key from a B-tree

B-TREE-DELETE(x, k)

B-TREE-DELETE(x, k) is asked to delete the key k from the subtree rooted at x.

- B-TREE-DELETE is designed to guarantee that whenever it is called recursively on a node x, the number of keys in x is $\geq t$.
  - $\Rightarrow$ Sometimes, a key may have to be moved into a child node before recursion descends to that child.
- Deleting is done in a single pass down the tree, but needs to return to the node with the deleted key if it is an internal node item. In the latter case, the key is first moved down to a leaf. Final deletion always takes place on a leaf.
- More complicated than insertion, but similar performance figures: $O(h)$ disk accesses, $O(t\,h) = O(t \log_t n)$ CPU time

Deleting a key from a B-tree: cases 1-2

We distinguish 3 cases. Let k be the key to be deleted, and x be the node containing the key.

(1) If k is a key in node x and x is a leaf, delete k from x.

(2) If key k is in node x and x is internal node, there are three cases:

   (a) If the child y that precedes k in node x has at least t keys (more than the minimum), then find the predecessor key k' in the subtree rooted at y. Recursively delete k' and replace k with k' in x.

   (b) Symmetrically, if the child z that follows k in node x has at least t keys, find the successor k' and delete and replace as before. Note that finding k' and deleting it can be performed in a single downward pass.

   (c) Otherwise, if both y and z have only $t - 1$ (minimum number) keys, merge k and all of z into y, so that both k and the pointer to z are removed from x. y now contains $2\,t - 1$ keys, and subsequently k is deleted.

Deleting a key from a B-tree: case 3

(3) If key k is not present in an internal node x, determine x such that x->$c[i]$ contains key k. If x->$c[i]$ has only $t-1$ keys, execute either of the following two cases to ensure that we descend to a node containing at least t keys, including k. Then, finish by recursing on the appropriate child of x.

(a) If x->$c[i]$ has only $t-1$ keys but has a sibling with t keys, give the root an extra key by moving a key from x to the root, moving a key from the root's immediate left or right sibling up into x, and moving the appropriate child from the sibling to x.

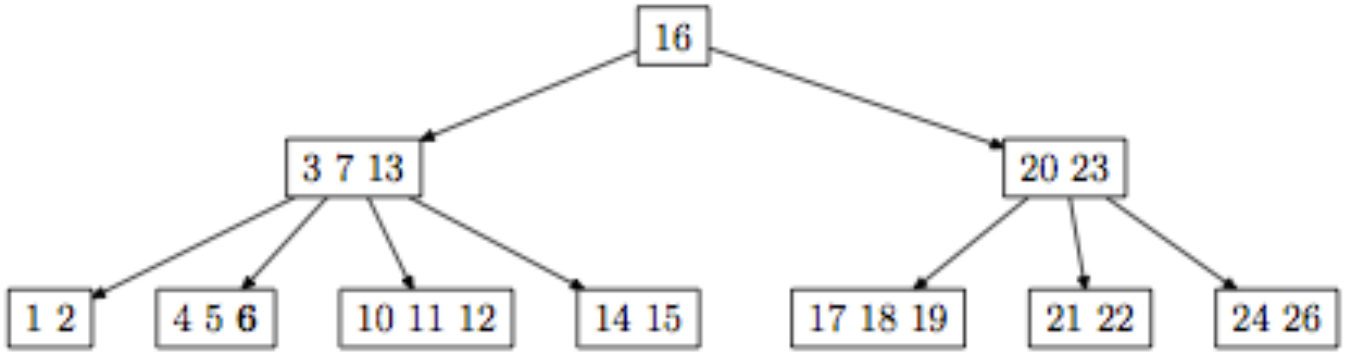
(b) If x->$c[i]$ and all of its siblings have $t-1$ keys, merge x->$c[i]$ with one sibling. This involves moving a key down from x into the new merged node to become the median key for that node.

Deleting a key — Case 1

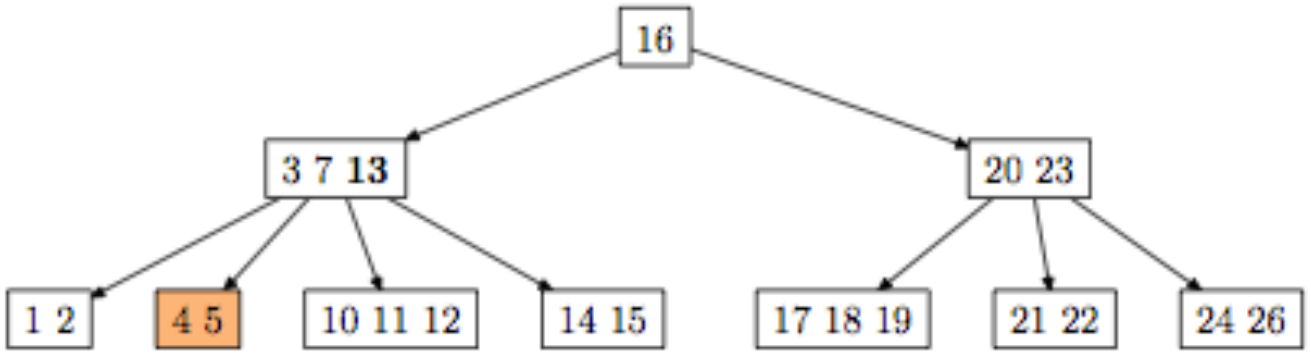
- This is the simple case; it involves deleting the key from the leaf. $n - 1$ keys remain.

$t = 3$

Initial tree:



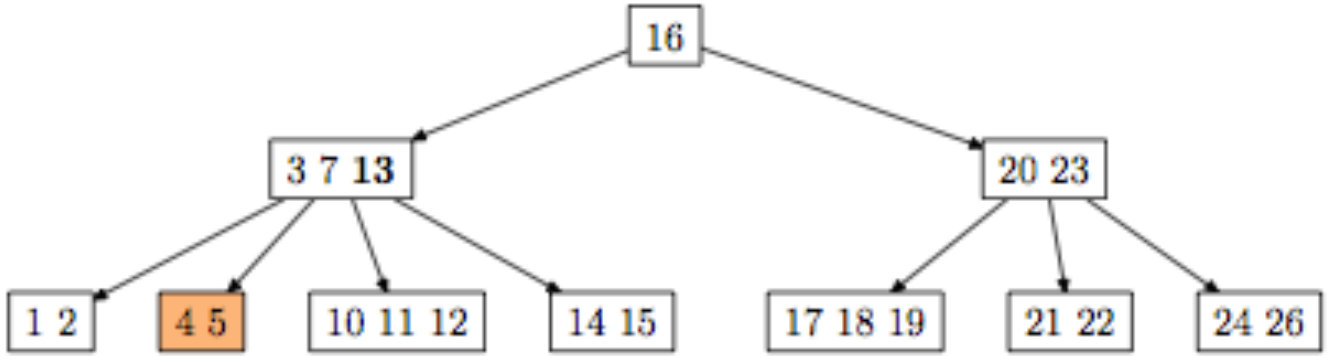
6 deleted:



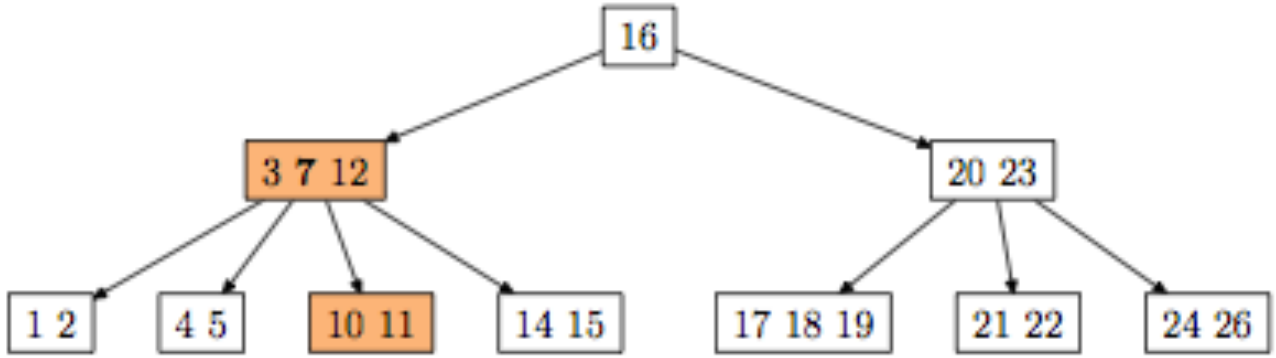
Deleting a key — Case 2a

$t = 3$

Initial tree:



13 deleted:

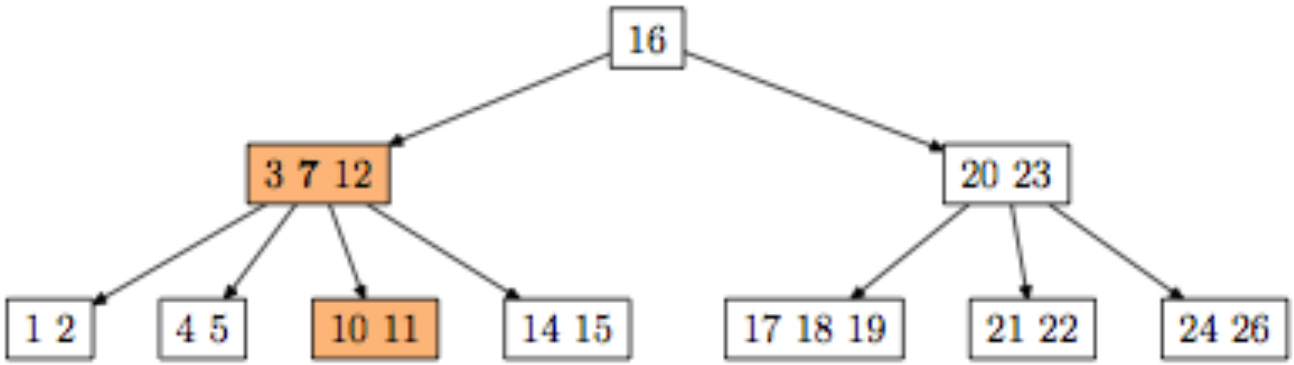


- The predecessor of 13, which lies in the preceding child of x, is moved up and takes 13's position. The preceding child had a key to spare in this case.

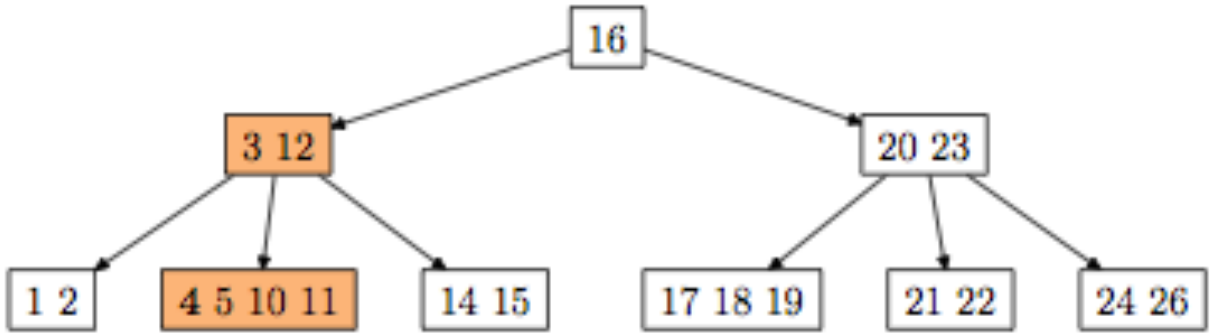
Deleting a key — Case 2c

$t = 3$

Initial tree:



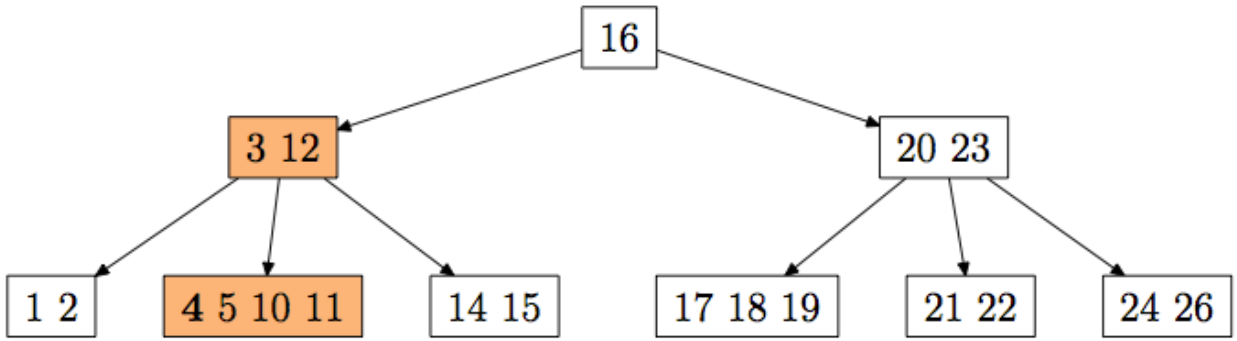
7 deleted:



- Both the preceding and successor children have $t - 1$ keys, the minimum allowed. 7 is initially pushed down and between the children nodes to form one leaf, and is subsequently removed from that leaf.

Deleting a key — Case 3b

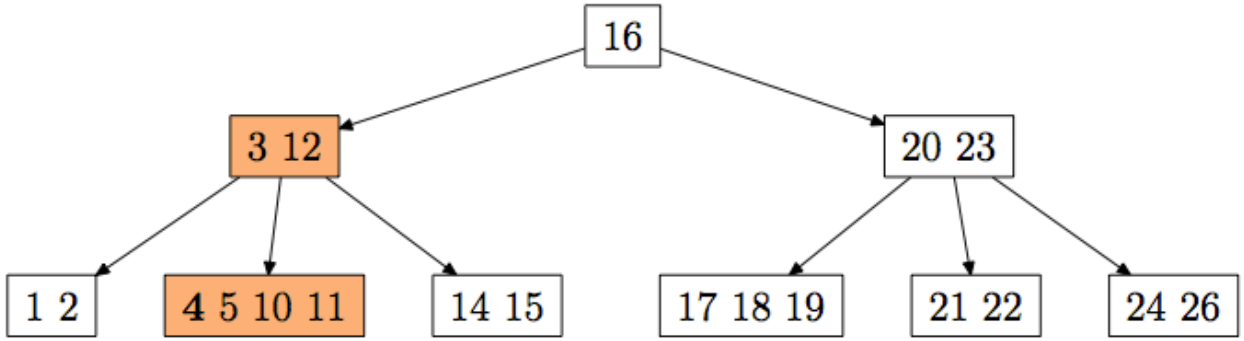
Initial tree: Key **4** to be deleted



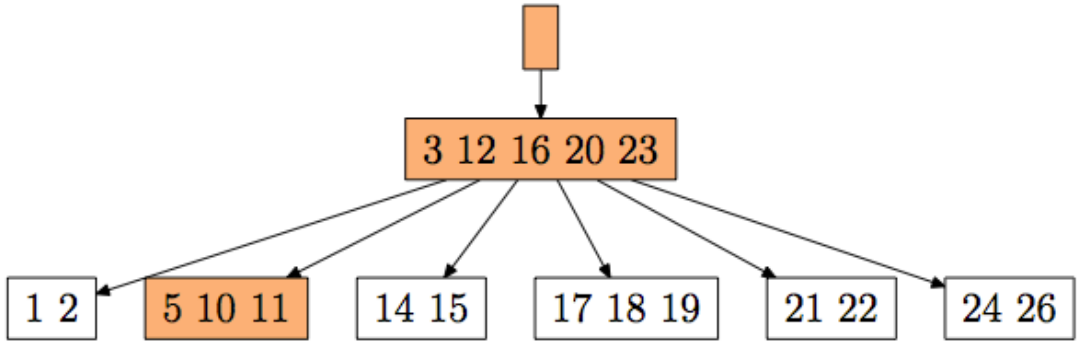
- The catchy part. Recursion cannot descend to node with keys (3, 12) because it has $t - 1$ keys. In case the two leaves to the left and right had more than $t - 1$ keys, (3, 12) could take one and 3 would be moved down.
- Also, the sibling of (3, 12) has also $t - 1$ keys, so it is not possible to move the root to the left and take the leftmost key from the sibling to be the new root.
- Therefore the root has to be pushed down merging its two children, so that 4 can be safely deleted from the leaf.

Deleting a key — Case 3b (continued)

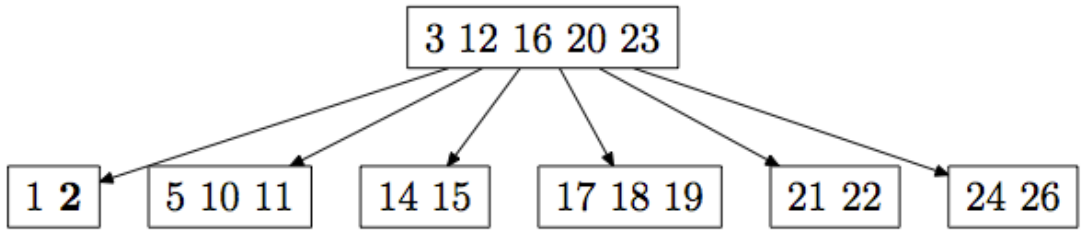
Initial tree:



4 deleted:

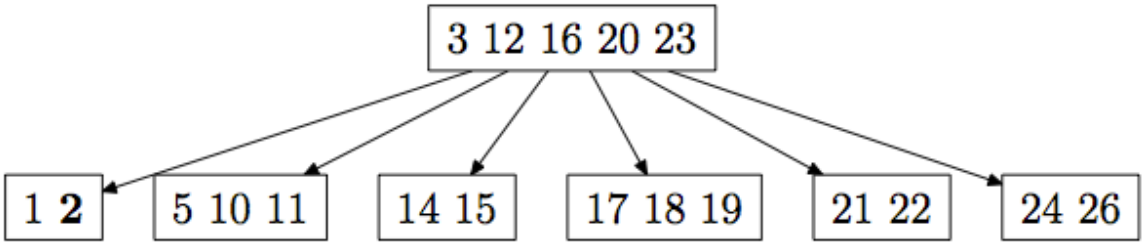


Outcome:



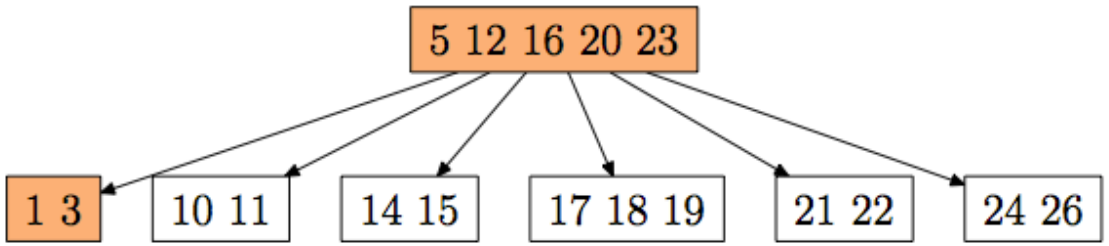
Deleting a key — Case 3a

Initial tree:



2 deleted:

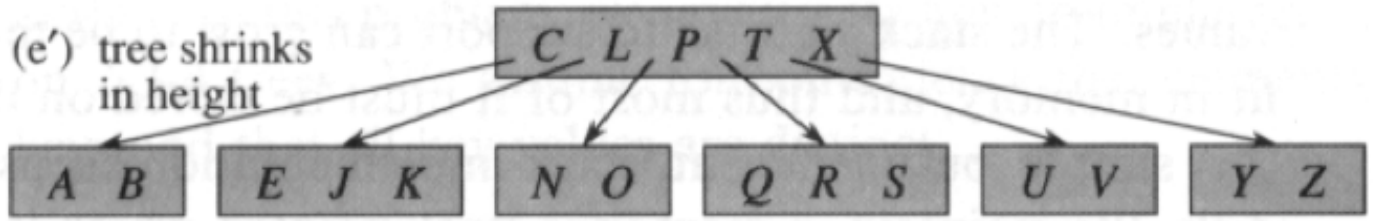
(to the previous one)



- In this case, node (1,2) has $t - 1$ keys, but the sibling to the right has t keys. Recursion moves 5 to fill 3's position, 5 is moved to the appropriate leaf, and deleted from there.

Exercises

1. Explain how to find the minimum key stored in a B-tree and how to find the predecessor of a key stored in a B-tree.
2. Show the results of deleting C, P, V, in order, from the tree depicted below.



3. Write the pseudocode for `B-Tree-Delete`.

References

- T. H. Cormen, C. E. Leiserson, R. L. Rivest. Introduction to Algorithms. The MIT Press. 2000.

  Chapter 20. B-Trees.