Advanced Algorithm Design and Analysis (Lecture 1)

SW5 fall 2004

Simonas Šaltenis

E1-215b

simas@cs.aau.dk

Overview

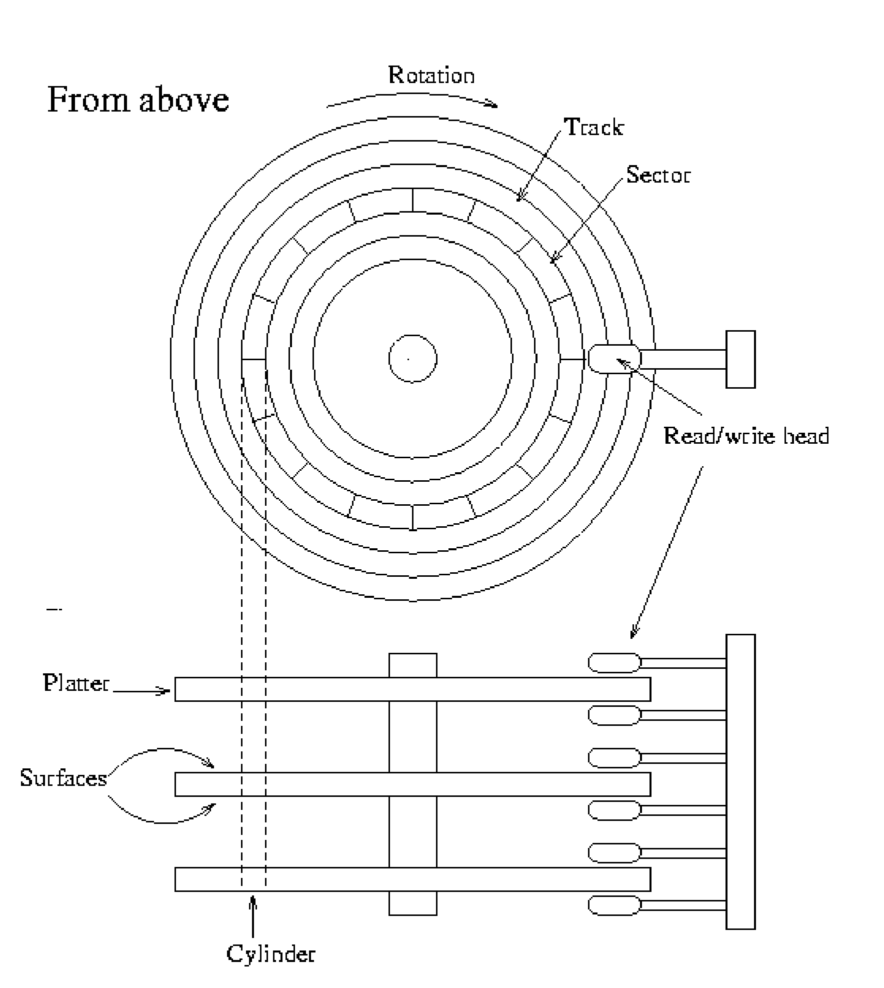
- Why do we need this course?
- Goals of the course
- Mode of work
- Prerequisites, textbook
- The first lecture – external data structures

External Mem. Data Structures

- Goals of the lecture:
  - *to understand the external memory model and the principles of analysis of algorithms and data structures in this model;*
  - *to understand why main-memory algorithms are not efficient in external memory;*
  - *to understand the algorithms of B-tree and its variants and to be able to analyze them.*

Hard disk I

- In *real systems*, we need to cope with data that does not fit in main memory
- Reading a data element from the hard-disk:
  - *Seek* with the head
  - *Wait* while the necessary sector rotates under the head
  - *Transfer* the data



Hard disk II

- *Example:* Seagate Cheetah 10K.6, 146.8Gb
  - *Seek time:* ~5ms
  - *Half of rotation:* ~3ms
  - *Transferring 1 byte:* 0.000016ms
- Conclusions:
  1. It makes sense to read and write in large blocks – *disk pages* (2 – 16Kb)
  2. *Sequential* access is much faster than *random* access
  3. Disk access is much slower than main-memory access

External memory model

- Running time: in *page accesses* or “I/Os”
- B – page size is an important parameter:
  - Not “just” a constant: $O(log_2 n) \neq O(log_B n)$
- Constant size main memory buffer of “current” pages is assumed.
- Operations:
  - DiskRead(x:pointer_to_a_page)
  - DiskWrite(x:pointer_to_a_page)
  - AllocatePage():pointer_to_a_page

Writing algorithms

- The typical working pattern for algorithms:

```
…
x ← a pointer to some object
DiskRead(x)
operations that access and/or modify x
DiskWrite(x) //omitted if nothing changed
other operations, only access no modify
…
```

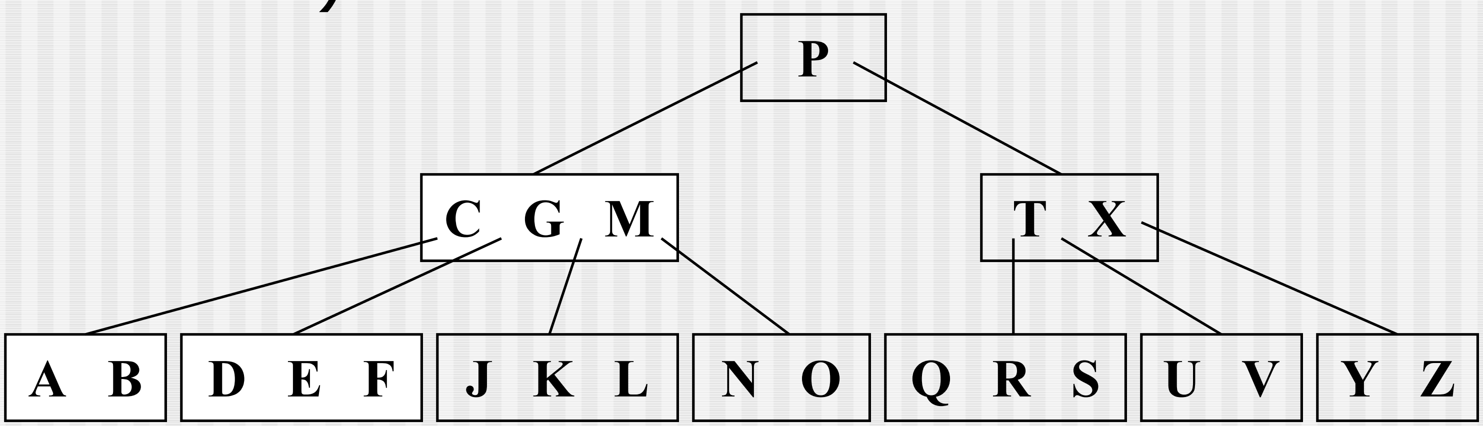
- Pointers in data-structures point to disk-pages, not locations in memory

“Porting” main-memory DSs

- Why not “just” use the main-memory data structures and algorithms in external memory?
- Consider a balanced binary search tree.
  - *A, B, C, D, E, F, G, H, I*
- Options:
  - Each node gets a separate disk page – waist of space and search is just $O(log_2 n)$
  - Nodes are somehow packed to make disk pages full – search may still be $O(log_2 n)$ in the worst-case

B-tree: Definition I

- We are concerned only with keys
- B-tree is a balanced tree, and all leaves have the same depth: *h*
- The nodes have high *fan-out* (many children)

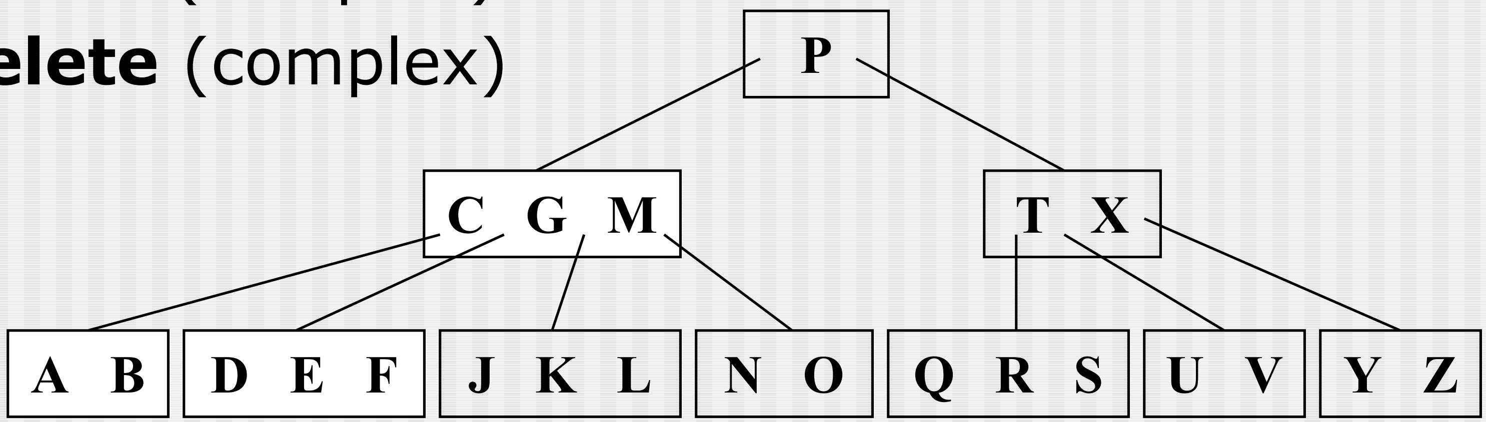


B-tree: Definition II

- Non-leaf node structure:
  - A list of alternating pointers to children and keys: p_1, key_1, p_2, key_2 … p_n, key_n, p_{n+1}
  - $key_1 \leq key_2 \leq \ldots \leq key_n$
  - For any key k in a sub-tree rooted at p_i, it is true: $key_i \leq k \leq key_{i+1}$
- Leaf node is a sorted list of keys.
- Lets draw a B-tree:
  - *A, B, C, D, E, F, G, H, I, J, K*

B-tree operations

- An implementation needs to support the following B-tree operations (corresponds to Dictionary ADT operations):
  - **Search** (simple)
  - **Create** an empty tree (trivial)
  - **Insert** (complex)
  - **Delete** (complex)



Btree and Bnode ADTs

- Btree ADT:
  - *root*():Bnode – *T.root*() gives a pointer to a root node of a tree T
- Bnode ADT:
  - *n*():int – *x.n*() the number of keys in node *x*
  - *key*(*i:*int):key_t – *x.key*(*i*) the *i*-th key in *x*
  - *p*(*i:*int):Bnode – *x.p*(*i*) the *i*-th pointer in *x*
  - *leaf*():bool - *x.leaf*() is true if *x* is a leaf
- Simplified syntax for *set* methods:
  - e.g., *x.n*() ← 0, instead of *x.setn*(*0*)

Search

- Straightforward generalization of a binary tree search:
  - Initial call **BtreeSearch(**T.root(), k**)**

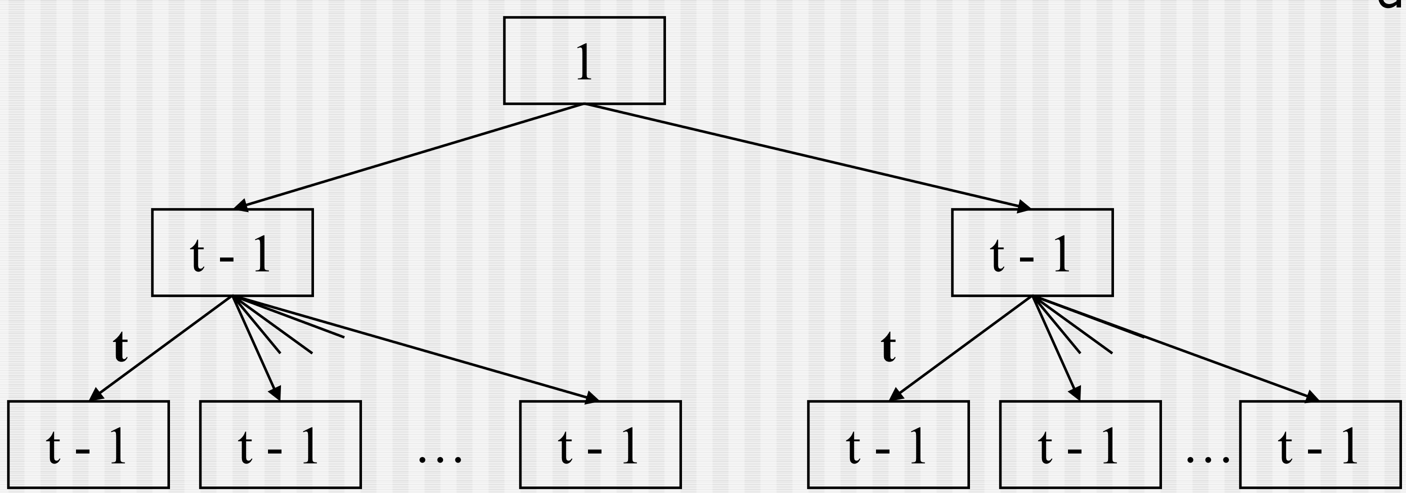
```
BTreeSearch(x,k)
i ← 1
while i ≤ x.n() and k > x.key(i)
   i ← i+1
if i ≤ x.n() and k = x.key(i) then
   return (x,i)
if x.leaf() then
   return NIL
   else DiskRead(x.p(i))
        return BTtreeSearch(x.p(i),k)
```

Analysis of Search I

- B-tree of a *minimum degree* t ($t \geq 2$):
  - All nodes except the root node have between t and $2t$ children (i.e., between $t-1$ and $2t-1$ keys).
  - The root node has between 0 and $2t$ children (i.e., between 0 and $2t-1$ keys)

Analysis of Search II

- For B-tree containing $n \geq 1$ keys and minimum degree $t \geq 2$, the following restriction on the height h holds: $h \leq \log_t \frac{n+1}{2}$
  - Why? The highest tree:



depth	#of nodes
0	1
1	2
2	2t

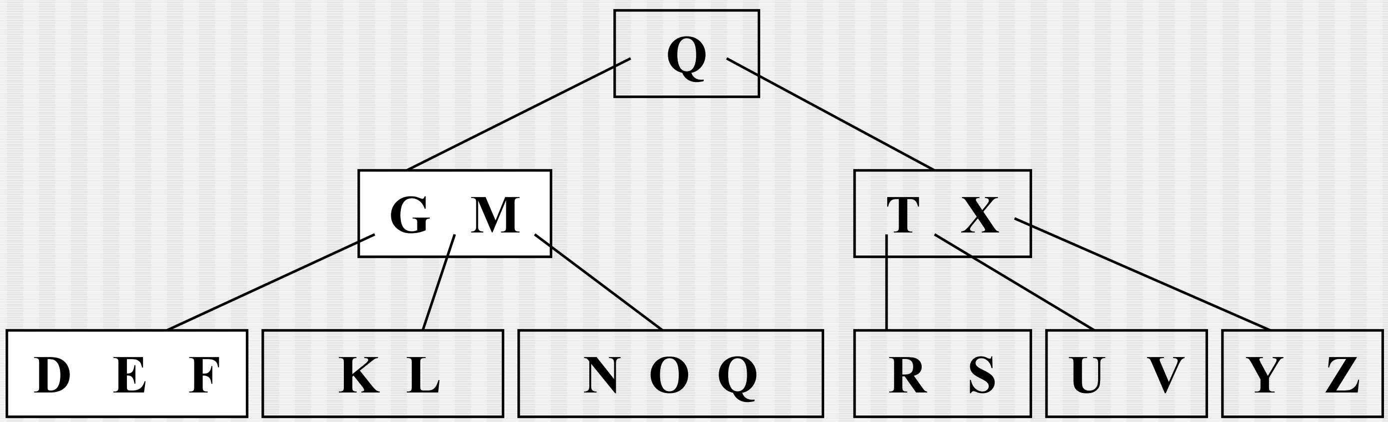
Analysis of search III

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 2t^h - 1 \quad \Rightarrow \quad h \leq \log_t \frac{n+1}{2}$$

- Thus, the worst-case running time is:
  - $O(h) = O(\log_t n) = O(\log_B n)$
- Comparing with the "straightforward" balanced binary search tree ($O(\log_2 n)$):
  - a factor of $O(\log_2 B)$ improvement

Insert

- Insertion is always performed at the leaf level
- Let’s do an example (t = 2):
  - Insert: *H, J, P*

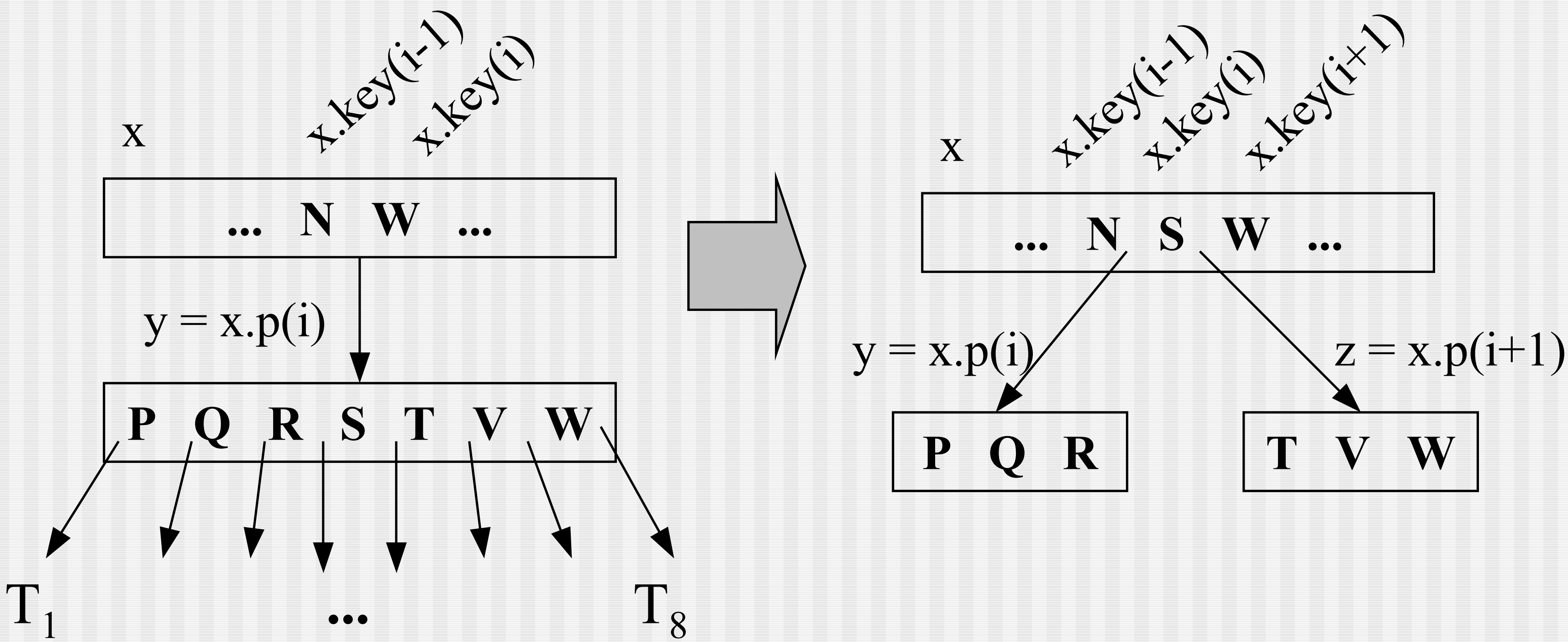


Splitting Nodes

- Nodes fill up and reach their maximum capacity $2t - 1$
- Before we can insert a new key, we have to “make room,” i.e., split a node

Splitting Nodes II

- Result: one key of *x* moves up to parent + 2 nodes with *t*-1 keys
  - How many I/O operations?



Insert I

- Skeleton of the algorithm:
  - *Down-phase:* recursively traverse down and find the leaf
  - Insert the key
  - *Up-phase:* if necessary, split and propagate the splits up the tree
- Assumptions:
  - In the *down-phase* pointers to traversed nodes are saved in the stack. Function *parent*(*x*) returns a parent node of *x* (pops the stack)
  - *split*(*y*:Bnode):(*zk*:key_t, *z*:Bnode)

Insert II

```
DownPhase(x,k)
i ← 1
while i ≤ x.n() and k > x.key(i)
   i ← i+1
if x.leaf() then
   return x
   else DiskRead(x.p(i))
        return DownPhase(x.p(i),k)
```

```
Insert(T,k)
x ← DownPhase(T.root(), k)
UpPhase(x, k, nil)
```

Insert III

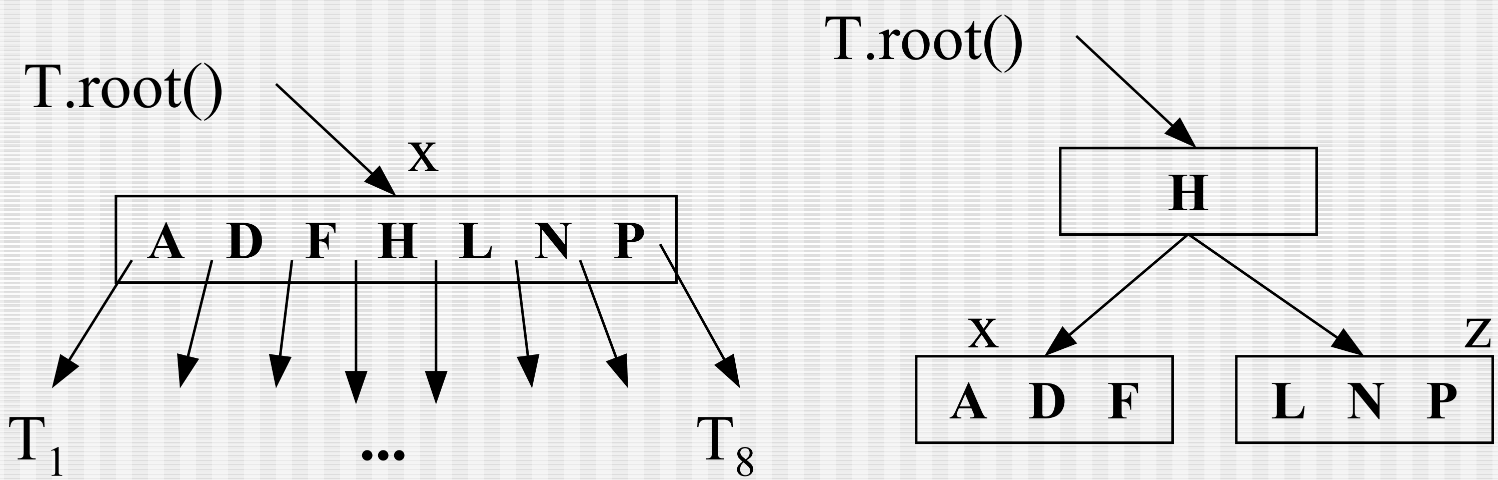
```
UpPhase(x,k,p)
if x.n() = 2t-1 then
   (zk,z) ← split(x)
   if k ≤ zk then InsertIntoNode(x,k,p)
             else InsertIntoNode(z,k,p)
   if parent(x)=nil then (Create new root)
   else UpPhase(parent(x),zk,z)
else InsertIntoNode(x,k,p)
```

InsertIntoNode(x,k,p)

Inserts the hey *k* and the following pointer *p* (if not *nil*) into the sorted order of keys of *x*, so that all the keys before *k* are smaller or equal to *k* and all the keys after *k* are greater than *k*

Splitting the Root

- Splitting the root requires the creation of a new root



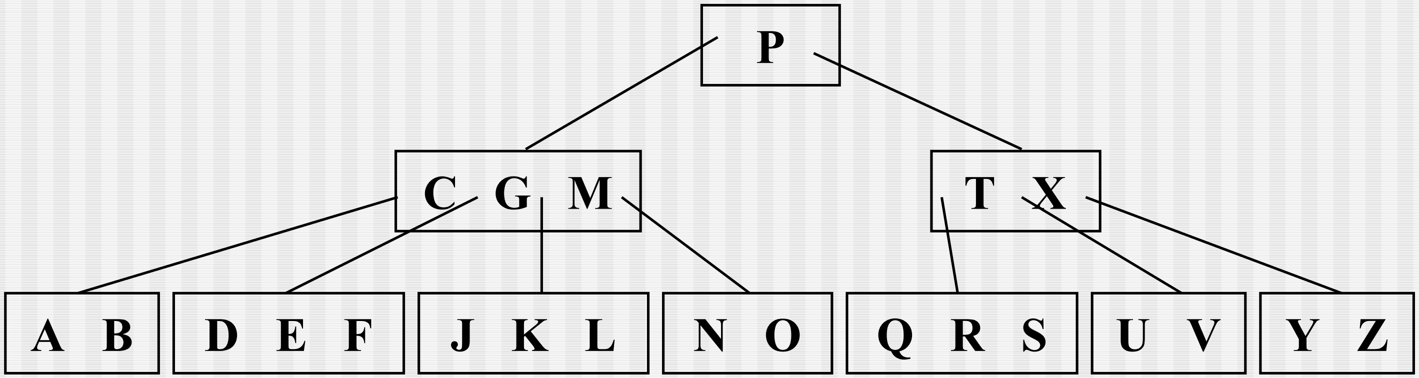
- The tree grows at the top instead of the bottom

One/Two Phase Algorithms

- Running time: $O(h) = O(log_B n)$
- Insert could be done in one traversal down the tree (by splitting all full nodes that we meet, "just in case")
- Disadvantage of the two-phase algorithm:
  - Buffer of $O(h)$ pages is required

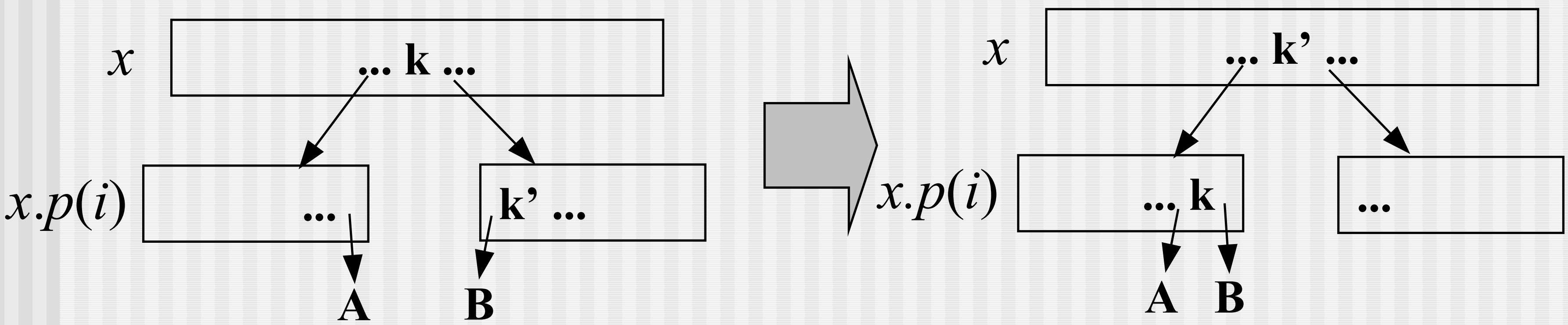
Deletion

- Case 1: A key *k* is in a non-leaf node
  - Delete its predecessor (which is always in a leaf, thus case 2) and put it in *k's* place.
- Case 2: A key is in a leaf-node:
  - Just delete it and handle under-full nodes
- Try: delete *M, B, K* (t =3)

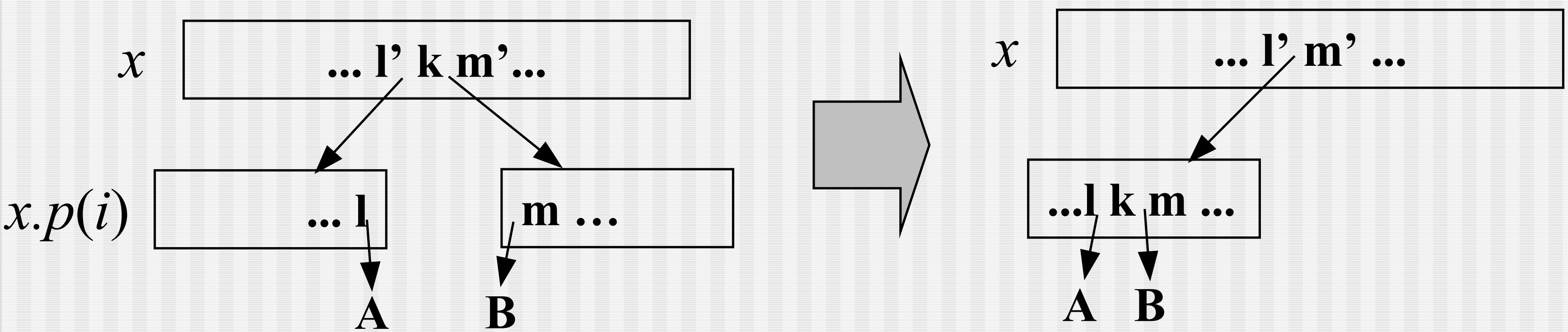


Handling Under-full Nodes

- Distributing:



- Merging:



Sequential access

- Other useful ADT operator: *successor*
  - For example, range queries: *find all accounts with the amount in the range* [100K – 200K].
  - How do you do that in B-trees?

B^+-trees

- B^+-trees is a variant of B-trees:
  - All data keys are in leaf nodes
    - The split does not *move* the middle key to the parent, but *copies* it to the parent!
  - Leaf-nodes are connected into a (doubly) linked list
  - How the range query is performed?
    - Compare with the B-tree