Data Structures and Algorithms for External Storage

External sorting. Index files.

External Storage

- Secondary memory
  - Typically organized in *blocks*
  - Basic operations involve *buffers*
- Cost measure
  - Disk: seek time, latency time
  - Block accesses
- Data typically stored in *files*
- Files
  - Sequential access
  - Direct access

Files

- All algorithms so far assumed that all elements of a (large) array can be accessed randomly.
- If the array is too large to fit in main memory, it has to be kept on a secondary storage device.
- Typically, if data is organized as *sequential files*, which guarantee (in average) constant access time only for *strictly sequential* read and write operations.

Storing Information in Files

- Typical operations on files:
  - *insert* a particular record into a particular file.
  - *delete* from a particular file all records having a designated key value in each of a designated set of fields.
  - *modify* all records in a particular file by setting to designated values certain fields in those records that have a designated value in each of another set of fields.
  - *retrieve* all records having designated values in each of a designated set of fields.

External Sorting

- External sorting: sorting data stored on secondary memory (typically as files)
- Cost measures:
  - *Number of block accesses*
    - (The number of steps required to sort n records)
    - (The number of comparisons between keys needed to sort n records (if the comparison is expensive))
    - (The number of times the records must be moved)
    - Note that the items in paranthesis refer to main memory

Merge Sort

- Idea: organize file into progressively larger *runs*
  - *run*: sequence of records $r_1, \ldots, r_k$, where key(r_1) ≤ key(r_2) ≤…≤ key(r_k)
  - *length of run*
  - *tail*
  - Example

7 15 29 32	8 11 13 41	16 22 31 32	1 14

- Begin with two files, say f_1 and f_2, organized into runs of length k
- Assume that:
  - The numbers of runs, including tails, on f_1 and f_2 differ by at most one,
  - At most one of f_1 and f_2 has a tail, and
  - The one with a tail has at least as many runs as the other.

Merge Sort for Files

```
procedure getrecord ( i: integer ); { advance file
f_i, but
    not beyond the end of the file or the end of the run.
    Set fin[i] if end of file or run found }
  begin
    used[i] := used[i] + 1;
    if (used[i] = k) or
        (i = 1) and eof(f1) or
        (i = 2) and eof(f2) then fin[i]:= true
    else if i = 1 then read(f1, current[1])
    else read(f2, current[2])
  end; { getrecord }
```

```
procedure merge ( k: integer; { the input run length }
    f1, f2, g1, g2: file of recordtype );

var
outswitch: boolean; { tells if writing g1 (true) or g2 (false) }
winner: integer; { selects file with smaller key in current record }
used: array [1..2] of integer; { used[j] tells how many
  records have been read so far from the current run of file f_j }
fin: array [1..2] of boolean; { fin[j] is true if we have
  finished the run from f_j - either we have read k records,
  or reached the end of the file of f_j }
current: array [1..2] of recordtype; { the current records
  from the two files }
```

Merge Sort for Files

```
begin { merge }
  outswitch := true; { first merged run goes to g 1 }
  rewrite(g 1); rewrite(g 2);
  reset(f 1); reset(f 2);
  while not eof(f 1) or not eof(f 2) do begin
{ merge two file }
     { initialize }
     used[1] := 0; used[2] := 0;
     fin[1] := false; fin[2] := false;
     getrecord(1); getrecord(2);
     while not fin[1] or not fin[2] do begin { merge two runs
}
        { select winner }
        if fin[1] then winner : = 2
           { f 2 wins by "default" - run from f 1 exhausted }
        else if fin[2] then winner := 1
           { f 1 wins by default }
        else { neither run exhausted }
             if current[1].key < current[2].key then
inner := 1
             else winner := 2;
           { write winning record }
           if outswitch then write(g 1,
urrent[winner])
           else write(g 2, current[winner]);
           { advance winning file }
           getrecord(winner)
        end;
        { we have finished merging two runs - switch output
           file and repeat }
        outswitch := not outswitch
     end
  end; { merge }
```

Mergesort Example

28 3 93 10 54 65 30 90 10 69 8 22
31 5 96 40 85 9 39 13 8 77 10

(a) initial files

28 31 | 93 96 | 54 85 | 30 39 | 8 10 | 8 10
3 5 | 10 40 | 9 65 | 13 90 | 69 77 | 22

(b) organized into runs of length 2

3 5 28 31 | 9 54 65 85 | 8 10 69 77
10 40 93 96 | 13 30 39 90 | 8 10 22

(c) organized into runs of length 4

3 5 10 28 31 40 93 96 | 8 8 10 10 22 69 77
9 13 30 39 54 65 85 90 |

(d) organized into runs of length 8

3 5 9 10 13 28 30 31 39 40 54 65 85 90 93 96
8 8 10 10 22 69 77

(e) organized into runs of length 16

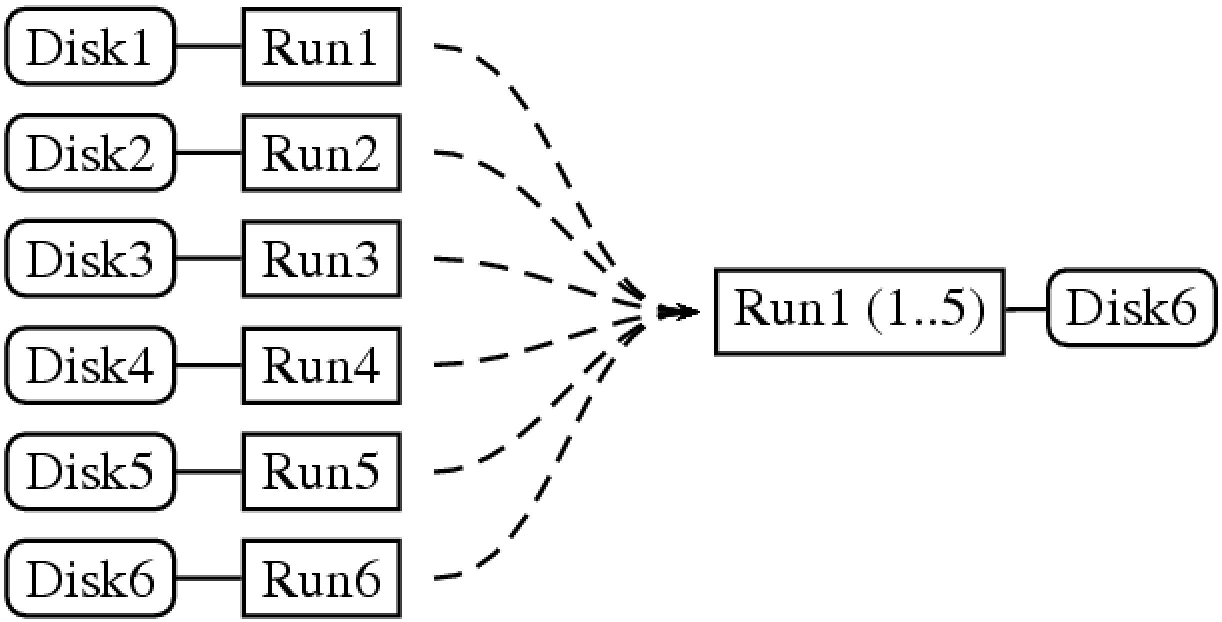
3 5 8 8 9 10 10 10 13 22 28 30 31 39 40 54 65 69 77 85 90 93 96

(f) organized into runs of length 32

Speed up Mergesort

- Begin with a pass that:
  - reads k records in memory,
  - sorts them with (quicksort),
  - writes them back,
  - then merge
- Use more channels to secondary memory
  - to make efficient use of processor speed
- Carefully select run to replenish if runs are much larger than block size
  - Based on the last keys compared

Speed up Mergesort Example



Multiway Merge

- If reading and writing between main and secondary memory is the bottleneck, perhaps we could save time if we had more the one data channel. Suppose that
  - We have $2m$ disk units, each with its own channel. We could place m files, $f_1, f_2, \ldots, f_m$ on m of the disk units, say organized as runs of length k.
  - We can read m runs, one from each file, and merge them into one run of length mk. This run is placed on one of m output files g_1, $g_2, \ldots, g_m$, each getting a run in turn.
- The merging process in main memory can be carried out in $O(\log m)$ steps per record if we organize *candidate records*, into a heap
  - If we have n records, and the length of runs is multiplied by m with each pass, then after i passes runs will be of length m^i.
  - If $m^i \geq n$, that is, after $i = \log_m n$ passes, the entire list will be sorted. As $\log_m n = \log_2 n \,/\, \log_2 m$, we save by a factor of $\log_2 m$ in the number of times we read each record

Polyphase Sort

- We can perform an m-way merge sort with only $m+1$ files, as an alternative to the $2m$-file strategy:
  - In one pass, when runs from each of m files are merged into runs of the $m+1^{st}$ file, we need not use all the runs on each of the m input files. Rather, <u>each file</u>, when it becomes the <u>output file</u>, is filled with runs of a <u>certain length</u>. It uses some of these runs to help fill each of the other m files when it is their turn to be the output file.
  - Each pass produces files of a different length. Since each of the files loaded with runs on the previous m passes contributes to the runs of the current pass, the length on one pass is the sum of the lengths of the runs produced on the previous m passes. (If fewer than m passes have taken place, regard passes prior to the first as having produced runs of length 1.)

Polyphase Sort Example

after pass	f_1	f_2	f_3
initial	13(1)	21(1)	empty
1	empty	8(1)	13(2)
2	8(3)	empty	5(2)
3	3(3)	5(5)	empty
4	empty	2(5)	3(8)
5	2(13)	empty	1(8)
6	1(13)	1(21)	empty
7	empty	empty	1(34)

Alternative File Organizations

Many alternatives exist, *each ideal for some situation , and not so good in others:*

- Heap files: Suitable when typical access is a file scan retrieving all records.
- Sorted Files: Best if records must be retrieved in some order, or only a `range' of records is needed.
- Hashed Files: Good for equality selections.
  - File is a collection of *buckets.* Bucket = *primary* page plus zero or more *overflow* pages.
  - *Hashing function* h: $h(r)$ = bucket in which record r belongs. h looks at only some of the fields of r, called the *search fields.*

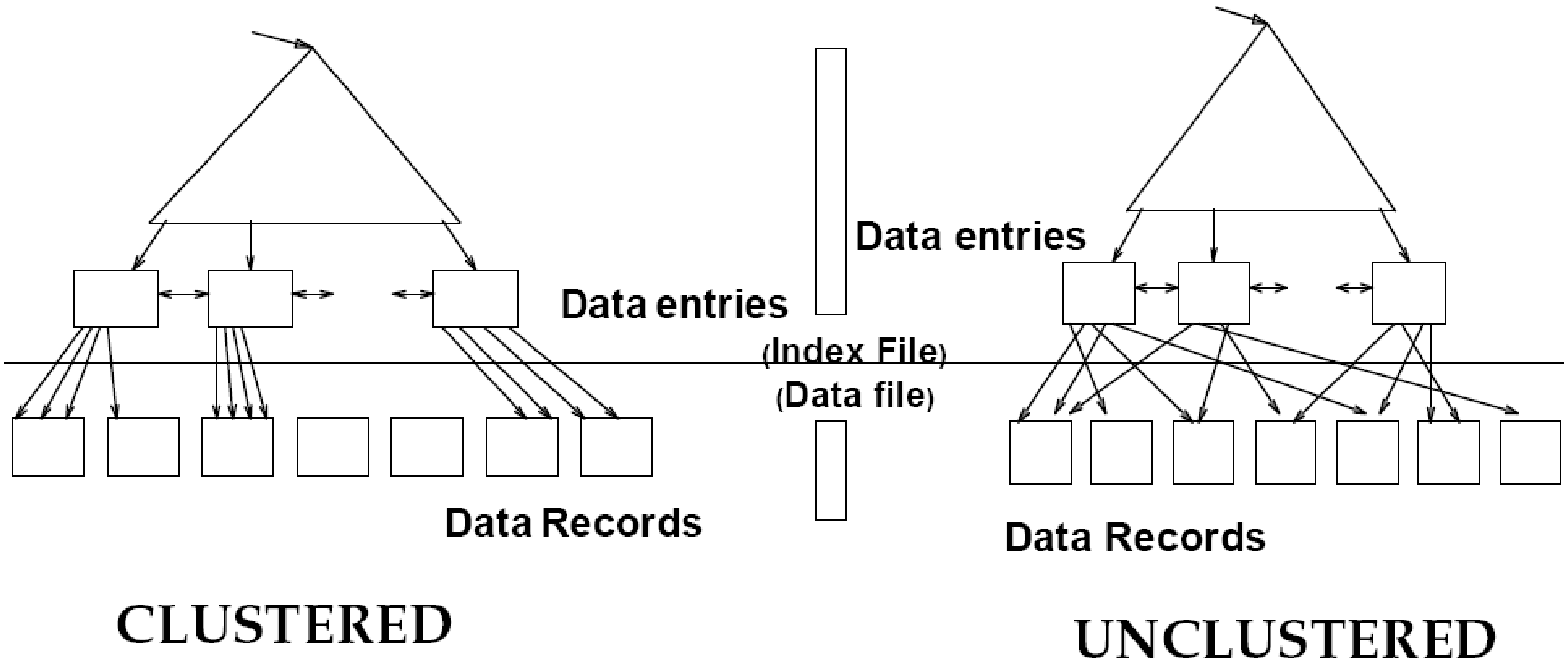
Indexes

- An *index* on a file speeds up selections on the *search key field(s)*
- Search key = any subset of the fields of a record
  - *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record).
  - Entries in an index: $(\boldsymbol{k}, \boldsymbol{r})$, where:
  - $\boldsymbol{k}$ = the key
  - $\boldsymbol{r}$ = the record OR record id OR record ids

Index Classification

- Clustered/unclustered
  - **Clustered = records *sorted* in the key order**
  - **Unclustered = no**
- Dense/sparse
  - **Dense = each record has an entry in the index**
  - **Sparse = only some records have**
- Primary/secondary
  - **Primary = on the primary key**
  - **Secondary = on any key**
  - **Some books interpret these differently**
- • B^+ tree / Hash table / …

Clustered vs. Unclustered Index



Multiway Search Trees

- Multiway Search Trees (MWSTs) are a <u>generalization</u> of BSTs
- MWST of order n:
  - Each node has n or fewer sub-trees: $S_1\ S_2 \ldots . S_m,\ m \leq n$
  - Each node has $n - 1$or fewer keys
  - $k_1\ k_2\ \ldots k_{m-1}$: $m-1$ keys in ascending order $k(S_i) \leq k_i \leq k(S_i+1)$, $k(S_{m-1}) < k(S_m)$
- Suitable for disks:
  - Nodes correspond to disk pages
  - Pros:
    - tree height is low for large n
    - fewer disk accesses
  - Cons:
    - low space utilization if non-full
    - MWSTs are non-balanced in general!

MWST Example

- Example: 4000 keys, n=5
  - At least 4000/(5–1) nodes (pages)
  - 1st level(root): 1 node, 4 keys, 5 sub-trees
  - **+**2ndlevel: 5 nodes, 20 keys, 25 sub-trees
  - **+**3rdlevel: 25 nodes, 100 keys, 125 sub-trees
  - **+**4thlevel: 125 nodes, 500 keys, 525 sub-trees
  - **+**5th level: 525 nodes, 2100 keys, 2625 sub-tress
  - **+**6th level: 2625 nodes, 10500 keys, …
  - tree height = 6 (including root)
  - If n = 11 at least 400 nodes
  - **tree height** = 3

Operations and Issues on MWSTs

- Operations
  - Search: returns pointer to node containing the key and position of key in the node
  - Insert: new key if not the tree
  - Delete: existing key
- Important Issues
  - Keep MWST balanced after insertions or deletions
  - Balanced MWSTs: B-trees, B+-trees
  - Reduce number of disk accesses
  - Data storage: two alternatives
    1. inside nodes: less sub-trees, nodes
    2. pointers from the nodes to data pages

B Trees

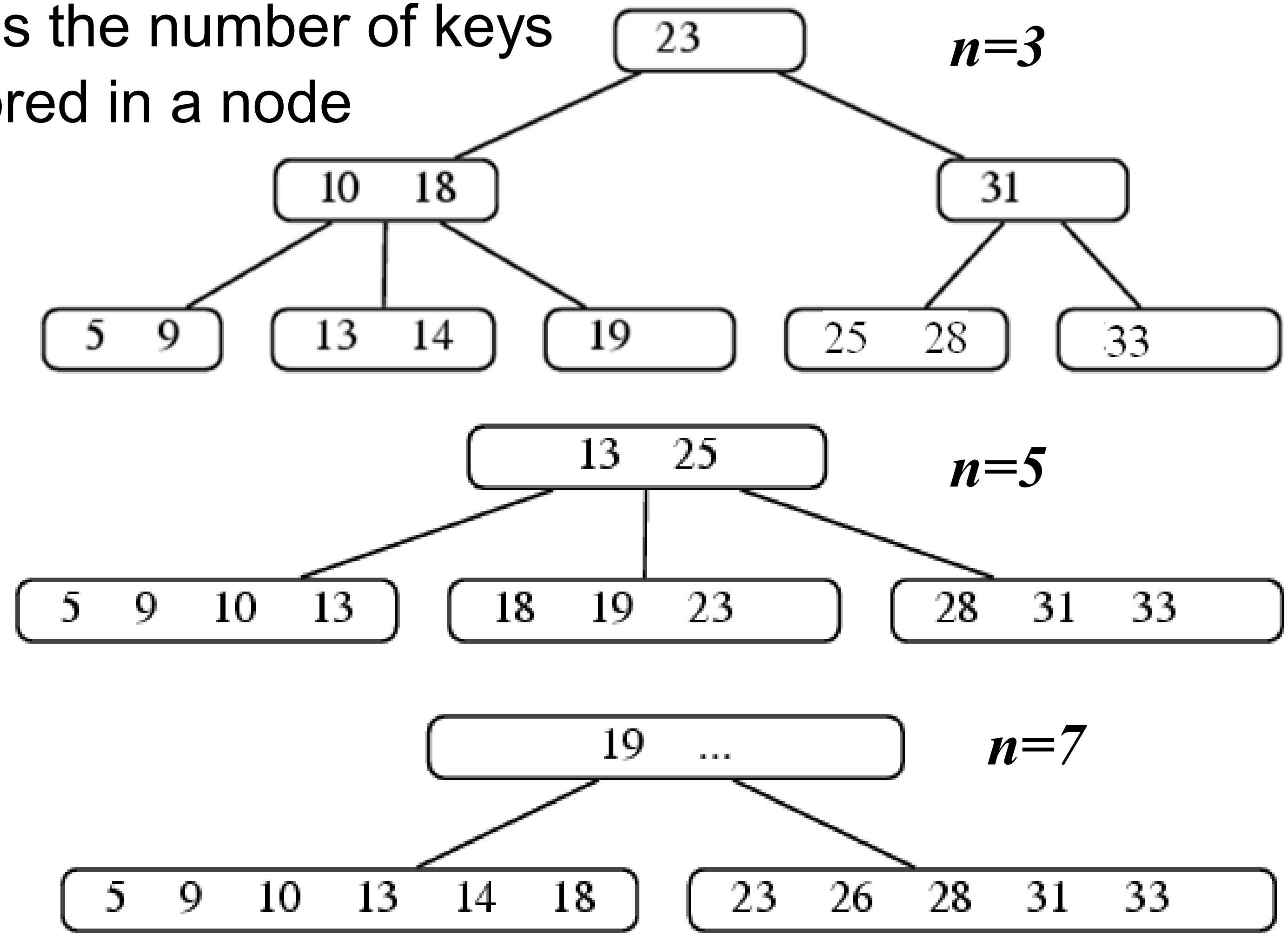
- So far search trees were limited to main memory structures
  - Assumption: the dataset organized in a search tree fits in main memory (including the tree overhead)
- Counter-example: transaction data of a bank > 1 GB per day
  - use secondary storage media (punch cards, hard disks, magnetic tapes, etc.)
- Consequence: make a search tree structure secondary-storage-enabled
- **B Trees** - Proposed by R. Bayer and E. M. McCreigh in 1972.

B-tree Definitions

- Node x has fields
  - $n[x]$: the number of keys of that the node
  - $key_1[x] \le \ldots \le key_{n[x]}[x]$: the keys in ascending order
  - $leaf[x]$: true if leaf node, false if internal node
  - if internal node, then $c_1[x], \ldots, c_{n[x]+1}[x]$: pointers to children
- Keys separate the ranges of keys in the subtrees. If k_i is an arbitrary key in the subtree $c_i[x]$ then $k_i \le key_i[x] \le k_{i+1}$
- Every leaf has the same depth
- In a B-tree of a degree t 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)

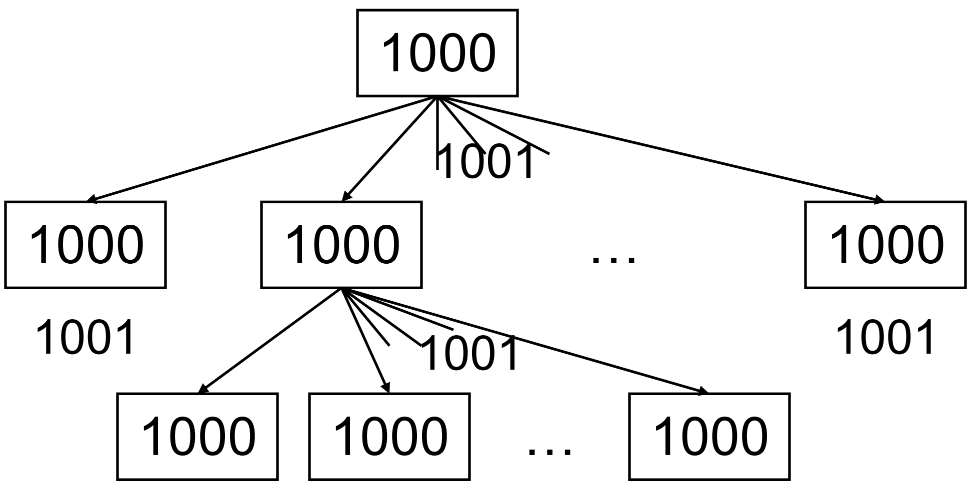
B Tree Examples

- n is the number of keys stored in a node



Binary-trees vs. B-trees

- Size of B-tree nodes: determined by the page size. One page = one node.
- A B-tree of height 2 may contain > 1 billion keys!
- Heights of Binary-tree and B-tree are logarithmic
  - **Binary-tree: logarithm of base 2**
  - **B-tree: logarithm of base, e.g., 1000**



root:1 node
1000 keys

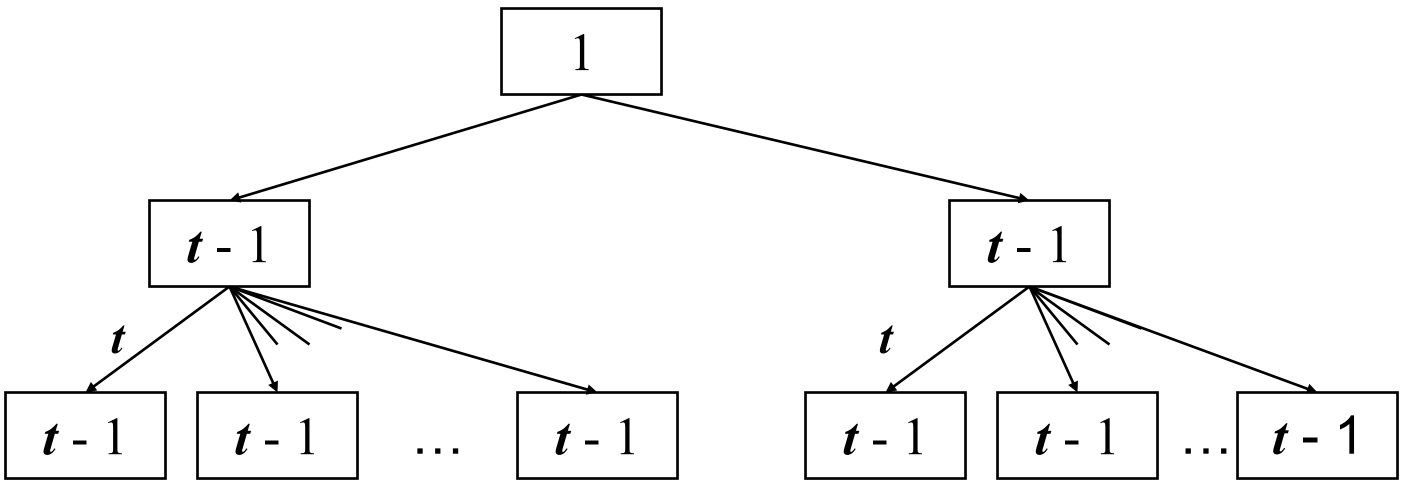
level 1:1001 nodes,
1,001,000 keys

level 2: 1,002,001 nodes,
1,002,001,000 keys

Height of a B-tree

- B-tree $\boldsymbol{T}$ of height $\boldsymbol{h}$, containing $\boldsymbol{n} \geq 1$ keys and minimum degree $\boldsymbol{t} \geq 2$, the following restriction on the height holds:

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



depth	#of nodes
0	1
1	2
2	2$\boldsymbol{t}$

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

B-tree Operations

- An implementation needs to support the following B-tree operations
  - **Searching** (simple)
  - **Creating** an empty tree (trivial)
  - **Insertion** (complex)
  - **Deletion** (complex)

Creating an Empty Tree. Searching

- **Creating:**
  - Empty B-tree = create a root & write it to disk!
- **Searching**
  - Straightforward generalization of a binary tree search

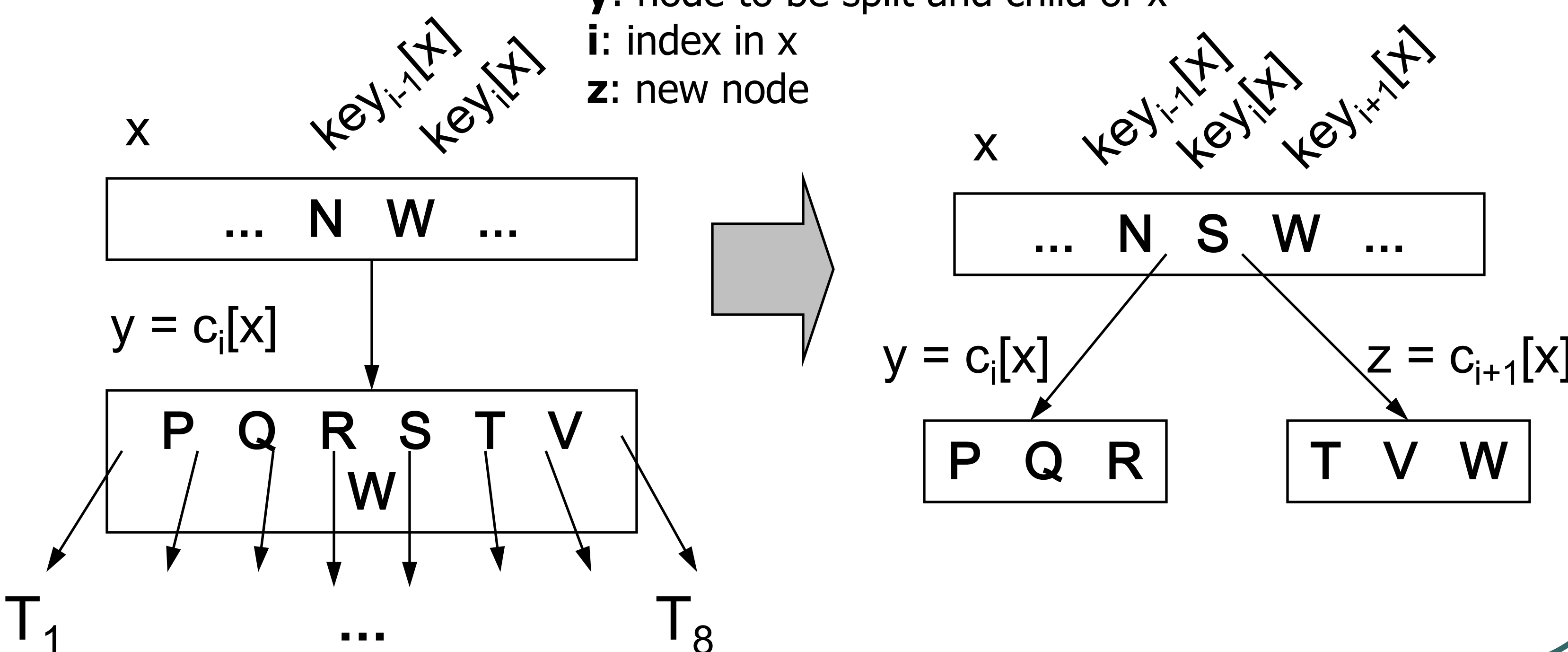
```
BTreeCreate(T)
x ← AllocateNode();
leaf[x] ← TRUE;
n[x] ← 0;
DiskWrite(x);
root[T] ← x
```

```
BTreeSearch(x,k)
i ← 1
while i ≤ n[x] and k > key_i[x]
    i ← i+1
if i ≤ n[x] and k = key_i[x] then
    return(x,i)
if leaf[x] then
    return NIL
    else DiskRead(c_i[x])
return BTtreeSearch(c_i[x],k)
```

Splitting Nodes (1)

- 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 nodes
- Result: one key of x moves up to parent + 2 nodes with $t-1$ keys

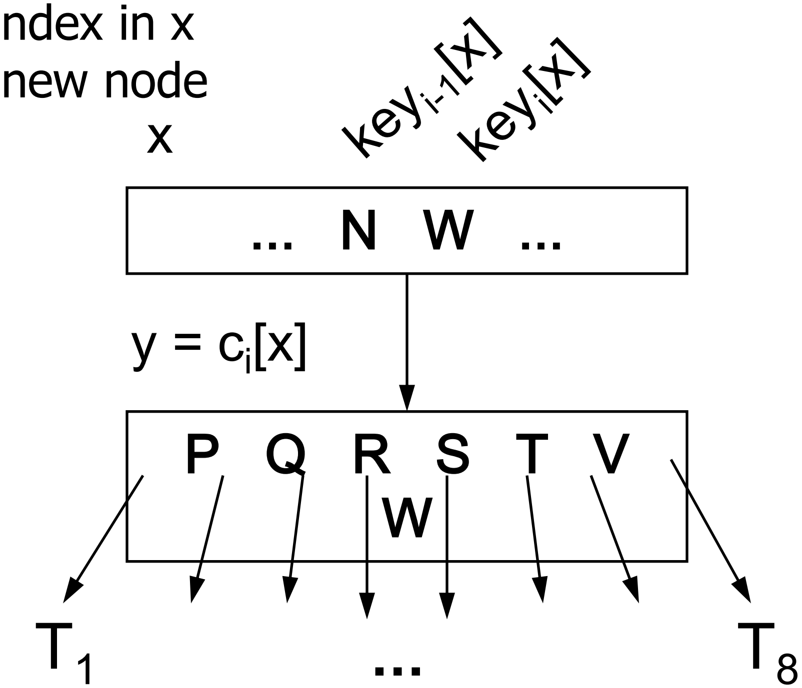
x: parent node
y: node to be split and child of x
i: index in x
z: new node



Splitting Nodes (2)

```
BTreeSplitChild(x,i,y)
z ← AllocateNode()
leaf[z] ← leaf[y]
n[z] ← t-1
for j ← 1 to t-1
    key_j[z] ← key_{j+t}[y]
if not leaf[y] then
    for j ← 1 to t
        c_j[z] ← c_{j+t}[y]
n[y] ← t-1
for j ← n[x]+1 downto i+1
    c_{j+1}[x] ← c_j[x]
c_{i+1}[x] ← z
for j ← n[x] downto i
    key_{j+1}[x] ← key_j[x]
key_i[x] ← key_t[y]
n[x] ← n[x]+1
DiskWrite(y)
DiskWrite(z)
DiskWrite(x)
```

x: parent node
y: node to be split and child of x
i: index in x
z: new node



Running Time:

- A local operation that does not traverse the tree
- $\Theta(t)$ CPU-time, since two loops run t times
- 3 I/Os

Inserting Keys

- Done recursively, by starting from the root and recursively traversing down the tree to the leaf level
- Before descending to a lower level in the tree, make sure that the node contains less than $2t - 1$ keys:
  - so that if we split a node in a lower level we will have space to include a new key

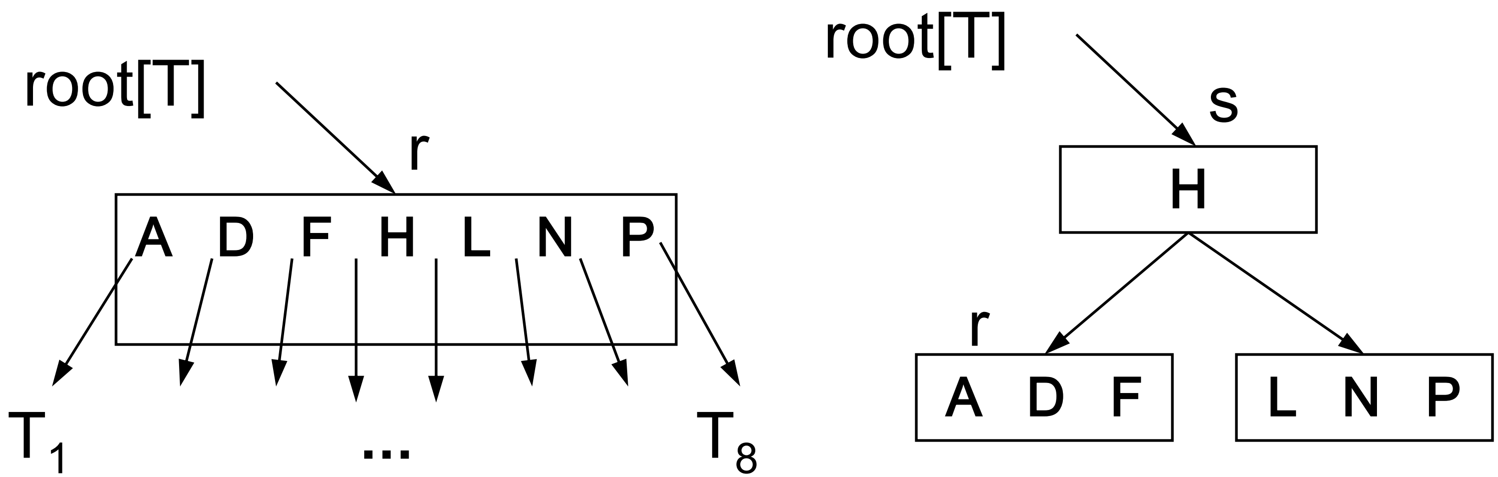
Inserting Keys (2)

- Special case: root is full (***BtreeInsert***)

```
BTreeInsert(T)
r ← root[T]
if n[r] = 2t - 1 then
    s ← AllocateNode()
    root[T] ← s
    leaf[s] ← FALSE
    n[s] ← 0
    c1[s] ← r
    BTreeSplitChild(s,1,r)
    BTreeInsertNonFull(s,k)
else BTreeInsertNonFull(r,k)
```

Splitting the Root

- Splitting the root requires the creation of a new root



- The tree grows at the top instead of the bottom

Inserting Keys

- ***BtreeNonfull*** tries to insert a key $\boldsymbol{k}$ into a node $\boldsymbol{x}$, which is **assumed to be non-full** when the procedure is called
- ***BTreeInsert*** and the recursion in ***BTreeInsertNonfull*** guarantees that this assumption is true!

Inserting Keys

BTreeInsertNonFull(x,k)

```
i ← n[x]
if leaf[x] then
   while i ≥ 1 and k < key_i[x]
      key_{i+1}[x] ← key_i[x]
      i ← i - 1
   key_{i+1}[x] ← k
   n[x] ← n[x] + 1
   DiskWrite(x)
```

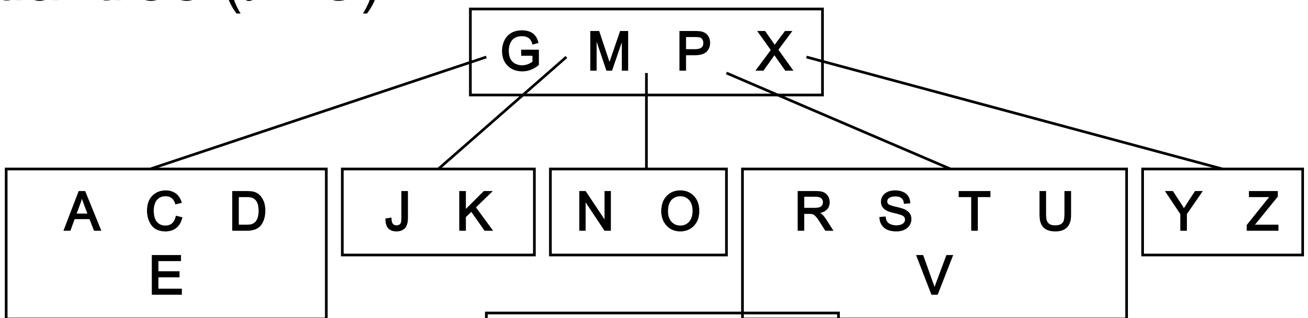
leaf insertion

```
else while i ≥ 1 and k < key_i[x]
      i ← i - 1
   i ← i + 1
   DiskRead c_i[x]
   if n[c_i[x]] = 2t - 1 then
      BTreeSplitChild(x,i,c_i[x])
      if k > key_i[x] then
         i ← i + 1
   BTreeInsertNonFull(c_i[x],k)
```

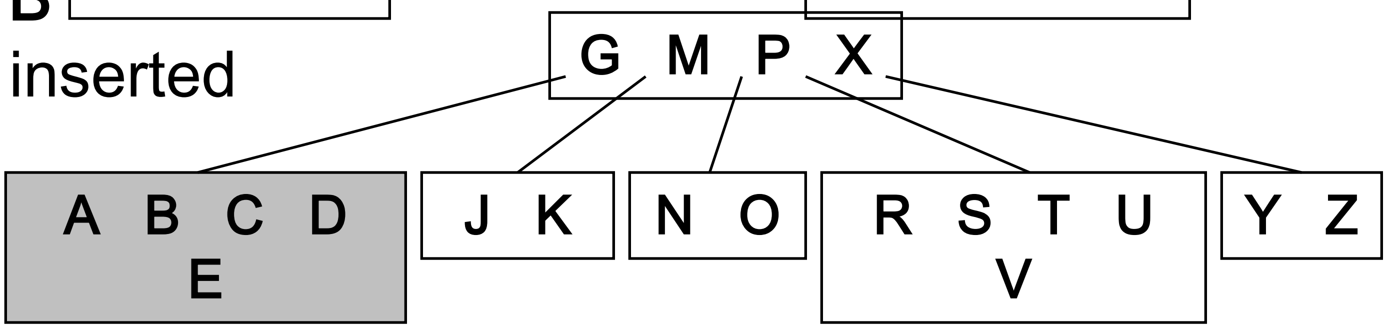
internal node: traversing tree

Insertion: Example

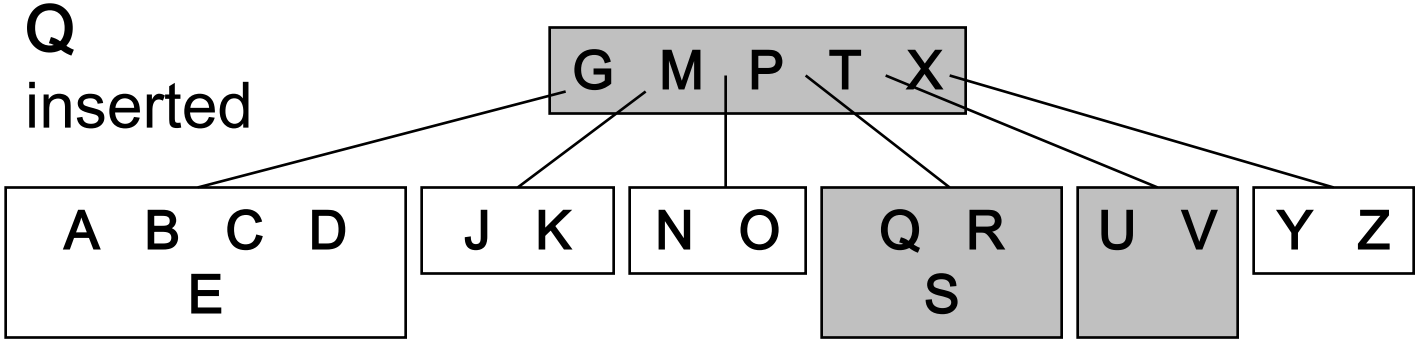
initial tree ($t = 3$)



B inserted

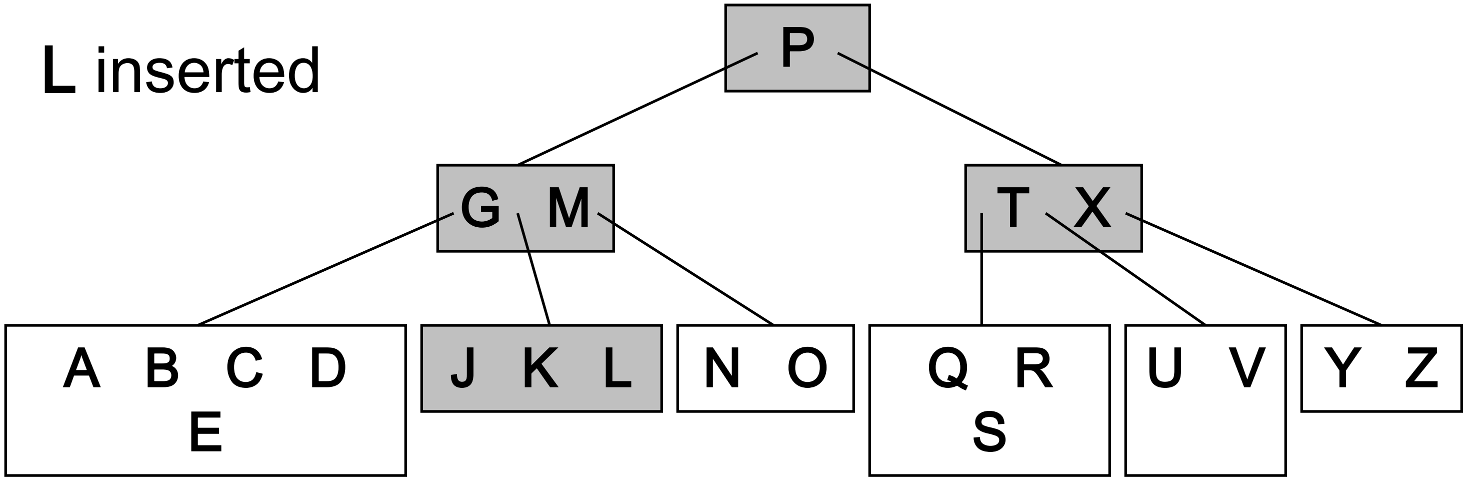


Q inserted

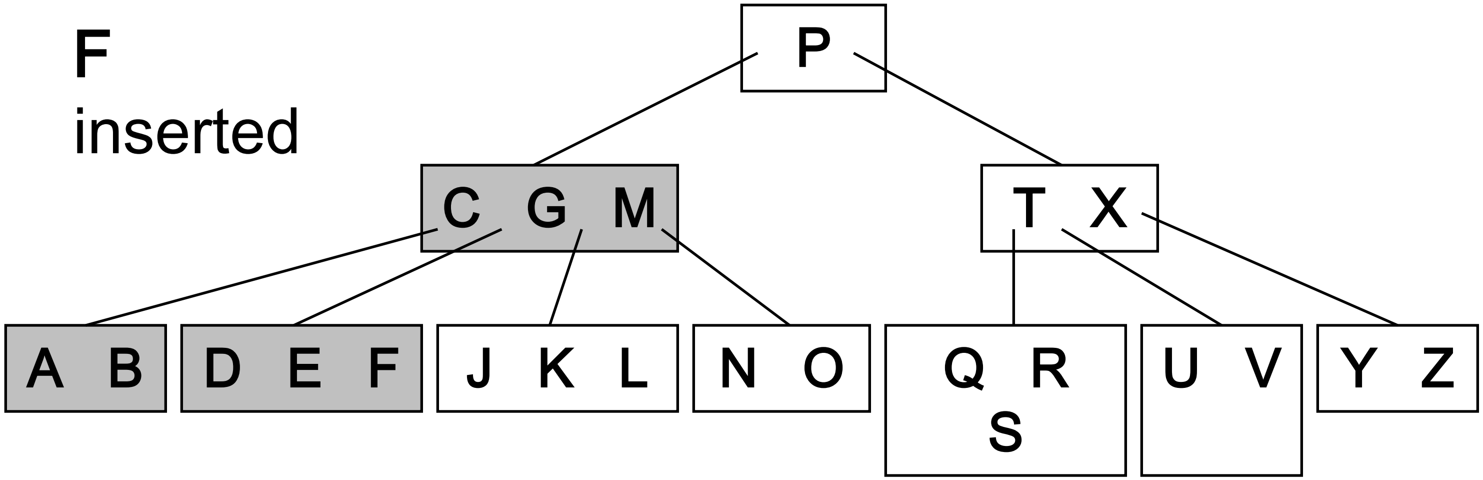


Insertion: Example (2)

L inserted



F inserted



Insertion: Running Time

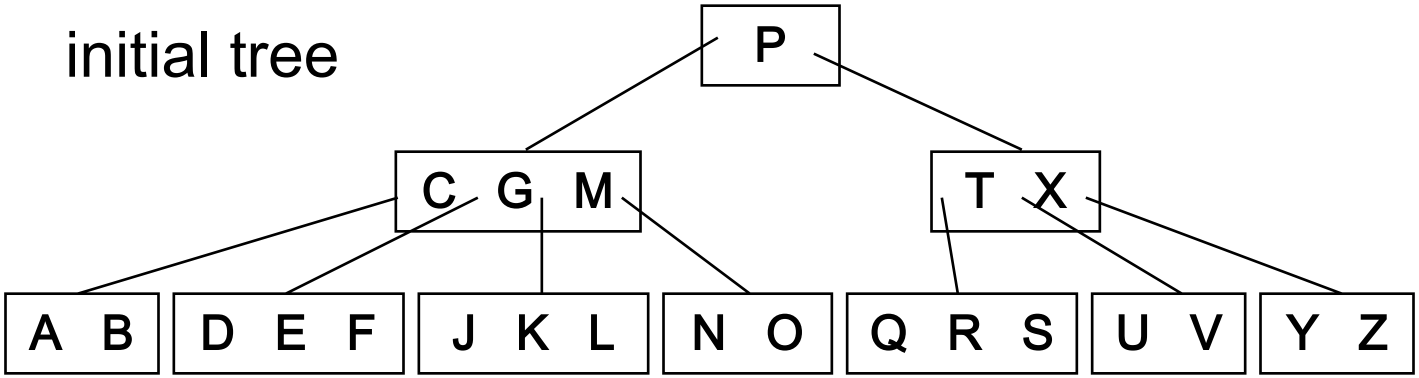
- Disk I/O: $O(h)$, since only $O(1)$ disk accesses are performed during recursive calls of BTreeInsertNonFull
- CPU: $O(th) = O(t \log_t n)$
- At any given time there are $O(1)$ number of disk pages in main memory

Deleting Keys

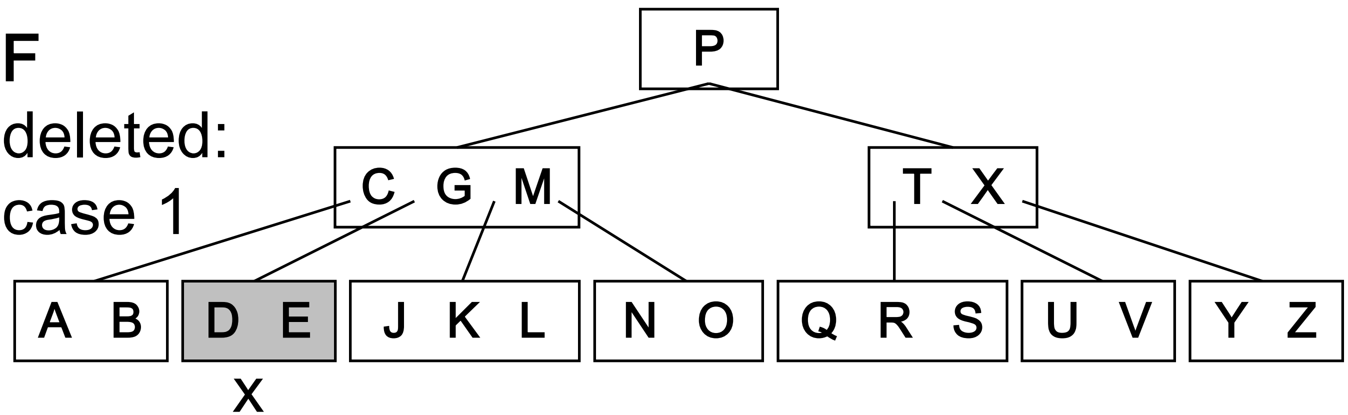
- Done *recursively*, by starting from the root and recursively *traversing down the tree to the leaf level*
- Before descending to a lower level in the tree, make sure that the node contains at least *t* keys (cf. insertion less than $2t-1$ keys)
- ***BtreeDelete*** distinguishes three different stages/scenarios for deletion
  - Case 1: key $\boldsymbol{k}$ found in leaf node
  - Case 2: key $\boldsymbol{k}$ found in internal node
  - Case 3: key $\boldsymbol{k}$ suspected in lower level node

Deleting Keys (2)

initial tree



F deleted: case 1

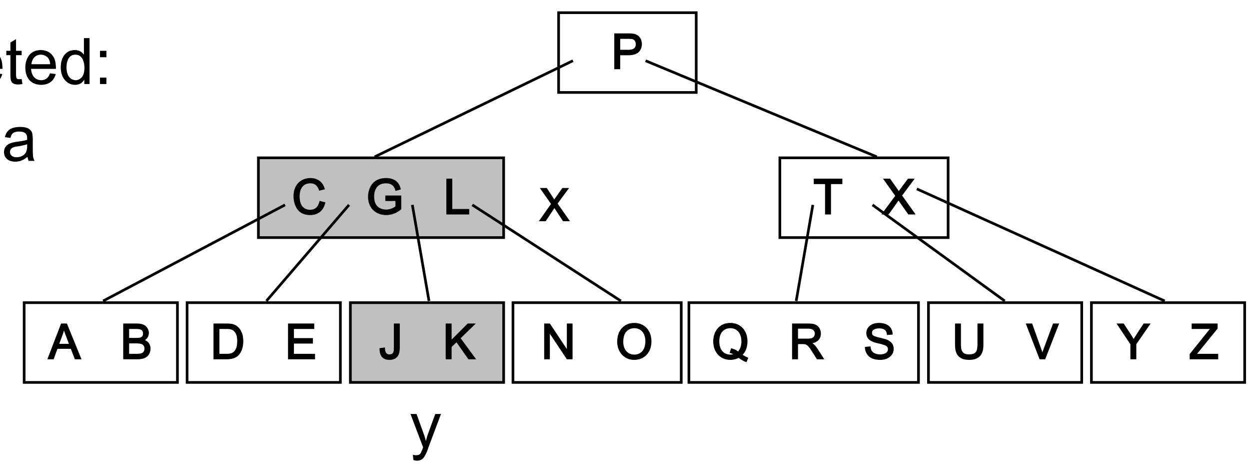


- Case 1: If the key k is in node x, and x is a leaf, delete k from x

Deleting Keys (3)

- Case 2: If the key k is in node x, and x is not a leaf, delete k from x
  - 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 sub-tree rooted at y. Recursively delete k', and replace k with k' in x.
  - b) Symmetrically for successor node z

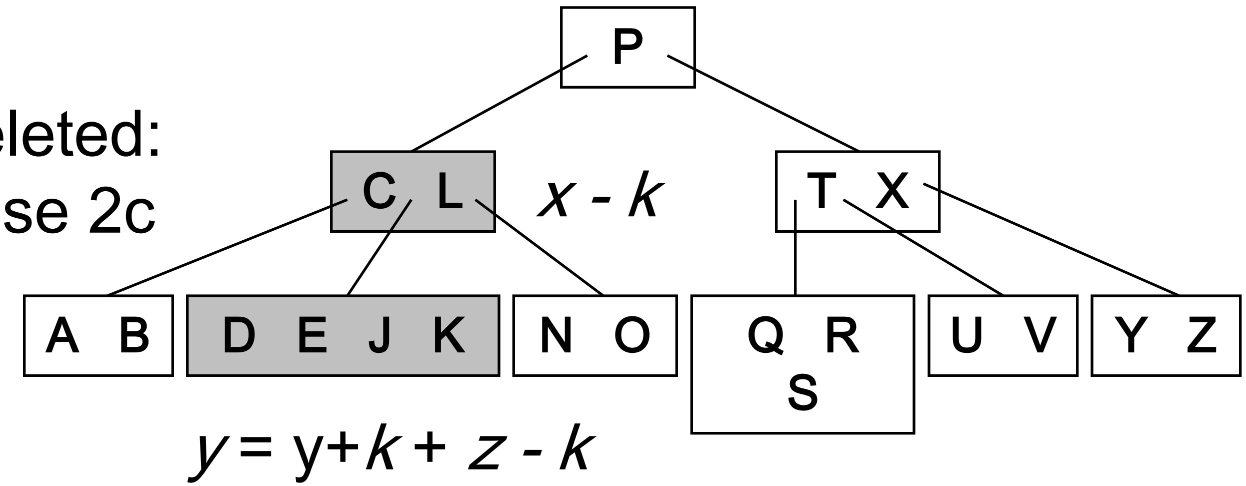
M deleted:
case 2a



Deleting Keys (4)

- If both y and z have only $t-1$ keys, ***merge*** k with the contents of z into y, so that x loses both k and the pointers to z, and y now contains $2t-1$ keys. Free z and recursively delete k from y.

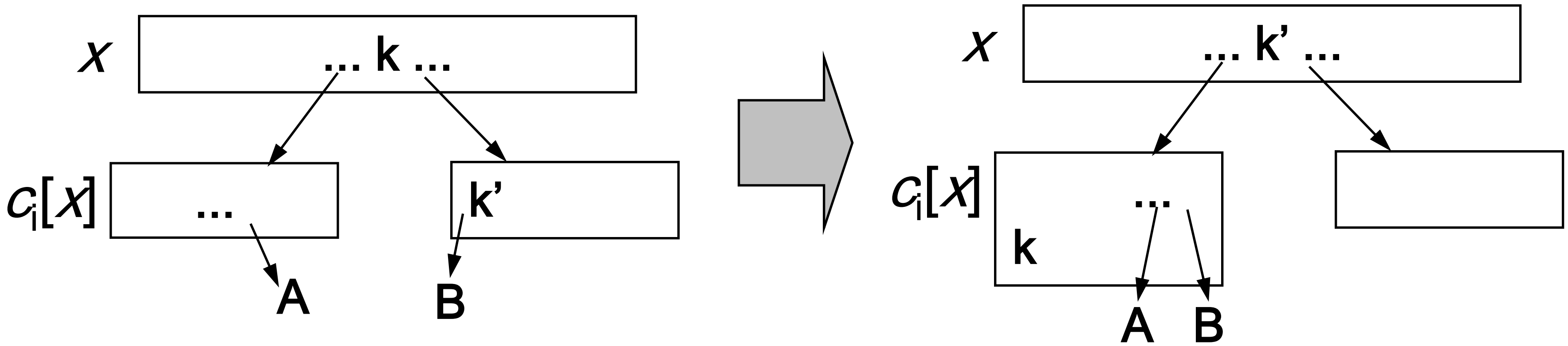
G deleted: case 2c

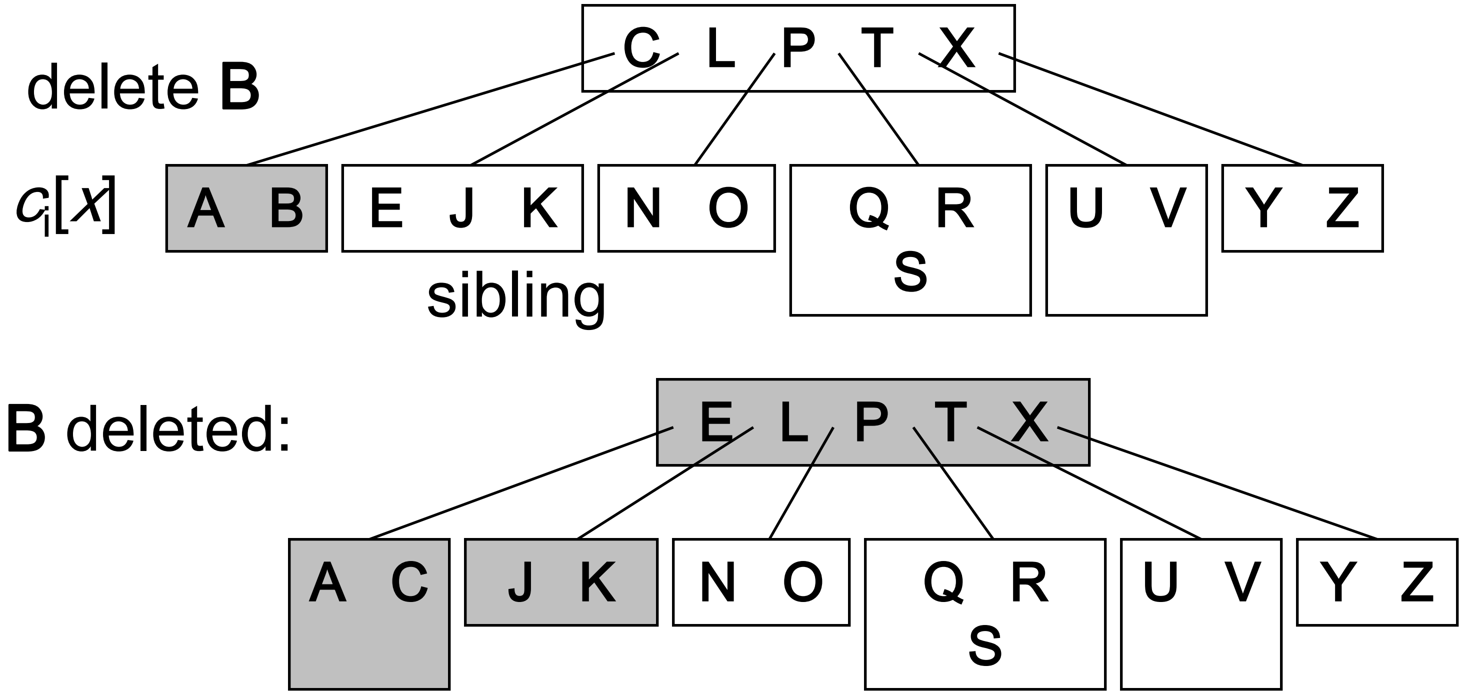


Deleting Keys - Distribution

- Descending down the tree: if k not found in current node x, find the sub-tree $c_i[x]$ that has to contain k.
- If $c_i[x]$ has only $t-1$ keys take action to ensure that we descent to a node of size at least t.
- **Case 1** (two cases exist): if $c_i[x]$ has only $t-1$ keys, but a sibling with at least t keys, give $c_i[x]$ an extra key by:
  - moving a key from x to $c_i[x]$,
  - moving a key from $c_i[x]$'s immediate left and right sibling up into x, and
  - moving the appropriate child from the sibling into $c_i[x]$ - ***distribution***

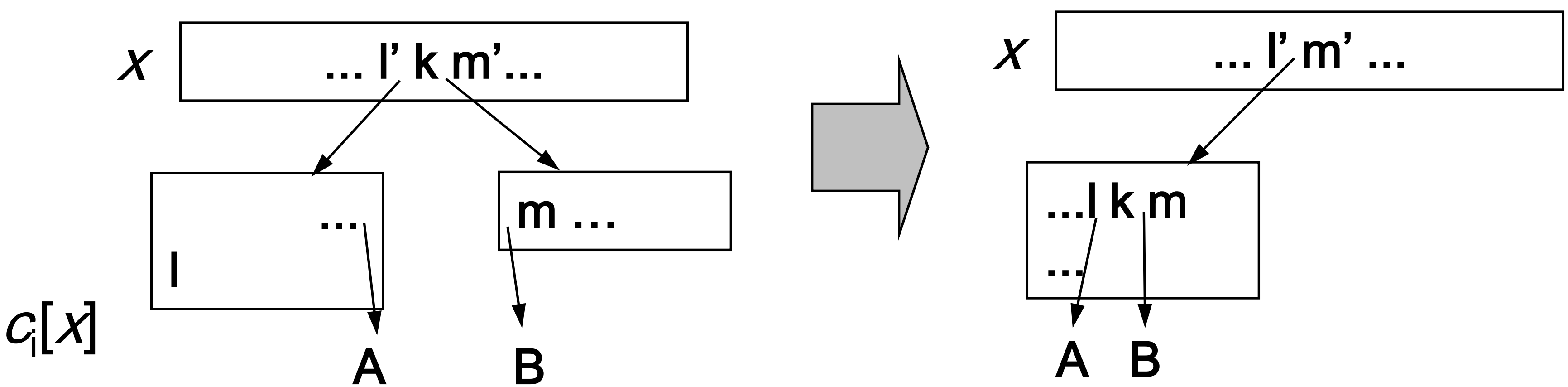
Deleting Keys – Distribution(2)





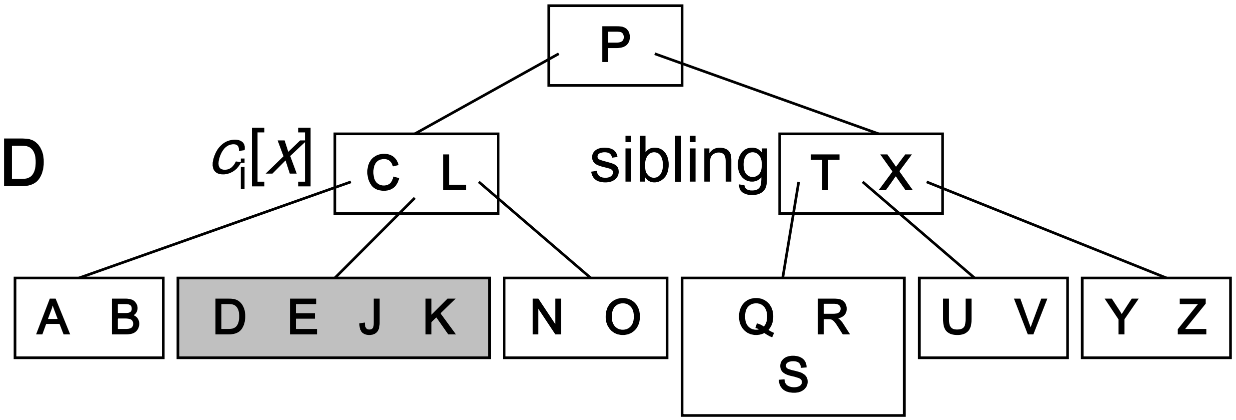
Deleting Keys - Merging

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

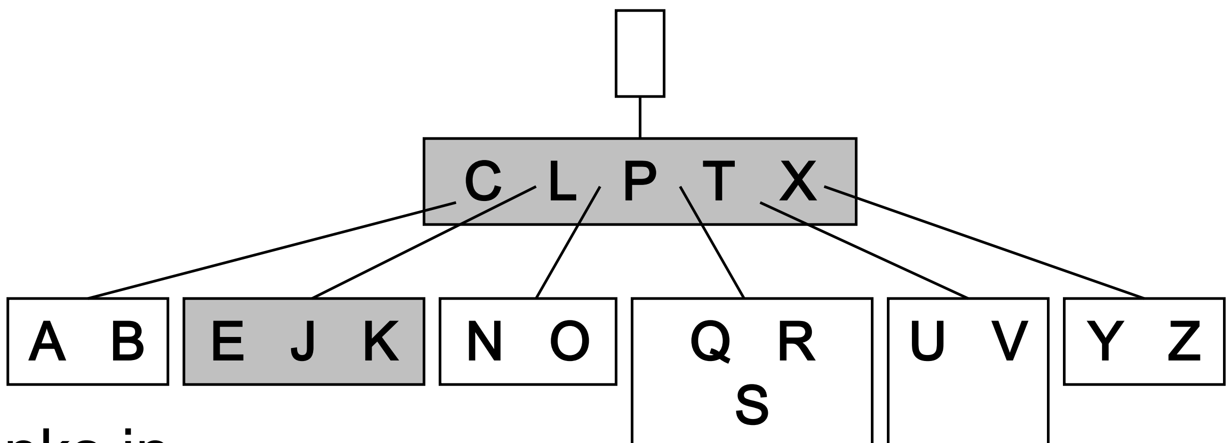


Deleting Keys – Merging (2)

delete **D**



D deleted:



tree shrinks in height

Deletion: Running Time

- Most of the keys are in the leaf, thus deletion most often occurs there!
- In this case deletion happens in one downward pass to the leaf level of the tree
- **Case 2:** Deletion from an internal node might require "backing up"
- **Running time**:
  - Disk I/O: $O(h)$, since only $O(1)$ disk operations are produced during recursive calls
  - CPU: $O(th) = O(t \log_t n)$

Two-pass Operations

- Simpler, practical versions of algorithms use two passes (down and up the tree):
  - *Down* – Find the node where deletion or insertion should occur
  - *Up* – If needed, split, merge, or distribute; propagate splits, merges, or distributes up the tree
- To avoid reading the same nodes twice, use a buffer of nodes

B-Tree / B+Tree animations

- B-Tree
  - http://slady.net/java/bt/view.php
  - http://www.youtube.com/watch?v=coRJrcIYbF4
  - http://ats.oka.nu/b-tree.en.html
  - http://www.cs.auckland.ac.nz/software/AlgAnim/n_ary_trees.html
- B+tree
  - http://www.seanster.com/BplusTree/BplusTree.html

Reading

- AHU, chapter 11
- CLR, chapter 19, CLRS chapter 18
- Notes