B Trees

Prof. Zheng Zhang

Harbin Institute of Technology, Shenzhen

OUTLINE

- *Introduction*
- Definition
- Basic Operations
- Applications

INTRODUCTION

- Motivation
- Overview
- Original Publication
- B-tree …

Introduction--Motivation

- When data is too large to fit in main memory, then the number of disk accesses becomes important.
- Disk access is unbelievably expensive compared to a typical computer instruction (mechanical limitations).
- One disk access is worth about 200,000 instructions.
- The number of disk accesses will dominate the running time.

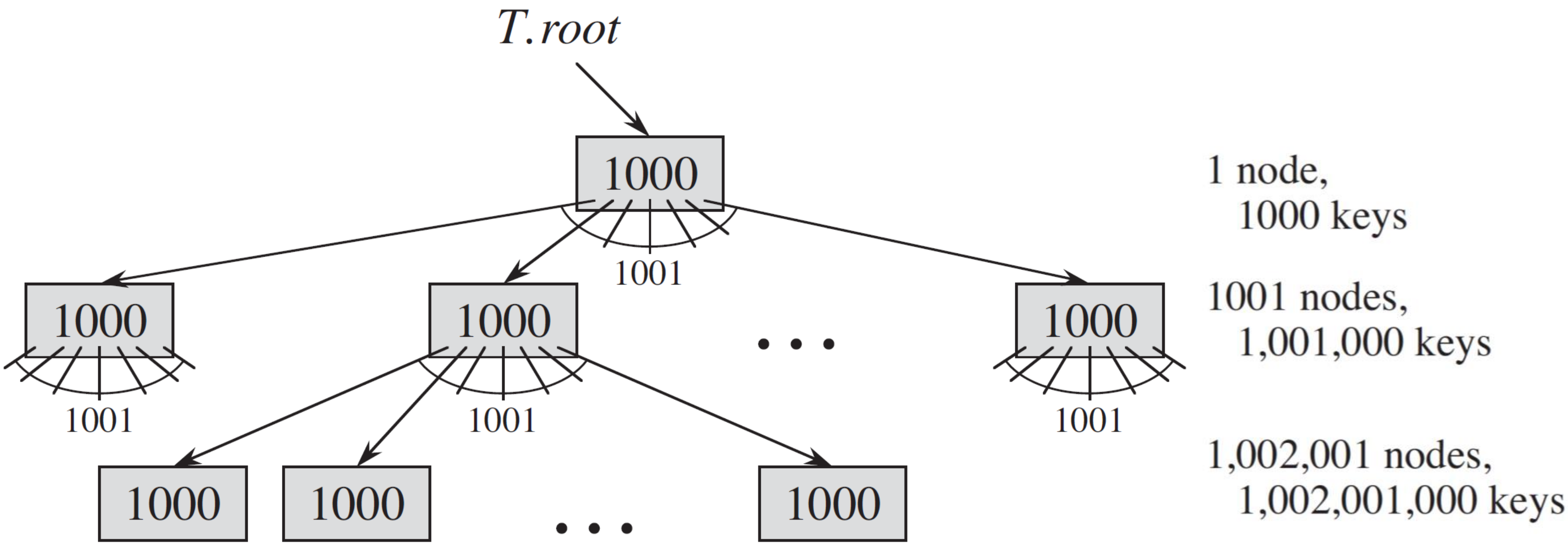
Introduction--Motivation

- Secondary memory (disk) is divided into equal-sized blocks (typical sizes are 512, 2048, 4096 or 8192 bytes).
- Basic I/O operation transfers the contents of one disk block to/from main memory.
- Goal is to devise a multiway search tree that will minimize file accesses (by exploiting disk block read).

Overview

- Have many children, from a handful to thousands.
- The **"branching factor"** of a B-tree can be quite large, although it is usually determined by characteristics of the disk unit used.
- B-trees are *similar* to red-black trees in that every n-node B-tree has height $O(\lg n)$.
- Although the height of a B-tree can be considerably less than that of a red-black tree because its branching factor can be much larger.

EXAMPLE



A B-tree of height 2 containing over one billion keys.

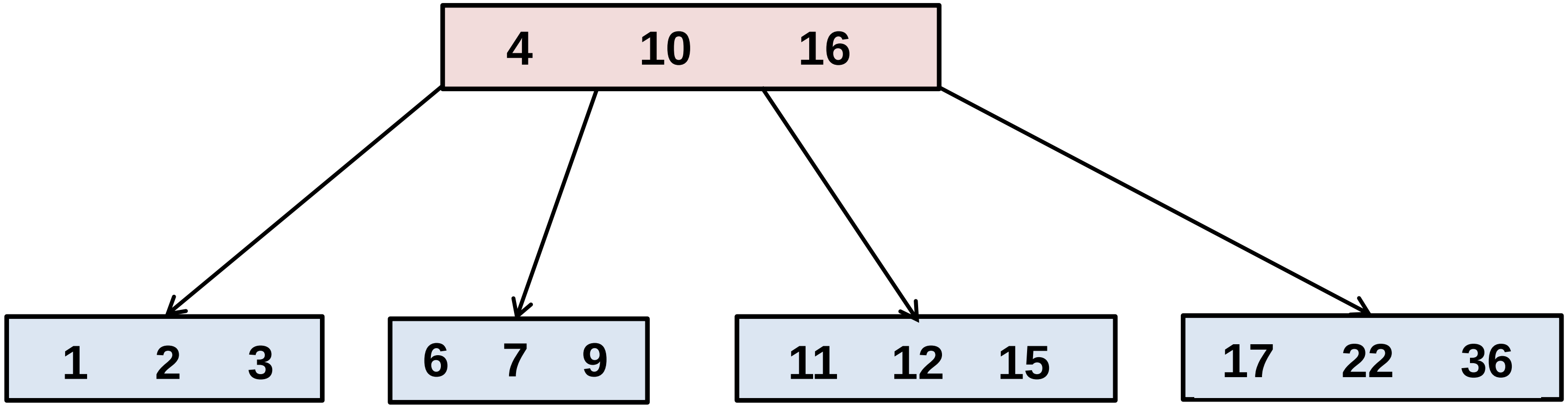
EXAMPLE

- For a large B-tree stored on a disk, branching factors between 50 and 2000 are *often* used, depending on the size of a key relative to the size of a page.
- A large branching factor *dramatically reduces* both the *height* of the tree and the *number* of disk accesses required to **find any key**.
- This B-tree with a *branching factor of 1001* and height 2 that can store over one **billion** keys.
- Nevertheless, since the ***root*** node can be kept permanently in main memory, *only two disk accesses* at most are required to find any key in this tree!

Overview--Structures

- If an **internal** B-tree node *x* contains *n*[*x*] keys, then *x* has *n*[*x*]+1 children.
- The <u>keys in node *x*</u> are used as dividing points separating the range of keys handled by *x* into *n*[*x*]+1 subranges, each handled by one child of *x*.
- When searching for a key in a B-tree, we make an (*n*[*x*]+1)-way decision based on comparisons with the *n*[*x*] keys stored at node *x*.
- The *structure of leaf nodes* differs from that of internal nodes; we will examine these differences later.

Example



OVERVIEW--HOW TO WORK

- In a typical B-tree application, the amount of data handled is so large that all the data do not fit into main memory at once.
- The B-tree algorithms copy selected pages from disk into main memory as **needed** and write back onto disk the pages that have **changed**.
- B-tree algorithms are designed so that only a constant number of pages are in main memory at any time.
- Thus, the size of main memory does not limit the size of B-trees that can be handled.

Overview--The Original Publication

- Rudolf Bayer, Edward M. McCreight
- Organization and Maintenance of Large Ordered Indices
- 1972

OVERVIEW--B-TREE…

- **Also known as** balanced multiway tree
- **Generalization** (I am a kind of ...)

  balanced tree, *search tree*.
- **Specialization** (... is a kind of me.)

  2-3-4 tree, *B*-tree*, *2-3* tree . . .

OVERVIEW--B-TREE…

- The B-tree's creators, Rudolf Bayer and Ed McCreight, have not explained what, if anything, the **B** stands for.
- The most common belief is that **B stands for** *balanced*, as all the leaf nodes are at the same level in the tree.
- B may also stand for *Bayer*, or for *Boeing*, because they were working for Boeing Scientific Research Labs at the time.

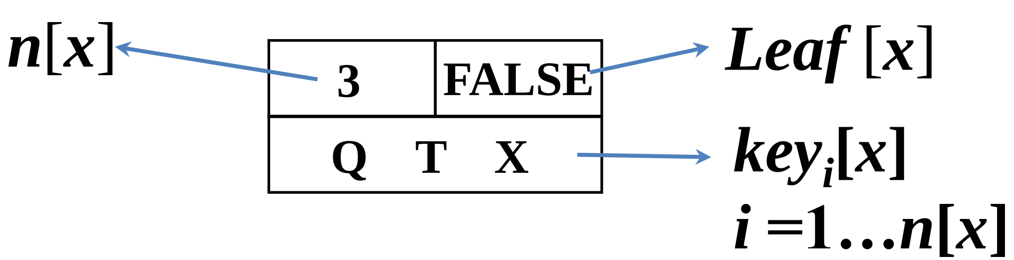
OUTLINE

- Introduction
- *Definition*
- Basic Operations
- Applications

DEFINITION

A ***B-tree T*** is a rooted tree having the following properties:

- **1. Every node** *x* has the following fields
  - $n[x]$, the number of keys currently stored in node x.
  - The $n[x]$ keys themselves stored in **nondecreasing** order, so that $key_1[x] \le key_2[x] \le \ldots \le key_{n[x]}[x]$.
  - Leaf[x], a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.

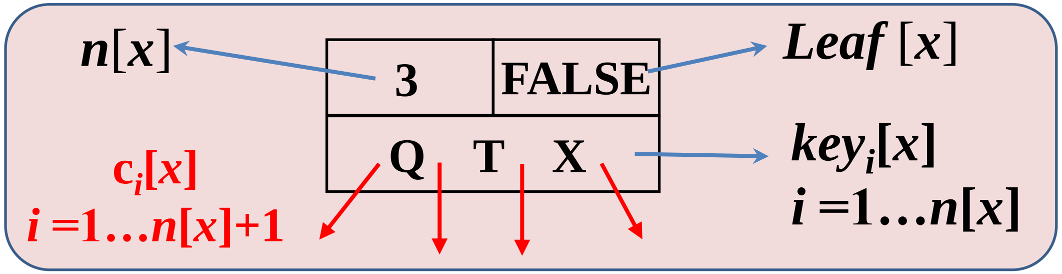


DEFINITION

A ***B-tree T*** is a rooted tree having the following properties:

- **2.** Each internal node x also contains $n[x]+1$ pointers $c_1[x]$, $c_2[x]$, … , $c_{n[x]+1}[x]$ to its children. Leaf nodes have no children, so their c_i fields are undefined.
- **3.** The keys $key_i[x]$ separate the ranges of keys stored in each subtree: if k_i is *any key* stored in the subtree with the **root** $c_i[x]$, then

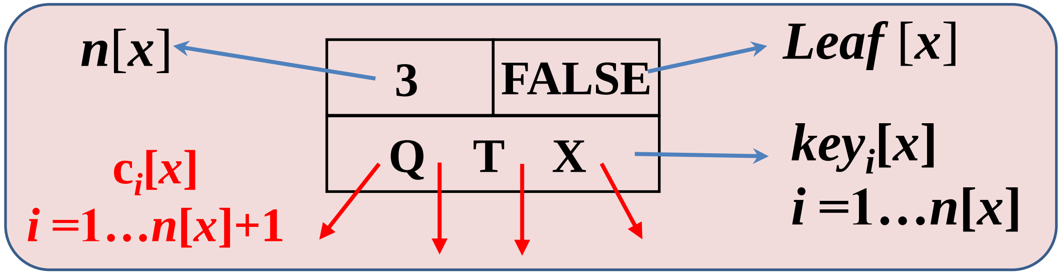
$$k_1 \le key_1[x] \le k_2 \le key_2[x] \le \ldots \le key_{n[x]}[x] \le k_{n[x]+1}$$



DEFINITION

A ***B-tree T*** is a rooted tree having the following properties:

- **4.** All leaves have <u>the same depth</u>, which is the tree's height h.
- **5.** There are <u>lower</u> and <u>upper</u> bounds on the number of keys that a node can contain. These bounds can be expressed in terms of a fixed integer $t \geq 2$ called ***a minimum degree*** of the B-tree:

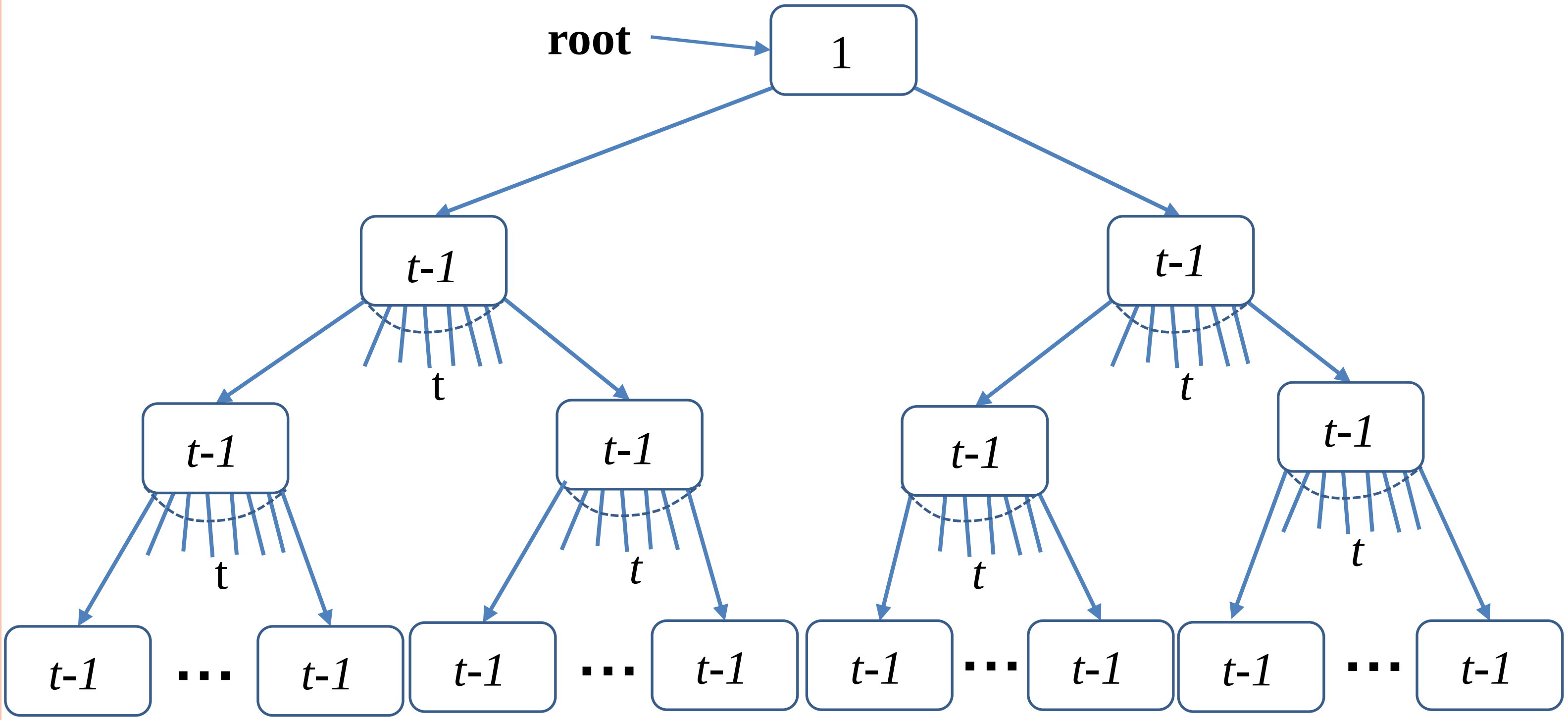


DEFINITION

A ***B-tree T*** is a rooted tree having the following properties:

- **5.** There are lower and upper bounds on the number of keys that a node can contain. These bounds: a minimum degree of the B-tree:
  - Every node other than the root must have at least t-1 keys. Every internal node other than the root thus has at least t *children*. If the tree is **nonempty**, the root must have at least one key.
  - Every node can contain at most $2t$-1 keys, therefore, an internal node can have at most *2t children*. we say that a node is ***full*** if it contains exactly $2t$-1 keys.

DEFINITION--B-TREE HEIGHT



Depth	Nodes
0	1
1	2
2	$2t$
3	$2t^2$

DEFINITION--B-TREE HEIGHT

- $h \le \log_t((n+1)/2)$ ($t \ge 2$, page 489-490)
- The worst case height is O(log *n*). Since the "branchiness" of a B-tree can be large compared to many other balanced tree structures, *the base of the logarithm* tends to be large.
- Therefore, the number of nodes visited **during a search** tends to be smaller than the one required by other tree structures.
- Although this does not affect the *asymptotic* worst case height, B-trees tend to have smaller heights than other trees with the same *asymptotic* height.

OUTLINE

- Introduction
- Definition
- *Basic Operations*
- Applications

Two Conventions

- The root of the B-tree is always in main memory, so that we never need to perform a DISK-READ on the root; we do have to perform a DISK-WRITE of the root, 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.

Basic Operations

- *Searching a B-tree*
- Create an empty B-tree
- Splitting a node
- Inserting a key
- Deleting a key

Basic Operations--Search

- The search operation on a B-tree is *analogous to a search on a binary tree*.
- Instead of choosing between a left and a right child as in a binary tree, a B-tree search must make an $(n[x]+1)$-way choice.
- The correct child is chosen by performing *a linear search* of the values in the node.

Basic Operations--Search

B-TREE-SEARCH (x, k)

$i \leftarrow 1$

while $i \leq n[x]$ *and* $k > \text{key}_i[x]$

    do $i \leftarrow i + 1$

if $i \leq n[x]$ *and* $k == \text{key}_i[x]$

    then return (x, i)

elseif leaf$[x]$

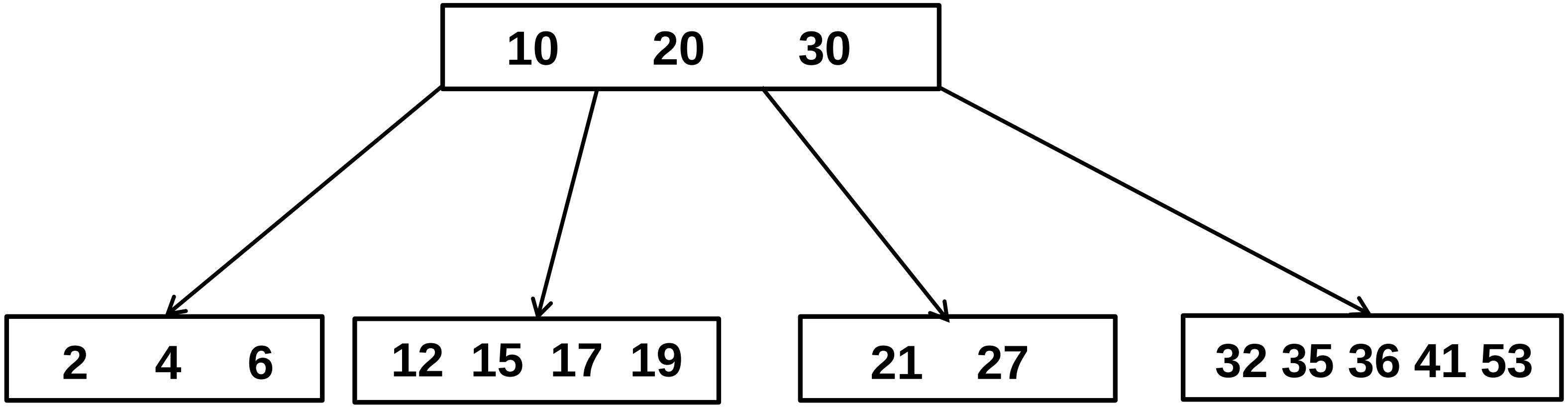
    then return NIL

else Disk-Read$(c_i[x])$

        return B-Tree-Search$(c_i[x], k)$

Basic Operations--Search

B-Tree; Minimum Degree t = 3;
Minimum keys = 2; Maximum keys = 5



Search(21)

Basic Operations—Search Analysis

- After finding the value greater than or equal to the desired value, the child pointer to the immediate *left* of that value is followed. If all values are less than the desired value, the *rightmost* child pointer is followed.
- Of course, the search can be terminated as soon as the desired node is found.
- $O(\log_t n)$ disk operation
- $O(t \log_t n)$ CPU time

BASIC OPERATIONS

- Searching a B-tree
- *Create an empty B-tree*
- Splitting a node
- Inserting a key
- Deleting a key

BASIC OPERATIONS--CREATE

- The *B-Tree-Create* operation creates an empty B-tree by allocating a new root node that has *no* keys and is a *leaf* node.
- Only the ***root*** node is permitted to have these properties; all other nodes must meet the criteria outlined previously.
- $O(1)$ disk operation
- $O(1)$ CPU time

B-Tree-Create(T)

1 $x \leftarrow$ Allocate-Node()
2 leaf[x] $\leftarrow$ TRUE
3 $n[x] \leftarrow 0$
4 Disk-Write(x)
5 root[T] $\leftarrow x$

Basic Operations

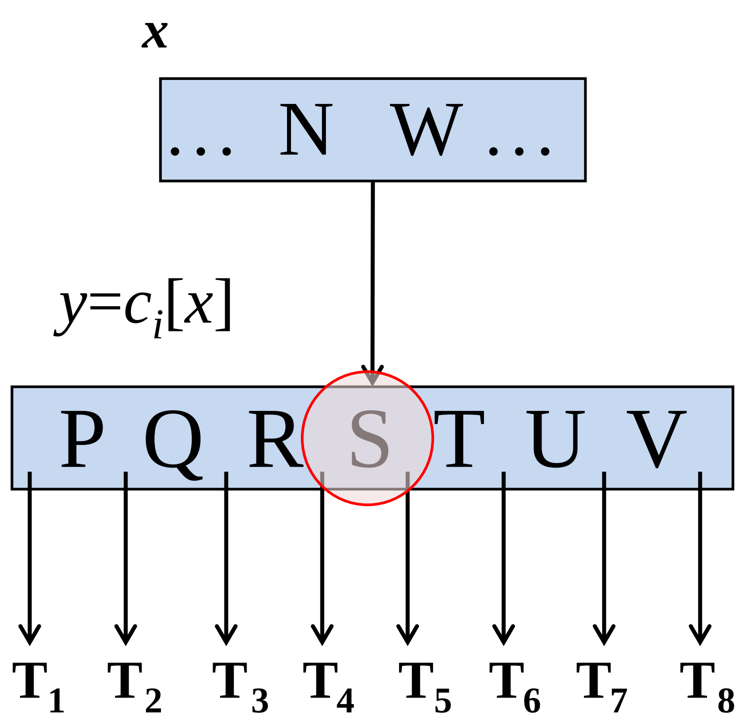
- Searching a B-tree
- Create an empty B-tree
- *Splitting a node*
- Inserting a key
- Deleting a key

BASIC OPERATIONS--WHY NEED SPLIT

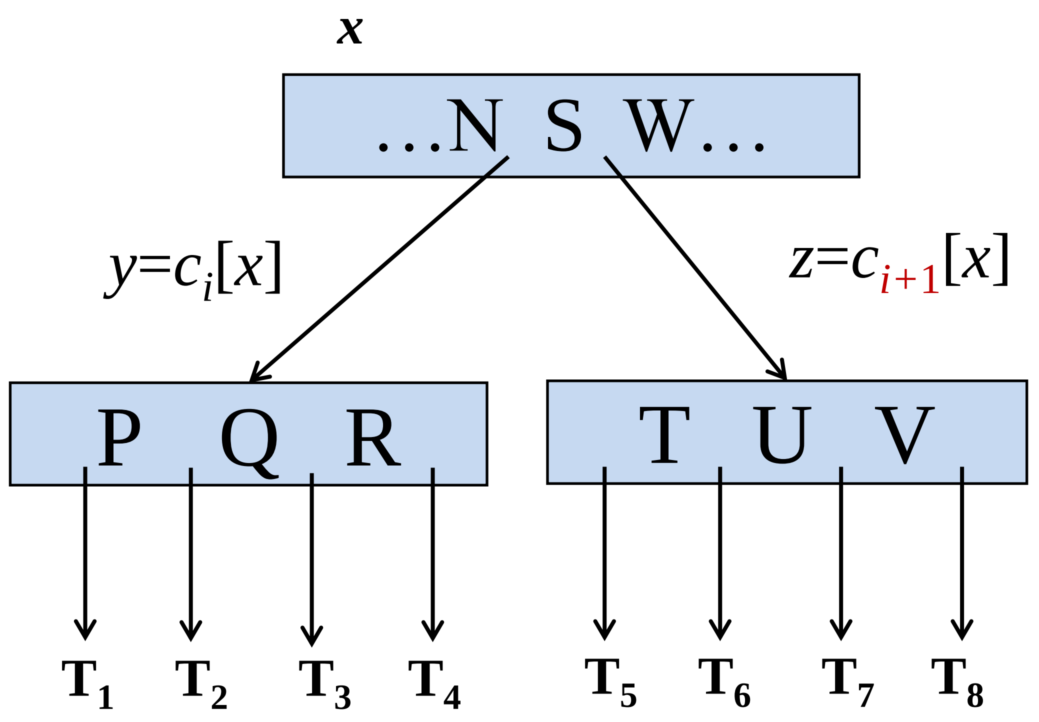
- When inserting a key into a B-tree, we **can't** simply create a new leaf node and insert it, as the result tree would fail to be **a valid B-tree**.
- If is node becomes "too *full*" (having 2*t*-1 keys), it is necessary to perform a split operation.

- It splits a full node *y* (*having 2t-1 keys*) around its median key $key_t[y]$ into two nodes having *t*-1 keys each.
- The median key *moves up* into *y*'s **parent** to identify the dividing point between the two new trees.

BASIC OPERATIONS--SPLITTING



BASIC OPERATIONS--SPLITTING



Basic Operations--Splitting

B-Tree-Split-Child(*x*, *i*, *y*)

z $\leftarrow$ Allocate-Node()

leaf[z] $\leftarrow$ leaf[y]

n[z] $\leftarrow t - 1$

for $j \leftarrow 1$ **to** $t - 1$

    do key_j [z] $\leftarrow$ key_{j+t}[y]

if not leaf[y]

    then for $j \leftarrow 1$ **to** t

        do c_j[z] $\leftarrow$ c_{j+t}[y]

n[y] $\leftarrow t - 1$

for $j \leftarrow n[x] + 1$ **downto** $i+1$

    do $c_{j+1}[x] \leftarrow c_j\,[x]$

$c_{i+1}[x] \leftarrow$ z

for $j \leftarrow n[x]$ **downto** i

    do $key_{j+1}[x] \leftarrow key_j\,[x]$

$key_i\,[x] \leftarrow key_t$ [y]

$n[x] \leftarrow n[x] + 1$

Disk-Write(y)

Disk-Write(z)

Disk-Write(x)

BASIC OPERATIONS--SPLITTING ANALYSIS

- The split operation moves the median key of node *y* into its parent *x* where *y* is the *i*-th child of ***x***.
- A new node, **z**, is allocated, and all keys in *y* right of the median key are moved to **z**. The keys left of the median key remain in the original node ***y***.
- The new node, **z**, becomes the child immediately to the right of the median key that was moved to the parent *x*, and the original node, *y*, becomes the child immediately to the left of the median key that was moved into the parent *x*.

$c_i[x]$

Basic Operations--Splitting Analysis

- The split operation transforms a full node with $2t - 1$ keys into two nodes with $t - 1$ keys each. Note that one key is moved into the <u>parent</u> node.
- $O(1)$ disk operations
- $\Theta(t)$ CPU times

BASIC OPERATIONS

- Searching a B-tree
- Create an empty B-tree
- Splitting a node
- *Inserting a key*
- Deleting a key

Basic Operations--Insertion

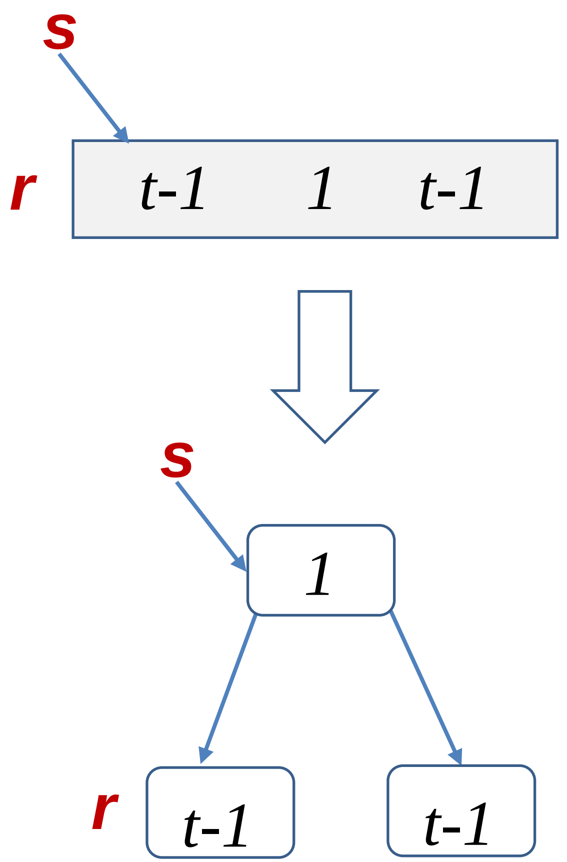
- To perform an insertion on a B-tree, the appropriate node for the key must be located using an algorithm similar to *B-Tree-Search*.
- Next, the key must be inserted into the node.
  - If the node is **not full** prior to the insertion, no special action is required;
  - However, if the node is **full**, the node must be split to make room for the new key.
- Since **splitting** the node results in moving one key to the parent node, **the parent node** must <u>not be full or another split operation</u> is required.
- This process may repeat all the way <u>up to the root</u> and may require splitting the root node.

Basic Operations--Insertion

- This approach requires two passes:
  - The first pass locates the node where the key should be inserted (**Search**);
  - The second pass performs any required splits on the ancestor nodes (**Split**).
- We **don’t wait** to find out <u>whether</u> we will need to split a full node. Instead, as we travel down the tree searching for the new key belongs, we split each <u>full node</u> we come to along the way.
- Thus, whenever we want to split a full node *y*, we are assumed that its <u>parents</u> is <u>not full</u>.

Basic Operations--Insertion

```
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
        c_1 ← r
        B-Tree-Split-Child(s, 1, r)
        B-Tree-Insert-Nonfull(s, k)
  else
        B-Tree-Insert-Nonfull(r, k)
```

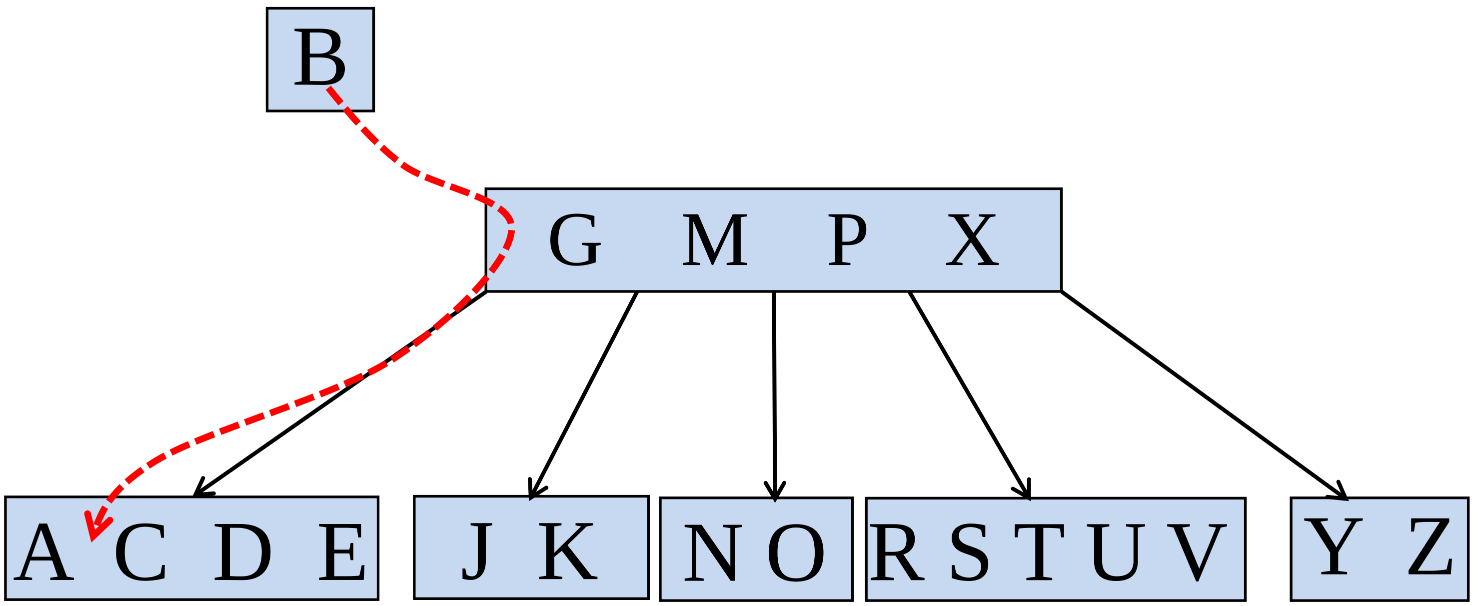


BASIC OPERATIONS--INSERTION

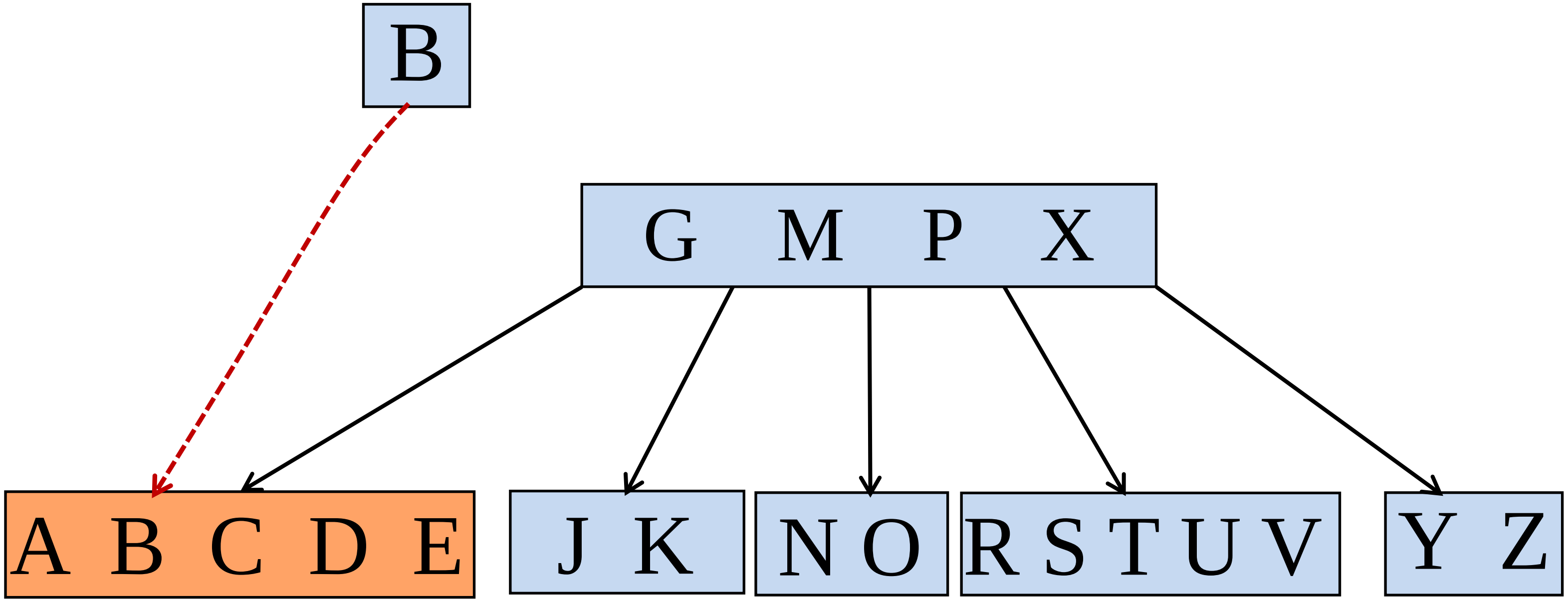
B-Tree-Insert-Nonfull(x, k)

$i \leftarrow n[x]$
if leaf[x]
  then while $i \geq 1$ and $k < key_i[x]$
      do $key_{i+1}[x] \leftarrow key_i[x]$
        $i \leftarrow i - 1$
  $key_{i+1}[x] \leftarrow k$
  $n[x] \leftarrow n[x] + 1$
  Disk-Write(x)
else while $i \geq 1$ and $k < key_i[x]$
      do $i \leftarrow i - 1$
  $i \leftarrow i + 1$
  Disk-Read($c_i[x]$)
  if $n[c_i[x]] == 2t - 1$
    then
      B-Tree-Split-Child(x, i, $c_i[x]$)
      if $k > key_i[x]$
        then $i \leftarrow i + 1$
  B-Tree-Insert-Nonfull($c_i[x]$, k)

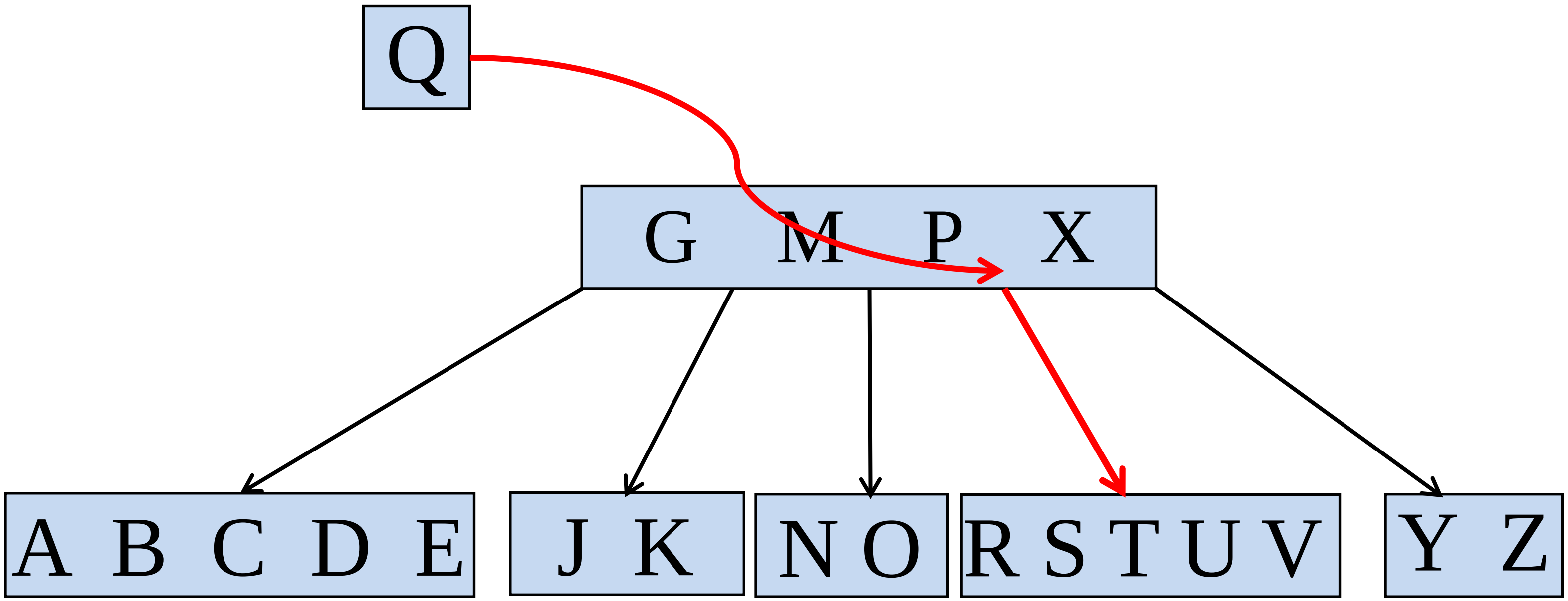
Basic Operations--Insertion



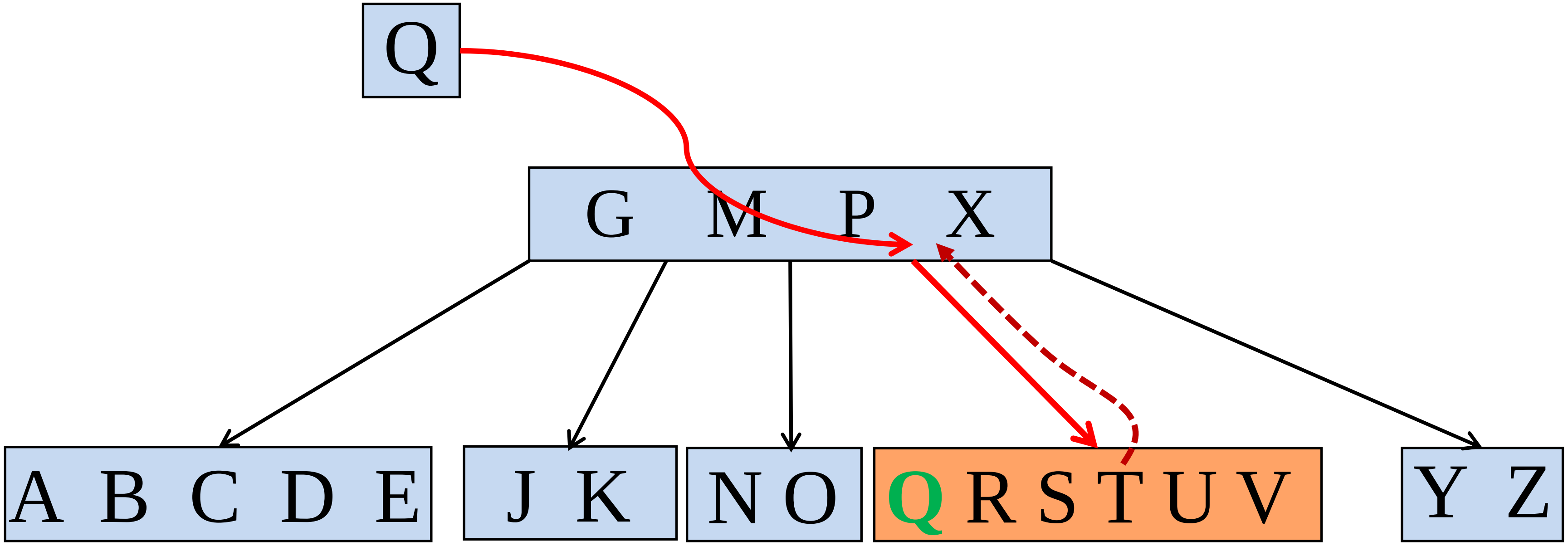
Basic Operations--Insertion



BASIC OPERATIONS--INSERTION



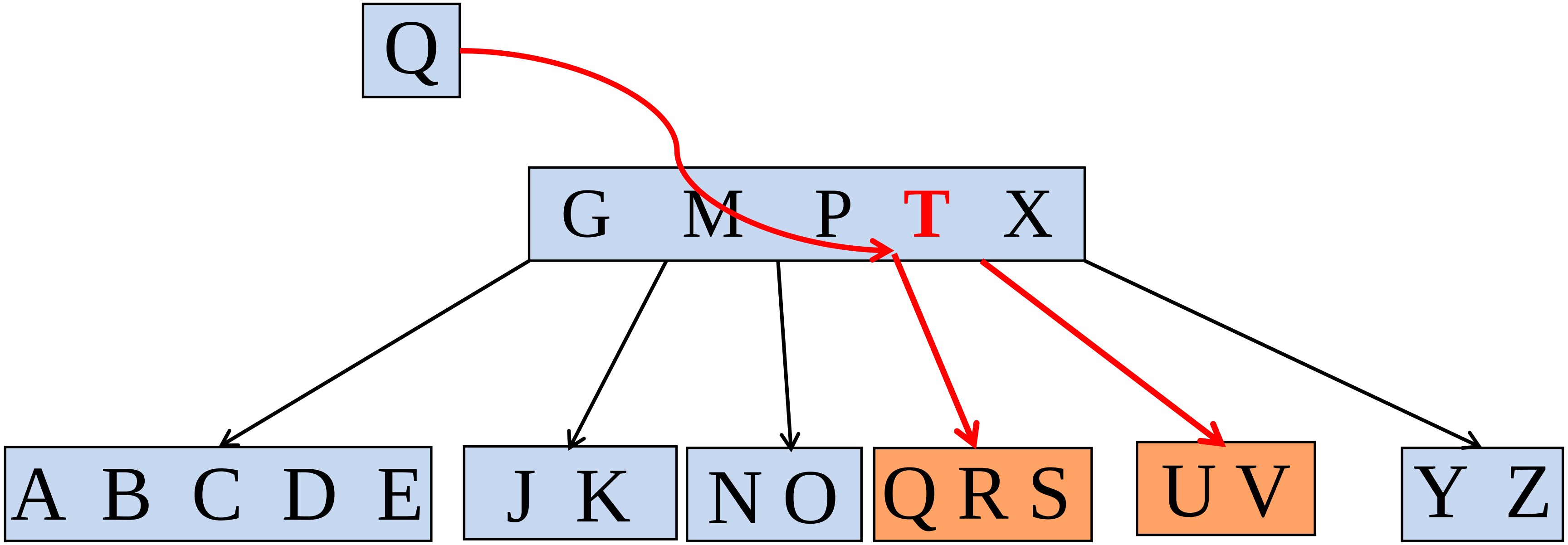
BASIC OPERATIONS--INSERTION



2*t* – 1

Minimum Degree *t* = 3

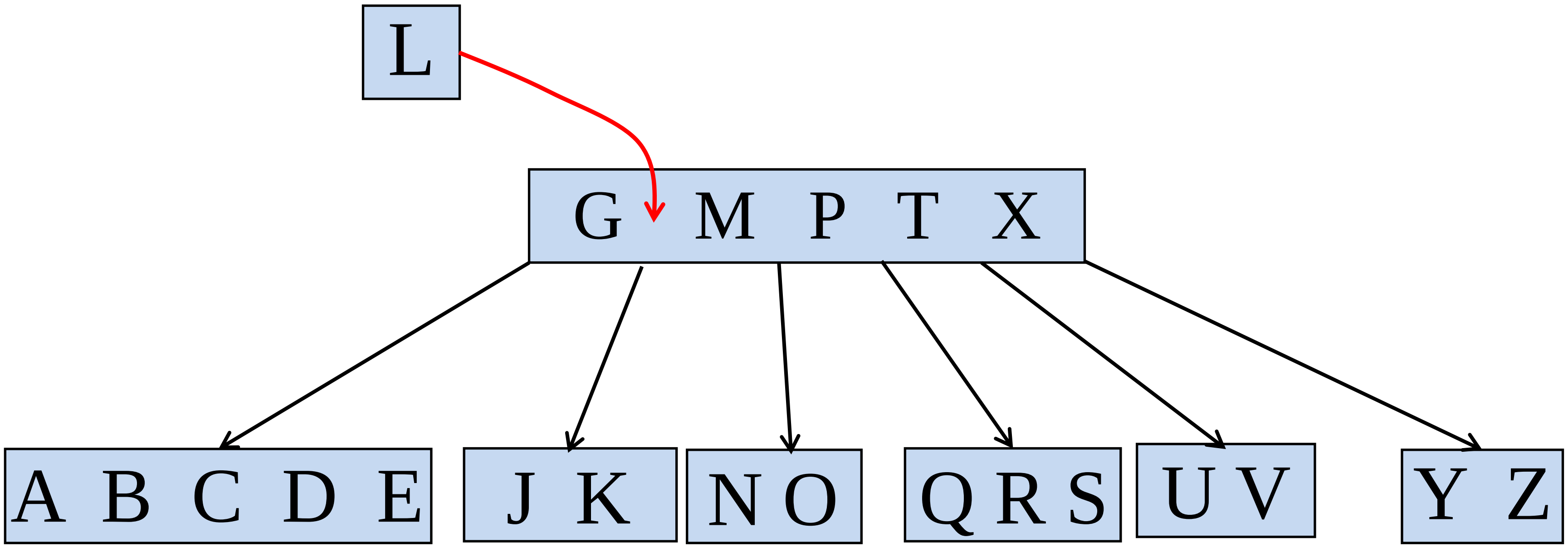
BASIC OPERATIONS--INSERTION



2*t* – 1

Minimum Degree *t* = 3

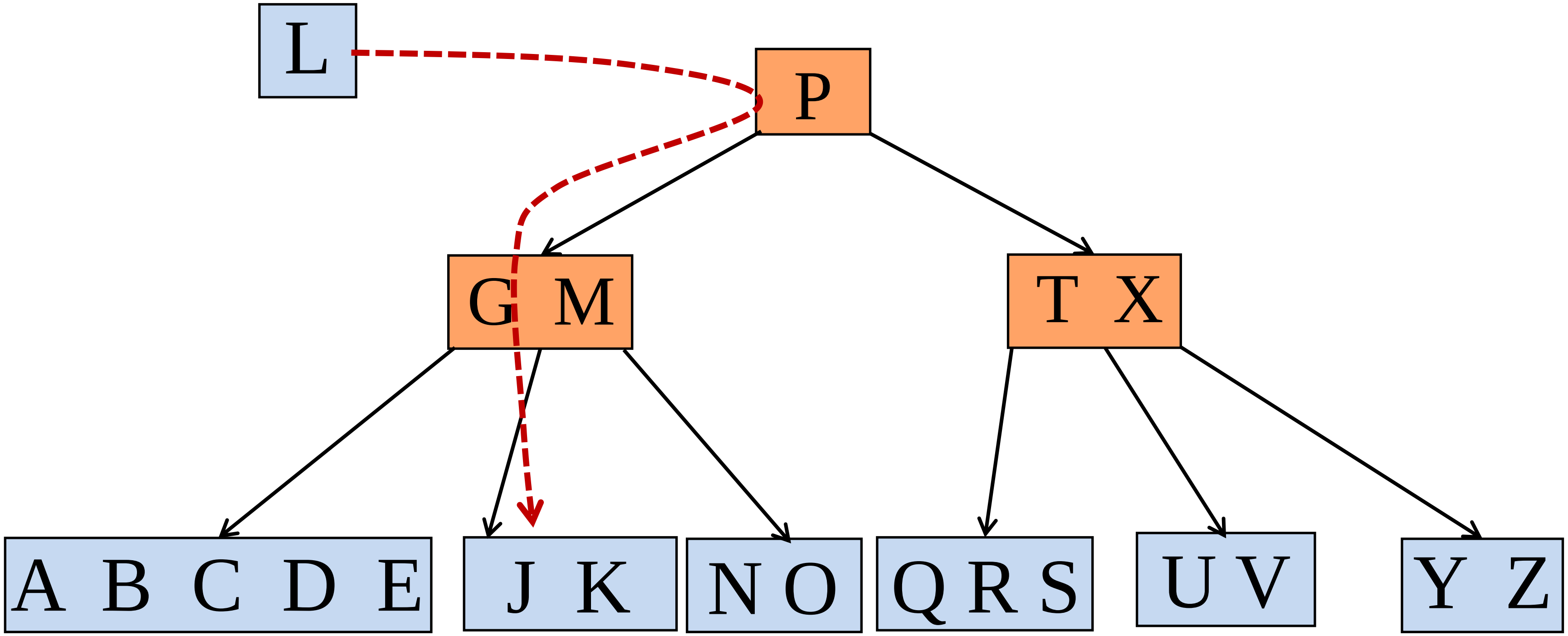
Basic Operations--Insertion



2*t* – 1

Minimum Degree *t* = 3

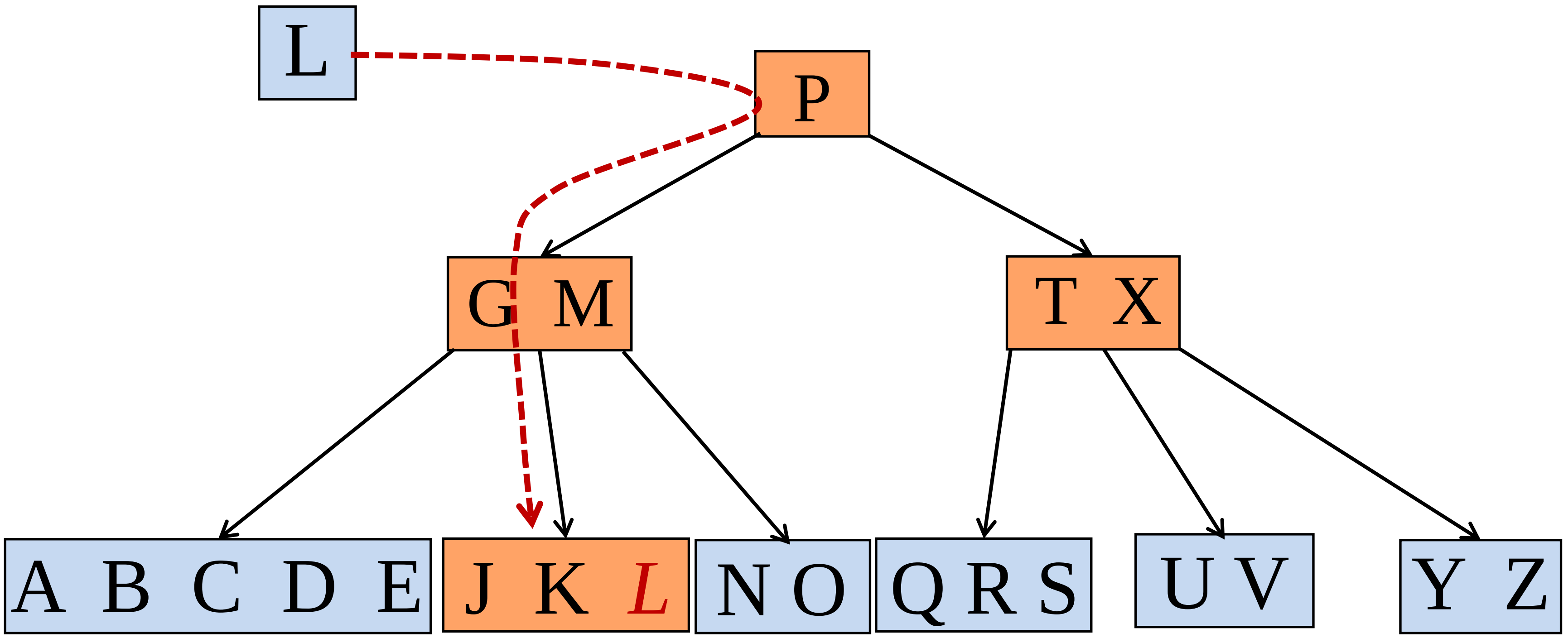
BASIC OPERATIONS--INSERTION



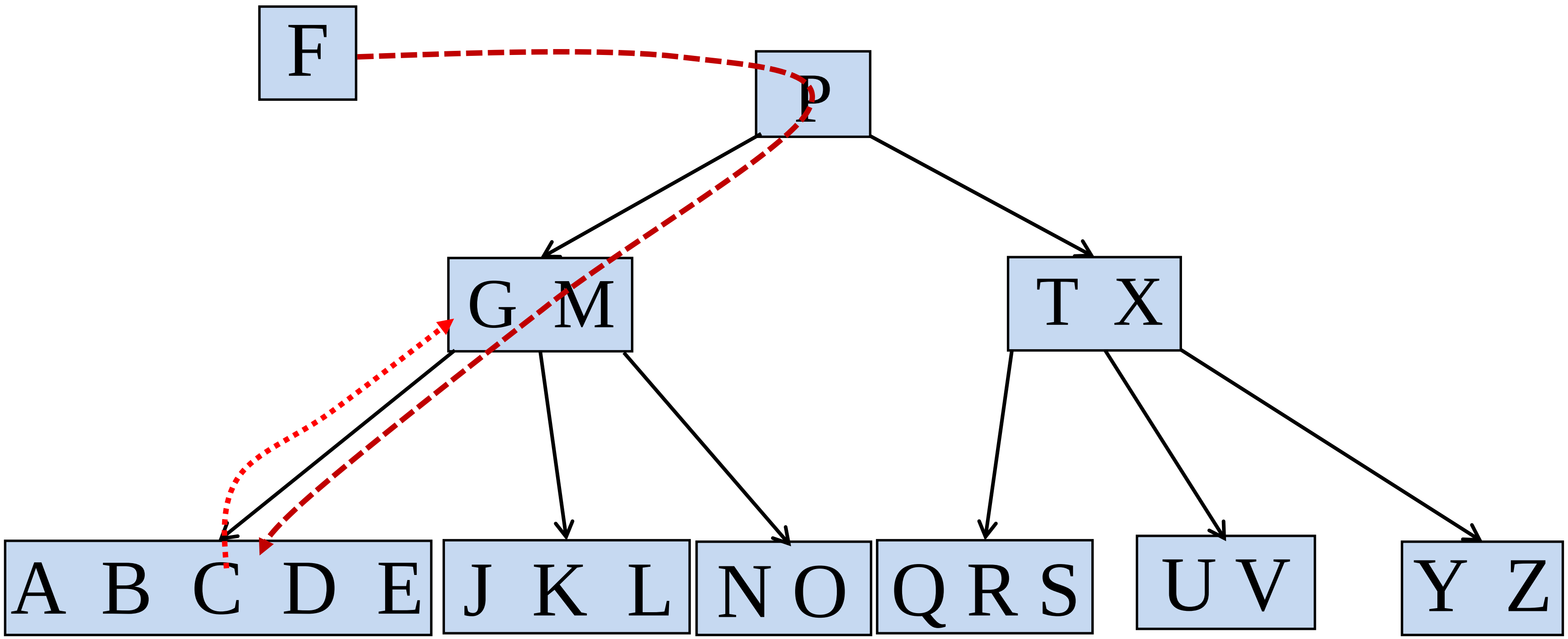
2*t* – 1

Minimum Degree *t* = 3

Basic Operations--Insertion



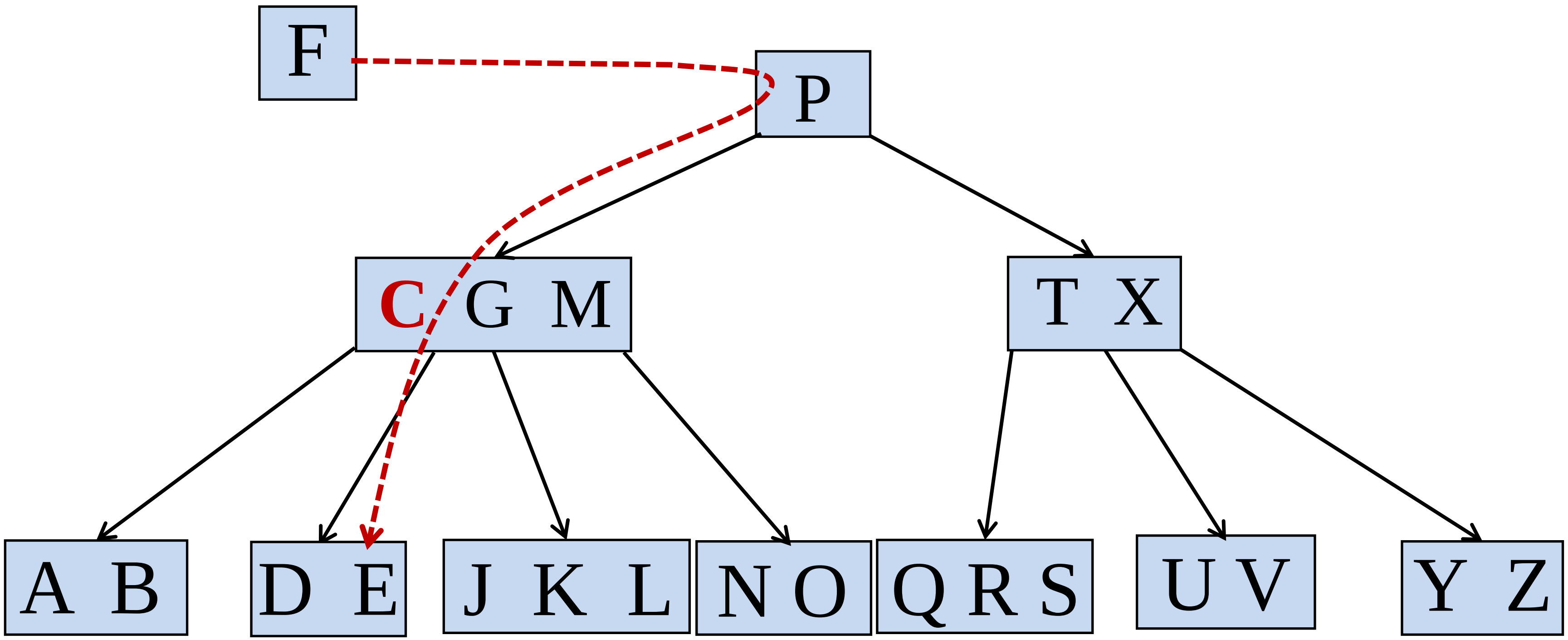
Basic Operations--Insertion



2*t* – 1

Minimum Degree *t* = 3

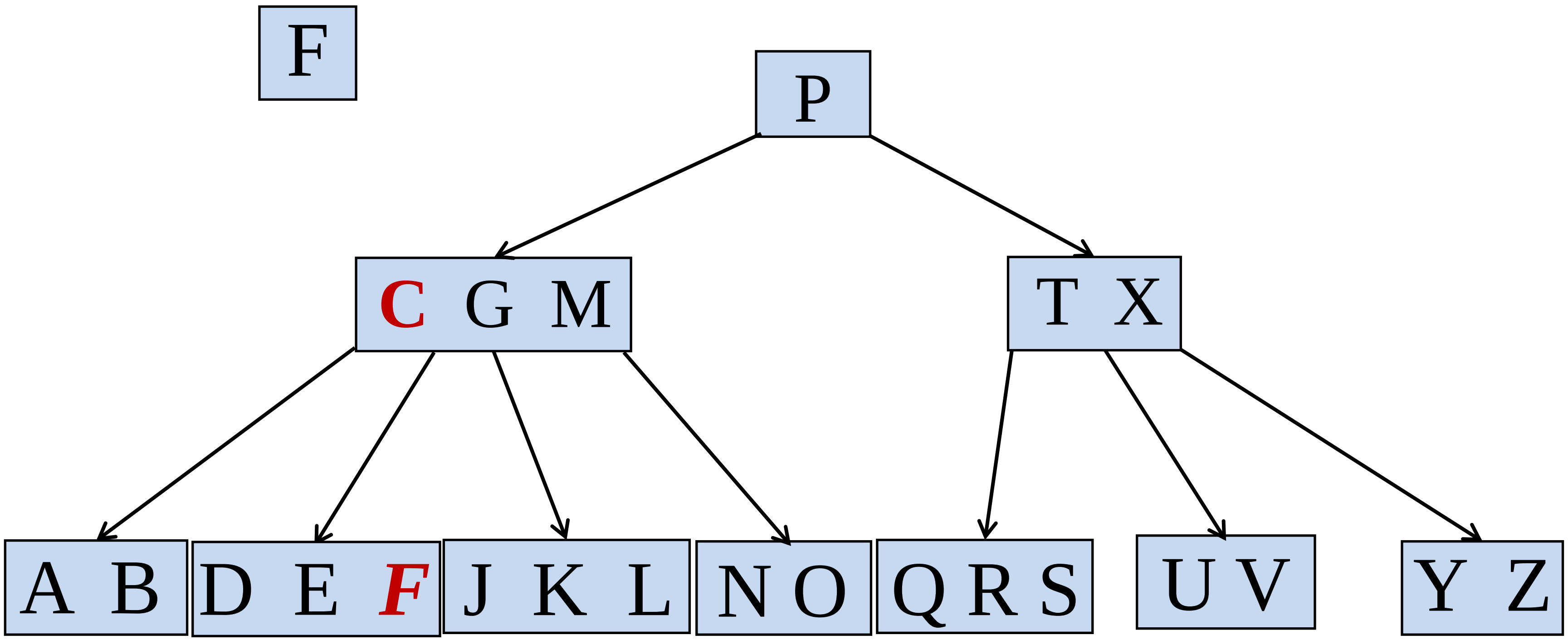
Basic Operations--Insertion



2*t* – 1

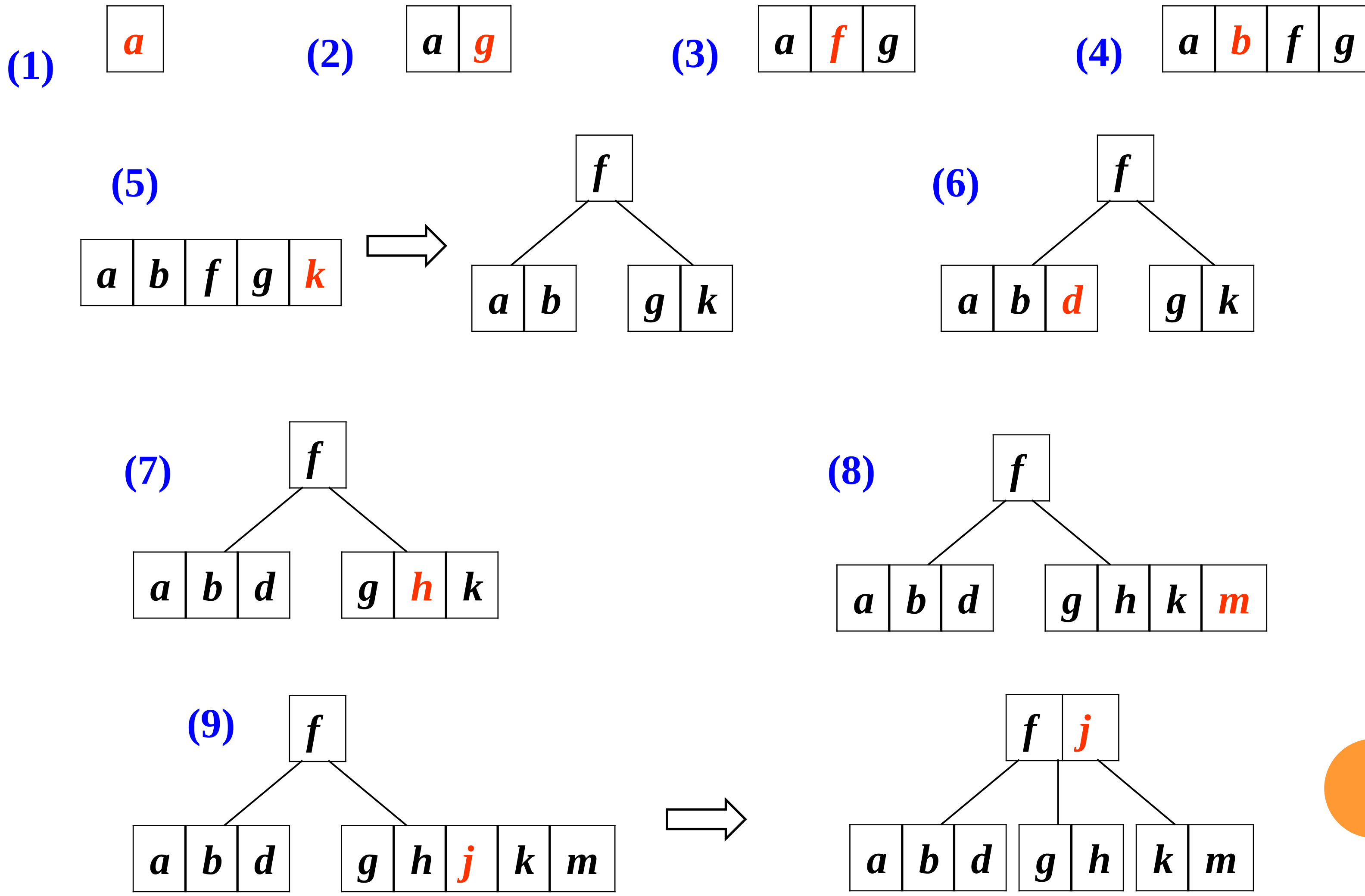
Minimum Degree *t* = 3

Basic Operations--Insertion

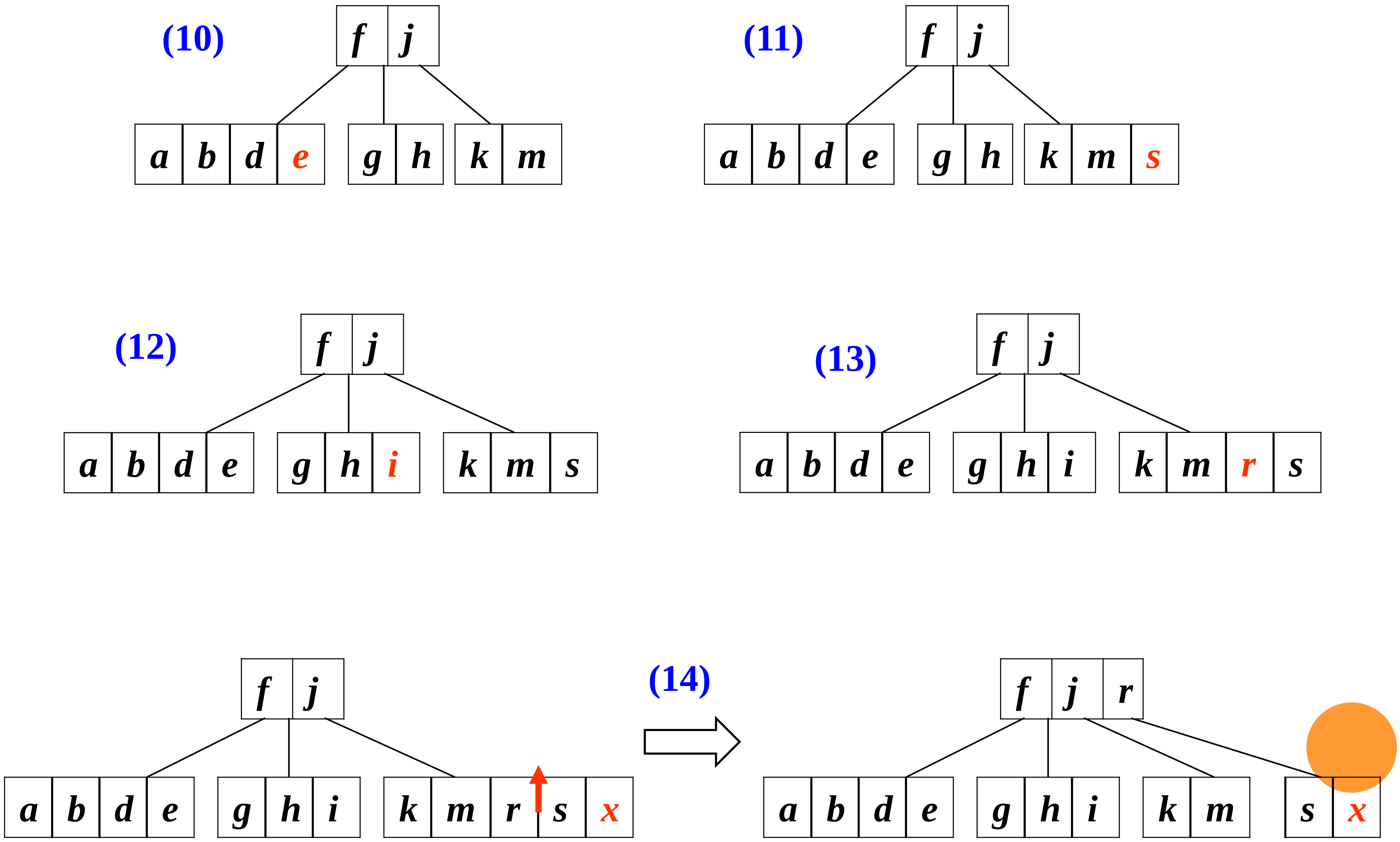


Basic Operations--Insertion

Sequence: (*a, g, f, b, k, d, h, m, j, e, s, i, r, x, c, l, n, t, u, p*) ***t*** **=5**

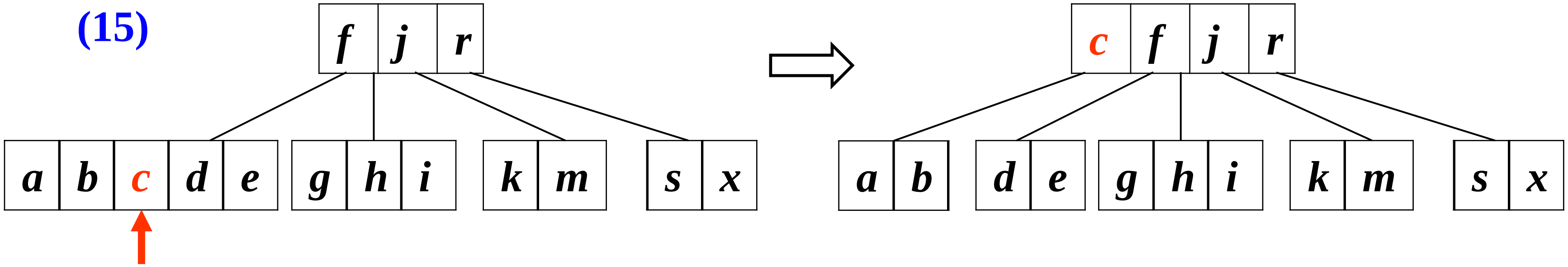


BASIC OPERATIONS--INSERTION

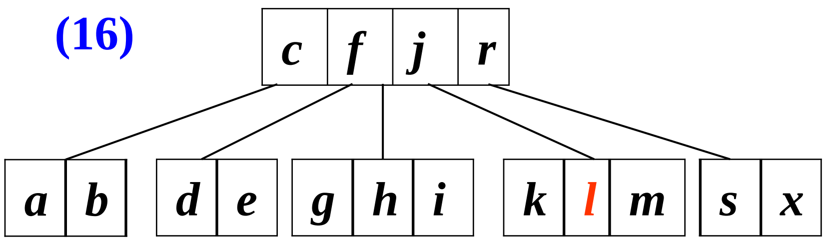


BASIC OPERATIONS--INSERTION

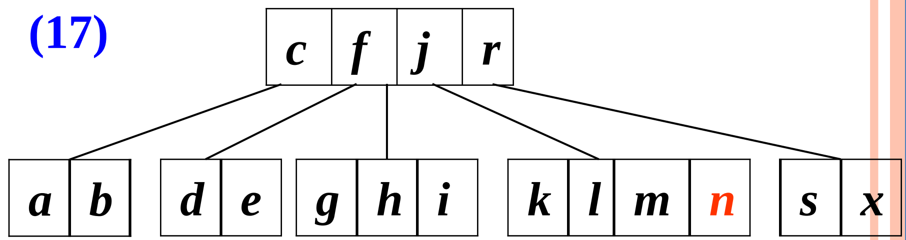
(15)



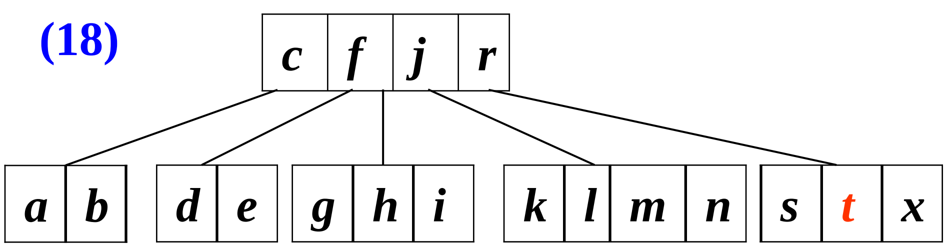
(16)



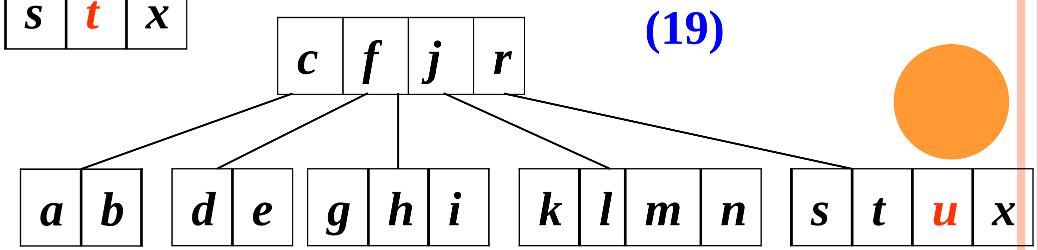
(17)



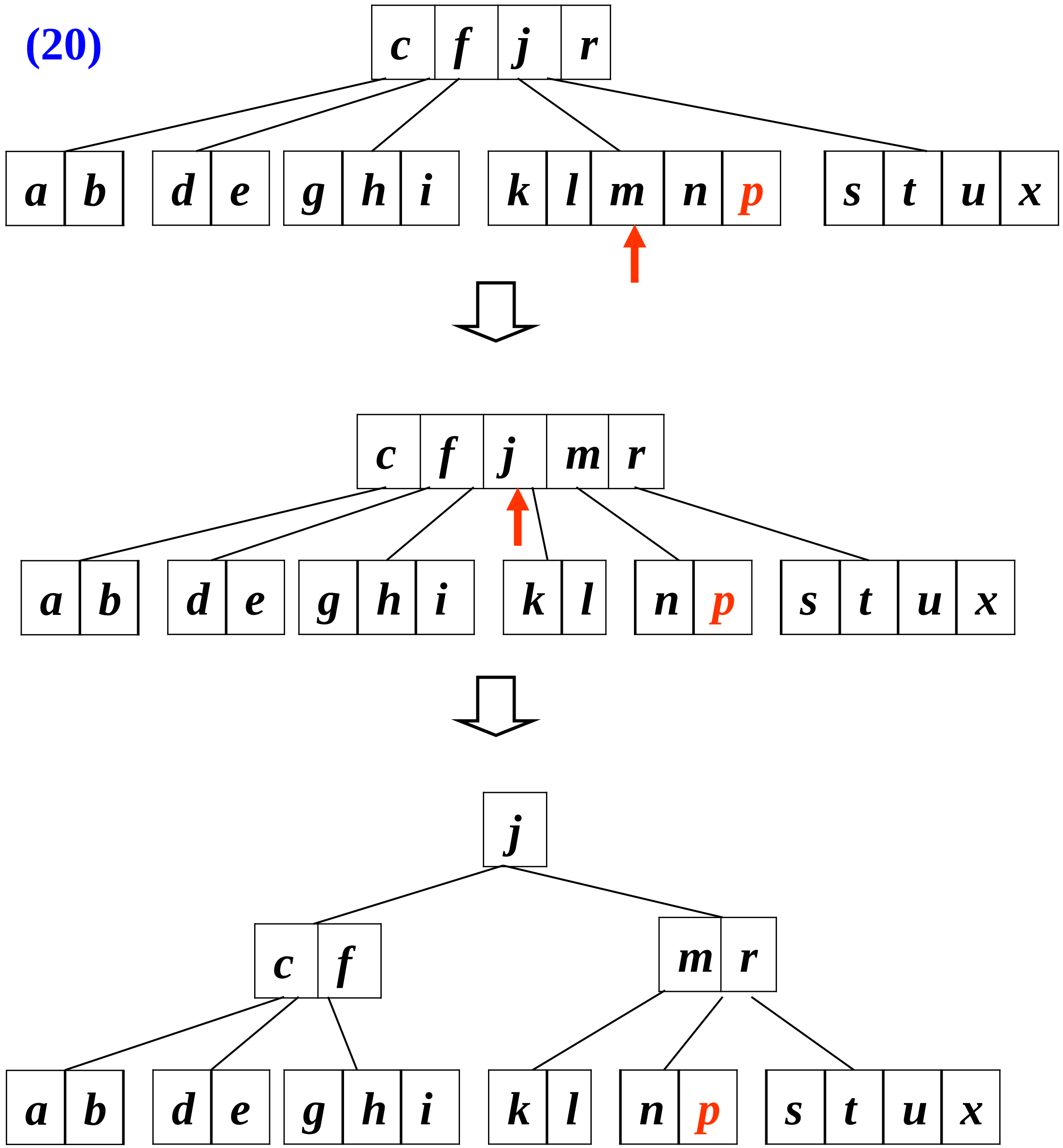
(18)



(19)



BASIC OPERATIONS--INSERTION



BASIC OPERATIONS--INSERTION ANALYSIS

- Since each access to a node may correspond to a costly disk access, it is desirable to avoid the second pass by ensuring that the parent node is never ***full***.
- To accomplish this, the presented algorithm splits any full nodes encountered while descending the tree.
- Although this approach may result in unnecessary split operations, it guarantees that the parent never needs to be split and eliminates the need for a second pass up the tree.
- Since a split runs in linear time, it has little effect on the $\boldsymbol{O(t \log_t n)}$ running time of *B-Tree-Insert*.

Basic Operations

- Searching a B-tree
- Create an empty B-tree
- Splitting a node
- Inserting a key
- *Deleting a key*

Basic Operations--Deletion

- There are *two popular strategies* for deletion from a B-Tree.
  - Locate and <u>delete the item, then restructure</u> the tree to regain its invariants.
  - Do a single pass down the tree, but <u>before entering (visiting) a node, restructure the tree</u> so that once the key to be deleted is encountered, it can be deleted without triggering the need for any further restructuring.

Basic Operations--Deletion

- **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 <u>an internal node</u>, do the following:
  - ***a.*** If the child y that precedes k in node x has <u>at least t keys</u>, 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.)

BASIC OPERATIONS--DELETION

- **2**. If the key *k* is in node *x* and *x* is an internal node, do the following:
  - ***b*. (y has fewer than *t* keys)** 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*.

BASIC OPERATIONS--DELETION

- **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 $c_i[x]$.

BASIC OPERATIONS--DELETION

- **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*.
  - ***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.

Basic Operations--Varieties

- **2-3-4 tree** is a B-tree of order 4
- **2-3 tree** is a B-tree that can contain only 2-nodes (a node with 1 field and 2 children) and 3-nodes (a node with 2 fields and 3 children)
- **B+ tree** (also known as a Quarternary Tree) is a variety of B-tree, in contrast to a B-tree, all records are stored at the lowest level of the tree; only keys are stored in interior blocks.
- **B*-tree** is a tree data structure, a variety of B-tree used in the HFS and Reiser4 file systems, which requires non-root nodes to be at least 2/3 full instead of 1/2.

OUTLINE

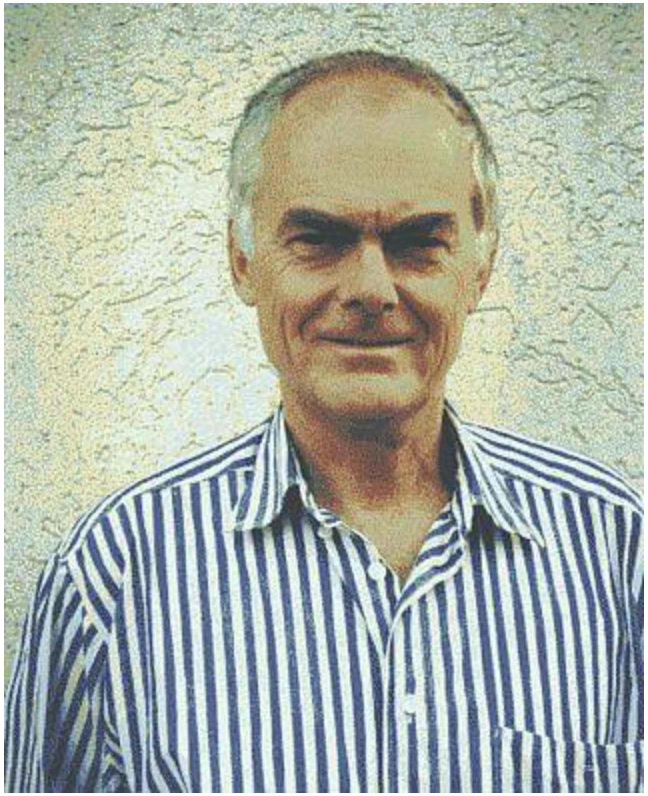
- Introduction
- Definition
- Basic Operations
- *Applications*

APPLICATIONS

- As databases cannot typically be maintained entirely in memory, B-trees are often used to index the data and to provide fast access.
  - For example, searching an unindexed and unsorted database containing n key values will have a worst case running time of O(*n*).
- *Search Engine Index*
- . . . .

Rudolf Bayer

- He is a professor emeritus of Informatics at the Technical University of Munich where he has been employed since 1972.
- He is noted for inventing three data sorting structures: the B-tree (with Edward M. McCreight), the UB-tree (with Volker Markl) and the Red–black tree.
- A recipient of 2001 ACM SIGMOD Edgar F. Codd Innovations Award



Edward M. McCreight

- Co-invented the B-tree, and improved Weiner's algorithm to compute the suffix tree of a string.
- Co-designed the *Xerox Alto workstation*.
- Co-led the design and construction of the Xerox Dorado computer at Xerox Palo Alto Research Center.
- Worked at Adobe Systems.
- Guest professor at the University of Washington, Stanford University, the Technical University of Munich, and the Swiss Federal Institute of Technology in Zürich.