B-Trees

Autumn 2008
Intelligence Computing Research Center
Shenzhen Graduate School, Harbin Institute of Technology

http://219.223.242.33/algorithm/

Outline

- *Introduction*
- Definition
- Basic operations
- Applications

Introduction

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

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.

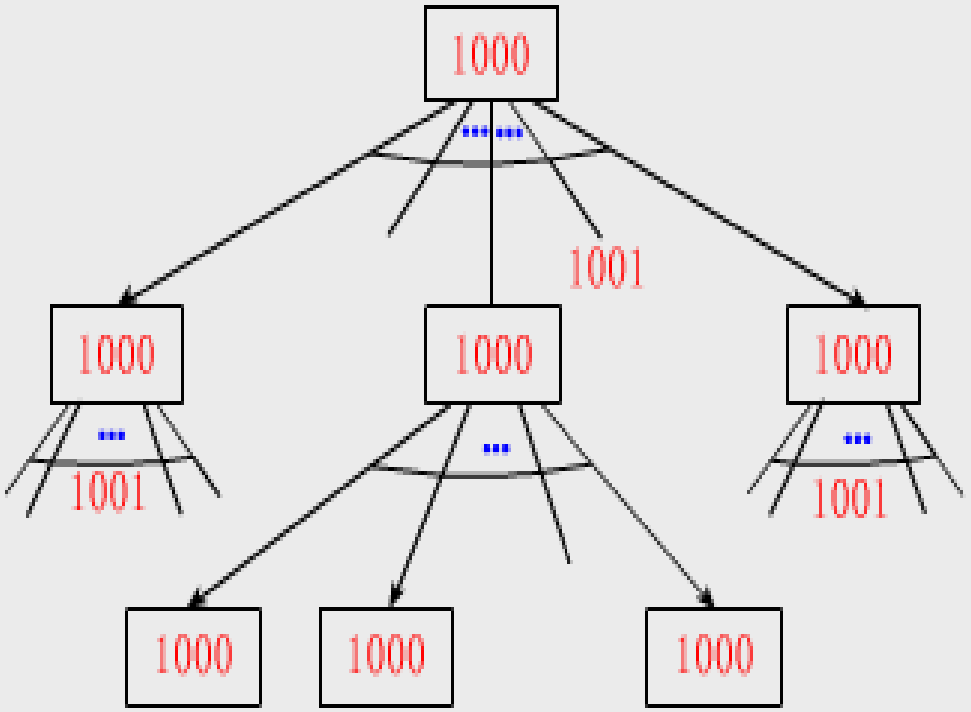
Motivation (cont.)

- 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



第一层：1个结点
1000个关键字

第二层：1001个结点
1,001,000个关键字

第三层：1001个结点
1,002,001,000个关键字

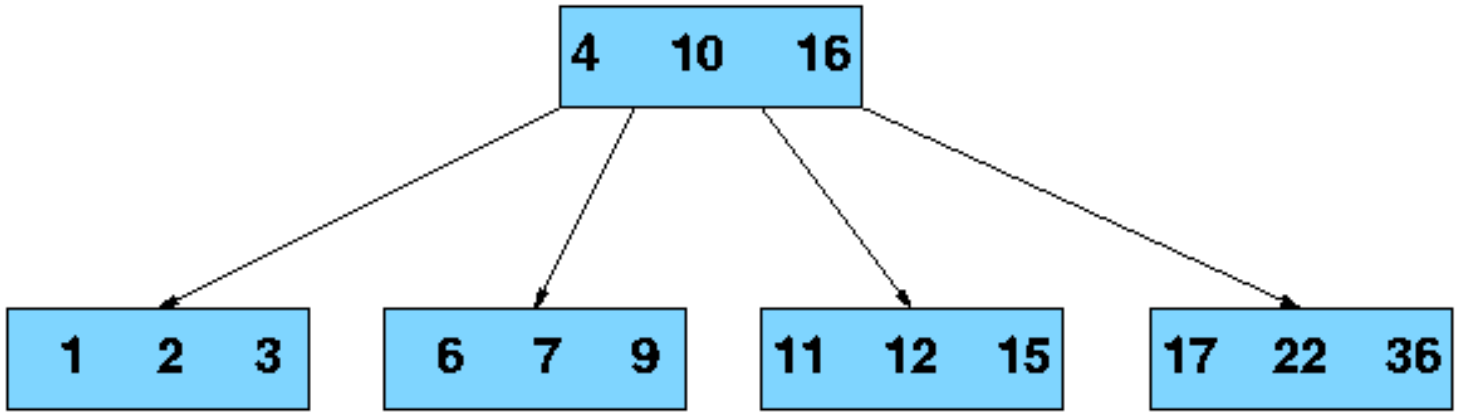
Example (cont.)

- 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 keys in node x 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

B-tree example



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.

The Original Publication

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

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*. .

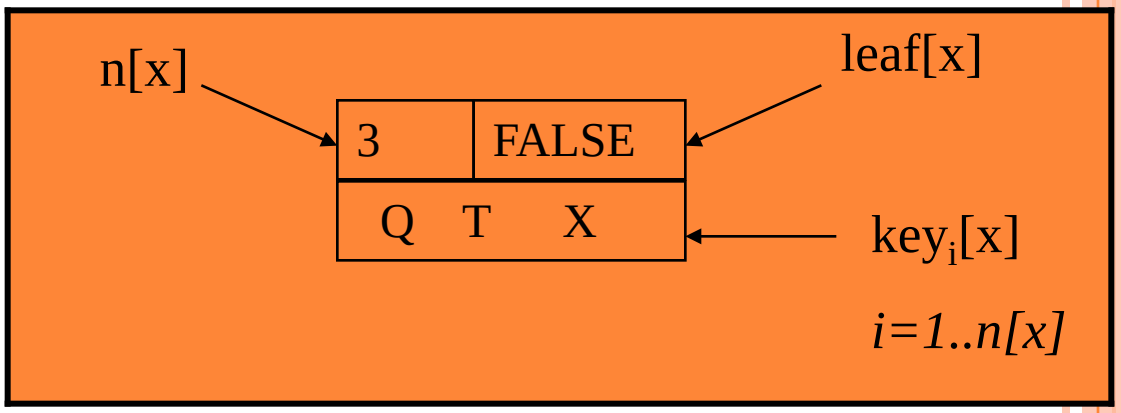
B-tree (cont.)

- 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

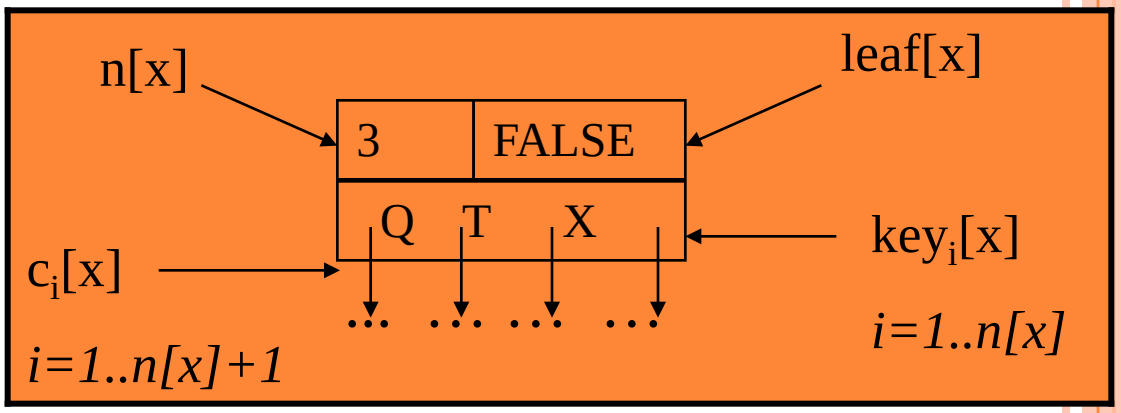
Definition



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 $key_1[x] \leq key_2[x] \leq \ldots \leq key_{n[x]}[x]$
  - Leaf[x],a boolean value that is *TRUE* if x is a leaf and *FALSE* if x is an internal node.

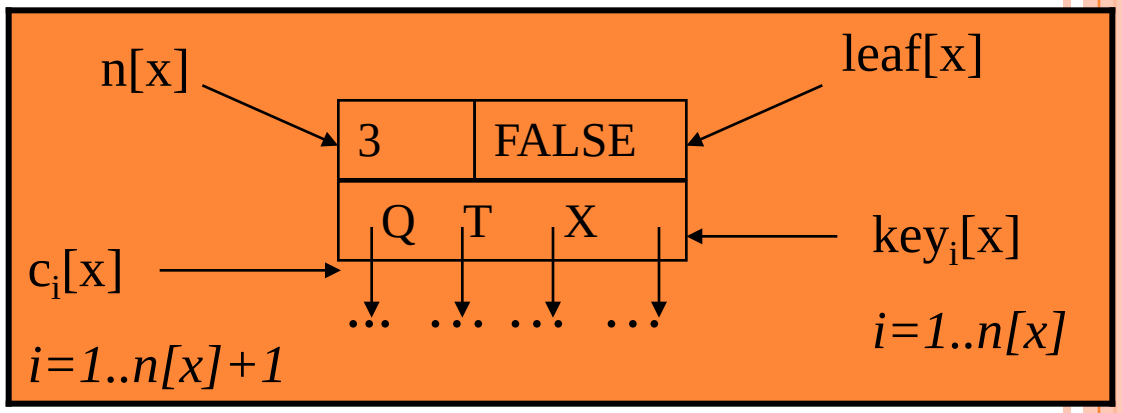
DEFINITION (CONT.)



- 2 Each internal node x also contains n[x]+1 pointers $c_1[x], c_2[x],\ldots,c_{n[x]+1}[x]$ 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 root $c_i[x]$, then

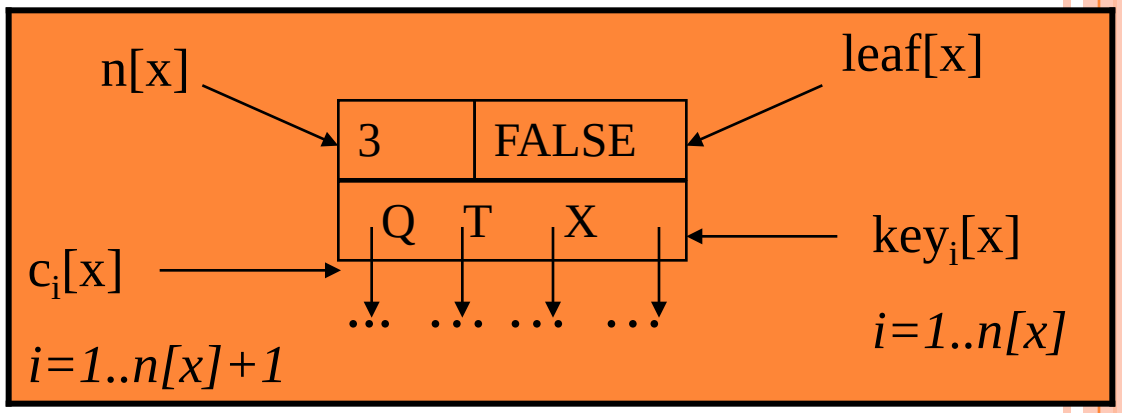
$$k_1 \le key_1[x] \le k_2 \le key_2[x] \le \ldots \le key_{n[x]}[x] \le k_{n[x]+1}$$

Definition (cont.)



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

Definition (cont.)



- 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

B-TREE HEIGHT

- $h \leq \log_t ((n+1)/2)$ (page 439-440)
- 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 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

Basic operations

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

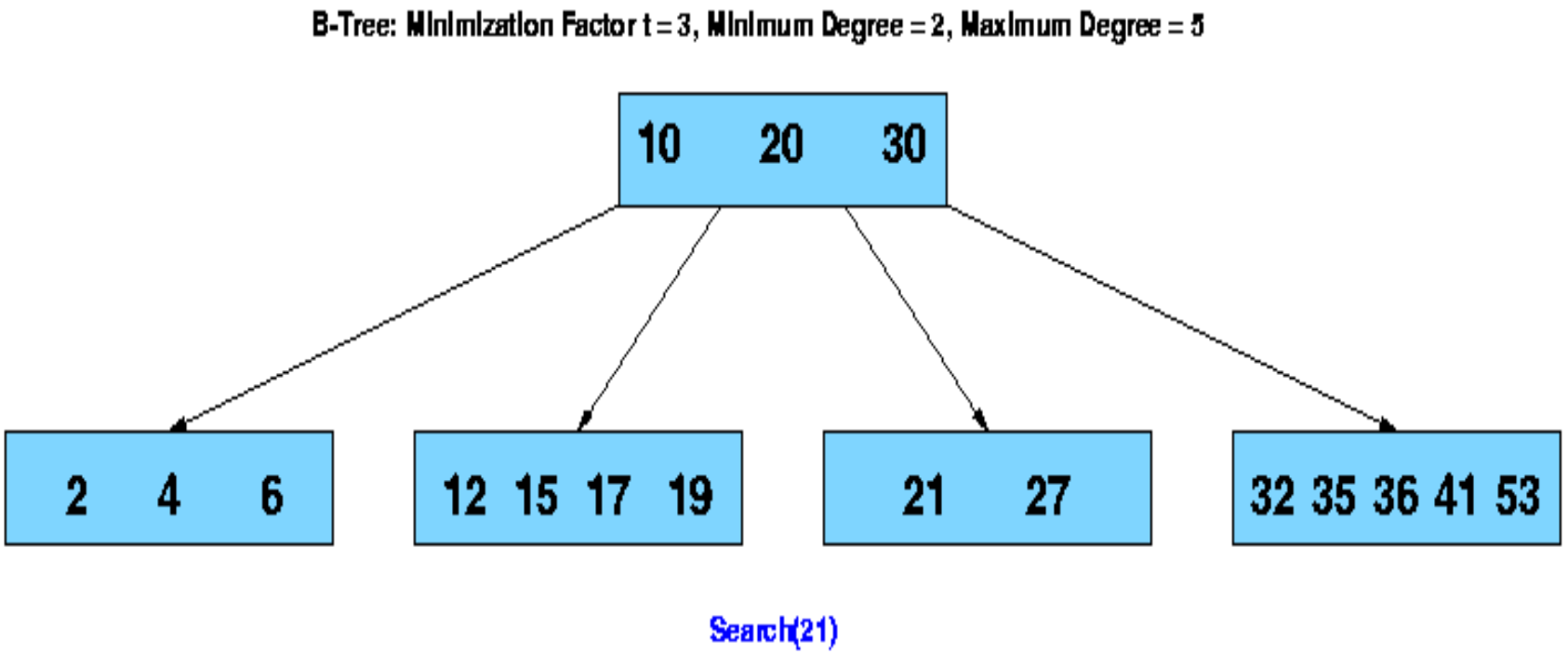
SEARCH-OVERVIEW

- 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.

SEARCH-PSEUDOCODE

```
B-TREE-SEARCH(x,k)
1  i ← 1
2  while i ≤ n[x] and k > key_i[x]
3      do i ← i + 1
4  if i ≤ n[x] and k = key_i[x]
5      then return (x, i)
6  if leaf[x]
7      then return NIL
8      else Disk-Read(c_i[x])
9           return B-Tree-Search(c_i[x], k)
```

SEARCH-EXAMPLE



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.
- $\theta(\log_t n)$ 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

CREATE-PSEUDOCODE

B-Tree-Create(T)

1 x ← Allocate-Node()

2 leaf[x] ← TRUE

3 n[x] ← 0

4 Disk-Write(x)

5 root[T] ← x

CREATE-ANALYSIS

- 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

Basic operations

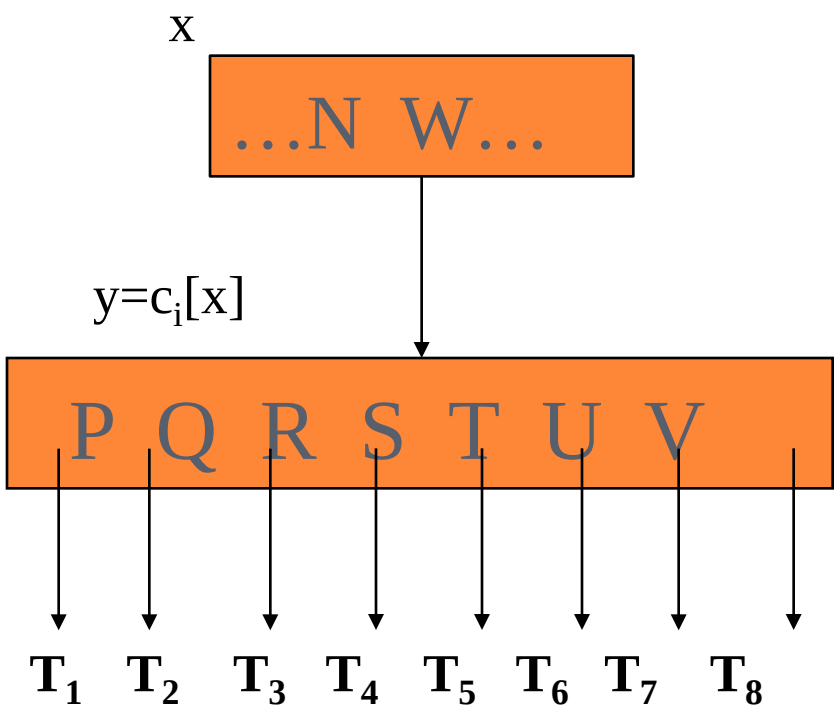
- Searching a B-tree
- Create an empty B-tree
- *Splitting a node*
- Inserting a key
- Deleting a key

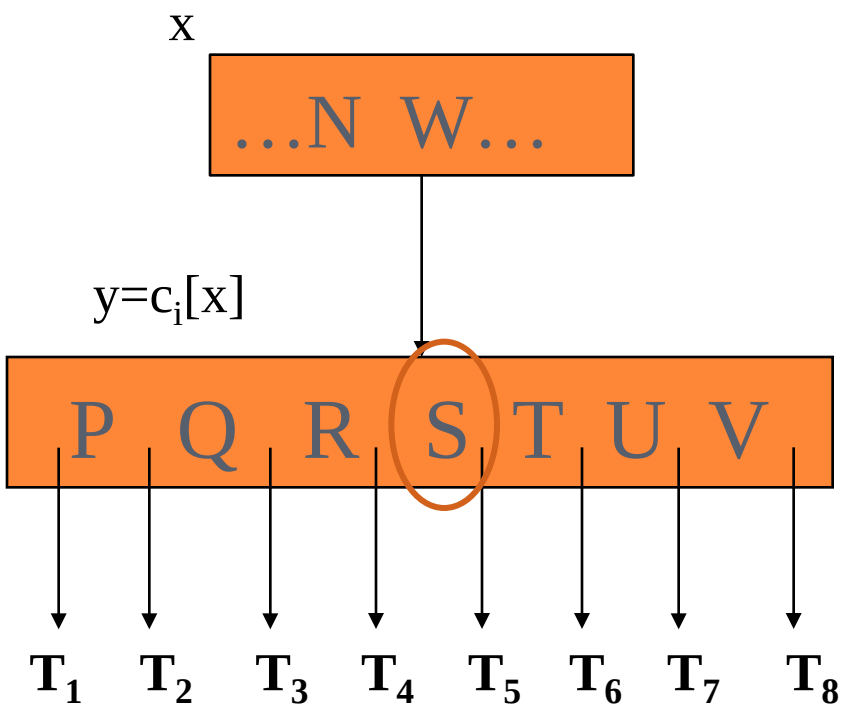
Why need split

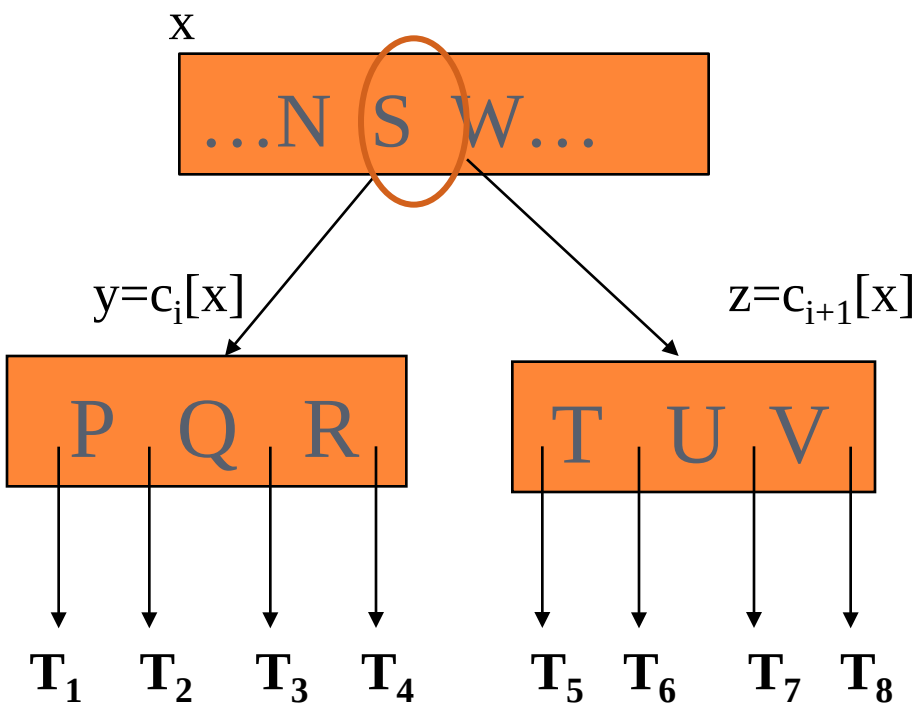
- When inserting a key into a B-tree ,we can’t simply create a new leaf node and insert it, as the result tree would fail to be a valid B-tree
- If is node becomes "too full“, (having 2t-1 keys) it is necessary to perform a split operation

SPLIT-OVERVIEW

- It splites 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


x
…N W…
y=c_i[x]
P Q R S T U V
T1 T2 T3 T4 T5 T6 T7 T8


x
…N W…
y=c_i[x]
P Q R S T U V
T1 T2 T3 T4 T5 T6 T7 T8


x
…N S W…
y=c_i[x]
z=c_i+1[x]
P Q R
T U V
T1 T2 T3 T4
T5 T6 T7 T8

SPLIT-PSEUDOCODE

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

```
1  z← Allocate-Node()
2  leaf[z] ← leaf[y]
3  n[z] ← t - 1
4  for j ← 1 to t - 1
5      do key_j[z] ← key_{j+t}[y]
6  if not leaf[y]
7      then for j ← 1 to t
8          do c_j[z] ← c_{j+t}[y]
9  n[y] ← t - 1
```

SPLIT-PSEUDOCODE (CONT.)

10 **for** $j \leftarrow n[x] + 1$ **downto** $i + 1$
11 **do** $c_{j+1}[x] \leftarrow c_j[x]$
12 $c_{i+1}[x] \leftarrow z$
13 **for** $j \leftarrow n[x]$ **downto** i
14 **do** $key_{j+1}[x] \leftarrow key_j[x]$
15 $key_i[x] \leftarrow key_t[y]$
16 $n[x] \leftarrow n[x] + 1$
17 Disk-Write(y)
18 Disk-Write(z)
19 Disk-Write(x)

Split-analysis

- The split operation moves the median key of node *y* into its parent *x* where *y* is the *ith* 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*.

SPLIT-ANALYSIS (CONT.)

- 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 parent 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

INSERT-OVERVIEW

- 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 not be full or another split operation is required. This process may repeat all the way up to the root and may require splitting the root node.

INSERT-OVERVIEW (CONT.)

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

INSERT-PSEUDOCODE (1)

```
B-Tree-Insert(T, k)
1  r ← root[T]
2  if n[r] = 2t - 1
3      then s ← Allocate-Node()
4           root[T] ← s
5           leaf[s] ← FALSE
6           n[s] ← 0
7           c1 ← r
8           B-Tree-Split-Child(s, 1, r)
9           B-Tree-Insert-Nonfull(s, k)
10     else B-Tree-Insert-Nonfull(r, k)
```

INSERT-PSEUDOCODE (2)

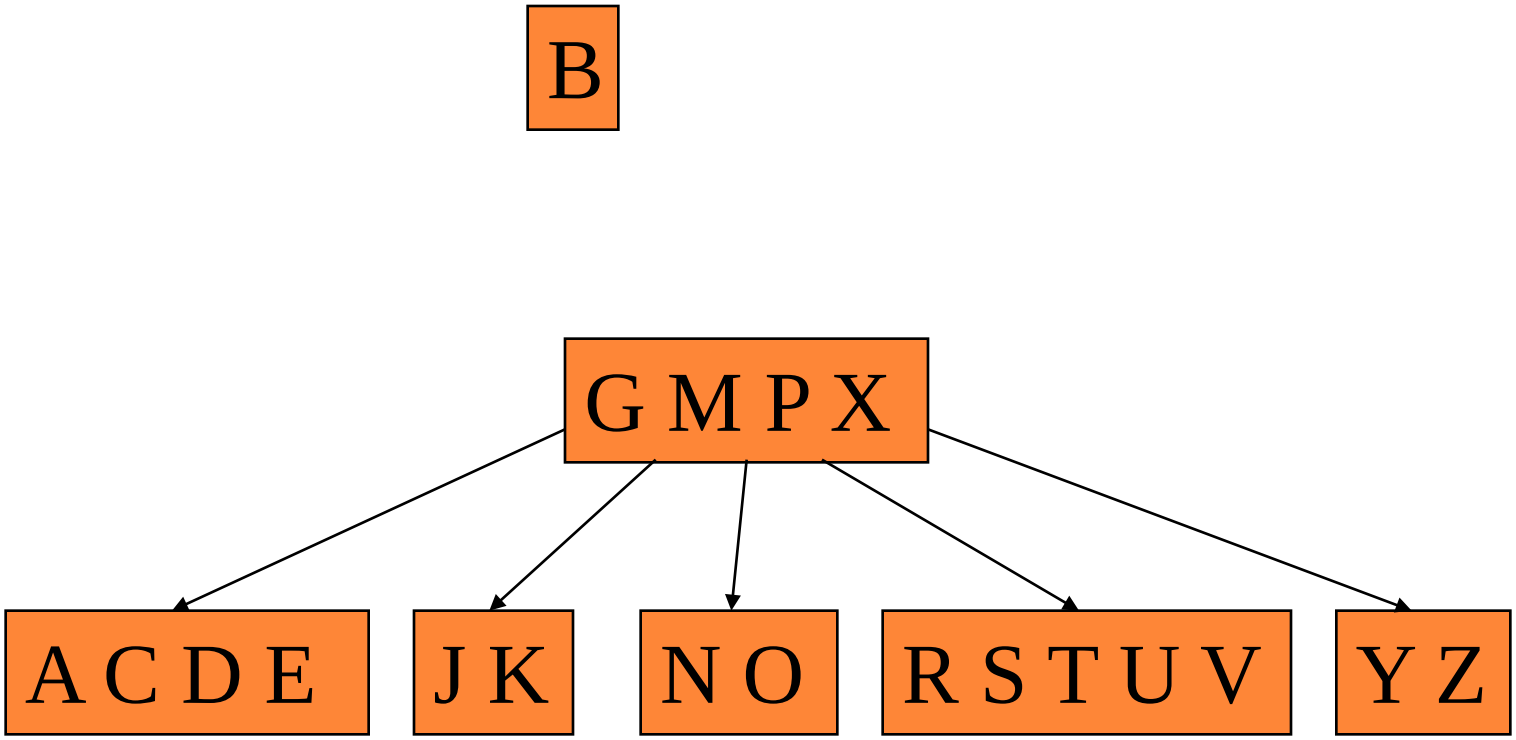
B-Tree-Insert-Nonfull(*x*, *k*)

```
1  i ← n[x]
2  if leaf[x]
3      then while i ≥1 and k < key_i[x]
4                do key_{i+1}[x] i ← key_i[x]
5           i ← i - 1
6           key_{i+1}[x] ← k
7           n[x] ← n[x] + 1
8           Disk-Write(x)
```

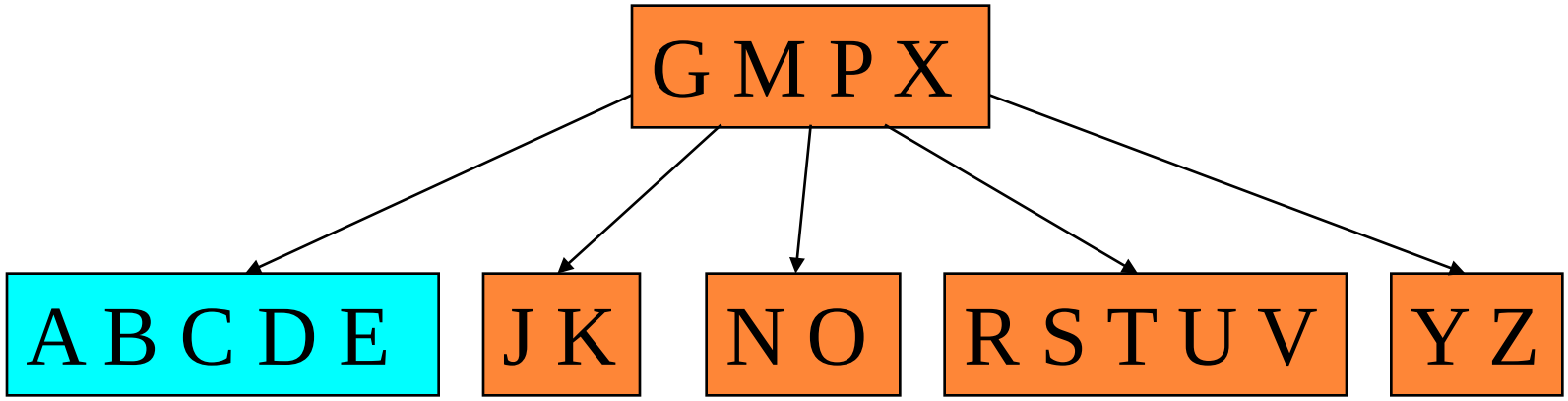
INSERT-PSEUDOCODE (2 CONT.)

```
9     else while i ≥1 and k < key_i[x]
10              do i ← i - 1
11          i ← i + 1
12          Disk-Read(c_i[x])
13          if n[c_i[x]] = 2t - 1
14             then B-Tree-Split-Child(x, i, c_i[x])
15                     if k > key_i[x]
16                        then i ← i + 1
17          B-Tree-Insert-Nonfull(c_i[x], k)
```

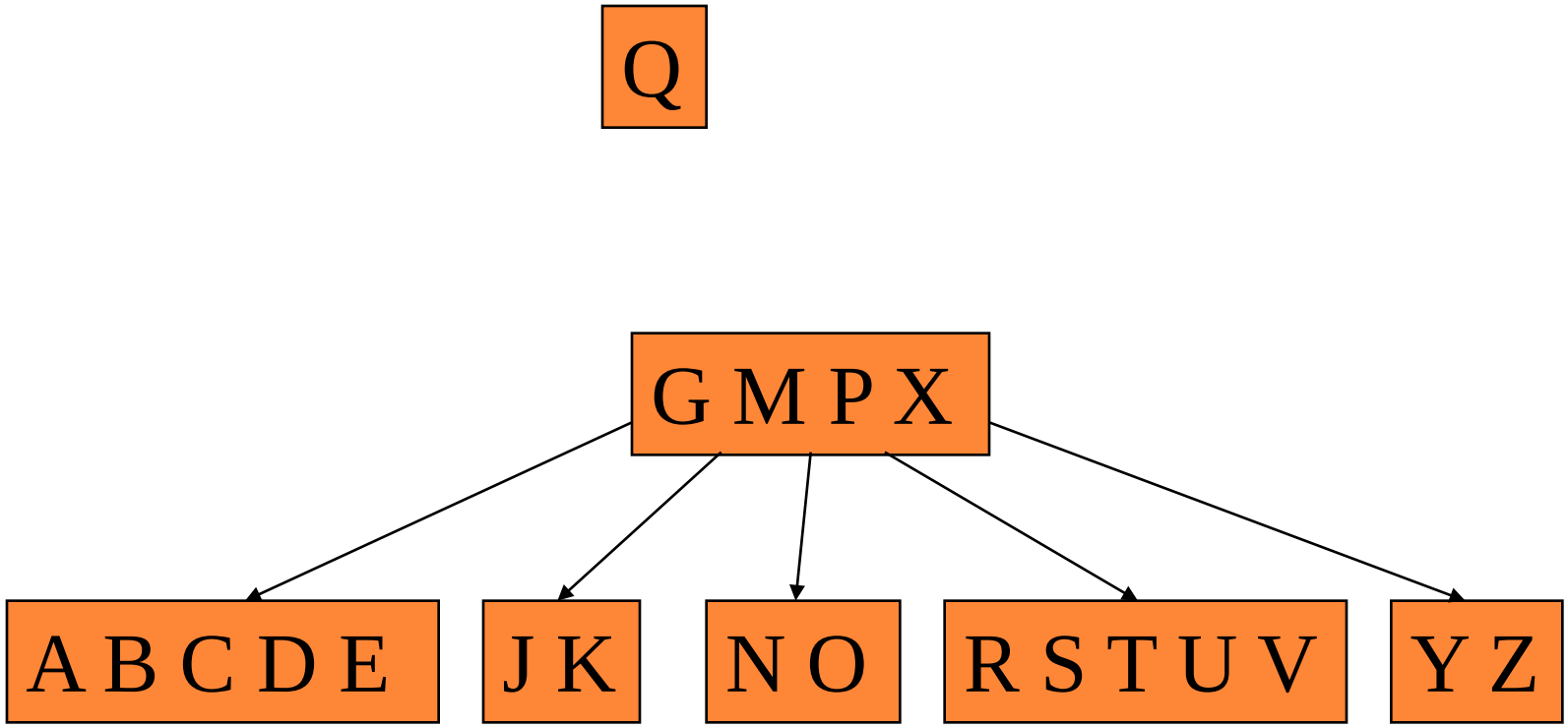
Insert B



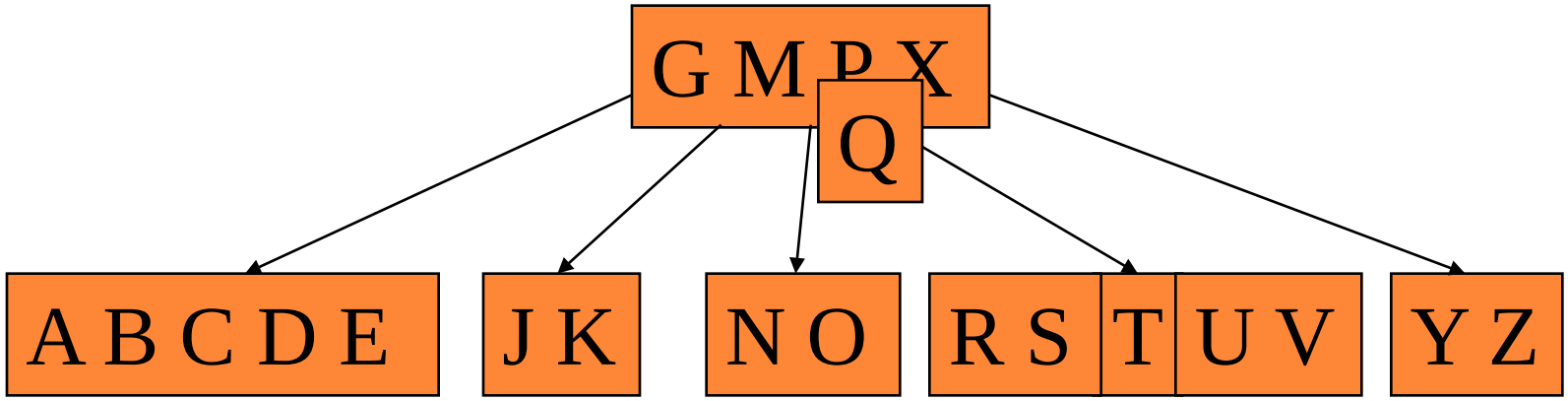
Insert B



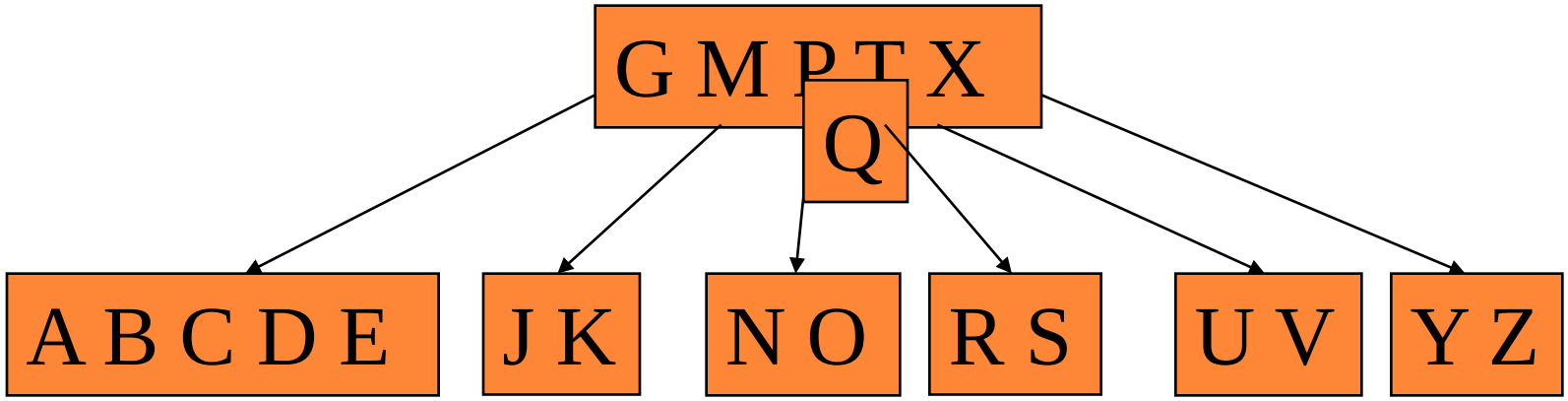
Insert Q



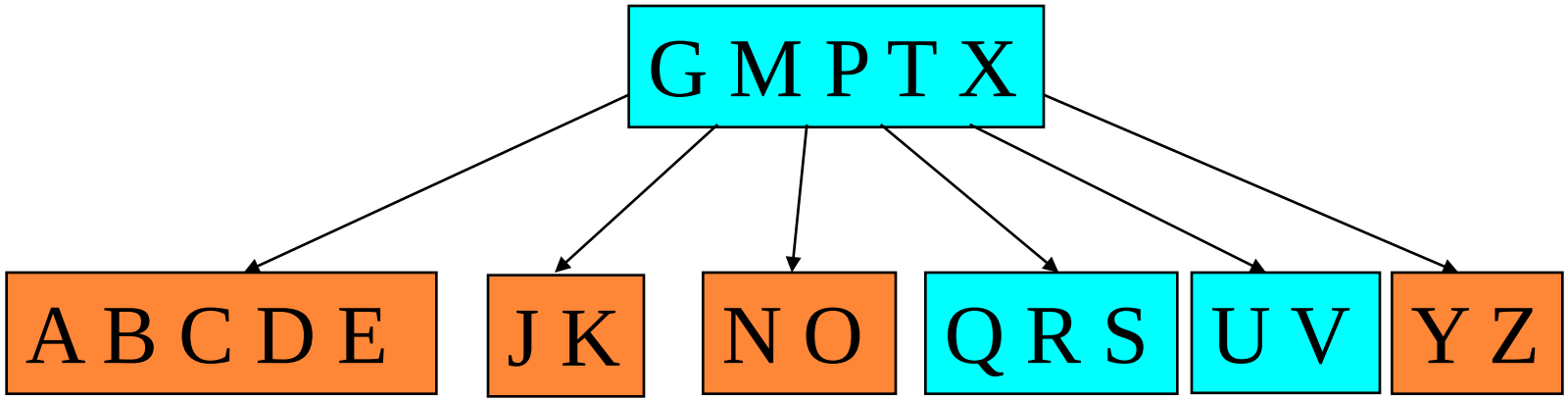
Insert Q



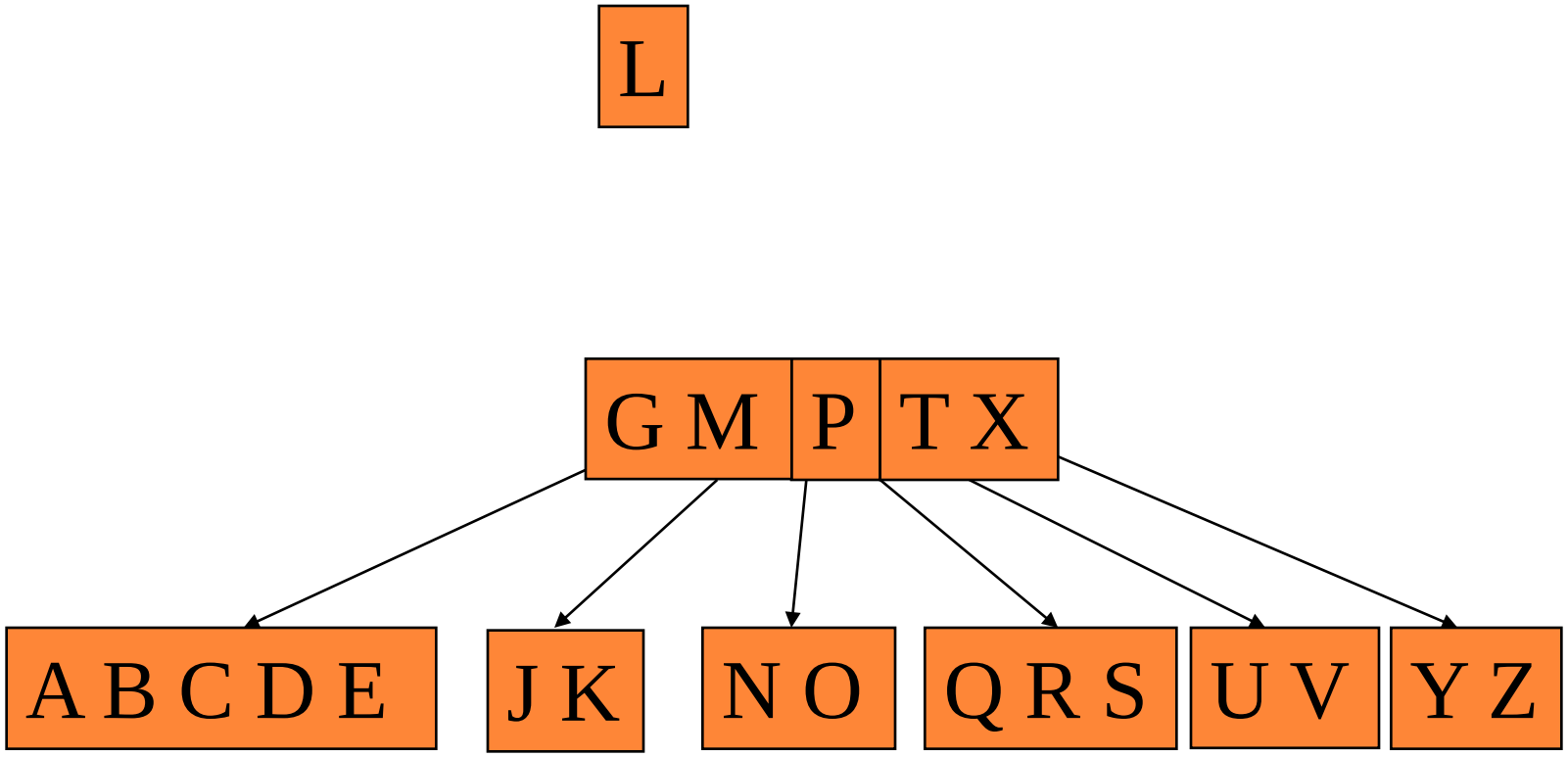
Insert Q



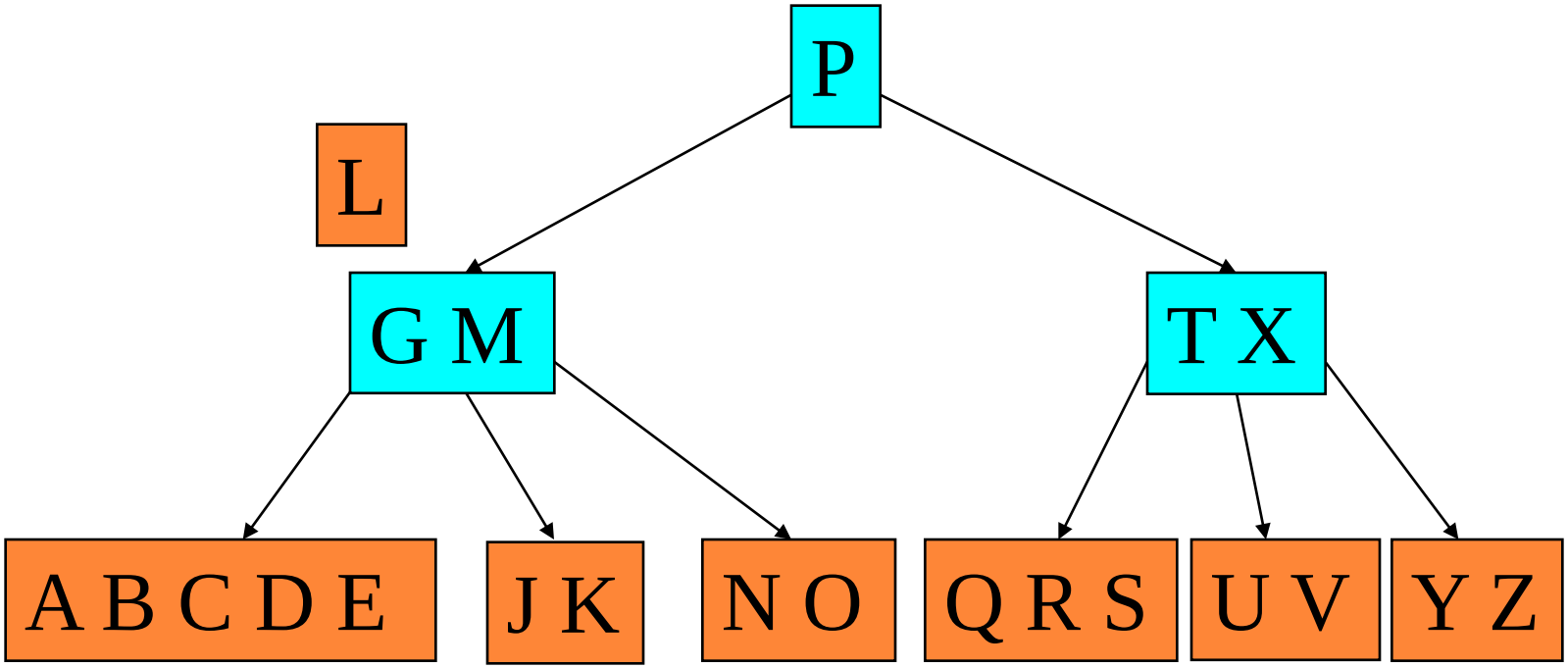
Insert Q



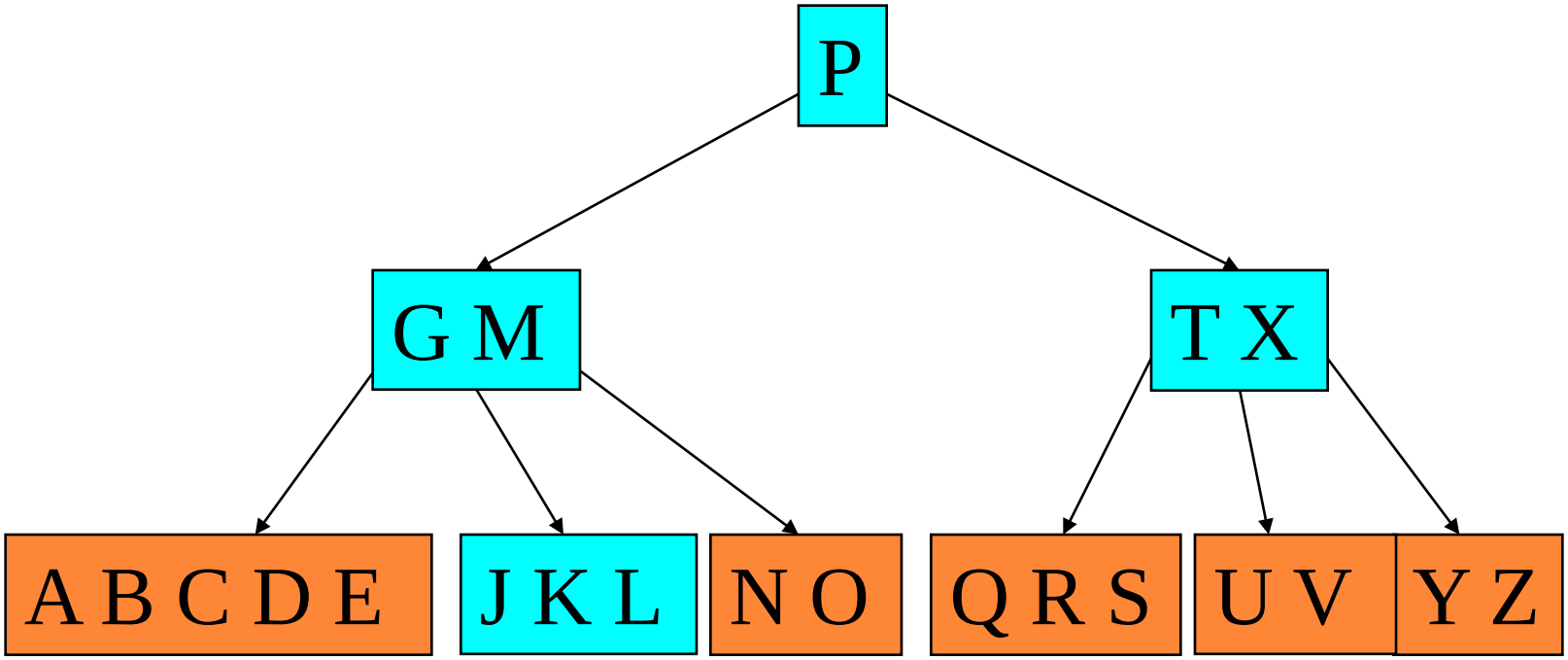
Insert L



Insert L

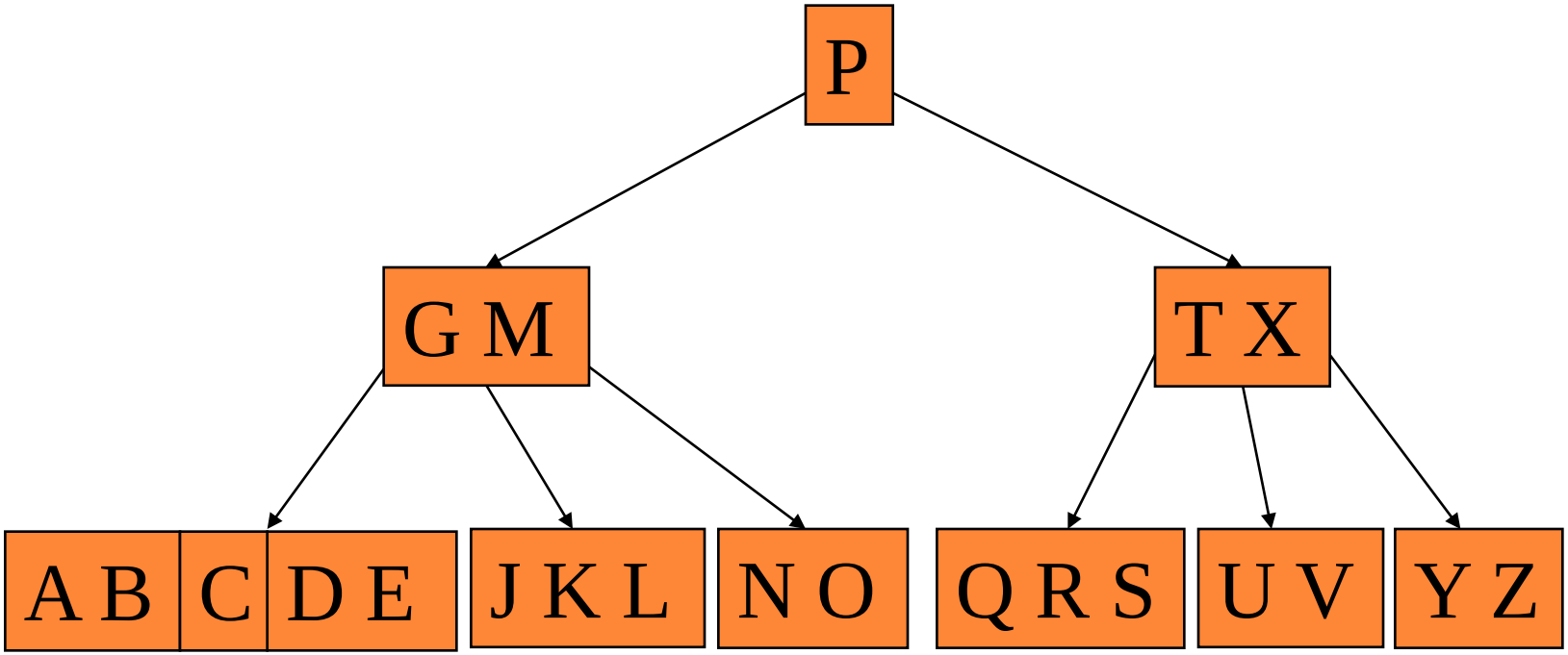


Insert L

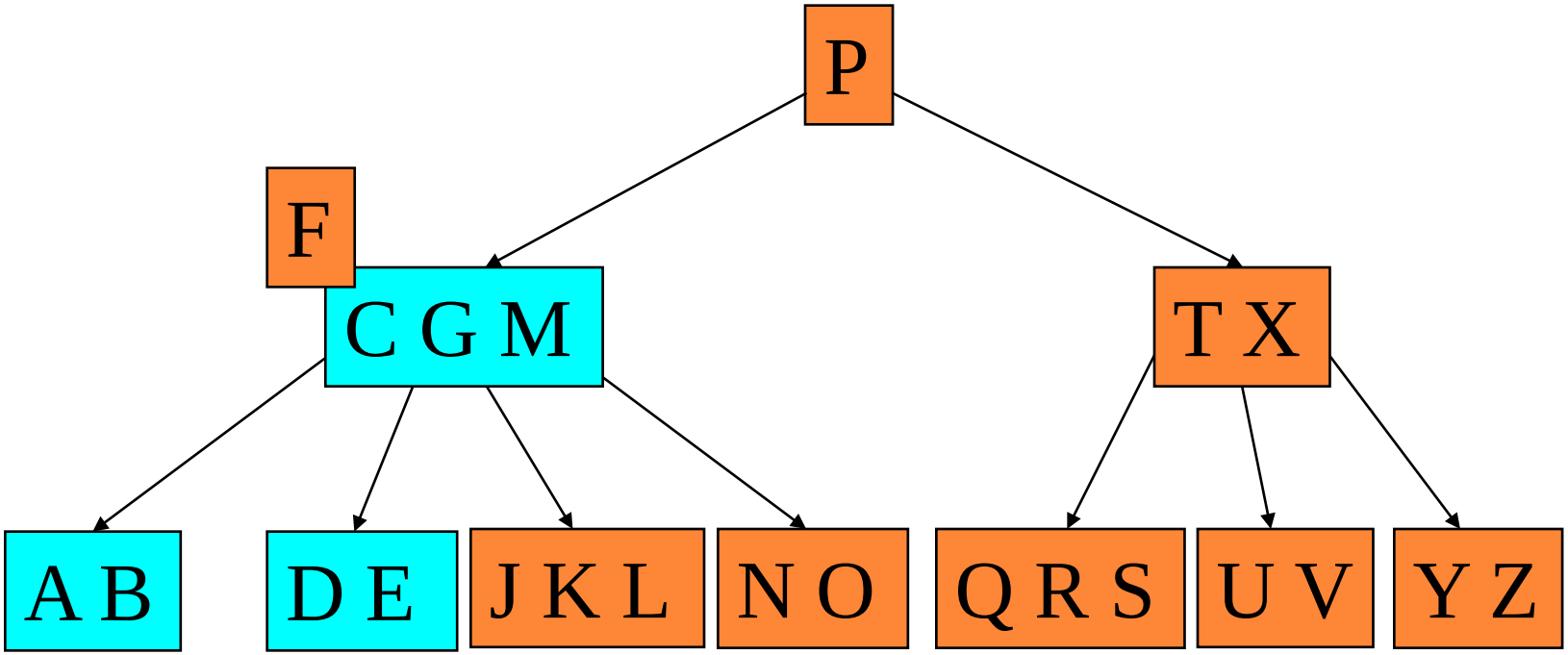


Insert F

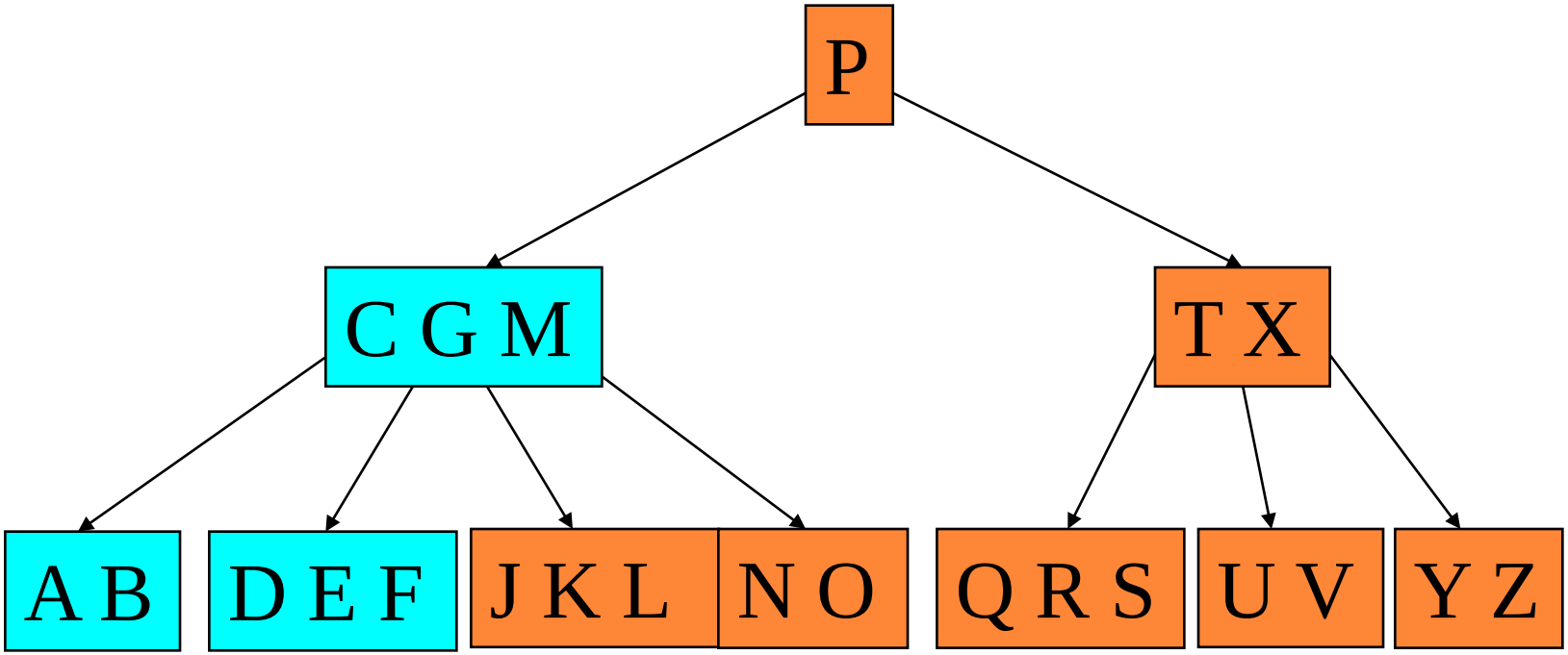
F



Insert F



Insert F



INSERT-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 unecessary 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 $O(t \log_t n)$ 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*

DELETE-OVERVIEW

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

DELETE-CASES

- 1. If the key *k* is in node *x* and *x* is a leaf, delete the key *k* from *x*. example
- 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 downwardpass.) example

DELETE-CASES (CONT.)

- 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.*example*

DELETE-CASES (CONT.)

- 3. If the key *k* is not present in internal node *x*, determine the root *ci*[*x*] of the appropriate subtree that must contain *k*, if *k* is in the tree at all. If *ci*[*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 *ci*[*x*] has only *t* - 1 keys but has an immediate sibling with at least *t* keys, give *ci*[*x*] an extra key by moving a key from *x* down into *ci*[*x*], moving a key from *ci*[*x*]'s immediate left or right sibling up into *x*, and moving the appropriate child pointer from the sibling into *ci*[*x*]. example

DELETE-CASES (CONT.)

- b. If *ci*[*x*] and both of *ci*[*x*]'s immediate siblings have *t* - 1 keys, merge *ci*[*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. example

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 nonroot nodes to be at least 2/3 full instead of 1/2

Outline

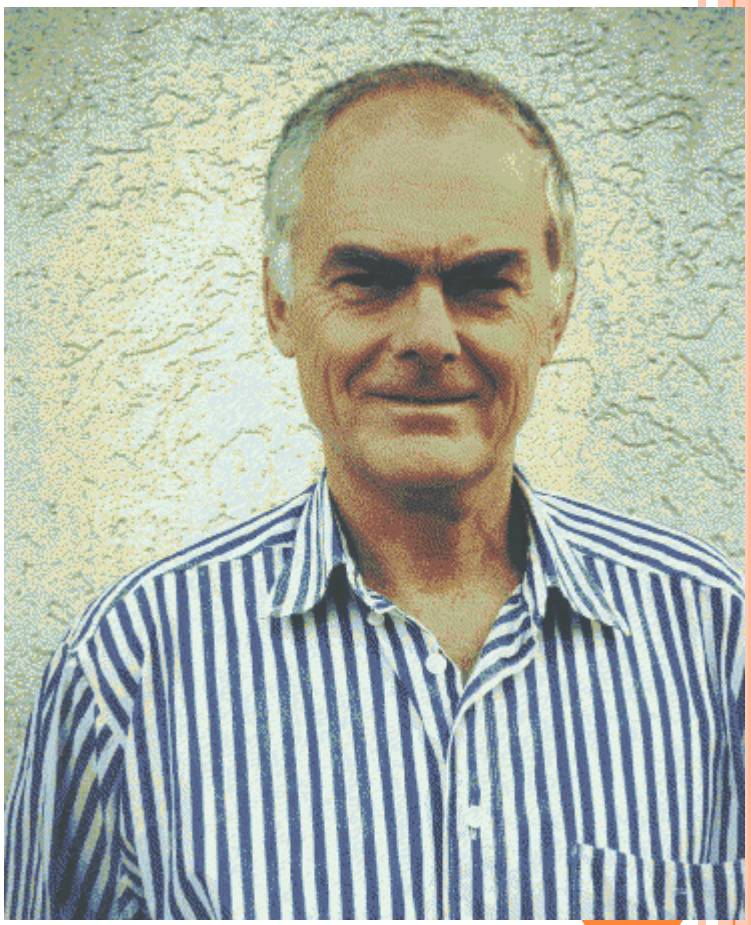
- Introduction
- Definition
- Basic operations
- *Applications*

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

- Professor (emeritus) of Informatics at the Technical University of Munich since 1972
- Famous for inventing two data sorting structures: the B-tree with Edward M. McCreight, and later the UB-tree with Volker Markl
- a recipient of 2001 ACM SIGMOD Edgar F. Codd Innovations Award



Edward M. McCreight

- Attended the College of Wooster ('66) and Carnegie-Mellon University
- Have worked at Boeing Aircraft and Xerox PARC
- Guest professor at the University of Washington, Stanford University, the Technical University of Munich, and the Swiss Federal Institute of Technology in Zürich