File Structures

Ch09. B. B-Tree Search, Insert

2020. Spring

Instructor: Joonho Kwon

jhkwon@pusan.ac.kr

Data Science Lab @ PNU

Slides are based on the textbook.

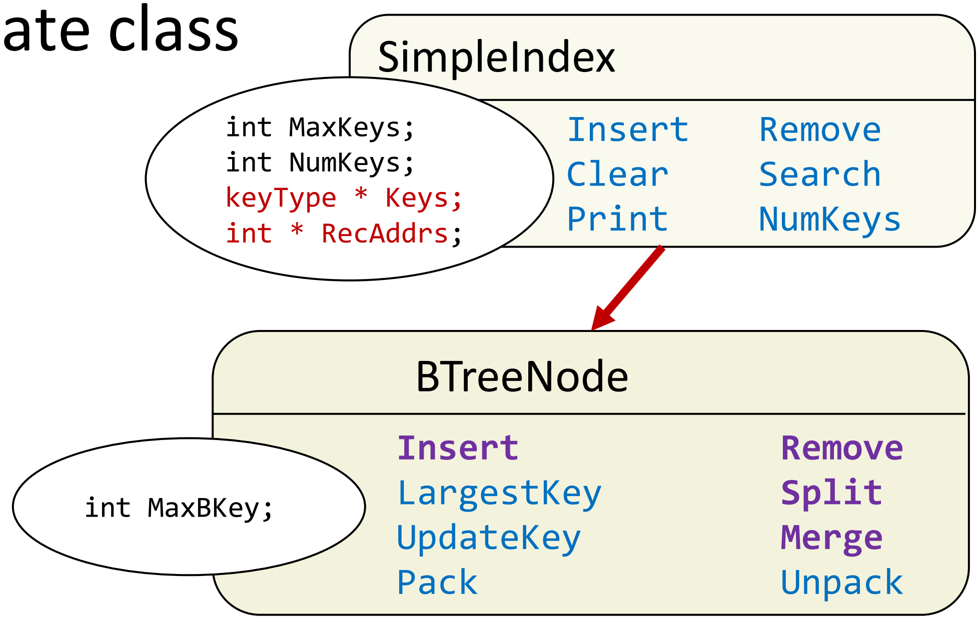
datalab
data science laboratory

Outline

- 9.7 An Object-Oriented Representation of B-Trees
- 9.8 B-Tree Methods Search, Insert, and Others
- 9.10. Formal Definition of B-tree properties
- 9.11 Worst-case search depth
- 9.9 B-Tree Nomenclature

An OO representation of B-Trees

- OO representation
  - B-tree is an index file associated with a data file
  - Most of operations on B-trees, insertion and deletion, are applied to the B-tree nodes in memory
  - The template BTreeNode class based on the SimpleIndex template class



Recap: SimpleIndex

- SimpleIndex Class

```
template <class keyType>
class SimpleIndex
{
public:
    SimpleIndex (int maxKeys = 100, int unique = 1);
    ~SimpleIndex ();
    void Clear (); // remove all keys from index
    int Insert (const keyType key, int recAddr);
    int Remove (const keyType key, const int recAddr = -1);
    int Search (const keyType key, const int recAddr = -1,
            const int exact = 1) const;
    …
protected:
    int MaxKeys;
    int NumKeys;
    keyType * Keys;
    int * RecAddrs;
    int Find (const keyType key, const int recAddr = -1,
            const int exact = 1) const;
    …
};
```

Recap: SimpleIndex

- Insert()

```
template <class keyType>
int SimpleIndex<keyType>::Insert (const keyType key, int recAddr)
{
    int i;
    int index = Find (key);
    if (Unique && index >= 0) return 0; // key already in
    if (NumKeys == MaxKeys) return 0; //no room for another key
    for (i = NumKeys-1; i >= 0; i--)
    {
        if (key > Keys[i]) break; // insert into location i+1
        Keys[i+1] = Keys[i];
        RecAddrs[i+1] = RecAddrs[i];
    }
    Keys[i+1] = key;
    RecAddrs[i+1] = recAddr;
    NumKeys ++;
    return 1;
}
```

Class BTreeNode (1/2)

- Representing B-Tree Nodes in memory
- Public methods:
  - insert : simply calls SimpleIndex::Insert and then check for overflow
  - remove a key, split and merge nodes
  - Search
    - inherited from SimpleIndex class(works perfectly well)
  - pack/unpack
    - manage the difference between the memory and the disk representation of BTreeNode objects
- Protected member
  - store the file address of the node and the minimum and maximum number of keys

Class BTreeNode (2/2)

```
// this is the in-memory version of the BTreeNode
template <class keyType>
class BTreeNode: public SimpleIndex <keyType>
{
  public:
    BTreeNode(int maxKeys, int unique = 1);
    int Insert (const keyType key, int recAddr);
    int Remove (const keyType key, int recAddr = -1);
    int LargestKey (); // returns value of largest key
    int Split (BTreeNode<keyType> * newNode); // move keys into newNode
    int Merge (BTreeNode<keyType> * fromNode); // move keys from fromNode
    int UpdateKey (keyType oldKey, keyType newKey, int recAddr = -1);
    int Pack (IOBuffer& buffer) const;
    int Unpack (IOBuffer& buffer);
    static int InitBuffer (FixedFieldBuffer & buffer,
        int maxKeys, int keySize = sizeof(keyType));
protected:
    int MaxBKeys; // maximum number of keys in a node
    friend class BTree<keyType>;
};
```

BTreeNode constructor

- # of keys in the BTreeNode
  - Actually one more than the order of the tree
  - The SimpleIndex constructor creates an index record with maxKeys + 1 elements

```
template <class keyType>
BTreeNode<keyType>::BTreeNode(int maxKeys, int unique)
:SimpleIndex<keyType>(maxKeys+1, unique)
{   Init ();}
```

BTreeNode::Insert()

- Method Insert simply calls SimpleIndex::Insert
  - Making the index record larger allows to create an overfull node
    - Respond to the overflow in an appropriate fashon
    - 1 for success, 0 for failure and -1 for overflow

```
template <class keyType>
int BTreeNode<keyType>::Insert (const keyType key, int recAddr)
{
    int result;
    result = SimpleIndex<keyType>::Insert (key, recAddr);
    if (!result) return 0; // insert failed
    // this is modified
    // accessing protected members of superclass in C++ with templates
    if (this->NumKeys >= this->MaxKeys) return -1; // node overflow
    //if (NumKeys >= MaxKeys) return -1; // node overflow
    return 1;
}
```

BTreeNode::Remove()

- Can create an underfull node
  - 1 for success, 0 for failure and -1 for underflow

```
template <class keyType>
int BTreeNode<keyType>::Remove (const keyType key, int recAddr)
{
    int result;
    result = SimpleIndex<keyType>::Remove (key, recAddr);
    if (!result) return 0; // remove failed
    // modified
    // accessing protected members of superclass in C++ with templates
    if (this->NumKeys < MinKeys) return -1; // node underflow
    //if (NumKeys < MinKeys) return -1; // node underflow
    return 1;
}
```

Class BTree (1/3)

- **Supporting Files of B-Tree Nodes**
  - Uses in-memory BTreeNode objects
  - adds the file access portion
  - enforces the consistent size of the nodes
- Methods
  - Create, Open, Close a B-Tree
  - Search, Insert, Remove key-reference pairs
- Protected area
  - Fetch: transfer nodes from disk to memory
  - Store: transfer nodes back to disk
  - root node, height of the tree, file of index records
  - BTNode **Node
    - used to keep a collection of tree nodes in memory and reduce disk access

Class BTree (2/3)

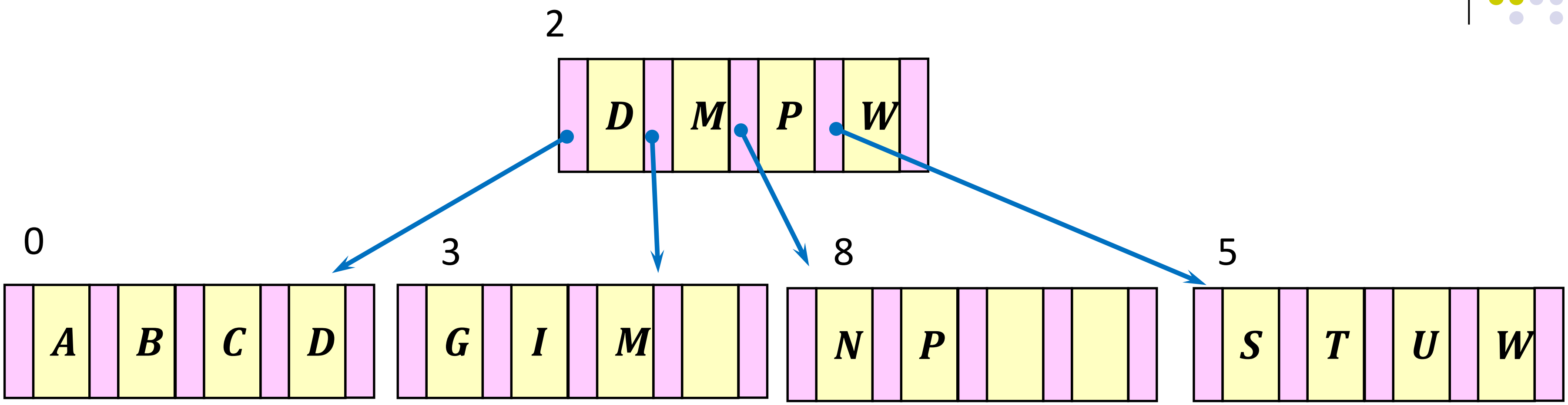
```
// this is the full version of the BTree
template <class keyType>
class BTree
{
    public:
    BTree(int order, int keySize = sizeof(keyType), int unique = 1);
    ~BTree();
    //int Open (char * name, int mode);
    int Open (char * name, ios_base::openmode mode);
    //int BTree<keyType>::Open (char * name, ios_base::openmode mode)
    int Create (char * name, ios_base::openmode mode);
    //int Create (char * name, int mode);
    int Close ();
    int Insert (const keyType key, const int recAddr);
    int Remove (const keyType key, const int recAddr = -1);
    int Search (const keyType key, const int recAddr = -1);
    void Print (ostream &);
    void Print (ostream &, int nodeAddr, int level);
protected:
```

Class BTree (3/3)

```
protected:
    typedef BTreeNode<keyType> BTNode;// useful shorthand
    BTNode * FindLeaf (const keyType key);

    // load a branch into memory down to the leaf with key
    BTNode * NewNode ();
    BTNode * Fetch(const int recaddr);
    int Store (BTNode *);
    BTNode Root;
    int Height; // height of tree
    int Order; // order of tree
    int PoolSize;
    BTNode ** Nodes; // pool of available nodes
    // Nodes[1] is level 1, etc. (see FindLeaf)
    // Nodes[Height-1] is leaf
    FixedFieldBuffer Buffer;
    RecordFile<BTNode> BTreeFile;
};
```

Page Structure



KEYCOUNT

	KEYCOUNT	KEY array	CHILD array
Page 2	4	*D M P W*	0 3 8 5
Page 3	3	*G I M*	Nil Nil Nil Nil

content of PAGE 2, 3

Outline

- 9.7 An Object-Oriented Representation of B-Trees
- 9.8 B-Tree Methods Search, Insert, and Others
- 9.10. Formal Definition of B-tree properties
- 9.11 Worst-case search depth
- 9.9 B-Tree Nomenclature

Algorithm for Search (1/4)

- Searching procedure
  - iterative
  - work in two stages
    - operating alternatively on entire pages (Class BTree)
      - **BTree::Search(key, recAddr);**
    - and then within pages (Class BTreeNode)
      - **SimpleIndex::Search(key, recAddr);**

  - Step1: Loading a page into memory
  - Step2: Searching through a page, looking for the key along the tree until it reaches the leaf level

Algorithm for Search (2/4)

- Specifications of Search and FindLeaf methods

```
Template <class keyType>
int BTree<keyType>::Search(const keyType key, const int recAddr)

template <class keyType>
BTreeNode<keyType>* BTree<keyType>::FindLeaf(const keyType key)
```

Search method

```
recAddr = btree.Search('L')
call FindLeaf('L');
Search key in the leaf node, and then
   if key exists, return the data file address of record with key 'L'
   otherwise, return -1
```

FindLeaf method

```
Search down to leafNode, beginning of the root
return the address of leafNode
```

Algorithm for Search (3/4)

- Example

```
RecAddr = btree.Search('L');
==>leafNode = FindLeaf(key);
==> recAddr = Nodes[level-1]->Search(key, -1, 0)
==> Nodes[level] = Fetch(recAddr);
```

```
template <class keyType>
int BTree<keyType>::Search (const keyType key, const int recAddr)
{
    BTNode * leafNode;
    leafNode = FindLeaf (key);
    return leafNode -> Search (key, recAddr);
}
```

Algorithm for Search (4/4)

- FindLeaf

```
// load a branch into memory down to the leaf with key
template <class keyType>
BTreeNode<keyType> * BTree<keyType>::FindLeaf (const keyType key)
{
    int recAddr, level;
    for (level = 1; level < Height; level++) {
        //inexact search
        recAddr = Nodes[level-1]->Search(key,-1,0);
        Nodes[level]=Fetch(recAddr);
    }
    return Nodes[level-1];
}
```

- Fetch

```
// load this node from File into a new BTreeNode
template <class keyType>
BTreeNode<keyType> * BTree<keyType>::Fetch(const int recaddr)
{
    int result;
    BTNode * newNode = new BTNode(Order);
    result = BTreeFile.Read (*newNode, recaddr);
    if (result == -1) return NULL;
    newNode -> RecAddr = result;
    return newNode;
}
```

Algorithm for Insertion (1/3)

- Observations of Insertion, Splitting, and Promotion
  - proceed all the way down to the leaf level
  - after finding the insertion location at the leaf level, the work proceeds upward from the bottom
- Iterative procedure as having three phases
  - Search to the leaf level, using FindLeaf method
  - Insertion, overflow detection, and splitting on the upward path
  - Creation of a new root node, if the current root was split

Algorithm for Insertion (2/3)

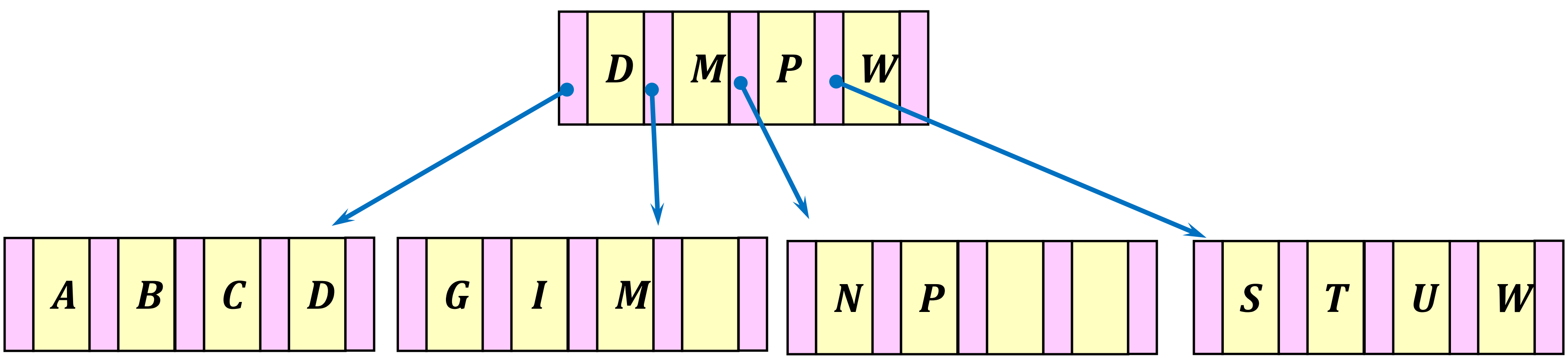
- With no redistribution
- (Step 1) Locate node on bottom most level in which to insert record
  - Location is determined by key search.
- (Step 2) If vacant record slot is available
  - insert the record so that key sequencing is maintained.
  - Then, update the pointer associated with the record (Pointer is null for level 0 records). Then Stop!
- (Step 3) If no vacant record slot exists
  - identify median record.
  - All records and pointers to the left of the median records are stored in one node (the original) and those to the right are stored in another node(the new node).

Algorithm for Insertion (3/3)

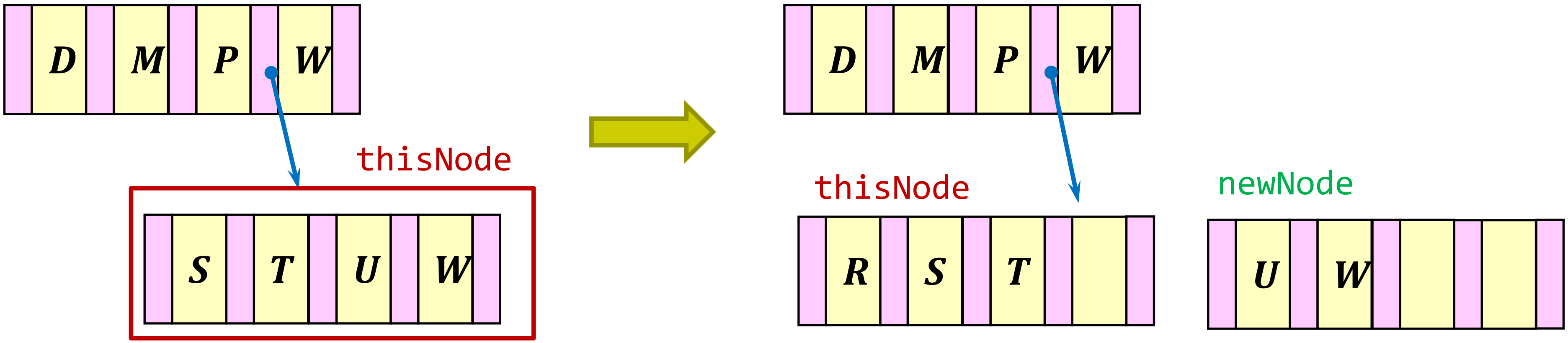
- (Step 4) If the topmost node was split
  - create a new topmost node which contains the median record identified in Step 3, filled with pointers to the original and split nodes.
  - Update the root node to point to the new topmost node. Then Stop!
- (Step 5) If topmost node was not split
  - prepare to insert median record identified in Step 3 and a pointer to the new node (created in Step 3).
  - Then Goto Step 2.
- Note : Step 4 makes B-tree increase in height by 1 level
- B-trees have 70% occupancy(like B+-trees) on an average

Example of Inserting R with codes (1/7)

- Current State at creating a B-tree (4/10)
  - Try to Insert R



Example of Inserting R with codes (2/7)



- (1) Search the leaf node for key R using FindLeaf

```
thisNode = FindLeaf (key);
```

- (2) Insert R into the leaf node

```
result = thisNode -> Insert (key, recAddr);
```

- (3) Detect an overflow
  - The node must be split into two nodes

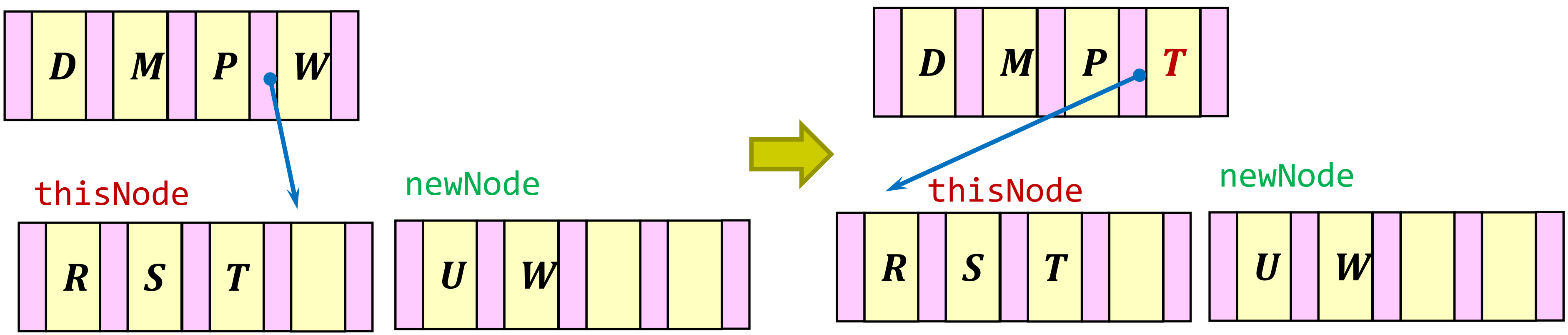
```
newNode = NewNode();
thisNode->Split(newNode);
Store(thisNode); Store(newNode);
```

Example of Inserting R with codes (3/7)

- BTreeNode::Split
  - Distribute the keys between the original page and the new page

```
template <class keyType>
int BTreeNode<keyType>::Split (BTreeNode<keyType> * newNode)
{
    // check for sufficient number of keys
    if (this->NumKeys < this->MaxKeys) return 0;
    // find the first Key to be moved into the new node
    int midpt = (this->NumKeys+1)/2;
    int numNewKeys = this->NumKeys - midpt;
    // check that number of keys for newNode is ok
    if (numNewKeys > newNode -> MaxBKeys|| numNewKeys < newNode -> MinKeys)
        return 0;
    // move the keys and recaddrs from this to newNode
    for (int i = midpt; i< this->NumKeys; i++) {
        newNode->Keys[i-midpt] = this->Keys[i];
        newNode->RecAddrs[i-midpt] = this->RecAddrs[i];
    }
    newNode->NumKeys = numNewKeys; // set number of keys in the two Nodes
    this->NumKeys = midpt; // Link the nodes together
    return 1;
}
```

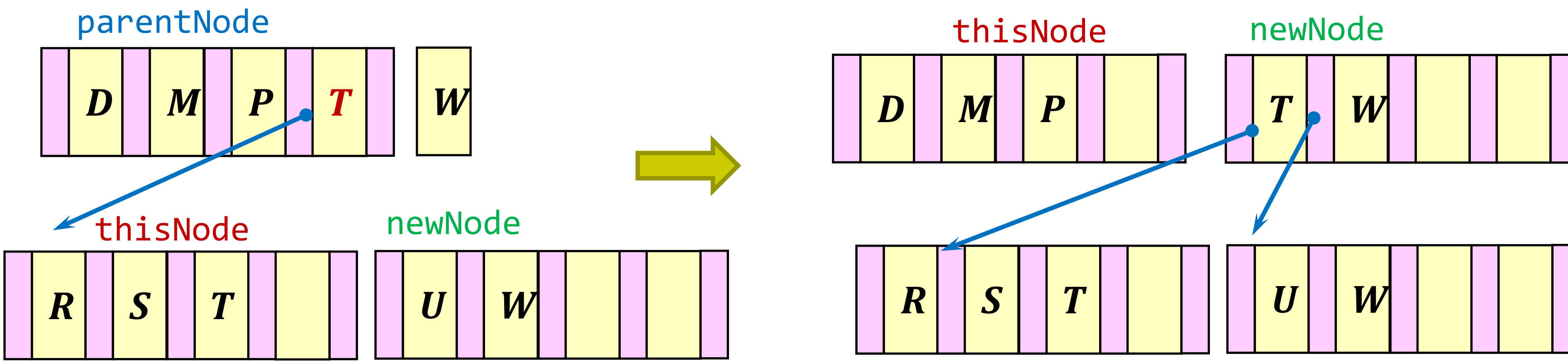
Example of Inserting R with codes (4/7)



- (4) Update the parent node
  - The largest key in thisNode has changed
  - thisNode->LargestKey() = ‘T’

```
level--; // go up to parent level
if (level < 0) break;
// insert newNode into parent of thisNode
parentNode = Nodes[level];
result = parentNode->UpdateKey(largestKey,thisNode->LargestKey());
```

Example of Inserting R with codes (5/7)



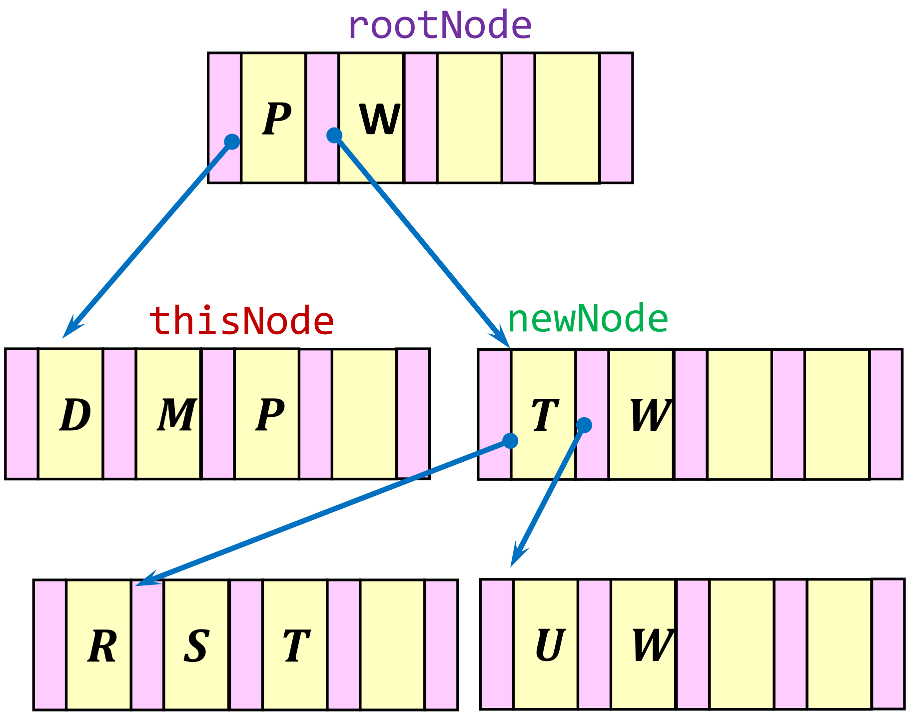
- (5) Insert the largest value(W) in the new node into the root

```
result = parentNode->Insert (newNode->LargestKey(),newNode->RecAddr);
thisNode=parentNode;
```

  - Result = -1
- Promote the key W (thisNode = paretNode)
  - cause the root to overflow with five keys
  - a new root node is created and the keys P and W are inserted : (D,M,P), (T,W)

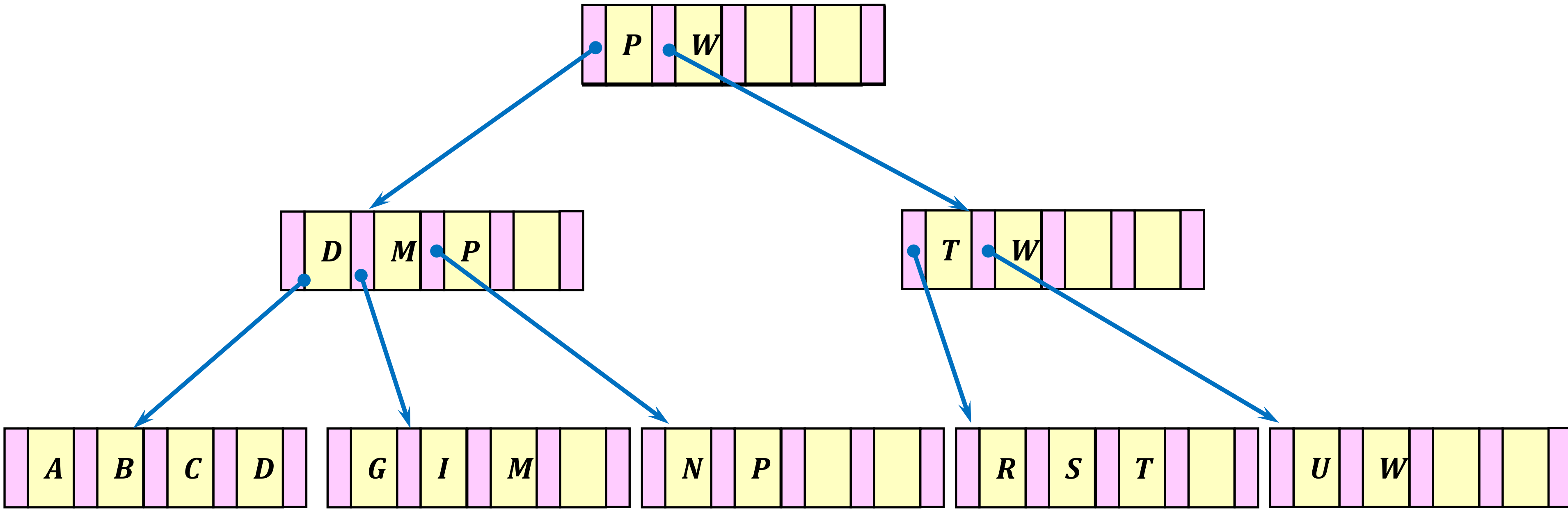
Example of Inserting R with codes (6/7)

- (6) Create a new root node, and insert the keys P & W into it




```
// put previous root into file
int newAddr = BTreeFile.Append(Root);
// insert 2 keys in new root node
Root.Keys[0]=thisNode->LargestKey();
Root.RecAddrs[0]=newAddr;
Root.Keys[1]=newNode->LargestKey();
Root.RecAddrs[1]=newNode->RecAddr;
Root.NumKeys=2;
Height++;
```

Example of Inserting R with codes (7/7)



Create, Open, and Close (1/2)

- Create()
  - Write the empty root node into the file BTreeFile
- Open()
  - Open BTreeFile and load the root node into memory from the first record in the file
- Close()
  - Store the root node into BTreeFile and close it

Create, Open, and Close (2/2)

```
template <class keyType>
int BTree<keyType>::Create (char * name, ios_base::openmode mode)
{
    int result;
    result = BTreeFile.Create(name, mode);
    if (!result) return result;
    // append root node
    result = BTreeFile.Write(Root);
    Root.RecAddr=result;
    return result != -1;
}
```

```
template <class keyType>
int BTree<keyType>::Close ()
{
    int result;
    result = BTreeFile.Rewind();
    if (!result) return result;
    result = BTreeFile.Write(Root);
    if (result==-1) return 0;
    return BTreeFile.Close();
}
```

Testing the B-Tree

- Full code of program to test creation and insertion of a B-tree

```
const char * keys="CSDTAMPIBWNGURKEHOLJYQZFXV";
const int BTreeSize = 4;
int main (int argc, char ** argv)
{
    int result, i;
    BTree <char> bt (BTreeSize);
    result = bt.Create ("testbt.dat",ios::in|ios::out);
    if (!result){cout<<"Please delete testbt.dat"<<endl;return 0;}
    for (i = 0; i<26; i++)
    {
        cout<<"Inserting "<<keys[i]<<endl;
        result = bt.Insert(keys[i],i);
        bt.Print(cout);
    }
    bt.Search(1,1);
    return 1;
}
```

Outline

- 9.7 An Object-Oriented Representation of B-Trees
- 9.8 B-Tree Methods Search, Insert, and Others
- 9.10. Formal Definition of B-tree properties
- 9.11 Worst-case search depth
- 9.9 B-Tree Nomenclature

Motivation

- Assume
  - We store 1,000,000 keys using a B-tree of order 512 (maximum of 511 keys per page)
- Question

In the worst case, what will be **the maximum number of disk accesses** requires to locate a key in the tree?

$\|$ **Same question**

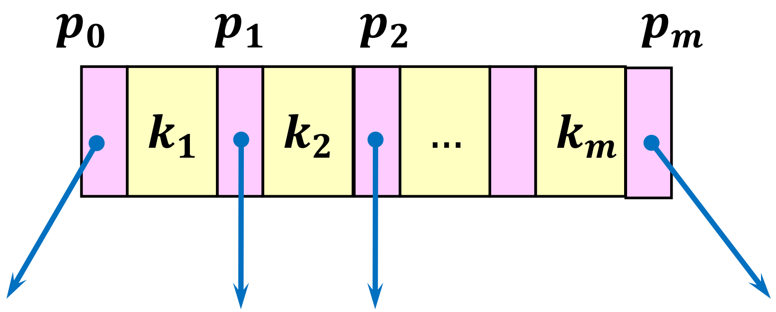
How deep the B-tree will be?

How to answer the worst case search depth?

- Observe the formal definition of B-tree properties
  - To calculate the minimum number of descendants with a B-tree of some given order
- The worst case occurs
  - when every page of the tree has only the minimum # of descendants

→ A maximal height with a minimum breadth

Recap: Formal definition of B-Tree Properties



- The properties of a B-tree of order m
  - 1. Every page has a maximum of m descendants
  - 2. Every page, except for the root and the leaves, has at least ceiling of m/2 (=$\lceil m/2 \rceil$) descendants
  - 3. The root has at least two descendants (unless it is a leaf)
  - 4. All the leaves appear **on the same level**
  - 5. The leaf level forms a complete, ordered index of the associated data file

Extract general pattern (1/3)

- A B-tree of order m
  - Level 1 (root)
    - Minimum number of descendants from the root page is 2

  → 2 descendants

3. The root has at least two descendants

2. Every page,…, has at least ceiling of m/2 (=$\lceil m/2 \rceil$) descendants

  - Level 2:
    - The second level of the tree has 2 pages
    - Each page of these pages, in turn, has at least $\lceil m/2 \rceil$ descendants

  → $2 \cdot \lceil m/2 \rceil$

Extract general pattern (2/3)

- A B-tree of order m
  - Level 3
    - The second level of the tree has $2 \cdot \lceil m/2 \rceil$ pages
    - Each page of these pages, in turn, has at least $\lceil m/2 \rceil$ descendants

  → $2 \cdot \lceil m/2 \rceil^2$

  - Level d
    - $2 \cdot \lceil m/2 \rceil^{d-1}$

Extract general pattern (3/3)

- The general pattern of the relation
  - between depth and the minimum number of descendants

Level	Minimum number of keys (children)
1 (root)	2
2	$2 \cdot \lceil m/2 \rceil$
3	$2 \cdot \lceil m/2 \rceil^2$
4	$2 \cdot \lceil m/2 \rceil^3$
…	…
d	$2 \cdot \lceil m/2 \rceil^{d-1}$

Worst case search depth (1/2)

- For a tree with N keys in its leaves
  - The relationship between keys and the minimum height d
    - $N \geq 2 \cdot \lceil m/2 \rceil^{d-1}$
- Solving the equation for d
  - $d \leq 1 + log_{\lceil m/2 \rceil}(\frac{N}{2})$

Worst case search depth (2/2)

- Solving the equation for d
  - $d \leq 1 + log_{\lceil m/2 \rceil}(\frac{N}{2})$

- Example
  - B-tree order d= 512, N=1,000,000 keys
    - $d \leq 1 + log_{256}(500{,}000) = 3.37$
    - Dept of no more than level 3

Outline

- 9.7 An Object-Oriented Representation of B-Trees
- 9.8 B-Tree Methods Search, Insert, and Others
- 9.10. Formal Definition of B-tree properties
- 9.11 Worst-case search depth
- 9.9 B-Tree Nomenclature

B-Tree Nomenclature

- Be aware that terms are not uniform in the literature
  - Definitions are also quite different
  - In fact, there are a number of B-tree variations

- In this text book
  - "B tree" for B+ tree by other books
  - "B+ tree" is B+ tree with a linked list of sorted data blocks

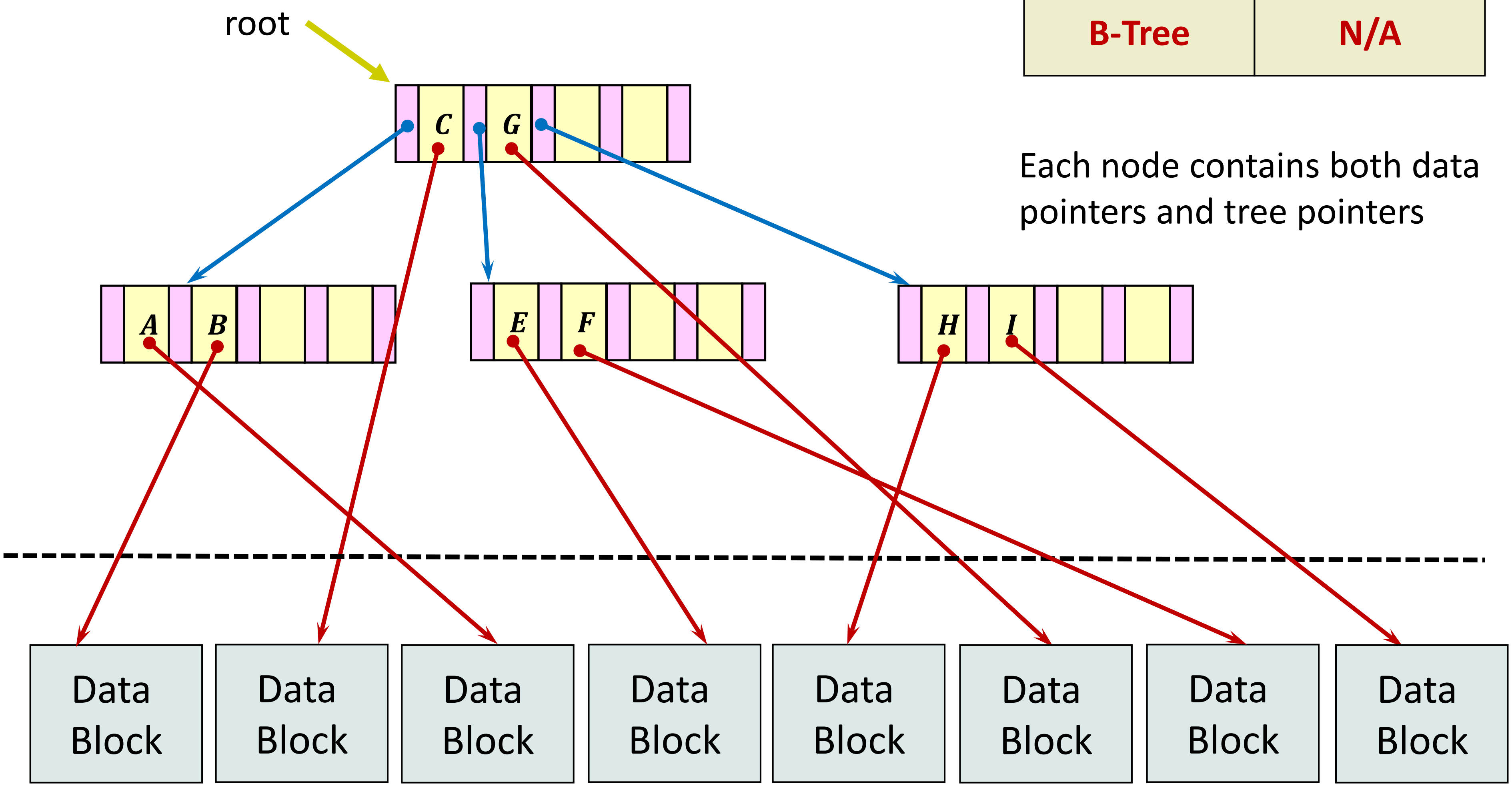
Difference between B-tree and B+-tree

- In a B-tree
  - pointers to data records exist at all levels of the tree

- In a B+-tree
  - all pointers to data records exists only at the leaf-level nodes
  - A B+-tree can have less levels (or higher capacity of search values) than the corresponding B-tree

B-tree in other books

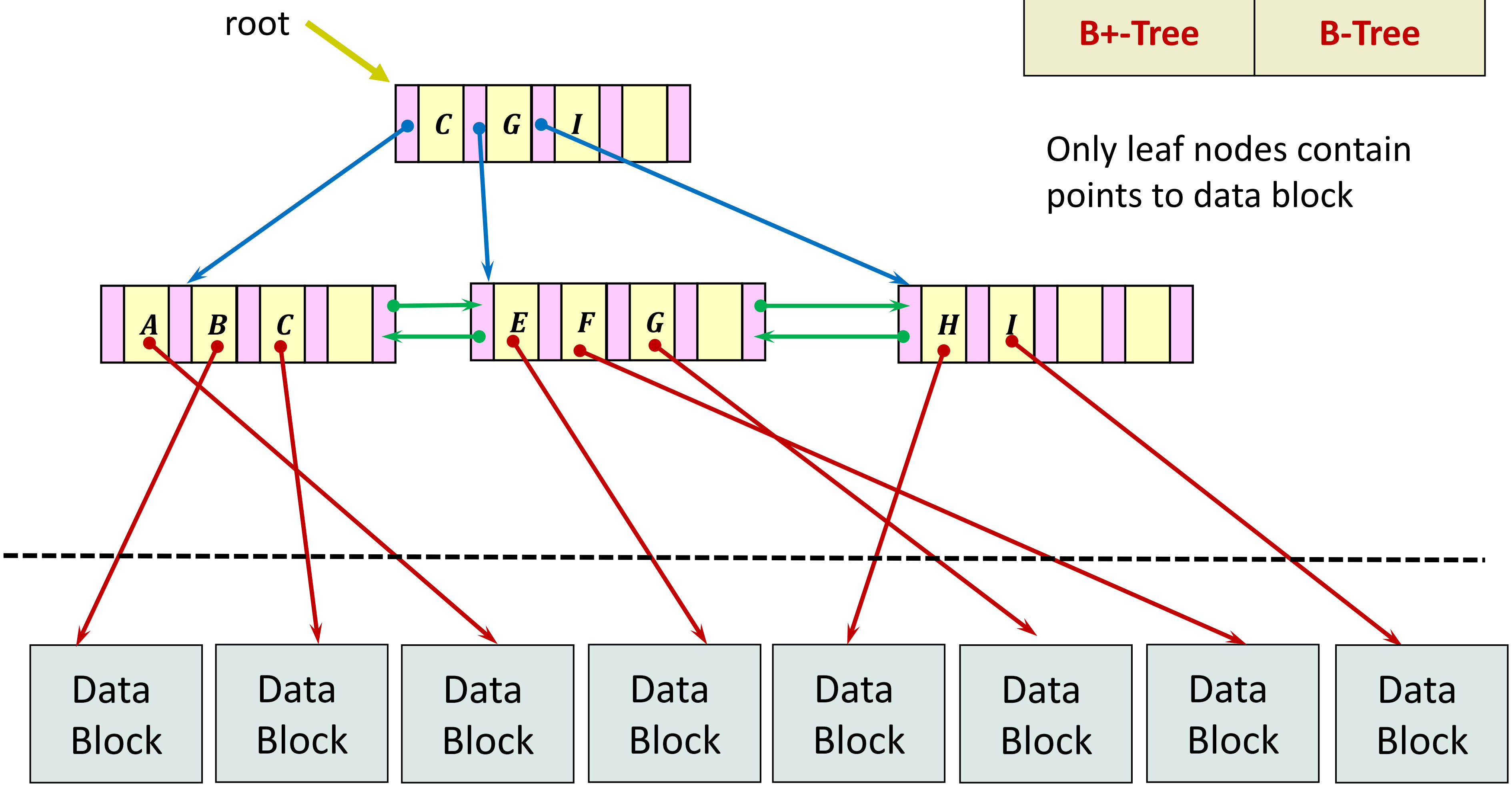
Other Book	Our Book
B-Tree	**N/A**



Each node contains both data pointers and tree pointers

B+-tree in other books

Other Book	Our Book
B+-Tree	**B-Tree**

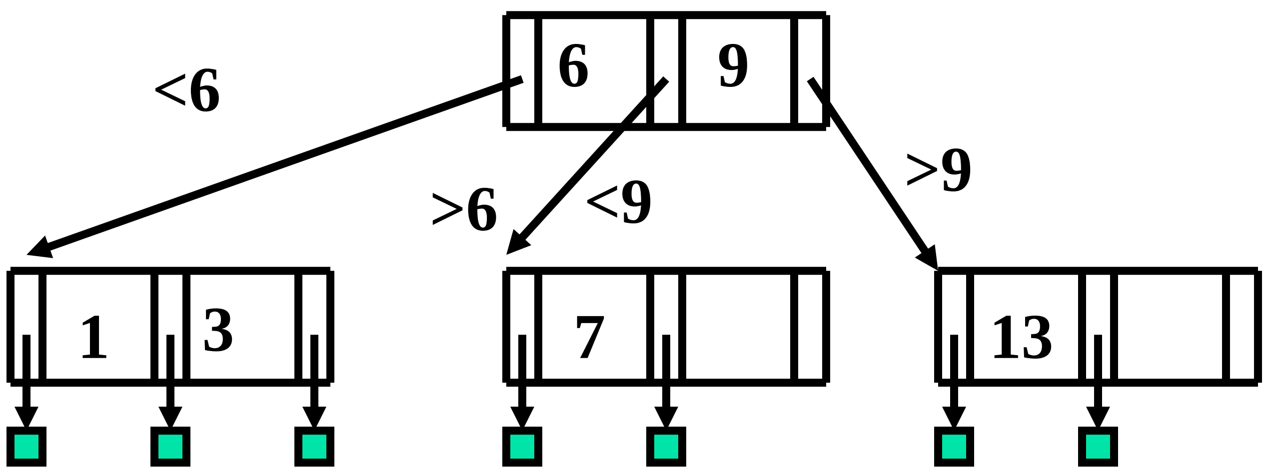


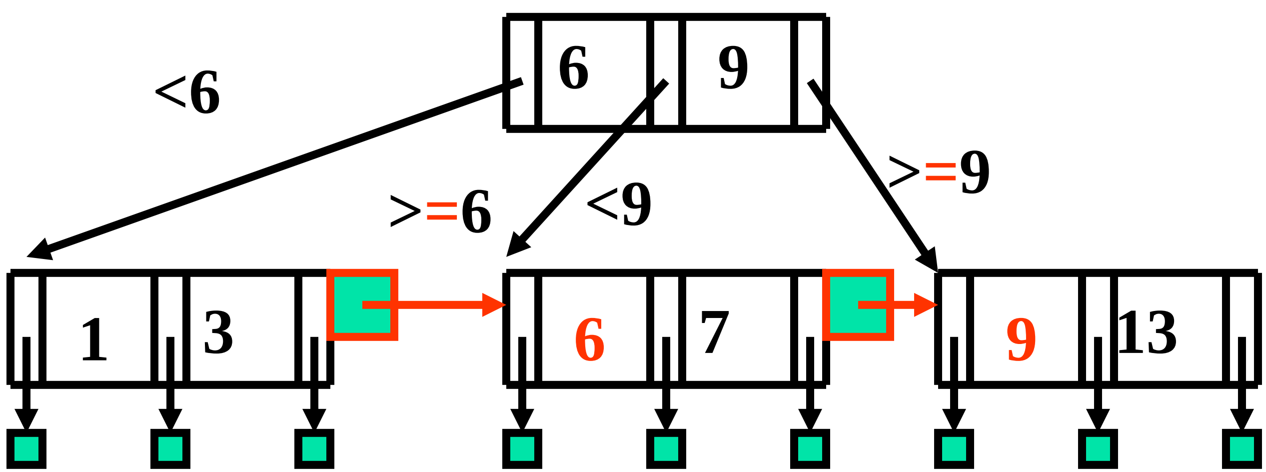
Only leaf nodes contain points to data block

Another aspect (node structures)

- B-Tree in other text
  - Homogeneous trees
    - leaf nodes and interior nodes have same structures;
    - Each contains both data pointers and tree pointers
  - Average search length less for homogeneous trees
    - because some searches may conclude before reaching a leaf node
- B+-Tree in other text
  - Heterogeneous trees
  - leaf nodes and interior nodes have different structures

Recap: B-tree and B+ tree





Comparison of B-Tree and B+-Tree in other text

Topic	B-tree	B+ -Tree
Algorithm complexity for insertion	Rather complexity	More simple
Retrieval efficiency	Less efficiency (B-tree is tall & spindle)	More efficient B+-tree is short & bushy
Storage efficiency	Slightly more efficient (is less space)	Less efficient (is more space)
1-pass structure creation algorithms	Rather complex	Simple

Q&A