Chapter 6: Multiway Trees

• Tree whose outdegree is not restricted to 2 while retaining the general properties of binary search trees.
M-Way Search Trees

• Each node has $m - 1$ data entries and $m$ subtree pointers.

• The key values in a subtree such that:
  - $\geq$ the key of the left data entry
  - $<$ the key of the right data entry.
M-Way Search Trees

35 45
35
60 70
75

85 95

125 135
175

50 100 150
M-Way Node Structure

entry
  key <key type>
  data <data type>
  rightPtr <pointer>
end entry

node
  firstPtr <pointer>
  numEntries <integer>
  entries <array[1 .. m-1] of entry>
end node
B-Trees

• M-way trees are unbalanced.

B-Trees

• A B-tree is an m-way tree with the following additional properties (m >= 3):
  
  - The root is either a leaf or has at least 2 subtrees.
  - All other nodes have at least \([m/2]\) - 1 entries.
  - All leaf nodes are at the same level.
B-Trees

m = 5
B-Tree Insertion

• Insert the new entry into a leaf node.

• If the leaf node is overflow, then split it and insert its median entry into its parent.
B-Tree Insertion

Insert 78, 21, 14, 11

Insert 97

Insert 85, 74, 63

Insert 45, 42, 57

Cao Hoang Tru
CSE Faculty - HCMUT

17 November 2008
B-Tree Insertion

Insert 20, 16, 19

Insert 52, 30, 21
B-Tree Insertion

**Algorithm**  
BTreeInsert (val root <pointer>, val data <record>)

Inserts data into B-tree. Equal keys placed on right branch.

**Pre**  
root is a pointer to the B-tree. May be null.

**Post**  
data inserted.

**Return**  
pointer to B-tree root.

1  taller = insertNode(root, data, upEntry)
2  if (taller true)

Tree has grown. Create new root.

1  allocate (newPtr)
2  newPtr -> entries[1] = upEntry
3  newPtr -> firstPtr = root
4  newPtr -> numEntries = 1
5  root = newPtr

3  return root

End  
BTreeInsert
B-Tree Insertion

**Algorithm**

`insertNode (val root <pointer>, val data <record>, ref upEntry <entry>)`

Recursively searches tree to locate leaf for data. If node overflow, inserts median key's data into parent.

- **Pre**
  - `root` is a pointer to tree or subtree. May be null.
- **Post**
  - `data` inserted.
  - `upEntry` is overflow entry to be inserted into parent.
- **Return**
  - `tree taller <boolean>`.

1. if (root null)
   1. `upEntry.data = data`
   2. `upEntry.rightPtr = null`
   3. `taller = true`
2. else
B-Tree Insertion

else

   entryNdx = searchNode (root, data.key)
   if (entryNdx > 0)
      subTree = root -> entries[entryNdx].rightPtr
   else
      subTree = root -> firstPtr

else

   taller = insertNode(subTree, data, upEntry)
   if (taller)
      if (node full)
         splitNode (root, entryNdx, upEntry)
         taller = true
      else
         insertEntry (root, entryNdx, upEntry)
         taller = false

   root -> numEntries = root -> numEntries + 1
return taller

End insertNode
B-Tree Insertion

**Algorithm**

```
searchNode (val nodePtr <pointer>, val target <key>)
```

Search B-tree node for data entry containing key <= target.

**Pre**
nodePtr is pointer to non-null node.
target is key to be located.

**Return**
index to entry with key <= target.
0 if key < first entry in node

1   if (target < nodePtr -> entry[1].data.key)
   1     walker = 0
2    else
1     walker = nodePtr -> numEntries
2     loop (target < nodePtr -> entries[walker].data.key)
1       walker = walker - 1
3    return walker

**End**

searchNode

Cao Hoang Tru
CSE Faculty - HCMUT
B-Tree Insertion

Algorithm  splitNode (val node <pointer>, val entryNdx <index>,
                   ref upEntry <entry>)

Node has overflowed. Split node. No duplicate keys allowed.

Pre  node is pointer to node that overflowed.
     entryNdx contains index location of parent.
     upEntry contains entry being inserted into split node.

Post  upEntry now contains entry to be inserted into parent.

1  minEntries = minimum number of entries
2  allocate (rightPtr)
   Build right subtree node
3  if (entryNdx <= minEntries)
   1  fromNdx = minEntries + 1
4  else
B-Tree Insertion

```c
4 else
   1 fromNdx = minEntries + 2
5 toNdx = 1
6 rightPtr -> numEntries = node -> numEntries - fromNdx + 1
7 loop (fromNdx <= node -> numEntries)
   1 rightPtr -> entries[toNdx] = node -> entries[fromNdx]
   2 fromNdx = fromNdx + 1
   3 toNdx = toNdx + 1
8 node -> numEntries = node -> numEntries - rightPtr -> numEntries
9 if (entryNdx <= minEntries)
   1 insertEntry (node, entryNdx, upEntry)
10 else
```

Cao Hoang Tru  
CSE Faculty - HCMUT  
17 November 2008
B-Tree Insertion

11 else
12   insertEntry (rightPtr, entryNdx − minEntries, upEntry)
13   node -> numEntries = node -> numEntries − 1
14   rightPtr -> numEntries = rightPtr -> numEntries + 1

Build entry for parent
12   medianNdx = minEntries + 1
13   upEntry.data = node -> entries[medianNdx].data
14   upEntry.rightPtr = rightPtr
15   rightPtr -> firstPtr = node -> entries[medianNdx]. rightPtr
16   return
End   splitNode
B-Tree Insertion

Algorithm insertEntry (val node <pointer>, val entryNdx <index>,
val newEntry <entry>)

Inserts one entry into a node by shifting nodes to make room.

Pre node is pointer to node to contain data.
newEntry contains data to be inserted.
entryNdx is index to location for new data.

Post data have been inserted in sequence.

1 shifter = node -> numEntries + 1
2 loop (shifter > entryNdx + 1)
   1 node -> entries[shifter] = node -> entries[shifter - 1]
   2 shifter = shifter - 1
3 node -> entries[shifter] = newEntry
4 node -> numEntries = node -> numEntries + 1
5 return
End insertEntry

Cao Hoang Tru
CSE Faculty - HCMUT
17 November 2008
B-Tree Deletion

- It must take place at a leaf node.
- If the data to be deleted are not in a leaf node, then replace that entry by the largest entry on its left subtree.
B-Tree Deletion

Delete 78

Delete 63
B-Tree Deletion

Delete 85

Underflow

(node has fewer than the min num of entries)

Delete 21
Reflow

• For each node to have sufficient number of entries:
  - **Balance**: shift data among nodes.
  - **Combine**: join data from nodes.
Balance

Borrow from right

Original node

Rotate parent
data down

Rotate data to
parent

Shift entries
left

Cao Hoang Tru
CSE Faculty - HCMUT

23 November 2008

CuuDuongThanCong.com
https://fb.com/tailieudientucntt
Balance

Borrow from left

Original node

Shift entries right

Rotate parent data down

Rotate data up

Cao Hoang Tru
CSE Faculty - HCMUT

17 November 2008
Combine

1. After underflow

2. After moving root to subtree

3. After moving right entries

4. After shifting root

Cao Hoang Tru
CSE Faculty - HCMUT

17 November 2008
B-Tree Traversal
B-Tree Traversal

Algorithm BTreeTraversal (val root <pointer>)

Processes tree using inorder traversal

Pre root is a pointer to B-tree
Post Every entry has been processed in order

1 scanCount = 0
2 ptr = root -> firstPtr
3 loop (scanCount <= root -> numEntries)
   1 if (ptr not null)
      1 BTreeTraversal (ptr)
      2 scanCount = scanCount + 1
     3 if (scanCount <= root -> numEntries)
        1 process (root -> entries[scanCount].data)
        2 ptr = root -> entries[scanCount].rightPtr
   4 return

End BTreeTraversal
B-Tree Search

Algorithm

BTreeSearch (val root <pointer>, val target <key>,
             ref node <pointer>, ref entryNo <index>)

Recursively searches a B-tree for the target key

Pre

root is a pointer to a tree or subtree
target is the data to be located

Post

if found --
    node is pointer to located node
    entryNo is entry within node
if not found --
    node is null and entryNo is zero

Return

found <boolean>
B-Tree Search

1. if (empty tree)
   1. node = null
   2. entryNo = 0
   3. found = false
2. else
   1. if (target < first entry)
      1. return BTreeSearch (root -> firstPtr, target, node, entryNo)
   2. else
      1. entryNo = root -> numEntries
      2. loop (target < root -> entries[entryNo].data.key)
         1. entryNo = entryNo - 1
      3. if (target = root -> entries[entryNo].data.key)
         1. found = true
         2. node = root
      4. else
         1. return BTreeSearch (root -> entries[entryNo].rightPtr, target, node, entryNo)
   4. return found

End BTreeTraversall
Cao Hoang Tru
CSE Faculty - HCMUT

17 November 2008
B-Tree Variations

- **B*Tree**: the minimum number of (used) entries is two thirds.

- **B+Tree**:
  - Each data entry must be represented at the leaf level.
  - Each leaf node has one additional pointer to move to the next leaf node.
Reading

• Pseudo code of algorithms for B-Tree Insertion