B-Trees

- Large degree B-trees used to represent very large dictionaries that reside on disk.

- Smaller degree B-trees used for internal-memory dictionaries to overcome cache-miss penalties.
B-Trees

\[
x \leftarrow \text{a pointer to some object}
\]

\textbf{DISK - READ}(x)

operations that access and/or modify the fields of \( x \)

\textbf{DISK - WRITE}(x)

others operations that access but do not modify the fields of \( x \)
AVL Trees

- $n = 2^{30} = 10^9$ (approx).
- $30 \leq \text{height} \leq 43$.
- When the AVL tree resides on a disk, up to $43$ disk access are made for a search.
- This takes up to (approx) 4 seconds.
- Not acceptable.
Red-Black Trees

- \( n = 2^{30} = 10^9 \) (approx).
- \( 30 \leq \text{height} \leq 60 \).
- When the red-black tree resides on a disk, up to 60 disk access are made for a search.
- This takes up to (approx) 6 seconds.
- Not acceptable.
A Disk Page

an AVL node

Useless content

A Search Tree Node
m-way Search Trees

- Each node has up to $m - 1$ pairs and $m$ children.
- $m = 2 \Rightarrow$ binary search tree.
4-Way Search Tree

k < 10

10 < k < 30

30 < k < 35

k > 35
Maximum # Of Pairs

• Happens when all internal nodes are $m$-nodes.
• Full degree $m$ tree.
• # of nodes $= 1 + m + m^2 + m^3 + \ldots + m^{h-1}$
  $= (m^h - 1)/(m - 1)$.
• Each node has $m - 1$ pairs.
• So, # of pairs $= m^h - 1$. 
## Capacity Of m-Way Search Tree

<table>
<thead>
<tr>
<th>h</th>
<th>m = 2</th>
<th>m = 200</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>$8 \times 10^6 - 1$</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
<td>$3.2 \times 10^{11} - 1$</td>
</tr>
<tr>
<td>7</td>
<td>127</td>
<td>$1.28 \times 10^{16} - 1$</td>
</tr>
</tbody>
</table>
Definition Of B-Tree

- Definition assumes external nodes (extended $m$-way search tree).

- B-tree of order $m$.
  - $m$-way search tree.
  - Not empty $\Rightarrow$ root has at least 2 children.
  - Remaining internal nodes (if any) have at least $\lceil m/2 \rceil$ children.
  - External (or failure) nodes on same level.
2-3 And 2-3-4 Trees

• B-tree of order $m$.
  ▪ $m$-way search tree.
  ▪ Not empty $\Rightarrow$ root has at least 2 children.
  ▪ Remaining internal nodes (if any) have at least $\lceil m/2 \rceil$ children.
  ▪ External (or failure) nodes on same level.

• 2-3 tree is B-tree of order 3.
• 2-3-4 tree is B-tree of order 4.
B-Trees Of Order 5 And 2

• B-tree of order $m$.
  ▪ $m$-way search tree.
  ▪ Not empty => root has at least 2 children.
  ▪ Remaining internal nodes (if any) have at least $\lceil m/2 \rceil$ children.
  ▪ External (or failure) nodes on same level.

• B-tree of order 5 is 3-4-5 tree (root may be 2-node though).
• B-tree of order 2 is full binary tree.
**Minimum # Of Pairs**

- \( n = \) # of pairs.
- # of external nodes = \( n + 1 \).
- Height = \( h \) => external nodes on level \( h + 1 \).

<table>
<thead>
<tr>
<th>Level</th>
<th># of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>( \geq 2 )</td>
</tr>
<tr>
<td>3</td>
<td>( \geq 2 \cdot \text{ceil}(m/2) )</td>
</tr>
<tr>
<td>( h + 1 )</td>
<td>( \geq 2 \cdot \text{ceil}(m/2)^{h-1} )</td>
</tr>
</tbody>
</table>

\( n + 1 \geq 2 \cdot \text{ceil}(m/2)^{h-1}, \ h \geq 1 \)
## Minimum # Of Pairs

\[ n + 1 \geq 2^{\text{ceiling}(m/2)^{h-1}}, \ h \geq 1 \]

- \( m = 200 \).

<table>
<thead>
<tr>
<th>height</th>
<th># of pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>( \geq 199 )</td>
</tr>
<tr>
<td>3</td>
<td>( \geq 19,999 )</td>
</tr>
<tr>
<td>4</td>
<td>( \geq 2 \times 10^6 - 1 )</td>
</tr>
<tr>
<td>5</td>
<td>( \geq 2 \times 10^8 - 1 )</td>
</tr>
</tbody>
</table>

\[ h \leq \log_{\text{ceiling}(m/2)} \left[ \frac{(n+1)}{2} \right] + 1 \]
Choice Of m

- Worst-case search time.
  - \((\text{time to fetch a node} + \text{time to search node}) \times \text{height}\)
• convention:
  - Root of the B-tree is always in main memory.
  - Any nodes that are passed as parameters must already have had a DISK_READ operation performed on them.

• Operations:
  - Searching a B-Tree.
  - Creating an empty B-tree.
  - Splitting a node in a B-tree.
  - Inserting a key into a B-tree.
  - Deleting a key from a B-tree.
Node Structure

\[ n \ c_0 \ k_1 \ c_1 \ k_2 \ c_2 \ldots \ k_n \ c_n \]

- \( c_i \) is a pointer to a subtree.
- \( k_i \) is a dictionary pair (KEY).
Search

BT_Search(x, k)

\[ i \leftarrow 0 \]

while \( i < n \) and \( k > k_{i+1}[x] \)

\[ \text{do } i \leftarrow i + 1 \]

if \( i < n \) and \( k = k_{i+1}[x] \)

then return \( (x, i + 1) \)

if \( \text{leaf}[x] \) then return NULL

else DISK-READ\((C_i[x])\)

return B-Tree-Search\((C_i[x], k)\)
• **B-Tree-Created(T) :**
  
  ▪ **Algorithm :**
  
  ```
  B-Tree-Create(T) 
  {
  x ← Allocate - Node() 
  Leaf[x] ← TRUE 
  n[x] ← 0 
  DISK - WRITE(x) 
  root[T] ← x 
  } 
  ```

  ▪ **time :** $O(1)$
Insertion into a full leaf triggers bottom-up node splitting pass.
Split An Overfull Node

$m \ c_0 \ k_1 \ c_1 \ k_2 \ c_2 \ldots \ k_m \ c_m$

- $c_i$ is a pointer to a subtree.
- $k_i$ is a dictionary pair (KEY).

$\text{ceil}(m/2)-1 \ c_0 \ k_1 \ c_1 \ k_2 \ c_2 \ldots \ k_{\text{ceil}(m/2)-1} \ c_{\text{ceil}(m/2)-1}$

$m-\text{ceil}(m/2) \ c_{\text{ceil}(m/2)} \ k_{\text{ceil}(m/2)+1} \ c_{\text{ceil}(m/2)+1} \ldots \ k_m \ c_m$

- $k_{\text{ceil}(m/2)}$ plus pointer to new node is inserted in parent.
• Insert a pair with key = 2.
• New pair goes into a 3-node.
**Insert Into A Leaf 3-node**

- Insert new pair so that the 3 keys are in ascending order.

  ![Diagram](1 2 3)

- Split overflowed node around middle key.

  ![Diagram](1 2 3)

- Insert middle key and pointer to new node into parent.

  ![Diagram](1 2 3)
• Insert a pair with key = 2.
Insert a pair with key = 2 plus a pointer into parent.
• Now, insert a pair with key = 18.
Insert Into A Leaf 3-node

- Insert new pair so that the 3 keys are in ascending order.

- Split the overflowed node.

- Insert middle key and pointer to new node into parent.
• Insert a pair with key = 18.
• Insert a pair with key = 17 plus a pointer into parent.
• Insert a pair with key = 17 plus a pointer into parent.
Now, insert a pair with key = 7.
Insert a pair with key = 6 plus a pointer into parent.
- Insert a pair with key = 4 plus a pointer into parent.
• Insert a pair with key = 8 plus a pointer into parent.
• There is no parent. So, create a new root.
• Height increases by 1.
• Btree::InsertNode(Key k, Element e)
{
    bool overflow = Insert(root, k, e);
    if (overflow)
        <Key, Node*> newpair = split(root);
        root = new Node(root, newpair);
    return;
}
• Bool Insert(node* x, Key k, Element e)
{
    if (leaf(x))
        insertLeaf(x, k, e);
        if (size(x) > m-1) return true;
        else return false;
    idx = keySearch(x, k);
    bool overflow = Insert(x->C[idx], k, e);
if (overflow)
    <Key, Node*> newpair = split(x->C[idx]);
InsertPair(x, newpair);
if(size(x) > m-1)
    return true;
else return false;
• **Exercises**: P609-3