Multi-Way Search Trees
(B Trees)
Multiway Search Trees

- An *m-way search tree* is a tree in which, for some integer *m* called the order of the tree, each node has at most *m* children.
- If *n* ≤ *m* is the number of children of a node, then it contains exactly *n*-1 keys, which partition all the keys into *n* subsets
  - consisting of all the keys less than the first key in the node, all the keys between a pair of keys in the node, and all keys greater than the largest key in the node.
Formal Definition

- A m-way search tree $T$ is a tree in which all nodes are of degree $\leq m$.
- If $T = \text{Null}$, then $T$ is a m-way search tree.
- If $T \neq \text{Null}$ then it should satisfy the properties listed on next slide.
Properties of m-way search tree

Properties:

i. T is a node of the type

\[ [n, T_0, (k_1, T_1), (k_2, T_2), \ldots, (k_{n-1}, T_{n-1}) ] \]

where \( T_i (0 \leq i < n) \) are pointers to the sub-trees of \( T \) and \( k_i (1 \leq i < n) \) are keys; and \( 1 \leq n < m \)

ii. \( k_i < k_{i+1} , (1 \leq i < n) \)

III. All the keys in sub-tree \( T_i < k_{i+1} , (0 \leq i < n-1) \)

IV. All the keys in sub-tree \( T_{n-1} > k_{n-1} \)

V. The sub-tree \( T_i , (0 \leq i < n) \) are also m-way search trees
Example M-way Search Tree
Example – 3-way search tree
Cont...

- In m-way search tree,
  - each node may have more than two child nodes
  - there is a specific ordering relationship among the nodes
- m-way search tree if it is unbalanced then the purpose is defeated.
- To achieve a better performance, it has to be balanced m-way search tree.
- Particular variety of balanced m-way search tree is called B-tree.
Motivation - (for B-Trees)

- Remember that performance is related to the height of the tree.
- We want to minimize the height of the tree.
- Used to process external records (information too large to put into memory), minimizes number of accesses to secondary peripheral.
- Index structures for large datasets that cannot be stored in main memory.
- Storing it on disk requires different approach to efficiency.
Motivation – contd…

- Assume that we use an AVL tree to store about 20 million records
- We end up with a very deep binary tree with lots of different disk accesses; $\log_2 20,000,000$ is about 24, so this takes about 0.2 seconds
- We know we can’t improve on the $(\log_2 n)$ lower bound on search for a binary tree.
- But, the solution is to use more branches and thus reduce the height of the tree!
  - As branching increases, depth decreases
When searching tables held on disc, the cost of each disc transfer is high but doesn't depend much on the amount of data transferred, especially if consecutive items are transferred

- A B-tree of order 101 and height 3 can hold $101^4 - 1$ items (approximately 100 million) and any item can be accessed with 3 disc reads (assuming we hold the root in memory)
Definition of a B-tree

- A B-tree of order $m$ is an $m$-way tree (i.e., a tree where each node may have up to $m$ children) in which:
  - all leaves are on the same level
  - all non-leaf nodes except the root have at least $\lceil m / 2 \rceil$ children
  - the root is either a leaf node, or it has from 2 to $m$ children
  - a leaf node contains no more than $m - 1$ keys
- The number $m$ should always be odd.
Example B tree
Analysis of B-Trees

- The maximum number of items in a B-tree of order $m$ and height $h$:
  
<table>
<thead>
<tr>
<th>Level</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>$m - 1$</td>
</tr>
<tr>
<td>Level 1</td>
<td>$m(m - 1)$</td>
</tr>
<tr>
<td>Level 2</td>
<td>$m^2(m - 1)$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Level h</td>
<td>$m^h(m - 1)$</td>
</tr>
</tbody>
</table>

- So, the total number of items is
  
  $$(1 + m + m^2 + m^3 + \ldots + m^h)(m - 1) = [\frac{m^{h+1} - 1}{(m - 1)}] (m - 1) = m^{h+1} - 1$$

- When $m = 5$ and $h = 2$ this gives $5^3 - 1 = 124$
An example B-Tree

A B-tree of order 5 containing 26 items

Note that all the leaves are at the same level
Inserting into a B-Tree

- Attempt to insert the new key into a leaf.
- If this results in leaf becoming too big, split the leaf into two, moving up the middle key to the parent of that leaf.
- If this again results in the parent becoming too big, split the parent into two, move the middle key up.
- This strategy might have to be repeated all the way to the top.
- If necessary, the root is split in two and the middle key is promoted to a new root, making the tree one level higher.
Construction of B-tree of order 3

Example:
Insert 40:

Insert 80:

Insert 10:
Temporarily placed in node, split and move middle value up

Insert 20:
Contd...

Insert 60:

```
  40
 /   \
10, 20  60, 80
```

Insert 30:
temporarily placed in 10,20 node, middle value goes up

```
  40
 /   \
10, 20, 30  60, 80
```

comes:

```
  20, 40
 /   \
10    30  60, 80
```
Insert 100:
temporarily placed in 60,80 node, middle value
goes up

becomes
Now root must split, and middle value goes up, root grows in height by 1:
Now insert 70...
Insert 70:
temporarily placed in 50, 60 node, middle value goes up

becomes:

Now insert 90…
90 Inserted
Constructing a B-tree of order 5 - Example

- Create B tree of order with the following sequence of keys:

1, 8, 2, 14, 12, 6, 14, 28, 17, 7, 52, 16, 48, 68, 3, 26, 29, 53, 55, 45,
Constructing a B-tree of order 5 - Example

6, 14, 28 get added to the leaf nodes:
Example – Cont…

Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf

7, 52, 16, 48 get added to the leaf nodes
Contd.

Adding 68 causes us to split the right most leaf, promoting 48 to the root, and adding 3 causes us to split the left most leaf, promoting 3 to the root; 26, 29, 53, 55 then go into the leaves

Adding 45 causes a split of and promoting 28 to the root then causes the root to split
Constructing a B-tree (contd.)
Deletion from a B-tree

- During insertion, the key always goes into a leaf. For deletion, there are three possible ways we can do this:
  1: If the key is already in a leaf node, and removing it doesn’t cause that leaf node to have too few keys, then simply remove the key to be deleted.
  2: If the key is not in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf
     - In this case we can delete the key and promote the predecessor or successor key to the non-leaf deleted key’s position.
Deletion – Contd..

If (1) or (2) leads to a leaf node containing less than the minimum number of keys then look at the siblings immediately adjacent to the leaf in question:

3: if one of them has more than the min. number of keys, then we can promote one of its keys to the parent and take the parent key into our lacking leaf.

4: if neither of them has more than the min. number of keys then the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key) and the new leaf will have the correct number of keys; if this step leave the parent with too few keys then we repeat the process up to the root itself, if required.
Original tree:

Delete 70…
Delete 70:
swap with inorder successor (80)
Delete value from leaf:

80, 90

60 - 100

Merge nodes

80, 90

60 - 100

Move 80 down

90

60, 80 100
Final Tree:

```
    50
   /\    /
 30 90  60 100
 /  \    /   /
10, 20 40 60, 80
```

Now delete 100…
Now, delete 100:
delete value from leaf

90

60, 80
-

now redistribute values:

80

60

90
Now delete 80…

New Tree:

```
          50
         /   \
        30    80
       /     /   \
      10, 20 60   90
     /    
    40
```
Now delete 80: swap with inorder successor (90)
Delete value from leaf, Merge and move 90 down:
Merge and move 50 down:
Remove empty root:

```
30, 50
/ |  \
10, 20 40 60, 90
```
Example: Simple leaf deletion

Assuming a B-Tree of order 5.

DELETE: 2, 52, 72, 22
Delete 2: Since there are enough keys in the node, just delete it
Simple non-leaf deletion

DELETE: 2, 52, 72, 22
Delete 52:

Note when printed: this slide is animated
Too few keys in node and its siblings

DELETE: 2, 52, 72, 22
Delete 72:

Note when printed: this slide is animated
Cont...Too few keys in node and its siblings

DELETE: 2, 52, 72, 22
Delete 22:

Note when printed: this slide is animated
Enough siblings

Delete 22:

Note when printed: this slide is animated
Cont.. Enough siblings

Note when printed: this slide is animated
2-3 Search Trees

- It is B-Tree of degree 3. It is balanced search tree.
- Either the tree T is empty or it has two children:
  - root contains 1 data item
  - Root value is greater than each value in left subtree
  - Root value is less than each value in right subtree
- T is of the form: root, left subtree, middle subtree, right subtree
  - Root has two data items
  - Smaller value in root is greater than everything in left subtree and smaller than everything in middle subtree
  - Larger value in root is greater than everything in middle subtree and smaller than everything in right subtree
Example

- Nodes with 2 children must have 1 item
- Nodes with 3 children must have 2 items
- Leaves may contain 1 or 2 items
Multiway Trees

- Height balanced binary tree (AVL) provides a way to search, insert and delete entries from a table of size \( n \) using almost \( O(\log n) \) time.
- The AVL tree would itself reside on a disk and if nodes are retrieved from the disk one at a time, then search of an index with \( n \) entries would require almost \( (1.4 \log n) \) disk accesses (max. depth of AVL tree).
- For million entries (indices), this means about 23 accesses in the worst case.
- We can do better than 23 accesses by using a balanced tree based on m-way search tree rather than one based on a binary search tree.

**M-way search tree** \( T \): It is a tree in which all nodes are of degree \( \leq m \). If \( T = \text{nil} \) then \( T \) is an m-way search tree. If \( T \neq \text{nil} \) then it has the following properties.
(i) \( T \) is a node of the type

\[ n, A_0, (K_1, A_1), (K_2, A_2), \ldots, (K_n, A_n) \]

where \( A_i \) \((0 \leq i \leq n)\) are pointers to the subtrees of \( T \) and \( k_i \) \((1 \leq i \leq n)\) are the key values, and \( 1 \leq n < m \)

(ii) \( k_i < k_{i+1}, 1 \leq i \leq n \)

(iii) All the keys in the subtree \( A_i \) are less than the key value \( k_{i+1} \), \((0 \leq i < n)\)

(iv) All the keys in subtree \( A_n \) are greater than \( k_n \)

(iv) The subtrees \( A_i \), \(0 \leq i \leq n\) are also \( m \)-way search trees.

\[ \text{Ex} \]

\[ \begin{array}{c}
\bullet 20 \quad 40 \\
\downarrow \quad \downarrow \\
10 \quad 15 \quad 25 \quad 30 \quad 45 \quad 50 \\
\downarrow \quad \downarrow \\
35 \\
\end{array} \]

- \( m \)-way search trees play the same role among \( m \)-way trees (general tree with order \( m \)) that the binary search trees play among binary trees if they are used for the same purpose.
- If \( T \) is unbalanced, then purpose will be defeated.
• To achieve a performance close to that of the best $m$-way search trees for a given number of entries $n$, it is necessary that the search tree be balanced.

• The particular variety of balanced $m$-way search tree is called B-tree.

• Let us define the concept of failure node. A failure node represents a node which can be reached during a search only if the value $x$ being searched for is not in the tree.

• Every subtree with root $x$ is a point that is reached during search iff $x$ is not in the tree.

• For convenience, these empty subtrees will be replaced by hypothetical nodes called failure nodes.

• These are represented by $\square$

![Diagram of a 3-way search tree]
Definition: A B-tree, $t$, of order $m$ is an $m$-way search tree that is either empty or is of height $\geq 1$ and satisfies the following properties:

(i) the root node has at least 2 children
(ii) all nodes other than the root node (and failure nodes also) have at least $\lceil m/2 \rceil$ children (pointers)
(iii) All failure nodes are at the same level.

![B-tree of order 3]

Since in B-tree of order 3, all non-failure nodes must be of degree 2 or 3, such trees are also known as 2-3 trees.
Insertion in B-Tree

- Here insertion is done at leaf node.
  There are three common situations

(i) A key is placed in a leaf that still has some place as shown below.

```
       12
   5  8  13  15
```

Insert 7

```
       12
   5  7  8  13  15
```

- Order of keys in a node is preserved.

(ii) The leaf node in which a key should be placed is full. In this case, the leaf is split, creating a new leaf, and half of the keys are moved from full leaf to the new leaf.
Now a new leaf has to be incorporated into a B-tree. The last key of the old leaf is moved to the parent and a pointer to the new leaf is placed in the parent. The same procedure can be repeated for each internal node of the B-tree.

Ex.

```
\begin{center}
\begin{tikzpicture}
  \node (root) {12}
  \node (left) [below left of=root] {2 5 7 8}
  \node (right) [below right of=root] {13 15}
  \draw (root) -- (left);
  \draw (root) -- (right);
\end{tikzpicture}
\end{center}
```

Insert: 6

```
\begin{center}
\begin{tikzpicture}
  \node (root) {12}
  \node (left) [below left of=root] {2 5 6}
  \node (middle) [below right of=root] {7 8}
  \node (right) [below right of=middle] {13 15}
  \draw (root) -- (left);
  \draw (root) -- (middle);
  \draw (middle) -- (right);
\end{tikzpicture}
\end{center}
```

- Such split guarantees that each leaf never has less than \( \lceil m/2 \rceil - 1 \) keys.
(iii) A special case arises if the root of B-tree is full. In this case, a new root and a new sibling of the existing root have to be created. This split results in two new nodes in the B-tree.

**Ex:** After inserting the key 13 in the 3rd leaf in Fig 1(a), the leaf is split (as in case (ii)) a new leaf is created and the key 15 is about to be moved to the parent. But here parent has no place (Fig 1(b)). The parent is split (Fig 1c), but now two B-trees have to be combined into one. This is achieved by creating a new root and moving the last key from the old root to it (Fig 1d).

- In this case height is increased by one.
- Fig 2 shows the growth of a B-tree of orders in the course of inserting new keys.
FIGURE 1. Inserting the number 13 into a full leaf.

(a) 

Insert 13

(b) 

Move

(c) 

(d)
Building a B-tree of order 5

Insert 8, 14, 2, 15

(a)

Insert 3

(b)

Insert 1, 16, 6, 5

(c)

Insert 27, 37

(d)

Insert 18, 25, 7, 13, 20

(e)

Insert 22, 23, 24

(f)
Deletion of key from B-Tree

- It is to great extent a reversal of insertion.
- There are two main cases:

  - Deletion from a leaf node
  - Deletion from a non-leaf node

Deletion from a leaf:

1. If after deleting a key \( k \), the number of keys left in the leaf are at least half \( \lceil n/2 \rceil - 1 \), then do nothing (see Fig 3.4-b).

2. If after deletion, the number of keys in the leaf are less than \( \lceil n/2 \rceil - 1 \), then underflow is caused.
   - If \( l \) left or right sibling with the number of keys exceeding the minimal \( \lceil n/2 \rceil - 1 \), then all keys from this leaf and selected left or right sibling are redistributed between them by moving the separator key from the parent to the leaf and moving one key from sibling to the parent (Fig 3.6-c).
• If both the siblings have \( \frac{m}{2} \) keys then the leaf (from which deletion is done) and right sibling are merged along with separating key from the parent (Fig 3.c-d) and all are put in the leaf. This may initiate a chain of operations of the parent underflows. The process is repeated (Fig 3.e).

In Fig 3.e, we see that a new root is created and two nodes disappear at one time.

**Deletion from non-leaf node.**

This may lead to problems with tree reorganization. So deletion from non-leaf node is reduced to deleting a key from a leaf as follows:

• The key to be deleted from non-leaf node is replaced by its immediate successor (or predecessor) which can only be found in leaf node. (Fig 3.e-f)
Deleting keys from a B-tree.

(a) Delete 6

(b) Delete 7

(c) Select 16

(d) Underflow

(e) Delete 8 cont.

(f) Delete 16

Replace by 15
• B-tree are guaranteed to be at least 50% full, so it may happen that 50% of space is basically wasted.

• Analyses and simulations however indicate that after a series of numerous random insertions and deletions, B-tree is approximately 69% full.

• It is very unlikely that B-tree will ever be filled to the full.

• B*-tree is a variant of B-tree introduced by D. Knuth.

• In B*-tree, all nodes except the root are required to be at least 2/3 full not just 1/2 in B-tree.

• The frequency of node splitting is decreased by delaying a split.

• The average utilization of B*-tree is 81%.

• A split in B*-tree is delayed by attempting to redistribute the keys between a node and its sibling when the node overflows.
$B^*$-tree of order 9.

Insert 6: Instead of splitting leaf node all the keys from this node, separating key from parent and keys from its sibling are evenly distributed in both the nodes. Median key is put into the parent.

0, 1, 2, 5, 6, 7, 9, 10, 12, 16, 18, 25, 27, 28, 30
If the sibling is full, a split occurs:
One new node is created, the keys from
the node and its sibling along with the
separating key from parent are evenly
divided among three nodes, and two
separating keys are put in the parent.

```
0 1 2 5 6 7 8 9
```
```
12 16 18 25 27 28 29 30
```

```
0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 12, 16, 18, 25, 27, 28, 29, 30
```

```
6 16
```
```
0 1 2 4 5...
7 8 9 10 12...
18 25 27 28 29 30
```

All three nodes participating in the split
are guaranteed to be 2/3 full.

```
Inser y
```
```
0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 12, 16, 18, 25, 27, 28, 29, 30
```