Definitions

- **Tree**: a non-empty collection of vertices & edges
- **Vertex (node)**: can have a name and carry other associated information
- **Path**: list of distinct vertices in which successive vertices are connected by edges

Trees

- Definitions
- Representation
- Binary trees
- Traversals
- Expression trees

Definitions

- Any two vertices must have one and only one path between them. Else, it is not a tree
- A tree with N nodes has N-1 edges
Definitions

- **Root**: starting point (top) of the tree
- **Parent (ancestor)**: the vertex “above” this vertex
- **Child (descendent)**: the vertices “below” this vertex

- **Leaves (terminal nodes)**: have no children
- **Level**: the number of edges between this node and the root
- **Ordered tree**: where children’s order is significant

Definitions

- **Depth of a node**: the length of the path from the root to that node
  - root: depth 0
- **Height of a node**: the length of the longest path from that node to a leaf
  - any leaf: height 0
- **Height of a tree**: The length of the longest path from the root to a leaf

Balanced Trees

- The difference between the height of the left sub-tree and the height of the right sub-tree is not more than 1
Trees - Example

Level
0

1
A  E  R
  Child (of root)

2
A  M  P  L  E
  Leaves or terminal nodes

3
Depth of T: 2
Height of T: 1

Example

Tree Representation

class TreeNode
{
  object    element;
  TreeNode  firstChild;
  TreeNode  nextSibling;
}

Binary Tree

Example
## Height of a Complete Binary Tree

At each level the number of the nodes is doubled.

Total number of nodes:

\[ 1 + 2 + 2^2 + 2^3 = 2^4 - 1 = 15 \]

## Nodes and Levels in a Complete Binary Tree

Number of the nodes in a tree with \( M \) levels:

\[ 1 + 2 + 2^2 + \ldots + 2^M = 2^{M+1} - 1 = 2*2^M - 1 \]

Let \( N \) be the number of the nodes.

\[ N = 2*2^M - 1, \quad 2*2^M = N + 1 \]

\[ 2^M = (N+1)/2 \]

\[ M = \log((N+1)/2) \]

\( N \) nodes: \( \log((N+1)/2) = O(\log(N)) \) levels

\( M \) levels: \( 2^{(M+1)} - 1 = O(2^M) \) nodes

## Binary Tree Node

```java
class BinaryNode {
    object Element; // the data in the node
    BinaryNode left; // Left child
    BinaryNode right; // Right child
}
```

## Binary Tree - Preorder Traversal

- **Root**
- **Left**
- **Right**

First letter – at the root  
Last letter – at the rightmost node
**Preorder Algorithm**

preorderVisit(tree)
{
    if (current != null) {
        process(current);
        preorderVisit(left_tree);
        preorderVisit(right_tree);
    }
}

**Binary Tree – Inorder Traversal**

**Inorder Algorithm**

inorderVisit(tree)
{
    if (current != null) {
        inorderVisit(left_tree);
        process(current);
        inorderVisit(right_tree);
    }
}

**Binary Tree – Postorder Traversal**

First letter - at the leftmost node
Last letter – at the root
**Postorder Algorithm**

```java
postorderVisit(tree)
{
    if (current != null) {
        postorderVisit(left_tree);
        postorderVisit(right_tree);
        process(current);
    }
}
```

**Expression Trees**

The stack contains references to tree nodes (bottom is to the left)

Post-fix notation: 1 2 + 3 *

(1+2)*3

In-order traversal:

\[(1 + 2) \times (3)\]

Post-order traversal:

\[1 \ 2 + 3 *\]
Binary Search Trees

- Definitions
- Operations and complexity
- Advantages and disadvantages
- AVL Trees
  - Single rotation
  - Double rotation
- Splay Trees
- Multi-Way Search

Definitions

- Each node contains:
  - a record with a key and a value
  - a left link
  - a right link
- All records with smaller keys
  - in the left subtree
- All records with larger keys
  - in the right subtree

Example

Operations

- Search: Compare the values and proceed either to the left or to the right
- Insertion: Unsuccessful search - insert the new node at the bottom where the search has stopped
- Deletion: Replace the value in the node with the smallest value in the right subtree or the largest value in the left subtree
- Retrieval in sorted order: Inorder traversal
**Complexity**

- Logarithmic, depends on the shape of the tree
- In the worst case – $O(N)$ comparisons

**Advantages of BST**

- Simple
- Efficient
- Dynamic

- One of the most fundamental algorithms in CS
- The method of choice in many applications

**Disadvantages of BST**

- The shape of the tree depends on the order of insertions, and it can be degenerated
- When inserting or searching for an element, the key of each visited node has to be compared with the key of the element to be inserted/found
  - Keys may be long and the run time may increase much

**Improvements on BST**

- Keeping the tree balanced:
  - AVL trees (Adelson - Velskii and Landis)
    - Balance condition: left and right subtrees of each node can differ by at most one level
    - It can be proven that if this condition is observed, the depth of the tree is $O(\log N)$

- Reducing the time for key comparison:
  - Radix trees: comparing only the leading bits of the keys
Radix Search Trees

• Radix Searching
• Digital Search Trees
• Radix Search Trees
• Multi-Way Radix Trees

Radix Searching

Idea: Examine the search keys one bit at a time

Advantages:
• Reasonable worst-case performance
• Easy way to handle variable length keys
• Some savings in space by storing part of the key within the search structure
• Competitive with both binary search trees and hashing

Disadvantages:
• Biased data can lead to degenerate trees with bad performance
• For some methods, use of space is inefficient
• Dependent on computer’s architecture - difficult to do efficient implementations in some high-level languages
Radix Searching

- Methods
  - Digital Search Trees
  - Radix Search Trees
  - Multiway Radix Searching

Digital Search Trees

- Similar to binary tree search
- Difference:
  - Branch in the tree by comparing the key’s bits, not the key as a whole

Example

<table>
<thead>
<tr>
<th>A</th>
<th>00001</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>10011</td>
</tr>
<tr>
<td>E</td>
<td>00101</td>
</tr>
<tr>
<td>R</td>
<td>10010</td>
</tr>
<tr>
<td>C</td>
<td>00011</td>
</tr>
<tr>
<td>H</td>
<td>01000</td>
</tr>
<tr>
<td>I</td>
<td>01001</td>
</tr>
<tr>
<td>N</td>
<td>01110</td>
</tr>
<tr>
<td>G</td>
<td>00111</td>
</tr>
<tr>
<td>X</td>
<td>11000</td>
</tr>
<tr>
<td>M</td>
<td>01101</td>
</tr>
<tr>
<td>P</td>
<td>10000</td>
</tr>
<tr>
<td>L</td>
<td>01100</td>
</tr>
</tbody>
</table>

 inserting Z = 11010
 go right twice
 go left – external node
 attach Z to the left of X

Example

 inserting Z = 11010
 go right twice
 go left – external node
 attach Z to the left of X

A | 00001 |
S | 10011 |
E | 00101 |
R | 10010 |
C | 00011 |
H | 01000 |
I | 01001 |
N | 01110 |
G | 00111 |
X | 11000 |
M | 01101 |
P | 10000 |
L | 01100 |
Digital Search Trees

- Things to remember about digital search trees:
  - Equal keys to be avoided: Must be kept in separate data structures, linked to the nodes
  - Worst case: Better than for binary search trees - the length of the longest path is equal to the longest match in the leading bits between any two keys

Radix Search Trees

- If the keys are long, digital search trees have low efficiency
- Radix search trees: Do not store keys in the tree at all, the keys are in the external nodes of the tree
- Called tries from "retrieval"

Digital Search Trees

- Search or insertion requires about $\log(N)$ comparisons on the average and $b$ comparisons in the worst case in a tree built from $N$ random $b$-bit keys
- No path will ever be longer than the number of bits in the keys

Radix Search Trees

- Two types of nodes:
  - Internal: contain only links to other nodes
  - External: contain keys and no links
Radix Search Trees

To insert a key:
1. Go along the path described by the leading bit pattern of the key until an external node is reached.
2. If the external node is empty, store there the new key.
   If the external node contains a key, replace it by an internal node linked to the new key and the old key. If the keys have several bits equal, more internal nodes are necessary.

**NOTE:** Insertion does not depend on the order of the keys.

---

Example

<table>
<thead>
<tr>
<th>Key</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00001</td>
</tr>
<tr>
<td>S</td>
<td>10011</td>
</tr>
<tr>
<td>E</td>
<td>00101</td>
</tr>
<tr>
<td>R</td>
<td>10010</td>
</tr>
<tr>
<td>C</td>
<td>00011</td>
</tr>
</tbody>
</table>

---

Example: Insertion

<table>
<thead>
<tr>
<th>Key</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00001</td>
</tr>
<tr>
<td>S</td>
<td>10011</td>
</tr>
<tr>
<td>E</td>
<td>00101</td>
</tr>
<tr>
<td>R</td>
<td>10010</td>
</tr>
<tr>
<td>C</td>
<td>00011</td>
</tr>
<tr>
<td>H</td>
<td>01000</td>
</tr>
</tbody>
</table>

---

Radix Search Trees

To search for a key:
1. Branch according to its bits
2. Don’t compare it to anything, until we get to an external node
3. One full key comparison there completes the search

---

External node - empty
**Example: Insertion**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00001</td>
</tr>
<tr>
<td>S</td>
<td>10011</td>
</tr>
<tr>
<td>E</td>
<td>00101</td>
</tr>
<tr>
<td>R</td>
<td>10010</td>
</tr>
<tr>
<td>C</td>
<td>00011</td>
</tr>
<tr>
<td>H</td>
<td>01000</td>
</tr>
<tr>
<td>I</td>
<td>01001</td>
</tr>
</tbody>
</table>

**Radix Search Trees - summary**

- **Program implementation:**
  - Necessity to maintain two types of nodes
  - Low-level implementation
- **Complexity:** About \( \log N \) bit comparisons in average case and \( b \) bit comparisons in the worst case in a tree built from \( N \) random \( b \)-bit keys.
- **Annoying feature:** One-way branching for keys with a large number of common leading bits:
  - The number of the nodes may exceed the number of the keys.
  - On average: \( N/\ln 2 = 1.44N \) nodes

**Multi-Way Radix Trees**

- The height of the tree is limited by the number of the bits in the keys
- If we have larger keys - the height increases. One way to overcome this deficiency is using a multi-way radix tree searching
- The branching is not according to 1 bit, but rather according to several bits (most often 2)
- If \( m \) bits are examined at a time - the search is speeded up by a factor of \( 2^m \)
- Problem: If \( m \) bits at a time, the nodes will have \( 2^m \) links, may result in considerable amount of wasted space due to unused links

**Example**

- **Search** – take left, right or middle links according to the first two bits
- **Insert** – replace external node by the key (e.g. insert T 10100)
Multi-Way Radix Trees

- Wasted space – due to the large number of unused links
- Worse if \( M \), the number of bits considered, gets higher
- The running time: \( \log_M N \) – very efficient
- Hybrid method:
  - Large \( M \) at the top,
  - Small \( M \) at the bottom

Recursive Size of Binary Tree

Recursive view used to calculate the size of a tree:
\[
S_T = S_L + S_R + 1
\]

Recursive Height of Tree

Recursive view of the node height calculation:
\[
H_T = \max (H_L + 1, H_R + 1)
\]
Balanced Trees

• A balanced tree is one where no node has two subtrees that differ in height by more than 1
  - visually, balanced trees look wider and flatter

AVL Trees

• Observation: BSTs that are balanced will have \( \log n \) search time for contains.
  - It is desirable to have a BST that stays balanced
• AVL tree: A binary search tree that uses modified add and remove operations to stay balanced as items are added into it
  - Invented in 1960 by two Russian mathematicians:
    • Georgii Maksimovich Adelson-Velskii
    • Evgenii Mikhailovich Landis
  - One of several auto-balancing trees

Tree Balance: Probabilities

• Binary search trees that can result from adding a random permutation of 1, 2, and 3:
  - Which is most likely to occur, given random input?
  - Which input orderings are "bad" or "good"?

AVL Trees: Formal Definition

• Balance factor, for a tree node \( n \):
  - Height of \( n \)'s right subtree minus height of \( n \)'s left subtree
    \[ BF_n = \text{Height}_{\text{right}} - \text{Height}_{\text{left}} \]
  - Reminder: Height of empty tree is -1
• A binary search tree is an AVL tree if:
  - balance factor of each node is 0, 1, or -1
    (if no node's two child subtrees differ in height by more than 1)
AVL Tree Examples

Two binary search trees:
(a) an AVL tree
(b) not an AVL tree (unbalanced nodes are darkened)

More AVL Tree Examples

Which are AVL Trees?

Not AVL Tree Examples
Problem Cases for AVL add
1. Insertion into left subtree of node’s left child
2. Insertion into right subtree of node’s left child

AVL Tree Data Structure
• Potential balancing problems occur when a new element is added or removed
• Maintain balance using rotations
  - The idea: reorganize the nodes of an unbalanced subtree until they are balanced, by "rotating" a trio of parent-leftChild-rightChild
  - Tree will maintain its balance so that searches (contains) will take $O(\log n)$

Insertion Problem Cases (cont’d)
3. Insertion into left subtree of node’s right child
4. Insertion into right subtree of node’s right child

Right Rotation to Fix Case 1
• Right rotation (clockwise): left child becomes parent; original parent demoted to right

(a) Before rotation
(b) After rotation
Right Rotation Steps

1. Detach left child (7)'s right subtree (10) (don't lose it!)
2. Consider left child (7) be the new parent
3. Attach old parent (13) onto right of new parent (7)
4. Attach old left child (7)'s old right subtree (10) as left subtree of new right child (13)

Right Rotation Example

Initial tree  |  After insertion  |  Right Rotation
--- | --- | ---
7 (-1)  |  7 (-2)  |  5 (0)
5 (0)  |  9 (0)  |  3 (-1)
3 (-1)  |  6 (0)  |  1 (0)
...  |  ...  |  ...

New node

Code for Right Rotation

```java
private TreeNode rightRotate(TreeNode parent) {
    // 1. detach left child's right subtree
    TreeNode leftright = parent.left.right;

    // 2. consider left child to be the new parent
    TreeNode newParent = parent.left;

    // 3. attach old parent onto right of new parent
    newParent.right = parent;

    // 4. attach old left child's old right subtree as left subtree of new right child
    newParent.right.left = leftright;

    return newParent;
}
```
Left Rotation to Fix Case 4

- **Left rotation** (counter-clockwise): right child becomes parent; original parent demoted to left

![Diagram](image)

(a) After rotation  
(b) Before rotation

Left Rotation Steps

1. Detach right child (70)’s left subtree (60) *(don’t lose it!)*
2. Consider right child (70) be the new parent
3. Attach old parent (50) onto left of new parent (70)
4. Attach old right child (70)’s old left subtree (60) as right subtree of new left child (50)

Code for Left Rotation

```java
private TreeNode leftRotate(TreeNode parent) {
    // 1. detach right child's left subtree
    TreeNode rightleft = parent.right.left;

    // 2. consider right child to be the new parent
    TreeNode newParent = parent.right;

    // 3. attach old parent onto left of new parent
    newParent.left = parent;

    // 4. attach old right child's old left subtree as right subtree of new left child
    newParent.left.right = rightleft;

    return newParent;
}
```

Problem: Cases 2, 3

- A single right rotation does not fix Case 2
- A single left rotation also does not fix Case 3

![Diagram](image)

(a) Before rotation  
(b) After rotation
### Left-Right Rotation for Case 2

- **Left-right double rotation**: A left rotation of the left child, followed by a right rotation at the parent.

![Diagram of Left-Right Rotation for Case 2](image)

(a) Before rotation  
(b) After rotation

### Left-Right Rotation Steps

1. Perform left-rotate on left child
2. Perform right-rotate on parent (current node)

![Diagram of Left-Right Rotation Steps](image)

### Left-Right Rotation Example

![Diagram of Left-Right Rotation Example](image)

(a) Before rotation  
(b) After rotation

### Right-Left Rotation for Case 3

- **Right-left double rotation**: A right rotation of the right child, followed by a left rotation at the parent.

![Diagram of Right-Left Rotation for Case 3](image)

(a) Before rotation  
(b) After rotation
**Right-left Rotation Steps**

1. Perform right-rotate on right child
2. Perform left-rotate on parent (current node)

**Implementing AVL add**

- After normal BST add, update heights from new leaf up towards root
  - If balance factor changes to $>+1$ or $<-1$, then use rotation(s) to rebalance
- Let $n$ be the first unbalanced node found
  - **Case 1:** $n$ has balance factor $-2$ and $n$'s left child has balance factor of $-1$
    - fixed by performing right-rotation on $n$
  - **Case 2:** $n$ has balance factor $-2$ and $n$'s left child has balance factor of $1$
    - fixed by perform left-rotation on $n$'s left child, then right-rotation on $n$ (left-right double rotation)

**AVL add (cont'd)**

- **Case 3:** $n$ has balance factor 2 and $n$'s right child has balance factor of $-1$
  - fixed by perform right-rotation on $n$'s right child, then left-rotation on $n$ (right-left double rotation)
- **Case 4:** $n$ has balance factor 2 and $n$'s right child has balance factor of 1
  - fixed by performing left-rotation on $n$
- After rebalancing, continue up the tree updating heights
  - What if $n$'s child has balance factor 0?
  - What if another imbalance occurs higher up?

**Code for AVL add**

```java
protected TreeNode add(TreeNode node, E element) {
    node = super.add(node, element);
    node = rebalance(node);
    return node;
}

protected TreeNode rebalance(TreeNode node) {
    int bf = balanceFactor(node);
    if (bf < -1) {
        if (balanceFactor(node.left) < 0) { // case 1 (R)
            node = rightRotate(node);
        } else { // case 2 (LR)
            node.left = leftRotate(node.left);
            node = rightRotate(node);
        }
    } else if (bf > 1) {
        if (balanceFactor(node.right) < 0) { // case 3 (RL)
            node = rightRotate(node.right);
            node = leftRotate(node);
        } else { // case 4 (L)
            node = leftRotate(node);
        }
    }
    return node;
}
```
Problems for AVL remove

- Removal from AVL tree can also unbalance the tree

AVL remove, cont'd

1. Perform normal BST remove (with replacement of node to be removed with its successor)
2. Update heights from successor node location upwards towards root
   - If balance factor changes to +2 or -2, then use rotation(s) to rebalance
• remove has the same 4 cases (and fixes) as insert
   - Are there any additional cases?
• After rebalancing, continue up the tree updating heights; must continue checking for imbalances in balance factor, and rebalancing if necessary

Right-Left Rotation on remove

SPLAY TREES
Problems with BSTs

- Because the shape of a BST is determined by the order that data is inserted, we run the risk of trees that are essentially lists.

Splay Tree Sequence of Operations

- Splay trees guarantee that a sequence of $M$ operations takes at most $O(M \times \log N)$ time.
- We say that the splay tree has amortized running time of $O(\log N)$ cost per operation. Over a long sequence of operations, some may take more than $\log N$ time, some will take less.

BST Sequence of Operations

- Worst case for a single BST operation can be $O(N)$.
- Not so bad if this happens only occasionally.
- BUT... It is not uncommon for an entire sequence of “bad” operations to occur. In this case, a sequence of $M$ operations take $O(M \times N)$ time and the time for the sequence of operations becomes noticeable.

Splay Tree Sequence of Operations (cont’d)

- Does not preclude the possibility that any particular operation is still $O(N)$ in the worst case.
  - Therefore, amortized $O(\log N)$ not as good as worst case $O(\log N)$.
- If any particular operation is $O(N)$ and we still want amortized $O(\log N)$ performance, then whenever a node is accessed, it must be moved. Otherwise, its access time is always $O(N)$.
Splay Trees

- The basic idea of the splay tree is that every time a node is accessed, it is pushed to the root by a series of tree rotations. This series of tree rotations is known as “splaying”.
- If the node being “splayed” is deep, many nodes on the path to that node are also deep and by restructuring the tree, we make access to all of those nodes cheaper in the future.

Splay Operation

- To “splay node x”, traverse up the tree from node x to root, rotating along the way until x is the root. For each rotation:
  - If x is root, do nothing.
  - If x has no grandparent, rotate x about its parent.
  - If x has a grandparent,
    - If x and its parent are both left children or both right children, rotate the parent about the grandparent, then rotate x about its parent.
    - If x and its parent are opposite type children (one left and the other right), rotate x about its parent, then rotate x about its new parent (former grandparent).
Node and Parent are Same Side
Zig-Zig

Node and Parent are Different Sides
Zig-Zag

Operations in Splay Trees

- **insert**
  - First insert as in normal binary search tree
  - Then, splay inserted node
  - If there is a duplicate, the node holding the duplicate element is splayed

- **find/contains**
  - Search for node
  - If found, splay it; otherwise, splay last node accessed on the search path

Operations on Splay Trees (cont)

- **remove**
  - Splay element to be removed
    - If the element to be deleted is not in the tree, the node last visited on the search path is splayed
  - Disconnect left and right subtrees from root
  - Do one or both of:
    - Splay max item in $T_L$ (then $T_L$ has no right child)
    - Splay min item in $T_R$ (then $T_R$ has no left child)
  - Connect other subtree to empty child of root
Insertion in order into a Splay Tree

- In a BST, building a tree from N sorted elements was $O(N^2)$. What is the performance of building a splay tree from N sorted elements?
Splay Tree Code

- The splaying operation is performed “up the tree” from the node to the root.
- How do we traverse “up” the tree?
- How do we know if X and P are both left/right children or are different children?
- How do we know if X has a grandparent?
- What disadvantages are there to this technique?

Top-Down Splay Trees

- Rather than write code that traverses both up and down the tree, “top-down” splay trees only traverse down the tree. On the way down, rotations are performed and the tree is split into three parts depending on the access path (zig, zig-zig, zig-zag) taken
  - X, the node currently being accessed
  - Left - all nodes less than X
  - Right - all nodes greater than X
- As we traverse down the tree, X, Left, and Right are reassembled
- This method is faster in practice, uses only $O(1)$ extra space and still retains $O(\log N)$ amortized running time

Large Trees

- Tailored toward applications where tree doesn’t fit in memory
  - CPU operations much faster than disk accesses
  - Want to limit levels of tree (because each new level requires a disk access)
  - Keep root and top level in memory
An Alternative to BSTs

- Up until now we assumed that each node in a BST stored the data (except for radix trees)
- What about having the data stored only in the leaves? The internal nodes just guide our search to the leaf which contains the data we want
- We’ll restrict this discussion of such trees to those in which all leaves are at the same level

Observations

- Store data only at leaves; all leaves at same level
  - Interior and exterior nodes have different structure
  - Interior nodes store one key and two subtree pointers
  - All search paths have same length: \([\log n]\) (assuming one element per leaf)
  - Can store multiple data elements in a leaf

M-Way Trees

- A generalization of the previous BST model
  - Each interior node has \(M\) subtrees pointers and \(M-1\) keys
    - the previous BST would be called a “2-way tree” or “M-way tree of order 2”
  - As \(M\) increases, height decreases: \([\log M n]\) (assuming one element per leaf)
  - A perfect \(M\)-way tree of height \(h\) has \(M^h\) leaves
An M-Way Tree of Order 3

- Figure 2 (next page) shows the same data as Figure 1, stored in an M-way tree of order 3. In this example $M=3$ and $h=2$, so the tree can support 9 leaves, although it contains only 8.

- One way to look at the reduced path length with increasing $M$ is that the number of nodes to be visited in searching for a leaf is smaller for large $M$.

- We’ll see that when data is stored on the disk, each node visited requires a disk access, so reducing the nodes visited is essential.

Searching in an M-Way Tree

- Different from standard BST search
  - Search always terminates at a leaf node
  - Might need to scan more than one element at a leaf
  - Might need to scan more than one key at an interior node

- Trade-offs
  - Tree height decreases as $M$ increases
  - Computation at each node during search increases as $M$ increases

```
Search (MWayNode v, DataType element, boolean foundIt) {
  if (v == NULL) foundIt=false;
  if (v is a leaf)
    search the list of values looking for element
    if found, foundIt=succeed
    otherwise foundIt=failure
  else (if v is an interior node)
    search the keys to find the subtree the element is in
    recursively search the subtree
}
```
Search Algorithm: Traversing the M-way Tree

In any interior node, find the first key > search item, and traverse the link to the left of that key. Search for any item >= the last key in the subtree pointed to by the rightmost link. Continue until search reaches a leaf.

Is It Worth It?

- Is it worthwhile to reduce the height of the search tree by letting $M$ increase?
- Although the number of nodes visited decreases, the amount of computation at each node increases.
- Where is the payoff?

Searching in an M-way Tree of Order 4

An Example

- Consider storing $10^7$ items in a balanced BST and in an M-way tree of order 10.
- The height of the BST will be $\log(10^7) \sim 24$.
- The height of the M-Way tree will be $\log(10^7) = 7$ (assuming that we store just 1 record per leaf).
- However, in the BST, just one comparison will be done at each interior node, but in the M-Way tree, 9 will be done (worst case).
How can This Be Worth the Price?

• Only if it somehow takes longer to descend the tree than it does to do the extra computation
• This is exactly the situation when the nodes are stored externally (e.g., on disk)
• Compared to disk access time, the time for extra computation is insignificant
• We can reduce the number of accesses by sizing the M-way tree to match the disk block and record size.

A Generic M-Way Tree Node

```java
public class MWayNode {
    // code for public interface here
    // constructors, accessors, mutators
    private boolean isLeaf;
    private int m;
    private int mKeys;
    private ArrayList<Ktype> keys;
    private MWayNode subtrees[];
    private int nElems;
    private List<Dtype> data;
}
```

B-Tree Definition

A B-Tree of order M is an M-Way tree with the following constraints

1. The root is either a leaf or has between 2 and M subtrees
2. All interior nodes (except maybe the root) have between \([M/2]\) and M subtrees (i.e., each interior node is at least “half full”)
3. All leaves are at the same level. A leaf must store between \([L/2]\) and L data elements, where L is a fixed constant \(\geq 1\) (i.e., each leaf is at least “half full”, except when the tree has fewer than \(L/2\) elements)

A B-Tree Example

• The following figure (Figure 3) shows a B-Tree with \(M=4\) and \(L=3\)
• The root node can have between 2 and \(M=4\) subtrees
• Each other interior node can have between \([M/2] = [4/2] = 2\) and \(M=4\) subtrees and up to \(M-1=3\) keys.
• Each exterior node (leaf) can hold between \([L/2] = [3/2] = 2\) and \(L=3\) data elements
A B-Tree with \( M=4 \) and \( L=3 \)

Designing a B-Tree

- Recall that \( M \)-way trees (and therefore B-trees) are often used when there are too much data to fit in memory. Therefore, each node or leaf access costs one disk access.

- When designing a B-Tree (choosing the values of \( M \) and \( L \)), we need to consider the size of the data stored in the leaves, the size of the keys and pointers stored in the interior nodes, and the size of a disk block.

Student Record Example

- Suppose our B-Tree stores student records which contain name, address, etc., and other data totaling 1024 bytes.

- Further, assume that the key to each student record (ID) is 8 bytes long.

- Assume also that a pointer (really a disk block number, not a memory address) requires 4 bytes.

- And finally, assume that our disk block is 4096 bytes.

Calculating \( L \)

- \( L \) is the number of data records that can be stored in each leaf. Since we want to do just one disk access per leaf, this is the same as the number of data records per disk block.

- Since a disk block is 4096 and a data record is 1024, we choose \( L = \left\lfloor \frac{4096}{1024} \right\rfloor = 4 \) data records per leaf.
**Calculating M**

- Each interior node contains $M$ pointers and $M-1$ keys. To maximize $M$ (and therefore keep the tree flat and wide) and yet do just one disk access per interior node, we have the following relationship:
  
  
  $4M + 8(M-1) \leq 4096$
  
  $12M \leq 4104$
  
  $M \leq 342$

- So, choose the largest possible $M$ (making tree as shallow as possible) of 342

**Performance of Our B-Tree**

- With $M=342$, the height of our tree for $N$ students will be $\lceil \log_{342} [N/L] \rceil$

- For example, with $N = 100,000$ (about 10 times the size of student population) the height of the tree with $M=342$ would be no more than 2, because $\lceil \log_{342}(25000) \rceil = 2$

- So, any student record can be found in 3 disk accesses. If the root of the B-Tree is stored in memory, then only 2 disk accesses are needed

**Insertion of X in a B-Tree**

- Search to find the leaf into which X should be inserted
- If the leaf has room (fewer than $L$ elements), insert X and write the leaf back to the disk.
- If the is leaf full, split it into two leaves, each with half of elements. Insert X into the appropriate new leaf and write new leaves back to the disk.
  - Update the keys in the parent
  - If the parent node is already full, split it in the same manner
  - Splits may propagate all the way to the root, in which case, the root is split (this is how the tree grows in height)

**Insert 33 into This B-Tree**

- Insertion process involves finding the appropriate leaf, splitting the leaf if necessary, and updating the parent nodes.

![Figure 5 - Before inserting 33](image-url)
Inserting 33

- Traversing the tree from the root, we find that 33 is less than 36 and is greater than 22, leading us to the 2nd subtree. Since 33 is greater than 32 we are led to the 3rd leaf (the one containing 32 and 34).
- There is room for an additional data item in the leaf it is inserted (in sorted order which means reorganizing the leaf).

Now Insert 35

- This item also belongs in the 3rd leaf of the 2nd subtree. However, that leaf is full.
- Split the leaf in two and update the parent to get the tree in Figure 7.
Inserting 21

- This item belongs in the 4th leaf of the 1st subtree (the leaf containing 18, 19, 20)
- Since the leaf is full, we split it and update the keys in the parent
- However, the parent is also full, so it must be split and its parent (the root) updated
- But this would give the root 5 subtrees which is not allowed, so the root must also be split
- This is the only way the tree grows in height

B-Tree Deletion

- Find leaf containing element to be deleted
- If that leaf is still full enough (still has at least $\lceil L/2 \rceil$ elements after remove) write it back to disk without that element. Then, change the key in the ancestor if necessary
- If leaf is now too empty (has less than $\lceil L/2 \rceil$ elements), borrow an element from a neighbor.
  - If neighbor would be too empty, combine two leaves into one
  - This combining requires updating the parent, which may now have too few subtrees
  - If necessary, continue the combining up the tree