Trees

CptS 223 – Advanced Data Structures

Larry Holder
School of Electrical Engineering and Computer Science
Washington State University
Trees (e.g.)

- Image processing
- Phylogenetics
- Organization charts
- Large databases
Overview

- Tree data structure
- Binary search trees
  - Support $O(\log_2 N)$ operations
  - Balanced trees
- B-trees for accessing secondary storage
- STL set and map classes
- Applications
Trees

G is parent of N and child of A

M is child of F and grandchild of A

Generic Tree:

- root
  - $T_1$
  - $T_2$
  - $T_3$
  - $T_4$
  - ...
Definitions

- A tree $T$ is a set of nodes
  - Each non-empty tree has a root node and zero or more sub-trees $T_1, \ldots, T_k$
  - Each sub-tree is a tree
  - The root of a tree is connected to the root of each subtree by a directed edge

- If node $n_1$ connects to sub-tree rooted at $n_2$, then
  - $n_1$ is the parent of $n_2$
  - $n_2$ is a child of $n_1$

- Each node in a tree has only one parent
  - Except the root, which has no parent
Definitions

- Nodes with no children are **leaves**
- Nodes with the same parent are **siblings**
- A **path** from nodes $n_1$ to $n_k$ is a sequence of nodes $n_1, n_2, \ldots, n_k$ such that $n_i$ is the parent of $n_{i+1}$ for $1 \leq i < k$
  - The **length** of a path is the number of edges on the path (i.e., $k-1$)
  - Each node has a path of length 0 to itself
  - There is exactly one path from the root to each node in a tree
- Nodes $n_i, \ldots, n_k$ are **descendants** of $n_i$ and **ancestors** of $n_k$
- Nodes $n_{i+1}, \ldots, n_k$ are **proper descendants**
- Nodes $n_i, \ldots, n_{k-1}$ are **proper ancestors**
Definitions

B, C, D, E, F, G are siblings

B, C, H, I, P, Q, K, L, M, N are leaves

K, L, M are siblings

The path from A to Q is A – E – J – Q
A, E, J are proper ancestors of Q
E, J, Q (and I, P) are proper descendants of A
Definitions

- The **depth** of a node $n_i$ is the length of the unique path from the root to $n_i$
  - The root node has a depth of 0
  - The depth of a tree is the depth of its deepest leaf

- The **height** of a node $n_i$ is the length of the longest path from $n_i$ to a leaf
  - All leaves have a height of 0
  - The height of a tree is the height of its root node

- The height of a tree equals its depth
Trees

Height of each node?
Height of tree?
Depth of each node?
Depth of tree?
Implementation of Trees

- Solution 1: Vector of children

```cpp
struct TreeNode
{
    Object element;
    vector<TreeNode> children;
}
```

- Solution 2: List of children

```cpp
struct TreeNode
{
    Object element;
    list<TreeNode> children;
}
```
Implementation of Trees

Solution 3: First-child, next-sibling

```c
struct TreeNode
{
    Object element;
    TreeNode *firstChild;
    TreeNode *nextSibling;
}
```
A **binary tree** is a tree where each node has no more than two children.

If a node is missing one or both children, then that child pointer is `NULL`.

```c
struct BinaryTreeNode {
    Object element;
    BinaryTreeNode *leftChild;
    BinaryTreeNode *rightChild;
}
```
Example: Expression Trees

- Store expressions in a binary tree
  - Leaves of tree are operands (e.g., constants, variables)
  - Other internal nodes are unary or binary operators
- Used by compilers to parse and evaluate expressions
  - Arithmetic, logic, etc.
- E.g., \((a + b \times c) + ((d \times e + f) \times g)\)
Example: Expression Trees

- Evaluate expression
  - Recursively evaluate left and right subtrees
  - Apply operator at root node to results from subtrees
  - **Post-order** traversal: left, right, root

- Traversals
  - **Pre-order** traversal: root, left, right
  - **In-order** traversal: left, root, right
Traversals

- Pre-order:
- Post-order:
- In-order:
Example: Expression Trees

- Constructing an expression tree from postfix notation
  - Use a stack of pointers to trees
  - Read postfix expression left to right
  - If operand, then push on stack
  - If operator, then:
    - Create a BinaryTreeNode with operator as the element
    - Pop top two items off stack
    - Insert these items as left and right child of new node
    - Push pointer to node on the stack
Example: Expression Trees

E.g., $a \ b + c \ d \ e + \ * \ *$

(1) $\top$

(2) $\top$

(3) $\top$

(4) $\top$
Example: Expression Trees

- E.g., $a \ b + c \ d \ e + * *$

(5) $\begin{array}{c}
\text{top} \\
\text{+} \\
a \\
b \\
* \\
c \\
+ \\
d \\
e
\end{array}$

(6) $\begin{array}{c}
\text{top} \\
* \\
+ \\
a \\
b \\
c \\
+ \\
d \\
e
\end{array}$
Binary Search Trees

- Complexity of searching for an item in a binary tree containing \( N \) nodes is \( O(\log N) \)
- Binary search tree (BST)
  - For any node \( n \), items in left subtree of \( n \) \( \leq \) item in node \( n \) \( \leq \) items in right subtree of \( n \)
Searching in BSTs

Contains \((T, x)\)

\[
\begin{aligned}
\text{if } (T == \text{NULL}) & \text{ then return NULL} \\
\text{if } (T->\text{element} == x) & \text{ then return } T \\
\text{if } (x < T->\text{element}) & \text{ then return } \text{Contains } (T->\text{leftChild}, x) \\
\text{else return } & \text{Contains } (T->\text{rightChild}, x)
\end{aligned}
\]

Typically assume no duplicate elements. If duplicates, then store counts in nodes, or each node has a list of objects.
Searching in BSTs

- Complexity of searching a BST with \( N \) nodes is \( O(?) \)
- Complexity of searching a BST of height \( h \) is \( O(h) \)
- \( h = f(N) \)?
Searching in BSTs

- Finding the minimum element
  - Smallest element in left subtree

```c
findMin (T)
{
    if (T == NULL) then return NULL
    if (T->leftChild == NULL) then return T
    else return findMin (T->leftChild)
}
```

- Complexity ?
Searching in BSTs

- Finding the maximum element
  - Largest element in right subtree

```c
findMax (T)
{
    if (T == NULL)
        then return NULL
    if (T->rightChild == NULL)
        then return T
    else return findMax (T->rightChild)
}
```

- Complexity?
Printing BSTs

- In-order traversal

```
PrintTree (T)
{
    if (T == NULL)
        then return
    PrintTree (T->leftChild)
    cout << T->element
    PrintTree (T->rightChild)
}
```

- Complexity?
Inserting into BSTs

- E.g., insert 5
Inserting into BSTs

- “Search” for element until reach end of tree; insert new element there

```c
Insert (x, T)
{
    if (T == NULL)
        then T = new Node(x)
    if (x < T->element)
        then if (T->leftChild == NULL)
                        then T->leftChild = new Node(x)
                        else Insert (x, T->leftChild)
        else if (T->rightChild == NULL)
                        then (T->rightChild = new Node(x)
                        else Insert (x, T->rightChild)
    }
```

Complexity?
Removing from BSTs

- Case 1: Node to remove has 0 or 1 child
  - Just remove it
- E.g., remove 4
Removing from BSTs

- Case 2: Node to remove has 2 children
  - Replace node element with successor
  - Remove successor (case 1)
- E.g., remove 2

```
  6
  / \   \
 2   8
 / \   \
1   5   \
 3   4
```

```
  6
  /   \
 3     8
 /     \
1 5   4
```
Removing from BSTs

Remove \( (x, T) \)
{
    if (T == NULL) then return
    if (x == T->element)
        then if ((T->left == NULL) && (T->right != NULL))
            then T = T->right // implied delete
            else if ((T->right == NULL) && (T->left != NULL))
                then T = T->left // implied delete
                else successor = findMin (T->right) // Case 2
                    T->element = successor->element
                    Remove (T->element, T->right)
        else if (x < T->element)
            then Remove (x, T->left)
        else Remove (x, T->right)
}

Complexity?
Implementation of BST

```cpp
template <typename Comparable>
class BinarySearchTree {
    public:
        BinarySearchTree();
        BinarySearchTree(const BinarySearchTree & rhs);
        ~BinarySearchTree();

        const Comparable & findMin() const;
        const Comparable & findMax() const;
        bool contains(const Comparable & x) const;
        bool isEmpty() const;
        void printTree() const;

        void makeEmpty();
        void insert(const Comparable & x);
        void remove(const Comparable & x);

        const BinarySearchTree & operator=(const BinarySearchTree & rhs);
};
```
private:
  struct BinaryNode
  {
    Comparable element;
    BinaryNode *left;
    BinaryNode *right;

    BinaryNode( const Comparable & theElement, BinaryNode *lt, BinaryNode *rt )
      : element( theElement ), left( lt ), right( rt ) { }
  };

BinaryNode *root;

void insert( const Comparable & x, BinaryNode * & t ) const;
void remove( const Comparable & x, BinaryNode * & t ) const;
BinaryNode * findMin( BinaryNode *t ) const;
BinaryNode * findMax( BinaryNode *t ) const;
bool contains( const Comparable & x, BinaryNode *t ) const;
void makeEmpty( BinaryNode * & t );
void printTree( BinaryNode *t ) const;
BinaryNode * clone( BinaryNode *t ) const;

Pointer to tree node passed by reference so it can be reassigned within function.
/*
 * Returns true if x is found in the tree.
 */
bool contains( const Comparable & x ) const
{
    return contains( x, root );
}

/**
 * Insert x into the tree; duplicates are ignored.
 */
void insert( const Comparable & x )
{
    insert( x, root );
}

/**
 * Remove x from the tree. Nothing is done if x is not found.
 */
void remove( const Comparable & x )
{
    remove( x, root );
}
/**
 * Internal method to test if an item is in a subtree.
 * x is item to search for.
 * t is the node that roots the subtree.
 */

bool contains( const Comparable & x, BinaryNode *t ) const
{
    if( t == NULL )
        return false;
    else if( x < t->element )
        return contains( x, t->left );
    else if( t->element < x )
        return contains( x, t->right );
    else
        return true;    // Match
}
/**
 * Internal method to find the smallest item in a subtree t.
 * Return node containing the smallest item.
 */

BinaryNode * findMin( BinaryNode *t ) const
{
    if( t == NULL )
        return NULL;
    if( t->left == NULL )
        return t;
    return findMin( t->left );
}

/**
 * Internal method to find the largest item in a subtree t.
 * Return node containing the largest item.
 */

BinaryNode * findMax( BinaryNode *t ) const
{
    if( t != NULL )
        while( t->right != NULL )
            t = t->right;
    return t;
}
/**
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */

void insert( const Comparable & x, BinaryNode * & t )
{
    if( t == NULL )
        t = new BinaryNode( x, NULL, NULL );
    else if( x < t->element )
        insert( x, t->left );
    else if( t->element < x )
        insert( x, t->right );
    else
        ; // Duplicate; do nothing
}
/**
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */

void remove( const Comparable & x, BinaryNode * & t )
{
    if( t == NULL )
        return; // Item not found; do nothing
    if( x < t->element )
        remove( x, t->left );
    else if( t->element < x )
        remove( x, t->right );
    else if( t->left != NULL && t->right != NULL ) // Two children
    {
        t->element = findMin( t->right )->element;
        remove( t->element, t->right );
    }
    else
    {
        BinaryNode *oldNode = t;
        t = ( t->left != NULL ) ? t->left : t->right;
        delete oldNode;
    }
}
1     /**
2     * Destructor for the tree
3     */
4     ~BinarySearchTree()
5     {
6     makeEmpty();
7     }
8     /**<
9     * Internal method to make subtree empty.
10    */
11    void makeEmpty( BinaryNode * & t )
12    {
13      if( t != NULL )
14      {
15        makeEmpty( t->left );
16        makeEmpty( t->right );
17        delete t;
18      }
19      t = NULL;
20    }
/**
 * Deep copy.
 */

const BinarySearchTree & operator=( const BinarySearchTree & rhs )
{
    if( this != &rhs )
    {
        makeEmpty( );
        root = clone( rhs.root );
    }
    return *this;
}

/**
 * Internal method to clone subtree.
 */

BinaryNode * clone( BinaryNode *t ) const
{
    if( t == NULL )
        return NULL;

    return new BinaryNode( t->element, clone( t->left ), clone( t->right ) );
}
BST Analysis

- `printTree`, `makeEmpty` and `operator=`
  - Always $O(N)$

- `insert`, `remove`, `contains`, `findMin`, `findMax`
  - $O(d)$, where $d =$ depth of tree
  - Worst case: $d =$ ?
  - Best case: $d =$ ? (not when $N=0$)
  - Average case: $d =$ ?
BST Average-Case Analysis

- Internal path length
  - Sum of the depths of all nodes in the tree
- Compute average internal path length over all possible insertion sequences
  - Assume all insertion sequences are equally likely
    - E.g., “1 2 3 4 5 6 7”, “7 6 5 4 3 2 1”,..., “4 2 6 1 3 5 7”
    - Result: $O(N \log_2 N)$
- Thus, average depth $= O(N \log_2 N) / N = O(\log_2 N)$
Randomly Generated 500-node BST (insert only)

Average node depth = 9.98
\[ \log_2 500 = 8.97 \]
Previous BST after $500^2$
Random Insert/Remove Pairs

Average node depth = 12.51
$\log_2 500 = 8.97$
BST Average-Case Analysis

- After randomly inserting $N$ nodes into an empty BST
  - Average depth $= \mathcal{O}(\log_2 N)$
- After $\Theta(N^2)$ random insert/remove pairs into an $N$-node BST
  - Average depth $= \Theta(N^{1/2})$

Why?

Solutions?
- Overcome problematic average cases?
- Overcome worst case?
Balanced BSTs

- **AVL trees**
  - Height of left and right subtrees at every node in BST differ by at most 1
  - Maintained via rotations
  - BST depth always $O(\log_2 N)$

- **Splay trees**
  - After a node is accessed, push it to the root via AVL rotations
  - Average depth per operation is $O(\log_2 N)$
AVL Trees

- AVL (Adelson-Velskii and Landis, 1962)
- For every node in the BST, the heights of its left and right subtrees differ by at most 1
- Height of BST is $O(\log_2 N)$
  - Actually, $1.44 \log_2(N+2) – 1.328$
  - Minimum nodes $S(h)$ in AVL tree of height $h$
    - $S(h) = S(h-1) + S(h-2) + 1$
    - Similar to Fibonacci recurrence
AVL Trees

AVL tree?  

AVL tree?
Maintaining Balance Condition

- If we can maintain balance condition, then all BST operations are $O(\log_2 N)$
- Maintain height $h(t)$ at each node $t$
  - $h(t) = \max (h(t->left), h(t->right)) + 1$
  - $h($empty tree$) = -1$
- Which operations can upset balance condition?
AVL Remove

- Assume `remove` accomplished using lazy deletion
  - Removed nodes only marked as deleted, but not actually removed from BST
  - Unmarked when same object re-inserted
    - Re-allocation time avoided
  - Does not affect $O(\log_2 N)$ height as long as deleted nodes are not in the majority
  - Does require additional memory per node
- Can accomplish `remove` without lazy deletion
AVL Insert

- Insert can violate AVL balance condition
- Can be fixed by a rotation

Inserting 6 violates AVL balance condition

```
2
/   \
1     4
/     /
3     7
```

Rotating 7-8 restores balance

```
2
/   \
1     4
/     /
3     6
```

```
      5
    /    \
  2      7
 /       /  \
1  4     6  8
```

```
      5
    /    \
  2      7
 /       /  \
1  4     6  8
```
AVL Insert

- Only nodes along path to insertion have their balance altered
- Follow path back to root, looking for violations
- Fix violations using single or double rotations
AVL Insert

- Assume node \( k \) needs to be rebalanced
- Four cases leading to violation
  1. An insertion into the left subtree of the left child of \( k \)
  2. An insertion into the right subtree of the left child of \( k \)
  3. An insertion into the left subtree of the right child of \( k \)
  4. An insertion into the right subtree of the right child of \( k \)
- Cases 1 and 4 handled by single rotation
- Cases 2 and 3 handled by double rotation
AVL Insert

- Case 1: Single rotation right

Violation

AVL balance condition okay. BST order okay.
AVL Insert

- Case 1 example
AVL Insert

- Case 4: Single rotation left

AVL balance condition okay. BST order okay.
AVL Insert

- Case 2: Single rotation fails
AVL Insert

- Case 2: Left-right double rotation

Violation

AVL balance condition okay.
BST order okay.
AVL Insert

- Case 3: Right-left double rotation

Violation

AVL balance condition okay.
BST order okay.
AVL Tree Implementation

```c
1 struct AvlNode
2 {
3     Comparable element;
4     AvlNode *left;
5     AvlNode *right;
6     int    height;
7
8     AvlNode( const Comparable & theElement, AvlNode *lt,
9                   AvlNode *rt, int h = 0 )
10     : element( theElement ), left( lt ), right( rt ), height( h )
11     };
```
AVL Tree Implementation

1     /**
2     * Return the height of node t or -1 if NULL.
3     */
4     int height( AvlNode *t ) const
5     {
6         return t == NULL ? -1 : t->height;
7     }
/**
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void insert( const Comparable & x, AvlNode * & t )
{
    if( t == NULL )
        t = new AvlNode( x, NULL, NULL );
    else if( x < t->element )
    {
        insert( x, t->left );
        if( height( t->left ) - height( t->right ) == 2 )
            if( x < t->left->element )
                rotateWithLeftChild( t );
            else
                doubleWithLeftChild( t );
    }
    else if( t->element < x )
    {
        insert( x, t->right );
        if( height( t->right ) - height( t->left ) == 2 )
            if( t->right->element < x )
                rotateWithRightChild( t );
            else
                doubleWithRightChild( t );
    }
    else
        ; // Duplicate; do nothing
    t->height = max( height( t->left ), height( t->right ) ) + 1;
}
/**
 * Rotate binary tree node with left child.
 * For AVL trees, this is a single rotation for case 1.
 * Update heights, then set new root.
 */

void rotateWithLeftChild( AvlNode * & k2 )
{
    AvlNode *k1 = k2->left;
    k2->left = k1->right;
    k1->right = k2;
    k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
    k1->height = max( height( k1->left ), k2->height ) + 1;
    k2 = k1;
}
```c
1     /**
2     * Double rotate binary tree node: first left child
3     * with its right child; then node k3 with new left child.
4     * For AVL trees, this is a double rotation for case 2.
5     * Update heights, then set new root.
6     */
7     void doubleWithLeftChild( AvlNode * & k3 )
8     {
9         rotateWithRightChild( k3->left );
10        rotateWithLeftChild( k3 );
11     }
```
Splay Tree

- After a node is accessed, push it to the root via AVL rotations
- Guarantees that any $M$ consecutive operations on an empty tree will take at most $O(M \log_2 N)$ time
- Amortized cost per operation is $O(\log_2 N)$
- Still, some operations may take $O(N)$ time
- Does not require maintaining height or balance information
Splay Tree

Solution 1
- Perform single rotations with accessed/new node and parent until accessed/new node is the root

Problem
- Pushes current root node deep into tree
- In general, can result in $O(M \times N)$ time for $M$ operations
- E.g., insert 1, 2, 3, ..., $N$
Splay Tree

- Solution 2
  - Still rotate tree on the path from the new/accessed node X to the root
  - But, rotations are more selective based on node, parent and grandparent
  - If X is child of root, then rotate X with root
  - Otherwise, ...
Splaying: Zig-zag

- Node X is right-child of parent, which is left-child of grandparent (or vice-versa)
- Perform double rotation (left, right)
Splaying: Zig-zig

- Node X is left-child of parent, which is left-child of grandparent (or right-right)
- Perform double rotation (right-right)
Splay Tree: Example

- Consider previous worst-case scenario: insert 1, 2, ..., N; then access 1
Splay Tree: Remove

- Access node to be removed (now at root)
- Remove node leaving two subtrees $T_L$ and $T_R$
- Access largest element in $T_L$
  - Now at root; no right child
- Make $T_R$ right child of root of $T_L$
Balanced BSTs

- **AVL trees**
  - Guarantees $O(\log_2 N)$ behavior
  - Requires maintaining height information

- **Splay trees**
  - Guarantees amortized $O(\log_2 N)$ behavior
  - Moves frequently-accessed elements closer to root of tree

- Both assume N-node tree can fit in main memory
  - If not?
# Top 10 Largest Databases

<table>
<thead>
<tr>
<th>Organization</th>
<th>Database Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>WDCC</td>
<td>6,000 TBs</td>
</tr>
<tr>
<td>NERSC</td>
<td>2,800 TBs</td>
</tr>
<tr>
<td>AT&amp;T</td>
<td>323 TBs</td>
</tr>
<tr>
<td>Google</td>
<td>33 trillion rows (91 million insertions per day)</td>
</tr>
<tr>
<td>Sprint</td>
<td>3 trillion rows (100 million insertions per day)</td>
</tr>
<tr>
<td>ChoicePoint</td>
<td>250 TBs</td>
</tr>
<tr>
<td>Yahoo!</td>
<td>100 TBs</td>
</tr>
<tr>
<td>YouTube</td>
<td>45 TBs</td>
</tr>
<tr>
<td>Amazon</td>
<td>42 TBs</td>
</tr>
<tr>
<td>Library of Congress</td>
<td>20 TBs</td>
</tr>
</tbody>
</table>


How many bytes in a “yotta”-byte?
Use a BST?

- Google: 33 trillion items
- Indexed by IP (duplicates)
- Access time
  - $h = \log_2 33 \times 10^{12} = 44.9$
  - Assume 120 disk accesses per second
  - Each search takes 0.37 seconds
  - Assumes exclusive use of data
Idea

- Use a 3-way search tree
- Each node stores 2 keys and has at most 3 children
- Each node access brings in 2 keys and 3 child pointers
- Height of a balanced 3-way search tree?
Bigger Idea

- Use an M-ary search tree
- Each node access brings in M-1 keys and M child pointers
- Choose M so node size = disk page size
- Height of tree = $\log_M N$
Example

- Standard disk page size = 8192 bytes
- Assume keys use 32 bytes, pointers use 4 bytes
  - Keys uniquely identify data elements
- $32*(M-1) + 4*M = 8192$
- $M = 228$
- $\log_{228} 33 \times 10^{12} = 5.7$ (disk accesses)
- Each search takes 0.047 seconds
A **B-tree** (also called a **B⁺ tree**) of order $M$ is an $M$-ary tree with the following properties:

1. Data items are stored at the leaves.
2. Non-leaf nodes store up to $M-1$ keys.
   - Key $i$ represents the smallest key in subtree $i+1$.
3. Root node is either a leaf or has between 2 and $M$ children.
4. Non-leaf nodes have between $\lceil M/2 \rceil$ and $M$ children.
5. All leaves at same depth and have between $\lceil L/2 \rceil$ and $L$ data items.

Requiring nodes to be half full avoids degeneration into binary tree.
B-tree

- B-tree of order 5
  - Node has 2-4 keys and 3-5 children
  - Leaves have 3-5 data elements
B-tree: Choosing L

- Assuming a data element requires 256 bytes
- Leaf node capacity of 8192 bytes implies $L=32$
- Each leaf node has between 16 and 32 data elements
- Worst case for Google
  - Leaves = $33 \times 10^{12} / 16 = 2 \times 10^{12}$
  - $\log_{M/2} 2 \times 10^{12} = \log_{114} 2 \times 10^{12} = 5.98$
B-tree: Insertion

- **Case 1: Insert into a non-full leaf node**
  - E.g., insert 57 into previous order 5 tree
**B-tree: Insertion**

- **Case II:** Insert into full leaf, but parent has room
  - Split leaf and promote middle element to parent
  - E.g., insert 55 into previous tree

![B-tree diagram]

- 41 66 87
- 8 18 26 35
- 48 51 54 57
- 72 78 83
- 92 97

2 4 6
- 8 10 12 14 16
- 35 36 37 38 39
- 41 42 44 46
- 51 52 53
- 54 55 56 57 58 59
- 66 68 69 70
- 72 73 74 76
- 83 84 85
- 87 89 90 92 93 95 97 98 99
B-tree: Insertion

- Case III: Insert into full leaf, parent has no room
  - Split parent, promote parent’s middle element to grandparent
  - Continue until non-full parent or split root
  - E.g., insert 40 into previous tree

Insert 43 and 45?
B-tree: Deletion

- Case 1: Leaf node containing item not at minimum
  - E.g., remove 16 from previous tree
B-tree: Deletion

- Case 2: Leaf node containing item has minimum elements, neighbor not at minimum
  - Adopt element from neighbor
  - E.g., remove 6 from previous tree
**B-tree: Deletion**

- Case 3: Leaf node containing item has minimum elements, neighbors have minimum elements
  - Merge with neighbor and intermediate key
  - If parent now below minimum, continue up the tree
  - E.g., remove 99 from previous tree
B-trees

- B-trees are ordered search trees optimized for large N and secondary storage
- B-trees are M-ary trees with height \( \log_M N \)
  - \( M = O(10^2) \) based on disk page sizes
  - E.g., trillions of elements stored in tree of height 6
- Basis of many database architectures
C++ STL Sets and Maps

- `vector` and `list` STL classes inefficient for search
- STL `set` and `map` classes guarantee logarithmic insert, delete and search
STL set Class

- STL set class is an ordered container that does not allow duplicates
- Like lists and vectors, sets provide iterators and related methods: begin, end, empty and size
- Sets also support insert, erase and find
Set Insertion

- `insert` adds an item to the set and returns an iterator to it.
- Because a set does not allow duplicates, `insert` may fail.
  - In this case, `insert` returns an iterator to the item causing the failure.
- To distinguish between success and failure, `insert` actually returns a pair of results.
  - This `pair` structure consists of an iterator and a Boolean indicating success.

```cpp
pair<iterator,bool> insert (const Object & x);
```
Sidebar: STL pair Class

- `pair<Type1,Type2>`
- Methods: `first`, `second`, `first_type`, `second_type`

```cpp
#include <utility>

pair<iterator,bool> insert (const Object & x) {
  iterator itr;
  bool found;
  ...
  return pair<itr,found>;
}
```
Set Insertion

- Giving `insert` a hint
  ```
  pair<iterator,bool> insert (iterator hint, const Object & x);
  ```
- For good hints, `insert` is O(1)
- Otherwise, reverts to one-parameter `insert`
- E.g.,
  ```
  set<int> s;
  for (int i = 0; i < 1000000; i++)
      s.insert (s.end(), i);
  ```
Set Deletion

- `int erase (const Object & x);`
  - Remove x, if found
  - Return number of items deleted (0 or 1)
- `iterator erase (iterator itr);`
  - Remove object at position given by iterator
  - Return iterator for object after deleted object
- `iterator erase (iterator start, iterator end);`
  - Remove objects from start up to (but not including) end
  - Returns iterator for object after last deleted object
Set Search

- `iterator find (const Object & x) const;`
  - Returns iterator to object (or end() if not found)
  - Unlike `contains`, which returns Boolean
- `find` runs in logarithmic time
**STL map Class**

- STL `map` class stores items, where an item consists of a key and a value.
- Like a `set` instantiated with a key/value pair.
- Keys must be unique.
- Different keys can map to the same value.
- `map` keeps items in order by key.
STL map Class

- Methods
  - `begin`, `end`, `size`, `empty`
  - `insert`, `erase`, `find`
- Iterators reference items of type `pair<KeyType,ValueType>`
- Inserted elements are also of type `pair<KeyType,ValueType>`
STL `map` Class

- Main benefit: overloaded `operator[]`
  ```cpp
  ValueType & operator[](const KeyType & key);
  ```
- If key is present in map
  - Returns reference to corresponding value
- If key is not present in map
  - Key is inserted into map with a default value
  - Reference to default value is returned

```cpp
map<string,double> salaries;
salaries["Pat"] = 75000.0;
```
Example

```cpp
struct ltstr {
    bool operator()(const char* s1, const char* s2) const {
        return strcmp(s1, s2) < 0;
    }
};

int main() {
    map<const char*, int, ltstr> months;
    months["january"] = 31;
    months["february"] = 28;
    months["march"] = 31;
    months["april"] = 30;
    ...
```

Example Comparator if key type not primitive
Example (cont.)

... 

months["may"] = 31;
months["june"] = 30;
months["july"] = 31;
months["august"] = 31;
months["september"] = 30;
months["october"] = 31;
months["november"] = 30;
months["december"] = 31;
cout << "june -> " << months["june"] << endl;
map<const char*, int, ltstr>::iterator cur = months.find("june");
map<const char*, int, ltstr>::iterator prev = cur;
map<const char*, int, ltstr>::iterator next = cur;
++next; --prev;
cout << "Previous (in alphabetical order) is " << (*prev).first << endl;
cout << "Next (in alphabetical order) is " << (*next).first << endl;
}
Implementation of set and map

- Support insertion, deletion and search in worst-case logarithmic time
- Use balanced binary search tree
- Support for iterator
  - Tree node points to its predecessor and successor
  - Use only un-used tree left/right child pointers
  - Called a “threaded tree”
Summary: Trees

- Trees are ubiquitous in software
- Search trees important for fast search
  - Support logarithmic searches
  - Must be kept balanced (AVL, Splay, B-tree)
- STL `set` and `map` classes use balanced trees to support logarithmic insert, delete and search