Chapter 4
Trees
Introduction

– for large input, even _______ access time may be prohibitive
  – we need data structures that exhibit __________ running times closer to $O(\log N)$
  – binary search tree
- recursive definition of **tree**
  - collection of nodes (may be __________)
  - distinguished node, $r$, is the **root**
  - zero or more nonempty subtrees $T_1$, $T_2$, … $T_k$, each of whose roots are connected by a directed **edge** from $r$
  - _______ of each subtree is a **child** of $r$
  - $r$ is the **parent** of each subtree
  - tree of $N$ nodes has $N – 1$ ___________
Terminology

- example tree

- nodes with no ___________ are called leaves (e.g., B, C, H, I, P, Q, K, L, M, N)
- nodes with the same ___________ are siblings (e.g., K, L, M)
- parent, grandparent, grandchild, ancestor, descendant, proper ancestor, proper descendant
- **example tree**

- **path** from \( n_1 \) to \( n_k \) is a sequence of nodes \( n_1, n_2, \ldots, n_k \) where \( n_1 \) is the parent of \( n_{i+1} \) for \( 1 \leq i < k \)

- **length** of path is number of \___________\ on path \( (k - 1) \)
  - path of length 0 from every node to \___________\ 
  - exactly one \___________\ from the root to each node
Terminology

- example tree

- **depth** from \( n_i \) is the length of the unique path from the _________ to \( n_i \)
  - _________ is at depth 0
- **height** of \( n_i \) is the length of the _______ path from \( n_i \) to a leaf
  - all ___________ at height 0
- height of the tree is equal to the height of the root
- example tree

- ___ is at depth 1 and height 2
- ___ is at depth 1 and height 1
- depth of tree is ____
Implementation of Trees

- each node could have ______ and a link to each ________
  - number of children is unknown and may be large, which could lead to ________________
  - instead, keep children in a linked list

```c
1   struct TreeNode
2   { 
3       Object  element;
4       TreeNode *firstChild;
5       TreeNode *nextSibling;
6   }
```

- null links not shown
Tree Traversals with Application

- many _______________ for trees
  - subdirectory structure in Unix
  - pathname built into tree
goal: list all files in a directory
- depth denoted by tabs
- begins at root

```c++
void FileSystem::listAll( int depth = 0 ) const
{
    printName( depth ); // Print the name of the object
    if( isDirectory() )
        for each file c in this directory (for each child)
            c.listAll( depth + 1 );
}
```
Tree Traversals with Application

- code prints directories/files in ____________________ traversal
- runs in $O(N)$
for postorder traversal, numbers in parentheses represent the number of ___________ for each file
- size method to find number of __________ for each file
  - directories use 1 block of space

```cpp
int FileSystem::size() const
{
    int totalSize = sizeOfFile();

    if( isDirectory( ) )
        for each file c in this directory (for each child)
            totalSize += c.size( );

    return totalSize;
}
```
- in binary trees, nodes can have no more than _____ children
- binary tree below consists of a root and two subtrees, $T_L$ and $T_R$, both of which could possibly be ___________
- depth of a binary tree is considerably smaller than _____
- average depth is $O(\sqrt{N})$
- average depth for a binary search tree is ____________
  - depth can be as large as ___________
Binary Tree Implementation

− since a binary tree has two children at most, we can keep ________________ to each of them
  − element plus two pointers, left and right

```c
struct BinaryNode
{
    Object element;  // The data in the node
    BinaryNode *left;  // Left child
    BinaryNode *right;  // Right child
};
```

− drawn with circles and lines (graph)
− many applications, including compiler design
Binary Tree Example: Expression Trees

- expression tree
  - leaves represent ___________ (constants or variable names)
  - interior nodes represent ______________
- binary tree since most operators are binary, but not required
  - some operations are ___________
- evaluate expression tree, T, by applying operator at root to values obtained by evaluating left and right subtrees

- left subtree: a + (b * c)
- right subtree: ((d * e) + f) * g

- (a + (b * c)) + (((d * e) + f) * g)
Binary Tree Example: Expression Trees

- __________ traversal
  - recursively produce left expression
  - print operator at root
  - recursively produce right expression
- __________ traversal
  - result: a b c * + d e * f + g * +
- __________ traversal
  - result: + + a * b c * + * d e f g
- goal: convert a postorder expression into an expression tree
- read expression one \[\text{operator}\] at a time
- if \[\text{operator}\], create node and push a pointer to it on the stack
- if \[\text{operator}\], pop pointers to two trees \[T_1\] and \[T_2\] from the stack
  - form new tree with operator as root
  - pointer to this tree is then pushed on the stack
Binary Tree Example: Expression Trees

- example: a b + c d e + * *
- first two symbols are ___________ and are pushed on the stack

[Diagram of binary tree with nodes labeled 'a' and 'b']
Binary Tree Example: Expression Trees

- example: `a b + c d e + * *`
- after `+` is read, two pointers are _____________ and new tree formed with a pointer pushed on the stack
Binary Tree Example: Expression Trees

- example: a b + c d e + * *
- next, c, d, and e are read, with _____________ tree created for each and pushed on the stack
Binary Tree Example: Expression Trees

- example: \( a \ b \ + \ c \ d \ e \ + \ * \ * \)
- after + is read, two trees are ____________

```
+    +
\|   /\|
|a=|  |c=|
  \|  /\|
   b |  d |
    \   /  
     e   
```
- example: \( a \ b + c \ d \ e + * \ *
- after * is read, two trees are popped to form a new tree with \( a \ * \) as root
Binary Tree Example: Expression Trees

- example: $a \ b + \ c \ d \ e + * \ *$
- finally * is read, two trees are popped to form a _______ tree, which is left on the stack
Binary Search Tree ADT

- binary trees often used for ________________
- assume each node in the tree stores one ________ (integer)
- binary search tree
  - for every node $X$ in the tree
    - all items in left subtree are ________________ than $X$
    - all items in right subtree are greater than $X$
  - items in tree must be order-able
- common operations on binary search trees
  - often written ______________________
  - since average depth is $O(\log N)$, no worry about stack space
- binary search tree interface
  - searching depends on $<$ operator, which must be defined for Comparable type
  - only data member is ______________________
template <typename Comparable>
class BinarySearchTree
{
    public:
        BinarySearchTree( );
        BinarySearchTree(const BinarySearchTree & rhs );
        BinarySearchTree(const BinarySearchTree && rhs );
        ~BinarySearchTree( );

        const Comparable & findMin( ) const;
        const Comparable & findMax( ) const;
        bool contains( const Comparable & x ) const;
        bool isEmpty( ) const;
        void printTree( ostream & out = cout ) const;

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

        BinarySearchTree & operator=( const BinarySearchTree & rhs );
        BinarySearchTree & operator=( BinarySearchTree && rhs );
Binary Search Tree ADT

```cpp
private:
  struct BinaryNode
  {
    Comparable element;
    BinaryNode *left;
    BinaryNode *right;

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

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

  BinaryNode *root;

  void insert( const Comparable & x, BinaryNode * & t );
  void insert( Comparable && x, BinaryNode * & t );
  void remove( const Comparable & x, BinaryNode * & t );
  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, ostream & out ) const;
  BinaryNode * clone( BinaryNode *t ) const;
};
```
Binary Search Tree ADT

- test for item in subtree

```cpp
1  /**
2   * Internal method to test if an item is in a subtree.
3   * x is item to search for.
4   * t is the node that roots the subtree.
5   */
6  bool contains( const Comparable &x, BinaryNode *t ) const
7  {
8      if( t == nullptr )
9          return false;
10     else if( x < t->element )
11          return contains( x, t->left );
12     else if( t->element < x )
13          return contains( x, t->right );
14     else
15         return true;    // Match
16  }
```
Binary Search Tree ADT

- `findMin` and `findMax` methods return pointer to smallest/largest elements in the tree
  - to find the minimum, start at the root and go ______ as long as possible
  - similar for finding the maximum
Binary Search Tree ADT

- ________________ version of `findMin`

```cpp
1  /**
2   * Internal method to find the smallest item in a subtree t.
3   * Return node containing the smallest item.
4   */
5  BinaryNode * findMin( BinaryNode *t ) const
6  {
7      if( t == nullptr )
8          return nullptr;
9      if( t->left == nullptr )
10         return t;
11      return findMin( t->left );
12  }
```
--- ________________ version of findMax

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

BinaryNode * findMax( BinaryNode *t ) const
{
    if( t != nullptr )
        while( t->right != nullptr )
            t = t->right;
    return t;
}
```
Binary Search Tree ADT

- insertion for binary search trees
  - to insert $X$ into tree $T$, proceed ________ the tree, as in the contains function
  - if $X$ is found, _____________________________
  - otherwise, insert $X$ at the last spot on the path traversed
- example: insert 5 into binary search tree

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

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

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

Duplicates can be handled by adding a _______ to the node record

- better than inserting ________________________ in tree
- may not work well if key is only small part of larger structure
Binary Search Tree ADT

- deletion in binary search tree may be difficult
- multiple cases
  - if node is ________, it can be deleted immediately
  - if node has only one child, node can be deleted after its parent adjusts a link to ____________ the node
- multiple cases (cont.)
  - complicated case: node with __________ children
    - replace data of this node with smallest data of right subtree and recursively delete the node
    - since smallest node in right subtree cannot have a left child, the second remove is easy
- if number of deletions small, ______ deletion may be used
  - node is marked deleted rather than actually being deleted
  - especially popular if ____________ allowed
    - count of duplicates can be decremented
    - incurs only small penalty on tree since height not affected greatly
  - if deleted node ________________, some benefits
Binary Search Tree Average-Case Analysis

we expect most operations on binary search trees will have time

- average depth over all nodes can be shown to be $O(\log N)$
- all insertions and deletions must be equally ________
- sum of the depths of all nodes in a tree is known as __________________________
Binary Search Tree Average-Case Analysis

-the run time of binary search trees depends on the _________ of the tree, which in turn depends on the _________ that the keys are inserted

-let $D(N)$ be the internal path length for a tree of $N$ nodes
- we know that $D(1) = 0$
- a tree of an $i$-node left subtree and an $(N - i - 1)$-node right subtree, plus a root at depth zero for $0 \leq i \leq N$
- total number of nodes in tree = left subtree + right subtree + 1
- all nodes except the _________ are one level deeper,

$$D(N) = D(i) + D(N - i - 1) + N - 1$$
Binary Search Tree Average-Case Analysis

-if all subtree sizes are \( \frac{1}{N} \), then the average for each subtree is

\[
\left( \frac{1}{N} \right) \sum_{j=1}^{N-1} D(j)
\]

therefore, for the total number of nodes

\[
D(N) = \left( \frac{2}{N} \right) \left[ \sum_{j=1}^{N-1} D(j) \right] + N - 1
\]

- once this recurrence relation is evaluated, the result is

\[D(N) = O(N \log N)\]

and the _______________ number of nodes is \( O(\log N) \)
- example: randomly generated 500-node tree has _________ depth of 9.98
- deletions, however, bias the _______ subtrees to be longer because we always replace a deleted node with a node from the _______ subtree
- exact effect of deletions still unknown
- if insertions and deletions are \( \Theta(N^2) \) times, then expected depth is \( \Theta(\sqrt{N}) \)
- after 250,000 random insert/delete pairs, tree becomes _________________, with depth = 12.51
Binary Search Tree Average-Case Analysis

- could randomly choose between smallest element in the right subtree and largest element in the left subtree when replacing deleted element
  - should keep ________ low, but not yet proven
- bias does not show up for ___________ trees
- if $o(N^2)$ insert/remove pairs used, tree actually gains balance
- average case analysis extremely difficult
- two possible solutions
  - balanced trees
  - self-adjusting trees
AVL Trees

- Adelson-Velskii and Landis (AVL) tree is a binary search tree with a ____________ condition
- balance condition in general
  - must be easy to maintain
  - ensures depth of tree is ________________
- simplest idea: left and right subtrees have the ____________
  - does not always work
AVL Trees

- alternate balance condition: _____________ must have left and right subtrees of the same height
  - only ______________ balanced trees of \(2^k - 1\) nodes would work
  - condition too rigid
AVL Trees

- AVL tree
  - for each node in the tree, height of left and right subtrees differ by at most ____
    - height _______ = height of right subtree – height of left
    - height of an empty tree: -1
  - height information kept in the node structure
AVL Trees

- example AVL tree
  - fewest nodes for a tree of height _____
  - left subtree contains fewest nodes for height _____
  - right subtree contains fewest nodes for height _____
number of nodes, $S(h)$, in an AVL tree of height $h$

$$S(h) = S(h - 1) + S(h - 2) + 1 \quad S(0) = 1, \quad S(1) = 2$$

closely related to _________________ numbers

- all operations can be performed in $O(\log N)$ time, except

______________________________
AVL Trees

- insertion
  - update all balance information in the nodes on the path back to the root
  - could violate the balance condition
  - _______________ used to restore the balance property
- deletion
  - perform same promotion as in a __________ search tree, updating the balance information as necessary
  - same balancing operations for __________ can then be used
AVL Trees

- if $\alpha$ is the node requiring _________________ (the heights of its left and right subtrees differ by 2), the violation occurred in one of four cases
  - an insertion into the left subtree of the left child of $\alpha$
  - an insertion into the right subtree of the left child of $\alpha$
  - an insertion into the left subtree of the right child of $\alpha$
  - an insertion into the right subtree of the right child of $\alpha$
- cases 1 and 4 are mirror image symmetries with respect to $\alpha$ and can be resolved with a _____________ rotation
- cases 2 and 3 are mirror image symmetries with respect to $\alpha$ and can be resolved with a _____________ rotation
- single rotation
  - only possible case 1 scenario
  - to balance, imagine “picking up” tree by $k_1$

- new tree has ________________ as original tree
AVL Trees

- example tree
- when adding 6, node 8 becomes ________________
- to balance, perform single rotation between 7 and 8
AVL Trees

- example tree

- __________________ case for case 4
AVL Trees

- example
  - insert 3, 2, and 1 into an empty tree
AVL Trees

- example
  - insert 4 and 5
AVL Trees

- example
- insert 6
AVL Trees

- example
  - insert 7

```
before:  
1   2   3   4
    |   |
   5   6

after: 
1   2   3
    |   |
   5   6
    |
  7
```

AVL Trees

- double rotation
  - for cases 2 and 3, a ____________ rotation will not work

- tree $Y$ can be expanded to a node with two ____________
AVL Trees

- double rotation
  - __________________ double rotation for case 2

- __________________ double rotation for case 3
AVL Trees

- example
  - insert 16 and 15
AVL Trees

- example
  - insert 14
AVL Trees

- example
  - insert 13
AVL Trees

- example
- insert 12
AVL Trees

- example
- insert 11, 10, and 8
AVL Trees

- example
  - insert 9

[Diagram of AVL tree with nodes 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, and root node 7.]
AVL Trees

- implementation
- node definition

```c
1  struct AvlNode
2  {
3      Comparable element;
4      AvlNode  *left;
5      AvlNode  *right;
6      int        height;
7
8      AvlNode( const Comparable & ele, AvlNode *lt, AvlNode *rt, int h = 0 )
9          : element{ ele }, left{ lt }, right{ rt }, height{ h } { }
10     
11      AvlNode( Comparable & & ele, AvlNode *lt, AvlNode *rt, int h = 0 )
12          : element{ std::move( ele ) }, left{ lt }, right{ rt }, height{ h } { }
13   };
```
AVL Trees

- implementation
  - function to compute height of AVL node

```c
1   /**
2   * Return the height of node t or -1 if nullptr.
3   */
4   int height( AvlNode *t ) const
5   {
6       return t == nullptr ? -1 : t->height;
7   }
```
AVL Trees

- implementation
- insertion

```c
/**
 * 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 == nullptr )
        t = new AvlNode{ x, nullptr, nullptr };
    else if( x < t->element )
        insert( x, t->left );
    else if( t->element < x )
        insert( x, t->right );
    balance( t );
}
```
AVL Trees

- implementation

```c
19  static const int ALLOWED_IMBALANCE = 1;
20
21  // Assume t is balanced or within one of being balanced
22  void balance( AvlNode * & t )
23  {
24    if( t == nullptr )
25      return;
26
27    if( height( t->left ) - height( t->right ) > ALLOWED_IMBALANCE )
28      if( height( t->left->left ) >= height( t->left->right ) )
29        rotateWithLeftChild( t );
30      else
31        doubleWithLeftChild( t );
32    else
33      if( height( t->right ) - height( t->left ) > ALLOWED_IMBALANCE )
34        if( height( t->right->right ) >= height( t->right->left ) )
35          rotateWithRightChild( t );
36      else
37        doubleWithRightChild( t );
38
39    t->height = max( height( t->left ), height( t->right ) ) + 1;
40  }
```
AVL Trees

- implementation
  - single rotation

```c
/**
 * 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;
}
```
AVL Trees

- implementation
- double rotation

```c
/**
 * Double rotate binary tree node: first left child
 * with its right child; then node k3 with new left child.
 * For AVL trees, this is a double rotation for case 2.
 * Update heights, then set new root.
 */
void doubleWithLeftChild( AvlNode * & k3 )
{
    rotateWithRightChild( k3->left );
    rotateWithLeftChild( k3 );
}
```

Before rotation:
```
        k3
       /|
      /  |
     k1   k2
      /    /
     A     D
    /\     /
   /   \   /   \
  B     k2 C     
```

After rotation:
```
        k2
       /|
      /  |
     k1   k3
      /    /
     A     B
    /\     /
   /   \   /   \
  C     k3 D```

```
AVL Trees

- implementation
- deletion

```c
/**
 * 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, AvlNode * & t )
{
    if( t == nullptr )
        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 != nullptr && t->right != nullptr ) // Two children
    {
        t->element = findMin( t->right )->element;
        remove( t->element, t->right );
    }
    else
    {
        AvlNode *oldNode = t;
        t = ( t->left != nullptr ) ? t->left : t->right;
        delete oldNode;
    }

    balance( t );
}
Splay Trees

- different approach to ensuring $O(\log N)$ behavior for tree operations (searches, insertions, and deletions).
- worst case
  - splay trees operations may take $N$ time
- however, splay trees make slow operations ________________
  - guarantee that $M$ ________________ operations (insertions or deletions) requires at most $O(M \log N)$, so, on average, operations are $O(\log N)$
- $O(\log N)$ is an ________________ complexity
  - derivation is complex
- common for binary search trees to have a sustained sequence of bad accesses
Splay Trees

- basic idea: when a node is accessed, it is moved to the top of the tree, with the thought that we might want to revisit ____________ more frequently
  - use double rotations similar to AVL to move nodes to top of tree
  - along the way, more branching is introduced in the tree, which __________ the height of the tree and thus the cost of tree operations
Splay Trees

- single rotations ______________

- access $k_1$
Splay Trees

A

B

C

D

E

F

A

B

C

D

E

F
Splay Trees
Splay Trees

- double rotations consider parent and ________________ of accessed node
  - zig: single branch (in one direction)
  - zag: secondary branch (in opposite direction)
- when the parent node is the ____________, a single rotation for the zig is sufficient
Splay Trees

- access $X$
- zig-zag
Splay Trees

- access $X$
- zig-zig
- consider tree from previous example
- access $k_1$
Splay Trees

- consider tree from previous example
  - access $k_1$
  - ________________
Splay Trees

- consider tree from previous example
  - access \( k_1 \)

- \( k_1 \) is now at the root
- final tree has \___________\ the distance of most nodes on the access path to the root
Splay Trees

- example 2
  - access 1

- tree starts as __________ case and results in much better structure for performance
Splay Trees

- example 3: tree with only left children – access 1
Splay Trees

example 3: tree with only left children – access 2
Splay Trees

- example 3: tree with only left children – access 3
Splay Trees

- example 3: tree with only left children – access 4
- example 3: tree with only left children – access 5

Splay Trees
Splay Trees

- example 3: tree with only left children – access 6
example 3: tree with only left children – access 7
Splay Trees

- example 3: tree with only left children – access 8
Splay Trees

- example 3: tree with only left children – access 9
Splay Trees

− deleting nodes
  − first, access the node, which moves it to the _________ of the tree
    − let $T_L$ and $T_R$ be the left and right subtrees of the new root
  − find $e$, the ____________ element of $T_L$
  − rotate $e$ to the root of $T_L$
  − since $e$ is the largest element of $T_L$, it will have no _________ child, so we can attach $T_R$ there
    − rather than the largest element of $T_L$, we could use the smallest element of $T_R$ and modify $T_R$
Tree Traversals

- easy to list all elements of a _____________ search tree in sorted order
  - inorder traversal
  - postorder traversal
  - preorder traversal
- implemented with ________________ functions
- all $O(N)$
inorder traversal

```cpp
/**
 * Print the tree contents in sorted order.
 */
void printTree( ostream & out = cout ) const
{
    if( isEmpty( ) )
        out << "Empty tree" << endl;
    else
        printTree( root, out );
}

/**
 * Internal method to print a subtree rooted at t in sorted order.
 */
void printTree( BinaryNode *t, ostream & out ) const
{
    if( t != nullptr )
    {
        printTree( t->left, out );
        out << t->element << endl;
        printTree( t->right, out );
    }
}
Tree Traversals

- preorder traversal
  - visit node first, then left subtree, then right subtree
- postorder traversal
  - visit left subtree, right subtree, then node
- graphic technique for traversals
- level-order traversal
  - all nodes at depth \( d \) are processed before any node at depth \( d + 1 \)
    - not implemented with _______________
  - queue
Tree Traversals

- height method using ________________ traversal

```c
/**
 * Internal method to compute the height of a subtree rooted at t.
 */
int height( BinaryNode *t )
{
    if ( t == nullptr )
        return -1;
    else
        return 1 + max( height( t->left ), height( t->right ) );
}
```
Top-Down Splay Trees

- previous method requires traversal from root down to node, then a bottom-up traversal to implement the splaying
  - can be accomplished by maintaining ______________ links
  - or by storing ______________________ on the stack
- both methods require substantial ______________________
- both must handle a variety of special cases
Top-Down Splay Trees

- instead, perform rotations on ________________
  - result is faster
  - uses extra space \( O(1) \)
  - retains amortized time bound of \( O(\log N) \)
- suppose we wish to access key $i$
- during the access and concurrent splaying operation, the tree is broken into __________ parts
  - a left tree, which contains all the keys from the original tree known at the time to be __________ than $i$
  - a right tree, which contains all the keys from the original tree known at the time to be ________________ than $i$
  - a middle tree, which consists of the subtree of the original tree rooted at the current node on the access path

- initially, the left and right trees are empty and the middle tree is the entire tree
- at each step we tack bits of the middle tree onto the left and right trees
Top-Down Splay Trees

- rotations for zig, zig-zig, and zig-zag cases
Top-Down Splay Trees

- zig-zag case can be simplified to just a _______ since no rotations are performed
- instead of making $Z$ the root, we make $Y$ the root

- simplifies coding, but only descends one level
  - requires more ________________
Top-Down Splay Trees

- after final splaying
Top-Down Splay Trees

- example: access 19
Top-Down Splay Trees

- example: access 19
Top-Down Splay Trees

- example: access 19

```
12
  5
  15
  13

18
  16

20
  25
    24
    30

reassemble
```

```
18
  12
    5
    15
    13
  20
    25
      24
      30
```
Top-Down Splay Trees

- use a ________ to hold the roots of the left and right subtrees
  - left pointer will contain root of right subtree
  - right pointer will contain root of left subtree
  - easy to __________________ at end of splaying
Top-Down Splay Trees

- example 2: access 14
example 2: access 14
- start at root and look down two nodes along path to 14
Top-Down Splay Trees

- example 2: access 14
Example 2: Access 14
Continuing down the tree, this is a ____________ condition
Top-Down Splay Trees

- example 2: access 14
- tree is reconfigured
Top-Down Splay Trees

- example 2: access 14
- simple zig
Top-Down Splay Trees

- example 2: access 14
- move accessed node to _________ and reassemble tree

```
    14
   /   
 11    21
    /  
   A    19
      /  
     B   20
        /  
        28 42
          /    /
         35  54
            /  
           49
```
- B-trees were developed in the late 1960s by Rudolf Bayer and Edward McCreight:


- originally motivated by applications in ________________
- B-trees shown here really B+ tree
thus far, we have frequently treated the key as if it were the _________ being stored, but that is rarely the case.

example: student records in Banner

most effective search key is W&M ID (e.g., 930…) since it is unique

the record (value) associated with each key contains much more information

- Student Information
- Student Academic Transcript
- Student Active Registrations
- Student Schedule
- Student E-mail Address
- Student Address and Phones ...
B-Trees

- B-trees are particularly useful when we cannot fit all of our data in ____________, but have to perform reads and writes from secondary storage (e.g., disk drives)
- disk accesses incredibly ____________, relatively speaking
- consider a disk drive that rotates at 7200 rpm
  - the rotational speed plays a role in retrieval time; for a 7200 rpm disk, each revolution takes $60/7200 = 1/120$ s, or about 8.3 ms
  - a typical seek time (the time for the disk head to move to the location where data will be read or written) for 7200 rpm disks is around 9 ms
- this means we can perform 100-120 random disk accesses per second
B-Trees

- meanwhile, our CPU can perform > 1,000,000,000 operations per second
- suppose we have a database with \( N = 10,000,000 \) entries that we organize in a tree
  - in an _______ tree, a worst-case search requires \( 1:44 \lg N \approx 33 \) disk accesses
  - at 9 ms per access, this requires about 300 ms, so on average we can perform less than 4 searches per second
- we would expect 1000 worst-case searches to take 300,000 ms = 300 s, or about ________________
- in this application, search trees with height \( \lg N \) are still too high!
- Height can be reduced if we allow more ________________
- Binary search trees only allow 2-way branching
- Example: 5-ary 31-node tree with height 3
B-Trees

- B-tree of order $M$ is an $M$-ary tree with the following properties
  1. data items are stored at _____________
  2. nonleaf nodes (internal nodes) store up to $M - 1$ keys to guide the searching: key $i$ represents the _____________ key in subtree $i + 1$
  3. root is either a _____ or has between two and $M$ children
  4. all nonleaf nodes (except the root) have between $\lceil M/2 \rceil$ and $M$ children
  5. all leaves are at the same depth and have between $\lceil L/2 \rceil$ and $L$ data items, for some $L$
B-Trees

- examples
  - for $M = 2$, there are between $\lfloor 2/2 \rfloor = 1$ to 2 children
  - for $M = 3$, there are between $\lfloor 3/2 \rfloor = 2$ to 3 children
  - for $M = 4$, there are between $\lfloor 4/2 \rfloor = 2$ to 4 children
  - for $M = 5$, there are between $\lfloor 5/2 \rfloor = 3$ to 5 children
  - for $M = 42$, there are between $\lfloor 42/2 \rfloor = 21$ to 42 children
  - requiring nodes to be ____________ guarantees that the tree will not degenerate into a simple binary search tree
- examples: $M = 5$

- all nonleaf nodes have between 3 and 5 children (and thus between 2 and 4 keys)
- root could have just 2 children
- here $L$ is also 5: each leaf has between 3 and 5 data items
− choosing $M$ and $L$
− each node will occupy a disk block, say 8192 bytes, so we choose $M$ and $L$ based on the _________ of the items being stored
− suppose each key uses 32 bytes and a ______ to another node uses 8 bytes
− a node in a B-tree of order $M$ has $M-1$ keys and $M$ links, so a node requires

$$32(M - 1) + 8M = 40M - 32 \text{ bytes}$$

− we choose the __________ $M$ that will allow a node to fit in a block

$$M = \left\lfloor \frac{8192 + 32}{40} \right\rfloor = 205$$


B-Trees

- choosing $M$ and $L$ (cont.)
  
  - if the values are each 256 bytes, then we can fit

  $L = \left\lfloor \frac{8192}{256} \right\rfloor = 32$

  in a single ___________

  - each ______ has between 16 and 32 values, and each internal node branches in at least 103 ways

  - if there are 1,000,000,000 values to store, there are at most 62,500,000 leaves

  - the leaves would be, in the worst case, on level

    $1 + \log_{103} 62,500,000 = 5$

    so we can find data in at most 5 disk access

  - a BST would have at least $1 + \log_2 62,500,000 = 27$ levels!
B-Trees

- insertion: easy case – insert 57
  - first, follow the search tree to the correct leaf (external node)
  - if there are fewer than $L$ items in the leaf, insert in the correct location
  - cost: 1 disk access

![B-Tree Diagram]

1. Insertion Easy Case: Example Insertion of 57.
B-Trees

- insertion: splitting a leaf – insert 55
  - if there are already $L$ items in the ___________
    - add the new item, split the node in two, and update the links in the parent node
  - cost: 3 disk accesses (one for each new node and one for the update of the ______________ node)
B-Trees

- insertion: splitting a leaf – insert 55 (cont.)
- the splitting rule ensures we still have a ___________: each new node has at least \( \lfloor L/2 \rfloor \) values (e.g., if \( L = 3 \), there are 2 values in one node and 1 in the other, and if \( L = 4 \), each new node has 2 keys)
B-Trees

− insertion: splitting a parent – insert 40
  − what if the parent node already has all the child nodes it can possibly have?
    − split the ___________ node, and update its parent
    − repeat until we arrive at the ___________
  − if necessary, split the root into two nodes and create a new root with the two nodes as ___________
    − this is why the root is allowed as few as 2 children
    − thus, a B-tree grows at the root
B-Trees

− insertion: splitting a parent – insert 40 (cont.)
B-Trees

- insertion: other techniques – insert 29
  - put a child up for adoption if a _______________ has room
  - here, move 32 to the next leaf
  - modifies parent, but keeps nodes _____________ and saves space in the long run
B-Trees

deletion: delete 99
  - could bring leaf below ____________ number of data items
  - adopt neighboring item if neighbor not at minimum
  - otherwise, ____________ with neighbor to form a full leaf
  - process could make its way up to the root
    - if root left with 1 child, remove root and make its child the new root of the tree
−deletion: delete 99 (cont.)