Chapter 4
Trees

1

Introduction

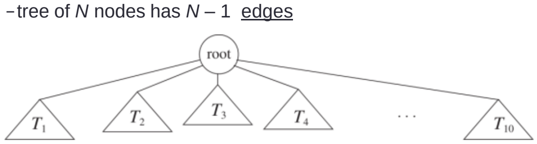
- for large input, even <u>linear</u> access time may be prohibitive
  - we need data structures that exhibit <u>average</u> running times closer to $O(\log N)$
  - binary search tree

2

2

Terminology

- recursive definition of **tree**
  - collection of nodes (may be <u>empty</u>)
  - 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
- <u>root</u> of each subtree is a **child** of r
- r is the **parent** of each subtree
- tree of N nodes has $N – 1$ <u>edges</u>

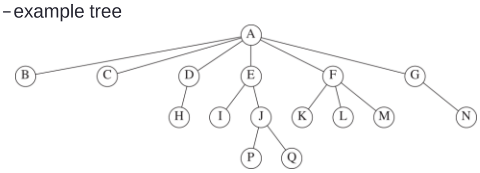


3

3

Terminology

- example tree



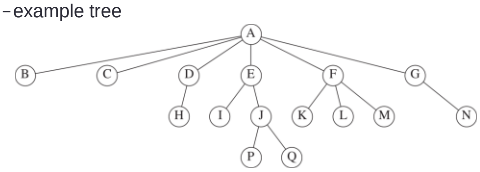
- nodes with no <u>children</u> are called **leaves** (e.g., B, C, H, I, P, Q, K, L, M, N)
- nodes with the same <u>parent</u> are **siblings** (e.g., K, L, M)
- **parent**, **grandparent**, **grandchild, ancestor, descendant, proper ancestor, proper descendant**

4

4

Terminology

- example tree



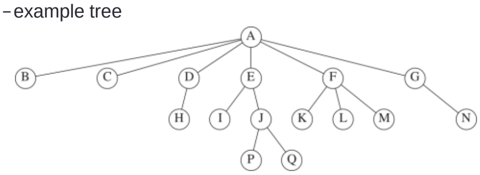
- **path** from n_1 to n_k is a sequence of nodes n_1, n_2, …, n_k where n_1 is the parent of n_{i+1} for $1 <= i < k$
- **length** of path is number of <u>edges</u> on path ($k – 1$)
  - path of length 0 from every node to <u>itself</u>
  - exactly one <u>path</u> from the root to each node

5

5

Terminology

- example tree



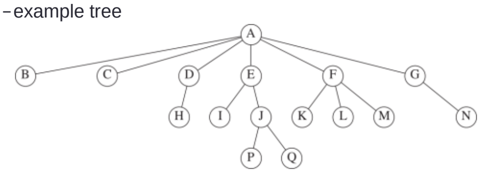
- **depth** from n_i is the length of the unique path from the <u>root</u> to n_i
  - <u>root</u> is at depth 0
- **height** of n_i is the length of the <u>longest</u> path from n_i to a leaf
  - all <u>leaves</u> at height 0
  - height of the tree is equal to the height of the root

6

6

Terminology

- example tree



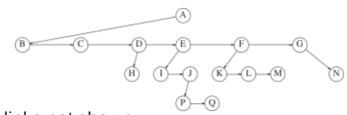
- E is at depth 1 and height 2
- F is at depth 1 and height 1
- depth of tree is 3

7

Implementation of Trees

- each node could have data and a link to each child
  - number of children is unknown and may be large, which could lead to wasted space
  - instead, keep children in a linked list

```
struct TreeNode
{
    Object    element;
    TreeNode *firstChild;
    TreeNode *nextSibling;
};
```

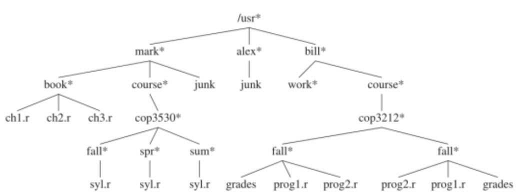


  - null links not shown

8

Tree Traversals with Application

- many applications for trees
  - subdirectory structure in Unix
  - pathname built into tree



9

Tree Traversals with Application

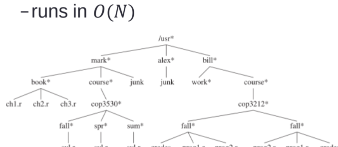
- goal: list all files in a directory
  - depth denoted by tabs
  - begins at root

```
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 );
}
```

10

Tree Traversals with Application

- code prints directories/files in preorder traversal
  - runs in $O(N)$

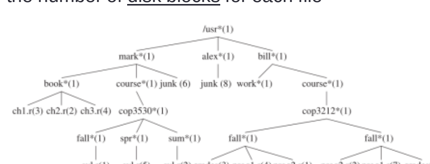


```
/usr
    mark
        book
            ch1.r
            ch2.r
            ch3.r
        course
            cop3530
                fall
                    syl.r
                spr
                    syl.r
                sum
                    syl.r
        junk
    alex
        junk
    bill
        work
        course
            cop3212
                fall
                    grades
                    prog1.r
                    prog2.r
                fall
                    prog2.r
                    prog1.r
                    grades
```

11

Tree Traversals with Application

- for postorder traversal, numbers in parentheses represent the number of disk blocks for each file



12

Tree Traversals with Application

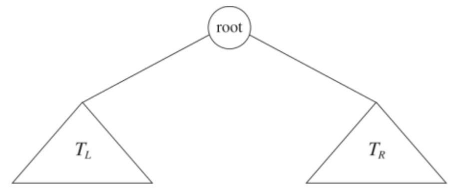
- size method to find number of blocks for each file
  - directories use 1 block of space

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

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

     return totalSize;
}
```

```
            ch1.r          3
            ch2.r          2
            ch3.r          4
        book              10
                  syl.r    1
              fall         2
                  syl.r    5
              spr          6
                  syl.r    2
              sum          3
            cop3530       12
        course            13
        junk               6
    mark                  30
        junk               8
    alex                   9
        work               1
                  grades   3
                  prog1.r  4
                  prog2.r  1
              fall         9
                  prog2.r  2
                  prog1.r  7
                  grades   9
              fall        19
            cop3212       29
        course            30
    bill                  32
/usr                      72
```

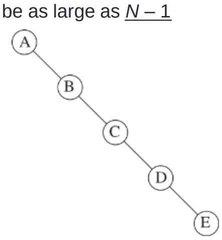

Binary Trees

- in binary trees, nodes can have no more than two children
- binary tree below consists of a root and two subtrees, T_L and T_R, both of which could possibly be empty



Binary Trees

- depth of a binary tree is considerably smaller than N
- average depth is $O(\sqrt{N})$
- average depth for a binary search tree is $O(\log N)$
  - depth can be as large as $N - 1$



Binary Tree Implementation

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

```
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

Tree Traversals

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

Tree Traversals

- inorder traversal

```
/**
 * 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 <u>recursion</u>
    - queue

19

Tree Traversals

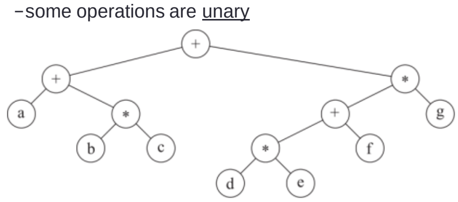
- height method using <u>postorder</u> traversal

```
/**
 * 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 ) );
}
```

20

Binary Tree Example: Expression Trees

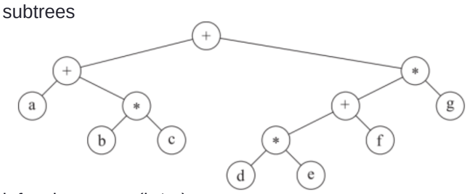
- expression tree
  - leaves represent <u>operands</u> (constants or variable names)
  - interior nodes represent <u>operators</u>
  - binary tree since most operators are binary, but not required
    - some operations are <u>unary</u>



21

Binary Tree Example: Expression Trees

- evaluate expression tree, T, by applying operator at root to values obtained by <u>recursively</u> evaluating left and right subtrees

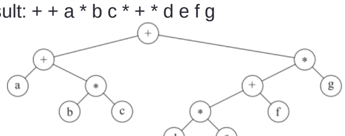


- left subtree: a + (b * c)
- right subtree: ((d * e) + f) * g
- <u>complete</u> tree: (a + (b * c)) + (((d * e) + f) * g)

22

Binary Tree Example: Expression Trees

- <u>inorder</u> traversal
  - recursively produce left expression
  - print operator at root
  - recursively produce right expression
- <u>postorder</u> traversal
  - result: a b c * + d e * f + g * +
- <u>preorder</u> traversal
  - result: + + a * b c * + * d e f g



23

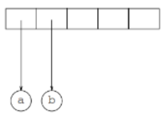
Binary Tree Example: Expression Trees

- goal: convert a postorder expression into an expression tree
  - read expression one <u>symbol</u> at a time
  - if <u>operand</u>, create node and push a pointer to it on the stack
  - if <u>operator</u>, 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

24

Binary Tree Example: Expression Trees

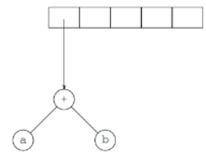
- example: a b + c d e + * *
- first two symbols are <u>operands</u> and are pushed on the stack



25

Binary Tree Example: Expression Trees

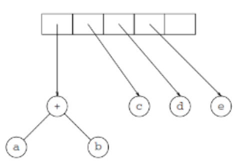
- example: a b + c d e + * *
- after + is read, two pointers are <u>popped</u> and new tree formed with a pointer pushed on the stack



26

Binary Tree Example: Expression Trees

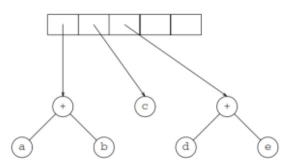
- example: a b + c d e + * *
- next, c, d, and e are read, with <u>one-node</u> tree created for each and pushed on the stack



27

Binary Tree Example: Expression Trees

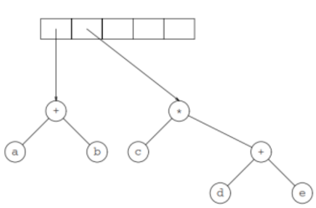
- example: a b + c d e + * *
- after + is read, two trees are <u>merged</u>



28

Binary Tree Example: Expression Trees

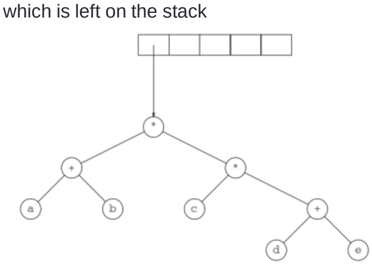
- example: a b + c d e + * *
- after * is read, two trees are popped to form a new tree with a * as root



29

Binary Tree Example: Expression Trees

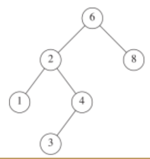
- example: a b + c d e + * *
- finally * is read, two trees are popped to form a <u>final</u> tree, which is left on the stack

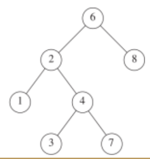


30

Binary Search Tree ADT

- binary trees often used for <u>searching</u>
- assume each node in the tree stores one <u>element</u> (integer)
- binary search tree
  - for every node X in the tree
    - all items in left subtree are <u>smaller</u> than X
    - all items in right subtree are greater than X
  - items in tree must be order-able





31

Binary Search Tree ADT

- common operations on binary search trees
  - often written <u>recursively</u>
  - 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 <u>root pointer</u>

32

Binary Search Tree ADT

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


33

Binary Search Tree ADT

```
  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;
};
```


34

Binary Search Tree ADT

- test for item in subtree

```
/**
 * 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 == nullptr )
        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
}
```


35

Binary Search Tree ADT

- **`findMin`** and **`findMax`**
  - <u>private</u> methods return pointer to smallest/largest elements in the tree
  - to find the minimum, start at the root and go <u>left</u> as long as possible
  - similar for finding the maximum

36

Binary Search Tree ADT

- recursive version of **findMin**

```
/**
 * Internal method to find the smallest item in a subtree t.
 * Return node containing the smallest item.
 */
BinaryNode * findMin( BinaryNode *t ) const
{
    if( t == nullptr )
        return nullptr;
    if( t->left == nullptr )
        return t;
    return findMin( t->left );
}
```

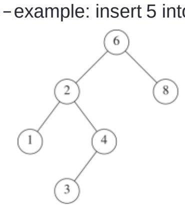

Binary Search Tree ADT

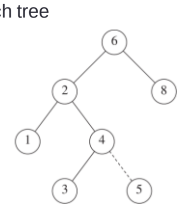
- nonrecursive version of **findMax**

```
/**
 * 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 down the tree, as in the **contains** function
  - if X is found, do nothing
  - otherwise, insert X at the last spot on the path traversed
  - example: insert 5 into binary search tree



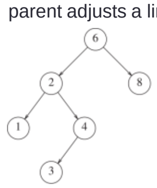


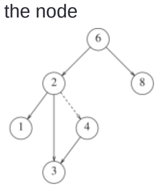
Binary Search Tree ADT

- duplicates can be handled by adding a count to the node record
  - better than inserting duplicates in tree
  - may not work well if key is only small part of larger structure

Binary Search Tree ADT

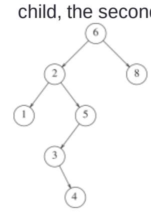
- deletion in binary search tree may be difficult
- multiple cases
  - if node is leaf, it can be deleted immediately
  - if node has only one child, node can be deleted after its parent adjusts a link to bypass the node

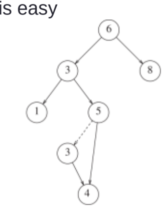




Binary Search Tree ADT

- multiple cases (cont.)
  - complicated case: node with two 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





Binary Search Tree ADT

- if number of deletions small, lazy deletion may be used
  - node is marked deleted rather than actually being deleted
  - especially popular if duplicates allowed
    - count of duplicates can be decremented
  - incurs only small penalty on tree since height not affected greatly
  - if deleted node reinstated, some benefits

43

Binary Search Tree Average-Case Analysis

- we expect most operations on binary search trees will have $O(\log N)$ time
  - average depth over all nodes can be shown to be $O(\log N)$
  - all insertions and deletions must be equally likely
  - sum of the depths of all nodes in a tree is known as internal path length

44

Binary Search Tree Average-Case Analysis

- the run time of binary search trees depends on the depth of the tree, which in turn depends on the order 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 $0 \le i \le N$
- total number of nodes in tree = left subtree + right subtree + 1
- all nodes except the root are one level deeper,

$$D(N) = D(i) + D(N - i - 1) + N - 1$$

45

Binary Search Tree Average-Case Analysis

- if all subtree sizes are equally likely, 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

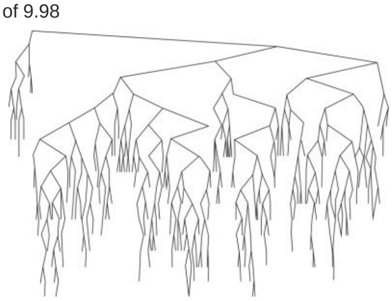
$$D(N) = O(N \log N)$$

and the average number of nodes is $O(\log N)$

46

Binary Search Tree Average-Case Analysis

- example: randomly generated 500-node tree has expected depth of 9.98



47

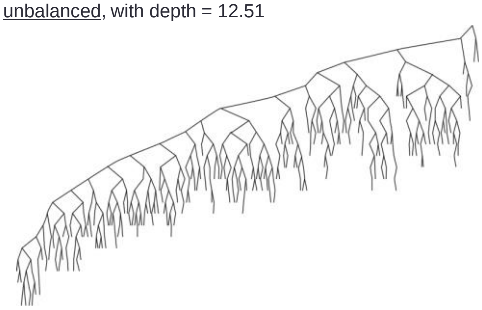
Binary Search Tree Average-Case Analysis

- deletions, however, bias the left subtrees to be longer because we always replace a deleted node with a node from the right subtree
- exact effect of deletions still unknown
- if insertions and deletions are alternated $\Theta(N^2)$ times, then expected depth is $\Theta(\sqrt{N})$

48

Binary Search Tree Average-Case Analysis

- after 250,000 random insert/delete pairs, tree becomes unbalanced, with depth = 12.51



49

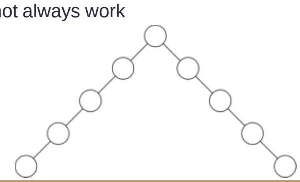
Binary Search Tree Average-Case Analysis

- could randomly choose between smallest element in the right subtree and largest element in the left subtree when replacing deleted element
  - should keep bias low, but not yet proven
- bias does not show up for small 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

50

AVL Trees

- Adelson-Velskii and Landis (AVL) tree is a binary search tree with a balance condition
- balance condition in general
  - must be easy to maintain
  - ensures depth of tree is $O(\log N)$
- simplest idea: left and right subtrees have the same height
  - does not always work



51

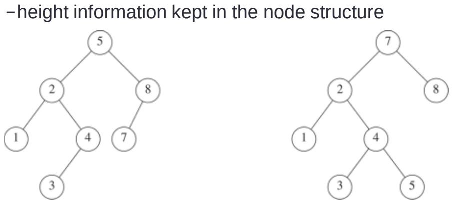
AVL Trees

- alternate balance condition: every node must have left and right subtrees of the same height
  - only perfectly balanced trees of $2^k - 1$ nodes would work
  - condition too rigid

52

AVL Trees

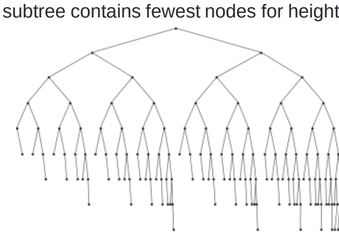
- AVL tree
  - for each node in the tree, height of left and right subtrees differ by at most 1
    - height balance = height of right subtree – height of left
    - height of an empty tree: -1
    - height information kept in the node structure



53

AVL Trees

- example AVL tree
  - fewest nodes for a tree of height 9
  - left subtree contains fewest nodes for height 7
  - right subtree contains fewest nodes for height 8



54

AVL Trees

- <u>minimum</u> number of nodes, $S(h)$, in an AVL tree of height h

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

  - closely related to <u>Fibonacci</u> numbers
- all operations can be performed in $O(\log N)$ time, except <u>insertion and deletion</u>

55

AVL Trees

- insertion
  - update all balance information in the nodes on the path back to the root
  - could violate the balance condition
  - <u>rotations</u> used to restore the balance property
- deletion
  - perform same promotion as in a <u>binary</u> search tree, updating the balance information as necessary
    - same balancing operations for <u>insertion</u> can then be used

56

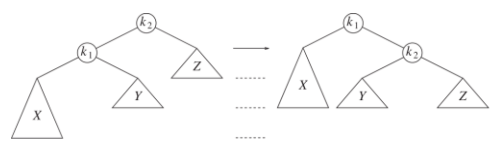
AVL Trees

- if α is the node requiring <u>rebalancing</u> (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 α
  - an insertion into the right subtree of the left child of α
  - an insertion into the left subtree of the right child of α
  - an insertion into the right subtree of the right child of α
- cases 1 and 4 are mirror image symmetries with respect to α and can be resolved with a <u>single</u> rotation
- cases 2 and 3 are mirror image symmetries with respect to α and can be resolved with a <u>double</u> rotation

57

AVL Trees

- single rotation
  - only possible case 1 scenario
  - to balance, imagine "picking up" tree by k_1

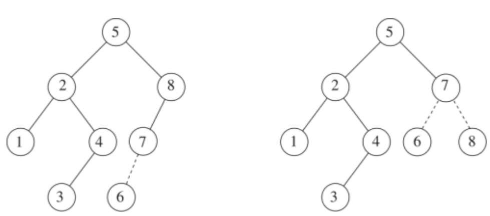


  - new tree has <u>same height</u> as original tree

58

AVL Trees

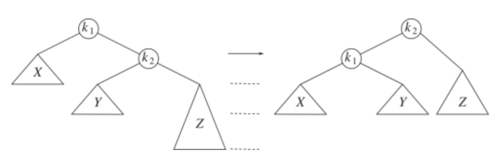
- example tree
  - when adding 6, node 8 becomes <u>unbalanced</u>
  - to balance, perform single rotation between 7 and 8



59

AVL Trees

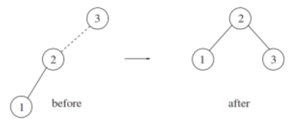
- example tree
  - <u>symmetric</u> case for case 4



60

AVL Trees

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



61

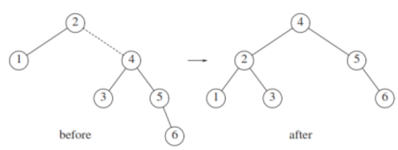
AVL Trees

- example
  - insert 4 and 5

62

AVL Trees

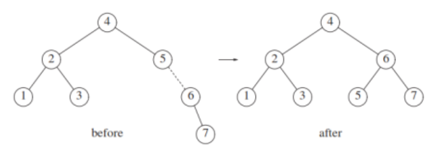
- example
  - insert 6



63

AVL Trees

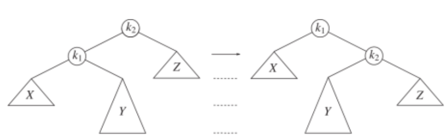
- example
  - insert 7



64

AVL Trees

- double rotation
  - for cases 2 and 3, a <u>single</u> rotation will not work

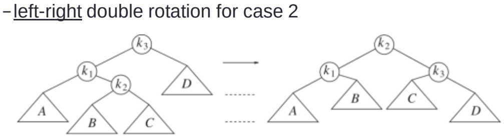


  - tree Y can be expanded to a node with two <u>subtrees</u>

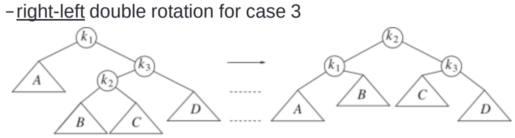
65

AVL Trees

- double rotation
  - <u>left-right</u> double rotation for case 2



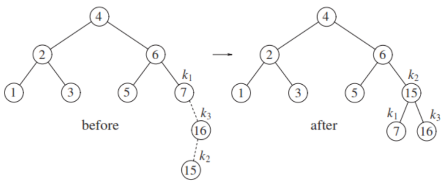
  - <u>right-left</u> double rotation for case 3



66

AVL Trees

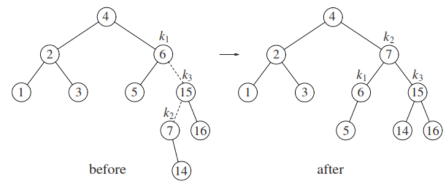
- example
  - insert 16 and 15



67

AVL Trees

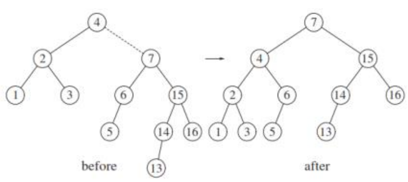
- example
  - insert 14



68

AVL Trees

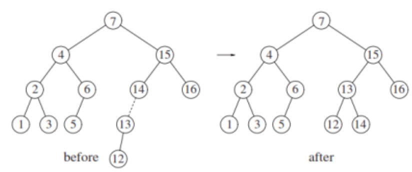
- example
  - insert 13



69

AVL Trees

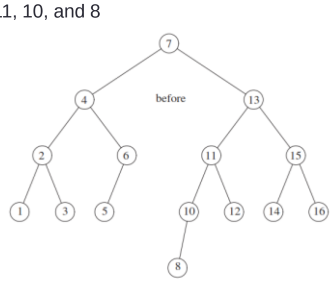
- example
  - insert 12



70

AVL Trees

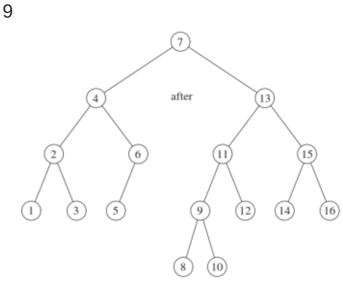
- example
  - insert 11, 10, and 8



71

AVL Trees

- example
  - insert 9



72

AVL Trees

- implementation
  - node definition

```
struct AvlNode
{
    Comparable element;
    AvlNode   *left;
    AvlNode   *right;
    int       height;

    AvlNode( const Comparable & ele, AvlNode *lt, AvlNode *rt, int h = 0 )
      : element{ ele }, left{ lt }, right{ rt }, height{ h } { }

    AvlNode( Comparable && ele, AvlNode *lt, AvlNode *rt, int h = 0 )
      : element{ std::move( ele ) }, left{ lt }, right{ rt }, height{ h } { }
};
```

73

AVL Trees

- implementation
  - function to compute height of AVL node

```
/**
 * Return the height of node t or -1 if nullptr.
 */
int height( AvlNode *t ) const
{
    return t == nullptr ? -1 : t->height;
}
```

74

AVL Trees

- implementation
  - insertion

```
/**
 * 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 );
}
```

75

AVL Trees

- implementation

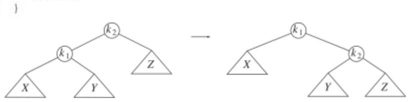
```
static const int ALLOWED_IMBALANCE = 1;

// Assume t is balanced or within one of being balanced
void balance( AvlNode * & t )
{
    if( t == nullptr )
        return;

    if( height( t->left ) - height( t->right ) > ALLOWED_IMBALANCE )
        if( height( t->left->left ) >= height( t->left->right ) )
            rotateWithLeftChild( t );
        else
            doubleWithLeftChild( t );
    else
    if( height( t->right ) - height( t->left ) > ALLOWED_IMBALANCE )
        if( height( t->right->right ) >= height( t->right->left ) )
            rotateWithRightChild( t );
        else
            doubleWithRightChild( t );

    t->height = max( height( t->left ), height( t->right ) ) + 1;
}
```

76

AVL Trees

- implementation
  - single rotation

```
/**
 * 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;
}
```

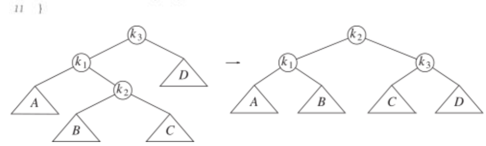


77

AVL Trees

- implementation
  - double rotation

```
/**
 * 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 );
}
```



78

AVL Trees

- implementation
  - deletion

```
/**
 * 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 );
}
```

79

79

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 infrequently
  - guarantee that M consecutive operations (insertions or deletions) requires at most $O(M \log N)$, so, on average, operations are $O(\log N)$
  - $O(\log N)$ is an amortized complexity
    - derivation is complex
  - common for binary search trees to have a sustained sequence of bad accesses

80

80

Splay Trees

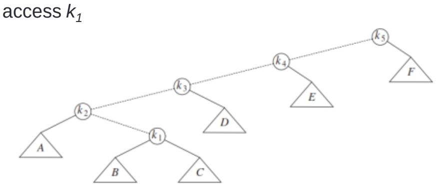
- basic idea: when a node is accessed, it is moved to the top of the tree, with the thought that we might want to revisit recently accessed nodes more frequently
  - use double rotations similar to AVL to move nodes to top of tree
  - along the way, more branching is introduced in the tree, which reduces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

81

81

Splay Trees

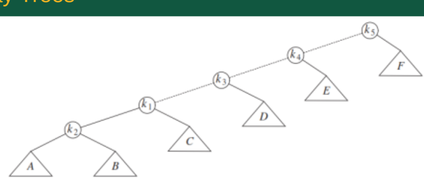
- single rotations don't work
  - access k_1

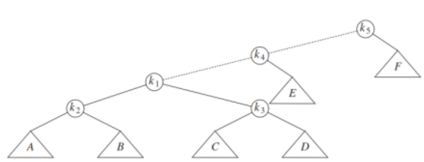


82

82

Splay Trees

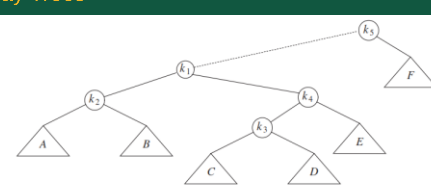


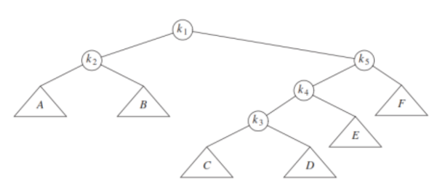


83

83

Splay Trees





84

84

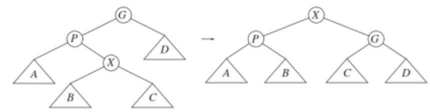
Splay Trees

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

85

Splay Trees

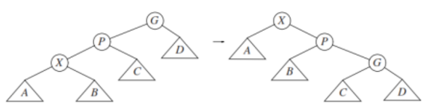
- access *X*
  - zig-zag



86

Splay Trees

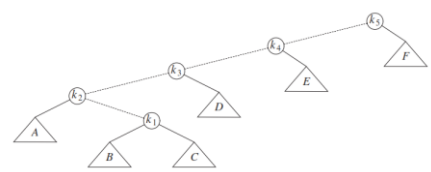
- access *X*
  - zig-zig



87

Splay Trees

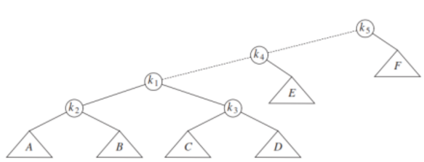
- consider tree from previous example
  - access k_1
  - zig-zag



88

Splay Trees

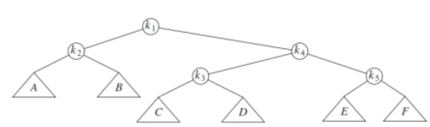
- consider tree from previous example
  - access k_1
  - zig-zig



89

Splay Trees

- consider tree from previous example
  - access k_1



  - k_1 is now at the root
  - final tree has halved the distance of most nodes on the access path to the root

90

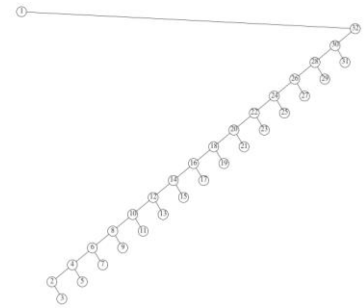
Splay Trees

- example 2
  - access 1

  - tree starts as <u>worst</u> case and results in much better structure for performance

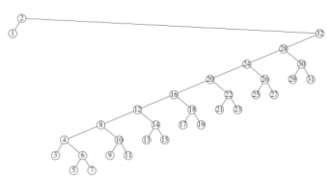
Splay Trees

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



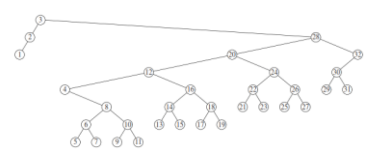
Splay Trees

- example 3: tree with only left children – access 2



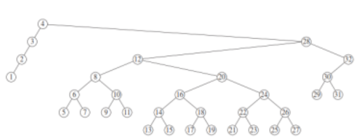
Splay Trees

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



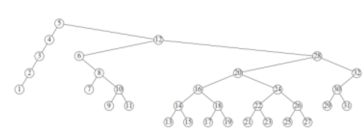
Splay Trees

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



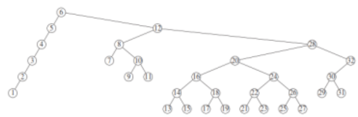
Splay Trees

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



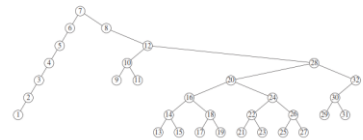
Splay Trees

– example 3: tree with only left children – access 6



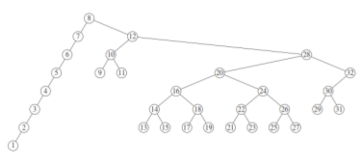
Splay Trees

– example 3: tree with only left children – access 7



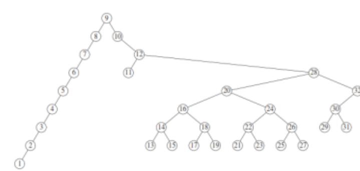
Splay Trees

– 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 <u>root</u> of the tree
    - let T_L and T_R be the left and right subtrees of the new root
  - find *e*, the <u>largest</u> 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 <u>right</u> 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

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 <u>parent</u> links
  - or by storing <u>access path</u> on the stack
  - both methods require substantial <u>overhead</u>
  - both must handle a variety of special cases

Top-Down Splay Trees

- instead, perform rotations on <u>initial access path</u>
  - result is faster
  - uses extra space $O(1)$
  - retains amortized time bound of $O(\log N)$

103

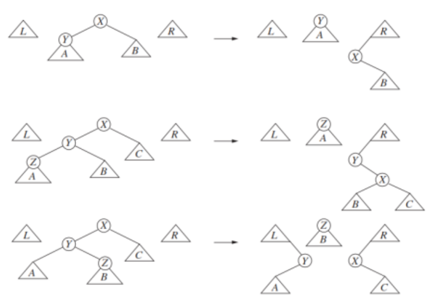
Top-Down Splay Trees

- suppose we wish to access key i
- during the access and concurrent splaying operation, the tree is broken into <u>three</u> parts
  - a left tree, which contains all the keys from the original tree known at the time to be <u>less</u> than i
  - a right tree, which contains all the keys from the original tree known at the time to be <u>greater</u> 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

104

Top-Down Splay Trees

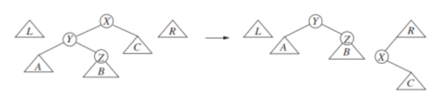
- rotations for zig, zig-zig, and zig-zag cases



105

Top-Down Splay Trees

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

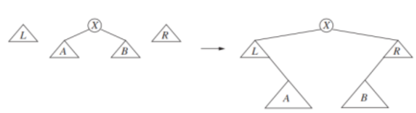


- simplifies coding, but only descends one level
  - requires more <u>iterations</u>

106

Top-Down Splay Trees

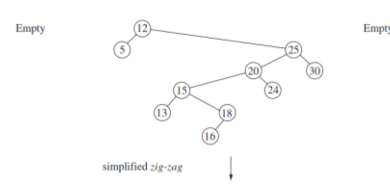
- after final splaying



107

Top-Down Splay Trees

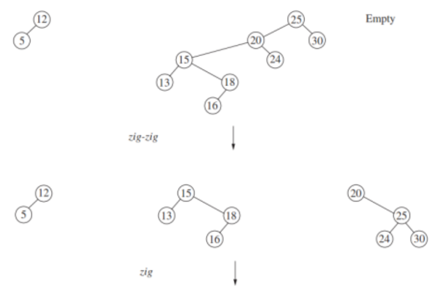
- example: access 19



108

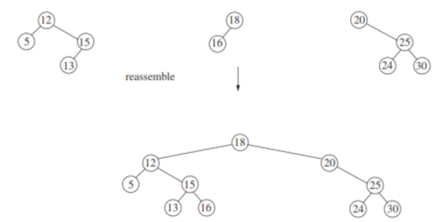
Top-Down Splay Trees

– example: access 19



Top-Down Splay Trees

– example: access 19

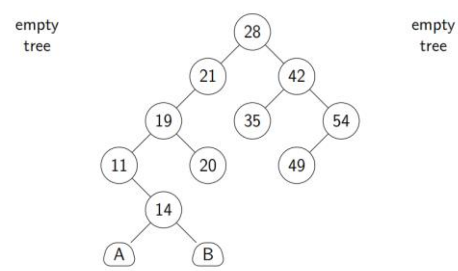


Top-Down Splay Trees

– use a <u>header</u> 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 <u>reconstruct</u> at end of splaying

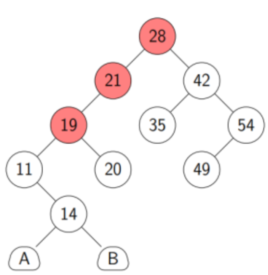
Top-Down Splay Trees

– example 2: access 14



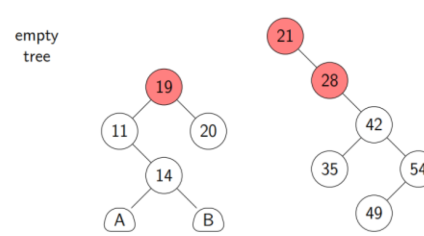
Top-Down Splay Trees

– example 2: access 14
– start at root and look down two nodes along path to 14



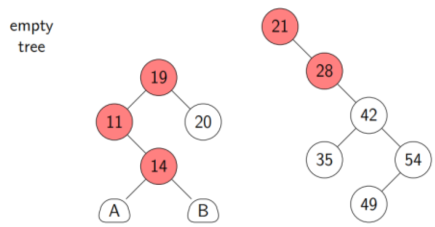
Top-Down Splay Trees

– example 2: access 14



Top-Down Splay Trees

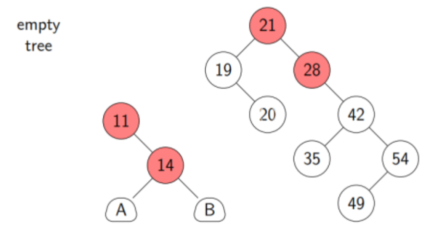
- example 2: access 14
- continuing down the tree, this is a zig-zag condition



115

Top-Down Splay Trees

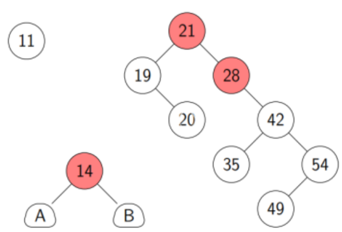
- example 2: access 14
- tree is reconfigured



116

Top-Down Splay Trees

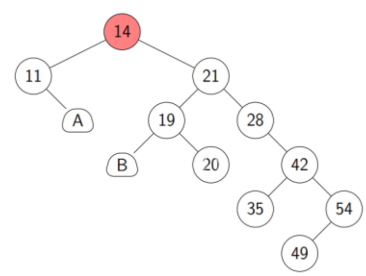
- example 2: access 14
- simple zig



117

Top-Down Splay Trees

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



118

B-Trees

- B-trees were developed in the late 1960s by Rudolf Bayer and Edward McCreight:

  R. Bayer and E. McCreight, *Organization and maintenance of large ordered indexes*, Acta Informatica vol. 1, no. 3 (1972), pp. 173-189.

  - originally motivated by applications in databases
  - B-trees shown here really B+ tree

119

B-Trees

- thus far, we have frequently treated the key as if it were the data 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 ...

120

B-Trees

- B-trees are particularly useful when we cannot fit all of our data in memory, but have to perform reads and writes from secondary storage (e.g., disk drives)
  - disk accesses incredibly expensive, 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

121

121

B-Trees

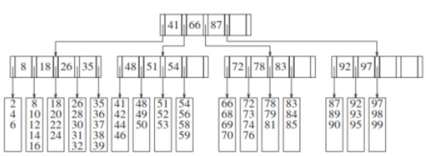
- meanwhile, our CPU can perform > 1,000,000,000 operations per second
- suppose we have a database with N = 10,000,000 entries that we organize in a tree
  - in an AVL tree, a worst-case search requires 1:44 lg N ≈ 33 disk accesses
  - at 9 ms per access, this requires about 300 ms, so on average we can perform less than 4 searches per second
  - we would expect 1000 worst-case searches to take 300,000 ms = 300 s, or about 5 minutes
  - in this application, search trees with height $\lg N$ are still too high!

122

122

B-Trees

- height can be reduced if we allow more branching
  - binary search trees only allow 2-way branching
  - example: 5-ary 31-node tree with height 3



123

123

B-Trees

- B-tree of order M is an M-ary tree with the following properties
  1. data items are stored at leaves
  2. nonleaf nodes (internal nodes) store up to $M - 1$ keys to guide the searching: key i represents the smallest key in subtree $i + 1$
  3. root is either a leaf 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

124

124

B-Trees

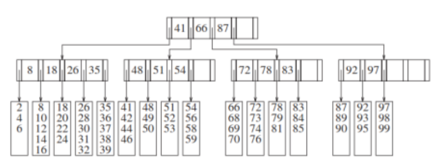
- examples
  - for $M = 2$, there are between $\lceil 2/2 \rceil$ = 1 to 2 children
  - for $M = 3$, there are between $\lceil 3/2 \rceil$ = 2 to 3 children
  - for $M = 4$, there are between $\lceil 4/2 \rceil$ = 2 to 4 children
  - for $M = 5$, there are between $\lceil 5/2 \rceil$ = 3 to 5 children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

125

125

B-Trees

- 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

126

126

B-Trees

- choosing M and L
  - each node will occupy a disk block, say 8192 bytes, so we choose M and L based on the size of the items being stored
  - suppose each key uses 32 bytes and a link 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 largest M that will allow a node to fit in a block

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

127

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 block
  - each leaf 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

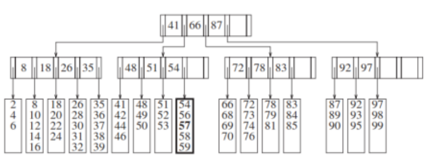
$$1+\log_{103} 62{,}500{,}000 = 5$$

  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!

128

B-Trees

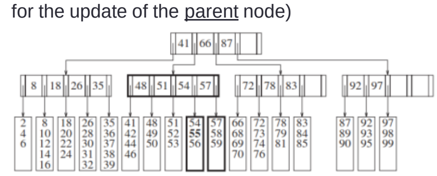
- insertion: easy case – insert 57
  - first, follow the search tree to the correct leaf (external node)
  - if there are fewer than L items in the leaf, insert in the correct location
    - cost: 1 disk access
- insert 55?



129

B-Trees

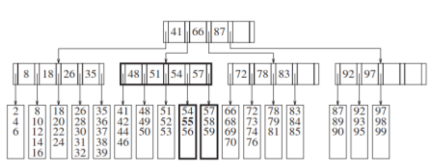
- insertion: splitting a leaf – insert 55
  - if there are already L items in the leaf
    - add the new item, split the node in two, and update the links in the parent node
    - cost: 3 disk accesses (one for each new node and one for the update of the parent node)



130

B-Trees

- insertion: splitting a leaf – insert 55 (cont.)
  - the splitting rule ensures we still have a B-tree: each new node has at least $\lceil L/2 \rceil$ 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)



131

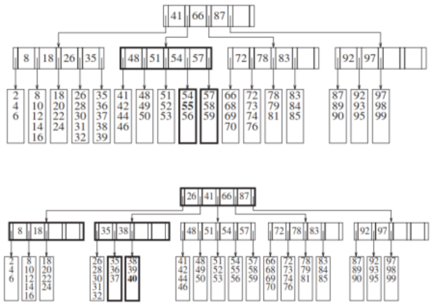
B-Trees

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

132

B-Trees

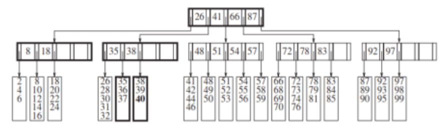
- insertion: splitting a parent – insert 40 (cont.)



133

B-Trees

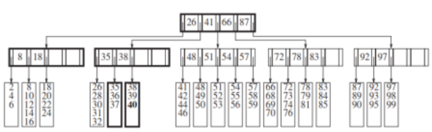
- insertion: other techniques – insert 29
  - put a child up for adoption if a <u>neighbor</u> has room
  - here, move 32 to the next leaf
  - modifies parent, but keeps nodes <u>fuller</u> and saves space in the long run



134

B-Trees

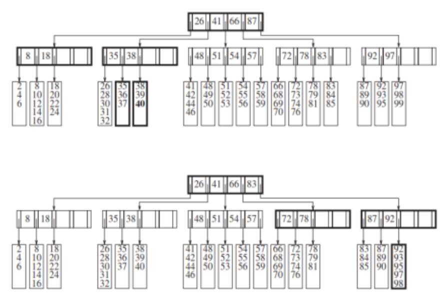
- deletion: delete 99
  - could bring leaf below <u>minimum</u> number of data items
    - adopt neighboring item if neighbor not at minimum
    - otherwise, <u>combine</u> with neighbor to form a full leaf
    - process could make its way up to the root
      - if root left with 1 child, remove root and make its child the new root of the tree



135

B-Trees

- deletion: delete 99 (cont.)



136