CSCI 335
Software Design and Analysis III
Lecture 8a: Trees

Professor Anita Raja

09-22-22

1

Agenda

- Implementation of a List
  - Const_iterator and iterator
- Trees:
  - Traversal
  - Binary Search: Insert, Remove, Big 5
  - Ave Case Analysis

2

Trees

- Why?
- What: Binary Search: Insert
- This chapter
  - How trees are used to implement file systems of several popular OS?
  - How they are used to evaluate arithmetic expressions
  - How to use trees to support O(logN) search?
  - Basis for Set and map classes.

3

Trees

- Recursive definitions
  - collection of nodes
  - collection either empty or consists of a root node *r* and zero or more non-empty subtrees, the roots of which are connected by a directed edge from *r*.
- Terms:
  - Root, child, parent, leaves, siblings, path from n_1 to n_k, length of path, parent, grandparent, grandchild, ancestor, and descendents.

4

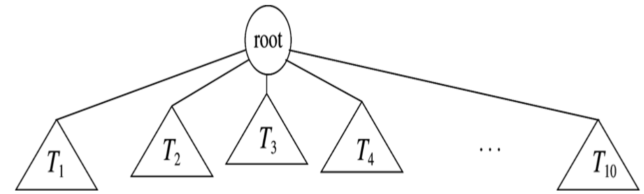
Trees

- For any node n_i
  - depth is length of unique path from root of n_i.
  - depth of root is zero.
  - height is length of longest path from n_i to leaf.
  - all leaves are at height zero.
- Height of a tree == Height of root.
- Depth of a tree == depth of deepest leaf.

5

5

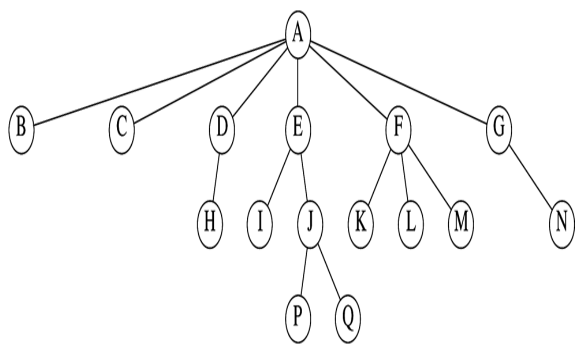
Trees (general)



6

6

Tree (example)



7

7

Terms

Term	Definition
Root	Distinguished node which is the topmost or start of tree
Child	The root of each subtree is said to be child of r
Parent	r is the parent of each subtree root.
Leaves	Nodes with no children
Siblings	Nodes with the same parents
Path from node n_1 to n_k	Sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is parent of n_{i+1} for $1 \le i < k$
Length of a path	The number of edges on the path
Grandparent	A grandparent of a node is the parent of its parent.
Grandchild	A grandchild of a node is the child of its child.
Ancestor	parent, grandparent, grand-grandparent, etc.
Descendants	child, grandchild, grand-grandchild, etc.

8

8

Trees: Depth and Height

- For any node n-:
  - depth is length of the unique path from the root to n-.
  - depth of root is zero.
  - height is length of longest path from n- to a leaf. All leaves are at height zero.
- The height of a tree is height of root.
- The depth of the tree is depth of deepest leaf.

9

9

Implementation

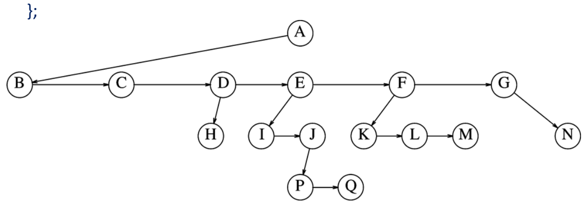
- What if the number of children per parent is not known?

10

10

Implementation

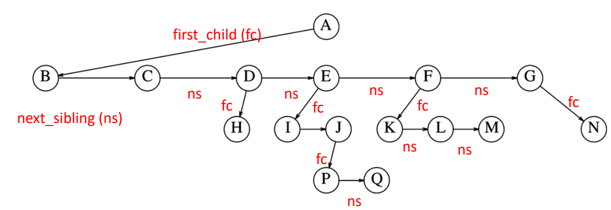
```
template <typename Object>
struct TreeNode {
  Object element;
  TreeNode *first_child;
  TreeNode *next_sibling;
};
```



11

11

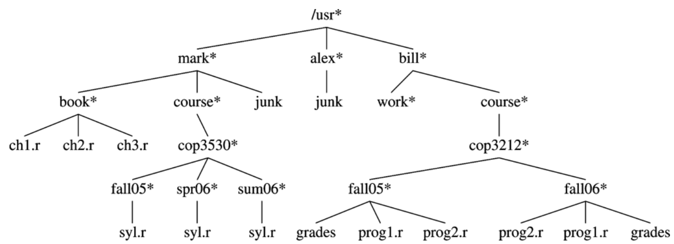
Implementation



12

12

Example (Unix file system) Traversals?



13

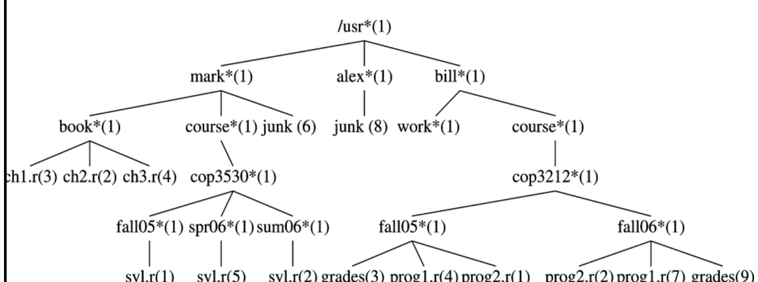
Preorder Tree Traversal Code

```
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 );
}
```

```
/usr
    mark
        book
            ch1.r
            ch2.r
            ch3.r
        course
            cop3530
                fall05
                    syl.r
                spr06
                    syl.r
                sum06
                    syl.r
        junk
    alex
        junk
    bill
        work
        course
            cop3212
                fall05
                    grades
                    prog1.r
                    prog2.r
                fall06
                    prog2.r
                    prog1.r
                    grades
```


14

Calculate Directory Size



15

Calculate size of directory

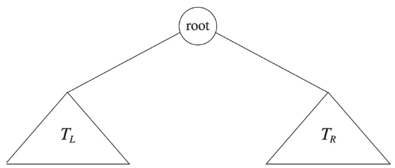
```
int FileSystem::size( ) const
{
    int totalSize = sizeOfThisFile( );

    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
                fall05      2
                    syl.r   5
                spr06       6
                    syl.r   2
                sum06       3
            cop3530        12
        course             13
        junk                6
    mark                   30
        junk                8
    alex                    9
        work                1
                    grades  3
                    prog1.r 4
                    prog2.r 1
                fall05      9
                    prog2.r 2
                    prog1.r 7
                    grades  9
                fall06     19
            cop3212        29
        course             30
    bill                   32
/usr                       72
```


16

Binary Trees

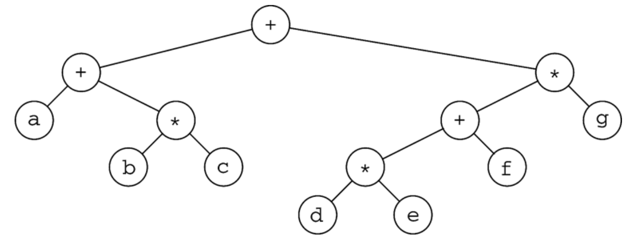
- Tree in which no node can have more than 2 children



17

Expression Trees

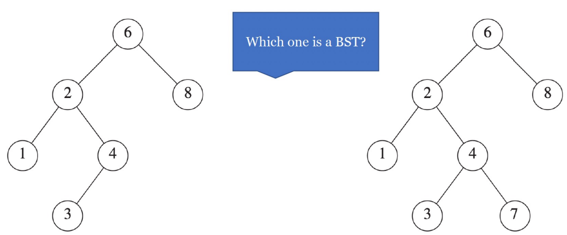
- `Leaves are operands and other nodes are operators`
- `General strategy Inorder: (a+b*c)+((d*e + f)*g)`
- Algorithm: from postfix expression to expression tree



- Postorder: Use stack to create the tree
  - Example `a b c * + d e * f + g * +`

18

The Search Tree ADT: Binary Search Trees



19

Binary search trees

- Implementation header (fig 4.16)
- Public methods for insert, remove, contains (fig 4.17)
- Contains method (fig 4.18)
- Insertion routine (fig 4.23)
- BST using a function object for less (fig 4.19)
- Deletion routine (fig 4.26)

20

Binary Node (4.16)

```
//
// Binary Search Tree implementation
// Usage: BinarySearchTree<int> a_tree;
//        a_tree.Insert(10);

template <typename Comparable>
class BinarySearchTree {
 public: // … Big five.
 private:
   struct BinaryNode {
       Comparable element_;
       BinaryNode *left_;
       BinaryNode *right_;
       BinaryNode(const Comparable &the_element, BinaryNode *lt,
BinaryNode *rt):
         element_{the_element}, left_{lt}, right_{rt} { }
       BinaryNode(Comparable &&the_element, BinaryNode *lt,
BinaryNode *rt):
         element_{std::move(the_element)}, left_{lt}, right_{rt} { }
    };
    BinaryNode *root_;
    // …
};
```

21

21

Regular Insert

```
// Internal method to insert into a subtree.
// @x is the item to insert.
// @t is the pointer to the node that roots the subtree.
// Set the new root of the subtree
// x  is inserted in the subtree, and t is updated.
// No insertion if x is already in the tree.
void Insert(const Comparable &x, BinaryNode * &t) {
     if (t == nullptr)
      t = new BinaryNode{x, nullptr, nullptr};
     else if (x < t->element_)
        Insert(x, t->left_);
     else if (t->element < x)
      Insert(x, t->right);
     //else Duplicate; do nothing
}
```

22

22

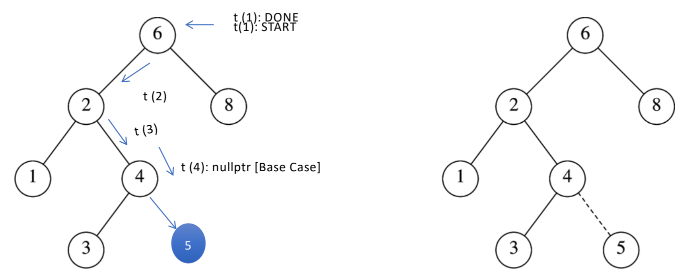
Move Insert (4.23)

```
// Internal method to insert into a subtree.
// @x is the item to insert by moving
// @t is the pointer to the node that roots the subtree.
// Set the new root of the subtree
// x  is inserted in the subtree, and t is updated.
// No insertion if x is already in the tree.
void Insert(const Comparable &&x, BinaryNode * &t) {
   if (t == nullptr)
      t = new BinaryNode{std::move(x), nullptr, nullptr};
   else if (x < t->element_)
      Insert(std::move(x), t->left_);
   else if (t->element_ < x)
      Insert(std::move(x), t->right_);
   else
             ;  // Duplicate; do nothing
}
```

23

23

Inserting 5



24

24

Remove (4.26)

```
// Internal method to remove from a subtree.
// @x is the item to remove.
// @t is the pointer to the node that roots the subtree.
void Remove(const Comparable &x, BinaryNode * &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) {  // 2 children.
         t->element = FindMin(t->right_)->element_;
         Remove(t->element, t->right);
      } else {// One or no children.
         BinaryNode *old_node = t;
         t = (t->left != nullptr) ? t->left : t->right;
         delete old_node;
      }
}  // End remove
```

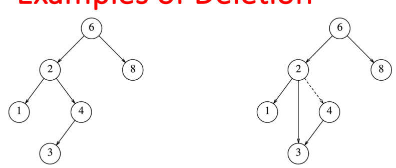

25

FindMin

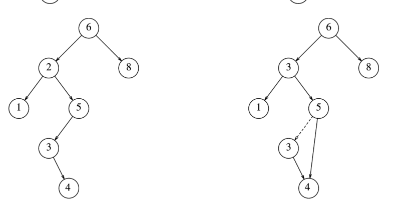
```
//
// @t: a given node of the tree.
// Find the node of minimum value under t.
// @return the node of minimum value.
//  if t is nullptr returns nullptr.
// Recursive implementation.
BinaryNode *FindMin(BinaryNode *t) const {
  return t == nullptr ? nullptr: (t->left == nullptr ? t:
FindMin(t->left));
}  // End remove
```


26

Examples of Deletion



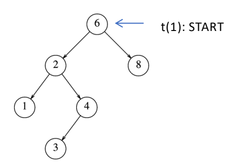
Remove 4



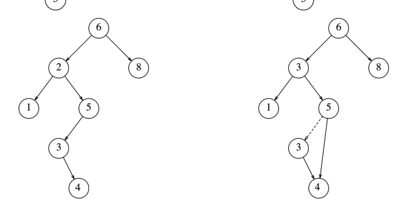
Remove 2

27

Examples of Deletion



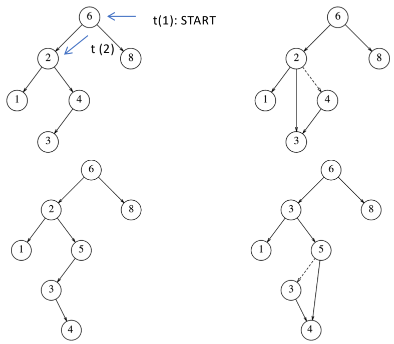
Remove 4



Remove 2

28

Examples of Deletion



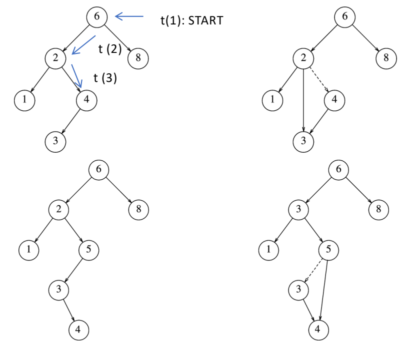
Remove 4

Remove 2

29

29

Examples of Deletion



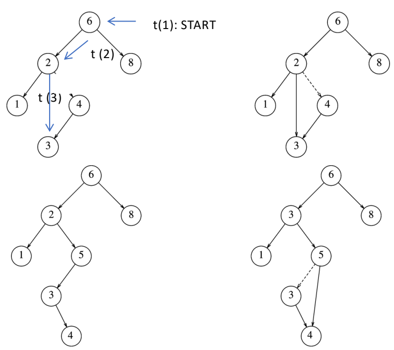
Remove 4

Remove 2

30

30

Examples of Deletion



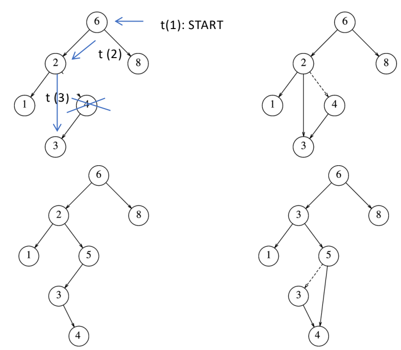
Remove 4

Remove 2

31

31

Examples of Deletion



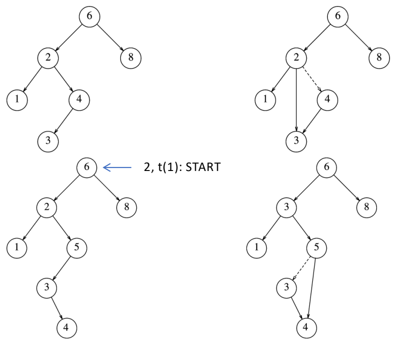
Remove 4

Remove 2

32

32

Examples of Deletion



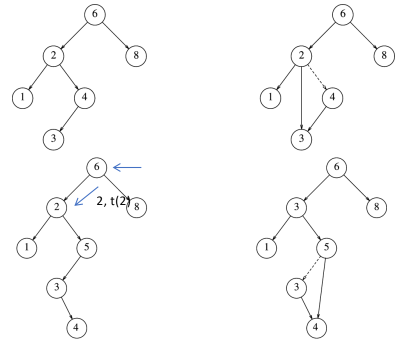
Remove 4

Remove 2

33

33

Examples of Deletion



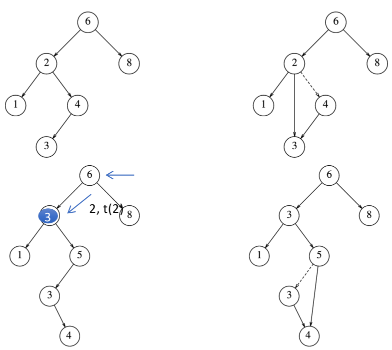
Remove 4

Remove 2

34

34

Examples of Deletion



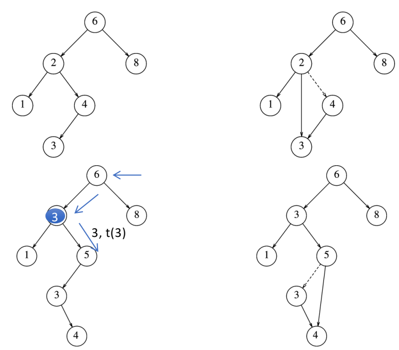
Remove 4

Remove 2

35

35

Examples of Deletion



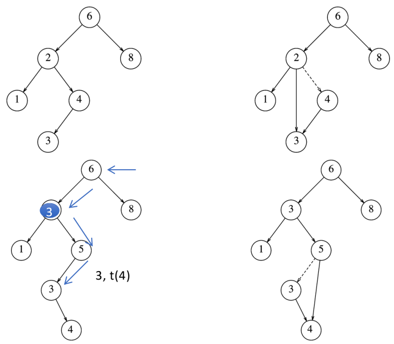
Remove 4

Remove 2

36

36

Examples of Deletion

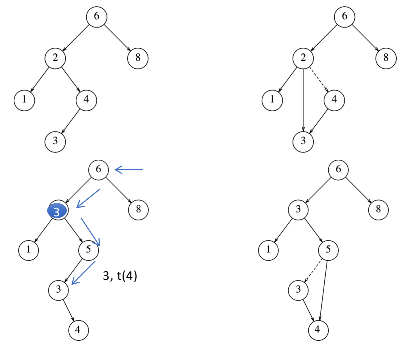


Remove 4

Remove 2

37

Examples of Deletion

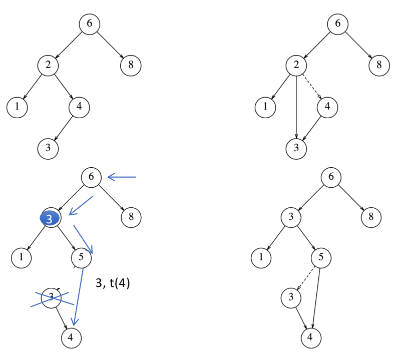


Remove 4

Remove 2

38

Examples of Deletion



Remove 4

Remove 2

39

Big Five

```
BinarySearchTree() : root{nullptr} { } // 0-parameter
constructor.


// Copy Constructor.
BinarySearchTree(const BinarySearchTree & rhs) :
root{nullptr} {
      root = clone(rhs.root);
}


// Move Constructor.
BinarySearchTree(BinarySearchTree &&rhs) : root{rhs.root_} {
        rhs.root = nullptr;
}


~BinarySearchTree() {  // Destructor.
       makeEmpty();
}
```


40

Big Five

```
// Deallocates memory of subtree with root t.
void makeEmpty(BinaryNode *&t) { // -> TRAVERSAL ??
  if (t == nullptr) return;
  MakeEmpty(t->left);
  MakeEmpty(t->right);
  delete t;
  t = nullptr;
}

// Clones the subtree with root t, and returns the root of the
// cloned tree.
BinaryNode *clone(BinaryNode *t) const {
 return t == nullptr ? nullptr:
      BinaryNode{t->element, Clone(t->left), Clone(t->right) };
}
```

41

41

Big Five (cont..)

```
// Copy Assignment.
BinarySearchTree &operator=(const BinarySearchTree & rhs) {
      BinarySearchTree copy = rhs;
      std::swap(*this, copy);
      return *this;
}

// Move Assignment.
BinarySearchTree &operator=(BinarySearchTree &&rhs) {
       std::swap(root, rhs.root);
       return *this;
}
```

42

42

Lazy Deletion

- when an element is deleted, it is left in the tree and merely marked as being deleted.

43

43

Lazy Deletion

- Small Pros
  - Easy to handle if a deleted item is reinserted, do not need to allocate a new node.
  - Do not need to handle finding a replacement node.
- Con
  - The depth of the tree will increase. However this increase is usually a small amount relative to the size of the tree.

44

44

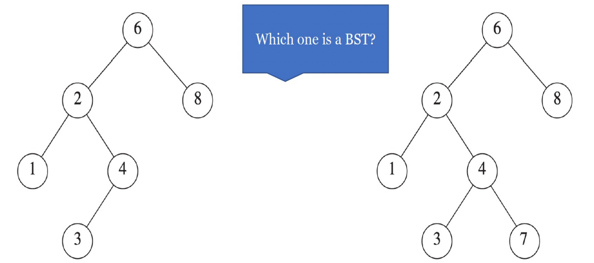
Average-Case Analysis

- All operations except makeEmpty should take $O(\log N)$
- Prove that average depth over all nodes in a tree is O(logN) on the assumption that all insertion sequences are equally likely.
- Internal path length
  - Sum of the depths of all nodes in the tree
- Calculate average internal path length of BST where average is taken over all possible insertion sequences into BSTs.

45

45

Internal path length



46

46

Average-Case Analysis

- $D(N)$: Internal Path Length of tree of N nodes
  - $D(1) = 0$
  - $D(N) = D(i) + D(N - i - 1) + (N - 1)$
  - Why ?
- In a BST, sizes of the left and right subtree depend on the first element inserted.

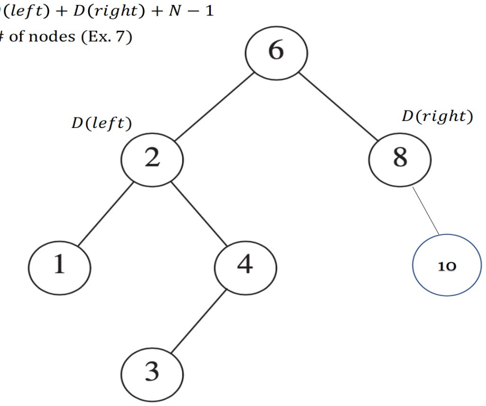
47

47

Internal path length (recursive)

$D(N) = D(left) + D(right) + N - 1$

N = # of nodes (Ex. 7)

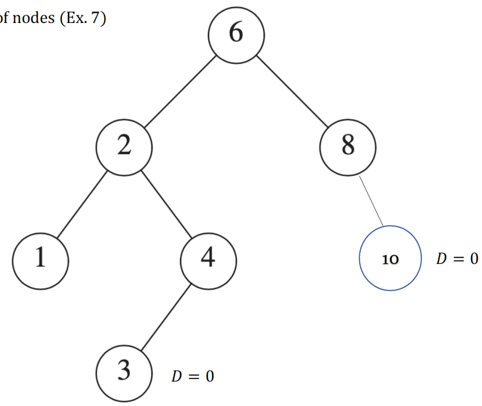


48

Internal path length (recursive)

$D(N) = D(left) + D(right) + N - 1$

N = # of nodes (Ex. 7)

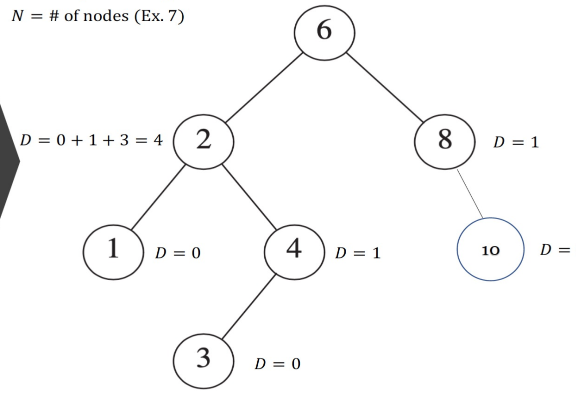


49

Internal path length (recursive)

$D(N) = D(left) + D(right) + N - 1$

N = # of nodes (Ex. 7)



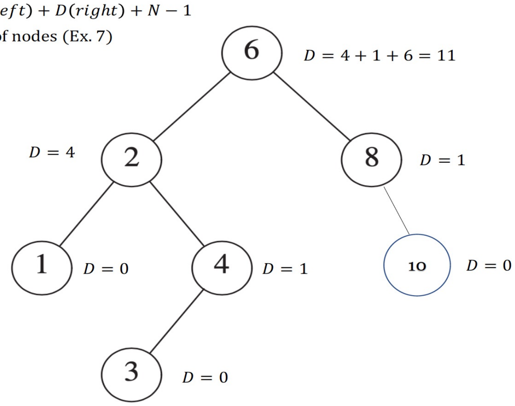
50

50

Internal path length (recursive)

$D(N) = D(left) + D(right) + N - 1$

N = # of nodes (Ex. 7)



51

51

Average-Case Analysis

- In BST all subtree sizes are equally likely so,

$$D(N) = \frac{2}{N}\left[\sum_{j=0}^{N-1} D(j)\right] + N - 1$$

- Solving that recurrence we can show that

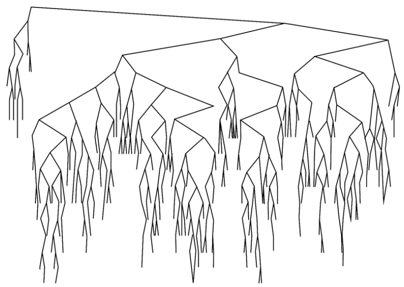
$$D(N) = O(N \log N)$$

- Thus expected depth of any node is $O(\log N)$
- Example: 500 random nodes, expected depth 9.98
- However not all inputs are equally likely!

52

52

Average case analysis

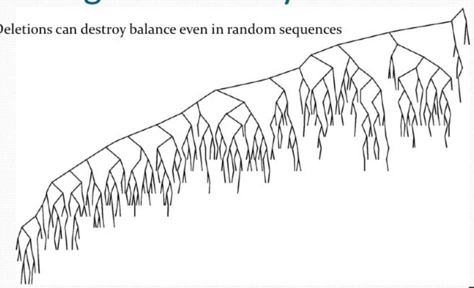


Randomly generated BST

54

Average case analysis

Deletions can destroy balance even in random sequences



After $\Theta(N^2)$ insert/remove pairs, expected depth is $\Theta(\sqrt{N})$

55

Average case analysis

- Deletion algorithm favors making left subtrees deeper than right subtrees, because we always replace a deleted node from the right subtree.
  - Can use a different replacement strategy with less bias but there is no proof this would be better.
- What is the worst-case input for a BST?

56

Balanced Trees

- Try to balance after tree operations and make operations logarithmic
  - AVL tree (balance is always preserved)
  - Splay trees (self-adjusts towards balance)

57

Announcements

- Next class: Continue with AVL Splay Trees
- No class Monday 9/26 – Hunter closed
- HW2 released by Friday night
  - go over it in entirety and understand requirements end-to-end ; take notes to plan solutions
- Gradescope will be opened by next Tuesday.

58

58