CS112: Data Structures

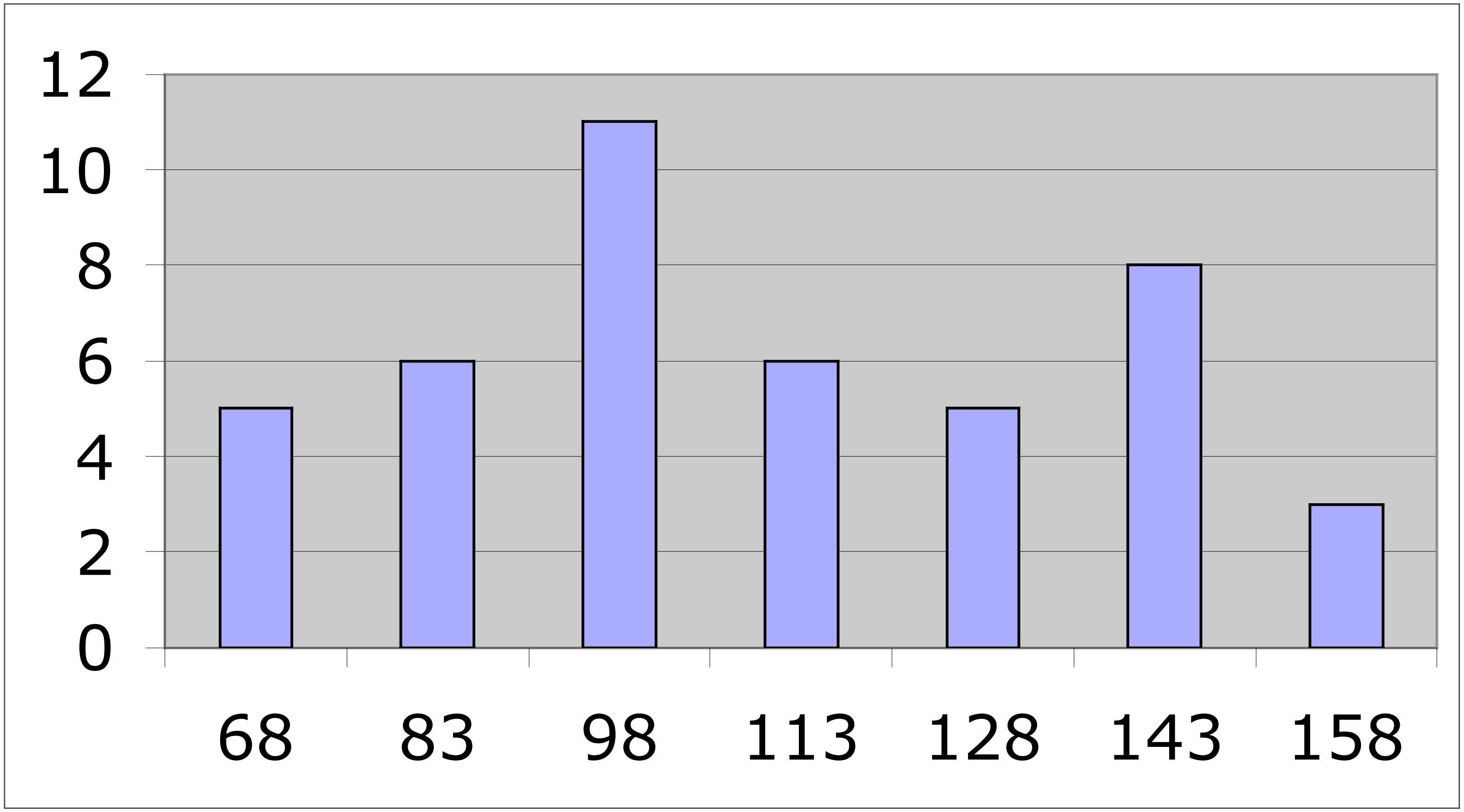
Lecture 8

More About Trees

Exam in 1 results

- :-(
- Tentatively adding 15 points (Out of 150)
- Sakai will stay the raw score

Raw Scores



Review: Hashing

- Suppose we want to store a set of numbers
  - add number to set, delete from set, test if in set should all be O(1)
- If range of numbers is small, e.g. 0 .. 9, we can use a boolean array

0	1	2	3	4	5	6	7	8	9
t	f	f	f	t	t	f	t	f	f

- What if range of numbers is large, e.g. 0…500,0000?
  - but only a small number of numbers, e.g. 10

Hashing

- If we use array of 500,000 elements, they will nearly all be false.
- Idea: divide the range into 500 blocks of 1000 numbers

Block 9:

Block 10: (numbers 9,001 to 10,000)

Is any set element in this range? True

Which one: 9,251

Hashing

- Array of 500 objects
  - Insert n: put in object at index n/1000
  - Lookup n: look in object at index n/1000
    - is any number in this object?
    - is it the right number?
  - All O(1)

Hash Function

- **What if numbers not random, eg likely to be near each other?**
  - **convert n to index in some other way, e.g. index = n mod 500**
  - **In general, function that makes each index equally likely: "makes hash out of any pattern in the numbers" -**
- **Hash function: converts data to hash code**
- **Mapping function: converts hash code to array index. (Why separate this?)**

Collisions

- Even with 500 indices for 10 numbers, it is possible that more than one number will hash to same index
- As we reduce number of indices probability of collision grows
- => must be some way to handle collisions

Linear Probing

- On insert n, if already data at hash(n), try hash(n)+1, hash(n)+2, …
- On lookup n, look at hash(n), hash(n)+1, hash(n)+2, … until
  - find n
  - find empty object
- Problem: clumping

Quadratic Probing

- **If hash(n) full try hash(n)+1, hash(n)+4, hash(n)+9, … hash(n)+j^2**
- **Does not have clumping effect**
- **Does have problem that it only tries at most half the indices**

Chaining

- Instead of moving to other indices on collision, have a linked list of items at each index

Complexity

- **Worst case: O(n)**
  - all items hash to same index
- **Average: depends on load factor α = n / size**

alpha	linear	quadratic	chaining
.1	1.06	1.06	1.05
.5	1.5	1.4	1.3
.8	3	2	1.4
.9	5.5	2.6	1.45
. 99	50.5	4.6	1.5

New: Built-in Hashing in Java

- The class java.util.HashMap<K, V>
  - Mapping from (unique) key to a value
  - Note: generic with two class parameters:
    - K: class of keys
    - V: class of values
  - E.g. Driver's license ID (String) => Driver object (name, address, etc.): java.util.HashMap<String, Driver>

Built-in Hashing in Java

- The class java.util.HashMap<K, V>
  - Mapping from (unique) key to a value
  - Note: generic with two class parameters:
    - K: class of keys
    - V: class of values
  - E.g. Driver's license ID (String)
    => Driver object (name, address, etc.):
    java.util.HashMap<String, Driver>
  - See Driver.java and UseDriverMap.java

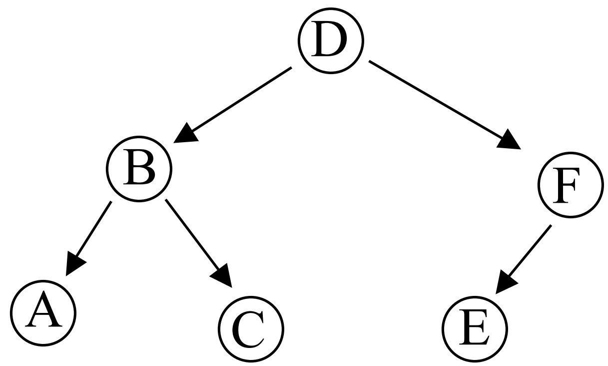
Traversals

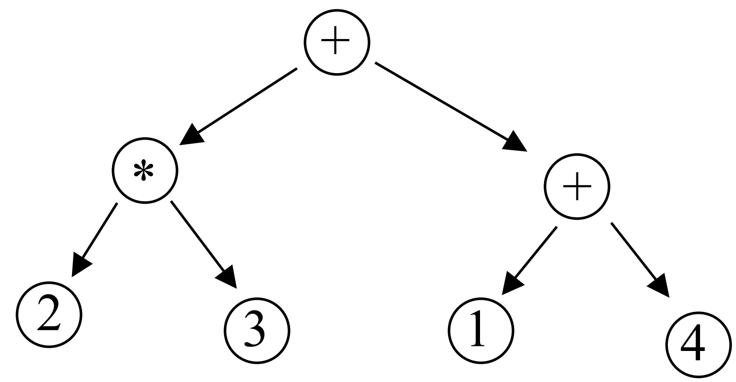
```
preOrderPrint(tree):
   if (tree.root == null)
      return
   print(tree.node)
   preOrderPrint(tree.lst)
   preOrderPrint(tree.rst)
```

```
postOrderPrint(tree):
   if (tree.root == null)
      return
   postOrderPrint(tree.lst)
   postOrderPrint(tree.rst)
   print(tree.node)
```

```
inOrderPrint(tree):
   if (tree.root == null)
      return
   inOrderPrint(tree.lst)
   print(tree.node)
   inOrderPrint(tree.rst)
```

Traversals



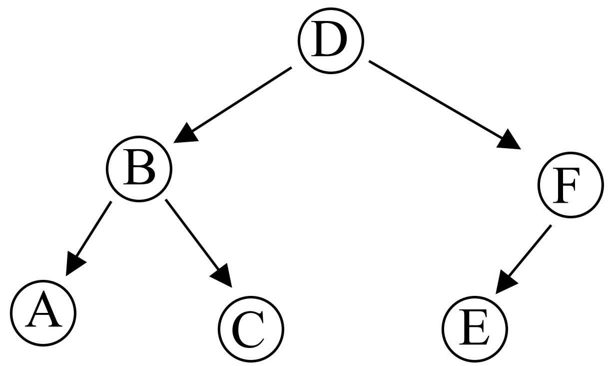


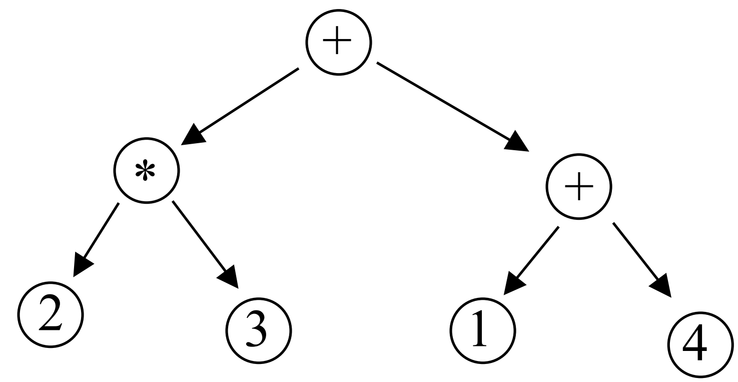
PreOrder ______________________________

InOrder ______________________________

PostOrder ______________________________

Traversals





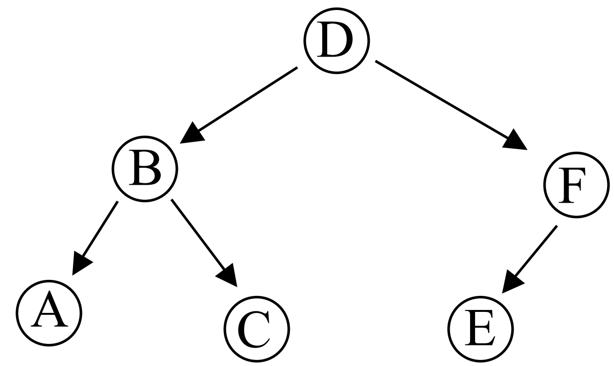
PreOrder __D B A C F E	+ * 2 3 + 1 4
InOrder __A B C D E F	2 * 3 + 1 + 4
PostOrder A C B E F D	2 3 * 1 4 + +

Non-recursive Traversals

- **Problem: If a node has more than one child**
  - **can't work on all children, grandchildren, … at once**
  - **have to store children that have been found but not processed**
- **Solution: store in stack or queue**

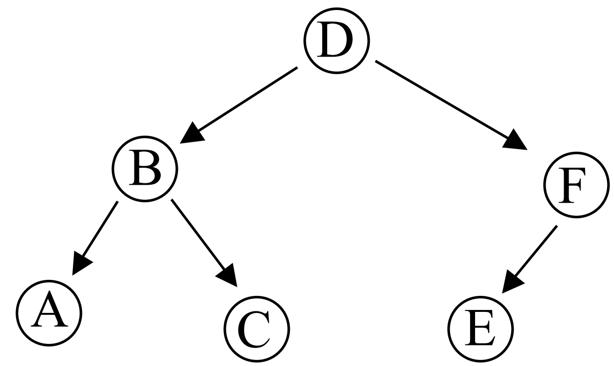
Stack-based Traversal

```
push(root);
while(! isEmpty( )){
  next = pop( );
  if (next != null){
    print next.data;
    push next.leftSubTree;
    push next.rightSubTree
}}
```



Queue-based Traversal

```
enqueue root;
while(! isEmpty( )){
  next = dequeue( );
  if (next != null){
    print next.data;
    enqueue next.leftSubTree;
    enqueue enqueue next.rightSubTree
}}
```



Breadth vs Depth first

- **Stack: depth first**
  - **do all children before anything else**
- **Queue: breadth first**
  - **do all at same level before anything else**

Size of Stack / Queue

- Stack: path from root to leaf: O(depth)
- Queue: entire level: $O(2^{depth})$
  - That's a lot!
  - Solution: Iterative Deepening

Iterative Deepening

- print all nodes at depth d:

```
idfs(tree, d)
    if d == 0
        print tree.data
    else idfs(tree.lst, limit-1)
         idfs(tree.rst, limit-1)
```

- Try all depths

```
for(j=0; j<maxDepth; j++)
    idfs(tree, j)
```

Iterative Deepening

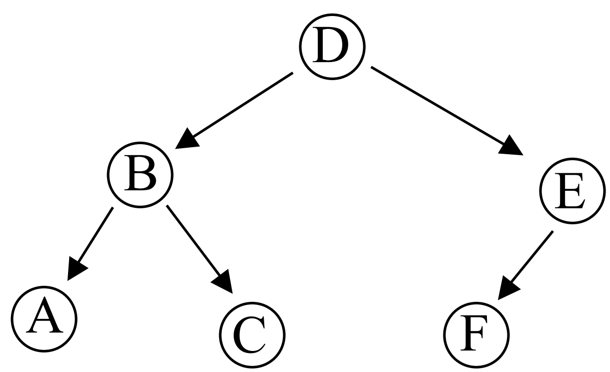
- How much extra work?
  - How many leaves in complete binary tree of depth d? 2^d
  - How many non-leaves: $2^{d-1}-1$
- Time overhead: roughly a factor of 2

Signature of a Binary Tree

- **Signature of a data structure**
  - Store off line
  - Use later to reconstruct the data structure
- **For binary tree: can we use traversals?**
  - No traversal by itself is enough to reconstruct a tree
  - But combination of preorder and inorder will do the job

Traversals -> Tree

- **Preorder: D B A C E F**
- **Inorder: A B C D F E**



Traversals -> Tree

- **Preorder: D B A C E F**
- **Inorder: A B C D F E**

- **First node in inorder is root of the tree**

Traversals -> Tree

- **Preorder: D B A C E F**
- **Inorder: A B C D F E**
- **First node in inorder is root of the tree**
- **Everything in inorder to left of root is left subtree, so recur**
- **Everything in inorder to right of root is right subtree, so recur**

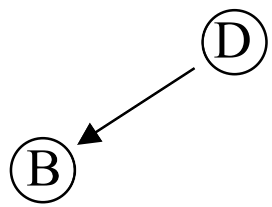
You draw the tree

Pre D B A C E F G H I

In   A B C D F E H G I

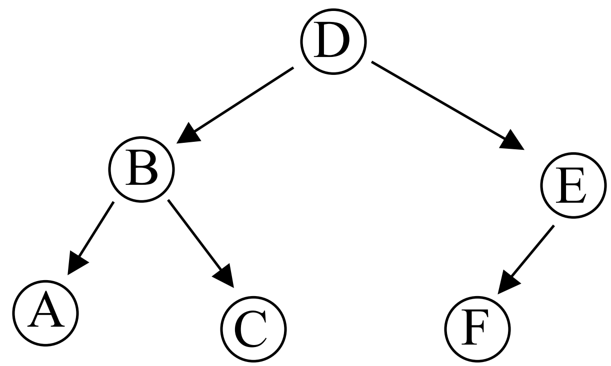
Another Signature

- **Imagine a function**
  newNode(data, leftSTree, rightSTree)



newNode(D, newNode(B, null, null), null)

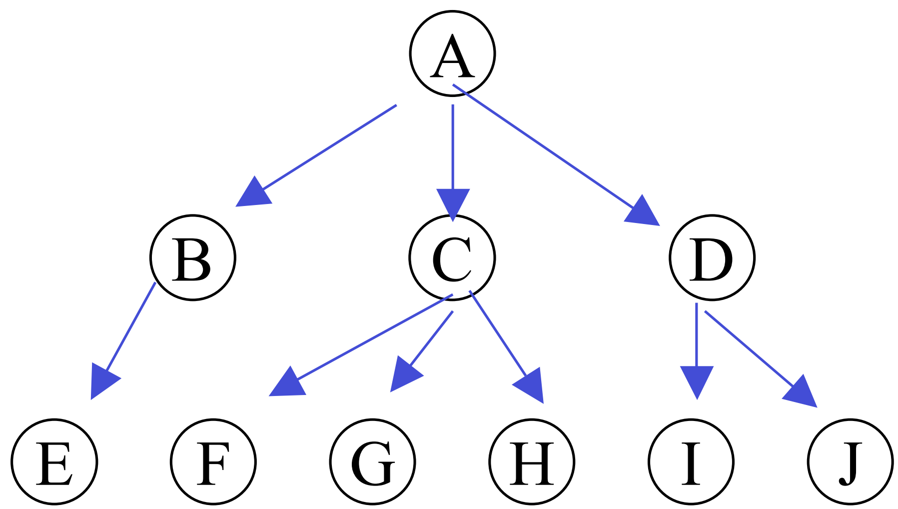
- **Write the signature:**



- **Draw the tree**
  - **newNode(B,**
    newNode(A, null, null),
    newNode(C, newNode(D, null, null),
    null))

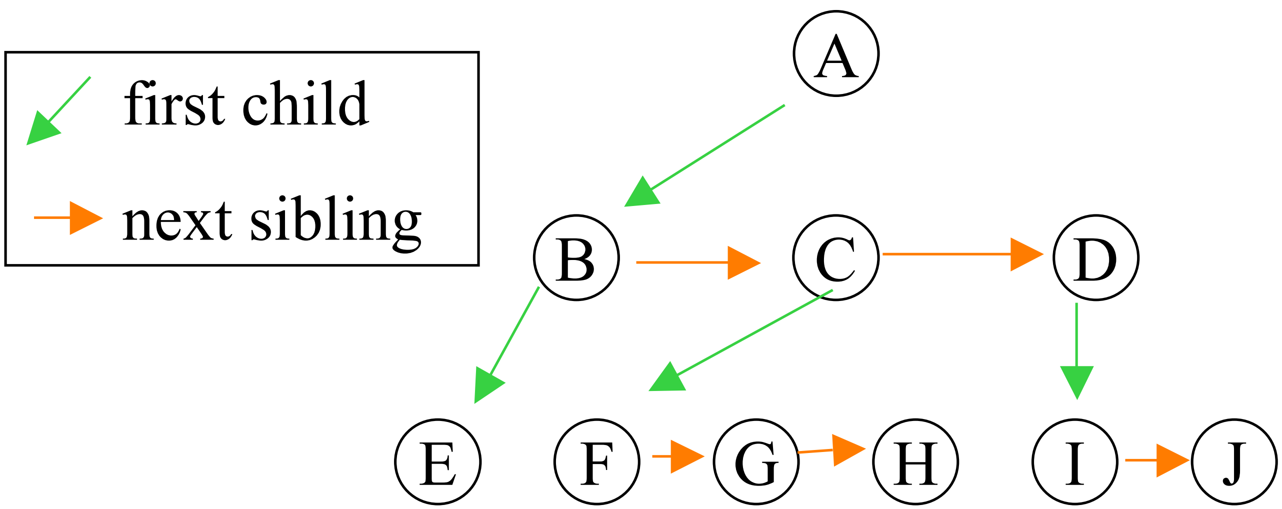
General Trees

- Each node has an arbitrary number of children
- Problem: representation of a node



General Trees

- Each node has an arbitrary number of children
- Problem: representation
- Solution: linked list of children



General Tree as Binary

- First child <=> Left child
- Next sib <=> Right child