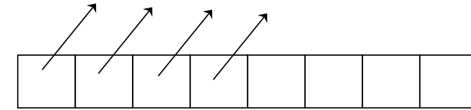
What's the Difference Here?

- How does find-a-track work? Fast forward?

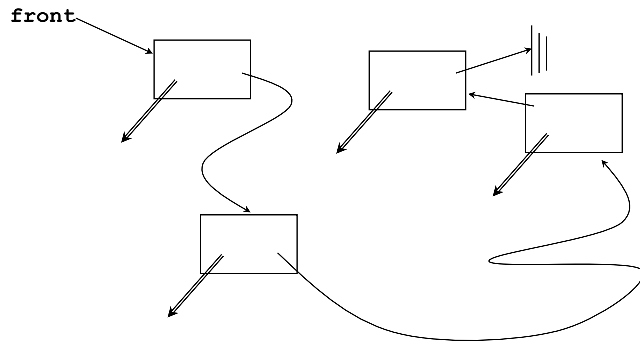
Contrast LinkedList and ArrayList

- See ISimpleList, SimpleLinkedList, SimpleArrayList
  - Meant to illustrate concepts, not industrial-strength
  - Very similar to industrial-strength, however
- ArrayList --- why is access O(1) or constant time?
  - Storage in memory is contiguous, all elements same size
  - Where is the 1st element? 40th? 360th?
  - Doesn't matter what's in the ArrayList, everything is a pointer or a reference (what about null?)



What about LinkedList?

- Why is access of Nth element linear time?
- Why is adding to front constant-time O(1)?

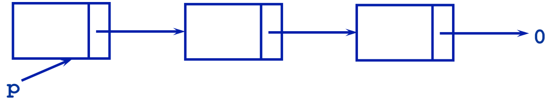


Linked list applications continued

- If programming in C, there are no "growable-arrays", so typically linked lists used when # elements in a collection varies, isn't known, can't be fixed at compile time
  - Could grow array, potentially expensive/wasteful especially if # elements is small.
  - Also need # elements in array, requires extra parameter
  - With linked list, one pointer used to access all the elements in a collection
- Simulation/modeling of DNA gene-splicing
  - Given list of millions of CGTA… for DNA strand, find locations where new DNA/gene can be spliced in
    - Remove target sequence, insert new sequence

Linked lists, CDT and ADT

- As an ADT
  - A list is empty, or contains an element and a list
  - `( )` or `(x, (y, ( ) ) )`
- As a picture



- As a CDT (concrete data type) pojo: plain old Java object

```
public class Node
{                               Node p = new Node();
     String value;                  p.value = "hello";
     Node next;                     p.next = null;
};
```


Building linked lists

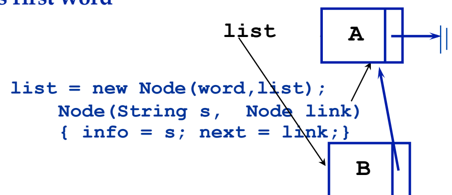
- Add words to the front of a list (draw a picture)
  - Create new node with next pointing to list, reset start of list

```
public class Node {
   String value;
   Node next;
   Node(String s, Node link){
     value = s;
     next = link;
   }
};
// … declarations here
head = list = new Node(scanner.next(), null);
while (scanner.hasNext()) {
    list = list.next = new Node(scanner.next(), null);
```

- What about adding to the end of the list?

Dissection of add-to-front

- List initially empty
- First node has first word



- Each new word causes new node to be created
  - New node added to front
- Rhs of operator = completely evaluated before assignment

Standard list processing (iterative)

- Visit all nodes once, e.g., count them or *process* them

```
public int size(Node list){
     int count = 0;
     while (list != null) {
        System.out.print(list.info);
        list.info += "s";
        list = list.next;
     }
     return count;
}
```

- What changes in code if we generalize what *process* means?
  - Print nodes?
  - Append "s" to all strings in list?

Standard list processing (recursive)

- Visit all nodes once, e.g., count them

```
public int recsize(Node  list) {
    if (list == null) return 0;
    return 1 + recsize(list.next);
}
```

- Base case is almost always empty list: null pointer
  - Must return correct value, perform correct action
  - Recursive calls use this value/state to anchor recursion
  - Sometimes one node list also used, two “base” cases
- Recursive calls make progress towards base case
  - Almost always using `list.next` as argument

Recursion with pictures

- Counting recursively

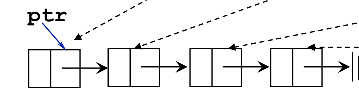
```
int recsize(Node list){
  if (list == null)
     return 0;
  return 1 +
     recsize(list.next);
}
```

```
recsize(Node list)
return 1+
recsize(list.next)
```

```
recsize(Node list)
return 1+
recsize(list.next)
```

```
recsize(Node list)
return 1+
recsize(list.next)
```

```
recsize(Node list)
return 1+
recsize(list.next)
```




```
System.out.println(recsize(ptr));
```


Recursion and linked lists

- Print nodes in reverse order
  - Print all but first node and…
    - Print first node before or after other printing?

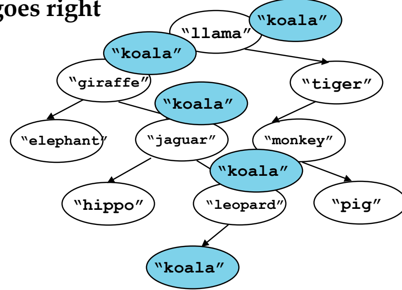
```
public void print(Node list) {
    if (list != null) {
        print(list.next);
        System.out.println(list.info);
        System.out.println(list.info);
        print(list.next);
    }
}
```


Binary Trees

- Linked lists: efficient insertion/deletion, inefficient search
  - ArrayList: search can be efficient, insertion/deletion not
- Binary trees: efficient insertion, deletion, and search
  - trees used in many contexts, not just for searching, e.g., expression trees
  - search in O(log n) like sorted array
  - insertion/deletion O(1) like list, *once location found!*
  - binary trees are inherently recursive, difficult to process trees non-recursively, but possible
    - recursion never required, often makes coding simpler

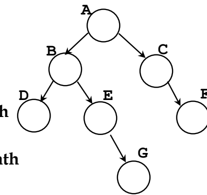
From doubly-linked lists to binary trees

- Instead of using prev and next to point to a linear arrangement, use them to divide the universe in half
  - Similar to binary search, everything less goes left, everything greater goes right
  - How do we search?
  - How do we insert?



Basic tree definitions

- Binary tree is a structure:
  - empty
  - *root* node with *left* and *right subtrees*
- terminology: parent, children, leaf node, internal node, depth, height, path
  - link from node N to M then N is *parent* of M
    - M is *child* of N
  - *leaf* node has no children
    - internal node has 1 or 2 children
  - *path* is sequence of nodes, N_1, N_2, ... N_k
    - N_i is parent of N_{i+1}
    - sometimes edge instead of node
  - *depth* (level) of node: length of root-to-node path
    - level of root is 1 (measured in nodes)
  - *height* of node: length of longest node-to-leaf path
    - height of tree is height of root
- Trees can have many shapes: short/bushy, long/stringy
  - If height is *h*, how many nodes in tree?



A TreeNode by any other name…

- What does this look like?
  - What does the picture look like?

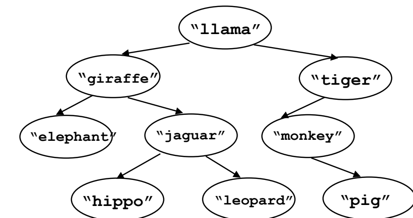
```
public class TreeNode
{
    TreeNode left;
    TreeNode right;
    String info;
    TreeNode(String s,
             TreeNode llink, TreeNode rlink){
        info = s;
        left = llink;
        right = rlink;
    }
}
```

Printing a search tree in order

- When is *root* printed?
  - After left *subtree*, before right *subtree*.

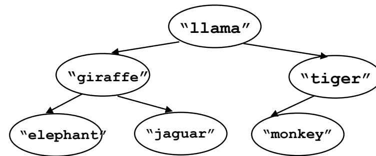
```
void visit(TreeNode t)
{
    if (t != null) {
        visit(t.left);
        System.out.println(t.info);
        visit(t.right);
    }
}
```

- *Inorder traversal*
- Big-Oh?



Tree traversals

- Different traversals useful in different contexts
  - Inorder prints search tree in order
    - Visit left-subtree, process root, visit right-subtree
  - Preorder useful for reading/writing trees
    - Process root, visit left-subtree, visit right-subtree
  - Postorder useful for destroying trees
    - Visit left-subtree, visit right-subtree, process root



Tree functions

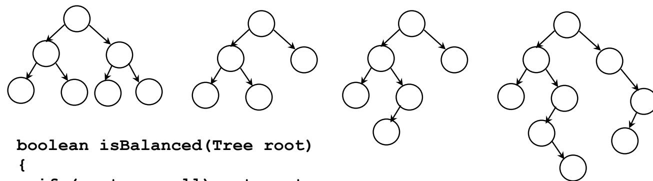
- Compute height of a tree, what is complexity?

```
int height(Tree root)
{
     if (root == null) return 0;
     else {
        return 1 + Math.max(height(root.left),
                            height(root.right) );
     }
}
```

- Modify function to compute number of nodes in a tree, does complexity change?
  - What about computing number of leaf nodes?

Balanced Trees and Complexity

- A tree is height-balanced if
  - Left and right subtrees are height-balanced
  - Left and right heights differ by at most one



```
boolean isBalanced(Tree root)
{
  if (root == null) return true;
  return
     isBalanced(root.left) && isBalanced(root.right) &&
     Math.abs(height(root.left) - height(root.right)) <= 1;
    }
}
```


What is complexity?

- Assume trees are "balanced" in analyzing complexity
  - Roughly half the nodes in each subtree
  - Leads to easier analysis
- How to develop recurrence relation?
  - What is T(n)?
  - What other work is done?
- How to solve recurrence relation
  - Plug, expand, plug, expand, find pattern
  - A real proof requires induction to verify correctness