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 $O(1)$?

ArrayLists and linked lists as ADTs

- As an ADT (abstract data type) ArrayLists support
  - Constant-time or $O(1)$ access to the k-th element
  - Amortized linear or $O(n)$ storage/time with add
    - Total storage used in n-element vector is approx. 2n, spread over all accesses/additions (why?)
    - Adding a new value in the middle of an ArrayList is expensive, linear or $O(n)$ because shifting required
- Linked lists as ADT
  - Constant-time or $O(1)$ insertion/deletion anywhere, but...
  - Linear or $O(n)$ time to find where, sequential search
- Good for sparse structures: when data are scarce, allocate exactly as many list elements as needed, no wasted space/copying (e.g., what happens when vector grows?)
Linked list applications

- Remove element from middle of a collection, maintain order, no shifting. Add an element in the middle, no shifting
  - What’s the problem with a vector (array)?
  - Naively keep characters in a linked list, but in practice too much storage, need more esoteric data structures

- What’s \((3x^3 + 2x^2 + x + 5) + (2x^4 + 5x^3 + x^2 + 4x)\)?
  - As a vector \((3, 0, 2, 0, 1, 5)\) and \((0, 2, 5, 1, 4, 0)\)
  - As a list \(((3,5), (2,3), (1,1), (5,0))\) and ________?
  - Most polynomial operations sequentially visit terms, don’t need random access, do need “splicing”

- What about \((3x^{100} + 5)\)?

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

```java
public class Node {
    String value;
    Node next;
    Node(String s, Node link){
        value = s;
        next = link;
    }
}
```

// declarations here
Node list = null;
while (scanner.hasNext()) {
    list = new Node(scanner.next(), list);
}

- What about adding to the end of the list?

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

```java
public class Node {
    String value;
    Node next;
    Node(String s, Node link){
        value = s;
        next = link;
    }
}
```

// declarations here
Node list = null;
while (scanner.hasNext()) {
    list = new Node(scanner.next(), list);
}

- 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

```
list = new Node(word, list);
```

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) {
        count++;
        list = list.next;
    }
    return count;
}
```

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

Building linked lists continued

- What about adding a node to the end of the list?
  - Can we search and find the end?
  - If we do this every time, what’s complexity of building an N-node list? Why?

- Alternatively, keep pointers to first and last nodes of list
  - If we add node to end, which pointer changes?
  - What about initially empty list: values of pointers?
    - Will lead to consideration of header node to avoid special cases in writing code

- What about keeping list in order, adding nodes by splicing into list? Issues in writing code? When do we stop searching?

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

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

Recursion and linked lists

```
public void print(Node list) {
    if (list != null) {
        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?

```java
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.

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

Insertion and Find? Complexity?

- How do we search for a value in a tree, starting at root?
  - Can do this both iteratively and recursively, contrast to printing which is very difficult to do iteratively
  - How is insertion similar to search?
- What is complexity of print? Of insertion?
  - Is there a worst case for trees?
  - Do we use best case? Worst case? Average case?
- How do we define worst and average cases
  - For trees? For vectors? For linked lists? For arrays of linked-lists?
Tree functions

- Compute height of a tree, what is complexity?
  ```java
  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?

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

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

```java
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;
}
```