Why Linked Lists are Important

Linked list problems are often used as interview questions. The structure itself is simple, operations such as "reverse a list" or "delete a list" are easy to describe and understand. They are short to state, but may have complex, pointer intensive solutions.

Even though linked lists are simple, the algorithms that operate on them can be as complex and beautiful as you want. It's easy to find linked list algorithms that are complex, and pointer intensive. No one really cares if you can build linked lists, but they do want to see if you have programming agility for complex algorithms and pointer manipulation.

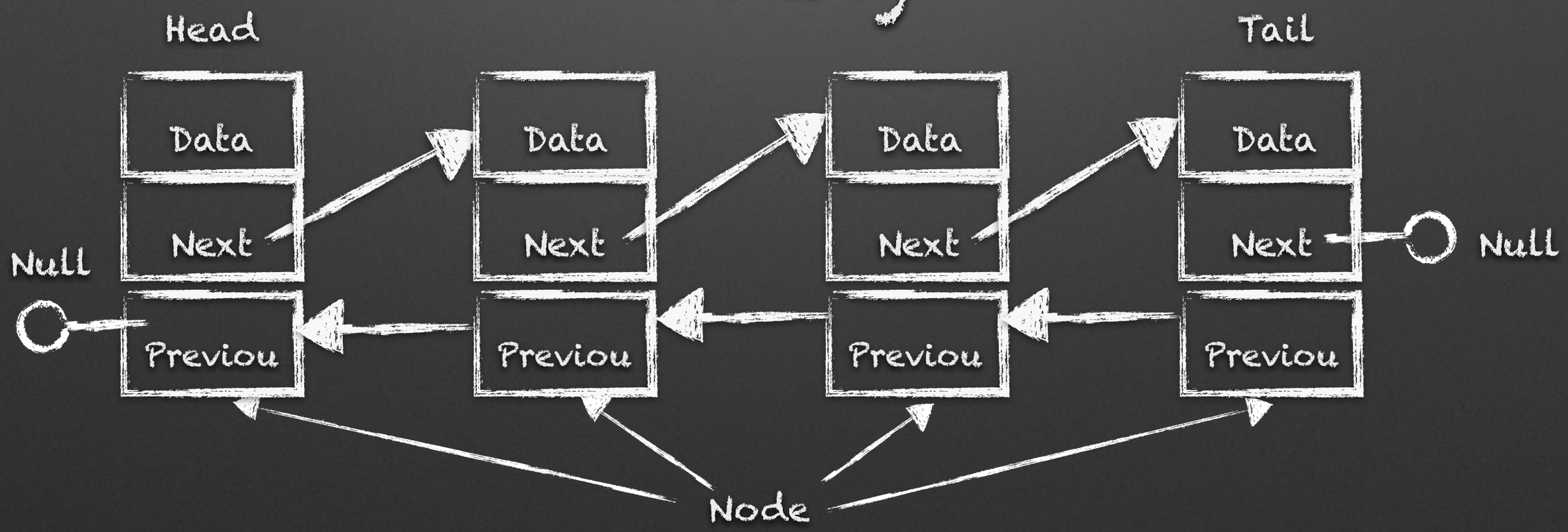
Linked Lists

- Common Variations
  - Singly Linked - each node has a link to the next
  - Doubly Linked - each node includes links to next and previous
  - circular linked list - last node points to the first node
- Review will focus primarily on singly linked lists

Linked Lists

Singly

Doubly



Advantages

- In languages like C where there are no "growable-arrays", linked lists are typically used when # elements varies, isn't known, and can't be fixed at compile time

- Insertions and deletions are simpler than with an array

- When node points to a large data item, adding or moving items in the list is faster than moving the items themselves.

- 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?)

Disadvantages

- Difficult to reverse traverse (single link)
- Require extra space for pointers
- Random access to items is inefficient
- Arrays allow better memory cache performance than random pointer jumping

Real World Uses

- Hashtables that use chaining to resolve hash collisions typically have one linked list per bucket for the elements in that bucket.
- adjacency list representation of a graph
- Simple memory allocators track list of unused memory regions
- In a FAT filesystem, the metadata of a large file is organized as a linked list of FAT entries.
- Symbol table in compiler design (one option)

Real World Uses (cont.)

- stacks and queues, including the C-language "call stack" in the x86 (and most other) binary APIs
- Representing a polynomial efficiently - list is sparse vs array
- M(ost)R(recently)U(used) list, ie. of file names
- Browser BACK button (a linked list of URLs)
- Undo in applications (a linked list of state)

Linked lists vs. dynamic arrays

Comparison of list data structures

	Linked list	Array	Dynamic array	Balanced tree	Random access list
Indexing	O(n)	O(1)	O(1)	O(log n)	O(log n)
Insert/delete at beginning	O(1)	N/A	O(n)	O(log n)	O(1)
Insert/delete at end	O(n) when last element is unknown;	N/A	O(1) amortized	O(log n)	O(log n) updating
Insert/delete in middle	search time + O(1)	N/A	O(n)	O(log n)	O(log n) updating
Wasted space (average)	O(n)	0	O(n)	O(n)	O(n)

A dynamic array is a data structure that allocates all elements contiguously in memory. If the space reserved for the dynamic array is exceeded, it is reallocated and (possibly) copied.

A random access list is a list with support for fast random access. One possible implementation is a skew binary random access list

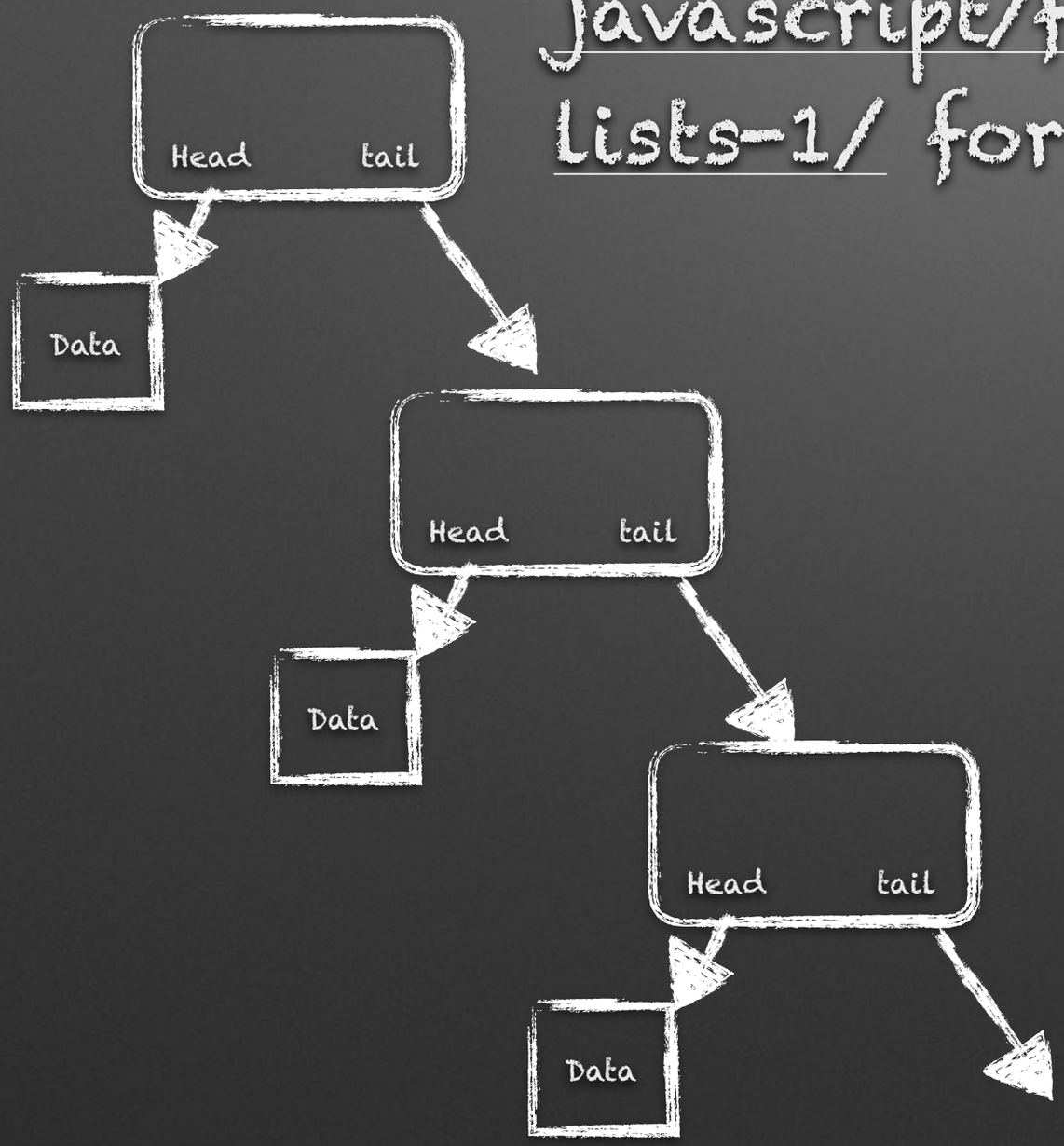
Key Considerations

- Minimal Implementation
  - Must track head
  - Pass head by reference (insert, remove)
  - Head = null; // empty list
  - Methods to implement depend on anticipated use:
    - Stack: push (insert before head), pop (remove head) and peek (return value of head)
    - Queue: Enqueue (same as push), Dequeue (remove tail)
    - Dictionary: Add, Remove, Find, Next, Previous, Minimum, Maximum

- Lists are recursive objects - Chopping the first element off a linked list leaves a smaller linked list - allows functional approach

Functional Linked List

http://blog.jeremyfairbank.com/javascript/functional-javascript-lists-1/ for additional details



Push(data) - Pass head by reference

```
/* C */
void Push(struct node** headRef, int data)
```

```
// Javascript
function List() {this.head = undefined;}
var list = new List();
List.prototype.push = function(data);
```

Push(data) - create new node

```
/* C */
struct node* newNode;
newNode = malloc(sizeof(struct node));
newNode->data = data;

// Javascript
var newNode = new Node();
newNode.data = data;
```

Push(data) - Link next

```
/* C */
newNode->next = *headRef;


// Javascript
newNode.next = list.head;
```

Push(data) - set head

```c
/* C */
*headRef = newNode;


// Javascript
list.head = newNode;
```

Pop() - pass head by reference

```
/* C */
  Node* pop (Node** head)
```

```
// Javascript
  function List() {this.head = undefined;}
  var list = new List();
  List.prototype.pop = function();
```

Pop() - set temp

```c
/* C */
  Node* temp = *head;
```

```javascript
// Javascript

  var temp = this.head;
```

Pop() - update head

```c
/* C */
  *head = temp->next;
```

```javascript
// Javascript

  this.head = temp.next;
```

Pop() - clear temp next and return

```c
/* C */
  temp->next = NULL;
  return temp;
```

```javascript
// Javascript
  temp.next = null;
  return temp;
```

Resources

- Linked List Basics, (in C) (http://cslibrary.stanford.edu/103/LinkedListBasics.pdf)
- Linked List Problems, http://cslibrary.stanford.edu/105/LinkedListProblems.pdf problems are, in rough order of difficulty Count, GetNth, DeleteList, Pop, InsertNth, SortedInsert, InsertSort, Append,FrontBackSplit, RemoveDuplicates, MoveNode, AlternatingSplit, ShuffleMerge, SortedMerge, SortedIntersect, Reverse, and RecursiveReverse
- Cracking the Coding Interview - has pointers on how to approach solving list based questions
- Algorithm Design Manual, Skiena - in depth review of arrays vs lists; excellent discussion of sorted & unsorted, singly or doubly linked list based Dictionary implementation