Lectures: Data Structures

Programming Systems and Tools

State University of New York at Binghamton

April 11, 2024

Lists

Motivation

Vectors have problems

C dynamic vectors are:

- **Inflexible**: Vectors store in statically sized arrays, meaning once the size is allocated the original array cannot change.
- **Wasteful**: Over-allocation happens “in case you need it”
- **Cumbersome**: Insertion requires resizing, resizing requires reallocation.

Vectors Hide Underlying Complexity

- Suppose we have a dynamic vector v with `.array={1,4,10,19,6}, .capacity=5`.
- Then suppose we want to call `insert(v,7,2)`. What vector operations have to happen?

Vectors Hide Underlying Complexity

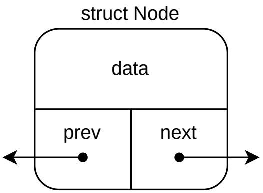
- Suppose we have a dynamic vector v with `.array={1,4,10,19,6}`, `.capacity=5`.
- Then suppose we want to call `insert(v,7,2)`. What vector operations have to happen?
  1. `resize(v,8)`:
     1. Traverse heap free list to find a new array storage: $O(n)$
     2. Acquire new heap storage: $O(1)$
     3. Copy array contents from old heap storage to new: $O(n)$
     4. Deallocate old heap storage: $O(1)$
     5. Update v.capacity $= 8$: $O(1)$
  2. Copy 3 array items to next index: $O(n)$
  3. Insert 7 at index 2: $O(1)$
  4. Update v.size $= 6$: $O(1)$

Linked Lists Remove Underlying Complexity

- The problem with arrays is that they must be contiguous in memory and statically sized.
  - Resizing is entirely dependent on the current state of memory.
- What if we could link non-contiguous segments of memory?
  - Insertion and deletion are less cumbersome – we just link to the subsequent or previous segment.
  - Resizing, inserting or deleting are a matter of changing links between segments.

Node Objects

- We will create our list using objects called **nodes**.
- Each node represents one piece of linked data and is comprised of two or three members:
  - Its data payload, and either:
  - A pointer to the next node only (*singly linked*)
  - A pointer to both the previous and next nodes (*doubly linked*)
- Every node contains the address of the next node and previous node with two exceptions:
  - The **head** contains `null` in its previous pointer.
  - The **tail** contains `null` in its next pointer.



```
typedef struct Node{
  Data data;
  struct Node *next;
  struct Node *prev;
} Node;
```

What Goes in the Node?

Allowing mixed types in a List

- No matter what kind of data is stored in the Data member, the algorithms for our `List`'s "CRUD" (Create, Read, Update, Delete) operations will be the same.
- In other languages, we can use *templates* to hold the data, making the data structure type agnostic.
- In C, we *could* make our `Node`s able to hold multiple types of data by typedefs or payload structs

What Goes in the Node?

Allowing mixed types in a List

```c
#include <stdint.h>

/* typedef option */
typedef uint64_t Data;

/* payload struct option */
typedef struct Data {
  union {
  uint64_t uint64;
  int64_t int64;
  } data;
  enum type { UINT64, INT64 };
} Data;
```

What Goes in the Node?

Pros of typedefs

- Easy to implement
- Single point to change

Cons of typedefs

- Every Data member has to be same type throughout the program
- Must keep uses of Data synchronized to type

Pros of struct Data

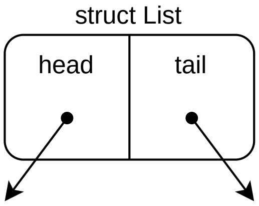
- We can have multiple payload types throughout the program
- We can have mixed payload types, even within the same data structure!

Cons of struct Data

- Harder to implement
- Uses more storage space
- Must access the payload through the union
- Must keep Data's enumeration in sync

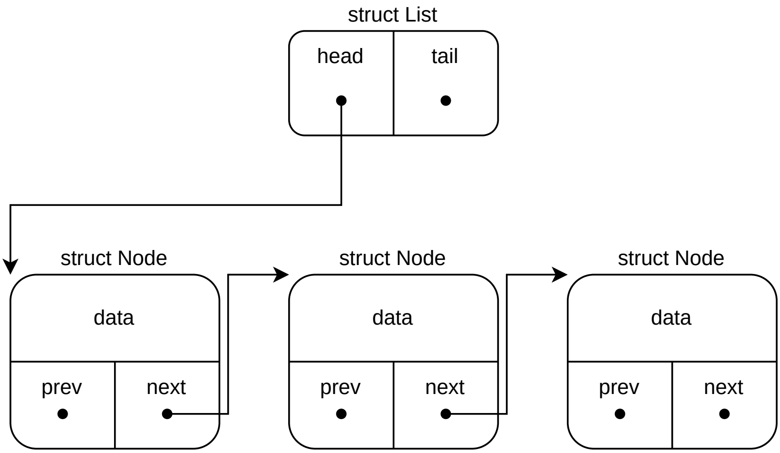
List Objects

- A linked list object's only *required* member is a pointer to the head of the list
- For a doubly-linked list, we can add a pointer to the tail for convenience, and a count of the number of linked `Node`s in the `List`
- For an empty list, we set the head (and tail) pointers to NULL, and the size to 0
- Do not use the head or tail pointers to traverse the list. Copy them into another pointer first. . . . **Why?**
- We don't include any data payload in the list object itself . . . **Why?**

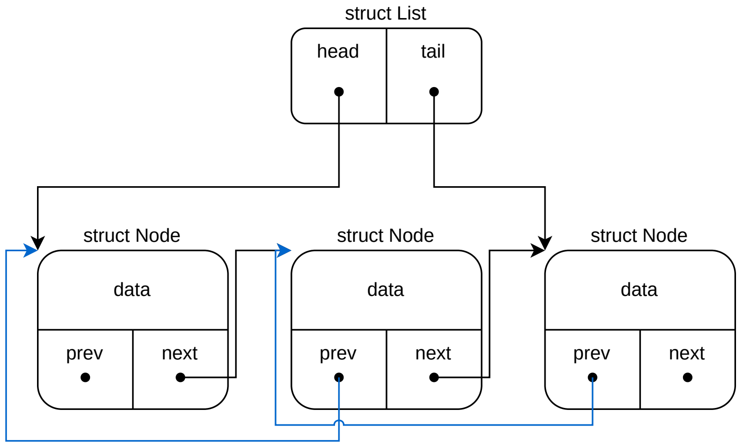


```
typedef struct List {
  Node *head,
  Node *tail;
  size_t size;
} List;
```

Singly Linked List



Doubly Linked List



List Object Definition

We will implement the List with the following definition:

```c
#ifndef DATA
#define DATA
typedef uint64_t Data;
#endif // DATA

#ifndef LIST_H
#define LIST_H

typedef struct Node {
  Data data;
  struct Node *next, *prev;
} Node;

typedef struct List {
  Node *head, *tail;
  size_t size;
} List;

```

List Object Definition

```c
Node *newNode(Data d);
void deleteNode(Node *n);
int printNode(const void *n);

List *newList();
void deleteList(List *l);
int printList(const void *l);

List *append(List *l, Node *n);
void clear(List *l);
Node *find(List *l, Data d);
List *insert(List *l, Node *n, Node *pos);
size_t length(List *l);
List *remove(List *l, Node *pos);
void traverse(List *l, int (*func)(const void *));
```

All of the object management functions work as with Vectors:

- The `new` function returns a pointer to the newly created object
- The `delete` function frees the memory of the object pointed to by the argument. In the case of the `deleteList()` function, it must also delete all the Nodes in the List first.

Copying a Linked List

What happens when we use the assignment operator with two pointers to List objects? `struct List new_list = old_list;`

Shallow Copy	Only copy the pointer address. Both list pointers point to the same list
Deep Copy	Copy the sub-objects from the old object into the new object, including their values. Each list pointer points to one of two identical but separate lists

Copying a Linked List

What happens when we use the assignment operator with two pointers to List objects? `struct List new_list = old_list;`

Shallow Copy	Only copy the pointer address. Both list pointers point to the same list
Deep Copy	Copy the sub-objects from the old object into the new object, including their values. Each list pointer points to one of two identical but separate lists

C will do a shallow copy with the assignment operator. We must implement a `List *copyList(List *l)` function to get deep copy through list traversal . . . This is an exercise to the reader.

Getting the length of a List

```
size_t length(List *l);
```

Two options:

- Traverse the `List`, counting the number of nodes we encounter until we reach either a NULL pointer or the tail node: $O(n)$
- Update the `List.size` field after each `append()`, `clear()`, `insert()`, `remove()` operation: $O(1)$ amortized

Getting a Node from the List

```
Node *find(List *l, Data d);
```

1. Create an iterator pointer and initialize it to `l->head`
2. Check the contents of the node for a match,
   - if matched, return
   - If no match, advance the iterator to `iter->next`
3. If `iter == NULL`, break and return

Return values:

- We found the first node with the desired contents: return a pointer to that node
- We go past the tail of the list to a `nullptr`: we return a NULL pointer to indicate we couldn't find the right node

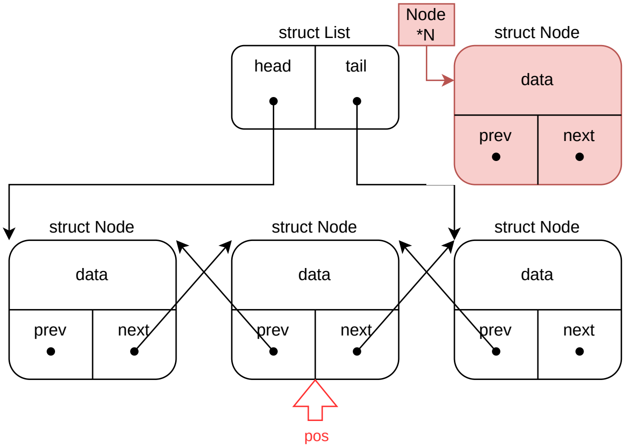
→ Either way, `iter` points to the value we want to return!

Adding Nodes to the List

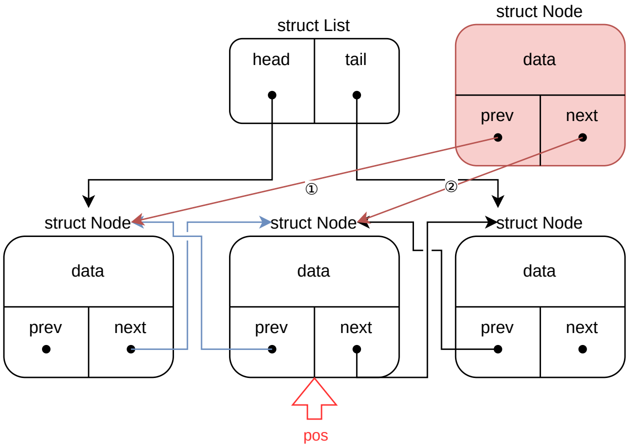
```
List *insert(List *l, Node *n, Node *pos);
List *append(List *l, Node *n);
```

- The difference between the two functions is that `*insert_node(List *l, Node *n, Node *pos)` inserts `n` before `pos`. `List *append_Node(List *l, Node *n)` inserts the node after the last node in the List (i.e. at a new `l->tail`).
- When inserting, connect the new node to the list before updating pointers already in the lists' nodes.
- Once the new node is properly connected, it can be used to update the existing `List` node pointers.
- Failure to properly connect the new node will cause a memory leak due to lost nodes, and/or segmentation faults when traversing the list.

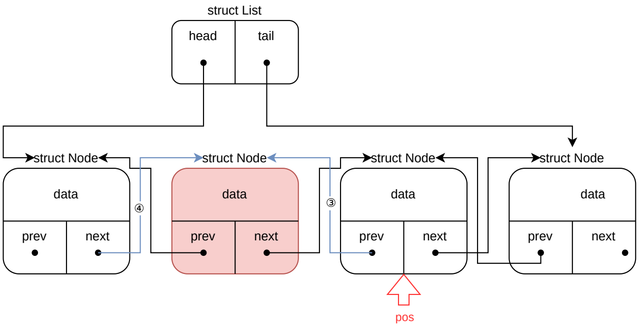
Adding Nodes to the List



Adding Nodes to the List



Adding Nodes to the List

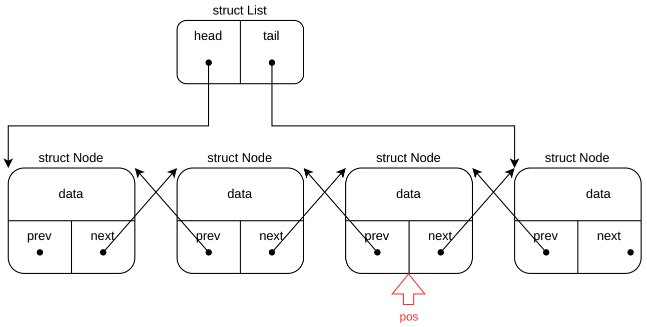


Removing Nodes from the List

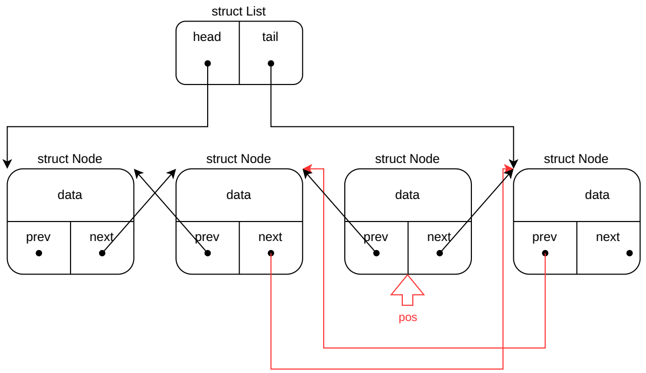
```
List *remove(List *l, Node *pos);
```

1. Find the node before `pos` if singly-linked (the predecessor, `pred`). The node after (the successor, `succ`) can be found using `pos` itself.
2. Update `pred->next` to point to successor, `succ->prev` to point to predecessor.
3. Delete the node pointed to by `pos` to prevent a memory leak and return a pointer to the newly updated list.

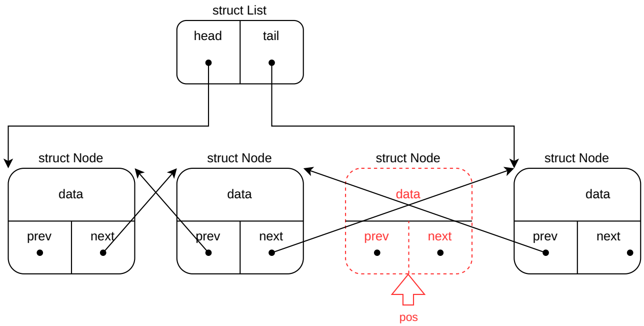
Removing Nodes from the List



Removing Nodes from the List



Removing Nodes from the List


struct List
head
tail
struct Node
data
prev
next
struct Node
data
prev
next
struct Node
data
prev
next
struct Node
data
prev
next
pos