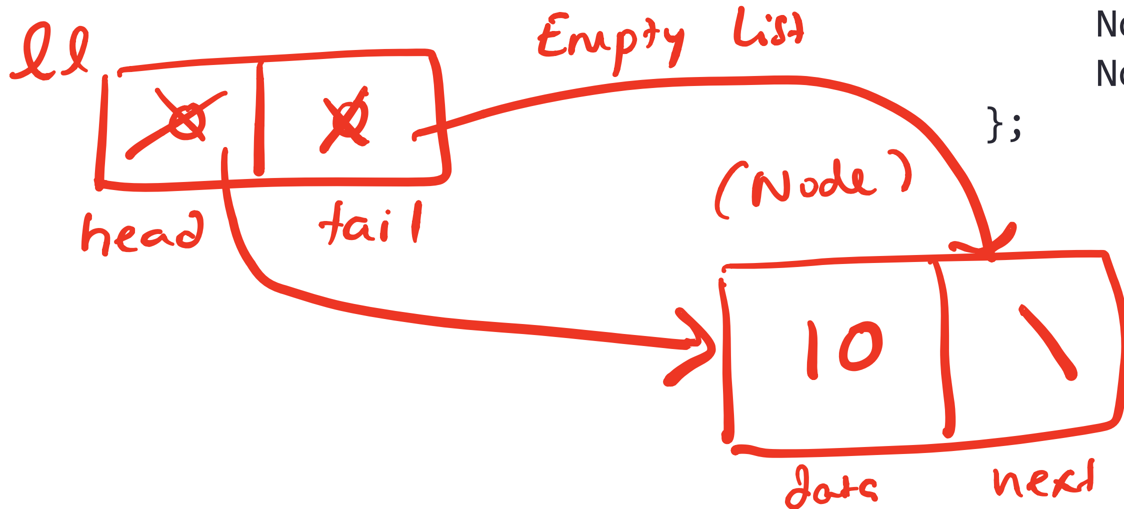
LINKED LISTS & DYNAMIC MEMORY ERRORS

Problem Solving with Computers-I

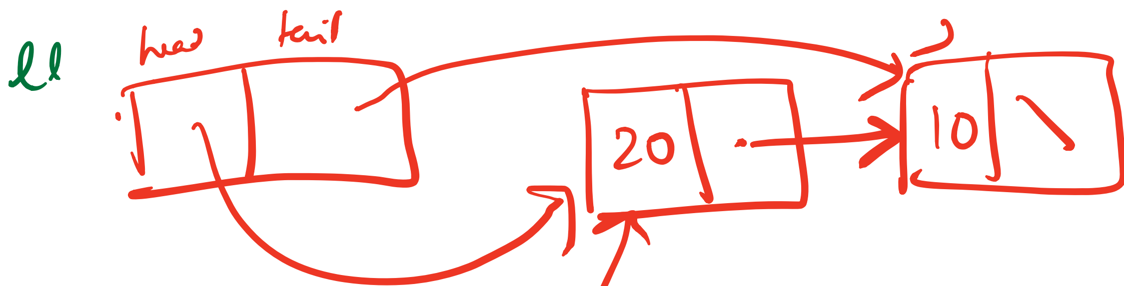
Creating a small list

- Define an empty list
- Add a node to the list with data = 10

```
struct Node {
        int data;
        Node* next;
};

struct LinkedList {
        Node* head;
        Node* tail;
};
```



Insert into a non-empty list



Iterating through a list (ll)

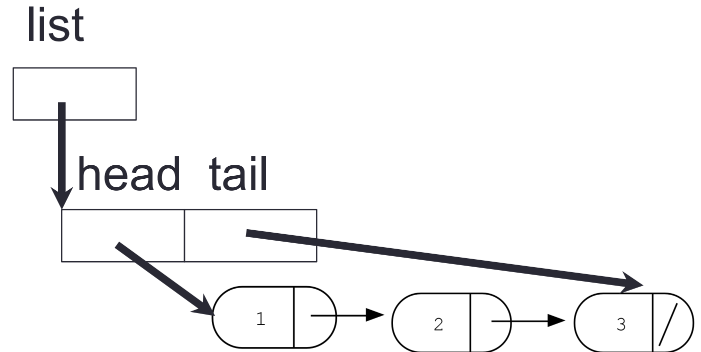
```
Node * p = ll.head;  // p is a traversal pointer

while (p) {
    // Do something with the current node that
    // p is pointing to
    p = p + next;  // Move p to the next node
}
```

Inserting a node at the head of a linked list

```
void insert(LinkedList& h, int value) ;
```

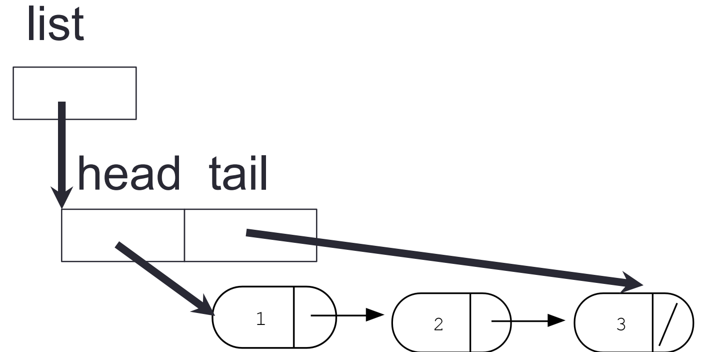
Iterating through the list

```
/* Find the number of elements in the list */

int count(LinkedList& list);
```

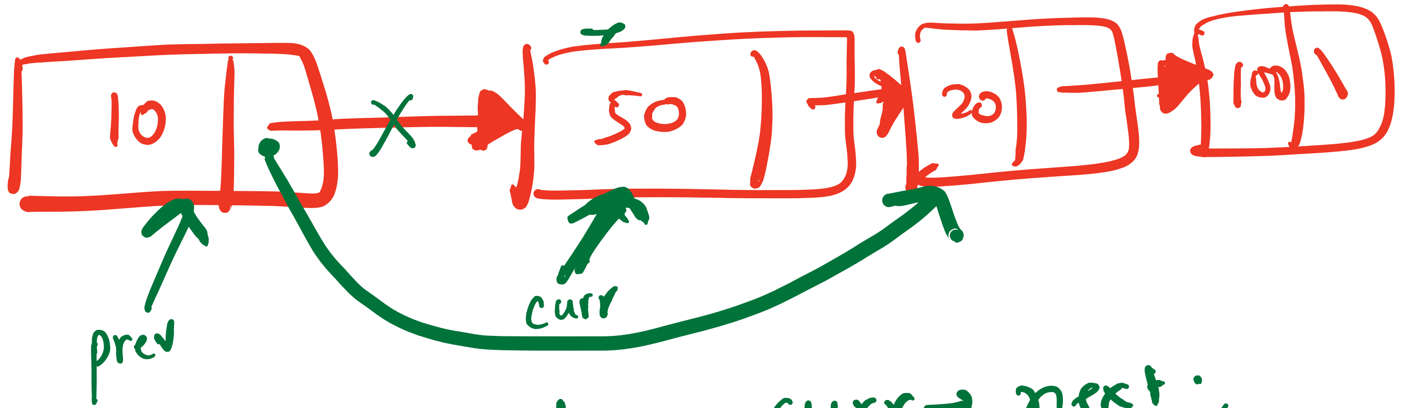


Deleting the list

```
/* Free all the memory that was created on the heap*/

void freeList(LinkedList& list);
```

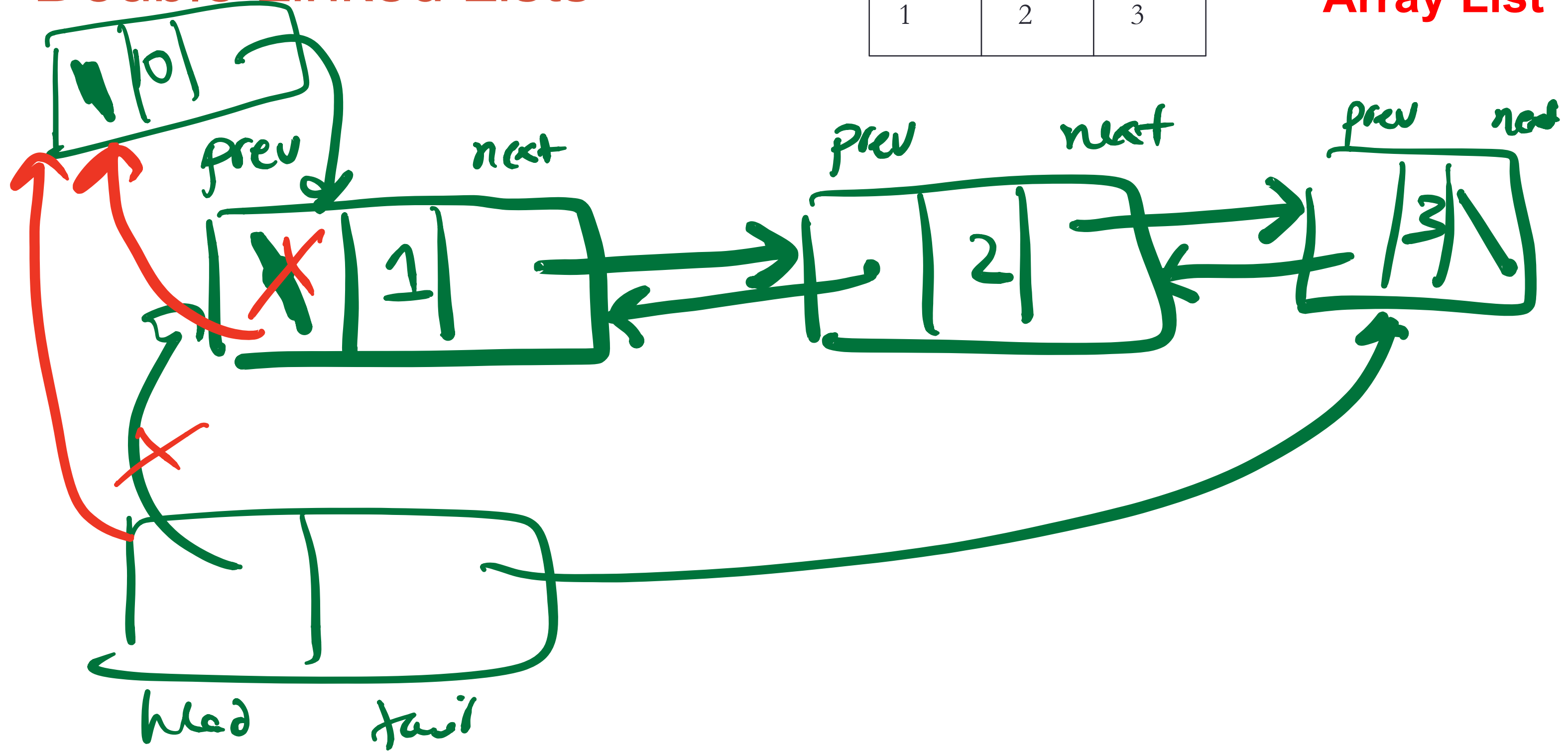


Deleting a node from the linked list




```
prev->next = curr->next;
delete curr;
// Need a special case to delete the first node
```

Double Linked Lists



Implementing a double-linked list

- Define a node in a double linked list
- Write functions to
  - insert a node to the head/tail of the linked list
  - Print all the elements of the list
  - Delete a node with a given value
  - Free the list

Dangling pointers and memory leaks

–Dangling pointer: Pointer points to a memory location that no longer exists

–Memory leaks (tardy free):

- Heap memory not deallocated before the end of program
- Heap memory that can no longer be accessed

```
int * p;      // p is a dangling pointer
int * q = 0;  // q is not dangling.
// Problem with dangling pointers is that dereferencing them can
// cause a segfault
cout << *p ;  // Possible segfault
```

If instead the pointer was initialized to 0, we could do a null check before dereferencing.

Dynamic memory pitfalls

Memory leaks (tardy free):

Heap memory not deallocated before the end of program

Heap memory that can no longer be accessed

Example

```
void foo(){
     int* p = new int;

}
```

Null check

```
if (q) {
    cout << *q;
}
```

only dereferences q if its not null.

Memory errors can cause your program to crash

- **Segmentation faults**: Program crashes because it attempted to access a memory location that either doesn't exist or doesn't have permission to access

- Examples of code that results in undefined behavior and potential segmentation fault

```
int arr[] = {50, 60, 70};

for(int i=0; i<=3; i++){
   cout<<arr[i]<<endl;
}
```

```
int x = 10;
int* p;
cout<<*p<<endl;
```

Detecting memory errors

- Valgrind is a tool that reports errors related to dynamic memory allocation, access and deletion
- Run valgrind on your program using the following command:

```
valgrind --leak-check=full ./myprog
```

Next time

- Recursion