University of Southern California

Viterbi School of Engineering

Software Design

Data Structures – Arrays, and Linked-Lists

Elementary Data Structures

- Elementary data structures such as stacks, queues, lists, and heaps are the “off-the-shelf” components we build our algorithms on

- There are two aspects to any data structure:
  - The abstract operations which it supports
  - The implementation of the operations

Data Abstraction

- Data structures help us describe the operations in abstract level

- Data structures do not have a unique implementation, therefore optimization for performance may be necessary

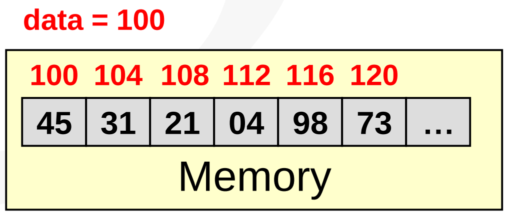
Different Categories of Data Structures

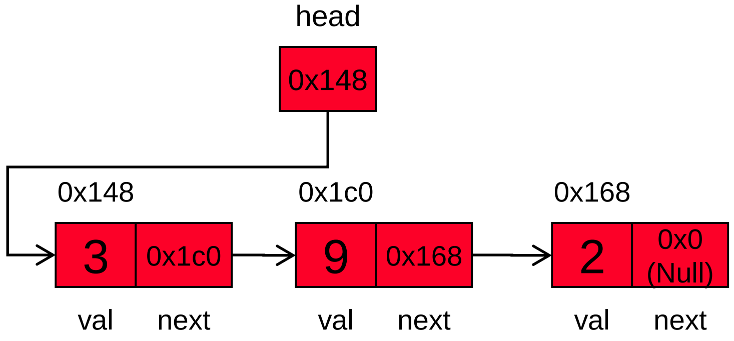
- Contiguous vs. Linked Data Structures
  - Data structures can be neatly classified as either contiguous or linked depending upon whether they are based on arrays or pointers:
    - Contiguously-allocated structures are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables
    - Linked data structures are composed of multiple distinct chunks of memory bound together by pointers, and include lists, trees, and graph adjacency lists
- Static vs. Dynamic Data Structures
  - Example: Static Array & Dynamic Array

Arrays

- Arrays are contiguous pieces of memory
- To find a single value, computer only needs
  - The start address
    - Remember the name of the array evaluates to the starting address (e.g. data = 120)
  - Which element we want
    - Provided as an index, e.g. [20]
  - This is all thanks to the fact that items are contiguous in memory
  - If we know integer element i is at location 108 do we know where element i+1 is?

```
#include<iostream>
using namespace std;

int main()
{
  int data[25];
  data[20] = 7;
  return 0;
}
```





Array Benefits

- Advantages of contiguously-allocated arrays include:
  - Constant-time access given the index
  - Arrays consist purely of data, so no space is wasted with links or other formatting information
  - Physical continuity (memory locality) between successive data accesses helps exploit the high-speed cache memory on modern computer architectures

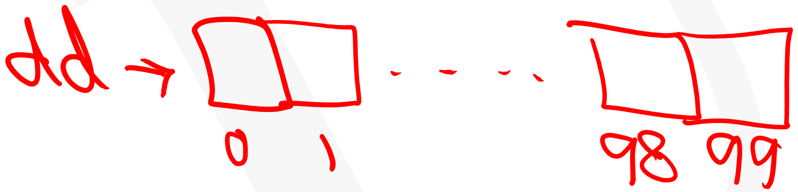
Limitations of Arrays

- You can dynamically allocate arrays once you know their size
- Example: Ask the user how many items they will need, then allocate an array for that size
- Problem: What if the user doesn't know how many they will create or simply changes their mind
  - Example:
    - cout << "How many numbers do you think you will input?" << endl;
      cin >> size;
      int *ptr = new int[size];
    - What if later the user wants to input an additional number??
    - Could allocate a new array of size+1 and copy items over but that becomes a time sink!
- Main point: Arrays, whether allocated static or dynamic cannot be easily resized easily later on

Static Arrays

- Arrays are one of the most common data structures to store collections of elements
- An array is a structure of fixed-size data records such that each element can be efficiently located by its index or (equivalently) address

int dd [100]; // dd[0] to dd[99] are uninitialized



dd[7]=29;

Dynamic Arrays

- Unfortunately we cannot adjust the size of simple arrays in the middle of a program's execution
- Compensating by allocating extremely large arrays can waste a lot of space
- With dynamic arrays we start with an array of size 1, and double its size from m to 2m each time we run out of space

```
int Size = 100;
int* dd;
dd = (int*) malloc(Size * Sizeof(int));
dd[7]=29;
free(dd)
```

```
int* dd=new int[100];
dd[7]=29;
delete[] dd;
```

Amortized Delay

- Doubling size means inserting n elements overall takes O(n) times, i.e., the amortized delay is constant

How Much Total Work?

- The apparent waste in this procedure involves the recopying of the old contents on each expansion
- If half the elements move once, a quarter of the elements twice, and so on, the total number of movements M is given by

$$M = \sum_{i=1}^{\lg n} i \cdot n/2^i = n \sum_{i=1}^{\lg n} i/2^i \leq n \sum_{i=1}^{\infty} i/2^i = 2n$$

- Thus each of the n elements move an average of only twice, and the total work of managing the dynamic array is O(n), i.e., the same as in simple array

Linked Structures (Implemented using Pointers)

- Includes pointers to represent the address of the item locations in memory

- Analogy: A cellphone number can be thought of as a pointer to its owner as they move about the planet

- In C, *p denotes the item pointed to by p, and &x denotes the address, i.e., the pointer of a particular variable x

- A special NULL pointer value is used to denote structure-terminating or unassigned pointers

Advantages of Linked Lists

- The relative advantages of linked lists over static arrays include:
  - Overflow on linked structures can never occur unless the memory is actually full
  - Insertions and deletions are simpler than those for contiguous (array) lists
  - With large records, moving pointers is easier and faster than moving the items themselves
- Compared to static arrays, dynamic arrays provide us with more flexibility on how and where we use our limited storage resources, however adding to randomly selected locations is easier in LL than in dynamic arrays

Analogy

- Natural analogy when we have a set of items that can change is to create a list
  - Write down what you know now
  - Can add more items later (usually to the end of the list)
  - Remove (cross off) others when done with them
- Can only do this with an array if you know max size of list ahead of time (which is sometimes fine)

```
1. Do the lab
2. Join ACM or IEEE
3. Play Video Games
4. Watch a movie
5. Exercise
```

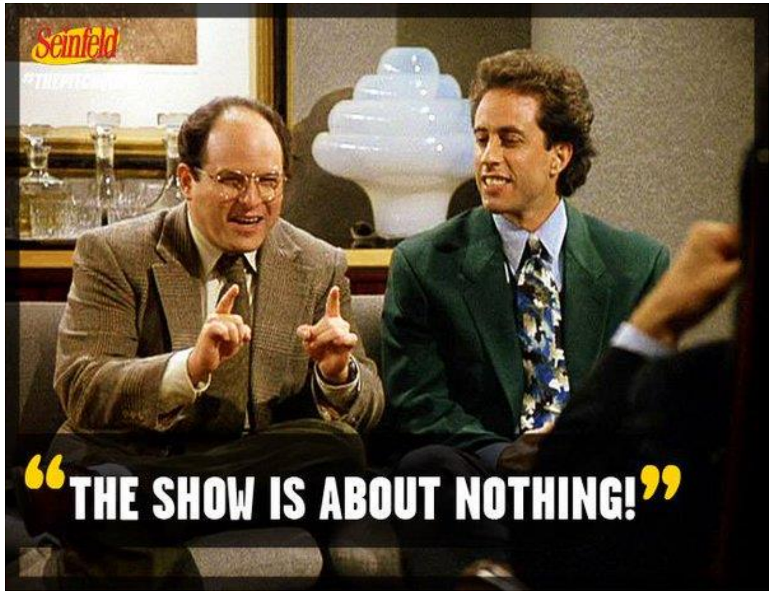
~~1. Do EE lab~~

```
2. Join ACM or IEEE
3. Play Video Games
4. Watch a movie
5. Exercise
6. Have dinner
```

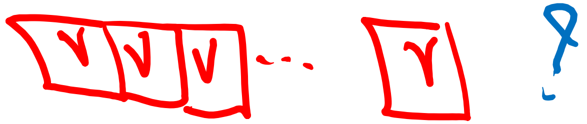
Just for Fun: Make a Linked-List ☺

NULL Pointer

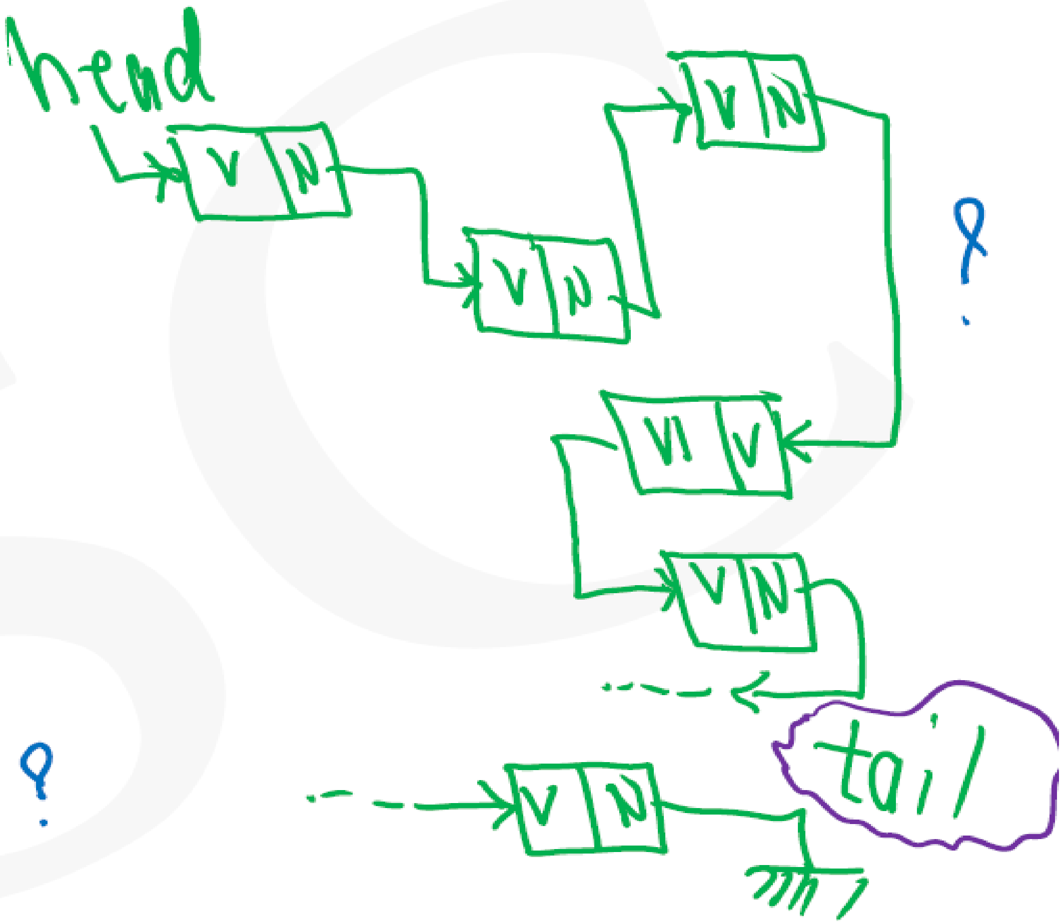
- Just like there was a null character in ASCII = '\0' whose value was 0
- There is a NULL pointer whose value is 0
  - NULL is "keyword" you can use in C/C++ that is defined to be 0
  - Used to represent a pointer to "nothing"
  - Nothing ever lives at address 0 of memory so we can use it to mean "pointer to nothing"
- int* ptr = NULL; // ptr has 0 in it now
- if(ptr != NULL){ ...   } // if it is a good pointer



Motivation – Advantages over Arrays



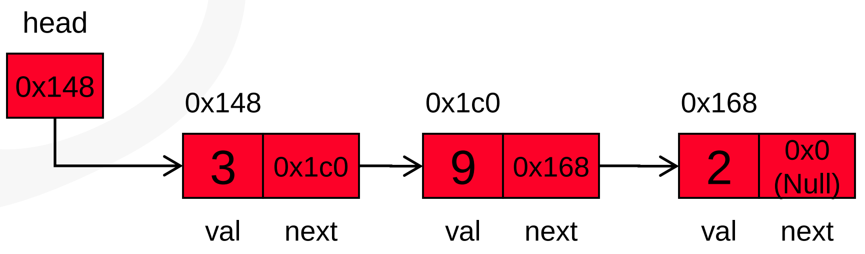
Arrays	Linked Lists
Contiguous (physically & Logically)	not physically but logically contiguous
Fast access (indexing)	Slow access (traversal)?
Expensive erase/insert	efficient resizing?
	easy erase/insert?



Complexity

Linked Lists

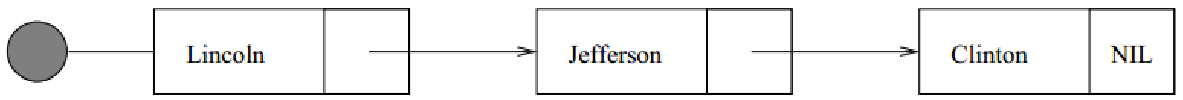
- A linked list stores values in separate chunks of memory, i.e., a dynamically allocated object
- To know where the next one is, each one stores a pointer to the next
- We can allocate more or delete old ones as needed so we only use memory as needed
- All we do is track where the first object is, i.e. the head pointer



Linked List Structures

LL using struct :

```
typedef struct list {
        item_type item;
        struct list *next;
} list;
```



Searching a List

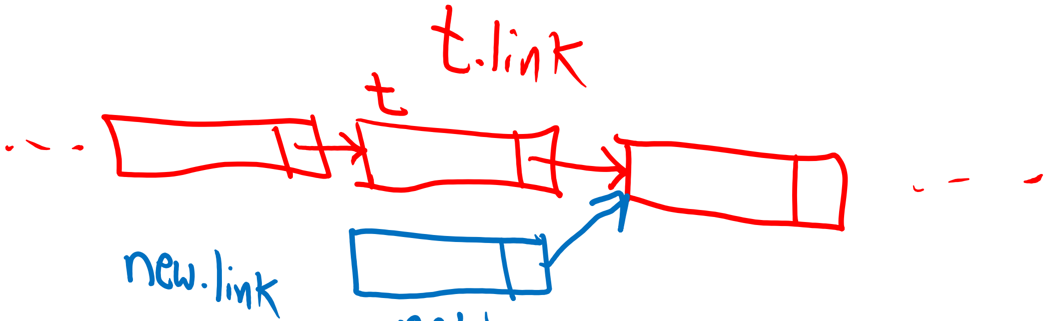
- Searching in a linked list can be done iteratively or recursively

```
list *search_list(list *l, item_type x)
{
        if (l == NULL) return(NULL);

        if (l->item == x)
                return(l);
        else
                return( search_list(l->next, x) );
}
```

Insertion into a List

- **Since we have no need to maintain the list in any particular order, we might as well insert each new item at the head**

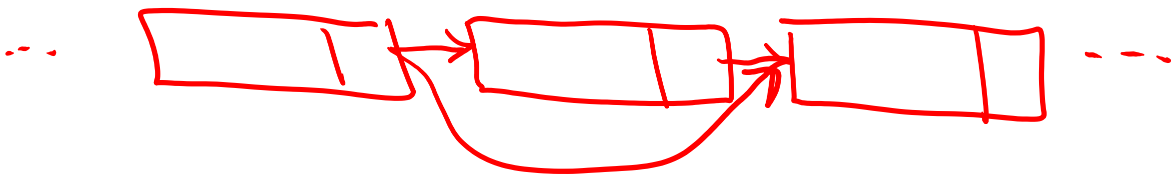
```
void insert_list(list **l, item_type x)
{
        list *p;

        p = malloc( sizeof(list) );
        p->item = x;
        p->next = *l;
        *l = p;
}
```



- **Note the **l, since the head element of the list changes**

Deleting from a List

```
delete_list(list **l, item_type x)
{
        list *p; (* item pointer *)
        list *last = NULL; (* predecessor pointer *)

        p = *l;
        while (p->item != x) { (* find item to delete *)
                last = p;
                p = p->next;
        }

        if (last == NULL) (* splice out of the list *)
                *l = p->next;
        else
                last->next = p->next;

        free(p); (* return memory used by the node *)
}
```



Linked List: class vs struct

- **Use structures/classes and pointers to make ‘linked’ data structures**
- **List**
  - **Arbitrarily sized collection of values**
  - **Can add any number of new values via dynamic memory allocation**
  - **Usually supports following set of operations:**
    - **Append (“push_back”)**
    - **Prepend (“push_front”)**
    - **Remove back item (“pop_back”)**
    - **Remove front item (“pop_front”)**
    - **Find (look for particular value)**

```
#include<iostream>

using namespace std;

struct Item {
  int val;
  Item* next;
};

class List
{
  public:
   List();
   ~List();
   void push_back(int v); ...
  private:
   Item* head;
};
```

struct Item blueprint:

int val	Item * next

class List:

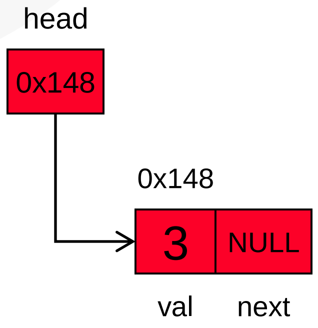
head

0x0

Rule of thumb: Still use ‘structs’ for objects that are purely collections of data and don’t really have operations associated with them. Use ‘classes’ when data does have associated functions/methods.

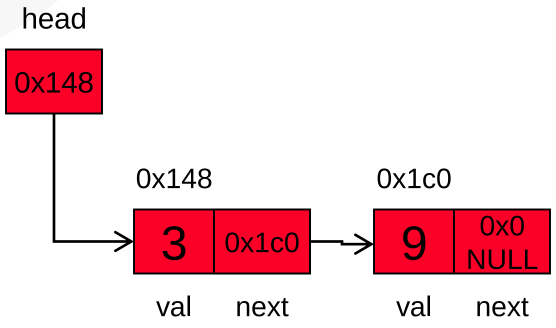
LL (cont.)

```
#include<iostream>
using namespace std;

List::List()
{
  head = NULL;
}
void List::push_back(int v){
  if(head == NULL){
    head = new Item;
    head->val = v; head->next = NULL;
  }
  else { ... }
}
int main()
{
  List mylist;
  mylist.push_back(3);
}
```



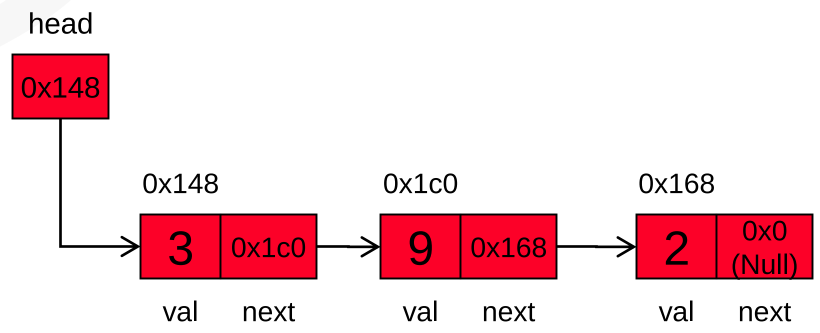
LL (cont.)


```
#include<iostream>
using namespace std;

List::List()
{
  head = NULL;
}
void List::push_back(int v){
  if(head == NULL){
    head = new Item;
    head->val = v; head->next = NULL;
  }
  else { ... }
}
int main()
{
  List mylist;
  mylist.push_back(3);  mylist.push_back(9);
}
```



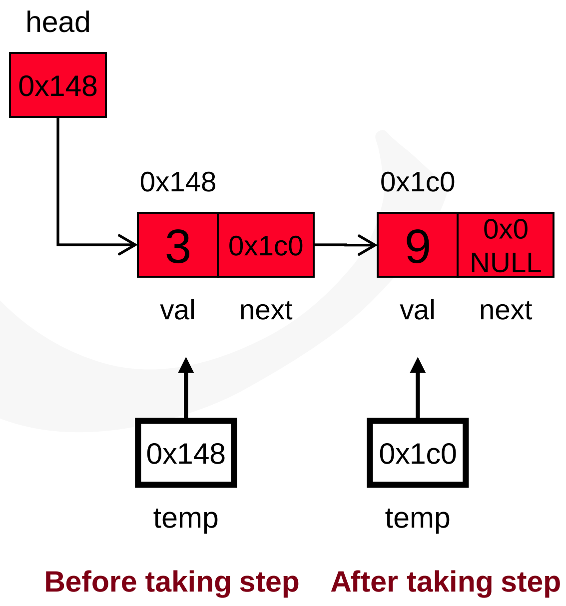
LL (cont.)

```
#include<iostream>
using namespace std;

List::List()
{
  head = NULL;
}
void List::push_back(int v){
  if(head == NULL){
    head = new Item;
    head->val = v; head->next = NULL;
  }
  else { ... }
}
int main()
{
  List mylist;
  mylist.push_back(3);  mylist.push_back(9);
  mylist.push_back(2);
}
```



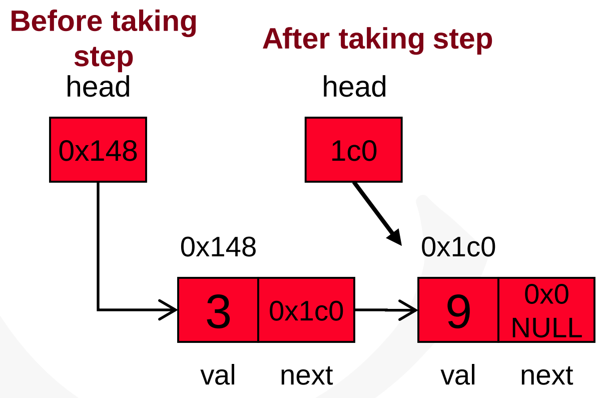
Common Linked Task/Mistake 1

- What is the C++ code to take a step from one item to the next
- Answer:
  - temp = temp->next
- Lesson: To move a pointer to the next item use: 'ptr = ptr->next'



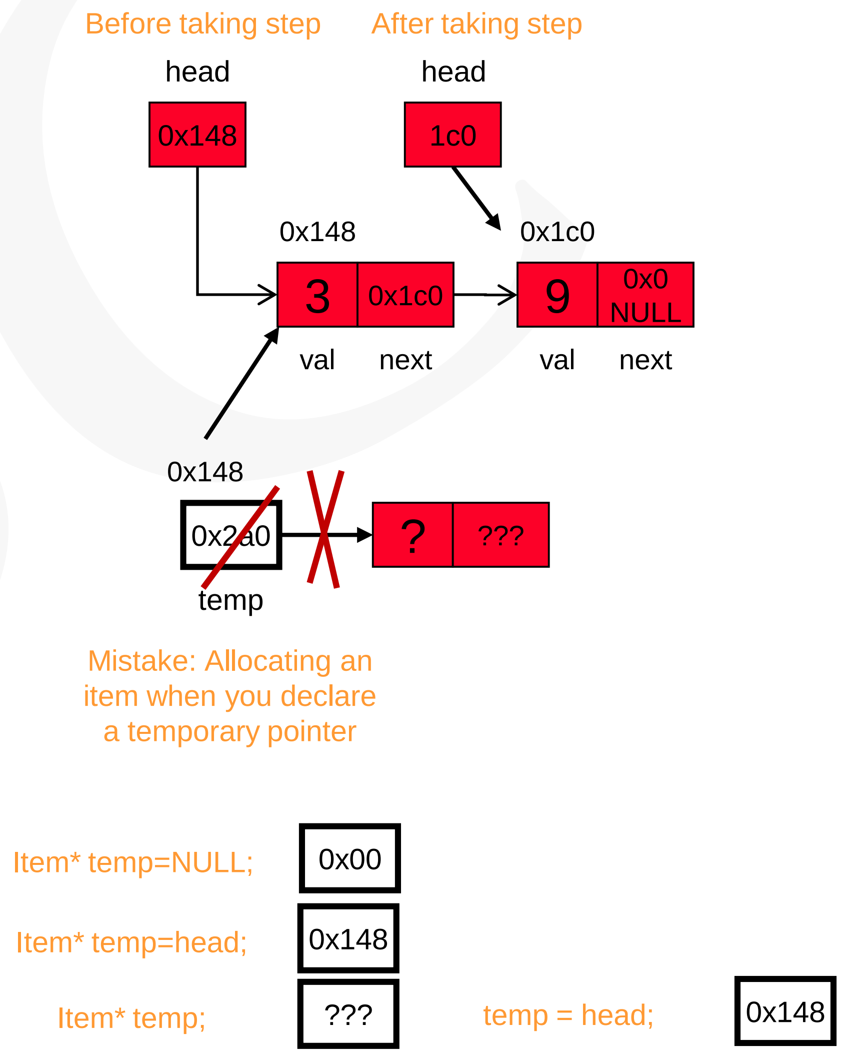
Common Linked Task/Mistake 2

- Why do we need a temp pointer? Why can't we just use head to take a step as in:
  - head = head->next;
- Because if we change head we have no record of where the first item is
  - Once we take a step we have "amnesia" and forget where we came from and can't retrace our steps
- Lesson: Don't lose your head!



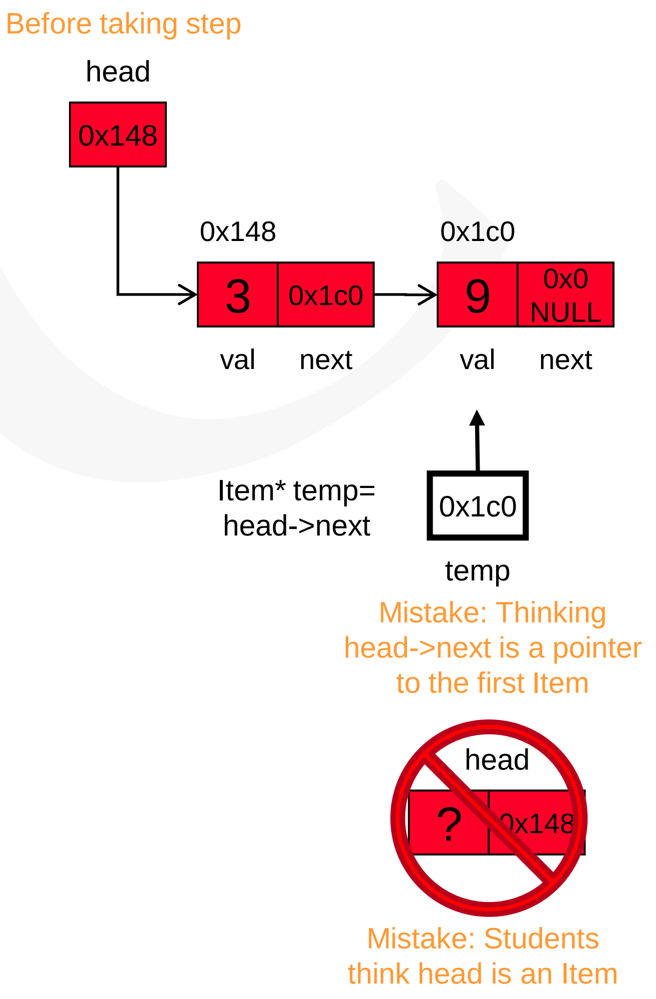
Common Linked Task/Mistake 3

- Common errors we see is that to create a temporary pointer students also dynamically allocate an item and then immediately point it at something else, causing a memory leak
  - Item* temp = new Item;
  - temp = head; or temp = head->next;
- You may declare pointers w/o having to allocate anything
  - Item* temp;
  - Item* temp = NULL;
  - Item* temp = head;
- **Lesson: Only use 'new' when you really want a new Item to come alive**



Common Linked Task/Mistake 4

- Mistake: Many students use the following code to get a pointer to the first item:
  - Item* temp = head->next;
- head (or first) pointer is NOT an actual ITEM struct
- head is just a pointer
  - It is special in that it is the only thing that is not actually holding any data...it just points at the first data-filled struct
  - head->next actually points to the 2^{nd} item, not the 1^{st} because head already points to the 1^{st} item
- Lesson: To get a pointer to the first item, just use 'head'



Exercises

- **monkey_traverse**
- **monkey_addstart**

Childs toy "Barrel of Monkeys" lets children build a chain of monkeys that can be linked arm in arm

Hatchback of monkeys

Orangutan babies

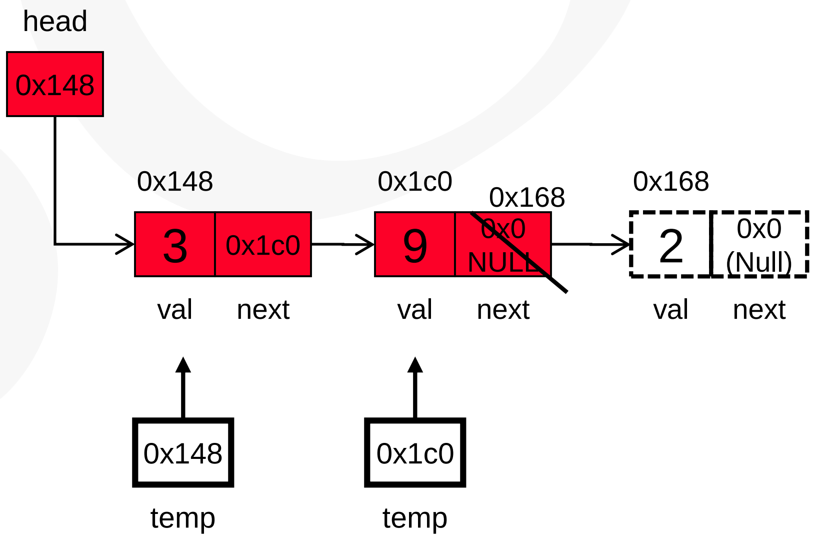
Wheelbarrow of monkeys and two guys :)

Exercise

- Write an integer linked list class
  - `listint.h, listint.cpp, listint_test.cpp`
- Examine the prototypes in listint.h (complete)
- Complete the functions in listint.cpp
- Compile and test your program the code in listint_test.cpp

Append

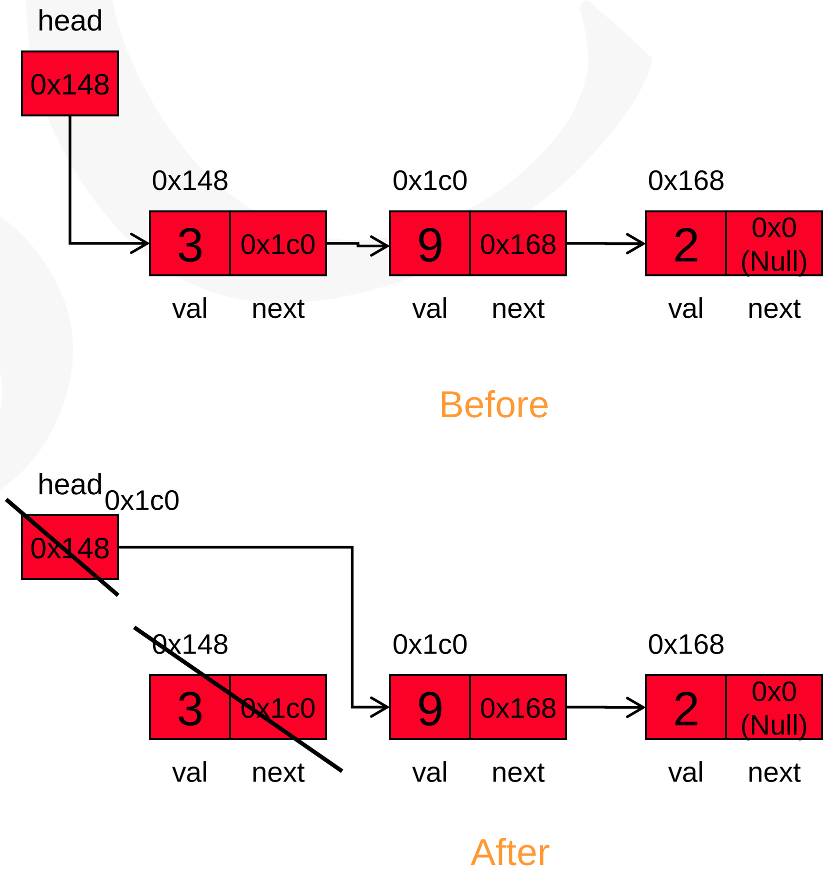
- Write a function to add new item to back of list
- Start from head and iterate to end of list
  - Copy head to a temp pointer
  - Use temp pointer to iterate through the list until we find the tail (element with next field = NULL)
  - Allocate new item and fill it in
  - Update old tail item to point at new tail item



I don't know where the list ends so I have to traverse it

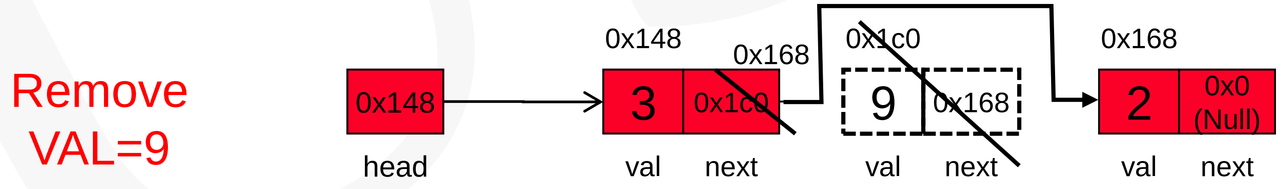
Remove First

- Write a function to remove first item
  - Copy address of first item to a temp pointer
  - Set head to point at new first item (only second item)
  - Deallocate old first item



Other Functions

- Write a function to print all items in list
  - Copy head to a temp pointer then use it to iterate over the items until the next pointer is NULL
  - Print each item as you iterate
- Find if an item in the list (return address of struct if present or NULL)
  - Copy head to a temp pointer then use it to iterate over the items until you find an item with the desired value or until next pointer is NULL
- Remove item with given value [i.e. find and remove]
  - If found, need to change the next link of the previous item to point at the item after the item found



Comparing Performance

Arrays

- Go to element at index I
  - O(1)
- Add something to the tail (assume you have a tail index)
  - O(1)
- Adding something to the front of the list after there are already n elements
  - O(n)

Linked Lists

- Go to element at index i
  - O(i)
- Add something to the tail (assume you have only head pointer and n elements in the list)
  - O(n)
- Adding something to the front of the list after there are already n elements
  - O(1)