CSCI 104

Rafael Ferreira da Silva

rafsilva@isi.edu

Slides adapted from: Mark Redekopp and David Kempe

USC Viterbi School of Engineering

XKCD #138

Courtesy of Randall Munroe @ http://xkcd.com

RECURSION (cont.)

Recursive Definitions

- N = Non-Negative Integers and is defined as:
  - The number 0 [Base]
  - n + 1 where n is some non-negative integer [Recursive]
- String
  - Empty string, ε
  - String concatenated with a character (e.g. 'a'-'z')
- Palindrome (string that reads the same forward as backwards)
  - Example: dad, peep, level
  - Defined as:
    - Empty string [Base]
    - Single character [Base]
    - xPx where x is a character and P is a Palindrome [Recursive]
- Recursive definitions are often used in defining grammars for languages and parsers (i.e. your compiler)

C++ Grammar

- Languages have rules governing their syntax and meaning
- These rules are referred to as its grammar
- Programming languages also have grammars that code must meet to be compiled
  - Compilers use this grammar to check for syntax and other compile-time errors
  - Grammars often expressed as “productions/rules”
- ANSI C Grammar Reference:
  - http://www.lysator.liu.se/c/ANSI-C-grammar-y.html#declaration

Simple Paragraph Grammar

Substitution	Rule
subject	"I" | "You" | "We"
verb	"run" | "walk" | "exercise" | "eat" | "play" | "sleep"
sentence	subject verb '.'
sentence_list	sentence | sentence_list sentence
paragraph	[TAB = \t] sentence_list [Newline = \n]

Example:

```
    I run. You walk. We exercise.
    subject verb. subject verb.
subject verb.

    sentence sentence sentence
    sentence_list sentence sentence
    sentence_list sentence
    sentence_list
    paragraph
```

Example:

```
I eat You sleep
Subject verb subject verb
Error
```

C++ Grammar

Rule	Expansion
expr	constant | variable_id | function_call | assign_statement | '(' expr ')' | expr binary_op expr | unary_op expr
assign_statement	variable_id '=' expr
expr_statement	';' | expr ';'

Example:

```
5 * (9 + max);
```

expr * (*expr* + *expr*) ;
expr * (*expr*) ;
expr * *expr*;
expr;
expr_statement

Example:

```
x + 9 = 5;
```

expr + *expr* = *expr*;
expr = *expr*;

NO SUBSTITUTION
Compile Error!

C++ Grammar

Rule	Substitution
statement	expr_statement | compound_statement | if (expr) statement | while (expr) statement …
compound_statement	'{' statement_list '}'
statement_list	statement | statement_list statement

Example:

```
while(x > 0) { doit(); x = x-2; }
while(expr) { expr; assign_statement; }
while(expr) { expr; expr; }
while(expr) { expr_statement expr_statement }
while(expr) { statement statement }
while(expr) { statement_list statement }
while(expr) { statement_list }
while(expr) compound_statement
while(expr) statement
statement
```

Example:

```
while(x > 0)
   x--;
   x = x + 5;
```

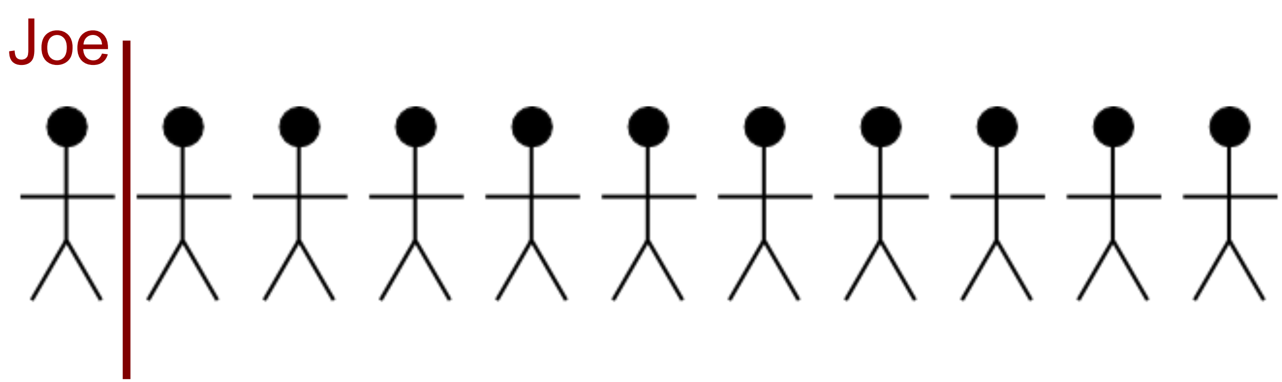
```
while(expr)
   statement
   statement
```

```
statement
statement
```

MORE EXAMPLES

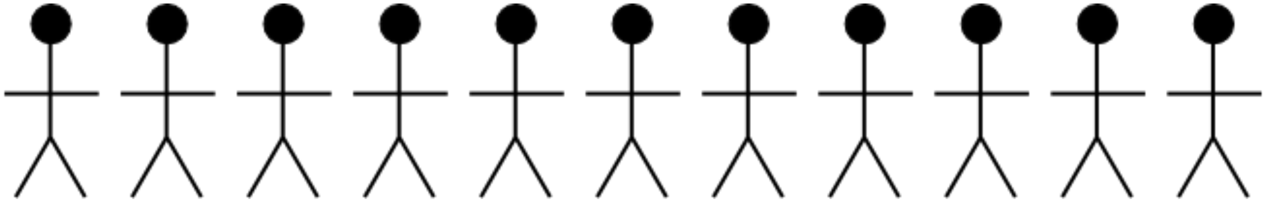
Combinatorics Examples

- Given n things, how can you choose k of them?
  - Written as C(n,k)
- How do we solve the problem?
  - Pick one person and single them out
    - Groups that contain Joe => *C(n-1, k-1)*
    - Groups that don't contain Joe => *C(n-1, k)*
  - Total number of solutions: *C(n-1,k-1) + C(n-1,k)*
  - What are base cases?



Combinatorics Examples

- You're going to Disneyland and you're trying to pick 4 people from your dorm to go with you
- Given n things, how can you choose k of them?
  - Written as C(n,k)
  - Analytical solution: C(n,k) = n! / [k! * (n-k)!]
- How do we solve the problem?



Recursive Solution

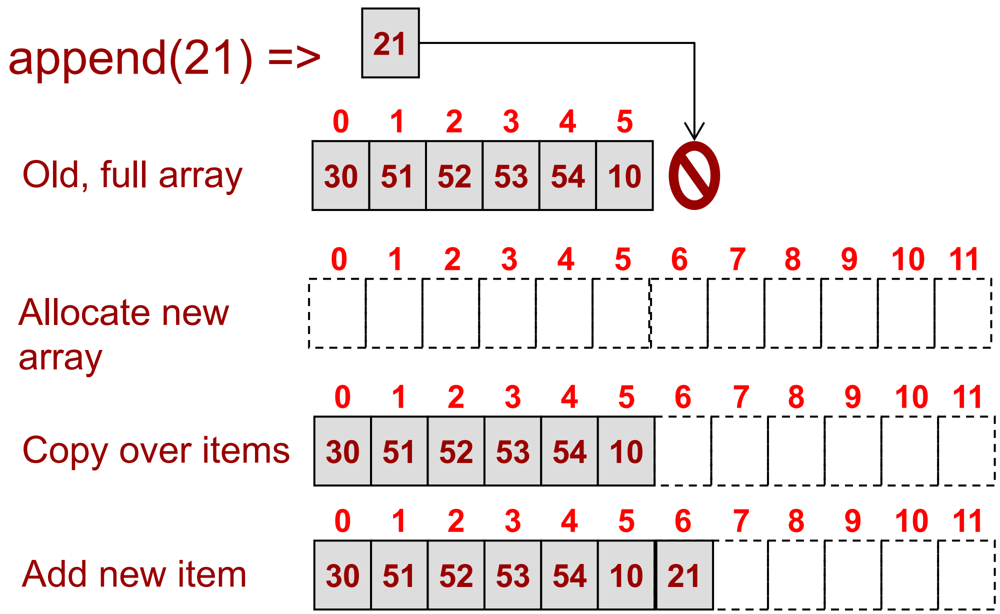
- Sometimes recursion can yield an incredibly simple solution to a very complex problem
- Need some base cases
  - C(n,0) = 1
  - C(n,n) = 1

```
int C(int n, int k)
{
if(k == 0 || k == n)
  return 1;
else
  return C(n-1,k-1) + C(n-1,k);
}
```

LINKED LISTS

Array Problems

- Once allocated an array cannot grow or shrink
- If we don't know how many items will be added we could just allocate an array larger than we need but…
  - We might waste space
  - What if we end up needing more…would need to allocate a new array and copy items
- Arrays can't grow with the needs of the client



Motivation for Linked Lists

- Can we create a list implementation that can easily grow or shrink based on the number of items currently in the list
- Observation: Arrays are allocated and deallocated in LARGE chunks
  - It would be great if we could allocate/deallocate at a finer granularity
- Linked lists take the approach of allocating in small chunks (usually enough memory to hold one item)

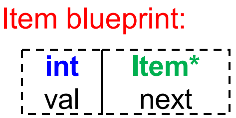
Bulk Item
(i.e. array)

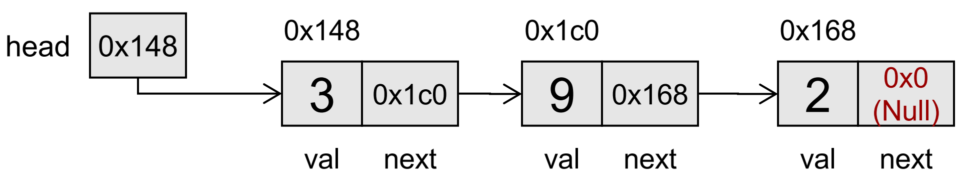
Single Item
(i.e. linked list)

Linked List

- Use structures/classes and pointers to make 'linked' data structures
- A List is…
  - Arbitrarily sized collection of values
  - Can add any number of new values via dynamic memory allocation
  - Supports typical List ADT operations:
    - Insert
    - Get
    - Remove
    - Size
    - Empty
- Can define a List class

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

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

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





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.

Don't Need Classes

- We don't have to use classes…
  - The class just acts as a wrapper around the head pointer and the operations
  - So while a class is probably the correct way to go in terms of organizing your code, for today we can show you a less modular, procedural approach
- Define functions for each operation and pass it the head pointer as an argument

```
#include<iostream>
using namespace std;
struct Item {
  int val;
  Item* next;
};

void append(Item*& head, int v);
bool empty(Item* head);
int size(Item* head);

int main()
{
  Item* head1 = NULL;
  Item* head2 = NULL;
  int size1 = size(head1);
  bool empty2 = empty(head2);
  ...
}
```

Item blueprint:

int val	Item* next

class List:

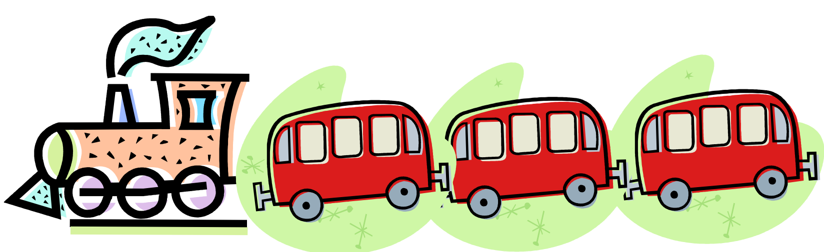
head_

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.

Linked List Implementation

- To maintain a linked list you need only to keep one data value: head
  - Like a train engine, we can attach any number of 'cars' to the engine
  - The engine looks different than all the others
    - In our linked list it's just a single pointer to an Item
    - All the cars are Item structs
    - Each car has a hitch for a following car (i.e. next pointer)



Engine = "head"

Each car = "Item"

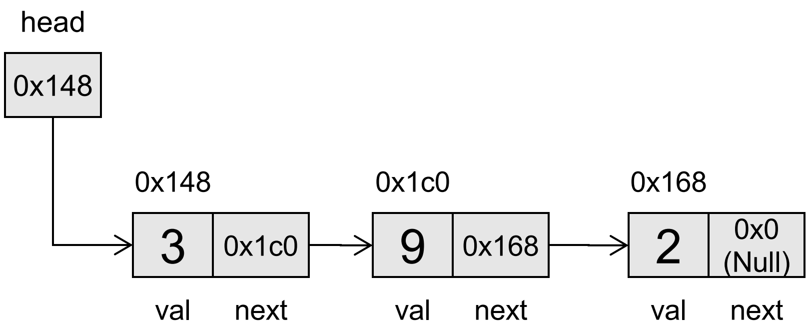
```
#include<iostream>
using namespace std;
struct Item {
   int val;
   Item* next;
};

void append(Item*& head, int v);

int main()
{
   Item* head1 = NULL;
   Item* head2 = NULL;

}
```

head1

0x0 NULL

A Common Misconception

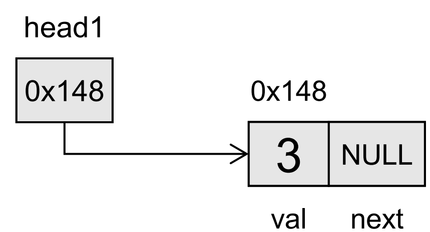
- Important Note:
  - **'head' is NOT an Item, it is a pointer to the first item**
  - Sometimes folks get confused and think head is an item and so to get the location of the first item they write 'head->next'
  - In fact, 'head->next' evaluates to the 2nd items address



Append

- Adding an item (train car) to the back can be split into 2 cases:
  - Attaching the car to the engine (i.e. the list is empty and we have to change the head pointer)
  - Attaching the car to another car (i.e. the list has other Items already) and so we update the next pointer of an Item

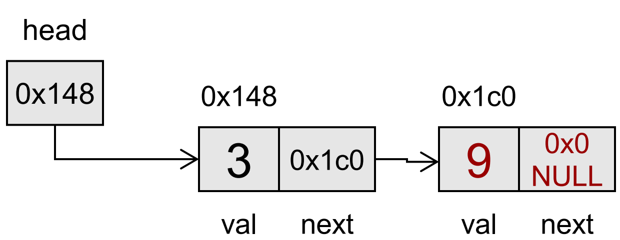
```
#include<iostream>
using namespace std;
struct Item {
  int val;
  Item* next;
};

void append(Item*& head, int v)
{
  if(head == NULL){
    head = new Item;
    head->val = v; head->next = NULL;
  }
  else {...}
}

int main()
{
  Item* head1 = NULL;
  Item* head2 = NULL;
  append(head1, 3)
}
```



Linked List

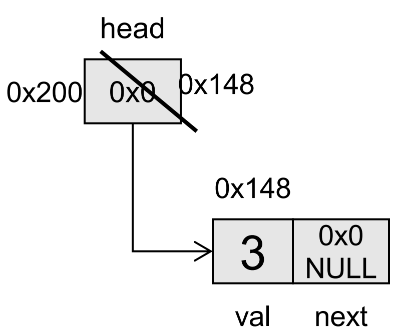
- Adding an item (train car) to the back can be split into 2 cases:
  - Attaching the car to the engine (i.e. the list is empty and we have to change the head pointer)
  - Attaching the car to another car (i.e. the list has other Items already) and so we update the next pointer of an Item

```
#include<iostream>
using namespace std;
struct Item {
  int val;
  Item* next;
};

void append(Item*& head, int v)
{
  if(head == NULL){
    head = new Item;
    head->val = v; head->next = NULL;
  }
  else {...}
}

int main()
{
  Item* head1 = NULL;
  Item* head2 = NULL;
  append(head1, 3)
}
```



Append()

- Look at how the head parameter is passed…Can you explain it?
  - Head might need to change if it is the 1st item that we are adding
  - We've passed the head pointer BY VALUE so if we modify 'head' in append() we'll only be modifying the copy
  - We need to pass the pointer by reference
  - We choose Item*& but we could also pass an Item**




```
void append(Item*& head, int v)
{
  Item* newptr = new Item;
  newptr->val = v; newptr->next = NULL;

  if(head == NULL){
    head = newptr;
  }
  else {
    Item* temp = head;
    // iterate to the end
    ...
  }
}
```

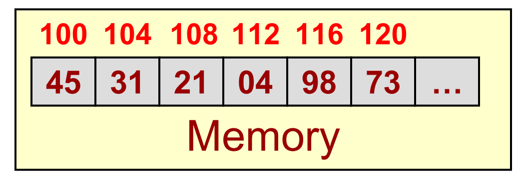
```
void append(Item** head, int v)
{
  Item* newptr = new Item;
  newptr->val = v; newptr->next = NULL;

  if(*head == NULL){
    head = newptr;
  }
  else {
    Item* temp = head;
    // iterate to the end
    ...
  }
}
```

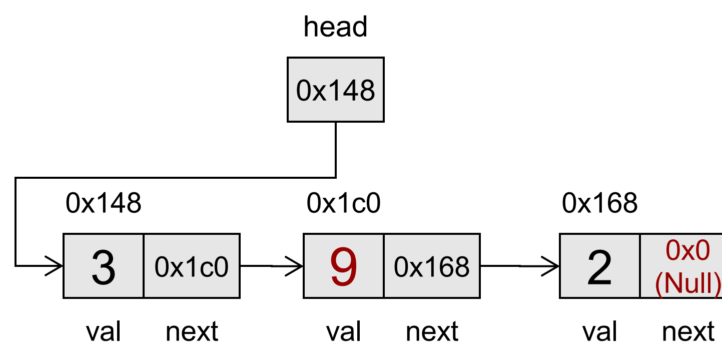
Arrays/Linked List Efficiency

- 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
- Linked list items are not contiguous
  - Thus, linked lists have an explicit field to indicate where the next item is
  - This is "overhead" in terms of memory usage
  - Requires iteration to find an item or move to the end

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

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

data = 100

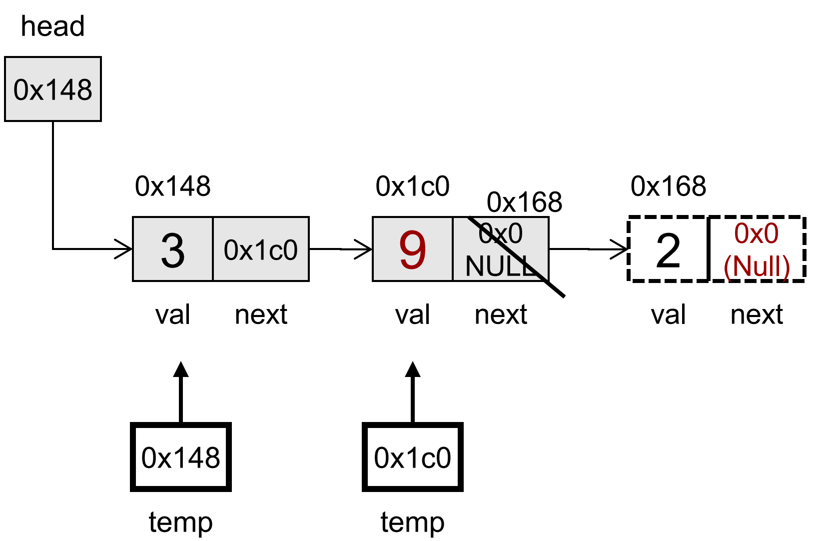




Append()

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

```
void append(Item*& head, int v)
{
  Item* newptr = new Item;
  newptr->val = v; newptr->next = NULL;

  if(head == NULL){
    head = newptr;
  }
  else {
    Item* temp = head;
    // iterate to the end
    ...
  }
}
```

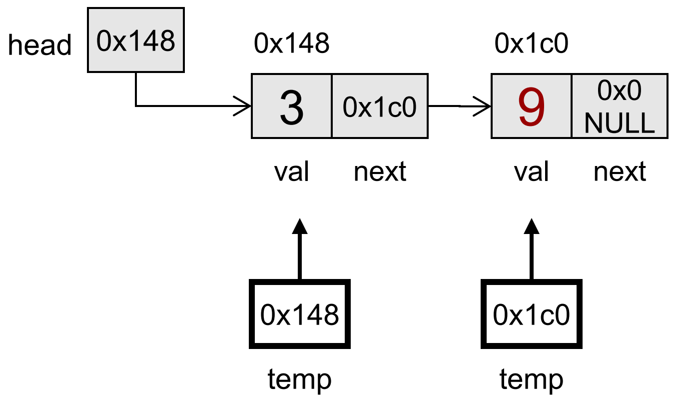


I don't know where the list ends so I have to traverse it

Iterating Over a Linked List

- To iterate we probably need to create a copy of the head pointer (because if we modify 'head' we'll never remember where the list started
- How do we take a step (advance one Item) given the temp pointer
  - temp = temp->next;

```
void append(Item*& head, int v)
{
  Item* newptr = new Item;
  newptr->val = v; newptr->next = NULL;

  if(head == NULL){
    head = newptr;
  }
  else {
    Item* temp = head;
    while(temp->next){
      temp = temp->next;
    }
    temp->next = newptr;
  }
}
```



Using a For loop

```
void append(Item*& head, int v)
{
  Item* newptr = new Item;
  newptr->val = v; newptr->next = NULL;

  if(listPtr == NULL){
    head = newptr;
  }
  else {
    Item* temp = head;     // init
    while(temp->next){     // condition
      temp = temp->next;   // update
    }
    temp->next = newptr;
  }
}
```

```
void append(Item*& head, int v)
{
  Item* newptr = new Item;
  newptr->val = v; newptr->next = NULL;

  if(listPtr == NULL){
    head = newptr;
  }
  else {
    Item* temp;
    for(temp = head;           // init
        temp->next;            // condition
        temp = temp->next) ;   // update

    temp->next = newptr;
  }
}
```

Printing Out Each Item

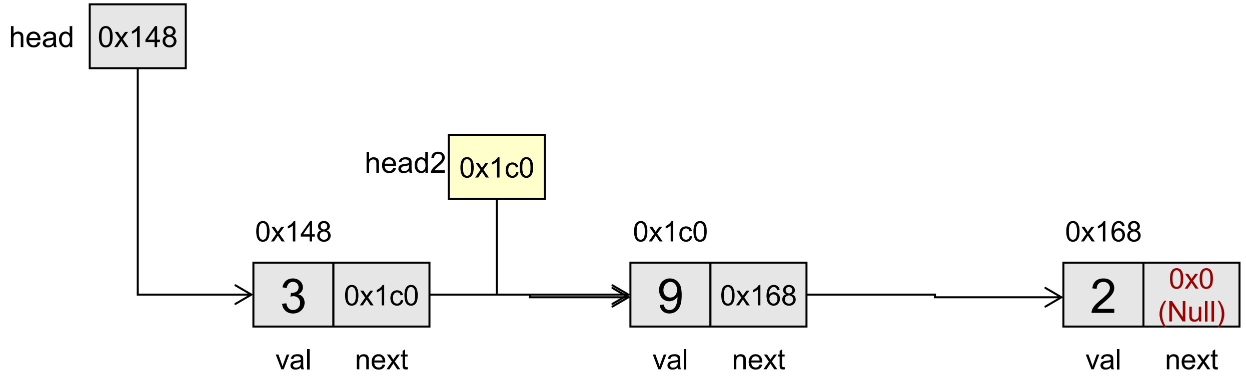
```
void print(Item* head)
{
  Item* temp = head;     // init
  while(temp) {          // condition
    cout << temp->val << endl;
    temp = temp->next;   // update
  }
}
```

```
void print(Item* head)
{
  Item* temp;
  for(temp = head;           // init
      temp;                  // condition
      temp = temp->next){    // update
    cout << temp->val << endl;
  }
}
```

RECURSION & LINKED LISTS

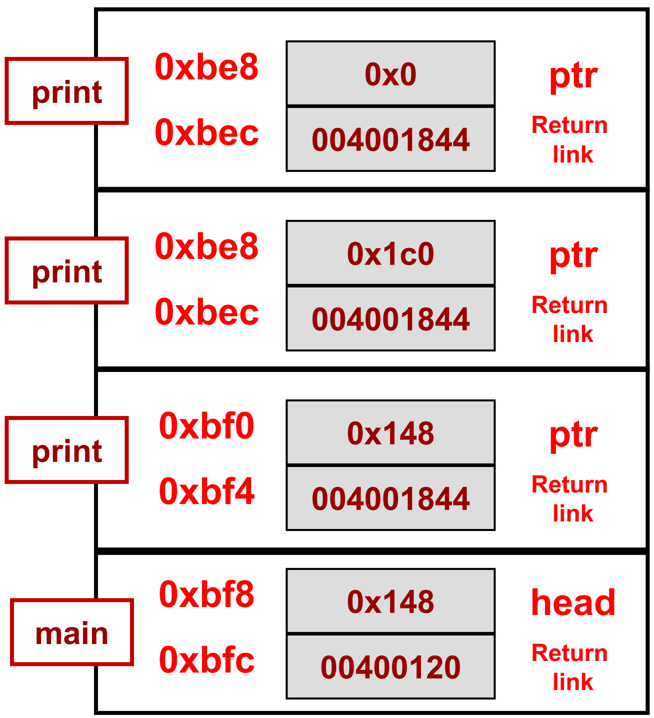
Recursion and Linked Lists

- Notice that one Item's next pointer looks like a head pointer to the remainder of the linked list
  - If we have a function that processes a linked list by receiving the head pointer as a parameter we can recursively call that function by passing our 'next' pointer as the 'head'

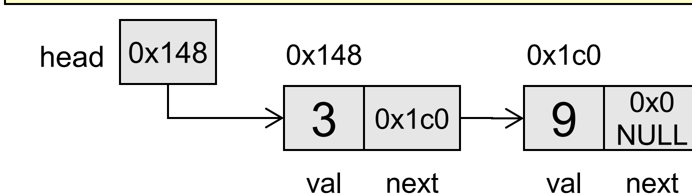


Recursive Operations on Linked List

- Many linked list operations can be recursively defined
- Can we make a recursive iteration function to print items?
  - Recursive case: Print one item then the problem becomes to print the n-1 other items.
    - Notice that any 'next' pointer can be though of as a 'head' pointer to the remaining sublist
  - Base case: Empty list (i.e. Null pointer)
- How could you print values in reverse order?

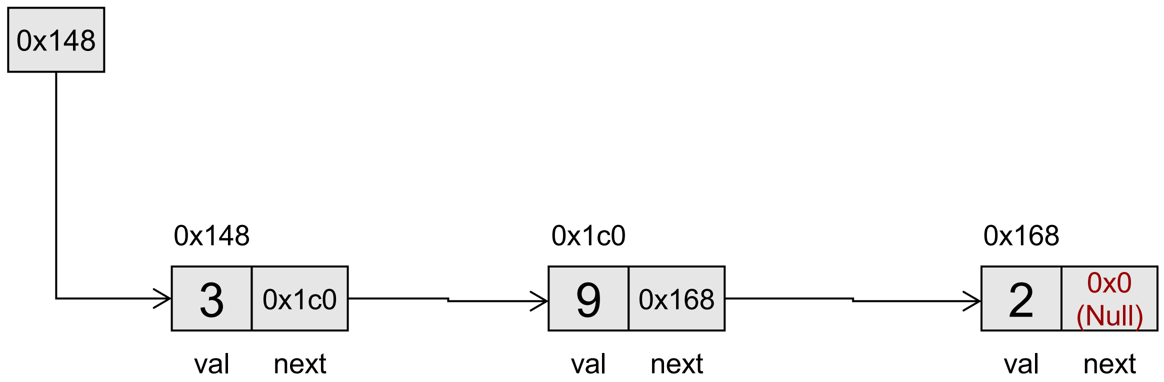



```
void print(Item* ptr)
{
  if(ptr == NULL) return;
  else {
    cout << ptr->val << endl;
    print(ptr->next);
  }
}
int main()
{ Item* head;
  ...
  print(head);
}
```



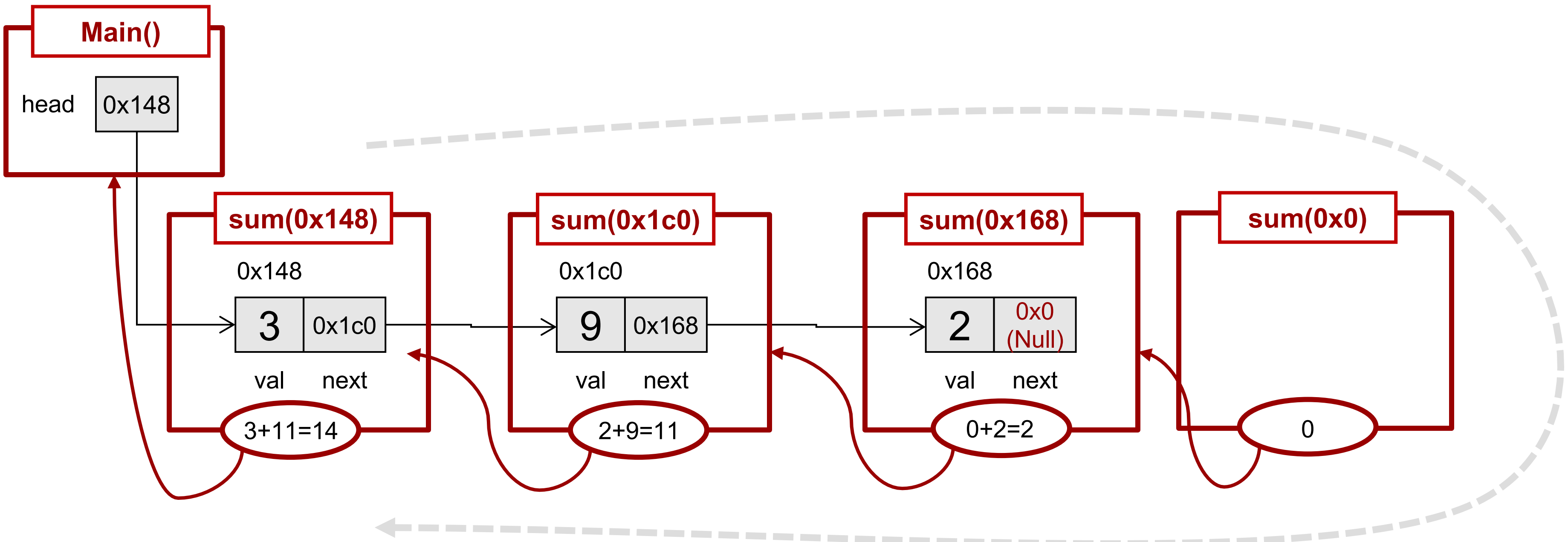
Summing the Values

- Write a recursive routine to sum the values of a linked list
  - Head Recursion (recurse first, do work on the way back up)
  - Tail Recursion (do work on the way down, then recurse)



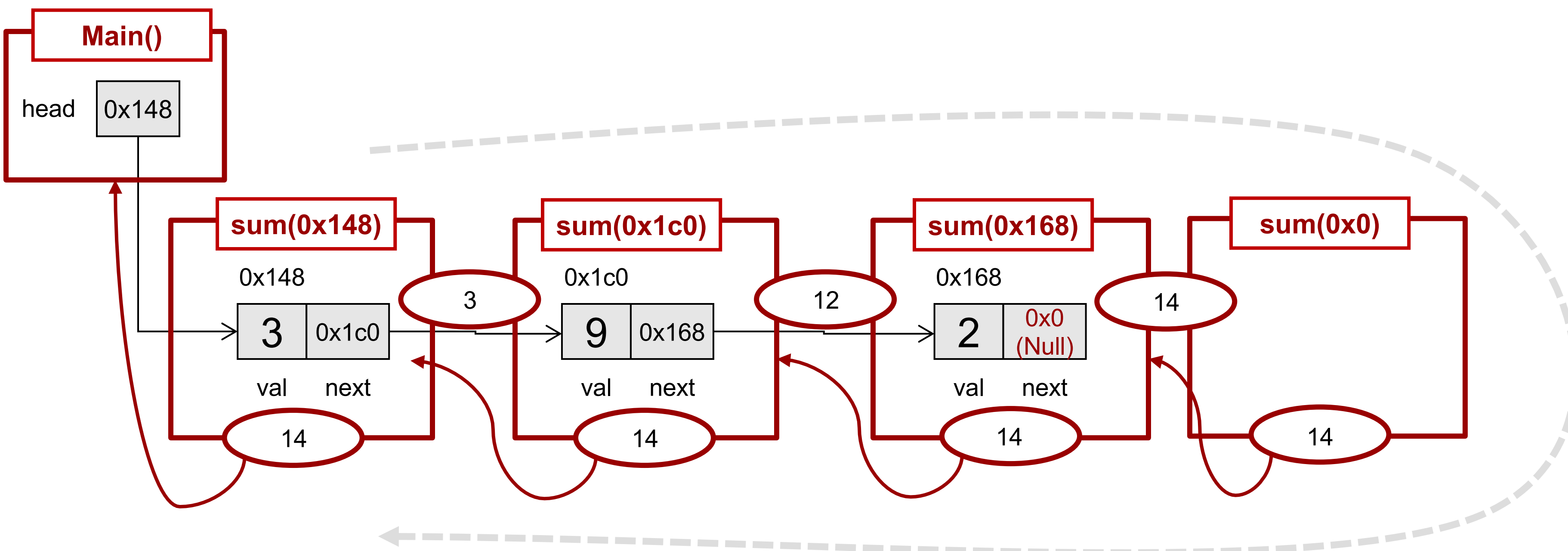
Head Recursion

- Recurse to the end of the chain (head == NULL) and then start summing on the way back up
  - What should the base case return
  - What should recursive cases (normal nodes) return?



Tail Recursion

- Produce sum as you walk down the list then just return the final answer back up the list



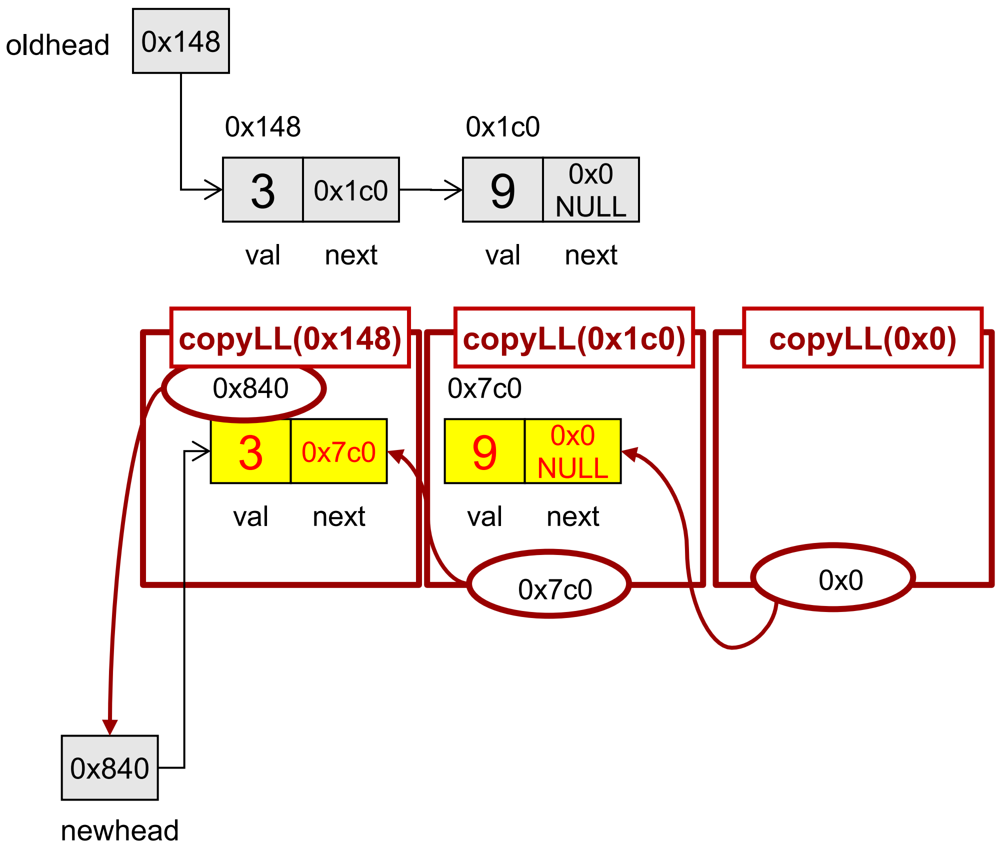
Exercises

- llsum_head
- llsum_tail

http://bits.usc.edu/cs104/exercises.html

Recursive Copy

- How could you make a copy of a linked list using recursion

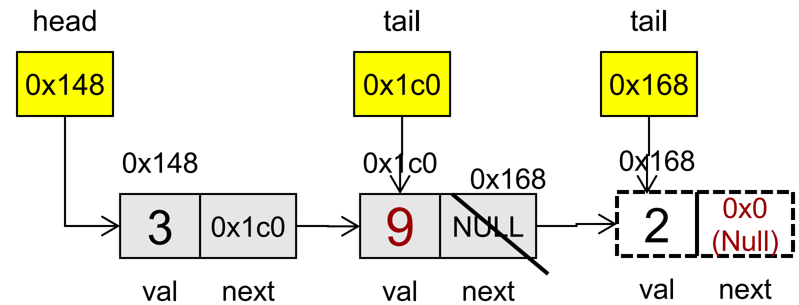



```
struct Item {
 int val;
 Item* next;
 Item(int v, Item* n){
   val = v; next = n;
 }
};

Item* copyLL(Item* head)
{
  if(head == NULL) return NULL;
  else {
    return new Item(head->val,
                    copyLL(head->next));
  }
}
int main()
{ Item* oldhead, *newhead;
  ...
  newhead = copyLL(oldhead);
}
```

INCREASING EFFICIENCY OF OPERATIONS + DOUBLY LINKED LISTS

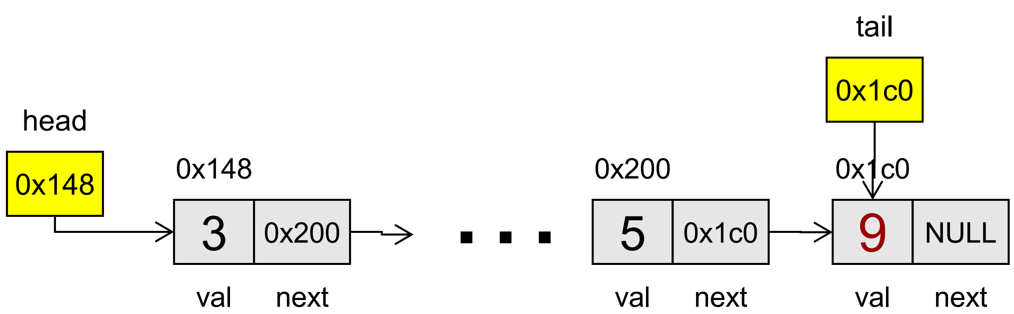
Adding a Tail Pointer

- If in addition to maintaining a head pointer we can also maintain a tail pointer
- A tail pointer saves us from iterating to the end to add a new item
- Need to update the tail pointer when…
  - We add an item to the end (fast)
  - We remove an item from the end (slow)



Removal

- To remove the last item, we need to update the 2nd to last item (set it's next pointer to NULL)
- We also need to update the tail pointer
- But this would require us to traverse the full list
- ONE SOLUTION: doubly-linked list

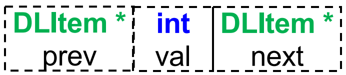


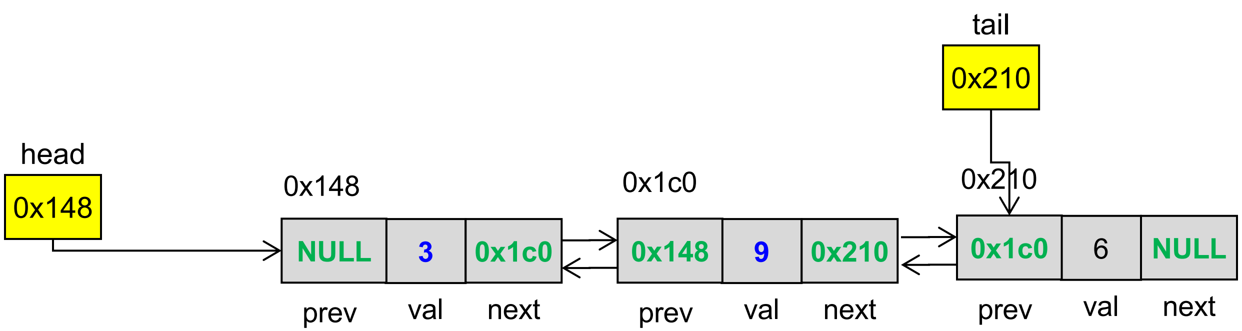
Doubly-Linked Lists

- Includes a previous pointer in each item so that we can traverse/iterate backwards or forward
- First item's previous field should be NULL
- Last item's next field should be NULL

```
#include<iostream>

using namespace std;
struct DLItem {
  int val;
  DLItem* prev;
  DLItem* next;
};

int main()
{
  DLItem* head, *tail;
};
```

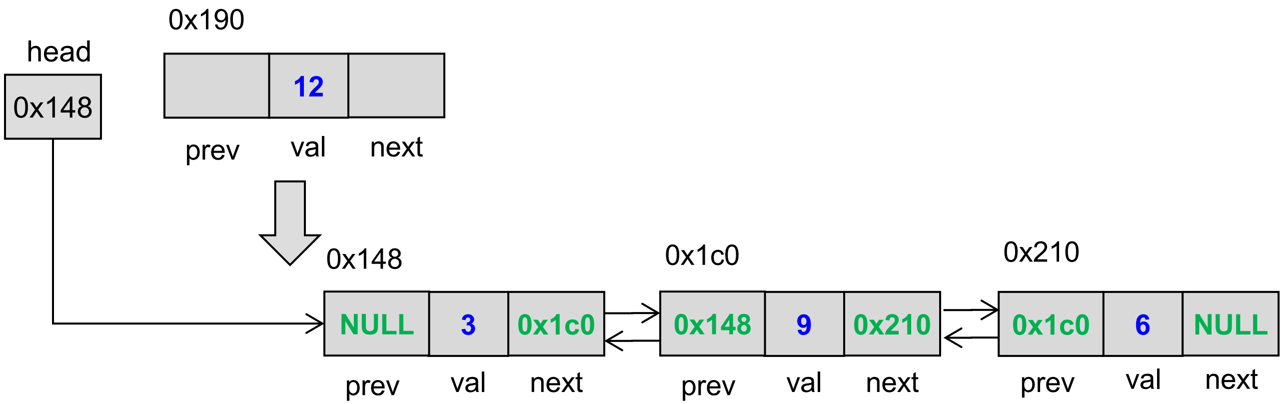
struct Item blueprint:





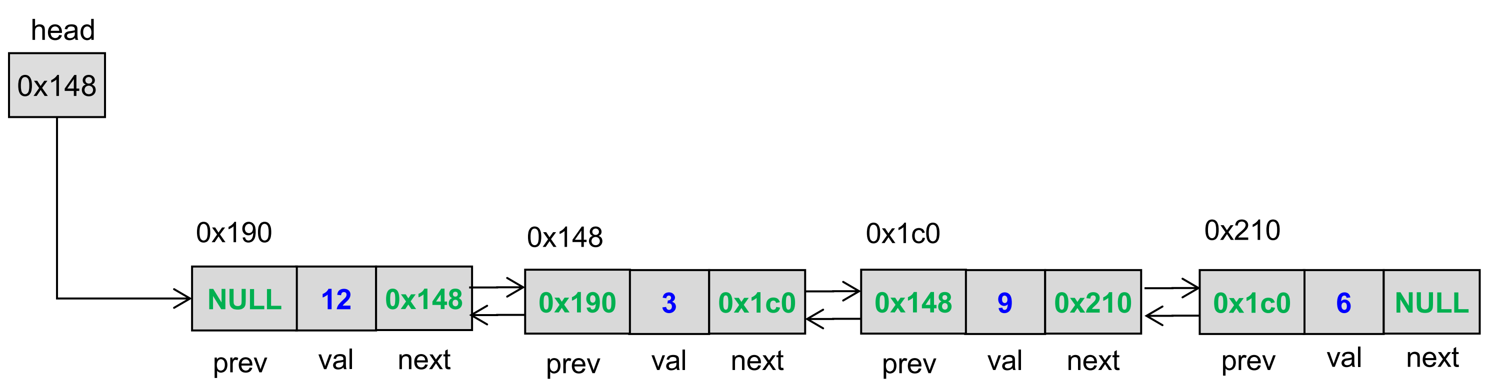
Doubly-Linked List Add Front

- Adding to the front requires you to update…
- …Answer
  - Head
  - New front's next & previous
  - Old front's previous



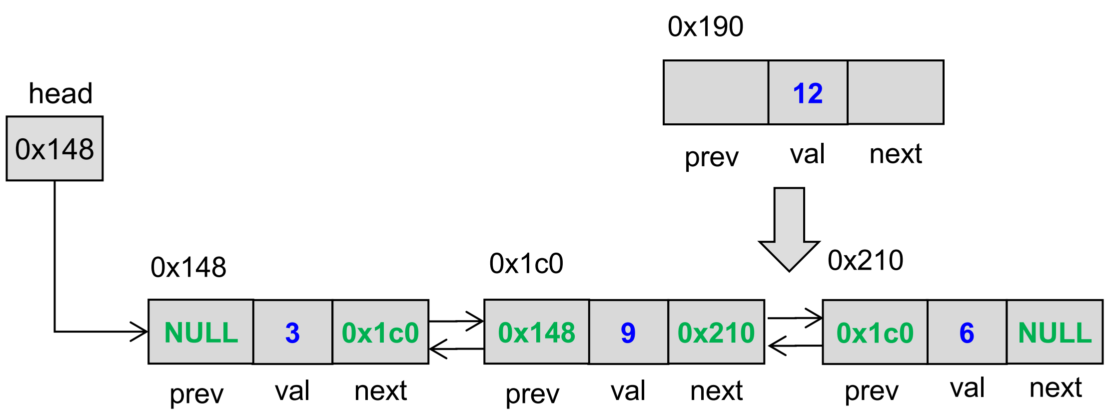
Doubly-Linked List Add Front

- Adding to the front requires you to update…
  - Head
  - New front's next & previous
  - Old front's previous



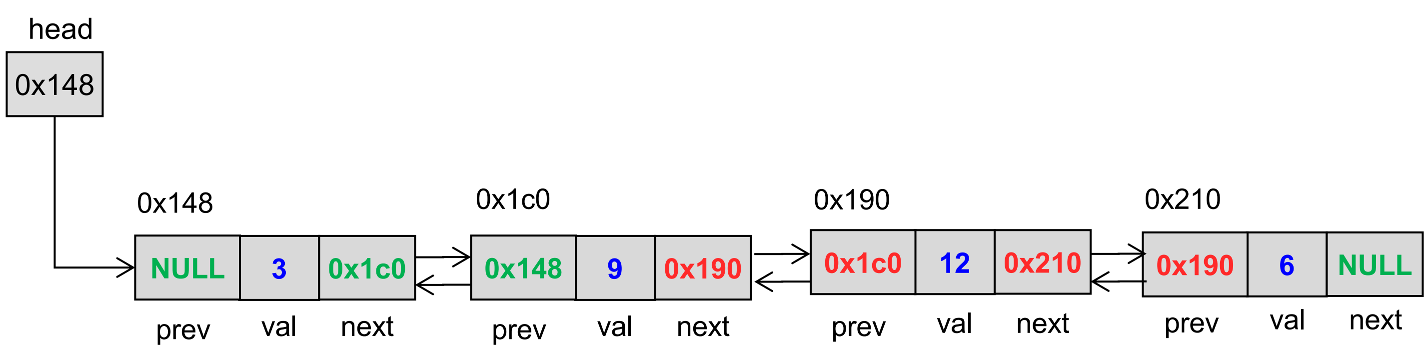
Doubly-Linked List Add Middle

- Adding to the middle requires you to update…
  - Previous item's next field
  - Next item's previous field
  - New item's next field
  - New item's previous field



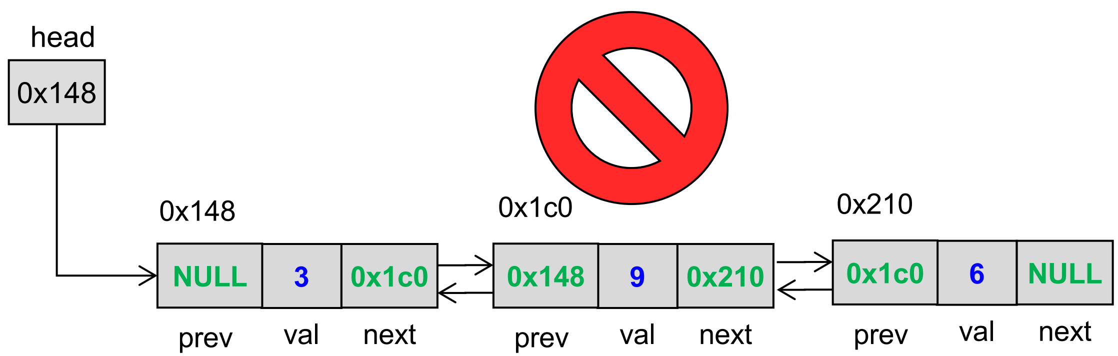
Doubly-Linked List Add Middle

- Adding to the middle requires you to update…
  - Previous item's next field
  - Next item's previous field
  - New item's next field
  - New item's previous field



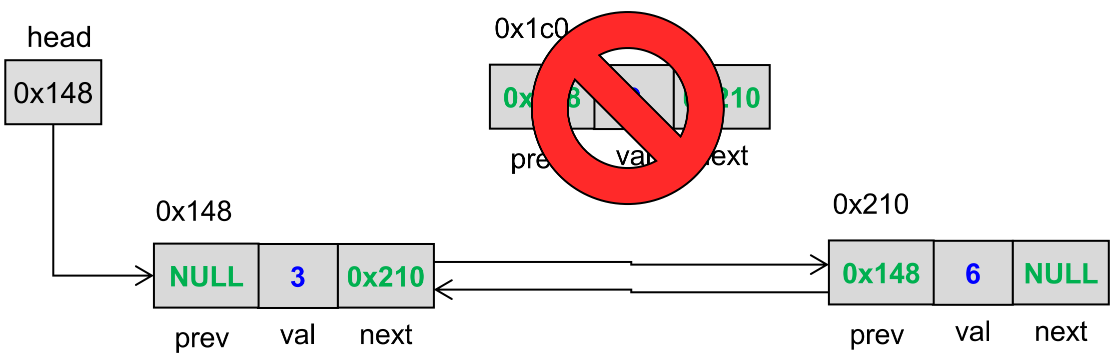
Doubly-Linked List Remove Middle

- Removing from the middle requires you to update…
  - Previous item's next field
  - Next item's previous field
  - Delete the item object



Doubly-Linked List Remove Middle

- Removing from the middle requires you to update…
  - Previous item's next field
  - Next item's previous field
  - **Delete the item object**



ABSTRACT DATA TYPE (ADT)

Abstract Data Types

- DAPS defines an **abstract data type**, or ADT, as:
  - Specification/model for a group of values/data and the operations on those values
- The model allows us to separate…
  - The decision of what data structure to use and how it will be used in our higher level application
  - And the implementation of the specific data structure
- DAPS defines a **data structure** as:
  - An implementation of an ADT in a given programming language
- Each ADT we will examine in this course has certain:
  - Well defined operations and capabilities that are often useful
  - Time & space advantages
  - Time & space disadvantages
- You need to know those operations, advantages and disadvantages

Data Abstraction & Problem Solving with C++, Carrano and Henry will henceforth be abbreviated as DAPS

3 Popular ADTs

- List
- Dictionary/Map
- Set
- (Possible 4th: Priority Queue)

Lists

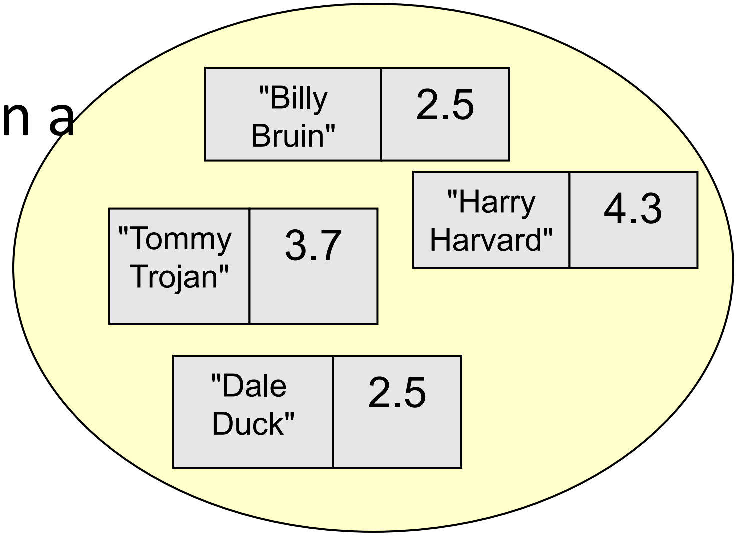
- Ordered collection of items, which may contain duplicate values, usually accessed based on their position (index)
  - Ordered = Each item has an index and there is a front and back (start and end)
  - Duplicates allowed (i.e. in a list of integers, the value 0 could appear multiple times)
  - Accessed based on their position (list[0], list[1], etc.)
- What are some operations you perform on a list?

List Operations

Operation	Description	Input(s)	Output(s)
insert	Add a new value at a particular location shifting others back	Index : int Value	
remove	Remove value at the given location	Index : int	Value at location
get / at	Get value at given location	Index : int	Value at location
set	Changes the value at a given location	Index : int Value	
empty	Returns true if there are no values in the list		bool
size	Returns the number of values in the list		int
push_back / append	Add a new value to the end of the list	Value	
find	Return the location of a given value	Value	Int : Index

Maps / Dictionaries

- Stores key,value pairs
  - Example: Map student names to their GPA
- Keys must be unique (can only occur once in the structure)
- No constraints on the values
- What operations do you perform on a map/dictionary?
- No inherent ordering between key,value pairs
  - Can't ask for the 0th item…



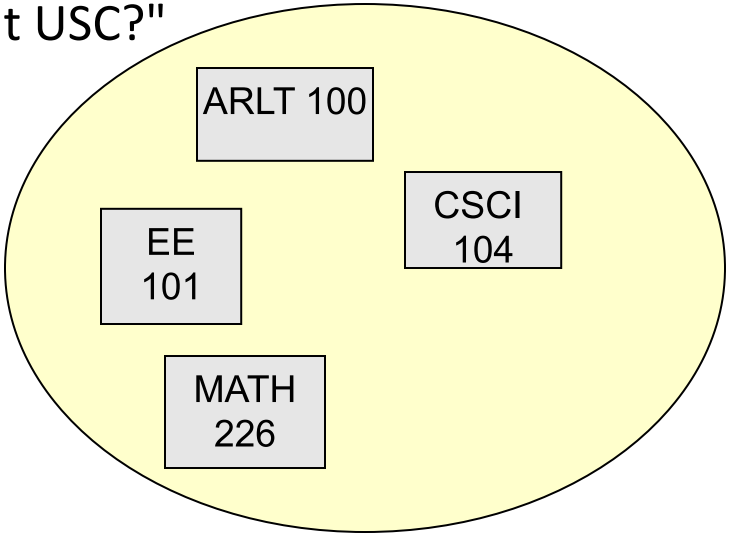
USC Viterbi School of Engineering

Map / Dictionary Operations

Operation	Description	Input(s)	Output(s)
Insert / add	Add a new key,value pair to the dictionary (assuming its not there already)	Key, Value	
Remove	Remove the key,value pair with the given key	Key	
Get / lookup	Lookup the value associated with the given key or indicate the key,value pair doesn't exist	Key	Value associated with the key
In / Find	Check if the given key is present in the map	Key	bool (or ptr to pair/NULL)
empty	Returns true if there are no values in the list		bool
size	Returns the number of values in the list		int

Set

- A set is a dictionary where we only store keys (no associated values)
  - Example: All the courses taught at USC (ARLT 100, ..., CSCI 104, MATH 226, ...)
- Items (a.k.a. Keys) must be unique
  - No duplicate keys (only one occurrence)
- Not accessed based on index but on value
  - We wouldn't say, "What is the 0^{th} course at USC?"
- In DAPS textbook Chapter 1, this is the 'bag' ADT
- What operations do we perform on a set?



Set Operations

Operation	Description	Input(s)	Output(s)
Insert / add	Add a new key to the set (assuming its not there already)	Key	
Remove	Remove	Key	
In / Find	Check if the given key is present in the map	Key	bool (or ptr to item/NULL)
empty	Returns true if there are no values in the list		bool
size	Returns the number of values in the list		Int
intersection	Returns a new set with the common elements of the two input sets	Set1, Set2	New set with all elements that appear in both set1 and set2
union	Returns a new set with all the items that appear in either set	Set1, Set2	New set with all elements that appear in either set1 and set2
difference	Returns a set with all items that are just in set1 but not set2	Set1, Set2	New set with only the items in set1 that are not in set2