LinkedLists

Dov Kruger

Department of Electrical and Computer Engineering
Rutgers University

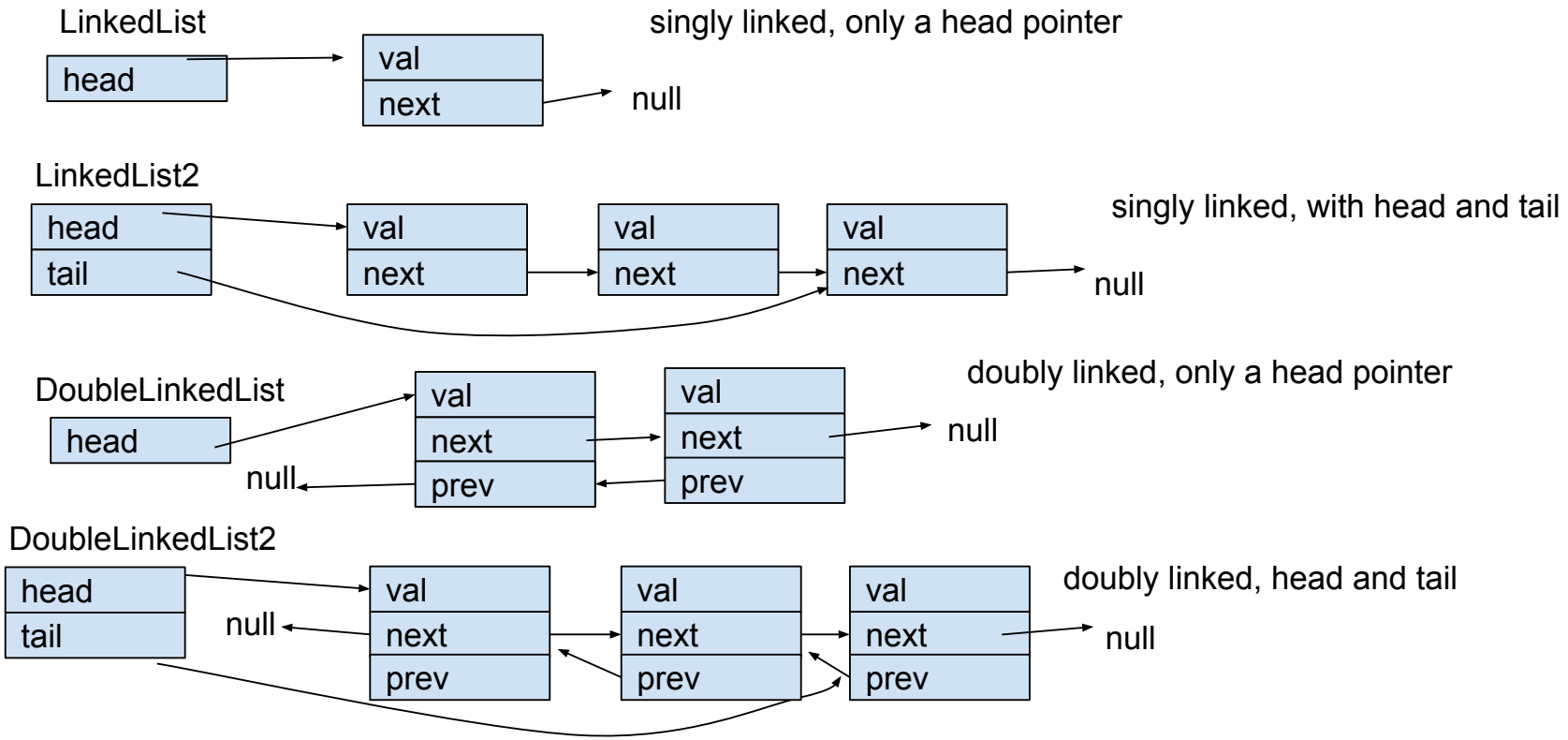
January 23, 2024

Introduction

A linked list is a data structure

- Composed of individual nodes
- Each with pointers to the next (sometimes also previous)
- The linked list can either point to the first (head) or head and tail

Four kinds of LinkedList



LinkedLists vs. Dynamic Arrays

There is no "best" list

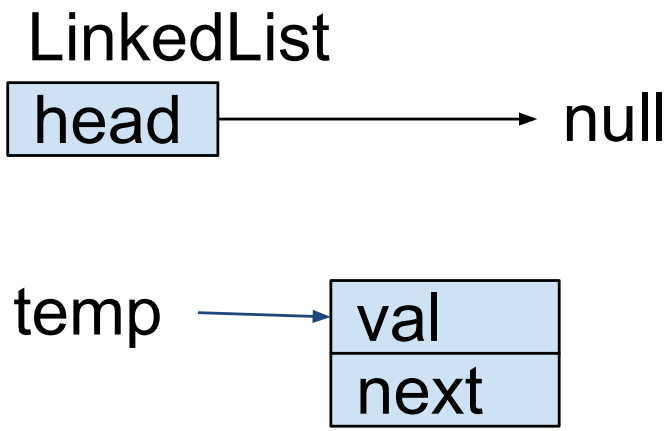
DynamicArrays

- Faster sequential append to end
- Less overall memory used

LinkedList

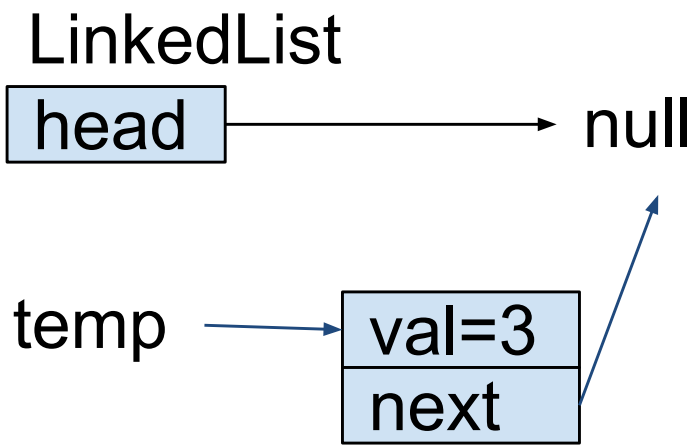
- Efficient insertion and deletion in the beginning and middle

LinkedList addStart(3)



allocate new node

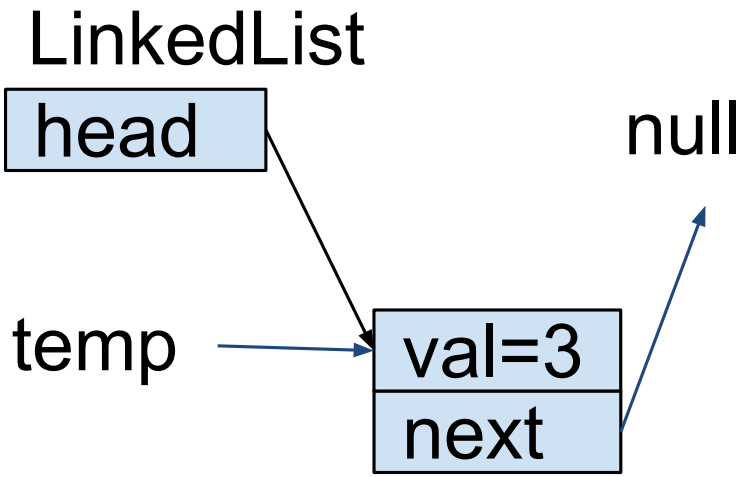
LinkedList addStart(3)



temp.val = 3
temp.next = head

set value and next pointer

LinkedList addStart(3)



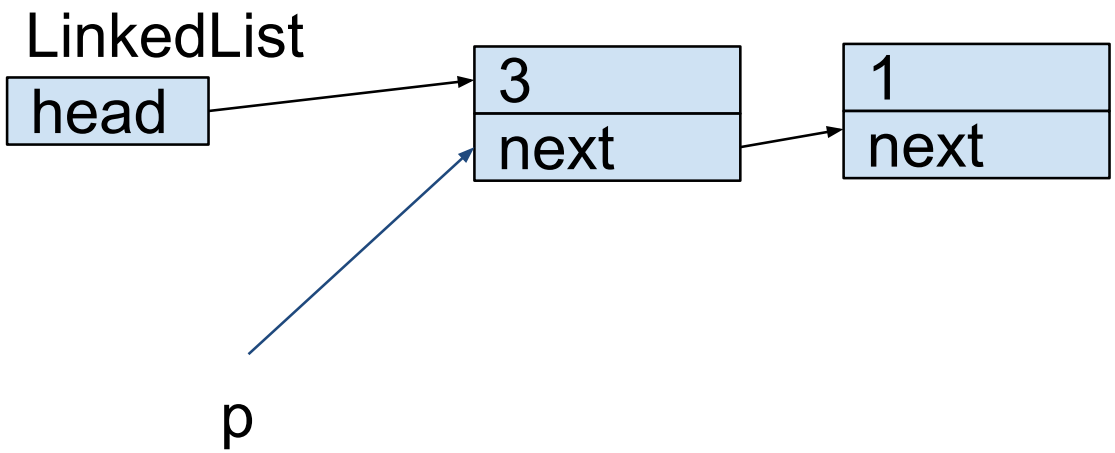
set head to point to the new node

Complexity of LinkedList addStart(v)

```
Node* temp = new Node();  // O(1)
temp->val = v;            // O(1)
temp->next = head;        // O(1)
head = temp;              // O(1)
```

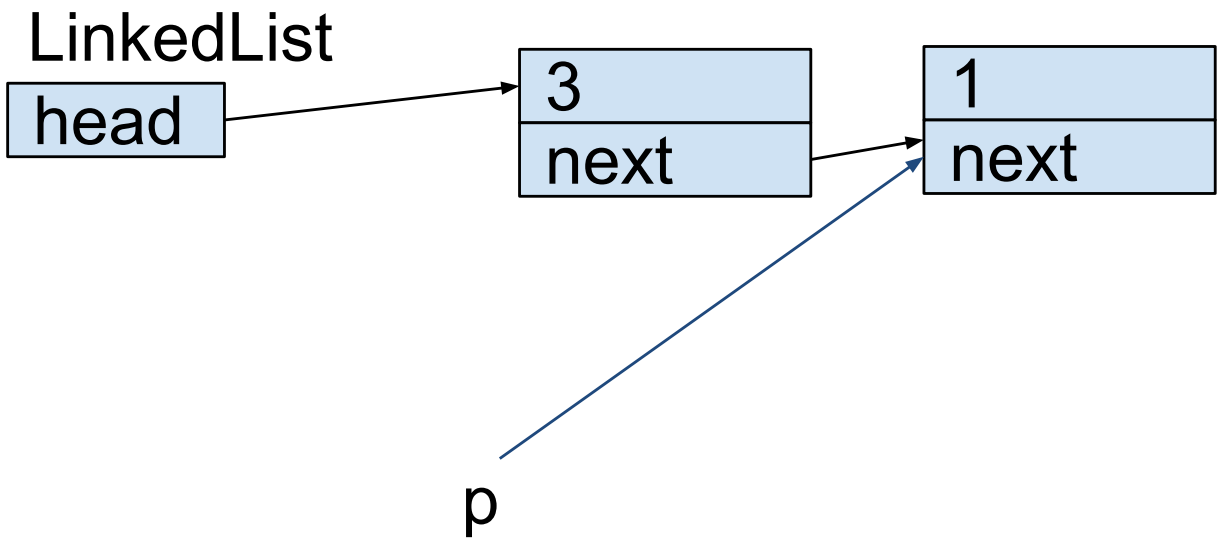
Complexity of addStart is O(1)

LinkedList addEnd(4)



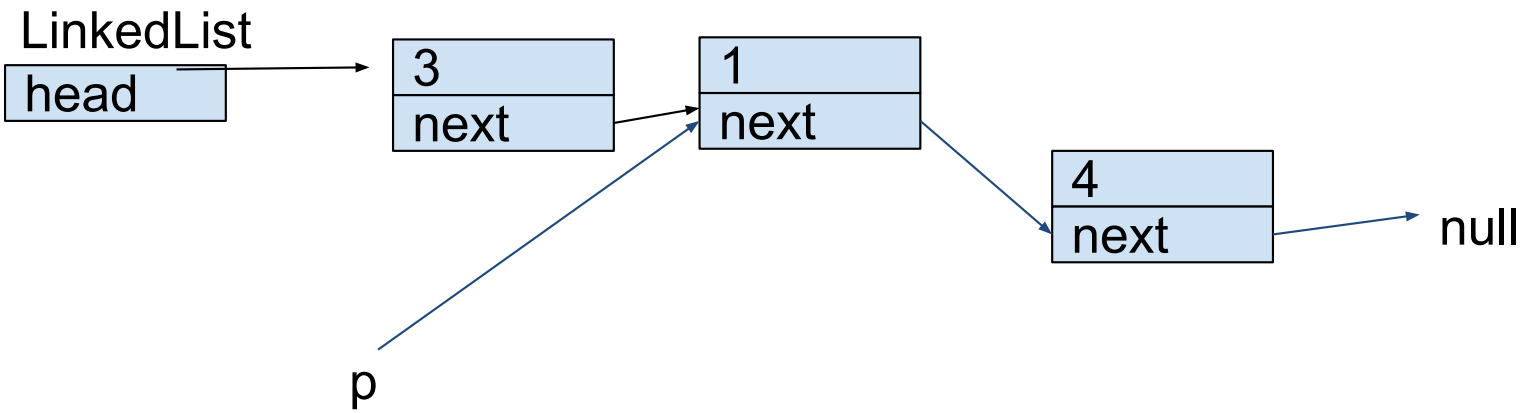
point to first element in list

LinkedList addEnd(4)



… find last element in list

LinkedList addEnd(4)



add new node to the end and set its value to 4 and next pointer to null

LinkedList addEnd(4) steps

In C++

```
Node* p = head;
while (p->next != nullptr)
  p = p->next;
Node* temp = new Node();
temp->val = 4;
value->next = nullptr;
p->next = temp;
```

Note: Not while (p != null)

Otherwise at the end of the loop, p is null, and all you know is, you arrived at the end!

LinkedList addEnd(4) Complexity

In C++

```
Node* p = head;                    //O(1)
while (p->next != nullptr)         //O(n)
   p = p->next;
Node* temp = new Node();           //O(1)
temp->val = 4;                     //O(1)
value->next = nullptr;             //O(1)
p->next = temp;                    //O(1)
```

LinkedList removeStart() Complexity

```
Node* p = head;          //O(1)
head = p->next;          //O(1)
delete p;                //O(1)
```

Implementation Issues: C++

There are many classes where the word "Node" is used
Generally, nest the class Node inside so it does not collide with the other nodes
Also, just because a class is nested inside does not mean its private components are accessible to the outer class. Make everything in Node public

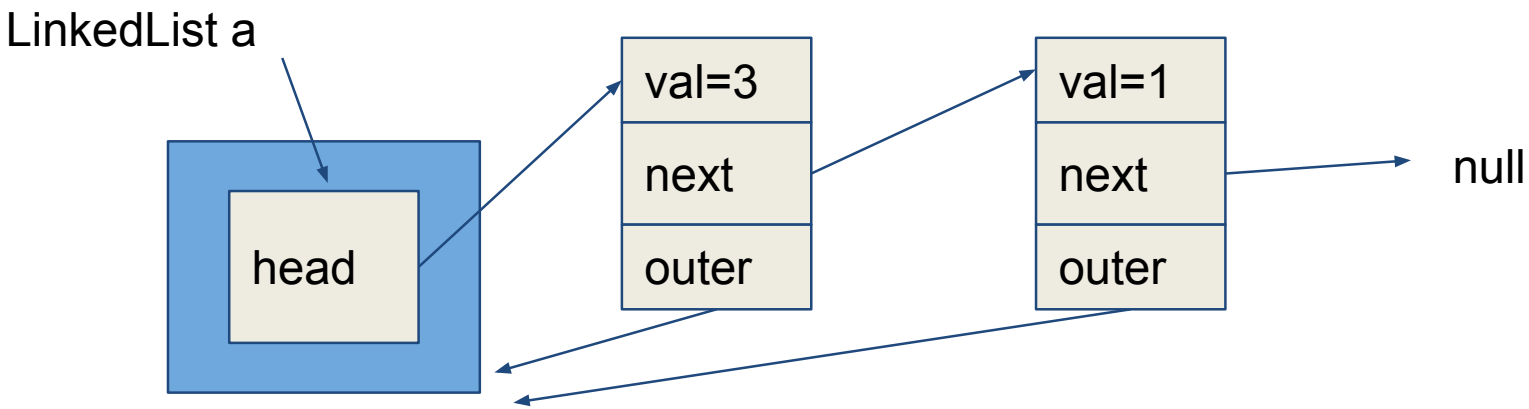
```
class LinkedList {
  class Node {
    int val;
    Node* next; // this is all private by default!
  };
  Node* head;
  // class LinkedList cannot access val and next inside Node
```

C++ LinkedList Overview

```
class LinkedList {
private:
  class Node {// no one else needs to know about
Node but me!
  public: // everything in Node is accessible to
LinkedList
    int val;
    Node* next;
  };
  Node* head; // pointer to first Node
  ...
}
```

Implementation Issues: Java

In Java, a class Node declared within the outer class is called an inner class Inner classes are completely available to the outer class They also have an extra pointer to the outer class. The following diagram shows the situation



Memory Allocation Overhead

A system to allocate memory requires overhead to track the blocks of memory.

- In C++ each block of memory is preceded by one 64-bit number.

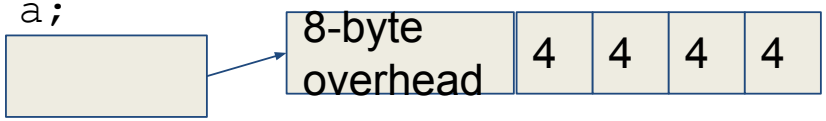
Overhead of memory allocation is 8 bytes

- In Java, every object incurs 12 bytes overhead
- No inline objects, so there is also a 4-byte pointer to the block

Allocating a single contiguous chunk of memory is much faster than many separate chunks

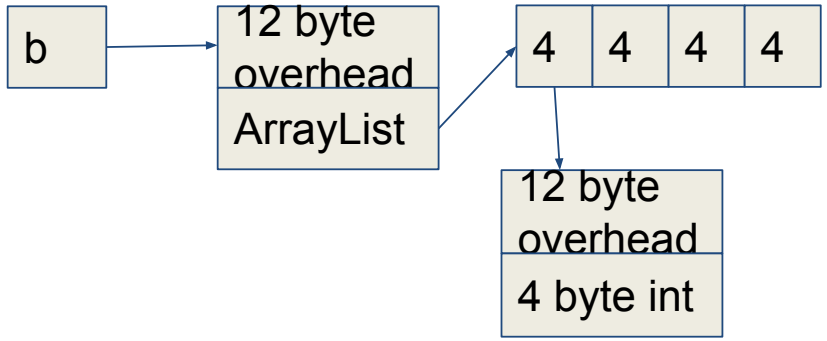
Example: C++ vector<int> vs. Java ArrayList<Integer>

```
vector<int> a;
```



size = 8 + n * sizeof(int)
sizeof(int) = 4

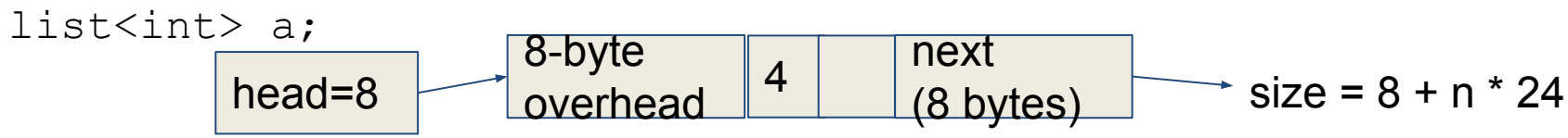
```
ArrayList<Integer> b = new ArrayList<>();
```



including the variable b
size = 20 + n * 20

this is 5 times larger

Example: C++ list<int> vs. Java LinkedList<Integer>



C++ Pointers are 64-bit (8 bytes each)

For speed, 64-bit objects are placed on 8-byte alignment

The blank space in the diagram is 4 bytes wasted in order to align the next pointer

Every time memory is allocated with new or malloc

- 8 bytes (1 pointer) overhead is used

Example: Java LinkedList<Integer> (Poorly)

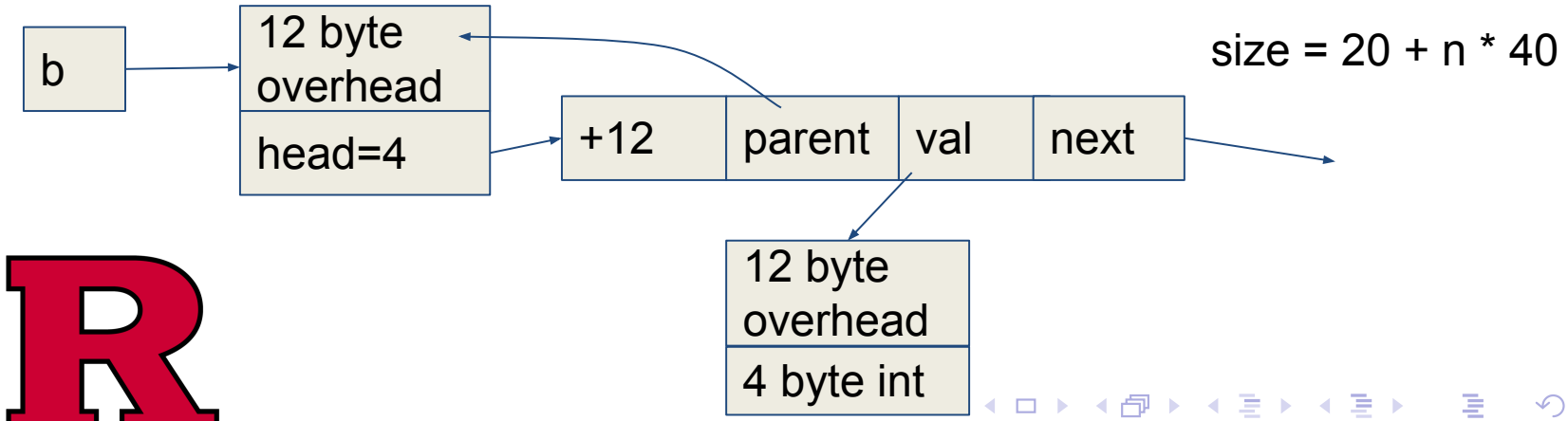
Inner class not declared static

- Each Node has extra pointer to parent
- Integer is a separate object

Java Pointers are 4 bytes not 8

- Size advantage over C++

```
public class LinkedList {
  private class Node {
    Integer val;
    Node    next;
  }
LinkedList<Integer> b = new
LinkedList<>();
```

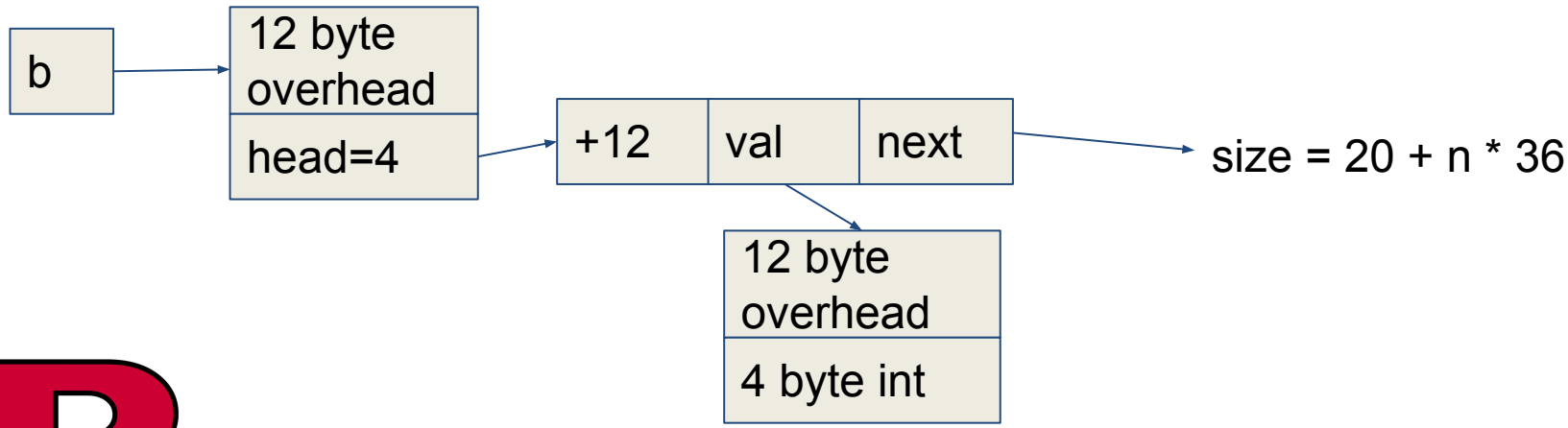


Example: Java LinkedList<Integer> (Better)

This is slightly better

- Inner node declared static
- Integer is still a separate object
- This is the library implementation

```
public class LinkedList {
  private static class Node {
    Integer val;
    Node    next;
  }
LinkedList<Integer> b =
  new LinkedList<>();
```



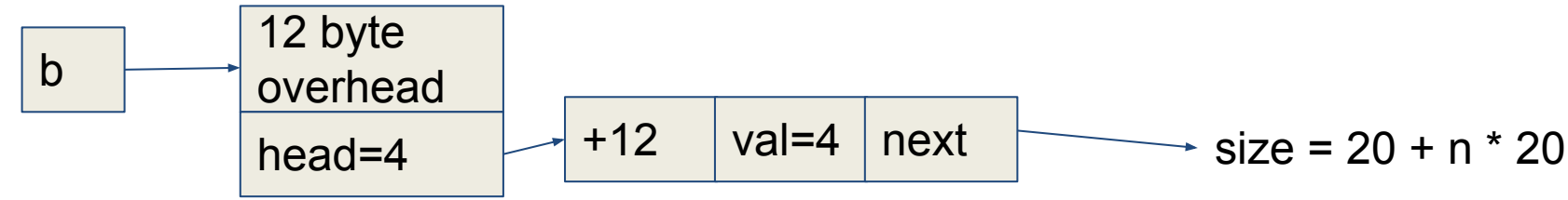
Example: Java More Efficient LinkedList of int

[

2]
For int, dramatic space savings are possible

- Inner node declared static
- int is stored in the Node

```
public class IntLinkedList
  private static class Nod
    int val;
    Node     next;
  }
IntLinkedList b =
  new IntLinkedList();
```



Insertion into a LinkedList

O(1) time if you have the pointer to node available

O(n) if you only have the integer position

Iterators

In order to traverse a list efficiently, we need to know where we are

- For LinkedLists, using an integer position is not efficient
- Integer positions require starting at the beginning and finding the location each time
- A pointer to node is efficient but requires the programmer to know the structure of the list

An Iterator is a popular design pattern

- Tracks position in something complex, hiding the details
- Write an iterator in C++ or Java to maintain optimal efficiency, keep the list simple

Example of Iterators in C++ Library

```
list<int> a;
for (int i = 0; i < 10; i++)
  a.push_back(i);

for (list<int>::iterator i = a.begin(); i != a.end(); ++i)
 out << *i << ' ';

for (auto i = a.begin(); i != a.end(); ++i)
  out << *i << ' ';
```

Example of Iterators in Java Library

```
LinkedList<Integer> a = new LinkedList<>();
for (int i = 0; i < 10; i++)
  a.add(i);

for (Iterator i = a.iterator(); i.hasNext(); ) {
  System.out.print(i.next() + "␣");
```

Implementing an Iterator

Implementing an Iterator is beyond the scope of this course

- It involves a knowledge of the implementation language and style
- See CPE-553 (C++) or CPE-552(Java) past course videos for examples

A stack is an abstract data type with 3 basic operations
Last In, First Out (LIFO)
Only the top of the stack can be accessed

- push() (add to the top of the stack)
- pop() (remove from the top of the stack)
- isEmpty() (return true if the stack has no elements)

There can also be an optional top() operation to get the value at the top of the stack It is considered an error to pop from a stack

that is empty

The Call Stack

At a low level, computers have a stack to implement function calls
This stack is typically limited to 4 MBytes
This means that practically recursion depth is now limited unless you explicitly request more.
Our data Structure stack can allocate from the heap, limited only by available RAM.

A queue is similar to a stack, an abstract data type with 3 basic operations
FIFO: First In, First Out
Queues are *fair*. Would you like to be on a list implementing LIFO?

- enqueue() (add to the front of the queue)
- dequeue() (remove from the end of the queue)
- isEmpty() (return true if the queue has no elements

It is considered an error to attempt to dequeue from an empty queue.

Implementing a Stack with LinkedList (head only)

Implementing a Stack with LinkedList (head and tail)