

## CSE 143

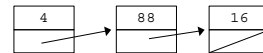
### Pointer-Based Linked Lists

[Chapter 4 p.157]

07/31/01 L-1

## Linked Lists

- A **linked list** is a collection of “nodes” containing data
- Each node points to the next node in the list.
- That's it!
- Example: a list of 3 integers:



07/31/01 L-2

## Metacomment

- Linked lists -- a Great Idea In Programming
  - Simple, natural
  - Flexible
    - Many variations are possible, once basic idea is mastered
- Linked lists are commonly implemented with dynamically allocated nodes
- But after all, this is C++.
  - So expect complications!

07/31/01 L-3

## Implementing Linked Lists

- Each node has two members: a **data item** and a **next link** field which points to the successor node.
- The "next link" field of one node points to the next node in the list.
- Use a “head” or “front” variable to point to first node
- Example: a list of 3 integers:



07/31/01 L-4

## What's the “data item”?

- Data is the same in every node of the list
  - Just like with arrays
- Could be ANY type: integer, double, Book, Bookshelf, Appointment, BankAccount, etc.
  - Most of our examples use int for simplicity

07/31/01 L-5

## Nodes for an int Linked List

- First we'll declare a struct which we'll use to represent a node:

```
struct Node {
    int item;
    Node* next;
};
```
- Now we can create new nodes:

```
Node* p;
p = new Node;
p->item = 100; // shorthand for: (*p).item = 100
p->next = NULL; // shorthand for: (*p).next = NULL
```
- Note the use of the -> operator
- Draw the picture!

07/31/01 L-6

## Manipulating Nodes

- Draw the picture that results from the following code:

```
Node* front;
Node* temp;

front = new Node;
front->item = 1;

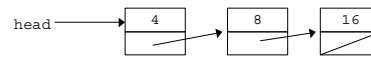
front->next = new Node;
front->next->item = 2;
front->next->next = NULL; // what did we just do?

temp = front; front = front->next;
delete temp; // what did we just do?
```

07/31/01 L-7

## Inserting a new link

Before:



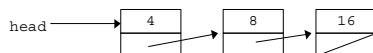
Insert "5" after 4.

After:

07/31/01 L-8

## Deleting a link

Before:



Delete "8"

After:

07/31/01 L-9

## Tips for Getting the Code Right

- Draw pictures
- These special cases often need slightly different code
  - Middle of the list
  - Beginning of the list
  - End of the list
  - Empty list
- Helper variables such as prev, curr
  - make sure they have the right values!
- Careful as usual with dynamic memory
- Fail-safe programming: asserts, etc.

07/31/01 L-10

## Printing a Linked List

```
// print data in list given pointer to first node
void print(Node* first) {
    Node *p; // *p is the current node in the list
    p = _____ ;

    while ( _____ ) {

    }

}
```

07/31/01 L-11

## Recursion and Linked Lists

- A linked list is a recursive data structure
- Recursive algorithms are natural with linked lists
  - But not always as efficient (space required for recursive function calls)

07/31/01 L-12

## Printing a Linked List

```
// print data in list given pointer to first node
void print(Node* first) {
    if (first == NULL)
        return;
    else {
        cout << endl << first->item;
        print(first->next);
    }
}
```

- How many recursive calls are needed?

07/31/01 L-13

## Printing in Reverse Order

- At first, seems difficult  
All the pointers point only forward.
- Recursion to the rescue!

```
void RPrint(Node* first) {
    if (first == NULL)
        return;
    else {
        RPrint(first->next);
        cout << endl << first->item;
    }
}
```

- Challenge: Try doing this without recursion

07/31/01 L-14

## Summing a List

```
int listSum(Node* list) {
    if (list == NULL)
        return 0; // empty list has sum == 0
    else
        return list->item + listSum(list->next);
}
```

- Common pattern for a list "traversal"
- How would you modify this to...
  - Count the length of a list?
  - Add N to each element of a list?
  - Determine if a particular value occurred in the list?

07/31/01 L-15

## Puzzler: List Remove

- Make new list (copy), same data as old, except: don't include nodes with a given data value in the new list

- The original list is to be unchanged!

```
Node* ListRemove(Node *first, int v);
```

- Draw a picture of an example first!
  - If you can't draw the picture, how can you hope to program it?

07/31/01 L-16

## Node\* ListRemove(Node \*first, int v)

```
{
    if (first == NULL)
        return NULL;
    else if (first->item != v) {
        //make a node for the new list, copy data
        Node* newNode = new Node;
        newNode->item = first->item;
        newNode->next = ListRemove(first->next, v);
        return newNode;
    }
    else
        return ListRemove(first->next, v);
}
```

07/31/01 L-17

## Another Approach

- Some people use a slightly different approach to implementation
  - 1. Have a permanent, dummy node as the header
  - 2. Point the last link of the chain back to the dummy (header) node
- All the code changes!
  - On balance, may be a little simpler; fewer special cases when inserting and deleting

07/31/01 L-18