Announcements

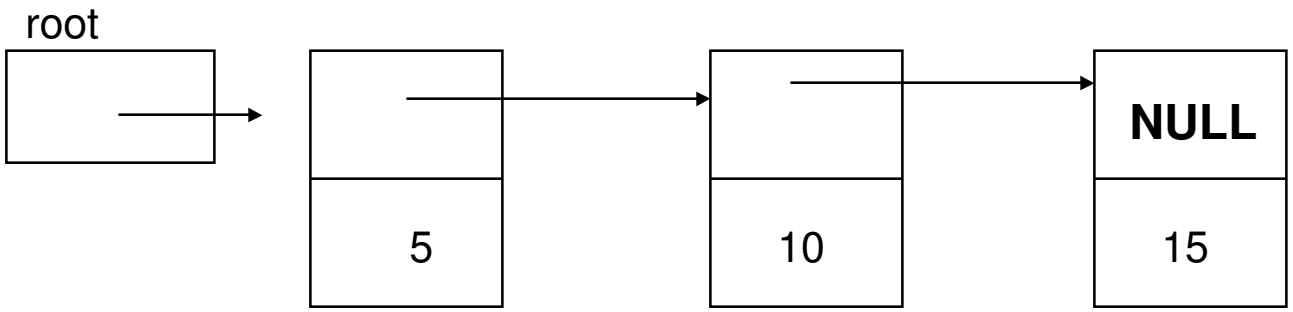
- Exam #1 and Project #1
  - if you have not picked something up, see me after class
- Exam #1 Re-grade Requests
  - due by end of class today (both electronic and paper)
- Program #2
  - Due on March 15 at 8:00 pm (Tuesday)
  - Don't free NULL pointers – will be flagged as an error and cause you to fail tests
  - createTable – confusion about interpretation
    - passing in a "size" just needs to be in range
    - does not need to be exactly one of the primes in the list
- Reading
  - Chapter 12, 10.2 (Today)
  - Notes (Tuesday)

Linked Data Structures

- Like classes with refs to same class in Java
- Have pointers to a struct of the same time
- Example Declaration:

```
typedef struct NODE {
        struct NODE *next;
        int value;
} Node;
```

- Next is a pointer to another Node

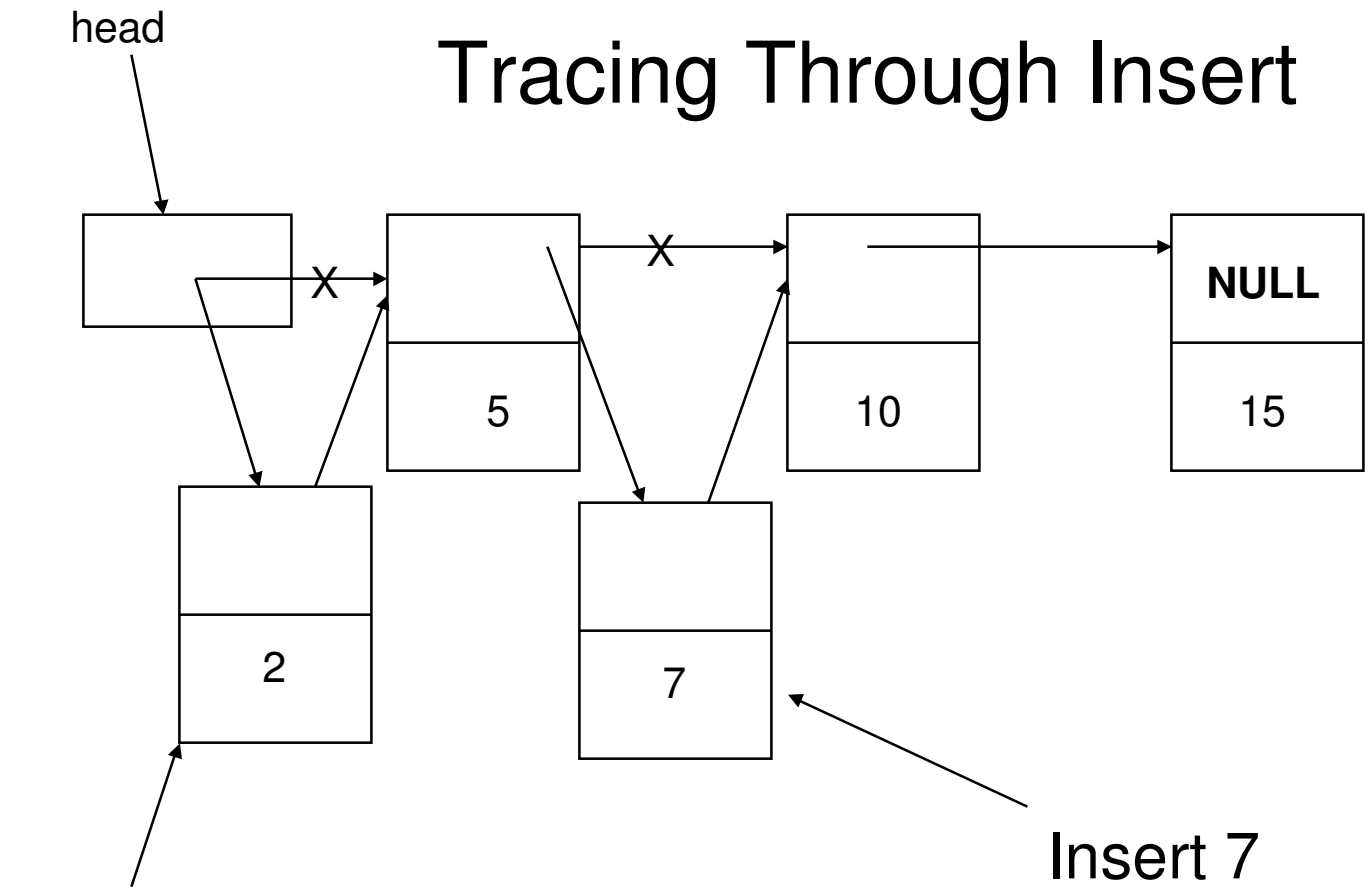


Finding a value in a Linked Structure

```
Node  *find(Node *current, int val) {
        while (current && current->value != val)  {
            current = current->next;
        }
        return current;
}
```

```
int IsIn(Node *current, int val) {
        while (current && current->value != val)  {
            current = current->next;
        }
        if (current) {
            return 1;
        } else {
            return  0;
        }
}
```

Tracing Through Insert

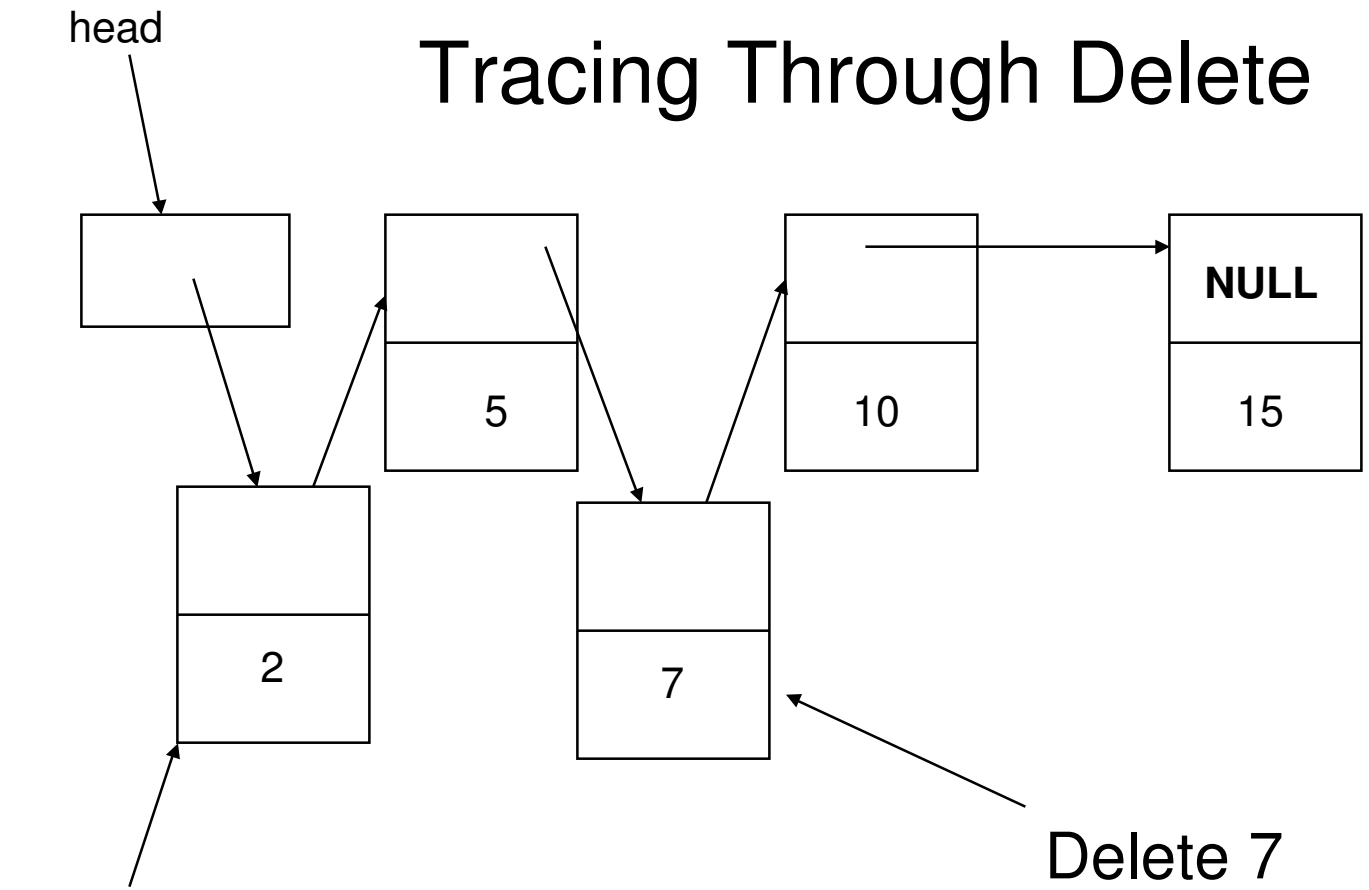


Inserting Into a Linked List

```
int insert(Node **head, int new_value) {
        Node *current = *head;
        Node *pred = NULL;
        Node *newItem;
        while (current && current->value < new_value)  {
            pred = current;
            current = current->next;
        }
        newItem = (Node *) malloc(sizeof(Node));
        if (!newItem) return -1;
        newItem->value = new_value;
        newItem->next = current;

        if (!pred) {
            *head = newItem;
        } else {
            pred->next = newItem;
        }
        return 0;
}
```

Tracing Through Delete



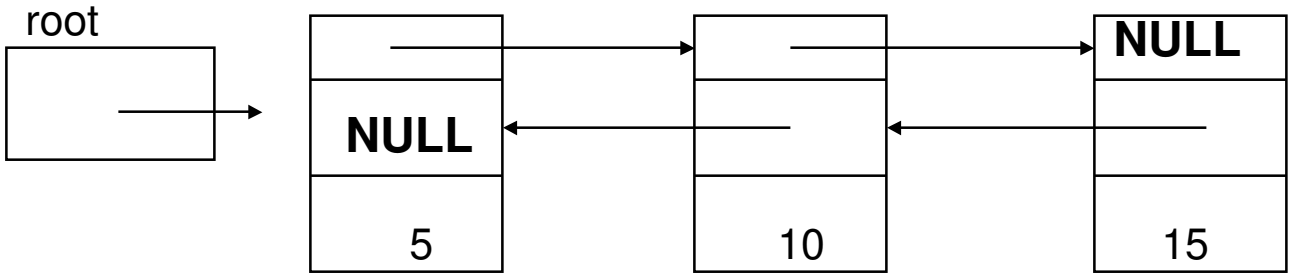
Deleting From A Single Linked List

```
int delete(Node **head, int new_value) {
        Node *pred = NULL;
        Node *current = *head;
        while (current && (current->value != new_value))  {
            pred = current;
            current = current->next;
        }
        if (!current) {
            return -1;        /* not found */
        }
        if (pred) {
            pred->next = current->next;
        } else {
            *head = current->next;  /* deleted first item */
        }
        free (current);
        return 0;
}
```

Doubled Linked Lists

- Each nodes
  - contains a value
  - a pointer the next **and** previous element
- Typical Declaration:

```
typedef struct NODE {
    struct NODE *next;
    struct NODE *prev;
    int value;
} Node;
```



Insert into a doubly Linked list

- Think about 4 cases:
  - **Middle of the list**
    - **Update previous element's next pointer**
      - **new node's previous pointer set to this node**
    - **Update next element's previous pointer**
      - **new node's next point set to that node**
  - **End of the list**
    - **Update previous element's next pointer**
      - **new node's previous pointer set to this node**
      - **new node's next pointer set to NULL**
  - **Start of the list**
    - **Update head of list**
    - **Update old head of list's previous element**
      - **new node's next pointer set to this node**
      - **new node's previous pointer set to NULL**
  - **An initially empty list**
    - **Update head of the list**
      - **new node's next and previous pointer both NULL**

Inserting Into A Doubly Linked List

```
int insert(Node **head, int new_value) {
        Node *pred = NULL, *newItem;
        Node *current = *head;
        while (current && current->value < new_value)  {
              pred = current;
              current = current->next;
        }
        newItem = (Node *) malloc(sizeof(Node));
        if (!newItem) return -1;
        newItem->value = new_value;
        newItem->next = current;
        newItem->prev = pred;
        if (!pred) {
              *head = newItem;
        } else {
               pred->next = newItem;
        }
        if (current) {
           current->prev = next;
        }
}
```

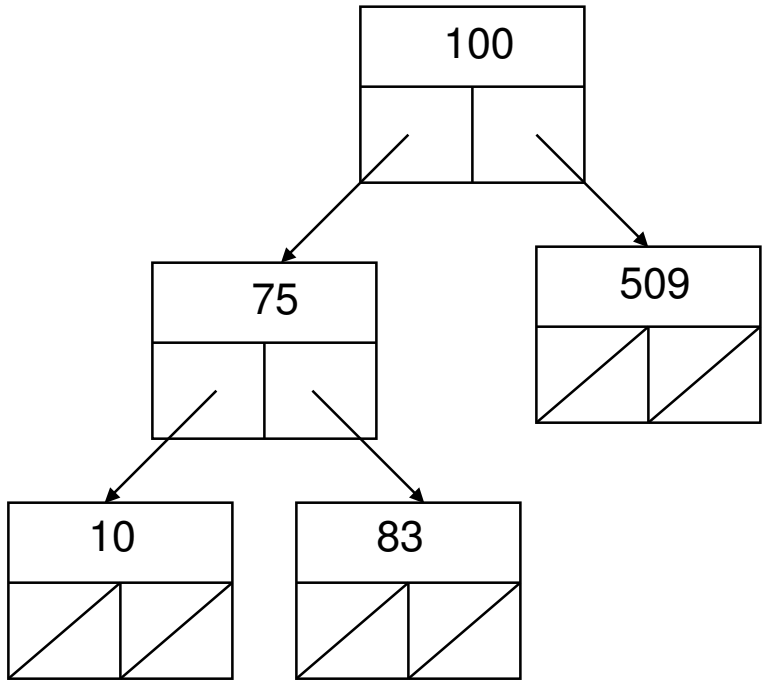
Deleting from a Doubly Linked List

```
int delete(Node **head, int new_value) {
        Node *pred = NULL;
        Node *current = *head;
        while (current && (current->value != new_value))  {
            pred = current;
            current = current->next;
        }
        if (!current)     return -1;   /* not found */
        if (pred) {
            pred->next = current->next;
        } else {
            *head = current->next; /* deleted first item */
        }
        if (current->next) {
            current->next->prev = pred;
        }
        free (current);
        return 0;
}
```

Binary Trees

- Each node
  - contains a value
  - a pointer a left child and a right child
- Typical Declaration:

```
typedef struct NODE {
    struct NODE *left;
    struct NODE *right;
    int value;
} Node;
```



Lookup In A Binary Search Tree

- If the element is less than the current node
  - Look in the left child tree
- If the element is greater than current node
  - Look in the right child tree
- Example (assumes values >= 0):

```
int Lookup(Node *root, int value) {
    if (!root) return -1;
    if (root->value == value) {
        return 1;
    } else if (root->value > value) {
        return Lookup(root->left, value);
    } else {
        return Lookup(root->right, value);
    }
}
```

Insert Into a Binary SearchTree

```
int insert(Node **root, int value) {
      Node *new;
      if (!(*root)) {
         new = (Node *) malloc(sizefof(Node));
         if (!new) return -1;
         *root = new;
         new->left = new->right = NULL;
         new->value = value;
         return 0;
      } if ((*root)->value > value) {
            return insert(&((*root)->left), value);
      } else {
            return insert(&((*root)->right), value);
      }
}
```

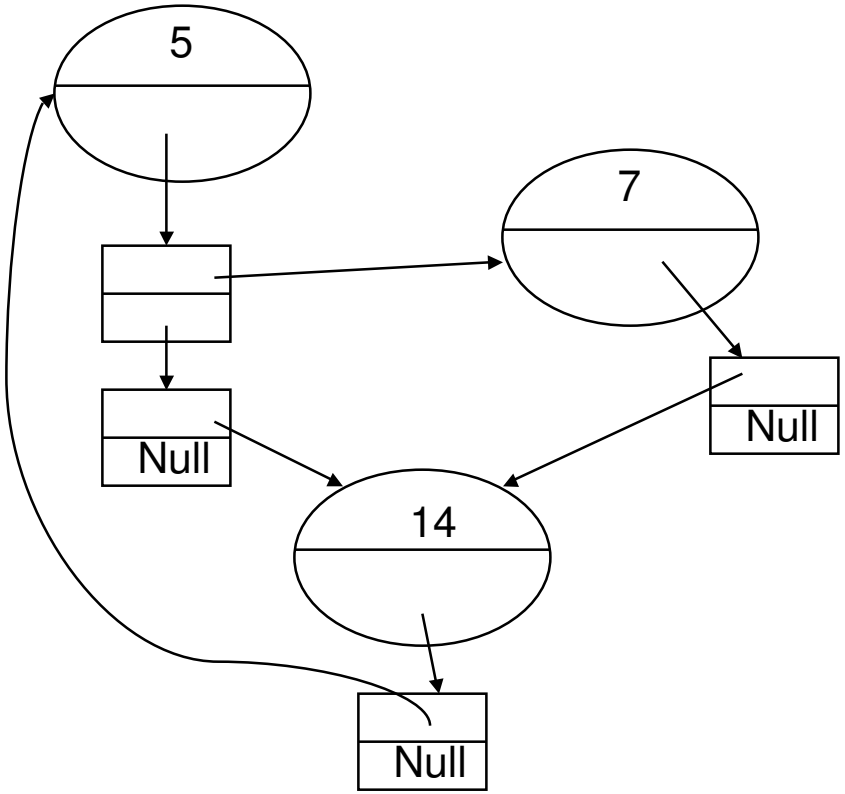
Graphs - Directed

- In a Graph
  - there can be an arbitrary number of nodes
  - each node can have an arbitrary number of out arcs
- Declarations

```
typedef struct ARC {
    struct NODE *nodePtr;
    struct ARC *next;
} Arc;

typedef struct NODE {
    Arc *outArcs;
    int value;
} Node;
```

Graphs



key

Node

Arc

Adding Node To a Graph

```
Node *AddNode(int value) {
   Node *new;
   new = (Node *) malloc(sizeof(Node));
   new->value = value;
   new->outArcs = NULL;
   return new;
}
```

You need to store this node into some type of structure so you don't lose it

- linked list of nodes
- array of nodes
- tree of nodes
- etc.

Adding Arcs To a Graph

```
int AddArc(Node *from, Node *to) {
      Arc *new, *pred, *current;
      pred = NULL;
      for (current=from->outArcs; current; current=current->next) {
         if (current->nodePtr == to) return 1;  /* already defined this arc */
         pred = current;
      }
      new = (Arc *) malloc(sizeof(Arc));
      if (!new) return -1;
      new->nodePtr = to;
      new->next = NULL;
      if (!pred)  {
           from->outArcs = new;  /* first out arc for this node */
      } else {
          pred->next = new;
      }
}
```