EECS 211, Fall 2014

Linked lists

William Hendrix

Lecture 24

Today

- Additional operators
- Linked lists
- Doubly-linked lists
- Pros/cons
- Exercise

Other bad programming practices

- Overuse of “shortcut” operations can generate overly confusing code
- Pre-increment vs. post-increment
  - Increment (++/--) operator can appear before or after variable
  - Pre-increment (++x) is executed before the rest of the statement
  - Post-increment (x++) is executed afterwards
- **Example**

```
cout << i++;
cout << ++i;
while (str[i++] != ‘\0’);
```

- Conditional operator (?:)
  - a.k.a. the “ternary operator”
  - Cannot be overloaded
- Syntax

```
(condition)? value_if_true : value_if_false
```

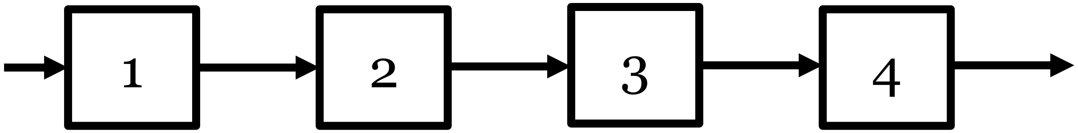
- Evaluates to a different value depending on condition
- Shorthand for `if` statement

Data structures

- Underlying representation of data
- Choice of data structure can strongly influence performance
- Different data structures have different advantages/disadvantages
- Vectors: array-based data structure
  - Accessing elements is fast: *O(1)*
  - Appending elements is not bad: *O(1)* on average
    - Inserting in the middle is not so good: *O(n)*
  - Searching is pretty good: *O(n)* unsorted, *O(ln(n))* sorted
- 2D data structures
  - 1D dynamic array/2D static arrays
    - Faster, fewer memory allocations
  - 2D dynamic array
    - Can store non-rectangular (ragged) arrays
  - All 3 are array-based
    - Performance similar to vectors

Linked lists

- Alternative to arrays/vectors for storing sequence of data
- **Main idea**
  - Instead of storing values in a contiguous block, store individually
  - Each value points to the next
  - List is terminated with a `NULL` pointer
- **Picture**



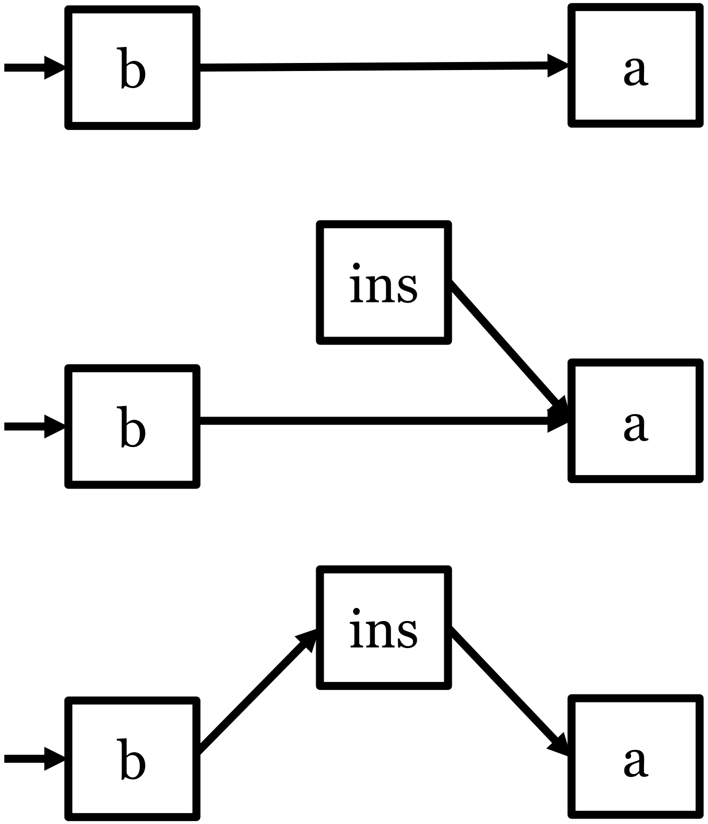
- **Implementation**
  - `Node` class stores an individual value and a `Node*`
  - `LinkedList` class stores pointer to head of list
    - Handles operations on `Nodes`
    - May optionally store list size and tail pointer to avoid traversal

Linked list operations

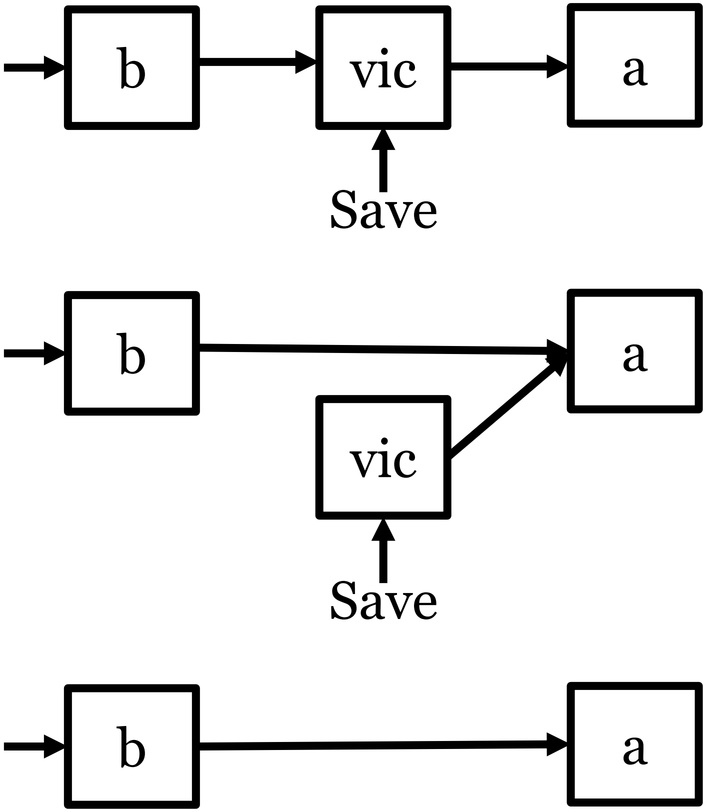
- Access element *n*
  1. Start at head
  2. Follow *n* pointers
- Append
  1. Construct new `Node`
  2. Set tail pointer to constructed `Node`
- Insert
  1. Construct `Node ins`
  2. Advance to `Node b` before insertion point
  3. Set `ins.next` to `b.next`
  4. Point `b.next` to `ins`
- Delete
  1. Move to `Node b` before `victim` node
  2. Save pointer to `victim`
  3. Point b to `victim.next`
  4. Delete `victim`
- *Special case*:
  – Need to update `head` pointer if deleting last `Node` or adding to empty list

Insertion and deletion

Insertion



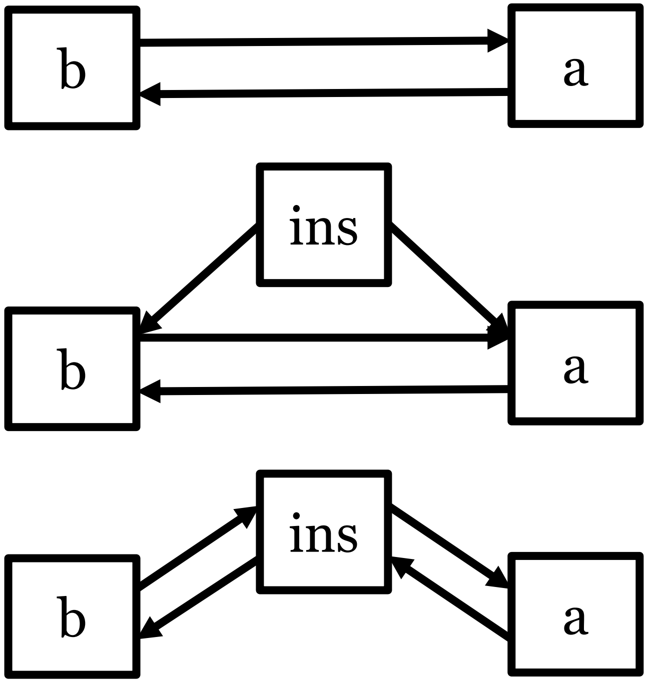
Deletion



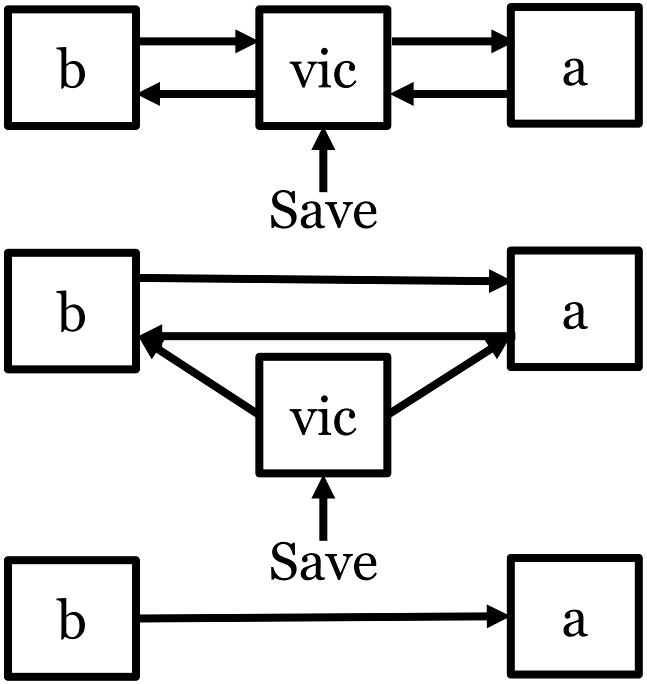
Doubly-linked lists

- By adding previous (`prev`) pointer to `Node`, we can scan backwards
  - Faster access to elements near `tail`
- Don't need to "restart" scan if we pass an element
  - E.g., finding the previous `Node` in order to delete this one
- Insertion and deletion need to update `prev` pointer as well as `next`

Insert



Delete



Comparison to arrays/vectors

- Use more space (1-2 pointers per element)
- *Much* slower to access an element
  - Doubly-linked lists are faster, but still *O(n)*
- Not as bad if scanning the entire list or array
  - However, a page of memory not likely to contain several `Nodes`
    - Lack of "cache coherence" relative to arrays
- Much slower to search a sorted list
  - No way to jump to midpoint, so no binary search
- Appending or deleting from end is comparable
  - Linked lists make more calls to `new`
  - Vectors make bigger, less frequent allocations but copy data
- Much faster to add/remove elements to beginning or middle
  - Array requires copying
- Concatenation *much* faster with linked list
- **Bottom line:** arrays usually faster, lists offer more flexibility for expansion

Tonight

- **Lab 5** due Monday

- **Recommended reading:** Sections 12.1-3