CSE 143

Vector ADT as Linked List

[Chapter 4 p.170]

Recall Vector ADT

- Earlier we defined a Vector ADT
  - Data encapsulated inside the class
  - Operations available only through the public interface
- More than one implementation is possible
  - Array-based
    - + Easy to implement
    - +/- Efficiency?
    - - Must know size
    - - Can't change size
    - - Run out of space or waste it; rarely "just right"
  - Alternative: Reimplement based on Linked lists
    - + Size varies dynamically
    - - a little more work to implement
    - +/- Efficiency?

Vector ADT

```
class Vector {
public:
   Vector ( );              // construct empty vector
   bool isEmpty( );         // = "this Vector is empty"
   int    length ( );       // = # of items in this Vector
                            // Insert newItem in this Vector at newPosition
   void Insert (int newPosition, Item newItem);
                            // Delete item at position and return a copy of it
   Item Delete (int position);
                            // Return a copy of the item at position
   Item Retrieve (int position);
   ...
private:
   ...
   };
```


Internal Data

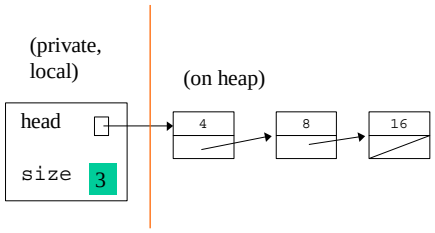
- Declare a struct to represent a node:

```
struct Node {
   int data;
   Node *next;
};

class Vector {
   public:
      ...
   private:
      int  size;       //number of items in the Vector
      Node *head;  //ptr to linked list of items
...};
```

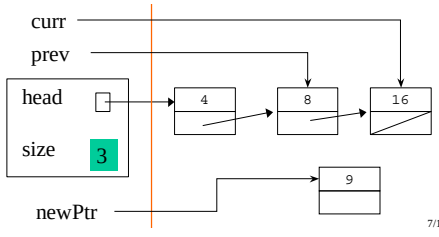

Portrait of a Vector

- Now a Vector variable might look like this:



Inserting 9 at position X

- We have to find node X
  - Better yet, get a pointer to X (curr) and X-1(prev)
- Example: X = 2



Finding Position X

- Write a function "PtrTo" to traverse the list, return a pointer to the Xth element (code: p.175)
- Should be a member function

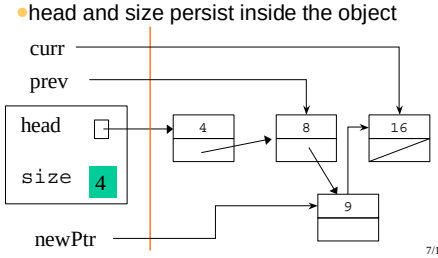
  *listNode * PtrTo (int X) const;*
  - Style point: not part of the interface, so should be private
- Special cases: X outside the range of the list
  - return NULL
- Given this, curr and prev are easy to get:
  - curr = PtrTo(X); prev = PtrTo (X-1)
- Better yet:
  - prev = PtrTo(X-1); curr = prev->next;

Relinking for Insert (p.176)

- Given prev and curr (via PtrTo function):
  - *newPtr->next = curr;*
  - *prev->next = newPtr;*
- Inserting at beginning is a special case (X=0)
  - *newPtr ->next = head;*
  - *head = newPtr;*
- What about inserting at end of list?
  - How to recognize?
  - Is the code special?

Final Picture

- curr, prev, and newPtr are local variables that go away
- head and size persist inside the object



One More Problem

- When a function is exited, only the local memory is released
  - If a local variable points to dynamic memory, that dynamic memory stays allocated
  - results in a memory leak

    `unless there is another pointer to the data`
- We will see the solution later: a "destructor" function called automatically

Variations on a theme

- Lists are useful and common data structures
- Many variations are possible, for example:
  - Doubly linked lists

    `Point backwards as well as forwards`
    `Makes finding the previous pointer a breeze`
    `Takes a little more space and complexity to manage the extra pointers`
  - Circular lists

    `last node points back to first node, instead of to NULL`
- Many implementation tricks are possible, for example:
  - Head and tail pointers.

    `Good for "queues" (always add at tail, always remove at head)`
  - Dummy nodes at front or rear

    `Can remove some special cases`