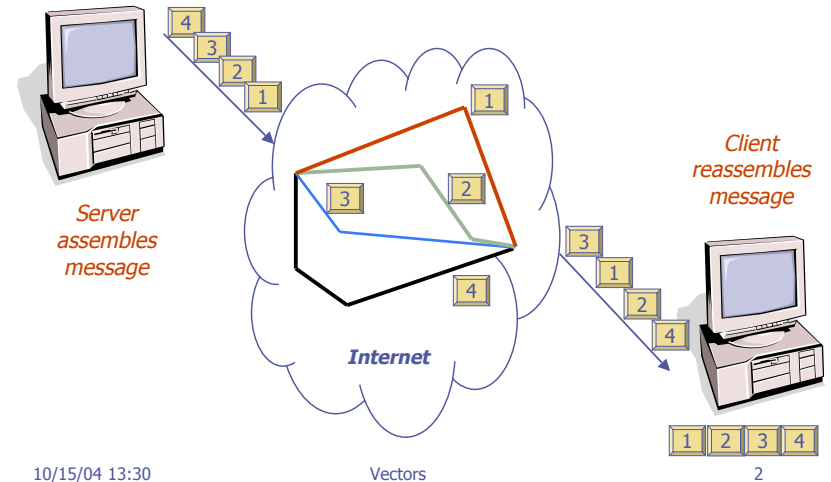


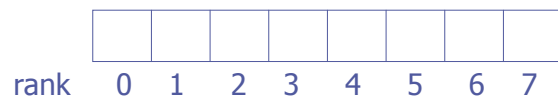
Vectors, Lists, and Sequences

Sequences of Items



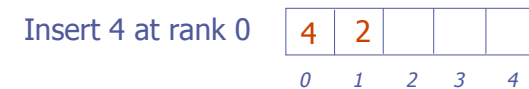
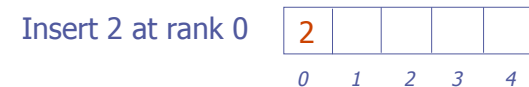
The Vector ADT

- ◆ The **Vector** ADT extends the notion of array by storing a sequence of arbitrary objects
- ◆ An element can be accessed, inserted or removed by specifying its rank (number of elements preceding it)
- ◆ An exception is thrown if an incorrect rank is specified (e.g., a negative rank)



Rank Operations

- ◆ Operations based on rank



Rank Operations

◆ Operations based on rank

Access element at rank 1

4	9	2		
0	1	2	3	4

Remove element at rank 1

4	2			
0	1	2	3	4

Access element at rank 4
ERROR

4	2			
0	1	2	3	4

The Vector ADT

◆ Main vector operations:

- **elemAtRank**(int r): returns the element at rank r without removing it
- **replaceAtRank**(int r, Object o): replace the element at rank r with o
- **insertAtRank**(int r, Object o): insert a new element o to have rank r
- **removeAtRank**(int r): removes the element at rank r

◆ Additional operations **size()** and **isEmpty()**

Array-based Vector

- ◆ Use an array V of size N
- ◆ A variable n keeps track of the size of the vector (number of elements stored)
- ◆ Operation **elemAtRank**(r) is implemented in $O(1)$ time by returning $V[r]$



Performance

- ◆ In the array based implementation of a Vector
 - The space used by the data structure is $O(N)$
 - **size**, **isEmpty**, **elemAtRank** and **replaceAtRank** run in $O(1)$ time
 - **insertAtRank** and **removeAtRank** run in $O(n)$ time
- ◆ In an **insertAtRank** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one

Extendable Array

- ◆ Allocate new array of larger size
- ◆ Copy old array to new array
- ◆ Deallocate old array and use new array

10/15/04 13:30

Vectors

9

Extendable Array

```
#define SIZE 10
class Vector {
private:
    int* a;
    int size;
public:
    Vector ( ) {
        a = new int [SIZE];
        size = SIZE;
        bzero ( a, size * sizeof(int) );
    };
};
```

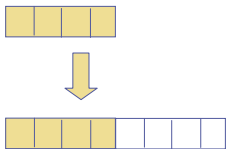
10/15/04 13:30

Vectors

10

Extendable Array

- ◆ In class exercise - write the extend function for the Vector class
 - Double array size with each extend



10/15/04 13:30

Vectors

11

Lists

- ◆ Extends a linked list to store items based on a position in the list
- ◆ **Position** denotes object location in a list.

10/15/04 13:30

Vectors

12

Lists - Node based operations

- ◆ Use node as a parameter to insert and remove in $O(1)$ time for doubly linked list
 - `removeAtNode (Node* v)`
 - `insertAfterNode (Node* afterMe, Node* newNode)`
- ◆ First call a search function to find the node to use as the parameter to these functions
- ◆ Not good - user could modify internal structure of list such as next or prev

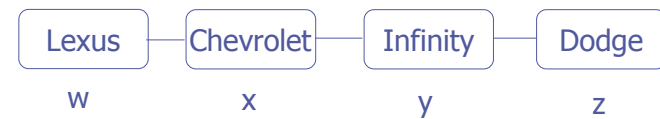
10/15/04 13:30

Vectors

13

Positions

- ◆ Positions are defined relatively (in terms of neighbors)
 - Position y is after position x and before position z



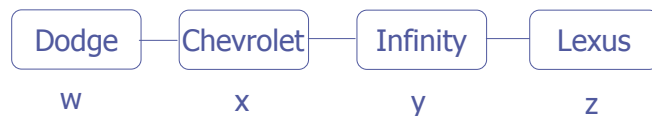
10/15/04 13:30

Vectors

14

Positions

- ◆ A position never changes even if a rank changes unless the element at a position is removed thus destroying the position
- ◆ Position don't change even if we swap elements at positions



10/15/04 13:30

Vectors

15

List ADT

- ◆ Operations based on position
 - `p first ()`
 - `p last ()`
 - `isFirst (p)`
 - `isLast (p)`
 - `p before (p)`
 - `p after (p)`

10/15/04 13:30

Vectors

16

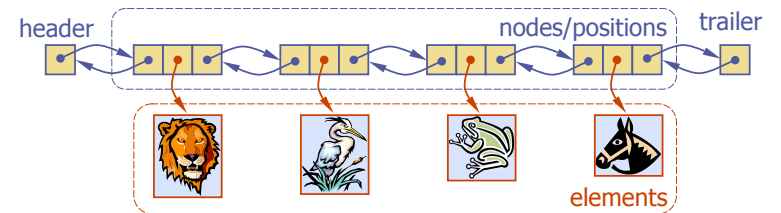
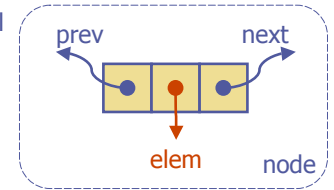
List ADT

◆ Update functions

- replaceElement (p, e)
- swapElements (p, q)
- insertFirst (e)
- insertLast (e)
- insertBefore (p,e)
- insertAfter (p, e)
- remove (p)

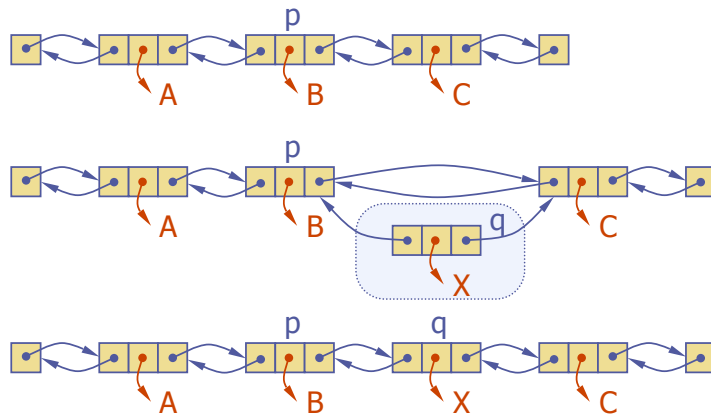
Doubly Linked List

- ◆ A doubly linked list provides a natural implementation of the List ADT
- ◆ Nodes implement Position and store:
 - element
 - link to the previous node
 - link to the next node



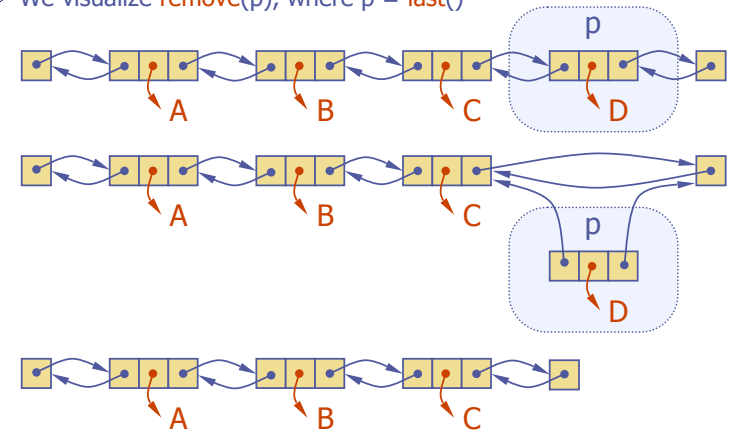
Insertion

- ◆ We visualize operation `insertAfter(p, X)`, which returns position `q`

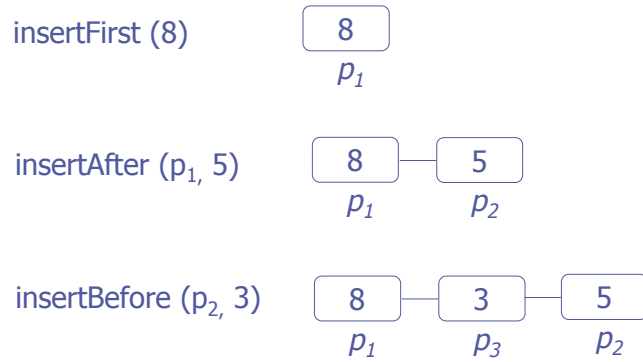


Deletion

- ◆ We visualize `remove(p)`, where `p = last()`



Operations

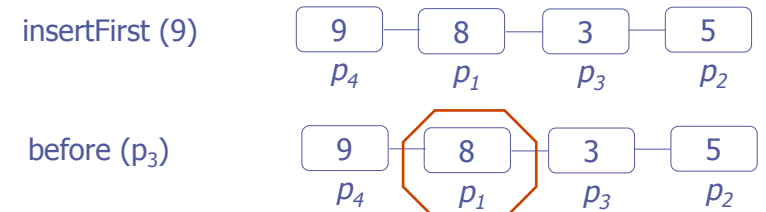


10/15/04 13:30

Vectors

21

Operations

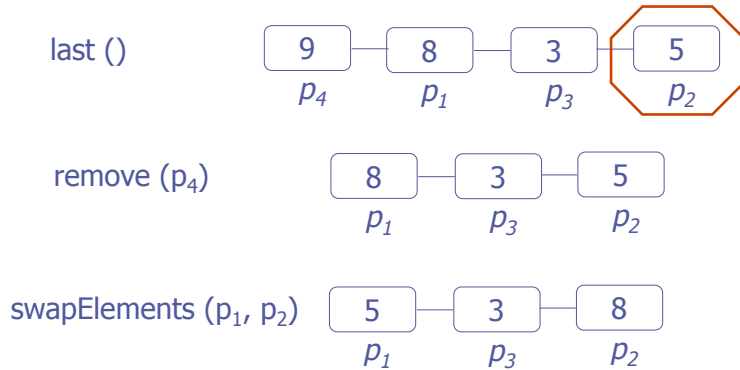


10/15/04 13:30

Vectors

22

Operations

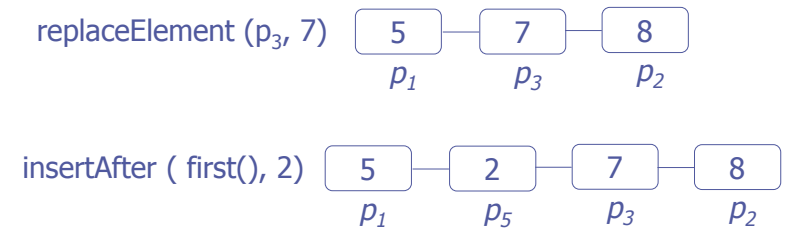


10/15/04 13:30

Vectors

23

Operations



10/15/04 13:30

Vectors

24

List ADT

- ◆ List application
 - Cursor in an editor
 - ◆ Updates are relative to cursor
- ◆ Position is generic
 - Array based list or linked list based list?
 - Position representation will be different

10/15/04 13:30

Vectors

25

Sequence ADT

- ◆ The **Sequence** ADT is the union of the Vector and List ADTs
- ◆ Elements accessed by
 - Rank, or
 - Position
- ◆ Generic methods:
 - `size()`, `isEmpty()`
- ◆ Vector-based methods:
 - `elemAtRank(r)`, `replaceAtRank(r, o)`, `insertAtRank(r, o)`, `removeAtRank(r)`
- ◆ List-based methods:
 - `first()`, `last()`, `before(p)`, `after(p)`, `replaceElement(p, o)`, `swapElements(p, q)`, `insertBefore(p, o)`, `insertAfter(p, o)`, `insertFirst(o)`, `insertLast(o)`, `remove(p)`
- ◆ Bridge methods:
 - `atRank(r)`, `rankOf(p)`

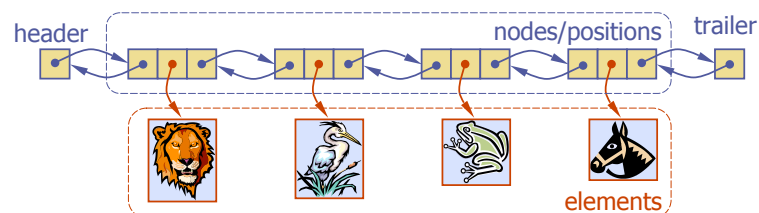
10/15/04 13:30

Vectors

26

Sequence with a Doubly Linked List

- ◆ Any rank operation would have to traverse the list



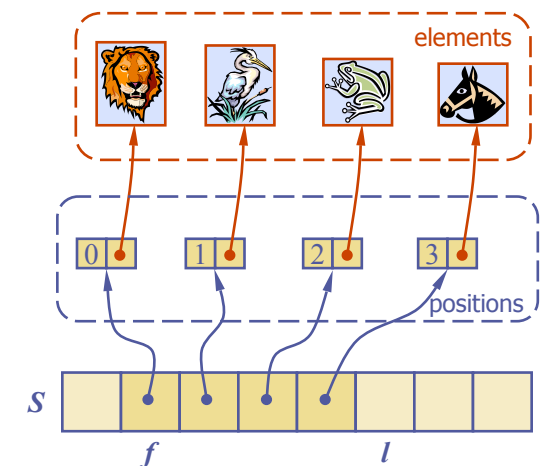
10/15/04 13:30

Vectors

27

Sequence With an Array

- ◆ We use a circular array storing positions
- ◆ A position object stores:
 - Element
 - Rank
- ◆ Indices f and l keep track of first and last positions



10/15/04 13:30

Vectors

28

Sequence Implementations

Operation	Array	List
size, isEmpty		
atRank, rankOf, elemAtRank		
first, last, before, after		
replaceElement, swapElements		
replaceAtRank		
insertAtRank, removeAtRank		
insertFirst, insertLast		
insertAfter, insertBefore		
remove		