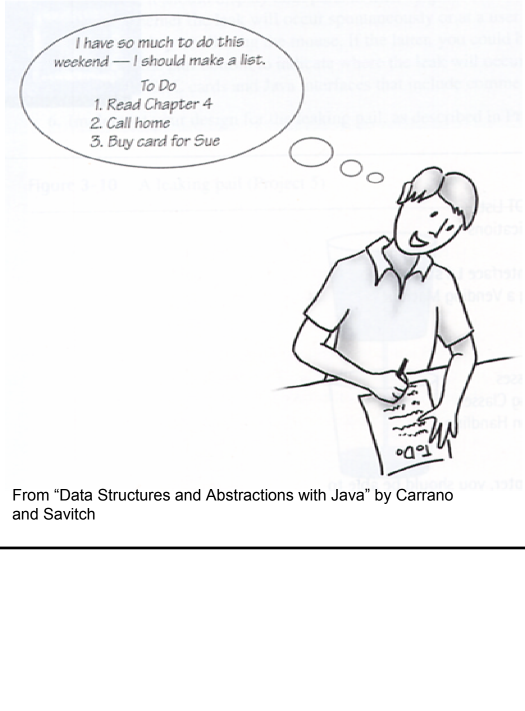
Lists

- Lists as an abstract data type (ADT)
- Different list implementations and the tradeoffs of each approach

What Is A List?

- A method of organizing data

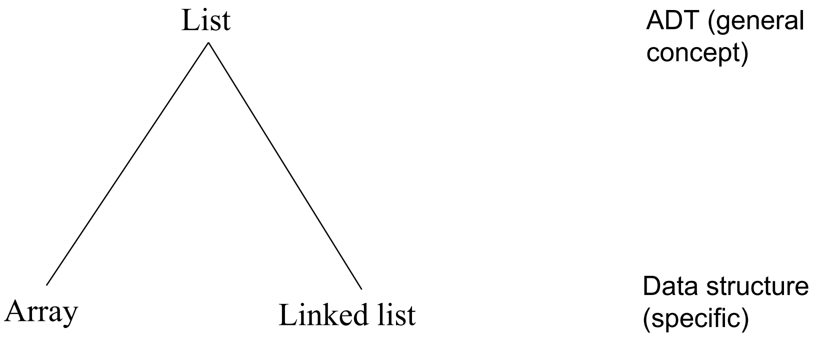


From “Data Structures and Abstractions with Java” by Carrano and Savitch

Common List Operations

- •Adding new elements
  - Ordered by time
  - Ascending/descending order
  - Ordered by frequency
- •Removing an element/elements
- •Replace an element with a new value
- •Searching the list for an element
- •Counting the elements in the list
- •Checking if the list is full or empty
- •Display all elements

List Implementations



List

Array

Linked list

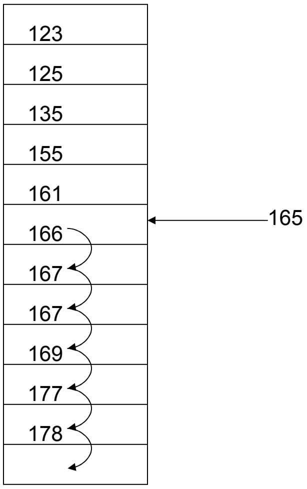
ADT (general concept)

Data structure (specific)

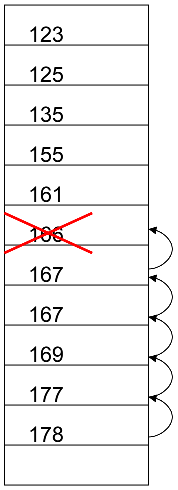
Lists Implemented As Arrays

- Advantages
  - Simple to use (often a built-in type)
  - Retrievals are quick if the index is known (*O*(n1))
- Disadvantages
  - Adding/removing elements may be awkward
  - Fixed size arrays either limits the size of the list or wastes space
  - Dynamic sized arrays requires copying

Arrays: Adding Elements In The Middle



Arrays: Deleting Elements From The Middle



123

125

135

155

161

~~166~~

167

167

169

177

178

Arrays: Dynamic Sized Arrays

```
int [] arr = new int[4];


 :        :          :

int [] temp = arr;

int [] arr = new int[8];


// Copy from temp to arr is needed
```

Lists Implemented As Linked Lists

- Types Of Linked Lists
  1. Singly linked
  2. Circular
  3. Doubly linked

Singly Linked List

Example:

The full example can be found in the directory:
/home/331/tamj/examples/lists/singlyLinked

```
class ListManager
{
   private Node head;
   private int length;
   private int currentDataValue = 10;
   private static final int MAX_DATA = 100;
               :                    :
}
```

List Operations: Arrays Vs. Singly Linked Lists

Operation	Array	Singly Linked List
Initialization	*O* (n)	*O* (1)

Examples Of List Initializations

- Array

```
for (i = 0; i < list.length; i++)
   list[i] = -1;
```

- Linked list

```
public ListManager ()
{
   head = null;
   length = 0;
}
public ListManager (Node newHead)
{
   head = newHead;
   length = 1;
}
```

List Operations: Arrays Vs. Singly Linked Lists

Operation	Array	Singly Linked List
Search	*O* (n) Sequential *O* ($\log_2$n) Binary	*O* (n)

Example Of A Linked List Search

```
public int search (int key)
{
   Node temp = head;
   boolean isFound = false;
   int index = 1;
```

Example Of A Linked List Search (2)

```
    while ((temp != null) && (isFound == false))
    {
        if (temp.data == key)
        {
            isFound = true;
        }
        else
        {
            temp = temp.next;
            index++;
        }
    }
```

Example Of A Linked List Search (3)

```
    if (isFound == true)
        return index;
    else
        return -1;
}
```

List Operations: Arrays Vs. Singly Linked Lists

Operation	Array	Singly Linked List
Insertion	*O* (1) No shifting *O* (n) Shifting	*O* (n)

Example Of A Linked List Insertion

```
public void addToEnd ()
{
   Node anotherNode = new Node (currentDataValue);
   currentDataValue += 10;
   Node temp;

   if (isEmpty() == true)
   {
      head = anotherNode;
      length++;
   }
```

Example Of A Linked List Insertion (2)

```
    else
    {
        temp = head;
        while (temp.next != null)
        {
            temp = temp.next;
        }
        temp.next = anotherNode;
        length++;
    }
}
```

Another Example Of A Linked List Insertion

```
public void addToPosition (int position)
{
    Node anotherNode = new Node (currentDataValue);
    Node temp;
    Node current;
    int index;

    if ((position < 1) || (position > (length+1)))
    {
        System.out.println("Position must be a value between 1-" +
             (length+1));
    }
```

Another Example Of A Linked List Insertion (2)

```
else
{
    if (isEmpty() == true)
    {
        if (position == 1)
        {
            length++;
            head = anotherNode;
        }
        else
            System.out.println("List empty");
    }
    else if (position == 1)
    {
        anotherNode.next = head;
        head = anotherNode;
```

Another Example Of A Linked List Insertion (3)

```
        else
        {
            current = head;
            index = 1;
            while (index < (position-1))
            {
                current = current.next;
                index++;
            }
            anotherNode.next = current.next;
            current.next = anotherNode;
            length++;
        }
    }
}
```

List Operations: Arrays Vs. Singly Linked Lists

Operation	Array		Singly Linked List
Deletion	*O* (1) *O* (n)	No shifting Shifting	*O* (n)

An Example Of A Linked List Deletion

```
public void delete (int key)
{
   int indexToDelete;
   int indexTemp;
   Node previous;
   Node toBeDeleted;

   indexToDelete = search(key);
   if (indexToDelete == -1)
   {
      System.out.println("Cannot delete element because it was not found in
        the list.");
   }
```

An Example Of A Linked List Deletion (2)

```
    else
    {
       if (indexToDelete == 1)
       {
          head = head.next;
          length--;
       }
```

An Example Of A Linked List Deletion (3)

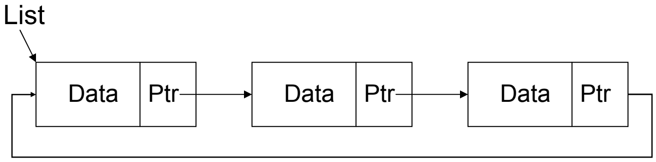
```
       else
       {
          previous = null;
          toBeDeleted = head;
          indexTemp = 1;
          while (indexTemp < indexToDelete)
          {
             previous = toBeDeleted;
             toBeDeleted = toBeDeleted.next;
             indexTemp++;
          }
          previous.next = toBeDeleted.next;
          length--;
       }
    }
}
```

Recursively Processing A List

```
public void displayReverse ()
{
   Node temp = head;
   System.out.println("Displaying list in reverse order");
   if (isEmpty() == false)
       reverse(temp);
   else
       System.out.println("Nothing to display, list is empty");
}
private void reverse (Node temp)
{
   if (temp.next != null)
       reverse(temp.next);
   System.out.println(temp.data);
}
```

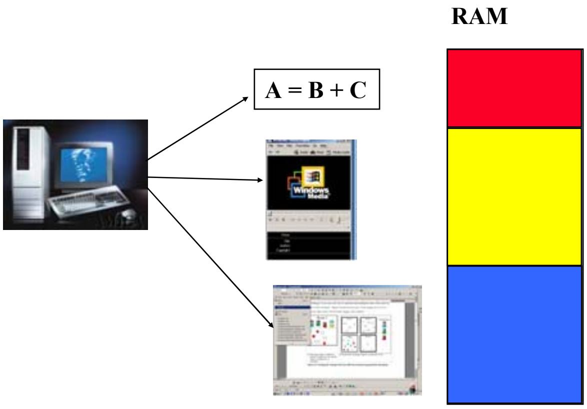
Circular Linked Lists

- An extra link from the end of the list to the front forms the list into a ring



Uses Of A Circular List

- e.g., Memory management by operating systems

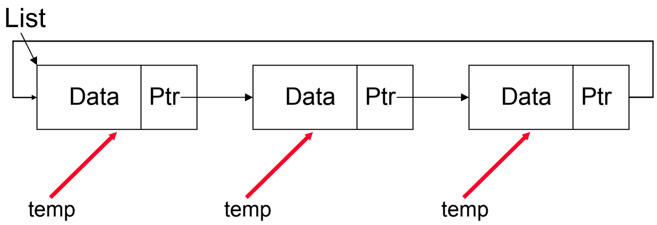


RAM

A = B + C

Searches With A Circular Linked Lists

- Cannot use a null reference as the signal that the end of the list has been reached.
- Must use the list reference as a point reference (stopping point) instead



List

Data Ptr

Data Ptr

Data Ptr

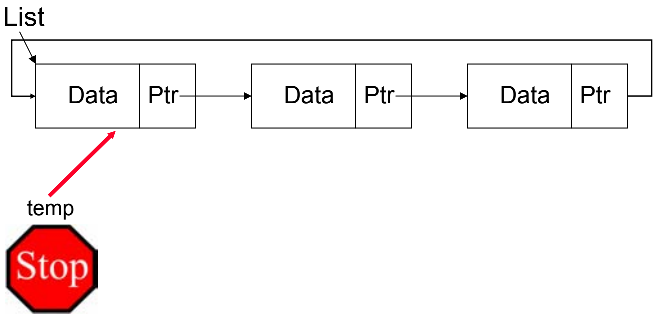
temp

temp

temp

Traversing A Circular Linked List

- Cannot use a null reference as the signal that the end of the list has been reached.
- Must use the list reference as a point reference (stopping point) instead



An Example Of Traversing A Circular Linked List

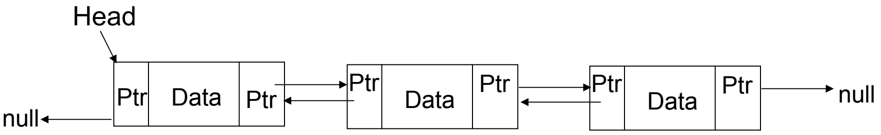
```
public void display ()
{
    Node temp = list;
    System.out.println("Displaying list");
    if (isEmpty() == true)
    {
        System.out.println("Nothing to display, list
          is empty.");
    }
    do
    {
        System.out.println(temp.data);
        temp = temp.next;
    } while (temp != list);
    System.out.println();
}
```

Worse Case Times For Circular Linked Lists

Operation	Time
Search	*O*(n)
Addition	*O*(n)
Deletion	*O*(n)

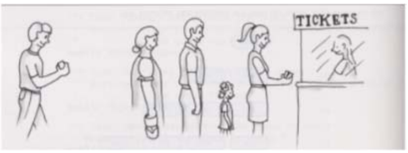
Doubly Linked Lists

- Each node has a reference or pointer back to the previous nodes



Pros Of Doubly Linked Lists

- Pros
  - Traversing the list in reverse order is now possible.
  - You can traverse a list without a trailing reference (or by scanning ahead)
  - It's more efficient for lists that require frequent additions and deletions near the front and back



From "Data Structures and Abstractions with Java" by Carrano and Savitch

Cons Of Doubly Linked Lists

- Cons
  - An extra reference is needed
  - Additions and deletions are more complex (especially near the front and end of the list)

Doubly Linked List

Example:

The full example can be found in the directory:
/home/331/tamj/examples/lists/doublyLinked

```
class ListManager
{
   private Node head;
   private int length;
   private int currentDataValue = 10;
   private static final int MAX_DATA = 100;
              :                  :                     :                     :
}
```

Doubly Linked List: Adding To The End

```
public void addToEnd ()
{
   Node anotherNode = new Node (currentDataValue);
   Node temp;

   if (isEmpty() == true)
      head = anotherNode;
```

}

Doubly Linked List: Adding To The End (2)

```
    else
    {
       temp = head;
       while (temp.next != null)
       {
           temp = temp.next;
       }
       temp.next = anotherNode;
       anotherNode.previous = temp;
    }
    currentDataValue += 10;
    length++;
}
```

Doubly Linked List: Adding Anywhere

```
public void addToPosition (int position)
{
    Node anotherNode = new Node (currentDataValue);
    Node temp;
    Node prior;
    Node after;
    int index;
    if ((position < 1) || (position > (length+1)))
    {
        System.out.println("Position must be a value between 1-" +
                           (length+1));
    }
```

Doubly Linked List: Adding Anywhere (2)

```
else
{
   // List is empty
   if (head == null)
   {
      if (position == 1)
      {
         currentDataValue += 10;
         length++;
         head = anotherNode;
      }
      else
         System.out.println("List empty, unable to add node to " +
                            "position " + position);
   }
```

Doubly Linked List: Adding Anywhere (3)

```
// List is not empty, inserting into first position.
else if (position == 1)
{
   head.previous = anotherNode;
   anotherNode.next = head;
   head = anotherNode;
   currentDataValue += 10;
   length++;
}
```

Doubly Linked List: Adding Anywhere (4)

```
// List is not empty inserting into a position other than the first
else
{
    prior = head;
    index = 1;
    // Traverse list until current is referring to the node in front
    // of the position that we wish to insert the new node into.
    while (index < (position-1))
    {
        prior = prior.next;
        index++;
    }
    after = prior.next;
```

Doubly Linked List: Adding Anywhere (5)

```
        // Set the references to the node before the node to be
        // inserted.
        prior.next = anotherNode;
        anotherNode.previous = prior;

        // Set the references to the node after the node to be
        // inserted.
        if (after != null)
            after.previous = anotherNode;
        anotherNode.next = after;

        currentDataValue += 10;
        length++;
    }
  }
}
```

Doubly Linked List: Deleting A Node

```
public void delete (int key)
{
   int indexToDelete;
   int indexTemp;
   Node previous;
   Node toBeDeleted;
   Node after;
```

Doubly Linked List: Deleting A Node (2)

```
    indexToDelete = search(key);
   // No match, nothing to delete.
   if (indexToDelete == -1)
   {
      System.out.println("Cannot delete element with a data value of "
                     + key + " because it was not found.");
   }
   else
   {
      // Deleting first element.
      if (indexToDelete == 1)
      {
         head = head.next;
         length--;
      }
```

Doubly Linked List: Deleting A Node (3)

```
else
{
   previous = null;
   toBeDeleted = head;
   indexTemp = 1;
   while (indexTemp < indexToDelete)
   {
      previous = toBeDeleted;
      toBeDeleted = toBeDeleted.next;
      indexTemp++;
   }
   previous.next = toBeDeleted.next;
   after = toBeDeleted.next;
   after.previous = previous;
   length--;
    :        :          :
```

Tracking Two-Dimensional Information

- Example: Student grades[1]

Students

Courses

	[0]	[1]	[2]	…	[30000]
[0]					
[1]					
[2]					
:					
[300]					

[1] Example based on the described in “Data Structures and Algorithms in Java” by Adam Drozdek

Tracking Two-Dimensional Information

- Example: Student grades[1]
- Problem: Wasted space

Students

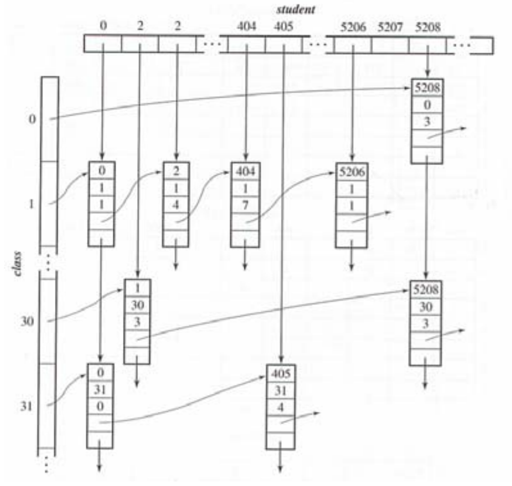
Courses

	[0]	[1]	[2]	...	[30000]
[0]	A	F			
[1]		W			
[2]	B-				
:					
[300]		D			

1 Example based on the described in “Data Structures and Algorithms in Java” by Adam Drozdek

Sparse Matrices/ Sparse Table

- Memory is allocated only as needed (compile arrays and linked lists)



Example based on the described in “Data Structures and Algorithms in Java” by Adam Drozdek

You Should Now Know

- The advantages and disadvantages of implementing a list as an array and as a linked list.
  - The amount of time taken to perform different list operations on an array vs. a linked list.
- How different types of linked lists are implemented, issues associated with each implementation and the speed of different list operations.
- What is a sparse table and what is the advantage and disadvantage of implementing it as an array vs. as a linked list.

Sources Of Lecture Material

- *Data Structures and Abstractions with Java* by Frank M. Carrano and Walter Savitch
- Data Abstraction and Problem Solving With Java: Walls and Mirrors by Frank M. Carrano and Janet J. Prichard
- "Data Structures and Algorithms in Java" by Adam Drozdek
- CPSC 331 course notes by Marina L. Gavrilova http://pages.cpsc.ucalgary.ca/~marina/331/