Abstract Data Types and Lists

Linked Lists
Abstract Data Types

1. An ADT is a set of objects with a set of operations.
   • Sound familiar? Should – that’s a data structure!

2. But how the operations are implemented is **not** defined. (unlike a data structure)

Once we define the implementation of the operations, an ADT becomes a Data Structure.
ADT as Abstract Class

- Those with a background in OO-programming will recognize an ADT as an abstract class!

- The operations are abstract methods that will be completed by the child class.
Examples of ADTs

- sets of integers with operations for “union” and “intersection.” (But we don’t specify how to do these operations – there could be slow and fast ways to do this.)

- lists of words with operations of “order”, “search.” (We already know that the binarySearch would be better than the bruteForceSearch – but we leave the implementation up to the user.)

- genealogical tree with operations “getFather” and “getFirstAncestor”
Are the Following ADTs?

• Groups of cars with operations “getVintage” and “sortByColor”.

• Arrays of integers with operators “addition,” “multiplication,” and “binarySearch.”

• Packs of dogs with only one operation “alphabetizeByBreed”.

More Abstract ADTs

• sets with operations for “union” and “intersection.” (Could be sets of integers, cars, people, whatever.)

• lists with operations of “order”, “search.” (Could be a list of words, endangered species, …)

• tree with operations “getFather” and “getFirstAncestor” (could be genealogical, or a pine tree, or a Java/C++ class hierarchy)
Detailed Example.
The List ADT: Definitions

- Consider the list $A_1, A_2, A_3, \ldots, A_N$.
- Vocabulary:
  - Say that the *size* of the list is $N$.
  - Say that the *empty list has size* 0.
  - Say that $A_{i-1}$ *precedes* $A_i$.
  - Say that $A_i$ *follows* $A_{i-1}$.
  - Say the *position* of $A_i$ is $i$.
- For simplicity, assume lists of integers, but could be anything.
The List ADT: Operations

- **Popular choices:**
  - `insert(x, k)` (add element at kth position)
  - `delete(k)` (delete elem. at kth position)
  - `findKth` (find element at position k)
  - `find(x)` (find position of element)
  - `next` (find element after a specified element)
  - `previous` (find element before a specified element)
  - `printList` (display list)
  - `makeEmpty` (get rid of contents)
How Are the List ADT Operations Implemented?

• **Answer:** However you want!

• `find(52)` might use `binarySearch` or `bruteForceSearch`.

• `insert(x, k)` might use an array to store values, and might move elements around to make room for new one.

• **But some choices are better than others!**
(Obvious/Simple) Array Implementation of List ADT

• Store the list in an array, A[i], of size N.
  • findkth(k) works beautifully. Just return A[k].
    » Takes constant time (independent of which k you choose). O(1).
    » Can you write the code?
  • printList. Just run through the array and print one at a time.
    » Uses one “for” loop. Linear time. O(N).
    » Can you write the code?
  • find(n). Can use brute force or binary search if list is ordered.
    » O(N) or O(log(N))
  • insert? Uh oh…
List ADT: Array Insertion

- Suppose we insert $x$ at position 0.
  \( \text{insert}(x, 0) \)
  - Must move every element in the array down one position. That takes $N$ operations (one for moving each element in the array).
    
    1, 4, 32, 6, 29, -3
    
    $x$, 1, 4, 32, 6, 29, -3

  - So inserting one number is $O(N)$.
  - And inserting $N$ numbers could take up to $N^2$ time!
    
    » We often have to do many inserts – three, twelve, or $N$.
    » That’s the worst case, depending on where they are inserted.

Bad, bad, bad…
List ADT: Array Deletion

- Same problem as insertion.
  - On average will have to move at least half the list up one position.
  - If don’t move elements, then findKth won’t work.

- And if delete 0th position will have to move the whole list; N operations or O(N).

- So deleting a single element will be O(N).
- And deleting N elements will take O(N^2).
  - And we usually have to delete more than once – five, twelve, N times.

- Bad, bad, bad! Doom and gloom.

But we can offer a better solution!
Linked List

- Concept: Don’t store things contiguously (next to each other in memory).
  - Why not? Because if all next to each other, then have to move big parts of the list to do inserts.

- Use **links** that point to the next element
  - In C will use pointers! In Java will just keep track of which object comes next in the list.
Linked List Picture

Linked List

Linked List in C (with pointers)
Deleting From a Linked List

• To remove $A_3$, just change the pointer.

Don’t have to remove this link. $A_3$ is “lost” so it won’t matter. No way to get to $A_3$. 
Same Deletion Showing Pointer Numbers

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>501</td>
<td>A₂</td>
<td>542</td>
<td>A₃</td>
</tr>
<tr>
<td>100</td>
<td>501</td>
<td>542</td>
<td>137</td>
<td>137</td>
</tr>
</tbody>
</table>

Now points to A₄ instead of A₃!

Don’t have to remove this pointer. A₃ is “lost” so it won’t matter. No way to get to A₃.
Inserting Into a Linked List

- Just change the pointers. Notice that I never lose track of any element in the original list – there’s always an arrow pointing from the first element to each other original element.
### Same Insertion Showing Pointer Numbers

<table>
<thead>
<tr>
<th></th>
<th>A&lt;sub&gt;1&lt;/sub&gt;</th>
<th>A&lt;sub&gt;2&lt;/sub&gt;</th>
<th>A&lt;sub&gt;3&lt;/sub&gt;</th>
<th>A&lt;sub&gt;4&lt;/sub&gt;</th>
<th>A&lt;sub&gt;5&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>501</td>
<td>542</td>
<td>137</td>
<td>641</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>933</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>501</td>
<td>542</td>
<td>137</td>
<td>641</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>542</td>
<td></td>
<td>933</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>501</td>
<td>933</td>
<td>542</td>
<td>641</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>542</td>
<td></td>
<td>933</td>
<td></td>
</tr>
</tbody>
</table>

1. Number 933 is inserted at position 2.
2. Number 933 is inserted at position 3.
Why Is This Linked List Better?

• **Delete takes constant time, O(1).**
  - Just move one pointer!
    » Note this assumes we are already at the element we want to delete. So we don’t have to find that element which would add O(N).

• **Insert takes constant time, O(1).**
  - Just create a new element and move two pointers!
    » Note this assumes we are already at the element we want to insert. So we don’t have to find that element which would add O(N).

• **Can insert/delete N elements in O(N) time.**
  - Recall that the array implementation of the List ADT required O(N^2). So this is much faster!
But What About the Other Methods?

- **printList.** Start with the first element and then follow the pointers. \(O(N)\) as before.

- **find.** Ditto. \(O(N)\) which is no worse than the brute force, but binary search no longer works.

- **findkth(i).** Takes \(O(N)\) time. Worst case: have to follow links through all \(N\) elements. Uh, oh. Not as good as the \(O(1)\) from array implementation.
So Which Do I Use?
Array or Linked List?

• Use whichever is appropriate to the problem at hand.

• Will you be doing 1000’s of inserts? Or will you be accessing some element 1000’s of times? Or both?

• Which should you use in each case?
Pseudo-Code For Linked List: Node

Node
{
    int value;
    pointer to next node;
}

Real Code For Node

**In Java:**

```java
public class ListNode {
    public int value;
    public ListNode next;

    //constructor
    public ListNode (int nodeValue, ListNode nextNode) {
        value = nodeValue;
        next = nextNode;
    }
}
```

**In C:**

```c
struct Node;
typedef struct Node *PtrToNode;
typedef PtrToNode Position
struct Node
{
    int value;
    Position next;
};
```

// C code is shorter because pointers are // more transparent
Pseudo-Code For Linked List: insert

```java
insert(int n, Node currentNode)
{
    if(list is not empty)
    {
        nextNode = get “next node” from “current node”; 
        newNode = create a node;
        set value of newNode to n;
        set link of newNode to nextNode;
        set link of currentNode to newNode;
    }
}
Can draw out how this works with boxes and arrows.
```
How About the Real Code?

• See handout for Java versions.

• Read ‘em. Not too complicated.
  • We are now at the part of the course where ability to read/write code is essential!
Example

• Consider storing the rankings of all baseball teams, and store them in order.
  • Rankings change constantly.
  • Array list or linked list?

• Consider storing the sizes of every major city in the world, and store them in order.
  • Relative sizes do not change all that often.
  • Array list or linked list?
Doubly Linked Lists

- Sometimes want to be able to go backwards. Grab the previous element in the list.

- Easy: add another pointer (link) that points you backwards to the previous elements.