Introduction to Data Structures

Two ways of storing data in memory

Part II. Linked Lists

Lecture 6 by Marina Barsky
Outline

• Discuss two alternative ways of storing a *sequence* of values:
  • Array
  ➢ Linked List

• Functionality:
  • Get element by position (index)
  • Search for a position of a target element
  • Add new element at a given position
  • Remove an element at a given position

Java implementation:
  • ArrayList
  • LinkedList
  • Java Generics
Alternative way of storing things: Linked Structures

Reference to (address of) the first element of an array: A[0]

Reference to the first node

Array

Linked list
Linked List – recursive definition

- Main element of the linked list: `Node`
- Each `Node` contains inside a link to the next `Node`

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```
What we do with Linked Lists?
Same things as with Arrays

• Read operations:
  • get (index i)
  • find (Object o)

• Edit operations:
  • add()
  • remove()
Traversing the list

1. Reference to *head* is all we need to know
2. We follow the sequence by following the links
3. We stop when there is no *next*

```java
public class Node{
    int data;
    Node next;
}

static void traverse (Node head) {
    Node current = head;
    while(current.next != null)
        current = current.next;
}
```
What we do with Linked Lists

• Read operations:
  • get (index i)
  • find (Object o)

• Edit operations:
  ➢ add()
  • remove()
Add in front

1. Create new node
Add in front

1. Create new node
data: 1
next: NULL

2. Chain it into the list by setting its next link to the old first node, pointed to by head

node A
- data: 2
- next: node B

node B
- data: 3
- next: node C

node C
- data: 4
- next: NULL

node D
- data: 1
- next: NULL

head: points to node A
Add in the middle (at position 1)

1. Create new node
2. Chain it into the list by setting its next link to the old first node, pointed to by head
3. Update the head of the list: new node D becomes the head
Add in the middle

1. Create new node

node D
- data: 1
- next: NULL

node A
- data: 2
- next: node B

node B
- data: 3
- next: node C

node C
- data: 4
- next: NULL

head: points to node A
Add in the middle

1. Create new node D
2. Advance one step: current node is node A
Add in the middle

1. Create new node D
2. Advance one step: current node is node A
3. Set `next` of D to the `next` of A
Add in the middle

1. Create new node D
2. Advance one step: current node is node A
3. Set next of D to the next of A
4. Set next of A to node D

node A  
data: 2  
next: node D

node B  
data: 3  
next: node C

node C  
data: 4  
next: NULL

head: points to node A
Add in the middle

Just 3 rewiring operations are required to add after a given node
1. In order to add to the end of the list we first need to find the end
2. We need to traverse the entire list until we find a node with `next = NULL`
We can enhance Linked List by storing the pointer to the last node - the **tail pointer** - and update it after each add/remove.

If we know the address of the last node, then we can add an element at the end of the list by a single chaining operation - think: how?
Which statement(s) are True?

A. It is faster to find an element by position in Array than in the Linked List
B. It is faster to find the position of a given element in Array than in the Linked List
C. It is faster to resize the storage capacity of the Array than that of the Linked List
D. All of the above
E. None of the above
Summary. Linked list

**Advantages**
- Can hold unlimited number of elements
- Adding/Removing in the beginning and the end is cheap
- Adding/Removing in the middle requires some work, but does not require moving other elements (like as in Array)

**Disadvantages**
- No constant-time access to an element by its position
- Memory overhead (to store links)
- To search we must to traverse the entire list
Java List interface

- We saw that both *Array* and *Linked List* have the same functionality.
- In *java.util* library we find *List* Interface, which factors out the common methods for working with a sequence of elements.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get (index)</td>
<td>Searches for an object and returns its position</td>
</tr>
<tr>
<td>indexOf (object)</td>
<td></td>
</tr>
<tr>
<td>add (element)</td>
<td></td>
</tr>
<tr>
<td>add (position, element)</td>
<td>Adds at the end</td>
</tr>
<tr>
<td>remove (index)</td>
<td></td>
</tr>
</tbody>
</table>
Array and Linked List in Java

<<interface>> List

get (index)
indexOf (object)
add (element)
add (position, element)
remove (index)

In doubly-linked list each node stores link to its child and to its parent node:

class Node {
    int data;
    Node next;
    Node previous;
}
You’re implementing your own Array List, and you want it to be able to hold any object (i.e. Integers, Strings, People ...). What is the best way to specify the return type for the get method?

A. This is impossible: I would need to implement a separate class for each data type

B. Have get return type Object

C. Somehow set the type only when we create the ArrayList

D. Do something else
Lists for storing elements of any type

- We can implement all the methods for a list of integers
- What if now we want to store Strings? Or Dogs? Should we rewrite all the methods?
- Maybe we should write the code to always store Objects?
- But in this case:
  - We would be able to add anything into the same list!
  - We would need to cast each Object back to String or Dog to be able to do something useful with it
  - The cast may fail during the runtime! Type safety is violated
Java’s solution: Generics

```java
public class Node<T> {
    private T data;
    private Node<T> next;

    public Node(T data) {
        this.data = data;
    }

    public T getData() {
        return this.data;
    }

    public Node<T> getNext(){
        return this.next;
    }

    public void setNext(Node<T> next) {
        this.next = next;
    }
}
```

- Java's solution to this is to allow classes to parameterize types
- Here is a class that can be used with any type of data
- `<T>` is a type parameter
- Note reference to type T through the class body
- All Ts will be substituted with an actual type during compilation
Generic class can be used with any data type

- `<T>` is just a placeholder for the future data type
- A specific instance of Node must choose which actual type should be used, and all Ts in code will be substituted with this type

```java
public class Node<T> { ...
}

public static void main(String [] args) {
    Node<String> n1 = new Node <String> ("message");
    System.out.println(n1.getData());
}

public class Node {
    private String data;
    private Node next;

    public Node(String data) {
        this.data = data;
    }

    public String getData() {
        return this.data;
    }
    ...
}
```

The `Node` class looks like this after compilation
Reference types only

- The actual types that can substitute type parameters `<T>` MUST be of *reference type*
- Primitive types, such as `int`, `boolean`, and `float` are not allowed
- Fortunately, Java provides *wrapper* classes for each of the primitive types:
  - For example, `Integer` is a Java class that holds a single `int` value
  - Java even automatically wraps and unwraps primitive types:

```java
Node<Integer> n2 = new Node<Integer>(55);
int num = n2.getData();
```
What is faster?

<table>
<thead>
<tr>
<th>Operation</th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get i-th element</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search for an element</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add new element at the end</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add element in the middle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove from the end</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove from the middle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove from the middle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resize when full</td>
<td>Red</td>
<td></td>
</tr>
</tbody>
</table>

We need to learn algorithm analysis tools to answer this

See next lecture...