Lists and List Processing

Data Structures

Introduction

- In this section we will look at perhaps the most general-purpose data structure there is: the list.
  - You will already have used lists in some fashion (arrays, ArrayList, etc.).
- We will consider a number of idioms for creating a list-based data structure, including using arrays and node linking ("linked lists").
  - Different pros and cons with different approaches.
- We will consider the built-in methods in Python for using Lists and how to build a List class from scratch.

Lists

- A list is an ordered collection or sequence of elements.
- Elements in a list are numbered/indexed, allowing users to access elements by index.
  - Accessed elements can typically be read, modified, and deleted. Insertion of a new element at an index point is also generally supported.
- Because lists are sequences, it is easy to search or process a list sequentially (using a loop/iteration).
- Duplicate entries are usually permitted in a list.
  - Application logic may restrict this if required.

Array-Based Lists

- Arrays are ordered collections that hold a fixed/specified number of items of a given type.
  - They are, essentially, fixed or statically-sized lists.
- Arrays are typically stored internally as contiguous blocks of memory.
  - Allocated when the array is created, and cannot be extended.
- Index 0 is the first item in an array (zero-based indexing) and index "len(array)-1" is the last.

Continued…

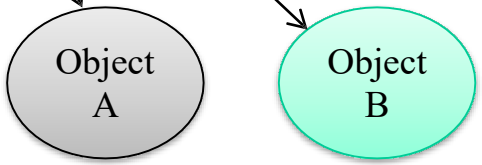
- All elements in an array must be of the same type.
  - Though that type could be Object (or some other generic class type e.g. Animal, Shape) which would allow for different types of objects to be stored in it.
    - Well, technically, only object references are stored in such an array and the objects themselves are elsewhere in memory. For arrays of primitive types (e.g. int) the data is stored in the array itself.

int[]

0	1	2	3	4
34	2	89	-46	443

Object[]

0	1	2	3	4
		null	null	null



Advantages of Arrays

- The major advantage of arrays is speed and ease of access to elements.
- As each element "slot" in an array is of a fixed size, that size multiplied by the index of the element required easily calculates the byte offset into the array from its starting position in memory. This allows for immediate and direct access to the element by memory address (there is no need to "hop count" until you reach the element required).
  - Simple example: to read element index 57 in an int array stored at memory location 5000, go to memory location 4*57+5000=5228 and read 4 bytes (an int is 4 bytes).

Continued…

- Another advantage of arrays are that they are efficient when storing primitive types (raw numbers!) as the elements themselves are not objects.
  - Every object created has an overhead in terms of memory, etc. For instance, an Integer object takes more resources than an int primitive.
- Another advantage is the ease of indexing and the nice and short syntax offered by the [] square brackets.
  - somearray[4]
  - Even for multi-dimensional arrays e.g. array[5][2][0]
  - Note that some array-based lists have class wrappers, so the resultant objects have to be used in the usual object.method() way ([] square brackets won't work!).

Disadvantages of Arrays

- The main disadvantage of arrays is that they are fixed in size when created.
  - They are statically and not dynamically sized.
- This leaves the programmer with a dilemma:
  - Either create an array that is potentially too small (leading to problems/error/failure if it fills up!)
    - In such cases, a common solution is to create a new bigger array and copy the contents of the old one into it.
  - Or create an array that is potentially too big (leading to memory being wasted as slots go unused)
    - As the array's allocation must be in contiguous memory, this may be problematic for very large arrays too.

Continued…

- Another disadvantage of arrays is that what parts are being used, and what parts are free/available, has to be managed by the programmer.
  - For instance, keeping an index reference to the next "free" slot and incrementing it as slots are used.
  - Alternatively, free slots could be "marked" with a special constant value. These free slots could be sequentially searched for in order to use them. (Efficiency? Gaps?)
- Array insertions/deletions may require all existing elements above the insertion/deletion point to shift up/down by one.
  - While this shifting is logically easy to implement, it is inefficient for large arrays with frequent insertions/deletions.
  - Can delete without shifting if gaps are acceptable (mark slot as free).

Simple / Singly Linked Lists

- A linked list is a dynamically-sized data structure consisting of linked nodes representing elements in the list.
- In a simple linked list, nodes are singly linked (i.e. they are unidirectional, or one way).
- The general idiom for linked lists is that the node type contains an attribute of the same type to reference the next linked node.
- It is important to maintain a reference to the head element in a linked list; other elements can be sequentially accessed by following the "next" link in each node.

Continued…

- For example:

Define list node to include "next" link of same type

```
public class Node {
        public Node next=null; //Or private with getter/setter methods
        …
        //Add other attributes and methods etc.
}
```

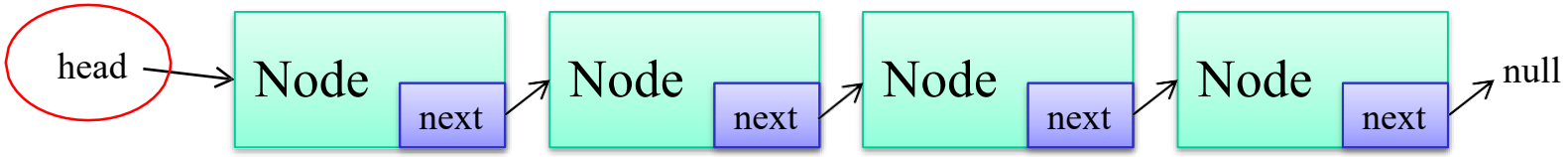
Keep reference to list "head"

```
Node head; //Keep reference to first element
```

Sequential access by following "next" links in turn

```
Node temp=head;

while(temp!=null) {
        //Process node referenced by temp
        temp=temp.next;
}
```



Advantages of Linked Lists

- The main advantage of linked lists is that they are dynamically sized and can grow on demand one node at a time.
  - This is memory efficient insofar that sections of contiguous memory don't have to be reserved up front as with arrays.
  - There is no problem of a linked list ever "filling up". Logically, they can be as long as you like (though available memory will technically run out at some point).
- There are no potential gaps or free/available slots to maintain/manage as with arrays.

Continued…

- Insertion into a linked list doesn't require other elements to "shift up" to make room as with arrays.
  - A new Node object is created and the inter-node links adjusted to complete the insertion at the required position. Nothing is moved or shifted!
  - Insertion at the head of a linked list is particularly quick and efficient. Ditto at the end of the list if a reference to the "last" element is also maintained.
  - Insertion at any other point in the list is less efficient as it requires the link sequence to be followed node-by-node until the node after which the insertion will occur is reached.

Continued…

- Deletions from linked lists offer similar advantages to insertions.
- Deletion from a linked list doesn't require other elements to "shift down" to close the gap as with arrays.
  - Deletion at the head of a linked list is particularly quick and efficient (the head simply becomes its own next). However, unlike insertions, deletion at the end of the list can't be done with a reference to the "last" element.
    - Q. Why?
  - Deletion at any other point in the list requires the link sequence to be followed node-by-node until the node after which the deletion will occur is reached.
  - Note that the automatic garbage collector in Java will deal with (i.e. delete or deallocate) unreachable/orphaned objects.

Disadvantages of Linked Lists

- Access to individual elements in a linked list requires a "hop count" approach that sequentially follows the node links, counting if appropriate, to reach/retrieve an element.
  - This is much less efficient than the immediate direct access offered by arrays.
  - Accessing elements deep into long linked lists can be quite inefficient/slow. Accessing elements near the head can be quite fast.
  - No simple square brackets [] notation to access by index.
- Linked lists require node objects to represent individual elements (to include the "next" reference, etc.). Even primitive types would need to be "wrapped" in this way.
  - Not as efficient at storing large amounts of primitive/numeric data as arrays.
  - As already noted, there are resource overheads for "objects".
  - Note than non-OO languages have other non-object ways of representing nodes for linked lists (e.g. struct in C), but the principles are the same.
- In many respects, the disadvantages of linked lists are the advantages of array-based lists, and vice versa.

Doubly Linked Lists

- There are variants on the simple linked list which offer additional advantages (or mitigate disadvantages) in some situations. The doubly linked list is one such variant.
- A doubly linked list is bidirectional (two way) as nodes are doubly linked: one link to the next node (as before), and also one link to the previous node.
- A reference to the head of the list must still be maintained (as before) but a doubly linked list will usually also have a reference to the last (trailer) node.
  - This allows the list to be processed in either direction.

Continued…

- For example:

```
public class DLNode {
        public DLNode next=null, previous=null;
        …
        //Add other attributes and methods etc.
}
```

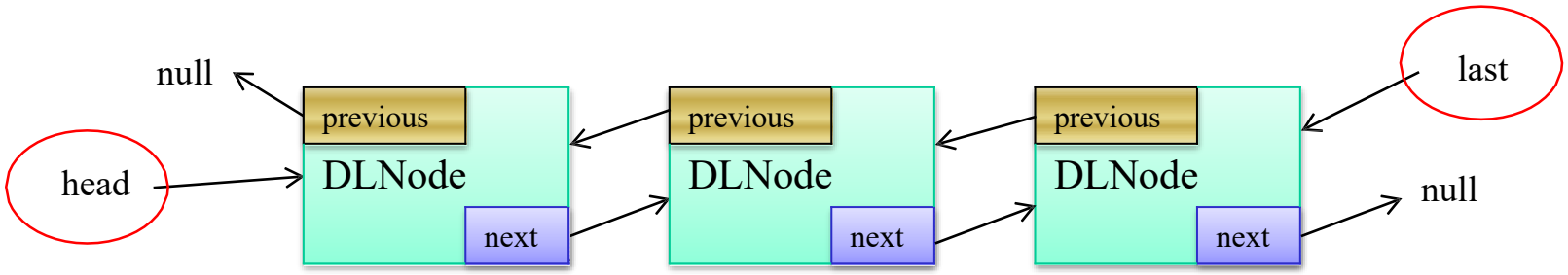
```
DLNode head, last; //Keep reference to first and last elements
```

```
DLNode temp=head;

while(temp!=null) {
          //Process node referenced by temp
          temp=temp.next;
}
```

```
DLNode temp=last;

while(temp!=null) {
          //Process node referenced by temp
          temp=temp.previous;
}
```



Advantages of Doubly Linked Lists

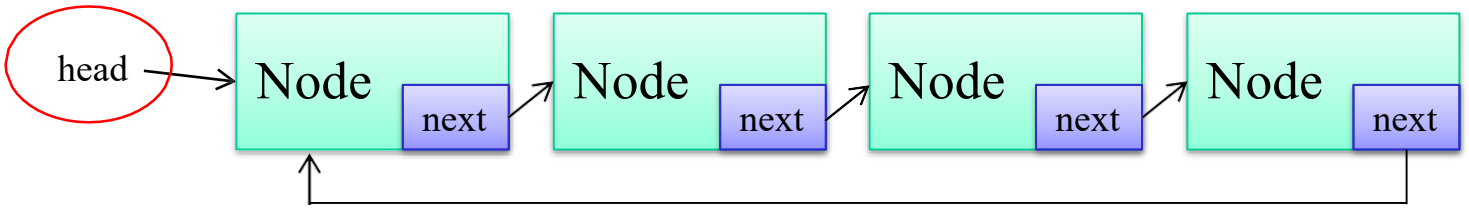
- Long lists can be traversed starting from either direction. Whichever end is likely to be more efficient can be used as the starting point.
  - For instance, if we had a doubly linked list with 100 elements in it and wanted to access element 96, we could start from "last" and follow the "previous" links 4 times (a simply linked list would need 95 hops from the head). To access element 22, say, we would start from the "head". This would require us to maintain a list size attribute, but it would be worth it for long lists.
- Easy to delete from the end of the list.

Continued…

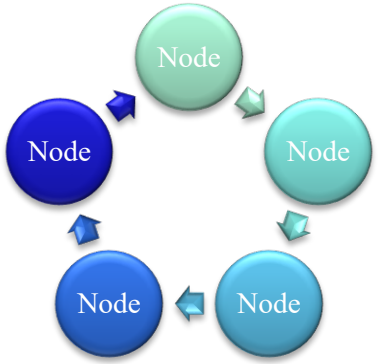
- Easy to insert a new node before another given node in the list.
  - Easy to do after a node in a singly linked list, but you would need a reference to the previous node to insert after.
- Nodes having access to their immediate neighbouring nodes (both predecessor and successor) can be useful for some forms of (re)iterative processing, sorting, etc.
  - As you can dynamically navigate “backwards” as well as “forwards” as required.

Circularly Linked Lists

- Another variant is a circularly linked list, whereby the next reference of the last element refers back to the head.
  - And the previous reference of the head element refers to the last in the case of a doubly circular linked list.



- It essentially forms a ring-type structure suitable for iterative/repetitive processing without having to manage the last element differently.



  - Just keep following "next" indefinitely.
  - Can check to see if "head" has been returned to in order to circulate once.
  - Can be unidirectional (singly linked) or bidirectional (doubly linked).

Continued…

- Examples of processing for a circular linked list (using Node objects as before, assuming all links are correct):

```
//Process indefinitely
Node temp=head;

while(true) {
        //Process node referenced by temp
        temp=temp.next;
}
```

```
//Process once
Node temp=head;

do{
        //Process node referenced by temp
        temp=temp.next;
}while(temp!=head);
```

List Processing and Iteration

- List processing is usually sequential or linear whereby each item is processed in turn starting with (e.g.) the first one.
  - Applies to array-based and linked lists.
- The general idiom of linear list processing is to use an iteration structure (i.e. a loop).
  - Recursion can also be used (more detail later on in the module).

Linear Search

- A common use for linear processing is to search for a particular element by considering each element of the list in turn.
- This is a "linear search".
- The search may return the found element, a reference index to the element, or some indicator that the element wasn't found.
- Linear search is a brute force, exhaustive, blind technique. It will always work, but it is impacted (linearly) by the size (N) of the list being searched.
  - Ideally, the sought element will be found early in search (e.g. towards head of the list).
  - However, the element may not be found (won't know until search has been exhausted) or found late in search (e.g. towards the last element).

Continued…

- Examples:

```
//Linear search of array of Student objects by student id.
//Assume getStudentId() is defined.
for(Student s : studentArray)
        if(s.getStudentId()==soughtId) {
                //Student object s is the sought object here
        }
```

```
//Another variant if the array index is required
for(int i=0;i<studentArray.length;i++)
        if(studentArray[i].getStudentId()==soughtId) {
                //studentArray[i] is the sought object here
                break;
        }
```

```
//Linear search of linked list of Student objects by student id.
//Assume "public Student next" reference is also defined in Student
//and "head" is the first Student node.
Student temp=head;
while(temp!=null && temp.getStudentId()!=soughtId)
        temp=temp.next;
if(temp!=null)
{
        //Student object temp is the sought object here
}
```

Summary

- In this section we have looked at:
  - The general-purpose “list” data structure.
  - Array-based lists and linked lists (and their respective advantages and disadvantages).
  - Linked list variants (doubly and circularly linked).
  - List processing (linear search).