Dynamic Array

An Array-Based Implementation - Summary

- **Good** things:
  - Fast, random access of elements
  - Very memory efficient, very little memory is required other than that needed to store the contents (but see bellow)
- **Bad** things:
  - Slow deletion and insertion of elements
  - Size must be known when the array is created and is fixed (static)

ADT List using Dynamic arrays

- A dynamic data structure is one that changes size, as needed, as items are inserted or removed
  - The Java **`ArrayList`** class is implemented using a dynamic array
- There is usually no limit on the size of such structures, other than the size of main memory
- Dynamic arrays are arrays that grow (or shrink) as required
  - In fact a new array is created when the old array becomes full by creating a new array object, copying over the values from the old array and then assigning the new array to the existing array reference variable

Dynamic Array

top = 4

6	1	7	8		
0	1	2	3	4	5

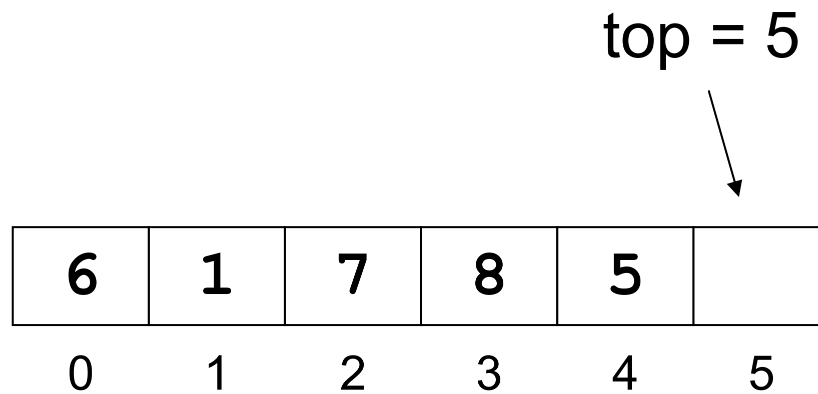
Dynamic Array

top = 4

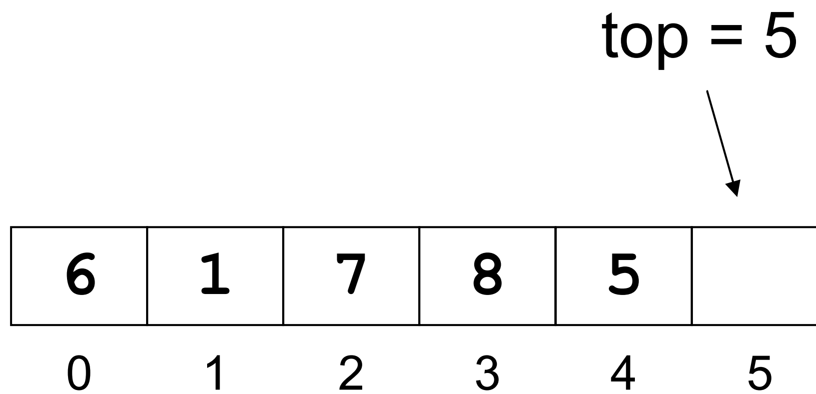
insert 5

6	1	7	8		
0	1	2	3	4	5

Dynamic Array

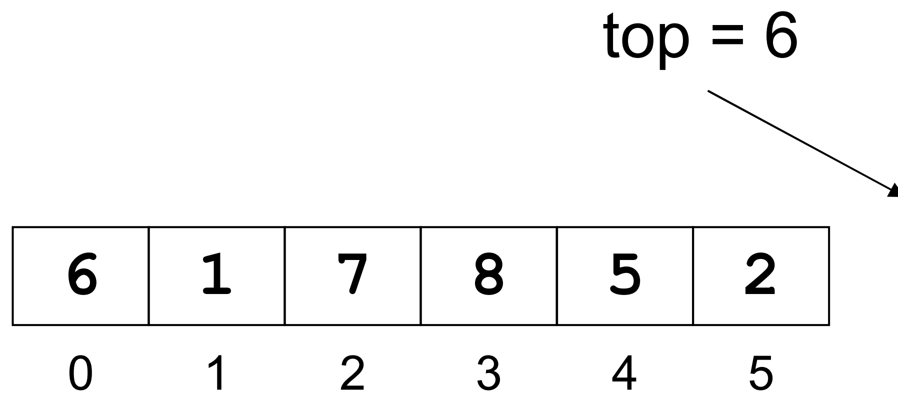


Dynamic Array



insert 2

Dynamic Array



insert 3

Dynamic Array

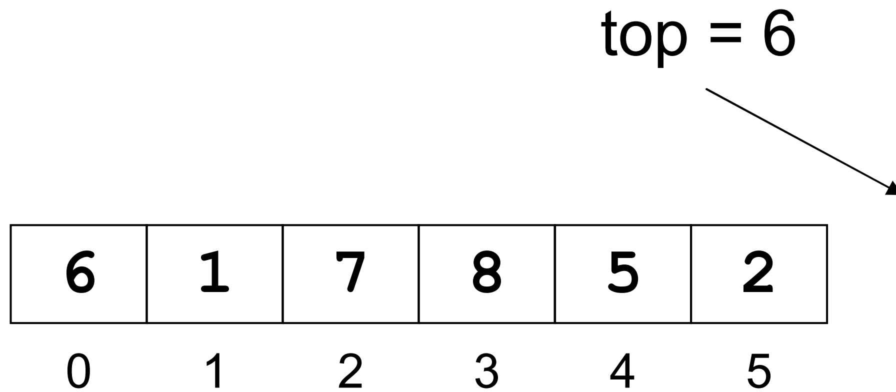
top = 6

insert 3

6	1	7	8	5	2
0	1	2	3	4	5

!The array is full and there is no room for a new item!

Dynamic Array



insert 3

So we will create a new, bigger array …

Dynamic Array

top = 6

insert 3

6	1	7	8	5	2
0	1	2	3	4	5

So we will create a new, bigger array …

0	1	2	3	4	5	6	7	8	9	10	11

Dynamic Array

top = 6

insert 3

6	1	7	8	5	2
0	1	2	3	4	5

… copy the elements of the old array into it …

0	1	2	3	4	5	6	7	8	9	10	11

Dynamic Array

insert 3

6	1	7	8	5	2
0	1	2	3	4	5

top = 6

… copy the elements of the old array into it …

6	1	7	8	5	2						
0	1	2	3	4	5	6	7	8	9	10	11

Dynamic Array

insert 3

6	1	7	8	5	2
0	1	2	3	4	5

top = 7

… and finally insert 3 into the new array.

6	1	7	8	5	2	3					
0	1	2	3	4	5	6	7	8	9	10	11

Dynamic Array

6	1	7	8	5	2
0	1	2	3	4	5

The old array will eventually be deleted by Java's garbage collector

top = 7

6	1	7	8	5	2	3					
0	1	2	3	4	5	6	7	8	9	10	11

```
insert(itemType x){
        if(top < capacity){
                //there is room to insert x
                items[top++]=x;
        }
        else{
                //there is no room to insert x.
                //the array has to be expanded.
                expand();
                //inset x as usual
                items[top++]=x;
        }
}

expand(){
        capacity = 2*capacity;
        itemType newItems[] = new itemTypes(capacity);
        for(i=0 through top-1)
                newItems[i] = items[i];
        items = newItems;
}
```

Dynamic Array Summary

- Before every insertion, check to see if the array needs to grow
- Question: When growing array, how much to grow it?
  - Memory efficient? (by 1)
  - Time efficient?

Dynamic Array Summary

- Before every insertion, check to see if the array needs to grow
- Growing by doubling works well in practice, because it grows very large very quickly
  - 10, 20, 40, 80, 160, 320, 640, 1280, …
  - Very few array re-sizings must be done
  - To insert n items you need to do $\approx \log(n)$ re-sizings
- While the copying operation is expensive it does not have to be done often

Dynamic Array Problems

- When the doubling does happen it may be time-consuming
- And, right after a doubling half the array is empty
- Re-sizing after each insertion would be prohibitively slow
- The classes *`java.util.Vector`* and `java.util.ArrayList` use a similar technique to implement a growable array of objects.

```java
import java.util.Vector;

public class VectorTest{
        public static void main(){
                Vector v = new Vector();

                //inserting elements into the Vector
                for(int i=0; i<100; i++)
                        v.add(new Integer(i));

                //deleting elements from the vector
                while(!v.isEmpty())
                        //remove the last element.
                        System.out.println(v.remove(v.size()-1));
        }
}
```

Output:
99
98
…
0

Linked Lists.

Options for implementing an ADT List

- Array
  - Advantages
    - Fast random access of elements (takes constant to access an element).
    - Efficient in terms of memory usage (almost no memory overhead)
  - Disadvantages
    - Has a fixed size
    - Data must be shifted during insertions and deletions
- Dynamic array
  - Advantages
    - Fast random access of elements.
    - Efficient in terms of memory usage (almost no memory overhead)
  - Disadvantages
    - Data must be shifted during insertions and deletions
    - Data must be copied during the resizing of the array
- Linked list
  - Advantages
    - Is able to grow in size as needed
    - Does not require the shifting of items during insertions and deletions
  - Disadvantage
    - Accessing an element does not take a constant time
    - Need to store links to maintain the logical order of the elements (memory overhead).

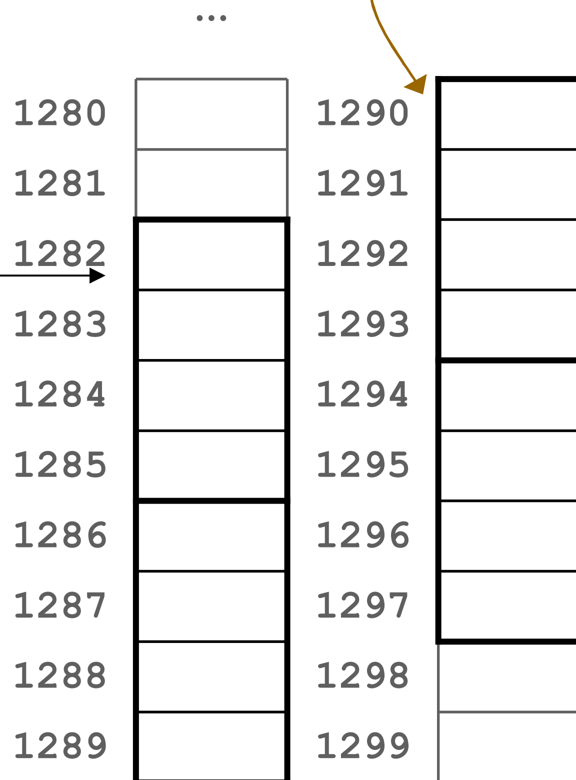
Why random access is fast in array?

- The elements in an array are stored at consecutive locations in the main memory.
- Therefore if we know the address of the first element in the array and the size of each element we can easily calculate the address of any element of the array in the main memory.

Example

```
int grade[]=new int[4];
grade[2]=23;
```

grade



Stores a reference to the start of the array, in this case byte 1282

The `int` stored at `grade[2]` is located at byte:

1282 + 2 * 4 =

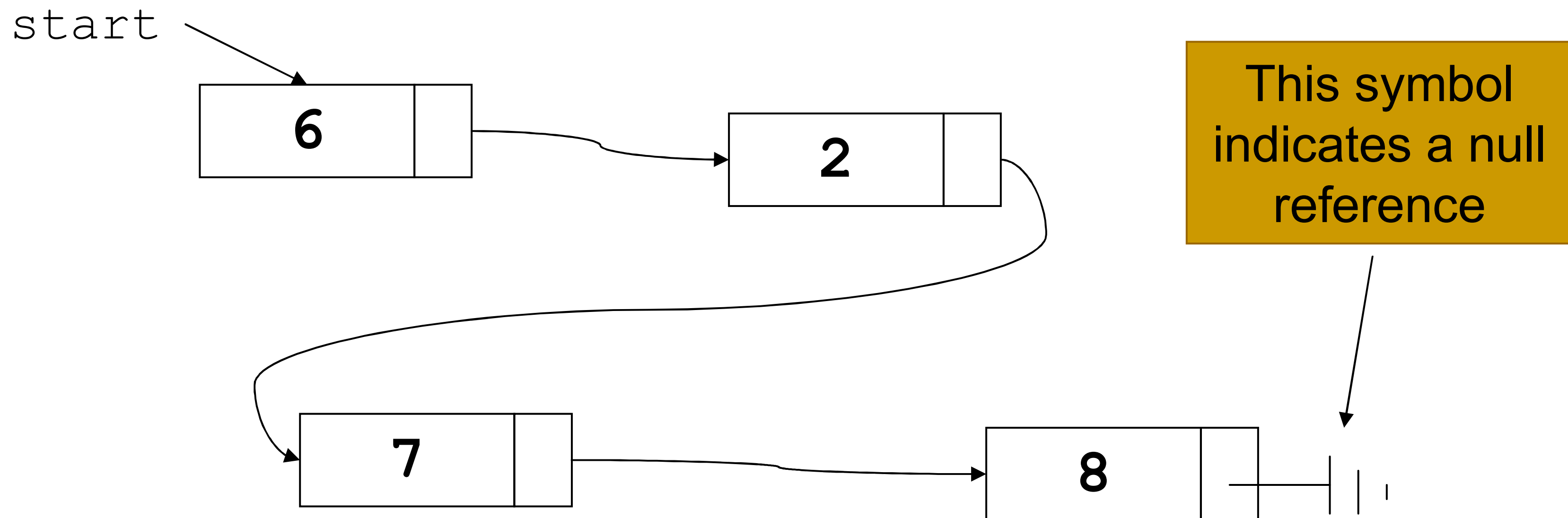
1290

- In general
  address of the element at index = address of first element + ***index*** * ***size of stored type***
- The above calculation can be done in constant time no matter what the value of the *index* is.
- However, to maintain the physical order of the elements in the array we may need to shift some elements when inserting new elements or deleting an element from the array.
- Suppose an array has n elements.
  - In the best case we do not need to shift element when an new element is inserted at the end of array (0 shifts)
  - In the worst case we need to shift all the elements down when an element is inserted in the beginning of the array (n shifts)
  - Therefore on average we need to do (0+n)/2 shifts for inserting a random element in the array.
  - Normally, when we talk about the complexity of operations (i.e the number of steps needed to perform that operation) we don't care about the multiplied or added constants.
  - Therefore, we simply say the insert operation *is of order of n or is O(n).*

- If you need to use a list in a context where insertion and deletion are very often then the array-based implementation of the list in not the best solution.
- In this context a link list implementation of the list is a more efficient solution.

Linked Lists

- A linked list is a dynamic data structure consisting of nodes and links:



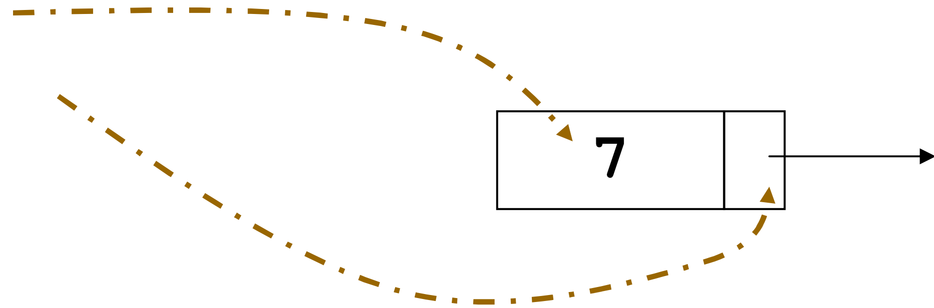
- The physical location of the nodes in the main memory are random (i.e there is no relation between the logical order and the physical order of the elements of a linked list).
- Therefore the logical order of the elements are maintained through the links.

Linked Lists

- Each node contains two things
  - The data
  - A pointer to the next node

Linked Lists

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



Linked Lists

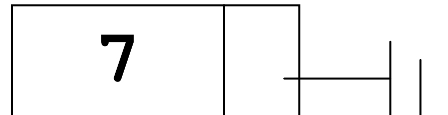
```
class Node {
    public int data;
    public Node next;
}
```

A node points to another node, so the pointer must be of type Node

Building a Linked List

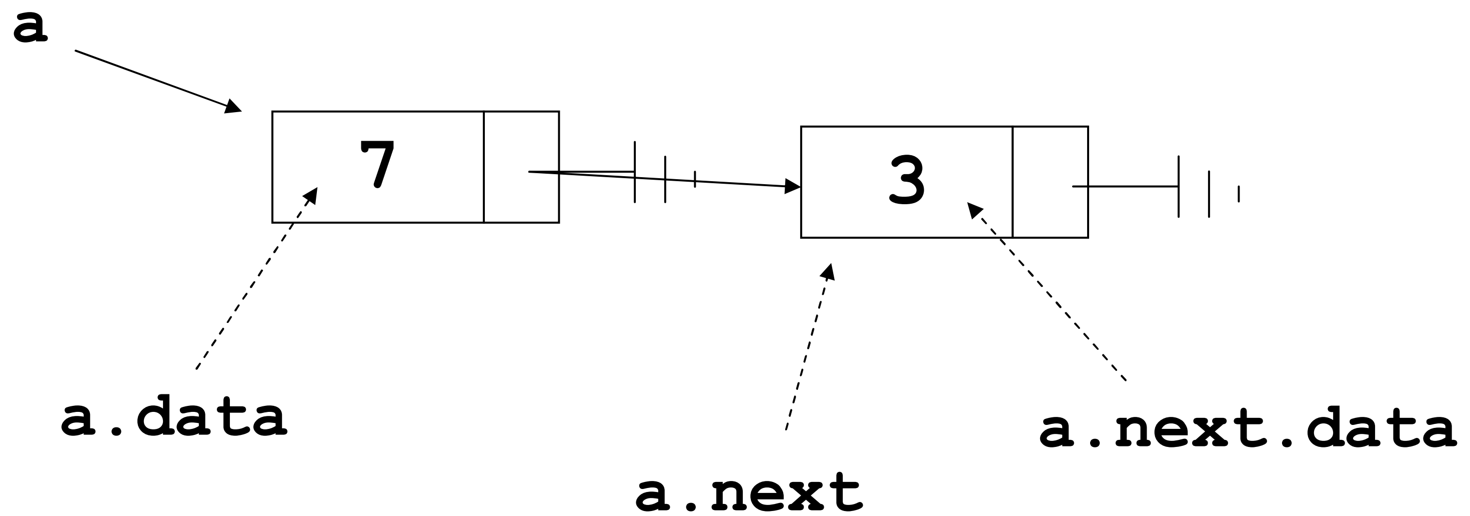
```
Node a = new Node(7, null);
```

Assume we've added a constructor to the `Node` class that lets us initialize data and next like this.



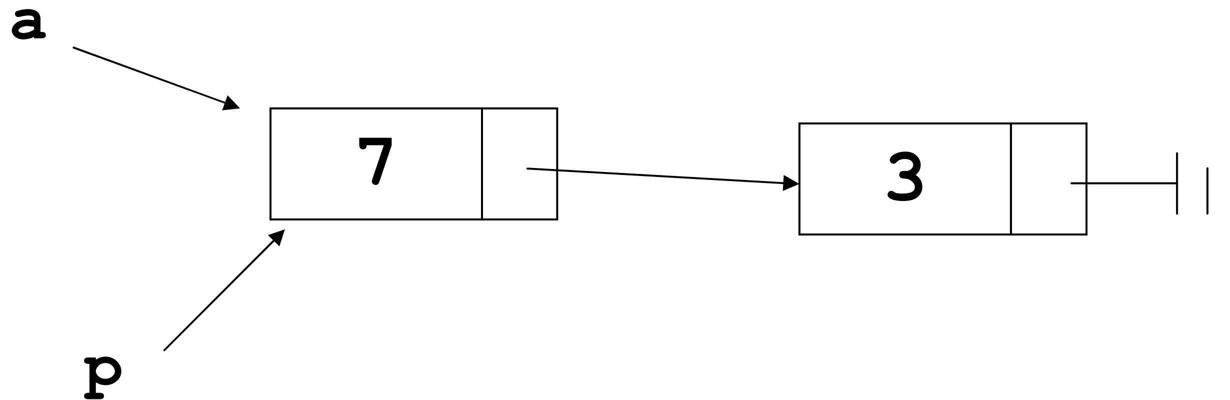
Building a Linked List

```
Node a = new Node(7, null);
a.next = new Node(3, null);
```



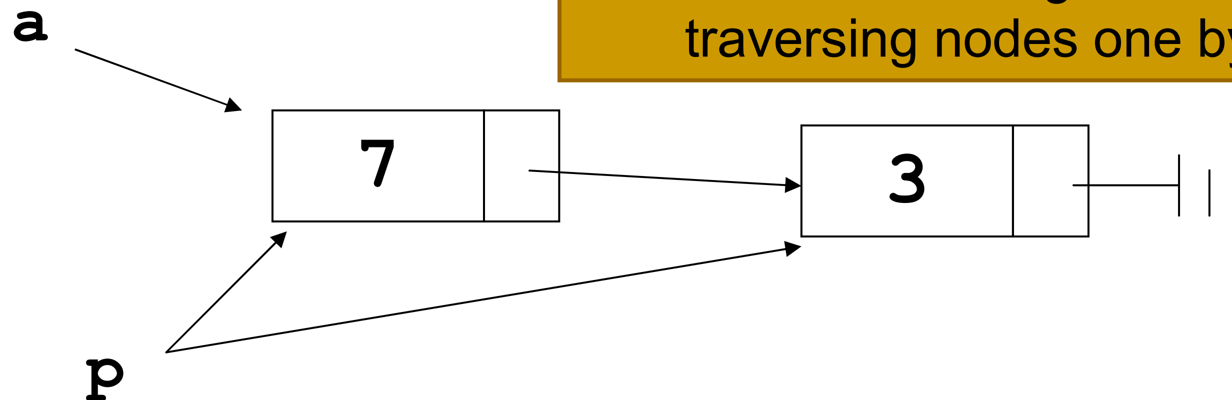
Traversing a Linked List

```
Node a = new Node(7, null);
a.next = new Node(3, null);
Node p = a;
```



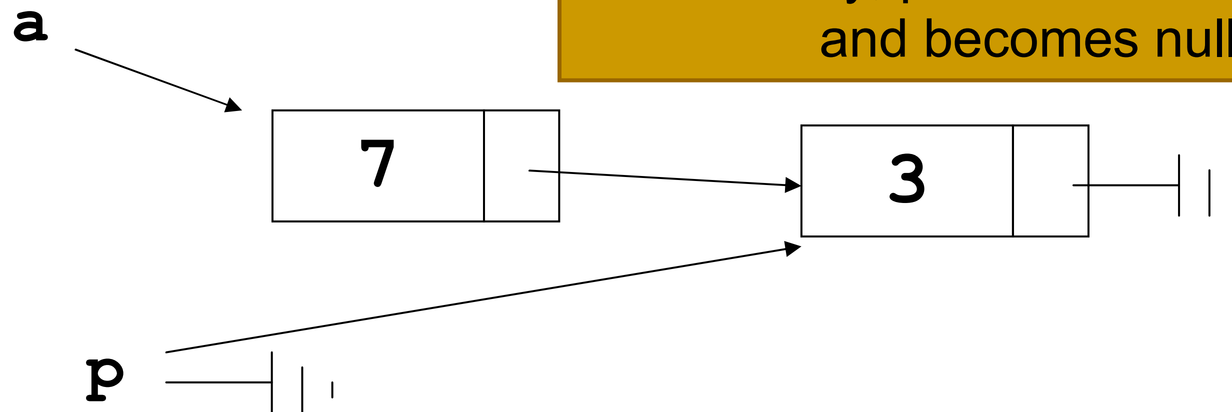
Building a Linked List

```
Node a = new Node(7, null);
a.next = new Node(3, null);
Node p = a;
p = p.next;  // go to the next node
```



Traversing a Linked List

```
Node a = new Node(7, null);
a.next = new Node(3, null);
Node p = a;
p = p.next;  // go to the next node
p = p.next;
```



Eventually, p hits the end of the list and becomes null.

Traversing a Linked List

- The link field of the last node in the list is always null.
- Therefore, if you have the reference to the first node in the list (called head) you can traverse the list (i.e visit all the nodes) using a simple while loop

Example: printing the elements of the list.

```
void public printList(Node head){
        while (head != NULL){
                System.out.println(head.data);
                head = head.next;
        }
}
```