Lecture No.02

Data Structures

Implementing Lists

- We have designed the interface for the List; we now must consider how to implement that interface.

Implementing Lists

- We have designed the interface for the List; we now must consider how to implement that interface.
- Implementing Lists using an array: for example, the list of integers (2, 6, 8, 7, 1) could be represented as:

A	2	6	8	7	1			
	1	2	3	4	5			

<u>current</u>	<u>size</u>
3	5

List Implementation

- add(9); current position is 3. The new list would thus be: (2, 6, 8, 9, 7, 1)
- We will need to *shift* everything to the right of 8 one place to the right to make place for the new element ‘9’.

step 1: A

2	6	8		7	1		
1	2	3	4	5	6		

current	size
3	5

step 2: A

2	6	8	**9**	7	1		
1	2	3	4	5	6		

current	size
4	6

notice: current points to new element

Implementing Lists

- next():

A	2	6	8	**9**	7	1		
	1	2	3	4	5	6		

<u>current</u>

~~4~~ 5

<u>size</u>

6

Implementing Lists

- There are special cases for positioning the current pointer:
    a. past the last array cell
    b. before the first cell

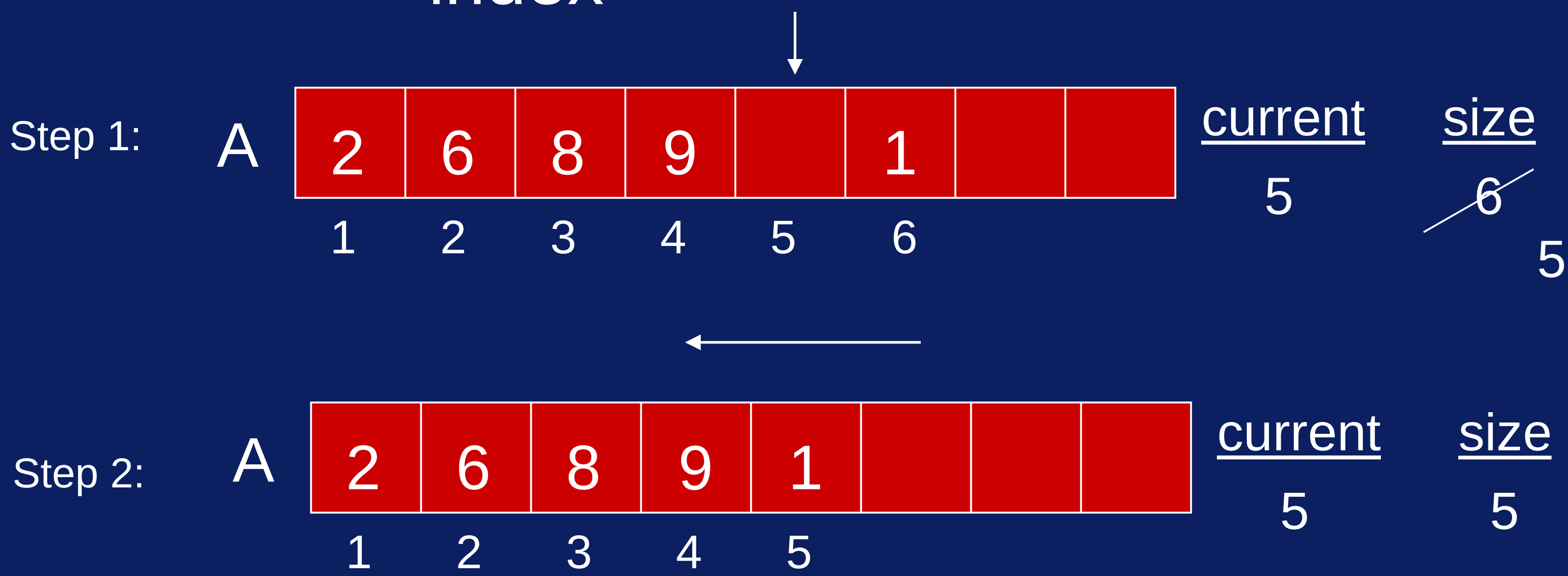
Implementing Lists

- There are special cases for positioning the current pointer:
    a. past the last array cell
    b. before the first cell

- We will have to worry about these when we write the actual code.

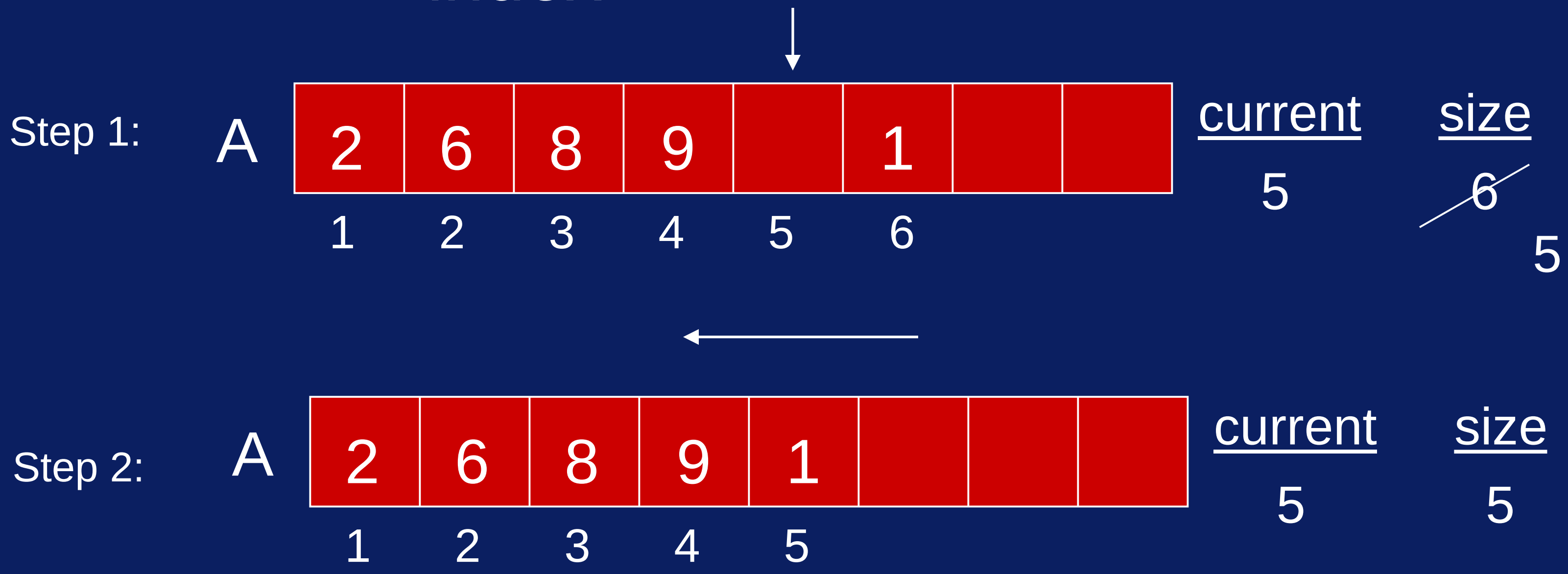
Implementing Lists

- remove(): removes the element at the current index



Implementing Lists

- remove(): removes the element at the current index



- We fill the blank spot left by the removal of 7 by shifting the values to the right of position 5 over to the left one space.

Implementing Lists

find(X): traverse the array until X is located.

```
int find(int X)
{
    int j;
    for(j=1; j < size+1; j++ )
        if( A[j] == X ) break;

    if( j < size+1 ) {        // found X
        current = j;       // current points to where X found
        return 1;   // 1 for true
    }
    return 0;  // 0 (false) indicates not found
}
```

Implementing Lists

- Other operations:

get()	→ return A[current];
update(X)	→ A[current] = X;
length()	→ return size;
back()	→ current--;
start()	→ current = 1;
end()	→ current = size;

Analysis of Array Lists

- add
  - we have to move every element to the right of current to make space for the new element.
  - Worst-case is when we insert at the beginning; we have to move every element right one place.
  - Average-case: on average we may have to move half of the elements

Analysis of Array Lists

- remove
  - Worst-case: remove at the beginning, must shift all remaining elements to the left.
  - Average-case: expect to move half of the elements.
- find
  - Worst-case: may have to search the entire array
  - Average-case: search at most half the array.
- Other operations are one-step.

List Using Linked Memory

- Various cells of memory are not allocated consecutively in memory.

List Using Linked Memory

- Various cells of memory are not allocated consecutively in memory.
- Not enough to store the elements of the list.

List Using Linked Memory

- Various cells of memory are not allocated consecutively in memory.
- Not enough to store the elements of the list.
- With arrays, the second element was right next to the first element.

List Using Linked Memory

- Various cells of memory are not allocated consecutively in memory.
- Not enough to store the elements of the list.
- With arrays, the second element was right next to the first element.
- Now the first element must *explicitly* tell us where to look for the second element.

List Using Linked Memory

- Various cells of memory are not allocated consecutively in memory.
- Not enough to store the elements of the list.
- With arrays, the second element was right next to the first element.
- Now the first element must *explicitly* tell us where to look for the second element.
- Do this by holding the memory address of the second element

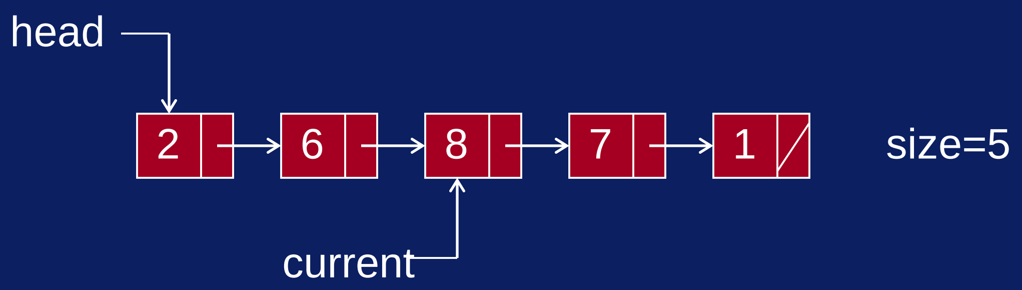
Linked List

- Create a structure called a *Node*.

- The *object* field will hold the actual list element.
- The *next* field in the structure will hold the starting location of the next node.
- Chain the nodes together to form a *linked* list.

Linked List

- Picture of our list (2, 6, 7, 8, 1) stored as a linked list:



Linked List

Note some features of the list:

- Need a *head* to point to the first node of the list. Otherwise we won’t know where the start of the list is.

Linked List

Note some features of the list:

- Need a *head* to point to the first node of the list. Otherwise we won’t know where the start of the list is.
- The *current* here is a pointer, not an index.

Linked List

Note some features of the list:

- Need a *head* to point to the first node of the list. Otherwise we won’t know where the start of the list is.
- The *current* here is a pointer, not an index.
- The next field in the last node points to *nothing*. We will place the memory address NULL which is guaranteed to be inaccessible.

Linked List

- Actual picture in memory: