

Lecture # 2

LIST Data Structure

- The List is among the most generic of data structures.
- In Real life:
 - shopping list,
 - groceries list,
 - list of people to invite to dinner
 - List of presents to get

Lists

- A **list** is collection of items that are all of the **same type** (grocery items, integers, names)
- The items, or elements of the list, are stored in some particular order
- It is possible to insert new elements into various positions in the list and remove any element of the list

Lists

- List is a set of elements in a linear order.
For example, data values a1, a2, a3, a4 can be arranged in a list:

(a3, a1, a2, a4)

In this list, a3, is the first element, a1 is the second element, and so on

- The order is important here; this is not just a random collection of elements, it is an *ordered* collection

List Operations

Useful operations

- `createList()`: create a new list (presumably empty)
- `copy()`: set one list to be a copy of another
- `clear()`: clear a list (remove all elements)
- `insert(X, ?)`: Insert element X at a particular position in the list
- `remove(?)`: Remove element at some position in the list
- `get(?)`: Get element at a given position
- `update(X, ?)`: replace the element at a given position with X
- `find(X)`: determine if the element X is in the list
- `length()`: return the length of the list.

List Operations

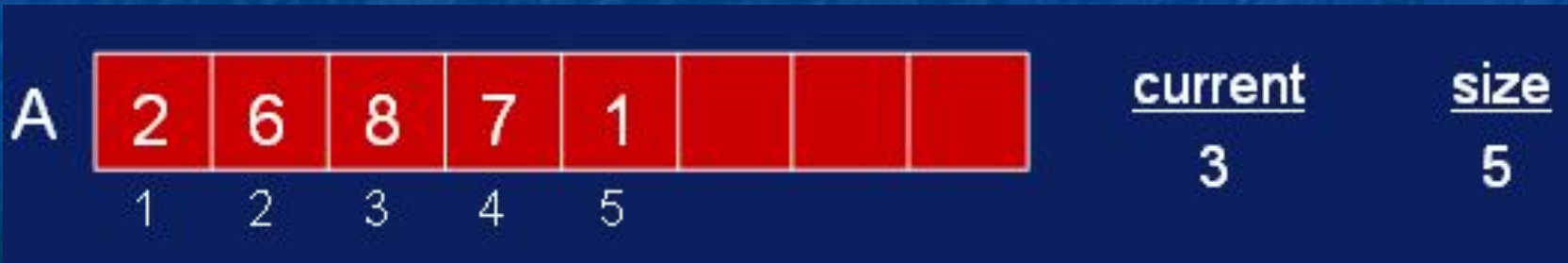
- We need to decide what is meant by “particular position”; we have used “?” for this.
- There are two possibilities:
 - Use the actual index of element: insert after element 3, get element number 6. This approach is taken by arrays
 - Use a “current” marker or **pointer** to refer to a particular position in the list.

List Operations

- If we use the “current” marker, the following four methods would be useful:
- **start()**: moves to “current” pointer to the very first element.
- **tail()**: moves to “current” pointer to the very last element.
- **next()**: move the current position forward one element
- **back()**: move the current position backward one element

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:



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'.



Implementing Lists

- next():

A	2	6	8	9	7	1			<u>current</u>	<u>size</u>
	1	2	3	4	5	6			4 5	6

Implementing Lists

- There are special cases for positioning the current pointer:
 - past the last array cell
 - 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



- 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;

Assignment # 1

- Implement a **List Data Structure** discussed above including following operations using Array ADT.
 - `get()`
 - `update()`
 - `length()`
 - `back()`
 - `Next()`
 - `start()`
 - `end()`
 - `Remove()`
 - `Add()`
- NOTE: Implement above operations only using **Pointers** *without* using any **indexes** of arrays.
- Assignment will be graded on the basis of relevant quiz of this assignment.

List Using **Linked Memory**

- Various cells of memory are **not allocated consecutively** in memory.
- Array is Not enough to store the future elements of the list after full exhaust of array locations.
- With arrays, the second element was right next to the first element.
- Now in **Linked Memory approach**, 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