Abstract Data Types (ADT): Lists

Ph.D. Truong Dinh Huy

(Chapter 3 of textbook 1)

Usual Data Types

Integer, Double, Boolean

You do not know the implementation of these.

There are some operations which can be done with these (add, multiply, read, write)

Programmer just uses these operations without knowing their implementation.

Data structure and Abstract Data Types (ADT)

Data structure: A construct that is defined in a program to store collection of data.

ADT: Mathematical description of a data structure (or an object) and the set of operations on the data structure (or the object).

- Implement the data structure once in the program
- Implement these operations of this data structure once
- Use these operations again and again without going into the implementation
- Ex: List, Set, Graph,…

An ADT may use a different ADT

List ADT

Sequence of elements: $A_0, A_1, A_2, \ldots, A_{n-1}$

We say: A_{i+1} follows (succeeds) A_i or A_i precedes A_{i+1}

Operations:

Add an element at the beginning/end/ith position

Delete an element at the beginning/end/ith position

Access(read/change) an element at the beginning/end/ith position.

Size, Finding, PrintList,…

Array Implementation

List can be implemented as an array

Need to know the maximum number of elements in the list at the start of the program

Adding/Deleting an element can take O(n) operations if the list has n elements.

Accessing/changing an element anywhere takes O(1) operations independent of n

Adding an element

Normally first position (A[0])stores the current size of the list

Actual number of elements currsize + 1

Adding at the beginning:

Move all elements one position behind

Add at position 1; Increment the current size by 1

```
For (j = A[0]+1;  j > 0; j--)
     A[j] = A[j-1];
A[1] = new element;
A[0]→ A[0]+1;
```

Complexity: O(n)

Adding at the End

Add the element at the end

Increment current size by 1;

```
A[A[0]+1] = new element;
A[0]→ A[0]+1;
```

Complexity: $O(1)$

Adding at kth position

Move all elements one position behind, kth position onwards;

Add the element at the kth position

Increment current size by 1;

```
For (j = A[0]+1;  j > k; j--)
    A[j] = A[j-1];
A[k] = new element;
A[0]→ A[0]+1;
```

Complexity: $O(n-k)$

Deleting an Element

Deleting at the beginning:

Move all elements one position ahead;

Decrement the current size by 1

```
For (j = 1;  j  < A[0] ; j++)
    A[j] = A[j+1];
A[0]→ A[0]-1;
```

Complexity: O(n)

Deleting at the End

Delete the element at the end

Decrement current size by 1;

```
A[0]→ A[0]-1;
```

Complexity: O(1)

Deleting at the kth position

Move all elements one position ahead, k+1th position onwards;

Decrement the current size by 1;

```
For (j = k;  j < A[0]+1; j++)
    A[j] = A[j+1];
A[0]→ A[0]-1;
```

Complexity: O(n-k)

Accessing an Element at the kth position

A[k];

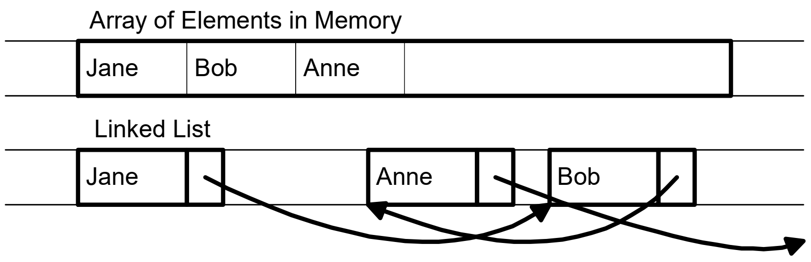
O(1) operation;

Limitation of Arrays

- An array has a limited number of elements and the size must be known before using.
  - routines inserting a new value have to check that there is room
- Insertion/ Deletion is complex. We need to shift the existing elements.
- A better approach: use a *Linked List*

Dynamically Allocating Elements

- Allocate elements one at a time as needed, have each element keep track of the *next* element
- Result is referred to as linked list of elements, track next element with a pointer



Linked List

Consists of a sequence of nodes

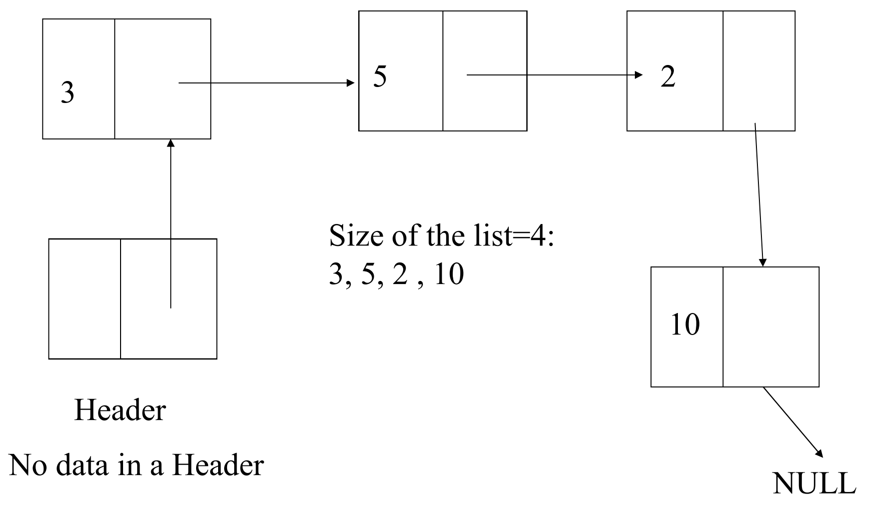
A node in the linked list consists of two elements

Value of the corresponding element (data of node) in the list

Pointer to the next node

The pointer is a null pointer for the last element.

May be a special node at the beginning, known as the header node



No data in a Header

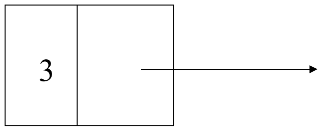
Size of the list=4:
3, 5, 2 , 10

Linked List Type

- Type declaration:

```
typedef struct Node {
  ElementType Element;  /*ElementType is type for element of list */
  struct Node *next;
};
```

- For example: list of integers
  - typedef int ElementType;
- typedef struct Node *PtrToNode;
- typedef PtrToNode **List**;
- typedef PtrToNode **Position**;



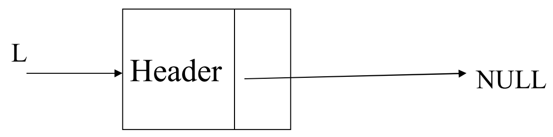
Node

Linked List Notes

- Divide a node into element (or data) and pointer
- Need way to indicate end of list (NULL pointer)
- Need to know where list starts (header pointer)
- Each Node needs pointer to next Node (its link)
- Need way to allocate new Node (use malloc)
- Need way to free memory of Node not needed any more (use free)

Create a empty List with a dummy header

```
void main()
{
    List L = NULL;
    L = malloc (size of (struct Node));
    L->next = NULL;
}
```



Adding a Node at the beginning

Create a new node;

Element in the node has the same value as the new element;

Node pointer points to the first element (non-header)

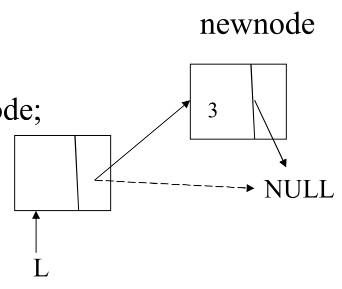
Pointer from the header points to new node;

Create(newnode);

Newnode->next→ L->next->next

L->next→ newnode;

O(1)



Adding a Node at the end

Create a new node;

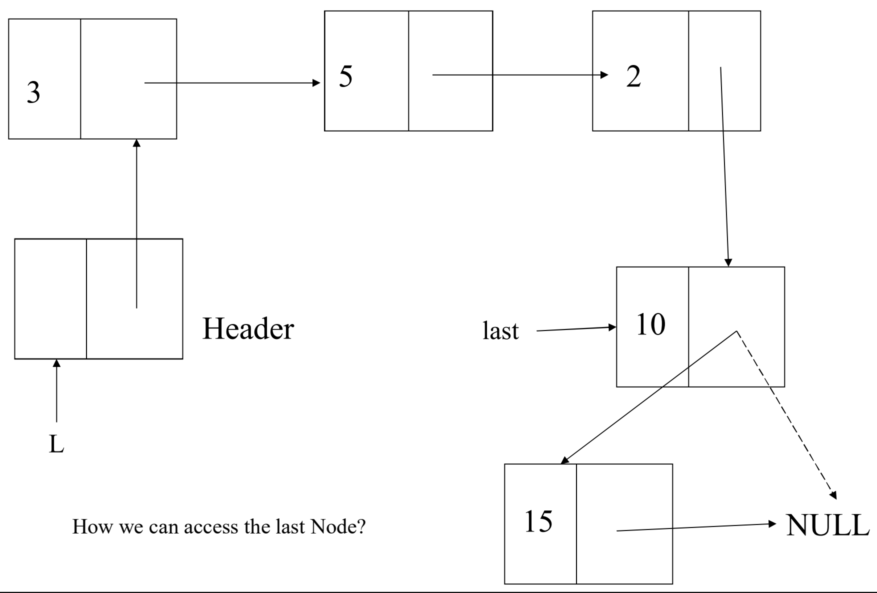
Element in the node has the same value as the new element;

Node pointer **last** points to the last element;

Pointer from the last element points to new node;

```
Create(newnode);
Newnode->next→ last->next
last->next→ newnode;
```

O(n);



How we can access the last Node?

Accessing the last Node

Move from the beginning till you find the correct element:

```
last = L;
while (last->next != NULL) /* stop at */
        last = last->next;
```

Accessing a Node;

Finding a Node with element X:

Move from the beginning till you find the element;

```
Position P;
P = L->next;
While(P != NULL && P->Element != X)
        P = P->next;

// P points to Node with element X
```

Complexity: O(n)

Delete a Node from linked list

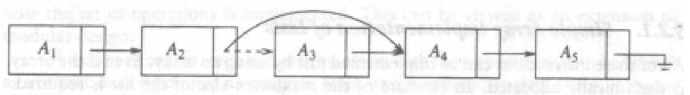


Figure 3.3 Deletion from a linked list

For example, we want to delete A3

```
Position P = L;
while(P->next !=NULL && P->next ->Element !=A3)
        P = P->next;
// P points to A2
Position DeleteNode = P->next;
P->next = DeleteNode ->next;
Free(DeleteNode);
```

Delete at the kth position

```
Access the k-1th node;
Deletenode = (k-1)thnode.next;
(k-1)thnode.next = Deletenode.next;
Delete (Deletenode);
```

Complexity depends on access complexity;

O(1) for deleting first element;

O(1) or O(n) for deleting the last element;

O(k) for any other element;

Insertion after a Node

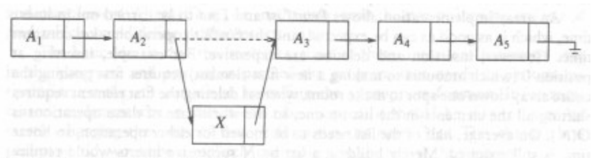


Figure 3.4 Insertion into a linked list

For example, we want to insert after A2

```
Position P = L->next;
while(P!=NULL && P->Element !=A2)
        P=P->next;
//P points to node with A2

TemptNode-> = P->next
P->next = TemptNode
```

Quiz

In our linked list, we have a header node with no information. Is it possible to delete this header?

Advantage and Disadvantage of Linked List

Advantages:

Need not know the maximum number of Nodes

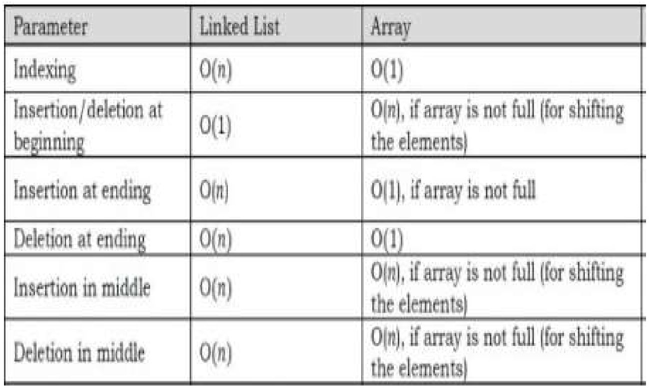
Insertion/Deletion at the beginning is O(1)

Disadvantages:

Access time to individual node is O(n)

Wasted space for pointer pointing to next node

Linked List vs Array



Parameter	Linked List	Array
Indexing	$O(n)$	$O(1)$
Insertion/deletion at beginning	$O(1)$	$O(n)$, if array is not full (for shifting the elements)
Insertion at ending	$O(n)$	$O(1)$, if array is not full
Deletion at ending	$O(n)$	$O(1)$
Insertion in middle	$O(n)$	$O(n)$, if array is not full (for shifting the elements)
Deletion in middle	$O(n)$	$O(n)$, if array is not full (for shifting the elements)

Next week

Doubly Linked List, Circularly Linked List

Stack ADT

Queue ADT

Homework