Linked Lists

abstract data type (adt)

- abstract data types represent abstract information
- it's a specification of a set of data and the set of operations that can be performed on that data
- but also it means to be considered without regard to its implementation
  - e.g. we can implement a generic list using arrays or pointers or something entirely new
  - so is an array an adt? not really since it has no true defined operations
  - suppose we defined a generic list by its interface (how to interact with it and use it)
  - suppose we implemented the "backend" with an array or some other method
  - is this list an adt? yup!

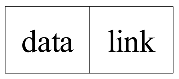
- basic list operations we might consider
  - initialize the list
  - determine whether the list is empty
  - display the list
  - find the length of the list
  - retrieve the information contained in the first element
  - retrieve the information contained in the last element
  - search the list for a given item
  - insert an item in the list
  - delete an item from the list
  - make a copy of the list

motivation for something other than an array for this task

- array size must be known at declaration
- we would like to have a list that items can be added to as we need them

definition

- linked lists are versatile general purpose storage data structures
- a collection of components (nodes)
- each node (except for the last) contains the address of the next node
- so each node has 2 components (a value and a link to the next node)
  - data and link
- address of the first node is stored separately (head or first)

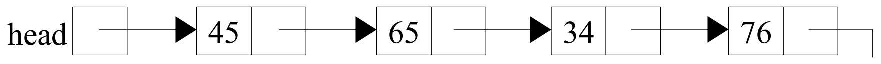


- we define a node as follows:

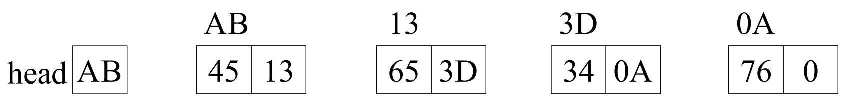
```
struct Node
{
     int data; // some value to store in the node
     Node* link;    // a pointer to a Node
};

Node* head;
```

- e.g.

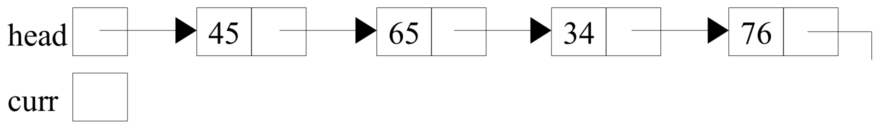


more accurate e.g.

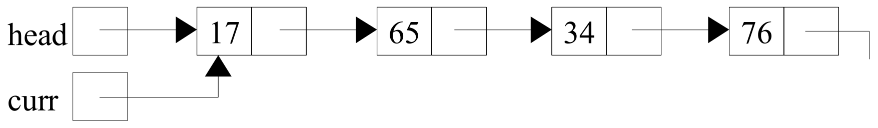


adding a curr node for assistance

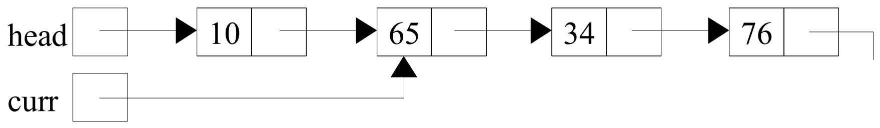
```
    Node* curr;
```



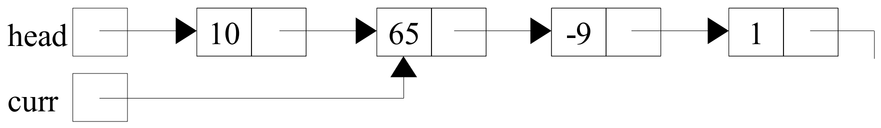
```
head = new Node;
head->data = 17;     // note the arrow as opposed to the "dot" which is for pointers
curr = head;
```



```
curr->data = 10;
curr = curr->link;
```



```
curr->link->data = -9;
curr->link->link->data = 1;
```



traversal

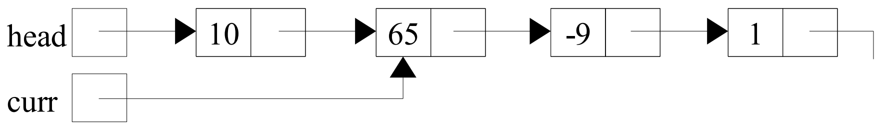
suppose head points to the first node in the list
suppose the link of the last node in the list is null
then:

```
    curr = head;

    while (curr != NULL)
    {
         cout << curr->data << " ";
         curr = curr.Link;
    }
    cout << endl;
```

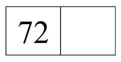
insertion

consider:



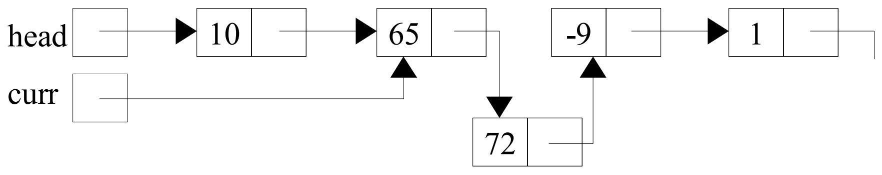
and we wish to insert a new node after curr:

```
Node* newNode = new Node;
newNode->data = 72;
```



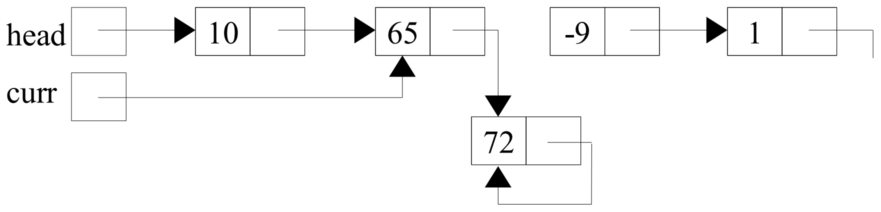
to insert, we can:

```
newNode->link = curr->link;
curr->link = newNode;
```



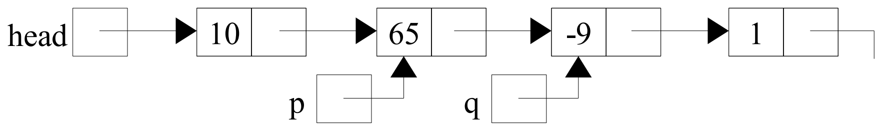
order is critical!

```
curr->link = newNode;
newNode->link = curr->link;
```



oops, now we've lost the end of the list!
to help with this, we can use two separate node references:

```
Node *p, *q;
```



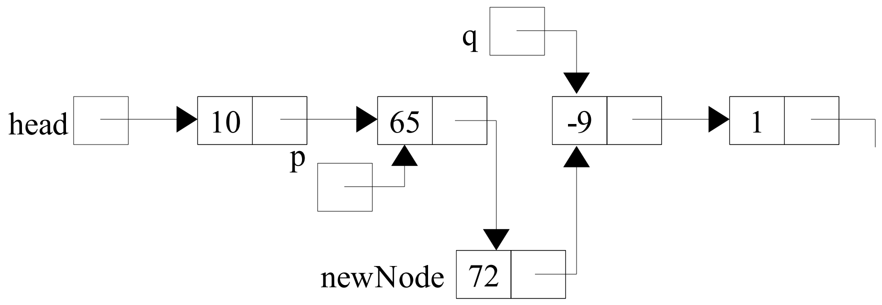
either order works now:

```
p->link = newNode;
newNode->link = q;
```

or

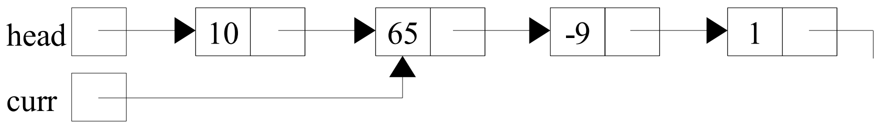
```
newNode->link = q;
p->link = newNode;
```

the end result is the same:



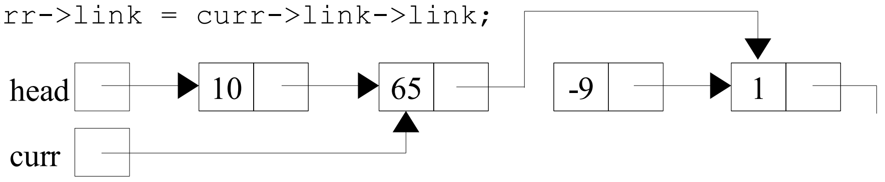
deletion

consider:



now we wish to delete the node with -9:

```
curr->link = curr->link->link;
```



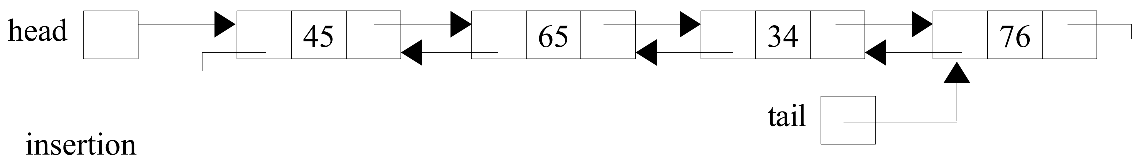
building a linked list
- we need 3 reference nodes to build a linked list
  - head, curr, tail
- for unsorted data, we can build a list in one of two ways
  - forward: add new nodes at the end
  - backward: add new nodes at the beginning
- forward
  - *WORK ON CODE IN CLASS
- backward
  - *WORK ON CODE IN CLASS
  - since a new node is inserted at the beginning, we don't need to know where the tail is

complexity
- insertion: $O(1)$
- deletion: $O(n)$
- search: $O(n)$
- traversal: $O(n)$

ordered linked list
- we insert nodes in sorted order
- a sequential search is performed to find where to insert the new node
- this is like a priority queue
- complexity: $O(n)$

doubly-linked

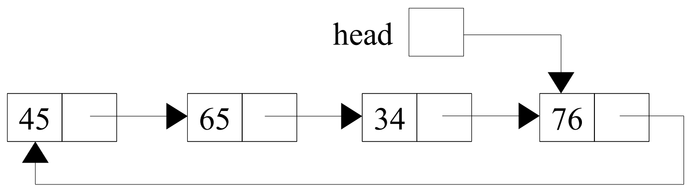
- every node has two links
    - one points to the next node and one points to the previous node
- every node contains the location of the next node (except for the last node)
- every node contains the location of the previous node (except for the first node)
- we can now traverse the list forwards or backwards easily using head, tail, and the links



- insertion
    - several cases exist
        1. insertion in an empty list
        2. insertion at the beginning of a non-empty list
        3. insertion at the end of a non-empty list
        4. insertion somewhere in the middle of a non-empty list
    - we must typically modify two nodes
- deletion
    - again, we must typically modify two nodes
- traversal
    - forward
        - *WORK ON CODE IN CLASS
    - backward
        - *WORK ON CODE IN CLASS

circular-linked

- the last node points to the first node
- typically, we make head point to the last node in the list
    - so `head->link` points to the first node in the list



- traversal
    - *WORK ON CODE IN CLASS

so, is a linked list an adt? yup!

iterators

- allows us to treat an adt (e.g. a list) like an array (using [])

```
int operator[](int i)
{
        return data[i];
}
```

HANDOUT iterator