DATA STRUCTURES AND ALGORITHMS

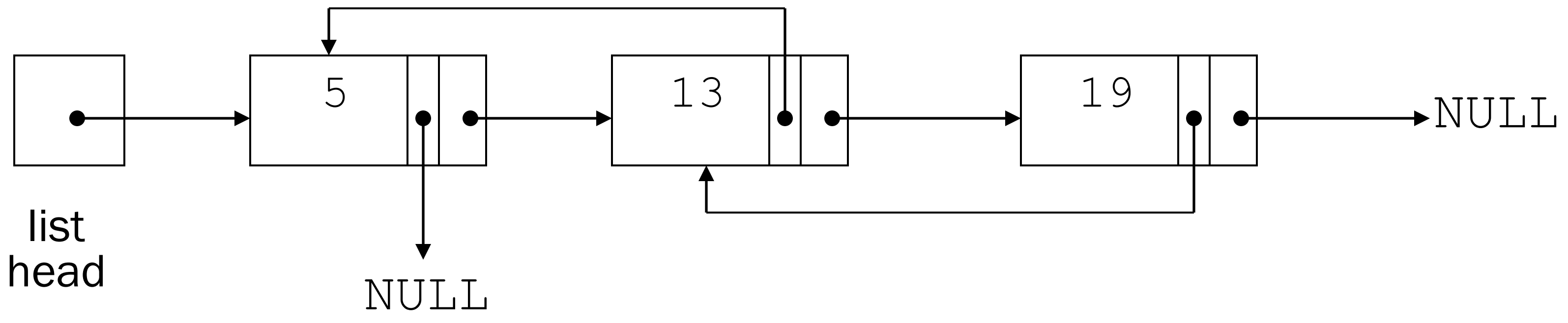
DR SAMABIA TEHSIN

BS (AI)

Variations of the Linked List

Other linked list organizations:

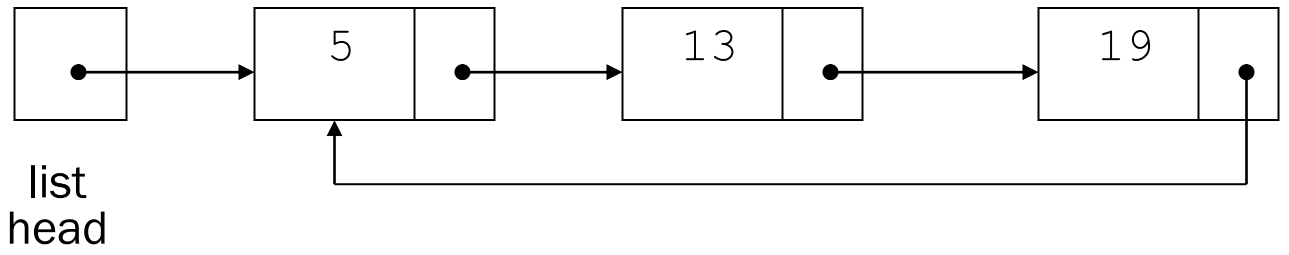
- doubly-linked list: each node contains two pointers: one to the next node in the list, one to the previous node in the list



Variations of the Linked List

Other linked list organizations:

- circular linked list: the last node in the list points back to the first node in the list, not to `NULL`

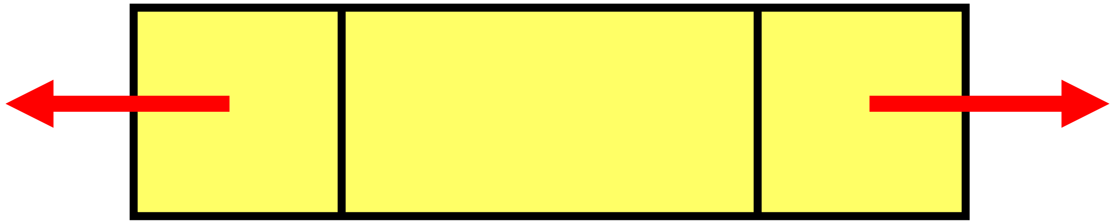


Doubly Linked Lists

Node data

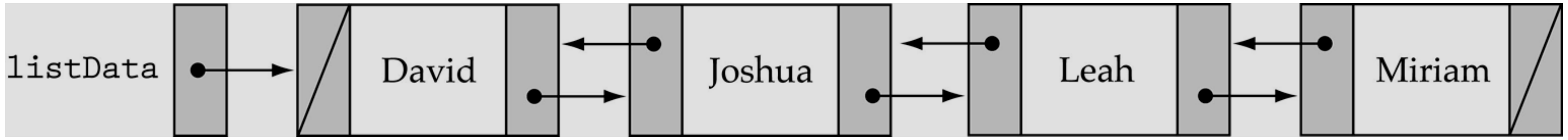
info: the user's data

next, back: the address of the next and previous node in the list



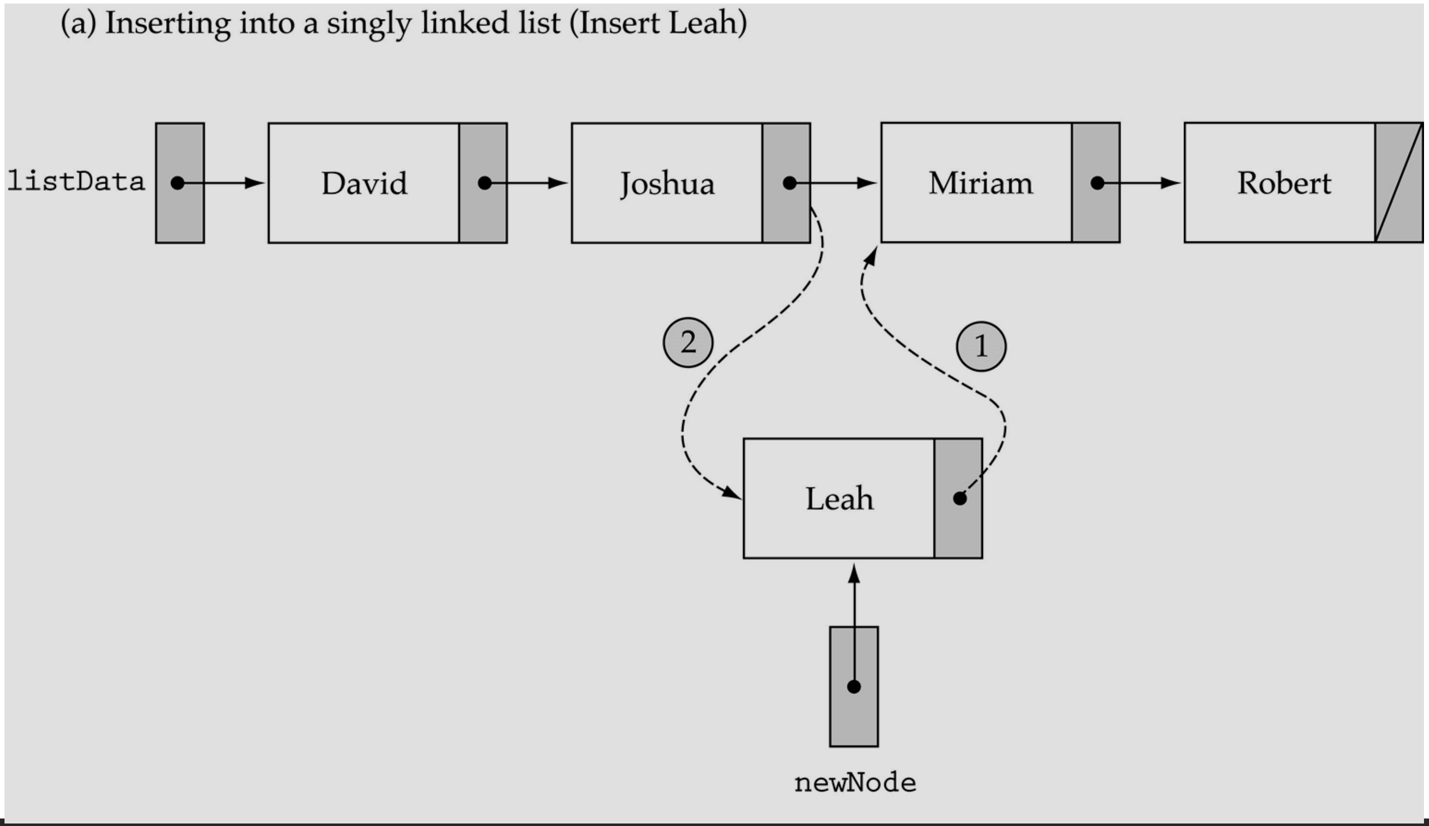
Node data (cont.)

```
template<class ItemType>
struct NodeType {
  ItemType info;
  NodeType<ItemType>* next;
  NodeType<ItemType>* back;
};
```

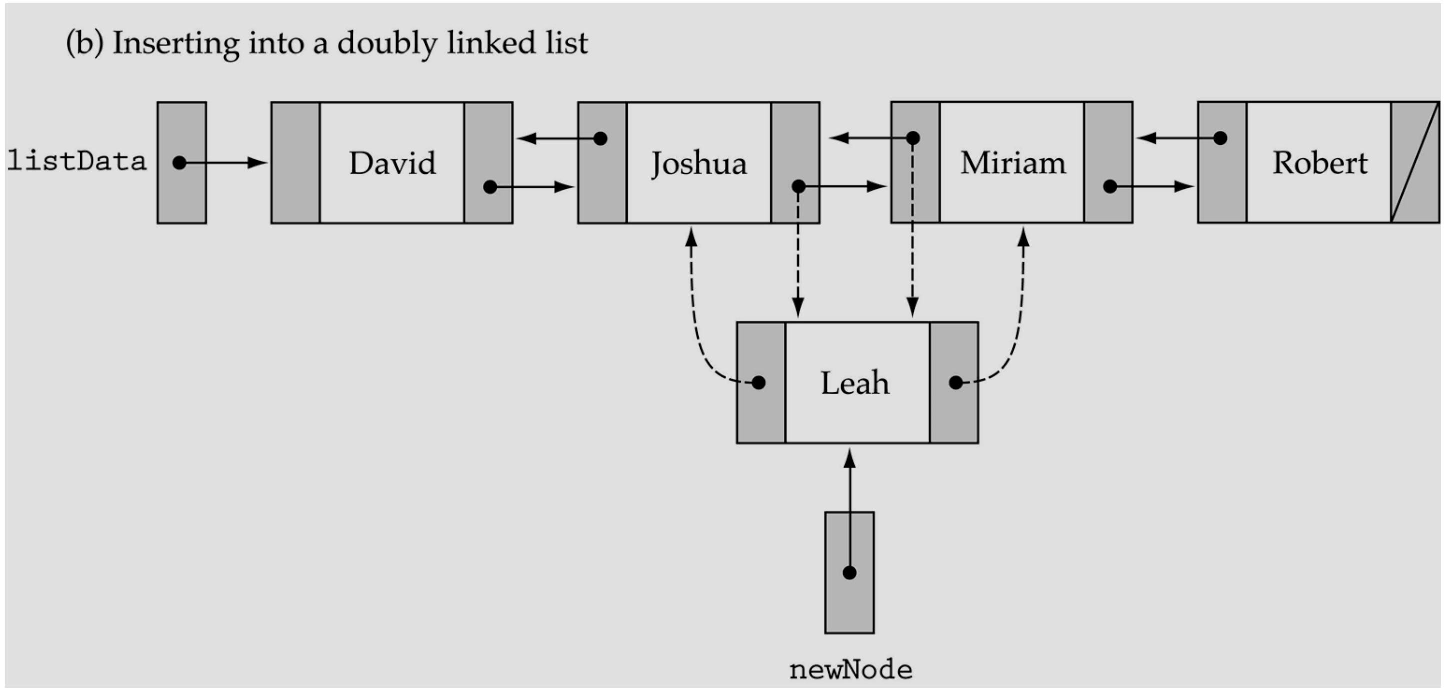


Inserting a List Item

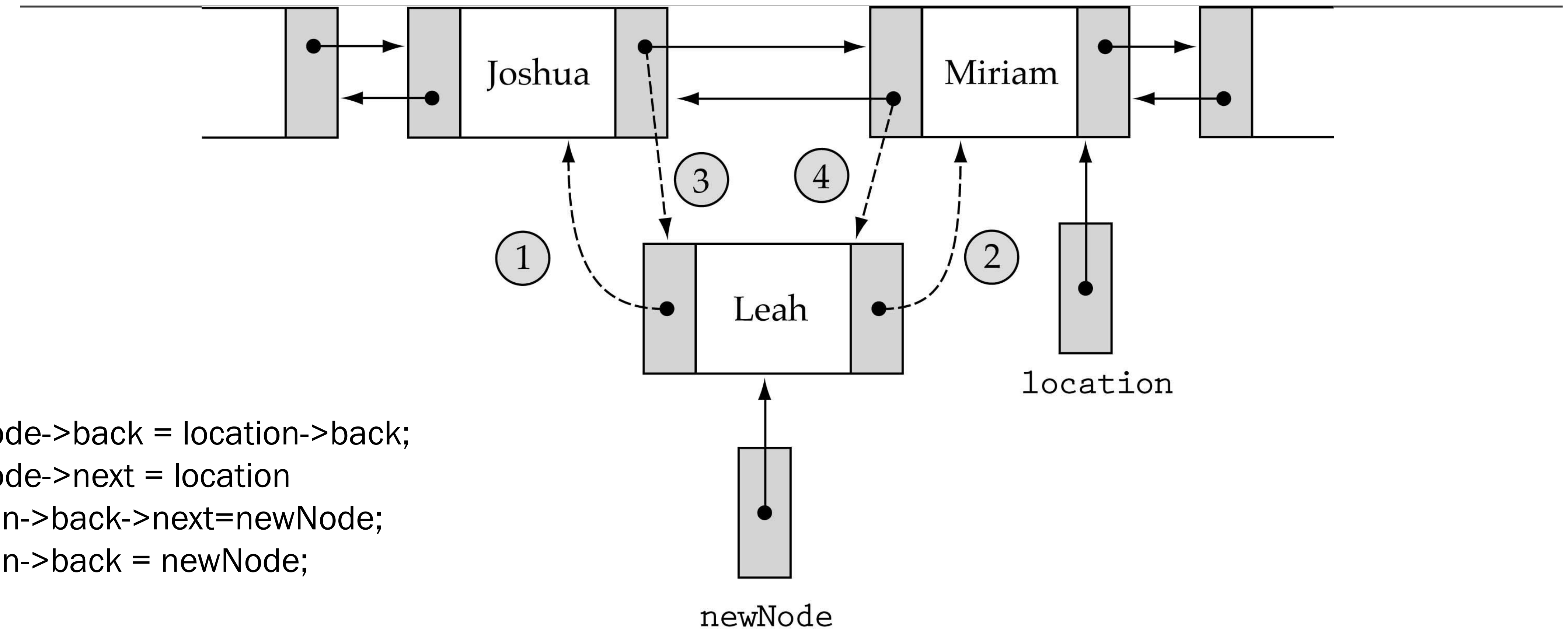
We no longer need to use *prevLocation* (we can get the predecessor of a node using its *back* member)



Inserting a List Item (cont.)



Inserting into a Doubly Linked List



1. newNode->back = location->back;
2. newNode->next = location
3. location->back->next=newNode;
4. location->back = newNode;

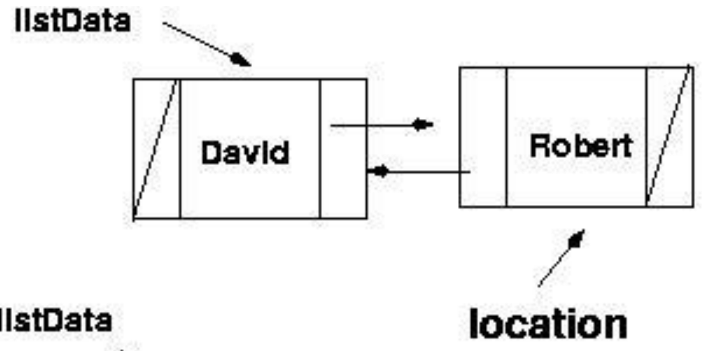
FindItem(listData, item, location, found)

RetrieveItem, **InsertItem**, and **DeleteItem** all require a search !

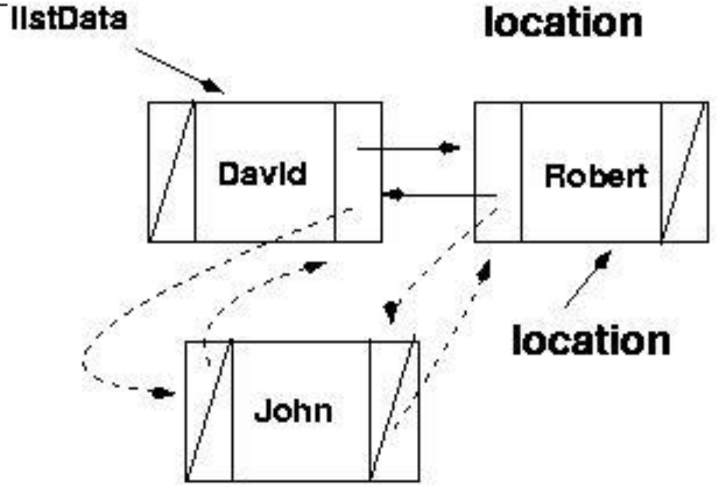
Write a general non-member function **FindItem** that takes *item* as a parameter and returns *location* and *found.*

InsertItem and DeleteItem need *location* (ignore *found*)

RetrieveItem needs *found (*ignores *location)*



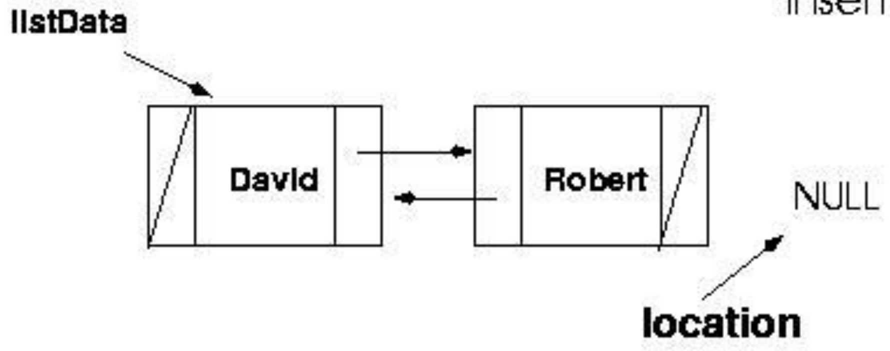
Retrieve: **Robert**



Insert: **John**

(similar for Delete)

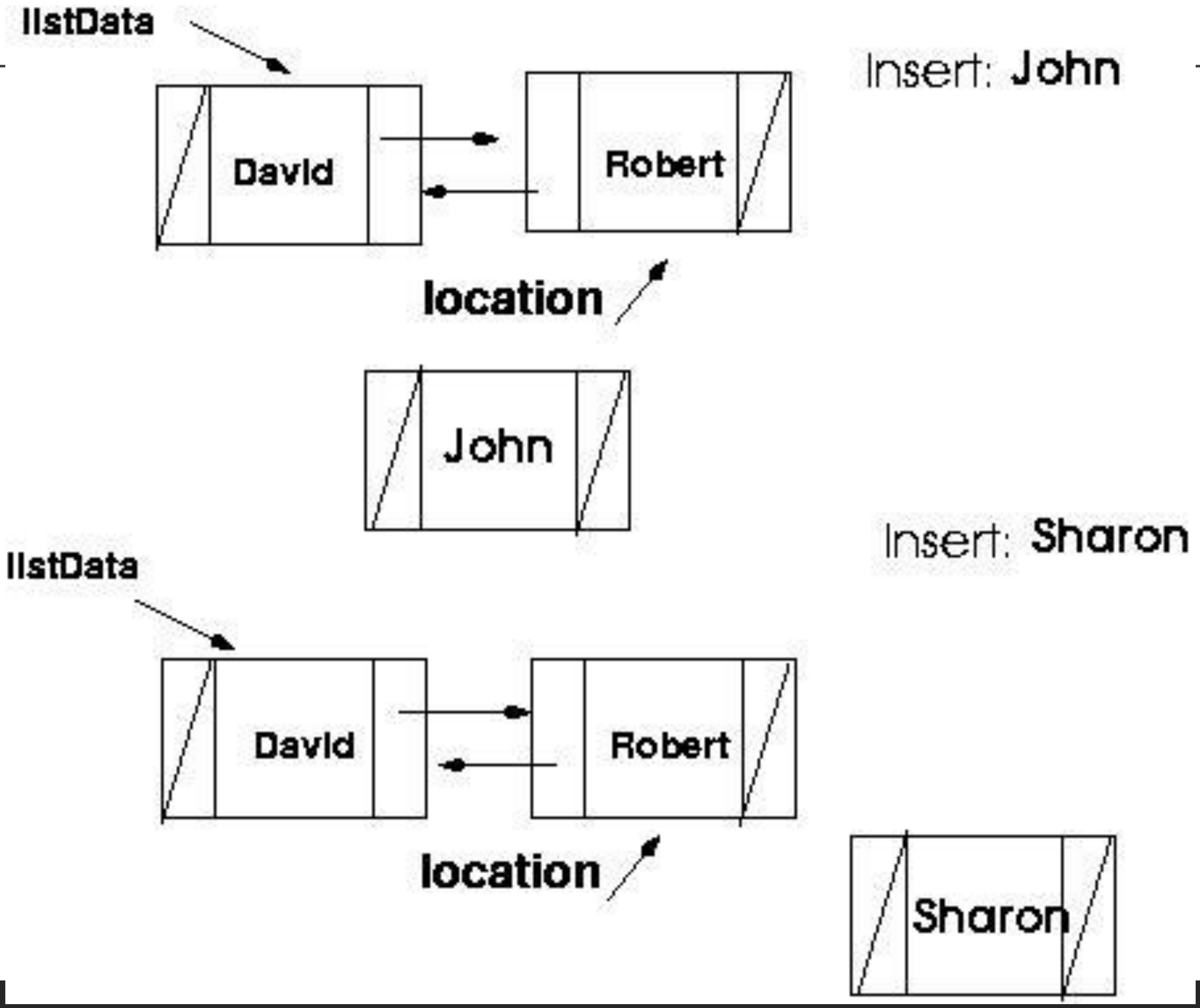
Insert: **Sharon**



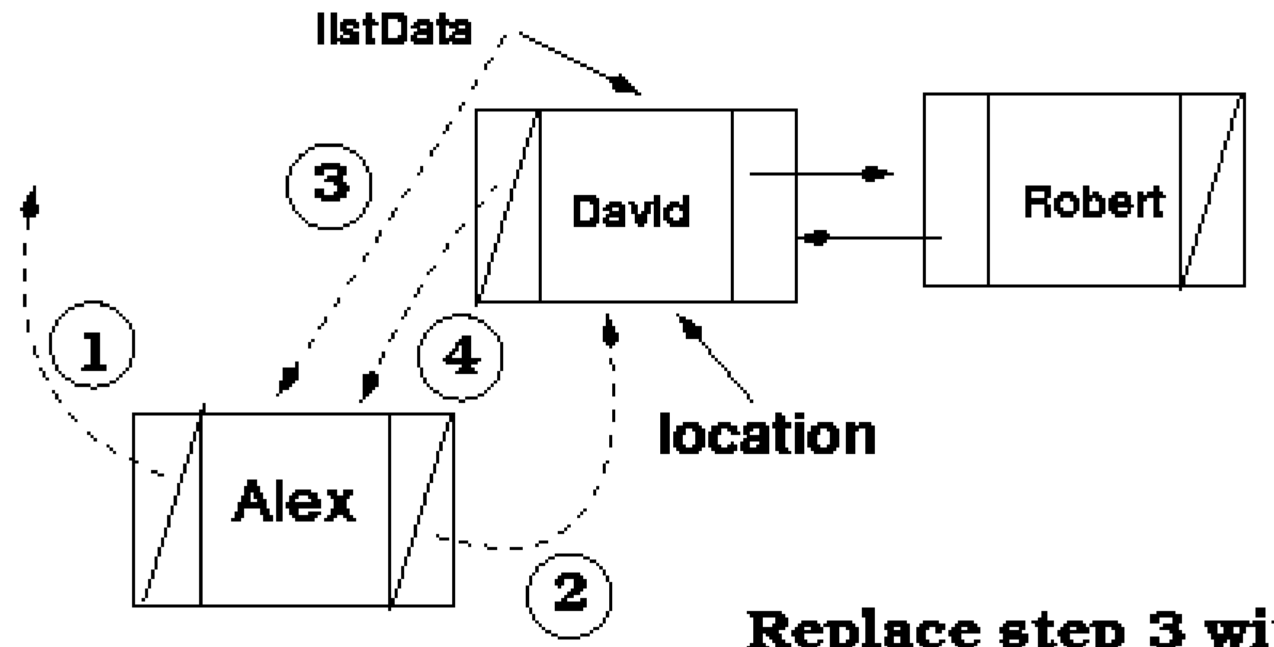
Finding a List Item (cont.)

```
template<class ItemType>
void FindItem(NodeType<ItemType>* listData, ItemType item,
 NodeType<ItemType>* &location, bool &found)
{
// precondition: list is not empty
 bool moreToSearch = true;
  location = listData;
 found = false;

 while( moreToSearch && !found) {
    if(item < location->info)
     moreToSearch = false;
   else if(item == location->info)
     found = true;
   else {
     if(location->next == NULL)
       moreToSearch = false;
     else
       location = location->next;
   }
 }
}
```

How can we distinguish between the following two cases?



Special case: inserting in the beginning



Insert: **Alex**

Replace step 3 with

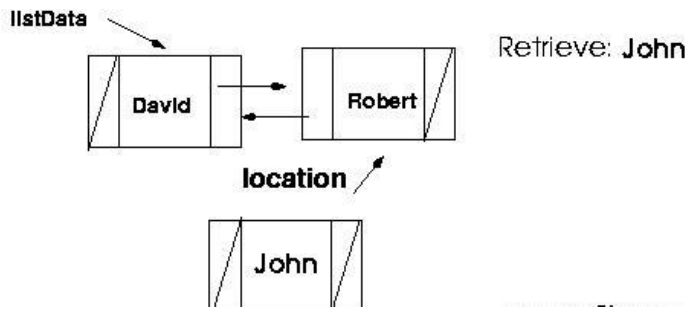
listData = newNode;

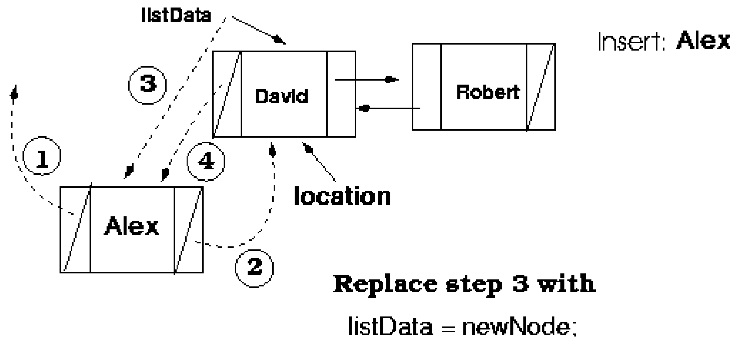
Inserting into a Doubly Linked List

```
template<class ItemType>
void SortedType<ItemType>::InsertItem(ItemType item)
{
```

```
NodeType<ItemType>* newNode;
 NodeType<ItemType>* location;
 bool found;

 newNode = new NodeType<ItemType>;
 newNode->info = item;
 if (listData != NULL) {

   FindItem(listData, item, location, found);

   if (location->info > item) {
     newNode->back = location->back;
     newNode->next = location;
```

```
 if (location != listData) // special case
      (location->back)->next = newNode;
else
   listData = newNode;
 location->back = newNode;
}
```

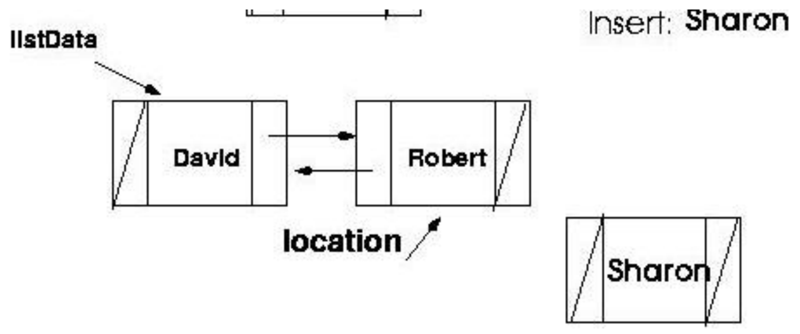




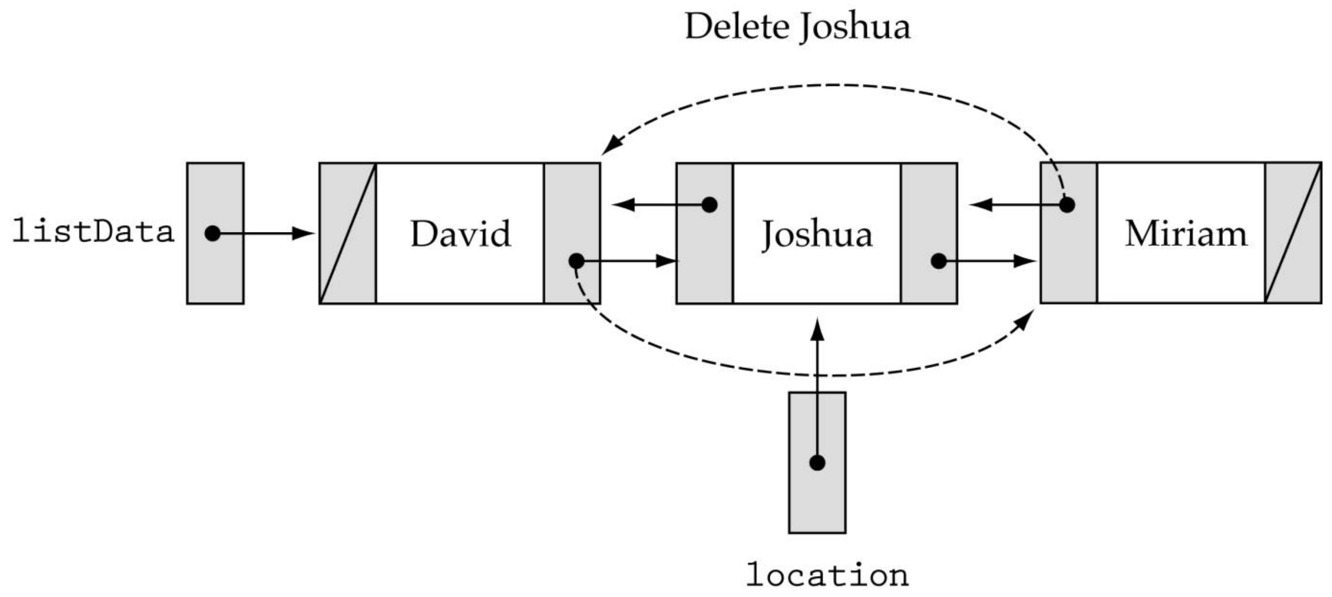
Replace step 3 with
listData = newNode;

Inserting into a Doubly Linked List (cont.)

```
else {                    // insert at the end
    newNode->back = location;
    location->next = newNode;
    newNode->next = NULL;
  }
}
else {                    // insert into an empty list
  listData = newNode;
  newNode->next = NULL;
  newNode->back = NULL;
}
length++;
}
```



Deleting from a Doubly Linked List



Be careful about the end cases!!

Headers and Trailers

Special cases arise when we are dealing with the first or last nodes

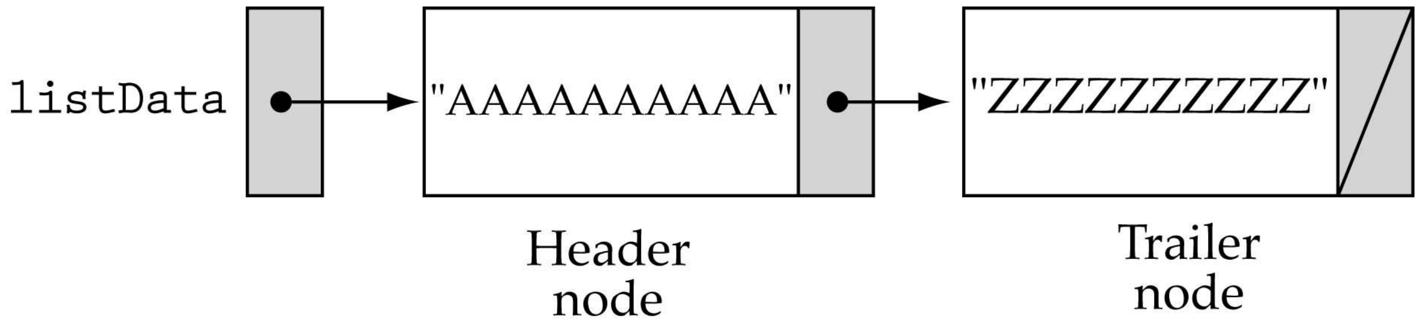
How can we simplify the implementation?

- Idea: make sure that we never insert or delete the ends of the list
- How? Set up dummy nodes with values outside of the range of possible values

Headers and Trailers (cont.)

Header Node: contains a value smaller than any possible list element

Trailer Node: contains a value larger than any possible list element



Credits and Acknowledgements

Lectures by Prof. Yung Yi, KAIST, South Korea.

Lectures by **Selim Aksoy**, Bilkent University, Ankara, Turkey