Linked Lists

Linked List Basics

- Linked lists and arrays are similar since they both store collections of data.
- The *array's* features all follow from its strategy of allocating the memory for all its elements in one block of memory.
- *Linked lists* use an entirely different strategy: linked lists allocate memory for each element separately and only when necessary.

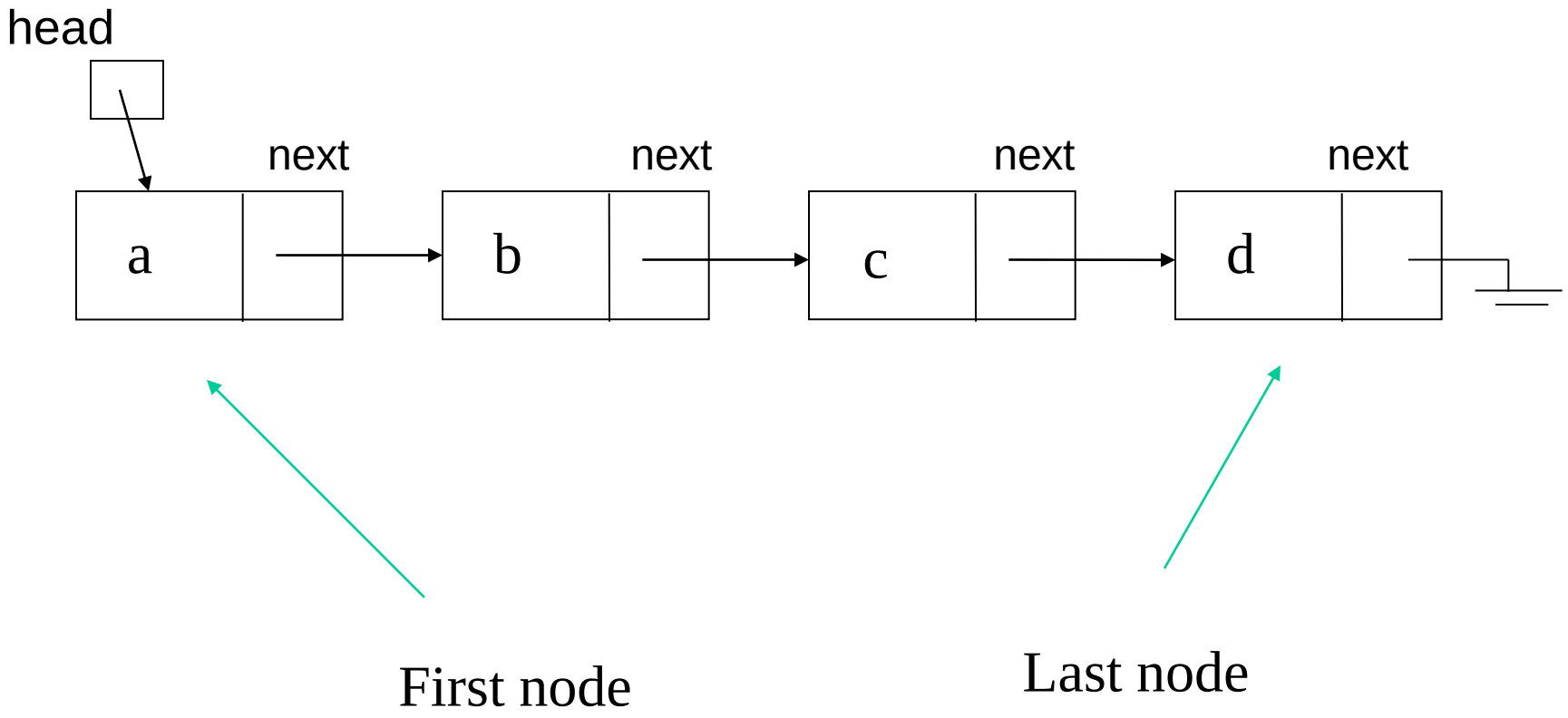
Disadvantages of Arrays

1. **The size of the array is fixed**.
   - In case of **dynamically resizing** the array from size S to 2S, we need 3S units of available memory.
   - Programmers allocate arrays which seem **"large enough**" This strategy has two disadvantages: (a) most of the time there are just 20% or 30% elements in the array and 70% of the space in the array really is wasted. (b) If the program ever needs to process more than the declared size, the code breaks.
2. **Inserting (and deleting)** elements into the middle of the array is potentially expensive because existing elements need to be shifted over to make room

Linked lists

- Linked lists are appropriate when the number of data elements to be represented in the data structure at once is unpredictable.
- Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
- Each node does not necessarily follow the previous one physically in the memory.
- Linked lists can be maintained in sorted order by inserting or deleting an element at the proper point in the list.

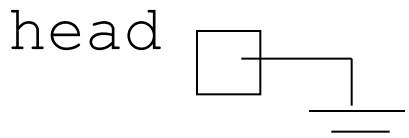
Singly Linked Lists



Empty List

- Empty Linked list is a single pointer having the value of NULL.

```
head = NULL;
```



Basic Ideas

- Let's assume that the node is given by the following type declaration:

```
struct Node {
   Object element;
   Node *next;
};
```

Basic Linked List Operations

- List Traversal
- Searching a node
- Insert a node
- Delete a node

Traversing a linked list

```
Node *pWalker;
int count = 0;

cout <<"List contains:\n";

for (pWalker=pHead; pWalker!=NULL;
     pWalker = pWalker->next)
{
   count ++;
   cout << pWalker->element << endl;
}
```

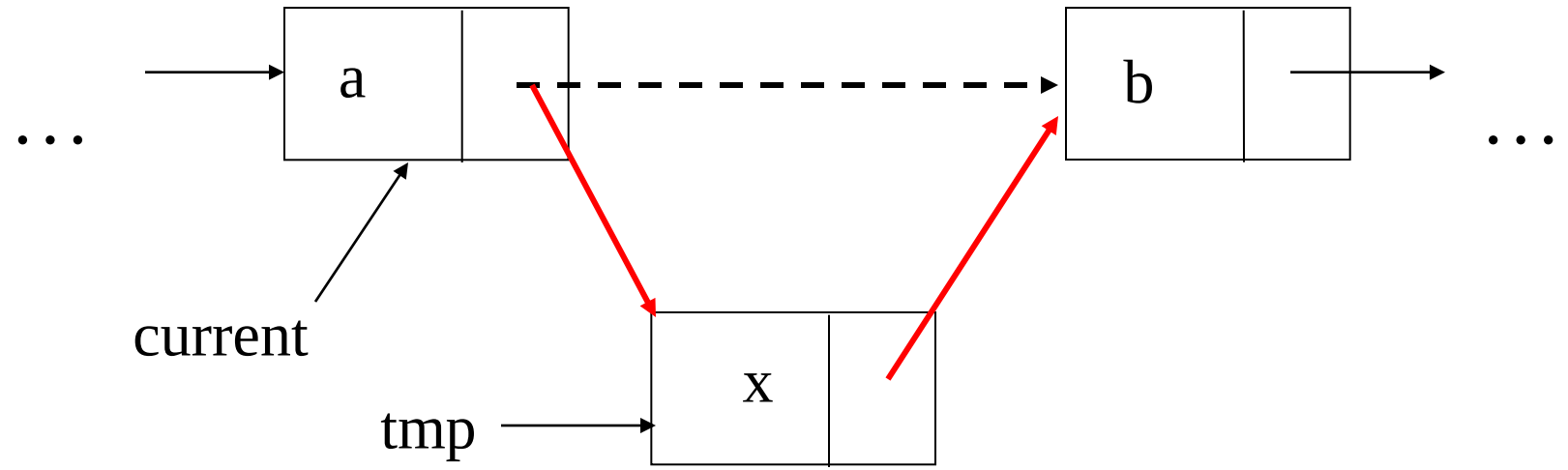
Searching a node in a linked list

```
pCur = pHead;

// Search until target is found or we reach
// the end of list
while (pCur != NULL &&
       pCur->element != target)
{
   pCur = pCur->next;
}

//Determine if target is found
if (pCur) found = 1;
else found = 0;
```

Insertion in a linked list

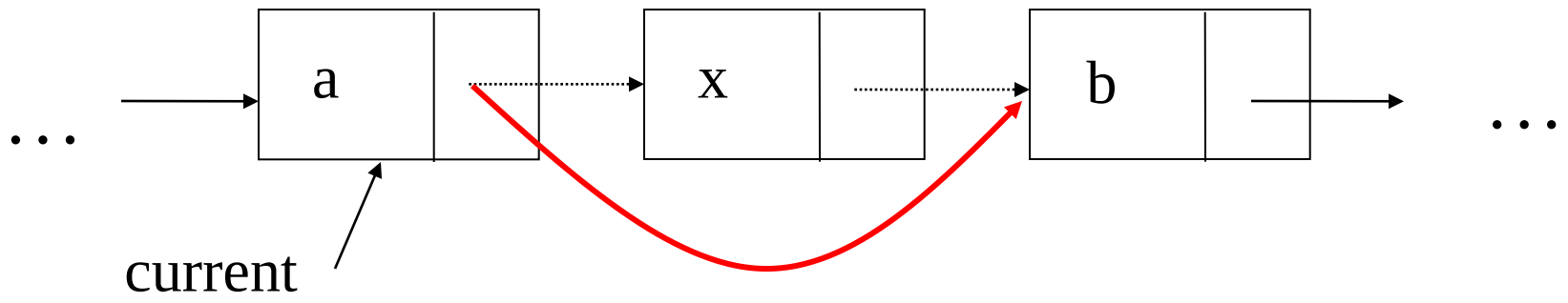



```
tmp = new Node;
tmp->element = x;
tmp->next = current->next;
current->next = tmp;
```

Or simply (if Node has a constructor initializing its members):

```
current->next = new Node(x,current->next);
```

Deletion from a linked list




```
Node *deletedNode = current->next;
current->next = current->next->next;
delete deletedNode;
```

Special Cases (1)

- Inserting before the first node (or to an empty list):

```
tmp = new Node;
tmp->element = x;
if (current == NULL){
   tmp->next = head;
   head = tmp;
}
else {  // Adding in middle or at end
   tmp->next = curent->next;
   current->next = tmp;
}
```

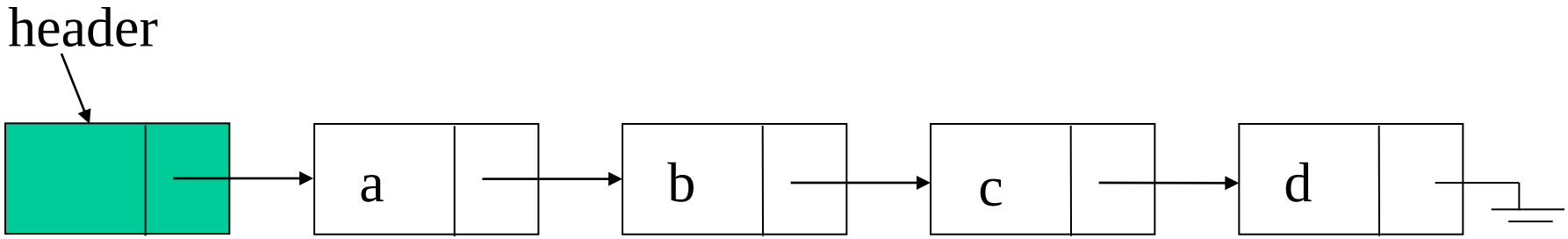
Special Cases (2)

```
Node *deletedNode;
if (current == NULL){
   // Deleting first node
   deletedNode = head;
   head = head ->next;
}
else{
   // Deleting other nodes
   deletedNode = current->next;
   current->next = deletedNode ->next;
}
delete deletedNode;
```

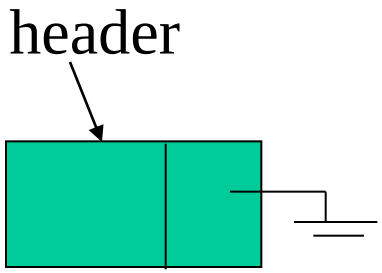
Header Nodes

- One problem with the basic description: it assumes that whenever an item x is removed (or inserted) some previous item is always present.
- Consequently removal of the first item and inserting an item as a new first node become special cases to consider.
- In order to avoid dealing with special cases: introduce a **header node (dummy node)**.
- A header node is an extra node in the list that holds no data but serves to satisfy the requirement that every node has a previous node.

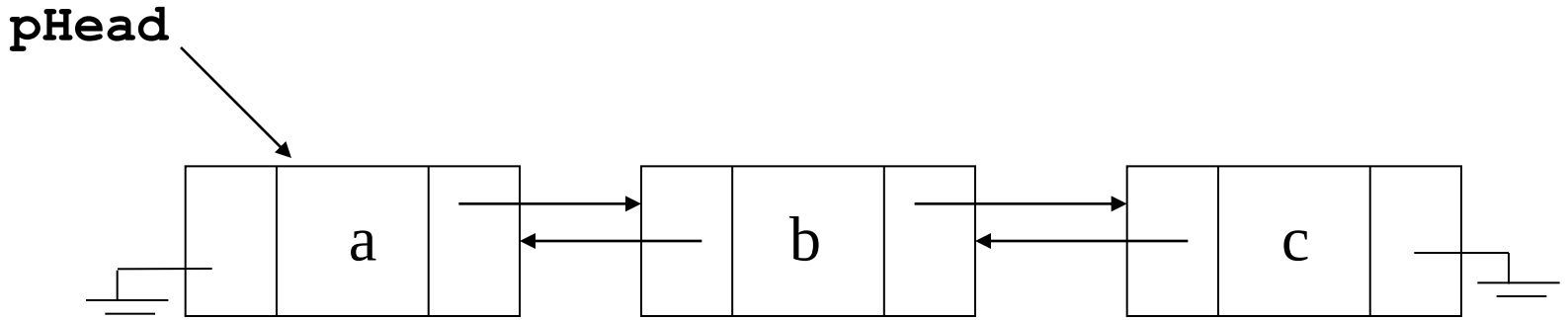
List with a header node



Empty List



Doubly Linked Lists



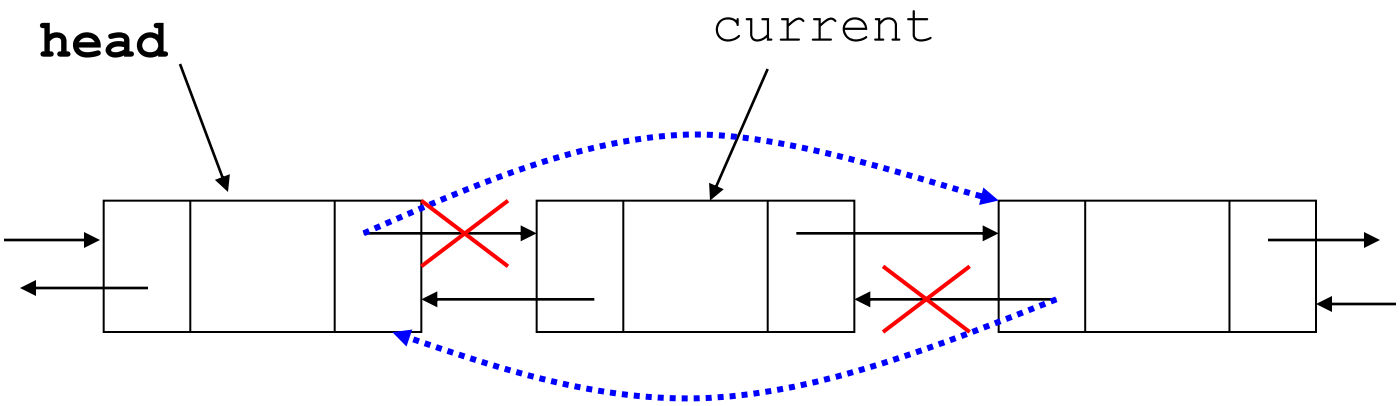
Advantages:

- Convenient to traverse the list backwards.
- Simplifies insertion and deletion because you no longer have to refer to the previous node.

Disadvantage:

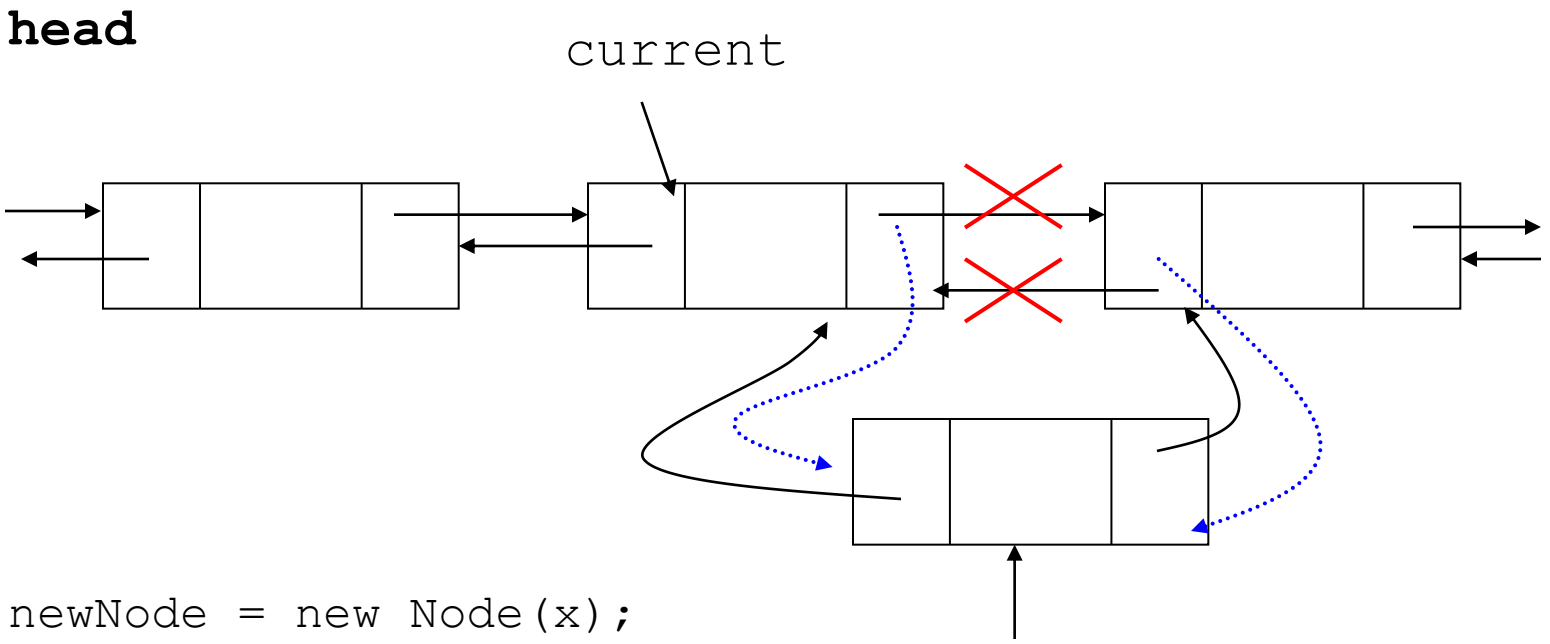
- Increase in space requirements.

Deletion




```
oldNode = current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
delete oldNode;
current = head;
```

Insertion




```
newNode = new Node(x);
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```

The List ADT in C++

- A list is implemented as three separate classes:

1. List itself (`List`)
2. Node (`ListNode`)
3. Position iterator(`ListItr`)

Linked List Node

```
template <class Object>
class List;     // Incomplete declaration.

template <class Object>
class ListItr;     // Incomplete declaration.

template <class Object>
class ListNode
{
   ListNode(const Object & theElement = Object(),
            ListNode * n = NULL )
     : element( theElement ), next( n ) { }
   Object   element;
   ListNode *next;

   friend class List<Object>;
   friend class ListItr<Object>;
};
```

Iterator class for linked lists

```
template <class Object>
class ListItr
{
 public:
   ListItr( ) : current( NULL ) { }
   bool isValid( ) const
   { return current != NULL; }
   void advance( )
   { if(isValid( ) ) current = current->next; }
   const Object & retrieve( ) const
   { if( !isValid( ) ) throw BadIterator( );
     return current->element; }

 private:
   ListNode<Object> *current;    // Current position

   ListItr( ListNode<Object> *theNode )
     : current( theNode ) { }

   friend class List<Object>; //Grant access to constructor
};
```

`List` Class Interface

```
template <class Object>
class List
{
  public:
    List( );
    List( const List & rhs );
    ~List( );

    bool isEmpty( ) const;
    void makeEmpty( );
    ListItr<Object> zeroth( ) const;
    ListItr<Object> first( ) const;
    void insert(const Object &x, const  ListItr<Object> & p);
    ListItr<Object> find( const Object & x ) const;
    ListItr<Object> findPrevious( const Object & x ) const;
    void remove( const Object & x );

    const List & operator=( const List & rhs );

  private:
    ListNode<Object> *header;
};
```

Some `List` one-liners

```
/* Construct the list */
template <class Object>
List<Object>::List( )
{
   header = new ListNode<Object>;
}
/* Test if the list is logically empty.
 * return true if empty, false otherwise.*/
template <class Object>
bool List<Object>::isEmpty( ) const
{
   return header->next == NULL;
}
```

```
/* Return an iterator representing the header node. */
template <class Object>
ListItr<Object> List<Object>::zeroth( ) const
{
   return ListItr<Object>( header );
}
/* Return an iterator representing the first node in
  the list. This operation is valid for empty lists.*/
template <class Object>
ListItr<Object> List<Object>::first( ) const
{
    return ListItr<Object>( header->next );
}
```

Other `List` methods

```
template <class Object>
ListItr<Object> List<Object>::find( const Object & x )
   const
{
  ListNode<Object> *itr = header->next;

  while(itr != NULL && itr->element != x)
     itr = itr->next;

  return ListItr<Object>( itr );
}
```

Deletion routine

```
/* Remove the first occurrence of an item x. */
template <class Object>
void List<Object>::remove( const Object & x )
{
   ListItr<Object> p = findPrevious( x );

   if( p.current->next != NULL )
   {
     ListNode<Object> *oldNode = p.current->next;
     p.current->next = p.current->next->next;
     delete oldNode;
   }
}
```

Finding the previous node

```
/* Return iterator prior to the first node containing an
   item x. */
template <class Object>
ListItr<Object> List<Object>::findPrevious( const Object
   & x ) const
{
    ListNode<Object> *itr = header;

    while(itr->next!=NULL && itr->next->element!=x)
        itr = itr->next;

    return ListItr<Object>( itr );
}
```

Insertion routine

```
/* Insert item x after p  */
template <class Object>
void List<Object>::insert(const Object & x,
                          const ListItr<Object> & p )
{
  if( p.current != NULL )
     p.current->next = new ListNode<Object>(x,
                                  p.current->next );
}
```

Memory Reclamation

```
/* Make the list logically empty        */
template <class Object>
void List<Object>::makeEmpty( )
{
   while( !isEmpty( ) )
      remove( first( ).retrieve( ) );
}
/* Destructor */
template <class Object>
List<Object>:: ~List( )
{
   makeEmpty( );
   delete header;
}
```

operator =

```
/* Deep copy of linked lists.   */
template <class Object>
const List<Object> & List<Object>::operator=( const
   List<Object> & rhs )
{
  ListItr<Object> ritr = rhs.first( );
  ListItr<Object> itr = zeroth( );

  if( this != &rhs )
  {
     makeEmpty( );
     for( ; ritr.isValid(); ritr.advance(), itr.advance())
        insert( ritr.retrieve( ), itr );
  }
  return *this;
}
```

Copy constructor

```
/* copy constructor.   */
template <class Object>
List<Object>::List( const List<Object> & rhs )
{
  header = new ListNode<Object>;
  *this = rhs;       // operator= is used here
}
```

Testing Linked List Interface

```
#include <iostream.h>
#include "LinkedList.h"

// Simple print method

template <class Object>
void printList( const List<Object> & theList )
{
  if( theList.isEmpty( ) )
    cout << "Empty list" << endl;
  else {
    ListItr<Object> itr = theList.first( );
    for( ; itr.isValid( ); itr.advance( ) )
       cout << itr.retrieve( ) << " ";
  }
  cout << endl;
}
```

```
int main( )
{  List<int>    theList;
   ListItr<int> theItr = theList.zeroth( );
   int i;

   printList( theList );
   for( i = 0; i < 10; i++ )
   {  theList.insert( i, theItr );
      printList( theList );
      theItr.advance( );
   }
   for( i = 0; i < 10; i += 2 )
      theList.remove( i );

   for( i = 0; i < 10; i++ )
      if((i % 2 == 0)!=(theList.find(i).isValid()))
         cout << "Find fails!" << endl;

    cout << "Finished deletions" << endl;
    printList( theList );

    List<int> list2;
    list2 = theList;
    printList( list2 );
    return 0;
}
```

Comparing Array-Based and Pointer-Based Implementations

- Size
  - Increasing the size of a resizable array can waste storage and time
- Storage requirements
  - Array-based implementations require less memory than a pointer-based ones

Comparing Array-Based and Pointer-Based Implementations

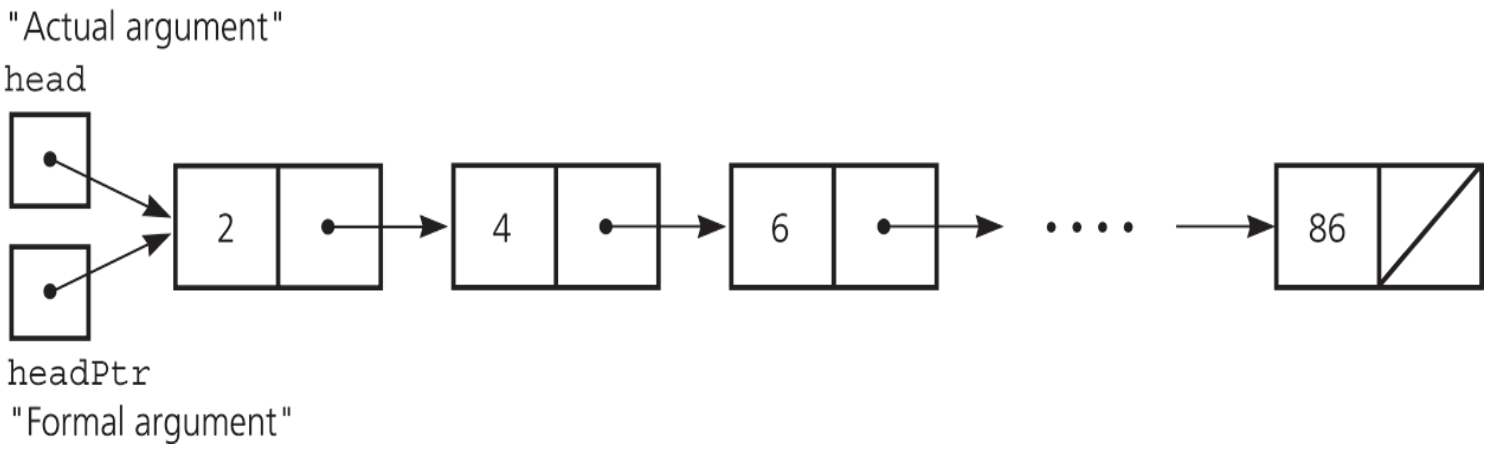
- Access time
  - Array-based: constant access time
  - Pointer-based: the time to access the i^{th} node depends on i
- Insertion and deletions
  - Array-based: require shifting of data
  - Pointer-based: require a list traversal

Saving and Restoring a Linked List by Using a File

- Use an external file to preserve the list
- Do not write pointers to a file, only data
- Recreate the list from the file by placing each item at the end of the list
  - Use a tail pointer to facilitate adding nodes to the end of the list
  - Treat the first insertion as a special case by setting the tail to head

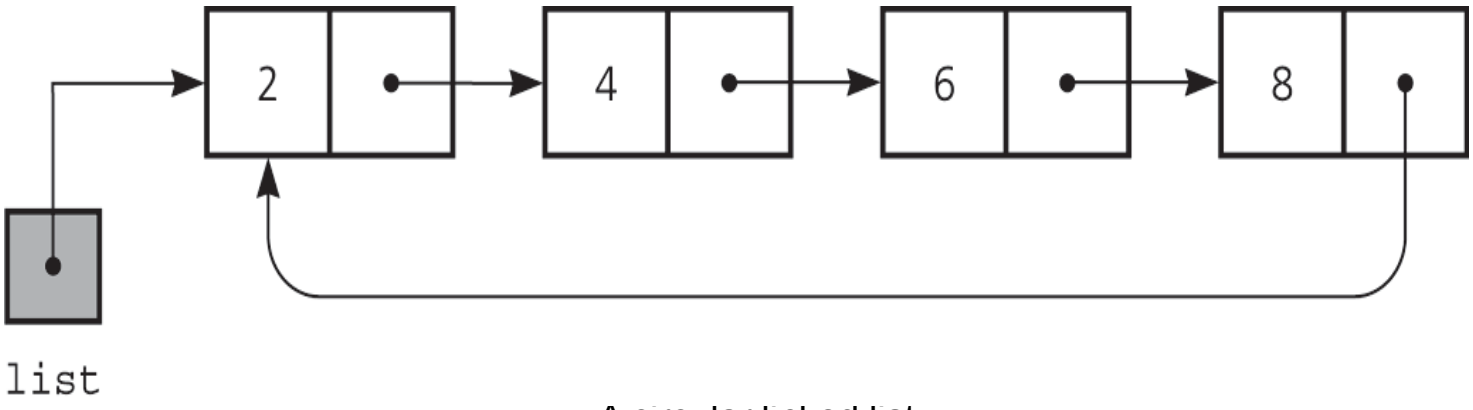
Passing a Linked List to a Function

- A function with access to a linked list's `head` pointer has access to the entire list
- Pass the head pointer to a function as a reference argument



Circular Linked Lists

- Last node references the first node
- Every node has a successor
- No node in a circular linked list contains *NULL*

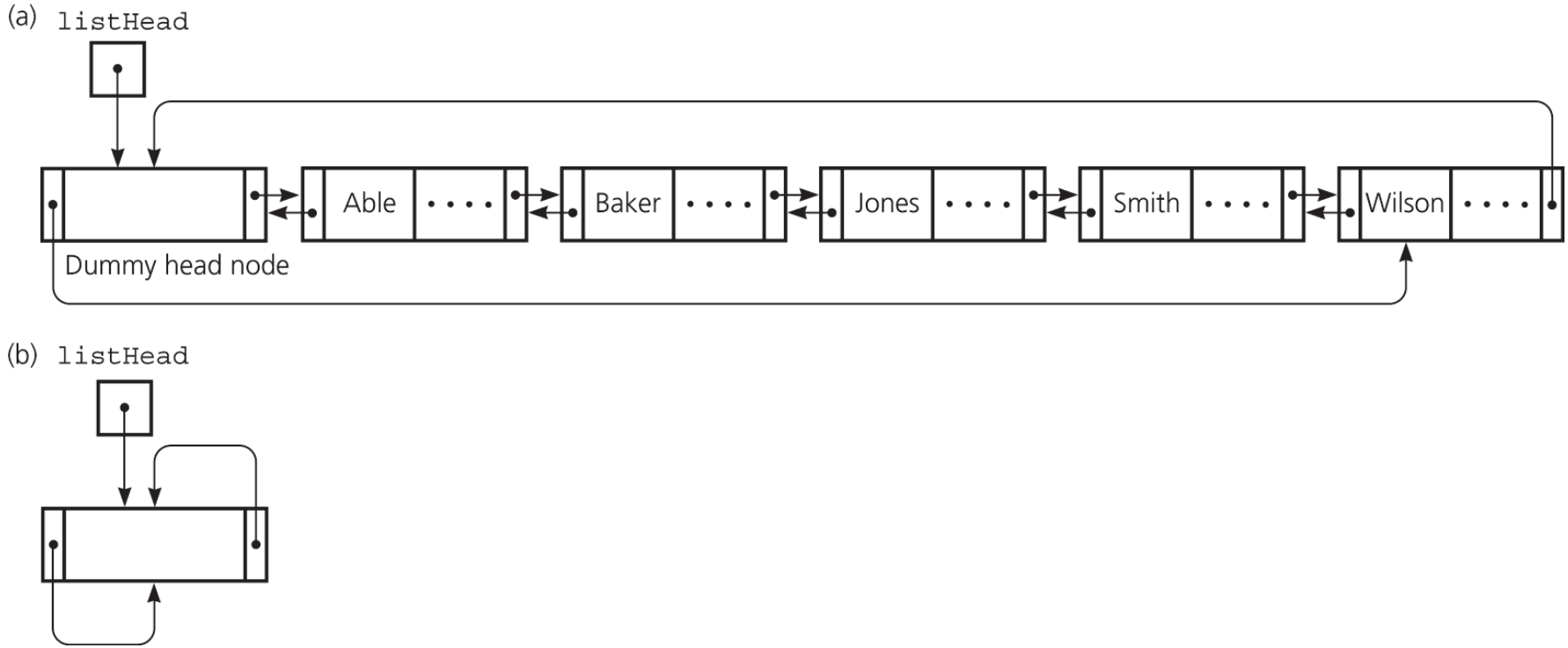


A circular linked list

Circular Doubly Linked Lists

- Circular doubly linked list
  - `prev` pointer of the dummy head node points to the last node
  - `next` reference of the last node points to the dummy head node
  - No special cases for insertions and deletions

Circular Doubly Linked Lists



(a) A circular doubly linked list with a dummy head node

(b) An empty list with a dummy head node

Processing Linked Lists Recursively

- Recursive strategy to display a list
  - Write the first node of the list
  - Write the list minus its first node
- Recursive strategies to display a list backward
  - `writeListBackward` strategy
    - Write the last node of the list
    - Write the list minus its last node backward

Processing Linked Lists Recursively

- `writeListBackward2` strategy
  - Write the list minus its first node backward
  - Write the first node of the list
- Recursive view of a sorted linked list
  - The linked list to which `head` points is a sorted list if
    - `head` is *`NULL`* *or*
    - `head->next` is *`NULL`* *or*
    - `head->item < head->next->item`, and `head->next` points to a sorted linked list