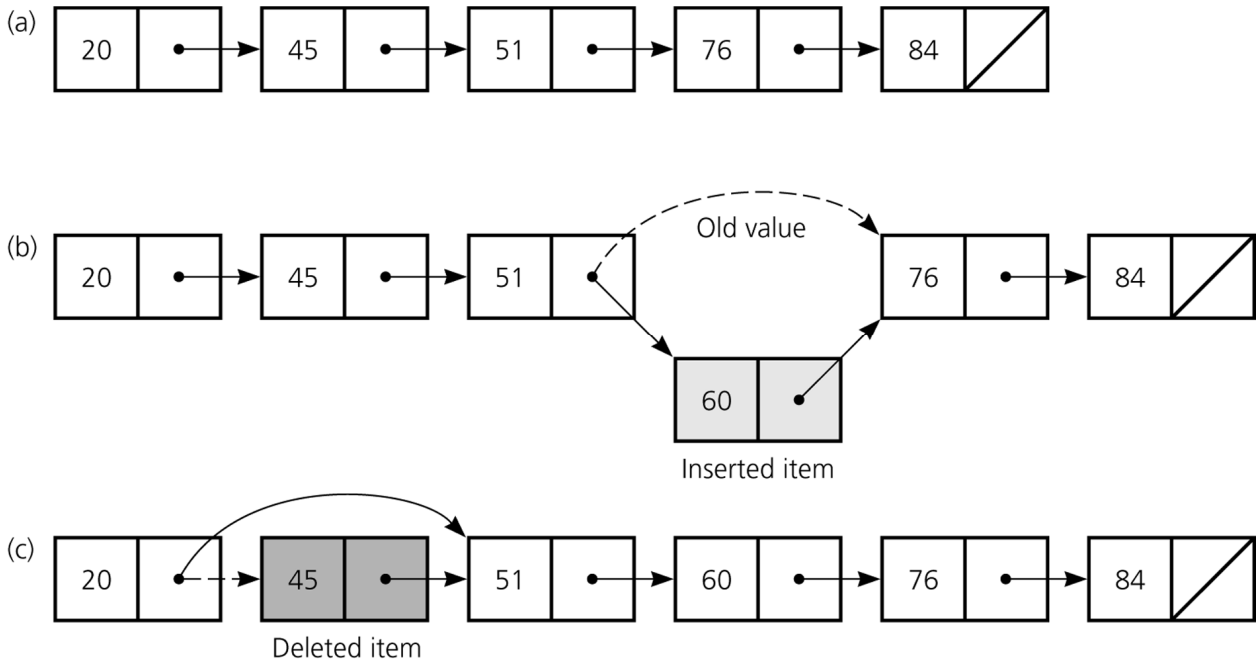
C++ Pointers and Lists

- Pointers can be used to construct and process Linked Lists of values.
- A NODE is a data structure (a struct record) that contains at least two fields: one or more data values and one or more pointer values.
- A linked list is a collection of Nodes where the pointer field in each Node is used to link the Node to the next Node in the list.
- It is possible to insert new Nodes into the list, or remove Nodes from the list, by simply manipulating the Node pointers.

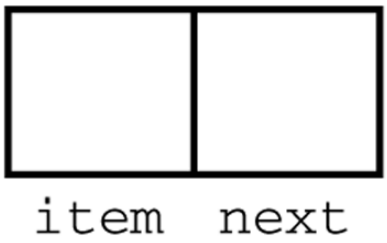


(a) A linked list of integers; (b) insertion; (c) deletion

```
// ptrtest.cpp: connect 2 Nodes together into a short list

#include <iostream>
using namespace std;

// ------------------------------------------------------
struct Node;                // Prototype Declaration

typedef Node *  ptrType;    // Pointer typedef


struct Node {               // The Node structure
   int      m_SomeInfo;
   ptrType  m_Next;         // Linked List pointer
};

// ------------------------------------------------------

int main()
{
   // Create a couple of Nodes
   Node NodeA;
   Node NodeB;

   // Initialize the Nodes.  Link NodeA to NodeB.
   NodeA.m_SomeInfo = 1;
   NodeA.m_Next = & NodeB;

   NodeB.m_SomeInfo = 2;
   NodeB.m_Next = NULL;


   // Now, use a pointer to traverse the list of Nodes
   ptrType p;

   p = &NodeA;

   while(p != NULL)
   {
      cout << "Node Info = " << p->m_SomeInfo << endl;

      p = p->m_Next;    // Move p to next Node in List
   }

   cout << "End of Node List" << endl;

   return(0);
}
```

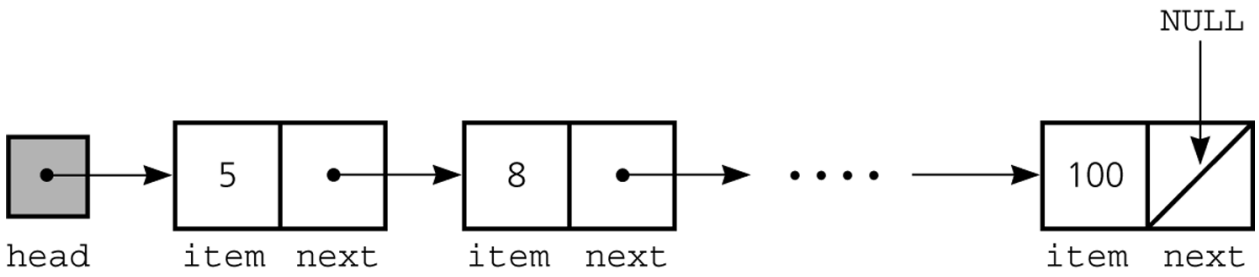
```
// ptrlist.cpp: connect several dynamically-allocated Nodes
//              into a linked list, with Head pointer
// This example links Nodes in reverse order
//              (insert at front of list)

#include <iostream>
using namespace std;

// ------------------------------------------------------
struct Node {               // The Node structure
   int      m_SomeInfo;
   Node    *m_Next;         // Linked List pointer
};

// ------------------------------------------------------
int main()
{
   // Create a dynamic linked list
   Node *pHead;
   Node *p;

   pHead = new Node;        // Create initial Node

   pHead->m_SomeInfo = 1;
   pHead->m_Next = NULL;    // end of list Node has NULL Next ptr

   for(int i=2; i<=10; ++i)
   {
      p = new Node;         // Create another Node

      p->m_SomeInfo = i;

      p->m_Next = pHead;    // Link new Node to front of list

      pHead = p;            // Update Head ptr: Make p the front Node
   }

   // Now, traverse and print the list of Nodes
   p = pHead;               // Start with first (Head) Node

   while(p != NULL)         // Traverse in order until end-of-list
   {
      cout << "Node Info = " << p->m_SomeInfo << endl;

      p = p->m_Next;        // Move p to next Node in List
   }

   cout << "End of Node List" << endl;
   return(0);
}
```

Linked Lists




```
struct Node {
public:

   ItemType    item;
   Node        *next;
};
```

```
typedef  Node*   ptrType;    // ptrType is convenient alias
ptrType  head;               // Node *head;
```



Contrast:

Array implementation:

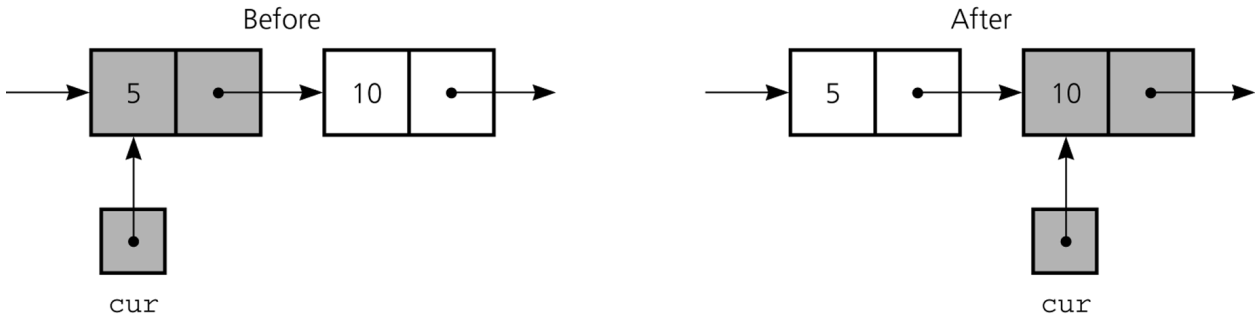
Sequential "storage" of the list items in memory.
Direct access to any item in the list: A [i]
Maximum list capacity dependent on array size

Linked list implementation:

Random "storage" of list items in memory. Each Node separately allocated.
Must use **sequential access** to reach any specific item in the list. "**Traversal**".
No inherent limitation to capacity of list. Just link new Nodes into the list.

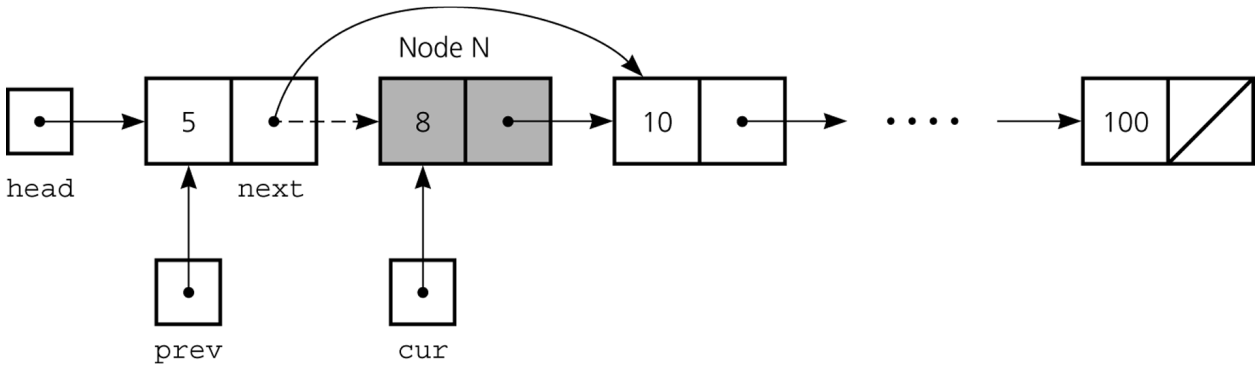
Traversing a Linked List (e.g., "visit" each node in the list)

```
ptrType  cur;
cur = head;

while(cur != NULL)
{
   // "visit" Node cur.  For example, print out item
   cout << cur->item << endl;

   // Move to next item in list
   cur = cur->next;
}
```

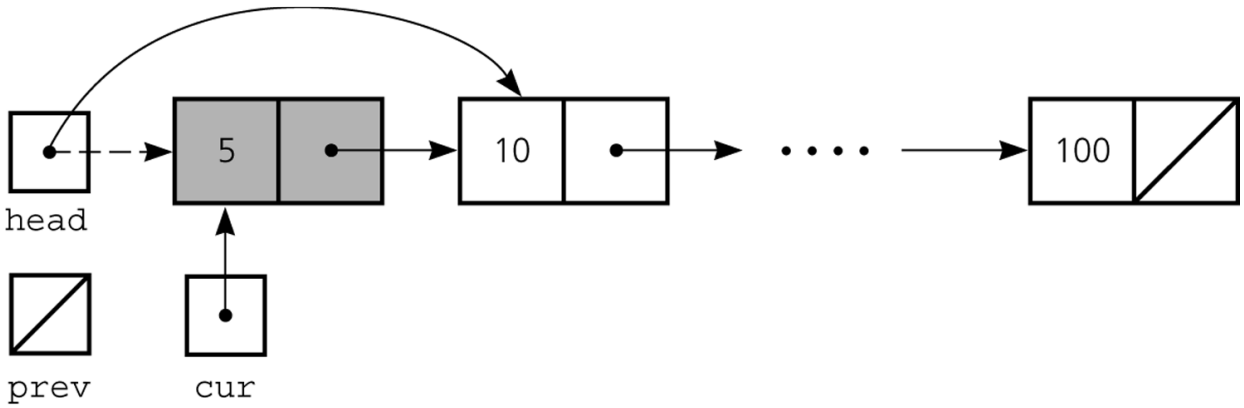


The effect of the assignment *cur = cur->next*

Deleting a node from a linked list



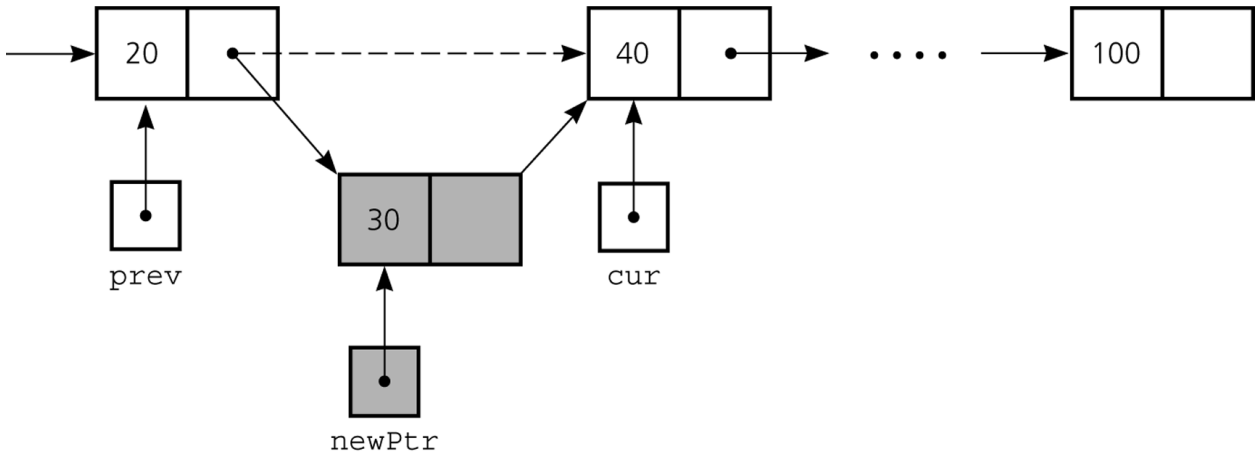

```
// Assume prev points to Node before cur Node
//   → this must be established before deleting

prev->next = cur->next;

// DeAllocate Node cur
cur->next = NULL;          // Safe programming
delete cur;
cur = NULL;                // Safe programming
```



Special case if deleting the first node. prev is NULL.

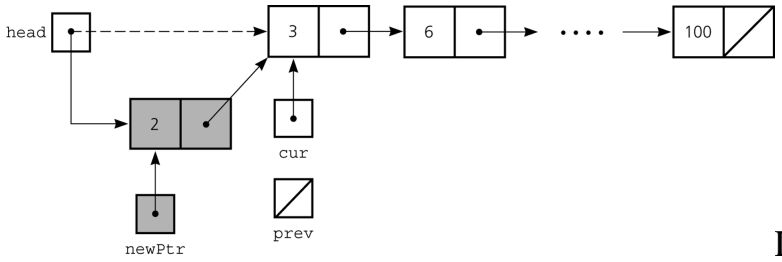
```
if(prev == NULL)                  // or, if(head == cur)
   head = cur->next;
else
   prev->next = cur->next;

delete cur;
```

Inserting a node into a linked list



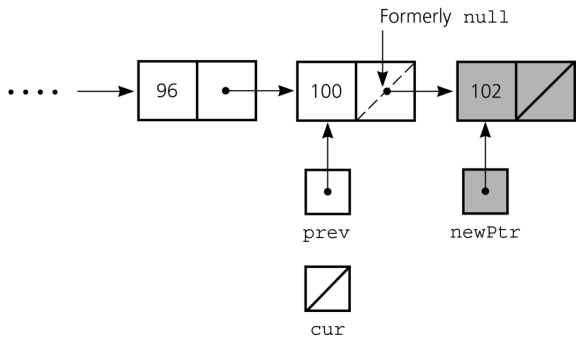

```
// Assume prev points to Node before cur Node
//   → this must be established before inserting

Node *newPtr;
newPtr = new Node;          // Allocate new Node

newPtr->item = 30;          // Set item data value(s)

newPtr->next = cur;         // Link newPtr Node into list
prev->next   = newPtr;
```

Special case if inserting Node at front of list. prev is NULL.




```
newPtr->next = cur;

if(prev == NULL)
   head = newPtr;
else
   prev->next = newPtr;
```

Insert Node at end. cur is NULL.



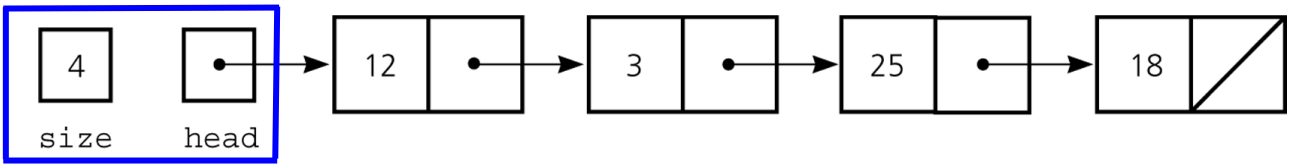
List ADT: Class-based Linked implementation
By-Position List

Head, Size: become private member variables of List class
List class utilizes internal private Node linked-list

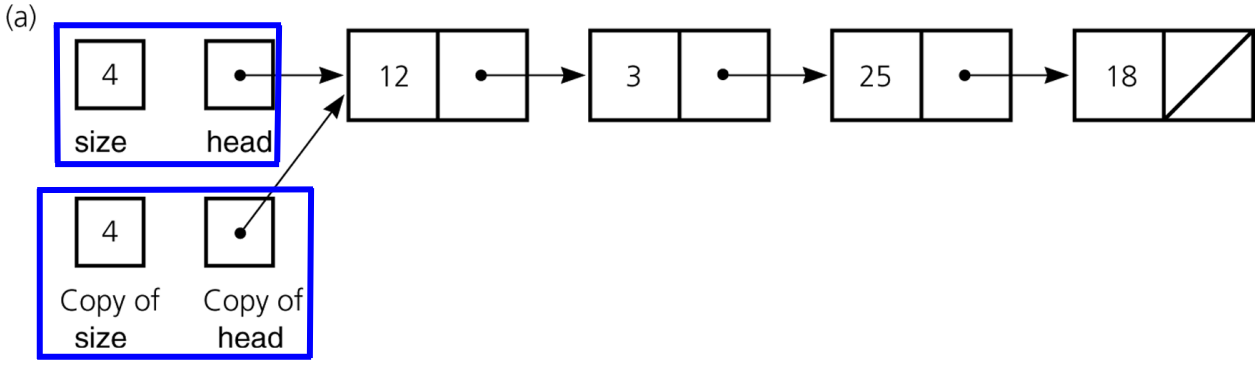
```
#ifndef  LIST
#define  LIST

typedef  int  ListItemType;    // Modify to configure for list data

class List
{
private:
   // Internal class definition, used locally within List
   //    Can utilize either class or struct, as desired
   //    Convention: use struct if only data members, no methods
   class Node                        // struct Node
   {                                 // {
    public:                          //  public:  // default for struct
      ListItemType  item;            //    ListItemType  item;
      Node  *next;                   //    Node  *next;
      // Addt'l member fcns          // };
      // . . . . .
   };

   typedef  Node*  ptrNode;

public:
   // List: public interface member functions
   List();                                        // constructor
   ~List();                                       // destructor
   List(const List &origList);                    // copy constructor
   const List& operator=(const List& rhs);    // assignment oper

   bool isEmpty() const;
   int  getSize() const;
   bool insert(int position, ListItemType &data);
   bool remove(int position);
   bool retrieve(int position, ListItemtype &data) const;
   // . . . .

private:
   // List: private member data items
   ptrNode   head;      // or   Node*  head;
   int       size;
};

#endif
```

Shallow vs. Deep Copying of Dynamically Allocated List Structures

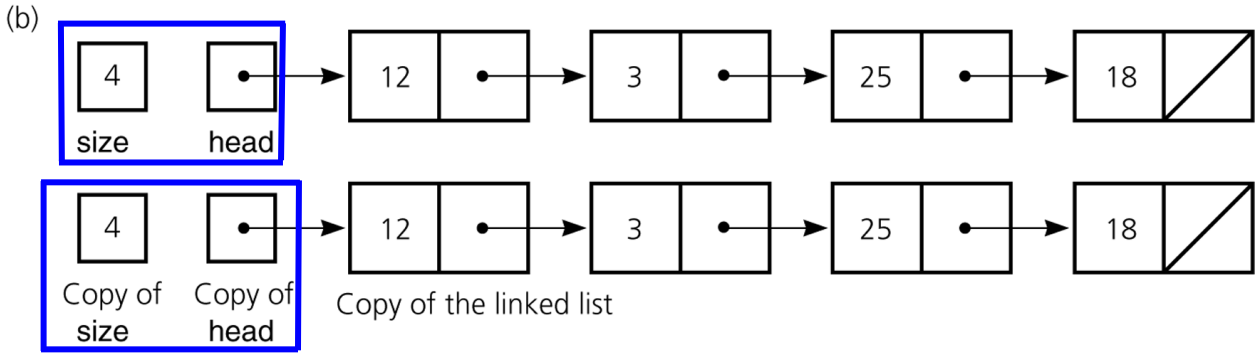
A pointer-based implementation of the ADT list:
Member variables: size, head



(a) A shallow copy of the list. Only the member variables are copied (Default).



(b) A deep copy of the list. Member variables + entire list must be replicated.
Requires explicit Copy-Constructor( ) and operator=( ) methods.



(c) ~Destructor( ) method required to de-allocate the dynamic memory.

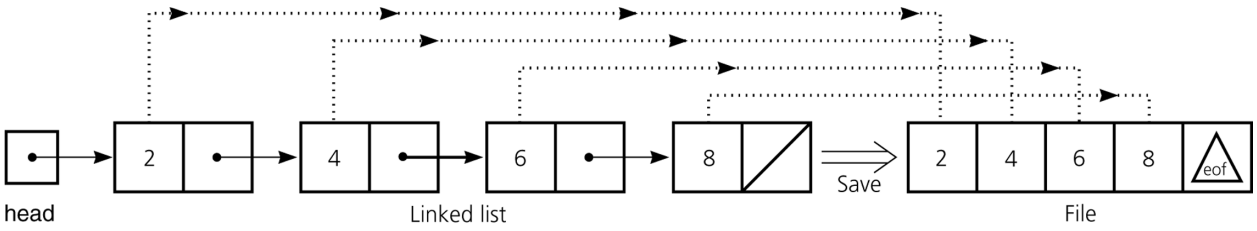
Saving the contents of a linked list to a File.

Traverse the list

Write only the data item values to the file in sequential order.

Do not write out the pointers themselves (which have no relevance externally)

Since the data values are in sequence, can rebuild the list during input processing.

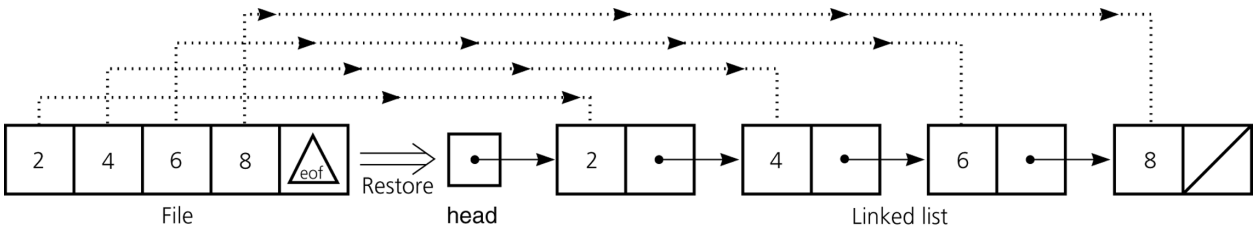


Reading and Restoring the contents of a linked list from a File.

Read the sequential data value items from the file one at a time.

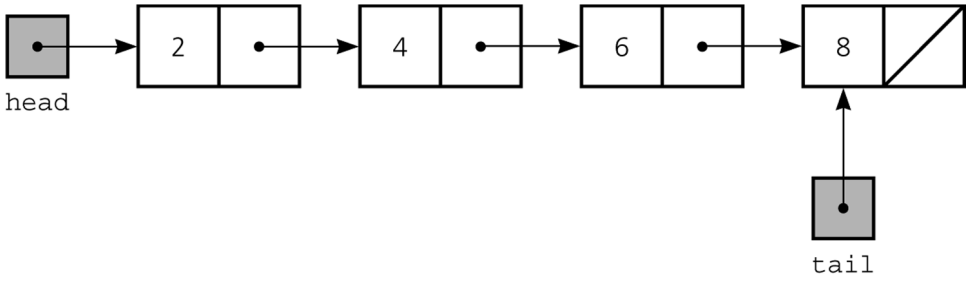
For each input item, allocate a new Node and save the data value(s).

Insert each Node into the linked list at the end.

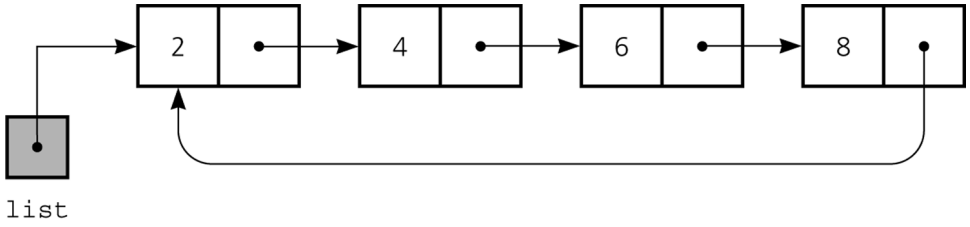


Linked List Variations

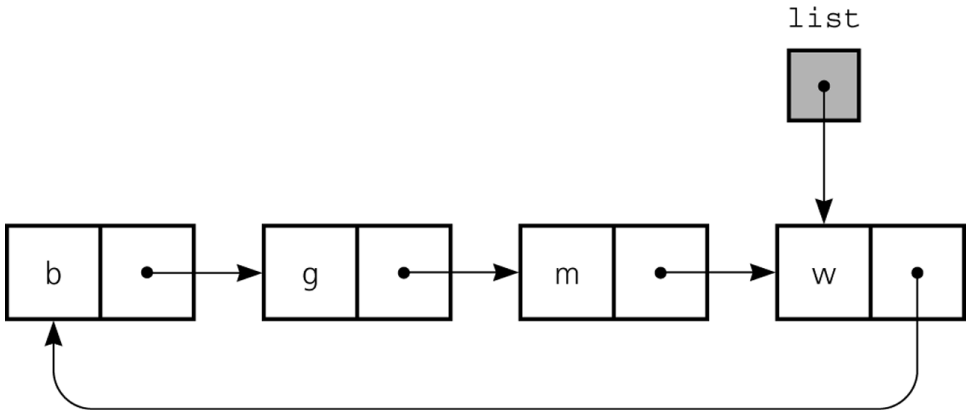
A linked list with both head and tail pointers



A circular linked list

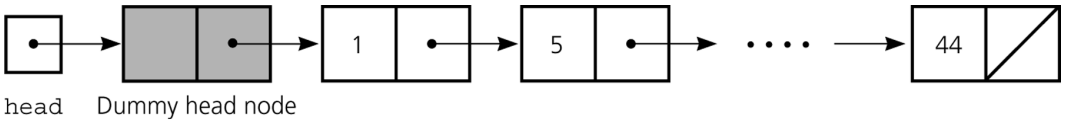


A circular linked list with an external pointer to the last node



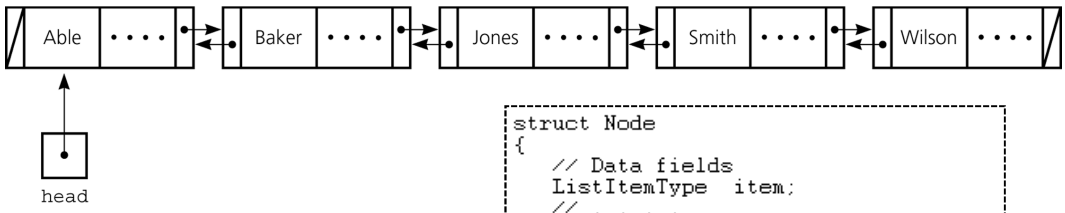
A dummy header node

Allocate one extra node, so that "head" is never NULL.



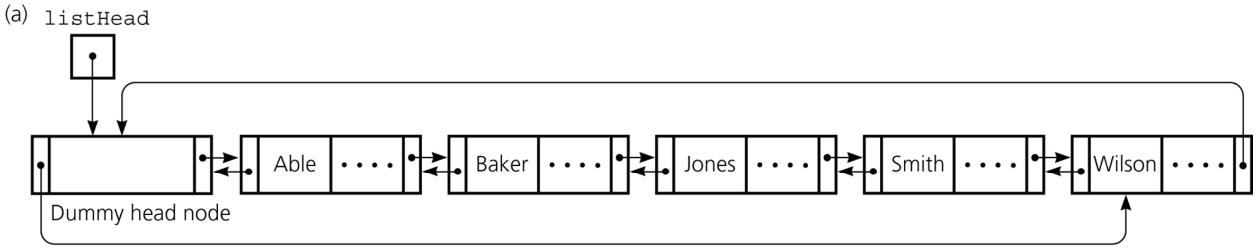
A doubly-linked list

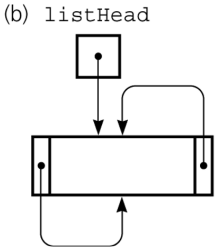
Each Node contains two pointers: (**next** and **prev**) or (**rightLink** and **leftLink**)




```
struct Node
{
   // Data fields
   ListItemType  item;
   // . . . .

   // Linkages
   Node  *next;   // rlink, rightlink
   Node  *prev;   // llink, leftlink
};
```

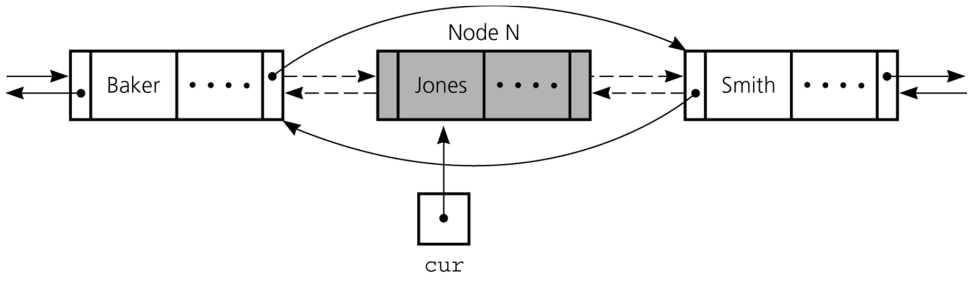
A circular doubly-linked list with a dummy header node






```
An empty list: Dummy header node always allocated
  Node *listHead;
  listHead = new Node;
  listHead.next = listHead;
  listHead.prev = listHead;
  // Don't care about data fields -- unused
```

Pointer changes for deletion.



Pointer changes for insertion.