Linked Lists

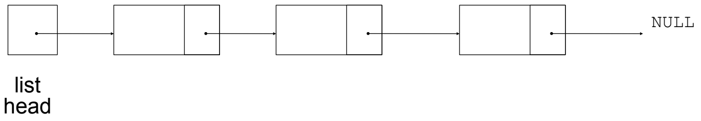
Ch 17 (Gaddis)

CS 3358
Summer I 2012

Jill Seaman

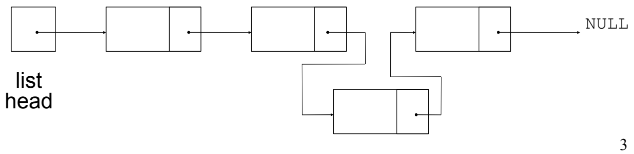
Introduction to Linked Lists

- A data structure representing a list
- A series of nodes chained together in sequence
  - Each node points to one other node.
- A separate pointer (the head) points to the first item in the list.
- The last element points to nothing (NULL)



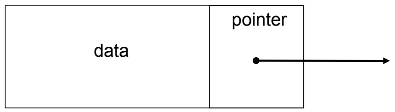
Introduction to Linked Lists

- The nodes are dynamically allocated
  - The list grows and shrinks as nodes are added/ removed.
- Linked lists can easily insert a node between other nodes
- Linked lists can easily delete a node from between other nodes



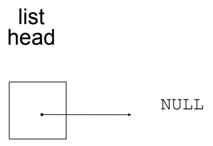
Node Organization

- Each node contains:
  - data field – may be organized as a structure, an object, etc.
  - a pointer – that can point to another node



Empty List

- An empty list contains 0 nodes.
- The list head points to NULL (address 0)
- (There are no nodes, it’s empty)



Declaring the Linked List data type

- We will be defining a class for a linked list data type that can store values of type double.
- The data type will describe the values (the lists) and operations over those values.
- In order to define the values we must:
  - define a data type for the nodes
  - define a pointer variable (head) that points to the first node in the list.

Declaring the Node data type

- Use a struct for the node type

```
struct ListNode {
   double value;
   ListNode *next;
};
```

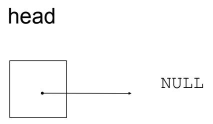
- (this is just a data type, no variables declared)
- next can hold the address of a ListNode.
  - it can also be NULL
  - “self-referential data structure”

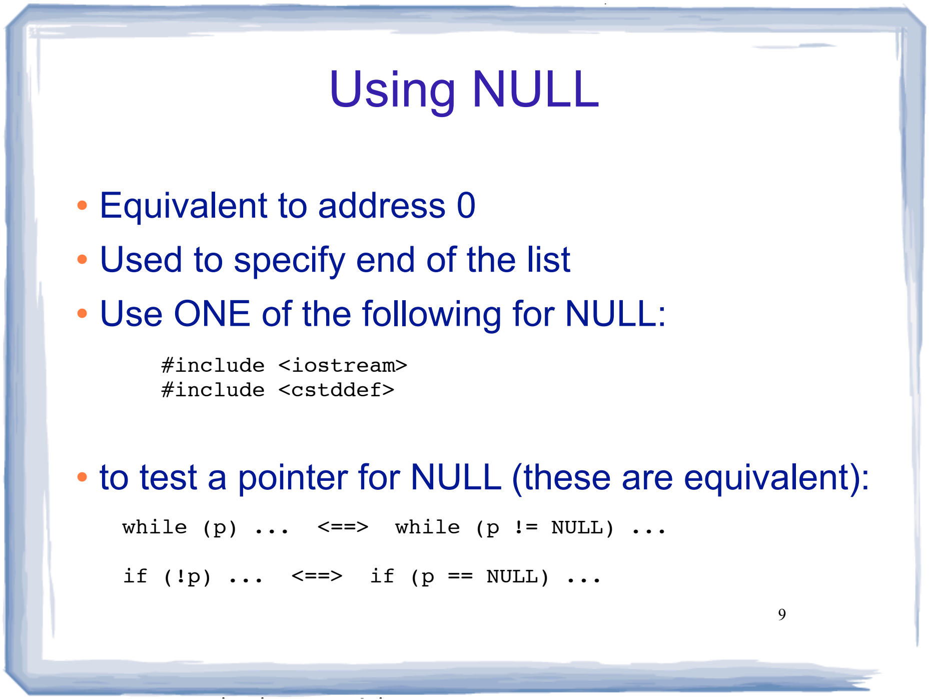
Defining the Linked List variable

- Define a pointer for the head of the list:

```
ListNode *head = NULL;
```

- It must be initialized to NULL to signify the end of the list.
- Now we have an empty linked list:





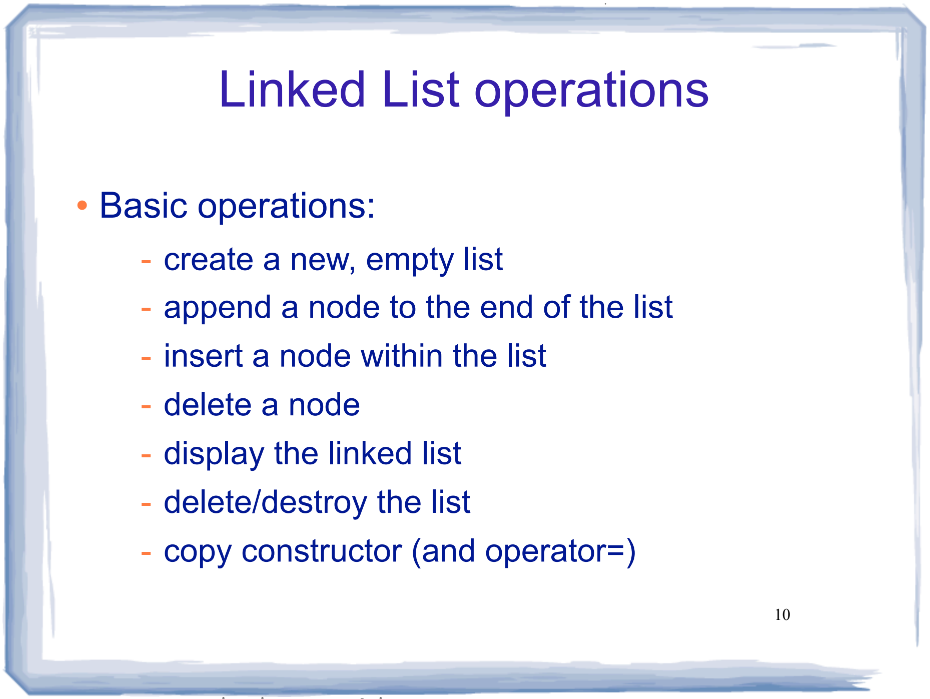
Using NULL

- Equivalent to address 0
- Used to specify end of the list
- Use ONE of the following for NULL:

```
#include <iostream>
#include <cstddef>
```

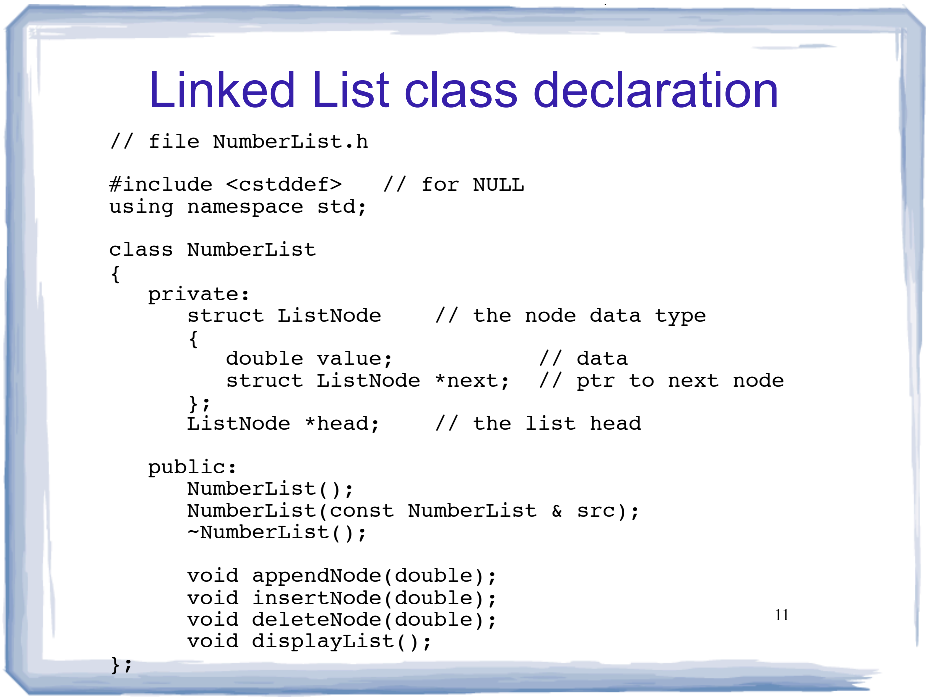
- to test a pointer for NULL (these are equivalent):

```
while (p) ...  <==>  while (p != NULL) ...

if (!p) ...  <==>  if (p == NULL) ...
```



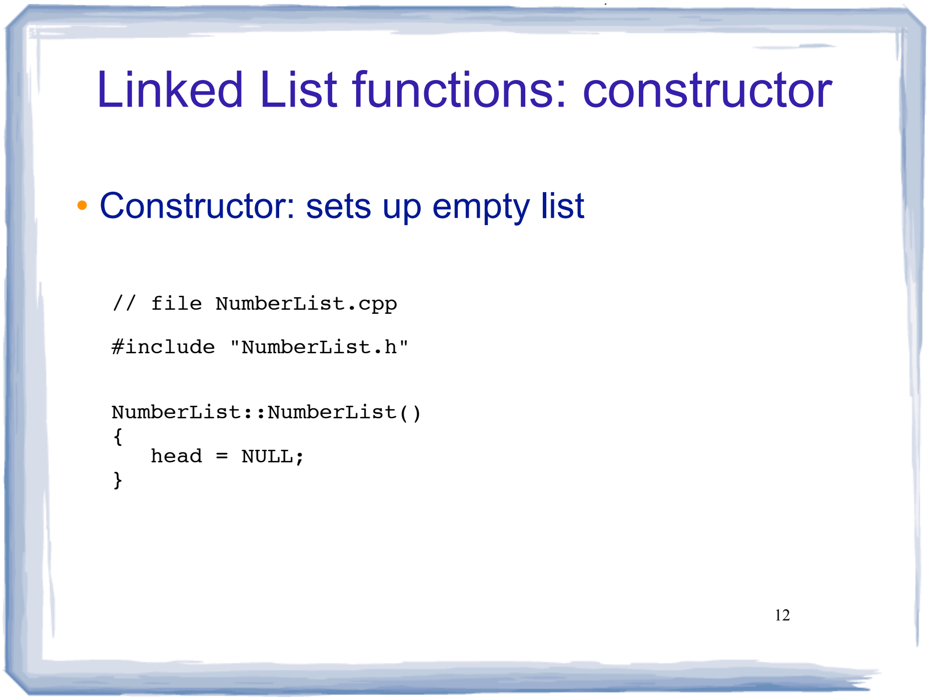
Linked List operations

- Basic operations:
  - create a new, empty list
  - append a node to the end of the list
  - insert a node within the list
  - delete a node
  - display the linked list
  - delete/destroy the list
  - copy constructor (and operator=)



Linked List class declaration

```
// file NumberList.h

#include <cstddef>   // for NULL
using namespace std;

class NumberList
{
   private:
      struct ListNode    // the node data type
      {
         double value;           // data
         struct ListNode *next;  // ptr to next node
      };
      ListNode *head;    // the list head

   public:
      NumberList();
      NumberList(const NumberList & src);
      ~NumberList();

      void appendNode(double);
      void insertNode(double);
      void deleteNode(double);
      void displayList();
};
```



Linked List functions: constructor

- Constructor: sets up empty list

```
// file NumberList.cpp

#include "NumberList.h"


NumberList::NumberList()
{
   head = NULL;
}
```

Linked List functions: appendNode

- appendNode: adds new node to end of list
- Algorithm:

```
Create a new node and store the data in it
If the list has no nodes (it's empty)
  Make head point to the new node.
Else
  Find the last node in the list
  Make the last node point to the new node
```

When defining list operations, always consider special cases:
- Empty list
- First element, front of the list (when head pointer is involved)

Linked List functions: appendNode

- How to find the last node in the list?
- Algorithm:

```
Make a pointer p point to the first element
while (the node p points to) is not pointing to NULL
  make p point to (the node p points to) is pointing to
```

- In C++:

```
ListNode *p = head;
while ((*p).next != NULL)
   p = (*p).next;
```

<==>

```
ListNode *p = head;
while (p->next)
   p = p->next;
```

p=p->next is like i++

Linked List functions: appendNode

```
void NumberList::appendNode(double num) {

   ListNode *newNode;  // To point to the new node

   // Create a new node and store the data in it
   newNode = new ListNode;
   newNode->value = num;
   newNode->next = NULL;

   // If empty, make head point to new node
   if (!head)
      head = newNode;

   else {
      ListNode *nodePtr;  // To move through the list
      nodePtr = head;     // initialize to start of list

      // traverse list to find last node
      while (nodePtr->next)          //it's not last
         nodePtr = nodePtr->next;   //make it pt to next

      // now nodePtr pts to last node
      // make last node point to newNode
      nodePtr->next = newNode;
   }
}
```

Traversing a Linked List

- Visit each node in a linked list, to
  - display contents, sum data, test data, etc.
- Basic process:

```
set a pointer to point to what head points to
while pointer is not NULL
  process data of current node
  go to the next node by setting the pointer to
     the pointer field of the current node
end while
```

Linked List functions: displayList

```
void NumberList::displayList() {
   ListNode *nodePtr;  //ptr to traverse the list

   // start nodePtr at the head of the list
   nodePtr = head;

   // while nodePtr pts to something (not NULL), continue
   while (nodePtr)
   {
      //Display the value in the current node
      cout << nodePtr->value << endl;

      //Move to the next node
      nodePtr = nodePtr->next;
   }
}
```

Or the short version:

```
void NumberList::displayList() {
   ListNode *nodePtr;
   for (nodePtr = head; nodePtr; nodePtr = nodePtr->next)
      cout << nodePtr->value << endl;
}
```

Driver to demo NumberList

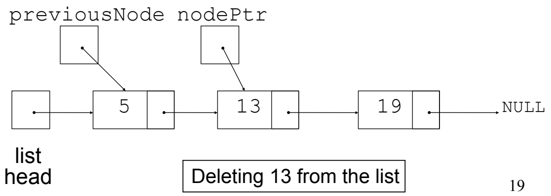
- ListDriver.cpp

```
//ListDriver.cpp: using NumberList

#include "NumberList.h"

int main() {

   // Define the list
   NumberList list;

   // Append some values to the list
   list.appendNode(2.5);
   list.appendNode(7.9);
   list.appendNode(12.6);

   // Display the values in the list.
   list.displayList();
   return 0;
}
```

```
Output:
2.5
7.9
12.6
```

Deleting a Node from a Linked List

- deleteNode: removes node from list, and deletes (deallocates) the removed node.
- Requires two pointers:
  - one to point to the node to be deleted
  - one to point to the node <u>before</u> the node to be deleted.

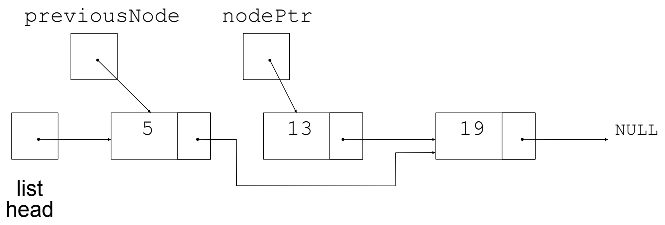


Deleting 13 from the list

Deleting a node

- Change the pointer of the previous node to point to the node <u>after</u> the one to be deleted.

```
previousNode->next = nodePtr->next;
```

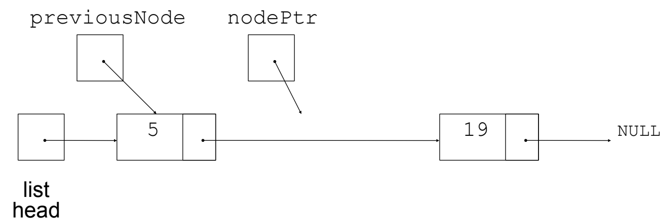


- Now just “delete” the nodePtr node

Deleting a node

- After the node is deleted:

```
delete nodePtr;
```



Delete Node Algorithm

- Delete the node containing num

```
If list is empty, exit
If first node is num
   make p point to first node
   make head point to second node
   delete p
else
   use p to traverse the list, until it points to num or NULL
   --as p is advancing, make n point to the node before
   if (p is not NULL)
      make n’s node point to what p’s node points to
      delete p’s node
```

Linked List functions: deleteNode

```
void NumberList::deleteNode(double num) {

   if (!head)   // empty list, exit
      return;

   ListNode *nodePtr;         // to traverse the list
   if (head->value == num) {  // if first node is num
      nodePtr = head;
      head = nodePtr->next;
      delete nodePtr;
   }
   else {
      ListNode *previousNode;  // trailing node pointer
      nodePtr = head;      // traversal ptr, set to first node

      // skip nodes not equal to num, stop at last
      while (nodePtr && nodePtr->value != num) {
         previousNode = nodePtr;   // save it!
         nodePtr = nodePtr->next;  // advance it
      }

      if (nodePtr) {      // num is found, set links + delete
         previousNode->next = nodePtr->next;
         delete nodePtr;
      }
      // else: end of list, num not found in list, do nothing
   }
}
```

Driver to demo NumberList

Nice test case, checks all three cases

- ListDriver.cpp

```
// set up the list
NumberList list;
list.appendNode(2.5);
list.appendNode(7.9);
list.appendNode(12.6);
list.displayList();

cout << endl << "remove 7.9:" << endl;
list.deleteNode(7.9);
list.displayList();

cout << endl << "remove 8.9: " << endl;
list.deleteNode(8.9);
list.displayList();

cout << endl << "remove 2.5: " << endl;
list.deleteNode(2.5);
list.displayList();
```

```
Output:
2.5
7.9
12.6

remove 7.9:
2.5
12.6

remove 8.9:
2.5
12.6

remove 2.5:
12.6
```

Destroying a Linked List

- The destructor must “delete” (deallocate) all nodes used in the list
- To do this, use list traversal to visit each node
- For each node,
  - save the address of the next node in a pointer
  - delete the node

Linked List functions: destructor

- ~NumberList: deallocates all the nodes

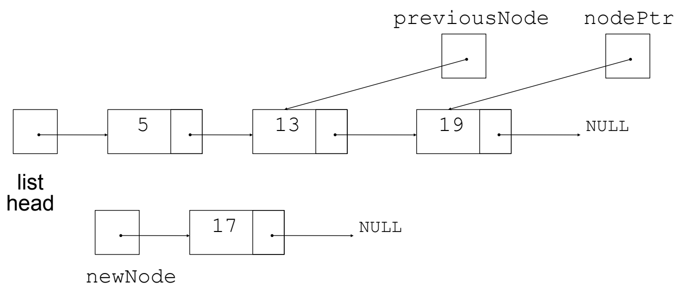
```
NumberList::~NumberList() {

   ListNode *nodePtr;   // traversal ptr
   ListNode *nextNode;  // saves the next node

   nodePtr = head;      //start at head of list

   while (nodePtr) {

      nextNode = nodePtr->next;  // save the next
      delete nodePtr;            // delete current
      nodePtr = nextNode;        // advance ptr
   }

}
```

Inserting a Node into a Linked List

- Requires two pointers:
  - pointer to point to the node after the insertion point
  - pointer to point to node before point of insertion
- New node is inserted between the nodes pointed at by these pointers
- The before and after pointers move in tandem as the list is traversed to find the insertion point
  - Like delete

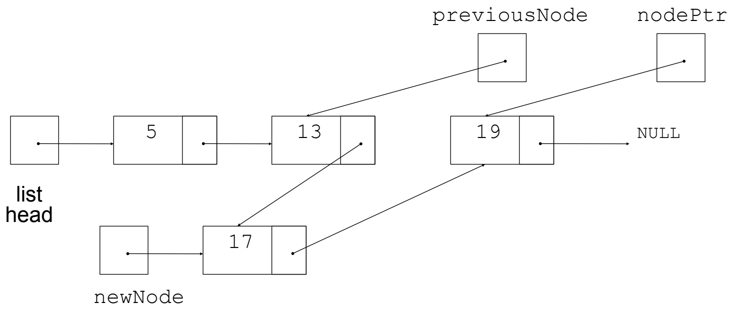
Inserting a Node into a Linked List

- New node created, new position located:



Inserting a Node into a Linked List

- Insertion completed:



Insert Node Algorithm

- Insert node in a certain position

```
Create the new node, store the data in it
If list is empty,
  make head point to new node, new node to null
else
  use p to traverse the list,
    until it points to node after insertion point or NULL
    --as p is advancing, make n point to the node before
  if p points to first node (n is null)
    make head point to new node
    new node to p's node
  else
    make n's node point to new node
    make new node point to p's node
```

Insert Node Algorithm

- Note that the insertNode implementation that follows assumes that the list is sorted.
- The insertion point is
  - immediately before the first node in the list that has a value greater than the value being inserted or
  - at the end if the value is greater than all items in the list.
- Not always applicable, but it is a good demonstration of inserting a node in the middle of a list.

Linked List functions: insertNode

- insertNode: inserts num into middle of list

```
void NumberList::insertNode(double num) {
   ListNode *newNode;        // ptr to new node
   ListNode *nodePtr;        // ptr to traverse list
   ListNode *previousNode;   // node previous to nodePtr

   //allocate new node
   newNode = new ListNode;
   newNode->value = num;

   // empty list, insert at front
   if (!head) {
      head = newNode;
      newNode->next = NULL;
   }

   //else is on the next slide . . .
```

Linked List functions: insertNode

```
   else {
      // initialize the two traversal ptrs
      nodePtr = head;
      previousNode = NULL;

      // skip all nodes less than num
      while (nodePtr && nodePtr->value < num) {
         previousNode = nodePtr;   // save
         nodePtr = nodePtr->next;  // advance
      }

      if (previousNode == NULL) { //insert before first
         head = newNode;
         newNode->next = nodePtr;
      }
      else {                //insert after previousNode
         previousNode->next = newNode;
         newNode->next = nodePtr;
      }
   }
}
```

What if num is bigger than all items in the list?

Driver to demo NumberList

- ListDriver.cpp

```
int main() {

   // set up the list
   NumberList list;
   list.appendNode(2.5);
   list.appendNode(7.9);
   list.appendNode(12.6);

   list.insertNode(10.5);

   list.displayList();

   return 0;
}
```

```
Output:
2.5
7.9
10.5
12.6
```

Copy Constructor

- Pointers + dynamic allocation => deep copy
- Don’t copy any pointers

```
Initialize head to NULL
For each item in the src list (in order)
  append item.value to this list
```

Linked List functions: copy constructor

```
NumberList::NumberList(const NumberList & src) {

    head = NULL;         // initialize empty list

    // traverse src list, append its values to end of this list
    ListNode *nodePtr;

    for (nodePtr=src.head; nodePtr; nodePtr=nodePtr->next)
    {
        appendNode(nodePtr->value);
    }

}
```

Chapter 17 in Weiss book

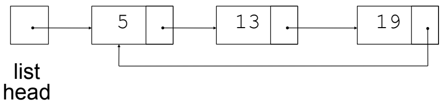
- Elegant implementation of linked lists
- It uses a “header node”
  - empty node, immediately before the first node in the list
  - eliminates need for most special cases
  - internal traversals must skip that node
  - not visible to users of the class

Chapter 17 in Weiss book

- It uses three separate classes (w/ friend access)
  - LListNode (value+next ptr)
  - LListItr (an iterator)
  - LList (the linked list, with operations)
- What is an iterator?
  - a reference/ptr to a specific element in a specific list
  - Operations:
    - advance: make it “point” to next element
    - retrieve: return the value it “points” to (dereference)
    - isValid: returns true if not NULL (it points to an element)

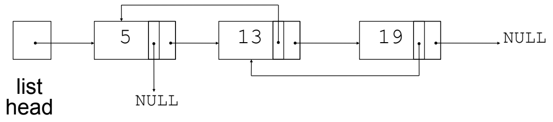
Linked List variations

- Circular linked list
  - last cell’s next pointer points to the first element.



Linked List variations

- Doubly linked list
  - each node has two pointers, one to the next node and one to the previous node
  - head points to first element, tail points to last.
  - can traverse list in reverse direction by starting at the tail and using `p=p->prev`.



Advantages of linked lists
(over arrays)

- A linked list can easily grow or shrink in size.
  - The programmer doesn’t need to predict how many values could be in the list.
  - The programmer doesn’t need to resize (copy) the list when it reaches a certain capacity.
- When a value is inserted into or deleted from a linked list, none of the other nodes have to be moved.

Advantages of arrays
(over linked lists)

- Arrays allow random access to elements: array[i]
  - linked lists allow only sequential access to elements (must traverse list to get to i’th element).
- Arrays do not require extra storage for “links”
  - linked lists are impractical for lists of characters or booleans (pointer value is bigger than data value).