Linked List Mechanics

Slides

1. Table of Contents
2. Linear Lists
3. List Node Design
4. Self Referencing Structures
5. List Formation Example
6. List Traversal
7. Linked List Operations
8. Data Element Dependency
9. Some Basic Operations
10. List Insertion Cases 1 & 2
11. List Insertion Case 3
12. List Insertion Case 4
13. Ordered Insertion
14. List Remove Case 1
15. List Remove Case 2
16. List Remove Case 3
17. Ordered Remove
18. Ordered Modify
19. Linear Linked-List Variations
20. Linked-List Variation Declarations
21. Circular Double Linked Insertion
22. Circular Double Linked Deletion
23. Ordered List: Array of Records
24. Array of Records: Manipulations
25. Array of Records: Insert

 ©1995-2001 Barnette ND, McQuain WD

Linear Lists

Sequential Lists

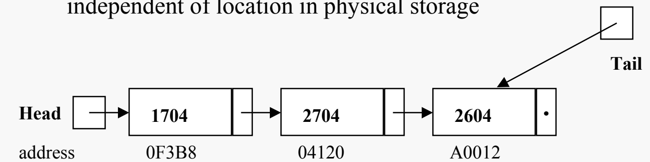
- Array
- Logical order matches physical order
- i.e.: Class List = (1704, 2704, 2604)

Inserting & deleting elements in a sequential list requires copying & shifting of elements.

position	1	2	3
value	1704	2704	2604
address	004A08	004A0C	004A10

Linked Lists

- Logical group of ordered elements whose physical order is independent of location in physical storage



List is a sequence of nodes. Each node contains data and a pointer to the next node in the list.

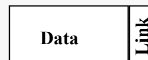
Here, Head and Tail pointers are used to keep track of the ends of the list.

Inserting & deleting elements in linked lists only require changes to the links.

 ©1995-2001 Barnette ND, McQuain WD

List Node Design

Each node of a linked list is a structured variable that contains both data and (at least) a pointer to the next node in the list.



There should be a declaration of a node type. That type may be either a `struct` or a `class`. We will consider both approaches in these notes.

Whichever approach is taken, the Data field "**should**" (must) encapsulate all logical data stored in the node and separate that data from the Link field which points to the next node in the linked list.

So, the Data field should also be a structured variable (struct or class) and the node type will then be hierarchical.

Ideally, the linked list is implemented as a "container" for data; that is, the list will accept data elements of essentially any type.

We'll achieve that goal by treating the data element as a type defined "capsule".

We will revisit the issue of the proper engineering of a linked list later; for now, we will focus on the mechanics of creating and manipulating a linked list.

©1995-2001 Barnette ND, McQuain WD

Self Referencing Structures

Linked List Node Declaration Issues

A linked list node must contain a pointer to a linked list node. That poses a delicate scope issue in C++.

```
typedef int Item;
typedef struct {
   Item   Element;
   Node*  Next;
} Node; //error
```

This code won't compile. The declaration of the pointer field uses the type name `Node` before `Node` has been declared.

```
typedef int Item;
struct Node {
   Item   Element;
   Node*  Next;
};
```

This code will compile, but it's better practice to `typedef` the pointers. And that brings us back to the situation above. You can't `typedef Node*` before `Node` has been declared.

Forward Declaration

Fortunately, C++ provides a solution. We can simply make a forward declaration of `Node`, prior to defining the `Node` type:

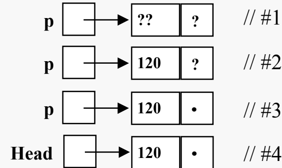
```
typedef int Item;

struct  Node;          // forward declaration
typedef Node* NodePtr;

struct Node {
   Item     Element;
   NodePtr  Next;
};


NodePtr Head = NULL;  // head pointer for list
```


©1995-2001 Barnette ND, McQuain WD

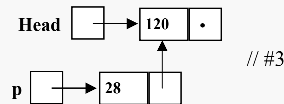
List Formation Example

Assume the declarations from the previous slide.

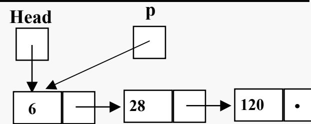
```
NodePtr p = NULL,
        q = NULL;
p = new Node;          // #1
p->Element = 120;      // #2
p->Next = NULL;        // #3
Head = p;              // #4
p = NULL;
```




```
p = new Node;          // #1
p->Element = 28;       // #2
p->Next = Head;        // #3
Head = p;              // #4
p = NULL;
```



```
p = new Node;
p->Element = 6;
p->Next = Head;
Head = p;
```



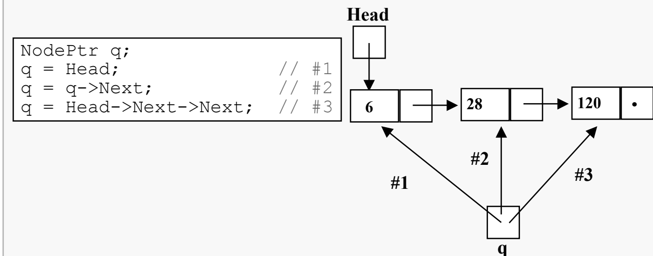

```
// What would the following stmt do?
delete p;
```

This is NOT how lists are actually formed and accessed.

List Traversal

Assume the list formed on the previous slide. By using the `Next` field of each `Node`, we may advance a pointer from `Node` to `Node`, "walking" the linked list, "visiting" each node in turn.

```
NodePtr q;
q = Head;                    // #1
q = q->Next;                 // #2
q = Head->Next->Next;        // #3
```



Statement #3 is inefficient; here's an improvement:

```
q = q->Next;                 // #3b
```

Using that logic, we can write a simple loop to traverse the list from the first node to the last:

```
q = Head;                // start at first node

while (q != NULL) {      // continue until NULL

   // put code to do something with node here

   q = q->Next;          // step to next node

}//while
```

```
//alternatively
while (q) {
```

Be sure you understand how the loop above works. Traversal from beginning to end is the most basic operation you will perform on a linked list. Almost every other list operation depends upon this.

Basic Sorted/Ordered List Operations: Design

The following operations are necessary for a robust, well-defined minimal interface to an ordered linked-list ADT:

```
makeList() : creates & initializes List to be empty


emptyList()  : tests if the List is empty

fullList()   : tests if the List is full



insert() : inserts a given item in L in the correct
   ordered position

remove(): searches List for an item with some given
   "key" value & deletes the  element if located

retrieveElem() :
    searches List for an item with some given "key"
   value and returns the  element if located

modify() : searches List for an item with some given
   "key" value & replaces the item with a given,
   (updated), item if located


destroyList(): erases all items from the List
```

The ordered list ADT operations are responsible for maintaining the sorted list item ordering, and the integrity of the list structure NOT the user/client of the ADT. Error checking responsibilities are shared between the ordered list ADT & the user/client of the ADT.

An unordered list ADT embodies the concept of a list position and entails a different set of operations.

Type Defined Data Element Declaration

```
typedef int Item;

struct  Node;          //forward declaration
typedef Node* NodePtr;

struct Node {
   Item     Element; //type defined data field
   NodePtr  Next;
};
```

Considerations

In the implementation of the list operations, one needs to be able to compare variables of type `item` to each other to maintain the ordering; but an `item` variable could be anything from a simple type (such as an `int`) to a highly complex structured type. The client/user should decide how the list will be ordered, (i.e. upon what struct field or class data member(s), termed the key/primary field/member).

Thus, the implementation must depend on the internal details of `item` and that dependency should be localized as much as possible. If the `item` type is a simple base type then the C++ language comparison operations provide this service. This can be easily accomplished for a class by overloading the relational operators, which will be covered later.

For a non-class non-base item type, (e.g., struct, array, etc.), the list ADTuser/client would need to provide relational comparison operations:

```
bool lessThan(item elem1, item elem2);
bool equalTo(item elem1, item elem2);
bool greaterThan(item elem1, item elem2);
```

Some Basic Operations

List Initialization

```
// Pre:  List is undefined or NULL
// Post: List is empty
// makeList ()
   Head = NULL;
```

Head pointer is NULL when list is empty.

Empty List Test

```
// Pre:  None
// Post: List is unchanged
// emptyList )
   return( Head == NULL );
```

Head pointer contains address of first node if list is NOT empty.

Full List Test

```
// Pre:  None
// Post: None
// fullList ( )
   //check if more memory is available
   if (available) return true;
   else return false;
```

Dependent upon underlying implementation

Destroy List

```
// Pre:  List is defined
// Post: List is empty, memory deallocated
// destroyList()

  NodePtr p = Head;

  while (Head != NULL) {
     Head = Head->Next;
     delete p;
     p = Head;
  }
```

```
//alternatively
while (Head)  {
```


©1995-2001 Barnette ND, McQuain WD

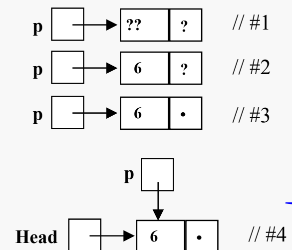
List Insertion Case 1 and 2

Declarations

```
NodePtr  Head = NULL;
NodePtr  p    = NULL;
```

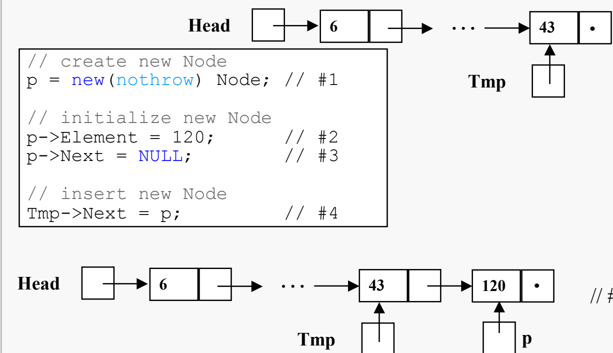
#1: Insert into empty list

```
// create a new Node
p = new(nothrow) Node; // #1

// initialize new Node
p->Element = 6;          // #2
p->Next = NULL;          // #3

// insert new Node
Head = p;                // #4
```



#2: Insert at end (tail) of list

Here we assume we have a pointer Tmp to the last node in the list.



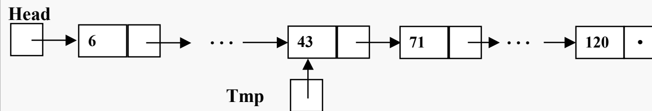

```
// create new Node
p = new(nothrow) Node; // #1

// initialize new Node
p->Element = 120;        // #2
p->Next = NULL;          // #3

// insert new Node
Tmp->Next = p;           // #4
```


©1995-2001 Barnette ND, McQuain WD

List Insertion Case 3

#3: Insert into middle of list

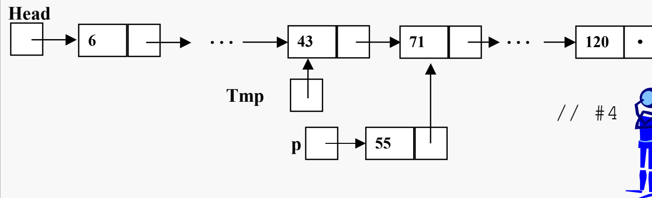
Here we assume we have a pointer Tmp to the Node that will precede the Node that's to be inserted.

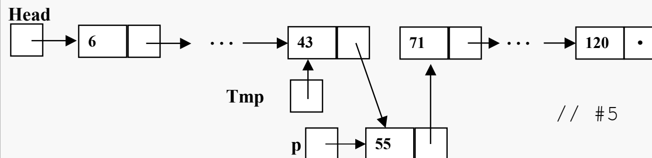



```
// create and init new Node
p = new(nothrow) Node;  // #1
p->Element = 55;        // #2
p->Next     = NULL;     // #3

// insert new Node
p->Next = Tmp->Next;    // #4
Tmp->Next = p;          // #5
```

Note that #4 and #5 must be performed in the correct order.

What happens if the statements are reversed?

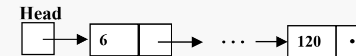




©1995-2001 Barnette ND, McQuain WD

List Insertion Case 4

#4: Insert at the front (head) of the list




```
// create and initialize new Node
p = new(nothrow) Node;  // #1
p->Element = 1;         // #2
p->Next = NULL;         // #3

// insert new Node
p->Next = Head;         // #4
Head = p;               // #5
```

What happens if statements #4 and #5 are reversed here?

©1995-2001 Barnette ND, McQuain WD

Ordered Insertion

Ordered (ascending) List Insertion code:

```
// Pre:  List is initialized
// Post: List contains the inserted elem in order
// insert()
{

   NodePtr prevPtr, currPtr, newPtr;
   newPtr = new(nothrow) Node;

   if (newPtr == NULL)
      return false;          //no memory available

   newPtr->Element = elem;   //elem is item to
   newPtr->Next = NULL;      //be inserted

   prevPtr = NULL;
   currPtr = Head;

   while ((currPtr != NULL) &&
          (elem > currPtr->Element) {
      prevPtr = currPtr;
      currPtr = currPtr->Next;
   }

   if (prevPtr == NULL) {    //insert at head, or
      newPtr->Next = Head;   //          empty list
      Head = newPtr;
   }
   else {
      prevPtr->Next = newPtr;  //insert in middle
      newPtr->Next = currPtr;  //       or at tail
   }
   return true ;            // successful insertion
}
```

prevPtr is used as a 'trailer' pointer

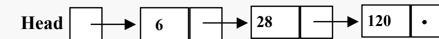
depends on Boolean short-circuiting

Assumes relational item comparison

©1995-2001 Barnette ND, McQuain WD

List Remove Case 1

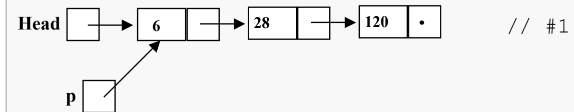
Consider the list:

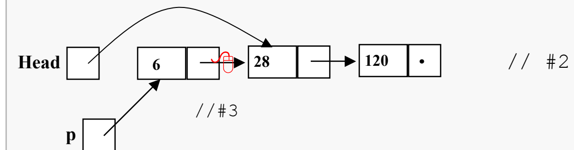


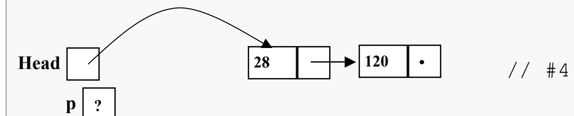
Case #1: remove the head of the list

```
// get a ptr to the target Node
p = Head;                  // #1

// reset Head "around" target Node
Head = Head->Next;         // #2
p->Next = NULL;            // #3

// delete the target Node
delete p;                  // #4
```

Pointer reset in deleted node (#3) is logically unnecessary since Node will be deleted in the next statement.





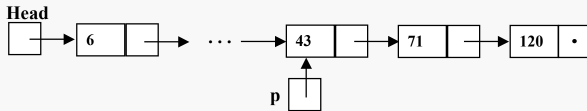


©1995-2001 Barnette ND, McQuain WD

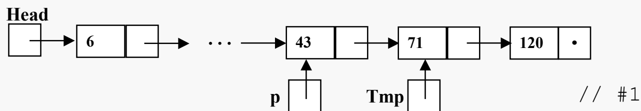
List Remove Case 2

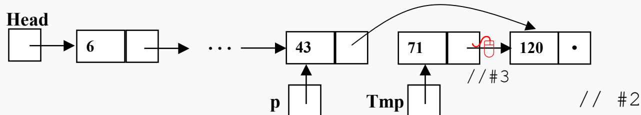
Case #2: remove from the middle of the list

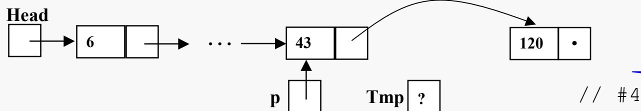
Assume the initial list below. Now we will need a pointer p to the Node that PRECEDES the targeted Node (containing 71 in this case).




```
// get ptr to targeted Node
NodePtr Tmp = p->Next;            // #1

// reset ptr in preceding Node "around" target Node
p->Next   = Tmp->Next;            // #2
Tmp->Next = NULL;                 // #3

// delete targeted Node
delete Tmp;                       // #4
```





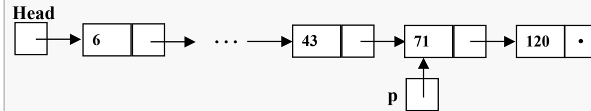


©1995-2001 Barnette ND, McQuain WD

List Remove Case 3

Case #3: remove the last Node (tail) of the list

Assume the initial list below. Again, we will need a pointer p to the Node that PRECEDES the targeted Node (containing 120 in this case).

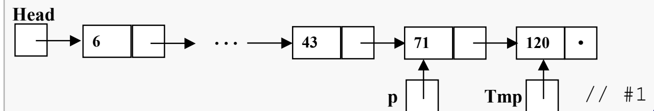


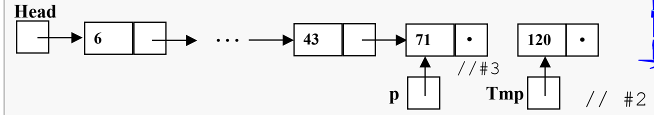

```
// get ptr to targeted (tail) Node
NodePtr Tmp = p->Next;           // #1

// reset ptr in preceding Node
p->Next   = NULL;                // #2
Tmp->Next = NULL;                // #3

// delete targeted Node
delete Tmp;                      // #4
```

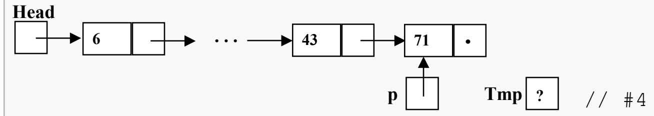
#2 could also be:

p->Next = Tmp->Next;

So, this case is really just the same as deleting in the middle of the list.







©1995-2001 Barnette ND, McQuain WD

Ordered Remove

Ordered (ascending) List Removal Function

```
// Pre:  List is initialized
// Post: delElem Item has been removed from the List
// remove()
{
   NodePtr ptr, delPtr;

   ptr = Head;

   if (emptyList(Head))
      return false;              // removal failure

   if (delElem == Head->Element) {
       Head = Head->Next;         // delete head node
       ptr->Next = NULL;
       delete ptr;
       return true;              // successful removal
   }
   // check for 1-element list
   if (ptr->Next == NULL)
      return false;

   // list has > 1-element
   // perform 1-element look-ahead search
   while( (ptr->Next->Next != NULL) &&
          (!(delElem == ptr->Next->Element))
      ptr = ptr->Next;

   // remove middle or tail node
   if (delElem == ptr->Next->Element){
      delPtr = ptr->Next;
      ptr->Next = ptr->Next->Next;
      delPtr->Next = NULL;
      delete delPtr;
      return true;               // successful removal
   }

   //end of list && delElem !found
   return false;                 // removal failure
}
```

Uses a one-node lookahead technique. Trailer pointer method is also applicable

Assumes relational item comparison

©1995-2001 Barnette ND, McQuain WD

Ordered Modify

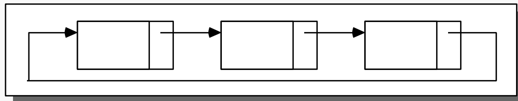
modify

```
// Pre:  List is initialized
// Post: modElem Item has been replaced in the List
// modify()
{
  NodePtr p = Head;
  bool foundElement= false;
  bool endList = (Head == NULL);

  if (!endList)
     foundElement = (p->Element == modElem);

  while (!endList && !foundElement) {
     p = p->Next;
     endList = (p == NULL);
     if (!endList)
        foundElement = (p->Element == modElem);
  }//while

  if (foundElement) {  //replace list element
     p->Element = modElem;
     return true;      //successful modification
  }

  return false;        //unsuccessful modification
}
```

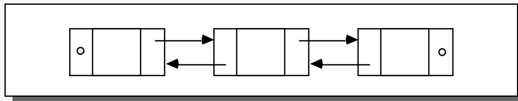
modify does NOT rely upon Boolean short-circuiting.

©1995-2001 Barnette ND, McQuain WD

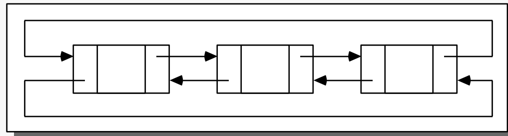
Circular List



Double Linked-List (non-circular)



Circular Double Linked-List



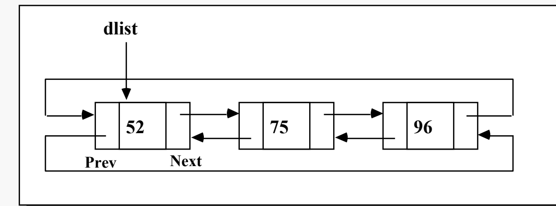
©1995-2001 Barnette ND, McQuain WD

Double Linked-List Declaration (non-circular)

```
typedef int Item;

struct  Node;
typedef Node* NodePtr;
struct  Node {
   Item Element;
   NodePtr  Prev, Next;
};

struct dblList{
   NodePtr First, Last;
};

dblList  dlist;
```

Double Linked-List Declaration (circular)

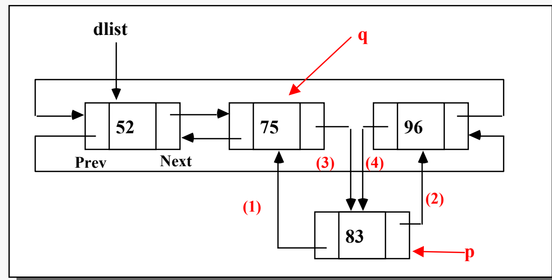
```
typedef int Item;

struct  Node;
typedef Node* dblList;
struct  Node {
   Item Element;
   dblList Prev, Next;
};

dblList  dlist;
```



©1995-2001 Barnette ND, McQuain WD

Circular Double Linked-List Insertion

Insert 83 into the ordered list:



code:

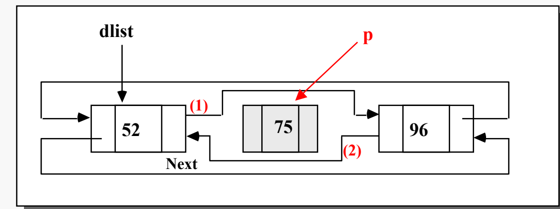
```
struct Node;
typedef  struct Node* dblList;
struct Node {
   infoType Element;
   dblList  Prev, Next;
};

dblList  dlist;

p->Prev = q;          // #1
p->Next = q->next;    // #2
q->Next = p;          // #3
p->Next->Prev = p;    // #4
```

Order is important!

©1995-2001 Barnette ND, McQuain WD

Circular Double Linked-List Deletion

Delete 75 from the list:



code:

```
p->Prev->Next = p->Next;   // #1
p->Next->Prev = p->Prev;   // #2
p->Next = p->Prev = NULL;
delete p;                  // #3
```

deleting the head:

```
if (dlist == p)
   dlist = p->Prev;

p->Prev->Next = p->Next;    // #1
p->Next->Prev = p->Prev;    // #2
P->Next = p->Prev = NULL;
delete p;                   // #3
```

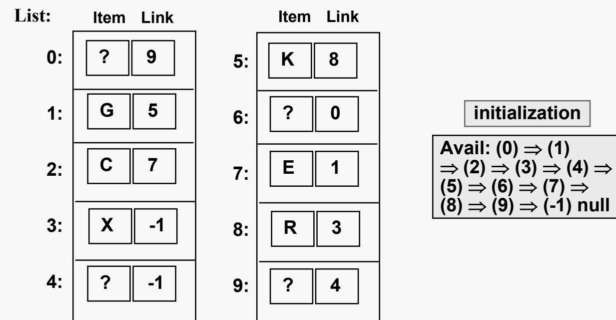
What if list has one element?

©1995-2001 Barnette ND, McQuain WD

Representation

- Array of Records (structures)

Implementation Instance (nodes structure array)



List: (2) C ⇒ (7) E ⇒ (1) G ⇒ (5) K ⇒ (8) R ⇒ (3) X ⇒ (-1) null

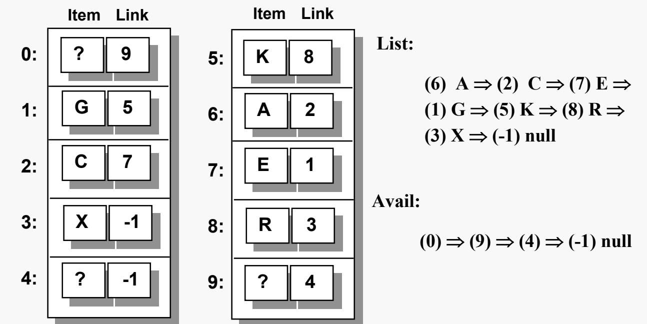
Avail: (6) ⇒ (0) ⇒ (9) ⇒ (4) ⇒ (-1) null

Available Pool

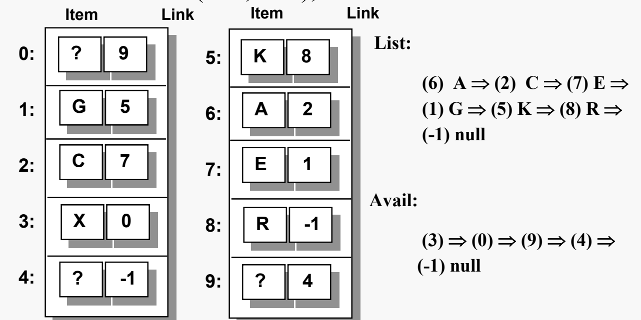
- Linked List of Free Nodes
- Deletion performed by list ADT function **AllocateNewNode()**
- Insertion performed by list ADT function **FreeNode()**
- No ordering of available pool list
- Operations must be efficient
  - Head of List — Inserts/Deletes
- Link field contains index to next available node.
- Item fields contain 'leftover' values which are ignored

©1995-2001 Barnette ND, McQuain WD

Results of insert(List, “A”);



Results of remove(List, “X”);



©1995-2001 Barnette ND, McQuain WD

Array of Records: Insert

Sorted (ascending) List Insertion Function

```
// Pre:  List is initialized
// Post: List contains the inserted elem in order
// insert()
{
  nodePtr prevPtr, currPtr, newPtr;

  newPtr = AllocateNewNode( );      // make new node

  if (newPtr == null)
     return false;                  // heap is empty

  nodes[newPtr].Element = elem;
  nodes[newPtr].link = null;
  prevPtr = null;
  currPtr = List;

  while ((currPtr != null) &&
        (elem > nodes[currPtr].Element) {

     prevPtr = currPtr;
     currPtr = nodes[currPtr].link;
  }

  if (prevPtr == null) {            //insert at head or
     nodes[newPtr].link = list;     //  in empty list
     List = newPtr;
  }
  else {
     nodes[prevPtr].link = newPtr;  //insert in middle
     nodes[newPtr].link = currPtr;  //or at tail
  }
  return true;  // successful insertion
}
```

Logic is identical to insert( ) for pointer representation.
Syntax for access to underlying structure has changed.

©1995-2001 Barnette ND, McQuain WD