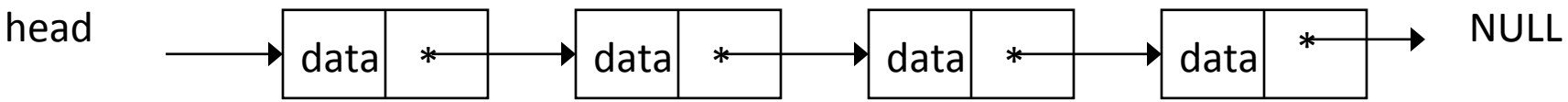
# Linked Lists

## Introduction

- A linked list consists of a number of nodes, each node containing a pointer to the next node in the list.
- A linked list can grow or shrink without limit.
  - New inserted nodes are allocated dynamically.
  - Deleted nodes are freed.
- Each node consists of:
  - data
  - a pointer to the next node
- In C++, a structure or class is usually defined for a node. E.g.

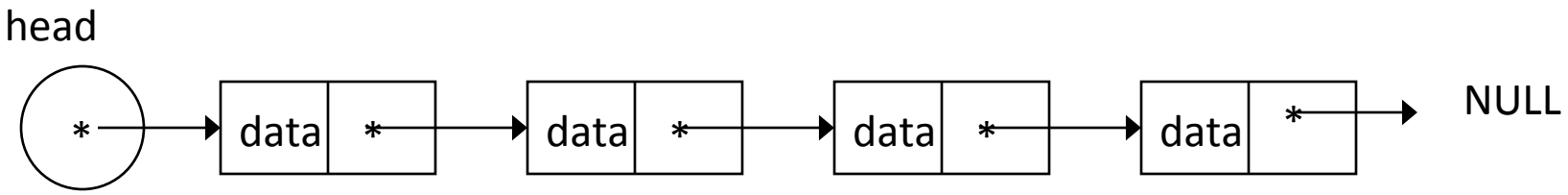



Linked List with 4 nodes

## Node Structure (continued)

```
struct Node {
    int data;
    Node *next;
};
```

- A head pointer variable is needed to point to the first node in the linked list.
  - E.g. `Node *head;`
  - The head pointer is represented with a circle in this diagram.

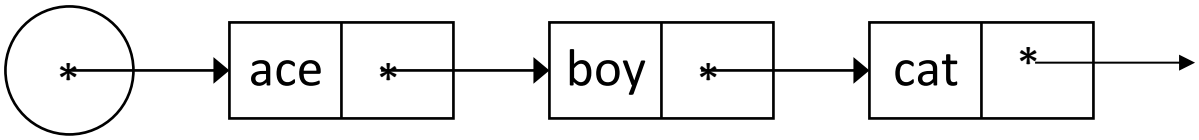


## Operations on Linked Lists

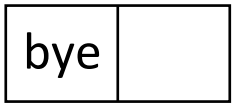
- Common operations include:
  - insert
  - delete
  - create
  - traverse
  - Search
- To insert a new item into a linked list:
  - allocate a new node
  - fill the node's data field(s)
  - find the insertion position in the current list
  - “splice” the new node into the list, adjusting pointers as necessary
- Example:

# Insert Operation (continued)

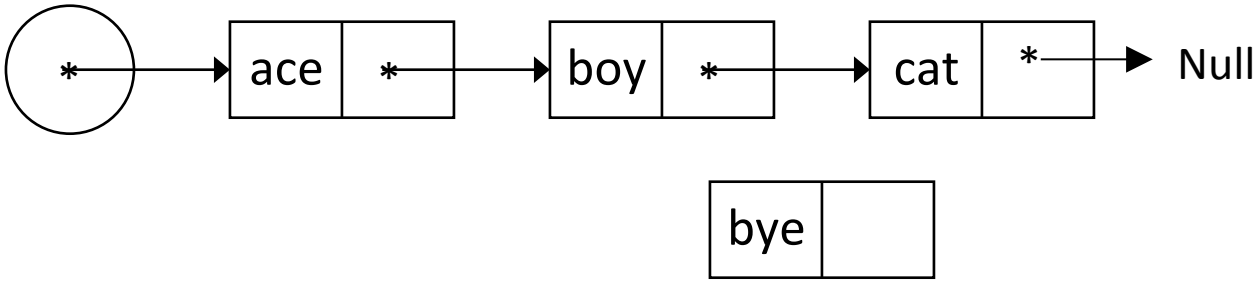
Original list




Allocate a new node, fill in the data field

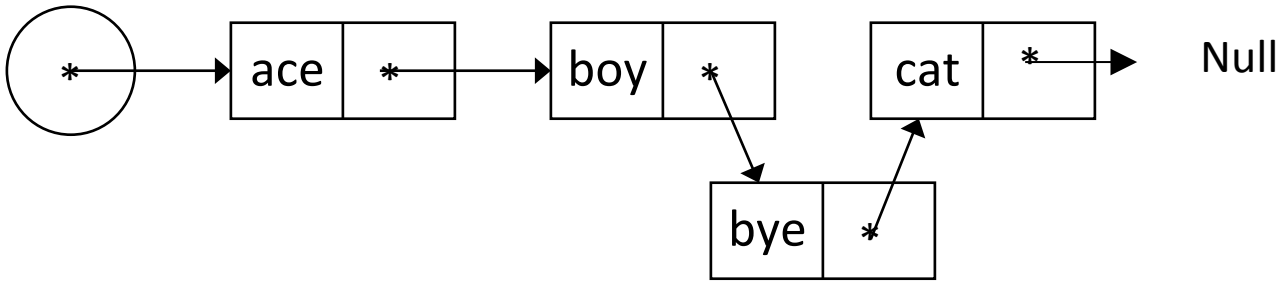



Find insertion position



## Insert Operation (continued)

Insert by adjusting pointers

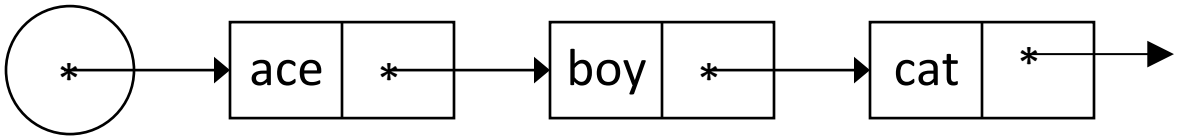



- Insertion at the beginning or end of a linked list is subtly different than inserting into the middle.
  - You will have to write special case code to handle one or both of these cases.

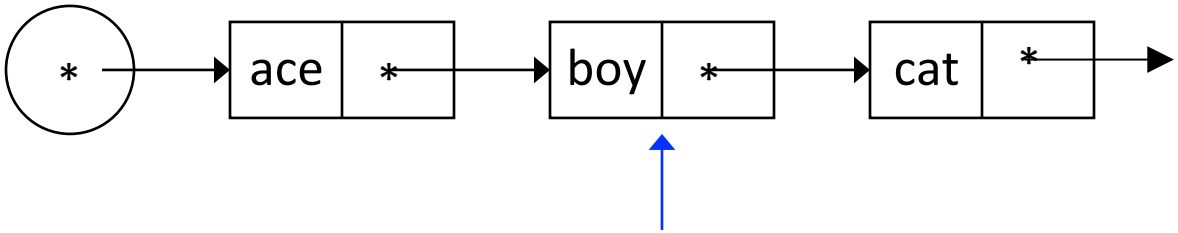
## Delete Operation

- To delete an item from a linked list:
  - find the node to delete in the list
  - "splice out" the node from the list, adjusting pointers as necessary
  - deallocate the memory for the deleted node
- Example:

Original list

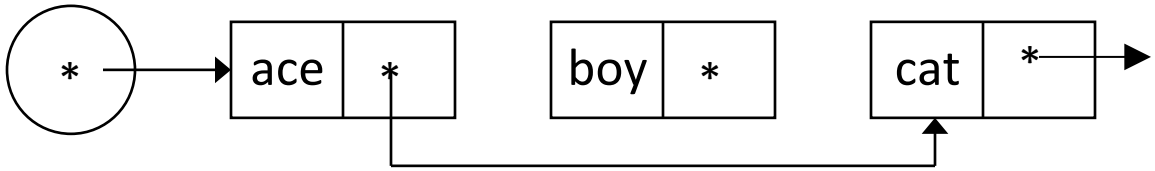



Find the node to delete

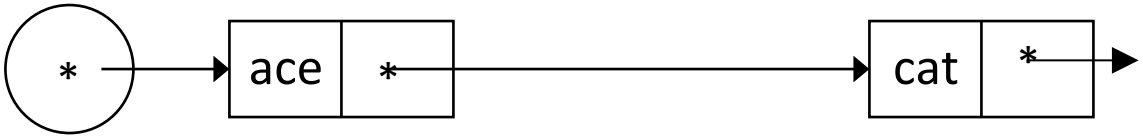


# Delete Operation (continued)

Splice out node by adjusting pointers




Free memory for deleted node




- Deletion at the beginning or end of a linked list is subtly different than deleting from the middle.
  - You will have to write special case code to handle one or both of these cases.

**Create Operation**

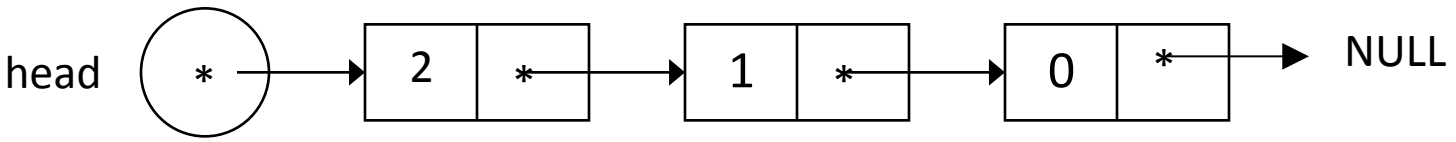
- A linked list can be created by doing successive inserts to the end of the list. For example using a while loop or for loop.
- The following example creates a linked list of integer numbers. The entire operation has been simplified in a main function to give a general idea on building, and using linked list. Pay attention to the code in the display function to traverse from one node to the next node. In other words to move the pointer to the next node, you have to use:

ptr = ptr ->next; **(NOT ptr++)**.

## Create Operation (continued)

```
int main(){
    int i;
    Node *head = NULL, *temp_ptr;
/*  Create first node manually  */
    head = new Node;
    head->data = 0;
    head->next = NULL;
/*  Create rest of nodes in a loop  */
    for (i = 1; i < 3; i++) {
        temp_ptr = New Node;
        temp_ptr->data = i;
        temp_ptr->next = head;
        head = temp_ptr;
}
    display(head);
    return 0;
}
```

**Result:**



## Traverse Operation

- Process each node in the list, from head to tail.
- Algorithm, using a loop:

  Check head pointer

  WHILE not at end of list

  Process node

  Check next pointer

```
void display (Node *ptr)
{
   while (ptr !=NULL){

     cout<< "data is:“ << ptr->data);
     ptr = ptr ->next;
   }
}
```

## Create a Class List

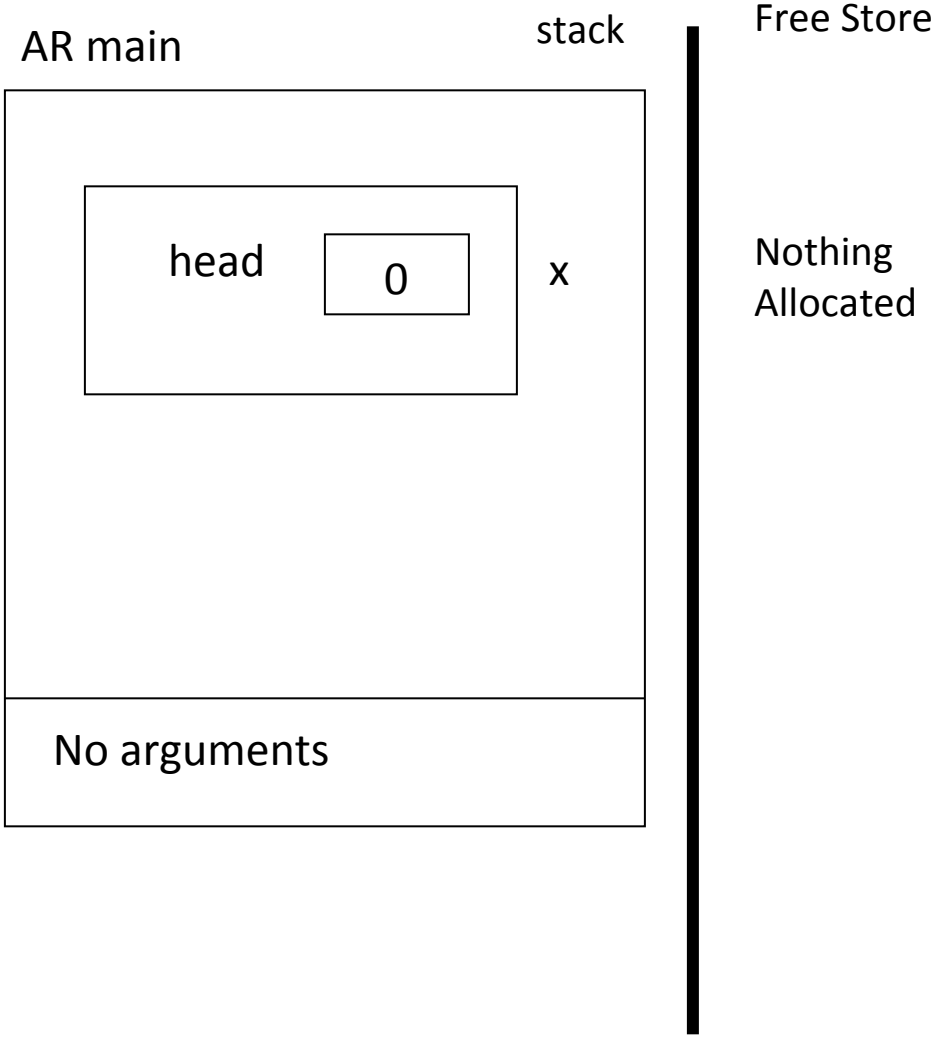
```
// File: list.h
class List {
        public:
                List();
                void insert (int data);
                void display()const;

                // many other operations

        private:
                 Node *head;
};
```

# List Operations

```
// File: list.cpp
#include <iostream.h>
#include <assert.h>

// List constructor
List::List(): head(0)
 {

 }

void List::insert(int n)
 {
        Node *newnode;
        newnode = new Node;
        assert (newnode != 0);
        newnode->data = n;
        newnode->next = head;
        head = newnode;
}
```

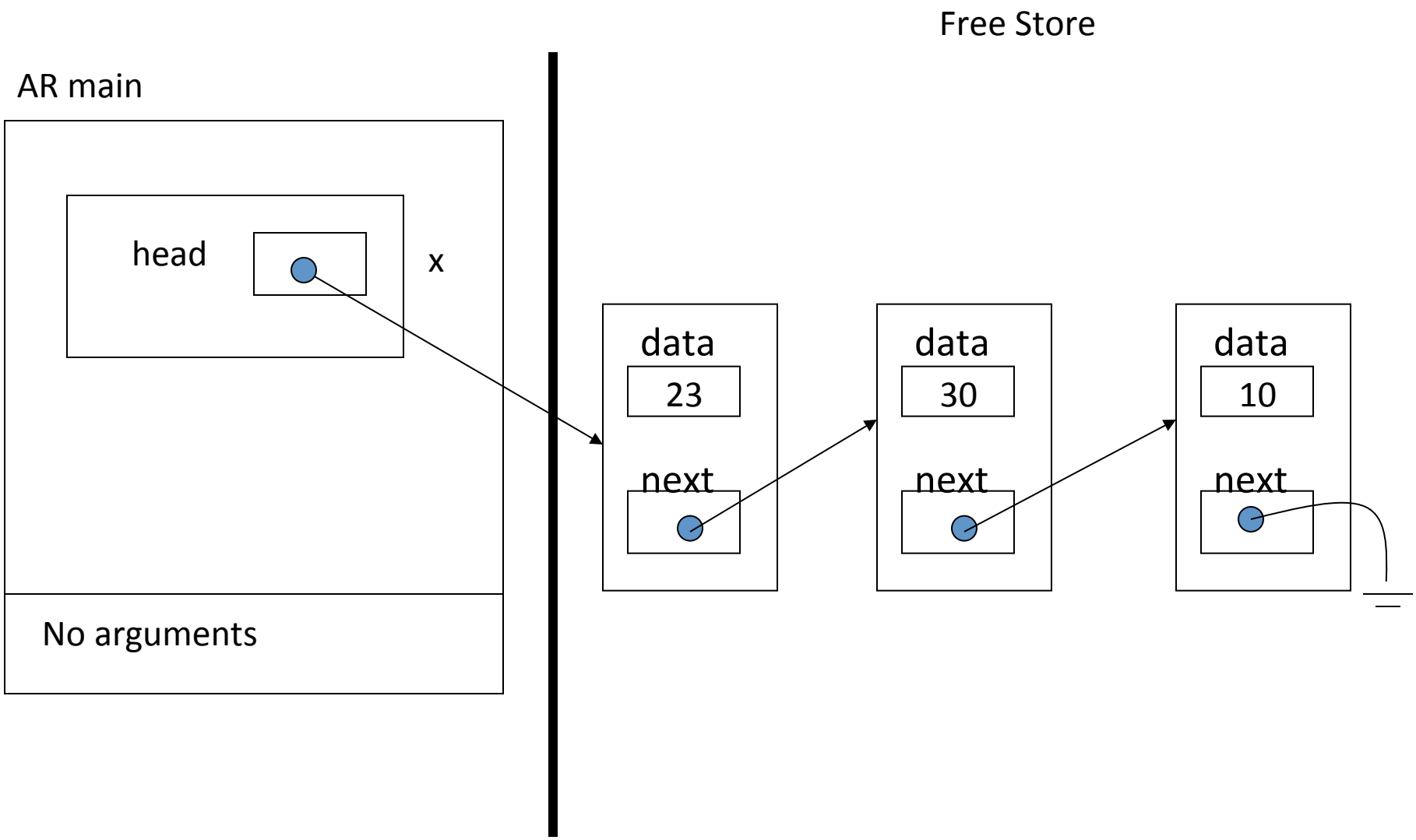
```
void List::display () const
{
   Node* ptr = head;
   cout << "\nList contains: ";
   while (ptr !=NULL){
    cout<< " " << ptr->data;
    ptr = ptr ->next;
  }
}
```

# AR at Point 1




```
#include "list.h"
int main()
{
        List x;
        // point 1

        x.insert(10);
        x.insert(30);
         x.insert(23);
         // point 2
         x.display();
          return 0;
}
```

# AR at Point 2


Free Store
AR main
head
x
data
23
next
data
30
next
data
10
next
No arguments

# An Ordered List

# How `insert` function in class OLList handout works:

- The following slides shows the process of inserting new nodes, step by step, according to given code at the top of each slide, and based on the given main function in the handout

```
struct Node {
  ListItem item;
  Node *next;
};
```

```
class OLList {
public:
  OLList();

  ~OLList();

  void insert(const ListItem& itemA);

  void remove(const ListItem& itemA);

  void print() const;

private:
  Node *headM;
};
```

```
OLList::OLList()
  : headM(0) {
}

OLList::~OLList() {
  destroy();
}

void OLList::print() const
{
  if (headM == 0)
    cout << "  LIST IS EMPTY.";
  else
    for (Node *p = headM; p != 0; p = p->next)
      cout << "  " << p->item << '¥n';
}
```

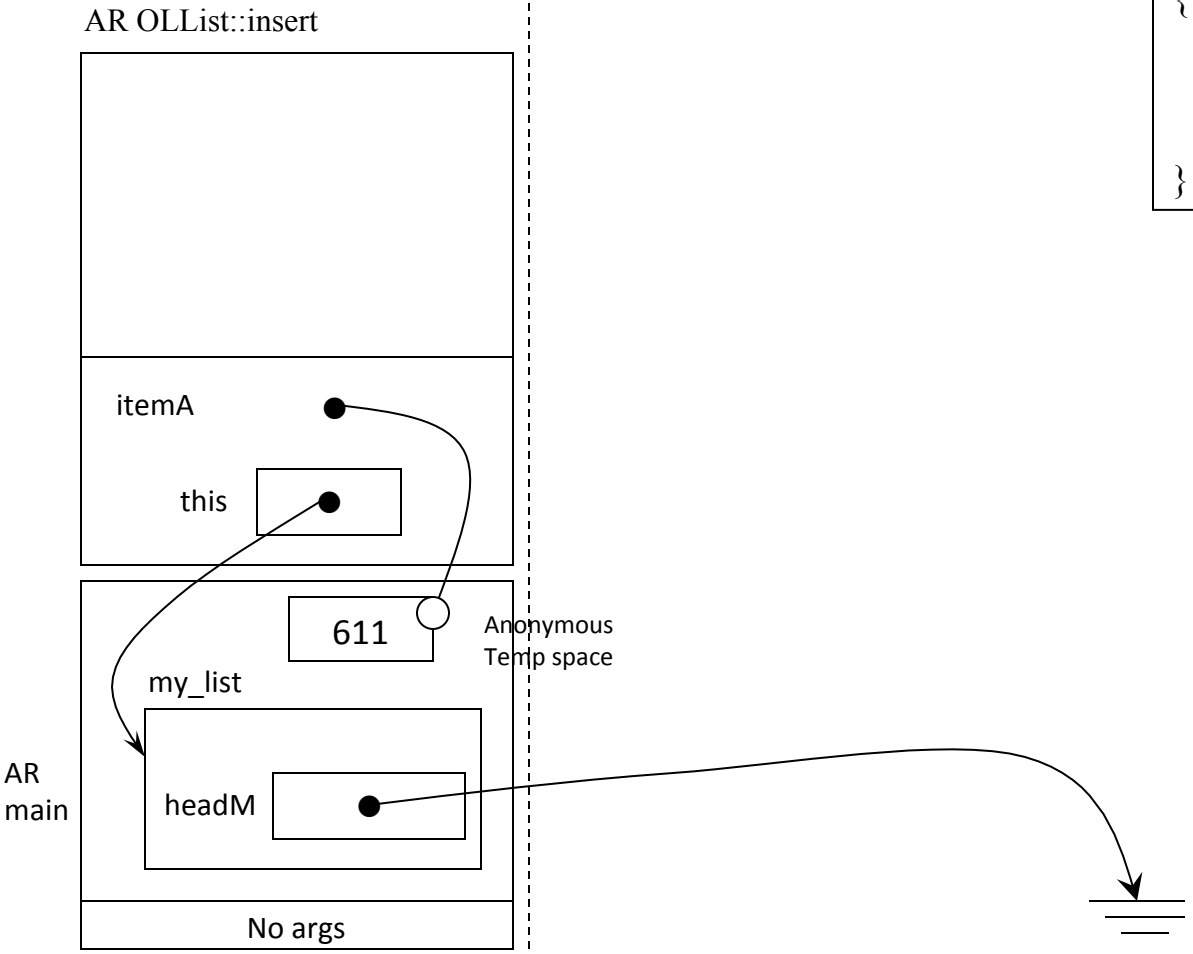
```
void OLList::insert(const ListItem& itemA)
{
  Node *new_node = new Node;
  new_node->item = itemA;

  if (headM == 0 || itemA <= headM->item) {
    new_node->next = headM;
    headM = new_node;
  }

    // MORE CODE GOES HERE

    new_node->next = after;
    before->next = new_node;
  }
}
```

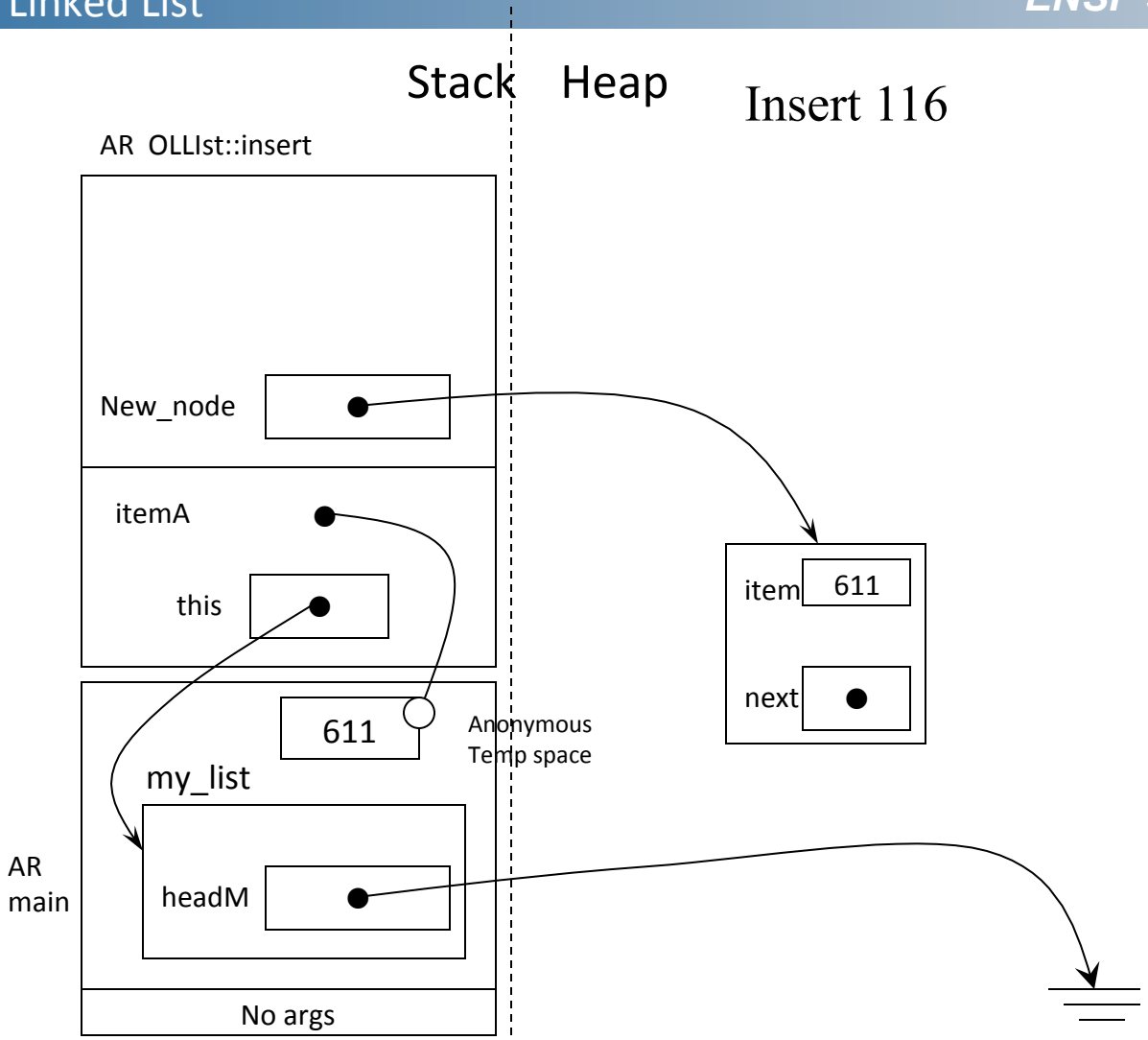
```
int main()
{
  OLList my_list;

  my_list.insert(611);
  my_list.insert(511);
  my_list.insert(811);
  my_list.insert(711);
  my_list.insert(411);

  cout << "List contents after 4 insertions:\n";
  my_list.print();
  return 0;
}
```

Stack Heap

Insert 116

```
void OLList::insert(const ListItem& itemA)
{
  // point


}
```

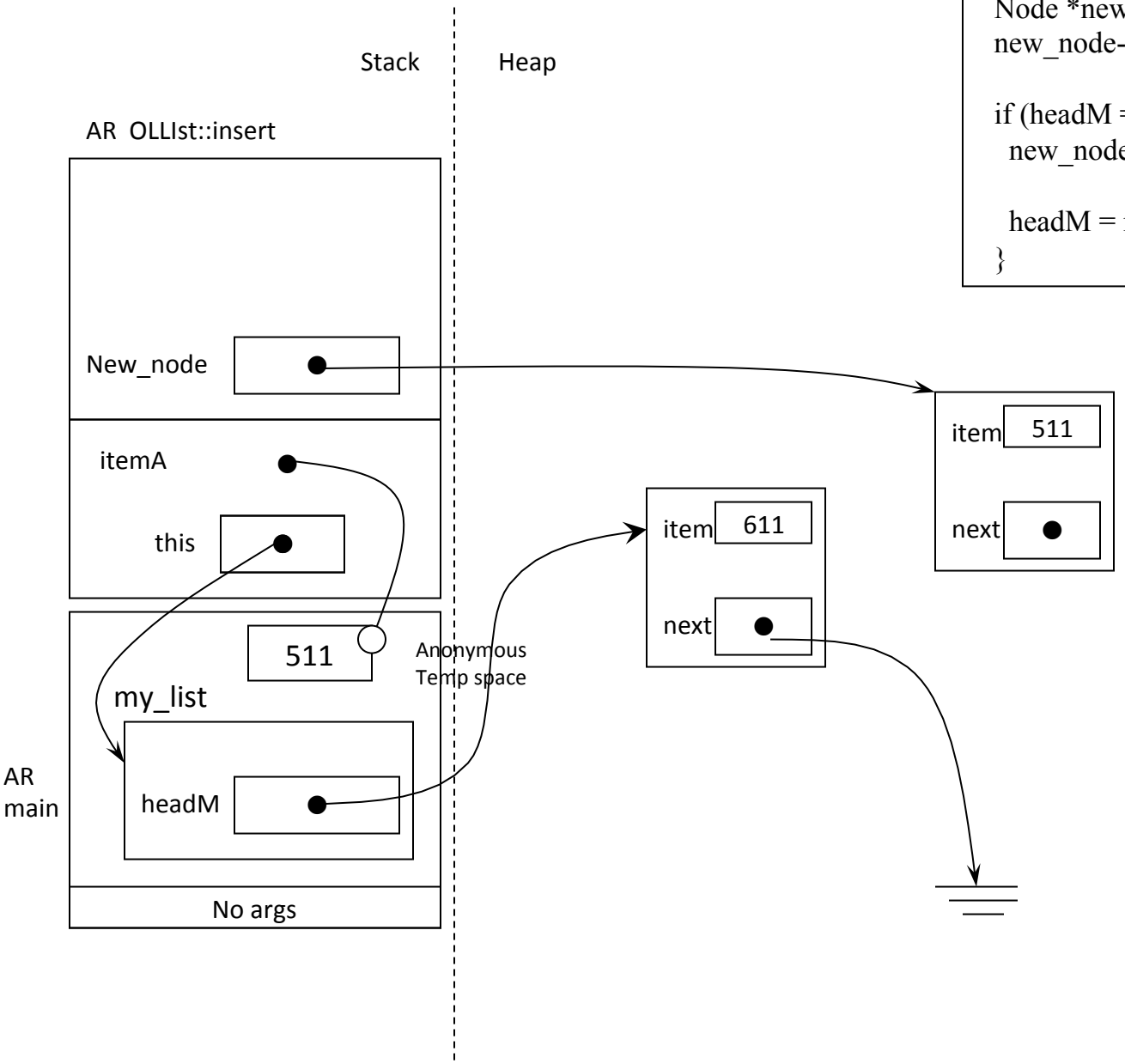


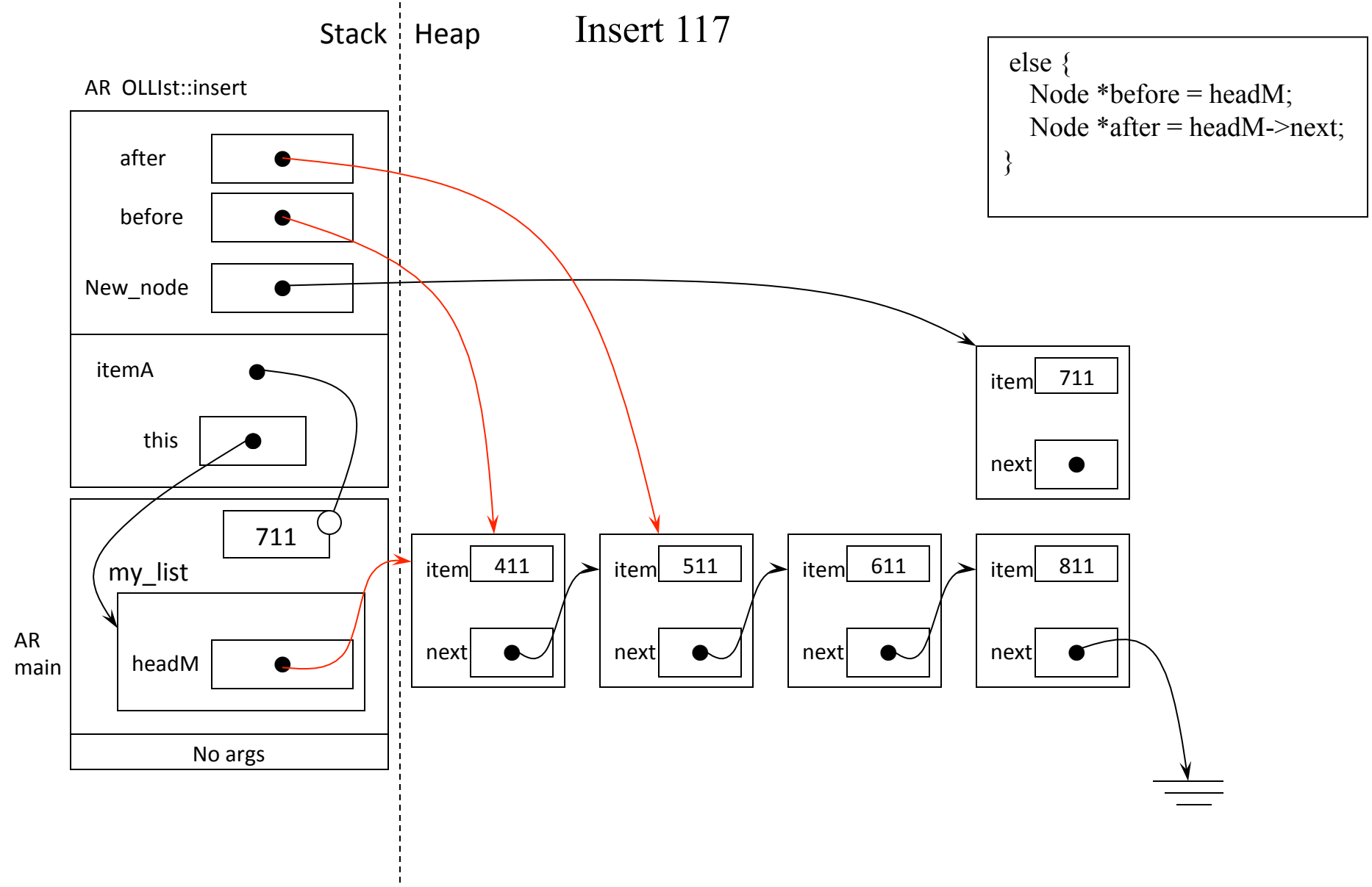


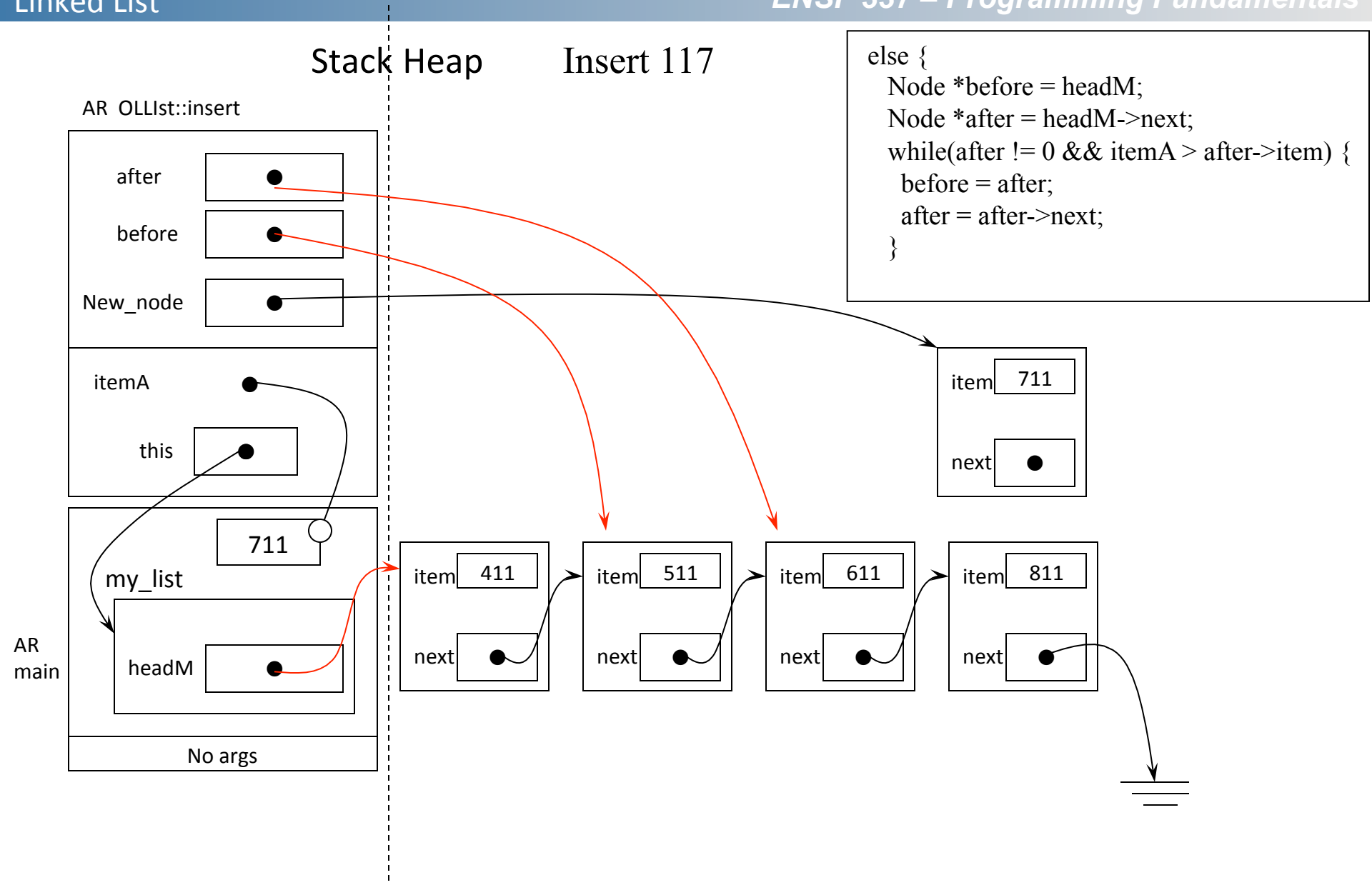

```
Node *new_node = new Node;
new_node->item = itemA;
```

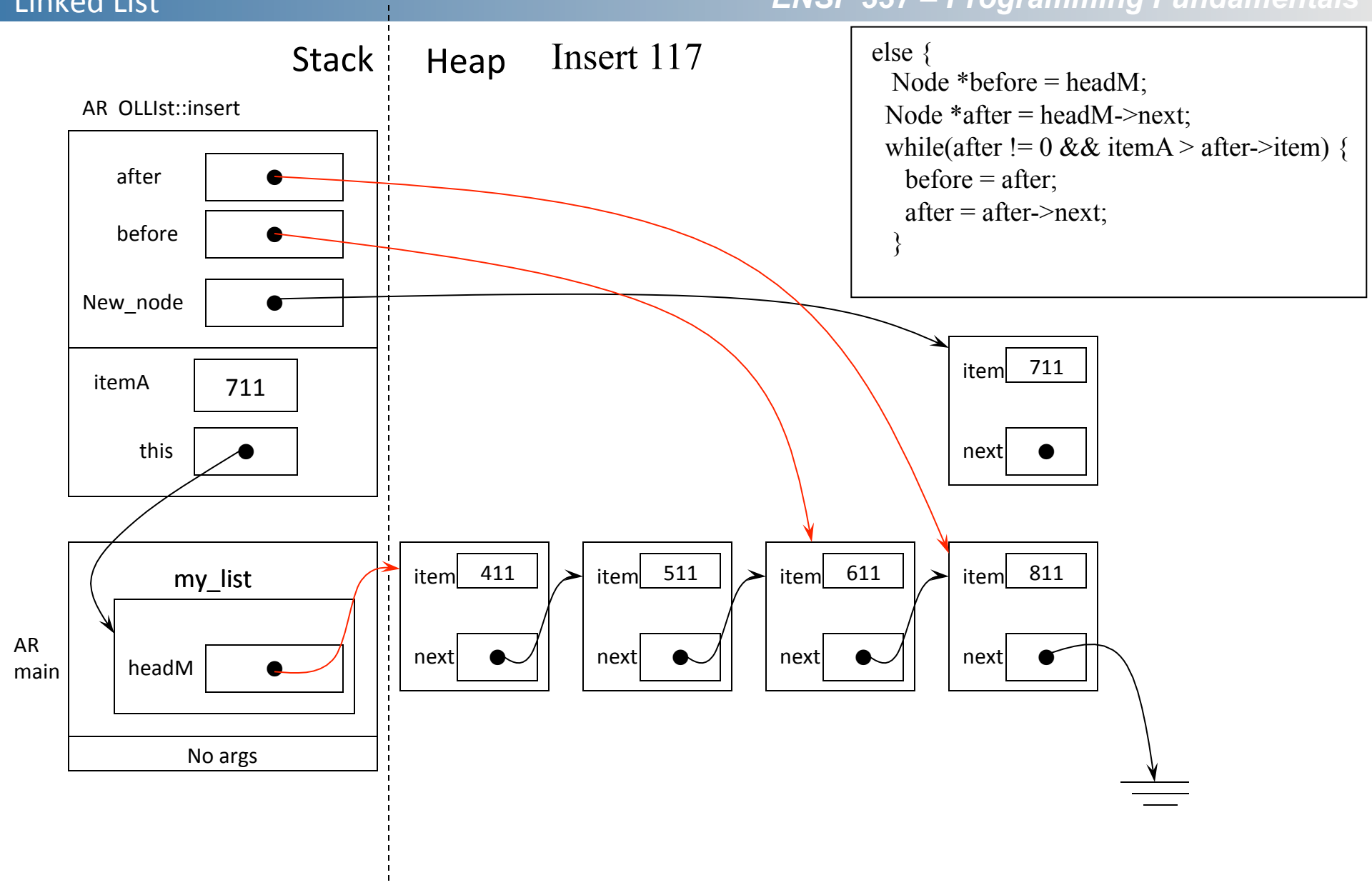
# Insert 115

```
void OLList::insert(const ListItem& itemA)
{
  Node *new_node = new Node;
  new_node->item = itemA;

  if (headM == 0 || itemA <= headM->item) {
    new_node->next = headM;

    headM = new_node;
  }
```

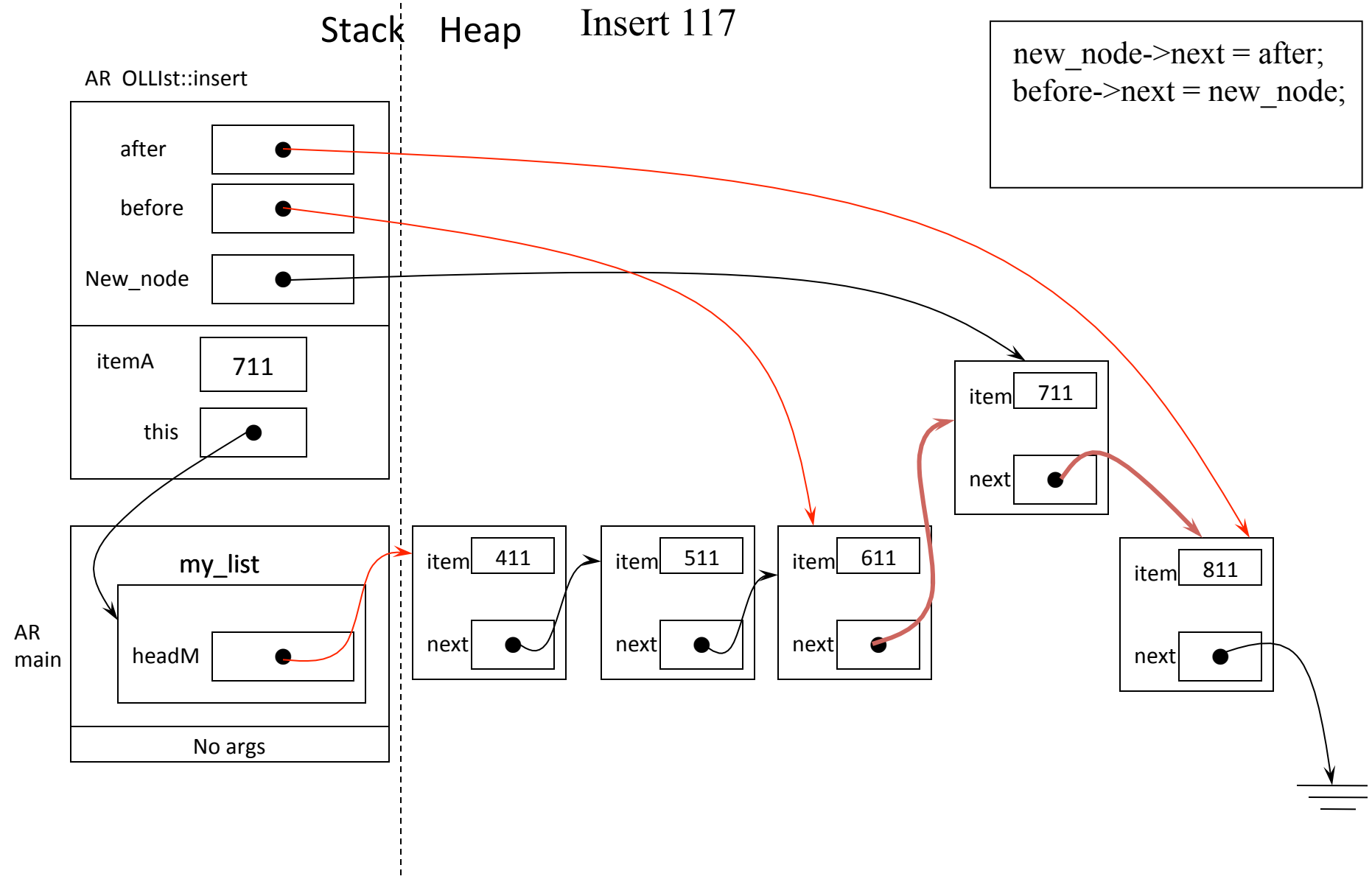


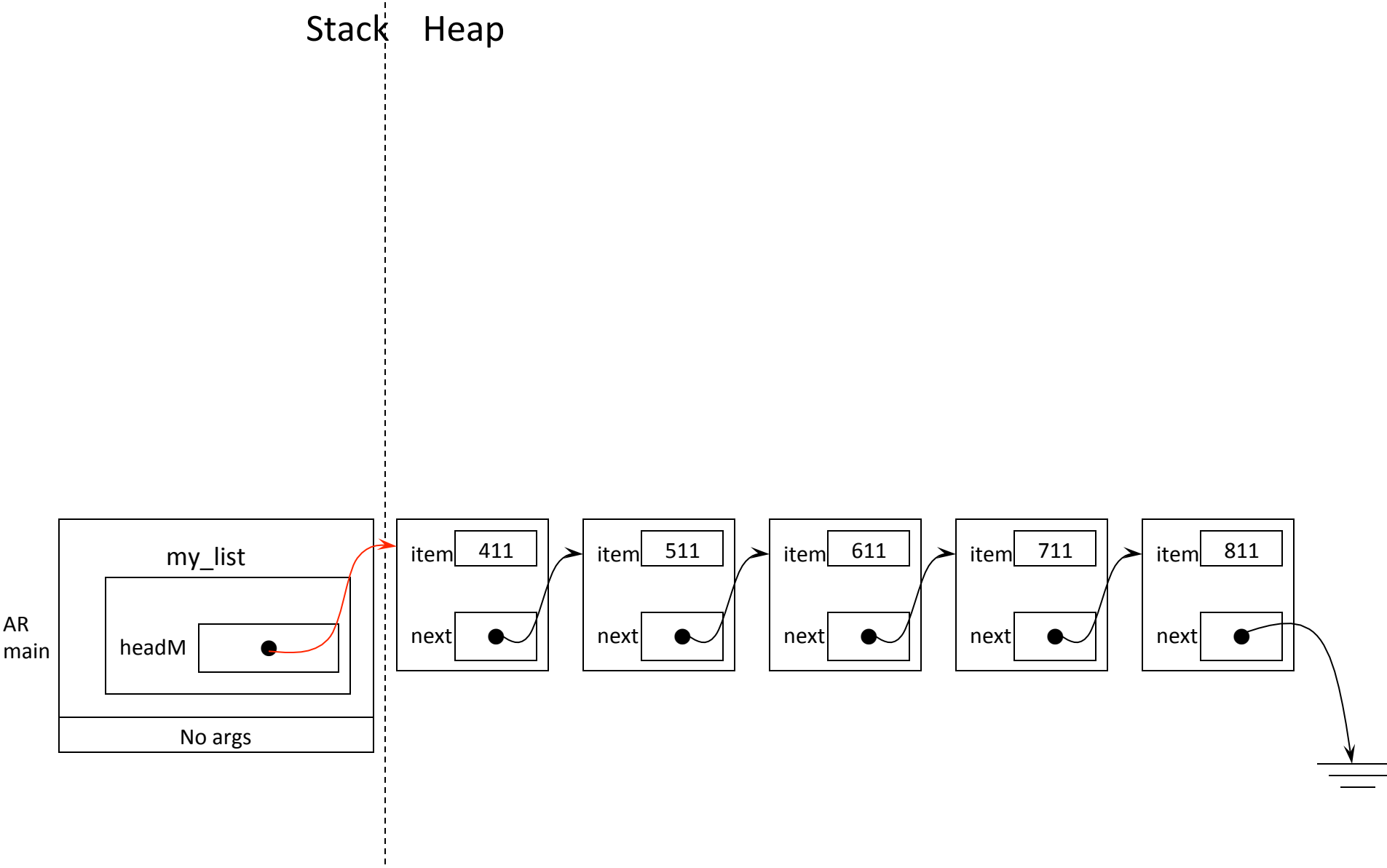

Stack
Heap
Insert 117
else {
Node *before = headM;
Node *after = headM->next;
}
AR OLLIst::insert
after
before
New_node
itemA
this
711
my_list
AR
main
headM
No args
item
711
next
item
411
next
item
511
next
item
611
next
item
811
next


Stack Heap
Insert 117
else {
Node *before = headM;
Node *after = headM->next;
while(after != 0 && itemA > after->item) {
before = after;
after = after->next;
}
AR OLLIst::insert
after
before
New_node
itemA
this
item 711
next
711
my_list
headM
AR
main
item 411
next
item 511
next
item 611
next
item 811
next
No args




```
else {
  Node *before = headM;
 Node *after = headM->next;
 while(after != 0 && itemA > after->item) {
   before = after;
   after = after->next;
 }
```


Stack
Heap
Insert 117
new_node->next = after;
before->next = new_node;
AR OLLIst::insert
after
before
New_node
itemA
711
this
item
711
next
my_list
headM
AR
main
item
411
next
item
511
next
item
611
next
item
811
next
No args


Stack
Heap
AR
main
my_list
headM
No args
item
411
next
item
511
next
item
611
next
item
711
next
item
811
next

**What are the issues with the design of class OLList in the previous slides**

- Answer: It fails if someone wants to make copies of the instances of this linked list.
- What is the solution?
  - Needs copy constructor
  - And, assignment operator

```
struct Node {
  ListItem item;
  Node *next;
};
```

```
class OLList {
public:
  OLList();

  ~OLList();

  void insert(const ListItem& itemA);
  OLList(const OLList& src);
  OLList& operator=(const OLList& rhs);
  void remove(const ListItem& itemA);

  void print() const;

private:
  Node *headM;
  void destroy();
  void copy(const OLList& source);
};
```

Now you should implement a copy-constructor and assignment-operator, using destroy and copy helper functions to fix the problem. Try it now.