About Reference Parameters

Program Under Consideration

```
#include "wrapper.h"
#include "Queue\Queue1.hpp"
int main(…)
{
   Queue1<Integer> q1;
   Integer k;

   // Code to enqueue onto q1
   // q1 = <3,17,73>
   k = 28;
   q1.enqueue(k);
   // outgoing q1 = <3,17,73,28>
}
```

The client program to the left does the following:

- Declares 1 queue variable and 1 Integer
- Has some code (not shown) to enqueue the values 3, 17, and 73 onto q1
- Finally, enqueues 28 onto q1

Main's Activation Record

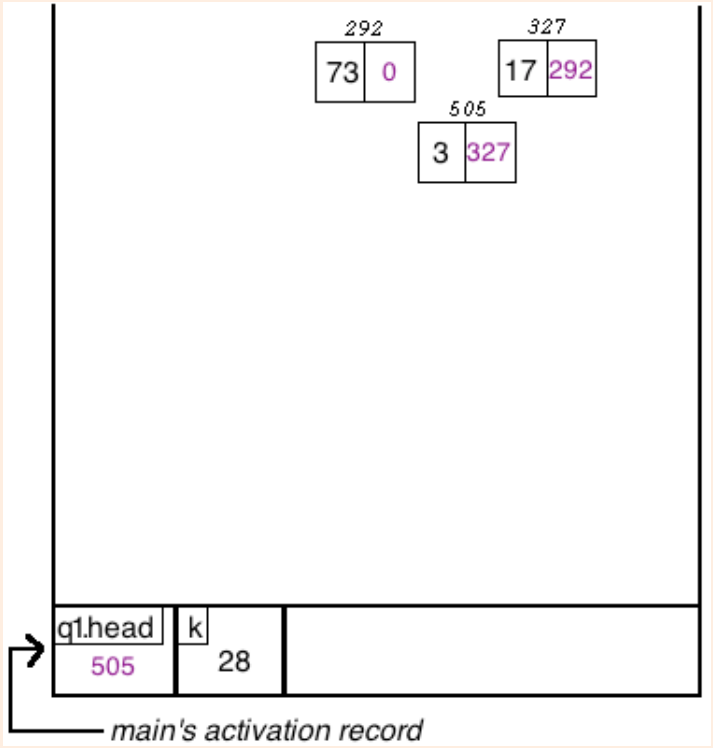
```
#include "wrapper.h"
#include "Queue\Queue1.hpp"
int main(…)
{
   Queue1<Integer> q1;
   Integer k;

   // Code to enqueue onto q1
   // q1 = <3,17,73>
   k = 28;
   q1.enqueue(k);
   // outgoing q1 = <3,17,73,28>
}
```

main's activation record diagramed below contains:

- Storage for two locally declared variables
- q1's *head* and *k*

- The activation record below illustrates the call stack and heap memory just prior to calling enqueue



Call to enqueue

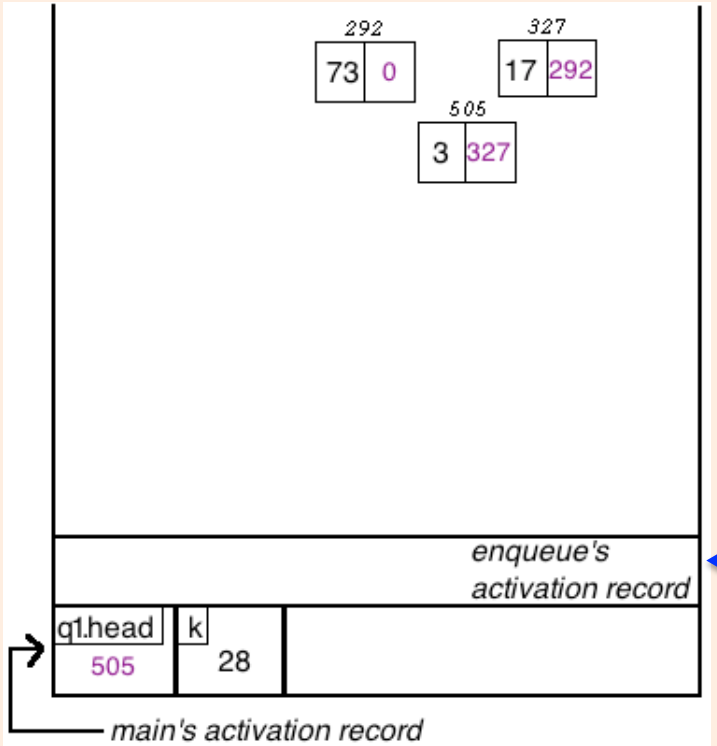
```
template <class T>
void Queue<T>::enqueue (T& item)
{
   insertAtEnd(head, item);
} // enqueue
```

```
#include "wrapper.h"
#include "Queue\Queue1.hpp"
int main(…)
{
   Queue1<Integer> q1;
   Integer k;

   // Code to enqueue onto q1
   // q1 = <3,17,73>
   k = 28;
   q1.enqueue(k);
   // outgoing q1 = <3,17,73,28>
}
```

Call to enqueue

enqueue's activation record is pushed onto the call stack



Enqueue's activation

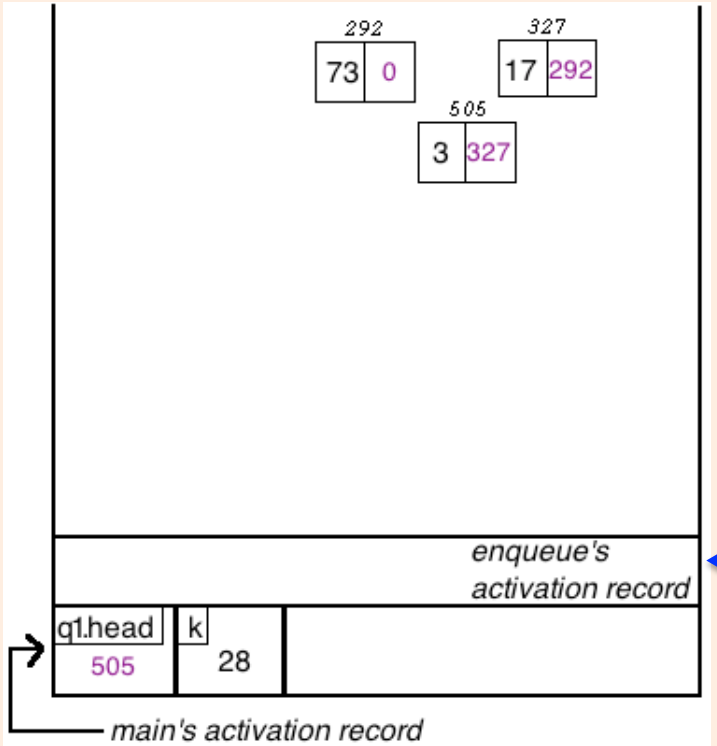
```
template <class T>
void Queue<T>::enqueue (T& item)
{
   insertAtEnd(head, item);
} // enqueue
```

- No value parameters
- No locally declared variables

- So no variables appear in enqueue's activation record

```
#include "wrapper.h"
#include "Queue\Queue1.hpp"
int main(…)
{
   Queue1<Integer> q1;
   Integer k;

   // Code to enqueue onto q1
   // q1 = <3,17,73>
   k = 28;
   q1.enqueue(k);
   // outgoing q1 = <3,17,73,28>
}
```



Enqueue's Reference Parameter

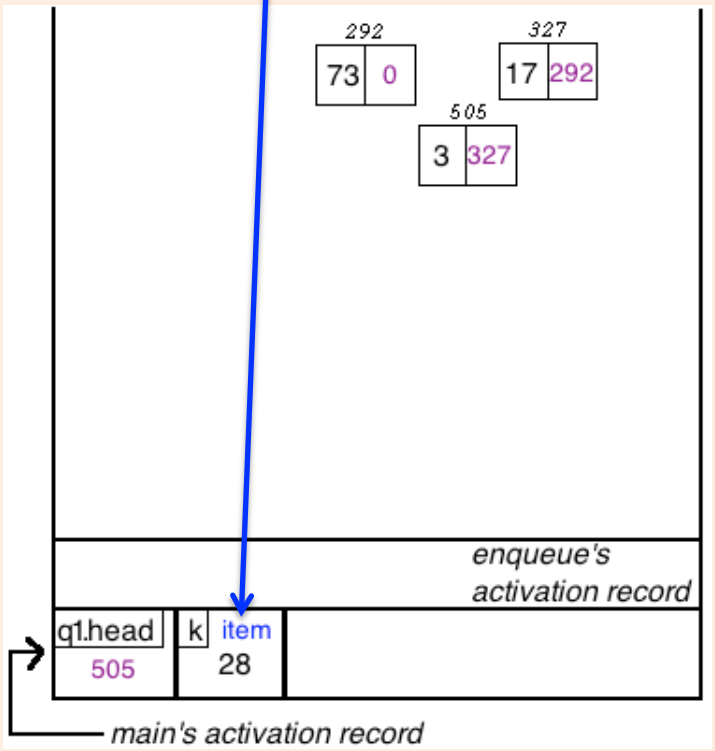
```
template <class T>
void Queue<T>::enqueue (T& item)
{
   insertAtEnd(head, item);
} // enqueue
```

```
#include "wrapper.h"
#include "Queue\Queue1.hpp"
int main(…)
{
   Queue1<Integer> q1;
   Integer k;

   // Code to enqueue onto q1
   // q1 = <3,17,73>
   k = 28;
   q1.enqueue(k);
   // outgoing q1 = <3,17,73,28>
}
```

- enqueue's formal reference parameter *item*
- The call's actual parameter *k*

- *item* references actual parameter *k*'s storage depicted by placing item in *k*'s storage in main's activation record



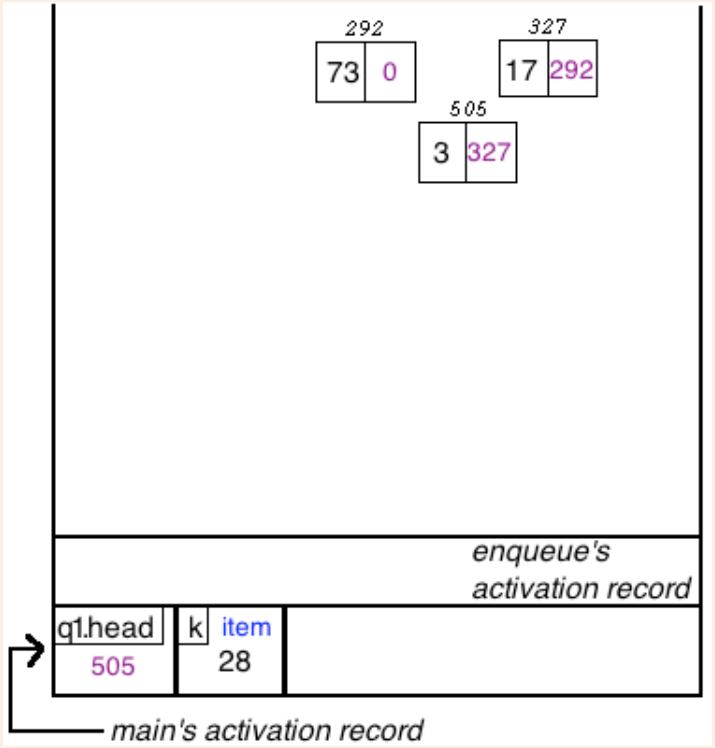
Meaning of Reference Parameter

```
template <class T>
void Queue<T>::enqueue (T& item)
{
   insertAtEnd(head, item);
} // enqueue
```

```
#include "wrapper.h"
#include "Queue\Queue1.hpp"
int main(…)
{
   Queue1<Integer> q1;
   Integer k;

   // Code to enqueue onto q1
   // q1 = <3,17,73>
   k = 28;
   q1.enqueue(k);
   // outgoing q1 = <3,17,73,28>
}
```

- Because *item* directly references the *k*'s storage, then any changes made to *item* will immediately change what is stored in *k*
- This is what it means to be a reference parameter, i.e., any changes made to the formal reference parameter immediately affect the actual parameter's storage



enqueue's implementation

- Looking at enqueue's implementation we see that it consists of a call to the operation insertAtEnd
- This makes enqueue a client of insertAtEnd

```
template <class T>
void Queue<T>::enqueue (T& item)
{
   insertAtEnd(head, item);
} // enqueue
```

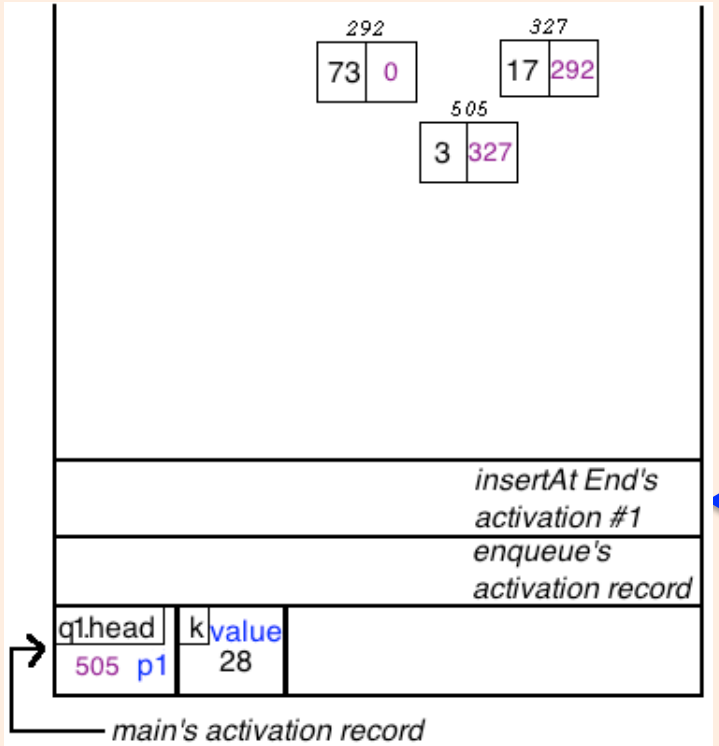
Call to insertAtEnd

```
template <class T>
void Queue<T>::insertAtEnd (
      NodeRecord*& p,
      T& value
   )
{
   if (p == NULL) {
      p = new NodeRecord;
      p->value.transferFrom(value);
      p->next = NULL;
   } else {
      insertAtEnd(p->next, value);
   } // end if
} // insertAtEnd
```

```
template <class T>
void Queue<T>::enqueue (T& item)
{
   insertAtEnd(head, item);
} // enqueue
```

Call to insertAtEnd

insertAtEnd's activation record is pushed onto the call stack



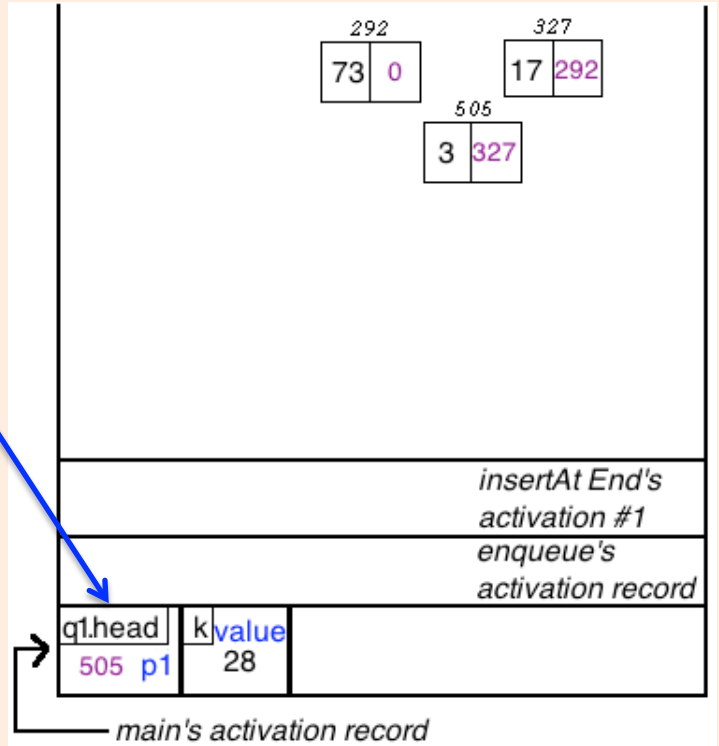
insertAtEnd's Reference Parameters

```
template <class T>
void Queue<T>::insertAtEnd (
      NodeRecord*& p,
      T& value
   )
{
   if (p == NULL) {
      p = new NodeRecord;
      p->value.transferFrom(value);
      p->next = NULL;
   } else {
      insertAtEnd(p->next, value);
   } // end if
} // insertAtEnd
```

```
template <class T>
void Queue<T>::enqueue (T& item)
{
   insertAtEnd(head, item);
} // enqueue
```

- *p* references actual parameter *head* so *p* references *head*'s storage in main's activation record

- This is depicted by *p1* appearing in *head*'s storage in main's activation record
- We use *p1* because insertAtEnd is a recursive operation and this is the 1st activation of insertAtEnd
- Activations 2, 3, 4, etc., will use *p2*, *p3*, *p4*, etc.



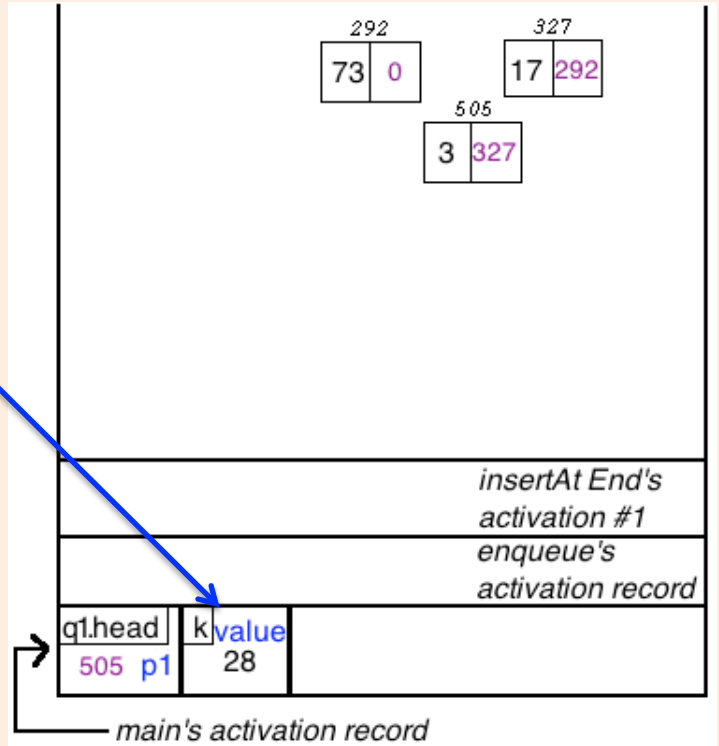
insertAtEnd's Reference Parameters

```
template <class T>
void Queue<T>::insertAtEnd (
      NodeRecord*& p,
      T& value
   )
{
   if (p == NULL) {
      p = new NodeRecord;
      p->value.transferFrom(value);
      p->next = NULL;
   } else {
      insertAtEnd(p->next, value);
   } // end if
} // insertAtEnd
```

```
template <class T>
void Queue<T>::enqueue (T& item)
{
   insertAtEnd(head, item);
} // enqueue
```

- *value* references actual parameter *item*
- But recall that *item* referenced *k*'s storage in main's activation, so *value* now references *k*'s storage

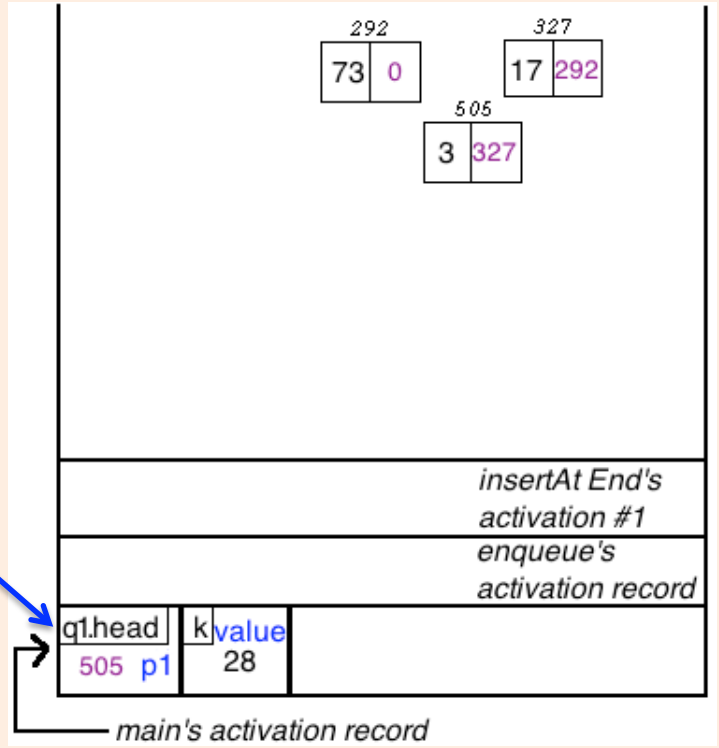
- This is depicted by *value* appearing in *k*'s storage area in main's activation record



Tracing insertAtEnd

```
template <class T>
void Queue<T>::insertAtEnd (
      NodeRecord*& p,
      T& value
   )
{
   if (p == NULL) {
      p = new NodeRecord;
      p->value.transferFrom(value);
      p->next = NULL;
   } else {
      insertAtEnd(p->next, value);
   } // end if
} // insertAtEnd
```

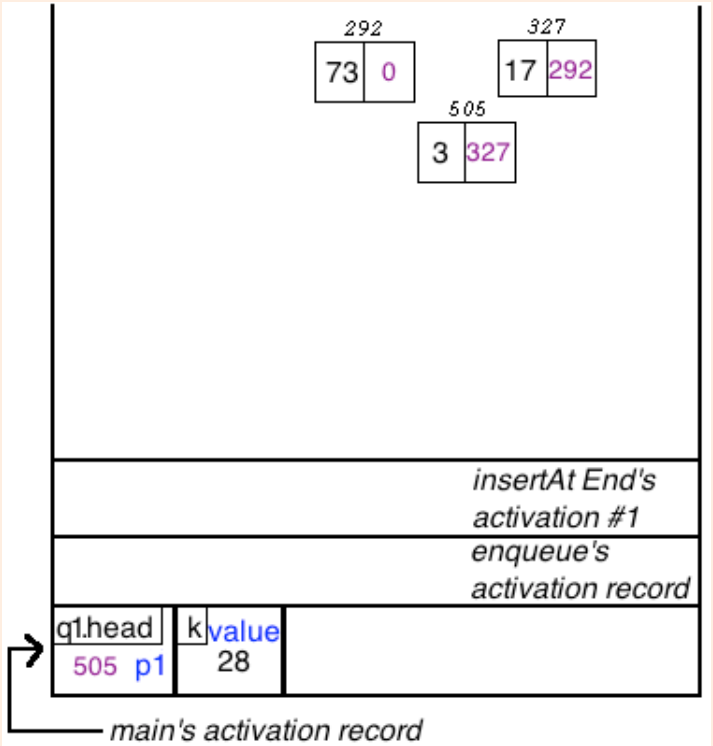
- The *if* determines if the base case has been reached
- We look at the diagram for *p1* in order to evaluate p == NULL
- We see that address 505 is stored at the location referenced by *p1*, so p == NULL evaluates to false



Tracing insertAtEnd

```
template <class T>
void Queue<T>::insertAtEnd (
      NodeRecord*& p,
      T& value
   )
{
   if (p == NULL) {
      p = new NodeRecord;
      p->value.transferFrom(value);
      p->next = NULL;
   } else {
      insertAtEnd(p->next, value);
   } // end if
} // insertAtEnd
```

- Since p == NULL evaluates to false, insertAtEnd is not at the base case
- So the recursive call is made from the *else* branch of the *if*

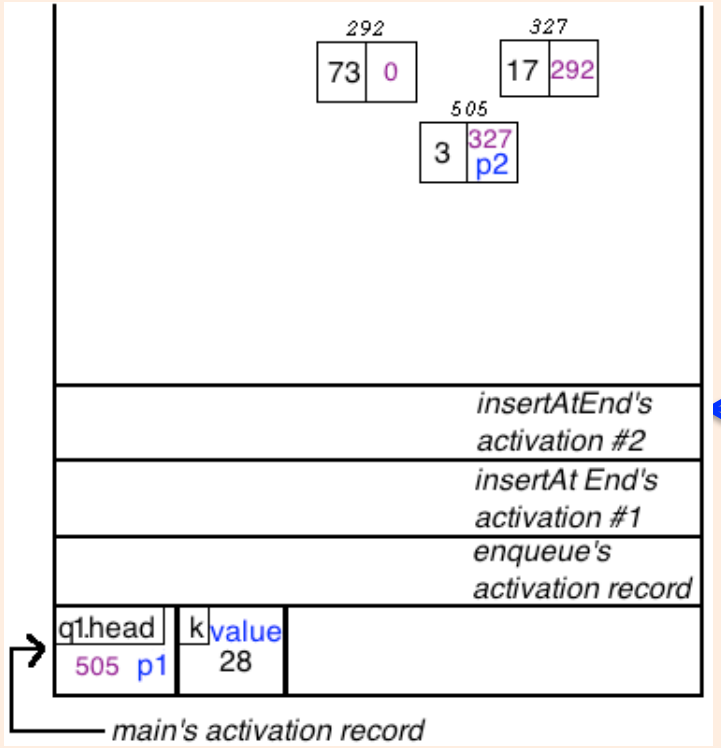


Recursive Call to insertAtEnd

```
template <class T>
void Queue<T>::insertAtEnd (
      NodeRecord*& p,
      T& value
   )
{
   if (p == NULL) {
      p = new NodeRecord;
      p->value.transferFrom(value);
      p->next = NULL;
   } else {
      insertAtEnd(p->next, value);
   } // end if
} // insertAtEnd
```

Recursive call to insertAtEnd

insertAtEnd's activation record #2 is pushed onto the call stack

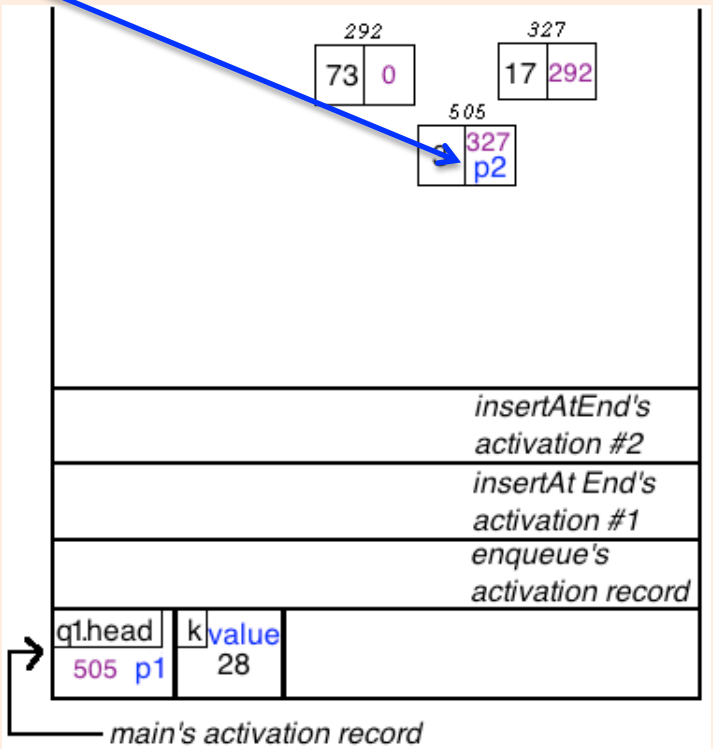


insertAtEnd's Reference Parameters

```
template <class T>
void Queue<T>::insertAtEnd (
      NodeRecord*& p,
      T& value
   )
{
   if (p == NULL) {
      p = new NodeRecord;
      p->value.transferFrom(value);
      p->next = NULL;
   } else {
      insertAtEnd(p->next, value);
   } // end if
} // insertAtEnd
```

```
template <class T>
void Queue<T>::enqueue (T& item)
{
   insertAtEnd(head, item);
} // enqueue
```

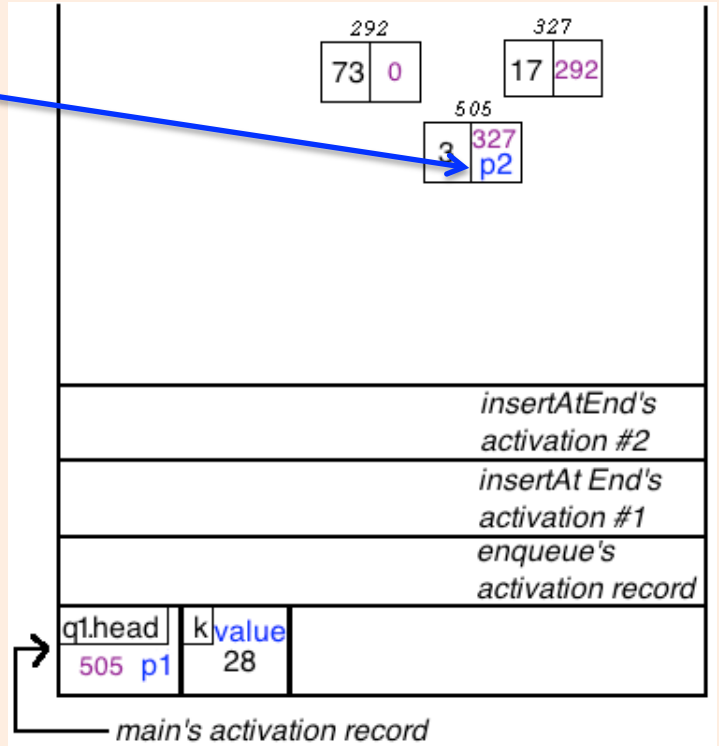
- *p* in activation #2 (or *p2*) references actual parameter *p->next*
- But *p->next* is from activation #1, so this is really *p1->next*
- *p1* holds the address 505, so *p2* references the 505's *next* field



Tracing insertAtEnd

```
template <class T>
void Queue<T>::insertAtEnd (
      NodeRecord*& p,
      T& value
   )
{
   if (p == NULL) {
      p = new NodeRecord;
      p->value.transferFrom(value);
      p->next = NULL;
   } else {
      insertAtEnd(p->next, value);
   } // end if
} // insertAtEnd
```

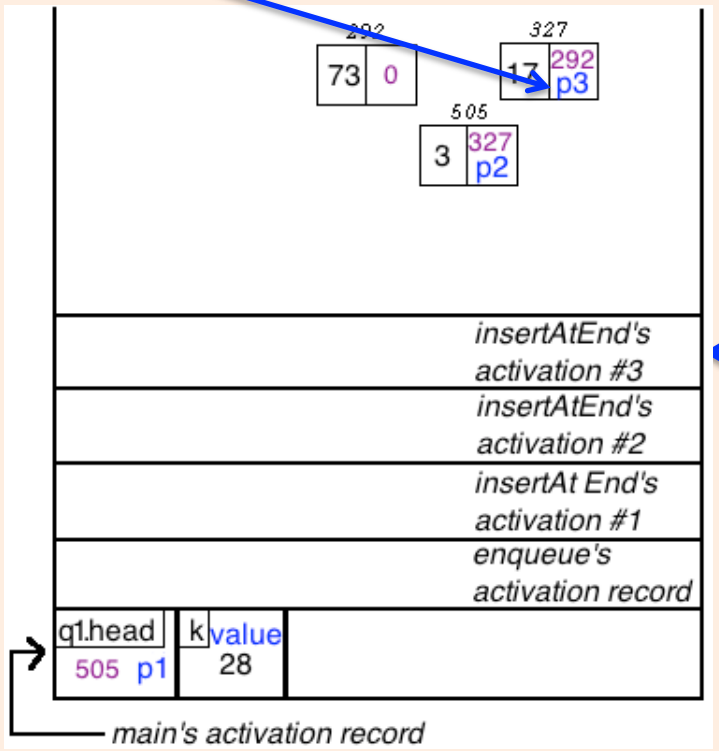
- To evaluate p == NULL we look in the diagram for *p2* because now we are in activation #2
- We see that *p2* holds the address 327, so p == NULL evaluates to false
- A 2nd recursive call is made from the *else* branch



Recursive Call to insertAtEnd

```
template <class T>
void Queue<T>::insertAtEnd (
      NodeRecord*& p,
      T& value
   )
{
   if (p == NULL) {
      p = new NodeRecord;
      p->value.transferFrom(value);
      p->next = NULL;
   } else {
      insertAtEnd(p->next, value);
   } // end if
} // insertAtEnd
```

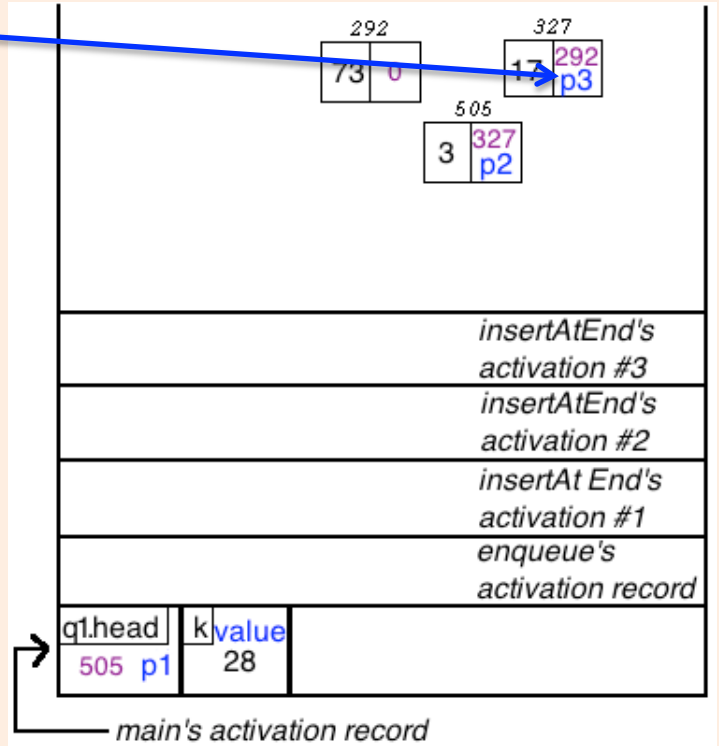
- insertAtEnd's activation record #3 is pushed onto the call stack
- *p3* references *p2->next*, but recall that *p2* holds the address 327, so *p3* references *327->next* field



Tracing insertAtEnd

```
template <class T>
void Queue<T>::insertAtEnd (
      NodeRecord*& p,
      T& value
   )
{
   if (p == NULL) {
      p = new NodeRecord;
      p->value.transferFrom(value);
      p->next = NULL;
   } else {
      insertAtEnd(p->next, value);
   } // end if
} // insertAtEnd
```

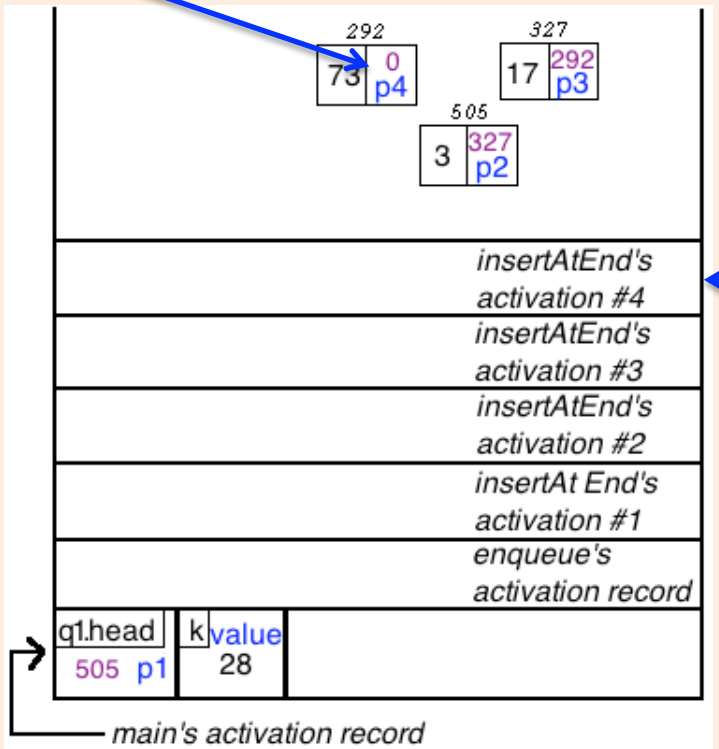
- To evaluate p == NULL we look in the diagram for *p3* because now we are in activation #3
- We see that *p3* holds the address 292, so p == NULL evaluates to false
- A 3rd recursive call is made from the *else* branch



Recursive Call to insertAtEnd

```
template <class T>
void Queue<T>::insertAtEnd (
      NodeRecord*& p,
      T& value
   )
{
   if (p == NULL) {
      p = new NodeRecord;
      p->value.transferFrom(value);
      p->next = NULL;
   } else {
      insertAtEnd(p->next, value);
   } // end if
} // insertAtEnd
```

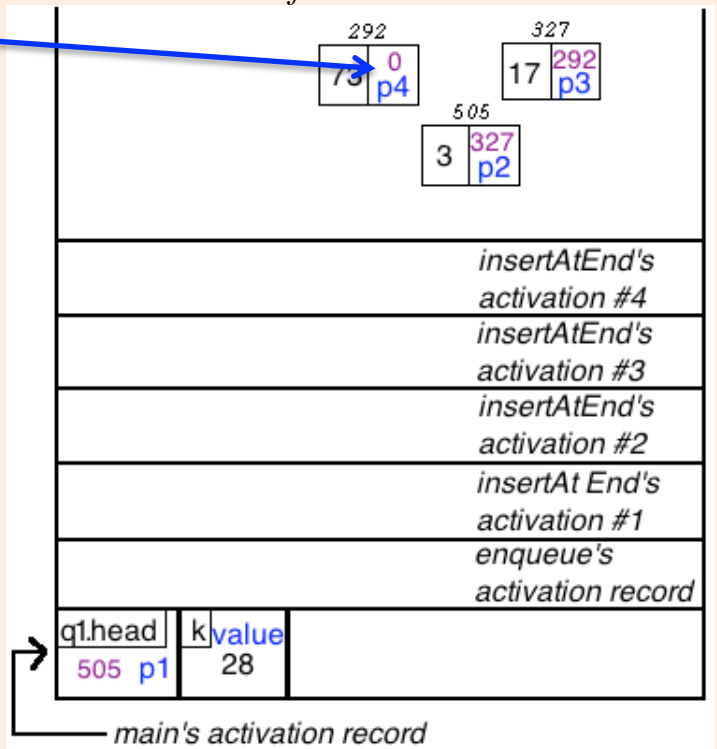
- insertAtEnd's activation record #4 is pushed onto the call stack
- *p4* references *p3->next*, but recall that *p3* holds the address 292, so *p4* references *292->next* field



Tracing insertAtEnd

```
template <class T>
void Queue<T>::insertAtEnd (
      NodeRecord*& p,
      T& value
   )
{
   if (p == NULL) {
      p = new NodeRecord;
      p->value.transferFrom(value);
      p->next = NULL;
   } else {
      insertAtEnd(p->next, value);
   } // end if
} // insertAtEnd
```

- To evaluate p == NULL we look in the diagram for *p4* because now we are in activation #4
- We see that NULL (depicted as 0) is stored at the location referenced by *p4*, so p == NULL evaluates to true
- We have arrived at the base case and take the *then* branch of the *if*

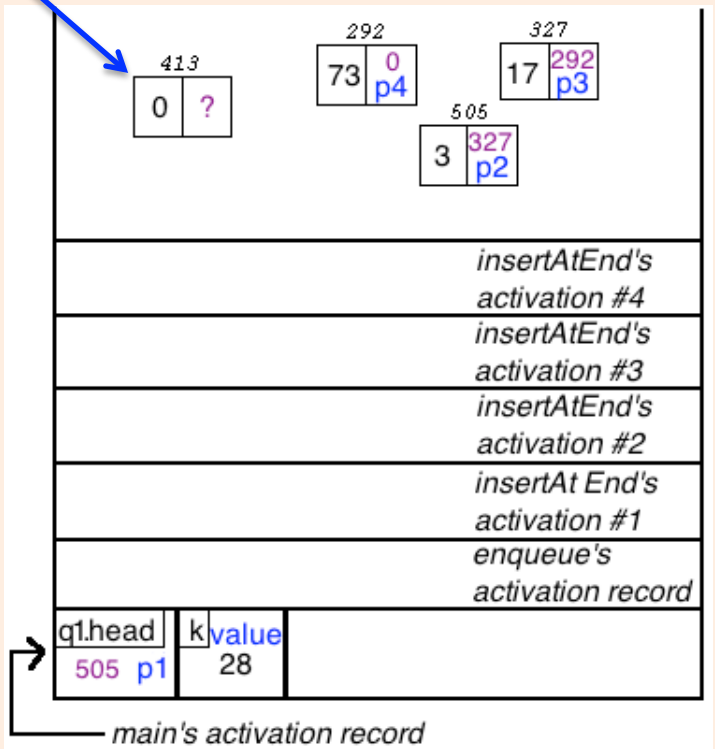


Tracing insertAtEnd – Base Case

```
template <class T>
void Queue<T>::insertAtEnd (
      NodeRecord*& p,
      T& value
   )
{
   if (p == NULL) {
      p = new NodeRecord;
      p->value.transferFrom(value);
      p->next = NULL;
   } else {
      insertAtEnd(p->next, value);
   } // end if
} // insertAtEnd
```

new is called and allocates a node

- In the diagram this node has the address 413
- Its *value* field is automatically initialized to zero by Integer's constructor
- Its *next* field is a Dead pointer and is depicted by ?

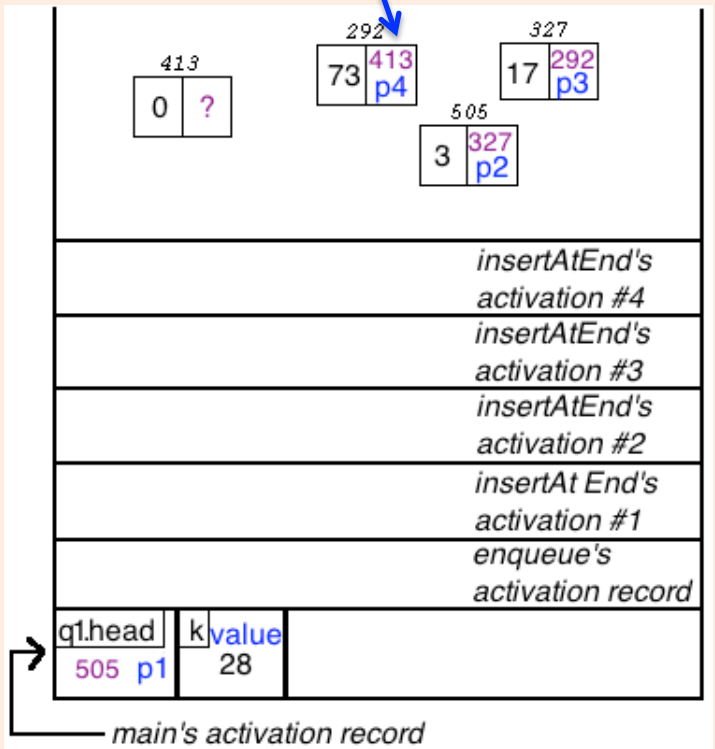


Tracing insertAtEnd – Base Case

```
template <class T>
void Queue<T>::insertAtEnd (
      NodeRecord*& p,
      T& value
   )
{
   if (p == NULL) {
      p = new NodeRecord;
      p->value.transferFrom(value);
      p->next = NULL;
   } else {
      insertAtEnd(p->next, value);
   } // end if
} // insertAtEnd
```

new returns the address 413 and it is stored in *p*

- But this is activation #4, so it is stored in the memory location referenced by *p4*

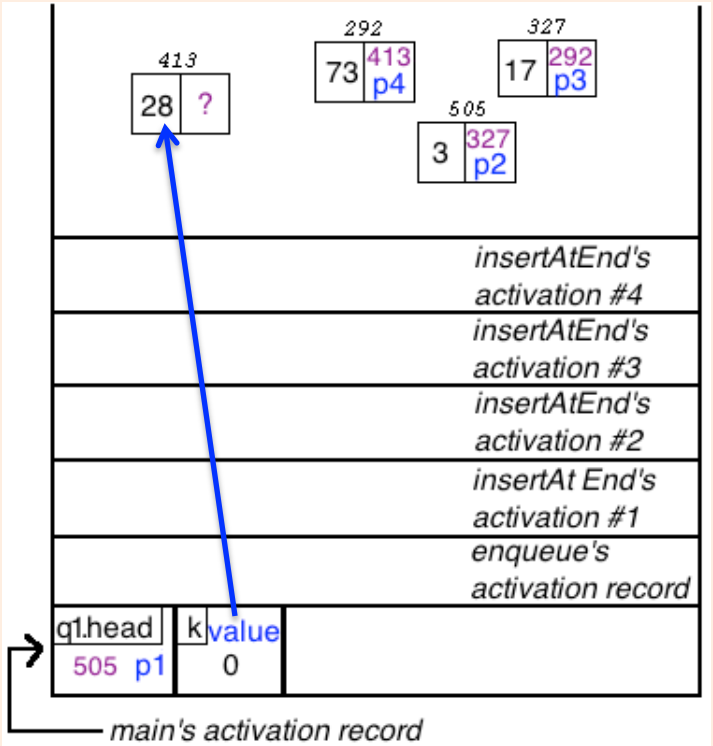


Tracing insertAtEnd – Base Case

```
template <class T>
void Queue<T>::insertAtEnd (
      NodeRecord*& p,
      T& value
   )
{
   if (p == NULL) {
      p = new NodeRecord;
      p->value.transferFrom(value);
      p->next = NULL;
   } else {
      insertAtEnd(p->next, value);
   } // end if
} // insertAtEnd
```

In the tranferFrom line, *p->value* references the 413 node's *value* field (because *p4* holds the address 413) and *value* references *k*'s storage down in main's activation

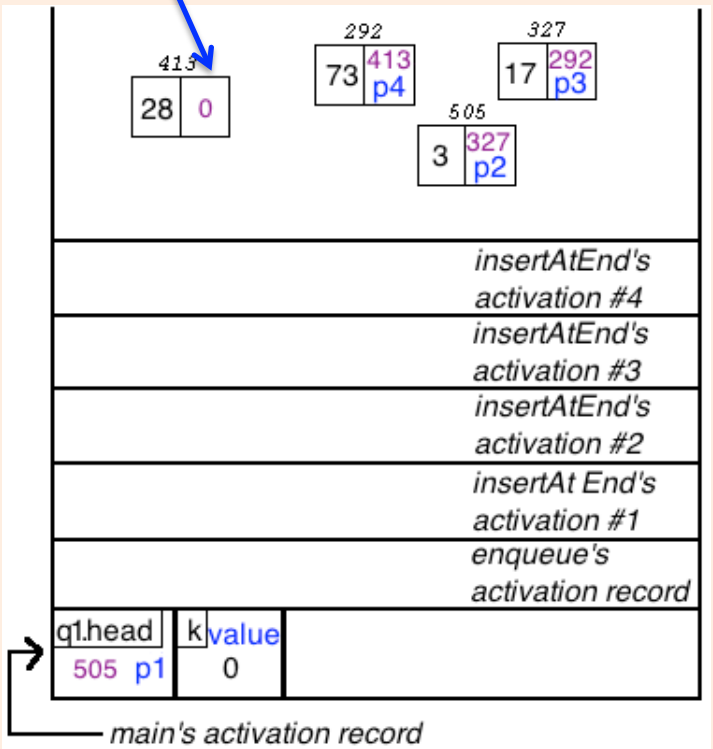
- So the transferFrom transfers the 28 from *k*'s memory to the *value* field of the 413 node, leaving *k* with zero (the initial value for type Integer)



Tracing insertAtEnd – Base Case

```
template <class T>
void Queue<T>::insertAtEnd (
      NodeRecord*& p,
      T& value
   )
{
   if (p == NULL) {
      p = new NodeRecord;
      p->value.transferFrom(value);
      p->next = NULL;
   } else {
      insertAtEnd(p->next, value);
   } // end if
} // insertAtEnd
```

On the line `p->next = NULL`, *p->next* references the 413 node's *next* field (because *p4* holds the address 413), so NULL is assigned to this field



Returning From the Recursive Calls

```
template <class T>
void Queue<T>::insertAtEnd (
      NodeRecord*& p,
      T& value
   )
{
   if (p == NULL) {
      p = new NodeRecord;
      p->value.transferFrom(value);
      p->next = NULL;
   } else {
      insertAtEnd(p->next, value);
   } // end if
} // insertAtEnd
```

- insertAtEnd is a *tail recursive* operation, that is, there is *no code to execute* after the recursive call
- So each of the 4 activations returns, first #4 returns to #3, then #3 to #2, #2 to #1, and #1 to enqueue
- But enqueue has nothing to do after calling insertAtEnd, so it immediately returns to main leaving the call stack and heap as depicted below