CS302 - Data Structures

using C++

Topic: Linked Lists – Core LinkedBag Methods

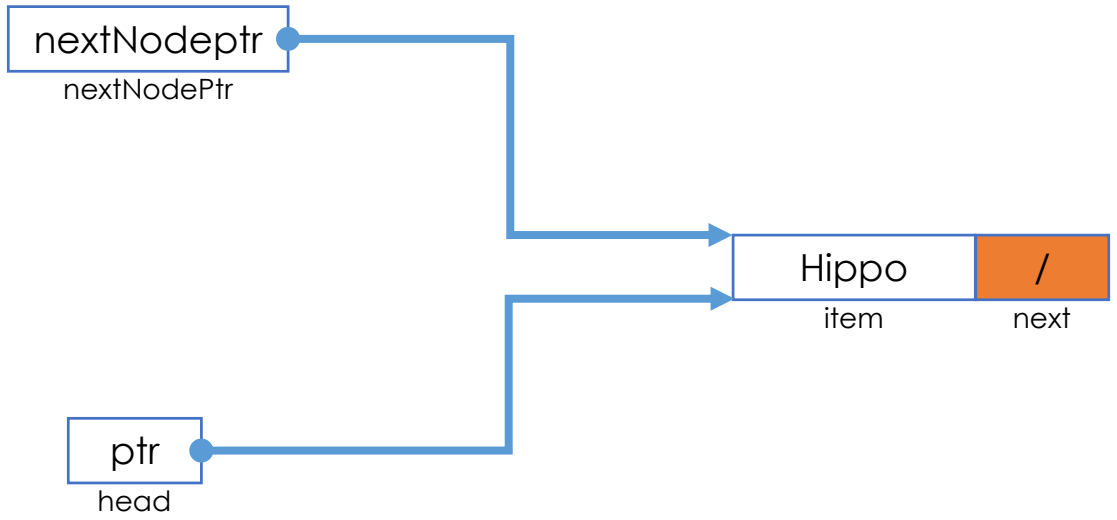
Kostas Alexis

Implementing Core Methods

- We have to define and implement the core methods of the LinkedBag, namely:
  - the default constructor
  - add
  - toVector
  - getCurrentSize
  - isEmpty

Implementing Core Methods

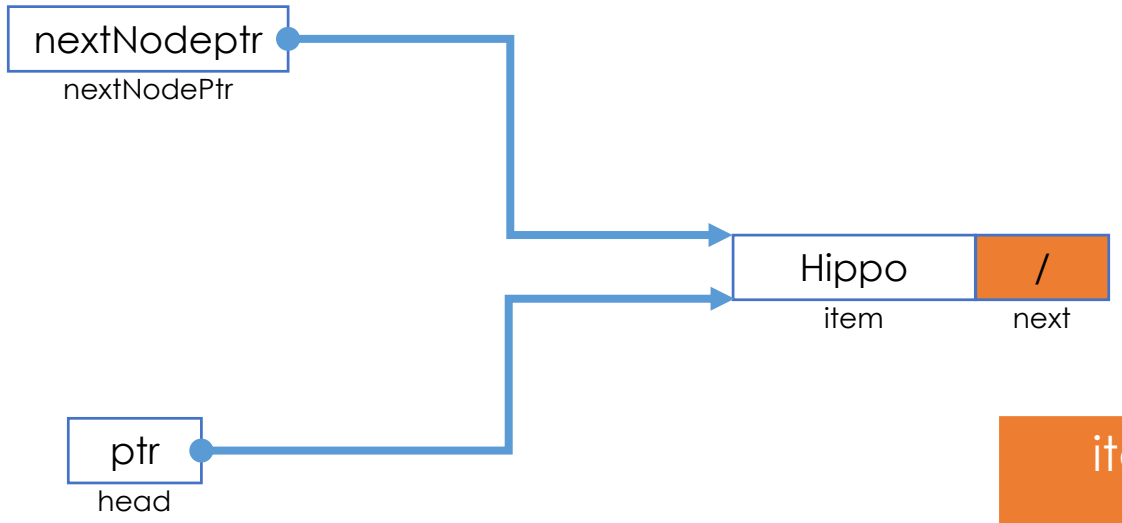
- Place items into bag
  - Create a node and store referenced item
  - Copy reference in head (heatPtr) to next field in node
  - Store reference to new node in head




```
/** #file LinkedBag.cpp (segment of) */

template<class ItemType>
Bool LinkedBag<ItemType>::add(const ItemType& newEntry)
{
      // Add to beginning of chain: new node references rest of chain
      // (headPtr is nullptr if chain is empty)
      Node<ItemType>* newNodePtr = new Node<ItemType>();
      newNodePtr->setItem(newEntry);
      newNodePtr->setNext(headPtr); // New node points to chain
      headPtr = newNodePtr;         // New node is now first node
      itemCount++;
      return true;                  // The method is always successful
}     // end add
```

Implementing Core Methods

- Place items into bag
  - Create a node and store referenced item
  - Copy reference in head (heatPtr) to next field in node
  - Store reference to new node in head



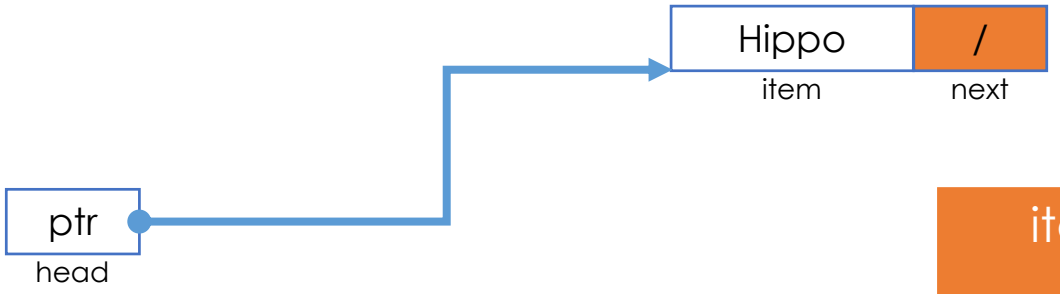

```
/** #file LinkedBag.cpp (segment of) */

template<class ItemType>
Bool LinkedBag<ItemType>::add(const ItemType& newEntry)
{
      // Add to beginning of chain: new node references rest of chain
      // (headPtr is nullptr if chain is empty)
      Node<ItemType>* newNodePtr = new Node<ItemType>();
      newNodePtr->setItem(newEntry);
      newNodePtr->setNext(headPtr); // New node points to chain
      headPtr = newNodePtr;         // New node is now first node
      itemCount++;
      return true;                  // The method is always successful
}     // end add
```

itemCount
0

Implementing Core Methods

- Place items into bag
  - Create a node and store referenced item
  - Copy reference in head (heatPtr) to next field in node
  - Store reference to new node in head

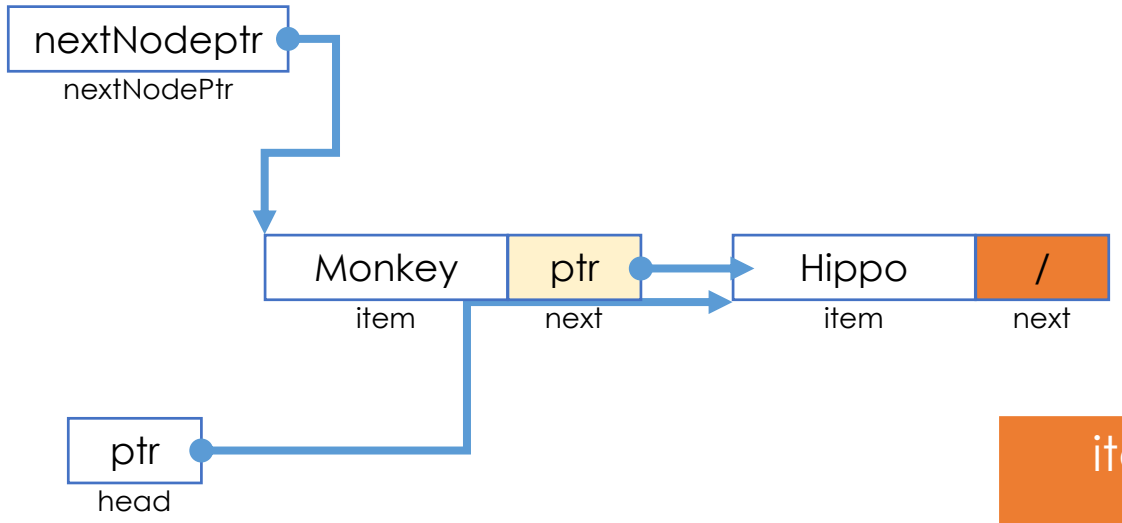
```
/** #file LinkedBag.cpp (segment of) */

template<class ItemType>
Bool LinkedBag<ItemType>::add(const ItemType& newEntry)
{
      // Add to beginning of chain: new node references rest of chain
      // (headPtr is nullptr if chain is empty)
      Node<ItemType>* newNodePtr = new Node<ItemType>();
      newNodePtr->setItem(newEntry);
      newNodePtr->setNext(headPtr); // New node points to chain
      headPtr = newNodePtr;         // New node is now first node
      itemCount++;
      return true;                  // The method is always successful
}     // end add
```



itemCount
1

Implementing Core Methods

- Place items into bag
  - Create a node and store referenced item
  - Copy reference in head (heatPtr) to next field in node
  - Store reference to new node in head

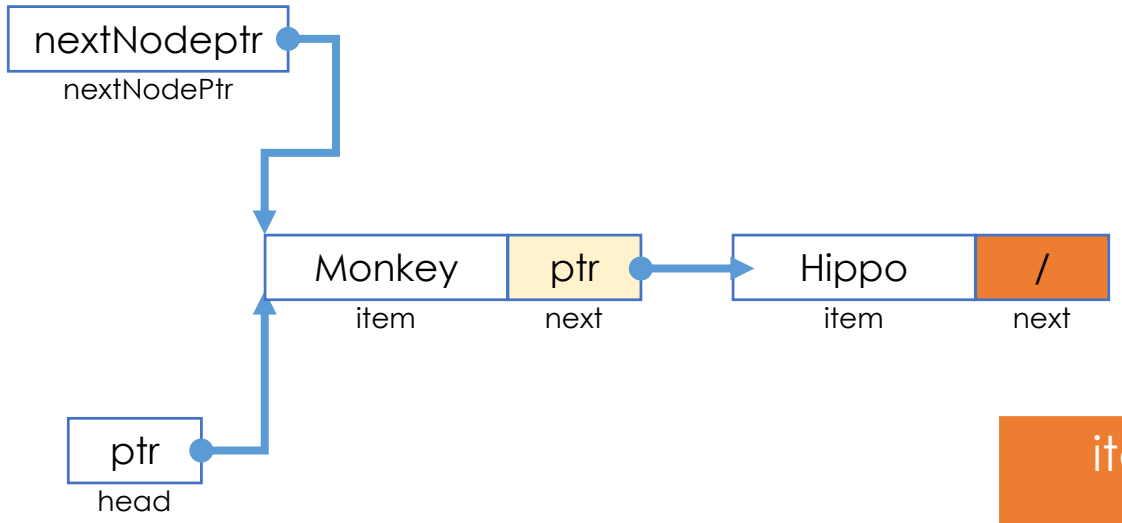



```
/** #file LinkedBag.cpp (segment of) */

template<class ItemType>
Bool LinkedBag<ItemType>::add(const ItemType& newEntry)
{
      // Add to beginning of chain: new node references rest of chain
      // (headPtr is nullptr if chain is empty)
      Node<ItemType>* newNodePtr = new Node<ItemType>();
      newNodePtr->setItem(newEntry);
      newNodePtr->setNext(headPtr); // New node points to chain
      headPtr = newNodePtr;         // New node is now first node
      itemCount++;
      return true;                  // The method is always successful
}     // end add
```

itemCount
1

Implementing Core Methods

- Place items into bag
  - Create a node and store referenced item
  - Copy reference in head (heatPtr) to next field in node
  - Store reference to new node in head



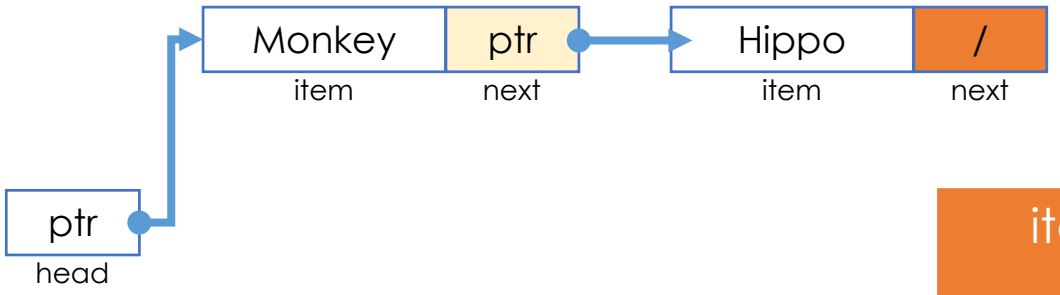

```
/** #file LinkedBag.cpp (segment of) */

template<class ItemType>
Bool LinkedBag<ItemType>::add(const ItemType& newEntry)
{
      // Add to beginning of chain: new node references rest of chain
      // (headPtr is nullptr if chain is empty)
      Node<ItemType>* newNodePtr = new Node<ItemType>();
      newNodePtr->setItem(newEntry);
      newNodePtr->setNext(headPtr); // New node points to chain
      headPtr = newNodePtr;         // New node is now first node
      itemCount++;
      return true;                  // The method is always successful
}     // end add
```

itemCount
1

Implementing Core Methods

- Place items into bag
  - Create a node and store referenced item
  - Copy reference in head (heatPtr) to next field in node
  - Store reference to new node in head

```
/** #file LinkedBag.cpp (segment of) */

template<class ItemType>
Bool LinkedBag<ItemType>::add(const ItemType& newEntry)
{
      // Add to beginning of chain: new node references rest of chain
      // (headPtr is nullptr if chain is empty)
      Node<ItemType>* newNodePtr = new Node<ItemType>();
      newNodePtr->setItem(newEntry);
      newNodePtr->setNext(headPtr); // New node points to chain
      headPtr = newNodePtr;         // New node is now first node
      itemCount++;
      return true;                  // The method is always successful
}     // end add
```



itemCount
2

Implementing Core Methods

- Report on items in object
  - Allows us to determine if the items were added properly
- Pseudocode

Let a current pointer reference the first node in the chain

while (the current pointer is not the null pointer)

{

    Assign the data portion of the current node to the next element in a vector

    Set the current pointer to the next pointer of the current node

}

Implementing Core Methods

- Report on items in object
  - Allows us to determine if the items were added properly
- Pseudocode

Let a current pointer reference the first node in the chain

while (the current pointer is not the null pointer)

{

Assign the data portion of the current node to the next element in a vector

Set the current pointer to the next pointer of the current node

}

```
/** #file LinkedBag.cpp (segment of) */


template<class ItemType>

std::vector<ItemType> LinkedBag<ItemType>::toVector() const

{

      std::vector<ItemType> bagContents;

      Node<ItemType>* curPtr = headPtr;

      int counter = 0;

      while ((curPtr != nullptr) && (counter < itemCount))

      {

            bagContents.push_back(curPtr->getItem());

            curPtr = curPtr->getNext();

            counter++;

      } // end while

      return bagContents;

}     // end toVector
```

Implementing Core Methods

- Report on items in object
  - Allows us to determine if the items were added properly
- Pseudocode

Let a current pointer reference the first node in the chain

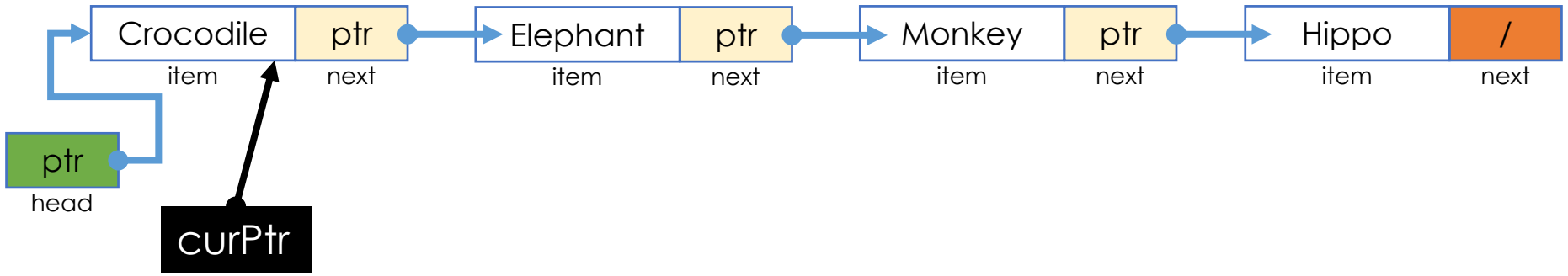
while (the current pointer is not the null pointer)

{

Assign the data portion of the current node to the next element in a vector

Set the current pointer to the next pointer of the current node

}



Counter
1

Implementing Core Methods

- Report on items in object
  - Allows us to determine if the items were added properly
- Pseudocode

Let a current pointer reference the first node in the chain

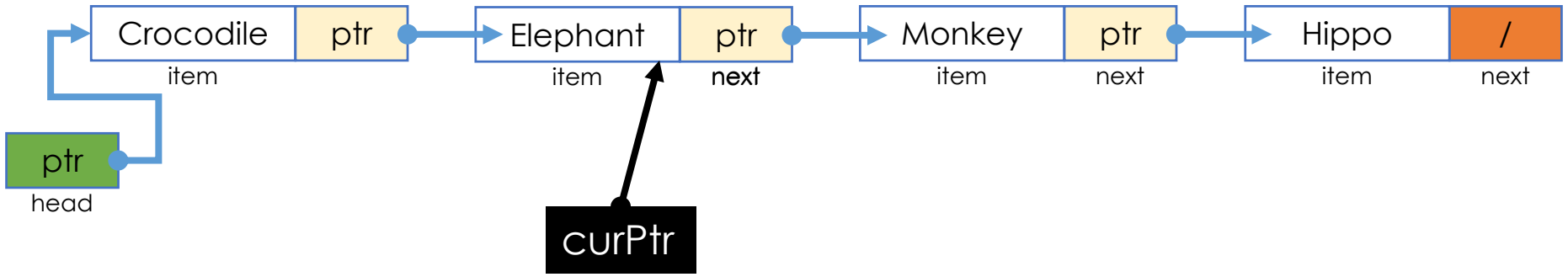
while (the current pointer is not the null pointer)

{

Assign the data portion of the current node to the next element in a vector

Set the current pointer to the next pointer of the current node

}



Counter 2

Implementing Core Methods

- Report on items in object
  - Allows us to determine if the items were added properly
- Pseudocode

Let a current pointer reference the first node in the chain

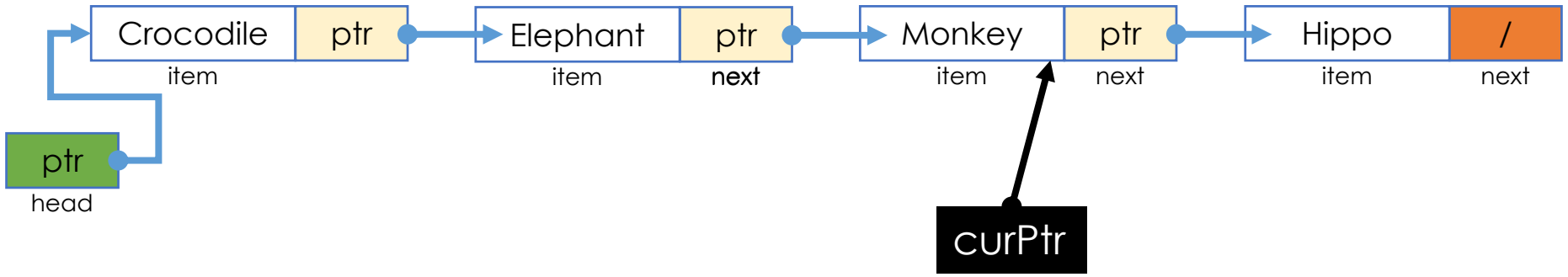
while (the current pointer is not the null pointer)

{

Assign the data portion of the current node to the next element in a vector

Set the current pointer to the next pointer of the current node

}



Counter 3

Implementing Core Methods

- Report on items in object
  - Allows us to determine if the items were added properly
- Pseudocode

Let a current pointer reference the first node in the chain

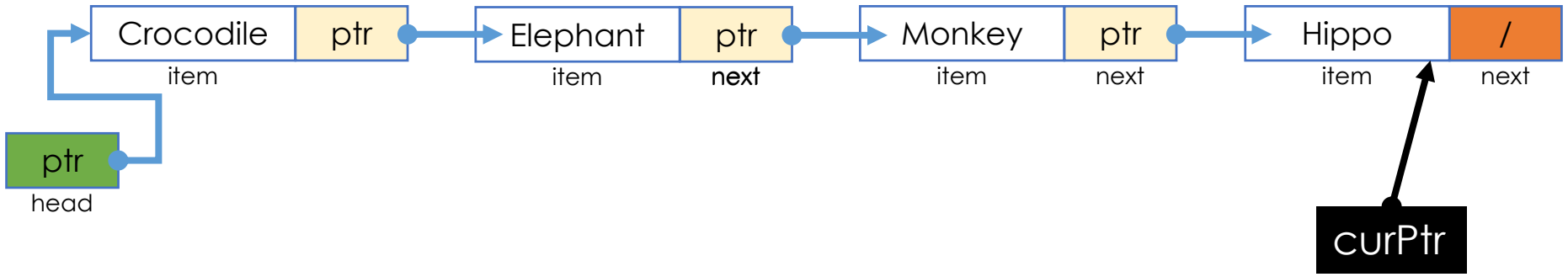
while (the current pointer is not the null pointer)

{

Assign the data portion of the current node to the next element in a vector

Set the current pointer to the next pointer of the current node

}



Counter
4

Remove a Specific Item

- Pseudocode

```
remove(anEntry)
{
        Find the node that contains anEntry
        Replace anEntry with the entry in the first node
        Delete the first node
}
```

Remove a Specific Item

- Pseudocode

```
remove(anEntry)
{
        Find the node that contains anEntry
        Replace anEntry with the entry in the first node
        Delete the first node
}
```

```
/** #file LinkedBag.cpp (segment of) */
template<class ItemType>
bool LinkedBag<ItemType>::remove(const ItemType& anEntry)
{
      Node<ItemType>* entryNodePtr = getPointerTo(anEntry);
      bool canRemoveItem = !isEmpty() && (entryNodePtr != nullptr);
      if (canRemoveItem)
      {
            // Copy data from first node to located node
            entryNodePtr->setItem(headPtr->getItem());
            // Disconnect first node
            Node<ItemType>* nodeToDeletePtr = headPtr;
            headPtr = headPtr->getNext();
            // Return node to the system
            nodeToDeletePtr->setNext(nullptr);
            delete nodeToDeletePtr;
            nodeToDeletePtr = nullptr;
            itemCount--;
      } // end if
      return canRemoveItem
} // end remove
```

Remove a Specific Item

- After the method remove deletes a node, the system can use this returned memory and possibly even reallocate it to your program as a result of the **new** operator.
- Programming Tip: Every time you allocate memory by using **new**, you must eventually deallocate it by using **delete**.

```
/** #file LinkedBag.cpp (segment of) */
template<class ItemType>
bool LinkedBag<ItemType>::remove(const ItemType& anEntry)
{
      Node<ItemType>* entryNodePtr = getPointerTo(anEntry);
      bool canRemoveItem = !isEmpty() && (entryNodePtr != nullptr);
      if (canRemoveItem)
      {
            // Copy data from first node to located node
            entryNodePtr->setItem(headPtr->getItem());
            // Disconnect first node
            Node<ItemType>* nodeToDeletePtr = headPtr;
            headPtr = headPtr->getNext();
            // Return node to the system
            nodeToDeletePtr->setNext(nullptr);
            delete nodeToDeletePtr;
            nodeToDeletePtr = nullptr;
            itemCount--;
      } // end if
      return canRemoveItem
} // end remove
```

Remove a Specific Item

- After the method remove deletes a node, the system can use this returned memory and possibly even reallocate it to your program as a result of the **new** operator.
- Programming Tip: Every time you allocate memory by using **new**, you must eventually deallocate it by using **delete**.
  - Note: For a pointer p, delete p deallocates the node to which p points; it does not deallocate p.
  - The pointer p still exists but it contains an undefined value.
  - You should not reference p or any other pointer variable that still points to the deallocated node.
  - To help avoid this kind of error, assign nullptr to p after executing delete p.

```
/** #file LinkedBag.cpp (segment of) */
template<class ItemType>
bool LinkedBag<ItemType>::remove(const ItemType& anEntry)
{
      Node<ItemType>* entryNodePtr = getPointerTo(anEntry);
      bool canRemoveItem = !isEmpty() && (entryNodePtr != nullptr);
      if (canRemoveItem)
      {
            // Copy data from first node to located node
            entryNodePtr->setItem(headPtr->getItem());
            // Disconnect first node
            Node<ItemType>* nodeToDeletePtr = headPtr;
            headPtr = headPtr->getNext();
            // Return node to the system
            nodeToDeletePtr->setNext(nullptr);
            delete nodeToDeletePtr;
            nodeToDeletePtr = nullptr;
            itemCount--;
      } // end if
      return canRemoveItem
} // end remove
```

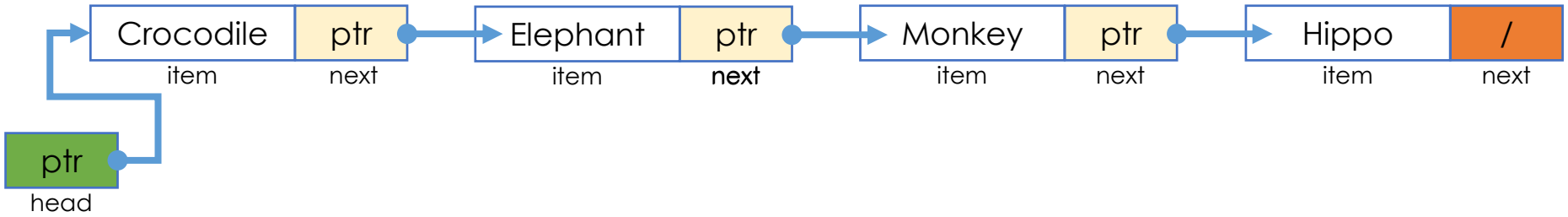
Remove a Specific Item

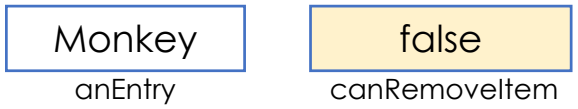
- After the method remove deletes a node, the system can use this returned memory and possibly even reallocate it to your program as a result of the **new** operator.
- Programming Tip: Every time you allocate memory by using **new**, you must eventually deallocate it by using **delete**.
  - Note: For a pointer p, delete p deallocates the node to which p points; it does not deallocate p.
  - The pointer p still exists but it contains an undefined value.
  - You should not reference p or any other pointer variable that still points to the deallocated node.
  - To help avoid this kind of error, assign nullptr to p after executing delete p.

```
/** #file LinkedBag.cpp (segment of) */

template<class ItemType>

Node<ItemType>* LinkedBag<ItemType>::getPointerTo(const ItemType&
      anEntry) const

{

      bool found = false;

      Node<ItemType>* curPtr = headPtr;

      while(!found && (curPtr != nullptr))

      {

            if (anEntry == curPtr->getItem())

                  found = true;

            else

                  curPtr = curPtr->getNext();

      } // end while

      return curPtr;

} // end remove
```

Remove a Specific Item

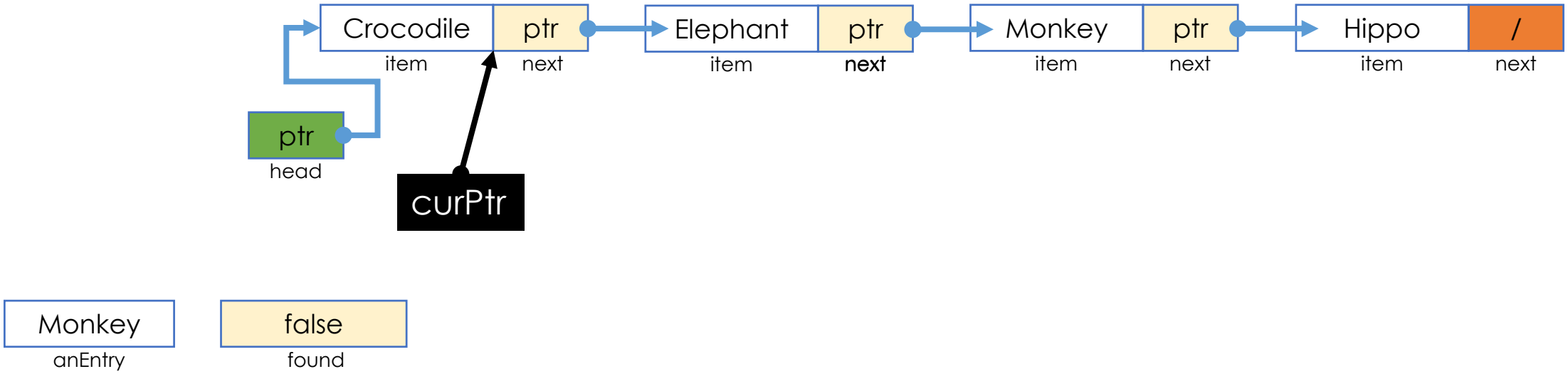
- An example: running `getPointerTo`





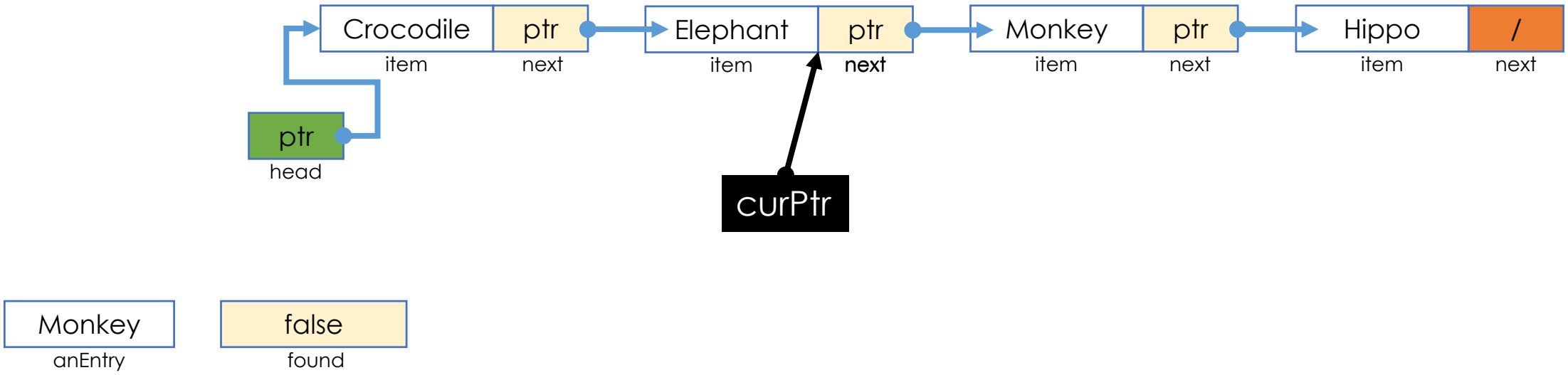
Remove a Specific Item

- An example: running `getPointerTo`



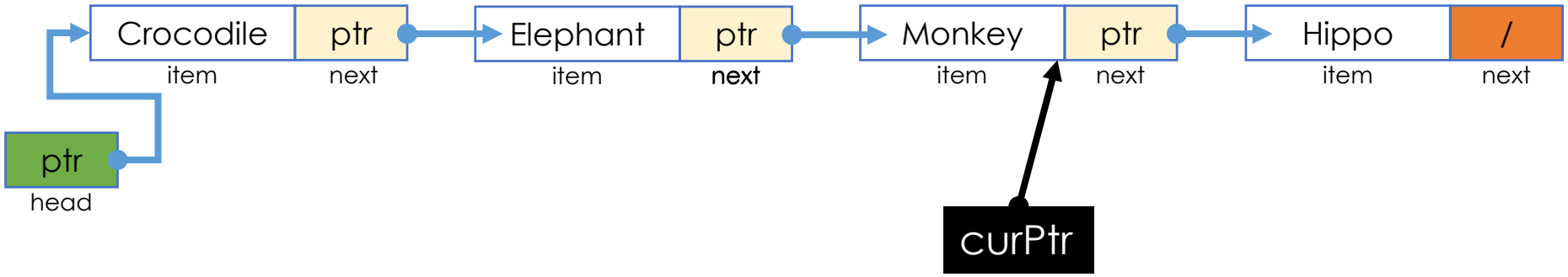
Remove a Specific Item

- An example: running `getPointerTo`



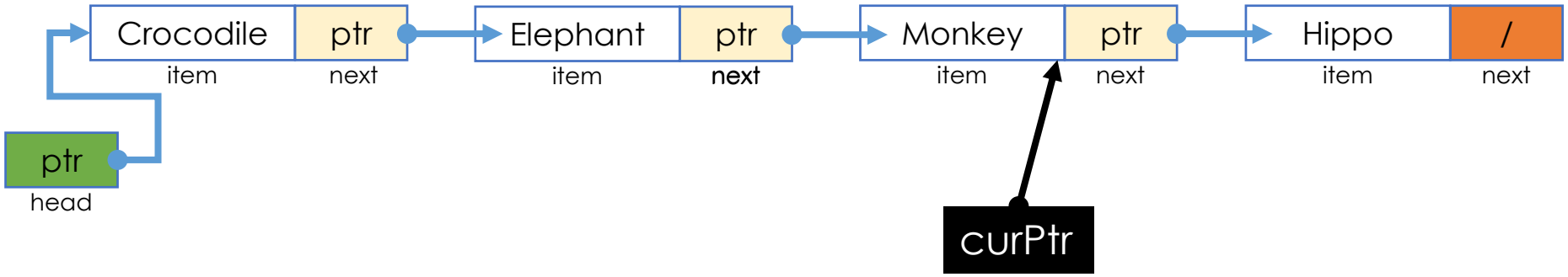
Remove a Specific Item

- An example: running `getPointerTo`



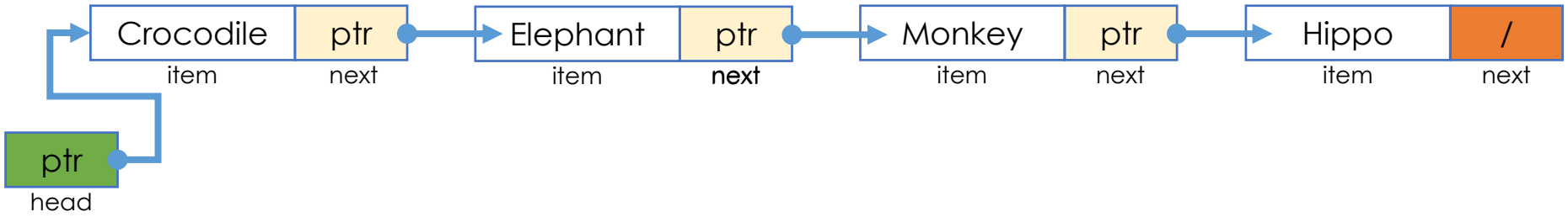
Remove a Specific Item

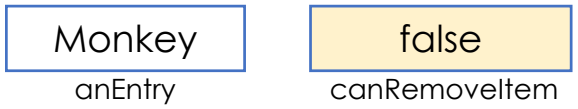
- An example: running `getPointerTo`



Remove a Specific Item

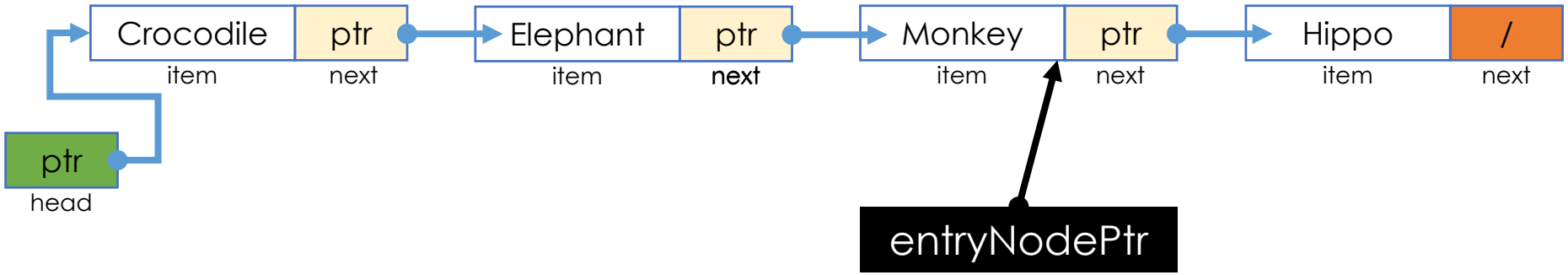
- An example: running `remove`

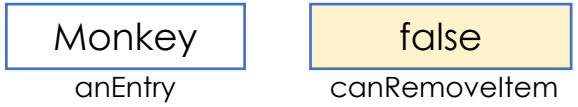




Remove a Specific Item

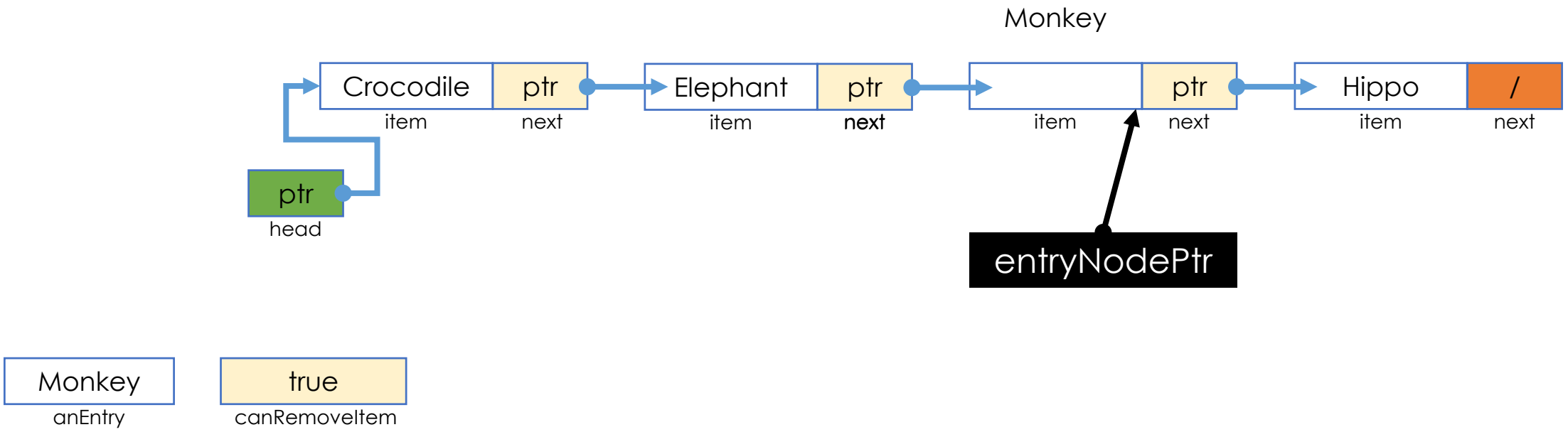
- An example: running `remove`





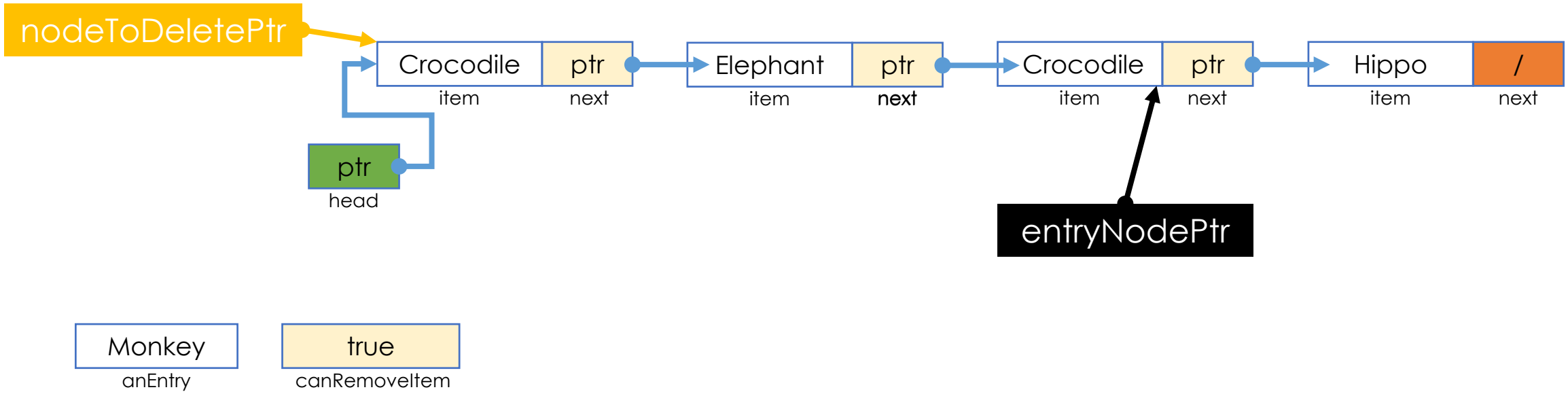
Remove a Specific Item

- An example: running `remove`



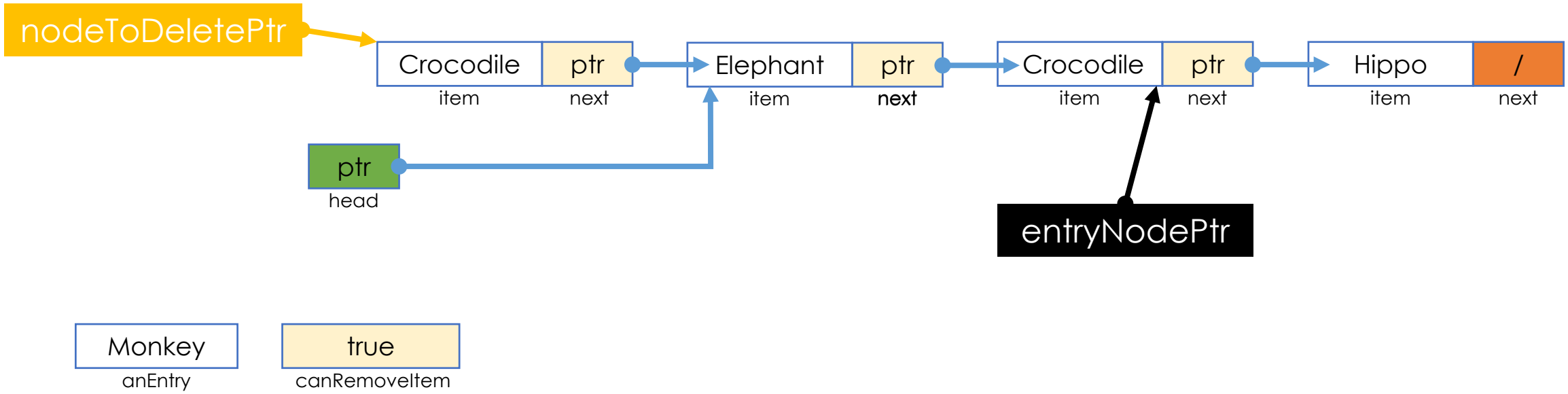
Remove a Specific Item

- An example: running `remove`



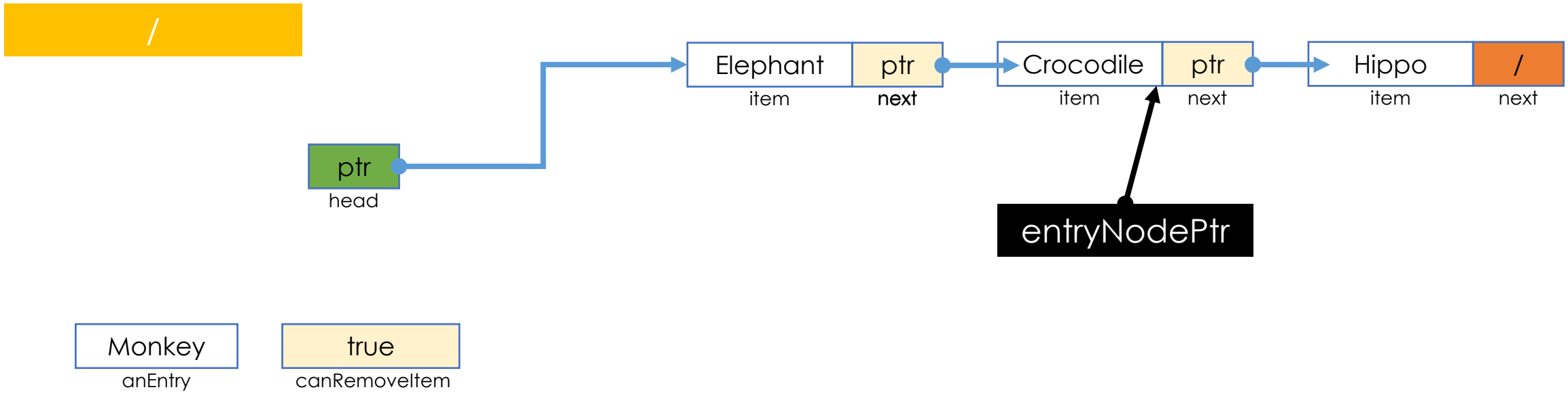
Remove a Specific Item

- An example: running `remove`



Remove a Specific Item

- An example: running `remove`



The Method `contains`

- Use the private helper method getPointerTo
  - If the helper returns nullptr
    - The item is not in the bag
  - If the helper returns a reference to a node
    - The item is in the bag

The Method `contains`

- Use the private helper method getPointerTo
  - If the helper returns nullptr
    - The item is not in the bag
  - If the helper returns a reference to a node
    - The item is in the bag

```
/** #file LinkedBag.cpp (segment of) */

template<class ItemType>

bool LinkedBag<ItemType>::contains ItemType& anEntry) const

{

     return (getPointerTo(anEntry) != nullptr)

} // end contains
```

The Method `clear`

- The method clear cannot simply set `ItemCount` to zero, thereby ignoring all the entries in the linked chain.
- Because the nodes in the chain were allocated dynamically, `clear` must deallocate them.

The Method `clear`

- Use the private helper method getPointerTo
  - If the helper returns nullptr
    - The item is not in the bag
  - If the helper returns a reference to a node
    - The item is in the bag

```
/** #file LinkedBag.cpp (segment of) */
template<class ItemType>
void LinkedBag<ItemType>::clear()
{
      Node<ItemType>* nodeToDeletePtr = headPtr;
      while (headPtr != nullptr)
      {
            headPtr = headPtr->getNext();
            // Return node to the system
            nodeToDeletePtr->setNext(nullptr);
            delete nodeToDeletePtr;
            nodeToDeletePtr = headPtr;
      } // end while
      // headPtr is nullptr; nodeToDeletePtr is nullptr
      itemCount = 0;
} // end clear
```

Thank you