Lists, Queues and Summary on Linear ADTs

The List

A “List” is an ADT where the order of the items can be decided (in contrast to the “Bag” there is a specific order).

Axiomatic Definition:

```
- (new List()).isEmpty() = true
- (new List()).getLenght()=0
- aList.getLength()=aList.insert(i.x).getLength()-1
- aList.getLength()=aList.remove(i).getLength()+1
- aList.insert(i.x).isEmpty()=false
- (new List()).remove(i) = false
- aList.insert(i,x).remove(i)=aList
- (new List()).getEntry(i)=error
- aList.insert(i.x).getEntry(i)=x
- aList.getEntry(i)=(aList.insert(i,x)).getEntry(i+1)
- aList.getEntry(i+1)=(aList.remove(i)).getEntry(i)
- (new List()).setEnrty(i,x)=error
- (aList.setEntry(i,x)).getEntry(i)=x
```

The List: Abstract Class

```
template<class T>
class ListInterface{

    virtual bool isEmpty() = 0;

    virtual int getLength() const = 0;

    virtual bool insert(int newPosition, const T& newEntry) = 0;

    virtual bool remove(int position) = 0;

    virtual void clear() = 0;

    virtual T getEntry(int position) const = 0;

    virtual void setEntry(int position, const T& newEntry) = 0;
};
```

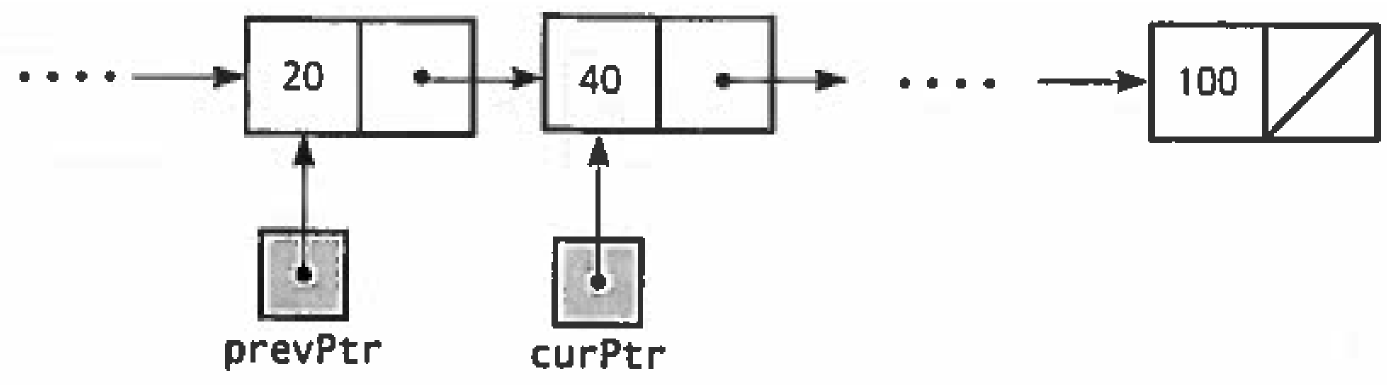
Exercise: infer what exactly the abstract methods do, given the previous List definition and the parameters/return values.

List Implementations

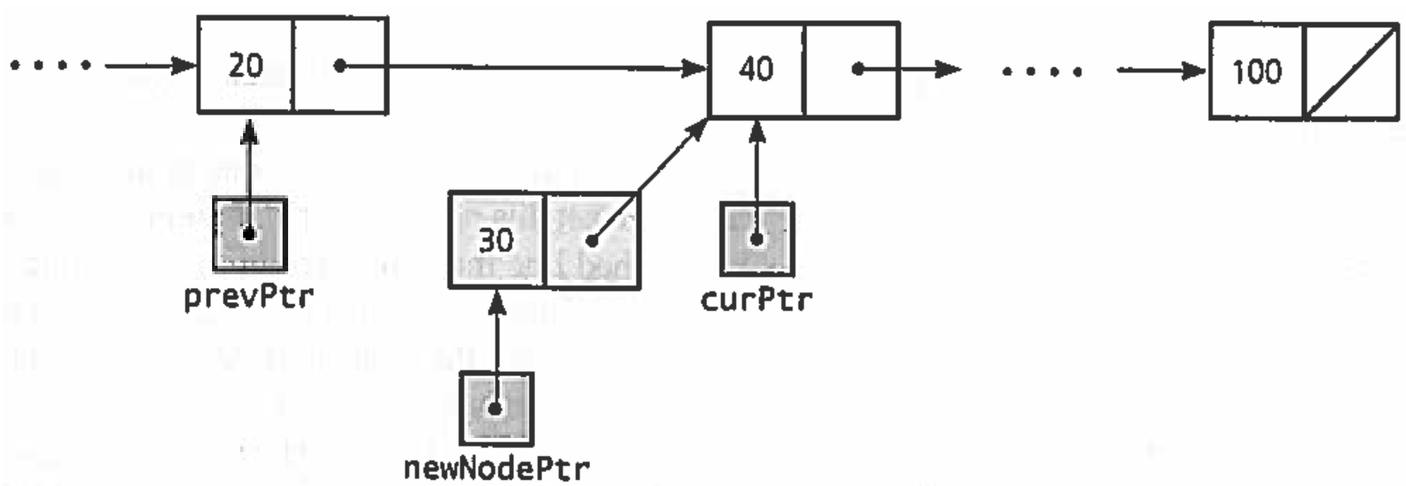
An implementation based on arrays is straightforward.
A more interesting case is the use of a linked list (what are the advantages?).
We are going to discuss the most interesting methods:

```
- bool insert(int position, const T& newEntry)

- bool remove(int position)
```

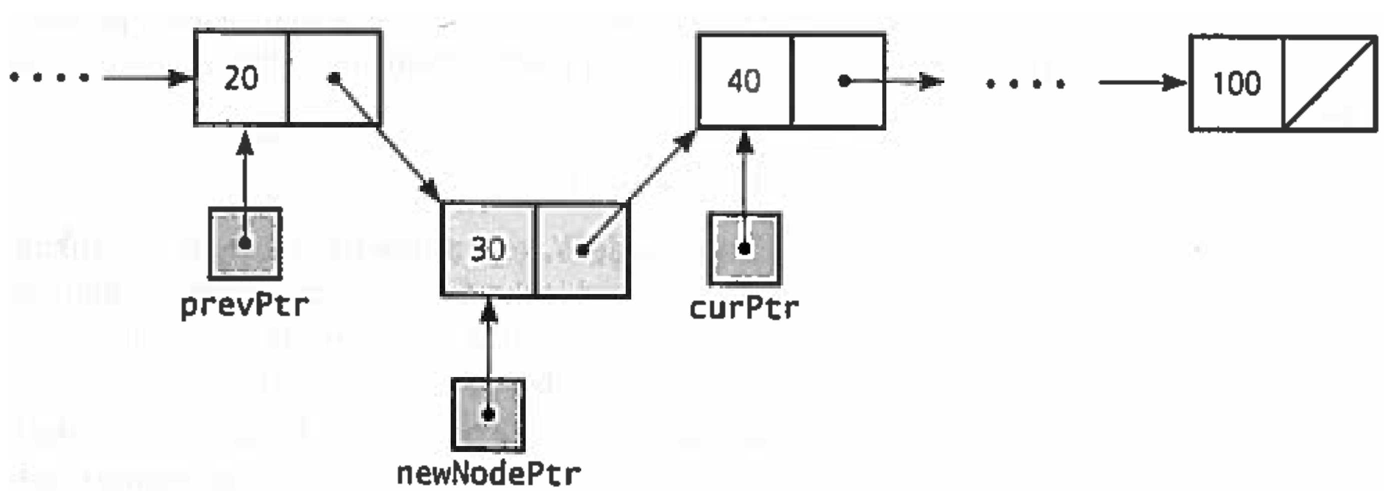
List Insertion (middle insertion case)



Initial List



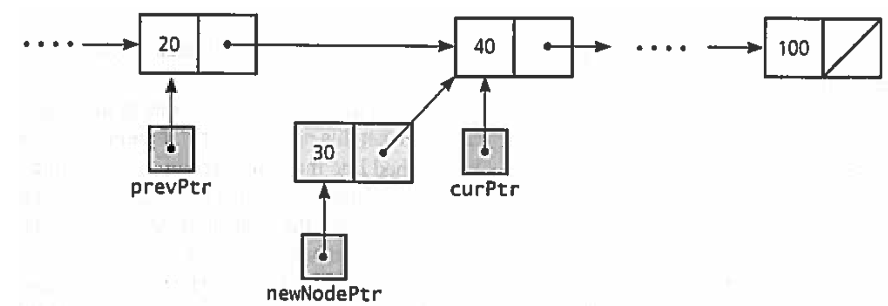
Create a new node and assign its pointer

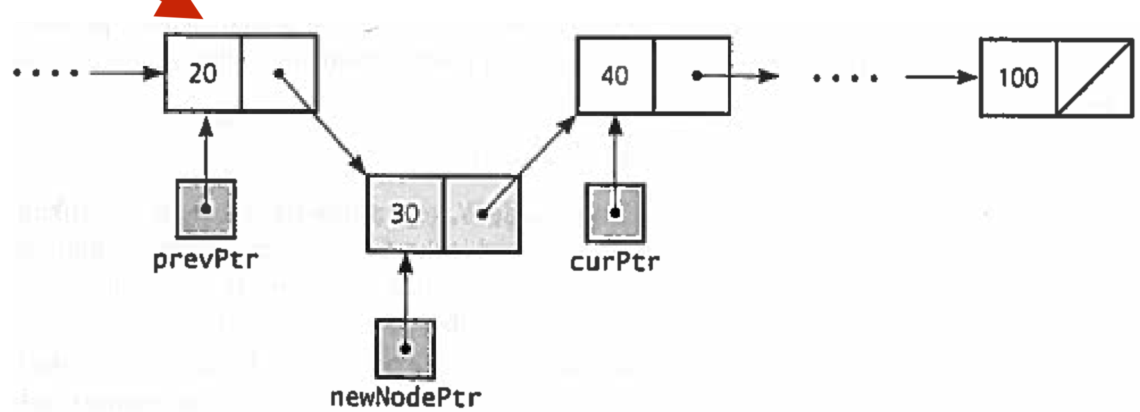


Reassign the previous node.

List Insertion

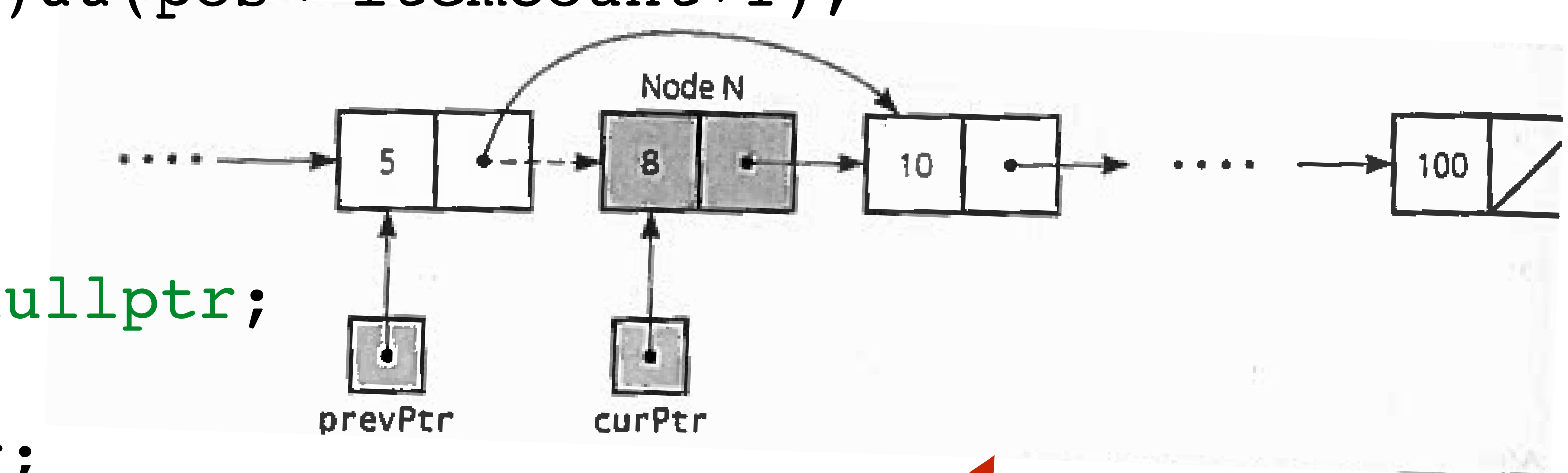
```
template<class T>
bool LinkedList<T>::insert(int pos, T& newEntry){

bool oktoinsert = (pos>=1)&&(pos<=itemCount+1);

if (oktoinsert){

        Node<T>* newNodePtr = new Node<T>(newEntry);
        if (pos==1){
            newNodePtr->setNext(headPtr);
            headPtr = newNodePtr;
        } else {
            Node<T>* prevPtr = getNodeAt(pos-1);
            newNodePtr->setNext(prevPtr->getNext());
            prevPtr->setNext(newNodePtr);
        }
        itemCount++;
    }
    return oktoinsert;
}
```





List Removal

```
template<class T>
bool LinkedList<T>::remove(int pos){

bool oktoremove = (pos>=1)&&(pos<=itemCount+1);

if (oktoremove){

        Node<T>* curPtr = nullptr;
        if (pos==1){
            curPtr = headPtr;
            headPtr = headPtr->getNext();
        } else {
            Node<T>* prevPtr = getNodeAt(pos-1);
            curPtr = prevPtr->getNext();
            prevPtr->setNext(curPtr->getNext());
        }
        curPtr->setNext(nullptr);delete curPtr;curPtr=nullptr;
        itemCount—;
    }
    return oktoremove;
}
```



The Sorted List

- An important variant of the List
- Random insertion is not possible.
- Elements are always inserted in a sorted position.
- Can be constructed “inheriting” from the List ADT.

```
template<class T>
bool LinkedSList<T>::insertSorted(const T& newEntry){
   Node<T>* newNodePtr = new Node<T>(newEntry);
   Node<T>* prevPtr = getNodeBefore(newEntry);
   if (isEmpty() || (prevPtr==nullptr)){
      newNodePtr->setNext(headPtr);
      headPtr = newNodePtr;
   } else {
      Node<T>* aftPtr = prevPtr->getNext();
      newNodePtr->setNext(aftPtr);
      prevPtr->setNext(newNodePtr);
   }
   itemCount++;
}
```

The Sorted List

```
template<class T>
Node<T>* LinkedSList<T>::getNodeBefore(const T& anEntry) const {

   Node<T>* curPtr = headPtr;
   Node<T>* prevPtr = nullptr;

   while ((curPtr!=nullptr)&&(anEntry > curPtr->getItem())){

      prevPtr = curPtr;
      curPtr = curPtr->getNext();

   }
   return prevPtr;
}
```

This is the method which enforces the sorted insertion.

The Queue

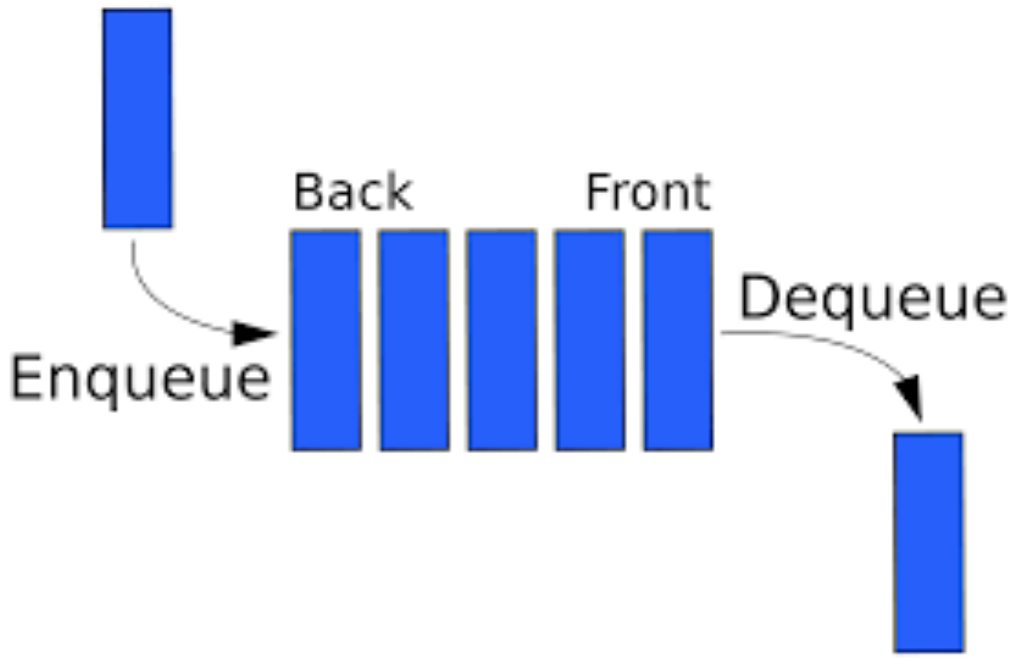
The queue is a “**FIFO**” (first-in first-out) data structure.
Another description (closer to many real-world situations) could be:
“first-come first-serve”.

Examples:

- A line of people in front of a clerk.
- The way a printer prints the submitted jobs.
- The way a CPU executes jobs.
- The construction flow in a factory
- … …

The Queue: Abstract Class

```
template<class T>
class QueueInterface
{
   virtual bool isEmpty() = 0;

   virtual bool enqueue(const T& newEntry) = 0;

   virtual bool dequeue() = 0;

   virtual T peekFront() const = 0;
};
```



The Queue: Linked List Implementation

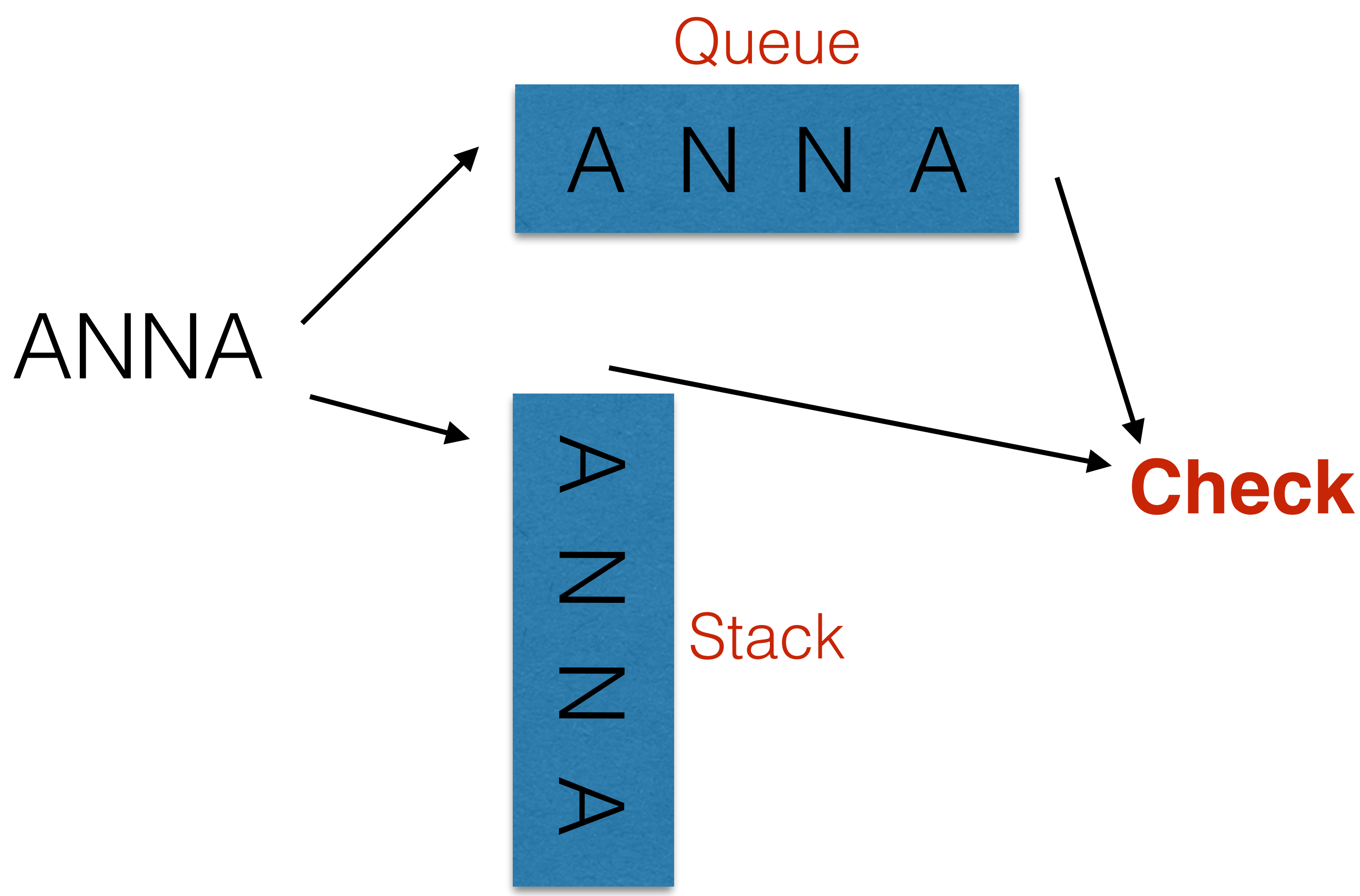
```
template<class T> bool LinkedQueue<T>::enqueue(const T& newEntry)
{
   Node<T>* newNodePtr = new Node<T>(newEntry);
   if (isEmpty()) frontPtr = newNodePtr;
   else backPtr = newNodePtr;
   return true;
}

template<class T> bool LinkedQueue<T>::dequeue(const T& newEntry)
{
   bool res = false;
   if (!isEmpty()){
      Node<T>* nodeToDeletePtr = frontPtr;
      if (frontPtr==backPtr) {frontPtr=nullptr;backPtr=nullptr;}
      else frontPtr = frontPtr->getNext();

      nodeToDeletePtr->setNext(nullptr);
      delete nodeToDeletePtr;
      nodeToDeletePtr = nullptr;
      res = true;
   }
   return res;
}
```

A Classic Problem: Recognize Palindromes

A palindrome is a string which reads the same from left to right or *vice versa*.

Examples:

radar
anna
stack cats
borrow or rob
my gym
….



A conceptually simple way to recognize palindromes is to **combine a stack with a queue**!

The Priority Queue

A **priority queue** is similar to a queue with some properties of the Sorted List.

Priority queues are quite common in practical situations:
- Task scheduling by a CPU.
- ER procedures
- Time management systems
- … …

We can inherit the properties of a SortedList for building a priority queue, where the insertion of a new task is done according to the assigned priority.
When you “peek” the list, the highest-priority object is returned.

More on priority queues later on, when we will discuss “heaps”.

“Linear” ADTs so far

- Bag
- Stack
- List
- Sorted List
- Queue
- Priority Queue

Computational Complexity Considerations (Linear ADTs)

ArrayList:

- Add: O(1)
- Remove: O(n)
- Contains: O(n)
- Size: O(1)

Linked List:

- Inserting: O(n)
- Deleting: O(n)
- Searching: O(n)

Doubly-Linked List:

- Inserting: O(n)
- Deleting: O(n)
- Searching: O(n)

Stack:

- Push: O(1)
- Pop: O(1)
- Top (peek): O(1)

Queue:

- enqueue: O(1)
- dequeue: O(1)
- Size: O(1)

Sorted List:

In some cases we can get O(log n): e.g. searching with an array implementation.

Linear ADTs and Recursion

- Although not strictly required, some tasks on LADTs can be recast in recursive form.

- It is an useful exercise for learning how to develop recursive algorithms.

- It will be very useful with more complicated data structures.

Recursive Search: Basic Idea

```
Node RecursiveSearch (Node N, int value)
  {
    if (N == null)
      return null; //the base case is an empty ADT
    else
      if (N.value == value)
        return N;
      else
        return RecursiveSearch(N.next, value);
  }
```

Question: which kind of recursion is this?

Another example : sum all the items in the ADT

```
int sumAll (Node N)
  {
    if (N == null)
      return 0;
    else
      return N.value + sumAll(N.next);
  }
```

A concrete example for the "Bag" data type

We defined a method for getting the pointer to a specific item in the bag: getPointerTo. The implementation was iterative. Here a recursive variant of it:

```
template <class T>
Node<T>* getPointerTo(const T& target, Node<T>* curPtr) const {

    Node<T>* result = nullptr;
    if (curPtr != nullptr){

        if (target == curPtr->getItem()) result = curPtr;
        else result = getPointerTo(target,curPtr->getNext());
    }
    return result;
}
```