Member Variables of the `class linkedListType`

To maintain a linked list, we use two pointers—`first` and `last`. The pointer `first` points to the first node in the list, and `last` points to the last node in the list. We also keep a count of the number of nodes in the list. Therefore, the `class linkedListType` has three instance variables, as follows:

```
protected:
   int count; //variable to store the number of elements in the list
   nodeType<Type> *first; //pointer to the first node of the list
   nodeType<Type> *last;  //pointer to the last node of the list
```

Linked List Iterators

One of the basic operations performed on a list is to process each node of the list. This requires the list to be traversed starting at the first node. Moreover, a specific application requires each node to be processed in a very specific way. A common technique to accomplish this is to provide an iterator. So what is an iterator? An **iterator** is an object that produces each element of a container, such as a linked list, one element at a time. The two most common operations on iterators are `++` (the increment operator) and `*` (the dereferencing operator). The increment operator advances the iterator to the next node in the list while the dereferencing operator returns the `info` of the current node.

Note that an iterator is an object. So we need to define a class, which we will call `linkedListIterator`, to create iterators to objects of the `class linkedListType`. The iterator class would have one member variable pointing to (the current) node.

```
//***********************************************************
// Author: D.S. Malik
//
// This class specifies the members to implement an iterator
// to a linked list.
//***********************************************************

template <class Type>
class linkedListIterator
{
public:
    linkedListIterator();
      //Default constructor
      //Postcondition: current = NULL;

    linkedListIterator(nodeType<Type> *ptr);
      //Constructor with a parameter.
      //Postcondition: current = ptr;

    Type operator*();
      //Function to overload the dereferencing operator *.
      //Postcondition: Returns the info contained in the node.

    linkedListIterator<Type> operator++();
      //Overload the preincrement operator.
      //Postcondition: The iterator is advanced to the next node.
```

```
    bool operator==(const linkedListIterator<Type>& right) const;
      //Overload the equality operator.
      //Postcondition: Returns true if this iterator is equal to
      //    the iterator specified by right, otherwise it returns
      //    false.

    bool operator!=(const linkedListIterator<Type>& right) const;
      //Overload the not equal to operator.
      //Postcondition: Returns true if this iterator is not equal to
      //    the iterator specified by right, otherwise it returns
      //    false.

private:
    nodeType<Type> *current; //pointer to point to the current
                             //node in the linked list
};
```


Figure 5-18 shows the UML class diagram of the `class linkedListIterator`.

linkedListIterator<Type>
- *current: nodeType<Type>
+linkedListIterator() +linkedListIterator(nodeType<Type>) +operator*(): Type +operator++(): linkedListIterator<Type> +operator==(const linkedListIterator<Type>&) const: bool +operator!=(const linkedListIterator<Type>&) const: bool

FIGURE 5-18 UML class diagram of the `class linkedListIterator`

The definitions of the functions of the `class linkedListIterator` are as follows:

```
template <class Type>
linkedListIterator<Type>::linkedListIterator()
{
    current = NULL;
}

template <class Type>
linkedListIterator<Type>::
                 linkedListIterator(nodeType<Type> *ptr)
{
    current = ptr;
}

template <class Type>
Type linkedListIterator<Type>::operator*()
{
    return current->info;
}
```

```
template <class Type>
linkedListIterator<Type> linkedListIterator<Type>::operator++()
{
    current = current->link;

    return *this;
}

template <class Type>
bool linkedListIterator<Type>::operator==
               (const linkedListIterator<Type>& right) const
{
    return (current == right.current);
}

template <class Type>
bool linkedListIterator<Type>::operator!=
                 (const linkedListIterator<Type>& right) const
{
    return (current != right.current);
}
```

From the definitions of the functions and constructors of the `class linkedListIterator`, it follows that each function and the constructors are of $O(1)$.

Now that we have defined the classes to implement the node of a linked list and an iterator to a linked list, next, we describe the `class linkedListType` to implement the basic porperties of a linked list.

The following abstract class defines the basic properties of a linked list as an ADT:

```
//***********************************************************
// Author: D.S. Malik
//
// This class specifies the members to implement the basic
// properties of a linked list. This is an abstract class.
// We cannot instantiate an object of this class.
//***********************************************************

template <class Type>
class linkedListType
{
public:
    const linkedListType<Type>& operator=
                         (const linkedListType<Type>&);
      //Overload the assignment operator.

    void initializeList();
      //Initialize the list to an empty state.
      //Postcondition: first = NULL, last = NULL, count = 0;
```

```
bool isEmptyList() const;
  //Function to determine whether the list is empty.
  //Postcondition: Returns true if the list is empty, otherwise
  //    it returns false.

void print() const;
  //Function to output the data contained in each node.
  //Postcondition: none

int length() const;
  //Function to return the number of nodes in the list.
  //Postcondition: The value of count is returned.

void destroyList();
  //Function to delete all the nodes from the list.
  //Postcondition: first = NULL, last = NULL, count = 0;

Type front() const;
  //Function to return the first element of the list.
  //Precondition: The list must exist and must not be empty.
  //Postcondition: If the list is empty, the program terminates;
  //    otherwise, the first element of the list is returned.

Type back() const;
  //Function to return the last element of the list.
  //Precondition: The list must exist and must not be empty.
  //Postcondition: If the list is empty, the program
  //               terminates; otherwise, the last
  //               element of the list is returned.

virtual bool search(const Type& searchItem) const = 0;
  //Function to determine whether searchItem is in the list.
  //Postcondition: Returns true if searchItem is in the list,
  //    otherwise the value false is returned.

virtual void insertFirst(const Type& newItem) = 0;
  //Function to insert newItem at the beginning of the list.
  //Postcondition: first points to the new list, newItem is
  //    inserted at the beginning of the list, last points to
  //    the last node in the list, and count is incremented by
  //    1.

virtual void insertLast(const Type& newItem) = 0;
  //Function to insert newItem at the end of the list.
  //Postcondition: first points to the new list, newItem is
  //    inserted at the end of the list, last points to the
  //    last node in the list, and count is incremented by 1.

virtual void deleteNode(const Type& deleteItem) = 0;
  //Function to delete deleteItem from the list.
  //Postcondition: If found, the node containing deleteItem is
  //    deleted from the list. first points to the first node,
  //    last points to the last node of the updated list, and
  //    count is decremented by 1.
```

5

```
    linkedListIterator<Type> begin();
      //Function to return an iterator at the beginning of the
      //linked list.
      //Postcondition: Returns an iterator such that current is set
      //    to first.

    linkedListIterator<Type> end();
      //Function to return an iterator one element past the
      //last element of the linked list.
      //Postcondition: Returns an iterator such that current is set
      //    to NULL.

    linkedListType();
      //default constructor
      //Initializes the list to an empty state.
      //Postcondition: first = NULL, last = NULL, count = 0;

    linkedListType(const linkedListType<Type>& otherList);
      //copy constructor

    ~linkedListType();
      //destructor
      //Deletes all the nodes from the list.
      //Postcondition: The list object is destroyed.

protected:
    int count; //variable to store the number of list elements
               //
    nodeType<Type> *first; //pointer to the first node of the list
    nodeType<Type> *last;  //pointer to the last node of the list

private:
    void copyList(const linkedListType<Type>& otherList);
      //Function to make a copy of otherList.
      //Postcondition: A copy of otherList is created and assigned
      //    to this list.
};
```

Figure 5-19 shows the UML class diagram of the `class linkedListType`.

```
                  linkedListType<Type>
-count: int
-*first: nodeType<Type>
-*last: nodeType<Type>
+operator=(const linkedListType<Type>&):
                    const linkedListType<Type>&
+initializeList(): void
+isEmptyList() const: bool
+print() const: void
+length() const: int
+destroyList(): void
+front() const: Type
+back() const: Type
+search(const Type&) const = 0: bool
+insertFirst(const Type&) = 0: void
+insertLast(const Type&) = 0: void
+deleteNode(const Type&) = 0: void
+begin(): linkedListIterator<Type>
+end(): linkedListIterator<Type>
+linkedListType()
+linkedListType(const linkedListType<Type>&)
+~linkedListType()
-copyList(const linkedListType<Type>&): void
```


FIGURE 5-19 UML class diagram of the `class linkedListType`

Note that, typically, in the UML class diagram the names of an abstract class and abstract function are shown in italic.

The instance variables `first` and `last`, as defined earlier, of the `class linkedListType` are `protected`, not `private`, because as noted previously, we will derive the `classes` `unorderedLinkedList` and `orderedLinkedList` from the `class linkedListType`. Because each of the `classes` `unorderedLinkedList` and `orderedLinkedList` will provide separate definitions of the functions `search`, `insertFirst`, `insertLast`, and `deleteNode`, and because these functions would access the instance variable, to provide direct access to the instance variables, the instance variables are declared as `protected`.

The definition of the `class linkedListType` includes a member function to overload the assignment operator. For classes that include pointer data members, the assignment operator must be explicitly overloaded (see Chapters 2 and 3). For the same reason, the definition of the class also includes a copy constructor.

Notice that the definition of the `class linkedListType` contains the member function `copyList`, which is declared as a `private` member. This is because this function is used only to implement the copy constructor and overload the assignment operator.

Next, we write the definitions of the nonabstract functions of the `class LinkedListClass`.

The list is empty if `first` is `NULL`. Therefore, the definition of the function `isEmptyList` to implement this operation is as follows:

```
template <class Type>
bool linkedListType<Type>::isEmptyList() const
{
    return (first == NULL);
}
```

Default Constructor

The default constructor, `linkedListType`, is quite straightforward. It simply initializes the list to an empty state. Recall that when an object of the `linkedListType` type is declared and no value is passed, the default constructor is executed automatically.

```
template <class Type>
linkedListType<Type>::linkedListType() //default constructor
{
    first = NULL;
    last = NULL;
    count = 0;
}
```

From the definitions of the functions `isEmptyList` and the default constructor, it follows that each of these functions is of $O(1)$.

Destroy the List

The function `destroyList` deallocates the memory occupied by each node. We traverse the list starting from the first node and deallocate the memory by calling the operator `delete`. We need a temporary pointer to deallocate the memory. Once the entire list is destroyed, we must set the pointers `first` and `last` to `NULL` and `count` to `0`.

```
template <class Type>
void linkedListType<Type>::destroyList()
{
    nodeType<Type> *temp;   //pointer to deallocate the memory
                            //occupied by the node
    while (first != NULL)   //while there are nodes in the list
    {
        temp = first;        //set temp to the current node
        first = first->link; //advance first to the next node
        delete temp;   //deallocate the memory occupied by temp
    }

    last = NULL; //initialize last to NULL; first has already
                 //been set to NULL by the while loop
    count = 0;
}
```

If the list has n items, the **while** loop executes n times. From this, it follows that the function `destroyList` is of $O(n)$.

Initialize the List

The function `initializeList` initializes the list to an empty state. Note that the default constructor or the copy constructor has already initialized the list when the list object was declared. This operation, in fact, reinitializes the list to an empty state, and so it must delete the nodes (if any) from the list. This task can be accomplished by using the `destroyList` operation, which also resets the pointers `first` and `last` to `NULL` and sets `count` to `0`.

```
template <class Type>
void linkedListType<Type>::initializeList()
{
    destroyList(); //if the list has any nodes, delete them
}
```

5

The function `initializeList` uses the function `destroyList`, which is of $O(n)$. Therefore, the function `initializeList` is of $O(n)$.

Print the List

The member function `print` prints the data contained in each node. To print the data contained in each node, we must traverse the list starting at the first node. Because the pointer `first` always points to the first node in the list, we need another pointer to traverse the list. (If we use `first` to traverse the list, the entire list will be lost.)

```
template <class Type>
void linkedListType<Type>::print() const
{
    nodeType<Type> *current; //pointer to traverse the list

    current = first; //set current point to the first node
    while (current != NULL) //while more data to print
    {
        cout << current->info << " ";
        current = current->link;
    }
}//end print
```

As in the case of the function `destroyList`, the function `print` is of $O(n)$.

Length of a List

The length of a linked list (that is, how many nodes are in the list) is stored in the variable `count`. Therefore, this function returns the value of this variable.

```
template <class Type>
int linkedListType<Type>::length() const
{
    return count;
}
```

Retrieve the Data of the First Node

The function `front` returns the `info` contained in the first node, and its definition is straightforward.

```
template <class Type>
Type linkedListType<Type>::front() const
{
    assert(first != NULL);

    return first->info; //return the info of the first node
}//end front
```

Notice that if the list is empty, the `assert` statement terminates the program. Therefore, before calling this function check, you have to check to see whether the list is nonempty.

Retrieve the Data of the Last Node

The function `back` returns the `info` contained in the last node. Its definition is as follows:

```
template <class Type>
Type linkedListType<Type>::back() const
{
    assert(last != NULL);

    return last->info; //return the info of the last node
}//end back
```

Notice that if the list is empty, the `assert` statement terminates the program. Therefore, before calling this function, you have to check to see whether the list is nonempty.

From the definitions of the functions `length`, `front`, and `back`, it follows easily that each of these functions are of $O(1)$.

Begin and End

The function `begin` returns an iterator to the first node in the linked list and the function `end` returns an iterator to the last node in the linked list. Their definitions are as follows:

```
template <class Type>
linkedListIterator<Type> linkedListType<Type>::begin()
{
    linkedListIterator<Type> temp(first);

    return temp;
}

template <class Type>
linkedListIterator<Type> linkedListType<Type>::end()
{
    linkedListIterator<Type> temp(NULL);

    return temp;
}
```

From the definitions of the functions length, front, back, begin, and end, it follows easily that each of these functions are of $O(1)$.

Copy the List

The function copyList makes an identical copy of a linked list. Therefore, we traverse the list to be copied starting at the first node. Corresponding to each node in the original list, we do the following:

1. Create a node and call it newNode.
2. Copy the info of the node (in the original list) into newNode.
3. Insert newNode at the end of the list being created.

The definition of the function copyList is as follows:

```
template <class Type>
void linkedListType<Type>::copyList
                   (const linkedListType<Type>& otherList)
{
    nodeType<Type> *newNode; //pointer to create a node
    nodeType<Type> *current; //pointer to traverse the list

    if (first != NULL) //if the list is nonempty, make it empty
       destroyList();

    if (otherList.first == NULL) //otherList is empty
    {
        first = NULL;
        last = NULL;
        count = 0;
    }
    else
    {
        current = otherList.first; //current points to the
                                   //list to be copied
        count = otherList.count;

            //copy the first node
        first = new nodeType<Type>;  //create the node
        first->info = current->info; //copy the info
        first->link = NULL; //set the link field of the node to NULL
        last = first;   //make last point to the first node
        current = current->link; //make current point to the next
                                 // node

           //copy the remaining list
        while (current != NULL)
        {
            newNode = new nodeType<Type>;  //create a node
            newNode->info = current->info; //copy the info
            newNode->link = NULL; //set the link of newNode to NULL
```

```
            last->link = newNode; //attach newNode after last
            last = newNode; //make last point to the actual last
                            //node
            current = current->link; //make current point to the
                                     //next node
        }//end while
    }//end else
}//end copyList
```

The function `copyList` contains a `while` loop. The number of times the `while` loop executes depends on the number of items in the list. If the list contains *n* items, the `while` loop executes *n* times. Therefore, the function `copyList` is of $O(n)$.

Destructor

The destructor deallocates the memory occupied by the nodes of a list when the class object goes out of scope. Because memory is allocated dynamically, resetting the pointers `first` and `last` does not deallocate the memory occupied by the nodes in the list. We must traverse the list, starting at the first node, and delete each node in the list. The list can be destroyed by calling the function `destroyList`. Therefore, the definition of the destructor is as follows:

```
template <class Type>
linkedListType<Type>::~linkedListType() //destructor
{
    destroyList();
}
```

Copy Constructor

Because the `class linkedListType` contains pointer data members, the definition of this class contains the copy constructor. Recall that, if a formal parameter is a value parameter, the copy constructor provides the formal parameter with its own copy of the data. The copy constructor also executes when an object is declared and initialized using another object.

The copy constructor makes an identical copy of the linked list. This can be done by calling the function `copyList`. Because the function `copyList` checks whether the original is empty by checking the value of `first`, we must first initialize the pointer `first` to `NULL` before calling the function `copyList`.

The definition of the copy constructor is as follows:

```
template <class Type>
linkedListType<Type>::linkedListType
                     (const linkedListType<Type>& otherList)
{
    first = NULL;
    copyList(otherList);
}//end copy constructor
```

Overloading the Assignment Operator

The definition of the function to overload the assignment operator for the **class** `linkedListType` is similar to the definition of the copy constructor. We give its definition for the sake of completeness.

```
     //overload the assignment operator
template <class Type>
const linkedListType<Type>& linkedListType<Type>::operator=
                      (const linkedListType<Type>& otherList)
{
    if (this != &otherList) //avoid self-copy
    {
        copyList(otherList);
    }//end else

     return *this;
}
```

5

The destructor uses the function `destroyList`, which is of $O(n)$. The copy constructor and the function to overload the assignment operator use the function `copyList`, which is of $O(n)$. Therefore, each of these functions are of $O(n)$.

TABLE 5-6 Time-complexity of the operations of the `class linkedListType`

Function	Time-complexity
`isEmptyList`	$O(1)$
default constructor	$O(1)$
`destroyList`	$O(n)$
`front`	$O(1)$
`end`	$O(1)$
`initializeList`	$O(n)$
`print`	$O(n)$
`length`	$O(1)$
`front`	$O(1)$
`back`	$O(1)$
`copyList`	$O(n)$