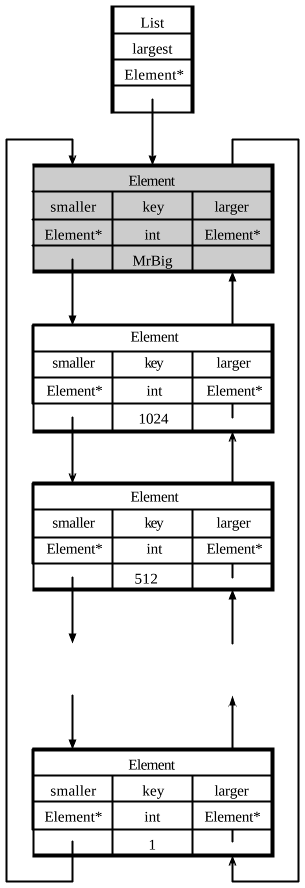


Figure 1. Ordered circular linked-list with sentinel.

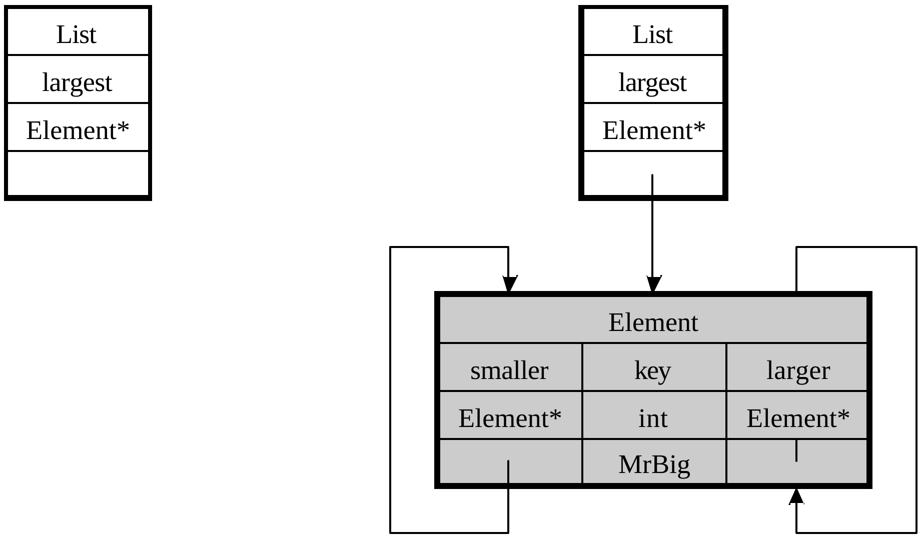
```
class List {
    struct Element {
        Element* smaller;
        int key;
        Element* larger;
        Element(int k);
        Element(int k,Element* s,Element* l);
    };
    Element* largest;
    const int MrBig;

    void Kill(Element* e);
public:
    List();
    ~List();
    void Insert(int key);
    void Remove(int key);
};
```

Figure 2. class *List*.

```
List::Element::Element(int k):key(k) {}
List::Element::Element(Element* s, int k,Element* l):smaller(s),key(k),larger(l) {}
```

Figure 2. Constructors for **struct** *Element*.



Before After

Figure 3. Diagram of constructor *List*()

```
List::List():MrBig(INT_MAX)
{    Element* e=new Element(MrBig);
     largest=e->smaller=e->larger=e;
}
```

Figure 4. Code for constructor *List*()

List::*List*()

1. Initialize member *MrBig* to *INT_MAX*. *INT_MAX* is the largest integer value. *INT_MAX* is defined in <*limits.h*>
2. In the remaining steps allocate and initialize the sentinel element.
3. Declare local variable *e* of type *Element**.
4. Allocate storage for a new element initializing *Element* member *key* to *MrBig*. Assign the pointer to the new element to variable *e*.
5. To make the list circular assign the address of the new element to Element members *smaller* and *larger*. The address of the new element is stored in variable *e*.
6. Bind the sentinel element to the list by assigning the address of the sentinel created in steps 3, 4, and 5 to *List* member *largest*. The address of the new element is stored in variable *e*.

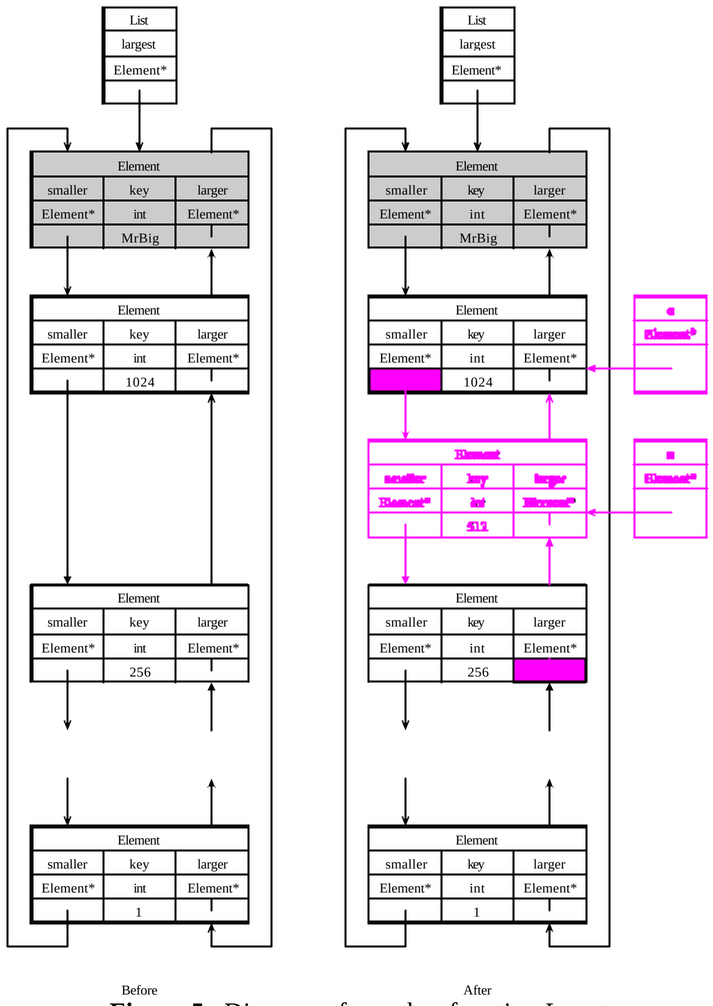


Figure 5. Diagram of member function *Insert*

```
void List::Insert(int key)
{   Element* e=largest->larger;
    while (key>e->key) e=e->larger;
    if (key==e->key) return;
    Element* n=new Element(key,e->smaller,e);
    e->smaller->larger=n;
    e->smaller=n;
}
```

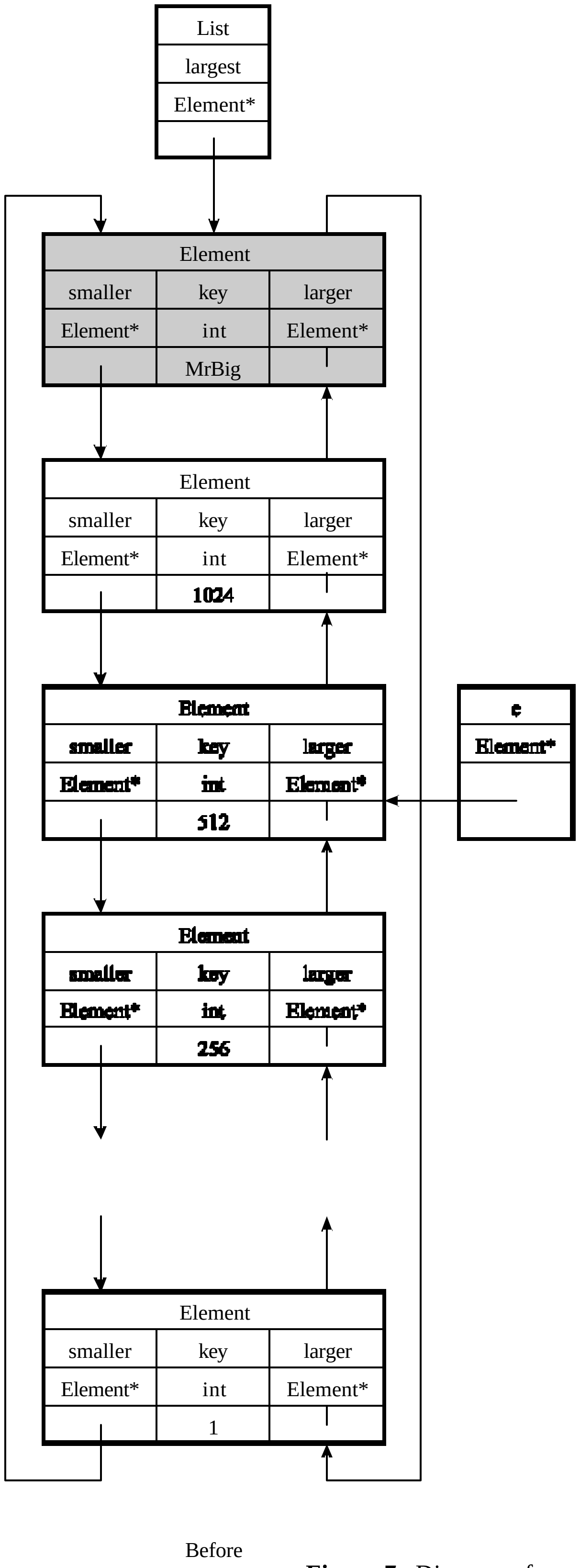
Figure 6. Code for member function *Insert*

void *List*::*Insert*(**int** *key*)

1. Search the list, starting with the smallest element, until an element just larger or, possibly equal, to the key is found. The sentinel at the largest end of the list guarantees that the search will find an element larger than the input key.
    1.1. Find the smallest element on the list. The element larger than the largest element is the smallest element.
    1.2. Search the list moving from the smallest element to an element containing a key just larger than the input key. Element member *larger* points to the next larger element. Traverse the list by assigning the value of *Element* member *larger* to local variable *e*. Making sure that the value of

the input parameter *key* is greater than *Element* member *key* guarantees that the search will stop. The search will stop on the sentinel if nowhere else.

2. Compare input parameter *key* against the key in the element found in the search. If the two keys are equal retum. Duplicate keys are not permitted.
3. At this point we are sure that the value of the input parameter *key* is unique. Further we are sure that local variable *e* points to an element having a *key* just larger than the value of the input parameter *key*.
4. Create a new element and bind the new element to the list.
   - 4.1. Initialize new *Element* member *key* to the value of the input parameter *key*.
   - 4.2. Initialize new *Element* member *smaller* to point to the element just *smaller* than the element referenced by local variable *e*. Make the new element point to the element just smaller than the element found in the search.
   - 4.3. Initialize new *Element* member *larger* to point to the element found in the search: it is, as previously declared, just larger than the value of the input parameter key.
5. Bind the list to the new element. Since the list is a doubly linked list, there are two members in the list that need to be changed. Member *smaller* in some element needs to be changed to point to the new element. Member *larger* in some element needs to be changed to point to the new element.
   - 5.1. Find the element just smaller than the element found by the search (*e->smaller*). Member *larger* in that element (*e->larger->smaller*) must now point to the new element.
   - 5.2. Find the element (*e*) just larger than the new element. That element is the element found in the search. Member *smaller* (*e->smaller*) must be changed to point to the new element.



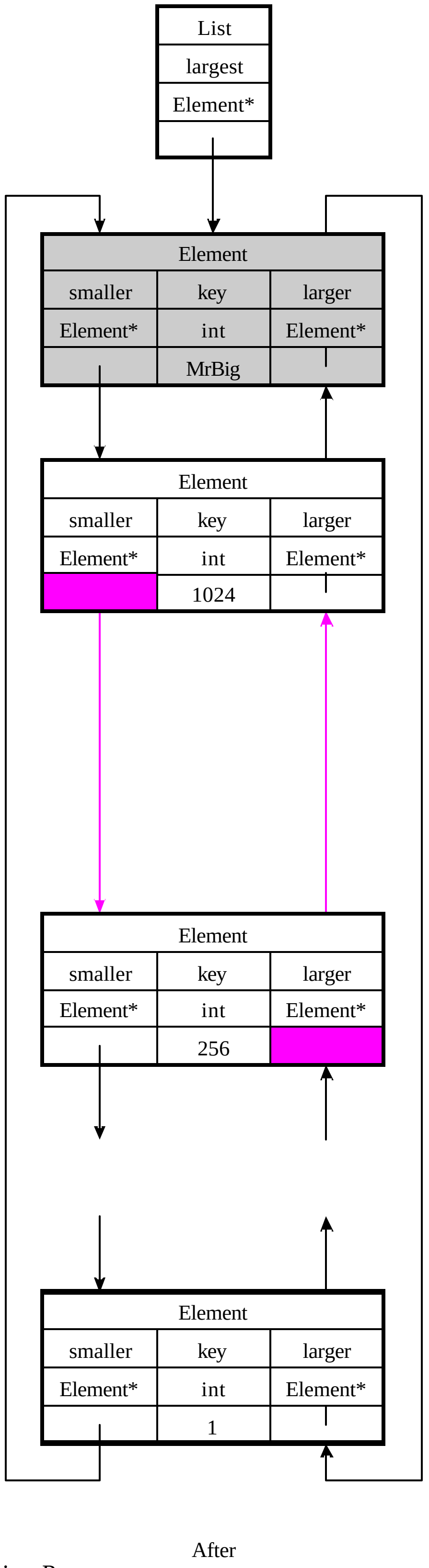


Figure 7. Diagram of member function *Remove*

```
void List::Remove(int key) { … }
```

Figure 8. Code for member function *Remove*

void *List*::*Remove*(**int** *key*)

1. Search the list, starting with the smallest element, until an element just larger or, possibly equal, to the key is found. The sentinel at the largest end of the list guarantees that the search will find an element larger than the input key.
   1.1. Find the smallest element on the list. The element larger than the largest element is the smallest element.
   1.2. Search the list moving from the smallest element to an element containing a key just larger than the input key. Element member *larger* points to the next larger element. Traverse the list by assigning the value of *Element* member *larger* to local variable *e*. Making sure that the value of the input parameter *key* is greater than *Element* member *key* guarantees that the search will stop. The search will stop on the sentinel if nowhere else.
2. Compare input parameter *key* against the key in the element found in the search. If the two keys are not equal return. An element having a key equal to the input parameter key does not exist.
3. At this point we are sure that the search has resulted in finding the element we wish to delete.
4. Bind the smaller element to the larger and vice versa so that the element found by the search is freed from list.
   4.1. Bind the smaller element to the larger. The smaller element is *e->smaller*. Member *larger* must be bound to the larger element. The larger element is *e->larger*.
   4.2. Bind the larger element to the smaller. The larger element is *e->larger*. Member *smaller* must be bound to the smaller element. The smaller element is *e->smaller*.

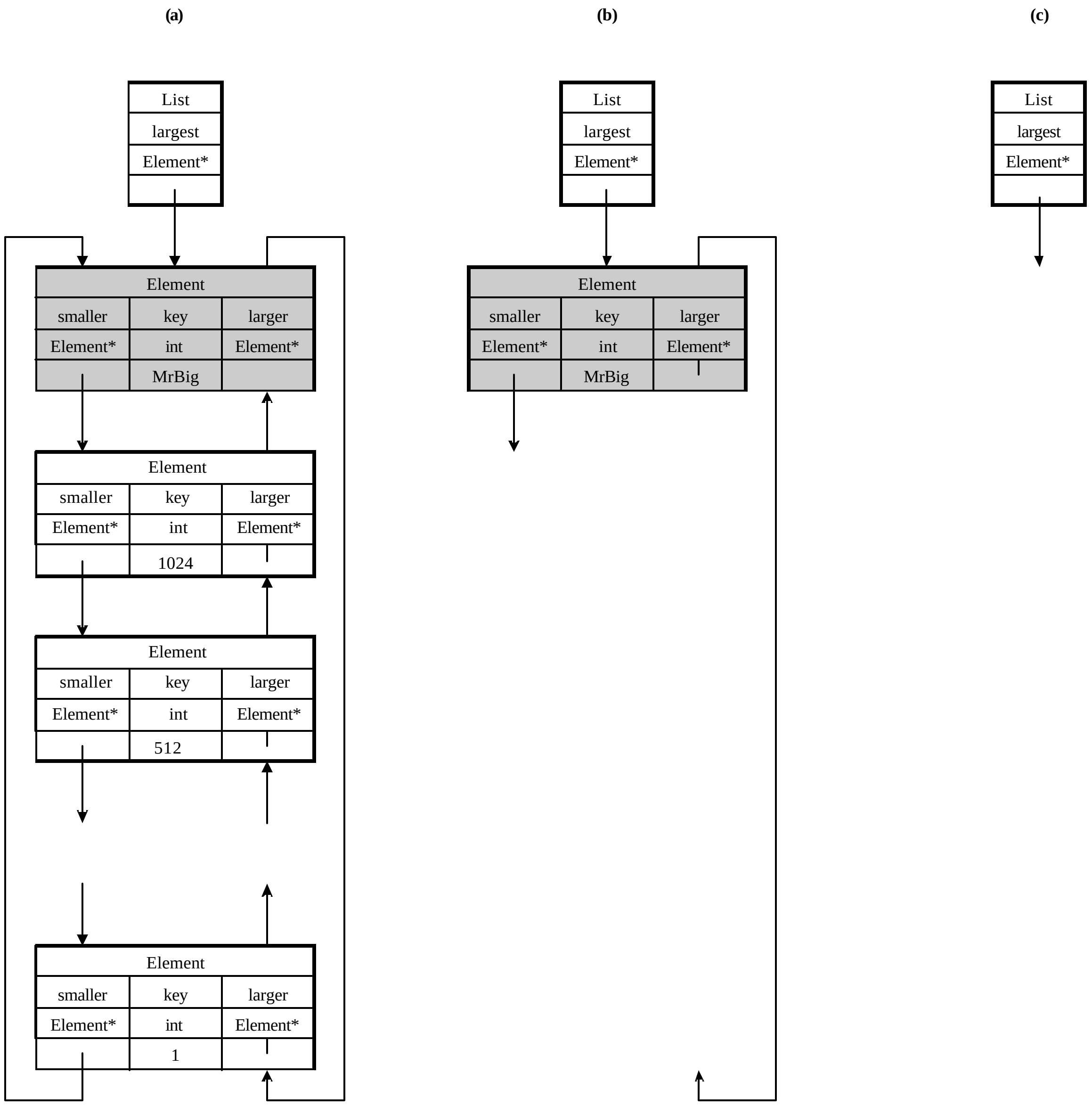


Figure 9. Diagram for the destructor **~*List*()**

```
List::~List()
{    Kill(largest->larger);
     delete largest;
}
```

Figure 10. Member function **~*List*()**

List::~*List*()

1. Member function *Kill* removes all elements on the list except the sentinel. Figure 9 (a) shows the list before function *Kill* is called and Figure 9 (b) shows the list after the *Kill* returns.
2. After all elements are removed except the sentinel, remove the sentinel also. Figure 9 (c) shows the list after the sentinel has been removed.

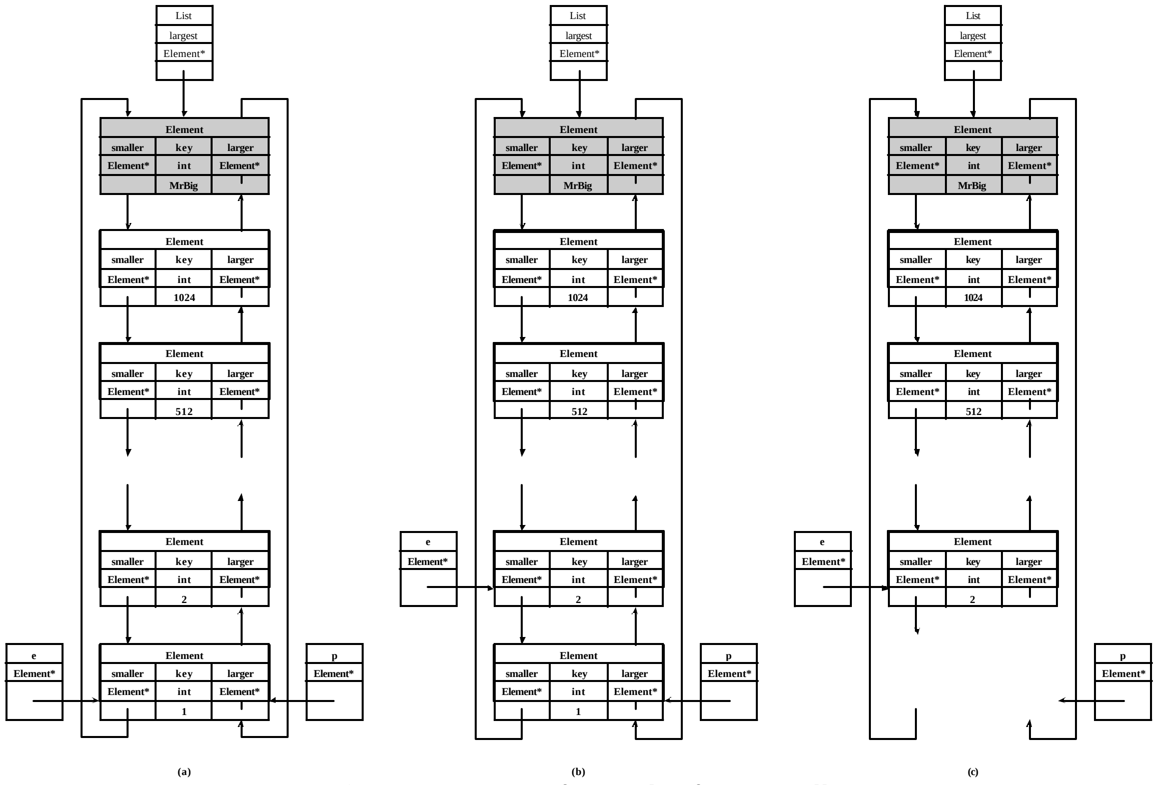


Figure 11. Diagram for member function *Kill*

```
void List::Kill(Element* e) { … }
```

Figure 12. Member function *Kill*

void *List*::~*Kill*(*Element** *e*)

1. Parameter *e* points to the smallest element when function *Kill* is called.
2. The *while-statement* iterates through all elements on the list except the sentinel.
3. After entering the body of the *while-statement* local variable *p* is assigned to point to the same element as parameter *e*. Figure 11 (a) depicts the list just after the body of the *while-statement* has been entered and parameter *e* points to the first element to be removed.
4. Parameter *e* is advanced to the next larger element as shown in Figure 11 (b) leaving local variable *p* to point to the smaller element.
5. Local variable *p* is used to remove the element it points to as shown in Figure 11 (c).
6. Local variable *p* goes out of scope at the end of the *while-statement* body.
7. The foregoing process defined by steps 3 through 6 is repeated on all the remaining elements except the sentinel.