Computer Science 102
Lab 8

In this lab you will you will extend the C++ version of your linked list manager. A sample *main.c, list.h and entity.h* are provided for you. You will note in *entity.h* that the *e_t* is now a true *C*++ class.

Step 1: create ~*list_t* and ~*link_t* destructors. The *list_t* destructor should process the entire list using *delete* to delete each link. The ~*link_t* destructor should use *delete* to delete the associated instances of the *e_t* class*.*

Step 2: make sure that your *add()* method sets the *current* pointer to the *last element* in the list (i.e. the one that was just added).

Step 3: implement the new methods *insert()* and *remove()* as class methods in the *list_t* class.

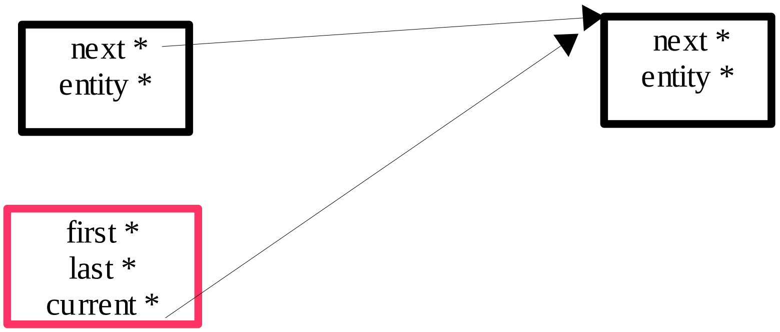
```
class list_t
{
public:
   list_t(void);                    // constructor
   ~list_t (void);                  // destructor
   void   add(void *);              // add entity to end of list
   void   insert(void *);           // insert entity before current
   void   remove(void );            // delete current entity
   void   start(void);              // set current to start of list
   void   get_next(void);           // advance to next element in list

private:
   link_t *first;                   // first link
   link_t *last;                    // last link
   link_t *current;                 // current link.
};
```

void insert(void *)

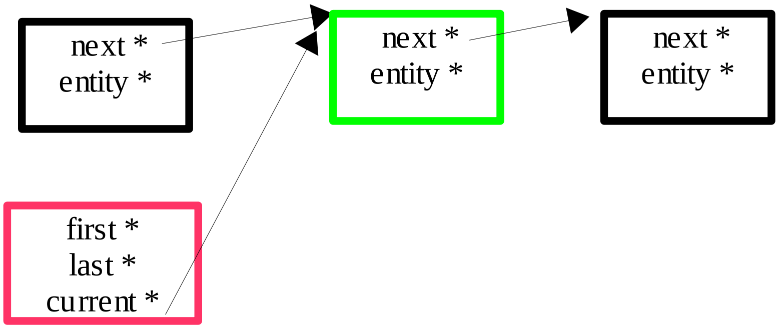
The *insert(void *)* method has a mission somewhat similar to *add()* in that does cause a new *e_t* to enter the list. However, unlike *add()* which always appends the new *link_t* to the end of the existing list*,* the *insert()* method must insert the new *link_t* directly in front of the existing *current* link.

When the *insert()* operation is complete the *next* pointer of the preexisting predecessor of *current* should point to the new *link_t,* the *next* pointer of the *new link_t* should have the previous value of *current* and *current* should have the address of the new *link_t.* In the diagram shown below existing *link_t's* are shown in black, the *list_t* in red and the inserted *link_t* in green.

BEFORE:



AFTER:



As with *add* the best way to make this work reliably is to have special case code that depends on the existing state of the list. For *add*() there were two relevant states *empty* and *not empty.* Here there are 3:

1 - the list is presently empty (*current* == *first* == *last* == *NULL*). In this case *insert()* should just call *add()*

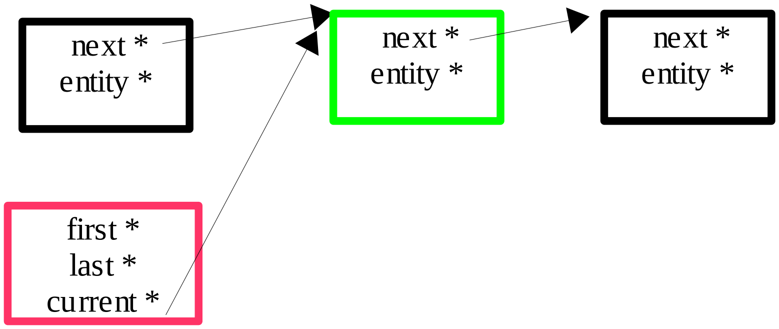
2 - Current points to the start of the list (*current* == *first).* In this case the *next* pointer of the new *link_t* must point to the pre-existing first element and the *first* and *current* pointers set to the new *link_t.*

3 - The current element is not first. In this case you will need a local variable *link_t *traverse.* Set *traverse* to *first* and then process the list until *traverse->getnext()* == *current.* At this point in the *BEFORE* picture *traverse* will point to the *link_t* on the *left* and *current* to the *link_t on the right.*

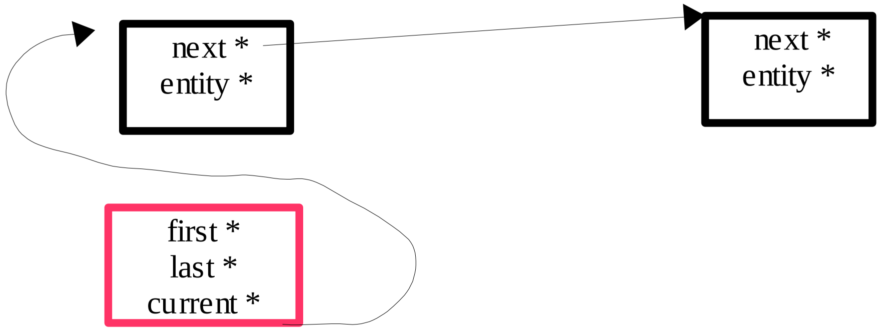
void remove(void)

The *remove* method should *delete* the *current* link_t. If the deleted element is not the first element in the list, the *current* pointer should point to the pre-existing predecessor of the deleted link. If the first link is deleted then *current* should point to its successor.

BEFORE



AFTER



Special cases to be considered include.

1- List already empty (do nothing)
2- List contains only a single element. Delete it and initialize first, last, and current
3- Deleting the first element of a list that has more than one element (have to reset first)
4- Deleting not the first element (need to traverse the list looking for the element that points to the current element)
4b- Deleting the last element of a list containing more than one element (have to reset last)

Debugging approach -

You will note that the *main.c* program has large sections disabled via the

```
#if 0
  disabled stuff

#endif
```

The proper way to debug this stuff is to enable the operations that are disabled *ONE AT A TIME.* Output produced with *all* code enabled may be found in lab8.txt.

In this lab you will submit a single file, list.cpp that includes the new and updated class methods constructed as part of this lab.

```
     sendlab.102.labsection#  lab#  list.cpp
```