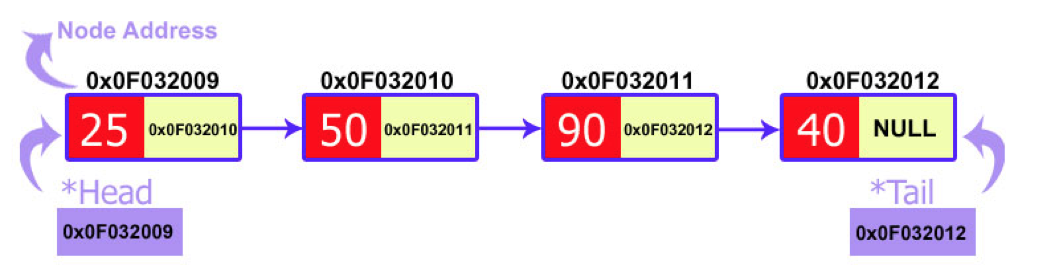
# To do list

* There is a python way of implementing linked list or circular linked list. Check it out and get familiar with more python.
* Some examples for circular linked list. [1] One way of implementing shared pointer as in IB? [2] If no insertion or deletion involved, I may use vectors as in the way of shuffling request of market data.

# A Comprehensive Guide To Singly Linked List Using C++

First check the LinkedList.cpp and comments. If I can understand very well about that file, then I don’t need read the following.

No matter how many nodes are present in the linked list, the very first node is called **head** and the last node is called the **tail**. If there is just one node created then it is called both head and tail. **My Comments**: Note head is not so special as other pointers to the node. We of course can have head->data, and head->next etc. In fact when the first temp pointer to node is created, both head and tail are assigned with temp.



## Implementation of Linked List Using C++

**struct** **node**

{

**int** data;

node \*next;

};

## Creation of Linked List Using C++

Now, we need a class which will contain the functions to handle the nodes. This class should have two important pointers, i.e. head and tail. The constructer will make them **NULL** to avoid any garbage value.

**class** **list**

{

Private:

node \*head, \*tail;

**public**:

list():head(NULL), tail(NULL) {}

};

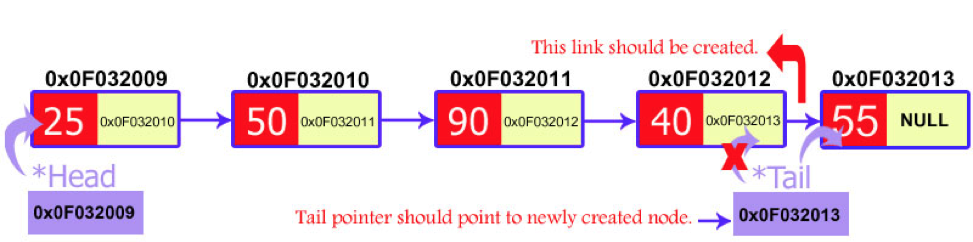
Now, we will write a function for the node creation. The process of creating node is very simple. We need a pointer of a node type (which we defined) and we will insert the value in its data field. The next field of node would be declared as NULL as it would be the last node of linked list.

Now, the function will have a very special case that we want to know what would happen if the linked list is still empty? We will have to check it. Do you remember that the head points to the first node? It means if the head is equal to NULL then we can conclude that the linked list is empty. I have also told you before that if there is just one node (which we are going to create) in linked lists, then it is called both head and tail.

And if a linked list is created already, then we would insert this node at the end of the linked list. We know that the last node is called a tail. So we are going to create this newly created node next to a tail node.

The creation of a new node at the end of linked list has 2 steps:

1. Linking the newly created node with tail node. Means passing the address of a new node to the next pointer of a tail node.
2. The tail pointer should always point to the last node. So we will make our tail pointer equal to a new node.



The C++ code for the creation of new a node would like this:

**void** **createnode**(**int** value) // Note this is a member function of class list and thus head, tail

{

//node \*temp = (node\*) malloc(sizeof(node)); If we do this way, then we need cast into (node\*)

//because the return type of malloc() is void pointer.

node \*temp=**new** node;

temp->data=value;

temp->next=NULL; //New node is final node and thus next = NULL, but this is not always?

**if**(head==NULL){

head=temp;

tail=temp;

temp=NULL; //**Why do I need this here but not in the following else branch**?

}

**else** {//\*\*\*\* See the picture of this step earlier to easily write the code.

tail->next = temp;

tail = temp;

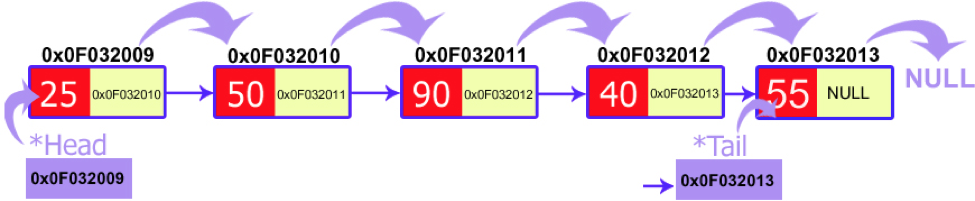
}

}

My Comments: Sensitive to the difference in two local branches above. In the head == NULL branch, involve both head and tail and no ->next involved. In the second branch, only tail is involved, and one of two assignments involve ->next. [

## Displaying Linked List Using C++

Now we have a working linked list which allows creating nodes. If we want to see that what is placed in our linked list then we will have to make a display function. The logic behind this function is that we make a temporary node (why? See comments in the code below) and pass the address of the head node to it. Now we want to print all the nodes on the screen. So we need a loop which runs as many times as nodes exist. Every node contains the address of the next node so the temporary node walks through the whole linked list. If the temporary node becomes equal to NULL then the loop would be terminated.



**void** **display**()

{

node \*temp=**new** node;

temp=head; //We must set a temporary node here because we cannot modify head.

**while**(temp!=NULL)

{

cout<<temp->data<<"\t";

temp=temp->next;

}

}

Both createnode() and display() functions would be written under **public** section of class.

The basic framework of a singly-linked list is ready. Now it is the time to perform some other operations on the list. Basically, two operations are performed on linked lists:

1. Insertion
2. Deletion

### Insertion

Inserting a new node in the linked list is called insertion.

A new node is created and inserted in the linked list.

There are three cases considered while inserting a node:

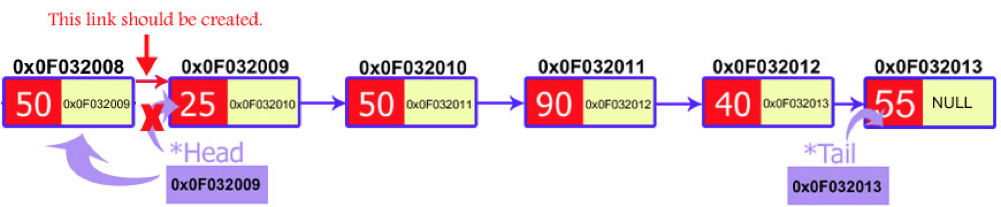
1. Insertion at the start
2. Insertion at the end
3. Insertion at a particular position

#### Insertion at the Start

Insertion of a new node is quite simple. It is just a 2-step algorithm which is performed to insert a node at the start of a singly linked list.

1. New node should be connected to the first node, which means the head. This can be achieved by putting the address of the head in the next field of the new node.
2. New node should be considered as a head. It can be achieved by declaring head equals to a new node.

The diagrammatic demonstration of this process is given below:



The code for this process is:

**void** **insert\_start**(**int** value)

{

//\*\*\*\* Here I assume there is already non-NULL list there. Otherwise,

//\*\*\*\* we need consider the case of head = NULL.

node \*temp=**new** node;

temp->data=value;

temp->next=head; //Be very careful about the order of the two statements?

head=temp;

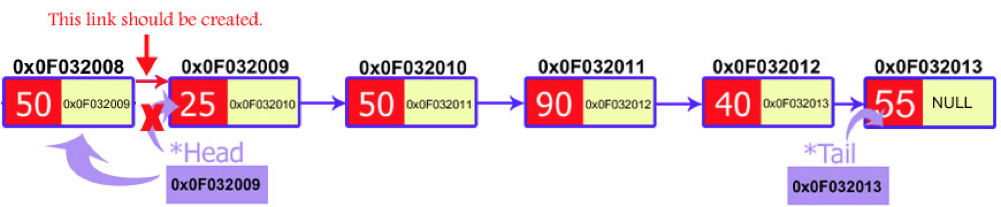
}

#### Insertion at the End

The insertion of a node at the end of a linked list is the same as we have done in node creation function. If you noticed then, we inserted the newly created node at the end of the linked list. So this process is the same.

#### Insertion at Particular Position

The insertion of a new node at a particular position is a bit difficult to understand. In this case, we don’t disturb the head and tail nodes. Rather, a new node is inserted between two consecutive nodes. So, these two nodes should be accessible by our code. We call one node as current and the other as previous, and the new node is placed between them. This process is shown in a diagram below:



Now the new node can be inserted between the previous and current node by just performing two steps:

1. Pass the address of the new node in the next field of the previous node.
2. Pass the address of the current node in the next field of the new node.

We will access these nodes by asking the user at what position he wants to insert the new node. Now, we will start a loop to reach those specific nodes. We initialized our current node by the head and move through the linked list. At the end, we would find two consecutive nodes.

**void** **insert\_position**(**int** pos, **int** value)

{

node \*pre=**new** node;

node \*cur=**new** node;

node \*temp=**new** node;

cur=head; //First set cur to node 0 or head.

**for**(**int** i=1; i<pos; i++) //Start from 1.

{

pre=cur; **//\*\*\*\* two similar statements used in many places (see below)**

cur=cur->next;

}

temp->data=value;

pre->next=temp; // Key is got pre and current pointer, and then insert is very easy.

temp->next=cur;

}

### Deletion:

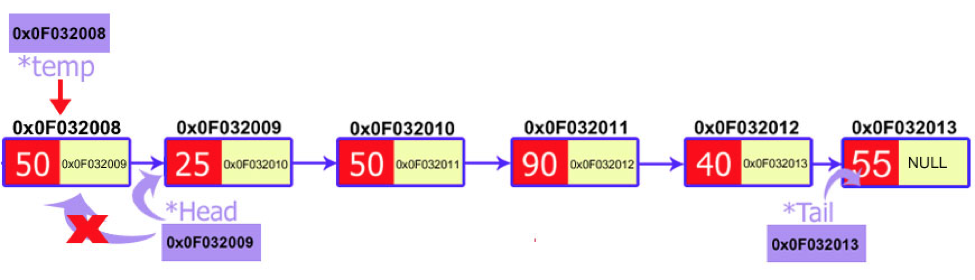
So, you have become familiar with linked list creation. Now, it’s time to do some manipulation on the linked list created. Linked lists provide us the great feature of deleting a node. The process of deletion is also easy to implement. The basic structure is to declare a temporary pointer which points the node to be deleted. Then a little bit of working on links of nodes. There are also three cases in which a node can be deleted:

1. Deletion at the start
2. Deletion at the end
3. Deletion at a particular position

#### Deletion at the Start

In this case, the first node of the linked list is deleted. I know, you remember that the first node is called the head. So, we are going to delete the head node. The process of deletion includes:

1. Declare a **temp** pointer and pass the address of the first node, i.e. head to this pointer.
2. Declare the second node of the list as head as it will be the first node of linked list after deletion.
3. Delete the temp node.



The C++ code for this process is given below:

**void** **delete\_first**()

{

node \*temp=**new** node;

temp=head; //I need keep a copy for this for deleting it later. Otherwise only head = head->next is enough

head=head->next;

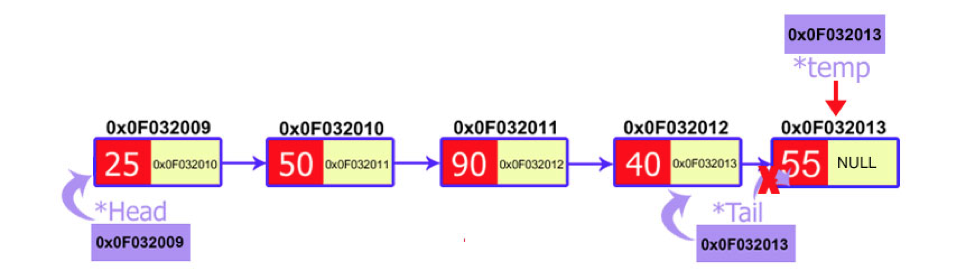
**delete** temp; // **Never forget about this**!

}

My Comments: [1] temp = head; head = head->next cannot change order. Otherwise when I delete temp; there will be big error.

#### Deletion at the End

Deletion of the last node is a bit difficult to understand than the first node. In the case of the first node, you just need access to the head and you can delete it. But in the case of the last node, you also need access to the second to the last node of the linked list as you will delete the last node and make the previous node as the tail of linked list.



Our algorithm should be capable of finding a node that comes before the last node. This can be achieved by traversing the linked list. We would make two temporary pointers and let them move through the whole linked list. At the end, the previous node will point to the second to the last node and the current node will point to the last node, i.e. node to be deleted. We would delete this node and make the previous node as the tail.

**void** **delete\_last**()

{

node \*current=**new** node;

node \*previous=**new** node;

current=head; // Same trick as other cases (see before).

**while**(current->next!=NULL) //Use this but not for loop. When inserting a specific position, use for loop.

{

previous=current; **//\*\*\*\* two similar statements used in many places (see below)**

current=current->next;

}

tail=previous;

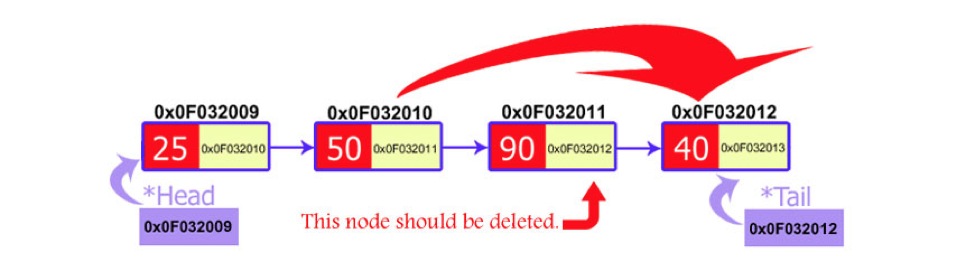
previous->next=NULL; //This is the key part.

**delete** current;

}

#### Deletion at a Particular Position

In linked list, we can delete a specific node. The process of deletion is simple. Here we don’t use the head and tail nodes. We ask the user to input the position of the node to be deleted. After that, we just move two temporary pointers through the linked list until we reach our specific node. Now, we delete our current node and pass the address of the node after it to the previous pointer. This way, the current node is removed from the linked list and the link is established between its previous and next node.



**void** **delete\_position**(**int** pos)

{

node \*current=**new** node;

node \*previous=**new** node;

current=head;

**for**(**int** i=1; i<pos; i++)

{

previous=current; **//\*\*\*\* two similar statements used in many places (see below)**

current=current->next;

}

previous->next=current->next;

}

My Comments: Why the author did not free the space of the deleted node?

I have made a complete project of a linked list for you. It performs all the functions described above. You can download it from this [Github](https://github.com/kamal-choudhary/singly-linked-list" \t "_blank). I have downloaded and may study it later.

# Circular Singly Linked List | Insertion

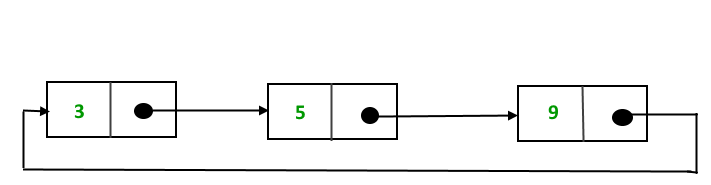
**First check the code I wrote. Read the following only when I cannot understand the code there.**

**Advantages of Circular Linked Lists:**  
**1)**Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

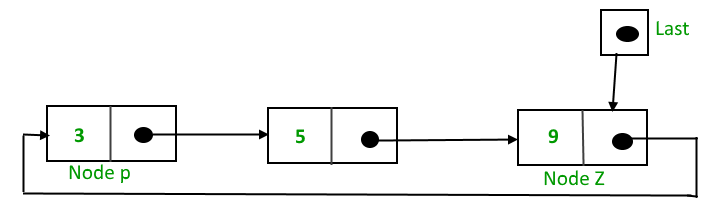
**2)**Useful for implementation of queue. Unlike [this](http://quiz.geeksforgeeks.org/queue-set-2-linked-list-implementation/)implementation, we don’t need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.

**3)** Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list. (Source <http://web.eecs.utk.edu/~bvz/cs140/notes/Dllists/>)

**4)** Circular Doubly Linked Lists are used for implementation of advanced data structures like [Fibonacci Heap](http://en.wikipedia.org/wiki/Fibonacci_heap).



In this post, implementation and insertion of a node in a Circular Linked List using singly linked list are explained.

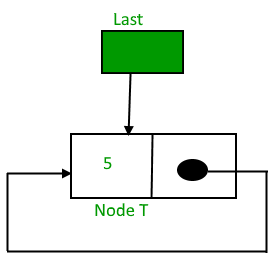
**Implementation**  
To implement a circular singly linked list, we take an external pointer that points to the last node of the list. If we have a pointer last pointing to the last node, then last -> next will point to the first node.  
  
The ponter *last* points to node Z and last -> next points to node P.

***Why have we taken a pointer that points to the last node instead of first node ?***  
For insertion of node in the beginning we need traverse the whole list. Also, for insertion at the end, the whole list has to be traversed. If instead of *start* pointer we take a pointer to the last node then in both the cases there won’t be any need to traverse the whole list. So insertion in the begging or at the end takes constant time irrespective of the length of the list.

**Insertion**  
A node can be added in three ways:

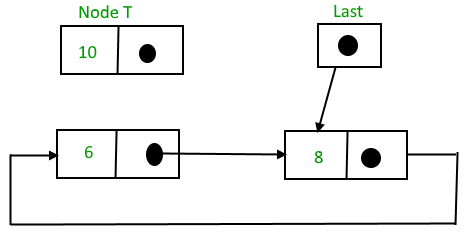
* Insertion in an empty list
* Insertion at the beginning of the list
* Insertion at the end of the list
* Insertion in between the nodes

**Insertion in an empty List**

Initially when the list is empty, *last* pointer will be NULL.  
  
After inserting a node T,  
  
After insertion, T is the last node so pointer *last* points to node T. And Node T is first and last node, so T is pointing to itself.

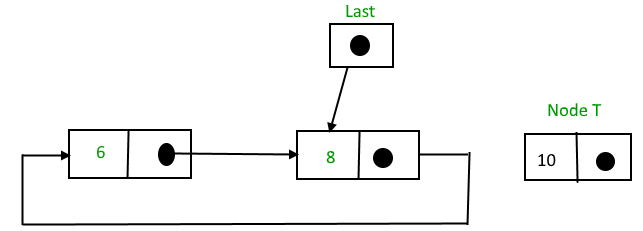
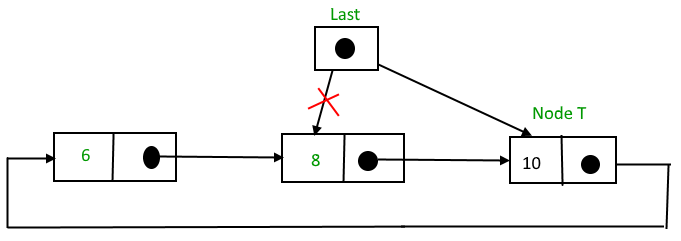
|  |
| --- |
| struct Node \*addToEmpty(struct Node \*last, int data)  {       if (last != NULL)  return last;      struct Node \*last =  (struct Node\*)malloc(sizeof(struct Node));      last -> data = data;      last -> next = last;      return last;  } |

**Insertion at the beginning of the list**

To Insert a node at the beginning of the list, follow these step:  
1. Create a node, say T.  
2. Make T -> next = last -> next.  
3. last -> next = T.  
  
After insertion,

|  |
| --- |
| https://cdncontribute.geeksforgeeks.org/wp-content/uploads/CircularSinglLinkedList5.png  struct Node \*addBegin(struct Node \*last, int data)  {    if (last == NULL)       return addToEmpty(last, data);      // Creating a node dynamically.    struct Node \*temp          = (struct Node \*)malloc(sizeof(struct Node));      // Assigning the data.    temp -> data = data;      // Adjusting the links.    temp -> next = last -> next;    last -> next = temp;      return last;  } |

**Insertion at the end of the list**

To Insert a node at the end of the list, follow these step:  
1. Create a node, say T.  
2. Make T -> next = last -> next;  
3. last -> next = T.  
4. last = T.  
  
After insertion,  
  
Function to insert node in the end of the List,

|  |
| --- |
| struct Node \*addEnd(struct Node \*last, int data)  {    if (last == NULL)       return addToEmpty(last, data);      // Creating a node dynamically.    struct Node \*temp =          (struct Node \*)malloc(sizeof(struct Node));      // Assigning the data.    temp -> data = data;      // Adjusting the links.    temp -> next = last -> next;    last -> next = temp;    last = temp;      return last;  } |

**Insertion in between the nodes**

To Insert a node at the end of the list, follow these step:  
1. Create a node, say T.  
2. Search the node after which T need to be insert, say that node be P.  
3. Make T -> next = P -> next;  
4. P -> next = T.

|  |
| --- |
| Suppose 12 need to be insert after node having value 10, https://contribute.geeksforgeeks.org/wp-content/uploads/circularll-1.png After searching and insertion, https://contribute.geeksforgeeks.org/wp-content/uploads/CircularSinglyLinkedList9.png struct Node \*addAfter(struct Node \*last, int data, int item)  {      if (last == NULL)         return NULL;        struct Node \*temp, \*p;      p = last -> next;        // Searching the item.      do      {          if (p ->data == item)          {              // Creating a node dynamically.              temp = (struct Node \*)malloc(sizeof(struct Node));                // Assigning the data.              temp -> data = data;                // Adjusting the links.              temp -> next = p -> next;                // Adding newly allocated node after p.              p -> next = temp;                // Checking for the last node.              if (p == last)                  last = temp;                return last;          }          p = p -> next;      } while (p != last -> next);        cout << item << " not present in the list." << endl;      return last;  } |

Following is a complete program that uses all of the above methods to create a circular singly linked list.

|  |
| --- |
| #include<bits/stdc++.h>  using namespace std;    struct Node  {      int data;      struct Node \*next;  };    struct Node \*addToEmpty(struct Node \*last, int data)  {      // This function is only for empty list      if (last != NULL)        return last;        // Creating a node dynamically.      struct Node \*temp =             (struct Node\*)malloc(sizeof(struct Node));        // Assigning the data.      temp -> data = data;      last = temp;        // Creating the link.      last -> next = last;        return last;  }    struct Node \*addBegin(struct Node \*last, int data)  {      if (last == NULL)          return addToEmpty(last, data);        struct Node \*temp =              (struct Node \*)malloc(sizeof(struct Node));        temp -> data = data;      temp -> next = last -> next;      last -> next = temp;        return last;  }    struct Node \*addEnd(struct Node \*last, int data)  {      if (last == NULL)          return addToEmpty(last, data);        struct Node \*temp =          (struct Node \*)malloc(sizeof(struct Node));        temp -> data = data;      temp -> next = last -> next;      last -> next = temp;      last = temp;        return last;  }    struct Node \*addAfter(struct Node \*last, int data, int item)  {      if (last == NULL)          return NULL;        struct Node \*temp, \*p;      p = last -> next;      do      {          if (p ->data == item)          {              temp = (struct Node \*)malloc(sizeof(struct Node));              temp -> data = data;              temp -> next = p -> next;              p -> next = temp;                if (p == last)                  last = temp;              return last;          }          p = p -> next;      }  while(p != last -> next);        cout << item << " not present in the list." << endl;      return last;    }    void traverse(struct Node \*last)  {      struct Node \*p;        // If list is empty, return.      if (last == NULL)      {          cout << "List is empty." << endl;          return;      }        // Pointing to first Node of the list.      p = last -> next;        // Traversing the list.      do      {          cout << p -> data << " ";          p = p -> next;        }      while(p != last->next);    }    // Driven Program  int main()  {      struct Node \*last = NULL;        last = addToEmpty(last, 6);      last = addBegin(last, 4);      last = addBegin(last, 2);      last = addEnd(last, 8);      last = addEnd(last, 12);      last = addAfter(last, 10, 8);        traverse(last);        return 0;  } |