Linked Lists

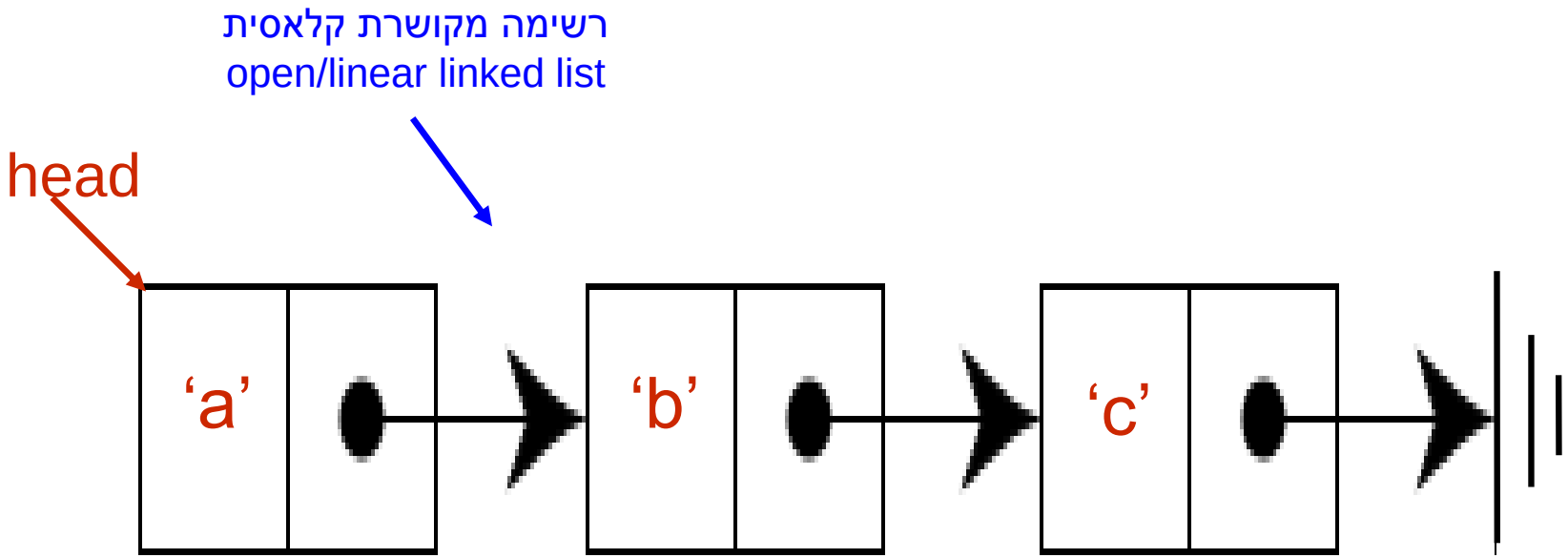
- כדי לייצג אוסף נתונים בזיכרון המחשב השתמשנו עד כה במבנה הנתונים שנקרא מערך – array. כידוע גודל המערך קבוע ונקבע מראש. בנוסף איברי המערך מאוחסנים בצורה רציפה בזיכרון המחשב.

- חלק מהחסרונות של מבנה הנתונים הזה
  - ✓ הצורך לתפוס חלק מהזיכרון מראש
  - ✓ המערך הוא בגודל סטטי
  - ✓ הוספת אבר פנימי במערך כרוכה בהזזת שאר האיברים

- האם קיים מבנה נתונים דינאמי פשוט יחסית ואשר מאפשר ביצוע פעולות בדומה למערך?
  - ✓ כן רשימה מקושרת (linked list).

Linked List
רשימה מקושרת

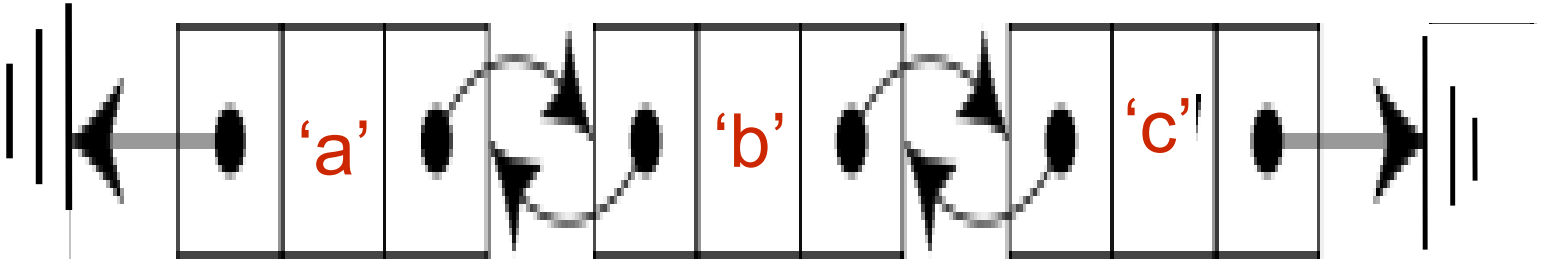
- אחד ממבני הנתונים הבסיסים במדעי המחשב
- מאפשר שמירת נתונים בצורה דינאמית ויעילה
- הרשימה המקושרת בנויה מאוסף איברים המפוזרים בזיכרון המחשב
- כל איבר מהווה צומת ברשימה (node)
- כל איבר מכיל מצביע לאיבר הבא ברשימה
- האיבר האחרון מצביע לשום מקום (null)



רשימה מקושרת
דו-כיוונית

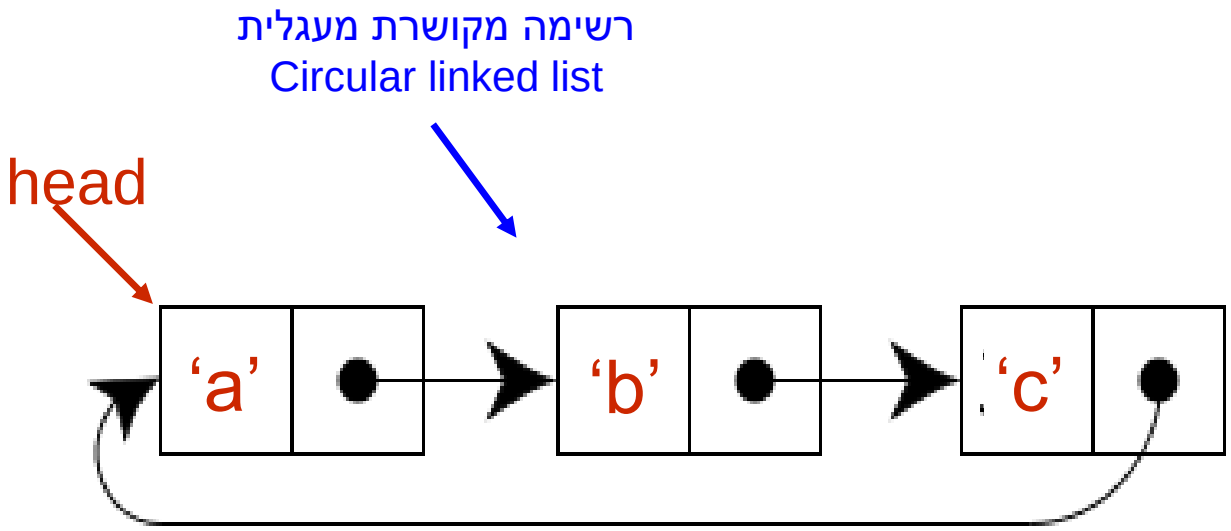
- רשימה מקושרת שבה כל איבר מצביע על האיבר שאחריו וזה שלפניו נקראת רשימה מקושרת דו-כיוונית.
- דרושים שני מצביעים לכך
- זה מאפשר הגעה לאיבר הקודם בעלות O(1)
- ברשימה המקושרת הקלאסית, שבה יש מצביע רק לאיבר הבא ברשימה, הגעה אל האיבר הקודם ברשימה דורש סריקה סדרתית מחדש של הרשימה O(n)

רשימה מקושרת דו-כיוונית
Doubly linked list



רשימה מקושרת
מעגלית

רשימה בה האיבר האחרון מצביע על האיבר הראשון נקראת רשימה מקושרת מעגלית



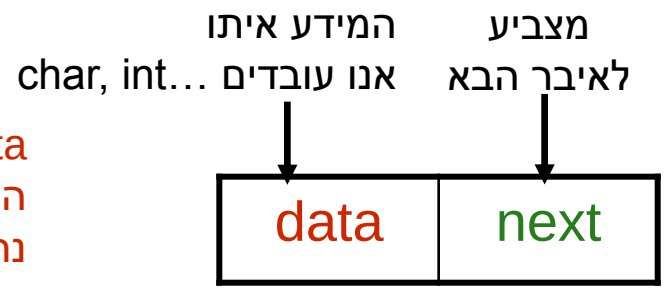
ייצוג איבר ברשימה
מקושרת ב- C

איך מייוצג איבר (Node) ברשימה ב C?
תשובה: ע"י שימוש Structure

```
typedef struct Node
{
    int data;
    struct Node* next
} Node;
```

data
הערה: המידע יכול להיות נתון אחד או יותר

pointer : מצביע לאיבר הבא ברשימה
הערה: המצביע של האיבר הבא ברשימה הוא מאותו הטיפוס של כל אברי הרשימה.



- הרשימה מכילה איברים (Nodes) כל איבר מורכב משני אלמנטים:
  - ✓הנתון, data, זהו המידע איתו אנו עובדים ולשמו יצרנו את מבנה הנתונים.
  - ✓מצביע לאיבר הבא ברשימה next.

פעולות בסיסיות על רשימה מקושרת

נתאר שלוש פעולות בסיסיות המתבצעות על רשימה מקושרת בגרסתה הקלאסית (single/linear list)

- ✓הוספת איבר
- ✓הסרת איבר
- ✓חיפוש איבר

הערה1
מצביע יכול לקבל את הערך Null כלומר המצביע אינו מצביע לשום מקום

הערה3
אם השדה head מצביע אל עבר Null, משמעות הדבר שהרשימה היא רשימה ריקה (רשימה שאין בה אף איבר).

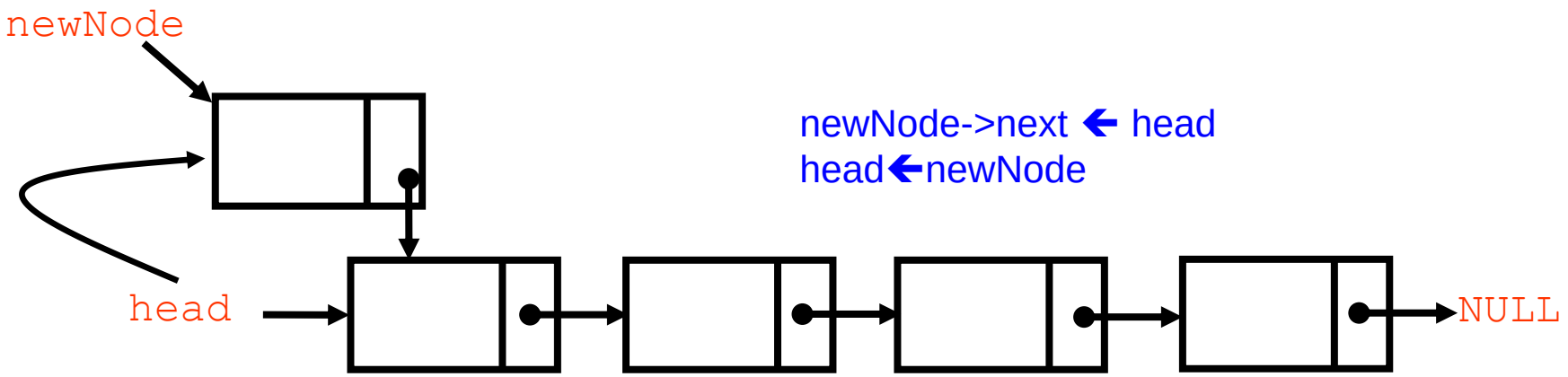
הערה2
אם שדה ה-next באיבר מסוים מצביע ל-Null, משמעות הדבר שזהו האיבר האחרון ברשימה

הערה 4
במקרה שבו head מצביע אל עבר Null (רשימה ריקה) יש להיזהר מכל ניסיון לגשת לשדה next ב-Null דבר שמוביל לשגיאת הרצה. לכן בעת תכנות עלינו לוודא שהרשימה אינה ריקה לפני ביצוע פעולה זו

הוספת איבר
בהתחלה

כדי להכניס את האיבר החדש לתחילת הרשימה, כלומר להופכו לאיבר הראשון, נצטרך לעדכן את המצביע head, אשר מראה מיהו האיבר הראשון ברשימה :-

1. צור את האיבר החדש
2. כוון את מצביע האיבר החדש אל עבר האיבר הראשון הקיים, שיהפוך עכשיו לאיבר השני ברשימה.
3. כוון את מצביע head אל עבר האיבר החדש, שכעת יהיה האיבר הראשון ברשימה

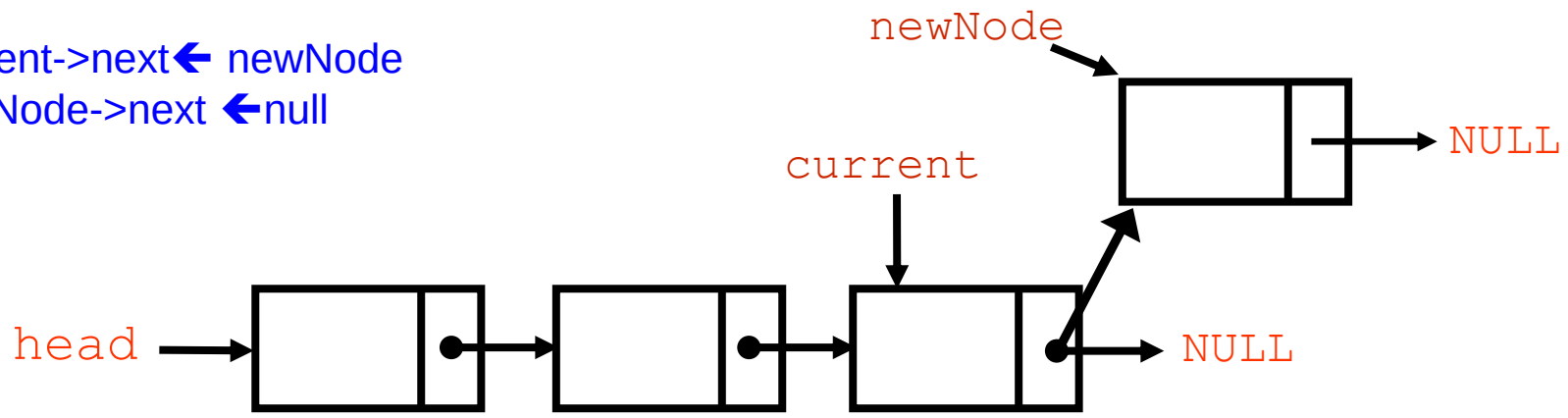


הוספת איבר
בסוף

כדי להכניס את האיבר החדש לסוף הרשימה, ולהופכו לאיבר האחרון – נצטרך לעדכן את מצביע האיבר האחרון ברשימה באופן הבא:

1. צור את האיבר החדש
2. כוון את מצביע האיבר האחרון ברשימה המקורית אל עבר האיבר החדש שיהפוך עכשיו לאיבר האחרון ברשימה.
3. עדכן את מצביע האיבר החדש שיצביע על null -כעת האיבר החדש יהיה האיבר האחרון ברשימה

Current->next← newNode
newNode->next ←null

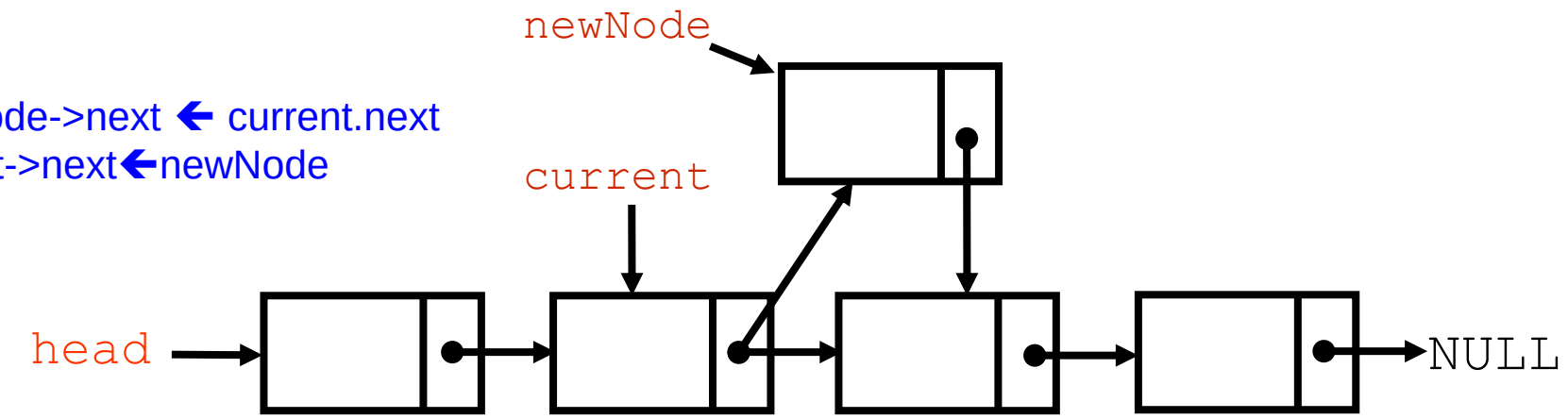


הוספת איבר
בין שני איברים עוקבים

ההכנסה מתבצעת בשלושה שלבים פשוטים (יש לשים לב לכך שסדר הפעולות הוא קריטי אם נכוון קודם את מצביע current אל עבר האיבר החדש נאבד את הקשר לכל שאר אברי הרשימה) :

1. צור את האיבר החדש - newNode
2. כוון את מצביע האיבר החדש אל עבר איבר הבא ברשימה
3. כוון את מצביע האיבר הקודם (current) אל עבר האיבר החדש

newNode->next ← current.next
current->next←newNode

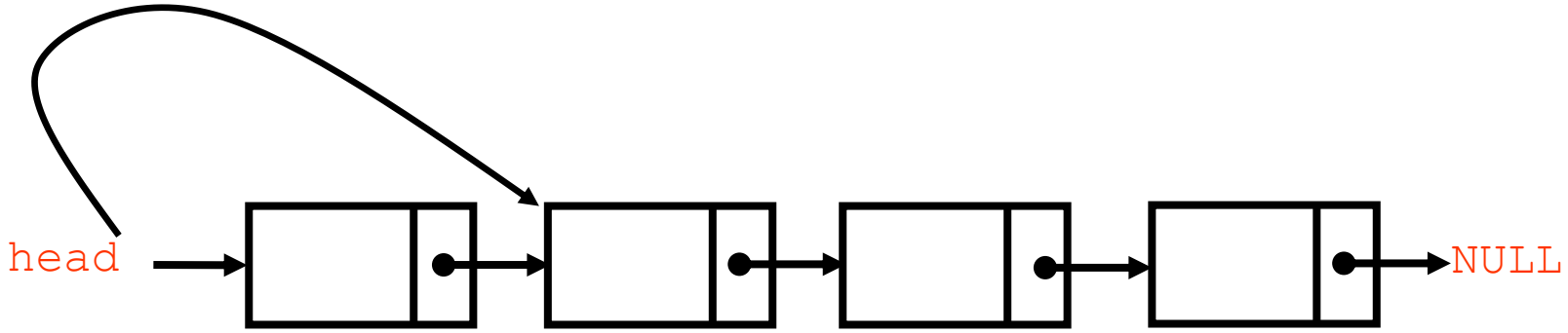


הסרת האיבר הראשון

כדי למחוק את האיבר הראשון ברשימה, יש לעדכן את המצביע head, אשר מראה מיהו האיבר הראשון ברשימה :

1. כוון את head אל האיבר השני
2. מחק את האיבר הראשון (ב java המחיקה מתבצעת באופן אוטומטי ברגע שבו הוטה המצביע לעבר איבר אחר אולם ב C/C++ יש לבצע שיחרור לזיכרון שהוקצה דינמית)

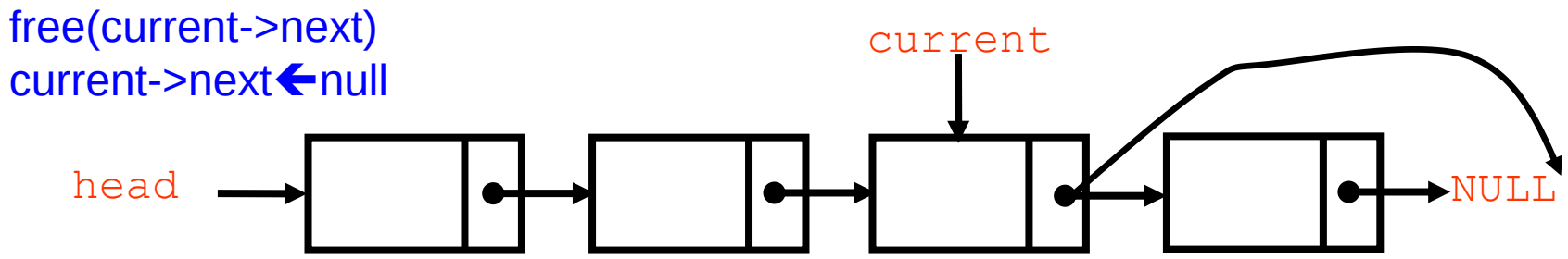
temp = head
head ← head->next
Free(temp)



הסרת האיבר האחרון

כדי למחוק את האיבר האחרון ברשימה, נצטרך לעדכן את המצביע של האיבר הלפני האחרון ברשימה :

1. עדכן את המצביע של האיבר הלפני אחרון שיצביע על Null
2. מחק את האיבר האחרון

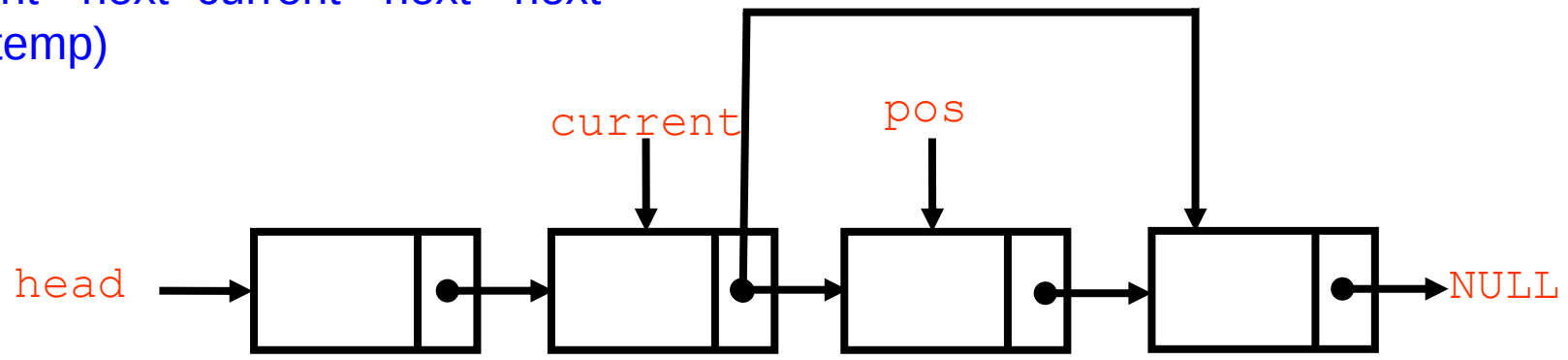


הסרת איבר
אמצעי

כדי למחוק איבר החסום בין שני איברים :

1. אתר את המקום המיועד להסרה
2. כוון את מצביע האיבר הקודם לאיבר שמיועד להסרה אל עבר האיבר שאחרי האיבר שמיועד להסרה.
3. מחק את האיבר שברצונך להסיר

```
temp = current->next
current->next=current->next->next
free(temp)
```



חיפוש איבר
ברשימה

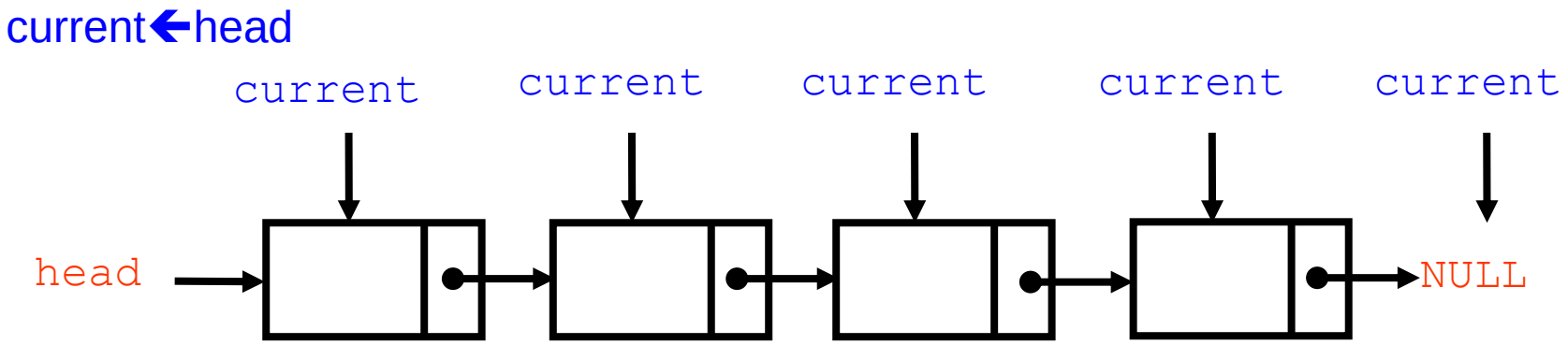
- חיפוש איבר נעשה ע"י מעבר סידרתי על איברי הרשימה עד אשר נמצא את האיבר המתאים או שהגענו לסוף הרשימה.

✓ התחל מהאיבר הראשון ברשימה

✓ כל עוד האיבר אינו Null

- אם זהו האיבר הרצוי – החזר את האיבר
- אם לא , עבור לאיבר הבא ברשימה

✓ החזר שהאיבר אינו קיים ברשימה



נתון המבנה Node אשר מתאר קודקוד המכיל int. צור רשימה בת שלושה איברים מסוג Node , והדפס אותה.

```
typedef struct Node
{
   int num;
   struct Node* next
}Node;
```

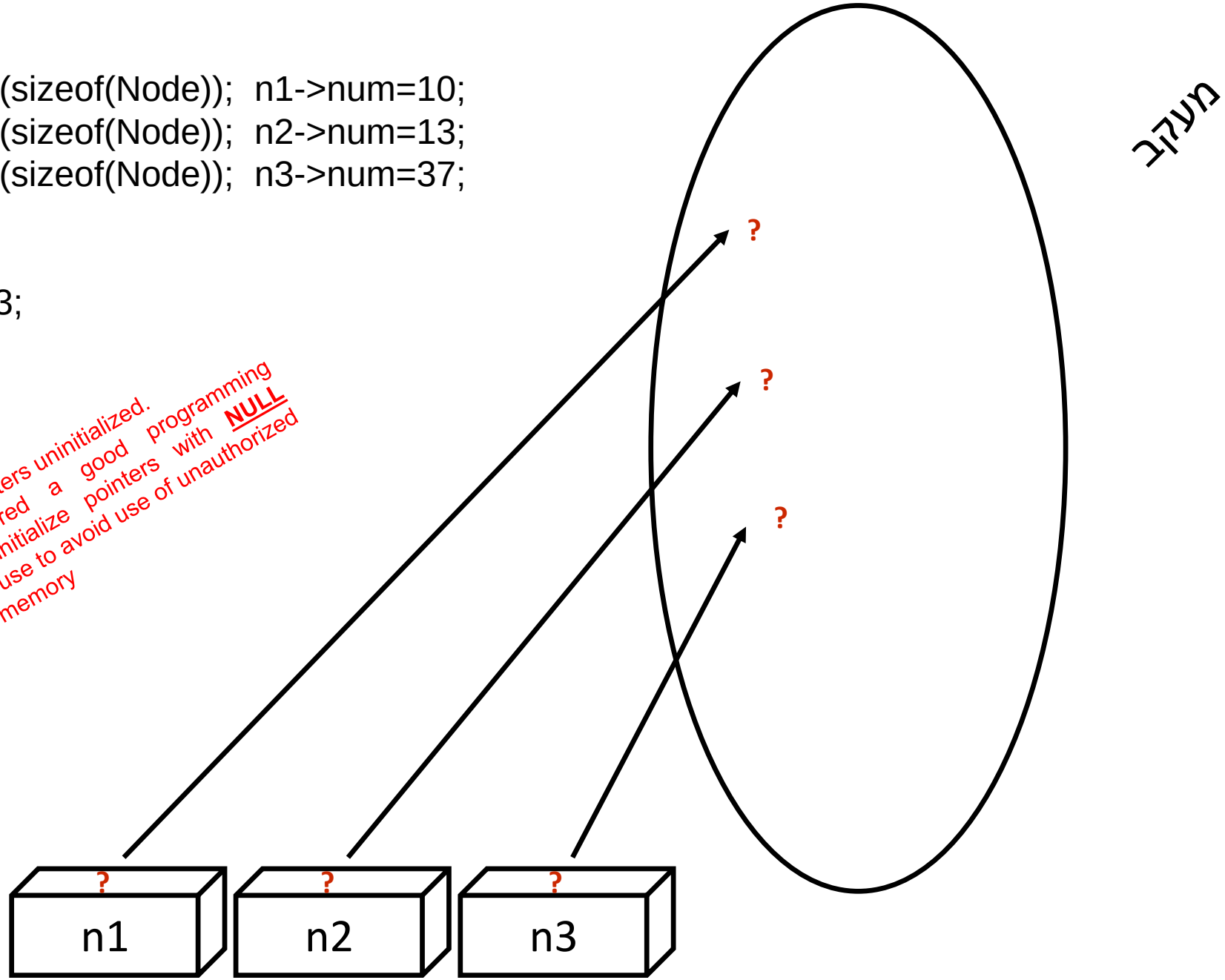
```c
#include <stdio.h>
#include <stdlib.h>
typedef struct Node
{
    int num;
    struct Node* next;
} Node;
int main()
{
    Node* n1=NULL,*n2=NULL,*n3=NULL;
    n1=(Node *) malloc(sizeof(Node));  n1->num=10;
    n2=(Node *) malloc(sizeof(Node));  n2->num=13;
    n3=(Node *) malloc(sizeof(Node));  n3->num=37;

    n1->next = n2;
    n1->next->next = n3;

    printf("%d %d %d\n",n1->num,n1->next->num, n1->next->next->num);
    printf("%d %d %d\n", (*n1).num,(*((*n1).next)).num, (*((*((*n1).next)).next)).num );

    free(n1);    free(n2);    free(n3);
    return 1;
}
```

יצרנו 3 אובייקטים מסוג Node
כדי לאתחל את הרשימה.
האם אנו צריכים את המזהים n1,n2,n3 ?
תשובה: אנו זקוקים רק למזהה אחד שמסמן את ראש הרשימה head נניח n1. הבא ונסתכל בעמוד הבא ←

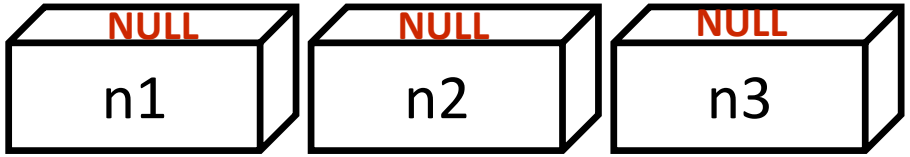
Output:
10 13 37
10 13 37

```
Node* n1,*n2,*n3;
n1=(Node *) malloc(sizeof(Node));  n1->num=10;
n2=(Node *) malloc(sizeof(Node));  n2->num=13;
n3=(Node *) malloc(sizeof(Node));  n3->num=37;

n1->next = n2;
n1->next->next = n3;
```

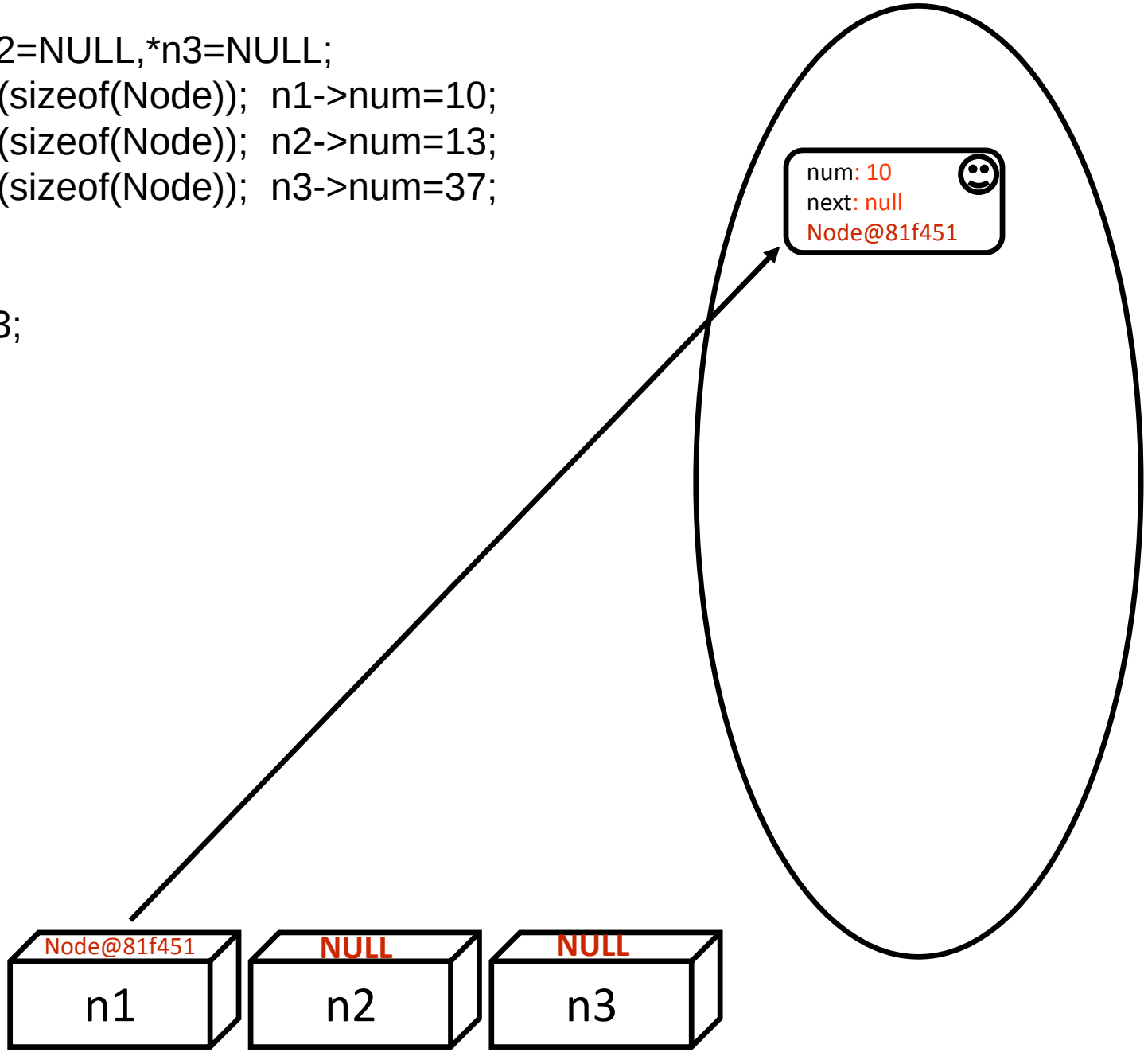
✓ Never keep pointers uninitialized.
✓ It is considered a good programming practice to initialize pointers with **NULL** before/after use to avoid use of unauthorized location in memory



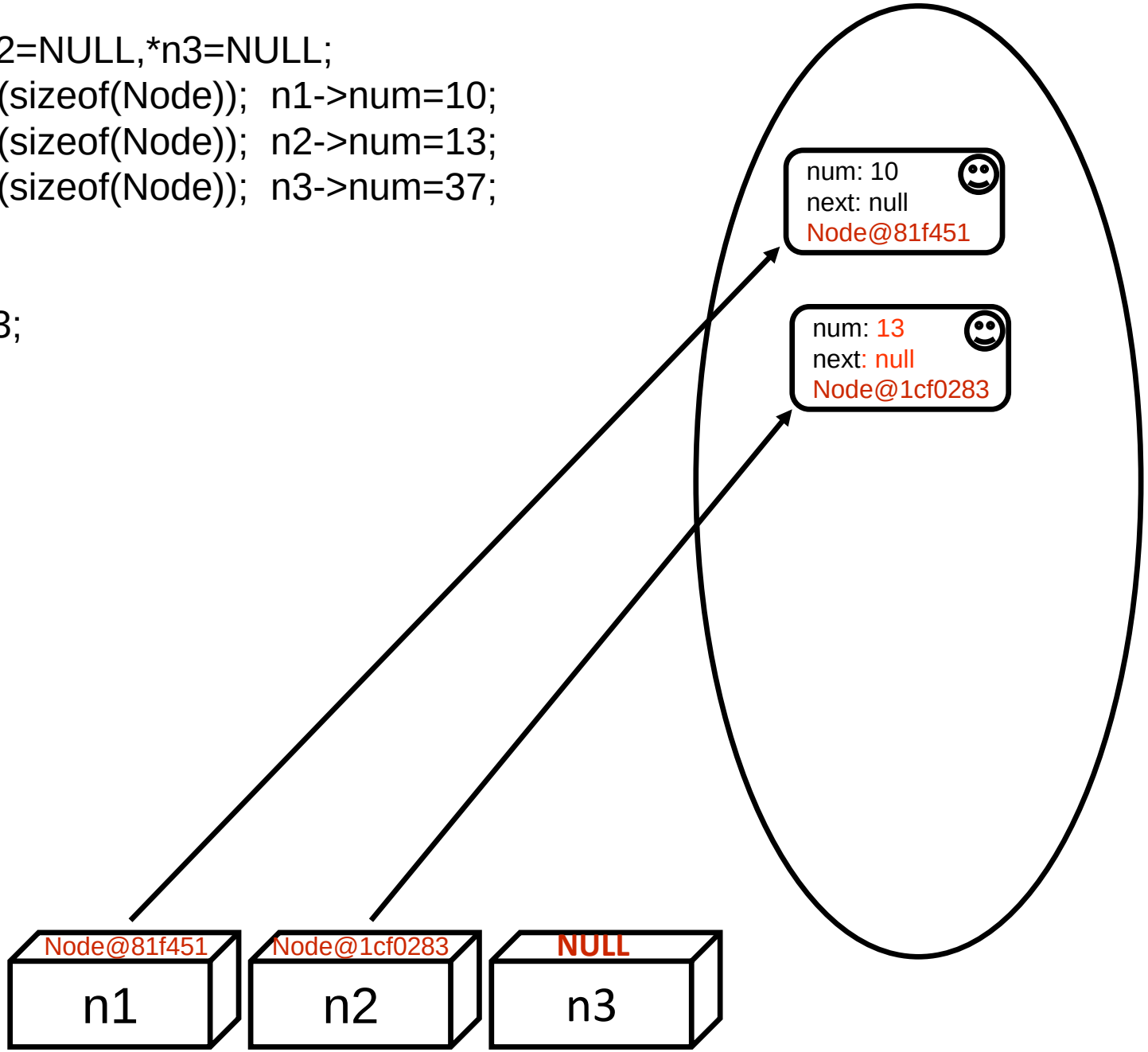
```
Node* n1=NULL,*n2=NULL,*n3=NULL;
n1=(Node *) malloc(sizeof(Node));  n1->num=10;
n2=(Node *) malloc(sizeof(Node));  n2->num=13;
n3=(Node *) malloc(sizeof(Node));  n3->num=37;

n1->next = n2;
n1->next->next = n3;
```



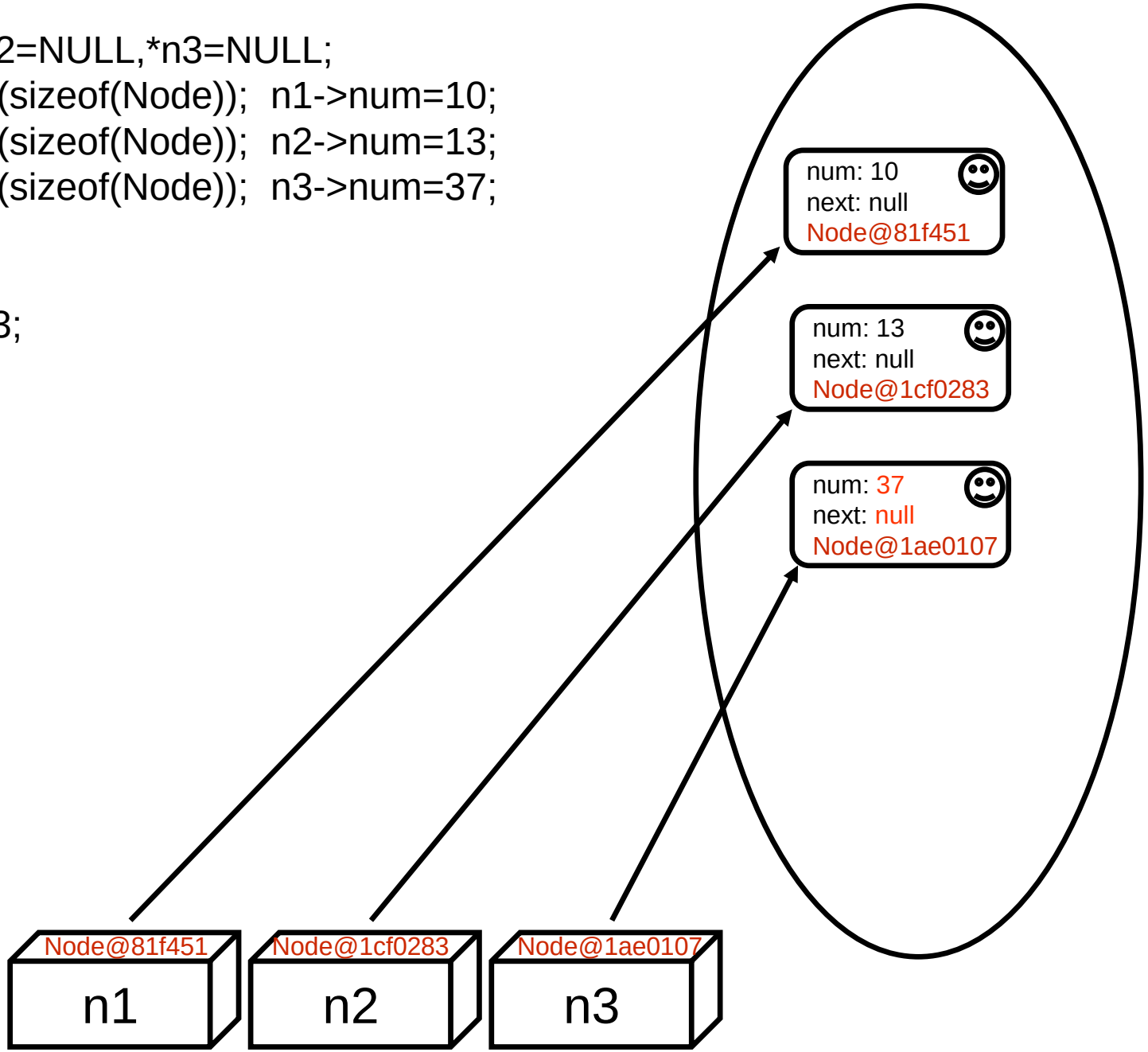
```
Node* n1=NULL,*n2=NULL,*n3=NULL;
n1=(Node *) malloc(sizeof(Node));  n1->num=10;
n2=(Node *) malloc(sizeof(Node));  n2->num=13;
n3=(Node *) malloc(sizeof(Node));  n3->num=37;

n1->next = n2;
n1->next->next = n3;
```



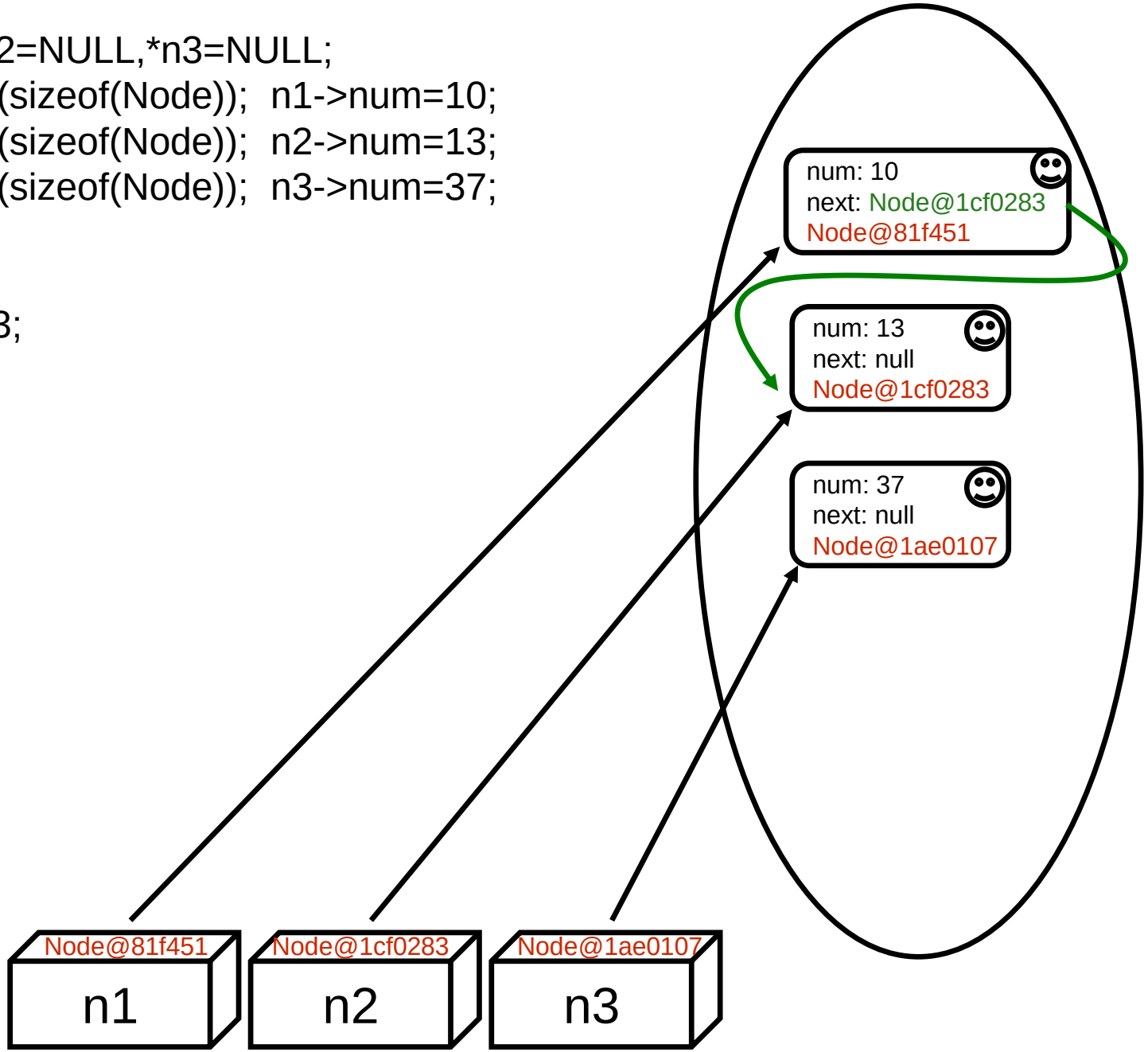
```
Node* n1=NULL,*n2=NULL,*n3=NULL;
n1=(Node *) malloc(sizeof(Node));  n1->num=10;
n2=(Node *) malloc(sizeof(Node));  n2->num=13;
n3=(Node *) malloc(sizeof(Node));  n3->num=37;

n1->next = n2;
n1->next->next = n3;
```



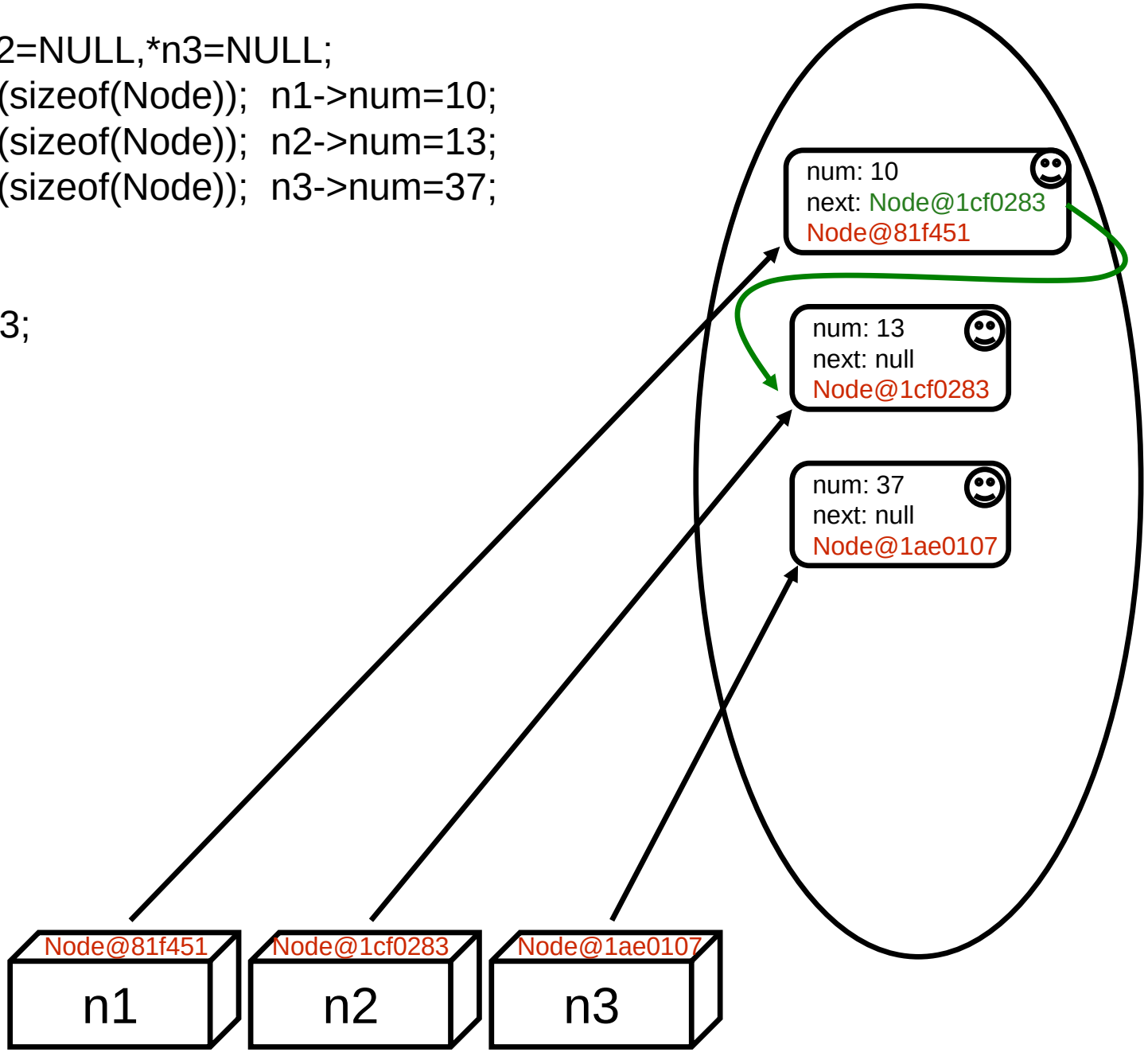
```
Node* n1=NULL,*n2=NULL,*n3=NULL;
n1=(Node *) malloc(sizeof(Node));  n1->num=10;
n2=(Node *) malloc(sizeof(Node));  n2->num=13;
n3=(Node *) malloc(sizeof(Node));  n3->num=37;

n1->next = n2;
n1->next->next = n3;
```



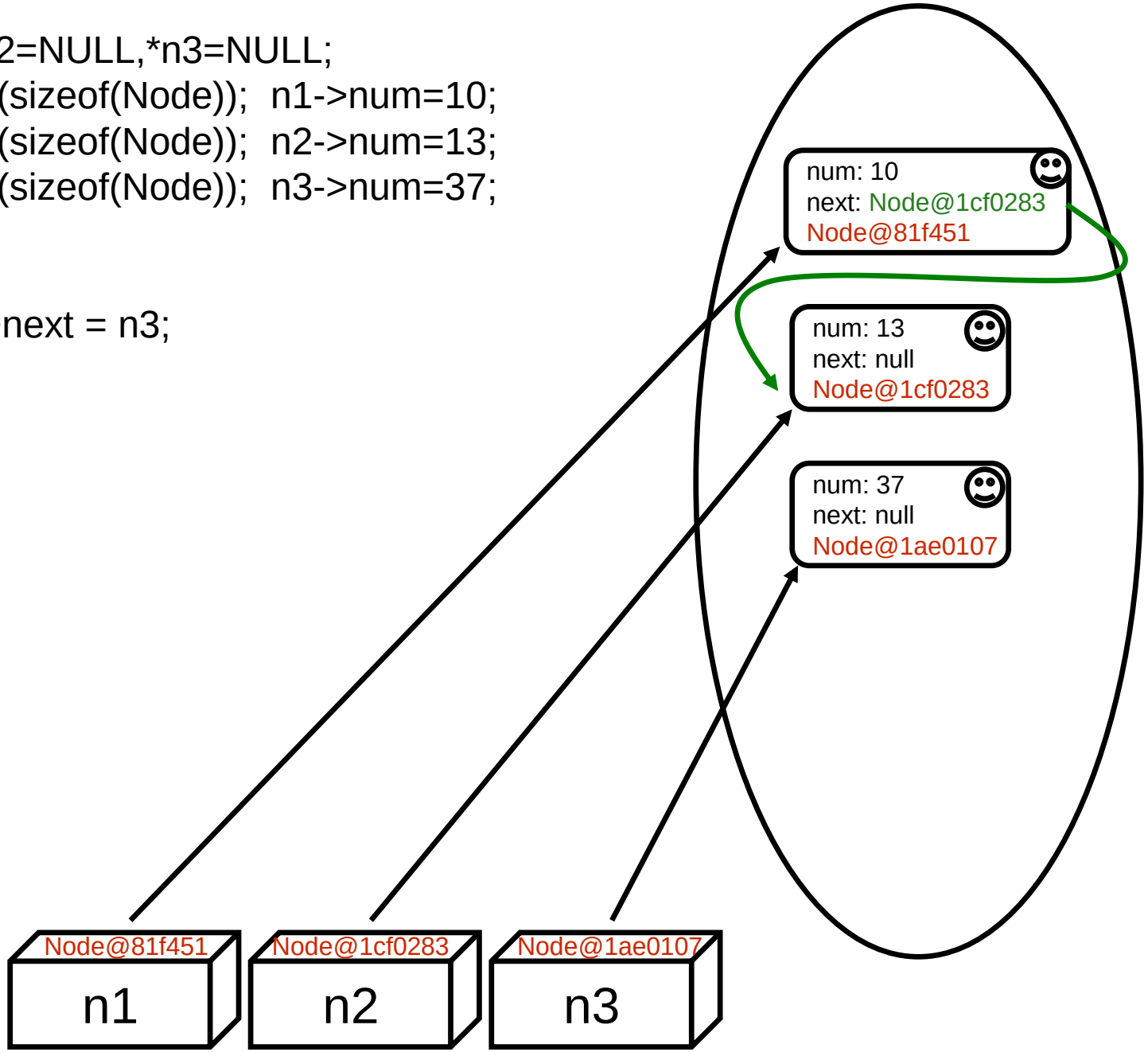
```
Node* n1=NULL,*n2=NULL,*n3=NULL;
n1=(Node *) malloc(sizeof(Node));  n1->num=10;
n2=(Node *) malloc(sizeof(Node));  n2->num=13;
n3=(Node *) malloc(sizeof(Node));  n3->num=37;

n1->next = n2;
n1->next->next = n3;
```



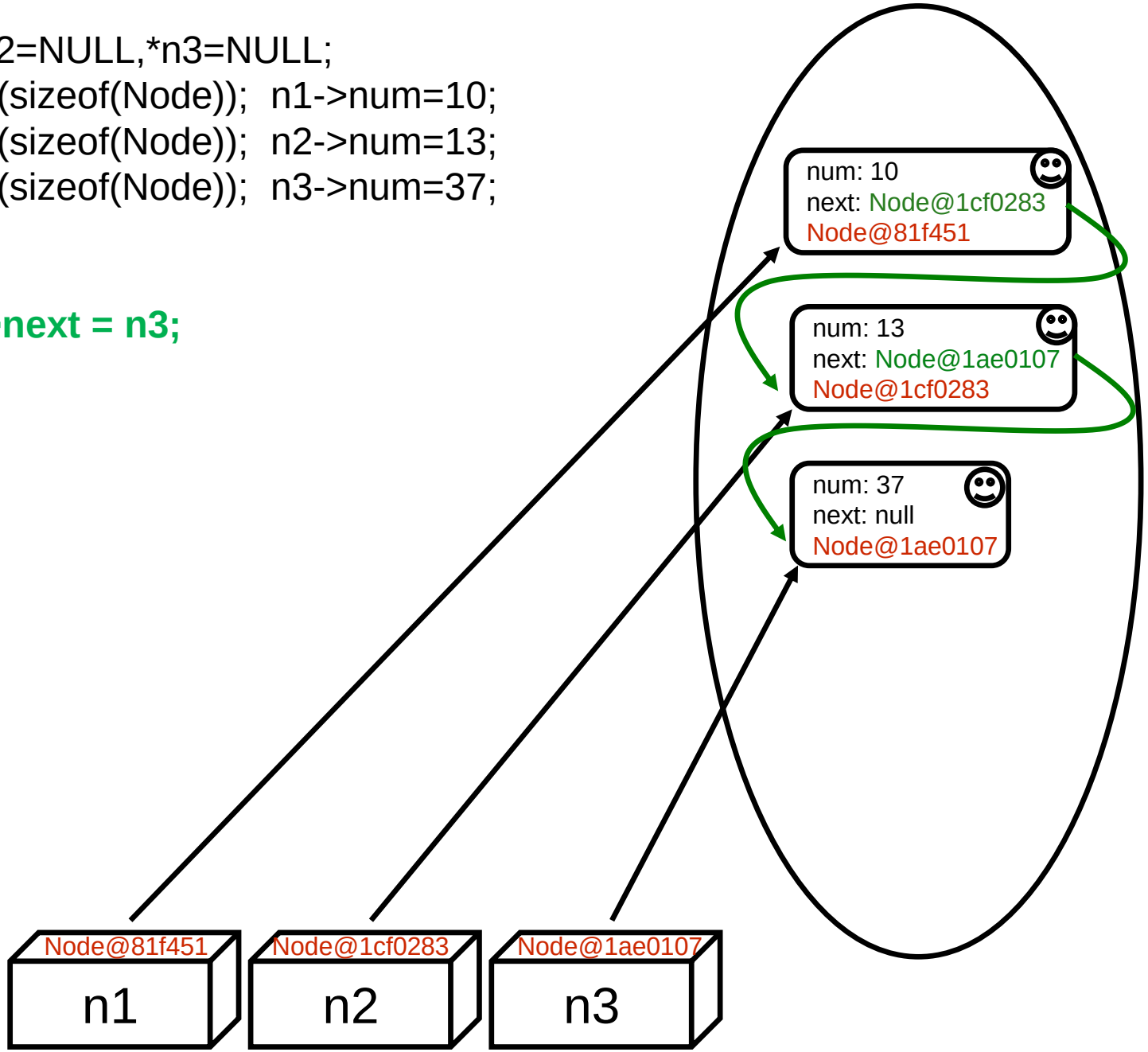
```
Node* n1=NULL,*n2=NULL,*n3=NULL;
n1=(Node *) malloc(sizeof(Node));  n1->num=10;
n2=(Node *) malloc(sizeof(Node));  n2->num=13;
n3=(Node *) malloc(sizeof(Node));  n3->num=37;

n1->next = n2;
n1->next->next = n3;
```



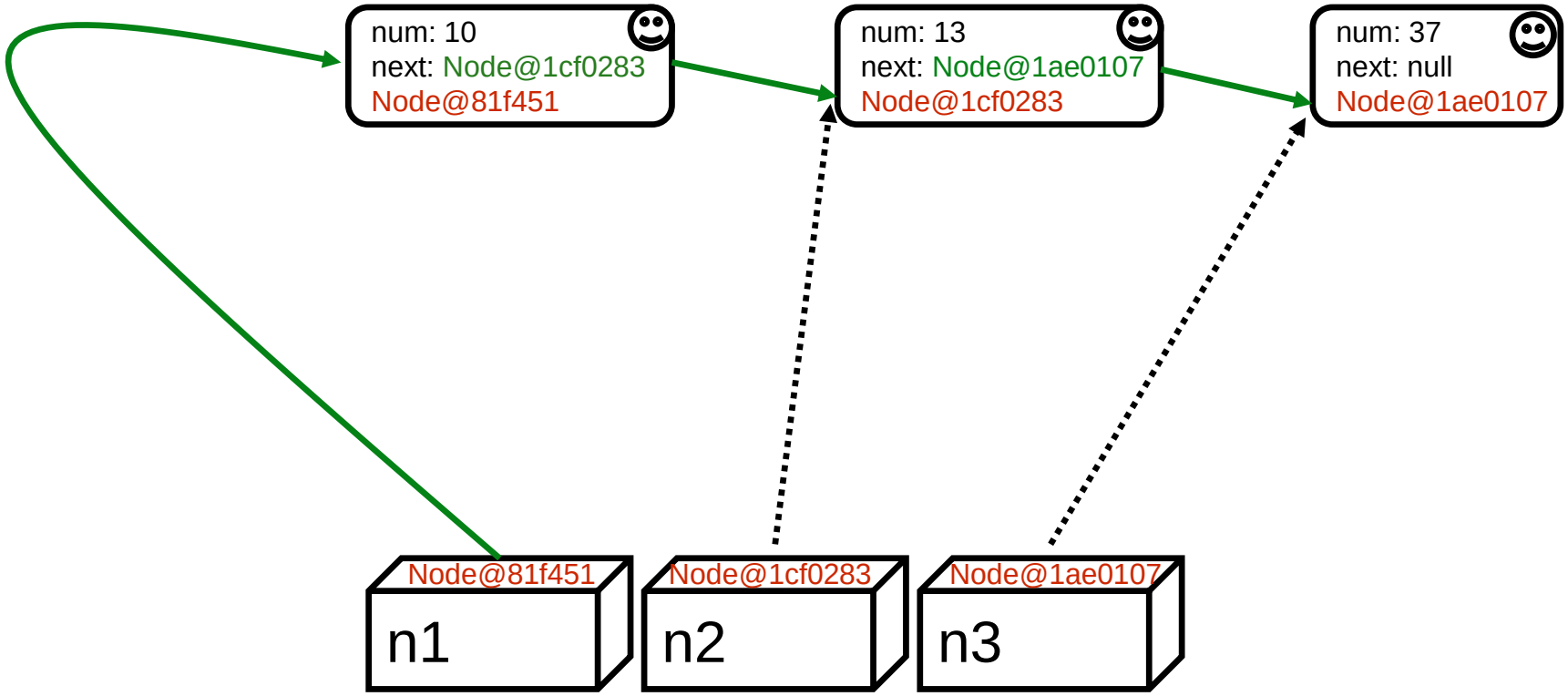
```
Node* n1=NULL,*n2=NULL,*n3=NULL;
n1=(Node *) malloc(sizeof(Node));  n1->num=10;
n2=(Node *) malloc(sizeof(Node));  n2->num=13;
n3=(Node *) malloc(sizeof(Node));  n3->num=37;

n1->next = n2;
(Node@1cf0283)->next = n3;
```



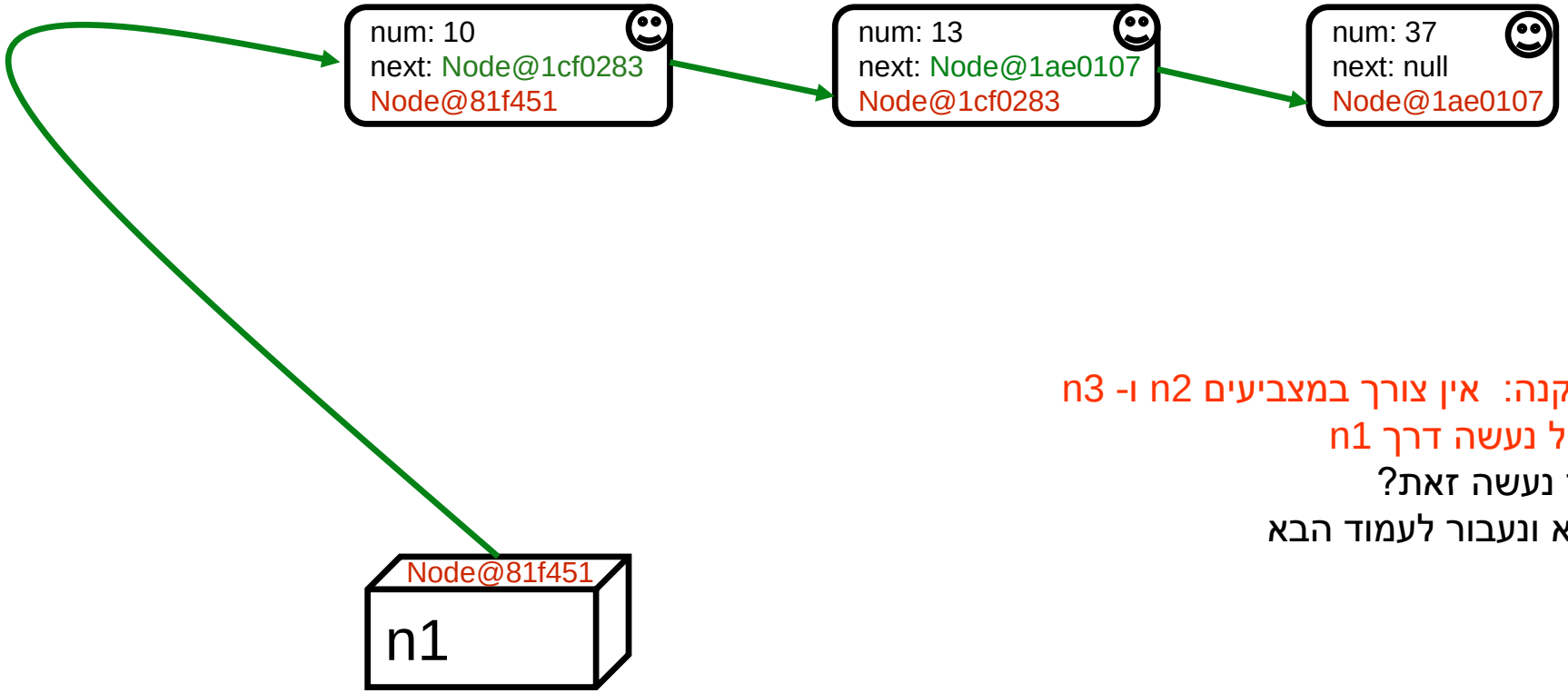
```
Node* n1=NULL,*n2=NULL,*n3=NULL;
n1=(Node *) malloc(sizeof(Node));  n1->num=10;
n2=(Node *) malloc(sizeof(Node));  n2->num=13;
n3=(Node *) malloc(sizeof(Node));  n3->num=37;

n1->next = n2;
(Node@1cf0283)->next = n3;
```



מה יקרה אם נוותר על שני המצביעים n2 ו- n3 ?
האם האובייקטים המקושרים אליהם נמחקים מהזיכרון?
האם בכלל יש צורך באותם מצביעים?



תשובה: האובייקט המוצבע ע"י n1 מצביע בעצמו על האובייקט המוצבע ע"י n2 וזה מצביע על האובייקט שמוצבע ע"י n3. לכן אם נוותר על n2 ו- n3 האובייקטים ימשיכו להתקיים



מסקנה: אין צורך במצביעים n2 ו- n3
הכול נעשה דרך n1
איך נעשה זאת?
הבא ונעבור לעמוד הבא

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node
{
    int num;
    struct Node* next;
} Node;
int main() {
    Node* head=NULL;
    head=(Node *) malloc(sizeof(Node));
    head->num=10;
    head->next=(Node *) malloc(sizeof(Node));
    head->next->num=13;
    head->next->next=(Node *) malloc(sizeof(Node));
    head->next->next->num=37;
    printf("%d %d %d\n",head->num,head->next->num, head->next->next->num);
    free(head->next->next);
    free(head->next);
    free(head);
    return 1;
}
```

שיחרור המקום שהוקצה בזיכרון בסדר הפוך

תרגיל : נתון המבנה Node אשר מייצגת קודקוד שמכיל מספר שלם ומצביע לקודקוד הבא.
כתוב את התוכנית LIST.c אשר מייצגת רשימה של מספרים המיוצגים ע"י קודקודים Node.
יש להוסיף את השיטות הבאות :

- length : מחזירה את אורך הרשימה
- insertToStart : הוספת איבר להתחלה
- insertToEnd : הוספת איבר לסוף
- Insert : הוספת איבר למקום מסויים
- removeFirst : הסרת איבר מהתחלה
- removeLast : הסרת איבר מהסוף
- removeFrom: הסרת איבר ממקום מסויים
- search: מחזירה את המקום של מספר כלשהו אם קיים
- equals:מקבלת שתי רשימות ובודקת אם הן שוות.
- newNode : יוצרת קודקוד חדש
- destroy : מוחקת רשימה שלימה
- print : מדפיסה את אברי הרשימה

main.c

```c
#include <stdio.h>
#include "LIST.h"
int main() {
    Node* head1=NULL,*head2=NULL;
    head1 = insertToStart(head1,1);
    head1 = insertToEnd(head1,3);
    head1 = insert(head1,2,2);
    printf("%d\n", search(head1,2));
    printf("%d\n", length(head1));
    print(head1);
    head2 = insertToEnd(head2,1);
    head2 = insertToEnd(head2,2);
    head2 = insertToEnd(head2,3);
    if(equals(head1,head2))
        printf("true\n");
    else
        printf("false\n");
    head1 = removeFrom(head1,2);
    head1 = removeLast(head1);
    head1 = removeFirst(head1);
    destroy(head1);
    destroy(head2);
    return 1;
}
```

Output:
2
3
1 2 3
true

LIST.h

```c
typedef struct Node {
    int num;
    struct Node* next;
} Node;

typedef enum Boolean {
    false, true
} Boolean;

Node* newNode(int x);
void destroy(Node* head);
void print(Node* head);
int search(Node* head, int num);
int length(Node* head);
Node* insertToStart(Node* head, int num);
Node* insertToEnd(Node* head, int num);
Node* insert(Node* head, int num, int pos);
Node* removeFirst(Node* head);
Node* removeLast(Node* head);
Node* removeFrom(Node* head, int pos);
Boolean equals(Node* head1, Node* head2);
```

LIST.c

version1

```
#include <stdio.h>
#include <stdlib.h>
#include "LIST.h"
Node* newNode(int x)
{
    Node *tmp=NULL;
    tmp = (Node *) malloc(sizeof(Node));
    if(tmp != NULL)
    {
        tmp->num=x;
        tmp->next = NULL;
    }
    return tmp;
}
:
:
```

LIST.c

version2

```
#include <stdio.h>
#include <stdlib.h>
#include "LIST.h"
Node* newNode(int x)
{
    Node *tmp=NULL;
    tmp = (Node *) malloc(sizeof(Node));
    if(tmp == NULL)
    {
      fprintf(stderr, "out of memory\n");
      exit(1);
    }
    tmp->num=x;
    tmp->next = NULL;
    return tmp;
}
:
:
```

version1

```
void destroy(Node* head)
{
    if(head == NULL)
        return;

    while(head != NULL)
    {
       Node* tmp = head;
       head = head->next;
       free(tmp);
    }
}
```

version2

```
void destroy(Node* head)
{
    if(head == NULL)
        return;

    destroy(head->next);

    free(head);
}
```

version1

```
int length(Node* head)
{
    int s=0;
    while(head != NULL)
    {
        s+=1;
        head=head->next;
    }
    return s;
}
```

version2

```
int length(Node* head)
{
    if(head==NULL)
        return 0;
    return length(head->next) + 1;
}
```

version1

```
void print(Node* head)
{
    while(head != NULL)
    {
        printf("%d ", head->num);
        head = head->next;
    }
    putchar('\n');
}
```

version2

```
void print(Node* head)
{
    if(head == NULL)
    {
        putchar('\n');
        return;
    }
    printf("%d ",head->num);
    print(head->next);
}
```

version1

```
int search(Node* head, int num)
{
    int pos=1;

    while(head != NULL)
    {
        if(head->num==num)
            return pos;
        head=head->next;
        pos++;
    }

    return -1;
}
```

version2

```
int search(Node* head,int num)
{
    return search2(head,num,1);
}
int search2(Node* head, int n, int pos)
{
    if(head == NULL)
        return -1;

    if(head->num == num)
        return pos;

    return search2(head->next, n, pos+1);
}
```

version1

```
Boolean equals(Node* head1, Node* head2)
{
    while(head1!=NULL && head2!=NULL)
    {
        if(head1->num != head2->num)
            return false;
        head1=head1->next;
        head2=head2->next;
    }

    if(head1==NULL && head2==NULL)
        return true;
    else
        return false;
}
```

version2

```
Boolean equals(Node* head1, Node* head2)
{
    if(head1==NULL && head2==NULL)
        return true;

    if(head1==NULL || head2==NULL)
        return false;

    if(head1->num != head2->num)
        return false;

    return equals(head1->next, head2->next);
}
```

Update head (if head was not created outside the function) from inside the function will not impact the head outside the function. Hence the value must be returned so, head outside the function will be updated.

head is declared outside 'insertToStart' and is being update from inside the function.

version1

```
Node* insertToStart(Node* head, int num)
{
    Node* tmp = newNode(num);
    if(tmp!=NULL)
    {
        tmp->next=head;
        head=tmp;
    }
    return head;
}
```

```
Node* head1=NULL;
head1 = insertToStart(head1,1);
```

version2

```
void insertToStart(Node** head, int num)
{
    Node* tmp = newNode(num);
    if(tmp!=NULL)
    {
        tmp->next=*head;
        *head=tmp;
    }
}
```

```
Node* head1=NULL;
insertToStart2(&head1,1);
```

version1

```
Node* insertToEnd(Node* head, int num)
{
    if(head==NULL)
        return newNode(num);
    else
    {
        Node* tmp = head;

        while(tmp->next!=NULL)
            tmp=tmp->next;

        tmp->next = newNode(num);
    }
    return head;
}
```

version2

```
Node* insertToEnd(Node* head, int num)
{
    if(head==NULL)
        return newNode(num);

    if(head->next==NULL)
    {
        head->next=newNode(num);
        return head;
    }

    insertToEnd(head->next,num);
    return head;
}
```

version1

```
Node* insert(Node* head, int num, int pos) {
   if(head == NULL || pos == 1)
      return insertToStart(head,num);
   else
   {
      Node* tmp=head;
      int i=1;
      while(tmp!=NULL && i<pos-1)
      {
         tmp=tmp->next;
         i++;
      }
      if(i==pos-1)
      {
         Node* tmp2=newNode(num);
         tmp2->next=tmp->next;
         tmp->next=tmp2;
      }
   }
   return head;
}
```

version2

```
Node* insert(Node* head, int num, int pos)
{
   if(head == NULL || pos==1)
      return insertToStart(head,num);

   if(pos-1==1)
   {
      Node* tmp = newNode(num);
      tmp->next = head->next;
      head->next = tmp;
      return head;
   }

   insert(head->next,num,pos-1);
   return head;
}
```

Update head from inside the function will not impact the head outside the function. Hence the value must be returned so, head outside the function will be updated.

head is updated directly from inside he function.

version1

```
Node* removeFirst(Node* head)
{
   if(head != NULL)
   {
      Node* tmp = head;
      head=head->next;
      free(tmp);
   }
   return head;
}
```

version2

```
void removeFirst(Node** head)
{
   if(*head != NULL)
   {
      Node* tmp = *head;
      *head=(*head)->next;
      free(tmp);
   }
}
```

version1

```
Node* removeLast(Node* head)
{
   if(head==NULL || head->next==NULL)
   {
      free(head);
      head=NULL;
      return head;
   }
   else
   {
      Node* tmp=head;
      while(tmp->next->next != NULL)
         tmp=tmp->next;
      free(tmp->next);
      tmp->next=NULL;
   }
   return head;
}
```

version2

```
Node* removeLast(Node* head)
{
   if(head==NULL || head->next==NULL)
   {
      free(head);
      head=NULL;
      return head;
   }
   if(head->next->next == NULL)
   {
      free(head->next);
      head->next=NULL;
      return head;
   }
   removeLast(head->next);
   return head;
}
```

LIST.c cont.

version1

```
Node* removeFrom(Node* head, int pos)
{
   if(pos==1)
      removeFirst(head);
   else
   {
      Node* tmp=head;
      int i=1;
      while(tmp!=NULL && i<pos-1)
      {
         tmp=tmp->next;
         i++;
      }
      if(i==pos-1)
      {
         Node* tmp2 = tmp->next;
         tmp->next = tmp->next->next;
         free(tmp2);
      }
   }
   return head;
}
```

version2

```
Node* removeFromr(Node* head, int pos)
{
   if(head==NULL)
      return NULL;

   if(pos==1)
      return removeFirst(head);

   if(pos-1==1 && head->next!=NULL)
   {
      Node* tmp = head->next;

      head->next = head->next->next;
      free(tmp);

      return head;
   }
   removeFromr(head->next,pos-1);
   return head;
}
```

END