first(): Return the position of the first element of S; an error occurs if S is empty.

last(): Return the position of the last element of S; an error occurs if S is empty.

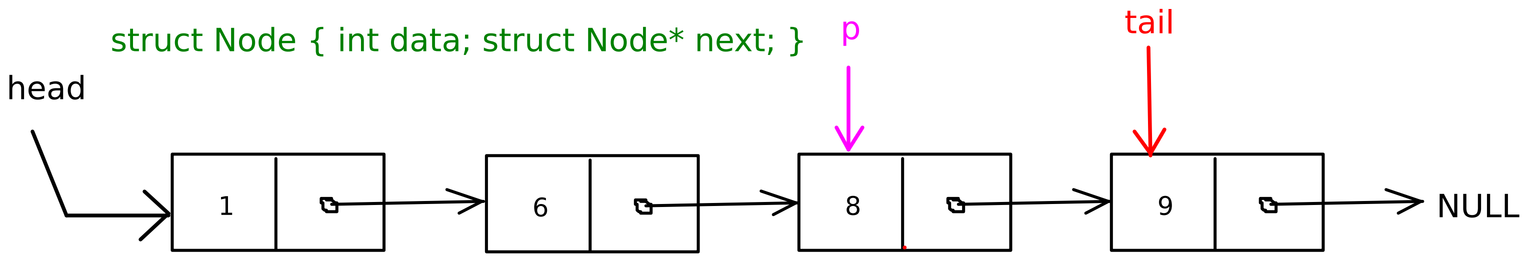
before(p): Return the position of the element of S preceding the one at position p; an error occurs if p is the first position.

after(p): Return the position of the element of S following the one at position p; an error occurs if p is the last position.

```
struct Node* before(p) {
   struct Node* currNode = head;
   if (p == NULL || p == head)
       return NULL;
   while (currNode->next != p)
       currNode = currNode->next;
   return currNode; }

struct Node* after(p) { return p->next; }
```

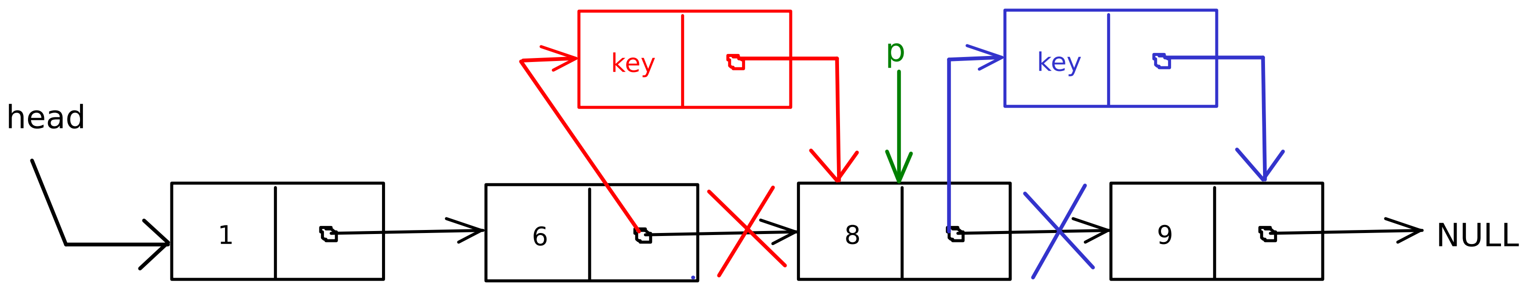
```
struct Node* first() { return head; }

struct Node* last() {
   struct Node* currNode;
   currNode = head;
   if (head == NULL)
      return NULL;
   while (currNode->next != NULL)
        currNode = currNode->next;
   return currNode;
}

struct Node* last() { return tail; }

int getDataAt(struct Node* p) {
     return p->data;
}
```

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



```
int getDataAtPos(int k) {
i = 1;
struct Node* currNode = head;
while( i < k  && currNode->next != NULL ) {
    currNode = currNode->next;  i = i + 1; }
if(i==k)     return currNode->data;
else         return NULL; }
```

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



```
insertBefore(struct Node* p, int key)
{
       struct Node* newNode;
       newNode = ....;  // allocate memory
       newNode->data = key;
       newNode->next = p;
       struct node* currNode = head;
       while(currNode->next != p)
             currNode = currNode->next;
       currNode->next = newNode;
}
```

```
insertAfter(struct Node* p, int key)
{
       struct Node* newNode;
       newNode = ....;  // allocate memory
       newNode->data = key;
       newNode->next = p->next;
       p->next = newNode;
}
```

first(): Return the position of the first element of S; an error occurs if S is empty.

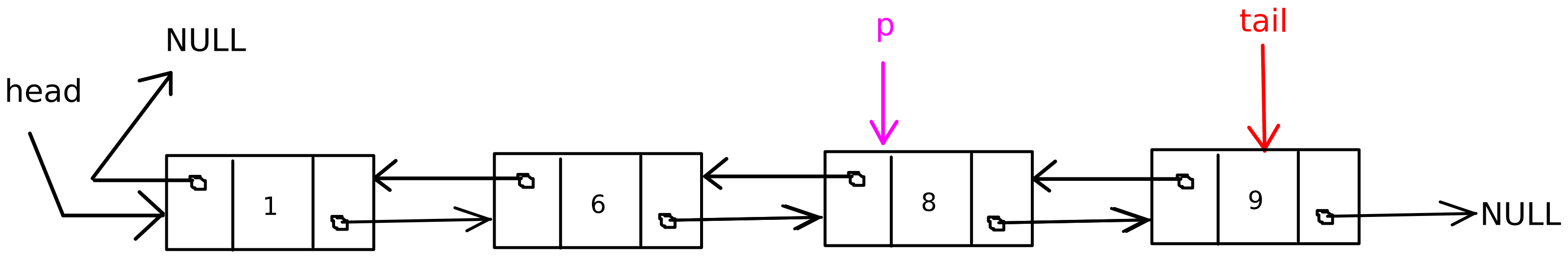
last(): Return the position of the last element of S; an error occurs if S is empty.

before(p): Return the position of the element of S preceding the one at position p; an error occurs if p is the first position.

after(p): Return the position of the element of S following the one at position p; an error occurs if p is the last position.

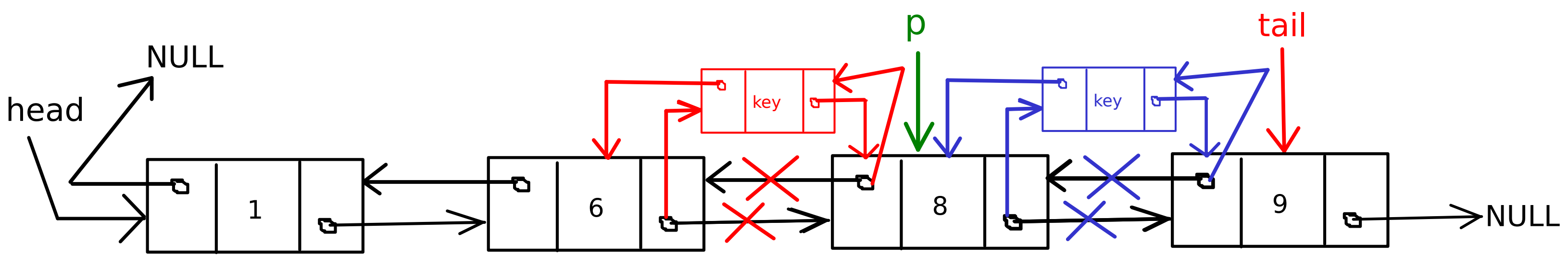
```
struct Node* first() { return head; }

struct Node* last() {
   struct Node* currNode;
   currNode = head;
   if (head == NULL)
      return NULL;
   while (currNode->next != NULL)
        currNode = currNode->next;
   return currNode;
}

struct Node* last() { return tail; }

int getDataAt(struct Node* p) {
     return p->data;
}
```

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

struct DLL {
        int size;
        struct Node* head;
        struct Node* tail;
}

struct Node* before(p) {  return p->prev;  }

struct Node* after(p) {  return p->next;  }
```



```
int getDataAtPos(int k) {
i = 1;
struct Node* currNode = head;
while( i < k  && currNode->next != NULL ) {
    currNode = currNode->next;  i = i + 1; }
if(i==k)     return currNode->data;
else         return NULL; }
```

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



```
insertBefore(struct Node* p, int key)
{
       struct Node* newNode;
       newNode = ....;  // allocate memory
       newNode->data = key;
       newNode->prev = p->prev;
       newNode->next = p;
       p->prev->next = newNode;
       p->prev = newNode;
}
```

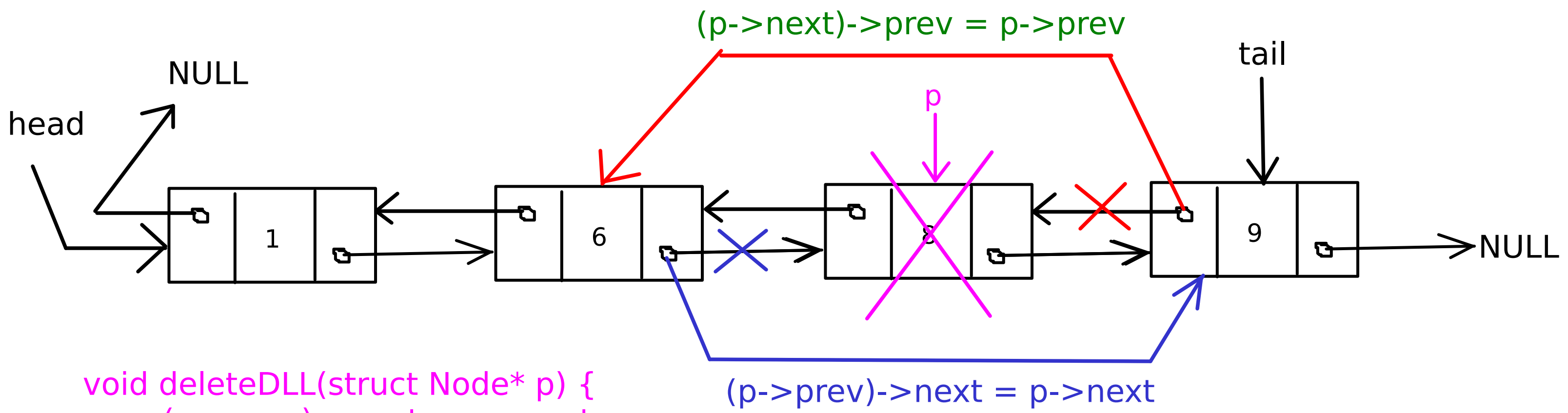
```
insertAfter(struct Node* p, int key)
{
       struct Node* newNode;
       newNode = ...;  // allocate memory
       newNode->data = key;
       newNode->prev = p;
       newNode->next = p->next;
       p->next->prev = newNode;
       p->next = newNode;
}
```

```
void insertAtBeginningSLL(head, key) {
        struct Node* newNode;
        newNode = ...;  // allocate memory
        newNode->data = key;
        newNode->next = head;
        head = newNode;
}
```

```
void insertAtBeginningDLL(head, key) {
        struct Node* newNode;
        newNode = ...;  // allocate memory
        newNode->data = key;
        newNode->prev = NULL;
        newNode->next = head;
        if( head != NULL)
                head->prev = newNode;
        head = newNode;
}
```

```
void insertAtEndSLL(head, key) {
        struct Node* newNode;
        newNode = ...;  // allocate memory
        newNode->data = key;
        newNode->next = NULL;
        if( head == NULL ) {
                head = newNode;
                // tail = newNode;
        }
        else {
                struct Node* currNode = head;
                while( currNode->next != NULL)
                        currNode = currNode->next;
                currNode->next = newNode;
                // tail->next = newNode;
                // tail = newNode;
        }
}
```

```
void insertAtEndDLL(head, key) {
        struct Node* newNode;
        newNode = ...;  // allocate memory
        newNode->data = key;
        newNode->next = NULL;
        if( head == NULL ) {
                head = newNode;
                // tail = newNode;
        }
        else {
                struct Node* currNode = head;
                while( currNode->next != NULL)
                        currNode = currNode->next;
                currNode->next = newNode;
                // tail->next = newNode;
                // newNode->prev = tail;
                // tail = newNode;
        }
}
```

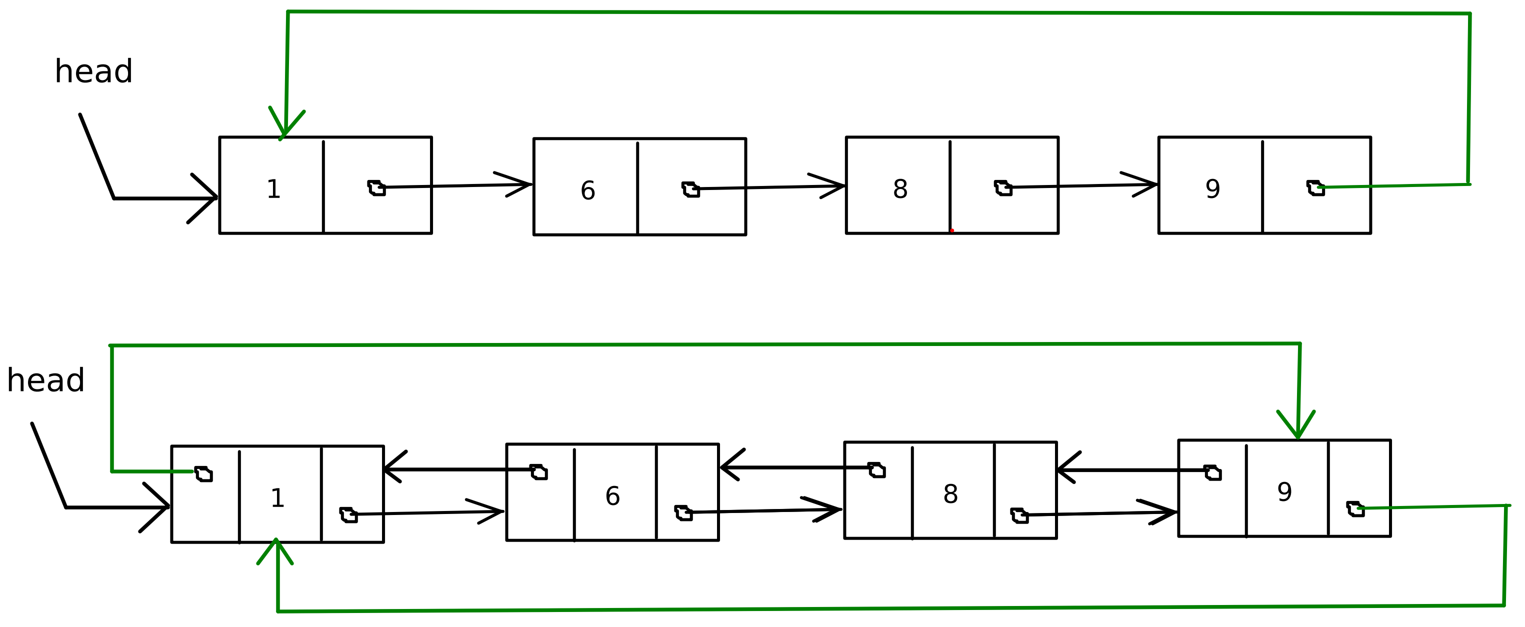


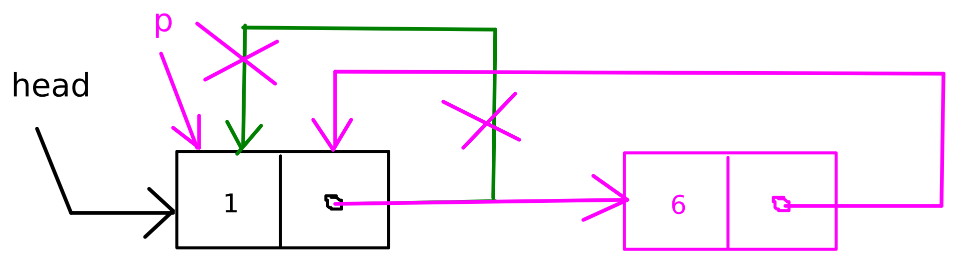
```
void deleteDLL(struct Node* p) {
        (p->prev)->next = p->next;
        (p->next)->prev = p->prev;
        free(p);  // de-allocate Node p
}
```

```
void deleteSLL(struct Node* p) {
        struct Node* currNode = head;
        while(currNode->next != p)
                currNode = currNode->next;
        currNode->next = p->next;
        free(p);  // de-allocate Node p
}
```

```
void swapNodes(Node* p, Node* q) {
        (p->prev)->next = q;
        (p->next)->prev = q;
        (q->prev)->next = p;
        (q->next)->prev = p;
        struct Node* tmp;
        tmp = p->next;
        p->next = q->next;
        q->next = tmp;
        tmp = p->prev;
        p->prev = q->prev;
        q->prev = tmp;
}
```

Circular Lists (both Singly-linked and Doubly-linked)





```
newNode->next = p->next
p->next = newNode;
```