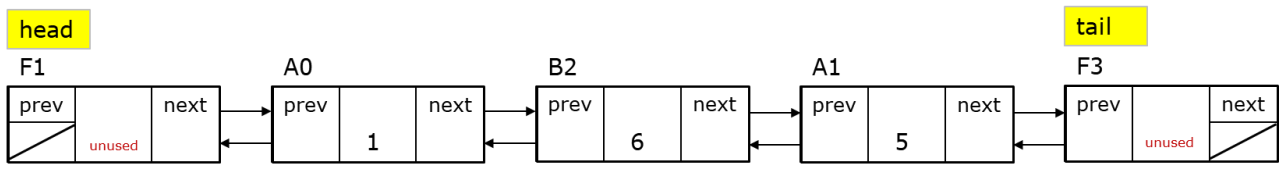
The following materials have been collected from the numerous sources including my own and my students over the years of teaching and experiences of programming. Please help me to keep this tutorial up-to-date by reporting any issues or questions. Please send any comments or criticisms to idebtor@gmail.com. Your assistances and comments will be appreciated.

A doubly linked list with sentinel nodes(ver 2.)

Table of Contents

Warming-up

This problem set consists of implementing a doubly-linked list with two sentinel nodes shown below as an example:

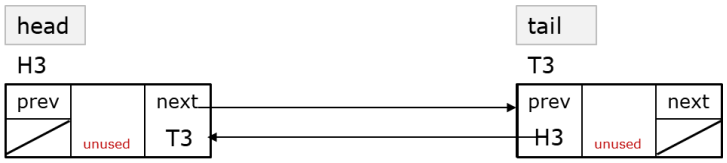


These extra nodes are sometimes known as **sentinel nodes**, specifically, the node at the front is known as **head** node, and the node at the end is known as a **tail** node.

When the doubly linked list is initialized, the head and tail nodes are created. The purpose of these nodes is to simply the insert, push/pop front and back, remove methods by eliminating all need for special-case code when the list empty, or when we insert at the head or tail of the list. **This would greatly simplify the coding unbelievably.**

For instance, if we do not use a head node, then removing the first node becomes a special case, because we must reset the list's link to the first node during the remove and because the remove algorithm in general needs to access the node prior to the node being removed (and without a head node, the first node does not have a node prior to it).

An empty may be constructed with the head and tail nodes only as shown in the following figure.



An **empty** doubly linked list with sentinel nodes

The following two data structures, **Node** and **List**, can be used to hold a doubly-linked nodes as well as two sentinel nodes

```
struct Node {
  int     item;
  Node*   prev;
  Node*   next;
  Node(const int d = 0, Node* p = nullptr, Node* n = nullptr) {
    item = d; prev = p; next = n;
  }
  ~Node() {}
};

struct List {
  Node* head;
  Node* tail;
  List() { head = new Node;      tail = new Node;
           head->next = tail;    tail->prev = head;
           head->prev = nullptr; tail->next = nullptr;
  }
  ~List() {}
};

using pNode = Node*;
using pList = List*;
```

key functions: begin() and end()

The function **begin()** returns the first node that the head node points.

```
// Returns the first node which List::head points to in the container.
pNode begin(pList p) {
  return p->head->next;
```

```
}
```

The function **end()** returns the tail node referring to the past -the last- node in the list. The past -the last- node is the sentinel node **which is used only as a sentinel** that would follow the last node. It does not point to any node next, and thus shall not be dereferenced. Because the way we are going use during the iteration, we don't want to include the node pointed by this. this function is often used in combination with **List::begin** to specify a range including all the nodes in the list. This is a kind of simulated used in STL. If the container is empty, this function returns the same as **List::begin**.

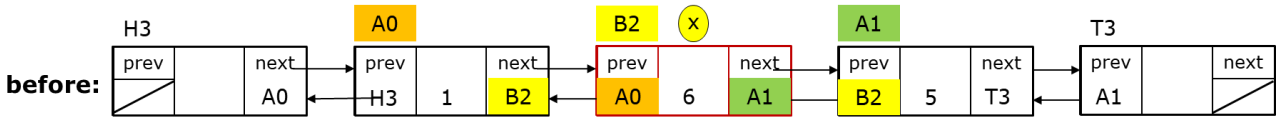
```
pNode end(pList p) {
  return p->tail;          // not tail->next
}
```

key functions: erase() and insert()

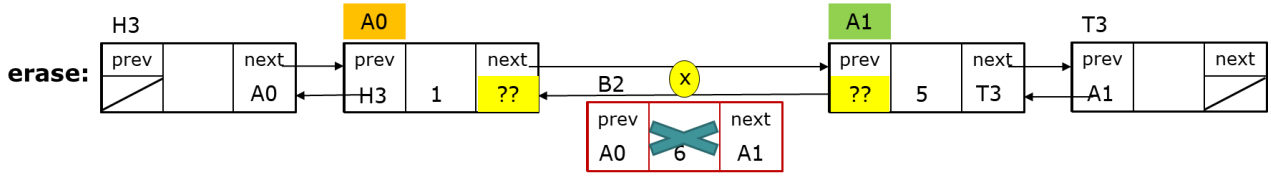
The function **erase()** removes from the list a single node x given. This effectively reduces the container by one which is destroyed. It is specifically designed to be efficient inserting and removing a node regardless of its positions in the list such as front, back or in the middle of the list.

Let us suppose that we want to remove the node x. The erase() function has only one argument x as shown below:

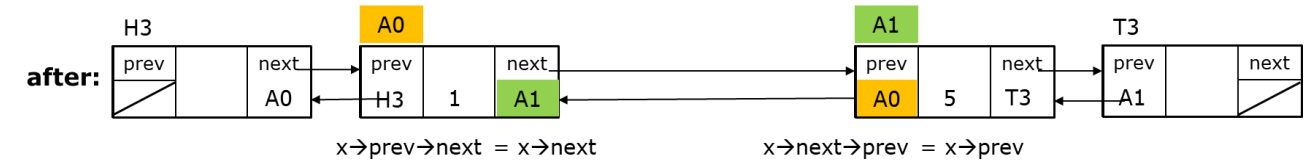
```
void erase(pNode x);
```



After removal of node x, the list looks like this.



Since node x or B2 is removed, now, we must link **node A0** and **node A1**. This means that **A0→next** must set to **A1** and **A1→prev** must set to **A0.**




```
void erase(pNode x) {
  x->prev->next = x->next;
  x->next->prev = x->prev;
  delete x;
}
```

We may extend this erase() function with an extra argument list p such that it can handle some erroneous cases as shown below:

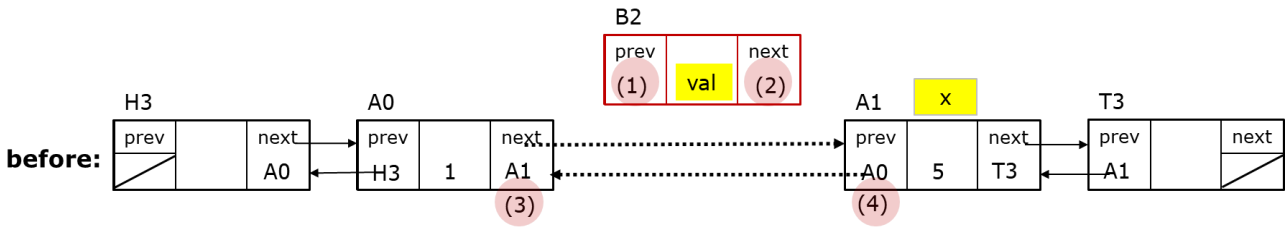
```
  void erase(pList p, pNode x) {// checks if x is neither tail nor head
    if (x == p->tail || x == p->head || x == nullptr) return;
    x->prev->next = x->next;
    x->next->prev = x->prev;
    delete x;
  }
```

The function **insert()** extends the container inserting a new node with **val before** the node at the specified position **x.** This effectively increases the list size by one. For example, if **begin(p)** is specified as an insertion position, the new node becomes the first one in the list.

Let us suppose that we want to insert the node **x** with **val**. The insert() function has two arguments as shown below:

```
void insert(pNode x, int val)
```

As it is shown the figure below, we want to create a new node(B2) and insert it at node x (A1). In other words, insert the new node B2 between node A0 and node A1.



Now we must set the values in four places mnemonically.

(1) B2→prev

(2) B2→next

(3) A0→next or A1

(4) A1→prev or A0

These can be expressed in real code

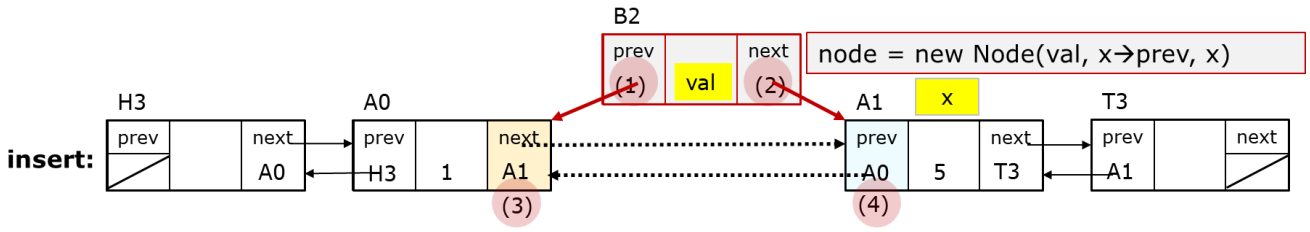
(1) node→prev

(2) node→next

(3) x or x→prev→next
(4) x→prev

For (1) and (2) links, we can set the two links in the new node while initiating the node with its initialization as shown in the following figure.

```
pNode node = new Node(val, x->prev, x);
```

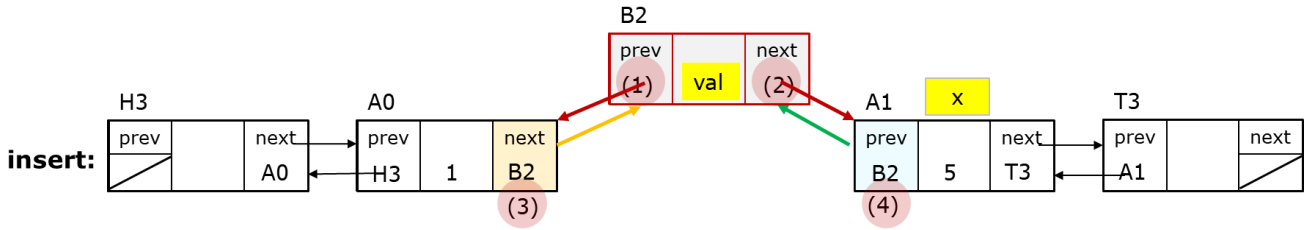


For the link (3), it is to set A0→next to the new node:

```
    x->prev->next = node;
```

For the link (4), it is to set A1→prev to the new node:

```
   x->prev = node;
```



Since we completed the linking of the new node, we just put them together in a function.

```
void insert(pNode x, int val) {
  pNode node = new Node(val, x->prev,  x);
  x->prev = x->prev->next = node;
}
```

key functions: find(), _more() and _less()

The function find() returns the node of which item is the same as x firstly encountered in the list and **tail** if not found.

```
// returns the first node with a value found, nullptr otherwise.
pNode find(pList p, int val) {
  for (pNode c = begin(p); c != end(p); c = c->next)
    if (c->item == val) return c;
  return c;
}
```

The function **_more()** returns the node of which item is greater than x firstly encountered in the list and **tail** if not found.

```
pNode _more(pList p, int x)
  for (pNode c = begin(p); c != end(p); c = c->next)
     if (c->item > x) return c;
  return c;
```

The function **_less()** returns the node of which item is smaller than x firstly encountered in the list and **tail** if not found.

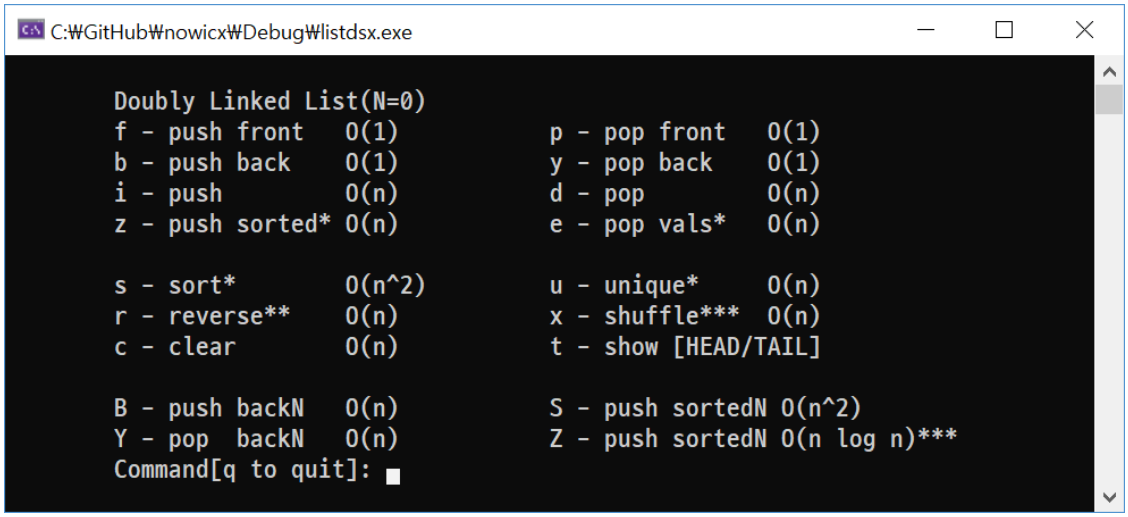
```
pNode _less(pList p, int x)
  for (pNode c = begin(p); c != end(p); c = c->next)
    if (c->item < x) return c;
  return c;
}
```

Getting started

Your job is to complete the given program, **listds.cpp**. The following files are provided for your references.

1. listds.h – Don't change this file.
2. listdsDriver.cpp – Don't change this file.
3. listds.cpp – A skeleton code provided, your code goes here.
4. listsort.cpp - A skeleton code provided, your code goes here.
5. quicksort.cpp - Don't change this file.
6. listdsx.exe – It is provided for your reference;
7. 04-3linkedlist.pdf, 04-4linkedlist.pdf – a good reference for your work provided through github as you know

Sample run: listdsx.exe



Step 1: push(), pop() and pop_all()*

The function **push()** inserts a new node with val at the position of the node with x. The new node is actually inserted in front of the node with x. It returns the first node of the list. This effectively increases the container size by one. (Hint: Consider to use find() and insert().)

```
void push(pList p, int val, int x)
```

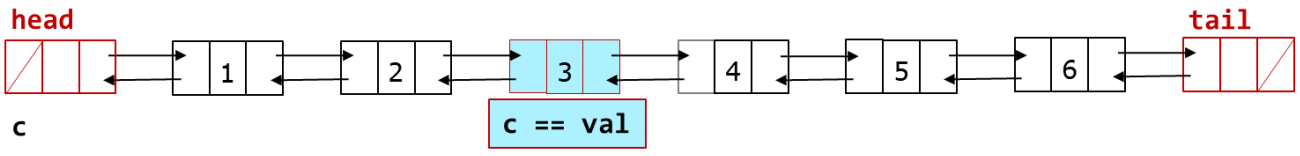
The function **pop()** removes the first node with val from the list and does nothing if not found. Unlike member function **List::erase** which erases a node by its position, this function removes a node by its value. Unlike **pop()**, **pop_all()** removes all the nodes with the value given. (Hint: Consider to use find() and erase().)

```
void pop(pList p, int val);
```

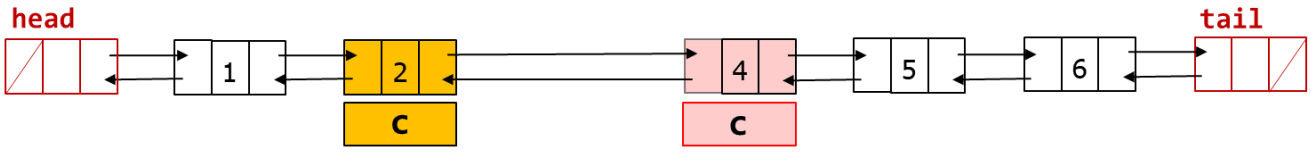
Pop_all() removes from the list all the nodes with the same value given. **This should be done in O(n) and go through the list once**, not multiple times. tructor of these objects and reduces the list size by the number of nodes removed. Unlike **erase()**, which erases a node by its position, this function removes nodes by its value. Unlike **pop_all()**, **pop()** removes the first occurrence of the node with the value given.

A skeleton of **pop_all()** with a bug is provided.

```
// remove all occurrences of nodes with val given in the list.
void pop_all(pList p, int val) {
  for (pNode c = begin(p); c != end(p); c = c->next)
    if (c->item == val)
      erase(c);
} // with bugs
```

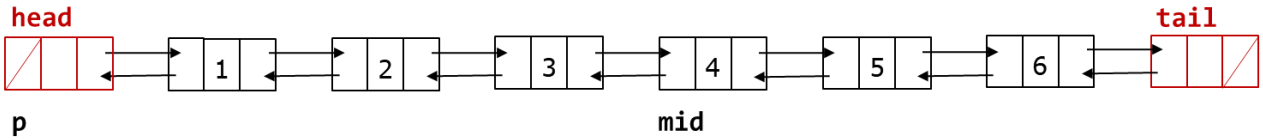


For example, this code removes 3 successfully. You may observe some failures when it has some consecutive occurrences of 3's in the list. Also add a print statement right after **erase()** and see what it prints after the removal of "3". To fix the bug, ask yourself a question about "Where should c be pointing right after **erase(c)**, 2 or 4 in for-loop?"

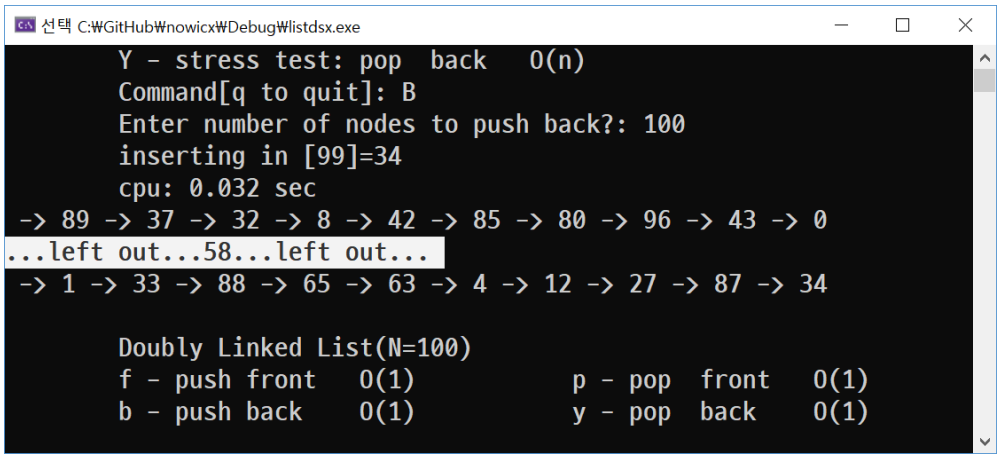


Step 2: half()

This function returns a node at a midpoint of the list. Even number of nodes, it returns the first node of the second half which is 6th node if there are ten nodes. If there are five nodes in the list, it returns the third one.



This function is used in **show()** and **shuffle().** You may see its functionality in action when you set **"show [HEAD/TAIL]"** option and display a list in the main menu. The **show()** displays **"58"** which is at the midpoint among 100 nodes as shown below;



Method 1: Count how many nodes there are in the list, then scan to the halfway point, breaking the last link followed.

```
pNode half(pList p) {
  int N = size(p);
  // go through the list
  // break at the halfway point
  // return the current pointer
}
```

Method 2: It works by sending rabbit and turtle down the list: turtle moving at speed one, and rabbit moving at speed two. As soon as the rabbit hits the end, you know that the turtle is at the halfway point as long as the rabbit gets asleep at the halfway.

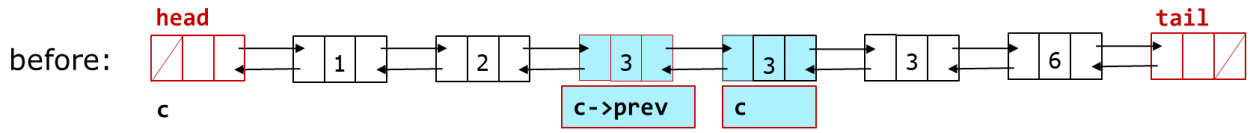
```
pNode half(pList p) {
  pNode rabbit = begin(p);
  pNode turtle = begin(p);
  while (rabbit != end(p)) {
    rabbit = rabbit->next->next;
    turtle = turtle->next;
```

```
  }
  return turtle;
} // buggy on purpose
```

Step 3: unique()*

This function removes extra nodes from the list that has duplicate values It removes all but the **first** node from every **consecutive group** of equal nodes in the list. Notice that a node is only removed from the list if it compares equal to the node immediately **preceding** it. Thus, this function especially works for sorted lists in either ascending or descending. It should be done in O(n).

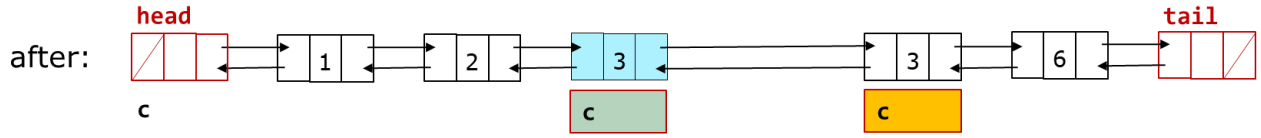
For example, the second and third occurrence of 3 must be removed in the following list.



A skeleton code is provided with some bugs

```
// removes extra nodes that have duplicate values from the list.
void unique(pList p) {
  if (size(p) <= 1) return;
  for (pNode c = begin(p); c != end(p); c = c->next)
    if (c->item == c->prev->item)
      erase(c);
} // with bugs
```

The following figure shows right after the first erase(c) or the first removal of 3. To debug the skeleton code, answer "Where should c be pointing **right after the first erase(c)**?"



Step 4-1: selectionSort()

There are many ways of doubly-linked list sorting. As example, bubbleSort() and quicksort() for doubly-linked list are provided in listsort.cpp You are supposed to implement selectioSort() in listsort.cpp.

```
void bubbleSort(pList p, int(*comp)(int, int)) {
  // if (sorted(p)) { reverse(p); return; }
  pNode tail = end(p);
```

```
  pNode curr;
  for (pNode i = begin(p); i != end(p); i = i->next) {
    for (curr = begin(p); curr->next != tail; curr = curr->next) {
      if (comp(curr->item, curr->next->item) > 0)
    swap(curr->item, curr->next->item);
    }
    tail = curr;
  }
}
```

Step 4-2: sorted()

There are many things to do in sort related functions. Thus, let us begin with implementing **sorted()** functions first.

There are two functions of **sorted()** overloaded. One invokes the other one and is already implemented as shown below. You must implement the second one which does actual comparison between nodes. It compares each node to its following node and determine whether it is sorted or not. Use the function pointer passed for comparing.

```
// returns true if the list is sorted either ascending or descending.
bool sorted(pList p) {
  return sorted(p, ascending) || sorted(p, descending);
}

// returns true if the list is sorted according to comp() function.
// com() function may be either ascending or descending.
bool sorted(pList p, int(*comp)(int a, int b)) {
  if (size(p) <= 1) return true;

  cout << "your code here\n";

  return true;
}

int ascending (int a, int b) { return a - b; };
int descending(int a, int b) { return b - a; };
```

Once you implement this function, you may test it using the menu option "s" temporarily since there is no option available for this functionality.

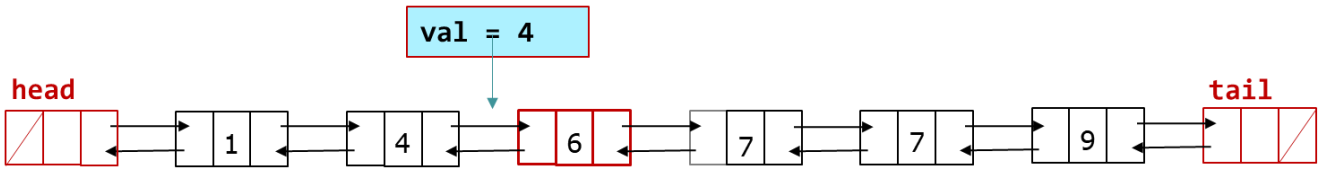
Step 4-3: push_sorted() and push_sortN()

The function **push_sorted()** used in option z inserts a new node with val in sorted order to the list in either ascending or descending order.

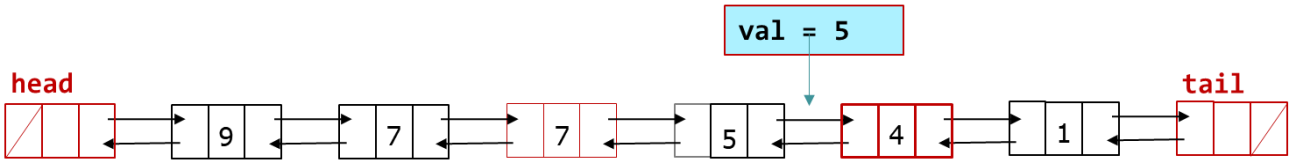
```
void push_sorted(pList p, int val)
```

If the list is sorted in ascending, you may invoke **insert()** with the position node x of which the value is greater than **val**. Use **_more()** to find the node x which is greater than val.

To insert **val = 4**, for example, you must find the position node x with **val = 6** using **_more()** function.,



Take the similar steps for descending ordered list. To insert **val = 5**, for example, you must find the position node x with **val = 4** using **_less()** function.,



Hint: You may need the following functions to implement this part:
sorted(), insert(), _more(), _less(), ascending(), descending()

```
// inserts a new node with val in sorted order
void push_sorted(pList p, int val) {
  if sorted(p, "ascending order")
     insert("find a node _more() than val", val);
  else
     insert("find a node _less() than val", val);
}
```

The function **push_sortN()** inserts N number of nodes in sorted in the sorted list. **Don't invoke push_sort() by N times** since it is expensive computationally. But if you may do something like push_sort(), its time complexity will be **O(n^2)** or larger. The values for new nodes are randomly generated in the range of **[0..(N + size(p))].**

For mac, use **rand()** as usual.

For pc, use **(rand() * RAND_MAX + rand())** instead of rand().

```
void push_sortedN(pList p, int N)
```

Hint: Use some function used in push_sort(). Invoke sorted() once and use the result N times in the loop. Don't use push_sort(), but use **sorted(), insert(), _more(), _less().** But it is still O(n^2) time complexity.

Optionally, you may implement this functionality in O(n log n) in later step.

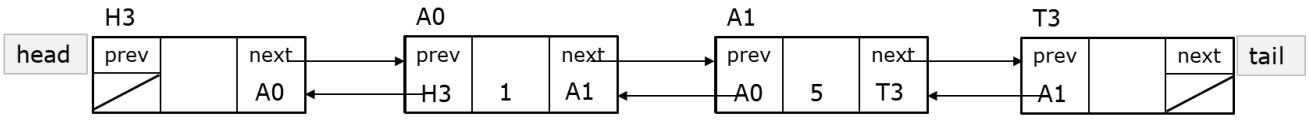
Step 5: reverse()**

This function reverses the order of the nodes in the list. The entire operation does not involve the construction, destruction or copy of any node. Nodes are not moved, but pointers are moved within the list. It must perform in O(n),

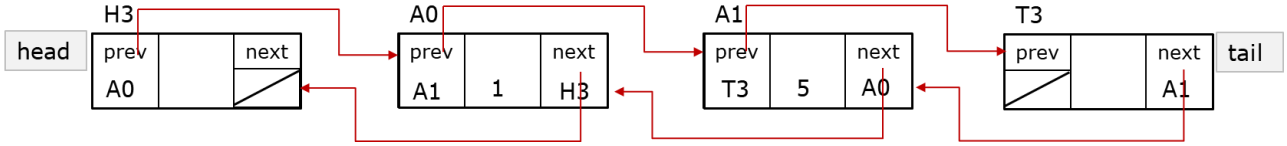
```
// reverses the order of the nodes in the list. Its complexity is O(n).
void reverse(pList p) {
  if (size(p) <= 1) return;
  // your code here
}
```

It may be the most difficult part of this pset. The following diagram shows two step procedures that may help you a bit.

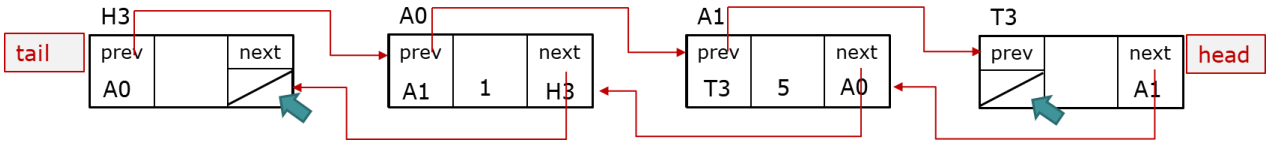
Original list given:



step 1: swap prev and next in every node.



step 2: swap head and tail node.



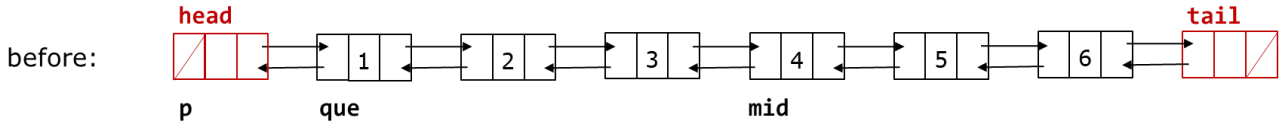
Step 6: shuffle()***

This function returns so called "perfectly shuffled" list. The first half and the second half are interleaved each other. The shuffled list begins with the second half of the original. For example, 1234567890 returns 617283940.

Algorithm:

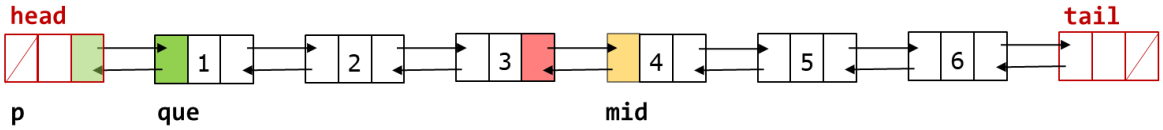
1) find the mid node of the list p to split it into two lists at the mid node.
2) remove the 1st half from the list p, and keep it as a list "que" to add.
3) set the list p head such that it points the "mid" of the list p.

4) keep on interleaving nodes until the "que" is exhausted.
save away next pointers of mid and que.
interleave nodes in the "que" into "mid" in the list of p.
(insert the first node in "que" at the second node in "mid".)



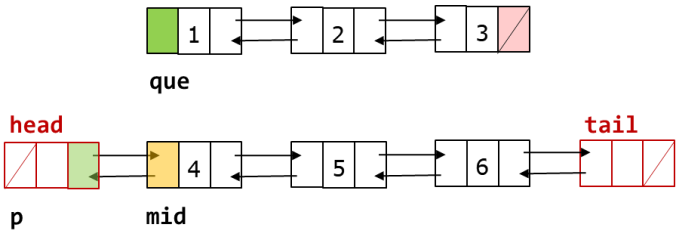
Step 1) find the mid node of the list p to split it into two lists at the mid node.

```
pNode mid = half(p);
pNode que = begin(p);
```



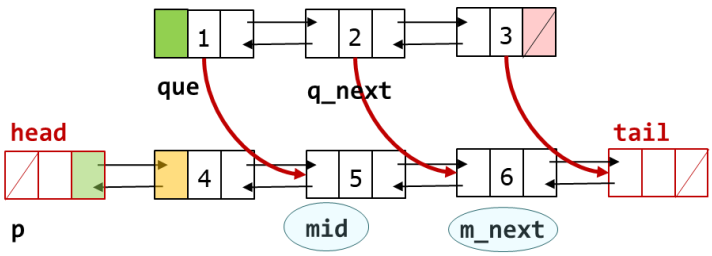
Step 2) remove the 1st half from the list p, and keep it as a list "que" to add.

Step 3) set the list p head such that it points the "mid" of the list p.



Step 4) keep on interleaving nodes until the "que" is exhausted.

- save away next pointers of mid and que.
  interleave nodes in the "que" into "mid" in the list of p.
  (insert the first node in "que" at the second node in "mid".)



Step 7: push_sortNlog in O(n log n) ***

The `push_sortNlog()` function inserts N number of nodes in sorted in the sorted list. The goal of this function is to make it O(n log n) instead of O(n^2).

Algorithm:

Step 1. Generate N numbers to insert. Let's name this array, vals.

Step 2. Sort vals using quicksort() of which time complexity is O(n log n), in ascending or descending depending on the list. .

Step 3. Merge two lists.
Compare two values from the list and vals one by one. For example, if sorted ascending and vals is smaller, insert the vals into the list and go for the next val. The list pointer does not increment. If vals is larger, then the list pointer increment, but vals index does not increment.

Step 4. If the list is exhausted, then exit the loop. If vals is not exhausted, insert the rest of vals at the end of the list.

Make sure that you go through a loop the list and vals together once. This is the same concept used in the most famous "mergesort" algorithm except recursion. The values for new nodes are randomly generated in the range of `[0..(N + size(p))]`.

For mac, use `rand()`. For pc, use `(rand() * RAND_MAX + rand())`

Step 8. Test scores

In this problem set, testing your algorithms is crucial. Unless you implement algorithms with the time complexity specified, your solution may not be valid. The **push_backN()** helps you build a large test vector (or list).

1. Running push_backN() once will be enough for most cases. For "pop vals" or "unique", you may run it twice as shown below.
   ① create 50,000 nodes filled with random values at the first run.
   ② create 50,000 nodes filled with a fixed value such as 100,000 or 3 at the second run.
2. Now you are ready to use this vector to test O(n), O(n log n), and O(n^2)
3. You may test your code with 1,000,000 nodes and compare them with **listdsx.exe.**

N		10,000	100,000	1,000,000	
push sorted O(n)	my code				
	listdsx				
selection sort O(n^2)	my code			xxxx	takes too long unless use quicksort
	listdsx			xxxx	
reverse O(n)	my code				
	listdsx				
pop vals O(n)	my code				
	listdsx				
unique O(n)	my code				
	listdsx				
shuffle O(n)	my code				
	listdsx				
push sortedN O(n^2)	my code				N = 10, 000, 100,000, 1,000,000
	listdsx				
push sortedN O(n log n)	my code				
	listdsx				

Submitting your solution

- Include the following line at the top of your every file with your name signed. On my honour, I pledge that I have neither received nor provided improper assistance in the completion of this assignment. Signed: _____
- Make sure your code **compiles** and **runs** right before you submit it.
- If you only manage to work out the homework partially before the deadline, you still need to turn it in. However, don't turn it in if it does not compile and run.
- Place your source files in the folder you and I are sharing.
- After submitting, if you realize one of your programs is flawed, you may fix it and submit again as long as it is **before the deadline**. You may submit as often as you like. **Only the last version** you submit before the deadline will be graded.

Files to submit

- Submit **two source** files, and test score file: listds.cpp, listsort.cpp, testscores.docx
- Use pset8 for step 1- 4 and extra2 folder for Bonus Step 5 - 7.

Due and Grade points

- Due for Step 1 - 4: 11:55 pm, May. 3, 2019
- Due for **Bonus Step** 5 – 7: 11:55 pm, May. 6, 2019
- Grade points:
  - Step 1, 2, 3, 4.1, 4.2, 4.3: 1 point each
  - **Bonus Step** 5, 6, 7: 2 points each