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.

PSet listdbl: a doubly-linked list

Table of Contents

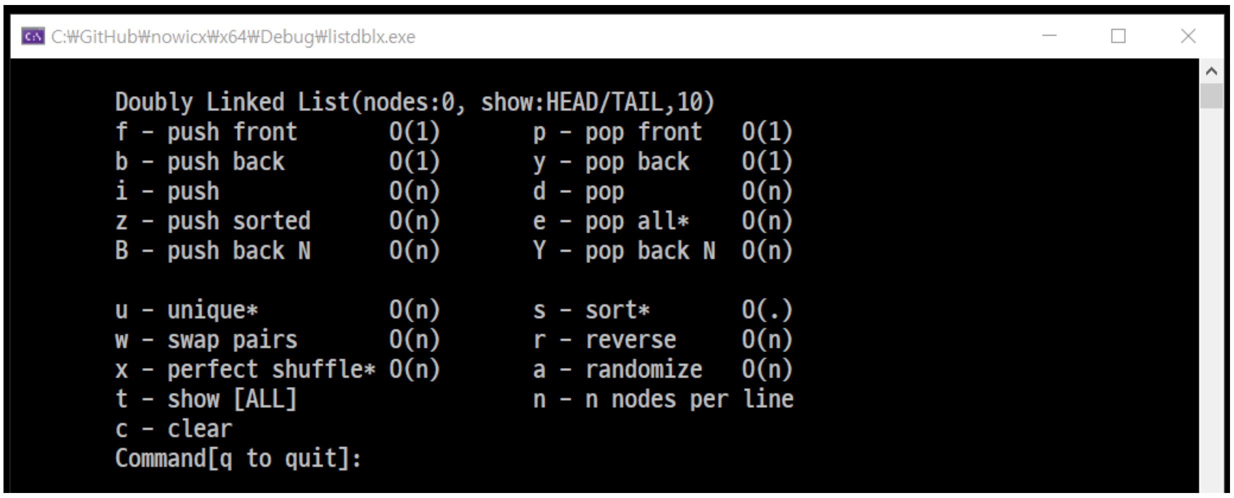
Getting started 1
Warming-up 2
    Key operation 1: begin() and end() 3
    Key Operation 2: erase() and insert() 3
Step 1. Basic operations: find(), more() and less() 5
Step 2: push commands 6
    helper functions: rand_extended() 6
Step 3: pop commands – pop_all()* 7
Step 4: show()*, half() 8
Step 5: swap_pairs() 10
Step 6: sort*, selectionSort() and insertionSort() 10
    helper functions: sorted() 10
Step 7: push_sorted() 11
Step 8: unique()* 12
Step 9: reverse() 12
Step 10: Randomize() 13
Step 11: perfect shuffle()** 14
Step 12: Self-Testing 16
Submitting your solution 16
    Files to submit 16
    Due and Grade points 16

Getting started

This problem set consists of implementing a doubly-linked list with two sentinel nodes. Your job is to complete the given program, **listdbl.cpp**. The following files are provided.

1. listdbl.h – Don't change this file.
2. listdblDriver.cpp – Don't change this file.
3. listdbl.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. random.cpp - Don't change this file
7. listdblx.exe, listdblx – Provided for your references only. It may have bugs.
8. selftest.docx – Self testing.

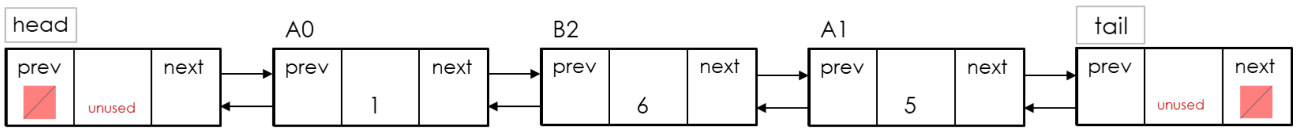
Sample run:



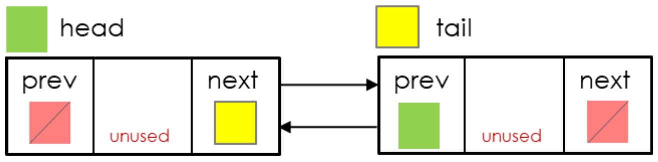
```
Doubly Linked List(nodes:0, show:HEAD/TAIL,10)
f - push front       O(1)       p - pop front   O(1)
b - push back        O(1)       y - pop back    O(1)
i - push             O(n)       d - pop         O(n)
z - push sorted      O(n)       e - pop all*    O(n)
B - push back N      O(n)       Y - pop back N  O(n)

u - unique*          O(n)       s - sort*       O(.)
w - swap pairs       O(n)       r - reverse     O(n)
x - perfect shuffle* O(n)       a - randomize   O(n)
t - show [ALL]                  n - n nodes per line
c - clear
Command[q to quit]:
```

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     data;
  Node*   prev;
  Node*   next;
};

struct List {
  Node* head; //sentinel
  Node* tail; //sentinel
  int   size; //size of list, optional

  List() {
      head = new Node{};    tail = new Node{};
      head->next = tail;    tail->prev = head;
      size = 0;
  }
  ~List() {}
};

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

Key operation 1: 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**.

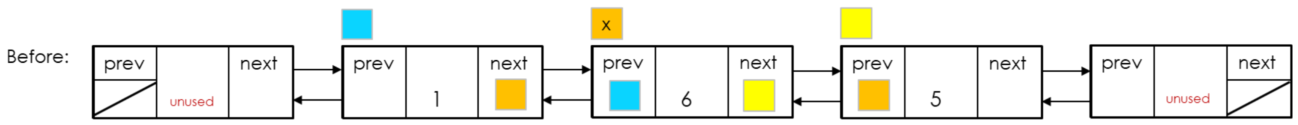
```
pNode end(pList p) {
  return p->tail;          // not tail->next
}
```

Key Operation 2: erase() and insert()

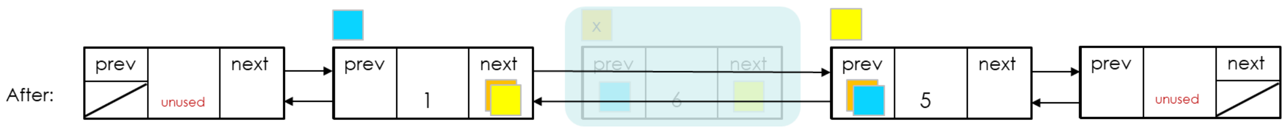
The function **erase()** removes from the list a single node x given. This effectively reduces the container by one which is destroyed. It is specifically designed to be efficient inserting and removing a node regardless of its positions in the list such as front, back or in the middle of the list.

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 (orange) is removed, now, we must link **node(blue)** and **node(yellow)**. This means that the first **orange**(**blue→next or x→prev→next)** must set by yellow(**x→next**) and the second orange(**yellow→prev** or **x->next->prev**) must be set by **x→prev.**

```
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:

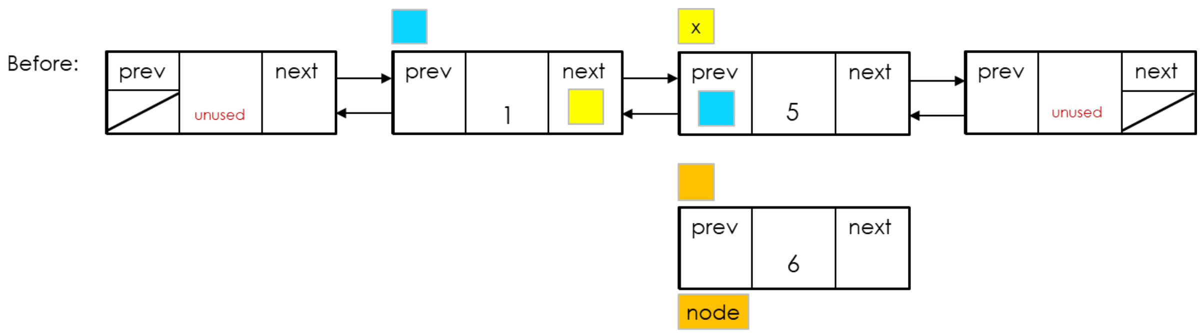
```
void erase(pList p, pNode x) { // checks if x is neither tail nor head
       if (x == p->tail || x == p->head || x == nullptr) return;
       x->prev->next = x->next;
       x->next->prev = x->prev;
       delete x;
}
```

The function **insert()** extends the container inserting a new node with **value 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.

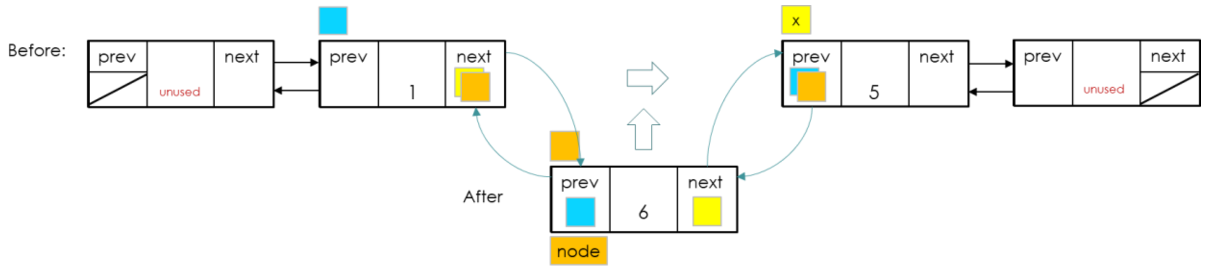
Let us suppose that we want to insert the node **x** with **value**. The insert() function has two arguments as shown below:

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

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.



We create one node (orange) with the value 6 and insert it between the node blue and the node yellow as shown below:



Then we set four places with new pointers:

These can be expressed in real code

1. orange: node→prev
2. orange: node→next
3. yellow: x or x→prev→next
4. blue: 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(value, x->prev, x);
```

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

Step 1. Basic operations: find(), more() and less()

The find() returns the node of which data item is the same as x firstly encountered in the list and **tail** if not found. In skeleton code, it is given as shown below:

```
pNode find(pList p, int value) {
  pNode curr = begin(p);
  for (; curr != end(p); curr = curr->next)
    if (curr->data == value) return curr;
  return curr;
}
```

Your job is to rewrite find() such that it does not use for-loop and if-statement, but use one while() loop. The idea is to add two conditions in while(.....).

Copy find() code and modify it to Implement **more()** and **less()** that returns the node of which data item is greater or smaller than x firstly encountered in the list and **tail** if not found.

```
pNode more(pList p, int x) {
   cout << "your code here\n");
}
```

```
pNode less(pList p, int x) {
  cout << "your code here\n");
}
```

Step 2: push commands

There are a few basic push functions related to push commands shown below:

- push()
- push_back()
- push_backN(int N)
- push_backN(int N, int value)
- push_front()

The function **push()** inserts a new node with value at the position of the node with x. The new node is actually inserted in front of the node with x. If the position x is not found, then it does **not** push the value to the list. Just ignore the input.

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

Hint: Use find() and insert(). Then this can be implemented in 2~3 lines only.

The **push_backN**() functions adds N number of new nodes at the end of the list. If the value is **not** given, randomly generated numbers are pushed in the range of [0..(N + size(p))]. Otherwise, simply insert the same value for N times. You may invoke push_back() by N times since it is the time complexity of O(n).

```
void push_backN(pList p, int N)
void push_backN(pList p, int N, int value)
```

Refer to the skeleton code or header file provided for further details.

helper functions: rand_extended()

One note about using rand() for random number generation When you use rand() to generate a random number, it generates numbers that are too small for our need since RAND_MAX is usually defined 32767. You may use the helper function called rand_extended().

Use a helper function, rand_extended(), provided with a skeleton code.

```
// returns an extended random number of which the range is from 0
// to (RAND_MAX + 1)^2 - 1. // We do this since rand() returns too
// small range [0..RAND_MAX) where RAND_MAX is usually defined as
// 32767 in cstdlib. Refer to the following link for details
// https://stackoverflow.com/questions/9775313/extend-rand-max-range

unsigned long rand_extended(int max_range) {
      if (max_range < RAND_MAX) return rand();
      return rand() * RAND_MAX + rand();
}
```

Step 3: pop commands – pop_all()*

A few basic pop functions that are related to pop commands are listed below:

- pop()
- pop_front()
- pop_back()
- pop_backN()
- pop_all()

The function **pop()** removes the first node with value 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 value);
```

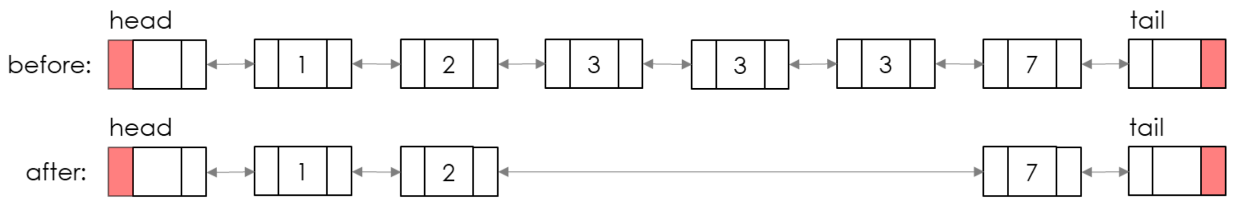
Pop_all() removes from the list all the nodes with the same value given. The following code works, but it goes through the list many times.

```
#if 1
// remove all occurrences of nodes with the value given in the list.
void pop_all(pList p, int value) {
  while (find(p, value) != end(p)) {
    pop(p, value);
  }
} // version.1
#else
// remove all occurrences of nodes with the value given in the list.
void pop_all(pList p, int value) {
  for (pNode c = begin(p); c != end(p); c = c->next)
```

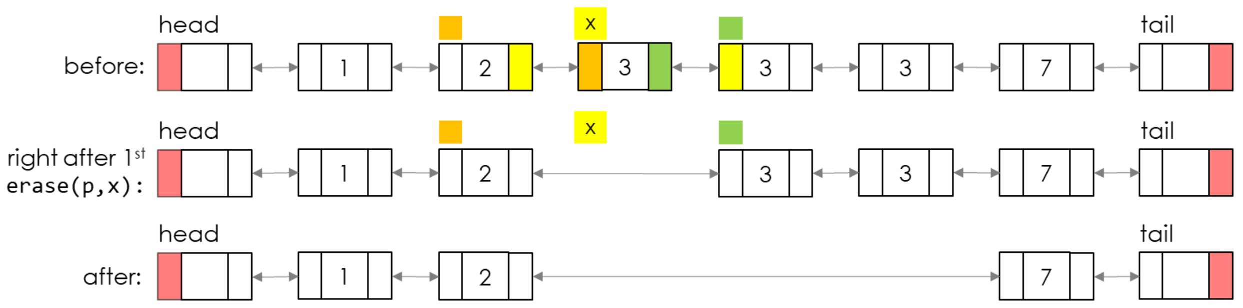
```
        if (c->data == value)  erase(p, c);
    } // version.2 fast but buggy
    #endif
```

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 as shown above. The following figure explains a potential problem in the code.



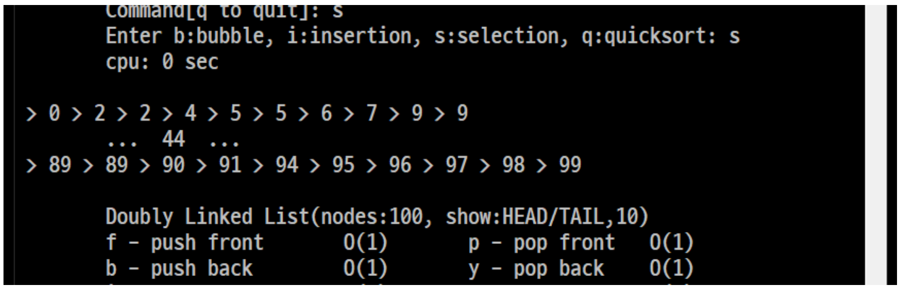
For example, this code removes 3 successfully. You may observe some failures when it has some consecutive occurrences of 3's in the list. Also add a print statement right after **erase()** and see what it prints after the removal of "3". To fix the bug, ask yourself a question about "Can x point the next 3 node right after **erasing x** in for-loop?"



Refer to the skeleton code or header file provided for further details.

Step 4: show()*, half()

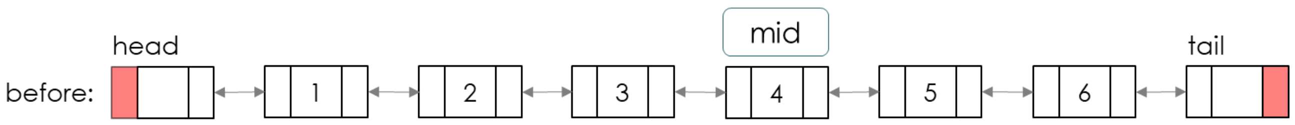
The show() function in your skeleton code displays all the nodes. Complete the show() function such that it works with two arguments, **show_all** and **show_n**. Also display the middle node if **show_all** is false.



```
        Command[q to quit]: s
        Enter b:bubble, i:insertion, s:selection, q:quicksort: s
        cpu: 0 sec

 > 0 > 2 > 2 > 4 > 5 > 5 > 6 > 7 > 9 > 9
        ...  44  ...
 > 89 > 89 > 90 > 91 > 94 > 95 > 96 > 97 > 98 > 99

        Doubly Linked List(nodes:100, show:HEAD/TAIL,10)
        f - push front       O(1)       p - pop front   O(1)
        b - push back        O(1)       y - pop back    O(1)
```

To display the middle node, we must find a midpoint of the list using the function named **half()**. 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 **"51"** which is at the midpoint among 1 ~ 100 nodes as shown above.

- 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 (odd number) nodes in the list, it returns the third one (or middle one).



There are a couple of ways to implement the **half()** function. You must implement both method 1 and 2. The method 2 is a faster as we studied in the class.

Implement Method 1:

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

```
#if 1
pNode half(pList p) {
  int N = size(p);
  // go through the list
  // break at the halfway point
  // return the current pointer
}
#endif
```

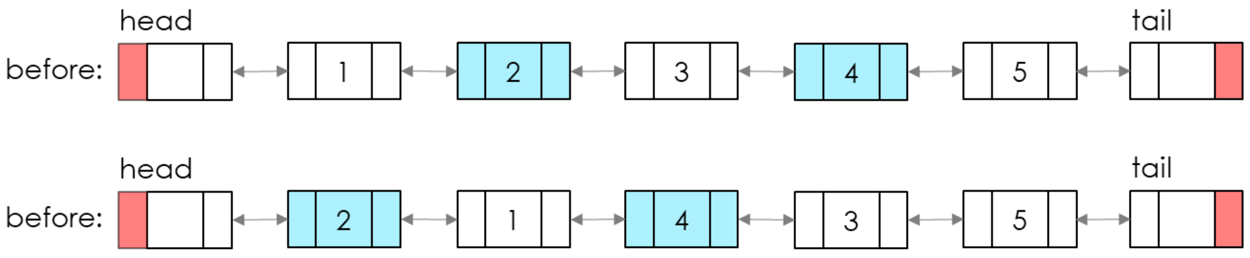
Implement Method 2:

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.

```
#if 0
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
#end
```

Step 5: swap_pairs()

This function is simple, but provide you with a good insight of a doubly linked list coding. It swaps two values of adjacent nodes in the list. It does not swap the links, but the values only. If the list has an odd number of nodes, then the last one remains as it is. It goes through the list once, its time complexity is O(n).



Step 6: sort*, selectionSort() and insertionSort()

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 selectionSort() and insertionSort 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->data, curr->next->data) > 0)
        swap(curr->data, curr->next->data);
    }
    tail = curr;
  }
}
```

helper functions: 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.

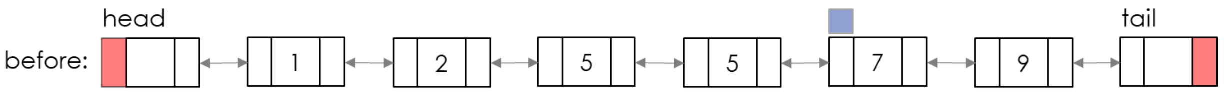
Step 7: push_sorted()

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

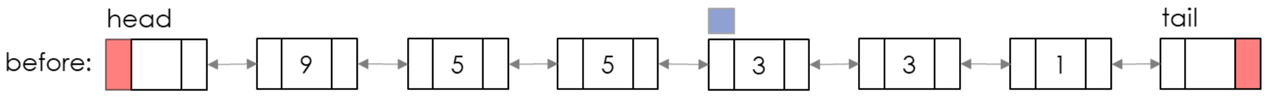
```
void push_sorted(pList p, int value)
```

If the list is sorted in ascending, you may invoke **insert()** with the position node x of which the value is greater than **value**. Use **more()** to find the node x which is greater than value, and **less()** for descending case.

To insert **value = 5**, for example, you must find the position node x with **value = 7** using **more()** function.,



Take the similar steps for descending ordered list. To insert **value = 5**, for example, you must find the position node x with **value = 3** using **less()** function.,



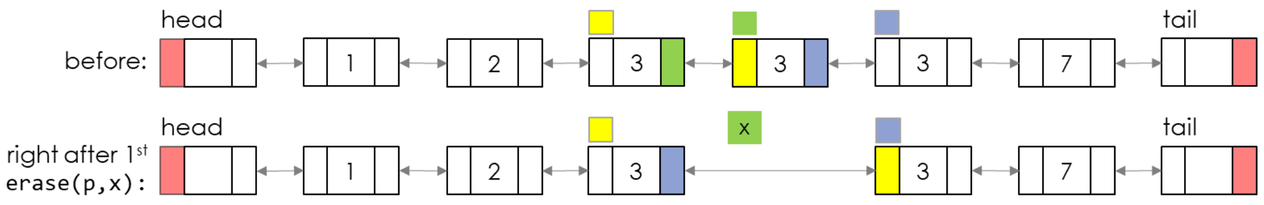
Hint: You may need the following functions to implement this part:
sorted(), insert(), more(), less(), ascending(), descending()

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

Step 8: 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

```
void unique(pList p) {
  if (size(p) <= 1) return;
  for (pNode x = begin(p); x != end(p); x = x->next)
    if (x->data == x->prev->data) erase(p, x);
} // version.1 buggy - it may not work in some machines or a large list.
```

To debug the skeleton code, answer the following question by yourself:
"Can x point the next node 3 **right after the first erase(p, x)**?"

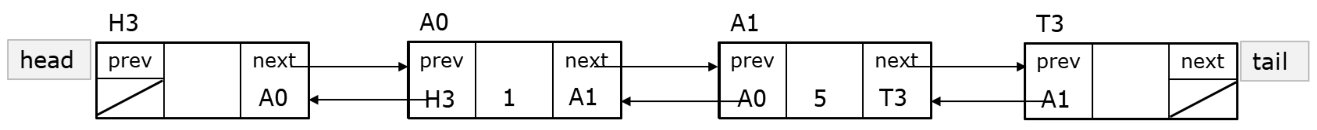
Step 9: reverse()

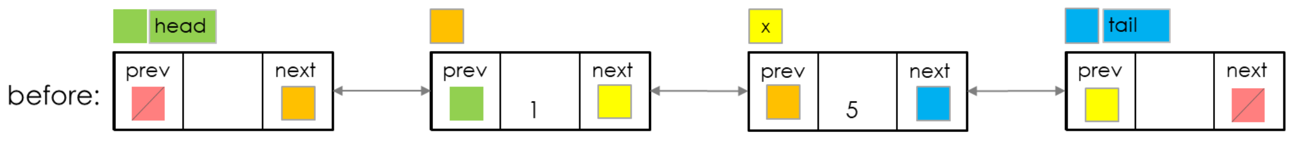
This function reverses the order of the nodes in the list. The entire operation does not involve the construction, destruction or copy of any node. Nodes are not moved, but pointers are moved within the list. It must perform in O(n),

```
// 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
}
```

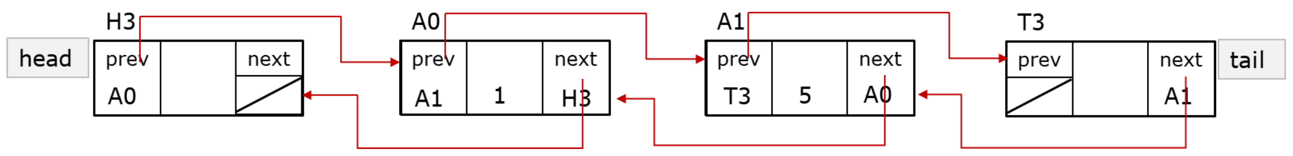
It may be the most difficult part of this pset. The following diagram shows two step procedures that may help you a bit.

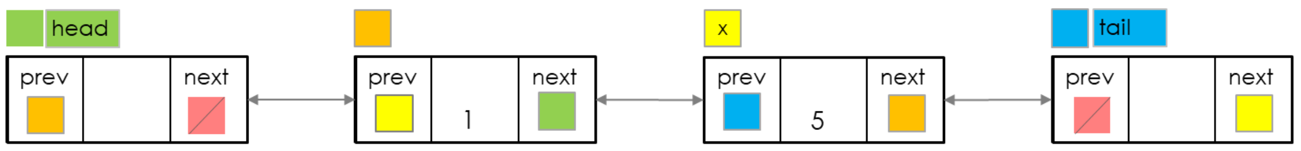
Original list given:





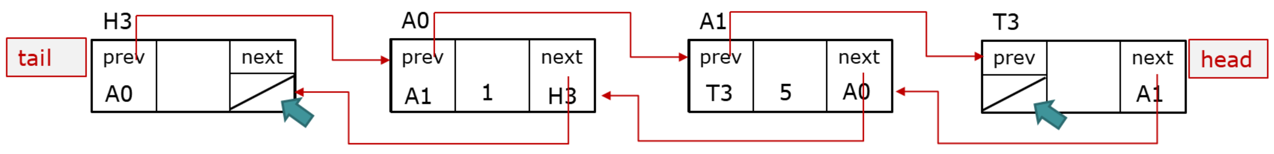
step 1: swap prev and next in every node.

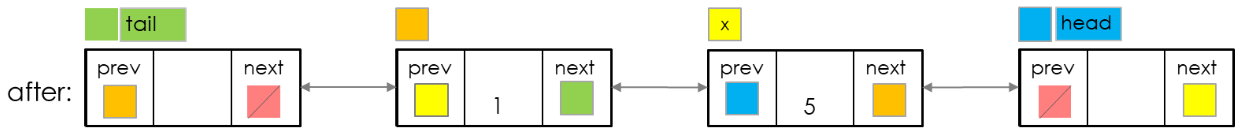




Notice that prev and next in sentinel nodes are also swapped.

step 2: swap head and tail node.





Step 10: Randomize()

There are many ways to implement this function. The most well-known algorithm is called the Fisher-Yates shuffle. You may refer to Wikipedia or **random.cpp** for detail. **The naive method** we use here swaps each element with another element chosen randomly from all elements. Even though this method is not the best, however, it is still acceptable for our practice purpose now.

The following functions are already implemented and given in the skeleton code.

```
// a helper function
pNode find_by_index(pList p, int n_th) {
  pNode curr = begin(p);
  int n = 0;
  while (curr != end(p)) {
    if (n++ == n_th) return curr;
      curr = curr->next;
  }
  return curr;
}

void randomize(pList p) {
  int N = size(p);
  if (N <= 1) return;
  pNode curr = begin(p);
  srand((unsigned)time(nullptr));

  curr = begin(p);
  while (curr != end(p)) {
    int x = rand_extended(N) % N;
    pNode xnode = find_by_index(p, x);
    swap(curr->data, xnode->data);
    curr = curr->next;
  }
}
```

The time complexity of the randomize() function shown above is O(n^2). Your job is to rewrite this code such that its time complexity becomes O(n). The main culprit of O(n^2) is to use find_by_index() O(n) inside while loop.

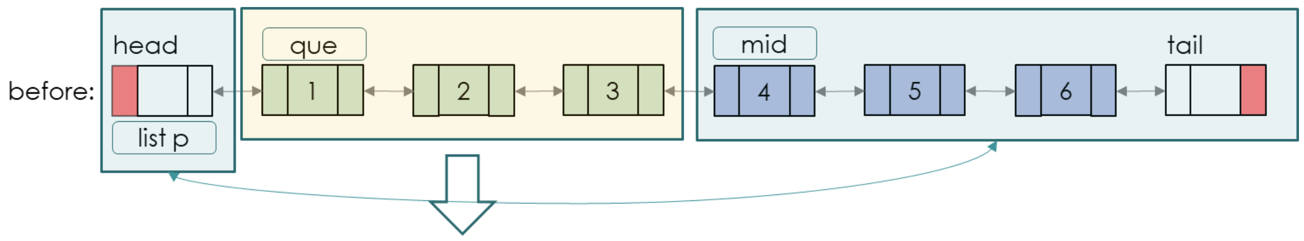
One way to solve is to extract all the list values before while() loop and save them into an array. Then, in while() loop, use the array of which the time complexity is O(1). Since it goes through the list twice, the time complexity becomes O(n) + O(n). One is to get the list of values into an array, and the other is to swap in while loop. Therefore the time complexity overall for the function is O(n).

Step 11: perfect 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 6172839405.

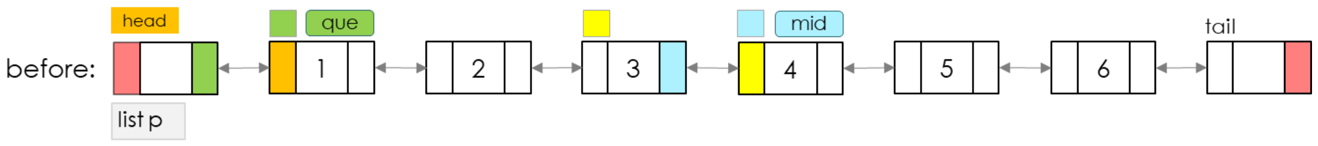
Algorithm:

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".)



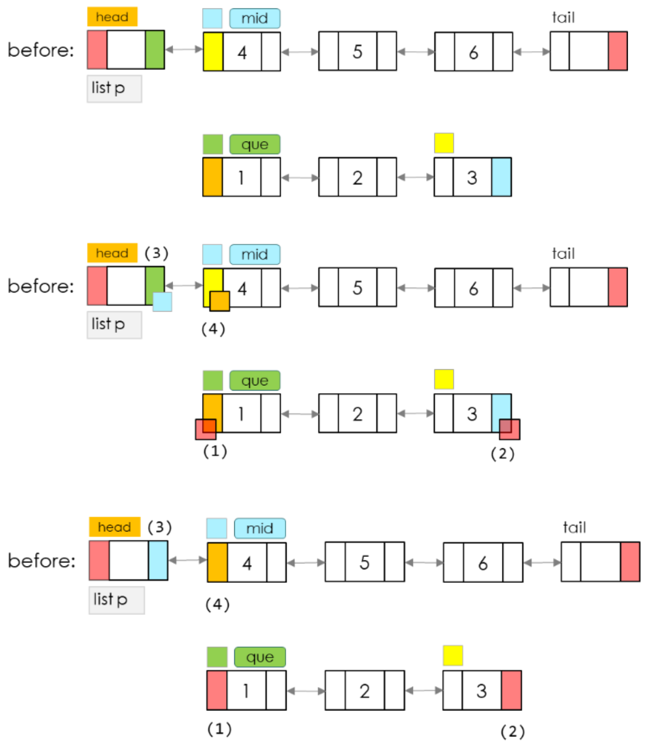
Step 1) find the mid node of the list p to split it into two lists at the mid node.

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



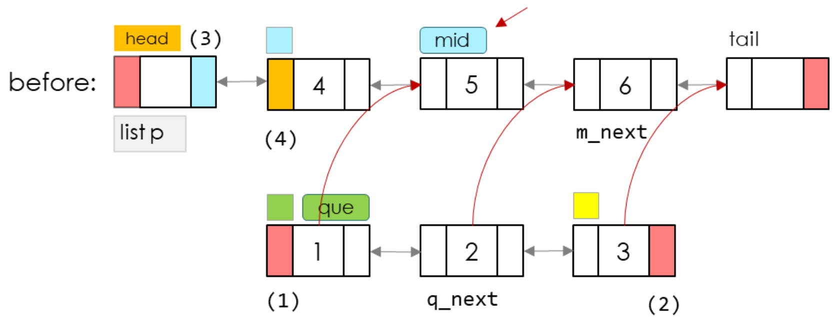
Step 2) remove the 1st half from the list p, and keep it as a list "que" to add.

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 12: Self-Testing

Refer to the file settest.docx provided.

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 the following files.
  - listdbl.cpp, listsort.cpp, self-test.docx
- Use pset 8 folder in piazza to upload your files.

Due and Grade points

- Due: 11:55 pm
- Grade points: Refer to listdblTest.docx