The following materials have been collected from the numerous sources such as Stanford CS106 and Harvard CS50 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 listnode: a singly-linked list

Table of Contents

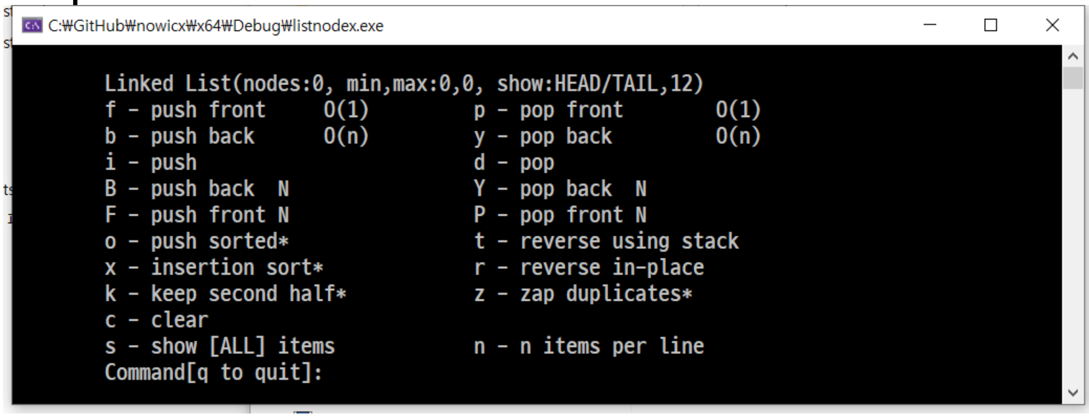
Getting Started 1
Step 1: push _front(), push_back(), push() 2
Step 2: pop_front(), pop_back(), pop() 2
Step 3: show(), clear(), minmax() – Copied from liststack.cpp 3
Step 4: "B", "F" and "Y", "P" 3
One note about using rand() for random number generation 4
Step 5: Reverse a singly-linked list – reverse_using_stack() 4
Step 6: Reverse a singly-linked list – reverse_in_place() 5
Step 7: push_sorted()* 5
Step 8: insertion_sort()* 6
Step 9: keep_second_half() 6
Step 10: zap_duplicates() 7
Step 11: Time complexity – write a report 7
Submitting your solution 8
Files to submit 8
Due and Grade 8

Getting Started

This problem set consists of implementing a simple singly-linked list of nodes. Your job is to complete the given program, **listnode.cpp**, that implements a singly linked list that don't have a header structure nor sentinel nodes. It simply links node to node and the first node always becomes a head node.

- listnode.cpp – a skeleton code, most of **your implementation goes here**.
  listnode.h – an interface file, you are **not** supposed to modify this file.
- listnodeDriver.cpp – a driver code to test your implementation.
- listnodex.exe – a sample solution for pc

Sample Run:.



```
        Linked List(nodes:0, min,max:0,0, show:HEAD/TAIL,12)
        f - push front     O(1)         p - pop front        O(1)
        b - push back      O(n)         y - pop back         O(n)
        i - push                        d - pop
        B - push back  N                Y - pop back  N
        F - push front N                P - pop front N
        o - push sorted*                t - reverse using stack
        x - insertion sort*             r - reverse in-place
        k - keep second half*           z - zap duplicates*
        c - clear
        s - show [ALL] items            n - n items per line
        Command[q to quit]:
```

Step 1: push _front(), push_back(), push()

The first function **push_front()** insert a node in front of the list. Since we don't have any extra header structure to maintain, the first node always acts like the head node. Since the new node is inserted at the front, the function must return the new pointer to the head node to the caller. The caller must reset the first node information. This is O(1) operation. By the way, this function is already implemented in the skeleton code.

For **push_back()** function, add a new node to the end of the existing list of nodes. If no list exists, the new node becomes the head node. To add a new node at the end, you must go through the list to find the last node. Once you find the last node, then you may add the new node there. This is O(n) operation.

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.

- If the list is empty, the new node with **val** becomes the first node or head of the list. In this case, the function argument **x** is ignored.
- If a node with **x** is not found in the list, it returns the original head pointer.
- Pay attention when you have just one node and push a node in that position. In this case, you can find the positional node with x, but there is no **prev** node to link the new node with **val**.

Hint: To insert a new node at (in front of) the node with x, you must have both the previous node of x and the node with x itself. Since the exactly same thing is used during **clear()**, you may refer to the code example in **clear()** provided.

```
pNode push(pNode p, int val, int x);
```

Important Note: To have a credit point for this step, you must simplify this part of code used during the lecture. This function is named "insert()" in the lecture. You may copy this part of code from the lecture material, but simplify while() loop such that it does not use "if" statement. You may combine two conditions of both while() and if() into one while loop.

Step 2: pop_front(), pop_back(), pop()

These functions delete a node from the linked list of nodes. The **pop_front()** removes the first node in the list and the second node becomes the first node or head node. This function return the pointer the newly first node which used to be a second node. This is O(1) operation.

The **pop_back()** removes the last node in the list. To remove the last one, you must go through the list to locate the **last node** as well as the **previous node**. Since the previous node of the last node becomes the last node, we must set its **next field to nullptr**. Therefore, you must get the last node as well as the previous node of the last one.

The **pop()** deletes a node with a specific value that the user choose. It could be any node in the list.

Important Note: To have a credit point for this step, you must simplify this part of code used during the lecture. You may copy this part of code from the lecture material, but simplify while() loop such that it does not use "if" statement. You may combine two conditions of both while() and if() into one while loop.

Step 3: show(), clear(), minmax() – Copied from liststack.cpp

The **show()** displays the stack, or the linked list of nodes from top to bottom. The function has an optional argument called **bool show_all = true**. Through the menu option in the driver, the user can toggle between "show [ALL]" and "show [HEAD/TAIL]". This functionality is useful when you debug your code.

It has another optional argument i**nt show_n = 12**. If the size of the list is less than or equal to show_n *2, then it displays all the items in the list. If "show [ALL]" option is chosen, it toggles to "show [HEAD/TAIL]" option and vice versa. The [HEAD/TAIL] option displays the show_n items at most from the beginning and the end of the list.

The **clear()** function deallocates all the nodes in the list. Make sure that you call "delete" N times where N is the number of nodes. To remove a node with x, you must have both the previous node of the node x and the node x itself. Since the exactly same thing is used during **clear(),** you may refer to the code example in **clear()** provided. This function is already implemented for you. It is a grace~.

The **minmax()** function finds the minimum and maximum values in the list. In fact, there is no necessity to have the min or max value in stack ADT. Nevertheless, we have this operation to check out the integrity of the code on purpose.

```
// sets the min and max values which are passed as references in the list
void minmax(pNode p, int& min, int& max);
```

Step 4: "B", "F" and "Y", "P"

All of these options ask the user to specify the number of nodes to be added or removed and performs the task. For example, B and F options works exactly same except one function call. The "B" calls push_back(), and "F" option calls push_front() N number of times respectively. The same thing can be applied for Y and P options.

- "B" – invokes push_back() N times with random numbers.
- "F" – invokes push_front() N times with random numbers.
- “Y" – invokes pop_back() N times.
- “P" – invokes pop_front() N times.

In "Y' and "P" options, if the user specifies a number out of the range, it simply removes all the nodes.

Implement one function which performs two options using a function pointer. Let the user (or driver) pass a function pointer (push_front or push_back) of their choice as shown below:

```
    case 'B':
     val = GetInt("\tEnter number of nodes to push back?: ");
     begin = clock();
     head = push_N(head, val, push_back);
     break;

    case 'F':
     val = GetInt("\tEnter number of nodes to push front?: ");
     begin = clock();
     head = push_N(head, val, push_front);
     break;
```

Then push_N() takes the function pointer as an argument, simply use the function pointer to invoke the necessary function which is either push_front() or push_back() as shown below:

```
// adds N number of new nodes at the front or back of the list.
// Values of new nodes are randomly generated in the range of
// [0..(N + size(p))].
// push_fp should be either a function pointer to push_front() or push_back().
Node* push_N(Node* p, int N, Node* (*push_fp)(Node *, int)) {

   // your code here
   push_fp(p, ...);
   ...
}
```

During you implementing this step, **you must edit the driver file** such that four options invoke pushN() and popN() functions, respectively, with appropriate function pointer.

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. 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 range) {
  if (range < RAND_MAX) return rand();
  return rand() * RAND_MAX + rand();
}
```

Step 5: Reverse a singly-linked list – reverse_using_stack()

Reverse a singly-linked list using a stack. This algorithm has been **explained during the lecture rather thoroughly** and the lecture material provided.

The last node of the input list becomes the head node of the newly formed list. The algorithm is shown below:

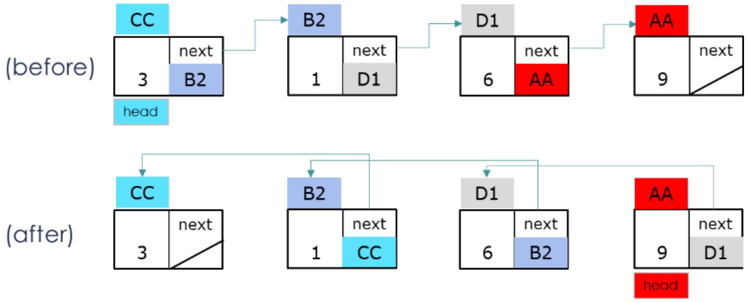
1. It pushes all of its nodes of the list onto a stack.
2. It tops nodes one at a time from the stack and relink them one by one again.
3. It finally returns the new head of the list.

Even though it goes through the list twice (push and pop) and takes a longer than in-place reverse algorithm.

Step 6: Reverse a singly-linked list – reverse_in_place()

Reverse a singly-linked list in-place which does not require a stack or extra memory. The function reverses a singly-linked list and returns the new head. The last node of the input list becomes the head node of the newly formed list.

This function expects to run faster by at least two times than reverse_using_stack().



Tips and Hints:

While you go through the list and swap two pointers of two nodes, you must save a couple of pointers before you make assignments.

Before while() loop, set prev = nullptr, and curr = head. During while() loop,

(1) Before setting curr→next to a new pointer, store curr→next as a temporary node.

(2) Before going for the next node in while loop, make sure two things:

A. set prev to curr (e.g. curr becomes prev)

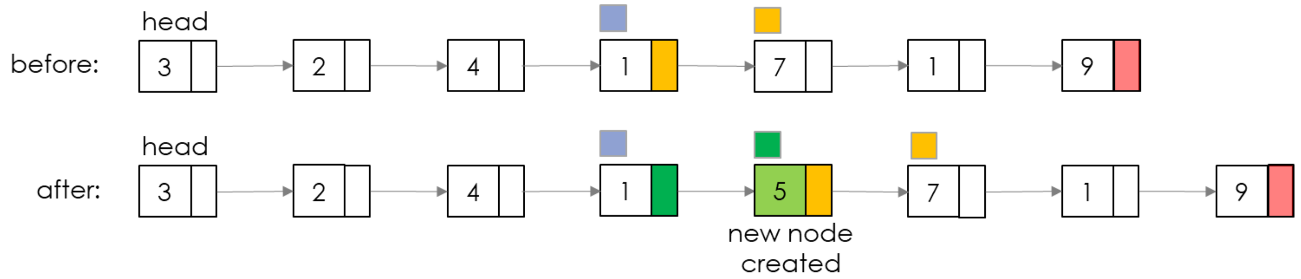
B. set curr to the next node you will process.

Step 7: push_sorted()*

The function **push_sorted()** inserts a new node in sorted ascending order. The basic strategy is to iterate down the list looking for the place to insert the new node. That could be the end of the list, or a point just before a node.

```
void push_sorted(Node* p, int value)
```

When you locate a place to insert the new node (7 in this example), you make sure that its address is stored in the previous node(1 in this example). The new node created is set to have 7's address and the previous node's next needs to be updated with the node's address.



Step 8: insertion_sort()*

The function **insertion_sort()** sorts the singly linked list using insertion sort and returns a new list sorted.

Repeatedly, invoke push_sorted() with a value in the list such that push_sorted() returns a newly formed list head..

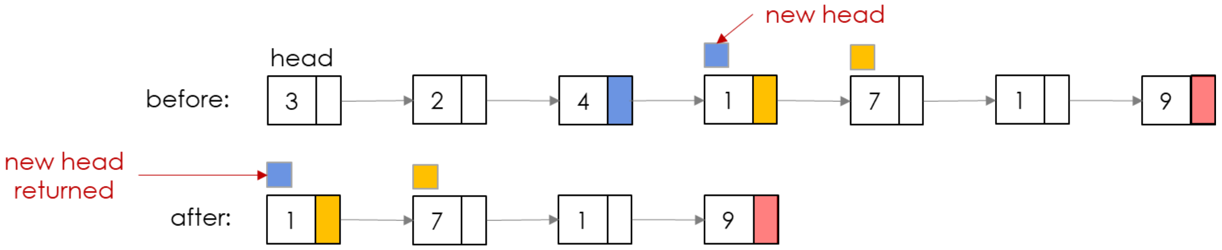
```
// inserts a new node with value in sorted order
Node* insertion_sort(Node* p) {
  if (empty(p)) return nullptr;
  if (size(p) < 2) return p;

  Node* sorted = nullptr;  // new list which will be formed using push_sorted()

  // your code here - a while loop invokes push_sorted() for all nodes in p

  clear(p);
  return sorted;
}
```

Step 9: keep_second_half()

The function **keep_second_half()** removes the first half of the list and returns the new head of the list which begins with the second half. If there are odd number of nodes, remove less and keep more. For example, keep 5 nodes if there are 9 nodes.

Just return the address of the middle node once you locate it in the list, Remember that the address of the middle node is stored in the previous node's next. Do not forget deallocating the first half nodes, one by one. It would be a bit easier to start if you copy and modify **clear()** function.

```
Node* keep_second_half(Node* p) {    // hint: copy and modify clear() provided.
  if (empty(p)) return nullptr;
  // your code here
  return new head ... ;
}
```

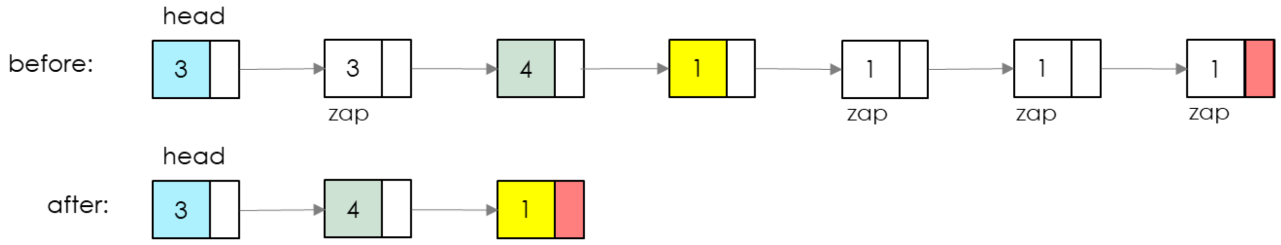


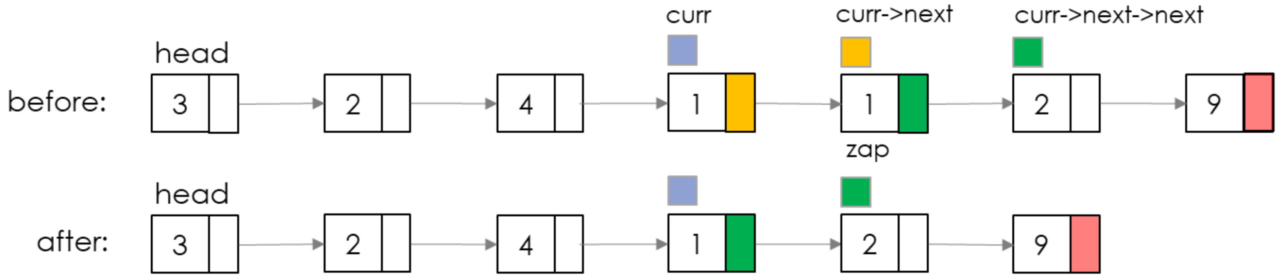
Step 10: zap_duplicates()

The function zap_duplicates Removes consecutive items in the list, and leaves its neighbors unique. We can proceed down the list and compare adjacent nodes.

When adjacent nodes are the same, remove the second one. There's a tricky case where the node after the next node needs to be noted before the deletion.

Your implementation must go through the list **only once**





Step 11: Time complexity – write a report

1. [2.5p] Complete the time complexity using big O notation of the following operations we implemented in this pset.

Operations	Time complexity	Operations	Time complexity
f - push front	O(1)	p - pop front	O(1)
b - push back	O(n)	y - pop back	O(n)
i - push		d - pop	
B - push back N		Y - pop back N	
F - push front N		P - pop front N	
o - push sorted		t - reverse in-stack	
x - insertion sort		r - reverse in-place	
k - keep second half		z - zap duplicates	

2. [1.5p] If you marked any operations listed above as O(n^2) and above, then
   A. For each operation, prove or explain how you got the answer.
   B. For each operation, compute the **growth rate b** as we learned before in the lab for each command. Show the timing measured and the steps of your calculation. Compute an estimated elapsed time for one million samples.

Submitting your solution

- Include the following line at the top of your every source file with your name signed.
  On my honor, I pledge that I have neither received nor provided improper assistance in the completion of this assignment.
  Signed: ______________ Section: _______ Student Number: __________________
- Make sure your code **compiles** and **runs** right before you submit it. Every semester, we get dozens of submissions that don't even compile. Don't make "a tiny last-minute change" and assume your code still compiles. You will not get sympathy for code that "almost" works.
- If you only manage to work out the problem sets 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 will have to resubmit any related files together, even if you only change one. 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 listed below in **pset7-2** folder in Piazza

- listnode.cpp
- timecomplexity.docx or in other standard formats

Due and Grade

- Due: 11:55 pm
- Grade: 18 points
  - Step 1 ~ 6: 1 point per step
  - Step 7 ~10: 2 points per step
  - Step 11: 4 points