Pset listnode: a singly-linked list

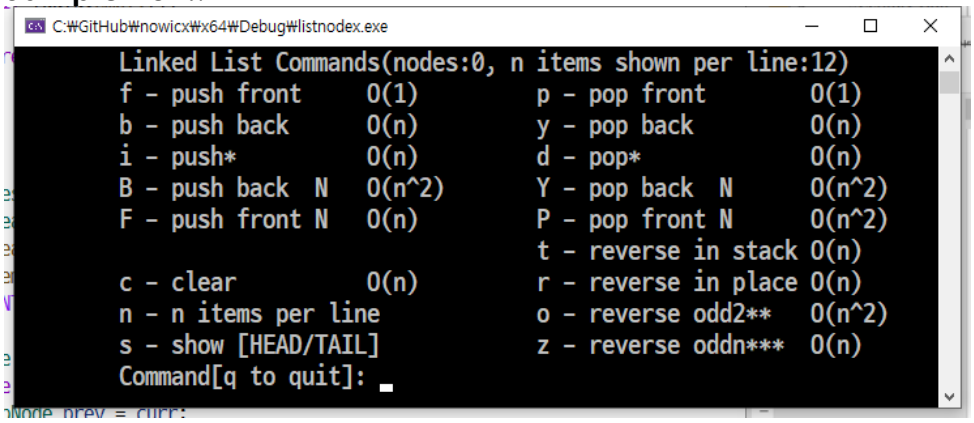
Table of Contents

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, listnodex – a sample solution for pc or macOS
- SelfGrading.docx – your test results as well as self-evaluation on your

Sample Run:.



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);
```

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.

Step 3: show() and clear()

The **show()** function displays the linked list of nodes. The function has an optional argument called **bool 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.

The show() function takes another optional argument **int 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 no matter which **option "all"** was chosen. If show[all] option is chosen, then display them all. If show[HEAD/TAIL] option is chosen, it displays the show_n items at most from the begging and the end of the list.

There is a command menu item to set show_n interactively.

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.
The good new is ..... This function is already implemented for you. It is a grace~.

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

Implement these stress test codes to check your implementation. The one of the four function, **pop_frontN()**, is provided for your reference.

- "F" – Implemented in push_frontN() which invokes push_front() N times with random numbers. It takes O(n)
- "B" – Implemented in push_backN() which invokes push_back() N times with random numbers. It takes $O(n^2)$
- "P" – Implemented in pop_frontN() which invokes pop_front() N times. It takes O(n)
- "Y" – Implemented in pop_backN() which invokes pop_back() N times. It takes $O(n^2)$

Each option asks the user to specify the number of nodes to be added or removed and performs the task. In "P" and "Y' options, if the user specifies a number out of the range, it simply removes all the nodes.

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_exteded(), 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: push_N() for "F" and "B", pop_N() for "P" and "Y"

Once you implemented those three options above, you realizes that two functions implemented F and B options or "P" and "Y" are exactly same except one function call. "F" option calls push_front() and "B" calls push_back(). It violates one of the programming principle – DRY. How can we avoid it?

We must implement a function that takes care of 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.

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

Reverse a singly-linked list using a stack. 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.

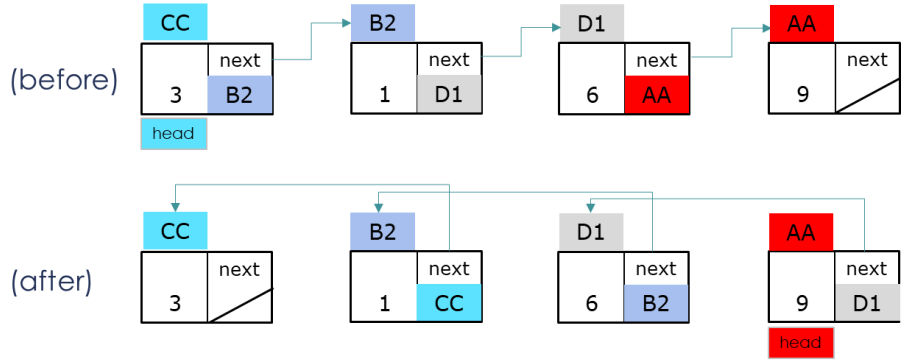
2. It tops nodes one at a time from the stack and relink them one by one again.
3. It finally returns the new head of the list.

Even though it goes through the list twice (push and pop) and takes a longer than in-place reverse algorithm. Its time complexity is still O(n), but the space complexity is O(n).

Step 7: 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. Since it goes through the list once, the time complexity of this function is O(n). Its space complexity is O(1)

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 temp.

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

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

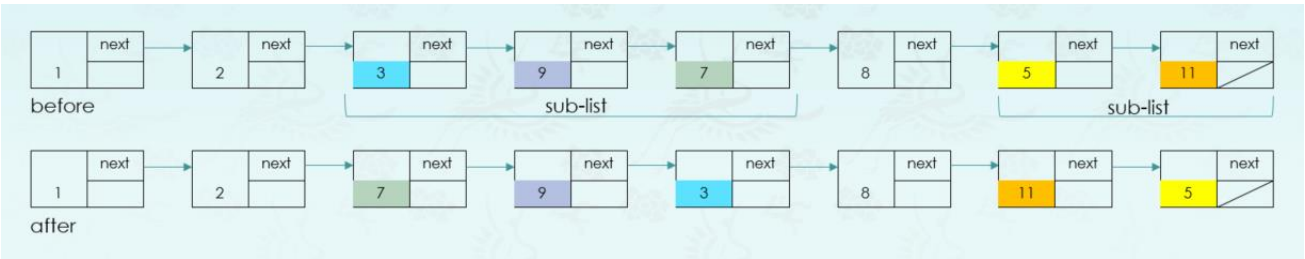
B. set curr to the next node you will process.

Step 8 & 9: Reverse odd number sub-lists

This function performs the reverse operation on sub-lists of nodes in a linked list that contains N integers. The sub-lists of the list contain only odd integers. Do not count one element as a sub-list. It reverses elements in each sub-group.

For example,

- If the list is {1, 2, 3, 9, 7, 8, 5, 11}, then the selected sub-lists will be {3, 9, 7} and {5, 11}. Then, reverse elements in the selected sub-lists such as {7, 9, 3} and {11, 5}. Now, this function returns the original list except odd numbers reversed. In this example, the function returns {1, 2, 7, 9, 3, 8, 11, 5}
- For a list of all odd numbers, its result will be reversed all the way.
- For a list of all even numbers, it would not change the list at all since there is no sub-lists of odd numbers



Step 8: Reverse odd number sub-lists O(n^2) – reverse_odd2()

This function reverses elements in sub-lists of odd numbers using stack and push_back(). Since it uses push_back() of which the time complexity is O(n^2, its time complexity becomes O(n^2).

The following skeleton code and algorithm are provided for your guidance. You may develop your own algorithm:

- `head` is the original list head.
- `head2` is the new list as a result.
- `odd_stack` stacks up odd(s) until an even shows up.
- You may use either `stack<int>` or `stack<Node*>`, but recall that `push_back()` takes a data item, not a node.

```
while (head != nullptr) {  // head is the original list head
  if the node data is odd {
    push it to odd_stack   // odd_stack keeps odd(s) until even number show up
    go for the next node
    continue;
  }
  // even node occurred
  while (odd_stack is not empty) {
    push_back top of odd_stack to the head2 // head2 - list of odds reversed
    pop odd_stack
  }
  add even node to head2
  go for the next node
}

// The following code takes care of odds which were left on the odd_stack
// Those numbers happened to be at the end of the original list.
if there is any items left on the odd_stack
  push_back them into head2

clear the original list
return head2
```

Step 9: Reverse odd number sub-lists O(n) – reverse_oddn()

This function reverses elements in sub-lists of odd numbers using stack.

- It does not use `push_back()` such that it can achieve its time complexity of O(n).
- Do not use `stack<int>`, but `stack<Node*>` to reuse the nodes.
- It does not create new nodes at all, but recycles the nodes of the original list

and relink them appropriately.

- It maintains the `head2 and tail2` of the new list such that it can relink the existing node at the back(or tail) in O(1), while scanning the original list once.
- Do not clear the list `head` since you reused all the nodes to construct the new list `head2`.

```
while (head != nullptr) {
  if the node is odd {
    push it to odd_stack
    go for the next node
    continue
  } // even node encountered
  while (odd_stack is not empty) {
    get top of odd_stack & pop
    head2 is null, set head2
    add it to the tail2 & set tail2
  }
  head2 is null, set head2
  add even node to tail2 & set tail2
  go for the next node
}
while( odd_stack is not empty) {
  get top of odd_stack & pop
  head2 is null, set head2
  add it to the tail2 & set tail2
}  // no clear head necessary
return head2
```

Since you construct a new list that consists of nodes from original list by yourself (not by `push_back()`), you can maintain two pointers (`head2` and `tail2`). When you have the tail of list, you can add a new element at O(n). This algorithm is almost same as above, but you have extra variable **tail2** by yourself. Refer to the he skeleton code provided with your pset.

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** folder in Piazza

- listnode.cpp
- Self-Grading.docx - required for submission, may be plus to your grade

Due and Grade

- Due: 11:55 pm
- Grade: 17 points – Midterm 2
  - (4 points) Step 1 – 4: 1.0 point per step
  - (8 points) Step 5 – 8: 2.0 points per step
  - (3 points) Step 9
  - (2 points) Self-Grading