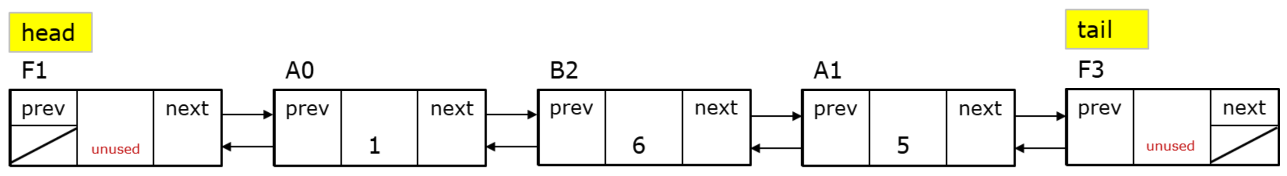
The following materials have been collected from the numerous sources including my own and my students over the years of teaching and experiences of programming. Please help me to keep this tutorial up-to-date by reporting any issues or questions. Please send any comments or criticisms to idebtor@gmail.com.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

# a doubly linked list with sentinel nodes

## Table of Contents

## Warming-up

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

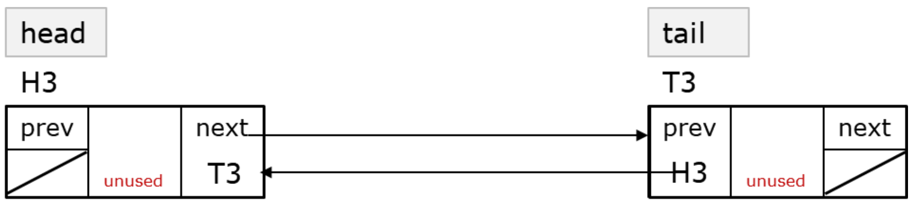


These extra nodes are sometimes known as **sentinel nodes**, specifically, the node at the front is known as **head** node, and the node at the end is known as a **tail** node.

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**.

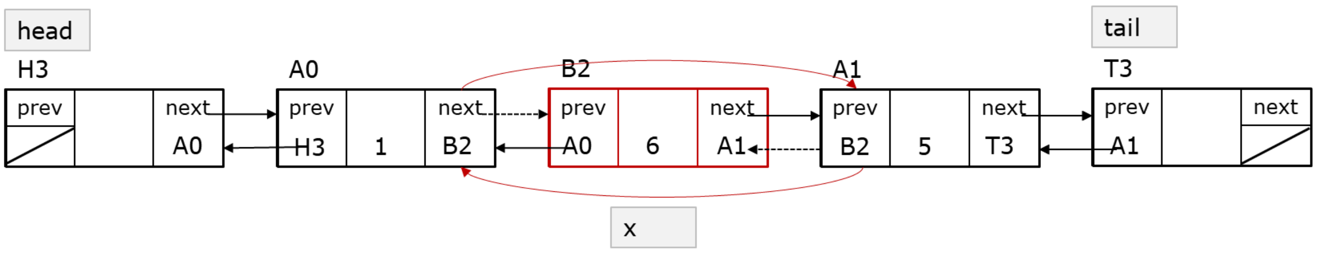
```
pNode end(pList p) {
  return p->tail;          // not tail->next
}
```

## key functions: erase() and insert()

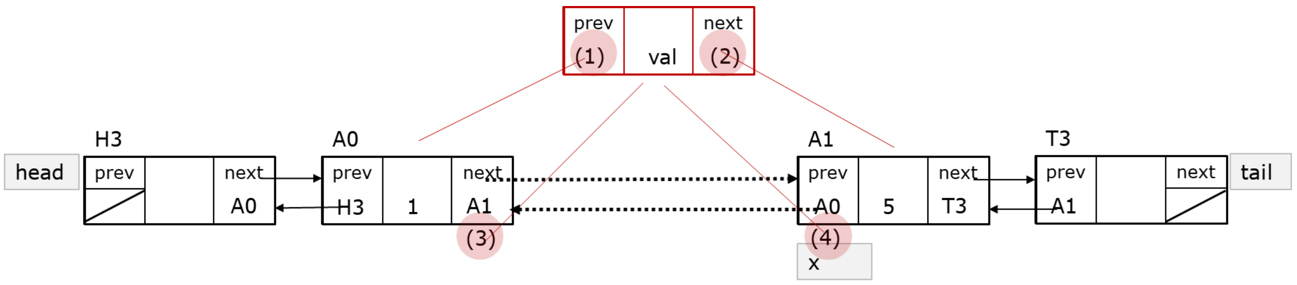
The function **erase()** removes from the list a single node x given.  This effectively reduces the container by one which is destroyed. It is specifically designed to be efficient inserting and removing a node regardless of its positions in the list such as front, back or in the middle of the list.

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

For example:



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.



## key functions: _more() and _less()

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

```
pNode _more(pList p, int x)
```

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

```
pNode _less(pList p, int x)
```

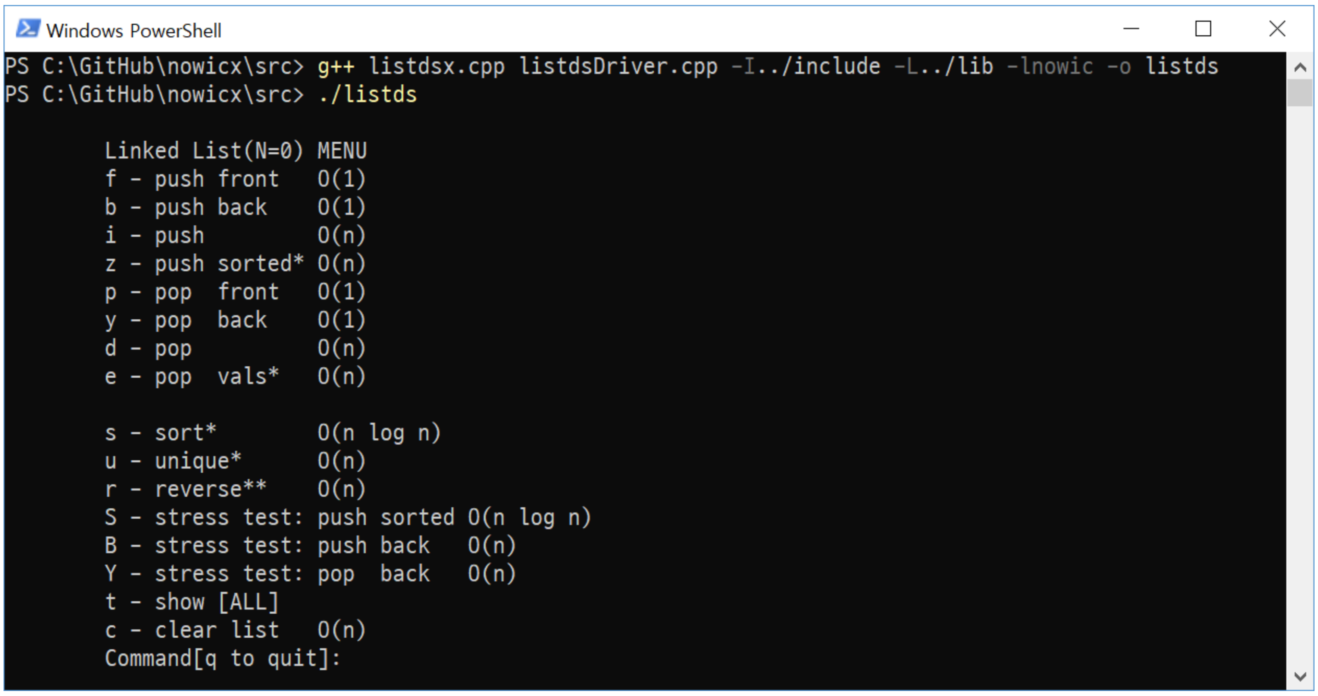
## Getting started

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 goes here.
4. listdsx.exe – It is provided for your reference;
5. 04-3linkedlist.pdf, 04-4linkedlist.pdf – a good reference for your work provided through github as you know

Sample run: listdsx.exe



```
PS C:\GitHub\nowicx\src> g++ listdsx.cpp listdsDriver.cpp -I../include -L../lib -lnowic -o listds
PS C:\GitHub\nowicx\src> ./listds

        Linked List(N=0) MENU
        f - push front   O(1)
        b - push back    O(1)
        i - push         O(n)
        z - push sorted* O(n)
        p - pop  front   O(1)
        y - pop  back    O(1)
        d - pop          O(n)
        e - pop  vals*   O(n)

        s - sort*        O(n log n)
        u - unique*      O(n)
        r - reverse**    O(n)
        S - stress test: push sorted O(n log n)
        B - stress test: push back   O(n)
        Y - stress test: pop  back   O(n)
        t - show [ALL]
        c - clear list   O(n)
        Command[q to quit]:
```

## 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);
```

The functional **pop_all()** removes from the list all the nodes with the same value given. This calls the destructor 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. (Hint: Consider to use erase().)

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

## Step 2: unique()

This function removes extra nodes that have duplicate values from the list. It removes all but the last 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 is especially useful for sorted lists. It should be done in O(n).

```
void unique(pList p)
```

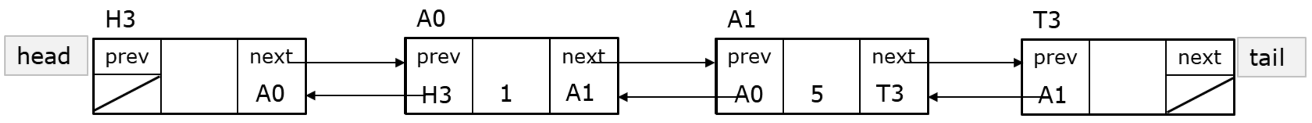
## Step 3: reverse()

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),

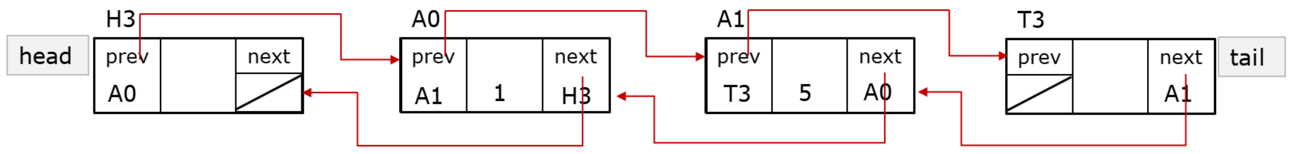
```
// reverses the order of the nodes in the list. Its complexity is O(n).
void reverse(pList p) {
  if (size(p) <= 1) return;
  // your code here
}
```

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

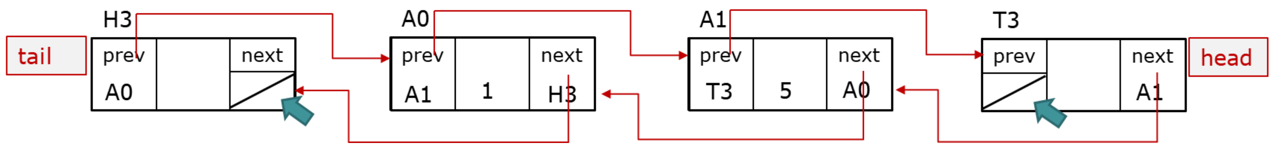
Original list given:



step 1: swap prev and next in every node.



step 2: swap head and tail node.



## Step 4-1: 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 or not it is sorted. Use the function pointer passed for comparing.

```
// returns true if the list is sorted either ascending or descending.
bool sorted(pList p) {
  if (size(p) <= 1) return true;
  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 4-2: push_sorted()

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

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

If the list is sorted in ascending, you may invoke **insert()** with the node of which item is greater than val.

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

## Step 4-3: sort()

If the list is already sorted, it simply reverses the list such that the ascending ordered list becomes a descending order list and vice versa. This operation is O(n).

If the list is not sorted, it simply removes one node from the head and put into a new list in sorted order using insert() and **_more()** functions as shown below:

```
insert(newp, _more(newp, val), val);  // newp is a new list sorted
```

A helper function **_more()** returns a node which is greater than the **val**. Thus the node **_more()** returned becomes the position where the new node with val goes in.. This algorithm is one of the worst one at this point. You are welcome to have your own algorithm to this part. The function **sort()** returns a new list of nodes sorted in ascending order if not sorted.

Hint: You may need the following functions to implement this part:
reverse(), insert(), pop_front(), empty()

## Step 4-4: push_sortN()

The function **push_sortN()** inserts N number of nodes in sorted order.

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

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

## 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 **one** file: listds.cpp,
- Use pset8 folder.

### Due and Grade points

- Due: 11:55 pm, May. 1, 2019
- Grade points:
  - Step 1: 1
  - Step 2: 1
  - Step 3: 2
  - Step 4: 2