Singly Linked list

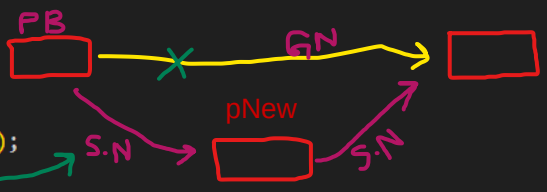
Insert first, insert middle end , delete from first, middle, end, delete maximum and minimum

D: > Linked List > Linked List > Singly.cpp > ...

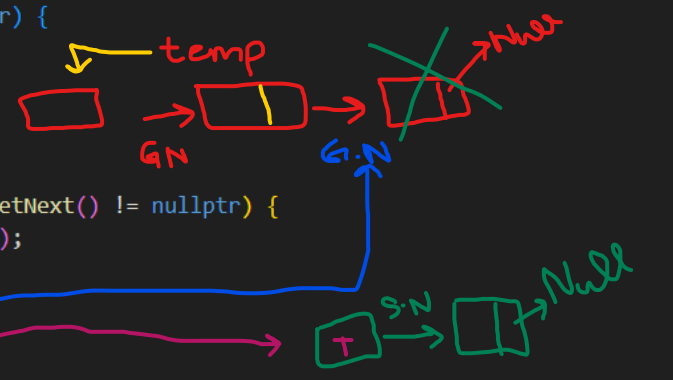
```cpp
#include <iostream>
using namespace std;

class Node {
private:
    int data;
    Node* next;

public:
    // Constructor
    Node(int element) : data(element), next(nullptr) {}

    // Setters
    void setData(int element) {
        data = element;
    }

    void setNext(Node* node) {
        next = node;
    }

    // Getters
    int getData() {
        return data;
    }

    Node* getNext() {
        return next;
    }
};

class List {
private:
    Node* head;

public:
    // Constructor
```

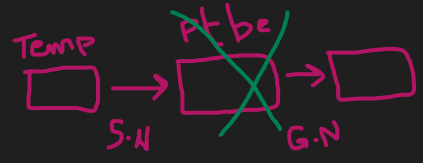
```
32  class List {
33  private:
34      Node* head;
35
36  public:
37      // Constructor
38      List() : head(nullptr) {}
39
40      // Insert at the beginning
41      void InsertBeginning(Node* pNew) {
42          pNew->setNext(head);
43          head = pNew;
44      }
45
46      // Insert at the end
47      void InsertEnd(Node* pNew) {
48          if (head == nullptr) {
49              head = pNew;
50          }
51          else {
52              Node* temp = head;
53              while (temp->getNext() != nullptr) {
54                  temp = temp->getNext();
55              }
56              temp->setNext(pNew);
57          }
58      }
59
60      // Insert in the middle (after pBefore)
```


```
32  class List {
60      // Insert in the middle (after pBefore)
61      void InsertMiddle(Node* pBefore, Node* pNew) {
62          if (pBefore == nullptr) {
63              head = pNew;
64          }
65          else {
66              pNew->setNext(pBefore->getNext());
67              pBefore->setNext(pNew);
68          }
69      }
70
71      // Delete from the beginning
72      void DeleteFromBeginning() {
73          if (head == nullptr) {
74              cout << "List is empty" << endl;
75              return;
76          }
77          Node* temp = head;
78          head = head->getNext();
79          delete temp;
80      }
81
82      // Delete from the end
```

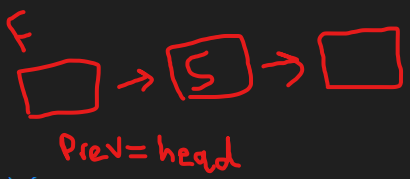


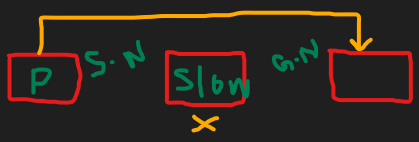
```
class List {
    // Delete from the end
    void DeleteFromEnd() {
        if (head == nullptr) {
            cout << "List is empty" << endl;
            return;
        }
        if (head->getNext() == nullptr) {
            delete head;
            head = nullptr;
        }
        else {
            Node* temp = head;
            while (temp->getNext()->getNext() != nullptr) {
                temp = temp->getNext();
            }
            delete temp->getNext();
            temp->setNext(nullptr);
        }
    }

    // Delete from the middle using a specific node
    void DeleteFromMiddle(Node* pToBeDeleted) {
        if (pToBeDeleted == head) {
            DeleteFromBeginning();
        }
        else {
            Node* temp = head;
            while (temp->getNext() != pToBeDeleted) {
                temp = temp->getNext();
            }
            temp->setNext(pToBeDeleted->getNext());
            delete pToBeDeleted;
        }
    }
```



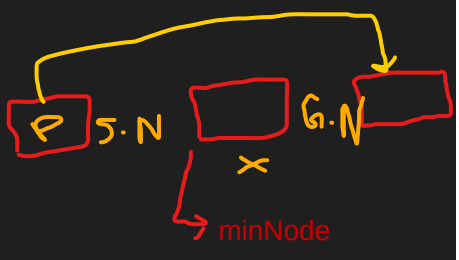


```
32  class List {
117     // Delete the middle node using the slow and fast pointer technique
118     void DeleteFromMiddle() {
119         if (head == nullptr || head->getNext() == nullptr) {
120             DeleteFromBeginning();
121             return;
122         }
123
124         Node* slow = head;
125         Node* fast = head;
126         Node* prev = nullptr;
127
128         // Traverse the list with fast and slow pointers
129         while (fast != nullptr && fast->getNext() != nullptr) {
130             prev = slow;
131             slow = slow->getNext();
132             fast = fast->getNext()->getNext();
133         }
134           DeletefromMiddle(slow);
135         // 'slow' is the middle node, and 'prev' is the node before the middle node
136         if (prev != nullptr) {
137             prev->setNext(slow->getNext());
138             delete slow;
139         }
140     }
141
142     // Delete the minimum value node
```






```
32  class List {
141
142     // Delete the minimum value node
143     void DeleteMinimum() {
144         if (head == nullptr) return;
145
146         Node* minNode = head;
147         Node* temp = head;
148         Node* prev = nullptr;
149         Node* prevMin = nullptr;
150
151         while (temp != nullptr) {      // Transverse
152             if (temp->getData() < minNode->getData()) {
153                 prevMin = prev;
154                 minNode = temp;
155             }
156             prev = temp;
157             temp = temp->getNext();
158         }
159
160         if (minNode == head) {
161             DeleteFromBeginning();
162         }
163         else if (prevMin != nullptr) {
164             prevMin->setNext(minNode->getNext());
165             delete minNode;
166         }
167     }
168
169     // Delete the maximum value node
```

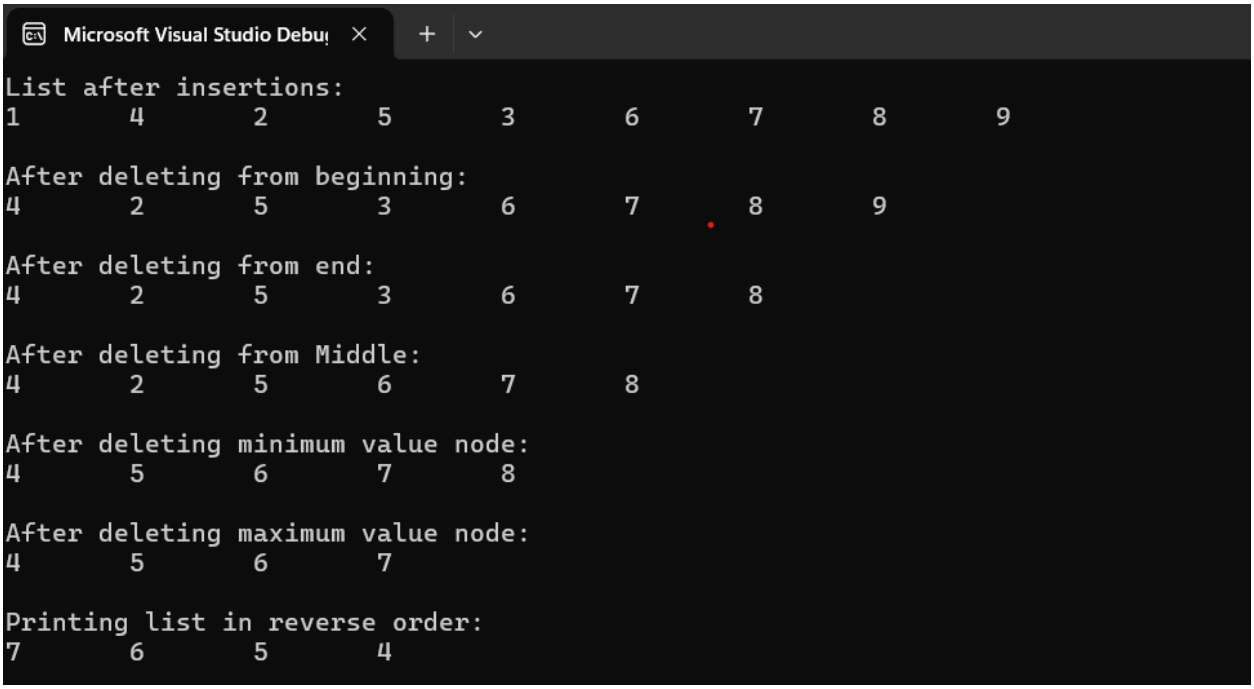


```
32  class List {
170     void DeleteMaximum() {
171         if (head == nullptr) return;
172
173         Node* maxNode = head;
174         Node* temp = head;
175         Node* prev = nullptr;
176         Node* prevMax = nullptr;
177
178         while (temp != nullptr) {
179             if (temp->getData() > maxNode->getData()) {
180                 prevMax = prev;
181                 maxNode = temp;
182             }
183             prev = temp;
184             temp = temp->getNext();
185         }
186
187         if (maxNode == head) {
188             DeleteFromBeginning();
189         }
190         else if (prevMax != nullptr) {
191             prevMax->setNext(maxNode->getNext());
192             delete maxNode;
193         }
194     }
195
196     // Print the list
197     void PrintList() {
198         Node* temp = head;
199         while (temp != nullptr) {
200             cout << temp->getData() << "\t";
201             temp = temp->getNext();
202         }
203         cout << endl;
204     }
```

```
32  class List {
206     // Recursive function to print list in reverse order
207     void PrintReverse(Node* root) {
208         if (root == nullptr) return;
209         PrintReverse(root->getNext());
210         cout << root->getData() << "\t";
211     }
212
213     // Public function to call recursive print
214     void PrintInReverse() {
215         PrintReverse(head);
216         cout << endl;
217     }
218 };
219
220 int main() {
221     Node* a = new Node(1);
222     Node* b = new Node(2);
223     Node* c = new Node(3);
224     Node* d = new Node(4);
225     Node* e = new Node(5);
226     Node* f = new Node(6);
227     Node* k = new Node(7);
228     Node* p = new Node(8);
229     Node* z = new Node(9);
230
231     List* list = new List();
232
233     list->InsertBeginning(a);  // Insert first node at the beginning
234     list->InsertEnd(b);        // Insert at the end
235     list->InsertEnd(c);
236     list->InsertMiddle(a, d);  // Insert in the middle
237     list->InsertMiddle(b, e);
238     list->InsertEnd(f);
239     list->InsertEnd(k);
240     list->InsertEnd(p);
241     list->InsertEnd(z);
```

Base Case

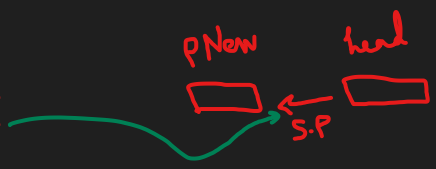
```
220 int main() {
236     list->InsertMiddle(a, d);  // Insert in the middle
237     list->InsertMiddle(b, e);
238     list->InsertEnd(f);
239     list->InsertEnd(k);
240     list->InsertEnd(p);
241     list->InsertEnd(z);
242
243     cout << "List after insertions:" << endl;
244     list->PrintList();
245
246     list->DeleteFromBeginning();
247     cout << "\nAfter deleting from beginning:" << endl;
248     list->PrintList();
249
250     list->DeleteFromEnd();
251     cout << "\nAfter deleting from end:" << endl;
252     list->PrintList();
253
254     list->DeleteFromMiddle();
255     cout << "\nAfter deleting from Middle:" << endl;
256     list->PrintList();
257
258     list->DeleteMinimum();
259     cout << "\nAfter deleting minimum value node:" << endl;
260     list->PrintList();
261
262     list->DeleteMaximum();
263     cout << "\nAfter deleting maximum value node:" << endl;
264     list->PrintList();
265
266     cout << "\nPrinting list in reverse order:" << endl;
267     list->PrintInReverse();
268
269     return 0;
270 }
```



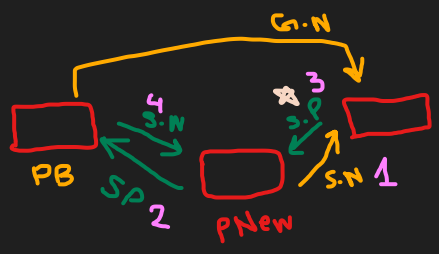
- **Doubly Linked list**

```cpp
class Node {
private:
    int data;
    Node* next;
    Node* prev;

public:
    // Constructor
    Node(int element) : data(element), next(nullptr), prev(nullptr) {}

    // Setters
    void setData(int element) {
        data = element;
    }

    void setNext(Node* node) {
        next = node;
    }

    void setPrev(Node* node) {
        prev = node;
    }

    // Getters
    int getData() {
        return data;
    }

    Node* getNext() {
        return next;
    }

    Node* getPrev() {
        return prev;
    }
};
```

```
41  class DList {
42  private:
43      Node* head;
44
45  public:
46      // Constructor
47      DList() : head(nullptr) {}
48
49      // Insert at the beginning
50      void InsertBeginning(Node* pNew) {
51          if (head == nullptr) {
52              head = pNew;
53          }
54          else {
55              pNew->setNext(head);
56              head->setPrev(pNew);
57              head = pNew;
58          }
59      }
60
61      // Insert at the end
62      void InsertEnd(Node* pNew) {
63          if (head == nullptr) {
64              head = pNew;
65          }
66          else {
67              Node* temp = head;
68              while (temp->getNext() != nullptr) {
69                  temp = temp->getNext();
70              }
71              temp->setNext(pNew);
72              pNew->setPrev(temp);
73          }
74      }
75
76      // Insert in the middle (after pBefore)
```

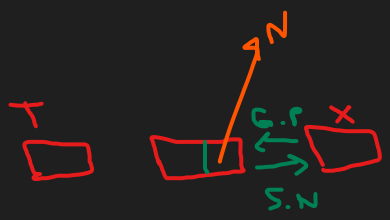


```cpp
    // Insert in the middle (after pBefore)
    void InsertMiddle(Node* pBefore, Node* pNew) {
        if (pBefore == nullptr) {
            head = pNew;
        }
        else if (pBefore->getNext() == nullptr) {
            pBefore->setNext(pNew);
            pNew->setPrev(pBefore);
        }
        else {
            pNew->setNext(pBefore->getNext());
            pNew->setPrev(pBefore);
            pBefore->getNext()->setPrev(pNew);
            pBefore->setNext(pNew);
        }
    }

    // Delete from the beginning
    void DeleteFromBeginning() {
        if (head == nullptr) {
            cout << "List is empty" << endl;
            return;
        }
        Node* temp = head;
        head = head->getNext();
        if (head != nullptr) {
            head->setPrev(nullptr);
        }
        delete temp;
    }

    // Delete from the end
    void DeleteFromEnd() {
        if (head == nullptr) {
            cout << "List is empty" << endl;
            return;
```

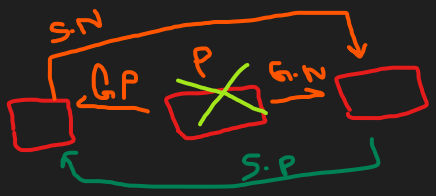


D: > Linked List > Linked List > Singly.cpp > ...

```cpp
// Delete from the end
void DeleteFromEnd() {
    if (head == nullptr) {
        cout << "List is empty" << endl;
        return;
    }
    if (head->getNext() == nullptr) {
        delete head;
        head = nullptr;
    }
    else {
        Node* temp = head;
        while (temp->getNext() != nullptr) {
            temp = temp->getNext();
        }
        temp->getPrev()->setNext(nullptr);
        delete temp;
    }
}

// Delete from the middle using a specific node (overload)
void DeleteFromMiddle(Node* pToBeDeleted) {
    if (pToBeDeleted == head) {
        DeleteFromBeginning();
    }
    else if (pToBeDeleted->getNext() == nullptr) {
        DeleteFromEnd();
    }
    else {
        pToBeDeleted->getPrev()->setNext(pToBeDeleted->getNext());
        pToBeDeleted->getNext()->setPrev(pToBeDeleted->getPrev());
        delete pToBeDeleted;
    }
}

// Delete the middle node using the slow and fast pointer technique
void DeleteFromMiddle() {
```



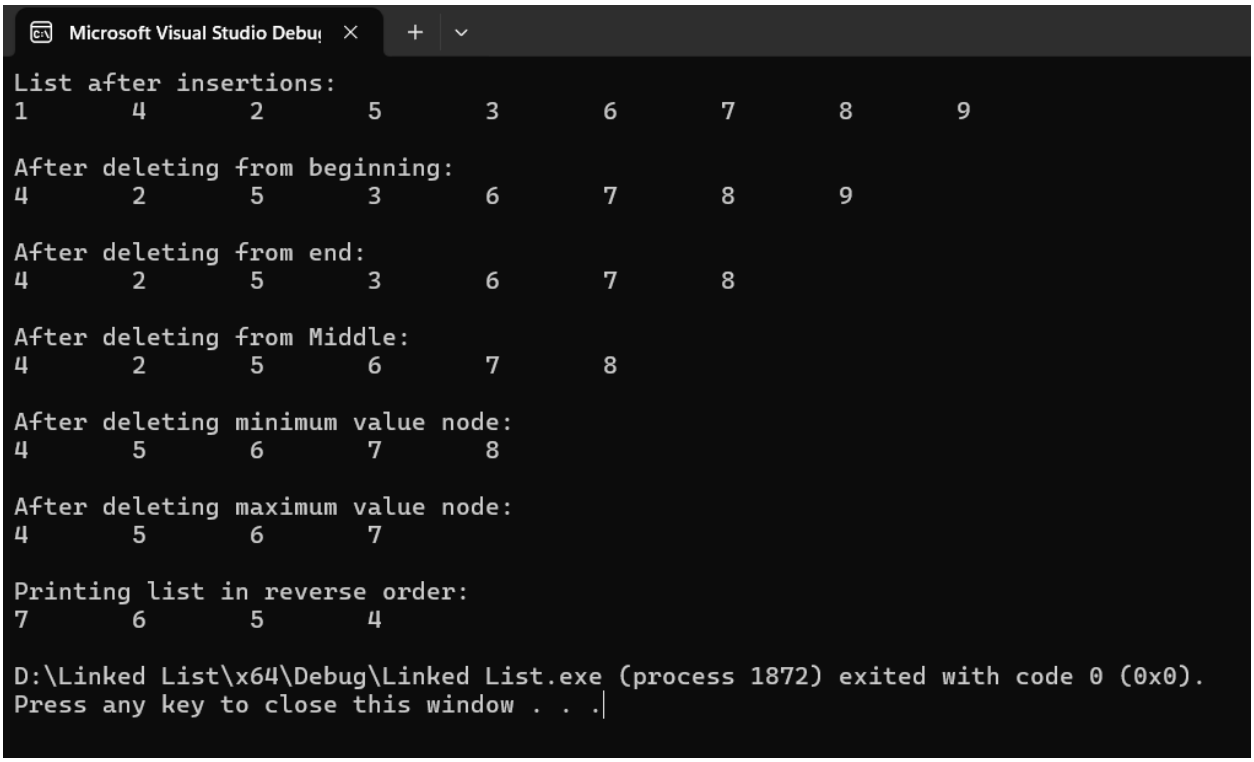
1st take getprev()



```
142     // Delete the middle node using the slow and fast pointer technique
143     void DeleteFromMiddle() {
144         if (head == nullptr || head->getNext() == nullptr) {
145             // If the list is empty or has only one element, delete the head
146             DeleteFromBeginning();
147             return;
148         }
149 
150         Node* slow = head;
151         Node* fast = head;
152 
153         // Traverse the list with fast and slow pointers
154         while (fast != nullptr && fast->getNext() != nullptr) {
155             slow = slow->getNext();
156             fast = fast->getNext()->getNext();
157         }
158 
159         // Now 'slow' is pointing to the middle node
160         DeleteFromMiddle(slow);
161     }
162 
163     // Delete the minimum value node
164     void DeleteMinimum() {
165         if (head == nullptr) return;
166 
167         Node* minNode = head;
168         Node* temp = head;
169         while (temp != nullptr) {
170             if (temp->getData() < minNode->getData()) {
171                 minNode = temp;
172             }
173             temp = temp->getNext();
174         }
175         DeleteFromMiddle(minNode);  // Correctly pass the minimum node
176     }
177 
```

```
179     void DeleteMaximum() {
180         if (head == nullptr) return;
181 
182         Node* maxNode = head;
183         Node* temp = head;
184         while (temp != nullptr) {
185             if (temp->getData() > maxNode->getData()) {
186                 maxNode = temp;
187             }
188             temp = temp->getNext();
189         }
190         DeleteFromMiddle(maxNode);  // Correctly pass the maximum node
191     }
192 
193     // Print the list
194     void PrintList() {
195         Node* temp = head;
196         while (temp != nullptr) {
197             cout << temp->getData() << "\t";
198             temp = temp->getNext();
199         }
200         cout << endl;
201     }
202 
203     // Recursive function to print list in reverse order
204     void PrintReverse(Node* root) {
205         if (root == nullptr) return;
206         PrintReverse(root->getNext());
207         cout << root->getData() << "\t";
208     }
209 
210     // Public function to call recursive print
211     void PrintInReverse() {
212         PrintReverse(head);
213         cout << endl;
214     }
215 };
```

```cpp
int main() {
    Node* a = new Node(1);
    Node* b = new Node(2);
    Node* c = new Node(3);
    Node* d = new Node(4);
    Node* e = new Node(5);
    Node* f = new Node(6);
    Node* k = new Node(7);
    Node* p = new Node(8);
    Node* z = new Node(9);

    DList* list = new DList();

    list->InsertBeginning(a);  // Insert first node at the beginning
    list->InsertEnd(b);        // Insert at the end
    list->InsertEnd(c);
    list->InsertMiddle(a, d);  // Insert in the middle
    list->InsertMiddle(b, e);
    list->InsertEnd(f);
    list->InsertEnd(k);
    list->InsertEnd(p);
    list->InsertEnd(z);

    cout << "List after insertions:" << endl;
    list->PrintList();

    list->DeleteFromBeginning();
    cout << "\nAfter deleting from beginning:" << endl;
    list->PrintList();

    list->DeleteFromEnd();
    cout << "\nAfter deleting from end:" << endl;
    list->PrintList();
    list->DeleteFromMiddle();
    cout << "\nAfter deleting from Middle:" << endl;
    list->PrintList();
    list->DeleteMinimum();
    cout << "\nAfter deleting minimum value node:" << endl;
    list->PrintList();
    list->DeleteMaximum();
    cout << "\nAfter deleting maximum value node:" << endl;
    list->PrintList();
    cout << "\nPrinting list in reverse order:" << endl;
    list->PrintInReverse();
    return 0;
```


Microsoft Visual Studio Debu
List after insertions:
1 4 2 5 3 6 7 8 9
After deleting from beginning:
4 2 5 3 6 7 8 9
After deleting from end:
4 2 5 3 6 7 8
After deleting from Middle:
4 2 5 6 7 8
After deleting minimum value node:
4 5 6 7 8
After deleting maximum value node:
4 5 6 7
Printing list in reverse order:
7 6 5 4
D:\Linked List\x64\Debug\Linked List.exe (process 1872) exited with code 0 (0x0).
Press any key to close this window . . .