Linked Lists

In this paper we'll discuss different strategies for managing linked lists. Singly or 1-way linked lists maintain a single forward pointer in each node that points to the next item in the list. Doubly or 2-way linked lists maintain two pointers in each node: a pointer to the next item, and a pointer to the previous item.

1-Way Linked Lists

Linked lists are a convenient way to store data for sequential access. Let's assume we want to store a list of integers. The following code illustrates a typical node in the list, and sets up a **head** pointer with a **NULL** value.

```
struct Node {
   Node *next;
   int value;
};

Node *head = NULL;
```

To add and remove items from the front of the list we'll define functions **push_front** and **pop_front**. Function **displayList** will output the contents of the entire list.

```
// add an item to front of list
void push_front(int value) {
   // construct node
   Node *p = new Node;
   p->value = value;

   // insert node
   p->next = head;
   head = p;
}

// remove an item from front of list
void pop_front() {
   Node *p = head;
   head = head->next;
   delete p;
}

// display entire list
void displayList() {
   Node *p;
   for (p = head; p; p = p->next)
       cout << p->value << endl;
}
```

To add and remove items from the back of the list we need to know the address of the last node on the list so we can update its next pointer. The following versions of **push_back** and **pop_back** chain through the list to add or remove items from the end of the list.

```
// add item to end of list
void push_back(int value) {
   Node *p, *last;

   // construct node
   p = new Node;
   p->value = value;
   p->next = NULL;

   // find last node
   for (last = head; last && last->next; last = last->next)
      ;

   // insert at end of list
   if (last)
      last->next = p;
   else
      head = p;      // list must be empty
   }

// remove item from end of list
void pop_back() {
   Node *p, *prev;

   prev = NULL;
   p = head;

   // chain until p points to last node
   // and prev points to next-to-last node
   while (p && p->next) {
      prev = p;
      p = p->next;
   }

   // adjust previous node
   if (prev)
      prev->next = NULL;
   else
      head = NULL;   // deleted first node

   delete p;
}
```

Practice problem #1

Given **t** is a pointer to node in the list and **s** is a pointer to a node not in the list.

1. Insert **s** after node **t**.
2. Insert **s** before node **t**.
3. Delete node **t**.

For faster access to the back of the list let's maintain a tail pointer that points to the last node on the list. First we'll fix **push_front** and **pop_front**, incorporating tail pointer logic.

```
Node *tail = NULL;   // points to last item on list
Node *head = NULL;   // points to 1st item on list

void push_front(int value) {
   // construct node
   Node *p = new Node;
   p->value = value;

   // insert node
   p->next = head;
   head = p;
   if (!tail) tail = p;
}

void pop_front() {
   Node *p = head;
   head = head->next;
   delete p;
   if (!head) tail = NULL;
}
```

Now we can add the improved version of **push_back** that uses the tail pointer to find the last node.

```
void push_back(int value) {
   // construct new node
   Node *p = new Node;
   p->value = value;
   p->next = NULL;

   // insert node
   if (tail)
      tail->next = p;
   else
      head = p;      // list must be empty
   tail = p;
}
```

To code **pop_back** we need a pointer to the next-to-last node so we can update its next pointer. Then, if we do another **pop_back**, we'll need a pointer to the node before that. This is a never-ending problem, so for this purpose the tail pointer won't suffice.

How would you output the list in reverse order? That is, display the last item, the next-to-last item, and so on. In summary, 1-way linked lists work well for adding and removing items at the front of the list. However, adding and removing items from the back of the list, or displaying the list in reverse order, can be an expensive operation. This is true even if a tail pointer is included in the design.

2-Way Linked Lists

There are several strategies you can use to implement 2-way linked lists. We'll explore two possibilities: NULL-terminated lists and circular lists with a sentinel node. The following structure will be used in our examples.

```
struct Node {
   Node *next;
   Node *prev;
   int value;
} Node;
```

NULL-Terminated Lists

For this method we'll maintain a **head** pointer that's initially NULL. The **next** pointers are forward pointers that point to the next node, and the **prev** pointers are backward pointers that point to the previous node. The last **next** pointer at the end of the foward chain is NULL, and the last **prev** pointer (1st node in list) is NULL.

```
Node *tail = NULL;
Node *head = NULL;

// add an item to front of list
void push_front(int value) {
   // construct node
   Node *p = new Node;
   p->value = value;

   // insert node
   p->prev = NULL;
   p->next = head;
   if (head)
      head->prev = p;
   else
      tail = p;      // inserting 1st node
   head = p;
}

// add an item to end of list
void push_back(int value) {
   // construct node
   Node *p = new Node;
   p->value = value;

   // insert node
   p->next = NULL;
   p->prev = tail;
   if (tail)
      tail->next = p;
   else
      head = p;      // inserting 1st node
   tail = p;
}
```

```
// remove an item from front of list
void pop_front() {
   // remove from list
   Node *p = head;
   head = head->next;
   delete p;

   // adjust pointers
   if (head)
      head->prev = NULL;
   else
      tail = NULL;      // removing 1st node
}

// remove an item from end of list
void pop_back() {
   Node *p = tail;
   tail = tail->prev;
   delete p;

   // adjust pointers
   if (tail)
      tail->next = NULL;
   else
      tail = NULL;      // removing 1st node
}

// display entire list
void displayList() {
   Node *p;
   for (p = head; p; p = p->next)
       cout << p->value << endl;
}
```

Using 2-way lists we no longer have to chain through the entire list to implement **pop_back**. How would you display the list in reverse order?

Practice Problem #2

Given **t** is a pointer to node in the list and **s** is a pointer to a node not in the list.

1. Insert **s** after node **t**.
2. Insert **s** before node **t**.
3. Delete node **t**.

Circular Lists with Sentinel Nodes

To avoid complications associated with head/tail pointers, 2-way lists are often implemented as circular lists with a sentinel node. When the list is empty the sentinel node simply points to itself. The following code illustrates logic using a sentinel node. Prior to using the data structure the user must call **initList** to allocate and initialize the sentinel node.

```
Node *head;

// allocate an initialize sentinel node
void initList() {
   head = new Node;
   head->next = head->prev = head;
}

// add an item to front of list
void push_front(int value) {
   // construct node
   Node *p = new Node;
   p->value = value;

   // insert node
   p->next = head->next;
   p->prev = head;
   p->next->prev = p;
   p->prev->next = p;
}


// add an item to end of list
void push_back(int value) {
   // construct node
   Node *p = new Node;
   p->value = value;

   // insert node
   p->next = head;
   p->prev = head->prev;
   p->next->prev = p;
   p->prev->next = p;
}
```

```
// remove an item from front of list
void pop_front() {
   Node *p = head->next;
   p->next->prev = p->prev;
   p->prev->next = p->next;
   delete p;
}

// remove an item from end of list
void pop_back() {
   Node *p = head->prev;
   p->next->prev = p->prev;
   p->prev->next = p->next;
   delete p;
}

// display entire list
void displayList() {
   Node *p;
   for (p = head->next; p != head; p = p->next)
       cout << p->value << endl;
}
```

Since the head is also a node, no special logic is required for edge cases. How would you display the list in reverse order?

Practice Problem #3

Given **t** is a pointer to node in the list and **s** is a pointer to a node not in the list.

1. Insert **s** after node **t**.
2. Insert **s** before node **t**.
3. Delete node **t**.