VE281

Data Structures and Algorithms

Operator Overloading and Linked List

Teaching Assistant

- Rao, Guoteng (饶国腾)
- Email: DSAAfall2012@gmail.com
- Cell phone: 15216708894

Review

- Operational Methods of Linear List
  - const member function: **size** and **query**
  - **insert** and **remove**
- Operator Overloading
  - Member function versus non-member function
  - Friend mechanism
- Overloading **Operator[]**
  - const version versus non-const version

Outline

- **Operator<<** for Linear List
- Basics of Linked List
- Linked List Traversal

Linear List

Overloading Output Operator <<

- We want to redefine the **operator<<** for the class linear list, so that it prints all the elements in the list in sequence.
- Convention of the IO library
  - The **operator<<** should take an **ostream&** as its first parameter and a reference to a const object of the class type as its second.
  - The **operator<<** should return a reference to its ostream parameter.

```
ostream &operator<<(ostream &os, const List &l) {
  ...
  return os;
}
```

Linear List

Overloading Output Operator <<

```
ostream &operator<<(ostream &os, const List &l) {
  ...
  return os;
}
```

- Why should **operator<<** return a reference to its ostream parameter?
  - Because **operator<<** can be chained together:

    ```
    cout << "hello " << "world!" << endl;
    ```

  - It is equivalent to

    ```
    cout << "hello ";
    cout << "world!";
    cout << endl;
    ```

Linear List

Overloading Output Operator <<

- **operator<<** must be a nonmember function!
  - The first operand is not of the class type.

- We can implement **operator<<** as follows

```
ostream &operator<<(ostream &os, const List &l) {
  for(int i = 0; i < l.size(); i++)
    os << l[i] << " ";
  return os;
}
```

Question: Which version of operator[] i

- Now we can write **cout << l << endl;**

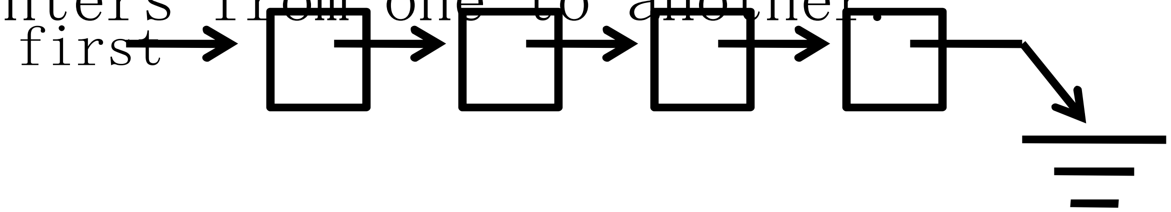
Outline

- `Operator<<` for Linear List
- Basics of Linked List
- Linked List Traversal

Linked List

Introduction

- Another foundational data structure.
- A linked structure is one with a series of zero or more data containers, connected by pointers from one to another.




```
class LinkedList {
  node *first;
 public:
   ...
};
```

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

Linked List

Basic Methods

```
class LinkedList {
  node *first;
public:
  bool isEmpty();
  void insertFirst(int v);
  int removeFirst();
  LinkedList();              //default ctor
  LinkedList(const LinkedList& l);//copy ctor
  ~LinkedList();                     //dtor
  LinkedList &operator=(const LinkedList &l);
};
```

Linked List

isEmpty()

- A list is empty if there is no node in the list, or first is NULL:

```
bool LinkedList::isEmpty() {
   return !first;
}
```

Linked List

insertFirst()

- To implement **insertFirst**, the first thing we need to do is to create a new node to hold the new "first" element.
- Next, we need to establish the invariants on the new node.
- This means setting the value field to v, and the next field to the "rest of the list" - this is precisely the start of the current list "**first**".

```
void LinkedList::insertFirst(int v) {
  node *np = new node;
  np->value = v;
  np->next = first;
  ...
}
```

Linked Lists

insertFirst()

- Finally, we need to reestablish the representation invariant:
  first currently points to the **second** node in the list, and must point to the first node of the new list instead:

```
void LinkedList::insertFirst(int v){
  node *np = new node;
  np->value = v;
  np->next = first;
  first = np;
}
```

Linked Lists

removeFirst()

- Specification of **removeFirst()** is

```
int removeFirst();
  // MODIFIES: this
  // EFFECTS: if list is empty, throw
  // listIsEmpty. Otherwise, remove the
  // first node and return the value
  // in that node
```

Linked Lists

removeFirst()

- Removal is a bit trickier since there are lots of things we need to accomplish, and they have to happen in precisely **the right order**.

- We should advance first:

  ```
  first = first->next;
  ```

- We should al Is this possible? he “old” **first** node.

- We need to introduce a local variable to remember the “old” **first** node, which we will call the **victim**

Linked Lists

removeFirst()

- We first create the **victim**.
- We then skip the first node.
- Finally, we delete the node **victim**.

```
int LinkedList::removeFirst() {
    node *victim = first;
    ...
    first = victim->next;
    ...
    delete victim;
    ...
}
```

Linked Lists

removeFirst()

- Two more things to do:
  - Return the value that was stored in the node.
  - Cope with an empty list, and throw an exception if we have one.

```
int LinkedList::removeFirst() {
  node *victim = first;
  int result;
  if (isEmpty()) throw listIsEmpty();
  first = victim->next;
  result = victim->value;
  delete victim;
  return result;
}
```

Exercise

- What is the complexity of the following methods?

```
bool isEmpty();
void insertFirst(int v);
int removeFirst();
```

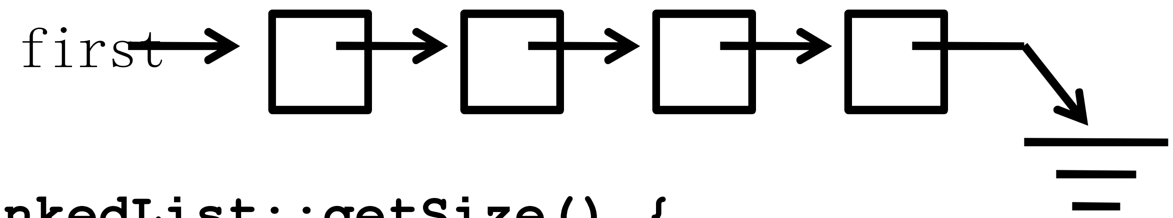
Outline

- **Operator<<** for Linear List
- Basics of Linked List
- Linked List Traversal

Linked List

Get the size: getSize()

- How to get the size of a linked list?




```
int LinkedList::getSize() {
  int count = 0;
  node *current = first;
  while(current){
    count++;
    current = current->next;
  }
  return count;
}
```

Traverse through the list.

Linked List

Get the size: getSize()

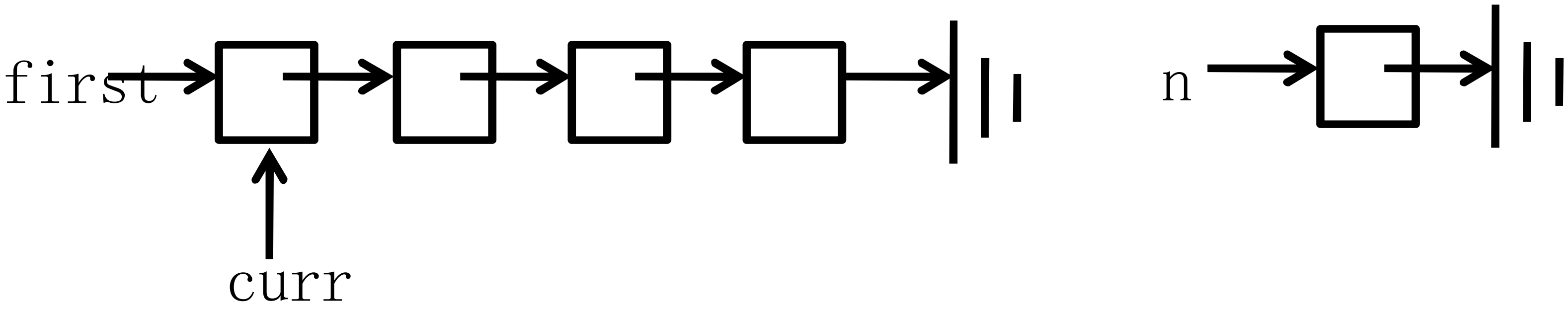
- What is the complexity? Suppose the number of nodes is *n*.

```
int LinkedList::getSize() {
  int count = 0;
  node *current = first;
  while(current){
    count++;
    current = current->next;
  }
  return count;
}
```

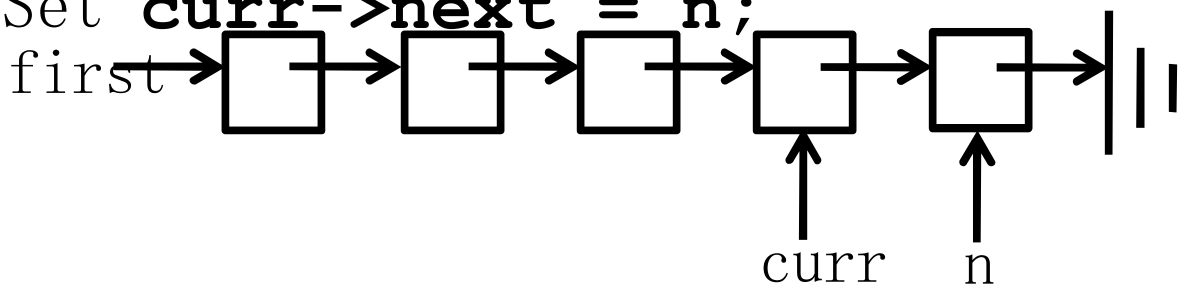
Linked List

Append a Node

```
void appendNode(node *n)
// EFFECTS: append the node at the end of
// the current linked list
```



- Search down the list until **curr->next == NULL**
- Set **curr->next = n;**



Linked List

Append a Node

- Implementation of **appendNode(node *n)**

```
void LinkedList::appendNode(node *n){
  if(!first) {
    first = n;
    return;
  }
  node *curr = first;
  while (curr->next != NULL)
    curr = curr->next;
  curr->next = n;
}
```

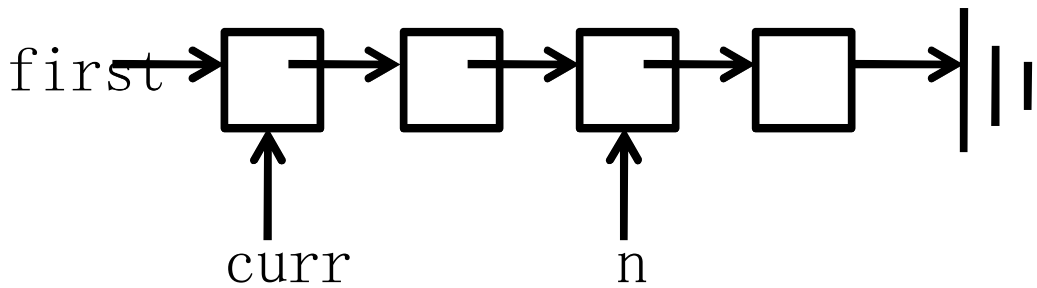
Linked List

Remove a Node

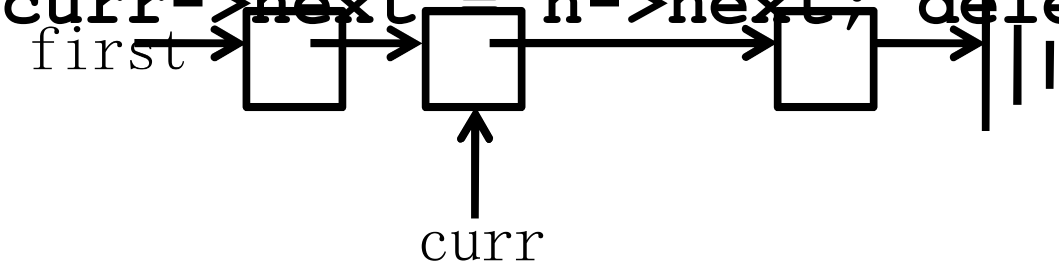
`void removeNode(node *n)`

`// REQUIRES: n is in the list`

`// EFFECTS: remove the node n`

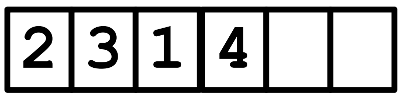


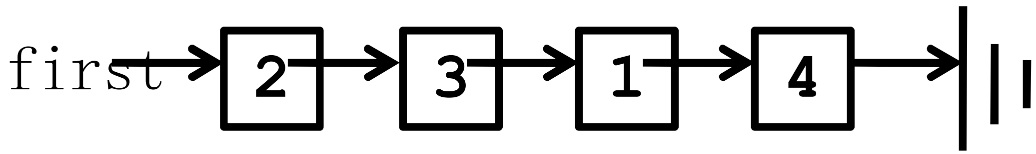
- Search down the list until **`curr->next == n`**
- Set **`curr->next = n->next; delete n;`**



Arrays versus Linked Lists

Worst Case Time Complexity

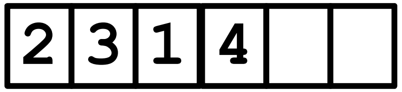


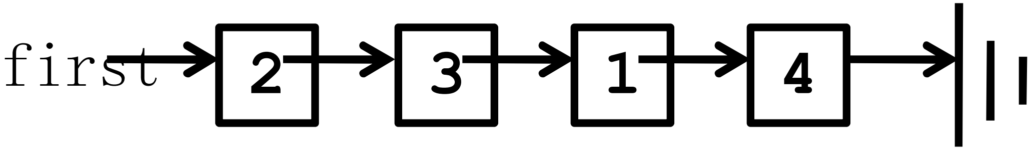


	Array	Linked List
Random access	$O(1)$ time	$O(n)$ time
insertFirst	$O(n)$ time	$O(1)$ time
removeFirst	$O(n)$ time	$O(1)$ time
appendNode	$O(1)$ time	$O(n)$ time
removeNode	$O(n)$ time	$O(n)$ time

Arrays versus Linked Lists

Memory Requirement





	Array	Linked List
Bookkeeping	Pointer to the beginning Size or pointer to the "end"	Pointer to the first no "next" pointer in eac
Memory	Free in $O(1)$ time Wastes memory if size is too large. Requires reallocation if too small.	Free in $O(n)$ time Allocates memory as needed. Allocation and de-allocation costly.