DATA STRUCTURES AND ALGORITHMS

LECTURE 03

LINKED LIST

IMRAN IHSAN
ASSISTANT PROFESSOR
AIR UNIVERSITY, ISLAMABAD

LINKED LISTS

DEFINITION

- A linked list is a data structure where each object is stored in a node
- As well as storing data, the node must also contains a reference/pointer to the node containing the next item of data
- We must dynamically create the nodes in a linked list
- Thus, because new returns a pointer, the logical manner in which to track a linked lists is through a pointer
- A Node class must store the data and a reference to the next node (also a pointer)

LINKED LISTS
NODE CLASS

The node must store data and a pointer:

```
class Node {
    private:
        int element;
        Node *next_node;
    public:
        Node( int = 0, Node * = nullptr );

        int retrieve() const;
        Node *next() const;
};
```


LINKED LISTS
NODE CONSTRUCTOR

The constructor assigns the two member variables based on the arguments

```
Node::Node( int e, Node *n ):
element( e ),
next_node( n ) {
    // empty constructor
}
```

The default values are given in the class definition:

```
class Node {
    private:
        int element;
        Node *next_node;
    public:
        Node( int = 0, Node * = nullptr );
        int retrieve() const;
        Node *next() const;
};
```

LINKED LISTS
ACCESSORS

The two member functions are accessors which simply return the **element** and the **next_node** member variables, respectively

```
int Node::retrieve() const {
     return element;
}

Node *Node::next() const {
    return next_node;
}
```

- *Member functions that do not change the object acted upon are variously called accessors, readonly functions, inspectors, and, when it involves simply returning a member variable, getters*

LINKED LISTS
ACCESSORS

- In C++, a member function cannot have the same name as a member variable
  - Possible solutions:

	Member Variables	**Member Functions**
Vary capitalization	next_node	Next_node() or NextNode()
Prefix with "get"	next_node	get_next_node() / getNextNode()
Use an underscore	next_node_	next_node()
Different names	next_node	next()

- Always use the naming convention and coding styles used by your employer— even if you disagree with them Consistency aids in maintenance

LINKED LIST
CLASS

- Because each node in a linked lists refers to the next, the linked list class need only link to the first node in the list
- The linked list class requires member variable: a pointer to a node

```
class List {
    private:
        Node *list_head;
    // ...
};
```


LINKED LIST
STRUCTURE

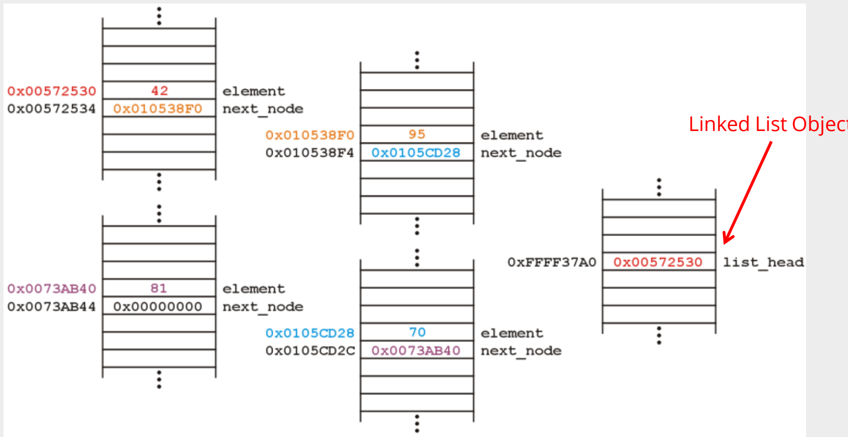
- To begin, let us look at the internal representation of a linked list
- Suppose we want a linked list to store the values

42 95 70 81

- in this order
- A linked list uses linked allocation, and therefore each node may appear anywhere in memory
- Also the memory required for each node equals the memory required by the member variables
    - 4 bytes for the linked list (a pointer)
    - 8 bytes for each node (an int and a pointer)
    - We are assuming a 32-bit machine

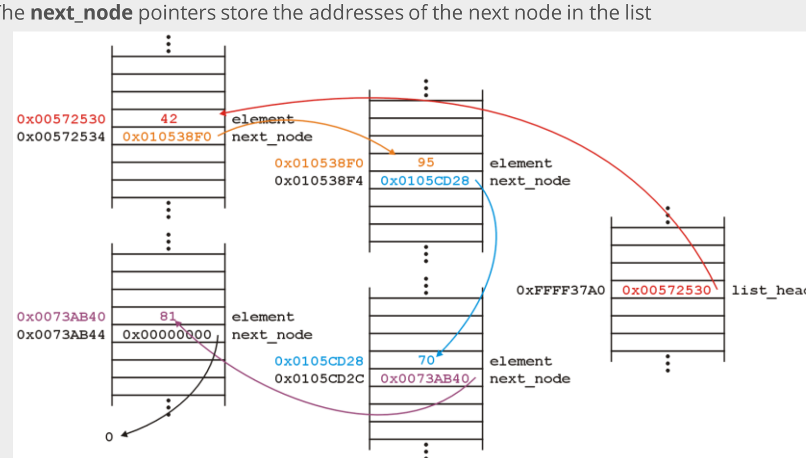
LINKED LIST
STRUCTURE

- Such a list could occupy memory as follows:



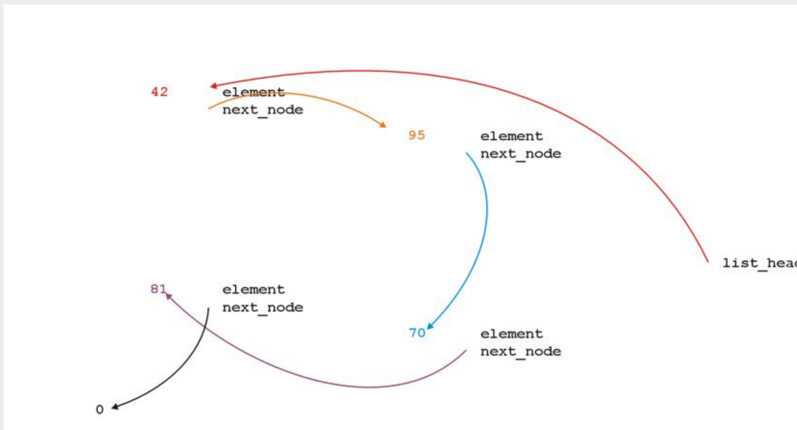
LINKED LIST
STRUCTURE

- The **next_node** pointers store the addresses of the next node in the list



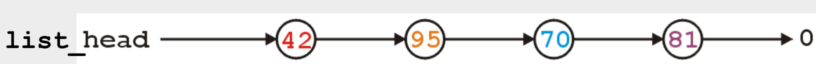
LINKED LIST
STRUCTURE

- Because the addresses are arbitrary, we can remove that information:



LINKED LIST
STRUCTURE

- We will clean up the representation as follows:



- We do not specify the addresses because they are arbitrary and:
  - The contents of the circle is the element
  - The next_node pointer is represented by an arrow

LINKED LIST
OPERATIONS

- First, we want to create a linked list
- We also want to be able to:
  - insert into,
  - access, and
  - erase from
- the elements stored in the linked list

LINKED LIST
OPERATIONS

- We can do them with the following operations:
- Adding, retrieving, or removing the value at the front of the linked list

```
void push_front( int );
int front() const;
void pop_front();
```

- We may also want to access the head of the linked list

```
Node *head() const;
```

Member functions that may change the object acted upon are variously called mutators, modifiers, and, when it involves changing a single member variable, setters

LINKED LIST
OPERATIONS

- All these operations relate to the first node of the linked list
- We may want to perform operations on an arbitrary node of the linked list, for example:
  - Find the number of instances of an integer in the list:

```
int count( int ) const;
```

  - Remove all instances of an integer from the list:

```
int erase( int );
```


LINKED LIST
ADDITIONAL FUNCTIONS

- Is the linked list empty?

```
bool empty() const;
```

- How many objects are in the list?

```
int size() const;
```

- The list is empty when the `list_head` pointer is set to `nullptr`

LINKED LIST
SIMPLE BUT INCOMPLETE CLASS

```
class List {
    private:
        Node *list_head;

    public:
        List();
        // Accessors
        bool empty() const;
        int size() const;
        int front() const;
        Node *head() const;
        int count( int ) const;

        // Mutators
        void push_front( int );
        int pop_front();
        int erase( int );
};
```


LINKED LIST
CONSTRUCTOR

- The constructor initializes the linked list
- We do not count how may objects are in this list, thus:
  - we must rely on the last pointer in the linked list to point to a special value
  - in C++, that standard value is `nullptr`
- Thus, in the constructor, we assign `list_head` the value `nullptr`

```
List::List():list_head( nullptr ) {
    // empty constructor
}
```

- We will always ensure that when a linked list is empty, the list head is assigned `nullptr`

LINKED LIST
ALLOCATION

- The constructor is called whenever an object is created, either:
- Statically
  - The statement `List ls;` defines `ls` to be a linked list and the compiler deals with memory allocation

- Dynamically
  - The statement

```
List *pls = new List();
```

  - requests sufficient memory from the OS to store an instance of the class

  - In both cases, the memory is allocated and then the constructor is called

LINKED LIST
STATIC ALLOCATION

```
int f() {
      List ls;   // ls is declared as a local variable on the stack

      ls.push_front( 3 );
      cout << ls.front() << endl;

      // The return value is evaluated
      // The compiler then calls the destructor for local variables
      // The memory allocated for 'ls' is deallocated

      return 0;
   }
```

BOOL EMPTY() CONST
LINKED LIST MEMBER FUNCTION

```
bool List::empty() const {
    if ( list_head == nullptr ) {
        return true;
    } else {
        return false;
    }
}
```

Better yet:

```
bool List::empty() const {
    return ( list_head == nullptr );
}
```


NODE *HEAD() CONST
LINKED LIST MEMBER FUNCTION

The member function Node *head() const is easy enough to implement:

```
Node *List::head() const {
    return list_head;
}
```

This will always work: if the list is empty, it will return `nullptr`

INT FRONT() CONST

LINKED LIST MEMBER FUNCTION

- To get the first element in the linked list, we must access the node to which the list_head is pointing
- Because we have a pointer, we must use the → operator to call the member function:

```
int List::front() const {
    return head()->retrieve();
}
```

- The member function int front() const requires some additional consideration, however:
  - what if the list is empty?
- If we tried to access a member function of a pointer set to nullptr, we would access restricted memory
- The operating system would terminate the running program

INT FRONT() CONST

LINKED LIST MEMBER FUNCTION

- Instead, we can use an exception handling mechanism where we thrown an exception
- We define a class

```
class underflow {
    // emtpy
};
```

- and then we throw an instance of this class:

```
throw underflow();
```

- Thus, the full function is

```
int List::front() const {
    if ( empty() ) {
        throw underflow();
    }

    return head()->retrieve();
}
```

INT FRONT() CONST

LINKED LIST MEMBER FUNCTION

- Why is `emtpy()` better than

```
int List::front() const {
    if ( list_head == nullptr ) {
        throw underflow();
    }

    return list_head->element;
}
```

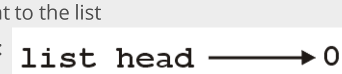
?

- Two benefits:
  - More readable
  - If the implementation changes we do nothing

VOID PUSH_FRONT(INT)

LINKED LIST MEMBER FUNCTION

- Next, let us add an element to the list
- If it is empty, we start with:



- and, if we try to add 81, we should end up with:

- To visualize what we must do:
  - We must create a new node which:
    - stores the value 81, and
    - is pointing to `0`
- We must then assign its address to `list_head`
- We can do this as follows:

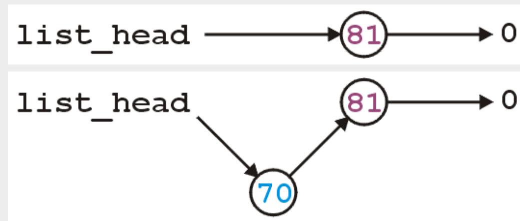
```
list_head = new Node( 81, nullptr );
```

- We could also use the default value...

VOID PUSH_FRONT(INT)

LINKED LIST MEMBER FUNCTION

- Suppose however, we already have a non-empty list
- Adding **70**, we want:



- To achieve this, we must we must create a new node which:
  - stores the value **70**, and
  - is pointing to the current list head
- we must then assign its address to `list_head`
- We can do this as follows:

```
list_head = new Node( 70, list_head );
```


VOID PUSH_FRONT(INT)

LINKED LIST MEMBER FUNCTION

- Thus, our implementation could be:

```
void List::push_front( int n ) {
    if ( empty() ) {
        list_head = new Node( n, nullptr );
    } else {
        list_head = new Node( n, head() );
    }
}
```

- We could, however, note that when the list is empty, list_head == 0, thus we could shorten this to:

```
void List::push_front( int n ) {
    list_head = new Node( n, list_head );
}
```

VOID PUSH_FRONT(INT)

LINKED LIST MEMBER FUNCTION

- Are we allowed to do this?

```
void List::push_front( int n ) {
    list_head = new Node( n, head() );
}
```

- **Yes:** The right-hand side of an assignment is evaluated first
  The original value of list_head is accessed first before the function call is made

VOID PUSH_FRONT(INT)

LINKED LIST MEMBER FUNCTION

- Question: does this work?

```
void List::push_front( int n ) {
    Node new_node( n, head() );
    list_head = &new_node;
}
```

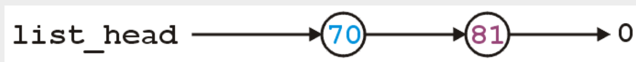
- Why or why not? What happens to new_node?
- How does this differ from

```
void List::push_front( int n ) {
    Node *new_node = new Node( n, head() );
    list_head = new_node;
}
```

VOID POP_FRONT(INT)

LINKED LIST MEMBER FUNCTION

- Erasing from the front of a linked list is even easier:
  - We assign the list head to the next pointer of the first node
- Graphically, given:



- we want:

VOID POP_FRONT(INT)

LINKED LIST MEMBER FUNCTION

- Easy Enough

```
int List::pop_front() {
    int e = front();
    list_head = head()->next();
    return e;
}
```

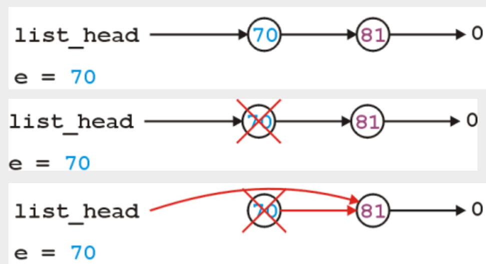
- Unfortunately, we have some problems:
  - The list may be empty
  - We still have the memory allocated for the node containing `70`

VOID POP_FRONT(INT)

LINKED LIST MEMBER FUNCTION

- Does this work?

```
int List::pop_front() {
    if ( empty() ) {
        throw underflow();
    }

    int e = front();

    delete head();

    list_head = head()->next();

    return e;
}
```



VOID POP_FRONT(INT)

LINKED LIST MEMBER FUNCTION

- The problem is, we are accessing a node which we have just deleted
- Unfortunately, this will work more than 99% of the time:
- The running program (process) may still own the memory
  - Once in a while it will fail ...
  - ... and it will be almost impossible to debug

VOID POP_FRONT(INT)

LINKED LIST MEMBER FUNCTION

- The correct implementation assigns a temporary pointer to point to the node being deleted:

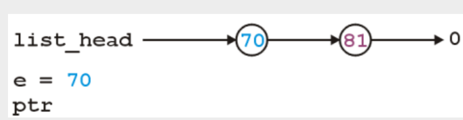
```
int List::pop_front() {
     if ( empty() ) {
          throw underflow();
     }

    int e = front();


    Node *ptr = list_head;
    list_head = list_head->next();
    delete ptr;
    return e;
}
```


VOID POP_FRONT(INT)

LINKED LIST MEMBER FUNCTION

- The correct implementation assigns a temporary pointer to point to the node being deleted:

```
int List::pop_front() {
     if ( empty() ) {
          throw underflow();
     }

    int e = front();


    Node *ptr = list_head;
    list_head = list_head->next();
    delete ptr;
    return e;
}
```

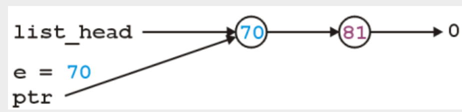


VOID POP_FRONT(INT)

LINKED LIST MEMBER FUNCTION

- The correct implementation assigns a temporary pointer to point to the node being deleted:

```
int List::pop_front() {
     if ( empty() ) {
          throw underflow();
     }

    int e = front();
    Node *ptr = list_head;


    list_head = list_head->next();
    delete ptr;
    return e;
}
```

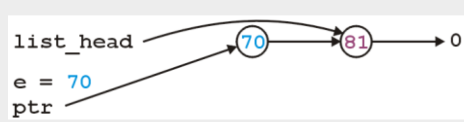


VOID POP_FRONT(INT)

LINKED LIST MEMBER FUNCTION

- The correct implementation assigns a temporary pointer to point to the node being deleted:

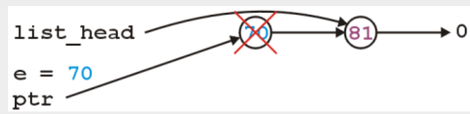
```
int List::pop_front() {
     if ( empty() ) {
          throw underflow();
     }

    int e = front();
    Node *ptr = list_head;
    list_head = list_head->next();


    delete ptr;
    return e;
}
```



VOID POP_FRONT(INT)
LINKED LIST MEMBER FUNCTION

- The correct implementation assigns a temporary pointer to point to the node being deleted:

```
int List::pop_front() {
     if ( empty() ) {
          throw underflow();
     }

    int e = front();
    Node *ptr = list_head;
    list_head = list_head->next();
    delete ptr;


    return e;
}
```



STEPPING THROUGH
A LINKED LIST

- The next step is to look at member functions which potentially require us to step through the entire list:

```
int size() const;
int count( int ) const;
int erase( int );
```

- The second counts the number of instances of an integer, and the last removes the nodes containing that integer
- The process of stepping through a linked list can be thought of as being analogous to a for-loop:
  - We initialize a temporary pointer with the list head
  - We continue iterating until the pointer equals nullptr
  - With each step, we set the pointer to point to the next object

STEPPING THROUGH
A LINKED LIST

- Thus we have:

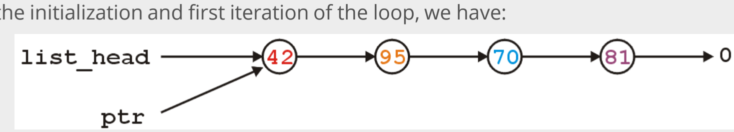
```
for ( Node *ptr = head(); ptr != nullptr; ptr = ptr->next() ) {
        // do something
        // use ptr->fn() to call member functions
        // use ptr->var to assign/access member variables
  }
```

- Analogously

```
for ( Node *ptr = head(); ptr != nullptr; ptr = ptr->next() )
for ( int i = 0;             i != N;         ++i              )
```

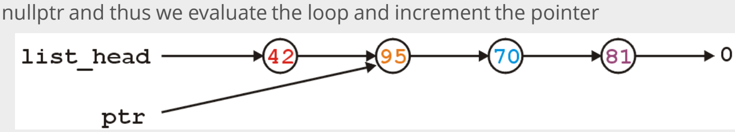

STEPPING THROUGH
A LINKED LIST

- With the initialization and first iteration of the loop, we have:



- ptr != nullptr and thus we evaluate the body of the loop and then set ptr to the next pointer of the node it is pointing to

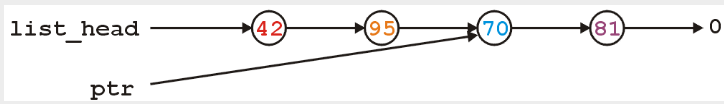
- ptr != nullptr and thus we evaluate the loop and increment the pointer



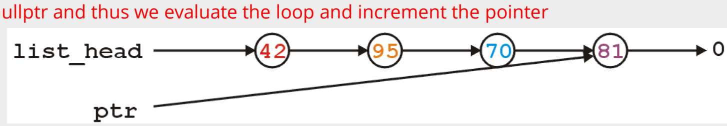
- In the loop, we can access the value being pointed to by using ptr->retrieve()

STEPPING THROUGH
A LINKED LIST

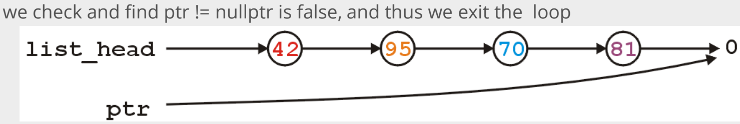
- ptr != nullptr and thus we evaluate the loop and increment the pointer



- Also, in the loop, we can access the next node in the list by using ptr->next()
- ptr != nullptr and thus we evaluate the loop and increment the pointer



- This last increment causes ptr == nullptr
- Here, we check and find ptr != nullptr is false, and thus we exit the loop



- Because the variable ptr was declared inside the loop, we can no longer access it

INT COUNT(INT) CONST
LINKED LIST MEMBER FUNCTION

- To implement int count(int) const, we simply check if the argument matches the element with each step
  - Each time we find a match, we increment the count
  - When the loop is finished, we return the count
  - The size function is simplification of count

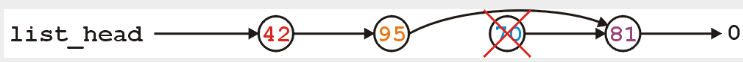
```
int List::count( int n ) const {
    int node_count = 0;

    for ( Node *ptr = list(); ptr != nullptr; ptr = ptr->next() ) {
        if ( ptr->retrieve() == n ) {
            ++node_count;
        }
    }

    return node_count;
}
```

INT ERASE (INT)

LINKED LIST MEMBER FUNCTION

- To remove an arbitrary element, i.e., to implement int erase(int), we must update the previous node
- For example, given

- if we delete 70, we want to end up with



ACCESS PRIVATE MEMBER VARIABLES

IN A LINKED LIST

- Notice that the erase function must modify the member variables of the node prior to the node being removed
- Thus, it must have access to the member variable next_node
- We could supply the member function

```
void set_next( Node * );
```

- however, this would be globally accessible
- Possible solutions:
  - Friends
  - Nested classes
  - Inner classes

C++
FRIENDS

- In C++, you explicitly break encapsulation by declaring the class List to be a ***friend*** of the class Node:

```
class Node {
    Node *next() const;
    // ... declaration ...
    friend class List;
};
```

- Now, inside erase (a member function of List), you can modify all the member variables of any instance of the Node class

INT ERASE (INT)
LINKED LIST MEMBER FUNCTION

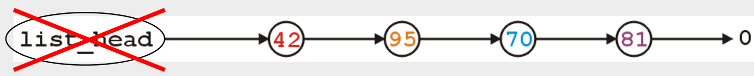
- For example, the erase member function could be implemented using the following code

```
int List::erase( int n ) {
    int node_count = 0;
    // ...

    for ( Node *ptr = head(); ptr != nullptr; ptr = ptr->next() ) {
        // ...

        if ( some condition ) {
            ptr->next_node = ptr->next()->next();
            // ...
            ++node_count;
        }
    }

    return node_count;
}
```

DESTRUCTOR

LINKED LIST

- We dynamically allocated memory each time we added a new **`int`** into this list
- Suppose we delete a list before we remove everything from it
  - This would leave the memory allocated with no reference to it



- Thus we need

```
class List {
    private:
        Node *list_head;
    public:
        List();
        ~List();
        // ...etc...
};
```


DESTRUCTOR

LINKED LIST

- The destructor has to delete any memory which had been allocated but has not yet been deallocated
- This is straight-forward enough:

```
while ( !empty() ) {
    pop_front();
}
```

COPY CONSTRUCTOR

- If such a function is defined, every time an instance is passed by value, the copy constructor is called to make that copy
- Additionally, you can use the copy constructor as follows:

```
List ls1;
ls1.push_front( 4 );
ls1.push_front( 2 );

List ls2( ls1 );  // make a copy of ls1
```

- When an object is returned by value, again, the copy constructor is called to make a copy of the returned value

ASSIGNMENT

- Suppose you have linked lists

```
List lst1, lst2;

lst1.push_front( 35 );
lst1.push_front( 18 );
lst2.push_front( 94 );
lst2.push_front( 72 );
```

LINKED LIST
COMPLETE CLASS

```
class List {
    private:
        Node *list_head;
        void swap( List & );

    public:
        // Constructors and destructors
        List();
        List( List const & );
        List( List && );
        ~List();

        // Assignment operators
        List &operator = ( List const & );
        List &operator = ( List && );

        // Accessors
        bool empty() const;
        int size() const;
        int front() const;
        Node *head() const;
        int count( int ) const;

        // Mutators
        void push_front( int );
        int pop_front();
        int erase( int );
};
```


REFERENCE
FOR THIS LECTURE

Donald E. Knuth, The Art of Computer Programming, Volume 3:
Sorting and Searching, 2nd Ed., Addison Wesley, 1998, §5.4, pp.248-379.

Wikipedia, https://en.wikipedia.org/wiki/Linked_list

http://stackoverflow.com/error?aspxerrorpath=/questions/8848363/rvalue-reference-with-assignement-operator