CS101 Algorithms and Data Structures

Array and Linked List

Textbook Ch 10.2

Outline

- List ADT
- Array
- Linked list
- Doubly linked list
- Node-based storage with arrays

Representation of polynomial coefficients a_n

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n$$

```
double fpoly1 ( int n, double a[ ], double x )
{ int i;
  double p = a[0];
  for (i = 1; i <=n; i++)
       p += (a[i]) * pow( x, i));
  return p;
}
```

- **Method 1: array**

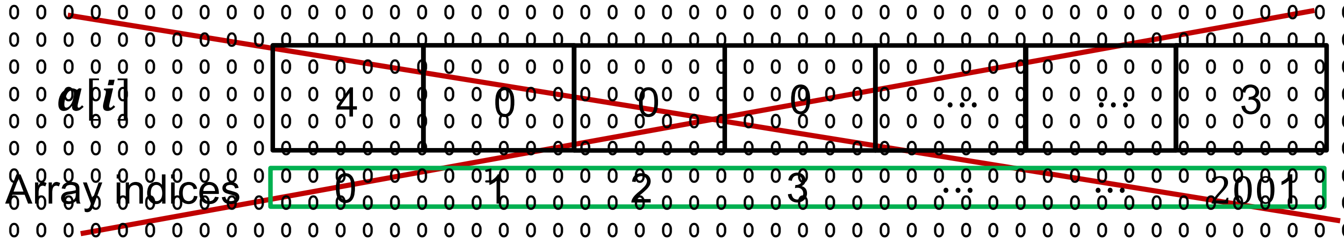
$$f(x) = 4x^5 - 3x^2 + 1$$

$a[i]$	1	0	-3	0	0	4	⋯
Array indices	0	1	2	3	4	5	⋯

Problem?

Discussion 1

How to present coefficients for $f(x) = 4 + 3x^{2001}$?



Method 2: structure array

- For each non-zero term, need to know two components: the coefficient $\boldsymbol{a_i}$, the index no. $\boldsymbol{i}$.
- We can use a structure array $(\boldsymbol{a_i}, \boldsymbol{i})$. (x, y)
- Ex:

$$P_1(x) = 3x^{100} + 10x^{50} + 15 \quad \& \quad P_2(x) = 4x^{100} + 30x^{60} + 5$$

$\boldsymbol{a}[\boldsymbol{i}]$	3	10	15	
Expon index i	100	50	0	
Array indices	0	1	2	$\cdots$

$\boldsymbol{a}[\boldsymbol{i}]$	4	30	5	
Expon index i	100	60	0	
Array indices	0	1	2	$\cdots$

Store the coefficients in descent order of exponential index.

Addition of Two Polynomials?

$$P_1(x) = 3x^{100} + 10x^{50} + 15 \quad \& \quad P_2(x) = 4x^{100} + 30x^{60} + 5$$

$\boldsymbol{a}[\boldsymbol{i}]$	3	10	15		4	30	5	
Expon index i	100	50	0		100	60	0	
Array indices	0	1	2	$\cdots$	0	1	2	$\cdots$

Addition of Two Polynomials?

$P_1(x) = 3x^{100} + 10x^{50} + 15$ & $P_2(x) = 4x^{100} + 30x^{60} + 5$

$\boldsymbol{a[i]}$	3	10	15		4	30	5	
Expon index i	100	50	0		100	60	0	
Array indices	0	1	2	⋯	0	1	2	⋯

Addition of Two Polynomials?

$P_1(x) = 3x^{100} + 10x^{50} + 15$ & $P_2(x) = 4x^{100} + 30x^{60} + 5$

$\boldsymbol{a[i]}$	3	10	15	
Expon index i	100	50	0	
Array indices	0	1	2	⋯

4	30	5	
100	60	0	
0	1	2	⋯

$\boldsymbol{a[i]}$								
Expon index i								
Array indices	0	1	2	3	4	5	6	⋯

Addition of Two Polynomials?

$P_1(x) = 3x^{100} + 10x^{50} + 15$ & $P_2(x) = 4x^{100} + 30x^{60} + 5$

$\boldsymbol{a}[\boldsymbol{i}]$	3	10	15	
Expon index i	100	50	0	
Array indices	0	1	2	⋯

$\boldsymbol{a}[\boldsymbol{i}]$	4	30	5	
Expon index i	100	60	0	
Array indices	0	1	2	⋯

$\boldsymbol{a}[\boldsymbol{i}]$	7							
Expon index i	100							
Array indices	0	1	2	3	4	5	6	⋯

Addition of Two Polynomials?

$$P_1(x) = 3x^{100} + 10x^{50} + 15 \quad \& \quad P_2(x) = 4x^{100} + 30x^{60} + 5$$

$\boldsymbol{a[i]}$	3	10	15	
Expon index i	100	50	0	
Array indices	0	1	2	⋯

$\boldsymbol{a[i]}$	4	30	5	
Expon index i	100	60	0	
Array indices	0	1	2	⋯

$\boldsymbol{a[i]}$	7							
Expon index i	100							
Array indices	0	1	2	3	4	5	6	⋯

Addition of Two Polynomials?

$$P_1(x) = 3x^{100} + 10x^{50} + 15 \quad \& \quad P_2(x) = 4x^{100} + 30x^{60} + 5$$

$\boldsymbol{a[i]}$	3	10	15	
Expon index i	100	50	0	
Array indices	0	1	2	⋯

4	30	5	
100	60	0	
0	1	2	⋯

$\boldsymbol{a[i]}$	7	30						
Expon index i	100	60						
Array indices	0	1	2	3	4	5	6	⋯

Addition of Two Polynomials?

$$P_1(x) = 3x^{100} + 10x^{50} + 15 \quad \& \quad P_2(x) = 4x^{100} + 30x^{60} + 5$$

$\boldsymbol{a}[\boldsymbol{i}]$	3	10	15	
Expon index i	100	50	0	
Array indices	0	1	2	⋯

$\boldsymbol{a}[\boldsymbol{i}]$	4	30	5	
Expon index i	100	60	0	
Array indices	0	1	2	⋯

$\boldsymbol{a}[\boldsymbol{i}]$	7	30						
Expon index i	100	60						
Array indices	0	1	2	3	4	5	6	⋯

Addition of Two Polynomials?

$$P_1(x) = 3x^{100} + 10x^{50} + 15 \quad \& \quad P_2(x) = 4x^{100} + 30x^{60} + 5$$

$\boldsymbol{a}[\boldsymbol{i}]$	3	10	15	
Expon index i	100	50	0	
Array indices	0	1	2	…

$\boldsymbol{a}[\boldsymbol{i}]$	4	30	5	
Expon index i	100	60	0	
Array indices	0	1	2	…

$\boldsymbol{a}[\boldsymbol{i}]$	7	30	10					
Expon index i	100	60	50					
Array indices	0	1	2	3	4	5	6	…

Addition of Two Polynomials?

$$P_1(x) = 3x^{100} + 10x^{50} + 15 \quad \& \quad P_2(x) = 4x^{100} + 30x^{60} + 5$$

$\boldsymbol{a}[\boldsymbol{i}]$	3	10	15	
Expon index i	100	50	0	
Array indices	0	1	2	···

4	30	5	
100	60	0	
0	1	2	···

$\boldsymbol{a}[\boldsymbol{i}]$	7	30	10					
Expon index i	100	60	50					
Array indices	0	1	2	3	4	5	6	···

Addition of Two Polynomials?

$$P_1(x) = 3x^{100} + 10x^{50} + 15 \quad \& \quad P_2(x) = 4x^{100} + 30x^{60} + 5$$

$\boldsymbol{a}[\boldsymbol{i}]$	3	10	15	
Expon index i	100	50	0	
Array indices	0	1	2	⋯

$\boldsymbol{a}[\boldsymbol{i}]$	4	30	5	
Expon index i	100	60	0	
Array indices	0	1	2	⋯

$\boldsymbol{a}[\boldsymbol{i}]$	7	30	10	20				
Expon index i	100	60	50	0				
Array indices	0	1	2	3	4	5	6	⋯

Addition of Two Polynomials?

$P_1(x) = 3x^{100} + 10x^{50} + 15$ & $P_2(x) = 4x^{100} + 30x^{60} + 5$

$\boldsymbol{a}[\boldsymbol{i}]$	3	10	15	
Expon index i	100	50	0	
Array indices	0	1	2	⋯

$\boldsymbol{a}[\boldsymbol{i}]$	4	30	5	
Expon index i	100	60	0	
Array indices	0	1	2	⋯

$\boldsymbol{a}[\boldsymbol{i}]$	7	30	10	20				
Expon index i	100	60	50	0				
Array indices	0	1	2	3	4	5	6	⋯

$$P_3(x) = P_1(x) + P_2(x) = 7x^{100} + 30x^{60} + 10x^{50} + 20$$

Can we store the coefficients in an increase order of exponential index?

1. *Different data types can be used for the same type of problem.*
2. *There exists a common problem: the organization and management of ordered linear data.*

Outline

- List ADT
- Array
- Linked list
- Doubly linked list
- Node-based storage with arrays

List ADT

An Abstract List (or List ADT) is linearly ordered data (with same data type)

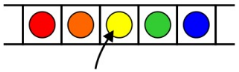
$$(\; A_1 \;\; A_2 \;\; \ldots \;\; A_{n-1} \;\; A_n \;)$$

- The number of elements in the List denotes the length of the List.
- When there is no element it is an empty List.
- The beginning of a List is called the List head; the end of a List is called the List tail.
- The same value may occur more than once.

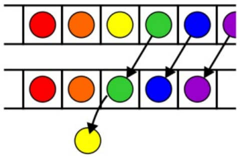
Operations

Operations at the k^{th} entry of the list include:

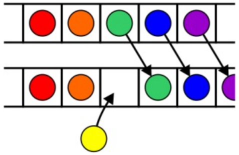
Access to the object



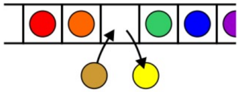
Erasing an object



Insertion of a new object

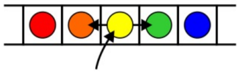


Replacement of the object



Operations

Given access to the k^{th} object, gain access to either the previous or next object



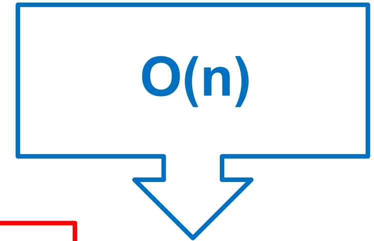
Given two abstract lists, we may want to

- Concatenate the two lists
- Determine if one is a sub-list of the other

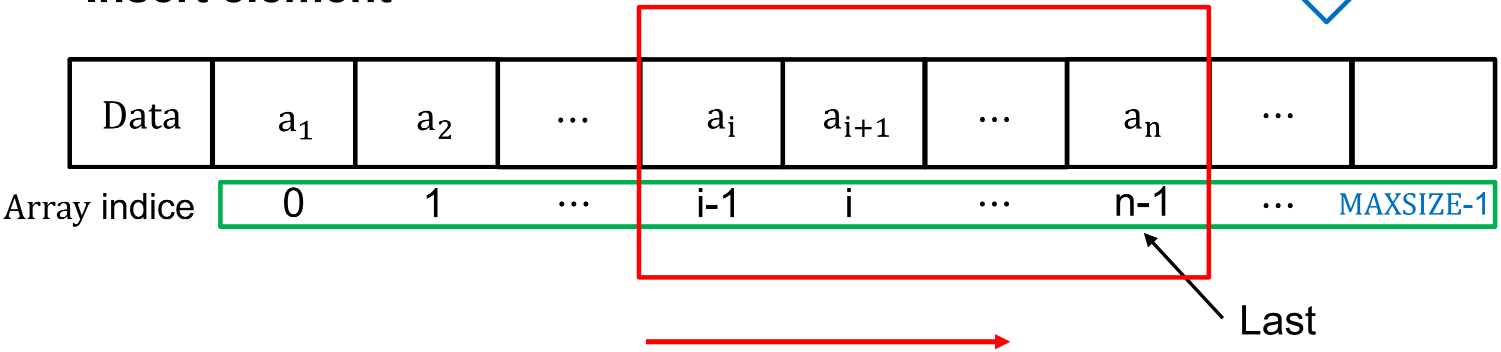
List based on array

Data	a_1	a_2	…	a_i	a_{i+1}	…	a_n	…	
Array indice	0	1	…	i-1	i	…	n-1	…	MAXSIZE-1

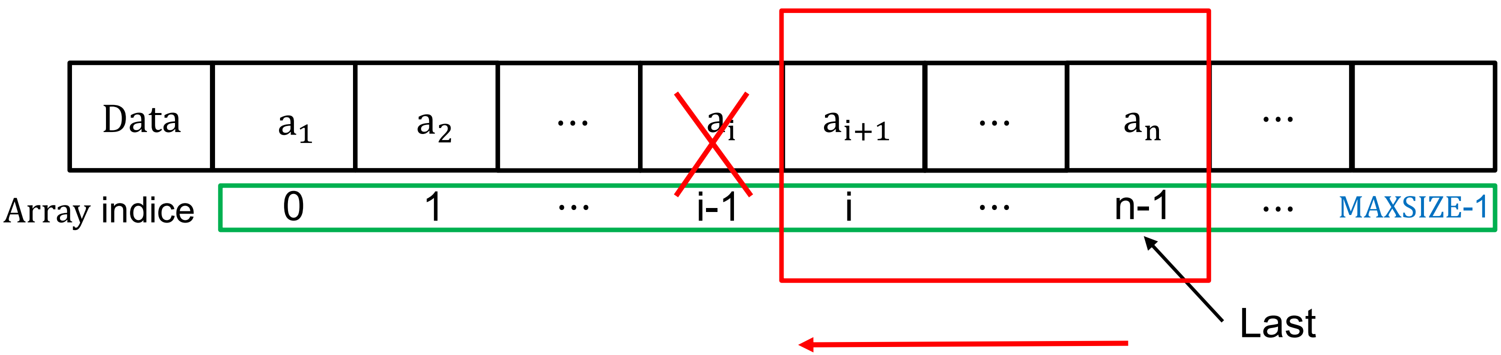
List based on array



- **Insert element**



- **Delete element**



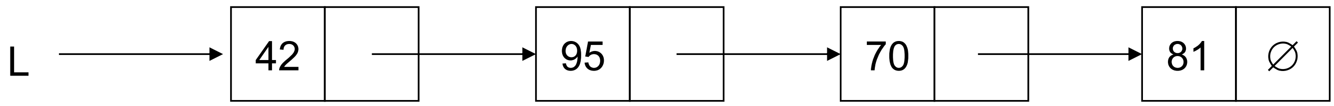
Outline

- List ADT
- Linked list
- Doubly linked list
- Node-based storage with arrays
- Application

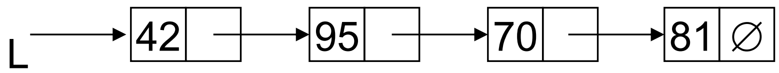
Definition

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



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;
};
```

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:

```
Node( int = 0, Node * = nullptr );
```

Structure

Let us look at the internal representation of a linked list

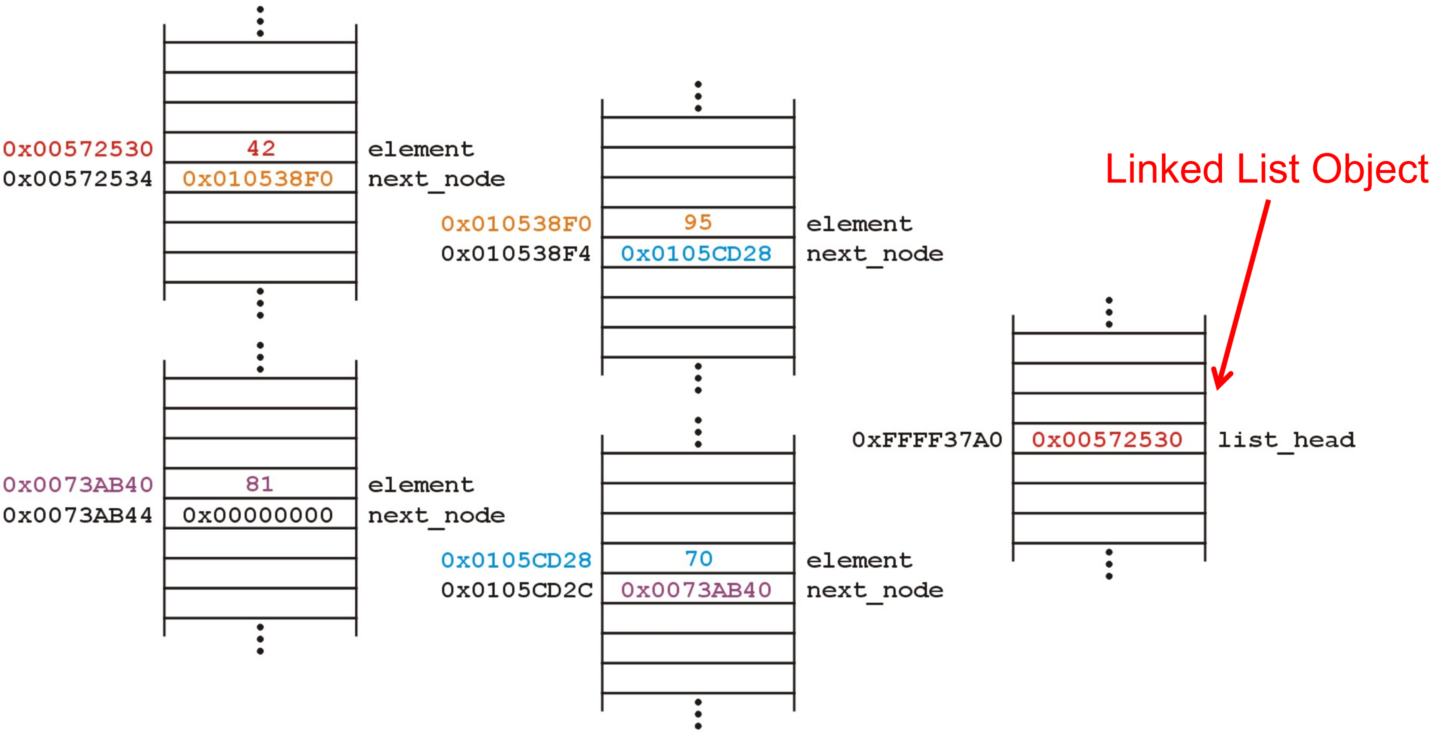
Suppose we want a linked list to store the values

`42   95   70   81`

in this order

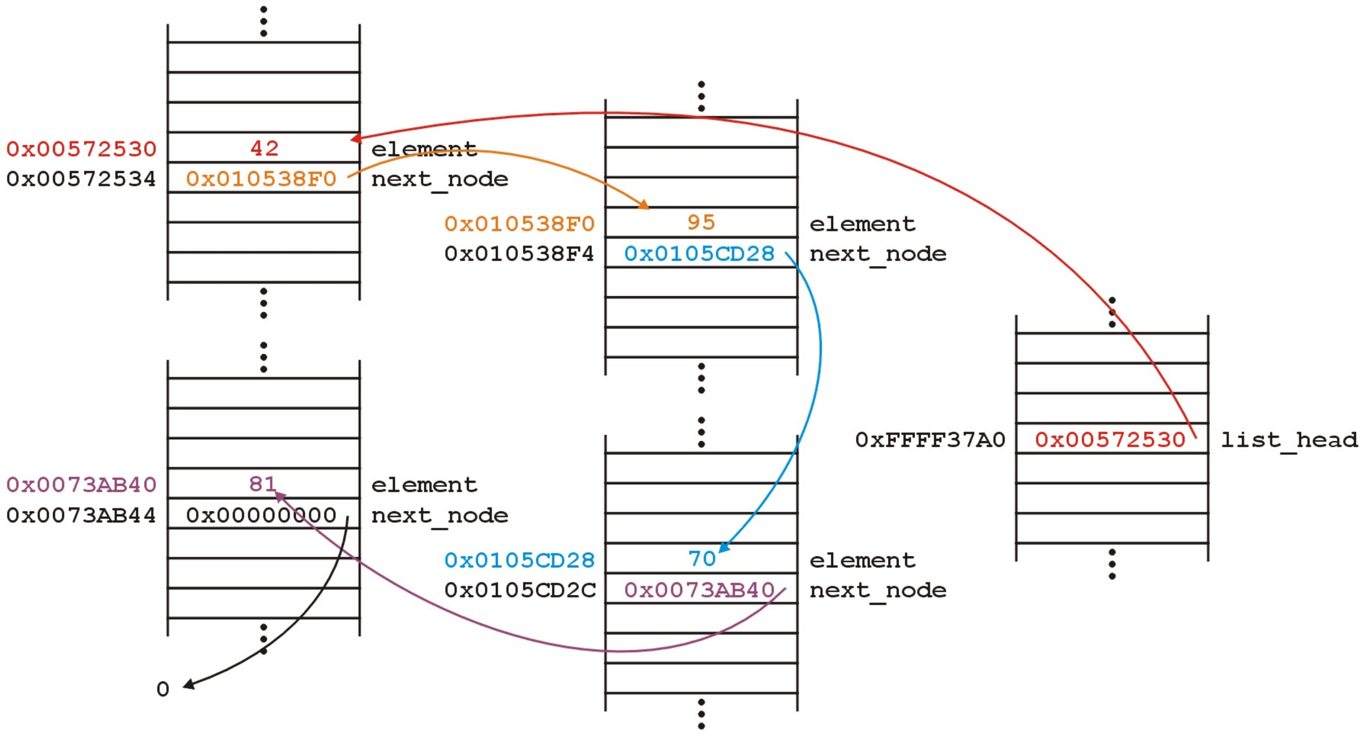
Structure

A linked list uses linked allocation, and therefore each node may appear anywhere in memory:



Structure

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

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

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;
```

void push_front(int)

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:

`void push_front( int )`

We must:

- create a new node which:
  - stores the value `81`, and
  - is pointing to `0`
- assign its address to `list_head`

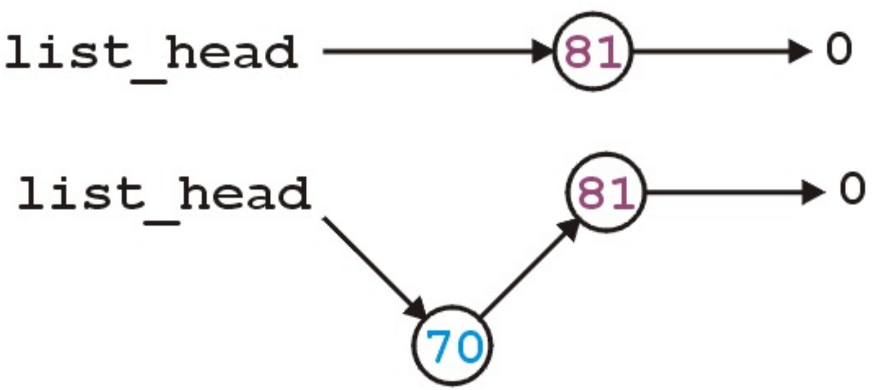
We can do this as follows:

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

void push_front(int)

Suppose however, we already have a non-empty list

Adding `70`, we want:



`void push_front( int )`

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)

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() );
    }
}
```

`void push_front( int )`

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 );
}
```

int pop_front()

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:

int pop_front()

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`

int pop_front()

Does this work?

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

    int e = front();
    delete head();
    list_head = head()->next();
    return e;
}
```

int pop_front()

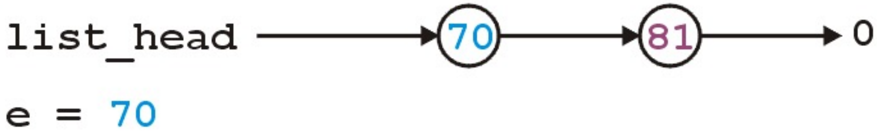
```
int List::pop_front() {
    if ( empty() ) {
        throw underflow();
    }

    int e = front();

    delete head();

    list_head = head()->next();

    return e;
}
```



int pop_front()

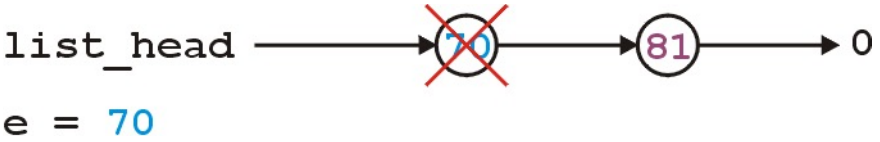
```
int List::pop_front() {
    if ( empty() ) {
        throw underflow();
    }

    int e = front();

    delete head();

    list_head = head()->next();

    return e;
}
```



int pop_front()

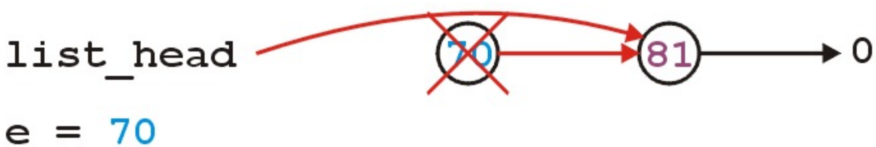
```
int List::pop_front() {
      if ( empty() ) {
          throw underflow();
      }

    int e = front();

    delete head();

    list_head = head()->next();

    return e;
}
```



Any problem with the above code?

int pop_front()

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

Stepping through a Linked List

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

`int erase( int )`

To remove an arbitrary element, *i.e.*, to implement `int erase( int )`, we must update the previous node

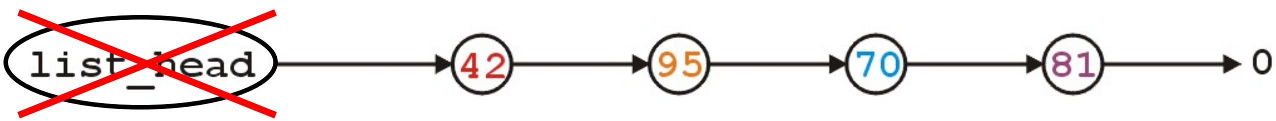
For example, given

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

Destructor

We dynamically allocated memory each time we added a new **`int`** into this list

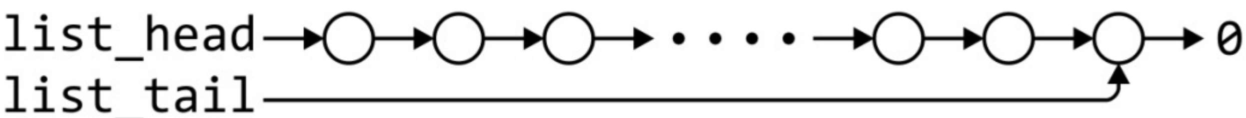
Suppose we delete a list before we remove everything from it
- This would leave the memory allocated with no reference to it



Linked list

	Front/1st node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$\mathrm{O}(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\mathrm{O}(n)$	$\Theta(n)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\mathrm{O}(n)$	$\Theta(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$\mathrm{O}(n)$	$\Theta(n)$

* These assume we have already accessed the k^{th} entry—an $\mathrm{O}(n)$ operation

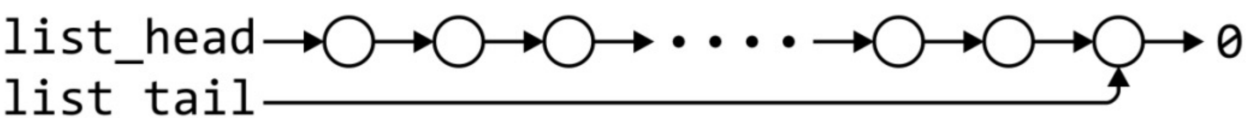


Assume we have a tail pointer

Linked list

	Front/1^{st} node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$O(n)$	$\Theta(n)$

By replacing the value in the node in question, we can speed things up



Assume we have a tail pointer

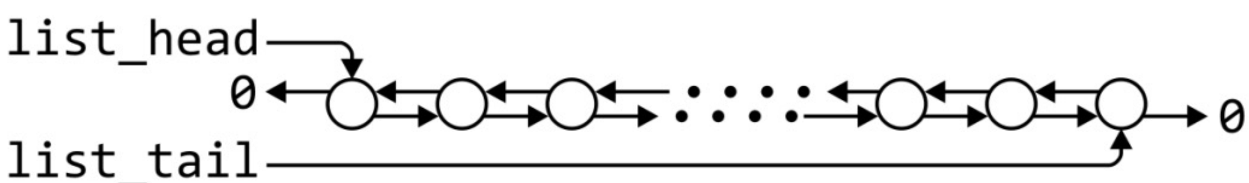
Outline

- List ADT
- Linked list
- Doubly linked list
- Node-based storage with arrays
- Application

Doubly linked lists

	Front/1^{st} node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$\Theta(1)^*$	$\Theta(1)$

* These assume we have already accessed the k^{th} entry—an $O(n)$ operation



Memory usage versus run times

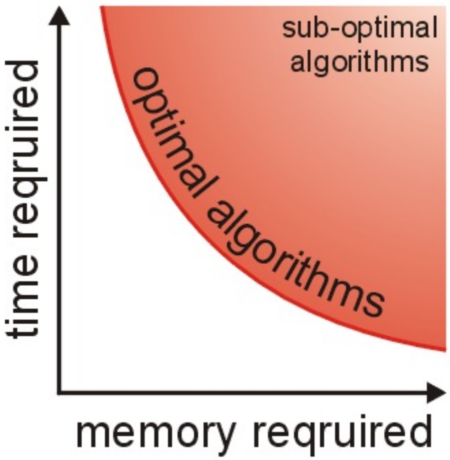
Using a doubly linked list requires $\Theta(n)$ additional memory, but it speeds up many operations

Memory usage versus run times

In general, there is an interesting relationship between memory and time efficiency

For a data structure/algorithm:

- Improving the run time usually requires more memory
- Reducing the required memory usually requires more run time



Memory usage versus run times

Warning: programmers often mistake this to suggest that given any solution to a problem, any solution which may be faster must require more memory

This guideline not true in general: there may be different data structures and/or algorithms which are both faster and require less memory

– This requires thought and research

Outline

- List ADT
- Linked list
- Doubly linked list
- Node-based storage with arrays
- Application

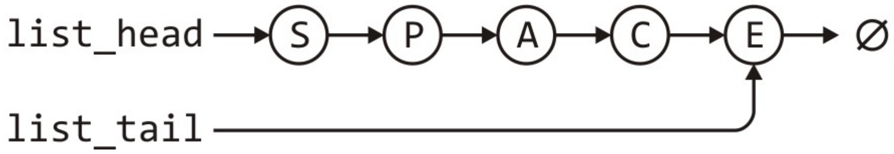
The issue

A significant issue with linked lists: node-based data structures require $\Theta(n)$ calls to `new`

- Each `new` operation requires a call to the operating system requesting a memory allocation

Using an array?

Suppose we store this linked list in an array?




```
list_head = 5;
list_tail = 2;
```

0	1	2	3	4	5	6	7
A		E	P		S	C	
6		-1	0		3	2	

Using an array?

Rather than using, `-1`, use a constant assigned that value

- This makes reading your code easier

```
list_head = 5;
list_tail = 2;
```

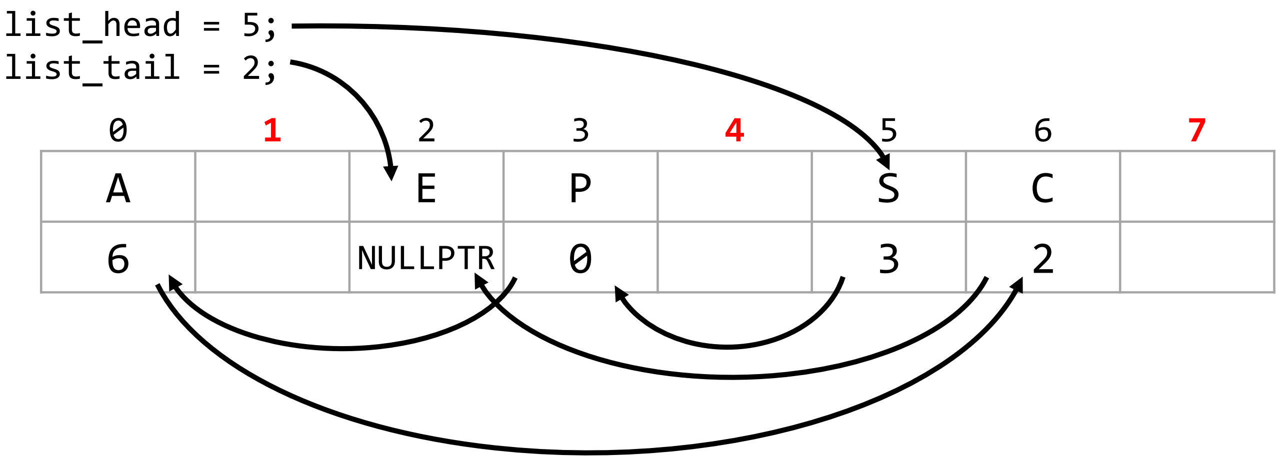
0	1	2	3	4	5	6	7
A		E	P		S	C	
6		NULLPTR	0		3	2	

A solution

Problem: when inserting a new element…

how do you know which cell to use?

– Solution: keep a container (a stack) of the indices of unused nodes




```
stack_size = 3;
```

0	1	2	3	4	5	6	7
7	1	4					

A better solution

Problem:

- Our solution requires $\Theta(N)$ additional memory
- In our initial example, the unused nodes are 1, 4 and 7
- How about using these to define a second stack-as-linked-list?

```
list_head = 5;
list_tail = 2;
```

0	1	2	3	4	5	6	7
A		E	P		S	C	
6		NULLPTR	0		3	2	

A better solution

Problem:

- Our solution requires $\Theta(N)$ additional memory
- In our initial example, the unused nodes are 1, 4 and 7
- How about using these to define a second stack-as-linked-list?

```
list_head = 5;
list_tail = 2;
stack_top = 1;
```

0	1	2	3	4	5	6	7
A		E	P		S	C	
6	4	NULLPTR	0	7	3	2	NULLPTR

- We only need a head pointer for the stack-as-linked-list

Analysis

This solution:

- Requires only three more member variable than our linked list class
- It still requires $O(N)$ additional memory over an array
- All the run-times are identical to that of a linked list
- Only one call to `new`, as opposed to $\Theta(n)$
- There is a potential for up to $O(N)$ wasted memory

Question: What happens if we run out of memory?

Reallocation of memory

Suppose we start with a capacity N but after a while, all the entries have been allocated

- We can double the size of the array and copy the entries over

```
list_head = 6;
list_tail = 4;
list_size = 8;
list_capacity = 8;
stack_top = NULLPTR;
```

0	1	2	3	4	5	6	7
C	R	U	T	R	U	S	T
7	2	0	1	NULLPTR	4	3	5

Reallocation of memory

Suppose we start with a capacity N but after a while, all the entries have been allocated

- We can double the size of the array and copy the entries over
- Only the stack needs to be updated and the old array deleted

```
list_head = 6;
list_tail = 4;
list_size = 8;
list_capacity = 16;
stack_top = 8;
```

0	1	2	3	4	5	6	7
C	R	U	T	R	U	S	T
7	2	0	1	NULLPTR	4	3	5

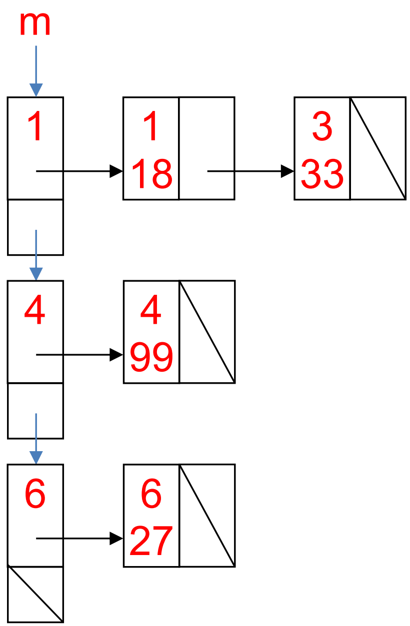
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C	R	U	T	R	U	S	T								
7	2	0	1	NULLPTR	4	3	5	9	10	11	12	13	14	15	NULLPTR

Outline

- List ADT
- Linked list
- Doubly linked list
- Node-based storage with arrays
- Application

Sparse Matrices

18	0	33	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	99	0	0
0	0	0	0	0	0
0	0	0	0	0	27



Summary

- List ADT
  - A sequence of elements (special case: string)
  - Array
- Linked list
  - Accessors and mutators
  - Stepping through a linked list
  - Copy and assignment operator
- Doubly linked list
  - Memory usage versus run times
- Node-based storage with arrays
  - No longer need to call `new` for each new node
- Application
  - Polynomial, sparse matrix