CS 2420: Lists

Dr. Tsung-Wei (TW) Huang

Department of Electrical and Computer Engineering

University of Utah, Salt Lake City, UT

Outline

- We will now look at our first abstract data structure
  - Relation: explicit linear ordering
  - Operations
  - Implementations of an abstract list with:
    - Linked lists
    - Arrays
  - Memory requirements
  - Strings as a special case
  - The STL `vector` class

List

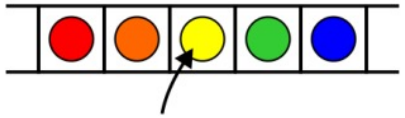
- An Abstract List (or List ADT) is linearly ordered data where the programmer explicitly defines the ordering

- We will look at the most common operations that are usually
  - The most obvious implementation is to use either an array or linked list
  - These are, however, not always the most optimal

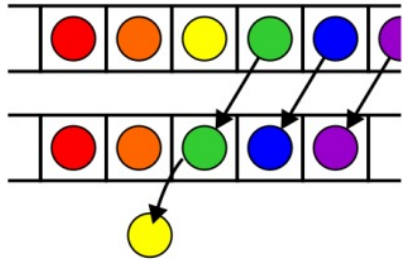
List Operations

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

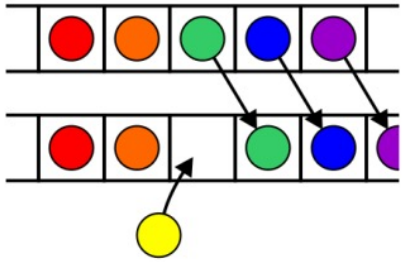
Access to the object



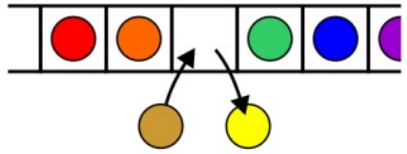
Erasing an object



Insertion of a new object

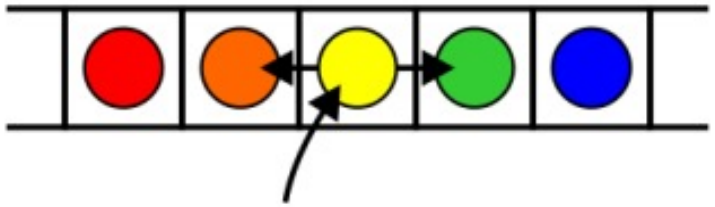


Replacement of the object



List 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

Locations and run times

- The most obvious data structures for implementing an abstract list are arrays and linked lists
  - We will review the run time operations on these structures

- We will consider the amount of time required to perform actions such as finding, inserting new entries before or after, or erasing entries at
  - the first location (the *front*)
  - an arbitrary (k^{th}) location
  - the last location (the *back* or n^{th})

- The run times will be $\Theta(1)$, $\mathrm{O}(n)$ or $\Theta(n)$

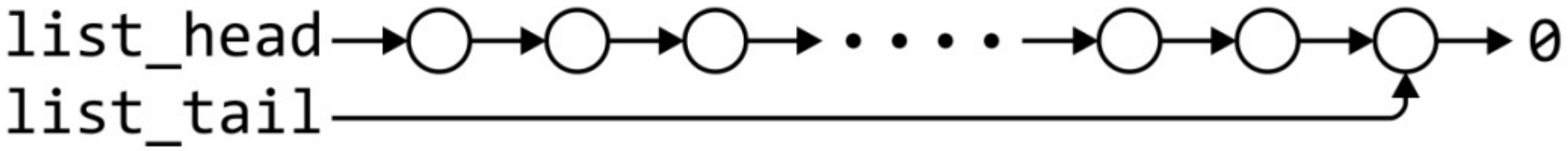
Linked lists

- We will consider these for
  - Singly linked lists
  - Doubly linked lists

Singly linked list

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

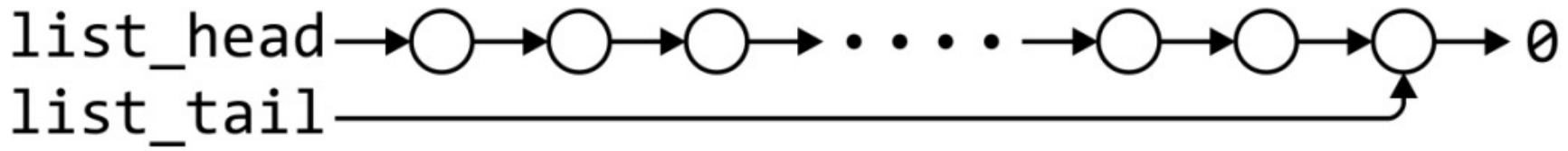
* These assume we have already accessed the k^{th} entry—an $O(n)$ operation



Singly 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	$O(n)$	$\Theta(n)$

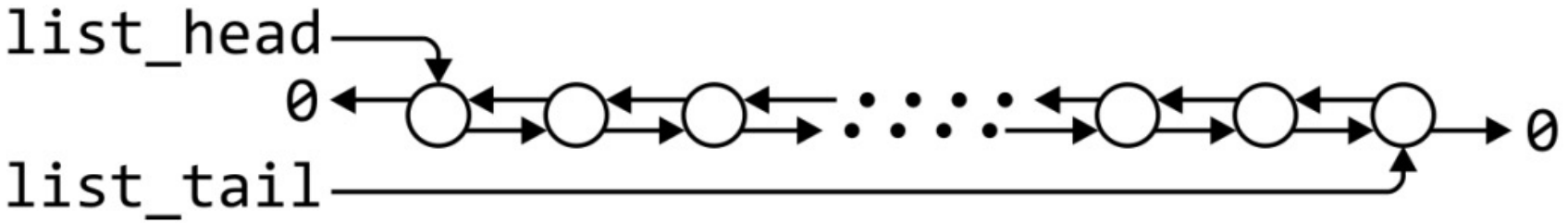
By replacing the value in the node in question, we can speed things up



Doubly linked lists

	Front/1st 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	$\Theta(1)^*$	$\Theta(1)$

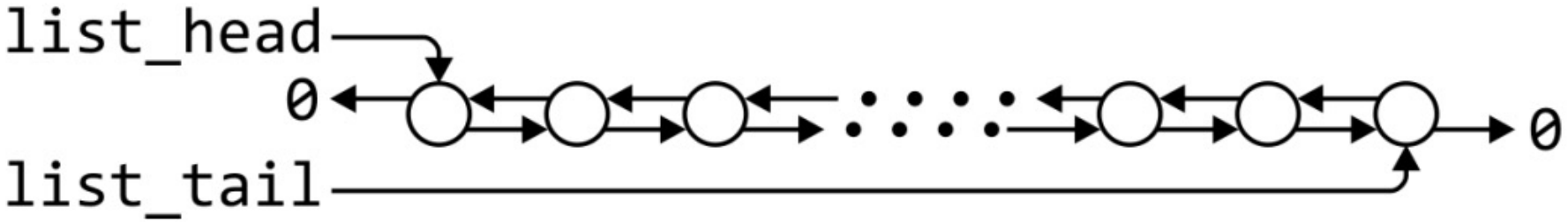
* These assume we have already accessed the k^{th} entry—an $O(n)$ operation



Doubly linked lists

	k^{th} node
Insert Before	$\Theta(1)$
Insert After	$\Theta(1)$
Replace	$\Theta(1)$
Erase	$\Theta(1)$
Next	$\Theta(1)$
Previous	$\Theta(1)$

Accessing the k^{th} entry is $O(n)$



Other operations on linked lists

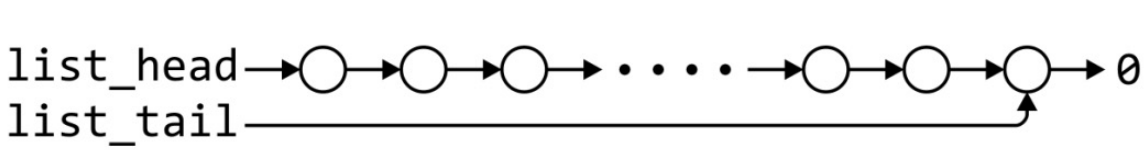
- Other operations on linked lists include:
  - Allocation and deallocating the memory requires $\Theta(n)$ time
  - Concatenating two linked lists can be done in $\Theta(1)$
    - This requires a tail pointer

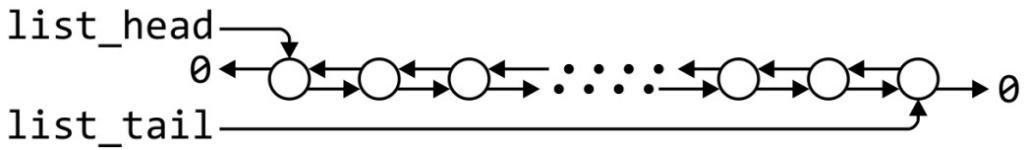
Standard arrays

- We will consider these operations for arrays, including:
  - Standard or one-ended arrays
  - Two-ended arrays

Run times

	Accessing the k^{th} entry	Insert or erase at the Front	Insert or erase at the k^{th} entry	Insert or erase at the Back
Singly linked lists	$O(n)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$ or $\Theta(n)$
Doubly linked lists				$\Theta(1)$
Arrays	$\Theta(1)$	$\Theta(n)$	$O(n)$	$\Theta(1)$
Two-ended arrays		$\Theta(1)$		





Data Structures

- In general, we will only use these basic data structures if we can restrict ourselves to operations that execute in $\Theta(1)$ time, as the only alternative is $\mathbf{O}(n)$ or $\Theta(n)$

- Interview question: in a singly linked list, can you speed up the two $\mathbf{O}(n)$ operations of
  - Inserting before an arbitrary node?
  - Erasing any node that is not the last node?

- If you can replace the contents of a node, the answer is “yes”
  - Replace the contents of the current node with the new entry and insert after the current node
  - Copy the contents of the next node into the current node and erase the next node

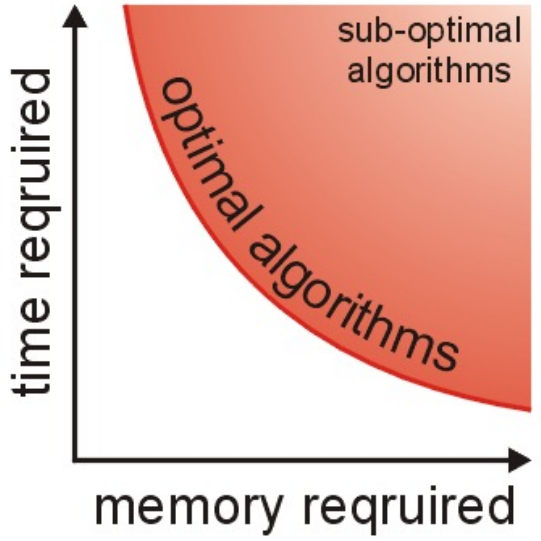
Memory usage versus run times

- All of these data structures require $\Theta(n)$ memory
  - Using a two-ended array requires one more member variable, $\Theta(1)$, in order to significantly speed up certain operations
  - Using a doubly linked list, however, required $\Theta(n)$ additional memory to speed up other operations

Memory usage versus run times

- As well as determining run times, we are also interested in memory usage
- 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

The `sizeof` Operator

- In order to determine memory usage, we must know the memory usage of the various built-in data types and classes
  - The **`sizeof`** operator in C++ returns the number of bytes occupied by a data type
  - This value is determined at compile time
    - It is **not** a function

The `sizeof` Operator

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

int main() {
     cout << "bool     " << sizeof( bool )     << endl;
     cout << "char     " << sizeof( char )     << endl;
     cout << "short    " << sizeof( short )    << endl;
     cout << "int      " << sizeof( int )      << endl;
     cout << "char *   " << sizeof( char * )   << endl;
     cout << "int *    " << sizeof( int * )    << endl;
     cout << "double   " << sizeof( double )   << endl;
     cout << "int[10]  " << sizeof( int[10] )  << endl;

     return 0;
}
```

```
{eceunix:1} ./a.out # output
bool      1
char      1
short     2
int       4
char *    4
int *     4
double    8
int[10]   40
{eceunix:2}
```

Standard Template Library

- In this course, you must understand each data structure and their associated algorithms
  - In industry, you will use other implementations of these structures

- The C++ Standard Template Library (STL) has an implementation of the `std::list` data structure
  - Excellent reference: https://en.cppreference.com/w/cpp/container/list

Standard Template Library

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    list<int> v;
    cout << "Is the list empty?  " << v.empty() << endl;
    cout << "Size of list:  "      << v.size()  << endl;
    v.push_back(42);
    v.push_back(91);
    for ( auto item : v ) {
        cout << item << endl;
    }
    return 0;
}
```

```
$ g++ vec.cpp
$ ./a.out
Is the list empty?  1
Size of list:  0
42
91
$
```

Summary

- In this topic, we have introduced Abstract Lists
  - Explicit linear orderings
  - Implementable with arrays or linked lists
    - Each has their limitations
    - Introduced modifications to reduce run times down to $\Theta(1)$
  - Discussed memory usage and the `sizeof` operator
  - Looked at the String ADT
  - Looked at the `std::list` class in the STL

LAB

- Write a list using the `std::list` library to manipulate the following operations
  - Insert five integers, 2, 5, 9, 10, -3
  - Print the elements in the list
  - Erase the element 9
  - Print the elements in the list
  - Insert a new integer -100 right after 10
  - Print the elements in the list
  - Remove the first element
  - Remove the last element
  - Print the elements in the list