Dynamic Structures, Singly Linked Lists

Michael R. Nowak
Texas A&M University

Some of the slides presented today were created by J. Michael Moore

Array

- Recall
  - Arrays are created to be a specific size.
    Once you run out of slots, you can't add any more elements.
  - Now that you know how to use dynamic memory, so
    you could create a new larger array and copy the elements over.
    - That's what vector does!
- Linked Lists
  - Can grow as large as needed (provided sufficient memory)

Array Insert

0	1	2	3	4	5	6
3	9	4	7	5	9	

- Insert 11 into the first position (i.e. index 0)

1. **Shift all elements**
2. Insert

Array Insert

0	1	2	3	4	5	6
3	9	4	7	5		9

- Insert 11 into the first position (i.e. index 0)

1. **Shift all elements**
2. Insert

Array Insert

0	1	2	3	4	5	6
3	9	4	7		5	9

- Insert 11 into the first position (i.e. index 0)

1. **Shift all elements**
2. Insert

Array Insert

0	1	2	3	4	5	6
3	9	4		7	5	9

- Insert 11 into the first position (i.e. index 0)

1. **Shift all elements**
2. Insert

Array Insert

0	1	2	3	4	5	6
3	9		4	7	5	9

- Insert 11 into the first position (i.e. index 0)

1. **Shift all elements**
2. Insert

Array Insert

0	1	2	3	4	5	6
3		9	4	7	5	9

- Insert 11 into the first position (i.e. index 0)

1. **Shift all elements**
2. Insert

Array Insert

0	1	2	3	4	5	6
	3	9	4	7	5	9

- Insert 11 into the first position (i.e. index 0)

1. **Shift all elements**
2. Insert

Array Insert

0	1	2	3	4	5	6
11	3	9	4	7	5	9

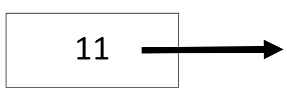
- Insert 11 into the first position (i.e. index 0)

1. Shift all elements
2. **Insert**

Dynamic Alternative

Note:
Data could be complex like a Class,
or simple like an int.

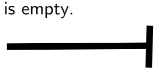
- Create Node
  - Contains Data
  - Contains Pointer/Reference to next element



Linked List

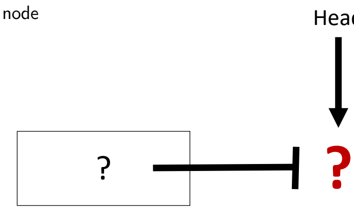
- Program starts with a pointer to the first node in the list.
- Normally pointer to start node is called **head**.

- Set to nullptr if the list is empty.



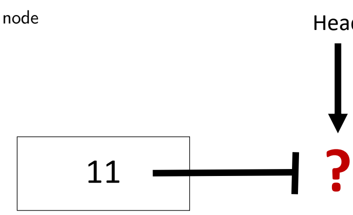
Add Item to Front of List

1. Create a new node



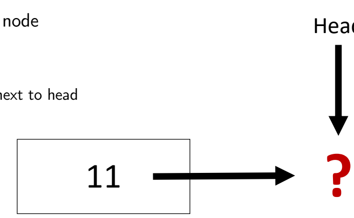
Add Item to Front of List

1. Create a new node
2. Set its value



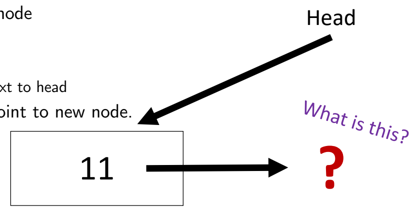
Add Item to Front of List

1. Create a new node
2. Set its value
3. Attach to list
   1. Set Node's next to head



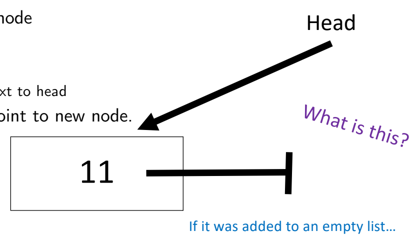
Add Item to Front of List

1. Create a new node
2. Set its value
3. Attach to list
   1. Set Nodes next to head
4. Set Head to point to new node.



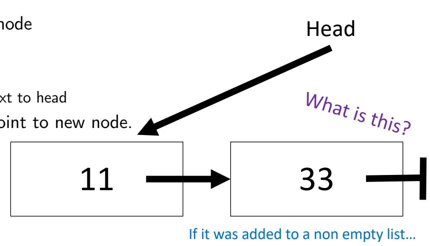
Add Item to Front of List

1. Create a new node
2. Set its value
3. Attach to list
   1. Set Nodes next to head
4. Set Head to point to new node.



Add Item to Front of List

1. Create a new node
2. Set its value
3. Attach to list
   1. Set Nodes next to head
4. Set Head to point to new node.



Linked List vs. Array

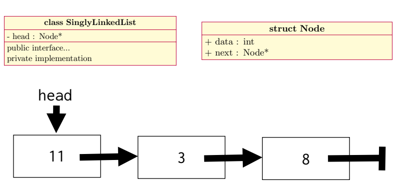
Linked List
- More memory
- Faster to insert item in middle
- Slower to get to item in list
- Can grow as needed

Array
- Less memory
- Slower to insert item in middle
- Faster to get to item in list
- Fixed size

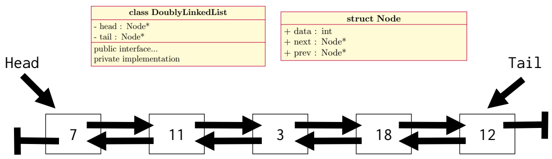
Dynamic Structures

- Dynamic
  - Memory is allocated during runtime (dynamic memory allocation)
- Structures
  - Aggregations of data of some type, usually pointers to other data, and perhaps some functionality, all encapsulated together
    - `struct Node` is frequently used
  - We can link dynamic objects of this nature together using pointers
  - These connected objects are know as data structures
  - A data structure's attributes and behaviors are commonly encapsulated together in a `class`

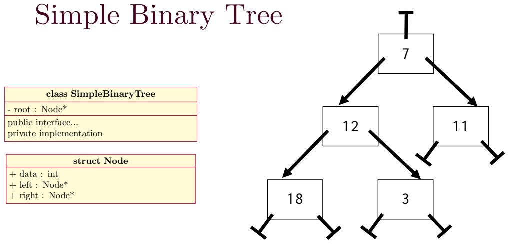
Dynamic Structure Example : Singly Linked List



Dynamic Structure Example : Doubly Linked List



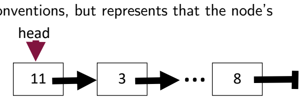
Dynamic Structure Example : Simple Binary Tree



Singly Linked Lists

Aside : Diagrams

- We will illustrate the singular linked list using diagrams with the following general representation
  - (arrows) represent memory addresses, so if two arrows point to the same thing, they have the same address / value
  - # (box) represents a node with the value #
  - # # (box-arrow-box) represents that the node with the tail end of the arrow's next points to node at the arrow's head
  - # follows the same conventions, but represents that the node's next points to `nullptr` (`0`)



Singly linked list : Properties

- Successive elements are connected by pointers
- The last element points to `nullptr`, which is defined to have the value `0`
- Can grow or shrink in size during run-time with dynamic memory allocation

Singly linked list : Common operations

- Insert
  - Inserts an element into the list
- Find
  - Find and return a specified node in the list
- Delete
  - Removes and returns an element residing at a specified position
- Empty
  - Check whether the list is empty

Singly linked list : Reasonable interface?

We will write our singly linked list under the assumption that it will store int data.

```
#ifndef NODE_H
#define NODE_H

struct Node
{
        int value;
        Node* next;
        Node(int value) : value(value),
next(nullptr) {}
        Node() : value(0), next(nullptr) {}
};

#endif
```

```
#ifndef MYLINKEDLIST_H
#define MYLINKEDLIST_H
#include "Node.h"

class MyLinkedList {
public:
    MyLinkedList();
    MyLinkedList(int);
    ~MyLinkedList();
    void insert(int);
    bool insert_at(int, int);
    Node& find(int);
    Node delete_at(int);
    bool is_empty();
private:
    Node* head;
    MyLinkedList(MyLinkedList const&);
    MyLinkedList& operator=(
                  MyLinkedList const&);
};

#endif
```

Singly linked list : Constructors

```
MyLinkedList::MyLinkedList() : head(nullptr) {}
```

```
MyLinkedList::MyLinkedList(int i) : head(new
Node(i)) {}
```

Singly linked list : `void insert(int i)`

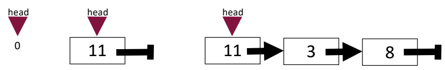
- For this example, my main() is in Source.cpp
- We can create a new MyLinkedList in main()
- We will do so invoking the `MyLinkedList()` constructor
  - In this case, the new list's head pointer will start out as nullptr (that is, 0)
- We would like to be able to insert an int value to the *end of our list*

```
#include <iostream>
#include "MyLinkedList.h"
#include "Node.h"
using namespace std;

int main() {
    MyLinkedList ll;
    ll.insert(20);
}
```

Singly linked list : `void insert(int i)`

- We would like to write a function `insert` that adds a node to the end of the list
  - We know whether or not we have at least one node in our list by looking at its $head$ pointer



Singly linked list : `void insert(int i)`

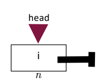
- First, we will declare a `Node* n` to a `new Node(i)`
- If we don't have any nodes in our list, we can set the $head$ pointer to the actual argument n and return from this function

```
// in MyLinkedList.cpp
void MyLinkedList::insert(int i)
{
    Node* n = new Node(i);

    // empty list case
    if (is_empty()) {
        head = n;
        return;
    }
    // otherwise non-empty list case
    // need to do something else
}
```

```
// in MyLinkedList.cpp
bool MyLinkedList::is_empty()
{
    return !head;
}
```

After calling insert(int), head points to that pointed to by n, which is the address where the new node with the value i lives

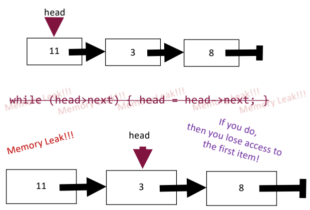


Singly linked list : `void insert(int i)`

- If at least one node in our list, we can set traverse the list until we arrive at the last node
- Recall that, the last element points to `nullptr`, which is defined to have the value `0`
  - Moreover that, calling `new Node()`initializes the object's next pointer to `nullptr`; our `insert()` function does not change the address to which the new node's (i.e., that created in the insert body for the passed value) next pointer points to
- Therefore, we can traverse the list by following each element's next pointer to the subsequent object in the list until we arrive at the object whose next pointer is set to the `nullptr`

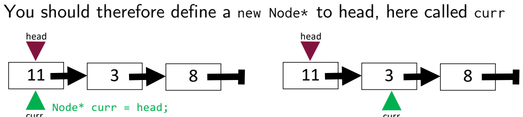
Singly linked list : `void insert(int i)`

- You have to be careful about how you walk through the list … if you try to use the head pointer to do this… well,



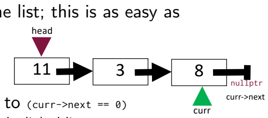
Singly linked list : `void insert(int i)`

- You should therefore define a `new Node*` to head, here called `curr`



- And the walk to the last element of the list; this is as easy as

```
while (curr) { curr = curr->next; }
```



- We know we've reached the last element of the list when `(curr->next == nullptr)`, which is equivalent to `(curr->next == 0)`
  - At this point, curr points to the last node in the linked list

Singly linked list : `void insert(int i)`

- Once you've arrive at the element at the end of the list, you simply assign that object's next pointer to `n`
- Therefore, putting things together, this function can be written as

```
void MyLinkedList::insert(int i)
{
    Node *n = new Node(i);
    if (is_empty()) {
        head = n;
        return;
    }
    Node* curr = head;
    while (curr->next) { curr = curr->next;}
    curr->next = n;

}
```

Note how this function performs the insertion operation differently for:
(1) An empty list
(2) A non-empty list

Singly linked list : `void insert(int i)`

1. Creating a new MyLinkedList in main `MyLinkedList l1;` creates an empty list

```
.----.
|Head|
  |
|Tail|
'----'
```

2. Now we can insert an integer to our list `l1.insert(20)`

```
.----.
|Head|
.----.
| 20 |
'----'
|Tail|
'----'
```

3. And if we wanted, we could insert another `l1.insert(10)`

```
.----.
|Head|
.----.
| 20 |
'----'
  |
.----.
| 10 |
'----'
|Tail|
'----'
```

Singly linked list : `void insert(int i)`

- Our void insert(int i) traverses the linked list to insert an item at its tail
  - This operation could be performed more efficiently if MyLinkList contained a pointer to the tail of the list: we could simply jump to the end of the list and add the new element
    - How might you go about implementing this?
- We would like to implement a LinkedList member function `insertAt(int i, int pos)` which adds a new Node element with value i element at position pos in the list. How would you implement this?

Singly linked list

- At this point, we are able to construct a linked list and add elements to it
- Let's say that we execute a block of code such as,

```
{
        MyLinkedList l1;
        for (int i = 0 ; i < 100 ; ++i) {
            int temp = randInt(1, 100);
            l1.insert(temp);
        }
}
```

- Given that `MyLinkedList l1` was created on the stack, memory allocated for `l1` is automatically deallocated once it goes out of scope
- Recall that `insert(node* n)` has multiple calls to `new` to create a `Node` object on the free store… we never `delete`d those Nodes… memory leak?

Singly linked list

- Following our suspicion that a memory leak might occur, we have run

```
int main()
{
    {
        MyLinkedList l1;
        for (int i = 0 ; i < 100 ; ++i) {
            int temp = randInt(1, 100);
            l1.insert(temp);
        }
    }
}
```

through a dynamic memory analysis tool

- Results:

```
LEAK SUMMARY:
   definitely lost: 16 bytes in 1 blocks
   indirectly lost: 1,584 bytes in 99 blocks
     possibly lost: 0 bytes in 0 blocks
```

- These result suggest that memory is indeed being leaked
  - Recall we have multiple calls to `new` in insert without any paired calls to `deleted` anywhere our class declaration

Singly linked list : `~MyLinkedList()`

- An appropriate place to deallocate an instance of MyLinkedList's free store / heap member objects (i.e., the `Node`s of the list) is in the destructor, `~MyLinkedList()`
- To understand why, let's shift focus and discuss destructors

Destructor : Responsibility

- As we have seen, automatic variables deallocate their memory once as they leave the scope from which they were declared
- Furthermore, we have seen how dynamic memory for an object can be freed by by calling delete on a pointer to that object
- In both cases, the respective object's destructor is implicitly called
- The *destructor* is ***responsible*** for *freeing* any *dynamic memory* that *belongs* to the *object*, ***before*** the *object's memory* is *freed*

Destructor : Responsibility

- When the *object's memory* is *freed*
  - The automatic memory allocated for non-static member variables is freed
  - This includes ***pointers***, meaning that we will no longer be able to use them to access to objects created on free store
- Accordingly, the destruction process proceeds by
  1. Calling the object's destructor function
  2. Calling the destruction functions for each data member that is derived from a class
     - Again, for emphasis, pointers to a class instance are not an object of the type defined by that class; they variables of the pointer datatype whose values are memory addresses
       - Consequently, if a respective pointer refers to a dynamically allocated memory object, that object will remain on the free-store unless we have already deleted it by this time (e.g., in 2)
  3. Calling the destructor function of the object's base classes
     - Don't worry about this until we get to inheritance

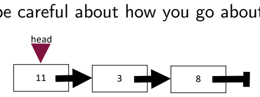
Destructor : Anatomy

- Destructor uses the Class name pre-pended by a tilde (~)
- No parameters allowed
- If you have to write one, then you are probably using '**delete**' in it to deallocate the '**new**' objects that were created in your class

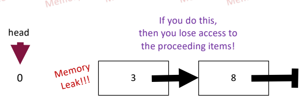
```
Class MyLinkedList {
public:
        // public interface
        // ...
        ~MyLinkedList(); // destructor
private:
        // private members
        // ...
}
```

Singly linked list : ~MyLinkedList()

- You have to be careful about how you go about this as well, for instance:

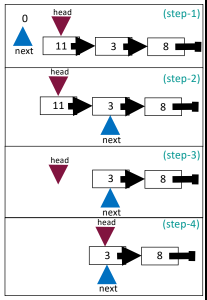


~~MyLinkedList::~MyLinkedList{ delete head; head = 0; }~~



Singly linked list : ~MyLinkedList()

- We need to traverse the list, deleting each node one-by-one
- To do this, we can make use of the head pointer, along with a Node* next initialized to nullptr (step-1)
- While we haven't reached the end of the list,
  - we can assign next the value of head->next (step-2)
  - delete head (step-3)
  - assign head the value of next (step-4)
- This process (step-2 through step-4) will continue until the last node, where the assignment of its next value to next will be nullptr, prompting the value of head to become nullptr after last element is deleted
  - This will prompt the while-statement's conditional that we're using for this process to evaluate to false



Singly linked list : ~MyLinkedList()

- Following the logic presented on the previous slide, ~MyLinkedList() can be defined as:

```
MyLinkedList::~MyLinkedList()
{
    Node* next = nullptr;
    while (head) {
        next = head->next;
        delete head;
        head = next;
    }
}
```

- Recall, it is always good practice to assign a pointer to a deleted object nullptr; in the code above, I did not write this explicitly for head because it is assigned to nullptr in the while-statement, after the last node element has been deleted

Singly linked list

- With the destructor now written to delete each dynamically allocated Node object, we again run

```
int main()
{
    {
        MyLinkedList l1;
        for (int i = 0 ; i < 100 ; ++i) {
            int temp = randInt(1, 100);
            l1.insert(temp);
        }
    }
}
```

  through a dynamic memory analysis tool

- Results:

```
LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
```

Singly linked list : Copy Constructor

- Need to implement as `MyLinkedList` objects have data members residing on the free store
  - Need to ensure that a deep copy is performed, and not the default member-wise copy (i.e., shallow copy)
- How would you write this?

Singly linked list : Copy Assignment Operator

- Need to implement as `MyLinkedList` objects have data members residing on the free store
  - Need to ensure that a deep copy is performed, and not the default member-wise copy (i.e., shallow copy)
- How would you write this?