CSC 212
Data Structures & Algorithms

Fall 2022 | Jonathan Schrader

Linked Lists

Housekeeping

Lab 4

- [more with] Dynamic Arrays

Assignment 2 Due

- Classes & Structs | CLASSES vs STRUCTS in C++ CLASSES in C++
- Pointers / References | POINTERS in C++ REFERENCES in C++
- Arrow operator & Dot Notation | The Arrow Operator in C++

Gradescope v Brightspace

- Check your grades in both
  - Discrepancies may appear due to:
    - Names / email addresses improperly unsigned
    - Late submissions

Code Sample

```
import time

n =- 100000

start  = time.time()
array = []
for i in range(n):
  array.append('s')
print(time.time() - start)

start  = time.time()
array = []
for i in range(n):
  array = array + ['s']
print(time.time() - start)
```

How are lists implemented in CPython

CPython's lists are really variable-length arrays, not Lisp-style linked lists. The implementation uses a contiguous array of references to other objects, and keeps a pointer to this array and the array's length in a list head structure.

This makes indexing a list $a[i]$ an operation whose cost is independent of the size of the list or the value of the index.

When items are appended or inserted, the array of references is resized. Some cleverness is applied to improve the performance of appending items repeatedly; when the array must be grown, some extra space is allocated so the next few times don't require an actual resize.

***CPython** is the reference implementation of the Python programming language*

Some STL Containers

Sequence Containers

Sequence containers maintain the ordering of inserted elements that you specify.

array

An `array` container has some of the strengths of vector, but the length isn't as flexible. For more information, see array Class.

vector

A `vector` container behaves like an array, but can automatically grow as required. It is random access and contiguously stored, and length is highly flexible. For these reasons and more, vector is the preferred sequence container for most applications. When in doubt as to what kind of sequence container to use, start by using a vector!

forward-list

A `forward_list` container is a singly linked list—the forward-access version of list.

list

array

```cpp
#include <string>
#include <iterator>
#include <iostream>
#include <algorithm>
#include <array>

int main()
{
  // construction uses aggregate initialization
  std::array<int, 3> a1{ {1, 2, 3} }; // double-braces required in C++11 prior to
  // the CWG 1270 revision (not needed in C++11 after the revision and in C++14 and beyond)
  std::array<int, 3> a2 = {1, 2, 3};  // double braces never required after =

  std::array<std::string, 2> a3 = { std::string("a"), "b" };
  // container operations are supported
  std::sort(a1.begin(), a1.end());
  std::reverse_copy(a2.begin(), a2.end(), std::ostream_iterator<int>(std::cout, " "));

  std::cout << '\n';
  // ranged for loop is supported
  for(const auto& s: a3)
      std::cout << s << ' ';
  // deduction guide for array creation (since C++17)
  [[maybe_unused]] std::array a4{3.0, 1.0, 4.0};  // -> std::array<double, 3>
}
```

https://en.cppreference.com/w/cpp/container/array

vector

```cpp
#include <iostream>
#include <vector>

int main()
{
  // Create a vector containing integers
  std::vector<int> v = { 7, 5, 16, 8 };

  // Add two more integers to vector
  v.push_back(25);
  v.push_back(13);

  // Print out the vector
  std::cout << "v = { ";
  for (int n : v) {
      std::cout << n << ", ";
  }
  std::cout <<b "}; \n";
}
```

https://en.cppreference.com/w/cpp/container/vector

forward_list

```cpp
#include    <forward_list>
#include    <iostream>

int main()  {
  std::forward_list<int>    numbers;
  std::cout <<  "Initially, numbers.empty():    "   <<  numbers.empty() <<  '\n';
  numbers.push_front(42);
  numbers.push_front(13317);
  std::cout <<  "After  adding  elements,   numbers.empty():    "   <<  numbers.empty() <<b '\n';
}
```

https://en.cppreference.com/w/cpp/container/forward_list

list

```cpp
#include <algorithm>
#include <iostream>
#include <list>

int main()
{
  // Create a list containing integers
  std::list<int> l = { 7, 5, 16, 8 };

  // Add an integer to the front of the list
  l.push_front(25);
  l.push_back(13);

  // Insert an integer before 16 by searching
  auto it = std::find(l.begin(), l.end(), 16);
  if (it != l.end())
      l.insert(it, 42);

  // Print out the list
  std::cout << "l = { ";
  for (int n : l)
      std::cout << n << ", ";
  std::cout <<b "};\n";
}
```

https://en.cppreference.com/w/cpp/container/list

Linked Lists

Arrays

Think about making *insertions* and *deletions* efficiently…

What is the computational cost of inserting or deleting 1 element?

- rear?
- front?
- middle?

Linked Lists

Collections of sequential elements stored at *non-contiguous* locations in memory

Elements are stored in *nodes*

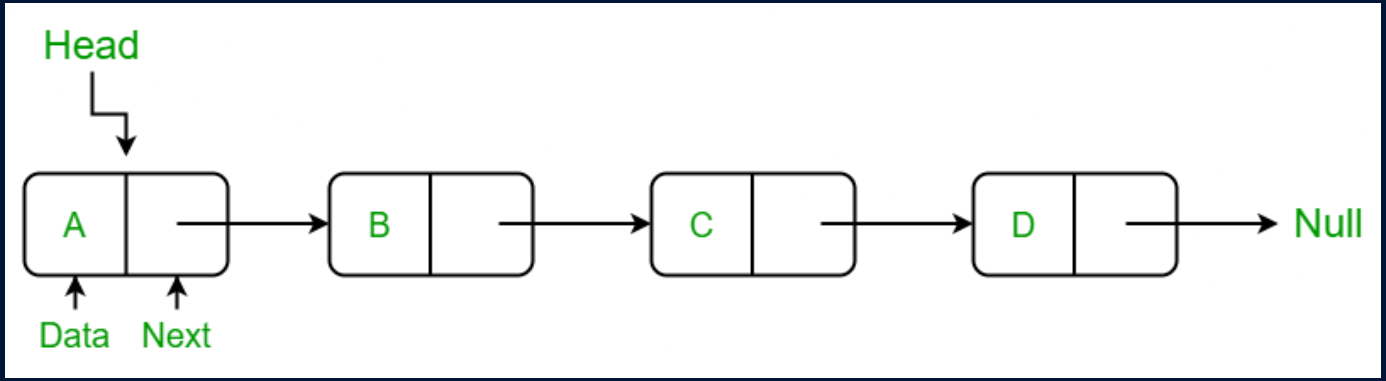
Nodes are connected by *links*

- every node keeps a pointer to the next node

Can *grow* and *shrink* dynamically

Allow for fast insertions/deletions

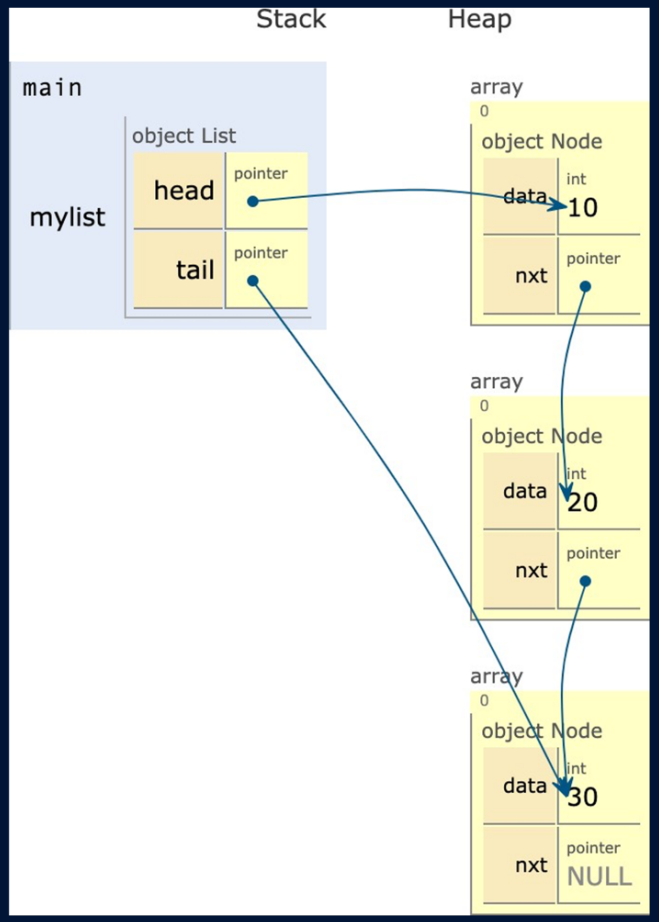
Singly Linked List



Node Left: data
Node Right: memory location of next element in LL

Pseudo-implementation

```
int main()  {
  List mylist;
  mylist.insert_end(10);
  mylist.insert_end(20);
  mylist.insert_end(30);
}
```



Operations on Linked Lists

Linked lists are just *collections* of sequential data

- can *insert* 1 or more elements
  - *front, end, by index, by value (sorted lists)*
- can *delete* 1 or more elements
  - *front, end, by index, by value*
- can *search* for a specific element
- can *get* an element at a given index
- can *traverse* the list
  - visit all nodes and perform an operation (e.g. print or destroy)

Implementing a Singly Linked List

Linked lists in C++ (prereqs)

C++ Classes

Pointers

- `NULL` pointers

Dynamic Memory Allocation

- `new`
- `delete`

Pointers and Classes

- dot notation (`.`)
- arrow notation (`->`)

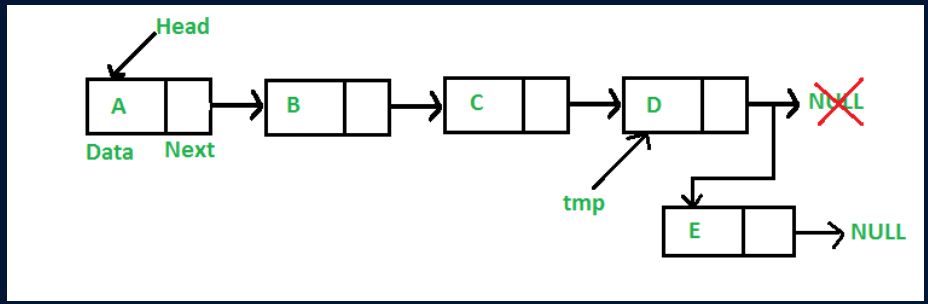
class Node

```cpp
#include <iostream>

using namespace std;

// A linked list node
class Node
{
  public:
    int data;
    Node *next;
};
// This code is contributed by rathbhupendra
```

class Node | https://www.geeksforgeeks.org/what-is-linked-list/

Append (insert at end)

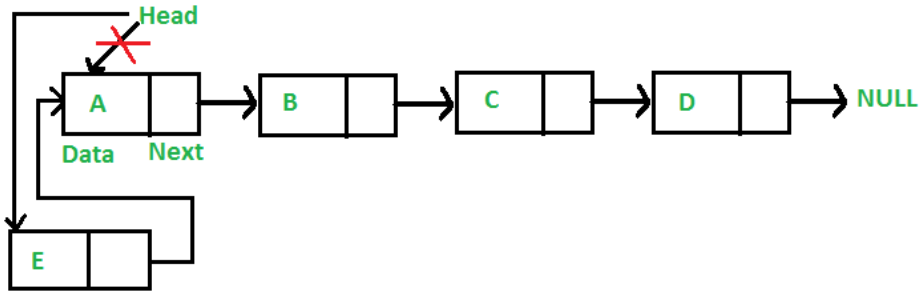
```
// Given a reference (pointer to pointer) to
// the head of a list and an int, appends a
// new node at the end
void insertTail(Node** head_ref, int new_data)
{
  Node* new_node = new Node();
  Node *last = *head_ref;
  new_node->data = new_data;
  new_node->next = NULL;
  if (*head_ref == NULL) {
    *head_ref = new_node;
    return;
  }
  while (last->next != NULL) {
    last = last->next;
  }
  last->next = new_node;
  return;
}
```



append() | https://www.geeksforgeeks.org/linked-list-set-2-inserting-a-node/

Prepend (insert at front)

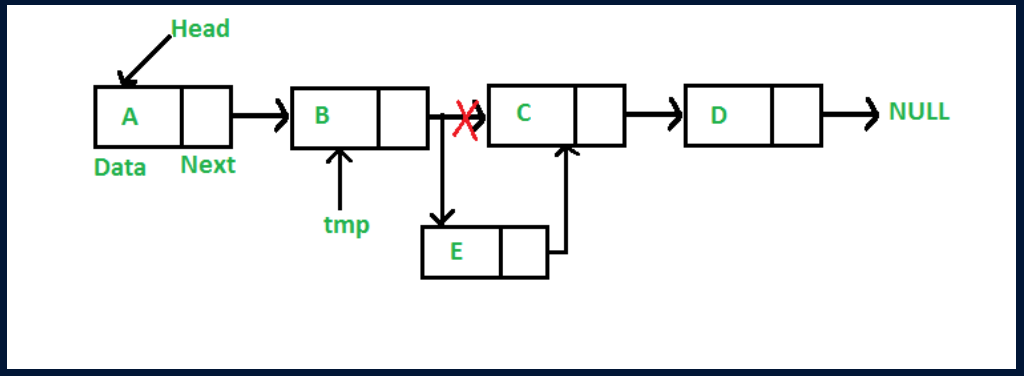
```
void insertHead(Node** head_ref, int new_data)
{
  Node* new_node = new Node();
  new_node->data = new_data;
  new_node->next = (*head_ref);
  (*head_ref) = new_node;
}
```



push() | https://www.geeksforgeeks.org/linked-list-set-2-inserting-a-node/

Insert by index

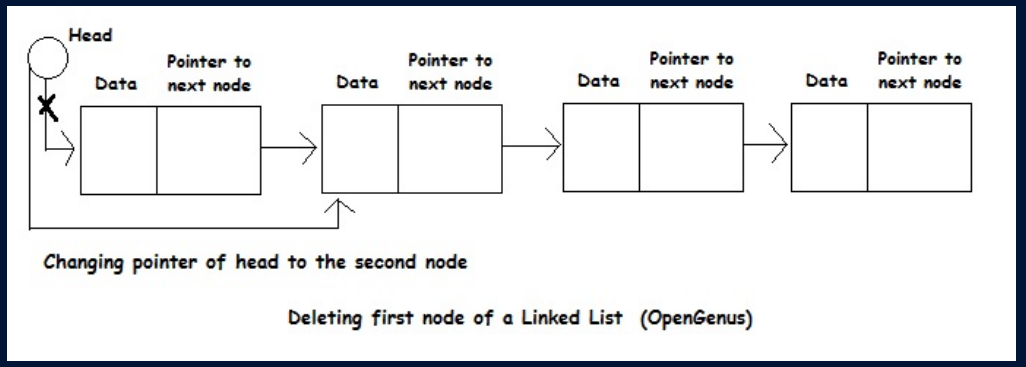
```
int insertAtIdx(Node* head, int index,
                int new_data)
{

    Node* current = head;
    Node* prev_node;

    int count = 0;
    while (current != NULL) {
        if (count == index) {
      Node* new_node = new Node();
      new_node->data = new_data;
      new_node->next = prev_node->next;
      prev_node->next = new_node;
      return (current->data);
        }
        count++;
        prev_node = current;
        current = current->next;
    }
    assert(0);
}
```



push() | https://www.geeksforgeeks.org/linked-list-set-2-inserting-a-node/

Delete at front

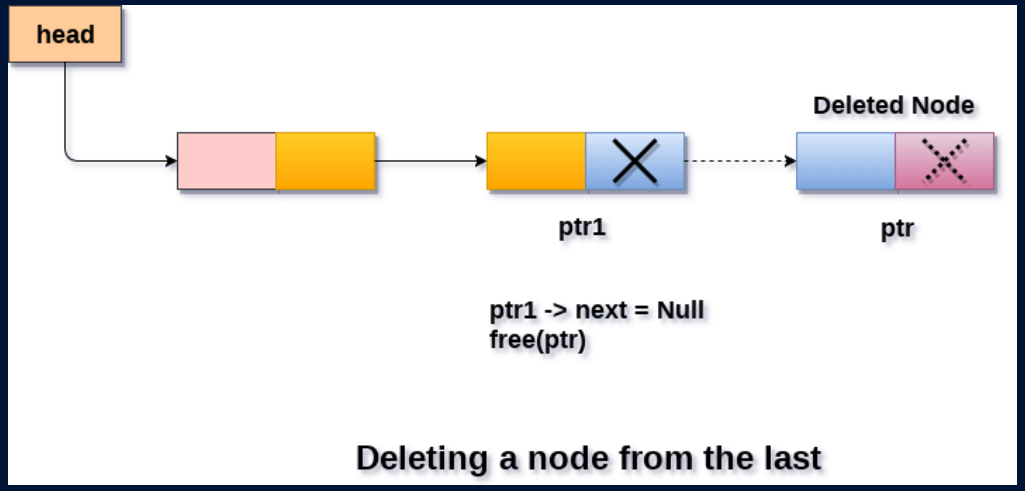
```
void deleteHead(Node** head_ref)
{

  // Store head node
  Node* temp = *head_ref;
  Node* prev = NULL;

  // If head node itself holds
  // the key to be deleted
  if (temp != NULL) {

  // Changed head
    *head_ref = temp->next;

  // free old head
    delete temp;
    return;
  }
}
```



deleteNode() | https://www.geeksforgeeks.org/linked-list-set-3-deleting-node/?ref=lbp

Delete at end

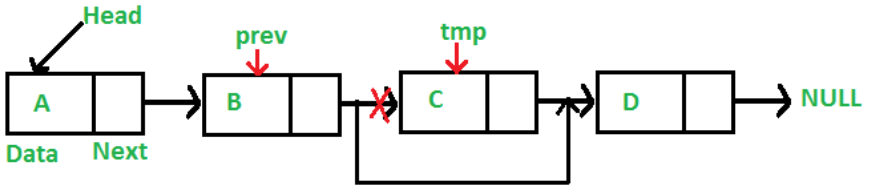
```
void deleteTail(Node** head){
  Node* prev = NULL;
  Node* temp = *head;
  while(temp->next!=NULL){
    prev = temp;
    temp = temp->next;
  }
  delete temp;
  prev->next = NULL;
  return;
}
```



deleteatTail() | https://www.tutorialspoint.com/delete-a-tail-node-from-the-given-singly-linked-list-using-cplusplus

Delete at value

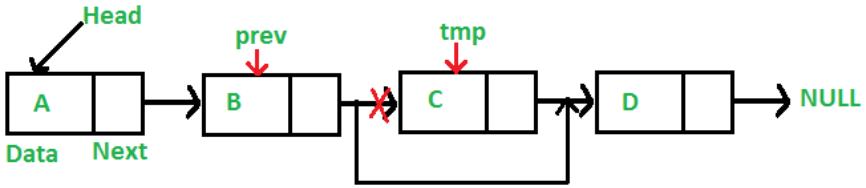
```
void deleteNode(Node** head_ref, int key)
{

  Node* temp = *head_ref;
  Node* prev = NULL;

  if (temp != NULL && temp->data == key) {
    *head_ref = temp->next;
    delete temp;
    return;
  }
  else {
    while (temp != NULL && temp->data != key) {
      prev = temp;
      temp = temp->next;
    }
    if (temp == NULL)
      return;
    prev->next = temp->next;
    delete temp;
  }
}
```



deleteNode() | https://www.geeksforgeeks.org/linked-list-set-3-deleting-node/

Delete at index

```
void deleteByPos(Node** head_ref, int position)
{
    if (*head_ref == NULL)
        return;
    Node* temp = *head_ref;
    if (position == 0) {
        *head_ref = temp->next;
        free(temp);
        return;
    }
    for (int i = 0; temp != NULL && i
                < position - 2; i++)
        temp = temp->next;
    if (temp == NULL || temp->next == NULL)
        return;
    Node* next = temp->next->next;
    free(temp->next); // Free memory
    temp->next = next;
}
```



deleteNode() | https://www.geeksforgeeks.org/delete-a-linked-list-node-at-a-given-position/

Get

```
int getNth(Node* head, int index)
{
  Node* current = head;

  int count = 0;
  while (current != NULL) {
    if (count == index)
      return (current->data);
    count++;
    current = current->next;
  }

  assert(0);
}
```

GetNth() | https://www.geeksforgeeks.org/write-a-function-to-get-nth-node-in-a-linked-list/

Search

```
bool search(Node* head, int x)
{
  Node* current = head; // Initialize current
  while (current != NULL) {
    if (current->data == x)
      return true;
    current = current->next;
  }
  return false;
}
```

search() | https://www.geeksforgeeks.org/search-an-element-in-a-linked-list-iterative-and-recursive/

Destroy

```
void destroyList(Node** head_ref)
{

  Node* current = *head_ref;
  Node* next = NULL;

  while (current != NULL)
  {
    next = current->next;
    free(current);
    current = next;
  }

  *head_ref = NULL;
}
```

deleteList() | https://www.geeksforgeeks.org/write-a-function-to-delete-a-linked-list/

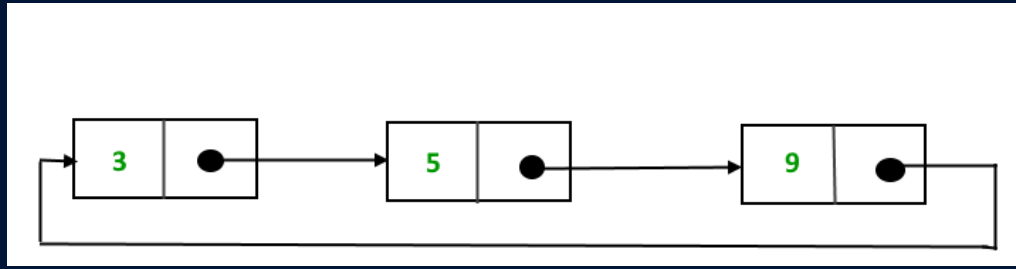
Traverse

```
void printList(Node* node)
{
  while (node != NULL)
  {
    cout << node->data << "->";
    node = node->next;
  }
  cout << "NULL" << endl;
}
```

printList() | https://www.geeksforgeeks.org/what-is-linked-list/?ref=lbp

Circular Singly Linked List

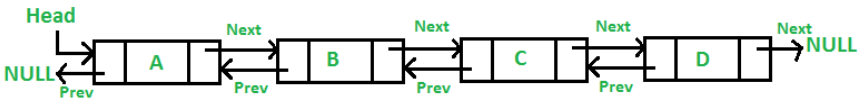
```
// Initialize the Nodes.
Node one = new Node(3);          // head
Node two = new Node(5);
Node three = new Node(9);        // tail

// Connect nodes
one.next = two;
two.next = three;
three.next = one;
```



https://www.geeksforgeeks.org/circular-linked-list/?ref=lbp

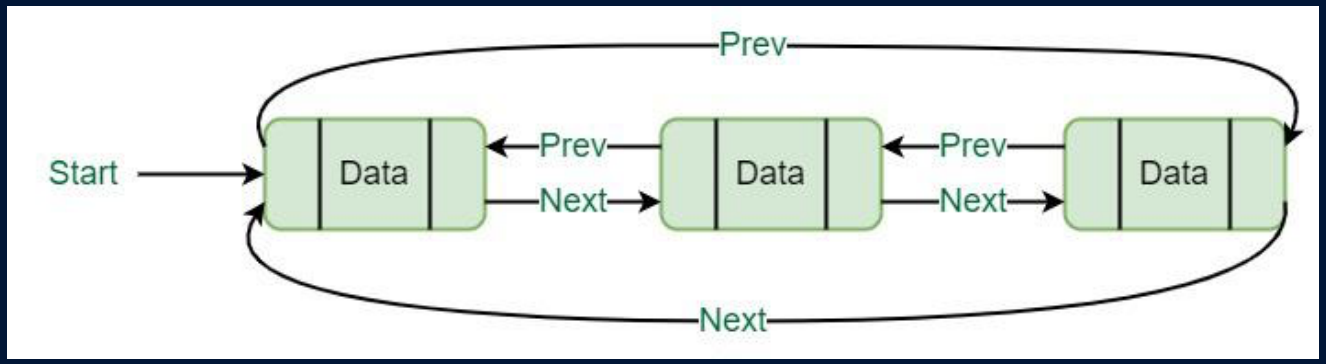
Doubly Linked List

```
// Node of a doubly linked list
class Node {
public:
    int data;

    // Pointer to next node in DLL
    Node* next;

    // Pointer to previous node in DLL
    Node* prev;
};

// This code is contributed by shivanisinghss2110
```



https://www.geeksforgeeks.org/doubly-linked-list/?ref=lbp

Circular Doubly Linked List



https://www.geeksforgeeks.org/doubly-circular-linked-list-set-1-introduction-and-insertion/