Data Structures & Algorithms

Week 2 — Lists, Elements of Complexity

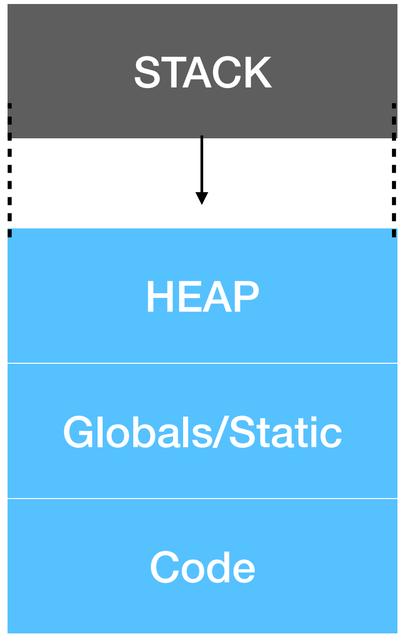
Subodh Sharma, Rahul Garg
{svs,rahulgarg}@iitd.ac.in

Recap — Week 1

- Introduction to C++
  - Variables and Basic Types
  - Functions and Selection operator
  - Loops, Arrays
  - Scopes — Global, Local and Block
  - C++ Pointers
  - Classes, Access Specifiers (Public, Private, Protected), Inheritance
- **Assignment 1 is OUT! Start early and earnestly! Ask questions on Piazza!**

Clarifications - Process Memory

```cpp
#include <iostream>
#include <stdlib.h>

int globalVar; // BSS Segment - Uninitialized Global Variable

int globalVarInit = 10; // Data Segment - Initialized Global Variable

int main() {

    int localVar = 10; // Stack - Local Variable

    int *ptr = new int; // Heap - Dynamically Allocated Variable

    std::cout << "Address of Text Segment: " << (void*)&main << std::endl;
    std::cout << "Address of Data Segment: " << (void*)&globalVarInit << std::endl;
    std::cout << "Address of BSS Segment: " << (void*)&globalVar << std::endl;
    std::cout << "Address of Heap: " << ptr << std::endl;
    std::cout << "Address of Stack: " << (void*)&localVar << std::endl;

    delete ptr; // Remember to free the dynamically allocated memory
    return 0;
}
```



Clarifications - Inheritance Keywords

```
#include<iostream>

class Base {
public:
    virtual void display() {
      std::cout << "Base Display"
                << std::endl;
    }
};

class Derived : public Base {
private:
    int *pointer;

public:
    Derived(int val) {
        pointer = new int(val);
    }

    void display() {
      std::cout << "Derived Display, Value: "
                << *pointer << std::endl;
    }
};
```

- Public Inheritance:
  - Public members of Base —> Public members of Derived
  - Protected members of Base —> Protected members of Derived
- Protected Inheritance:
  - Public and Protected members of Base —> Protected members of Derived
- Private Inheritance:
  - All derived members become private

```
int cols);
```

Clarifications - Pointers & Arrays

```
int main() {
  int rows = 3, cols = 4;

  // Dynamically allocate memory for a 2D array
  int **arr = new int*[rows];
  for(int i = 0; i < rows; i++) {
    arr[i] = new int[cols];
  }

  // Initialize the 2D array
     ...

  foo(arr, rows, cols);

  // Don't forget to free the memory
  for(int i = 0; i < rows; i++) {
    delete [] arr[i];
  }
  delete [] arr;

  return 0;
}
```

Clarifications — Variables escaping scope

- This leads to undefined behaviour
  - Bad programming practice

```
int* dangerousFunction() {
    int temp = 10;
    return &temp;
}
```

- Safe option

```
int* safeFunction() {
    int* heapVariable = new int(5);
    return heapVariable;
}
```

Debug Challenge

Scan me!

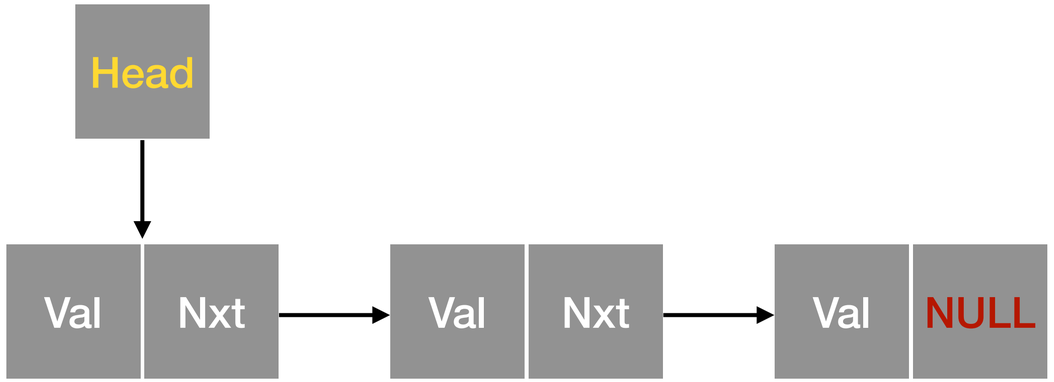
Debug Challenge

```cpp
#include<iostream>

class Base {
public:
    virtual void display() {
      std::cout << "Base Display"
                << std::endl;
    }
};

class Derived : public Base {
private:
    int *pointer;

public:
    Derived(int val) {
        pointer = new int(val);
    }

    void display() {
      std::cout << "Derived Display, Value: "
                << *pointer << std::endl;
    }
};
```

```cpp
int main() {
 1: Derived *d =
        new Derived(5);

 2: Base *b = d;
 3: b->display();

 4: delete b;

 5:d->display();

  return 0;
}
```

Linked Lists

- An abstract data type which represents values in a sequence
- Homogeneous container
- Need not be contiguous
- Dynamic data structure:
  - The list can grow and shrink
- Efficient memory utilisation
- Inventors: Allen Newell, Cliff Shaw, Herbert Simon in 1955 (RAND Corp. + CMU)

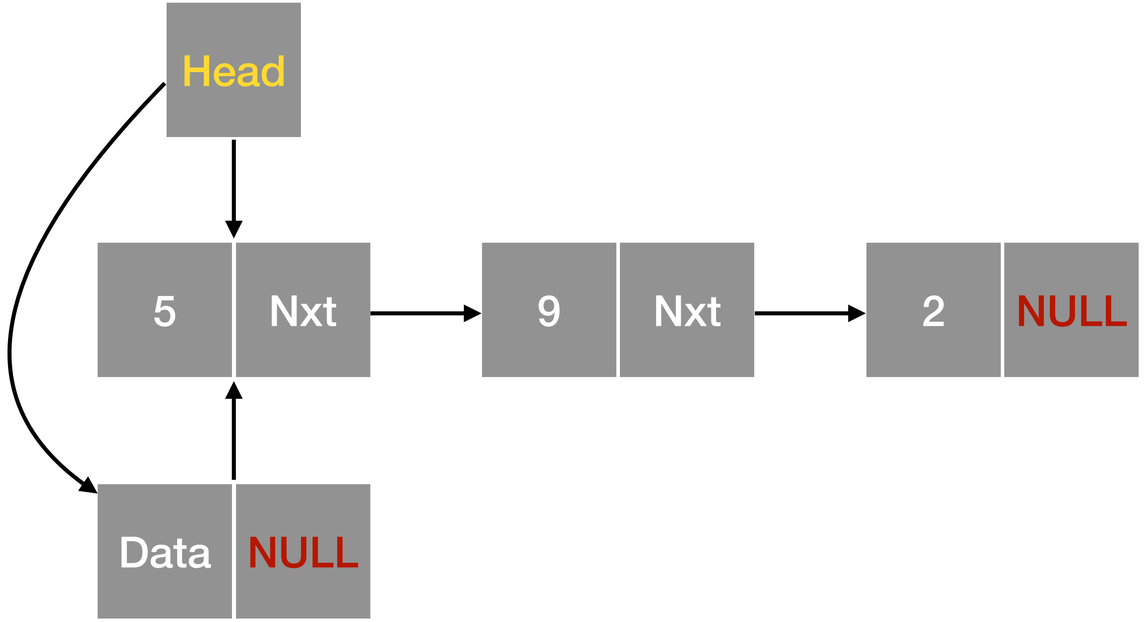



```
class Node{
  private:
    int val;
    Node * next;
  public:
    Node(int value, Node *n=nullptr){
      val = value;
      next = n;
    }
};
```

Insertion in a Linked List

- Insert @ head

```
void insertAtHead(int data) {
    Node *newNode = new Node(data);
    newNode->next = head;
    head = newNode;
  }
```



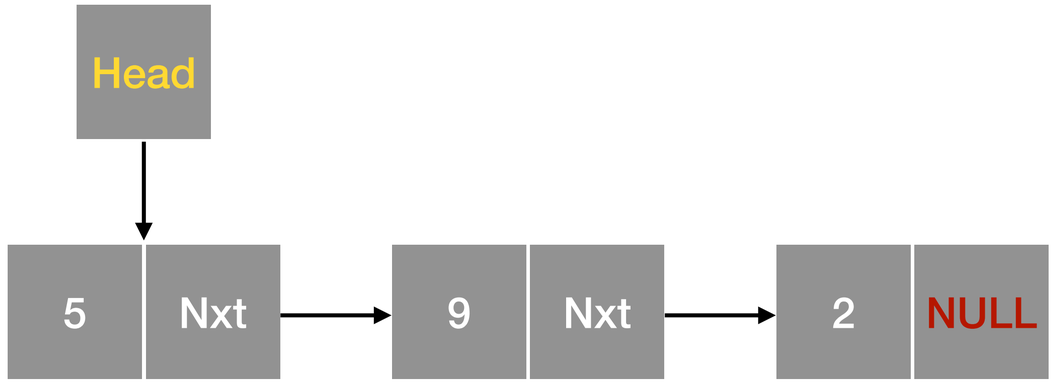
- Similarly, insert after a given node

```
void insertAfter(Node *n, int data) {
  assert (n != nullptr);
  Node newNode = new Node(data);
  newNode->next = n->next;
  n->next = newNode;

}
```

- Q: How to insert at the end?

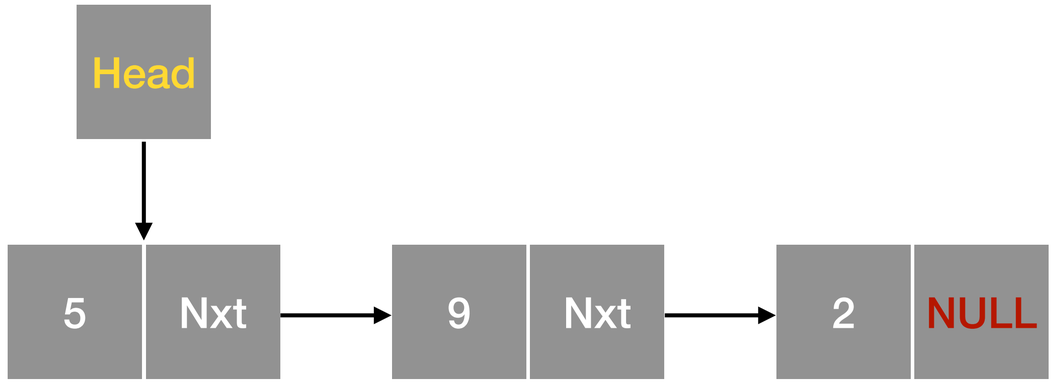
Deletion in a Linked List

- Delete from head:
  - `tmp = head;`
  - `head = head->next;`
  - `delete tmp;`
- Practice Q:
  - Delete after a specified node
  - Delete from the end



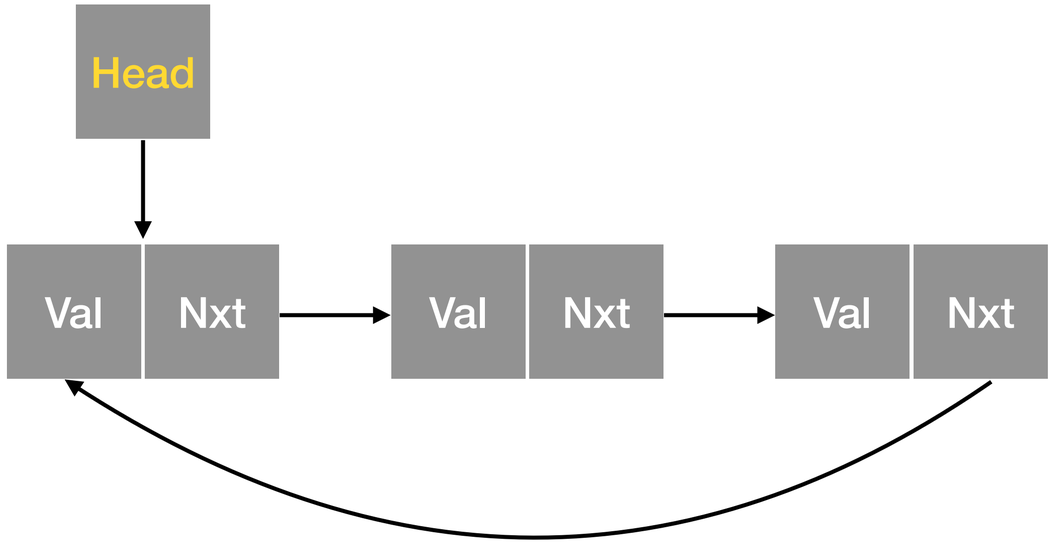
Complexity of Operations in Linked List

- Singly Linked List:
  - Traversal of all elements: `O(n)`
  - Insert/Delete from head: `O(1)`
  - Insert/Delete from end: `O(n)`



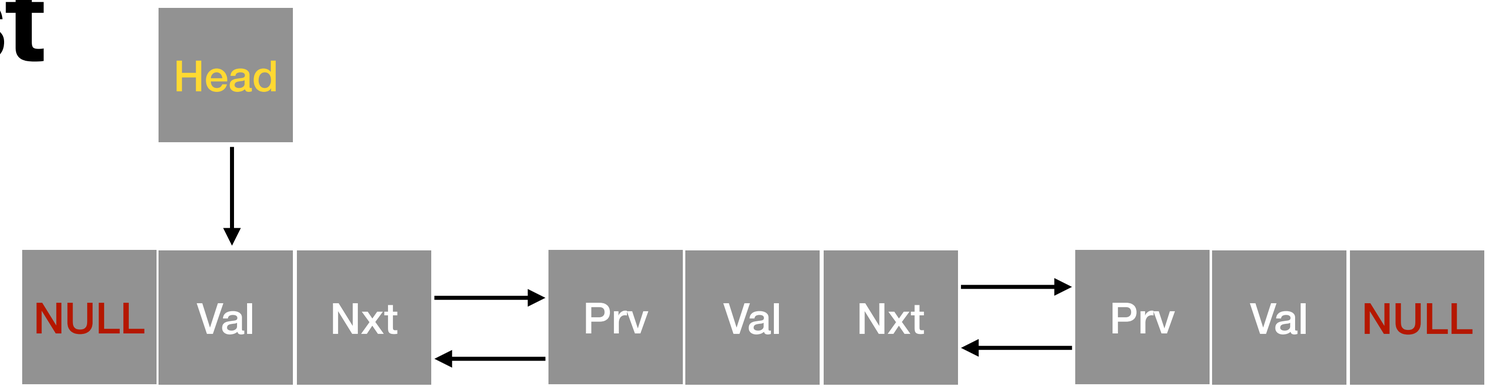
Circular Linked List

- The end node is connected to the first node
- Because it is a cycle — no need to maintain a head ptr



Doubly Linked List

- Pointers both for next and previous nodes
- Efficient insertions/deletions
- Can retrieve previous node in `O(1)`



Linked List: Applications

- In implementing Operating Systems:
  - Round-robin scheduling requires runnable processes to be kept in a linked list.
  - Scheduled process is removed from the head and added to the tail
  - Eg: Linux, FreeRTOS, Cisco IOS for pkt management
- Web Browsers:
  - Eg: Implementing the back button!
- Computer Networks:
  - Implementation of Routers: Pat management, Route planning etc.
- Memory Management: Keeping track of allocated and deallocated mem blocks.