NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

SC1008 C and C++ Programming

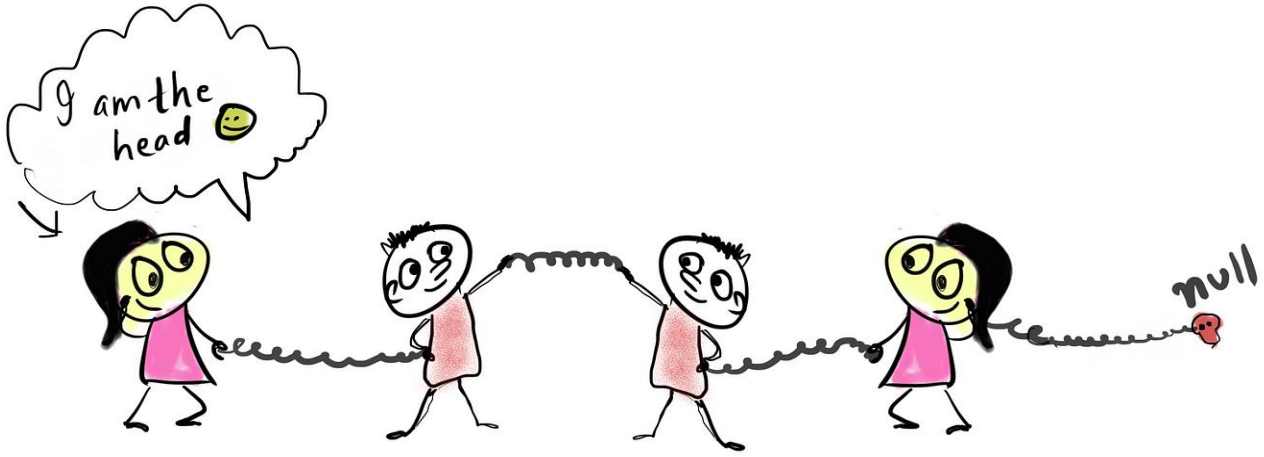
Assistant Professor WANG Yong

yong-wang@ntu.edu.sg

CCDS, Nanyang Technological University

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

Week 9
Linked List



Outline

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- Introduction to Linked List
- Linked List Operations
  - add a node to the end of the list
  - delete a node
  - traverse a linked list
  - delete/destroy a linked list
- Variations of Linked List

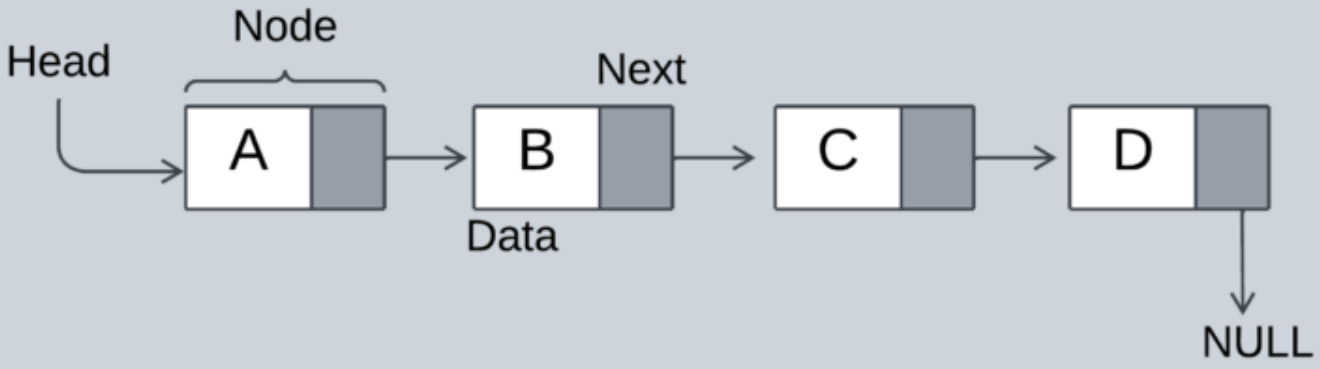
Why Learning Linked List?

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- Linked list is a common Abstract Data Type (ADT) in C++
- Compared with arrays, linked lists have distinct advantages:

- Linked list can easily grow or shrink in size. The programmer do not need to know how many nodes will be in the list in advance

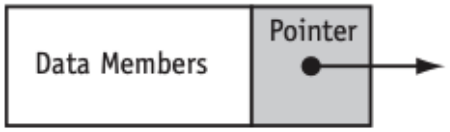
- Node insertion and deletion from a linked list is fast, since none of other nodes have to be moved when a node is inserted into or deleted from a linked list



Introduction to Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- A linked list is a series of connected ***nodes***, where each node is a data structure
- A node contains:
  - data: one or more data fields; can be different data types
  - a pointer that can point to another node

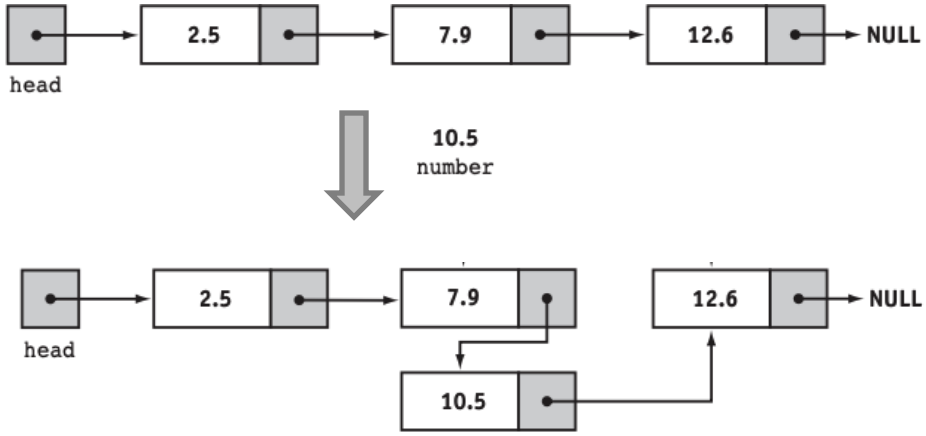


- A linked list can contain 0 or more nodes
- The below example shows a linked list of 3 nodes, with the list head (i.e., a pointer) pointing to the first node and the NULL pointer signifying the end of the list

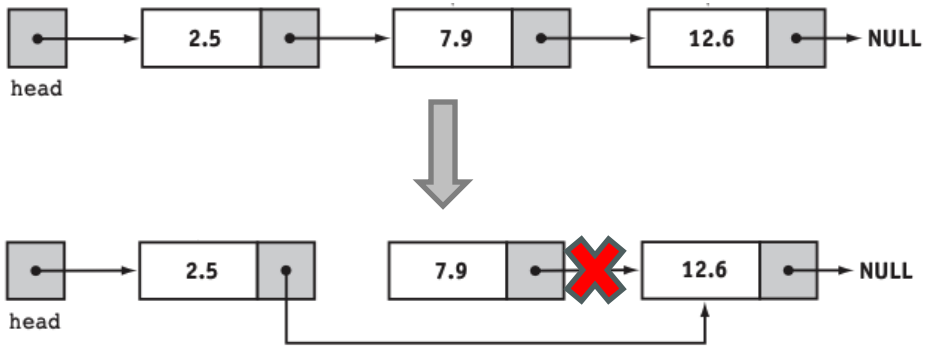
Introduction to Linked List

- The nodes of a linked list are usually dynamically allocated. Nodes can be added to or removed from the linked list, allowing the linked list to grow or shrink in size as the program runs

Adding a node



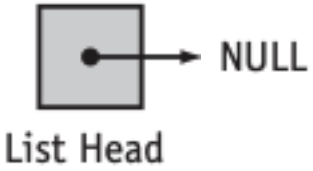
Deleting a node



An Empty Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- If a list currently contains 0 nodes, it is the empty list
- In this case, the list head points to a NULL pointer



NULL** vs. **nullptr :

- C++ inherited NULL from C. NULL is often defined as 0 or (void*) 0, indicating that a pointer does not point to a valid memory location
- NULL can be implicitly converted to integer, potentially leading to unexpected errors
- C++ 11 introduces nullptr with its own type (std::nullptr_t)

Make **nullptr** your default choice for representing null pointers in C++

Define a Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

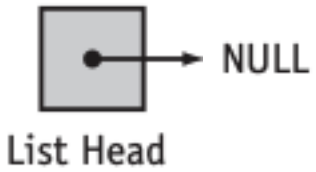
- Declare a node:

```
struct Node
{
        double value; //It can be other data types
        Node *next;
};
```

No memory is allocated at this time

- Define a pointer to be used as the list head and initialize it to nullptr

```
Node* head = nullptr;
```



Outline

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- Introduction to Linked List
- Linked List Operations
  - add a node to the end of the list
  - delete a node
  - traverse a linked list
  - delete/destroy a linked list
- Variations of Linked List

Two Common Pointer Bugs in implementing linked list operations

- Attempting to dereference a pointer via `*p` or `p->` when `p==NULL` or `p == nullptr`
- Attempting to dereference a pointer via `*p` or `p->` when `p` is not properly initialized

- NOTE: this error does not cause a syntax error, but instead causes errors:
  - Bus Error
  - Segmentation violation
  - Address protection violation

Add a Node at the End of the List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

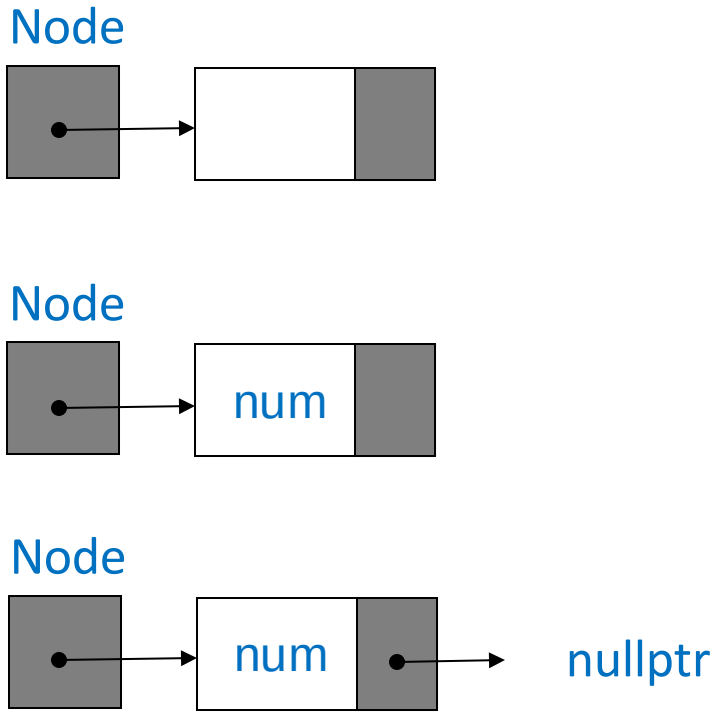
- Basic process:
  - Create the new node
  - Add node to the end of the list:
    - If list is empty, set head pointer to this node
    - Else,
      - traverse the list to the end
      - set pointer of last node to point to new node
- It is the basic operation for creating a linked list

Create a New Node

- Allocate memory for the new node:
  Node = new Node;
- Initialize the contents of the node:

  Node->value = num;
- Set the pointer field to `nullptr`:

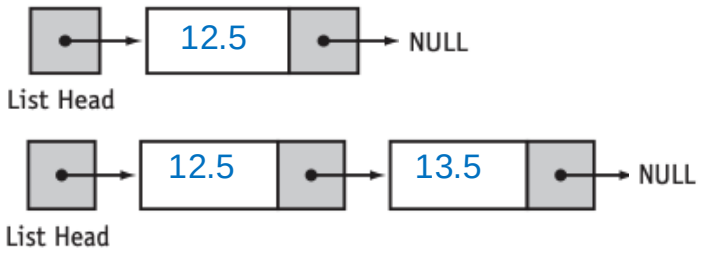
  Node->next = nullptr;



Add a Node at the End of the List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

```
#include <iostream>
using namespace std;
struct Node {
  double value; // Can be any data type
  Node* next;
};

void insertNode2ListEnd(Node*& head, double newValue) {
  Node* newNode = new Node;
  newNode->value = newValue;
  newNode->next = nullptr; // New node is the last node

  if (head == nullptr) { // The list is empty
    head = newNode;
  }
  else {// The list is not empty
    Node* temp = head;
    while (temp->next != nullptr) {
      temp = temp->next;
    }
    temp->next = newNode; // Link last node to new node
  }
}

int main() {
  Node* head = nullptr; // Create the head pointer
  insertNode2ListEnd(head, 12.5);
  insertNode2ListEnd(head, 13.5);
  cout<<"1st Node: "<< head->value <<endl;
  cout<<"2nd Node: "<< head->next->value<<endl;
  return 0;
}
```



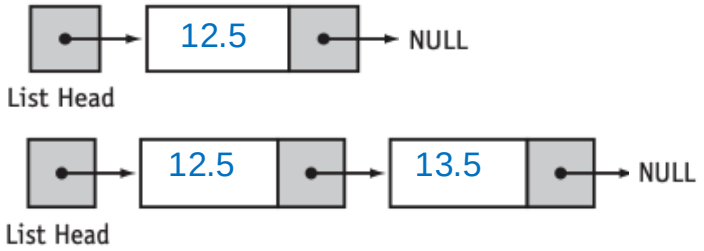
Program output:

```
1st Node: 12.5
2nd Node: 13.5
```

Add a Node at the End of the List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

```
#include <iostream>
using namespace std;
struct Node {
  double value; // Can be any data type
  Node* next;
};

void insertNode2ListEnd(Node*& head, double newValue) {
  Node* newNode = new Node;
  newNode->value = newValue;
  newNode->next = nullptr; // New node is the last node

  if (head == nullptr) { // The list is empty
    head = newNode;
  }
  else {// The list is not empty
    Node* temp = head;
    while (temp->next != nullptr) {
      temp = temp->next;
    }
    temp->next = newNode; // Link last node to new node
  }
}

int main() {
  Node* head = nullptr; // Create the head pointer
  insertNode2ListEnd(head, 12.5);
  insertNode2ListEnd(head, 13.5);
  cout<<"1st Node: "<< head->value <<endl;
  cout<<"2nd Node: "<< head->next->value<<endl;
  return 0;
}
```



One common bug here: use Node* head instead of Node*& head!

A Reference to Pointer

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- Node*& head *vs.* Node* head

  - A reference to a pointer: Node*& head

  - A regular pointer: Node* head

- A reference to a pointer can ensure that any modifications to head within the function can affect the original linked list

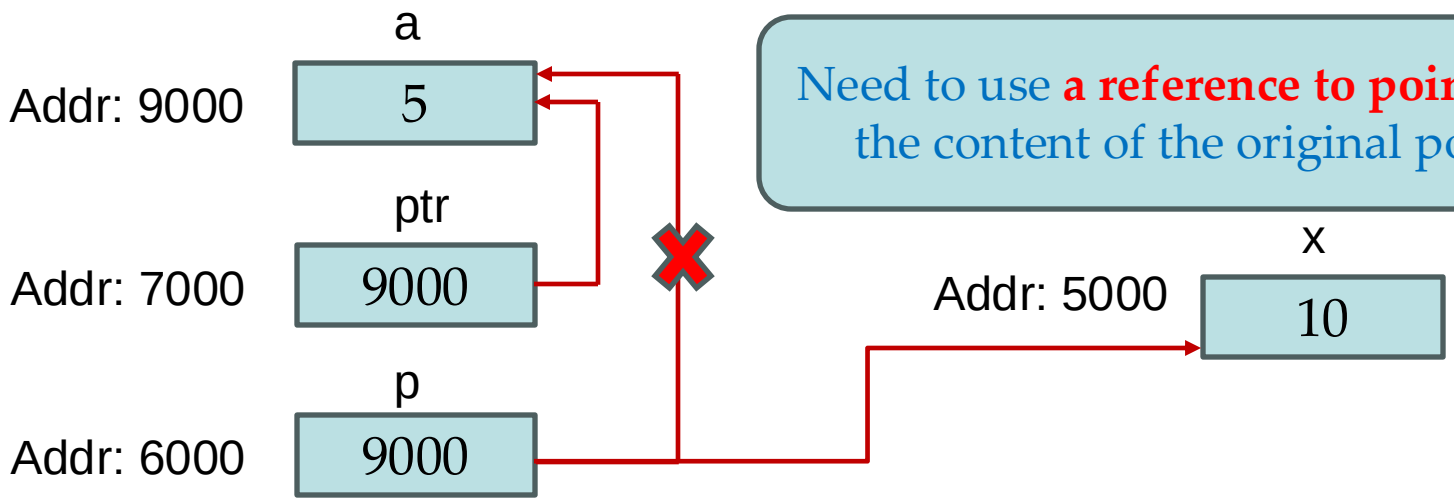
A Reference to Pointer vs a Regular Pointer

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

```
#include <iostream>
using namespace std;
void modifyPointer(int* p) {
  int x = 10;
  p = &x; // This only modifies the local copy of pointer p, not the original pointer p in main
}

int main() {
  int a = 5;
  int* ptr = &a;

  cout << "Before modifyPointer: " << *ptr << endl;
  modifyPointer(ptr);
  cout << "After modifyPointer: " << *ptr << endl; //Still 5
  return 0;
}
```

Program output:

```
Before modifyPointer: 5
After modifyPointer: 5
```



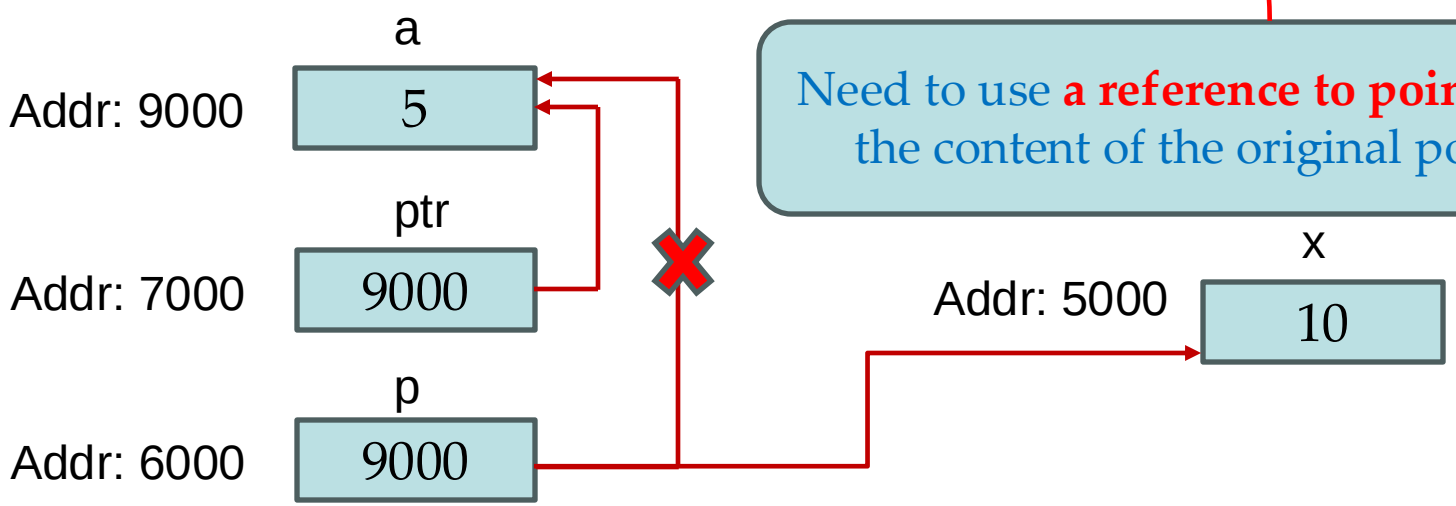
Need to use **a reference to pointer** to change the content of the original pointer here!

A Reference to Pointer vs a Regular Pointer

```
#include <iostream>
using namespace std;
void modifyPointer(int* p) {
  int x = 10;
  p = &x; // This only modifies the local copy of pointer p, not the original pointer p in main
}

int main() {
  int a = 5;
  int* ptr = &a;

  cout << "Before modifyPointer: " << *ptr << endl;
  modifyPointer(ptr);
  cout << "After modifyPointer: " << *ptr << endl; //Still 5
  return 0;
}
```

Program output:

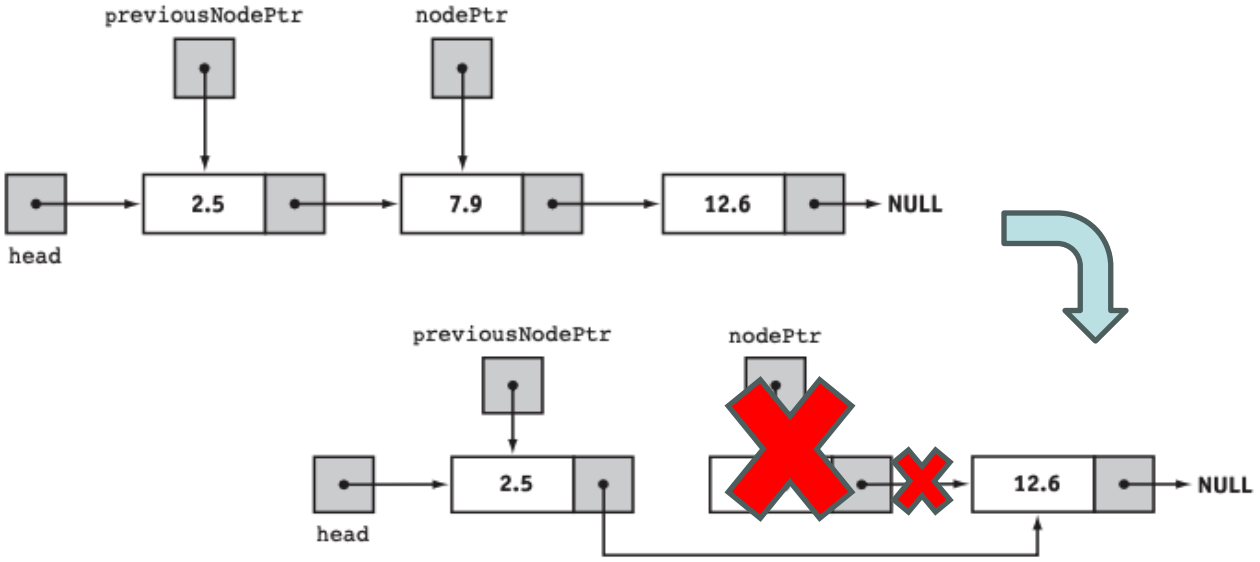
```
Before modifyPointer: 5
After modifyPointer: 5
```



Need to use **a reference to pointer** to change the content of the original pointer here!

Delete a Node

- Used to remove a node from a linked list
- If list uses dynamic memory, then delete node from memory
- Requires two pointers: one to locate the node to be deleted, one to point to the node before the node to be deleted
- Example: Suppose we want to delete the node of 7.9 from the following linked list



Delete a Node

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- Basic process:

  - Locating the node containing the element to be removed

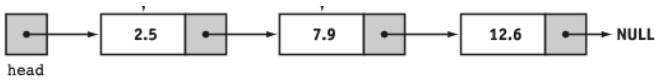
  - Unhooking the node from the list

  - Deleting the memory allocated to the node

```
void remove(Node*& head, double number) {
  Node *nodePtr, *previousNodePtr;

  if (head == nullptr) return; // If the list is empty
  // Determine if the first node is the one to delete
  if (fabs(head->value - number) < 1e-9)
  {
    nodePtr = head;
    head = head->next;
    delete nodePtr;
    nodePtr = nullptr;
  } else
  { // Initialize nodePtr to the list head
    nodePtr = head;

   // Skip nodes whose value member is not number
   while (nodePtr != nullptr && fabs(nodePtr->value -number)>1e-9)
    {
      previousNodePtr = nodePtr;
      nodePtr = nodePtr->next;
    }

// Link the previous node to the node after nodePtr, then delete nodePtr.
    if (nodePtr != nullptr)
    {
      previousNodePtr->next = nodePtr->next;
      delete nodePtr;
      nodePtr = nullptr;
    }
  }
}
```

Delete a Node

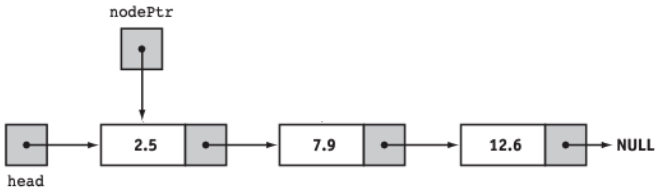
- Basic process:




```
void remove(Node*& head, double number) {
  Node *nodePtr, *previousNodePtr;

  if (head == nullptr) return; // If the list is empty
  // Determine if the first node is the one to delete
  if (fabs(head->value - number) < 1e-9)
  {
    nodePtr = head;
    head = head->next;
    delete nodePtr;
    nodePtr = nullptr;
  } else
  { // Initialize nodePtr to the list head
    nodePtr = head;

   // Skip nodes whose value member is not number
   while (nodePtr != nullptr && fabs(nodePtr->value -number)>1e-9)
    {
      previousNodePtr = nodePtr;
      nodePtr = nodePtr->next;
    }

// Link the previous node to the node after nodePtr, then delete nodePtr.
    if (nodePtr != nullptr)
    {
      previousNodePtr->next = nodePtr->next;
      delete nodePtr;
      nodePtr = nullptr;
    }
  }
}
```

Delete a Node

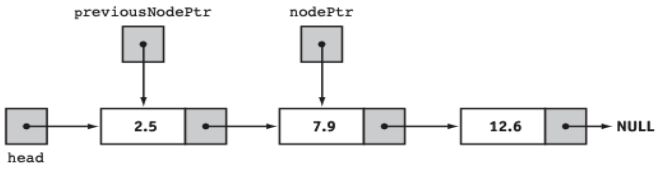
- Basic process:




```
void remove(Node*& head, double number) {
  Node *nodePtr, *previousNodePtr;

  if (head == nullptr) return; // If the list is empty
  // Determine if the first node is the one to delete
  if (fabs(head->value - number) < 1e-9)
  {
    nodePtr = head;
    head = head->next;
    delete nodePtr;
    nodePtr = nullptr;
  } else
  { // Initialize nodePtr to the list head
    nodePtr = head;

   // Skip nodes whose value member is not number
   while (nodePtr != nullptr && fabs(nodePtr->value -number)>1e-9)
    {
      previousNodePtr = nodePtr;
      nodePtr = nodePtr->next;
    }

// Link the previous node to the node after nodePtr, then delete nodePtr.
    if (nodePtr != nullptr)
    {
      previousNodePtr->next = nodePtr->next;
      delete nodePtr;
      nodePtr = nullptr;
    }
  }
}
```

Delete a Node

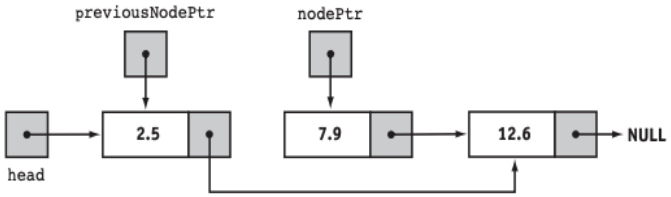
- Basic process:




```
void remove(Node*& head, double number) {
  Node *nodePtr, *previousNodePtr;

  if (head == nullptr) return; // If the list is empty
  // Determine if the first node is the one to delete
  if (fabs(head->value - number) < 1e-9)
  {
    nodePtr = head;
    head = head->next;
    delete nodePtr;
    nodePtr = nullptr;
  } else
  { // Initialize nodePtr to the list head
    nodePtr = head;

   // Skip nodes whose value member is not number
   while (nodePtr != nullptr && fabs(nodePtr->value -number)>1e-9)
    {
      previousNodePtr = nodePtr;
      nodePtr = nodePtr->next;
    }

// Link the previous node to the node after nodePtr, then delete nodePtr.
    if (nodePtr != nullptr)
    {
      previousNodePtr->next = nodePtr->next;
      delete nodePtr;
      nodePtr = nullptr;
    }
  }
}
```

Delete a Node

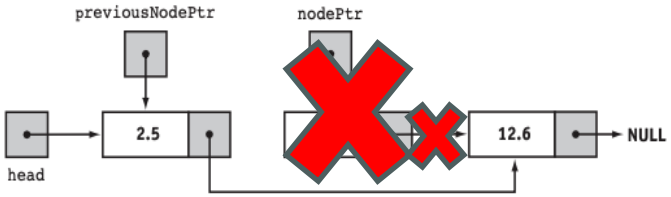
- Basic process:




```
void remove(Node*& head, double number) {
  Node *nodePtr, *previousNodePtr;

  if (head == nullptr) return; // If the list is empty
  // Determine if the first node is the one to delete
  if (fabs(head->value - number) < 1e-9)
  {
    nodePtr = head;
    head = head->next;
    delete nodePtr;
    nodePtr = nullptr;
  } else
  { // Initialize nodePtr to the list head
    nodePtr = head;

   // Skip nodes whose value member is not number
   while (nodePtr != nullptr && fabs(nodePtr->value -number)>1e-9)
    {
      previousNodePtr = nodePtr;
      nodePtr = nodePtr->next;
    }

// Link the previous node to the node after nodePtr, then delete nodePtr.
    if (nodePtr != nullptr)
    {
      previousNodePtr->next = nodePtr->next;
      delete nodePtr;
      nodePtr = nullptr;
    }
  }
}
```

Delete a Node

- Basic process:




```
void remove(Node*& head, double number) {
  Node *nodePtr, *previousNodePtr;

  if (head == nullptr) return; // If the list is empty
  // Determine if the first node is the one to delete
  if (fabs(head->value - number) < 1e-9)
  {
    nodePtr = head;
    head = head->next;
    delete nodePtr;
    nodePtr = nullptr;
  } else
  { // Initialize nodePtr to the list head
    nodePtr = head;

   // Skip nodes whose value member is not number
   while (nodePtr != nullptr && fabs(nodePtr->value -number)>1e-9)
    {
      previousNodePtr = nodePtr;
      nodePtr = nodePtr->next;
    }

// Link the previous node to the node after nodePtr, then delete nodePtr.
    if (nodePtr != nullptr)
    {
      previousNodePtr->next = nodePtr->next;
      delete nodePtr;
      nodePtr = nullptr;
    }
  }
}
```

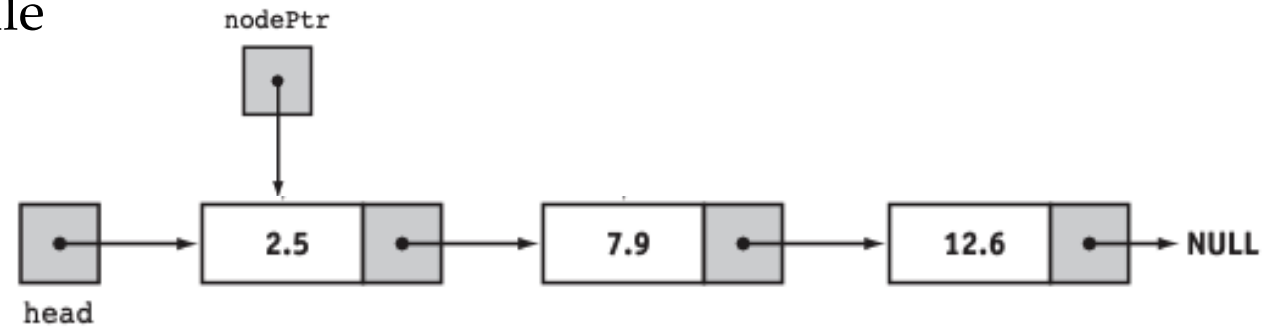
Outline

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- Introduction to Linked List
- Linked List Operations
  - add a node to the end of the list
  - delete a node
  - traverse a linked list
  - delete/destroy a linked list
- Variations of Linked List

Traverse a Linked List

- Visit each node in a linked list: display contents, validate data, etc.
- Basic process:
  - set a pointer to the contents of the head pointer
  - while pointer is not `nullptr`
    - process data
    - go to the next node by setting the pointer to the pointer field of the current node in the list
  - end while



nodePtr points to the node containing *2.5*, then the node containing *7.9*, then the node containing *12.6*, then points to `nullptr`, and the list traversal stops

Traverse a Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

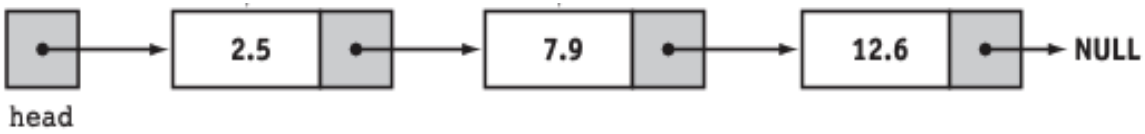
- Example: printList()
- Basic process:
  - set a pointer to the contents of the head pointer
  - while pointer is not *nullptr*
    - process data
    - go to the next node by setting the pointer to the pointer field of the current node in the list
  - end while

```
void printList(Node* head) {
  Node* current = head;// Start at the head of the list

  while (current) { //Equivalent to "current != nullptr"
    cout << current->value << " -> ";
    current = current->next;
  }
  cout << "NULL" << endl;
}
```

Traverse a Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

```
void printList(Node* head) {
  Node* current = head;// Start at the head of the list

  while (current) { //Equivalent to "current != nullptr"
    cout << current->value << " -> ";
    current = current->next;
  }
  cout << "NULL" << endl;
}
```

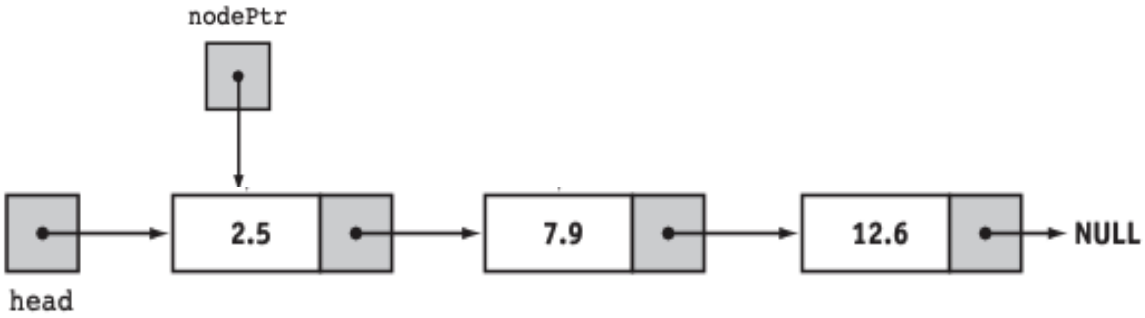


Program output:

Traverse a Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

```
void printList(Node* head) {
  Node* current = head;// Start at the head of the list

  while (current) { //Equivalent to "current != nullptr"
    cout << current->value << " -> ";
    current = current->next;
  }
  cout << "NULL" << endl;
}
```

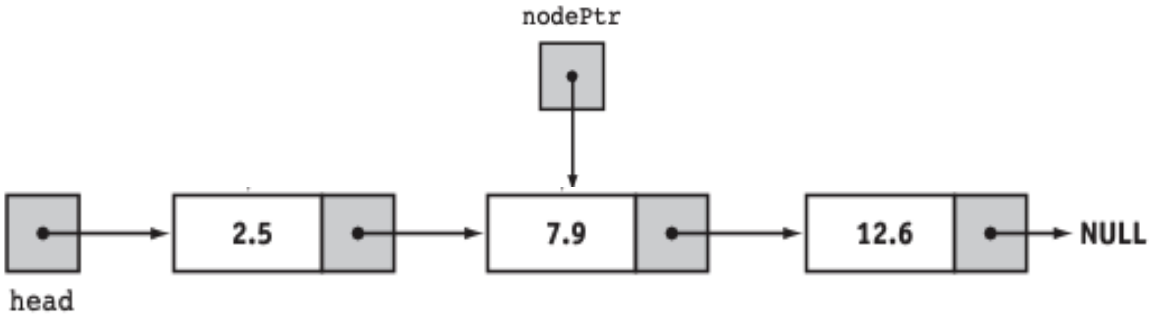


Program output:

```
2.5 ->
```

Traverse a Linked List

```
void printList(Node* head) {
  Node* current = head;// Start at the head of the list

  while (current) { //Equivalent to "current != nullptr"
    cout << current->value << " -> ";
    current = current->next;
  }
  cout << "NULL" << endl;
}
```



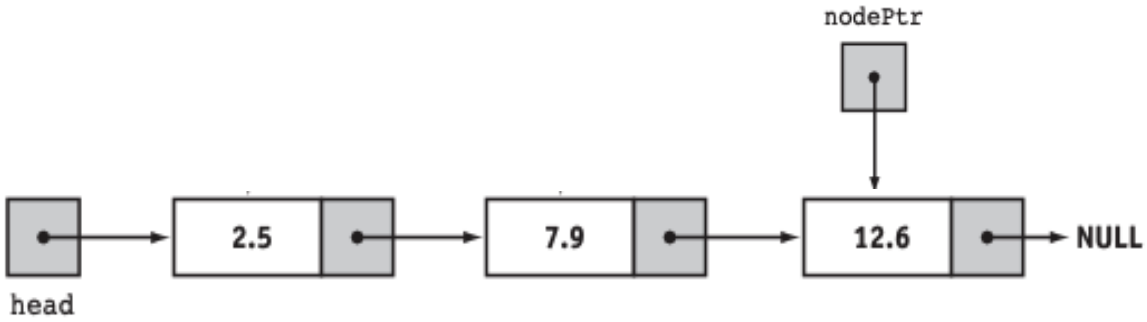
Program output:

```
2.5 ->

2.5 -> 7.9 ->
```

Traverse a Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

```
void printList(Node* head) {
  Node* current = head;// Start at the head of the list

  while (current) { //Equivalent to "current != nullptr"
    cout << current->value << " -> ";
    current = current->next;
  }
  cout << "NULL" << endl;
}
```



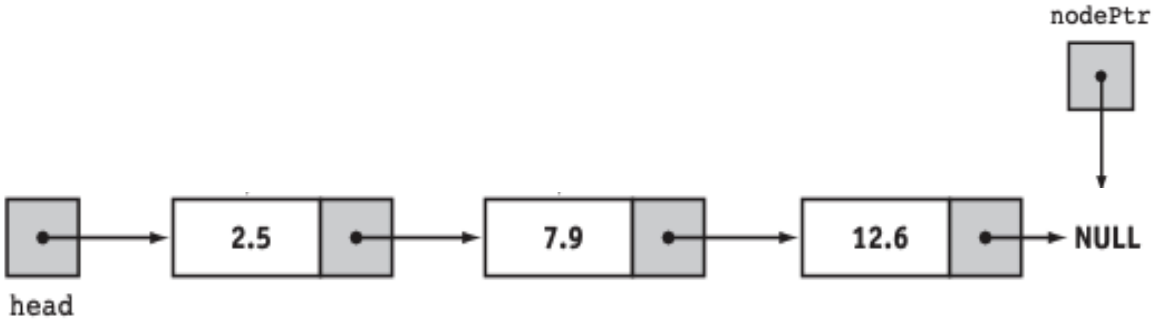
Program output:

```
2.5 ->

2.5 -> 7.9 ->

2.5 -> 7.9 -> 12.6 ->
```

Traverse a Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

```
void printList(Node* head) {
  Node* current = head;// Start at the head of the list

  while (current) { //Equivalent to "current != nullptr"
    cout << current->value << " -> ";
    current = current->next;
  }
  cout << "NULL" << endl;
}
```



Program output:

2.5 ->

2.5 -> 7.9 ->

2.5 -> 7.9 -> 12.6 ->

2.5 -> 7.9 -> 12.6 -> NULL

Delete/Destroy a Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- The nodes of a linked list are dynamically allocated
- When deleting or destroying a linked list, we must remove all nodes used in the list, where we need to traverse the linked list

- Basic process:
  - set a pointer `nodePtr` to the contents of the head pointer
  - while pointer is not `nullptr`
    - use another pointer `garbage` to keep track of the node to be deleted
    - go to the next node by setting the pointer `nodePtr` to the pointer field of the current node in the list
    - deallocate the memory of the current node
  - end while

Delete/Destroy a Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

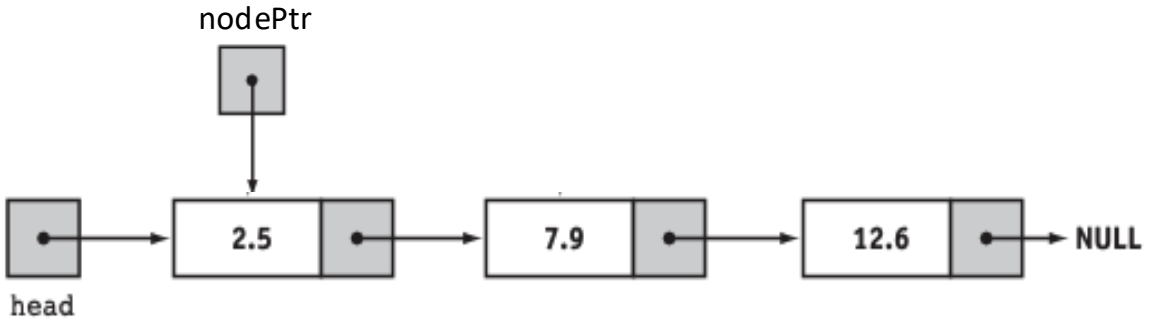
- Basic process:
  - set a pointer *nodePtr* to the contents of the head pointer
  - while pointer is not *nullptr*
    - use another pointer *garbage* to keep track of the node to be deleted
    - go to the next node by setting the pointer *nodePtr* to the pointer field of the current node in the list
    - deallocate the memory of the current node
  - end while

```
void destroyList(Node*& head)
{
   Node *nodePtr = head; // Start at head of list
   Node *garbage = nullptr;
   while (nodePtr != nullptr)
   {
      // garbage keeps track of node to be deleted
      garbage = nodePtr;
      // Move on to the next node, if any
      nodePtr = nodePtr->next;
      // Delete the "garbage" node
      delete garbage;
      garbage = nullptr;
   }
   head = nullptr;
}
```

Delete/Destroy a Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

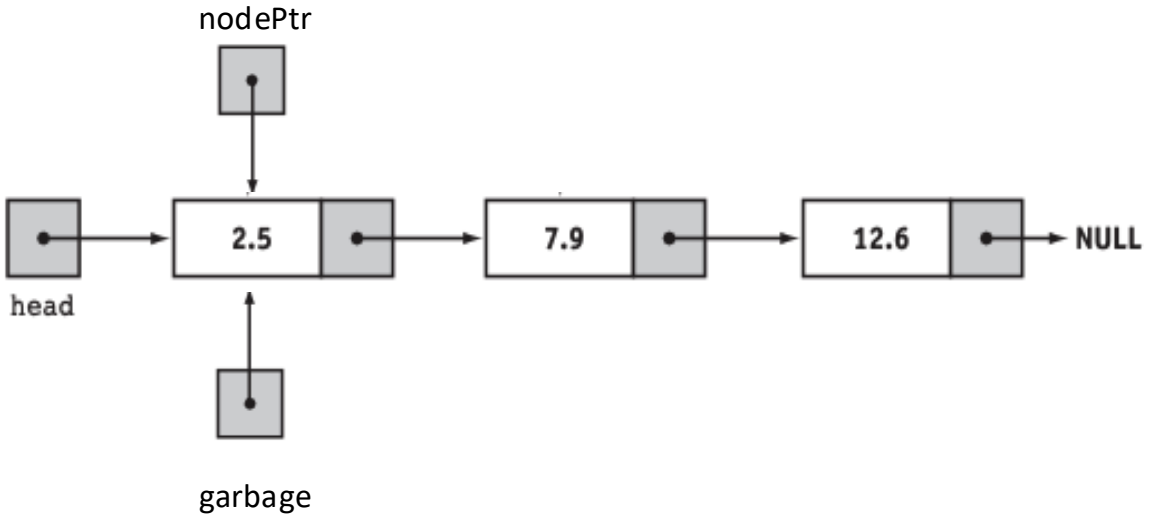
```
void destroyList(Node*& head)
{
   Node *nodePtr = head; // Start at head of list
   Node *garbage = nullptr;
   while (nodePtr != nullptr)
   {
      // garbage keeps track of node to be deleted
      garbage = nodePtr;
      // Move on to the next node, if any
      nodePtr = nodePtr->next;
      // Delete the "garbage" node
      delete garbage;
      garbage = nullptr;
   }
   head = nullptr;
}
```



Delete/Destroy a Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

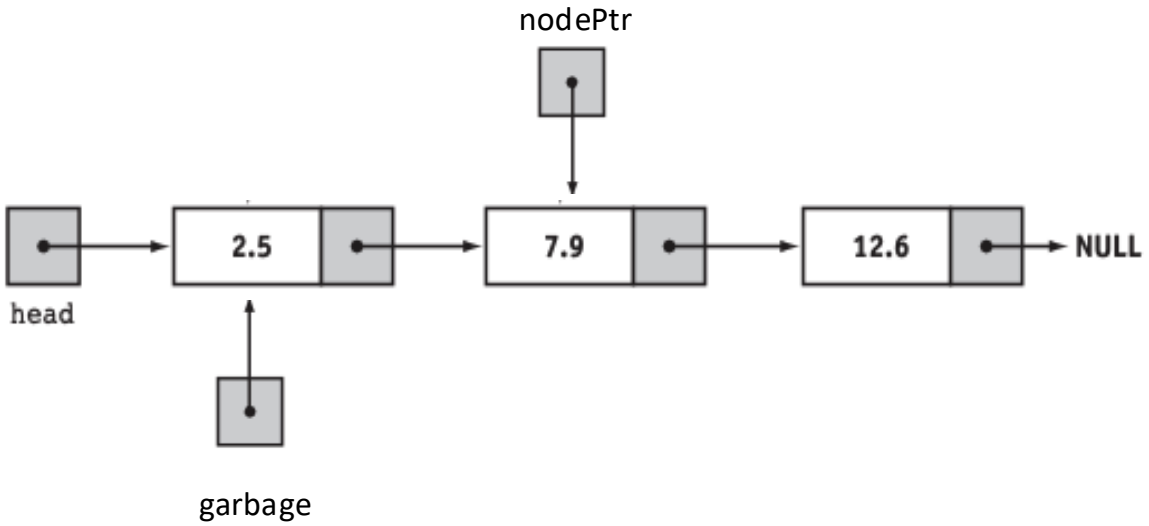
```
void destroyList(Node*& head)
{
   Node *nodePtr = head; // Start at head of list
   Node *garbage = nullptr;
   while (nodePtr != nullptr)
   {
      // garbage keeps track of node to be deleted
      garbage = nodePtr;
      // Move on to the next node, if any
      nodePtr = nodePtr->next;
      // Delete the "garbage" node
      delete garbage;
      garbage = nullptr;
   }
   head = nullptr;
}
```



Delete/Destroy a Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

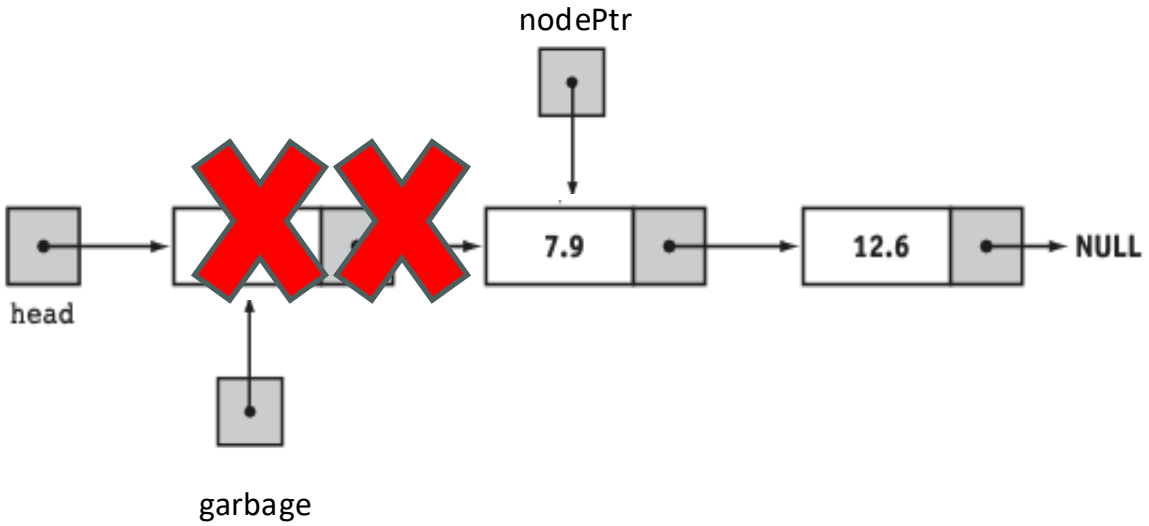
```
void destroyList(Node*& head)
{
   Node *nodePtr = head; // Start at head of list
   Node *garbage = nullptr;
   while (nodePtr != nullptr)
   {
      // garbage keeps track of node to be deleted
      garbage = nodePtr;
      // Move on to the next node, if any
      nodePtr = nodePtr->next;
      // Delete the "garbage" node
      delete garbage;
      garbage = nullptr;
   }
   head = nullptr;
}
```



Delete/Destroy a Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

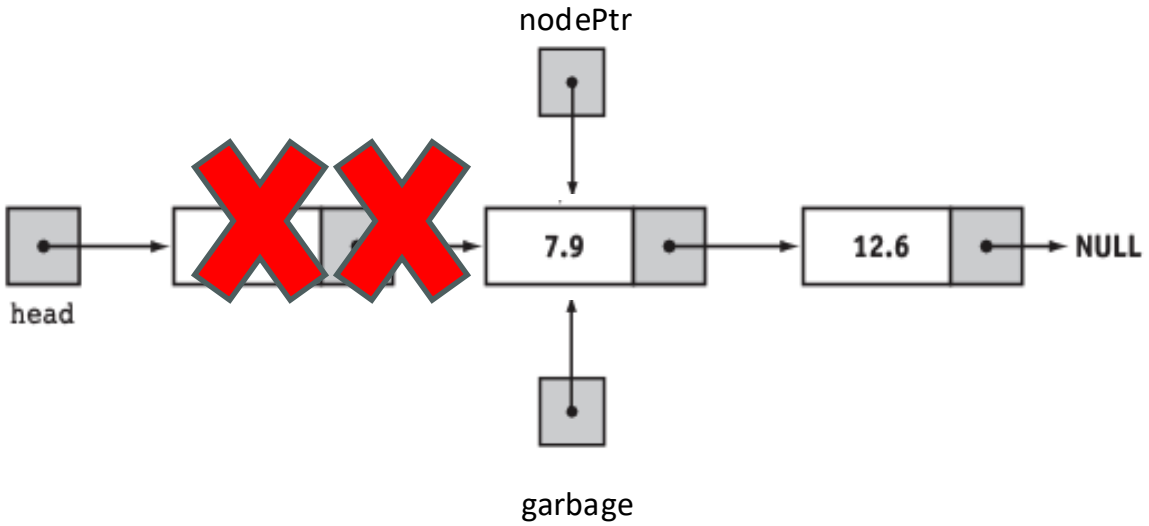
```
void destroyList(Node*& head)
{
   Node *nodePtr = head; // Start at head of list
   Node *garbage = nullptr;
   while (nodePtr != nullptr)
   {
      // garbage keeps track of node to be deleted
      garbage = nodePtr;
      // Move on to the next node, if any
      nodePtr = nodePtr->next;
      // Delete the "garbage" node
      delete garbage;
      garbage = nullptr;
   }
   head = nullptr;
}
```



Delete/Destroy a Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

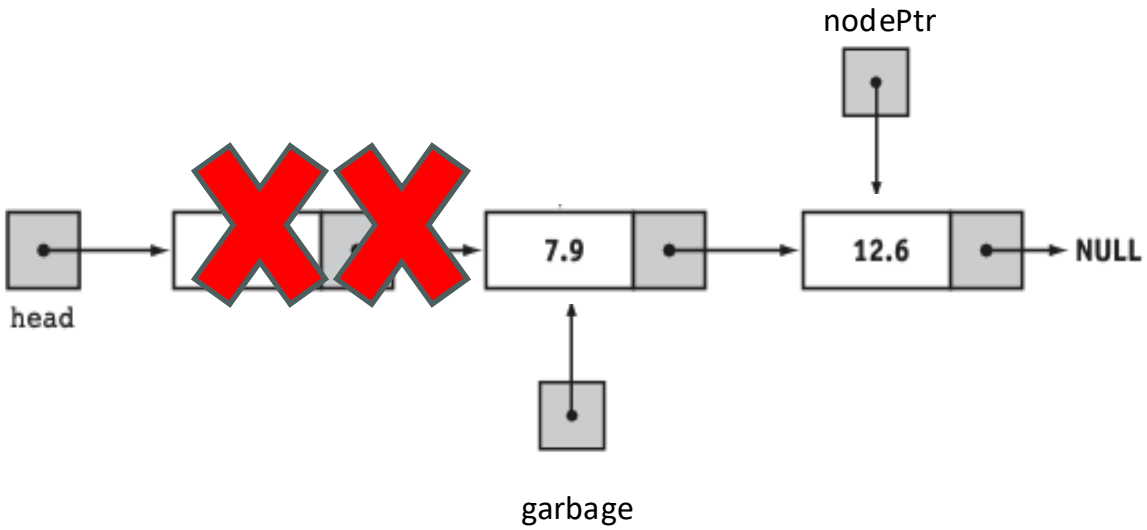
```
void destroyList(Node*& head)
{
   Node *nodePtr = head; // Start at head of list
   Node *garbage = nullptr;
   while (nodePtr != nullptr)
   {
      // garbage keeps track of node to be deleted
      garbage = nodePtr;
      // Move on to the next node, if any
      nodePtr = nodePtr->next;
      // Delete the "garbage" node
      delete garbage;
      garbage = nullptr;
   }
   head = nullptr;
}
```



Delete/Destroy a Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

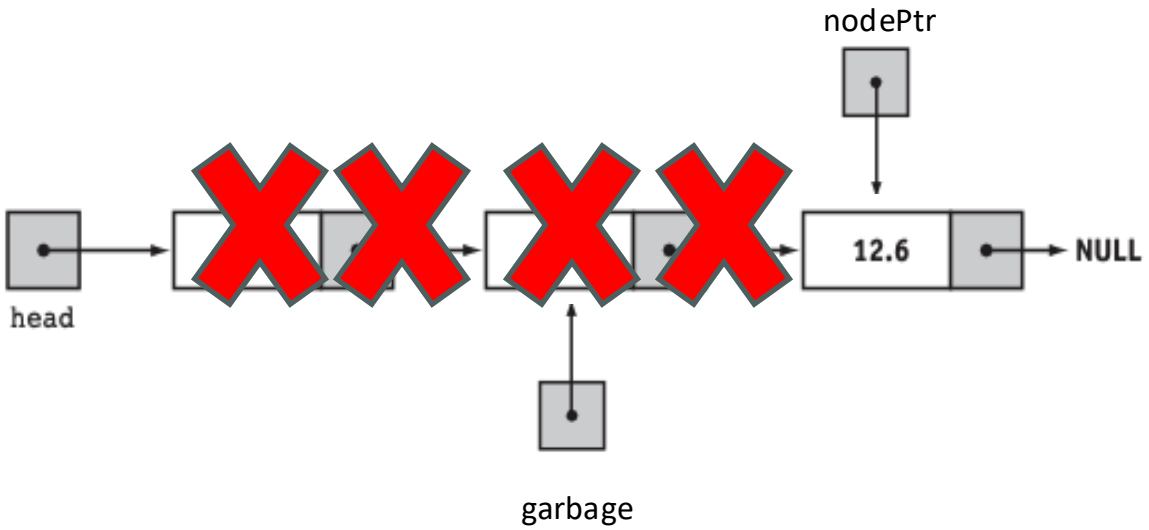
```
void destroyList(Node*& head)
{
   Node *nodePtr = head; // Start at head of list
   Node *garbage = nullptr;
   while (nodePtr != nullptr)
   {
      // garbage keeps track of node to be deleted
      garbage = nodePtr;
      // Move on to the next node, if any
      nodePtr = nodePtr->next;
      // Delete the "garbage" node
      delete garbage;
      garbage = nullptr;
   }
   head = nullptr;
}
```



Delete/Destroy a Linked List

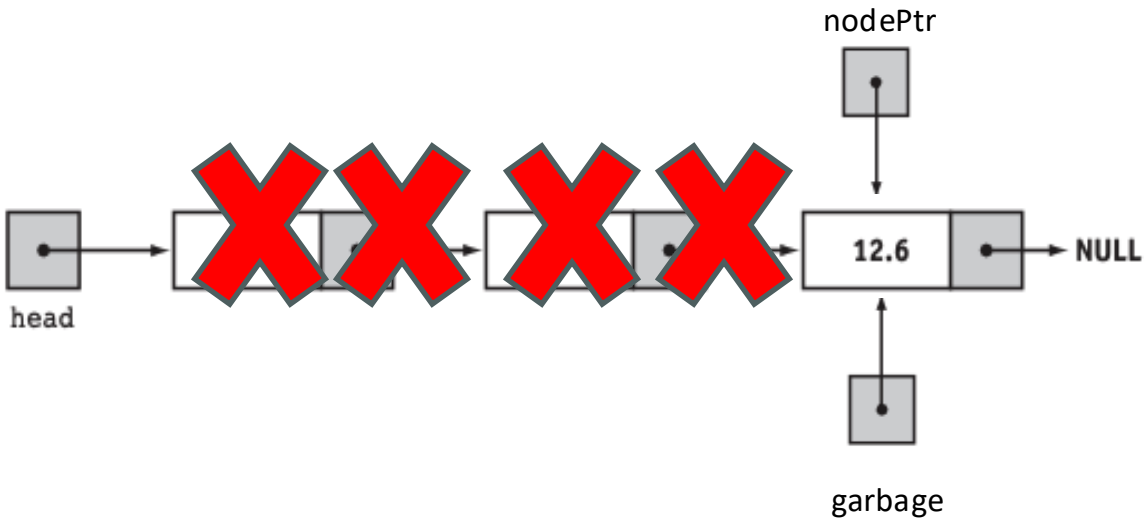
NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

```
void destroyList(Node*& head)
{
   Node *nodePtr = head; // Start at head of list
   Node *garbage = nullptr;
   while (nodePtr != nullptr)
   {
      // garbage keeps track of node to be deleted
      garbage = nodePtr;
      // Move on to the next node, if any
      nodePtr = nodePtr->next;
      // Delete the "garbage" node
      delete garbage;
      garbage = nullptr;
   }
   head = nullptr;
}
```



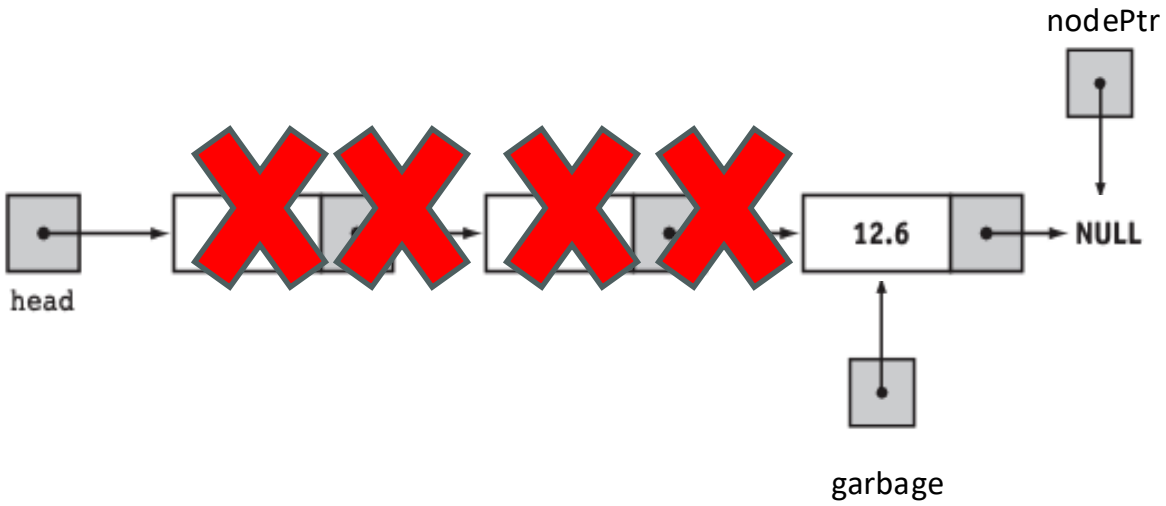
Delete/Destroy a Linked List

```
void destroyList(Node*& head)
{
   Node *nodePtr = head; // Start at head of list
   Node *garbage = nullptr;
   while (nodePtr != nullptr)
   {
      // garbage keeps track of node to be deleted
      garbage = nodePtr;
      // Move on to the next node, if any
      nodePtr = nodePtr->next;
      // Delete the "garbage" node
      delete garbage;
      garbage = nullptr;
   }
   head = nullptr;
}
```



Delete/Destroy a Linked List

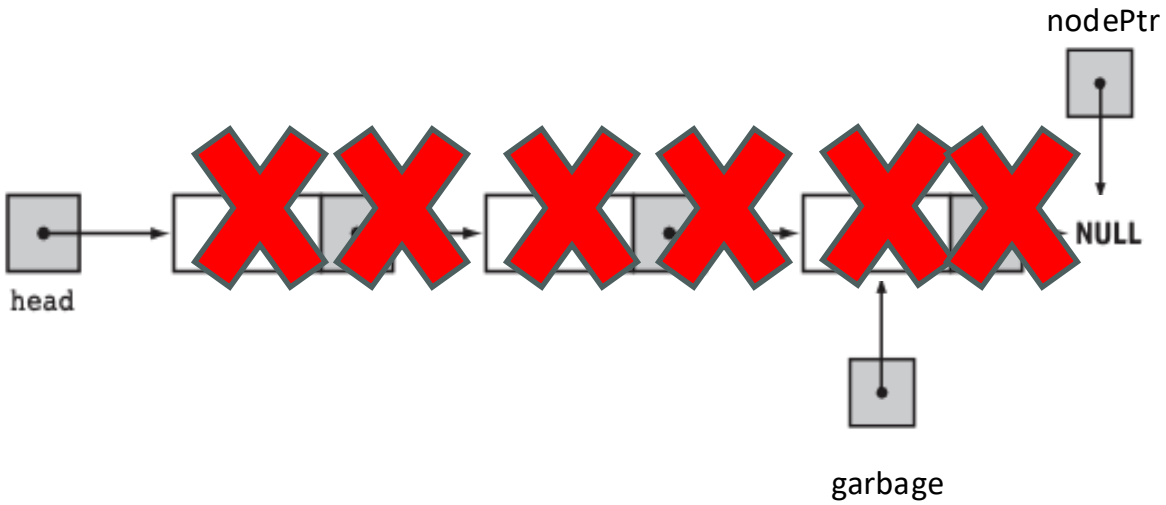
```
void destroyList(Node*& head)
{
   Node *nodePtr = head; // Start at head of list
   Node *garbage = nullptr;
   while (nodePtr != nullptr)
   {
      // garbage keeps track of node to be deleted
      garbage = nodePtr;
      // Move on to the next node, if any
      nodePtr = nodePtr->next;
      // Delete the "garbage" node
      delete garbage;
      garbage = nullptr;
   }
   head = nullptr;
}
```



Delete/Destroy a Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

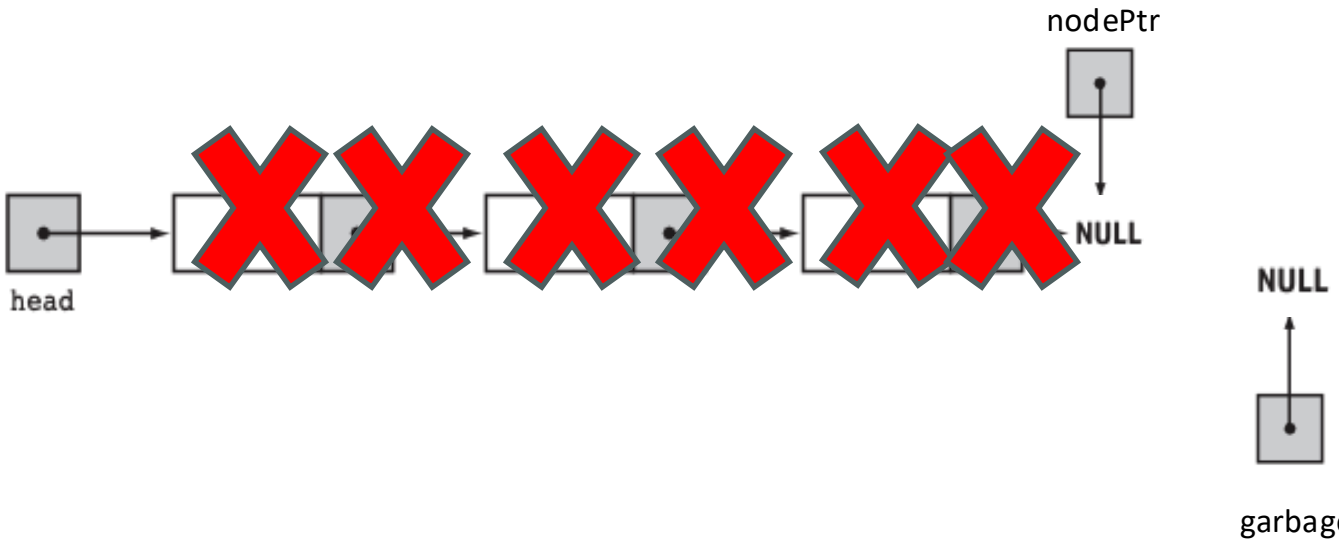
```
void destroyList(Node*& head)
{
   Node *nodePtr = head; // Start at head of list
   Node *garbage = nullptr;
   while (nodePtr != nullptr)
   {
      // garbage keeps track of node to be deleted
      garbage = nodePtr;
      // Move on to the next node, if any
      nodePtr = nodePtr->next;
      // Delete the "garbage" node
      delete garbage;
      garbage = nullptr;
   }
   head = nullptr;
}
```



Delete/Destroy a Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

```
void destroyList(Node*& head)
{
   Node *nodePtr = head; // Start at head of list
   Node *garbage = nullptr;
   while (nodePtr != nullptr)
   {
      // garbage keeps track of node to be deleted
      garbage = nodePtr;
      // Move on to the next node, if any
      nodePtr = nodePtr->next;
      // Delete the "garbage" node
      delete garbage;
      garbage = nullptr;
   }
   head = nullptr;
}
```



Outline

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

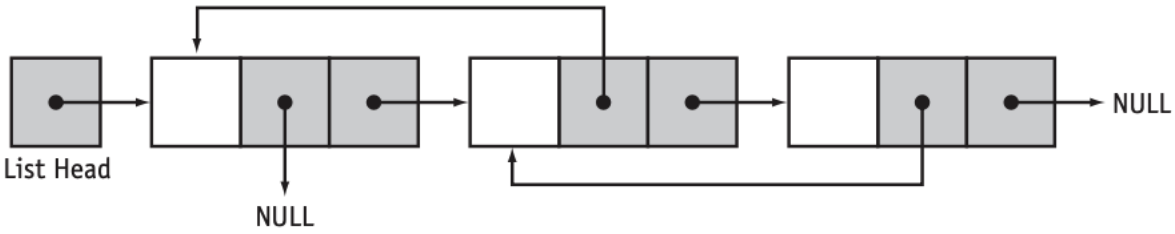
- Introduction to Linked List
- Linked List Operations
  - add a node to the end of the list
  - delete a node
  - traverse a linked list
  - delete/destroy a linked list
- Variations of Linked List

Variations of the Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- There are many ways to link dynamically allocated data structures together. Two variations of the linked list are the ***doubly linked list*** and the ***circular linked list***.

- **Doubly linked list**: unlike the above ***singly-linked list***, where each node is linked to a single other node, each node of a doubly linked list points both the ***next node*** and the ***previous node***.

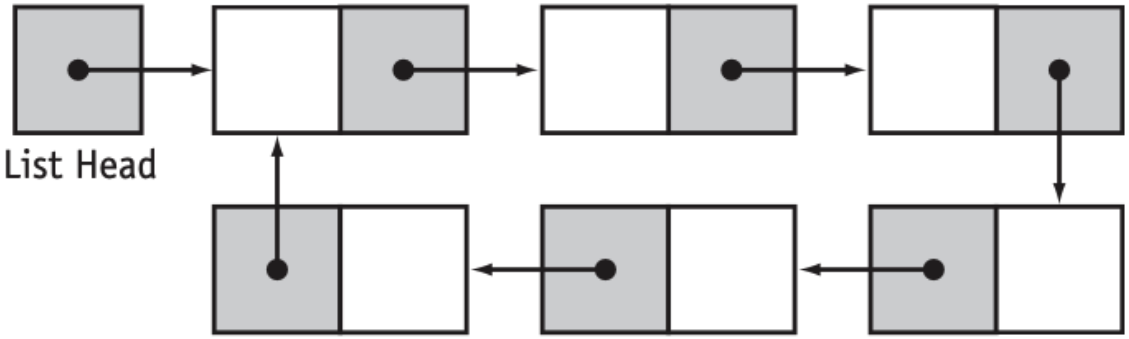


The last node and first node have pointers to the NULL address, which can be used to check if we have reached either end.

Variations of the Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- **Circular linked list**: The last node in this type of linked list points to the first node, not to the NULL address



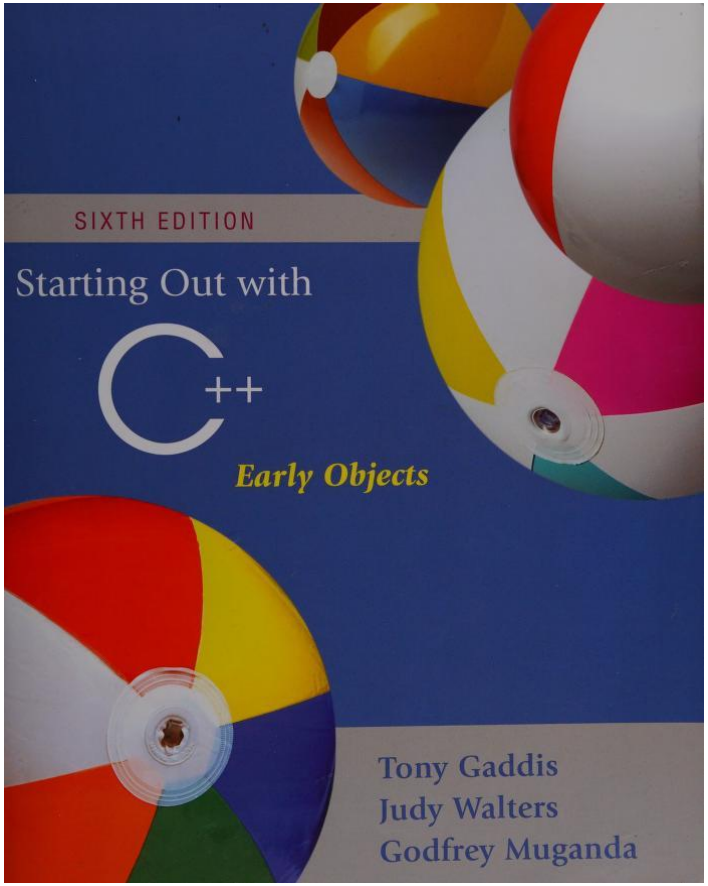
More about Linked List

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- There are other linked list operations, e.g., list copy, list append, compute the length of a linked list, etc., but the basic ideas are more or less the same

- *Better implementation*: define a linked list class!! This is object oriented programming after all

- The Standard Template Library (**STL**) provides a linked list container (more specifically, doubly linked list), which will be introduced in Week 12



References:

[1] Gaddis, Tony, Judy Walters, and Godfrey Muganda. Starting Out with C++ Early Objects, 6th edition. Ch17. Pearson, 2016.

Questions?

Thank You!