# AST20105 Data Structures & Algorithms

## CHAPTER 4 – ARRAYS & LINKED LISTS

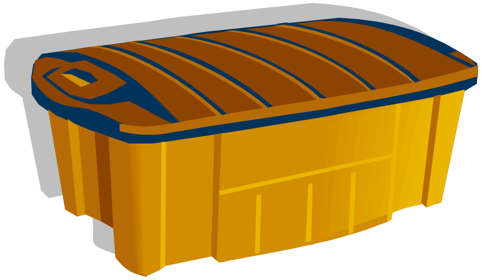
Instructed by Garret Lai

# Before Start

- In programming, what are you going to use for storing data?

# Variables

- Variables are used as storages for calculations.
- It can be defined as a portion of memory to store a determined value.

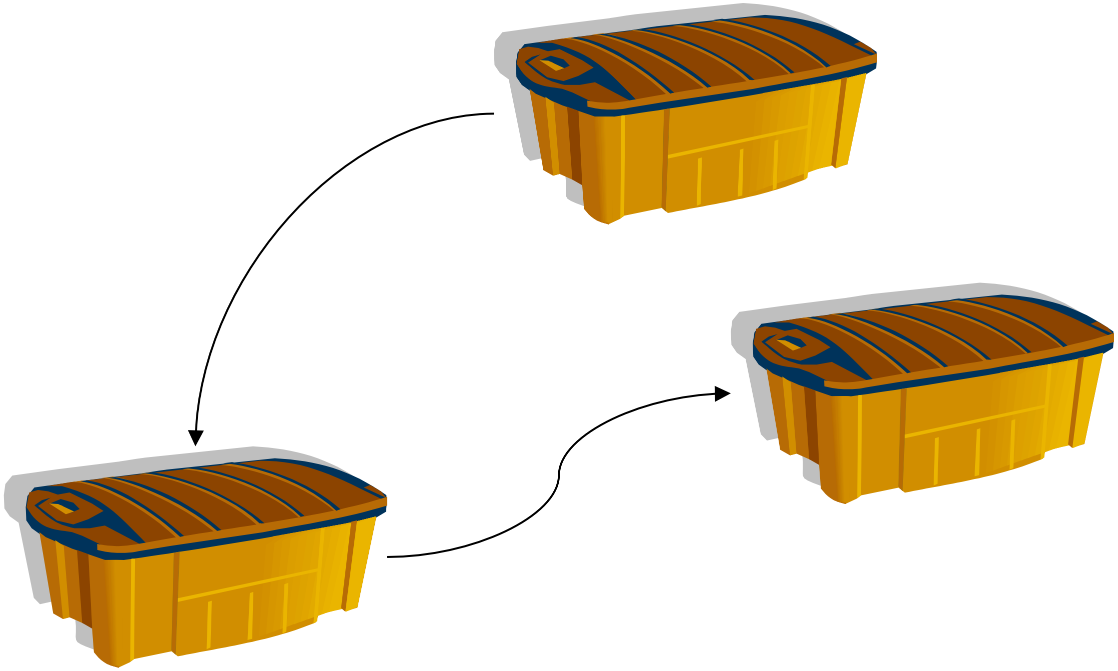


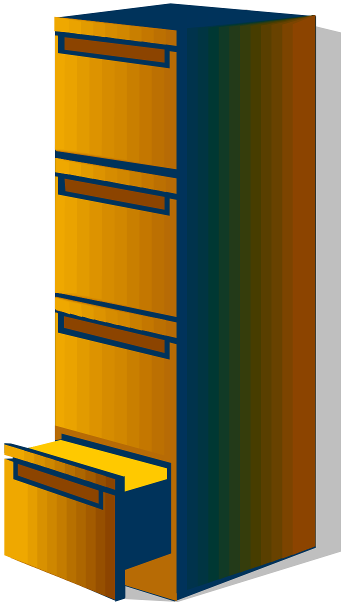
# Data Structure

- Recall, data structure refers to the way of storing and organizing data in a computer so that the data could be accessed and manipulated efficiently

- To make this possible, the following are needed for every data structure:
  - A set of variables / objects
  - A set of functions / operations

# What is Arrays and Linked Lists?





# ARRAYS AND LINKED LISTS

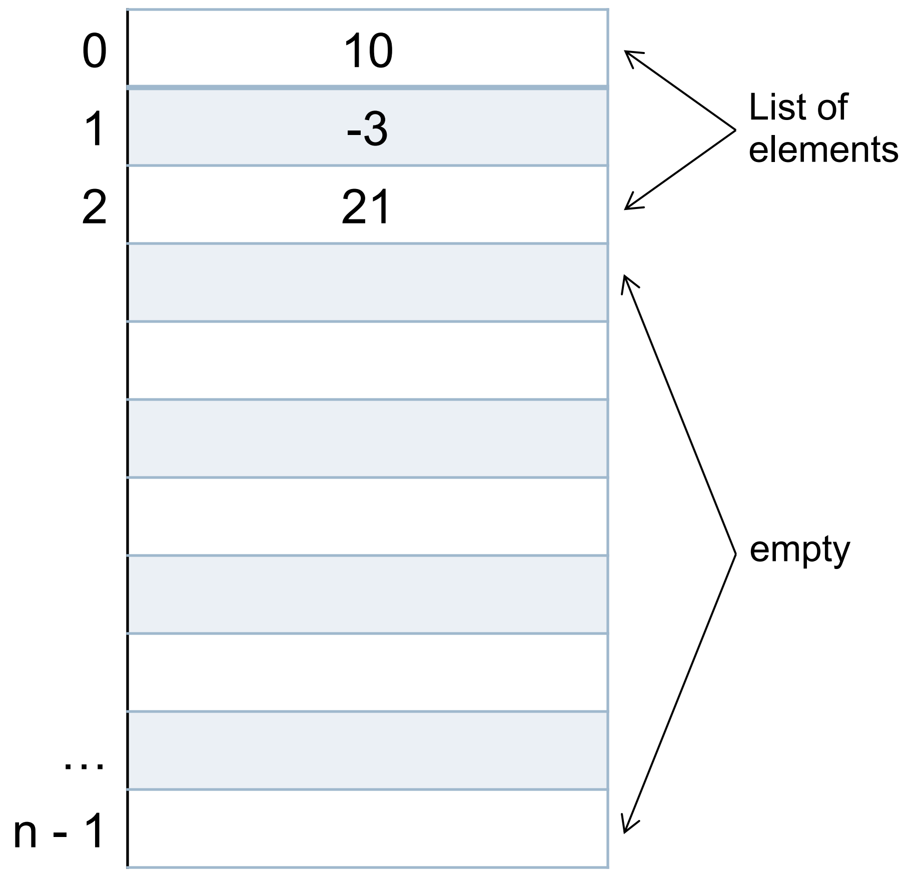
# What is Array?

- Array is a very basic data structure representing a group of similar elements, accessed by index.

- Array is an ordered collection of data contained in a variable and referenced by a single variable name.

# Array

- Array can be used to store a list of values / objects
- The size of array is fixed once it is created (no matter you create it using new or without new in C++)
- Values / objects are stored in contiguous positions in the memory



# PROS

- Array data structure
  - Can be effectively stored inside the computer,
  - Can efficiently return the value of element at certain position (using subscript operator [] in C++)
  - Can search for value / object with binary search if the list is sorted

# CONS

- Array data structure is not completely dynamic.
  - Requires an estimation of the maximum size of the list
  - Many programming languages provides an opportunity to allocate arrays with arbitrary size (dynamically allocated array), but when this space is used up,
    a new array of greater size must be allocated and old data is copied to it.
  - If it cannot be fully utilized, memory space is wasted

# CONS

- Insertion and deletion of element is slow because it requires to shift elements.
  - e.g. insert or delete an element at the very beginning, i.e. position 0.
    - Since all the elements in the list may need to be moved / re-located

# Static Vs Dynamic

- There are two types of arrays, which differ in the method of allocation.

  - **Static array** has constant size and exists all the time, application being executed.

  - **Dynamically allocated array** is created during program run and may be deleted when it is not more needed.

# Static Array In C++

- A typical declaration for a static array in C++ is:

type name [elements];

- where
  - type is a valid type (like int, float...),
  - name is a valid identifier and
  - the elements field (which is always enclosed in square brackets []), specifies how many of these elements the array has to contain.

# Static Array In C++

- For example,

int number[5];

number

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |

# Dynamic Array In C++

- A typical declaration for a dynamic array in C++ is:

pointer = new type [elements];

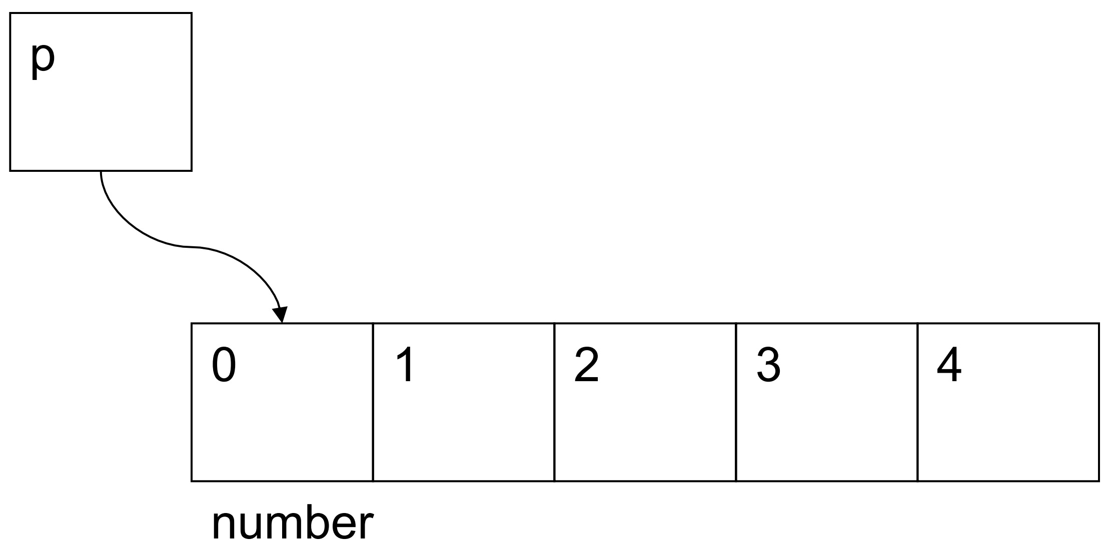
- where
  - type is a valid type (like int, float…),
  - pointer is a variable to store the memory address of the array
  - the elements field (which is always enclosed in square brackets []), specifies how many of these elements the array has to contain.

# DYNAMIC ARRAYS IN C++

- For example,

int* p;

p = new int[5];



# Fixed-size and Dynamic Arrays

- As it mentioned above, arrays can't be resized.
- In this case array is called **fixed-size array.**
- But we can use a simple trick to construct a **dynamic array,** which can be resized.

# Fixed-size and Dynamic Arrays

- The idea is simple.
  - Let us allocate some space for the dynamic array and imaginary divide it into two parts.
  - One part contains the data and the other one is free space.
  - When new element is added, free space is reduced and vice versa.
  - This approach results in overhead for free space, but we have all advantages of arrays and capability of changing size dynamically.

# Fixed-size and Dynamic Arrays

- Dynamic array has its **capacity**, which shows the maximum number of elements, it can contain.
- Also, such an array has the **logical size**, which indicates, how much elements it actually contains.
- **Example:**
  - Dynamic array with capacity 10, logical size 5.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | ? | ? | ? | ? | ? |

# Sample Class – DynamicArray

```
class DynamicArray {
private:
    int size;
    int capacity;
    int *storage;
public:
    DynamicArray() {
        capacity = 10;
        size = 0;
        storage = new int[capacity];
    }

    DynamicArray(int capacity) {
        this->capacity = capacity;
        size = 0;
        storage = new int[capacity];
    }

    ~DynamicArray() {
        delete[] storage;
    }
    void setCapacity(int newCapacity);

    void ensureCapacity(int minCapacity);

    void rangeCheck(int index);

    void set(int index, int value);

    int get(int index);

    void removeAt(int index);

    void insertAt(int index, int value);
};
```

# Sample Class – DynamicArray

```
void DynamicArray::setCapacity(int newCapacity)
{
    int *newStorage = new int[newCapacity];
    memcpy(newStorage, storage, sizeof(int) * size);
    capacity = newCapacity;
    delete[] storage;
    storage = newStorage;
}

void DynamicArray::ensureCapacity(int minCapacity)
{
    if (minCapacity > capacity) {
        int newCapacity = (capacity * 3) / 2 + 1;
        if (newCapacity < minCapacity)
            newCapacity = minCapacity;
        setCapacity(newCapacity);
    }
}
```

# Sample Class – DynamicArray

```
void DynamicArray::rangeCheck(int index)
{
    if (index < 0 || index >= size)
        throw "Index out of bounds!";
}

void DynamicArray::set(int index, int value)
{
    rangeCheck(index);
    storage[index] = value;
}

int DynamicArray::get(int index)
{
    rangeCheck(index);
    return storage[index];
}
```

```
void DynamicArray::removeAt(int index) {
    rangeCheck(index);
    int moveCount = size - index - 1;
    if (moveCount > 0)
        memmove(storage + index, storage +
(index + 1), sizeof(int) * moveCount);
    size--;
    pack();
}

void DynamicArray::insertAt(int index, int value) {
    if (index < 0 || index > size)
        throw "Index out of bounds!";
    ensureCapacity(size + 1);
    int moveCount = size - index;
    if (moveCount != 0)
        memmove(storage + index + 1, storage +
index, sizeof(int) * moveCount);
    storage[index] = value;
    size++;
}
```

# Sample Class – DynamicArray

- DynamicArray myArray(15);

myArray.storage

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |

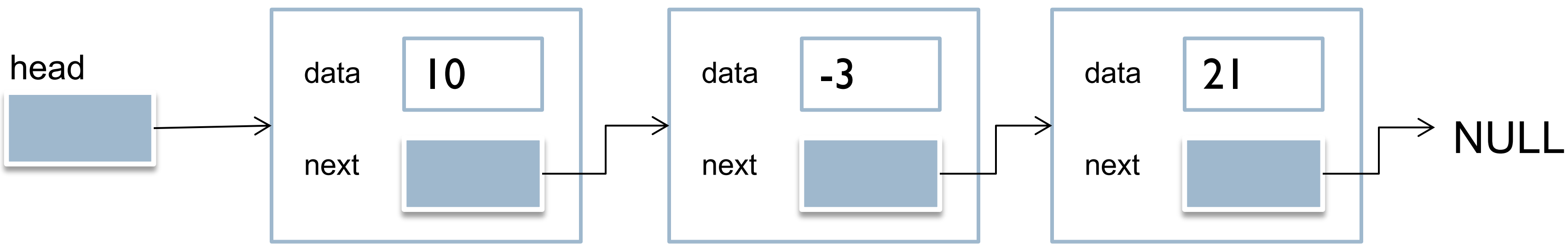
- myArray.set(5, 8);

myArray.storage

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ? | ? | ? | ? | ? | 8 | ? | ? | ? | ? | ? | ? | ? | ? | ? |

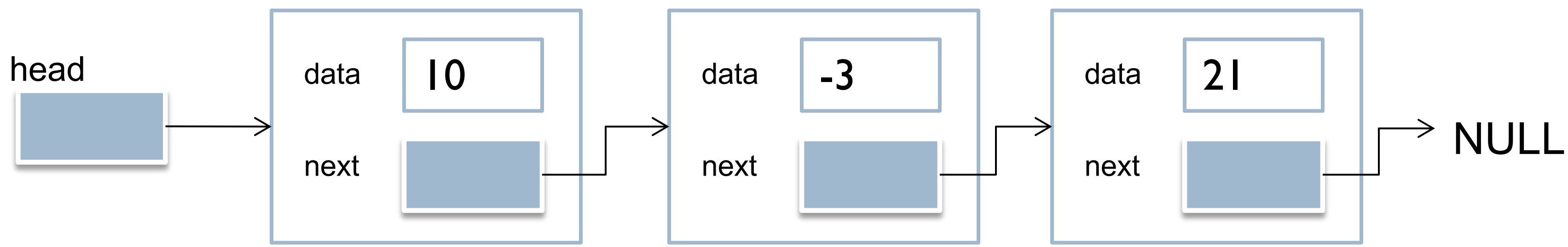
# Singly Linked List

- A singly linked list is another way of storing a list of values / objects
- Idea:
  - Each object stores the address to the object after it

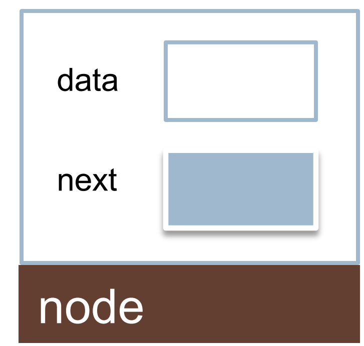



- Unlike array, objects are not stored contiguously in the memory
- The last object stores the address of the next object as NULL, since no more object.

# Singly Linked List




- A singly linked list is a series of connected "nodes"
- Each node contains
  - Data
  - A pointer to the next node in the list, generally named as "next"
- head: It is a pointer to the first node
- The pointer in the last node points to NULL



# Singly Linked List in C++

- Two user-defined types,
  i.e. two classes are needed
  - Node type / class
  - SinglyList type / class

- Node class

Node.h

```
class Node
{
   public:                // Making all the data members
     // data              // as public is very exceptional.
     double data;            // Since this facilitates rapid
     // pointer to next node   // access of data members
     Node* next;
};
```

# Singly Linked List in C++

- SinglyList class

SinglyList.h

```
class SinglyList
{
   private:
     Node* head; // a pointer to the first node in the list
   public:
      SinglyList();    // constructor
     ~SinglyList();   // destructor
     // isEmpty determines whether the list is empty or not
     bool isEmpty();
     // insertNode inserts a new node at position "index"
     Node* insertNode(int index, double x);
     // findNode finds the position of the node with a given value
     int findNode(double x);
     // deleteNode deletes a node with a given value
     int deleteNode(double x);
     // displayList prints all the nodes in the list
     void displayList();
};
```

# Creating and Destroying the List

- SinglyList()
  - The list is empty initially, so head pointer is set to NULL

SinglyList.cpp

```
SinglyList::SinglyList() {
    head = NULL;
}
```

- ~SinglyList()
  - Use the destructor to release all the memory used by the list
  - Walk through the list and delete each node one by one

SinglyList.cpp

```
SinglyList::~SinglyList() {
    Node* currNode = head;
    Node* nextNode = NULL;
    while(currNode != NULL) {
      nextNode = currNode->next;
      delete currNode;
      currNode = nextNode;
    }
}
```

# Operations

- Three common operations we can do on a singly-linked list
  - Traversal
  - Adding a node
  - Removing a node

# Traversal

- Traversal is the very basic operation, which presents as a part in almost every operation on a singly-linked list.
- For instance, algorithm may traverse a singly-linked list to find a value, find a position for insertion, etc.
- For a singly-linked list, only forward direction traversal is possible.

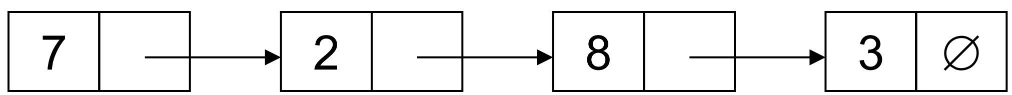
# Traversal Algorithm

- Beginning from the head,

1. check, if the end of a list hasn't been reached yet;
2. do some actions with the current node, which is specific for particular algorithm;
3. current node becomes previous and next node becomes current. Go to the step 1.

# Example

- Summing up values in a singly-linked list



# Finding a Node in the List

- int findNode(double x)
  - Search for a node with the value x in the list
  - If such a node is found, return its position index. Otherwise, return 0

```
int SinglyList::findNode(double x) {
   Node* currNode = head;
   int currIndex = 1;
   while(currNode && currNode->data != x)
   {
     currNode = currNode->next;
     currIndex++;
   }
   if(currNode)
     return currIndex;
   return 0;
}
```

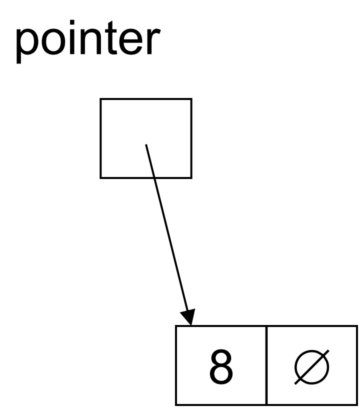
SinglyList.cpp

# Addition (Insertion)

- Insertion into a singly-linked list has four cases.
  - Empty list
  - Before the head
    (to the very beginning of the list)
  - After the tail
    (to the very end of the list).
  - In the middle of the list and so,
    has a predecessor and successor in the list.

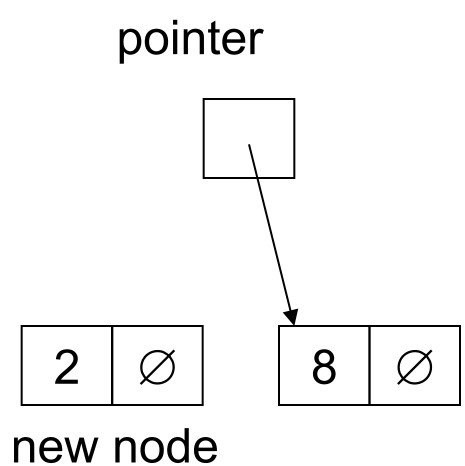
# Empty List

- When list is empty, the insertion is quite simple. Algorithm sets the pointer to point to the new node.



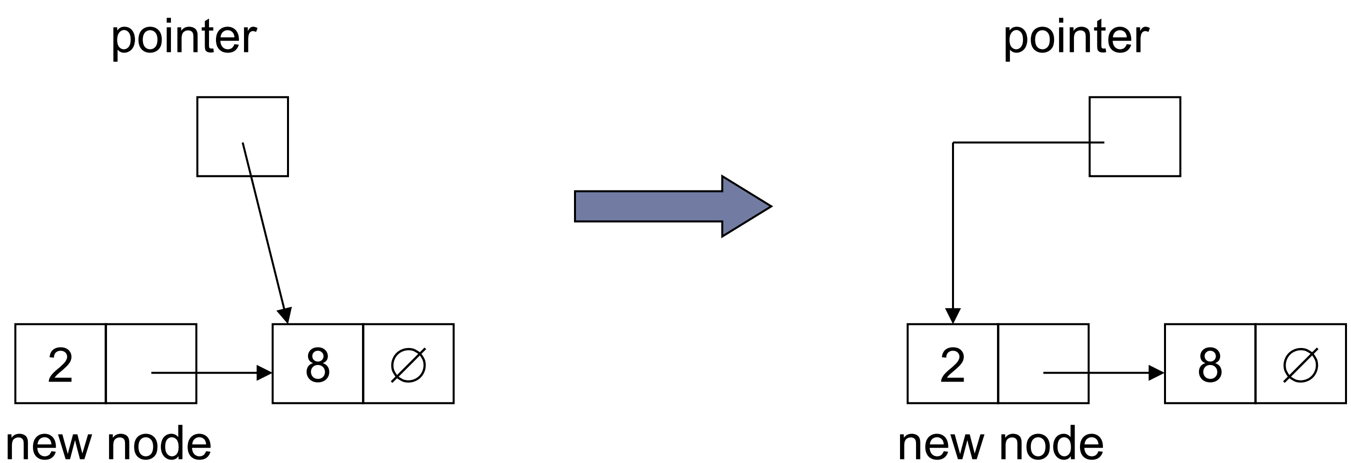
# Add First

- In this case, new node is inserted right before the current head node.



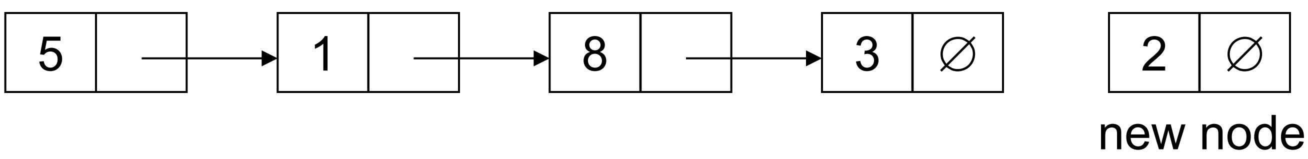
# Add First

- It can be done in two steps:
  1. Update the next link of a new node, to point to the current head node.
  2. Update the pointer to point to the new node.



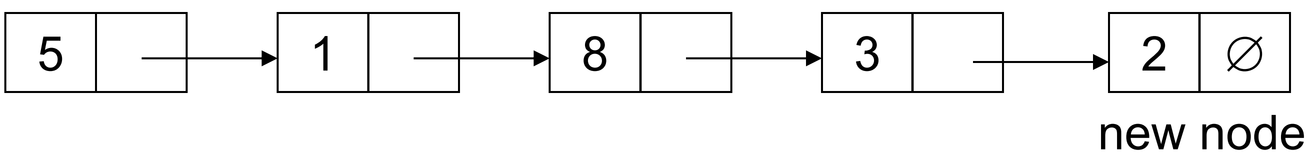
# Add Last

- In this case, new node is inserted right after the current tail node.



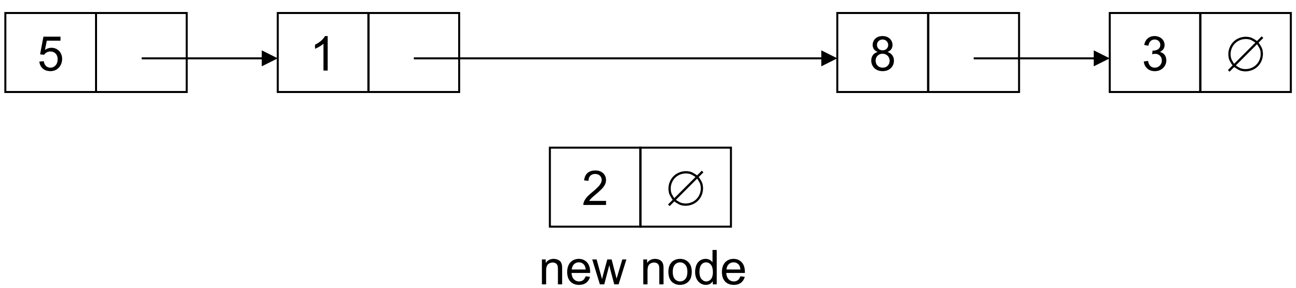
# Add Last

- It can be done in one steps:
  1. Update the next link of the current tail node, to point to the new node.



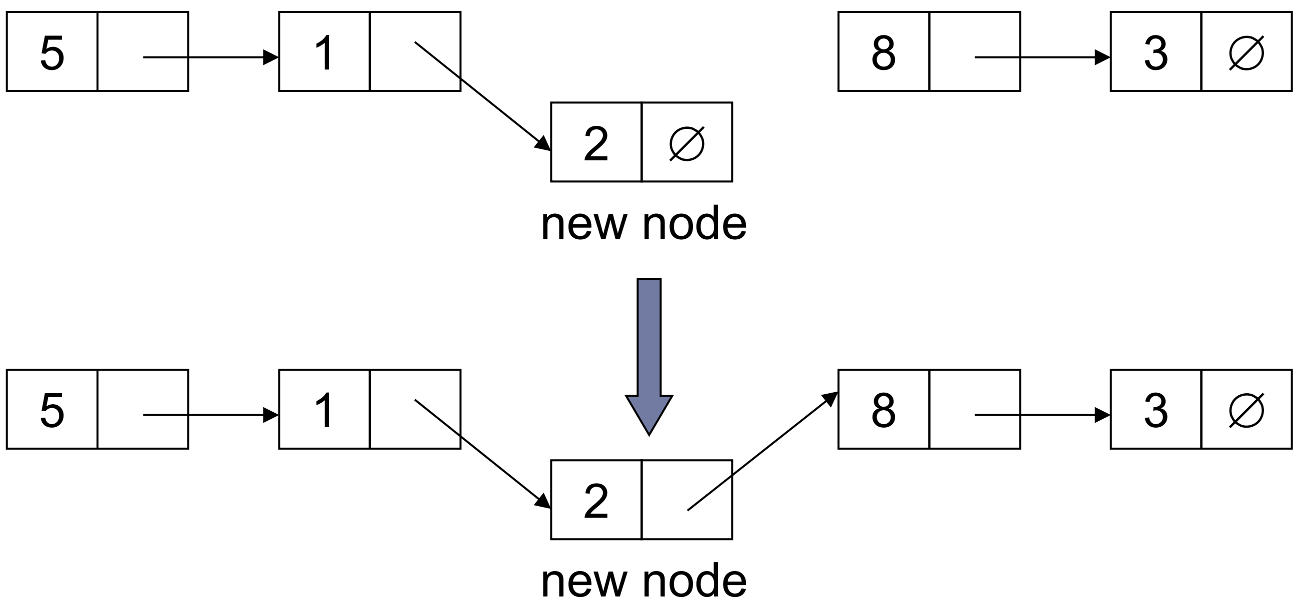
# Insert Between Two Nodes

- In general case, new node is **always inserted between** two nodes, which are already in the list.



# Insert Between Two Nodes

- Such an insert can be done in two steps:
  1. Update link of the "previous" node, to point to the new node.
  2. Update link of the new node, to point to the "next" node.

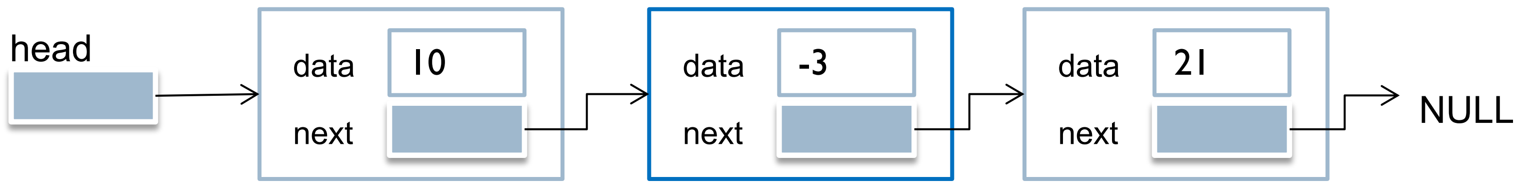


# Inserting a New Node to the List

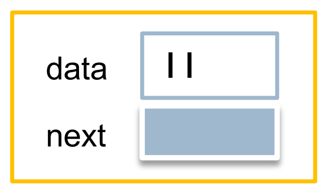
- Node* insertNode(int index, double x)
  - Insert a node with data x at position "index"
    - When index = 0, insert the node as the first element
    - When index = 1, insert the node after the first element
  - If the insertion is successful, return the pointer of the inserted node. Otherwise, return NULL
    - If index is less than 0 or greater than length of the list, this means invalid index and thus the insertion will fail
- Steps:
  1. Locate the node at position "index"
  2. Dynamically allocate memory for the new node and put value into it
  3. Point the new node to its successor
  4. Point the new node's predecessor to the new node

# Example: Inserting a New Node to the List

- Steps:
  1. Locate the node at position index, say index = 2

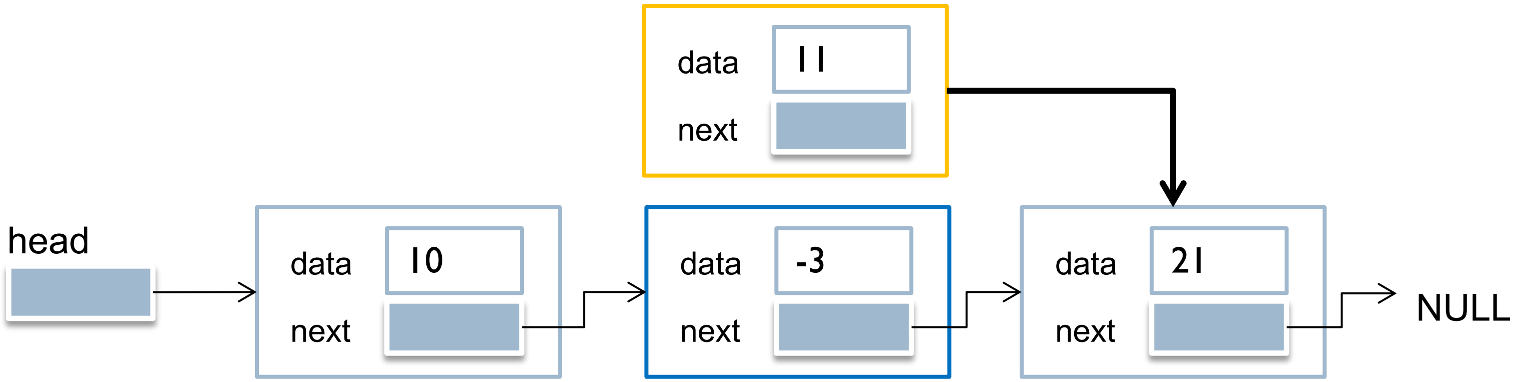



  2. Dynamically allocate memory for the new node and put value into it, say 11

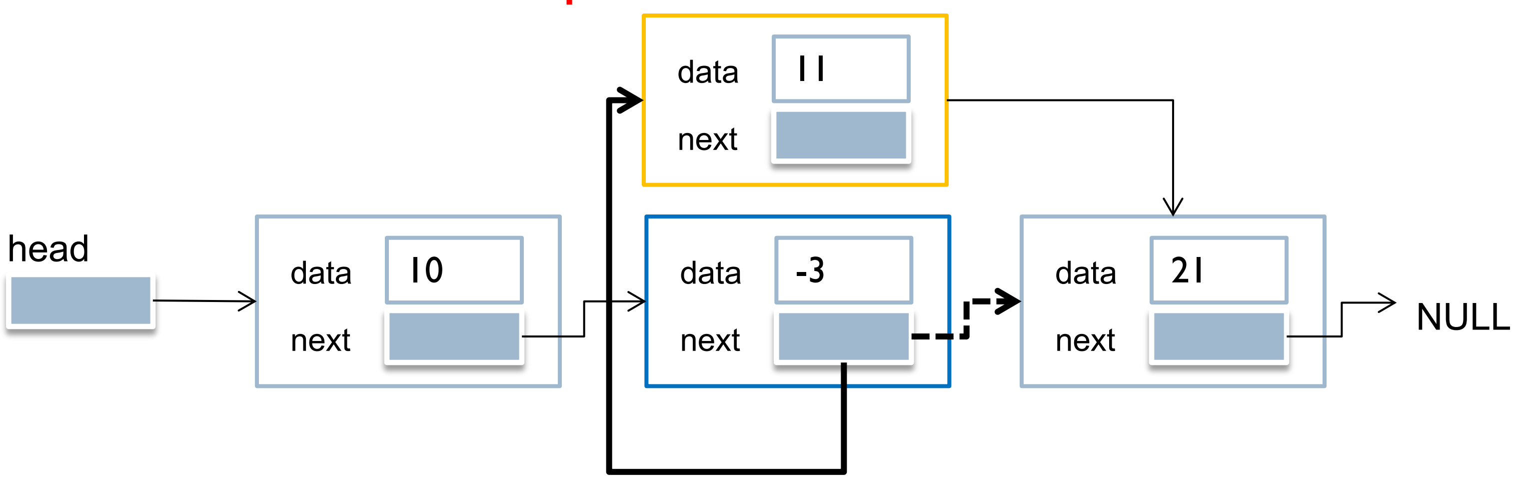


# Inserting a New Node to the List

- Point the new node to its successor




- Point the new node's predecessor to the new node



# Inserting a New Node to the List

- Possible cases of insertNode
  1. Insert a new node into an empty list
  2. Insert a new node to the very beginning of the list
  3. Insert a new node to the end of the list
  4. Insert a new node in middle
- But in fact, we only have to handle two cases
  - Insert as the first node
    (Case 1 & 2)
  - Insert a new node in
    middle or at the end of the list
    (Case 3 & 4)

# Implementation of insertNode Function

```
Node* SinglyList::insertNode(int index, double x) {
  if(index < 0)
     return NULL;
  int currIndex = 1;
  Node* currNode = head;
  while(currNode && index > currIndex) {
    currNode = currNode->next;
    currIndex++;
  }
  if(index > 0 && currNode == NULL)
    return NULL;
  Node* newNode = new Node;
  newNode->data = x;
  if(index == 0) {
    newNode->next = head;
    head = newNode;
  }
  else {
    newNode->next = currNode->next;
    currNode->next = newNode;
  }
  return newNode;
}
```

Locate the node at position index

Create a new node

Insert as first element

Insert after currNode

SinglyList.cpp

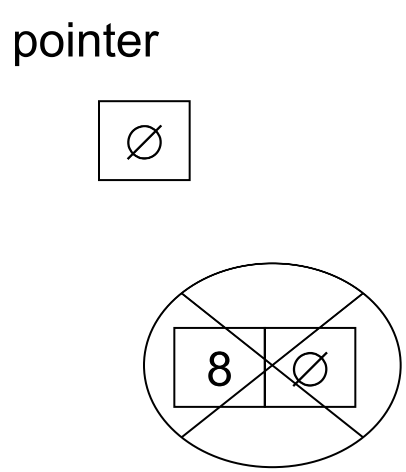
# Removal (Deletion)

- There are four cases, which can occur while removing the node.

- These cases are similar to the cases in add operation. We have the same four situations, but the order of algorithm actions is opposite.

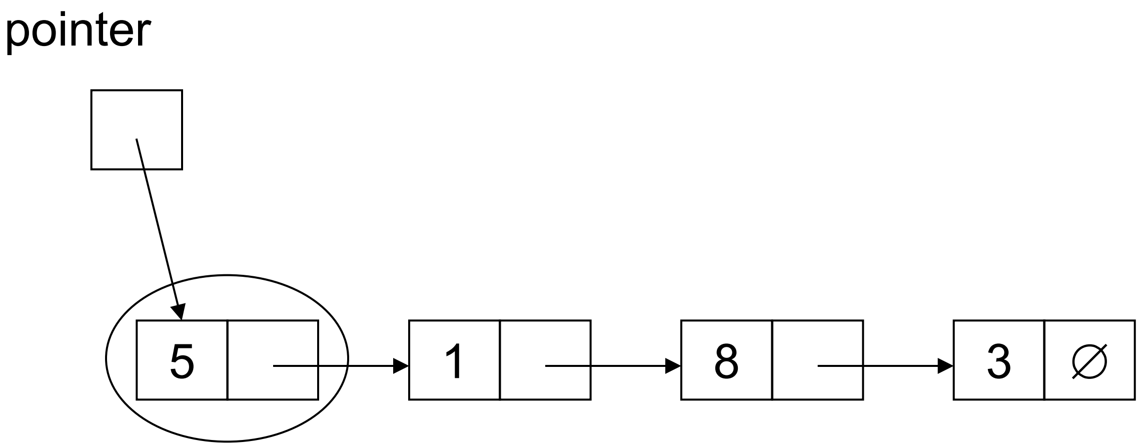
# One Node

- When list has only one node, the removal is quite simple. Algorithm disposes the node, and sets the pointer to *NULL*.



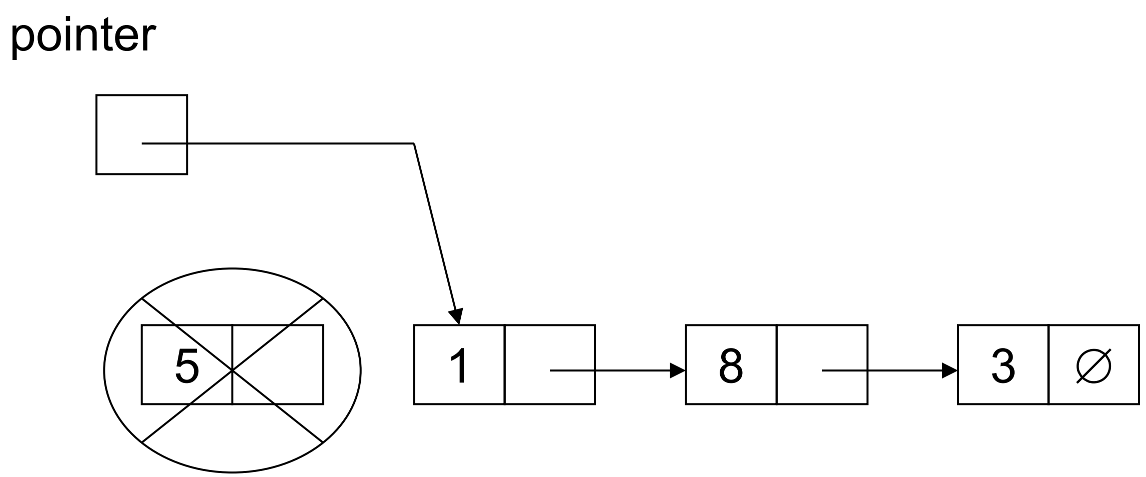
# Remove First

- In this case, first node (current head node) is removed from the list.



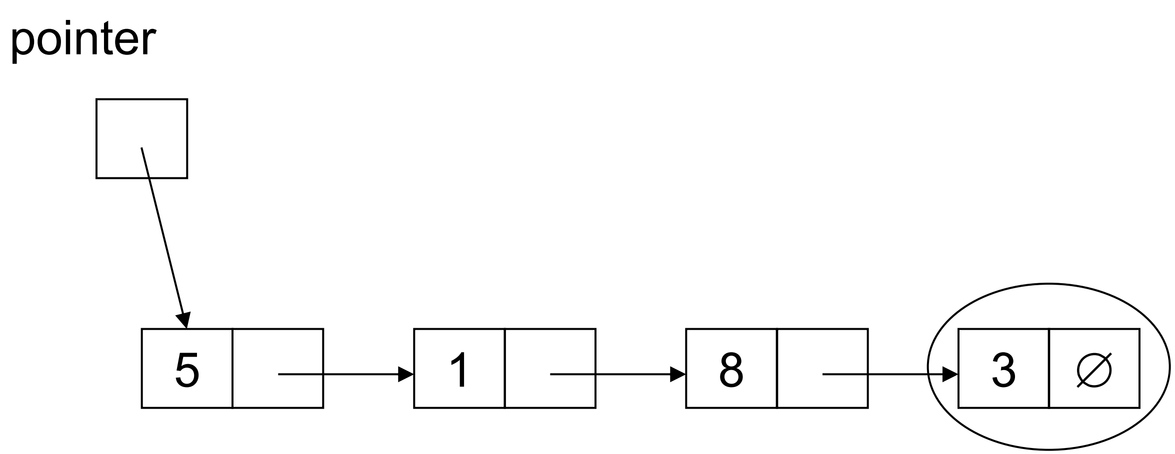
# Remove First

- It can be done in two steps:
  1. Update the pointer to point to the node, next to the head.
  2. Dispose removed node.



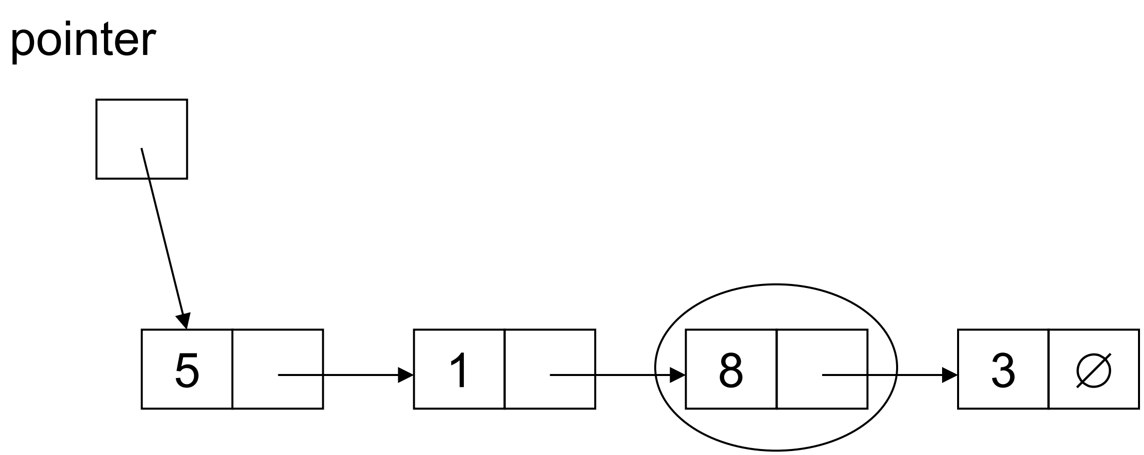
# Remove Last

- In this case, last node (current tail node) is removed from the list. This operation is a bit more tricky than removing the first node, because algorithm should find a node, which is previous to the tail first.



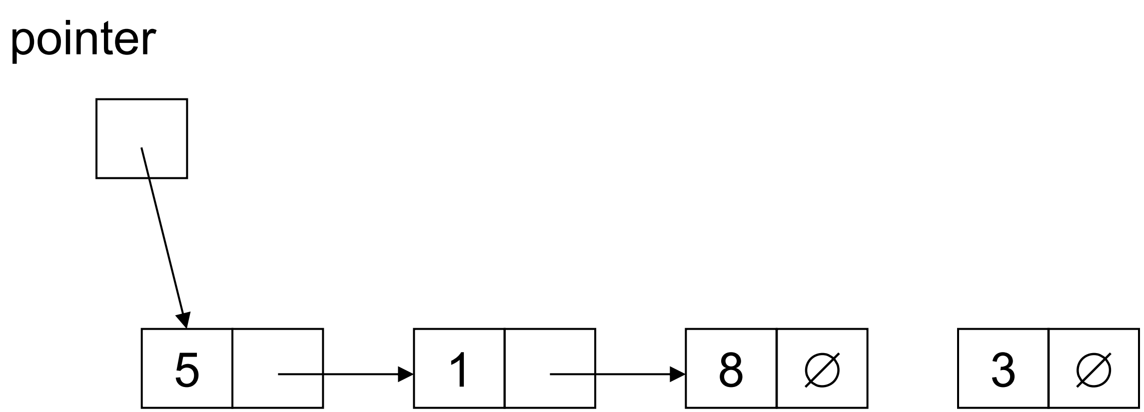
# Remove Last

- It can be done in three steps:
  1. Find the node before the tail.
     In order to find it, list should be traversed first, beginning from the head.



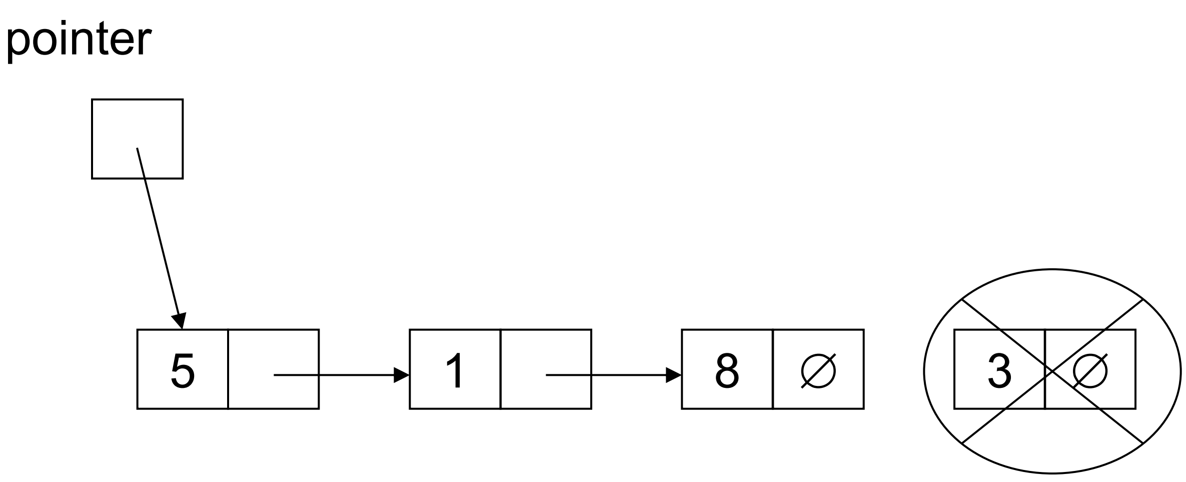
# Remove Last

- **It can be done in three steps:**
  2. Set next link of the new tail to NULL.



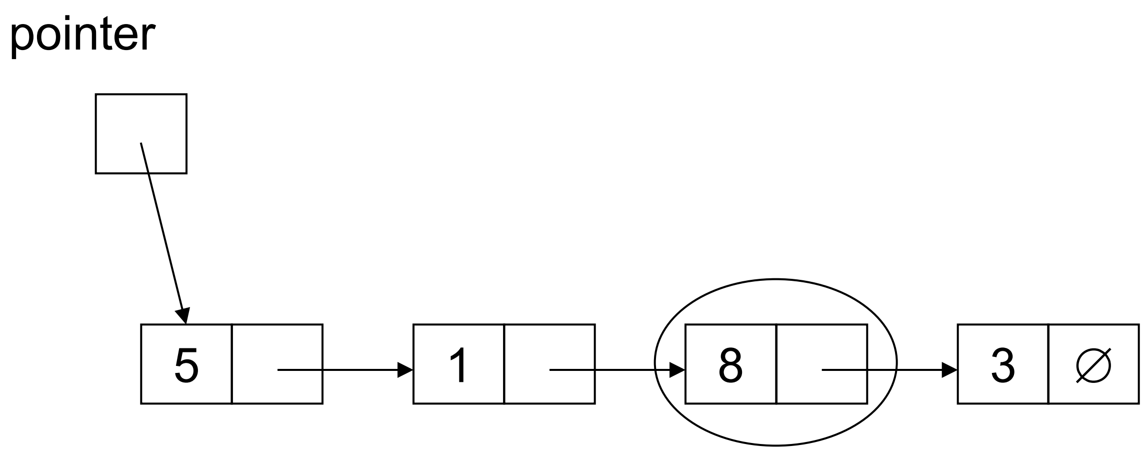
# Remove Last

- It can be done in three steps:
  3. Dispose removed node.



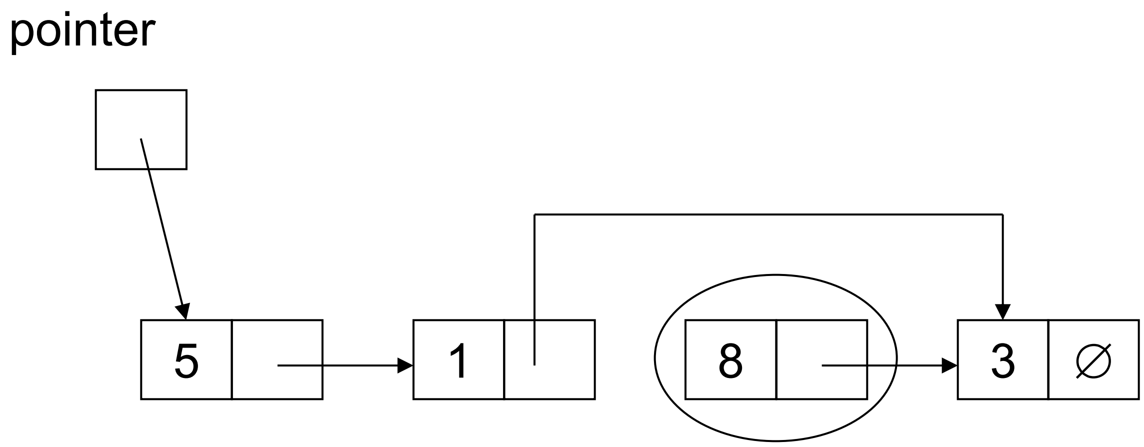
# Remove Between Two Nodes

- In general case, node to be removed is **always located between** two list nodes.



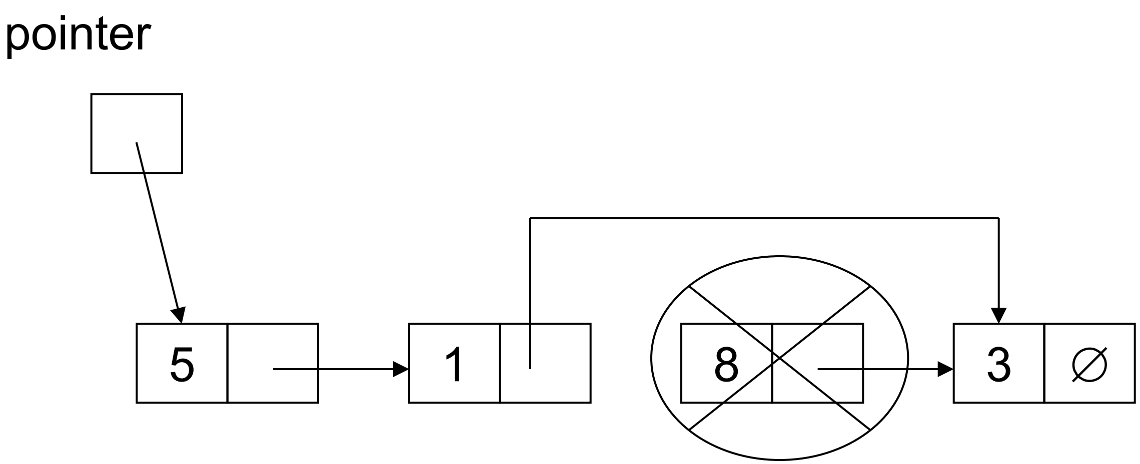
# Remove Between Two Nodes

- Such a removal can be done in two steps:
  1. Update next link of the previous node, to point to the next node, relative to the removed node.



# Remove Between Two Nodes

- Such a removal can be done in two steps:
  2. Dispose removed node.



# Deleting a Node

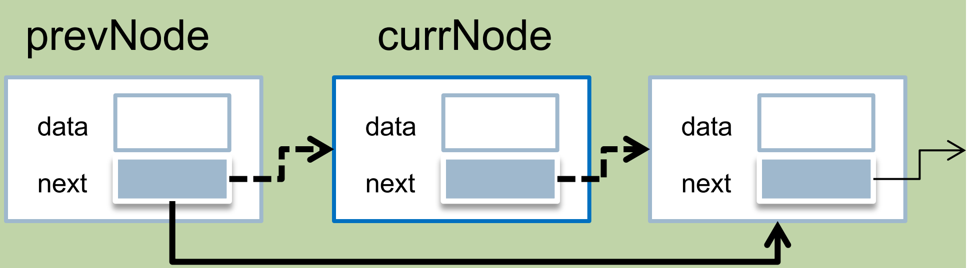
- int deleteNode(double x)
  - Delete a node with the value x from the list
  - If such a node is found, return its position index. Otherwise, return 0
- Steps:
  1. Find the target node (similar to findNode)
  2. Release the memory occupied by the found node
  3. Set the pointer of the predecessor of the found node to the successor of the found node
- Similar to insertNode, two cases
  - Delete first node
  - Delete the node in the middle or at the end of the list

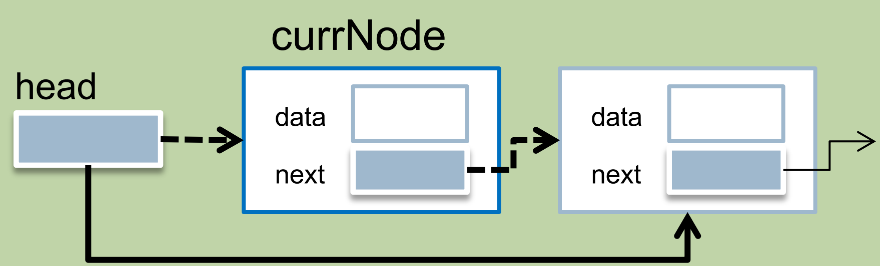
# Implementation of deleteNode Function

SinglyList.cpp

```
int SinglyList::deleteNode(double x) {
    Node* prevNode = NULL;
    Node* currNode = head;
    int currIndex = 1;
    while(currNode &&
          currNode->data != x)
    {
      prevNode = currNode;
      currNode = currNode->next;
      currIndex++;
    }
    if(currNode) {
      if(prevNode) {
        prevNode->next =
        currNode->next;
        delete currNode;
      }
      else {
        head = currNode->next;
        delete currNode;
      }
      return currIndex;
    }
    return 0;
}
```

Find the node with value x



# Printing all the Elements

- void displayList()
  - Print the data of all the elements
  - Print the number of the nodes in the list

SinglyList.cpp

```
void SinglyList::displayList(){
   int num = 0;
   Node* currNode = head;
   while(currNode != NULL) {
     cout << currNode->data << endl;
     currNode = currNode->next;
     num++;
   }
   cout << "Number of nodes in the list: " << num << endl;
}
```

# Using SinglyList Class

```
int main() {
    SinglyList list;
    list.insertNode(0, 6.0);
    list.insertNode(1, 4.0);
    list.insertNode(-1, 3.0);
    list.insertNode(0, 2.0);
    list.insertNode(8, 5.0);
    list.displayList();
    if(list.findNode(4.0) > 0)
        cout << "4.0 is found" << endl;
    else
        cout << "4.0 is not found" << endl;
    if(list.findNode(5.6) > 0)
        cout << "5.6 is found" << endl;
    else
        cout << "5.6 is not found" << endl;
    list.deleteNode(6.0);
    list.displayList();
    return 0;
}
```

main.cpp

Output:

```
2
6
4
Number of nodes in the list: 3
4.0 is found
5.6 is not found
2
4
Number of nodes in the list: 2
```

# Singly Linked List

- Advantages:
  - No fixed size, it grows one object at a time. So, it only uses as much space as in needed for the objects in the list
  - The insertion and deletion of a value / object is relatively easier than array
- Disadvantages:
  - Coding is more complex than array
  - With space overhead. Since each object needs to store the address of the next object
  - Not easy to access the value of object at certain position. Since we need to go through all the objects from the beginning to that object

# One more problem

- One more problem of singly linked lists is
  - The nodes in such lists contain only pointers to the successors
  - Therefore, there is no immediate access to the predecessors.

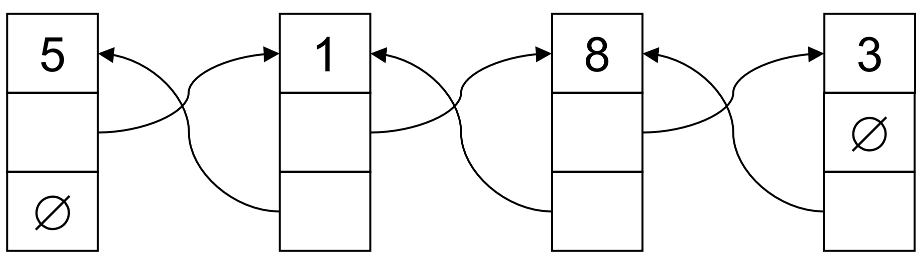
# One more problem

- To do the remove last, we have to scan the entire list to stop right in front of tail.

- For long lists, this may be an impediment to swift list processing.

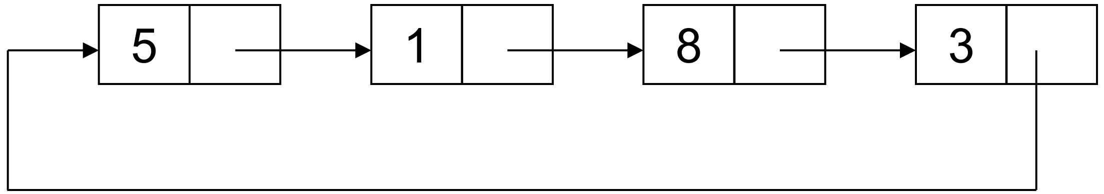
# Doubly Linked List

- To avoid this problem, the linked list is redefined so that each node in the list has two pointers,
  - One to the successor and
  - One to the predecessor
- A list of this type is called a doubly linked list



# CIRCULAR LISTS

- In some situations, a circular list is needed in which nodes form a ring
  - The list is finite and each node has a successor.



# Skip Lists

- Linked lists have one serious drawback:
  - They require sequential scanning to locate a searched-for element.
  - The search starts from the beginning of the list and stops when either a searched-for element is found or the end of the list is reached without finding this element.

# Skip Lists

- Linked lists have one serious drawback:
  - Ordering elements on the list can speed up searching, but a sequential search is still required.
  - Therefore, we may think about lists that allow for skipping certain nodes to avoid sequential processing.

# Skip Lists

- A skip list is an interesting variant of the ordered linked list that makes such a non sequential search possible.

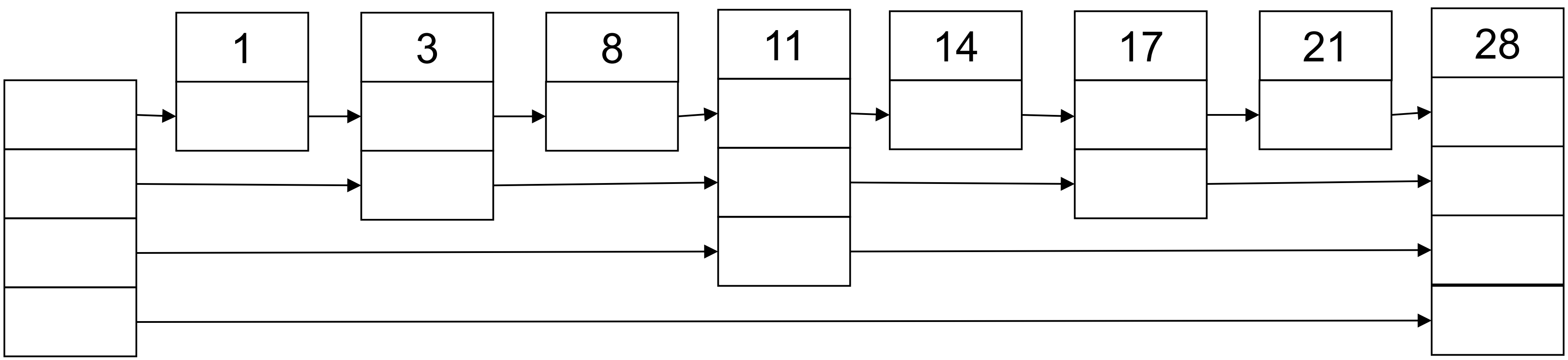
# Skip Lists

- In a skip list
  - Every second node points to the node two positions ahead
  - Every forth node points to the node four positions ahead

# Skip Lists

- In a skip list
  - The list is accomplished by having different numbers of pointers in nodes on the list
    - Half of the nodes have just one pointer
    - One-fourth of the nodes have two pointers
    - One-eighth of the nodes have three pointers
  - The number of pointers indicates the level of each node.

# Skip Lists



# Conclusion

# Conclusion

- Linked lists have been introduced to overcome limitations of arrays by allowing dynamic allocation of necessary amounts of memory.

- Also, linked lists allow easy insertion and deletion of information, because such operations have a local impact on the list.

# Conclusion

- To insert a new element at the beginning of an array, all elements in the array have to be shifted to make room for the new item;

- Hence, insertion has a global impact on the array.

- Deletion is the same.

# Conclusion

- Arrays have some advantages over linked lists, namely that they allow random accessing.
  - To access the tenth node in a linked list, all nine preceding nodes have to be passed.
  - In the array, we can go to the tenth cell immediately.

# Conclusion

- Another advantage in the use of arrays is space.
    - To hold items in arrays, the cells have to be of the size of the items.
    - In linked lists, we store one item per node, and the node also includes at least one pointer;
    - In doubly linked lists, the node contains two pointers.

# CHAPTER 4 END