NWEN 241
Systems Programming

Sue Chard

suechard@ecs.vuw.ac.nz

Content

- Review Generics C++
- Data structures in systems programming C and C++

Generics

C++

- Writing code in a way that is independent of any particular type
- Generic functions and Generic classes have type parameters
- A type parameter may be a basic /fundamental / primitive / built-in type such as int or double or a user defined type such as class or structure

- Example of a generic function

```
template <typename T>
  T mymax(T a, T b) {
  return a > b ? a : b;
}
```

```
cout << mymax (3, 7) << endl;
cout<< mymax <int>(3,7)<< endl;
```

- A class template provides a specification for generating classes based on parameters
- Class templates are generally used to implement containers
- A class template is instantiated by passing a given set of types to it as template arguments
- Types can be both basic / fundamental /primitive / built-in datatypes and user defined datatypes (classes / structures)

Generics

C++

- Standard Template Library (STL) has **Containers, Algorithms, Iterators**
- One of the containers is **vector**
- Vectors are similar to dynamic arrays with the ability to resize automatically when an element is inserted or deleted
- Memory management is handled automatically by the container
- There are built in safety features such as bounds checking
- There is an overhead, vectors use more memory and have higher processing overheads than arrays
- A note about auto addition in the C++ 11 standard

These two statements are equivalent as both store an int variable on the stack

```
auto int i = 0;        int i = 0;
```

however from the C++ 11 standard, you can code variable declarations as shown below, this allows more flexibility in code written , it is known as a placeholder type specifier

```
auto j  = i;
```

The datatype for j is deduced from the assigned variables' datatype this syntax is

```
auto variablename = another variablename;
```

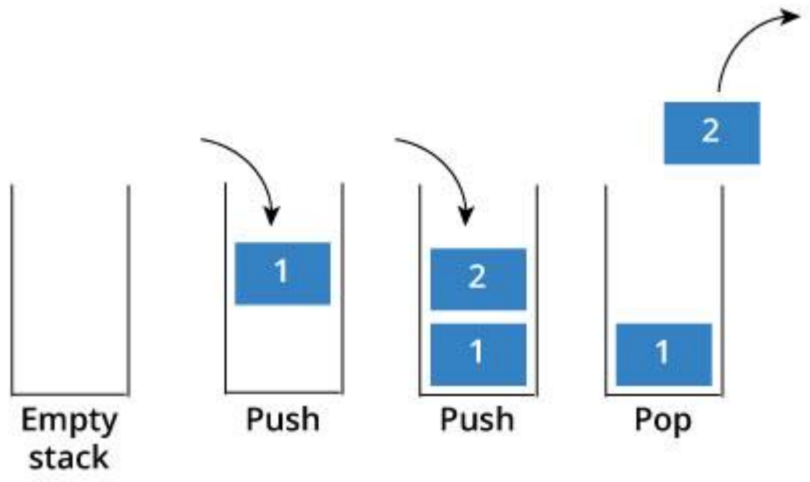
```cpp
// C++ program to illustrate the capacity function in vector
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector<int> g1;

   for (int i = 1; i <= 5; i++)
      g1.push_back(i);

   cout << "Size : " << g1.size();
   cout << "\nCapacity : " << g1.capacity();
   cout << "\nMax_Size : " << g1.max_size();

   // resizes the vector size to 4
   g1.resize(4);

   // prints the vector size after resize()
   cout << "\nSize : " << g1.size();

   // checks if the vector is empty or not
   if (g1.empty() == false)
      cout << "\nVector is not empty";
   else
      cout << "\nVector is empty";

   // Shrinks the vector
   g1.shrink_to_fit();
   cout << "\nVector elements are: ";
   for (auto it = g1.begin(); it != g1.end();
it++)
      cout << *it << " ";

   return 0;
}
```

Output:
Size : 5
Capacity : 8
Max_Size : 4611686018427387903
Size : 4
Vector is not empty
Vector elements are: 1 2 3 4

Data Structures - Stack

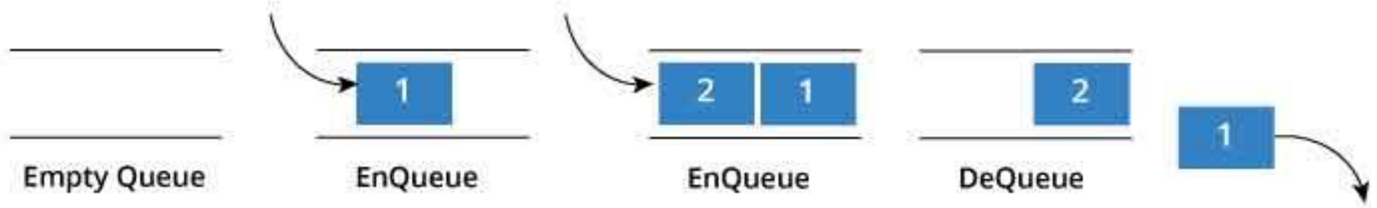
C and C++

- Stack, Last in First out LIFO
  - Push: Add element to the top of the stacjk
  - Pop: Remove element for the top of the stack
  - IsEmpty: check if stack is empty
  - isFull: Check if stack is full
  - Peek: Get value form the top element without removing it
- A pointer is used to keep track of the top of the stack
  - needs an initial value to indicate stack is empty usually -1
- Most common implantation is using an array
- Used for: return addresses, expression evaluation, local variable storage, parameters



Queue

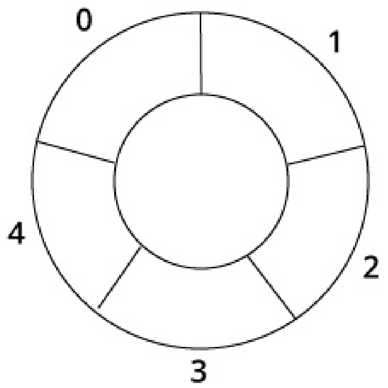
- Queue, First in First out FIFO
  - Enqueue: add element to the end of the queue
  - Dequeue: remove element from the front of the queue
  - IsEmpty: check if the queue is empty
  - IsFull: check if the queue is full
  - Peek: get the value form the front of the queue without removing it
- Uses two pointers, one to the front of the queue and one to the end of the queue
  - Must set initial values of both usually -1
  - May be implemented with an array
- Used for anything where you need to maintain order, examples are input keyboard buffer, printer queue

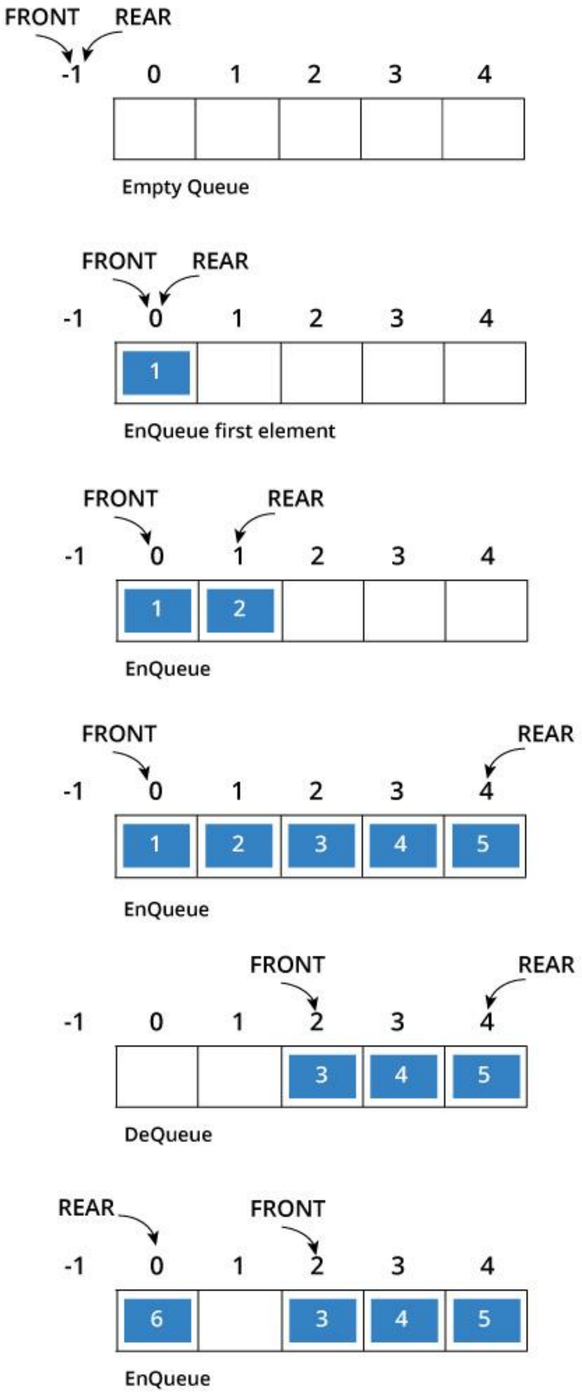


Circular Queue

C and C++

- Circular Queue First in First out, but reuses space
- Works using the process of a circular increment i.e. when the end of the queue is reached, start from the beginning of queue
- Uses modulus division with the queue size
- Two pointers are used to keep track of the front and the rear
- Implemented using an array





Priority Queue

C and C++

- Priority Queue First in First out, but includes priority of each item
- Managing the queue Is based on the order of priority
  - An element with high priority is dequeued before an element with low priority
  - Two elements with the same priority are dequeued according to their order in the queue
- May be implemented using arrays, linked lists or heaps
- Used to handle execution processes

Linked List

C and C++

- Linked list, elements are linked using pointers. It is a list consisting of nodes
- Each node contains data and the address of the next node
- A reference is kept to the first element in the list known as the head of the list
  - The head   points to NULL if the list is empty
- The last node points to null

- Each node can be stored in a struct containing, data and a pointer to the next node

```
struct node {
        int data;
        struct node *next;
};
```

Linked List

C and C++

- Linked list operations include
  - **Adding a node**
    - Allocate memory for a node
    - Set the previous node "next pointer" to the address of the node just created
  - **Traverse / iterate through the list**
    - Start with the address of node stored in head, follow the pointers until the pointer in next is Null
  - **Add a node to the end of the list**
    - Iterate through the list to the last item, create the new node, link the new item to the tail of the list
  - **Add a node to the beginning of the list**
    - Create a new node
    - Link the new node to point to the head of the list
    - Set the head of the list to be the new item
  - **Add a node to the middle of the list**
    - Create a new node
    - Iterate through the list until the position you want to insert the new node
    - Link the previous item in the list to the new node
    - Link the new node to the next item in the list
  - **Remove the first item**
  - **Remove the last item**
  - **Remove a specific item**

- Disadvantages no random access have to traverse the list

Example code for linked list

C

```c
#include <stdio.h>

#include <stdlib.h>

struct node

{

 int data;

 struct node *next;

};

int main(){

 /* Initialize nodes */

 struct node *head;

 struct node *one = NULL;

 struct node *two = NULL;

 struct node *three = NULL;


 /* Allocate memory */

 one = malloc(sizeof(struct node));

 two = malloc(sizeof(struct node));

 three = malloc(sizeof(struct node));

/* Assign data values */
 one->data = 1;
 two->data = 2;
 three->data=3;

 /* Connect nodes */
 one->next = two;
 two->next = three;
 three->next = NULL;

 /* Save address of first node in head */
 head = one;

//add to beginning
  struct node *newNode;
  newNode = malloc(sizeof(struct node));
  newNode->data = 4;
  newNode->next = head;
  head = newNode;

 //add to end
 newNode = malloc(sizeof(struct node));
  newNode->data = 5;
  newNode->next = NULL;

  temp = head;
  while(temp->next != NULL){
   temp = temp->next;
  }
  temp->next = newNode;

  //iterator
  temp = head;
  printf("\n\nList elements are - \n");
  while(temp != NULL)
  {
   printf("%d --->",temp->data);
   temp = temp->next;
  }
 //free dynamic memory
  return 0;
 }
```

```
List elements are -
4 --->1 --->2 --->3 --->5 --->
```

Doubly Linked List

- Singly Linked List is the most common, each node has data and a pointer to the next node.
  - Can only move forward
- Doubly linked list has pointers to both, the next item and the previous item in the list
  - Can move forward or backward

```
struct node {
        int data;
        struct node *next;
        struct node *previous;
};
```

Other types of Linked List

- Ordered linked list, maintains items in sorted order, linked lists make it easy to add items in the middle of a list without copying the list contents

- Circular list, the last element is linked to the first element, this forms a circular loop.
  - Used for applications where each item in the list must be visited equally and the list can grow and shrink such as timesharing in an operating system.
  - It is usually implemented with singly linked list
  - Can be used to implement queues

Abstract Data Types (ADT)

C and C++

- Data structures such as Stacks, Queues, Lists
- Defines the operations that can be performed on the structure – the minimal expected interface for the implementation
- Does not define how the operations are performed or the way the data is stored
- Slides 6-11 have details of the expected interface (operations) for stacks queues and linked lists

Example Linked List - definitions

C

```c
#include <stdio.h>
#include <stdlib.h>

struct node
{
  int data;
  struct node *next;
  struct node *prev;
};

int main(){
```

C++

```cpp
#include <cstdio>
#include <cstdlib>

struct node
{
  int data;
  struct node *next;
  struct node *prev;
};

int main(){
```

- Different libraries to use in C and C++
- Struct is defined the same
- Singly linked list has a next pointer
- Doubly linked list has a previous pointer as well
- Just definitions no memory allocated

C	C++

C

```c
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;
```

C++

```cpp
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;
```

Head

?

one

?

two

?

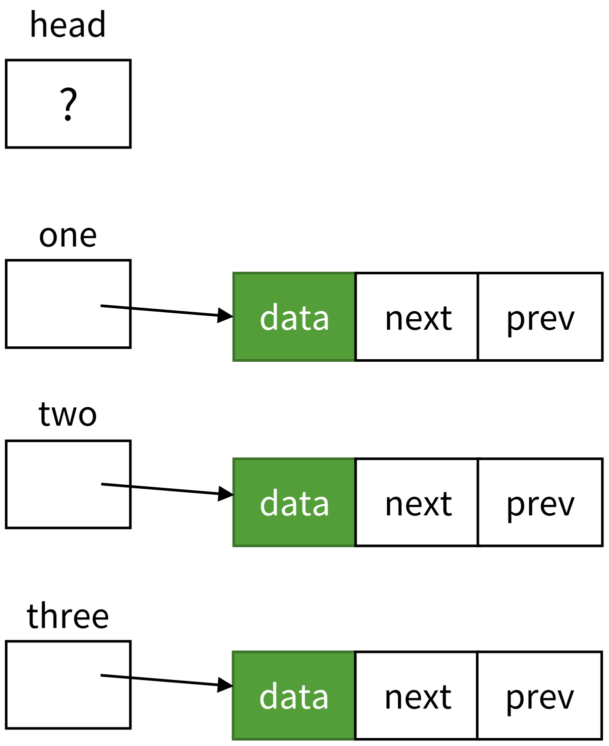
three

?

- Allocates memory for 4 pointers to struct node’s
- No values have been assigned to them

C and C++

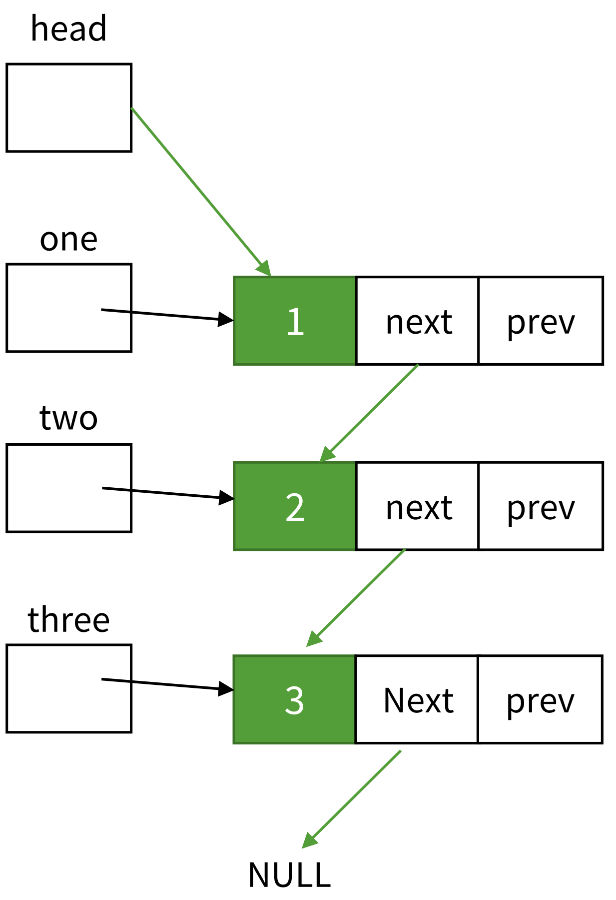
```
one = (struct node *)malloc(sizeof(struct node));
if(!one) {
  printf("one not allocated");
  return 0;
}

two = (struct node *)malloc(sizeof(struct node));
if(two == NULL) {
  printf("two not allocated");
  return 0;
}
three = (struct node *) malloc(sizeof(struct node));
if(three == NULL) {
  printf("three not allocated");
  return 0;
}
```



- Allocates memory for 4 struct node's and assigns the addresses to one, two and three
- In C++ must typecast the pointer returned by malloc , in C this isn't required so the code does not require (struct node *) in each assign statement

C and C++

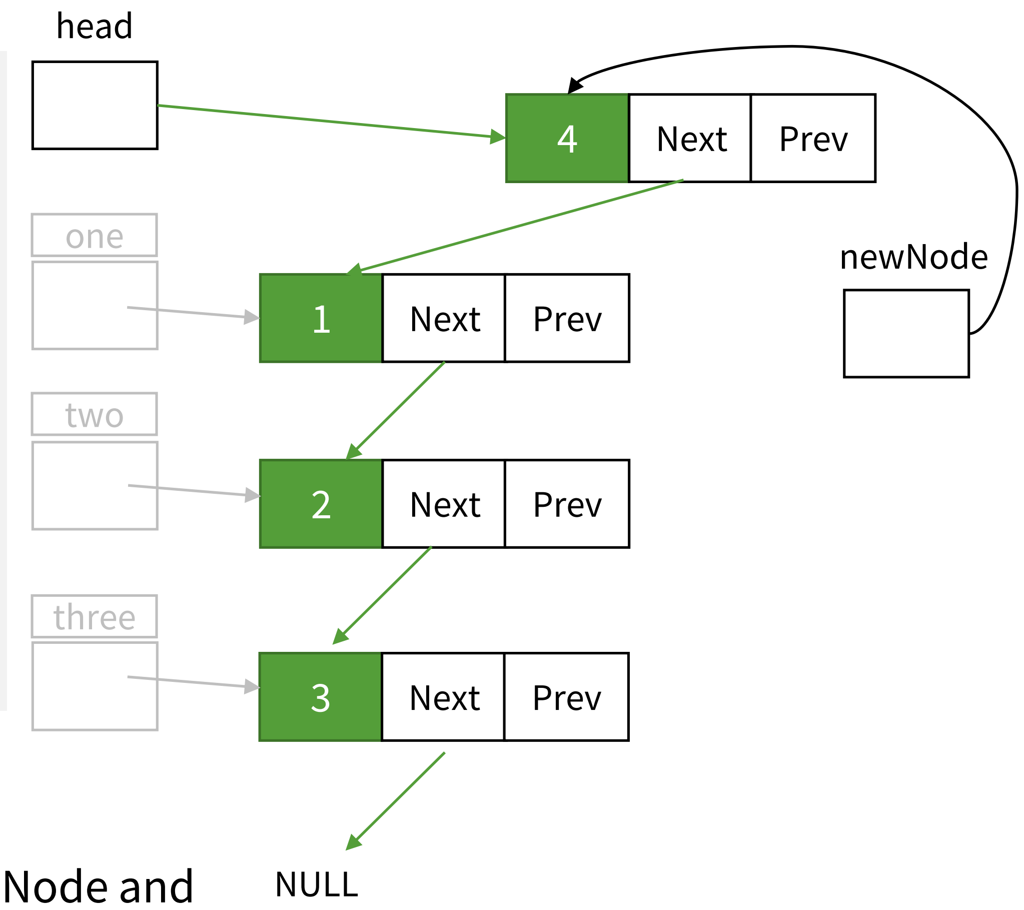
```
one->data = 1;
two->data = 2;
three->data=3;

/* Connect nodes */

one->next = two;
two->next = three;
three->next = NULL;

head = one;
```



- Assigns values to data in each of the nodes
- Connects the nodes, by setting the next pointers
- Sets the head pointer to the address of node one

C and C++

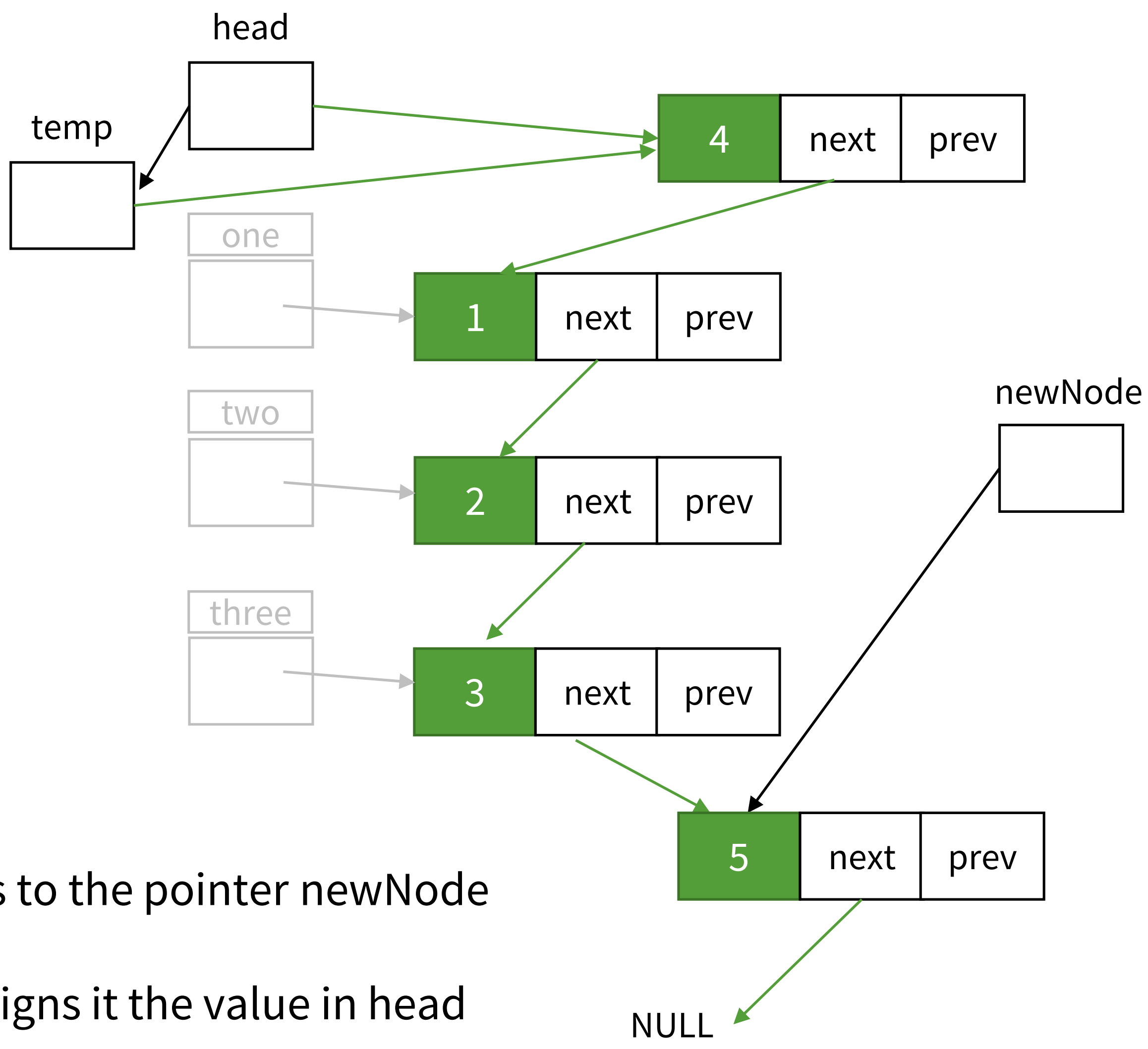
```c
  //add to beginning

struct node *newNode = (struct node *)malloc(sizeof(struct node));

if(newNode == NULL) {
  printf("newNode1 not allocated");
  return 0;
}
newNode->data = 4;
newNode->next = head;
head = newNode;
```



- Allocates memory for another pointer to a struct node called newNode and
- Allocates memory for the newNode
- Checks to ensure the memory allocation was successful
- Assigns values to the data and next pointer for newNode
- Sets the head pointer to the address of node newNode

C and C++

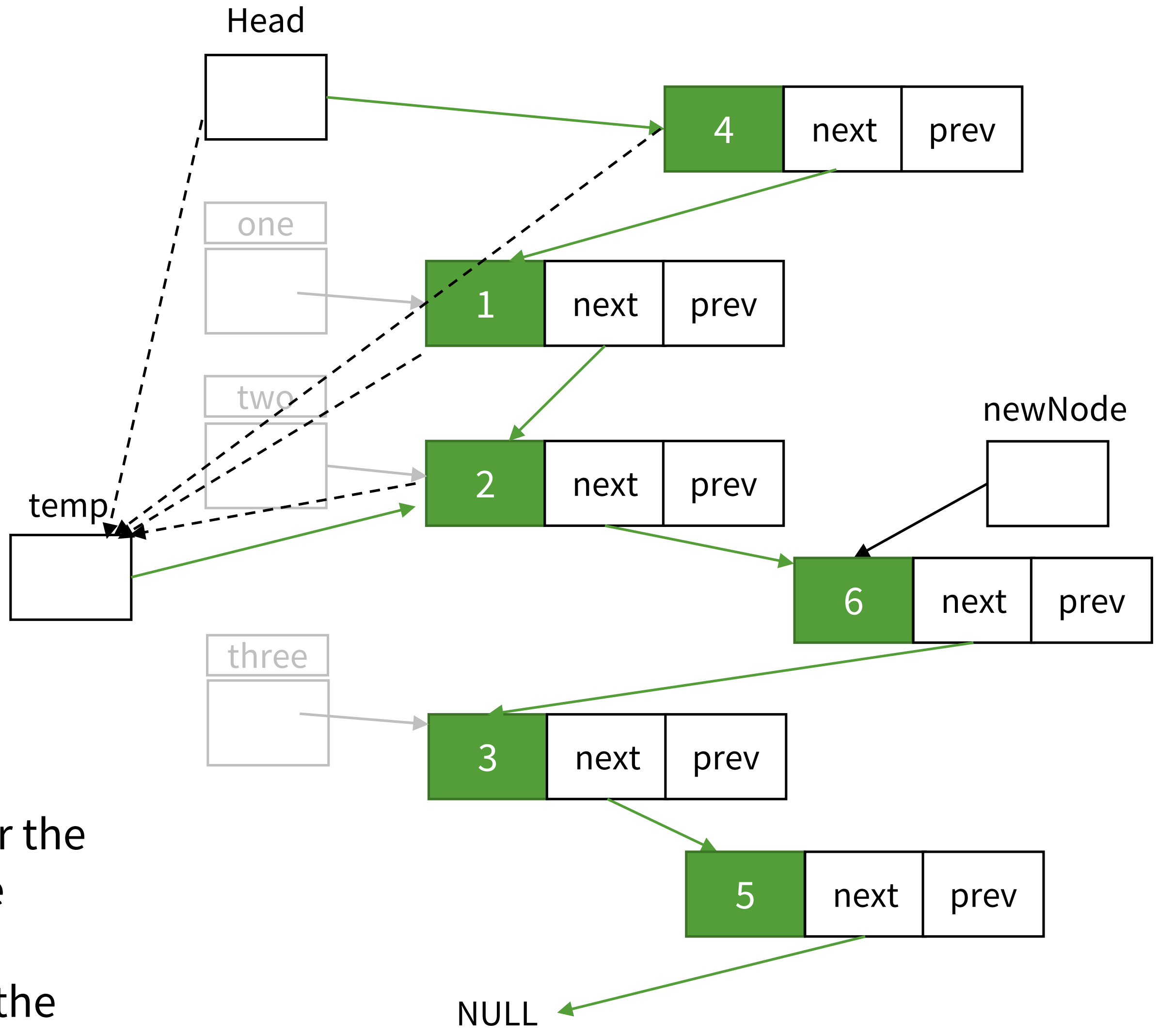
```
//add to end
 newNode = (struct node *)malloc(sizeof(struct node));
 if(newNode == NULL) {
   printf("newNode2 not allocated");
   return 0;
 }
 newNode->data = 5;
 newNode->next = NULL;

 struct node *temp = head;
 while(temp->next != NULL){
   temp = temp->next;
 }
 temp->next = newNode;
```



- reuses struct node pointer newNode
- Allocates memory for the new struct and assigns its address to the pointer newNode
- Assigns values to the data and next pointer for newNode
- Allocates memory for a struct pointer named temp, and assigns it the value in head
- Iterates though the list while the next pointer is not NULL
- Sets the next pointer in the previous last element to point to the newNode

C and C++

```
//Add element to the middle – after 2 and before 3

  newNode = (struct node *)malloc(sizeof(struct node));
  if(newNode == NULL) {
    printf("newNode2 not allocated");
    return 0;
  }
  newNode->data = 6;

  temp = head;
  while(temp->data != 2){
    temp = temp->next;
  }
  newNode->next = temp->next;
  temp->next = newNode;
```

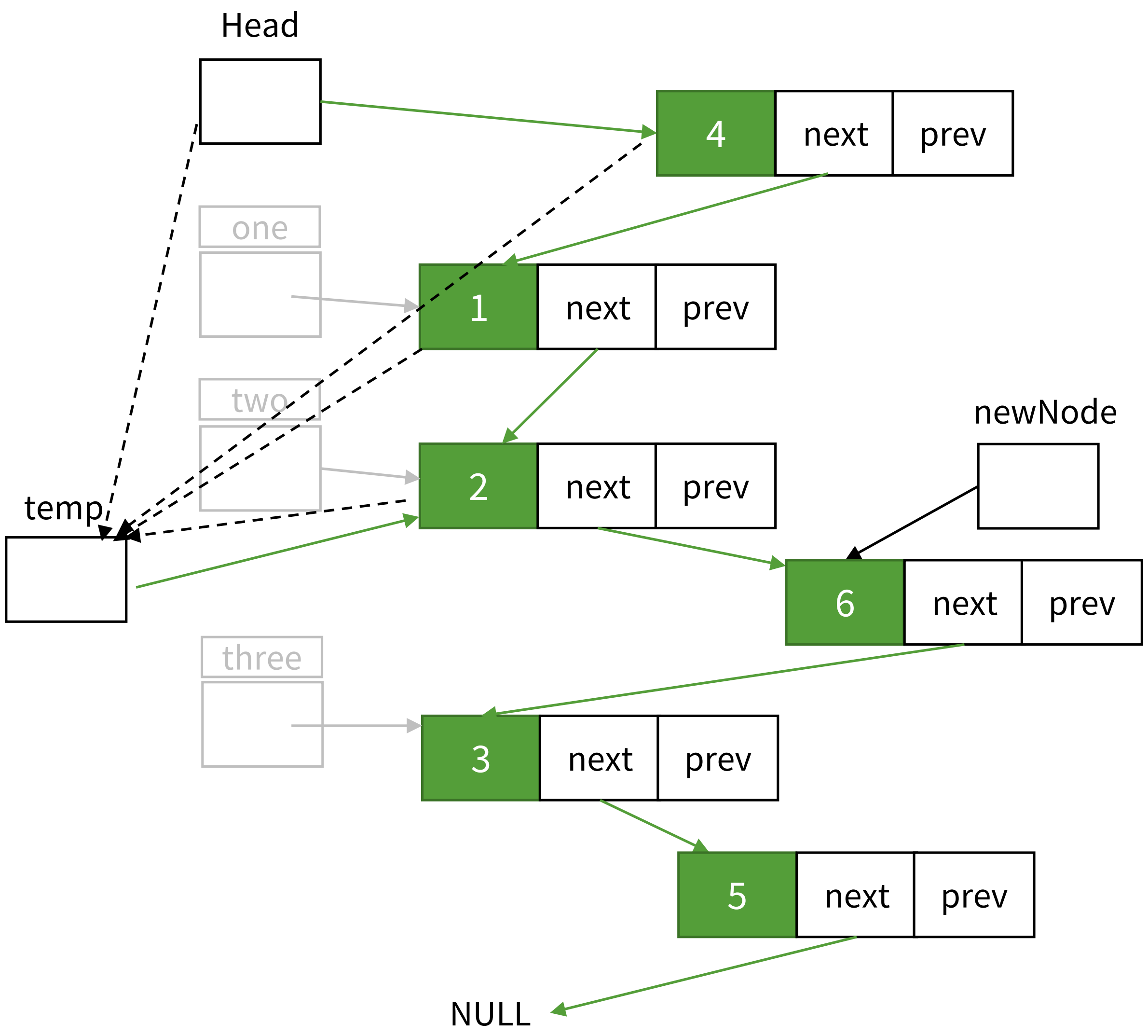


- reuses struct node pointer newNode, allocates memory for the new struct and assigns its address to the pointer newNode
- Assigns a value to the data in newNode
- Assigns the pointer from head to temp, and iterates while the data pointed to from temp is not = 2
- Assigns the values in the next pointers for temp and newNode

```
//iterate through list outputting each element

temp = head;
  printf("\n\nList elements are - \n");
  while(temp != NULL)
  {
   printf("%d --->",temp->data);
   temp = temp->next;
  }
```

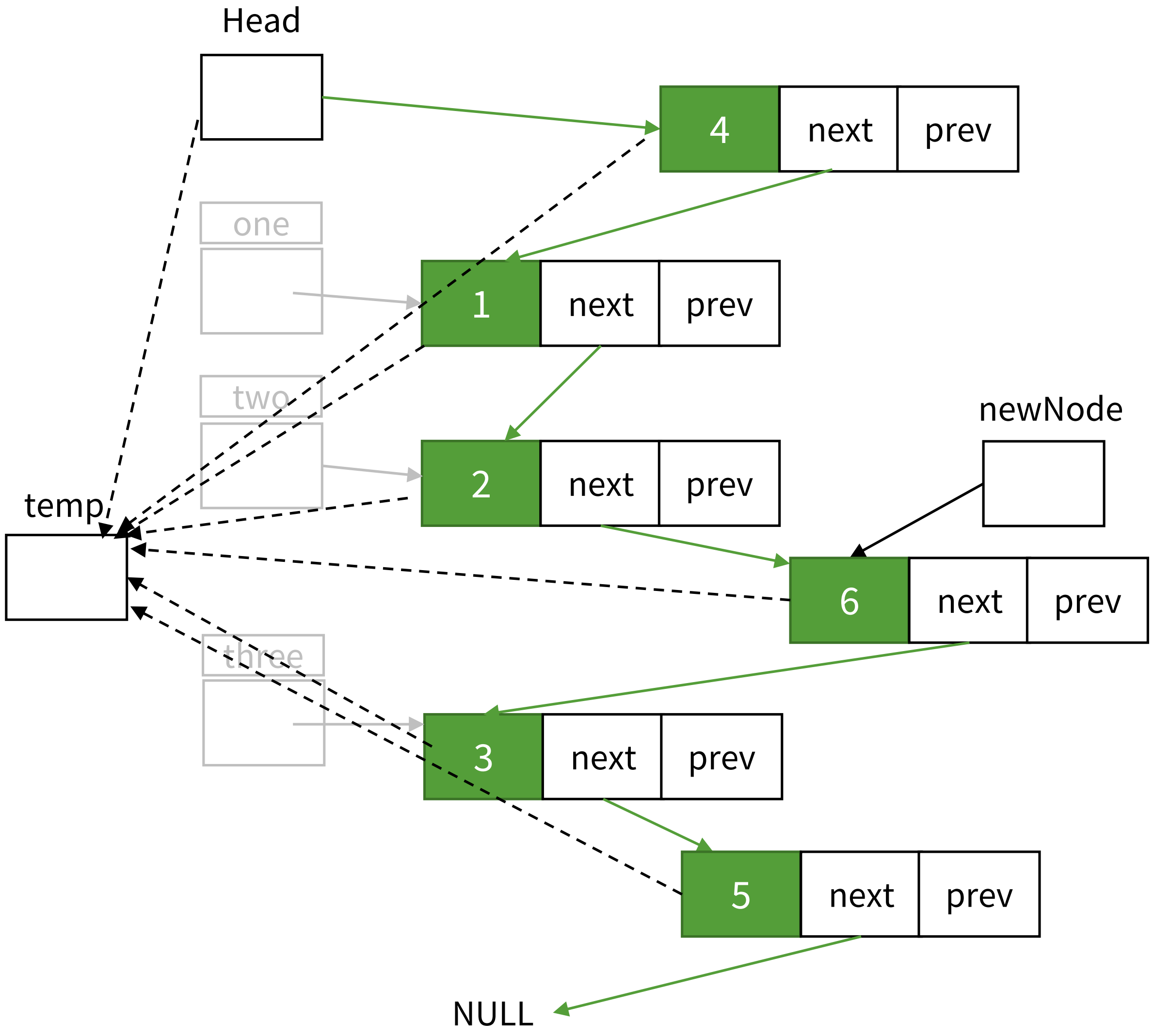
- Iterates through the list while next pointer is not = NULL

```
List elements are -
4 --->1 --->2 --->6 --->3 --->5 --->
Press any key to continue . . .
```



C and C++

```
//free each node
  struct node * nodeToFree;
  temp = head;
  while(temp != NULL)
  {
    nodeToFree = temp;
    temp = temp->next;
    if (nodeToFree != NULL)
      free( nodeToFree);
  }
return 0;
}
```



- Starts at the head
- Iterates through the list while next pointer is not = NULL
- And frees each node

Linked list

- What code is required for removing:
  - first item from the list,
  - last item from the list
  - specific item from the list

- What changes are required in the code to turn this into a doubly linked list?

- What changes are required to order the list into numerical order?
- What changes are required to make this a list of structs?

STL

C++

The STL includes implementations for these data structures

vector
list
Deque (double ended queue)
arrays
forward_list(Introduced in C++11)
queue
priority_queue
Stack