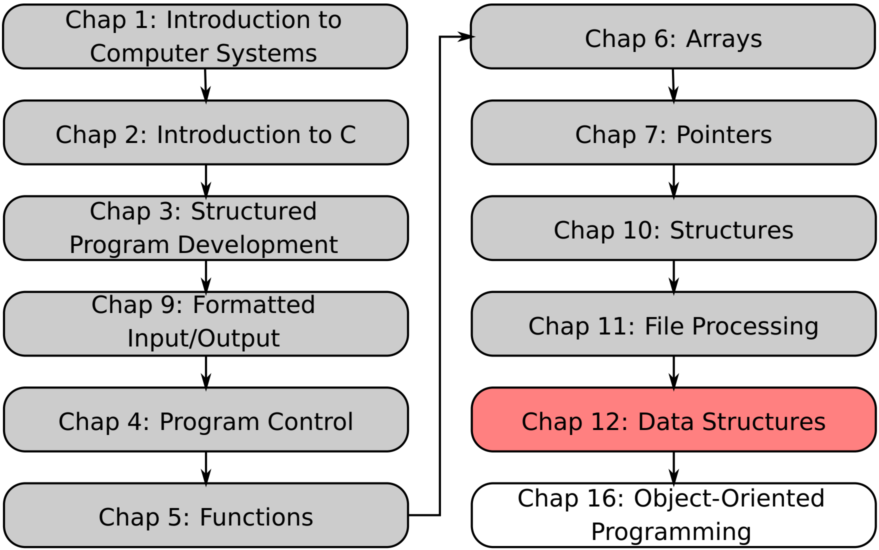
Computer Programming 143 – Lecture 28
Dynamic Data Structures I

Electrical and Electronic Engineering Department
University of Stellenbosch

Prof Johan du Preez
Mr Callen Fisher
Dr Willem Jordaan
Dr Hannes Pretorius
Mr Willem Smit

Copyright & Disclaimer

Module Overview



Lecture Overview

1. Introduction (12.1)
2. Self-referencing structures (12.2)
3. Dynamic memory allocation (12.3)
4. Linked lists (12.4)

12.1 Introduction

Static data types

- Memory requirements remains unchanged for the entire program execution
  - Single subscript arrays
  - Multiple subscript arrays
  - Structures

Dynamic data types

- Memory requirements change during the execution of the program
  - Linked lists
  - Stacks
  - Queues
  - Trees

12.1 Introduction

Linked lists

- Number of elements stored in a “line”
- Insertion and deletion can happen at any point

Stacks

- Used by compilers and operating systems
- Insertion and deletion only takes place at one point (FILO)

Queues

- Used as waiting queue/buffer
- Data inserted at one end (tail) and removed at the other (head) (FIFO)

Trees

- Used to implement very efficient search and sort algorithms
- Binary trees are the most basic type

12.1 Introduction

Dynamic data structures ...

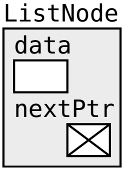
- Each of the data structures have unique and interesting applications
- Almost any program of any significance contains these data structures
- Dynamic data structures rely HEAVILY on pointers and structures
- **Good understanding of pointers and structures is very important**

12.2 Self-referencing structures

Self-referencing structures

- Self-referencing structures contains a member that is a pointer of the same type as the structure
  - Pointer `nextPtr` points to a variable of the same structure type
  - Pointer `nextPtr` can be seen as a link between two elements of that type

```c
struct listNode {
   char data;
   struct listNode *nextPtr;
};
typedef struct listNode ListNode;
```



Self-referencing structures

- Self-referencing structures can therefore be “linked” together to form more useful data structures that can be used more effectively
- A NULL pointer indicates that there is no subsequent node
- A NULL pointer usually indicates the end of the data structure

12.3 Dynamic memory allocation

Dynamic structures and dynamic memory allocation

- Static data structures require variable declaration which ensures that memory is allocated statically
- Dynamic data structures require dynamic memory allocation and memory management
- Dynamic memory allocation makes it possible for a program to request more memory
- Dynamic memory management is made possible by the following functions:
  - `malloc()` – memory allocation
  - `free()` – memory deallocation
  - `sizeof()` – calculates memory requirements

12.3 Dynamic memory allocation

`void * malloc(size_t size);`

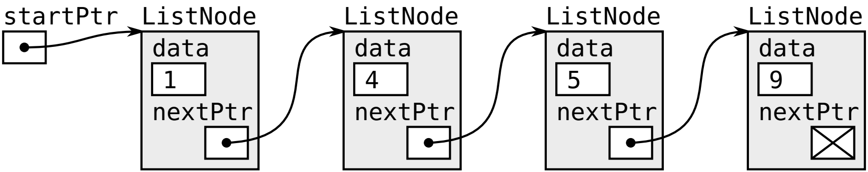
- The argument defines the number of bytes of memory that will be allocated
- The number of bytes that are required are generally calculated using the `sizeof()` function
- The function returns a generic pointer that points to the block of memory that has been allocated – i.e. the address of the first memory position in the block
- The `malloc()` function returns a NULL pointer if it was unable to allocate the memory

`void free( void *ptr );`

- The argument provides the pointer to memory that should be returned to the operating system so that it can be reused
- All allocated memory must be “freed” once it is no longer used – otherwise memory leaks will occur

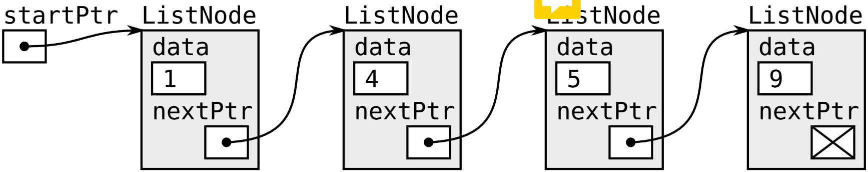
Linked lists

- A linked list is a linear collection of self-referential structures
- Each element that forms part of the list is called a node
- Nodes are connected to one another via pointers called links
- Linked lists are accessed via a pointer to the first element
- Subsequent nodes are accessed via the link pointer member of the current node



Linked lists

- By convention, the link in the last node must be set to NULL
  - This indicates that the current element is the last element
- Data is stored in linked lists dynamically
  - Nodes are created as they are required
- A node can contain any data – even other structures – but must contain a self-referential member
- **NB! Linked lists can only be accessed via a pointer to the first node – the linked list is "lost" if one incorrectly make assignments to this pointer**



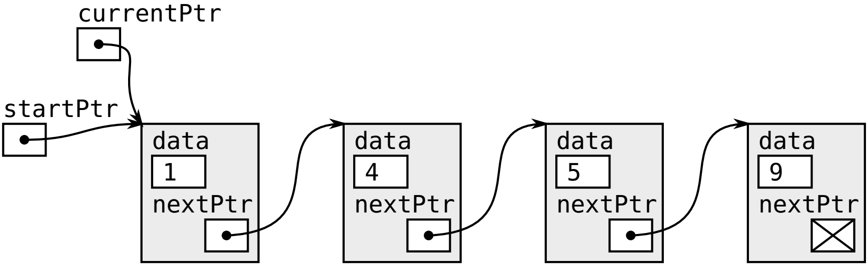
12.4 Linked lists

The most important functions associated with linked lists are

- **searching**
- **insertion**
- **deletion**

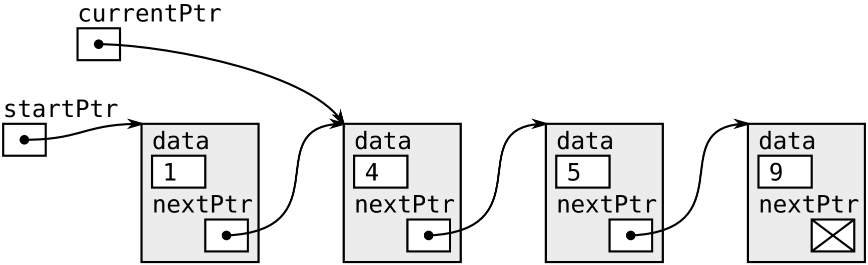
12.4 Searching Linked Lists

```c
value = 5;
currentPtr = startPtr;
while ( currentPtr != NULL && value != currentPtr->data ) {
   currentPtr = currentPtr->nextPtr;
}
```



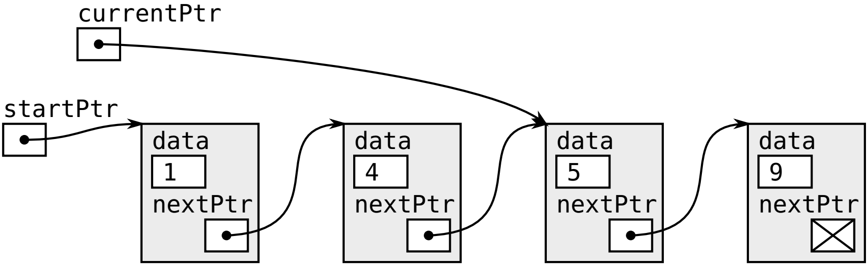
12.4 Searching Linked Lists

```
value = 5;
currentPtr = startPtr;
while ( currentPtr != NULL && value != currentPtr->data ) {
   currentPtr = currentPtr->nextPtr;
}
```



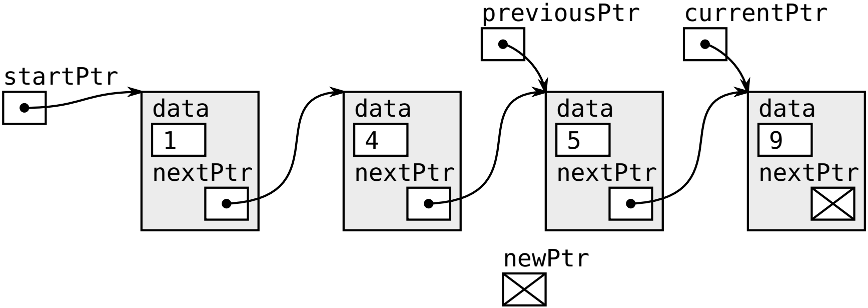
12.4 Searching Linked Lists

```
value = 5;
currentPtr = startPtr;
while ( currentPtr != NULL && value != currentPtr->data ) {
   currentPtr = currentPtr->nextPtr;
}
```



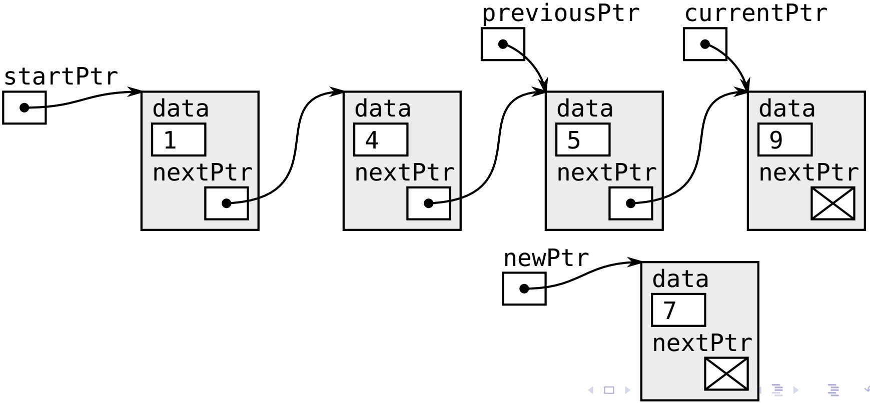
12.4 Linked List Insertion

```
newPtr = malloc( sizeof( ListNode ) );
newPtr->data = value;
newPtr->nextPtr = NULL;
previousPtr->nextPtr = newPtr;
newPtr->nextPtr = currentPtr;
```



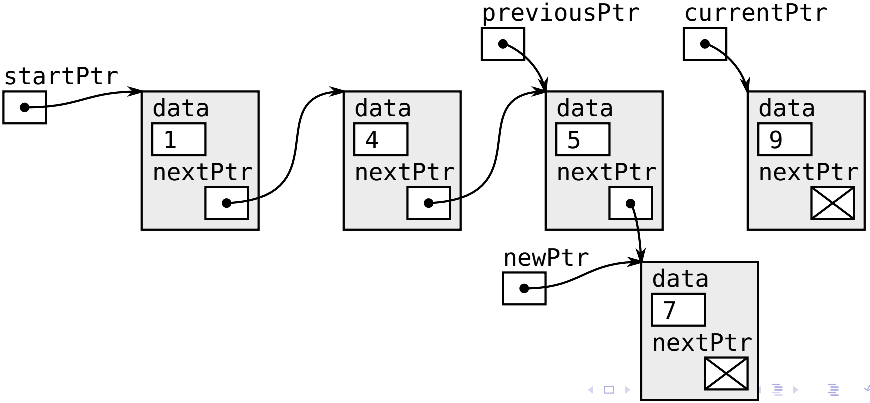
12.4 Linked List Insertion

```
newPtr = malloc( sizeof( ListNode ) );
newPtr->data = value;
newPtr->nextPtr = NULL;
previousPtr->nextPtr = newPtr;
newPtr->nextPtr = currentPtr;
```



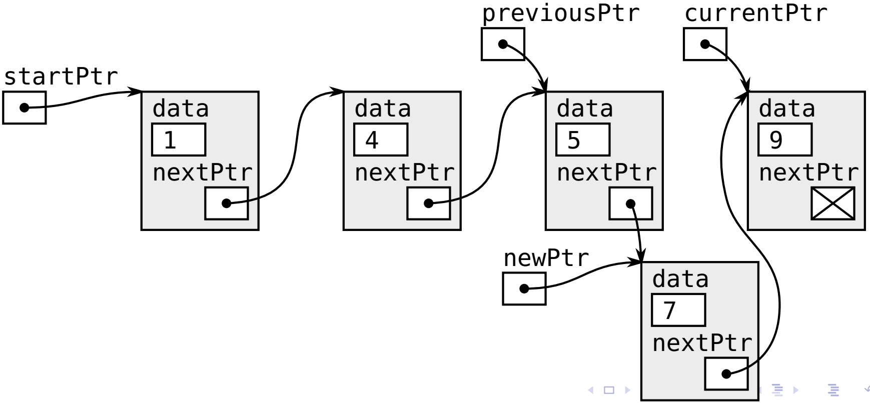
12.4 Linked List Insertion

```
newPtr = malloc( sizeof( ListNode ) );
newPtr->data = value;
newPtr->nextPtr = NULL;
previousPtr->nextPtr = newPtr;
newPtr->nextPtr = currentPtr;
```



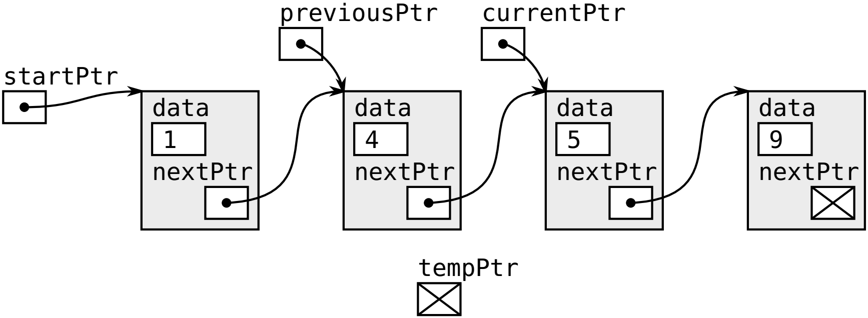
12.4 Linked List Insertion

```
newPtr = malloc( sizeof( ListNode ) );
newPtr->data = value;
newPtr->nextPtr = NULL;
previousPtr->nextPtr = newPtr;
newPtr->nextPtr = currentPtr;
```



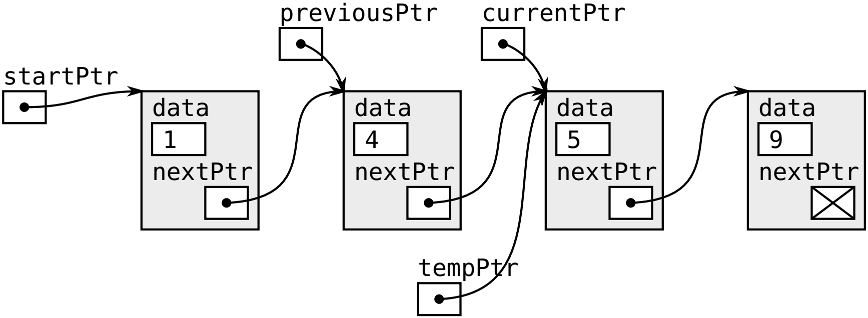
12.4 Linked List Deletion

```
tempPtr = currentPtr;
previousPtr->nextPtr = currentPtr->nextPtr;
free( tempPtr );
```



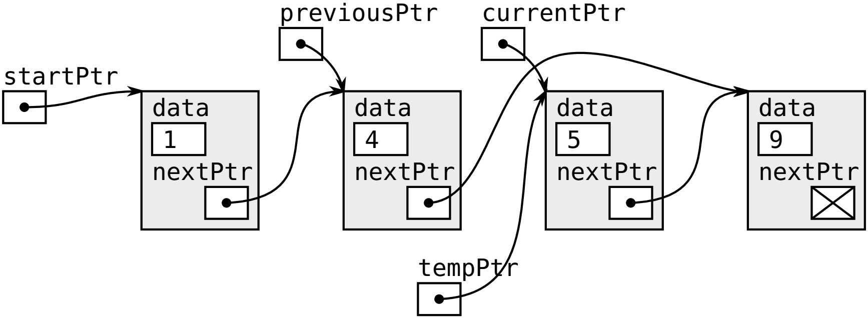
12.4 Linked List Deletion

```
    tempPtr = currentPtr;
    previousPtr->nextPtr = currentPtr->nextPtr;
    free( tempPtr );
```



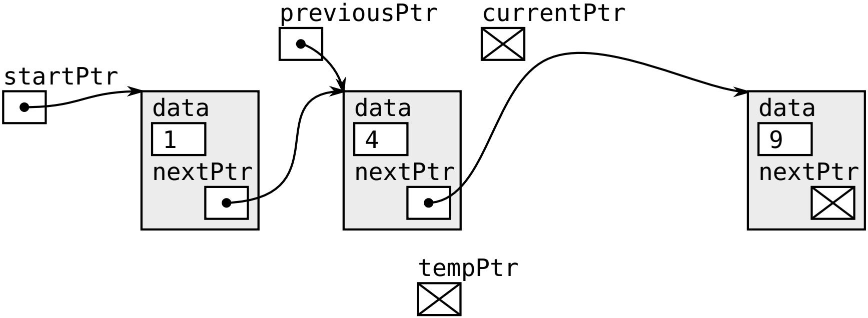
12.4 Linked List Deletion

```
tempPtr = currentPtr;
previousPtr->nextPtr = currentPtr->nextPtr;
free( tempPtr );
```



12.4 Linked List Deletion

```
tempPtr = currentPtr;
previousPtr->nextPtr = currentPtr->nextPtr;
free( tempPtr );
```



12.4 Linked lists

See Fig. 12.3 in Deitel & Deitel for a complete example

Linked lists vs. arrays

- The information in linked lists can be stored in arrays, but linked lists provide many advantages:
  - Linked lists are applicable when the number of elements in the list at any given time is not known in advance
  - Linked lists are dynamic and its size can change with the requirements; arrays are static
  - Arrays can get full; linked lists can almost always grow to accommodate more data

Perspective

Today

Dynamic Data Structures I

- Self-referential structures
- Dynamic data allocation
- Linked lists

Next lecture

Dynamic Data Structures II

- Stacks
- Queues

Homework

1. Study Sections 12.1-12.4 in Deitel & Deitel
2. Do Self Review Exercise 12.4 in Deitel & Deitel
3. Do Exercises 12.6, 12.9 in Deitel & Deitel