Developing Technical Software

Week 4: Dynamic data structures: Linked list

Don’t be accused of cheating!

Academic integrity is a serious matter

- Unethical behaviour can lead to:
  - confiscation of possessions
  - suspension from university premises
  - failing a subject
  - exclusion from your course

SWINBURNE UNIVERSITY OF TECHNOLOGY

It’s best to be informed

- Swinburne has designed the online Academic Integrity Training Module to give you an understanding of the pitfalls and repercussions of:
  - poor referencing and plagiarism
  - exam misconduct
  - collusion and copying
  - other unethical activities

- So you can confidently maintain your academic integrity throughout your studies.

SWINBURNE UNIVERSITY OF TECHNOLOGY

To complete the module, simply

- Go to Canvas
- Log in
- Navigate to the module on your Dashboard
- Begin!

- There are 4 topics and each includes a short quiz.

SWINBURNE UNIVERSITY OF TECHNOLOGY

Who should complete the module and when?

- All Unilink students are expected to undertake this module.

- It’s important to do so before you begin your first assignment.

SWINBURNE UNIVERSITY OF TECHNOLOGY

Questions?

- Ask now!

- Or please contact a faculty Academic Development Adviser (ADA): www.swinburne.edu.au/ada

SWINBURNE UNIVERSITY OF TECHNOLOGY

Revisit

- Functions – Recursive function
- Arrays
- Typecasting
- Typedef
- Structures
- Pointers
- Dynamic Memory Allocation

Arrays

- Holds more than one element
- Holds only homogenous elements (very data type specific)
- Stores elements in contiguous manner (Continuous or sequential).
- Index based (can access the elements faster)
- It is an internal pointer variable that holds the base address of the memory block

Self-referential structures

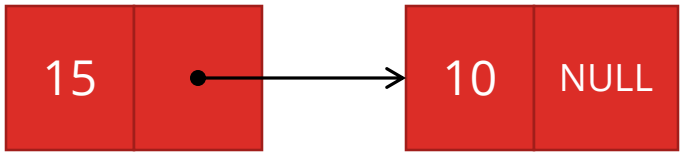
- a self-referential structure contains a pointer member that points to a structure of the same structure type.
- The following definition defines a type, struct node.

```
struct  node {
        int  data;
        struct node  *nextPtr;

};
```

Self-referential structures

- Self-referential structures can be *linked* together to form useful data structures, e.g. linked lists, queues, stack, etc.
- The following figure illustrates two self-referential structure objects to form a list.



NULL pointer

- A NULL pointer is placed in the linked member of the second self-referential structure, to indicate the end of a data.
- Not setting the link in the last node of a list to NULL will lead to runtime errors.

Dynamic Memory Allocation

- Static memory allocation means fixed size
- Dynamic memory allocation means the size can be increased or decreased based on our requirement
- To allocate memory dynamically, we need to take the help of predefined functions available in <stdlib.h> library

Dynamic Memory Allocation

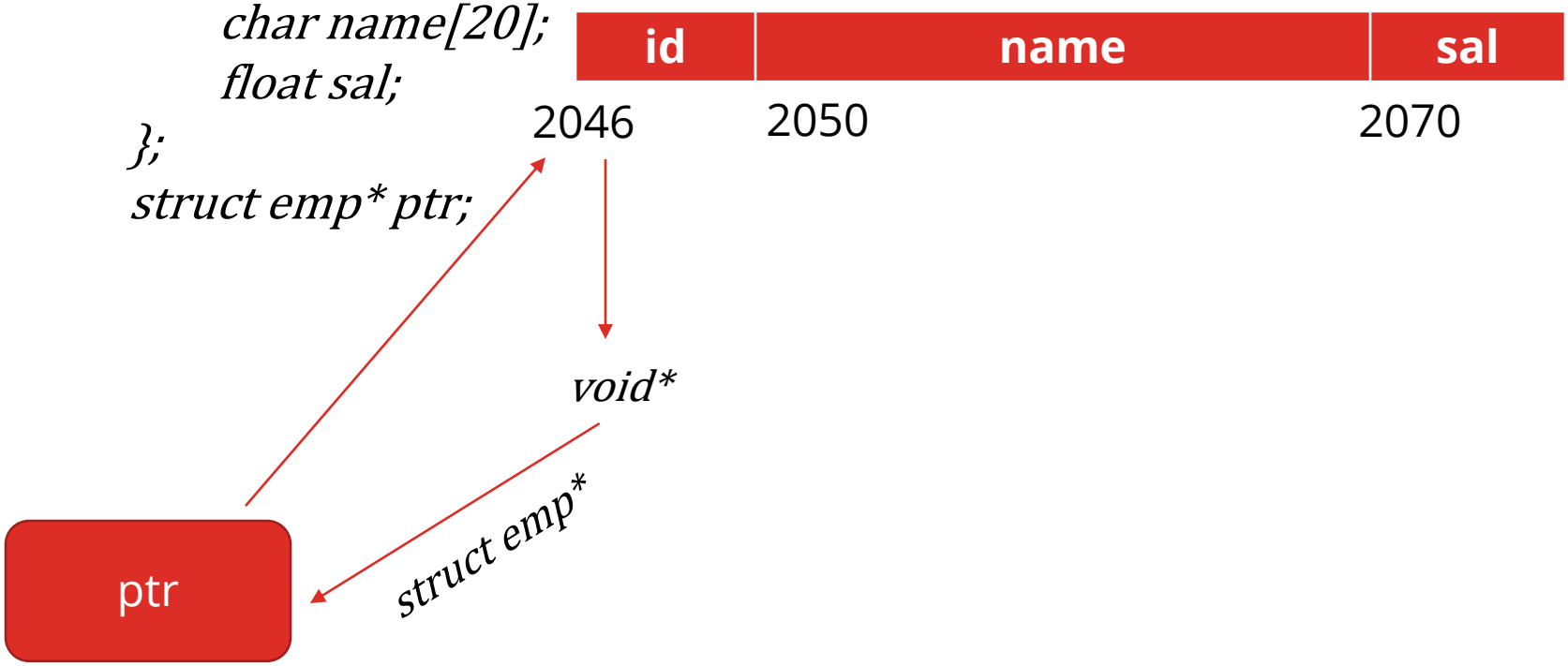
- malloc() – allocates memory to user defined datatypes such as structures
- calloc() – allocates memory to derived datatypes such as arrays
- realloc() – increases/decreases the size of an array
- free() – Releases the memory

malloc() – allocate memory to structures

- void* malloc(size_t size);
- Void is the return type and a generic pointer which means it can point to any datatype such as int, char etc
- Size_t is an unsigned positive integer
- Size is the size of the memory block needed
- On successful allocation, it returns a base address of memory block
- On failure, it returns a null pointer

malloc() – Example

```
struct emp{
        int id;
        char name[20];
        float sal;
};
struct emp* ptr;
```




```
ptr = (struct emp*) malloc (sizeof(struct emp));
```

Data Structures

- A way of organizing and storing data in order to efficiently manage
- Its not a programming language
- A data structure is a set of algorithms that can be used to structure the information
- To structure data, many number of algorithms are proposed and can be referred to as abstract data types (set of instructions)

What is a Linked list?

- a linear collection of self-referential structures, called nodes(memory blocks), connected by pointer links
- accessed via a pointer to the 1st node of the list
- Subsequent nodes are accessed via the link pointer member stored in each node.
- the link pointer in the last node of a list is set to NULL (as the end of the list)

Array -- Linked list

The size of an array created at compile time, however, cannot be altered. Arrays can become full.

An array can be declared to contain more elements than the number of data items expected, but this can waste memory.

Insertion and deletion in a sorted array can be time consuming - all the elements following the inserted or deleted element must be shifted appropriately.

Linked lists become full only when the system has insufficient memory to satisfy dynamic storage allocation requests.

Linked lists can provide better memory utilization in these situations.

Insertion and deletion are easier in Linked-list

Array -- Linked list

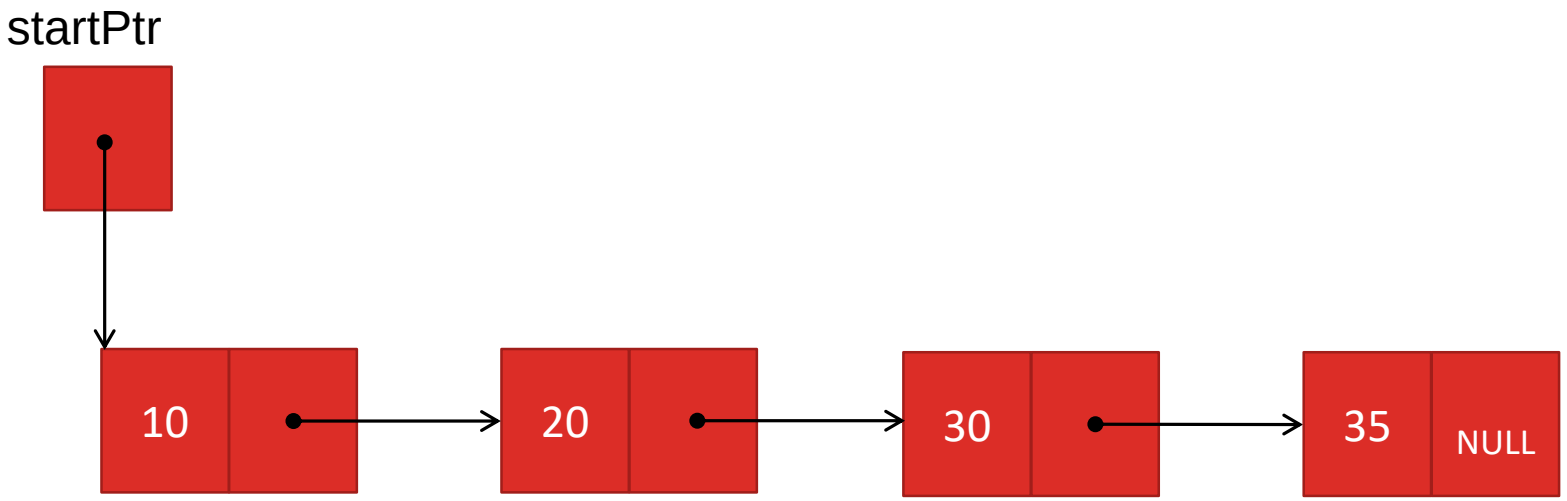
The elements of an array are stored contiguously in memory. This allows immediate access to any array element because the address of any element can be calculated directly based on its position relative to the beginning of the array.

Linked lists do not afford such immediate access to their elements.
Linked-list nodes are normally not stored contiguously in memory.
Logically, however, the nodes of a linked list appear to be contiguous.

More…

- a linear **dynamic** collection
- It is impossible to create a static linked list
- Not possible to fix a limit for linked list size
- In linked lists, insertions and deletions are much faster than arrays
- No direct access to elements of the linked list but only traversing or travelling through all nodes

A linked list with several nodes



To begin with …

```
#include<stdio.h>
#include<stdlib.h>
/* This is the node structure*/
struct node {
  int data;
  struct node *next;
} *start = NULL;

void create(void);
void display(void);

int main()
{
        create();
        display();
        return 0;
}
```

To create node

```
void create()
{
        char ch;
        do{
                struct node *new_node, *current;
                new_node = (struct node *) malloc(sizeof(struct node));
                printf(" \nEnter the data : ");
                scanf("%d", &new_node -> data);
                new_node -> next = NULL;
                if (start == NULL) {
                        start = new_node;
                        current = new_node;
                }
                else {
                        current -> next = new_node;
                        current = new_node;
                }
                printf("\nAnother node? (y/n): ");
                ch = getche();
        } while (ch != 'n');
}
```

To display nodes

```
void display()
{
     struct node *new_node;
     printf(" \nThe Linked List : ");
     new_node = start;
     while (new_node != NULL){
          printf("%d -> ", new_node-
>data);
          new_node = new_node -> next;
     }
     printf("NULL");
}
```

Output

- Enter the data : 10
- Another node? (y/n): y
- Enter the data : 20
- Another node? (y/n): y
- Enter the data : 30
- Another node? (y/n): n
- The Linked List :
- 10 -> 20 -> 30 -> NULL

Why use a linked list?

- Lists of data can be stored in arrays, but linked lists provide several advantages:
  - dynamic, each node is created as necessary.
  - the length of a list can increase or decrease at execution time as necessary.
  - appropriate when the number of data elements to be represented in the data structure is unpredictable.
  - A node can contain data of any type including other structs.
  - can be maintained in sorted order by inserting each new element at the proper point in the list.

Primary Functions

- Insert - insert a character in the list in alphabetical order
- Delete - delete a character from the list
- is_empty - determines whether the list is empty (i.e., the pointer to the first node of the list is NULL). If the list is empty, 1 is returned; otherwise, 0 is returned.
- print_list - to print list

- Attached is an example C program (linkedList.c) to manipulate a list of characters with these functions (and illustrated with figures in following slides).

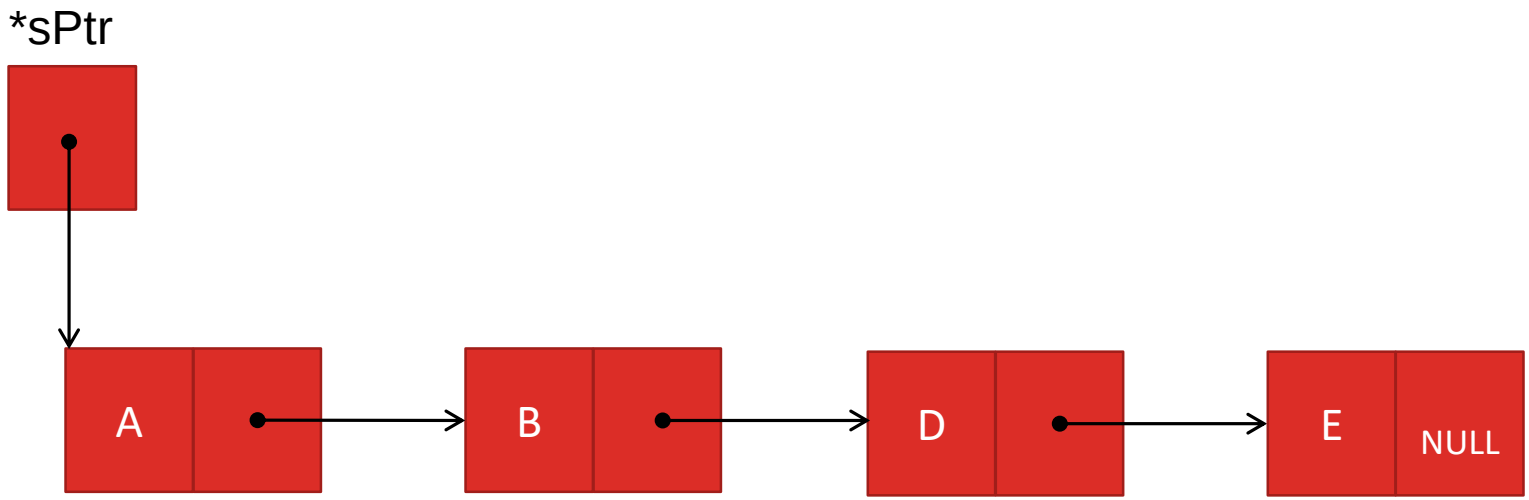
Function insert

The steps for inserting a character in the list are as follows:

1. *Create a node* by calling malloc, assigning to newPtr the address of the allocated memory, assigning the character to be inserted to newPtr->data, and assigning NULL to newPtr->nextPtr.
2. Initialize previousPtr to NULL and currentPtr to*sPtr – the pointer to the start of the list. Pointers previousPtr and currentPtr store the locations of the node *preceding* the insertion point and the node *after* the insertion point.
3. While currentPtr is not NULL and the value to be inserted is greater than current Ptr->data, assign currentPtr to previousPtr and advance currentPtr to the next node in the list. This locates the *insertion point* for the value.
4. If previousPtr is NULL (line 107), insert the new node as the *first* node in the list. Assign *sPtr to newPtr->nextPtr (the *new node link* points to the *former first node)* and assign newPtr to *sPtr (*sPtr points to the *new node).* Otherwise, if previousPtr is not NULL, the new node is inserted in place.
5. Assign newPtr to previousPtr->nextPtr (the *previous* node points to the *new* node) and assign currentPtr to newPtr->nextPtr (the *new* node link points to the *current* node).

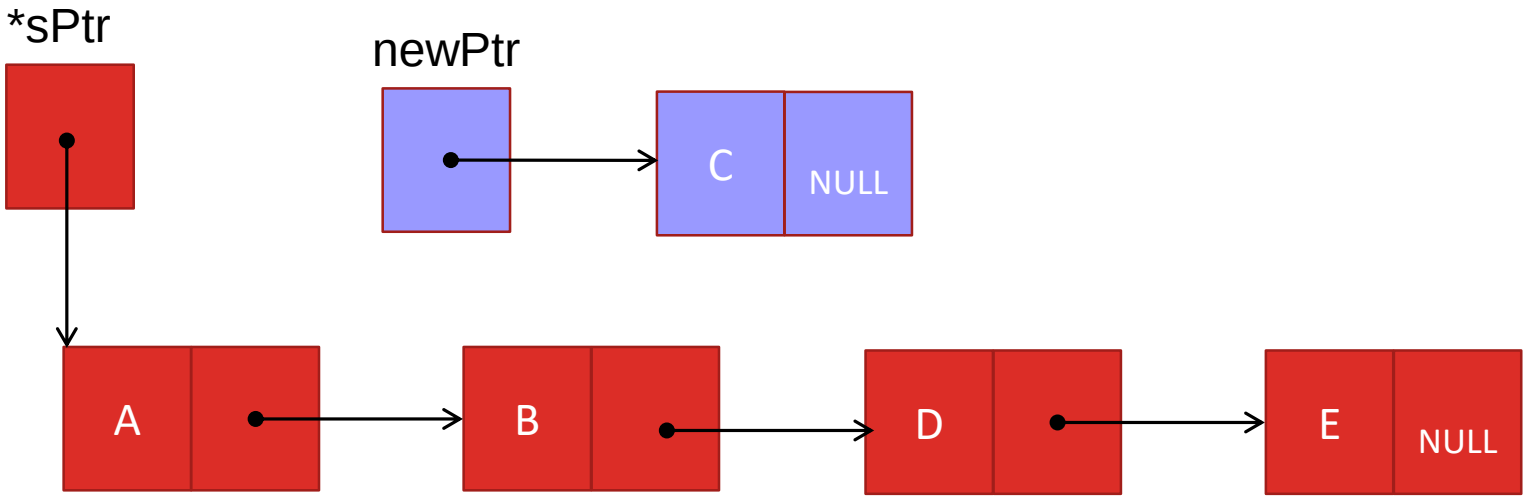
An Ordered List

A node containing the character 'C' will be inserted into an ordered list that contains characters of 'A', 'B', 'D', and 'E'.



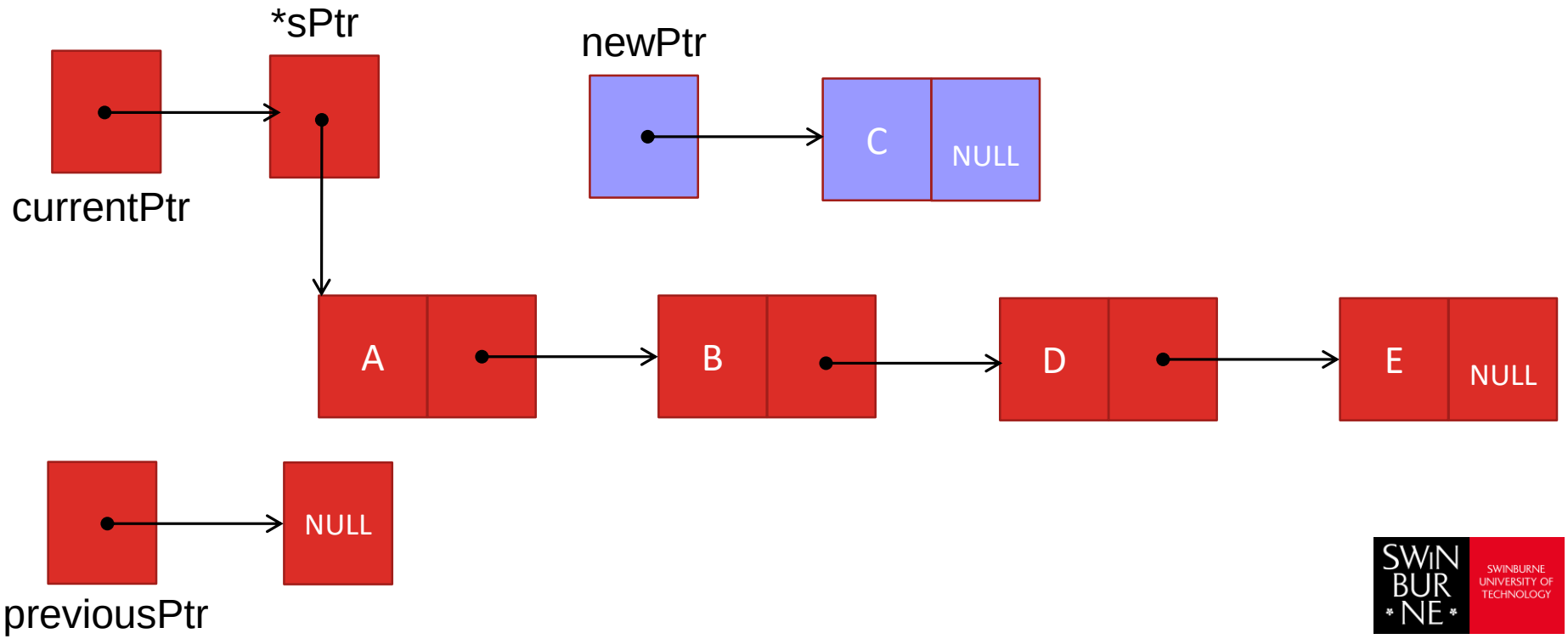
Create a node

Create a node by calling malloc, assigning to newPtr the address of the allocated memory, assigning the character to be inserted to newPtr->data, and assigning NULL to newPtr->nextPtr.



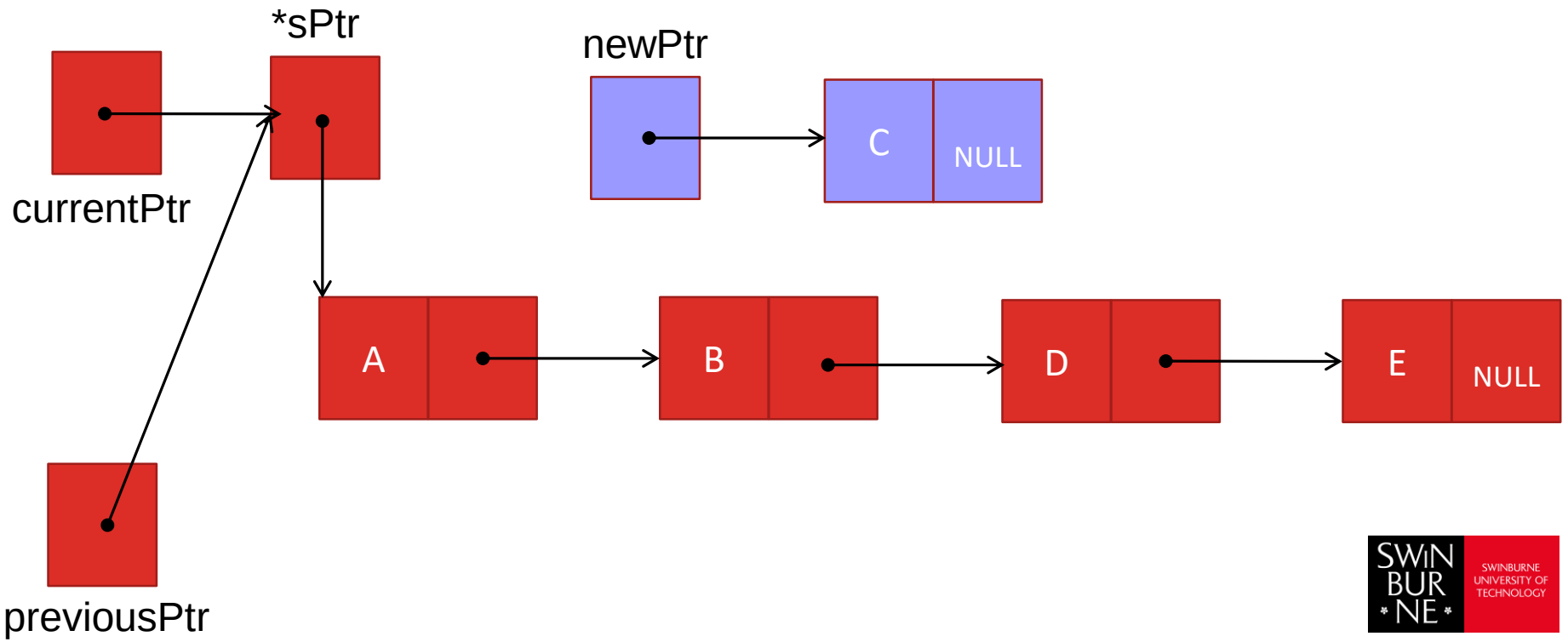
Initialize previousPtr & currentPtr

Initialize previousPtr to NULL and currentPtr to*sPtr – the pointer to the start of the list. Pointers previousPtr and currentPtr store the locations of the node *preceding* the insertion point and the node *after* the insertion point



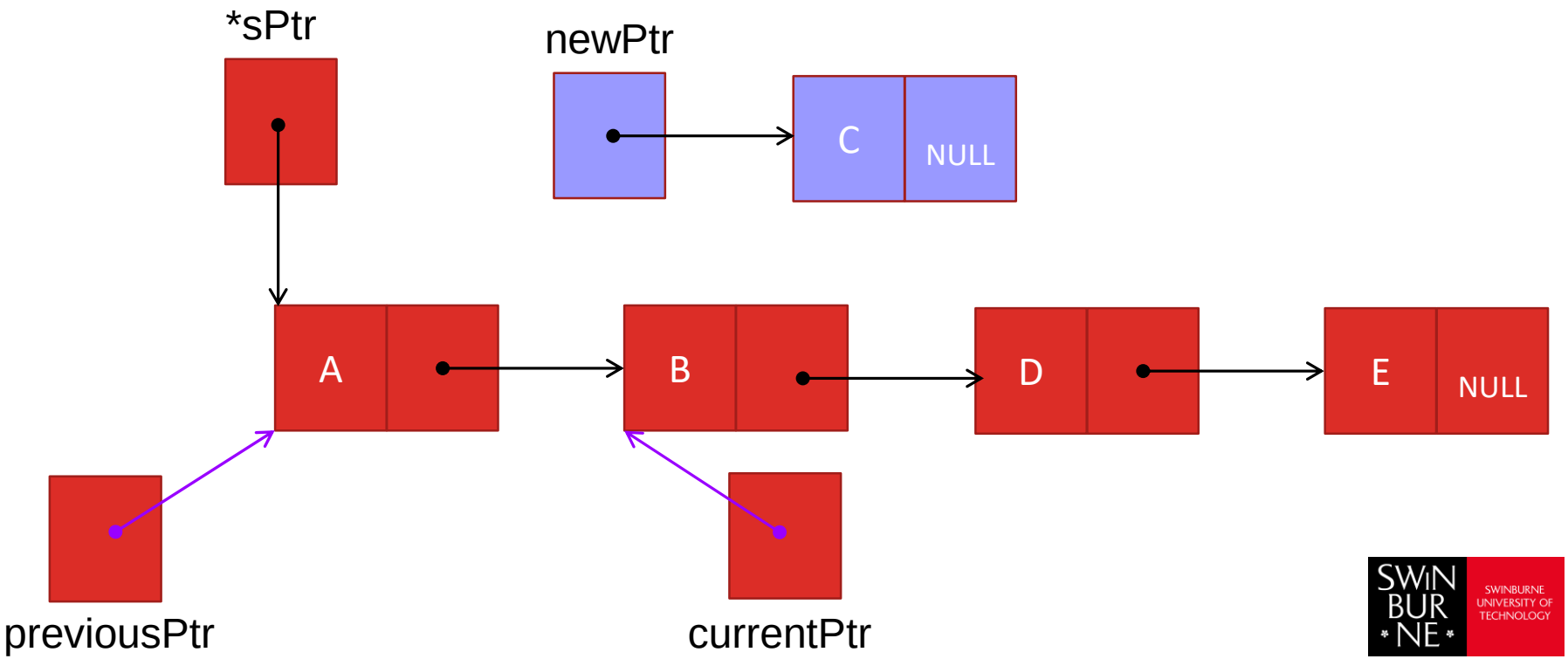
Locates the Insertion Point

While currentPtr is not NULL and the value to be inserted is greater than current Ptr->data, assign currentPtr to previousPtr and advance currentPtr to the next node in the list. This locates the *insertion point* for the value.



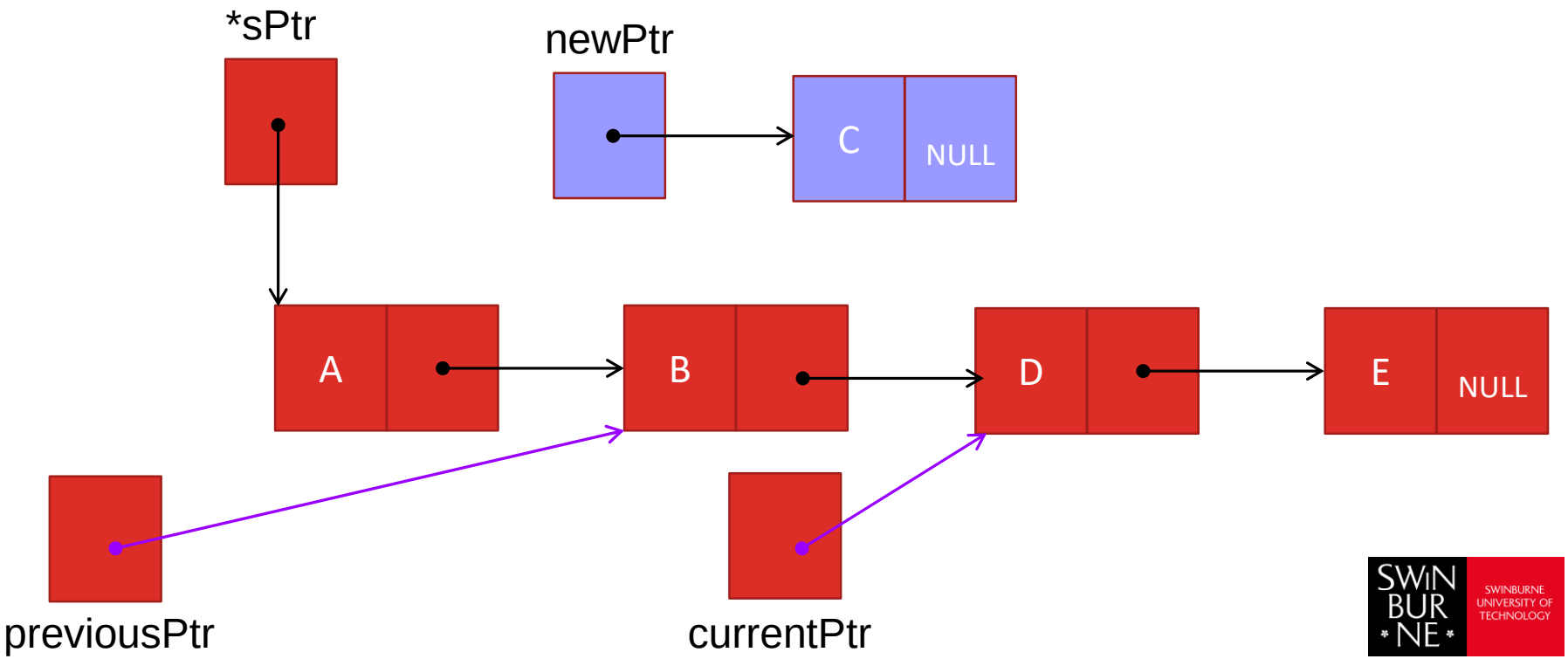
Locates the Insertion Point

While currentPtr is not NULL and the value to be inserted is greater than current Ptr->data, assign currentPtr to previousPtr and advance currentPtr to the next node in the list. This locates the *insertion point* for the value.



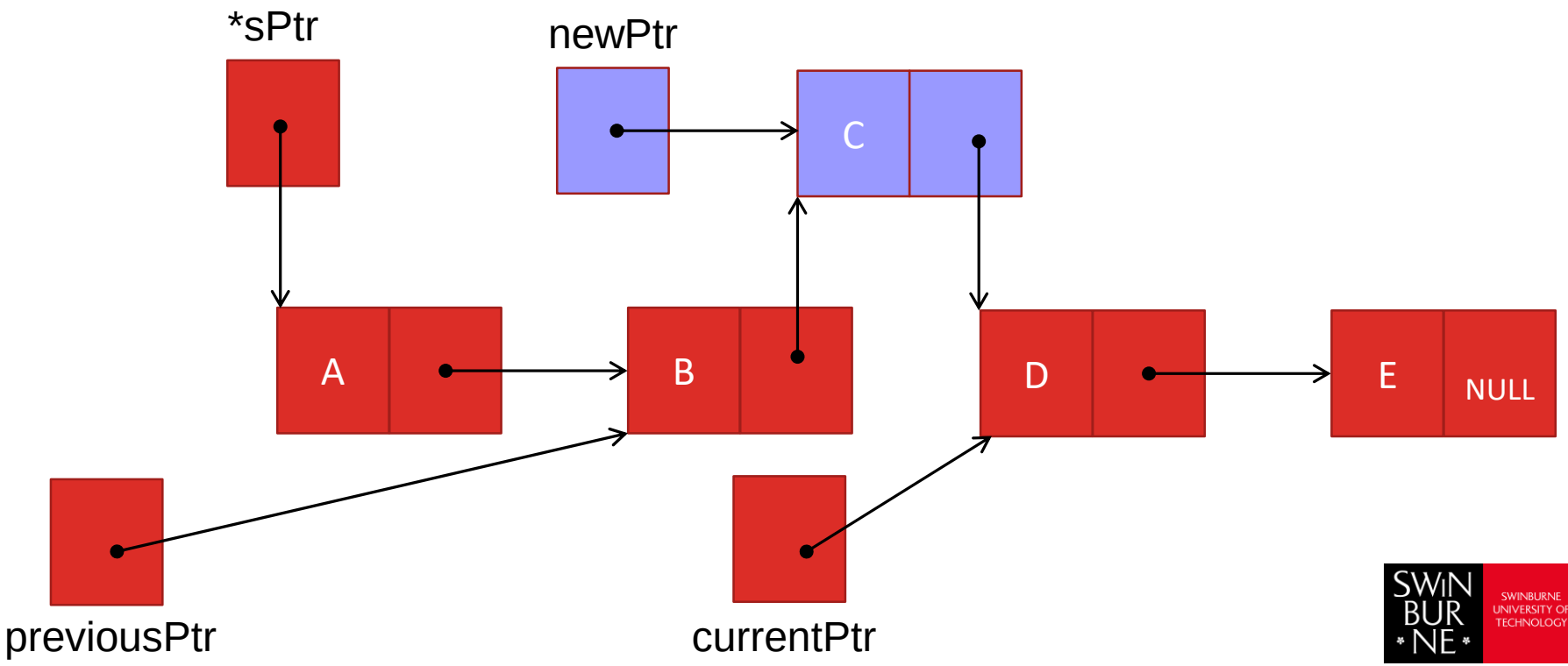
Locates the Insertion Point

While currentPtr is not NULL and the value to be inserted is greater than current Ptr->data, assign currentPtr to previousPtr and advance currentPtr to the next node in the list. This locates the *insertion point* for the value.



Assign newPtr

If previousPtr is NULL, insert the new node as the *first* node in the list. Assign *sPtr to newPtr->nextPtr (the *new node link* points to the *former first node)* and assign newPtr to *sPtr (*sPtr points to the *new node).* Otherwise, if previousPtr is not NULL, the new node is inserted in place. Assign newPtr to previousPtr->nextPtr (the *previous* node points to the *new* node) and assign currentPtr to newPtr->nextPtr (the *new* node link points to the *current* node).



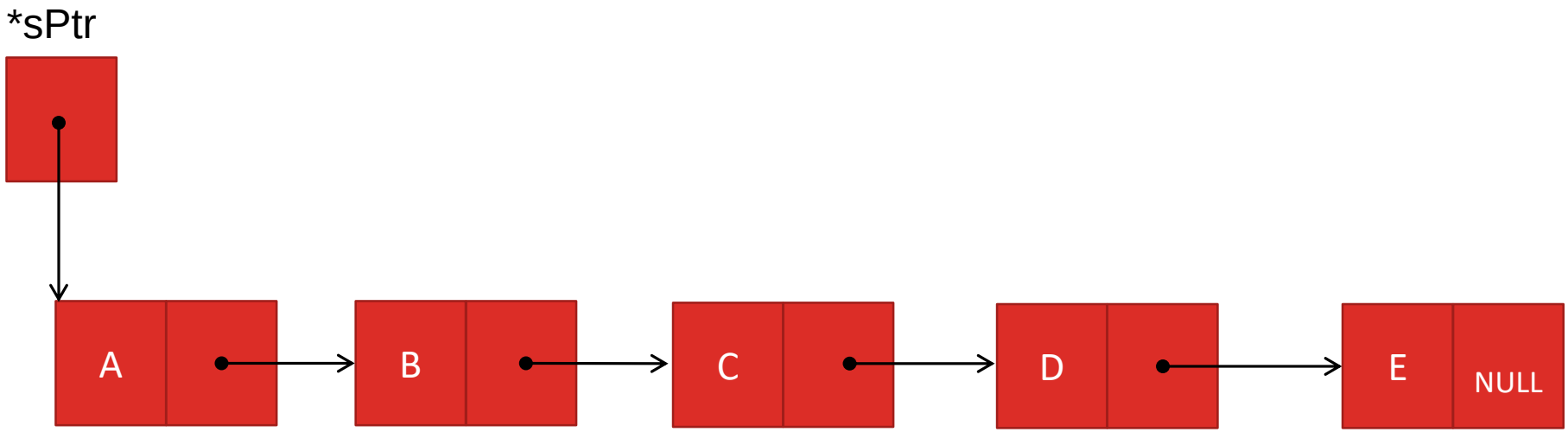
Function delete

The steps for deleting a character from the list are as follows:

1. If the character to be deleted matches the character in the *first* node of the list, assign *sPtr to tempPtr (tempPtr will be used to free the unneeded memory), assign (*sPtr) ->nextPtr to *sPtr (*sPtr now points to the *second* node in the list), free the memory pointed to by tempPtr, and return the character that was deleted.
2. Otherwise, initialize previousPtr with *sPtr and initialize currentPtr with (*sPtr)->nextPtr to advance to the second node.
3. While currentPtr is not NULL and the value to be deleted is not equal to currentPtr->data, assign currentPtr to previousPtr and assign currentPtr->nextPtr to currentPtr. This locates the character to be deleted if it's contained in the list.
4. If currentPtr is not NULL, assign currentPtr to tempPtr, assign currentPtr->nextPtr to previousPtr->nextPtr, *free* the node pointed to by tempPtr, and return the character that was deleted from the list. If currentPtr is NULL, return the null character ('\0') to signify that the character to be deleted was *not* found in the list.

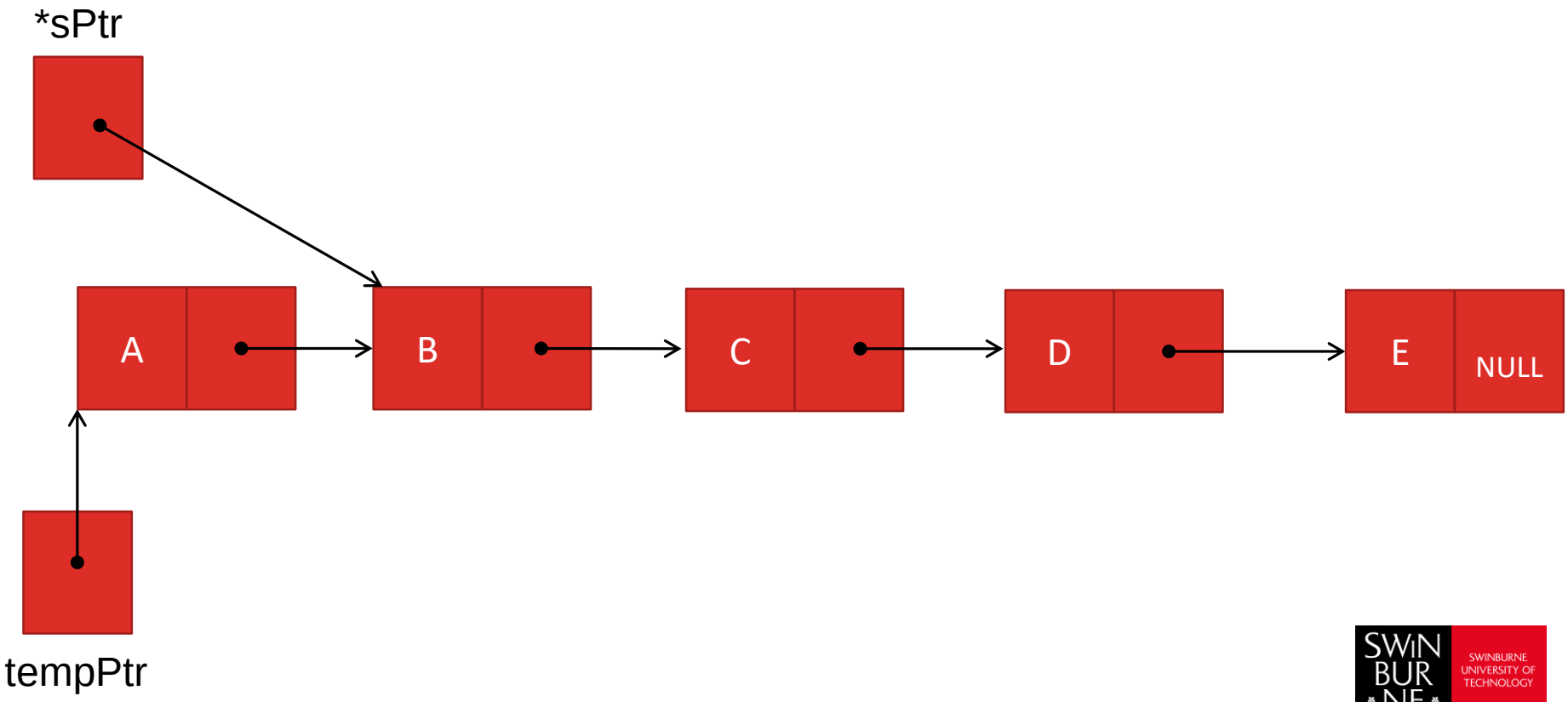
Initial Linked List

A list contains characters of 'A' to 'E'



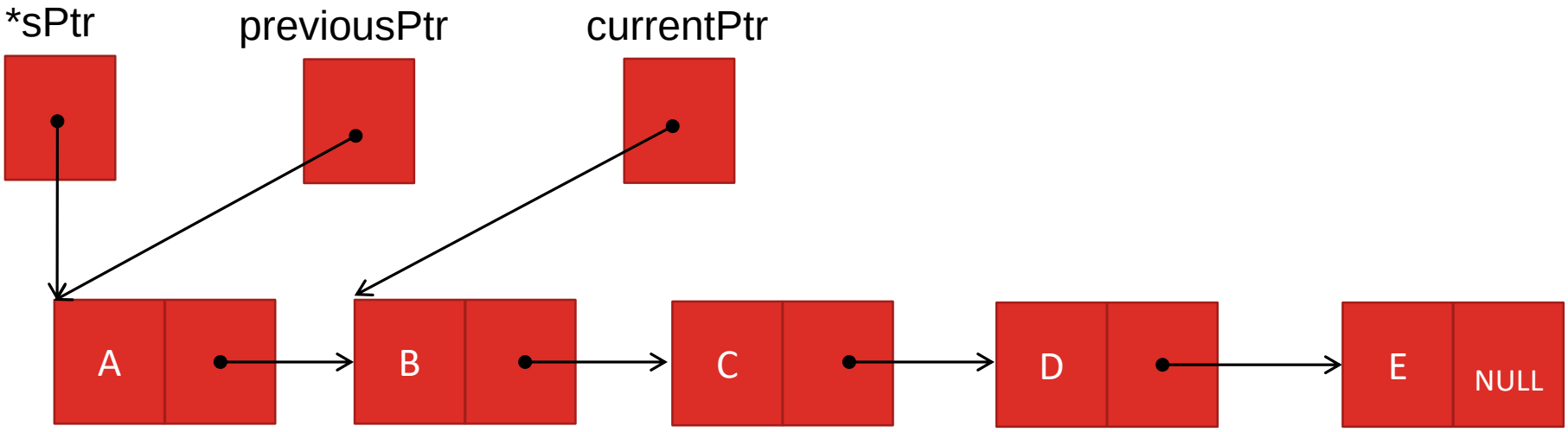
To delete the first node

If the character to be deleted matches the character in the *first* node of the list, 'A' in this case, assign *sPtr to tempPtr (tempPtr will be used to free the unneeded memory), assign (*sPtr) ->nextPtr to *sPtr (*sPtr now points to the *second* node in the list), free the memory pointed to by tempPtr, and return the character that was deleted.



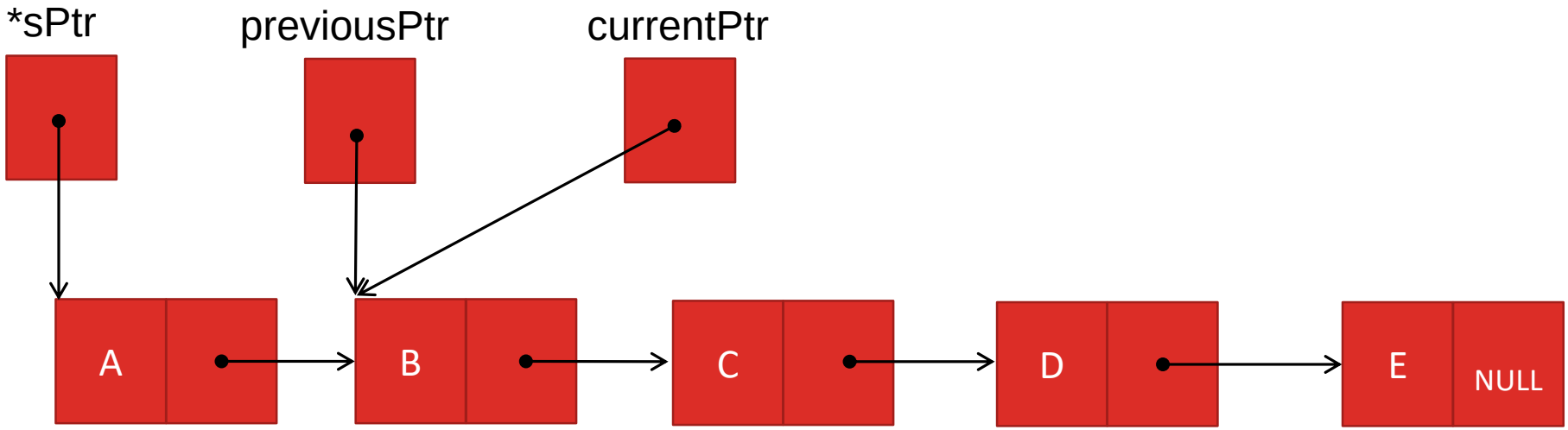
Advance to the next node

If the character to be deleted is 'C', two pointers, i.e. previousPtr and currentPtr are needed. Initialize previousPtr with *sPtr and initialize currentPtr with (*sPtr)->nextPtr.



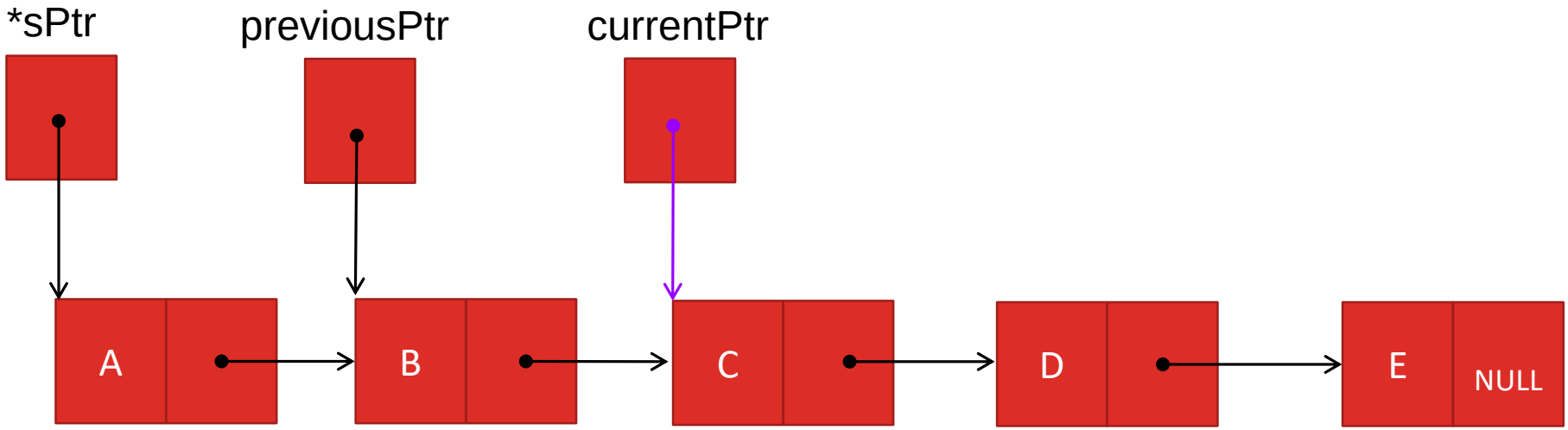
Locates the character

While currentPtr is not NULL and the value to be deleted is not equal to currentPtr->data, assign currentPtr to previousPtr and assign currentPtr->nextPtr to currentPtr. This locates the character to be deleted if it's contained in the list.



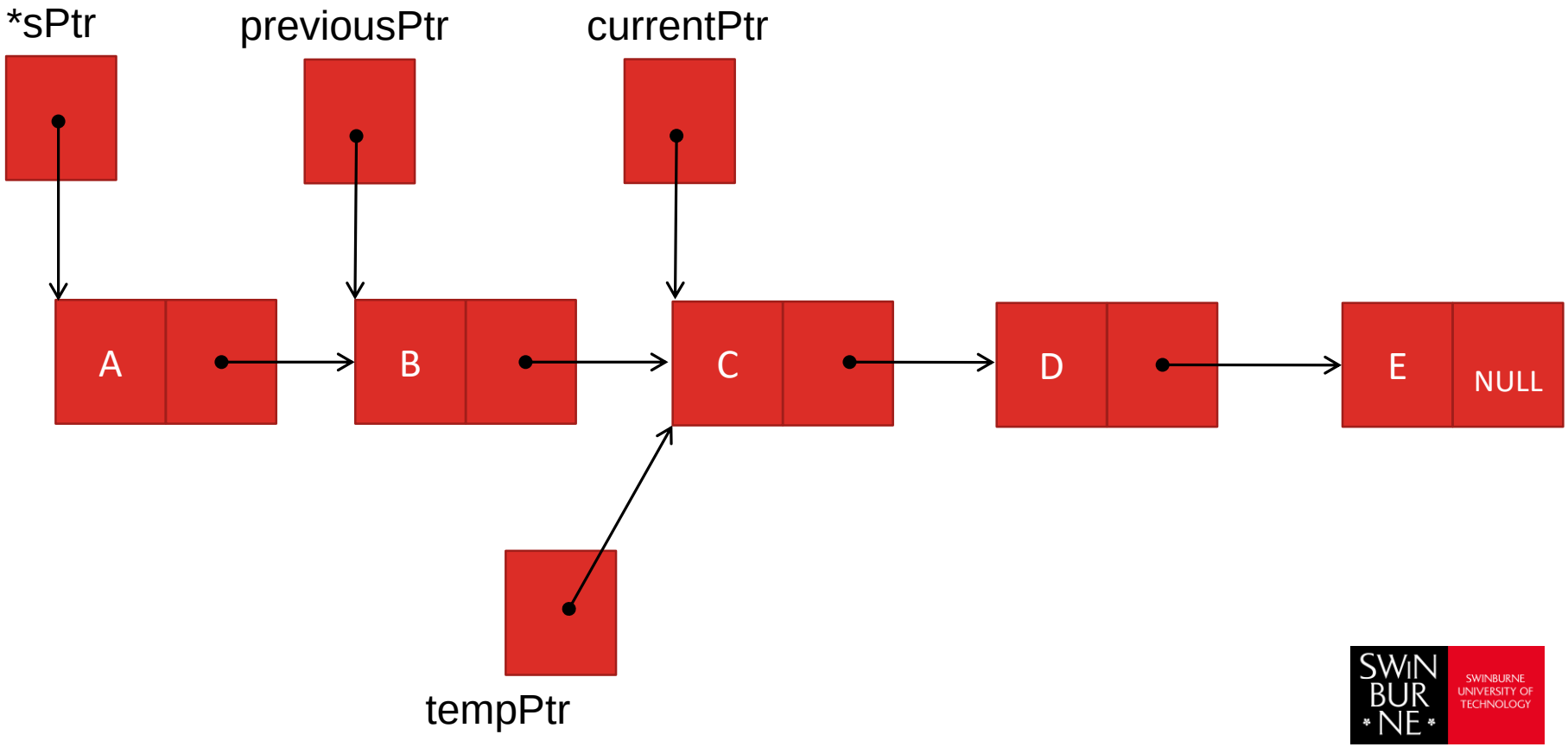
Locates the character

While currentPtr is not NULL and the value to be deleted is not equal to currentPtr->data, assign currentPtr to previousPtr and assign currentPtr->nextPtr to currentPtr. This locates the character to be deleted if it's contained in the list.



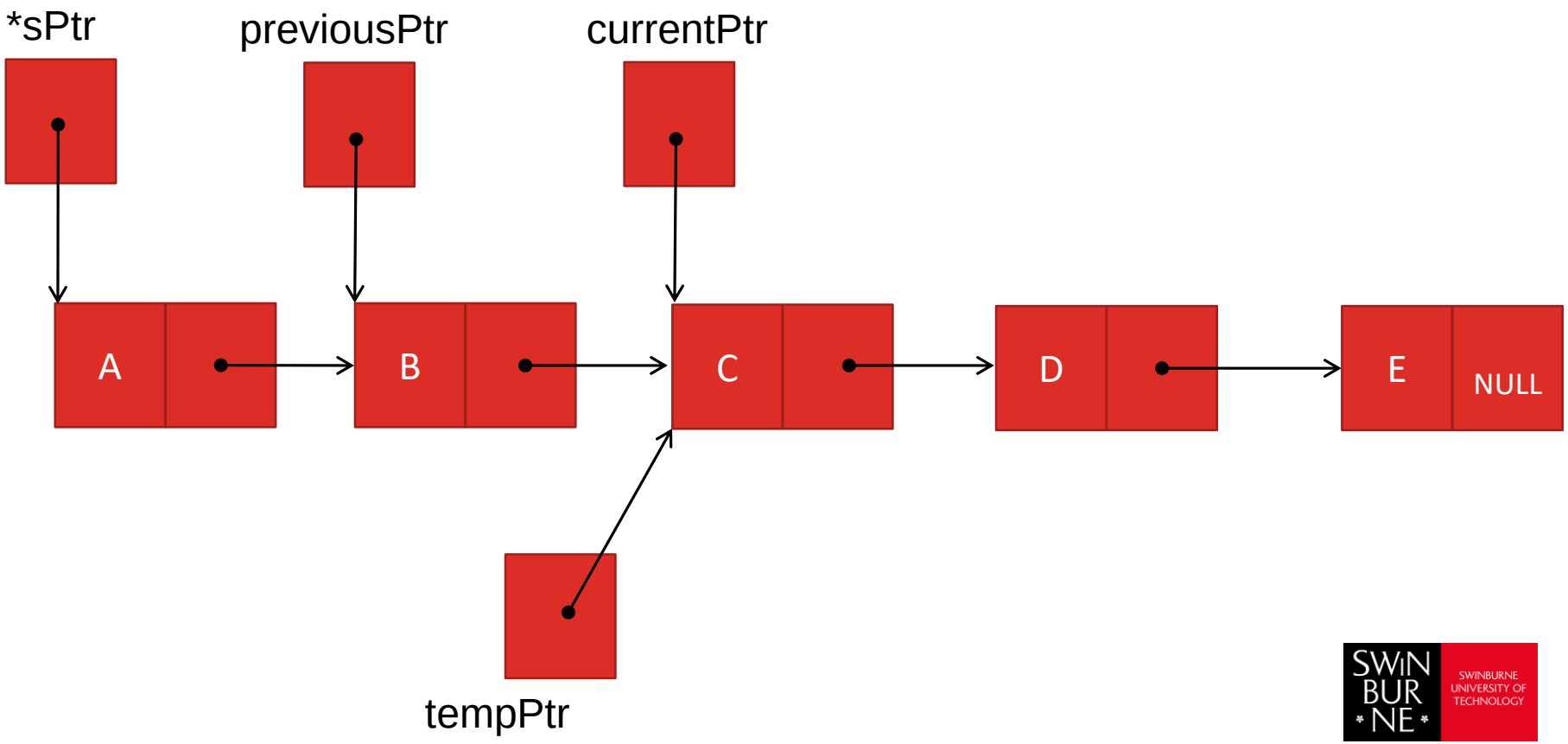
Delete a node

If currentPtr is not NULL, assign currentPtr to tempPtr, assign currentPtr->nextPtr to previousPtr->nextPtr, *free* the node pointed to by tempPtr, and return the character that was deleted from the list.



Character not found

If currentPtr is NULL, return the null character ('\0') to signify that the character to be deleted was *not* found in the list.



Function printList

- It receives a pointer to the start of the list as an argument and refers to the pointer as currentPtr.
- It first determines whether the list is *empty* and, if so, prints “List is empty.” and terminates.
- Otherwise, it prints the data in the list.
- While currentPtr is not NULL, the value of currentPtr->data is printed by the function, and currentPtr->nextPtr is assigned to currentPtr to advance to the next node.
- If the link in the last node of the list is not NULL, the printing algorithm will try to print *past the end of the list*, and an error will occur.

Thank you

SWINBURNE UNIVERSITY OF TECHNOLOGY