RMIT UNIVERSITY
Computer Science and Information Technology

Advanced Programming Techniques

(a.k.a. Programming in ANSI / ISO C)

Module 08 ─ Dynamic Memory Management

"....Every morning is the dawn of a new error…"

-- anon.

Memory allocation – implicit approach

- Variables declared in programs are allocated the necessary memory space – the memory allocation is implicit in the declaration.

  e.g. integer 4 or 8 bytes

  pointer 4 bytes

- Be aware though, that a pointer declaration allocates memory for the <u>pointer only</u>. It does not allocate memory for the variable pointed to by the pointer!

Memory allocation – explicit approach

- Memory may be explicitly allocated through the use of dynamic (run-time) memory allocation library functions.
- The typical explicit memory allocation transaction consists of:
  - Programmer requests a piece of memory
    - Size of requested memory should be provided by the programmer
  - System returns (if the request is granted)
    - a piece of memory of the given size
    - a pointer to its first byte
    - or; in case of error the returned pointer is set to NULL

Memory allocation – comparison

- Implicit method
  - Data size has to be defined before compilation
  - Data size is rigid/fixed/static, not modifiable (at run-time)

- Explicit method
  - Data size is defined run-time
  - Allows the efficient management of variable/dynamic size data
    - Data always fits into the allocated space
    - System resources not wasted: no extra memory is allocated

Memory terms and concepts

- Any program can be divided into two parts: code and data
- Data can be divided into three parts according to where it is stored:
  - **Static data**: its storage space is compiled into the program
  - **Stack data**: allocated run-time, to hold information used inside functions; it is managed in stack space
  - **Heap data**: also allocated run-time, allows the programmer to dynamically manage memory allocations; it is managed in heap space

Dynamic Memory

- Dynamic memory – what, why and how?
- `malloc()`
- `free()`
- `calloc()`
- `realloc()`
- Self-referential structures
- A simple linked list

Dynamic memory – what, why and how?

- What ?
  - Memory that is allocated during the execution of a program
  - i.e. dynamically / “at run-time”
- Why ?
  - We often don’t know how much memory we might need
- How ?
  - Using the C library functions

malloc()

- Function prototype:

```
void *malloc(size_t numBytes);
```

- Attempts to allocate `numBytes` of memory
- If successful, return value is the address of the (first byte of) the memory allocated
- Otherwise, NULL is returned … and should always be tested for

malloc() – example

```
#define STR_SIZE 100
...
char *ptr;
if ( (ptr = (char *) malloc(STR_SIZE)) == NULL )
     printf( "Unable to allocate %d bytes\n", STR_SIZE);
else
     strcpy(ptr, "Hello world");
```

Notes:
- dynamic memory is <u>un-named</u> (unlike a variable) and contiguous
- the only access to this memory is via pointer `ptr`
- cast to `(char *)` is not essential since ANSI C first standardised
- if you assign some other value to `ptr` then access to this memory is lost … "forever!"
- unpredictable program behaviour can occur if the allocated memory space is overrun

free()

- Function prototype:

```
void free(void *ptr);
```

- De-allocates the previously allocated dynamic memory pointed to by `ptr`
- It is good programming practice to free any dynamically allocated memory once it is no longer required
- `ptr` is ***not*** automatically re-set to NULL after calling `free()`
- Be very careful never to access memory that has been `free()`'ed

calloc()

- Function prototype:

```
void *calloc(size_t numObj, size_t sizeObj);
```

- Attempts to allocate memory for `numObj` objects, each of size `sizeObj` bytes
- If successful, return value is the address of the memory, otherwise NULL
- Memory allocated, if granted, is guaranteed to be contiguous and initialised with zeros
- This function is often used for allocating dynamic array

calloc() – example

```
int *ptr, numItems = 100;

ptr = (int *) calloc(numItems, sizeof *ptr);
if ( ptr == NULL )
      printf("Unable to get %d bytes\n",
              (numItems * sizeof(*ptr)));
else
      for (j=0; j<numItems; j++)
            ptr[j] = 0.0;  /* or *(ptr + j) = 0.0;  */
```

realloc()

- Function prototype:

```
void *realloc(void *ptr, size_t numBytes);
```

- Attempts to re-allocate memory pointed to by `ptr` to size `numBytes` bytes
- If successful, return value is the address of the memory, otherwise NULL
- Memory allocated, if granted, is guaranteed to be contiguous – i.e. can be used like an array
- Data values (but not location!) of previous memory is preserved

realloc() – example

```
#define  INC  100

int *array = NULL, *temp, num, avail = 0, used = 0;

while( scanf("%d", &num) == 1)
{
  if ( used == avail )
  {
    avail += INC;
    if ((temp = (int *)realloc(array, avail * sizeof *temp))==NULL)
    {
      printf(stderr, "Too many\n");  break;
    }
    else
      array = temp;
  }
  array[used++] = num;
}
```

realloc() – notes

- If request cannot be satisfied, NULL is returned and old region is left untouched
- If first argument is NULL, behaves like `malloc()`
- If new size is smaller than old size then truncates
- If new is larger than old then:
  - extra memory is appended to previous
  - cannot guarantee pointer(s) into old region are still valid (i.e.array and temp may be different)
  - reallocated memory is a contiguous block

Memory leaks

- A memory leak occurs when memory (dynamically) allocated is never `free()`'ed. This can be a particular problem if a program runs for a long time, as eventually memory will become unavailable.
- If nothing points to a piece of allocated memory, the memory is inaccessible but remains allocated, i.e. it is not available for re-allocation. For example:

```
char *strptr = (char *) malloc (strlen(text));
if (strptr)
    strcpy (strptr, text);
strptr = &something_else;
```

- Now the memory allocated to `strptr` has "leaked away".

Checking for memory leak

- Manually, by going through the code
- Automatically, using tools (there are many tools available)
  - Eg `valgrind` under Linux

```
$ valgrid –leak-check=full ./a.out
     ...
     ==34105== LEAK SUMMARY:
     ==34105==    definitely lost: 12 bytes in 1 blocks
     ==34105==    indirectly lost: 0 bytes in 0 blocks
     ==34105==      possibly lost: 0 bytes in 0 blocks
     ==34105==    still reachable: 0 bytes in 0 blocks
```

IntList - intlist-main.c (revisited)

```
#include <stdio.h>
#include <stdlib.h>
#include "intlist.h"

#define SIZE 10

int main(void)
{
   IntList il;
   int i;
```

IntList - intlist-main.c (cont'd)

```
if (MakeList(&il, SIZE) == FAILURE)
{
   fprintf(stderr, "MakeList(): failed\n");
   return EXIT_FAILURE;
}

/* fill the IntList with random numbers */

for(i=0; i<SIZE; i++)
{
   if (AddList(&il, rand()) == FAILURE)
   {
      fprintf(stderr, "AddList(): failed\n");
      break;
   }
}
```

IntList -- intlist-main.c (cont'd)

```
   printf("IntList size is %u\n", SizeList(&il));

   DisplayList(&il);

   FreeList(&il);

   return EXIT_SUCCESS;
}
```

intlist.h, using a dynamic array

```
/*
 * IntList
 * -- simple unordered array implementation that uses dynamic memory
 *
 * type:
 *  IntList
 *
 * constants:
 *     SUCCESS
 *     FAILURE
 *
 * interface routines:
 *    int MakeList(IntList* pil, int size)
 *       attempts to initialise an IntList variable (passed by address)
 *       if insufficient memory is available for the size list requested
 *       then MakeList() retuns FAILURE, otherwise it returns SUCCESS
 *       MakeList() must be applied to an IntList before any other
 *       function.
```

intlist.h (cont'd)

```
 *  void FreeList(IntList* pil)
 *       attempts to reset an IntList variable (passed by address) to
 *       the "empty" state, depending on the implementation this may
 *       involve deallocation of memory. IntList must be initialised
 *       with MakeList() again before use. Typically FreeList() is
 *       the last function to be applied to an IntList
 *
 *  int AddList(IntList* pil,int data);
 *       attempts to add a new int (data) to an IntList variable
 *       (passed by address). If the addition was successful AddList()
 *       will return SUCCESS, otherwise FAILURE
 *
 *  void DisplayList(IntList* pil);
 *       displays all integers currently stored in the IntList
 *       values displayed one per line on standard output
 *
```

intlist.h (cont'd)

```
 * unsigned SizeList(IntList* pil);
 *      returns the current size of the IntList
 *      ie. how many data items are currently stored within the list
 *
 */

#include <stdlib.h>

#define INTLISTSIZE 100

#define SUCCESS 1
#define FAILURE 0

typedef struct
{
   int *array;
   size_t size;
} IntList;
```

intlist.h (cont'd)

```
int MakeList(IntList*,int);

void FreeList(IntList*);

int AddList(IntList*,int);

void DisplayList(IntList*);

unsigned SizeList(IntList*);

/*
 *
 */
```

intlist.c using a dynamic array

```
/* IntList
 * -- simple unordered array implementation that uses dynamic memory
 */

#include <stdio.h>
#include <stdlib.h>
#include "intlist-dyn-array.h"

int MakeList(IntList* pil)
{
   if ((pil->array = malloc(INTLISTSIZE * sizeof *(pil->array)) == NULL)
   {
      return FAILURE;
   }
   pil->size = 0;
   return SUCCESS;
}
```

intlist.c (cont'd)

```
void FreeList(IntList* pil)
{
   free(pil->array);
   pil->size = 0;
}

int AddList(IntList* pil, int num)
{
   if (pil->size >= INTLISTSIZE)
   {
      return FAILURE;
   }

   pil->array[pil->size] = num;
   pil->size += 1;

   return SUCCESS;
}
```

intlist.c (cont'd)

```
void DisplayList(IntList* pil)
{
   int i, size, *array;

   size = pil->size;
   array = pil->array;

   for(i=0; i<size; i++)
   {
      printf("%d\n", array[i]);
   }
}

unsigned SizeList(IntList* pil)
{
   return pil->size;
}
```

Self-referential structures

- Structs which have a member that can point to itself - e.g.

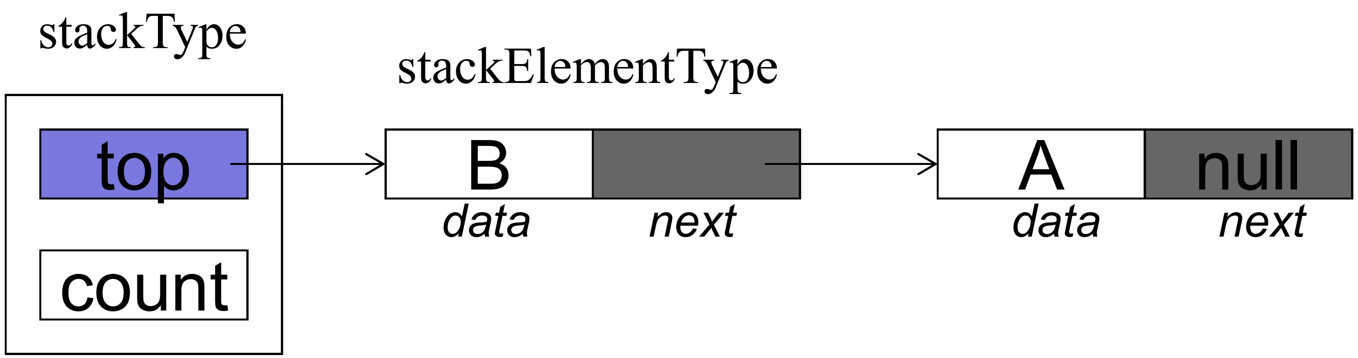
```
typedef struct stack_element
{
    char data;
    struct stack_element *next;
}   StackElementType;
```

- Useful for building complex data structures such as lists, trees and graphs
  - E.g, a simple linked stack

A stack

```
typedef struct stack_element
{
   char data;
    struct stack_element *next;
}  stackElementType;


typedef struct stack
{
   int count;
   stackElementType *top;
} stackType;
```

A stack



- ‘top’ – points to the top of the stack
- New elements are pushed (i.e. added) to the ‘top’ of the stack
- An element is popped (i.e. removed) from the ‘top’ of the stack
- ‘count’ is the count of how many items are in the stack

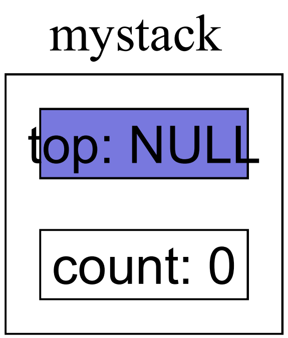
A stack

```
void reset(stackType *stk) {
   stk->count = 0;
   stk->top = NULL;
}

void push (char c, stackType *stk) {
   stackElementType *p;

   if ((p = malloc(sizeof(stackElementType))) == NULL) {
       fprintf(stderr,"push failed to malloc\n");
       exit(EXIT_FAILURE);
   }
   p->data = c;
   p->next = stk->top;
   stk->top = p;
   stk->count++;
}
```

A stack

```
void reset(stackType *stk) {
        stk->count = 0;
        stk->top = NULL;
}
```

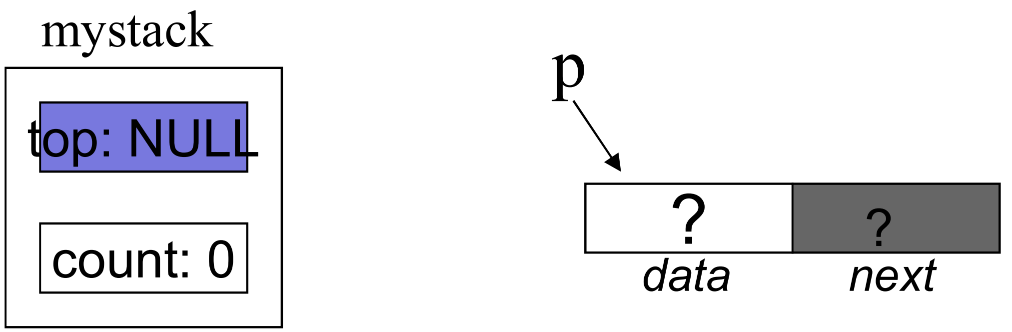
reset(&mystack)



A stack

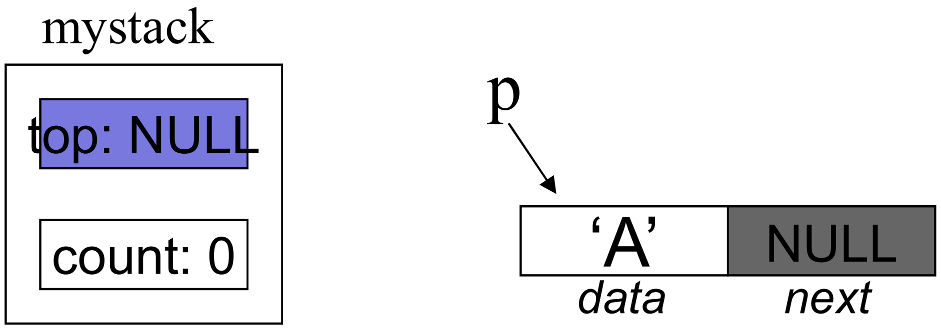
```
void push (char c, stackType *stk) {
      stackElementType *p;

      if ((p = malloc(sizeof(stackElementType))) == NULL) {
            fprintf(stderr,"push failed to malloc\n");
            exit(EXIT_FAILURE);
      }
```

push('A',&mystack)

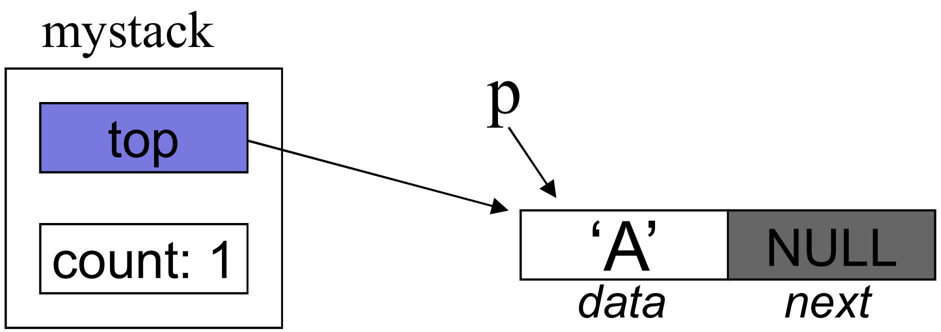


A stack

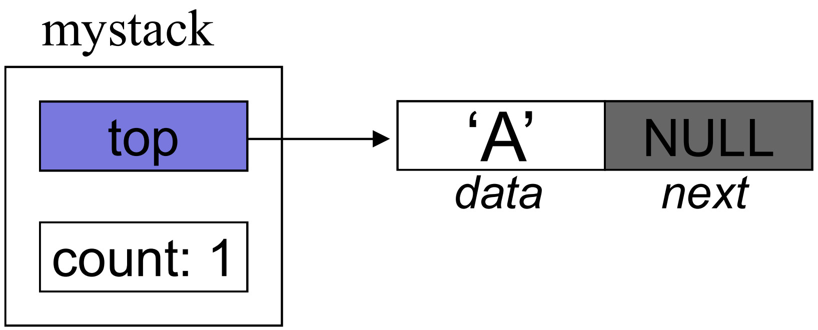
```
p->data = c;
p->next = stk->top;
```




```
stk->top = p;
stk->count++;
```

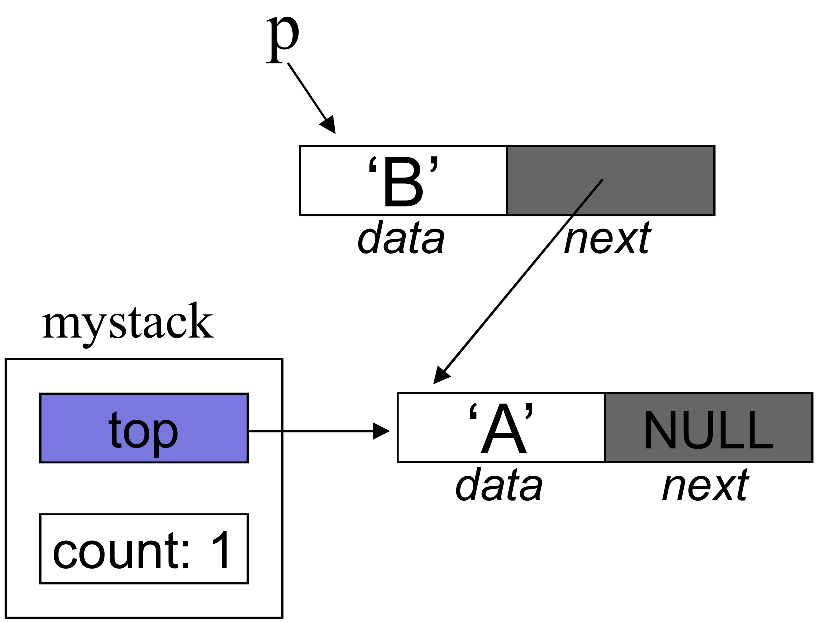


A stack

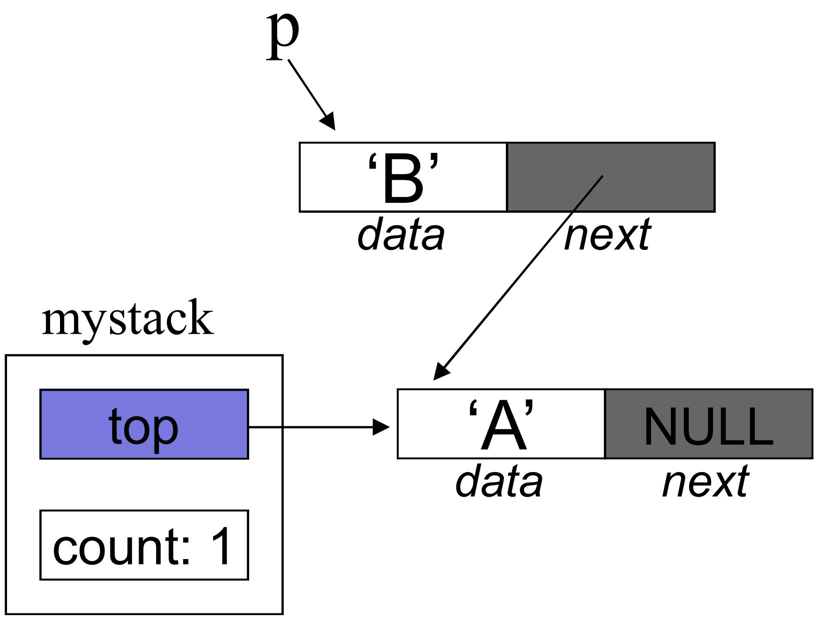


push('B',&mystack)

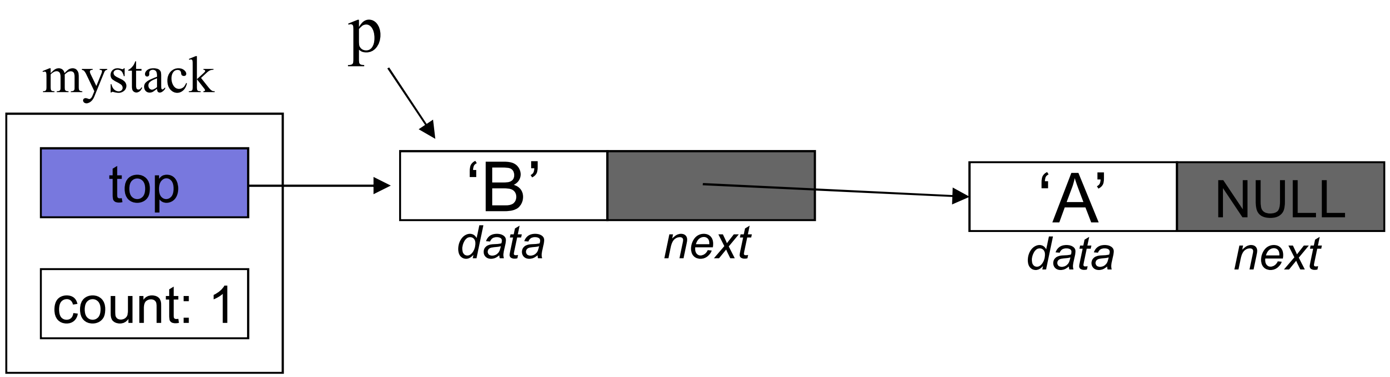
```
p = malloc(...etc..)
p->data = c;
p->next = stk->top;
```



A stack




```
stk->top = p;
stk->count++;
```



A linked list

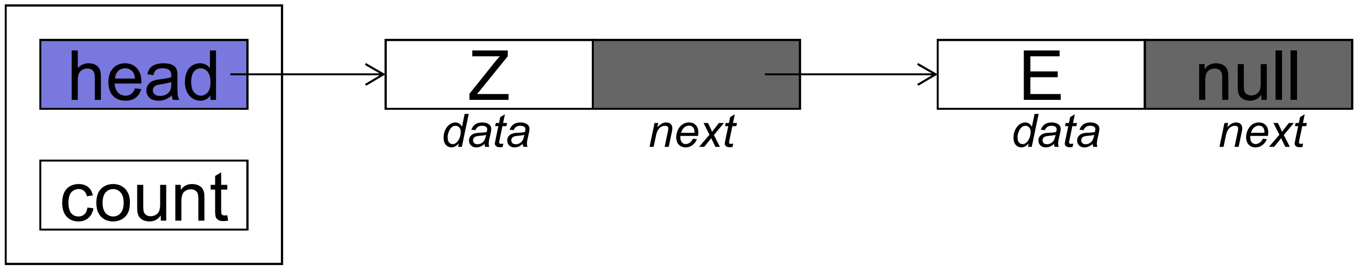
```
typedef struct list_element
{
   char data;
   struct list_element *next;
}  listElementType;


typedef struct list
{
  int count;
  listElementType *head;
} listType;

listType *mylist = NULL;
listElementType *curr, *prev
```

Simple linked list operations

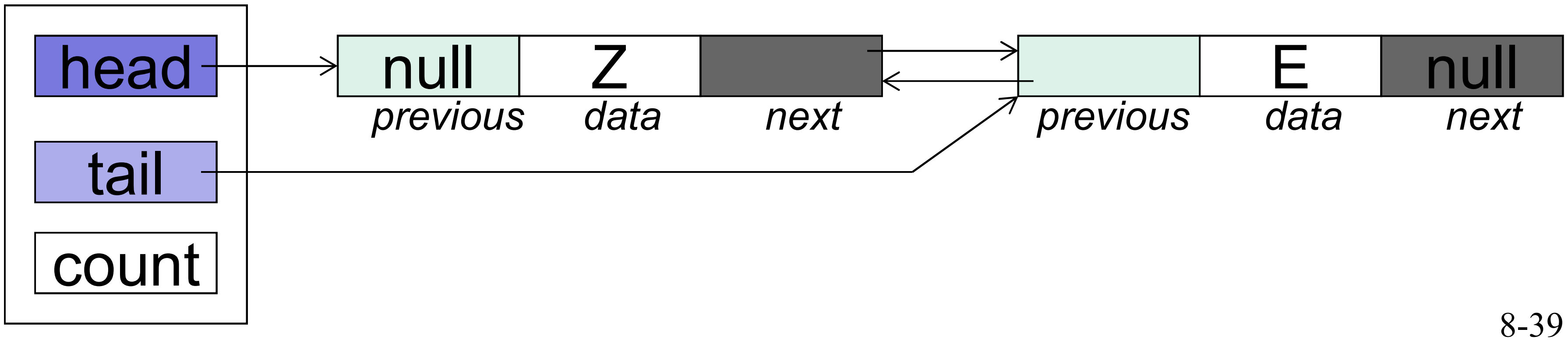
- A set of common operations for a list are (i.e. uses):
  - ❑ **Initialise** the list to some known safe state
  - ❑ **Add** elements to the list
  - ❑ **Delete** elements from the list
  - ❑ **Find** an element in the list
  - ❑ **Clean-up** the list → free memory, perform any other housekeeping.

How Linked Lists Work (1)

- Singly Linked List

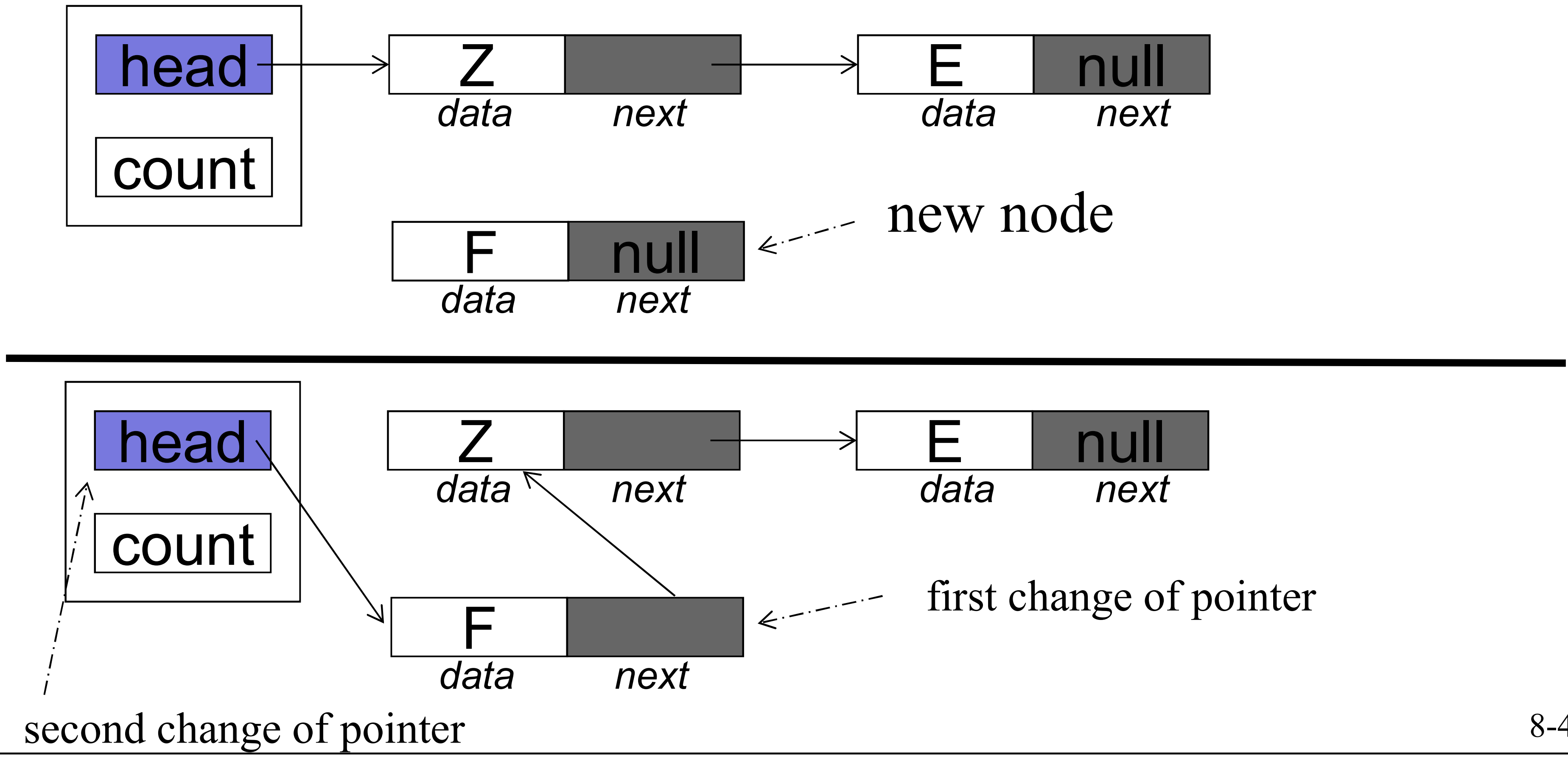


- Doubly Linked List



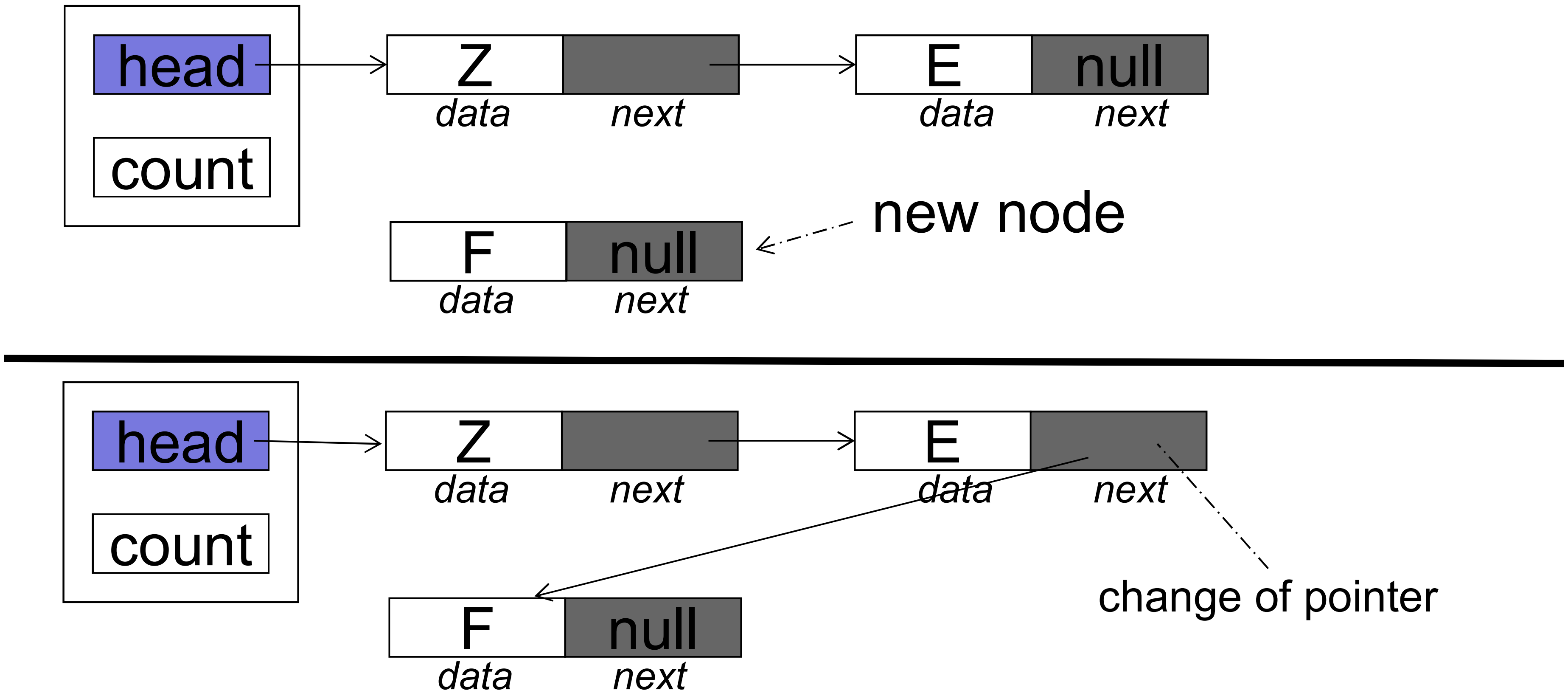
How Linked Lists Work (2)

- Inserting to the head (singly linked list)



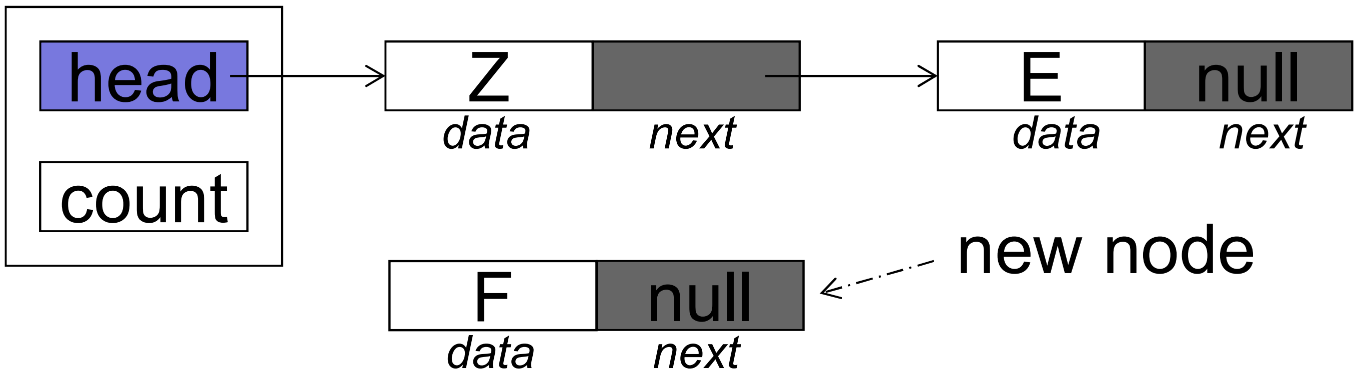
How Linked Lists Work (3)

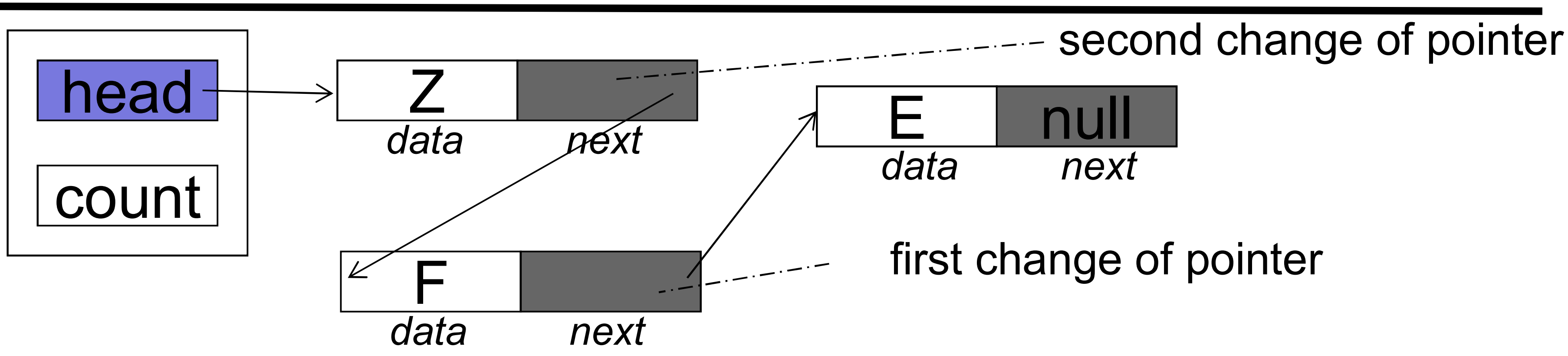
- Inserting to the tail (singly linked list)



How Linked Lists Work (4)

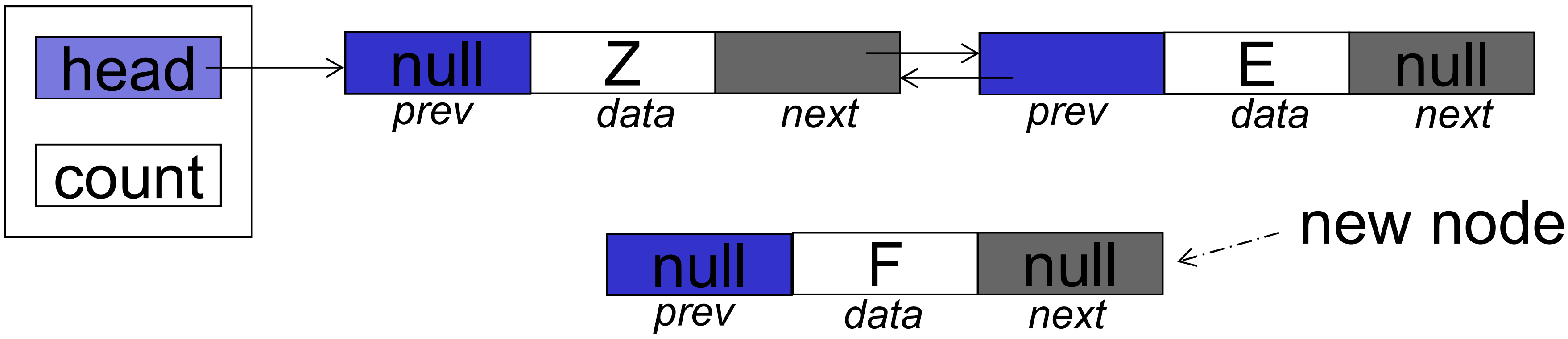
- Inserting to the middle (singly linked list, insert in-order)

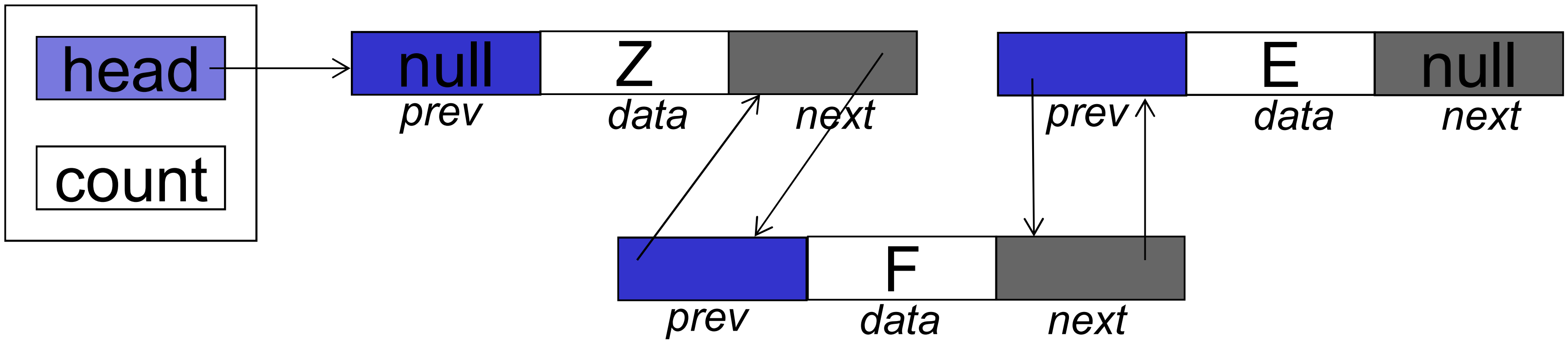




How Linked Lists Work (5)

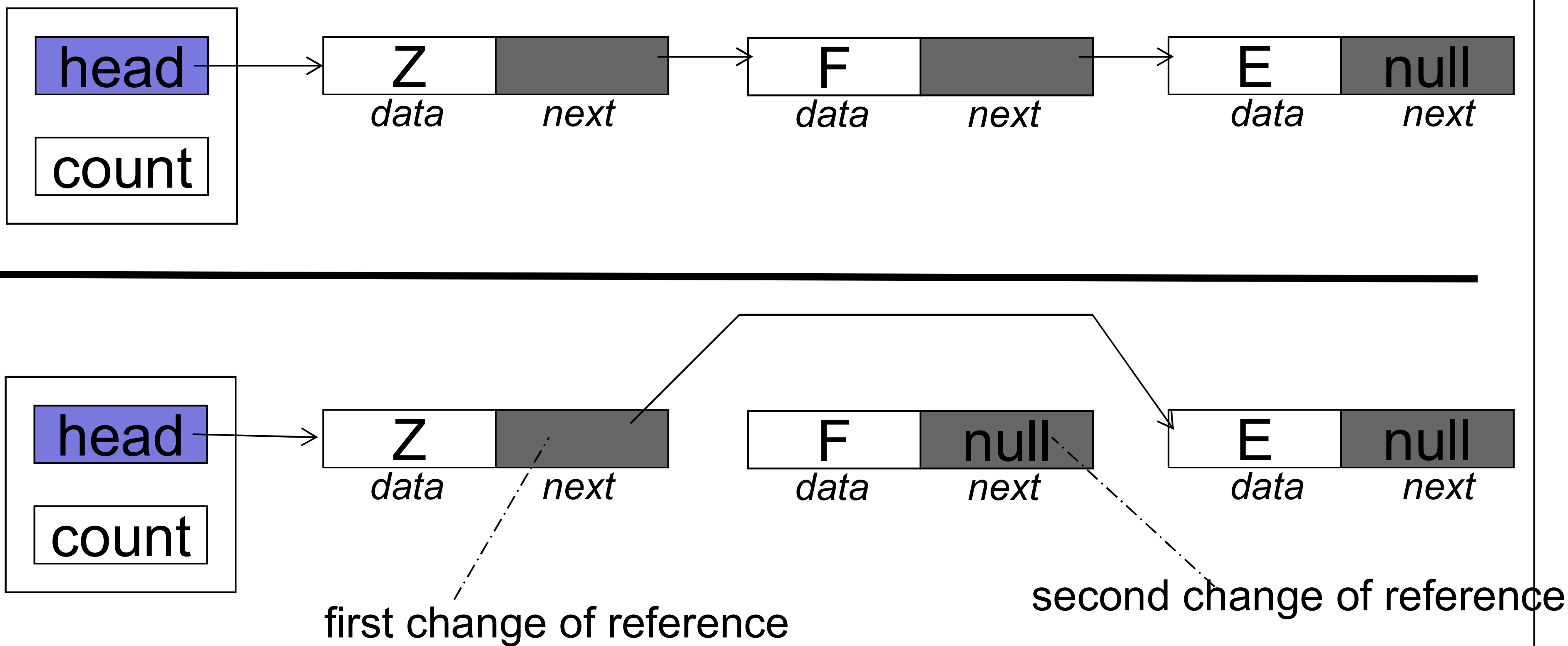
- Inserting to the middle (doubly linked list, insert in-order)





How Linked Lists Work (6)

- Deleting a node (singly linked list)



Traversing a list – iterative solution

- Assume the following function call

```
printList(head);
```

one version of printList() could be:

```
void printList(listElementType *list)
{
      while ( list != NULL )
      {
         printf("%c\n", list->data);
         list = list->next;
      }
}
```

IntList: intlist.h – using a linked list

```
/* IntList
 * -- simple unordered linked list implementation using dynamic memory
 */

… as per previous versions …

typedef struct intlistnode * IntListNodePtr;
typedef struct intlistnode
{  int num;
   IntListNodePtr next;
} IntListNode;

typedef struct
{  IntListNodePtr head;
   unsigned size;
} IntList;

… as per previous version …
```

IntList - intlist.c using a linked list

```
/*
 * IntList
 * -- simple unordered linked list implementation using dynamic memory
 */

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

#include "intlist-linked-list.h"

int MakeList(IntList* pil,int size)
{
   pil->head = NULL;
   pil->size = 0;

   return SUCCESS;
}
```

intlist.c (cont'd)

```
void FreeList(IntList* pil)
{
   IntListNodePtr current, next;

   current = pil->head;

   while (current != NULL)
   {
      next = current->next;
      free(current);
      current = next;
   }

   pil->head = NULL;
   pil->size = 0;
}
```

intlist.c (cont'd)

```
int AddList(IntList* pil,int num)
{
   IntListNodePtr newNode;

   if ((newNode = malloc(sizeof *newNode)) == NULL)
   {
      return FAILURE;
   }

   newNode->num = num;
   newNode->next = pil->head;
   pil->head = newNode;
   pil->size += 1;

   return SUCCESS;

}
```

intlist.c (cont'd)

```
void DisplayList(IntList* pil)
{
   IntListNodePtr current;

   current = pil->head;
   while (current != NULL)
   {
      printf("%d\n", current->num);
      current = current->next;
   }
}

unsigned SizeList(IntList* pil)
{
   return pil->size;
}
```

Function Pointers

- Function pointers are a special kind of pointer – rather than pointing to a general chunk of memory, these pointers **point to functions**.

- The **name of a function is itself a pointer**, just like an array name is.

- A function pointer points to **the physical location of a function in memory**.

Function Pointers - Syntax

- Function pointers take the following general form:

```
returntype (*fnPointerName)(parameter list…);
```

- For example:

```
int (*cmp)(person*, person*);
```

- Or

```
int (*cmp)(person* p1, person* p2);
```

- In both cases we have a pointer variable whose name is `cmp` that points to a function that takes two `person *` parameters and returns an `int`.
- `typedef int (*cmp)(person*, person*);`

Function Pointers - Syntax

- We can easily use typedef to make a new function pointer type:

```
typedef int (*cmp)(person*, person*);
```

- We now have type *cmp*, a pointer to a function. We could define a pointer variable *fpoint* of this type:

```
cmp fpoint;
```

Function Pointers Use Cases

- What are function pointers used for?
  - Function pointers come in handy as a way to change **which function gets called** by another function.
- For example, assume that we have a sort function for an array of students.
- What sorting algorithm should we use? (quick sort, bubble sort etc)
  - Could pass an argument saying what algorithm to use to a function that then calls the appropriate algorithm…
  - … Not very re-usable – what if we want to use a new sort algorithm with an existing ADT interface?

Function Pointers Use Cases

- Pass a pointer that points to our sort function to the existing ADT interface. It can call our sort function via the pointer.

- Such *callback functions* allow an existing algorithm to be customised by another programmer (e.g. API user)

Function pointer example

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

   /* compareFn is a new type: a pointer to a function.
    * The function must have an int return type and take two
    * int parameters
    */
typedef int (*compareFn)(int,int);
```

Function pointer example

```
/* Function that returns 1 if a < b, else returns 0 */
int lessThan(int a, int b)
{
   if (a < b)
        return 1;
   return 0;
}

/* Function that returns 1 if a > b, else returns 0 */
int greaterThan(int a, int b)
{
   if (a > b)
        return 1;
   return 0;
}
```

Function pointer example

```
int main(int argc, char *argv[])
{
   int choice, result;
        /* fpoint is of type compareFn - i.e. a pointer to a
        function */
   compareFn fpoint;

   printf("Enter 1 to use function lessThan, 2 for greaterThan: ");
   scanf("%d",&choice);
```

Function pointer example

```
   switch (choice)
   {
        case 1: fpoint = lessThan;
                break;
        case 2:  fpoint = greaterThan;
                break;
        default:
                return EXIT_FAILURE;
   }

   result = fpoint(9,5);
   printf("result was %d\n",result);
   return EXIT_SUCCESS;
}
```

Function pointer example

- Remember: function pointers are just a variable
- We can also pass function pointers as arguments

RMIT UNIVERSITY Computer Science and Information Technology

Another Function pointer example

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

typedef void (*printFn)(int,int);

void print1(int x, int y)
{
   printf("this is print1 with x %d and y %d\n",x,y);
}

void print2(int x, int y)
{
   printf("this is print2 with x %d and y %d\n",x,y);
}
```

Function pointer example (cont).

```
void doprint(printFn myFunct)
{
   int a = 2, b = 3;
   myFunct(a,b);
}

int main(int argc, char *argv[])
{
    doprint(print1);
    doprint(print2);
    printf("Bye!\n");
    return 0;
}
```