COMP9024: Data Structures and Algorithms

Week 2: Dynamic Data Structures

Contents

- Dynamic Memory Management
- Singly Linked Lists
- Doubly Linked Lists

Storage Classes in C

- Automatic variables *auto*
- Register variables *register*
- External variables *extern*
- Static variables *static*

Automatic Variables

Any variable defined in a function

- Also called local variable
- Its lifetime is the execution period of the function
- Its scope is within the function
- The keyword *auto* can be used to define a local automatic variable. However, it's not required.

```
void DemoFunction(void);
  {
    auto float Local_variable=0;
    …
  }

int main(void) {
   Automatic_Variable = 10;
   DemoFunction();
   …
   return 0;
 }
```

Register Variables

- A register variable is stored in a register by the compiler
  - Specified by the key word *register*

```
register int Global_Variable=1;

int main(void) {
  register int Local_Variable = 10;
  …;
  return 0;
}
```

External Variables

- An external variable is a variable defined outside any function block, also called global variable.
  - Its lifetime is the entire program.
  - When a function in one file references an external variable in another file, the key word *extern* must be used.

File1:

```
int Global_Variable;
void DemoFunction(void);
 int main(void) {
   Global_Variable = 10;
   DemoFunction();
   return 0;
 }
```

File2:

```
extern int Global_Variable;
void DemoFunction(void) {
   ++Global_Variable;
 }
```

Static Variables

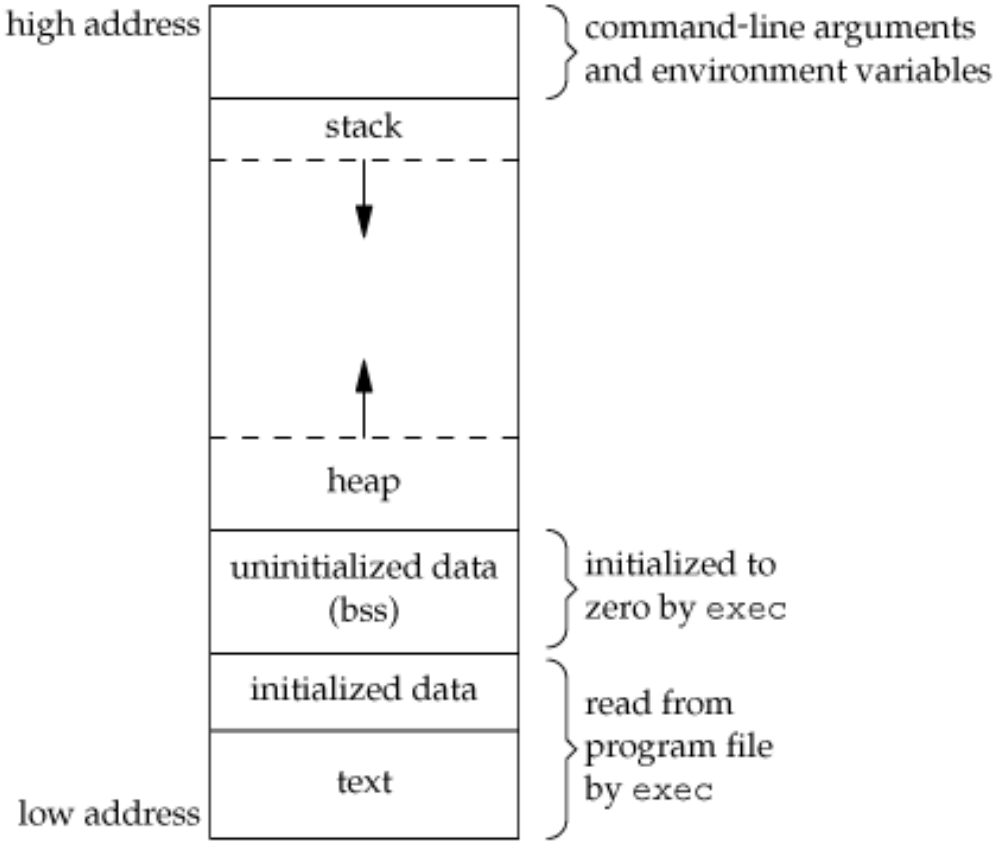
- A static variable is a variable that is allocated statically, meaning that its lifetime is the entire run of the program.
- A static variable is initialized only once.

```
#include <stdio.h>

void func() {
        static int x = 0;
        // x is initialized only once across five calls of func()
        x++;
        printf("%d\n", x); // outputs the value of x
}

int main() {
        func(); // prints 1
        func(); // prints 2
        func(); // prints 3
        func(); // prints 4
        func(); // prints 5
        return 0;
}
```

Memory Layout of A C Program (1/4)

- Text (code) segment
  - Stores the machine instructions that the CPU executes
- Initialized data segment
  - containing global variables and static local variables that are specifically initialized in the program. For example, maxcount and i are in this segment:

```
int  maxcount = 99;
…
int main()
 {  static int i=0;
    …
 }
```



Memory Layout of A C Program (2/4)

- Uninitialized data segment
  - Called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol“.
  - bss contains all the global variables and static local variables that are not initialised. Data in this segment is initialized to 0 or null pointers. The C declaration

```
long  sum[1000];
```

appearing outside any function causes this global variable to be stored in the uninitialized data segment.

Memory Layout of A C Program (3/4)

- When to initialize uninitialized data segment bss?
  - Typically only the length of the bss section, but no data, is stored in the object file. The program loader of the operating system allocates memory for the bss section when it loads the program.
  - In embedded software, the bss segment is mapped into memory that is initialized to zero by the C run-time system before main() is entered.

Memory Layout of A C Program (4/4)

- Stack
  - ➢ where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so that the variables of one instance of the function do not interfere with the variables of another instance.
- Heap
  - ➢ where dynamic memory allocation takes place.

Example

```
int numbers[] = { 40, 20, 30 };

void insertionSort(int array[], int n) {
  int i, j;
  for (i = 1; i < n; i++) {
      int element = array[i];
      while (j >= 0 && array[j] > element) {
         array[j+1] = array[j];
         j--;
       }
      array[j+1] = element;
    }
 }

int main(void) {
  insertionSort(numbers, 3);
  return 0;
}
```

Which memory section are the following objects located in?

1. insertionSort()
2. numbers[0]
3. i
4. element

---

1. Code segment
2. Initialized data segment
3. Stack segment
4. Stack segment

Dynamic Memory Allocation (1/3)

So far, we have considered static memory allocation

- all objects completely defined at compile-time
- sizes of all objects are known to compiler

Examples:

```
int   x;
char *cp;

typedef struct {float x; float y;} Point;
Point p;
char  s[20];
```

Dynamic Memory Allocation (2/3)

In many applications, fixed-size data is ok.

In many other applications, we need flexibility.

Examples:

```
char name[MAXNAME];
char item[MAXITEMS];
char dictionary[MAXWORDS][MAXWORDLENGTH];
```

With fixed-size data, we need to guess sizes ("large enough").

Dynamic Memory Allocation (3/3)

Fixed-size memory allocation:

- allocate as much space as we might ever possibly need

Dynamic memory allocation:

- allocate as much space as we actually need
- determine size based on inputs

But how to do dynamic memory allocation in C?

The malloc() Function (1/3)

malloc() function interface

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

What the function does:

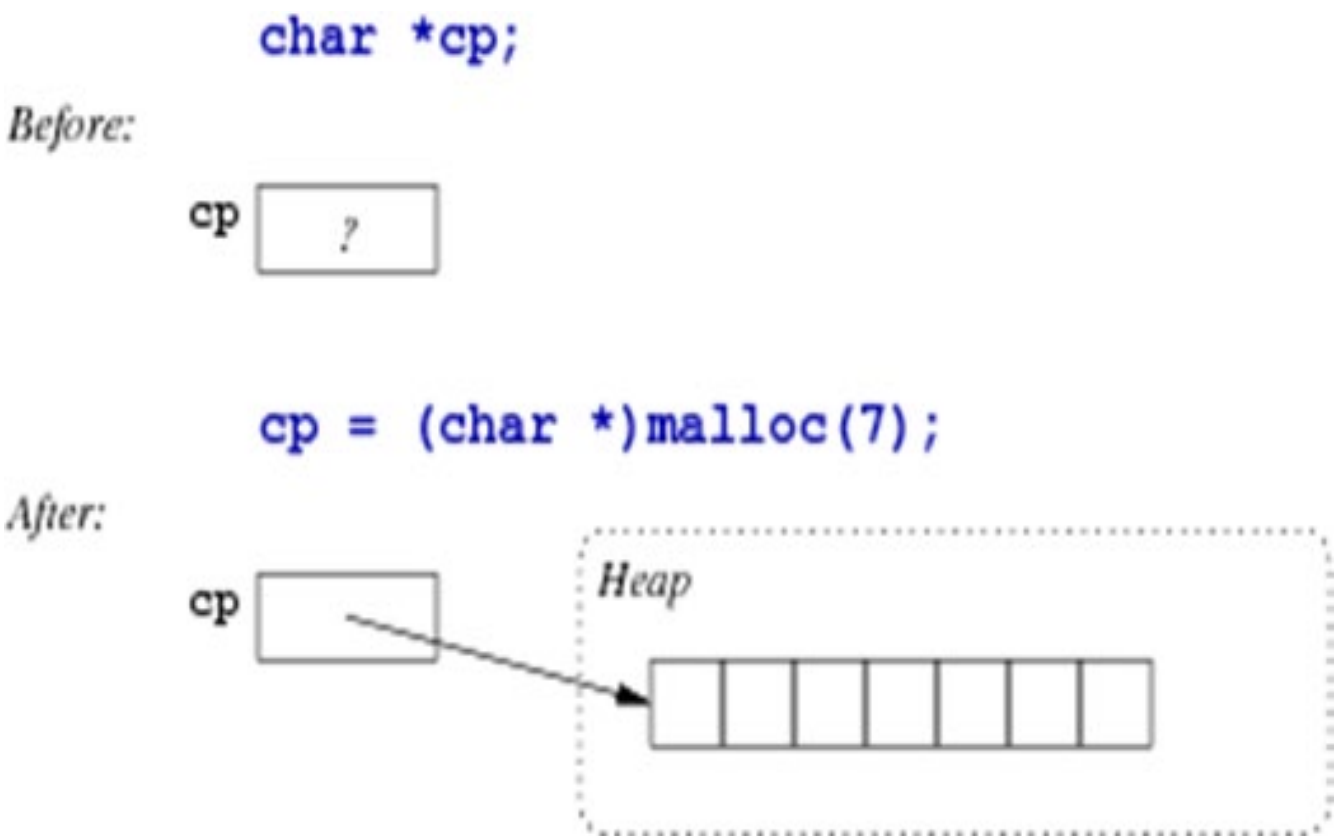
- attempts to reserve a block of n bytes in the heap
- returns the address of the start of this block
- if insufficient space left in the heap, returns NULL

Note: size_t is essentially an unsigned int

- but has specialised interpretation of applying to memory sizes measured in bytes

The malloc() Function (2/3)

Example use of malloc:



The malloc() Function (3/3)

Things to note about void *malloc(size_t):

- it is defined as part of stdlib.h
- its parameter is a size in units of bytes
- its return value is a generic pointer (void *)
- the return value must always be checked (may be NULL)

Required size is determined by #Elements * sizeof(ElementType)

Example

Create a dynamic m×n-matrix of floating point numbers, given m and n.

How many bytes need to be reserved?

Matrix:

```
float *matrix = malloc(m * n * sizeof(float));
assert(matrix != NULL);
```

4mn bytes allocated

void assert(int expression) is a C built-in macro. If expression evaluates to TRUE, assert() does nothing. If expression evaluates to FALSE, assert() displays an error message on stderr (standard error stream to display error messages and diagnostics) and aborts program execution.

Example

Create space for 1,000 speeding tickets (cf. Lecture Week 1)

How many bytes need to be reserved?

Speeding tickets:

```
typedef struct {
        int day, month, year;
} DateT;
typedef struct {
        int hour, minute;
} TimeT;
typedef struct {
        char  plate[7];
        DateT d;
        TimeT t;
} TicketT;

TicketT *tickets = malloc(1000 * sizeof(TicketT));
assert(tickets != NULL);
```

28,000 bytes allocated

More about malloc() (1/2)

malloc() returns a pointer to a data object of some kind.

Things to note about objects allocated by malloc():

- they exist until explicitly removed (program-controlled lifetime)
- they are accessible while some variable references them
- if no active variable references an object, it is garbage

The function free() releases objects allocated by malloc()

More about malloc() (2/2)

Usage of malloc() should always be guarded:

```
int *vector, length, i;
…
vector = malloc(length*sizeof(int));

assert(vector != NULL);

for (i = 0; i < length; i++) {
        … vector[i] …
}
```

Alternatively:

```
int *vector length, i;
…
vector = malloc(length*sizeof(int));

if (vector == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(1);
}

for (i = 0; i < length; i++) {
        … vector[i] …
}
```

- fprintf(stderr, …) outputs text to a stream called stderr (the screen, by default)
- exit(v) terminates the program with return value v

Memory Management (1/5)

void free(void *ptr)

- releases a block of memory allocated by malloc()
- *ptr is a dynamically allocated object
- if *ptr was not malloc()'d, chaos will follow

Things to note:

- the contents of the memory block are not changed
- all pointers to the block still exist, but are not valid
- the memory may be re-used as soon as it is free()'d

Memory Management (2/5)

Warning! Warning! Warning! Warning!

Careless use of malloc() / free() / pointers

- can mess up the data in the heap
- so that later malloc() or free() cause run-time errors
- possibly well after the original error occurred

Such errors are very difficult to track down and debug.

Must be very careful with your use of malloc() / free() / pointers.

Memory Management (3/5)

If an uninitialized or otherwise invalid pointer is used, or an array is accessed with a negative or out-of-bounds index, one of a number of things might happen:

- program aborts immediately with a "segmentation fault"
- a mysterious failure much later in the execution of the program
- incorrect results, but no obvious failure
- correct results, but maybe not always, and maybe not when executed on another day, or another machine

The first is the most desirable, but cannot be relied on.

Memory Management (4/5)

Given a pointer variable:

- you can check whether its value is NULL
- you can (maybe) check that it is an address
- you cannot check whether it is a valid address

Memory Management (5/5)

Typical usage pattern for dynamically allocated objects:

```
Type *ptr = malloc(sizeof(Type));
assert(ptr != NULL);

free(ptr);
```

```
int nelems = NumberOfElements;
ElemType *arr = malloc(nelems*sizeof(ElemType));
assert(arr != NULL);

free(arr);
```

Memory Leaks

Well-behaved programs do the following:

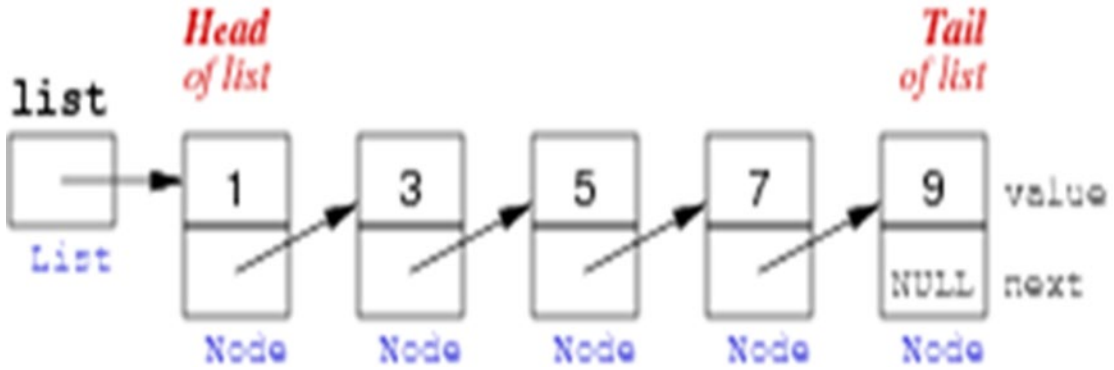
- allocate a new object via malloc()
- use the object for as long as needed
- free() the object when no longer needed

A program which does not free() an object before the last reference to it is lost contains a memory leak.

Such programs may eventually exhaust available heap space.

Singly Linked Lists in C (1/18)

A singly linked list is a chain of nodes, where each node contains a pointer to the next node



To represent a chained (linked) list of nodes:

- we need a pointer to the first node and possibly a tail pointer to the last node
- each node contains a pointer to the next node
- the next pointer in the last node is NULL

Singly Linked Lists in C (2/18)

A node has two components:

- data, which can be a single value (e.g. int), or a collection of values, depending on specific application
- a pointer to the next node

An example node in C:

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

Singly Linked Lists in C (3/18)

Typical operations on singly linked lists:

- Create a new linked list
- Create a new node
- Delete a node
- Insert a node
- Find a node containing particular data

Singly Linked Lists in C (4/18)

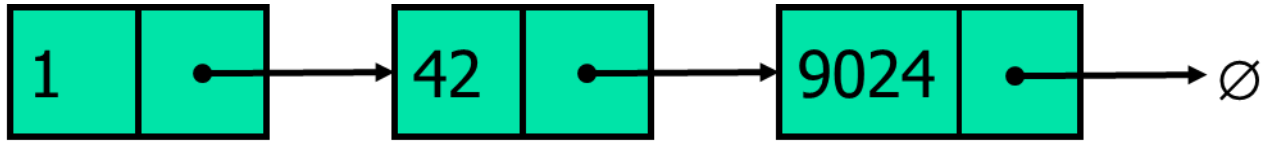
Create a new node:

```
NodeT * createNode(int v) {
   NodeT *new = malloc(sizeof(NodeT));
   assert(new != NULL);
   new->data = v;
   new->next = NULL;
   return new;
}
```

Singly Linked Lists in C (5/18)

Create a new singly linked list of three nodes:

```
NodeT *list = createNode(1);
list->next  = createNode(42);
list->next->next = createNode(9024);
```



Singly Linked Lists in C (6/18)

Add a node at the end:

1. Create a new node
2. Find the last node
3. Make the last node point to new node

If we maintain a tail pointer in the list, we can locate the last node directly.

```
NodeT* addNode(NodeT *head, int v){ // head points to the first node
    NodeT *temp, *p; // declare two node pointers temp and p
    temp = createNode(); //create a new node
    temp->data = v; // add element's value to data part of node
    if(head == NULL){
        head = temp;     //when linked list is empty
      }
    else {
        p  = head; //assign head to p
        while (p->next != NULL){
            p = p->next; //traverse the list until p is the last node
        }
        p->next = temp; // Make the previous last node point to the new node
    }
    return head;
}
```

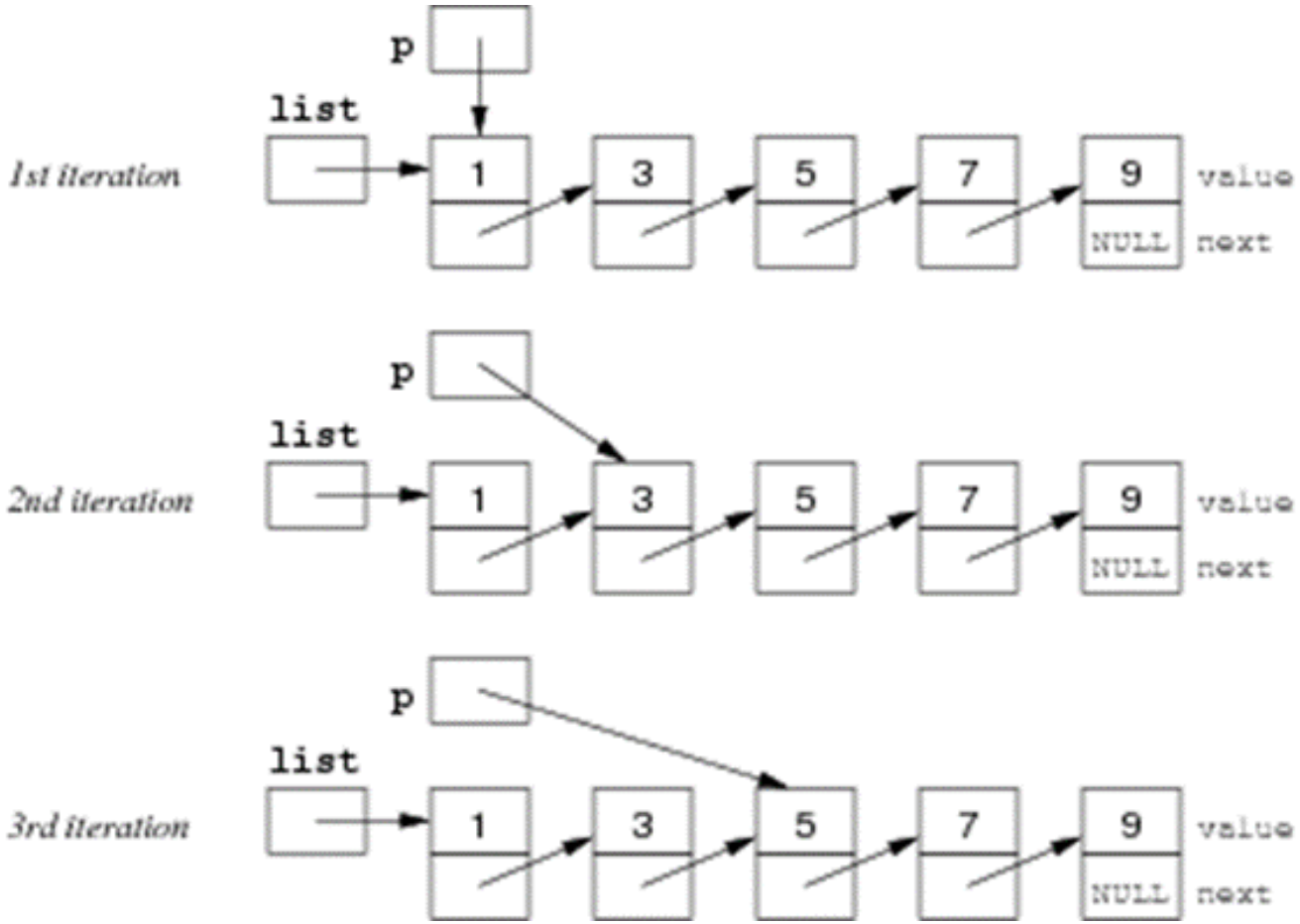
Singly Linked Lists in C (7/18)

Iterate over a singly linked list:

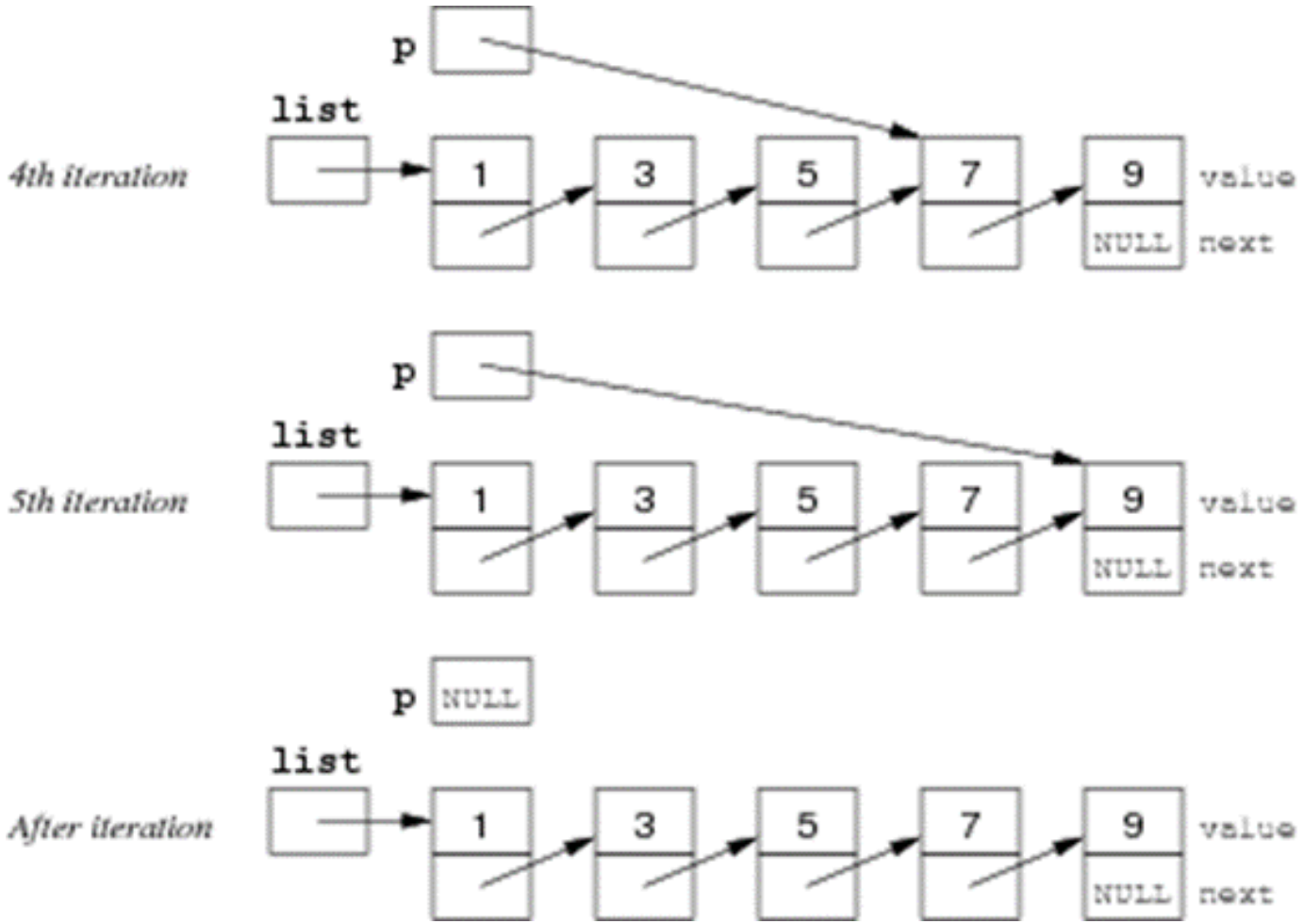
- set p to point to first node (head)
- examine node pointed to by p
- change p to point to next node
- stop when p reaches end of list (NULL)

```
NodeT *list;
NodeT *p;

p = list;
while (p != NULL) {
        printf("%d", p->data);
        p = p->next;
}

for (p = list; p != NULL; p = p->next) {
        printf("%d", p->data);
}
```

Singly Linked Lists in C (8/18)



Singly Linked Lists in C (9/18)



Singly Linked Lists in C (10/18)

Find an element in a singly linked list:

```
int findElement(NodeT *list, int d) {
  NodeT *p;
  for (p = list; p != NULL; p = p->next)
    if (p->data == d)
      return 1;
  return 0;
}
```

Print all elements:

```
showElements(NodeT *list) {
  NodeT *p;
  for (p = list; p != NULL; p = p->next)
    printf("%6d", p->data);
}
```

Singly Linked Lists in C (11/18)

Linked list nodes are typically located in the heap

- because nodes are dynamically created

Variables containing pointers to list nodes

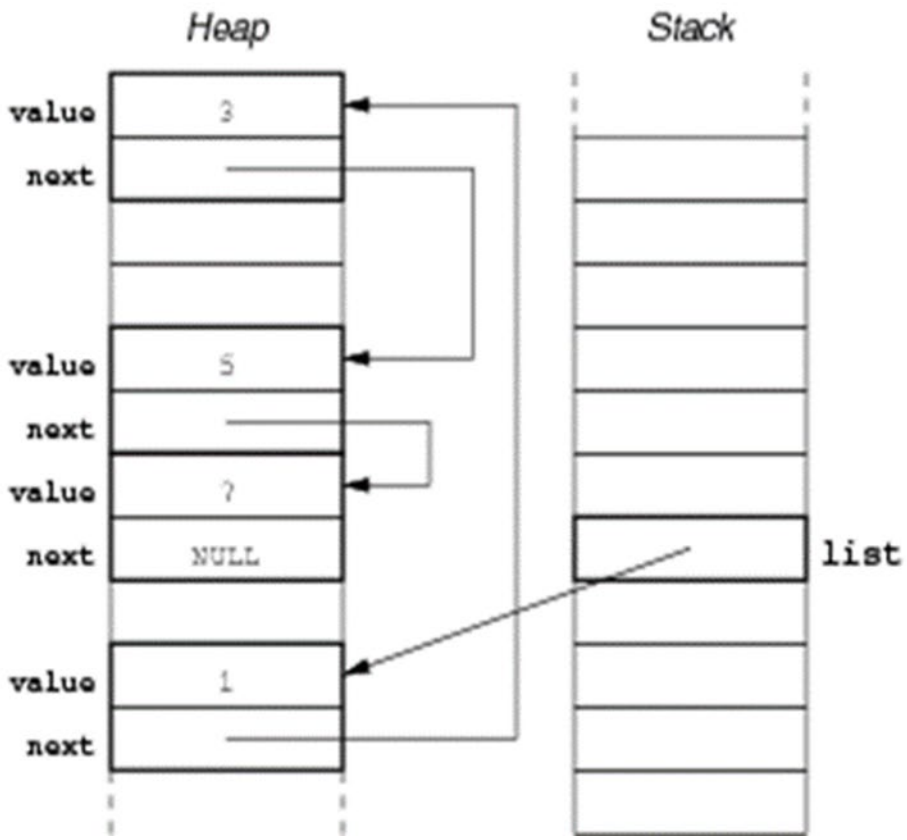
- are likely to be local variables (in the stack)

Pointers to the start of lists are often

- passed as parameters to function
- returned as function results

Nodes may be created in any order in the heap.

Depends on `malloc()`



Singly Linked Lists in C (12/18)

Linked lists are more flexible than arrays:

- values do not have to be adjacent in memory
- values can be rearranged simply by altering pointers
- the number of values can change dynamically
- values can be added or removed in any order

Disadvantages:

- it is not difficult to get pointer manipulations wrong
- each value also requires storage for next pointer
- creating a node needs OS support, and thus is slow.

Singly Linked Lists in C (13/18)

What does this code do?

```
1  NodeT *p = list;
2  while (p != NULL) {
3     printf("%6d", p->data);
4     if (p->next != NULL)
5        p = p->next->next;
6     else
7        p = NULL;
9  }
```

What is the purpose of the conditional statement in line 4?

_____

Every second list element is printed.

If *p happens to be the last element in the list, then p->next->next does not exist.

The if-statement ensures that we do not attempt to assign an invalid address to p in line 5.

Singly Linked Lists in C (14/18)

Rewrite showElements() as a recursive function.

```
void printElements(NodeT *list) {
   if (list != NULL) {
      printf("%6d", list->data);
      printElements(list->next);
   }
}
```

Singly Linked Lists in C (15/18)

Insert a new element at the beginning:

```
NodeT *insertAtHead(NodeT *list, int d) {
  NodeT *new = createNode(d);
  new->next = list;
  return new;
}
```

Singly Linked Lists in C (16/18)

Delete the first element:

```
NodeT *deleteHead(NodeT *list) {
  assert(list != NULL);
  NodeT *head = list;
  list = list->next;
  free(head);
  return list;
}
```

What would happen if we didn't free the memory pointed to by head?

Singly Linked Lists in C (17/18)

Delete a specific element (recursive version):

```
NodeT *deleteLL(NodeT *list, int d) {
  if (list == NULL) {
    return list;

  } else if (list->data == d) {
    return deleteHead(list);

  } else {
    list->next = deleteLL(list->next, d);
    return list;
  }
}
```

Singly Linked Lists in C (18/18)

Write a C-function to destroy an entire list.

Iterative version:

```
void freeLL(NodeT *list) {
  NodeT *p;

  p = list;
  while (p != NULL) {
    NodeT *temp = p->next;
    free(p);
    p = temp;
  }
}
```

Why do we need the extra variable temp?

Stack ADT Implementation Using Linked List (1/4)

Interface (in stack.h)

```
// provides an opaque view of ADT

typedef struct StackRep *stack;
// set up empty stack
stack newStack();
// remove unwanted stack
void dropStack(stack);
// check whether stack is empty
int StackIsEmpty(stack);
// insert an int on top of stack
void StackPush(stack, int);
// remove int from top of stack
int StackPop(stack);
```

ADT stack defined as a pointer to an unspecified struct

Stack ADT Implementation Using Linked List (2/4)

```
#include <stdlib.h>
#include <assert.h>
#include "stack.h"

typedef struct node {
   int data;
   struct node *next;
} NodeT;

typedef struct StackRep {
   int    height;
   NodeT *top;
} StackRep;
```

```
stack newStack() {
   stack S = malloc(sizeof(StackRep));
   S->height = 0;
   S->top = NULL;
   return S;
}

int StackIsEmpty(stack S) {
   return (S->height == 0);
}
```

Stack ADT Implementation Using Linked List (3/4)

```
void StackPush(stack S, int v) {
   NodeT *new = malloc(sizeof(NodeT));
   assert(new != NULL);
   new->data = v;


   new->next = S->top;
   S->top = new;
   S->height++;
}
```

```
void dropStack(stack S) {
   NodeT *curr = S->top;
   while (curr != NULL) {
      NodeT *temp = curr->next;
      free(curr);
      curr = temp;
   }
   free(S);
}
```

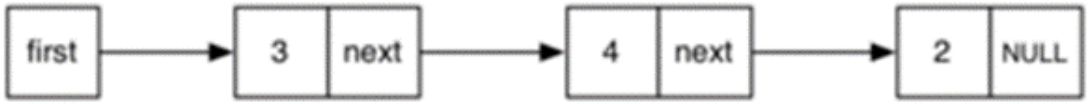
Stack ADT Implementation Using Linked List (4/4)

```
int StackPop(stack S) {
  assert(S->height > 0);
  NodeT *head = S->top;

  S->top = S->top->next;
  S->height--;

  int d = head->data;
  free(head);
  return d;
}
```

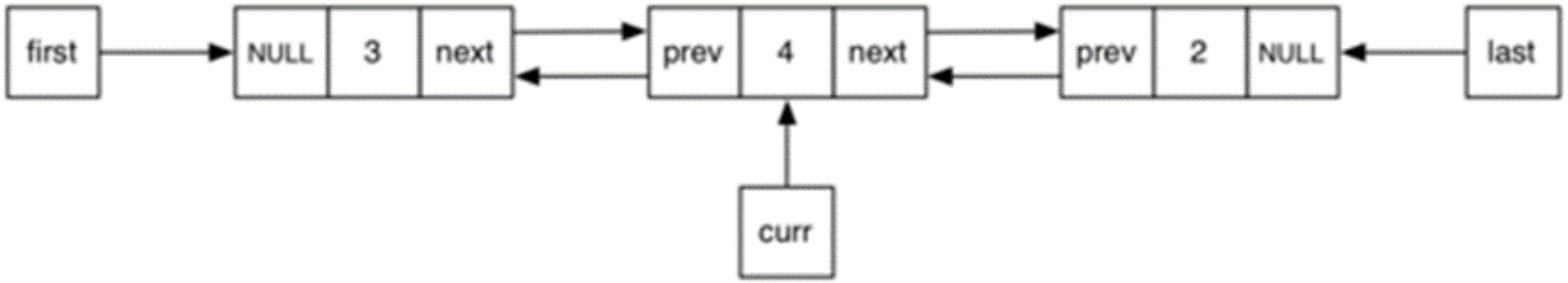
Doubly Linked Lists (1/2)

Doubly-linked lists are a variation on "standard" linked lists where each node has a pointer to the previous node as well as a pointer to the next node.

Singly-linked List



Doubly-linked List



Doubly Linked Lists (1/2)

- The doubly-linked list also has a notion of a "current" node, and the current node can move backwards and forwards along the list.
- The doubly-linked list does insertions either immediately before or immediately after the current node.
- Unlike the singly linked list, deleting the last node does not need to traverse the entire list.

Summary (1/3)

void *malloc(size_t nbytes)

- aim: allocate some memory for a data object
- attempt to allocate a block of memory of size nbytes in the heap
- if successful, returns a pointer to the start of the block
- if insufficient space in heap, returns NULL

Things to note:

- the location of the memory block within heap is random
- the initial contents of the memory block are random

Summary (2/3)

void free(void *ptr)

- releases a block of memory allocated by malloc()
- *ptr is the start of a dynamically allocated object
- if *ptr was not malloc()'d, chaos will ensue

Things to note:

- the contents of the memory block are not changed
- all pointers to the block still exist, but are not valid
- the memory may be re-used as soon as it is free()'d

Summary (3/3)

- Singly linked lists
- Doubly linked lists
- Suggested reading:
  - Moffat, Ch.10.1-10.2
  - Sedgewick, Ch.3.3-3.5,4.4,4.6