Summer 2013

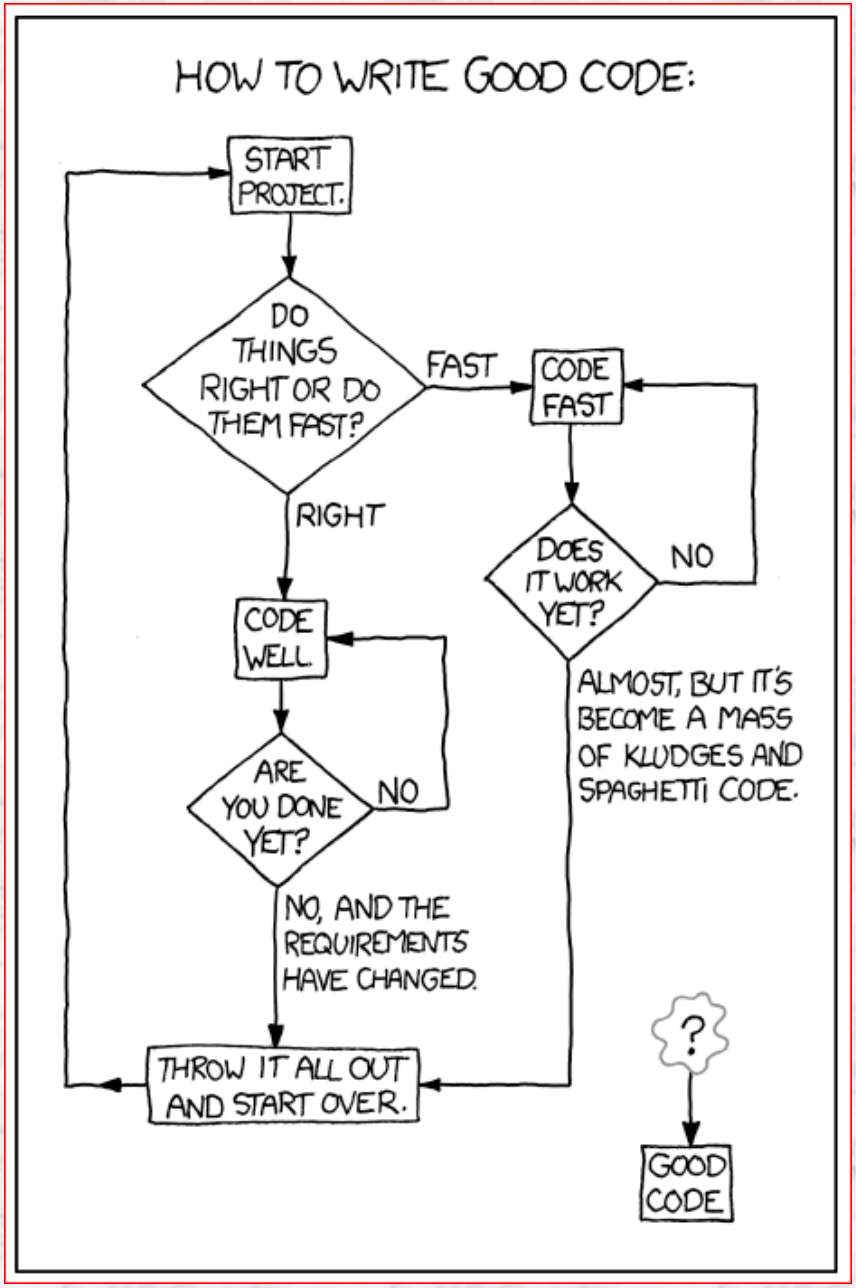
CSE2421 Systems1

Introduction to Low-Level Programming and Computer Organization

Kitty Reeves

MTWR 10:20am-12:25pm

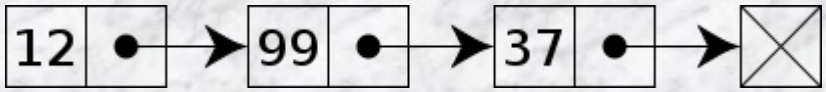
How to write good code?



Linked List Node structure

- A linked list is…
  - a dynamic data structure consisting of a group of nodes which together represent a sequence and whose length can be increased or decreased at run time
- Simply, each node is composed of a data and a reference (in other words, a *link*) to the next node in the sequence
- Allows for efficient insertion or removal of elements from any position in the sequence (vs an array).
- Data items need not be stored contiguously in memory
- Major Disadvantage:
  - does not allow random access to the data or any form of efficient indexing

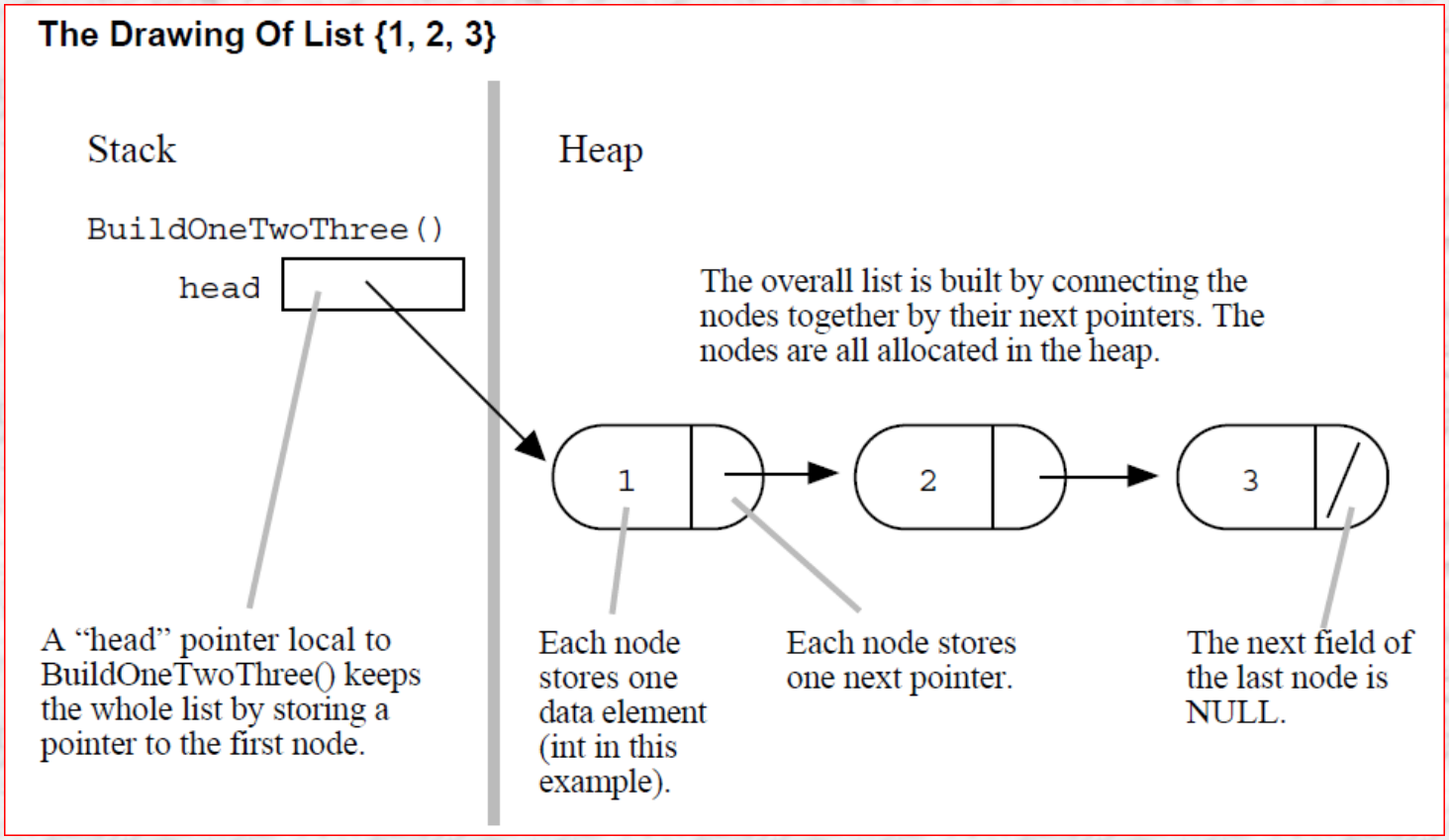
```
/* Node Structure */
struct node {
    int data;
    struct node *next; }
```



A linked list whose nodes contain two fields: an integer value and a link to the next node. The last node is linked to a terminator used to signify the end of the list.

Linked List stack and heap

- Each node is allocated in the heap with a call to malloc(), so the node memory continues to exist until it is explicitly deallocated with a call to free().



Dynamic Memory Functions

- Can be found in the stdlib.h library:
  - To allocate space for an array in memory you use
    - calloc()
  - To allocate a memory block you use
    - malloc()
  - To de-allocate previously allocated memory you use
    - free()
- Each function is used to initialize a pointer with memory from free store (a section of memory available to all programs i.e. the heap)

malloc

- The function malloc() will allocate a block of memory that is size bytes large. If the requested memory can be allocated a pointer is returned to the beginning of the memory block.
  - **Note:** the content of the received block of memory is not initialized.
- **Usage of malloc():**
  - void * malloc (size_t size);
- **Parameters:**
  - Size of the memory block in bytes.
- **Return value:**
  - If the request is successful then a pointer to the memory block is returned.
  - If the function failed to allocate the requested block of memory, a null pointer is returned.
- Example
  - http://www.codingunit.com/c-reference-stdlib-h-function-malloc
- Another example:
  - **#include <stdlib.h>**
  - **int *ptr = malloc(sizeof (int));**
    - set ptr to point to a memory address of size int
  - **int *ptr = malloc(sizeof (*ptr));**
    - is slightly cleaner to write malloc statements by taking the size of the variable pointed to by using the pointer directly
  - **float *ptr = malloc(sizeof (*ptr));**
    - float *ptr;
    - /* hundreds of lines of code */
    - ptr = malloc(sizeof(*ptr));

calloc

- **Usage of calloc():**
  - void * calloc (size_t num, size_t size);
- **Parameters:**
  - Number of elements (array) to allocate and the size of elements.
- **Return value:**
  - Will return a pointer to the memory block. If the request fails, a NULL pointer is returned.
- Example:
  - http://www.codingunit.com/c-reference-stdlib-h-function-calloc
  - note: ptr_data = (int*) calloc (a,sizeof(int));

malloc and calloc examples

```
#include<stdio.h>
int main() {
   int *ptr_one;
   ptr_one = (int *)malloc(sizeof(int));
   if (ptr_one == 0) {
      printf("ERROR: Out of memory\n");
      return 1;
   }
   *ptr_one = 25;
   printf("%d\n", *ptr_one);
   free(ptr_one);
   return 0;
}
```

```
#include<stdio.h>
typedef struct rec {
   int i; float PI; char A; } RECORD;
int main() {
   RECORD *ptr_one;
   ptr_one = (RECORD *) malloc (sizeof(RECORD));
   (*ptr_one).i = 10;
   (*ptr_one).PI = 3.14;
   (*ptr_one).A = 'a';
   printf("First value: %d\n",(*ptr_one).i);
   printf("Second value: %f\n", (*ptr_one).PI);
   printf("Third value: %c\n", (*ptr_one).A);
   free(ptr_one); return 0; }
```

Difference Between Malloc and Calloc

- The number of arguments. malloc() takes a single argument (memory required in bytes), while calloc() needs two arguments.
- malloc() does not initialize the memory allocated, while calloc() initializes the allocated memory to ZERO.

free

- The free function returns memory to the operating system.
  - **free(ptr);**
  - After freeing a pointer, it is a good idea to reset it to point to 0.

NOTE: When 0 is assigned to a pointer, the pointer becomes a null pointer…in other words, it points to nothing. By doing this, when you do something foolish with the pointer (it happens a lot, even with experienced programmers), you find out immediately instead of later, when you have done considerable damage.

Build {1,2,3} linked list

```
/* Build the list {1, 2, 3} in the heap and store its head pointer in a local stack
variable. Returns the head pointer to the caller. */
struct node* BuildOneTwoThree() {
        struct node* head = NULL;
        struct node* second = NULL;
        struct node* third = NULL;
// allocate 3 nodes in the heap
        head = malloc(sizeof(struct node));
        second = malloc(sizeof(struct node));
        third = malloc(sizeof(struct node));
        head->data = 1; // setup first node
        head->next = second; // note: pointer assignment rule
        second->data = 2; // setup second node
        second->next = third;
        third->data = 3; // setup third link
        third->next = NULL;
// At this point, the linked list referenced by "head“ matches the list in the drawing.
        return head;  }
```

Linked List example

```
#include<stdlib.h>
#include<stdio.h>
struct list_el {
    int val;
    struct list_el * next; };
typedef struct list_el item;
void main() {
    item * curr, * head;
    int i;
    head = NULL;
    for(i=1;i<=10;i++) {
        curr = (item *) malloc(sizeof(item));
        curr->val = i;
        curr->next = head;
        head = curr; }
    curr = head;
    while(curr) {
        printf("%d\n", curr->val);
        curr = curr->next ; }
}
```

- What does this do?

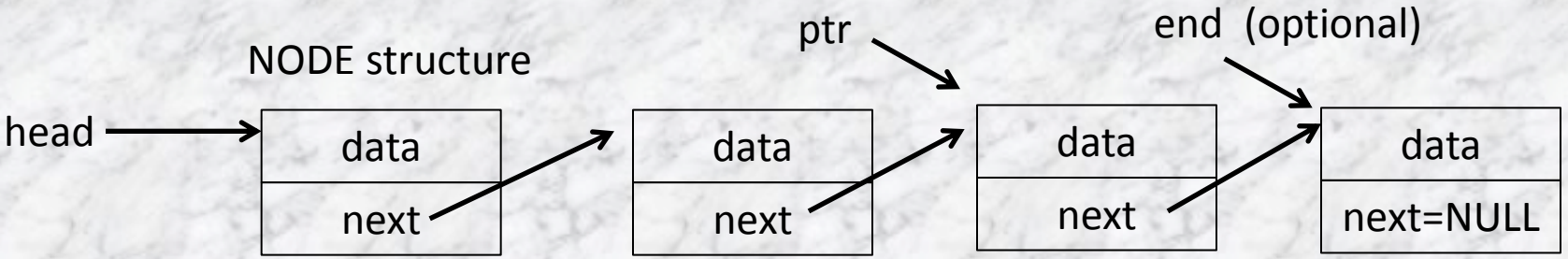
Linked list basics

- The first node is always made accessible through a global 'head' pointer.
  - This pointer is adjusted when first node is deleted.
- Similarly there can be an 'end' pointer that contains the last node in the list.
  - This is also adjusted when last node is deleted.
  - The last node in a list always has a NULL value so you don't *have* to keep track of the end of the list, just check for a NULL pointer.
- Whenever a node is added to linked list, it is always checked if the linked list is empty then add it as the first node.
- You can pass the list by passing the head pointer.

Initialize node

```
/* this initializes a node, allocates memory for the node, and returns   */
/* a pointer to the new node. Must pass it the node details, name and id */
struct node * initnode( char *name, int id )
{
  struct node *ptr;
  ptr = (struct node *) calloc( 1, sizeof(struct node ) );

  if( ptr == NULL ) {                  /* error allocating node?      */
    return (struct node *) NULL;}      /* then return NULL, else      */
  else {                               /* allocated node successfully */
     strcpy( ptr->name, name );        /* fill in name details        */
     ptr->id = id;                     /* copy id details             */
     return ptr;                       /* return pointer to new node  */
  }
}
```

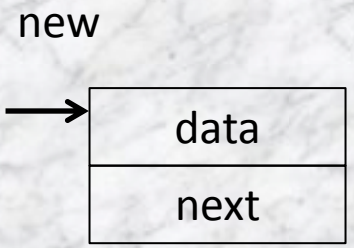
Linked List setup



NEED TO:

Allocate a new node structure with DMA
(dynamic memory allocation)

Add information to data section



Linked List ADD/DELETE node

OPERATION	FRONT	END	MIDDLE
INSERT	new->next = head head = new	ptr->next = new new->next = NULL	new->next = ptr->next ptr-> next= new

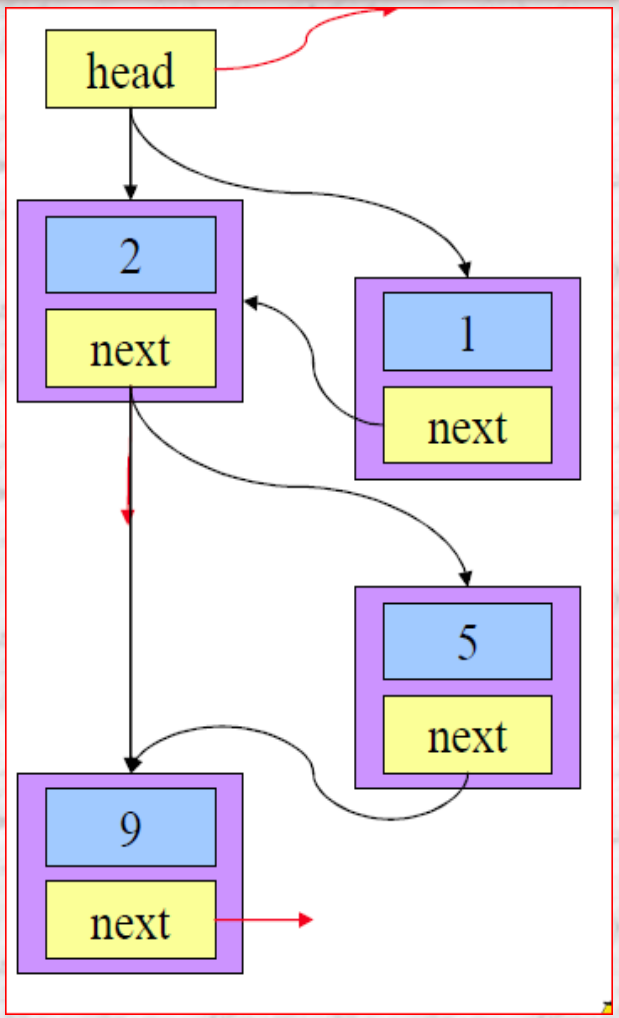
SEARCH

```
found = false;
ptr = head;
while(ptr != NULL)                  //what if nothing in list?
{    if(ptr->data = = val)          // found what searching for
     {    found = true;
          break; }
     else { ptr = ptr->next; }
}                                   // if found still false, didn't find
```

DELETE
(fyi: free ptr)
	head = ptr->next
	// what if only node?
		//if ptr->next=NULL
		prev = ptr
		prev->next = NULL
			prev->next = ptr->next

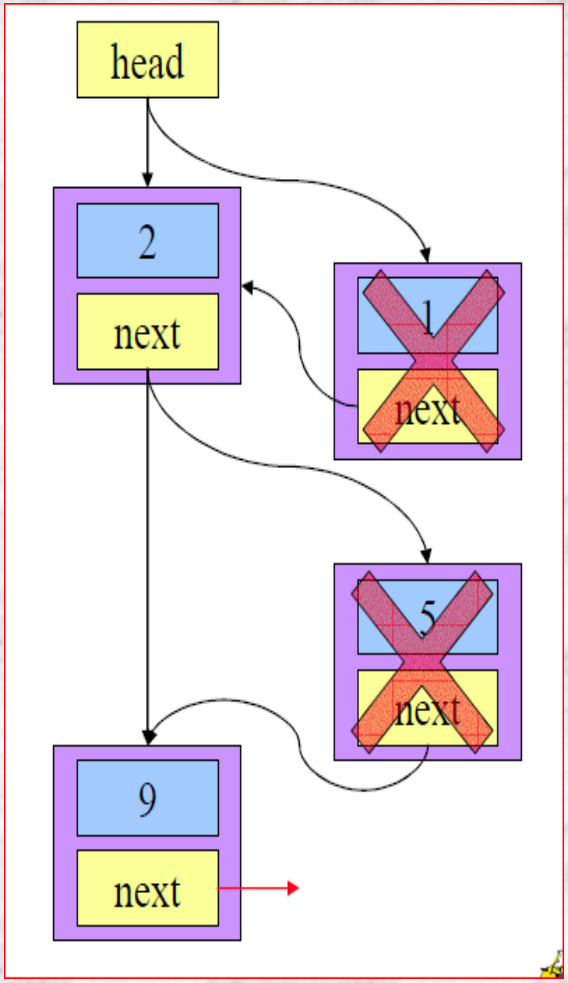
Link list pictorial view - INSERT

- Inserting into a link list has two cases
  - First in the list
  - Not first in the list
- If going at head, modify head reference (only)
- If going elsewhere, need reference to node before insertion point
  - New node.next = cur node.next
  - Cur node.next = ref to new node
  - Must be done in this order!



Link list pictorial view - DELETE

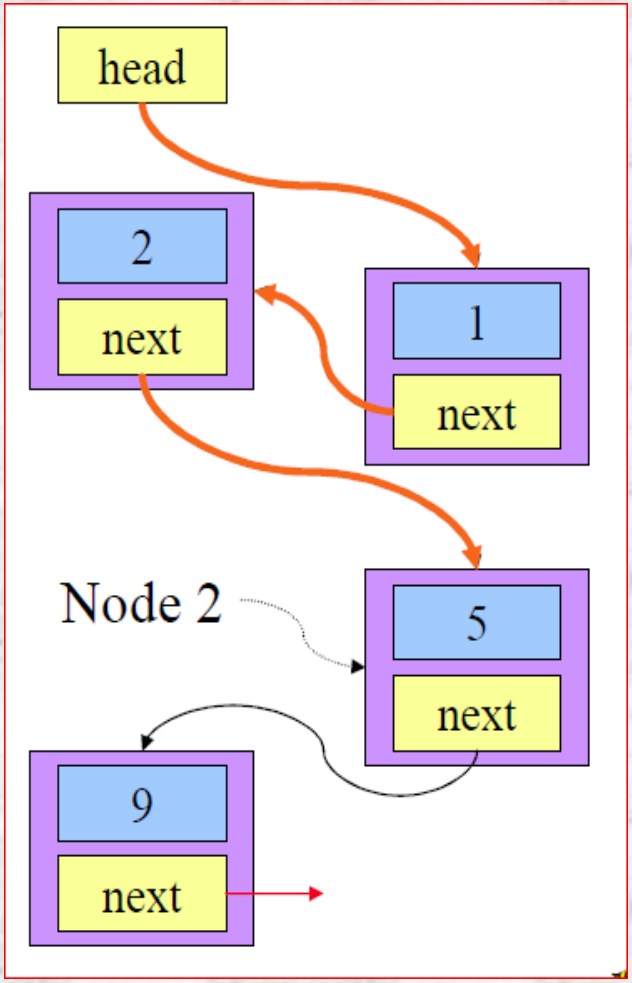
- Deleting a link list has two cases
  - First in the list
  - Not first in the list
- If deleting from head, modify head reference (only)
- If deleting elsewhere, simply “point around” the deleted node
- Be sure to free the deleted nodes



Stop here

Link list pictorial view - TRAVERSE

- Start at the head
  - Use a "current" reference for the current node
- Use the "next" reference to find the next node in the list
- Repeat this to find the desired node
  - *N* times to find the *n*th node
  - Until the object matches if looking for a particular object
    - Caution: objects can "match" even if the references aren't the same…
- Don't forget to check to see if this is the last node



Linked List operations

- Initialize the list
- Push/Insert a value onto the list
- Search the list
- Pop/Remove a value off of the list
- Print the list

```
void InitList(struct list *sList);

/* Initializes the list structure */
void InitList(struct list *sList) {
    sList->start = NULL; }
```

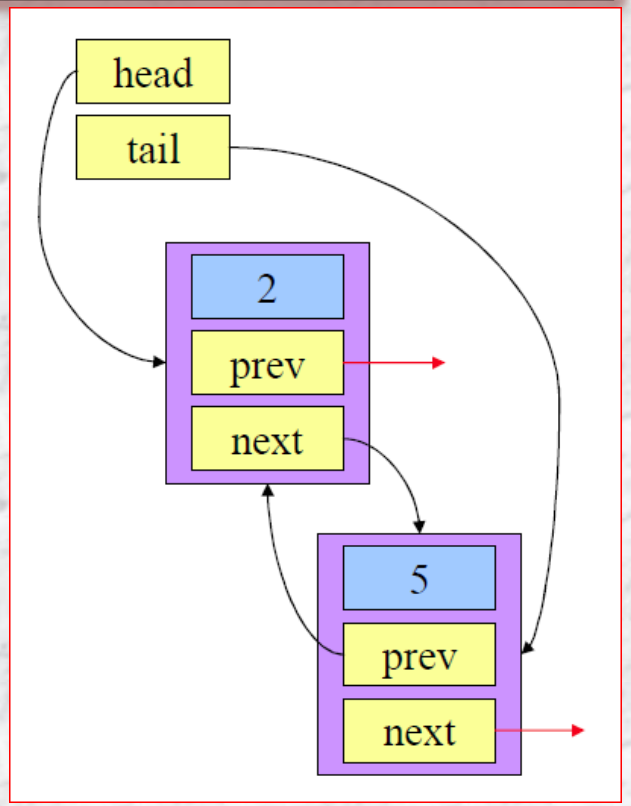
```
void push(struct list *sList, int data);

/* Adds a value to the front of the list */
void push(struct list *sList, int data) {
    struct node *p;
    p = malloc(sizeof(struct node));
     p->data = data;
    p->next = sList->start;
    sList->start = p; }
```

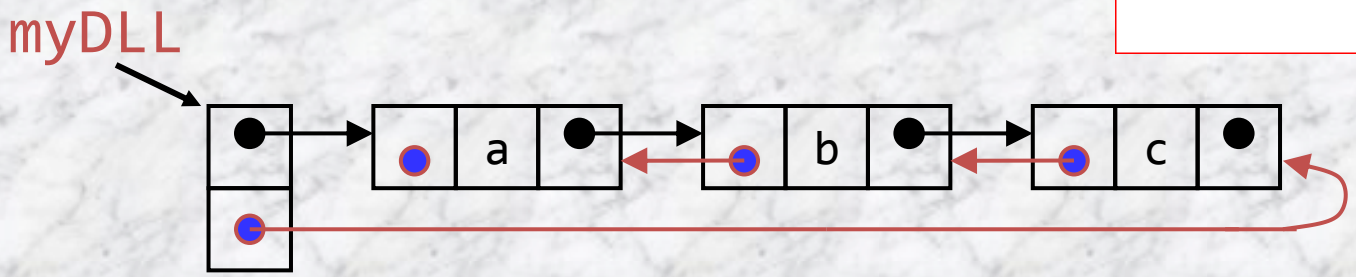
```
void pop(struct list *sList)

/* Removes the first value of the list */
void pop(struct list *sList) {
    if(sList->start != NULL) {
        struct node *p = sList->start;
        sList->start = sList->start->next;
        free(p); } }
```

(see linklst2.c)

Double Linked List (DLL)

- A more sophisticated form of a linked list data structure.
- Each node contains a value, a link to the next node (if any) and a link to the previous node (if any)
- The header points to the first node in the list and to the last node in the list (or contains null links if the list is empty



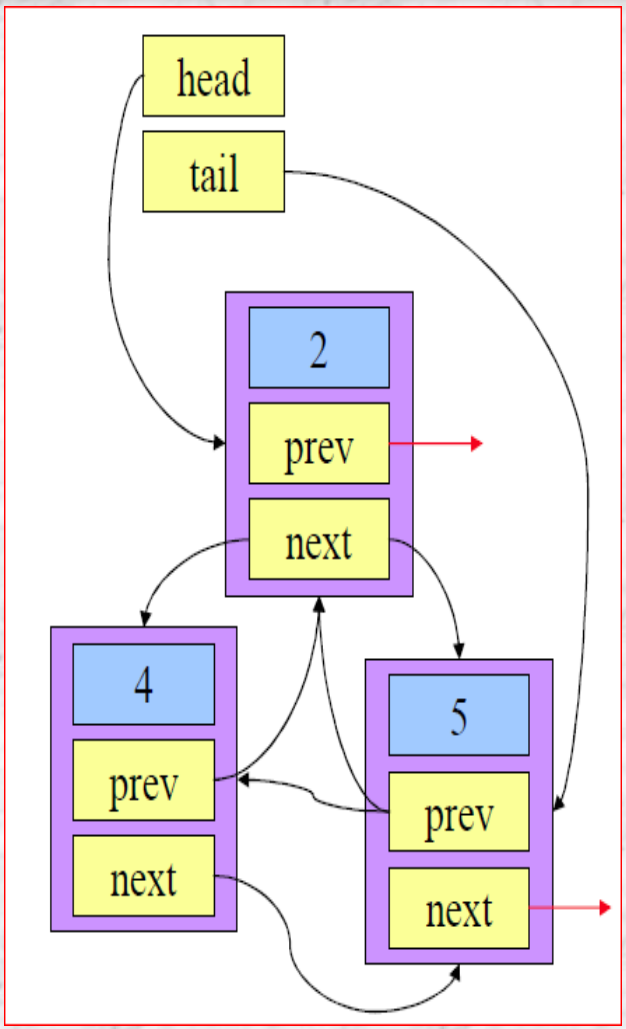


DLLs compared to SLLs

- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier
- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

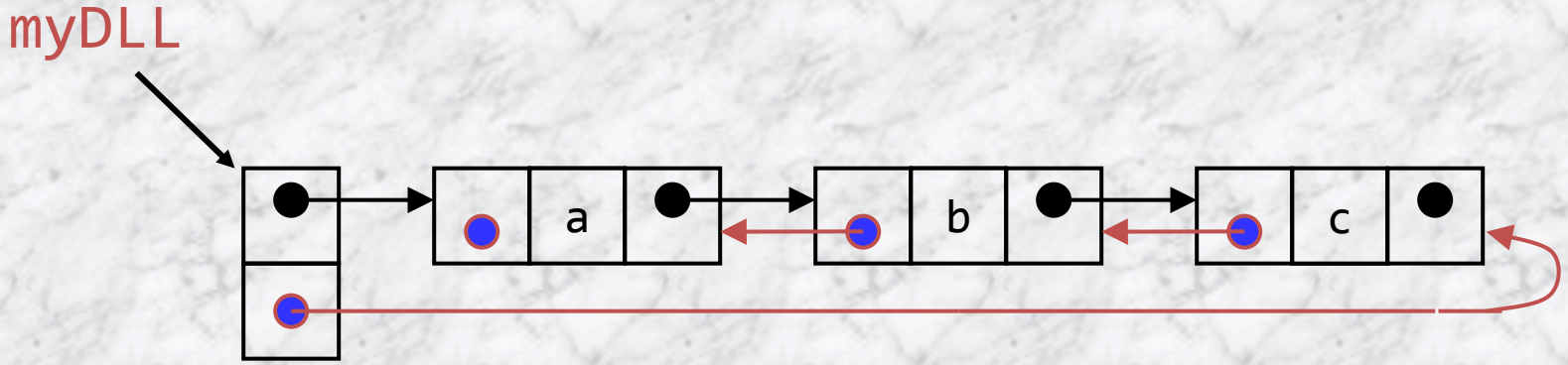
Double linked list – INSERT

- As with singly linked lists, special case for head
  - Also special case for tail
- Need to update two nodes
  - Node before new node
  - Node after new node
- Hook up new node *before* modifying other nodes
  - Don't overwrite necessary information before relocating it!
- Head & tail: if a link is null, update the head or tail as appropriate



Deleting a node from a DLL

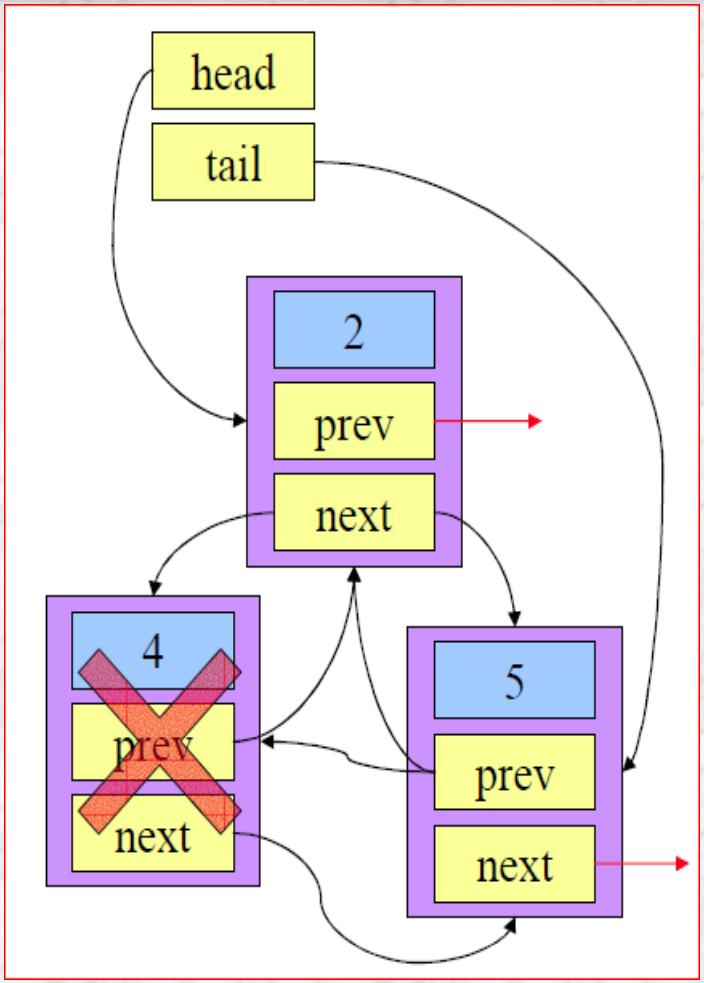
- Node deletion from a DLL involves changing *two* links
- In this example,we will delete node b



- Deletion of the first node or the last node is a special case

Double linked list - DELETE

- As with singly linked lists, special case for head
  - Also special case for tail
- Need to update two nodes
  - Node before new node
  - Node after new node
- Hook up new node *before* modifying other nodes
  - Don't overwrite necessary information before relocating it!
- Head & tail: if a link is null, update the head or tail as appropriate



Other operations on linked lists

- Most "algorithms" on linked lists—such as insertion, deletion, and searching—are pretty obvious; you just need to be careful
- Sorting a linked list is just messy, since you can't directly access the n^{th} element—you have to count your way through a lot of other elements

Double linked lists - SETUP

```
//DECLARATIONS

/* The type link_t needs to be forward-
declared in order that a  self-reference
can be made in "struct link" below. */
typedef struct link link_t;

/* A link_t contains one of the links of
the linked list. */
struct link                  {
    const void * data;
    link_t * prev;
    link_t * next;           };

/* linked_list_t contains a linked list. */
typedef struct linked_list  {
   link_t * first;
   link_t * last;   }  linked_list_t;
```

```
/* The following function initializes
the linked list by putting zeros into
the pointers  containing the first and
last links of the linked list. */
static void
linked_list_init (linked_list_t * list)
{     list->first = list->last = 0;     }
```

Double linked lists - ADD

```
/* The following function adds a new link to the end of the linked   list. It allocates
memory for it. The contents of the link are   copied from "data". */
static void
linked_list_add (linked_list_t * list, void * data){
      link_t * link;
      link = calloc (1, sizeof (link_t));       /* calloc sets the "next" field to zero. */
      if (! link) {
            fprintf (stderr, "calloc failed.\n");
            exit (EXIT_FAILURE);    }
      link->data = data;
      if (list->last) {                 /* Join the two final links together. */
            list->last->next = link;
            link->prev = list->last;
            list->last = link;    }
      else {
            list->first = link;
            list->last = link;    }}
```

Double linked lists - DELETE

```
static void
linked_list_delete (linked_list_t * list, link_t * link) {
        link_t * prev;
        link_t * next;

        prev = link->prev;
        next = link->next;
        if (prev) {
                if (next) {
                                prev->next = next;
                                next->prev = prev; }
                else {
                                prev->next = 0;
                                list->last = prev; } }
        else {
                if (next) {
                                next->prev = 0;
                                list->first = next; }
                else {
                                list->first = 0;
                                list->last = 0; } }
        free (link); }
```

Both the previous and next links are valid, so just bypass “link” without altering “list” at all

Only the previous link is valid, so "prev" is now the last link in "list".

Only the next link is valid, not th eprevious one, so “next” is now the first link in the “list”

Neither previous not next links are valid, so the list is now empty

Double linked lists - FREE

```
 /* Free the list's memory. */ s
tatic void
linked_list_free (linked_list_t * list)
{
      link_t * link;
      link_t * next;
      for (link = list->first; link; link = next)
      {
/* Store the next value so that we
don't  access freed  memory. */
            next = link->next;
            free (link);
       }
}
```

Don't FORGET!