Data Structures in C

C Programming and Software Tools

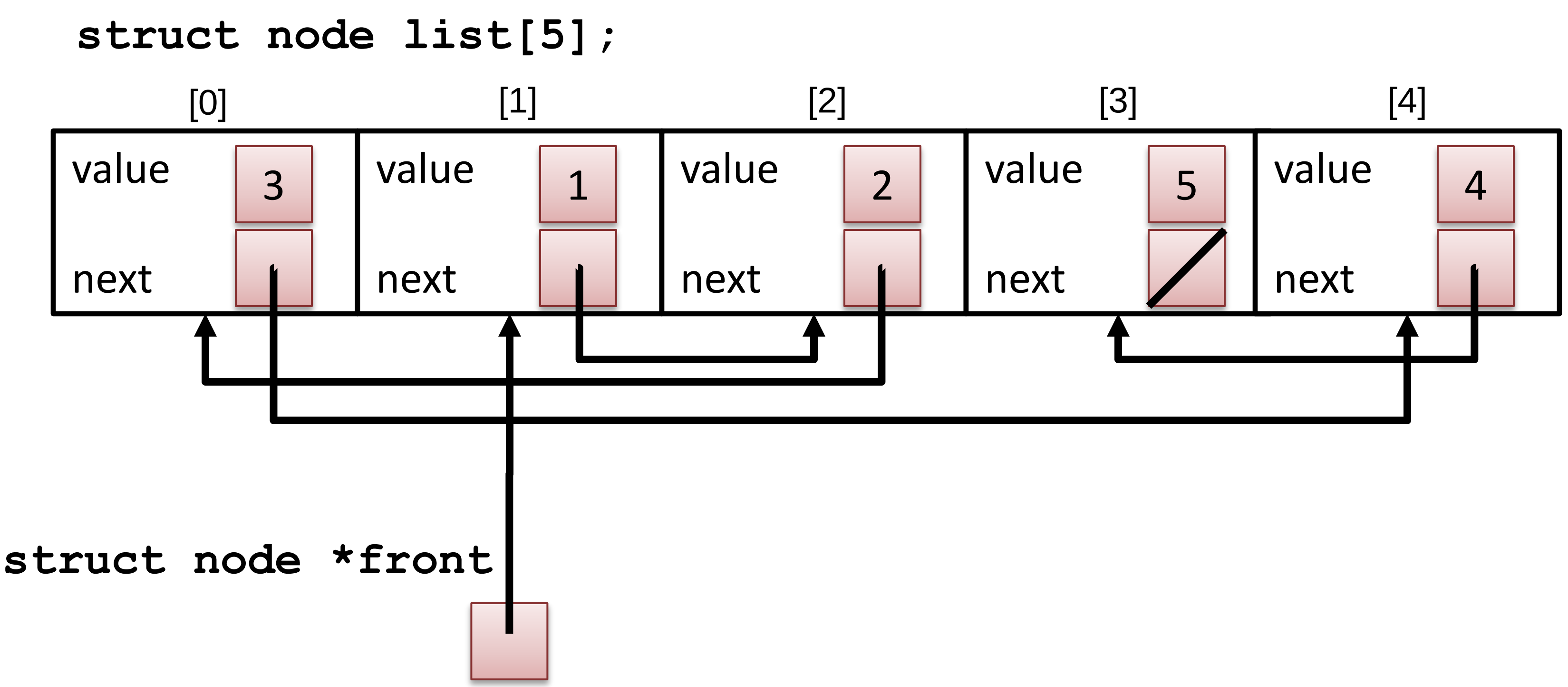
N.C. State Department of Computer Science

Data Structures in C

- The combination of pointers, structs, and dynamic memory allocation allows for creation of data structures
  - **Linked lists**
  - Trees
  - Graphs

Data Structures with Arrays

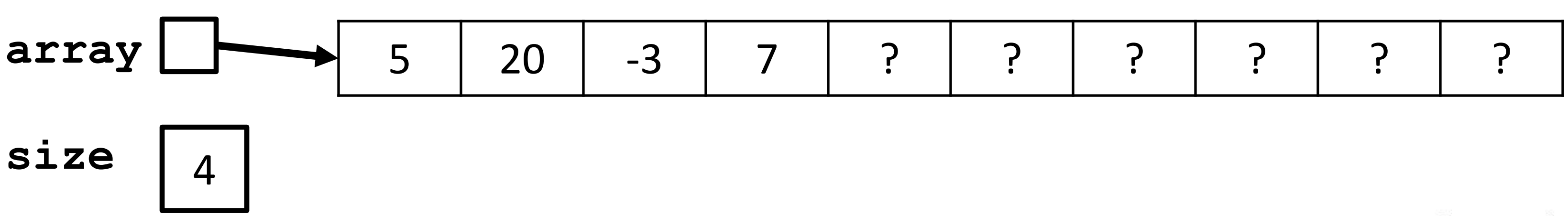
- Without dynamic memory allocation, you could still create these data structures within an array



Array Lists

- Array Lists
  - Elements stored in a partially filled array
  - Size of collection can quickly identify next place to add element (if adding at end of the list)
  - If **`size == capacity`** of array, the array grows “automatically” through the creation of a new, larger array, with the elements copied

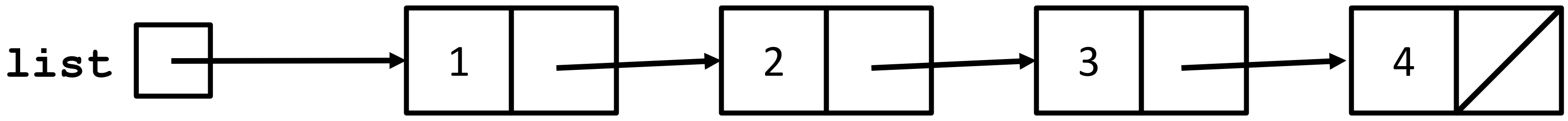
```
int array[CAPACITY];
int size = 0; //initialized
```



Linked Lists

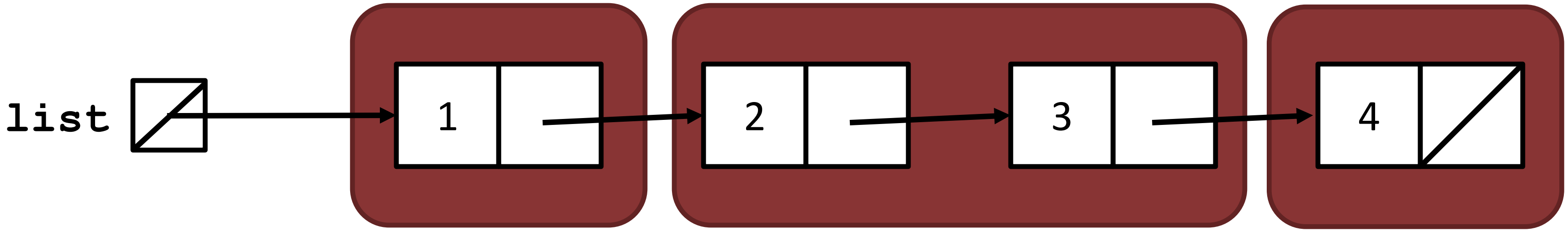
- Linked Lists
  - A **`struct`** represents a single node in the list
  - A node contains a pointer to the **`next`** node in the list
  - A **`NULL`** value represents the end of the list
  - If the **`front`** of the list is **`NULL`**, the list is empty

```
struct node *list = NULL;
```



Lists

- When considering any functionality related to a list collection always consider:
  - An empty list
  - Beginning of the list
  - Middle of the list
  - End of the list

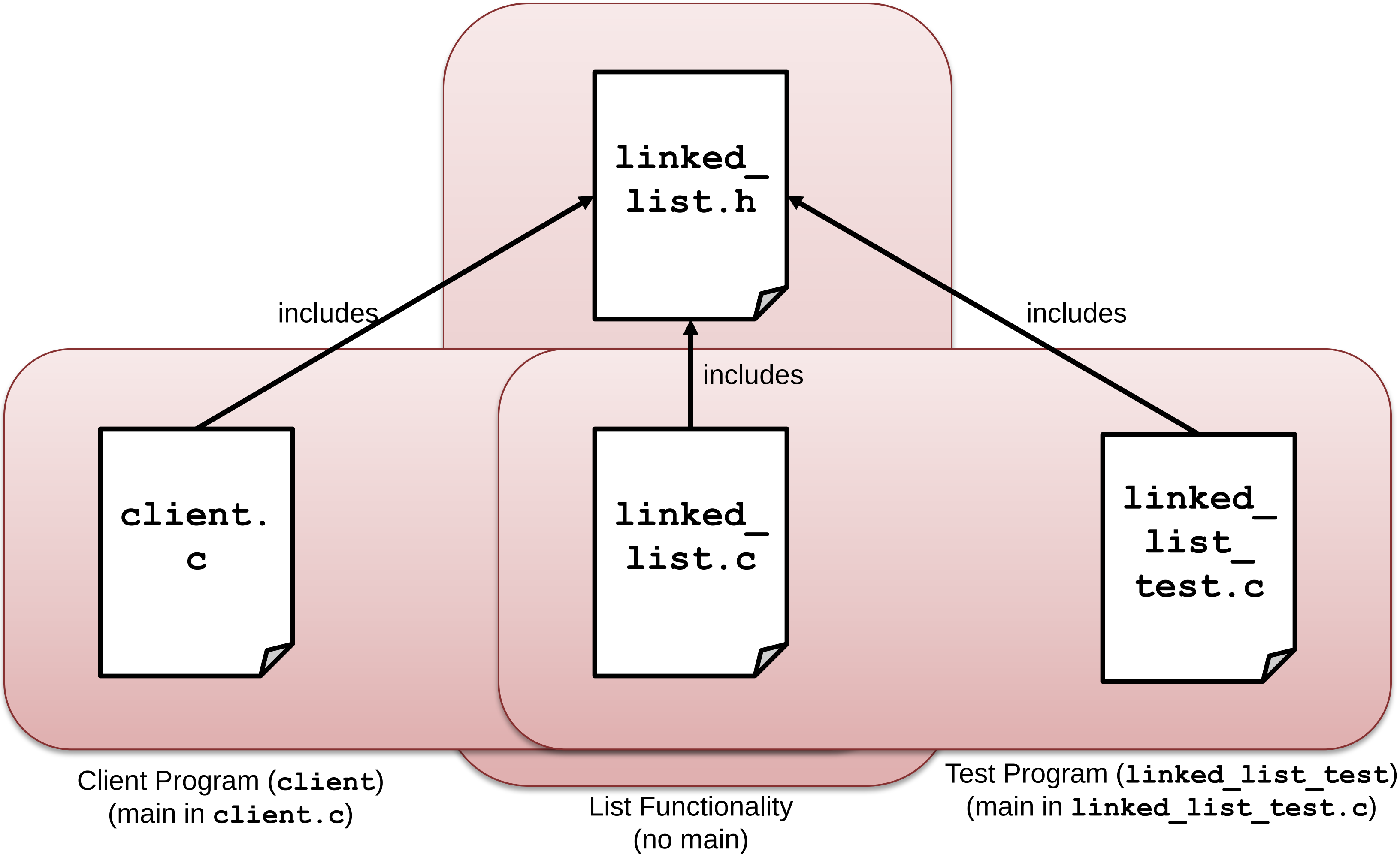


Array List vs. Linked List

For each of the following characteristics, identify if it describes an Array List or a Linked List.

Characteristic	Array-List?	Linked-List?
Access any element via an index in the list in constant time.	Yes ☺	No ☹
Easily grow or shrink the list.	No ☹	Yes ☺
Space only allocated for elements currently in the list.	No ☹	Yes ☺
May have unused space.	Yes ☹	Usually not ☺
Linear runtime efficiency to get an item from the list at a particular index.	No ☺ (it's constant time)	Yes ☹
Adding or removing an element in the middle of the list requires a shift of other elements (as appropriate for the operation).	Yes ☹	No ☺

Designing a List



Declaring a Node Type

- Each node contains data and a pointer to the next node
  - Typically goes in the header file
  - Use a **`struct`**
  - Data can be multiple members of the **`struct`**
  - Be careful with **`typedef`** when defining node type for a linked list.

```
struct node {
   int value;
   struct node *next;
};
```

```
typedef struct node {
   int value;
   struct node *next;
} node;
```

More Complex Node Types

- Put all information a node has in the node struct with the pointer

```
struct node {
    struct node *prev;
    int value;
    char *name;
    double array[LEN];
    struct node *next;
};
```

Doubly Linked Lists

- require a pointer to the previous node
- the **prev** node of the first item in the list is **NULL**

Generic Lists

- Use **void *** and casts

- Abstract the "object" and the node

```
struct object {
    int value;
    char *name;
    double array[LEN];
};

struct node {
    struct node *prev;
    struct object *o;
    struct node *next;
};
```

Creating a List

- A **`list`** is a pointer to the first node in the **`list`**.
  - The **`list`** initially is empty (**`NULL`**)

```
struct node *front = NULL;
```

- Procedural decomposition of list functionality
  - Create functions that represent discrete operations on a list (similar to the **`LinkedList`** and **`ArrayList`** methods)
  - **`add`**, **`remove`**, **`find`**, etc.
  - The functions go in the header file

List Considerations

- Global List
  - Reference to the list in the header file
    - All modules have access to the list
  - Reference to the list in the list module
    - Access restricted to those that include the *.c file, unless static
  - Benefit – Can use return type to signal errors
  - Limitation – ONLY ONE LIST!
- Local List
  - Reference to the list must be passed into all functions
  - Modified list returned from functions
  - Benefit – Many lists
  - Limitation – Other means for signaling error

```
x = change(x);
```

Best Development Practices

- Getting Started
  - Create Makefile from design
  - Stub out the program with the appropriate functions returning something
  - Compile with no warnings
  - Download the starter zip, which has a linked list program stubbed out
- Test Driven Development
  - Write the tests BEFORE starting development
  - Use them to drive development forward

Testing a List

- General Procedure
  - Manipulate the list
  - See if the manipulations result in the correct list
- Baseline Functions for Testing
  - **`size()`**
  - **`get_at()`**
  - **`add_at()`**
- Considerations
  - An empty list
  - Beginning of the list
  - Middle of the list
  - End of the list

Create Tests for `size()`

Inputs	List	Size
Empty list	[]	0
Add 1 to index 0	[1]	1
Add 2 to index 1	[1, 2]	2
Remove 1 from index 0	[2]	1
Remove 2 from index 0	[]	0

```
void test_size()
{
   //Create list
   node *list = NULL;
   check_int("Empty list", 0, size(list)); //Test 1

   //Add element to index 0
   list = add_at(list, 0, 1);
   check_int("Add 1 to index 0", 1, size(list));

   //Add the rest of the tests here
}
```

Implementing `size()`

- Algorithm: Traverse the list and count the nodes
- Alternative: Keep track of the nodes as added/removed
  - Requires variable for each list - doesn't fit with our design
  - Alternative design to accommodate size with the list later
- Tests will not fully pass until we implement **`add_at()`** and **`remove_at()`**

Traversing a List

- Algorithm
  - Start at first element
  - Manipulate data for element
  - Move to next element
  - If the element is **`NULL`**, you're done
- Make sure you don't lose your list!
  - Create a pointer that you use specifically for traversing the list

Create Tests for `add_at()`

Inputs	List	Size
Empty List: add_at(list, 0, 3)	[3]	1
Add Front: add_at(list, 0, 2)	[2, 3]	2
Add Middle: add_at(list, 1, 7)	[2, 7, 3]	3
Add End: add_at(list, 3, 45)	[2, 7, 3, 45]	4

```
void test_add_at()
{
   //Create list
   node *list = NULL;
   list = add_at(list, 0, 3);
   //Check size AND contents
   check_int("Add to empty list", 1, size(list));
   check_int("Index 0", 3, get_at(list, 0));

   //Add the rest of the tests here
}
```

Adding a Node to a List

- Adding a node to a list has three steps
  1. Allocating memory for the node
  2. Storing data in the node
  3. Inserting the node into the list
     - Consider empty list, front of the list, middle of the list, and end of the list
     - There may be specializations
- Other Considerations
  – If the index is out of bounds, just return the list (for the moment)

Getting a Node from a List

- Getting a node has the following steps:
  - Traverse the list until the given index
  - Return the value at the index
- What happens if the index is out of bounds?
  - Can't ONLY return -1 or 0, because that could be a value in the list

Using the **errno.h** Library

- Library for signaling errors
  - Special return type signals a check for possible error
  - If error, **errno** is set to constant value
  - Reset **errno** to 0 after the check to find the next error

```
int get_at(node *list, int idx)
{
   if (idx < 0 || idx >= size(list)) {
      errno = EIDXOUTOFBOUNDS;
      return -1;
   }
   //Implement get_at() for valid indices
}
```

Testing Error Paths

```
/* Checks the errno against the expected errno when testing
 * error paths. The test function should get the value out of
 * errno immediatly after a function call that should generate
 * an error and pass it into the check.
 * The errno is reset to 0 after a call to this function.
 */
void check_errno(char * description, int exp, int act)
{
   printf("%60s %20d %20d %4s\n", description, exp, act,
               assert_equals(exp, act));
   errno = 0;
}

void test_get_at()
{
   node *list = NULL;

   errno = 0;
   int rtn = get_at(list, -3);
   int my_errno = errno;
   check_int("Index -3 in empty list", -1, rtn);
   check_errno("Index -3 in empty list", EOUTOFBOUNDS, my_errno);
}
```

Cleaning Up Memory

```
valgrind --leak-check=yes ./linked_list_test
```

- Run Valgrind on our test program
  - Lots of leaked memory!!!
- If you create it, you must destroy it
  - Implement **`remove_at()`** and call for every node created in the tests
  - Implement a **`free_all()`** and call at the end of every test function (if needed)
- Run Valgrind again – No Leaks!

Trailing Pointer Technique

- Checking that the current node or the current's next node is NULL isn't sufficient
  - Instead, we want to stop at or maintain a pointer to the node BEFORE the one we want to delete while moving on to the next
  - "Trailing pointer" technique (see book Section 17.5)

```
void free_all(node *list)
{
   if (list == NULL) return;
   node *cur = list;
   while (cur->next != NULL) {
      node *p = cur; //Trailing pointer
      cur = cur->next;
      printf("%d\n", p->value);
      free(p);  //Saved so we can free previous node
   }
   printf("%d\n", cur->value);
   free(cur);
}
```

Removing a Node from a List

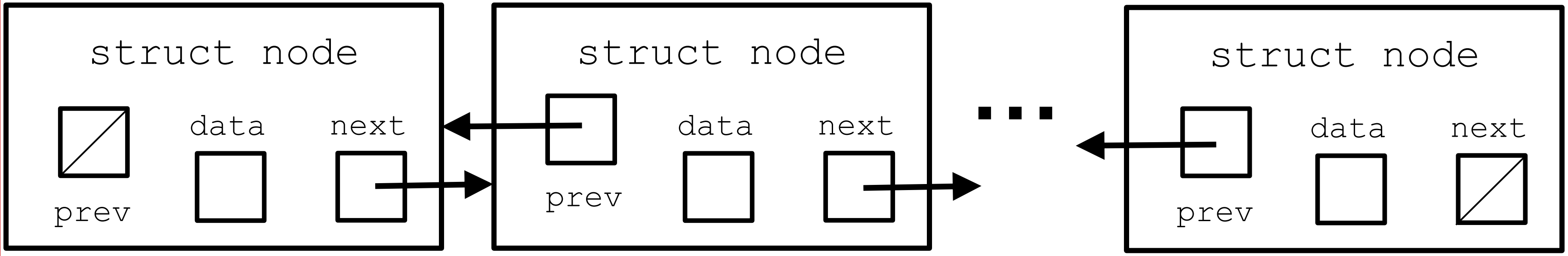
- Removing a node from a list has three steps
  1. Locate the node to be deleted
  2. Alter the previous node so that it "bypasses" the deleted node
  3. Free the memory to reclaim space of deleted node
- Other Considerations
  - If the index is out of bounds, set **`errno`** to **`EOUTOFBOUNDS`** and return **`NULL`**
    - WARNING: When a client of **`remove_at()`**, you should not store the result of **`remove_at()`** to your list immediately! Instead, you should create a temp, check for **`NULL`**, and then store.
- Updates
  - Update **`add_at()`** to have the same check

Specialized Data Structures

- More efficient on add/remove/search operations
- Sorted list
  - Faster search – can quit earlier if can't find value
- Stacks
  - Add and remove from same end
- Queues
  - Add at one end and remove from the other end
  - Optimize speed of access by creating a doubly linked list and maintaining a reference to the front and back of the list

Doubly Linked Lists

- A **`node`** maintains a pointer to the **`node`** *before* and *after* it in the list.
- All list operations need to update appropriate **`prev`** and **`next`** pointers.
- Can maintain a pointer to the back of the list




```
struct node {
    struct node *prev;
    int value;
    struct node *next;
};
```

Further Generalizing the List

```
struct list {
    void *front;
    void *back;
    int size;
};
```

```
struct node {
    struct node *prev;
    void *data;
    struct node *next;
};
```