C Data Structures

CS 2060

Prof. Jonathan Ventura

Data structures

- **Data structures** are particular ways of organizing data that provide useful properties.

Data structures

- **Data structures** are particular ways of organizing data that provide useful properties.
- So far we have used two simple kinds of **static data structures** in C:

Data structures

- **Data structures** are particular ways of organizing data that provide useful properties.
- So far we have used two simple kinds of **static data structures** in C:
  - **Arrays** provide fixed-size, contiguous lists of homogeneous data.

Data structures

- **Data structures** are particular ways of organizing data that provide useful properties.
- So far we have used two simple kinds of **static data structures** in C:
  - **Arrays** provide fixed-size, contiguous lists of homogeneous data.
  - **Structures** provide fixed-size collections of heterogeneous data.

Data structures

- **Dynamic data structures** can grow or shrink in size according to how much data they need to store.

Data structures

- **Dynamic data structures** can grow or shrink in size according to how much data they need to store.
  - **Linked lists** provide efficient *insertion* and *deletion* anywhere in the list.

Data structures

- **Dynamic data structures** can grow or shrink in size according to how much data they need to store.
  - **Linked lists** provide efficient *insertion* and *deletion* anywhere in the list.
  - **Stacks** provide *last-in, first-out* (LIFO) access to data.

Data structures

- **Dynamic data structures** can grow or shrink in size according to how much data they need to store.
  - **Linked lists** provide efficient *insertion* and *deletion* anywhere in the list.
  - **Stacks** provide *last-in, first-out* (LIFO) access to data.
  - **Queues** provide *first-in, first-out* (FIFO) access to data.

Data structures

- **Dynamic data structures** can grow or shrink in size according to how much data they need to store.
  - **Linked lists** provide efficient *insertion* and *deletion* anywhere in the list.
  - **Stacks** provide *last-in, first-out* (LIFO) access to data.
  - **Queues** provide *first-in, first-out* (FIFO) access to data.
  - **Binary trees** *recursively partition* data and provide fast *search* and *sorting*.

Dynamic array

- Let's first look at how to implement a **dynamic array** data structure which provides the following functionality:
  - Stores homogeneous data in a contiguous block of memory.
  - Allows the list to resized bigger or smaller.
- How could we implement this?
  - Assume that we will store `ints`.

Dynamic array

- Let's start by defining the `struct` and functions that we will need.

```
typedef struct {
    // ...
} DynamicArray;

DynamicArray * makeArray( size_t length );
void destroyArray( DynamicArray * array );
void resizeArray( DynamicArray *array,
                  size_t new_length );
```

Dynamic array

```
typedef struct {
    size_t length;
    int * data;
} DynamicArray;

DynamicArray * makeArray( size_t length );
void destroyArray( DynamicArray * array );
void resizeArray( DynamicArray *array,
                  size_t new_length );
```

Dynamic array

```
DynamicArray * makeArray( size_t length ) {
    // Allocate space for the struct
    DynamicArray * array = malloc( sizeof(DynamicArray) );

    // Set the array length
    // ...

    // Allocate space for length ints
    // ...

    // Return the struct pointer
    return array;
}
```

Dynamic array

```
DynamicArray * makeArray( size_t length ) {
    // Allocate space for the struct
    DynamicArray * array = malloc( sizeof(DynamicArray) );

    // Set the array length
    array->length = length;

    // Allocate space for length ints
    array->data = calloc( length, sizeof(int) );

    // Return the struct pointer
    return array;
}
```

Dynamic array

```
void destroyArray( DynamicArray * array ) {

    // Free the array data
    // ...

    // Free the array itself
    // ...

}
```

Dynamic array

```
void destroyArray( DynamicArray * array )
{

    // Free the array data
    free( array->data );

    // Free the array itself
    free( array );

}
```

Dynamic array

```
void resizeArray( DynamicArray *array,
                  size_t new_length )
{

    // Re-allocate the data
    // to accomodate new_length ints
    // ...

}
```

Dynamic array

```
void resizeArray( DynamicArray *array,
                  size_t new_length )
{

    // Re-allocate the data
    // to accomodate new_length ints
    array->data = realloc( array->data,
                           new_length * sizeof(int) );

}
```

Dynamic array usage

```c
int main() {

    // Read in integers from the console,

    // compute their average,

    // and count the number of inputs below average.

    return 0;
}
```

Dynamic array usage

```
// Read in integers from the console,
DynamicArray *array = makeArray( 0 );
int input;
while ( scanf( "%d", &input ) == 1 ) {
    // Store input at the end of the array
}
```

Dynamic array usage

```
// Read in integers from the console
DynamicArray *array = makeArray( 0 );
int input;
while ( scanf( "%d", &input ) == 1 ) {

    // Make array bigger
    resizeArray( array, array->length + 1 );

    // Store value at end of array
    array->data[ array->length-1 ] = input;
}
```

Dynamic array usage

```
// Compute the average
float avg = 0;
// ...
```

Dynamic array usage

```
// Compute the average
float avg = 0;
for ( int i = 0; i < array->length; i++ )
    avg += array->data[i];
avg /= array->length;
```

Dynamic array usage

```
// Count number of inputs below average
int num_below = 0;
// ...
```

Dynamic array usage

```
// Count number of inputs below average
int num_below = 0;
for ( int i = 0; i < array->length; i++ )
    if ( array->data[i] < avg ) num_below++;
printf( "%d\n", num_below );
```

Dynamic array usage – improving performance

```
// Read in integers from the console
DynamicArray *array = makeArray( 100 );
int num_inputs = 0, input;
while ( scanf( "%d", &input ) == 1 ) {
    // Increment input counter
    num_inputs++;

    // Make array bigger if necessary
    if ( num_inputs > array->length )
        resizeArray( array, array->length+100 );

    // Store value in array
    array->data[ num_inputs-1 ] = input;
}
```

Insertion and deletion

- Now imagine that we want to **insert** a value in the *middle* of a dynamic array.
- How could we implement this using the dynamic array we just developed?

Insertion and deletion

- To insert into the middle of a dynamic array, we would have to:
  - *Resize* the array to increment its length, and then
  - *Move* the data at the end to make room for the insertion.
- For example, assume we want to insert 3 at index 1:

Index	0	1	2	3
Value	5	2	4	7

Insertion and deletion

- To insert into the middle of a dynamic array, we would have to:
  - *Resize* the array to increment its length, and then
  - *Move* the data at the end to make room for the insertion.
- We have to first resize the array:

Index	0	1	2	3	4
Value	5	2	4	7	-

Insertion and deletion

- To insert into the middle of a dynamic array, we would have to:
  - *Resize* the array to increment its length, and then
  - *Move* the data at the end to make room for the insertion.
- Then we have to copy the data:

Index	0	1	2	3	4
Value	5	2	**2**	**4**	**7**

Insertion and deletion

- To insert into the middle of a dynamic array, we would have to:
  - *Resize* the array to increment its length, and then
  - *Move* the data at the end to make room for the insertion.
- Now we can write in the new value:

Index	0	1	2	3	4
Value	5	**3**	2	4	7

Insertion and deletion

- Memory re-allocation and copying are expensive operations.

Insertion and deletion

- Memory re-allocation and copying are expensive operations.
- The dynamic array is not ideal for insertion or deletion values in the middle of the array.

Insertion and deletion

- Memory re-allocation and copying are expensive operations.
- The dynamic array is not ideal for insertion or deletion values in the middle of the array.
- Insertion or deletion at the end is okay.

Linked list

- Like a dynamic array, a **linked list** maintains a sequence of data values which can grow or shrink in size.

Linked list

- Like a dynamic array, a **linked list** maintains a sequence of data values which can grow or shrink in size.
- However, linked lists are ideally suited for insertion or deletion anywhere in the list.

Linked list

- Like a dynamic array, a **linked list** maintains a sequence of data values which can grow or shrink in size.
- However, linked lists are ideally suited for insertion or deletion anywhere in the list.
- A linked list is a fundamental data structure which is the basis for many other data structures.

Linked list

- A linked list does not maintain a *contiguous* block of memory for the data.

Linked list

- A linked list does not maintain a *contiguous* block of memory for the data.
- Instead, each value is contained in its own structure.

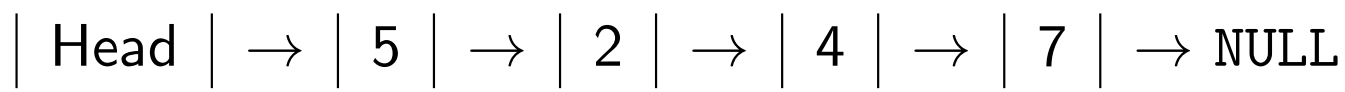
Linked list

- A linked list does not maintain a *contiguous* block of memory for the data.

- Instead, each value is contained in its own structure.
- In addition, we maintain *links* or connections between neighboring data values, to represent the sequence.
  - These links are implemented using **self-referencing structs**.

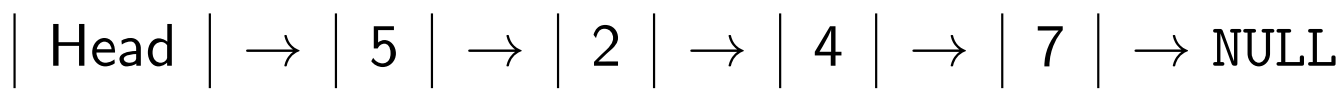
Singly-linked list

- In a singly-linked list, each element maintains a link to the *next* element.



Singly-linked list

- In a singly-linked list, each element maintains a link to the *next* element.



- The **head** node is always at the start of the list.
  - It does not contain any data.

Singly-linked list

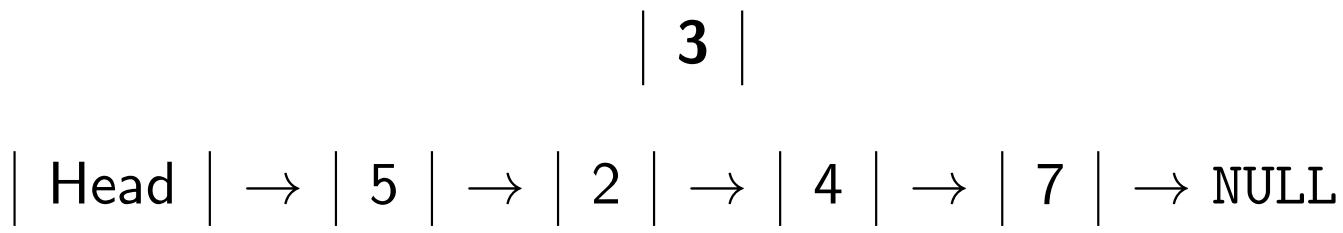
- In a singly-linked list, each element maintains a link to the *next* element.

$$\mid \text{Head} \mid \rightarrow \mid 5 \mid \rightarrow \mid 2 \mid \rightarrow \mid 4 \mid \rightarrow \mid 7 \mid \rightarrow \texttt{NULL}$$

- The **head** node is always at the start of the list.
  - It does not contain any data.
- The last node links to `NULL`, indicating the end of the list.

Singly-linked list

- **Inserting** an element is simple with a linked list.
- First, we create a new node with the element to be inserted.



Singly-linked list

- **Inserting** an element is simple with a linked list.
- Then, we detach and re-attach the links to insert the node.

| Head | → | 5 | → | **3** | → | 2 | → | 4 | → | 7 | → NULL

Linked list code

Let's write a linked list which contains integers.

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

struct LinkedListNode
{
    // ...
};
```

Linked list code

Each node contains a value and a link.

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

struct LinkedListNode
{
    // Data at this node
    int value;

    // Link to next node
    struct LinkedListNode *next;
};
```

Linked list code

```
// Make a new linked list node
// with empty next pointer
struct LinkedListNode * makeLLNode( int value )
{
    // Allocate new node
    struct LinkedListNode *node =
        malloc( sizeof( struct LinkedListNode ) );

    // Initialize node
    // ...

    return node;
};
```

Linked list code

```
// Make a new linked list node
// with empty next pointer
struct LinkedListNode * makeLLNode( int value )
{
    // Allocate new node
    struct LinkedListNode *node =
        malloc( sizeof( struct LinkedListNode ) );

    // Initialize node
    node->value = value;
    node->next = NULL;

    return node;
};
```

Linked list code

```
// Insert node after a given node
void insert( struct LinkedListNode *node,
             struct LinkedListNode *insert )
{
    // Set links so that insert is after node

    // Before: node -> next
    // After:  node -> insert -> next

    // ...
};
```

Linked list code

```
// Insert node after a given node
void insert( struct LinkedListNode *node,
             struct LinkedListNode *insert )
{
    // Set links so that insert is after node

    // Before: node -> next
    // After:  node -> insert -> next

    insert->next = node->next;
    node->next = insert;
};
```

Linked list code

```
// Delete node after a given node
void delete( struct LinkedListNode *node )
{
    // Before: node -> delete -> next
    // After:  node -> next

    // ...
};
```

Linked list code

```
// Delete node after a given node
void delete( struct LinkedListNode *node )
{
    // Before: node -> delete -> next
    // After:  node -> next

    // careful...
    node->next = node->next->next;
};
```

Linked list code

```
// Delete node after a given node
void delete( struct LinkedListNode *node )
{
    // Before: node -> delete -> next
    // After:  node -> next

    if ( node->next )
        node->next = node->next->next;

    // memory leak...
};
```

Linked list code

```
// Delete node after a given node
void delete( struct LinkedListNode *node )
{
    // Before: node -> delete -> next
    // After:  node -> next

    if ( node->next ) {
        struct LinkedListNode *save = node->next;
        node->next = node->next->next;
        free( save );
    }
};
```

Linked list code

```
int main() {
    // Make head node
    struct LinkedListNode *head =
        makeLLNode( 0 );

    // Insert numbers at beginning of list
    for ( int i = 0; i < 10; i++ ) {
        struct LinkedListNode *node =
            makeLLNode( i );

        insert( head, node );
    }
```

Linked list code

```
// Print the nodes
struct LinkedListNode *node = head->next;
while ( node ) {
    printf( "%d ",node->value );
    node = node->next;
}
```

Linked list code

```
    // Delete all the nodes
    while ( head->next ) {
        delete( head );
    }

    // Delete the head node
    free( head );

    return 0;
}
```

Stack

- A stack is a *last-in first-out* (LIFO) data structure.
- We *push* (insert) and *pop* (delete) at the top of the stack.

Empty stack

Stack

- A stack is a *last-in first-out* (LIFO) data structure.
- We *push* (insert) and *pop* (delete) at the top of the stack.

After pushing 5

Stack

- A stack is a *last-in first-out* (LIFO) data structure.
- We *push* (insert) and *pop* (delete) at the top of the stack.

| Top | **2** | 5 |

After pushing 2

Stack

- A stack is a *last-in first-out* (LIFO) data structure.
- We *push* (insert) and *pop* (delete) at the top of the stack.

| Top | **4** | 2 | 5 |

After pushing 4

Stack

- A stack is a *last-in first-out* (LIFO) data structure.
- We *push* (insert) and *pop* (delete) at the top of the stack.

| Top | 2 | 5 |

Pop 4

Stack

- A stack is a *last-in first-out* (LIFO) data structure.
- We *push* (insert) and *pop* (delete) at the top of the stack.

| Top | 5 |

Pop 2

Stack

- A stack is a *last-in first-out* (LIFO) data structure.
- We *push* (insert) and *pop* (delete) at the top of the stack.

| Top |

Pop 5

Implementing a stack

- A stack can be implemented like a linked list, with **nodes** and **links**.

Implementing a stack

- A stack can be implemented like a linked list, with **nodes** and **links**.
- What is the difference to a linked list?

Implementing a stack

- A stack can be implemented like a linked list, with **nodes** and **links**.
- What is the difference to a linked list?
- What functions do we need?

Stack

- Our strategy will be to maintain a **top** node that marks the top of the stack.
- We will only push or pop at the top of the stack.

Stack code

Let's write a stack which contains integers.

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

struct StackNode
{
    // ...
};
```

Stack code

Let's write a stack which contains integers.

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

struct StackNode
{
    // Data at this node
    int value;

    // Link to next node
    struct StackNode *next;
};
```

Stack code

```
// Make a new stack node
// with empty next pointer
struct StackNode * makeStackNode( int value )
{
    // Allocate new node
    struct StackNode *node =
        malloc( sizeof( struct StackNode ) );

    // Initialize node
    // ...

    return node;
};
```

Stack code

```
// Make a new stack node
// with empty next pointer
struct StackNode * makeStackNode( int value )
{
    // Allocate new node
    struct StackNode *node =
        malloc( sizeof( struct StackNode ) );

    // Initialize node
    node->value = value;
    node->next = NULL;

    return node;
}
```

Stack code

```
// Push node to top of stack
void push( struct StackNode *top,
           struct StackNode *insert )
{
    // Set links so that insert is after top

    // Before: top -> next
    // After:  top -> insert -> next

    // ...
}
```

Stack code

```
// Push node to top of stack
void push( struct StackNode *top,
           struct StackNode *insert )
{
    // Set links so that insert is after top

    // Before: top -> next
    // After:  top -> insert -> next

    insert->next = top->next;
    top->next = insert;
}
```

Stack code

```
// Pop node at top of stack and get value
// Return 1 for success, 0 otherwise
int pop( struct StackNode *top, int *value )
{
    // Before: top -> delete -> next
    // After:  top -> next

    // ...
}
```

Stack code

```
// Pop node at top of stack and get value
// Return 1 for success, 0 otherwise
int pop( struct StackNode *top, int *value )
{
    // Before: top -> delete -> next
    // After:  top -> next

    // careful...
    *value = top->next->value;
    top->next = top->next->next;

    return 1;
}
```

Stack code

```
// Pop node at top of stack and get value
// Return 1 for success, 0 otherwise
int pop( struct StackNode *top, int *value )
{
    // Before: top -> delete -> next
    // After:  top -> next

    if ( top->next ) {
        *value = top->next->value;
        top->next = top->next->next;
        // memory leak...
        return 1;
    }

    return 0;
```

Stack code

```
// Pop node at top of stack and get value
// Return 1 for success, 0 otherwise
int pop( struct StackNode *top, int *value )
{
    // Before: top -> delete -> next
    // After:  top -> next

    if ( top->next ) {
        *value = top->next->value;

        struct StackNode *save = top->next;
        top->next = top->next->next;
        free( save );

        return 1;
```

Stack usage example

```
int main() {
    // Make top node
    struct StackNode *top = makeStackNode( 0 );

    // Push numbers onto stack
    for ( int i = 0; i < 10; i++ ) {
        struct StackNode *node =
            makeStackNode( i );

        push( top, node );
    }
```

Stack usage example

```
// Print and empty the stack
// ... ?
```

Stack usage example

```
// Print and empty the stack
int value;
while ( pop( top, &value ) ) {
    printf( "%d ",value );
}
```

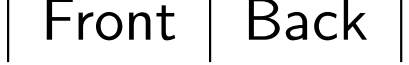
Stack usage example

```
    // Delete the top node
    free( top );

    return 0;
}
```

Queue

- A **queue** is a *first-in first-out* (FIFO) data structure.
- We **enqueue** (insert) at the **back** and **dequeue** (delete) at the **front** of the queue.



Empty queue

Queue

- A **queue** is a *first-in first-out* (FIFO) data structure.
- We **enqueue** (insert) at the **back** and **dequeue** (delete) at the **front** of the queue.

| Front | **5** | Back |

After enqueuing 5

Queue

- A **queue** is a *first-in first-out* (FIFO) data structure.
- We **enqueue** (insert) at the **back** and **dequeue** (delete) at the **front** of the queue.

| Front | 5 | **2** | Back |

After enqueuing 2

Queue

- A **queue** is a *first-in first-out* (FIFO) data structure.
- We **enqueue** (insert) at the **back** and **dequeue** (delete) at the **front** of the queue.

| Front | 5 | 2 | **4** | Back |

After enqueuing 4

Queue

- A **queue** is a *first-in first-out* (FIFO) data structure.
- We **enqueue** (insert) at the **back** and **dequeue** (delete) at the **front** of the queue.

| Front | 2 | 4 | Back |

After dequeuing 5

Queue

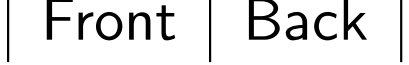
- A **queue** is a *first-in first-out* (FIFO) data structure.
- We **enqueue** (insert) at the **back** and **dequeue** (delete) at the **front** of the queue.

| Front | 4 | Back |

After dequeuing 2

Queue

- A **queue** is a *first-in first-out* (FIFO) data structure.
- We **enqueue** (insert) at the **back** and **dequeue** (delete) at the **front** of the queue.



After dequeuing 4

Implementing a queue

- A queue can also be implemented like a linked list, with **nodes** and **links**.

Implementing a queue

- A queue can also be implemented like a linked list, with **nodes** and **links**.
- What is the difference to a linked list?

Implementing a queue

- A queue can also be implemented like a linked list, with **nodes** and **links**.
- What is the difference to a linked list?
- What functions do we need?

Queue

- Our strategy will be to maintain a **front** node that marks the front of the queue.
- We also need a *pointer* to the **back** node; we won't use a special marker node for the back (for reasons to be seen shortly...)

| Front | 4 | 2 | 5 (Back) |

Queue code

Let's write a queue which contains integers.

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

struct QueueNode
{
    // ...
};
```

Queue code

Let's write a queue which contains integers.

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

struct QueueNode
{
    // Data at this node
    int value;

    // Link to next node
    struct QueueNode *next;
};
```

Queue code

```
// Make a new queue node
// with empty next pointer
struct QueueNode * makeQueueNode( int value )
{
    // Allocate new node
    struct QueueNode *node =
        malloc( sizeof( struct QueueNode ) );

    // Initialize node
    // ...

    return node;
};
```

Queue code

```
// Make a new queue node
// with empty next pointer
struct QueueNode * makeQueueNode( int value )
{
    // Allocate new node
    struct QueueNode *node =
        malloc( sizeof( struct QueueNode ) );

    // Initialize node
    node->value = value;
    node->next = NULL;

    return node;
}
```

Queue code

```
// Enqueue node at back of queue
// Returns the new back of the queue
struct QueueNode * enqueue( struct QueueNode *back,
                struct QueueNode *insert )
{
    // Set links so that insert is after back

    // Before: back -> NULL
    // After:  back -> insert -> NULL

    // ...
}
```

Queue code

```
// Enqueue node at back of queue
// Returns the new back of the queue
struct QueueNode * enqueue( struct QueueNode *back,
                struct QueueNode *insert )
{
    // Set links so that insert is after back

    // Before: back -> NULL
    // After:  back -> insert -> NULL

    insert->next = NULL;
    back->next = insert;

    // what to return?
}
```

Queue code

```
// Enqueue node at back of queue
// Returns the new back of the queue
struct QueueNode * enqueue( struct QueueNode *back,
                struct QueueNode *insert )
{
    // Set links so that insert is after back

    // Before: back -> NULL
    // After:  back -> insert -> NULL

    insert->next = NULL;
    back->next = insert;

    return insert;
}
```

Queue code

```
// Dequeue node at front of queue and get value
// Returns the new back of the queue
// or NULL on failure
struct QueueNode * dequeue( struct QueueNode *front, struct Queu
                int *value )
{
    // Before: front -> delete -> next
    // After:  front -> next

    // ...
}
```

Queue code

```
// Dequeue node at front of queue and get value
// Returns the new back of the queue or NULL on failure
struct QueueNode * dequeue( struct QueueNode *front, struct Queu
                int *value ) {
    // Before: front -> delete -> next
    // After:  front -> next

    if ( front->next ) {
        // ...
    }

    return NULL;
}
```

Queue code

```
// Dequeue node at front of queue and get value
// Returns the new back of the queue or NULL on failure
struct QueueNode * dequeue( struct QueueNode *front, struct QueueNode *back,
                int *value ) {
    // Before: front -> delete -> next
    // After:  front -> next

    if ( front->next ) {
        // Get value from first node
        *value = front->next->value;

        // Remove first node and free it
        struct StackNode *save = front->next;
        front->next = front->next->next;
        free( save );

        // ...
    }

    return NULL;
}
```

Queue code

```
// Dequeue node at front of queue and get value
// Returns the new back of the queue or NULL on failure
struct QueueNode * dequeue( struct QueueNode *front, struct QueueNode *back,
                int *value ) {
    if ( front->next ) {
        // Get value from first node
        *value = front->next->value;

        // Remove first node and free it
        struct StackNode *save = front->next;
        front->next = front->next->next;
        free( save );

        // Update back pointer if necessary
        if ( front->next == NULL ) back = front;

        return back;
    }

    return NULL;
}
```

Queue usage example

```
int main() {
    // Make front node
    struct QueueNode *front = makeQueueNode( 0 );

    // Set back pointer to front (empty queue)
    struct QueueNode *back = front;

    // Queue up numbers
    for ( int i = 0; i < 10; i++ ) {
        struct QueueNode *node =
            makeQueueNode( i );

        // ... ?
    }
```

Queue usage example

```
int main() {
    // Make front node
    struct QueuekNode *front = makeQueueNode( 0 );

    // Set back pointer to front (empty queue)
    struct QueuekNode *back = front;

    // Queue up numbers
    for ( int i = 0; i < 10; i++ ) {
        struct QueueNode *node =
            makeQueueNode( i );

        back = enqueue( back, node );
    }
```

Queue usage example

```
// Print and empty the queue
// ... ?
```

Queue usage example

```
// Print and empty the queue
int value;
while ( back = dequeue( front, back, &value ) ) {
    printf( "%d ",value );
}
// correct issue that back is now NULL
back = front;
```

Queue usage example

```
    // Delete the front node
    free( front );

    return 0;
}
```

Next time

- Homework due Friday 11:55 PM
- Next week: Review for final exam