Arrays and Linked Lists

Stacks and Queues

Outline

- Abstract Data Types
- Stacks
- Queues
- Priority Queues and Deques

Postfix

And Stacks

Reverse Polish Notation

- Reverse Polish Notation (RPN)
  - Also known as postfix notation
  - A mathematical notation
    - Where every operator follows its operands
  - Invented by Jan Łukasiewicz in 1920
- Example
  - Infix: 5 + ((1 + 2) * 4) − 3
  - RPN: 5 1 2 + 4 * + 3 −

RPN Example

To evaluate a postfix expression read it from left to right

5 1 2 + 4 * + 3 −

store 5

store 1

store 2

Apply + to the last two operands

2
1
5

RPN Example

To evaluate a postfix expression read it from left to right

5 1 2 + 4 * + 3 −

store 5 | store 1 | store 2

Apply + to the last two operands

store 3 | store 4

Apply * to the last two operands

4
3
5

RPN Example

To evaluate a postfix expression read it from left to right

5 1 2 + 4 * + 3 −

store 5

store 1

store 2

Apply + to the last two operands

store 3

store 4

Apply * to the last two operands

store 12

Apply + to the last two operands

12

5

RPN Example

To evaluate a postfix expression read it from left to right

5 1 2 + 4 * + 3 −

store 5

store 1

store 2

Apply + to the last two operands

store 3

store 4

Apply * to the last two operands

store 12

Apply + to the last two operands

store 17

store 3

Apply - to the last two operands

3
17

RPN Example

To evaluate a postfix expression read it from left to right

5 1 2 + 4 * + 3 −

store 5 | store 1 | store 2

Apply + to the last two operands

store 3 | store 4

Apply * to the last two operands

store 12

Apply + to the last two operands

store 17 | store 3

Apply - to the last two operands

store 14

14

retrieve answer

Calculating a Postfix Expression

- for each input symbol
  - if symbol is operand
    - store(operand)
  - if symbol is operator
    - LHS = remove()
    - RHS = remove()
    - result = LHS operator RHS
    - store(result)
- result = remove()

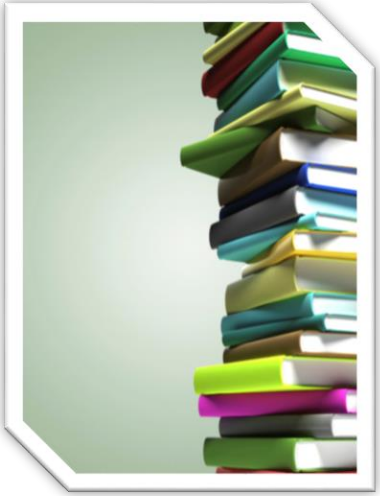
Describing a Data Structure

- What are the storage properties of the data structure that was used?
  - Specifically how are items stored and removed?
- Note that items are never inserted between existing items
  - The last item to be entered is the first item to be removed
  - Known as LIFO (Last In First Out)
- This data structure is referred to as a *stack*

Stack

Stacks

- A stack only allows items to be inserted and removed at *one end*
  - We call this end the *top* of the stack
  - The other end is called the bottom
- Access to other items in the stack is not allowed



Postfix and Stacks

- A stack is a natural choice to store data for postfix notation
  - Operands are stored at the top of the stack
  - And removed from the top of the stack
- Notice that we have not (yet) discussed how a stack should be implemented
  - Just *what* it does
- An example of an *Abstract Data Type*

Abstract Data Types

Abstract Data Types

- A collection of data
  - Describes ***what*** data is stored but ***not how*** it is stored
- Set of operations on the data
  - Describes precisely ***what*** effect the operations have on the data but
  - Does ***not*** specify ***how*** operations are carried out
- An ADT is not an actual (*concrete*) structure

Concrete Data Type

- The term *concrete data type* is usually used in contrast with an ADT
- An ADT is a collection of data and a set of operations on the data
- A concrete data type is an *implementation* of an ADT using a *data structure*
  - A construct that is defined in a programming language to store a collection of data
    - Such as an array

ADT Operators

- Mutators
- Accessors
- Constructors
- Other

ADT Operators

- Mutators
  - Often known as *setters*
  - Operations that change the contents of an ADT, usually subdivided into
    - Adding data to a data collection and
    - Removing data from a collection
  - Different ADTs allow data to be added and removed at different locations
- Accessors
- Constructors
- Other

ADT Operators

- Mutators
- Accessors
  - Often known as *getters*
  - Retrieve data from the collection
    - e.g. the item at the top of the stack
  - Ask questions about the data collection
    - Is it full?
    - How many items are stored?
    - …
- Constructors
- Other

ADT Operators

- Mutators
- Accessors
- Constructors
  - Constructors are used to create an ADT
    - Either empty
    - Or initialized with data
- Other

Implementation Hiding

- Information related to how storage is implemented should be hidden
- An ADT's operations can be used in the design of other modules or applications
  - Other modules do not need to know the *implementation* of the ADT operations
  - Which allows implementation of operations to be changed without affecting other modules

Specification of ADT Operations

- Operations should be specified in detail without discussing implementation issues
  - In C++ a class to implement an ADT is divided into header (*.h*) and implementation (*.cpp*) files
- The header file contains the class definition which only includes method prototypes
  - Occasionally there are exceptions to this
- The implementation file contains the definitions of the methods

The Call Stack

Another Stack Example

Functions

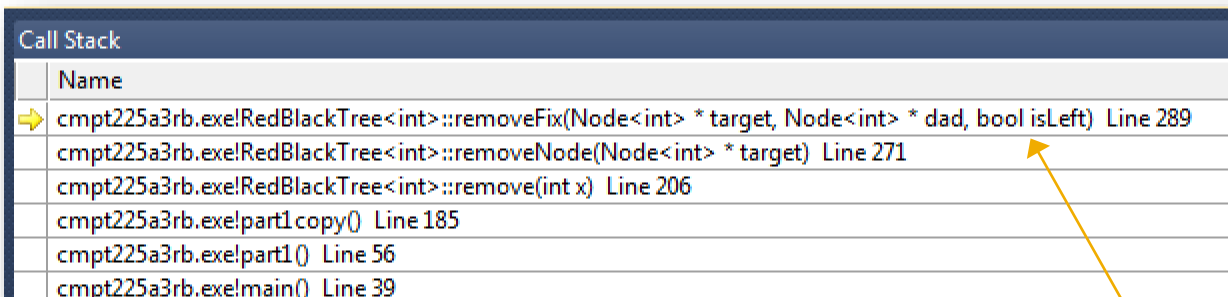
- Programs typically involve more than one function call and contain
  - A *main* function
  - Which calls other functions as required
- Each function requires space in main memory for its variables and parameters
  - This space must be allocated and de-allocated in some organized way

Organizing Function Calls

- Most programming languages use a *call stack* to implement function calling
  - When a method is called, its line number and other data are *pushed* onto the call stack
  - When a method terminates, it is *popped* from the call stack
  - Execution restarts at the indicated line number in the method currently at the top of the stack
- Stack memory is allocated and de-allocated without explicit instructions from a programmer
  - And is therefore referred to as *automatic* storage

The Call Stack

The call stack – from the Visual Studio Debug window



Bottom of the stack: least recently called method

Top of the stack: most recently called method.

Stack Frames

- Information stored on the call stack about a function is itself stored in a *stack frame*
  - Sometimes referred to as an *activation record*
- Stack frames store
  - The arguments passed to the function
  - The return address back to the calling function
  - Space for the function's local variables

Call Stack and Memory

- When a function is called space is allocated for it on the call stack
  - This space is allocated *sequentially*
- Once a function has run the space it used on the call stack is de-allocated
  - Allowing it to be re-used
- Execution returns to the previous function
  - Which is now at the top of the call stack

Call Stack and Functions

```
int main(){
	int n = 2;
	double arr[] = {5,17};
	squareArray(arr, n);
	int sum = sumArray(arr, n);
	cout << sum << endl;
	return 0;
}
```

```
 void squareArray(int a[], n){
	for(int i=0; i < n; i++){
		int x = a[i];
		a[i] = power(x, 2);
	}
}
```

```
double power(double x, int exp){
	double result = 1;
	for(int i=1; i <= exp; i++){
		result *= x;
	}
	return result;
}
```

```
double sumArray(double a[], int n){
	double sum = 0;
	for(int i=0; i < n; i++){
		sum += a[i];
	}
	return sum;
}
```

power	
x	5
exp	2
result	1
i	1
squareArray	
a	aff02b5c
n	2
i	0
x	5
main	
n	2
arr	5 17
sum	-

call stack

Call Stack and Functions

```
int main(){
	int n = 2;
	double arr[] = {5,17};
	squareArray(arr, n);
	int sum = sumArray(arr, n);
	cout << sum << endl;
	return 0;
}
```

```
 void squareArray(int a[], n){
	for(int i=0; i < n; i++){
		int x = a[i];
		a[i] = power(x, 2);
	}
}
```

```
double power(double x, int exp){
	double result = 1;
	for(int i=1; i <= exp; i++){
		result *= x;
	}
	return result;
}
```

```
double sumArray(double a[], int n){
	double sum = 0;
	for(int i=0; i < n; i++){
		sum += a[i];
	}
	return sum;
}
```

power	
x	5
exp	2
result	25
i	3
squareArray	
a	aff02b5c
n	2
i	0
x	5
main	
n	2
arr	25 17
sum	-

call stack

Call Stack and Functions

```
int main(){
      int n = 2;
      double arr[] = {5,17};
      squareArray(arr, n);
      int sum = sumArray(arr, n);
      cout << sum << endl;
      return 0;
}
```

```
double sumArray(double a[], int n){
      double sum = 0;
      for(int i=0; i < n; i++){
            sum += a[i];
      }
      return sum;
}
```

```
 void squareArray(int a[], n){
      for(int i=0; i < n; i++){
            int x = a[i];
            a[i] = power(x, 2);
      }
}
```

```
double power(double x, int exp){
      double result = 1;
      for(int i=1; i <= exp; i++){
            result *= x;
      }
      return result;
}
```

squareArray	
a	affo2b5c
n	2
i	1
x	17
main	
n	2
arr	25 17
sum	-

call stack

Call Stack and Functions

```
int main(){
	int n = 2;
	double arr[] = {5,17};
	squareArray(arr, n);
	int sum = sumArray(arr, n);
	cout << sum << endl;
	return 0;
}
```

```
 void squareArray(int a[], n){
	for(int i=0; i < n; i++){
		int x = a[i];
		a[i] = power(x, 2);
	}
}
```

```
double power(double x, int exp){
	double result = 1;
	for(int i=1; i <= exp; i++){
		result *= x;
	}
	return result;
}
```

```
double sumArray(double a[], int n){
	double sum = 0;
	for(int i=0; i < n; i++){
		sum += a[i];
	}
	return sum;
}
```

power	
x	17
exp	2
result	289
i	3
squareArray	
a	aff02b5c
n	2
i	2
x	17
main	
n	2
arr	25 17
sum	-

call stack

Call Stack and Functions

```
int main(){
	int n = 2;
	double arr[] = {5,17};
	squareArray(arr, n);
	int sum = sumArray(arr, n);
	cout << sum << endl;
	return 0;
}
```

```
 void squareArray(int a[], n){
	for(int i=0; i < n; i++){
		int x = a[i];
		a[i] = power(x, 2);
	}
}
```

```
double power(double x, int exp){
	double result = 1;
	for(int i=1; i <= exp; i++){
		result *= x;
	}
	return result;
}
```

```
double sumArray(double a[], int n){
	double sum = 0;
	for(int i=0; i < n; i++){
		sum += a[i];
	}
	return sum;
}
```

sumArray	
a	aff02b5c
n	2
sum	314
i	2
main	
n	2
arr	25 289
sum	314

call stack

Returning Values

- In the example, functions returned values assigned to variables in other functions
  - They did not affect the *amount of memory* required by previously called functions
  - That is, functions *below* them on the call stack
- Stack memory is sequentially allocated
  - It is not possible to increase memory assigned to a function previously pushed onto the stack

Implementing a Stack

With an Array

Stack Operations

- A stack should implement at least the first two of these operations
  - *push* – insert an item at the top of the stack
  - *pop* – remove and return the top item
  - *peek* – return the top item
- ADT operations should be performed efficiently
  - The definition of efficiency varies from ADT to ADT
  - The order of the items in a stack is based solely on the order in which they arrive

A Design Note

- Assume that we plan on using a stack that will store integers and have these methods
  - `void push(int)`
  - `int pop()`
- We can design other modules that use these methods
  - Without having to know anything about how they, or the stack itself, are implemented

Classes

- We will use classes to encapsulate stacks
  - Encapsulate – enclose in
- A class is a programming construct that contains
  - Data for the class, and
  - Operations of the class
  - More about classes later …

Implementing a Stack

- The stack ADT can be implemented using a variety of data structures, e.g.
  - Arrays
  - Linked Lists
- Both implementations must implement all the stack operations
  - In constant time
    - Time that is independent of the number of items in the stack

Array Implementation

- Use an array to implement a stack
- We need to keep track of the index that represents the top of the stack
  - When we insert an item increment this index
  - When we delete an item decrement this index
- Insertion or deletion time is independent of the number of items in the stack

Array Stack Example

6	1	7			
0	1	2	3	4	5

index of **top** is current size – 1

```
Stack st();
st.push(6); //top = 0
st.push(1); //top = 1
st.push(7); //top = 2
st.push(8); //top = 3
st.push(13); //top = 4
st.pop(); //top = 3
st.pop(); //top = 2
```

Array Implementation Summary

- Easy to implement a stack with an array
  - And *push* and *pop* can be performed in constant time
- Once the array is full
  - No new values can be inserted or
  - A new, larger, array has to be created
    - And the existing items copied to this new array
    - Known as a dynamic array

Array Review

Arrays

- Arrays contain identically typed values
  - These values are stored sequentially in main memory
- Values are stored at specific numbered positions in the array called indexes
  - The first value is stored at index 0, the second at index 1, the *i*th at index *i*-1, and so on
  - The last item is stored at position *n*-1, assuming that the array is of size *n*
  - Referred to as zero-based indexing

Array Indexing

- `int arr[] = {3,7,6,8,1,7,2};`
  - Creates an integer array with 7 elements
- To access an element refer to the array name and the index of that element
  - `int x = arr[3];` assigns the value of the fourth array element (8) to x
  - `arr[5] = 11;` changes the sixth element of the array from 7 to 11
  - `arr[7] = 3;` results in an error because the index is out of bounds

In C++ the error is an unexpected run-time or logic error

An IDE may raise a debug error after termination

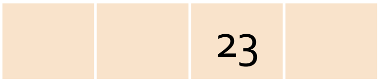
index	value
0	3
1	7
2	6
3	8
4	1
5	11
6	2

Arrays and Main Memory

```
int grade[4];
grade[2] = 23;
```

Declares an array variable of size 4

Assigns 23 to the third element of *grade*



The array is shown as not storing any values – although this isn't really the case

grade is a *constant pointer* to the array and stores the address of the array

But how does the program know where `grade[2]` is?

Memory Addresses

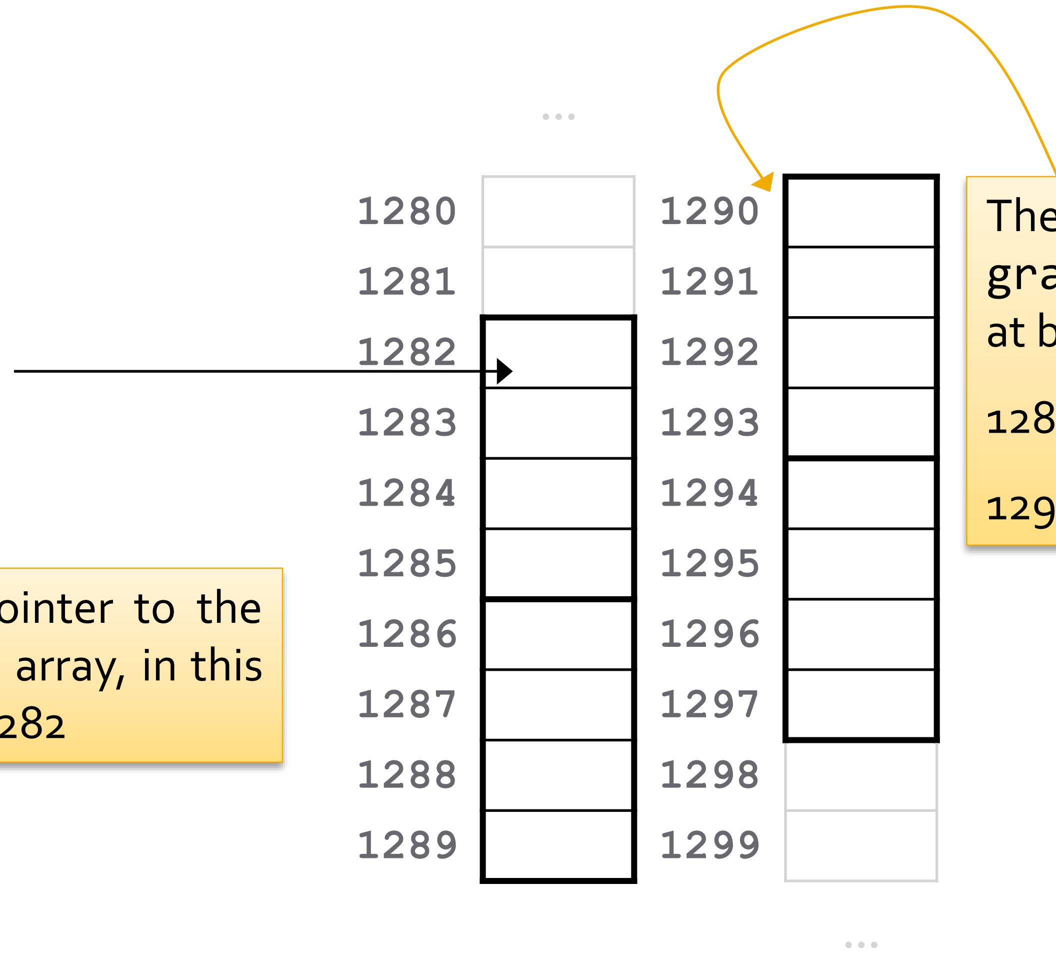
- Access to array elements is very fast
- An array variable refers to the array
  - Storing the main memory address of the first element
  - The address is stored as number, with each address referring to one byte of memory
    - Address 0 would be the first byte
    - Address 1 would be the second byte
    - Address 20786 would be the twenty thousand, seven hundred and eighty seventh byte
    - ...

Offset Calculations

- Consider `grade[2] = 23;`
  - How do we find this element in the array?
- Consider what we know
  - The *address* of the first array element
  - The *type* of the values stored in the array
    - And therefore the size of each of the array elements
  - The *index* of the element to be accessed
- We can therefore calculate the address of the element to be accessed, which equals
  - address of first element + (index * type size)

Offset Example

grade



Stores a pointer to the start of the array, in this case byte 1282

The integer stored at grade[2] is located at byte:

1282 + 2 * 4 =

1290

Passing Arrays to Functions

- Array variables are pointers
  - An array variable argument to a function passes the *address* of the array
    - And not a copy of the array
- Changes made to the array by a function are made to the original (one-and-only) array
  - If this is not desired, a copy of the array should be made within the function

Array Positions

- What if array *positions* carry meaning?
  - An array that is sorted by name, or grade or some other value
  - Or an array where the position corresponds to a position of some entity in the world
- The ordering should be maintained when elements are inserted or removed

Ordered Array Problems

- When an item is inserted either
  - Write over the element at the given index or
  - Move the element, *and all elements after it*, up one position
- When an item is removed either
  - Leave *gaps* in the array, i.e. array elements that don't represent values or
  - Move all the values after the removed value down one index

Arrays are Static

- The size of an array must be specified when it is created
  - And cannot then be changed
- If the array is full, values cannot be inserted
  - There are, time consuming, ways around this
  - To avoid this problem we can make arrays much larger than they are needed
  - However this wastes space

Array Summary

- Good things about arrays
  - Fast, random access, of elements using a simple offset calculation
  - Very storage space efficient, as little main memory is required other than the data itself
  - Easy to use
- Bad things about arrays
  - Slow deletion and insertion for ordered arrays
  - Size must be known when the array is created
    - Or possibly beforehand
    - An array is either full or contains unused elements

Arrays in C++

Another Review

Declaring (Static) Arrays

- Arrays are declared just like single variables except that the name is followed by []s
- The []s should contain the size of the array which must be a constant or literal integer
  - `int age[100];`
  - `const int DAYS = 365;`
  - `double temperatures[DAYS];`

Initializing Arrays

- Arrays can be initialized
  - One element at a time
  - By using a for loop
  - Or by assigning the array values on the same line as the declaration
    - `int fib[] = {0,1,1,2,3,5,8,13};`
  - Note that the size does not have to be specified since it can be derived

Array Assignments

- A new array *cannot* be assigned to an existing array

```
int arr1[4];
int arr2[4];
…
arr1 = arr2; //can't do this!
arr1 = {1,3,5,7}; //… or this …
```

- Array *elements* can be assigned values

```
for(int i=0; i < 4; i++) {
  arr1[i] = arr2[i];
}
```

Array Parameters

- An array parameter looks just like an array variable
  - Except that the size is not specified
- C++ arrays do not have a size member
  - Or any members, since they are not classes
  - Therefore, it is common to pass functions the size of array parameters
- For example
  - `int sum(int arr[], int n)`

Array Arguments

- Array variables are passed to functions in the standard way
  - `sum(grades, 4);`

What's in an Array Variable

- An array variable records the address of the first element of the array
  - This address cannot be changed after the array has been declared
  - It is therefore a *constant pointer*
- This explains why existing array variables cannot be assigned new arrays
- And why arrays passed to functions may be changed by those functions

Memory in C++

- C++ gives programmers a lot of control over where variables are located in memory
- There are three classes of main memory
  - Static
  - Automatic
  - Dynamic
- Automatic memory is generally used to allocate space for variables declared inside functions
  - Unless those variables are specifically assigned to another class of storage

Arrays and Memory in C++

- Arrays are allocated space in automatic storage
  - At least as they have been discussed so far, and
  - Assuming that they were declared in a function
- Variables allocated space on the call stack are not permitted to change size
  - As stack memory is allocated in sequence and this could result other variables being over-written

Dynamic Memory

- What happens if we want to determine how much memory to allocate at *run time*?
  - Stack memory size is determined at compile time so it would need to be allocated somewhere else
  - Let's call *somewhere else* the *heap* or the *free store*
- We still need automatic variables that refer or point to the dynamically allocated memory
  - In C++ such variables are *pointers*

Variables in Dynamic Memory

- Create a variable to store an address
  - A pointer to the type of data to be stored
  - Addresses have a fixed size
  - If there is initially no address it should be assigned a special value (*NULL*)
- Create new data in dynamic memory
  - This may be done when needed (i.e. at run time)
- Assign the address of the data to the pointer
- This involves more a more complex management system than using automatic memory

Creating an Array in Dynamic Memory

- Arrays created in dynamic memory are indexed just like other arrays

```
int* p_arr = new int[100];
for (int i=0; i < 100; ++i){
  p_arr[i] = i+1;
}
```

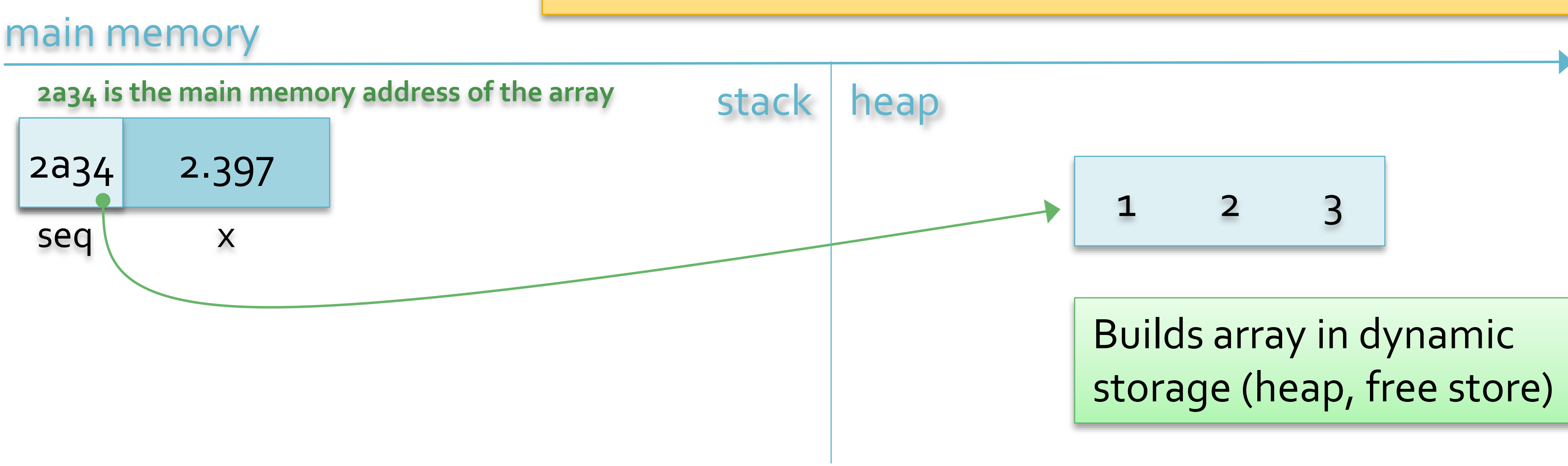
- Pointers to arrays can be assigned new arrays

```
delete[] p_arr; //release memory
p_arr = new int[1000000];
```

A Dynamic Array

```
int* seq = NULL;
double x = 2.397;
seq = sequence(1, 3);
```

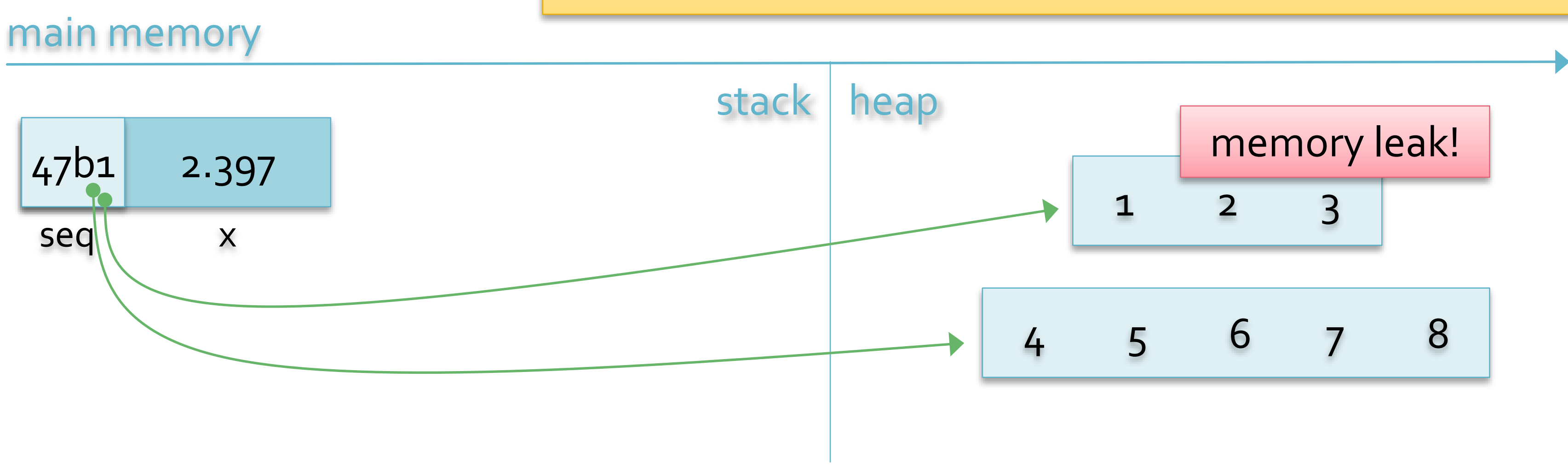
```
// Returns  pointer to array {start, start+1, … start+n-1}
int* sequence(int start, int n){
		int* result =  new int[n];
		for(int i=0; i < n; i++) {
			result[i] = start + i;
		}
		return result;
}
```



A Dynamic Array

```
int seq = NULL;
double x = 2.397;
seq = sequence(1, 3);
seq = sequence(4, 5);
```

```
// Returns  pointer to array {start, start+1, … start+n-1}
int* sequence(int start, int n){
		int* result =  new int[n];
		for(int i=0; i < n; i++) {
			result[i] = start + i;
		}
		return result;
}
```



Releasing Dynamic Memory

- When a function call is complete its stack memory is released and can be re-used
- Dynamic memory should also be released
  - Failing to do so results in a *memory leak*
- It is sometimes not easy to determine when dynamic memory should be released
  - Data might be referred to by more than one pointer
    - Memory should only be released when it is no longer referenced by *any* pointer

Dynamic vs Static

- When should a data object be created in dynamic memory?
  - When the object is required to change size, or
  - If it is not known if the object will be required
- Languages have different approaches to using static and dynamic memory
  - In C++ the programmer can choose whether to assign data to static or dynamic memory

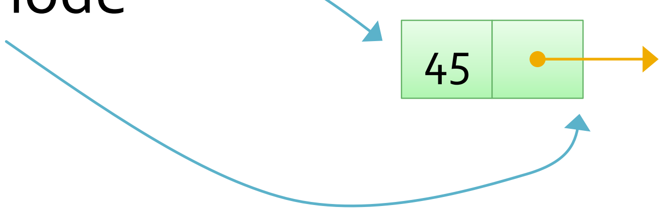
Linked Lists

A Dream Data Structure

- It would be nice to have a data structure that is
  - Dynamic
  - Does fast insertions/deletions in the middle
- We can achieve this using linked lists ...

Nodes

- A linked list is a dynamic data structure that consists of nodes linked together
- A *node* is a data structure that contains
  - data
  - the location of the next node

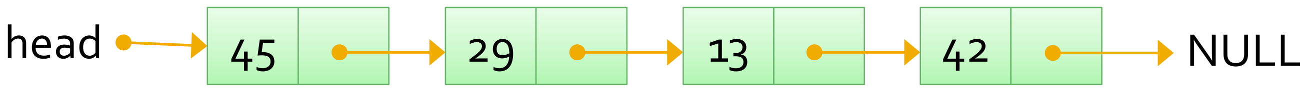


Node Pointers

- A node contains the address of the next node in the list
  - In C++ this is recorded as a pointer to a node
- Nodes are created in dynamic memory
  - And their memory locations are not in sequence
- The data attribute of a node varies depending on what the node is intended to store

Linked Lists

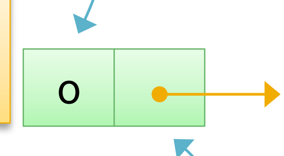
- A linked list is a *chain* of nodes where each node stores the address of the next node



Linked Lists

```
class Node {
public:
    int data;
    Node* next;
}
```

Nodes point to other nodes, so the pointer must be of type Node

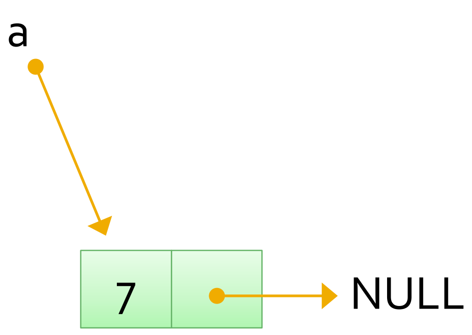


Building a Linked List

```
Node* a = new Node(7, null);
```

Assumes a constructor in the Node class

```
Node(int value, Node* nd){
    data = value;
    next = nd;
}
```

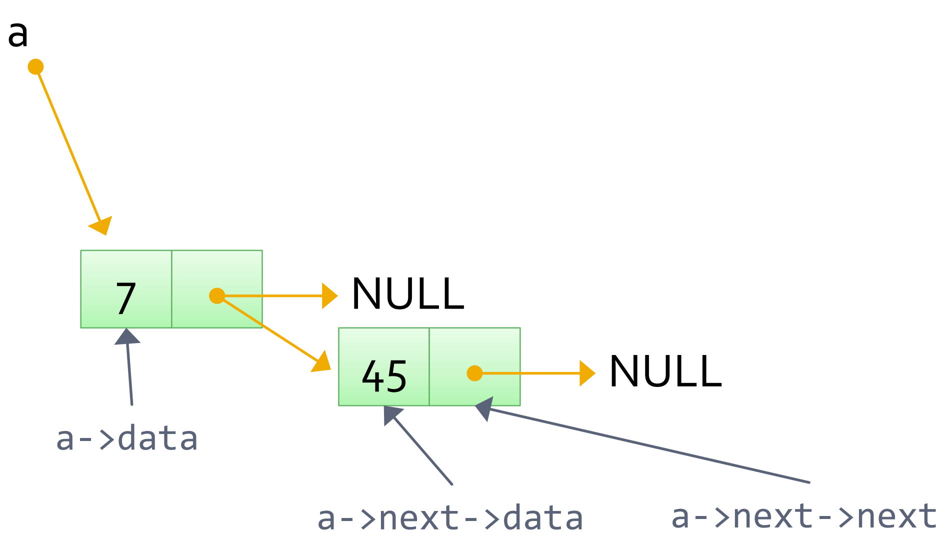


Building a Linked List

```
Node* a = new Node(7, null);
a->next = new Node(45, null);
```

Assumes a constructor in the Node class

```
Node(int value, Node* nd){
    data = value;
    next = nd;
}
```

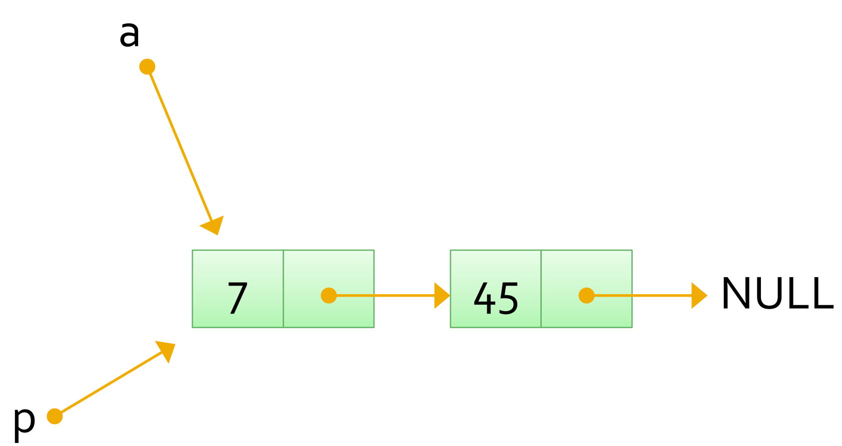


Traversing a Linked List

```
Node* a = new Node(7, null);
a->next = new Node(45, null);
Node* p = a;
```

Assumes a constructor in the Node class

```
Node(int value, Node* nd){
    data = value;
    next = nd;
}
```

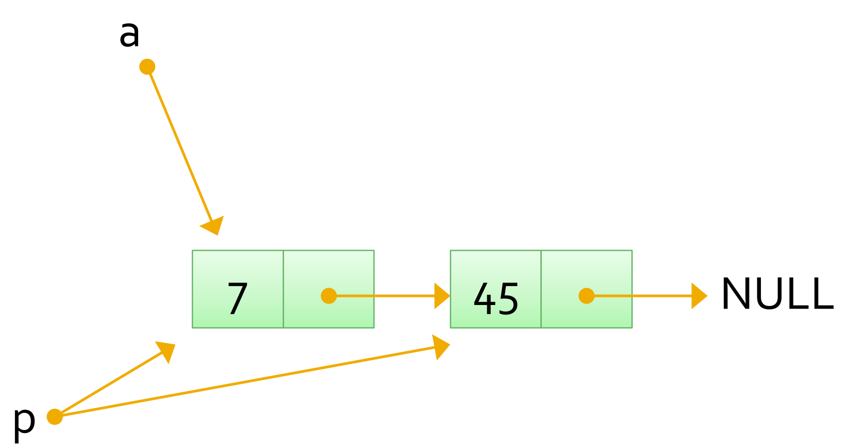


Traversing a Linked List

```
Node* a = new Node(7, null);
a->next = new Node(45, null);
Node* p = a;
p = p->next;  // go to next node
```

Assumes a constructor in the Node class

```
Node(int value, Node* nd){
    data = value;
    next = nd;
}
```

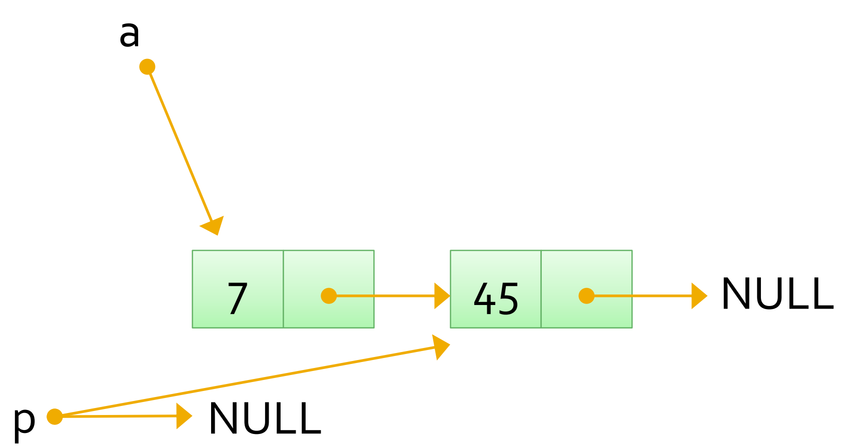


Traversing a Linked List

```
Node* a = new Node(7, null);
a->next = new Node(45, null);
Node* p = a;
p = p->next;  // go to next node
p = p->next;  // go to next node
```

Assumes a constructor in the Node class

```
Node(int value, Node* nd){
    data = value;
    next = nd;
}
```



In practice insertion and traversal would be methods of a linked list class

Encapsulating Linked Lists

- The previous example showed a list built out of nodes
- In practice a linked list is encapsulated in its own class
  - Allowing new nodes to be easily inserted and removed as desired
  - The linked list class has a pointer to the (node at the) head of the list
- Implementations of linked lists vary

Implementing a Stack

With a Linked List

Stack: Linked List

- Nodes should be inserted and removed at the head of the list
  - New nodes are pushed onto the front of the list, so that they become the top of the stack
  - Nodes are popped from the front of the list
- Straight-forward linked list implementation
  - Both *push* and *pop* affect the front of the list
    - There is therefore no need for either algorithm to traverse the entire list

Linked List Implementation

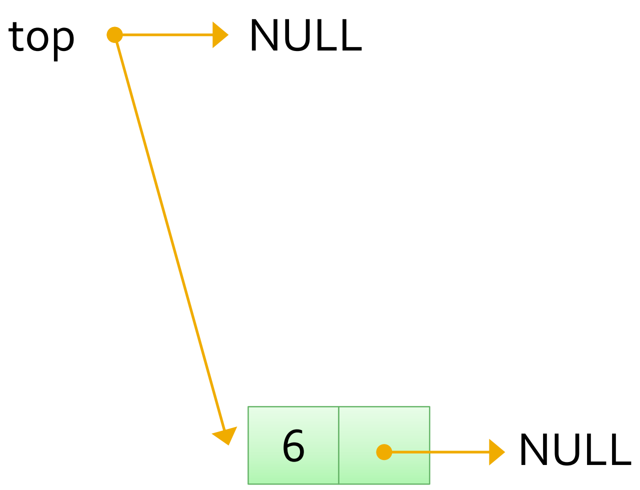
```
void push(int x){
// Make a new node whose next pointer is the
// existing list
      Node* newNode = new Node(x, top);
      top = newNode; //head points to new node
}
```

```
int pop(){
// Return the value at the head of the list
      int temp = top->data;
      Node* p = top;
      top = top->next;
      delete p; // deallocate old head
      return temp;
}
```

What happens if the list to be popped is empty?

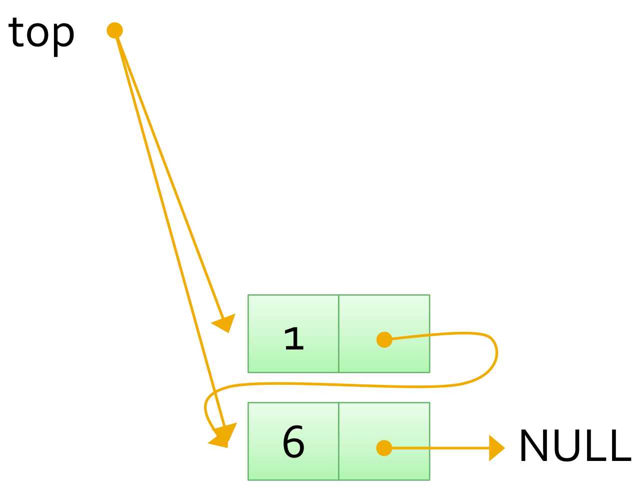
List Stack Example

```
Stack st;
st.push(6);
```



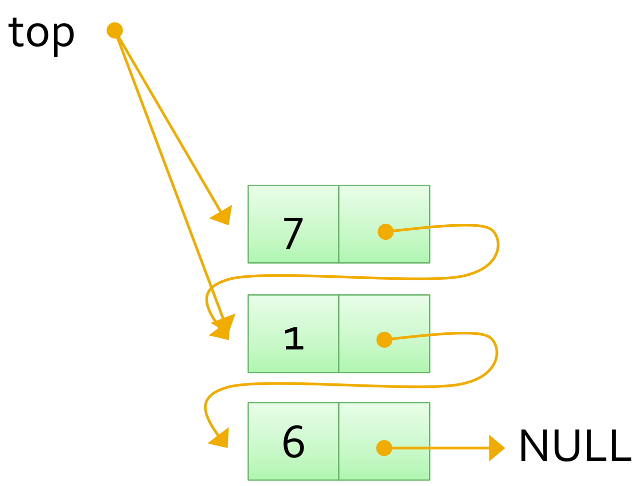
List Stack Example

```
Stack st;
st.push(6);
st.push(1);
```



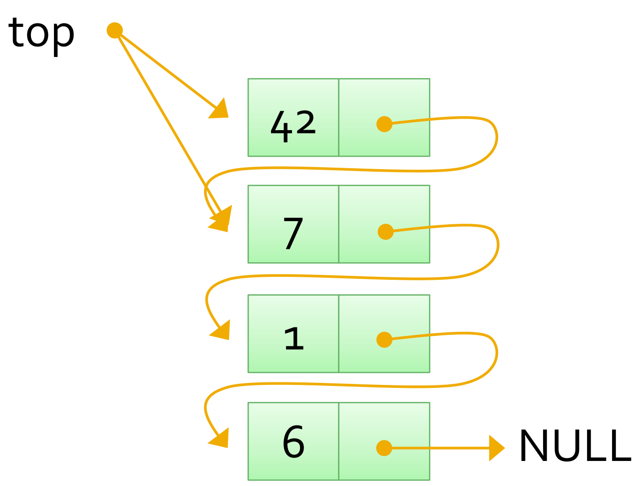
List Stack Example

```
Stack st;
st.push(6);
st.push(1);
st.push(7);
```

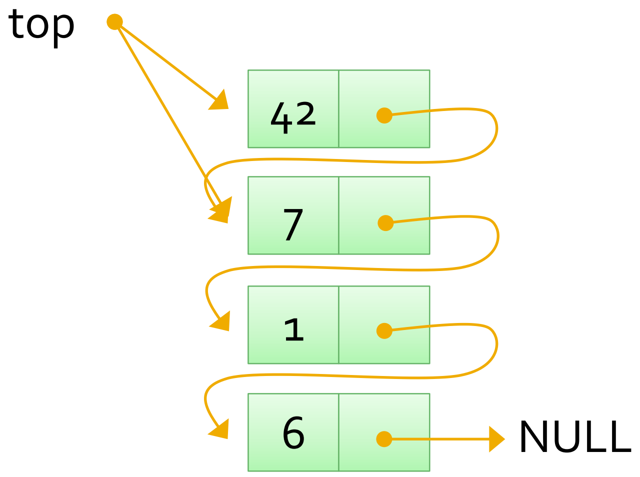


List Stack Example

```
Stack st;
st.push(6);
st.push(1);
st.push(7);
st.push(42);
```

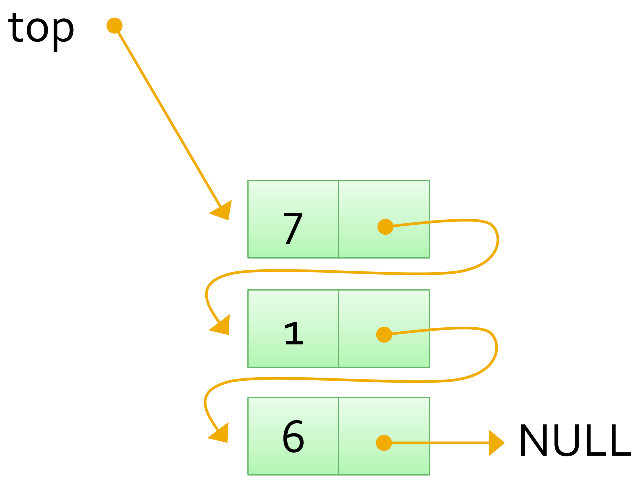


List Stack Example




```
Stack st;
st.push(6);
st.push(1);
st.push(7);
st.push(42);
st.pop();
```

List Stack Example




```
Stack st;
st.push(6);
st.push(1);
st.push(7);
st.push(42);
st.pop();
```

Postfix Example

Visual Studio Presentation

Queues

Print Queues

- Assume that we want to store data for a print queue for a student printer
  - Student ID
  - Time
  - File name
- The printer is to be assigned to file in the order in which they are received
  - A *fair* algorithm

Classes for Print Queues

- To maintain the print queue we would require at least two classes
  - Request class
  - Collection class to store requests
- The collection class should support the desired behaviour
  - FIFO (First In First Out)
  - The ADT is a *queue*

Queues

- In a queue items are inserted at the back and removed from the front
  - As an aside *queue* is just the British (i.e. correct☺) word for a line (or line-up)
- Queues are **FIFO** (First In First Out) data structures – *fair* data structures

What Can You Use a Queue For?

- Server requests
  - Instant messaging servers queue up incoming messages
  - Database requests
    - Why might this be a bad idea for all such requests?
- Print queues
- Operating systems often use queues to schedule CPU jobs
- Various algorithm implementations

Queue Operations

- A queue should implement at least the first two of these operations:
  - *insert* – insert item at the back of the queue
  - *remove* – remove an item from the front
  - *peek* – return the item at the front of the queue without removing it
- Like stacks, it is assumed that these operations will be implemented efficiently
  - That is, in constant time

Implementing a Queue

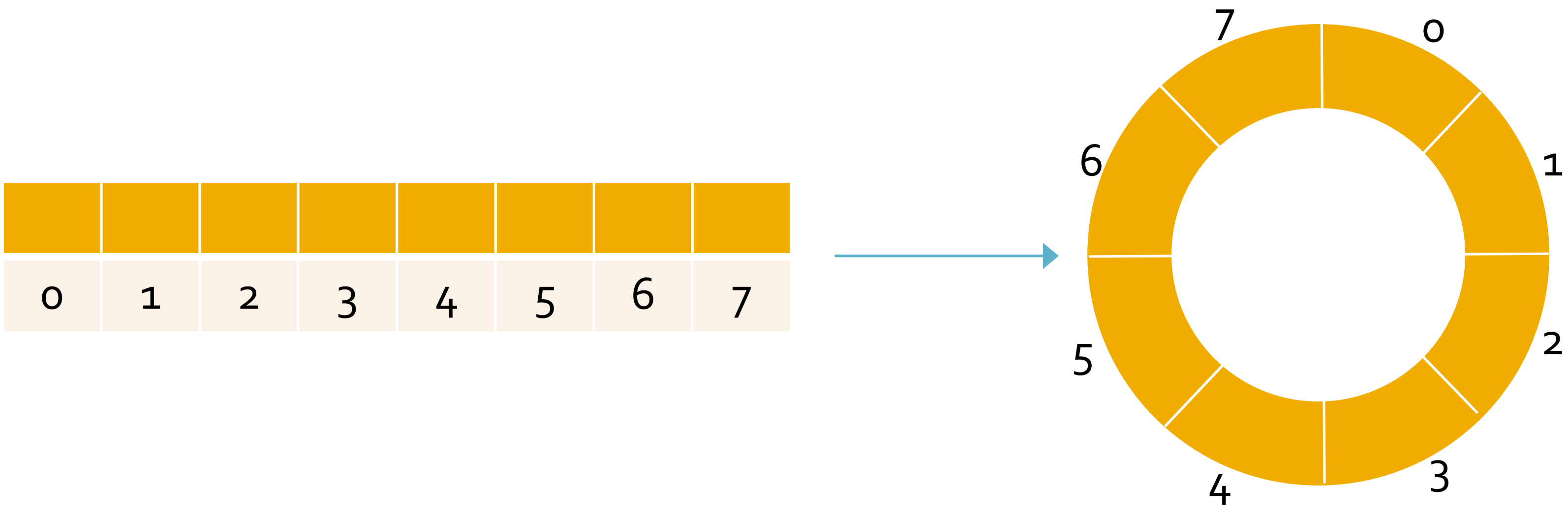
with an Array

Array Implementation

- Consider using an array as the underlying structure for a queue, we could
  - Make the back of the queue the current size of the array, much like the stack implementation
  - Initially make the front of the queue index 0
  - Inserting an item is easy
- What happens when items are removed?
  - Either move all remaining items down – **slow**
  - Or increment the front index – **wastes space**

Circular Arrays

- **Neat trick**: use a *circular array* to insert and remove items from a queue in constant time
- The idea of a circular array is that the end of the array "wraps around" to the start of the array



The mod Operator

- The *mod* operator (%) calculates remainders:
  - `1%5 = 1, 2%5 = 2, 5%5 = 0, 8%5 = 3`
- The *mod* operator can be used to calculate the front and back positions in a circular array
  - Thereby avoiding comparisons to the array size
  - The back of the queue is:
    - `(front + num) % queue.length`
    - where *num* is the number of items in the queue
  - After removing an item the front of the queue is:
    - `(front + 1) % queue.length;`

Array Stack Example

42		3	13	7	11
0	1	2	3	4	5

```
Queue q();
q.insert(6); //front = 0
q.insert(4); //front = 0
q.insert(3); //front = 0
q.insert(13); //front = 0
q.insert(7); //front = 0
q.remove(); //front = 1
q.insert(11); //front = 1
q.remove(); //front = 2
q.insert(42); //front = 2
```

Implementing a Queue

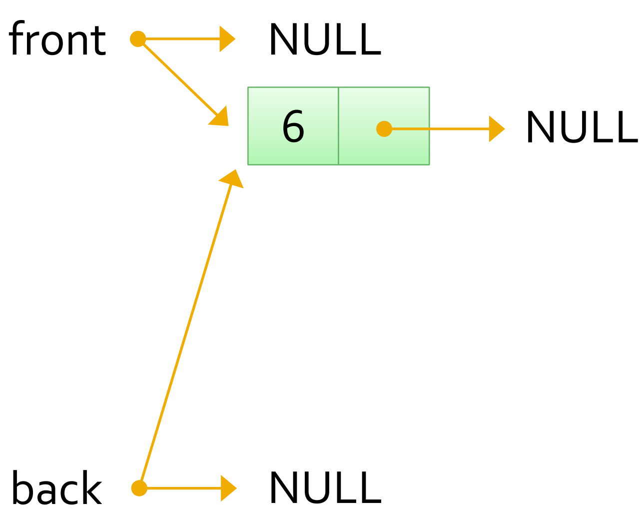
With a Linked List

Linked List Implementation

- Removing items from the front of the queue is straightforward
- Items should be inserted at the back of the queue in constant time
  - So we must avoid traversing through the list
  - Use a second node pointer to keep track of the node at the back of the queue
    - Requires a little extra administration

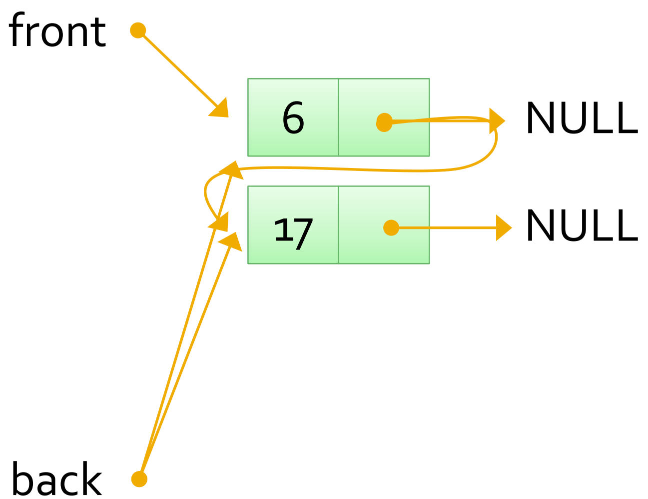
List Queue Example

```
Queue q;
q.insert(6);
```



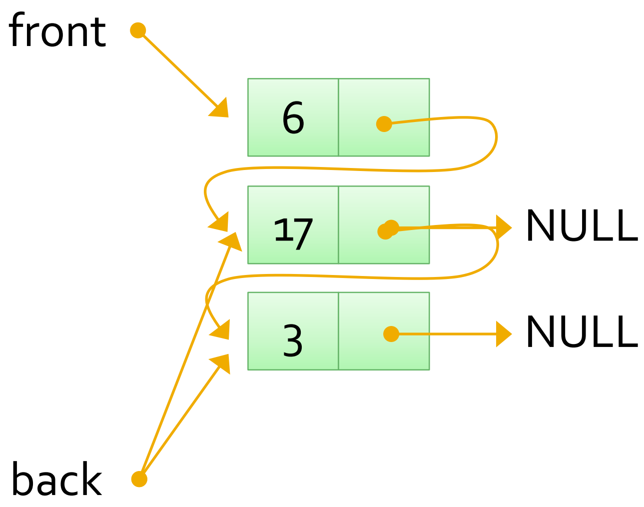
List Queue Example

```
Queue q;
q.insert(6);
q.insert(17);
```

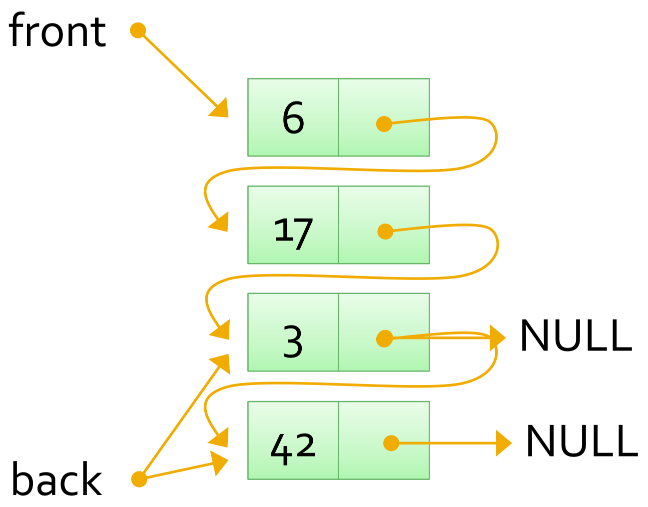


List Queue Example

```
Queue q;
q.insert(6);
q.insert(17);
q.insert(3);
```

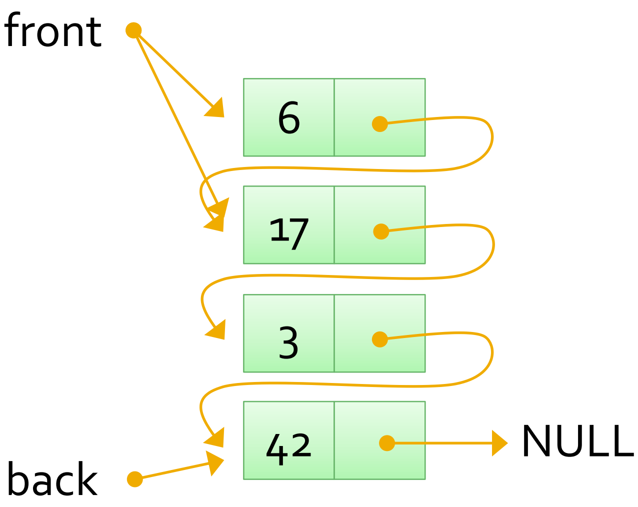


List Queue Example



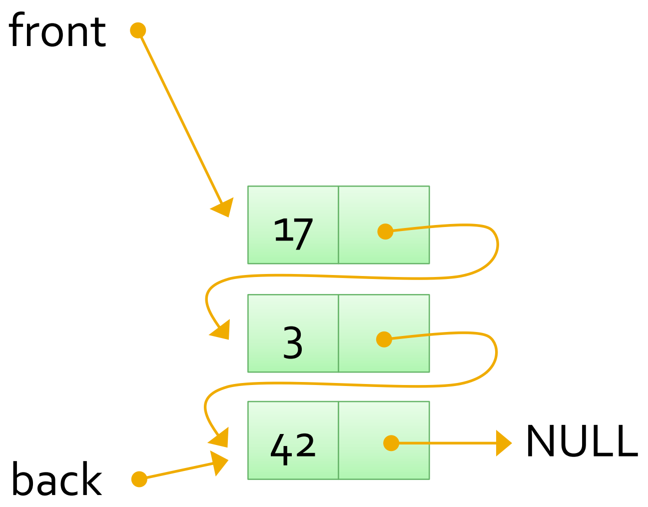

```
Queue q;
q.insert(6);
q.insert(17);
q.insert(3);
q.insert(42);
```

List Queue Example




```
Queue q;
q.insert(6);
q.insert(17);
q.insert(3);
q.insert(42);
q.remove();
```

List Queue Example




```
Queue q;
q.insert(6);
q.insert(17);
q.insert(3);
q.insert(42);
q.remove();
```

Other Simple Data Structures

Deques

- A deque is a double ended queue
  - That allows items to be inserted and removed from either end
- Deque implementations
  - Circular array, similar to the queue implementation
  - Linked List
    - Singly linked list implementations are not efficient

Priority Queues

- Items in a priority queue are given a priority value
  - Which could be numerical or something else
- The highest priority item is removed first
- Uses include
  - System requests
  - Data structure to support Dijkstra's Algorithm

Priority Queue Problem

- Can items be inserted and removed *efficiently* from a priority queue?
  - Using an array, or
  - Using a linked list?
- Note that items are not removed based on the order in which they are inserted
- We will return to priority queues later in the course

Template Example

Visual Studio Presentation