Lecture 13:
Abstract Data Types / Stacks
Plan for the next few weeks

LEC 11
LEC 12
LEC 13
LEC 14
LEC 15

Project 2G (HARD)

Project 2H (NORMAL)

Project 3A (HARD, depends on 2H)

Project 3B (HARD)

Project 1C (NORMAL, LAB)
Hank’s OH next week

• Monday 1030-1130: Zayd (PSC)
• Thursday 1230-130: Priya (PSC)
ANNOUNCEMENT:
Changing Grading Structure

• Project 3: 25%->20%
• Code Review: 0%->5%
Code Review

• Viet will meet with each of you for 20 minutes
• He will announce logistics for how to make an appointment with him
• 10 minutes: discuss your 2E submission
• 10 minutes: demonstrate some coding (sorting, like 2B)
Grading Rubric

• 5 points: attend and discuss your code
• 0 points: attend and sit mutely, refusing to participate
• 0 points: don’t attend

• (Joking aside, everyone will get 5/5 if they attend and make the most of the experience)
Why Do a Code Review?

• Get feedback from Viet
• On what you have submitted
• On how to develop code better

• Also: we are trying to understand how the course is working for you
  – Viet will ask some questions (previous experience with C, comfort level, etc.)
Quick Review
What is a Data Structure?

• Data structure definitions
  – Textbook: “a systematic way to organize data”
  – Wikipedia: “data organization, management and storage format that enables efficient access and modification”
Key Concept

• It organizes data
• It enables efficient access
  – What does efficient mean??
Asymptotic Analysis

• What happens when $N$ gets large?
  – ($N$ being the size of the input)

• Example:
  – # operations is $10000*N + 0.5*N^2$
  – When $N$ is really big, the 10000 and the 0.5 don’t matter
  – Hence $O(N^2)$
<table>
<thead>
<tr>
<th>$n$</th>
<th>$\log n$</th>
<th>$n$</th>
<th>$n \log n$</th>
<th>$n^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>8</td>
<td>24</td>
<td>64</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>32</td>
<td>160</td>
<td>1,024</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>64</td>
<td>384</td>
<td>4,096</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>128</td>
<td>896</td>
<td>16,384</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
<td>256</td>
<td>2,048</td>
<td>65,536</td>
</tr>
<tr>
<td>512</td>
<td>9</td>
<td>512</td>
<td>4,608</td>
<td>262,144</td>
</tr>
<tr>
<td>1,024</td>
<td>10</td>
<td>1,024</td>
<td>10,240</td>
<td>1,048,576</td>
</tr>
</tbody>
</table>
Making It Generic
Making it generic

- We just looked at an example: sorted arrays of students
- We could make this more generic:
  - Sorted arrays for *anything*
    - Doesn’t have to be students
  - Don’t even need to know we are using sorted arrays
    - Could just use “Search”
Making it generic

• We just looked at an example: sorted arrays of students

• We could make this more generic:
  – Sorted arrays for *anything*
    • Doesn’t have to be students
  – Don’t even need to know we are using sorted arrays
    • Could just use “Search”
void
Sort(void *array, int element_size, int num_elements, int (*Compare)(void *, void *))
{
    int i, j;
    void *tmp_location = malloc(element_size);
    for (i = 0 ; i < num_elements ; i++)
    {
        int smallest = i;
        for (j = i+1 ; j < num_elements ; j++)
        {
            int comp_value = Compare(array+j*element_size, array+smallest*element_size);
            if (comp_value < 0)
                smallest = j;
        }
        memcpy(tmp_location, array+i*element_size, element_size);
        memcpy(array+i*element_size, array+smallest*element_size, element_size);
        memcpy(array+smallest*element_size, tmp_location, element_size);
    }
    free(tmp_location);
}
Using Generic Sort

```c
int main()
{
    int i;
    char str[12];
    strcpy(str, "hello world");
    int A[10] = { 5, 3, 1, 8, 9, 0, 2, 4, 6, 7 };  
    Sort(str, 1, 11, CharCompare);
    Sort(A, sizeof(int), 10, IntCompare);
    printf("%s\n", str);
    for (i = 0 ; i < 10 ; i++) { printf("%d ", A[i]); }
    printf("\n");
}

int CharCompare(void *a, void *b)
{
    char *a_as_char = (char *) a;
    char *b_as_char = (char *) b;
    if (*a_as_char < *b_as_char)
        return -1;
    return 1;
}

int IntCompare(void *a, void *b)
{
    int *a_as_int = (int *) a;
    int *b_as_int = (int *) b;
    if (*a_as_int < *b_as_int)
        return -1;
    return 1;
}
```

Hanks-iMac:212 hank$ ./a.out
```
dehllllooorw
0 1 2 3 4 5 6 7 8 9
```
Making it generic

• We just looked at an example: sorted arrays of students

• We could make this more generic:
  – Sorted arrays for *anything*
    • Doesn’t have to be students
  – Don’t even need to know we are using sorted arrays
    • Could just use “Search”
“This will be Thursday’s lecture”

• Abstract data types
  – This about it from the perspective of the user
  – NOT the implementor
Data types

• Simple data types
  – float, double, int, char, unsigned char

• Complex data types
  – Defined with structs
  – image, rectangle/square/circle (2H)

• Abstract data type
  – Accomplished through function calls
  – You don’t have to know the details
Abstract data types

• Two pieces:
  – Define behavior (via function prototypes)
  – Define implementation (via functions)

• You can have more than one implementation for a given behavior
(Bad) Example: One Interface, Multiple Implementations

Bad example since we are talking about data types ... this is just a function that works on data
Better Example: Store/Fetch

• Abstract data type has two methods:
  – Store
    • Takes a “key” and a “value”
  – Fetch
    • Takes a ”key”, and returns a “value

• Example:
  – Key == UO ID
  – Value == student struct
Observation:
I don’t need to anything about ADT

How is it implemented?

How long does Store take?

How long does Fetch take?

Note: example too simple ... adt needs to be initialized.
To Motivate ADT, We Need Examples of Data Structures That Can Do Store/Fetch

• Two Examples:
  – Array
  – HashTable
One Data Structure for Store/Fetch:

- **Array**
  - Observation:
    - Not very generic (int key, Student value)
  - Why not pass in StoreFetchArray * instead of void *?
  - A: need this later

```c
#define MAX_STUDENTS 1000
typedef struct {
    int keys[MAX_STUDENTS];
    Student *values[MAX_STUDENTS];
    int curID;
} StoreFetchArray;

void Initialize(StoreFetchArray *arr) {
    arr->curID = 0;
}

void ArrayStore(void *a, int key, Student *v) {
    StoreFetchArray *arr = (StoreFetchArray *) a;
    if (arr->curID >= MAX_STUDENTS)
        exit(EXIT_FAILURE);
    arr->keys[arr->curID] = key;
    arr->values[arr->curID] = v;
    arr->curID++;
}

Student *ArrayFetch(void *a, int key) {
    StoreFetchArray *arr = (StoreFetchArray *) a;
    for (int i = 0; i < arr->curID; i++)
        if (arr->keys[i] == key)
            return arr->values[i];
    return NULL;
}
Complexity for Array

- Store: $O(1)$
- Fetch: $O(N)$
HashTable

• Idea:
  – Create a big array with keys and values
    • (Just like last slide!)
  – But: don’t insert starting from the beginning
  – Instead: insert into “random” places in the array
  – Not truly random, as it needs to be reproducible
  – Typical: take key and perform some math operation on it

This will be a whirlwind intro to hash tables. We will return to this idea later.
## HashTable

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>
**HashTable**

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>...34</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

- **Store:**
  UO_ID: 951001234
  Student: “xFF”

- **Idea:**
  Turn UO_ID into an index.
  In this case, %10.
  (much more complex ideas)
# HashTable

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>...34</td>
<td>-1</td>
<td>..66</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>NUL</td>
<td>xAF</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

Store:
UO_ID: 951003266
Student: “xAF”
### HashTable

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>...34</td>
<td>...44</td>
<td>..66</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>xAA</td>
<td>xAF</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

**Store:**
- UO_ID: 951012344
- Student: “xAA”

**Idea:**
- Slot 4 is full ... just use the next slot (slot 5).
- Keep going until you find one, including wraparounds
### HashTable

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>...34</td>
<td>...44</td>
<td>...66</td>
<td>...45</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>xAA</td>
<td>xAF</td>
<td>xB8</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

Store:
- UO_ID: 951012345
- Student: “xB8”
<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>...34</td>
<td>...44</td>
<td>..66</td>
<td>...45</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>xAA</td>
<td>xAF</td>
<td>xB8</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

Fetch:
UO_ID: 951045323
EASY: NULL
# HashTable

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key</strong></td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>...34</td>
<td>...44</td>
<td>...66</td>
<td>...45</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td><strong>Student</strong></td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>xAA</td>
<td>xAF</td>
<td>xB8</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

Fetch:
UO_ID: 951012345
More work ... walk from index 5 to 6 to 7 ... found it!
# HashTable

<table>
<thead>
<tr>
<th>Table index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>...34</td>
<td>...44</td>
<td>...66</td>
<td>...45</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Student</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>NUL</td>
<td>xFF</td>
<td>xAA</td>
<td>xAF</td>
<td>xB8</td>
<td>NUL</td>
<td>NUL</td>
</tr>
</tbody>
</table>

**Fetch:**
- **UO_ID:** 951012355
- More work ... walk from index 5 to 6 to 7 to 8... not there!
Another Data Structure for Store/Fetch: Hash Table

```c
typedef struct {
    int keys[MAX_STUDENTS];
    Student *values[MAX_STUDENTS];
} HashTable;

void Initialize(HashTable *arr) {
    for (int i = 0; i < MAX_STUDENTS; i++) {
        arr->keys[i] = -1;
        arr->values[i] = NULL;
    }
}
```

```c
void HashTableStore(void *a, int key, Student *v) {
    HashTable *ht = (HashTable *)a;
    int iter = 0;
    for (int i = 0; i < MAX_STUDENTS; i++) {
        int idx = (i+key)%MAX_STUDENTS;
        if (ht->keys[idx] == -1) {
            ht->keys[idx] = key;
            ht->values[idx] = v;
            return
        }
    }
    exit(EXIT_FAILURE);
}
```

```c
Student *HashTableFetch(void *a, int key) {
    HashTable *ht = (HashTable *)a;
    int iter = 0;
    for (int i = 0; i < MAX_STUDENTS; i++) {
        int idx = (i+key)%MAX_STUDENTS;
        if (ht->keys[idx] == key) {
            return ht->values[idx];
        }
        if (ht->keys[idx] == -1) {
            return NULL;
        }
    }
    return NULL;
}
```

- Observation:
- Still very generic (int key, Student value)
Complexity for Hash Table

• Store: it depends
  – Things go well: $O(1)$
  – Things go poorly: $O(n)$

• Fetch: it depends
  – Things go well: $O(1)$
  – Things go poorly: $O(N)$

• Gets into new topic ... expected performance.
To Motivate ADT, We Need Examples of Data Structures That Can Do Store/Fetch

- Two Examples:
  - Array
  - HashTable
typedef struct {
    void     *self;
    void     (*Store)(void *, int, Student *);
    Student  * (*Fetch)(void *, int);
} StoreFetchADT;

void ADTStore(StoreFetchADT *adt, int key, Student *s) {
    adt->Store(adt->self, key, s);
}

Student *ADTFetch(StoreFetchADT *adt, int key) {
    return adt->Fetch(adt->self, key);
}
Constructing ADT from StoreFetchArray or HashTable

```c
StoreFetchADT *CreateArrayStoreFetchADT()
{
    StoreFetchArray *sfa = malloc(sizeof(StoreFetchArray));
    ArrayInitialize(sfa);
    StoreFetchADT *adt = malloc(sizeof(StoreFetchADT));
    adt->self = sfa;
    adt->Store = ArrayStore;
    adt->Fetch = ArrayFetch;
    return adt;
}

StoreFetchADT *CreateHashTableStoreFetchADT()
{
    HashTable *ht = malloc(sizeof(StoreFetchArray));
    HashTableInitialize(ht);
    StoreFetchADT *adt = malloc(sizeof(StoreFetchADT));
    adt->self = ht;
    adt->Store = HashTableStore;
    adt->Fetch = HashTableFetch;
    return adt;
}
```

This is why ArrayStore/ArrayFetch needed to be void *

int main()
{
    StoreFetchADT *sf_adt = CreateArrayStoreFetchADT();
    Student s1;
    s1.name = "Hank Childs";
    Student s2;
    s2.name = "Henry Childs";
    ADTStore(sf_adt, 37, &s1);
    ADTStore(sf_adt, 42, &s2);
    Student *s = ADTFetch(sf_adt, 37);
    printf("Student is %s\n", s->name);
    s = ADTFetch(sf_adt, 3);
    printf("Student is %p\n", s);
    s = ADTFetch(sf_adt, 42);
    printf("Student is %s\n", s->name);
}

Hanks-iMac:212 hank$ gcc adt.c
Hanks-iMac:212 hank$ ./a.out
Student is Hank Childs
Student is 0x0
Student is Henry Childs
Data structure for specific data type
(example: array of Students)

Data structure for *any* data type
(example: array of “void *”)

Abstract data structure for specific data type
(example: search for student names)

Abstract data structure for *any* data type
(example: search for “void *”)

When Do I Use What?

- When I have developed a data structure I want others to use
- When I want to solve a problem for many, many people
- When writing something for my specific program
- Abstract data structure for any data type (example: search for "void *")
- Abstract data structure for specific data type (example: search for student names)
- Data structure for specific data type (example: array of Students)
- Data structure for any data type (example: array of "void *")
<table>
<thead>
<tr>
<th>No management/organization of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specific data structure for specific type</td>
</tr>
<tr>
<td>Specific data structure for generic type</td>
</tr>
<tr>
<td>Abstract data types</td>
</tr>
</tbody>
</table>
Going to Project 3A
Abstract and Concrete Types

- Abstract type: I can think about it abstractly, but it is *only* abstract.
- Concrete type: it is specific, and I can do things with it.
Abstract Type vs Abstract Data Type

- Abstract Type: a class without all the methods implemented
  - “Is A” test. A triangle “is a” shape.
- Abstract Data Type: just an interface, with no notion of implementation.
  - (what we just lectured on)
- But: the ideas are very similar (identical in some cases), and mechanism to implement both are very similar
Project 3A

• We will implement an abstract type and a concrete type
• Abstract type: Shape
• Concrete type: Rectangle/Circle/Triangle
  – (from 2H!)
CIS 212: Project #3A
Assigned: Thursday, November 1st, 2018
Due Sunday, November 11th, 2018
(which means submitted by 6am on November 2nd, 2018)
Worth 3% of your grade

Please read this entire prompt!

Assignment: You will build off your work from project 2H. In this project, you will build an abstract type (using only C) called Shape, complete with a dispatch table. You will also build 3 concrete types: Circle, Triangle, and Rectangle. The methods and structs for the concrete types will come from your project 2H.

1) Copy your 2H code: cp proj2H.c proj3A.c. All of the following instructions should be changes to the proj3A.c file you have just created.

2) Make an abstract type called Shape. It should have a data member called “self” of type “void **” and two data members that are pointers to functions – GetBoundingBox and GetArea. The signatures of these functions can be inferred from the starter code.
   a) Hint: define Shape immediately after the other three structs.

3) Modify the three “get area” functions and the three “get bounding box” functions to take void * arguments.
   a) Hint: You will need to do a cast inside the function. For example, if GetCircleArea has a “void **” function argument, then you will need to cast this void * to be a “Circle *” inside the function to be able to access its data members.

4) Rename the three Initialize functions to be Create functions. They should no longer take a circle/rectangle/triangle as an argument. Instead, they will create the correct shape in their function definition (with a malloc). They will also create a shape, and connect the shape’s dispatch table to circle/rectangle/triangle. The Create function should return a shape.
   a) Hint: move the three initialize functions to go after the 6 get area / bounding box functions. You need to do this because the shape’s dispatch table will want to point at functions like “GetCircleBoundingBox.” If GetCircleBoundingBox appears later in the file, then the compiler won’t have seen it yet, and it will complain.

5) Add functions called “GetBoundingBox” and “GetArea”. The signatures for these functions are in the new starter code (proj3A_starter.c).

6) Replace the main function that was copied over from proj2H.c with the new main function in the new starter code (proj3A_starter.c).
/* NEW FUNCTIONS */

double GetArea(Shape *s);
void GetBoundingBox(Shape *s, double *bbox);

/* NEW MAIN */

int main()
{
    Shape *shapes[9];
    int i;
    shapes[0] = CreateCircle(1, 0, 0);
    shapes[1] = CreateCircle(1.5, 6, 8);
    shapes[2] = CreateCircle(0.5, -3, 4);

    shapes[3] = CreateRectangle(0, 1, 0, 1);
    shapes[4] = CreateRectangle(1, 1.1, 10, 20);
    shapes[5] = CreateRectangle(1.5, 3.5, 10, 12);

    shapes[6] = CreateTriangle(0, 1, 0, 1);
    shapes[7] = CreateTriangle(0, 1, 0, 0.1);
    shapes[8] = CreateTriangle(0, 10, 0, 50);

    for (i = 0 ; i < 9 ; i++)
    {
        double bbox[4];
        printf("Shape %d\n", i);
        printf("\tArea: %f\n", GetArea(shapes[i]));
        GetBoundingBox(shapes[i], bbox);
        printf("\tBbox: %f-%f, %f-%f\n", bbox[0], bbox[1], bbox[2], bbox[3]);
    }
}
Stacks
Stacks

• A data structure
• 2 methods: push and pop
• Sometimes a third: peek
#define MAX_STACK_SIZE 100

typedef struct
{
    /* your data members go here */
} Stack;

void Initialize(Stack *s)
{
}

void Push(Stack *s, int X)
{
}

int Pop(Stack *s)
{
}

int main()
{
    Stack s;
    int X;
    Initialize(&s);
    Push(&s, 5);
    Push(&s, 6);
    X = Pop(&s);
    printf("Stacked popped %d\n", X);
    Push(&s, 7);
    X = Pop(&s);
    printf("Stacked popped %d\n", X);
    X = Pop(&s);
    printf("Stacked popped %d\n", X);
}
Stack: Asymptotic Complexity

- Push: $O(1)$
- Pop: $O(1)$
- Store: $O(1)$
- Fetch: $O(n)$
  - pop each element and look, and then restore?
Project 3B (1/2)

• Implement a stack:
  – My stack struct has two data members, one of which is an array
  – My Initialize is one line of code
  – My Push is two lines of code, with 5 more lines for error checking
  – My Pop is two lines of code, with 5 more lines for error checking
Project 3B (2/2)

• Implement “Reverse Polish Notation”
  – Traditional: 3+5
  – RPN: 3 5 +
  – Traditional: 3 - 4 x 5
    • Ambiguity: (3 - 4) x 5 or 3 – (4 x 5)
  – RPN: 3 4 5 x – ⇒ 3 - (4 x 5)
  – RPN: 3 4 – 5 x ⇒ (3 - 4) x 5

• A lot of calculators used to work this way
• This will be combined with your project 2E