CSCI 2270
Data Structures and Algorithms
Lecture 8—finish HW1 Bag functions, start HW2 Bag functions

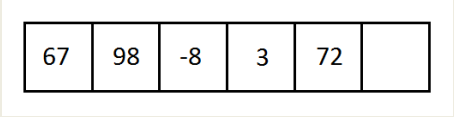
Elizabeth White
elizabeth.white@colorado.edu
Office hours: ECCS 128 or ECCS 112
Wed 9:30am-11:00am
Thurs 10:00am-11:30am

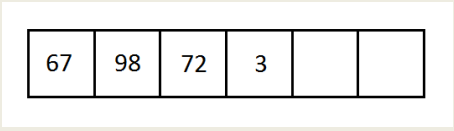
Administrivia

- HW1 posted
  - It will be due Sunday, Feb. 2nd
  - Help hours tonight and tomorrow. Not much Sunday.
- HW2 will post this weekend; resizable Bag
  - It will be due Sunday, Feb. 9th
- Read pp.152-154, 178-184 on resizing the Bag
- Clickers will post tomorrow (oof).

Removing items

Suppose the item we're removing is at array position q, and the number of items we have is greater than q. What code would move the last item over the item we're removing?





a) items[0] = items[q];
b) items[q] = items[itemCount - 1];
c) items[q] = items[itemCount];
d) items[itemCount] = items[q];
e) items[itemCount - 1] = items[q];

Removing items

If the Bag were an ordered collection of items, what would be bad about this gap filling plan?

How would we preserve order while removing items in the Bag, if we had to? (Don't do this for hw1 or hw2.)

How much work does preserving the order take, compared to what we did in the previous slides?

What would we have to watch out for here?

Listing items

Create an empty vector (another array-like template class in C++)

vector<ItemType> itemList;

Write a loop: for each item in the items array, add it to the itemList vector using a line like this (with no ?)

itemList.push_back(items[?]);

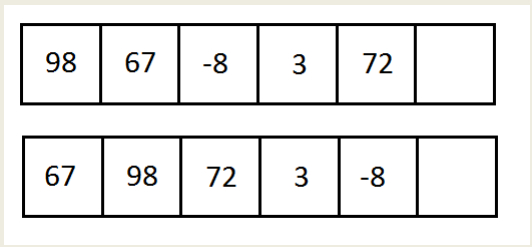
Seems a little redundant, no?

We're using an array to store data anyway

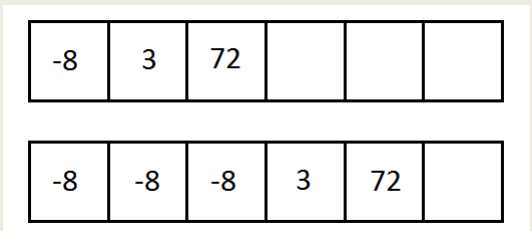
Later, when you write a B-tree container class, this routine will let you make a list from its items even if they're not stored in a linear way; then it will seem more useful.

Food for thought…

How would you decide if 2 Bags were equal, ?



equal Bags



not equal Bags

Remember this pointer slide?

```
void example8()
{
     int a = 5;
     int* a_ptr = &a;
     a++;            // a becomes 6
     (*a_ptr)++;     // a becomes 7
}
```

What does the last line do now?

```
void example8()
{
     int a = 5;
     int* a_ptr = &a;
     a++;            // a becomes 6
     (*a_ptr)++;     // a becomes 7
     a_ptr++;        // skip a_ptr over
                     // this int a, to
                     // the next address
                     // 4 bytes later
                     // in memory
}
```

Pointer arithmetic

In C++, you can increment pointer addresses using + (or decrement them with -)

```
int* c = new int[10];              // another array
for (int q = 0; q < 10; ++q)
        c[q] = q;
cout << c[5] << endl;              // prints 5
cout << c[0] + 5 << endl;          // also prints 5
cout << c[9] - 4 << endl;          // also prints 5
```

Pointer arithmetic and the copy command

In C++, the library <algorithm> includes the **copy** command to copy items from one array to another array. This function uses the same pointer arithmetic logic as you just saw.

Input arguments: where to start copying, where to stop copying, and destination.

Overlapping ranges for source and destination are not allowed.

Pointer arithmetic and the copy command

Specify: where to start copying, where to stop copying, and destination

```
int* c = new int[10];        // another array
for (int q = 0; q < 10; ++q)
        c[q] = q;            // c: 0 1 2 3 4 5 6 7 8 9
int* d = new int[10];
copy(c, c+5, d);             // d: 0 1 2 3 4
copy(c, c+5, d+5);           // d: 0 1 2 3 4 0 1 2 3 4
```

The new command

You can create variables in places other than the local memory… like the heap. You use the new command to do this.

```
// define an integer equal to -8 on the heap
// make a hold the address of this -8
int* b = new int(-8);
```

Notice, here, that the variable with the value -8 has no name. We only have a pointer to this value, which is a.

Heap variables are less mortal than ordinary ones

```
void lecture_8_example_4()
{
        int a = -4;
        int* b = new int(-8);
}
```

At the end of the function, the variable a is destroyed (which means that the operating system will let other variables overwrite it in memory). So that -4 goes away. This is the same behavior you saw in lecture 2.

Heap variables are less mortal than ordinary ones

```
void lecture_8_example_4()
{
        int a = -4;
        int* b = new int(-8);
}
```

The nameless -8 that lives on the heap, however, is not destroyed. The operating system protects its memory and won't overwrite it.

The pointer variable b, which is a local variable, gets destroyed at the end of the function. Now we have no way to get to our -8 on the heap; we've lost it.

How not to lose heap variables, part 1

Losing heap variables like this uses up memory; we sometimes call it a *memory leak*.

We could rewrite this example so it didn't lose the pointer to the -8 but instead returned that pointer:

```
int* lecture_8_example_5()
{
        int* b = new int(-8);
        return b;
}
```

Since we return the pointer to the -8 here, we can keep track of the -8.

How not to lose heap variables, part 2

Alternatively, we could rewrite this example so it passes in the pointer by reference:

```
void lecture_8_example_6(int*& b)
{
        b = new int(-8);
}
```

Now any changes to the pointer b propagate back to the calling program, and we can still keep track of that -8.

The delete command

To give back a heap variable's memory, we use the delete command:

```
int* a = new int(-8);
delete a;                    // poof! a disappears
```

If the heap variable is an array, we add [] to the delete command:

```
int* nummies = new int[10];
delete [] nummies;           // no more nummies
```

HW2, with the heap

In the next homework, we'll make the Bag expandable, so it can upsize as needed to hold more items.

First big change:

```
ItemType items[DEFAULT_CAPACITY];
```

becomes

```
ItemType* items;
```

So items becomes a pointer to an (as yet undefined) array of items.

HW2

Second change: your constructor

Your constructor must make the array, using new:

```
items = new ItemType[myCapacity];
```

After this, items is a pointer to an array on the heap. (This means that items is the address of the first item in the array.)

Your constructor also needs to keep track of the current capacity of the bag.

```
myCapacity = DEFAULT_CAPACITY;
```

or

```
myCapacity = capacity;
```

HW2

Third change: now you must write a function to give back the memory the Bag is using for the items. This function's called a destructor. Its job is to return the memory used by the items array to the heap. What should it say?

a) `delete items;`

b) `delete itemCount;`

c) `return items;`

d) `delete [] itemCount;`

e) `delete [] items;`

HW2

Fourth change: you will need to add a new method called resize() to the bag. When your items array gets full, this method will:

1. Make a new items array, twice as big as the old one
2. Copy all the items from the old items array to the new items array
3. Delete the old items array
4. Set items = new items array
5. Update myCapacity to the new array size

HW2

Fifth change: add a copy constructor

You will write a new constructor that initializes a Bag as a copy of another Bag (more on this Monday)

Sixth change: add an assignment operator

You will write the code to assign one Bag to another

More on this next week, along with deep and shallow copies