CS 261: Data Structures

Dynamic Arrays

Stacks and Bags

Interface View of Stack

Stack as ADT

- **Definition**: Maintains a collection of data elements in Last-In, First-Out format

- **Operations**:
  - Add an element to Stack
  - Remove an element from Stack
  - Read the top element of Stack
  - Check if an element is contained in Stack

Stack Interface

- Provide functions for operations, effectively hiding implementation

```
initStack (container);

pushStack (container, value);

topStack (container);

popStack (container);

isEmptyStack (container);
```

Interface file: **dynArr.h**

```
struct dyArr {
  TYPE * data; /*Pointer to data array */
  int size;    /*Number of elements */
  int capacity;/*Capacity of array */
};


/* Rest of dynarr.h on next slide */
```

Interface of DA (continued)

```
/* function prototypes */

void initDynArr (struct dyArr *da, int cap);

void freeDynArr (struct dyArr *da);

void addDynArr (struct dyArr *da, TYPE val);

TYPE getDynArr (struct dyArr *da, int idx);

TYPE putDynArr (struct dyArr *da, int idx, TYPE val);

int sizeDynArr (struct dyArr *da);

void _dyArrDoubleCapacity (struct dyArray * da);
```

Interface of Stack (continued)

```
/* function prototypes */

void initStack (struct dyArr *da, int cap);

void freeStack (struct dyArr *da);

void pushStack (struct dyArr *da, TYPE d); /*add*/

TYPE topStack (struct dyArr *da);/*only reads*/

void popStack (struct dyArr *da);/*moves 1 step down*/

int isEmptyStack (struct dyArr *da);
```

Implementation View of Stack as Dynamic Arrays

initStack -- Initialization

```
void initStack (struct dyArr *da, int cap)
{
     /* why reinvent the wheel? */
     initDynArr(da,cap);
}
```

freeStack -- Clean-up

```
void freeStack (struct dyArr * da)
{
    freeDynArr(da);
}
```

isEmpty

```
int isEmptyStack(struct dyArray *da)
{
     return (sizeDynArr(da) == 0);
}
```

Add a New Element to Stack

```
void pushStack (struct dyArr * da, TYPE val)
{
     /* why reinvent the wheel? */

     addDynArr(da,val);

     /* because, addDynArr already inserts
         a new element at the end */
}
```

Top Stack

```
/* reads the top elements of stack */
TYPE topStack (struct dyArr *da){
    /* make sure the stack is not empty */
    assert(sizeDynArr(da) > 0);
    return getDynArr(da, sizeDynArr(da)-1);
}
```

why?

Pop Stack

```
/* moves the pointer to the top element */
void popStack(struct dyArray *da){
    /* make sure the stack is not empty */
    assert(sizeDynArr(da) > 0);
    d->size--; /* decrement the size */
}
```

Interface View
of
Bag

Interface of Bag (continued)

```
/* function prototypes */

void initBag (struct dyArr *da, int cap);

void freeBag (struct dyArr *da);

void addBag (struct dyArr *da, TYPE val);

int containsBag (struct dyArr *da, TYPE val);

void removeBag (struct dyArr *da, TYPE val);

int sizeBag (struct dyArr *da);
```

Init, Free, Add, Size

- All are the same as for Dynamic Arrays

removeBag

- More useful to have two functions

```
void removeDyArr (struct dyArray *da, TYPE val)
```

```
void _removeAtDyArr (struct dyArray *da, int idx)
```

removeBag -- Single

```
void removeDyArr(struct dyArray *da, TYPE val){
  int i=0;

  while (i < da->size)
  {
      if ( EQ(val,da->data[i]) )
      {
            _removeAtDyArr(da, i);
            return;
      }
      i++;
  }
}
```

removeBag -- Single

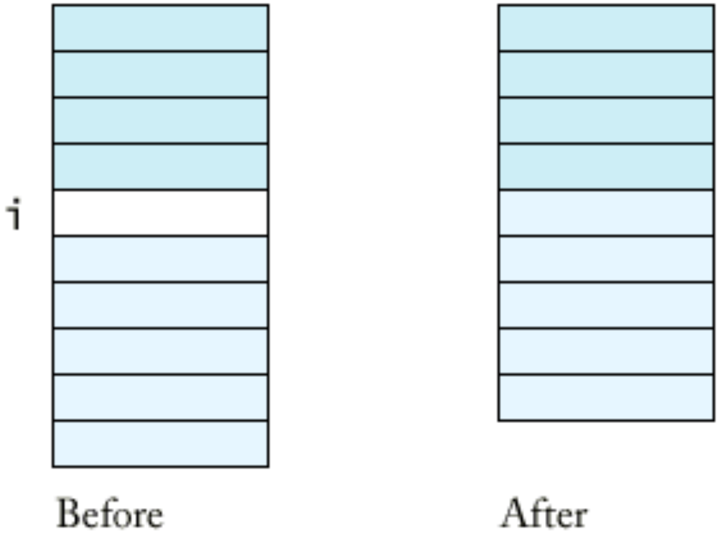
```
void removeDyArr(struct dyArray *da, TYPE val){
  int i=0;

  /*for (i=0; i < da->size; i++)ALTERNATIVE SOLUTION*/

  while (i < da->size)
  {
      if ( EQ(val,da->data[i]) )
      {
            _removeAtDyArr(da, i);
            return;
      }
      i++;
  }
}
```

removeBag -- Multiple

```
void removeDyArr(struct dyArray *da, TYPE val){
   int i=0;
   while (i < da->size)
   {
        if ( EQ(val,da->data[i]) )
        {
              _removeAtDyArr(da, i);
              i--;
        }
        i++;
   }
}
```

RemoveAt requires extra work

- Must move the elements
- Think: from `i` ? or from the end?



RemoveAT for a Bag

```
void _removeAtDyArr(struct dyArray *da, int idx){
  int i;

  assert(da->size > idx) && (idx >= 0));

  for (i = idx; i <= da->size - 2; i++)/*end before*/

      da->data[i] = da->data[i+1];/* copy from idx */

  da->size--;
}
```

Complexity?

RemoveAT for a Bag
– Not ordered, Single Remove –

```
void _removeAtDyArr(struct dyArray *da, int idx){
   int i;

  assert(da->size > idx) && (idx >= 0));

  /*put the last element in place of the element to be
   removed*/

  da->data[idx] = da->data[da->size - 1];

  da->size--;
}
```

Complexity?

Which version of _removeAtDyArr to use?

```
void removeDyArr(struct dyArray *da, TYPE val){
  int i=0;

  while (i < da->size)
  {
      if ( EQ(val,da->data[i]) )
      {
            _removeAtDyArr(da, i);
            return;
      }
      i++;
  }
}
```

Which version of _removeAtDyArr to use?

```
void removeDyArr(struct dyArray *da, TYPE val){
   int i=0;

  while (i < da->size)
   {
       if ( EQ(val,da->data[i]) )
       {
              _removeAtDyArr(da, i);
              return;
       }
       i++;
   }
```

Overall complexity is the same in both cases!

RemoveAT for a Bag
– Not ordered, Multiple Remove –

```
void _removeAtDyArr(struct dyArray *da, int idx){
   int i;

   assert(da->size > idx) && (idx >= 0));

   /*put the last element in place of the element to be
   removed*/

   da->data[idx] = da->data[da->size - 1];

   da->size--;
}
```

CAREFUL FOR MULTIPLE REMOVALS

removeBag -- Multiple

```
void removeDyArr(struct dyArray *da, TYPE val){
   int i=0;
   while (i < da->size)
   {
       if ( EQ(val,da->data[i]) )
       {
            _removeAtDyArr(da, i);
            i--;
       }
       i++;
   }
}
```

What about **`addAt`** ?

- We don’t need **`addAt`** for the bag (order doesn’t matter), but it is a nice complement to **`removeAt`**

- We will use **`addAt`** later

Worksheets 14, 16 & 21

- Finish the implementation of Stack and Bag.
- Your implementation will be used in the next programming assignment.