Binary Search Trees III

Generic Container

Goals

- void *
- compare function

Generic Storage

```
#ifndef __DYNARR_H
#define __DYNARR_H

# define TYPE int
# define TYPE_SIZE sizeof(TYPE)

# define LT(a, b) ((a) < (b)

# define EQ(a, b) ((a) == (b))

...  /* Rest of dynarr.h (on next slide). */

#endif
```

```
struct Node *_addNode(struct Node *cur, TYPE val){
  struct Node *newnode;
  if (cur == NULL){
      newnode = malloc(sizeof(struct Node));
      assert(newnode != 0);
      newnode->val = val;
      newnode->left = newnode->right = 0;
      return newnode;
  }
   if (LT(val, cur->val))
      cur->left = _addNode(cur->left,val);
      else if (GT(val, cur->val) || EQ(val, cur->val))
              cur->right = _addNode(cur->right, val);
  return cur;
}
```

Problem: must recompile our data structure if we change the type!

Generic Storage Alternative

- Write your structure to store void *
  - generic pointer type, indicates absence of a type
  - any pointer can be cast to void* and back again without loss of information
- Require that the user tell us how to 'compare' the elements that they are actually storing!
  - compare works on void *
  - casts them to actual types!

Generic Storage

```
#ifndef __DYNARR_H
#define __DYNARR_H

# define TYPE void *
```

/* function used to compare two TYPE values to each other, define this in your compare.c file
0 lif l =r
-1 if l < r
1 if l > r */

int compare(TYPE left, TYPE right);

Add with Void Pointers

```
struct Node *_addNode(struct Node *cur, TYPE val){
  struct Node *newnode;
  if (cur == NULL){
      newnode = malloc(sizeof(struct Node));
      assert(newnode != 0);
      newnode->val = val;
      newnode->left = newnode->right = 0;
      return newnode;
  }
   if (compare(val,cur->val) == -1)
      cur->left = _addNode(cur->left,val);
   else if (compare(val, cur->val) == 1)
      cur->right = _addNode(cur->right, val);
  return cur;
}
```

BST Add: Usage

```
void addBSTree(struct BSTree *tree, TYPE val)
{
        tree->root = _addNode(tree->root, val);
        tree->cnt++;
}
```

…

```
/* somewhere in main*/
    struct data {
      int val;
      char *name;
    }
    struct data myData;
    myData.val = 4;
    myData.name = "dataName";
    …
    add(myTree, &myData);
```

Usage with Void Pointers

```
/* somewhere in main…or a header */
int compare(TYPE left, TYPE right)
{
        struct data* data1;
        struct data* data2;
        data1=(struct data*)left;
        data2=(struct data*)right;

        if (data1->number < data2->number)
                return -1;
        else if (data1->number > data2->number)
                return 1;
        else
                return 0;

}
```

Summary – Generalizing your Data Structures

- Use void* as the type of stored elements
- Require that the user provide a compare function
- Even Better: Pass the compare function in with a pointer to a function!!