Lecture 14
Generic Data Structures

In this lecture

- Introduction to Generic Data structures
- Examples

An Introduction to Generic Data Structures

Programs use many different data structures such as arrays, linked lists, hash tables, general trees, binary search trees, heaps etc. Each data structure is a container that holds a particular data type. Operations changes from data type to data type, for example, how you add two integers is not the same as how you "add" two tables. While we understand how to add two integers, "adding" two tables can be custom defined. Generic data types are important in designing libraries that works with "any" data type. A dynamic binding between data type and data structure occurs at run time.

Generic data types are common in some high level languages. For example in Java 5, a generic data type can be defined using:

```
/**
 * Generic version of the Item class.
 */
public class Item<T> {

    private T t; // T for type is late binding

    public void init(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}

Item<Integer> intItem;
Item<Double> doubleItem;

Hence Java allows the flexibility of late binding to types thereby
allowing developers to develop generic libraries.
```

In C++, a Standard Template Library (STL) provides similar facilities. For example, an example of using generic vector class for strings can be defined as:

```
#include <vector>
#include <string>

using namespace std;

main()
{
   vector<string> V;

   V.push_back("10");
   V.push_back("20");
   V.push_back("30");
   V.pop_back();

   cout << "Loop by index:" << endl;

   int i;
   for(i=0; i < V.size(); i++)
   {
      cout << V[i] << endl;
   }
}
```

Other modern languages such as C# provides generic classes for flexible and reusable code development. Although C does not have a built in facilities for generic data types, one can use the power of function pointers to build a good generic data structure. The following examples show building a generic data structure for a linked list and a doubly linked list structure.

An Example

In this example we define a generic LIST_ELEM that can hold any data type as its data.

```
typedef struct LIST_ELEM
{
      void * data;
      struct LIST_ELEM  * next;
} LIST_ELEM;
```

The LINKED_LIST structure defined below contains three function pointers that define how to compare, print and free data of a particular data type. These functions can be passed during run time.

```
typedef struct LINKED_LIST
{
      LIST_ELEM  * head ;
      int (*cmpData)( void *, void *);
      void (*printData)( void *);
      void (*freeData)( void *);
} LINKED_LIST;
```

The functions defined below are standard functions for LL operations not written for any particular data type, but uses the dynamic function pointers for operations specific to a particular data type.

```
void initList( LINKED_LIST * list, int (*cmpData)( void *, void *),
void (*printData)( void *), void (*freeData)( void *) );
void freeAll(LINKED_LIST * list); /* free ALL dynamic memory in this
program */
void insertInOrder(LINKED_LIST * list,  void *data);
void removeAtFront(LINKED_LIST  * list  );
```

We will discuss how to develop these functions in class.

Another Example

Here is another example of a generic data structure representing a doubly linked list.

```
typedef struct DLL_NODE {
  void *key;              /* Key for node */
  void *value;            /* The value for the node */
  struct DLL_NODE *next;  /* pointer to next node */
  struct DLL_NODE *prev;  /* pointer to previous node */

} dll_node;

typedef struct DLL {
  dll_node *head;           /* head node of the list */
  int (*cmp)(void*, void*); /* Compare function pointer */
} dll_l;

int dll_init_list(dll_l *list);

int dll_set_cmp(dll_l *list, int (*cmp)(void*, void*));

int dll_insert(dll_l *list, void *key, void *value);

int dll_retrieve(dll_l *list, void *key, void **value);
```

Try developing above functions as homework. This data structure may also be a part of next lab.