Dynamic Data Structures with C

Terry Marris August 2010

6 Sets

We have looked at lists, iterators and pointers to functions. We now see how they can be used to implement sets.

6.1 Set

The defining property of a set is that it is a collection of objects without duplicates. The essential programming problem here is: since we cannot predict what data types users will store in our set, how can we ensure that each element in the set is not repeated? For example, two objects might refer to the same patient if their national health service (NHS) numbers are identical, two objects might refer to the same person if their names are identical and their date of births are identical and their addresses are identical. We rely on the user to provide an appropriate C function that determines whether two objects are identical.

Sets are a fundamental data structure of mathematics, and are used to model safety-critical systems, databases and artificial intelligence applications.

6.2 Set Interface

The interface for our set data type is shown below. We declare pointers to functions for *Equal*, *Duplicate* and *Key*. We introduce the *Set* type. We have a function to create a new set with pointers to functions for *equal()*, *duplicate()* and *key()*. We have functions to erase a set, to determine whether a set has any elements, to add an object to a set, remove an object from a set, and to determine whether a given object is a member of the set or not.

```
/* set.h */

#ifndef SET
#define SET

typedef int (*Equal)(void *, void *);
typedef void *(*Duplicate)(void *);
typedef char *(*Key)(void *);
typedef struct SetStruct *Set;
typedef struct SetIteratorStruct *SetIterator;

/* newSet: returns a new, empty set */
Set newSet(Equal, Duplicate, Key);

/* deleteSet: erases the given set, which must be empty */
Set deleteSet(Set *s);

/* setIsEmpty: returns 1 if the set has no elements */
int setIsEmpty(const Set s);

/* copySet: returns a duplicate copy of the given set */
```

```
Set copySet(const Set s);

/* includeInSet: adds the given object to the set if
   it is not already there */
Set includeInSet(const Set s, void * const obj);

/* removeFromSet: removes the given element from the set */
Set removeFromSet(const Set s, void * const element);

/* setContains: returns 1 if the given object is in the set */
int setContains(const Set s, void * const obj);

/* newSetIterator: returns a new iterator for the given set */
SetIterator newSetIterator(const Set s);

/* setIteratorHasNext: returns 1 if the set has an element not
   yet visited in the current iteration */
int setIteratorHasNext(const SetIterator iter);

/* setIteratorNext: goes on to the next element and returns it */
void *setIteratorNext(SetIterator iter);

/* setUnion: returns a new set containing all the elements
   from two sets, s and t */
Set setUnion(const Set s, const Set t);

/* setIntersection: returns a new set containing all the elements
   common to two sets, s and t */
Set setIntersection(const Set s, const Set t);

/* setDifference: returns a new set with all the elements of set t
   removed from set s */
Set setDifference(const Set s, const Set t);

#endif
```

6.3 Testing

We start by considering a telephone directory that might be used in a small business. Employees may have access to one or more telephones, some telephones may be unused. Each telephone has a four-digit number. We can imagine a telephone directory containing *<employeeName, telephoneNumber>* pairs.

```
typedef struct PhoneDirEntryStruct {
  char name[15];
  char number[6];
} *PhoneDirEntry;
```

newEntry() creates a new telephone directory entry from the given name and number.

```
/* newEntry: returns a new entry with the given name and number */
PhoneDirEntry newEntry(char *name, char *number)
{
  PhoneDirEntry entry = malloc(sizeof(struct PhoneDirEntryStruct));
  assert(entry != NULL);
  strcpy(entry->name, name);
  strcpy(entry->number, number);
  return entry;
}
```

We cannot allow duplicate entries in the directory.

```
/* equalEntries: returns 1 if the two given entries are identical */
int equalEntries(const PhoneDirEntry e1, const PhoneDirEntry e2)
{
  if (strcmp(e1->name, e2->name) == 0 &&
      strcmp(e1->number, e2->number) == 0)
    return 1;
  return 0;
}
```

When adding new entries to the directory, we want to create a duplicate copy of the entry to be stored. By using value semantics (rather than reference semantics) we insulate the entries in our directory from changes outside the directory.

```
/* duplicateEntry: returns a copy of the given entry */
PhoneDirEntry duplicateEntry(const PhoneDirEntry entry)
{
  PhoneDirEntry copy = malloc(sizeof(struct PhoneDirEntryStruct));
  assert(copy != NULL);
  strcpy(copy->name, entry->name);
  strcpy(copy->number, entry->number);
  return copy;
}
```

Notice that we have used the *assert()* function rather than a purpose-built error-reporting function to report out-of-memory errors.

Now, we require the user to specify precisely which fields should be considered when determining whether two objects are identical or not. For our telephone directory we specify that both fields are significant.

```
/* entryKey: returns a string that uniquely represents an entry */
char *entryKey(const PhoneDirEntry entry)
{
  char *s = malloc(sizeof(entry->name) +
      sizeof(entry->number) + 1); /* +1 for \0 */
  assert(s != NULL);
  strcpy(s, entry->name);
  strcat(s, entry->number);
  return s;
}
```

We concatenate the significant fields into one long string.

The code supporting the telephone directory could be better placed in its own files. But ...

Now we come to the *main()* function. First, we create a new set named *dir*, and initialise it with pointers to the *equalEntries()*, *duplicateEntry()* and *entryKey()* functions introduced above.

```
  Set dir = newSet((Equal)equalEntries,
                   (Duplicate)duplicateEntry,
                   (Key)entryKey);
```

Notice the cast operators *(Equal)*, *(Duplicate)* and *(Key)*. These types are defined in *set.h*. We shall see, when we come to consider the implementation, how these pointers to functions will be used.

A newly-created set must be empty.

```
printf("Testing setIsEmpty(). Expect to see empty: ");
setIsEmpty(dir)? printf("empty") : printf("not empty");
printf("\n");
```

We create a new telephone directory entry and attempt to include it in the directory twice. Again, we check to see if the set is empty (it should not be).

```
  PhoneDirEntry e1 = newEntry("bond", "007");
  includeInSet(dir, e1);
  includeInSet(dir, e1);
  printf("Testing setIsEmpty(). Expect to see not empty: ");
  setIsEmpty(dir)? printf("empty") : printf("not empty");
  printf("\n");
```

We remove just one element from the set. If just the one was stored there our removal should result in an empty set, but only if our add new objects function rejects duplicate entries.

```
  printf("Testing removeFromSet(). Expect to see empty: ");
  removeFromSet(dir, e1);
  setIsEmpty(dir)? printf("empty") : printf("not empty");
  printf("\n");
```

The union of two sets is a new set that contains the elements from both sets. So:

{ a, b, c } union { c, d, e } = { a, b, c, d, e }

The intersection of two sets is a new set that contains the elements that are common to both sets. So:

{ a, b, c } intersection { c, d, e } = { c }

The difference between two sets A and B is a new set that contains the elements from A with the elements from B removed. So:

{ a, b, c } difference { c, d, e } = { a, b }

As before, we include an iterator to visit each element in a set in turn.

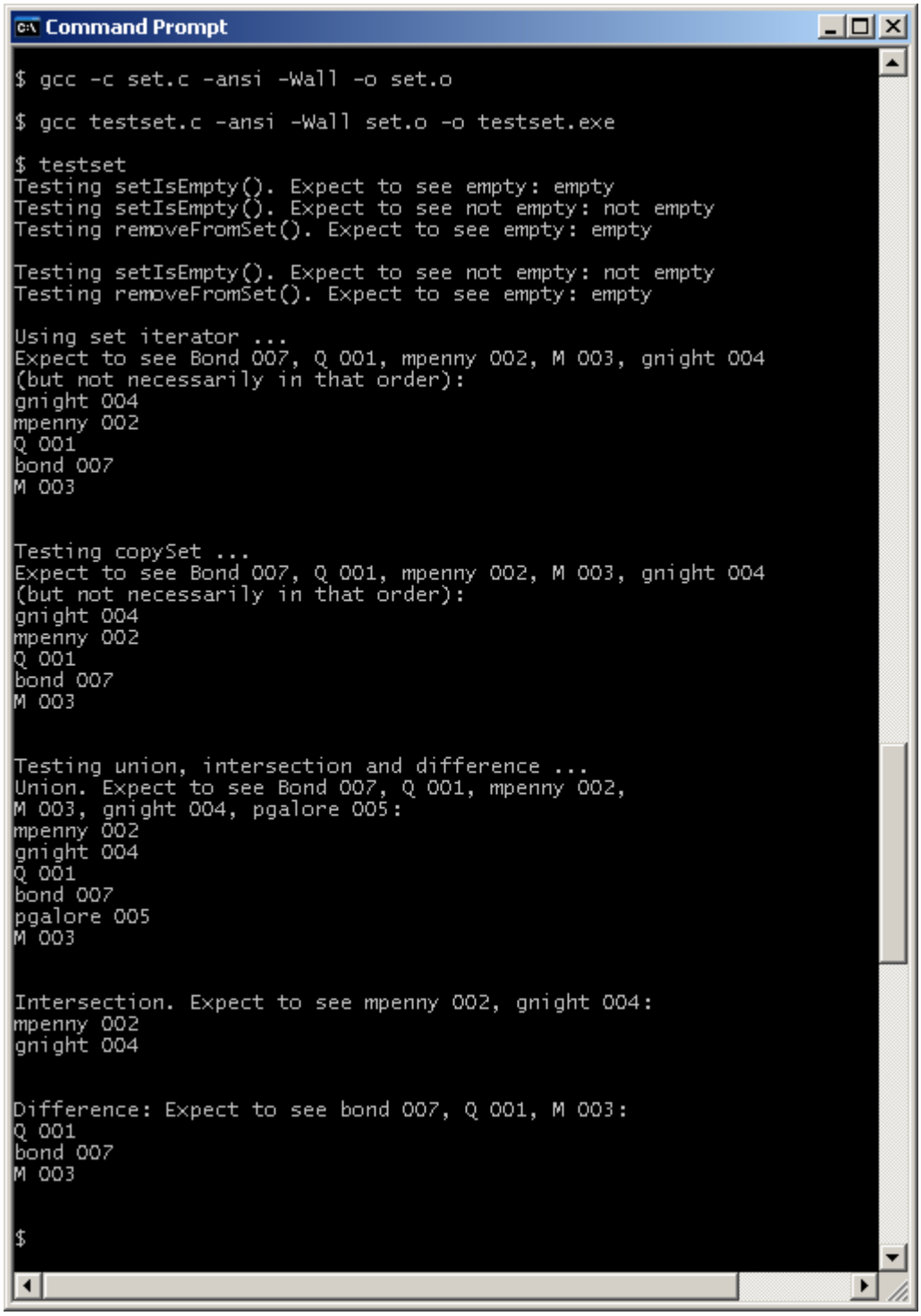
Here is the entire program and its run.

```
/* testset.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include "set.h"

typedef struct PhoneDirEntryStruct {
  char name[15];
  char number[6];
} *PhoneDirEntry;
```

```
/* newEntry: returns a new entry with the given name and number */
PhoneDirEntry newEntry(const char *name, const char *number)
{
  PhoneDirEntry entry = malloc(sizeof(struct PhoneDirEntryStruct));
  assert(entry != NULL);
  strcpy(entry->name, name);
  strcpy(entry->number, number);
  return entry;
}

/* equalEntries: returns 1 if the two given entries are identical */
int equalEntries(const PhoneDirEntry e1, const PhoneDirEntry e2)
{
  if (strcmp(e1->name, e2->name) == 0 &&
      strcmp(e1->number, e2->number) == 0)
    return 1;
  return 0;
}

/* duplicateEntry: returns a copy of the given entry */
PhoneDirEntry duplicateEntry(PhoneDirEntry entry)
{
  PhoneDirEntry copy = malloc(sizeof(struct PhoneDirEntryStruct));
  assert(copy != NULL);
  strcpy(copy->name, entry->name);
  strcpy(copy->number, entry->number);
  return copy;
}

/* entryKey: returns a string that uniquely represents an entry */
char *entryKey(PhoneDirEntry entry)
{
  char *s = malloc(sizeof(entry->name) + sizeof(entry->number) + 1);
  assert(s != NULL);
  strcpy(s, entry->name);
  strcat(s, entry->number);
  return s;
}

int printEntry(PhoneDirEntry entry)
{
  printf("%s %s\n", entry->name, entry->number);
  return 0;
}

int main()
{
  Set dir = newSet((Equal)equalEntries,
                   (Duplicate)duplicateEntry,
                   (Key)entryKey);

  printf("Testing setIsEmpty(). Expect to see empty: ");
  setIsEmpty(dir)? printf("empty") : printf("not empty");
  printf("\n");

  PhoneDirEntry e1 = newEntry("bond", "007");
  dir = includeInSet(dir, e1);
  dir = includeInSet(dir, e1);
  printf("Testing setIsEmpty(). Expect to see not empty: ");
  setIsEmpty(dir)? printf("empty") : printf("not empty");
  printf("\n");
```

```
printf("Testing removeFromSet(). Expect to see empty: ");
dir = removeFromSet(dir, e1);
setIsEmpty(dir)? printf("empty") : printf("not empty");
printf("\n\n");

e1 = newEntry("bond", "007");
dir = includeInSet(dir, e1);
dir = includeInSet(dir, e1);
printf("Testing setIsEmpty(). Expect to see not empty: ");
setIsEmpty(dir)? printf("empty") : printf("not empty");
printf("\n");

printf("Testing removeFromSet(). Expect to see empty: ");
dir = removeFromSet(dir, e1);
setIsEmpty(dir)? printf("empty") : printf("not empty");
printf("\n\n");

printf("Using set iterator ...\n");
PhoneDirEntry e2 = newEntry("Q", "001");
PhoneDirEntry e3 = newEntry("mpenny", "002");
PhoneDirEntry e4 = newEntry("M", "003");
PhoneDirEntry e5 = newEntry("gnight", "004");

dir = includeInSet(dir, e1);
dir = includeInSet(dir, e2);
dir = includeInSet(dir, e3);
dir = includeInSet(dir, e4);
dir = includeInSet(dir, e5);

printf("Expect to see Bond 007, Q 001, mpenny 002, ");
printf("M 003, gnight 004 \n");
printf("(but not necessarily in that order): \n");

SetIterator it = newSetIterator(dir);
while (setIteratorHasNext(it)) {
  printEntry(setIteratorNext(it));
}
printf("\n\n");

printf("Testing copySet ... \n");
Set dirCopy = copySet(dir);
printf("Expect to see Bond 007, Q 001, mpenny 002, ");
printf("M 003, gnight 004 \n");
printf("(but not necessarily in that order): \n");

it = newSetIterator(dirCopy);
while (setIteratorHasNext(it)) {
  printEntry(setIteratorNext(it));
}
printf("\n\n");

printf("Testing union, intersection and difference ...\n");
Set dir2 = newSet((Equal)equalEntries,
                  (Duplicate)duplicateEntry,
                  (Key)entryKey);

Set dir3 = newSet((Equal)equalEntries,
                  (Duplicate)duplicateEntry,
                  (Key)entryKey);
```

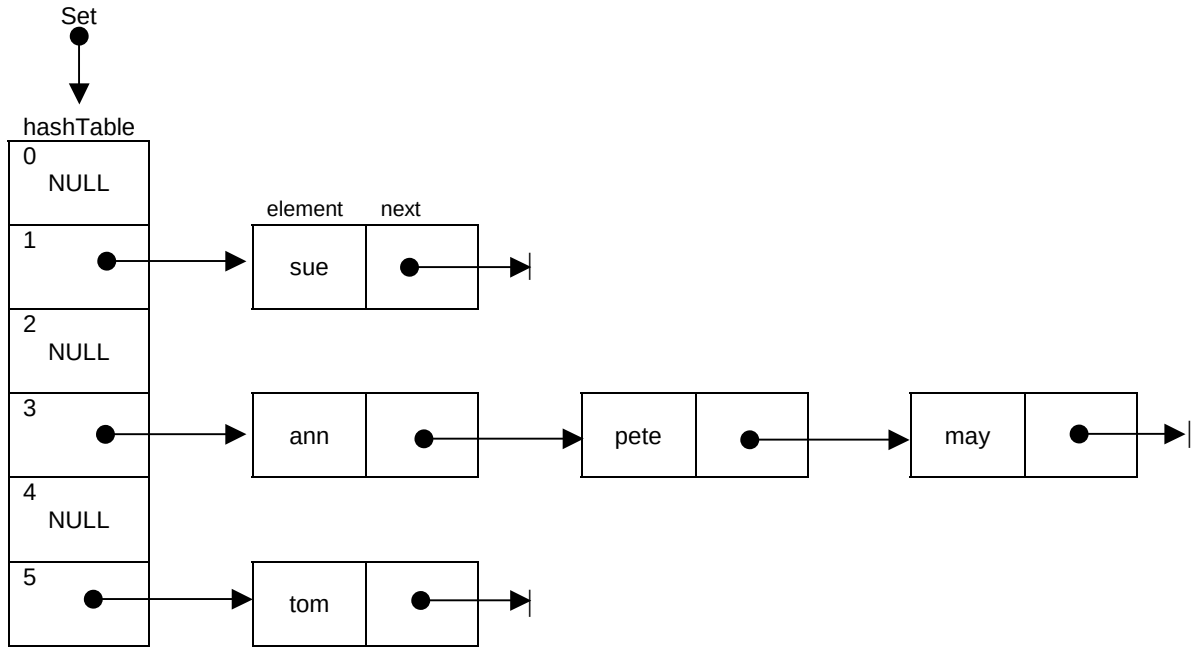
```
  PhoneDirEntry e6 = newEntry("pgalore", "005");
  dir2 = includeInSet(dir2, e3);
  dir2 = includeInSet(dir2, e5);
  dir2 = includeInSet(dir2, e6);

  printf("Union. Expect to see Bond 007, Q 001, mpenny 002, \n");
  printf("M 003, gnight 004, pgalore 005: \n");
  dir3 = setUnion(dir, dir2);
  it = newSetIterator(dir3);
  while (setIteratorHasNext(it)) {
    printEntry(setIteratorNext(it));
  }
  printf("\n\n");

  dir3 = newSet((Equal)equalEntries,
                  (Duplicate)duplicateEntry,
                  (Key)entryKey);
  printf("Intersection. Expect to see mpenny 002, gnight 004: \n");
  dir3 = setIntersection(dir, dir2);
  it = newSetIterator(dir3);
  while (setIteratorHasNext(it)) {
    printEntry(setIteratorNext(it));
  }
  printf("\n\n");

  dir3 = newSet((Equal)equalEntries,
                  (Duplicate)duplicateEntry,
                  (Key)entryKey);
  printf("Difference: Expect to see bond 007, Q 001, M 003: \n");
  dir3 = setDifference(dir, dir2);
  it = newSetIterator(dir3);
  while (setIteratorHasNext(it)) {
    printEntry(setIteratorNext(it));
  }
  printf("\n");

  return 0;
}
```



Now, we turn our attention to an implementation.

6.4 Set Implementation

The underlying data structures for our set implementation are a one-dimensional array and a linked list. We use a hash function to translate a string into an integer that represents a location in the array, and a linked list to accommodate the several items that hash to the same location.



Since order is not important, we could have used just a linked list or a binary tree (see a later chapter). But we have chosen to use a hash table because searching for a particular object is relatively quick. With a hash table you can go directly to where the object should be (but you might have to do a search through a small linked list to reach it). With a linked list you would have to search through the entire list before establishing that the item you are looking for is not there. However, the search time would be reduced if you kept the items in some sort of order e.g. alphabetical order. With an ordered binary tree you could also relatively quickly find the item you are looking for since you are discarding half the items from the search with each probe. But that is another story.

6.4.1 Set Element

SetElement is a *struct* with two fields: one a pointer to the *next* node, and one named *element*, a pointer to *void*, to refer to the data item being stored.

```
struct SetElement {
  struct SetElement *next;
  void *element;
};
```

6.4.2 Set Struct

The hash table is an array of pointers to *SetElement* (described above). We constrain its size to 101 elements, and notice that 101 is a prime number.

```
#define HASHSIZE 101

struct SetStruct {
  struct SetElement *hashTable[HASHSIZE];
  Equal equalObjects;
  Duplicate duplicateObject;
  Key objectKey;
};
```

If we find that our table becomes more than half filled we would consider expanding its size to e.g. 541, another prime number. Why use prime numbers? Well we are concerned with

spreading out our entries in the hash table and minimising the use of linked lists (which would occur if several different objects hash to the same value) and using prime numbers in our calculations helps achieve this.

We have also included, as members, the pointers to functions *Equal*, *Duplicate* and *Key*. We see how they are initialised in the next section.

6.4.3 New Set

We malloc a hole in memory for the set structure.

```
/* newSet: returns a new, empty set */
Set newSet(Equal equal, Duplicate duplicate, Key key)
{
  Set s = malloc(sizeof(struct SetStruct));
  assert(s != NULL);
  int i;
  for (i = 0; i < HASHSIZE; i++)
    s->hashTable[i] = NULL;
  s->equalObjects = equal;
  s->duplicateObject = duplicate;
  s->objectKey = key;
  return s;
}
```

Notice that we have used the *assert()* function to manage error reporting. It might be an improvement if we wrote our own error handling function. But you can do that.

Then we initialise each element of the array to *NULL*.

Finally, the pointers to functions that are passed to *newSet()* as argument values are used to initialise the pointer to function members.

6.4.4 Delete Set

Our implementation here is a bit naive. We require the set to be empty. We should write code to empty it in case it contained elements. But again, that is another improvement you could make.

```
/* deleteSet: erases the given set, which must be empty */
Set deleteSet(Set *s)
{
  assert(setIsEmpty(*s));
  free(*s);
  *s = NULL;
  return *s;
}
```

6.4.5 Empty Set

A set is empty if every element in the array *hashTable* has a *NULL* value.

```
/* setIsEmpty: returns 1 if the set has no elements */
int setIsEmpty(const Set s)
{
  int i;
  for (i = 0; i < HASHSIZE; i++)
    if (s->hashTable[i] != NULL)
      return 0;
  return 1;
}
```

6.4.6 Hash Function

The hash function converts its string parameter into an integer constrained to fall within the limits of the hash table defined above, the limits being *0..HASHSIZE-1*

The intention of writers of hash functions is to spread values throughout the hash table keeping linked lists to a minimum. A browse on the Internet will reveal several useful hash functions of varying complexity and effectiveness. But we shall lift the one found in Kernighan and Ritchie *The C Programming Language* Second Edition page 144.

```
/* hash: returns a hash value for the given string */
unsigned hash(char *s)
{
  unsigned hashVal;
  for (hashVal = 0; *s != '\0'; s++)
    hashVal = *s + 31 * hashVal;
  return hashVal % HASHSIZE;
}
```

If the string parameter had just one character, "d" say, then the *hashValue* is just its ASCII value, 100. Otherwise we look at each character in the string in turn and add the preceding hash value x 31 to the ASCII value of the current character. We use prime numbers as multipliers to help spread the entries across the hash table. Modulo *HASHSIZE* arithmetic ensures that the value returned is an index in the hash table.

You might use the values *c*, *d*, *e* and *f* to occupy both ends of the hash table when testing functions to add and delete elements from a set.

6.4.7 Membership

Before we can add a new object to a set we are obliged to see if it is already there (because sets cannot harbour duplicates).

For each element in the hash table we march a pointer along the linked list (if any) looking for a match. We use *equalObjects()* to check for equality. An *equalObjects()* function was written in *testset.c*, and a pointer to it was passed to *newSet()*.

```
/* setContains: returns 1 if the given element is in the set */
int setContains(const Set s, void * const obj)
{
  struct SetElement *p;
  unsigned hashVal = hash(s->objectKey(obj));
  for (p = s->hashTable[hashVal]; p != NULL; p = p->next)
    if (s->equalObjects(obj, p->element) == 1)
      return 1;
  return 0;
}
```

6.4.8 Add an Object

includeInSet() adds the given object to the set, but only if it is not already in the set in the first place. If the given object is already in the set, there is nothing to do, so we return the set unchanged.

```
/* includeInSet: adds the given object to the set if
   it is not already there */
Set includeInSet(const Set s, void * const obj)
{
  assert(s != NULL);
  if (setContains(s, obj))
    return s;
  Set t = copySet(s);
  unsigned hashVal = hash(s->objectKey(obj));
  struct SetElement *p;
  p = malloc(sizeof(struct SetElement));
  assert(p != NULL);
  p->next = t->hashTable[hashVal];
  p->element = t->duplicateObject(obj);
  t->hashTable[hashVal] = p;
  return t;
}
```

We obtain the given object's key (a string comprising its significant fields) and then its hash value because that determines the location in the array where it is to be located.

We *malloc* a hole in memory for a set element, *p*. We set its *next* element pointer to whatever is contained in its array location. This would be *NULL* if there is no element there, or the address of the element if (at least) one is already there.

Finally, we create a duplicate copy of the object (you may remember that the duplicate function was provided in *testset.c* and a pointer to it was passed to *newSet()*) and return the updated set.

Notice that *includeInSet()* makes a call to *copySet()*.

6.4.9 Copy Set

We make a duplicate copy of a given set.

```
/* copySet: returns a duplicate copy of the given set */
Set copySet(const Set s)
{
  Set t = newSet(s->equalObjects, s->duplicateObject, s->objectKey);
  struct SetElement *p;
  int i;
  for (i = 0; i < HASHSIZE; i++)
    for (p = s->hashTable[i]; p != NULL; p = p->next)
      t = includeInSet(t, p->element);
  return t;
}
```

For each non-null table element we copy each linked list element to the new table, *t*.

6.4.10 Remove an Object

We remove a given element from a set, but only if it is there in the first place.

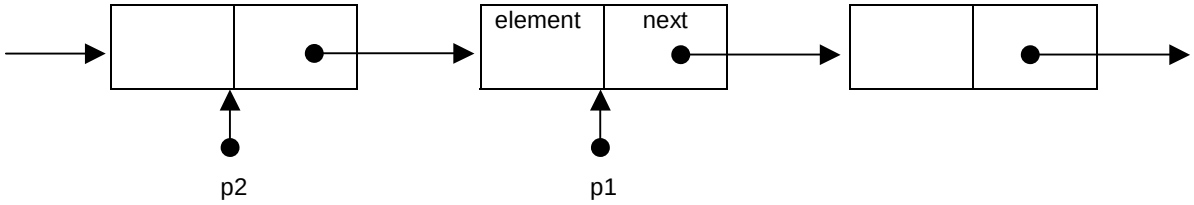
```
/* removeFromSet: removes the given element from the set */
Set removeFromSet(const Set s, void *const element)
{
  assert(s != NULL);
  assert(element != NULL);
  unsigned hashVal = hash(s->objectKey(element));
  struct SetElement *p1, *p2;
  if (!setContains(s, element))
    return s;
  Set t = copySet(s);
  for (p1 = t->hashTable[hashVal], p2 = NULL;
       p1 != NULL;
       p2 = p1, p1 = p1->next) {
    if (t->equalObjects(element, p1->element) == 1) { /*match found*/
      free(p1->element);
      p1->element = NULL;
      if (p2 == NULL)  /* at the beginning */
        t->hashTable[hashVal] = p1->next;
      else  /* in the middle or at the end */
        p2->next = p1->next;
      t->hashTable[hashVal] = NULL;
      free(p1);
      p1 = NULL;
      return t;
    }
  }
  return t;
}
```

If the set does not contain the object we wish to remove, there is nothing to do; return the set unchanged.
We apply our hash function to the given object's key, and proceed directly to its position in the hash table.

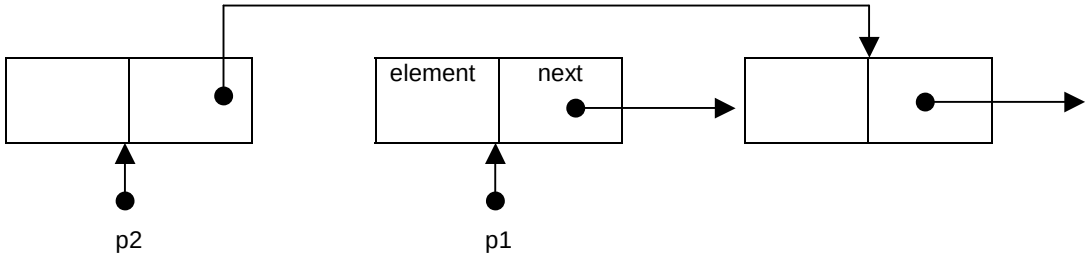
We aim to march a pair of pointers, *p1* and *p2*, along the linked list (if any). *p2* is immediately behind *p1*.

If the element pointed to by *p1* is identical to the object to be removed, we dispose of the element and set its pointer in the hash table to whatever *p1->next* contains. If this was the only element in the list, that would be *NULL*, if not, it would be the address of the next element in the list.

If we are in the middle of the list, we adjust *p2->next* to bypass the node to be deleted.




```
p2->next = p1->next;
```



Finally, we dispose of the node pointed to by *p1*.

6.4.11 Iterator

We use an iterator to visit each element in the set in turn. *newSetIterator()* returns a new iterator for the given set.

```
/* newSetIterator: returns a new iterator for the given set */
SetIterator newSetIterator(const Set s)
{
  SetIterator iter = malloc(sizeof(struct SetIteratorStruct));
  assert(iter != NULL);
  assert(s != NULL);
  iter->current = NULL;
  iter->index = -1;
  iter->set = s;
  return iter;
}
```

iter->current refers to the current node being visited. *iter->index* refers to the array element being processed.

We need to determine that there is a next element to move on to.

```
/* setIteratorHasNext: returns 1 if the set has an element not
   yet visited in the current iteration */
int setIteratorHasNext(const SetIterator iter)
{
  int i;
  /* looks for the next element in a list */
  if (iter->current != NULL && iter->current->next != NULL) {
    return 1;
  }
  /* looks for the next non-null table element */
  for (i = iter->index + 1; i < HASHSIZE; i++) {
    if (iter->set->hashTable[i] != NULL) {
      return 1;
    }
  }
  return 0;
}
```

If we are in the middle of a list, we look to see if there is another node to move on to. If not we look for the next non-null element in the array.

setIteratorNext() returns the next element in the set.

```
/* setIteratorNext: goes on to the next element and returns it,
   updates iter->current and iter->index */
void *setIteratorNext(SetIterator iter)
{
  int i;
  assert(setIteratorHasNext(iter) != 0);
  /* returns next element in a linked list */
  if (iter->current != NULL && iter->current->next != NULL) {
    iter->current = iter->current->next;
    return iter->set->duplicateObject(iter->current->element);
  }
  /* finds next non-null element */
  for (i = iter->index + 1; i < HASHSIZE; i++) {
    if (iter->set->hashTable[i] != NULL) {
      iter->current = iter->set->hashTable[i];
      iter->index = i;
      break;
    }
  }
  assert(iter->current != NULL);
  return iter->set->duplicateObject(iter->current->element);
}
```

If we are in the middle of a list, we update *iter->current* to point to the next element and then return it. If we are in the table we look for the next non-null element and return it.

6.4.12 Union

The union of two sets is a new set containing all the elements in both sets.

```
/* setUnion: returns a new set containing all the elements
   from two sets, s and t */
Set setUnion(const Set s, const Set t)
{
  Set u = copySet(s);
  SetIterator it = newSetIterator(t);
  while (setIteratorHasNext(it))
    u = includeInSet(u, setIteratorNext(it));
  return u;
}
```

We copy one of the given sets to the new set, then go through the second set element by element copying each element to the new set as we do so, remembering that our *includeInSet()* function does not duplicate elements already in the set.

6.4.13 Intersection

The intersection of two sets is a new set containing just the elements that are common to both sets.

```
/* setIntersection: returns a new set, u, containing all the elements
   common to two sets, s and t */
Set setIntersection(const Set s, const Set t)
{
  Set u = newSet(s->equalObjects, s->duplicateObject, s->objectKey);
  void *obj = NULL;
  SetIterator it = newSetIterator(s);
  while (setIteratorHasNext(it)) {
    obj = setIteratorNext(it);
    if (setContains(t, obj))
      u = includeInSet(u, obj);
  }
  return u;
}
```

We look at each element in one set in turn: if it is also in the second set we include it in our intersection.

6.4.14 Difference

The difference of two sets is a new set containing the elements that are in the first set but not in the second set. One way of accomplishing this is copy the entire contents of one set to a new set, then to remove from it each element in the second set.

```
/* setDifference: returns a new set with all the elements of set t
   removed from set s */
Set setDifference(const Set s, const Set t)
{
  Set u = copySet(s);
  void *obj = NULL;
  SetIterator it = newSetIterator(t);
  while (setIteratorHasNext(it)) {
    obj = setIteratorNext(it);
    if (setContains(u, obj))
      u = removeFromSet(u, obj);
  }
  return u;
}
```

6.4.15 Implementation

The entire implementation is shown below

```
/* set.c */

#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
#include "set.h"

#define HASHSIZE 101

struct SetElement {
  struct SetElement *next;
  void *element;
};

struct SetStruct {
  struct SetElement *hashTable[HASHSIZE];
  Equal equalObjects;
  Duplicate duplicateObject;
  Key objectKey;
};

struct SetIteratorStruct {
  struct SetElement *current;
  int index;
  Set set;
};
```

```
/* newSet: returns a new, empty set */
Set newSet(Equal equal, Duplicate duplicate, Key key)
{
  Set s = malloc(sizeof(struct SetStruct));
  assert(s != NULL);
  int i;
  for (i = 0; i < HASHSIZE; i++)
    s->hashTable[i] = NULL;
  s->equalObjects = equal;
  s->duplicateObject = duplicate;
  s->objectKey = key;
  return s;
}

/* deleteSet: erases the given set, which must be empty */
Set deleteSet(Set *s)
{
  assert(setIsEmpty(*s));
  free(*s);
  *s = NULL;
  return *s;
}

/* setIsEmpty: returns 1 if the set has no elements */
int setIsEmpty(const Set s)
{
  int i;
  for (i = 0; i < HASHSIZE; i++)
    if (s->hashTable[i] != NULL)
      return 0;
  return 1;
}

/* copySet: returns a duplicate copy of the given set */
Set copySet(const Set s)
{
  Set t = newSet(s->equalObjects, s->duplicateObject, s->objectKey);
  struct SetElement *p;
  int i;
  for (i = 0; i < HASHSIZE; i++)
    for (p = s->hashTable[i]; p != NULL; p = p->next)
      t = includeInSet(t, p->element);
  return t;
}

/* hash: returns a hash value for the given string */
unsigned hash(const char *s)
{
  unsigned hashVal;
  for (hashVal = 0; *s != '\0'; s++)
    hashVal = *s + 31 * hashVal;
  return hashVal % HASHSIZE;
}
```

```
/* includeInSet: adds the given object to the set if
   it is not already there */
Set includeInSet(const Set s, void * const obj)
{
  assert(s != NULL);
  if (setContains(s, obj))
    return s;
  Set t = copySet(s);
  unsigned hashVal = hash(s->objectKey(obj));
  struct SetElement *p;
  p = malloc(sizeof(struct SetElement));
  assert(p != NULL);
  p->next = t->hashTable[hashVal];
  p->element = t->duplicateObject(obj);
  t->hashTable[hashVal] = p;
  return t;
}

/* removeFromSet: removes the given element from the set */
Set removeFromSet(const Set s, void *const element)
{
  assert(s != NULL);
  assert(element != NULL);
  unsigned hashVal = hash(s->objectKey(element));
  struct SetElement *p1, *p2;
  if (!setContains(s, element))
    return s;
  Set t = copySet(s);
  for (p1 = t->hashTable[hashVal], p2 = NULL;
       p1 != NULL;
       p2 = p1, p1 = p1->next) {
    if (t->equalObjects(element, p1->element) == 1) { /*match found*/
      free(p1->element);
      p1->element = NULL;
      if (p2 == NULL)  /* at the beginning */
        t->hashTable[hashVal] = p1->next;
      else  /* in the middle or at the end */
        p2->next = p1->next;
      t->hashTable[hashVal] = NULL;
      free(p1);
      p1 = NULL;
      return t;
    }
  }
  return t;
}

/* setContains: returns 1 if the given element is in the set */
int setContains(const Set s, void * const obj)
{
  struct SetElement *p;
  unsigned hashVal = hash(s->objectKey(obj));
  for (p = s->hashTable[hashVal]; p != NULL; p = p->next)
    if (s->equalObjects(obj, p->element) == 1)
      return 1;
  return 0;
}
```

```
/* newSetIterator: returns a new iterator for the given set */
SetIterator newSetIterator(const Set s)
{
  SetIterator iter = malloc(sizeof(
                  struct SetIteratorStruct));
  assert(iter != NULL);
  assert(s != NULL);
  iter->current = NULL;
  iter->index = -1;
  iter->set = s;
  return iter;
}

/* setIteratorHasNext: returns 1 if the set has an element not
   yet visited in the current iteration */
int setIteratorHasNext(const SetIterator iter)
{
  int i;
  /* looks for the next element in a list */
  if (iter->current != NULL && iter->current->next != NULL) {
    return 1;
  }
  /* looks for the next non-null table element */
  for (i = iter->index + 1; i < HASHSIZE; i++) {
    if (iter->set->hashTable[i] != NULL) {
      return 1;
    }
  }
  return 0;
}

/* setIteratorNext: goes on to the next element and returns it,
   updates iter->current and iter->index */
void *setIteratorNext(SetIterator iter)
{
  int i;
  assert(setIteratorHasNext(iter) != 0);
  /* returns next element in a linked list */
  if (iter->current != NULL && iter->current->next != NULL) {
    iter->current = iter->current->next;
    return iter->set->duplicateObject(iter->current->element);
  }
  /* finds next non-null element */
  for (i = iter->index + 1; i < HASHSIZE; i++) {
    if (iter->set->hashTable[i] != NULL) {
      iter->current = iter->set->hashTable[i];
      iter->index = i;
      break;
    }
  }
  assert(iter->current != NULL);
  return iter->set->duplicateObject(iter->current->element);
}
```

```
/* setUnion: returns a new set containing all the elements
   from two sets, s and t */
Set setUnion(const Set s, const Set t)
{
  Set u = copySet(s);
  SetIterator it = newSetIterator(t);
  while (setIteratorHasNext(it))
    u = includeInSet(u, setIteratorNext(it));
  return u;
}

/* setIntersection: returns a new set, u, containing all the elements
   common to two sets, s and t */
Set setIntersection(const Set s, const Set t)
{
  Set u = newSet(s->equalObjects, s->duplicateObject, s->objectKey);
  void *obj = NULL;
  SetIterator it = newSetIterator(s);
  while (setIteratorHasNext(it)) {
    obj = setIteratorNext(it);
    if (setContains(t, obj))
      u = includeInSet(u, obj);
  }
  return u;
}

/* setDifference: returns a new set with all the elements of set t
   removed from set s */
Set setDifference(const Set s, const Set t)
{
  Set u = copySet(s);
  void *obj = NULL;
  SetIterator it = newSetIterator(t);
  while (setIteratorHasNext(it)) {
    obj = setIteratorNext(it);
    if (setContains(u, obj))
      u = removeFromSet(u, obj);
  }
  return u;
}
```

Exercise 6.1

1. The cardinality of a set is the number of elements it contains. Write and test a function that returns the cardinality of a set.

2. Set A is a subset of set B if every element in A is also in B. Write and test a function that returns whether a set is a subset of another set.

3. Two sets are equal if they have precisely the same elements. Write and test a function that returns whether two given sets are identical.

Bibliography

Kernighan B, Ritchie D, *The C Programming language*, Prentice Hall, 1988
Kernighan B, Plauger P, Software Tools in Pascal Addison-Wesley 1981 p153
Mark Williams Company, *ANSI C - A lexical Guide*, Prentice Hall 1988
Heathfield, R, Pietsch G *http://users.power.net.co.uk/eton/kandr2/krx605.html* July 2010
Haendel L *The function Pointer Tutorials http:*.//www.newty.de accessed August 2010