COMSM1201 : Data Structures & Algorithms

Dr. Neill Campbell
Neill.Campbell@bristol.ac.uk

University of Bristol

November 11, 2021

Table of Contents

Collections

- One of the simplest ADTs is the **Collection**.
- This is just a simple place to search for/add/delete data elements.
- Some collections allow duplicate elements and others do not (e.g. Sets).
- Some are ordered (for faster searching) and others unordered.
- Our Collection will be unsorted and will allow duplicates.

```
1  #include "../General/general.h"
2
3  typedef int colltype;
4
5  typedef struct coll coll;
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <assert.h>
10
11 // Create an empty coll
12 coll* coll_init(void);
13 // Add element onto top
14 void coll_add(coll* c, colltype i);
15 // Take element out
16 bool coll_remove(coll* c, colltype d);
17 // Does this exist ?
18 bool coll_isin(coll* c, colltype i);
19 // Return size of coll
20 int coll_size(coll* c);
21 // Clears all space used
22 bool coll_free(coll* c);
```

Collection ADT

- Note that the interface gives you no hints as to the actual underlying implementation of the ADT.
- A user of the ADT doesn't really need to know how it's implemented - ideally.
- The ADT developer could have several **different** implementations.
- Here we'll see *Collection* implemented using:
  - A fixed-size array
  - A dynamic array
  - A linked-list

Fixed/specific.h:

```
1  #pragma once
2
3  #define COLLTYPE "Fixed"
4
5  #define FIXEDSIZE 5000
6  struct coll {
7     // Underlying array
8     colltype a[FIXEDSIZE];
9     int size;
10 };
```

Collection ADT using a Fixed-size Array

Fixed/fixed.c:

```c
#include "../coll.h"
#include "specific.h"

coll* coll_init(void)
{
    coll* c = (coll*) ncalloc(sizeof(coll), 1);
    c->size = 0;
    return c;
}

int coll_size(coll* c)
{
    if(c==NULL){
        return 0;
    }
    return c->size;
}

bool coll_isin(coll* c, colltype  d)
{
    for(int i=0; i<coll_size(c); i++){
        if(c->a[i] == d){
            return true;
        }
    }
    return false;
}
```

```c
void coll_add(coll* c, colltype  d)
{
    if(c){
        c->a[c->size] = d;
        c->size = c->size + 1;
        if(c->size >= FIXEDSIZE){
            on_error("Collection overflow");
        }
    }
}

bool coll_remove(coll* c, colltype d)
{
    for(int i=0; i<coll_size(c); i++){
        if(c->a[i] == d){
            // Shuffle end of array left one
            for(int j=i; j<coll_size(c); j++){
                c->a[j] = c->a[j+1];
            }
            c->size = c->size - 1;
            return true;
        }
    }
    return false;
}

bool coll_free(coll* c)
{
    free(c);
    return true;
}
```

Collection ADT via an Array (Realloc)

Realloc/specific.h:

```
1   #pragma once
2
3   #define COLLTYPE "Realloc"
4
5   #define FIXEDSIZE 16
6   #define SCALEFACTOR 2
7   struct coll {
8      // Underlying array
9      colltype* a;
10     int size;
11     int capacity;
12  };
```

Realloc/realloc.c:

```
1   #include "../coll.h"
2   #include "specific.h"
3
4   coll* coll_init(void)
5   {
6      coll* c = (coll*) ncalloc(sizeof(coll), 1);
7      c->a = (colltype*) ncalloc(sizeof(colltype), FIXEDSIZE);
8      c->size = 0;
9      c->capacity= FIXEDSIZE;
10     return c;
11  }
12
13  void coll_add(coll* c, colltype d)
14  {
15     if(c){
16        c->a[c->size] = d;
17        c->size = c->size + 1;
18        if(c->size >= c->capacity){
19           c->a = (colltype*) nremalloc(c->a,
20                  sizeof(colltype)*c->capacity*SCALEFACTOR);
21           c->capacity = c->capacity*SCALEFACTOR;
22        }
23     }
```

Collection ADT via a Linked List

Linked/specific.h:

```
1  #pragma once
2
3  #define COLLTYPE "Linked"
4
5  struct dataframe {
6     colltype i;
7     struct dataframe* next;
8  };
9  typedef struct dataframe dataframe;
10
11 struct coll {
12    // Underlying array
13    dataframe* start;
14    int size;
15 };
```

Linked/linked.c:

```
#include "../coll.h"
#include "specific.h"

coll* coll_init(void)
{
   coll* c = (coll*) ncalloc(sizeof(coll), 1);
   return c;
}

int coll_size(coll* c)
{
   if(c==NULL){
      return 0;
   }
   return c->size;
}

bool coll_isin(coll* c, colltype d)
{
   if(c == NULL || c->start==NULL){
      return false;
   }
   dataframe* f = c->start;
   do{
      if(f->i == d){
         return true;
      }
      f = f->next;
   }while(f != NULL);
   return false;
}
```

Collection ADT via a Linked List II

```
void coll_add(coll* c, colltype d)
{
   if(c){
      dataframe* f = ncalloc(sizeof(dataframe), 1);
      f->i = d;
      f->next = c->start;
      c->start = f;
      c->size = c->size + 1;
   }

}

bool coll_free(coll* c)
{
   if(c){
      dataframe* tmp;
      dataframe* p = c->start;
      while(p!=NULL){
         tmp = p->next;
         free(p);
         p = tmp;
      }
      free(c);
   }
   return true;
}
```

```
bool coll_remove(coll* c, colltype d)
{
   dataframe* f1, *f2;
   if((c==NULL) || (c->start==NULL)){
      return false;
   }

   // If Front
   if(c->start->i == d){
      f1 = c->start->next;
      free(c->start);
      c->start = f1;
      c->size = c->size - 1;
      return true;
   }

   f1 = c->start;
   f2 = c->start->next;
   do{
      if(f2->i == d){
         f1->next = f2->next;
         free(f2);
         c->size = c->size - 1;
         return true;
      }
      f1 = f2;
      f2 = f1->next;
   }while(f2 != NULL);
   return false;
}
```

Collection Summary

- Any code using the ADT can be compiled against any of the implementations, e.g. the test (`testcoll.c`) code.
- The *Collection* interface (`coll.h`) is never changed.
- There are pros and cons of each implementation:
  - Fixed Array : Simple to implement - can't avoid the problems of it being a fixed-size. Deletion expensive.
  - Realloc Array : Implementation fairly simple. Deletion expensive. Every realloc() is very **expensive**. Need to tune SCALEFACTOR.
  - Linked : Slightly fiddly implementation - fast to delete an element.

Task	Fixed Array	Realloc Array	Linked List
Insert new element	O(1) at end *if space*	O(1) at end *but realloc()*	O(1) at front
Search for an element	O(n) *brute force*	O(n) *brute force*	O(n) *brute force*
Search + delete	O(n) + O(n) *move left*	O(n) + O(n) *move left*	O(n) + O(1) *delete 'free'*

- If we had ordered our ADT (ie. the elements were sorted), then the searches could be via a binary / interpolation search, leading to O(log n) or O(log log n) search times.

ADTs Making Coding Simpler

That Linked List code from the previous Chapter again:

```c
#include "coll.h"
#include "Fixed/specific.h"

int main(void)
{
    coll* c;
    int i;

    printf("Please type some numbers :");
    c = coll_init();
    while(scanf("%i", &i) == 1){
       coll_add(c, i);
    }
    // Do print etc.
    coll_free(c);
    return 0;
}
```

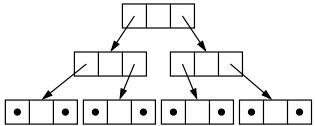
Table of Contents

ADTs

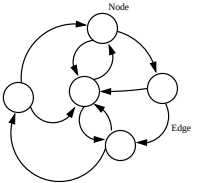
At the highest level of abstraction, ADTs that we can represent using both dynamic structures (pointers) and also fixed structures (arrays) include:

- Collections (Lists)
- Stacks
- Queues
- Sets
- Graphs
- Trees

Binary Trees:

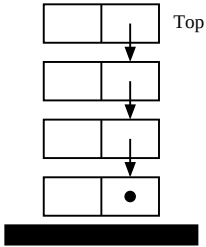


Unidirectional Graph:

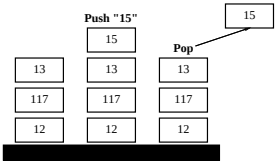


Stacks

The push-down stack:



LIFO (Last in, First out):



- Operations include `push` and `pop`.
- In the C run-time system, function calls are implemented using stacks.
- Most recursive algorithms can be re-written using stacks instead.
- But, once again, we are faced with the question : How best to implement such a data type ?

ADT:Stacks Arrays (Realloc) I

stack.h:

```c
#pragma once

#include "../General/general.h"

typedef int stacktype;

typedef struct stack stack;

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

/* Create an empty stack */
stack* stack_init(void);
/* Add element to top */
void stack_push(stack* s, stacktype i);
/* Take element from top */
bool stack_pop(stack* s, stacktype* d);
/* Clears all space used */
bool stack_free(stack* s);

/* Optional? */

/* Copy top element into d (but don't pop it) */
bool stack_peek(stack*s,  stacktype* d);
/* Make a string version - keep .dot in mind */
void stack_tostring(stack*, char* str);
```

Realloc/specific.h:

```c
#pragma once

#define FORMATSTR "%d"
#define ELEMSIZE 20

#define STACKTYPE "Realloc"

#define FIXEDSIZE 16
#define SCALEFACTOR 2

struct stack {
   /* Underlying array */
   stacktype* a;
   int size;
   int capacity;
};
```

ADT:Stacks Arrays (Realloc) II

Realloc/realloc.c

```c
#include "../stack.h"
#include "specific.h"

#define DOTFILE 5000

stack* stack_init(void)
{
   stack *s = (stack*) ncalloc(sizeof(stack), 1);
   /* Some implementations would allow you to pass
      a hint about the initial size of the stack */
   s->a = (stacktype*) ncalloc(sizeof(stacktype), FIXEDSIZE);
   s->size = 0;
   s->capacity= FIXEDSIZE;
   return s;
}

void stack_push(stack* s, stacktype d)
{
   if(s==NULL){
       return;
   }
   s->a[s->size] = d;
   s->size = s->size + 1;
   if(s->size >= s->capacity){
      s->a = (stacktype*) nremalloc(s->a,
            sizeof(stacktype)*s->capacity*SCALEFACTOR);
      s->capacity = s->capacity*SCALEFACTOR;
   }
}
```

```c
bool stack_pop(stack* s, stacktype* d)
{
   if((s == NULL) || (s->size < 1)){
      return false;
   }
   s->size = s->size - 1;
   *d = s->a[s->size];
   return true;
}

bool stack_peek(stack* s, stacktype* d)
{
   if((s==NULL) || (s->size <= 0)){
      /* Stack is Empty */
      return false;
   }
   *d = s->a[s->size-1];
   return true;
}
```

ADT:Stacks Arrays (Realloc) III

Realloc/realloc.c

```
1   void stack_tostring(stack* s, char* str)
2   {
3       char tmp[ELEMSIZE];
4       str[0] = '\0';
5       if((s==NULL) || (s->size <1)){
6           return;
7       }
8       for(int i=s->size-1; i>=0; i--){
9           sprintf(tmp, FORMATSTR, s->a[i]);
10          strcat(str, tmp);
11          strcat(str, "|");
12      }
13      str[strlen(str)-1] = '\0';
14  }
15
16  bool stack_free(stack* s)
17  {
18      if(s==NULL){
19          return true;
20      }
21      free(s->a);
22      free(s);
23      return true;
24  }
```

- We need a thorough testing program teststack.c
- See also revstr.c : a version of the string reverse code (for which we already seen an iterative (in-place) and a recursive solution).

ADT:Stacks Linked I

Linked/specific.h

```
1  #pragma once
2
3  #define FORMATSTR "%d"
4  #define ELEMSIZE 20
5  #define STACKTYPE "Linked"
6
7  struct dataframe {
8     stacktype i;
9     struct dataframe* next;
10 };
11 typedef struct dataframe dataframe;
12
13 struct stack {
14    /* Underlying array */
15    dataframe* start;
16    int size;
17 };
```

Linked/linked.c

```
1  #include "../stack.h"
2  #include "specific.h"
3
4  #define DOTFILE 5000
5
6  stack* stack_init(void)
7  {
8     stack* s = (stack*) ncalloc(sizeof(stack), 1);
9     return s;
10 }
11
12 void stack_push(stack* s, stacktype d)
13 {
14    if(s){
15       dataframe* f = ncalloc(sizeof(dataframe), 1);
16       f->i = d;
17       f->next = s->start;
18       s->start = f;
19       s->size = s->size + 1;
20    }
21 }
```

ADT:Stacks Linked II

```
1   bool stack_pop(stack* s, stacktype* d)
2   {
3      if((s==NULL) || (s->start==NULL)){
4         return false;
5      }
6
7      dataframe* f = s->start->next;
8      *d = s->start->i;
9      free(s->start);
10     s->start = f;
11     s->size = s->size - 1;
12     return true;
13  }
14
15  bool stack_peek(stack* s, stacktype* d)
16  {
17     if((s==NULL) || (s->start==NULL)){
18        return false;
19     }
20     *d = s->start->i;
21     return true;
22  }
```

```
1   void stack_tostring(stack* s, char* str)
2   {
3      char tmp[ELEMSIZE];
4      str[0] = '\0';
5      if((s==NULL) || (s->size <1)){
6         return;
7      }
8      dataframe* p = s->start;
9      while(p){
10        sprintf(tmp, FORMATSTR, p->i);
11        strcat(str, tmp);
12        strcat(str, "|");
13        p = p->next;
14     }
15     str[strlen(str)-1] = '\0';
16  }
17
18  bool stack_free(stack* s)
19  {
20     if(s){
21        dataframe* p = s->start;
22        while(p!=NULL){
23           dataframe* tmp = p->next;
24           free(p);
25           p = tmp;
26        }
27        free(s);
28     }
29     return true;
30  }
```