COMSM1201 : Data Structures & Algorithms

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

University of Bristol

November 26, 2021
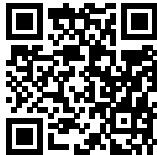
Table of Contents

Simple Recursion

- When a function calls itself, this is known as recursion.

Simple Recursion

- When a function calls itself, this is known as recursion.
- This is an important theme in Computer Science that crops up time & time again.

Simple Recursion

- When a function calls itself, this is known as recursion.
- This is an important theme in Computer Science that crops up time & time again.
- Can sometimes lead to very simple and elegant programs.

Simple Recursion

- When a function calls itself, this is known as recursion.
- This is an important theme in Computer Science that crops up time & time again.
- Can sometimes lead to very simple and elegant programs.
- Let's look at some toy examples to begin with.

Simple Recursion

- When a function calls itself, this is known as recursion.
- This is an important theme in Computer Science that crops up time & time again.
- Can sometimes lead to very simple and elegant programs.
- Let's look at some toy examples to begin with.

Simple Recursion

- When a function calls itself, this is known as recursion.
- This is an important theme in Computer Science that crops up time & time again.
- Can sometimes lead to very simple and elegant programs.
- Let's look at some toy examples to begin with.

```c
1   #include <stdio.h>
2   #include <string.h>
3
4   #define SWAP(A,B) {char temp; temp=A;A=B;B=temp;}
5
6   void strrev(char* s, int n);
7
8   int main(void)
9   {
10      char str[] = "Hello World!";
11      strrev(str, strlen(str));
12      printf("%s\n", str);
13      return 0;
14  }
15
16  /* Iterative Inplace String Reverse */
17  void strrev(char* s, int n)
18  {
19      for(int i=0, j=n-1; i<j; i++, j--){
20          SWAP(s[i], s[j]);
21      }
22  }
```

Execution :

```
!dlroW olleH
```

Recursion for *strrev()*

```c
1    #include <stdio.h>
2    #include <string.h>
3
4    #define SWAP(A,B) {char temp; temp=A;A=B;B=temp;}
5
6    void strrev(char* s, int start, int end);
7
8    int main(void)
9    {
10       char str[] = "Hello World!";
11       strrev(str, 0, strlen(str)-1);
12       printf("%s\n", str);
13       return 0;
14   }
15
16   /* Recursive : Inplace String Reverse */
17   void strrev(char* s, int start, int end)
18   {
19       if(start >= end){
20           return;
21       }
22       SWAP(s[start], s[end]);
23       strrev(s, start+1, end-1);
24   }
```

Execution :

!dlroW olleH

Recursion for *strrev()*

```c
#include <stdio.h>
#include <string.h>

#define SWAP(A,B) {char temp; temp=A;A=B;B=temp;}

void strrev(char* s, int start, int end);

int main(void)
{
    char str[] = "Hello World!";
    strrev(str, 0, strlen(str)-1);
    printf("%s\n", str);
    return 0;
}

/* Recursive : Inplace String Reverse */
void strrev(char* s, int start, int end)
{
    if(start >= end){
        return;
    }
    SWAP(s[start], s[end]);
    strrev(s, start+1, end-1);
}
```

- We need to change the function prototype.

Execution :

!dlroW olleH

Recursion for *strrev()*

```c
1   #include <stdio.h>
2   #include <string.h>
3
4   #define SWAP(A,B) {char temp; temp=A;A=B;B=temp;}
5
6   void strrev(char* s, int start, int end);
7
8   int main(void)
9   {
10      char str[] = "Hello World!";
11      strrev(str, 0, strlen(str)-1);
12      printf("%s\n", str);
13      return 0;
14  }
15
16  /* Recursive : Inplace String Reverse */
17  void strrev(char* s, int start, int end)
18  {
19      if(start >= end){
20          return;
21      }
22      SWAP(s[start], s[end]);
23      strrev(s, start+1, end-1);
24  }
```

Execution :

!dlroW olleH

- We need to change the function prototype.
- This allows us to track both the start and the end of the string.

The Fibonacci Sequence

A well known example of a recursive function is the Fibonacci sequence. The first term is 1, the second term is 1 and each successive term is defined to be the sum of the two previous terms, i.e. :

fib(1) is 1
fib(2) is 1
fib(n) is fib(n-1)+fib(n-2)

1,1,2,3,5,8,13,21, ...

Iterative & Recursive Fibonacci

```c
#include <stdio.h>

#define MAXFIB 24

int fibonacci(int n);

int main(void)
{

    for(int i=1; i<=MAXFIB; i++){
        printf("%d = %d\n", i, fibonacci(i));
    }

    return 0;

}

int fibonacci(int n)
{
    if(n <= 2){
        return 1;
    }
    int a = 1;
    int b = 1;
    int next;
    for(int i=3; i<=n; i++){
        next = a + b;
        a = b;
        b = next;
    }
    return b;
}
```

Iterative & Recursive Fibonacci

```c
#include <stdio.h>

#define MAXFIB 24

int fibonacci(int n);

int main(void)
{

    for(int i=1; i<=MAXFIB; i++){
        printf("%d = %d\n", i, fibonacci(i));
    }

    return 0;

}

int fibonacci(int n)
{
    if(n <= 2){
        return 1;
    }
    int a = 1;
    int b = 1;
    int next;
    for(int i=3; i<=n; i++){
        next = a + b;
        a = b;
        b = next;
    }
    return b;
}
```

Execution :

```
1  = 1
2  = 1
3  = 2
4  = 3
5  = 5
6  = 8
7  = 13
8  = 21
9  = 34
10 = 55
11 = 89
12 = 144
13 = 233
14 = 377
15 = 610
16 = 987
17 = 1597
18 = 2584
19 = 4181
20 = 6765
21 = 10946
22 = 17711
23 = 28657
24 = 46368
```

Iterative & Recursive Fibonacci

```c
1    #include <stdio.h>
2
3    #define MAXFIB 24
4
5    int fibonacci(int n);
6
7    int main(void)
8    {
9
10       for(int i=1; i<=MAXFIB; i++){
11           printf("%d = %d\n", i, fibonacci(i));
12       }
13
14       return 0;
15
16   }
17
18   int fibonacci(int n)
19   {
20       if(n == 1) return 1;
21       if(n == 2) return 1;
22       return( fibonacci(n-1)+fibonacci(n-2));
23   }
```

Iterative & Recursive Fibonacci

```c
1   #include <stdio.h>
2
3   #define MAXFIB 24
4
5   int fibonacci(int n);
6
7   int main(void)
8   {
9
10      for(int i=1; i<=MAXFIB; i++){
11          printf("%d = %d\n", i, fibonacci(i));
12      }
13
14      return 0;
15
16  }
17
18  int fibonacci(int n)
19  {
20      if(n == 1) return 1;
21      if(n == 2) return 1;
22      return( fibonacci(n-1)+fibonacci(n-2));
23  }
```

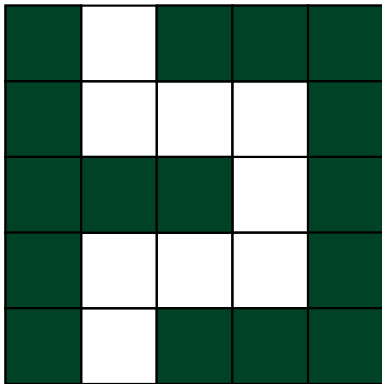It's interesting to see how run-time increases as the length of the sequence is raised.
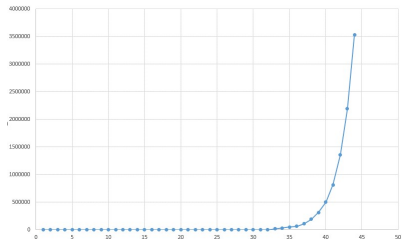
Maze Escape

The correct route through a maze can be obtained via recursive, rather than iterative, methods.
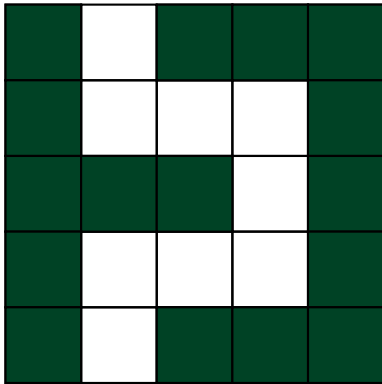
Maze Escape

The correct route through a maze can be obtained via recursive, rather than iterative, methods.


```
bool explore(int x, int y, char mz[YS][XS])
{
  if  mz[y][x] is exit return true;

  Mark mz[y][x] so we don't return here

  if we can go up :
    if(explore(x, y+1, mz)) return true

  if we can go right :
    if(explore(x+1, y, mz)) return true

  Do left & down in a similar manner

  return false; // Failed to find route
}
```

Permuting

- Here we consider the ways to permute a string (or more generally an array)

Permuting

- Here we consider the ways to permute a string (or more generally an array)
- Permutations are all possible ways of rearranging the positions of the characters.

  Execution :

  ABC
  ACB
  BAC
  BCA
  CBA
  CAB

Permuting

- Here we consider the ways to permute a string (or more generally an array)
- Permutations are all possible ways of rearranging the positions of the characters.

  Execution :

  ABC
  ACB
  BAC
  BCA
  CBA
  CAB

Permuting

- Here we consider the ways to permute a string (or more generally an array)
- Permutations are all possible ways of rearranging the positions of the characters.

Execution :

ABC
ACB
BAC
BCA
CBA
CAB

```c
1   // From e.g.  http://www.geeksforgeeks.org
2   #include <stdio.h>
3   #include <string.h>
4
5   #define SWAP(A,B) {char temp = *A; *A = *B; *B = temp;}
6
7   void permute(char* a, int s, int e);
8
9   int main()
10  {
11      char str[] = "ABC";
12      int n = strlen(str);
13      permute(str, 0, n-1);
14      return 0;
15  }
16
17  void permute(char* a, int s, int e)
18  {
19      if (s == e){
20        printf("%s\n", a);
21        return;
22      }
23      for (int i = s; i <= e; i++){
24        SWAP((a+s), (a+i)); // Bring one char to the front
25        permute(a, s+1, e);
26        SWAP((a+s), (a+i)); // Backtrack
27      }
28  }
```

Self-test : Power

- Raising a number to a power n = 2^5 is
  the same as multiple multiplications
  n = 2*2*2*2*2.

Self-test : Power

- Raising a number to a power $n = 2^5$ is the same as multiple multiplications $n = 2*2*2*2*2$.
- Or, thinking recursively, $n = 2 * (2^4)$.

Self-test : Power

- Raising a number to a power $n = 2\hat{\ }5$ is the same as multiple multiplications $n = 2*2*2*2*2$.
- Or, thinking recursively, $n = 2 * (2\hat{\ }4)$.

- Raising a number to a power $n = 2^5$ is the same as multiple multiplications $n = 2*2*2*2*2$.
- Or, thinking recursively, $n = 2 * (2^4)$.

```
1    /* Try to write power(a,b) to computer a^b
2       without using any maths functions other than
3       multiplication :
4       Try (1) iterative then (2) recursive
5       (3) Trick that for n%2==0, x^n = x^(n/2)*x^(n/2)
6
7    */
8
9    #include <stdio.h>
10
11   int power(unsigned int a, unsigned int b);
12
13   int main(void)
14   {
15
16       int x = 2;
17       int y = 16;
18
19       printf("%d^%d = %d\n", x, y, power(x,y));
20
21   }
22
23   int power(unsigned int a, unsigned int b)
24   {
25   }
```

Table of Contents

Sequential Search

- The need to search an array for a particular value is a common problem.

Sequential Search

- The need to search an array for a particular value is a common problem.
- This is used to delete names from a mailing list, or upgrading the salary of an employee etc.

Sequential Search

- The need to search an array for a particular value is a common problem.
- This is used to delete names from a mailing list, or upgrading the salary of an employee etc.
- The simplest method for searching is called the sequential search.

Sequential Search

- The need to search an array for a particular value is a common problem.
- This is used to delete names from a mailing list, or upgrading the salary of an employee etc.
- The simplest method for searching is called the sequential search.
- Simply move through the array from beginning to end, stopping when you have found the value you require.

Sequential Search

- The need to search an array for a particular value is a common problem.
- This is used to delete names from a mailing list, or upgrading the salary of an employee etc.
- The simplest method for searching is called the sequential search.
- Simply move through the array from beginning to end, stopping when you have found the value you require.

Sequential Search

- The need to search an array for a particular value is a common problem.
- This is used to delete names from a mailing list, or upgrading the salary of an employee etc.
- The simplest method for searching is called the sequential search.
- Simply move through the array from beginning to end, stopping when you have found the value you require.

```c
1   #include <stdio.h>
2   #include <string.h>
3   #include <assert.h>
4
5   #define NOTFOUND -1
6   #define NUMPEOPLE 6
7   typedef struct person{
8           char* name; int age;
9   } person;
10
11  int findAge(const char* name, const person* p, int n);
12
13  int main(void)
14  {
15      person ppl[NUMPEOPLE] = { {"Ackerby", 21}, {"Bloggs", 25},
16                      {"Chumley", 26}, {"Dalton",   25},
17                      {"Eggson", 22}, {"Fulton",  41} };
18
19      assert(findAge("Eggson",   ppl, NUMPEOPLE)==22);
20      assert(findAge("Campbell", ppl, NUMPEOPLE)==NOTFOUND);
21      return 0;
22  }
23
24  int findAge(const char* name, const person* p, int n)
25  {
26      for(int j=0; j<n; j++){
27          if(strcmp(name, p[j].name) == 0){
28              return p[j].age;
29          }
30      }
31      return NOTFOUND;
32  }
```

Sequential Search

- Sometimes our list of people may not be random.

Sequential Search

- Sometimes our list of people may not be random.
- If, for instance, it is sorted, we can use strcmp() in a slightly cleverer manner.

Sequential Search

- Sometimes our list of people may not be random.
- If, for instance, it is sorted, we can use strcmp() in a slightly cleverer manner.
- We can stop searching once the search key is alphabetically greater than the item at the current position in the list.

Sequential Search

- Sometimes our list of people may not be random.
- If, for instance, it is sorted, we can use strcmp() in a slightly cleverer manner.
- We can stop searching once the search key is alphabetically greater than the item at the current position in the list.
- This halves, on average, the number of comparisons required.

Sequential Search

- Sometimes our list of people may not be random.
- If, for instance, it is sorted, we can use strcmp() in a slightly cleverer manner.
- We can stop searching once the search key is alphabetically greater than the item at the current position in the list.
- This halves, on average, the number of comparisons required.

Sequential Search

- Sometimes our list of people may not be random.
- If, for instance, it is sorted, we can use strcmp() in a slightly cleverer manner.
- We can stop searching once the search key is alphabetically greater than the item at the current position in the list.
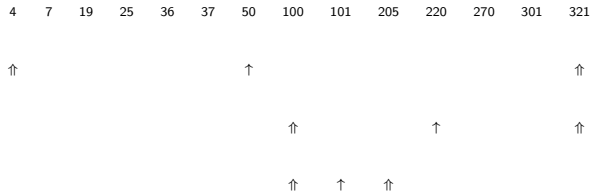- This halves, on average, the number of comparisons required.

```c
#include <stdio.h>
#include <string.h>
#include <assert.h>

#define NOTFOUND -1
#define NUMPEOPLE 6
typedef struct person{
        char* name; int age;
} person;

int findAge(const char* name, const person* p, int n);

int main(void)
{
    person ppl[NUMPEOPLE] = { {"Ackerby", 21}, {"Bloggs", 25},
                {"Chumley", 26}, {"Dalton",   25},
                {"Eggson", 22}, {"Fulton",  41} };

    assert(findAge("Eggson",   ppl, NUMPEOPLE)==22);
    assert(findAge("Campbell", ppl, NUMPEOPLE)==NOTFOUND);
    return 0;
}

int findAge(const char* name, const person* p, int n)
{
    for(int j=0; j<n; j++){
        int m = strcmp(name, p[j].name);
        if(m == 0) // Braces!
            return p[j].age;
        if(m < 0)
            return NOTFOUND;
    }
    return NOTFOUND;
}
```

Binary Search for *101*

- Searching small lists doesn't require much
  computation time.

```
4   7   19   25   36   37   50   100   101   205   220   270   301   321

⇑                               ↑                                 ⇑

                                    ⇑                 ↑           ⇑

                                ⇑      ↑      ⇑
```

Binary Search for *101*

- Searching small lists doesn't require much computation time.
- However, as lists get longer (e.g. phone directories), sequential searching becomes extremely inefficient.

4	7	19	25	36	37	50	100	101	205	220	270	301	321

⇑   ↑   ⇑

          ⇑   ↑   ⇑

       ⇑ ↑ ⇑

Binary Search for *101*

- Searching small lists doesn't require much computation time.
- However, as lists get longer (e.g. phone directories), sequential searching becomes extremely inefficient.
- A binary search consists of examining the middle element of the array to see if it has the desired value. If not, then half the array may be discarded for the next search.

4	7	19	25	36	37	50	100	101	205	220	270	301	321

⇑   ↑   ⇑

           ⇑   ↑   ⇑

         ⇑   ↑   ⇑

Binary Search for *101*

- Searching small lists doesn't require much computation time.
- However, as lists get longer (e.g. phone directories), sequential searching becomes extremely inefficient.
- A binary search consists of examining the middle element of the array to see if it has the desired value. If not, then half the array may be discarded for the next search.

4	7	19	25	36	37	50	100	101	205	220	270	301	321

⇑   ↑   ⇑

               ⇑   ↑   ⇑

               ⇑   ↑   ⇑

Binary Search for *101*

- Searching small lists doesn't require much computation time.
- However, as lists get longer (e.g. phone directories), sequential searching becomes extremely inefficient.
- A binary search consists of examining the middle element of the array to see if it has the desired value. If not, then half the array may be discarded for the next search.

| 4 | 7 | 19 | 25 | 36 | 37 | 50 | 100 | 101 | 205 | 220 | 270 | 301 | 321 |

⇑   ↑   ⇑

                  ⇑   ↑   ⇑

                  ⇑   ↑   ⇑

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>
#define NMBRS 1000000

int bin_it(int k, const int* a, int l, int r);

int main(void)
{
    int a[NMBRS];
    srand(time(NULL));

    // Put even numbers into array
    for(int i=0; i<NMBRS; i++){
        a[i] = 2*i;
    }

    // Do many searches for a random number
    for(int i=0; i<10*NMBRS; i++){
        int n = rand()%NMBRS;
        if((n%2) == 0){
            assert(bin_it(n, a, 0, NMBRS-1) == n/2);
        }
        else{ // No odd numbers in this list
            assert(bin_it(n, a, 0, NMBRS-1) < 0);
        }
    }
    return 0;
}
```

Iterative v. Recursion Binary Search

```
int bin_it(int k, const int* a, int l, int r)
{
    while(l <= r){
        int m = (l+r)/2;
        if(k == a[m]){
            return m;
        }
        else{
            if (k > a[m]){
                l = m + 1;
            }
            else{
                r = m - 1;
            }
        }
    }
    return -1;
}
```

Iterative v. Recursion Binary Search

```c
int bin_it(int k, const int* a, int l, int r)
{
    while(l <= r){
        int m = (l+r)/2;
        if(k == a[m]){
            return m;
        }
        else{
            if (k > a[m]){
                l = m + 1;
            }
            else{
                r = m - 1;
            }
        }
    }
    return -1;
}
```

```c
int bin_rec(int k, const int* a, int l, int r)
{
    if(l > r) return -1;
    int m = (l+r)/2;
    if(k == a[m]){
        return m;
    }
    else{
        if (k > a[m]){
            return bin_rec(k, a, m+1, r);
        }
        else{
            return bin_rec(k, a, l, m-1);
        }
    }
}
```

Interpolation Search

- When we look for a word in a dictionary, we don't start in the middle. We make an educated guess as to where to start based on the 1st letter of the word being searched for.

Interpolation Search

- When we look for a word in a dictionary, we don't start in the middle. We make an educated guess as to where to start based on the 1st letter of the word being searched for.
- This idea led to the interpolation search.

Interpolation Search

- When we look for a word in a dictionary, we don't start in the middle. We make an educated guess as to where to start based on the 1st letter of the word being searched for.
- This idea led to the interpolation search.
- In binary searching, we simply used the middle of an ordered list as a best guess as to where to begin the search.

Interpolation Search

- When we look for a word in a dictionary, we don't start in the middle. We make an educated guess as to where to start based on the 1st letter of the word being searched for.

- This idea led to the interpolation search.

- In binary searching, we simply used the middle of an ordered list as a best guess as to where to begin the search.

- Now we use an interpolation involving the key, the start of the list and the end.

$$i = (k - l[0])/(l[n-1] - l[0]) * n$$

Interpolation Search

- When we look for a word in a dictionary, we don't start in the middle. We make an educated guess as to where to start based on the 1st letter of the word being searched for.
- This idea led to the interpolation search.
- In binary searching, we simply used the middle of an ordered list as a best guess as to where to begin the search.
- Now we use an interpolation involving the key, the start of the list and the end.

$$i = (k - l[0])/(l[n-1] - l[0]) * n$$

- when searching for '15' :

$$0 \quad 4 \quad 5 \quad 9 \quad 10 \quad 12 \quad 15 \quad 20$$
$$\Uparrow$$

Interpolation Search

- When we look for a word in a dictionary, we don't start in the middle. We make an educated guess as to where to start based on the 1st letter of the word being searched for.
- This idea led to the interpolation search.
- In binary searching, we simply used the middle of an ordered list as a best guess as to where to begin the search.
- Now we use an interpolation involving the key, the start of the list and the end.

$$i = (k - l[0])/(l[n-1] - l[0]) * n$$

- when searching for '15' :

0 4 5 9 10 12 15 20

⇑

Interpolation Search

- When we look for a word in a dictionary, we don't start in the middle. We make an educated guess as to where to start based on the 1st letter of the word being searched for.

- This idea led to the interpolation search.

- In binary searching, we simply used the middle of an ordered list as a best guess as to where to begin the search.

- Now we use an interpolation involving the key, the start of the list and the end.

$$i = (k - l[0])/(l[n-1] - l[0]) * n$$

- when searching for '15' :

$$0 \quad 4 \quad 5 \quad 9 \quad 10 \quad 12 \quad 15 \quad 20$$

$$\Uparrow$$

```
int interp(int k, const int* a, int l, int r)
{

    int m;
    double md;

    while(l <= r){
        md = ((double)(k-a[l])/
              (double)(a[r]-a[l])*
              (double)(r-l)
             )
             +(double)(l);
        m  = 0.5 + md;
        if((m > r) || (m < l)){
            return -1;
        }
        if(k == a[m])
            return m;
        else{
            if (k > a[m]){
                l = m + 1;
            }
            else{
                r = m- 1;
            }
        }
    }
}
```

Algorithmic Complexity

```c
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4
5   #define CSEC (double)(CLOCKS_PER_SEC)
6   #define BIGLOOP 1000000000
7
8   int main(void)
9   {
10
11      clock_t c1 = clock();
12      for(int i=0; i<BIGLOOP; i++){
13          int j = i * 2;
14      }
15      clock_t c2 = clock();
16      printf("%f\n", (double)(c2-c1)/CSEC);
17      return 0;
18
19  }
```

- This code on an old Dell laptop took:

Algorithmic Complexity

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define CSEC (double)(CLOCKS_PER_SEC)
#define BIGLOOP 1000000000

int main(void)
{

    clock_t c1 = clock();
    for(int i=0; i<BIGLOOP; i++){
        int j = i * 2;
    }
    clock_t c2 = clock();
    printf("%f\n", (double)(c2-c1)/CSEC);
    return 0;

}
```

- This code on an old Dell laptop took:
  - 3.12 seconds using a non-optimzing compiler -O0

Algorithmic Complexity

```c
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include <time.h>
4
5    #define CSEC (double)(CLOCKS_PER_SEC)
6    #define BIGLOOP 1000000000
7
8    int main(void)
9    {
10
11       clock_t c1 = clock();
12       for(int i=0; i<BIGLOOP; i++){
13            int j = i * 2;
14       }
15       clock_t c2 = clock();
16       printf("%f\n", (double)(c2-c1)/CSEC);
17       return 0;
18
19   }
```

- This code on an old Dell laptop took:
    - 3.12 seconds using a non-optimzing compiler -O0
    - 0.00 seconds using an aggressive optimization -O3

Algorithmic Complexity

```c
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4
5   #define CSEC (double)(CLOCKS_PER_SEC)
6   #define BIGLOOP 1000000000
7
8   int main(void)
9   {
10
11      clock_t c1 = clock();
12      for(int i=0; i<BIGLOOP; i++){
13          int j = i * 2;
14      }
15      clock_t c2 = clock();
16      printf("%f\n", (double)(c2-c1)/CSEC);
17      return 0;
18
19  }
```

- This code on an old Dell laptop took:
    - 3.12 seconds using a non-optimzing compiler -O0
    - 0.00 seconds using an aggressive optimization -O3
- But "wall-clock" time is generally not the thing that excites Computer Scientists.

Algorithmic Complexity

```c
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4
5   #define CSEC (double)(CLOCKS_PER_SEC)
6   #define BIGLOOP 1000000000
7
8   int main(void)
9   {
10
11      clock_t c1 = clock();
12      for(int i=0; i<BIGLOOP; i++){
13          int j = i * 2;
14      }
15      clock_t c2 = clock();
16      printf("%f\n", (double)(c2-c1)/CSEC);
17      return 0;
18
19  }
```

- This code on an old Dell laptop took:
  - 3.12 seconds using a non-optimzing compiler -O0
  - 0.00 seconds using an aggressive optimization -O3
- But "wall-clock" time is generally not the thing that excites Computer Scientists.

Algorithmic Complexity

```c
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4
5   #define CSEC (double)(CLOCKS_PER_SEC)
6   #define BIGLOOP 1000000000
7
8   int main(void)
9   {
10
11      clock_t c1 = clock();
12      for(int i=0; i<BIGLOOP; i++){
13          int j = i * 2;
14      }
15      clock_t c2 = clock();
16      printf("%f\n", (double)(c2-c1)/CSEC);
17      return 0;
18
19  }
```

- Searching and sorting algorithms have a complexity associated with them, called big-O.

- This code on an old Dell laptop took:
  - 3.12 seconds using a non-optimzing compiler -O0
  - 0.00 seconds using an aggressive optimization -O3
- But "wall-clock" time is generally not the thing that excites Computer Scientists.

Algorithmic Complexity

```c
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4
5   #define CSEC (double)(CLOCKS_PER_SEC)
6   #define BIGLOOP 1000000000
7
8   int main(void)
9   {
10
11      clock_t c1 = clock();
12      for(int i=0; i<BIGLOOP; i++){
13          int j = i * 2;
14      }
15      clock_t c2 = clock();
16      printf("%f\n", (double)(c2-c1)/CSEC);
17      return 0;
18
19  }
```

- Searching and sorting algorithms have a complexity associated with them, called big-O.
- This complexity indicates how, for *n* numbers, performance deteriorates when *n* changes.

- This code on an old Dell laptop took:
  - 3.12 seconds using a non-optimzing compiler -O0
  - 0.00 seconds using an aggressive optimization -O3
- But "wall-clock" time is generally not the thing that excites Computer Scientists.

Algorithmic Complexity

```c
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4
5   #define CSEC (double)(CLOCKS_PER_SEC)
6   #define BIGLOOP 1000000000
7
8   int main(void)
9   {
10
11      clock_t c1 = clock();
12      for(int i=0; i<BIGLOOP; i++){
13          int j = i * 2;
14      }
15      clock_t c2 = clock();
16      printf("%f\n", (double)(c2-c1)/CSEC);
17      return 0;
18
19  }
```

- Searching and sorting algorithms have a complexity associated with them, called big-O.
- This complexity indicates how, for *n* numbers, performance deteriorates when *n* changes.
- Sequential Search : **O(n)**

- This code on an old Dell laptop took:
  - 3.12 seconds using a non-optimzing compiler -O0
  - 0.00 seconds using an aggressive optimization -O3
- But "wall-clock" time is generally not the thing that excites Computer Scientists.

Algorithmic Complexity

```c
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4
5   #define CSEC (double)(CLOCKS_PER_SEC)
6   #define BIGLOOP 1000000000
7
8   int main(void)
9   {
10
11      clock_t c1 = clock();
12      for(int i=0; i<BIGLOOP; i++){
13          int j = i * 2;
14      }
15      clock_t c2 = clock();
16      printf("%f\n", (double)(c2-c1)/CSEC);
17      return 0;
18
19  }
```

- Searching and sorting algorithms have a complexity associated with them, called big-O.
- This complexity indicates how, for *n* numbers, performance deteriorates when *n* changes.
- Sequential Search : **O(n)**
- Binary Search : **O(log n)**

- This code on an old Dell laptop took:
  - 3.12 seconds using a non-optimizing compiler -O0
  - 0.00 seconds using an aggressive optimization -O3
- But "wall-clock" time is generally not the thing that excites Computer Scientists.

Algorithmic Complexity

```c
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4
5   #define CSEC (double)(CLOCKS_PER_SEC)
6   #define BIGLOOP 1000000000
7
8   int main(void)
9   {
10
11      clock_t c1 = clock();
12      for(int i=0; i<BIGLOOP; i++){
13          int j = i * 2;
14      }
15      clock_t c2 = clock();
16      printf("%f\n", (double)(c2-c1)/CSEC);
17      return 0;
18
19   }
```

- This code on an old Dell laptop took:
    - 3.12 seconds using a non-optimzing compiler -O0
    - 0.00 seconds using an aggressive optimization -O3
- But "wall-clock" time is generally not the thing that excites Computer Scientists.

- Searching and sorting algorithms have a complexity associated with them, called big-O.
- This complexity indicates how, for n numbers, performance deteriorates when n changes.
- Sequential Search : **O(n)**
- Binary Search : **O(log n)**
- Interpolation Search : **O(log log n)**

Algorithmic Complexity

```c
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4
5   #define CSEC (double)(CLOCKS_PER_SEC)
6   #define BIGLOOP 1000000000
7
8   int main(void)
9   {
10
11      clock_t c1 = clock();
12      for(int i=0; i<BIGLOOP; i++){
13          int j = i * 2;
14      }
15      clock_t c2 = clock();
16      printf("%f\n", (double)(c2-c1)/CSEC);
17      return 0;
18
19  }
```

- This code on an old Dell laptop took:
  - 3.12 seconds using a non-optimzing compiler -O0
  - 0.00 seconds using an aggressive optimization -O3
- But "wall-clock" time is generally not the thing that excites Computer Scientists.

- Searching and sorting algorithms have a complexity associated with them, called big-O.
- This complexity indicates how, for n numbers, performance deteriorates when n changes.
- Sequential Search : **O(n)**
- Binary Search : **O(log n)**
- Interpolation Search : **O(log log n)**
- We'll discuss the dream of a **O(1)** search later in "Hashing".

Binary vs. Interpolation Timing

```c
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include <assert.h>
4    #include <time.h>
5
6    int bin_it(int k, const int *a, int l, int r);
7    int bin_rec(int k, const int *a, int l, int r);
8    int interp(int k, const int *a, int l, int r);
9    int* parse_args(int argc, char* argv[], int* n, int* srch);
10
11   int main(int argc, char* argv[])
12   {
13
14       int i, n, srch;
15       int* a;
16       int (*p[3])(int k, const int*a, int l, int r) =
17           {bin_it, bin_rec, interp};
18
19       a = parse_args(argc, argv, &n, &srch);
20
21       srand(time(NULL));
22       for(i=0; i<n; i++){
23           a[i] = 2*i;
24       }
25       for(i=0; i<5000000; i++){
26           assert((*p[srch])(a[rand()%n], a, 0, n-1) >= 0);
27       }
28
29       free(a);
30       return 0;
31
32   }
```

Binary vs. Interpolation Timing

```c
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <assert.h>
4   #include <time.h>
5
6   int bin_it(int k, const int *a, int l, int r);
7   int bin_rec(int k, const int *a, int l, int r);
8   int interp(int k, const int *a, int l, int r);
9   int* parse_args(int argc, char* argv[], int* n, int* srch);
10
11  int main(int argc, char* argv[])
12  {
13
14      int i, n, srch;
15      int* a;
16      int (*p[3])(int k, const int*a, int l, int r) =
17          {bin_it, bin_rec, interp};
18
19      a = parse_args(argc, argv, &n, &srch);
20
21      srand(time(NULL));
22      for(i=0; i<n; i++){
23          a[i] = 2*i;
24      }
25      for(i=0; i<5000000; i++){
26          assert((*p[srch])(a[rand()%n], a, 0, n-1) >= 0);
27      }
28
29      free(a);
30      return 0;
31
32  }
```

Execution :

```
Binary Search : Iterative
n =     100000 = 0.57
n =     800000 = 0.84
n =    6400000 = 2.20
n =   51200000 = 3.87
Binary Search : Recursive
n =     100000 = 1.23
n =     800000 = 1.79
n =    6400000 = 3.20
n =   51200000 = 4.85
Interpolation
n =     100000 = 0.20
n =     800000 = 0.28
n =    6400000 = 0.50
n =   51200000 = 0.70
```

Table of Contents

Linked Data Structures

- Linked data representations are useful when:

Linked Data Structures

- Linked data representations are useful when:
    - It is difficult to predict the size and the shape of the data structures in advance.

Linked Data Structures

- Linked data representations are useful when:
    - It is difficult to predict the size and the shape of the data structures in advance.
    - We need to efficiently insert and delete elements.

Linked Data Structures

- Linked data representations are useful when:
    - It is difficult to predict the size and the shape of the data structures in advance.
    - We need to efficiently insert and delete elements.
- To create linked data representations we use pointers to connect separate blocks of storage together. If a given block contains a pointer to a second block, we can follow this pointer there.

Linked Data Structures

- Linked data representations are useful when:
    - It is difficult to predict the size and the shape of the data structures in advance.
    - We need to efficiently insert and delete elements.
- To create linked data representations we use pointers to connect separate blocks of storage together. If a given block contains a pointer to a second block, we can follow this pointer there.
- By following pointers one after another, we can travel right along the structure.

Linked Data Structures

- Linked data representations are useful when:
    - It is difficult to predict the size and the shape of the data structures in advance.
    - We need to efficiently insert and delete elements.
- To create linked data representations we use pointers to connect separate blocks of storage together. If a given block contains a pointer to a second block, we can follow this pointer there.
- By following pointers one after another, we can travel right along the structure.

Linked Data Structures

- Linked data representations are useful when:
  - It is difficult to predict the size and the shape of the data structures in advance.
  - We need to efficiently insert and delete elements.
- To create linked data representations we use pointers to connect separate blocks of storage together. If a given block contains a pointer to a second block, we can follow this pointer there.
- By following pointers one after another, we can travel right along the structure.

```c
#include <stdio.h>
#include <stdlib.h>
#include "general.h"

typedef struct data{
    int i;
    struct data* next;
} Data;

Data* allocateData(int i);
void printList(Data* l);

int main(void)
{
    int i;
    Data* start , *current;
    start = current = NULL;
    printf("Enter the first number: ");
    if(scanf("%i", &i) == 1){
        start = current = allocateData(i);
    }
    else{
        on_error("Couldn't read an int");
    }

    printf("Enter more numbers: ");
    while(scanf("%i", &i) == 1){
        current->next = allocateData(i);
        current = current->next;
    }
    printList(start);
    // Should Free List
    return 0;
}
```

Linked Lists

```
Data* allocateData(int i)
{
    Data* p;
    p = (Data*) ncalloc(1, sizeof(Data));
    p->i = i;
    // Not really required
    p->next = NULL;
    return p;
}

void printList(Data* l)
{
    printf("\n");
    do{
        printf("Number : %i\n", l->i);
        l = l->next;
    }while(l != NULL);
    printf("END\n");
}
```

Linked Lists

```c
Data* allocateData(int i)
{
    Data* p;
    p = (Data*) ncalloc(1, sizeof(Data));
    p->i = i;
    // Not really required
    p->next = NULL;
    return p;
}

void printList(Data* l)
{
    printf("\n");
    do{
        printf("Number : %i\n", l->i);
        l = l->next;
    }while(l != NULL);
    printf("END\n");
}
```

Searching and Recursive printing:

```c
Data* inList(Data* n, int i)
{
    do{
        if(n->i==i){
            return n;
        }
        n = n->next;
    }while(n != NULL);
    return NULL;
}

void printList_r(Data* l)
{
    // Recursive Base-Case
    if(l == NULL) return;

    printf("Number: %i\n", l->i);
    printList_r(l->next);
}
```

Abstract Data Types

- But would we really code something like this **every** time we need flexible data storage ?

Abstract Data Types

- But would we really code something like this **every** time we need flexible data storage ?
- This would be horribly error-prone.

Abstract Data Types

- But would we really code something like this **every** time we need flexible data storage ?
- This would be horribly error-prone.
- Build something once, and test it well.

Abstract Data Types

- But would we really code something like this **every** time we need flexible data storage ?
- This would be horribly error-prone.
- Build something once, and test it well.
- One example of this is an **Abstract Data Type (ADT)**.

Abstract Data Types

- But would we really code something like this **every** time we need flexible data storage ?
- This would be horribly error-prone.
- Build something once, and test it well.
- One example of this is an **Abstract Data Type (ADT)**.
- Each ADT exposes its functionality via an *interface*.

Abstract Data Types

- But would we really code something like this **every** time we need flexible data storage ?
- This would be horribly error-prone.
- Build something once, and test it well.
- One example of this is an **Abstract Data Type (ADT)**.
- Each ADT exposes its functionality via an *interface*.
- The user only accesses the data via this interface.

Abstract Data Types

- But would we really code something like this **every** time we need flexible data storage ?
- This would be horribly error-prone.
- Build something once, and test it well.
- One example of this is an **Abstract Data Type (ADT)**.
- Each ADT exposes its functionality via an *interface*.
- The user only accesses the data via this interface.
- The user of the ADT doesn't need to understand how the data is being stored (e.g. array vs. linked lists etc.)

Table of Contents

Collections

- One of the simplest ADTs is the
  Collection.

Collections

- One of the simplest ADTs is the **Collection**.
- This is just a simple place to search for/add/delete data elements.

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).

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.

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.

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.

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.

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.

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.

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:

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

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

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

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

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
1    #include "../coll.h"
2    #include "specific.h"
3
4    coll* coll_init(void)
5    {
6        coll* c = (coll*) ncalloc(1, sizeof(coll));
7        c->size = 0;
8        return c;
9    }
10
11   int coll_size(coll* c)
12   {
13       if(c==NULL){
14           return 0;
15       }
16       return c->size;
17   }
18
19   bool coll_isin(coll* c, colltype d)
20   {
21       for(int i=0; i<coll_size(c); i++){
22           if(c->a[i] == d){
23               return true;
24           }
25       }
26       return false;
27   }
```

Collection ADT using a Fixed-size Array

Fixed/fixed.c:

```c
1    #include "../coll.h"
2    #include "specific.h"
3
4    coll* coll_init(void)
5    {
6        coll* c = (coll*) ncalloc(1, sizeof(coll));
7        c->size = 0;
8        return c;
9    }
10
11   int coll_size(coll* c)
12   {
13       if(c==NULL){
14           return 0;
15       }
16       return c->size;
17   }
18
19   bool coll_isin(coll* c, colltype  d)
20   {
21       for(int i=0; i<coll_size(c); i++){
22           if(c->a[i] == d){
23               return true;
24           }
25       }
26       return false;
27   }
```

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

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   };
```

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(1, sizeof(coll));
7      c->a = (colltype*) ncalloc(FIXEDSIZE, sizeof(colltype));
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         if(c->size >= c->capacity){
17             c->a = (colltype*) nremalloc(c->a,
18                     sizeof(colltype)*c->capacity*SCALEFACTOR);
19             c->capacity = c->capacity*SCALEFACTOR;
20         }
21         c->a[c->size] = d;
22         c->size = c->size + 1;
23     }
```

Linked/specific.h:

```c
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/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(1, sizeof(coll));
    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

```c
void coll_add(coll* c, colltype d)
{
    if(c){
        dataframe* f = ncalloc(1, sizeof(dataframe));
        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;
}
```

Collection ADT via a Linked List II

```c
void coll_add(coll* c, colltype d)
{
    if(c){
        dataframe* f = ncalloc(1, sizeof(dataframe));
        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;
}
```

```c
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.

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.

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:

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.

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.

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.

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.

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:

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)

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

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

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

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

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

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

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:

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:
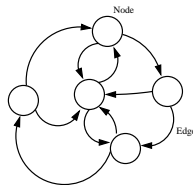
Stacks

The push-down stack:
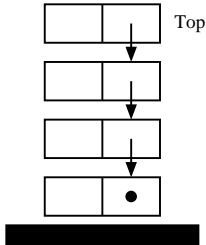
Stacks

The push-down stack:
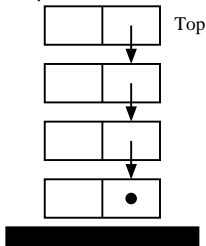
LIFO (Last in, First out):
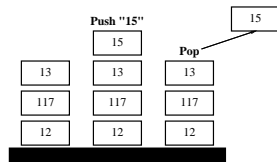
- Operations include push and pop.

Stacks

The push-down stack:
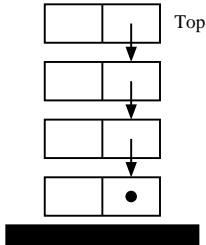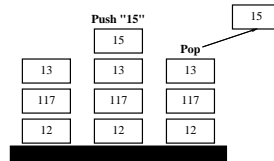
LIFO (Last in, First out):

- Operations include push and pop.
- In the C run-time system, function calls are implemented using stacks.

Stacks

The push-down stack:
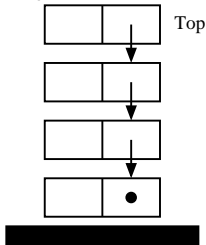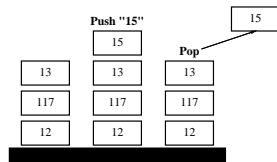
LIFO (Last in, First out):

- 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.

Stacks

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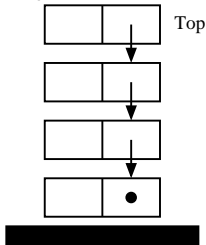
LIFO (Last in, First out):
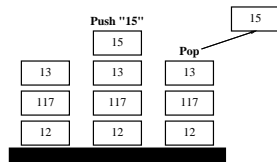
- Operations include push and pop.
- In the C run-time system, function calls are implemented using stacks.
- Most recursive algorithms can be re-written using stacks instead.
- But, once again, we are faced with the question : How best to implement such a data type ?

ADT:Stacks Arrays (Realloc) I

stack.h:

```
1   #pragma once
2
3   #include "../General/general.h"
4
5   typedef int stacktype;
6
7   typedef struct stack stack;
8
9   #include <stdio.h>
10  #include <stdlib.h>
11  #include <assert.h>
12  #include <string.h>
13
14  /* Create an empty stack */
15  stack* stack_init(void);
16  /* Add element to top */
17  void stack_push(stack* s, stacktype i);
18  /* Take element from top */
19  bool stack_pop(stack* s, stacktype* d);
20  /* Clears all space used */
21  bool stack_free(stack* s);
22
23  /* Optional? */
24
25  /* Copy top element into d (but don't pop it) */
26  bool stack_peek(stack*s, stacktype* d);
27  /* Make a string version - keep .dot in mind */
28  void stack_tostring(stack*, char* str);
```

ADT:Stacks Arrays (Realloc) I

stack.h:

```
1    #pragma once
2
3    #include "../General/general.h"
4
5    typedef int stacktype;
6
7    typedef struct stack stack;
8
9    #include <stdio.h>
10   #include <stdlib.h>
11   #include <assert.h>
12   #include <string.h>
13
14   /* Create an empty stack */
15   stack* stack_init(void);
16   /* Add element to top */
17   void stack_push(stack* s, stacktype i);
18   /* Take element from top */
19   bool stack_pop(stack* s, stacktype* d);
20   /* Clears all space used */
21   bool stack_free(stack* s);
22
23   /* Optional? */
24
25   /* Copy top element into d (but don't pop it) */
26   bool stack_peek(stack*s,  stacktype* d);
27   /* Make a string version - keep .dot in mind */
28   void stack_tostring(stack*, char* str);
```

Realloc/specific.h:

```
1    #pragma once
2
3    #define FORMATSTR "%i"
4    #define ELEMSIZE 20
5
6    #define STACKTYPE "Realloc"
7
8    #define FIXEDSIZE 16
9    #define SCALEFACTOR 2
10
11   struct stack {
12       /* Underlying array */
13       stacktype* a;
14       int size;
15       int capacity;
16   };
```

ADT:Stacks Arrays (Realloc) II

Realloc/realloc.c

```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(1, sizeof(stack));
9        /* Some implementations would allow you to pass
10           a hint about the initial size of the stack */
11       s->a = (stacktype*) ncalloc(FIXEDSIZE, sizeof(stacktype));
12       s->size = 0;
13       s->capacity= FIXEDSIZE;
14       return s;
15   }
16
17   void stack_push(stack* s, stacktype d)
18   {
19       if(s==NULL){
20           return;
21       }
22       if(s->size >= s->capacity){
23           s->a = (stacktype*) nrealloc(s->a,
24                   sizeof(stacktype)*s->capacity*SCALEFACTOR);
25           s->capacity = s->capacity*SCALEFACTOR;
26       }
27       s->a[s->size] = d;
28       s->size = s->size + 1;
29   }
```

ADT:Stacks Arrays (Realloc) II

Realloc/realloc.c

```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(1, sizeof(stack));
9        /* Some implementations would allow you to pass
10          a hint about the initial size of the stack */
11       s->a = (stacktype*) ncalloc(FIXEDSIZE, sizeof(stacktype));
12       s->size = 0;
13       s->capacity = FIXEDSIZE;
14       return s;
15   }
16
17   void stack_push(stack* s, stacktype d)
18   {
19       if(s==NULL){
20           return;
21       }
22       if(s->size >= s->capacity){
23           s->a = (stacktype*) nremalloc(s->a,
24                   sizeof(stacktype)*s->capacity*SCALEFACTOR);
25           s->capacity = s->capacity*SCALEFACTOR;
26       }
27       s->a[s->size] = d;
28       s->size = s->size + 1;
29   }
```

```c
1    bool stack_pop(stack* s, stacktype* d)
2    {
3        if((s == NULL) || (s->size < 1)){
4            return false;
5        }
6        s->size = s->size - 1;
7        *d = s->a[s->size];
8        return true;
9    }
10
11   bool stack_peek(stack* s, stacktype* d)
12   {
13       if((s==NULL) || (s->size <= 0)){
14           /* Stack is Empty */
15           return false;
16       }
17       *d = s->a[s->size-1];
18       return true;
19   }
```

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   }
```

ADT:Stacks Arrays (Realloc) III

Realloc/realloc.c

```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

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).

Linked/specific.h

```
1    #pragma once
2
3    #define FORMATSTR "%i"
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   };
```

ADT:Stacks Linked I

Linked/specific.h

```
1   #pragma once
2
3   #define FORMATSTR "%i"
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(1, sizeof(stack));
9       return s;
10  }
11
12  void stack_push(stack* s, stacktype d)
13  {
14      if(s){
15          dataframe* f = ncalloc(1, sizeof(dataframe));
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   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  }
```

Table of Contents

ADTs : Queues

FIFO (First in, First out):
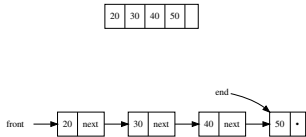
- Intuitively more "useful" than a stack.

ADTs : Queues

FIFO (First in, First out):
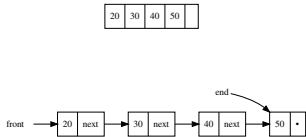
- Intuitively more "useful" than a stack.
- Think of implementing any kind of service
  (printer, web etc.)

ADTs : Queues

FIFO (First in, First out):

- Intuitively more "useful" than a stack.
- Think of implementing any kind of service (printer, web etc.)
- Operations include enqueue, dequeue and size.

ADTs : Queues

FIFO (First in, First out):

- Intuitively more "useful" than a stack.
- Think of implementing any kind of service (printer, web etc.)
- Operations include enqueue, dequeue and size.

ADTs : Queues

FIFO (First in, First out):

queue.h

- Intuitively more "useful" than a stack.
- Think of implementing any kind of service (printer, web etc.)
- Operations include enqueue, dequeue and size.

```c
1   #pragma once
2
3   #include "../General/general.h"
4
5   typedef int queuetype;
6
7   typedef struct queue queue;
8
9   #include <stdio.h>
10  #include <stdlib.h>
11  #include <string.h>
12  #include <assert.h>
13
14  /* Create an empty queue */
15  queue* queue_init(void);
16  /* Add element on end */
17  void queue_enqueue(queue* q, queuetype v);
18  /* Take element off front */
19  bool queue_dequeue(queue* q, queuetype* d);
20  /* Return size of queue */
21  int queue_size(queue* q);
22  /* Clears all space used */
23  bool queue_free(queue* q);
24
25  /* Helps with visualisation & testing */
26  void queue_tostring(queue* q, char* str);
```

ADTs : Queues (Fixed) I

specific.h

```
1   #pragma once
2
3   #define FORMATSTR "%d"
4   #define ELEMSIZE 20
5
6   #define QUEUETYPE "Fixed"
7
8   #define BOUNDED 5000
9
10  struct queue {
11      /* Underlying array */
12      queuetype a[BOUNDED];
13      int front;
14      int end;
15  };
16
17  #define DOTFILE 5000
```

specific.h

```
1   #pragma once
2
3   #define FORMATSTR "%d"
4   #define ELEMSIZE 20
5
6   #define QUEUETYPE "Fixed"
7
8   #define BOUNDED 5000
9
10  struct queue {
11      /* Underlying array */
12      queuetype a[BOUNDED];
13      int front;
14      int end;
15  };
16
17  #define DOTFILE 5000
```


Front (dequeue)

Ring Buffer

End (enqueue)

fixed.c

```
1    #include "../queue.h"
2    #include "specific.h"
3
4    void _inc(queuetype* p);
5
6    queue* queue_init(void)
7    {
8        queue* q = (queue*) ncalloc(1, sizeof(queue));
9        return q;
10   }
11
12
13   void queue_enqueue(queue* q, queuetype d)
14   {
15       if(q){
16           q->a[q->end] = d;
17           _inc(&q->end);
18           if(q->end == q->front){
19               on_error("Queue too large");
20           }
21       }
22   }
```

fixed.c

```c
1    #include "../queue.h"
2    #include "specific.h"
3
4    void _inc(queuetype* p);
5
6    queue* queue_init(void)
7    {
8        queue* q = (queue*) ncalloc(1, sizeof(queue));
9        return q;
10   }
11
12
13   void queue_enqueue(queue* q, queuetype d)
14   {
15       if(q){
16           q->a[q->end] = d;
17           _inc(&q->end);
18           if(q->end == q->front){
19               on_error("Queue too large");
20           }
21       }
22   }
```

```c
1    bool queue_dequeue(queue* q, queuetype* d)
2    {
3        if((q==NULL) || (q->front==q->end)){
4            return false;
5        }
6        *d = q->a[q->front];
7        _inc(&q->front);
8        return true;
9    }
10
11   void queue_tostring(queue* q, char* str)
12   {
13       char tmp[ELEMSIZE];
14       str[0] = '\0';
15       if((q==NULL) || (queue_size(q)==0)){
16           return;
17       }
18       for(int i=q->front; i != q->end;){
19           sprintf(tmp, FORMATSTR, q->a[i]);
20           strcat(str, tmp);
21           strcat(str, "|");
22           _inc(&i);
23       }
24       str[strlen(str)-1] = '\0';
25   }
```

```
1    int queue_size(queue* q)
2    {
3        if(q==NULL){
4            return 0;
5        }
6        if(q->end >= q->front){
7            return q->end-q->front;
8        }
9        return q->end + BOUNDED - q->front;
10   }
11
12   bool queue_free(queue* q)
13   {
14       free(q);
15       return true;
16   }
17
18   void _inc(queuetype* p)
19   {
20       *p = (*p + 1) % BOUNDED;
21   }
```

```
1   int queue_size(queue* q)
2   {
3       if(q==NULL){
4           return 0;
5       }
6       if(q->end >= q->front){
7           return q->end-q->front;
8       }
9       return q->end + BOUNDED - q->front;
10  }
11
12  bool queue_free(queue* q)
13  {
14      free(q);
15      return true;
16  }
17
18  void _inc(queuetype* p)
19  {
20      *p = (*p + 1) % BOUNDED;
21  }
```

- We need a thorough testing program

```
1    int queue_size(queue* q)
2    {
3        if(q==NULL){
4            return 0;
5        }
6        if(q->end >= q->front){
7            return q->end-q->front;
8        }
9        return q->end + BOUNDED - q->front;
10   }
11
12   bool queue_free(queue* q)
13   {
14       free(q);
15       return true;
16   }
17
18   void _inc(queuetype* p)
19   {
20       *p = (*p + 1) % BOUNDED;
21   }
```

- We need a thorough testing program
- We'll see queues again for traversing trees

```
1   int queue_size(queue* q)
2   {
3       if(q==NULL){
4           return 0;
5       }
6       if(q->end >= q->front){
7           return q->end-q->front;
8       }
9       return q->end + BOUNDED - q->front;
10  }
11
12  bool queue_free(queue* q)
13  {
14      free(q);
15      return true;
16  }
17
18  void _inc(queuetype* p)
19  {
20      *p = (*p + 1) % BOUNDED;
21  }
```

- We need a thorough testing program
- We'll see queues again for traversing trees
- Simulating a (slow) printer

specific.h

```c
1   #pragma once
2
3   #define FORMATSTR "%d"
4   #define ELEMSIZE 20
5
6   #define QUEUETYPE "Linked"
7
8   struct dataframe {
9       queuetype i;
10      struct dataframe* next;
11  };
12  typedef struct dataframe dataframe;
13
14  struct queue {
15      /* Underlying array */
16      dataframe* front;
17      dataframe* end;
18      int size;
19  };
```

ADTs : Queues (Linked) I

specific.h

```
1   #pragma once
2
3   #define FORMATSTR "%d"
4   #define ELEMSIZE 20
5
6   #define QUEUETYPE "Linked"
7
8   struct dataframe {
9       queuetype i;
10      struct dataframe* next;
11  };
12  typedef struct dataframe dataframe;
13
14  struct queue {
15      /* Underlying array */
16      dataframe* front;
17      dataframe* end;
18      int size;
19  };
```

linked.c

```
1   #include "../queue.h"
2   #include "specific.h"
3
4   queue* queue_init(void)
5   {
6       queue* q = (queue*) ncalloc(1, sizeof(queue));
7       return q;
8   }
9
10  void queue_enqueue(queue* q, queuetype d)
11  {
12      dataframe* f;
13      if(q==NULL){
14          return;
15      }
16
17      /* Copy the data */
18      f = ncalloc(1, sizeof(dataframe));
19      f->i = d;
20
21      /* 1st one */
22      if(q->front == NULL){
23          q->front = f;
24          q->end = f;
25          q->size = q->size + 1;
26          return;
27      }
28      /* Not 1st */
29      q->end->next = f;
30      q->end = f;
31      q->size = q->size + 1;
32  }
```

ADTs : Queues (Linked) II

```
1    bool queue_dequeue( queue* q, queuetype* d)
2    {
3        dataframe* f;
4        if((q==NULL) || (q->front==NULL) || (q->end==NULL)){
5            return false;
6        }
7        f = q->front->next;
8        *d = q->front->i;
9        free(q->front);
10       q->front = f;
11       q->size = q->size - 1;
12       return true;
13   }
14
15   bool queue_free( queue* q)
16   {
17       if(q){
18           dataframe* tmp;
19           dataframe* p = q->front;
20           while(p!=NULL){
21               tmp = p->next;
22               free(p);
23               p = tmp;
24           }
25           free(q);
26       }
27       return true;
28   }
```

ADTs : Queues (Linked) II

```c
bool queue_dequeue(queue* q, queuetype* d)
{
    dataframe* f;
    if((q==NULL) || (q->front==NULL) || (q->end==NULL)){
        return false;
    }
    f = q->front->next;
    *d = q->front->i;
    free(q->front);
    q->front = f;
    q->size = q->size - 1;
    return true;
}

bool queue_free(queue* q)
{
    if(q){
        dataframe* tmp;
        dataframe* p = q->front;
        while(p!=NULL){
            tmp = p->next;
            free(p);
            p = tmp;
        }
        free(q);
    }
    return true;
}
```

```c
void queue_tostring(queue* q, char* str)
{
    dataframe *p;
    char tmp[ELEMSIZE];
    str[0] = '\0';
    if((q==NULL) || (q->front == NULL)){
        return;
    }
    p = q->front;
    while(p){
        sprintf(tmp, FORMATSTR, p->i);
        strcat(str, tmp);
        strcat(str, "|");
        p = p->next;
    }
    str[strlen(str)-1] = '\0';
}

int queue_size(queue* q)
{
    if((q==NULL) || (q->front==NULL)){

        return 0;
    }
    return q->size;
}
```

Detour : Graphviz

- There exists a nice package, called Graphviz:

  `sudo apt install graphviz`

Detour : Graphviz

- There exists a nice package, called Graphviz:

  sudo apt install graphviz

- This allows the visualisation of graphs/dynamic structures using the simple .dot language:

  ```
  digraph {
     a -> b; b -> c; c -> a;
  }
  ```

Detour : Graphviz

- There exists a nice package, called Graphviz:

  sudo apt install graphviz

- This allows the visualisation of graphs/dynamic structures using the simple .dot language:

  ```
  digraph {
     a -> b; b -> c; c -> a;
  }
  ```

- To create a .pdf:

  dot -Tpdf -o graphviz.pdf examp1.dot

Detour : Graphviz

- There exists a nice package, called Graphviz:

  sudo apt install graphviz

- This allows the visualisation of graphs/dynamic structures using the simple .dot language:

  ```
  digraph {
     a -> b; b -> c; c -> a;
  }
  ```

- To create a .pdf:

  dot -Tpdf -o graphviz.pdf examp1.dot

Detour : Graphviz

- There exists a nice package, called Graphviz:

  sudo apt install graphviz

- This allows the visualisation of graphs/dynamic structures using the simple .dot language:

  digraph {
     a -> b; b -> c; c -> a;
  }

- To create a .pdf:

  dot -Tpdf -o graphviz.pdf examp1.dot

Table of Contents

ADTs : Binary Trees

- Binary trees are used extensively in computer science

ADTs : Binary Trees

- Binary trees are used extensively in computer science
- Game Trees

ADTs : Binary Trees

- Binary trees are used extensively in computer science
- Game Trees
- Searching

ADTs : Binary Trees

- Binary trees are used extensively in computer science
- Game Trees
- Searching
- Sorting

ADTs : Binary Trees

- Binary trees are used extensively in computer science
- Game Trees
- Searching
- Sorting

ADTs : Binary Trees

- Binary trees are used extensively in computer science
- Game Trees
- Searching
- Sorting
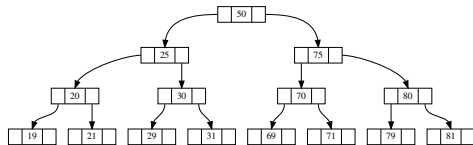
ADTs : Binary Trees

- Binary trees are used extensively in computer science
- Game Trees
- Searching
- Sorting

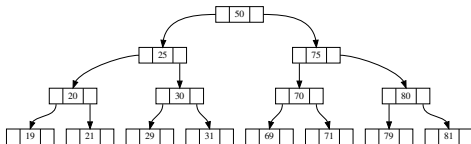- Trees drawn upside-down !

ADTs : Binary Trees

- Binary trees are used extensively in computer science
- Game Trees
- Searching
- Sorting

- Trees drawn upside-down !
- Ancestor relationships: '50' is the parent of '25' and '75'.
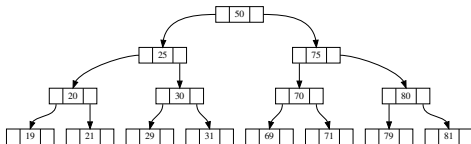
ADTs : Binary Trees

- Binary trees are used extensively in computer science
- Game Trees
- Searching
- Sorting

- Trees drawn upside-down !
- Ancestor relationships: '50' is the parent of '25' and '75'.
- Can refer to left and right children
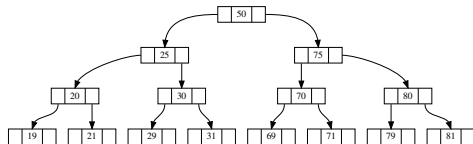
ADTs : Binary Trees

- Binary trees are used extensively in computer science
- Game Trees
- Searching
- Sorting
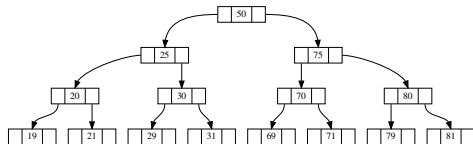
- Trees drawn upside-down !
- Ancestor relationships: '50' is the parent of '25' and '75'.
- Can refer to left and right children
- In a tree, there is only one path from the root to any child

ADTs : Binary Trees

- Binary trees are used extensively in computer science
- Game Trees
- Searching
- Sorting
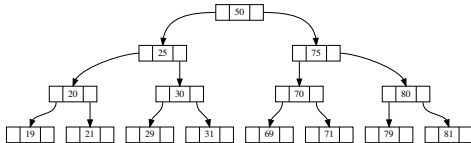
- Trees drawn upside-down !
- Ancestor relationships: '50' is the parent of '25' and '75'.
- Can refer to left and right children
- In a tree, there is only one path from the root to any child
- A node with no children is a leaf

ADTs : Binary Trees

- Binary trees are used extensively in computer science
- Game Trees
- Searching
- Sorting

- Trees drawn upside-down !
- Ancestor relationships: '50' is the parent of '25' and '75'.
- Can refer to left and right children
- In a tree, there is only one path from the root to any child
- A node with no children is a leaf
- Most trees need to be created dynamically

ADTs : Binary Trees

- Binary trees are used extensively in computer science
- Game Trees
- Searching
- Sorting
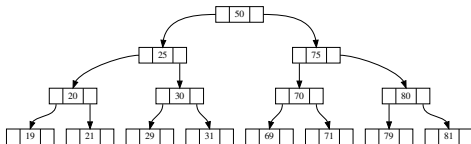
- Trees drawn upside-down !
- Ancestor relationships: '50' is the parent of '25' and '75'.
- Can refer to left and right children
- In a tree, there is only one path from the root to any child
- A node with no children is a leaf
- Most trees need to be created dynamically
- Empty subtrees are set to NULL

Binary Search Trees

In a binary search tree the left-hand tree of a parent contains all keys less than the parent node, and the right-hand side all the keys greater than the parent node.
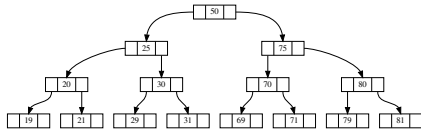
Binary Search Trees

In a binary search tree the left-hand tree of a parent contains all keys less than the parent node, and the right-hand side all the keys greater than the parent node.

bst.h

```
1   #include "../General/general.h"
2   #include "../Queue/queue.h"
3
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include <assert.h>
7
8   bst* bst_init(void);
9
10  /* Insert 1 item into the tree */
11  bool bst_insert(bst* b, treetype d);
12
13  /* Return number of nodes in tree */
14  int bst_size(bst* b);
15
16  /* Whether the data d is stored in the tree */
17  bool bst_isin(bst* b, treetype d);
18
19  /* Bulk insert n items from an array a into an initialised tree */
20  bool bst_insertarray(bst* b, treetype* a, int n);
21
22  /* Clear all memory associated with tree, & set pointer to NULL */
23  bool bst_free(bst* b);
24
25  /* Optional ? */
26
27  char* bst_preorder(bst* b);
28  void bst_printlevel(bst* b);
29  /* Create string with tree as ((head)(left)(right)) */
30  char* bst_printlisp(bst* b);
31  /* Use Graphviz via a .dot file */
32  void bst_todot(bst* b, char* dotname);
```

Binary Search Trees : Linked I

specific.h

```c
1    #include <string.h>
2
3    typedef int treetype;
4    #define FORMATSTR "%i"
5    #define ELEMSIZE 20
6    #define BSTTYPE "Linked"
7
8    struct dataframe {
9        treetype d;
10       struct dataframe* left;
11       struct dataframe* right;
12   };
13   typedef struct dataframe dataframe;
14
15   struct bst {
16       dataframe* top;
17       /* Data element size, in bytes */
18   };
19   typedef struct bst bst;
```

Binary Search Trees : Linked I

specific.h

```c
 1  #include <string.h>
 2
 3  typedef int treetype;
 4  #define FORMATSTR "%i"
 5  #define ELEMSIZE 20
 6  #define BSTTYPE "Linked"
 7
 8  struct dataframe {
 9      treetype d;
10      struct dataframe* left;
11      struct dataframe* right;
12  };
13  typedef struct dataframe dataframe;
14
15  struct bst {
16      dataframe* top;
17      /* Data element size, in bytes */
18  };
19  typedef struct bst bst;
```

```c
/* Based on geekforgeeks.org */
dataframe* _insert(dataframe* t, treetype d)
{
    dataframe* f;
    /* If the tree is empty, return a new frame */
    if (t == NULL){
        f = ncalloc(sizeof(dataframe), 1);
        f->d = d;
        return f;
    }
    /* Otherwise, recurs down the tree */
    if (d < t->d){
        t->left = _insert(t->left, d);
    }
    else if(d > t->d){
        t->right = _insert(t->right, d);
    }
    /* return the (unchanged) dataframe pointer */
    return t;
}
```

```
bool _isin(dataframe* t, treetype d)
{
    if(t==NULL){
        return false;
    }
    if(t->d == d){
        return true;
    }
    if(d < t->d){
        return _isin(t->left, d);
    }
    else{
        return _isin(t->right, d);
    }
    return false;
}
```

```c
bool _isin(dataframe* t, treetype d)
{
    if(t==NULL){
        return false;
    }
    if(t->d == d){
        return true;
    }
    if(d < t->d){
        return _isin(t->left, d);
    }
    else{
        return _isin(t->right, d);
    }
    return false;
}
```

```c
char* _printlisp(dataframe* t)
{
    char tmp[ELEMSIZE];
    char *s1, *s2, *p;

    if(t==NULL){
        /* \0 string */
        p = ncalloc(1,1);
        return p;
    }
    sprintf(tmp, FORMATSTR, t->d);
    s1 = _printlisp(t->left);
    s2 = _printlisp(t->right);
    p = ncalloc(strlen(s1)+strlen(s2)+strlen(tmp)+
        strlen("()() "), 1);
    sprintf(p, "%s(%s)(%s)", tmp, s1, s2);
    free(s1);
    free(s2);
    return p;
}
```

Binary Trees using Arrays ?

- Don't rush to assume a linked data structure must be used to implement trees.

Binary Trees using Arrays ?

- Don't rush to assume a linked data structure must be used to implement trees.
- You could use 1 cell of an array for the first node, the next two cells for its children, the next 4 cells for their children and so on.

Binary Trees using Arrays ?

- Don't rush to assume a linked data structure must be used to implement trees.
- You could use 1 cell of an array for the first node, the next two cells for its children, the next 4 cells for their children and so on.
- You need to mark which cells are in use & which aren't ...

Binary Trees using Arrays ?

- Don't rush to assume a linked data structure must be used to implement trees.
- You could use 1 cell of an array for the first node, the next two cells for its children, the next 4 cells for their children and so on.
- You need to mark which cells are in use & which aren't ...

Binary Trees using Arrays ?

- Don't rush to assume a linked data structure must be used to implement trees.
- You could use 1 cell of an array for the first node, the next two cells for its children, the next 4 cells for their children and so on.
- You need to mark which cells are in use & which aren't ...

Counting from cell 1, for a tree with n nodes:

To find	Use	Iff
The root	$A[1]$	A is nonempty
The left child of $A[i]$	$A[2i]$	$2i \leq n$
The parent of $A[i]$	$A[i/2]$	$i > 1$
Is $A[i]$ a leaf ?	True	$2i > n$

Binary Search Trees : Realloc

specific.h

```c
1   #include <stdbool.h>
2
3   typedef int treetype;
4   #define FORMATSTR "%i"
5   #define ELEMSIZE 20
6   #define BSTTYPE "Realloc"
7
8   // Probably (2^n) -1
9   #define INITSIZE 31
10  #define SCALEFACTOR 2
11
12  struct dataframe {
13      treetype d;
14      bool isvalid;
15  };
16  typedef struct dataframe dataframe;
17
18  struct bst {
19      dataframe* a;
20      int capacity;
21  };
22  typedef struct bst bst;
```

Binary Search Trees : Realloc

specific.h

```
1   #include <stdbool.h>
2
3   typedef int treetype;
4   #define FORMATSTR "%i"
5   #define ELEMSIZE 20
6   #define BSTTYPE "Realloc"
7
8   // Probably (2^n) -1
9   #define INITSIZE 31
10  #define SCALEFACTOR 2
11
12  struct dataframe {
13      treetype d;
14      bool isvalid;
15  };
16  typedef struct dataframe dataframe;
17
18  struct bst {
19      dataframe* a;
20      int capacity;
21  };
22  typedef struct bst bst;
```

Using a queue for Level-Order traversal:

```
void bst_printlevel(bst* b)
{
    treetype n;
    if((b==NULL) || (! _isvalid(b, 0))){
        return;
    }
    /* Make a queue of cell indices */
    queue* q = queue_init();
    queue_enqueue(q, 0);
    while(queue_dequeue(q, &n) && _isvalid(b, (int)n)){
        printf(FORMATSTR, b->a[n].d);
        putchar(' ');
        queue_enqueue(q, _leftchild((int)n));
        queue_enqueue(q, _rightchild((int)n));
    }
}
```

Binary Search Trees : Complexity

- So, in a nicely balanced tree, insertion, deletion and search are all $O(\log n)$.

Binary Search Trees : Complexity

- So, in a nicely balanced tree, insertion, deletion and search are all $O(\log n)$.
- But: if the root of the tree is not well chosen, or the keys to be inserted are ordered, the tree can become a linked list !

Binary Search Trees : Complexity

- So, in a nicely balanced tree, insertion, deletion and search are all $O(\log n)$.
- But: if the root of the tree is not well chosen, or the keys to be inserted are ordered, the tree can become a linked list !
- In this case, complexity becomes $O(n)$.

Binary Search Trees : Complexity

- So, in a nicely balanced tree, insertion, deletion and search are all $O(\log n)$.
- But: if the root of the tree is not well chosen, or the keys to be inserted are ordered, the tree can become a linked list !
- In this case, complexity becomes $O(n)$.
- The tree search performs best when well balanced trees are formed.

Binary Search Trees : Complexity

- So, in a nicely balanced tree, insertion, deletion and search are all $O(\log n)$.
- But: if the root of the tree is not well chosen, or the keys to be inserted are ordered, the tree can become a linked list !
- In this case, complexity becomes $O(n)$.
- The tree search performs best when well balanced trees are formed.
- Large body of literature about creating & re-balancing trees - Red-Black trees, Tries, 2-3 trees, AVL trees etc.

Binary Trees : Huffman Compression I

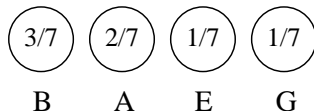- Often we wish to compress data, to reduce storage requirements, or to speed transmission.

Binary Trees : Huffman Compression I

- Often we wish to compress data, to reduce storage requirements, or to speed transmission.
- Text is particularly suited to compression since using one byte per character is wasteful - some letters occur much more frequently.

Binary Trees : Huffman Compression I

- Often we wish to compress data, to reduce storage requirements, or to speed transmission.
- Text is particularly suited to compression since using one byte per character is wasteful - some letters occur much more frequently.
- Need to give frequently occurring letters short codes, typically a few bits. Less common letters can have long bit patterns.

- Often we wish to compress data, to reduce storage requirements, or to speed transmission.
- Text is particularly suited to compression since using one byte per character is wasteful - some letters occur much more frequently.
- Need to give frequently occurring letters short codes, typically a few bits. Less common letters can have long bit patterns.

- To encode the string "BABBAGE":

Binary Trees : Huffman Compression I

- Often we wish to compress data, to reduce storage requirements, or to speed transmission.
- Text is particularly suited to compression since using one byte per character is wasteful - some letters occur much more frequently.
- Need to give frequently occurring letters short codes, typically a few bits. Less common letters can have long bit patterns.

- To encode the string "BABBAGE":

Binary Trees : Huffman Compression I

- Often we wish to compress data, to reduce storage requirements, or to speed transmission.

- Text is particularly suited to compression since using one byte per character is wasteful - some letters occur much more frequently.

- Need to give frequently occurring letters short codes, typically a few bits. Less common letters can have long bit patterns.

- To encode the string "BABBAGE":

$$\left(3/7\right) \left(2/7\right) \left(1/7\right) \left(1/7\right)$$

| B | A | E | G |

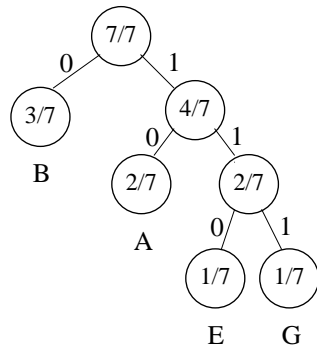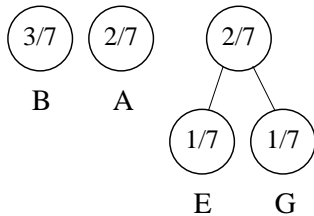- Keep a list of characters, ordered by their frequency

Binary Trees : Huffman Compression II

- Use the two least frequent to form a
  sub-tree, and re-order (sort) the nodes :

Binary Trees : Huffman Compression II

- Use the two least frequent to form a
  sub-tree, and re-order (sort) the nodes :

Binary Trees : Huffman Compression II

- Use the two least frequent to form a
  sub-tree, and re-order (sort) the nodes :
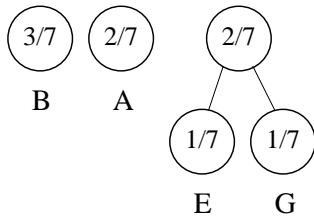
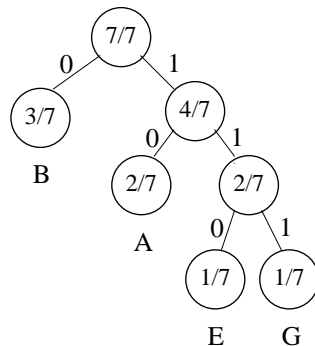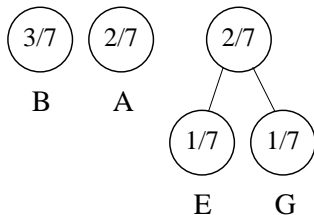# Binary Trees : Huffman Compression II

- Use the two least frequent to form a
  sub-tree, and re-order (sort) the nodes :

- Use the two least frequent to form a sub-tree, and re-order (sort) the nodes :

- A = 10, B = 0, E = 110, G = 111

Binary Trees : Huffman Compression II

- Use the two least frequent to form a sub-tree, and re-order (sort) the nodes :

- A = 10, B = 0, E = 110, G = 111
- String stored using 13 bits.

Table of Contents

ADTs : Hashing

- To keep records of employees we might index (search) them by using their National Insurance number:
  xx-##-##-##-x

ADTs : Hashing

- To keep records of employees we might index (search) them by using their National Insurance number:
  xx-##-##-##-x
- There are 17.6 billion combinations (around 2^{34}).

ADTs : Hashing

- To keep records of employees we might index (search) them by using their National Insurance number:
  xx-##-##-##-x
- There are 17.6 billion combinations (around 2^{34}).
- Could use an array of 17.6 billion entries, which would make searching for a particular entry trivial !

ADTs : Hashing

- To keep records of employees we might index (search) them by using their National Insurance number:
  xx-##-##-##-x
- There are 17.6 billion combinations (around 2^{34}).
- Could use an array of 17.6 billion entries, which would make searching for a particular entry trivial !
- Especially wasteful since only our (5000) employees need to be stored.

ADTs : Hashing

- To keep records of employees we might index (search) them by using their National Insurance number: xx-##-##-##-x
- There are 17.6 billion combinations (around 2^{34}).
- Could use an array of 17.6 billion entries, which would make searching for a particular entry trivial !
- Especially wasteful since only our (5000) employees need to be stored.

ADTs : Hashing

- To keep records of employees we might index (search) them by using their National Insurance number: xx-##-##-##-x
- There are 17.6 billion combinations (around 2^{34}).
- Could use an array of 17.6 billion entries, which would make searching for a particular entry trivial !
- Especially wasteful since only our (5000) employees need to be stored.

- Here we examine a method that, using an array of 6000 elements, would require 2.1 comparisons on average.

ADTs : Hashing

- To keep records of employees we might index (search) them by using their National Insurance number: xx-##-##-##-x
- There are 17.6 billion combinations (around 2^{34}).
- Could use an array of 17.6 billion entries, which would make searching for a particular entry trivial !
- Especially wasteful since only our (5000) employees need to be stored.

- Here we examine a method that, using an array of 6000 elements, would require 2.1 comparisons on average.
- A hash function is a mapping, $h(K)$, that maps from key K, onto the index of an entry.

ADTs : Hashing

- To keep records of employees we might index (search) them by using their National Insurance number: xx-##-##-##-x
- There are 17.6 billion combinations (around 2^{34}).
- Could use an array of 17.6 billion entries, which would make searching for a particular entry trivial !
- Especially wasteful since only our (5000) employees need to be stored.

- Here we examine a method that, using an array of 6000 elements, would require 2.1 comparisons on average.
- A hash function is a mapping, h(K), that maps from key K, onto the index of an entry.
- A black-box into which we insert a key (e.g. NI number) and out pops an array index.

ADTs : Hashing

- To keep records of employees we might index (search) them by using their National Insurance number: xx-##-##-##-x
- There are 17.6 billion combinations (around 2^{34}).
- Could use an array of 17.6 billion entries, which would make searching for a particular entry trivial !
- Especially wasteful since only our (5000) employees need to be stored.

- Here we examine a method that, using an array of 6000 elements, would require 2.1 comparisons on average.
- A hash function is a mapping, $h(K)$, that maps from key K, onto the index of an entry.
- A black-box into which we insert a key (e.g. NI number) and out pops an array index.
- As an example lets use an array of size 11 to store some airport codes, e.g. PHL, DCA, FRA.

ADTs : Aiport Codes

- In a three letter string $X_2 X_1 X_0$
  the letter 'A' has the value 0,
  'B' has the value 1 etc.

ADTs : Aiport Codes

- In a three letter string $X_2 X_1 X_0$
  the letter 'A' has the value 0,
  'B' has the value 1 etc.
- One hash function is:

  $h(K) = (X_2 * 26^2 + X_1 * 26 + X_0)\%11$

ADTs : Aiport Codes

- In a three letter string $X_2 X_1 X_0$
  the letter 'A' has the value 0,
  'B' has the value 1 etc.
- One hash function is:

  $h(K) = (X_2 * 26^2 + X_1 * 26 + X_0)\%11$

- Applying this to "DCA":
  $h("DCA") =$
  $(3 * 26^2 + 2 * 26 + 0)\%11$
  $h("DCA") = (2080)\%11$
  $h("DCA") = 1$

ADTs : Aiport Codes

- In a three letter string $X_2 X_1 X_0$
  the letter 'A' has the value 0,
  'B' has the value 1 etc.
- One hash function is:

  $h(K) = (X_2*26^2 + X_1*26 + X_0)\%11$

- Applying this to "DCA":
  $h("DCA") =$
  $(3 * 26^2 + 2 * 26 + 0)\%11$
  $h("DCA") = (2080)\%11$
  $h("DCA") = 1$

ADTs : Aiport Codes

- In a three letter string $X_2X_1X_0$ the letter 'A' has the value 0, 'B' has the value 1 etc.

- One hash function is:

  $h(K) = (X_2*26^2+X_1*26+X_0)\%11$

- Applying this to "DCA":
  $h("DCA") =$
  $(3 * 26^2 + 2 * 26 + 0)\%11$
  $h("DCA") = (2080)\%11$
  $h("DCA") = 1$

- Inserting "PHL", "ORY" and "GCM":

	0
	1
	2
	3
PHL	4
	5
GCM	6
	7
ORY	8
	9
	10

ADTs : Aiport Codes

- In a three letter string $X_2 X_1 X_0$ the letter 'A' has the value 0, 'B' has the value 1 etc.

- One hash function is:

$$h(K) = (X_2*26^2 + X_1*26 + X_0)\%11$$

- Applying this to "DCA":
  $h("DCA") =$
  $(3*26^2 + 2*26 + 0)\%11$
  $h("DCA") = (2080)\%11$
  $h("DCA") = 1$

- Inserting "PHL", "ORY" and "GCM":

- However, inserting "HKG" causes a collision.

ADTs : Collisions

- An ideal hashing function maps keys into
  the array in a *uniform* and *random*
  manner.

ADTs : Collisions

- An ideal hashing function maps keys into the array in a *uniform* and *random* manner.
- Collisions occur when a hash function maps two different keys onto the same address.

ADTs : Collisions

- An ideal hashing function maps keys into the array in a *uniform* and *random* manner.
- Collisions occur when a hash function maps two different keys onto the same address.
- It's very difficult to choose 'good' hashing functions.

ADTs : Collisions

- An ideal hashing function maps keys into the array in a *uniform* and *random* manner.
- Collisions occur when a hash function maps two different keys onto the same address.
- It's very difficult to choose 'good' hashing functions.
- Collisions are common - the **von Mises** paradox. When 23 keys are randomly mapped onto 365 addresses there is a 50% chance of a collision.

ADTs : Collisions

- An ideal hashing function maps keys into the array in a *uniform* and *random* manner.
- Collisions occur when a hash function maps two different keys onto the same address.
- It's very difficult to choose 'good' hashing functions.
- Collisions are common - the **von Mises** paradox. When 23 keys are randomly mapped onto 365 addresses there is a 50% chance of a collision.

ADTs : Collisions

- An ideal hashing function maps keys into the array in a *uniform* and *random* manner.
- Collisions occur when a hash function maps two different keys onto the same address.
- It's very difficult to choose 'good' hashing functions.
- Collisions are common - the **von Mises** paradox. When 23 keys are randomly mapped onto 365 addresses there is a 50% chance of a collision.

- The policy of finding another free location if a collision occurs is called open-addressing.
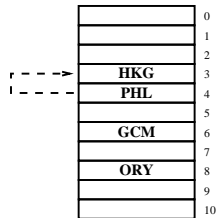
ADTs : Collisions

- An ideal hashing function maps keys into the array in a *uniform* and *random* manner.
- Collisions occur when a hash function maps two different keys onto the same address.
- It's very difficult to choose 'good' hashing functions.
- Collisions are common - the **von Mises** paradox. When 23 keys are randomly mapped onto 365 addresses there is a 50% chance of a collision.

- The policy of finding another free location if a collision occurs is called open-addressing.
- If a collision occurs then keep stepping backwards (with wrap-around) until a free location is encountered.

ADTs : Collisions

- An ideal hashing function maps keys into the array in a *uniform* and *random* manner.
- Collisions occur when a hash function maps two different keys onto the same address.
- It's very difficult to choose 'good' hashing functions.
- Collisions are common - the **von Mises** paradox. When 23 keys are randomly mapped onto 365 addresses there is a 50% chance of a collision.

- The policy of finding another free location if a collision occurs is called open-addressing.
- If a collision occurs then keep stepping backwards (with wrap-around) until a free location is encountered.

ADTs : Collisions

- An ideal hashing function maps keys into the array in a *uniform* and *random* manner.
- Collisions occur when a hash function maps two different keys onto the same address.
- It's very difficult to choose 'good' hashing functions.
- Collisions are common - the **von Mises** paradox. When 23 keys are randomly mapped onto 365 addresses there is a 50% chance of a collision.

- The policy of finding another free location if a collision occurs is called open-addressing.
- If a collision occurs then keep stepping backwards (with wrap-around) until a free location is encountered.

ADTs : Double Hashing

- This simple method of open-addressing is linear-probing.

ADTs : Double Hashing

- This simple method of open-addressing is linear-probing.
- The step taken each time (probe decrement) need not be 1.

ADTs : Double Hashing

- This simple method of open-addressing is linear-probing.
- The step taken each time (probe decrement) need not be 1.
- Open-addressing through use of linear-probing is a very simple technique, double-hashing is generally much more successful.

ADTs : Double Hashing

- This simple method of open-addressing is linear-probing.
- The step taken each time (probe decrement) need not be 1.
- Open-addressing through use of linear-probing is a very simple technique, double-hashing is generally much more successful.
- A second function p(K) decides the size of the probe decrement.

ADTs : Double Hashing

- This simple method of open-addressing is linear-probing.
- The step taken each time (probe decrement) need not be 1.
- Open-addressing through use of linear-probing is a very simple technique, double-hashing is generally much more successful.
- A second function $p(K)$ decides the size of the probe decrement.

ADTs : Double Hashing

- This simple method of open-addressing is linear-probing.
- The step taken each time (probe decrement) need not be 1.
- Open-addressing through use of linear-probing is a very simple technique, double-hashing is generally much more successful.
- A second function p(K) decides the size of the probe decrement.

- The function is chosen so that two keys which collide at the same address will have different probe decrements, e.g. :

$$p(K) = MAX(1, ((X_2 * 26^2 + X_1 * 26 + X_0)/11)\%11)$$

ADTs : Double Hashing

- This simple method of open-addressing is linear-probing.
- The step taken each time (probe decrement) need not be 1.
- Open-addressing through use of linear-probing is a very simple technique, double-hashing is generally much more successful.
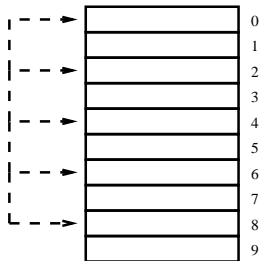- A second function p(K) decides the size of the probe decrement.

- The function is chosen so that two keys which collide at the same address will have different probe decrements, e.g. :

$$p(K) = MAX(1, ((X_2 * 26^2 + X_1 * 26 + X_0)/11)\%11)$$

- Although "PHL" and "HKG" share the same primary hash value of $h(K) = 4$, they have different probe decrements:
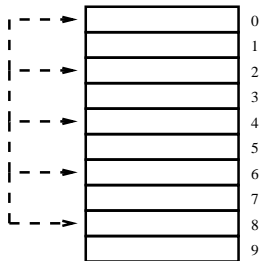$p("PHL") = 4$
$p("HKG") = 3$

ADTs : Primes and Chaining

- If the size of our array, M, was even and the probe decrement was chosen to be 2, then only half of the locations could be probed.

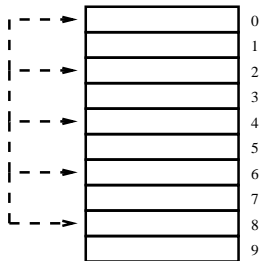## ADTs : Primes and Chaining

- If the size of our array, M, was even and
  the probe decrement was chosen to be 2,
  then only half of the locations could be
  probed.

- Often we choose our table size to be a
  prime number and our probe decrement
  to be a number in the range $1 \ldots M - 1$.
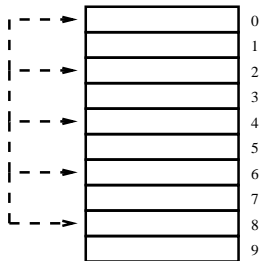
ADTs : Primes and Chaining

- If the size of our array, M, was even and the probe decrement was chosen to be 2, then only half of the locations could be probed.

- Often we choose our table size to be a prime number and our probe decrement to be a number in the range $1 \ldots M-1$.

ADTs : Primes and Chaining

- If the size of our array, M, was even and the probe decrement was chosen to be 2, then only half of the locations could be probed.

- Often we choose our table size to be a prime number and our probe decrement to be a number in the range $1 \ldots M - 1$.
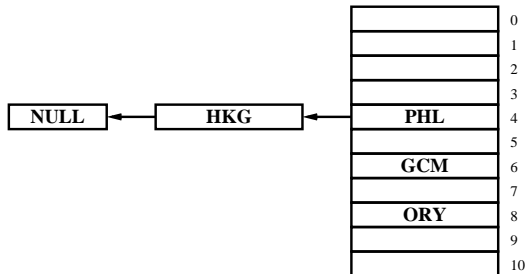
Open-addressing is not the only method of collision reduction. Another common one is separate chaining.

ADTs : A Practical Hash Function

```
1    #include <stdio.h>
2
3    int hash(unsigned int sz, char *s);
4
5    int main(void)
6    {
7
8        char str[] = "Hello World!";
9        // Hash modulus 7919
10       printf("%d\n", hash(7919, str));
11       return 0;
12
13   }
14
15   /*
16   Modified Bernstein hashing
17   5381 & 33 are magic numbers required by the algorithm
18   */
19   int hash(unsigned int sz, char *s)
20   {
21       unsigned long hash = 5381;
22       int c;
23       while((c = (*s++))){
24           hash = 33 * hash ^ c;
25       }
26       return (int)(hash%sz);
27   }
```

Execution :

5479

```c
#include <stdio.h>

int hash(unsigned int sz, char *s);

int main(void)
{

    char str[] = "Hello World!";
    // Hash modulus 7919
    printf("%d\n", hash(7919, str));
    return 0;

}

/*
Modified Bernstein hashing
5381 & 33 are magic numbers required by the algorithm
*/
int hash(unsigned int sz, char *s)
{
    unsigned long hash = 5381;
    int c;
    while((c = (*s++))){
        hash = 33 * hash ^ c;
    }
    return (int)(hash%sz);
}
```

Execution :

5479

Has similarities to the implementation of `rand()` :

```c
#include <stdio.h>

int rand_r(unsigned int* seed);

int main(void)
{
    unsigned int seed = 0;
    printf("%d\n", rand_r(&seed));
    return 0;
}

/* This algorithm is mentioned in the ISO C standard,
   here extended for 32 bits. */
int rand_r(unsigned int* seed)
{
    unsigned int next = *seed;
    int result;

    next *= 1103515245;
    next += 12345;
    result = (unsigned int) (next / 65536) % 2048;

    next *= 1103515245;
    next += 12345;
    result <<= 10;
```

Execution :

1012484

ADTs : Cuckoo Hashing

- We have two tables, each with their **own** hash function.

```
Empty: copied farandoles into table 0(4)
Empty: copied bronzine into table 0(12)
Empty: copied auscultatory into table 0(5)
Empty: copied bifer into table 0(13)
Empty: copied steepgrass into table 0(6)
Empty: copied prevised into table 0(7)
Empty: copied oomph into table 0(8)
empodium, so cuckooed out auscultatory from table 0(5)
Empty: copied auscultatory into table 1(10)
interquarreled, so cuckooed out bronzine from table 0(12)
Empty: copied bronzine into table 1(5)
ranseur, so cuckooed out empodium from table 0(5)
Empty: copied empodium into table 1(4)
Empty: copied megalodon into table 0(11)
geosynchronous, so cuckooed out megalodon from table 0(11)
Empty: copied megalodon into table 1(14)
Empty: copied osmeteria into table 0(14)
Table getting full -> rehashed old sz =16
```

ADTs : Cuckoo Hashing

- We have two tables, each with their **own** hash function.
- We only need to check two cells when searching.

```
Empty: copied farandoles into table 0(4)
Empty: copied bronzine into table 0(12)
Empty: copied auscultatory into table 0(5)
Empty: copied bifer into table 0(13)
Empty: copied steepgrass into table 0(6)
Empty: copied prevised into table 0(7)
Empty: copied oomph into table 0(8)
empodium, so cuckooed out auscultatory from table 0(5)
Empty: copied auscultatory into table 1(10)
interquarreled, so cuckooed out bronzine from table 0(12)
Empty: copied bronzine into table 1(5)
ranseur, so cuckooed out empodium from table 0(5)
Empty: copied empodium into table 1(4)
Empty: copied megalodon into table 0(11)
geosynchronous, so cuckooed out megalodon from table 0(11)
Empty: copied megalodon into table 1(14)
Empty: copied osmeteria into table 0(14)
Table getting full -> rehashed old sz =16
```

ADTs : Cuckoo Hashing

- We have two tables, each with their **own** hash function.
- We only need to check two cells when searching.
- On collision, the existing item is 'cuckooed' out of it's cell into the other table.

```
Empty: copied farandoles into table 0(4)
Empty: copied bronzine into table 0(12)
Empty: copied auscultatory into table 0(5)
Empty: copied bifer into table 0(13)
Empty: copied steepgrass into table 0(6)
Empty: copied prevised into table 0(7)
Empty: copied oomph into table 0(8)
empodium, so cuckooed out auscultatory from table 0(5)
Empty: copied auscultatory into table 1(10)
interquarreled, so cuckooed out bronzine from table 0(12)
Empty: copied bronzine into table 1(5)
ranseur, so cuckooed out empodium from table 0(5)
Empty: copied empodium into table 1(4)
Empty: copied megalodon into table 0(11)
geosynchronous, so cuckooed out megalodon from table 0(11)
Empty: copied megalodon into table 1(14)
Empty: copied osmeteria into table 0(14)
Table getting full -> rehashed old sz =16
```

ADTs : Cuckoo Hashing

- We have two tables, each with their **own** hash function.
- We only need to check two cells when searching.
- On collision, the existing item is 'cuckooed' out of it's cell into the other table.

```
Empty: copied farandoles into table 0(4)
Empty: copied bronzine into table 0(12)
Empty: copied auscultatory into table 0(5)
Empty: copied bifer into table 0(13)
Empty: copied steepgrass into table 0(6)
Empty: copied prevised into table 0(7)
Empty: copied oomph into table 0(8)
empodium, so cuckooed out auscultatory from table 0(5)
Empty: copied auscultatory into table 1(10)
interquarreled, so cuckooed out bronzine from table 0(12)
Empty: copied bronzine into table 1(5)
ranseur, so cuckooed out empodium from table 0(5)
Empty: copied empodium into table 1(4)
Empty: copied megalodon into table 0(11)
geosynchronous, so cuckooed out megalodon from table 0(11)
Empty: copied megalodon into table 1(14)
Empty: copied osmeteria into table 0(14)
Table getting full -> rehashed old sz =16
```

ADTs : Cuckoo Hashing

- We have two tables, each with their **own** hash function.
- We only need to check two cells when searching.
- On collision, the existing item is 'cuckooed' out of it's cell into the other table.

Empty: copied farandoles into table 0(4)
Empty: copied bronzine into table 0(12)
Empty: copied auscultatory into table 0(5)
Empty: copied bifer into table 0(13)
Empty: copied steepgrass into table 0(6)
Empty: copied prevised into table 0(7)
Empty: copied oomph into table 0(8)
empodium, so cuckooed out auscultatory from table 0(5)
Empty: copied auscultatory into table 1(10)
interquarreled, so cuckooed out bronzine from table 0(12)
Empty: copied bronzine into table 1(5)
ranseur, so cuckooed out empodium from table 0(5)
Empty: copied empodium into table 1(4)
Empty: copied megalodon into table 0(11)
geosynchronous, so cuckooed out megalodon from table 0(11)
Empty: copied megalodon into table 1(14)
Empty: copied osmeteria into table 0(14)
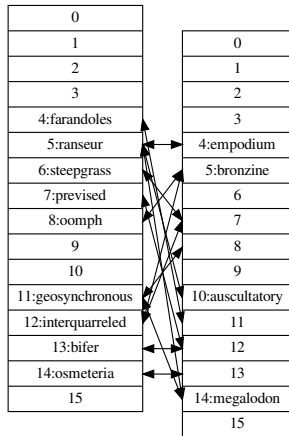Table getting full -> rehashed old sz =16

Table of Contents

ADTs : Graphs

- A graph, G, consists of a set of vertices (nodes), V, together with a set of edges (links), E, each of which connects two vertices.
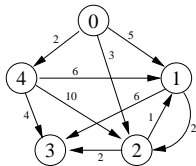
ADTs : Graphs

- A graph, G, consists of a set of vertices (nodes), V, together with a set of edges (links), E, each of which connects two vertices.

- This is a directed graph (digraph). Vertices are joined to adjacent vertices by these edges.
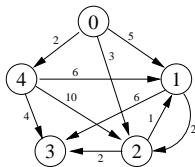
ADTs : Graphs

- A graph, *G*, consists of a set of vertices (nodes), *V*, together with a set of edges (links), *E*, each of which connects two vertices.

- This is a directed graph (digraph). Vertices are joined to adjacent vertices by these edges.

- Every edge has a non-negative weight attached which may correspond to time, distance, cost etc.
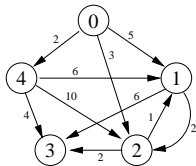
ADTs : Graphs

- A graph, G, consists of a set of vertices (nodes), V, together with a set of edges (links), E, each of which connects two vertices.

- This is a directed graph (digraph). Vertices are joined to adjacent vertices by these edges.

- Every edge has a non-negative weight attached which may correspond to time, distance, cost etc.
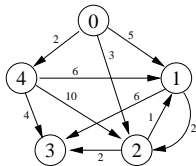
ADTs : Graphs

- A graph, G, consists of a set of vertices (nodes), V, together with a set of edges (links), E, each of which connects two vertices.

- This is a directed graph (digraph). Vertices are joined to adjacent vertices by these edges.

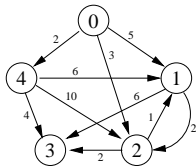- Every edge has a non-negative weight attached which may correspond to time, distance, cost etc.

`graph.h` (partial)

```c
#include <limits.h>
#define INF (INT_MAX)

/* Initialise an empty graph */
graph* graph_init(void);

/* Add new vertex */
int graph_addVert(graph* g, char* label);

/* Add new edge between two Vertices */
bool graph_addEdge(graph* g,  int from,
                   int to, edge weight);

/* Returns NO_VERT if not already a vert
   else 0 ... (size-1)              */
int graph_getVertNum(graph* g, char* label);

/* Returns label of vertex v */
char* graph_getLabel(graph* g, int v);

/* Returns edge weight - if none = INF */
edge graph_getEdgeWeight(graph* g, int from, int to);

/* Number of verts */
int graph_numVerts(graph* b);

/* Output edge weights e.g. "0->1 200 2->1 100" */
void graph_tostring(graph* g, char* str);

/* Clear all memory associated with graph */
bool graph_free(graph* g);
```

Graph ADT : 2D Realloc I

The graph type could be implemented in a large number of different ways.

- As two sets, one for vertices, one for edges. We haven't looked at an implentation for sets, but one could use lists.

Graph ADT : 2D Realloc I

The graph type could be implemented in a large number of different ways.

- As two sets, one for vertices, one for edges. We haven't looked at an implentation for sets, but one could use lists.
- As an adjacency table - simply encode the weighted edges in a 2D array.

	0	**1**	**2**	**3**	**4**
0	0	5	3	∞	2
1	∞	0	2	6	∞
2	∞	1	0	2	∞
3	∞	∞	∞	0	∞
4	∞	6	10	4	0

Graph ADT : 2D Realloc I

The graph type could be implemented in a large number of different ways.

- As two sets, one for vertices, one for edges. We haven't looked at an implentation for sets, but one could use lists.
- As an adjacency table - simply encode the weighted edges in a 2D array.

	0	**1**	**2**	**3**	**4**
0	0	5	3	∞	2
1	∞	0	2	6	∞
2	∞	1	0	2	∞
3	∞	∞	∞	0	∞
4	∞	6	10	4	0

The graph type could be implemented in a large number of different ways.

- As two sets, one for vertices, one for edges. We haven't looked at an implentation for sets, but one could use lists.
- As an adjacency table - simply encode the weighted edges in a 2D array.

	0	**1**	**2**	**3**	**4**
0	0	5	3	∞	2
1	∞	0	2	6	∞
2	∞	1	0	2	∞
3	∞	∞	∞	0	∞
4	∞	6	10	4	0

specific.h

```
1    #define GRAPHTYPE "Realloc"
2
3    #define INITSIZE 8
4    #define SCALEFACTOR 2
5
6    #define TMPSTR 1000
7
8    #define NO_VERT -1
9
10   typedef unsigned int edge;
11
12   struct graph {
13       edge** adjMat;
14       char** labels;
15       /* Actual number of verts */
16       int size;
17       /* Max verts before realloc() */
18       int capacity;
19   };
20   typedef struct graph graph;
```

2D Realloc II

```c
graph* graph_init(void)
{
    graph* g = (graph*) ncalloc(sizeof(graph), 1);
    int h = INITSIZE;
    int w = h;
    g->capacity = h;
    g->adjMat = (edge**) n2dcalloc(h, w, sizeof(edge));
    g->labels = (char**) n2dcalloc(h, MAXLABEL+1, sizeof(char));
    for(int j=0; j<h; j++){
        for(int i=0; i<w; i++){
            /* It's not clear if weight[j][j] should be 0 or INF */
            g->adjMat[j][i] = INF;
        }
    }
    return g;
}

edge graph_getEdgeWeight(graph* g, int from, int to)
{
    if((g==NULL) || (from >= g->size) || (to >= g->size)){
        return INF;
    }
    return g->adjMat[from][to];
}


int graph_numVerts(graph* g)
{
    if(g==NULL){
        return 0;
    }
    return g->size;
}
```

2D Realloc II

```c
graph* graph_init(void)
{
    graph* g = (graph*) ncalloc(sizeof(graph), 1);
    int h = INITSIZE;
    int w = h;
    g->capacity = h;
    g->adjMat = (edge**) n2dcalloc(h, w, sizeof(edge));
    g->labels = (char**) n2dcalloc(h, MAXLABEL+1, sizeof(char));
    for(int j=0; j<h; j++){
        for(int i=0; i<w; i++){
            /* It's not clear if weight[j][j] should be 0 or INF */
            g->adjMat[j][i] = INF;
        }
    }
    return g;
}

edge graph_getEdgeWeight(graph* g, int from, int to)
{
    if((g==NULL) || (from >= g->size) || (to >= g->size)){
        return INF;
    }
    return g->adjMat[from][to];
}


int graph_numVerts(graph* g)
{
    if(g==NULL){
        return 0;
    }
    return g->size;
}
```

```c
int graph_addVert(graph* g, char* label)
{
    if(g==NULL){
        return NO_VERT;
    }
    if(graph_getVertNum(g, label) != NO_VERT){
        return NO_VERT;
    }
    /* Resize */
    if(g->size >= g->capacity){
        g->adjMat = (edge**) n2drecalloc((void**)g->adjMat,
                        g->capacity, g->capacity*SCALEFACTOR,
                        g->capacity, g->capacity*SCALEFACTOR,
                        sizeof(edge));
        g->labels = (char**) n2drecalloc((void**)g->labels,
                        g->capacity, g->capacity*SCALEFACTOR,
                        MAXLABEL+1, MAXLABEL+1,  1);
        for(int j=0; j<g->capacity*SCALEFACTOR; j++){
            for(int i=0; i<g->capacity*SCALEFACTOR; i++){
                if((i>=g->capacity)||(j>=g->capacity)){
                    g->adjMat[j][i] = INF;
                }
            }
        }
        g->capacity = g->capacity*SCALEFACTOR;
    }
    strcpy(g->labels[g->size], label);
    g->size = g->size + 1;
    return g->size-1;
}
```

Graph ADT - Linked
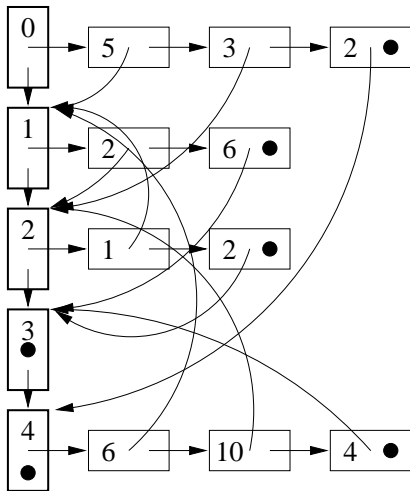
Graph ADT - Linked
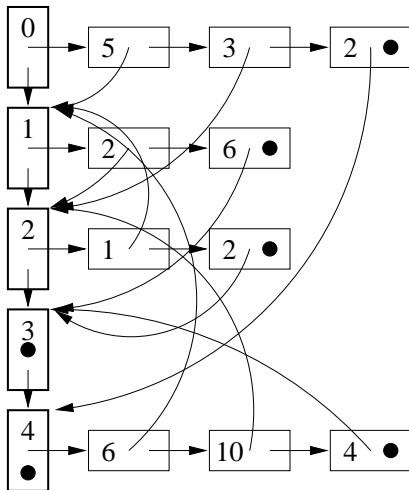
specific.h

```
1    #define GRAPHTYPE "Linked"
2
3    #define INITSIZE 8
4    #define SCALEFACTOR 2
5
6    #define TMPSTR 1000
7
8    #define NO_VERT -1
9
10   typedef unsigned int edge;
11
12   struct vertex {
13       char* label;
14       struct vertex* nextv;
15       void* firste;
16       int num;
17   };
18   typedef struct vertex vertex;
19
20   struct edge {
21       edge weight;
22       vertex* v;
23       struct edge* nexte;
24   };
25   typedef struct edge edgel;
26
27   struct graph {
28       vertex* firstv;
29       vertex* endv;
30       int size;
31   };
32   typedef struct graph graph;
```

```
graph* graph_init(void)
{
    graph* g = (graph*) ncalloc(1, sizeof(graph));
    return g;
}

edge graph_getEdgeWeight(graph* g, int from, int to)
{
    if((g==NULL) || (from >= g->size) || (to >= g->size)){
        return INF;
    }
    vertex* v = g->firstv;
    for(int i=0; i<from; i++){
        v = v->nextv;
    }
    if((v==NULL) || (v->num != from)){
        return INF;
    }
    edgel* e = v->firste;
    while(e != NULL){
        if(e->v->num == to){
            return e->weight;
        }
        e = e->nexte;
    }
    return INF;
}
```

```
graph* graph_init(void)
{
    graph* g = (graph*) ncalloc(1, sizeof(graph));
    return g;
}

edge graph_getEdgeWeight(graph* g, int from, int to)
{
    if((g==NULL) || (from >= g->size) || (to >= g->size)){
        return INF;
    }
    vertex* v = g->firstv;
    for(int i=0; i<from; i++){
        v = v->nextv;
    }
    if((v==NULL) || (v->num != from)){
        return INF;
    }
    edgel* e = v->firste;
    while(e != NULL){
        if(e->v->num == to){
            return e->weight;
        }
        e = e->nexte;
    }
    return INF;
}
```

```
bool graph_addEdge(graph* g, int from, int to, edge w)
{
    if((g==NULL) || (g->size == 0)){
        return false;
    }
    if((from >= g->size) || (to >= g->size)){
        return false;
    }
    vertex* f = g->firstv;
    for(int i=0; i<from; i++){
        f = f->nextv;
    }
    vertex* t = g->firstv;
    for(int i=0; i<to; i++){
        t = t->nextv;
    }
    return _addEdge(f, t, w);
}
```