COMSM1201 : Data Structures & Algorithms

Neill.Campbell@bristol.ac.uk

University of Bristol

Built : November 25, 2024

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

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

void strrev(char* s, int n);

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

/* Iterative Inplace String Reverse */
void strrev(char* s, int n)
{
    for(int i=0, j=n-1; i<j; i++, j--){
        SWAP(s[i], s[j]);
    }
}
```

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);
}
```

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);
}
```

Execution :

```
!dlroW olleH
```

- We need to change the function prototype.

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);
}
```

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
```

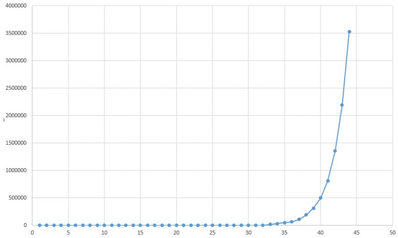
Iterative & Recursive Fibonacci

```
#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 == 1) return 1;
    if(n == 2) return 1;
    return( fibonacci(n-1)+fibonacci(n-2));
}
```

Iterative & Recursive Fibonacci

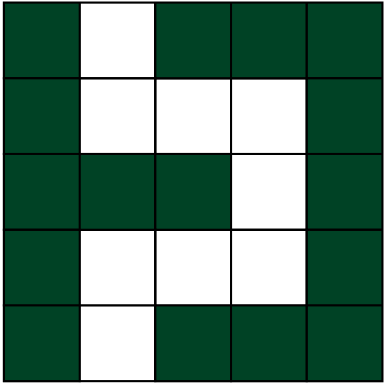
```c
#include <stdio.h>

#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 == 1) return 1;
    if(n == 2) return 1;
    return( fibonacci(n-1)+fibonacci(n-2));
}
```

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



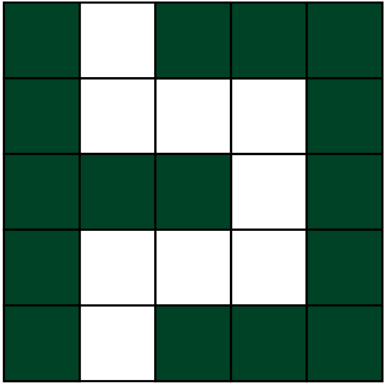
Maze Escape

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
// From e.g.  http://www.geeksforgeeks.org
#include <stdio.h>
#include <string.h>

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

void permute(char* a, int s, int e);

int main()
{
    char str[] = "ABC";
    int n = strlen(str);
    permute(str, 0, n-1);
    return 0;
}

void permute(char* a, int s, int e)
{
    if (s == e){
      printf("%s\n", a);
      return;
    }
    for (int i = s; i <= e; i++){
       SWAP((a+s), (a+i)); // Bring one char to the front
       permute(a, s+1, e);
       SWAP((a+s), (a+i)); // Backtrack
    }
}
```

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 = 2^5` 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` = 2^5 is the same as multiple multiplications `n` = 2*2*2*2*2.
- Or, thinking recursively, `n` = 2 * (2^4).

```c
/* Try to write power(a,b) to computer a^b
   without using any maths functions other than
   multiplication :
   Try (1) iterative then (2) recursive
   (3) Trick that for n%2==0, x^n = x^(n/2)*x^(n/2)

*/

#include <stdio.h>

int power(unsigned int a, unsigned int b);

int main(void)
{

   int x = 2;
   int y = 16;

   printf("%d^%d = %d\n", x, y, power(x,y));

}

int power(unsigned int a, unsigned int b)
{
}
```

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
#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++){
      if(strcmp(name, p[j].name) == 0){
         return p[j].age;
      }
   }
   return NOTFOUND;
}
```

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.

- 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.

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.

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
$\Uparrow$						$\uparrow$							$\Uparrow$
							$\Uparrow$			$\uparrow$			$\Uparrow$
							$\Uparrow$	$\uparrow$	$\Uparrow$				

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

```
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;
}
```

```
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' :

$$\begin{array}{cccccccc} 0 & 4 & 5 & 9 & 10 & 12 & 15 & 20 \\ & & & & & & \Uparrow & \end{array}$$

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' :

$$\begin{array}{cccccccc} 0 & 4 & 5 & 9 & 10 & 12 & 15 & 20 \\ & & & & & & \Uparrow & \end{array}$$

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' :

$$\begin{array}{cccccccc} 0 & 4 & 5 & 9 & 10 & 12 & 15 & 20 \\ & & & & & & \Uparrow & \end{array}$$

```
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
#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:

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
#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
  - 0.00 seconds using an aggressive optimization -O3

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
  - 0.00 seconds using an aggressive optimization -O3
- But "wall-clock" time is generally not the thing that excites Computer Scientists.

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
  - 0.00 seconds using an aggressive optimization -O3
- But "wall-clock" time is generally not the thing that excites Computer Scientists.

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
  - 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.

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
  - 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.

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
  - 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)**

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
  - 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)**

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
  - 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
#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
  - 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

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

int bin_it(int k, const int *a, int l, int r);
int bin_rec(int k, const int *a, int l, int r);
int interp(int k, const int *a, int l, int r);
int* parse_args(int argc, char* argv[], int* n, int* srch);

int main(int argc, char* argv[])
{

   int i, n, srch;
   int* a;
   int (*p[3])(int k, const int*a, int l, int r) =
       {bin_it, bin_rec, interp};

   a = parse_args(argc, argv, &n, &srch);

   srand(time(NULL));
   for(i=0; i<n; i++){
      a[i] = 2*i;
   }
   for(i=0; i<5000000; i++){
      assert((*p[srch])(a[rand()%n], a, 0, n-1) >= 0);
   }

   free(a);
   return 0;

}
```

Binary vs. Interpolation Timing

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

int bin_it(int k, const int *a, int l, int r);
int bin_rec(int k, const int *a, int l, int r);
int interp(int k, const int *a, int l, int r);
int* parse_args(int argc, char* argv[], int* n, int* srch);

int main(int argc, char* argv[])
{

    int i, n, srch;
    int* a;
    int (*p[3])(int k, const int*a, int l, int r) =
        {bin_it, bin_rec, interp};

    a = parse_args(argc, argv, &n, &srch);

    srand(time(NULL));
    for(i=0; i<n; i++){
        a[i] = 2*i;
    }
    for(i=0; i<5000000; i++){
        assert((*p[srch])(a[rand()%n], a, 0, n-1) >= 0);
    }

    free(a);
    return 0;

}
```

Execution :

```
Binary Search : Iterative
n =    100000 = 0.39
n =    800000 = 0.57
n =   6400000 = 1.00
n =  51200000 = 2.46
Binary Search : Recursive
n =    100000 = 0.40
n =    800000 = 0.56
n =   6400000 = 0.97
n =  51200000 = 2.42
Interpolation
n =    100000 = 0.05
n =    800000 = 0.05
n =   6400000 = 0.10
n =  51200000 = 0.13
```

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);
   return 0; // Should Free List
}
```

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");
}
```

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*. 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.)

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.)
- Actually, I'll sometimes blur the boundaries of Data Structures (e.g. a linked list) with ADTs (e.g. a dictionary) themselves.

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.

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

typedef int colltype;

typedef struct coll coll;

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

// Create an empty coll
coll* coll_init(void);
// Add element onto top
void coll_add(coll* c, colltype i);
// Take element out
bool coll_remove(coll* c, colltype d);
// Does this exist ?
bool coll_isin(coll* c, colltype i);
// Return size of coll
int coll_size(coll* c);
// Clears all space used
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:

```c
#pragma once

#define COLLTYPE "Fixed"

#define FIXEDSIZE 5000
struct coll {
   // Underlying array
   colltype a[FIXEDSIZE];
   int size;
};
```

Collection ADT using a Fixed-size Array

Fixed/fixed.c:

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

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

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

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

Collection ADT using a Fixed-size Array

Fixed/fixed.c:

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

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

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

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

```c
void coll_add(coll* c, colltype d)
{
   if(c){
      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:

```c
#pragma once

#define COLLTYPE "Realloc"

#define INITSIZE 16
#define SCALEFACTOR 2
struct coll {
   // Underlying array
   colltype* a;
   int size;
   int capacity;
};
```

Collection ADT via an Array (Realloc)

Realloc/specific.h:

```
#pragma once

#define COLLTYPE "Realloc"

#define INITSIZE 16
#define SCALEFACTOR 2
struct coll {
   // Underlying array
   colltype* a;
   int size;
   int capacity;
};
```

Realloc/realloc.c:

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

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

void coll_add(coll* c, colltype d)
{
   if(c){
      if(c->size >= c->capacity){
         c->a = (colltype*) nremalloc(c->a,
                sizeof(colltype)*c->capacity*SCALEFACTOR);
         c->capacity = c->capacity*SCALEFACTOR;
      }
      c->a[c->size] = d;
      c->size = c->size + 1;
   }
}
```

Collection ADT via a Linked List

Linked/specific.h:

```c
#pragma once

#define COLLTYPE "Linked"

struct dataframe {
   colltype i;
   struct dataframe* next;
};
typedef struct dataframe dataframe;

struct coll {
   // Underlying array
   dataframe* start;
   int size;
};
```

Collection ADT via a Linked List

Linked/specific.h:

```c
#pragma once

#define COLLTYPE "Linked"

struct dataframe {
   colltype i;
   struct dataframe* next;
};
typedef struct dataframe dataframe;

struct coll {
   // Underlying array
   dataframe* start;
   int size;
};
```

Linked/linked.c:

```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

```
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

Linked List code from the previous Chapter :

```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);
   return 0; // Should Free List
}
```

ADTs Making Coding Simpler

Linked List code from the previous Chapter :

```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);
   return 0; // Should Free List
}
```

Becomes :

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

int main(void)
{
   printf("Please type some numbers :");
   coll* c = coll_init();
   int i;
   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:

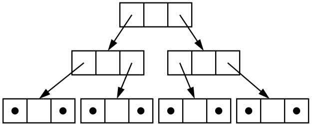
- Collections (Lists)
- Stacks
- Queues
- Sets
- Graphs
- Trees

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

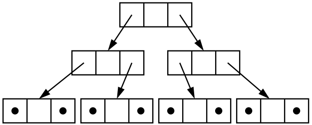
Binary Trees:



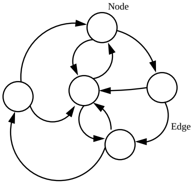
At the highest level of abstraction, ADTs that we can represent using both dynamic structures (pointers) and also fixed structures (arrays) include:

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

Binary Trees:

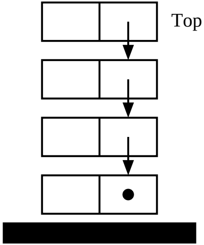


Unidirectional Graph:



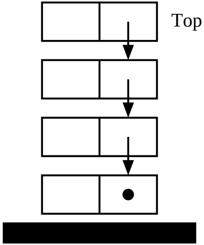
Stacks

The push-down stack:

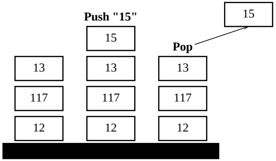


Stacks

The push-down stack:



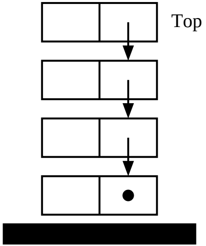
LIFO (Last in, First out):



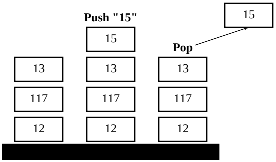
- Operations include `push` and `pop`.

Stacks

The push-down stack:



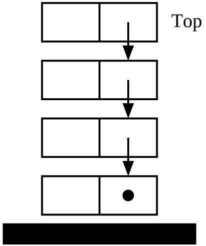
LIFO (Last in, First out):



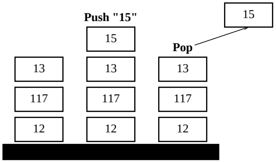
- Operations include `push` and `pop`.
- In the C run-time system, function calls are implemented using stacks.

Stacks

The push-down stack:



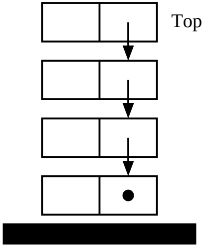
LIFO (Last in, First out):



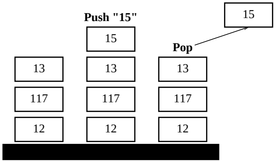
- Operations include `push` and `pop`.
- In the C run-time system, function calls are implemented using stacks.
- Most recursive algorithms can be re-written using stacks instead.

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:

```
#pragma once

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

typedef int stacktype;

typedef struct stack stack;

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

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

/* Optional? */

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

ADT:Stacks Arrays (Realloc) I

stack.h:

```
#pragma once

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

typedef int stacktype;

typedef struct stack stack;

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

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

/* Optional? */

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

Realloc/specific.h:

```
#pragma once

#define FORMATSTR "%i"
#define ELEMSIZE 20

#define STACKTYPE "Realloc"

#define FIXEDSIZE 16
#define SCALEFACTOR 2

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

ADT:Stacks Arrays (Realloc) II

Realloc/realloc.c

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

#define DOTFILE 5000

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

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

ADT:Stacks Arrays (Realloc) II

Realloc/realloc.c

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

#define DOTFILE 5000

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

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

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

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

ADT:Stacks Arrays (Realloc) III

Realloc/realloc.c

```c
void stack_tostring(stack* s, char* str)
{
   char tmp[ELEMSIZE];
   str[0] = '\0';
   if((s==NULL) || (s->size <1)){
      return;
   }
   for(int i=s->size-1; i>=0; i--){
      sprintf(tmp, FORMATSTR, s->a[i]);
      strcat(str, tmp);
      strcat(str, "|");
   }
   str[strlen(str)-1] = '\0';
}

bool stack_free(stack* s)
{
   if(s==NULL){
      return true;
   }
   free(s->a);
   free(s);
   return true;
}
```

ADT:Stacks Arrays (Realloc) III

Realloc/realloc.c

```c
void stack_tostring(stack* s, char* str)
{
   char tmp[ELEMSIZE];
   str[0] = '\0';
   if((s==NULL) || (s->size <1)){
      return;
   }
   for(int i=s->size-1; i>=0; i--){
      sprintf(tmp, FORMATSTR, s->a[i]);
      strcat(str, tmp);
      strcat(str, "|");
   }
   str[strlen(str)-1] = '\0';
}

bool stack_free(stack* s)
{
   if(s==NULL){
      return true;
   }
   free(s->a);
   free(s);
   return true;
}
```

- We need a thorough testing program teststack.c

Realloc/realloc.c

```
void stack_tostring(stack* s, char* str)
{
   char tmp[ELEMSIZE];
   str[0] = '\0';
   if((s==NULL) || (s->size <1)){
      return;
   }
   for(int i=s->size-1; i>=0; i--){
      sprintf(tmp, FORMATSTR, s->a[i]);
      strcat(str, tmp);
      strcat(str, "|");
   }
   str[strlen(str)-1] = '\0';
}

bool stack_free(stack* s)
{
   if(s==NULL){
      return true;
   }
   free(s->a);
   free(s);
   return true;
}
```

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

ADT:Stacks Linked I

Linked/specific.h

```c
#pragma once

#define FORMATSTR "%i"
#define ELEMSIZE 20
#define STACKTYPE "Linked"

struct dataframe {
   stacktype i;
   struct dataframe* next;
};
typedef struct dataframe dataframe;

struct stack {
   /* Underlying array */
   dataframe* start;
   int size;
};
```

ADT:Stacks Linked I

Linked/specific.h

```c
#pragma once

#define FORMATSTR "%i"
#define ELEMSIZE 20
#define STACKTYPE "Linked"

struct dataframe {
   stacktype i;
   struct dataframe* next;
};
typedef struct dataframe dataframe;

struct stack {
   /* Underlying array */
   dataframe* start;
   int size;
};
```

Linked/linked.c

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

#define DOTFILE 5000

stack* stack_init(void)
{
   stack* s = (stack*) ncalloc(1, sizeof(stack));
   return s;
}

void stack_push(stack* s, stacktype d)
{
   if(s){
      dataframe* f = ncalloc(1, sizeof(dataframe));
      f->i = d;
      f->next = s->start;
      s->start = f;
      s->size = s->size + 1;
   }
}
```

ADT:Stacks Linked II

```c
bool stack_pop(stack* s, stacktype* d)
{
   if((s==NULL) || (s->start==NULL)){
      return false;
   }

   dataframe* f = s->start->next;
   *d = s->start->i;
   free(s->start);
   s->start = f;
   s->size = s->size - 1;
   return true;
}

bool stack_peek(stack* s, stacktype* d)
{
   if((s==NULL) || (s->start==NULL)){
      return false;
   }
   *d = s->start->i;
   return true;
}
```

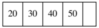
ADT:Stacks Linked II

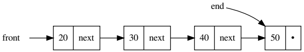
```c
bool stack_pop(stack* s, stacktype* d)
{
    if((s==NULL) || (s->start==NULL)){
        return false;
    }

    dataframe* f = s->start->next;
    *d = s->start->i;
    free(s->start);
    s->start = f;
    s->size = s->size - 1;
    return true;
}

bool stack_peek(stack* s, stacktype* d)
{
    if((s==NULL) || (s->start==NULL)){
        return false;
    }
    *d = s->start->i;
    return true;
}
```

```c
void stack_tostring(stack* s, char* str)
{
    char tmp[ELEMSIZE];
    str[0] = '\0';
    if((s==NULL) || (s->size <1)){
        return;
    }
    dataframe* p = s->start;
    while(p){
        sprintf(tmp, FORMATSTR, p->i);
        strcat(str, tmp);
        strcat(str, "|");
        p = p->next;
    }
    str[strlen(str)-1] = '\0';
}

bool stack_free(stack* s)
{
    if(s){
        dataframe* p = s->start;
        while(p!=NULL){
            dataframe* tmp = p->next;
            free(p);
            p = tmp;
        }
        free(s);
    }
    return true;
}
```

Table of Contents

FIFO (First in, First out):

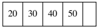


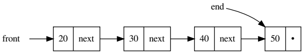


- Intuitively more "useful" than a stack.

ADTs : Queues

FIFO (First in, First out):

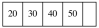


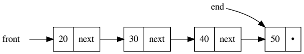


- Intuitively more “useful” than a stack.
- Think of implementing any kind of service (printer, web etc.)

ADTs : Queues

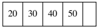
FIFO (First in, First out):

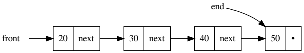




- Intuitively more "useful" than a stack.
- Think of implementing any kind of service (printer, web etc.)
- Operations include `enqueue`, `dequeue` and `size`.

FIFO (First in, First out):

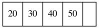


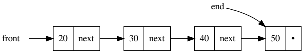


- Intuitively more “useful” than a stack.
- Think of implementing any kind of service (printer, web etc.)
- Operations include `enqueue`, `dequeue` and `size`.

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.

queue.h

```c
#pragma once

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

typedef int queuetype;

typedef struct queue queue;

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

/* Create an empty queue */
queue* queue_init(void);
/* Add element on end */
void queue_enqueue(queue* q, queuetype v);
/* Take element off front */
bool queue_dequeue(queue* q, queuetype* d);
/* Return size of queue */
int queue_size(queue* q);
/* Clears all space used */
bool queue_free(queue* q);

/* Helps with visualisation & testing */
void queue_tostring(queue* q, char* str);
```

ADTs : Queues (Fixed) I

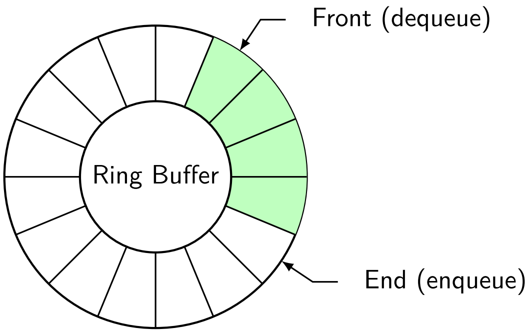
specific.h

```c
#pragma once

#define FORMATSTR "%d"
#define ELEMSIZE 20

#define QUEUETYPE "Fixed"

#define BOUNDED 5000

struct queue {
   /* Underlying array */
   queuetype a[BOUNDED];
   int front;
   int end;
};

#define DOTFILE 5000
```

ADTs : Queues (Fixed) I

specific.h

```
#pragma once

#define FORMATSTR "%d"
#define ELEMSIZE 20

#define QUEUETYPE "Fixed"

#define BOUNDED 5000

struct queue {
    /* Underlying array */
    queuetype a[BOUNDED];
    int front;
    int end;
};

#define DOTFILE 5000
```



ADTs : Queues (Fixed) II

fixed.c

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

void _inc(int* p);

queue* queue_init(void)
{
   queue* q = (queue*) ncalloc(1, sizeof(queue));
   return q;
}


void queue_enqueue(queue* q, queuetype d)
{
   if(q){
      q->a[q->end] = d;
      _inc(&q->end);
      if(q->end == q->front){
         on_error("Queue too large");
      }
   }
}
```

ADTs : Queues (Fixed) II

fixed.c

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

void _inc(int* p);

queue* queue_init(void)
{
   queue* q = (queue*) ncalloc(1, sizeof(queue));
   return q;
}


void queue_enqueue(queue* q, queuetype d)
{
   if(q){
      q->a[q->end] = d;
      _inc(&q->end);
      if(q->end == q->front){
         on_error("Queue too large");
      }
   }
}
```

```c
bool queue_dequeue(queue* q, queuetype* d)
{
   if((q==NULL) || (q->front==q->end)){
      return false;
   }
   *d = q->a[q->front];
   _inc(&q->front);
   return true;
}

void queue_tostring(queue* q, char* str)
{
   char tmp[ELEMSIZE];
   str[0] = '\0';
   if((q==NULL) || (queue_size(q)==0)){
      return;
   }
   for(int i=q->front; i != q->end;){
      sprintf(tmp, FORMATSTR, q->a[i]);
      strcat(str, tmp);
      strcat(str, "|");
      _inc(&i);
   }
   str[strlen(str)-1] = '\0';
}
```

ADTs : Queues (Fixed) III

```
int queue_size(queue* q)
{
   if(q==NULL){
      return 0;
   }
   if(q->end >= q->front){
      return q->end-q->front;
   }
   return q->end + BOUNDED - q->front;
}

bool queue_free(queue* q)
{
   free(q);
   return true;
}

void _inc(int* p)
{
   *p = (*p + 1) % BOUNDED;
}
```

ADTs : Queues (Fixed) III

```c
int queue_size(queue* q)
{
   if(q==NULL){
      return 0;
   }
   if(q->end >= q->front){
      return q->end-q->front;
   }
   return q->end + BOUNDED - q->front;
}

bool queue_free(queue* q)
{
   free(q);
   return true;
}

void _inc(int* p)
{
   *p = (*p + 1) % BOUNDED;
}
```

- We need a thorough testing program

ADTs : Queues (Fixed) III

```c
int queue_size(queue* q)
{
   if(q==NULL){
      return 0;
   }
   if(q->end >= q->front){
      return q->end-q->front;
   }
   return q->end + BOUNDED - q->front;
}

bool queue_free(queue* q)
{
   free(q);
   return true;
}

void _inc(int* p)
{
   *p = (*p + 1) % BOUNDED;
}
```

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

ADTs : Queues (Fixed) III

```c
int queue_size(queue* q)
{
   if(q==NULL){
      return 0;
   }
   if(q->end >= q->front){
      return q->end-q->front;
   }
   return q->end + BOUNDED - q->front;
}

bool queue_free(queue* q)
{
   free(q);
   return true;
}

void _inc(int* p)
{
   *p = (*p + 1) % BOUNDED;
}
```

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

ADTs : Queues (Linked) I

specific.h

```c
#pragma once

#define FORMATSTR "%d"
#define ELEMSIZE 20

#define QUEUETYPE "Linked"

struct dataframe {
   queuetype i;
   struct dataframe* next;
};
typedef struct dataframe dataframe;

struct queue {
   /* Underlying array */
   dataframe* front;
   dataframe* end;
   int size;
};
```

ADTs : Queues (Linked) I

specific.h

```c
#pragma once

#define FORMATSTR "%d"
#define ELEMSIZE 20

#define QUEUETYPE "Linked"

struct dataframe {
   queuetype i;
   struct dataframe* next;
};
typedef struct dataframe dataframe;

struct queue {
   /* Underlying array */
   dataframe* front;
   dataframe* end;
   int size;
};
```

linked.c

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

queue* queue_init(void)
{
   queue* q = (queue*) ncalloc(1, sizeof(queue));
   return q;
}

void queue_enqueue(queue* q, queuetype d)
{
   dataframe* f;
   if(q==NULL){
      return;
   }

   /* Copy the data */
   f = ncalloc(1, sizeof(dataframe));
   f->i = d;

   /* 1st one */
   if(q->front == NULL){
      q->front = f;
      q->end = f;
      q->size = q->size + 1;
      return;
   }
   /* Not 1st */
   q->end->next = f;
   q->end = f;
   q->size = q->size + 1;
}
```

ADTs : Queues (Linked) II

```
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;
}
```

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
  ```

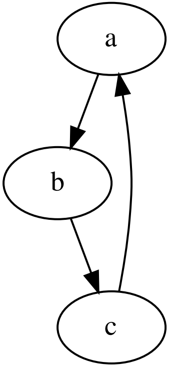


Table of Contents

Binary Trees : Data Structures

- Binary trees are used extensively in computer science

Binary Trees : Data Structures

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

Binary Trees : Data Structures

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

Binary Trees : Data Structures

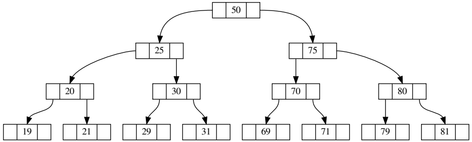
- Binary trees are used extensively in computer science
- Game Trees
- Searching
- Sorting

Binary Trees : Data Structures

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

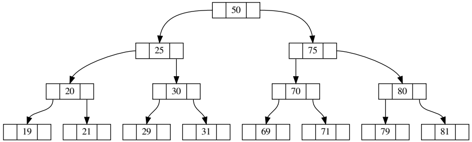
Binary Trees : Data Structures

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



Binary Trees : Data Structures

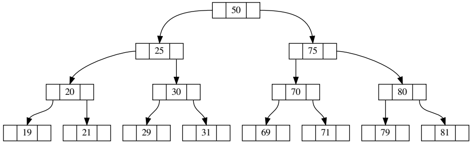
- Binary trees are used extensively in computer science
- Game Trees
- Searching
- Sorting



- Trees drawn upside-down !

Binary Trees : Data Structures

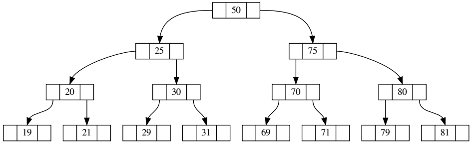
- Binary trees are used extensively in computer science
- Game Trees
- Searching
- Sorting



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

Binary Trees : Data Structures

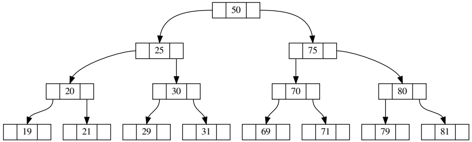
- Binary trees are used extensively in computer science
- Game Trees
- Searching
- Sorting



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

Binary Trees : Data Structures

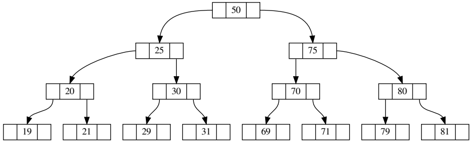
- Binary trees are used extensively in computer science
- Game Trees
- Searching
- Sorting



- 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

Binary Trees : Data Structures

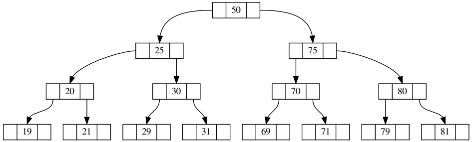
- Binary trees are used extensively in computer science
- Game Trees
- Searching
- Sorting



- 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

Binary Trees : Data Structures

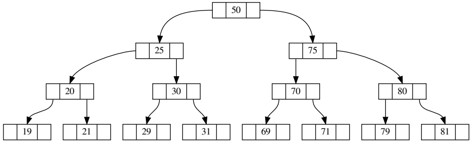
- Binary trees are used extensively in computer science
- Game Trees
- Searching
- Sorting



- 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

Binary Trees : Data Structures

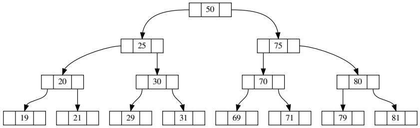
- Binary trees are used extensively in computer science
- Game Trees
- Searching
- Sorting



- 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

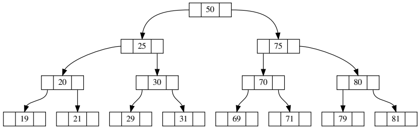
Binary Search Trees

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

```c
#include "../General/general.h"
#include "../Queue/queue.h"

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

bst* bst_init(void);

/* Insert 1 item into the tree */
bool bst_insert(bst* b, treetype d);

/* Return number of nodes in tree */
int bst_size(bst* b);

/* Whether the data d is stored in the tree */
bool bst_isin(bst* b, treetype d);

/* Bulk insert n items from an array a into an initialised tree */
bool bst_insertarray(bst* b, treetype* a, int n);

/* Clear all memory associated with tree, & set pointer to NULL */
bool bst_free(bst* b);

/* Optional ? */

char* bst_preorder(bst* b);
void bst_printlevel(bst* b);
/* Create string with tree as ((head)(left)(right)) */
char* bst_printlisp(bst* b);
/* Use Graphviz via a .dot file */
void bst_todot(bst* b, char* dotname);
```

Binary Search Trees : Linked I

specific.h

```
#include <string.h>

typedef int treetype;
#define FORMATSTR "%i"
#define ELEMSIZE 20
#define BSTTYPE "Linked"

struct dataframe {
   treetype d;
   struct dataframe* left;
   struct dataframe* right;
};
typedef struct dataframe dataframe;

struct bst {
   dataframe* top;
   /* Data element size, in bytes */
};
typedef struct bst bst;
```

Binary Search Trees : Linked I

specific.h

```c
#include <string.h>

typedef int treetype;
#define FORMATSTR "%i"
#define ELEMSIZE 20
#define BSTTYPE "Linked"

struct dataframe {
   treetype d;
   struct dataframe* left;
   struct dataframe* right;
};
typedef struct dataframe dataframe;

struct bst {
   dataframe* top;
   /* Data element size, in bytes */
};
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;
}
```

Binary Search Trees : Linked II

```
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;
}
```

Binary Search Trees : Linked II

```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

```
#include <stdbool.h>

typedef int treetype;
#define FORMATSTR "%i"
#define ELEMSIZE 20
#define BSTTYPE "Realloc"

// Probably (2^n) -1
#define INITSIZE 31
#define SCALEFACTOR 2

struct dataframe {
   treetype d;
   bool isvalid;
};
typedef struct dataframe dataframe;

struct bst {
   dataframe* a;
   int capacity;
};
typedef struct bst bst;
```

Binary Search Trees : Realloc

specific.h

```c
#include <stdbool.h>

typedef int treetype;
#define FORMATSTR "%i"
#define ELEMSIZE 20
#define BSTTYPE "Realloc"

// Probably (2^n) -1
#define INITSIZE 31
#define SCALEFACTOR 2

struct dataframe {
   treetype d;
   bool isvalid;
};
typedef struct dataframe dataframe;

struct bst {
   dataframe* a;
   int capacity;
};
typedef struct bst bst;
```

Using a queue for Level-Order traversal:

```c
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 $O(\texttt{log n})$.

Binary Search Trees : Complexity

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

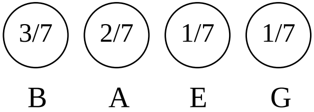
Binary Trees : Huffman Compression I

- 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":

- 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":



- Keep a list of characters, ordered by their frequency

Binary Trees : Huffman Compression II

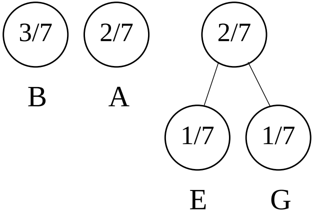
- Use the two least frequent to form a sub-tree, and re-order (sort) the nodes :

Binary Trees : Huffman Compression II

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

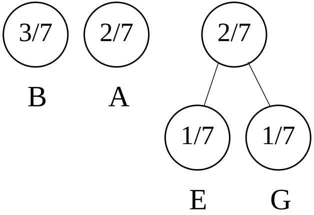
Binary Trees : Huffman Compression II

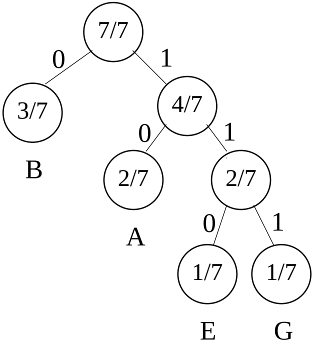
- Use the two least frequent to form a sub-tree, and re-order (sort) the nodes :



Binary Trees : Huffman Compression II

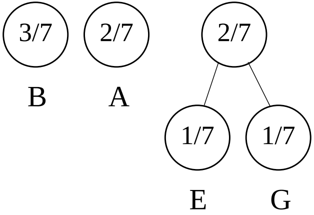
- Use the two least frequent to form a sub-tree, and re-order (sort) the nodes :

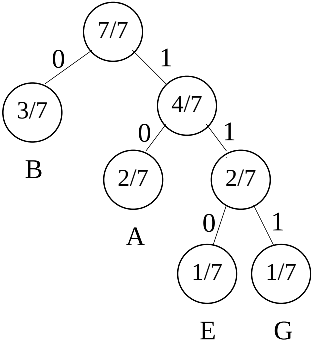




Binary Trees : Huffman Compression II

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

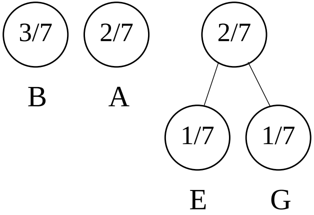


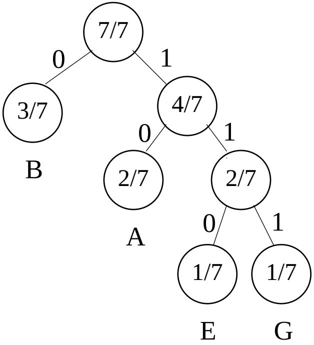


- A = 10, B = 0, E = 110, G = 111

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.

- 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. PEK, BRS, FRA.

Hashing : Aiport Codes

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

Hashing : 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:

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

Hashing : 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":
  $h(''DCA'') =$
  $(3 * 26^2 + 2 * 26 + 0)\%11$
  $h(''DCA'') = (2080)\%11$
  $h(''DCA'') = 1$

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

Hashing : 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

Hashing : Aiport Codes

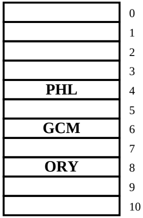
- In a three letter string $X_2X_1X_0$ the letter 'A' has the value 0, 'B' has the value 1 etc.
- One hash function is:

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

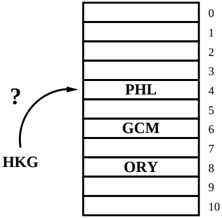
- Applying this to "DCA":

$h(''DCA'') =$
$(3 * 26^2 + 2 * 26 + 0)\%11$
$h(''DCA'') = (2080)\%11$
$h(''DCA'') = 1$

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



- However, inserting "HKG" causes a collision.



Hashing : Collisions

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

Hashing : 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.

Hashing : 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.

Hashing : 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.

Hashing : 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.

Hashing : 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.

Hashing : Collisions

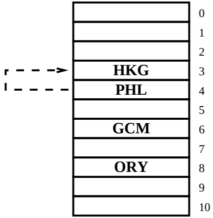
- An ideal hashing function maps keys into the array in a *uniform* and *random* manner.
- Collisions occur when a hash function maps two different keys onto the same address.
- It's very difficult to choose 'good' hashing functions.
- Collisions are common - the **von Mises** paradox. When 23 keys are 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.

Hashing : 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.

Hashing : 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.



Double Hashing

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

Double Hashing

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

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.

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.

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.

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. :

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

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.

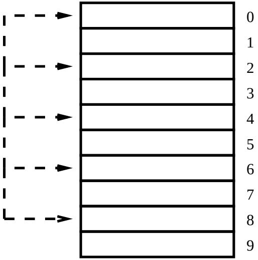
- The function is chosen so that two keys which collide at the same address will have different probe decrements, e.g. :

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

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

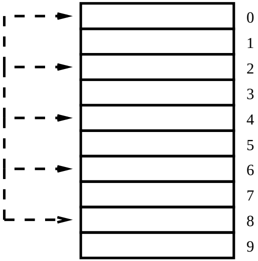
Hashing : Primes and Chaining

- 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.



Hashing : Primes and Chaining

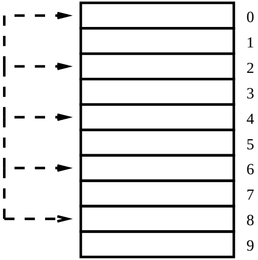
- If the size of our array, M, was even and the probe decrement was chosen to be 2, then only half of the locations could be probed.



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

Hashing : Primes and Chaining

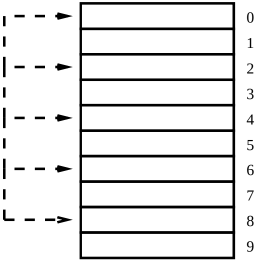
- If the size of our array, M, was even and the probe decrement was chosen to be 2, then only half of the locations could be probed.



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

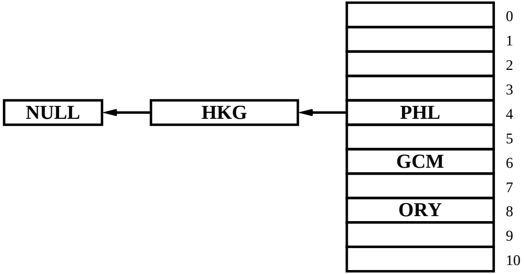
Hashing : Primes and Chaining

- 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.



A Practical Hash Function

```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
```

A Practical Hash Function

```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
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;
  result ^= (unsigned int) (next / 65536) % 1024;
  next *= 1103515245;
  next += 12345;
  result <<= 10;
  result ^= (unsigned int) (next / 65536) % 1024;
  *seed = next;
  return result;
```

Execution :

```
1012484
```

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
```

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
```

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
```

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
```

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
```

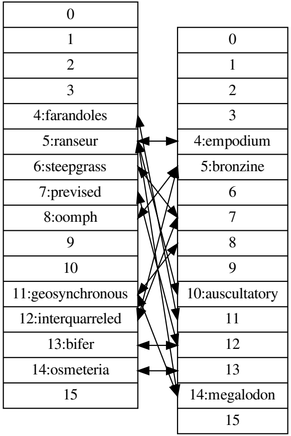


Table of Contents

```c
#define NUMS 6

void bubble_sort(int b[], int s);

int main(void)
{
    int a[] = {3, 4, 1, 2, 9, 0};
    bubble_sort(a, NUMS);
    for(int i=0; i<NUMS; i++){
        printf("%i ", a[i]);
    }
    printf("\n");
    return 0;
}

void bubble_sort(int b[], int s)
{
    bool changes;
    do{
        changes = false;
        for(int i=0; i<s-1; i++){
            if(b[i] > b[i+1]){
                SWAP(b[i], b[i+1]);
                changes = true;
            }
        }
    }while(changes);
}
```

Execution :

0 1 2 3 4 9

```c
#define NUMS 6

void bubble_sort(int b[], int s);

int main(void)
{
    int a[] = {3, 4, 1, 2, 9, 0};
    bubble_sort(a, NUMS);
    for(int i=0; i<NUMS; i++){
        printf("%i ", a[i]);
    }
    printf("\n");
    return 0;
}

void bubble_sort(int b[], int s)
{
    bool changes;
    do{
        changes = false;
        for(int i=0; i<s-1; i++){
            if(b[i] > b[i+1]){
                SWAP(b[i], b[i+1]);
                changes = true;
            }
        }
    }while(changes);
}
```

Execution :

0 1 2 3 4 9

- Bubblesort has complexity $O(n^2)$, therefore very inefficient.

Algorithms : Sorting

```c
#define NUMS 6

void bubble_sort(int b[], int s);

int main(void)
{
    int a[] = {3, 4, 1, 2, 9, 0};
    bubble_sort(a, NUMS);
    for(int i=0; i<NUMS; i++){
        printf("%i ", a[i]);
    }
    printf("\n");
    return 0;
}

void bubble_sort(int b[], int s)
{
    bool changes;
    do{
        changes = false;
        for(int i=0; i<s-1; i++){
            if(b[i] > b[i+1]){
                SWAP(b[i], b[i+1]);
                changes = true;
            }
        }
    }while(changes);
}
```

Execution :

0 1 2 3 4 9

- Bubblesort has complexity $O(n^2)$, therefore very inefficient.
- If an algorithm uses comparison keys to decide the correct order then the theoretical lower bound on complexity is $O(n \log n)$. From wiki:

Algorithms : Sorting

```c
#define NUMS 6

void bubble_sort(int b[], int s);

int main(void)
{
    int a[] = {3, 4, 1, 2, 9, 0};
    bubble_sort(a, NUMS);
    for(int i=0; i<NUMS; i++){
        printf("%i ", a[i]);
    }
    printf("\n");
    return 0;
}

void bubble_sort(int b[], int s)
{
    bool changes;
    do{
        changes = false;
        for(int i=0; i<s-1; i++){
            if(b[i] > b[i+1]){
                SWAP(b[i], b[i+1]);
                changes = true;
            }
        }
    }while(changes);
}
```

Execution :

0 1 2 3 4 9

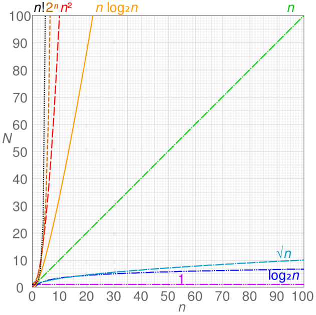
- Bubblesort has complexity $O(n^2)$, therefore very inefficient.
- If an algorithm uses comparison keys to decide the correct order then the theoretical lower bound on complexity is $O(n \log n)$. From wiki:

```c
#define NUMS 6

void bubble_sort(int b[], int s);

int main(void)
{
    int a[] = {3, 4, 1, 2, 9, 0};
    bubble_sort(a, NUMS);
    for(int i=0; i<NUMS; i++){
        printf("%i ", a[i]);
    }
    printf("\n");
    return 0;
}

void bubble_sort(int b[], int s)
{
    bool changes;
    do{
        changes = false;
        for(int i=0; i<s-1; i++){
            if(b[i] > b[i+1]){
                SWAP(b[i], b[i+1]);
                changes = true;
            }
        }
    }while(changes);
}
```

Execution :

0 1 2 3 4 9

- Bubblesort has complexity $O(n^2)$, therefore very inefficient.
- If an algorithm uses comparison keys to decide the correct order then the theoretical lower bound on complexity is $O(n \log n)$. From wiki:



Algorithms : Merge Sort

- Transposition (Bubblesort)

Algorithms : Merge Sort

- Transposition (Bubblesort)
- Insertion Sort (Lab Work)

Algorithms : Merge Sort

- Transposition (Bubblesort)
- Insertion Sort (Lab Work)
- Priority Queue (Selection sort, Heap sort)

Algorithms : Merge Sort

- Transposition (Bubblesort)
- Insertion Sort (Lab Work)
- Priority Queue (Selection sort, Heap sort)
- Divide & Conquer (Merge & Quick sorts)

Algorithms : Merge Sort

- Transposition (Bubblesort)
- Insertion Sort (Lab Work)
- Priority Queue (Selection sort, Heap sort)
- Divide & Conquer (Merge & Quick sorts)
- Address Calculation (Proxmap)

Algorithms : Merge Sort

- Transposition (Bubblesort)
- Insertion Sort (Lab Work)
- Priority Queue (Selection sort, Heap sort)
- Divide & Conquer (Merge & Quick sorts)
- Address Calculation (Proxmap)

Algorithms : Merge Sort

- Transposition (Bubblesort)
- Insertion Sort (Lab Work)
- Priority Queue (Selection sort, Heap sort)
- Divide & Conquer (Merge & Quick sorts)
- Address Calculation (Proxmap)

- Merge sort is divide-and-conquer in that you divide the array into two halves, mergesort each half and then merge the two halves into order.

Algorithms : Merge Sort

- Transposition (Bubblesort)
- Insertion Sort (Lab Work)
- Priority Queue (Selection sort, Heap sort)
- Divide & Conquer (Merge & Quick sorts)
- Address Calculation (Proxmap)

- Merge sort is divide-and-conquer in that you divide the array into two halves, mergesort each half and then merge the two halves into order.

Algorithms : Merge Sort

- Transposition (Bubblesort)
- Insertion Sort (Lab Work)
- Priority Queue (Selection sort, Heap sort)
- Divide & Conquer (Merge & Quick sorts)
- Address Calculation (Proxmap)

- Merge sort is divide-and-conquer in that you divide the array into two halves, mergesort each half and then merge the two halves into order.

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

void mergesort(int *src, int *spare, int l, int r);
void merge(int *src, int *spare, int l, int m, int r);

#define NUM 5000

int main(void)
{
    int a[NUM];
    int spare[NUM];

    for(int i=0; i<NUM; i++){
        a[i] = rand()%100;
    }

    mergesort(a, spare, 0, NUM-1);

    for(int i=0; i<NUM; i++){
        printf("%4d => %d\n", i, a[i]);
    }

    return 0;
}
```

Merge Sort II

```
void mergesort(int *src, int *spare, int l, int r)
{
    int m = (l+r)/2;
    if(l != r){
        mergesort(src, spare, l, m);
        mergesort(src, spare, m+1, r);
        merge(src, spare, l, m, r);
    }
}

void merge(int *src, int *spare, int l, int m, int r)
{
    int s1 = l;
    int s2 = m+1;
    int d = l;

    do{
        if(src[s1] < src[s2]){
            spare[d++] = src[s1++];
        }
        else{
            spare[d++] = src[s2++];
        }
    }while((s1 <= m) && (s2 <= r));

    if(s1 > m){
        memcpy(&spare[d], &src[s2], sizeof(spare[0])*(r-s2+1));
    }
    else{
        memcpy(&spare[d], &src[s1], sizeof(spare[0])*(m-s1+1));
    }
    memcpy(&src[l], &spare[l], (r-l+1)*sizeof(spare[0]));
}
```

Merge Sort II

```c
void mergesort(int *src, int *spare, int l, int r)
{
    int m = (l+r)/2;
    if(l != r){
        mergesort(src, spare, l, m);
        mergesort(src, spare, m+1, r);
        merge(src, spare, l, m, r);
    }
}

void merge(int *src, int *spare, int l, int m, int r)
{
    int s1 = l;
    int s2 = m+1;
    int d = l;

    do{
        if(src[s1] < src[s2]){
            spare[d++] = src[s1++];
        }
        else{
            spare[d++] = src[s2++];
        }
    }while((s1 <= m) && (s2 <= r));

    if(s1 > m){
        memcpy(&spare[d], &src[s2], sizeof(spare[0])*(r-s2+1));
    }
    else{
        memcpy(&spare[d], &src[s1], sizeof(spare[0])*(m-s1+1));
    }
    memcpy(&src[l], &spare[l], (r-l+1)*sizeof(spare[0]));
}
```

- Quicksort is also divide-and-conquer.

Merge Sort II

```c
void mergesort(int *src, int *spare, int l, int r)
{
    int m = (l+r)/2;
    if(l != r){
        mergesort(src, spare, l, m);
        mergesort(src, spare, m+1, r);
        merge(src, spare, l, m, r);
    }
}

void merge(int *src, int *spare, int l, int m, int r)
{
    int s1 = l;
    int s2 = m+1;
    int d = l;

    do{
        if(src[s1] < src[s2]){
            spare[d++] = src[s1++];
        }
        else{
            spare[d++] = src[s2++];
        }
    }while((s1 <= m) && (s2 <= r));

    if(s1 > m){
        memcpy(&spare[d], &src[s2], sizeof(spare[0])*(r-s2+1));
    }
    else{
        memcpy(&spare[d], &src[s1], sizeof(spare[0])*(m-s1+1));
    }
    memcpy(&src[l], &spare[l], (r-l+1)*sizeof(spare[0]));
}
```

- Quicksort is also divide-and-conquer.
- Choose some value in the array as the *pivot* key.

Merge Sort II

```
void mergesort(int *src, int *spare, int l, int r)
{
    int m = (l+r)/2;
    if(l != r){
        mergesort(src, spare, l, m);
        mergesort(src, spare, m+1, r);
        merge(src, spare, l, m, r);
    }
}

void merge(int *src, int *spare, int l, int m, int r)
{
    int s1 = l;
    int s2 = m+1;
    int d = l;

    do{
        if(src[s1] < src[s2]){
            spare[d++] = src[s1++];
        }
        else{
            spare[d++] = src[s2++];
        }
    }while((s1 <= m) && (s2 <= r));

    if(s1 > m){
        memcpy(&spare[d], &src[s2], sizeof(spare[0])*(r-s2+1));
    }
    else{
        memcpy(&spare[d], &src[s1], sizeof(spare[0])*(m-s1+1));
    }
    memcpy(&src[l], &spare[l], (r-l+1)*sizeof(spare[0]));
}
```

- Quicksort is also divide-and-conquer.
- Choose some value in the array as the *pivot* key.
- This key is used to divide the array into two partitions. The left partition contains keys $\leq$ pivot key, the right partition contains keys $>$ pivot.

Merge Sort II

```c
void mergesort(int *src, int *spare, int l, int r)
{
    int m = (l+r)/2;
    if(l != r){
        mergesort(src, spare, l, m);
        mergesort(src, spare, m+1, r);
        merge(src, spare, l, m, r);
    }
}

void merge(int *src, int *spare, int l, int m, int r)
{
    int s1 = l;
    int s2 = m+1;
    int d = l;

    do{
        if(src[s1] < src[s2]){
            spare[d++] = src[s1++];
        }
        else{
            spare[d++] = src[s2++];
        }
    }while((s1 <= m) && (s2 <= r));

    if(s1 > m){
        memcpy(&spare[d], &src[s2], sizeof(spare[0])*(r-s2+1));
    }
    else{
        memcpy(&spare[d], &src[s1], sizeof(spare[0])*(m-s1+1));
    }
    memcpy(&src[l], &spare[l], (r-l+1)*sizeof(spare[0]));
}
```

- Quicksort is also divide-and-conquer.
- Choose some value in the array as the *pivot* key.
- This key is used to divide the array into two partitions. The left partition contains keys $\leq$ pivot key, the right partition contains keys $>$ pivot.
- Once again, the sort is then applied recursively.

Algorithms : Quicksort

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

int partition(int *a, int l, int r);
void quicksort(int *a, int l, int r);

#define NUM 100000

int main(void)
{
    int a[NUM];

    for(int i=0; i<NUM; i++){
        a[i] = rand()%100;
    }
    quicksort(a, 0, NUM-1);

    return 0;
}

void quicksort(int *a, int l, int r)
{
    int pivpoint = partition(a, l, r);
    if(l < pivpoint){
        quicksort(a, l, pivpoint-1);
    }
    if(r > pivpoint){
        quicksort(a, pivpoint+1, r);
    }
}
```

Algorithms : Quicksort

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

int partition(int *a, int l, int r);
void quicksort(int *a, int l, int r);

#define NUM 100000

int main(void)
{
    int a[NUM];

    for(int i=0; i<NUM; i++){
        a[i] = rand()%100;
    }
    quicksort(a, 0, NUM-1);

    return 0;
}

void quicksort(int *a, int l, int r)
{
    int pivpoint = partition(a, l, r);
    if(l < pivpoint){
        quicksort(a, l, pivpoint-1);
    }
    if(r > pivpoint){
        quicksort(a, pivpoint+1, r);
    }
}
```

```c
int partition(int *a, int l, int r)
{
    int piv = a[l];
    while(l<r){
        /* Right -> Left Scan */
        while(piv < a[r] && l<r) r--;
        if(r!=l){
            a[l] = a[r];
            l++;
        }
        /* Left -> Right Scan */
        while(piv > a[l] && l<r) l++;
        if(r!=l){
            a[r] = a[l];
            r--;
        }
    }
    a[r] = piv;
    return r;
}
```

qsort()

- Theoretically both methods have a complexity $O(n \log n)$

qsort()

- Theoretically both methods have a complexity $O(n \log n)$
- Quicksort is preferred because it requires less memory and is generally faster.

qsort()

- Theoretically both methods have a complexity $O(n \log n)$
- Quicksort is preferred because it requires less memory and is generally faster.
- Quicksort can go badly wrong if the pivot key chosen is either the maximum or minimum value in the array.

qsort()

- Theoretically both methods have a complexity $O(n \log n)$
- Quicksort is preferred because it requires less memory and is generally faster.
- Quicksort can go badly wrong if the pivot key chosen is either the maximum or minimum value in the array.
- Quicksort is so loved by programmers that a library version of it exists in ANSI C.

qsort()

- Theoretically both methods have a complexity $O(n \log n)$
- Quicksort is preferred because it requires less memory and is generally faster.
- Quicksort can go badly wrong if the pivot key chosen is either the maximum or minimum value in the array.
- Quicksort is so loved by programmers that a library version of it exists in ANSI C.
- If you need an off-the-shelf sort, this is often a good option.

qsort()

- Theoretically both methods have a complexity $O(n \log n)$
- Quicksort is preferred because it requires less memory and is generally faster.
- Quicksort can go badly wrong if the pivot key chosen is either the maximum or minimum value in the array.
- Quicksort is so loved by programmers that a library version of it exists in ANSI C.
- If you need an off-the-shelf sort, this is often a good option.

qsort()

- Theoretically both methods have a complexity $O(n \log n)$
- Quicksort is preferred because it requires less memory and is generally faster.
- Quicksort can go badly wrong if the pivot key chosen is either the maximum or minimum value in the array.
- Quicksort is so loved by programmers that a library version of it exists in ANSI C.
- If you need an off-the-shelf sort, this is often a good option.

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

int intcompare(const void *a, const void *b);

int main(void)
{
   int a[10];

   for(int i=0; i<10; i++){
      a[i] = 9 - i;
   }

   qsort(a, 10, sizeof(int), intcompare);

   for (int i=0; i<10; i++){
      printf(" %d",a[i]);
   }
   printf("\n");
   return 0;

}

int intcompare(const void *a, const void *b)
{
    const int *ia = (const int *)a;
    const int *ib = (const int *)b;
    return *ia  - *ib;
}
```

Algorithms : The Radix Sort

- The radix sort is also know as the bin sort, a name derived from its origin as a technique used on (now obsolete) card sorters.

Algorithms : The Radix Sort

- The radix sort is also know as the bin sort, a name derived from its origin as a technique used on (now obsolete) card sorters.
- For integer data, repeated passes of radix sort focus on the right digit (the units), then the second digit (the tens) and so on.

Algorithms : The Radix Sort

- The radix sort is also know as the bin sort, a name derived from its origin as a technique used on (now obsolete) card sorters.
- For integer data, repeated passes of radix sort focus on the right digit (the units), then the second digit (the tens) and so on.
- Strings could be sorted in a similar manner.

Algorithms : The Radix Sort

- The radix sort is also know as the bin sort, a name derived from its origin as a technique used on (now obsolete) card sorters.
- For integer data, repeated passes of radix sort focus on the right digit (the units), then the second digit (the tens) and so on.
- Strings could be sorted in a similar manner.

Algorithms : The Radix Sort

- The radix sort is also know as the bin sort, a name derived from its origin as a technique used on (now obsolete) card sorters.
- For integer data, repeated passes of radix sort focus on the right digit (the units), then the second digit (the tens) and so on.
- Strings could be sorted in a similar manner.

459 254 472 534 649 239 432 654 477

0
1
2 472 432
3
4 254 534 654
5
6
7 477
8
9 459 649 239

Read out the new list:
472 432 254 534 654 477 459 649 239

Radix Sort II

472 432 254 534 654 477 459 649 239

0
1
2
3 432 534 239
4 649
5 254 654 459
6
7 472 477
8
9

432 534 239 649 254 654 459 472 477

Radix Sort II

472 432 254 534 654 477 459 649 239

0
1
2
3 432 534 239
4 649
5 254 654 459
6
7 472 477
8
9

432 534 239 649 254 654 459 472 477

432 534 239 649 254 654 459 472 477

0
1
2 239 254
3
4 432 459 472 477
5 534
6 649 654
7
8
9

239 254 432 459 472 477 534 649 654

Radix Sort Discussion and gprof

- This has a theoretical complexity of $O(n)$.

Radix Sort Discussion and gprof

- This has a theoretical complexity of $O(n)$.
- It is difficult to write an all-purpose radix sort - you need a different one for doubles, integers, strings etc.

Radix Sort Discussion and gprof

- This has a theoretical complexity of $O(n)$.
- It is difficult to write an all-purpose radix sort - you need a different one for doubles, integers, strings etc.
- $O(n)$ simply means that the number of operations can be bounded by $k.n$, for some constant k.

Radix Sort Discussion and gprof

- This has a theoretical complexity of $O(n)$.
- It is difficult to write an all-purpose radix sort - you need a different one for doubles, integers, strings etc.
- $O(n)$ simply means that the number of operations can be bounded by $k.n$, for some constant k.
- With the radix sort, k is often very large.

Radix Sort Discussion and gprof

- This has a theoretical complexity of $O(n)$.
- It is difficult to write an all-purpose radix sort - you need a different one for doubles, integers, strings etc.
- $O(n)$ simply means that the number of operations can be bounded by $k.n$, for some constant k.
- With the radix sort, k is often very large.
- For many lists this may be less efficient than more traditional $O(n \log n)$ algorithms.

Radix Sort Discussion and gprof

- This has a theoretical complexity of $O(n)$.
- It is difficult to write an all-purpose radix sort - you need a different one for doubles, integers, strings etc.
- $O(n)$ simply means that the number of operations can be bounded by $k.n$, for some constant k.
- With the radix sort, k is often very large.
- For many lists this may be less efficient than more traditional $O(n \log n)$ algorithms.

- Sometimes you'll want to profile your code.

Radix Sort Discussion and gprof

- This has a theoretical complexity of $O(n)$.
- It is difficult to write an all-purpose radix sort - you need a different one for doubles, integers, strings etc.
- $O(n)$ simply means that the number of operations can be bounded by $k.n$, for some constant k.
- With the radix sort, k is often very large.
- For many lists this may be less efficient than more traditional $O(n \log n)$ algorithms.
- Sometimes you'll want to profile your code.
- Compile with the -pg flag.

Radix Sort Discussion and gprof

- This has a theoretical complexity of $O(n)$.
- It is difficult to write an all-purpose radix sort - you need a different one for doubles, integers, strings etc.
- $O(n)$ simply means that the number of operations can be bounded by $k.n$, for some constant k.
- With the radix sort, k is often very large.
- For many lists this may be less efficient than more traditional $O(n \log n)$ algorithms.

- Sometimes you'll want to profile your code.
- Compile with the `-pg` flag.
- Executing your code produces a `gmon.out` file.

Radix Sort Discussion and gprof

- This has a theoretical complexity of $O(n)$.
- It is difficult to write an all-purpose radix sort - you need a different one for doubles, integers, strings etc.
- $O(n)$ simply means that the number of operations can be bounded by $k.n$, for some constant k.
- With the radix sort, k is often very large.
- For many lists this may be less efficient than more traditional $O(n \log n)$ algorithms.

- Sometimes you'll want to profile your code.
- Compile with the -pg flag.
- Executing your code produces a `gmon.out` file.
- Now: `gprof ./executable gmon.out` shows the function-call profile of your code.

Table of Contents

Algorithm : Huffman Compression

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

Algorithm : Huffman Compression

- 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.

Algorithm : Huffman Compression

- 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.

Algorithm : Huffman Compression

- 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":

Algorithm : Huffman Compression

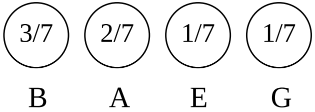
- Often we wish to compress data, to reduce storage requirements, or to speed transmission.
- Text is particularly suited to compression since using one byte per character is wasteful - some letters occur much more frequently.
- Need to give frequently occurring letters short codes, typically a few bits.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

- To encode the string "BABBAGE":

Algorithm : Huffman Compression

- 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":



- Keep a list of characters, ordered by their frequency

Huffman Compression II

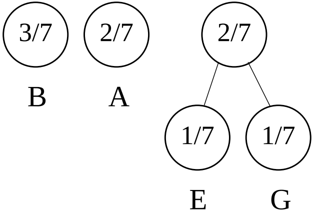
- Use the two least frequent to form a sub-tree, and re-order (sort) the nodes :

Huffman Compression II

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

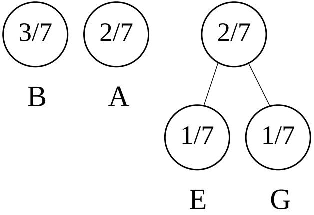
Huffman Compression II

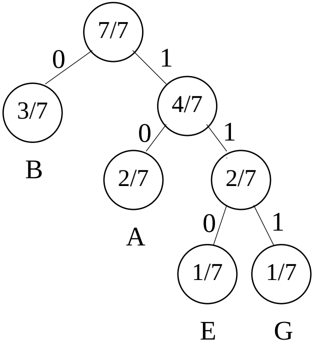
- Use the two least frequent to form a sub-tree, and re-order (sort) the nodes :



Huffman Compression II

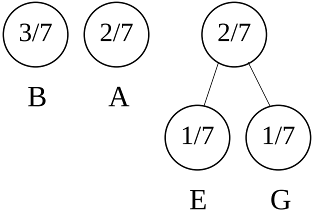
- Use the two least frequent to form a sub-tree, and re-order (sort) the nodes :

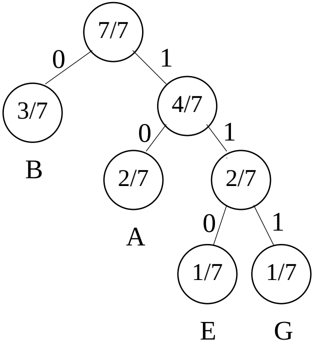




Huffman Compression II

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

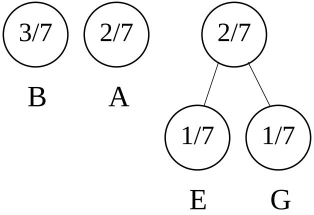


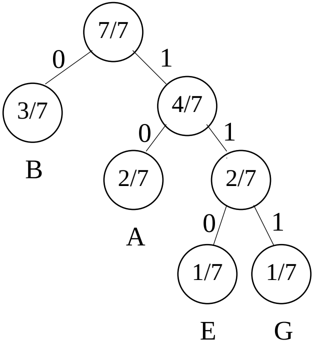


- A = 10, B = 0, E = 110, G = 111

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.

Algorithm : Rabin-Karp String Searching

- The task of searching for a string amongst a large amount of text is commonly required in word-processors, but more interestingly in massive Biological Databases e.g. searching for amino acids in protein sequences.

Algorithm : Rabin-Karp String Searching

- The task of searching for a string amongst a large amount of text is commonly required in word-processors, but more interestingly in massive Biological Databases e.g. searching for amino acids in protein sequences.
- How difficult can it be ? Don't you just do a character by character brute-force search ?

```
Master String : AAAAAAAAAAAAAH
Substring     : AAAAAAH
Substring     :  AAAAAAH
Substring     :   AAAAAAH
```

Algorithm : Rabin-Karp String Searching

- The task of searching for a string amongst a large amount of text is commonly required in word-processors, but more interestingly in massive Biological Databases e.g. searching for amino acids in protein sequences.
- How difficult can it be ? Don't you just do a character by character brute-force search ?

```
Master String : AAAAAAAAAAAAH
Substring     : AAAAAAH
Substring     :  AAAAAAH
Substring     :   AAAAAAH
```

Algorithm : Rabin-Karp String Searching

- The task of searching for a string amongst a large amount of text is commonly required in word-processors, but more interestingly in massive Biological Databases e.g. searching for amino acids in protein sequences.
- How difficult can it be ? Don't you just do a character by character brute-force search ?

  Master String : AAAAAAAAAAAAAH
  Substring : AAAAAAH
  Substring : AAAAAAH
  Substring : AAAAAAH

- If the master string has m characters, and the search string has n characters then this search has complexity: $O(mn)$

Algorithm : Rabin-Karp String Searching

- The task of searching for a string amongst a large amount of text is commonly required in word-processors, but more interestingly in massive Biological Databases e.g. searching for amino acids in protein sequences.
- How difficult can it be ? Don't you just do a character by character brute-force search ?

```
Master String : AAAAAAAAAAAAAH
Substring     : AAAAAAH
Substring     :  AAAAAAH
Substring     :   AAAAAAH
```

- If the master string has m characters, and the search string has n characters then this search has complexity: $O(mn)$
- Recall that to compute a hash function on a word we did something like:

$$h("NEILL") =$$

$$(13 \times 26^4 + 4 \times 26^3 + 8 \times 26^2 + 11 \times 26 + 11)\%P$$

where P is a big prime number.

Algorithm : Rabin-Karp String Searching

- The task of searching for a string amongst a large amount of text is commonly required in word-processors, but more interestingly in massive Biological Databases e.g. searching for amino acids in protein sequences.
- How difficult can it be ? Don't you just do a character by character brute-force search ?

  Master String : AAAAAAAAAAAAAH
  Substring : AAAAAAH
  Substring : AAAAAAH
  Substring : AAAAAAH

- If the master string has m characters, and the search string has n characters then this search has complexity: $O(mn)$
- Recall that to compute a hash function on a word we did something like:

$$h("NEILL") =$$

$$(13 \times 26^4 + 4 \times 26^3 + 8 \times 26^2 + 11 \times 26 + 11)\%P$$

  where P is a big prime number.
- This can be expanded by Horner's method to:

$$((((((13 \times 26) + 4) \times 26) + 8) \times 26) + 11) \times 26 + 11)\%P$$

Rabin-Karp II

- For a large search string, overflow can occur. We therefore move the *mod* operation inside the brackets:

$$((((((13 \times 26) + 4)\%P \times 26) + 8)\%P \times 26) + 11)\%P \times 26 + 11)\%P$$

Rabin-Karp II

- For a large search string, overflow can occur. We therefore move the *mod* operation inside the brackets:

  $((((((13 \times 26) + 4)\%P \times 26) + 8)\%P \times 26) + 11)\%P \times 26 + 11)\%P$

- We can compute a hash number for the search string, and for the initial part of the master string.

Rabin-Karp II

- For a large search string, overflow can occur. We therefore move the *mod* operation inside the brackets:

  $$((((((13 \times 26) + 4)\%P \times 26) + 8)\%P \times 26) + 11)\%P \times 26 + 11)\%P$$

- We can compute a hash number for the search string, and for the initial part of the master string.
- When we compute the hash number for the next part of the master, most of the computation is common, we just need to take out the effect of the first letter and add in the effect of the new one.

Rabin-Karp II

- For a large search string, overflow can occur. We therefore move the *mod* operation inside the brackets:

  $$((((((13 \times 26) + 4)\%P \times 26) + 8)\%P \times 26) + 11)\%P \times 26 + 11)\%P$$

- We can compute a hash number for the search string, and for the initial part of the master string.
- When we compute the hash number for the next part of the master, most of the computation is common, we just need to take out the effect of the first letter and add in the effect of the new one.
- One small calculation each time we move one place right in the master.

Rabin-Karp II

- For a large search string, overflow can occur. We therefore move the *mod* operation inside the brackets:

  $((((((13 \times 26) + 4)\%P \times 26) + 8)\%P \times 26) + 11)\%P \times 26 + 11)\%P$

- We can compute a hash number for the search string, and for the initial part of the master string.
- When we compute the hash number for the next part of the master, most of the computation is common, we just need to take out the effect of the first letter and add in the effect of the new one.
- One small calculation each time we move one place right in the master.
- Complexity $O(m + n)$ roughly, but need to check that two identical hash numbers really has identified two identical strings.

Rabin-Karp II

- For a large search string, overflow can occur. We therefore move the *mod* operation inside the brackets:

  $((((((13 \times 26) + 4)\%P \times 26) + 8)\%P \times 26) + 11)\%P \times 26 + 11)\%P$

- We can compute a hash number for the search string, and for the initial part of the master string.
- When we compute the hash number for the next part of the master, most of the computation is common, we just need to take out the effect of the first letter and add in the effect of the new one.
- One small calculation each time we move one place right in the master.
- Complexity $O(m + n)$ roughly, but need to check that two identical hash numbers really has identified two identical strings.

Rabin-Karp II

- For a large search string, overflow can occur. We therefore move the *mod* operation inside the brackets:

  $$((((((13\times 26)+4)\%P\times 26)+8)\%P\times 26)+11)\%P\times 26+11)\%P$$

- We can compute a hash number for the search string, and for the initial part of the master string.
- When we compute the hash number for the next part of the master, most of the computation is common, we just need to take out the effect of the first letter and add in the effect of the new one.
- One small calculation each time we move one place right in the master.
- Complexity $O(m+n)$ roughly, but need to check that two identical hash numbers really has identified two identical strings.

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

#define Q 33554393
#define D 26
#define index(C) (C-'A')

int rk(char *p, char *a);

int main(void)
{
   assert(rk("STING",
          "A STRING EXAMPLE CONSISTING OF ...")==22);
   return 0;
}

int rk(char *p, char *a)
{
   int i, dM = 1, h1=0, h2=0;
   int m = strlen(p);
   int n = strlen(a);
   for(i=1; i<m; i++) dM = (D*dM)%Q;
   for(i=0; i<m; i++){
      h1 = (h1*D+index(p[i]))%Q;
      h2 = (h2*D+index(a[i]))%Q;
   }
   // h1 = search string hash, h2 = master string hash
   for(i=0; h1!=h2; i++){
      h2 = (h2+D*Q-index(a[i])*dM) % Q;
      h2 = (h2*D+index(a[i+m])) % Q;
      if(i>n-m) return n;
   }
   return i;
}
```

Algorithm : Boyer-Moore String Searching

The Boyer-Moore algorithm uses (in part) an array flagging which characters form part of the search string and an array telling us how far to slide right if that character appears in the master and causes a mismatch.

Execution :

```
A STRING SEARCHING EXAMPLE CONSISTING OF ...
     |     |
STING      |
     STING
          STING
```

Algorithm : Boyer-Moore String Searching

The Boyer-Moore algorithm uses (in part) an array flagging which characters form part of the search string and an array telling us how far to slide right if that character appears in the master and causes a mismatch.

Execution :

```
A STRING SEARCHING EXAMPLE CONSISTING OF ...
     |     |
STING      |
     STING
          STING
```

- With a right-to-left walk through the search string we see that the G and the R mismatch on the first comparison.

Algorithm : Boyer-Moore String Searching

The Boyer-Moore algorithm uses (in part) an array flagging which characters form part of the search string and an array telling us how far to slide right if that character appears in the master and causes a mismatch.

Execution :

```
A STRING SEARCHING EXAMPLE CONSISTING OF ...
     |     |
STING      |
     STING
          STING
```

- With a right-to-left walk through the search string we see that the G and the R mismatch on the first comparison.
- Since R doesn't appear in the search string, we can take 5 steps to the right.

Algorithm : Boyer-Moore String Searching

The Boyer-Moore algorithm uses (in part) an array flagging which characters form part of the search string and an array telling us how far to slide right if that character appears in the master and causes a mismatch.

Execution :

```
A STRING SEARCHING EXAMPLE CONSISTING OF ...
     |     |
STING      |
     STING
         STING
```

- With a right-to-left walk through the search string we see that the G and the R mismatch on the first comparison.
- Since R doesn't appear in the search string, we can take 5 steps to the right.
- The next comparison is between the G and the S. We can slide the search string right until it matches the S in the master.

Boyer-Moore II

Execution :

```
A STRING SEARCHING EXAMPLE CONSISTING OF ...
             |    |    |    |    |
         STING    |    |    |    |
              STING    |    |    |
                   STING    |    |
                        STING    |
                             STING
```

- Now the C doesn't appear in the master and once again we can slide a full 5 places to the right.

Boyer-Moore II

Execution :

```
A STRING SEARCHING EXAMPLE CONSISTING OF ...
         |    |    |    |    |
     STING    |    |    |    |
          STING    |    |    |
               STING    |    |
                    STING    |
                         STING
```

- Now the C doesn't appear in the master and once again we can slide a full 5 places to the right.
- After 3 more full slides right we arrive at the T in CONSISTING.

Boyer-Moore II

Execution :

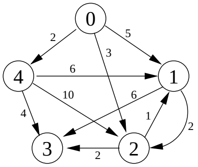
```
A STRING SEARCHING EXAMPLE CONSISTING OF ...
          |     |     |     |     |
       STING    |     |     |     |
             STING    |     |     |
                   STING    |     |
                         STING    |
                               STING
```

- Now the C doesn't appear in the master and once again we can slide a full 5 places to the right.
- After 3 more full slides right we arrive at the T in CONSISTING.
- We align the T's, and have found our match using 7 compares (plus 5 to verify the match).

Table of Contents

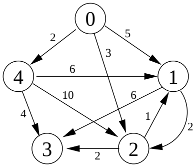
ADTs : Graphs

- 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.



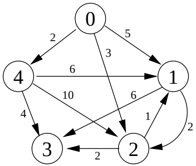
ADTs : Graphs

- 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.

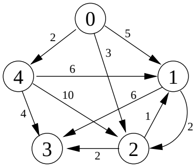
- A graph, G, consists of a set of vertices (nodes), V, together with a set of edges (links), E, each of which connects two vertices.



- 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.

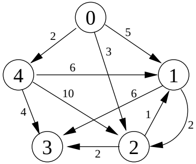
ADTs : Graphs

- 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.

- 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)

```
#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

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

specific.h

```c
#define GRAPHTYPE "Realloc"

#define INITSIZE 8
#define SCALEFACTOR 2

#define TMPSTR 1000

#define NO_VERT -1

typedef unsigned int edge;

struct graph {
   edge** adjMat;
   char** labels;
   /* Actual number of verts */
   int size;
   /* Max verts before realloc() */
   int capacity;
};
typedef struct graph graph;
```

2D Realloc II

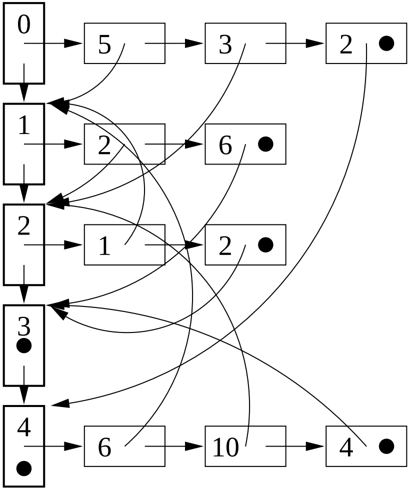
```c
graph* graph_init(void)
{
   graph* g = (graph*) ncalloc(sizeof(graph), 1);
   int h = INITSIZE;
   int w = h;
   g->capacity = h;
   g->adjMat = (edge**) n2dcalloc(h, w, sizeof(edge));
   g->labels = (char**) n2dcalloc(h, MAXLABEL+1, sizeof(char));
   for(int j=0; j<h; j++){
      for(int i=0; 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

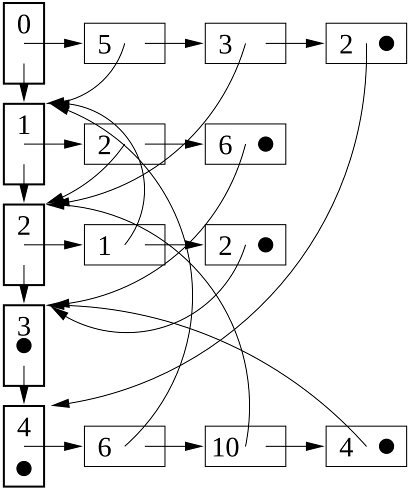
```c
graph* graph_init(void)
{
   graph* g = (graph*) ncalloc(sizeof(graph), 1);
   int h = INITSIZE;
   int w = h;
   g->capacity = h;
   g->adjMat = (edge**) n2dcalloc(h, w, sizeof(edge));
   g->labels = (char**) n2dcalloc(h, MAXLABEL+1, sizeof(char));
   for(int j=0; j<h; j++){
      for(int i=0; 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

```c
#define GRAPHTYPE "Linked"

#define INITSIZE 8
#define SCALEFACTOR 2

#define TMPSTR 1000

#define NO_VERT -1

typedef unsigned int edge;

struct vertex {
   char* label;
   struct vertex* nextv;
   void* firste;
   int num;
};
typedef struct vertex vertex;

struct edge {
   edge weight;
   vertex* v;
   struct edge* nexte;
};
typedef struct edge edgel;

struct graph {
   vertex* firstv;
   vertex* endv;
   int size;
};
typedef struct graph graph;
```

Linked II

```c
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;
}
```

Linked II

```c
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;
}
```

```c
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);
}
```

Sidebar : $P = NP$?

- The P versus NP problem is a major unsolved problem in theoretical computer science.

Sidebar : $P = NP$?

- The P versus NP problem is a major unsolved problem in theoretical computer science.
- It asks whether every problem whose solution can be quickly verified can also be quickly solved.

Sidebar : $P = NP$?

- The P versus NP problem is a major unsolved problem in theoretical computer science.
- It asks whether every problem whose solution can be quickly verified can also be quickly solved.
- e.g.the Subset Sum problem — for a set of numbers and a target sum, determine if there exists a subset of numbers that adds up to the target sum.

Sidebar : $P = NP$?

- The P versus NP problem is a major unsolved problem in theoretical computer science.
- It asks whether every problem whose solution can be quickly verified can also be quickly solved.
- e.g.the Subset Sum problem — for a set of numbers and a target sum, determine if there exists a subset of numbers that adds up to the target sum.
- Easy to verify a solution: For $\{3, 4, 5, 6, 7\}$ and a target of 9, a valid subset is $\{4, 5\}$. This is P.

Sidebar : $P = NP$?

- The P versus NP problem is a major unsolved problem in theoretical computer science.
- It asks whether every problem whose solution can be quickly verified can also be quickly solved.
- e.g.the Subset Sum problem — for a set of numbers and a target sum, determine if there exists a subset of numbers that adds up to the target sum.
- Easy to verify a solution: For $\{3, 4, 5, 6, 7\}$ and a target of 9, a valid subset is $\{4, 5\}$. This is P.
- But computing it takes $O(2^n)$ time (non-polynomial) NP.

Sidebar : $P = NP$?

- The P versus NP problem is a major unsolved problem in theoretical computer science.
- It asks whether every problem whose solution can be quickly verified can also be quickly solved.
- e.g.the Subset Sum problem — for a set of numbers and a target sum, determine if there exists a subset of numbers that adds up to the target sum.
- Easy to verify a solution: For $\{3, 4, 5, 6, 7\}$ and a target of 9, a valid subset is $\{4, 5\}$. This is P.
- But computing it takes $O(2^n)$ time (non-polynomial) NP.

Sidebar : $P = NP$?

- The P versus NP problem is a major unsolved problem in theoretical computer science.
- It asks whether every problem whose solution can be quickly verified can also be quickly solved.
- e.g.the Subset Sum problem — for a set of numbers and a target sum, determine if there exists a subset of numbers that adds up to the target sum.
- Easy to verify a solution: For $\{3, 4, 5, 6, 7\}$ and a target of 9, a valid subset is $\{4, 5\}$. This is P.
- But computing it takes $O(2^n)$ time (non-polynomial) NP.
- But maybe NP algorithms are ‘hard’ simply because we haven’t found a better solution?

Sidebar : $P = NP$?

- The P versus NP problem is a major unsolved problem in theoretical computer science.
- It asks whether every problem whose solution can be quickly verified can also be quickly solved.
- e.g.the Subset Sum problem — for a set of numbers and a target sum, determine if there exists a subset of numbers that adds up to the target sum.
- Easy to verify a solution: For $\{3, 4, 5, 6, 7\}$ and a target of 9, a valid subset is $\{4, 5\}$. This is P.
- But computing it takes $O(2^n)$ time (non-polynomial) NP.
- But maybe NP algorithms are 'hard' simply because we haven't found a better solution?
- It is thought that $P \neq NP$, meaning there are problems that can't be solved in polynomial time, but for which the answer could be verified in polynomial time.

Sidebar : $P = NP$?

- The P versus NP problem is a major unsolved problem in theoretical computer science.
- It asks whether every problem whose solution can be quickly verified can also be quickly solved.
- e.g.the Subset Sum problem — for a set of numbers and a target sum, determine if there exists a subset of numbers that adds up to the target sum.
- Easy to verify a solution: For $\{3, 4, 5, 6, 7\}$ and a target of 9, a valid subset is $\{4, 5\}$. This is P.
- But computing it takes $O(2^n)$ time (non-polynomial) NP.
- But maybe NP algorithms are 'hard' simply because we haven't found a better solution?
- It is thought that $P \neq NP$, meaning there are problems that can't be solved in polynomial time, but for which the answer could be verified in polynomial time.
- A proof either way would have profound implications for mathematics, cryptography, AI etc.

Algorithms : TSP on Graphs

- Imagine planning a delivery route around a graph, starting from a particular vertex.

Algorithms : TSP on Graphs

- Imagine planning a delivery route around a graph, starting from a particular vertex.
- What's the least cost by which you can visit every vertex without ever returning to one ?

Algorithms : TSP on Graphs

- Imagine planning a delivery route around a graph, starting from a particular vertex.
- What's the least cost by which you can visit every vertex without ever returning to one ?
- Finding the optimal path (to reduce travelling time) is an `NP-hard` problem - read up on Computational Complexity Theory to understand this better.

Algorithms : TSP on Graphs

- Imagine planning a delivery route around a graph, starting from a particular vertex.
- What's the least cost by which you can visit every vertex without ever returning to one ?
- Finding the optimal path (to reduce travelling time) is an `NP-hard` problem - read up on Computational Complexity Theory to understand this better.
- For small graphs you could do this exhaustively, but for very large graphs this combinatorial approach becomes untenable.

Algorithms : TSP on Graphs

- Imagine planning a delivery route around a graph, starting from a particular vertex.
- What's the least cost by which you can visit every vertex without ever returning to one ?
- Finding the optimal path (to reduce travelling time) is an `NP-hard` problem - read up on Computational Complexity Theory to understand this better.
- For small graphs you could do this exhaustively, but for very large graphs this combinatorial approach becomes untenable.
- One 'greedy' approach is to simply go to your closest unvisited neighbour each time.

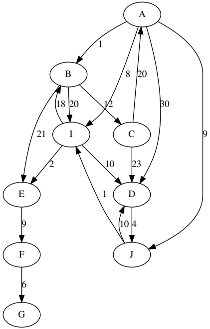
Algorithms : TSP on Graphs

- Imagine planning a delivery route around a graph, starting from a particular vertex.
- What's the least cost by which you can visit every vertex without ever returning to one ?
- Finding the optimal path (to reduce travelling time) is an `NP-hard` problem - read up on Computational Complexity Theory to understand this better.
- For small graphs you could do this exhaustively, but for very large graphs this combinatorial approach becomes untenable.
- One 'greedy' approach is to simply go to your closest unvisited neighbour each time.

Algorithms : TSP on Graphs

- Imagine planning a delivery route around a graph, starting from a particular vertex.
- What's the least cost by which you can visit every vertex without ever returning to one ?
- Finding the optimal path (to reduce travelling time) is an `NP-hard` problem - read up on Computational Complexity Theory to understand this better.
- For small graphs you could do this exhaustively, but for very large graphs this combinatorial approach becomes untenable.
- One 'greedy' approach is to simply go to your closest unvisited neighbour each time.
- Typically gives results within 25% of the optimal solution, but sometimes give a worst-case solution ...

- Imagine planning a delivery route around a graph, starting from a particular vertex.
- What's the least cost by which you can visit every vertex without ever returning to one ?
- Finding the optimal path (to reduce travelling time) is an `NP-hard` problem - read up on Computational Complexity Theory to understand this better.
- For small graphs you could do this exhaustively, but for very large graphs this combinatorial approach becomes untenable.
- One 'greedy' approach is to simply go to your closest unvisited neighbour each time.
- Typically gives results within 25% of the optimal solution, but sometimes give a worst-case solution . . .
- `A -> B -> C -> D -> J -> I -> E -> F -> G`



TSP II

```
edge graph_salesman(graph* g, int from, char* str)
{
   bool* unvis;
   int curr, ncurr, nvs;
   edge cst, bcst, e;

   nvs = graph_numVerts(g);
   if((g==NULL) || (from >= nvs) || (str==NULL)){
      return INF;
   }
   unvis = (bool*)ncalloc(nvs, sizeof(bool));
   for(int v=0; v<nvs; v++){
      unvis[v] = true;
   }
   curr = from;
   bcst = 0;
   strcpy(str, graph_getLabel(g, from));
   do{
      unvis[curr] = false;
      cst = INF;
      ncurr = NO_VERT;
      /* Look at neighbours of curr */
```

TSP II

```c
edge graph_salesman(graph* g, int from, char* str)
{
   bool* unvis;
   int curr, ncurr, nvs;
   edge cst, bcst, e;

   nvs = graph_numVerts(g);
   if((g==NULL) || (from >= nvs) || (str==NULL)){
      return INF;
   }
   unvis = (bool*)ncalloc(nvs, sizeof(bool));
   for(int v=0; v<nvs; v++){
      unvis[v] = true;
   }
   curr = from;
   bcst = 0;
   strcpy(str, graph_getLabel(g, from));
   do{
      unvis[curr] = false;
      cst = INF;
      ncurr = NO_VERT;
      /* Look at neighbours of curr */
      for(int v=0; v<nvs; v++){
         e = graph_getEdgeWeight(g, curr, v);
         if((v!=curr) && unvis[v] && (e!=INF)){
            if(e < cst){
               cst = e;
               ncurr = v;
            }
         }
      }
      /* Add in cost to go to closest */
      if(cst < INF){
         bcst += cst;
         curr = ncurr;
         strcat(str, " ");
         strcat(str, graph_getLabel(g, ncurr));
      }
   }while((cst < INF) && (curr != NO_VERT));
   free(unvis);
   return bcst;
}
```

Algorithms : Dijkstra on Graphs

- It's often important to find the shortest path through a graph from one vertex to another.

Algorithms : Dijkstra on Graphs

- It's often important to find the shortest path through a graph from one vertex to another.
- One way of doing this is the greedy algorithm due to Dijkstra discovered 1956.

Algorithms : Dijkstra on Graphs

- It's often important to find the shortest path through a graph from one vertex to another.
- One way of doing this is the greedy algorithm due to Dijkstra discovered 1956.
- Picks the unvisited vertex with the lowest distance, & calculate the distance through it to each unvisited neighbor, updating the neighbour's distance if smaller.

Algorithms : Dijkstra on Graphs

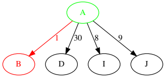
- It's often important to find the shortest path through a graph from one vertex to another.
- One way of doing this is the greedy algorithm due to Dijkstra discovered 1956.
- Picks the unvisited vertex with the lowest distance, & calculate the distance through it to each unvisited neighbor, updating the neighbour's distance if smaller.
- Mark visited when done with neighbors.

Algorithms : Dijkstra on Graphs

- It's often important to find the shortest path through a graph from one vertex to another.
- One way of doing this is the greedy algorithm due to Dijkstra discovered 1956.
- Picks the unvisited vertex with the lowest distance, & calculate the distance through it to each unvisited neighbor, updating the neighbour's distance if smaller.
- Mark visited when done with neighbors.

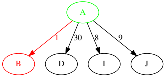
Algorithms : Dijkstra on Graphs

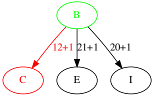
- It's often important to find the shortest path through a graph from one vertex to another.
- One way of doing this is the greedy algorithm due to Dijkstra discovered 1956.
- Picks the unvisited vertex with the lowest distance, & calculate the distance through it to each unvisited neighbor, updating the neighbour's distance if smaller.
- Mark visited when done with neighbors.



Algorithms : Dijkstra on Graphs

- It's often important to find the shortest path through a graph from one vertex to another.
- One way of doing this is the greedy algorithm due to Dijkstra discovered 1956.
- Picks the unvisited vertex with the lowest distance, & calculate the distance through it to each unvisited neighbor, updating the neighbour's distance if smaller.
- Mark visited when done with neighbors.





- It's often important to find the shortest path through a graph from one vertex to another.
- One way of doing this is the greedy algorithm due to Dijkstra discovered 1956.
- Picks the unvisited vertex with the lowest distance, & calculate the distance through it to each unvisited neighbor, updating the neighbour's distance if smaller.
- Mark visited when done with neighbors.