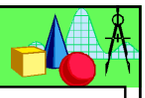


COMP2611 – Data Structures

Dr. John Charlery

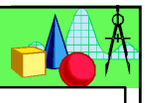
3rd Floor CMP Building

Tel.: 417-4363
Email: **john.charlery@cavehill.uwi.edu**

Office Hours: **Wed 1 – 3 pm**
Thu 4 – 6 pm

1

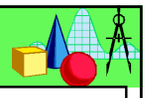
COMP2611 Timetable



Time	Mon	Tues	Wed	Thurs	Fri
9 - 10				LAB P03 CSL-2	
10 - 11					
11 - 12					
12 - 1					
1 - 2			Office Hours		
2 - 3					
3 - 4		LAB P01 CSL-1			
4 - 5				Office Hours	
5 - 6					
6 - 7			LECTURE		
7 - 8		LAB P02 CSL-2			
8 - 9					

2

COMP2611 – Data Structures



Assessment : **Course Work : 40%**
Final Exam : 60%

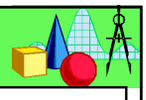
Dates:

Project 1	**–**	**Sun. 02 Oct, 2022**	**(10%)**
Mid-Term Exam	**–**	**Wed. 19 Oct, 2022**	**(15%)**
Project 2	**–**	**Sun. 28 Nov, 2022**	**(15%)**

All Projects' Submissions are due at midnight on the due dates.

3

COMP2611 – Data Structures



Text Books:

Larry R. Nyhoff : ADTs, Data Structures and Problem Solving with C++

Julian Smart : Cross-Platform GUI Programming with wxWidgets

Recommended Text

Mark Weiss : ***Data Structures and Algorithms using C++ .***

Bruno R. Preiss : ***Data Structures and Algorithms With Object Oriented-Design Patterns in C++.***

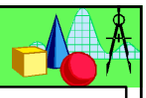
Deitel & Deitel : ***C++: How to Program***

Main & Savitch : ***Data Structures and Other Objects Using C++***

Cay Horstmann : ***Computing Concepts With C++ Essentials***

4

COMP2611 – Data Structures



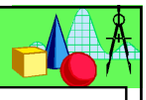
Operating Language	:	C++ GNU-GCC Compiler
Operating System	:	Linux – OpenSUSE (Leap)
GUI API	:	wxWidgets

WARNING:

You are free to privately use other flavours of Linux to build your code, but for examination of your work, make sure your code is compliant with the version of Linux in **CSL-1/2**.

5

COMP2611 – Data Structures



The Honour Rule:

All evidence of copying or plagiarizing will result in an automatic award of **ZERO** mark.

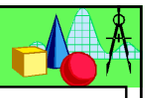
VERY, VERY, VERY IMPORTANT:

Your projects' code MUST compile and run!

Non-compiling code will result in an automatic award of **ZERO** mark for the project.

6

COMP2611 – Data Structures



How to pass this course?

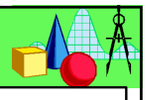
Very Simple:

1. Attend the lectures – be punctual and participate.
2. Attempt to do the lab exercises and assignments **on your own.**
3. Participate in the 10 minutes tutorial-like sessions at the start of each lectures.
4. Start working on your assignments as soon as you get them and continue fine-tuning your code throughout the time you have. Target your completion date to be at least a few days before the due date.
5. Attend the lab sessions; get help where you are falling short and make an effort to participate.
6. Make use of my office hours if you need more help from me.

Remember you cannot **READ** your way through this course – you have to practice!

7

For you personally…



You need Linux!

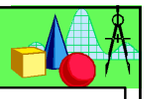
On your own Windows machine:

1. **Download and install VirtualBox (or VMware)**
2. **Install Linux OpenSUSE in VirtualBox**
3. **Add in the following packages in OpenSUSE:**
   - **gcc-c++**
   - **kate**
   - **wxWidgets-3.2-devel**
   - **make**
   - **automake**
   - **gimp**

These packages are for your own machines!

They are already on the machines in the Labs.

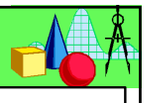
8



Quick Review

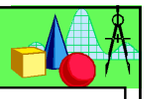
of

Functions, Pointers & Classes



FUNCTIONS

Function Documentation



Format:

Function Name:

Input:

Output:

Purpose or Method:

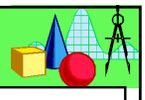
EXAMPLE:

```
/*
    Function Name: Square.
    Input: An integer whose square is required.
    Output: The squared value of the input integer as an integer.
    Purpose: To compute the square of an input value.
*/
```

3

3

Types of Functions



1. Functions with Empty parameter lists

- **void** or leave parameter list empty
- Indicates function takes no arguments

e.g. Function **print** takes no arguments and returns no value:

```
void print();
void print( void );
```

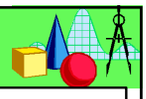
e.g. Function **getAge** takes no arguments and returns an int value:

```
int getAge();
```

4

4

Sample Program

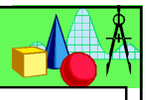


```
1   // Program : Program3.cpp
2   // Functions that take no arguments.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   void function1();          // function prototype
9   void function2( void );    // function prototype
10
11  int main()
12   {
13      function1();           // call function1 with no arguments
14      function2();           // call function2 with no arguments
15
16      return 0;              // indicates successful termination
17
18   } // end main
19
```

5

5

Sample Program (cont'd)



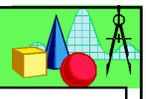
```
20  // function1 uses an empty parameter list to specify that
21  // the function receives no arguments
22  void function1()
23     {
24          cout << "function1 takes no arguments" << endl;
25      } // end function1
26
27
28  // function2 uses a void parameter list to specify that
29  // the function receives no arguments
30  void function2( void )
31     {
32         cout << "function2 also takes no arguments" << endl;
33      } // end function2
34
```

```
function1 takes no arguments
function2 also takes no arguments
```

6

6

Types of Functions



2. Functions containing parameter lists

- ❖ Contains data types and variables in the parameter list.
- ❖ Indicates function requires data to be provided in order to perform it's task.

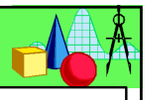
e.g. Function **square** and **sum** take one and two arguments respectively and return an integer value.

```
int square ( int x);
int sum( int x,  int y );
```

7

7

Sample Program

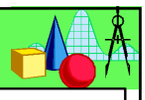


```
1   // Functions Prototypes
2   int square ( int );
3   int sum ( int,  int );
4
5   int square ( int x)
6       {
7           return (x * x);
8       }
9
10  int sum ( int x,  int y)
11      {
12          return (x + y);
13      }
```

8

8

Types of Functions: **Inline Functions**



3. Inline functions

- Keyword **inline** before function
- Asks the compiler to copy code into program instead of making function call
  - Reduce function-call overhead
  - Compiler can ignore **inline**
- Good for small, often-used functions

Example

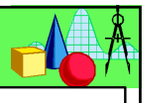
```
inline double cube( const double val )
        {
            return ( val * val * val );
        }
```

const *tells the compiler that function cannot modify the variable* ***val***

9

9

Sample Program

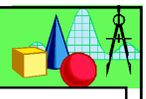


```
1   // Program : Program4.cpp
2   // Using an inline function to calculate the volume of a cube.
3
4   #include <iostream>
5
6   using std::cout;
7   using std::cin;
8   using std::endl;
9
10  // Definition of inline function cube. Definition of function appears before function
11  // is called,  so a function prototype is not required.
12  // First line of function definition acts as the prototype.
13
14  inline double cube( const double side )
15      {
16          return side * side * side;    // calculates cube and returns the value
17      } // end function cube
18
19
```

10

10

Sample Program (cont'd)



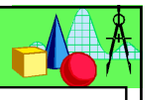
```
20  int main()
21      {
22          cout << "Enter the side length of your cube: ";
23
24          double sideValue;
25
26          cin >> sideValue;
27
28          // calculate cube of sideValue and display result
29          cout  << "Volume of cube with side "  << sideValue  << " is "
30                << cube( sideValue ) << endl;
31
32           return 0;  // indicates successful termination
33
34      } // end main
```

Note

Compiler replaces:
Call to: cube(sideValue)
with: (sideValue * sideValue * sideValue)

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```


Sample Program 2

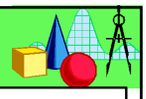


```
1   // Program : Program4.cpp
2   // Using an inline function to calculate the volume of a cube.
3
4   #include <iostream>
5
6   using std::cout;
7   using std::cin;
8   using std::endl;
9
10  // Definition of Functions Prototype Section.
11   int square ( int );
12   int sum ( int, int );
13   double cube( const double side ) { return (side * side * side); }
18   // etc. etc.
19
```

Second method of defining Inline Functions.

Function defined in the Function Prototype section of the program.

Types of Functions



4. Functions with Default Arguments

- Function call with missing OR omitted parameters
  - If not enough parameters, rightmost go to their defaults
  - Default values can be constants, global variables, or function calls
- Set defaults in function prototype!!!

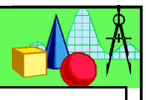
Example

```
int myFunction( int  x = 1,  int  y = 2,  int  z = 3 );

myFunction(3);          //x = 3, y and z get defaults (rightmost)
myFunction(3, 5);       //x = 3, y = 5 and z gets default
myFunction(3, 5, 8);    //x = 3, y = 5 and z = 8
```

13

13

Sample Program



```
1   // Program : Program5.cpp
2   // Using default arguments.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   // function prototype that specifies default arguments
9   int boxVolume( int length = 1, int width = 1,  int height = 1 );
10
11  int main()
12  {
13    // no arguments--use default values for all dimensions
14    cout << "The default box volume is: " << boxVolume();
15
16    // specify length; default width and height
17    cout << "\n\nThe volume of a box with length 10,\n"
18       << "width 1 and height 1 is: " << boxVolume( 10 );
19
20    // specify length and width; default height
21    cout << "\n\nThe volume of a box with length 10,\n"
22       << "width 5 and height 1 is: " << boxVolume( 10, 5 );
23
```

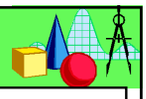
Set defaults in function prototype.

Function calls with some parameters missing – the rightmost parameters get their defaults.

14

14

Sample Program



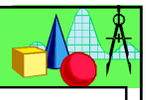
```
24    // specify all arguments
25    cout << "\n\nThe volume of a box with length 10,\n"
26       << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
27       << endl;
28
29    return 0;  // indicates successful termination
30
31 } // end main
32
33 // function boxVolume calculates the volume of a box
34 int boxVolume( int length,  int width,  int height )
35    {
36       return length * width * height;
37    } // end function boxVolume
38
```

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

15

15

Recursion



5. Recursive functions

- ❖ Functions that call themselves
- ❖ Can only solve a base case

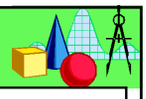
If not base case

- ➢ Break problem into smaller problem(s).
- ➢ Launch new copy of function to work on the smaller problem (recursive call/recursive step).
  - Slowly converges towards base case.
  - Function makes call to itself inside the return statement.
- ➢ Eventually base case gets solved.
  - Answer works way back up - solves entire problem.

16

16

Recursion



Example: factorial

$$n! = n * (n - 1) * (n - 2) * \ldots * 1$$

Recursive relationship (n! = n * (n – 1)!)

$5! = 5 * 4!$

$4! = 4 * 3!\ldots$

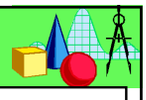
Base case (1! = 0! = 1)

5! = 5 * 4!
= 5 * (4 * 3!)
= 5 * (4 * (3 * 2!))
= 5* (4 * (3 * (2 *1!)))
= 5 * (4* (3 * (2 * (1))))
= 5 * (4* (3 * (2 * 1))))
= 5 * (4* (3 * (2)))
= 5 * (4* (3 * 2))
= 5 * (4* 6)
= 5 * 24
= 120

17

17

Sample Program



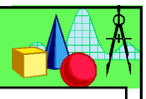
```
1   // Program : Program8.cpp
2   // Recursive factorial function.
3   #include <iostream>
4         using namespace  std;
5   #include <iomanip>
6
7   unsigned long factorial( unsigned long ); // function prototype
8
9   int main()
10  {
11        // Loop 10 times. During each iteration, calculate factorial ( i ) and display result.
12        for ( int i = 0; i <= 10; i++ )
13              cout << setw( 2 ) << i << "!  = "    << factorial ( i ) << endl;
14
15        return 0;  // indicates successful termination
16
17  } // end main
```

Data type unsigned long can hold an integer from 0 to 4 billion.

18

18

Sample Program



```
18
19  // recursive definition of function factorial
20  unsigned long factorial( unsigned long number )
21  {
22     // base case
23     if ( number  <=  1 )
24        return 1;
25
26     // recursive step
27     else
28        return number * factorial( number - 1 );
29
30  } // end function factorial
```

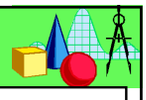
The base case occurs when we have 0! or 1!. All other cases must be split up (recursive step).

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

19

19

Example Using Recursion: **Fibonacci Series**



Fibonacci series: 0, 1, 1, 2, 3, 5, 8...

- Each number sum of two previous ones

Example of a recursive formula:

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

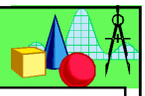
C++ code for Fibonacci function

```
long fibonacci( long n )
  {
    if ( n == 0 || n == 1 )   // base case
        return n;
    else
        return fibonacci( n - 1 ) + fibonacci( n – 2 );
  }
```

20

20

Recursion vs. Iteration



For Repetition

- Iteration: explicit loop
- Recursion: repeated function calls

For Termination

- Iteration: loop condition fails
- Recursion: base case recognized

Both can have infinite loops

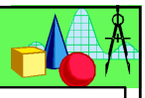
Desire?

Seek balance between performance (iteration) and good software engineering (recursion)

21

21

Function Overloading



Function overloading

- Functions with same name and different parameters
- Should perform similar tasks

  e.g, function to square **int**s and function to square **float**s

```
int square( int x) {return x * x;}
float square(float x) { return x * x; }
```

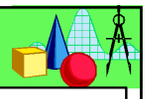
Overloaded functions distinguished by signature

- Based on name and parameter types (order matters).
- Name mangling
  - Encodes function identifier with parameters.
- Type-safe linkage
  - Ensures proper overloaded function is called.

22

22

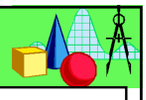
Sample Program



```
1   // Program : Program6.cpp - Using overloaded functions.
2   #include <iostream>
3
4   using std::cout;
5   using std::endl;
6
7   // function square for int values
8   int square( int x )
9     {
10       cout << "Called square with int argument: " << x << endl;
11       return x * x;
12    } // end int version of function square
13
14  // function square for double values
15  double square( double y )
16    {
17       cout << "Called square with double argument: " << y << endl;
18       return y * y;
19    } // end double version of function square
```

Overloaded functions have the same name, but the different parameters distinguish them.

Sample Program

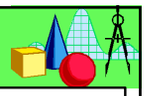


```
20  int main()
21  {
22    int intResult = square( 7 );          // calls int version
23    double doubleResult = square( 7.5 ); // calls double version
24
25    cout << "\nThe square of integer 7 is " << intResult
26         << "\nThe square of double 7.5 is " << doubleResult
27         << endl;
28
29    return 0;  // indicates successful termination
30
31  } // end main
```

The proper function is called based upon the argument (`int` or `double`).

```
Called square with int argument: 7
Called square with double argument: 7.5

The square of integer 7 is 49
The square of double 7.5 is 56.25
```

Function Templates



Compact way to make overloaded functions

Generate separate function for different data types

Format

- Begins with keyword **template.**
- Formal type parameters in brackets **< >.**

  Every type parameter preceded by **typename** or **class** (synonyms).

  Placeholders for built-in types (i.e., **int**) or user-defined types.

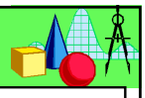
  Specifies arguments types, return types, declare variables.

- Function definition like normal, except formal types are used.

25

25

Function Templates



Example

```
template < class Tdata >    // or template< typename Tdata >
Tdata square( Tdata value1 )
      {
          return value1 * value1;
      }
```

- **Tdata** is a formal type, used as the parameter data type.

  Above function returns variable of same type as parameter.

- In function call, **Tdata** is replaced by actual data type.

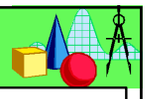
  If **int**, all **Tdata**'s become **int**s, etc.

  e.g. `int x;`
  `int y = square(x);`

26

26

Sample Program



```
1   // Program : Program7.cpp - Using a function template.
3   #include <iostream>
4
5   Using namespace std;
6
9   // definition of function template maximum
10  template < class Tdata >  // or template < typename Tdata >
11  Tdata maximum( Tdata value1, Tdata value2, Tdata value3 )
12     {
13        Tdata max = value1;
14
15        if ( value2 > max )
16              max = value2;
18        if ( value3 > max )
19              max = value3;
21        return max;
22
23  } // end function template maximum
24
```

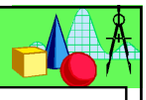
Formal type parameter **`Tdata`** placeholder for type of data to be tested by **`maximum`**.

`maximum` expects all parameters to be of the same type.

27

27

Sample Program



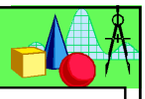
```
25  int main()
26  {
27     // demonstrate maximum with int values
28     int int1, int2, int3;
29
30     cout << "Input three integer values: ";
31     cin >> int1 >> int2 >> int3;
32
33     // invoke int version of maximum
34     cout << "The maximum integer value is: "  <<  maximum( int1, int2, int3 );
35
36     // demonstrate maximum with double values
37     double double1, double2, double3;
38
39     cout << "\n\nInput three double values: ";
40     cin >> double1 >> double2 >> double3;
41
42     // invoke double version of maximum
43     cout << "The maximum double value is: "  << maximum( double1, double2, double3 );
```

`maximum` called with various data types.

28

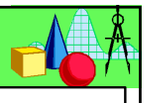
28

Sample Program



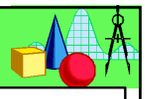
```
44    // demonstrate maximum with char values
45    char char1, char2, char3;
46
47    cout << "\n\nInput three characters: ";
48    cin >> char1 >> char2 >> char3;
49
50    // invoke char version of maximum
51    cout << "The maximum character value is: "
52         << maximum( char1, char2, char3 )
53         << endl;
54
55    return 0;  // indicates successful termination
56
57 } // end main
```

```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```

Header Files

Header Files



Standard Header Files

- Part of the compiler system
  - e.g. **iostream.h, math.h, conio.h, etc**.

User-Defined Header Files

- Contain user defined classes and functions.
- Must use **#include** statement to incorporate in code.
- Use macro guards for all header files.

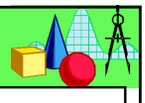
E.g.

```
#include <math.h>          // Include system header file
#include "myfile.h"        // Include user-defined header file
```

31

31

Header Files



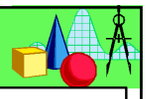
Macro Guards Format:

```
#ifndef  <label name>
#define  <label name>

….functions

#endif
```

32

32

Header Files - Example



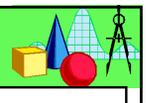
```
#ifndef  MYFILE_H
#define MYFILE_H
// function to compute the factorial of an integer
unsigned long factorial( int number )
      {
            if ( number  <=  1 )  return 1;
            else  return number * factorial( number - 1 );
      } // end function factorial

// function square for int values
int square( int x )
      {
            return x * x;
      } // end int version of function square

// function square for double values
double square( double y )
     {
            return y * y;
     } // end double version of function square
#endif
```

Macro Guards

33

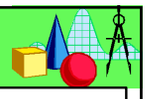
33

COMP2611 – Data Structures



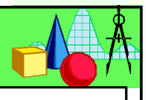
POINTERS

34



Pointers

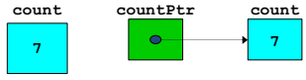
- Powerful, but sometimes perceived as difficult to master.
- Simulate pass-by-reference,
- Can be used to create and manipulate dynamic data structures.
- Close relationship with arrays and strings.



Pointer Variable Declarations and Initialization

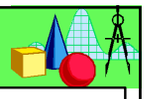
Pointer variables

- Contain memory addresses as values
- Normally, variable contains specific value (direct reference)
- Pointers contain address of variable that has specific value (indirect reference)



Indirection

- Referencing value through pointer



Pointer Variable Declarations and Initialization

Pointer declarations

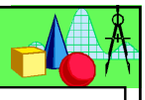
* indicates variable is pointer

```
int *myPtr;
```

declares pointer to **int**, pointer of type **int ***

Multiple pointers require multiple asterisks

```
int *myPtr1, *myPtr2;
```

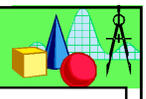



Pointer Variable Declarations and Initialization

- Can declare pointers to any data type.

Pointer initialization

- Initialized to **0**, **NULL**, or a RAM address.
  0 or **NULL** points to nothing.



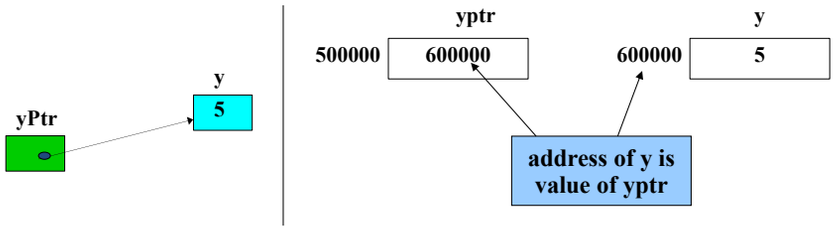
Pointer Operators

& (address operator)

Returns memory address of its operand

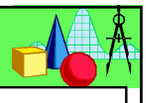
Example

```
int y = 5;
int *yPtr;
yPtr = &y;     // yPtr gets address of y -- yPtr "points to" y
```



39

39



Pointer Operators

```
int y = 5;
int *yPtr;
yPtr = &y;
```

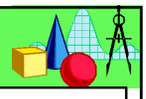
***** (indirection/dereferencing operator)

- Returns value in the RAM location its pointer operand points to.
- ***yPtr** returns **5** (because **yPtr** points to **y** and **y** has **5** in it).
- dereferenced pointer as lvalue
  - ***yptr = 9;** *// assigns 9 to y*

***** and **&** are inverses of each other

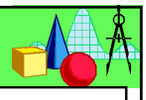
40

40



```
1   // Program : Program01.cpp
2   // Using the & and * operators.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   int main()
9   {
10    int a;        // a is an integer
11    int *aPtr;  // aPtr is a pointer to an integer
12
13    a = 7;
14    aPtr = &a;  // aPtr assigned address of a
15
16    cout << "The address of a is " << &a
17       << "\nThe value of aPtr is " << aPtr;
18
19    cout << "\n\nThe value of a is " << a
20       << "\nThe value of *aPtr is " << *aPtr;
21
22    cout << "\n\nShowing that * and & are inverses of "
23       << "each other.\n&*aPtr = " << &*aPtr
24       << "\n*&aPtr = " << *&aPtr << endl;
25
26    return 0;  // indicates successful termination
27
28  } // end main
```

*** and & are inverses of each other**

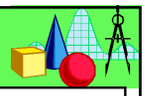


The address of a is 0012FED4
The value of aPtr is 0012FED4

The value of a is 7
The value of *aPtr is 7

Showing that * and & are inverses of each other.
&*aPtr = 0012FED4
***&aPtr = 0012FED4**

*** and & are inverses; same result when both applied to aPtr**



Calling Functions by Reference

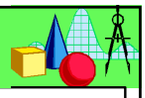
3 ways to pass arguments to function

1. Pass-by-value.
2. Pass-by-reference with reference arguments.
3. Pass-by-reference with pointer arguments.

return statement can return **only** one value from function.

Arguments passed to function using reference arguments

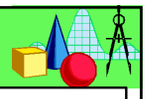
- Can modify original values of arguments.
- More than one value can be "returned".



Calling Functions by Reference

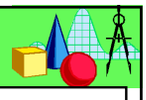
Pass-by-reference with pointer arguments

- Simulate pass-by-reference
  - Use pointers and indirection operator.
- Pass address of argument using **&** operator.
- Arrays **not** passed with **&** because array name is a pointer.
- * operator used as alias/nickname for variable inside of function.



```
1  // Program : Program02.cpp
2  // Cube a function using pass-by-value.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int cubeByValue( int );   // prototype
9
10 int main()
11 {
12   int number = 5;
13
14   cout << "The original value of number is " << number;
15
16   // pass number by value to cubeByValue
17   number = cubeByValue( number );
18
19   cout << "\nThe new value of number is " << number << endl;
20
21   return 0;  // indicates successful termination
22
23 } // end main
24
```

Pass number by value; result returned by cubeByValue

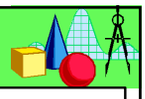


```
25 // calculate and return cube of integer argument
26 int cubeByValue( int n )
27 {
28    return n * n * n; // cube local variable n and return result
29
30 } // end function cubeByValue
```

cubeByValue receives parameter passed-by-value and calls it "n"

Cubes the value received and returns the product

The original value of number is 5
The new value of number is 125

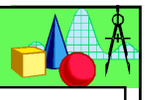


```
1   // Program : Program03.cpp
2   // Cube a function using pass-by-reference with a pointer argument.
3
4   #include <iostream>
5
6   using std::cout;
7   using std::endl;
8
9   void cubeByReference( int * );   // function prototype
10
11  int main()
12  {
13    int number = 5;
14
15    cout << "The original value of number is " << number;
16
17    // pass address of number to cubeByReference
18    cubeByReference( &number );
19
20    cout << "\nThe new value of number is " << number << endl;
21
22    return 0;  // indicates successful termination
23
24  } // end main
25
```

Prototype indicates parameter is pointer to int

Apply address operator & to pass address of number to cubeByReference

cubeByReference modified variable number



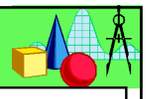
```
26  // calculate cube of *nPtr; modifies variable number in main
27  void cubeByReference( int *nPtr )
28  {
29    *nPtr = (*nPtr) * (*nPtr) * (*nPtr);    // cube *nPtr
30
31  } // end function cubeByReference
```

cubeByReference receives address of int variable,

i.e., pointer to an int

Modify and access int variable using indirection operator *

The original value of number is 5
The new value of number is 125

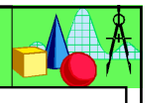


Using const with Pointers

const pointers

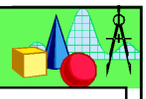
- Always point to same memory location.
- Default for array name.
- Must be initialized when declared.

See Review File for more details on using const with pointers.



Classes and Data Abstraction

Implementing an Abstract Data Type with a class



Classes

- Model objects
  - Attributes (data members)
  - Behaviors (member functions)
- Defined using keyword **class**
- Member functions
  - Methods
  - Invoked in response to messages

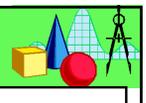
Member access specifiers:

public:	Accessible wherever object of class in scope
private:	Accessible only to member functions of class
protected:	*(hybrid – will be discussed later)*

51

51

Implementing an Abstract Data Type with a class



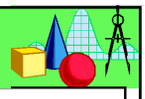
Constructor function

- Special member function
- Initializes data members
- Same name as class
- Called when object instantiated
- Can have several constructors (Function overloading)
- No return type

52

52

Implementing an Abstract Data Type with a class

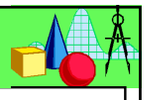


```
1   class Time
2       {
3           private:
4               int hour;          // 0 - 23 (24-hour clock format)
5               int minute;        // 0 - 59
6               int second;        // 0 - 59
7
8           public:
9               Time();                              // constructor
10              void setTime( int, int, int );       // set hour, minute, second
11              void printUniversal();               // print universal-time format
12              void printStandard();                // print standard-time format
13      };   // end class Time
```

53

53

Implementing an Abstract Data Type with a class



Objects of class

- After class definition
  - Class name becomes a new type specifier
  - C++ extensible language can then provide:
    - Object, array, pointer and reference declarations

Example:

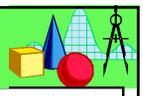
Class name becomes new type specifier.

```
Time sunset;                    // object of type Time
Time arrayOfTimes[ 5 ];         // array of Time objects
Time *pointerToTime;            // pointer to a Time object
Time &dinnerTime = sunset;      // reference to a Time object
```

54

54

Implementing an Abstract Data Type with a class



For Member functions defined outside of class

Binary scope resolution operator (**::**)

- ❑ "Ties" member name to class name
- ❑ Uniquely identify functions of particular class
- ❑ Different classes can have member functions with same name

Format for defining member functions

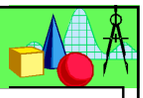
```
ReturnType  ClassName::MemberFunctionName( )
    {
        …
    }
```

Format does not change whether function is **public** or **private**

55

55

Implementing an Abstract Data Type with a class



For Member functions defined inside class

- ❑ Do not need scope resolution operator, class name
- ❑ Compiler creates **inline function**

NOTE: Apart from inline functions, member functions should NOT be defined within the class declaration!!!

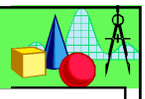
NOTE:

- Outside of class declaration, inline functions are defined explicitly with the keyword **inline.**

56

56

Implementing an Abstract Data Type with a class

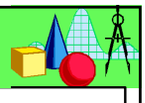


```
1   // Program : Program9.cpp
2   // Time class.
3   #include <iostream>
4   #include <iomanip>
5
6   using std::cout;
7   using std::endl;
8   using std::setfill;
9   using std::setw;
10
11  // Time abstract data type (ADT) definition
12  class Time
13     {
14        private:
15                    int hour;              // 0 - 23 (24-hour clock format)
16                    int minute;            // 0 - 59
17                    int second;            // 0 - 59
18           public:
19                    Time();                          // constructor
20                    void setTime( int, int, int );   // set hour, minute, second
21                    void printUniversal();           // print universal-time format
22                    void printStandard();            // print standard-time format
23     }; // end class Time
```

Define class **`Time`**.

57

Implementing an Abstract Data Type with a class



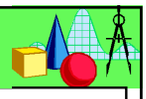
```
24
25  // Time constructor initializes each data member to zero.
26  Time::Time()
27        {
28           hour = minute = second = 0;
29        } // end Time constructor
30
31  // set new Time value using universal time, perform validity
32  // checks on the data values and set invalid values to zero
33  void Time::setTime( int hh, int mm, int ss )
34        {
35           hour   = ( hh >= 0 && hh < 24 ) ? hh : 0;
36           minute = ( mm >= 0 && mm < 60 ) ? mm : 0;
37           second = ( ss >= 0 && ss < 60 ) ? ss : 0;
38        } // end function setTime
```

Constructor initializes **`private`** data members to **`0`**.

`public` member function checks parameter values for validity before setting **`private`** data members.

58

Implementing an Abstract Data Type with a class

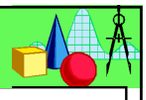


```
39  // print Time in universal format
40  void Time::printUniversal()
41          {
42                  cout  << setfill( '0' ) << setw( 2 ) << hour << ":"
43                        << setw( 2 ) << minute << ":" << setw( 2 )
44                        << second;
45                        } // end function printUniversal
46
47  // print Time in standard format
48  void Time::printStandard()
49          {
50                   cout  << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":" << setfill( '0' )
51                         << setw( 2 ) << minute  << ":" << setw( 2 ) << second
52                         << ( hour < 12 ? " AM" : " PM" );
53          } // end function printStandard
54
55  int main()
60          {
61                  Time obj;  // instantiate object t of class Time
62
```

No arguments (implicitly "know" purpose is to print data members); member function calls more concise.

Declare variable **`obj`** to be object of class **`Time`**.

Implementing an Abstract Data Type with a class



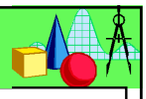
```
63          // output Time object t's initial values
64          cout << "The initial universal time is ";
65          obj.printUniversal();    // 00:00:00
66
67          cout << "\nThe initial standard time is ";
68          obj.printStandard();     // 12:00:00 AM
69
70          obj.setTime( 13, 27, 6 );              // change time
71
72          // output Time object t's new values
73          cout << "\n\nUniversal time after setTime is ";
74          obj.printUniversal();    // 13:27:06
75
76          cout << "\nStandard time after setTime is ";
77          obj.printStandard();     // 1:27:06 PM
78
79          obj.setTime( 99, 99, 99 );             // attempt invalid settings
80
81          // output t's values after specifying invalid values
82          cout << "\n\nAfter attempting invalid settings:"
83               << "\nUniversal time: ";
84          obj.printUniversal();    // 00:00:00
85
```

Invoke **`public`** member functions to print time.

Set data members using **`public`** member function.

Attempt to set data members to invalid values using **`public`** member function.

Implementing an Abstract Data Type with a class



```
86          cout << "\nStandard time: ";
87          obj.printStandard();    // 12:00:00 AM
88          cout << endl;
89
90          return 0;
91
92    } // end main
```

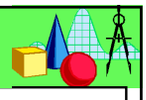
```
The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

Data members set to **0** after attempting invalid settings.

61

61

Implementing an Abstract Data Type with a class



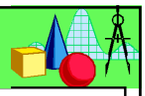
Destructors

- Same name as class
  - Preceded with tilde (~)
- No arguments
- Cannot be overloaded
- Performs "**termination housekeeping**"

62

62

Implementing an Abstract Data Type with a class



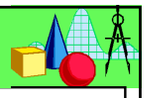
Advantages of using classes

- Simplify programming
- Interfaces
  - Hide implementation
- Software reuse
  - Composition (aggregation)
  - Class objects included as members of other classes
- Inheritance
  - New classes derived from old

63

63

Implementing an Abstract Data Type with a class



Class scope

Data members, member functions

Within class scope

Class members

- Immediately accessible by all member functions.
- Referenced by name.

Outside class scope

- Referenced through handles

  Object name, reference to object, pointer to object.

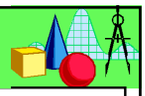
File scope

Non-member functions

64

64

Implementing an Abstract Data Type with a class



Function scope

- Variables declared in member function
- Only known to function
- Variables with same name as class-scope variables

  Class-scope variable are "**hidden**"

  May be access with scope resolution operator (**::**)

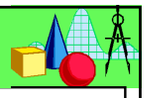
  ClassName::classVariableName

- Variables only known to function they are defined in
- Variables are destroyed after function completion

65

65

Implementing an Abstract Data Type with a class



Operators to access class members

- Identical to those for **struct**s
- Dot member selection operator (**.**)

  - Object
  - Reference to object

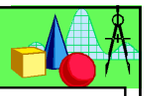
- Arrow member selection operator (**->**)

  - Pointers

66

66

Separating Interface from Implementation

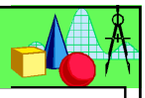


Header files

- Class definitions and function prototypes.
- File extension **.h**.
- Member function definitions.
- Same base name.

67

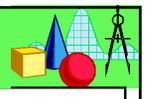
Separating Interface from Implementation



```
1   // Program: time1.h
2   // Declaration of class Time.
3   // Member functions are defined in time1.cpp
4
5   // prevent multiple inclusions of header file
6   #ifndef TIME1_H
7   #define TIME1_H
8
9   // Time abstract data type definition
10  class Time
11          {
12                  private:
13                          …blah…blah…
14                  public:
15                          …blah…blah…
16          };
17
18   <…blah…blah…> functions
19
20  <…blah…blah…> functions
21
24  #endif
25
```

68

Separating Interface from Implementation



```
1   // Program: time1.h
2   // Declaration of class Time.
3   // Member functions are defined in time1.cpp
4
5   // prevent multiple inclusions of header file
6   #ifndef TIME1_H
7   #define TIME1_H
8
9   // Time abstract data type definition
10  class Time
11  {
12          public:
13                  Time();                         // constru
14                  void setTime( int, int, int ); // set hour, minute, second
15                  void printUniversal();         // print universal-time format
16                  void printStandard();          // print standard-time format
17
18          private:
19                  int hour;      // 0 - 23 (24-hour clock format)
20                  int minute;    // 0 - 59
21                  int second;    // 0 - 59
22  }; // end class Time
23
24
```

Preprocessor code to prevent multiple inclusions.

Preprocessor directive defines name **`TIME1_H`**.

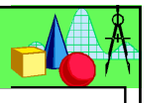
"If not defined"

Naming convention: header file name with underscore replacing period and in uppercase.

69

69

Separating Interface from Implementation

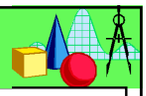


```
25  Time::Time()
26    {
27          hour = minute = second = 0;
28    } // end Time constructor
29
30  // Set new Time value using universal time. Perform validity checks on the data values. Set invalid values to zero.
31
32  void Time::setTime( int h, int m, int s )
33    {
34          hour = ( h >= 0 && h < 24 ) ? h : 0;
35          minute = ( m >= 0 && m < 60 ) ? m : 0;
36          second = ( s >= 0 && s < 60 ) ? s : 0;
37    } // end function setTime
38
39
40  // print Time in universal format
41   void Time::printUniversal()
42    {
43          cout << setfill( '0' ) << setw( 2 ) << hour << ":"
44                  << setw( 2 ) << minute << ":"
45                  << setw( 2 ) << second;
46    } // end function printUniversal
47
48          ...etc...
49  #endif
```

70

70

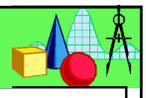
Separating Interface from Implementation



```
1   // Program : Program12.cpp
2   // Program to test class Time.
3   // NOTE: This file must be compiled with time1.cpp.
4   #include <iostream>
5
6   using std::cout;
7   using std::endl;
8
9   // include definition of class Time from time1.h
10  #include "time1.h"
11
12  int main()
13  {
14          Time obj;  // instantiate object t of class Time
15
16          // output Time object t's initial values
17           cout << "The initial universal time is ";
18           obj.printUniversal();   // 00:00:00
19           cout << "\nThe initial standard time is ";
20          obj.printStandard();    // 12:00:00 AM
21
22          obj.setTime( 13, 27, 6 );   // change time
23
```

Include header file **`time1.h`** to ensure correct creation/manipulation and determine size of **`Time`** class object.

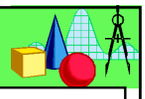
Separating Interface from Implementation



```
24  // output Time object t's new values
25  cout << "\n\nUniversal time after setTime is ";
26  obj.printUniversal();   // 13:27:06
27  cout << "\nStandard time after setTime is ";
28  obj.printStandard();    // 1:27:06 PM
29
30  obj.setTime( 99, 99, 99 );  // attempt invalid settings
31
32  // output t's values after specifying invalid values
33  cout << "\n\nAfter attempting invalid settings:"
34       << "\nUniversal time: ";
35  obj.printUniversal();   // 00:00:00
36  cout << "\nStandard time: ";
37  obj.printStandard();    // 12:00:00 AM
38  cout << endl;
39
40  return 0;
41
42  } // end main
```

```
The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM
```

Controlling Access to Members



Access modes

private

Default access mode

Accessible to member functions and **friend**s

public

Accessible to any function in program with handle to class object

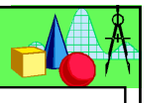
protected

To be discussed later.

73

73

Controlling Access to Members



Class member access

- Default **private**
- Explicitly set to **private**, **public**, **protected**

struct member access

Default **public**

Access to class' **private** data

Controlled with access functions:

Get function (**Accessor** functions)

Read **private** data

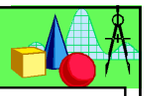
Set function (**Mutator** functions)

Modify **private** data

74

74

Access Functions and Utility Functions



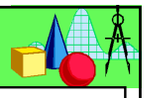
Utility Functions (**Helper** functions)

- **private**
- Support operation of **public** member functions
- Not intended for direct client use

75

75

Initializing Class Objects: Constructors



Constructors

- Initialize data members
  - Or can set later
- Same name as class
- No return type

Initializers

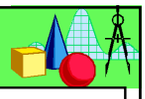
- Passed as arguments to constructor
- In parentheses to right of class name before semicolon

  Class-type ObjectName(value1,value2,…);

76

76

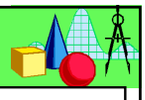
Destructors



Destructors

- Special member function
- Same name as class
  - Preceded with tilde (~)
- No arguments
- No return value
- Cannot be overloaded
- Performs "termination housekeeping"
  - Before system reclaims object's memory
  - Reuse memory for new objects
- No explicit destructor
  - Compiler creates "empty" destructor"

Class Inheritance



Inheritance is a mechanism for:

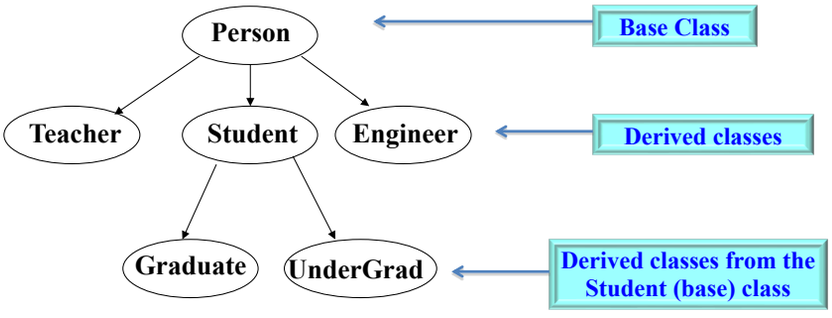
"building new class types from existing class types"

- **Existing class is described as a base class**
- **New class is described as a derived class**

New class types is defined to be:

- A specialization of the base class
- An augmentation of the base class

Class Inheritance



79

79

Class Inheritance

Format:

class ***derivedClass*** **:** ***Access-mode*** ***baseClass***

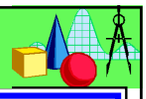
Access-mode:
private
public
protected

```
{
    private :      // Declarations of additional members, if needed.

    public:        // Declarations of additional members, if needed.

    protected:     // Declarations of additional members, if needed.

};
```

80

80

Class Inheritance – Example

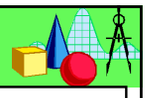


```
class Polygon
  {
     private:
          int width;
          int height;
     public:
          Polygon(){ width = height = 0;}
          void showDims();
          void setDims(int wide, int high);
                {width = wide; height = high;}
  };

void Polygon::showDims()
  {
     cout << "\n\nWidth = " << width
          << "Height = " << height << "\n";
  }
```

```
int main()
  {
     Polygon poly;
     Triangle tri;

     poly.showDims();
     poly.setDims(23, 46);
     poly.showDims();

     tri.showDims();
     tri.setDims(12, 35);
     tri.showDims();
     tri.display();

     return 0;
  }
```

```
class Triangle:public Polygon
  {
     private:
          double area;
     public:
          void setArea()
          void display();
  };
void Triangle::setArea()
  {
     area = width*height * 0.5;
  }

void Triangle::display()
  {
     showDims();
     cout << "Area = " << area <<"\n\n";
  }
```

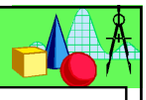
COMP2611 – Data structures



HASHING & COLLISION RESOLUTION

1

Hashing



The process of mapping a key value to a position in a table.

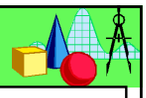
A hash function maps key values to positions. It is denoted by ***h***.

A hash table is an array that holds the records. It is denoted by ***T***.

The hash table has ***M*** slots, indexed from **0** to ***M* - 1**

2

Hashing



For any value ***K*** in the key range and some hash function ***h***,

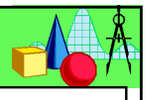
$h(K) = i$; $0 \leq i < M$, such that key $T[i]) = K$.

Load factor

The ratio of the number of items in the table to the table size

Load factor = Number of items / Table size

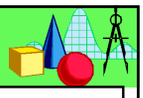
Hash Tables

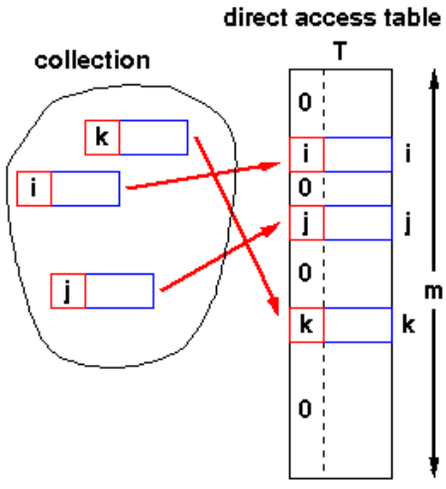


Direct Address Tables

If we have a collection of **n** elements whose keys are unique integers in (1 ... **m**), where **m** >= **n**, then we can store the items in a ***direct address*** table, **T**, where **T[i]** is either empty or contains one of the elements of our collection.

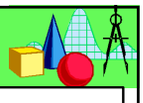
Hash Tables





5

Hash Tables



Direct Address Tables

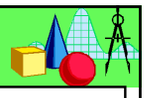
Searching a direct address table operation for a particular key, **k**, we access **T[k]**; if it contains an element, return it, if it doesn't then return a NULL.

There are two constraints here:

- the keys must be unique
- the range of the key must be severely bounded.

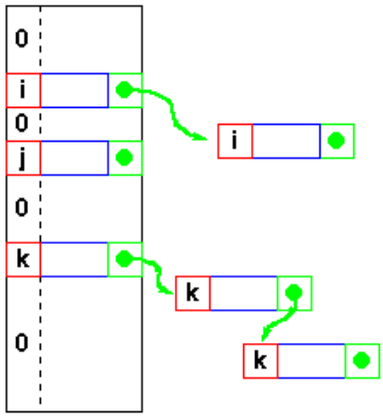
6

Hash Tables



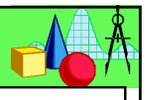
Direct Address Tables

If the keys are not unique, then we can simply construct a set of **m** lists and store the heads of these lists in the direct address table.



7

Hash Tables



Direct Address Tables

The range of the key determines the size of the direct address table and may be too large to be practical.

Therefore…

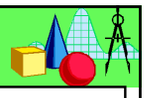
Direct addressing is easily generalized to the case where there is a function,

h(k) => (1, **m**)

which maps each value of the key, **k**, to the range (1,**m**). In this case, we place the element in **T[h(k)]** rather than **T[k]**.

8

Hash Tables



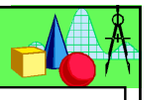
Mapping functions

The direct address approach requires that the function, **h(k)**, is a one-to-one mapping from each **k** to integers in (1,**m**). Such a function is known as a **perfect hashing function**.

The **PHF** maps each key to a distinct integer within some manageable range.

9

Hash Tables



Mapping functions

Finding a perfect hashing function is not always possible.

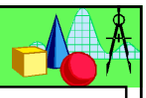
More realistically, a **hash function**, **h(k)**, maps ***most*** of the keys onto unique integers, but maps some other keys on to the same integer.

This clashing is called **Collision.**

If the number of **collisions** is sufficiently small, then the *hash tables* work quite well.

10

Hash Tables

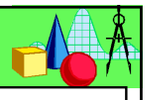


Collision Resolution Techniques

- Chaining
- Overflow Area
- Re-hashing
- Linear probing (using neighbouring slots)
- Quadratic probing
- Random probing.

11

Collision Resolution



1. Chaining

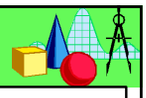
Chain all collisions in lists attached to the appropriate slot.

This allows an unlimited number of collisions to be handled and doesn't require ***a priori*** knowledge of how many elements are contained in the collection.

The tradeoff is the same as with linked lists versus array implementations of collections.
(Linked list has overhead in space and, to a lesser extent, in time.)

12

Collision Resolution



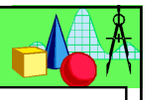
2. Overflow Area

Divide the pre-allocated table into two sections:

1. A ***primary area*** to which keys are mapped
2. An area for collisions, normally termed the ***overflow area***.

When a collision occurs, the colliding data is entered into the first available slot in the overflow area.

Collision Resolution



2. Overflow Area

Example:

Hashing Function: **F1**

Records:

- **Record A**
- **Record B**
- **Record C**

F1(A) = 2

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
…	
n	

---------- Primary Area ----------

– Overflow Area ---

Collision Resolution

2. Overflow Area

Example:

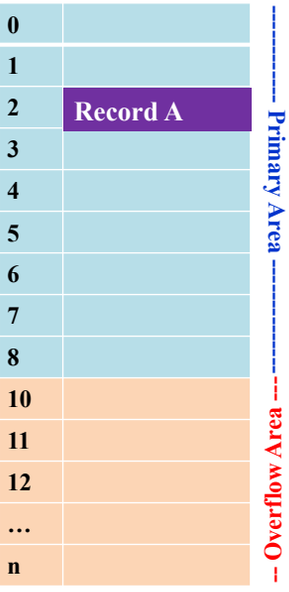
Hashing Function: **F1**

Records:

Record B

Record C

F1(B) = 5



15

Collision Resolution

2. Overflow Area

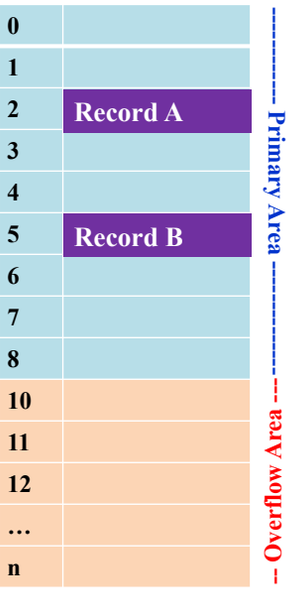
Example:

Hashing Function: **F1**

Records:

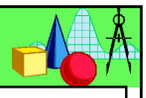
Record C

F1(C) = 2



16

Collision Resolution



2. Overflow Area

Example:

Hashing Function: **F1**

Records:

Record C

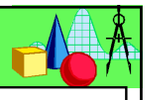
0	
1	
2	Record A
3	
4	
5	Record B
6	
7	
8	
10	
11	
12	
…	
n	

---------- Primary Area ----------

---------- Overflow Area ----------

17

Collision Resolution



2. Overflow Area

Example:

Hashing Function: **F1**

Records:

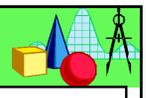
0	
1	
2	Record A
3	
4	
5	Record B
6	
7	
8	
10	Record C
11	
12	
…	
n	

---------- Primary Area ----------

---------- Overflow Area ----------

18

Collision Resolution



3. Re-hashing

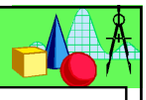
Use another hashing operation when there is a collision. If there is a further collision, use yet another hashing operation (or *re-hash)* until an empty "slot" in the table is found.

The re-hashing function can either be a new function or a modified re-application of the original one.

As long as the functions are applied to a key in the same order, then a sought key can always be located.

19

Collision Resolution



3. Rehashing

Example:

Function: **F1, F2, F3, …**

Records:

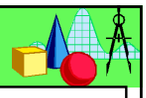
- **Record A**
- **Record B**
- **Record C**

F1(A) = 2

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
…	
n	

20

Collision Resolution



3. Rehashing

Example:

Functions: **F1, F2, F3, …**

Records:

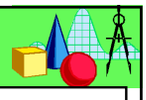
Record B

Record C

F1(B) = 5

0	
1	
2	Record A
3	
4	
5	
6	
7	
8	
10	
11	
12	
…	
n	

Collision Resolution



3. Rehashing

Example:

Functions: **F1, F2, F3, …**

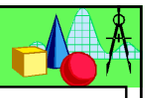
Records:

Record C

F1(C) = 2

0	
1	
2	Record A
3	
4	
5	Record B
6	
7	
8	
10	
11	
12	
…	
n	

Collision Resolution



3. Rehashing

Example:

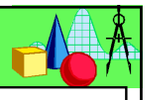
Functions: **F1, F2, F3, …**

Records:

Record C **F2(C) = 5**

0	
1	
2	Record A
3	
4	
5	Record B
6	
7	
8	
10	
11	
12	
…	
n	

Collision Resolution



3. Rehashing

Example:

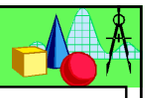
Functions: **F1, F2, F3, …**

Records:

Record C **F3(C) = 8**

0	
1	
2	Record A
3	
4	
5	Record B
6	
7	
8	
10	
11	
12	
…	
n	

Collision Resolution



3. Rehashing

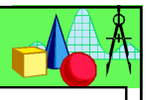
Example:

Functions: **F1, F2, F3, …**

Records:

0	
1	
2	Record A
3	
4	
5	Record B
6	
7	
8	Record C
10	
11	
12	
…	
n	

Collision Resolution



4. Linear Probing – (using neighbouring slots)

One of the simplest re-hashing functions is +1 (or -1)

i.e. on a collision, look in the neighbouring slot in the table.

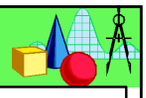
This calculates the new address extremely quickly.

Linear probing is subject to a **clustering** phenomenon.

Re-hashes from one location occupy a block of slots in the table which "grows" towards slots to which other keys hash.

This exacerbates the collision problem and the number of re-hashed can become large

Collision Resolution



5. Quadratic Probing

Better behaviour is usually obtained with **quadratic probing**, where the secondary hash function depends on the re-hash index:

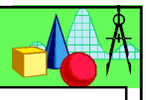
$$address = h(key) + c\ i^2$$

on the i^{th} re-hash.

Since keys which are mapped to the same value by the primary hash function follow the same sequence of addresses, quadratic probing shows **secondary clustering**. However, secondary clustering is not nearly as severe as the clustering shown by linear probes.

27

Collision Resolution



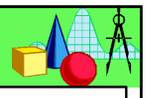
5. Quadratic Probing

Re-hashing schemes use the originally allocated table space and thus avoid linked list overhead, but require advance knowledge of the number of items to be stored.

However, the collision elements are stored in slots to which other key values map directly, thus the potential for multiple collisions increases as the table becomes full.

28

Collision Resolution



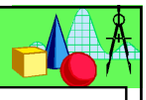
6. (Pseudo) Random probing

Use a random number generator to produce the sequence of addresses for the element.

Random number sequence must be predictable and capable of being regenerated from a seed, such as the key value - otherwise, we will not get the same sequence at insertion and query times!

29

Collision Resolution



Example:

Using the hash function:

h (k) = 2k modulus 10

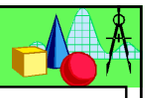
Insert the numbers:

2, 20, 7, 15, 3, 0, 4

into a one-dimensional array of 10 elements using linear probing to resolve collisions.

30

Collision Resolution



h (k) = 2k mod 10

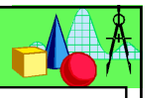
*= (2*k) % 10*

2, 20, 7, 15, 3, 0, 4, 14

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

31

Collision Resolution



h (k) = 2k mod 10

*= (2*k) % 10*

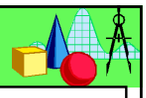
2, 20, 7, 15, 3, 0, 4, 14

h(2) = 2*2 % 10 = 4

0	
1	
2	
3	
4	2
5	
6	
7	
8	
9	

32

Collision Resolution



h (k) = 2k mod 10

*= (2*k) % 10*

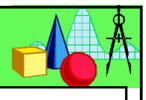
2, 20, 7, 15, 3, 0, 4, 14

h(20) = 2*20 % 10 = 0

0	20
1	
2	
3	
4	2
5	
6	
7	
8	
9	

33

Collision Resolution



h (k) = 2k mod 10

*= (2*k) % 10*

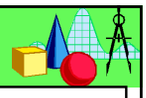
2, 20, 7, 15, 3, 0, 4, 14

h(7) = 2*7 % 10 = 4

0	20	
1		
2		
3		
4	2	7
5		
6		
7		
8		
9		

34

Collision Resolution



h (k) = 2k mod 10

*= (2*k) % 10*

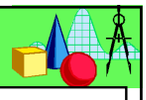
2, 20, 7, 15, 3, 0, 4, 14

h(7) = 2*7 % 10 = 4

0	20
1	
2	
3	
4	2
5	7
6	
7	
8	
9	

35

Collision Resolution



h (k) = 2k mod 10

*= (2*k) % 10*

2, 20, 7, 15, 3, 0, 4, 14

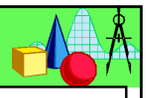
h(15) = 2*15 % 10 = 0

0	20
1	
2	
3	
4	2
5	7
6	
7	
8	
9	

15

36

Collision Resolution



h (k) = 2k mod 10

*= (2*k) % 10*

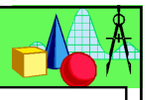
2, 20, 7, 15, 3, 0, 4, 14

h(3) = 2*3 % 10 = 6

0	20
1	15
2	
3	
4	2
5	7
6	
7	
8	
9	

37

Collision Resolution



h (k) = 2k mod 10

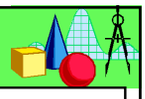
*= (2*k) % 10*

2, 20, 7, 15, 3, 0, 4, 14

0	20
1	15
2	0
3	
4	2
5	7
6	3
7	
8	4
9	14

38

Data Structures

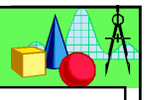


Abstract Data Types

(ADTs)

1

Abstract Data Type



Definition:

A data structure and a collection of functions or procedures which operate on the data structure.

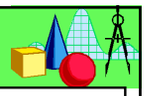
Characteristics of ADTs:

Must be able to:

- ❖ Create a new collection
- ❖ Add an item to a collection
- ❖ Delete an item from a collection
- ❖ Find an item matching some criterion in the collection
- ❖ Destroy the collection

2

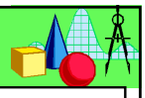
Abstract Data Type



Programming Properties:

1. **Not be limited by requirement for contiguous RAM locations for data storage.**
2. **Use memory RAM resources ONLY when strictly required.**
3. **Ability to grow and shrink structure with data availability.**
4. **Minimize disruption of other data nodes when adding or removing a node within the data structure.**
5. **And… Always... *Fast processing speed!!***

Introduction



Fixed-size data structures

Arrays, structs

Dynamic data structures

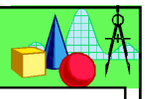
Linked lists (*singly and doubly linked lists*)
Stacks
Queues
Priority Queues
Double-Ended Queues

Linear Data Structures

Trees (Binary trees)

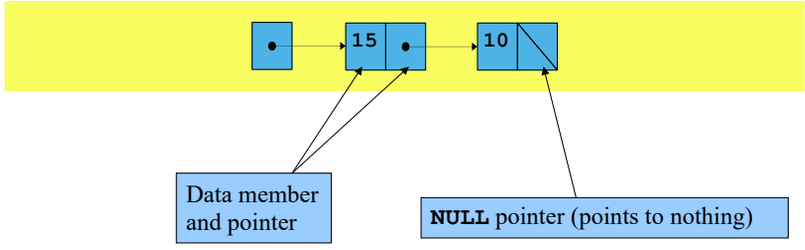
Hierarchical Data Structures

Self-Referential Classes



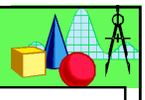
Self-referential class

- Has pointer to object of same class
- Link together to form useful data structures
  - Lists, stacks, queues, trees
- Terminated with **NULL** pointer



5

Self-Referential Classes



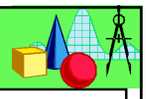
Sample code

```
class Node
    {
        private:
            int data;
            Node *nextPtr;
        public:
            Node( int );
            void setData( int );
            int getData() const;
            void setNextPtr( Node * );
            Node *getNextPtr()const;
    };
```

Pointer to object is called a ***link*** - **nextPtr** points to a **Node**

6

Dynamic Memory Allocation and Data Structures



Dynamic memory allocation

- Obtain and release memory during program execution
- Create and remove nodes

Operator new

- Takes type of object to create
- Returns pointer to newly created object
- Returns **bad_alloc** if not enough memory

e.g.

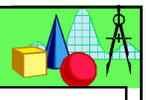
Node *newPtr = new Node(10);

10 is the node's object data

```
Node::Node ( int val )
  {
     data = val;
     NextPtr = NULL;
  }
```

```
Node *newPtr;
newPtr = new Node(10);
```


Dynamic Memory Allocation and Data Structures



Operator delete

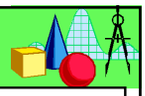
- Deallocates memory allocated by **new**, calls destructor
- Memory returned to system, can be used in future
- Pointer is not deleted, only the space it points to

Example

delete newPtr;

Note: **newPtr** continues to be a valid variable – only the data space is released.

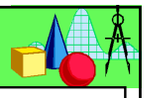
Linked Lists



Linked list

- Collection of self-referential class objects (nodes) connected by pointers (links)
- Accessed using pointer to first node of list
  - Subsequent nodes accessed using the links in each node
- Link in last node is NULL (zero)
  - Indicates end of list
- Data stored dynamically
  - Nodes created as necessary
  - Node can have data of any type

Linked Lists



Types of linked lists

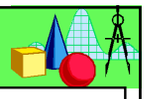
Singly linked list (used in example)

- Pointer to first node
- Node-to-node travel is in one direction
- Null-terminated

Circular, singly-linked

- As above, but last node points back to the first
- No Null-termination

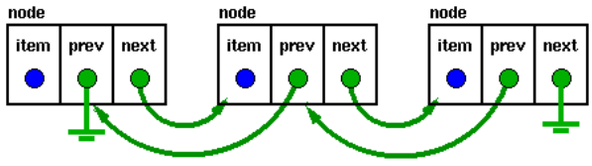
Linked Lists



Types of linked lists

Doubly-linked list

- Each node has a forward and backwards pointer
- Traverses forward or backward
- Previous pointer of first node points to NULL
- Next pointer of last node points to NULL

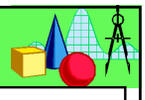


Circular, double-linked

As above, but first and last node joined

11

Queues



Characteristics:

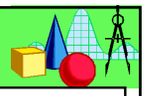
- Like waiting in line
- Nodes are added to the back (***tail***)
- Nodes are removed from the front (***head***)
- First-in, first-out (FIFO) data structure
- Insertion is called **Enqueue**
- Removal is called **Dequeue**

Applications:

- Print spooling
  - Documents wait in queue until printer is available
- Packets on network
- File requests from server

12

Queues



Typical Operations:

dequeue - Remove and return the Object from the front of queue or return an error if the queue is empty.

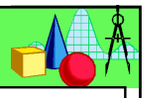
enqueue - Inserts an object at the rear of the queue.

isEmpty - Return a boolean indicating if the queue is empty

front - Return the Object at the front of the queue without removing the Object. If the queue is empty, return an error.

size - Return the number of Objects in the queue.

Deques



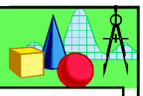
Double-Ended Queues

Characteristics:

- Nodes are added to both the back (***tail***) and to the front (***head).***
- Nodes are removed from both the tail and the head
- Not First-in, first-out (FIFO) data structure
- Insertion is also called **Enqueue**
- Removal is also called **Dequeue**

NB. Do not confuse Deque with Dequeue

Deques



Operations

insertFirst - Inserts an Object at the front of the deque.

insertLast - Inserts an Object at the rear of the deque.

removeFirst - Removes and returns the Object from the front of the deque.

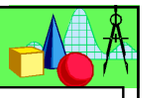
removeLast - Removes and returns the Object from the rear of the deque.

First / last - Returns the first / last Object in the deque without removing the Object.

isEmpty - Returns a boolean indicating if the deque is empty.

size - Returns the number of Objects in the deque.

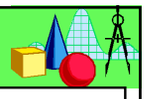
Priority Queues



Characteristics:

- Like an ordinary queue.
- Has a head and a tail.
- Elements are **ONLY removed from front** ***(dequeue)***.
- Elements are ordered so that the item with highest priority is always in front.
- When an item is inserted ***(enqueued)*** the order must be maintained.
- Said to be an **ascending-priority queue** if the item with smallest key has the highest priority.
- Said to be a **descending-priority queue** if the item with largest key has the highest priority.

Stacks



Characteristics:

- Nodes are added and/or removed from the top
  - Constrained version of linked list
  - Like a stack of plates
- Last-in, first-out (LIFO) data structure
- Last element (bottom) of stack points to NULL

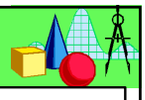
Stack operations

Push: add node to top

Pop : remove node from top

17

COMP2611 – Linear ADTs



Summary

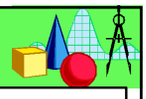
Queue

- Lists which employ a first-in, first-out (FIFO) concept.
- Nodes are removed only from the head and nodes are inserted only at the tail.

Priority Queue

- A list which imposes a sorted priority on the elements so that the element with the highest priority is always at the head of the list.
- This sorting is done whenever a new element is added to the list.
- Nodes are removed only from the head – i.e. the node with the highest priority.

18



Stack

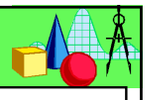
- A list which employs a last-in, first-out (LIFO) concept. Nodes are added and removed only from the top of the structure.

Deque (Double-Ended Queue)

- A double ended queue is a list which permits insertion and deletion at both the head and tail of the list.
- Insertion and deletion are only permitted at the two ends.

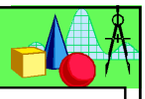
LIST ADT

(Using only a string data element as the payload)



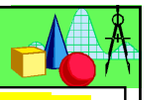
```
class Node
        {
                private:
                        string Value;  //This is the data part--it contains the payload
                        Node*  Next;   //This contains the address of the next node

                public:
                        Node ( void )        { Value = " "; Next = NULL; }
                        Node ( string Vx ) { Value = Vx; Next = NULL; }

                         //Define the accessor member functions...
                        string getValue ( void )   { return Value; }
                        Node*  getNext ( void )   { return Next;  }

                         //Define the mutators...
                         void setValue ( string Vx )  { Value = Vx; }
                         void setNext ( Node* Nx )   { Next  = Nx; }
        };
```

LIST ADT



```
class LinkedList
        {
                private:
                        Node* Head;

                public:
                        // Constructor function
                        LinkedList ( void )  { Head = NULL;}

                        // Mutator or Set functions
                        void addNode ( string );
                        void addBegin ( string );
                        void addEnd ( string );
                        void delNode ( string );

                        // Accessor or Get functions
                        string getBegin ( void );
                        string getEnd ( void );
                        bool find ( string );
                        void showNodes ( void );
                        int isEmpty ( void );
        };
```


LIST ADT

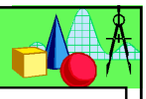


```
void LinkedList :: addBegin ( string Vx )
        {
                Node* Cur = Head;
                Head = new Node(Vx);
                Head → setNext(Cur);
        }
```

Create a new node,

Go to the node that Head is pointing to and give its next pointer field the address of Cur

LIST ADT

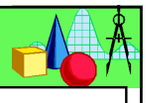


```
void LinkedList :: addEnd ( string Vx )
        {
                Node* Cur;
                if  ( Head == NULL )
                        Head = new Node(Vx);
                else
                   {
                        for ( Cur = Head ; Cur → getNext() != NULL ;
                             Cur = Cur → getNext());
                        Cur->setNext ( new Node ( Vx ) );
                    }
        }
```

Note:

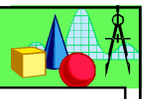
Since the for statement ends with a ';' only the incrementing part is executed.

LIST ADT



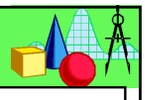
```
void LinkedList :: delNode ( string Vx )
        {
                Node* Cur;
                Node* Prev;

                cout << "\nDeleting..." << Vx << "...\n";

                //This is a special case--an empty list...
                 if ( Head == NULL)
                        return;  //It's OK to return from a void
```

LIST ADT



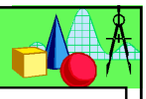
```
//Removing the first node is a special case...

if ( Head → getValue() == Vx )
        {
                Cur = Head;
                Head = Head → getNext();
                delete Cur;
                return;
         }
```


LIST ADT



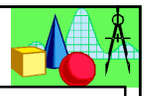
```
//Normal case...removing the second through to the last node...

Prev = Head;

for( Cur = Prev → getNext(); Cur != NULL &&
                              strcmp(Cur->getValue(), Vx)  != 0;   )
   {
        Cur = Cur → getNext();     //...Walk down through the list...
         Prev = Prev → getNext();
    }

 if ( Cur == NULL )                //...You have reached the end...
        cout << Vx <<" was not found in the list...\n";
 else                              //...You have found the value...
     {
```

LIST ADT



```
        else                    //...You have found the value...
            {
                    Prev → setNext ( Cur → getNext() );
                    delete Cur;
             }
}
```

COMP2611 – Data Structures

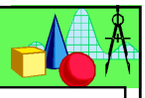


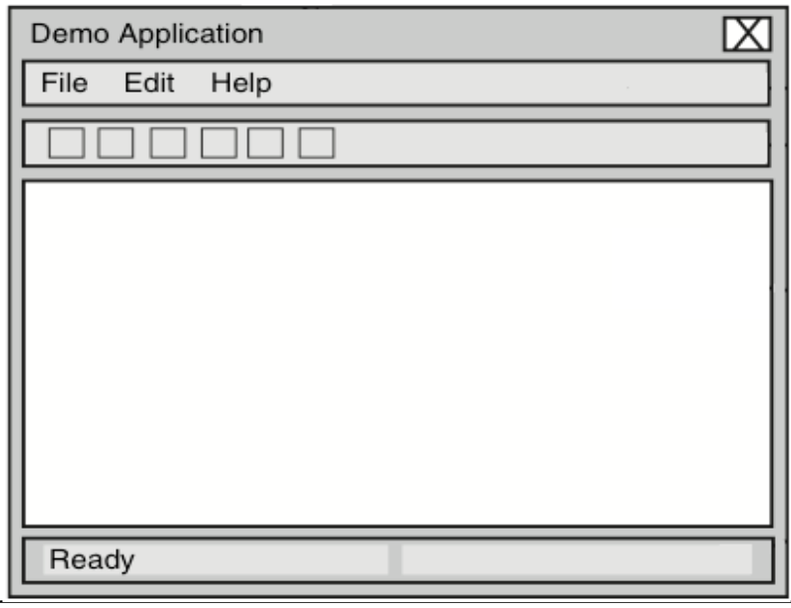
COMP2611

Lecture 04

1

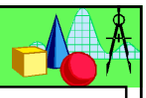
COMP2611 – Data Structures





2

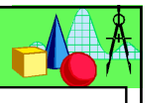
COMP2611 – Data Structures



wxWidgets

3

wxWidgets

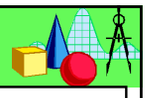


Introduction:

- **Application Program Interface (API)**
- **Provides tools for developing Graphical User Interfaces (GUI)**
- **Code is written in C++**
- **Supports multi-platform application**
- **Free of cost**

4

wxWidgets

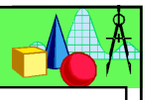


Coding using wxWidgets:

- We will inherit from TWO main classes in the wxWigets API:
  - wxApp
  - wxFrame
- wxApp – To create a running (executing) application
- wxFrame – To define our own frame for the GUI
- Also make use of special Compiler Directives which are already defined in the API.
  - i.e. : DECLARE_APP and IMPLEMENT_APP

5

wxWidgets

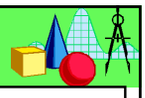


Coding using wxWidgets:

- API is largely defined in the header file "wx/wx.h"
- Define your own APPLICATION by inheriting from wxApp
- Define your own FRAME by inheriting from wxFrame
- Define events (tasks): Functionalities are described as events.
  - e.g. matching menu items to mouse clicks, etc. for execution.
- Events must be defined in an EVENT_TABLE

6

Coding Using wxWidgets



1. Always include the header file **wx.h**

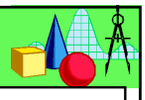
```
#include <wx/wx.h>
```

2. Name an inherited application class from **wxApp** and declare it with a function to execute the program:

```
class MyApp : public wxApp
    {
        public:
            virtual bool OnInit();  // Application to startup
    };
```

7

Coding Using wxWidgets



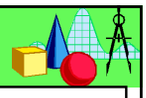
3. Declare the inherited main frame class from **wxFrame**. In this class also will ALL the events handlers be declared:

```
class MyFrame : public wxFrame
    {
        private:
            DECLARE_EVENT_TABLE()      // This handles events
        public:
            MyFrame(const wxString&  title);    // Constructor

            // Event handlers
            void OnQuit ( wxCommandEvent& event );
            void OnAbout ( wxCommandEvent& event );
                …etc…
    };
```

Note: wxCommandEvent& event is a mouse click parameter

8

Coding Using wxWidgets



4. Declare the compiler directives

- *Declare Application class*

```
DECLARE_APP(MyApp)
```

- *Create Application class object*

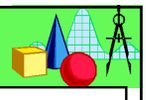
```
IMPLEMENT_APP(MyApp)
```

- *The Enumeration Table*

```
enum
    {
        wxID_Quit  =  wxID_HIGHEST + 1,
        wxID_ABOUT,
        …etc…
    };
```

9

Coding Using wxWidgets



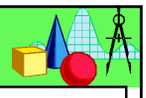
5. Define the Event table

- *Event table for inherited Frame class defined*

```
BEGIN_EVENT_TABLE ( MyFrame, wxFrame )
        EVT_MENU ( wxID_ABOUT, MyFrame::OnAbout)
        EVT_MENU ( wxID_EXIT, MyFrame::OnQuit )
END_EVENT_TABLE ( )
```

10

Coding Using wxWidgets



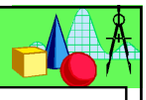
6. Define the Application class function to initialize the application

Frame Caption

```
bool MyApp::OnInit()
    {
        // Create the main application window
        MyFrame *frame = new MyFrame(wxT("Minimal
                                          wxWidgets App"));

        // Display it
        frame->Show(true);

        // Start the event loop
        return true;
    }
```

11

Coding Using wxWidgets

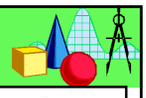


7. **Define member functions for the Frame class**

```
void MyFrame::OnAbout ( wxCommandEvent& event )
    {
        wxString msg;
        msg.Printf ( wxT ( "Hello and welcome to %s" ),
                     wxVERSION_STRING );
        wxMessageBox ( msg, wxT("About Minimal"), wxOK |
                     wxICON_INFORMATION, this );
    }

void MyFrame::OnQuit ( wxCommandEvent& event )
    {
        // Destroy the frame
        Close( );
    }
```

12

Coding Using wxWidgets



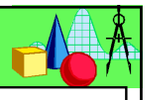
8. Define the Constructor functions for the Frame class

…this is where it all happens…

- **Frame icon** *(optional)*
- **Define the primary menu items**
- **Define the sub-menu items for the primary menu items**
- **Define a menu bar to put the primary menu items on**
- **Append the primary menu items to the menu bar with the appropriate labels**
- **Activate the menu bar in the GUI**
- **Activate a status bar at the bottom of the GUI** *(optional)*
- **Insert a string in the status bar**

13

Coding Using wxWidgets

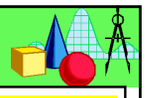


Constructor

```
MyFrame::MyFrame(const wxString& title) : wxFrame ( NULL,
wxID_ANY, title)
   {
        …

   }
```

14

Coding Using wxWidgets

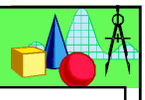


```
{
    // Set the frame icon - optional
    SetIcon(wxIcon(wxT("iconImage.xpm")));

    // Create menu items for menu bar
    wxMenu *fileMenu = new wxMenu;

    // The "About" item should be in the help menu
    wxMenu *helpMenu = new wxMenu;
    helpMenu->Append(wxID_ABOUT, wxT("&About...\tF1"),
                                 wxT("Show about dialog"));
    fileMenu->Append(wxID_EXIT, wxT("E&xit\tAlt-X"),
                                wxT("Quit this program"));
    …etc..
}
```

Image MUST be in "xpm" format

15

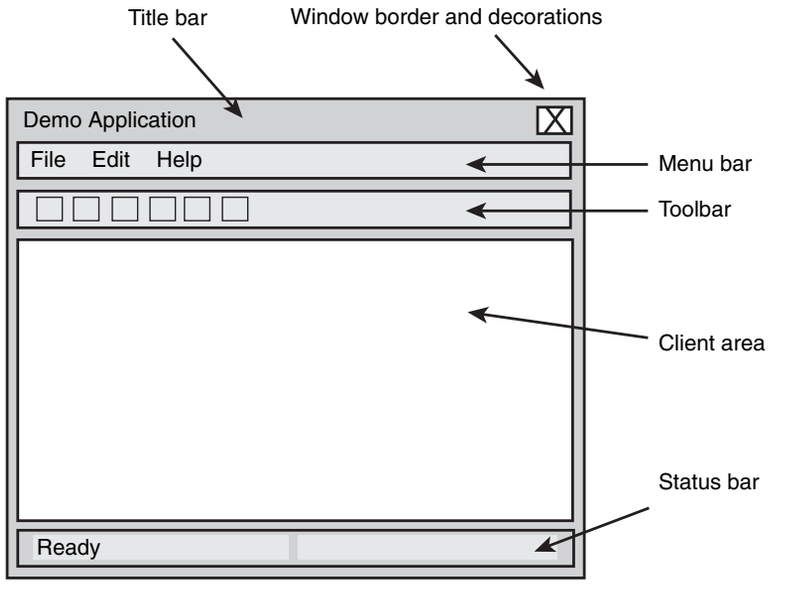
Coding Using wxWidgets



```
{
    …etc…
    // Now append the freshly created menu to the menu bar...
    wxMenuBar *menuBar = new wxMenuBar();
    menuBar->Append(fileMenu, wxT("&File"));
    menuBar->Append(helpMenu, wxT("&Help"));

    // ... and attach this menu bar to the frame
    SetMenuBar(menuBar);

    // Create a status bar just for fun
    CreateStatusBar(2);

    SetStatusText(wxT("Welcome to wxWidgets!"));
}
```

16

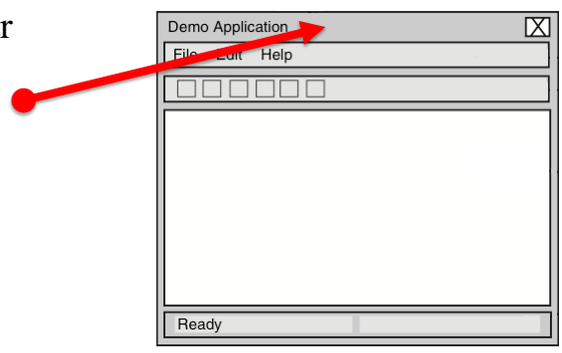
Things to remember…



17

Things to remember…

The text for the Title Bar is entered in the call to the Constructor function in your Frame class' initialization (**OnInit**) function.

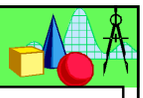


e.g.

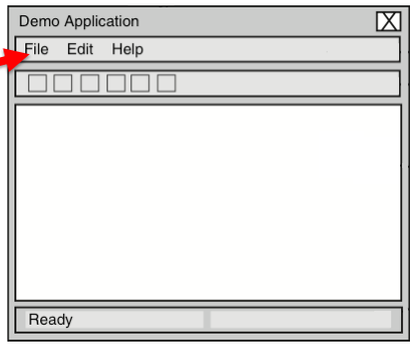
```
MyFrame *frame = new MyFrame (
                wxT( "Demo Application") ,
                wxPoint(50,50), wxSize(800,600) );
```

18

Things to remember…



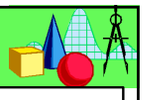
All the Main Menu items are defined in the Constructor function in your Frame class.



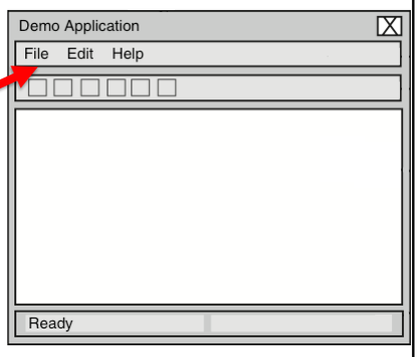
e.g
```
wxMenu *menuFile  = new wxMenu;
wxMenu *menuEdit  = new wxMenu;
wxMenu *menuHelp  = new wxMenu;
```

19

Things to remember…



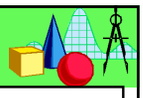
Displaying the Main Menu items on the main menu bar is also defined in the Constructor function in your Frame class.



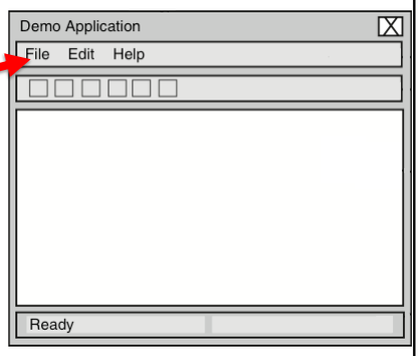
e.g
```
menuBar->Append( menuFile,   wxT ( "&File" ) );
menuBar->Append( menuEdit,   wxT ( "&Edit" ) );
menuBar->Append( menuHelp,   wxT ( "&Help" ) );
```

20

Things to remember…



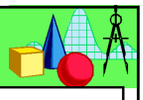
All the **Sub-Menu items** are defined in the **Constructor function** in your Frame Class and the **ID** must be added to the **enum list** and **Event Table.**



e.g **menuFile->Append (ID_Open, ("&Open"),**
(" Open an existing file"));

21

Things to remember…

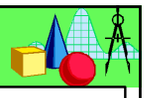


ABSOLUTELY IMPORTANT!

ALL sub-menu items must have an **enum ID** and the **ID** must be associated with a member function from your **frame class** in the **Event Table**.

22

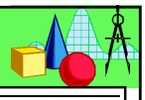
Things to remember…



```
class MyFrame: public wxFrame
    {
        private:
                DECLARE_EVENT_TABLE()

        public:
                MyFrame(const wxString& title, const wxPoint&
                            pos, const wxSize& size);
                void OpenFile (wxCommandEvent& (event));
                void SaveFile ( wxCommandEvent& event );
                void SaveFileAs ( wxCommandEvent& event );
                void OnQuit ( wxCommandEvent& event );
                void OnDisplay ( wxCommandEvent& event );
                …
```

23

Things to remember…



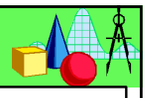
Class member function in frame class for sub-menu items	enum ID	EVENT_TABLE entry
OpenFile	ID_Open	EVT_MENU (ID_Open, MyFrame::OpenFile)
OnQuit	ID_Quit	EVT_MENU (ID_Quit, MyFrame::OnQuit)
OnDisplay	ID_Display	EVT_MENU (ID_Display, MyFrame::OnDisplay)
OnSave	ID_Save	EVT_MENU (ID_Save, MyFrame::OnSave)
OnSaveAs	ID_SaveAs	EVT_MENU (ID_SaveAs, MyFrame::OnSaveAs)

```
enum
{
   ID_Open = wxID_HIGHEST + 1,
   ID_Quit,
   ID_Display,
   ID_Save,
   ID_Sfile,
 };
```

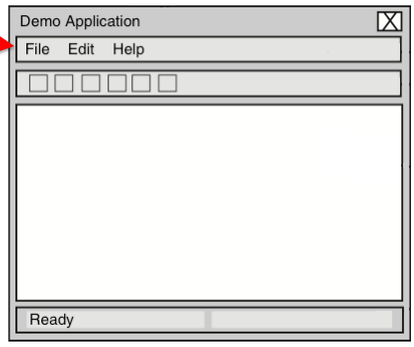
```
BEGIN_EVENT_TABLE ( MyFrame, wxFrame )
      EVT_MENU(ID_Open, MyFrame::OpenFile)
      EVT_MENU ( ID_Quit, MyFrame::OnQuit )
      EVT_MENU(ID_Display, MyFrame::DisplayFile)
      EVT_MENU(ID_SFile, MyFrame::SaveFile)
END_EVENT_TABLE ( )
```

24

Things to remember…



The **Main Menu Bar** is defined and set in the Constructor function in your Frame class.

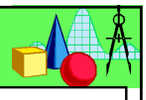


e.g

```
wxMenuBar *menuBar = new wxMenuBar;
SetMenuBar( menuBar );
```

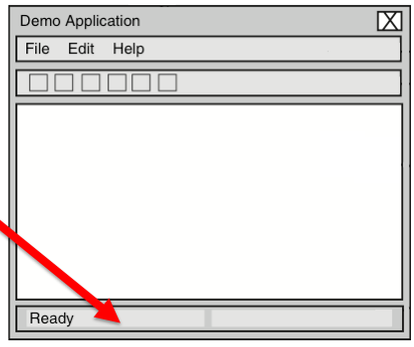
25

Things to remember…



The **Status Bar** is defined in the Constructor function in your Frame class.

The number of partitions is passed in the parameter list.



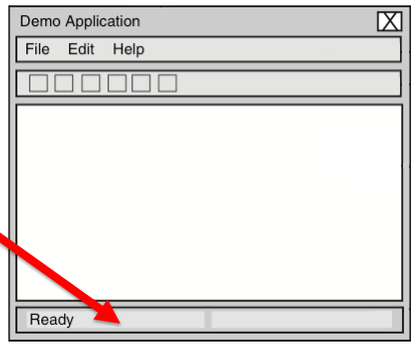
e.g

```
CreateStatusBar(2);
```

26

Things to remember…

The **Status Bar** is populated
With the first partition as **0**;
the second partition is **1**; etc.

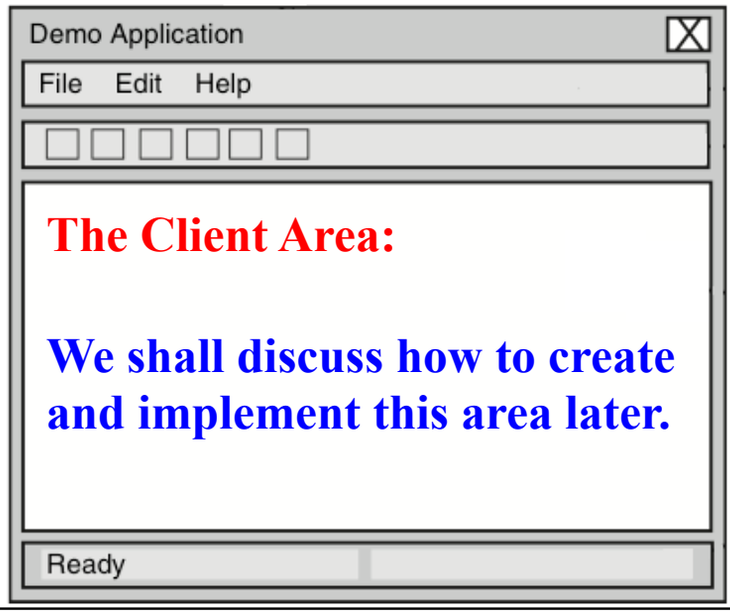


The first partition is not
normally enumerated.
e.g

```
CreateStatusBar ( 3 );
SetStatusText ( wxT("  Dr. John Charlery") );
SetStatusText ( wxT("  Project Assignment"), 1 );
SetStatusText ( wxT("  Ready..."), 2 );
```

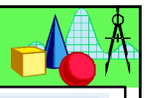
27

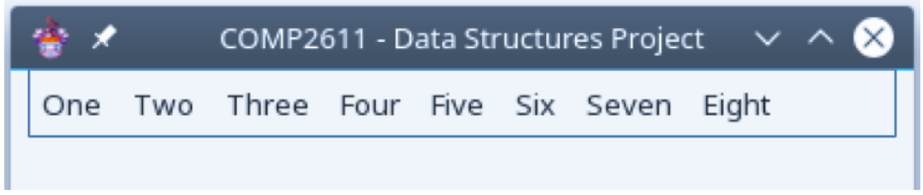
Things to remember…



28

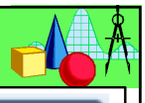
wxWidgets

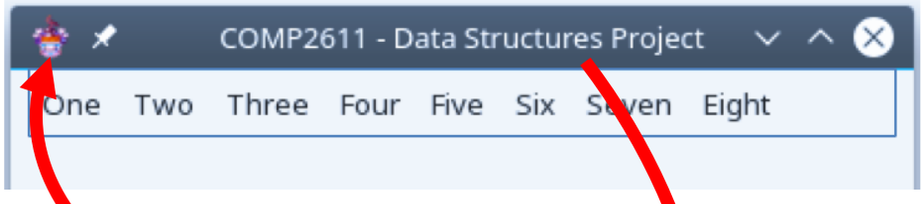




29

wxWidgets

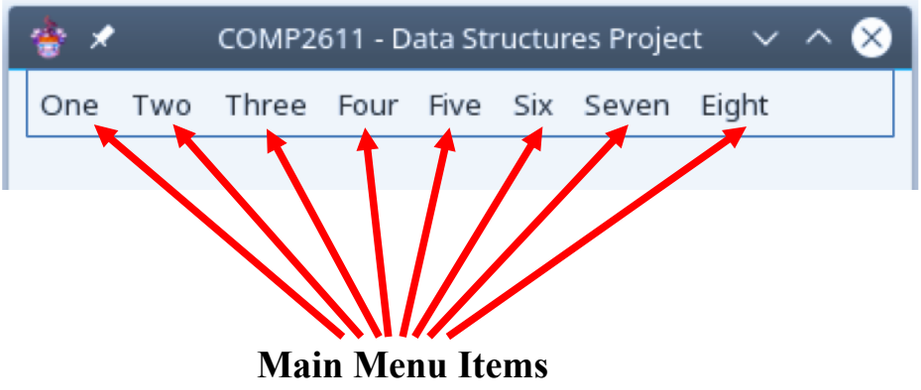





```
SetIcon(wxIcon(wxT("uwiIcon.xpm")));

bool MyApp::OnInit()
 {
    // Create the main application window
    MyFrame *frame = new MyFrame (
                        wxT("COMP2611 - Data Structures"),
                        wxPoint (50,50),  wxSize (600,400) );
```

30

wxWidgets

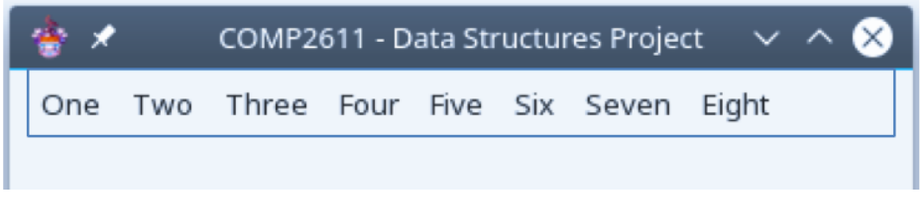


Needed:

8 wxMenu pointer variables for the 8 main menu items!

31

wxWidgets

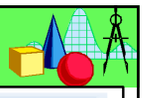


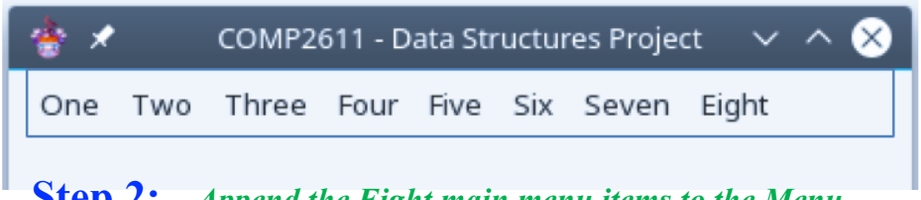
Step 1: *// Create Eight main-menu items*

```
wxMenu *menuOne   =  new wxMenu;
wxMenu *menuTwo   =  new wxMenu;
wxMenu *menuThree =  new wxMenu;
wxMenu *menuFour  =  new wxMenu;
wxMenu *menuFive  =  new wxMenu;
wxMenu *menuSix   =  new wxMenu;
wxMenu *menuSeven =  new wxMenu;
wxMenu *menuEight =  new wxMenu;
```

32

wxWidgets





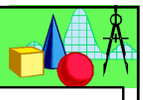
Step 2: *Append the Eight main menu items to the Menu Bar and list them as: One Two Three ... Eight*

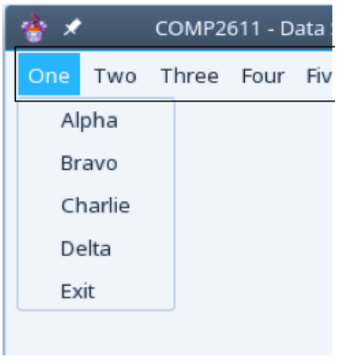
Pointer Variables Names Display Menu Items' Names

```
menuBar->Append   menuOne,     wxT("One") );
menuBar->Append   menuTwo,     wxT("Two") );
menuBar->Append   menuThree,   wxT("Three") );
menuBar->Append   menuFour,    wxT("Four") );
menuBar->Append   menuFive,    wxT("Five") );
menuBar->Append   menuSix,     wxT("Six") );
menuBar->Append   menuSeven,   wxT("Seven") );
menuBar->Append   menuEight,   wxT("Eight") );
```

33

Sub-Menu Items

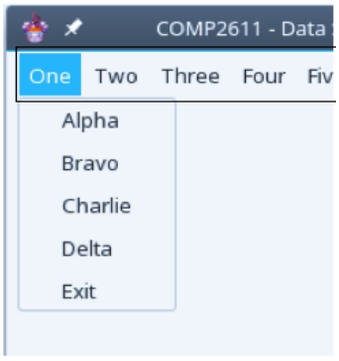




- Appending the five sub-menu items to the main menu item "**One**", requires 5 events.
- Each event **MUST** be defined with **Enumerations.**
- Each event **MUST** be tied to a member functions of either the **wxFrame class** or the **user-defined frame class**, in the Event table.

34

Sub-Menu Items



Note:

Main menu item "One", is the wxMenu pointer variable "menuOne".

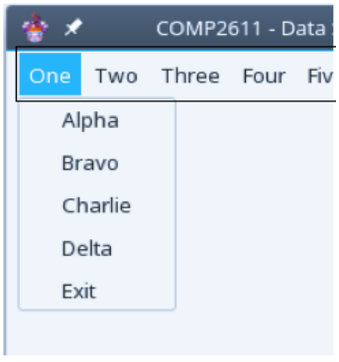
This is the variable that we will append the sub-menu items to.

Step 1: ***Enumeration names for the sub-menu items:***

```
enum
    {
        ID_Alpha = wxID_HIGHEST + 1,
        ID_Bravo,  ID_Charlie,  ID_Delta,  ID_Exit,
    };
```

35

Sub-Menu Items



Step 2: ***Append the enumerated values to the main menu item "MenuOne" in the frame constructor function.***

```
menuOne->Append( ID_Alpha,   wxT("&Alpha"),   wxT("") );
menuOne->Append( ID_Bravo,   wxT("&Bravo"),   wxT("") );
menuOne->Append( ID_Charlie, wxT("&Charlie"), wxT("") );
menuOne->Append( ID_Delta,   wxT("&Delta"),   wxT("") );
menuOne->Append( ID_Exit,    wxT("E&xit"),    wxT("") );
```

36

Sub-Menu Items

Step 3: ***Associate a member function to each sub-menu item (event) in the Event Table.***

Note: ***The function MUST be previously defined in either wxFrame or your user-defined frame class.***

```
BEGIN_EVENT_TABLE ( MyFrame, wxFrame )
        EVT_MENU ( ID_Alpha,    MyFrame::OnAlpha )
        EVT_MENU ( ID_Bravo,    MyFrame::OnBravo )
        EVT_MENU ( ID_Charlie,  MyFrame::OnCharlie )
        EVT_MENU ( ID_Delta,    MyFrame::OnDelata )
        EVT_MENU ( ID_Exit,     MyFrame::OnExit )
END_EVENT_TABLE ()
```

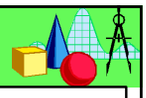
37

Member Functions

```
void MyFrame::OnAlpha ( wxCommandEvent& event )
    {
            …
    }

void MyFrame::OnDelta ( wxCommandEvent& event )
    {
            …
    }

void MyFrame::OnQuit ( wxCommandEvent& event )
    {
            // Destroy the frame
            Close( );
    }
```

38

Remember…



Step 1: Including the header file **wx.h**

Step 2: Naming an inherited application class from **wxApp** and declare it with the member function to execute the program.

Step 3: Declaring the inherited main frame class from **wxFrame**. In this class also will ALL the events handlers be declared.

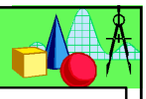
Step 4: Declare the compiler directives.

Step 5: Define the Application class function to initialize the application.

Step 6: Define the Constructor functions for the Frame class.

Step 7: Define member functions for the Frame class.

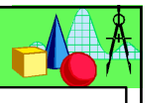
39



TREE

ADTs

Trees



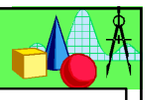
Linear Data Structures

Lists, queues, stacks, deques, etc.

Hierarchical Data Structures (Trees)

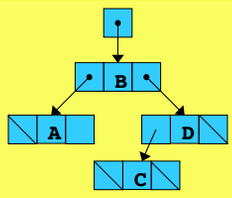
- Nonlinear, two-dimensional
- Tree nodes have 2 or more links
- Binary trees have exactly 2 links/node
  - None, both, or one link can be null

Trees



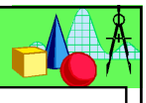
Terminology

- ***Root node***: first node on tree
- **Link** refers to ***child*** of node
  - **Left child** is root of ***left subtree***
  - **Right child** is root of ***right subtree***
- ***Leaf node***: node with no children
- Trees drawn from root downwards



3

Tree ADT



Definition

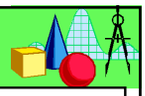
A tree **T** is a finite set of one or more nodes such that there is one designated node **r** called the root of **T**, and the remaining nodes in (**T – { r }**) are partitioned into n ≥ 0 disjoint subsets **T1, T2, ..., Tk**, each of which is a tree, and whose roots **r1 , r2 , ..., rk** , respectively, are children of **r**.

Characteristics

- A tree is an instance of a more general category called **graph**
- A tree consists of nodes connected by **edges**
- Typically, there is one node in the top row of the tree, with links connecting to more nodes on the second row, even more on the third, and so on.

4

Tree ADT

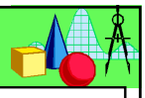


Root

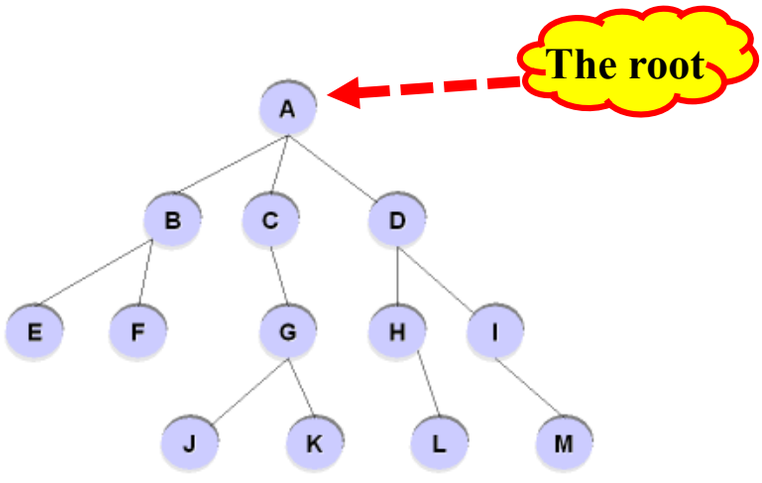
- The node at the top of the tree is called the root.
- There is **only one root** in a tree.
- For a collection of nodes and edges to be defined a tree, there must be one (and only one!) path from the root to any other node.
- There can be only **ONE** point of entry into a node, but there can be multiple exits out of the node. (For a BST, a maximum of two exits).

5

Tree ADT

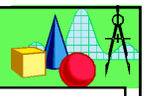


Root



6

Tree ADT

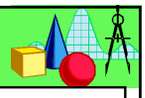


Path

If $n_1, n_2, \ldots, n_k$ is a set of nodes in the tree such that n_i, is the parent of n_{i+1} for $1 \leq i \geq k$, then this set is called a path from n_1 to n_k. The length of the path is $k - 1$.

The **number of edges** that connect nodes to the root of the tree is the length of the path.

Tree ADT



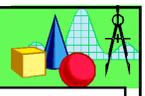
Parent

- Any node (except the root) has exactly one edge running upward to another node. The node above it is called the **parent** of the node.
- If there is a path from node *R* to node *M*, then *R* is an **ancestor** of *M*.
- Parents, grandparents, etc. are ancestors.

Child

- The nodes below a given node are called its **children**.
- If there is a path from node *R* to node *M*, then *M* is a **descendant** of *R*.
- Children, grandchildren, etc. are descendants

Tree ADT



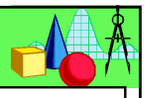
Leaf and Internal node

- A node that has no children is called a leaf node or simply a leaf.
- There can be only one root in a tree, but there can be many leaves.
- A node (apart from the root) that has children is an **internal node**.

Levels

The level of a particular node refers to **how many generations the node is away from the root**. The root is assumed to be on level 0, hence its children will be on level 1, its grandchildren will be on level 2, and so on.

Tree ADT



Depth

The depth of a node M in a tree is the **length of the path** from the root of the tree to M.

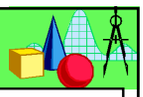
Height

The height of a tree is **one more than the depth of the deepest node** in the tree.

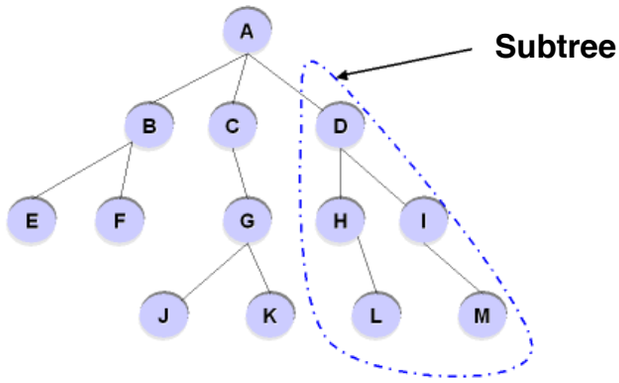
Subtree

Any node may be considered to be the root of the subtree, which consists of its children and its children's children, and so on. If you think in terms of families, a node's subtree contains all its descendants.

Tree ADT

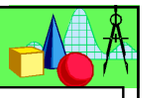


Subtree



11

Tree ADT



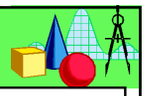
Visiting

A node is visited when program control arrives at the node, usually for the purpose of carrying out some operation on the node, such as checking the value of one of its data fields, or displaying it.

Merely passing over a node on the path form one node to another is NOT considered to be visiting the node.

12

Tree ADT



Traversing

To traverse a tree means to visit all of the nodes in some specified order.

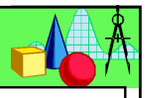
> **For example, you might visit all the nodes in order of ascending key value. There are other ways to traversea tree, as we'll see later.**

If traversal visits all of the tree nodes, it is called Enumeration (or we say the tree has been **Enumerated**).

Keys

One data item in an object is usually designated a key value. This value is used to search for the item or perform other operations on it.

Tree ADT



Binary Tree

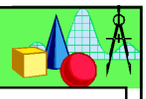
Made up of a finite set of nodes that is either empty or consists of a node called the root together with two binary trees, called the left and right subtrees, which are disjoint from each other and from the root.

Full Binary Tree

A binary tree where each node is either a leaf or is an internal node with exactly two non-empty children.

That means, in full binary trees, a node is allowed to have either none or two children (but not one!)

Tree ADT



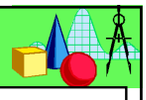
Complete Binary Trees

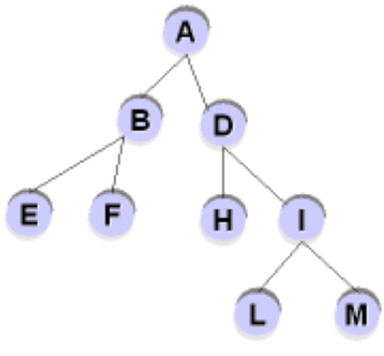
- A binary tree whereby if the height is d, and all levels, except possibly level d, are completely full.
- If the bottom level is incomplete, then it has all nodes to the left side.

  That is the tree has been filled in the level order from left to right (i.e. Top to Bottom and then Left to Right).

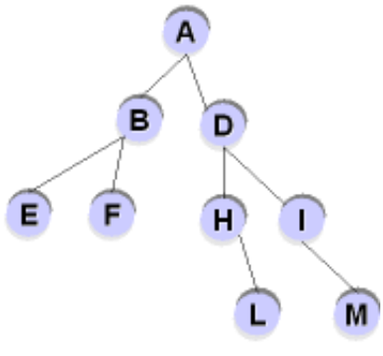
15

Tree ADT



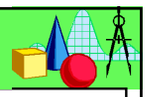


Full Binary Tree (Not Complete)



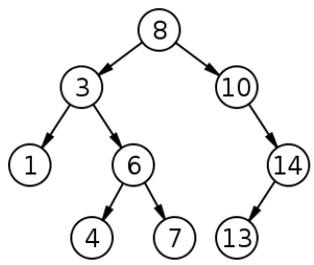
Not Complete (Nor Full) Binary Tree

16



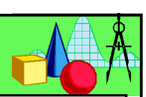
Binary Search Trees

(BST)



17

Binary Search Tree



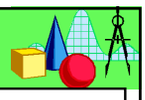
BST Property

All elements stored in the left subtree of a node whose value is ***K*** have values less than ***K***. All elements stored in the right subtree of a node whose value is ***K*** have values greater than or equal to ***K***.

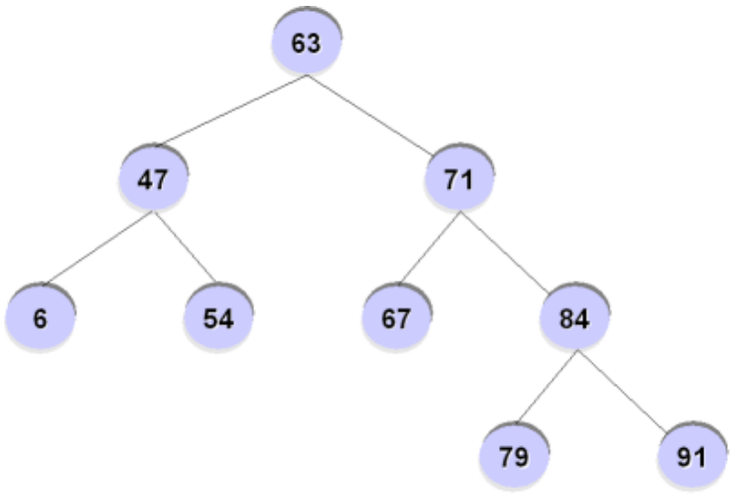
That is, a node's left child must have a key less than its parent, and a node's right child must have a key greater than its parent.

18

Binary Search Tree

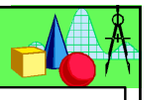


Example:



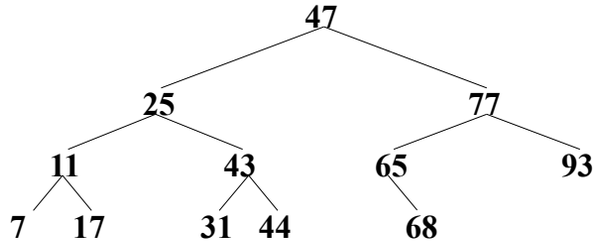
19

Trees



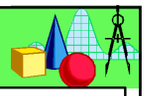
Binary Search Tree

- Values in left subtree less than parent node
- Values in right subtree greater than parent
  - Generally does not allow duplicate values (good way to remove them)
- Fast searches, $\log_2 n$ comparisons for a balanced tree



20

Trees



Inserting nodes into BST

- Use recursive function
- Begin at root
- If current node empty, insert new node here (base case)
- Otherwise,
  - If value > node, insert into right subtree
  - If value < node, insert into left subtree
  - If neither > nor <, must be =

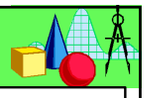
    Ignore duplicate?

Two Definitions:

1. **Purist** : No duplicates
2. **Operational** : >= on one side; < on other side

21

Trees



Tree traversals

In-order (print tree values from least to greatest)

Traverse left subtree (call function again)

Process node

Traverse right subtree

Preorder

Process node

Traverse left subtree

Traverse right subtree

Postorder

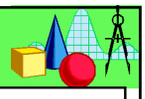
Traverse left subtree

Traverse right subtree

Process node

22

Binary Search Trees



All elements stored in the **left** subtree of a node, whose value is ***K,*** have values less than ***K***.

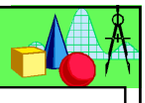
All elements stored in the **right** subtree of a node, whose value is ***K,*** have values greater than ***K***.

That is, a node's left child must have a key less than its parent, and a node's right child must have a key greater than its parent.

Key duplication is generally not permitted.

(Purist *versus* Operational definitions)

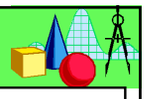
Binary Search Tree



Operations on BST ADT

- **Create a BST**
- **Insert an element**
- **Remove an element**
- **Find an element**
- **Clear (remove all elements)**
- **Display all elements in a sorted order**
- ***Etc.***

Binary Search Tree



Insert Algorithm

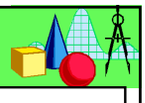
(New node is ALWAYS added as a new leaf in the tree)

If value we want to insert is less than key of current node, we have to go into the left subtree

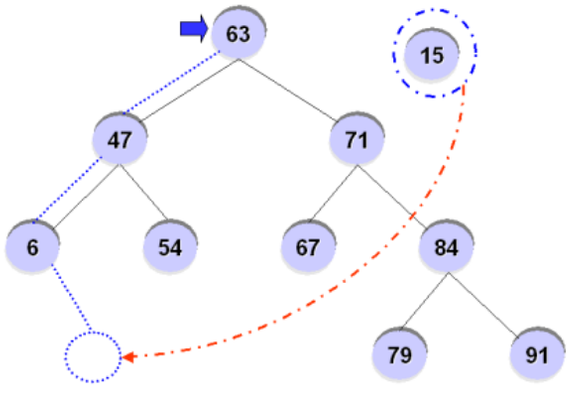
Otherwise, we have to go into the right subtree

If the current node is empty (not existing) create a new node with the value we are inserting and place it here.

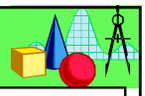
Binary Search Tree



For example, inserting '15' into the BST?



Binary Search Tree

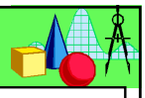


Insert Algorithm

Example: Graphically insert the following data:
5, 49, 61, 28, 23, 60, 70, 2, 7, 31, 3, 8 and 20
into a BST

27

Binary Search Tree



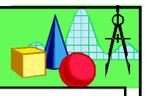
Delete Algorithm

How do we delete a node from BST?

Similar to the insert function, after deletion of a node, **the properties of the BST MUST be maintained**.

28

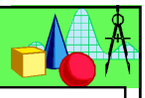
Binary Search Tree



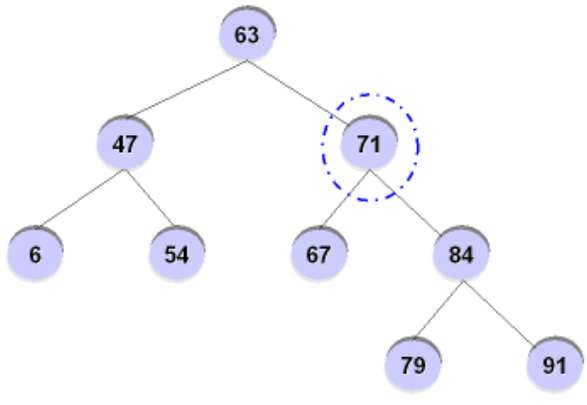
There are 3 possible cases

- Node to be deleted has no children
  → **We just delete the node.**
- Node to be deleted has only one child
  → Replace the node with its child and make the parent of the deleted node to be a parent of the child of the deleted node
- Node to be deleted has two children
  → Replace the node with **the smallest node in the RIGHT sub-tree OR the largest node in the LEFT sub-tree**.

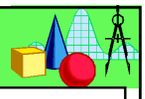
Binary Search Tree



Node to be deleted has two children



Binary Search Tree

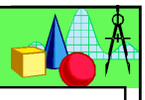


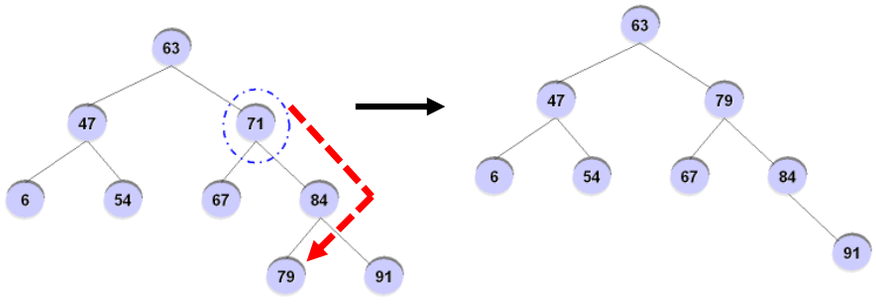
Node to be deleted has two children

Steps:

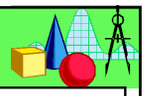
- **Find minimum value of right subtree.**
- **Delete minimum node of right subtree but keep its value.**
- **Replace the value of the node to be deleted by the minimum value whose node was deleted earlier.**

Binary Search Tree





Binary Search Tree



Inorder traversal

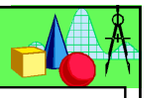
Visit the left subtree, then visit (process) the node, then visit the right subtree.

Algorithm:

1. If there is a left child inorder its sub-tree
2. Visit the node itself
3. If there is a right child inorder its sub-tree

33

Binary Search Tree



Postorder traversal

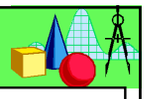
Visit each node after visiting its children.

Algorithm:

1. If there is a left child postorder its sub-tree
2. If there is a right child postorder its sub-tree
3. Visit the node itself

34

Binary Search Tree



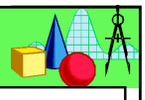
Preorder traversal

Visit each node then visit its children.

Algorithm:

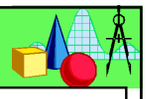
1. Visit the node itself
2. If there is a left child preorder its sub-tree
3. If there is a right child preorder its sub-tree

Binary Search Tree



```
class BSTNode
    {
        private:
            string  data;                   // Node value

        public:
            BSTNode   *left;                // pointer to left subtree
            BSTNode   *right;               // pointer to right subtree

            BSTNode ( string val )  {data = val; left = right= 0; }
            string getData ( ) { return data; }
            void setData ( string val )  { data = val; }
    };
```

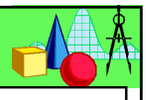
Binary Search Tree



```
class BSTree
  {
        private:
        BSTNode* root;

                  // Utility functions
        void    inOrderHelper(BSTNode * );
        void    preOrderHelper(BSTNode * );
        void    postOrderHelper(BSTNode* );
        void    insertHelper(BSTNode*, string );
        void    deleteNode(BSTNode* );
        BSTNode* findNode ( string );
        BSTNode* getMinimum (BSTNode * );
        BSTNode* getSuccessor (BSTNode * );
        BSTNode* getParent (BSTNode * );
```


37

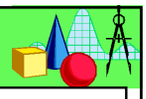
Binary Search Tree



```
        public:
        // Constructor function
        BST ( ) { root = NULL;}

        // Mutator functions
        void remove ( string );
        void insert ( string );              // Non-recursive method
        void newInsert ( string );           // Recursive method

        // Accessor functions
        void inOrder();
        void preOrder();
        void postOrder();
  };
```


38

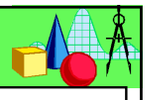
Binary Search Tree



```
void BSTree::insert( string val )
    {
        // Create the new node with the data
        BSTNode* newnode = new BSTNode ( val );

        if (root == NULL)          // Tree is empty – new node is first node
            {
                root = newnode;
                return;
            }
        else                       // Tree is NOT empty
            {
                BSTNode* parent = NULL;
                BSTNode* current = root;
```


39

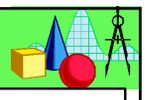
Binary Search Tree



```
        while (current != NULL)
            {
                parent = current;
                if ( newnode -> getData ( )  <  current -> getData ( ) )
                        current = current -> left;
                else
                        current = current -> right;
            }

          if ( newnode -> getData ( )  <  parent -> getData ( ) )
                 parent -> left = newnode;
          else
                parent -> right = newnode;
        }
    }     //End insert function
```


40

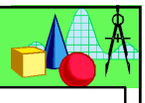
Binary Search Tree



```
void BSTree::newInsert( string val )
    {
            root = insertHelper(root, val);
    }
```


41

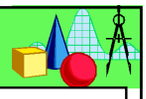
Binary Search Tree



```
BSTNode* BSTree::insertHelper ( BSTNode* ptr, string val  )
    {
       if ( ptr == NULL )          // The tree is empty - new first node!
            ptr = new BSTNode(val);
       else                                  // choose which sub-tree to follow
          {
            if (val >= ptr->getData())  // insert in right subtree – Op/defn.
                ptr->right = insertHelper ( ptr->right, val );
             else
                ptr->left = insertHelper ( ptr->left, val );
          }
       return ptr;
    }
```

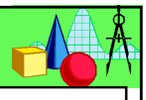

42

Binary Search Tree



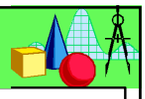
```
void BST::inOrder ( )
    {
        inOrderHelper ( root );
    }

void BST::inOrderHelper ( BSTNode* ptr )
    {
         if ( ptr != NULL )
             {
                 inOrderHelper ( ptr -> left );
                 cout << " " << ptr -> getData ( ) << "  ";
                 inOrderHelper ( ptr->right );
             }
         else return;
    }  //End inOrderHelper function
```


Binary Search Tree



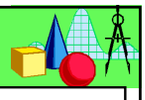
```
void BST::remove (string str )
    {
        BSTNode* markedBSTNode = findBSTNode ( str );
        deleteBSTNode(markedBSTNode);
    }
```

Binary Search Tree



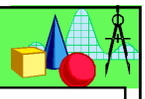
```
BSTNode* BST::findBSTNode ( string data )
    {
        BSTNode* ptr = root;

        while ( ptr != NULL )
            {
                if  ( ptr->getData ( ) == data )      // Found it!
                    return ptr;
                else if  (data < ptr -> getData ( ) )
                        ptr = ptr -> left;
                    else
                        ptr = ptr -> right;
            }
          return NULL;        // Tree is empty OR did not find it
       }
```


Binary Search Tree



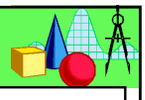
```
BSTNode*  BSTree::deleteNode(BSTNode* ptr, string val)
  {
      if ( ptr == NULL)          // Not found!
              return NULL;


      // Otherwise
      BSTNode* successor;
       if ( val > ptr->getData() )       // Search in Right subtree
              ptr->right = deleteNode(ptr->right, val);
      else if ( val < ptr->getData() )        // Search in Left subtree
               ptr->left = deleteNode(ptr->left, val);


  ...
```

Binary Search Tree



```
...
        else   //Node to to be deleted is found
           {
               if ( ptr->right != NULL)
                  {  // delete its inorder succesor
                        successor = ptr->right;
                        while ( successor->left != NULL)
                              successor = successor->left;

                        // Copy data from successor into ptr
                        ptr->copyData( successor );

...
```


Binary Search Tree

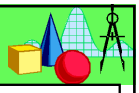


```
...
                  // Find successor now and delete it
                  ptr->right = deleteNode( ptr->right, val );
               }
            else
               return (ptr->left);
         }
   return ( ptr );
}
```

What was done?

1. Locate the node to be deleted.
2. If we find that node, then we proceed to find the node which will now succeed it.
3. Copy the data from the successor node into the node to be deleted.
4. Delete the successor node.

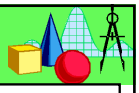
COMP2611 – Data Structures



DICTIONARIES

1

Balancing Trees



Definition

A **Perfectly Balanced** tree is one where the sub-trees of each node are of the same height.

A **Not Perfectly Balanced** tree is one where, for at least one node, the heights of the sub-trees differ by **at most 1**

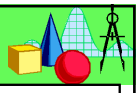
In order to balance binary trees, nodes must be re-positioned without altering the character of the tree.

Two Methods:

1. **Right Rotation**
2. **Left Rotation**

2

Balancing Trees



Right Rotation about a node

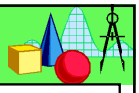
Algorithm:

1. Move left child into Node's position
2. Move right child of Node's former left child to become the Node's new left child.
3. Move Node to become the right-child of the Node's former left child

NB Right rotation can only be done on a node with a left child.

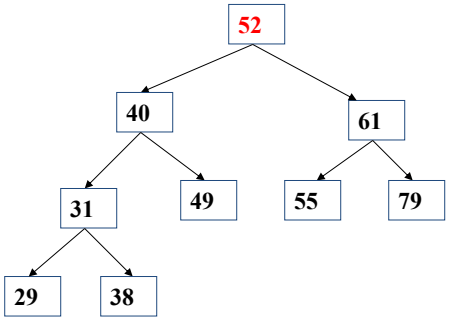
3

Balancing Trees



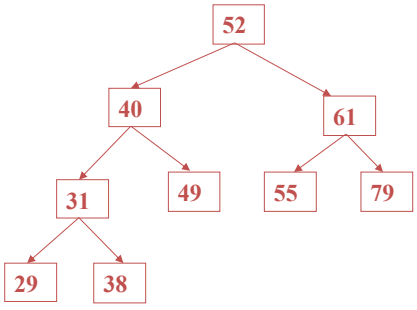
52 is our "Node"

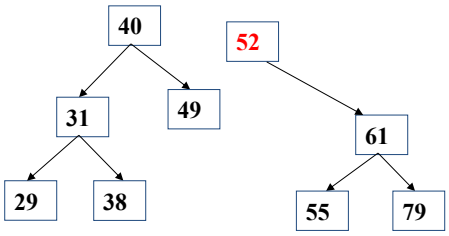
Right Rotate about 52



4

Balancing Trees

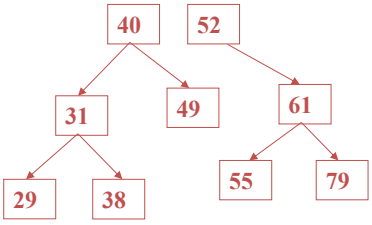


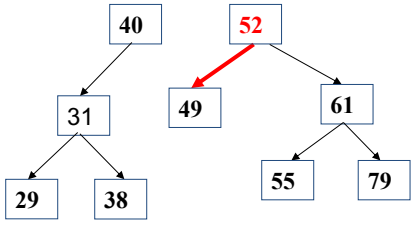


Node's Left-child is elevated to Node's position.

5

Balancing Trees

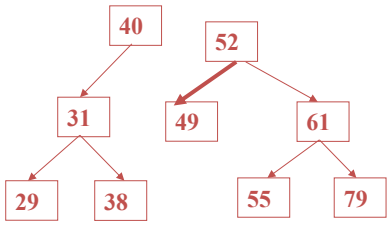


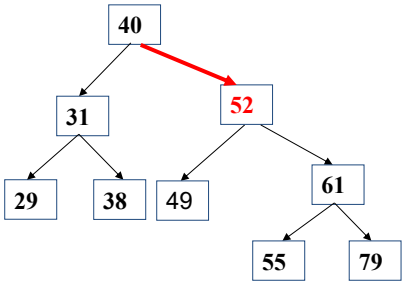


Right child of Node's former left child becomes Node's Left-Child.

6

Balancing Trees



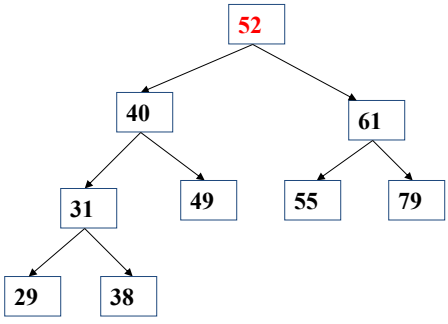


Node becomes former left child's Right-Child.

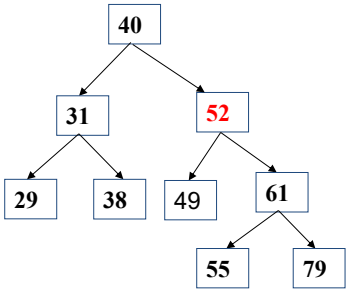
7

Balancing Trees

Right-Rotating 52



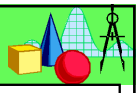
Before rotation



After rotation

8

Balancing Trees



Left Rotation about a node

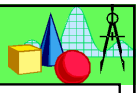
Algorithm:

1. Move right child into Node's position
2. Move left child of Node's former right child to become the Node's new right child.
3. Move Node to become the left-child of Node's former right child.

NB Left rotation can only be done on a node with a right child.

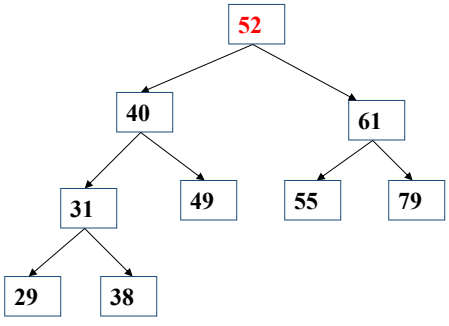
9

Balancing Trees



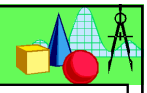
52 is our "Node"

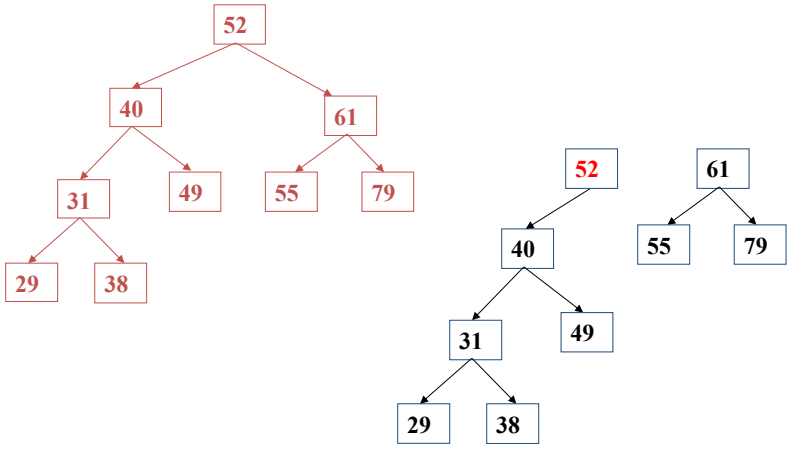
Left Rotate about 52



10

Balancing Trees

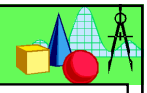


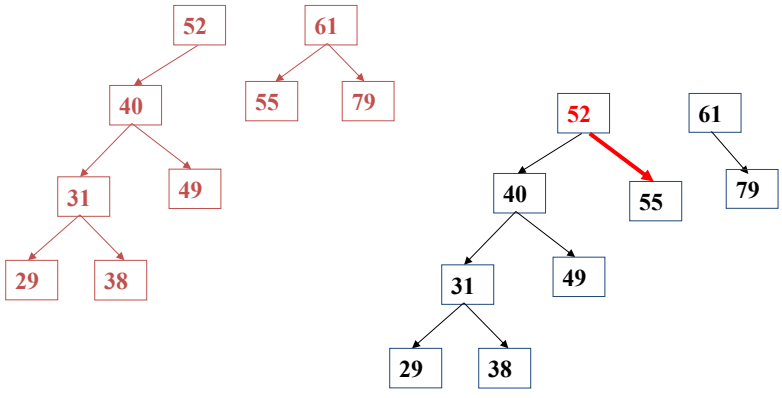


Node's Right-child is elevated to Node's position.

11

Balancing Trees

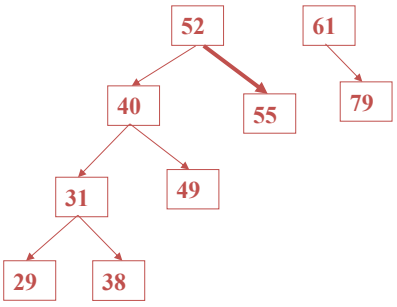


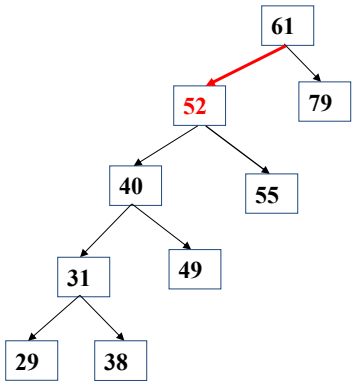


Left child of Node's former Right child becomes Node's Right-Child.

12

Balancing Trees





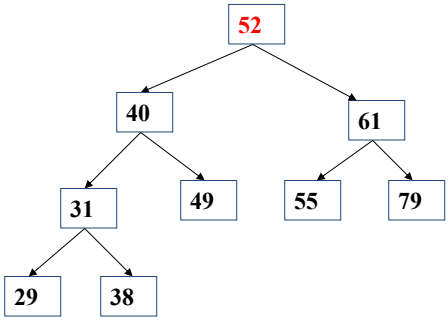
Node becomes former Right child's Left-Child.

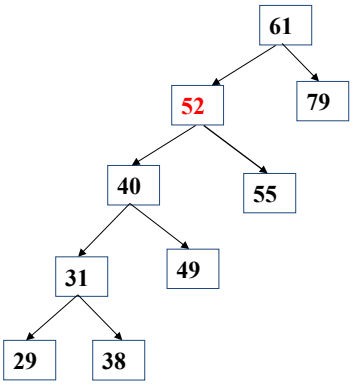
University of the West Indies

13

Balancing Trees

Left-Rotating 52

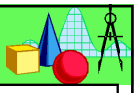




Before rotation **After rotation**

University of the West Indies

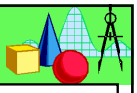
14



AVL TREES

15

AVL Trees



A balanced binary search tree.

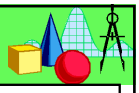
Named after their inventors, **A**delson-**V**elskii and **L**andis,

First dynamically balanced trees to be proposed.

Not perfectly balanced - **pairs of sibling sub-trees may differ in height by at most 1**

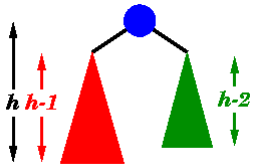
16

AVL Trees



Definition:

An AVL tree is a binary search tree which has the following properties:



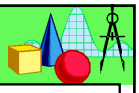
1. The sub-trees of every node differ in height by at most one.

2. Every sub-tree is an AVL tree.

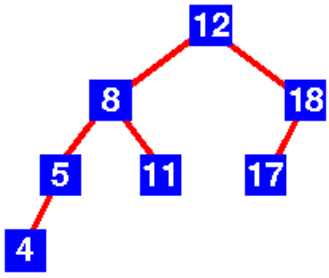
You need to be careful with this definition:
It permits some apparently unbalanced trees!

For example:

AVL Trees



AVL tree?

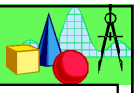


Yes

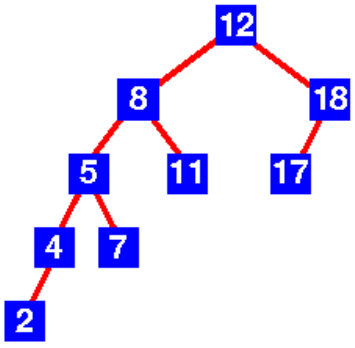
Examination shows that e*ach* left sub-tree has a height 1 greater than each right sub-tree.

Note: A single violation could render the entire tree unbalanced!

AVL Trees



AVL tree?



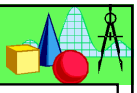
No

Sub-tree with root 8 has height 4 and sub- tree with root 18 has height 2.

AVL violation at nodes 12 and 8

19

AVL Trees



Nodes Insertion and Deletion

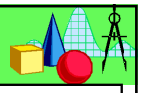
- Somewhat more complex than the ordinary BST.
- Requires extra attribute, called the **balance factor** to each node.
- Balancing factor indicates whether the tree is ***left-heavy***, ***balanced*** or ***right-heavy***.

Heavy **= height of sub-tree is 1 greater than the opposite sub-tree.**

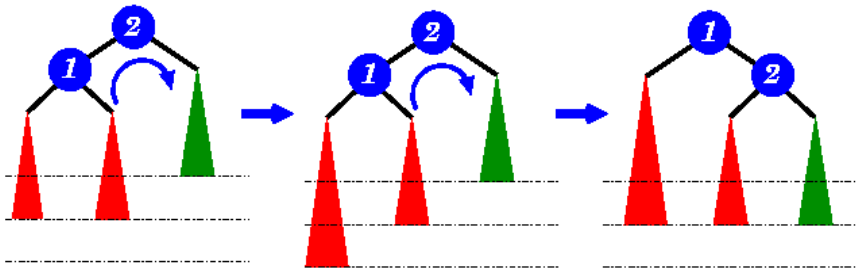
If the balance would be destroyed by an insertion or deletion, a rotation is performed to correct the balance.

20

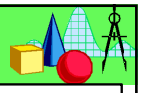
AVL Trees



Example:

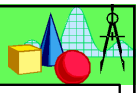


A new item added to the left subtree of node 1, causes its height to become 2 greater than 2's right sub-tree (shown in green). A right-rotation is performed to correct the imbalance.



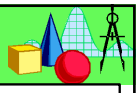
AVL TREES

AVL Tree



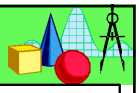
```
class AVLNode
  {
        private:
              int data;
              int height;

            public:
               AVLNode*  left;
               AVLNode*  right;
               AVLNode ( ) { right = left = NULL;  height = 0; }
               AVLNode ( int va l) { data = val;  height = 0;  right = left = 0; }
               void setData ( int val ) { data = val; }
               void setHeight ( int ht ) { height = ht; }
               int getData ( ) { return data; }
               int getHeight ( ) { return height;}
    };
```


AVL Tree



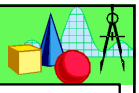
```
class AVLTree                          // To Start… from BST…
  {
        private:
              AVLNode *root;
              void insertHelper(AVLNode **,  int );
              string preOrderHelper( AVLNode * ) const;
              string inOrderHelper( AVLNode * ) const;
              string postOrderHelper( AVLNode * ) const;
              void deleteNode(AVLNode *,   int );
        public:
              AVLTree(){ root = 0; };
              void insert( int );
              string preOrder ( ) const;
              string inOrder ( ) const;
              string postOrder ( ) const;
              string remove ( int );
              void removeMin();
  };
```

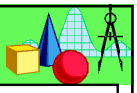
AVL Tree



```
class AVLTree                          // New stuff…To be added…
   {
        private:
             AVLNode*   rotateRight ( AVLNode*  );
             AVLNode*   rotateLeft ( AVLNode*  );
             AVLNode*   rotateDoubleRight ( AVLNode*  );
             AVLNode*   rotateDoubleLeft ( AVLNode*  );
             AVLNode*   rotateLeftRight ( AVLNode*  );
             AVLNode*   rotateRightLeft ( AVLNode*  );
             int   calcHeight ( AVLNode*  );
             int   calcBalance ( AVLNode*  );

        public:

   };
```


AVL Tree

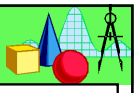




RED-BLACK TREES

27

Red-Black Trees



Definition

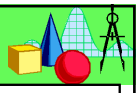
A binary search tree with one extra attribute for each node: the ***colour***, which is either **red** or **black**. We also need to keep track of the parent of each node, so that a red-black tree's node structure would be maintained

Properties:

1. Every node is either **red** or **black**.
2. Every leaf (NULL) is **black**.
3. The root node is **black**.
4. If a node is **red**, then both its children must be **black**.
5. Every simple path from a "root node" to a descendant leaf contains the same number of **black** nodes.

28

Red-Black Trees



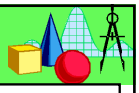
Properties:

All external nodes have the same black depth, which is defined as the number of black ancestors minus 1.

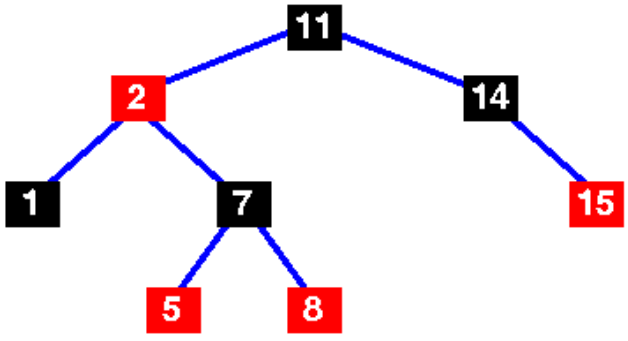
Implies that on any path from the root to a leaf, red nodes cannot be parents or children of other red nodes. However, any number of black nodes may appear in a sequence.

29

Red-Black Trees



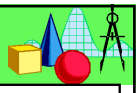
A basic red-black tree



Note colour of leaves may appear to violate properties definition

30

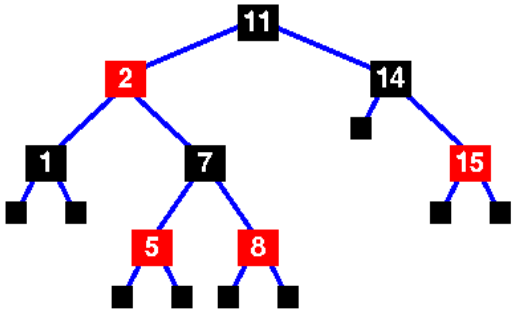
Red-Black Trees



Basic red-black tree with the **sentinel** nodes added.

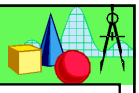
Implementations of the red-black tree algorithms will usually include the sentinel nodes as a convenient means of flagging that you have reached a leaf node.

They are the NULL black nodes of property definition.



31

Red-Black Trees



The number of black nodes on any path from, **but not including, a node *x*** to a leaf is called the ***black-height*** of that node.

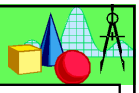
Additions and deletions from red-black trees **could** destroy the red-black property, so rotations are normally required to restore it.

Maintaining a red-black tree as new nodes are added or deleted primarily involves recoloring and rotation.

Algorithm for ADDITION is as follows:

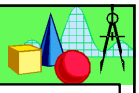
32

Red-Black Trees



1. **Create a new node n to hold the value.**
2. **If the tree is empty, make n the root. Otherwise, go left or right, as with normal insertion into a binary search tree.**
3. **If you pass through a node m with both its children red, colour those children black and re-colour m to red.**
4. **At the appropriate leaf, add n as a red child to its parent.**
6. **If either of the steps that adds red links creates 2 consecutive red nodes, colour the first red node black and it's parent red then rotate about the parent node to create a black node with 2 red children.**
7. **If the root is not black, re-colour it black.**

Red-Black Trees



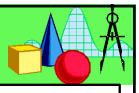
Example

Enter the following keys into a Red-black tree: 1, 2, 3, 4, 5, 6, 7, 8

Recall rules:

1. Enter nodes as in a normal BST – new nodes are coloured **RED**.
2. If you pass a **Black** node with **TWO RED** children, the **Black node** should be re-coloured **RED** and its two RED children re-coloured **BLACK**.
3. After the node is added, if there are two consecutive **Red nodes,** re-colour the **Red** parent to **Black** and the grand-parent **Red.** If the parent is a right-child of the grand-parent, perform a left-rotation with the grand-parent**.** *(If left-child, perform right-rotation).After the re-colouring and rotation, this will result in a sub-tree with a* ***Black sub-root*** *and two Red children.*
4. If needed, repeat step 3.
5. Finally, the root must be coloured **Black.**

Red-Black Trees



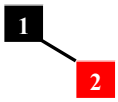
Example: Enter the following keys into a Red-black tree: 1, 2, 3, 4, 5, 6, 7, 8

SOLUTION

Enter key 1

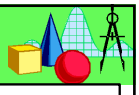
New node entered as RED but changed to black because it is the root.

Add key 2:



Nothing else to do here – the RB tree is fine

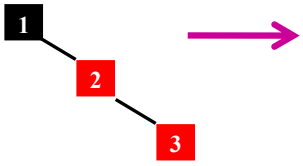
Red-Black Trees

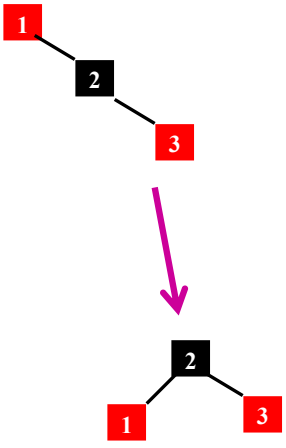


Example: Enter the following keys into a Red-black tree: 1, 2, 3, 4, 5, 6, 7, 8

SOLUTION

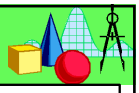
Add key 3:





Two consecutive Red nodes with 2 and 3. We must recolour the parent (2) BLACK and the grand-parent black node (1) RED and rotate the grand-parent (1).

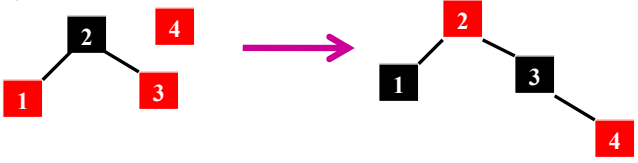
Red-Black Trees



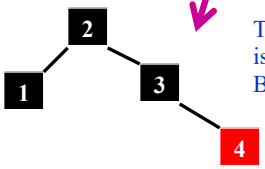
Example: Enter the following keys into a Red-black tree: 1, 2, 3, **4**, 5, 6, 7, 8

SOLUTION

Add key 4:



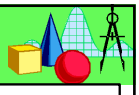
Passing a Black node (2) with two red children (1 and 3), means we must change the black node (2) RED and its two children (1 and 3) to BLACK.



The root node (2) is then reset to BLACK.

37

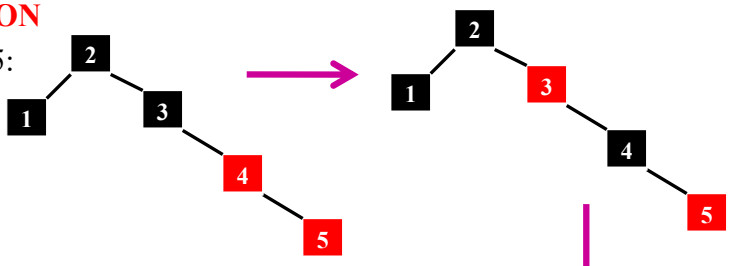
Red-Black Trees



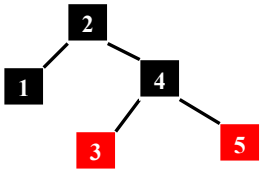
Example: Enter the following keys into a Red-black tree: 1, 2, 3, 4, **5**, 6, 7, 8

SOLUTION

Add key 5:

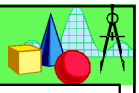


Two consecutive Red nodes with 4 and 5. We must rotate their parent (3) and recolour the parent (3) RED and the first red node (4) black.



38

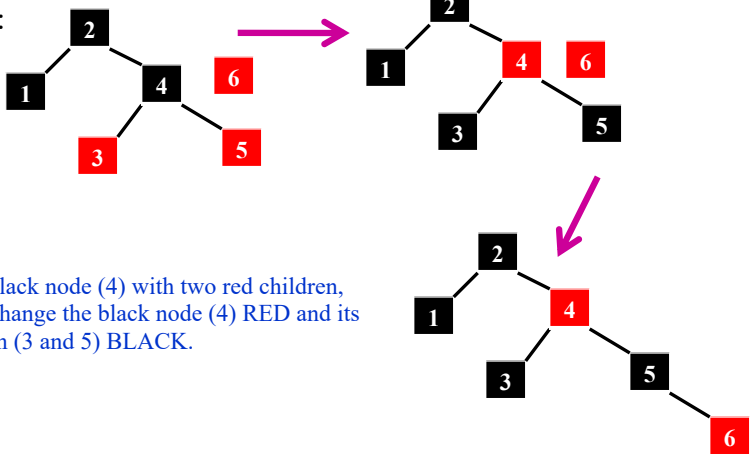
Red-Black Trees



Example: Enter the following keys into a Red-black tree: 1, 2, 3, 4, 5, 6, 7, 8

SOLUTION

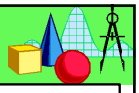
Add key 6:



Passing a Black node (4) with two red children, means we change the black node (4) RED and its two children (3 and 5) BLACK.

39

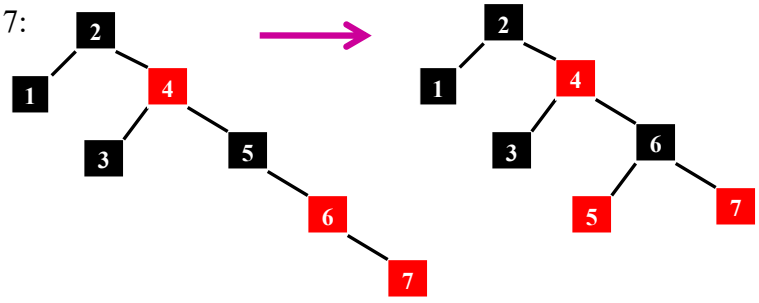
Red-Black Trees



Example: Enter the following keys into a Red-black tree: 1, 2, 3, 4, 5, 6, 7, 8

SOLUTION

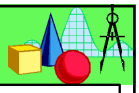
Add key 7:



Two consecutive Red nodes with 6 and 7. We must rotate their parent (5) and recolour the parent (5) RED and the first red node (6) black.

40

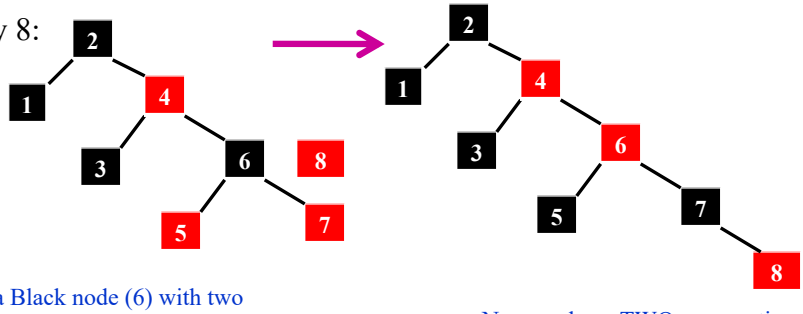
Red-Black Trees



Example: Enter the following keys into a Red-black tree: 1, 2, 3, 4, 5, 6, 7, 8

SOLUTION

Add key 8:

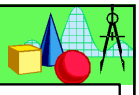


Passing a Black node (6) with two red children, means we change the black node (6) RED and its two children (5 and 7) BLACK.

Now we have TWO consecutive Red nodes with 4 and 6. We must recolour 4 (black) and 2 (red) then rotate the grand-parent (2).

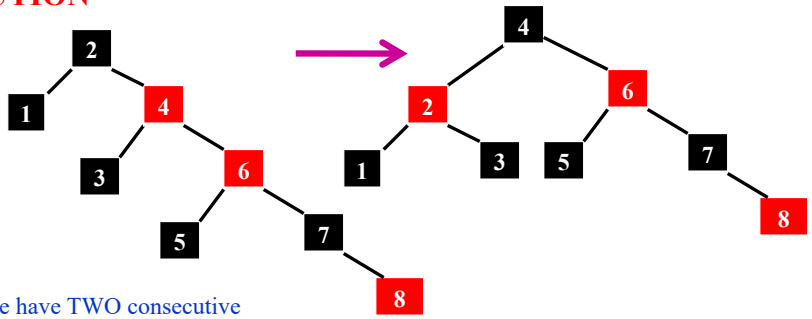
41

Red-Black Trees



Example: Enter the following keys into a Red-black tree: 1, 2, 3, 4, 5, 6, 7, 8

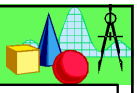
SOLUTION



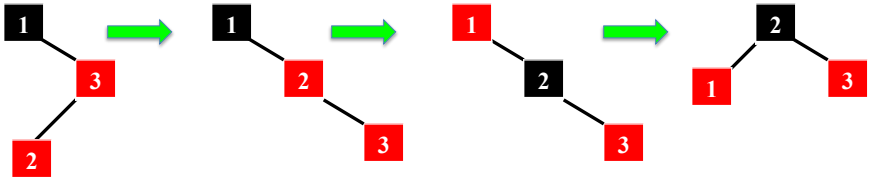
Now we have TWO consecutive Red nodes with 4 and 6. We must recolour 4 (black) and 2 (red) then rotate the grand-parent (2).

42

Red-Black Trees



Handling Special Cases…



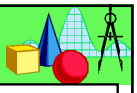
Step 1: Special Case

Step 2: Regularize the special case

Step 3: Perform re-colouring

Step 4: Perform rotation

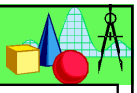
Red-Black Trees



Delete A Node:

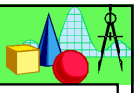
1. Nodes are deleted as defined by the Binary Search Tree

2. Colour of deleted node is retained for its replacement.



SPLAY TREES

Splay Trees



Self-adjusting binary search trees.

Represents an attempt to combine the benefits of an AVL tree with a simple random binary search tree.

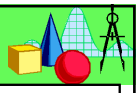
AVL trees are always balanced. However, this means continually keeping track of the height of the sub-trees at each node.

Splay tree does *not* store height data.

Instead every time we access a node, we move the accessed node to the root using AVL rotations (insertions may also be rotated to the root).

Sound a bit crazy…

Splay Trees



…In practice it will result in a tree which is NOT well balanced, but a tree with very fast access times.

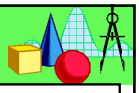
Highly unbalanced trees may be formed from the insertions, but each access will tend to move towards fixing the problem.

Benefits:

Given a good splaying algorithm, search times will be fast. As there is no work in storing heights a splay tree can approach (and sometimes improve on) an AVL tree.

Nodes that are frequently accessed will be very near to the root. This means for applications that frequently access the same nodes, search times will be especially fast.

Splay Trees



How to Splay?

The simplest approach is to rotate an accessed node all the way to the root using single rotations.

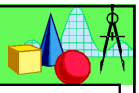
Problem:

Improves the situation for some nodes, others end up further away from the root.

We can obtain a better overall tree shape if we use double rotations as well.

How…

Splay Trees



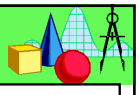
How to Splay?

If the accessed node is the inside node of its parent we use a double rotation (zig-zag or zag-zig), but if it is an outside node we use a new strategy again involving two single rotations (zig-zig or zag-zag) :

Zig = Right-Rotation

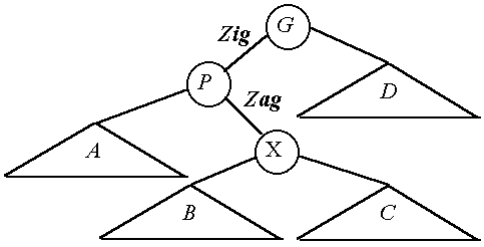
Zag = Left-Rotation

Splay Trees



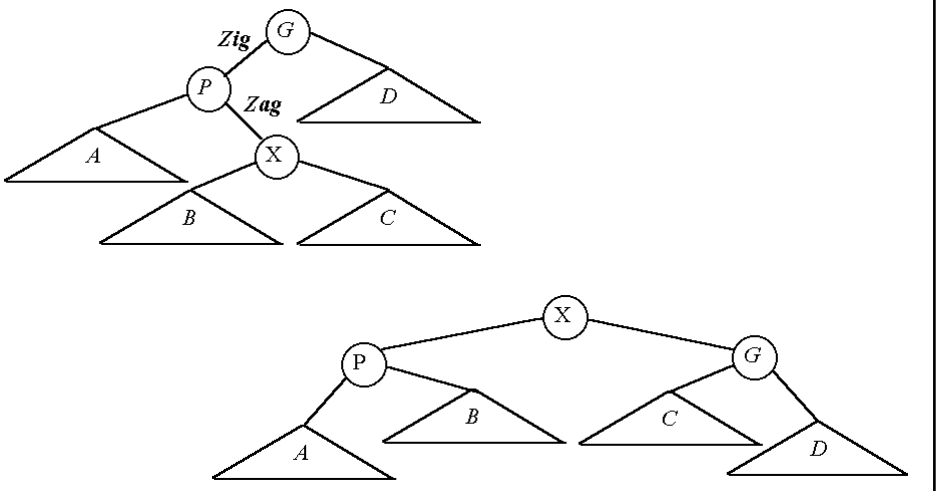
Example: Zig-Zag

X is our accessed node and we want to rotate it so that it becomes the root of the tree (or sub-tree as the case may be).



This is the same as a left right double rotation in an AVL tree.

Splay Trees



51

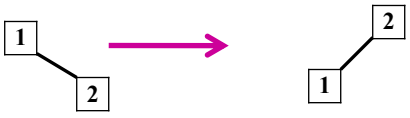
Splay Trees

EXAMPLE

Insert the following keys into a splay tree: 1, 2, 4, 3, 5

Solution:

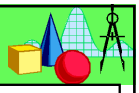
Insert 1: 

Insert 2: 

Insert 4: 

52

Splay Trees

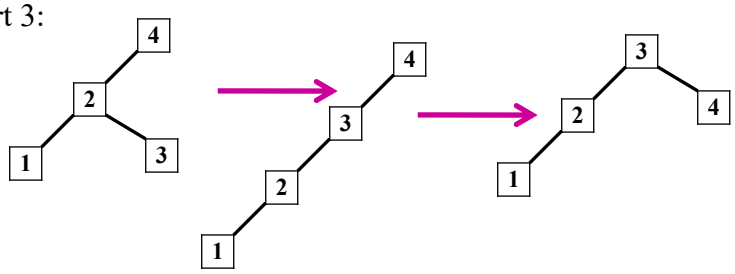


EXAMPLE

Insert the following keys into a splay tree: 1, 2, 4, 3, 5

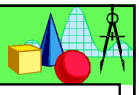
Solution:

Insert 3:



53

Splay Trees

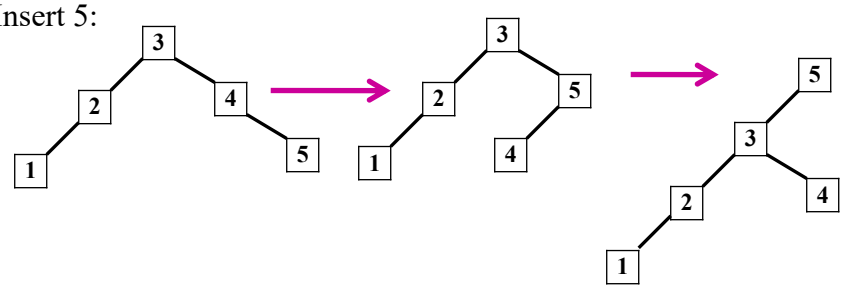


EXAMPLE

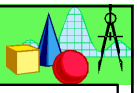
Insert the following keys into a splay tree: 1, 2, 4, 3, 5

Solution:

Insert 5:



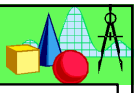
54



Binary Heap

1

Binary Heap



A special kind of binary tree. It has two properties that are not generally true for other trees:

Completeness

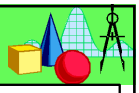
The tree is complete, which means that nodes are added from top to bottom, left to right, without leaving any spaces. A binary tree is completely full if it is of height, h, and has $2^{h+1}-1$ nodes.

Heapness

The item in the tree with the highest priority is at the top of the tree, and the same is true for every subtree.

2

Binary Heap



Binary tree of height, ***h***, is **complete *iff***

- It is empty

  or

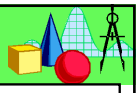
- its left subtree is complete of height ***h*-1** and its right subtree is completely full of height ***h*-2**

  or

- its left subtree is completely full of height ***h*-1** and its right subtree is complete of height ***h*-1**.

3

Binary Heap



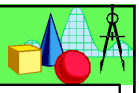
In simple terms:

A heap is a binary tree in which every parent is greater than its child(ren). This is referred to as a MaxHeap.

A **Reverse Heap** is one in which the rule is "**every parent is less than the child(ren)**". **This is referred to as a MinHeap.**

4

Binary Heap

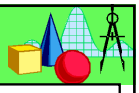


To build a heap, data is placed in the tree as it arrives.

Algorithm:

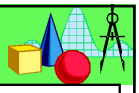
1. **add** a new node at the next 'logical' leaf position
2. **use** this new node as the current position
3. **While new data is greater than that in the parent of the current node (MaxHeap):**
   - **move** the parent down to the current node
   - **make** the parent (now vacant) the current node
   - **Place data** in current node

Binary Heap



- New nodes are always added on the deepest level in an orderly way.
- The tree never becomes unbalanced.
- There is no particular relationship among the data items in the nodes on any given level, even the ones that have the same parent
- A heap is not a sorted structure. Can be regarded as partially ordered.
- A given set of data can be formed into many different heaps (depends on the order in which the data arrives.)

Binary Heap

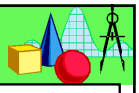


Example:

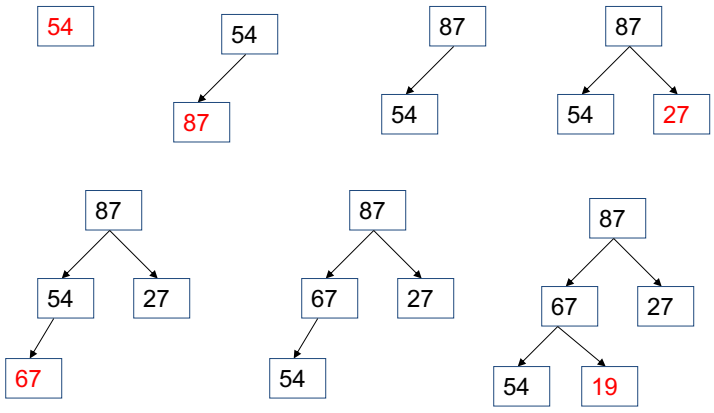
Data arrives to be heaped into a **MaxHeap** in the order:
54, 87, 27, 67, 19, 31, 29, 18, 32, 56, 7, 12, 31

7

Binary Heap

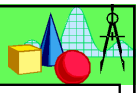


54, 87, 27, 67, 19, 31, 29, 18, 32, 56, 7, 12, 31

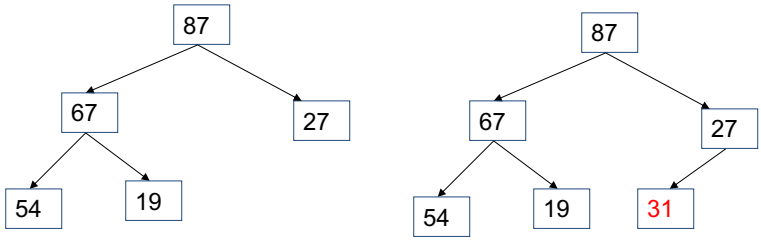


8

Binary Heap



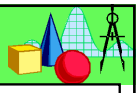
54, 87, 27, 67, 19, 31, 29, 18, 32, 56, 7, 12, 31



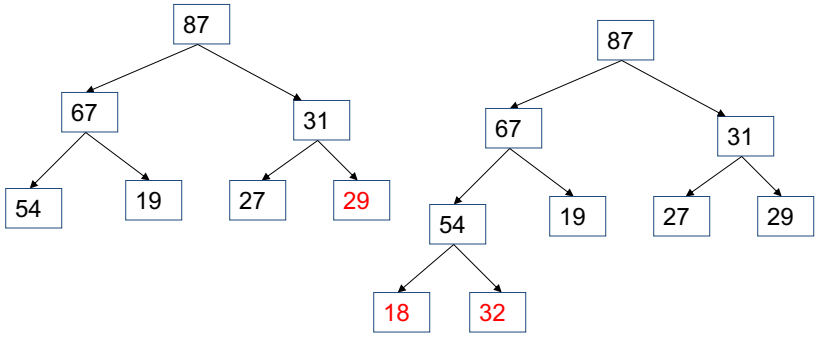
University of the West Indies

9

Binary Heap



54, 87, 27, 67, 19, 31, 29, 18, 32, 56, 7, 12, 31

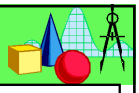


etc…

University of the West Indies

10

Binary Heap



To delete an element from the heap:

Algorithm:

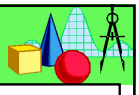
If node is the last "logical node" in the tree, simply delete it

Else:

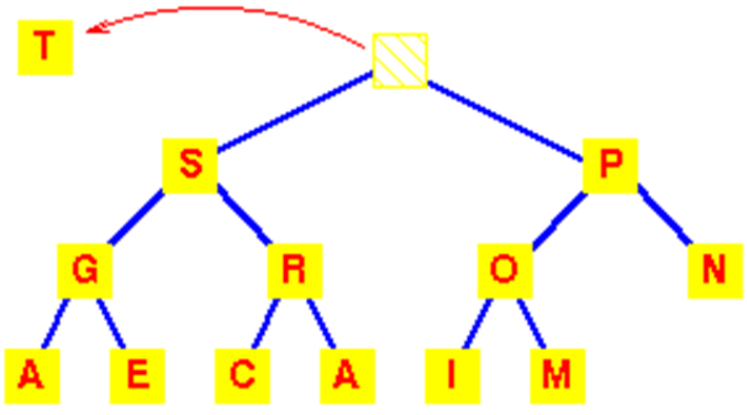
- **Replace the node with the last "logical node" in the tree**
- **Delete the last logical node from the tree**
- **Re-heapify**

11

Binary Heap

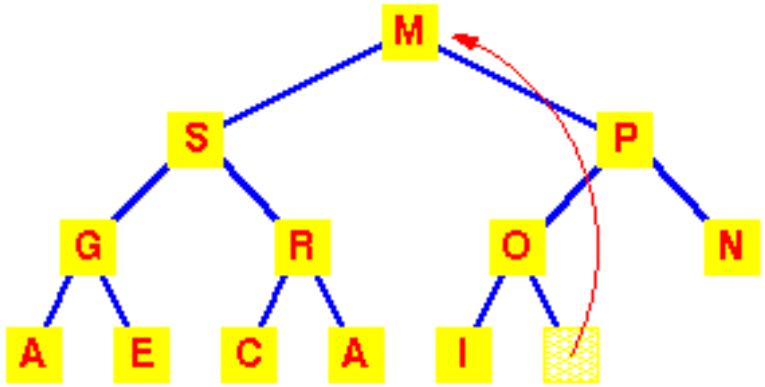


Example: **Deleting the root node, T**



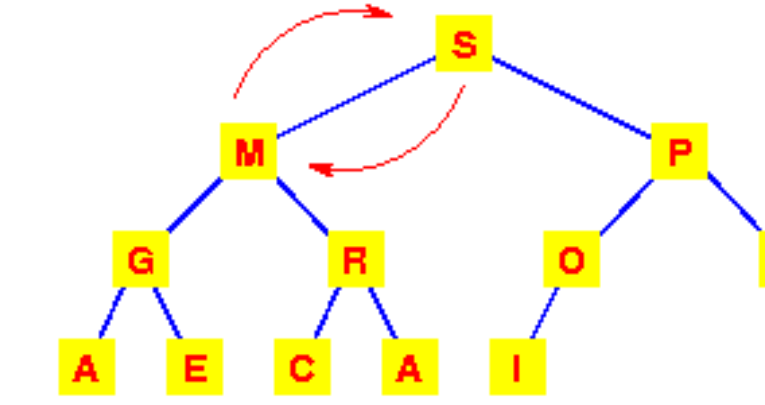
12

Binary Heap



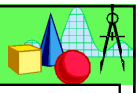
13

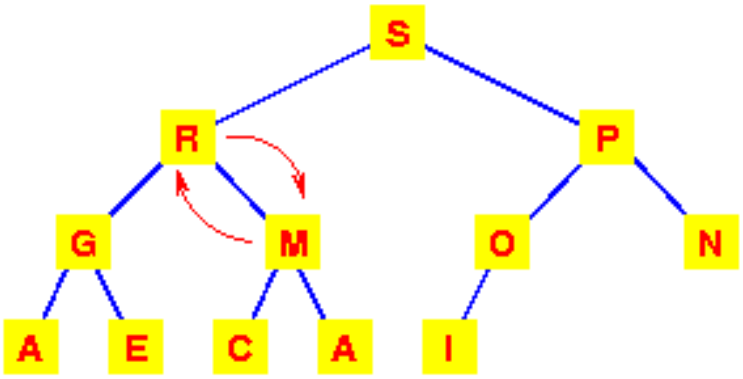
Binary Heap



14

Binary Heap

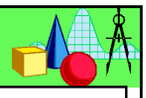




University of the West Indies

15

COMP2611 – Data Structures

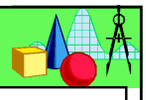


SET

Abstract Data Type

1

Set



Recall a data structure is a container for grouping a collection of data into a single object.

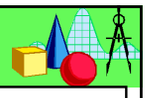
A set is an abstract data type on which such mathematical functions such as "***x is an element of***", "***set complement***", "***sets intersection***", etc can be performed.

An ordered set is therefore a collection of items with two defining properties:

1. **Ordering**
2. **Uniqueness**

2

Set



Ordering

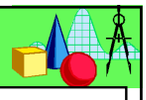
- The elements of the set have indices associated with them.
- These indices can be used to identify elements of the set.

Uniqueness

- No element appears in the set more than once.
- Adding an element which already exists in a set, has no effect.

3

Set



In addition, implementation of an ordered set will have the following property:

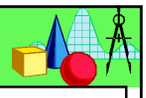
Arbitrary size

- As elements are added to the set, it expands to make room for new elements.
- To achieve uniqueness, the add function must search the set first to see if it already exists.

To make the set expand as elements are added, advantage can be taken of the resize function on vectors or use of a self referential structure.

4

Set

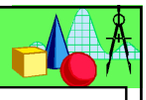


Beginning of a class definition for a Set.

```
class Set
    {
        private:
            vector <int> elements;     //data will be of type integer

        public:
            Set (void);                // Create an empty set
            Set (int n);               // Create a set with the initial value "n"

            int getNumElements () const;
            int getElement (int ) const;
            bool find (const int ) const;
            void add (const int );
            void remove (const int );
    };
```


5

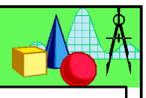
Set



```
Set::Set (void)
      {
          elements.resize(0);
      }

Set::Set (int n)
      {
          elements.resize (0);
          elements.push_back (n);
       }

int Set::getNumElements () const
      {
          return elements.size();
      }
```


6

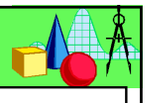
Set



```
int Set::getElement (int index) const
        {
           if (index <= elements.size ( ) )
                   return elements[index-1];
            else
                 {
                    cout << "Set index is out of range." << endl;
                    exit (1);
                 }
        }

bool Set:: find (const int  elem ) const
        {
           for (int x = 0;  x < elements.size( );  x++)
              if (elements[x] == elem)          // elem is found at subscript x
                    return true;                // Break from loop and return true
           return false;                        // elem is not found in elements
        }
```


7

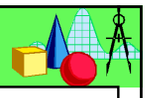
Set



```
void Set::add (const int elem)
        {
           // if the element is already in the set, simply return
           if ( find ( elem ) )
                   return;

           // else enter the new element at the end of the set
           else
                {
                   elements.push_back (elem);
                }
          return;
        }
```

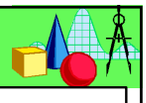

8

Set



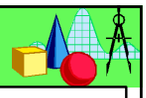
```
void Set::remove (const int elem)
        {
            int x;
           // search for elem – walk down to where it is
           for (x = 0; x < elements.size() && elements[x] != elem; x++);

           // was it found?
            if ( x == elements.size())                    //Didn't find it!
                  return;
            else          // found it at subscript x; slide the other elements to the left
                  for (int y = x; y < elements.size(); y++)
                        {
                           elements[y] = elements[y+1];
                        }
                  elements.resize( elements.size() -1);
           return;
      }
```


Sets - Intersection



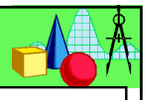
```
void Set::intersect(const Set *setA,  const Set *setB)
        {
            int x, y;

            elements.resize(0);
            for (x = 0;  x  <  setA->getNumElements();  x++)
                  for (y = 0;  y  <  setB->getNumElements();  y++)
                     {
                           if ( setA->elements[x] == setB->elements[y] )
                              {
                                    elements.push_back ( setA->elements[x] );
                              }
                     }
        }
```

Sets - Union



```
void Set::union(const Set *setA,  const Set *setB)
        {
                int x, y;
                elements.resize(0);
                elements = setA->elements;          //Copy everything from setA

                for (x = 0;  x < setB->numElements;  x++)
                    if ( SetA->find ( SetB->elements[x])  == false ) )
                        {
                            elements.push_back ( setB->elements[x] );
                        }
        }
```


Set



Disjoint Sets

If two sets A and B exist and A intersection B is the empty set, then A and B are described as **Disjoint Sets**.

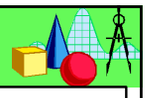
In other words: **Disjoint (A, B) = True** iff **A∩B = { }**

An immediate benefit of the disjoint-set data structure is its ability to perform very quick find and union operations.

Note

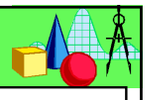
A disjoint-set data structure maintains a collection **S = {S1, S2,...,Sk}** of disjoint dynamic sets **S1, S2, ..., Sk**.

Set



```
void Set::disjoint ( const Set *setA )
        {
             for (int x = 0;  x < setA->numElements;  x++)
                {
                    if ( this->find ( SetA->elements[x])  == true ) )
                        {
                            return false;
                        }
                }
            return true;
        }
```


Set



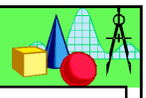
Weighted Union (or Union by Rank)

Union operations on two sets whereby the **smaller** set is "unioned" with the larger set.

Method has the benefit of performing fewer comparisons when members of the smaller set are compared with elements already in the "union set" than vice versa.

Using a general tree ADT to represent the sets, when the two trees are combined, the root of the smaller tree is simply made to point to the root of the larger tree.

Set

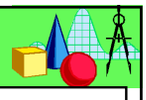


Path Compression

- Whenever a sequence of pointers is followed to locate the root of a tree, all the nodes along the way are updated to point directly to the root.
- Makes subsequent queries more efficient.
- Path compression is a two pass process:
  - First pass consists of finding the root.
  - Second pass updates the pointers that were followed in finding the root.

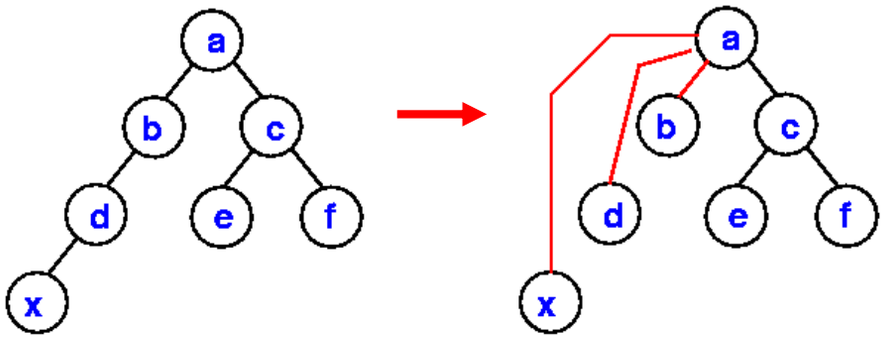
15

Set



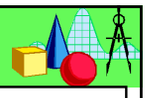
Path Compression

Example: **Find(x)**



16

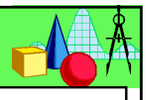
COMP2611 – Data Structures



Searching Techniques

1

Searching Techniques



Search algorithm:

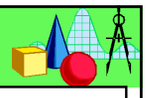
An algorithm that takes a problem as input and returns a solution to the problem, usually after evaluating a number of possible solutions.

Search Space.

The set of all possible solutions to a problem.

2

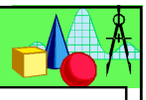
Searching Techniques



- Searching is the process of finding a target element within a group of items called the search pool
- The target may or may not be in the search pool
- The objective is to perform the search efficiently, minimizing the number of comparisons
- Two familiar classic searching approaches:
  - Linear or Sequential search
  - Binary search

3

Searching Techniques



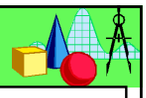
Linear Search:

- Each time a value is read from the list and it is not the "target" value, *only one item* from the list is eliminated.
- A linear search begins at one end of the search pool and examines each element consecutively.
- Eventually, either the item is found or the end of the search pool is encountered
- On average, it will search half of the pool

This algorithm is **O(n)** for finding a single element or determining it is not in the search pool.

4

Searching Techniques

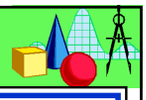


Binary Search

- Assumes the list of items is sorted.
- Each pass eliminates half of the search pool with one comparison.
- Search examines the middle element of the list - if the target is found, the search is over
- The process continues by comparing the target to the middle of the remaining ***viable candidates.***
- Eventually, the target is found or there are no remaining *viable candidates* (and the target has not been found)

5

Searching Techniques



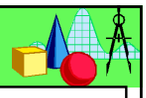
Binary Search

Function returns the index of the entry if the target value is found or -1 if it is not found

```
int binSearch(int list[], int target, int first, int last)
  {
    int result,  mid;

    if (first > last)
        result = -1;
    else
     {
       mid = (first + last)/2;

       if (target == list[mid])
          result = mid;
       else if (target < list[mid])
                result = binSearch(list, target, first, mid-1);
            else                  //(target > list[mid])
                result = search(list, target, mid+1, last);
     }

    return result;
  }
```

6

Searching Techniques



Example: Search for the number 33

target is 33
The array list looks like this:

Indexes	0	1	2	3	4	5	6	7	8	9
Contents	5	7	9	13	32	33	42	54	56	88

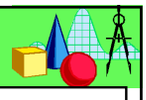
mid = (0 + 9) / 2 (which is 4)
33 > list[mid] (that is, 33 > list[4])
So, if 33 is in the array, then 33 is one of:

					5	6	7	8	9
					33	42	54	56	88

Eliminated half of the remaining elements from consideration because array elements are sorted.

7

Binary Search Example



The array list looks like this:

Indexes	0	1	2	3	4	5	6	7	8	9
Contents	5	7	9	13	32	33	42	54	56	88

mid = (5 + 9) / 2 (which is 7)
33 < list[mid] (that is, 33 < list[7])
So, if 33 is in the array, then 33 is one of:

					5	6			
					33	42			

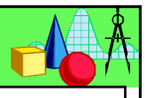
Eliminate half of the remaining elements

mid = (5 + 6) / 2 (which is 5)
33 == list[mid]
So we found 33 at index 5:

					5				
					33				

8

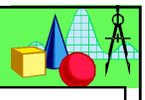
Searching Techniques



What are the real searching techniques used in Computer Science?

9

Searching Techniques



There are two kinds of search algorithm

Complete

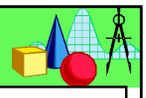
Guaranteed to find solution or prove there is none

Incomplete

- May not find a solution even when it exists.
- Often more efficient (or there would be no point).
- e.g. Genetic Algorithms.

10

Searching Techniques



Brute-force search algorithms

- Otherwise known as **naïve** or **uninformed**, algorithms.
- they use the simplest method of the searching through the search space

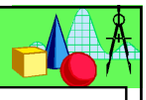
Uninformed search

- Does not take into account the specific nature of the problem.
- Can be implemented in general applications.
- Same implementation can be used in a wide range of problems due to abstraction.

The drawback is that most search spaces are extremely large, and an uninformed search (especially of a tree or graph) will take a reasonable amount of time ONLY for small examples. As such, to speed up the process, sometimes only an informed search will do.

11

Searching Techniques



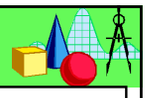
Informed search algorithms

- Use **heuristic functions** to apply knowledge about the structure of the **search space**
- To try to reduce the amount of time spent searching.

A **heuristic function** is a function which ranks alternatives in various search algorithms at each branching step of the search based on the available information in order to make a decision about which branch to follow during a search.

12

Searching Techniques

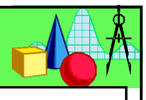


List search

- Most basic kind of search algorithm.
- Goal is to find one element of a set by some key (perhaps containing other information related to the key).
- As this is a common problem in computer science, the computational complexity of these algorithms has been well studied.
- Simplest such algorithm is linear search, which simply examines each element of the list in order.
- Has expensive O(n) running time, where *n* is the number of items in the list, but can be used directly on any unprocessed list.
- A more sophisticated list search algorithm is binary search; it runs in O(log *n*) time.
- BS is significantly better than linear search for large lists of data, but it requires that the list be sorted before searching.

13

Searching Techniques



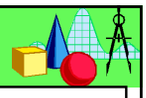
List search

- **Interpolation search** is better than binary search for large sorted lists with fairly even distributions (O(log (log *n*)) complexity), but has a worst-case running time of O(*n*).

- **Grover's algorithm** is a **quantum algorithm** that offers quadratic speedup over the classical linear search for unsorted lists. However, it requires a currently non-existent quantum computer on which to run.

14

Searching Techniques



Hash tables

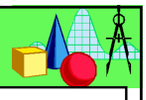
- Used for list search, requiring only constant time for search in the average case
- Requires more space overhead and has terrible O(n) worst-case search time.

Most list search algorithms, such as linear search and binary search, can be extended with little additional cost to find all values less than or greater than a given key, with an operation called ***range search***.

One such exception is hash tables, which cannot perform such a search efficiently.

15

Searching Techniques

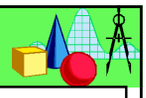


Tree Search

- The heart of searching techniques for structured data.
- These search trees of nodes, whether that tree is explicit or implicit (generated on the go).
- The basic principle is that a node is taken from a data structure, its successors examined and added to the data structure.
- By manipulating the data structure, the tree is explored in different orders
  - **Breadth-first search**
  - **Depth-first search**
  - **Iterative-deepening Depth-first search**
  - **Depth-limited search**
  - **Bidirectional search**
  - **Uniform-cost search**

16

Searching Techniques

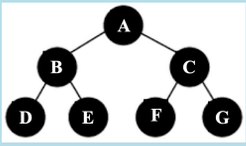


Breadth-First Search

Search begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.

Depth-First Search

Depth-first search (**DFS**) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) and **explores as far as possible along each branch before backtracking.**

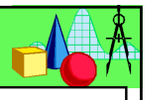


B.F.S. = A, B, C, D , E, F, G

D.F.S. = A, B, D, E, C, F, G

17

Searching Techniques

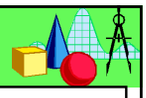


Depth-Limited Search

Depth-limited search is an algorithm to explore the vertices of a graph. It is a modification of depth-first search and is used for example in the iterative deepening depth-first search algorithm.

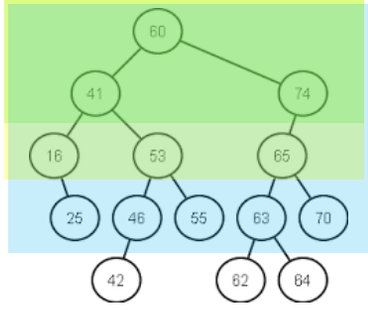
18

Searching Techniques



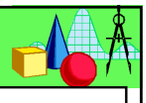
Iterative-Deepening Depth-First Search

- strategy in which a depth-limited search is run repeatedly, increasing the depth limit with each iteration until it reaches *d*, the depth of the shallowest goal state.
- On each iteration, IDDFS visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are first visited, assuming no pruning, it is effectively breadth-first.



19

Searching Techniques



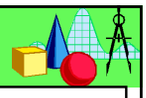
Bidirectional Search

A graph search algorithm that runs two simultaneous searches: one forward from the initial state, and one backward from the goal, and stopping when the two meet in the middle.

This does not come without a price: Aside from the complexity of searching two times in parallel, we have to decide which search tree to extend at each step; we have to be able to travel backwards from goal to initial state - which may not be possible without extra work; and we need an efficient way to find the intersection of the two search trees.

20

Searching Techniques

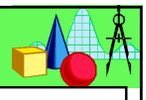


Uniform-Cost Search

- A tree search algorithm used for traversing or searching a weighted tree, tree structure, or graph.
- Intuitively, the search begins at the root node. The search continues by visiting the next node which has the least total cost from the root. Nodes are visited in this manner until a goal state is reached.
- Typically, algorithm involves expanding nodes by adding all unexpanded neighboring nodes that are connected by directed paths to a priority queue. In the queue, each node is associated with its total path cost from the root, where the least-cost paths are given highest priority.

21

Searching Techniques



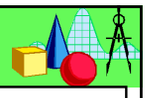
Tree Search

- Efficiency of a tree search (compared to other search methods) is **highly dependent upon the number** and **structure of nodes** in relation to the **number of items** on that node.
- If there are large numbers of items on one or more nodes, there may well be a requirement to utilize a specific different search technique for locating items within that particular set. In other words, **a tree search is not mutually exclusive with any other search technique** that may be used for specific sets. It is simply a method of reducing the number of relevant items to be searched (by whatever method) to those within certain branches of the tree.

Example:
In a large city the phone directory may contain entries for people whose surname is "**SMITH**" belonging on a tree branch "**surnames beginning S**". The list of names may, or may not be, further subdivided by subscribers initials. A binary search may be appropriate to locate a particular surname and perhaps thereafter a linear search to locate a particular address.

22

Searching Techniques



Graph search

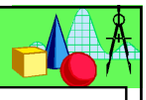
Many of the problems in **graph theory** can be solved using graph traversal algorithms, such as:

- **Dijkstra's algorithm**
- **Kruskal's algorithm**
- **The nearest neighbour algorithm**
- **Prim's algorithm**

These are all extensions of the tree-search algorithms.

23

Searching Techniques

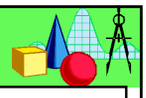


Informed search

- In an informed search, a heuristic that is specific to the problem is used as a guide.
- A good heuristic will make an informed search dramatically out-perform any uninformed search.
- There are ***few*** prominent informed list-search algorithms.
- A possible member of that category is a hash table with a hashing function that is a heuristic based on the problem at hand.
- Most informed search algorithms explore **trees.**
- Like the uninformed algorithms, they can be extended to work for graphs as well.

24

Searching Techniques



Adversarial search

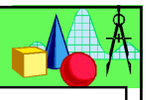
In games such as **chess**, there is a **game tree** of all possible moves by both players and the resulting board configurations, and a search can be made of this tree to find an effective playing strategy.

This type of problem has the unique characteristic that one must account for any possible move the opponent might make.

To account for this, game-playing computer programs, as well as other forms of artificial intelligence like **machine planning**, often use other search algorithms like the **Minimax algorithm**, **search tree pruning**, and **alpha-beta pruning** algorithms.

25

Searching Techniques



Constraint Satisfaction

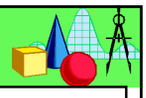
- This type of search solves **constraint satisfaction problems** where, rather than looking for a path, the solution is simply **a set of values assigned to a set of variables**. Because the variables can be processed in any order, the usual tree search algorithms are too inefficient.
- Methods of solving constraint problems include **combinatorial search** and **backtracking**. Common tricks or techniques involved in backtracking is **Constraint propagation**, which is a general form of Forward checking. Other local search algorithms, such as generic algorithms, which minimize the conflicts, also do a good job.

In **Minimax,** algorithm first takes all the minimum values then from among them, take the maximum value.

It's vice versa for the **Maximin** algorithm.

26

Searching Techniques



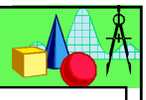
Other Types of Search Algorithms

String Searching algorithms

- sometimes called **string matching algorithms**
- try to find a place where one or several strings (also called patterns) are found within a larger string or text.

27

Searching Techniques



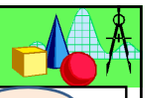
Other Types of Search Algorithms

Genetic Algorithms and Genetic Programming

- Used in computing to find exact or approximate solutions to optimization and search problems.
- Algorithms are categorized as global search heuristics.
- Belong to a particular class of **evolutionary algorithms** (also known as **evolutionary computation**) that use techniques inspired by evolutionary biology such as inheritance, mutation, selection, and crossover (also called recombination).

28

Searching Techniques



Other Types of Search Algorithms

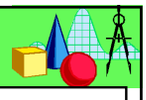
Find an acceptable approximation!

Simulated Annealing

- a generic probabilistic meta-heuristic for the global optimization problem of applied mathematics, namely **locating a good approximation to the global minimum of a given function in a large search space.**
- often used when the search space is discrete (e.g., Choosing a "good" restaurant to go to have dinner).
- For certain problems, simulated annealing may be more effective than exhaustive enumeration - provided that the goal is merely to find an acceptably good solution in a fixed amount of time, rather than the best possible solution.

29

Searching Techniques



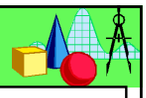
Other Types of Search Algorithms

Recommender Systems

- Use statistical methods to rank results in very large data sets.
- A specific type of **information filtering** (IF) technique.
- Attempts to present information items (movies, music, books, news, images, web pages, etc.) that are likely of interest to the user.
- Typically, a recommender system compares the user's profile to some reference characteristics, and seeks to predict the 'rating' that a user would give to an item they had not yet considered. These characteristics may be from the information item (the **content-based approach**) or the user's social environment (the **collaborative filtering approach**).

30

Searching Techniques



Other Types of Search Algorithms

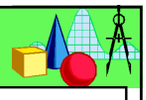
Tabu Search

- A technique to avoid discrete searches getting stuck in local minima.
- Enhances the performance of a local search method by using memory structures:

  Once a potential solution has been determined, it is marked as "**taboo**" (not to be visited again).

31

Searching Techniques



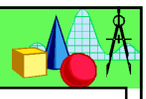
Other Types of Search Algorithms

Minimax

- highly optimized using alpha-beta pruning
- an algorithm to search for good moves in zero-sum games
- used in decision theory, game theory, statistics and philosophy for ***mini*mizing** the ***max***imum possible loss.
- Alternatively, it can be thought of as maximizing the minimum gain (**Maximin**).
- extended for more complex games and to general decision making in the presence of uncertainty.

32

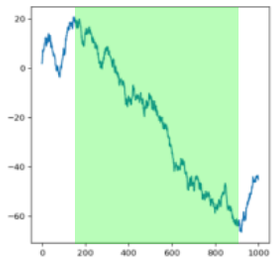
Searching Techniques



Other Types of Search Algorithms

Ternary Search

- technique for finding the minimum or maximum of a function that is either **strictly increasing** and then **strictly decreasing** or vice versa.
- Finds the minimum or maximum of a unimodal function.
- An example of a divide-and-conquer algorithm.



33

UNIVERSITY OF THE WEST INDIES

CAVE HILL CAMPUS

Department of Computer Science, Mathematics & Physics

COMP2611 – Data Structures

Project #1

DUE: Sunday, 16 October, 2022 at Midnight

Consider a text data file called "**Countries.dat**", which contains countries' record with the following data fields:

Country's name	(e.g. **Trinidad and Tobago**);
Country's size	(e.g. **1841 square miles**);
Country's population	(e.g. **1267145**);
Country's governance model	(e.g. **Democratic**).

In the data file, each field is separated by a comma:

e.g. **Trinidad and Tobago, 1841, 1267145, Democratic**

Write a **C++** program which will read the records from the data file and put the records into the following Abstract Data Types as follows:

1. **Queue**: All the records.
2. **Priority Queue**: All records in ascending order of the countries' names.
3. **Deque**: Countries with democratic governments at the **head** and others at the **tail**.
4. **Stack**: All records.

Things To Note:

1. You need to create a node class to store the records. All the ADTs can use that class.
2. Each ADT's class and member functions should be defined in its own header file. The header files should be named: "**Queue.h**", "**PQueue.h**", "**Deque.h**" and "**Stack.h**" respectively.
3. Along with a constructor function, each class should contain an accessor and a mutator function for each data attribute.
4. A main source program which will manipulate the ADTs. The main program should perform the following tasks:
    a. Read the data from the data file and load the records into the ADTs according to the ADTs' requirements above.
    b. Display the records in each ADT.
    c. Remove a record (e.g *dequeue, pop, dequeueHead, dequeueTail*) from the ADTs and display the resulting ADT's records.
5. **Very, very important:** Your code should be properly documented!!!!

Your program will be tested with an independent data file.

SUBMISSION:

Zip up all the files into a zip file with your ID number as the file's name (e.g. **123456789-01.zip**) and submit it through the course's eLearning portal through the label "**Assignment #1**" no later than **Sunday 16 October, 2022 at midnight**. You may submit your zipped file multiple times, only the most recent copy will be retained. However, the portal will close at exactly 12:01 am on Monday **17 October, 2022**.