

Data Structures and Algorithms Using C++

Jagadeesh Vasudevamurthy Ph. D
Instructor
UCSC Extension in Silicon Valley
Santa Clara, CA 95054 USA
jvasudev@ucsc.edu



CMPR.X406 - Data Structures and Algorithms Using C++

Dates: Sep 22, 2018 - Dec 1, 2018

Time: 09:00 AM PDT

Location

SANTA CLARA

Instructors

Jagadeesh Vasudevamurthy

**Draft 16
Sept 2018**

Recommended Texts:

Recommended Texts:

Data Structures and Algorithms in C++, 2nd edition, Michael T. Goodrich, et al., Wiley, 2011, ISBN-10: 0470383275, ISBN-13: 978-0470383278.

Algorithms, Sanjoy Dasgupta, et al., McGraw-Hill, 2006, ISBN-10: 0073523402, ISBN-13: 978-0073523408.

This work is dedicated to the greatest guru of the world

**Lord Krishna
Krishnam Vande Jagadgurum**

Chapter 1

Basic C++

1.1 Introduction

1.2 Installing Microsoft Visual C++ 2015 compiler

Getting free Microsoft Visual Studio Community 2015

Jagadeesh Vasudevamurthy

06/22/2016

1. Go this website:

<https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx>



Click on Download



Visual Studio

Community 2015
with Updates

Choose your installation location

C:\Program Files (x86)\Microsoft Visual Studio 14.0

...

Setup requires up to 18 GB across all drives.

By clicking the "Next" button, I acknowledge that I accept the [License Terms](#) and [Privacy Statement](#).

[Cancel](#)

[Next](#)



Community 2015 with Updates

Select features

- Programming Languages
 - Visual C++
 - Common Tools for Visual C++ 2015
 - Microsoft Foundation Classes for C++
 - Windows XP Support for C++
 - Visual F#
 - Python Tools for Visual Studio (March 2016)
- Windows and Web Development
 - ClickOnce Publishing Tools
 - Microsoft SQL Server Data Tools
 - Microsoft Web Developer Tools
 - PowerShell Tools for Visual Studio (3rd Party)

[Select All](#)

[Reset Defaults](#)

Setup requires up to 16 GB across all drives.

[Back](#)

[Next](#)



Community 2015 with Updates

Selected features

MICROSOFT SOFTWARE

C#/.NET (Xamarin v4.1.0)

[License Terms](#)

C#/.NET (Xamarin v4.0.2 or earlier) will be removed

[License Terms](#)

Windows 8.1 SDK and Universal CRT SDK

Common Tools for Visual C++ 2015

Tools for Universal Windows Apps (1.3.2) and Windows 10 SDK (10.0.10586)

THIRD-PARTY SOFTWARE

Android Native Development Kit (R10E, 32 bits)

[License Terms](#)

Android SDK Setup (API Level 23)

[License Terms](#)

Setup requires up to 16 GB across all drives.

By clicking "Install" you agree to the license terms of the software and components you have selected to install. You are responsible for reading and accepting these license terms. Microsoft grants you no rights for any third party software you have selected; it is provided by the third parties indicated in the applicable license terms.

[Back](#)

[Install](#)

**With this installation is you should see this on your desktop
Make a short cut on desktop**



1.3 Writing, compiling and debugging first program using Microsoft Visual C++ 2013 compiler

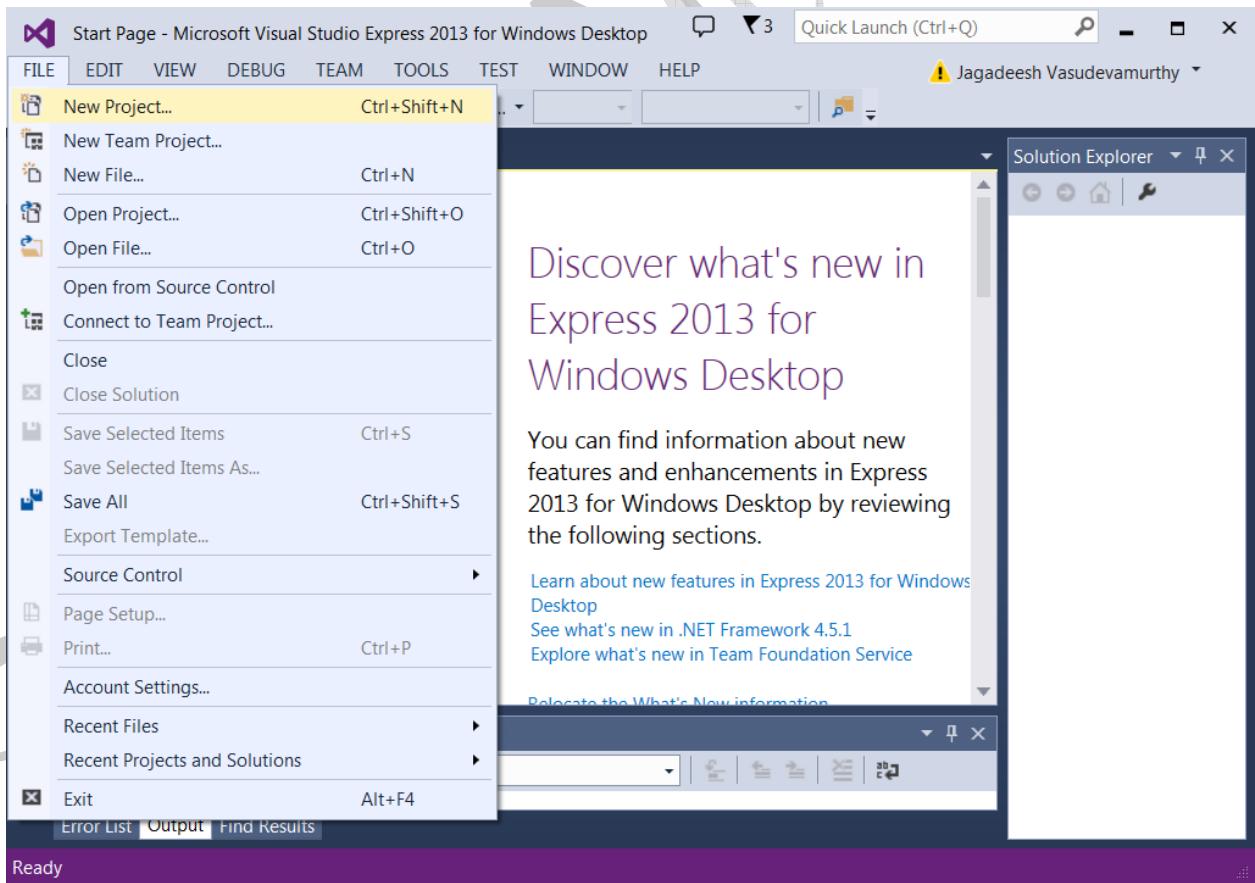
Writing, compiling and debugging first program

Jagadeesh Vasudevamurthy
04/08/2014

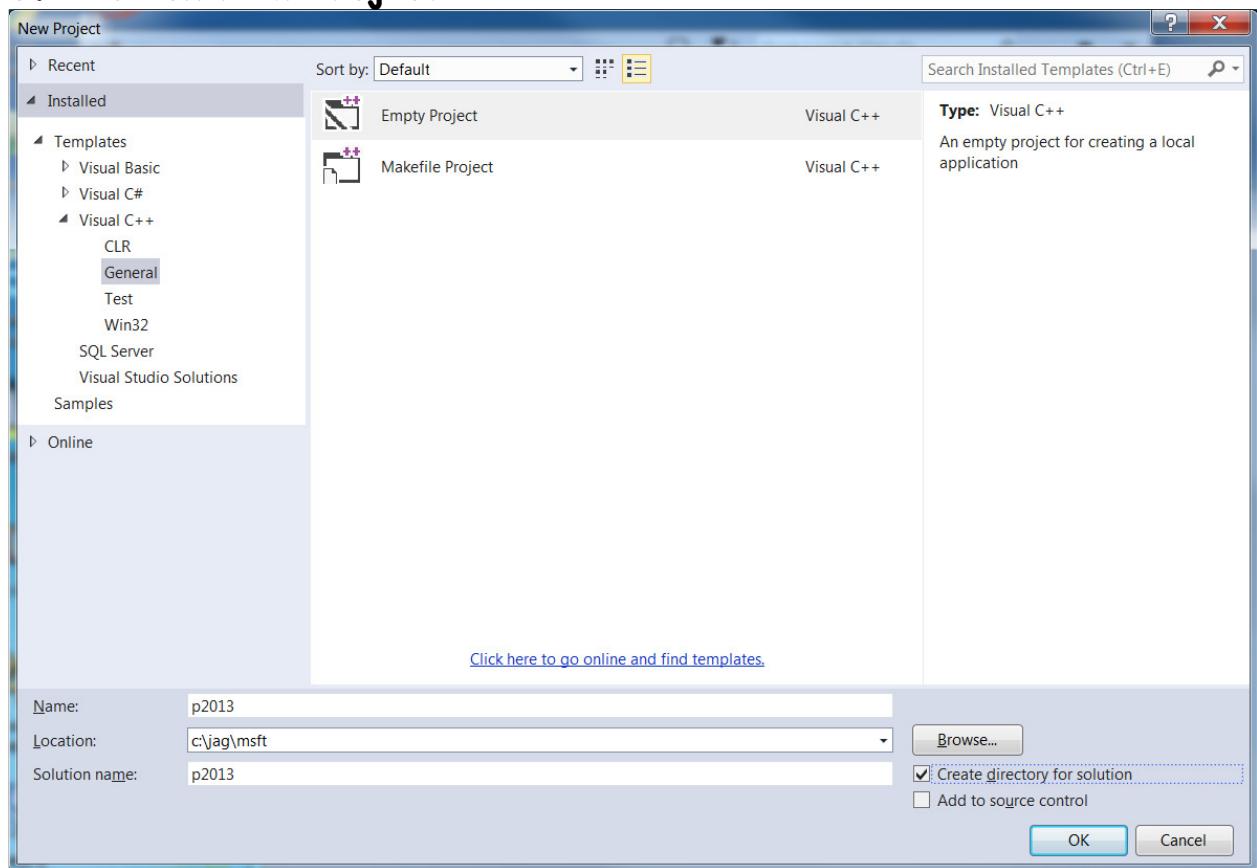
1. Double click on this icon



2. Create a new project

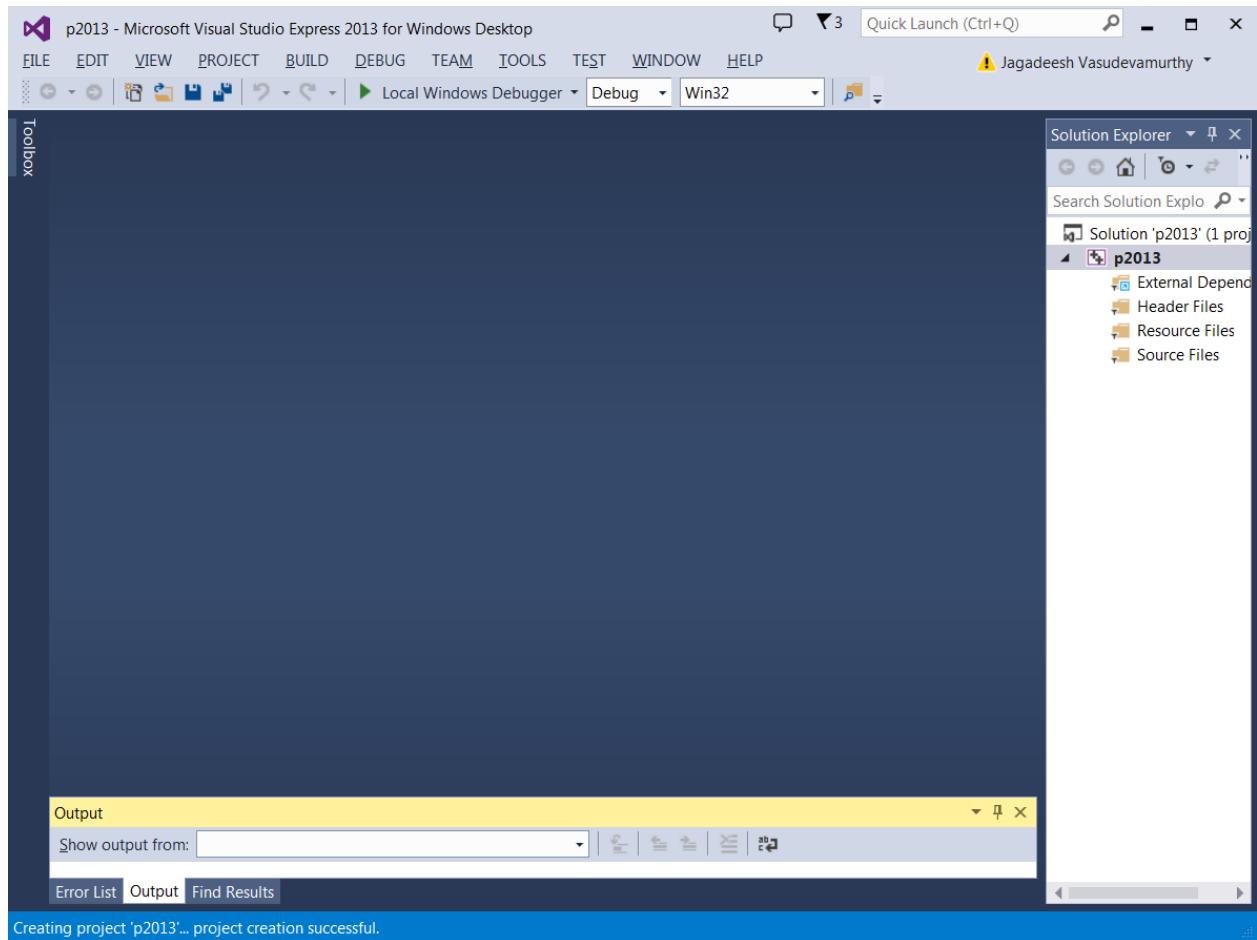


3. Click on New Project



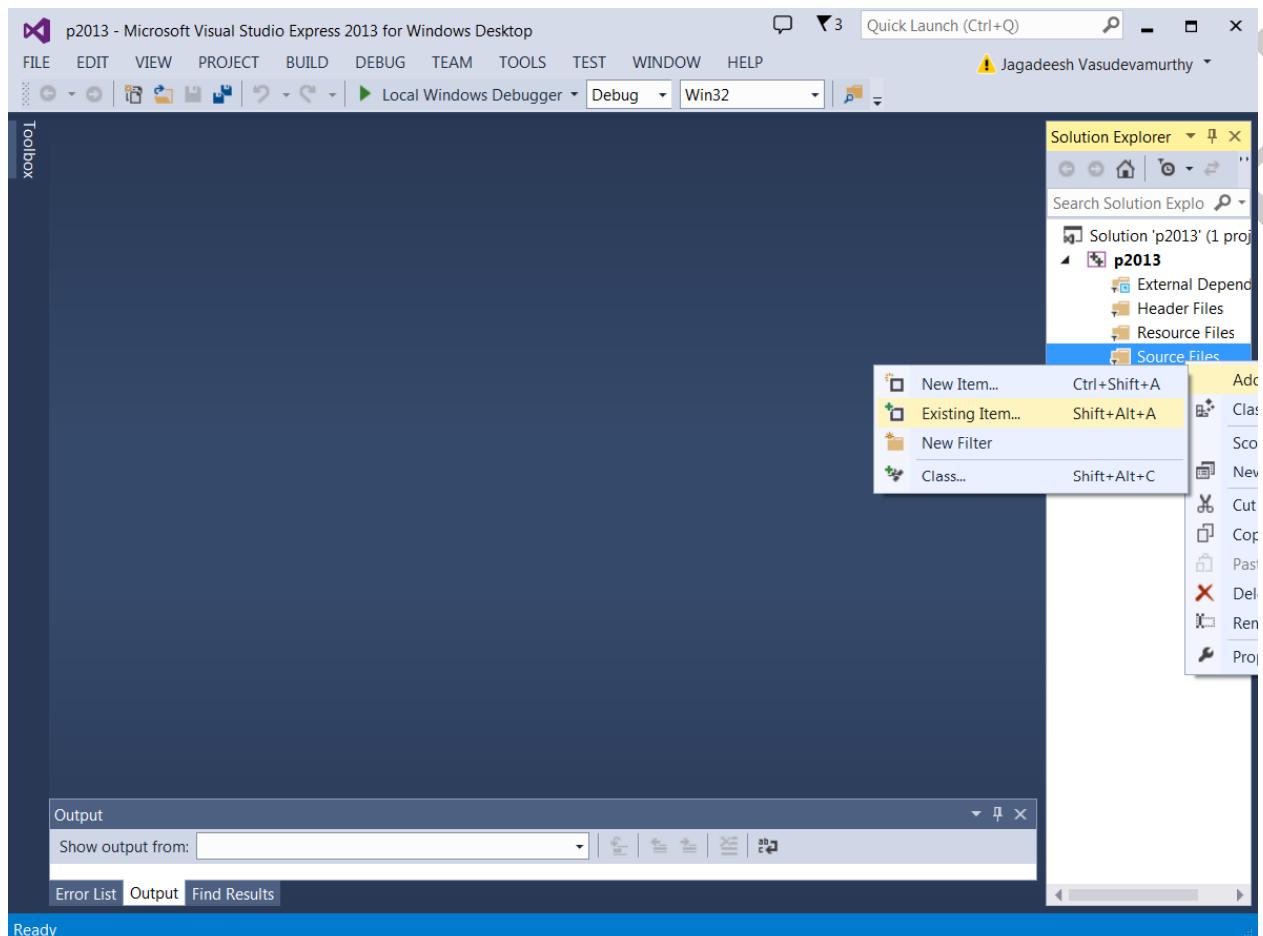
1. Select GENERAL
2. Select EMPTY PROJECT
3. Name: p2013
4. Location: C:\jag\msft
5. Click OK

4. You will then see a screen as follows:



5.

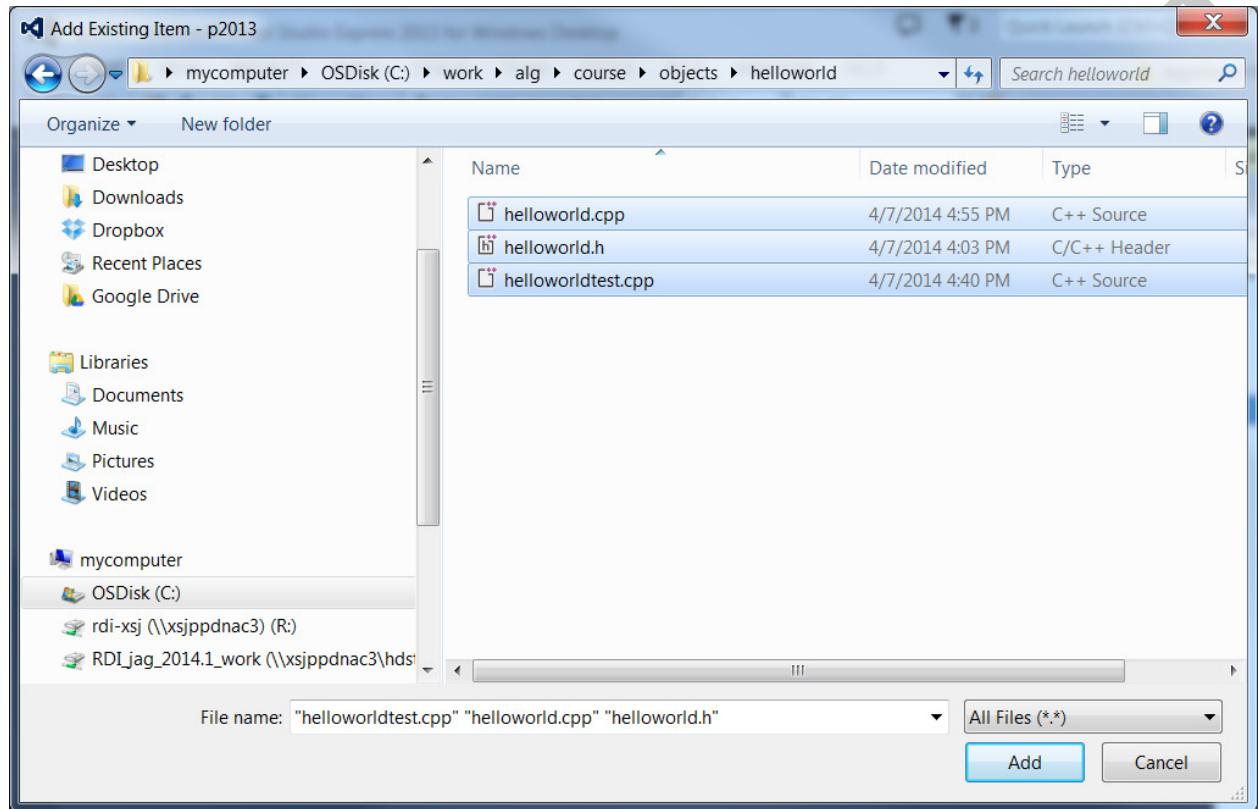
- 1. Right click on Source file**
- 2. select Add -> Existing Item as follows**



6.

Add 3 files as follows

7 Now you will a screen as follows:



8 Add files save the project

A screenshot of Microsoft Visual Studio Express 2013. The code editor window shows a file named "helloworldtest.cpp". A red dot, indicating a break point, is placed on the line before the closing brace of the main function. The code includes comments like "This file test helloworld object" and "All includes here". The Solution Explorer window on the right shows a solution named "p2013" with files "helloworld.cpp", "helloworld.h", and "helloworldtest.cpp". The Output window below the code editor shows "Build: 0 succeeded, 0 failed, 1 up-to-date, 0 skipped".

9. Compiling the above program by pressing F7

If everything is right, you should get this

```
===== Build: 0 succeeded, 0 failed, 1 up-to-date, 0 skipped =====
```

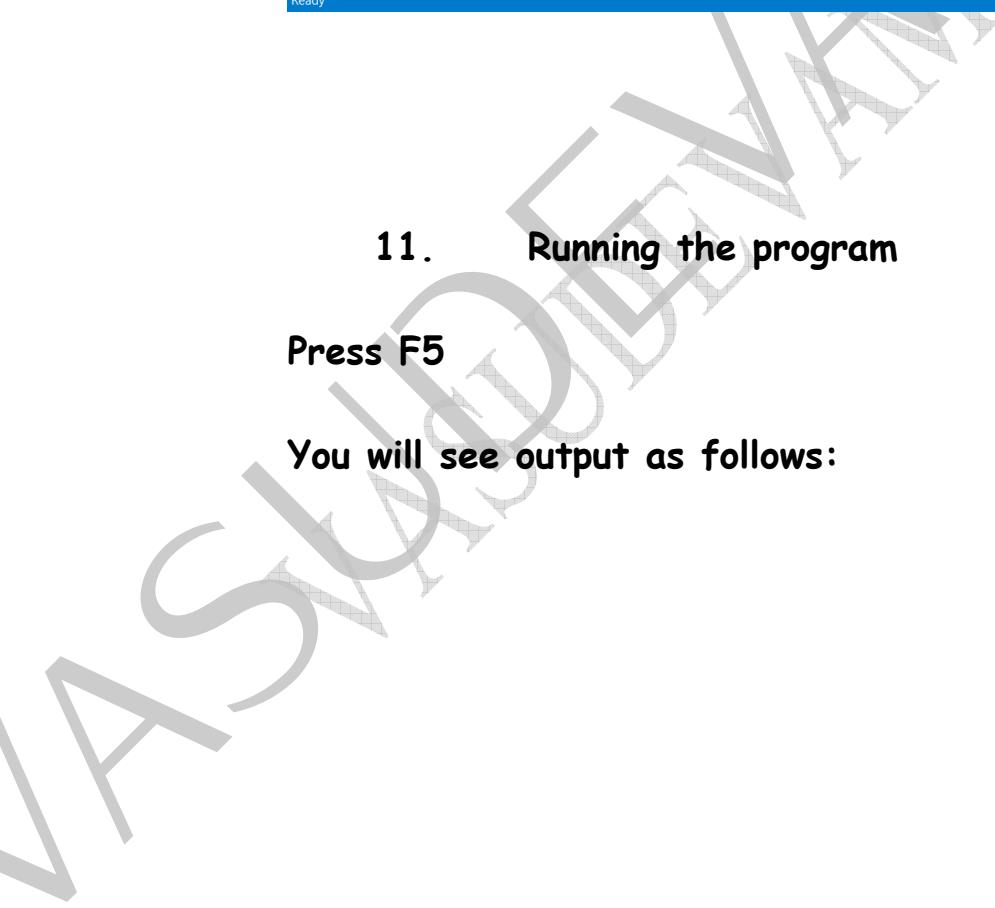
10. Setting a break point

Go to last line }

Left click on }

Press F9

You will see a red dot as follows

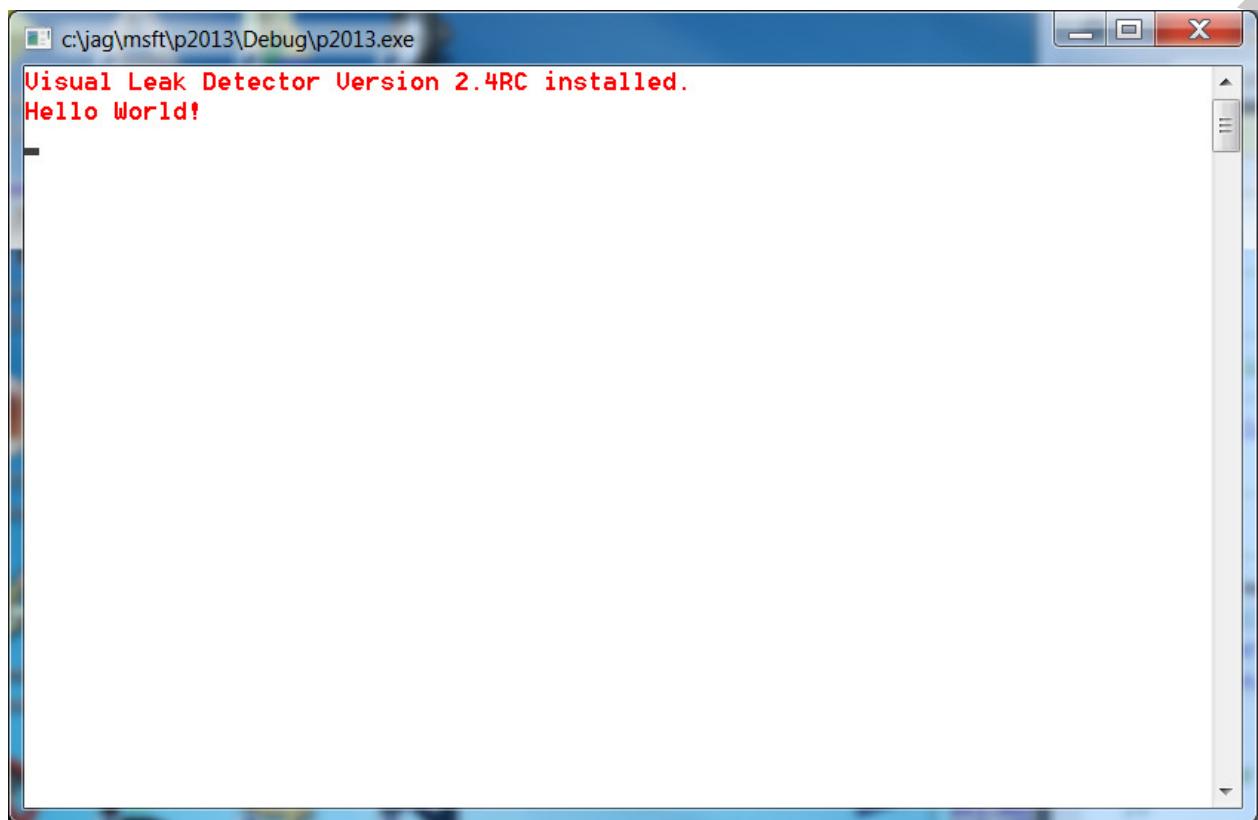


A screenshot of Microsoft Visual Studio Express 2013. The main window shows a code editor with the file "helloworldtest.cpp". The code contains a main() function that creates a new integer array. The Solution Explorer on the right shows a project named "p2013" with files "helloworld.cpp", "helloworld.h", and "helloworldtest.cpp". The Output window at the bottom shows a successful build message: "Build: 0 succeeded, 0 failed, 1 up-to-date, 0 skipped".

11. Running the program

Press F5

You will see output as follows:



Compiling using gnu g++ compiler on Linux system

Program: test.cpp

```
#include <iostream>
using namespace std;

int main() {
    cout << "Welcome to CE589B course\n" ;
    return 0 ;
}
```

Compiling using g++
g++ test.cpp -o test

Running the program
./test

Welcome to CE589B course

1.4 How to detect memory leaks?

Installing visual leak detector

Jagadeesh Vasudevamurthy

04-08-2014

<http://vld.codeplex.com/releases>

Click on

v2.4rc2

Average user rating: No reviews yet

Reviewed: 0 reviews

Downloads: 158

Dev status: Beta ?

RECOMMENDED DOWNLOAD



vld-2.4rc2-setup.exe

application, 1418K, uploaded Sun - 158 downloads

You can see installation at C:\Program Files (x86)\Visual Leak Detector

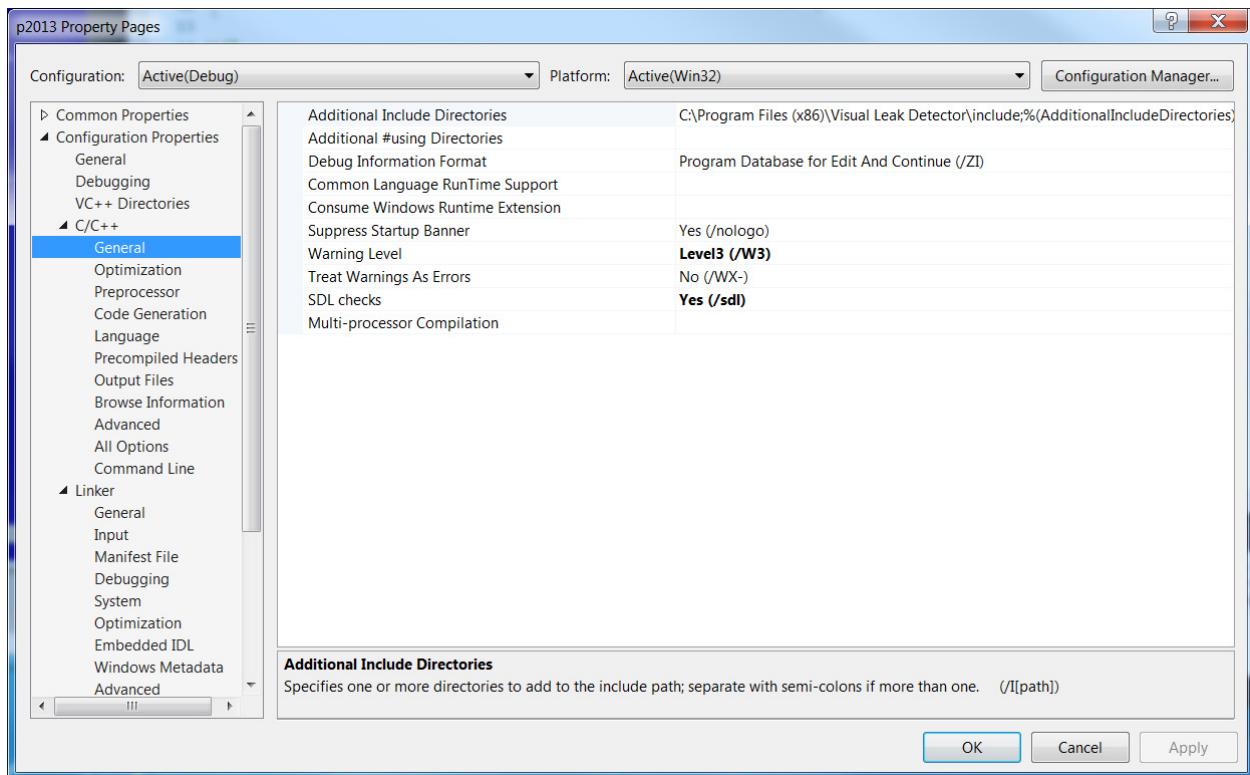
Do the following steps only once after installation

Step 1

Right click on p2013->Properties->C/C++>General->Additional Include
directories->edit

Add

C:\Program Files (x86)\Visual Leak Detector\include

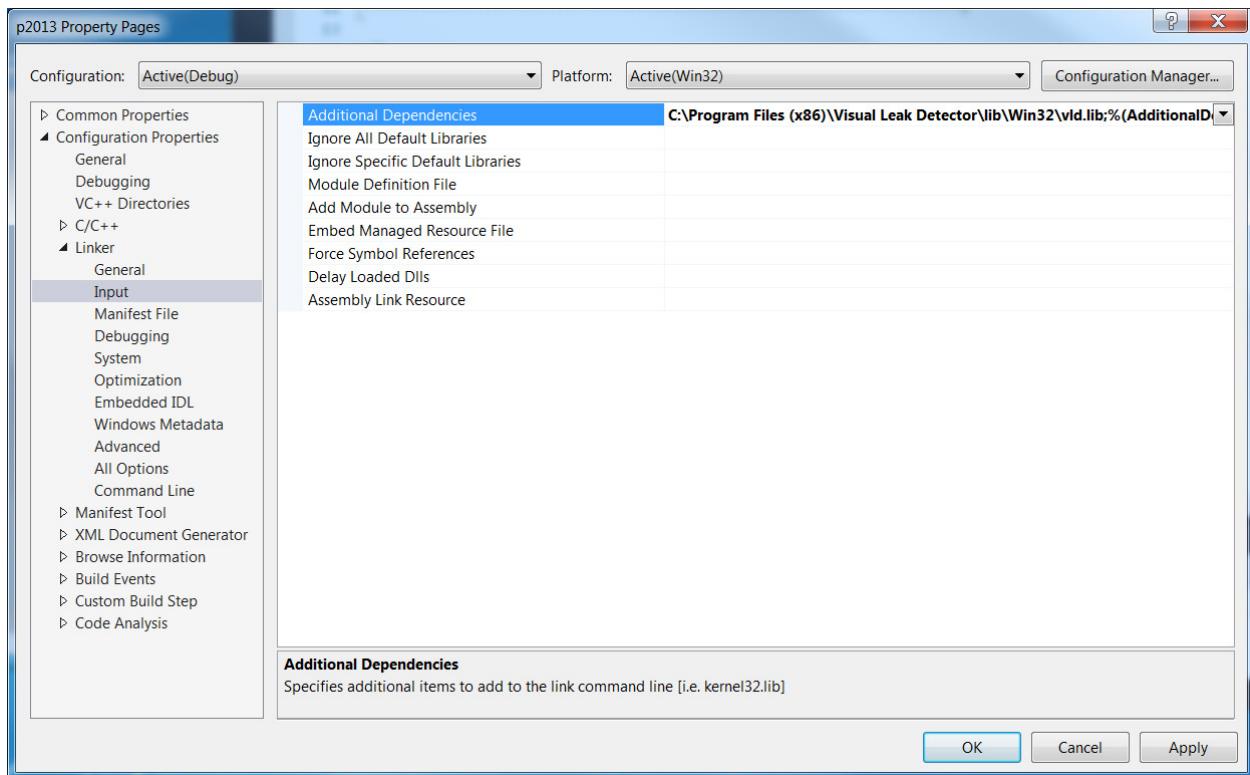


Step 2

Right click on p2013->Properties->Linker->input ->General->Additional Dependencies ->edit

Add

C:\Program Files (x86)\Visual Leak Detector\lib\Win32\vld.lib



Step 3

The project is at: **C:\jag\msft\p2013\Debug**

cd C:\Program Files (x86)\Visual Leak Detector\bin\Win32

Here you see: **vld_x86.dll**

cp * C:\jag\msft\p2013\Debug\.

Step 4

cd C:\Program Files (x86)\Visual Leak Detector\dbghelp\x86

Here you see: **Microsoft.DTfW.DHL.manifest dbghelp.dll**

cp * C:\jag\msft\p2013\Debug\.

How to detect memory leaks:

You need to include

```
#include "vld.h"
```

That's all. Run the program in debugger.

Visual Leak Detector Version 2.4RC installed..

WARNING: Visual Leak Detector detected memory leaks!

Visual Leak Detector detected 1 memory leak (40036 bytes).

Largest number used: 40036 bytes.

Total allocations: 40036 bytes.

1.5. HOW TO WRITE AN OBJECT?

1.5 How to write an object?

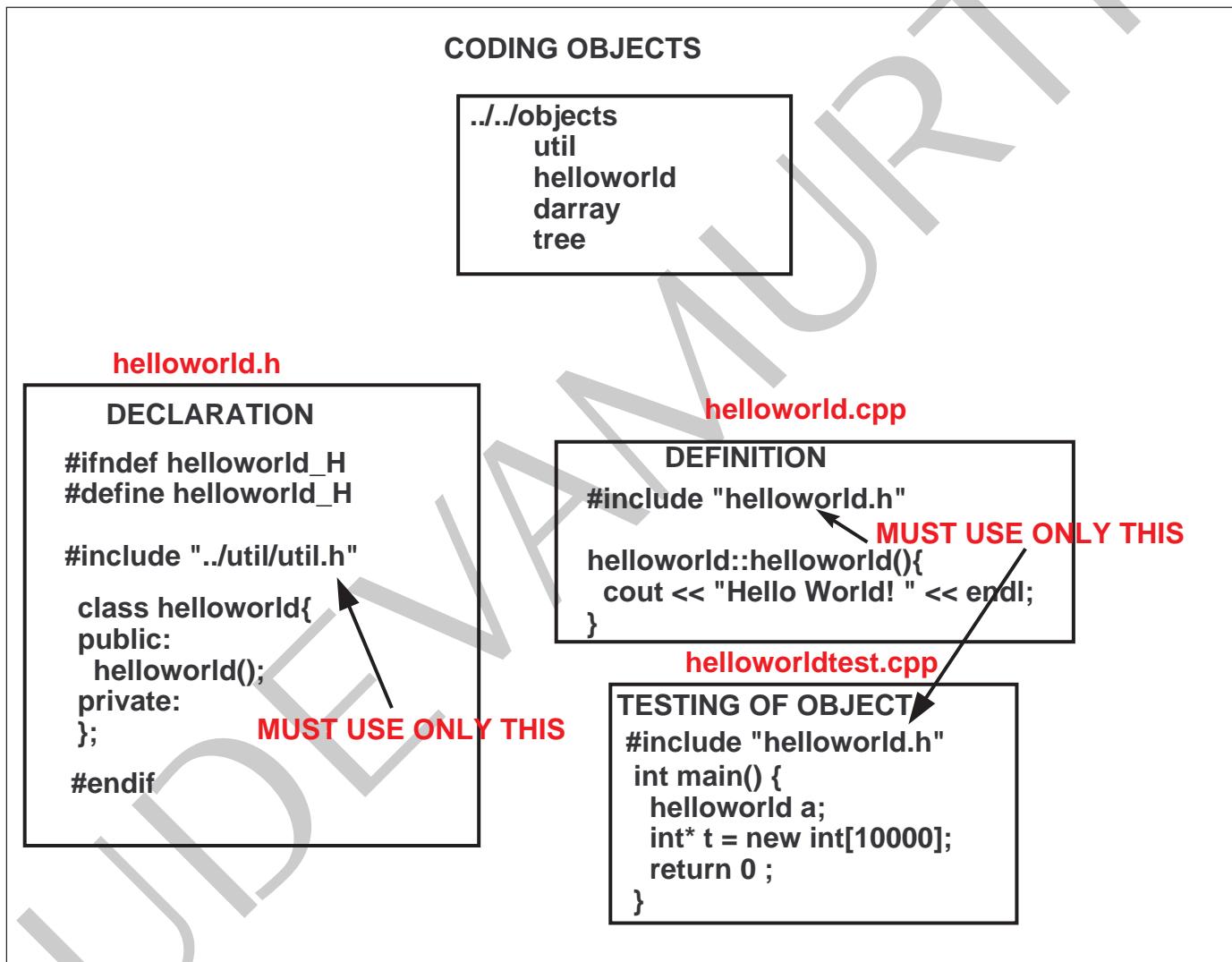


Figure 1.1: Object `helloworld`

1.6 Concept 1: Basic Data types

```
1 /*-----
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy
3 Filename: datatype.cpp
4 -----
5
6 #include <iostream>
7 using namespace std;
8
9 static void print_size() {
10 cout << "sizeof(int) = " << sizeof(int) << endl;
11 cout << "sizeof(char) = " << sizeof(char) << endl;
12 cout << "sizeof(bool) = " << sizeof(bool) << endl;
13 cout << "sizeof(long) = " << sizeof(long) << endl;
14 cout << "sizeof(float) = " << sizeof(float) << endl;
15 cout << "sizeof(double) = " << sizeof(double) << endl;
16 }
17
18 static void usage() {
19 int x = -77 ;
20 cout << "x = " << x << endl ;
21 char c = 'a' ;
22 cout << "c = " << c << endl ;
23 bool b = true ;
24 cout << "b = " << (b ? "true" : "false") << endl ;
25 b = false ;
26 cout << "b = " << (b ? "true" : "false") << endl ;
27 long l = 89789 ;
28 cout << "l = " << l << endl ;
29 float f = 67.89f ;
30 cout << "f = " << f << endl ;
31 double d = 1000000.00 ;
32 cout << "d = " << d << endl ;
33 }
34
35 int main() {
36 print_size();
37 usage();
38 return 0;
39 }
40
41 /*
42 sizeof(int) = 4
43 sizeof(char) = 1
44 sizeof(bool) = 1
45 sizeof(long) = 4 (8 on 64 bit linux)
46 sizeof(float) = 4
47 sizeof(double) = 8
48
49 x = -77
50 c = a
51 b = true
52 b = false
53 l = 89789
54 f = 67.89
55 d = 1e+006
```

1.6. CONCEPT 1: BASIC DATA TYPES

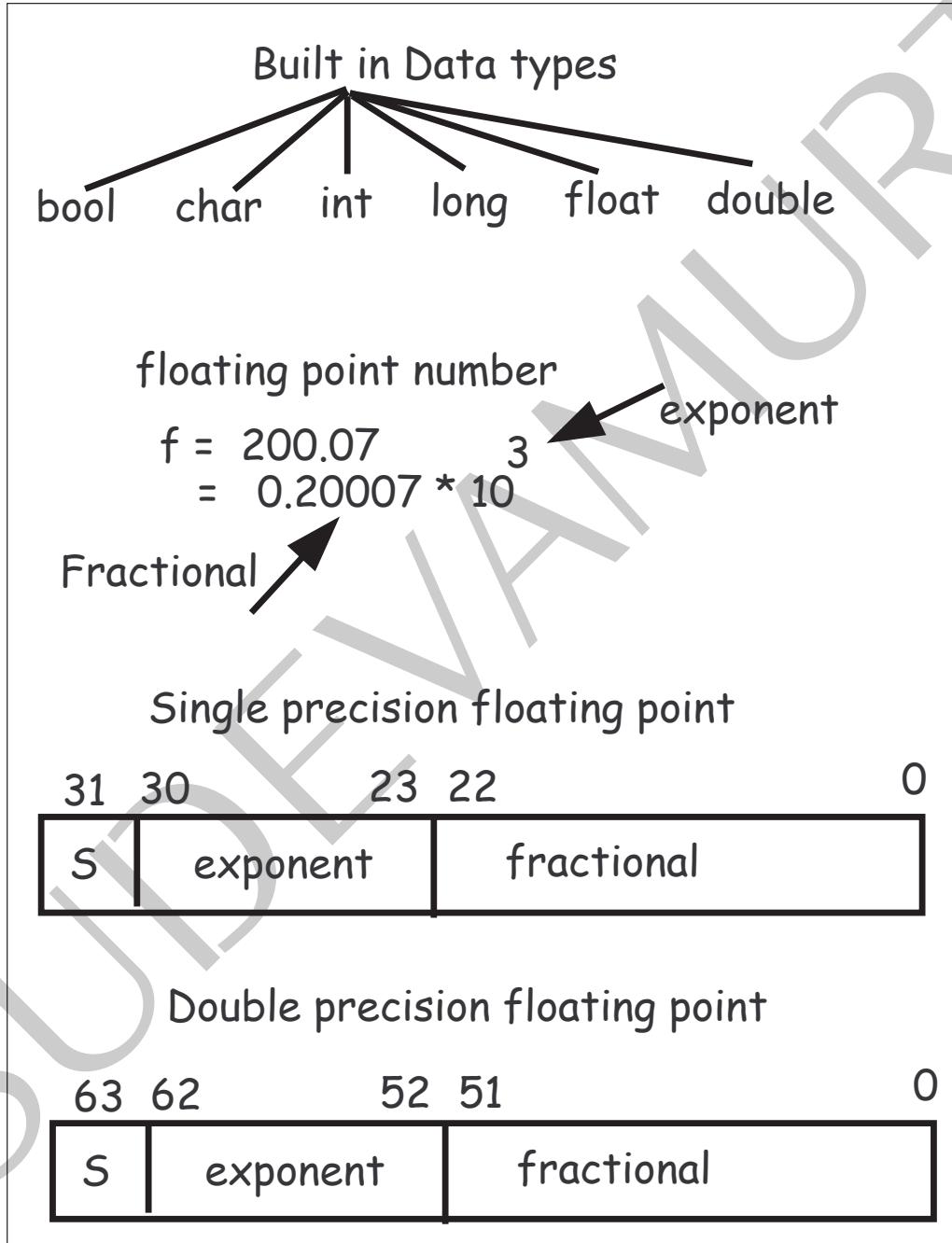


Figure 1.2: Fundamental data types in C++

1.7 Concept 2: User defined data objects

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy  
3 Filename: uobject.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 class book {  
10 public:  
11     char name[20];  
12     int cost;  
13     bool in;  
14 };  
15  
16 int main() {  
17     book b1;  
18     strcpy(b1.name,"Algorithms" );  
19     b1.cost = 23;  
20     b1.in = true ;  
21     cout << "b1: Name = " << b1.name << " Cost = " << b1.cost ;  
22     (b1.in) ? cout << " In " : cout << " Out " ;  
23     cout << endl ;  
24     return 0 ;  
25 }  
26  
27 /*  
28 b1: Name = Algorithms Cost = 23 In  
29 */  
30
```

1.8 Concept 3: Understanding pointers

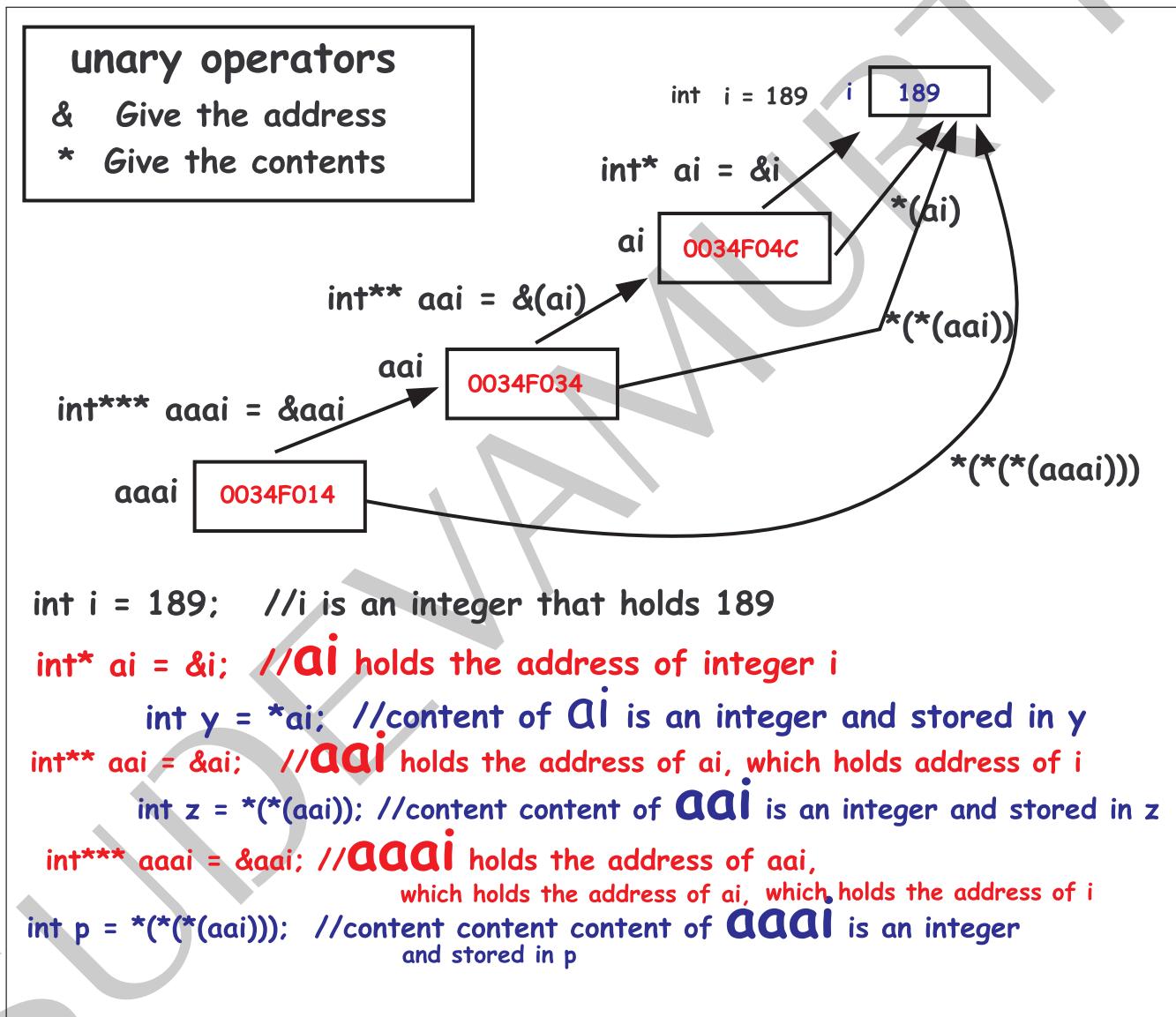


Figure 1.3: Address and content

1.8. CONCEPT 3: UNDERSTANDING POINTERS

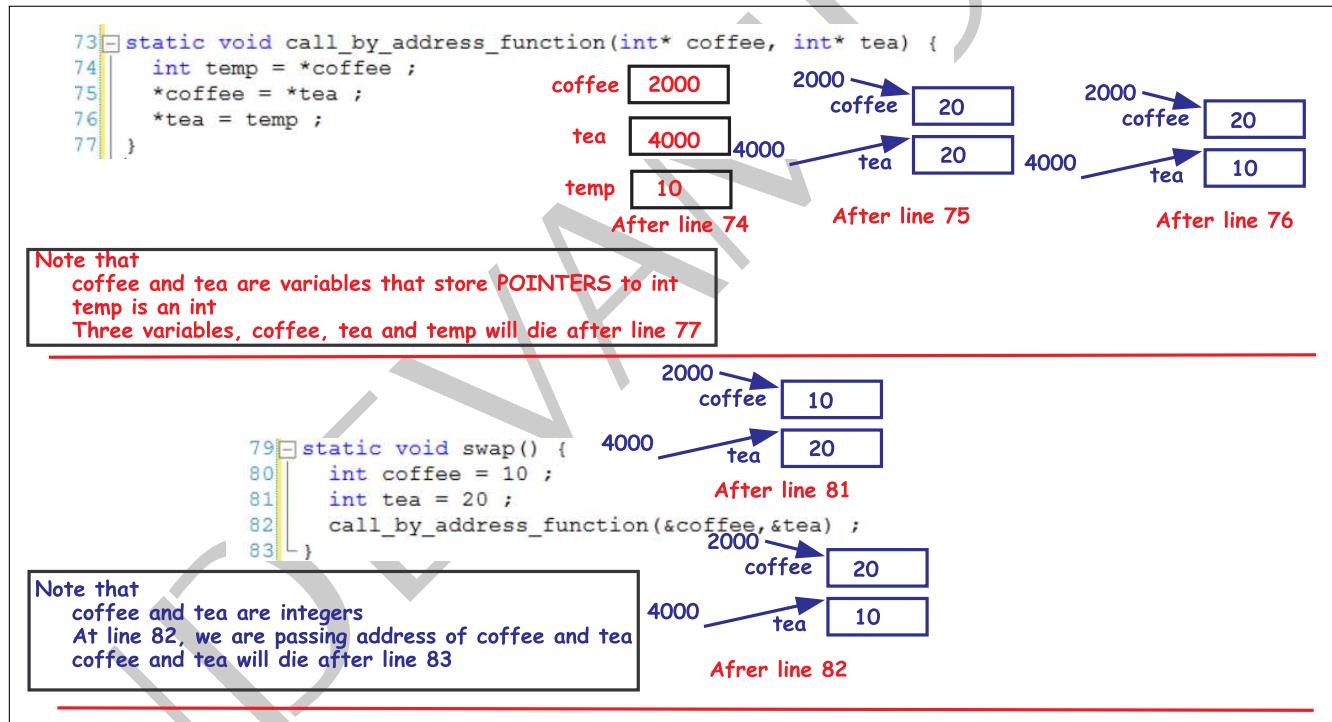


Figure 1.4: Swapping contents of two variables

```
1 /*-----
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy
3 Filename: pointer.cpp
4 -----
5
6 #include <iostream>
7 using namespace std;
8
9 static void p4(int*** i){
10 cout << "in p4: " << "address of address of address of i = " << i;
11 cout << " and i = " << *(*(i)) << endl ;
12 }
13
14 static void p3(int** i){
15 cout << "in p3: " << "address of address of i = " << i;
16 cout << " and i = " << *(*i )<< endl ;
17 int*** aaai = &i ;
18 p4(aaai);
19 }
20
21 static void p2(int* i){
22 cout << "in p2: " << "address of i = " << i;
23 cout << " and i = " << *i << endl ;
24 int** aai = &i ;
25 p3(aai);
26 }
27
28 /-----
29 in p1: i = 189
30 in p2: address of i = 0034F04C and i = 189
31 in p3: address of address of i = 0034F034 and i = 189
32 in p4: address of address of address of i = 0034F014 and i = 189
33 -----
34 static void p1() {
35 int i = 189 ;
36 cout << "in p1: " << "i = " << i << endl ;
37 int* ai = &i ;
38 p2(ai);
39 }
40
41 /-----
42 This function has no output, but has 2 inputs
43 -----
44 static void call_by_address_function(int* coffee, int* tea){
45 cout << "2 : coffee = " << *(coffee) << " tea = " << *(tea) << endl ;
46 int temp = *coffee ;
47 *coffee = *tea ;
48 *tea = temp ;
49 cout << "3 : coffee = " << *(coffee) << " tea = " << *(tea) << endl ;
50 }
51
52 /-----
53 1: coffee = 10 tea = 20
54 2 : coffee = 10 tea = 20
55 3 : coffee = 20 tea = 10
```

```
56 4 : coffee = 20 tea = 10
57 -----
58 static void swap() {
59     int coffee = 10 ;
60     int tea = 20 ;
61     cout << "1: coffee = " << coffee << " tea = " << tea << endl ;
62     call_by_address_function(&coffee,&tea) ;
63     cout << "4 : coffee = " << coffee << " tea = " << tea << endl ;
64 }
65
66 int main() {
67     p1();
68     swap();
69     return 0 ;
70 }
71
```

1.9 Concept 4: Call by value, address and reference

1.9.1 Example 1

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevanurthy  
3 Filename: swap.cpp  
4 -----*/  
5 #include <iostream>  
6 using namespace std;  
7  
8 void swap1(int a, int b){  
9     int t = a ;  
10    a = b ;  
11    b = t ;  
12 }  
13  
14 void swap2(int* a, int* b){  
15     int t = *a ;  
16     *a = *b ;  
17     *b = t ;  
18 }  
19  
20 void swap3(int& a, int& b){  
21     int t = a ;  
22     a = b ;  
23     b = t ;  
24 }  
25  
26 void print(const int a, const int b, const char *s){  
27     cout << s << " a = " << a << " b = " << b << endl ;  
28 }  
29  
30 int main(){  
31     int a = 55;  
32     int b = 69 ;  
33     print(a,b,"Before swap1") ;  
34     swap1(a,b) ;  
35     print(a,b,"after swap1") ;  
36  
37     a = 55; b = 69 ;  
38     print(a,b,"Before swap2") ;  
39     swap2(&a,&b) ;  
40     print(a,b,"after swap2") ;  
41  
42     a = 55; b = 69 ;  
43     print(a,b,"Before swap3") ;  
44     swap3(a,b) ;  
45     print(a,b,"after swap3") ;  
46  
47     return 0 ;  
48 }  
49 /*  
50  
51 Before swap1 a = 55 b = 69  
52 after swap1 a = 55 b = 69  
53 Before swap2 a = 55 b = 69  
54 after swap2 a = 69 b = 55  
55 Before swap3 a = 55 b = 69
```

```
56 after swap3 a = 69 b = 55
57
58 */
59
```

1.9. CONCEPT 4: CALL BY VALUE, ADDRESS AND REFERENCE

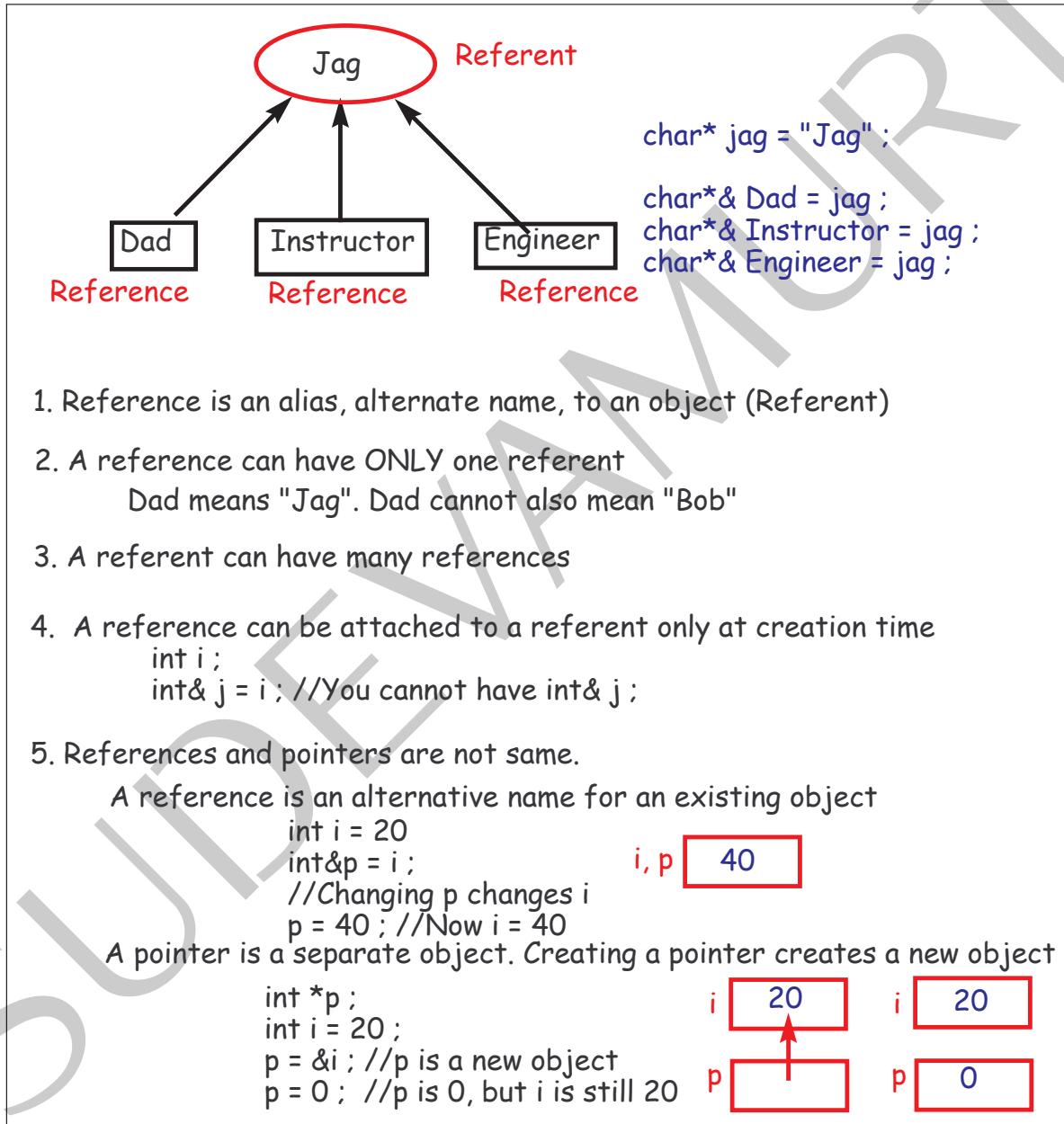


Figure 1.5: Alias and pointers

https://www.hasustorm.com/books/English/Addison.Wesley.Cpp.FAQs.2nd.Edition.internal.eBook-LiB.chm/0201309831_ch11lev1sec4.html

FAQ 11.04 What happens when a value is assigned to a reference?

The reference remains bound to the same referent,
and the value of the referent is changed.

```
void f(int& x, int* y, int z){  
    x = 5; //main()'s i changed to 5  
    *y = 6; //main()'s i changed to 6  
    z = 7; //no change to main()'s i  
}  
  
int main()  
{  
    int i = 4;  
    f(i, &i, i);  
}
```

Figure 1.6: FAQ 1

1.9. CONCEPT 4: CALL BY VALUE, ADDRESS AND REFERENCE

https://www.hasustorm.com/books/English/Addison.Wesley.Cpp.FAQs.2nd.Edition.internal.eBook-LiB.chm/0201309831_ch11lev1sec8.html

FAQ 11.08 Can a reference be made to refer to a different referent?

No, it can't.

Unlike a pointer, once a reference is bound to an object, it cannot be made to refer to a different object. The alias cannot be separated from the referent.

For example, the last line of the following example changes i to 6; it does not make the reference k refer to j. Throughout its short life, k will always refer to i.

```
int main()
{
    int i = 5;
    int j = 6;
    int& k = i; // 1
    k = j; // 2
}
```

- (1) Bind k so it is an alias for i
(2) Change i to 6—does NOT bind k to j

Figure 1.7: FAQ 2

https://www.hasustorm.com/books/English/Addison.Wesley.Cpp.FAQs.2nd.Edition.internal.eBook-LiB.chm/0201309831_ch11lev1sec11.html

FAQ 11.11 When are pointers needed?

References are usually preferred over pointers when aliases are needed for existing objects, making them useful in parameter lists and as return values.

Pointers are required when it might be necessary to change the binding to a different referent or to refer to a nonobject (a NULL pointer). Pointers often show up as local variables and member objects.

Figure 1.8: FAQ 3

https://www.hasustorm.com/books/English/Addison.Wesley.Cpp.FAQs.2nd.Edition.internal.eBook-LiB.chm/0201309831_ch11lev1sec10.html

FAQ 11.10 Aren't references just pointers in disguise?

No, they are not.

It is important to realize that references and pointers are quite different. A pointer should be thought of as a separate object with its own distinct set of operations (*p, p->blah, and so on). So creating a pointer creates a new object.

In contrast, creating a reference does not create a new object; it merely creates an alternative name for an existing object.

Furthermore the operations and semantics for the reference are defined by the referent; references do not have operations of their own.

In the following example, notice that assigning 0 to the reference j is very different than assigning 0 to the pointer p (the 0 pointer is the same as the NULL pointer).

```
int main()
{
    int i = 5;
    int& j = i;    <-- 1
    int* p = &i;   <-- 2

    j = 67;    <-- 3
    p = 0;     <-- 4
}
```

(1) j is an alias for i
(2) p is a new object, not an alias
(3) Changes i. At this point both i and j are 67
(4) Changes p; does not affect i

Figure 1.9: FAQ 4

1.10. CONCEPT 5: MEMORY ALLOCATION

1.10 Concept 5: Memory allocation

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy  
3 Filename: memalloc.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 static void test_int(){  
10 int* p = new int(27);  
11 cout << "address of p = " << p << " and the content of p is = " << *p << endl ;  
12 delete p;  
13 }  
14  
15 static void test_char(){  
16 char* p = new char('H');  
17 cout << "content of p is = " << *p << endl ;  
18 delete p;  
19 }  
20  
21 static void test_int_array(int size){  
22 int* p = new int [size];  
23 for (int i = 0; i < size; i++){  
24 p[i] = i + 1;  
25 }  
26 for (int i = 0; i < size; i++){  
27 cout << "p[" << i << "] = " << p[i];  
28 }  
29 cout << endl ;  
30 delete [] p;  
31 }  
32  
33 static void test_char_array(int size){  
34 char* p = new char [size];  
35 strcpy(p,"abcd");  
36 cout << "p = " << p << endl ;  
37 delete [] p;  
38 }  
39  
40 static void test_obj(int size){  
41 class student{  
42 public:  
43 int age ;  
44 char *id ;  
45 };  
46 student* a = new student ;  
47 a->age = 25 ;  
48 a->id = new char[size] ;  
49 strcpy(a->id,"Z567") ;  
50 cout << "Student id = " << a->id << " Student age = " << a->age << endl ;  
51 delete [] a->id ;  
52 delete a ;  
53 }  
54  
55
```

```
56 int main() {  
57     test_int();  
58     test_char();  
59     test_int_array(5);  
60     test_char_array(5);  
61     test_obj(5);  
62     return 0;  
63 }  
64 /*  
65 address of p = 003A8E0 and the content of p is = 27  
66 content of p is = H  
67 p[0] = 1 p[1] = 2 p[2] = 3 p[3] = 4 p[4] = 5  
68 p = abcd  
69 Student id = Z567 Student age = 25  
70 */  
71
```

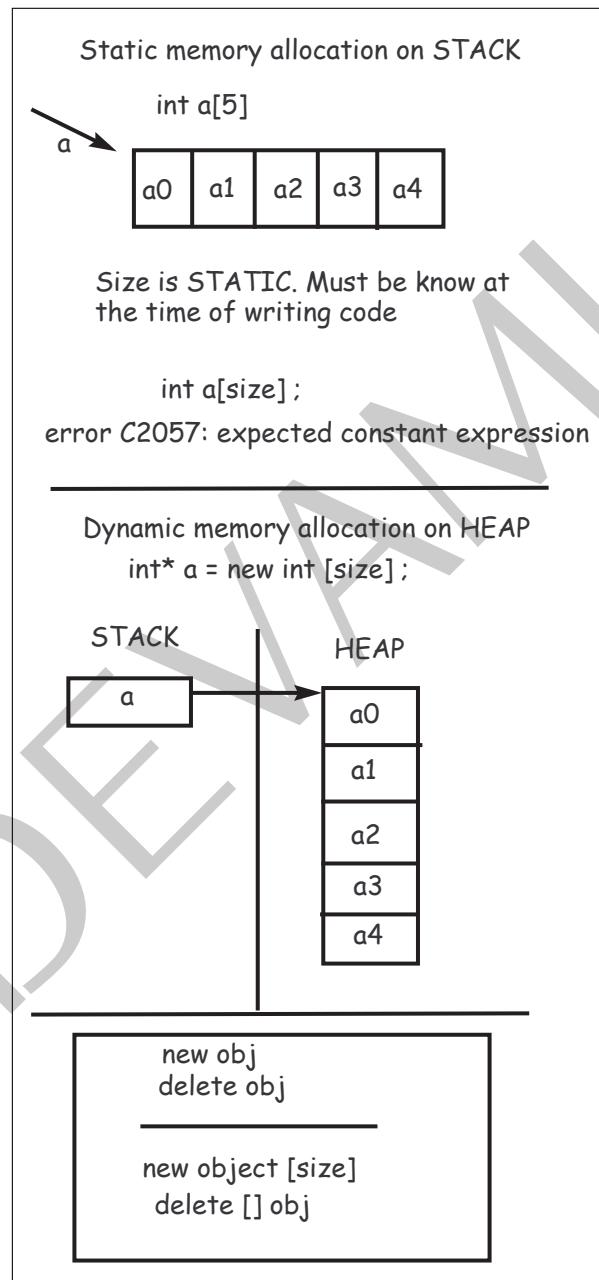


Figure 1.10: Static and dynamic memory allocation

1.11. CONCEPT 6: UNDERSTANDING CONST

1.11 Concept 6: Understanding const

```
char *p = "jag"; //Non constant pointer, Non constant data  
const char* p = "jag" ; //Non constant pointer, constant data  
char* const p = "jag" ; //constant pointer, non constant data  
const char* const p = "jag" ; //constant pointer, constant data
```

What's pointed to
is constant

```
char *  
const char *  
char *  
const char *
```

Pointer is
constant

```
p = "jag"  
p = "jag"  
const p = "jag"  
const p = "jag"
```

Use const whenever possible

Figure 1.11: Use **const** whenever possible

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy  
3 Filename: const.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 /*-----  
10 integer test  
11 -----*/  
12 static void int_test()  
13 {  
14     int x = 4;  
15     x = 10;  
16     //const int y = 2;  
17     //y = 89;  
18     //FAILS: error C3892: 'y' : you cannot assign to a variable that is const  
19 }  
20  
21 /*-----  
22 Generic print routine  
23 -----*/  
24 static void print(const char* title, const char* s)  
25 {  
26     cout << title << endl;  
27     int l = strlen(s);  
28     for (int i = 0; i < l; i++) {  
29         cout << s[i];  
30     }  
31     cout << endl;  
32 }  
33  
34 /*-----  
35 Non constant pointer Non constant data  
36 -----*/  
37 static void ncp_ncd()  
38 {  
39     char* p = new char[4];  
40     strcpy(p,"NPU");  
41     print("ncp_ncd BEFORE: ", p);  
42     p[0] = 'L';  
43     print("ncp_ncd AFTER: ", p);  
44     delete [] p;  
45 }  
46  
47 /*-----  
48 Non constant pointer Constant data  
49 -----*/  
50 static void ncp_cd()  
51 {  
52     const char* p = new char[4];  
53     //strcpy(p,"MIT");  
54     //error C2664: 'strcpy' : cannot convert parameter 1 from 'const char *' to 'char *'  
55     delete [] p;  
56  
57     char buffer[4];  
58     strcpy(buffer,"MIT");  
59     const char *p1 = buffer;
```

```
56 print("ncp_cd BEFORE: ", p1);
57 cout << "ncp_cd BEFORE: " << p1 << endl;
58 //p1[0] = 'N';
59 //error C3892: 'p1' : you cannot assign to a variable that is const
60 }
61
62 /*-----
63 Constant pointer Non constant data
64 -----*/
65 static void cp_ncd() {
66     char* const p = new char[4];
67     strcpy(p,"NPU");
68     print("cp_ncd BEFORE: ", p);
69     p[0] = 'L';
70     print("cp_ncd AFTER: ", p);
71
72     char* const p1 = new char[4];
73     strcpy(p1,"MIT");
74     print("cp_ncd BEFORE2: ", p1);
75     //p = p1;
76     // error C3892: 'p' : you cannot assign to a variable that is const
77     delete [] p;
78     delete [] p1;
79 }
80
81 /*-----
82 Constant pointer Constant data
83 -----*/
84 static void cp_cd() {
85     const char* const p = new char[4];
86     //strcpy(p,"MIT");
87     // error C2664: 'strcpy' : cannot convert parameter 1 from 'const char *const' to 'char *'
88     delete [] p;
89
90     char buffer1[4];
91     strcpy(buffer1,"MIT");
92
93     const char * const p1 = buffer1;
94     print("cp_cd buffer1: ", p1);
95     //p1[0] = 'K';
96     //error C3892: 'p' : you cannot assign to a variable that is const
97
98     char buffer2[4];
99     strcpy(buffer2,"KIT");
100    const char* p2 = buffer2;
101    print("cp_cd buffer2: ", p2);
102    // p1 = p2;
103    // error C3892: 'p1' : you cannot assign to a variable that is const
104 }
105
106 /*-----
107 Main
108 -----*/
109 int main() {
110     int_test();
```

```
111 cout << "_____ " << endl ;
112 ncp_ncd();
113 cout << "_____ " << endl ;
114 ncp_cd();
115 cout << "_____ " << endl ;
116 cp_ncd();
117 cout << "_____ " << endl ;
118 cp_cd();
119 cout << "_____ " << endl ;
120 return 0;
121 }
122
123 */
124
125 ncp_ncd BEFORE:
126 NPU
127 ncp_ncd AFTER:
128 LPU
129
130 ncp_cd BEFORE:
131 MIT
132 ncp_cd BEFORE: MIT
133
134 cp_ncd BEFORE:
135 NPU
136 cp_ncd AFTER:
137 LPU
138 cp_ncd BEFORE2:
139 MIT
140
141 cp_cd buffer1:
142 MIT
143 cp_cd buffer2:
144 KIT
145
146
147 */
148
```

1.12. CONCEPT 7: FUNCTION OVERLOADING

1.12 Concept 7: Function overloading

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy  
3 Filename: overload.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 void f(int i){  
10 cout << "In void f(int i) " << i << endl ;  
11 }  
12  
13 void f(bool i){  
14 cout << "In void f(bool i) " << i << endl ;  
15 }  
16  
17 void f(long i){  
18 cout << "In void f(long i) " << i << endl ;  
19 }  
20  
21 void f(float i){  
22 cout << "In void f(float i) " << i << endl ;  
23 }  
24  
25 void f(double i){  
26 cout << "In void f(double i) " << i << endl ;  
27 }  
28 void f(char i){  
29 cout << "In void f(char i) " << i << endl ;  
30 }  
31  
32 int main(){  
33 f(10);  
34 int i = 2147483647;  
35 f(i);  
36 long l = 2147483647;  
37 f(l);  
38 f(true);  
39 f(false);  
40 double d = 1.7e308;  
41 f(d);  
42 float f1 = 5.67f;  
43 f(f1);  
44 f('a');  
45 return 0;  
46 }  
47  
48 /*  
49  
50 In void f(int i) 10  
51 In void f(int i) 2147483647  
52 In void f(long i) 2147483647  
53 In void f(bool i) 1  
54 In void f(bool i) 0  
55 In void f(double i) 1.7e+308
```

```
56 In void f(float i) 5.67
57 In void f(char i) a
58
59 */
60
```

1.13 Concept 8: Default function argument

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy  
3 Filename: defaultfunc.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 void f1(const char *n, int m){  
10 cout << "f1: Name " << n << " Marks " << m << endl ;  
11 }  
12  
13 void f2(const char *n, int m, int b){  
14 cout << "f2: Name " << n << " Marks " << m + b << endl ;  
15 }  
16  
17 void f3(const char *n, int m, int b, char g){  
18 cout << "f3: Name " << n << " Marks " << m + b << " Grade " << g << endl ;  
19 }  
20  
21 void f(const char *n, int m, int b = 0 , char g = ' '){  
22 cout << "f: Name " << n << " Marks " << m + b ;  
23 if (g != ' '){  
24 cout << " Grade " << g ;  
25 }  
26 cout << endl ;  
27 }  
28  
29 int main(){  
30 f1("John Smith",78);  
31 f2("John Smith",78,10);  
32 f3("John Smith",78,10,'B');  
33 f("John Smith",78);  
34 f("John Smith",78,10);  
35 f("John Smith",78,10,'B');  
36 return 0 ;  
37 }  
38  
39 /*  
40 f1: Name John Smith Marks 78  
41 f2: Name John Smith Marks 88  
42 f3: Name John Smith Marks 88 Grade B  
43 f: Name John Smith Marks 78  
44 f: Name John Smith Marks 88  
45 f: Name John Smith Marks 88 Grade B  
46 */  
47
```

1.14 Concept 9: Constructors and destructors

```
1 /*-----
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy
3 Filename: constdest.cpp
4 -----
5
6 #include <iostream>
7 using namespace std;
8
9 class book {
10 public:
11     book(const char *s, int c = 0, bool x = false);
12     ~book();
13     void print() const;
14
15 private:
16     char* _name;
17     int _cost;
18     bool _in;
19 };
20
21 /*-----
22 Constructor
23 -----
24 book::book(const char *s, int c, bool x): _name(NULL),_cost(c),_in(x){
25     cout << "In book Constructor " << s << endl ;
26     int l = strlen(s) + 1 ;
27     this->_name = new char[l];
28     //you can write without this as: _name = new char[l];
29     strcpy(_name,s);
30 }
31
32 /*-----
33 Destructor
34 -----
35 book::~book() {
36     cout << "In book Destructor " << _name << endl ;
37     delete [] this->_name ;
38     //Without this: delete [] _name ;
39 }
40
41 /*-----
42 print
43 -----
44 void book::print() const {
45     cout << "Name " << _name << " Cost " << _cost ;
46     (_in)? cout << " In " : cout << " Out ";
47     cout << endl ;
48 }
49
50 /*-----
51 main
52 -----
53 int main() {
54     book b1("algorithm");
55     book b2("C++",79);
```

```
56 book b3("VLSI design", 168,true);
57 b1.print();
58 b2.print();
59 b3.print();
60 book* b4 = new book("A book on Java",289,1) ;
61 b4->print();
62 delete b4;
63 return 0;
64 }
65 /**
66 output of the above program
67 -----
68 */
69 In book Constructor algorithm
70 In book Constructor C++
71 In book Constructor VLSI design
72 Name algorithm Cost 0 Out
73 Name C++ Cost 79 Out
74 Name VLSI design Cost 168 In
75 In book Constructor A book on Java
76 Name A book on Java Cost 289 In
77 In book Destructor A book on Java
78 In book Destructor VLSI design
79 In book Destructor C++
80 In book Destructor algorithm
81 */
82
```

1.14. CONCEPT 9: CONSTRUCTORS AND DESTRUCTORS

Constructors

How do you know a constructor?

```
class obj {  
    obj(<input parameters, may be null>)  
};
```

1. Name of the constructor **MUST BE** name of the class
2. Constructor does **NOT** return any thing.
3. If you don't write a constructor, compiler will give one which does nothing as follows:

```
class obj {  
    obj() { }  
}
```
4. There can be many constructors, same name but different parameters.

When constructors are called ?

Case 1: obj p ;
Case 2: obj *p = new obj(<input parameters>) ;

Figure 1.12: Constructors

Destructor

How do you know a destructor?

```
class obj {  
    ~obj() ;  
}
```

1. The name of the destructor **MUST BE** name of the class preceded by ~
2. There are no inputs to destructor
3. Destructor does NOT return anything
4. If you don't write a destructor, compiler will give one with nothing inside

When destructor is called?

Case 1: When an object goes out of scope

```
{  
    obj p ;  
    <may be some code here>  
}
```

Case 2: When you call delete

```
{  
    obj *p = new(obj) ;  
    delete p ;  
}
```

Figure 1.13: Destructor

1.15. CONCEPT 10: COPY CONSTRUCTOR AND EQUAL OPERATOR

1.15 Concept 10: Copy constructor and equal operator

Shallow copy and equal of objects

```
class student {  
public:  
    student(const char* name,int age,bool m=true);  
    ~student();  
    void change_age(int i) {_age = i; }  
    void pass_by_value(student t);  
    student return_by_value();  
    void print() const ;  
private:  
    int _age ;  
    char* _name;  
    bool _is_male ;  
};
```

```
student::student(const char* name,int age,bool m) {  
    _name = new char[strlen(name)+1] ;  
    strcpy(_name,name) ;  
    _age = age ;  
    _is_male = m ;  
}  
student::~student() {  
    delete [] _name;  
}
```

```
static void test_shallow_copy_crash() {  
    student tom("Tom Jr",21);  
    tom.print() ;  
    student mary(tom);  
    mary.print() ;  
}
```

```
static void test_shallow_equal_crash() {  
    student tom("Tom Jr",21);  
    tom.print() ;  
    student mary("Mary Jr",25,false);  
    mary.print() ;  
    mary = tom ;  
    mary.print() ;  
}
```

```
void student::pass_by_value(student t) {  
    t.change_age(_age + t._age) ;  
    t.print() ;  
}  
  
static void test_pass_by_value() {  
    student tom("Tom Jr",21);  
    tom.print() ;  
    student mary("Mary Jr",25,false);  
    mary.print() ;  
    tom.pass_by_value(mary) ;  
    mary.print() ;  
}
```

```
student student::return_by_value() {  
    student t("ken",50) ;  
    t.change_age(_age + t._age) ;  
    t.print() ;  
    return t ;  
}  
  
static void test_return_by_value() {  
    student tom("Tom Jr",21);  
    tom.print() ;  
    student ken = tom.return_by_value();  
    ken.print() ;  
}
```

Figure 1.14: Problems with shallow copy and equal of objects

```
1  /*-----  
2  Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3  Filename: needforcpeq.cpp  
4  -----*/  
5  
6  #include <iostream>  
7  using namespace std;  
8  #include "vld.h"  
9  #pragma warning(disable: 4996) /* Disable deprecation */  
10  
11 /*-----  
12 Declaration of a class student  
13 -----*/  
14 class student {  
15 public:  
16     student(const char* name,int age,bool m=true) ;  
17     ~student();  
18     void change_age(int i) {_age = i; }  
19     void pass_by_value(student t) ;  
20     student return_by_value() ;  
21     void print() const ;  
22 private:  
23     int _age ;  
24     char* _name;  
25     bool _is_male ;  
26 };  
27  
28 /*-----  
29 Implementation of print  
30 -----*/  
31 void student::print() const {  
32     cout << _name << " " << _age << " " ;  
33     cout << (_is_male ? "male" : "female") << endl ;  
34 }  
35  
36 /*-----  
37 Implementation of constructor  
38 -----*/  
39 student::student(const char* name,int age,bool m):_name(nullptr),_age  
    (age),_is_male(m) {  
40     cout << "In student constructor " << name << ":" << age << ":" << boolalpha  
        << m << endl ;  
41     _name = new char[strlen(name)+1] ;  
42     strcpy(_name,name) ;  
43 }  
44  
45 /*-----  
46 Implementation of destructor  
47 -----*/
```

```
C:\work\alg\course\code\ch1\needforcpeq.cpp
48 student::~student() {
49     cout << "In student destructor " << _name << ":" << _age << ":" << boolalpha >
50         << _is_male << endl ;
51     delete [] _name;
52 }
53 /*
54 To show shallow copy operator
55 */
56 static void test_shallow_copy_crash() {
57     student tom("Tom Jr",21);
58     tom.print();
59     student mary(tom);
60     mary.print();
61 }
62 /*
63 To show shallow equal operator
64 */
65 static void test_shallow_equal_crash() {
66     student tom("Tom Jr",21);
67     tom.print();
68     student mary("Mary Jr",25,false);
69     mary.print();
70     mary = tom ;
71     mary.print();
72 }
73 */
74 /*
75 Passing an object t by value.
76 */
77 static void student::pass_by_value(student t) {
78     t.change_age(_age + t._age) ;
79     t.print();
80 }
81 */
82 /*
83 testing pass by value
84 */
85 static void test_pass_by_value() {
86     student tom("Tom Jr",21);
87     tom.print();
88     student mary("Mary Jr",25,false);
89     mary.print();
90     tom.pass_by_value(mary) ;
91     mary.print();
92 }
93 */
94 /*
95 */
```

C:\work\alg\course\code\ch1\needforcpqq.cpp

```
96  returning an object by value.  
97  -----*/  
98 student student::return_by_value() {  
99   student t("ken",50) ;  
100  t.change_age(_age + t._age) ;  
101  t.print() ;  
102  return t ;  
103 }  
104  
105 /*-----  
106 testing return by value  
107 -----*/  
108 static void test_return_by_value() {  
109   student tom("Tom Jr",21);  
110   tom.print() ;  
111   student ken = tom.return_by_value();  
112   ken.print() ;  
113 }  
114  
115  
116  
117 /*-----  
118 THIS PROGRAM WILL CRASH  
119 -----*/  
120 int main() {  
121   test_shallow_copy_crash() ;  
122   test_shallow_equal_crash() ;  
123   test_pass_by_value();  
124   test_return_by_value();  
125   return 0 ;  
126 }  
127  
128  
129
```

```
C:\work\alg\course\code\ch1\copyeq.cpp  
1  /*-----  
2  Copyright (c) 2016 Author: Jagadeesh Vasudevamurthy  
3  Filename: copyeq.cpp  
4  -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8 #pragma warning(disable: 4996) /* Disable deprecation */  
9  
10 class book {  
11 public:  
12     book(const char *s, int c = 0, bool x = false);  
13     ~book();  
14     book(const book& b);  
15     book& operator=(const book& rhs);  
16     void print() const;  
17 private:  
18     void _copy(const book& from);  
19     char* _name;  
20     int _cost;  
21     bool _in;  
22     void _release();  
23 };  
24  
25 /*-----  
26 Constructor  
27 -----*/  
28 book::book(const char *s, int c, bool x) : _name(nullptr), _cost(c), _in(x) {  
29     cout << "In book Constructor " << s << endl;  
30     int l = strlen(s) + 1;  
31     _name = new char[l];  
32     strcpy(_name, s);  
33 }  
34  
35 /*-----  
36 Release heap memory of this class  
37 -----*/  
38 void book::_release() {  
39     delete[] _name;  
40 }  
41  
42 /*-----  
43 Destructor  
44 -----*/  
45 book::~book() {  
46     cout << "In book Destructor " << _name << endl;  
47     _release();  
48 }  
49
```

```
50  /*
51 Helper: copy function
52 */
53 void book::_copy(const book& from) {
54     int l = strlen(from._name) + 1;
55     _name = new char[l];
56     strcpy(_name, from._name);
57     _cost = from._cost;
58     _in = from._in;
59 }
60
61 /*
62 Copy constructor
63 */
64 book::book(const book& b) {
65     cout << "In copy Constructor " << b._name << endl;
66     _copy(b);
67 }
68
69 /*
70 equal operator
71 */
72 book& book::operator=(const book& rhs) {
73     cout << "In equal operator " << rhs._name << endl;
74     if (this != &rhs) {
75         _release();
76         _copy(rhs);
77     }
78     return *this;
79 }
80
81 /*
82 print
83 */
84 void book::print() const {
85     cout << "Name " << _name << " Cost " << _cost;
86     (_in) ? cout << " In " : cout << " Out ";
87     cout << endl;
88 }
89
90 /*
91 main
92 */
93 int main() {
94     book b1("algorithm");
95     book b2("C++", 79);
96     book b3("VLSI design", 168, true);
97     book b4(b2);
98     book b5 = b1;
```

C:\work\alg\course\code\ch1\copyeq.cpp

```
99    b1.print();
100   b2.print();
101   b3.print();
102   b4.print();
103   b5.print();
104   b1 = b3;
105   b2 = b4;
106   b1.print();
107   b2.print();
108   b3.print();
109   b4.print();
110   return 0;
111 }
112 /*-----
113 output of the above program
114 -----
115 */
116 In book Constructor algorithm
117 In book Constructor C++
118 In book Constructor VLSI design
119 In copy Constructor C++
120 In copy Constructor algorithm
121 Name algorithm Cost 0 Out
122 Name C++ Cost 79 Out
123 Name VLSI design Cost 168 In
124 Name C++ Cost 79 Out
125 Name algorithm Cost 0 Out
126 In equal operator VLSI design
127 In equal operator C++
128 Name VLSI design Cost 168 In
129 Name C++ Cost 79 Out
130 Name VLSI design Cost 168 In
131 Name C++ Cost 79 Out
132 In book Destructor algorithm
133 In book Destructor C++
134 In book Destructor VLSI design
135 In book Destructor C++
136 In book Destructor VLSI design
137
138 */
139
```

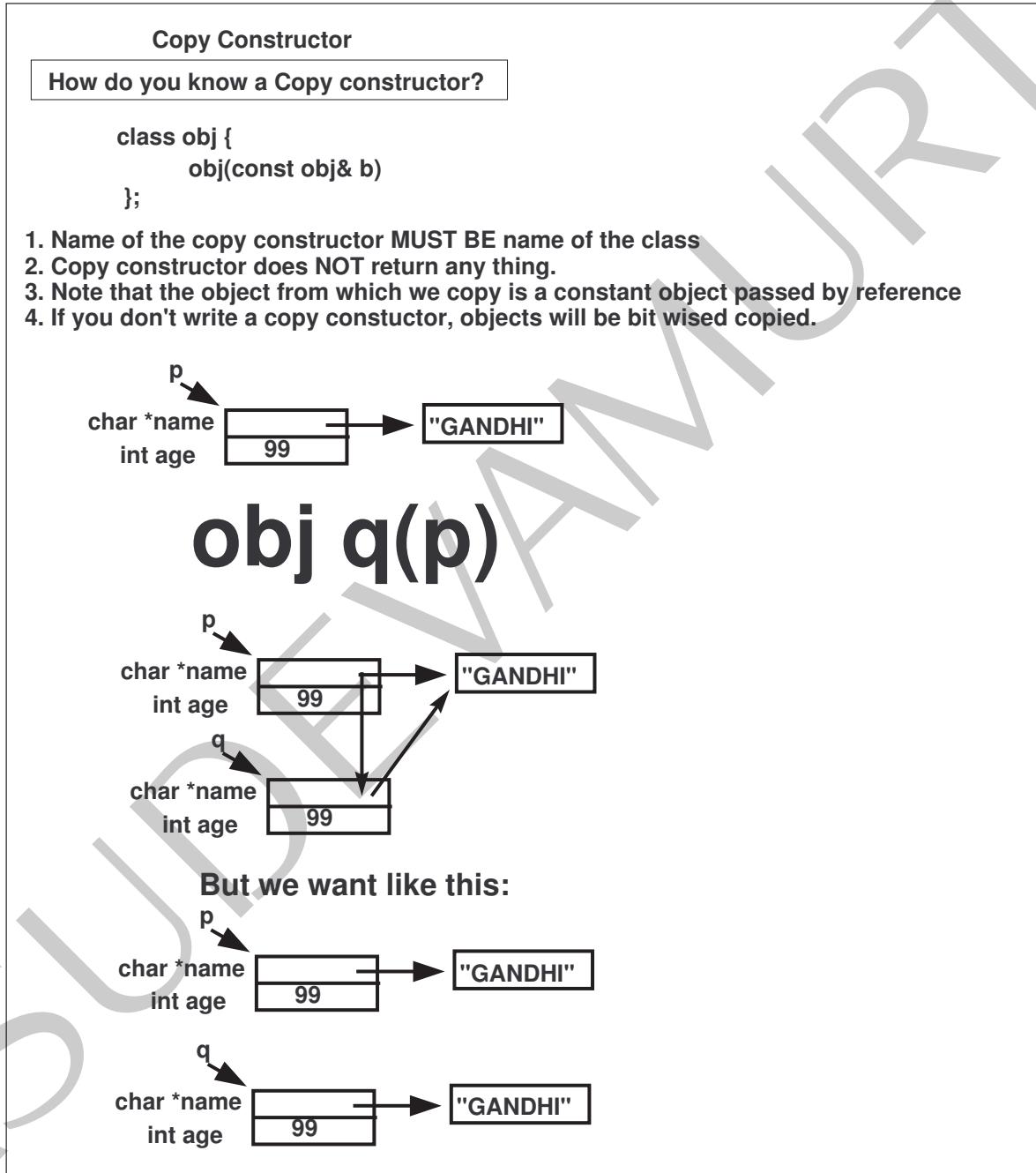


Figure 1.15: Copy constructor

1.15. CONCEPT 10: COPY CONSTRUCTOR AND EQUAL OPERATOR

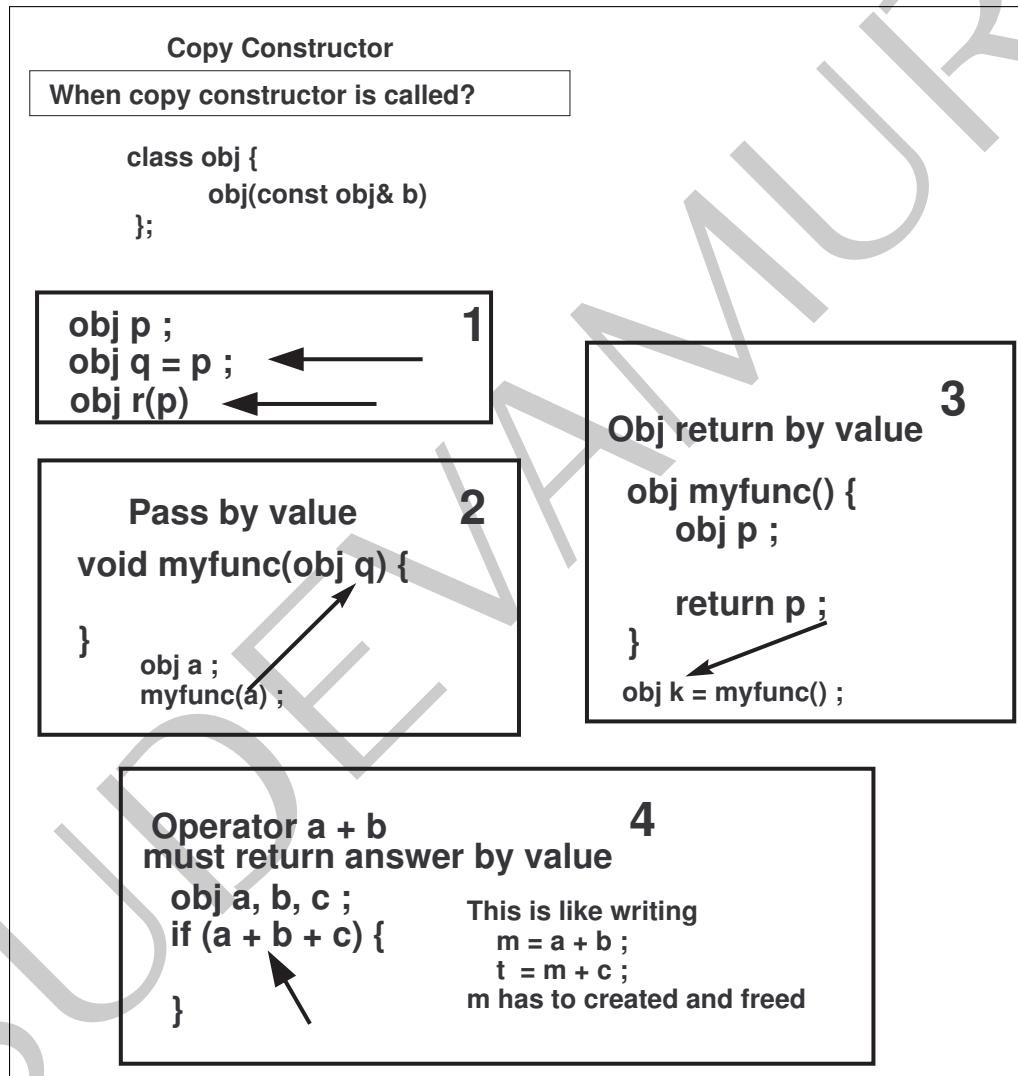


Figure 1.16: When copy constructor is called?

Difference between copy constructor and equal operator

```
class obj {
    obj(const obj& b) ;
};

obj a ;
obj b(a) ;
obj c = a ;
```

```
class obj {
    obj& operator=(const obj& b) ;
};

obj a, b ;
b = a;
```

```
void book::_release() {
    delete [] _name ;
}

void book::_copy(const book& from) {
    int l = strlen(from._name) + 1 ;
    _name = new char[l+1] ;
    strcpy(_name,from._name) ;
    _cost = from._cost ;
    _in = from._in ;
}
```

```
book::book(const book& b){
    _copy(b) ;
}
```

```
book& book::operator=(const book& rhs) {
    if (this != &rhs) {
        _release() ;
        _copy(rhs) ;
    }
    return *this ;
}
```

1. Copy constructor: No need to test to see if it is being initialized from itself.
2. Copy constructor: No need to release existing memory
3. Equal operator: A reference to the copied object is returned so that chaining can be done. EX; a = b = c = d ;

Figure 1.17: Difference between copy constructor and equal operator

1.16. NEED FOR FRIEND

1.16 Need for friend

1.16.1 friend function

```
1 /*-----
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy
3 Filename: f1.cpp
4 -----
5
6 #include <iostream>
7 using namespace std;
8
9 /*
10 class fraction:
11 -----
12 class fraction {
13 public:
14     fraction(int n = 0, int d = 1):_numerator(n),_denominator(d) {
15         cout << "In fraction constructor\n" ;
16     }
17     ~fraction() { cout << "In fraction distructor\n" ;}
18     int numerator() const { return _numerator ;}
19     int denominator() const { return _denominator ;}
20
21 private:
22     int _numerator ;
23     int _denominator;
24     fraction(const fraction& a);
25     fraction& operator=(const fraction& rhs);
26 };
27
28 /*
29 89
30 In fraction constructor
31 3/8
32 In fraction distructor
33 -----
34 int main() {
35     int y = 89 ;
36     cout << y << endl ;
37
38     fraction x(3,8) ;
39     //cout << x << endl ; -- cout does not know fraction. It knows basic types only
40     //no operator found which takes a right-hand operand of type 'fraction'
41     //cout << x._numerator ;// -- _numerator' : cannot access private member
42     cout << x.numerator() ;
43     cout << "/" ;
44     cout << x.denominator() ;
45     cout << "\n" ;
46     return 0 ;
47 }
48
```

Operator << works on basic types

```

9  /*
10 class fraction:
11 -----
12 class fraction {
13 public:
14     fraction(int n = 0, int d = 1):_numerator(n),_denominator(d) {
15         cout << "In fraction constructor\n" ;
16     }
17     ~fraction() { cout << "In fraction distructor\n" ;}
18     int numerator() const { return _numerator ;}
19     int denominator() const { return _denominator ;}
20
21 private:
22     int _numerator ;
23     int _denominator;
24     fraction(const fraction& a);
25     fraction& operator=(const fraction& rhs);
26 };
27
int y = 89 ;
cout << y << endl ;           1. cout is an object
                                2. cout has an overloaded operator
                                   cout << (basic_type)
                                   cout.operator<<(basic_type)

```

fraction x(3,8) ; This won't work because we are calling
cout << x << endl ; cout << (fraction)
cout does not know fraction

```

39 //cout << x << endl ; -- cout does not know fraction. It knows basic types only
40 //no operator found which takes a right-hand operand of type 'fraction'
41 //cout << x._numerator ;// -- _numerator' : cannot access private member
42 cout << x.numerator() ;
43 cout << "/" ;
44 cout << x.denominator() ;
45 cout << "\n" ;

```

We need to take basic types from fraction and give to cout

Figure 1.18: Using cout basic types to print

Consolidating fraction print

```

12 class fraction {
13 public:
14     fraction(int n = 0, int d = 1) : _numerator(n), _denominator(d) {
15         cout << "In fraction constructor\n" ;
16     }
17     ~fraction() { cout << "In fraction destructor\n" ;}
18     void print(ostream& o) const {
19         o << _numerator ;
20         o << "/" ;
21         o << _denominator ;
22         o << "\n" ;
23     }

```

**fraction x(3,8) ;
x.print(cout) ;**

It works, but different than basic types

```

int y = 98 ;
cout << y ;

```

**fraction x(3,8) ;
x.print(cout) ;**

Not consistent

Figure 1.19: Printing fraction object

```
1 /*-----
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy
3 Filename: f2.cpp
4 -----
5
6 #include <iostream>
7 using namespace std;
8
9 /*
10 class fraction:
11 -----
12 class fraction {
13 public:
14     fraction(int n = 0, int d = 1):_numerator(n),_denominator(d) {
15         cout << "In fraction constructor\n" ;
16     }
17     ~fraction() { cout << "In fraction distructor\n" ;}
18     void print(ostream& o) const {
19         o << _numerator ;
20         o << "/";
21         o << _denominator ;
22         o << "\n" ;
23     }
24
25 private:
26     int _numerator ;
27     int _denominator;
28     fraction(const fraction& a);
29     fraction& operator=(const fraction& rhs);
30 };
31
32 /*
33 89
34 In fraction constructor
35 3/8
36 In fraction distructor
37 -----
38 int main() {
39     int y = 89 ;
40     cout << y << endl ;
41     fraction x(3,8) ;
42     x.print(cout) ;
43     return 0 ;
44 }
45
```

Need for friend function

```

class fraction {
public:
    fraction(int n = 0, int d = 1):_numerator(n),_denominator(d) {
        cout << "In fraction constructor\n";
    }
    ~fraction() { cout << "In fraction destructor\n"; }
    void print1(ostream& o, const fraction& a) {
        o << a._numerator << "/" << a._denominator << endl;
    }
    friend void print(ostream& o, const fraction& a) {
        o << a._numerator << "/" << a._denominator << endl;
    }
    //Exactly as print. print name is replaced by operator<<
    friend void operator<<(ostream& o, fraction& a) {
        o << a._numerator << "/" << a._denominator << endl;
    }
};

instead of name: print
use: operator<<
private:
    int _numerator;
    int _denominator;
    fraction(const fraction& a);
    fraction& operator=(const fraction& rhs);
};

```

fraction x(3,8)

print1(cout,x) //What is print1? fails
x.print1(cout,x) : //Works ugly
print(cout,x) : //Works because of friend
//print is a NON member function
// Can be called WITHOUT an obj.print
//Must have obj as parameter in print function

Now everything works like basic types

```

int y = 89 ;
cout << y << endl ;

fraction x(3,8) ;
//print1(cout,x) ; //error C3861: 'print1': identifier not found
x.print1(cout,x) ; //Works, but ugly. User need to know print1
print(cout,x) ; // Works without invoking from x
                //But one of the parameter must be x
                //Still user should know function name "print"
operator<<(cout,x) ; //infix notation
cout << x ; //postfix

```

Figure 1.20: Friend function

```
1 /*-----
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy
3 Filename: f3.cpp
4 -----
5
6 #include <iostream>
7 using namespace std;
8
9 /*
10 class fraction:
11 -----
12 class fraction {
13 public:
14     fraction(int n = 0, int d = 1):_numerator(n),_denominator(d) {
15         cout << "In fraction constructor\n" ;
16     }
17     ~fraction() { cout << "In fraction distructor\n" ;}
18     void print1(ostream& o, const fraction& a) {
19         o << a._numerator << "/" << a._denominator << endl ;
20     }
21     friend void print(ostream& o, const fraction& a) {
22         o << a._numerator << "/" << a._denominator << endl ;
23     }
24     //Exactly as print. print name is replaced by operator<<
25     friend void operator<<(ostream& o, fraction& a) {
26         o << a._numerator << "/" << a._denominator << endl ;
27     }
28
29 private:
30     int _numerator ;
31     int _denominator;
32     fraction(const fraction& a);
33     fraction& operator=(const fraction& rhs);
34 };
35
36 -----
37
38 -----
39 int main() {
40     int a = 89 ;
41     cout << "a = " << a << endl ;
42     int b = 11 ;
43     cout << "b = " << b << endl ;
44     fraction x(3,8) ;
45     //print1(cout,x); //error C3861: 'print1': identifier not found
46     x.print1(cout,x); //Works, but ugly. User need to know print1
47     print(cout,x); // Without object x works. User need to know print
48     operator<<(cout,x); //infix notation
49     cout << x; //postfix
50     return 0 ;
51 }
52
```

Need for friend function

```

class fraction {
public:
    fraction(int n = 0, int d = 1): _numerator(n), _denominator(d) {
        cout << "In fraction constructor\n";
    }
    ~fraction() { cout << "In fraction destructor\n"; }
    void print1(ostream& o, const fraction& a) {
        o << a._numerator << "/" << a._denominator << endl;
    }
    friend void print(ostream& o, const fraction& a) {
        o << a._numerator << "/" << a._denominator << endl;
    }
    friend ostream& operator<<(ostream& o, const fraction& a) {
        o << a._numerator << "/" << a._denominator;
        return o;
    }
};

fraction x(3,8)
Non member function
Chaining
cout << x ;
operator<<(cout,x) ;
Must have fraction as a parameter
print1(cout,x) //What is print1? fails
x.print1(cout,x) : //Works ugly
print(cout,x) ; //Works because of friend
//print is a NON member function
// Can be called WITHOUT an obj.print
//Must have obj as parameter in print function

```

Note that we are invoking this function by cout(like print above)
and NOT by x. x is a parameter to cout function

Now everything works like basic types

```

int y = 98 ;
cout << y ;

fraction x(3,8) ;
fraction z(88,99) ;
cout << "x = " << x << " z = " << z << endl ;

```

Figure 1.21: Friend function with chaining of cout

```
1 /*-----
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy
3 Filename: f4.cpp
4 -----
5
6 #include <iostream>
7 using namespace std;
8
9 /*
10 class fraction:
11 -----
12 class fraction {
13 public:
14     fraction(int n = 0, int d = 1):_numerator(n),_denominator(d) {
15         cout << "In fraction constructor\n" ;
16     }
17     ~fraction() { cout << "In fraction distructor\n" ;}
18     void print1(ostream& o, const fraction& a) {
19         o << a._numerator << "/" << a._denominator << endl ;
20     }
21     friend void print(ostream& o, const fraction& a) {
22         o << a._numerator << "/" << a._denominator << endl ;
23     }
24
25     friend ostream& operator<< (ostream& o, const fraction& a) {
26         o << a._numerator << "/" << a._denominator;
27         return o ;
28     }
29
30 private:
31     int _numerator ;
32     int _denominator;
33     fraction(const fraction& a);
34     fraction& operator=(const fraction& rhs);
35 };
36
37 -----
38
39 -----
40 int main() {
41     int y = 89 ;
42     cout << y << endl ;
43
44     fraction x(3,8) ;
45     //print1(cout,x); //error C3861: 'print1': identifier not found
46     x.print1(cout,x) ;
47     print(cout,x) ;
48     cout << x << endl ;
49     operator<<(cout,x) ;
50     fraction z(88,99) ;
51     cout << "x = " << x << " z = " << z << endl ;
52     return 0 ;
53 }
```

Friend function

A non member function of the class

A function declared as friend in a class so that it has the same access as the class' members without having to be within the scope of the class.

Declaration

```
class cost {
    friend ostream& operator<<(ostream& o, const cost& c);
private:
    int _base;
};
```

Definition

```
ostream& operator<<(ostream& o, const cost& c){
    o << " base = " << c._base << endl;
    return o;
}
```

Usage

```
cost c;
cout << c;
operator<<(cout, c);
```

Note that this function is invoked by instance of cout and not by an instance of the user-defined class, cost

That means the function << must be declared as a binary friend of the class cost

1. The keyword friend is placed only in the function declaration of the friend function and not in the function definition.
2. A friend function, even though it is not a member function, would have the rights to access the private members of the class.
3. The function can be invoked without the use of an object
4. Friends are non-members hence do not get "this" pointer

Figure 1.22: Friend function

1.16. NEED FOR FRIEND

1.16.2 friend class

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy  
3 Filename: friendclass.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 class patient ;  
10  
11 class data {  
12     int _age ;  
13     int _sugar ;  
14     int _bp ;  
15     friend class patient ; // Try commenting this line  
16 };  
17  
18 class patient {  
19 public:  
20     patient(int a, int s = 0, int b = 0);  
21     ~patient() ;  
22     void print(const char* p) const ;  
23  
24 private:  
25     data _d;  
26 };  
27  
28 /*-----  
29 Constructor  
30 -----*/  
31 patient::patient(int a, int s, int b){  
32     //Without friend class patient ;  
33     //error C2248: 'data::_sugar' : cannot access private member declared in class 'data'  
34     _d._age = a ;  
35     _d._sugar = s ;  
36     _d._bp = b ;  
37     cout << "In patient constructor\n" ;  
38 }  
39  
40 /*-----  
41 Destructor  
42 -----*/  
43 patient::~patient(){  
44     cout << "In patient Destructor \n";  
45 }  
46  
47 /*-----  
48 print  
49 -----*/  
50 void patient::print(const char *s) const {  
51     cout << s << " age = " << _d._age << " sugar = " << _d._sugar << " bp = " << _d._bp << endl ;  
52 }  
53  
54 /*-----  
55 main
```

```
56 -----*/
57 int main() {
58     patient b1(25,90,118);
59     b1.print("b1");
60     return 0;
61 }
62 /*-----*
63 output of the above program
64 -----*/
65 /*
66 In patient constructor
67 b1 age = 25 sugar = 90 bp = 118
68 In patient Destructor
69
70 */
71
```

1.17 Concept 12: Function templates

```
1 /*-
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy
3 Filename: ftemp.cpp
4 */
5
6 #include <iostream>
7 using namespace std;
8
9 /*
10 int minimum
11 */
12 static void minimum(int i, int j) {
13     cout << " minimum(int i, int j)" << endl ;
14     if (i < j) {
15         cout << i << " is less than " << j << endl ;
16     }else {
17         cout << i << " is greater than or equal " << j << endl ;
18     }
19 }
20
21 /*
22 float minimum
23 */
24 static void minimum(double i, double j) {
25     cout << " minimum(double i, double j)" << endl ;
26     if (i < j) {
27         cout << i << " is less than " << j << endl ;
28     }else {
29         cout << i << " is greater than or equal " << j << endl ;
30     }
31 }
32
33 /*
34 ch minimum
35 */
36 static void minimum(char i, char j) {
37     cout << " minimum(char i, char j)" << endl ;
38     if (i < j) {
39         cout << i << " is less than " << j << endl ;
40     }else {
41         cout << i << " is greater than or equal " << j << endl ;
42     }
43 }
44
45 /*
46 string minimum
47 */
48 static void minimum(const char* i, const char* j) {
49     cout << " minimum(const char* i, const char* j)" << endl ;
50     int x = strcmp(i,j);
51     if (x < 0)
52         cout << i << " is less than " << j << endl ;
53     else if (x == 0){
54         cout << i << " is equal to " << j << endl ;
55     }else {
```

```
56 cout << i << " is greater than " << j << endl ;
57 }
58 }
59
60 /*-
61 template minimum
62 -----
63 template <typename T>
64 static void tminimum(T i, T j){
65 cout << "tminimum(T i, T j)" << endl ;
66 if (i < j) {
67 cout << i << " is less than " << j << endl ;
68 }else {
69 cout << i << " is greater than or equal " << j << endl ;
70 }
71 }
72
73 /*-
74 string minimum
75 -----
76 static void tminimum(const char* i, const char* j){
77 cout << "tminimum(const char* i, const char* j)" << endl ;
78 int x = strcmp(i,j);
79 if (x < 0)
80 cout << i << " is less than " << j << endl ;
81 else if (x == 0) {
82 cout << i << " is equal to " << j << endl ;
83 }else {
84 cout << i << " is greater than " << j << endl ;
85 }
86 }
87
88 /*-
89 template assignment
90 -----
91 template <typename T>
92 static void print_array(const char *s, const T& a, int size){
93 cout << s << "{";
94 for (int i = 0; i < size; i++) {
95 cout << a[i];
96 if (i < size - 1) {
97 cout << ",";
98 }
99 }
100 cout << "}" << endl ;
101 }
102
103 /*-
104 assignment
105 -----
106 static void assignment(){
107 int a[] = { 1, 2, 3, 4, 5 };
108 print_array("a is ",a,sizeof(a)/sizeof(int));
109 double b[] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
110 print_array("b is ",b,sizeof(b)/sizeof(double));
```

```
111 char c[] = "HELLO";
112 print_array("c is ",c,sizeof(c)/sizeof(char)):
113 }
114
115 /*
116 main
117 -----
118 int main() {
119 minimum(47,90);
120 minimum(47.9,90.8);
121 minimum('a','b');
122 minimum("jag","jag");
123 minimum("jaf","jag");
124 minimum("jah","jag");
125 tminimum(47,90);
126 tminimum(47.9,90.8);
127 tminimum('a','b');
128 tminimum("jag","jag");
129 tminimum("jaf","jag");
130 tminimum("jah","jag");
131 assignment();
132 return 0;
133 }
134
135 /*
136 output of the above program
137 -----
138 /*
139 minimum(int i, int j)
140 47 is less than 90
141 minimum(double i, double j)
142 47.9 is less than 90.8
143 minimum(char i, char j)
144 a is less than b
145 minimum(const char* i, const char* j)
146 jag is equal to jag
147 minimum(const char* i, const char* j)
148 jaf is less than jag
149 minimum(const char* i, const char* j)
150 jah is greater than jag
151 tminimum(T i, T j)
152 47 is less than 90
153 tminimum(T i, T j)
154 47.9 is less than 90.8
155 tminimum(T i, T j)
156 a is less than b
157 tminimum(const char* i, const char* j)
158 jag is equal to jag
159 tminimum(const char* i, const char* j)
160 jaf is less than jag
161 tminimum(const char* i, const char* j)
162 jah is greater than jag
163 a is {1, 2, 3, 4, 5}
164 b is {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7}
165 c is {H, E, L, L, O, }
```

1.18 Problem set

Problem 1.18.1. Write a program that computes the number of dollars, quarters, dimes, nickels and pennies that are contained with a specified amount of money. For example 37.79\$ should be written as **37 dollars, 3 quarters and 4 pennies** 0.31\$ dollars should be written as **1 quarter, 1 nickel and 1 penny**. Note that your program should have correct usage of singular and plurals. For example: **penny** and **pennies**.

Test your program on the following data inputs. The output should look as follows:

```
Enter Money: hsdh
You entered garbage: hsdh
Enter Money: 6.7
You entered: 6.7
6 dollars, 2 quarters and 2 dimes
Enter Money: 6.80
You entered: 6.80
6 dollars, 3 quarters and 1 nickel
Enter Money: 6.08
You entered: 6.08
6 dollars, 1 nickel and 3 pennies
Enter Money: 6.008
You entered: 6.008
Neglected changes less than 1 cents
6 dollars
Enter Money: 6.00007
You entered: 6.00007
Neglected changes less than 1 cents
6 dollars
Enter Money: .7
You entered: .7
2 quarters and 2 dimes
Enter Money: .1
You entered: .1
1 dime
Enter Money: .01
You entered: .01
1 penny
Enter Money: .001
You entered: .001
```

1.18. PROBLEM SET

Neglected changes less than 1 cents
Enter Money: 8.9.0
You entered garbage: 8.9.0
Enter Money: 89.79
You entered: 89.79
89 dollars, 3 quarters and 4 pennies
Enter Money: 89.71
You entered: 89.71
89 dollars, 2 quarters, 2 dimes and 1 penny
Enter Money: 89.07
You entered: 89.07
89 dollars, 1 nickel and 2 pennies
Enter Money: 89.01
You entered: 89.01
89 dollars and 1 penny
Enter Money: 4.5a
You entered garbage: 4.5a
Enter Money: 4.56
You entered: 4.56
4 dollars, 2 quarters, 1 nickel and 1 penny
Enter Money: 4.57
You entered: 4.57
4 dollars, 2 quarters, 1 nickel and 2 pennies
Enter Money: a.89
You entered garbage: a.89
Enter Money: 67
You entered: 67
67 dollars
Enter Money: 6.789
You entered: 6.789
Neglected changes less than 1 cents
6 dollars, 3 quarters and 3 pennies
Enter Money: 77.74
You entered: 77.74
77 dollars, 2 quarters, 2 dimes and 4 pennies
Enter Money: 77.73
You entered: 77.73
77 dollars, 2 quarters, 2 dimes and 3 pennies
Enter Money: 77.72
You entered: 77.72
77 dollars, 2 quarters, 2 dimes and 2 pennies

CHAPTER 1. BASIC C++

```
Enter Money: 77.71
You entered: 77.71
77 dollars, 2 quarters, 2 dimes and 1 penny
Enter Money: 77.0089
You entered: 77.0089
Neglected changes less than 1 cents
77 dollars
Enter Money: 77.01
You entered: 77.01
77 dollars and 1 penny
Enter Money:
```

Problem 1.18.2.

Amicable pairs

Consider the number 220

Find the sum of factors, except the number itself
(also called **proper divisors**)

$$\begin{aligned} 220 &= 1+2+4+5+10+11+20+22+44+55+110 \\ &= 284 \end{aligned}$$

Now find the sum of factors for 284

$$\begin{aligned} 284 &= 1+2+4+71+142 \\ &= 220 \end{aligned}$$

220 and 284 are called the amicable pairs

Write a program that prints all the amicable numbers.
You must write amicable.h and amicable.cpp
amicabletest.cpp is given below

FILE: amicablenumbers.cpp

```
#include "amicable.h"
int main() {
    const int MAX = 100000000;
    clock_t start = clock();
    amicable a(MAX);
    clock_t end = clock();
    cout << "Run time for amicable " << double(end - start)/CLOCKS_PER_SEC <<
        " secs" << endl;
}
```

objects
 amicable
 amicable.h
 amicable.cpp
 amicablenumbers.cpp

Directory

Figure 1.23: Amicable numbers

Amicable pairs MAX = 100000000

The following are amicable numbers

0: 220 and 284
1: 1184 and 1210
2: 2620 and 2924
3: 5020 and 5564
4: 6232 and 6368
5: 10744 and 10856
6: 12285 and 14595
7: 17296 and 18416
8: 63020 and 76084
9: 66928 and 66992
10: 67095 and 71145
11: 69615 and 87633
12: 79750 and 88730
13: 100485 and 124155
14: 122265 and 139815
15: 122368 and 123152
16: 141664 and 153176
17: 142310 and 168730
18: 171856 and 176336
19: 176272 and 180848
20: 185368 and 203432
21: 196724 and 202444
22: 280540 and 365084
23: 308620 and 389924
24: 319550 and 430402

204: 66854710 and 71946890
205: 67729064 and 69439576
206: 67738268 and 79732132
207: 68891992 and 78437288
208: 71015260 and 85458596
209: 71241830 and 78057370
210: 72958556 and 74733604
211: 73032872 and 78469528
212: 74055952 and 78166448
213: 74386305 and 87354495
214: 74769345 and 82824255
215: 75171808 and 77237792
216: 75226888 and 81265112
217: 78088504 and 88110536
218: 78447010 and 80960990
219: 79324875 and 87133365
220: 80422335 and 82977345
221: 83135650 and 85603550
222: 84591405 and 89590995
223: 86158220 and 99188788
224: 89477984 and 92143456
225: 90437150 and 94372450
226: 91996816 and 93259184
227: 93837808 and 99899792
228: 95629904 and 97580944
229: 96304845 and 96747315
230: 97041735 and 97945785

Run time for n = 100000000 is 50.09 secs

Can you beat me??

Figure 1.24: Amicable numbers

1.18. PROBLEM SET

Problem 1.18.3. Write a program that prints truth table of n inputs, as shown in the figure .

TRUTH TABLE OF 'n' inputs

$n = 3$

000
001
010
011
100
101
110
111

Create a folder under objects

objects

 truthtable
 truthtable.h
 truthtable.cpp
 truthtabletest.cpp

Write truthtable.h and truthtable.cpp

You must use the following main program in truthtabletest.cpp

```
#include "truthtable.h"
int main() {
    int n = 4;
    clock_t start = clock();
    truthtable a(4);
    clock_t end = clock();
    double d = double(end - start) / CLOCKS_PER_SEC;
    cout << "Run time for truthtable for n = " << n << " is " << d << " secs" << endl;
    return 0 ;
}
```

Test your program for $n = 1, 2, 3, 4 \dots 20$
and record the CPU time

Will this world exist for $n = 64$?

Figure 1.25: Printing truth tables

CHAPTER 1. BASIC C++

Problem 1.18.4. Write the **templated** version of the procedure **print_array**. Use the **main** shown below.

```
int main() {
    int a[] = { 1, 2, 3, 4, 5 };
    print_array("a is ",a,sizeof(a)/sizeof(int)) ;
    double b[] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 } ;
    print_array("b is ",b,sizeof(b)/sizeof(double)) ;
    char c[] = "HELLO";
    print_array("c is ",c,sizeof(c)/sizeof(char)) ;
    return 0 ;
}
```

The output of the program should be as follows:

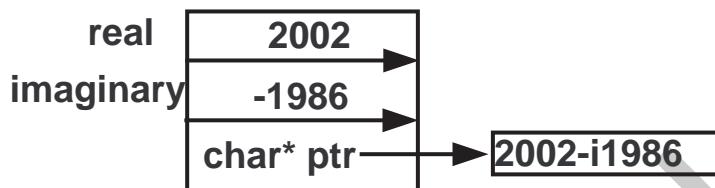
```
a is {1, 2, 3, 4, 5 }
b is {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 }
c is {H, E, L, L, O, }
```

1.18. PROBLEM SET

Problem 1.18.5. Write a class called as *complex* as shown in the figure 1.26

```
1 /*-----  
2 Copyright (c) 2014 Author: Jagadeesh Vasudevamurthy  
3 Filename: complextest.cpp  
4 compile: g++ complex.cpp complextest.cpp  
5 ==27185== All heap blocks were freed -- no leaks are possible  
6 -----*/  
7 #include "complex.h"  
8  
9 /*-----  
10 main  
11 -----*/  
12 int main() {  
13     complex c1(2,3) ;  
14     cout << c1 << endl ;  
15     complex c2(27,-200) ;  
16     cout << c2 << endl ;  
17     complex c3(-20,4) ;  
18     cout << c3 << endl ;  
19     complex c4(-18, -99) ;  
20     cout << c4 << endl ;  
21     c2.setxy(2,3) ;  
22     cout << c2 << endl ;  
23     if (c1 == c2) {  
24         cout << "c1 is equal to c2" << endl ;  
25     }else {  
26         cout << "c1 is NOT equal to c2" << endl ;  
27     }  
28     if (c1 != c3) {  
29         cout << "c1 is NOT equal to c3" << endl ;  
30     }else {  
31         cout << "c1 is equal to c3" << endl ;  
32     }  
33  
34     complex *c5 = new complex(-200,-800) ;  
35     cout << *c5 << endl ;  
36     delete c5 ;  
37     c1 = c2 = c3 = c4 ;  
38     cout << c3 << endl ;  
39     return 0 ;  
40 }  
41
```

Implementing complex class



1. create a class called **complex** that has
 1. Real number, x
 2. Imaginary number, y
 3. construct a name $x+iy$ on heap and store the pointer.
YOU CANNOT USE STL string
2. You must create 2 files: **complex.h**, **complex.cpp**
I will provide **complextest.cpp**, which you should use as is.
3. You must use only **util.h** in **complex.h**
4. You should be able to construct complex objects using 3 ways
`complex h ; //In that case x = 0, y = 0`
`complex h(5) ; //In that case x = 5, y = 0 ;`
`complex h(5,-8) ; //In that case x = 5 and y = -8`
5. Write all the necessary functions so that **complextest.cpp** works with no assertion failure or memory leak
6. e-mail 2 files (**complex.h** and **complex.cpp**). Include screen shot of the output as a pdf file

Figure 1.26: Complex number class

CHAPTER 1. BASIC C++

Problem 1.18.6. This is an assignment problem for my 3rd grade daughter at Tom Matsumoto Elementary School, San Jose, CA.

Problem: The cost of each letter is given in the table below.
Find a SINGLE LEGEAL WORD (that must be in a dictionary) such that the total cost of the word is EXACTLY 25\$.

Example:

```
begin 5.17  
beginner 7.42  
beginners 7.57
```

The cost of each alphabet is as follows:

a	= 1.00\$
b	= 0.01\$
c	= 0.02\$
d	= 0.03\$
e	= 2.00\$
f	= 0.04\$
g	= 0.05\$
h	= 0.06\$
i	= 3.00\$
j	= 0.07\$
k	= 0.08\$
l	= 0.09\$
m	= 0.10\$
n	= 0.11\$
o	= 4.00\$
p	= 0.12\$
q	= 0.13\$
r	= 0.14\$
s	= 0.15\$
t	= 0.16\$
u	= 5.00\$
v	= 0.17\$
w	= 0.18\$
x	= 0.19\$
y	= 6.00\$
z	= 0.20\$

1.18. PROBLEM SET

Problem 1.18.7. Long Multiplication:

LONG MULTIPLICATION (Assume max of 150 digits)

```
37975227936943673922808872755445627854565536638199 X
 40094690950920881030683735292761468389214899724061
 ==
15226050279225333605356183781326374297180681149
61380688657908494580122963258952897654000350692006139
```

1. Assume that number of digit <= 150. You are free to use char x[150] ;
Use static array. No dynamic memory management is required
2. The main program is in the file mult150test.cpp. You cannot change any thing in that file
3. Create a file called mult150.h. Declare a class called mult150. You can write only the
functions that are required to compile the code in mult150test.cpp
4. You should write all the guts of the function in the file: mult150.cpp
5. You cannot use any STL objects. You should use only basic built in type like
bool, int, char, char[150] in the class definition
6. I am not looking for a fast multiplication algorithm.
You can use elementary school paper pencil method of multiplication.

23958233
5830 x

00000000 (= 23,958,233 x 0)
71874699 (= 23,958,233 x 3)
191665864 (= 23,958,233 x 8)
119791165 (= 23,958,233 x 5)

139676498390 (= 139,676,498,390)

mult150 a("23958233","5830")
User calls like this
The object mult will
have x,y, and answer z
User can then print
cout << a << endl ;

7 Compile mult150.h, mult150.cpp and multtest150.cpp and run
8. The program must NOT crash. YOU CANNOT REMOVE ANY ASSERT.
9. There should be NO memory leak
10. Attach the screen shot of your program.
11. No marks will be given if program crash. E-mail only mult150.h, mult150.cpp and screen shot
12. Print how many basic multiplication and addition that were used to solve.

Figure 1.27: Long multiplication

```

1  /*
2  Copyright (c) 2016 Author: Jagadeesh Vasudevamurthy
3  file: mult150test.cpp
4
5  On linux:
6  g++ mult150.cpp mult150test.cpp
7  valgrind a.out
8  ==30529== All heap blocks were freed -- no leaks are possible
9  */
10
11 /*
12 This file test mult150 object
13 */
14
15 /*
16 All includes here
17 */
18 #include "mult150.h"
19
20 /*
21 test bench
22 */
23 void test_bench() {
24     mult150 t("5", "19");
25     cout << t << endl;
26     assert(t == "95");
27
28     mult150 t1("51", "19");
29     cout << t1 << endl;
30     assert(t1 == "969");
31
32     mult150 rsa100("37975227936943673922808872755445627854565536638199",
33                   "40094690950920881030683735292761468389214899724061");
34     cout << rsa100 << endl;
35     assert(rsa100 ==
36            "1522605027922533360535618378132637429718068114961380688657908494580122963258");
37     assert(rsa100 ==
38            "952897654000350692006139");
39
40     mult150 rsa110("6122421090493547576937037317561418841225758554253106999",
41                   "5846418214406154678836553182979162384198610505601062333");
42     cout << rsa110 << endl;
43     assert(rsa110 ==
44            "3579423417972586877499180783256845540300377802422822619353290819048467025236");
45     assert(rsa110 ==
46            "4677411513516111204504060317568667");
47
48     mult150 rsa129
49     ("3490529510847650949147849619903898133417764638493387843990820577",
50      "32769132993266709549961988190834461413177642967992942539798288533");
51     cout << rsa129 << endl;

```

```
C:\work\alg\course\objects\mult150\mult150test.cpp
2
42 assert(rsa129 ==
    "1143816257578888676692357799761466120102182967212423625625618429357069352457"
    "33897830597123563958705058989075147599290026879543541");
43 }
44
45
46 /*-----*/
47 main
48 -----*/
49 int main() {
50     clock_t start = clock();
51     test_bench();
52     clock_t end = clock();
53     double d = double(end - start) / CLOCKS_PER_SEC;
54     cout << "Run time " << " is " << d << " secs" << endl;
55     return 0;
56 }
57
58
59 //EOF
60
61
```

Problem 1.18.8. GOOGLE interview question:

Write a routine to draw a circle ($x^2 + y^2 = r^2$) without making use of any floating point computations at all.

Problem 1.18.9. GOOGLE interview question:

Give a one-line C expression to test whether a number is a power of 2. [No loops allowed – it's a simple test.]

Problem 1.18.10. GOOGLE interview question:

Give a very good method to count the number of ones in a 32 bit number. (caution: looping through testing each bit is not a solution).

Problem 1.18.11. GOOGLE interview question:

Switch the integer values stored in two registers without using any additional memory.

Problem 1.18.12. MICROSOFT interview question:

Devise an algorithm for detecting whether a given string is a palindrome (spelled the same way forwards and backwards). For example, "A man, a plan, a canal, Panama."

Problem 1.18.13. MICROSOFT interview question:

Count the number of set bits in a number without using a loop.

Problem 1.18.14. MICROSOFT interview question:

Write a routine to reverse a series of numbers without using an array.

VASUDEVAMURTHY

Chapter 2

Analysis of Algorithms

2.1 Introduction

2.1. INTRODUCTION

Algorithm

Algorithm is an effective method for solving a problem expressed as a finite sequence of instructions

Mohammed al-Khowarizmi



Algorithm LargestNumber

Input: A List L that has N numbers and $N \geq 1$
Output: The largest number in the list L.

```
largest = L[0]
for (i = 1 to N-1) {
    if (L[i] > largest)
        largest = item
}
return largest
```

- 1. Precise
- 2. Unambiguous
- 3. Mechanical
- 4. Efficient
- 5. Correct

- 1. Is it correct ?
- 2. How much time does it take, in terms of n
- 3. And can we do better?

Figure 2.1: Definition of algorithm

Complexity Theory

- Even if a problem is computationally solvable in principle, it may not be solvable in practice if the solution requires an inordinate amount of time or resources.



- **Complexity theory** deals with how hard/easy it is to solve computational problems.
- **Time** versus **space** complexity.

Figure 2.2: Complexity theory

2.2 Constant Behavior algorithms: $f(n) = K$

```
static void lessthan(int x, int y) {  
    bool r = (x < y) ? true:false ;  
    if (r) {  
        cout << x << " is less than " << y << endl ;  
    }else {  
        cout << x << " is NOT less than " << y << endl ;  
    }  
}
```

Figure 2.3: Finding smaller of two integers

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy  
3 Filename: o1.cpp  
4 -----*/  
5 #include <iostream>  
6 using namespace std;  
7  
8 /*-----  
9 O(1) algorithm  
10 -----*/  
11 static void lessthan(int x, int y) {  
12     bool r = (x < y) ? true:false ;  
13     if (r) {  
14         cout << x << " is less than " << y << endl ;  
15     }else {  
16         cout << x << " is NOT less than " << y << endl ;  
17     }  
18 }  
19  
20 /*-----  
21 -----*/  
22 -----*/  
23 int main() {  
24     lessthan(20,30) ;  
25     lessthan(30000, 29999) ;  
26     return 0 ;  
27 }  
28
```

2.3 Linear algorithms: $f(n) = n$

```
static int find1(int n, int* s, int tofind) {
    int j = 0 ;
    while (j < n) {
        if (s[j] == tofind) {
            return j;
        }else {
            j++ ;
        }
    }
    return -1;
}
```

```
static int find2(int n, int* s, int tofind) {
    assert(s[n] == tofind);
    int j = 0 ;
    while (1) {
        if (s[j] == tofind) {
            return j;
        }else {
            j++ ;
        }
    }
    assert(0); /* you can never come here */
}
```

Figure 2.4: Linear search

```
1 -----
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy
3 Filename: o2.cpp
4 -----*/
5 #include <iostream>
6 using namespace std;
7 #include <assert.h>
8
9 const int N = 10000 ;
10
11 -----
12 Init array in ascending order
13 -----*/
14 static void init(int k, int *s){
15     for (int i = 0 ; i < k ; i++){
16         s[i] = i ;
17     }
18 }
19
20 -----
21 Worst case is n. Two comparisions
22 -----*/
23 static int find1(int n, int* s, int tofind) {
24     int j = 0 ;
25     while (j < n) {
26         if (s[j] == tofind) {
27             return j;
28         }else {
29             j++ ;
30         }
31     }
32     return -1;
33 }
34
35 -----
36 Worst case is n. One comparision. 50% improvement
37 -----*/
38 static int find2(int n, int* s, int tofind) {
39     assert(s[n] == tofind);
40     int j = 0 ;
41     while (1) {
42         if (s[j] == tofind) {
43             return j;
44         }else {
45             j++ ;
46         }
47     }
48     assert(0); /* you can never come here */
49 }
50
51 -----
52 O(n) algorithm
53 -----*/
54 static void m1(int tofind) {
55     int s[N] ;
```

```
56 init(N,s);
57 int k = find1(N,s,tofind);
58 if (k == -1){
59     cout << "m1: NOT Found " << tofind << " after " << N << " iterations\n";
60 }else{
61     cout << "m1: Found " << tofind << " after " << k << " iterations\n";
62 }
63 }
64
65 /*-
66 O(n) algorithm 50% improved
67 -----*/
68 static void m2(int tofind){
69     int s[N+1];
70     init(N,s);
71     s[N] = tofind; //sentinal
72     int k = find2(N,s,tofind);
73     if (k == N){
74         cout << "m2: NOT Found " << tofind << " after " << N << " iterations\n";
75     }else{
76         cout << "m2: Found " << tofind << " after " << k << " iterations\n";
77     }
78 }
79
80 /*-
81 Main program
82 -----*/
83 int main(){
84     m1(567);
85     m1(N+67);
86     m2(567);
87     m2(N+67);
88     return 0;
89 }
90
91 /*-
92 Output of the above program
93 -----*/
94 /*
95 m1: Found 567 after 567 iterations
96 m1: NOT Found 10067 after 10000 iterations
97 m2: Found 567 after 567 iterations
98 m2: NOT Found 10067 after 10000 iterations
99
100 */
101
102
```

2.4 Logarithmic algorithms: $f(n) = \log n$

log n Behavior

A bar attender offers 10000\$ bet. If you win, you get 10000\$ or you should pay him 10000\$

You choose a number between 1 to 1,000,000. Bar attender will tell that number in 20 guesses.

After each guess, you should tell bar attender:

1. TOO HIGH
2. TOO LOW
3. YOU(bar attender) are right.

If bar attender cannot get your chosen number in 20 guesses, you will win 10000\$. Other wise, you have to pay 10000\$ to the bar attender.

Will you take that bet ?

How many **MAXIMUM** guesses are required to answer a number between 1-10 ?

$$4 = \log_2 10$$

In general $\log_2 N$ guesses

$$\log_2 1000000 = 19.93 = 20 \text{ guesses}$$

$$2^{20} = 1048576$$

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy  
3 Filename: barattender.cpp  
4 -----*/  
5 #include <iostream>  
6 using namespace std;  
7 #include <assert.h>  
8  
9 /*-----  
10 Enter a number between 1 to 1000000: 9766  
11 Bar attender: Your number is: 500000  
12 You : TOO HIGH  
13 Bar attender: Your number is: 250000  
14 You : TOO HIGH  
15 Bar attender: Your number is: 125000  
16 You : TOO HIGH  
17 Bar attender: Your number is: 62500  
18 You : TOO HIGH  
19 Bar attender: Your number is: 31250  
20 You : TOO HIGH  
21 Bar attender: Your number is: 15625  
22 You : TOO HIGH  
23 Bar attender: Your number is: 7813  
24 You : TOO LOW  
25 Bar attender: Your number is: 11719  
26 You : TOO HIGH  
27 Bar attender: Found 9766 in 9 trials  
28 -----*/  
29 static void guess(int findit,int low, int high) {  
30     int num_trail = 0 ;  
31     while (1) {  
32         int mid = (low + high)/2 ;  
33         num_trail++ ;  
34         if (mid == findit) {  
35             cout << "Bar attender: Found " << findit << " in " << num_trail << " trials\n" ;  
36             return ;  
37         }  
38         cout << "Bar attender: Your number is: " << mid << endl ;  
39         if (findit < mid) {  
40             cout << "You : TOO HIGH\n" ;  
41             high = mid ;  
42         }else {  
43             cout << "You : TOO LOW\n" ;  
44             low = mid ;  
45         }  
46     }  
47 }  
48  
49 /*-----  
50 Main program  
51 -----*/  
52 int main() {  
53     const int N = 1000000;  
54     int k = -1 ;  
55     do {  
56         cout << "Enter a number between 1 to " << N << ": " ;  
57         cin >> k ;  
58         if (k > 0 && k <= N){  
59             guess(k,1,N) ;  
60         }  
61     }while ((k > 0 && k <= N)) ;  
62     return 0 ;  
63 }
```

```
int iter = 0 ;
for (int i = 1; i < 1000; i = i * 2) {
    iter++ ;
    some_func() ;
}
```

```
int iter = 0 ;
for (int j = 1000; j >= 1 ; j = j / 2) {
    iter++ ;
    some_func() ;
}
```

iter	value of i	value of j
1	1	1000
2	2	500
3	4	250
4	8	125
5	16	62
6	32	31
7	64	15
8	128	7
9	256	3
10	512	1
exit	1024	0

$$f(n) = \log_2 n$$

Figure 2.6: Logarithmic loops

2.4. LOGARITHMIC ALGORITHMS: $F(N) = \log N$

◆ Example: `findElement(7)`

```

int l = 0 ;
int h = N ;
int ittr = 0 ;
while (1) {
    int m = (l + h) / 2 ;
    cout << "l = " << l << " h = " << h << " m = " << m << endl ;
    if (s[m] == tofind) {
        cout << "Found " << tofind << " after " << ittr << " iterations at location " << m << endl ;
        break ;
    }
    if (l >= h) {
        cout << "NOT Found " << tofind << " after " << ittr << " iterations\n" ;
        break ;
    }
    ittr++ ;
    if (s[m] < tofind) {
        l = m+1 ;
    } else {
        h = m-1 ;
    }
}

```

log n algorithm

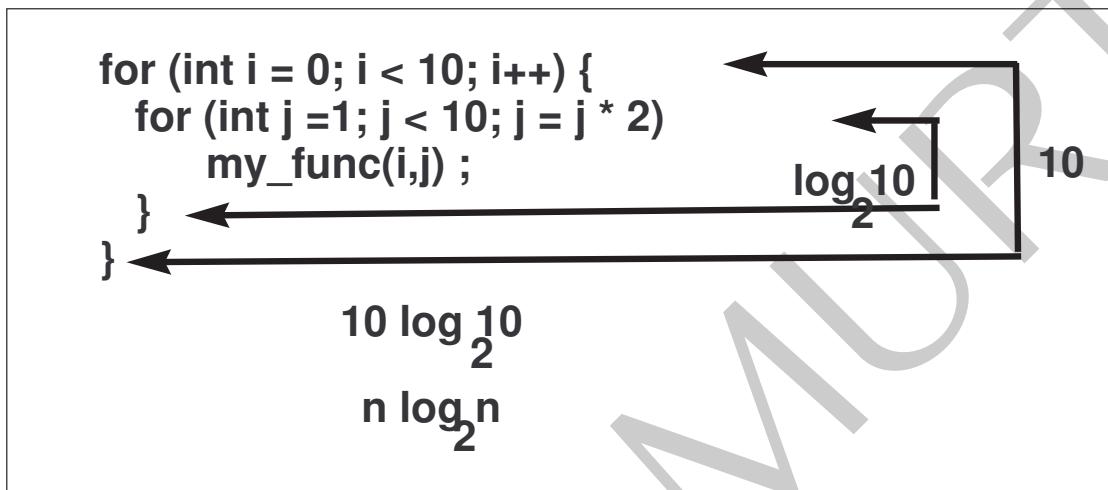
N	1+log ₂ N
10	4
100	7
1000	10
1 million	20
1 billion	30
1 billion billion	60

Figure 2.7: Binary search

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy  
3 Filename: binary_search.cpp  
4 -----*/  
5 #include <iostream>  
6 using namespace std;  
7 #include <assert.h>  
8  
9 /*-----  
10 Init array in ascending order  
11 -----*/  
12 static void init(int *s, int ssize) {  
13     for (int i = 0 ; i < ssize ; i++) {  
14         s[i] = i ;  
15     }  
16 }  
17  
18 /*-----  
19 log n algorithm  
20 N           1+logn  
21 10          4  
22 100         7  
23 1000        10  
24 1 million   20  
25 1 billion   30  
26 1 billion billion 60  
27  
28  
29 Number to find = 44  
30 l = 0 h = 10000 m = 5000  
31 l = 0 h = 4999 m = 2499  
32 l = 0 h = 2498 m = 1249  
33 l = 0 h = 1248 m = 624  
34 l = 0 h = 623 m = 311  
35 l = 0 h = 310 m = 155  
36 l = 0 h = 154 m = 77  
37 l = 0 h = 76 m = 38  
38 l = 39 h = 76 m = 57  
39 l = 39 h = 56 m = 47  
40 l = 39 h = 46 m = 42  
41 l = 43 h = 46 m = 44  
42 Found 44 after 11 iterations at location 44  
43 -----*/  
44 static void binary_search(int *s, int ssize, int tofind) {  
45     cout << "Number to find = " << tofind << endl ;  
46     //ASSUMPTION: Array in ascending order  
47     {  
48         int p = s[0] ;  
49         for (int i = 1; i < ssize; i++) {  
50             assert(p < s[i]) ;  
51             p = s[i] ;  
52         }  
53     }  
54     int l = 0 ;  
55     int h = ssize ;  
56     int ittr = 0 ;  
57     while (1) {  
58         int m = (l + h) / 2 ;  
59         cout << "l = " << l << " h = " << h << " m = " << m << endl ;  
60         if (s[m] == tofind) {  
61             cout << "Found " << tofind << " after " << ittr << " iterations at location " << m << endl ;  
62             break ;  
63         }  
64         if (l >= h) {  
65             cout << "NOT Found " << tofind << " after " << ittr << " iterations\n" ;  
66     }
```

```
67     break ;
68 }
69 ittr++ ;
70 if (s[m] < tofind) {
71     l = m+1 ;
72 } else {
73     h = m-1 ;
74 }
75 }
76 }
77 */
78 /*-
79 Main program
80 -----
81 int main() {
82     const int N = 10000;
83     int s[N] ;
84     init(s,N) ;
85     binary_search(s,N,44) ;
86     binary_search(s,N,N+78) ;
87     return 0 ;
88 }
89 */
90 /*-
91 Output of this program
92 -----
93 */
94 Number to find = 44
95 l = 0 h = 10000 m = 5000
96 l = 0 h = 4999 m = 2499
97 l = 0 h = 2498 m = 1249
98 l = 0 h = 1248 m = 624
99 l = 0 h = 623 m = 311
100 l = 0 h = 310 m = 155
101 l = 0 h = 154 m = 77
102 l = 0 h = 76 m = 38
103 l = 39 h = 76 m = 57
104 l = 39 h = 56 m = 47
105 l = 39 h = 46 m = 42
106 l = 43 h = 46 m = 44
107 Found 44 after 11 iterations at location 44
108 Number to find = 10078
109 l = 0 h = 10000 m = 5000
110 l = 5001 h = 10000 m = 7500
111 l = 7501 h = 10000 m = 8750
112 l = 8751 h = 10000 m = 9375
113 l = 9376 h = 10000 m = 9688
114 l = 9689 h = 10000 m = 9844
115 l = 9845 h = 10000 m = 9922
116 l = 9923 h = 10000 m = 9961
117 l = 9962 h = 10000 m = 9981
118 l = 9982 h = 10000 m = 9991
119 l = 9992 h = 10000 m = 9996
120 l = 9997 h = 10000 m = 9998
121 l = 9999 h = 10000 m = 9999
122 l = 10000 h = 10000 m = 10000
123 NOT Found 10078 after 13 iterations
124 */
125
```

2.5 $f(n) = n \log n$ algorithms

Figure 2.8: $n \log n$ algorithm

2.6 Quadratic algorithms: $f(n) = n^d$

```

for( int i = 0; i < 10; i++) {
    for (int j = 0 ; j < 10; j++) {
        func(i,j) ;
    }
}

```

outerloop executed 10 times
each inner loop is executed 10 times

$$10 * 10 = 100$$

$$f(n) = n * n = n^2$$

```

for( int i = 0; i < 10; i++) {
    for (int j = 0 ; j < i; j++) {
        func(i,j) ;
    }
}

```

outerloop executed 10 times
each inner loop is executed i times

i	outerloop executed	innerloop executed	total
0	1	0	1
1	1	1	2
2	1	2	3
3	1	3	4
4	1	4	5
5	1	5	6
6	1	6	7
7	1	7	8
8	1	8	9
9	1	9	10

$$k = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10$$

$$f(n) = \frac{n(n-1)}{2} = g(n^2)$$

Figure 2.9: n^2 algorithms

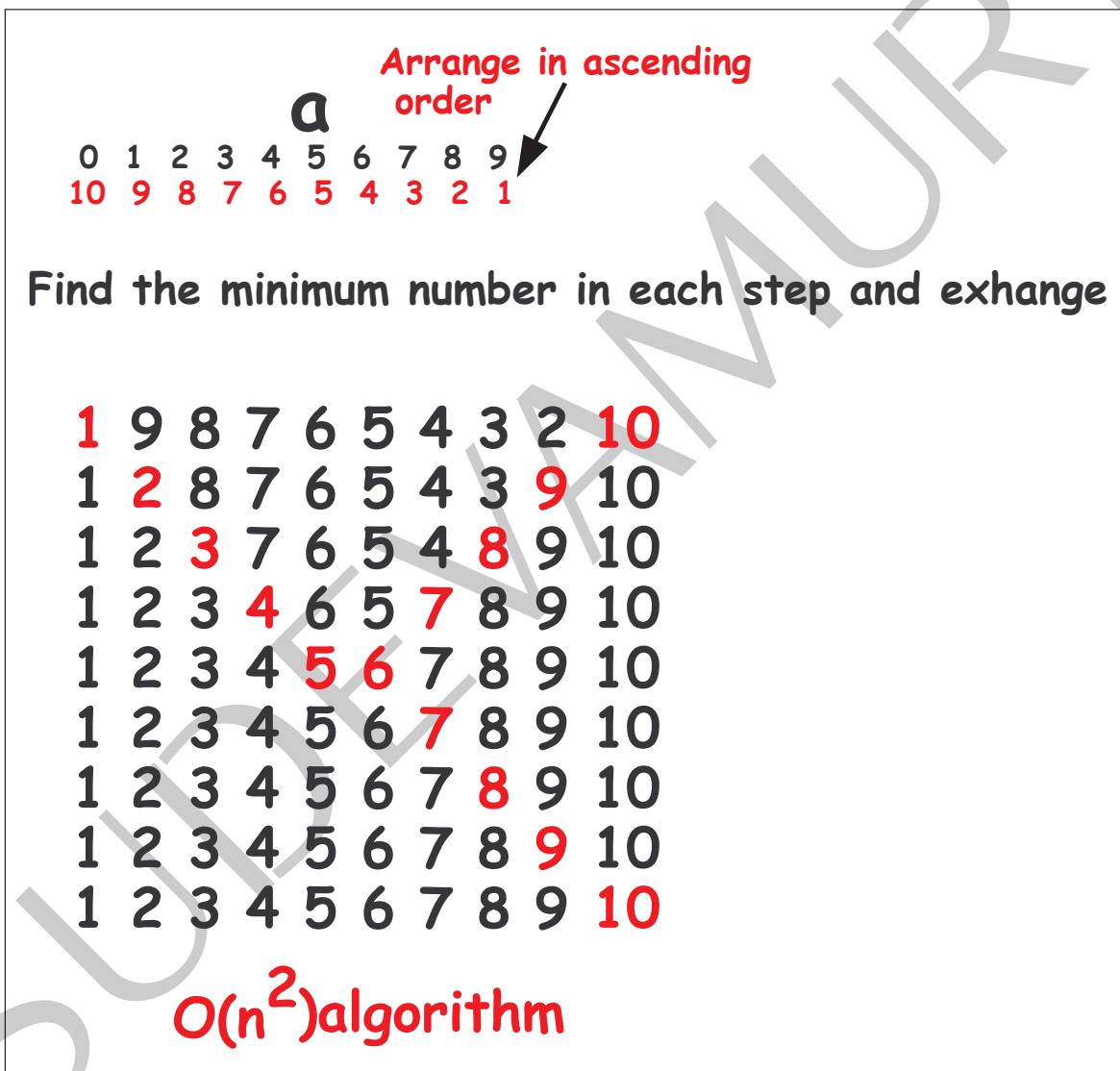


Figure 2.10: A stupid sorting algorithm

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy  
3 Filename: stupid_sort.cpp  
4 -----*/  
5 #include <iostream>  
6 using namespace std;  
7 #include <assert.h>  
8  
9 /*-----  
10 -----*/  
11 static void print(int n, int *a) {  
12     for (int i = 0; i < n; i++) {  
13         cout << a[i] << " ";  
14     }  
15     cout << endl ;  
16 }  
17  
18 /*-----  
19 Find the minimum element in array a and put in 'pos' of array a  
20 -----*/  
21 static void sort_one(int pos,int n, int *a){  
22     int min = a[pos] ;  
23     int minp = -1 ;  
24     for (int i = pos+1; i < n ; i++) {  
25         if (a[i] < min) {  
26             min = a[i] ;  
27             minp = i ;  
28         }  
29     }  
30     if (minp != -1) {  
31         int t = a[pos] ;  
32         a[pos] = min ;  
33         a[minp] = t ;  
34     }  
35     print(n,a) ;  
36 }  
37 }  
38  
39 /*-----  
40 Sort an array in ascending order  
41 -----*/  
42 static void stupid_sort(int n, int *a) {  
43     print(n,a) ;  
44     for (int i = 0; i < n; i++) {  
45         sort_one(i,n,a) ;  
46     }  
47 }  
48  
49 /*-----  
50 10 9 8 7 6 5 4 3 2 1  
51 1 9 8 7 6 5 4 3 2 10  
52 1 2 8 7 6 5 4 3 9 10  
53 1 2 3 7 6 5 4 8 9 10  
54 1 2 3 4 6 5 7 8 9 10  
55 1 2 3 4 5 6 7 8 9 10  
56 1 2 3 4 5 6 7 8 9 10  
57 1 2 3 4 5 6 7 8 9 10  
58 1 2 3 4 5 6 7 8 9 10  
59 1 2 3 4 5 6 7 8 9 10  
60 1 2 3 4 5 6 7 8 9 10  
61 -----*/  
62 int main() {  
63     int a[] = {10,9,8,7,6,5,4,3,2,1} ;  
64     int n = sizeof(a)/sizeof(int) ;  
65     stupid_sort(n,a) ;  
66     return 0 ;
```

2.7 Exponential algorithms: $f(n) = 2^n$

```
static void truth_table(int n) {  
    char t[100] ;  
    initialize_to_zero(n,t) ;  
    print_a_row(n,t) ;  
    do {  
        addone(n,t) ;  
        print_a_row(n,t) ;  
    }while (is_all_one(n,t) == false) ;  
}
```

$n = 3$
000
001
010
011
100
101
110
111

Will this world exist for $n = 64$?

Figure 2.11: Truth table of an n input Boolean function

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevanurthy  
3 Filename: o3.cpp  
4 -----*/  
5 #include <iostream>  
6 using namespace std;  
7 #include <assert.h>  
8  
9 /*-----  
10 Initialize array to zero  
11 -----*/  
12 static void initialize_to_zero(int n,char *t){  
13     for (int i = 0; i < n; i++) {  
14         t[i] = '0' ;  
15     }  
16 }  
17  
18 /*-----  
19 Is array has all ones  
20 -----*/  
21 static bool is_all_one(int n,char *t){  
22     for (int i = 0; i < n; i++) {  
23         if (t[i] != '1') {  
24             return false ;  
25         }  
26     }  
27     return true ;  
28 }  
29  
30 /*-----  
31 t = t + 1 ;  
32  
33 210  
34 ---  
35 011  
36 1  
37 ----  
38 100  
39 -----*/  
40 static void addone(int n,char *t){  
41     int b = 1;  
42     int c = 0 ;  
43     for (int i = 0; i < n ; i++) {  
44         int a = t[i] - '0';  
45         int s = a + b + c ;  
46         assert(s < 3) ;  
47         if (s == 0) {  
48             t[i] = '0' ;  
49             c = 0 ;  
50         }  
51         if (s == 1) {  
52             t[i] = '1' ;  
53             c = 0 ;  
54         }  
55         if (s == 2) {  
56             t[i] = '0' ;  
57             c = 1 ;  
58         }  
59         b = 0 ;  
60     }  
61 }  
62  
63 /*-----  
64 print t  
65  
66 210
```

```
67 ---
68 100
69 -----*/
70 static void print_a_row(int n,char *t){
71     for (int i = n-1; i >=0 ; i--) {
72         cout << t[i] ;
73     }
74     cout << endl ;
75 }
76
77 /*-----
78 Write truth table of n inputs
79 -----*/
80 static void truth_table(int n) {
81     char t[100] ;
82     initialize_to_zero(n,t) ;
83     print_a_row(n,t) ;
84     do {
85         addone(n,t) ;
86         print_a_row(n,t) ;
87     }while (is_all_one(n,t) == false) ;
88 }
89
90
91 /*-----
92
93 -----*/
94 int main() {
95     truth_table(3);
96     return 0 ;
97 }
98
```

Intractability

Intractable problem: Requires so much time and/or space. Although they are solvable in principle, they cannot be solved in reality.

Example:

1. Fold a paper
2. Doubling rice grains on a chess board
3. Writing a truth table of n variables
4. Tower of Hanoi

<http://www.mazeworks.com/hanoi/>

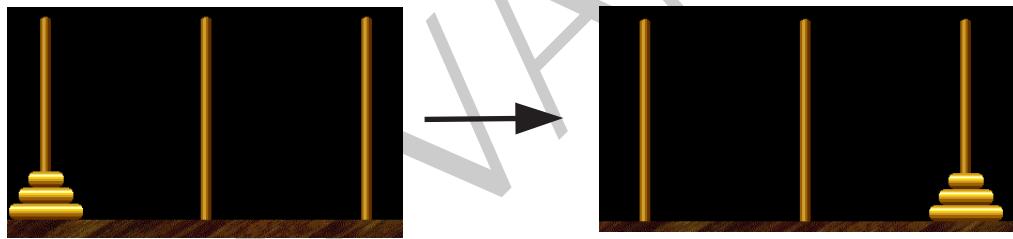


Figure 2.12: Intractability

2.8 Asymptotic Notations

Asymptotic means as n goes to Infinity, what is the running time

O

(Big Oh)

Worst case running time of an algorithm

\leq

O(n)

Worst case of searching
a student in a unorderd list

Ω

(Omega)

Best case running time of an algorithm

\geq

$\Omega(1)$

Best case of searching
a student in a unorderd list

Θ

(Theta)

Worst case = Best case

$=$

$$\text{Sigma}(n) = 1 + 2 + 3 \dots + n = n(n-1)/2$$

Worst case = Best case =

$\Theta(n)$

$$O(1) \ O(\log n) \ O(n) \ O(n \log n) \ O(n^2) \ O(n^c) \ O(2^n) \ O(n!)$$

Figure 2.13: Asymptotic Notations

2.9 Big-O notation

The Big-O Notation

- **Asymptotic analysis** seeks to understand the running time for large inputs.
- Considers only the higher order terms because they dominate on larger inputs.
- Example:
 - Running time is given by $f(n) = 6n^3 + 2n^2 + 20n + 45$, where n is the input size.
 - We say that f is asymptotically at most n^3
 - n^3 is an **asymptotic upper bound**
 - $f(n) = O(n^3)$
- Intuitively: $f(n) = O(g(n))$ means that f is less or equal to g if we ignore differences up to a constant factor.

Figure 2.14: Big-O notation

On the order of n: O(n)

Steps to derive O(n)

Step 1: In each term, set the coefficient to 1

Step 2: Keep the largest term in the function and discard the others.

Example 1: $f(n) = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$

Step1: Set coefficients to 1: $n^2 + n$
 Step2: Keep largest term: n^2
 $O(f(n)) = n^2$

Efficiency for n = 10,000			
Efficiency	Big-O	Iterations	Estimated times
Logarithmic	$O(\log n)$	14	Microseconds
Linear	$O(n)$	10000	Seconds
Linear logarithmic	$O(n\log n)$	140,000	seconds
Quadratic	$O(n^2)$	$10,000^2$	minutes
Polynomial	$O(n^k)$	$10,000^k$	hours
Exponential	$O(2^n)$	$10,000^2$	Intractable
Factorial	$O(n!)$	$10,000!$	Intractable

Figure 2.15: Efficiency of algorithms using Big-O notation

2.10 Polynomial and intractable problems

CHAPTER 2. ANALYSIS OF ALGORITHMS

Time complexity function	Size n					
	10	20	30	40	50	60
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second
n^3	.001 second	.008 second	.027 second	.064 second	.125 second	.216 second
n^4	.1 second	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13.0 minutes
2^n	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	.059 second	58 minutes	6.5 years	3855 centuries	2×10^6 centuries	1.3×10^{13} centuries

Garey, M.R.; Johnson, D.S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. New York: W.H. Freeman. ISBN 0-7167-1045-5.

Time complexity function	Size of Largest Problem Instance Solvable in 1 Hour		
	With present computer	With computer 100 times faster	With computer 1000 times faster
n	N_1	$100 N_1$	$1000 N_1$
n^2	N_2	$10 N_2$	$31.6 N_2$
n^3	N_3	$4.64 N_3$	$10 N_3$
n^4	N_4	$2.5 N_4$	$3.98 N_4$
2^n	N_5	$N_5 + 6.64$	$N_5 + 9.97$
3^n	N_6	$N_6 + 4.19$	$N_6 + 6.29$

Figure 2.16: Time complexity for different functions

2.11. GROWTH OF SEVERAL FUNCTIONS

2.11 Growth of several functions

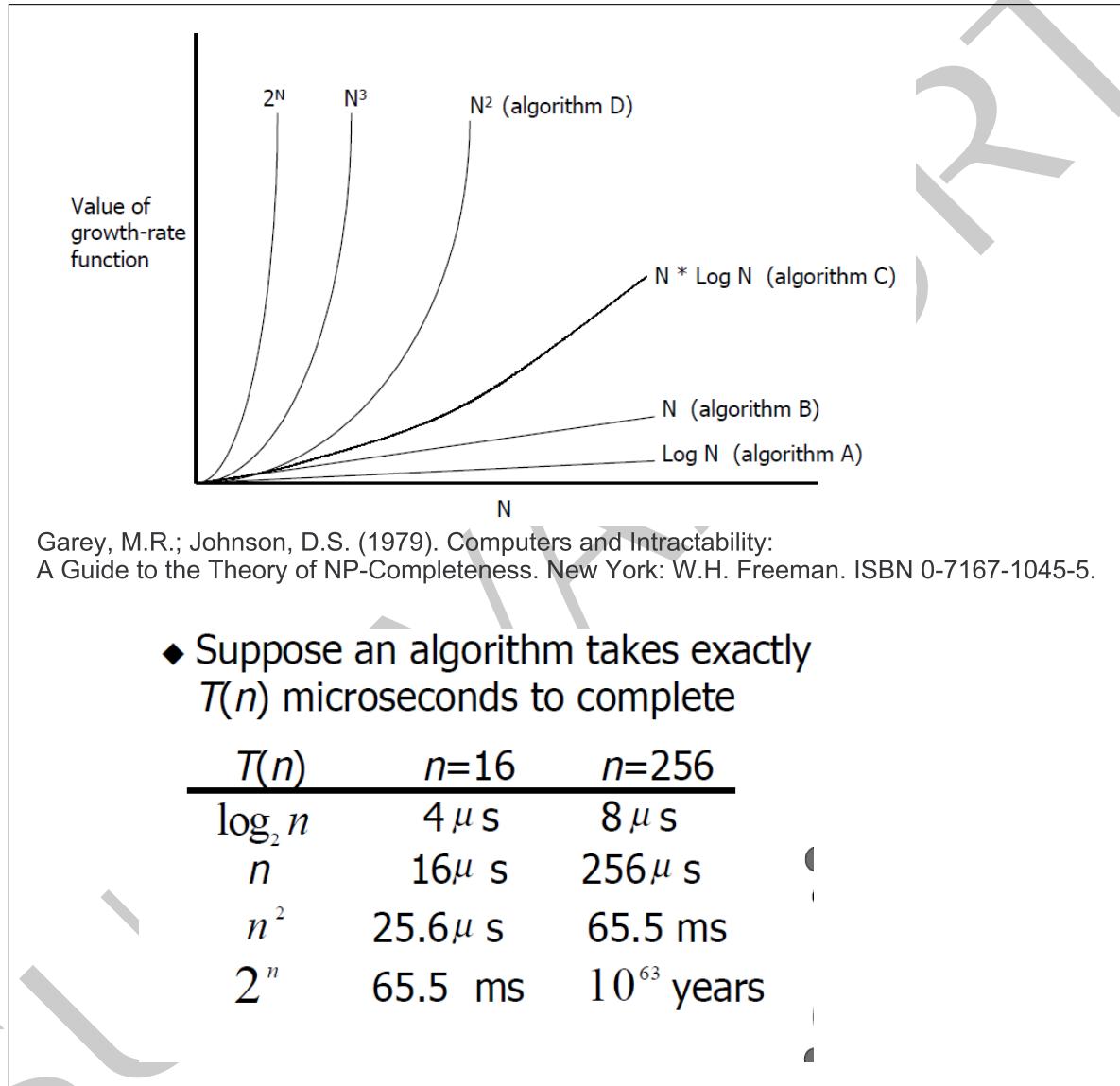


Figure 2.17: Growth of several functions

2.12 Algorithms Techniques

2.12.1 Greedy algorithms

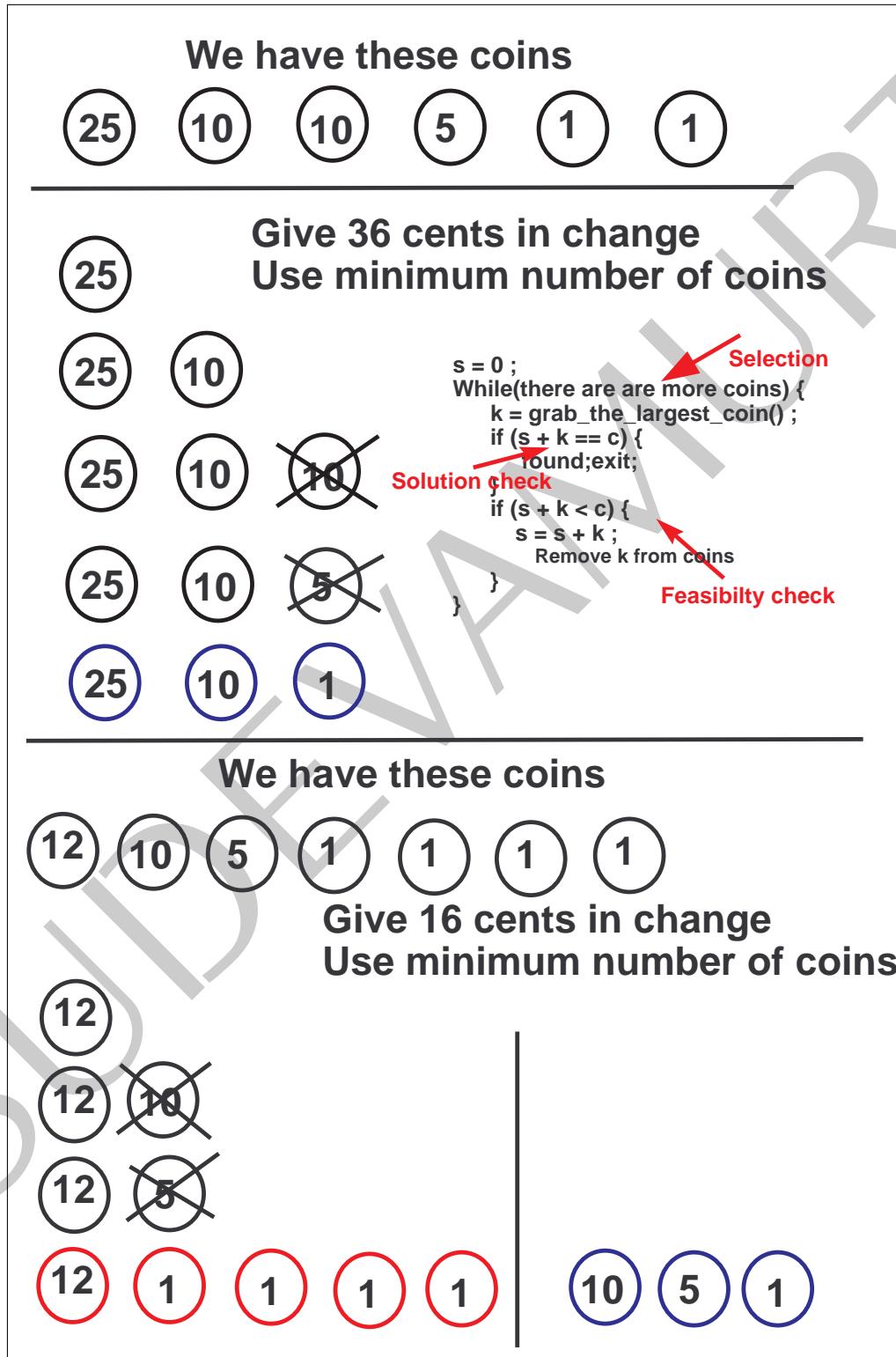


Figure 2.18: Finding minimum number of coins by greedy approach

2.12. ALGORITHMS TECHNIQUES

2.12.2 Divide-and-conquer algorithms

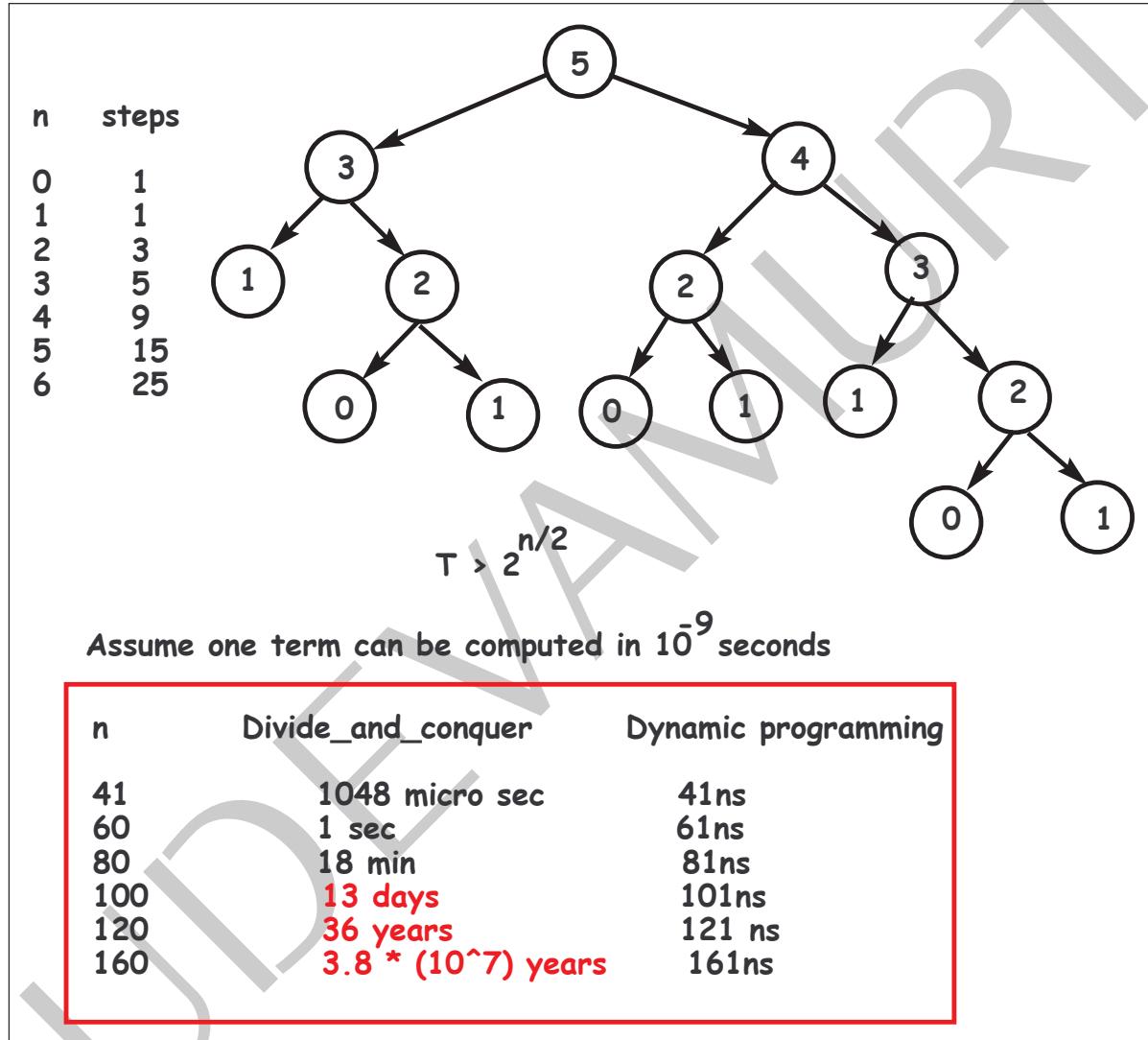


Figure 2.19: Finding Fibonacci number using divide-and-conquer algorithm

2.12.3 Dynamic programming

```
1 /*-----  
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy  
3 Filename:fib.cpp  
4 -----*/  
5 #include <iostream>  
6 #include <assert.h>  
7 #include <time.h>  
8  
9 using namespace std;  
10  
11 /*-----  
12 Get the starting and the ending CPU clock time and  
13 then divide by constant CLOCKS_PER_SEC defined in the time.h header file.  
14 -----*/  
15 static unsigned long measure(const char *s,unsigned long n, unsigned long(*f)(unsigned long n)) {  
16     clock_t start = clock();  
17     unsigned long l = f(n);  
18     clock_t end = clock();  
19     cout << s << ":";  
20     cout << "Run time for fib(" << n << ") : " << double(end - start)/CLOCKS_PER_SEC << " secs" << endl ;  
21     return l;  
22 }  
23  
24 /*-----  
25 Fibonacci recursive algorithm is a divide_and_conquer algorithm.  
26 1. Divides the problem that compute the nth term into two problems  
27 that computes (n-1) term and (n-2) term  
28 2. Conquers this two sub-problem.  
29 3. Combines by adding solutions of two sub-problems  
30  
31 Top down approach  
32  
33 Complexity is =  $2^{(n/2)}$   
34 -----*/  
35 static unsigned long fib_divide_and_conquer(unsigned long n){  
36     if (n <= 1){  
37         return 1;  
38     }  
39     return (fib_divide_and_conquer(n-1) + fib_divide_and_conquer(n-2));  
40 }  
41  
42 /*-----  
43 Fibonacci iterative alorithm is a dynamic programming algorithm.  
44  
45 Like divide_and_conquer algorithm, we divide the problem into sub-problems.  
46 However, we store these answers and later when ever we need the result  
47 we look into the stored answer instead of recomputing.  
48  
49 In the Fibonacci_dynamic_programmin algorithm, we need to store only two previous solutions.  
50 However, in Fibonacci recursive algorithm, it ends up computing the answers to sub-problems  
51 again and again.  
52  
53 Bottom up approach  
54 -----*/  
55 static unsigned long fib_dynamic_programming(unsigned long n){
```

```
56 if (n <= 0) {
57     return 0 ;
58 }
59 if (n == 1) {
60     return 1 ;
61 }
62 unsigned long i0 = 0 ;
63 unsigned long i1 = 1 ;
64 unsigned long answer = 0 ;
65 for (unsigned long i = 2; i <= n; i++) {
66     answer = i0 + i1 ;
67     i0 = i1 ;
68     i1 = answer ;
69 }
70 return answer ;
71 }
72
73 /*-----
74 fib_divide_and_conquer:Run time for fib(38) : 1.344 secs
75 fib_dynamic_programming:Run time for fib(38) : 0 secs
76 Fib(38)= 63245986
77 fib_divide_and_conquer:Run time for fib(39) : 2.156 secs
78 fib_dynamic_programming:Run time for fib(39) : 0 secs
79 Fib(39)= 102334155
80 fib_divide_and_conquer:Run time for fib(40) : 3.484 secs
81 fib_dynamic_programming:Run time for fib(40) : 0 secs
82 Fib(40)= 165580141
83 fib_divide_and_conquer:Run time for fib(41) : 5.656 secs
84 fib_dynamic_programming:Run time for fib(41) : 0 secs
85 Fib(41)= 267914296
86 fib_divide_and_conquer:Run time for fib(42) : 9.156 secs
87 fib_dynamic_programming:Run time for fib(42) : 0 secs
88 Fib(42)= 433494437
89 fib_divide_and_conquer:Run time for fib(43) : 14.811 secs
90 fib_dynamic_programming:Run time for fib(43) : 0 secs
91 Fib(43)= 701408733
92 fib_divide_and_conquer:Run time for fib(44) : 23.953 secs
93 fib_dynamic_programming:Run time for fib(44) : 0 secs
94 Fib(44)= 1134903170
95 fib_divide_and_conquer:Run time for fib(45) : 38.857 secs
96 fib_dynamic_programming:Run time for fib(45) : 0 secs
97 Fib(45)= 1836311903
98 fib_divide_and_conquer:Run time for fib(46) : 63.028 secs
99 fib_dynamic_programming:Run time for fib(46) : 0 secs
100 Fib(46)= 2971215073
101 fib_divide_and_conquer:Run time for fib(47) : 102.636 secs
102 fib_dynamic_programming:Run time for fib(47) : 0 secs
103 Fib(47)= 512559680
104 -----*/
105 int main() {
106     for (unsigned long i = 0; i < 48; i++) {
107         unsigned long a1 = measure("fib_divide_and_conquer", i, fib_divide_and_conquer);
108         unsigned long a2 = measure("fib_dynamic_programming", i, fib_dynamic_programming);
109         assert(a1 == a2);
110         cout << "Fib(" << i << ")= " << a2 << endl;
```

2.12.4 Back tracking

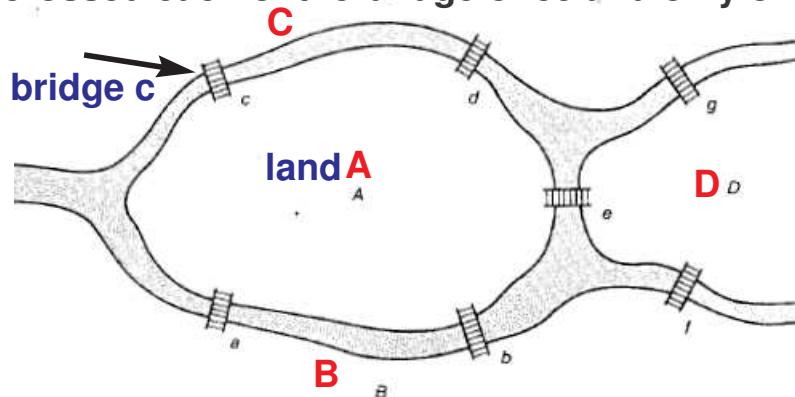
2.12.5 Branch-and-bound

2.13 Efficiency of algorithms

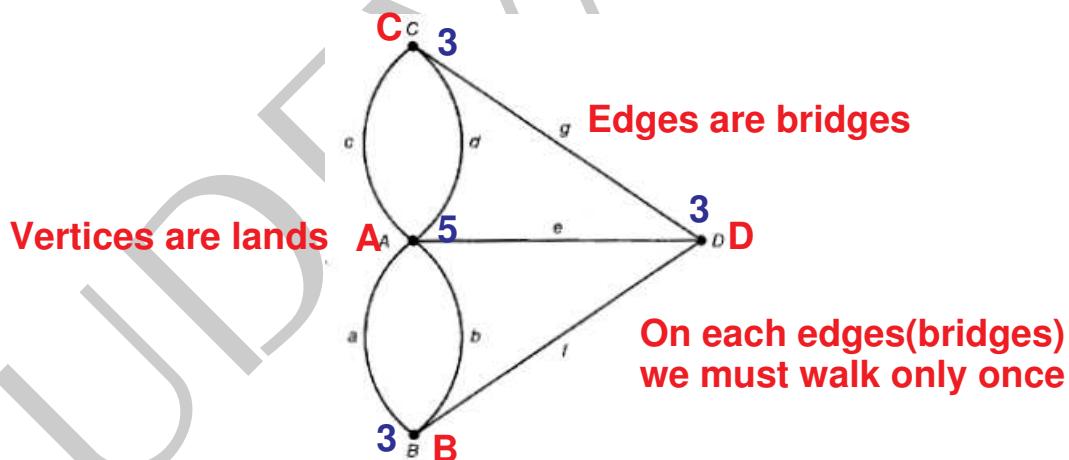
2.13.1 Problem 1

Eulerian path

Is it possible to walk through the park by a route that crossed each of the bridge once and only once?



Given the graph, is it possible to construct a path (or a cycle, i.e. a path starting and ending on the same vertex) which visits each edge exactly once?



Can you draw the above graph without lifting the pen from the paper?

Every vertices should have even degree
The graph is not Eulerian, hence NO solution

Figure 2.20: Eulerian path

2.13.2 Problem 2

Hamilton's Path

Instance: Graph $G = (V, E)$
 Question: Does G contain a Hamiltonian path?

Is there a path through a graph that touch each VERTEX EXACTLY ONCE and return to starting point

yes. See Green

On each vertex we must walk only once

A Hamiltonian path or traceable path is a path that visits each vertex exactly once
 No efficient methods are available, except exhaustive enumeration

But given a solution, we can verify in polynomial time

Figure 2.21: Hamilton path

2.13.3 Problem 3

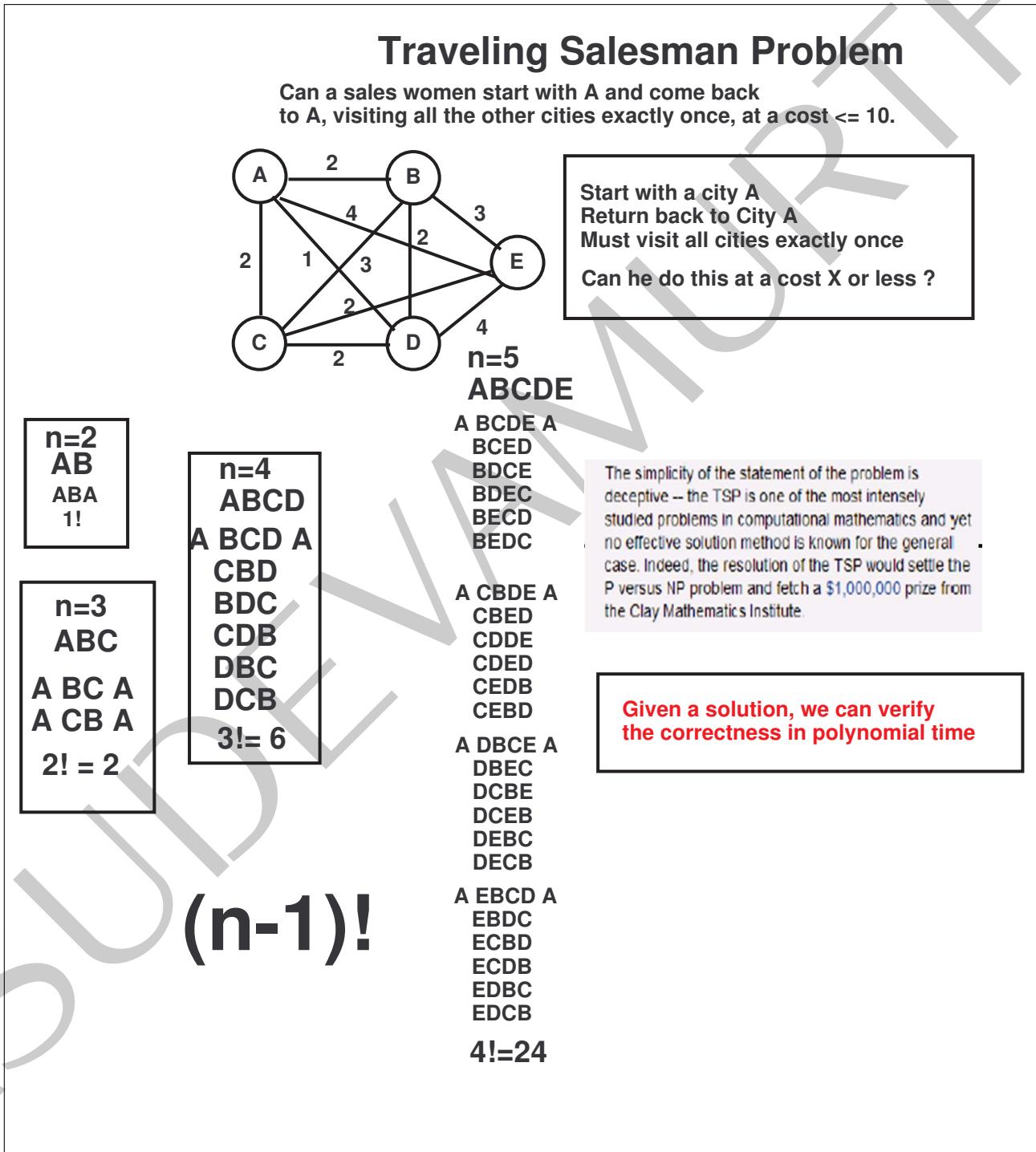


Figure 2.22: Travelling salesman problem

2.13.4 Problem 4

Composite number

**Can a number be written
as product of two numbers?**

The first 105 composite numbers

4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24, 25,
26, 27, 28, 30, 32, 33, 34, 35, 36, 38, 39, 40, 42, 44,
45, 46, 48, 49, 50, 51, 52, 54, 55, 56, 57, 58, 60, 62
, 63, 64, 65, 66, 68, 69, 70, 72, 74, 75, 76, 77, 78
, 80, 81, 82, 84, 85, 86, 87, 88, 90, 91, 92, 93, 94,
95, 96, 98, 99, 100, 102, 104, 105, 106, 108, 110,
111, 112, 114, 115, 116, 117, 118, 119, 120, 121,
122, 123, 124, 125, 126, 128, 129, 130, 132, 133,
134, 135, 136, 138, 140

**No known efficient procedure available
other than enumeration.**

**But given a solution, we can verify
in polynomial time.**

Figure 2.23: Composite number problem

2.13. EFFICIENCY OF ALGORITHMS

RSA Numbers

Ron Rivest, Adi Shamir, and Leonard Adleman
1978



Ronald Rivest (middle), with Alan Sherman (left) and David Chaum (right), 2007

Born	1947
Residence	United States
Fields	Cryptography
Institutions	Massachusetts Institute of Technology

A set of large semiprimes
(Numbers with exactly 2 factors)

p1: 11927 p1 and p2 are prime
p2: 20903

$p_1 p_2 = 11927 * 20903 = 249,310,081$

One way function: Given 249,310,081, can you find p1 and p2

RSA129

GIVEN P1P2=
 $P1P2 = 1143816257578888676692357799761466120$
 $1021829672124236256256184293$
 $5706935245733897830597123563$
 $958705058989075147599290026879543541$

Can you find P1 and P2?

p1 = 349052951084765094914784961990
3898133417764638493387843990820577

p2 = 327691329932667095499619881908
34461413177642967992942539798288533

p1 is a 64 digit long prime number
p2 is a 65 digit long prime number

Figure 2.24: RSA

2.13.5 Problem 5

Satisfiability problem

Given a Boolean equation:

$$f = (a + b + c') \cdot (b' + c) \cdot c'$$

Note that + is logical or
. is logical and
 a' is negation of a

Is it possible to assign 1/0 to each variables such that the overall value of f is 1?

a	b	c	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

No known efficient procedure available other than enumeration.

But given a solution, we can verify in polynomial time.

Figure 2.25: Satisfiability problem

2.13.6 Decision and optimization problems

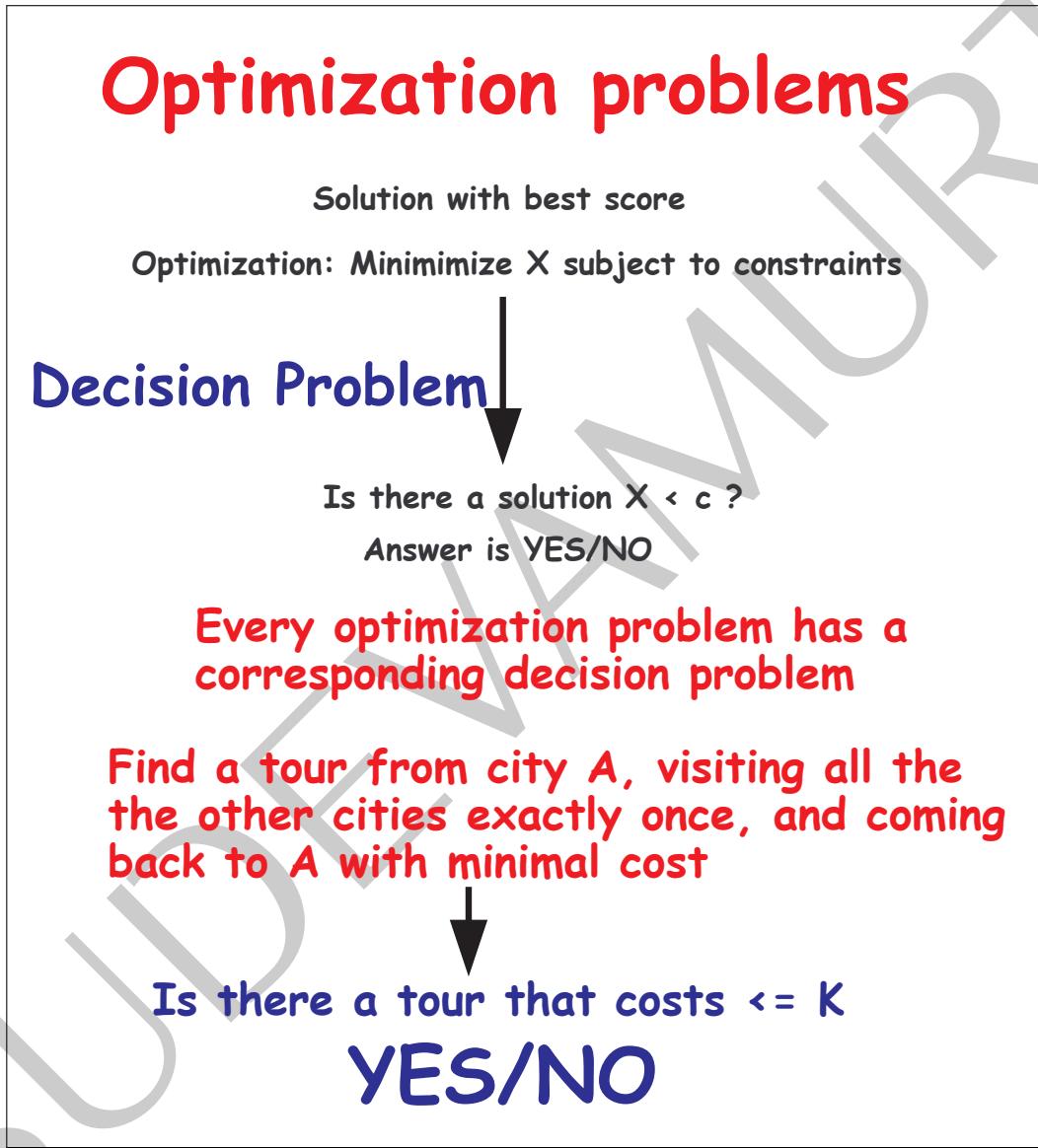


Figure 2.26: Reducing optimization problems to decision problems

2.13.7 Discovery versus verification

Discovery vs Verification

- Two important tasks for a scientist are discovery of solutions, and verification of other people's solutions.
- It is easier to check that a solution, say to a puzzle, is correct, rather than to find the solution.
- That is, *verifying* a solution is easier than *discovering* it.

Ex: Travelling salesman problem

Figure 2.27: Discovery versus verification

2.13.8 Polynomial problems

What is P?

Dealing with decision problems.
Answer is yes/no

1. Solution takes polynomial time
2. Can be verified in polynomial time

Answers can be "discovered" quickly
In time polynomial in the size of input

Ex1: Is n even?

Ex2: Given a graph, does an
Euler Tour exists?

Figure 2.28: P problem

2.13.9 Nondeterministic polynomial problems

What is NP?

Nondeterministic polynomial solution decision problems

Given a graph $G = (V, E)$, does the graph contain a Hamiltonian path ?

Is a given integer x a composite number ?

Given a set of numbers, can be divide them into two groups such that their sum are the same ?

Polynomial Verifiable

Magic coin

Polynomial-Time Verifiable

- Now, imagine that we have a super-smart computer, such that for each decision problem given to it, it has the ability to guess a short **proof** (if there is one)
- With the help of this powerful computer, all polynomial-time verifiable problems can be solved in polynomial time (how ?)
If there are many choices, such a machine possesses magical insight, always indicating
- Because of this, we use **NP** to denote the set of polynomial-time verifiable problems
 - N** stands for non-deterministic guessing power of our computer
 - P** stands for polynomial-time solvable

Figure 2.29: NP problem

2.13.10 Is P = NP?. Million dollar question

P and NP

- We can show that a problem is in P implies that it is in NP (why?)
 - Because if a problem is in P, and if its answer is YES, then there must be an algorithm that runs in polynomial-time to conclude YES ...
 - Then, the execution steps of this algorithm can be used as a "short" proof

P and NP

- On the other hand, after many people's efforts, some problems in NP (e.g., finding a Hamiltonian path) do not have a polynomial-time algorithm yet ...
- Question: Does that mean these problems are not in P ??
- The question whether P = NP is still open

Clay Mathematics Institute (CMI) offers US\$ 1 million for anyone who can answer this ...

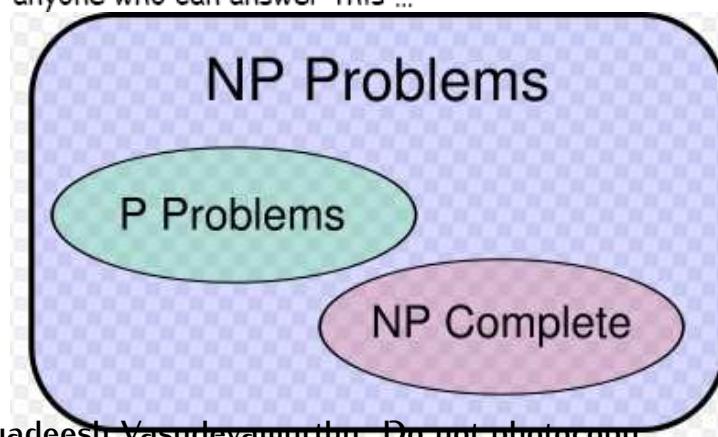
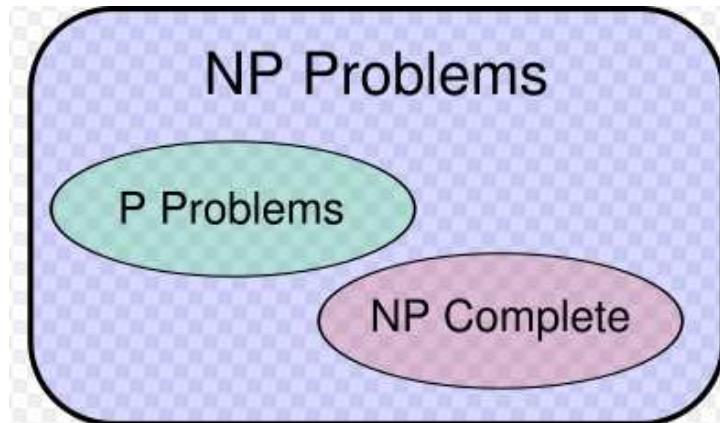


Figure 2.30: Is P = NP?



Why is a proof that $P \neq NP$ important?

- A number of important problems in industry (such as flight scheduling, chip layout, and protein folding) are NP-complete. A proof that $P \neq NP$ would tell us that we cannot expect to get optimal answers in practice.
- Cryptography is based on the assumption that $P \neq NP$. Proving that $P \neq NP$ is a stepping stone towards provably secure cryptography.
- A proof that $P \neq NP$ would give us deep insight into the nature of computation, which would have many ripple effects

Figure 2.31: Is $P \neq NP$?

2.13. EFFICIENCY OF ALGORITHMS

2.13.11 NP complete problems

- NP-complete problems are the “hardest” problems in NP

Ex: SAT problem

If there is a fast (polynomial-time) algorithm for *one* NP-complete problem, then there is a fast algorithm for *every* problem in NP!

The P vs. NP problem has been called
“**one of the deepest questions ever asked by human beings**”.

P vs. NP

- A mathematical issue, not a legal one
- P and NP:
 - Each is a set of computational problems
 - Each is described differently
 - Are they actually the same set?
- A million dollar problem
 - A Clay Millennium Prize
- Most everyone thinks $P \neq NP$
 - the problem is to *prove* it
- On August 6, Vinay Deolalikar proposed a proof

2.13. EFFICIENCY OF ALGORITHMS

2.13.11.1 Situation 1

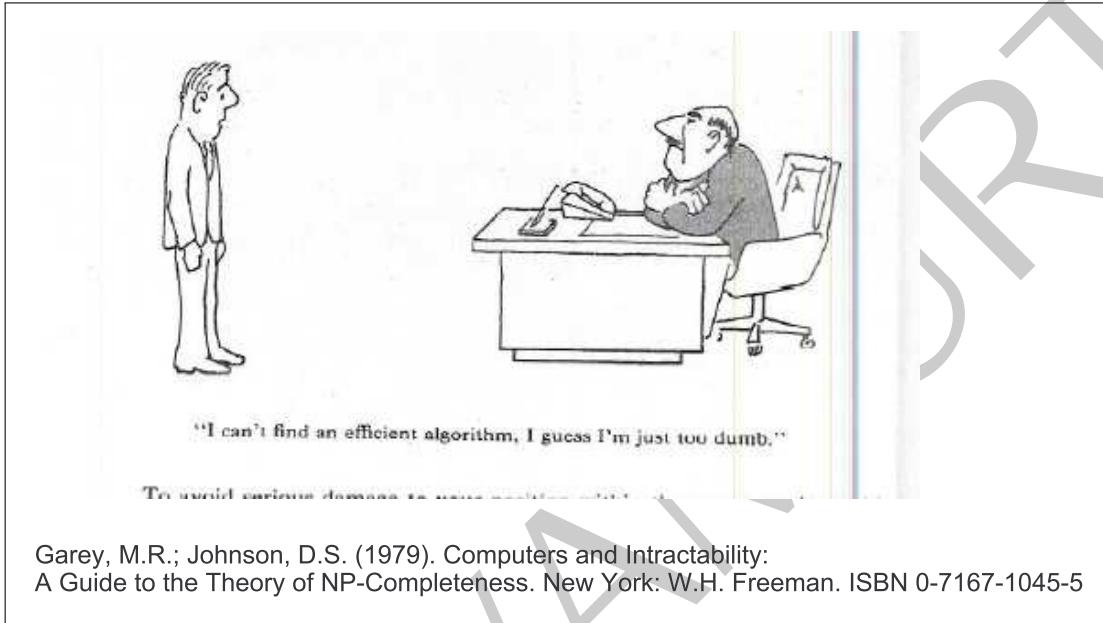


Figure 2.33: Situation 1

2.13.11.2 Situation 2

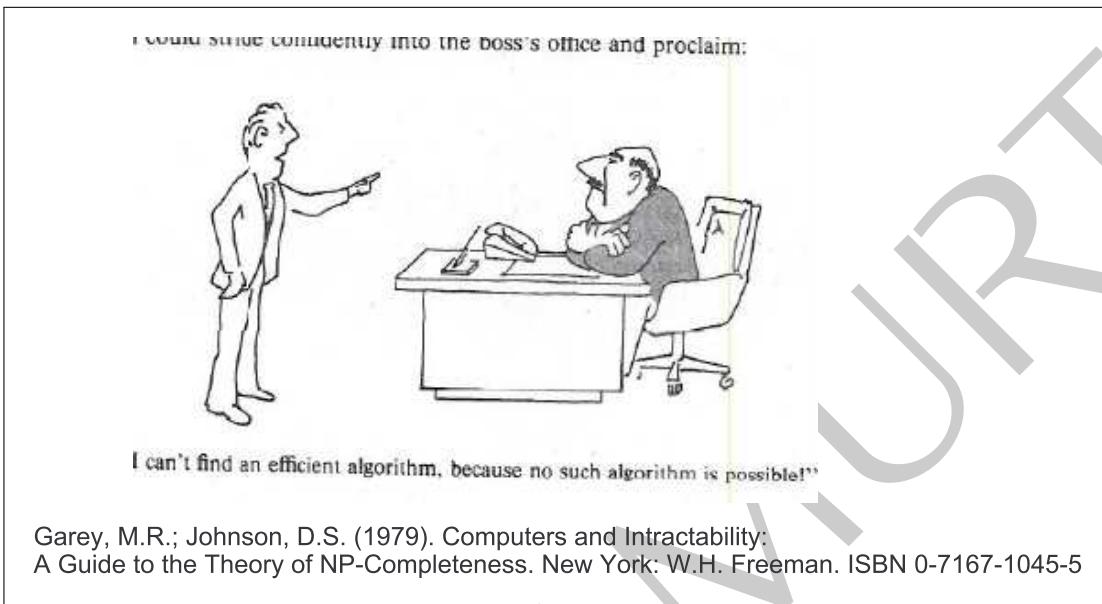


Figure 2.34: Situation 2

2.13.11.3 Situation 3

2.13. EFFICIENCY OF ALGORITHMS

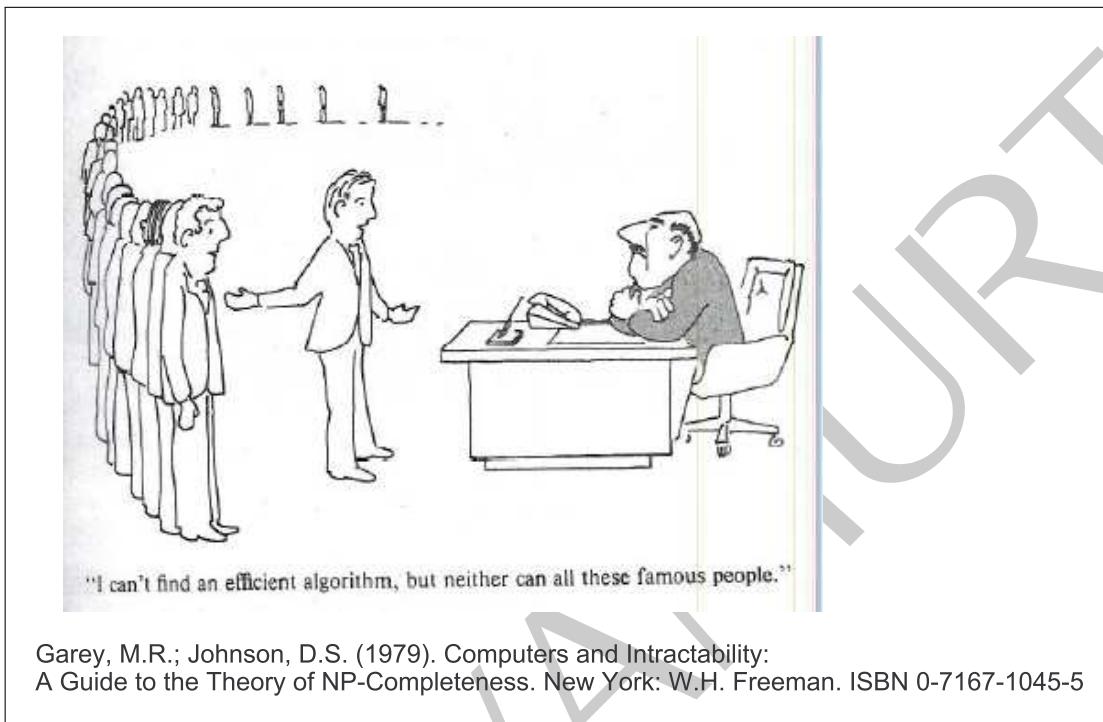


Figure 2.35: Situation 3

2.13.11.4 Magic coin and certificate

Magic coin

NP may be equivalently defined as the set of decision problems that can be solved in polynomial time on a nondeterministic Turing machine. That means you have a magic coin to guess.

Certificate

- Another interesting classification of decision problems is to see if the problem can be verified in time polynomial to the size of the input
- Precisely, for such a decision problem, whenever it has an answer YES, we can :
 1. Ask for a short proof, and
/* short means : polynomial in size of input */
 2. Be able to verify the answer is YES

Figure 2.36: Magic coin and certificate

2.13. EFFICIENCY OF ALGORITHMS

2.13.11.5 Polynomial reduction and proving a problem NP Complete(NPC)

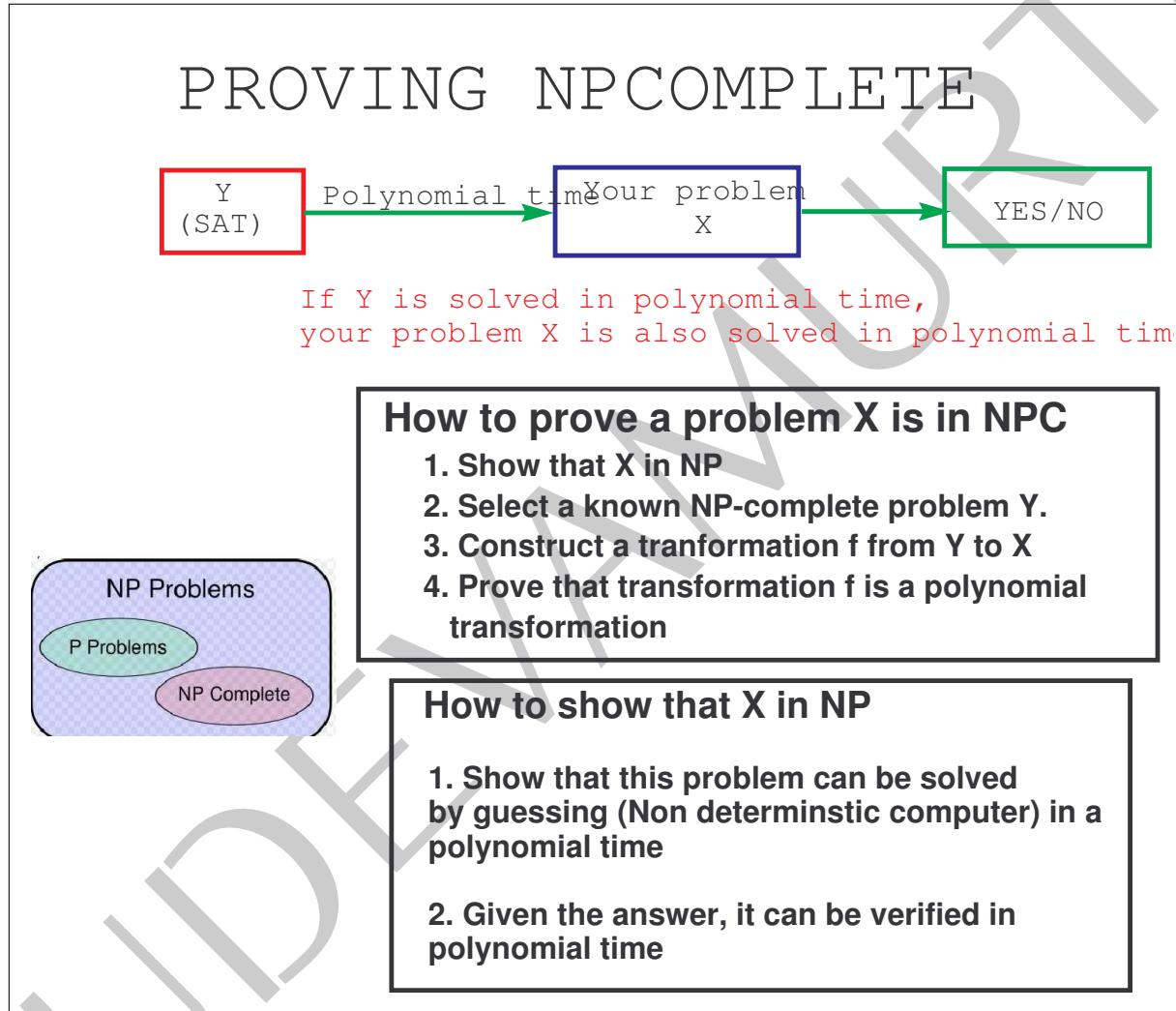


Figure 2.37: Proving NPC

2.13.11.6 NP Complete problems

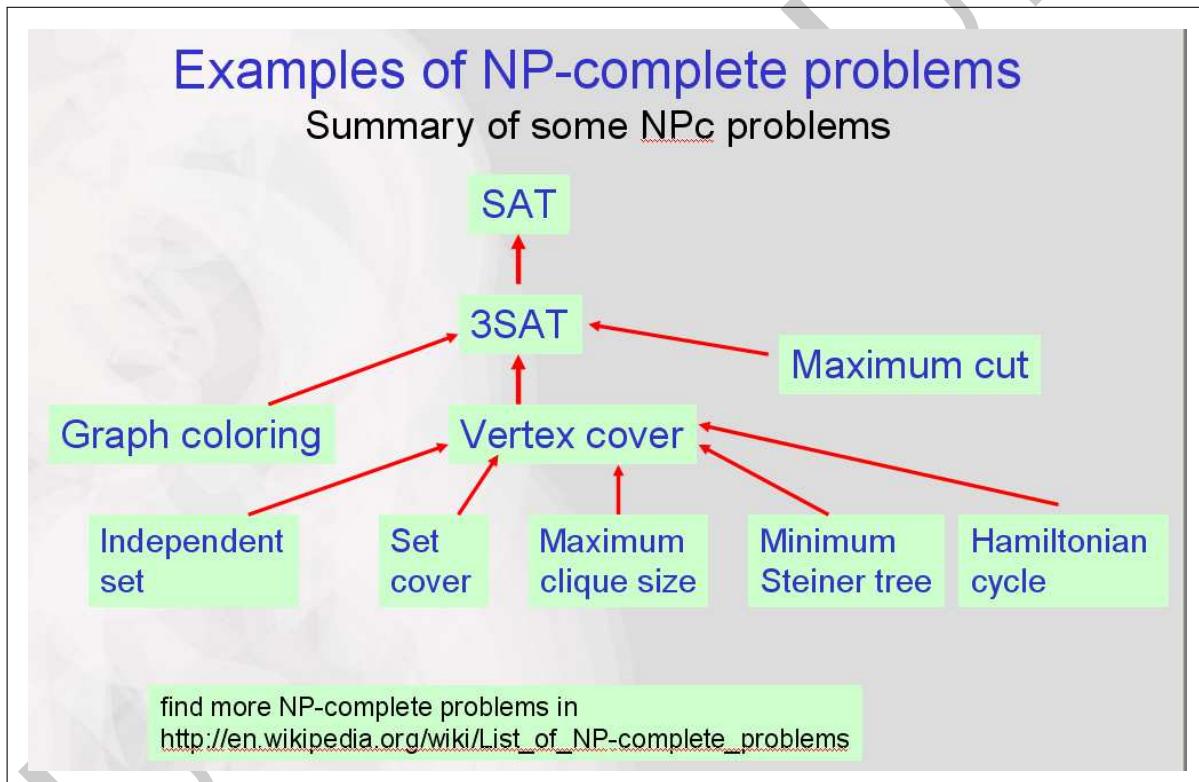


Figure 2.38: NP Complete problems

2.14. PROBLEM SET

2.14 Problem set

Problem 2.14.1. Implement solution to fake coin problem as decribed in fig 2.39



You are given 70 gold coins, one of which is a fake coin (the fake coin has lesser weight). You have a scale, shown in the picture. What is the smallest number of weighing you can do in order to find the fake coin?

1. Implement a class called `fakegold70`
2. Solve for only 70 coins. Not looking solution for arbitrary N.
3. Suppose the fake coin is in position 3, you should print
 1. How many weighing is required to find fake position 3
 2. You should print in what sequence you got that position.
4. Run your program by placing fake coin from position 0 to 69.
5. Record how many minimum and maximum weighing is required to find the fake coins?
6. email `fakegold70.h` `fakegold70.cpp` `fakegold70test.cpp`
7. Attach the output as a comment to `fakegold70test.cpp`

Figure 2.39: Finding a fake coin

CHAPTER 2. ANALYSIS OF ALGORITHMS

Problem 2.14.2. Implement solution to prime number generation problem as described in fig 2.40

Find all the prime numbers up to n

1. School/Brute force method

```
bool isPrime(int i) {
    count the number of basic step here
}
int p[n+1];
int k = 0;
for (int i = 0; i <=n ; ++i) {
    if(isPrime(i)) {
        p[k++] = i;
    }
}
```

2. Up to primenumbers

1. Keep an array of computed prime array
2. example: $p = \{2, 3, 5, 7\}$

Now to see if $n = 8$ is a prime
 n is divisible by $p[i]$. So 8 is not a prime

Now to see if $n = 11$ is a prime
 n cannot be divisible by $p[i]$. So 11 is a prime
add to p . $p = \{2, 3, 5, 7, 11\}$

This array must be equivalent

3. Sieve of Eratosthenes

As an example, consider the application of the algorithm to finding the list of primes not exceeding $n = 25$:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	
2	3		5	7		9		11		13		15		17		19		21		23		25		
2	3		5	7				11		13				17		19				23		25		
2	3		5	7					11		13				17		19				23		25	

For this example, no more passes are needed because they would eliminate numbers already eliminated on previous iterations of the algorithm. The remaining numbers on the list are the consecutive primes less than or equal to 25.
What is the largest number n whose multilas ...

1. Create an object prime
2. Implement all the 3 algorithms
3. assert all the three prime arrays are equivalent
4. Count the basic operation for each method
5. Record the CPU time for each method
6. Make a table as follows

#Steps and #Cpu time			
n	school	upto	sieve
10000			
20000			
150000			

Figure 2.40: Finding all the prime numbers up to n

2.14. PROBLEM SET

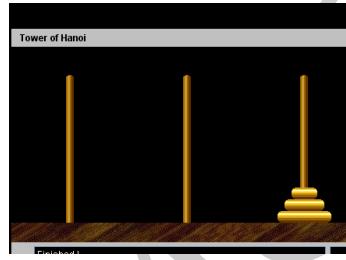
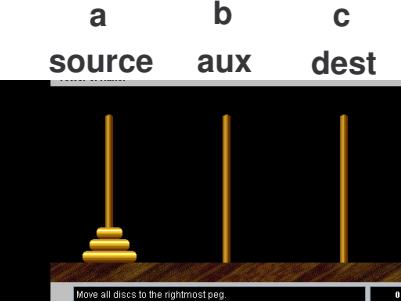
Problem 2.14.3. Write an object called **median** that finds the k 'th largest element in an unsorted random array of n integers. Run your program for large values of n up to 150,000 and let $k = n/2$. Record both the basic steps and the CPU time.

CHAPTER 2. ANALYSIS OF ALGORITHMS

Problem 2.14.4. Implement solution to tower of hanoi problem, fig 2.41, using both the recursive and non recursive methods. Implement the code in the file **tower.cpp**, given below.

2.14. PROBLEM SET

PLAY THIS GAME BEFORE WRITING CODE
<http://www.mazeworks.com/hanoi/index.html>



move (n-1) disk from a(source) to b(dest) using c(aux)
 move remaining disk from a(source) to c(dest) (TRIVAL)
 move (n-1) disk from b(source) to c(dest) using a(aux)

$th_r(n, \text{source}, \text{dest}, \text{aux})$

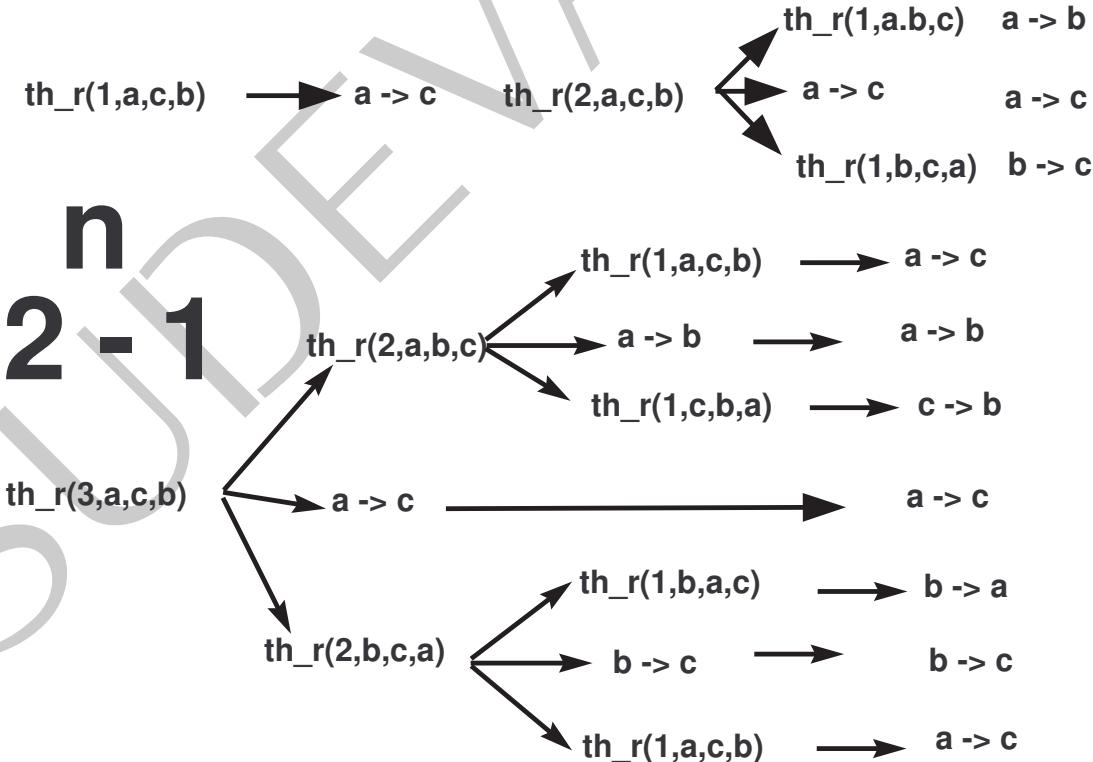


Figure 2.41: Tower of Hanoi

```
1 /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevanurthy  
3 Filename: tower.cpp  
4 -----*/  
5 #include <iostream>  
6 using namespace std;  
7  
8 /*-----  
9 WRITE YOUR RECURSIVE ROUTINE BELOW.  
10 YOU CANNOT CHANGE INTERFACE OF THIS PROCEDURE  
11 -----*/  
12 static void th_r(int n, char source, char dest, char aux) {  
13     /* Problem 1 */  
14     /* WRITE CODE HERE */  
15  
16 }  
17  
18 /*-----  
19 WRITE YOUR NON-RECURSIVE ROUTINE BELOW.  
20 YOU CANNOT CHANGE INTERFACE OF THIS PROCEDURE  
21 -----*/  
22 static void th(int n, char source, char dest, char aux) {  
23     /* Problem 2 */  
24     /* WRITE CODE HERE */  
25     /* th should not call th_r */  
26  
27 }  
28  
29 /*-----  
30 solve using recursion  
31 DO NOT CHANGE ANYTHING IN THIS PROCEDURE  
32 -----*/  
33 static void recursion() {  
34     int n = 1;  
35     cout << "----- " << n << " -----" << endl;  
36     th_r(n,'a','c','b');  
37     n = 2;  
38     cout << "----- " << n << " -----" << endl;  
39     th_r(n,'a','c','b');  
40     n = 3;  
41     cout << "----- " << n << " -----" << endl;  
42     th_r(n,'a','c','b');  
43     n = 4;  
44     cout << "----- " << n << " -----" << endl;  
45     th_r(n,'a','c','b');  
46 }  
47  
48 /*-----  
49 solve without recursion  
50 DO NOT CHANGE ANYTHING IN THIS PROCEDURE  
51 -----*/  
52 static void non_recursion() {  
53     int n = 1;  
54     cout << "----- " << n << " -----" << endl;  
55     th(n,'a','c','b');
```

```
56 n = 2 ;
57 cout << "----- " << n << " ----- " << endl ;
58 th(n,'a','c','b') ;
59 n = 3 ;
60 cout << "----- " << n << " ----- " << endl ;
61 th(n,'a','c','b') ;
62 n = 4 ;
63 cout << "----- " << n << " ----- " << endl ;
64 th(n,'a','c','b') ;
65 }
66
67 /*
68 main
69 -----
70 int main() {
71 recursion();
72 non_recursion();
73 return 0;
74 }
75
76 /*
77 Output of the program
78 Must be identical for both 1) recursion() and 2) non-recursion()
79 -----
80 */
81 ----- 1 -----
82 a --> c
83 ----- 2 -----
84 a --> b
85 a --> c
86 b --> c
87 ----- 3 -----
88 a --> c
89 a --> b
90 c --> b
91 a --> c
92 b --> a
93 b --> c
94 a --> c
95 ----- 4 -----
96 a --> b
97 a --> c
98 b --> c
99 a --> b
100 c --> a
101 c --> b
102 a --> b
103 a --> c
104 b --> c
105 b --> a
106 c --> a
107 b --> c
108 a --> b
109 a --> c
110 b --> c
```

Problem 2.14.5. Wine Puzzle

2.14. PROBLEM SET

Splitting 8 gallons into halves

We have 3 containers, of volumes 8, 5, and 3 gallons.

The 8-gallon container is full.

Using only the 3- and 5-gallon containers,
split the contents evenly: 4 gallons in the 8-gallon container,
and 4 gallons in the 5-gallon container.

There is NO measuring mark on the containers

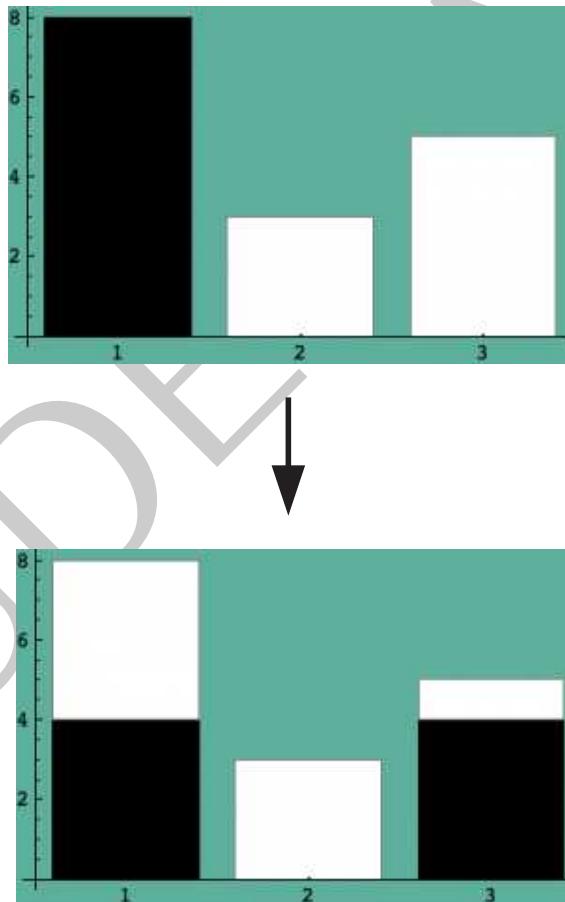


Figure 2.42: Wine puzzle

Problem 2.14.6. Coconut and Monkey Puzzle

Five sailors survive a shipwreck and swim to a tiny island where there is nothing but a coconut tree and a monkey. The sailors gather all the coconuts and put them in a big pile under the tree. Exhausted, they agree to go to wait until the next morning to divide up the coconuts.

At one o'clock in the morning, the first sailor wakes. He realizes that he can't trust the others, and decides to take his share now. He divides the coconuts into five equal piles, but there is one left over. He gives that coconut to the monkey, buries his coconuts, and puts the rest of the coconuts back under the tree.

At two o'clock, the second sailor wakes up. Not realizing that the first sailor has already taken his share, he too divides the coconuts up into five piles, leaving one over which he gives to the monkey. He then hides his share, and piles the remainder back under the tree.

At three, four and five o'clock in the morning, the third, fourth and fifth sailors each wake up and carry out the same actions.

In the morning, all the sailors wake up, and try to look innocent. No one makes a remark about the diminished pile of coconuts, and no one decides to be honest and admit that they've already taken their share. Instead, they divide the pile up into five piles, for the sixth time, and find that there is yet again one coconut left over, which they give to the monkey.

How many coconuts were there originally?

Problem 2.14.7. Microsoft Puzzle

Four students are trying for a job at Microsoft. The interview is at exactly 12'clock night. They must report on or before 12'clock night. It is pitch dark outside and they are near Microsoft. The time at that point is 11:43PM. That means they have to be at Microsoft in 17 minutes or less. They see a bridge connecting Microsoft and the place where they are standing. The bridge is clearly marked as: ONLY TWO PEOPLE CAN CROSS THE BRIDGE AT A TIME. As it is dark, they need to carry a flashlight while crossing the bridge.

We call these four students as 1, 2, 5 and 10, because student 1 can cross the bridge in one minute, student 2 can cross the bridge in two minutes, student 5 can cross the bridge in five minutes and student 10 can cross the bridge in ten minutes. That means, if student 5 and 10 crosses the bridge along with the flashlight, it takes 10 minutes to cross the bridge and assuming 5 come back, he require another 5 minutes to cross and get his other friend 1 or 2.

Give an arrangement of students so that they can ALL reach Microsoft in less than or equal to 17 minutes.

2.14. PROBLEM SET

Problem 2.14.8. Cannibals and Missionaries Puzzle Help the 3 cannibals and 3 missionaries to move to the other side of the lake. Note that when there are more cannibals on one side than missionaries, the cannibals eat them.



Figure 2.43: Cannibals and Missionaries Puzzle

Problem 2.14.9. GOOGLE interview question: Implement division (without using the divide operator, obviously)

Problem 2.14.10. GOOGLE interview question: You have eight balls all of the same size. 7 of them weigh the same, and one of them weighs slightly more. How can you find the ball that is heavier by using a balance and only two weighing?

Problem 2.14.11. GOOGLE interview question: You have 5 jars of pills. Each pill weighs 10 gram, except for contaminated pills contained in one jar, where each pill weighs 9 gm. Given a scale, how could you tell which jar had the contaminated pills in just one measurement?

Problem 2.14.12. GOOGLE interview question:

Give a fast way to multiply a number by 7.

Problem 2.14.13. MICROSOFT interview question:

Give me an algorithm and C code to shuffle a deck of cards, given that the cards are stored in an array of ints. Try to come up with a solution that does not require any extra space.

2.14. PROBLEM SET

VASUDEVAMURTHY

Chapter 3

Dynamic array

3.1 Introduction

3.2 Util file

```
1 /*-----  
2 Copyright (c) 2014 Author: Jagadeesh Vasudevamurthy  
3 file: util.h  
4 -----*/  
5  
6 /*-----  
7 include this file for all the programs  
8 -----*/  
9 #ifndef UTIL_H  
10 #define UTIL_H  
11  
12 /*-----  
13 Basic include files  
14 -----*/  
15 #include <iostream>  
16 #include <fstream>  
17  
18 #include <iomanip> // std::setprecision  
19 using namespace std;  
20  
21 #ifdef _WIN32  
22 #include <cassert>  
23 #include <ctime>  
24 #include "vld.h" //Comment this line, if you have NOT installed Visual leak detector  
25 #else  
26 #include <assert.h>  
27 #include <time.h>  
28 #include <string.h> //for strlen,strcat and strcpy on linux  
29 #endif  
30  
31 // 'sprintf': This function or variable may be unsafe. Consider using sprintf_s instead. To disable  
// deprecation,  
32 //use _CRT_SECURE_NO_WARNINGS  
33 //To overcome above warning  
34 #ifdef _MSC_VER  
35 #pragma warning(disable: 4996) /* Disable deprecation */  
36 #endif  
37  
38  
39 /*-----  
40 STL  
41 -----*/  
42 #include <stdexcept> //Without this catch will NOT work on Linux  
43 #include <vector>  
44 #include <string>  
45  
46 /*-----  
47 All external here  
48 -----*/  
49 extern int Strcmp(const char* s1, const char* s2);  
50 extern void Strcpy(char* s1, const char* s2);  
51 extern void print_integer(const int& x);  
52 extern void print_integer(const int*& x);  
53 extern void print_integer(int& x);  
54 extern void print_integer(int*& x);  
55 extern int intAscendingOrder(const int& c1, const int& c2);  
56 extern int intAscendingOrder(int* const& c1, int* const& c2);  
57 extern int intDescendingOrder(const int& c1, const int& c2);  
58 extern int intDescendingOrder(int* const& c1, int* const& c2);  
59 extern void delete_int(int*& c);  
60 extern void delete_charstar(char*& c);  
61 extern int charcompare(const char& c1, const char& c2);  
62 extern void print_char(char& c);  
63 extern void print_string(char*& c);  
64 extern void free_string(char*& c);  
65 extern int stringDescendingOrder(char* const& c1, char* const& c2);
```

```
66 extern int stringAscendingOrder(char* const& c1, char* const& c2);  
67  
68  
69 #endif  
70 //EOF  
71  
72  
73
```

```
1 /*-
2 Copyright (c) 2014 Author: Jagadeesh Vasudevamurthy
3 Filename: util.cpp
4 -----*/
5 #include "util.h"
6
7 /*
8 strcmp
9 -----*/
10 int Strcmp(const char* s1, const char* s2){
11     for (; *s1 == *s2; ++s1, ++s2)
12     if (*s1 == 0) {
13         return 0;
14     }
15     return *(unsigned char *)s1 < *(unsigned char *)s2 ? -1 : 1;
16 }
17
18 /*
19 strcpy
20 -----*/
21 void Strcpy(char* s1, const char* s2){
22     int i = 0;
23     while (1) {
24         s1[i] = s2[i];
25         if (s2[i] == '\0') {
26             return;
27         }
28         i++;
29     }
30 }
31
32 /*
33 Print an integer
34 -----*/
35 void print_integer(const int& x){
36     cout << x << " ";
37 }
38
39 /*
40 Print an integer
41 -----*/
42 void print_integer(const int*& x){
43     cout << *x << " ";
44 }
45
46 /*
47 Print an integer
48 -----*/
49 void print_integer(int& x){
50     cout << x << " ";
51 }
52
53 /*
54 Print an integer
55 -----*/
56 void print_integer(int*& x){
57     cout << *x << " ";
58 }
59
60 /*
61 7 9 1
62 9 7 -1
63 -----*/
64 int intAscendingOrder(const int& c1, const int& c2){
65     if (c1 == c2) {
66         return 0;
```

```
67     }
68     if (c1 < c2) {
69         return 1;
70     }
71     return -1;
72 }
73
74 /*-
75 pointer
76 -----
77 int intAscendingOrder(int* const& c1, int* const& c2){
78     return intAscendingOrder(*c1, *c2);
79 }
80
81 /*-
82 7 9 -1
83 9 7 1
84 -----
85 int intDescendingOrder(const int& c1, const int& c2){
86     int x = intAscendingOrder(c1, c2);
87     return -x;
88 }
89
90 /*-
91 pointer
92 -----
93 int intDescendingOrder(int* const& c1, int* const& c2){
94     return intDescendingOrder(*c1, *c2);
95 }
96
97 /*-
98 Delete a int object
99 -----
100 void deleteInt(int*& c){
101     delete(c);
102 }
103
104 /*-
105 Delete a char * object
106 -----
107 void deleteCharStar(char*& c){
108     delete[] c;
109 }
110
111 /*-
112 char compare
113 -----
114 int charCompare(const char& c1, const char& c2){
115     if (c1 == c2) {
116         return 0;
117     }
118     if (c1 > c2) {
119         return 1;
120     }
121     return -1;
122 }
123
124 /*-
125 char print
126 -----
127 void printChar(char& c){
128     cout << c << " ";
129 }
130
131 /*-
132 string print
```

```
133 -----*/
134 void print_string(char*& c){
135     cout << c << " ";
136 }
137
138 /*-----
139 Delete c which is allocated by new []
140 -----*/
141 void free_string(char*& c){
142     delete[] c;
143 }
144
145 /*-
146 henry  zoo  is in descending order: -1
147 tom    idiot is in decending order:   1
148 -----*/
149 int string_descending_order(char* const& c1, char* const& c2){
150     int x = strcmp(c1, c2);
151     return x;
152 }
153
154 /*-
155 henry  zoo  is in ascending order: -1  1
156 tom    idiot is in ascending order:   1  -1
157 -----*/
158 int stringAscending_order(char* const& c1, char* const& c2){
159     int x = string_descending_order(c1, c2);
160     return -x;
161 }
162
163
```

3.3 Writing a class

3.4 Static array of integer

```
1 /*-
2 Copyright (c) 2014 Author: Jagadeesh Vasudevamurthy
3 file: ../complex/complex.cpp sarraytest.cpp
4
5 On linux:
6 g++ ../complex/complex.cpp sarraytest.cpp
7 valgrind a.out
8 -- All heap blocks were freed -- no leaks are possible
9
10 -----*/
11
12 /*-
13 static array test
14 -----*/
15
16 /*-
17 All includes here
18 -----*/
19 #include "../complex/complex.h"
20
21 /*-
22 array of integers
23 -----*/
24 void test_array_of_integers(){
25     const int N = 5;
26     int a[N];
27     for (int i = 0; i < N; i++) {
28         a[i] = i * 10;
29     }
30     int b[N];
31     //b = a; CANNOT DO: error C2106: '=' : left operand must be l-value
32     for (int i = 0; i < N; i++) {
33         b[i] = a[i];
34     }
35     for (int i = 0; i < N; i++) {
36         cout << " b[" << i << "]=" << b[i];
37     }
38     cout << endl;
39     /* this won't work */
40     if (a == b) {
41         cout << " a eq b " << endl;
42     }
43     else {
44         cout << " a !eq b " << endl;
45     }
46     bool equal = true;
47     for (int i = 0; i < N; i++) {
48         if (a[i] != b[i]) {
49             equal = false;
50             break;
51         }
52     }
53     if (equal) {
54         cout << "array a and b is equal\n";
55     }
56     else {
57         cout << "array a and b is NOT equal\n";
58     }
59 }
60
61 /*-
62 array of integer pointers
63 -----*/
64 void test_array_of_ptr_integers(int n){
65     int** a = new int*[n];
66     for (int i = 0; i < n; i++) {
```

```
67     int *p = new int(i * 10);
68     a[i] = p;
69 }
70 int** b = new int*[n];
71 for (int i = 0; i < n; i++) {
72     b[i] = a[i];
73 }
74 bool equal = true;
75 for (int i = 0; i < n; i++) {
76     if (*(a[i]) != *(b[i])) {
77         equal = false;
78         break;
79     }
80 }
81 if (equal) {
82     cout << "array a and b is equal\n";
83 }
84 else {
85     cout << "array a and b is NOT equal\n";
86 }
87 for (int i = 0; i < n; i++) {
88     /* we allocated contents in a. So we delete it */
89     delete(a[i]);
90 }
91 delete [] a;
92 /* Why you should NOT do this */
93 /*
94 for (int i = 0; i < n; i++) {
95     delete(b[i]);
96 }
97 */
98 delete [] b;
99 }
100 /*
101 array of user defined type
102 -----
103 */
104 void test_array_of_udt(){
105     const int N = 5;
106     complex a[N];
107     for (int i = 0; i < N; i++) {
108         a[i].setxy(i, -i);
109     }
110     complex b[N];
111     for (int i = 0; i < N; i++) {
112         b[i] = a[i];
113     }
114     bool equal = true;
115     for (int i = 0; i < N; i++) {
116         if (a[i] != b[i]) {
117             equal = false;
118             break;
119         }
120     }
121     if (equal) {
122         cout << "array a and b is equal\n";
123     }
124     else {
125         cout << "array a and b is NOT equal\n";
126     }
127 }
128 /*
129 array of user defined pointer type
130 -----
131 */
132 void test_array_of_ptr_udt(int n){
```

```
133 complex** a = new complex*[n];
134 for (int i = 0; i < n; i++) {
135     complex *c = new complex (i, -i);
136     a[i] = c;
137 }
138 complex** b = new complex*[n];
139 for (int i = 0; i < n; i++) {
140     b[i] = a[i];
141 }
142 bool equal = true;
143 for (int i = 0; i < n; i++) {
144     if (*(a[i]) != *(b[i])) {
145         equal = false;
146         break;
147     }
148 }
149 if (equal) {
150     cout << "array a and b is equal\n";
151 }
152 else {
153     cout << "array a and b is NOT equal\n";
154 }
155 for (int i = 0; i < n; i++) {
156     /* we allocated contents in a. So we delete it */
157     delete(a[i]);
158 }
159 delete[] a;
160 /* Why you should NOT do this */
161 /*
162 for (int i = 0; i < n; i++) {
163     delete(b[i]);
164 }
165 */
166 delete[] b;
167 }
168 /**
169 -----
170 main
171 -----
172 int main() {
173     test_array_of_integers();
174     test_array_of_ptr_integers(5);
175     test_array_of_udt();
176     test_array_of_ptr_udt(5);
177     return 0;
178 }
179
180
181 //eof
182
```

3.5 Bag

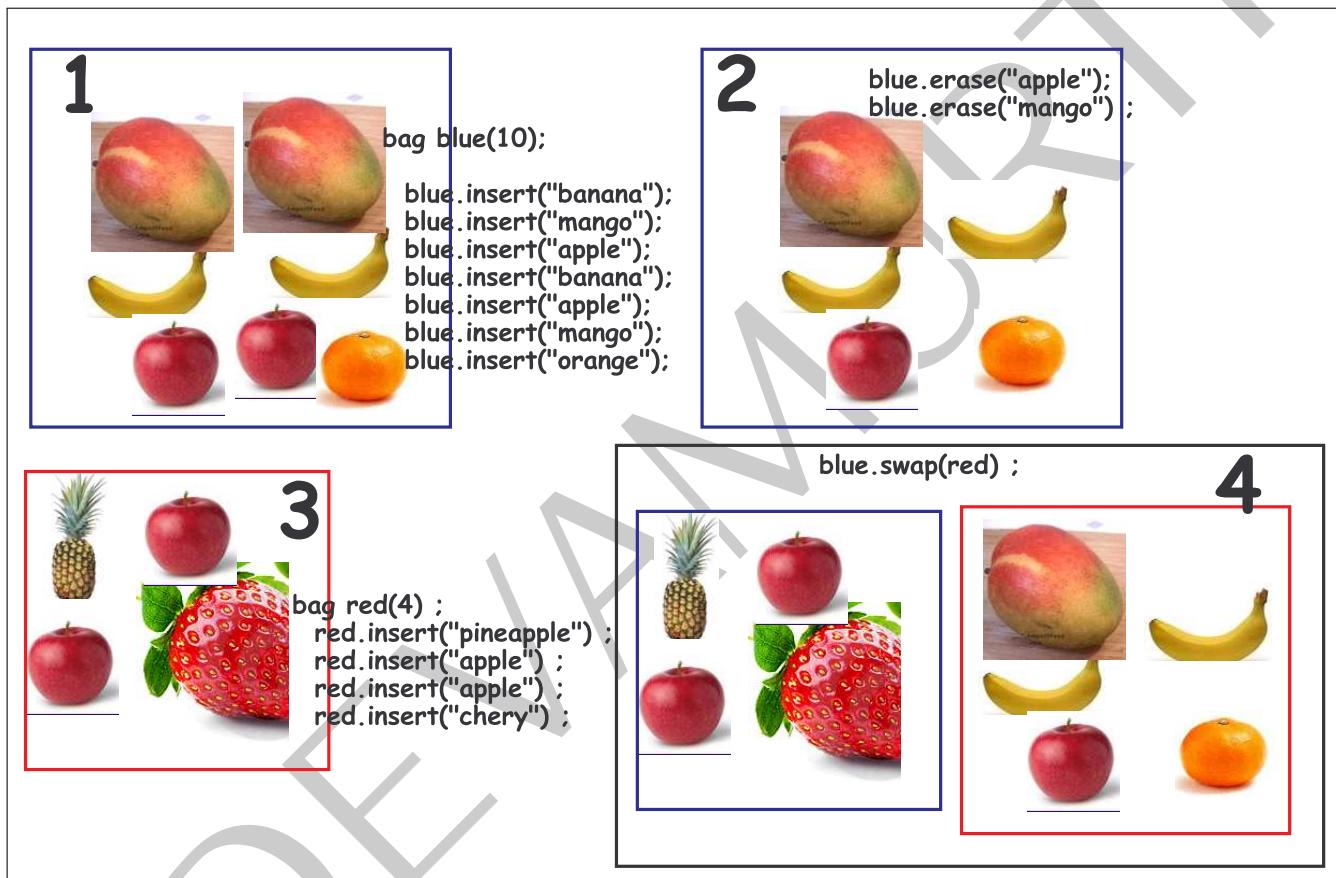


Figure 3.1: Bag of fruits

3.5. BAG

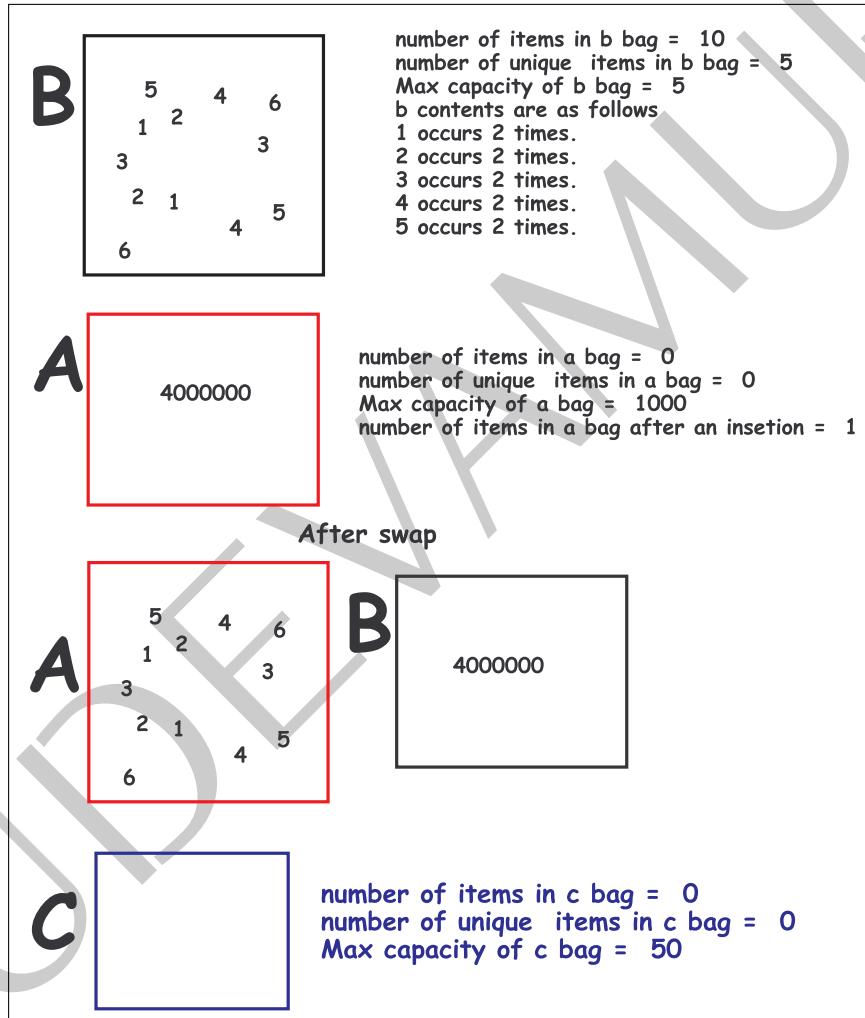


Figure 3.2: Bag of unsigned long numbers

3.5.1 Implementing bag in class *bag1* without using C++ templates

3.5.2 Implementing bag in class *bag* using C++ templates

3.6 Dynamic array of objects

```
c:\work\alg\course\objects\darray\darray.h
1  /*
2  Copyright (c) 2015 Author: Jagadeesh Vasudevamurthy
3  file: darray.h
4  */
5
6  /*
7  This file has darray class declaration
8  */
9
10 /*
11 All includes here
12 */
13 #ifndef darray_H
14 #define darray_H
15
16 #include "../util/util.h"
17
18 /*
19 Declaration of darray class
20 */
21 template <typename T>
22 class darray {
23 public:
24     explicit darray(int c = 50, bool d = false);
25     explicit darray(bool);
26     explicit darray(bool d, int c) = delete;
27     darray(const T f[], int c, bool display = false); //For constant object.
28     ~darray();
29     darray(const darray<T>& s);
30     darray<T>& operator=(const darray<T>& rhs);
31     T& operator[](int i); //For non constant objects
32     const T& operator[](int i) const; //for constant objects
33     bool display() const { return _display; }
34     void set_display(bool x) {
35         _display = x;
36     }
37     int size() const {
38         return _size;
39     }
40 private:
41     T* _element;
42     int _capacity;
43     bool _display;
44     int _size; //largest accessed + 1
45     void _copy(const darray<T>& s);
46     void _release(){ delete[] _element; }
47     void _grow(int i);
48 };
49
50 #include "darray.hpp"
51
52 #endif
```

```
1  /*
2  Copyright (c) 2015 Author: Jagadeesh Vasudevamurthy
3  file: darray.hpp
4
5  */
6
7  /*
8  This file has class definition
9  */
10 /*
11 Definition of routines of darray class
12 */
13 /*
14 */
15 /*
16 Constructor
17 */
18 template <typename T>
19 darray<T>::darray(int c, bool d) :_element(nullptr), _capacity(c), _display(d), _size(0){
20     if (display()) {
21         cout << "in darray constructor:" << endl;
22     }
23     _element = new T[_capacity]; //if T is by value, constructors will be called
24 }
25
26 /*
27 Constructor
28 */
29 template <typename T>
30 darray<T>::darray(bool d) :_element(nullptr), _capacity(50), _display(d), _size(0){
31     if (display()) {
32         cout << "in darray constructor:" << endl;
33     }
34     _element = new T[_capacity]; //if T is by value, constructors will be called
35 }
36
37 /*
38 Constructor that takes an array of const object T
39 and construct darray of constant object.
40 Note that after the object is constructed
41 you can NO longer add or change the content of darray
42 */
43 template <typename T>
44 darray<T>::darray(const T f[], int c, bool d) : _element(nullptr), _capacity(c), _size(c), _display(d){
45     if (display()) {
46         cout << "in darray constant constructor:" << endl;
47     }
48     _element = new T[_capacity]; //if T is by value, constructors will be called
49     for (int i = 0; i < _capacity; ++i) {
```

```

c:\work\alg\course\objects\darray\darray.hpp
50     _element[i] = f[i];
51 }
52 }
53
54 /*-----
55 Destructor
56 -----*/
57 template <typename T>
58 darray<T>::~darray() {
59     if (display()) {
60         cout << "In darray Destructor " << endl;
61     }
62     _release();
63     _element = nullptr;
64     _capacity = 0;
65     _size = 0;
66 }
67
68 /*-----
69 _copy
70 If by value: Calls copy constructor if written. Else bit wise copy
71 If by address: Just copies address. No deep copying.
72 (Two pointers will point to the same objects+
73 -----*/
74 template <typename T>
75 void darray<T>::_copy(const darray<T>& from) {
76     _capacity = from._capacity;
77     _size = from._size; //largest accessed + 1
78     _display = from._display;
79     _element = new T[_capacity]; //if T is by value, constructors will be called
80     for (int i = 0; i < from._size; i++) {
81         _element[i] = from._element[i]; //if T is by value, = operator will be called
82     }
83     //If T is by address, No constructors are called
84     //and array will be garbage as in C array
85     //NOTE THAT U cannot initialize this array to 0
86     //as you don't know T is by address
87 }
88
89 /*-----
90 Copy Constructor
91 -----*/
92 template <typename T>
93 darray<T>::darray(const darray<T>& s){
94     if (s.display()) {
95         cout << "in darray copy constructor:" << endl;
96     }
97     _copy(s);
98 }
99
100 /*-----
101 Equal Constructor

```

c:\work\alg\course\objects\darray\darray.hpp

```

102  -----*/
103  template <typename T>
104  darray<T>& darray<T>::operator=(const darray<T>& rhs) {
105      if (rhs.display()) {
106          cout << "In darray equal operator \n";
107      }
108      if (this != &rhs) {
109          _release();
110          _copy(rhs);
111      }
112      return *this;
113  }
114
115  /*
116  darray s ;
117  You can do
118  s[35] = obj ;
119  obj = s[78]
120  s[47] = p;
121  p = s[58] ;
122  s[89] = s[90] ;
123  -----
124  template <typename T>
125  T& darray<T>::operator[](int i) {
126      if (i < 0){
127          assert(0);
128      }
129      if (i >= _capacity) {
130          _grow(i);
131      }
132      if (i+1 > _size) {
133          _size = i+1; //largest accessed + 1
134      }
135      return _element[i]; /* Returned by alias so that user can modify */
136  }
137
138  /*
139  const darray s = "hello" ;
140  const ch = s[1] ;
141  -----
142  template <typename T>
143  const T& darray<T>::operator[](int i) const{
144      if (i < 0 || i >= _capacity) {
145          assert(0);
146      }
147      return _element[i]; /* Returned by alias so that user can modify */
148  }
149
150  /*
151  grow array
152  -----
153  template <typename T>
```

```
c:\work\alg\course\objects\darray\darray.hpp
154 void darray<T>::_grow(int i) {
155     if (display()) {
156         cout << "in darray grow. Will grow from " << _capacity << " to " << _capacity + i << endl;;
157     }
158     T* oldarray = _element;
159     int oldsize = _size; //largest accessed + 1
160     int oldcapacity = _capacity;
161     _capacity = _capacity + i;
162     _element = new T[_capacity]; //if T is by value, constructors will be called
163     for (int j = 0; j < oldsize; j++) { // largest accessed + 1
164         _element[j] = oldarray[j]; //if T is by value, = operator will be called
165     }
166     //If T is by address, No constructors are called
167     delete[] oldarray; //if T is by value, destructors will be called
168 }
169
170
171
172 //EOF
173
174
175
```

```
1 /*-
2 Copyright (c) 2015 Author: Jagadeesh Vasudevamurthy
3 file: ../complex/complex.cpp darraytest.cpp
4
5 On linux:
6 g++ ../complex/complex.cpp darraytest.cpp
7 valgrind --leak-check=full a.out
8 -- All heap blocks were freed -- no leaks are possible
9
10 -----
11 /*
12 This file test darray object
13 -----
14 */
15
16 /*
17 All includes here
18 */
19 #include "darray.h"
20 #include "../complex/complex.h"
21
22 /*
23 local to this file. Change verbose = true for debugging
24 */
25 static bool verbose = true;
26
27 /*
28 array of integers
29 */
30 static void test_array_of_integers(){
31     const int N = 5;
32     darray<int> a(3,verbose);
33     for (int i = 0; i < N; i++) {
34         a[i] = i * 10;
35     }
36     darray<int> b(a);
37     for (int i = 0; i < N; i++) {
38         cout << " b[" << i << "]=" << b[i];
39     }
40     cout << endl;
41     bool equal = true;
42     for (int i = 0; i < N; i++) {
43         if (a[i] != b[i]) {
44             equal = false;
45             break;
46         }
47     }
48     equal ? cout << " array a == b\n" : cout << " array a != b\n";
49     cout << endl;
50     cout << "We have not inserted 25th element. Let us see what we get " << endl;
51     cout << a[25] << endl;
52 }
53
54 /*
55 array of integer pointers
56 */
57 static void test_array_of_ptr_integers(int n){
58     darray<int*> a(3,verbose);
59     for (int i = 0; i < n; i++) {
60         a[i] = new int(i * 10);
61     }
62     darray<int*> b(a);
63     for (int i = 0; i < n; i++) {
64         cout << " b[" << i << "]=" << *(b[i]);
65     }
66     cout << endl;
```

```
67 bool equal = true;
68 for (int i = 0; i < n; i++) {
69     if (*(a[i]) != *(b[i])) {
70         equal = false;
71         break;
72     }
73 }
74 equal ? cout << " array a == b\n " : cout << " array a != b\n ";
75 cout << "We have not inserted 25th element. Let us see what we get " << endl;
76 cout << a[25] << endl;
77 for (int i = 0; i < n; i++) {
78     /* we allocated contents in a. So we delete it */
79     delete(a[i]);
80 }
81 /* Why you should NOT do this */
82 /*
83 for (int i = 0; i < n; i++) {
84     FREE(b[i]);
85 }
86 */
87 }
88
89 /**
90 array of user defined type
91 */
92 static void test_array_of_udt(){
93     const int N = 5;
94     darray<complex> a(3,verbose);
95     for (int i = 0; i < N; i++) {
96         a[i].setxy(i, -i);
97     }
98     darray<complex> b(a);
99     for (int i = 0; i < N; i++) {
100        cout << " b[" << i << "]=" << b[i];
101    }
102    cout << endl;
103    bool equal = true;
104    for (int i = 0; i < N; i++) {
105        if (a[i] != b[i]) {
106            equal = false;
107            break;
108        }
109    }
110    equal ? cout << " array a == b\n " : cout << " array a != b\n ";
111    cout << "We have not inserted 25th element. Let us see what we get " << endl;
112    cout << a[25] << endl;
113 }
114
115 /**
116 array of user defined pointer type
117 */
118 static void test_array_of_ptr_udt(int n){
119     darray<complex*> a(3,verbose);
120     for (int i = 0; i < n; i++) {
121         a[i] = new complex(i, -i);
122     }
123     darray<complex*> b(a);
124     for (int i = 0; i < n; i++) {
125         cout << " b[" << i << "]=" << *(b[i]);
126     }
127     cout << endl;
128     bool equal = true;
129     for (int i = 0; i < n; i++) {
130         if (*(a[i]) != *(b[i])) {
131             equal = false;
132             break;
133         }
134     }
135 }
```

```
133     }
134 }
135 equal ? cout << " array a == b\n" : cout << " array a != b\n";
136 cout << "We have not inserted 25th element. Let us see what we get " << endl;
137 //cout << a[25] << endl ;
138 for (int i = 0; i < n; i++) {
139     /* we allocated contents in a. So we delete it */
140     delete(a[i]);
141 }
142 /* Why you should NOT do this */
143 /*
144 for (int i = 0; i < n; i++) {
145     FREE(b[i]) ;
146 }
147 */
148 }
149 */
150 *-
151 array of constant integers
152 -----
153 static void test_constant_array_of_integer(){
154     static const int arr[] = { -89, -90, -78, -420, 1986, 2002 };
155     int size = sizeof(arr) / sizeof(int);
156     const darray<int> a(arr, size, verbose);
157     int k = a[2]; //WHICH EQUAL IS CALLED ?
158 //error C3892: 'a' : you cannot assign to a variable that is const
159 //a[1] = a[2] ;
160     for (int i = 0; i < size; ++i) {
161         cout << " a[" << i << "]=" << a[i];
162     }
163     const darray<int>b(a);
164     for (int i = 0; i < size; ++i) {
165         cout << " b[" << i << "]=" << b[i];
166     }
167 //'a' : you cannot assign to a variable that is const
168 //a[10] = 789 ;
169 }
170 */
171 *-
172 array of constant complex
173 -----
174 static void test_constant_array_of_complex(){
175     static const complex arr[] = { -89, -90, -78, -420, 1986, 2002 };
176     int size = sizeof(arr) / sizeof(complex);
177     const darray<complex> a(arr, size,verbose);
178     complex k = a[2]; //WHICH EQUAL IS CALLED ?
179 //error C3892: 'a' : you cannot assign to a variable that is const
180 //a[1] = a[2] ;
181     for (int i = 0; i < size; ++i) {
182         cout << " a[" << i << "]=" << a[i];
183     }
184     const darray<complex>b(a);
185     for (int i = 0; i < size; ++i) {
186         cout << " b[" << i << "]=" << b[i];
187     }
188 /**
189     complex c(90, -9);
190 //a[10] = c ;
191 }
192 */
193 *-
194 problem
195 */
196 MORAL: DO NOT STORE ALIAS TO THE CONTAINER OBJECT
197 -----
198 static void test_problem() {
```

```
199 darray<complex> vc(3,verbose);
200 vc[0] = 0;
201 vc[1] = 1;
202 vc[2] = 2;
203
204 complex& c1 = vc[1];
205 cout << " c1 = " << c1 << endl;
206
207 vc[3] = 3;
208 vc[4] = 4;
209
210 cout << " c1 = " << c1 << endl; //WHAT HAAPENS HERE ????????????
211 }
212
213
214 /*-----*
215 main
216 -----*/
217 int main() {
218     complex::set_display(verbose);
219     test_array_of_integers();
220     cout << " _____ " << endl;
221     test_array_of_ptr_integers(5);
222     cout << " _____ " << endl;
223     test_array_of_udt();
224     cout << " _____ " << endl;
225     test_array_of_ptr_udt(5);
226     cout << " _____ " << endl;
227     test_constant_array_of_integer();
228     cout << " _____ " << endl;
229     test_constant_array_of_complex();
230     cout << " _____ " << endl;
231     if (0) {
232         test_problem();
233         cout << " _____ " << endl;
234     }
235     return 0;
236 }
237
238 //EOF
239
240
```

3.7 Problem set

Problem 3.7.1. Implement the *bag1* class without using C++ template. You must write **bag1.h** and **bag1.cpp**. You must test your class using the given **bag1test.cpp**.

```
1 /*-----  
2 Copyright (c) 2014 Author: Jagadeesh Vasudevamurthy  
3 file: bag1test.cpp  
4  
5 On linux:  
6 g++ bag1.cpp bag1test.cpp  
7 valgrind a.out  
8 ==29182== All heap blocks were freed -- no leaks are possible  
9 ==15767== All heap blocks were freed -- no leaks are possible  
10 -----*/  
11  
12 /*-----  
13 This file test bag1 object. NOTHING HAS TO BE CHANGED IN THISFILE  
14 -----*/  
15  
16 /*-----  
17 All includes here  
18 -----*/  
19 #include "bag1.h"  
20  
21 #ifdef TEST_STRING  
22  
23 /*-----  
24 Testing strings  
25 -----*/  
26 static void string_tests() {  
27     bag1 blue(10);  
28     blue.insert("banana");  
29     blue.insert("mango");  
30     blue.insert("apple");  
31     blue.insert("banana");  
32     blue.insert("apple");  
33     blue.insert("mango");  
34     blue.insert("orange");  
35  
36     cout << "blue bag1 contents are as follows" << endl;  
37     for (blue.start(); !blue.ended(); blue.next()) {  
38         cout << blue.currentvalue() << " occurs " << blue.currentcount() << " times." << endl;  
39     }  
40     cout << "number of items in blue bag1 = " << blue.number_of_elements() << endl;  
41     cout << "number of unique items in blue bag1 = " << blue.size() << endl;  
42     cout << "Number of banana in blue bag1 = " << blue.count("banana") << endl;  
43     cout << "Number of apple in blue bag1 = " << blue.count("apple") << endl;  
44  
45     assert(blue.contains("apple"));  
46     assert(!(blue.contains("fig")));  
47     blue.erase("apple");  
48     blue.erase("mango");  
49     cout << "After erasing(eating) some fruits from blue bag1" << endl;  
50     cout << "blue bag1 contents are as follows" << endl;  
51     for (blue.start(); !blue.ended(); blue.next()) {  
52         cout << blue.currentvalue() << " occurs " << blue.currentcount() << " times." << endl;  
53     }  
54     cout << "number of items in blue bag1 = " << blue.number_of_elements() << endl;  
55     cout << "number of unique items in blue bag1 = " << blue.size() << endl;  
56     cout << "Number of banana in blue bag1 = " << blue.count("banana") << endl;  
57     cout << "Number of apple in blue bag1 = " << blue.count("apple") << endl;  
58  
59 //bag1 yellow(blue) ; //'bag1::bag1' : cannot access private member declared in class 'bag1'  
60 bag1 red(4);  
61 //red = blue ; //'bag1::operator =' : cannot access private member declared in class 'bag1'  
62 red.insert("pineapple");  
63 red.insert("apple");  
64 red.insert("apple");  
65 red.insert("chery");  
66
```

```
67 cout << "red bag1 contents are as follows" << endl;
68 for (red.start(); !red.ended(); red.next()) {
69     cout << red.currentvalue() << " occurs " << red.currentcount() << " times." << endl;
70 }
71 cout << "number of items in red bag1 = " << red.number_of_elements() << endl;
72 cout << "number of unique items in red bag1 = " << red.size() << endl;
73 cout << "Number of banana in red bag1 = " << red.count("banana") << endl;
74 cout << "Number of apple in red bag1 = " << red.count("apple") << endl;
75 blue.swap(red);
76 cout << "after swaping blue bag1 contents with red bag1 contents\n";
77 cout << "blue bag1 contents are as follows" << endl;
78 for (blue.start(); !blue.ended(); blue.next()) {
79     cout << blue.currentvalue() << " occurs " << blue.currentcount() << " times." << endl;
80 }
81 cout << "red bag1 contents are as follows" << endl;
82 for (red.start(); !red.ended(); red.next()) {
83     cout << red.currentvalue() << " occurs " << red.currentcount() << " times." << endl;
84 }
85 red.eraseAll("apple");
86 assert(!(red.contains("apple")));
87 }
88
89 #endif
90
91 #ifdef TEST_UNSIGNED_LONG
92 /**
93 Testing unsigned long
94 */
95 void unsigned_long_test() {
96     bag1 a(1000); // a can hold at most 1000 distinct items
97     bag1 b(5); // b can hold at most 5 distinct items
98     bag1 c; // c can hold at most 50 item. See default constructor
99
100
101    T v[6] = { 1, 2, 3, 4, 5, 6 };
102    // No failures inserting 5 distinct items twice each into b
103    for (int k = 0; k < 5; k++) {
104        {
105            assert(b.insert(v[k]));
106            assert(b.insert(v[k]));
107        }
108        cout << "number of items in b bag1 = " << b.number_of_elements() << endl;
109        cout << "number of unique items in b bag1 = " << b.size() << endl;
110        cout << "Max capacity of b bag1 = " << b.capacity() << endl;
111        cout << "b contents are as follows" << endl;
112        for (b.start(); !b.ended(); b.next()) {
113            cout << b.currentvalue() << " occurs " << b.currentcount() << " times." << endl;
114        }
115        cout << "number of items in a bag1 = " << a.number_of_elements() << endl;
116        cout << "number of unique items in a bag1 = " << a.size() << endl;
117        cout << "Max capacity of a bag1 = " << a.capacity() << endl;
118        a.insert(4000000);
119        cout << "number of items in a bag1 after an insertion = " << a.number_of_elements() << endl;
120        a.swap(b);
121        cout << "number of items in a bag1 = " << a.number_of_elements() << endl;
122        cout << "number of unique items in a bag1 = " << a.size() << endl;
123        cout << "number of items in b bag1 = " << b.number_of_elements() << endl;
124        cout << "number of unique items in b bag1 = " << b.size() << endl;
125
126        cout << "b contents are as follows" << endl;
127        for (b.start(); !b.ended(); b.next()) {
128            cout << b.currentvalue() << " occurs " << b.currentcount() << " times." << endl;
129        }
130        cout << "a contents are as follows" << endl;
131        for (a.start(); !a.ended(); a.next()) {
132            cout << a.currentvalue() << " occurs " << a.currentcount() << " times." << endl;
```

```
133 }
134 cout << "number of items in c bag1 = " << c.number_of_elements() << endl;
135 cout << "number of unique items in c bag1 = " << c.size() << endl;
136 cout << "Max capacity of c bag1 = " << c.capacity() << endl;
137 }
138 #endif
139
140 int main() {
141 #ifdef TEST_STRING
142     string_tests();
143 #endif
144 #ifdef TEST_UNSIGNED_LONG
145     unsigned_long_test();
146 #endif
147 }
148
149
150 //EOF
151
152
```

CHAPTER 3. DYNAMIC ARRAY

Problem 3.7.2. Implement the **bag** class, called **class bag**, using **template**. You must write **bag.h** and **bag.cpp**. You must test your class using the given **bagtest.cpp**.

Problem 3.7.3. Reimplement the **darray** class as **darray1** class. In this class use **T*** _element**, instead of **T* _element**. Implement all the code in **darray1.h** file as follows:

```
/*-----  
class darray1  
-----*/  
template <typename T>  
class darray1 {  
public:  
    darray1(int size = 50, bool d = false);  
    darray1(const T f[], int size, bool d = false) ; //For constant object.  
    ~darray1();  
    darray1(const darray1<T>& s);  
    darray1<T>& operator=(const darray1<T>& rhs) ;  
    T& operator[](int i); //For non constant objects  
    const T& operator[](int i) const ; //for constant objects  
    bool display()const {return _display;}  
    void set_display(bool x) {  
        _display = x;  
    }  
  
private:  
    T*** _element;  
    int _size ;  
    bool _display ;  
    void _copy(const darray1<T>& s) ;  
    void _release(){FREEVEC(_element);} ;  
    void _grow(int i) ;  
};
```

Write a word document, along with the pictures, that explains the advantages and disadvantages of using **T*** _element**, instead of **T* _element**.

Problem 3.7.4. Write a string class **str** using the dynamic array class **darray**. You cannot use any object other than **darray** in your **class str**. You must use the **strtest.cpp** to test your program.

```
1 /*-----  
2 Copyright (c) 2014 Author: Jagadeesh Vasudevamurthy  
3 file: strtest.cpp  
4  
5 On linux:  
6 g++ str.cpp strtest.cpp  
7 valgrind --leak-check=full -v a.out  
8  
9 -- All heap blocks were freed -- no leaks are possible  
10  
11 -----*/  
12  
13 /*-----  
14 This file test str object  
15 -----*/  
16  
17 /*-----  
18 All includes here  
19 -----*/  
20 #include "str.h"  
21  
22 /*-----  
23 local to this file. Change verbose = true for debugging  
24 -----*/  
25 static bool verbose = true;  
26  
27 /*-----  
28 main  
29 -----*/  
30 void test1() {  
31     const char *j = "jag";  
32     cout << "length of j = " << strlen(j) << endl;  
33     str s1('U', verbose);  
34     str s2("jag", verbose);  
35     cout << s2 << endl;  
36     s2.reverse();  
37     cout << s2 << endl;  
38     str s3(s2);  
39     cout << s1 << endl;  
40     cout << s2 << endl;  
41     cout << s3 << endl;  
42     s2 = s1;  
43     cout << s2 << endl;  
44     cout << s1 << endl;  
45     if (s1 == s2) {  
46         cout << "s1 == s2" << endl;  
47     }  
48     if (s1 != s3) {  
49         cout << "s1 != s3" << endl;  
50     }  
51     str s10("abcd", verbose);  
52     str s11("abc", verbose);  
53     int x = string_compare(s10, s11);  
54     s3 = 'a' + s1;  
55     cout << "s1 = " << s1 << endl;  
56     cout << "s3 = " << s3 << endl;  
57     s3 = s1 + 'p';  
58     cout << "s3 = " << s3 << endl;  
59 //s3 = "abc" + "123" ;  
60 // You cannot do this. At least one in RHS must be str  
61 s3 = str("abc") + "123";  
62 cout << "s3 = " << s3 << endl;  
63 s1 = "++";  
64 s2 = "Claaaa";  
65 cout << "Jag " << 'C' + s1 + '+' + '+' + ' ' << s2 << "s" << endl;  
66 }
```

```
67
68 -----
69 main
70 ----- */
71 int main() {
72     test1();
73     return 0;
74 }
75
76 //EOF
77
78
```

3.7. PROBLEM SET

Problem 3.7.5. Reimplement **darray** class. Use **linked list** instead of extending the array.

Problem 3.7.6. Implement **dpnarray** class, where user can access both the positive and negative index of the array. For example $a[-200] = a[88] = 8602;$. Write a main program that tests your array

Problem 3.7.7. Refer to the Figure 3.3. Implement **class ulongnum** that behaves like primitive data type **int**. You cannot use any object other than **str** in your **class ulongnum**. You must use the **ulongnumtest.cpp** to test your program.

```
1 /*-
2 Copyright (c) 2015 Author: Jagadeesh Vasudevamurthy
3 file: ../str/str.cpp ulongnum.cpp ulongnumtest.cpp
4
5 On linux:
6 g++ ..\str\str.cpp ulongnum.cpp ulongnumtest.cpp
7 valgrind a.out
8 valgrind --leak-check=full -v a.out
9 -- All heap blocks were freed -- no leaks are possible
10
11 -----*/
12
13 /*-
14 This file test ulongnum object
15 -----*/
16
17 /*
18 All includes here
19 -----*/
20 #include "ulongnum.h"
21
22 /*
23 local to this file. Change verbose = true for debugging
24 -----*/
25 static bool verbose = false;
26
27 /*
28 test multiplication
29 -----*/
30 static void test_multiplication() {
31     ulongnum a(789, verbose);
32     cout << "a = " << a << endl;
33     ulongnum b("56", verbose);
34     cout << "b = " << b << endl;
35     ulongnum ans = a * b;
36     cout << "ans = " << ans << endl;
37     assert(ans == 44184);
38
39     ulongnum rsa129(
40         "1143816257578886766923577997614661201021829672124236256256184293570693524573389783059712356395870",
41         "5058989075147599290026879543541", verbose);
42     ulongnum p1("3490529510847650949147849619903898133417764638493387843990820577", verbose);
43     ulongnum p2("32769132993266709549961988190834461413177642967992942539798288533", verbose);
44     ulongnum p1p2 = p1 * p2;
45     cout << "p1 = " << p1 << endl;
46     cout << "p2 = " << p2 << endl;
47     cout << "p1p2 = " << p1p2 << endl;
48     assert(p1p2 == rsa129);
49 }
50
51 /*-
52 test addition
53 -----*/
54 static void test_addition() {
55     ulongnum a(9789, verbose);
56     ulongnum b("100000", verbose);
57     ulongnum c('7', verbose);
58     cout << "a = " << a << endl;
59     cout << "b = " << b << endl;
60     cout << "c = " << c << endl;
61     ulongnum sum = a + 78 + b + c;
62     cout << "sum = a + 78 + b + c = " << sum << endl;
63     assert(sum == 109874);
64 }
```



```
103     ulongnum c1000;
104     c1000.set_display(verbose);
105     c1000.factorial(1000);
106     cout << "Factorial of 1000 = " << endl;
107     cout << c1000 << endl;
108     assert(c1000 == fact1000);
109     clock_t end = clock();
110     double d = double(end - start) / CLOCKS_PER_SEC;
111     cout << "Run time for !1000 = " << " is " << d << " secs" << endl;
112 }
113 }
114
115 /*-
116 main
117 -----*/
118 int main() {
119     test_basic();
120     test_addition();
121     test_multiplication();
122     test_factorial();
123     return 0;
124 }
125
126 //EOF
127
128
```

3.7. PROBLEM SET

unsigned longnum class

Write a class called longnum, in file **ulongnum.h**, which can hold arbitrarily long numbers.

The class **MUST USE ONLY** the following two fields:

1. str object as a data member
2. bool _display

```
class ulongnum {  
public:  
    /* CANNOT HAVE ANY DATA FIELDS */  
    /* CAN HAVE ANY PUBLIC FUNCTION */  
private:  
    str _ulongnum ;  
    bool _display ;  
    /* CAN HAVE ANY PRIVATE FUNCTION */  
};
```

1. The user should be able to use this class like built-in type int object
2. The class should have all the constructors, destructors, copy constructor and equal operator
3. The class should have operators + and *.
(you can implement add and mult if you don't know operator overloading)
4. One of the function in that class must be factorial function.
factorial (100) should produce result like:

```
9332621544394415268169923885626670049071596826438162146859  
29638952175999932299156089414639761565182862536979208272237582511852109168  
640000000000000000000000000000000000
```

Use **ulongnumtest.cpp** to test the program. **YOU CANNOT CHANGE ANY LINE** in **ulongnumtest.cpp**

On windows:

Must use Visual leak detector. At the end of execution, you must see
No memory leaks detected.
Visual Leak Detector is now exiting.

On Linux:

```
Finished a.out          ( 0 errors, 0 leaked bytes)  
Purify instrumented ./a.out (pid 14509 at Mon Oct 10 14:12:21 2011)  
Command-line: ./a.out  
Current file descriptors in use: 5  
Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)  
Program exited with status code 0.
```

Figure 3.3: unsigned long number class

CHAPTER 3. DYNAMIC ARRAY

Problem 3.7.8. Repeat problem 3.7.7. Implement long number class, that can works on both positive and negative numbers.

Problem 3.7.9. GOOGLE interview question: You are given an array with integers (both positive and negative) in any random order. Find the sub-array with the largest sum.

Problem 3.7.10. GOOGLE interview question: Given an array of size N in which every number is between 1 and N , determine if there are any duplicates in it. You are allowed to destroy the array if you like.

Problem 3.7.11. GOOGLE interview question: Write, efficient code for extracting unique elements from a sorted list of array. e.g. $(1, 1, 3, 3, 3, 5, 5, 5, 9, 9, 9) \rightarrow (1, 3, 5, 9)$.

3.7. PROBLEM SET

VASUDEVAMURTHY

Chapter 4

STL Vector Class

4.1 Introduction

4.2 Iterator design pattern

```
1 /*-----
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy
3 Filename: iterator.cpp
4 compile: g++ iterator.cpp
5 -----
6 #include <iostream>
7 using namespace std;
8
9 #ifndef _WIN32
10 #include <vld.h> //comment this line, if you have NOT installed Visual leak detector
11 #endif
12
13 /*
14 class iarray_iterator
15 -----
16 class iarray_iterator {
17 public:
18     iarray_iterator(int* x = 0): _current(x) {}
19     ~iarray_iterator() {}
20     // *(itt)
21     const int& operator*() const{
22         return *(_current);
23     }
24     //++itt
25     iarray_iterator& operator++() {
26         ++(_current);
27         return *this;
28     }
29     //if (itt != x.end())
30     bool operator!=(const iarray_iterator& rhs) {
31         return (_current != rhs._current);
32     }
33 private:
34     int* _current;
35 };
36
37 /*
38 class iarray
39 -----
40 class iarray {
41 enum {START = 0, END = 3};
42 public:
43     typedef iarray_iterator iterator;
44     iarray() { for (int i = START; i < END; i++) _a[i] = i*10 ; }
45     ~iarray(){}
46
47     iterator begin() {return iterator(&_a[START]);}
48     iterator end() {return iterator(&_a[END]);}
49
50 private:
51     int _a[END];
52 };
53
54 /*
55 Main program
```

```
56 -----*/
57 int main() {
58     iarray x;
59     //iarray_iterator itt = x.begin();
60     iarray::iterator itt = x.begin();
61     while (itt != x.end()) {
62         const int& p = *(itt);
63         cout << p << " ";
64         ++itt;
65     }
66     cout << endl;
67     return 0;
68 }
69
```

4.2. ITERATOR DESIGN PATTERN

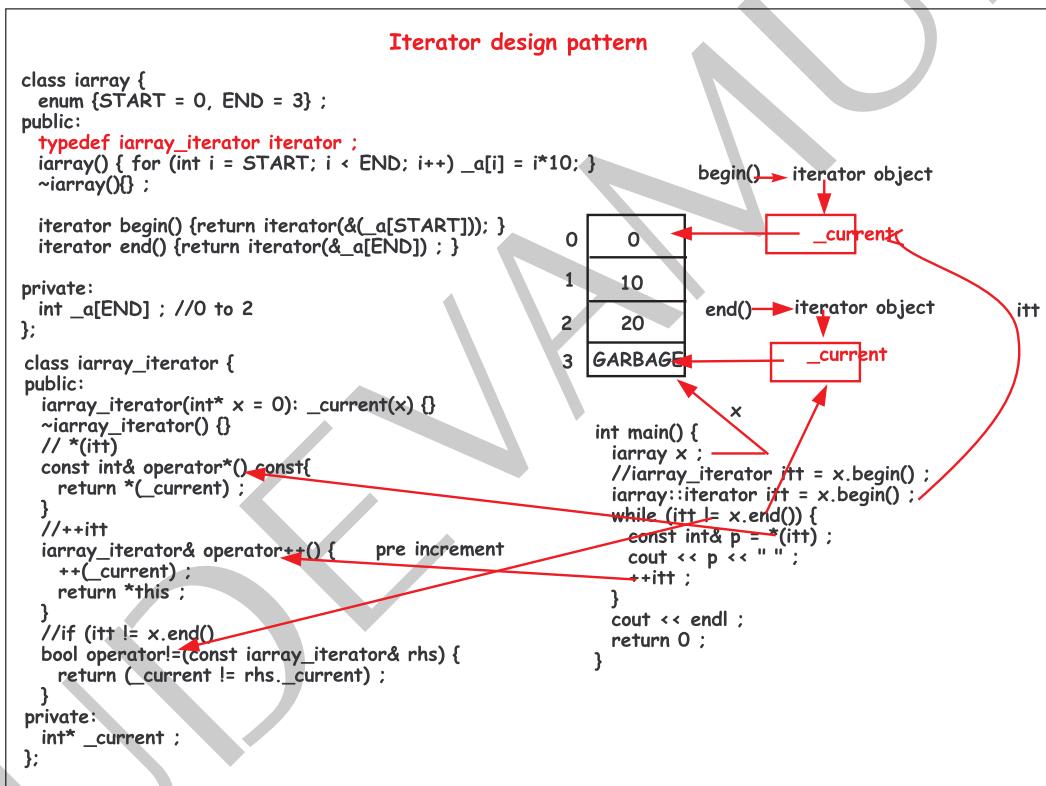


Figure 4.1: iterator design pattern

4.3 Vector class public functions

4.3. VECTOR CLASS PUBLIC FUNCTIONS

Constructors		
<code>vector<T> a</code>	Default constructor	$O(1)$
<code>vector<T> a(size)</code>	Default constructor with explicit size	$O(\text{size})$
<code>vector<T> a(size, T)</code>	Default constructor with explicit size and initial value	$O(\text{size})$
Size		
<code>a.size()</code>	Number of elements currently held in vector a	$O(1)$
<code>a.capacity()</code>	Maximum elements vector a can hold	$O(1)$
<code>a.clear()</code>	All elements are removed size = 0 and capacity = unchanged	$O(n)$
Element access		
<i>Returns object by alias</i>		
<code>a[i]</code>	Access i th element of array a i must be less than size of a boundary checking is not done	$O(1)$
<code>a.at[i]</code>	Access i th element of array a Boundary check is done before access. Throws exception if $(i < 0) \text{ } (i \geq \text{size})$	$O(1)$
<code>a.push_back(i)</code>	Add element to the end of the array a.	$O(1)$ OR $O(n)$
<code>a.pop_back()</code>	Remove element from the end of the array a.	$O(1)$

Figure 4.2: `vector` class functions

4.4 Vector class iterator

4.4. VECTOR CLASS ITERATOR

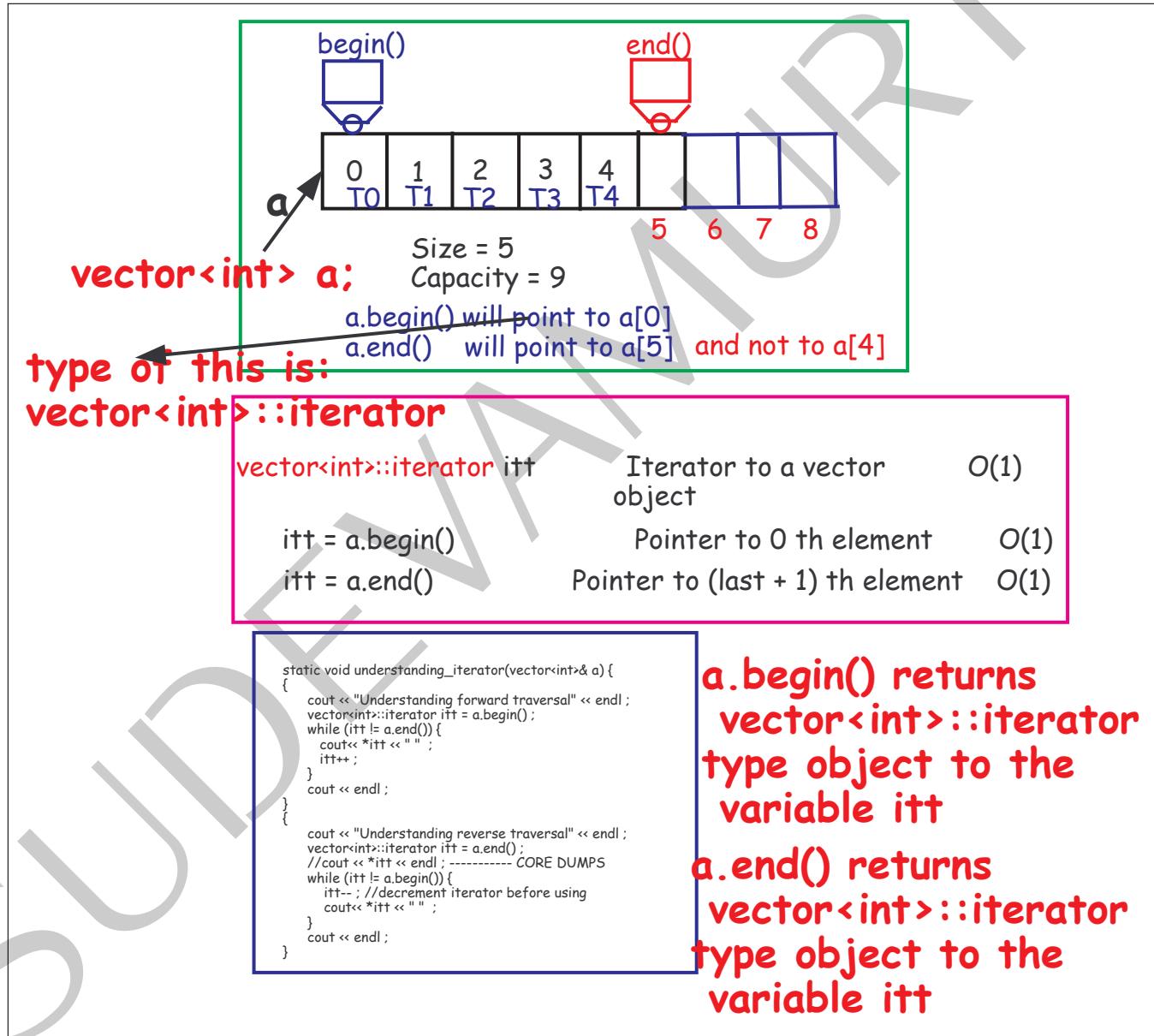


Figure 4.3: vector iterator functions

4.5 Understanding vector class and iterators

```
1  /*
2  Copyright (c) 2018 Author: Jagadeesh Vasudevamurthy
3  Filename: svector.cpp complex.cpp util.cpp
4  compile: g++ svector.cpp complex.cpp util.cpp
5  Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)
6  */
7 #include <vector>
8 #include <stdexcept> //Without this catch will NOT work on Linux
9
10 #ifdef _WIN32 //Will not work in windows7 if U write WIN32
11 #include "..\complex\complex.h"
12 #else
13 #include "../complex/complex.h"
14 #endif
15
16 /*
17 Multiply integer by 10
18 */
19 static void multiply_by_10(int& x) {
20     x = x * 10;
21 }
22
23 /*
24 apply a function pf on elements of vector a
25 */
26 template <typename T>
27 static void apply(const char* s, vector<T>& a, void(*pf)(T& x)) {
28     cout << s << endl;
29     cout << "-----" << endl;
30     auto itt = a.begin();
31     while (itt != a.end()) {
32         T& p = *itt;
33         pf(p);
34         itt++;
35     }
36     cout << endl;
37     /* apply using range */
38     for (auto& p : a) {
39         pf(p);
40     }
41 }
42
43 /*
44 a[0] ..... a[9]
45
46 begin() will point to a[0]
47 end() will NOT POINT to a[9], but to one element past a[9]
48 That means real end is: end()-1 ;
49 */

```

```
50 static void understanding_iterator(vector<int>& a) {
51 {
52     cout << "Understanding forward traversal" << endl;
53     vector<int>::iterator itt = a.begin();
54     while (itt != a.end()) {
55         cout << *itt << " ";
56         itt++;
57     }
58     cout << endl;
59 }
60 {
61     cout << "Understanding forward traversal using auto" << endl;
62     auto itt = a.begin();
63     while (itt != a.end()) {
64         cout << *itt << " ";
65         itt++;
66     }
67     cout << endl;
68 }
69 {
70     cout << "Understanding forward traversal using range" << endl;
71     for (const auto& i : a) {
72         cout << i << " ";
73     }
74     cout << endl;
75 }
76 {
77     cout << "Understanding reverse traversal" << endl;
78     vector<int>::iterator itt = a.end();
79 //cout << *itt << endl ; ----- CORE DUMPS
80     while (itt != a.begin()) {
81         itt--; //decrement iterator before using
82         cout << *itt << " ";
83     }
84     cout << endl;
85 }
86 {
87     cout << "Understanding reverse traversal using auto" << endl;
88     auto itt = a.end();
89 //cout << *itt << endl ; ----- CORE DUMPS
90     while (itt != a.begin()) {
91         itt--; //decrement iterator before using
92         cout << *itt << " ";
93     }
94     cout << endl;
95 }
96 cout << "-----" << endl;
97 }
```

```

99
100 /*-----
101 Understanding access using
102 1. a[i]
103 2. a.at(i) -- Same as a[i], but does out of bound checks
104 and throws exception that you can catch
105
106 -----*/
107 template <typename T>
108 static void understanding_access(const vector<T>& a, int i) {
109     //int p = a[i] ; //core dumps
110     try {
111         int p = a.at(i); //Throws exception. out of bound checks
112
113     } catch (std::out_of_range) {
114         cout << " accessing a[" << i << "] but size of array is " << a.size() << endl;
115     }
116     cout << "-----" << endl;
117 }
118
119 /*-----
120 print1
121 -----*/
122 template <typename T>
123 static void print1(const char* s, const vector<T>& a) {
124     cout << s << endl;
125     cout << "-----" << endl;
126     cout << "size = " << a.size() << " ";
127     cout << "capacity = " << a.capacity() << " ";
128     cout << endl;
129     for (int i = 0; i < int(a.size()); i++) {
130         cout << "a[" << i << "] = " << a[i] << " ";
131     }
132     cout << endl;
133     cout << "-----" << endl;
134 }
135
136 /*-----
137 basic
138 -----*/
139 static void basic() {
140     vector<int> a;
141     print1("begin with", a);
142     for (int i = 0; i < 5; i++) {
143         a.push_back(i);
144     }
145     print1("After Inserting 5 elements", a);
146     understanding_iterator(a);

```

```
147
148     understanding_access(a, 6);
149
150     a.pop_back();
151     a.pop_back();
152     print1("After deleting last 2 elements", a);
153
154     apply("multiply by 10", a, multiply_by_10);
155     cout << endl;
156     apply("print using iterator", a, print_integer);
157     cout << endl;
158
159     a.clear();
160     print1("Using clear", a);
161 }
162
163 /*-----
164 questions
165 -----*/
166 static void questions() {
167     vector<int> a(10, -1);
168     print1("Is it -1", a);
169     cout << "I am reading without assigning a[5] = " << a[5] << endl;
170
171     complex c(99, -420);
172     vector<complex> b(6, c);
173     print1("Is it 99,-420", b);
174     cout << "I am reading without assigning b[5] = " << b[5] << endl;
175
176 //cout << b[7] << endl ; -- Crashes because you need to push_back before U      ↵
177     access
178 }
179
180 /*-----
181 problem
182
183 MORAL: DO NOT STORE ALIAS TO THE CONTAINER OBJECT
184 -----*/
185 static void problem() {
186     vector<complex> vc;
187     print1("begin with ", vc);
188     vc.push_back(0);
189     vc.push_back(1);
190     vc.push_back(2);
191
192     print1("After Inserting 0, 1 and 2 ", vc);
193
194     complex& c1 = vc[1];
195     cout << " c1 = " << c1 << endl;
```

```

195
196     for (int i = 3; i < 10; ++i) {
197         vc.push_back(i);
198     }
199     print1("After Inserting 3 to 10 ", vc);
200
201     //WHAT HAPPENS HERE ??????????????
202     if (0) {
203         cout << " c1 = " << c1 << endl;
204     }
205 }
206
207 /*-----*
208 array of integer pointers
209 -----*/
210 static void test_vector_of_ptr_integers(int n) {
211     vector<int*> a;
212     for (int i = 0; i < n; i++) {
213         a.push_back(new(int)(i * 10));
214     }
215     vector<int*> b(a);
216     apply("Contents of vector b", b, print_integer);
217     bool equal = true;
218     for (int i = 0; i < n; i++) {
219         if (*(a[i]) != *(b[i])) {
220             equal = false;
221             break;
222         }
223     }
224     equal ? cout << " array a == b\n " : cout << " array a != b\n ";
225     //YOU CANNOT DO THIS with vector
226     //cout << "We have not inserted 25th element. Let us see what we get " <<
227     //endl ;
228     //cout << a[25] << endl ;
229     for (int i = 0; i < n; i++) {
230         /* we allocated contents in a. So we delete it */
231         delete(a[i]);
232     }
233     /* Why you should NOT do this */
234     /* */
235     for (int i = 0; i < n; i++) {
236         delete(b[i]) ;
237     }
238 }
239
240 /*-----*
241 array of user defined type
242 -----*/

```

4.6 Problem set

Problem 4.6.1. Refer to the problem 3.7.4. Implement the string class using **STL vector class** instead of class **darray**. The output of the program must be exactly same as in the problem 3.7.4

Problem 4.6.2. Refer to the problem 3.7.8. Implement longnum class using the string class obtained from 4.6.2, above.

VASUDEVAMURTHY

Chapter 5

STL string Class

5.1 Introduction

5.2 string class public functions

5.2. STRING CLASS PUBLIC FUNCTIONS

C++ STL STRING					
Member functions					
<table><tr><td>(constructor)</td><td>Construct string object (constructor member)</td></tr><tr><td>operator=</td><td>String assignment (public member function)</td></tr></table>		(constructor)	Construct string object (constructor member)	operator=	String assignment (public member function)
(constructor)	Construct string object (constructor member)				
operator=	String assignment (public member function)				
Iterators:					
begin	Return iterator to beginning (public member function)				
end	Return iterator to end (public member function)				
rbegin	Return reverse iterator to reverse beginning (public member function)				
rend	Return reverse iterator to reverse end (public member function)				
Capacity:					
size	Return length of string (public member function)				
length	Return length of string (public member function)				
max_size	Return maximum size of string (public member function)				
resize	Resize string (public member function)				
capacity	Return size of allocated storage (public member function)				
reserve	Request a change in capacity (public member function)				
clear	Clear string (public member function)				
empty	Test if string is empty (public member function)				
Element access:					
operator[]	Get character in string (public member function)				
at	Get character in string (public member function)				
Modifiers:					
operator+=	Append to string (public member function)				
append	Append to string (public member function)				
push_back	Append character to string (public member function)				
assign	Assign content to string (public member function)				
insert	Insert into string (public member function)				
erase	Erase characters from string (public member function)				
replace	Replace part of string (public member function)				
swap	Swap contents with another string (public member function)				

Figure 5.1: **string** class functions

Searching a string

string search member functions

Member function	Purpose
find()	find the first position of the specified substring
find_first_of()	equal to find(), but finds the first position of any character specified
find_last_of()	equal to find first of(), but finds the last position of any character specified
find_first_not_of()	equal to find first of(), but returns the position of the first character not of those specified
find_last_not_of()	equal to find last of(), but returns the last position of any characters not specified
rfind()	equal to find(), but searches backwards

All member functions return `string::size_type`

```

133  /*-
134   * testing find
135
136   str: Hello, can you find UCSC ext? at Santa clara UCSC
137   size = 49
138   First occurrence of UCSC was found at: 20
139   */
140 static void test_find() {
141     string str("Hello, can you find UCSC ext? at Santa clara UCSC");
142     string::size_type position = str.find("UCSC");
143     print1("str: ",str) ;
144     cout << "First occurrence of UCSC was found at: " << position << endl;
145 }
```

Figure 5.2: `string` class `find` functions

5.3 Testing string code

```
1 /*-
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy
3 Filename: string.cpp
4 compile: g++ sting.cpp
5 -----
6 #include <iostream>
7 #include <string>
8 #include <vector>
9
10 using namespace std;
11
12 /*-
13 print1
14 -----
15 static void print1(const char* title, const string& s) {
16     cout << title ;
17     cout << s << endl ;
18     cout << "size = " << s.size() << endl ;
19 }
20
21 /*-
22 basic tests
23 s4 is an empty string
24 s = : abcd
25 size = 4
26 s[3] = d
27 s1 = : abcd
28 size = 4
29 s1 == s
30 s != abc
31 s = : abcdbabu
32 size = 8
33 s = : abcdbabuz
34 size = 9
35 s = : abcdbabuzT
36 size = 10
37 s = : abcdbabuzT78
38 size = 12
39 s after clear:
40 size = 0
41 -----
42 static void test_basic(){
43     string s4 ;
44     /* How to test the string is empty */
45     if (s4.empty()) {
46         cout << "s4 is an empty string" << endl ;
47     }
48     string s = "abcd" ;
49     print1("s = : ",s) ;
50     cout << "s[3] = " << s[3] << endl ;
51     string s1(s) ;
52     print1("s1 = : ",s1) ;
53     if (s1 == s) {
54         cout << "s1 == s" << endl ;
55     }
56     if (s != "abc") {
57         cout << "s != abc" << endl ;
58     }
59     s = s + "babu" ;
60     print1("s = : ",s) ;
61     s = s + 'z' ;
62     print1("s = : ",s) ;
63     s.append("T") ;
64     print1("s = : ",s) ;
65     /* How to append an integer to string */
66     int x = 78 ;
```

```
67 char temp[10] ;
68 sprintf(temp,"%d",x) ;
69 s = s + temp ;
70 print1("s = : ",s) ;
71 /* How to clear a string */
72 s.clear() ;
73 print1("s after clear: ",s) ;
74 }
75
76 /-----
77 testing cin
78
79 Enter your name: jag murthy phd
80 s after getline >> s: jag murthy phd
81 size = 14
82 Enter your Country: san jose ca
83 s after getline >> s: san jose ca
84 size = 11
85 -----
86 static void test_cin() {
87     string s ;
88     cout << "Enter your name: " ;
89     getline(cin,s) ;
90     print1("s after getline >> s: ",s);
91     cout << "Enter your Country: " ;
92     getline(cin,s) ;
93     print1("s after getline >> s: ",s);
94
95 }
96
97 /-----
98 testing iterators
99
100 a = : A quick brown fix jumps over a lazy dog
101 size = 39
102 Understanding forward traversal
103 A   q   u   i   c   k   b   r   o   w   n   f   i   x   j   u   m   p   s   o   v   e   r   a   l   a   z   y   d   o   g
104 Understanding reverse traversal
105 g   o   d   y   z   a   l   a   r   e   v   o   s   p   m   u   j   x   i   f   n   w   o   r   b   k   c   i   u   q   A
106 -----
107 static void test_iterators(){
108     string a("A quick brown fix jumps over a lazy dog") ;
109     print1("a = : ",a) ;
110     {
111         cout << "Understanding forward traversal" << endl ;
112         string::iterator itt = a.begin() ;
113         while (itt != a.end()) {
114             cout << *itt << " " ;
115             itt++ ;
116         }
117         cout << endl ;
118     }
119     {
120         cout << "Understanding reverse traversal" << endl ;
121         string::iterator itt = a.end() ;
122         //cout << *itt << endl ; ----- CORE DUMPS
123         while (itt != a.begin()) {
124             itt-- ; //decrement iterator before using
125             cout << *itt << " " ;
126         }
127         cout << endl ;
128     }
129 }
130
131 /-----
132 testing find
```

```
133
134 str: Hello, can you find UCSC ext? at Santa clara UCSC
135 size = 49
136 First occurrence of UCSC was found at: 20
137 -----
138 static void test_find() {
139     string str("Hello, can you find UCSC ext? at Santa clara UCSC");
140     string::size_type position = str.find("UCSC");
141     print1("str: ",str) ;
142     cout << "First occurrence of UCSC was found at: " << position << endl;
143 }
144
145 /**
146 "A quick brown fix jumps over a lazy dog"
147 token[0] = A
148 token[1] = quick
149 token[2] = brown
150 token[3] = fix
151 token[4] = jumps
152 token[5] = over
153 token[6] = a
154 token[7] = lazy
155 token[8] = dog
156 /**
157 static int tokenize(const string& str, vector<string>& tokens, const string& delimiters = " ")
158 /**
159 "UCSC extension|UCSC|California|United States of America|North America|World", "|"
160
161 token[0] = UCSC extension
162 token[1] = UCSC
163 token[2] = California
164 token[3] = United States of America
165 token[4] = North America
166 token[5] = World
167 /**
168 static void test_tokenizer(const string& str, const string& delimiters = " ")
169
170 /**
171 main
172 /**
173
174 int main() {
175     test_basic() ;
176     test_cin() ;
177     test_iterators() ;
178     test_find() ;
179     test_tokenizer("A quick brown fix jumps over a lazy dog") ;
180     test_tokenizer("UCSC extension|UCSC|California|United States of America|North America|World", "|") ;
181     return EXIT_SUCCESS ;
182 }
183
```

5.4. PROBLEM SET

5.4 Problem set

Problem 5.4.1. Write the procedure **test_tokenizer** and **tokenize** in the code 5.3.

For the string "A quick brown fix jumps over a lazy dog",
your output should look as:

```
token[0] = A  
token[1] = quick  
token[2] = brown  
token[3] = fix  
token[4] = jumps  
token[5] = over  
token[6] = a  
token[7] = lazy  
token[8] = dog
```

For the string
"UCSC extension|UCSC|California|United States of America|North America|World",
your output should look as:

```
token[0] = UCSC extension  
token[1] = UCSC  
token[2] = California  
token[3] = United States of America  
token[4] = North America  
token[5] = World
```

Chapter 6

Dynamic stack

6.1 Introduction

6.1. INTRODUCTION

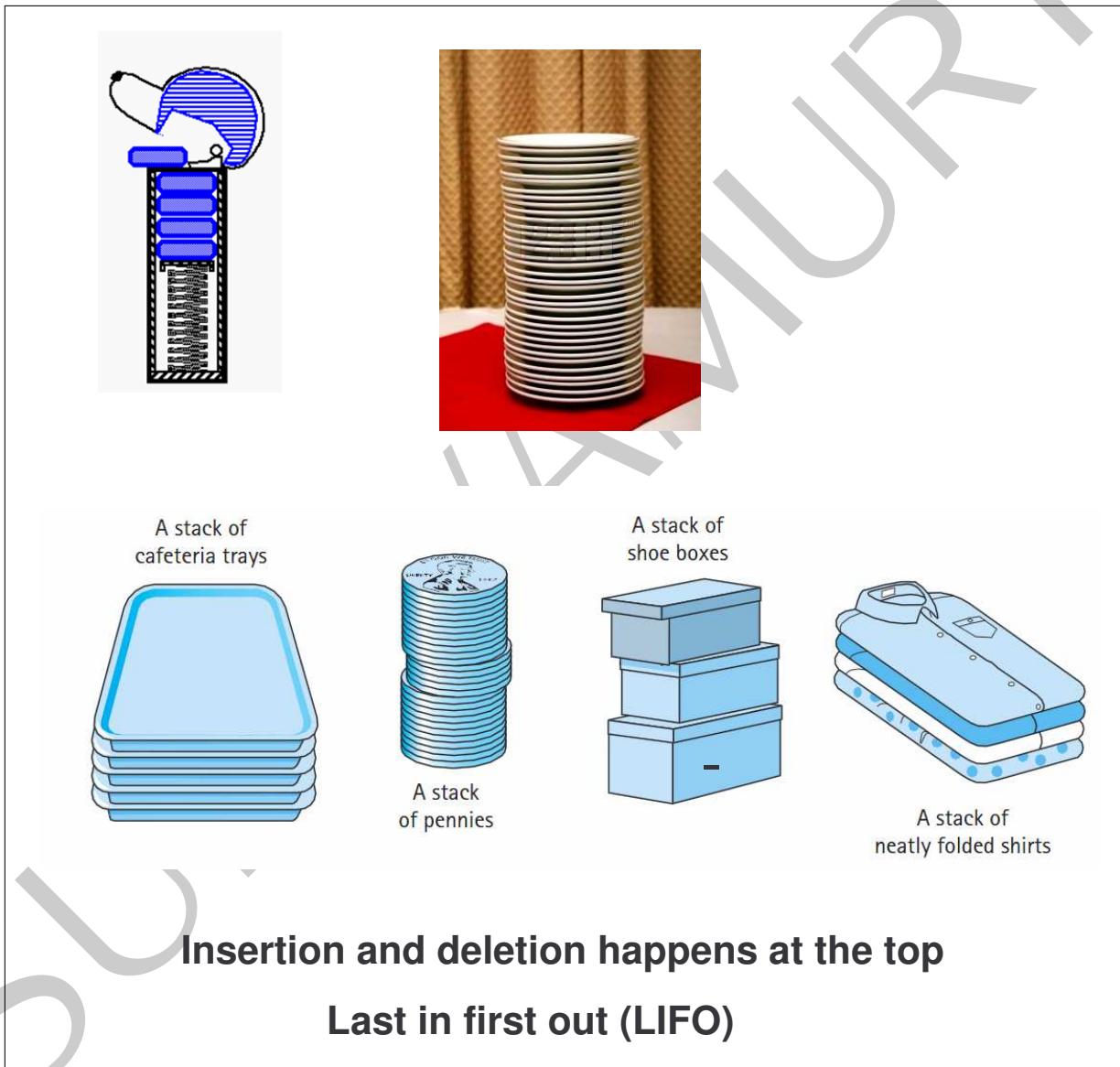


Figure 6.1: Real life stack

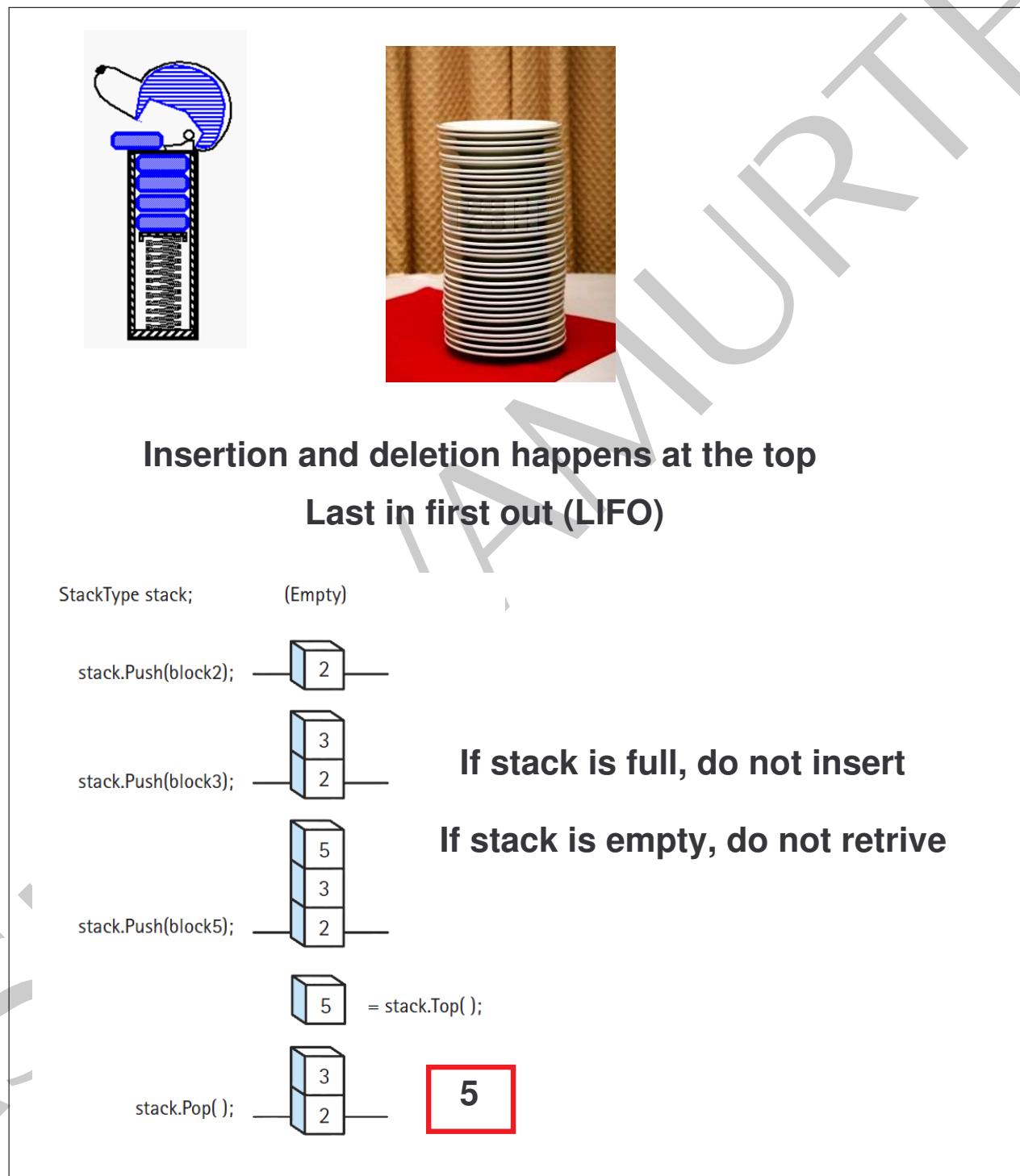


Figure 6.2: Property of stack

6.2 Dynamic stack of objects

```
1 /*-----  
2 Copyright (c) 2015 Author: Jagadeesh Vasudevamurthy  
3 file: dstack.h  
4 -----*/  
5  
6 /*-----  
7 This file has dstack class declaration  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #ifndef dstack_H  
14 #define dstack_H  
15  
16 #include "../util/util.h"  
17 #include "../darray/darray.h"  
18  
19 /*-----  
20 Declaration of dstack class  
21 -----*/  
22 template <typename T>  
23 class dstack {  
24 public:  
25     explicit dstack(int capacity = 50, bool display = false);  
26     explicit dstack(bool);  
27     explicit dstack(bool d, int c) = delete;  
28     ~dstack();  
29     int num_elements() const;  
30     bool isempty() const;  
31     bool isfull() const;  
32     void push(const T& b); // Stack copies b and holds. Now stack is the owner of b  
33     T& top(); // user can get top by alias. He can change its contents also. See explanation in  
            implementation  
34     void pop(); // Remove top element from the stack. Nothing returned  
35  
36     void for_each_element_of_stack_from_top_to_bottom(void(*pf) (T& c));  
37     bool display()const { return _display; }  
38     void set_display(bool x) {  
39         darray<T>::set_display(x);  
40         _display = x;  
41     }  
42     /* no body will copies or equal stack */  
43     dstack(const dstack<T>& s) = delete;  
44     dstack<T>& operator=(const dstack<T>& rhs) = delete;  
45 private:  
46     bool _display;  
47     int _sp;  
48     darray<T> _stack;  
49 };  
50  
51 #include "dstack.hpp"  
52  
53 #endif  
54 //EOF  
55  
56
```

```
1 /*-
2 Copyright (c) 2015 Author: Jagadeesh Vasudevamurthy
3 file: dstack.hpp
4
5 -----*/
6
7 /*
8 This file has class definition
9 -----*/
10
11 /*
12 Definition of routines of dstack class
13 -----*/
14
15 /*
16 Constructor
17 -----*/
18 template <typename T>
19 dstack<T>::dstack(int c, bool d) :_display(d), _sp(0), _stack(c,d){
20     if (display()) {
21         cout << "in dstack constructor:" << endl;
22     }
23 }
24
25 /*
26 Constructor
27 -----*/
28 template <typename T>
29 dstack<T>::dstack(bool d) :_display(d), _sp(0), _stack(50, d){
30     if (display()) {
31         cout << "in dstack constructor:" << endl;
32     }
33 }
34
35 /*
36 Destructor
37 -----*/
38 template <typename T>
39 dstack<T>::~dstack() {
40     if (display()) {
41         cout << "In dstack Destructor " << endl;
42     }
43     _sp = 0;
44     //calls _stack array destructor here from 0 to _sp-1
45 }
46
47 /*
48 Get num elements of the stack
49 -----*/
50 template <typename T>
51 int dstack<T>::num_elements() const {
52     return _sp;
53 }
54
55 /*
56 Is stack empty
57 -----*/
58 template <typename T>
59 bool dstack<T>::isempty() const {
60     return _sp ? false : true;
61 }
62
63 /*
64 Is stack full
65 Our stack can never be full.
66 -----*/
```

```
67 template <typename T>
68 bool dstack<T>::isfull() const {
69     return false;
70 }
71 /*
73 Get the top of the stack.
74
75 Stack is the owner of the object. Note that the object is
76 returned by alias so that NO copy constructor is called.
77
78 The caller has two options:
79
80 T& obj1 = s.top() ;
81 T obj2 = s.top() ;
82
83 In first case, if obj1 is changed, he is really changing the stored obj in s
84 In second case, if obj2 is changed he is changing the copied object.
85 It has no effect on the object that was in the stack. obj2 will die
86 at the end of its scope.
87 -----
88 template <typename T>
89 T& dstack<T>::top(){
90     if (isempty()) {
91         assert(0);
92     }
93     return (_stack[_sp - 1]);
94 }
95
96 /**
97 pop: Remove the top element from the stack.
98 NOTHING is returned.
99 This object still resides in darray which dies
100 when destructor of darray is called. But user
101 has no access to it.
102 -----
103 template <class T>
104 void dstack<T>::pop() {
105     if (isempty()) {
106         assert(0);
107     }
108     --_sp;
109 }
110
111 /**
112 pop: Push the element to the top of the stack
113 My stack is never full. I can always push to my stack
114
115 Note that b is copied into stack. The copied object
116 is the property of stack and not the user.
117 -----
118 template <class T>
119 void dstack<T>::push(const T& b) {
120     _stack[_sp++] = b;
121 }
122
123 /**
124 apply function pf to each element of the stack.
125 -----
126 template <class T>
127 void dstack<T>::for_each_element_of_stack_from_top_to_bottom(void(*pf) (T& c)) {
128     for (int i = _sp - 1; i >= 0; i--) {
129         pf(_stack[i]);
130     }
131 }
```

```
1 /*-
2 Copyright (c) 2015 Author: Jagadeesh Vasudevamurthy
3 file: ../complex/complex.cpp dstacktest.cpp
4
5 On linux:
6 g++ ../complex/complex.cpp dstacktest.cpp
7 valgrind a.out
8 valgrind --leak-check=full -v a.out
9 -- All heap blocks were freed -- no leaks are possible
10
11 -----*/
12
13 /*-
14 This file test dstack object
15 -----*/
16
17 /*
18 All includes here
19 -----*/
20 #include "dstack.h"
21 #include "../complex/complex.h"
22
23 /*-
24 local to this file. Change verbose = true for debugging
25 -----*/
26 static bool verbose = true;
27
28 /*
29 Print an integer - value given
30 -----*/
31 static void print(int& x) {
32     cout << x << " ";
33 }
34
35 /*
36 Print an integer - address given
37 -----*/
38 static void print(int*& x) {
39     print(*x);
40 }
41
42 /*
43 Print a complex - value given
44 -----*/
45 static void print(complex& x) {
46     cout << x << " ";
47 }
48
49 /*
50 Print a complex - address given
51 -----*/
52 static void print(complex*& x) {
53     print(*x);
54 }
55
56 /*
57 add by 2000 - value given
58 -----*/
59 static void add2000(int& x) {
60     x = x + 2000;
61 }
62
63 /*
64 add by 2000 - address given
65 -----*/
66 static void add2000(int*& x) {
```

```
67     add2000(*x);
68 }
69
70 /*-
71 add by 2000 - value given
72 -----*/
73 static void add2000(complex& c) {
74     int x, y;
75     c.getxy(x, y);
76     x = x + 2000;
77     y = y + 2000;
78     c.setxy(x, y);
79 }
80
81 /*-
82 add by 2000 - address given
83 -----*/
84 static void add2000(complex*& x) {
85     add2000(*x);
86 }
87
88 /*-
89 delete integer - address given
90 -----*/
91 static void delete_obj(int*& x) {
92     delete(x);
93 }
94
95 /*-
96 delete complex - address given
97 -----*/
98 static void delete_obj(complex*& x) {
99     delete(x);
100}
101
102
103 /*-
104 array of integers
105 -----*/
106 static void test_stack_of_integers(){
107     {
108         //THIS will not compile
109         //int x = 8;
110         //dstack<int> s(8,verbose);
111     }
112     dstack<int> s(3,verbose);
113     cout << "Number of element in the stack is: " << s.num_elements() << endl;
114     for (int i = 0; i < 8; i++) {
115         s.push(1000 + i);
116     }
117     s.for_each_element_of_stack_from_top_to_bottom(print);
118     cout << endl;
119     s.for_each_element_of_stack_from_top_to_bottom(add2000);
120     s.for_each_element_of_stack_from_top_to_bottom(print);
121     cout << endl;
122     for (int i = 0; i < 7; i++){
123         int& x = s.top();
124         cout << "Top element = " << x << endl;
125         s.pop();
126     }
127     int& x = s.top();
128     cout << "top element = " << x << endl;
129     for (int i = 0; i < 8; i++) {
130         s.push(-(8000 + i));
131     }
132     s.for_each_element_of_stack_from_top_to_bottom(print);
```

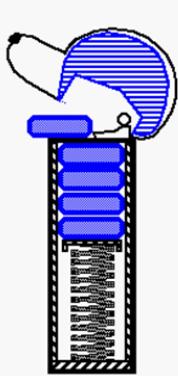
```
133 cout << endl;
134 int& y = s.top();
135 y = 9999;
136 s.for_each_element_of_stack_from_top_to_bottom(print);
137 cout << endl;
138 int z = s.top();
139 z = 8888;
140 s.for_each_element_of_stack_from_top_to_bottom(print);
141 cout << endl;
142 }
143
144 /*-----
145 array of integer pointers
146 -----*/
147 static void test_stack_of_ptr_integers(){
148 dstack<int*> s(3,verbose);
149 cout << "Number of element in the stack is: " << s.num_elements() << endl;
150 for (int i = 0; i < 8; i++) {
151 s.push(new(int)(1000 + i));
152 }
153 s.for_each_element_of_stack_from_top_to_bottom(print);
154 cout << endl;
155 s.for_each_element_of_stack_from_top_to_bottom(add2000);
156 s.for_each_element_of_stack_from_top_to_bottom(print);
157 cout << endl;
158 for (int i = 0; i < 7; i++){
159 int*& x = s.top();
160 cout << "top element = " << *x << endl;
161 delete(x);
162 s.pop();
163 }
164 int x = *(s.top());
165 cout << "Top element = " << x << endl;
166 for (int i = 0; i < 8; i++) {
167 s.push(new(int)(-(8000 + i)));
168 }
169 s.for_each_element_of_stack_from_top_to_bottom(print);
170 cout << endl;
171 s.for_each_element_of_stack_from_top_to_bottom(delete_obj);
172 }
173
174 /*-----
175 array of user defined type
176 -----*/
177 static void test_stack_of_udt(){
178 dstack<complex> s(3, verbose);
179 cout << "Number of element in the stack is: " << s.num_elements() << endl;
180 for (int i = 0; i < 8; i++) {
181 s.push(complex(1000 + i, -(1000 + i)));
182 }
183 s.for_each_element_of_stack_from_top_to_bottom(print);
184 cout << endl;
185 s.for_each_element_of_stack_from_top_to_bottom(add2000);
186 s.for_each_element_of_stack_from_top_to_bottom(print);
187 cout << endl;
188 for (int i = 0; i < 7; i++){
189 cout << "Top element = " << s.top() << endl;
190 s.pop();
191 }
192 cout << "Top element = " << s.top() << endl;
193 for (int i = 0; i < 8; i++) {
194 s.push(complex(-(8000 + i), -(-(8000 + i))));
195 }
196 s.for_each_element_of_stack_from_top_to_bottom(print);
197 cout << endl;
198 complex& y = s.top();
```

```
199 y = 9999;
200 s.for_each_element_of_stack_from_top_to_bottom(print);
201 cout << endl;
202 complex z = s.top();
203 z = 8888;
204 s.for_each_element_of_stack_from_top_to_bottom(print);
205 cout << endl;
206 }
207
208 /*-----
209 array of user defined pointer type
210 -----*/
211 static void test_stack_of_ptr_udt(){
212 dstack<complex*> s(3, verbose);
213 cout << "Number of element in the stack is: " << s.num_elements() << endl;
214 for (int i = 0; i < 8; i++) {
215 s.push(new(complex)(complex(1000 + i, -(1000 + i))));
```

```
216 }
217 s.for_each_element_of_stack_from_top_to_bottom(print);
218 cout << endl;
219 s.for_each_element_of_stack_from_top_to_bottom(add2000);
220 s.for_each_element_of_stack_from_top_to_bottom(print);
221 cout << endl;
222 for (int i = 0; i < 7; i++){
223 complex*& x = s.top();
224 cout << "top element = " << *(x) << endl;
225 delete(x);
226 s.pop();
227 }
228 cout << "top element = " << s.top() << endl;
229 for (int i = 0; i < 8; i++) {
230 s.push(new(complex)((-8000 + i), -(8000 + i))));
```

```
231 }
232 s.for_each_element_of_stack_from_top_to_bottom(print);
233 cout << endl;
234 s.for_each_element_of_stack_from_top_to_bottom(delete_obj);
235 }
236
237 /*-----
238 main
239 -----*/
240 int main() {
241 complex::set_display(verbose);
242 test_stack_of_integers();
243 cout << "-----" << endl;
244 test_stack_of_ptr_integers();
245 cout << "-----" << endl;
246 test_stack_of_udt();
247 cout << "-----" << endl;
248 test_stack_of_ptr_udt();
249 cout << "-----" << endl;
250 return 0;
251 }
252
253
254 //EOF
255
256
```

6.3 Stack applications



Stack Applications

1. Reversing a number
2. Converting decimal to binary
3. Palindrome of a string
 - A) abba
 - B) Madam, I'm adam
 - C) Able was I ere I saw Elba
 - D) So many dynamos
4. Infix to postfix conversions
$$A + B * C - D / E \Rightarrow A B C * + D E / -$$
5. Evaluation of postfix expression
6. Back tracking
7. Recursion

Figure 6.3: Applications

6.4. PROBLEM SET

6.4 Problem set

Problem 6.4.1. Write a program that takes an unsigned integer and converts to binary form.

Problem 6.4.2. Write a program that takes an arbitrary expression and evaluate it. For example:

$$f = a/b^c + d * e - a * c$$

If $a = 4$, $b = 2$, $c = 2$, $d = 3$ and $e = 3$,

$$\begin{aligned} f &= 4/(2^2) + (3 * 3) - (4 * 2) \\ &= (4/4) + 9 - 8 \\ &= 2 \end{aligned}$$

CHAPTER 6. DYNAMIC STACK

Problem 6.4.3. Write a class called recursion that implement the functions given in fig 6.4

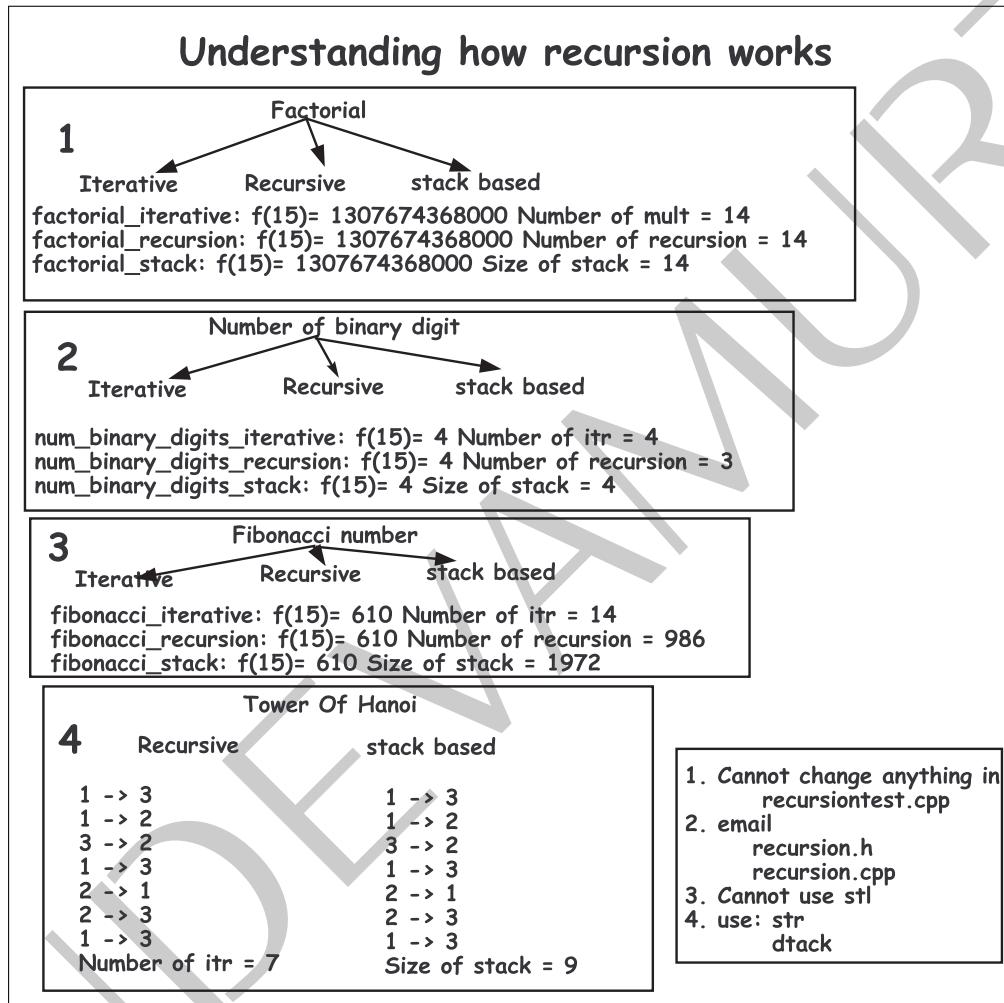


Figure 6.4: Understanding recursion

```
1 /*-----  
2 Copyright (c) 2015 Author: Jagadeesh Vasudevamurthy  
3 file: recursiontest.cpp  
4  
5 On linux:  
6 g++ ..\str\str.cpp factorial.cpp fibonacci.cpp numbinarydigit.cpp hanoi.cpp recursiontest.cpp  
7 valgrind a.out  
8 -----*/  
9  
10 /*-----  
11 This file test recursion object  
12 -----*/  
13  
14 /*-----  
15 All includes here  
16 -----*/  
17 #include "recursion.h"  
18  
19 /*-----  
20 local to this file. Change verbose = true for debugging  
21 -----*/  
22 static bool verbose = false;  
23  
24 /*-----  
25 test factorial  
26 -----*/  
27 void test_factorial() {  
28     recursion f(verbose);  
29     cout << "-----Factorial using iteration-----\n";  
30     for (int i = 0; i < 30; ++i) {  
31         int num_mult = 0;  
32         long long a = f.factorial_iterative(i, num_mult);  
33         cout << "factorial_iterative: f(" << i << ")= " << a << " Number of mult = " << num_mult << endl;  
34     }  
35     cout << "-----Factorial using recursion-----\n";  
36     for (int i = 0; i < 30; ++i) {  
37         int num_rec = 0;  
38         long long a = f.factorial_recursion(i, num_rec);  
39         cout << "factorial_recursion: f(" << i << ")= " << a << " Number of recursion = " << num_rec << endl;  
40     }  
41     cout << "-----Factorial using stack-----\n";  
42     for (int i = 0; i < 30; ++i) {  
43         int size_of_stack = 0;  
44         long long a = f.factorial_stack(i, size_of_stack);  
45         cout << "factorial_stack: f(" << i << ")= " << a << " Size of stack = " << size_of_stack << endl;  
46         int num_mult = 0;  
47         assert(a == f.factorial_iterative(i, num_mult));  
48     }  
49 }  
50  
51 /*-----  
52 test factorial  
53 -----*/  
54 void test_num_binary_digit() {  
55     recursion f(verbose);  
56     cout << "-----num_binary_digits using iteration-----\n";  
57     for (int i = 0; i < 30; ++i) {  
58         int num_itr = 0;  
59         int a = f.num_binary_digit_iterative(i, num_itr);  
60         cout << "num_binary_digits_iterative: f(" << i << ")= " << a << " Number of itr = " << num_itr << endl;  
61     }  
62     cout << "-----num_binary_digits using recursion-----\n";  
63     for (int i = 0; i < 30; ++i) {  
64         int num_rec = 0;
```

```
65     int a = f.num_binary_digit_recursion(i, num_rec);
66     cout << "num_binary_digits_recursion: f(" << i << ")= " << a << " Number of recursion = " <<
67     num_rec << endl;
68 }
69 cout << "-----num_binary_digits_using stack-----\n";
70 for (int i = 0; i < 30; ++i) {
71     int size_of_stack = 0;
72     int a = f.num_binary_digit_stack(i, size_of_stack);
73     cout << "num_binary_digits_stack: f(" << i << ")= " << a << " Size of stack = " << size_of_stack <<
74     endl;
75     int num_itr = 0;
76     assert(a == f.num_binary_digit_iterative(i, num_itr));
77 }
78 cout << "-----num_binary_digits_using stack TEST FOR 300000-----\n";
79 for (int i = 0; i < 300000; ++i) {
80     int size_of_stack = 0;
81     int a = f.num_binary_digit_stack(i, size_of_stack);
82     if (i % 10000 == 0) {
83         cout << "num_binary_digits_stack: f(" << i << ")= " << a << " Size of stack = " << size_of_stack <<
84         endl;
85     }
86     int num_itr = 0;
87     assert(a == f.num_binary_digit_iterative(i, num_itr));
88     assert(a == f.num_binary_digit_recursion(i, num_itr));
89 }
90 /*-----
91 test fibonacci
92 -----
93 void test_fibonacci() {
94     recursion f(verbose);
95     cout << "-----fibonacci using iteration-----\n";
96     for (int i = 0; i < 30; ++i) {
97         int num_itr = 0;
98         int a = f.fibonacci_iterative(i, num_itr);
99         cout << "fibonacci_iterative: f(" << i << ")= " << a << " Number of itr = " << num_itr << endl;
100    }
101    cout << "-----fibonacci using recursion-----\n";
102    for (int i = 0; i < 30; ++i) {
103        int num_rec = 0;
104        int a = f.fibonacci_recursion(i, num_rec);
105        cout << "fibonacci_recursion: f(" << i << ")= " << a << " Number of recursion = " << num_rec <<
106        endl;
107    }
108    cout << "-----fibonacci_using stack-----\n";
109    for (int i = 0; i < 30; ++i) {
110        int size_of_stack = 0;
111        int a = f.fibonacci_stack(i, size_of_stack);
112        cout << "fibonacci_stack: f(" << i << ")= " << a << " Size of stack = " << size_of_stack << endl;
113        int num_itr = 0;
114        assert(a == f.fibonacci_iterative(i, num_itr));
115    }
116    cout << "-----num_binary_digits_using stack TEST FOR 300000-----\n";
117    for (int i = 0; i < 35; ++i) {
118        int size_of_stack = 0;
119        int a = f.fibonacci_stack(i, size_of_stack);
120        cout << "fibonacci_stack: f(" << i << ")= " << a << " Size of stack = " << size_of_stack << endl;
121        int num_itr = 0;
122        assert(a == f.fibonacci_iterative(i, num_itr));
123        assert(a == f.fibonacci_recursion(i, num_itr));
124    }
125    cout << "Note that fibonacci_recursion sucks after 34\n";
126 }
```

```
125 /*-----  
126 test tower of hanoi  
127 -----*/  
128 void test_hanoi() {  
129     recursion f(verbose);  
130     cout << "-----hanoi using recursion-----\n";  
131     for (int i = 1; i < 5; ++i) {  
132         int num_itr = 0;  
133         str s;  
134         f.hanoi_recursion(i, num_itr, s);  
135         cout << s << "Number of itr = " << num_itr << endl;  
136     }  
137     cout << "-----hanoi_using stack-----\n";  
138     for (int i = 1; i < 10; ++i) {  
139         int size_of_stack = 0;  
140         str ss;  
141         f.hanoi_stack(i, size_of_stack, ss);  
142         cout << ss << "Size of stack = " << size_of_stack << endl;  
143         int num_itr = 0;  
144         str s;  
145         f.hanoi_recursion(i, num_itr, s);  
146         assert(ss == s);  
147     }  
148 }  
149  
150 /*-----  
151 main  
152 -----*/  
153 int main() {  
154     test_factorial();  
155     test_num_binary_digit();  
156     test_fibonacci();  
157     test_hanoi();  
158     return 0;  
159 }  
160  
161 //EOF  
162  
163 }
```

CHAPTER 6. DYNAMIC STACK

Problem 6.4.4. GOOGLE interview question: Design a stack. We want to push, pop, and also, retrieve the minimum element in constant time.

6.4. PROBLEM SET

VASUDEVAMURTHY

Chapter 7

Dynamic queue

7.1 Introduction

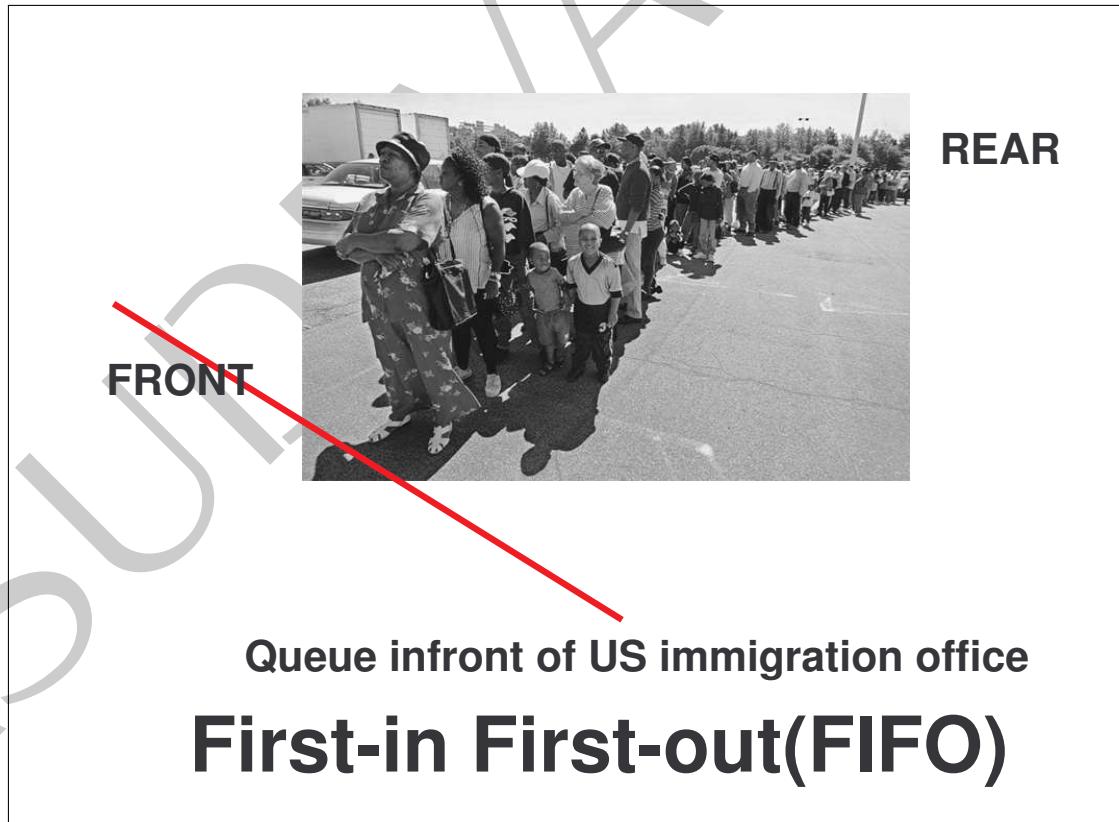


Figure 7.1: Real life queue

7.2 Dynamic queue of objects

```
1 /*-----  
2 Copyright (c) 2015 Author: Jagadeesh Vasudevamurthy  
3 file: dqueue.h  
4 -----*/  
5  
6 /*-----  
7 This file has dqueue class declaration  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #ifndef dqueue_H  
14 #define dqueue_H  
15  
16 #include "../util/util.h"  
17 #include "../darray/darray.h"  
18  
19 /*-----  
20 Declaration of dqueue class  
21 -----*/  
22 template <typename T>  
23 class dqueue {  
24 public:  
25     explicit dqueue(int capacity = 50, bool d = false);  
26     explicit dqueue(bool d);  
27     explicit dqueue(bool d, int c) = delete;  
28  
29     ~dqueue();  
30     int num_elements() const;  
31     bool isempty() const;  
32     bool isfull() const;  
33     void enqueue(const T& b); //add element to queue  
34     //Queue copies b and holds. Now Queue is the owner of b  
35     T& front(); // user can get front by alias  
36     const T& front() const; //for constant queue user can get top by alias. But cannot modify  
37     void dequeue(); // Remove front element from the queue. Nothing returned  
38     void for_each_element_of_queue_from_front_to_rear(void(*pf) (T& c));  
39     bool display() const { return _display; }  
40     void set_display(bool x) {  
41         darray<T>::set_display(x);  
42         _display = x;  
43     }  
44     /* no body will copies or equal stack */  
45     dqueue(const dqueue<T>& s) = delete;  
46     dqueue<T>& operator=(const dqueue<T>& rhs) = delete;  
47 private:  
48     bool _display;  
49     int _front;  
50     int _rear;  
51     darray<T> _queue;  
52 };  
53  
54 #include "dqueue.hpp"  
55  
56 #endif  
57 //EOF  
58  
59
```

```
1 /*-
2 Copyright (c) 2015 Author: Jagadeesh Vasudevamurthy
3 file: dqueue.hpp
4
5 -----
6 */
7 /**
8 This file has class definition
9 -----
10 */
11 /**
12 Definition of routines of dqueue class
13 -----
14 */
15 /**
16 Constructor
17 -----
18 template <typename T>
19 dqueue<T>::dqueue(int c, bool d) :_display(d), _front(0), _rear(0), _queue(c, d){
20     if (display()) {
21         cout << "in dqueue constructor:" << endl;
22     }
23 }
24
25 /**
26 Constructor
27 -----
28 template <typename T>
29 dqueue<T>::dqueue(bool d) :_display(d), _front(0), _rear(0), _queue(50, d){
30     if (display()) {
31         cout << "in dqueue constructor:" << endl;
32     }
33 }
34
35 /**
36 Destructor
37 -----
38 template <typename T>
39 dqueue<T>::~dqueue() {
40     if (display()) {
41         cout << "In dqueue destructor " << endl;
42     }
43     _front = 0;
44     _rear = 0;
45     //calls _queue array destructor here from 0 to _rear-1
46 }
47
48 /**
49 Get num_elements of the queue
50 -----
51 template <typename T>
52 int dqueue<T>::num_elements() const {
53     return _rear - _front;
54 }
55
56 /**
57 Is queue empty
58 -----
59 template <typename T>
60 bool dqueue<T>::isempty() const {
61     return (_front == _rear);
62 }
63
64 /**
65 Is queue full
66 Our queue can never be full.
```

```
67 -----*/
68 template <typename T>
69 bool dqueue<T>::isfull() const {
70     return false;
71 }
72 */
73 /*-
74 Get the front object of the queue for reading by alias.
75 Cannot modify
76 -----*/
77 template <typename T>
78 const T& dqueue<T>::front() const {
79     if (isempty()) {
80         assert(0);
81     }
82     return (_queue[_front]);
83 }
84 */
85 /*-
86 Get the front object of the queue by alias
87
88 Queue is the owner of the object. Note that the object is
89 returned by alias so that NO copy constructor is called.
90
91 The caller has two options:
92
93 T& obj1 = s.front();
94 T obj2 = s.front();
95
96 In first case, if obj1 is changed, he is really changing the stored obj in s
97 In second case, if obj2 is changed he is changing the copied object.
98 It has no effect on the object that was in the queue. obj2 will die
99 at the end of its scope
100 -----*/
101 template <typename T>
102 T& dqueue<T>::front(){
103     if (isempty()) {
104         assert(0);
105     }
106     return (_queue[_front]);
107 }
108 */
109 /*-
110 dequeue: Remove an element from the front of the queue.
111 NOTHING is returned.
112 This object still resides in darray which dies
113 when destructor of dqueue is called. But user
114 has no access to it.
115 -----*/
116 template <class T>
117 void dqueue<T>::dequeue() {
118     if (isempty()) {
119         assert(0);
120     }
121     _front++;
122 }
123 */
124 /*-
125 enqueue: Insert element to the queue at the rear.
126 My queue is never full. I can always enter elements to my queue
127
128 Note that b is copied into queue. The copied object
129 is the property of queue and not the user.
130 -----*/
131 template <class T>
132 void dqueue<T>::enqueue(const T& b) {
```

```
133     _queue[_rear++] = b;
134 }
135
136 /*-----
137 apply function pf to each element of the queue.
138 -----*/
139 template <class T>
140 void dqueue<T>::for_each_element_of_queue_from_front_to_rear(void(*pf) (T& c)) {
141     for (int i = _front; i < _rear; i++) {
142         pf(_queue[i]);
143     }
144 }
145
146 //EOF
147
148
149
```

```
1 /*-
2 Copyright (c) 2015 Author: Jagadeesh Vasudevamurthy
3 file: ../complex/complex.cpp dqueuetest.cpp
4
5 On linux:
6 g++ ../complex/complex.cpp dqueuetest.cpp
7 valgrind a.out
8 valgrind --leak-check=full -v a.out
9 -- All heap blocks were freed -- no leaks are possible
10
11 -----*/
12
13 /*-
14 This file test dqueue object
15 -----*/
16
17 /*
18 All includes here
19 -----*/
20 #include "dqueue.h"
21 #include "../complex/complex.h"
22
23 /*-
24 local to this file. Change verbose = true for debugging
25 -----*/
26 static bool verbose = true;
27
28 /*
29 Print an integer - value given
30 -----*/
31 static void print(int& x) {
32     cout << x << " ";
33 }
34
35 /*
36 Print an integer - address given
37 -----*/
38 static void print(int*& x) {
39     print(*x);
40 }
41
42 /*
43 Print a complex - value given
44 -----*/
45 static void print(complex& x) {
46     cout << x << " ";
47 }
48
49 /*
50 Print a complex - address given
51 -----*/
52 static void print(complex*& x) {
53     print(*x);
54 }
55
56 /*
57 add by 2000 - value given
58 -----*/
59 static void add2000(int& x) {
60     x = x + 2000;
61 }
62
63 /*
64 add by 2000 - address given
65 -----*/
66 static void add2000(int*& x) {
```

```
67     add2000(*x);
68 }
69
70 /*-
71 add by 2000 - value given
72 -----*/
73 static void add2000(complex& c) {
74     int x, y;
75     c.getxy(x, y);
76     x = x + 2000;
77     y = y + 2000;
78     c.setxy(x, y);
79 }
80
81 /*-
82 add by 2000 - address given
83 -----*/
84 static void add2000(complex*& x) {
85     add2000(*x);
86 }
87
88 /*-
89 delete integer - address given
90 -----*/
91 static void delete_obj(int*& x) {
92     delete(x);
93 }
94
95 /*-
96 delete complex - address given
97 -----*/
98 static void delete_obj(complex*& x) {
99     delete(x);
100}
101
102
103 /*-
104 array of integers
105 -----*/
106 static void test_queue_of_integers(){
107     dqueue<int> s(3, verbose);
108     cout << "Number of element in the queue is: " << s.num_elements() << endl;
109     for (int i = 0; i < 8; i++) {
110         s.enqueue(1000 + i);
111     }
112     s.for_each_element_of_queue_from_front_to_rear(print);
113     cout << endl;
114     s.for_each_element_of_queue_from_front_to_rear(add2000);
115     s.for_each_element_of_queue_from_front_to_rear(print);
116     cout << endl;
117     for (int i = 0; i < 7; i++){
118         int& x = s.front();
119         cout << "Front element = " << x << endl;
120         s.dequeue();
121     }
122     int& x = s.front();
123     cout << "Front element = " << x << endl;
124     for (int i = 0; i < 8; i++) {
125         s.enqueue(-(8000 + i));
126     }
127     s.for_each_element_of_queue_from_front_to_rear(print);
128     cout << endl;
129
130     int& y = s.front();
131     y = 9999;
132     s.for_each_element_of_queue_from_front_to_rear(print);
```

```
133 cout << endl;
134
135 int z = s.front();
136 z = 8888;
137 s.for_each_element_of_queue_from_front_to_rear(print);
138 cout << endl;
139 }
140
141 /*-----
142 array of integer pointers
143 -----*/
144 static void test_queue_of_ptr_integers(){
145 dqueue<int*> s(3, verbose);
146 cout << "Number of element in the queue is: " << s.num_elements() << endl;
147 for (int i = 0; i < 8; i++) {
148 s.enqueue(new int(1000 + i));
149 }
150 s.for_each_element_of_queue_from_front_to_rear(print);
151 cout << endl;
152 s.for_each_element_of_queue_from_front_to_rear(add2000);
153 s.for_each_element_of_queue_from_front_to_rear(print);
154 cout << endl;
155 for (int i = 0; i < 7; i++){
156 int*& x = s.front();
157 cout << "Front element = " << *x << endl;
158 delete(x);
159 s.dequeue();
160 }
161 int x = *(s.front());
162 cout << "Front element = " << x << endl;
163 for (int i = 0; i < 8; i++) {
164 s.enqueue(new int(-(8000 + i)));
165 }
166 s.for_each_element_of_queue_from_front_to_rear(print);
167 cout << endl;
168 s.for_each_element_of_queue_from_front_to_rear(delete_obj);
169 }
170
171 /*-----
172 array of user defined type
173 -----*/
174 static void test_queue_of_udt(){
175 dqueue<complex> s(3, verbose);
176 cout << "Number of element in the queue is: " << s.num_elements() << endl;
177 for (int i = 0; i < 8; i++) {
178 s.enqueue(complex(1000 + i, -(1000 + i)));
179 }
180 s.for_each_element_of_queue_from_front_to_rear(print);
181 cout << endl;
182 s.for_each_element_of_queue_from_front_to_rear(add2000);
183 s.for_each_element_of_queue_from_front_to_rear(print);
184 cout << endl;
185 for (int i = 0; i < 7; i++){
186 cout << "Front element = " << s.front() << endl;
187 s.dequeue();
188 }
189 cout << "Front element = " << s.front() << endl;
190 for (int i = 0; i < 8; i++) {
191 s.enqueue(complex(-(8000 + i), -(-(8000 + i))));
192 }
193 s.for_each_element_of_queue_from_front_to_rear(print);
194 cout << endl;
195
196 complex& y = s.front();
197 y = 9999;
198 s.for_each_element_of_queue_from_front_to_rear(print);  
263
```

```
199 cout << endl;
200
201 complex z = s.front();
202 z = 8888;
203 s.for_each_element_of_queue_from_front_to_rear(print);
204 cout << endl;
205 }
206
207 /*-----
208 array of user defined pointer type
209 -----*/
210 static void test_queue_of_ptr_udt(){
211 dqueue<complex*> s(3,verbose);
212 cout << "Number of element in the queue is: " << s.num_elements() << endl;
213 for (int i = 0; i < 8; i++) {
214 s.enqueue(new complex(1000 + i, -(1000 + i)));
215 }
216 s.for_each_element_of_queue_from_front_to_rear(print);
217 cout << endl;
218 s.for_each_element_of_queue_from_front_to_rear(add2000);
219 s.for_each_element_of_queue_from_front_to_rear(print);
220 cout << endl;
221 for (int i = 0; i < 7; i++){
222 complex*& x = s.front();
223 cout << "Front element = " << *(x) << endl;
224 delete(x);
225 s.dequeue();
226 }
227 cout << "Front element = " << s.front() << endl;
228 for (int i = 0; i < 8; i++) {
229 s.enqueue(new complex(-(8000 + i), -(-(8000 + i))));
230 }
231 s.for_each_element_of_queue_from_front_to_rear(print);
232 cout << endl;
233 s.for_each_element_of_queue_from_front_to_rear(delete_obj);
234 }
235
236 /*-----
237 main
238 -----*/
239 int main() {
240 complex::set_display(verbose);
241 test_queue_of_integers();
242 cout << "_____" << endl;
243 test_queue_of_ptr_integers();
244 cout << "_____" << endl;
245 test_queue_of_udt();
246 cout << "_____" << endl;
247 test_queue_of_ptr_udt();
248 cout << "_____" << endl;
249 return 0;
250 }
251
252
253 //EOF
254
255
```

7.3 Queue applications

7.4 Problem set

7.4. PROBLEM SET

VASUDEVAMURTHY

Chapter 8

STL Deque Class

8.1 Introduction

8.2 Deque class public functions

8.2. DEQUE CLASS PUBLIC FUNCTIONS

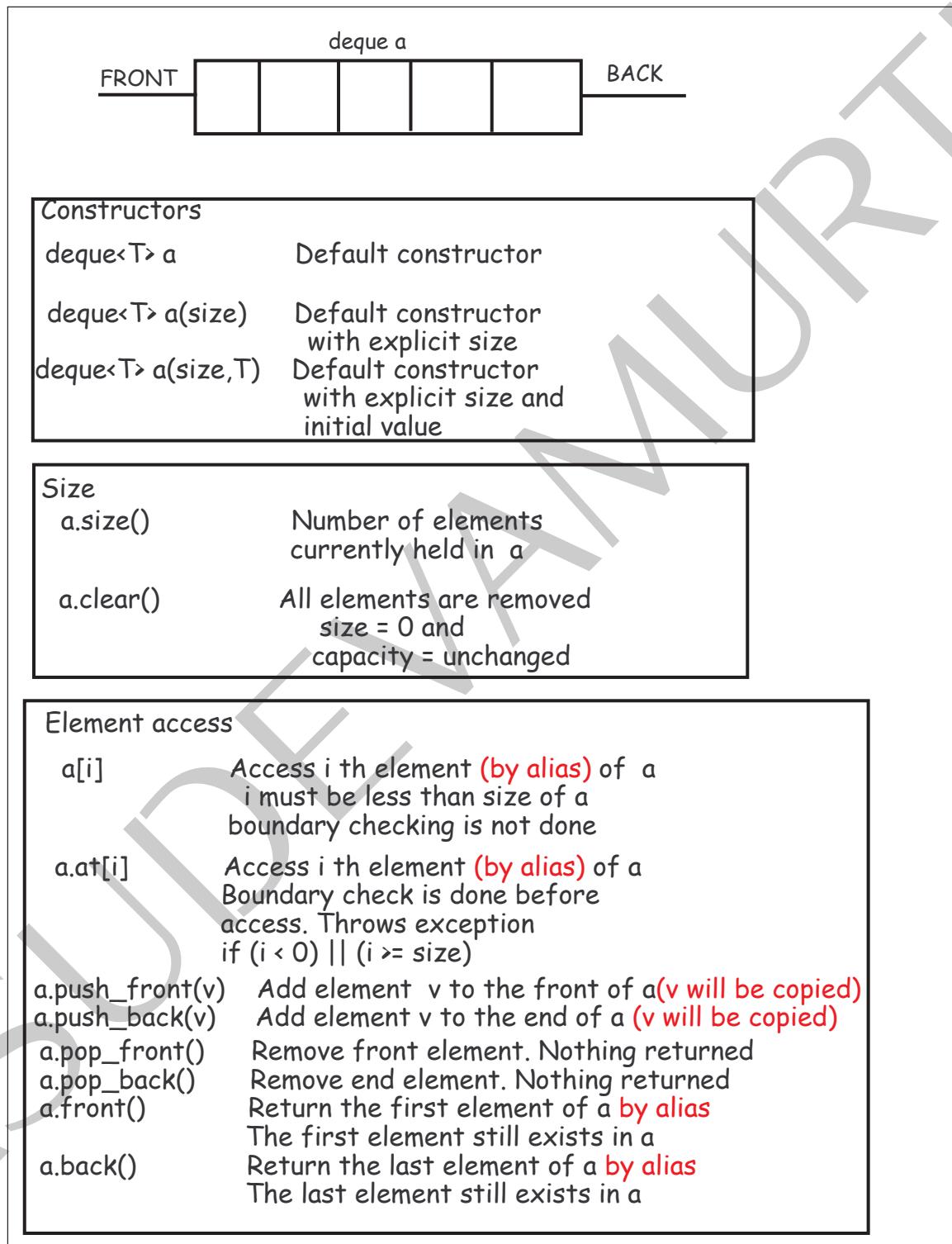
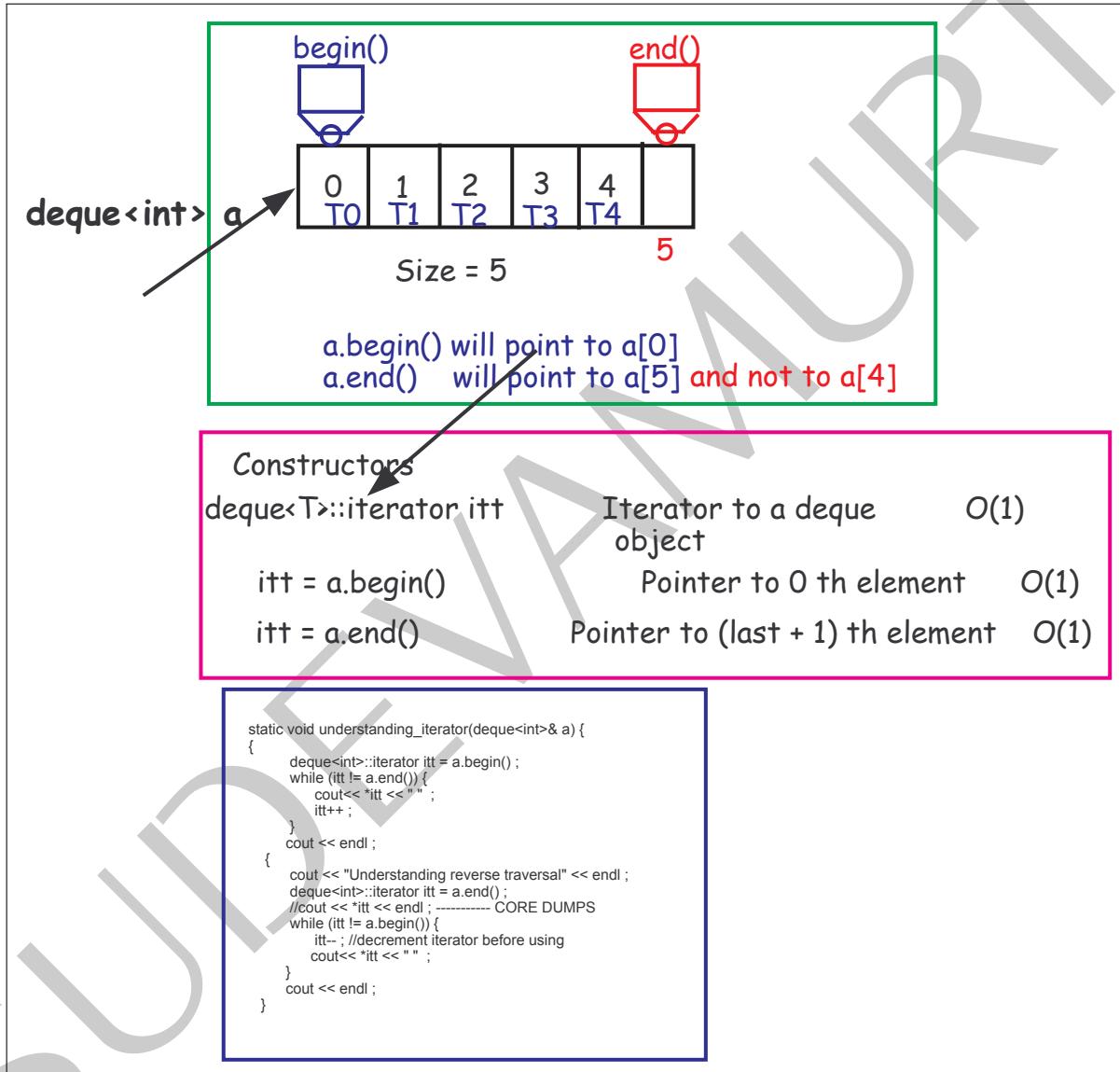


Figure 8.1: `deque` class functions

8.3 Deque class iterator

Figure 8.2: `deque` iterator functions

8.4 Understanding deque class and iterators

```
1 /*-----  
2 Copyright (c) 2010 Author: Jagadeesh Vasudevarmurthy  
3 Filename: sdeque.cpp  
4 compile: g++ sdeque.cpp  
5 Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)  
6 -----*/  
7 #include <iostream>  
8 #include <deque>  
9 #include <stdexcept> //Without this catch will NOT work on Linux  
10  
11  
12 #ifdef _WIN32  
13 #include "c:/jag/alg/course/code/c/complex.h"  
14 #else  
15 #include "/home/jag/jag/alg/c/complex.h"  
16 #endif  
17  
18 using namespace std;  
19  
20 /*-----  
21 static definition - only once at the start  
22 Change to false, if you don't need verbose  
23 -----*/  
24 bool complex::_display = true ;  
25  
26 /*-----  
27 Multiply integer by 10  
28 -----*/  
29 static void multiply_by_10(int& x){  
30     x = x * 10 ;  
31 }  
32  
33 /*-----  
34 print1  
35 -----*/  
36 template <typename T>  
37 static void print1(const char* s, const deque<T>& a){  
38     cout << s << endl ;  
39     cout << "-----" << endl ;  
40     cout << "size = " << a.size() << " " ;  
41     cout << endl ;  
42     for (int i = 0; i < int(a.size()); i++) {  
43         cout << "a[" << i << "] = " << a[i] << " " ;  
44     }  
45     cout << endl ;  
46     cout << "-----" << endl ;  
47 }  
48  
49 /*-----  
50 apply a function pf on elements of deque a  
51 -----*/  
52 template <typename T>  
53 static void apply(const char* s, deque<T>& a, void(*pf)(T& x) ){  
54     cout << s << endl ;  
55     cout << "-----" << endl ;
```

```
56 typename deque<T>::iterator itt = a.begin(); //Note typename here.
57 //WILL NOT COMPILE IN LINUX without typename but works on Visual studio
58 //Hard to find this solution.
59 while (itt != a.end()){
60     T& p = *itt;
61     pf(p);
62     itt++;
63 }
64 cout << endl ;
65 }
66
67 /*
68 a[0] .... a[9]
69
70 begin() will point to a[0]
71 end() will NOT POINT to a[9], but to one element past a[9]
72 That means real end is: end()-1;
73 */
74 static void understanding_iterator(deque<int>& a){
75 {
76     cout << "Understanding forward traversal" << endl ;
77     deque<int>::iterator itt = a.begin();
78     while (itt != a.end()) {
79         cout << *itt << " ";
80         itt++;
81     }
82     cout << endl ;
83 }
84 {
85     cout << "Understanding reverse traversal" << endl ;
86     deque<int>::iterator itt = a.end();
87     //cout << *itt << endl ; ----- CORE DUMPS
88     while (itt != a.begin()) {
89         itt-- ; //decrement iterator before using
90         cout << *itt << " ";
91     }
92     cout << endl ;
93 }
94 cout << "-----" << endl ;
95 }
96
97 /*
98 Understanding access using
99 1. a[i]
100 2. a.at(i) -- Same as a[i], but does out of bound checks
101 and throws exception that you can catch
102
103 */
104 static void understanding_at_access(const deque<int>& a, int i) {
105     //int p = a[i]; //core dumps
106     try {
107         int p = a.at(i); //Throws exception. out of bound checks
108
109     }catch(std::out_of_range) {
110         cout << " accessing a[" << i << "] but size of array is " << a.size() << endl ;
```

```
111 }
112 cout << "-----" << endl ;
113 }
114
115 /*-
116 a.front() -- Returns first element of the deque
117 a.back() -- Returns last element of the deque
118 a.push_front(v) -- v is inserted as the first element of the deque
119 a.push_back(v) -- v is inserted as the back element of the deque
120 a.pop_front() - First element of the deque is removed. Nothing is returned
121 a.pop_back() - Last element of the deque is removed. Nothing is returned
122 -----*/
123 static void understanding_access(deque<int>& a) {
124     print1("begin with",a);
125     for (int i = 0; i < 5; i++){
126         a.push_back(i);
127     }
128     print1("After Inserting 5 elements from the back",a);
129     for (int i = 0; i < 5; i++){
130         a.push_front(i*10+1);
131     }
132     print1("After Inserting 5 elements from the front",a);
133
134     int& y = a.front();
135     cout << "The front of the deque has " << y << endl ;
136     int& z = a.back();
137     cout << "The back of the deque has " << z << endl ;
138
139     print1("After front and back operation",a);
140
141 //y = a.pop_front(); WRONG. pop cannot return ;
142 a.pop_front();
143 a.pop_front();
144 print1("After Removing two elements from the front",a);
145 a.pop_back();
146 a.pop_back();
147 print1("After Removing two elements from the back",a);
148
149 int x = a.size() - 1;
150 y = a[x];
151 cout << "Random access of a[" << x << "] = " << y << endl ;
152 understanding_at_access(a,a.size());
153 }
154
155 /*-
156 basic
157 -----*/
158 static void basic() {
159     deque<int> a;
160     print1("begin with",a);
161     understanding_access(a);
162     understanding_iterator(a);
163
164     apply("multiply by 10",a,multiply_by_10);
165     cout << endl ;
```

```
166 apply("print using iterator",a,print_integer);
167 cout << endl;
168
169 a.clear();
170 print1("Using clear",a);
171 }
172
173 /*-----
174 array of integer pointers
175 -----*/
176 static void test_deque_of_ptr_integers(){
177 deque<int*> a;
178 print1("begin with",a);
179 for (int i = 0; i < 5; i++){
180 a.push_back(ALLOC(int)(i));
181 }
182 apply("After Inserting 5 elements from the back",a,print_integer);
183 for (int i = 0; i < 5; i++){
184 a.push_front(ALLOC(int)(i*10+1));
185 }
186 apply("After Inserting 5 elements from the front",a,print_integer);
187
188 int*& y = a.front();
189 cout << "The front of the deque has " << *y << endl;
190 int*& ba = a.back();
191 cout << "The back of the deque has " << *ba << endl;
192
193 //y = a.pop_front(); WRONG. pop cannot return;
194 FREE(a.front());
195 a.pop_front();
196
197 FREE(a.front());
198 a.pop_front();
199
200 apply("After Removing two elements from the front",a,print_integer);
201
202 FREE(a.back());
203 a.pop_back();
204
205 FREE(a.back());
206 a.pop_back();
207 apply("After Removing two elements from the back",a,print_integer);
208
209 deque<int*> b(a);
210 apply("Contents of deque b:",b,print_integer);
211 for (int i = 0; i < int(a.size()); i++) {
212 /* we allocated contents in a. So we delete it */
213 FREE(a[i]);
214 }
215 /* Why you should NOT do this */
216 /*
217 for (int i = 0; i < n; i++) {
218 FREE(b[i]);
219 }
220 */
```

```
221 }
222
223 /*-
224 array of user defined type
225 -----
226 static void test_deque_of_udt(){
227     deque<complex> a;
228     print1("begin with",a);
229     for (int i = 0; i < 5; i++){
230         a.push_back(complex(i,-i));
231     }
232     print1("After Inserting 5 elements from the back",a);
233     for (int i = 0; i < 5; i++){
234         a.push_front(complex((i*10+1),-(i*10+1)));
235     }
236     print1("After Inserting 5 elements from the front",a);
237
238     complex& y = a.front();
239     cout << "The front of the deque has " << y << endl;
240     complex& z = a.back();
241     cout << "The back of the deque has " << z << endl;
242     {
243         /* see what happens if U get complex by value */
244         complex vf = a.front();
245     }
246
247 //y = a.pop_front(); WRONG. pop cannot return;
248 a.pop_front();
249 a.pop_front();
250 print1("After Removing two elements from the front",a);
251 a.pop_back();
252 a.pop_back();
253 print1("After Removing two elements from the back",a);
254
255 int x = a.size() - 1;
256 complex &zz = a[x]; //Copy constructor called here
257 cout << "Random access of a[" << x << "] = " << zz << endl;
258
259 deque<complex> b(a);
260 //apply("Contents of deque b",b.print_complex);
261 bool equal = true;
262 for (int i = 0; i < int(a.size()); i++) {
263     if (a[i] != b[i]) {
264         equal = false;
265         break;
266     }
267 }
268 equal ? cout << "array a == b\n" : cout << "array a != b\n";
269 }
270
271 /*-
272 array of user defined pointer type
273 -----
274 static void test_deque_of_ptr_udt(){
275     deque<complex*> a;
```

```
276 print1("begin with",a);
277 for (int i = 0; i < 5; i++){
278     a.push_back(ALLOC(complex)(i,-i));
279 }
280 apply("After Inserting 5 elements from the back",a,print_complex);
281 for (int i = 0; i < 5; i++){
282     a.push_front(ALLOC(complex)(i*10+1,i*10+1));
283 }
284 apply("After Inserting 5 elements from the front",a,print_complex);
285
286 complex*& af = a.front();
287 cout << "The front of the deque has " << *af << endl;
288 complex*& ab = a.back();
289 cout << "The back of the deque has " << *ab << endl;
290
291 //y = a.pop_front(); WRONG. pop cannot return;
292 FREE(a.front());
293 a.pop_front();
294
295 FREE(a.front());
296 a.pop_front();
297
298 apply("After Removing two elements from the front",a,print_complex);
299
300 FREE(a.back());
301 a.pop_back();
302
303 FREE(a.back());
304 a.pop_back();
305
306 apply("After Removing two elements from the back",a,print_complex);
307
308 deque<complex*> b(a);
309 apply("Contents of deque b:",b,print_complex);
310 for (int i = 0; i < int(a.size()); i++) {
311     /* we allocated contents in a. So we delete it */
312     FREE(a[i]);
313 }
314 /* Why you should NOT do this */
315 /*
316 for (int i = 0; i < n; i++) {
317     FREE(b[i]);
318 }
319 */
320 }
321 /*
322 -----
323 main
324 -----
325 int main() {
326     basic();
327     test_deque_of_ptr_integers();
328     test_deque_of_udt();
329     test_deque_of_ptr_udt();
330     return 1;
```

8.5. CONTAINER ADAPTORS

8.5 Container Adaptors

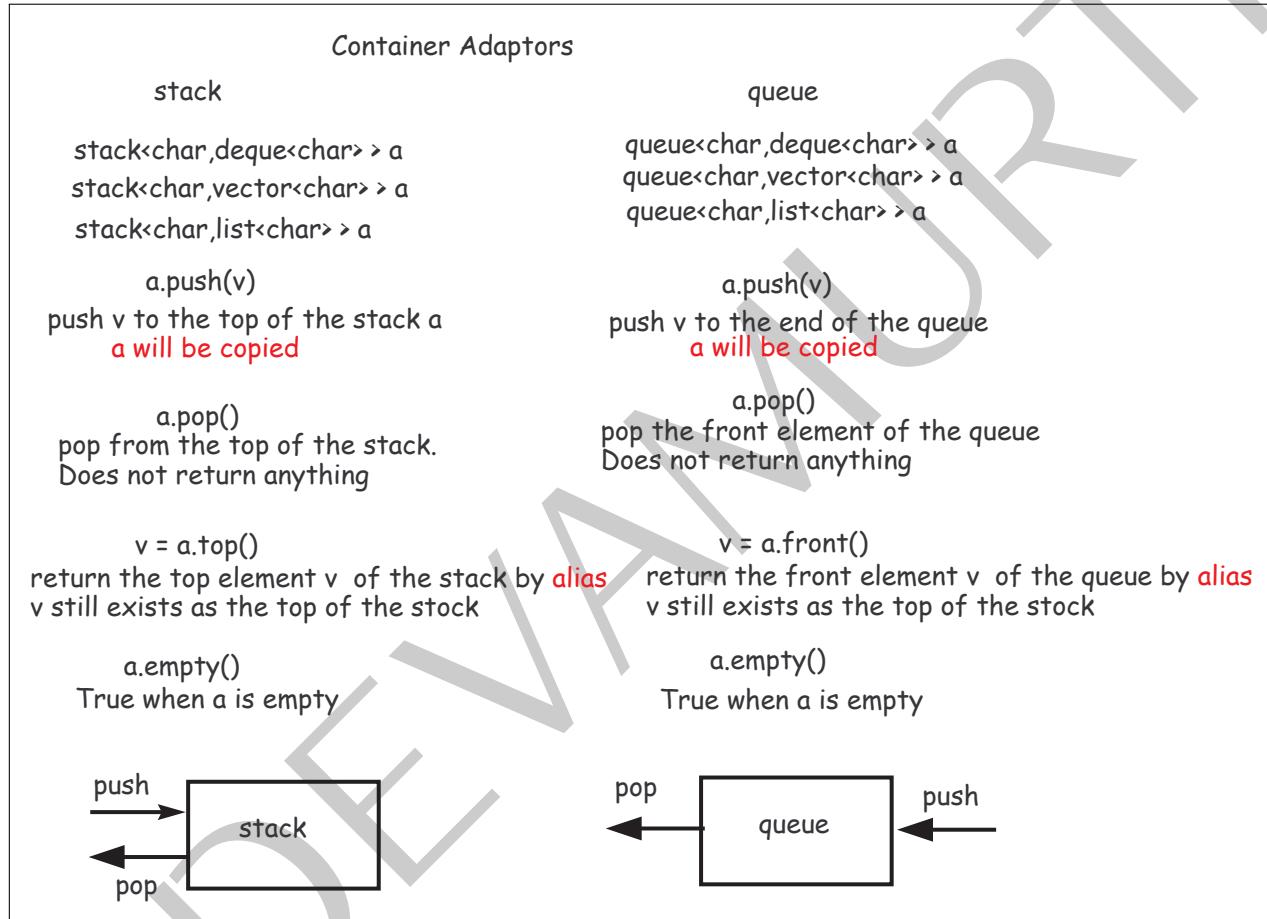


Figure 8.3: **stack** and **queue** as container adaptors

8.5.1 Stack Container Adaptor

```
1 /*-----
2 Copyright (c) 2010 Author: Jagadeesh Vasudevarmurthy
3 Filename: sstack.cpp
4 compile: g++ sstack.cpp
5 Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)
6 -----
7
8 #include <iostream>
9 #include <stack>
10 #include <deque>
11 #include <vector>
12 #include <list>
13
14 #ifdef _WIN32
15 #include "c:/jag/alg/course/code/c/complex.h"
16 #else
17 #include "/home/jag/jag/alg/c/complex.h"
18 #endif
19
20 using namespace std;
21
22
23 /*-
24 static definition - only once at the start
25 Change to false, if you don't need verbose
26 -----
27 bool complex::_display = true ;
28
29 /*-
30 basic
31
32 Note that the stack is a container adaptor that
33 can be built from
34 1. deque
35 2. vector
36 3. list
37 ex: stack<char,vector<char>> a
38 Note the space in vector<char>> a
39 You cannot do: vector<char>> a as compiler confuses with >>
40 -----
41 static void test_stack_of_char() {
42 //Use deque as a container. Note >space>
43 //stack<char,deque<char>> a ;
44
45 //Use vector as a container. Note >space>
46 //stack<char,vector<char>> a ;
47
48 //Use list as a container. Note >space>
49 stack<char,list<char>> a ;
50
51
52 cout << "The following elements are added to the stack\n" ;
53 for (int i = 0; i < 5; i++) {
54 char ch = char(i + 'a');
55 cout << ch << " ";
```

```
56     a.push(ch);
57 }
58 cout << endl;
59 cout << "Number of element in the stack = " << a.size() << endl;
60 cout << "Take out the elements from the stack\n";
61 while (a.empty() == false){
62     cout << a.top() << " ";
63     a.pop();
64 }
65 cout << endl;
66 }
67
68 /*-----
69 stack of complex objects
70 -----*/
71 static void test_stack_of_udt(){
72 //Use deque as a container. Note >space>
73 stack<complex,deque<complex>> a;
74
75 cout << "The following elements are added to the stack\n";
76 for (int i = 0; i < 5; i++) {
77     complex ch = complex(i,-i);
78     cout << ch << " ";
79     a.push(ch);
80 }
81 cout << endl;
82 cout << "Number of element in the stack = " << a.size() << endl;
83 cout << "Take out the elements from the stack\n";
84 while (a.empty() == false){
85     complex& c = a.top();
86     cout << c << " ";
87     a.pop();
88 }
89 cout << endl;
90 }
91
92 /*-----
93 stack of pointer to complex objects
94 -----*/
95 static void test_stack_of_pointer_to_udt(){
96 //Use vector as a container. Note >space>
97 stack<complex*,vector<complex*>> a;
98
99 cout << "The following elements are added to the stack\n";
100 for (int i = 0; i < 5; i++) {
101     complex* ch = ALLOC(complex)(i,-i);
102     cout << *ch << " ";
103     cout << endl;
104     a.push(ch);
105 }
106 cout << endl;
107 cout << "Number of element in the stack = " << a.size() << endl;
108 cout << "Take out the elements from the stack\n";
109 while (a.empty() == false){
110     complex*& u = a.top();
```

8.5.2 Queue Container Adaptor

```
1 /*-----  
2 Copyright (c) 2010 Author: Jagadeesh Vasudevarthy  
3 Filename: squeue.cpp  
4 compile: g++ squeue.cpp  
5 Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)  
6 -----*/  
7  
8 #include <iostream>  
9 #include <queue>  
10 #include <deque>  
11 #include <vector>  
12 #include <list>  
13  
14 #ifdef _WIN32  
15 #include "c:/jag/alg/course/code/c/complex.h"  
16 #else  
17 #include "/home/jag/jag/alg/c/complex.h"  
18 #endif  
19  
20 using namespace std;  
21  
22 /*-----  
23 static definition - only once at the start  
24 Change to false, if you don't need verbose  
25 -----*/  
26 bool complex::_display = true;  
27  
28 /*-----  
29 basic  
30  
31 Note that the queue is a container adaptor that  
32 can be built from  
33 1. deque  
34 2. vector  
35 3. list  
36 ex: queue<char,vector<char>> a  
37 Note the space in vector<char>> a  
38 You cannot do: vector<char>> a as compiler confuses with >>  
39 -----*/  
40 static void test_queue_of_char(){  
41 //Use deque as a container. Note >space>  
42 //queue<char,deque<char>> a;  
43  
44 //Use vector as a container. Note >space>  
45 //queue<char,vector<char>> a;  
46  
47 //Use list as a container. Note >space>  
48 queue<char,list<char>> a;  
49  
50  
51 cout << "The following elements are added to the queue\n";  
52 for (int i = 0; i < 5; i++){  
53 char ch = char(i + 'a');  
54 cout << ch << " ";  
55 a.push(ch);
```

```
56 }
57 cout << endl ;
58 cout << "Number of element in the queue = " << a.size() << endl ;
59 cout << "Take out the elements from the queue\n" ;
60 while (a.empty() == false) {
61     cout << a.front() << " " ;
62     a.pop() ;
63 }
64 cout << endl ;
65 }
66
67 /*-
68 queue of complex objects
69 -----
70 static void test_queue_of_udt() {
71     //Use deque as a container. Note >space>
72     queue<complex,deque<complex>> a ;
73
74     cout << "The following elements are added to the queue\n" ;
75     for (int i = 0; i < 5; i++) {
76         complex ch = complex(i,-i) ;
77         cout << ch << " " ;
78         a.push(ch) ;
79     }
80     cout << endl ;
81     cout << "Number of element in the queue = " << a.size() << endl ;
82     cout << "Take out the elements from the queue\n" ;
83     while (a.empty() == false) {
84         complex& c = a.front() ;
85         cout << c << " " ;
86         a.pop() ;
87     }
88     cout << endl ;
89 }
90
91 /*-
92 queue of pointer to complex objects
93 -----
94 static void test_queue_of_pointer_to_udt() {
95     //Use vector as a container. Note >space>
96     //queue<complex*,vector<complex*>> a ; //Compiles fails at a.pop
97     queue<complex*,deque<complex*>> a ;
98
99     cout << "The following elements are added to the queue\n" ;
100    for (int i = 0; i < 5; i++) {
101        complex* ch = ALLOC(complex)(i,-i) ;
102        cout << *ch << " " ;
103        cout << endl ;
104        a.push(ch) ;
105    }
106    cout << endl ;
107    cout << "Number of element in the queue = " << a.size() << endl ;
108    cout << "Take out the elements from the queue\n" ;
109    while (a.empty() == false) {
110        complex*& c = a.front() ;
```

8.6. PROBLEM SET

8.6 Problem set

Problem 8.6.1. Implement the **deque** class using **darray** class. Call this class as **ddeque**, which stands for **dynamic deque**. The main program is available to you in the file **ddequetest.cpp**. You cannot modify the code in file **ddequetest.cpp**. You need to write the code in the file **ddeque.h** and **ddeque.hpp**. Attach the output of the program, as a comment, at the end of the file **ddeque.hpp** and e-mail only **ddeque.h** and **ddeque.hpp**

```
//Filename: ddeque.h

template <typename T>
class ddeque {
public:
    /* WRITE ALL PUBLIC FUNCTION HERE */
    /* CANNOT HAVE ANY PUBLIC DATA HERE */

    /* for iterator */
    iterator begin() { }
    iterator end() { }
    bool display() const {return _display ; }

private:
    /* MUST USE ONLY darray<T>. CANNOT USE ANY OTHER CONTAINER OBJECTS FROM
       MY CLASS OR STL */
    /* Can have some variables */

    bool _display ;

    /* CAN HAVE ANY PRIVATE FUNCION */
};

}
```

```
1 /*-
2 Copyright (c) 2015 Author: Jagadeesh Vasudevamurthy
3 file: ../complex/complex.cpp ddequetest.cpp
4
5 On linux:
6 g++ ../complex/complex.cpp ddequetest.cpp
7 valgrind a.out
8 valgrind --leak-check=full -v a.out
9 -- All heap blocks were freed -- no leaks are possible
10
11 -----*/
12
13 /*-
14 This file test ddequeue object
15 -----*/
16
17 /*
18 All includes here
19 -----*/
20 #include "ddequeue.h"
21 #include "../complex/complex.h"
22 #include <deque> //Gold STL deque
23
24 /*
25 local to this file. Change verbose = true for debugging
26 -----*/
27 static bool verbose = false;
28
29 /*
30 Multiply integer by 10
31 -----*/
32 static void multiply_by_10(int& x){
33     x = x * 10;
34 }
35
36 /*
37 Multiply integer by 10
38 -----*/
39 static void multiply_by_10(int*& x){
40     multiply_by_10(*x);
41 }
42
43 /*
44 Multiply complex by 10
45 -----*/
46 static void multiply_by_10(complex& c){
47     int x, y;
48     c.getxy(x, y);
49     x = x * 10;
50     y = y * 10;
51     c.setxy(x, y);
52 }
53
54 /*
55 Multiply complex pointer by 10
56 -----*/
57 static void multiply_by_10(complex*& c){
58     multiply_by_10(*c);
59 }
60
61 /*
62 Print integer
63 -----*/
64 static void print_obj(int& x){
65     cout << x << " ";
66 }
```

```
67 /*-
68 *-----*
69 print pointer to integer
70 -----*/
71 static void print_obj(int*& x){
72     print_obj(*x);
73 }
74
75 /*-
76 Print complex
77 -----*/
78 static void print_obj(complex& c){
79     cout << c << " ";
80 }
81
82 /*-
83 Printcomplex pointer
84 -----*/
85 static void print_obj(complex*& c){
86     print_obj(*c);
87 }
88
89
90 /*-
91 compare integer
92 -----*/
93 static bool compare(const int& x, const int& y){
94     return (x == y);
95 }
96
97 /*-
98 compare integer pointer
99 -----*/
100 static bool compare(int* const& x, int* const& y){
101     return compare(*x, *y);
102 }
103
104 /*-
105 compare complex
106 -----*/
107 static bool compare(const complex& x, const complex& y){
108     return (x == y);
109 }
110
111 /*-
112 compare complex pointer
113 -----*/
114 static bool compare(complex* const& x, complex* const& y){
115     return (*x == *y);
116 }
117
118
119
120 /*-
121 construct an integer object
122 -----*/
123 static void construct_an_integer_object(int x, int& o){
124     o = x;
125 }
126
127 /*-
128 construct a integer* object
129 -----*/
130 static void construct_an_integer_star_object(int x, int*& o){
131     o = new(int)(x);
132 }
```

```
133 /*
134  *-----
135  construct a complex object
136  -----
137 static void construct_an_complex_object(int x, complex& o){
138     o.setxy(x, -x);
139 }
140
141 /*
142 construct a complex* object
143 -----
144 static void construct_an_complex_star_object(int x, complex*& o){
145     o = new(complex)(x, -x);
146 }
147
148 /*
149 delete an object
150 -----
151 template <typename T>
152 static void delete_obj(T& a) {
153     delete(a);
154 }
155
156
157 /*
158 swap
159 -----
160 template <typename T>
161 static void swap1(T& a, T& b) {
162     T t = a;
163     a = b;
164     b = t;
165 }
166
167 /*
168 print1
169 -----
170 template <typename T>
171 static void print1(const char* s, ddeque<T>& a, deque<T>& g, bool print = false) {
172     cout << s << endl;
173     cout << "-----" << endl;
174     assert(a.size() == g.size());
175     cout << "size = " << a.size() << " ";
176     cout << endl;
177     for (int i = 0; i < int(a.size()); i++) {
178         assert(compare(a[i], g[i]));
179         if (print) {
180             cout << "a[" << i << "] = " << a[i] << " ";
181         }
182     }
183     if (print) {
184         cout << endl;
185         cout << "-----" << endl;
186     }
187 }
188
189 /*
190 apply a function pf on elements of ddeque a
191 -----
192 template <typename T>
193 static void apply(const char* s, ddeque<T>& a, deque<T>& g, void(*pf)(T& x)) {
194     cout << s << endl;
195     cout << "-----" << endl;
196     typename ddeque<T>::iterator itt = a.begin(); //Note typename here.
197     //WILL NOT COMPILE IN LINUX without typename but works on Visual studio
198     //Hard to find this solution.
```

```
199  typename deque<T>::iterator ittg = g.begin();
200  while ((itt != a.end()) && (ittg != g.end())){
201      T& p = *itt;
202      T& pg = *ittg;
203      assert(compare(p, pg));
204      pf(p);
205      pf(pg);
206      ++itt;
207      ++ittg;
208  }
209  cout << endl;
210 }
211
212 /*-----*
213 a.front() -- Returns first element of the ddeque
214 a.back() -- Returns last element of the ddeque
215 a.push_front(v) -- v is inserted as the first element of the ddeque
216 a.push_back(v) -- v is inserted as the back element of the ddeque
217 a.pop_front() - First element of the ddeque is removed. Nothing is returned
218 a.pop_back() - Last element of the ddeque is removed. Nothing is returned
219 -----*/
220 template <typename T>
221 static void understanding_access(ddeque<T>& a, deque<T>& g, void(*pf)(int x, T& o)) {
222     print1("begin with", a, g);
223     const int MAX = 1000;
224     Random r;
225     for (int i = 0; i < MAX; i++){
226         T o;
227         T og;
228         int y = i + 77;
229         pf(y, o);
230         pf(y, og);
231         if (y % 2) {
232             //Even number- Push in front
233             a.push_back(o);
234             g.push_back(og);
235         } else {
236             //Even number- Push in back
237             a.push_front(o);
238             g.push_front(og);
239         }
240     }
241     print1("After inserting 1000 elements", a, g);
242     //shuffle now by MAX times
243
244     for (int i = 0; i < MAX; i++){
245         int x1 = i;
246         int x2 = (7777 - i) % MAX;
247         T& o1 = a[x1];
248         T& o1g = g[x1];
249         assert(compare(o1, o1g));
250         T& o2 = a[x2];
251         T& o2g = g[x2];
252         assert(compare(o2, o2g));
253         swap1(a[x1], a[x2]);
254         swap1(g[x1], g[x2]);
255         assert(compare(a[x1], g[x1]));
256         assert(compare(a[x2], g[x2]));
257     }
258     print1("After Shuffle", a, g);
259     T& y = a.front();
260     T& yg = g.front();
261     assert(compare(y, yg));
262     cout << "The front of the ddeque has " << y << endl;
263     T& z = a.back();
264     T& zg = g.back();
```

```
265 assert(compare(z, zg));
266 cout << "The back of the ddeque has " << z << endl;
267
268 print1("After front and back operation", a, g);
269
270 int x = a.size() - 1;
271 assert(compare(a[x], g[x]));
272 cout << "Random access of a[" << x << "] = " << a[x] << endl;
273 }
274
275 /*-----
276 a[0] ..... a[9]
277
278 begin() will point to a[0]
279 end() will NOT POINT to a[9], but to one element past a[9]
280 That means real end is: end()-1 ;
281 -----*/
282 template <typename T>
283 static void understanding_iterator(ddeque<T>& a, deque<T>& g, bool display = false) {
284 {
285     cout << "Understanding forward traversal" << endl;
286     typename ddeque<T>::iterator itt = a.begin();
287     typename deque<T>::iterator ittg = g.begin();
288     while ((itt != a.end()) && (ittg != g.end())) {
289         assert(compare(*itt, *(ittg)));
290         if (display) {
291             cout << *itt << " ";
292         }
293         ++itt;
294         ++ittg;
295     }
296     if (display) {
297         cout << endl;
298     }
299 }
300 {
301     cout << "Understanding reverse traversal" << endl;
302     typename ddeque<T>::iterator itt = a.end();
303     typename deque<T>::iterator ittg = g.end();
304 //cout << *itt << endl ; ----- CORE DUMPS
305     while ((itt != a.begin()) && (ittg != g.begin())) {
306         --itt; //decrement iterator before using
307         --ittg;
308         assert(compare(*(itt), *(ittg)));
309         if (display) {
310             cout << *itt << " ";
311         }
312     }
313     if (display) {
314         cout << endl;
315     }
316 }
317 cout << "-----" << endl;
318 }
319
320 /*-----
321 delete until empty
322 -----*/
323 template <typename T>
324 static void delete_until_empty(ddeque<T>& a, deque<T>& g, void(*df) (T& c)) {
325     cout << "Before deleteing the queue\n";
326     bool done = false;
327     while (!done) {
328         if (a.empty()) {
329             assert(g.empty());
330             done = true;
```

```
331 }
332     if (g.empty()) {
333         assert(a.empty());
334         done = true;
335     }
336     if (done) {
337         break;
338     } else {
339         /* delete back */
340         assert(a.size() == g.size());
341         T& z = a.back();
342         T& zg = g.back();
343         assert(compare(z, zg));
344         if (df) {
345             df(z);
346             df(zg);
347         }
348         a.pop_back();
349         g.pop_back();
350         assert(a.size() == g.size());
351     }
352     if (a.empty()) {
353         assert(g.empty());
354         done = true;
355     }
356 }
357     if (g.empty()) {
358         assert(a.empty());
359         done = true;
360     }
361     if (done) {
362         break;
363     } else {
364         /* delete front */
365         assert(a.size() == g.size());
366         T& z = a.front();
367         T& zg = g.front();
368         assert(compare(z, zg));
369         if (df) {
370             df(z);
371             df(zg);
372         }
373         a.pop_front();
374         g.pop_front();
375         assert(a.size() == g.size());
376     }
377 }
378 }
379 cout << "Before deleteing the queue\n";
380 }

383 /*
384 test
385 -----
386 template <typename T>
387 static void test(ddeque<T>& a, deque<T>& g, void(*pf)(int x, T& o), void(*df) (T& c)) {
388     print1("begin with", a, g);
389     understanding_access(a, g, pf);
390     understanding_iterator(a, g);
391     apply("multiply by 10", a, g, multiply_by_10);
392     cout << endl;
393     apply("print using iterator after multiplying by 10", a, g, print_obj);
394     cout << endl;
395     delete_until_empty(a, g, df);
396 }
```

```
397
398
399 /*-----*
400 main
401 -----*/
402 int main() {
403     complex::set_display(verbose);
404     {
405         ddequeue<complex> a;
406         a.push_back(complex(7, -8));
407     }
408     {
409         //TEST INTEGER
410         ddequeue<int> a;
411         deque<int> g; //Gold
412         test<int>(a, g, construct_an_integer_object, nullptr);
413     }
414     {
415         //TEST pointer to INTEGER
416         ddequeue<int*> a;
417         deque<int*> g; //Gold
418         test<int*>(a, g, construct_an_integer_star_object, delete_obj);
419     }
420     {
421         //TEST complex
422         ddequeue<complex> a;
423         deque<complex> g; //Gold
424         test<complex>(a, g, construct_an_complex_object, nullptr);
425     }
426     {
427         //TEST pointer to complex
428
429         ddequeue<complex*> a;
430         deque<complex*> g; //Gold
431         test<complex*>(a, g, construct_an_complex_star_object, delete_obj);
432     }
433     return 1;
434 }
435
436
437 //EOF
438
439
```

8.6. PROBLEM SET

VASUDEVAMURTHY

Chapter 9

Searching and Sorting

9.1 Introduction

9.2 Base *dsort* class

Base class *dsort*

```
template <typename T>
class dsort {
public:
    dsort(darray<T>& d,
          int s,
          int(*cf) (const T& c1, const T& c2),
          const char* t,
          bool dis = false);
    ~dsort();
    void sort();
    virtual void sorting_algorithm() = 0;
//YOU WILL WRITE SORT. I don't know how to write
protected:
```

```
/* private data */
darray<T>& _darray;
int _size;
int(*_cf) (const T& c1, const T& c2);
const char* _algnname;
bool _display;
statistics _stat;
```

```
}
```

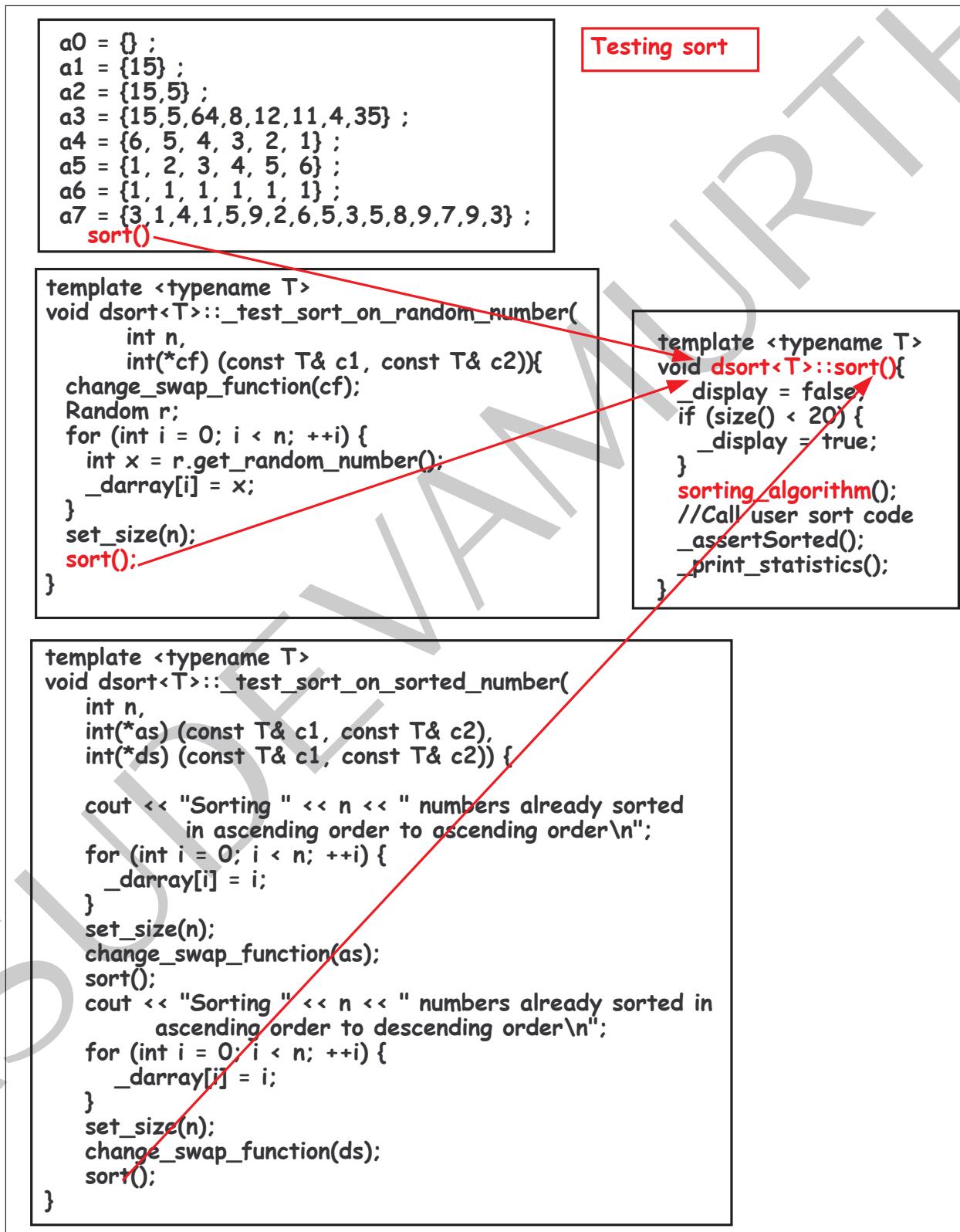
```
class statistics {
public:

private:
    int _nc; //num compare
    int _ns; //num_swap
    int _ni; //num_iteration
    int _nr; //byn_recursion
};
```

Figure 9.1: Base *dsort* class

9.3 Testing *dsort* class

9.3. TESTING DSORT CLASS



9.4 Measuring the complexity of sorting algorithms

Measuring basic steps taken by a sorting algorithm

```

template <typename T>
void dsort<T>::_print_statistics() {
    cout << "----- " << _algnname << " ----- \n";
    int n = size();
    int c = get_num_compare();
    int s = get_num_swap();
    int it = get_num_iteration();
    int r = get_num_recursion();
    int t = c + s;
    cout << "#size(n) = " << n << endl;
    cout << "#comparison(c) = " << c << endl;
    cout << "#swap(s) = " << s << endl;
    if (it) {
        cout << "#Num iteration(it) = " << it << endl;
    }
    if (r) {
        cout << "#Num recursion(r) = " << r << endl;
    }
    cout << "#c+s = " << t << endl;

    if (n > 0) {
        double x = double(t) / double(n);
        cout << "#T(n) = (c+s)/(n) = " << x << "(n)" << endl;
        double log2n = log2(n);
        x = double(t) / double(n*log2n);
        cout << "#T(n) = (c+s)/(n*log2n) = " << x << "(n *log2n)" << endl;
        x = double(t) / double(n*n);
        cout << "#T(n) = (c+s)/(n^2) = " << x << "(n^2)" << endl;
    } else {
        cout << "Zero element array\n";
    }
    cout << "----- \n";
}

```

Figure 9.3: Measuring number of basic steps

9.5 Correctness of sorting algorithm

Proving that the array is sorted correctly

```
template <typename T>
void dsort<T>::_assertSorted(int s, int e){
    int p = 0;
    for (int i = s; i <= e; ++i) {
        int r = _cf(_darray[p], _darray[i]);
        assert(r >= 0);
        p = i;
    }
}
```

```
template <typename T>
void dsort<T>::_assertSorted(){
    int s = size();
    _assertSorted(0, s - 1);
}
```

Figure 9.4: Proving that the array is sorted

9.6 Array based bubble sort

Bubble sort

Idea: Bubble a larger element to the end

Method: compare two neighbouring elements

0 1 2 3 4 5 6 7		original
25 57 48 37 12 92 86 33		<0,1>
25 57		<1,2>
48 57		<2,3>
37 57		<3,4>
12 57		<4,5>
57 92		<5,6>
86 92		<6,7>
33 92		
25 48 37 12 57 86 33 92		n comparison

Repeat until no bubbling possible

0:25 57 48 37 12 92 86 33		
1:25 48 37 12 57 86 33 92		n comparison
2:25 37 12 48 57 33 86 92		n-1 comparison
3:25 12 37 48 33 57 86 92		n-2 comparison
4:12 25 37 33 48 57 86 92		n-3 comparison
5:12 25 33 37 48 57 86 92		n-4 comparison
6:12 25 33 37 48 57 86 92		n-5 comparison

Worst case: $O(n^2)$ algorithm

Best case: $O(n)$

Do not use for larger n

Figure 9.5: Bubble sort - Concept

```

bubblesort.h
template <typename T>
class bubblesort: public dsort<T> {
public:
    bubblesort(darray<T>& d,
               int s,
               int(*cf) (const T& c1, const T& c2),
               const char* t,
               bool dis = false);
    ~bubblesort(){}
    virtual void sorting_algorithm();
    void bubble_sort();
private:
    bool _bubble_loop(int end);
};

bubblesort.hpp
template <typename T>
bool bubblesort<T>::_bubble_loop(int end){
    bool exchanged = false;
    for (int i = 0; i < end - 1; ++i) {
        int j = i + 1;
        dsort<T>::inc_num_compare();
        if (dsort<T>::_compare_obj(i, j) < 0) {
            dsort<T>::inc_num_swap();
            exchanged = true;
            dsort<T>::_swap_obj(i, j); O(n)
        }
    }
    return exchanged;
}

bubblesort.hpp
template <typename T>
void bubblesort<T>::sorting_algorithm(){
    dsort<T>::reset_stat();
    int n = dsort<T>::size();
    if (n) {
        bool exchanged = false;
        do {
            dsort<T>::inc_num_iteration();
            exchanged = _bubble_loop(n);
        } while (exchanged); O(n*n) = O(n2)
    } else {
        cout << "Zero sized array\n";
    }
    dsort<T>::_assertSorted();
}

bubblesort.hpp
template <typename T>
void bubblesort<T>::bubble_sort(){
    sorting_algorithm();
}

```

CONCRETE CLASS

0:25 57 48 37 12 92 86 33
1:25 48 37 12 57 86 33 92
2:25 37 12 48 57 33 86 92
3:25 12 37 48 33 57 86 92
4:12 25 37 33 48 57 86 92
5:12 25 33 37 48 57 86 92
6:12 25 33 37 48 57 86 92

Figure 9.6: Bubble sort - Code

9.6. ARRAY BASED BUBBLE SORT

bubblesorttest.cpp

```
#include "dsort.h"
#include "bubblesort.h"
#include "../complex/complex.h"

static void int_test(){
    int tests
    {
        darray<int> b;
        int a[] = { 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3 };
        int sz = sizeof(a) / sizeof(int);
        dsort<int>::fill(b, a, sz);
        bubblesort<int> bb(b,sz,int_descending_order,"Bubble sort",verbose);
        bb.sort(); Derived class test
    }
}

darray<int> b;
bubblesort<int> a(b,0, nullptr, "Bubble sort", verbose);
a.int_test(); Base class tests
```

```
static void complex_test(){
    Complex object tests
    {
        darray<complex> b;
        int a[] = { 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3 };
        int sz = sizeof(a) / sizeof(int);
        dsort<complex>::fill(b, a, sz);
        bubblesort<complex> bb(b, sz, complex_descending_order, "Bubble sort", verbose);
        bb.sort(); Derived class test
    }
}

darray<complex> b;
bubblesort<complex> a(b, 0, nullptr, "Bubble sort", verbose);
a.complex_test(); Base class tests
```

Figure 9.7: Testing Bubble sort for *int* and *complex* objects

bubblesorttest.cpp

```

#include "dsort.h"
#include "bubblesort.h"
#include "../complex/complex.h"

static void complex_pntr_test(){
{
    darray<complex*> b;
    int a[] = { 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3 };
    int sz = sizeof(a) / sizeof(int);
    dsort<complex*>::fill(b, a, sz);
    bubblesort<complex*> bb(b, sz, complex_descending_order, "Bubble sort", verbose);
    bb.sort(); ← Derived class test
    bb.free_complex_data(b, sz);
}

darray<complex*> b;
bubblesort<complex*> a(b, 0, nullptr, "Bubble sort", verbose);
a.complex_pntr_test();
}

```

Pntr to complex object tests

Base class tests

```

template <>
void dsort<complex*>::fill(darray<complex*>& da, const int* a, int size){
    for (int i = 0; i < size; ++i){
        da[i] = new complex(a[i]);
    }
}

```

dsort.hpp

```

template <typename T>
void dsort<T>::fill(darray<T>& da, const int* a, int size){
    for (int i = 0; i < size; ++i){
        da[i] = a[i];
    }
}

```

dsort.hpp

Figure 9.8: Testing Bubble sort for *complex** object

9.6. ARRAY BASED BUBBLE SORT

9.6.1 Measuring bubble sort

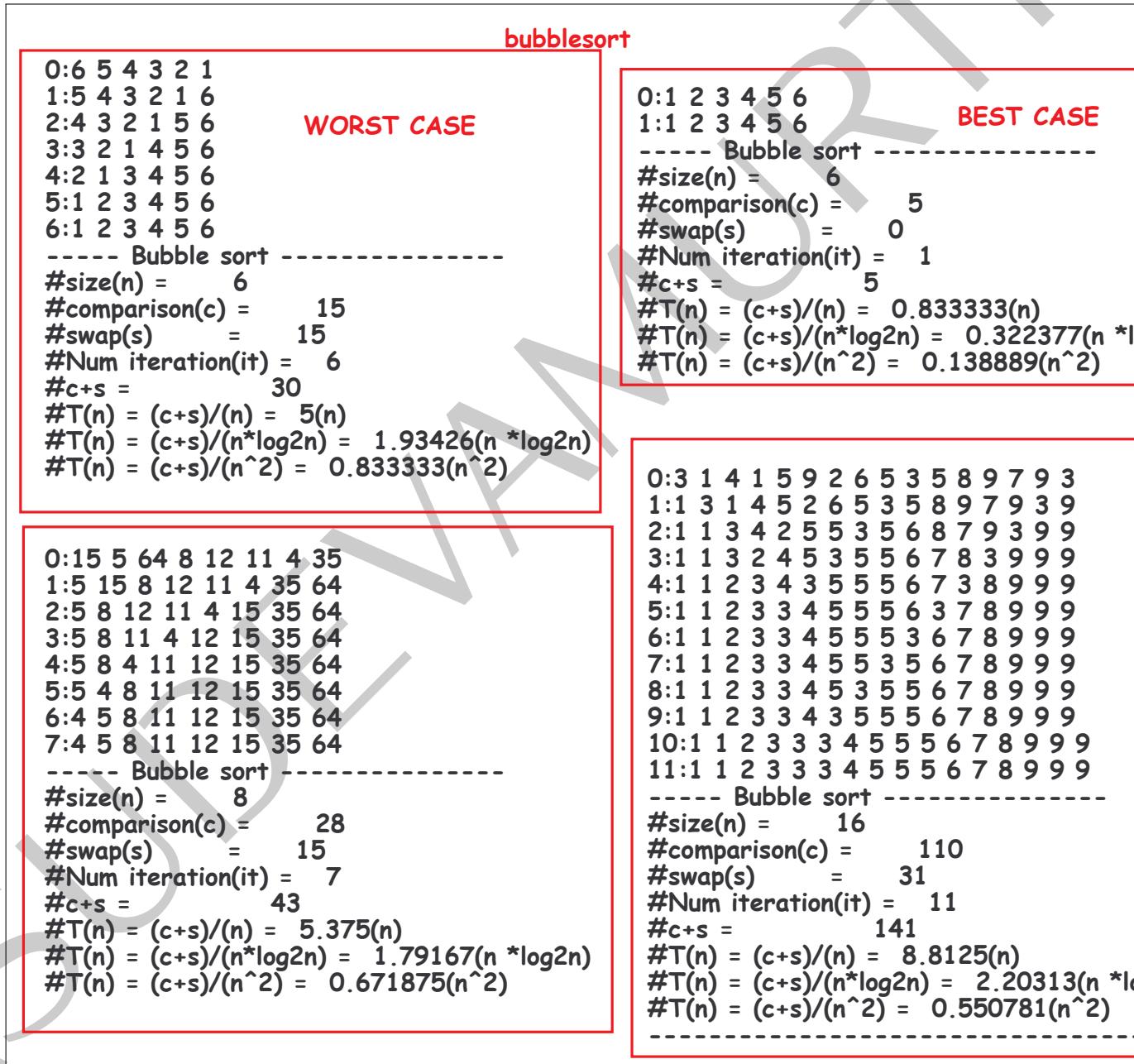


Figure 9.9: Measuring Bubble sort on various small arrays

9.6. ARRAY BASED BUBBLE SORT

bubblesort on n random numbers	
<pre>#size(n) = 1000 #comparison(c) = 499464 #swap(s) = 251528 #Num iteration(it) = 991 #c+s = 750992 #T(n) = (c+s)/(n) = 750.992(n) #T(n) = (c+s)/(n*log2n) = 75.357(n *log2n) #T(n) = (c+s)/(n^2) = 0.750992(n^2)</pre>	<pre>#size(n) = 2000 #comparison(c) = 1997674 #swap(s) = 997757 #Num iteration(it) = 1948 #c+s = 2995431 #T(n) = (c+s)/(n) = 1497.72(n) #T(n) = (c+s)/(n*log2n) = 136.581(n *log2n) #T(n) = (c+s)/(n^2) = 0.748858(n^2)</pre>
<pre>#size(n) = 3000 #comparison(c) = 4497554 #swap(s) = 2323494 #Num iteration(it) = 2956 #c+s = 6821048 #T(n) = (c+s)/(n) = 2273.68(n) #T(n) = (c+s)/(n*log2n) = 196.843(n *log2n) #T(n) = (c+s)/(n^2) = 0.757894(n^2)</pre>	<pre>#size(n) = 4000 #comparison(c) = 7984139 #swap(s) = 3977318 #Num iteration(it) = 3833 #c+s = 11961457 #T(n) = (c+s)/(n) = 2990.36(n) #T(n) = (c+s)/(n*log2n) = 249.91(n *log2n) #T(n) = (c+s)/(n^2) = 0.747591(n^2)</pre>
<p>Sorting 1000 numbers already sorted in ascending order to ascending order</p> <p>Running Bubble sort</p> <p>----- Bubble sort -----</p> <pre>#size(n) = 1000 #comparison(c) = 999 #swap(s) = 0 #Num iteration(it) = 1 #c+s = 999 #T(n) = (c+s)/(n) = 0.999(n) #T(n) = (c+s)/(n*log2n) = 0.100243(n *log2n) #T(n) = (c+s)/(n^2) = 0.000999(n^2)</pre>	<p>BEST CASE</p>
<p>Sorting 1000 numbers already sorted in ascending order to descending order</p> <p>Running Bubble sort</p> <p>----- Bubble sort -----</p> <pre>#size(n) = 1000 #comparison(c) = 499500 #swap(s) = 499500 #Num iteration(it) = 1000 #c+s = 999000 #T(n) = (c+s)/(n) = 999(n) #T(n) = (c+s)/(n*log2n) = 100.243(n *log2n) #T(n) = (c+s)/(n^2) = 0.999(n^2)</pre>	<p>WORST CASE</p>

Figure 9.10: Measuring Bubble sort on various big arrays

9.7 Linked list based bubble sort

9.8 Array based insertion sort

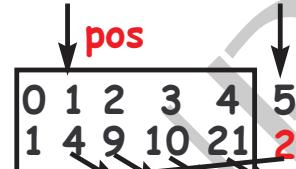
9.8. ARRAY BASED INSERTION SORT



Figure 2.1: Sorting a hand of cards using insertion sort.

0 1 2 3 4 5 6 7 8 9 10
10 2 9 21 1 4 45 0 -1 -3 -5

10
2 10
2 9 10
2 9 10 21
1 2 9 10 21
1 2 4 9 10 21
1 2 4 9 10 21 45
0 1 2 4 9 10 21 45
-1 0 1 2 4 9 10 21 45
-3 -1 0 1 2 4 9 10 21 45
-5 -3 -1 0 1 2 4 9 10 21 45



```
for (int j = i-1; j>=pos; j--) {
    a[j+1] = a[j];
}
```

10 is in the right place 0
Insert 2 in the right place: 0
Insert 9 in the right place: 1
Insert 21 in the right place: 3
Insert 1 in the right place: 0
Insert 4 in the right place: 2
Insert 45 in the right place: 6
Insert 0 in the right place: 7
Insert -1 in the right place: 0
Insert -3 in the right place: 0
Insert -5 in the right place: 0

Idea 1: Insert an object at the right place

Idea 2: Left side of the array is sorted

Idea 3: On the sorted array, insert position can be found scanning from right to left in $O(n)$

$$n(n) = O(n^2)$$

Figure 9.11: Insertion sort - concept

```
insertionsort.h
class insertionsort: public dsort<T> {
public:
    insertionsort(darray<T>& d,
                  int s,
                  int(*cf) (const T& c1, const T& c2),
                  const char* t,
                  bool dis = false);
    ~insertionsort(){}
    virtual void sorting_algorithm();
    void insertion_sort();
private:
};
```

Concrete class

```
insertionsort.hpp
template <typename T>
void insertionsort<T>::sorting_algorithm(){
    cout << "Running " << dsort<T>::_alname << endl;
    //WRITE CODE HERE
    dsort<T>::_assertSorted();
}

template <typename T>
void insertionsort<T>::insertion_sort(){
    sorting_algorithm();
}
```

```
insertionsorttest.cpp
static void complex_pntr_test(){
{
    darray<complex*> b;
    int a[] = { 3, 1, 4, 1, 5};
    int sz = sizeof(a) / sizeof(int);
    dsort<complex*>::fill(b, a, sz);
    insertionsort<complex*> bb(b, sz,
        complex_descending_order,
        "Insertion sort", verbose);
    bb.sort();
    bb.free_complex_data(b, sz);
```

Figure 9.12: Insertion sort - code

9.8. ARRAY BASED INSERTION SORT

9.8.1 Measuring insertion sort

CHAPTER 9. SEARCHING AND SORTING

Insertion sort	
<pre> ----- Search 5 in array below----- 0 1 2 3 4 5 6 5 4 3 2 1 Iteration 1: Inserting item 5 which is in position 1 to new position 0 ----- Search 4 in array below----- 0 1 2 3 4 5 5 6 4 3 2 1 Iteration 2: Inserting item 4 which is in position 2 to new position 0 ----- Search 3 in array below----- 0 1 2 3 4 5 4 5 6 3 2 1 Iteration 3: Inserting item 3 which is in position 3 to new position 0 ----- Search 2 in array below----- 0 1 2 3 4 5 3 4 5 6 2 1 Iteration 4: Inserting item 2 which is in position 4 to new position 0 ----- Search 1 in array below----- 0 1 2 3 4 5 2 3 4 5 6 1 Iteration 5: Inserting item 1 which is in position 5 to new position 0 ----- Insertion sort ----- #size(n) = 6 #comparison(c) = 15 #swap(s) = 15 #c+s = 30 #T(n) = (c+s)/(n) = 5(n) #T(n) = (c+s)/(n*log2n) = 1.93426(n *log2n) #T(n) = (c+s)/(n^2) = 0.833333(n^2) -----</pre>	<pre> BEST CASE ----- Search 2 in array below----- 0 1 2 3 4 5 1 2 3 4 5 6 Iteration 1: Inserting item 2 which is in position 1 to new position 1 ----- Search 3 in array below----- 0 1 2 3 4 5 1 2 3 4 5 6 Iteration 2: Inserting item 3 which is in position 2 to new position 2 ----- Search 4 in array below----- 0 1 2 3 4 5 1 2 3 4 5 6 Iteration 3: Inserting item 4 which is in position 3 to new position 3 ----- Search 5 in array below----- 0 1 2 3 4 5 1 2 3 4 5 6 Iteration 4: Inserting item 5 which is in position 4 to new position 4 ----- Search 6 in array below----- 0 1 2 3 4 5 1 2 3 4 5 6 Iteration 5: Inserting item 6 which is in position 5 to new position 5 ----- Insertion sort ----- #size(n) = 6 #comparison(c) = 5 #swap(s) = 0 #c+s = 5 #T(n) = (c+s)/(n) = 0.833333(n) #T(n) = (c+s)/(n*log2n) = 0.322377(n *log2n) #T(n) = (c+s)/(n^2) = 0.138889(n^2) -----</pre>
<pre> 0 1 2 3 4 5 6 7 15 5 64 8 12 11 4 35 Iteration 1: Inserting item 5 which is in position 1 to new position 0 ----- Search 64 in array below----- 0 1 2 3 4 5 6 7 5 15 64 8 12 11 4 35 Iteration 2: Inserting item 64 which is in position 2 to new position 2 ----- Search 8 in array below----- 0 1 2 3 4 5 6 7 5 15 64 8 12 11 4 35 Iteration 3: Inserting item 8 which is in position 3 to new position 1 ----- Search 12 in array below----- 0 1 2 3 4 5 6 7 5 8 15 64 12 11 4 35 Iteration 4: Inserting item 12 which is in position 4 to new position 2 ----- Search 11 in array below----- 0 1 2 3 4 5 6 7 5 8 12 15 64 11 4 35 Iteration 5: Inserting item 11 which is in position 5 to new position 2 ----- Search 4 in array below----- 0 1 2 3 4 5 6 7 5 8 11 12 15 64 4 35 Iteration 6: Inserting item 4 which is in position 6 to new position 0 ----- Search 35 in array below----- 0 1 2 3 4 5 6 7 4 5 8 11 12 15 64 35 Iteration 7: Inserting item 35 which is in position 7 to new position 6 ----- Insertion sort ----- #size(n) = 8 #comparison(c) = 20 #swap(s) = 15 #c+s = 35 #T(n) = (c+s)/(n) = 4.375(n) #T(n) = (c+s)/(n*log2n) = 1.45833(n *log2n) #T(n) = (c+s)/(n^2) = 0.546875(n^2) -----</pre>	

Figure 9.13: Measuring insertion sort on various small arrays

9.8. ARRAY BASED INSERTION SORT

insertionsort on n random numbers	
<pre>#size(n) = 1000 #comparison(c) = 254890 #swap(s) = 253896 #c+s = 508786 #T(n) = (c+s)/(n) = 508.786(n) #T(n) = (c+s)/(n*log2n) = 51.0533(n *log2n) #T(n) = (c+s)/(n^2) = 0.508786(n^2)</pre>	<pre>#size(n) = 2000 #comparison(c) = 1026369 #swap(s) = 1024377 #c+s = 2050746 #T(n) = (c+s)/(n) = 1025.37(n) #T(n) = (c+s)/(n*log2n) = 93.5066(n *log2n) #T(n) = (c+s)/(n^2) = 0.512687(n^2)</pre>
<pre>#size(n) = 3000 #comparison(c) = 2224620 #swap(s) = 2221626 #c+s = 4446246 #T(n) = (c+s)/(n) = 1482.08(n) #T(n) = (c+s)/(n*log2n) = 128.31(n *log2n) #T(n) = (c+s)/(n^2) = 0.494027(n^2)</pre>	<pre>#size(n) = 4000 #comparison(c) = 4061422 #swap(s) = 4057432 #c+s = 8118854 #T(n) = (c+s)/(n) = 2029.71(n) #T(n) = (c+s)/(n*log2n) = 169.626(n *log2n) #T(n) = (c+s)/(n^2) = 0.507428(n^2)</pre>
<p>Sorting 1000 numbers already sorted in ascending order to ascending order</p> <p>Running Insertion sort</p> <p>----- Insertion sort -----</p> <pre>#size(n) = 1000 #comparison(c) = 999 #swap(s) = 0 #c+s = 999 #T(n) = (c+s)/(n) = 0.999(n) #T(n) = (c+s)/(n*log2n) = 0.100243(n *log2n) #T(n) = (c+s)/(n^2) = 0.000999(n^2)</pre>	
<p>BEST CASE</p> <p>Sorting 1000 numbers already sorted in ascending order to descending order</p> <p>----- Insertion sort -----</p> <pre>#size(n) = 1000 #comparison(c) = 499500 #swap(s) = 499500 #c+s = 999000 #T(n) = (c+s)/(n) = 999(n) #T(n) = (c+s)/(n*log2n) = 100.243(n *log2n) #T(n) = (c+s)/(n^2) = 0.999(n^2)</pre>	
<p>WORST CASE</p>	

Figure 9.14: Measuring insertion sort on various big arrays

9.9 Linked list based insertion sort

9.10 Array based binary search

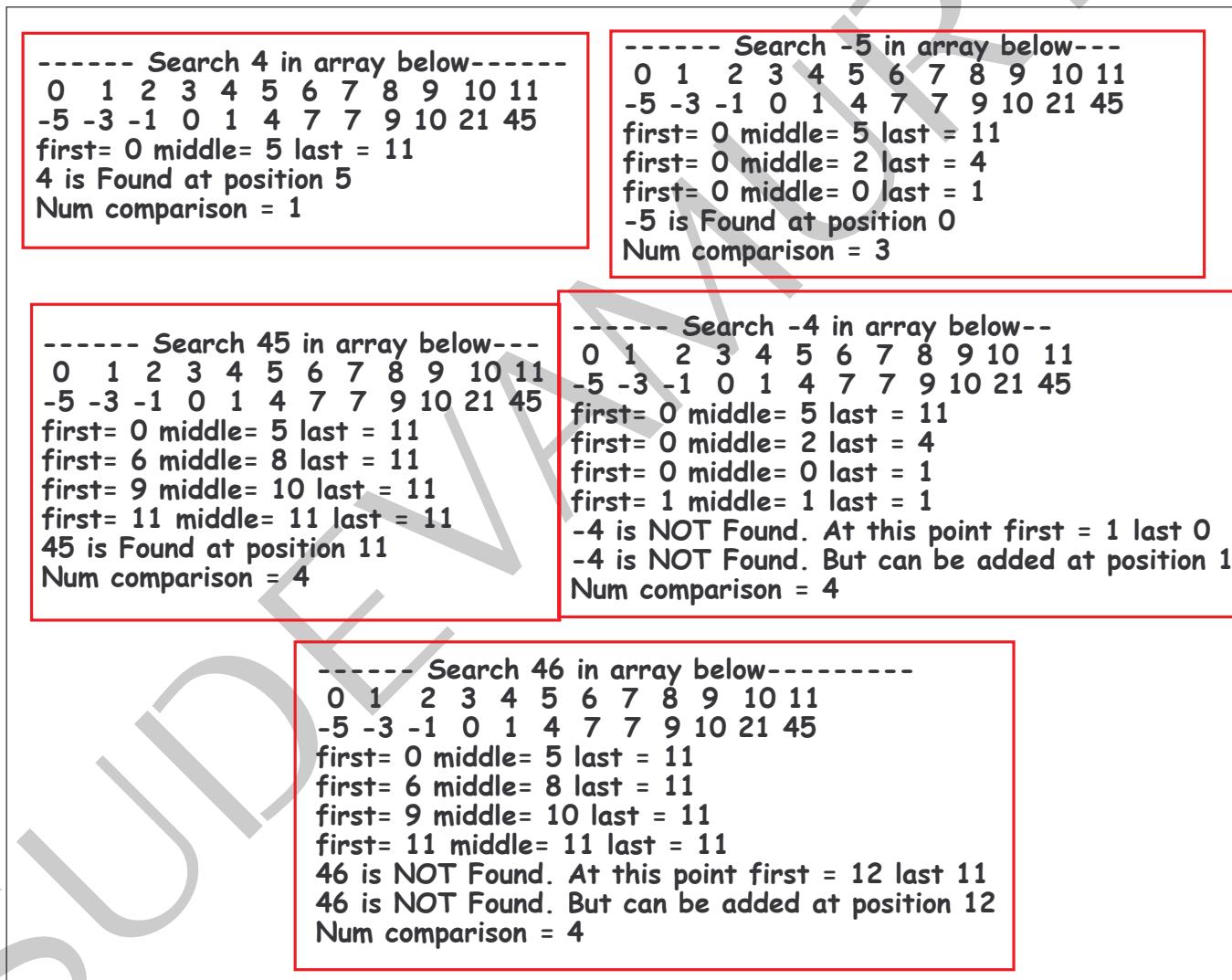


Figure 9.15: Binary search - Concept

9.10. ARRAY BASED BINARY SEARCH

```
binarysearch.h
class binarysearch {
public:
    binarysearch(darray<T>& d,
                 int s,
                 int(*cf) (const T& c1, const T& c2),
                 bool dis = false);
    ~binarysearch(){}
    bool binary_search(const T& tofind, int& pos, int& num_compare);
    bool binary_search(const T& tofind, int s, int e, int& pos, int& num_compare);

private:
    /* private data */
    darray<T>& _darray;
    int _size;
    int(*_cf) (const T& c1, const T& c2);
    bool _display;
};

binarysearch.hpp
template <typename T>
bool binarysearch<T>::binary_search(
    const T& r,
    int first,
    int last,
    int& pos,
    int& num_compare) {
    _assertSorted(first, (last+1)); //Array must be in ascending or descending order
    //WRITE YOUR CODE HERE
}

template <typename T>
bool binarysearch<T>::binary_search(const T& tofind, int& pos, int& num_compare){
    int n = _size;
    return _binary_search(tofind, 0, n-1, pos, num_compare);
}
```

Figure 9.16: Binary search - Code

```
binarysearchtest.cpp

static void int_test(){
{
    //          0   1   2   3   4   5   6   7   8   9   10  11
    int a[] = { -5, -3, -1, 0, 1, 4, 7, 7, 9, 10, 21, 45 };
    int sz = sizeof(a) / sizeof(int);
    darray<int> b;
    binarysearch<int>::fill(b, a, sz);
    binarysearch<int> bb(b, sz, intAscending_order, verbose);
    int pos = -1;
    int nc = 0;
    bb.binary_search(4, pos, nc);
    assert(pos == 5);
    bb.binary_search(-5, pos, nc);
    assert(pos == 0);
    bb.binary_search(45, pos, nc);
    assert(pos == 11);
    bb.binary_search(-4, pos, nc);
    assert(pos == 1);
    bb.binary_search(46, pos, nc);
    assert(pos == 12);
    bb.binary_search(-6, pos, nc);
    assert(pos == 0);
    bb.binary_search(44, pos, nc);
    assert(pos == 11);
}

{
    //          0   1   2   3   4   5   6   7   8   9   10  11
    int a[] = { 45, 21, 10, 9, 7, 7, 4, 1, 0, -1, -3, -5 };
    int sz = sizeof(a) / sizeof(int);
    darray<int> b;
    binarysearch<int>::fill(b, a, sz);
    binarysearch<int> bb(b, sz, intDescending_order, verbose);
    int pos = -1;
    int nc = 0;
    bb.binary_search(4, pos, nc);
    assert(pos == 6);
    bb.binary_search(-5, pos, nc);
    assert(pos == 11);
    bb.binary_search(45, pos, nc);
    assert(pos == 0);
    bb.binary_search(-4, pos, nc);
    assert(pos == 11);
    bb.binary_search(46, pos, nc);
    assert(pos == 0);
    bb.binary_search(-6, pos, nc);
    assert(pos == 12);
    bb.binary_search(44, pos, nc);
    assert(pos == 1);
}
```

Figure 9.17: Testing binary search

9.10. ARRAY BASED BINARY SEARCH

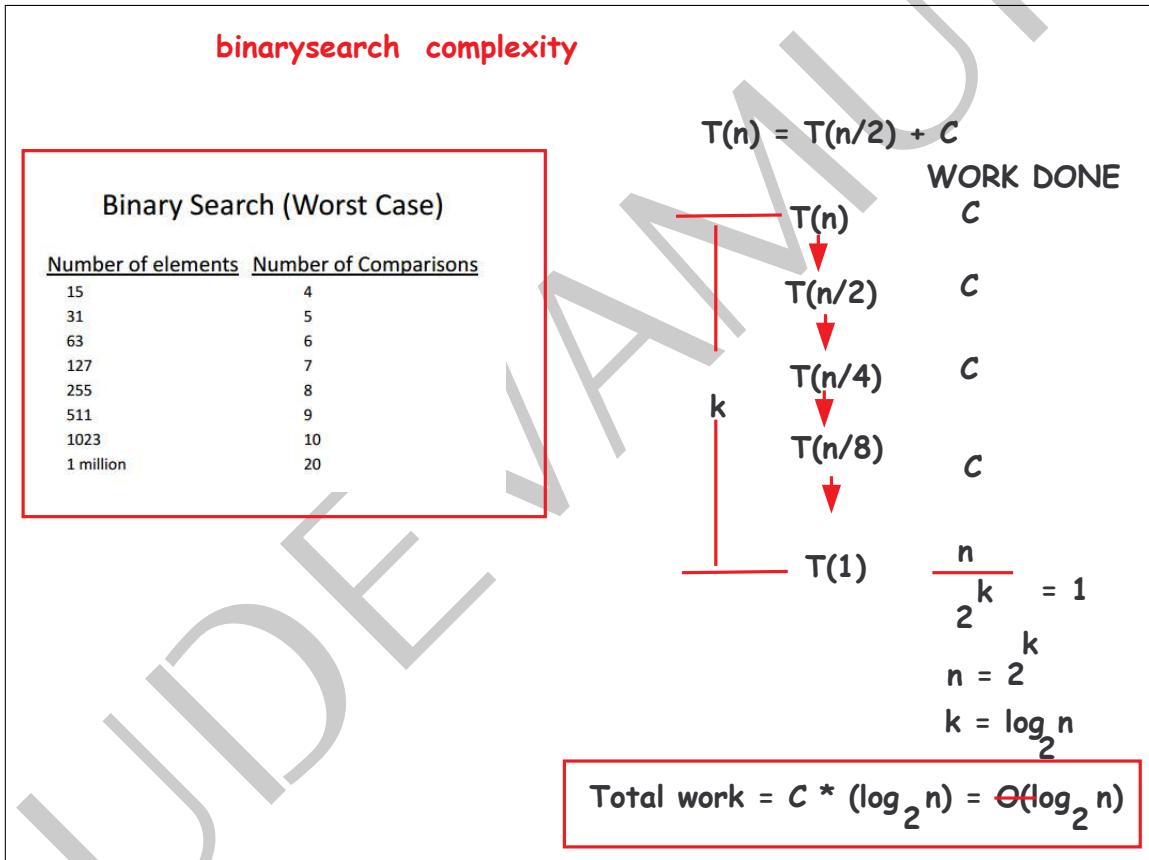


Figure 9.18: Complexity of binary search

9.11 Linked list based binary search

9.12 Array based binary insertion sort

9.12. ARRAY BASED BINARY INSERTION SORT

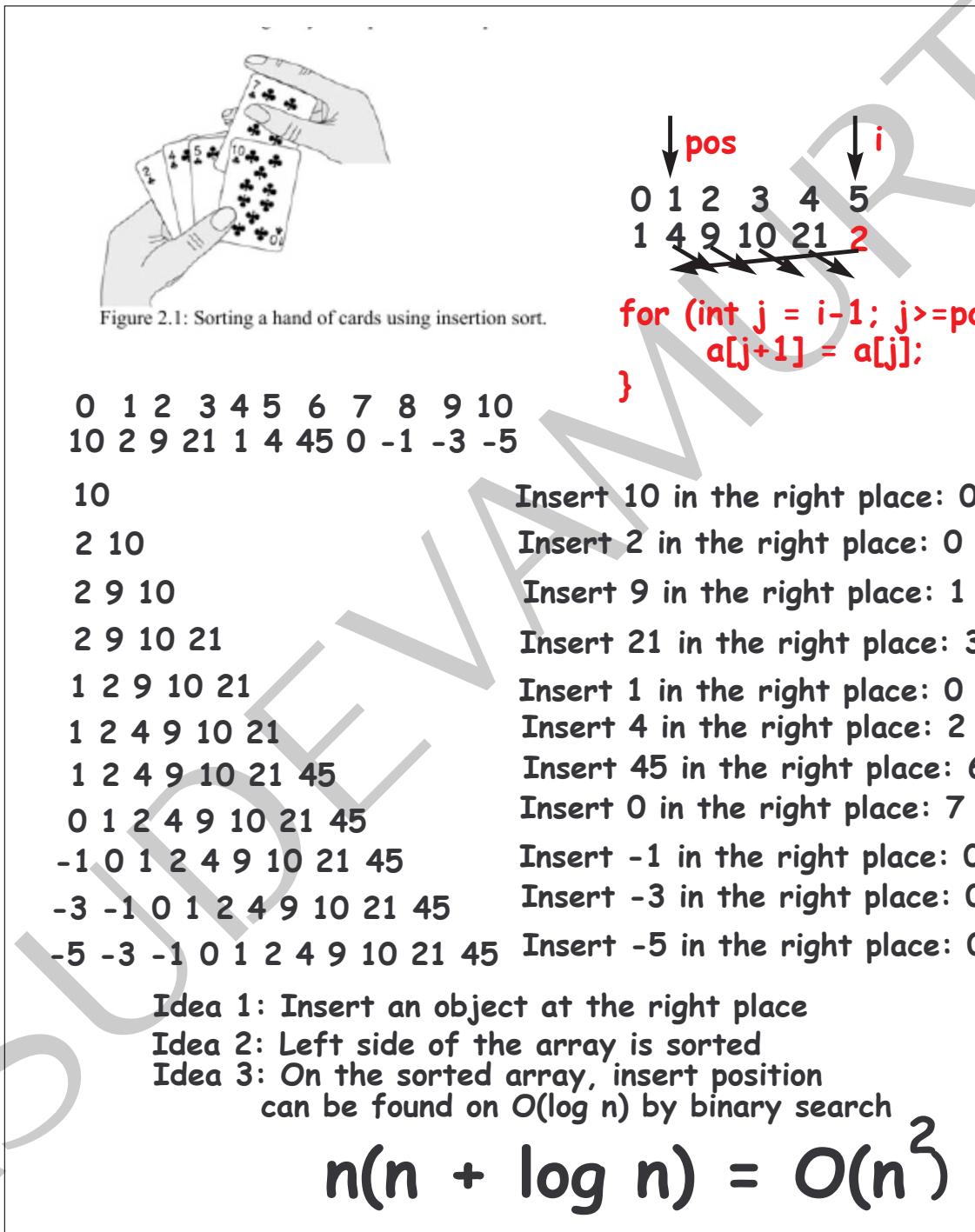


Figure 9.19: Binary insertion sort – concept

CHAPTER 9. SEARCHING AND SORTING

```
template <typename T>          binaryinsertionsort.h
class binaryinsertionsort: public dsort<T> {
public:
    binaryinsertionsort(darray<T>& d,
                        int s,
                        int(*cf) (const T& c1, const T& c2),
                        const char* t,
                        bool dis = false);
    ~binaryinsertionsort(){}
    virtual void sorting_algorithm();
    void binary_insertion_sort();
private:
};

template <typename T>          binaryinsertionsort.hpp
void binaryinsertionsort<T>::sorting_algorithm()
{
    cout << "Running " << dsort<T>::_algname << endl;
    //WRITE CODE HERE
    dsort<T>::_assertSorted();
}

template <typename T>
void binaryinsertionsort<T>::binary_insertion_sort(){
    sorting_algorithm();
}

static void complex_test()
{
    darray<complex> b;
    binaryinsertionsorttest.cpp
    int a[] = { 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3 };
    int sz = sizeof(a) / sizeof(int);
    dsort<complex>::fill(b, a, sz);
    binaryinsertionsort<complex> bb(b, sz, complex_descending_order, "BinaryInsertion sort", verbose);
    bb.sort();
}

darray<complex> b;
binaryinsertionsort<complex> a(b, 0, nullptr, "BinaryInsertion sort", verbose);
a.complex_test();
```

Figure 9.20: Binary Insertion sort – code

9.12. ARRAY BASED BINARY INSERTION SORT

9.12.1 Measuring binary insertion sort

CHAPTER 9. SEARCHING AND SORTING

WORST CASE	Binary insertion sort BEST CASE(Insertion sort is better)
<pre> Running BinaryInsertion sort 0 1 2 3 4 5 6 5 4 3 2 1 ----- Search 5 in array below----- 0 6 first= 0 middle= 0 last = 0 5 is NOT Found. At this point first = 0 last = -1 5 is NOT Found. But can be added at position 0 Num comparison = 1 ----- Iteration 1: Inserting item 5 which is in position 1 to new position 0 ----- Search 4 in array below----- 0 1 5 6 first= 0 middle= 0 last = 1 4 is NOT Found. At this point first = 0 last = -1 4 is NOT Found. But can be added at position 0 Num comparison = 1 ----- Iteration 2: Inserting item 4 which is in position 2 to new position 0 ----- Search 3 in array below----- 0 1 2 4 5 6 first= 0 middle= 1 last = 2 first= 0 middle= 0 last = 0 3 is NOT Found. At this point first = 0 last = -1 3 is NOT Found. But can be added at position 0 Num comparison = 2 ----- Iteration 3: Inserting item 3 which is in position 3 to new position 0 ----- Search 2 in array below----- 0 1 2 3 3 4 5 6 first= 0 middle= 1 last = 3 first= 0 middle= 0 last = 0 2 is NOT Found. At this point first = 0 last = -1 2 is NOT Found. But can be added at position 0 Num comparison = 2 ----- Iteration 4: Inserting item 2 which is in position 4 to new position 0 ----- Search 1 in array below----- 0 1 2 3 4 2 3 4 5 6 first= 0 middle= 2 last = 4 first= 0 middle= 0 last = 1 1 is NOT Found. At this point first = 0 last = -1 1 is NOT Found. But can be added at position 0 Num comparison = 2 ----- Iteration 5: Inserting item 1 which is in position 5 to new position 0 0 1 2 3 4 5 1 2 3 4 5 6 ----- BinaryInsertion sort ----- #size(n) = 6 #comparison(c) = 8 #swap(s) = 15 #c+s = 23 #T(n) = (c+s)/(n) = 3.83333(n) #T(n) = (c+s)/(n*log2n) = 1.48294(n *log2n) #T(n) = (c+s)/(n^2) = 0.638889(n^2) </pre>	<pre> Running BinaryInsertion sort 0 1 2 3 4 5 1 2 3 4 5 6 ----- Search 2 in array below----- 0 1 first= 0 middle= 0 last = 0 2 is NOT Found. At this point first = 1 last = 0 2 is NOT Found. But can be added at position 1 Num comparison = 1 ----- Iteration 1: Inserting item 2 which is in position 1 to new position 1 ----- Search 3 in array below----- 0 1 1 2 first= 0 middle= 0 last = 1 first= 1 middle= 1 last = 1 3 is NOT Found. At this point first = 2 last = 1 3 is NOT Found. But can be added at position 2 Num comparison = 2 ----- Iteration 2: Inserting item 3 which is in position 2 to new position 2 ----- Search 4 in array below----- 0 1 2 1 2 3 first= 0 middle= 1 last = 2 first= 2 middle= 2 last = 2 4 is NOT Found. At this point first = 3 last = 2 4 is NOT Found. But can be added at position 3 Num comparison = 2 ----- Iteration 3: Inserting item 4 which is in position 3 to new position 3 ----- Search 5 in array below----- 0 1 2 3 1 2 3 4 first= 0 middle= 1 last = 3 first= 2 middle= 2 last = 3 first= 3 middle= 3 last = 3 5 is NOT Found. At this point first = 4 last = 3 5 is NOT Found. But can be added at position 4 Num comparison = 3 ----- Iteration 4: Inserting item 5 which is in position 4 to new position 4 ----- Search 6 in array below----- 0 1 2 3 4 1 2 3 4 5 first= 0 middle= 2 last = 4 first= 3 middle= 3 last = 4 first= 4 middle= 4 last = 4 6 is NOT Found. At this point first = 5 last = 4 6 is NOT Found. But can be added at position 5 Num comparison = 3 ----- Iteration 5: Inserting item 6 which is in position 5 to new position 5 0 1 2 3 4 5 1 2 3 4 5 6 ----- BinaryInsertion sort ----- #size(n) = 6 #comparison(c) = 11 (Insertion sort 5) #swap(s) = 0 #c+s = 11 #T(n) = (c+s)/(n) = 1.83333(n) #T(n) = (c+s)/(n*log2n) = 0.70923(n *log2n) #T(n) = (c+s)/(n^2) = 0.305556(n^2) </pre>

Figure 9.21: Measuring binary insertion sort on various small arrays

9.12. ARRAY BASED BINARY INSERTION SORT

binary insertion sort on n random numbers	
<pre>#size(n) = 1000 #comparison(c) = 8547 #swap(s) = 256953 #c+s = 265500 #T(n) = (c+s)/(n) = 265.5(n) #T(n) = (c+s)/(n*log2n) = 26.6412(n *log2n) #T(n) = (c+s)/(n^2) = 0.2655(n^2)</pre>	<pre>#size(n) = 2000 #comparison(c) = 19026 #swap(s) = 994534 #c+s = 1013560 #T(n) = (c+s)/(n) = 506.78(n) #T(n) = (c+s)/(n*log2n) = 46.2147(n *log2n) #T(n) = (c+s)/(n^2) = 0.25339(n^2)</pre>
<pre>#size(n) = 3000 #comparison(c) = 30014 #swap(s) = 2268911 #c+s = 2298925 #T(n) = (c+s)/(n) = 766.308(n) #T(n) = (c+s)/(n*log2n) = 66.3428(n *log2n) #T(n) = (c+s)/(n^2) = 0.255436(n^2)</pre>	<pre>#size(n) = 4000 #comparison(c) = 41506 #swap(s) = 3996873 #c+s = 4038379 #T(n) = (c+s)/(n) = 1009.59(n) #T(n) = (c+s)/(n*log2n) = 84.3735(n *log2n) #T(n) = (c+s)/(n^2) = 0.252399(n^2)</pre>
<p>Sorting 1000 numbers already sorted in ascending order to ascending order</p> <p>Running BinaryInsertion sort ----- BinaryInsertion sort -----</p> <pre>#size(n) = 1000 #comparison(c) = 8977 #swap(s) = 0 #c+s = 8977 #T(n) = (c+s)/(n) = 8.977(n) #T(n) = (c+s)/(n*log2n) = 0.900782(n *log2n) #T(n) = (c+s)/(n^2) = 0.008977(n^2)</pre> <p style="text-align: right;">BEST CASE</p>	
<p>Sorting 1000 numbers already sorted in ascending order to descending order</p> <p>Running BinaryInsertion sort ----- BinaryInsertion sort -----</p> <pre>#size(n) = 1000 #comparison(c) = 7987 #swap(s) = 499500 #c+s = 507487 #T(n) = (c+s)/(n) = 507.487(n) #T(n) = (c+s)/(n*log2n) = 50.9229(n *log2n) #T(n) = (c+s)/(n^2) = 0.507487(n^2)</pre> <p style="text-align: right;">WORST CASE</p>	

Figure 9.22: Measuring binary insertion sort on various big arrays

9.13 Linked List based binary insertion sort

9.14 Array based merge sort

9.14. ARRAY BASED MERGE SORT

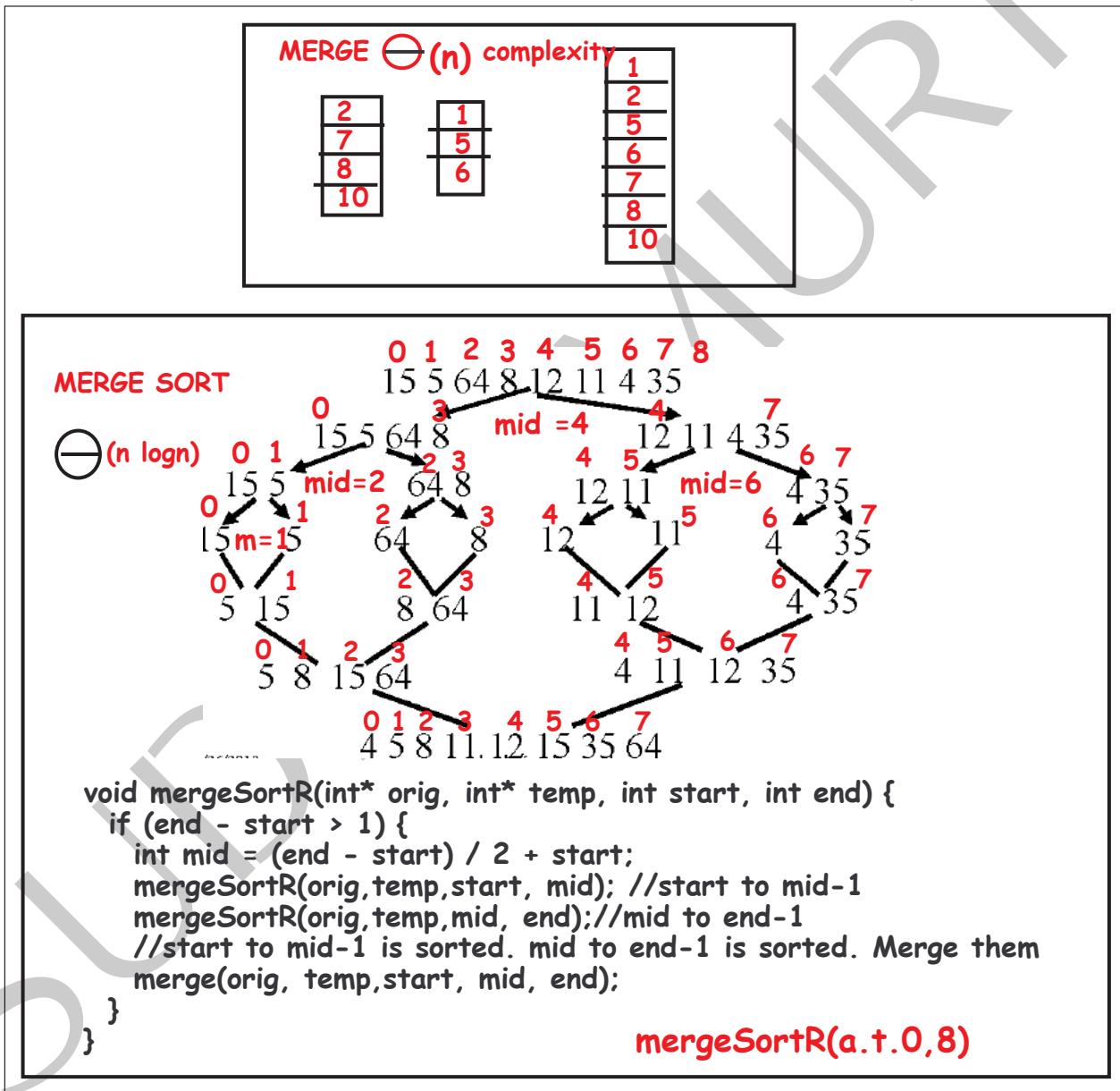


Figure 9.23: Merge and merge sort

CHAPTER 9. SEARCHING AND SORTING

```

Running merge sort
0 1 2 3 4 5 6 7
15 5 64 8 12 11 4 35
-----  

In divide: s = 0 m = 4 e = 8
Num element = 8 Dividing
In divide: s = 0 m = 2 e = 4
Num element = 4 Dividing
In divide: s = 0 m = 1 e = 2
Num element = 2 Dividing
In divide: s = 0 m = 0 e = 1
Num element = 1 NO work
In divide: s = 1 m = 1 e = 2
Num element = 1 NO work
-----  

In merge: s = 0 m = 1 e = 2
Array 1: {0,1} of size 1
Array 2: {1,2} of size 1
0 1 2 3 4 5 6 7
15 5 64 8 12 11 4 35
+++++++
0 1 2 3 4 5 6 7
5 15 64 8 12 11 4 35
-----  

In divide: s = 2 m = 3 e = 4
Num element = 2 Dividing
In divide: s = 2 m = 2 e = 3
Num element = 1 NO work
In divide: s = 3 m = 3 e = 4
Num element = 1 NO work
-----  

In merge: s = 2 m = 3 e = 4
Array 1: {2,3} of size 1
Array 2: {3,4} of size 1
0 1 2 3 4 5 6 7
5 15 64 8 12 11 4 35
+++++++
0 1 2 3 4 5 6 7
5 15 64 8 12 11 4 35
-----  

In merge: s = 0 m = 2 e = 4
Array 1: {0,2} of size 2
Array 2: {2,4} of size 2
0 1 2 3 4 5 6 7
5 15 8 64 12 11 4 35
+++++++
0 1 2 3 4 5 6 7
5 8 15 64 12 11 4 35
-----  

In merge: s = 4 m = 6 e = 8
Array 1: {4,6} of size 2
Array 2: {6,8} of size 2
0 1 2 3 4 5 6 7
5 8 15 64 11 12 4 35
+++++++
0 1 2 3 4 5 6 7
5 8 15 64 4 11 12 35
-----  

In merge: s = 0 m = 4 e = 8
Array 1: {0,4} of size 4
Array 2: {4,8} of size 4
0 1 2 3 4 5 6 7
5 8 15 64 4 11 12 35
+++++++
0 1 2 3 4 5 6 7
4 5 8 11 12 15 35 64
-----  

#size(n) = 8
#comparison(c) = 17
#swap(s) = 48
#Num recursion(r) = 14
#c+s = 65
#T(n) = (c+s)/(n) = 8.125(n)
#T(n) = (c+s)/(n*log2n) = 2.70833(n *log2n)
#T(n) = (c+s)/(n^2) = 1.01563(n^2)
-----  

In divide: s = 6 m = 7 e = 8
Num element = 2 Dividing
In divide: s = 6 m = 6 e = 7
Num element = 1 NO work
In divide: s = 7 m = 7 e = 8
Num element = 1 NO work
-----  

In merge: s = 6 m = 7 e = 8
Array 1: {6,7} of size 1
Array 2: {7,8} of size 1
0 1 2 3 4 5 6 7
5 8 15 64 11 12 4 35
+++++++
0 1 2 3 4 5 6 7
5 8 15 64 11 12 4 35

```

Figure 9.24: Merge sort in action

9.14. ARRAY BASED MERGE SORT

```

template <typename T>
class mergesort: public dsort<T> {           mergesort.h
public:
    mergesort(darray<T>& d,
              int s, int(*cf)(const T& c1, const T& c2),
              const char* t,
              bool dis = false);
    ~mergesort(){}
    virtual void sorting_algorithm();
    void merge_sort();

private:
    void _merge_sort();
    void _merge_sort_r(int start, int end);
    void _merge(int start,int mid, int end);
};

template <typename T>
void mergesort<T>::_merge(int start,int mid, int end){          mergesort.cpp
    //WRITE CODE HERE
}

template <typename T>
void mergesort<T>::_merge_sort_r(int start, int end){
    int mid = (end - start) / 2 + start;
    bool d = dsort<T>::display();
    if (d) {
        cout << "In divide: s = " << start << " m = " << mid << " e = " << end << endl;
        int h = end - start;
        if (h > 1) {
            cout << " Num element = " << h << " Dividing" << endl;
        } else {
            cout << " Num element = " << h << " NO work" << endl;
        }
    }
    if (end - start > 1){ //More than 1 element
        //WRITE CODE HERE
    }
}

template <typename T>
void mergesort<T>::sorting_algorithm(){
    dsort<T>::reset_stat();
    int n = dsort<T>::size();
    if (n) {
        _merge_sort();
    } else {
        cout << "Zero sized array\n";
    }
    dsort<T>::_assertSorted();
}

template <typename T>
void mergesort<T>::_merge_sort(){          mergesort.cpp
    int n = dsort<T>::size();
    bool d = dsort<T>::display();
    if (d) {
        dsort<T>::_print_darray();
    }
    _merge_sort_r(0, n);
}

template <typename T>
void mergesort<T>::_merge_sort_r(){          mergesort.cpp
    sorting_algorithm();
}

```

Figure 9.25: Merge sort code

9.14.1 Measuring merge sort

9.14. ARRAY BASED MERGE SORT

merge sort on n random numbers	
<pre>#size(n) = 1000 #comparison(c) = 8714 #swap(s) = 19952 #Num recursion(r) = 1998 #c+s = 28666 #T(n) = (c+s)/(n) = 28.666(n) #T(n) = (c+s)/(n*log2n) = 2.87644(n *log2n) #T(n) = (c+s)/(n^2) = 0.028666(n^2)</pre>	<pre>#size(n) = 2000 #comparison(c) = 19427 #swap(s) = 43904 #Num recursion(r) = 3998 #c+s = 63331 #T(n) = (c+s)/(n) = 31.6655(n) #T(n) = (c+s)/(n*log2n) = 2.88766(n *log2n) #T(n) = (c+s)/(n^2) = 0.0158328(n^2)</pre>
<pre>#size(n) = 3000 #comparison(c) = 30972 #swap(s) = 69808 #Num recursion(r) = 5998 #c+s = 100780 #T(n) = (c+s)/(n) = 33.5933(n) #T(n) = (c+s)/(n*log2n) = 2.90833(n *log2n) #T(n) = (c+s)/(n^2) = 0.0111978(n^2)</pre>	<pre>#size(n) = 4000 #comparison(c) = 42861 #swap(s) = 95808 #Num recursion(r) = 7998 #c+s = 138669 #T(n) = (c+s)/(n) = 34.6673(n) #T(n) = (c+s)/(n*log2n) = 2.8972(n *log2n) #T(n) = (c+s)/(n^2) = 0.00866681(n^2)</pre>
<p>Sorting 1000 numbers already sorted in ascending order to ascending order Running merge sort</p> <pre>#size(n) = 1000 #comparison(c) = 4932 #swap(s) = 19952 #Num recursion(r) = 1998 #c+s = 24884 #T(n) = (c+s)/(n) = 24.884(n) #T(n) = (c+s)/(n*log2n) = 2.49694(n *log2n) #T(n) = (c+s)/(n^2) = 0.024884(n^2)</pre>	
<p>Sorting 1000 numbers already sorted in ascending order to descending order Running merge sort</p> <pre>----- merge sort ----- #size(n) = 1000 #comparison(c) = 5044 #swap(s) = 19952 #Num recursion(r) = 1998 #c+s = 24996 #T(n) = (c+s)/(n) = 24.996(n) #T(n) = (c+s)/(n*log2n) = 2.50818(n *log2n) #T(n) = (c+s)/(n^2) = 0.024996(n^2)</pre>	

Figure 9.26: Measuring merge sort on various big arrays

9.14.2 Complexity of merge sort

9.14. ARRAY BASED MERGE SORT

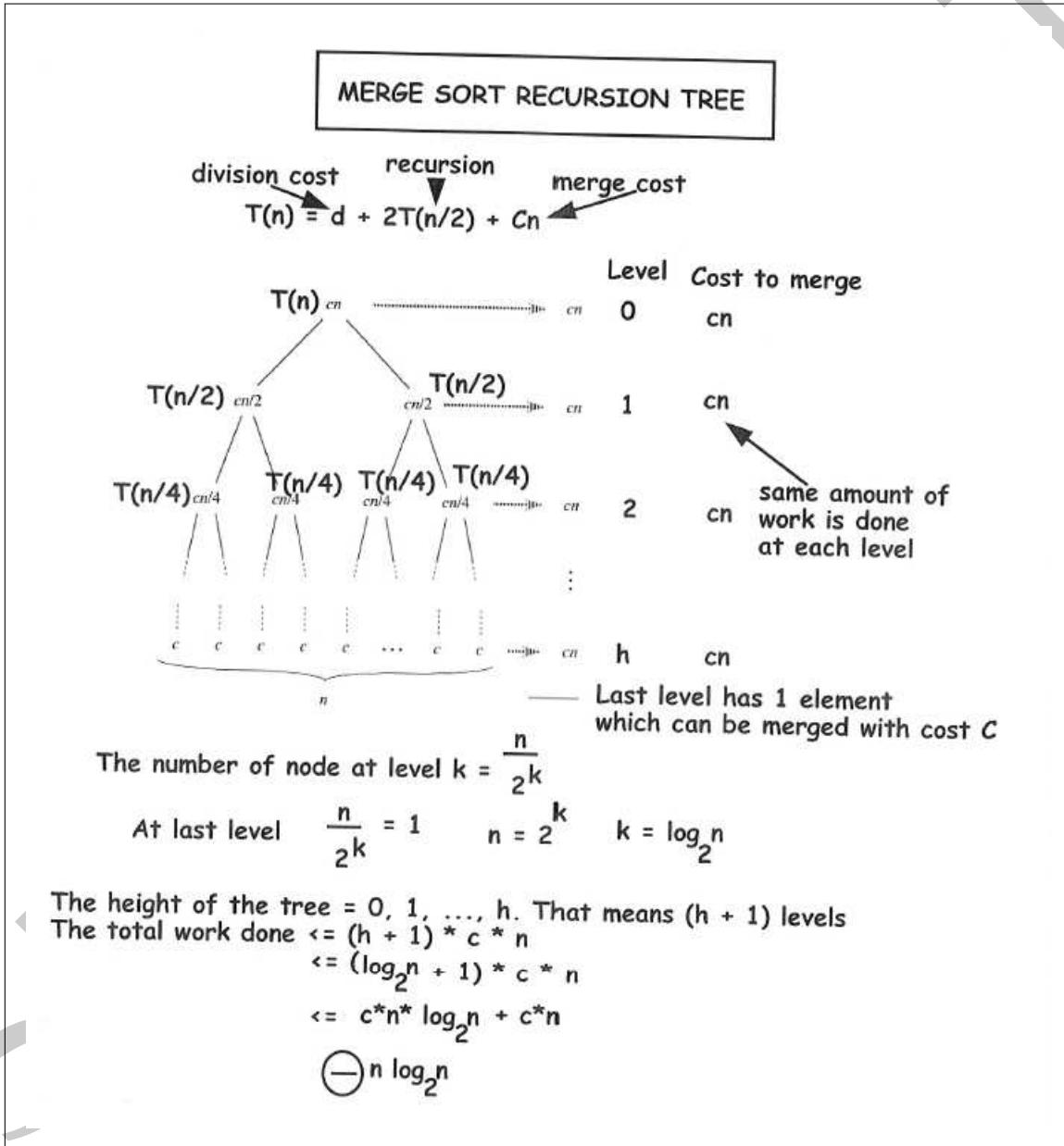


Figure 9.27: Complexity of merge sort

What happens if you can invent merge routine of $O(1)$ complexity?

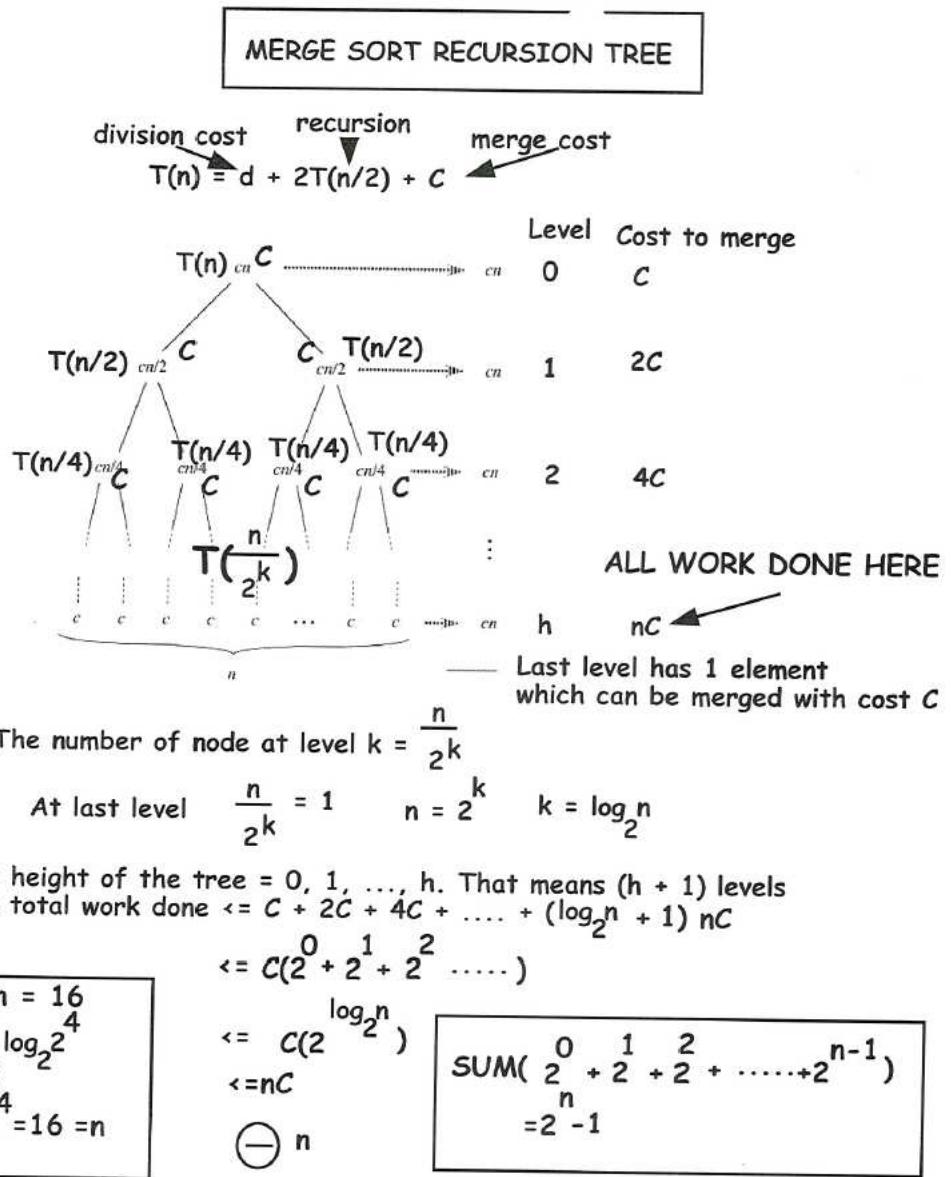


Figure 9.28: Complexity of merge sort if we invent $O(1)$ merge routine

9.14. ARRAY BASED MERGE SORT

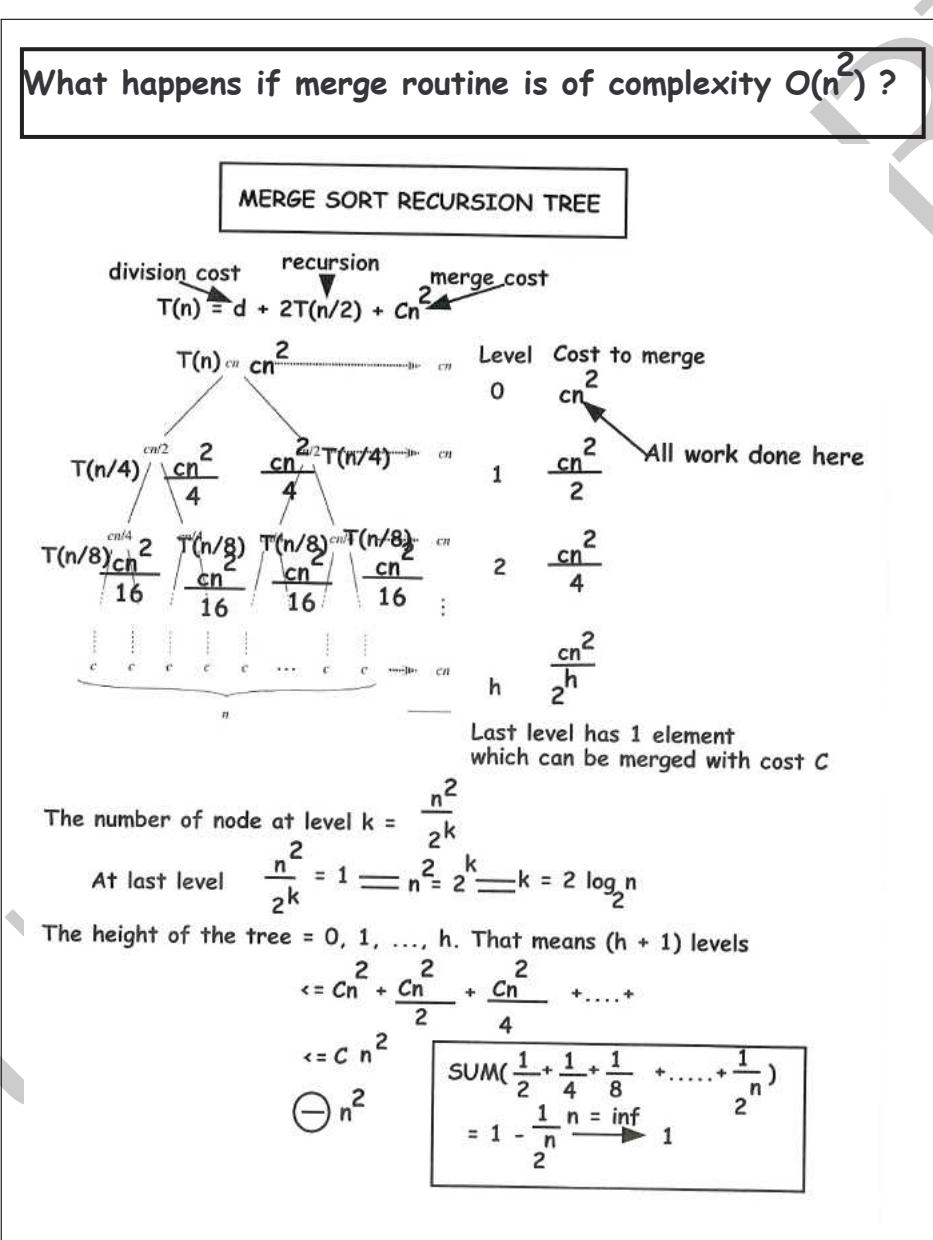


Figure 9.29: Complexity of merge sort if we use $O(n^2)$ merge routine

9.15 Linked list based merge sort

9.16 Array based quick sort

9.16. ARRAY BASED QUICK SORT

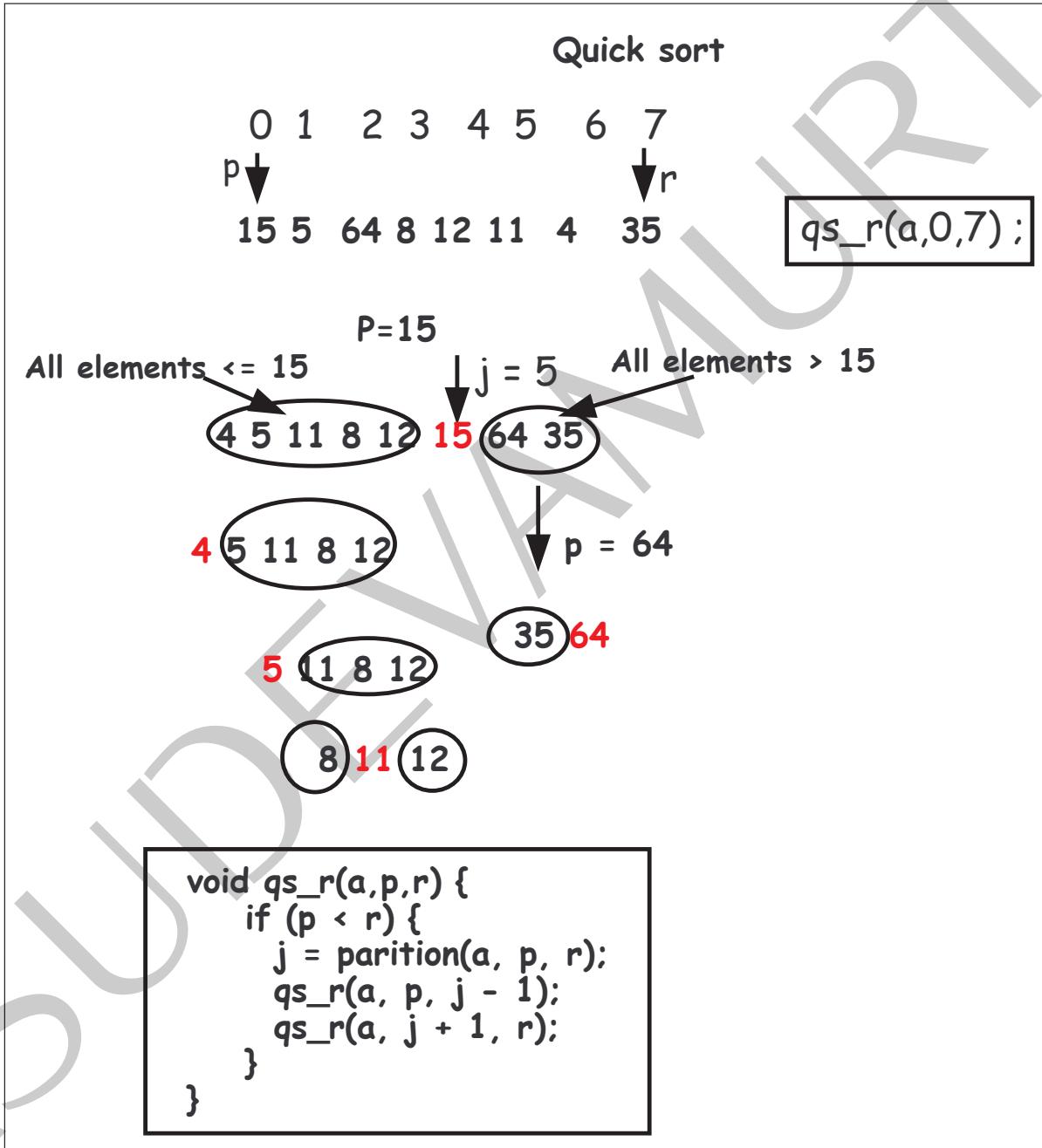


Figure 9.30: Partition and quick sort

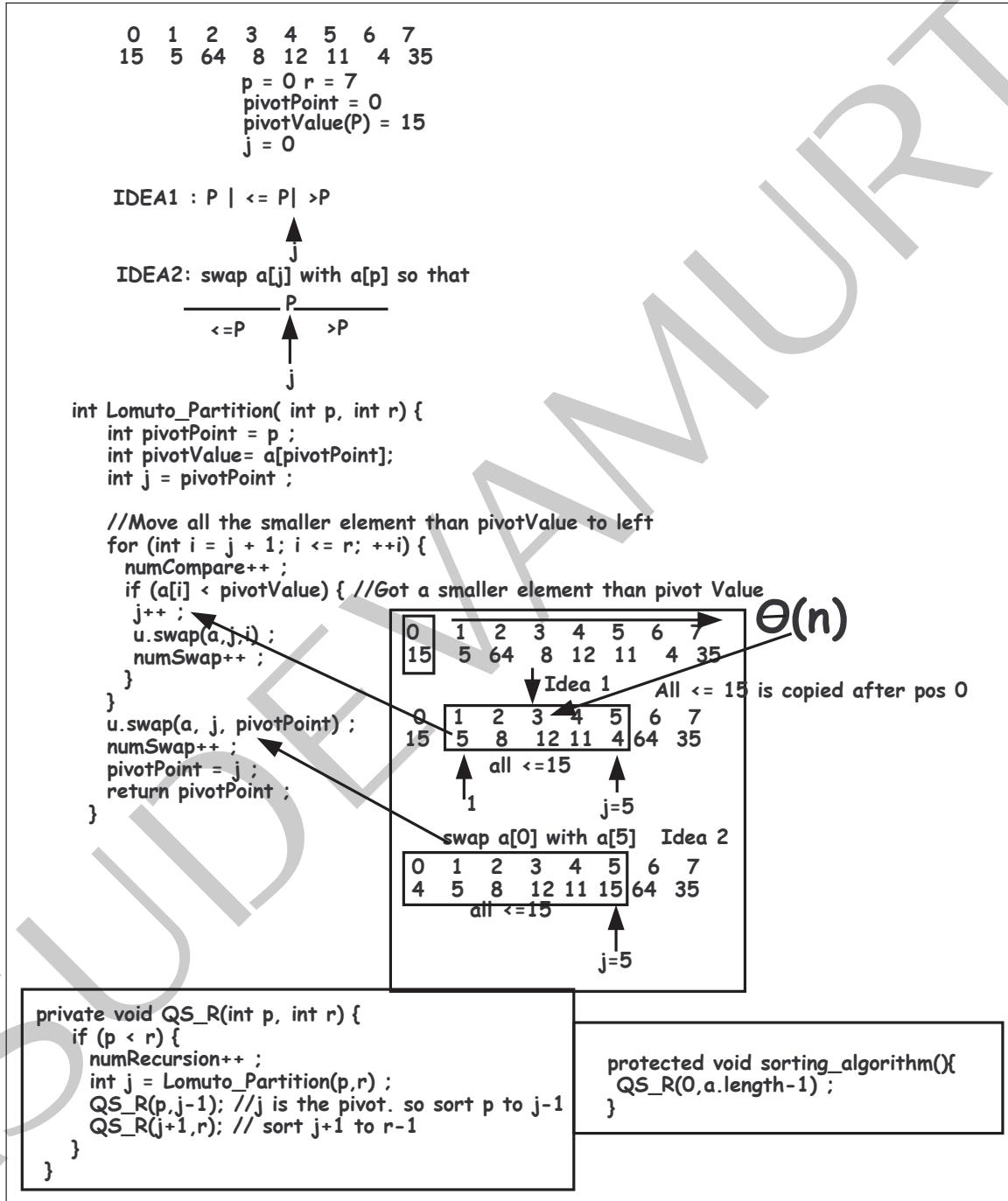


Figure 9.31: Lomuto-Partition and quick sort

9.16. ARRAY BASED QUICK SORT

Average case	Worst case	Worst case																																																																																																																																																																																																																																																																																																																																																																																																																								
<p>Sort from 0 to 7 using 15 as pivot Value</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr> <tr><td>15</td><td>5</td><td>64</td><td>8</td><td>12</td><td>11</td><td>4</td><td>35</td></tr> </table> <p>Pivot Value = 15. Pivot position = 5</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr> <tr><td>4</td><td>5</td><td>8</td><td>12</td><td>11</td><td>15</td><td>64</td><td>35</td></tr> </table> <p>Sort from 0 to 4 using 4 as pivot Value</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>4</td><td>5</td><td>8</td><td>12</td><td>11</td></tr> </table> <p>Pivot Value = 4. Pivot position = 0</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr> <tr><td>4</td><td>5</td><td>8</td><td>12</td><td>11</td><td>15</td><td>64</td><td>35</td></tr> </table> <p>Sort from 1 to 4 using 5 as pivot Value</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>4</td><td>5</td><td>8</td><td>12</td><td>11</td></tr> </table> <p>Pivot Value = 5. Pivot position = 1</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr> <tr><td>4</td><td>5</td><td>8</td><td>12</td><td>11</td><td>15</td><td>64</td><td>35</td></tr> </table> <p>Sort from 2 to 4 using 8 as pivot Value</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>4</td><td>5</td><td>8</td><td>12</td><td>11</td></tr> </table> <p>Pivot Value = 8. Pivot position = 2</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr> <tr><td>4</td><td>5</td><td>8</td><td>12</td><td>11</td><td>15</td><td>64</td><td>35</td></tr> </table> <p>Sort from 3 to 4 using 12 as pivot Value</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>4</td><td>5</td><td>8</td><td>12</td><td>11</td></tr> </table> <p>Pivot Value = 12. Pivot position = 4</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr> <tr><td>4</td><td>5</td><td>8</td><td>11</td><td>12</td><td>15</td><td>64</td><td>35</td></tr> </table> <p>Sort from 6 to 7 using 64 as pivot Value</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr> <tr><td>4</td><td>5</td><td>8</td><td>11</td><td>12</td><td>15</td><td>64</td><td>35</td></tr> </table> <p>Pivot Value = 64. Pivot position = 7</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr> <tr><td>4</td><td>5</td><td>8</td><td>11</td><td>12</td><td>15</td><td>35</td><td>64</td></tr> </table> <p>----- Quick sort -----</p> <pre>#size(n) = 6 #comparison(c) = 15 #swap(s) = 14 #Num recursion(r) = 5 #c+s = 29 #T(n) = (c+s)/(n) = 4.83333(n) #T(n) = (c+s)/(n*log2n) = 1.86979(n*log2n) #T(n) = (c+s)/(n^2) = 0.805556(n^2)</pre> <p>----- Quick sort -----</p> <pre>#size(n) = 8 #comparison(c) = 18 #swap(s) = 13 #Num recursion(r) = 6 #c+s = 31 #T(n) = (c+s)/(n) = 3.875(n) #T(n) = (c+s)/(n*log2n) = 1.29167(n *log2n) #T(n) = (c+s)/(n^2) = 0.484375(n^2)</pre>	0	1	2	3	4	5	6	7	15	5	64	8	12	11	4	35	0	1	2	3	4	5	6	7	4	5	8	12	11	15	64	35	0	1	2	3	4	4	5	8	12	11	0	1	2	3	4	5	6	7	4	5	8	12	11	15	64	35	0	1	2	3	4	4	5	8	12	11	0	1	2	3	4	5	6	7	4	5	8	12	11	15	64	35	0	1	2	3	4	4	5	8	12	11	0	1	2	3	4	5	6	7	4	5	8	12	11	15	64	35	0	1	2	3	4	4	5	8	12	11	0	1	2	3	4	5	6	7	4	5	8	11	12	15	64	35	0	1	2	3	4	5	6	7	4	5	8	11	12	15	64	35	0	1	2	3	4	5	6	7	4	5	8	11	12	15	35	64	<p>Sort from 0 to 5 using 6 as pivot Value</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td></tr> </table> <p>Pivot Value = 6. Pivot position = 5</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>5</td><td>4</td><td>3</td><td>2</td><td>6</td></tr> </table> <p>Sort from 0 to 4 using 1 as pivot Value</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>5</td><td>4</td><td>3</td><td>2</td><td>6</td></tr> </table> <p>Pivot Value = 1. Pivot position = 0</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table> <p>Sort from 1 to 4 using 5 as pivot Value</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>5</td><td>4</td><td>3</td><td>2</td><td>6</td></tr> </table> <p>Pivot Value = 5. Pivot position = 4</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>4</td><td>3</td><td>5</td><td>6</td></tr> </table> <p>Sort from 1 to 3 using 2 as pivot Value</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>4</td><td>3</td><td>5</td><td>6</td></tr> </table> <p>Pivot Value = 2. Pivot position = 1</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>4</td><td>3</td><td>5</td><td>6</td></tr> </table> <p>Sort from 2 to 3 using 4 as pivot Value</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>4</td><td>3</td><td>5</td><td>6</td></tr> </table> <p>Pivot Value = 4. Pivot position = 3</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table> <p>----- Quick sort -----</p> <pre>#size(n) = 6 #comparison(c) = 15 #swap(s) = 5 #Num recursion(r) = 5 #c+s = 20 #T(n) = (c+s)/(n) = 3.33333(n) #T(n) = (c+s)/(n*log2n) = 1.28951(n *log2n) #T(n) = (c+s)/(n^2) = 0.555556(n^2)</pre>	0	1	2	3	4	5	6	5	4	3	2	1	0	1	2	3	4	5	1	5	4	3	2	6	0	1	2	3	4	5	1	5	4	3	2	6	0	1	2	3	4	5	1	2	3	4	5	6	0	1	2	3	4	5	1	5	4	3	2	6	0	1	2	3	4	5	1	2	4	3	5	6	0	1	2	3	4	5	1	2	4	3	5	6	0	1	2	3	4	5	1	2	4	3	5	6	0	1	2	3	4	5	1	2	4	3	5	6	0	1	2	3	4	5	1	2	3	4	5	6	<p>Running Quick sort</p> <p>Sort from 0 to 5 using 1 as pivot value</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table> <p>Pivot Value = 1. Pivot position = 0</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table> <p>Sort from 1 to 5 using 2 as pivot value</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table> <p>Pivot Value = 2. Pivot position = 1</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table> <p>Sort from 2 to 5 using 3 as pivot value</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table> <p>Pivot Value = 3. Pivot position = 2</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table> <p>Sort from 3 to 5 using 4 as pivot value</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table> <p>Pivot Value = 4. Pivot position = 3</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table> <p>Sort from 4 to 5 using 5 as pivot value</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table> <p>Pivot Value = 5. Pivot position = 4</p> <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table> <p>----- Quick sort -----</p> <pre>#size(n) = 6 #comparison(c) = 15 #swap(s) = 5 #Num recursion(r) = 5 #c+s = 20 #T(n) = (c+s)/(n) = 3.33333(n) #T(n) = (c+s)/(n*log2n) = 1.28951(n *log2n) #T(n) = (c+s)/(n^2) = 0.555556(n^2)</pre>	0	1	2	3	4	5	1	2	3	4	5	6	0	1	2	3	4	5	1	2	3	4	5	6	0	1	2	3	4	5	1	2	3	4	5	6	0	1	2	3	4	5	1	2	3	4	5	6	0	1	2	3	4	5	1	2	3	4	5	6	0	1	2	3	4	5	1	2	3	4	5	6	0	1	2	3	4	5	1	2	3	4	5	6	0	1	2	3	4	5	1	2	3	4	5	6	0	1	2	3	4	5	1	2	3	4	5	6	0	1	2	3	4	5	1	2	3	4	5	6
0	1	2	3	4	5	6	7																																																																																																																																																																																																																																																																																																																																																																																																																			
15	5	64	8	12	11	4	35																																																																																																																																																																																																																																																																																																																																																																																																																			
0	1	2	3	4	5	6	7																																																																																																																																																																																																																																																																																																																																																																																																																			
4	5	8	12	11	15	64	35																																																																																																																																																																																																																																																																																																																																																																																																																			
0	1	2	3	4																																																																																																																																																																																																																																																																																																																																																																																																																						
4	5	8	12	11																																																																																																																																																																																																																																																																																																																																																																																																																						
0	1	2	3	4	5	6	7																																																																																																																																																																																																																																																																																																																																																																																																																			
4	5	8	12	11	15	64	35																																																																																																																																																																																																																																																																																																																																																																																																																			
0	1	2	3	4																																																																																																																																																																																																																																																																																																																																																																																																																						
4	5	8	12	11																																																																																																																																																																																																																																																																																																																																																																																																																						
0	1	2	3	4	5	6	7																																																																																																																																																																																																																																																																																																																																																																																																																			
4	5	8	12	11	15	64	35																																																																																																																																																																																																																																																																																																																																																																																																																			
0	1	2	3	4																																																																																																																																																																																																																																																																																																																																																																																																																						
4	5	8	12	11																																																																																																																																																																																																																																																																																																																																																																																																																						
0	1	2	3	4	5	6	7																																																																																																																																																																																																																																																																																																																																																																																																																			
4	5	8	12	11	15	64	35																																																																																																																																																																																																																																																																																																																																																																																																																			
0	1	2	3	4																																																																																																																																																																																																																																																																																																																																																																																																																						
4	5	8	12	11																																																																																																																																																																																																																																																																																																																																																																																																																						
0	1	2	3	4	5	6	7																																																																																																																																																																																																																																																																																																																																																																																																																			
4	5	8	11	12	15	64	35																																																																																																																																																																																																																																																																																																																																																																																																																			
0	1	2	3	4	5	6	7																																																																																																																																																																																																																																																																																																																																																																																																																			
4	5	8	11	12	15	64	35																																																																																																																																																																																																																																																																																																																																																																																																																			
0	1	2	3	4	5	6	7																																																																																																																																																																																																																																																																																																																																																																																																																			
4	5	8	11	12	15	35	64																																																																																																																																																																																																																																																																																																																																																																																																																			
0	1	2	3	4	5																																																																																																																																																																																																																																																																																																																																																																																																																					
6	5	4	3	2	1																																																																																																																																																																																																																																																																																																																																																																																																																					
0	1	2	3	4	5																																																																																																																																																																																																																																																																																																																																																																																																																					
1	5	4	3	2	6																																																																																																																																																																																																																																																																																																																																																																																																																					
0	1	2	3	4	5																																																																																																																																																																																																																																																																																																																																																																																																																					
1	5	4	3	2	6																																																																																																																																																																																																																																																																																																																																																																																																																					
0	1	2	3	4	5																																																																																																																																																																																																																																																																																																																																																																																																																					
1	2	3	4	5	6																																																																																																																																																																																																																																																																																																																																																																																																																					
0	1	2	3	4	5																																																																																																																																																																																																																																																																																																																																																																																																																					
1	5	4	3	2	6																																																																																																																																																																																																																																																																																																																																																																																																																					
0	1	2	3	4	5																																																																																																																																																																																																																																																																																																																																																																																																																					
1	2	4	3	5	6																																																																																																																																																																																																																																																																																																																																																																																																																					
0	1	2	3	4	5																																																																																																																																																																																																																																																																																																																																																																																																																					
1	2	4	3	5	6																																																																																																																																																																																																																																																																																																																																																																																																																					
0	1	2	3	4	5																																																																																																																																																																																																																																																																																																																																																																																																																					
1	2	4	3	5	6																																																																																																																																																																																																																																																																																																																																																																																																																					
0	1	2	3	4	5																																																																																																																																																																																																																																																																																																																																																																																																																					
1	2	4	3	5	6																																																																																																																																																																																																																																																																																																																																																																																																																					
0	1	2	3	4	5																																																																																																																																																																																																																																																																																																																																																																																																																					
1	2	3	4	5	6																																																																																																																																																																																																																																																																																																																																																																																																																					
0	1	2	3	4	5																																																																																																																																																																																																																																																																																																																																																																																																																					
1	2	3	4	5	6																																																																																																																																																																																																																																																																																																																																																																																																																					
0	1	2	3	4	5																																																																																																																																																																																																																																																																																																																																																																																																																					
1	2	3	4	5	6																																																																																																																																																																																																																																																																																																																																																																																																																					
0	1	2	3	4	5																																																																																																																																																																																																																																																																																																																																																																																																																					
1	2	3	4	5	6																																																																																																																																																																																																																																																																																																																																																																																																																					
0	1	2	3	4	5																																																																																																																																																																																																																																																																																																																																																																																																																					
1	2	3	4	5	6																																																																																																																																																																																																																																																																																																																																																																																																																					
0	1	2	3	4	5																																																																																																																																																																																																																																																																																																																																																																																																																					
1	2	3	4	5	6																																																																																																																																																																																																																																																																																																																																																																																																																					
0	1	2	3	4	5																																																																																																																																																																																																																																																																																																																																																																																																																					
1	2	3	4	5	6																																																																																																																																																																																																																																																																																																																																																																																																																					
0	1	2	3	4	5																																																																																																																																																																																																																																																																																																																																																																																																																					
1	2	3	4	5	6																																																																																																																																																																																																																																																																																																																																																																																																																					
0	1	2	3	4	5																																																																																																																																																																																																																																																																																																																																																																																																																					
1	2	3	4	5	6																																																																																																																																																																																																																																																																																																																																																																																																																					
0	1	2	3	4	5																																																																																																																																																																																																																																																																																																																																																																																																																					
1	2	3	4	5	6																																																																																																																																																																																																																																																																																																																																																																																																																					
0	1	2	3	4	5																																																																																																																																																																																																																																																																																																																																																																																																																					
1	2	3	4	5	6																																																																																																																																																																																																																																																																																																																																																																																																																					

Figure 9.32: quick sort in action

9.16.1 Measuring quick sort

9.16. ARRAY BASED QUICK SORT

quick sort on n random numbers	
<pre>#size(n) = 1000 #comparison(c) = 10826 #swap(s) = 6561 #Num recursion(r) = 673 #c+s = 17387 #T(n) = (c+s)/(n) = 17.387(n) #T(n) = (c+s)/(n*log2n) = 1.74467(n *log2n) #T(n) = (c+s)/(n^2) = 0.017387(n^2)</pre>	<pre>#size(n) = 2000 #comparison(c) = 25175 #swap(s) = 13745 #Num recursion(r) = 1326 #c+s = 38920 #T(n) = (c+s)/(n) = 19.46(n) #T(n) = (c+s)/(n*log2n) = 1.77461(n *log2n) #T(n) = (c+s)/(n^2) = 0.00973(n^2)</pre>
<pre>#size(n) = 3000 #comparison(c) = 37149 #swap(s) = 20388 #Num recursion(r) = 2010 #c+s = 57537 #T(n) = (c+s)/(n) = 19.179(n) #T(n) = (c+s)/(n*log2n) = 1.66041(n *log2n) #T(n) = (c+s)/(n^2) = 0.006393(n^2)</pre>	<pre>#size(n) = 4000 #comparison(c) = 52715 #swap(s) = 28612 #Num recursion(r) = 2657 #c+s = 81327 #T(n) = (c+s)/(n) = 20.3318(n) #T(n) = (c+s)/(n*log2n) = 1.69916(n *log2n) #T(n) = (c+s)/(n^2) = 0.00508294(n^2)</pre>
<p>Sorting 1000 numbers already sorted in ascending order to ascending order</p> <p>Running Quick sort</p> <p>----- Quick sort -----</p> <pre>#size(n) = 1000 #comparison(c) = 499500 #swap(s) = 999 #Num recursion(r) = 999 #c+s = 500499 #T(n) = (c+s)/(n) = 500.499(n) #T(n) = (c+s)/(n*log2n) = 50.2217(n *log2n) #T(n) = (c+s)/(n^2) = 0.500499(n^2)</pre>	<p>WORST CASE</p>
<p>Sorting 1000 numbers already sorted in ascending order to descending order</p> <p>Running Quick sort</p> <p>----- Quick sort -----</p> <pre>#size(n) = 1000 #comparison(c) = 499500 #swap(s) = 250999 #Num recursion(r) = 999 #c+s = 750499 #T(n) = (c+s)/(n) = 750.499(n) #T(n) = (c+s)/(n*log2n) = 75.3076(n *log2n) #T(n) = (c+s)/(n^2) = 0.750499(n^2)</pre>	<p>WORST CASE</p>

Figure 9.33: Measuring quick sort on various big arrays

9.16.2 Complexity of quick sort

9.16. ARRAY BASED QUICK SORT

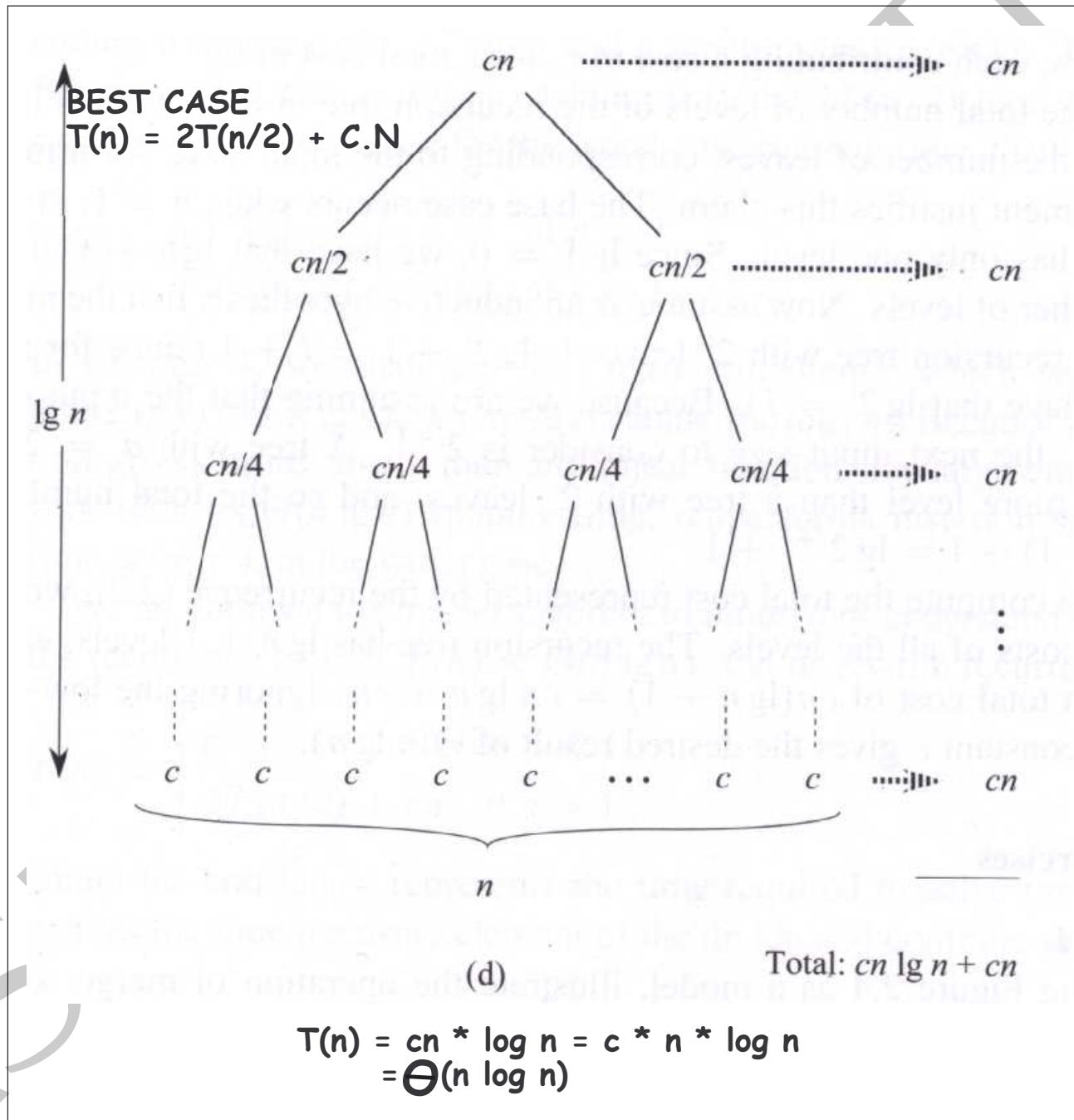


Figure 9.34: Best case complexity of quick sort

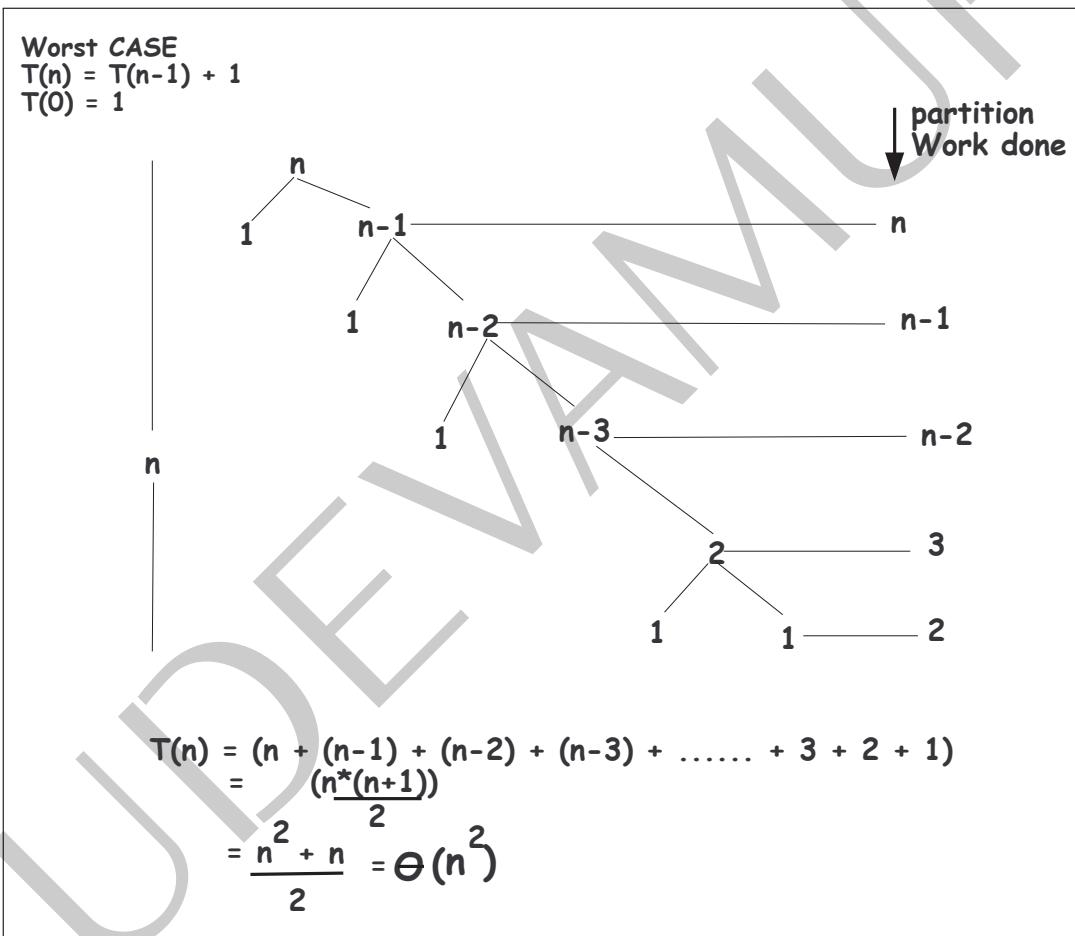


Figure 9.35: Worst case complexity of quick sort

9.16. ARRAY BASED QUICK SORT

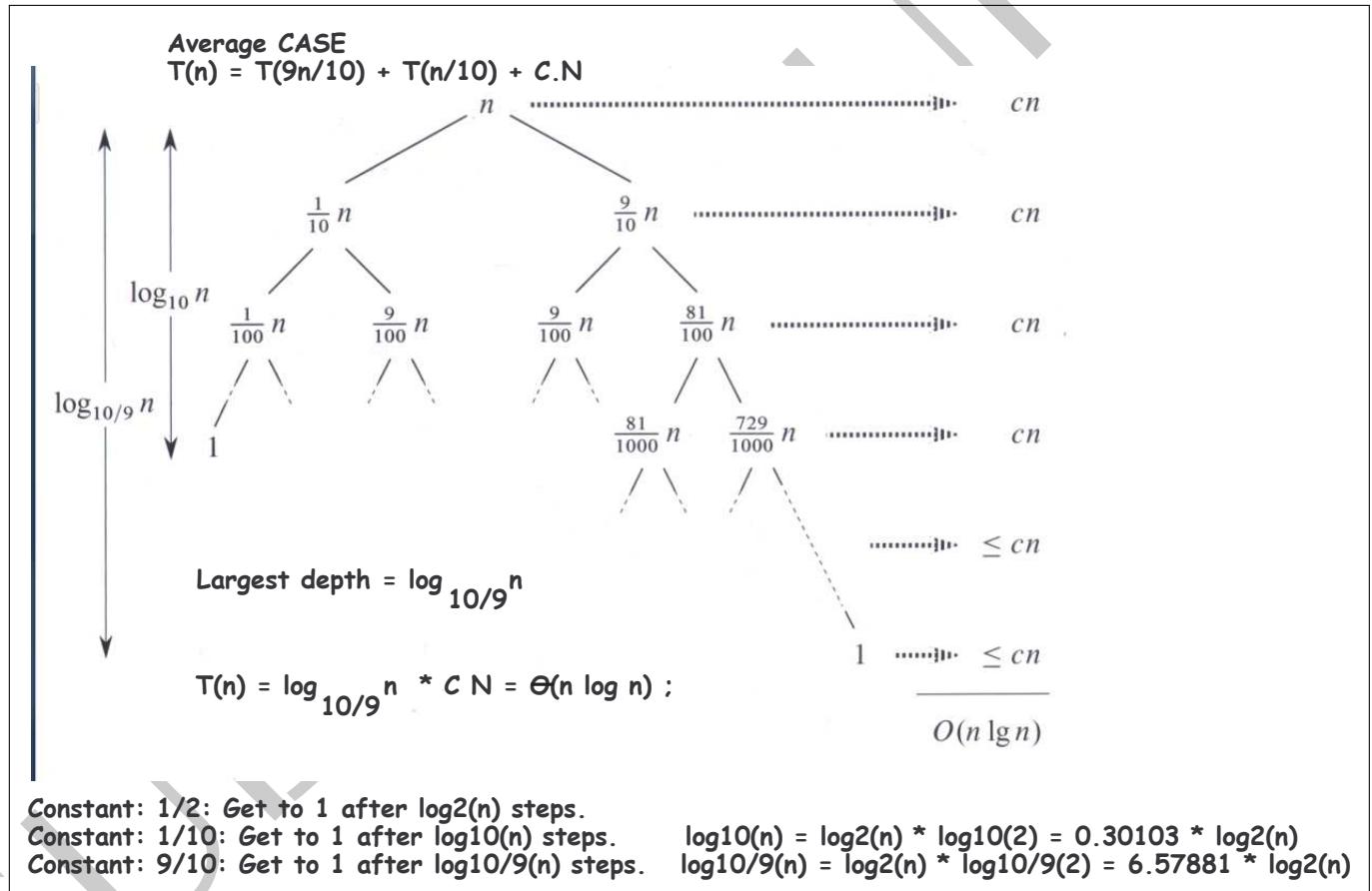


Figure 9.36: Expected complexity of quick sort

9.17 Linked list based quick sort

9.18 Quick select

9.18. QUICK SELECT

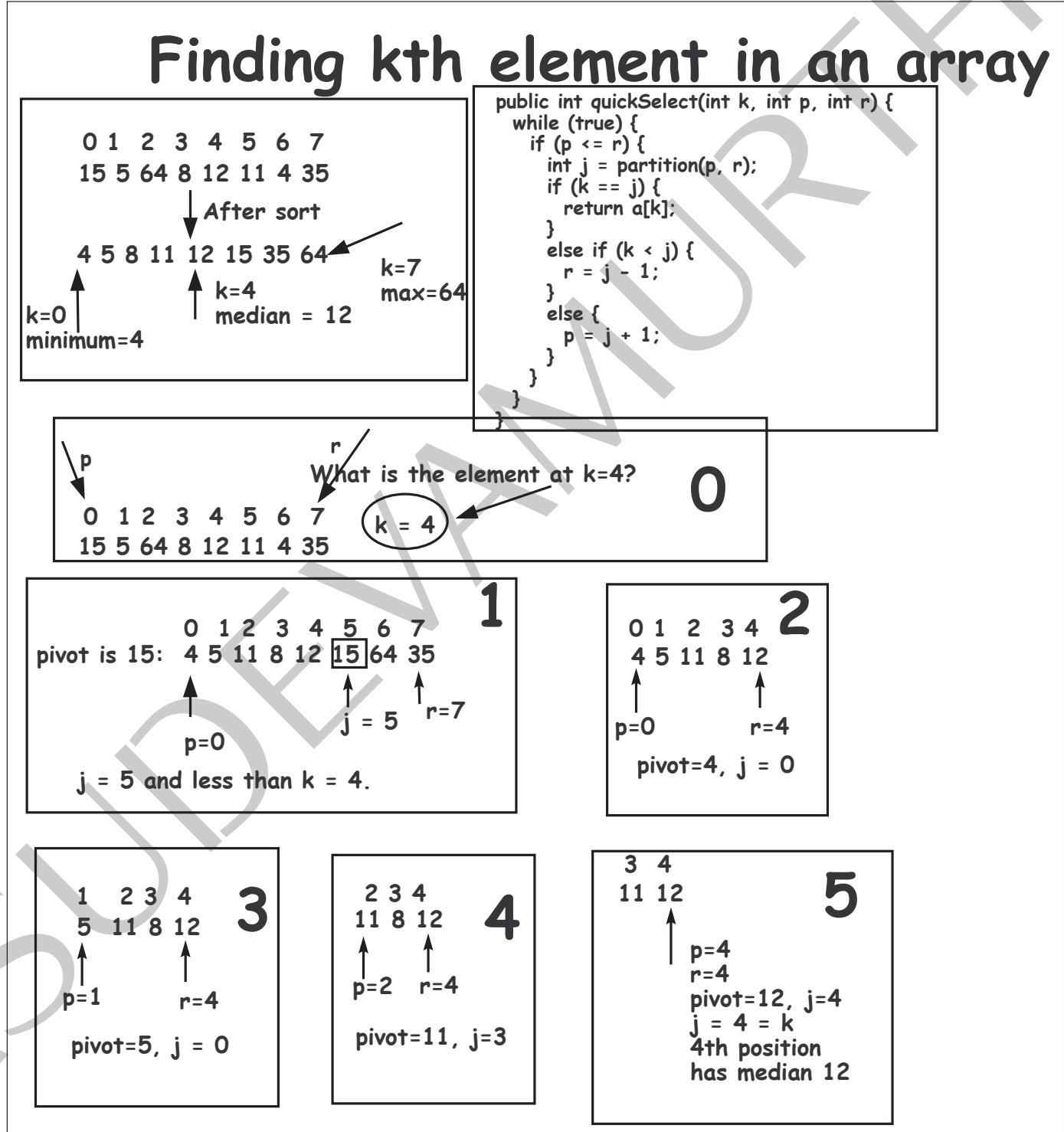


Figure 9.37: Finding k-th element using partition algorithm of quick sort

9.19 Counting Sort

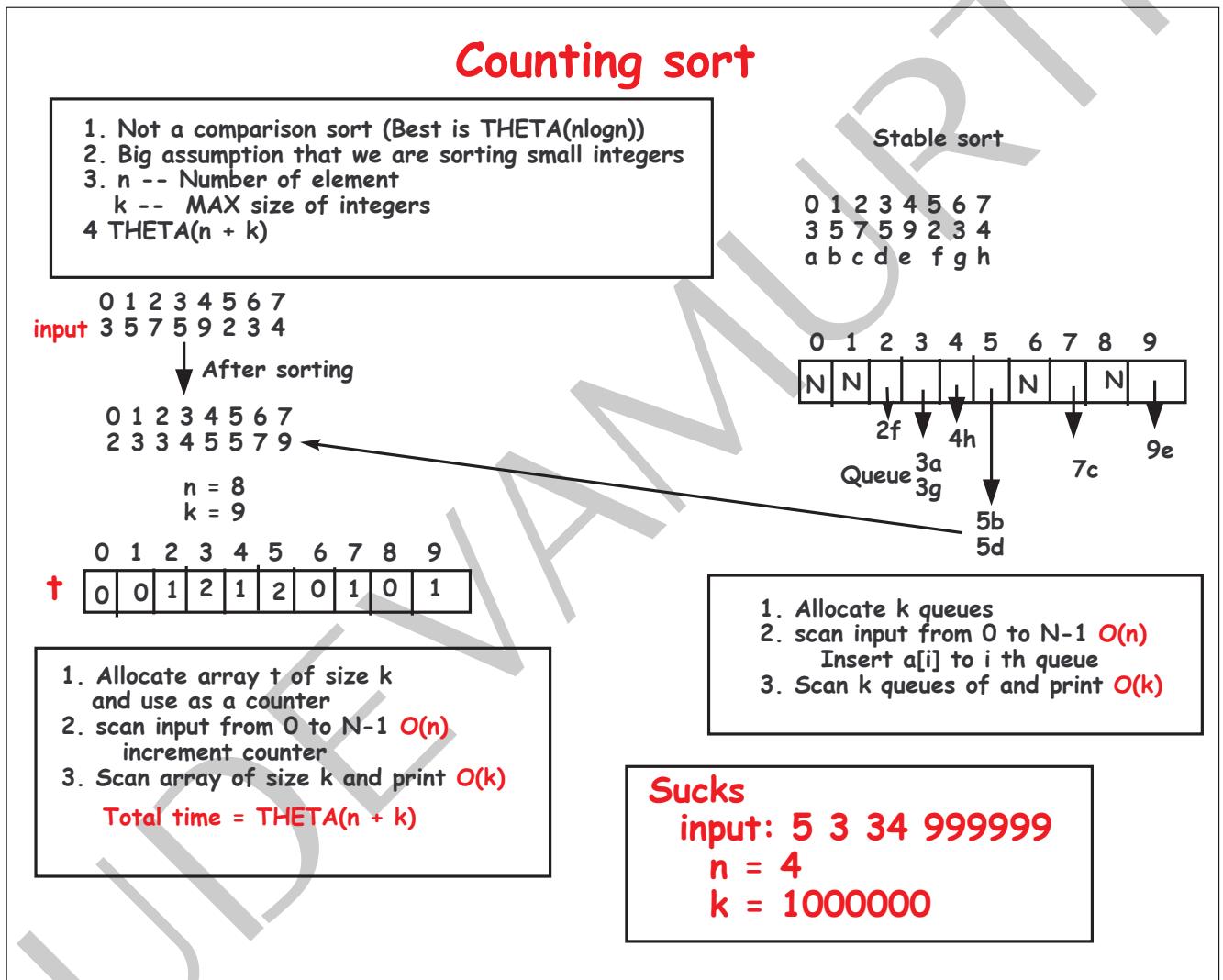


Figure 9.38: Counting sort and stable counting sort

9.20 Straight radix sort: from LSB to MSB

9.20. STRAIGHT RADIX SORT: FROM LSB TO MSB

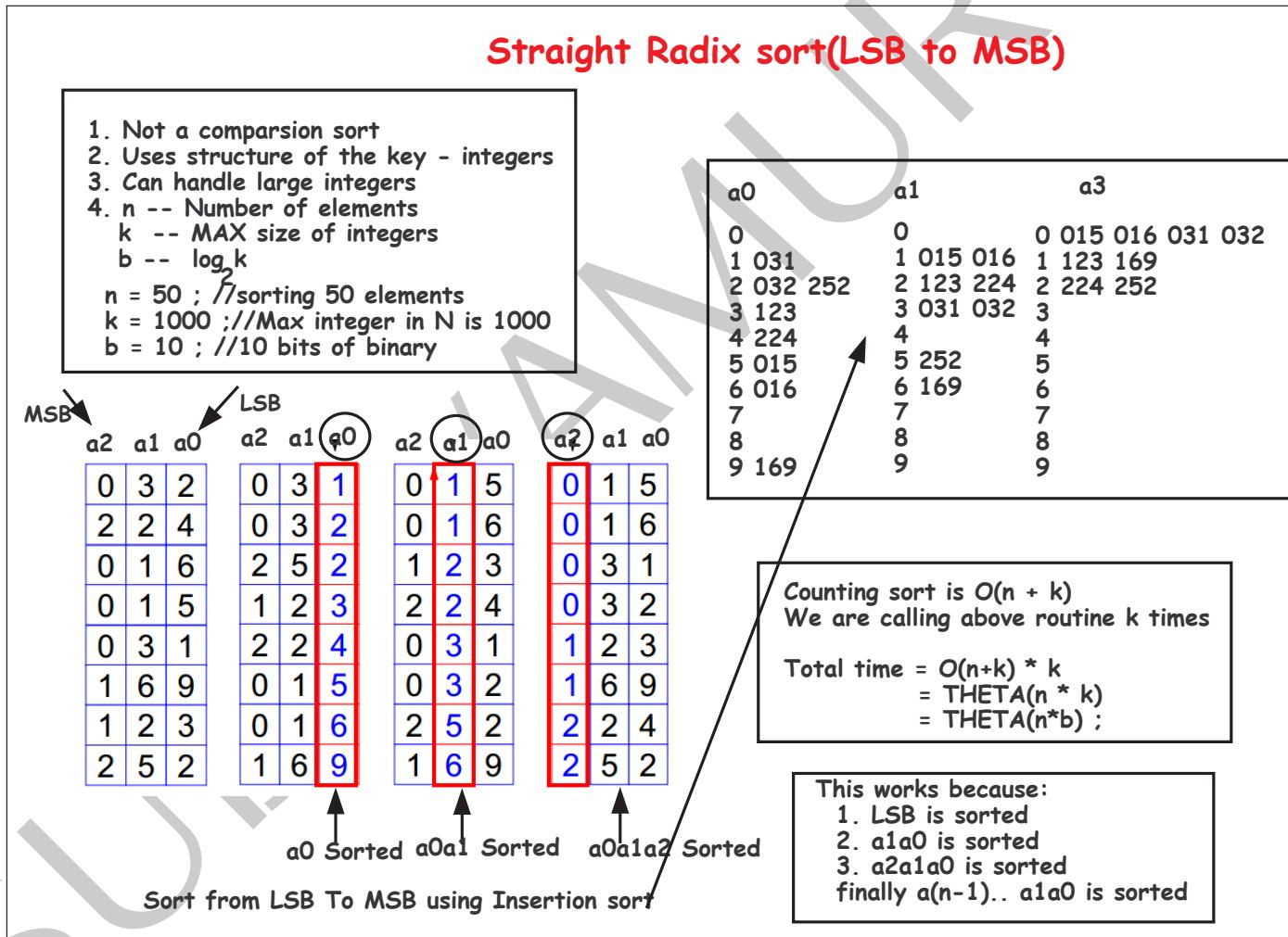
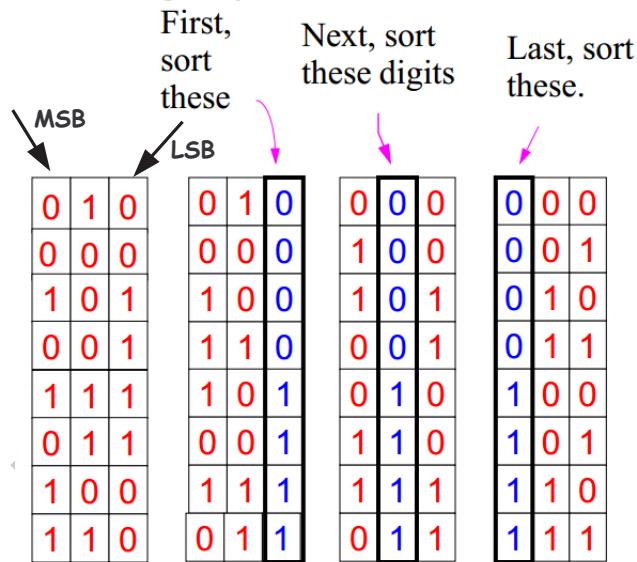


Figure 9.39: Straight radix sort, from LSB to MSB. Uses decimal number for illustration

Straight Radix Sort using Binary digits(LSB to MSB)

Examines bits from *right* to *left*

```
for k := 0 to b-1
    sort the array in a stable way,
    looking only at bit k
```



Note order of these bits after sort.

Figure 9.40: Straight radix sort, from LSB to MSB using binary representation of a number

9.20. STRAIGHT RADIX SORT: FROM LSB TO MSB

StraightRadixSort

```

private int numberOfBits() {
    int n = a.length ;
    if (n > 0) {
        int max = a[0] ;
        assert(max >= 0) ;
        for (int i = 1; i < n ; ++i) {
            assert(a[i] >= 0) ;
            if (a[i] > max) {
                max = a[i] ;
            }
        }
        if (max == 0) {
            return 1 ;
        }
        int b = 0 ;
        while (max != 0) {
            max = max/2 ;
            b++ ;
        }
        return b ; Computes max  
number of bits  
required
    }
    return 0 ;
}

```

```

private void rs1(int bit) {
    int n = a.length ;
    int mask = 1 << bit ;
    int [] zeroa = new int[n] ;
    int t0 = 0 ;
    int [] onea = new int[n] ;
    int t1 = 0 ;

    for (int i = 0 ; i < n; ++i) {
        numCompare++ ;
        numSwap++ ;
        if ((a[i] & mask) != 0) { //Looking for 1
            onea[t1++] = a[i] ;
        }else {
            zeroa[t0++] = a[i] ;
        }
    }
    int k = 0 ;
    for (int i = 0; i < t0; ++i) {
        numSwap++ ;
        a[k++] = zeroa[i] ;
    }
    for (int i = 0; i < t1; ++i) {
        numSwap++ ;
        a[k++] = onea[i] ;
    }
    u.myassert(k == n) ;
}

```

SORTS one bit
 $\Theta(n)$

```

private void rs(int nb) {
    for (int i = 0; i < nb; ++i) {  $\Theta(n+k)$ 
        rs1(i);
    }
}

```

```

protected void sorting_algorithm() {
    int nb = numberOfBits();
    rs(nb) ;
}

```

Figure 9.41: Code and complexity of straight radix sort

9.21 Radix exchange sort: from MSB to LSB

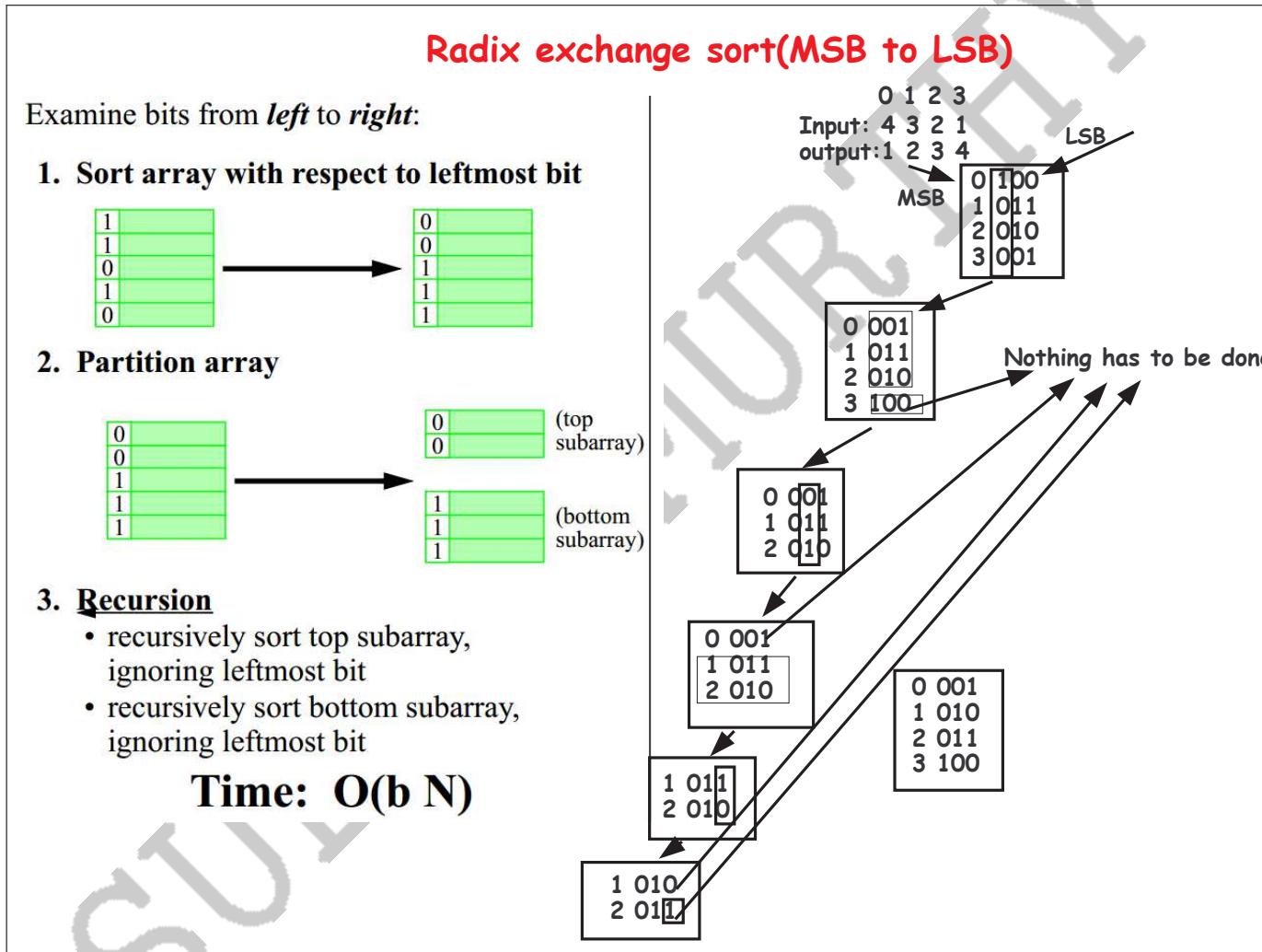


Figure 9.42: Radix exchange sort, from MSB to LSB

9.22 Problem set

Problem 9.22.1. Airport flight information system: Refer to the data file airport/data/flights.dat. The information is available in the following form:

DESTINATION|CARRIER|FLIGHT TIME|TERMINAL NUMBER
 New York - LaGuardia LGA |USAir|22:05|4|

9.22. PROBLEM SET

Burbank Bob Hope BUR |Delta|01:09|12|
Long Beach LGB |Delta|10:17|1|

You should output the data in the ascending order, as follows:

```
01:30 Albuquerque International Sunport ABQ    Continental 12  
01:30 Billings Logan International BIL    AirCanada 12  
01:30 Honolulu International HNL    Skywest 4  
01:31 Portland International Jetport PWM    Continental 16  
01:33 Portland International Jetport PWM    American 6
```

Use all the four sort discussed in this chapter. You should output 4 files as follows:

```
53881 Mar 6 16:54 bubble.dat  
53881 Mar 6 16:54 merge.dat  
53881 Mar 6 16:54 insertion.dat  
53881 Mar 6 16:54 quick.dat
```

The files should be exactly identical. Make sure that you run *Unix diff* command. Report the CPU time for each of the sort. You must write code in the file *airportsol.cpp*.

Problem 9.22.2. GOOGLE interview question: You have two identical eggs. Standing in front of a 100 floor building, you wonder what is the maximum number of floors from which the egg can be dropped without breaking it. What is the minimum number of tries needed to find out the solution?

Problem 9.22.3. GOOGLE interview question: " Given a file of 4 billion 32-bit integers, how to find one that appears at least twice?

Chapter 10

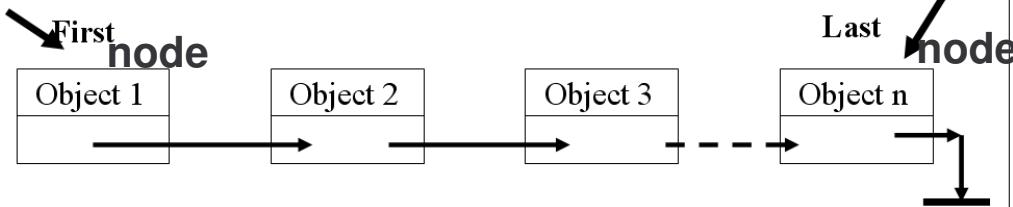
Singly linked list

10.1 Introduction

10.2 Linked list

Singly Un-Ordered Linked List

- Can have any number of element

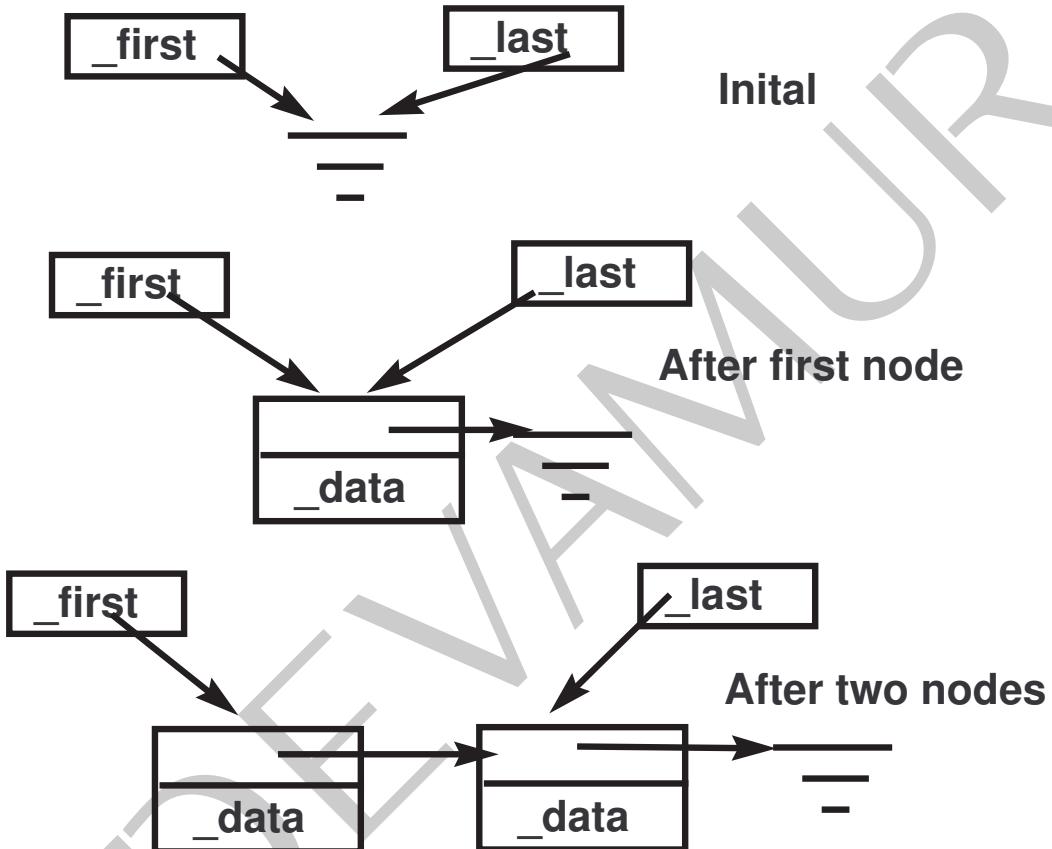


**Insertion at the end - O(1) time
Search may take n operation**

Figure 10.1: Singly linked list - Concept

10.3 Operation: Append

Append data to the end of the slist



```
template <class T>
const node<T>* slist<T>::append(const T& data) {
    node<T> *c = _create_a_node(data);
    if (!_first) {
        _first = c;
    } else {
        _last->_next = c;
    }
    _last = c;
    return c;
}
```

10.4 Operation: Unlink

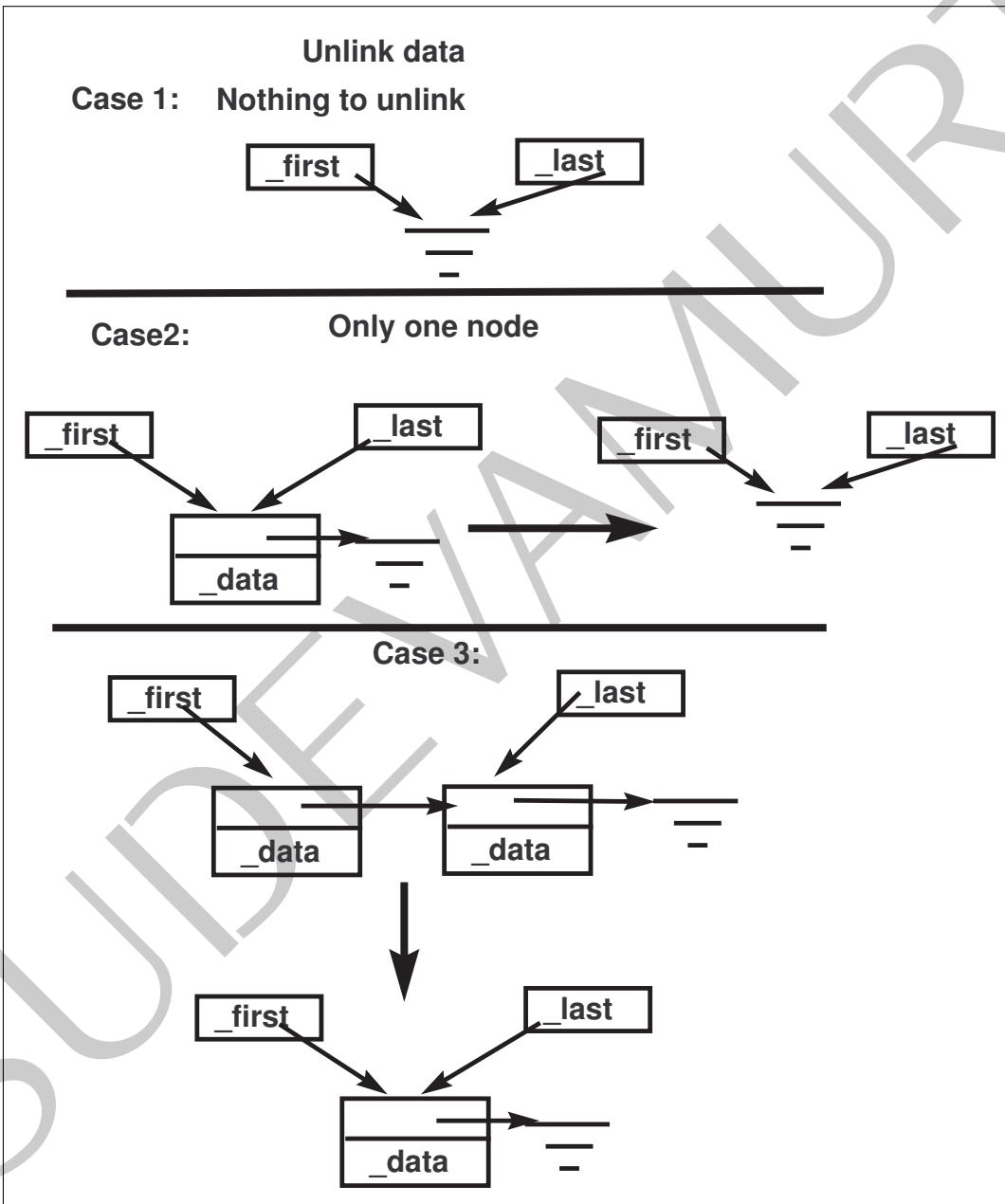


Figure 10.3: Operation unlink

10.5. STORING UDT AND POINTERS TO UDT IN A SINGLY LINKED LIST

10.5 Storing UDT and pointers to UDT in a singly linked list

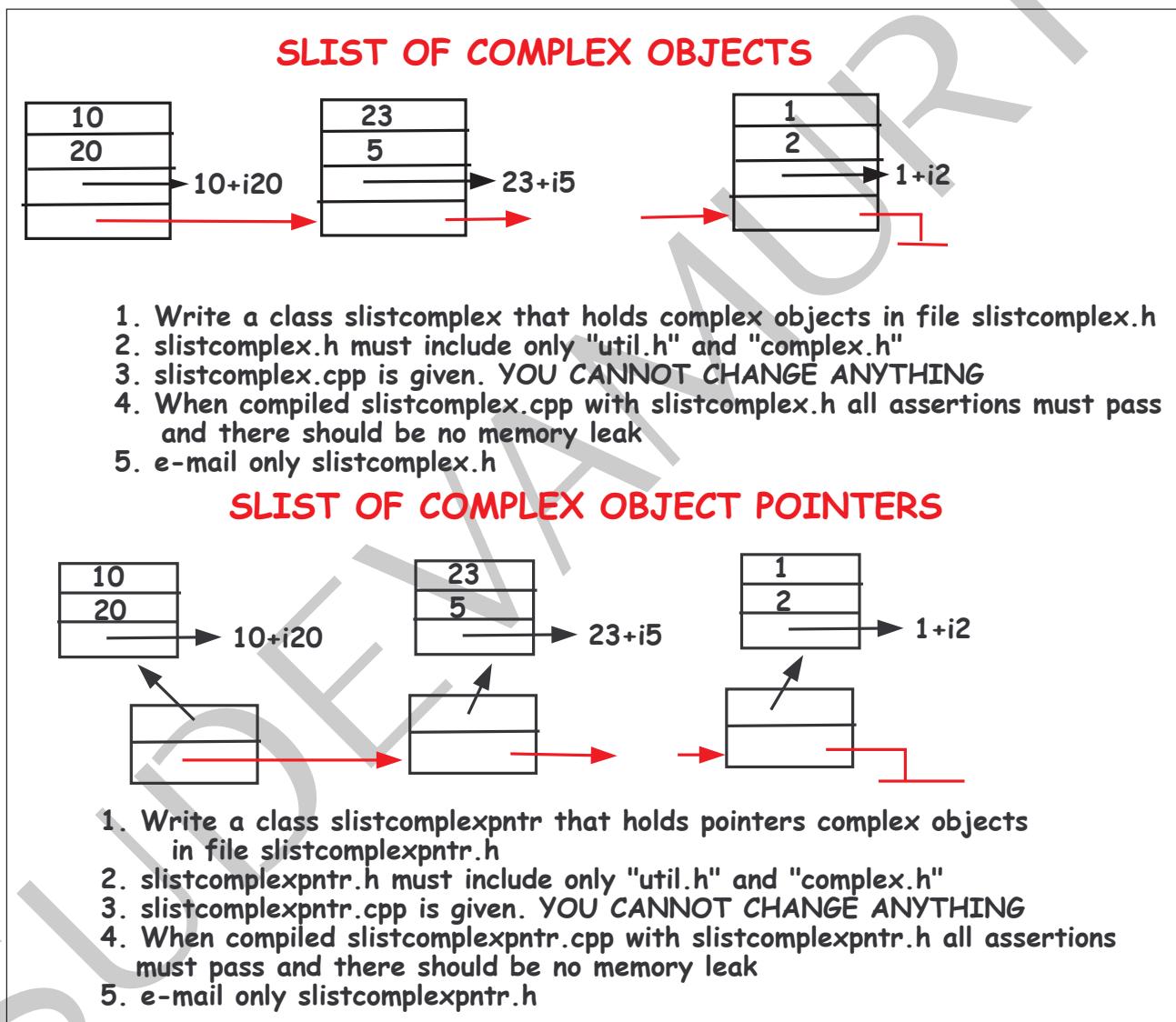


Figure 10.4: Implementing slist of complex objects and pointers to complex objects

Problems with storing UDT and pointers to UDT

```

class node {
public:
    node(const complex& data):_data(data),_next(NULL){}
private:
    complex _data;
    node* _next;
};

node* slistcomplex::_create_a_node(const complex& data){
    node* x = new node(data);
    return x;
}

void slistcomplex::_delete_a_node(node* n){
    delete n; //distructor of complex called
}

```

```

class node {
public:
    node(complex* data):_data(data),_next(NULL){}
private:
    complex* _data;
    node* _next;
};

node* slistcomplexptr::_create_a_node(complex* data){
    node* x = new node(data);
    return x;
}

void slistcomplexptr::_delete_a_node(node* n){
    //delete n->_data; //CAN WE DO???
    delete n; //distructor of complex not called
}

```

YOU cannot do delete n->data as complex object can be in heap or on stack

```

node *n = new complex();
node n = complex();
node(&n);

```

- 1. When complex* objects are removed through unlink, the node will go away
- 2. Distructor of complex is NOT called. Memory leak
- 3. You will loose the pointers.
- 4. When slistcomplex distructor is called, you will loose all the pointers. Memory leak
- 5. Only way is to get pointer first, and then unlink.

```

node* slistcomplex::_find(const complex& data){
    node* f = _first;
    while (f) {
        if (data == f->_data){
            //Calls the equal operator of complex
            return f;
        }
        f = f->_next;
    }
    return NULL;
}

```

```

node* slistcomplexptr::_find(complex* data){
    node* f = _first;
    while (f) {
        //if (*(data) == *(f->_data)){
        if (data == f->_data){
            return f;
        }
        f = f->_next;
    }
    return NULL;
}

```

- 1. Equal operator of complex is not called.
It will do pointer comparision. So unless you store the original pointers, it will never match
- 2. In a templated code, how do you know to do if `(*(data) == *(f->_data))`

```

slist(void(*pv)(T& c) = NULL, int(*cf)(const T& c1, const T& c2) = NULL);

```

Pointer to delete function

```

void delete_complex(complex*& c){
    delete c;
}

```

Pointer to compare function

```

int complex_larger_compare(complex* const& c1, complex* const& c2){
    return (*c1 == *c2);
}

```

Figure 10.5: Problems with storing UDT and pointers to UDT

10.6. CONCEPT OF ITERATORS

10.6 Concept of iterators

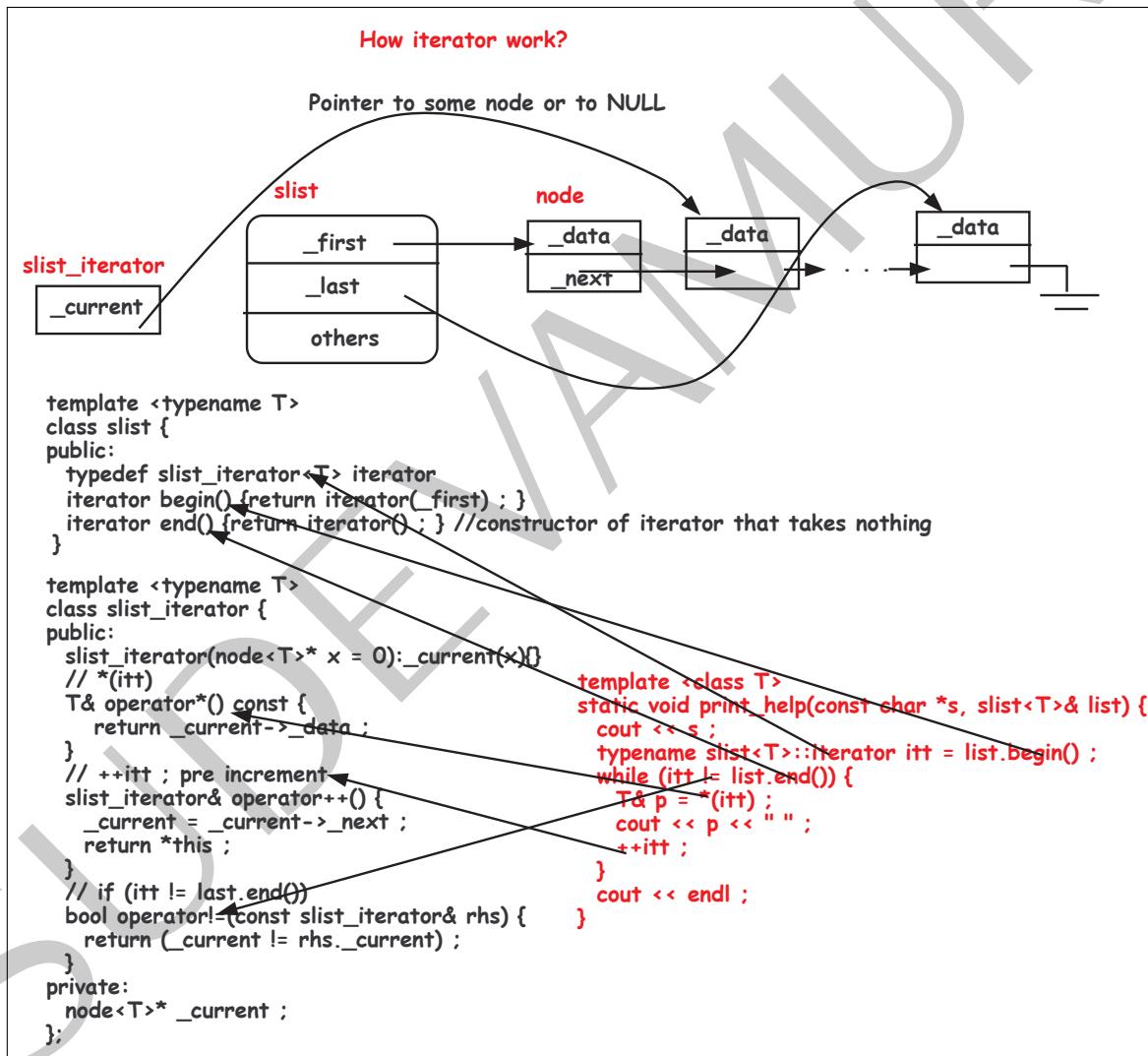


Figure 10.6: How iterator works?

10.7 slist Code

10.8 Problem set

Problem 10.8.1. Implement *slist* of complex objects and *slist* of pointers to complex objects as shown in figure 10.7

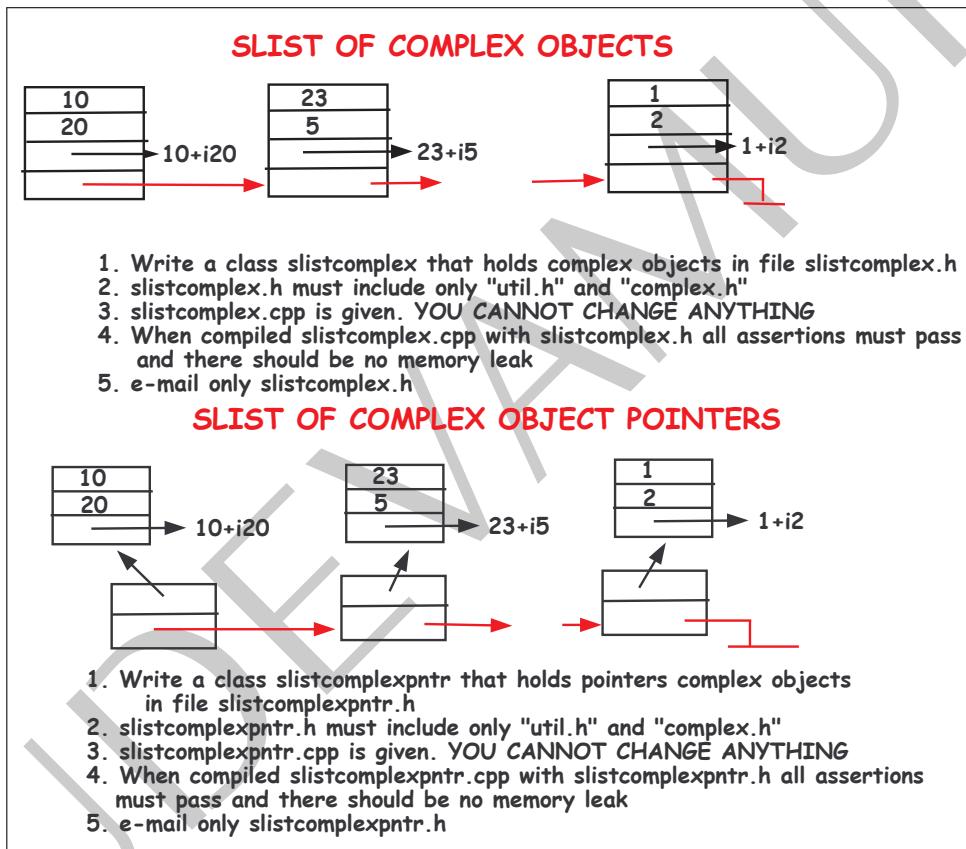


Figure 10.7: Implementing *slist* of complex objects and pointers to complex objects

10.8. PROBLEM SET

Problem 10.8.2. Look for **WRITE YOUR CODE HERE** in the file **slist.h** and **slist.hpp** and write the code. Use **slisttest.cpp** and test your code. You cannot modify anything in the file **slisttest.cpp**.

Problem 10.8.3. Implement the ordered slist as shown in the figure 10.8:

ORDERED SLIST oslist

3, 7, 1, 5, 2

in slist: we store as 3,7,1,5,2

Let us say we get 4 now

in slist: we store as 3,7,1,5,2,4

In ordered list, oslist: we store as 1,2,3,5,7

Let us say we get 4 now

In ordered list, oslist: we store as 1,2,3,4,5,7

Let os1 = { 1, 5, 7}

Let os2 = {2, 3, 5, 7, 10, 11 }

UNION osu = {1,2,3,5,7,10,11}

INTERSECTION osi = {2,5,7}

**Implement generic templated oslist
that supports:**

1. insertion
2. deletion
3. is_in_list
4. union of two lists
5. intersection of two lists

Figure 10.8: Ordered slist

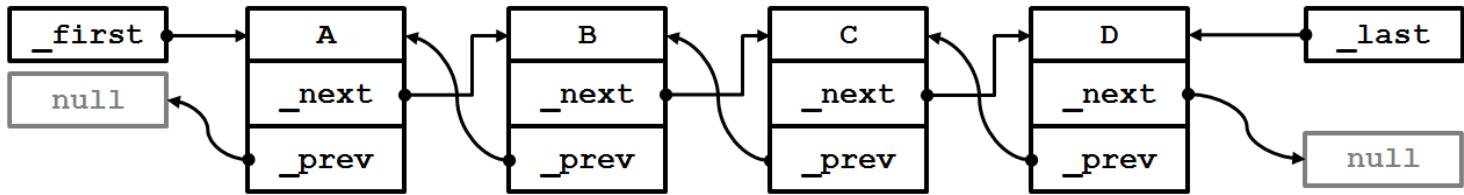
10.8. PROBLEM SET

Problem 10.8.4. Implement a **doubly linked** class. Write a test module that tests this class by inserting and deleting **complex class** objects by **value** and by **address**.

CHAPTER 10. SINGLY LINKED LIST

Problem 10.8.5. Implement a **doubly linked** class using only single pointer as explained below.

Doubly Linked Lists



Advantages:

- Allows for both forwards and backwards traversal of list

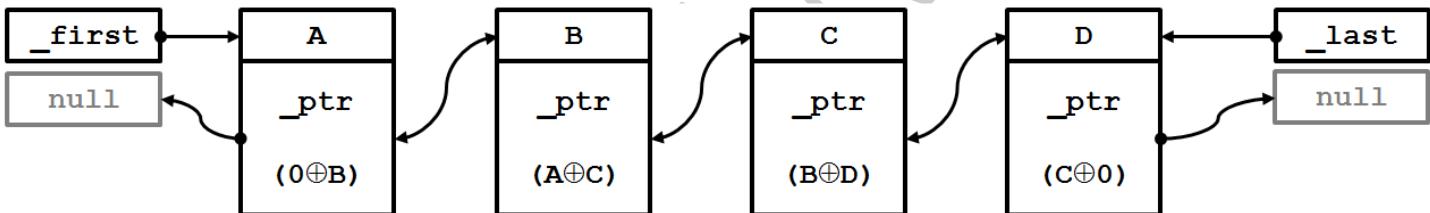
Disadvantages:

- Doubles the necessary space for storing connection pointers

A Doubly Linked List which requires only a single pointer may be created by leveraging XOR properties.

X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

$$\begin{aligned} L \oplus (L \oplus R) &= R \\ R \oplus (R \oplus L) &= L \end{aligned}$$



The value of `_ptr` would be stored as the XOR of the preceding and following nodes.

e.g.

$$B._ptr = \text{address}(A) \oplus \text{address}(C)$$

Note that you require both the current node `_ptr` value as well as the previous pointer that you have just iterated over in order to calculate the next value.

e.g.

Traversing left to right:

```

Address(A) = (must be given) e.g. _first
Address(B) = NULL ⊕ A._ptr
Address(C) = Address(A) ⊕ B._ptr
Address(D) = Address(B) ⊕ C._ptr
Etc...
  
```

Homework Assignment

- Write a doubly linked list using single pointer
- For easiness, you need NOT have to write a template version. Assume the data is only int.
- Call this class as: dlistintxor
- Add 10 elements
- Print dlist
- Remove 5 elements
- Print
- Add another 4 elements
- E-mail dlistintxor.h, dlistintxor.cpp and dlistintxor.cpp

CHAPTER 10. SIMPLY LINKED LIST

Problem 10.8.6. GOOGLE interview question: You are given a list of numbers. When you reach the end of the list you will come back to the beginning of the list (a circular list). Write the most efficient algorithm to find the minimum # in this list. Find any given # in the list. The numbers in the list are always increasing but you don't know where the circular list begins, ie: 38, 40, 55, 89, 6, 13, 20, 23, 36.

Problem 10.8.7. GOOGLE interview question: You are given an array [a1 To an] and we have to construct another array [b1 To bn] where $b_i = (a_1 * a_2 * \dots * a_n) / a_i$. you are allowed to use only constant space and the time complexity is O(n). No divisions are allowed.

Problem 10.8.8. GOOGLE interview question: Write a function to find the middle node of a single linked list.

Problem 10.8.9. MICROSOFT interview question: How can you print singly linked list in reverse order? (it's a huge list and you cant use recursion)

Problem 10.8.10. MICROSOFT interview question: How would you reverse the bits of a number with $\log N$ arithmetic operations, where N is the number of bits in the integer (eg 32,64..)

Problem 10.8.11. MICROSOFT interview question: Implement an algorithm to reverse a doubly linked list.

10.8. PROBLEM SET

VASUDEVAMURTHY

Chapter 11

STL List Class

11.1 Introduction

11.2 List class public functions

11.2. LIST CLASS PUBLIC FUNCTIONS

Constructors		
<code>list<T> l</code>	Default constructor	$O(1)$
<code>list<T> l(alist)</code>	Copy constructor with explicit size	$O(n)$
<code>l = alist</code>	Equal operator	$O(n)$
Size		
<code>l.size()</code>	Number of elements currently held in list l	$O(1)$
<code>l.empty()</code>	True if l is empty	$O(1)$
Insertion		
<code>l.push_front(i)</code>	Add element in front	$O(1)$
<code>l.push_back(i)</code>	Add element in back	$O(1)$
<code>itt1 = l.insert(itt,i)</code>	Insert i immediately BEFORE the element specified by i. An iterator to the element i is returned	$O(1)$
Deletion		
<code>l.pop_front()</code>	Remove FIRST element NOTHING RETURNED	$O(1)$
<code>l.pop_back()</code>	Remove LAST element NOTHING RETURNED	$O(1)$
<code>itt1 = l.erase(itt)</code>	Remove element pointed by itt. Returns an iterator AFTER the one removed	$O(1)$
<code>remove(const T& a)</code>	Remove element with the value a from the list NOTHING RETURNED	$O(n)$
Random access		
<code>l.front()</code>	Return the FIRST element by alias. First element still exists as FIRST in list	$O(1)$
<code>l.back()</code>	Return the LAST element by alias. Last element still exists as LAST in list	$O(1)$
Random access	NOT POSSIBLE Vector is a random access	
Copyright © Jagadeesh Vasudevan. This is my personal notes. Do not copy.		

Figure 11.1: List class functions

11.3 Understanding list class and iterators

```
1 /*-----  
2 Copyright (c) 2010 Author: Jagadeesh Vasudevarmurthy  
3 Filename: slist.cpp  
4 compile: g++ slist.cpp  
5 Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)  
6 -----*/  
7 #include <iostream>  
8 #include <list>  
9 #include <stdexcept> //Without this catch will NOT work on Linux  
10  
11  
12 #ifdef _WIN32 //Will not work in Window7 if U write WIN32  
13 #include "..\..\c\complex.h"  
14 #else  
15 #include "../c/complex.h"  
16 #endif  
17  
18 using namespace std;  
19  
20 /*-----  
21 static definition - only once at the start  
22 Change to false, if you don't need verbose  
23 -----*/  
24 bool complex::_display = true;  
25  
26 /*-----  
27 Multiply integer by 10  
28 -----*/  
29 static void multiply_by_10(int& x){  
30     x = x * 10;  
31 }  
32  
33 /*-----  
34 print1  
35 -----*/  
36 template <typename T>  
37 static void print1(const char* s, const list<T>& a){  
38     cout << s << endl;  
39     cout << "-----" << endl;  
40     cout << "size = " << a.size() << " "  
41     cout << endl;  
42     typename list<T>::const_iterator itt = a.begin();  
43     int i = 0;  
44     while (itt != a.end()) {  
45         cout << "a[" << i++ << "] = " << *(itt) << " "  
46         itt++;  
47     }  
48     cout << endl;  
49  
50     i = a.size()-1;  
51     while (itt != a.begin()) {  
52         --itt;  
53         cout << "a[" << i-- << "] = " << *(itt) << " "  
54     }  
55     cout << endl;
```

```
56
57 cout << "-----" << endl ;
58 }
59
60 /*-
61 apply a function pf on elements of list a
62 -----*/
63 template <typename T>
64 static void apply(const char* s, list<T>& a, void(*pf)(T& x)) {
65 cout << s << endl ;
66 cout << "-----" << endl ;
67 typename list<T>::iterator itt = a.begin(); //Note typename here.
68 //WILL NOT COMPILE IN LINUX without typename but works on Visual studio
69 //Hard to find this solution.
70 while (itt != a.end()){
71 T& p = *itt ;
72 pf(p) ;
73 itt++ ;
74 }
75 cout << endl ;
76 }
77
78 /*-
79 Find an element b in the list.
80 If the element is not there, return a.end
81
82 Calling this routine fails like this:
83 slist.cpp: error: conversion from `std::_List_iterator<int>' to non-scalar type
84 `std::_List_iterator<int*>' requested
85 -----*/
86 template <typename T>
87 static typename list<int>::iterator find_an_element(list<T>& a, const T& b) {
88 typename list<T>::iterator itt = a.begin();
89 while (itt != a.end()) {
90 if (*itt == b) {
91 break ;
92 }
93 itt++ ;
94 }
95 return itt ;
96 }
97
98 /*-
99 Insert an element x before element b
100 -----*/
101 template <typename T>
102 static void insert_an_element_before(list<T>& a, const T& b, const T& x) {
103 cout << "Let us insert " << x << " before " << b << endl ;
104 //typename list<T>::iterator itt = find_an_element(a,b) ;
105 typename list<T>::iterator itt = a.begin() ;
106 while (itt != a.end()) {
107 if (*itt == b) {
108 break ;
109 }
110 itt++ ;
```

```
111 }
112 if (itt != a.end()) {
113     typename list<T>::iterator itt1 = a.insert(itt,x);
114     cout << "itt1 is pointing to " << *itt1 << endl;
115     print1("After inserting ",a);
116 }else {
117     cout << " There is no element " << b << " in the list. Hence " << x << " is NOT inserted\n";
118 }
119 }
120
121 /*-
122 Remove an element x if it exists
123 -----*/
124 template <typename T>
125 static void remove_an_element(list<T>& a, const T& x, void (*pntr_to_func_to_delete_data)(T& c) = NULL) {
126     cout << "Let us remove " << x << "\n";
127     //typename list<T>::iterator itt = find_an_element(a,x);
128     typename list<T>::iterator itt = a.begin();
129     while (itt != a.end()) {
130         if (*itt == x) {
131             break;
132         }
133         itt++;
134     }
135     if (itt != a.end()) {
136         if (pntr_to_func_to_delete_data) {
137             pntr_to_func_to_delete_data(*(itt));
138         }
139         a.erase(itt);
140         print1("After removing ",a);
141     }else {
142         cout << " There is no element " << x << " in the list. Hence " << x << " is NOT deleted\n";
143     }
144 }
145
146
147 */
148 /*-
149 a[0] .... a[9]
150
151 begin() will point to a[0]
152 end() will NOT POINT to a[9], but to one element past a[9]
153 That means real end is: end()-1 ;
154 -----*/
155 static void understanding_iterator(list<int>& a) {
156 {
157     cout << "Understanding forward traversal" << endl;
158     list<int>::const_iterator itt = a.begin();
159     while (itt != a.end()) {
160         cout << *itt << " ";
161         itt++;
162     }
163     cout << endl;
164 }
165 {
```

```
166 cout << "Understanding reverse traversal" << endl ;
167 list<int>::const_iterator itt = a.end() ;
168 //cout << *itt << endl ; ----- CORE DUMPS
169 while (itt != a.begin()){
170     itt-- ; //decrement iterator before using
171     cout << *itt << " " ;
172 }
173 cout << endl ;
174 }
175 cout << "-----" << endl ;
176 }
177
178
179 /*-----*
180 a.front() -- Returns first element of the list
181 a.back() -- Returns last element of the list
182 a.push_front(v) -- v is inserted as the first element of the list
183 a.push_back(v) -- v is inserted as the back element of the list
184 a.pop_front() - First element of the list is removed. Nothing is returned
185 a.pop_back() - Last element of the list is removed. Nothing is returned
186 -----*/
187 static void understanding_access(list<int>& a){
188     print1("begin with",a);
189     for (int i = 0; i < 5; i++){
190         a.push_back(i);
191     }
192     print1("After Inserting 5 elements from the back",a);
193     for (int i = 0; i < 5; i++){
194         a.push_front(i*10+1);
195     }
196     print1("After Inserting 5 elements from the front",a);
197
198     int& y1 = a.front();
199     cout << "The front of the list has " << y1 << endl ;
200     int& y2 = a.back();
201     cout << "The back of the list has " << y2 << endl ;
202
203     print1("After front and back operation",a);
204
205 //y = a.pop_front(); WRONG. pop cannot return ;
206 a.pop_front();
207 a.pop_front();
208 print1("After Removing two elements from the front",a);
209 a.pop_back();
210 a.pop_back();
211 print1("After Removing two elements from the back",a);
212
213 int x = a.size() - 1;
214 //Vector is a random access object. list is sequential access object
215 // you cannot access a random position of a list
216 //y = a[x];
217 insert_an_element_before(a,0,999);
218 remove_an_element(a,21);
219 }
```

```
221 /*-----  
222 test_list_of_integers  
223 -----*/  
224 static void test_list_of_integers(){  
225     list<int> a;  
226     print1("begin with",a);  
227     understanding_access(a);  
228     understanding_iterator(a);  
229  
230     apply("multiply by 10",a,multiply_by_10);  
231     cout << endl ;  
232     apply("print using iterator",a,print_integer);  
233     cout << endl ;  
234  
235     a.clear();  
236     print1("Using clear",a);  
237 }  
238  
239 /*-----  
240 array of integer pointers  
241 -----*/  
242 static void test_list_of_ptr_integers(){  
243     list<int*> a;  
244     print1("begin with",a);  
245     int* a1ptr = NULL ;  
246     int* a11ptr = NULL ;  
247     for (int i = 0; i < 5; i++){  
248         int *p = ALLOC(int)(i*10+1);  
249         if (i == 1){  
250             a1ptr = p ;  
251         }  
252         a.push_back(p);  
253     }  
254     apply("After Inserting 5 elements from the back",a,print_integer);  
255     for (int i = 0; i < 5; i++){  
256         int *p = ALLOC(int)(i*10+1);  
257         if (i*10+1 == 11){  
258             a11ptr = p ;  
259         }  
260         a.push_front(p);  
261     }  
262     apply("After Inserting 5 elements from the front",a,print_integer);  
263  
264     int*& y = a.front();  
265     cout << "The front of the list has " << *y << endl ;  
266     int*& ba = a.back();  
267     cout << "The back of the list has " << *ba << endl ;  
268  
269 //y = a.pop_front(); WRONG. pop cannot return;  
270 FREE(a.front());  
271 a.pop_front();  
272  
273 FREE(a.front());  
274 a.pop_front();  
275
```

```
276 apply("After Removing two elements from the front",a.print_integer);
277
278 FREE(a.back());
279 a.pop_back();
280
281 FREE(a.back());
282 a.pop_back();
283 apply("After Removing two elements from the back",a.print_integer);
284
285 int *xx = ALLOC(int)(999);
286 insert_an_element_before(a,a1ptr,xx);
287 remove_an_element(a,a1ptr,delete_int);
288
289 list<int*> b(a);
290 apply("Contents of list b:",b.print_integer);
291
292 apply("Delete all the contents of a ",a.delete_int);
293 /* Why you should NOT do this */
294 //apply("Delete all the contents of b ",b.delete_int);
295 }
296
297 /*-----
298 array of user defined type
299 -----*/
300 static void test_list_of_udt(){
301 list<complex> a;
302 print1("begin with",a);
303 for (int i = 0; i < 5; i++){
304 a.push_back(complex(i,-i));
305 }
306 print1("After Inserting 5 elements from the back",a);
307 for (int i = 0; i < 5; i++){
308 a.push_front(complex((i*10+1),-(i*10+1)));
309 }
310 print1("After Inserting 5 elements from the front",a);
311
312 complex& y = a.front();
313 cout << "The front of the list has " << y << endl;
314 complex& z = a.back();
315 cout << "The back of the list has " << z << endl;
316 {
317 /* see what happens if U get complex by value */
318 complex vf = a.front();
319 }
320
321 //y = a.pop_front(); WRONG. pop cannot return ;
322 a.pop_front();
323 a.pop_front();
324 print1("After Removing two elements from the front",a);
325 a.pop_back();
326 a.pop_back();
327 print1("After Removing two elements from the back",a);
328
329 {
330 complex c1(1,-1);
```

```
331 complex c2(999,-999);
332 insert_an_element_before(a,c1,c2);
333 complex c3(21,-21);
334 remove_an_element(a,c3);
335 }
336 }
337
338 /*-----
339 array of user defined pointer type
340 -----*/
341 static void test_list_of_ptr_udt(){
342 list<complex*> a;
343 print1("begin with",a);
344 complex* a1ptr = NULL;
345 complex* a11ptr = NULL;
346 for (int i = 0; i < 5; i++){
347 complex *p = ALLOC(complex)(i,-i);
348 if (i == 1){
349 a1ptr = p;
350 }
351 a.push_back(p);
352 }
353 apply("After Inserting 5 elements from the back",a,print_complex);
354 for (int i = 0; i < 5; i++){
355 int x = i * 10 + 1;
356 complex *p = ALLOC(complex)(x,-x);
357 if (x == 11){
358 a11ptr = p;
359 }
360 a.push_front(p);
361 }
362 apply("After Inserting 5 elements from the front",a,print_complex);
363
364 complex*& af = a.front();
365 cout << "The front of the list has " << *af << endl;
366 complex*& ab = a.back();
367 cout << "The back of the list has " << *ab << endl;
368
369 //y = a.pop_front(); WRONG. pop cannot return;
370 FREE(a.front());
371 a.pop_front();
372
373 FREE(a.front());
374 a.pop_front();
375
376 apply("After Removing two elements from the front",a,print_complex);
377
378 FREE(a.back());
379 a.pop_back();
380
381 FREE(a.back());
382 a.pop_back();
383
384 apply("After Removing two elements from the back",a,print_complex);
385
```

11.4 STL like slist implementation

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy  
3 Filename: slist_like_stl.h  
4 -----*/  
5 #ifndef slist_h  
6 #define slist_h  
7  
8 #include "util.h"  
9  
10 /*-----  
11 All forward declaration  
12 -----*/  
13 template <typename T>  
14 class node ;  
15  
16 template <typename T>  
17 class slist ;  
18  
19 template <typename T>  
20 class slist_iterator ;  
21  
22 /*-----  
23 typename node  
24 -----*/  
25 template <typename T>  
26 class node {  
27 public:  
28     node(const T& data):_data(data),_next(NULL){}  
29     ~node(){  
30         _next = NULL ;  
31     }  
32     const T& get_data() const {return _data;}  
33  
34 private:  
35     T _data ;  
36     node<T>* _next ;  
37  
38     friend class slist<T> ; //slist can access nodes private part  
39     friend class slist_iterator<T> ; //slist_iterator can access node private part  
40  
41     /* no body can copy node or equal node */  
42     node(const node<T>& x);  
43     node& operator=(const node<T>& x);  
44 };  
45  
46 /*-----  
47 typename slist  
48 -----*/  
49 template <typename T>  
50 class slist {  
51 public:  
52     typedef slist_iterator<T> iterator ;  
53     typedef const slist_iterator<T> const_iterator ;  
54  
55     slist(void(*pv)(T& c) = NULL, int(*cf)(const T& c1, const T& c2) = NULL);  
76
```

```
56 ~slist();
57
58 void change_functions(void(*pv)(T& c),int(*cf)(const T& c1, const T& c2));
59 int size() const;
60 void append(const T& data);
61 void append_after(const T& p, const T& data);
62 bool find(const T& data);
63 bool unlink_data(const T& data);
64 void reverse();
65
66 void create_a_loop(int from, int to);
67 bool detect_loop() const;
68 bool display() const {return _display;}
69
70 /* for iterators */
71 iterator begin() {return iterator(_first); }
72 iterator end() {return iterator(); } //constructor of iterator that takes nothing
73 const const_iterator begin() const {return iterator(_first); }
74 const const_iterator end() const {return iterator(); } //constructor of iterator that takes nothing
75
76 private:
77 node<T>* _first;
78 node<T>* _last;
79 void (*_ptr_to_func_to_delete_data)(T& c) ;
80 int(*_ptr_to_compare_func)(const T& c1, const T& c2);
81 int _num_nodes_allocated;
82 int _num_nodes_freed;
83 static bool _display;
84
85 node<T>* _create_a_node(const T& data);
86 void _delete_a_node(node<T> *n);
87 node<T>* _find(const T& data);
88 bool _unlink_data(const node<T>* p);
89
90 /* no body can copy slist or equal slist */
91 slist(const slist<T>& x);
92 slist& operator=(const slist<T>& x);
93 };
94
95 /*
96 typename slist iterator
97 -----
98 template <typename T>
99 class slist_iterator {
100 public:
101 slist_iterator(node<T>* n = 0):_current(n){}
102 ~slist_iterator(){}
103 slist_iterator(const slist_iterator<T>& x) { _current = x._current; }
104 slist_iterator& operator=(const slist_iterator<T>& x) {
105 _current = x._current;
106 return *this;
107 }
108
109 // *(itt)
110 const T& operator*() const { return _current->_data; } 377
```

```
111 // ++itt ; pre increment
112 slist_iterator& operator++() {
113     _current = _current->_next ;
114     return *this ;
115 }
116
117 // itt++ ; post increment
118 slist_iterator operator++(int i) {
119     slist_iterator t(*this) ;
120     _current = _current->_next ;
121     return *this ;
122 }
123
124 // if (itt == last.end())
125 bool operator==(const slist_iterator& rhs) {
126     return (_current == rhs._current) ;
127 }
128
129 // if (itt != last.end())
130 bool operator!=(const slist_iterator& rhs) {
131     return (_current != rhs._current) ;
132 }
133
134
135 private:
136     node<T>* _current ;
137 };
138
139
140 /*-
141 slist Constructor
142 */
143 template <typename T>
144 slist<T>::slist(void(*pv) (T& c), int(*cf) (const T& c1, const T& c2)):
145     _first(NULL), _last(NULL), _ptr_to_compare_func(cf), _ptr_to_func_to_delete_data(pv),
146     _num_nodes_allocated(0), _num_nodes_freed(0)
147 {
148     if (display()) {
149         cout << "in slist constructor:" << endl ;
150     }
151 }
152
153 /*-
154 slist Distructor
155 */
156 template <typename T>
157 slist<T>::~slist() {
158     if (display()) {
159         cout << "in slist distructor:" << endl ;
160     }
161     node<T>* t = _first ;
162     while (t) {
163         node<T>* ct = t ;
164         t = t->_next ;
165         _delete_a_node(ct) ;
```

```
166 }
167 if (_num_nodes_allocated != _num_nodes_freed) {
168     assert(0);
169 }
170 _first = NULL;
171 _last = NULL;
172 }
173
174 /*-
175 change function
176 */
177 template <typename T>
178 void slist<T>::change_functions(void(*pv)(T& c), int(*cf)(const T& c1, const T& c2)) {
179     _ptr_to_compare_func = cf;
180     _ptr_to_func_to_delete_data = pv;
181 }
182
183 /*-
184 How many elements are in the list
185 */
186 template <typename T>
187 int slist<T>::size() const
188 {
189 }
190
191 /*-
192 create a node in slist
193 */
194 template <typename T>
195 node<T>* slist<T>::_create_a_node(const T& data){
196     _num_nodes_allocated++;
197     node<T>* x = ALLOC(node<T>)(data); /* if T is by value, copy const will be called for T */
198     return x;
199 }
200
201
202 /*-
203 delete a node from slist
204 */
205 template <typename T>
206 void slist<T>::_delete_a_node(node<T> *n){
207     _num_nodes_freed++;
208     if (_ptr_to_func_to_delete_data) {
209         _ptr_to_func_to_delete_data(n->_data); /* Distructor of T will be called*/
210     }
211     FREE(n);
212 }
213
214
215 /*-
216 append data to the end of the slist
217 */
218 template <typename T>
219 void slist<T>::append(const T& data)
220 {
221 }
222
223 /*-
224 append data after p.
225 if p is not there, add data at the end
226 */
227
```

```
237 template <typename T>
238 void append_after(const T& p, const T& data) {
239 //WRITE YOUR CODE HERE
240 }
241
242 /*-
243 find data in the slist
244 */
245 template <typename T>
246 node<T>* slist<T>::_find(const T& data)
259
260 /*-
261 find data in the slist
262 */
263 template <typename T>
264 bool slist<T>::find(const T& data){
265 node<T>* f = _find(data);
266 return (f) ? true : false ;
267 }
268
269 /*-
270 Unlink node p from slist
271 */
272 template <typename T>
273 bool slist<T>::_unlink_data(const node<T>* f)
306
307 /*-
308 Unlink node p from slist
309 */
310 template <typename T>
311 bool slist<T>::unlink_data(const T& data) {
312 node<T>* f = _find(data);
313 if (f) {
314 return _unlink_data(f);
315 }
316 return false ;
317 }
318
319 /*-
320 1->4->8->7
321
322 7->8->4->1
323 */
324 template <typename T>
325 void slist<T>::reverse() {
326 //WRITE YOUR CODE HERE
327 }
328
329 /*-
330 create a loop
331 */
332 template <typename T>
333 void slist<T>::create_a_loop(int from, int to) {
334 int i = 0 ;
335 node<T> *fp = NULL ;
```

```
336 node<T> *tp = NULL ;
337 node<T> *f = _first ;
338 while (f) {
339     i++ ;
340     if (i == from) {
341         fp = f ;
342     }
343     if (i == to) {
344         tp = f ;
345     }
346     f = f->_next ;
347 }
348 if (fp && tp) {
349     fp->_next = tp ;
350 }
351 }
352
357
358 template <typename T>
359 bool slist<T>::detect_loop() const
378
379 #endif
380
381
```

11.4. STL LIKE SLIST IMPLEMENTATION

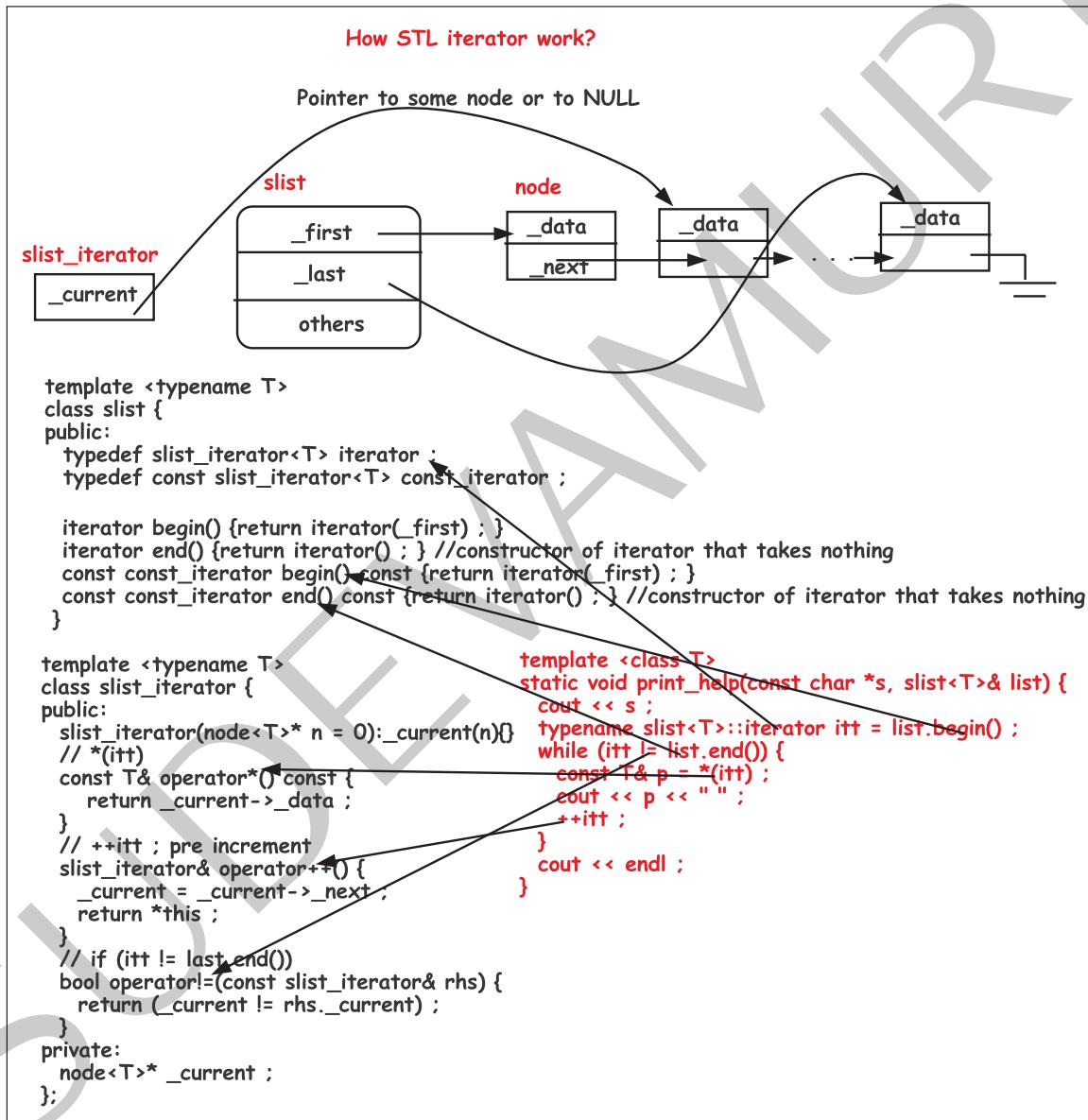


Figure 11.2: How iterator works?

```
1 /*-
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy
3 Filename: slist_like_stl.cpp
4 compile: g++ slist_like_stl
5 comment test_loop
6
7     Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)
8 Program exited with status code 0.
9 */
10 #include "slist_like_stl.h"
11 #include "complex.h"
12
13 /*-
14 static definition - only once at the start
15 Change to false, if you don't need verbose
16 */
17 bool complex::_display = true;
18
19 template <typename T>
20 bool slist<T>::_display = true;
21
22
23 /*-
24 printing help
25 */
26 template <class T>
27 static void print_help(const char *s, slist<T>& list) {
28     cout << s;
29     typename slist<T>::iterator itt = list.begin();
30     while (itt != list.end()) {
31         const T& p = *(itt);
32         cout << p << " ";
33         ++itt;
34     }
35     cout << endl;
36 }
37
38 /*-
39 in slist constructor:
40 After appending:0 1 2 3 4 5 6 7 8 9
41 After unlinking even items:1 3 5 7 9
42 After reverse:1 3 5 7 9
43 data 1 is there in the slist. I am going to unlink it
44 After unlinking data 1:3 5 7 9
45 After unlinking data 100:3 5 7 9
46 After unlinking data 9:3 5 7
47 in slist distructor:
48 */
49 static void test_slist_of_integers(){
50     const int N = 10;
51     slist<int> a(NULL,int_descending_order);
52     for (int i = 0; i < N; i++) {
53         a.append(i);
54     }
55     print_help("After appending:",a);
```

```
56 for (int i = 0; i < N; i+= 2) {
57     a.unlink_data(i);
58 }
59 print_help("After unlinking even items:",a);
60 a.reverse();
61 print_help("After reverse:",a);
62 bool k = a.find(1);
63 if (k) {
64     cout << "data 1 is there in the slist. I am going to unlink it \n";
65     a.unlink_data(1);
66 }else {
67     cout << "data 1 is NOT there in the slist \n";
68 }
69 print_help("After unlinking data 1:",a);
70 k = a.find(100);
71 if (k) {
72     a.unlink_data(100);
73 }
74 print_help("After unlinking data 100:",a);
75 k = a.find(9);
76 if (k) {
77     a.unlink_data(9);
78 }
79 print_help("After unlinking data 9:",a);
80 }
81
82 /*-
83 slist of complex
84
85 In slist constructor:
86 In complex constructor:
87 In complex copy Constructor 0+i0
88 In complex copy Constructor 1+i1
89 In complex copy Constructor 2+i2
90 In complex copy Constructor 3+i3
91 In complex copy Constructor 4+i4
92 In complex copy Constructor 5+i5
93 In complex copy Constructor 6+i6
94 In complex copy Constructor 7+i7
95 In complex copy Constructor 8+i8
96 In complex copy Constructor 9+i9
97 After appending:0+i0 1+i1 2+i2 3+i3 4+i4 5+i5 6+i6 7+i7 8+i8 9+i9
98 In complex Destructor 0+i0
99 In complex Destructor 2+i2
100 In complex Destructor 4+i4
101 In complex Destructor 6+i6
102 In complex Destructor 8+i8
103 After unlinking even items:1+i1 3+i3 5+i5 7+i7 9+i9
104 After reverse:1+i1 3+i3 5+i5 7+i7 9+i9
105 data 1+i1 is there in the slist. I am going to unlink it
106 In complex Destructor 1+i1
107 After unlinking data 1+i1:3+i3 5+i5 7+i7 9+i9
108 After unlinking data 100+i100:3+i3 5+i5 7+i7 9+i9
109 In complex Destructor 9+i9
110 After unlinking data 9+i9:3+i3 5+i5 7+i7
```

```
111 In complex Destructor 9+i9
112 in slist distructor:
113 In complex Destructor 3+i3
114 In complex Destructor 5+i5
115 In complex Destructor 7+i7
116 -----
117 static void test_slist_of_udt(){
118     const int N = 10 ;
119     slist<complex> a(NULL,complex_larger_compare) ;
120     complex c ;
121     for (int i = 0; i < N; i++) {
122         c.setxy(i,i) ;
123         a.append(c) ;
124     }
125
126     print_help("After appending:",a) ;
127
128     for (int i = 0; i < N; i=i + 2) {
129         c.setxy(i,i) ;
130         a.unlink_data(c) ;
131     }
132     print_help("After unlinking even items:",a) ;
133     a.reverse() ;
134     print_help("After reverse:",a) ;
135     c.setxy(1,1) ;
136     bool k = a.find(c) ;
137     if (k) {
138         cout << "data " << c << " is there in the slist. I am going to unlink it \n" ;
139         a.unlink_data(c) ;
140     }else {
141         cout << "data " << c << " is NOT there in the slist \n" ;
142     }
143
144     print_help("After unlinking data 1+i1:",a) ;
145
146     c.setxy(100,100) ;
147     k = a.find(c) ;
148     if (k) {
149         a.unlink_data(c) ;
150     }
151     print_help("After unlinking data 100+i100:",a) ;
152     c.setxy(9,9) ;
153     k = a.find(c) ;
154     if (k) {
155         a.unlink_data(c) ;
156     }
157     print_help("After unlinking data 9+i9:",a) ;
158 }
159
160 /**
161 list of pointers to complex
162 in complex constructor:
163 in complex constructor:
164 in complex constructor:
```

```
166 in complex constructor:  
167 in complex constructor:  
168 in complex constructor:  
169 in complex constructor:  
170 in complex constructor:  
171 in complex constructor:  
172 in complex constructor:  
173 After appending:0+i0 1+i1 2+i2 3+i3 4+i4 5+i5 6+i6 7+i7 8+i8 9+i9  
174 in complex constructor:  
175 In complex Destructor 0+i0  
176 In complex Destructor 2+i2  
177 In complex Destructor 4+i4  
178 In complex Destructor 6+i6  
179 In complex Destructor 8+i8  
180 After unlinking even items:1+i1 3+i3 5+i5 7+i7 9+i9  
181 After reverse:1+i1 3+i3 5+i5 7+i7 9+i9  
182 data 1+i1 is there in the slist. I am going to unlink it  
183 In complex Destructor 1+i1  
184 After unlinking data 1+i1 3+i3 5+i5 7+i7 9+i9  
185 After unlinking data 100+i100 3+i3 5+i5 7+i7 9+i9  
186 In complex Destructor 9+i9  
187 After unlinking data 9+i9:3+i3 5+i5 7+i7  
188 In complex Destructor 9+i9  
189 in slist distructor:  
190 In complex Destructor 3+i3  
191 In complex Destructor 5+i5  
192 In complex Destructor 7+i7  
193 -----*/  
194 static void test_slist_of_ptr_to_udt(){  
195     const int N = 10 ;  
196     slist<complex*> a(delete_complex ,complex_larger_compare);  
197     for (int i = 0; i < N; i++){  
198         complex* c = ALLOC(complex)(i,i) ;  
199         a.append(c) ;  
200     }  
201     print_help("After appending:",a) ;  
202     complex c;  
203     for (int i = 0; i < N; i=i + 2){  
204         c.setxy(i,i) ;  
205         a.unlink_data(&c) ;  
206     }  
207     print_help("After unlinking even items:",a) ;  
208     a.reverse() ;  
209     print_help("After reverse:",a) ;  
210     c.setxy(1,1) ;  
211     bool k = a.find(&c) ;  
212     if (k){  
213         cout << "data " << c << " is there in the slist. I am going to unlink it \n" ;  
214         a.unlink_data(&c) ;  
215     }else {  
216         cout << "data " << c << " is NOT there in the slist \n" ;  
217     }  
218     print_help("After unlinking data 1+i1 ",a) ;  
219     c.setxy(100,100) ;
```

```
221 k = a.find(&c) ;
222 if (k) {
223   a.unlink_data(&c) ;
224 }
225 print_help("After unlinking data 100+i100 ",a) ;
226 c.setxy(9,9) ;
227 k = a.find(&c) ;
228 if (k) {
229   a.unlink_data(&c) ;
230 }
231 print_help("After unlinking data 9+i9:",a) ;
232 }
233
234 /*-----*
235 -----*/
236 static void test_loop(){
237   const int N = 10 ;
238   slist<int> a(NULL,int_descending_order) ;
239   for (int i = 0; i < N; i++) {
240     a.append(i) ;
241   }
242   print_help("After appending:",a) ;
243   bool p = a.detect_loop() ;
244   if (p) {
245     cout << "There is a loop in the list \n" ;
246   }else {
247     cout << "There is NO loop in the list\n" ;
248   }
249 }
250
251 a.create_a_loop(7,1) ;
252 //print_help("After making a loop:",itt) ; -- LOOP FOR EVER
253 p = a.detect_loop() ;
254 if (p) {
255   cout << "There is a loop in the list \n" ;
256 }else {
257   cout << "There is NO loop in the list\n" ;
258 }
259 //PROGRAM WILL CRASH HERE.
260 }
261
262 /*-----*
263 main
264 -----*/
265 int main() {
266   test_slist_of_integers() ;
267   cout << "_____" << endl ;
268   test_slist_of_udt() ;
269   cout << "_____" << endl ;
270   test_slist_of_ptr_to_udt() ;
271   cout << "_____" << endl ;
272   test_loop() ; //Comment this to run completely
273   cout << "_____" << endl ;
274   return 0 ;
275 }
```


11.5 Problem set

VASUDEVAMURTHY

Chapter 12

Hash

12.1 Introduction

12.2 Why hash is required?

12.2. WHY HASH IS REQUIRED?

For 2010 Summer Semester

Please Choose Course:

Abbreviations: P: Presence A: Absence L: Coming Late E: Leaving Early N: Not Available

2010 Spring EE510B (Thur 6:00 pm)

#	Student Name	Program	Status	WK1	W1
1	Mr. Brahmabhatt, Jaimin Harmanay (8626)	MSEE	Credit	<input type="button" value="A"/>	<input type="button" value="A"/>
2	Mr. Das, Abhinav (7802)	BSEE	Credit	<input type="button" value="P"/>	<input type="button" value="A"/>
3	Mr. Ding, Beier (8675)	MSEE	Credit	<input type="button" value="P"/>	<input type="button" value="P"/>
4	Mr. Guntupalli, Raja Shekar (8250)	MSEE	Credit	<input type="button" value="P"/>	<input type="button" value="P"/>
5	Mr. Huang, Chiao Yu (8778)	MSEE	Credit	<input type="button" value="P"/>	<input type="button" value="P"/>
6	Mr. Li, Bo (8238)	MSEE	Credit	<input type="button" value="P"/>	<input type="button" value="P"/>
7	Mr. Lu, Yun (8133)	MSEE	Credit	<input type="button" value="A"/>	
8	Mr. Mubashir, Mohammed Najamuddin (8578)	MSEE	Credit	<input type="button" value="P"/>	
9	Mr. Poudel, Saurav (8696)	MSEE	Credit	<input type="button" value="A"/>	
10	Ms. Sandur, Nalina S (7942)	Non-D	Audit	<input type="button" value="P"/>	
11	Mr. Surasah, Srinivas Reddy (8776)	MSEE	Credit	<input type="button" value="P"/>	
12	Mr. Thakur, Nabeen Kumar (8688)	MSEE	Credit	<input type="button" value="A"/>	
13	Mr. Tran, Hien Van (8556)	MSEE	Credit	<input type="button" value="P"/>	
14	Mr. Vakulabharanam, Vamsi Krishna (8741)	MSEE	Credit	<input type="button" value="P"/>	

**I need to mark Student 8741.
I prefer number 14, rather than 8741**

Student #
Student IDNO #
Student Name

Figure 12.1: NPU EE510 class of 2010

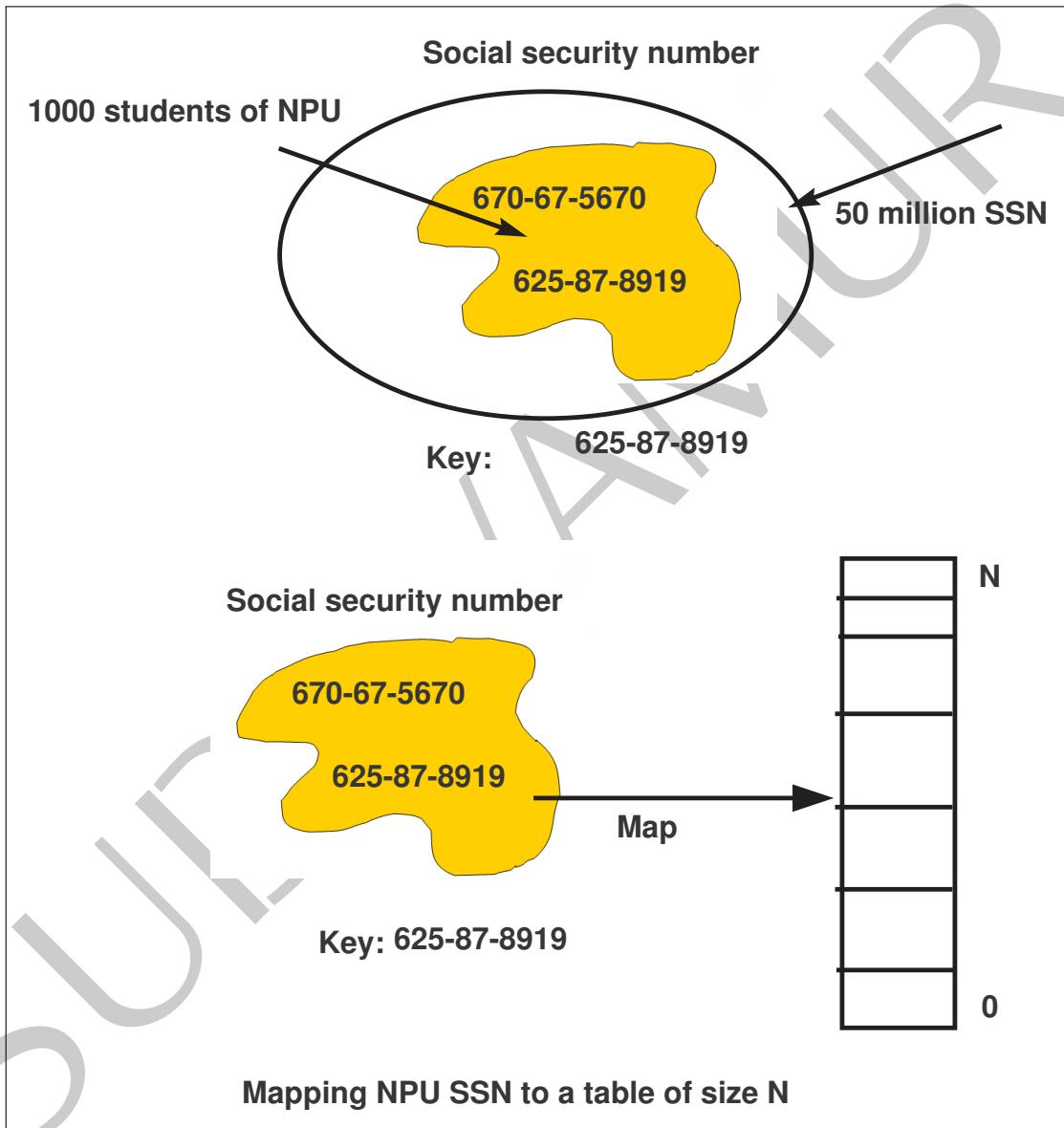


Figure 12.2: Why hash is required?

12.3 Example of an hash

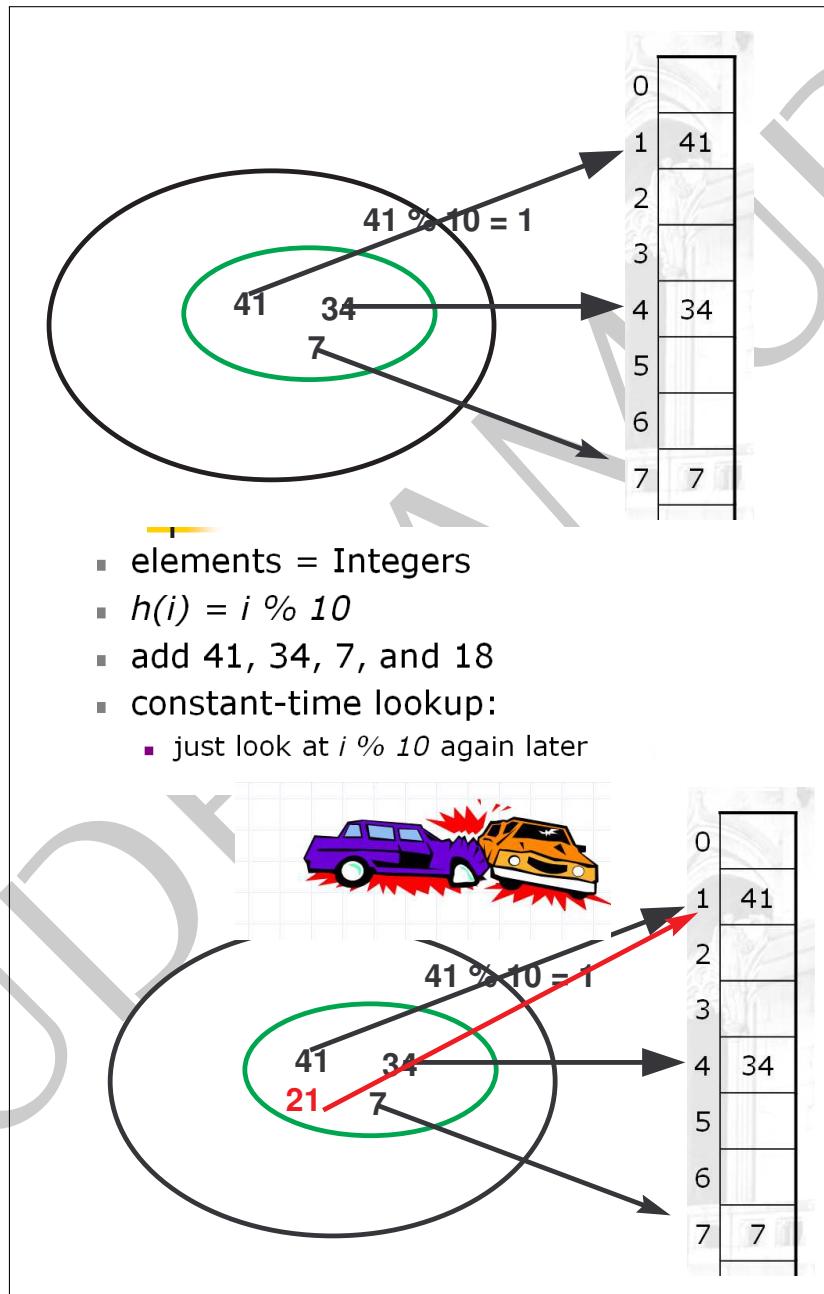


Figure 12.3: Hash function and collision

12.4. HASH FUNCTION AND COLLISION

12.4 Hash function and collision

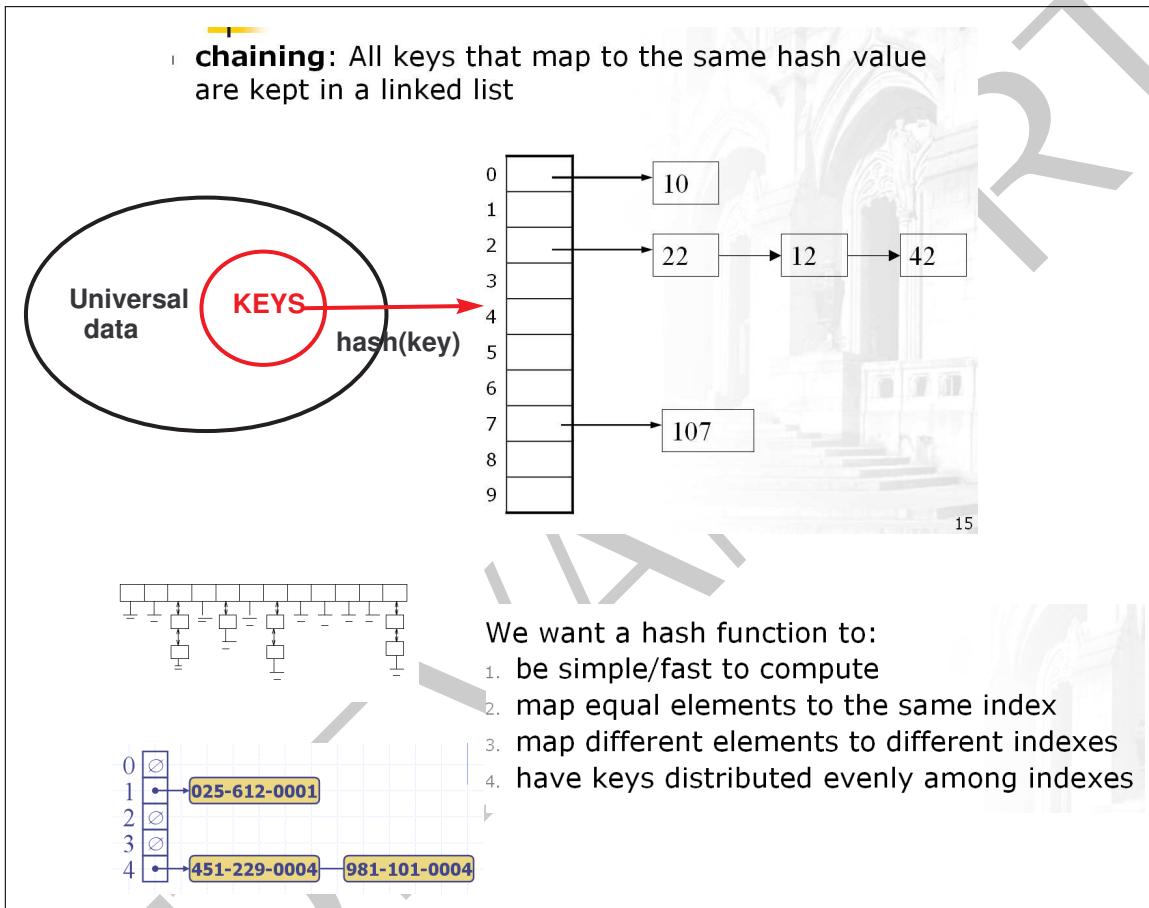


Figure 12.4: Hash function and collision handling by chaining

12.5 String hash function

String hash function

const char *s is the string
n is the length of s -> strlen(s)

$$\begin{array}{r} 3 \ 2 \ 1 \ 0 \\ \text{C} \ \text{A} \ \text{L} \ \text{L} \end{array}$$

in ASCII A=65, C=67 and L=76

$$\begin{aligned} & 3^3 \times 67 + 3^2 \times 65 + 3^1 \times 76 + 3^0 \times 76 \\ & = 1995997 + 62465 + 2356 + 76 \\ & = 2060894 \end{aligned}$$

$$31^{(n-1)} s[n-1] + 31^{(n-2)} s[n-2] + 31^{(n-3)} s[n-3] + 31^0 s[0]$$

$$\begin{aligned} & 3^3 \times 67 + 3^2 \times 65 + 3^1 \times 76 + 3^0 \times 76 \\ & = 76 + 31(76 + 31(65 + 31(67))) \\ & = 76 + 31(76 + 31(65 + 2077)) \\ & = 76 + 31(76 + 31(2142)) \\ & = 76 + 31(76 + 66402) \\ & = 76 + 31(66478) \\ & = 76 + 2060818 \\ & = 2060894 \end{aligned}$$

Horner's Rule

```
static int string_hash_func(char* const& s, int size) {
    unsigned h = 0;
    int n = strlen(s);
    for (int i = 0; i < n; i++) {
        unsigned r = s[i];
        h = 31 * h + r;
    }
    return (int(h % size));
}
```

Figure 12.5: String hash function

12.6. POINTER HASH FUNCTION

12.6 Pointer hash function

Pointer Hash Function

```
template <typename T>
int pointer_hash_func(T* const& ptr, int size) {
    char* const p = reinterpret_cast<char* const>(ptr);
    int x = string_hash_func(p, size);
    return x;
}
```

Figure 12.6: Pointer hash function

12.7 Integer hash function

Integer Hash Function

<http://www.concentric.net/~Ttwang/tech/inthash.htm>

32 bit mix functions

```
int integer_hash(int const& k, int size) {
    int key = k;
    key = ~key + (key << 15); // key = (key << 15) - key - 1;
    key = key ^ (key >> 12);
    key = key + (key << 2);
    key = key ^ (key >> 4);
    key = key * 2057; // key = (key + (key << 3)) + (key << 11);
    key = key ^ (key >> 16);
    return (key % size);
}
```

Figure 12.7: Integer hash function

12.8 Complete Code

```
1 /*-----  
2 Copyright (c) 2014 Author: Jagadeesh Vasudevamurthy  
3 file: dhash.h  
4 -----*/  
5  
6 /*-----  
7 This file has dhash class declaration  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #ifndef dhash_H  
14 #define dhash_H  
15  
16 #include "../util/util.h"  
17 #include "../slist/slist.h"  
18  
19 /*-----  
20 static definition - only once at the start  
21 Change to false, if you don't need verbose  
22 -----*/  
23 template <typename T>  
24 bool slist<T>::_display = false;  
25  
26 /*-----  
27 class dhash  
28 -----*/  
29 template <typename T>  
30 class dhash {  
31 public:  
32     dhash(int size = 100, int(*hf)(const T& c, int size) = nullptr, void(*df)(T& c) = nullptr, int(*cf)  
            (const T& c1, const T& c2) = nullptr);  
33     ~dhash();  
34     void insert(const T& data);  
35     bool find(const T& key) const;  
36     void delete_data(const T& data);  
37     void hash_statistics();  
38     int num_elements() const;  
39     /* no body can copy dhash or equal dhash */  
40     dhash(const dhash<T>& x) = delete;  
41     dhash& operator=(const dhash<T>& x) = delete;  
42     static void set_display(bool x) {  
43         slist<T>::set_display(x);  
44     }  
45  
46 private:  
47     int _tablesize;  
48     slist<T>* _hash;  
49  
50     int(*_pntr_to_hashfunc) (const T& c, int size);  
51     void(*_pntr_to_func_to_delete_data) (T& c);  
52     int(*_pntr_to_compare_func) (const T& c1, const T& c2);  
53     void _free_hash_table();  
54 };  
55  
56 #include "dhash.hpp"  
57  
58 #endif  
59 //EOF  
60  
61
```

12.8. COMPLETE CODE

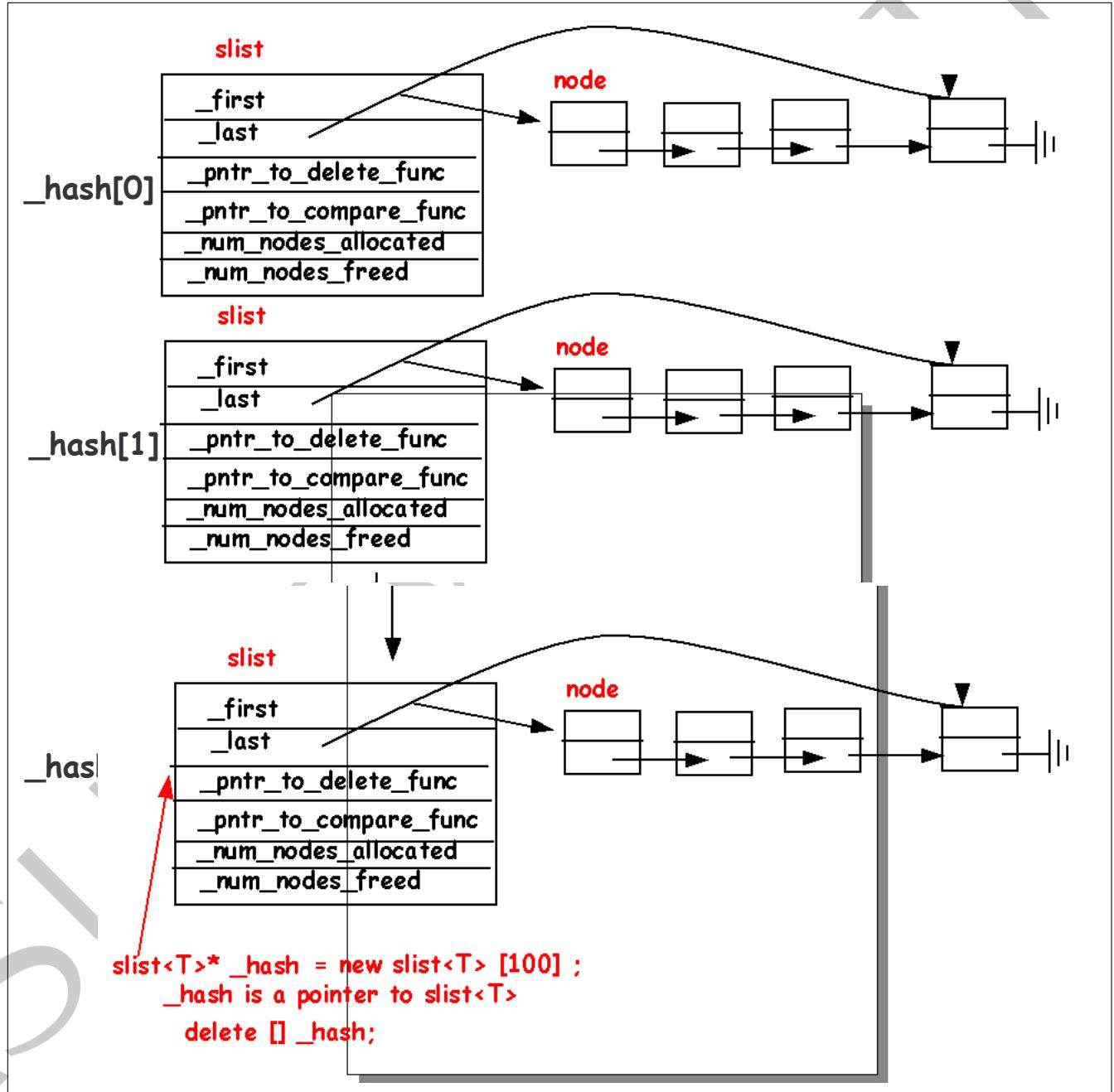


Figure 12.8: Hash data structure using array of `slist`

```
1 /*-
2 Copyright (c) 2014 Author: Jagadeesh Vasudevamurthy
3 file: dhash.hpp
4
5 -----*/
6
7 /*
8 This file has class definition
9 -----*/
10
11 /*
12 Definition of routines of dhash class
13 -----*/
14
15 /*
16 d hash Constructor
17 -----*/
18 template <typename T>
19 dhash<T>::dhash(int size, int(*hf)(const T& c, int size), void(*df)(T& c), int(*cf)(const T& c1, const
20 T& c2)) :
21 _tablesize(size), _ptr_to_hashfunc(hf), _ptr_to_func_to_delete_data(df),
22 _ptr_to_compare_func(cf)
23 {
24     _hash = new slist<T>[size];
25     for (int i = 0; i < size; i++) {
26         slist<T>& x = _hash[i];
27         x.change_functions(df, cf);
28     }
29 }
30 /*
31 dhash Distructor
32 -----*/
33 template <typename T>
34 dhash<T>::~dhash() {
35     delete[] _hash;
36 }
37
38 /*
39 Returns number of elements in the heap
40 -----*/
41 template <typename T>
42 int dhash<T>::num_elements() const {
43     int t = 0;
44     for (int i = 0; i < _tablesize; i++) {
45         slist<T>& x = _hash[i];
46         int y = x.size();
47         t = t + y;
48     }
49     return t;
50 }
51
52 /*
53 print statistic of dhash
54 -----*/
55 template <typename T>
56 void dhash<T>::hash_statistics() {
57     int total = 0;
58     int max = 0;
59     int min = _tablesize;
60     int std = 0;
61     int tot = num_elements();
62     int avr = tot / _tablesize;
63     for (int i = 0; i < _tablesize; i++) {
64         slist<T>& x = _hash[i];
65         int y = x.size();
```

```
66     if (y > max) {
67         max = y;
68     }
69     if (y < min) {
70         min = y;
71     }
72     int m = (y - avr);
73     int m2 = m * m;
74     std = std + m2;
75     cout << "BUCKET " << i << " has " << y << " elements " << endl;
76     total = total + y;
77 }
78 assert(tot == total);
79 cout << "TOTAL ELEMENTS      = " << total << endl;
80 cout << "NUM buckets        = " << _tablesize << endl;
81 cout << "IDEAL num in a buket = " << (total / _tablesize) << endl;
82 cout << "MAX in a buket      = " << max << endl;
83 cout << "MIN in a buket      = " << min << endl;
84 int z = std / total;
85 int zsd = int(sqrt(double(z)));
86 cout << "Standad deviation    = " << zsd << endl;
87 }
88
89 /*-----
90 Insert item to hash table
91 -----*/
92 template <typename T>
93 void dhash<T>::insert(const T& data) {
94     int x = _pntr_to_hashfunc(data, _tablesize);
95     slist<T>& s = _hash[x];
96     s.append(data);
97 }
98
99 /*-----
100 Get the inserted data if matches.
101 Else return NIL.
102
103 Note that the data is still in hash table.
104 -----*/
105 template <typename T>
106 bool dhash<T>::find(const T& key) const{
107     int x = _pntr_to_hashfunc(key, _tablesize);
108     slist<T>& list = _hash[x];
109     bool s = list.find(key);
110     return s;
111 }
112
113
114 //EOF
115
116
```

```
1 /*-
2 Copyright (c) 2014 Author: Jagadeesh Vasudevamurthy
3 file: dhashtest.cpp
4 you require: ../util/util.cpp dhashtest.cpp
5 On linux:
6 g++ ../util/util.cpp dhashtest.cpp
7 valgrind a.out
8 -- All heap blocks were freed -- no leaks are possible
9 -----
10 */
11 /*
12 This file test dhash object
13 -----
14 */
15 /*
16 All includes here
17 -----
18 #include "dhash.h"
19 */
20 /*
21 local to this file. Change verbose = true for debugging
22 -----
23 static bool verbose = false;
24 */
25 /*
26 43210
27 hello:
28 unicode h = 104 e = 101 l = 108 l = 108 o = 111
29
30 104 * 31^4 + 101. 31^3 + 108. 31^2 + 108.31 + 111 = 99162322
31 -----
32 static int string_hash_func(char* const& s, int size) {
33     unsigned h = 0;
34     int n = strlen(s);
35     for (int i = 0; i < n; i++) {
36         unsigned r = s[i];
37         h = 31 * h + r;
38     }
39     return (int(h % size));
40 }
41 */
42 /*
43 Pointer to any object T
44
45 convert pointer to any object T to char * pointer.
46 Use the above function
47
48 The reinterpret_cast operator also allows any integral type to be converted into
49 any pointer type and vice versa. Misuse of the reinterpret_cast operator can easily be unsafe.
50 Unless the desired conversion is inherently low-level, you should use one of the other cast operators.
51
52 The reinterpret_cast operator can be used for conversions such as char* to int*,
53 or One_class* to Unrelated_class*, which are inherently unsafe.
54
55 The result of a reinterpret_cast cannot safely be used for anything other than being cast back
56 to its original type. Other uses are, at best, nonportable
57
58 -----
59 template <typename T>
60 int pointer_hash_func(T* const& ptr, int size) {
61     char* const p = reinterpret_cast<char*>(ptr);
62     int x = string_hash_func(p, size);
63     return x;
64 }
65 */
66 */
```

```
67 integer hash
68 -----
69 int integer_hash(int const& k, int size) {
70     int key = k;
71     key = ~key + (key << 15); // key = (key << 15) - key - 1;
72     key = key ^ (key >> 12);
73     key = key + (key << 2);
74     key = key ^ (key >> 4);
75     key = key * 2057; // key = (key + (key << 3)) + (key << 11);
76     key = key ^ (key >> 16);
77     return (key % size);
78 }
79
80 -----
81 test integers by value
82 -----
83 static void test_integers_by_value() {
84     dhash<int> h(1000, integer_hash, NULL, intAscendingOrder);
85     dhash<int>::set_display(verbose);
86     Random r;
87     for (int i = 0; i < 10000; i++) {
88         int num = r.getRandomNumber(0, 9999999);
89         h.insert(num);
90     }
91     h.hash_statistics();
92     int how_many = 9999;
93     int num_found = 0;
94     int num_not_found = 0;
95     for (int i = 0; i < how_many; i++) {
96         int num = r.getRandomNumber(0, 9999999);
97         bool p = h.find(num);
98         if (p) {
99             ++num_found;
100            //cout << num << " found\n" ;
101        }
102        else {
103            ++num_not_found;
104            //cout << num << " not found\n" ;
105        }
106    }
107    cout << "in 9999999 numbers " << " found = " << num_found << " not found = " << num_not_found << endl;
108 }
109
110 -----
111 test integers by pointers
112 -----
113 static void test_integers_by_pointers() {
114     const int MAX = 10;
115     dhash<int*> h(100, pointerHashFunc, deleteInt, intAscendingOrder);
116     dhash<int*>::set_display(verbose);
117     int *ptarray[MAX];
118     int valarray[MAX];
119     int j = 0;
120     Random r;
121     for (int i = 0; i < 1000; i++) {
122         int num = r.getRandomNumber(0, 9999999);
123         int *p = new int(num);
124         h.insert(p);
125         if (i % 7) {
126             if (j < MAX) {
127                 valarray[j] = num;
128                 ptarray[j++] = p;
129             }
130         }
131     }
```

```
132     h.hash_statistics();
133
134     for (int i = 0; i < j; i++) {
135         bool p = h.find(ptarray[i]);
136         if (p) {
137             cout << " found\n";
138         }
139         else {
140             cout << " not found\n";
141         }
142     }
143 }
144
145 /*-----*/
146 test string
147 -----*/
148 static void test_strings() {
149     dhash<char *> h(1000, string_hash_func, delete_charstar, NULL);
150     dhash<char *>::set_display(verbose);
151 #ifdef _WIN32
152     ifstream in("C:/work/alg/course/code/c/data/english.dat");
153 #else
154     ifstream in("/home/jag/jag/alg/c/data/english.dat");
155 #endif
156
157     assert(in);
158     char buffer[1024];
159     int i = 0;
160     while (!in.getline(buffer, 1024).eof()) {
161         int l = strlen(buffer);
162         char *s = new char[l + 1];
163         strcpy(s, buffer);
164         h.insert(s);
165     }
166     h.hash_statistics();
167 }
168
169
170 /*-----*/
171 main
172 -----*/
173 int main() {
174     test_integers_by_value();
175     cout << "-----" << endl;
176     test_integers_by_pointers();
177     cout << "-----" << endl;
178     test_strings();
179     return 0;
180 }
181
182
183 //EOF
184
185
```

12.9. PROBLEM SET

12.9 Problem set

Problem 12.9.1. Refer to

<http://www.cplusplus.com/reference/stl/set/>

Use STL **set** class to insert and delete **flight** class objects by **value** and by **address**. Use **time** as the key. Try to use as many public functions as possible.

Problem 12.9.2. Refer to

<http://www.cplusplus.com/reference/stl/map/>

Use STL **map** class to insert and delete **flight** class objects by **value** and by **address**. Use **time** as the key and all other data as values. Try to use as many public functions as possible.

Chapter 13

STL hash

13.1 Introduction

13.2 STL unordered_set

Unordered_set

Unordered sets are containers that store unique elements in no particular order, and which allow for fast retrieval of individual elements based on their value.

Internally, the elements in the unordered_set are not sorted in any particular order, but organized into buckets depending on their hash values to allow for fast access to individual elements directly by their values (with a constant average time complexity on average).

```
/* -----  
A pair of int as an unit  
----- */  
  
class intint {  
public:  
    friend struct intintHasher ; //intintHasher can access private intint  
    intint(int f = 0, int s = 0) :_f(f), _s(s) {}  
    ~intint() {}  
  
    friend bool operator == (const intint& left, const intint& right) {  
        return (left._f == right._f);  
    }  
  
private:  
    int _f;  
    int _s;  
};  
  
/* -----  
Hash function for intint  
----- */  
  
struct intintHasher {  
    size_t operator()(const intint& t) const {  
        int key = (t._f + t._s) ;  
        key = ~key + (key << 15); // key = (key << 15) - key - 1;  
        key = key ^ (key >> 12);  
        key = key + (key << 2);  
        key = key ^ (key >> 4);  
        key = key * 2057; // key = (key + (key << 3)) + (key << 11);  
        key = key ^ (key >> 16);  
        return key ;  
    }  
  
    /* -----  
    assert all pairs of a and all pairs of b are same  
    a: {4, 6} {1, 9}  
    b: {1, 9} {4, 6}  
    ----- */  
  
    void asserts_arrays_are_equal(vector<int>& a, vector<int>& b) {  
        assert(a.size() == b.size());  
        unordered_set<intint, intintHasher> s ;  
        int n = a.size();  
        //Insert every pair of A in set  
        for (int i = 0 ; i < n-1; i = i + 2) {  
            intint u(a[i], a[i+1]);  
            auto it = s.find(u);  
            //This should not be there  
            assert(it == s.end());  
            s.insert(u);  
        }  
        //Now make sure every pair of B is there  
        for (int i = 0; i < n - 1; i = i + 2) {  
            intint u(b[i], b[i + 1]);  
            auto it = s.find(u);  
            //This should be there  
            assert(it != s.end());  
            s.erase(it);  
        }  
        //Set should be empty now  
        assert(s.size() == 0);  
    }  
}
```

13.3 STL `unordered_map`

Unordered_map

Unordered maps are associative containers that store elements formed by the combination of a key value and a mapped value, and which allows for fast retrieval of individual elements based on their keys.

Internally, the elements in the `unordered_map` are not sorted in any particular order with respect to either their key or mapped values, but organized into buckets depending on their hash values to allow for fast access to individual elements directly by their key values (with a constant average time complexity on average).

CHAPTER 13. STL HASH

Problem 13.3.1. Remove redundant lines from the input data file and write a new output file. Write a function to make sure that the new file has no redundant data.

13.3. STL UNORDERED_MAP

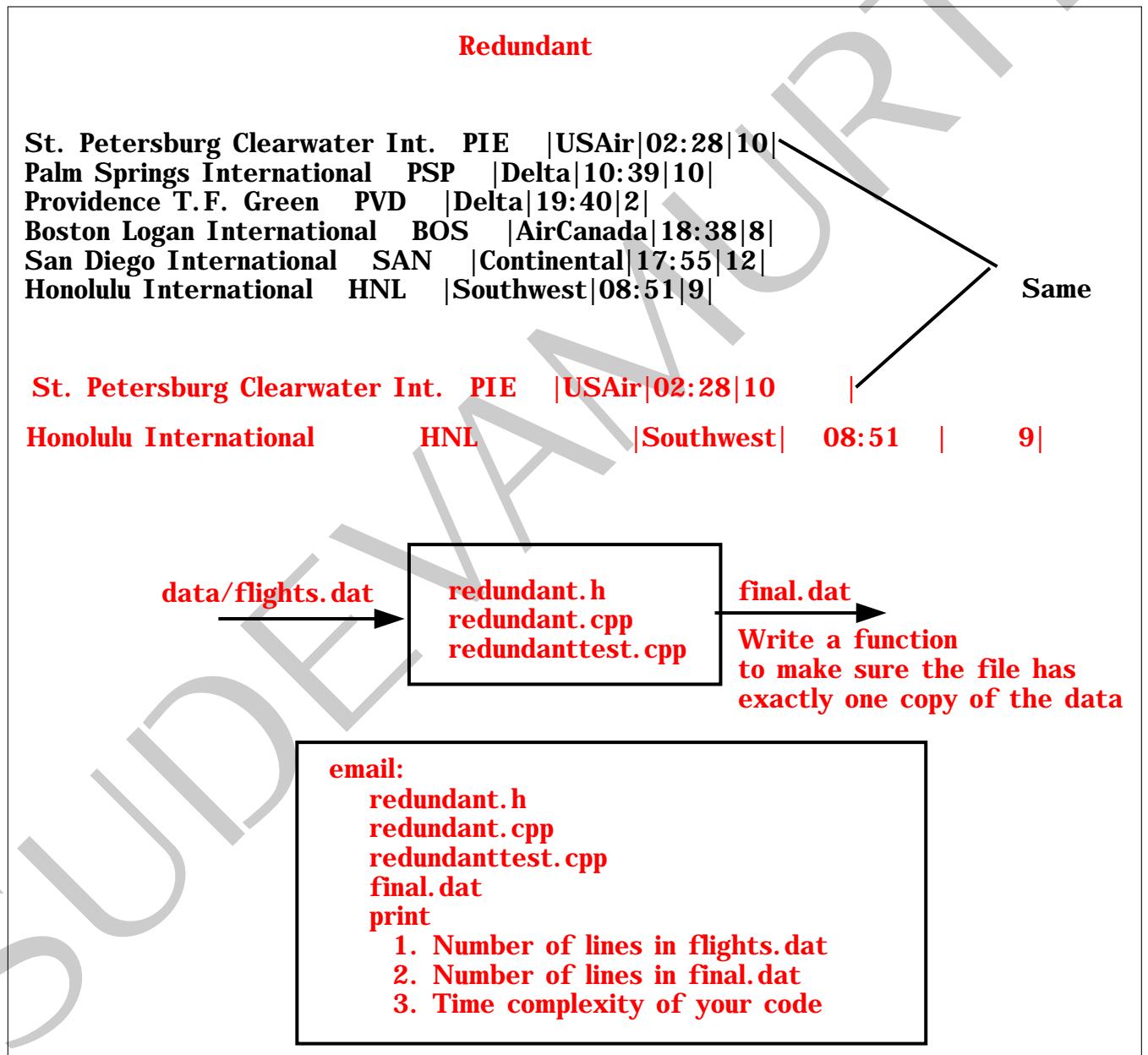


Figure 13.1: Remove redundant lines

Chapter 14

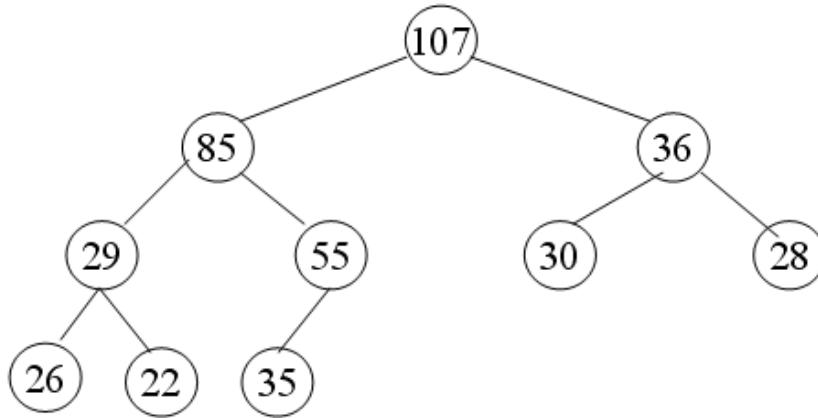
Heap

14.1 Introduction

14.2 What is heap?

Heap

- A heap T is a complete Binary tree in which either T is empty or
- each item in $\text{Left}(T)$ is \leq Root item of T
- each item in $\text{Right}(T)$ is \leq Root item of T
- Left and Rights are heaps



- The ordering in a heap is *top-down, but not left or right*. Each root item is greater or equal to each of its children, but some left siblings may be greater than their right siblings and some be less. For example ($85 > 36$ but $29 < 55$)

Figure 14.1: Definition of heap

14.3 Representation of heap

Representation of Heap as Array

- $\text{Parent} \geq \text{child}$
- $\text{Parent}(I) = I/2. \text{ If } (I \leq 1 \text{ or } I > N) \text{ Parent} = \text{NIL}$
- $\text{Left}(I) = I * 2. \text{ If } (I > N) \text{ Left} = \text{NIL};$
- $\text{Right}(I) = I * 2 + 1. \text{ If } (I > N) \text{ Right} = \text{NIL};$

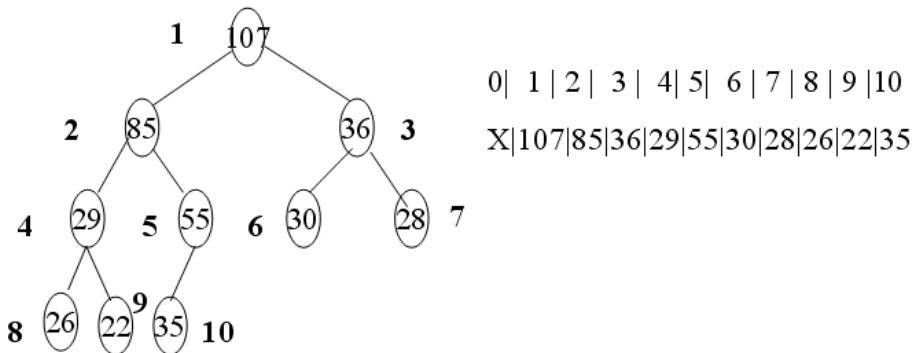


Figure 14.2: Heap as an array

14.4 Operation 1: Finding minimum or maximum

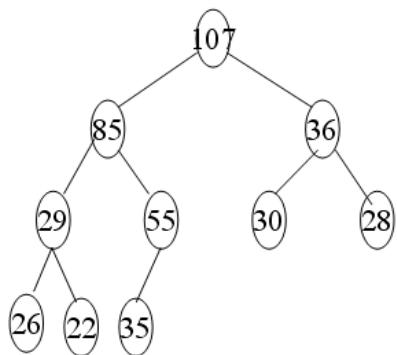
14.5. OPERATION 2: ADDING AN ELEMENT TO HEAP

Operation 1: Finding Min or Max

Finding Maximum or Minimum

- Return Heap[1]

O(1)



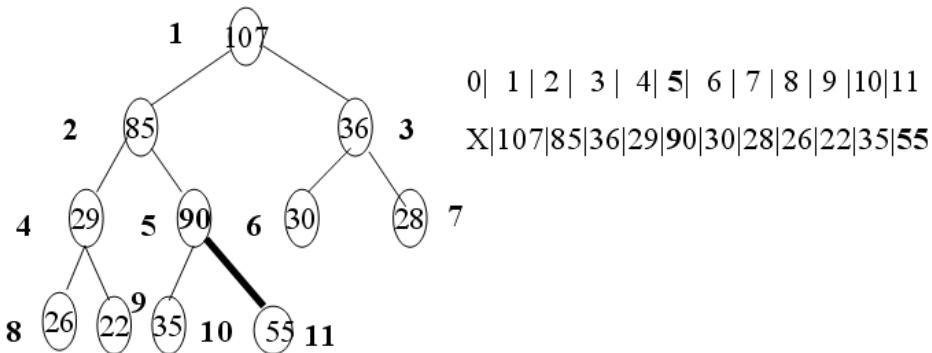
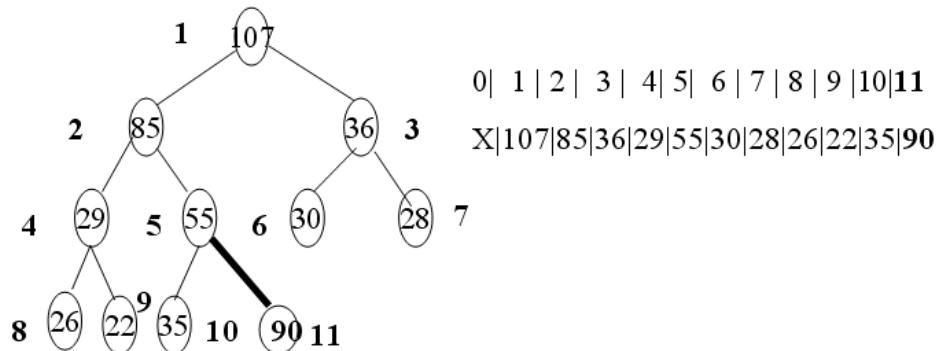
0| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
X|107|85|36|29|55|30|28|26|22|35

Figure 14.3: Minimum or maximum

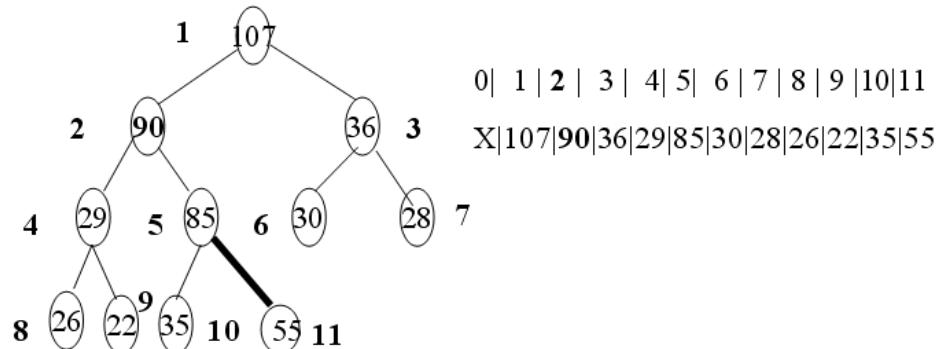
14.5 Operation 2: Adding an element to heap

Insert An Element To Heap

- Add an element with value 90



Go from Bottom to top on the leg of a tree until you can swap



Go from Bottom to top on the leg of a tree until you can swap

Figure 14.4: Adding an element to heap

14.6 Operation 3: Deleteing an element to heap

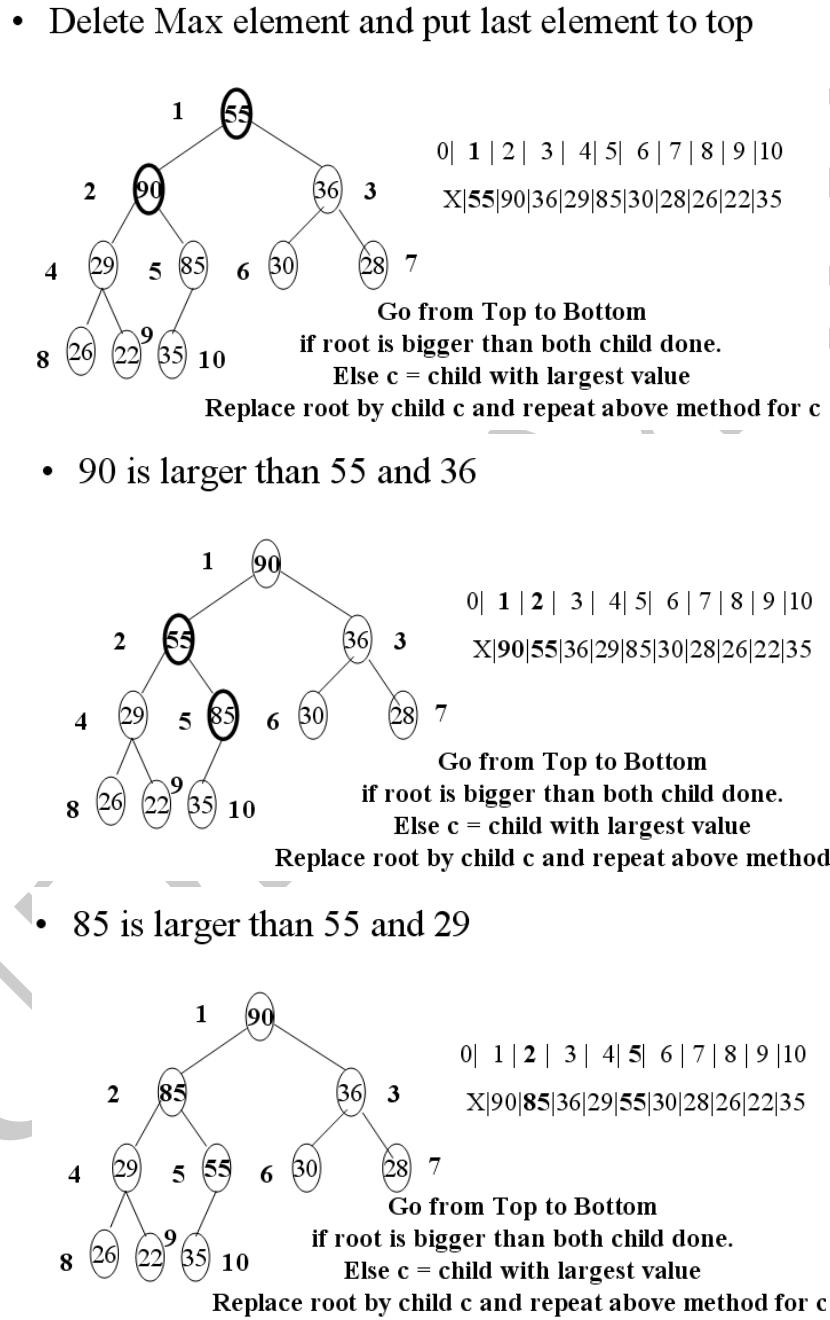


Figure 14.5: Deleting an element to heap

14.7 Heap sort

Heap Sort

- Let us say we have unordered N elements
- for ($I = 0; I < N; I++$) Insert $\text{element}(I)$ to Heap H
- The top element of the heap H has the max element
- Keep on deleting the element from the Heap H until H is empty

Figure 14.6: Heap sort

14.8 Visualization of data structures

How to get Graphviz package

Graphviz - Graph Visualization Software

<http://www.graphviz.org/>

1. Download
2. Windows
[Stable and development Windows Install packages](#)
3. Windows [graphviz-2.18.exe](#)
3. C:\Program Files\Graphviz2.18\Bin/dot.exe
4. You need not have to add in path

TO RUN

```
dot -Tps junk -o graph.ps  
dot -Tpdf junk > graph.pdf
```

Figure 14.7: Getting Graphviz package

14.9. VISULIZATION OF HEAP DATA STRUCTURES

14.9 Visualization of HEAP data structures

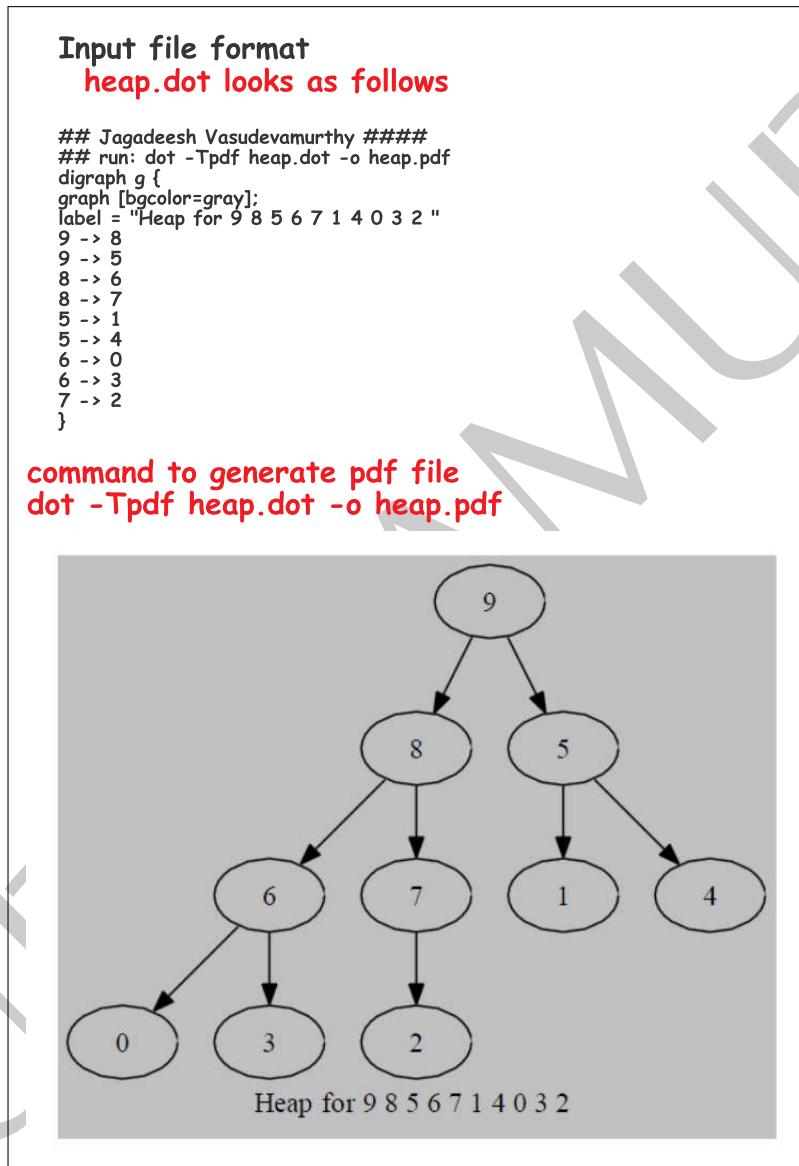


Figure 14.8: Visulization of heap

14.10 Complete Code

```
1 /*-
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy
3 Filename: dheap.h
4 */
5 #ifndef dheap_h
6 #define dheap_h
7
8 #include "util.h"
9 #include "darray.h"
10
11 /*
12 class dheap
13
14 A heap T is a almost complete binary tree in which
15 T is either empty or
16
17 1. Each item in Left T is <= root item of T
18 2. Each item in Right T is <= root item of T
19 3. Left and right trees are heap
20
21 The ordering in a heap is top down, but not left to right.
22 Right can > left and left can be > right
23
24 0 th position of the array is not used in our implementation
25 */
26 template <typename T>
27 class dheap {
28 public:
29     dheap(void(*df)(T& c) = NULL, int(*cf)(const T& c1, const T& c2) = NULL);
30     ~dheap();
31     void insert(const T& data);
32     bool get_top(T& a); // if dheap exists: top of the dheap is copied to a and returned. true
33     // else false and a is unchanged
34
35     void change_compare_function(int(*cf)(const T& c1, const T& c2)) {_pntr_to_compare_func = cf ;}
36     bool display()const {return _display;}
37
38 private:
39     int _kount; //0 is temp. 1..kount (including kount, real elements
40     darray<T> _dheap;
41
42     void(*_pntr_to_func_to_delete_data)(T& c) ;
43     int(*_pntr_to_compare_func)(const T& c1, const T& c2);
44
45
46     int _parent(int i) const {return i/2; }
47     int _left(int i) const {return 2 * i; }
48     int _right(int i) const {return 2 * i + 1; }
49     void _heapify_down_to_up();
50     void _heapify_up_to_down();
51     void _swap(int f, int s);
52     bool _compare(int f, int s);
53     void _free_user_data(int i);
54     template <typename U> friend ostream& operator<<(ostream& o, const dheap<U>& t);
```

```
56 //NOTE U above. Will not compile on Linux with T
57 static bool _display; /* ONLY ONCE for all object */
58
59 /* no body can copy dheap or equal dheap */
60 dheap(const dheap<T>& x);
61 dheap& operator=(const dheap<T>& x);
62 };
63
64 /*-
65 dheap Constructor
66 -----*/
67 template <typename T>
68 dheap<T>::dheap(void(*df)(T& c),int(*cf)(const T& c1, const T& c2)):
69 _kount(0),_pntr_to_func_to_delete_data(df),_pntr_to_compare_func(cf)
70 {
71 if (display()){
72 cout << "in dheap constructor:" << endl ;
73 }
74 }
75
76 /*-
77 free user data
78 -----*/
79 template <typename T>
80 void dheap<T>::_free_user_data(int i){
81 if (_pntr_to_func_to_delete_data){
82 assert(i >= 0 && i <= _kount);
83 _pntr_to_func_to_delete_data(_dheap[i]);
84 }
85 }
86
87
88 /*-
89 dheap Distructor
90 -----*/
91 template <typename T>
92 dheap<T>::~dheap(){
93 if (display()){
94 cout << "in dheap disstractor:" << endl ;
95 }
96 //WRITE YOUR CODE HERE
97 }
98
99 /*-
100 Print a tree as dot file
101
102 ## Jagadeesh Vasudevamurthy #####
103 ## run: dot -Tpdf heap.dot -o heap.pdf
104 digraph g {
105 label = "Heap for 9 8 5 6 7 1 4 0 3 2 "
106 9 -> 8
107 9 -> 5
108 8 -> 6
109 8 -> 7
110 5 -> 1
```

```
111 5 -> 4
112 6 -> 0
113 6 -> 3
114 7 -> 2
115 }
116
117 -----
118 template <typename T>
119 ostream& operator<<(ostream& o, const dheap<T>& q){
120   o << "## Jagadeesh Vasudevamurthy #####" << endl ;
121   o << "## run: dot -Tpdf heap.dot -o heap.pdf" << endl ;
122
123   o << "digraph g {" << endl ;
124   o << "graph [bgcolor=gray];" << endl ;
125
126   o << "label = \"Heap for " ;
127
128 //WRITE YOUR CODE HERE
129
130 }
131
132
133 /*-
134 swap
135 Uses darray location 0 as a temp location
136 contents of s and f are swaped
137 -----
138 template <class T>
139 void dheap<T>::_swap(int f, int s){
140   _dheap[0] = _dheap[f]; /* IF T is by value equal constructor is called */
141   _dheap[f] = _dheap[s];
142   _dheap[s] = _dheap[0];
143 }
144
145 /*-
146 is bigger
147 if data(f) < data(s) return true
148 -----
149 template <class T>
150 bool dheap<T>::_compare(int f, int s) {
151   //WRITE YOUR CODE HERE
152   return false; //WRONG
153 }
154
155 /*-
156 insert d to dheap maintaining dheap property
157 0 th position of the array is not used
158 -----
159 template <class T>
160 void dheap<T>::insert(const T& d){
161   //WRITE YOUR CODE HERE
162 }
163
164 /*-
165 0 th position of the array is not used
```

```
166 get the top of the element from position 1 by alias
167
168 The top object is COPIED to user object a and returned
169 It is the user responsibility to delete that object
170
171 After giving the top, dheap will maintain its property
172 -----
173 template <class T>
174 bool dheap<T>::get_top(T& a) {
175 //WRITE YOUR CODE HERE
176 return false; //WRONG
177 }
178
179 /*-
180 0 th position of the array is not used
181 Move the _kount element(bottom) to the top until swap is possible
182 INSERTION
183 -----
184 template <class T>
185 void dheap<T>::_heapify_down_to_up() {
186 //WRITE YOUR CODE HERE
187 }
188
189
190 /*-
191 0 th position of the array is not used
192 Move the _kount element(bottom) to the top until swap is possible
193 DELETION
194 -----
195 template <class T>
196 void dheap<T>::_heapify_up_to_down() {
197 //WRITE YOUR CODE HERE
198 }
199
200 #endif
201
```

```
1 /*-----  
2 Copyright (c) 2010 Author: Jagadeesh Vasudevanurthy  
3 Filename: dheap.cpp  
4 compile: g++ dheap.cpp  
5  
6 Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)  
7 -----*/  
8  
9 #include "dheap.h"  
10  
11 /*-----  
12 static definition - only once at the start  
13 Change to false, if you don't need verbose  
14 -----*/  
15 template <typename T>  
16 bool darray<T>::_display = true ;  
17  
18 template <typename T>  
19 bool dheap<T>::_display = true ;  
20  
21 /*-----  
22 dheap of integers  
23 -----*/  
24 static void test_integers_dheap(){  
25     dheap<int> h(NULL,int_descending_order);  
26     for (int i = 0; i < 10; i++) {  
27         h.insert(i);  
28     }  
29     cout << h << endl ;  
30  
31     cout << "Sorted array in descending order " << endl ;  
32     int x ;  
33     while (h.get_top(x)) {  
34         cout << x << " ";  
35     }  
36     cout << endl ;  
37  
38     int array[] = {10,7,7,9,21,1,4,45,0,-1,-3,-5};  
39     int size = sizeof(array)/sizeof(int) ;  
40     int j = sizeof(array)/sizeof(int) ;  
41     h.change_compare_function(int_ascending_order);  
42  
43     for (int i = 0; i < j; i++) {  
44         h.insert(array[i]);  
45     }  
46     cout << h << endl ;  
47  
48     cout << "Sorted array in ascending order" << endl ;  
49     while (h.get_top(x)) {  
50         cout << x << " ";  
51     }  
52     cout << endl ;  
53 }  
54  
55 /*-----
```

```
56 dheap of strings
57 -----
58 static void test_string_dheap(){
59 {
60     char* names[] = {"idiot", "zoo", "henry", "tom", "dick", "harry"};
61     int size = sizeof(names)/sizeof(char *);
62     dheap<char*> h(NULL,string_descending_order);
63     for (int i = 0; i < size; i++) {
64         h.insert(names[i]);
65     }
66     cout << h << endl;
67
68     cout << "Sorted array in descending order" << endl;
69     char* x = NULL ;
70     while (h.get_top(x)) {
71         cout << x << " ";
72     }
73     cout << endl;
74 }
75 {
76     char* names[] = {"idiot","zoo","henry","tom","dick","harry"};
77     int size = sizeof(names)/sizeof(char *);
78     dheap<char*> h(NULL,string_ascending_order);
79     for (int i = 0; i < size; i++) {
80         h.insert(names[i]);
81     }
82     cout << h << endl;
83
84     cout << "Sorted array in ascending order" << endl;
85     char* x = NULL ;
86     while (h.get_top(x)) {
87         cout << x << " ";
88     }
89     cout << endl;
90 }
91 }
92 /**
93 main
94 -----
95 int main(){
96     test_integers_dheap();
97     cout << "_____ " << endl;
98     test_string_dheap();
99     cout << "_____ " << endl;
100    return 0;
101 }
102 }
103 }
```

14.11 Problem set

Problem 14.11.1. Refer to *heap.h* and *heap.cpp* files in the section 14.10. Write the code, in the place marked as **WRITE CODE HERE**. You should also generate the **dot files**. Read section 14.9 for an example. Attach the **pdf** files as your solution along with the code.

VASUDEVAMURTHY

Chapter 15

Tree

15.1 Introduction

15.2 Tree

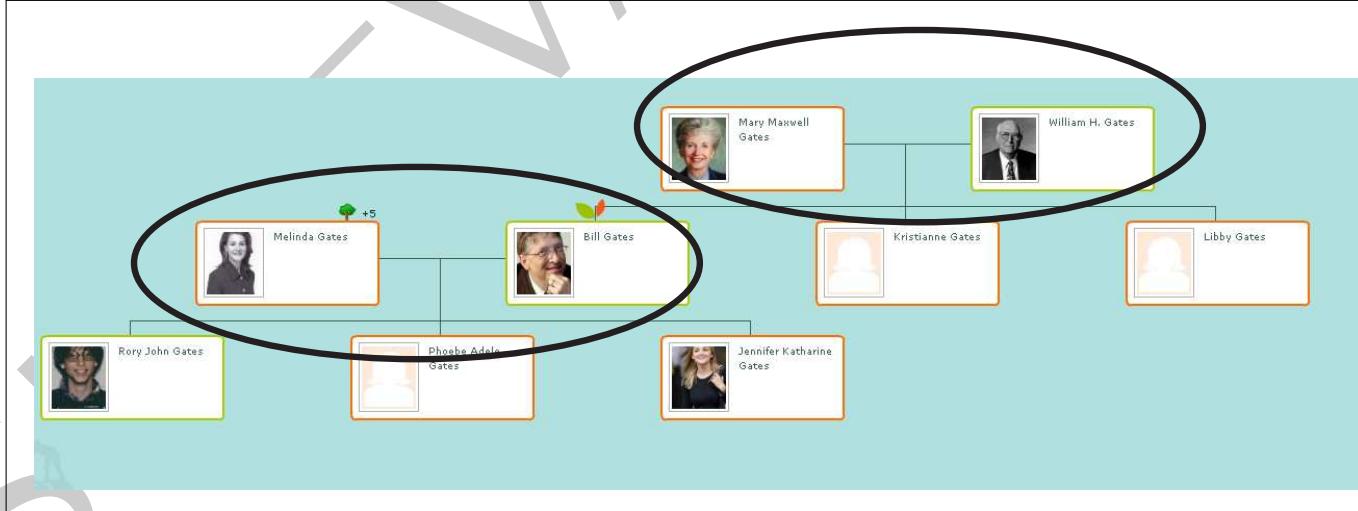


Figure 15.1: Family tree of Bill Gates

15.3. BINARY TREE

Example: UNIX Directory

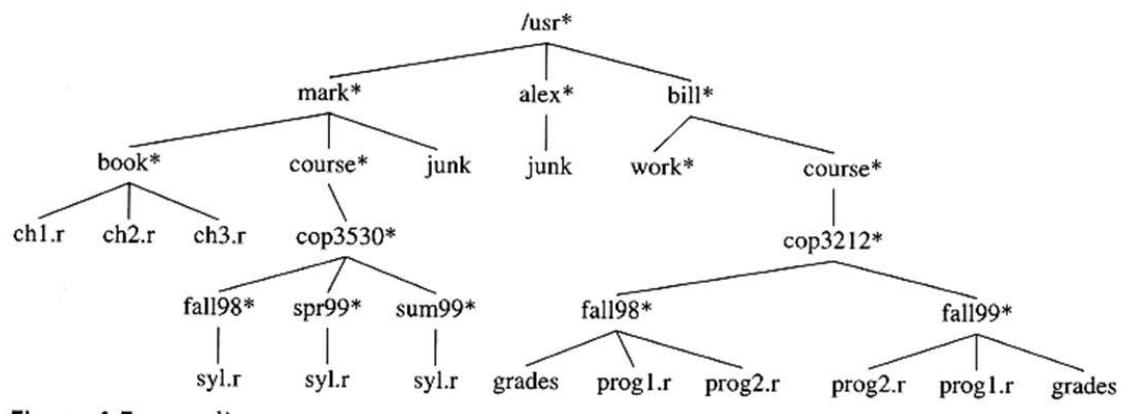


Figure 15.2: An example of a tree

15.3 Binary tree

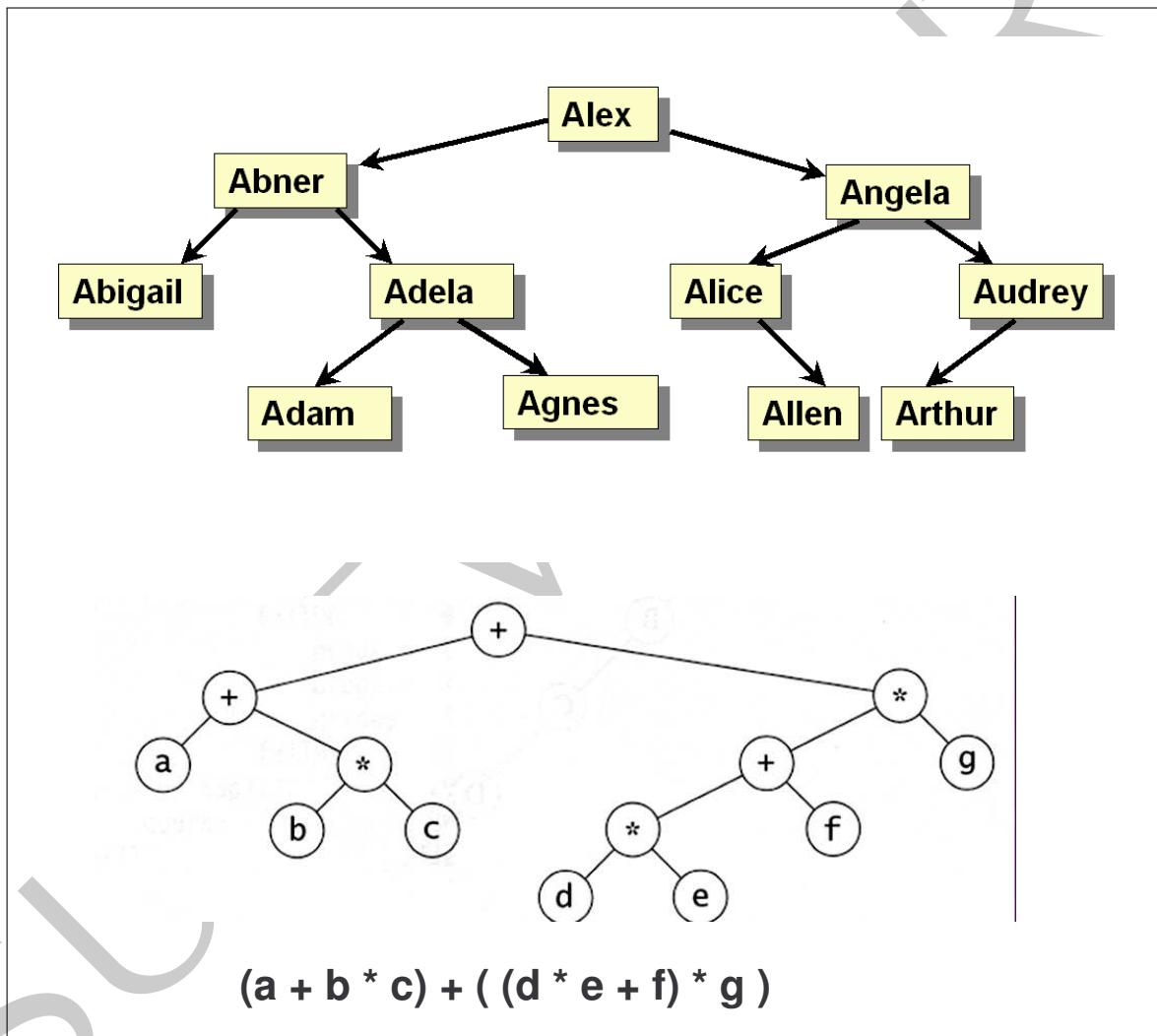
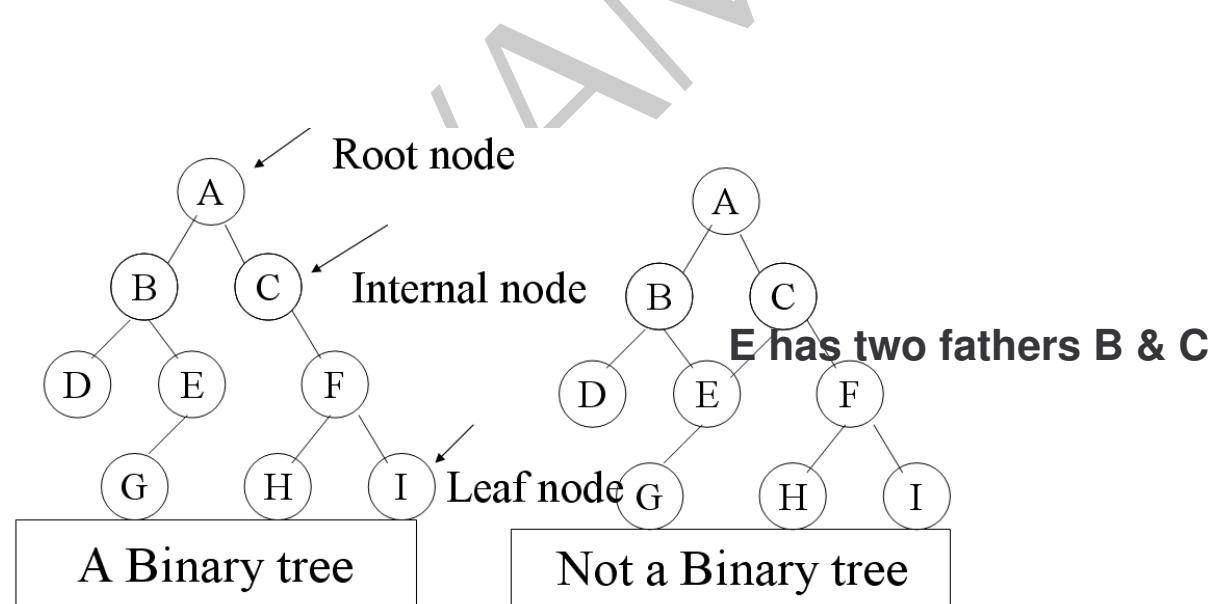
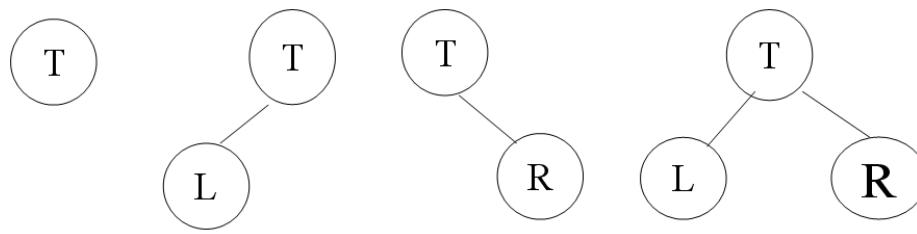


Figure 15.3: An example of a binary tree

15.4 Definition of a binary tree

- A Binary Tree T is either empty or consists of an item, called the *root* item, and two disjoint binary trees, called the *left subtree* and *right subtree* of T .



Leaf: Node that has no child, Root: Node that has no parent.

Internal node: Node has parent and at least one child

Root node : A
Internal node: B,C,F,F
Leaves: D,G,H,I

Figure 15.4: Definition of a binary tree

15.5 Visualizations Binary Tree

```

# Filename: tree.dot
# dot -Tpdf tree.dotdot -o tree.pdf
graph TD
node [shape = record,height=.1];
graph [bgcolor=gray];

node0[label = "<f0> |<f1> G|<f2> "];
node1[label = "<f0> |<f1> E|<f2> "];
node2[label = "<f0> |<f1> B|<f2> "];
node3[label = "<f0> |<f1> F|<f2> "];
node4[label = "<f0> |<f1> R|<f2> "];
node5[label = "<f0> |<f1> H|<f2> "];
node6[label = "<f0> |<f1> Y|<f2> "];
node7[label = "<f0> |<f1> A|<f2> "];
node8[label = "<f0> |<f1> C|<f2> "];

"node0":f2 -> "node4":f1 [color=red];
"node0":f0 -> "node1":f1 [color=blue];
"node1":f0 -> "node2":f1 [color=blue];
"node1":f2 -> "node3":f1 [color=red];
"node2":f2 -> "node8":f1 [color=red];
"node2":f0 -> "node7":f1 [color=blue];
"node4":f2 -> "node6":f1 [color=red];
"node4":f0 -> "node5":f1 [color=blue];
}

```

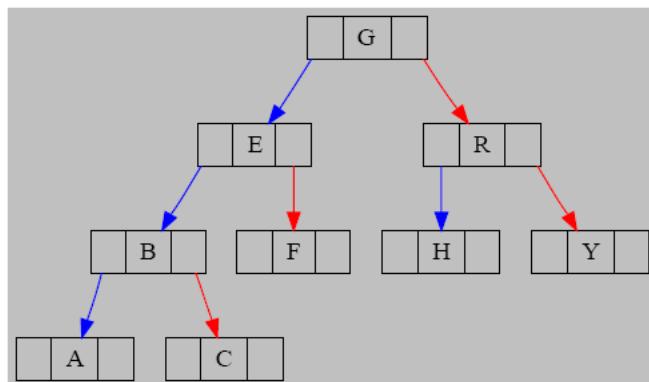


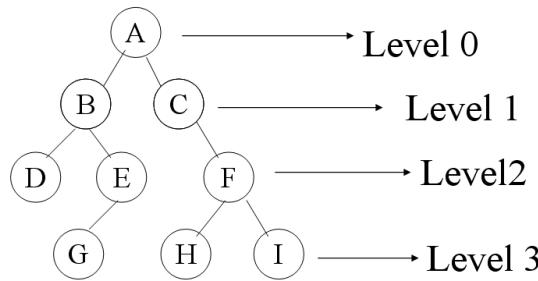
Figure 15.5: Visualizations Binary Tree

15.6. LEVEL(DEPTH) AND HEIGHT OF A TREE

15.6 Level(Depth) and Height of a tree

Level or Depth of a node

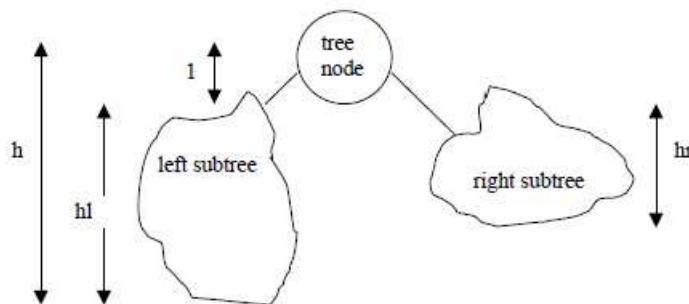
- The root of the tree has level 0, and the level of any other node in the tree is one more than the level of father



Recursive definition:

If root, level = 0
else 1 + level of parent

Height of a tree



Height of a tree = $1 + \max(\text{Height of left tree}, \text{Height of right tree})$

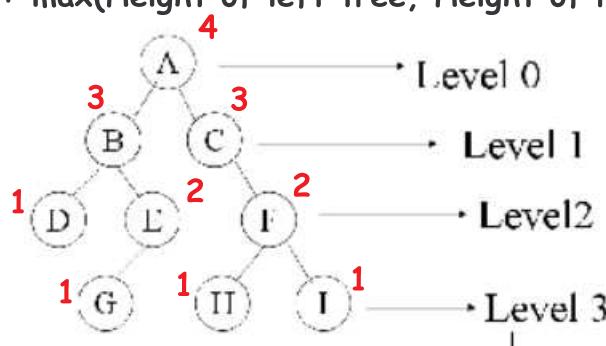
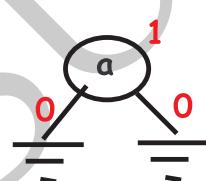


Figure 15.6: Definition of level(depth) and height of a binary tree

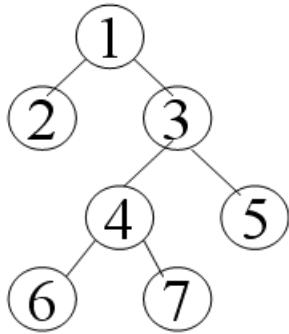
15.7 Strictly Binary Tree(SBT)

Strictly Binary Tree (SBT)

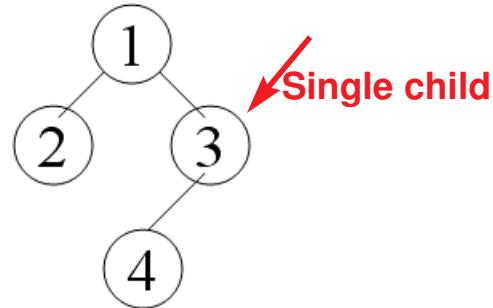
If every non leaf node in a binary tree has non empty left **and** right sub trees, the tree is called as strictly binary tree.

Every parent has two child

If every non leaf node in a binary tree has non empty left **and** right sub trees, the tree is called as strictly binary tree.



Strictly Binary Tree
4 leaves, 7 Nodes



Not a strictly binary tree

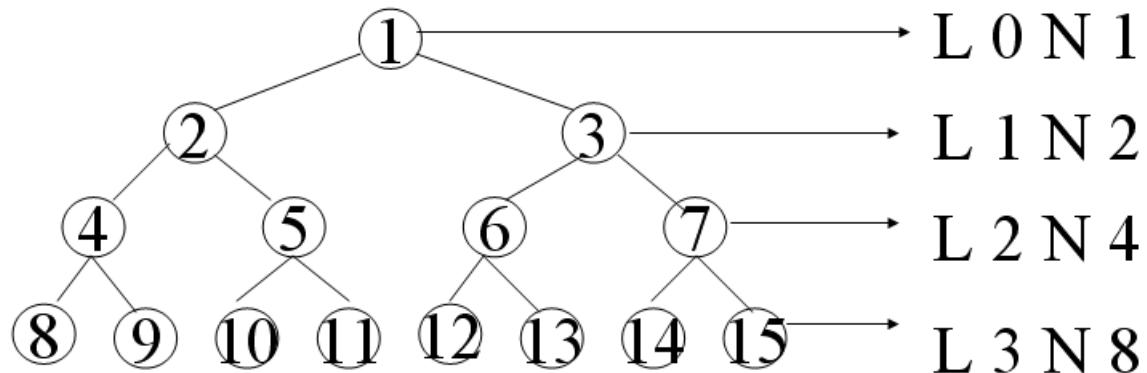
A STB with n leaves always contain $2n-1$ nodes

Figure 15.7: Strictly Binary Tree

15.8 Complete Binary Tree(CBT)

Complete Binary Tree

- A complete binary tree of **depth d** is a strict binary tree, all of whose leaves are at level d .



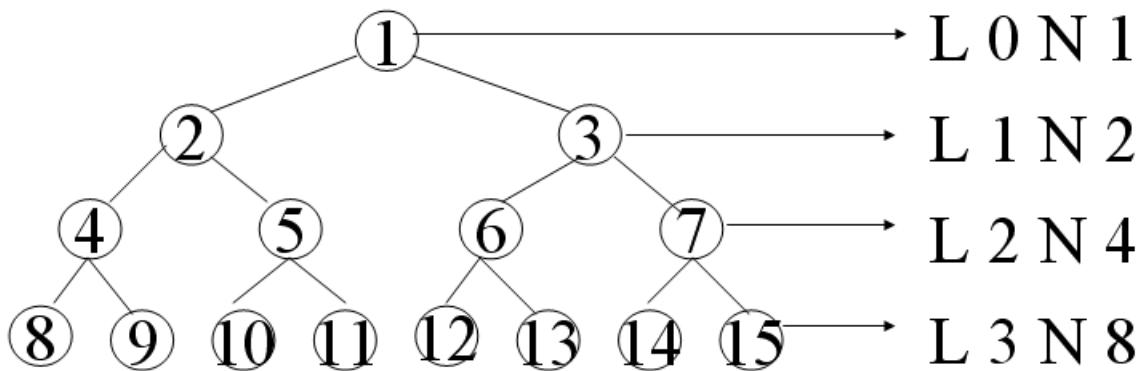
All nodes except leaves has both right and left child

Figure 15.8: Complete Binary Tree

15.9 Properties of Binary tree

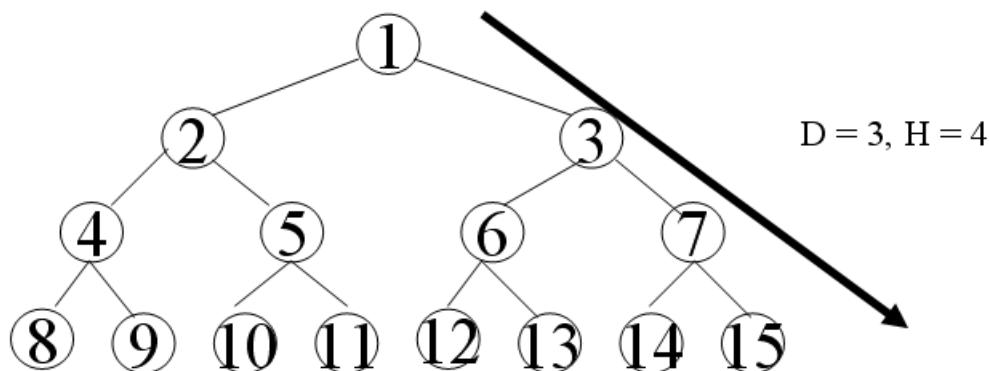
Property 1

- In any binary tree, maximum number of nodes on a level l is 2^l , where $l \geq 0$.



Property 2

- Maximum number of nodes possible in a binary tree of height $h = 2^h - 1$
- Maximum number of nodes possible in binary tree of depth $d = 2^{(d+1)} - 1$

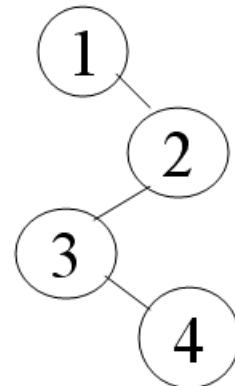
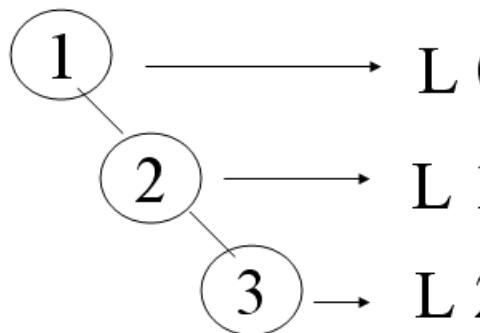


$$\text{Max node} = 2^4 - 1 = 15 \text{ or } \text{Max node} = 2^{(3+1)} - 1 = 15$$

Figure 15.9: Property 1 and 2

Property 3

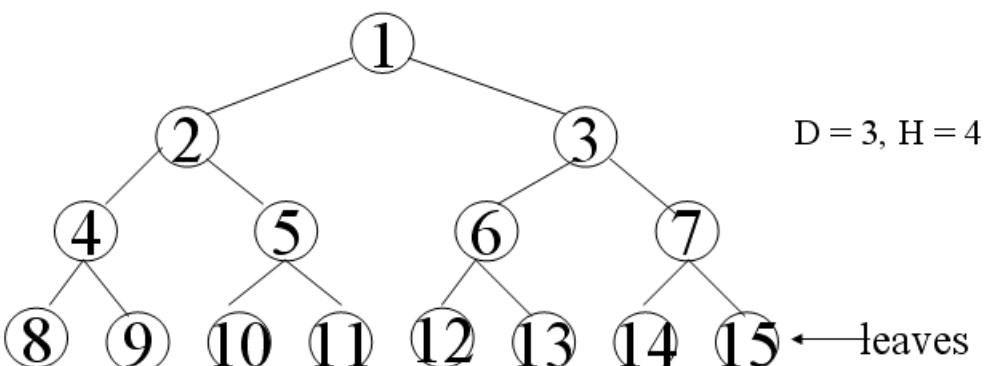
Minimum number of nodes possible in a binary tree of height h is h



Property 4

In a binary tree of **height h** there are at least $2^{(h-1)}$ leaves.

In a binary tree of **depth d** there are at least 2^d leaves.

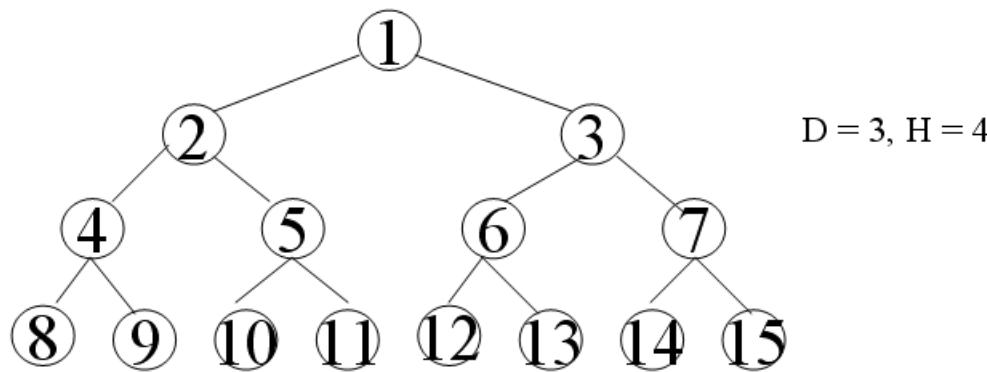


Max Internal node = $2^{(4-1)} = 8$ leaves, $2^3 = 8$ leaves.

Figure 15.10: Property 3 and 4

Property 5

- Number of internal node in a tree = number of nodes - number of leaves.

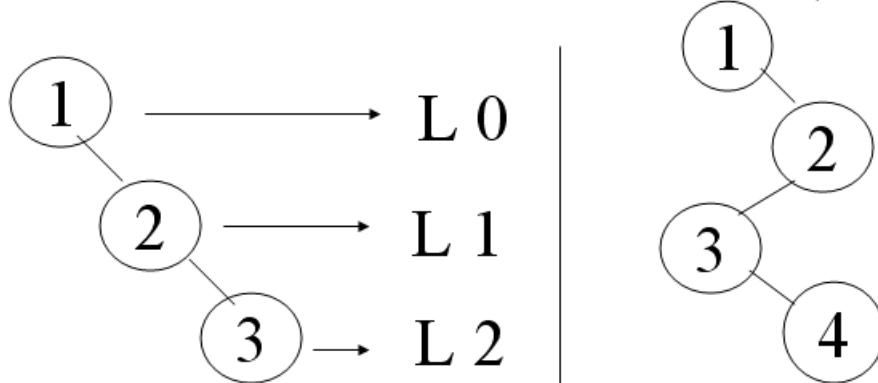


Number of internal nodes = $15 - 8 = 7$ nodes.

Figure 15.11: Property 5

Property 6

Maximum and Minimum level that are possible for binary tree with n nodes are $l_{max} = n - 1$, $l_{min} = \ln(n+1)-1$.

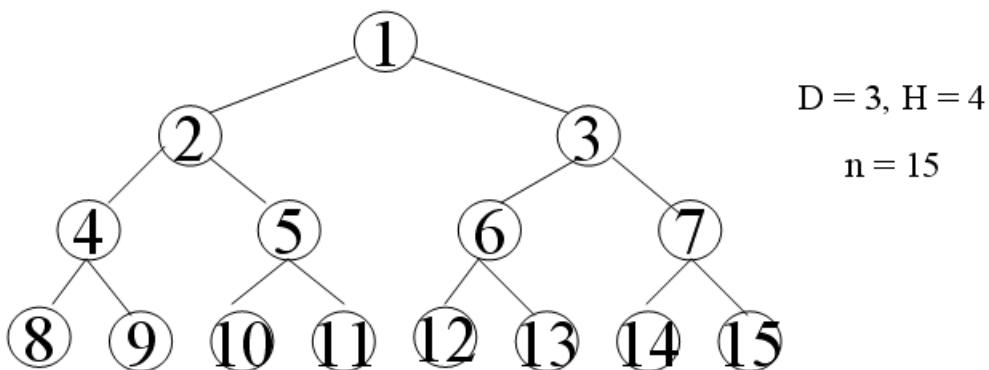


$W = 2, H = 3, N = 3, l_{max} = 2$

$W = 3, H = 4, N = 4, l_{max} = 3$

Property 7

Maximum and Minimum level that are possible for binary tree with n nodes are $l_{max} = n - 1$, $l_{min} = \ln(n+1)-1$.

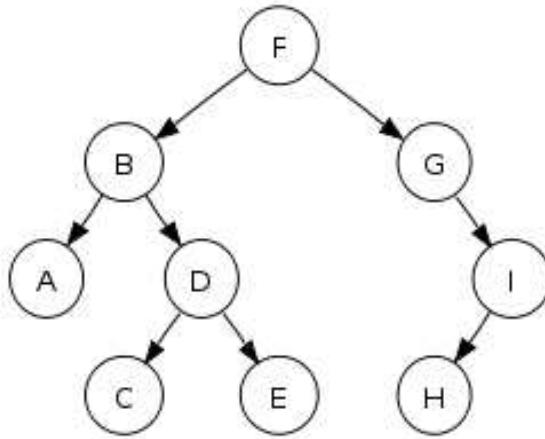


$$l_{min} = \ln(15+1)-1 = 3$$

Figure 15.12: Property 6 and 7

15.10 Binary tree traversal

<http://nova.umuc.edu/~jarc/idsv/lesson1.html>



In this [binary search tree](#)

- Preorder traversal sequence: F, B, A, D, C, E, G, I, H (root, left, right)
- Inorder traversal sequence: A, B, C, D, E, F, G, H, I (left, root, right)
- Postorder traversal sequence: A, C, E, D, B, H, I, G, F (left, right, root)
- Level-order traversal sequence: F, B, G, A, D, I, C, E, H

Figure 15.13: Binary tree traversal

15.10.1 Printing table of contents of a book using preorder traversal

15.10. BINARY TREE TRAVERSAL

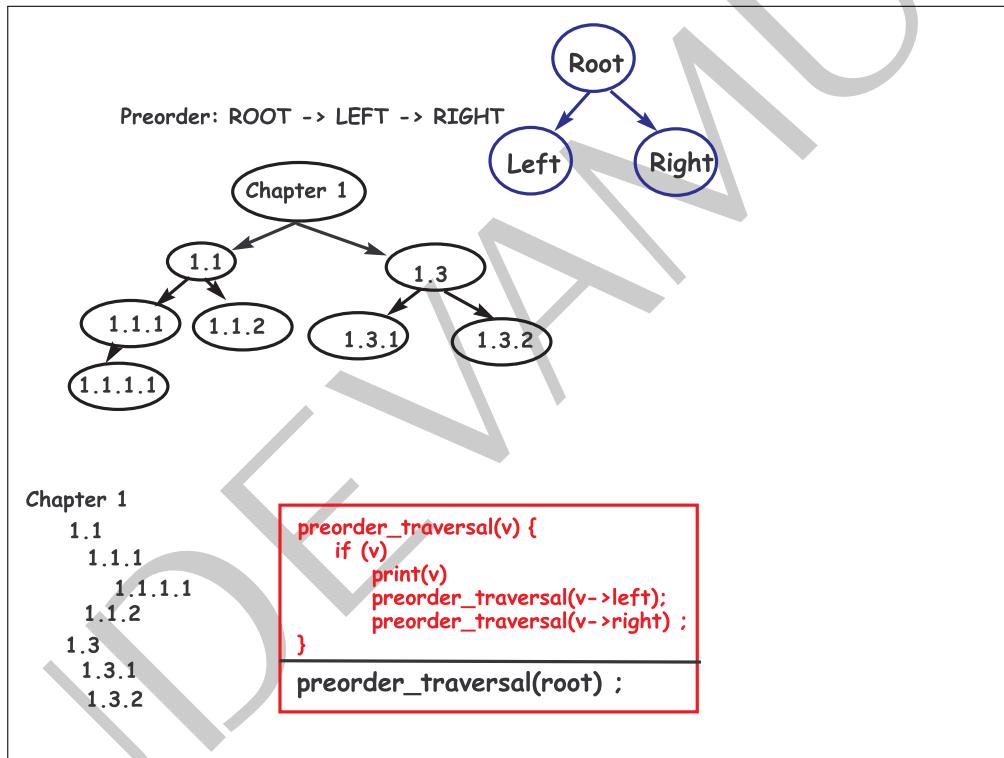


Figure 15.14: Applying preorder traversal

15.10.2 Deleting a tree using postorder traversal

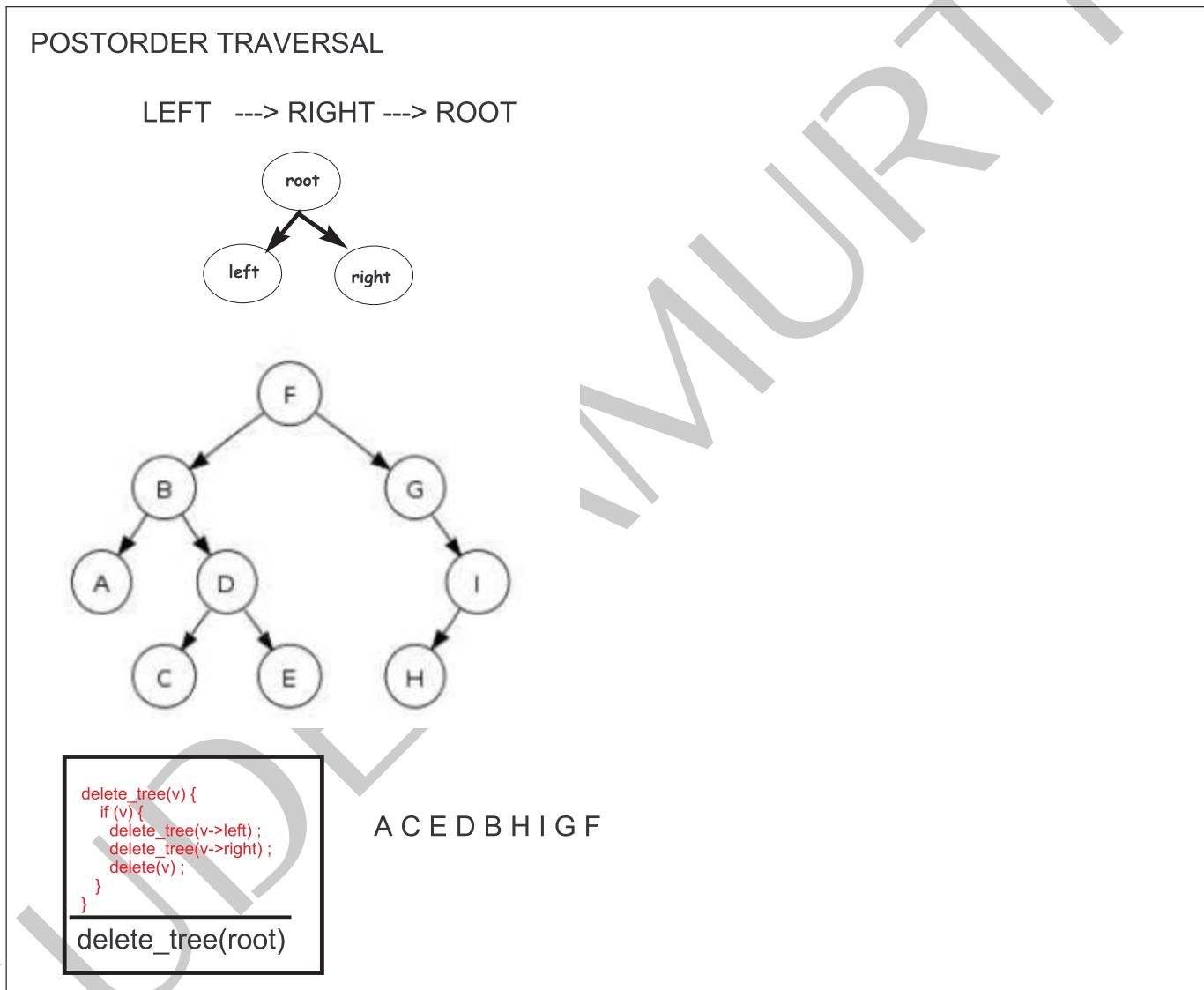


Figure 15.15: Applying postorder traversal to delete a tree

15.10.3 Sorting data using inorder traversal

15.10. BINARY TREE TRAVERSAL

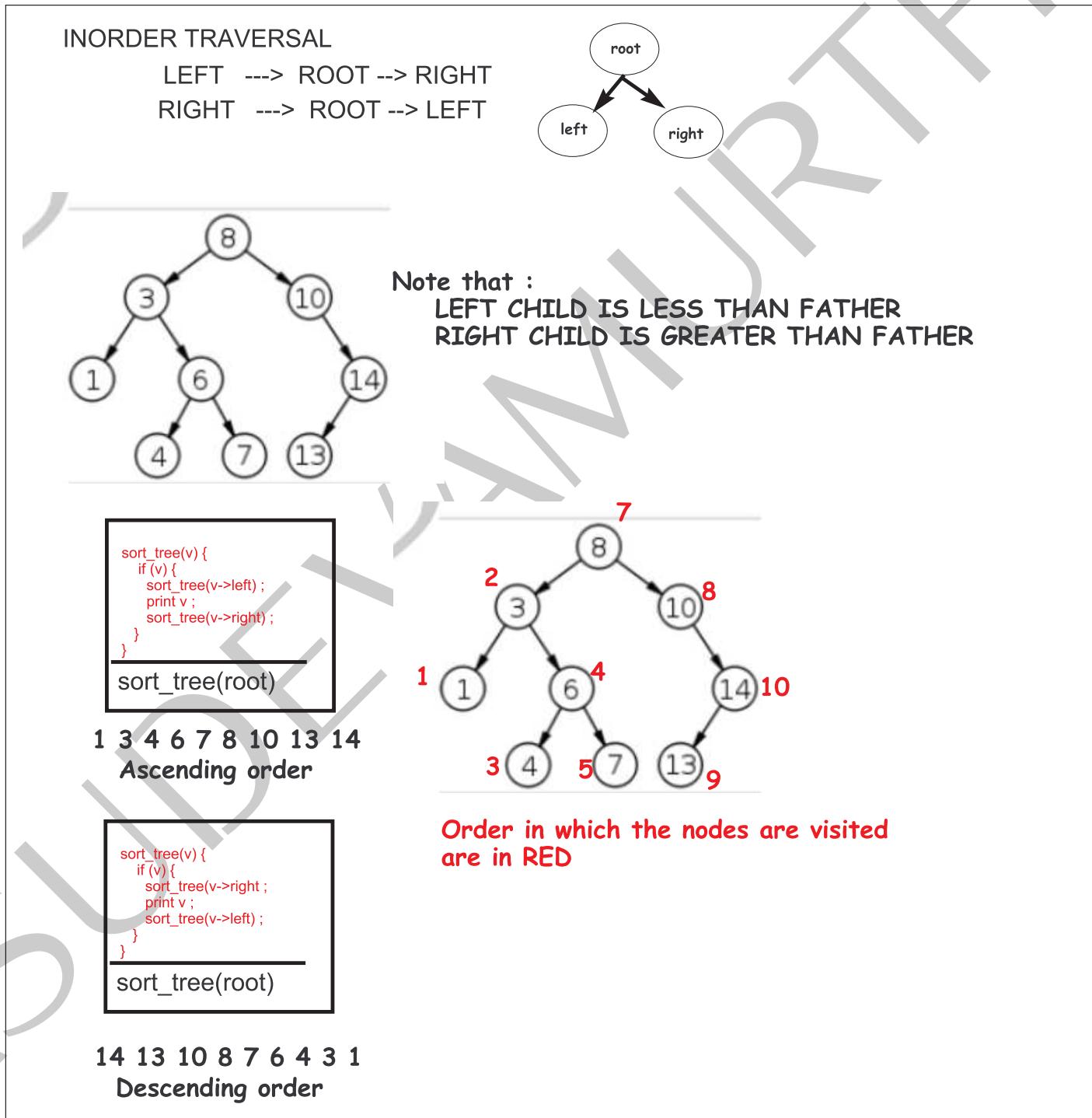


Figure 15.16: Applying inorder traversal to sort data

15.10.4 Printing tree using level-order traversal

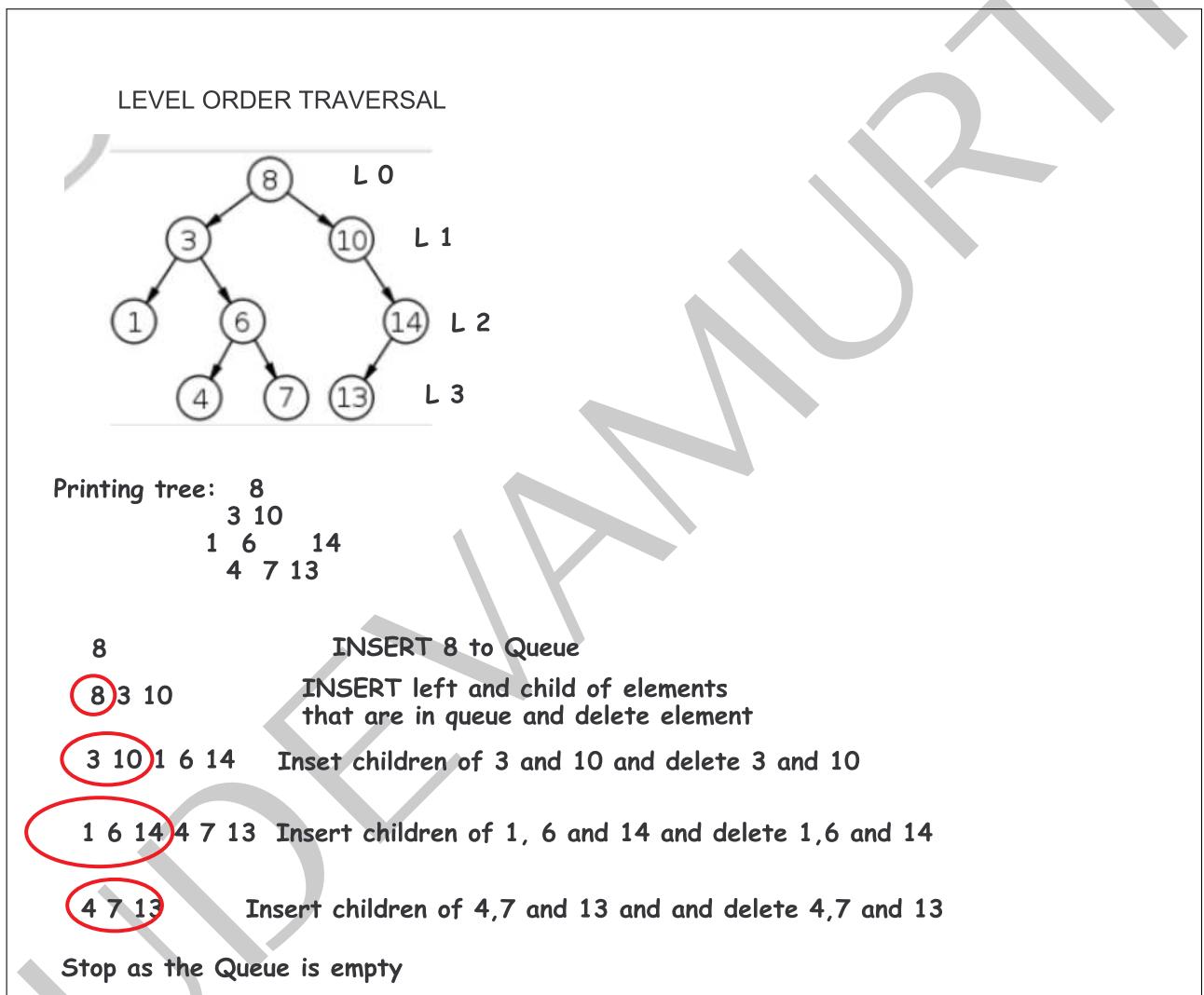
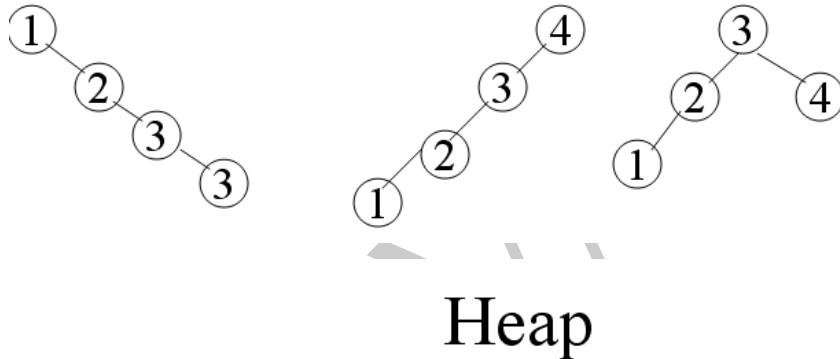


Figure 15.17: Printing tree using level-order traversal

15.11 Binary search tree(BST)

Binary Search Tree (BST)

- A BST T is a binary tree such that either T is empty or
- Each item in $\text{Left}(T)$ is $<$ *root item of T*
- each item in $\text{Right}(T)$ is \geq *root item of T*
- $\text{Left}(T)$ and $\text{Right}(T)$ are BST



- A heap T is a complete Binary tree in which either T is empty or
- each item in $\text{Left}(T)$ is \leq *Root item of T*
- each item in $\text{Right}(T)$ is \leq *Root item of T*
- *Left and Rights are heaps*

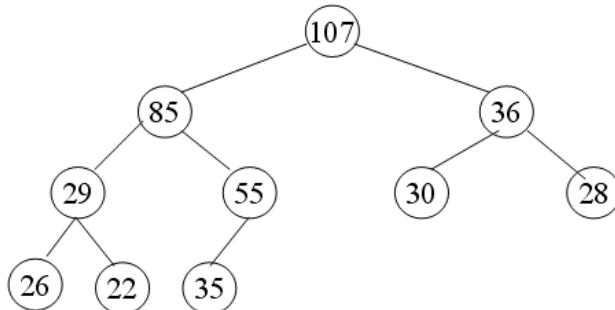
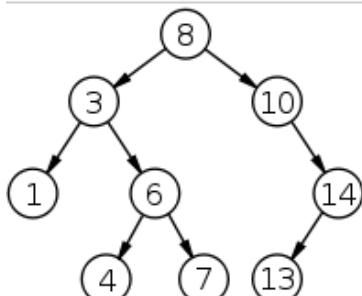
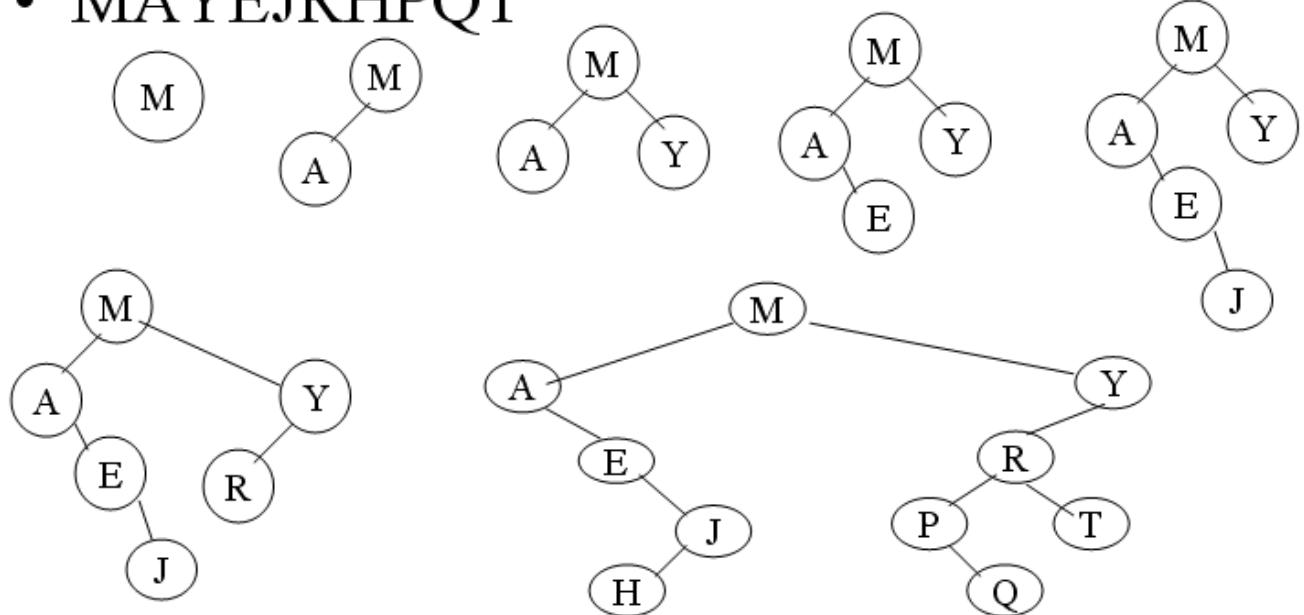


Figure 15.18: Binary search tree

15.12 Insertion to BST

Insertion in BST

- MAYEJRHPQT



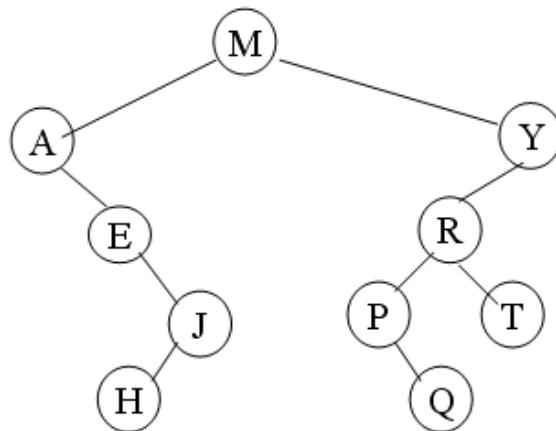
A binary search tree of size 9 and depth 3, with root 8 and leaves 1, 4, 7 and 13

Figure 15.19: Insertion to BST

15.13 Searching a key in BST

Searching A Key In BST

- MAYEJRHPQT



To search for the key P in the BST, follow the path M->Y->R->P

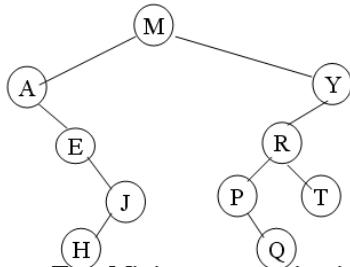
To search for the key H in the BST, follow the path M->A->E->J->H

Figure 15.20: Searching a key in BST

15.14 BST minimum and maximum of a node

Tree Minimum Of Node X

- The minimum of Node X is the node with smallest Value.

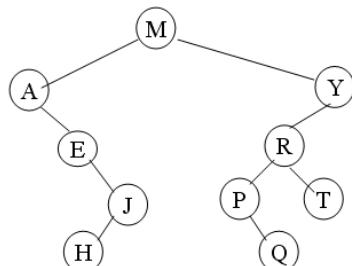


Tree Min(E) = E
Tree Min(Y) = P
Tree Min(M) = A

*Tree Minimum operation is done without comparing Keys
The minimum key is found by left pointers from Node X*

Tree Maximum Of Node X

- The maximum of Node X is the node with largest Value.



Tree Max(Y) = Y
Tree Max(A) = J
Tree Max(M) = Y

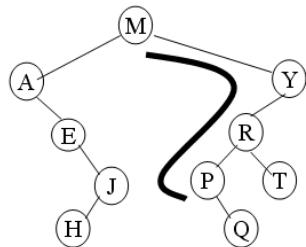
*Tree Maximum operation is done without comparing Keys
The maximum key is found by left pointers from Node X*

Figure 15.21: Minimum and maximum of a node

15.15 Successor and Predecessor of a node

Tree Successor Of Node X

- The successor of Node X is the node with the next possible smallest value greater than value[x].



$$\text{Tree Succ}(Y) = Y$$

$$\text{Tree Succ}(A) = E$$

$$\text{Tree Succ}(M) = P$$

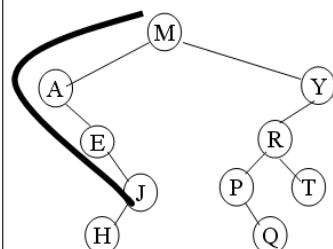
Tree Successor operation is done without comparing Keys

The Successor key is found by right followed by left pointers from Node X

Tree Successor has NO left Child

Tree Predecessor Of Node X

- The predecessor of Node X is the node with the next possible smallest value smaller than value[x].



$$\text{Tree Pred}(Y) = T$$

$$\text{Tree Pred}(A) = A$$

$$\text{Tree Pred}(M) = J$$

Tree Predecessor operation is done without comparing Keys

The Predecessor key is found by left followed by right pointers from Node X

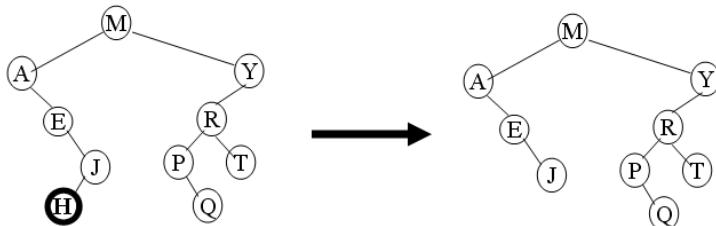
Tree Predecessor has NO right Child

Figure 15.22: Successor and Predecessor of a node

15.16 Deleting a key in BST

Deleting A Key In BST(Case 1)

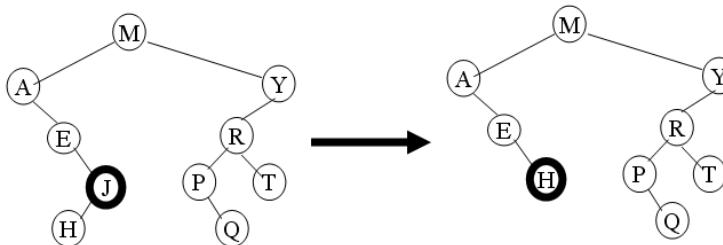
- MAYEJRHPQT



*Case 1: Deleting leaf node
Delete Leaf*

Deleting A Key In BST(Case 2)

MAYEJRHPQT

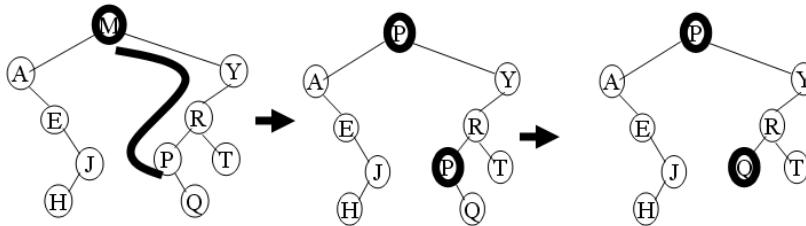


*Case 2: Deleting a node With One Child
Copy left or right child value to node and delete child*

Figure 15.23: Deleting a key in BST, Case 1 and Case 2

Deleting A Key In BST(Case 3a)

- MAYEJRHPQT



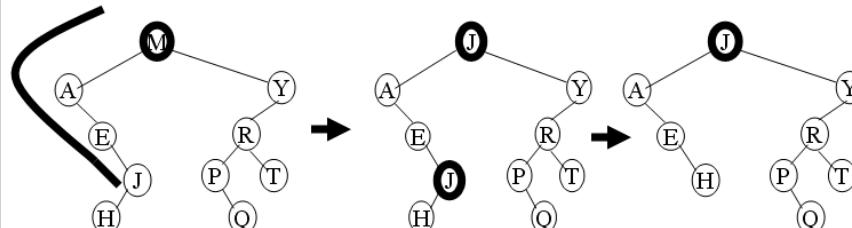
Case 3: Deleting a node With Two Child

Replace node by its successor and handle successor by case 2

Note that successor has no left child

Deleting A Key In BST(Case 3b)

- MAYEJRHPQT



Case 3: Deleting a node With Two Child

Replace node by its predecessor and handle predecessor by case 2

Note that predecessor has no right child

Figure 15.24: Deleting a key in BST, Case 3a and Case 3b

15.17 Huffman coding

a	45000
b	13000
c	12000
d	16000
e	9000
f	5000
<hr/>	
	1000000
<hr/>	

$a = 000 \ b = 001 \ c = 010 \ d = 011 \ e = 100 \ f = 101$

Total # bits = $1000000 * 3 = 3000000$

Suppose

a	= 0	b	= 101	c	= 100	d	= 111	e	= 1101	f	= 1100
---	-----	---	-------	---	-------	---	-------	---	--------	---	--------

Total # bits = $1(45) + 3(13 + 12 + 16) + 4(9 + 5)$
 $= 224 * 1000 = 224000$ bits

$3000000 \rightarrow 224000 = 25\% \text{ reduction}$

How do we get variable length code, like
 $a = 0 \ b = 101 \ c = 100 \ d = 111 \ e = 1101 \ f = 1110$

Figure 15.25: Concept of variable length code

15.17. HAUFFMAN CODING

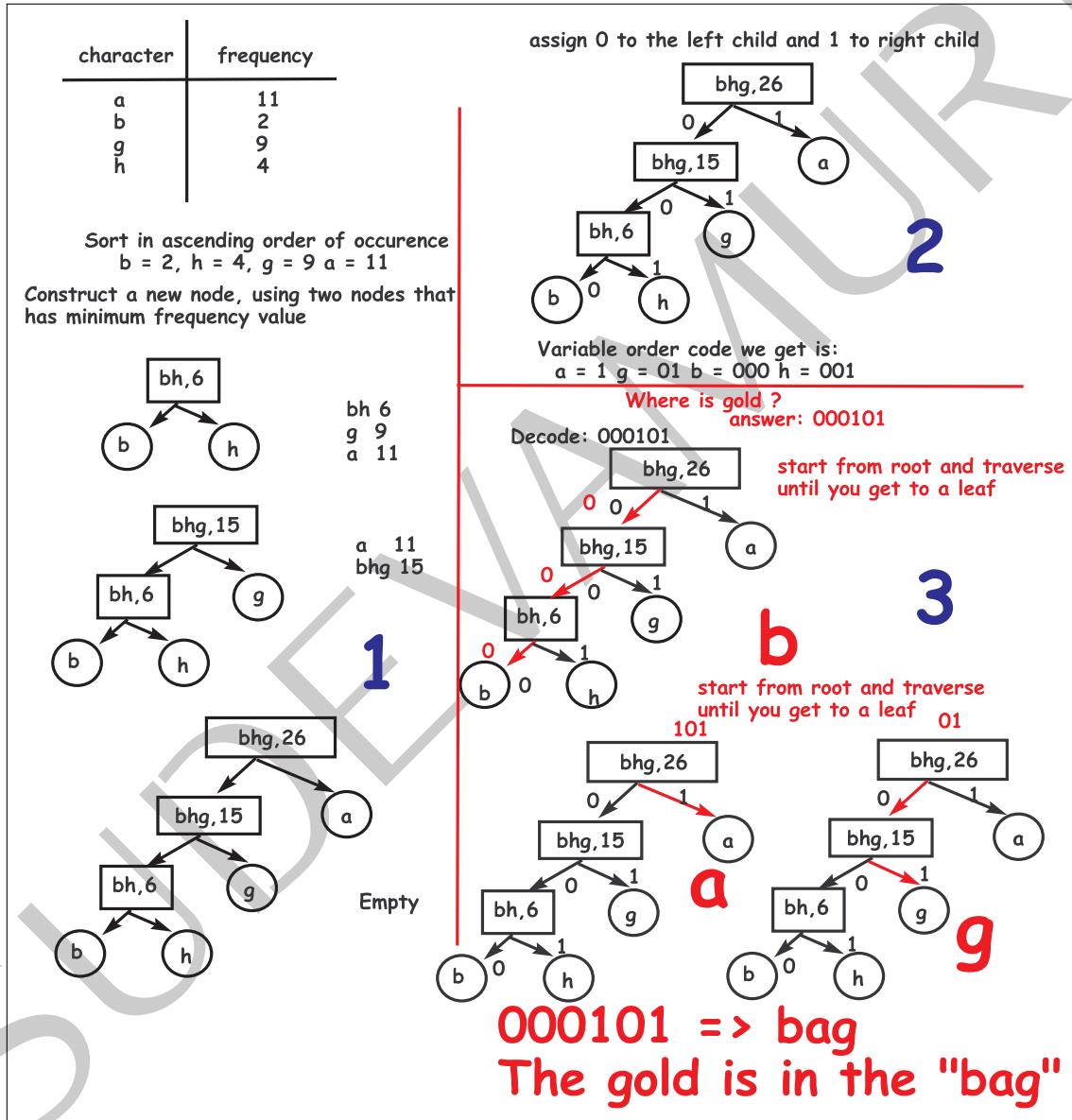


Figure 15.26: Building and decoding code from Hauffman tree

15.18 Complete Code

```
1 /*-
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy
3 Filename: dtree.h
4 */
5 #ifndef dtree_h
6 #define dtree_h
7
8 #include "dqueue.h"
9
10
11 /*-
12 All forward declarations
13 */
14 template <class T>
15 class bst ;
16
17 template <class T>
18 class node ;
19
20 /*-
21 class node
22 A node is leaf if it does not have either a right or left child
23 */
24 template <class T>
25 class node {
26 public:
27     node(const T& d):_data(d),_level(0),_left(NULL),_right(NULL){}
28     ~node() {}
29     bool isleaf() const { return (!_left && !_right); }
30 private:
31     T _data;
32     int _level ;
33     node<T*>* _left;
34     node<T*>* _right;
35
36     friend class bst<T>; //class bst can access nodes private part
37
38     /* no body can copy or equal node */
39     node(const node& x);
40     node& operator=(const node<T>& x);
41 };
42
43 /*-
44 class tree
45
46 A Binary Search Tree T is a binary tree such that
47 T is either empty or
48
49 1. Each item in Left T is < root item of T
50 2. Each item in Right T is  $\geq$  root item of T
51 3. Left and right trees are Binary Search Tree
52 */
53 template <class T>
54 class bst {
55 public:
```

```
56 bst(int(*cf) (const T& c1, const T& c2), void(*df) (T& c) = NULL);
57 ~bst();
58
59 int height_of_binary_tree() const;
60 void insert(const T& d);
61 void inorder(void(*some_func_ptr) (T& c));
62 void preorder(void(*some_func_ptr) (T& c));
63 void postorder(void(*some_func_ptr) (T& c));
64 void bfs(void(*some_func_ptr) (T& c) = NULL, bool fill_level_as_level =true);
65 int num_leaves() const;
66 void stats();
67 template <typename U> friend ostream& operator<<(ostream& o,bst<U>& t);
68 //NOTE U above. Will not compile on Linux with T
69 void print_tree_as_dot_r(ostream& o,node<T>* n) const;
70 T* find(const T& key) const;
71 T* tree_minimum(const T& key) const ;
72 T* tree_maximum(const T& key) const ;
73 T* tree_successor(const T& key) const ;
74 T* tree_predecessor(const T& key) const ;
75 bool deleteitem(const T& d);
76 bool display()const {return _display;}
77
78 private:
79 int _num_nodes_allocated;
80 int _num_nodes_freed;
81 node<T>* _root;
82 int(*_pntr_to_compare_func) (const T& c1, const T& c2);
83 void (*_pntr_to_func_to_delete_data) (T& c) ;
84
85 void _free_user_data(node<T>* n);
86 void _free_node(node<T>* n);
87 void _free_tree_r(node<T>* n);
88 void _insert_r(node<T>* root, node<T>*& n);
89 void _inorder_r(node<T>* root, void(*some_func_ptr) (T& c));
90 void _preorder_r(node<T>* root, void(*some_func_ptr) (T& c));
91 void _postorder_r(node<T>* root, void(*some_func_ptr) (T& c));
92 void bst<T>::num_leaves_r(node<T>* n, int& x) const;
93 int _height_of_binary_tree_r(node<T>* n) const;
94 node<T>* _find_r(node<T>* root, const T& key, node<T>** father = NULL) const;
95 node<T>* _tree_minimum(node<T>* key) const;
96 node<T>* _tree_maximum(node<T>* key) const ;
97 node<T>* _tree_successor(node<T>* key, node<T>*& father) const ;
98 node<T>* _tree_predecessor(node<T>* key, node<T>*& father) const ;
99 bool _deleteitem_case1(node<T>* n, node<T>* father);
100 bool _deleteitem_case2(node<T>* n);
101 static bool _display; /* ONLY ONCE for all object */
102 };
103
104
105 /**
106 tree Constructor
107 -----
108 template <typename T>
109 bst<T>::bst(int(*cf) (const T& c1, const T& c2), void(*df) (T& c));
110 _root(NULL),_pntr_to_func_to_delete_data(df),_pntr_to_compare_func(cf),
```

```
111 _num_nodes_allocated(0), _num_nodes_freed(0)
112 {
113     if (display()) {
114         cout << "in tree constructor:" << endl ;
115     }
116 }
117
118 /*-----
119 free user data
120 -----*/
121 template <typename T>
122 void bst<T>::_free_user_data(node<T>* n) {
123     if (_ptr_to_func_to_delete_data) {
124         _ptr_to_func_to_delete_data(n->_data);
125     }
126 }
127
128 /*-----
129 Free node of a tree
130 -----*/
131 template <typename T>
132 void bst<T>::_free_node(node<T>* n) {
133     _num_nodes_freed++;
134     _free_user_data(n);
135     FREE(n);
136 }
137
138 /*-----
139 Free tree
140 LEFT RIGHT ROOT
141 -----*/
142 template <typename T>
143 void bst<T>::_free_tree_r(node<T>* n) {
144     if (n) {
145         _free_tree_r(n->left);
146         _free_tree_r(n->right);
147         _free_node(n);
148     }
149 }
150
151 /*-----
152 tree Destructor
153 -----*/
154 template <typename T>
155 bst<T>::~bst() {
156     if (display()) {
157         cout << "in tree disstructor:" << endl ;
158     }
159     _free_tree_r(_root);
160     if (_num_nodes_allocated != _num_nodes_freed) {
161         assert(0);
162     }
163 }
164
165 /*-----
```

```
166
167 -----*/
168 template <typename T>
169 void bst<T>::print_tree_as_dot_r(ostream& o,node<T>* n) const {
170 if (n) {
171     //node0[label = <f0> |<f1> G|<f2> ];
172     o << "node" << n->_level << "[label = \"<f0> |<f1> " << n->_data << "|<f2> \"]" << endl ;
173     if (n->left) {
174         //node0:f0 -> "node1":f1 [color=blue];
175         o << "\"node" << n->_level << "\":f0 -> " << "\"node" << n->left->_level << "\":f1 [color=blue];" << endl ;
176
177     }
178     if (n->right) {
179         //node0:f2 -> "node4":f1 [color=red];
180         o << "\"node" << n->_level << "\":f2 -> " << "\"node" << n->right->_level << "\":f1 [color=red];" << endl ;
181     }
182     print_tree_as_dot_r(o,n->left);
183     print_tree_as_dot_r(o,n->right);
184 }
185 }
186
187 /*-----
188 Print a tree as dot file
189 digraph g {
190 node [shape = record,height=.1];
191 graph [bgcolor=gray];
192 node0[label = <f0> |<f1> 8|<f2> ]
193 "node0":f0 -> "node1":f1 [color=blue];
194 "node0":f2 -> "node2":f1 [color=red];
195 node1[label = <f0> |<f1> 3|<f2> ]
196 "node1":f0 -> "node3":f1 [color=blue];
197 "node1":f2 -> "node4":f1 [color=red];
198 node3[label = <f0> |<f1> 1|<f2> ]
199 node4[label = <f0> |<f1> 4|<f2> ]
200 "node4":f2 -> "node6":f1 [color=red];
201 node6[label = <f0> |<f1> 7|<f2> ]
202 "node6":f0 -> "node8":f1 [color=blue];
203 node8[label = <f0> |<f1> 6|<f2> ]
204 node2[label = <f0> |<f1> 10|<f2> ]
205 "node2":f2 -> "node5":f1 [color=red];
206 node5[label = <f0> |<f1> 14|<f2> ]
207 "node5":f0 -> "node7":f1 [color=blue];
208 node7[label = <f0> |<f1> 13|<f2> ]
209 }
210 -----*/
211 template <typename T>
212 ostream& operator<<(ostream& o,bst<T>& g){
213     g.bfs(NULL,false); //Fill _level fields as label
214     o << "## Jagadeesh Vasudevamurthy #####" << endl ;
215     o << "## run: dot -Tpdf graph.dot -o graph.pdf" << endl ;
216
217     o << "digraph g {" << endl ;
218     o << "node [shape = record,height=.1];" << endl ;
219     o << "graph [bgcolor=gray];" << endl ;
220     g.print_tree_as_dot_r(o,g._root);
```

```
221 o << "}" << endl ;
222 return o ;
223 }
224
225 /*-
226 Height of a null tree is 0
227 Height of a tree = 1 + MAX(height of left tree, height of right tree)
228 -----*/
229 template <typename T>
230 int bst<T>::_height_of_binary_tree_r(node<T>* n) const {
231 if (n == NULL){
232 return 0 ;
233 }
234 else {
235 int left_tree_height = _height_of_binary_tree_r(n->left) ;
236 int right_tree_height = _height_of_binary_tree_r(n->right) ;
237 int max = left_tree_height ;
238 if (right_tree_height > max){
239 max = right_tree_height ;
240 }
241 return (1 + max) ;
242 }
243 }
244
245 /*-
246 Height of binary tree:
247 The height of a binary tree is the maximum level of any leaf in the tree.
248 -----*/
249 template <typename T>
250 int bst<T>::height_of_binary_tree() const{
251 return _height_of_binary_tree_r(_root) ;
252 }
253
254 /*-
255 LEFT -> RIGHT -> ROOT
256 -----*/
257 template <typename T>
258 void bst<T>::_num_leaves_r(node<T>* n, int& x) const {
259 if (n) {
260 if (n->isleaf()){
261 x++ ;
262 }else {
263 _num_leaves_r(n->left,x) ;
264 _num_leaves_r(n->right,x) ;
265 }
266 }
267 }
268
269 /*-
270 Number of leaves in binary tree
271 -----*/
272 template <typename T>
273 int bst<T>::num_leaves() const{
274 int x = 0 ;
275 _num_leaves_r(_root,x) ;
```

```
276     return x ;
277 }
278
279 /*-----
280 BFS
281
282 This routine will fill _level field of the node
283 -----*/
284 template <typename T>
285 void bst<T>::bfs(void(*pv) (T& data), bool fill_level_as_level) {
286     dqueue<node<T*>> q ;
287     darray<node<T*>> a ;
288     int level = 0 ;
289     _root->_level = level ;
290     q.enqueue(_root) ;
291     while (1) {
292         int i = 0 ;
293         do {
294             node<T*>* n = q.front() ;
295             if (pv) {
296                 cout << n->_level << " " ;
297                 pv(n->_data) ;
298             }
299             if (n->left) a[i++] = n->left ;
300             if (n->right) a[i++] = n->right ;
301             q.dequeue() ;
302         }while (q.isEmpty() == false) ;
303         if (fill_level_as_level) {
304             level++ ;
305         }
306         for (int j = 0 ; j < i; j++) {
307             if (fill_level_as_level == false) {
308                 level++ ;
309             }
310             a[j]->_level = level ;
311             q.enqueue(a[j]) ;
312         }
313         if (i == 0) {
314             break ;
315         }
316     }
317     if (pv) {
318         cout << endl ;
319     }
320 }
321
322 /*-----
323 Strictly binary tree:
324 If every nonleaf node in a binary tree has nonempty
325 left and right subtress, the tree is called as strictly binary tree.
326
327 A strictly binary tree with n leaves always contain 2n-1 nodes
328 Number of internal node = number of nodes - number of leaves
329
330 Level of a node :
```

```
331 The root of the tree has level 0, and the level of
332 any other node in the tree is one more than the level
333 of father.
334
335 Height of a binary tree:
336 The height of a binary tree is the maximum level of any leaf in the tree.
337 -----
338 template <typename T>
339 void bst<T>::stats() {
340     int h = height_of_binary_tree();
341     int l = num_leaves();
342     cout << "DEPTH OF BINARY TREE " << h << endl;
343     cout << "NUMBER OF LEAVES " << l << endl;
344     int n = _num_nodes_allocated - _num_nodes_freed;
345     cout << "NUMBER OF NODES " << n << endl;
346     cout << "NUMBER OF INTERNAL NODE " << n - l << endl;
347     if (l * 2 - 1 == n) {
348         cout << "NUMBER OF LEAVES * 2 - 1 = NUMBER OF NODES. Strictly binary tree" << endl;
349     } else {
350         cout << "Not a Strictly binary tree" << endl;
351     }
352 }
353
354 /**
355 preorder traversal of tree
356 ROOT -> LEFT -> RIGHT
357 -----
358 template <typename T>
359 void bst<T>::_preorder_r(node<T>* n, void(*some_func_ptr)(T& data)) {
360     if (n) {
361         some_func_ptr(n->data);
362         _preorder_r(n->left,some_func_ptr);
363         _preorder_r(n->right,some_func_ptr);
364     }
365 }
366
367 /**
368 preorder traversal of tree from TOP
369 -----
370 template <typename T>
371 void bst<T>::preorder(void(*some_func_ptr)(T& data)) {
372     _preorder_r(_root,some_func_ptr);
373 }
374
375 /**
376 inorder traversal of tree
377 LEFT -> ROOT -> RIGHT
378 -----
379 template <typename T>
380 void bst<T>::_inorder_r(node<T>* n, void(*some_func_ptr)(T& data)) {
381     if (n) {
382         _inorder_r(n->left,some_func_ptr);
383         some_func_ptr(n->data);
384         _inorder_r(n->right,some_func_ptr);
385     }
386 }
```

```
386 }
387
388 -----
389 inorder traversal of tree from TOP
390 ----- */
391 template <typename T>
392 void bst<T>::inorder(void(*some_func_ptr) (T& data)) {
393     _inorder_r(_root,some_func_ptr);
394 }
395
396 -----
397 postorder traversal of tree
398 LEFT -> RIGHT -> ROOT
399 ----- */
400 template <typename T>
401 void bst<T>::_postorder_r(node<T>* n, void(*some_func_ptr) (T& data)) {
402     if (n) {
403         _postorder_r(n->_left,some_func_ptr);
404         _postorder_r(n->_right,some_func_ptr);
405         some_func_ptr(n->_data);
406     }
407 }
408
409 -----
410 postorder traversal of tree from TOP
411 ----- */
412 template <typename T>
413 void bst<T>::postorder(void(*some_func_ptr) (T& data)) {
414     _postorder_r(_root,some_func_ptr);
415 }
416
417 -----
418 insert d to bst maintaining bst property
419 Note that root is passed by value
420 n is passed by alias
421 ----- */
422 template <typename T>
423 void bst<T>::_insert_r(node<T>* root, node<T>*& n) {
424     if (_pntr_to_compare_func(root->_data,n->_data) <= 0) {
425         /* ROOT IS SMALLER - go on Right side*/
426         if (root->_right) {
427             root = root->_right;
428         } else {
429             root->_right = n;
430             return ;
431         }
432     } else {
433         /* ROOT IS BIGGER - go on Left side*/
434         if (root->_left) {
435             root = root->_left;
436         } else {
437             root->_left = n;
438             return ;
439         }
440     }
}
```

```
441 _insert_r(root,n);
442 }
443
444 /*-----
445 insert d to bst maintaining bst property
446 -----*/
447 template <typename T>
448 void bst<T>::insert(const T& d){
449     node<T>* n = ALLOC(node<T>)(d); /* IF T is by value copy constructor is called */
450     _num_nodes_allocated++;
451     if (!_root) {
452         _root = n;
453     } else {
454         _insert_r(_root,n);
455     }
456 }
457
458 /*-----
459 find the object recursively
460 -----*/
461 template <typename T>
462 node<T>* bst<T>::_find_r(node<T>* root, const T& key,node<T>** father) const{
463     if (!root) {
464         if (father) {
465             *father = NULL;
466         }
467         return NULL;
468     }
469     int x = _ptr_to_compare_func(root->_data,key);
470     if (x == 0) {
471         return root;
472     }
473     if (father) {
474         *father = root;
475     }
476     if (x <= 0) {
477         /* ROOT IS SMALLER - go on Right side*/
478         root = root->_right;
479     } else {
480         /* ROOT IS BIGGER - go on Left side*/
481         root = root->_left;
482     }
483     return _find_r(root,key,father);
484 }
485
486 }
487
488 /*-----
489 If T is by value you get address of T or NIL
490 if T is by ptr you get address of ptr that points to object T or NIL
491 -----*/
492 template <typename T>
493 T* bst<T>::find(const T& key) const{
494     node<T>* n = _find_r(_root,key);
495     if (n){
```

```
496     return &n->_data;
497 }
498 return NULL;
499 }
500 */
501 /*-
502 Tree minimum
503 Note that you are getting minimum without checking data
504 */
505 template <typename T>
506 node<T>* bst<T>::_tree_minimum(node<T>* key) const {
507     while (key->_left) {
508         key = key->_left;
509     }
510     return key;
511 }
512 */
513 /*-
514 Tree minimum
515 */
516 template <typename T>
517 T* bst<T>::tree_minimum(const T& key) const {
518     node<T>* n = _find_r(_root, key);
519     if (n) {
520         node<T>* k = _tree_minimum(n);
521         return &k->_data;
522     }
523     return NULL;
524 }
525 */
526 /*-
527 Tree maximum
528 Note that you are getting maximum without checking data
529 */
530 template <typename T>
531 node<T>* bst<T>::_tree_maximum(node<T>* key) const {
532     while (key->_right) {
533         key = key->_right;
534     }
535     return key;
536 }
537 */
538 /*-
539 Tree maximum
540 */
541 template <class T>
542 T* bst<T>::tree_maximum(const T& key) const {
543     node<T>* n = _find_r(_root, key);
544     if (n) {
545         node<T>* k = _tree_maximum(n);
546         return &k->_data;
547     }
548     return NULL;
549 }
550 
```

```
551
552 -----
553 Tree successor of a node
554 Note that you are getting successor without checking data
555 The successor of Node X is the node with the next possible large value
556 Go to the right and follow left pointers to the end
557 -----*/
558 template <typename T>
559 node<T>* bst<T>::_tree_successor(node<T>* key, node<T>*& father) const {
560     if (key->_right) {
561         father = key;
562         key = key->_right;
563         while (key->_left) {
564             father = key;
565             key = key->_left;
566         }
567     }
568     return key;
569 }
570
571 -----
572 Tree successor
573 -----*/
574 template <typename T>
575 T* bst<T>::tree_successor(const T& key) const {
576     node<T>* n = _find_r(_root, key);
577     node<T>* father = NULL;
578     if (n) {
579         node<T>* k = _tree_successor(n, father);
580         return &k->_data;
581     }
582     return NULL;
583 }
584
585 -----
586 Tree predecessor of a node
587 Note that you are getting predecessor without checking data
588 The predecessor of Node X is the node with the next possible small value
589 Go to the left and follow right pointers to the end
590 -----*/
591 template <typename T>
592 node<T>* bst<T>::_tree_predecessor(node<T>* key, node<T>*& father) const {
593     if (key->_left) {
594         father = key;
595         key = key->_left;
596         while (key->_right) {
597             father = key;
598             key = key->_right;
599         }
600     }
601     return key;
602 }
603
604 -----
605 Tree predecessor
```

```
606 -----*/
607 template <typename T>
608 T* bst<T>::tree_predecessor(const T& key) const {
609     node<T>* n = _find_r(_root, key);
610     node<T>* father = NULL;
611     if (n) {
612         node<T>* k = _tree_predecessor(n, father);
613         return &k->_data;
614     }
615     return NULL;
616 }
617 */
618 /*-
619 delete a key from tree case 1
620 -----*/
621 template <typename T>
622 bool bst<T>::_deleteitem_case1(node<T>* n, node<T>* father) {
623     if (n->isleaf()) {
624         father->_left = NULL;
625         father->_right = NULL;
626         _free_node(n);
627         return true;
628     }
629     return false;
630 }
631 */
632 /*-
633 delete a key from tree case 2
634 -----*/
635 template <typename T>
636 bool bst<T>::_deleteitem_case2(node<T>* n) {
637     if ((n->_right && n->_left == NULL) || (n->_left && n->_right == NULL)) {
638         if (n->_right && n->_left == NULL) {
639             n->_data = n->_right->_data; /* copy constructor called by if data is by value */
640             _free_node(n->_right);
641             n->_right = NULL;
642             return true;
643         }
644         n->_data = n->_left->_data; /* copy constructor called by if data is by value */
645         _free_node(n->_left);
646         n->_left = NULL;
647         return true;
648     }
649     return false;
650 }
651 */
652 */
653 /*-
654 delete a key from tree
655 -----*/
656 template <typename T>
657 bool bst<T>::deleteitem(const T& key){
658     node<T>* father = NULL;
659     node<T>* n = _find_r(_root, key, &father);
660     if (n){
```

```
661 bool x = _deleteitem_case1(n,father);
662 if (x) {
663     return true;
664 }
665 x = _deleteitem_case2(n);
666 if (x) {
667     return true;
668 }
669 node<T>* k = _tree_successor(n,father);
670 n->_data = k->_data; /* copy constructor called by if data is by value */
671 x = _deleteitem_case1(n,father);
672 if (x) {
673     return true;
674 }
675 _deleteitem_case2(k);
676 return true;
677 }
678 return false;
679 }
680
681 #endif
682
```

```
1 /*-----  
2 Copyright (c) 2010 Author: Jagadeesh Vasudevanurthy  
3 Filename: dtree.cpp  
4 compile: g++ dtree.cpp  
5  
6 Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)  
7 -----*/  
8  
9 #include "dtree.h"  
10  
11 /*-----  
12 static definition - only once at the start  
13 Change to false, if you don't need verbose  
14 -----*/  
15 template <typename T>  
16 bool bst<T>::_display = true;  
17  
18 template <typename T>  
19 bool darray<T>::_display = true;  
20  
21 template <typename T>  
22 bool deque<T>::_display = true;  
23  
24 /*-----  
25 -----*/  
26 static void test_bst_int(){  
27     bst<int> h(int_descending_order);  
28     int array[] = {8,3,10,4,7,6,14,1,13};  
29     int j = sizeof(array)/sizeof(int);  
30     for (int i = 0; i < j; i++) {  
31         cout << array[i] << " ";  
32     }  
33     cout << endl;  
34     for (int i = 0; i < j; i++) {  
35         h.insert(array[i]);  
36     }  
37     cout << h;  
38 }  
39  
40 /*-----  
41 -----*/  
42 static void test_bst_char(){  
43     bst<char> h(charcompare);  
44     int i,j ;  
45     char *ch ;  
46     char array[] = {'M', 'A', 'Y', 'E', 'J', 'R', 'H', 'P', 'Q', 'T'};  
47     cout << "CHAR ARRAY" << endl;  
48     j = sizeof(array)/sizeof(char);  
49     for (i = 0; i < j; i++) {  
50         cout << array[i] << " ";  
51     }  
52     cout << endl;  
53     for (i = 0; i < j; i++) {
```

```
56 h.insert(array[i]);  
57 }  
58  
59 h.stats();  
60 cout << h;  
61 cout << "PREORDER" << endl;  
62 h.preorder(print_char);  
63 cout << endl;  
64 cout << "INORDER" << endl;  
65 h.inorder(print_char);  
66 cout << endl;  
67 cout << "POSTORDER" << endl;  
68 h.postorder(print_char);  
69 cout << endl;  
70 cout << "LEVELORDER" << endl;  
71 h.bfs(print_char);  
72 for (i = 0; i < j; i++) {  
73     ch = h.find(array[i]);  
74     if (ch) {  
75         cout << *ch << " Found" << endl;  
76     } else {  
77         assert(0);  
78     }  
79 }  
80 ch = h.find('Z');  
81 if (ch == NULL) {  
82     cout << "Z not found" << endl;  
83 } else {  
84     assert(0);  
85 }  
86 /* find minimum in tree */  
87 ch = h.tree_minimum('E');  
88 cout << "TREE MINIMUM of E: " << *ch << endl;  
89 ch = h.tree_minimum('Y');  
90 cout << "TREE MINIMUM of Y: " << *ch << endl;  
91 ch = h.tree_minimum('M');  
92 cout << "TREE MINIMUM of M: " << *ch << endl;  
93  
94 /* find maximum in tree */  
95 ch = h.tree_maximum('Y');  
96 cout << "TREE MAXIMUM of Y: " << *ch << endl;  
97 ch = h.tree_maximum('A');  
98 cout << "TREE MAXIMUM of A: " << *ch << endl;  
99 ch = h.tree_maximum('M');  
100 cout << "TREE MAXIMUM of M: " << *ch << endl;  
101  
102 /* find successor in tree */  
103 ch = h.tree_successor('Y');  
104 cout << "TREE SUCCESSOR of Y: " << *ch << endl;  
105 ch = h.tree_successor('A');  
106 cout << "TREE SUCCESSOR of A: " << *ch << endl;  
107 ch = h.tree_successor('M');  
108 cout << "TREE SUCCESSOR of M: " << *ch << endl;  
109  
110 /* find predecessor in tree */
```

```
111 ch = h.tree_predecessor('Y');
112 cout << "TREE PREDECESSOR of Y: " << *ch << endl;
113 ch = h.tree_predecessor('A');
114 cout << "TREE PREDECESSOR of A: " << *ch << endl;
115 ch = h.tree_predecessor('M');
116 cout << "TREE PREDECESSOR of M: " << *ch << endl;
117
118 /* delete a node */
119 h.stats();
120 if (h.deleteitem('H')) {
121     cout << " H deleted " << endl;
122     h.bfs(print_char);
123 } else {
124     cout << " H not found in tree " << endl;
125 }
126 cout << "-----" << endl;
127
128 h.stats();
129 if (h.deleteitem('P')) {
130     cout << " P deleted " << endl;
131     h.bfs(print_char);
132 } else {
133     cout << " P not found in tree " << endl;
134 }
135
136 cout << "-----" << endl;
137
138 h.stats();
139 if (h.deleteitem('M')) {
140     cout << " M deleted " << endl;
141     h.bfs(print_char);
142 } else {
143     cout << " M not found in tree " << endl;
144 }
145
146
147 /*
148 main
149 */
150 int main() {
151     test_bst_int();
152     cout << " _____" << endl;
153     test_bst_char();
154     cout << " _____" << endl;
155     return 0;
156 }
157
```

15.19. PROBLEM SET

15.19 Problem set

integer tree

```
inttree.h
```

```
class node {  
    //ALL REQUIRED PUBLIC FUNCTION  
    private:  
    int _x;  
    //ANY PRIVATE MEMBERS/FUNCTION  
}  
class inttree {  
    //ALL REQUIRED PUBLIC FUNCTION  
  
    private:  
    node* _root;  
    //ANY PRIVATE MEMBERS/FUNCTION  
};
```

```
inttree.cpp
```

Write all inttree class definitions

1. tree should hold integer data for nodes
2. if new node is less than or equal to root/parent, it goes on the left, if more than root/parent, it goes on the right
3. write recursive and non-recursive functions for each of the three traversals:
preorder, postorder, and inorder.
Each of the above procedure should store the answers in an array

```
inttree.cpp
```

```
test(int n){  
    inttree h;  
    1. for (int i = 0; i < n; ++i) {  
        generate a random number  
        r, between 0 to 1000000  
        h.insert(r)  
    }  
  
    2. print height of the tree h  
  
    3. int* a = new int(n);  
    int* b = new int(n)  
  
    h.preorder(a);  
    print CPU TIME  
    h.preorder_non_recursive(b);  
    print CPU TIME  
  
    for (i = 0; i < n; ++i){  
        assert(a[i] == b[i]);  
    }  
    4. Repeat 2 for postorder  
    5. Repeat 2 for inorder  
    6. h should die with no memory leak  
  
void main(){  
    for(int i = 10; i <=100000;i=i*10) {  
        test(i);  
    }  
}
```

email all the 3 files.
There should be NO memory leaks

Figure 15.27: Integer tree

Problem 15.19.1.

CHAPTER 15. TREE

Problem 15.19.2. Implement the functions, in program `hauffman.h` and `hauffman.cpp` using **Huffman coding** algorithm.

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy  
3 Filename: hauffman.h  
4 -----*/  
5 #ifndef hauffman_h  
6 #define hauffman_h  
7  
8 #include "util.h"  
9 #include <string>  
10 #include <map>  
11  
12 /*-----  
13 class hauffman  
14 -----*/  
15 class hauffman {  
16 public:  
17 hauffman(const char* s) {}  
18 ~hauffman(){}  
19  
20 void get_code_for_each_character(const char* s, map<char,string>& table);  
21 string code_message(const char* s, const map<char,string>& table);  
22 string decode_message(const string& s);  
23 };  
24  
25 /*-----  
26 s = 'aabbbggggghhhhaaaggggaaaa' ;  
27  
28 table after filling:  
29 table: a = "1" g = "01" b = "000" h = "001"  
30 -----*/  
31 void hauffman::get_code_for_each_character(const char* s, map<char,string>& table){  
32 /* WRITE YOUR CODE HERE */  
33  
34 }  
35  
36 /*-----  
37 s = 'gold' ;  
38 table: a = "1" g = "01" b = "000" h = "001"  
39  
40 return "000101"  
41 -----*/  
42 string hauffman::code_message(const char* s, const map<char,string>& table){  
43 string s1 ;  
44 /* WRITE YOUR CODE HERE */  
45 return s1 ;  
46 }  
47  
48 /*-----  
49 s = "000101"  
50  
51 return "gold" ;  
52 -----*/  
53 string hauffman::decode_message(const string& s){  
54 string s1 ;  
55 /* WRITE YOUR CODE HERE */
```

```
56   return s1;
57 }
58
59 #endif
60
```

```
1 /*-----
2 Copyright (c) 2010 Author: Jagadeesh Vasudevarnurthy
3 Filename: hauffman.cpp
4 compile: g++ hauffman.cpp
5
6 Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)
7 -----
8 #include "hauffman.h"
9
10 /*
11 test1
12 -----
13 static void test1() {
14 const char *original_speech =
15 "Long years ago we made a tryst with destiny, and now the time comes when \
16 we shall redeem our pledge, not wholly or in full measure, but very substantially.\ \
17 At the stroke of the midnight hour, when the world sleeps, India will awake \
18 to life and freedom. A moment comes, which comes but rarely in history, when \
19 we step out from the old to the new, when an age ends, and when the soul of \
20 a nation, long suppressed, finds utterance." ;
21
22 hauffman h(original_speech);
23 map<char,string> code_table ;
24 h.get_code_for_each_character(original_speech,code_table);
25 string coded_string = h.code_message(original_speech,code_table);
26 string decoded_string = h.decode_message(coded_string);
27 assert(decoded_string.c_str() == original_speech);
28 }
29
30 /*
31 test2
32 -----
33 static void test2() {
34 const char *original_speech =
35 "The quick brown fox jumps over the lazy dog. Pack my box with five dozen liquor jugs" ;
36 hauffman h(original_speech);
37 map<char,string> code_table ;
38 h.get_code_for_each_character(original_speech,code_table);
39 const char*s1 = "jumps over the lazy dog. Pack my box with";
40 string coded_string = h.code_message(s1,code_table);
41 string decoded_string = h.decode_message(coded_string);
42 assert(decoded_string.c_str() == s1);
43 }
44
45 /*
46 main
47 -----
48 int main() {
49 test1();
50 test2();
51 return 0;
52 }
53
54
```

Problem 15.19.3. Implement 8 puzzle problem as described below.

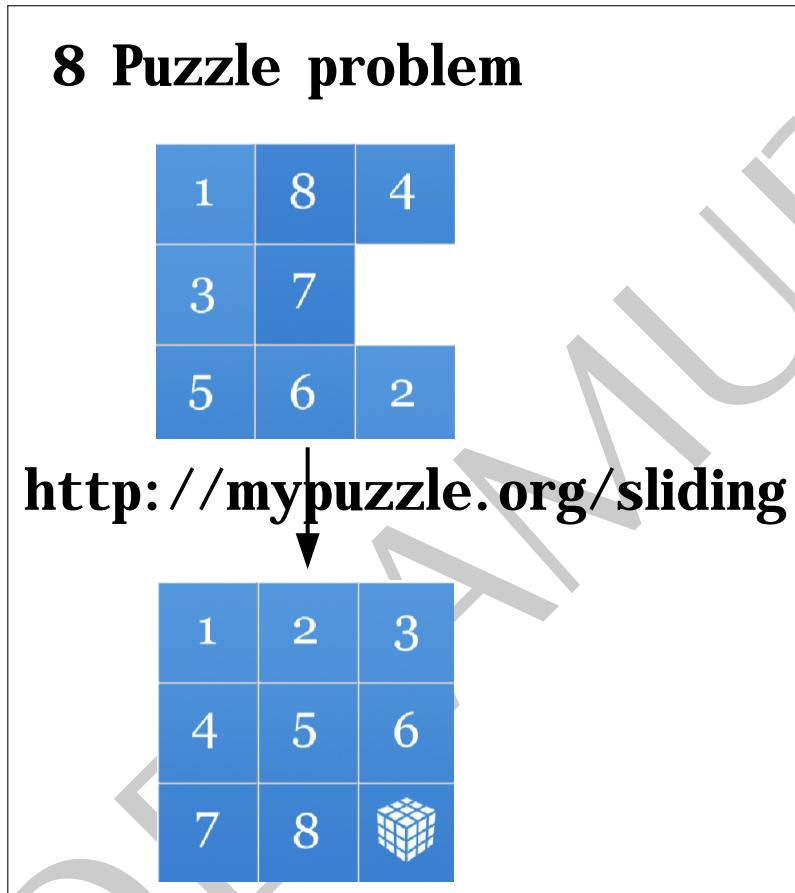


Figure 15.28: 8 puzzle problem

15.19. PROBLEM SET

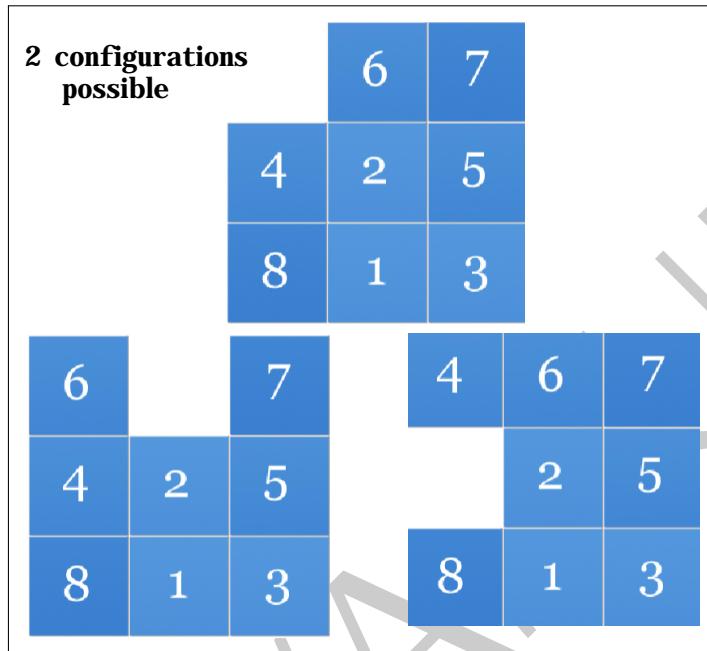


Figure 15.29: Possible two moves

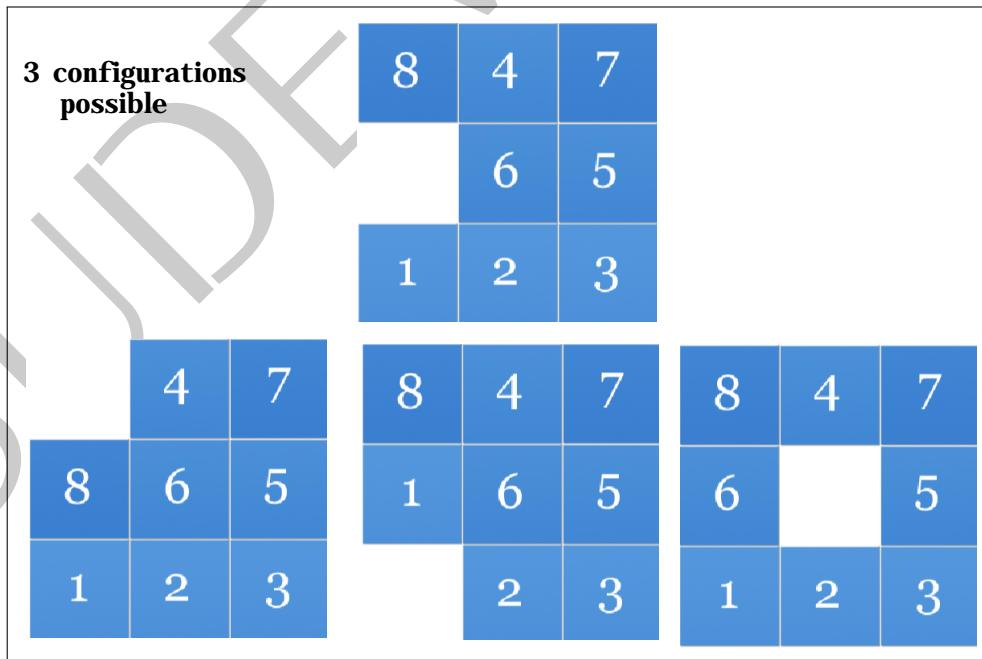


Figure 15.30: Possible three moves

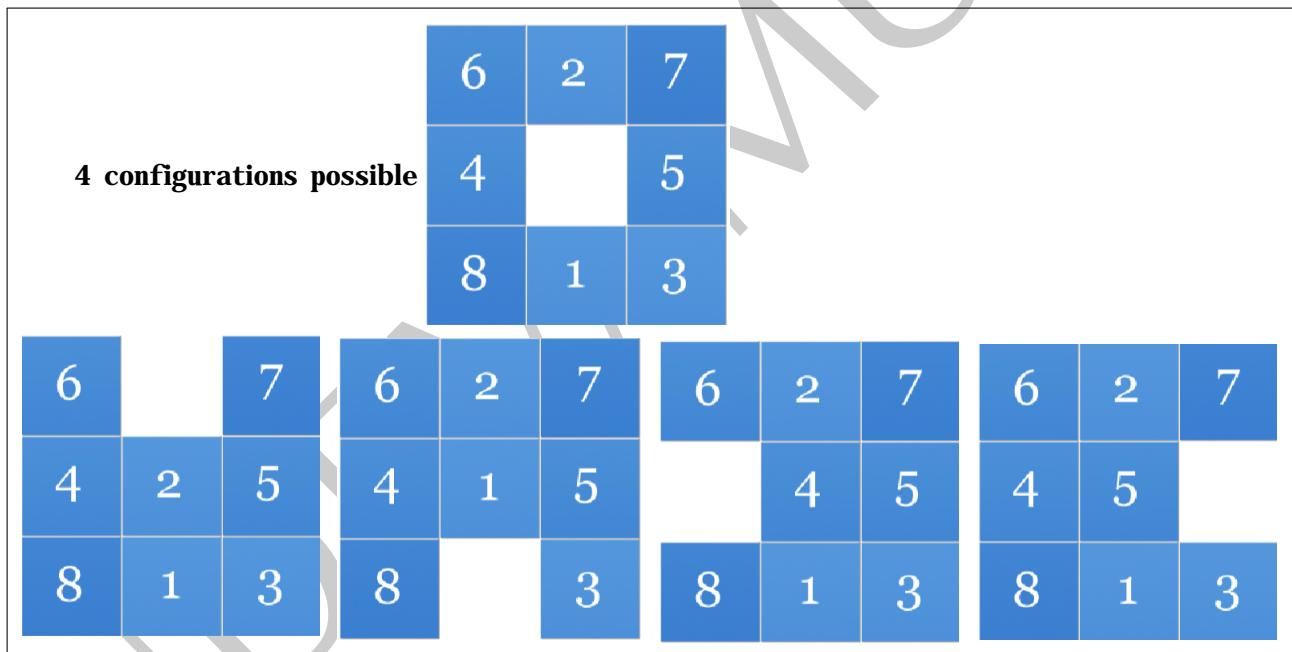


Figure 15.31: Possible four moves

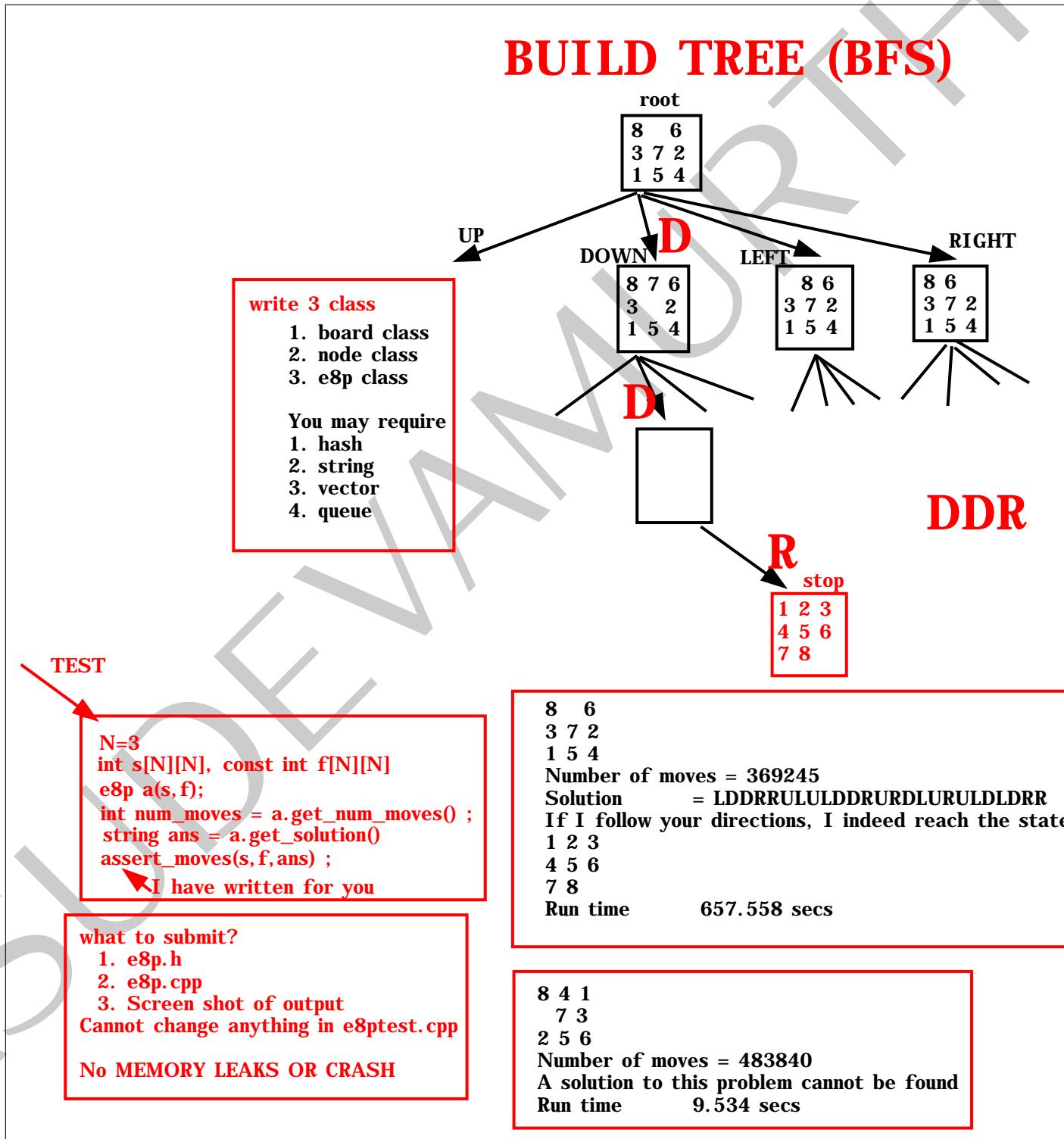


Figure 15.32: Solving the puzzle

Problem 15.19.4. GOOGLE interview question: How do you put a Binary Search Tree in an array in an efficient manner. Hint :: If the node is stored at the i th position and its children are at $2i$ and $2i+1$ (I mean level order wise)Its not the most efficient way.

Problem 15.19.5. GOOGLE interview question: How do you find out the fifth maximum element in an Binary Search Tree in efficient manner. Note: You should not use use any extra space. i.e sorting Binary Search Tree and storing the results in an array and listing out the fifth element

Problem 15.19.6. GOOGLE interview question: Given two binary trees, write a compare function to check if they are equal or not. Being equal means that they have the same value and same structure

VASUDEVAMURTHY

Chapter 16

STL set Class

16.1 Introduction

16.2 STL set class

```
1 /*-----  
2 Copyright (c) 2012 Author: Jagadeesh Vasudevamurthy  
3 Filename: set.cpp  
4 compile: g++ set.cpp  
5 Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)  
6 -----*/  
7  
8 #include <iostream>  
9 using namespace std;  
10 #include <set>  
11 #ifdef _WIN32  
12 #include "vld.h"  
13 #endif  
14  
15 /*-----  
16 An Obj  
17 -----*/  
18 class Obj {  
19 public:  
20     //Constructor  
21     Obj(const char *s) {  
22         cout << "In constructor " << s << endl;  
23         int l = strlen(s);  
24         _size = l;  
25         _dc = new int[l];  
26         for (int i = 0; i < l; ++i) {  
27             _dc[i] = s[i] - '0';  
28         }  
29     }  
30     //Destructor  
31     ~Obj() {  
32         cout << "Object deleted is " << *this;  
33         cout << endl;  
34         _delete();  
35     }  
36  
37     void _delete() {  
38         delete[] _dc;  
39         _dc = 0;  
40     }  
41  
42     void _copy(const Obj& from) {  
43         _size = from._size;  
44         _dc = new int[_size];  
45         cout << "IN COPY CONSTRUCTOR = " << from;  
46         for (int i = 0; i < _size; ++i) {  
47             _dc[i] = from._dc[i];  
48         }  
49         cout << endl;  
50     }  
51  
52     //Copy constructor  
53     Obj(const Obj& rhs) {  
54         _copy(rhs);  
55     }  
56  
57     //Equal operator.  
58     Obj& operator=(const Obj& rhs) {  
59         cout << "IN EQUAL OPERATOR\n";  
60         if (this != &rhs) {  
61             _delete();  
62             _copy(rhs);  
63         }  
64         return *this;  
65     }  
66 }
```

```
67 //print
68 friend ostream& operator<<(ostream& o, const Obj& c) {
69     for (int i = 0; i < c._size; ++i) {
70         cout << c._dc[i];
71     }
72     return o;
73 }
74
75
76 //less than operator
77 friend bool operator<(const Obj& c1, const Obj& c2) {
78     cout << "IN MY < operator " << c1 << " and " << c2 << endl;
79     if (c1._size < c2._size) {
80         return true;
81     }
82     if (c1._size > c2._size) {
83         return false;
84     }
85     //At this point equal size
86     for (int i = 0; i < c1._size; ++i) {
87         if (c1._dc[i] < c2._dc[i]) {
88             return true;
89         }
90         if (c1._dc[i] > c2._dc[i]) {
91             return false;
92         }
93     }
94     //At this point, exactly same
95     return false;
96     //Remember this rule: ALWAYS HAVE COMPARISON FUNCTION RETURN FALSE FOR EQUAL VALUE
97 }
98
99 //== operator
100 friend bool operator==(const Obj& c1, const Obj& c2) {
101     cout << "IN MY == operator " << c1 << " and " << c2 << endl;
102     return (!(c1 < c2) && !(c2 < c1));
103 }
104
105 //!= operator
106 friend bool operator!=(const Obj& c1, const Obj& c2) {
107     cout << "IN MY != operator " << c1 << " and " << c2 << endl;
108     return !(c1 == c2);
109 }
110
111 private:
112     int _size;
113     int* _dc;
114 };
115
116 /*-----*/
117 find
118 -----
119 static bool find(const set<Obj>& s, const Obj& o) {
120     cout << "In find " << o << endl;
121     set<Obj>::iterator itt = s.find(o);
122     return (itt != s.end()) ? true : false;
123 }
124
125 /*-----*/
126 test
127 -----
128 void test() {
129     set<Obj> s;
130     Obj o1("1245");
131     Obj o2("1345");
132     if (o1 == o2) {
```

```
133     cout << "(o1 == o2)" << endl;
134 }
135 else {
136     cout << "(o1 != o2)" << endl;
137 }
138 if (o1 != o2) {
139     cout << "(o1 != o2)" << endl;
140 }
141 else {
142     cout << "(o1 == o2)" << endl;
143 }
144 //From this point, put a break point in == and != operator
145 //Set never call == operator or != operator even if you written one.
146 //It calls only < operator
147 s.insert(o1);
148 s.insert(o2);
149
150 Obj f("12451");
151 if (find(s, f)) {
152     cout << "Already there\n";
153 }
154 else {
155     cout << "NOT there\n";
156 }
157
158 Obj f1("1245");
159 if (find(s, f1)) {
160     cout << "Already there\n";
161 }
162 else {
163     cout << "NOT there\n";
164 }
165 }
166
167 /*
168 main
169 -----
170 int main() {
171     test();
172     return 1;
173 }
174
175
176 //EOF
177
178
179
180
181
182
183
184
```

Chapter 17

STL map Class

17.1 Introduction

17.2 STL map class

map<Key, Data, Compare, Alloc>

Maps are a kind of associative containers that stores elements formed by the combination of a key value and a mapped value

1. Unique key values: no two elements in the map have keys that compare equal to each other

Key: Type of the key values. Each element in a map is uniquely identified by its key value

2. Each element is composed of a key and a mapped value

Value: T: Type of the mapped value. Each element in a map is used to store some data as its mapped value

```
typedef pair<const Key, T> value_type;
map<Key, T>::iterator it;
```

3. Elements follow a strict weak ordering at all times

```
it->first;           // same as (*it).first (the key value)
it->second;          // same as (*it).second (the mapped value)
```

Compare: Comparison class: A class that takes two arguments of the key type and returns a bool. The expression `comp(a,b)`, where `comp` is an object of this comparison class and `a` and `b` are key values, shall return true if `a` is to be placed at an earlier position than `b` in a strict weak ordering operation. This can either be a class implementing a function call operator or a pointer to a function.

```
template < class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T> > > class map;
```

Element access:

`insert` Insert element (public member function)
`operator[]` Access element (public member function)

Find:

`find` Get iterator to element (public member function)

Figure 17.1: STL map

17.3 Testing map code

```
1 /*-----  
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy  
3 Filename: map.h  
4 -----*/  
5  
6 #ifndef map_h  
7 #define map_h  
8  
9 #include <iostream>  
10 #include <string>  
11 #include <map>  
12 using namespace std;  
13  
14 #ifdef _WIN32  
15 #include "c:/work/alg/course/code/c/complex.h"  
16 #else  
17 #include "/home/jag/jag/alg/c/complex.h"  
18 #endif  
19  
20 /*-----  
21 Comparison function for the associate container  
22 Note that this Comparison function isn't operator < or even less.  
23 It is user defined predicate. (Page 85,effective STL)  
24 -----*/  
25 template <class T>  
26 struct value_based_user_defined_predicate {  
27     //Compare function. The object should have < operator defined  
28     bool operator()(const T& left, const T& right) const {  
29         // this should implement a total ordering on MyClass, that is  
30         // it should return true if "left" precedes "right" in the ordering  
31  
32         // We are relying on < opearator of the class as a predicate  
33         // You can rewrite it.  
34  
35         //Associate containers NEVER uses == operator  
36         // ITEM 21  
37         //Always have comparison functions return false for equal values  
38         return (left < right) ;  
39     }  
40 };  
41  
42 /*-----  
43 T must pointer to an object.  
44 It looks at the guts comparison  
45 -----*/  
46 template <class T>  
47 struct content_based_user_defined_predicate {  
48     bool operator()(const T& left, const T& right) const {  
49         //Get the object and use the  value_based_user_defined_predicate  
50         return (*left < *right) ;  
51     }  
52 };  
53  
54 /*-----  
55 typedef for <string,int>  
56 -----*/  
57 typedef pair<string, int> string_int_pair ;  
58 typedef map<string, int, value_based_user_defined_predicate<const string> > hash_string_int ;  
59 typedef hash_string_int::const_iterator hash_string_int_itr ;  
60  
61 #endif  
62
```

```
1 /*-----  
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy  
3 Filename: map.cpp  
4 compile: g++ map.cpp  
5 Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)  
6 -----*/  
7 #include "map.h"  
8  
9 /*-----  
10 Difference between map and map  
11  
12 map uses a red-black tree as the data structure, so the elements you put in there are sorted,  
13 and insert/delete is O(log(n)). The elements need to be implement at least operator<.  
14  
15 hashmap uses a hash, so elements are unsorted, insert/delete is O(1).  
16 Elements need to implement at least operator== and you need a hash function.  
17 -----*/  
18  
19 /*-----  
20 static definition - only once at the start  
21 Change to false, if you don't need verbose  
22 -----*/  
23 bool complex::_display = true ;  
24  
25  
26 /*-----  
27 test names  
28 -----*/  
29 static void test_names() {  
30     map<string,char> grades;  
31  
32     // Illustrating insertion to map  
33     grades["john"] = 'a' ;  
34     grades["josh"] = 'd' ;  
35  
36     //Illustrating Insertion to map  
37     grades.insert(pair<string,char>("robert",'b')) ;  
38     pair<map<string,char>::iterator, bool> res = grades.insert(pair<string,char>("adam",'b')) ;  
39     if (res.second) {  
40         cout << "adam is inserted " << endl ;  
41     }  
42  
43     // Illustrating map requires unique key  
44     pair<map<string,char>::iterator, bool> res1 = grades.insert(pair<string,char>("adam",'c')) ;  
45     if (res1.second) {  
46         cout << "adam is inserted " << endl ;  
47     }else {  
48         cout << "adam is NOT inserted as it is already there " << endl ;  
49     }  
50  
51     // Illustrating changing value  
52     // Now adam improves grade to a from b  
53     grades.erase("adam") ;  
54     pair<map<string,char>::iterator, bool> res2 = grades.insert(pair<string,char>("adam",'a')) ;  
55     if (res2.second) {  
56         cout << "adam is inserted " << endl ;  
57     }else {  
58         cout << "adam is NOT inserted as it is already there " << endl ;  
59     }  
60  
61  
62     //Illustrating map traversal  
63     map<string,char>::iterator p = grades.begin() ;  
64     while(p != grades.end()) {  
65         cout << p->first << " " << p->second << endl ;  
66         p++ ;
```

```
67 }
68 //Illustrating Finding a key in hash
69 if (grades.find("tom") == grades.end()) {
70     cout << "tom is NOT in the hash" << endl ;
71 }else {
72     cout << "tom is in the hash" << endl ;
73 }
74 if (grades.find("adam") == grades.end()) {
75     cout << "adam is NOT in the hash" << endl ;
76 }else {
77     cout << "adam is in the hash" << endl ;
78 }
79 }
80 //Illustrating Finding a key in hash using []
81 cout << "Grade of josh is " << grades["josh"] << endl ;
82 //Illustrating Finding a key in hash using find
83 map<string,char>::iterator itr1 = grades.find("josh") ;
84 if (itr1 != grades.end()) {
85     cout << "Grade of josh is " << itr1->second << endl ;
86 }
87 itr1 = grades.find("jag") ;
88 if (itr1 != grades.end()) {
89     cout << "Grade of jag is " << itr1->second << endl ;
90 }else {
91     cout << "Student jag is not there. Hence he has no grade\n" ;
92 }
93 }
94 //If the item is NOT there, the key is automatically inserted
95 //Value will zero for built-in type
96 //Value will be default constructor for UDT
97 cout << "Grade of jag is " << grades["jag"] << endl ;
98 }
99 //Illustrating size of hash
100 cout << "Number of element is hash is: " << grades.size() << endl ;
101
102 //Illustrating cleaing hash
103 grades.clear() ;
104 cout << "Number of element is hash is after clear : " << grades.size() << endl ;
105 }
106 }
107 */
108 -----
109 test_names_using_your_hash_function
110 -----
111 static void test_names_using_your_hash_function() {
112     map<string,char,value_based_user_defined_predicate<const string> > grades;
113
114     // Illustrating insertion to map
115     grades["john"] = 'a' ;
116     grades["josh"] = 'd' ;
117
118     //Illustrating Insertion to map
119     grades.insert(pair<string,char>("robert",'b')) ;
120     pair<map<string,char,value_based_user_defined_predicate<const string> >::iterator, bool> res = grades
121         .insert(pair<string,char>("adam",'b')) ;
122     if (res.second) {
123         cout << "adam is inserted " << endl ;
124     }
125
126     // Illustrating map requires unique key
127     pair<map<string,char,value_based_user_defined_predicate<const string> >::iterator, bool> res1 =
128         grades.insert(pair<string,char>("adam",'c')) ;
129     if (res1.second) {
130         cout << "adam is inserted " << endl ;
131     }else {
132         cout << "adam is NOT inserted as it is already there " << endl ;
```

```
131 }
132 // Illustrating changing value
133 // Now adam improves grade to a from b
134 grades.erase("adam") ;
135 pair<map<string,char,value_based_user_defined_predicate<const string> ::iterator, bool> res2 =
136 grades.insert(pair<string,char>("adam",'a')) ;
137 if (res2.second) {
138     cout << "adam is inserted " << endl ;
139 }else {
140     cout << "adam is NOT inserted as it is already there " << endl ;
141 }
142
143
144 //Illustrating map traversal
145 map<string,char,value_based_user_defined_predicate<const string> ::iterator p = grades.begin() ;
146 while(p != grades.end()) {
147     cout << p->first << " " << p->second << endl ;
148     p++ ;
149 }
150
151 //Illustrating Finding a key in hash
152 if (grades.find("tom") == grades.end()) {
153     cout << "tom is NOT in the hash" << endl ;
154 }else {
155     cout << "tom is in the hash" << endl ;
156 }
157 if (grades.find("adam") == grades.end()) {
158     cout << "adam is NOT in the hash" << endl ;
159 }else {
160     cout << "adam is in the hash" << endl ;
161 }
162
163 //Illustrating Finding a key in hash using []
164 cout << "Grade of josh is " << grades["josh"] << endl ;
165 //Illustrating Finding a key in hash using find
166 map<string,char,value_based_user_defined_predicate<const string> ::iterator itr1 = grades.find("josh")
167 ");
168 if (itr1 != grades.end()) {
169     cout << "Grade of josh is " << itr1->second << endl ;
170 }
171 itr1 = grades.find("jag") ;
172 if (itr1 != grades.end()) {
173     cout << "Grade of jag is " << itr1->second << endl ;
174 }else {
175     cout << "Student jag is not there. Hence he has no grade\n" ;
176 }
177
178 //If the item is NOT there, the key is automatically inserted
179 //Value will zero for built-in type
180 cout << "Grade of jag is " << grades["jag"] << endl ;
181
182 //Illustrating size of hash
183 cout << "Number of element is hash is: " << grades.size() << endl ;
184
185 //Illustrating cleaing hash
186 grades.clear() ;
187 cout << "Number of element is hash is after clear : " << grades.size() << endl ;
188 }
189
190 /-----
191 test dictionary
192 -----
193 static void test_dictionary() {
194     hash_string_int d;
```

```
195 #ifdef _WIN32
196     ifstream in("C:/work/alg/course/code/c/data/english.dat") ;
197 #else
198     ifstream in("/home/jag/jag/alg/c/data/english.dat") ;
199 #endif
200     assert(in) ;
201     char buffer[1024] ;
202     int i = 0 ;
203     while (!in.getline(buffer,1024).eof()) {
204         string str(buffer) ;
205         int l = str.size() ;
206         string_int_pair p(str,l) ;
207         d.insert(p) ;
208     }
209 }
210
211
212
213 /*-----*
214 test_udt
215 -----*/
216 static void test_udt() {
217     map<complex,char,value_based_user_defined_predicate<const complex>> myhash;
218
219 // Illustrating insertion to map
220 for (int i = 0; i < 26; i++) {
221     complex c(i,-i) ;
222     myhash[c] = i+'a' ;
223 }
224
225 //Illustrating map traversal
226 map<complex,char,value_based_user_defined_predicate<const complex>>::iterator p = myhash.begin() ;
227 while(p != myhash.end()) {
228     cout << p->first << " " << p->second << endl ;
229     p++ ;
230 }
231
232 //illustring find
233 {
234     {
235         complex c(24,-24) ;
236         map<complex,char,value_based_user_defined_predicate<const complex>>::iterator p = myhash.find(c) ;
237         if (p == myhash.end()) {
238             cout << "The object " << c << " is not in the hash\n" ;
239         }else {
240             cout << "The object " << c << " exists. " << "The value associated with this object is = " << p->second << endl ;
241         }
242     }
243     {
244         complex c(214,-24) ;
245         map<complex,char,value_based_user_defined_predicate<const complex>>::iterator p = myhash.find(c) ;
246         if (p == myhash.end()) {
247             cout << "The object " << c << " is not in the hash\n" ;
248         }else {
249             cout << "The object " << c << " exists. " << "The value associated with this object is = " << p->second << endl ;
250         }
251     }
252 }
253 }
254
255 /*-----*
256 test pointer udt
```

```
257 -----*/
258 static void test_pointer_to_udt() {
259     map<complex*,char,value_based_user_defined_predicate<const complex*> > myhash;
260     complex* c24 = NULL ;
261     complex* c214 = new complex(241,-24) ;
262
263     // Illustrating insertion to map
264     for (int i = 0; i < 26; i++) {
265         complex* c = new complex(i,-i) ;
266         myhash[c] = i+'a' ;
267         if (i == 24) {
268             c24 = c ;
269         }
270     }
271
272     //Illustrating map traversal
273     map<complex*,char,value_based_user_defined_predicate<const complex*> ::iterator p = myhash.begin() ;
274     while(p != myhash.end()) {
275         cout << p->first << " " << p->second << endl ;
276         p++ ;
277     }
278
279     //illustring find
280     {
281         complex* c = c24 ;
282         {
283             map<complex*,char,value_based_user_defined_predicate<const complex*> ::iterator p = myhash.find (c) ;
284             if (p == myhash.end()) {
285                 cout << "The object " << c << " is not in the hash\n" ;
286             }else {
287                 cout << "The object " << c << " exists. " << "The value associated with this object is = " << p->second << endl ;
288             }
289         }
290         {
291             complex* c = c214;
292             map<complex*,char,value_based_user_defined_predicate<const complex*> ::iterator p = myhash.find (c) ;
293             if (p == myhash.end()) {
294                 cout << "The object " << c << " is not in the hash\n" ;
295             }else {
296                 cout << "The object " << c << " exists. " << "The value associated with this object is = " << p->second << endl ;
297             }
298         }
299     }
300
301     //free all the memory
302     {
303         map<complex*,char,value_based_user_defined_predicate<const complex*> ::iterator p = myhash.begin () ;
304         while(p != myhash.end()) {
305             delete (p->first) ;
306             p++ ;
307         }
308     }
309     delete c214 ;
310     //You should not delete c24. Why ?
311 }
312
313 */
314 test pointer udt
315 -----
316 static void test_pointer_to_udt_find_through_object() {
317     map<complex*,char,content_based_user_defined_predicate<const complex*> > myhash;
```

```
318 // Illustrating insertion to map
319 for (int i = 0; i < 26; i++) {
320     complex* c = new complex(i,-i) ;
321     myhash[c] = i+'a' ;
322 }
323
324
325 //Illustrating map traversal
326 map<complex*,char,content_based_user_defined_predicate<const complex*> ::iterator p = myhash.begin() ↵
327 ;
328 while(p != myhash.end()) {
329     cout << p->first << " " << p->second << endl ;
330     p++ ;
331 }
332
333 //illustring find
334 {
335     complex c(24,-24) ;
336     map<complex*,char,content_based_user_defined_predicate<const complex*> ::iterator p = myhash. ↵
337     find(&c) ;
338     if (p == myhash.end()) {
339         cout << "The object " << c << " is not in the hash\n" ;
340     }else {
341         cout << "The object " << c << " exists. " << "The value associated with this object is = " << ↵
342         p->second << endl ;
343     }
344     {
345         complex c(214,-24) ;
346         map<complex*,char,content_based_user_defined_predicate<const complex*> ::iterator p = myhash. ↵
347         find(&c) ;
348         if (p == myhash.end()) {
349             cout << "The object " << c << " is not in the hash\n" ;
350         }else {
351             cout << "The object " << c << " exists. " << "The value associated with this object is = " << ↵
352             p->second << endl ;
353         }
354     }
355
356 //free all the memory
357 {
358     map<complex*,char,content_based_user_defined_predicate<const complex*> ::iterator p = myhash. ↵
359     begin() ;
360     while(p != myhash.end()) {
361         delete (p->first) ;
362         p++ ;
363     }
364
365
366 /*-----*/
367 main
368 -----*/
369 int main() {
370     test_names() ;
371     test_names_using_your_hash_function() ;
372     test_udt() ;
373     test_pointer_to_udt() ;
374     test_pointer_to_udt_find_through_object() ;
375     test_dictionary();
376     return 1 ;
377 }
```

17.4 Problem set

VASUDEVAMURTHY

Chapter 18

Graph

18.1 Introduction

18.2 Graph applications

18.2.1 Transportation problem

18.2. GRAPH APPLICATIONS

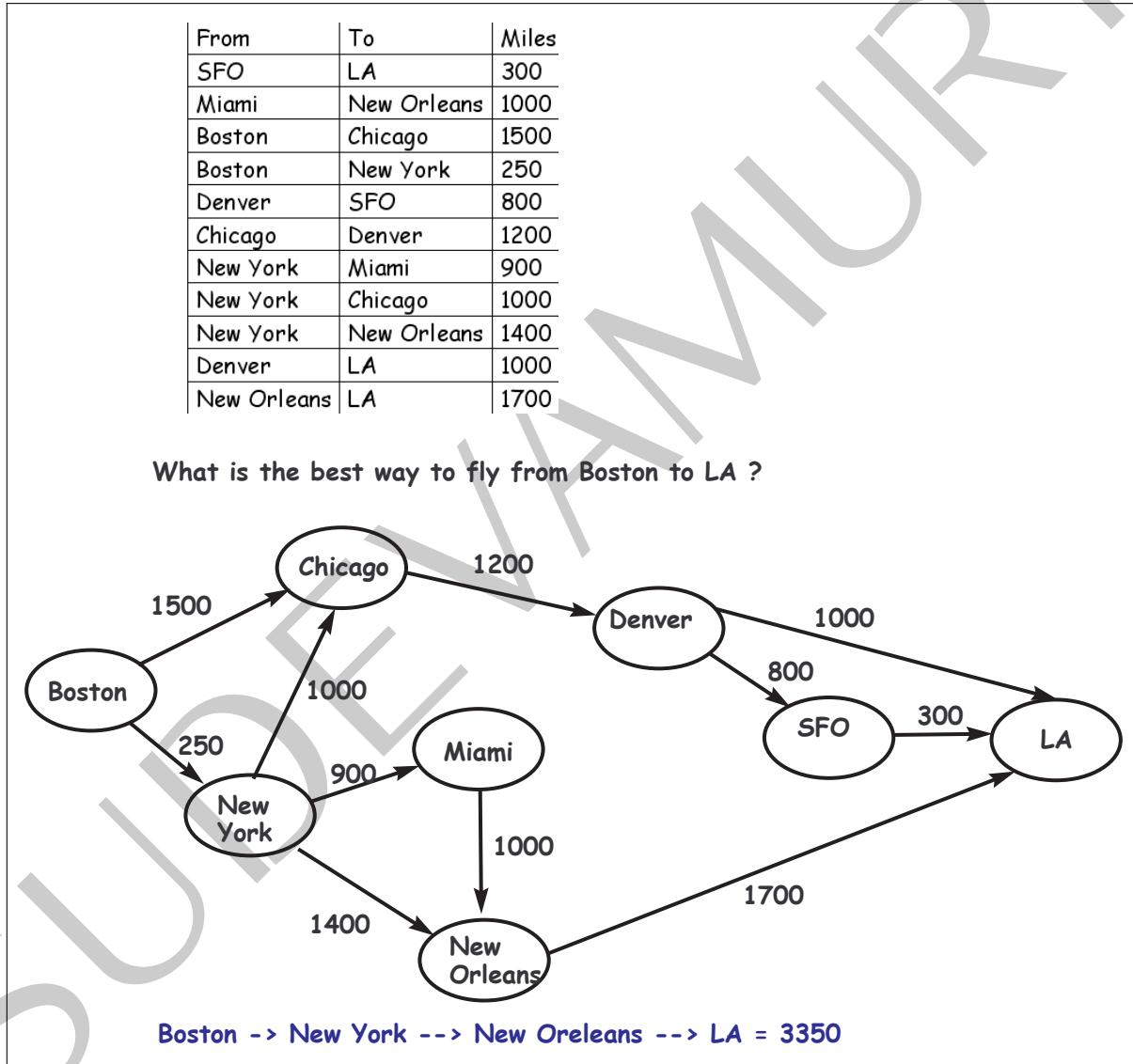


Figure 18.1: What is the best way to fly from Boston to LA

18.2.2 Travelling salesman's problem

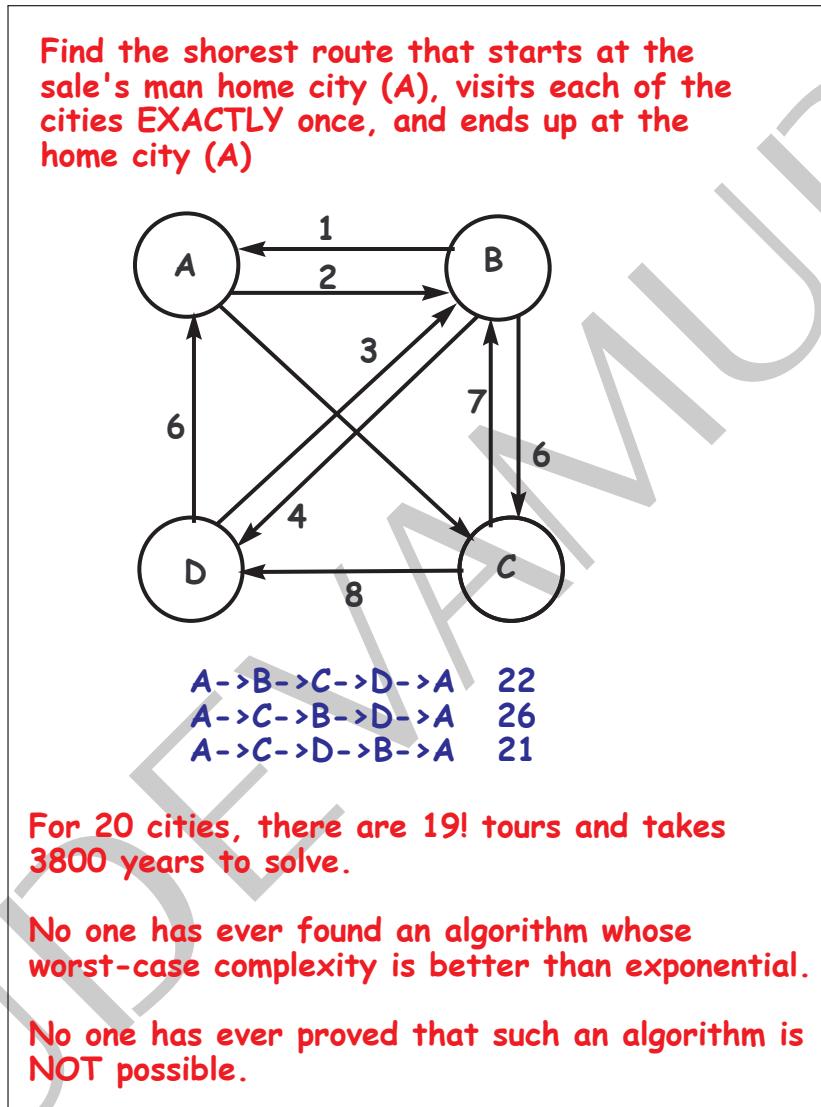


Figure 18.2: Travelling salesman problem

18.2.3 Minimum connector problem

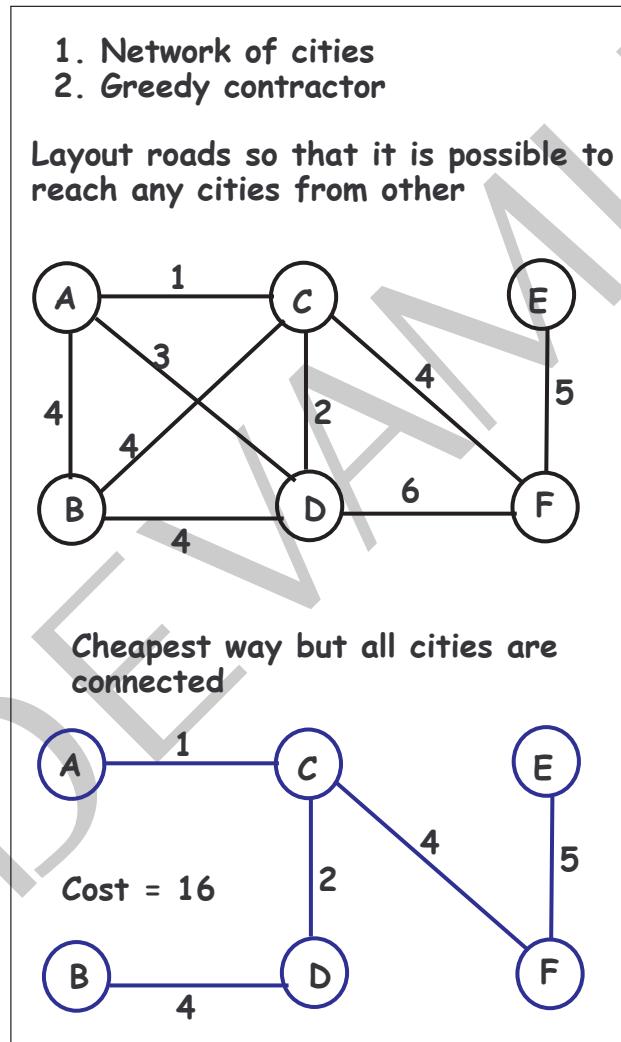


Figure 18.3: Laying cheapest road

18.2.4 Scheduling problem

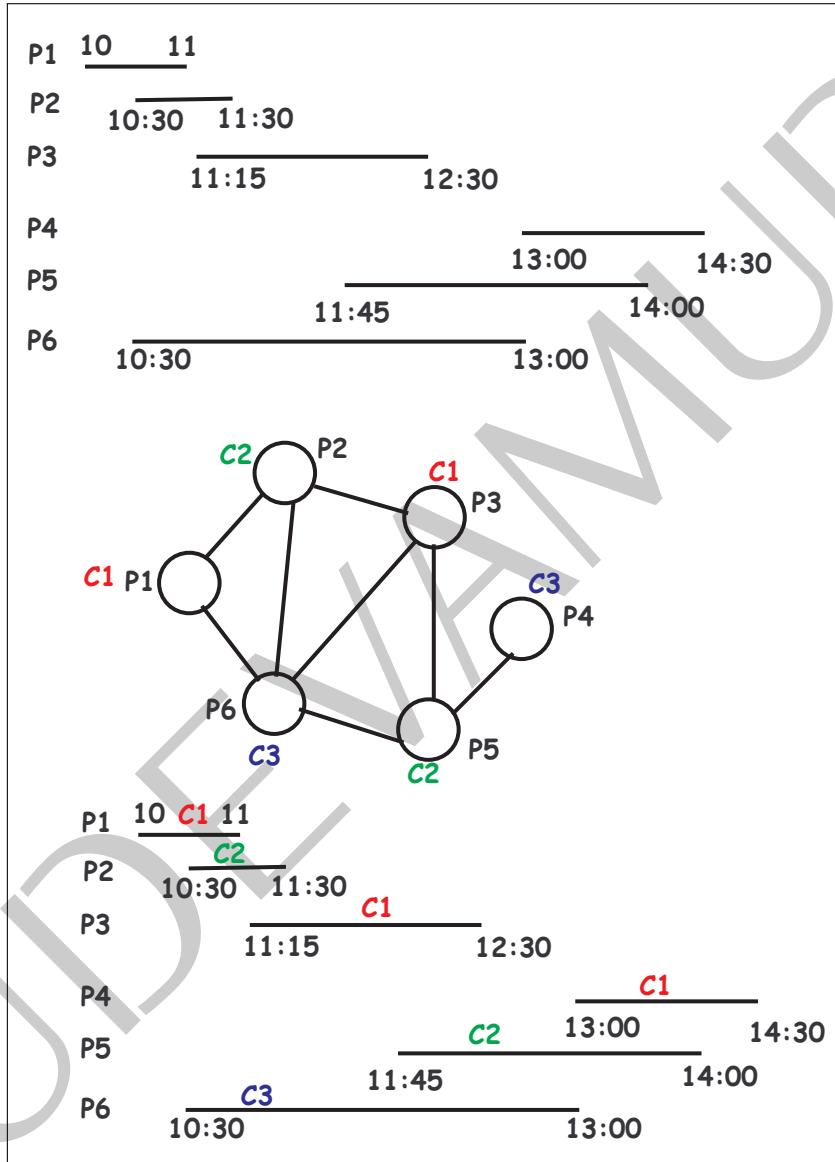


Figure 18.4: Minimum channels required to broadcast seven programs

18.2.5 Activity network or Topological sorting problem

18.2. GRAPH APPLICATIONS

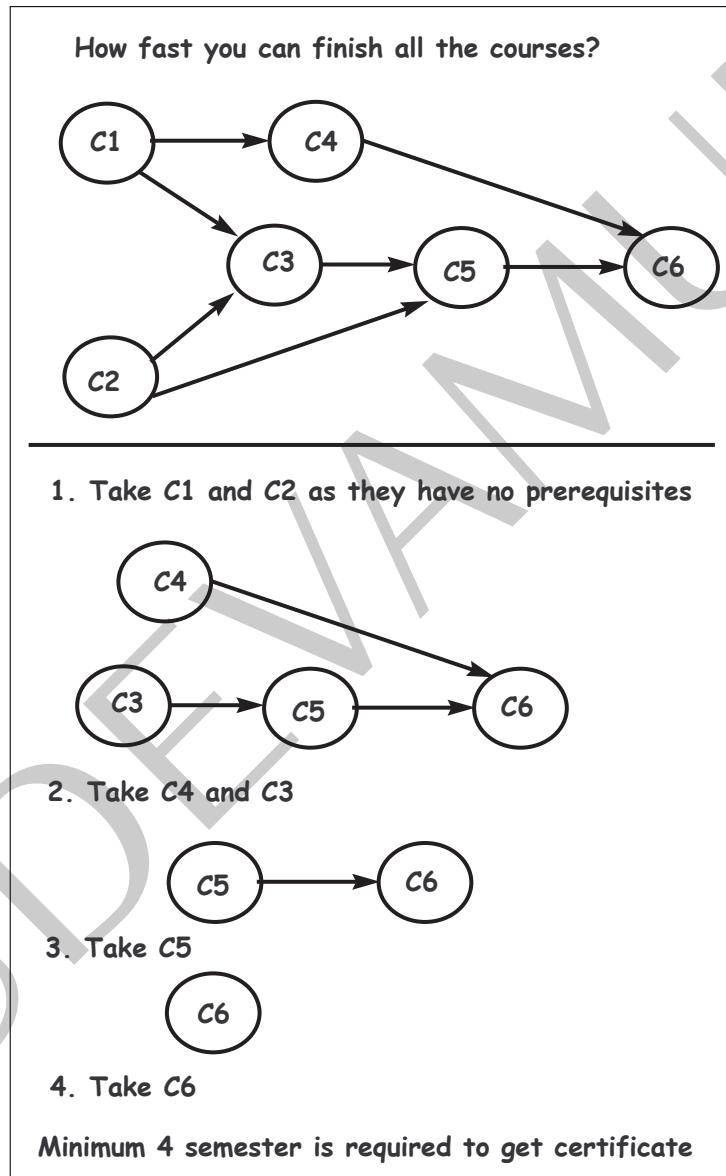


Figure 18.5: Completing courses in an university

18.2.6 Critical path analysis

18.2. GRAPH APPLICATIONS

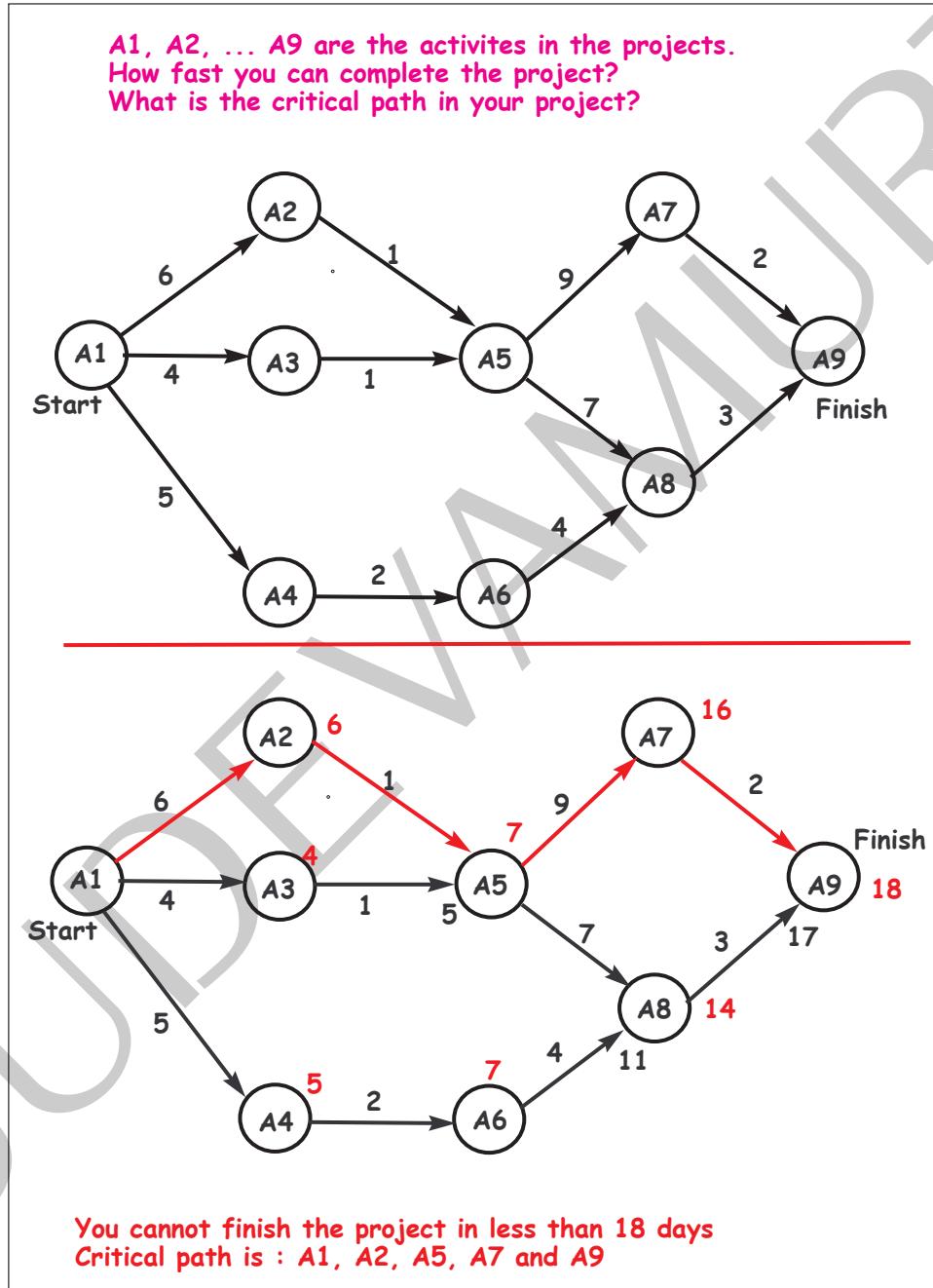


Figure 18.6: Critical path of a project

18.2.7 Flow problem

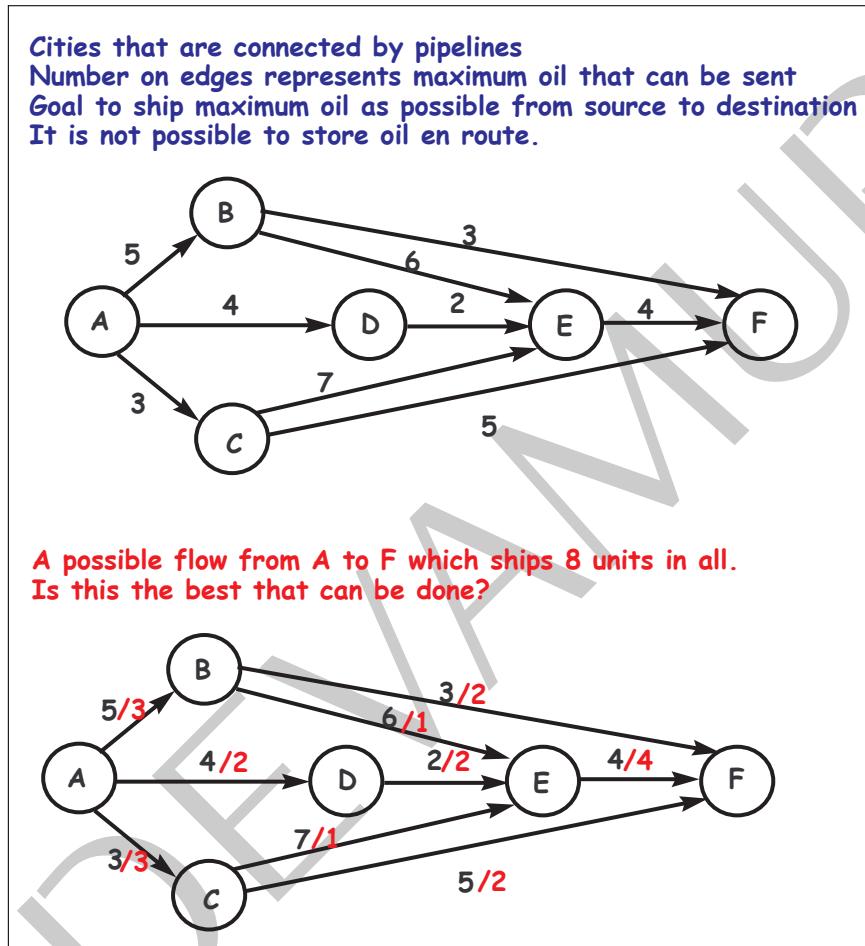


Figure 18.7: Maximum flow possible

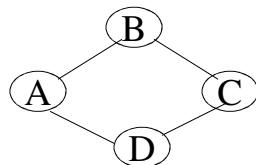
Graph

A Graph G consists of two sets:

Set V is called as a set of nodes or vertices.

Set E is called as a set of edges or arcs.

This set E is the set of pair of elements from V .



$$\begin{aligned}G &= \{V, E\} \\V &= \{A, B, C, D\} \\E &= \{(A, B), (A, D), (B, C), (D, C)\}\end{aligned}$$

	A	B	C	D
A	0	1	0	1
B	1	0	1	0
C	0	1	0	1
D	1	0	1	0

4 vertices – Size of matrix is 16

Adjacency Matrix

4 edges – There are 8 ones in matrix

6/12/2011

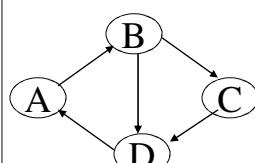
Jagadeesh Vasudevamurthy

Digraph

Diagraph G is a directed graph consisting of two sets:

Set V is called as a set of nodes or vertices.

Set E is called as a set of edges or arcs. This set E is the *ordered* set of pairs of elements of V .



$$\begin{aligned}G &= \{V, E\} \\V &= \{A, B, C, D\} \\E &= \{(A, B), (B, D), (B, C), (C, D), (D, A)\}\end{aligned}$$

	A	B	C	D
A	0	1	0	0
B	0	0	1	1
C	0	0	0	1
D	1	0	0	0

4 vertices – Size of matrix is 16

Adjacency Matrix

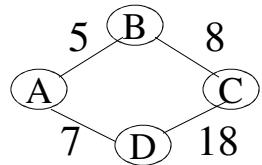
5 edges – 5 ones in the matrix

6/12/2011

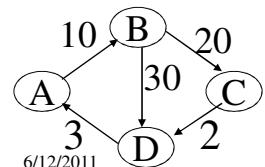
Jagadeesh Vasudevamurthy

Weighted Graph(Digraph)

A Graph(Digraph) is termed as weighted graph(digraph) if all the edges in it are labeled with some weights.



	A	B	C	D
A	-	5	-	7
B	5	-	8	-
C	-	8	-	18
D	7	-	18	-

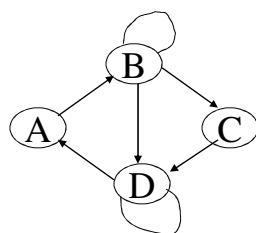


	A	B	C	D
A	-	10	-	-
B	-	-	20	30
C	-	-	-	2
D	-	-	-	-

Jagadeesh Vasudevamurthy

Self loop

- If there is an edge whose starting and ending vertices are same, $e=\{vk, vk\}$ is an edge, then it is called a self loop.



	A	B	C	D
A	0	1	0	0
B	0	1	1	1
C	0	0	1	1
D	1	0	0	0

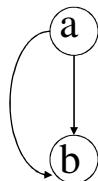
Diagonal 1 means self loop

6/12/2011

Jagadeesh Vasudevamurthy

Parallel Edges

- If there are more than one edge between same pair of vertices, then that edge is called a parallel edge.



	a	b
a	0	1
b	0	0

Cannot represent

6/12/2011

Jagadeesh Vasudevamurthy

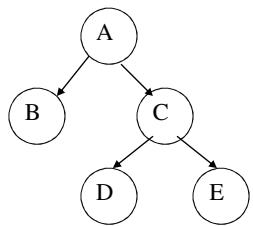
Multigraph and Simple Graph

- A graph which has either self loop, parallel edges or both is called a multigraph
- A graph which does not have either self loop or parallel edges is called simple graph.
- A directed graph that has no cycles is called a Directed Acyclic Graph (DAG)

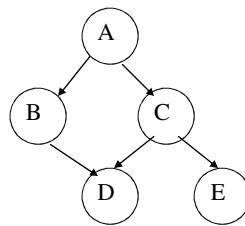
6/12/2011

Jagadeesh Vasudevamurthy

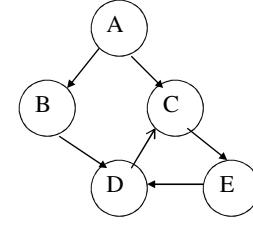
Tree, DAG and Digraph



TREE



Digraph with no cycle



DAG

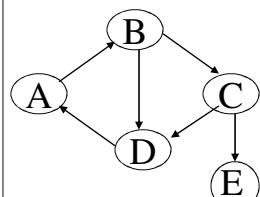
All the three graphs above are Digraphs – Directed graphs

6/12/2011

Jagadeesh Vasudevamurthy

Adjacent Vertices

Vertex vi is adjacent to another vertex vj ,
if there is an edge from vi to vj



A is adjacent to B

B is adjacent to C and D

C is adjacent to D and E

D is adjacent to A

E is empty. Has no Adjacent vertices

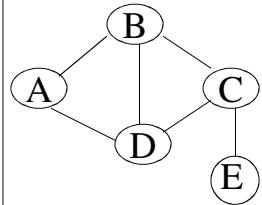
	A	B	C	D	E
A	1				
B			1	1	
C				1	1
D	1				
E					

6/12/2011

Jagadeesh Vasudevamurthy

Adjacent Vertices(Undirected Graph)

Vertex vi is adjacent to another vertex vj ,
if there is an edge from vi to vj



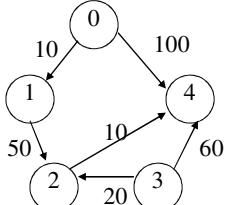
- A is adjacent to B,D
- B is adjacent to A,D,C
- C is adjacent to B,D,E
- D is adjacent to A,B,C
- E is adjacent to C

	A	B	C	D	E
A	1			1	
B	1		1	1	
C		1		1	1
D	1	1	1		
E			1		

6/12/2011

Jagadeesh Vasudevamurthy

Fanout and Fanin(Directed Graph)



	0	1	2	3	4
0		1			1
1				1	
2					1
3			1		1
4					

Column of i gives fanins of vertex i

Ex: Vertex 0 has no fanins and vertex 4 has 3 fanins - 0,2 and 3

Row of i gives fanout of vertex I

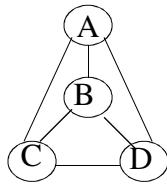
Ex: Vertex 4 has no fanouts and vertex 3 has fanouts 2 and 4

6/12/2011

Jagadeesh Vasudevamurthy

Complete Graph

- A graph(digraph) G is said to be complete if each vertex v_i is adjacent to every other vertex v_j in G .



	A	B	C	D
A		1	1	1
B	1		1	1
C	1	1		1
D	1	1	1	

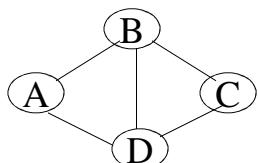
All are 1's except diagonal entry

6/12/2011

Jagadeesh Vasudevamurthy

Degree of a vertex

- The number of edges connected with a vertex v_i is called the degree of vertex v_i and is denoted as $\text{degree}(v_i)$



	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	1	0	1
D	1	1	1	0

2	3	2	3
---	---	---	---

A has a degree of 2

B has a degree of 3

C has a degree of 2

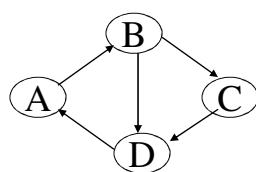
D has a degree of 3

6/12/2011

Jagadeesh Vasudevamurthy

Degree of a Vertex in Digraph

In Degree(vi) = Number of edges going into vi
Out Degree(vi) = Number of edges going out of vi



	A	B	C	D
A	0	1	0	0
B	0	0	1	1
C	0	0	0	1
D	1	0	0	0

1	Out degree of A
2	Out degree of B
1	Out degree of C
1	Out degree of D

In Degree of

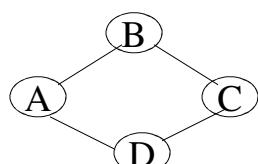
A B C D

6/12/2011

Jagadeesh Vasudevamurthy

Transpose of Graph

- If $A = AT$ then G is a Simple Undirected Graph.



A

	A	B	C	D
A	0	1	0	1
B	1	0	1	0
C	0	1	0	1
D	1	0	1	0

AT

	A	B	C	D
A	0	1	0	1
B	1	0	1	0
C	0	1	0	1
D	1	0	1	0

6/12/2011

Jagadeesh Vasudevamurthy

Matrix Multiplication

- http://www.bluebit.gr/matrix-calculator/matrix_multiplication.aspx

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix}$$

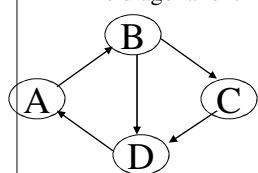
In this case, we multiply a 2×2 matrix by a 2×2 matrix and we get a 2×2 matrix as the result.

6/12/2011

Jagadeesh Vasudevamurthy

Transpose of Digraph

- $A \neq AT$
- The diagonal entries of $A \cdot AT$ gives out degree of all vertices of G
- The diagonal entries of $AT \cdot A$ gives in degree of all vertices of G



$$\begin{array}{l} A * A[T] \\ \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \end{array} \quad \begin{array}{l} A[T] * A \\ \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 2 \end{matrix} \end{array}$$

	A	B	C	D	
A	0	1	0	0	1
B	0	0	1	1	2
C	0	0	0	1	1
D	1	0	0	0	1

	A	B	C	D	
A	0	0	0	1	
B	1	0	0	0	
C	0	1	0	0	
D	0	1	1	0	

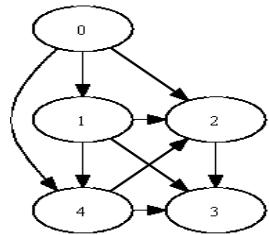
indegree

1	1	1	2
---	---	---	---

6/12/2011

Jagadeesh Vasudevamurthy

Graph and Adjacency matrix of a Directed graph



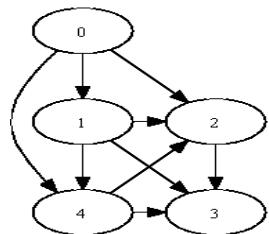
0 1 1 0 1
0 0 1 1 1
0 0 0 1 0
0 0 0 0 0
0 0 1 1 0

```
digraph g {  
    0 -> 1  
    0 -> 2  
    0 -> 4  
    1 -> 2  
    1 -> 3  
    1 -> 4  
    2 -> 3  
    4 -> 2  
    4 -> 3  
}
```

6/12/2011

Jagadeesh Vasudevamurthy

Out degree and in degree



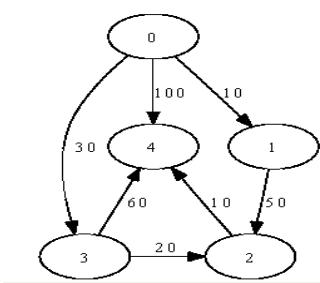
3 2 0 0 1 0 0 0 0 0
2 3 1 0 2 0 1 1 0 1
0 1 1 0 1 0 1 3 2 2
0 0 0 0 0 0 0 2 3 1
1 2 1 0 2 0 1 2 1 2

outdegree of vertex 0 = 3
outdegree of vertex 1 = 3
outdegree of vertex 2 = 1
outdegree of vertex 3 = 0
outdegree of vertex 4 = 2

6/12/2011

Jagadeesh Vasudevamurthy

Graph and Adjacency matrix of a Weighted Directed graph



OUTDEGREE
0 1 0 0 30 100
0 0 50 0 0
0 0 0 0 10
0 0 20 0 60
0 0 0 0 0

INDEGREE
0 0 0 0 0
0 1 0 1 1
0 1 0 1 1
0 0 2 0 1
0 1 0 1 1
0 1 1 1 3

WARSHEL
0 1 0 1 1
0 0 1 0 0
0 0 0 0 1
0 0 1 0 1
0 0 0 0 0

6/12/2011

Jagadeesh Vasudevamurthy

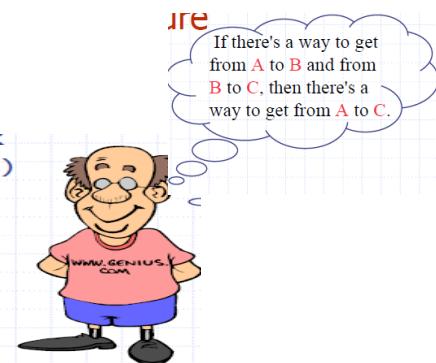
Warshall Algorithm

- Determines whether there is a path from any vertex vi to any another vertex vj either directly or through some intermediate vertices.
- Does not take weights into account.
- Detects presence of cycle. $A[i,i] == 1$ means vertex i has a cycle(s).

6/12/2011

Jagadeesh Vasudevamurthy

Idea



IDEA

If there's a way to get from A to B and from B to C, then there's a way to get from A to C.

6/12/2011

Jagadeesh Vasudevamurthy

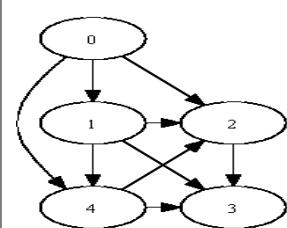
Warshall Algorithm(contd)

- $P[i,j] = P[i,j] \text{ or } (P[i,k] \text{ and } P[k,j])$ – Directed path or path through k
- $P[1,4] = P[1,4] \text{ or } (P[1,2] \text{ and } P[2,4])$
- **For** ($k = 0; k < n; k++$)
- **For** ($i = 0; i < n; i++$)
- **For** ($j = 0; j < n; j++$)
- *BEST : $O(v^3)$ Average $O(v^3)$ Worst $O(v^3)$*

6/12/2011

Jagadeesh Vasudevamurthy

Warshall Algorithm (after 1,2,4,5 ittr)



0 1 1 0 1	0 1 1 1 1
0 0 1 1 1	0 0 1 1 1
0 0 0 1 0	0 0 0 1 0
0 0 0 0 0	0 0 0 0 0
0 0 1 1 0	0 0 1 1 0

0 1 1 0 1	0 1 1 1 1
0 0 1 1 1	0 0 1 1 1
0 0 0 1 0	0 0 0 1 0
0 0 0 0 0	0 0 0 0 0
0 0 1 1 0	0 0 1 1 0

6/12/2011

Jagadeesh Vasudevamurthy

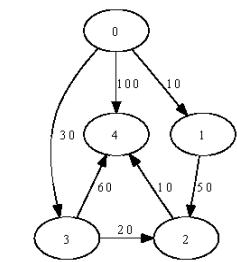
Floyd Algorithm

- Takes weights into account.
- Determines whether there is a path from vertex vi to vj .
- Gives the shortest path from vi to vj and also its path.
- **BEST : $O(v^3)$ Average $O(v^3)$ Worst $O(v^3)$**

6/12/2011

Jagadeesh Vasudevamurthy

Floyd In Action



L 10 L 30 100	L 10 60 30 100
L L 50 L L	L L 50 L L
L L L L 10	L L L L 10
L L 20 L 60	L L 20 L 60
L L L L L	L L L L L
L 10 60 30 70	L 10 50 30 60
L L 50 L 60	L L 50 L 60
L L L L 10	L L L L 10
L L 20 L 30	L L 20 L 30
L L L L L	L L L L L

6/12/2011

Jagadeesh Vasudevamurthy

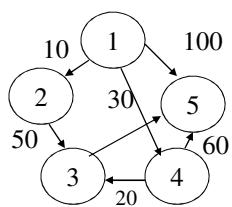
Method

- $\text{for}(i = 0; i < n; i++) {$
- $\text{for}(j = 0; j < n; j++) {$
- $w[i,j] = \text{get_weight}(i,j) ;$
- $p[i,j] = j ;$
- $}$
- $}$
- $P[i,j] = P[i,j] \text{ or } (P[i,k] \text{ and } P[k,j]) - \text{Directed path or path through } k$
- $P[1,4] = P[1,4] \text{ or } (P[1,2] \text{ and } P[2,4])$
- **For (k = 0; k <n; k++)**
- **For (i =0; i < n; i++)**
- **For(j = 0; j < n; j++)**
- – If ($w[i,k]+w[k,j] < w[i,j]$) {
- – $W[i,j] = w[i,k]+w[k,j]$;
- – $p[i,j] = k ;$
- – }

6/12/2011

Jagadeesh Vasudevamurthy

Getting the Path



	1	2	3	4	5
1	1	2	L	4	5
2	L	0	3	L	L
3	L	L	3	L	5
4	L	L	3	4	5
5	L	L	L	L	5

	1	2	3	4	5
1	1	2	4	4	4
2	L	2	3	L	3
3	L	L	3	L	5
4	L	L	3	4	3
5	L	L	L	L	5

6/12/2011 Jagadeesh Vasudevamurthy

$$i,j = i \rightarrow k, k \rightarrow j$$

$$\begin{aligned} 1 \rightarrow 5 &= (1 \rightarrow 4) (4 \rightarrow 5) (3 \rightarrow 5) \\ 1 \rightarrow 4 \rightarrow 3 \rightarrow 5 \end{aligned}$$

Graph Traversal

- Directed Graphs
- Is Directed Graph a DAG ?
- Depth First Search
- Breadth First Search
- Topological Sort

6/12/2011

Jagadeesh Vasudevamurthy

18.3 Graph data structure

18.3.1 Graph representation using adjacency list

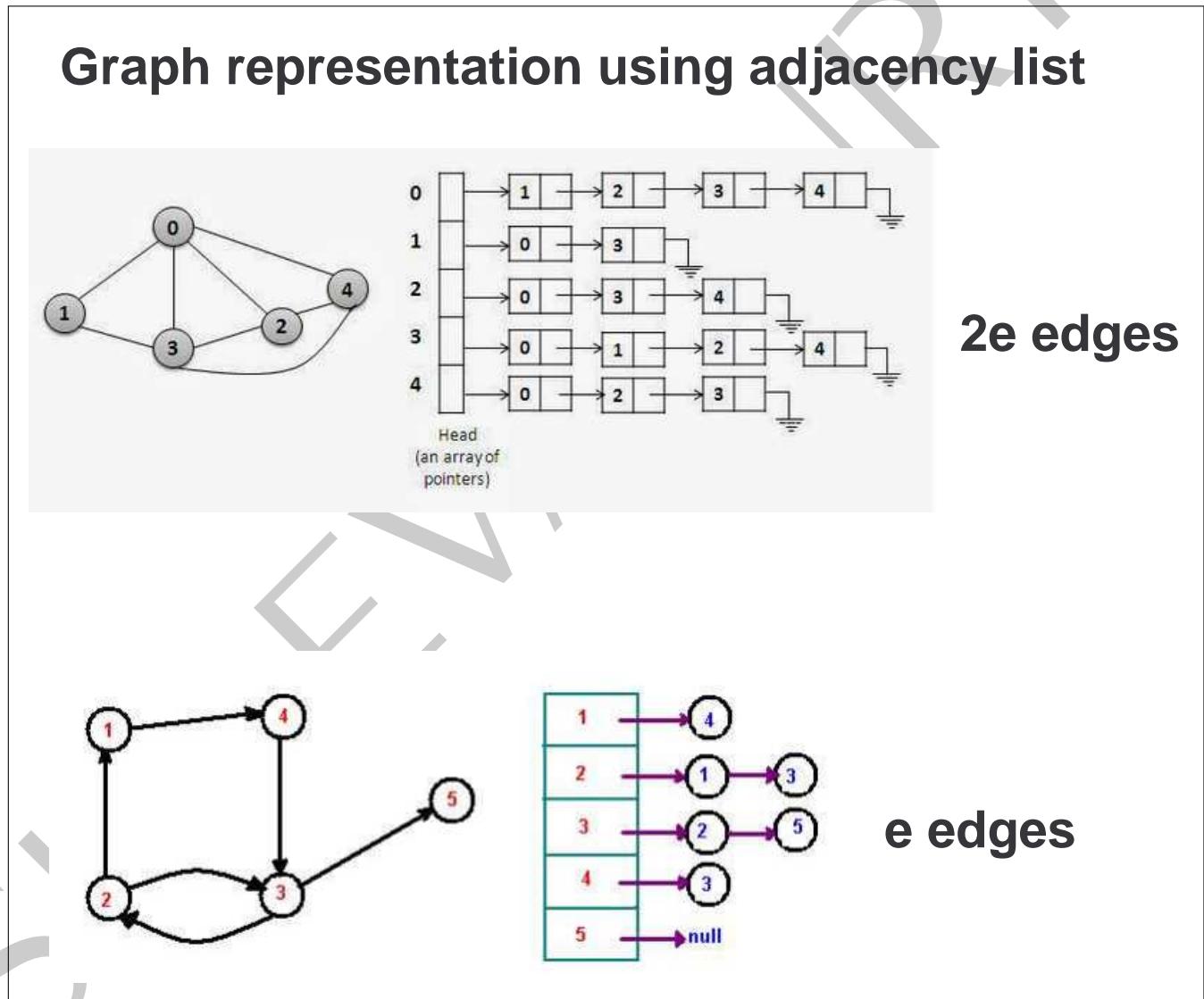


Figure 18.8: Graph representation using adjacency list

18.3.2 Graph representation using fanin and fanout list

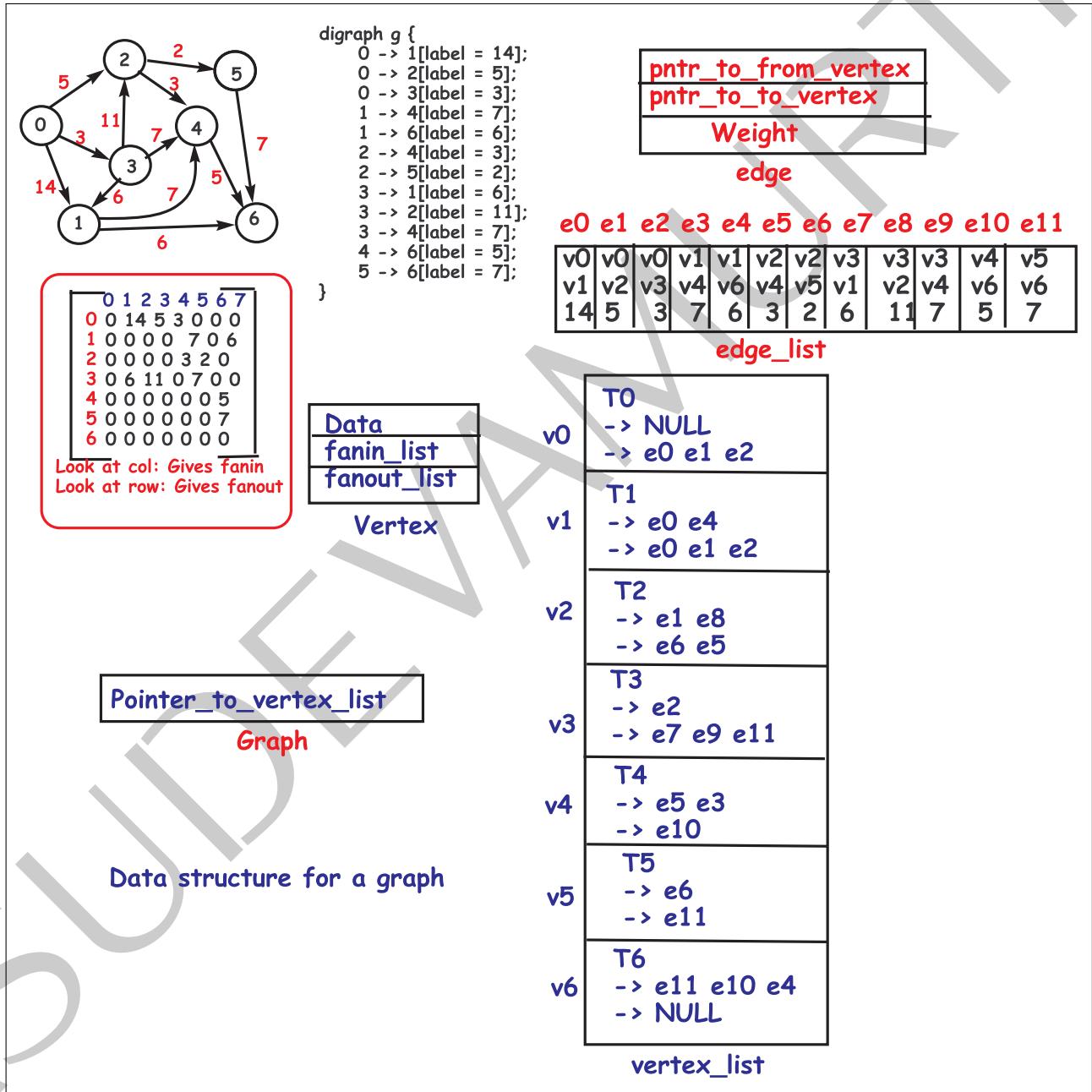


Figure 18.9: Graph data structure

18.4 Graph traversals

18.4.1 Depth first search

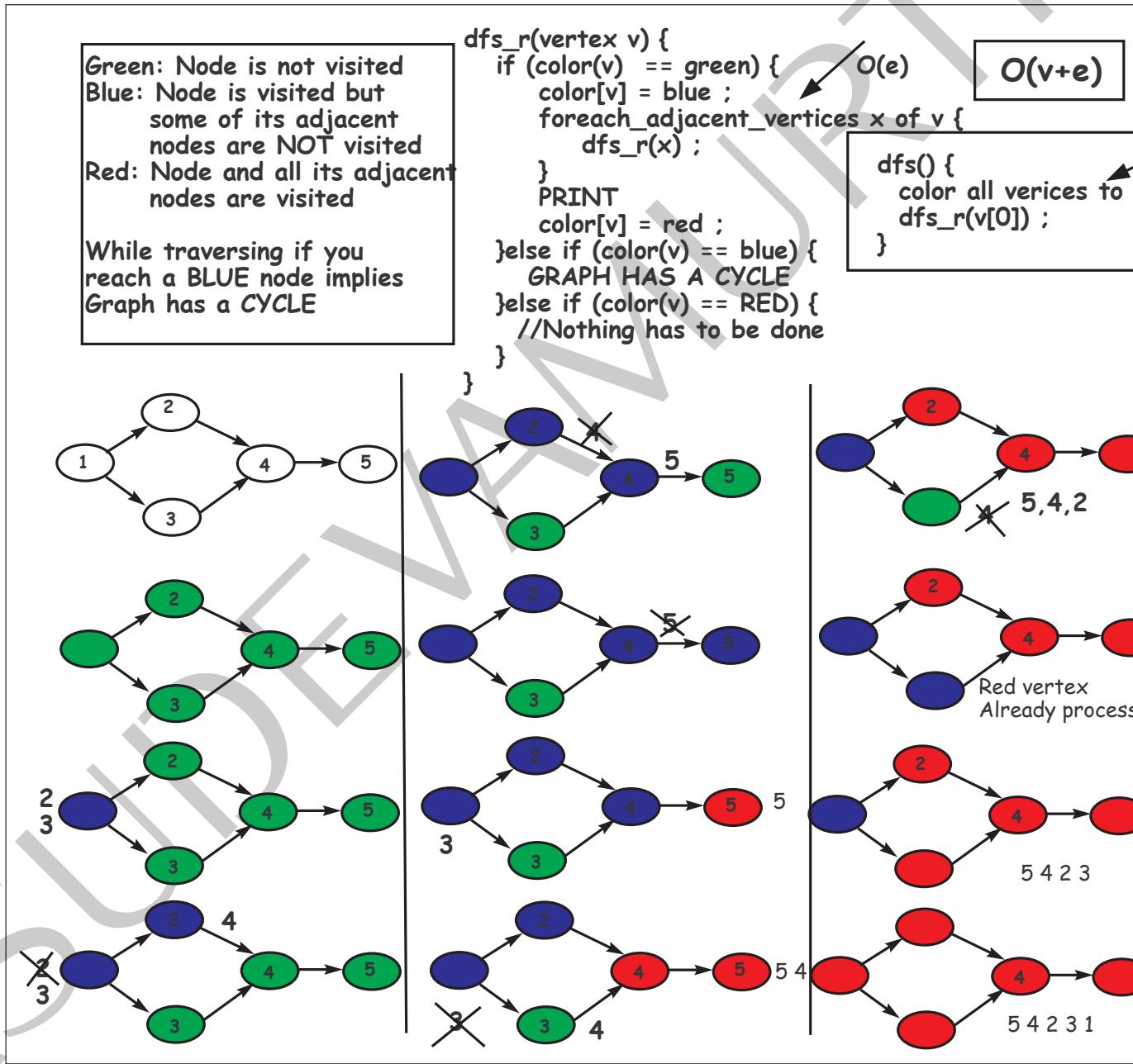


Figure 18.10: Depth first search on a graph that has no loop

18.4. GRAPH TRAVERSALS

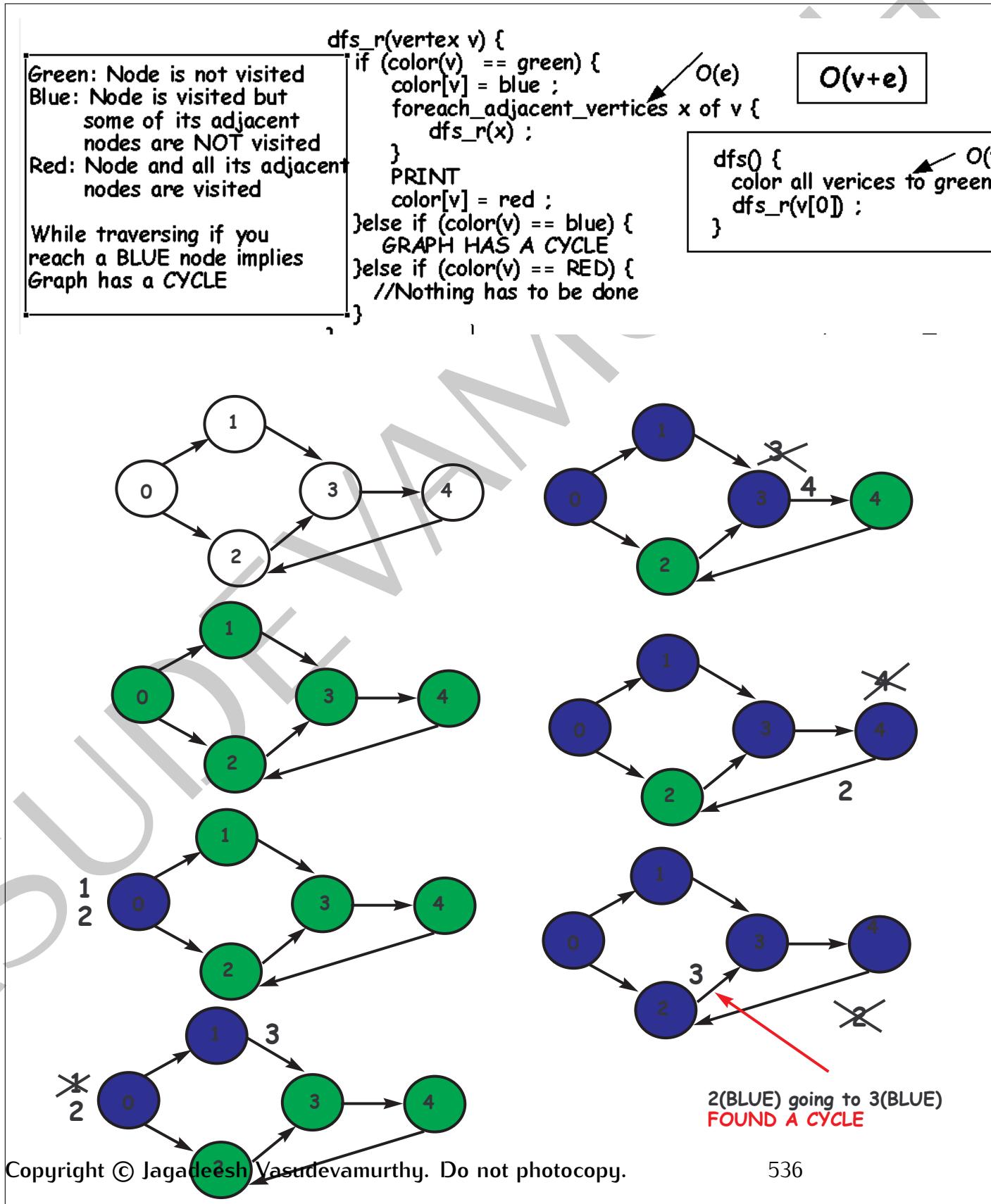


Figure 18.11: Depth first search on a graph that has a loop

18.4.2 Ordering of vertices through DFS

18.4. GRAPH TRAVERSALS

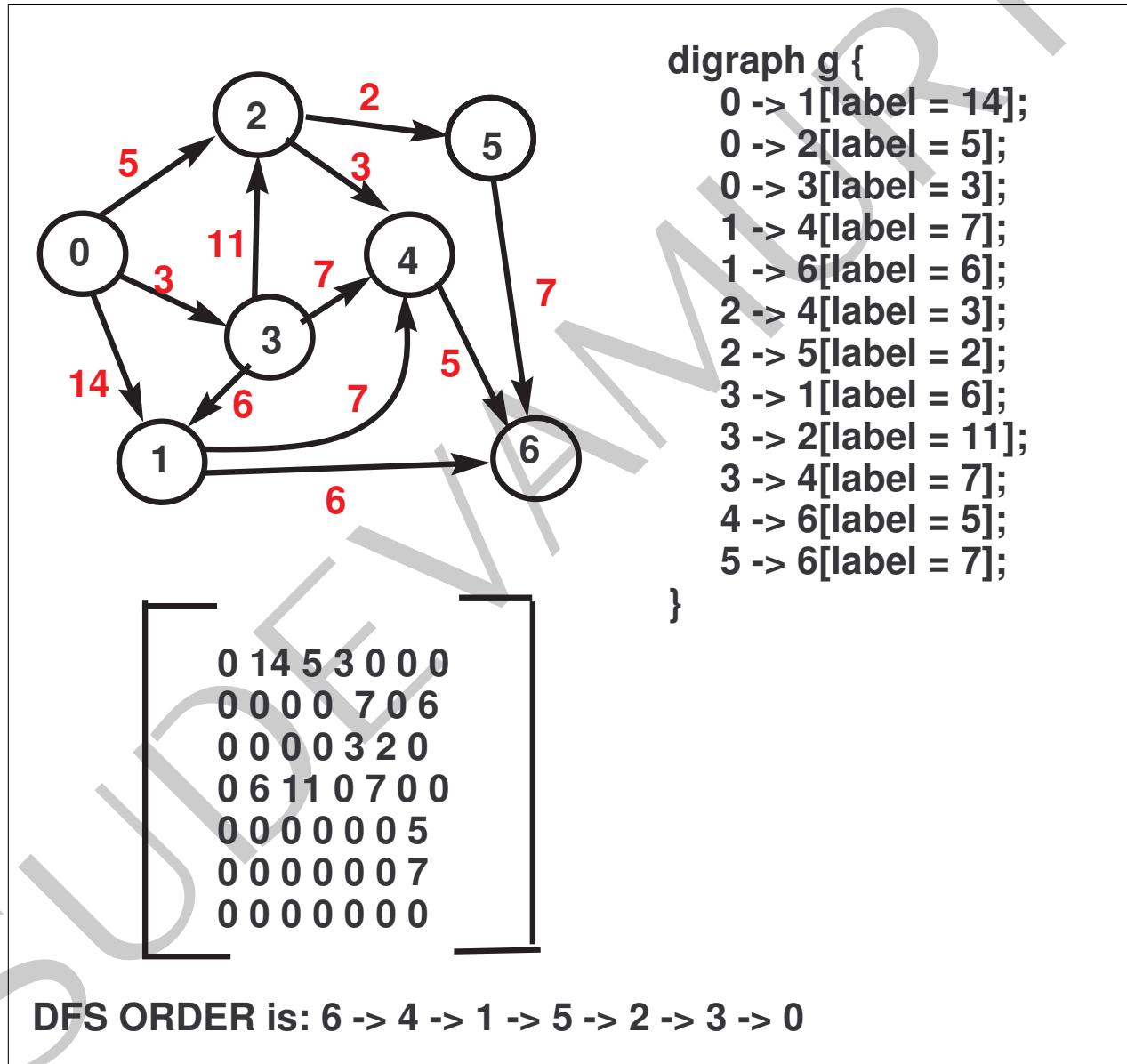


Figure 18.12: Dfs

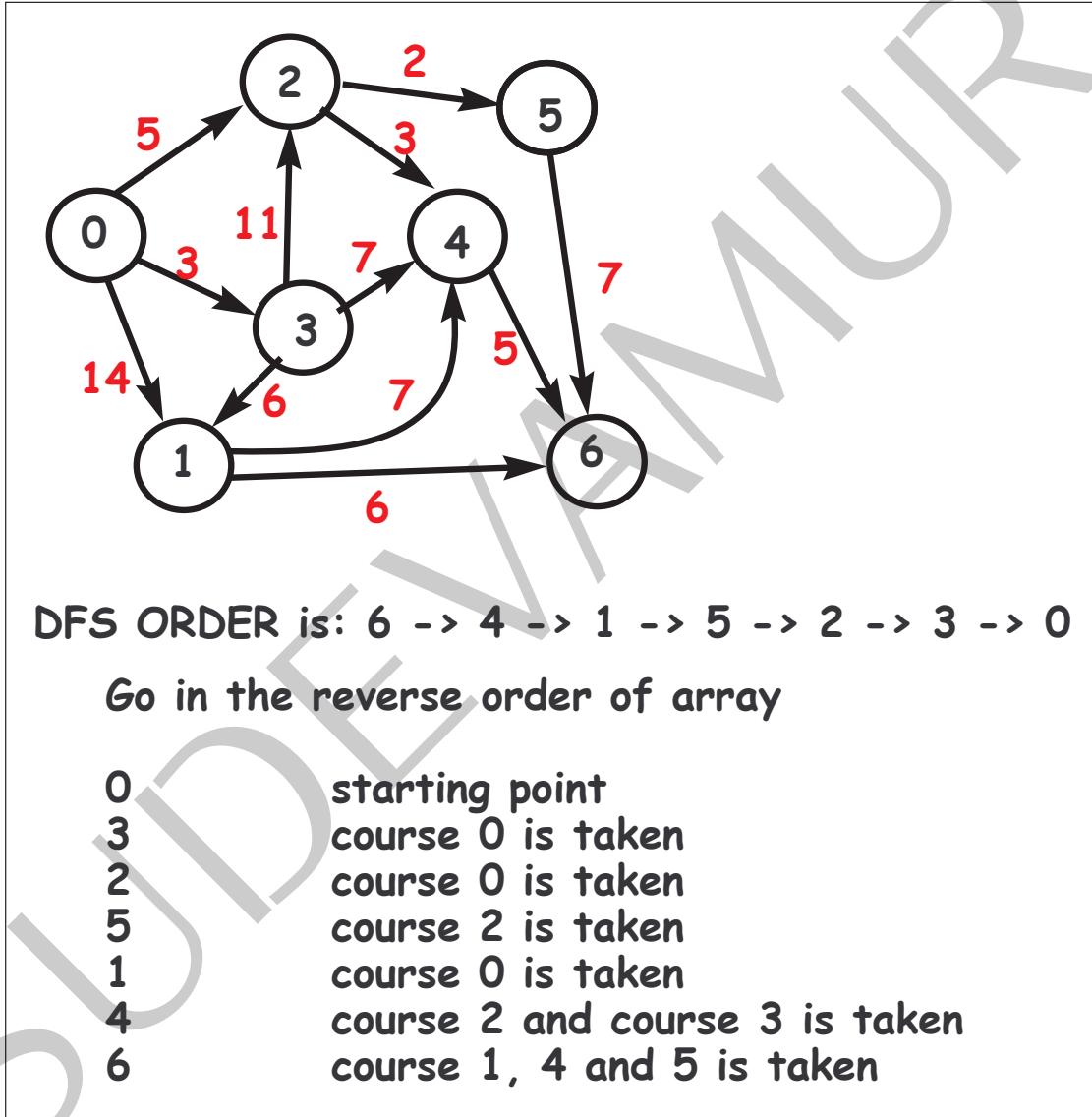


Figure 18.13: Ordering of vertices

18.4.3 Breadth first search

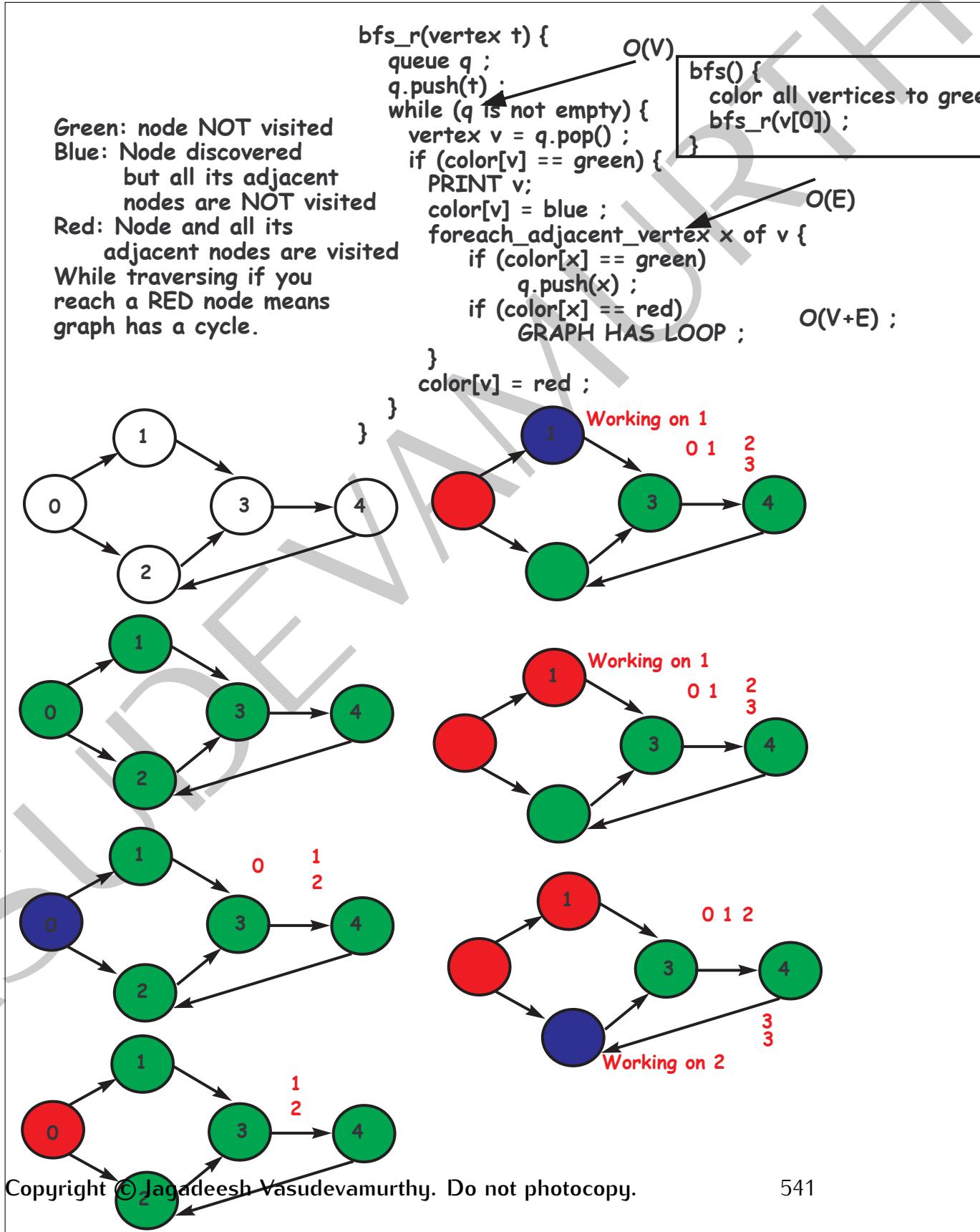


Figure 18.14: Breadth first search on a graph that has a loop

18.4. GRAPH TRAVERSALS

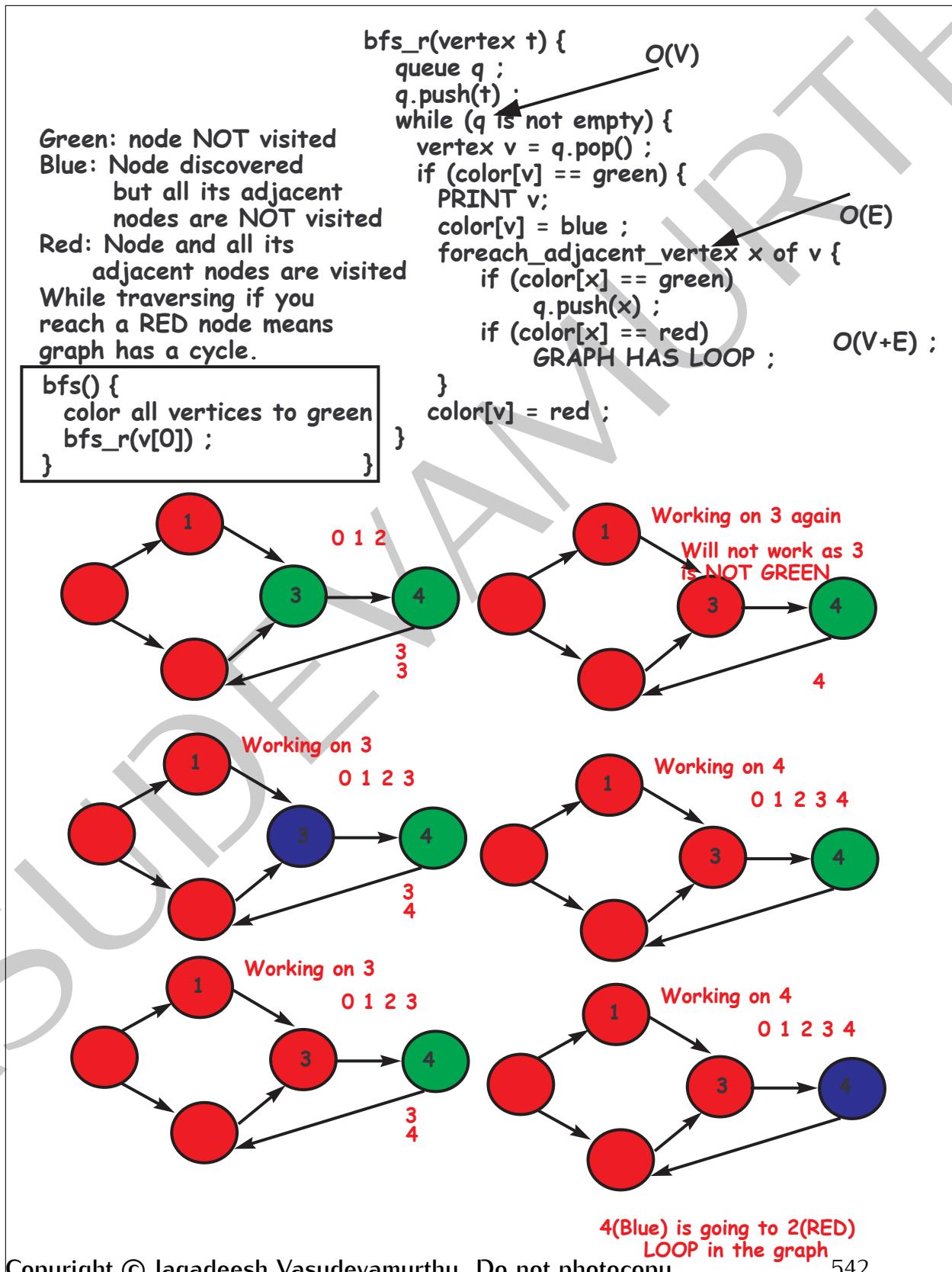


Figure 18.15: Breadth first search on a graph that has a loop(contd)

18.5 Topological sort

18.5. TOPOLOGICAL SORT

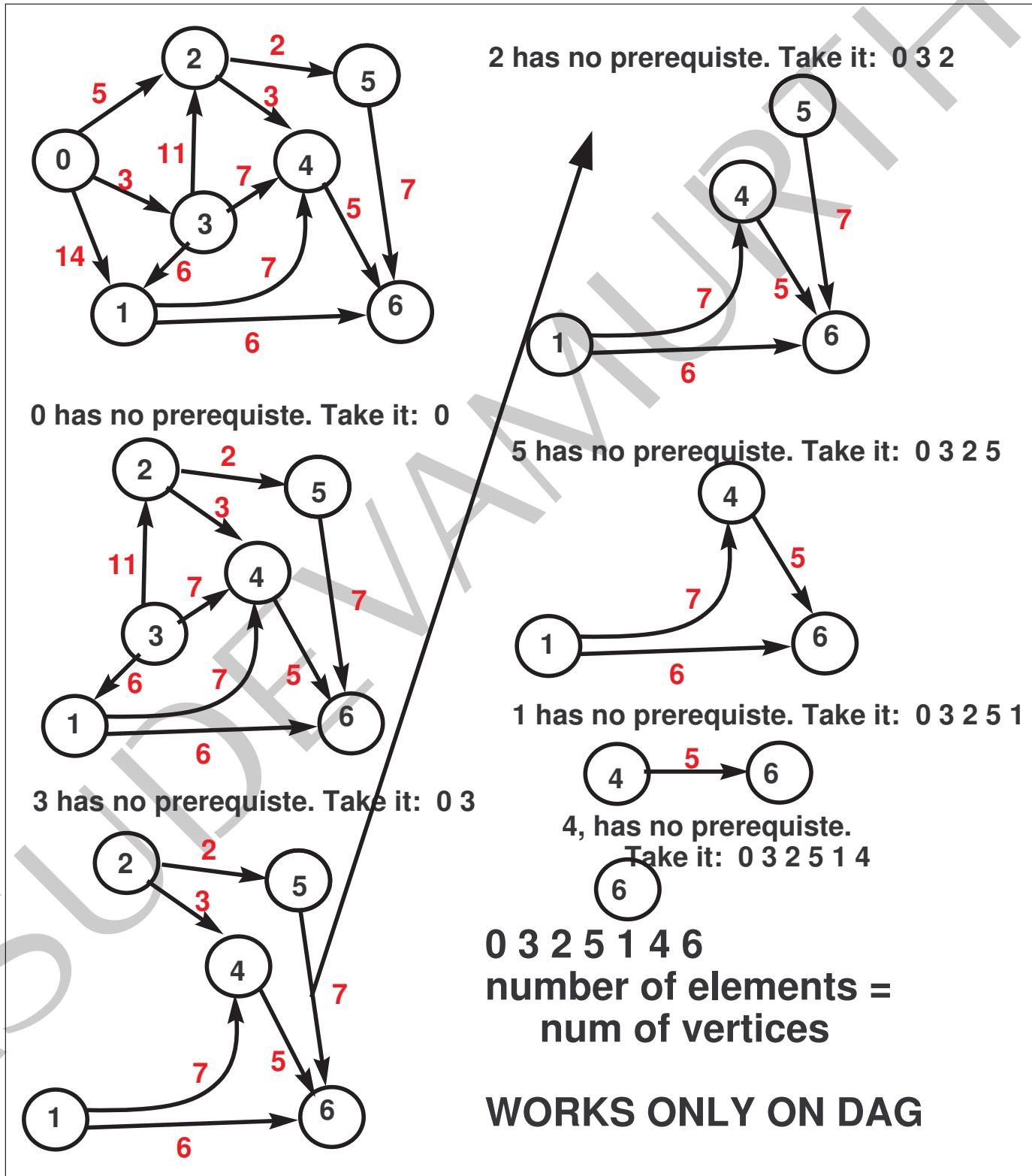


Figure 18.16: Topological sort on a DAG

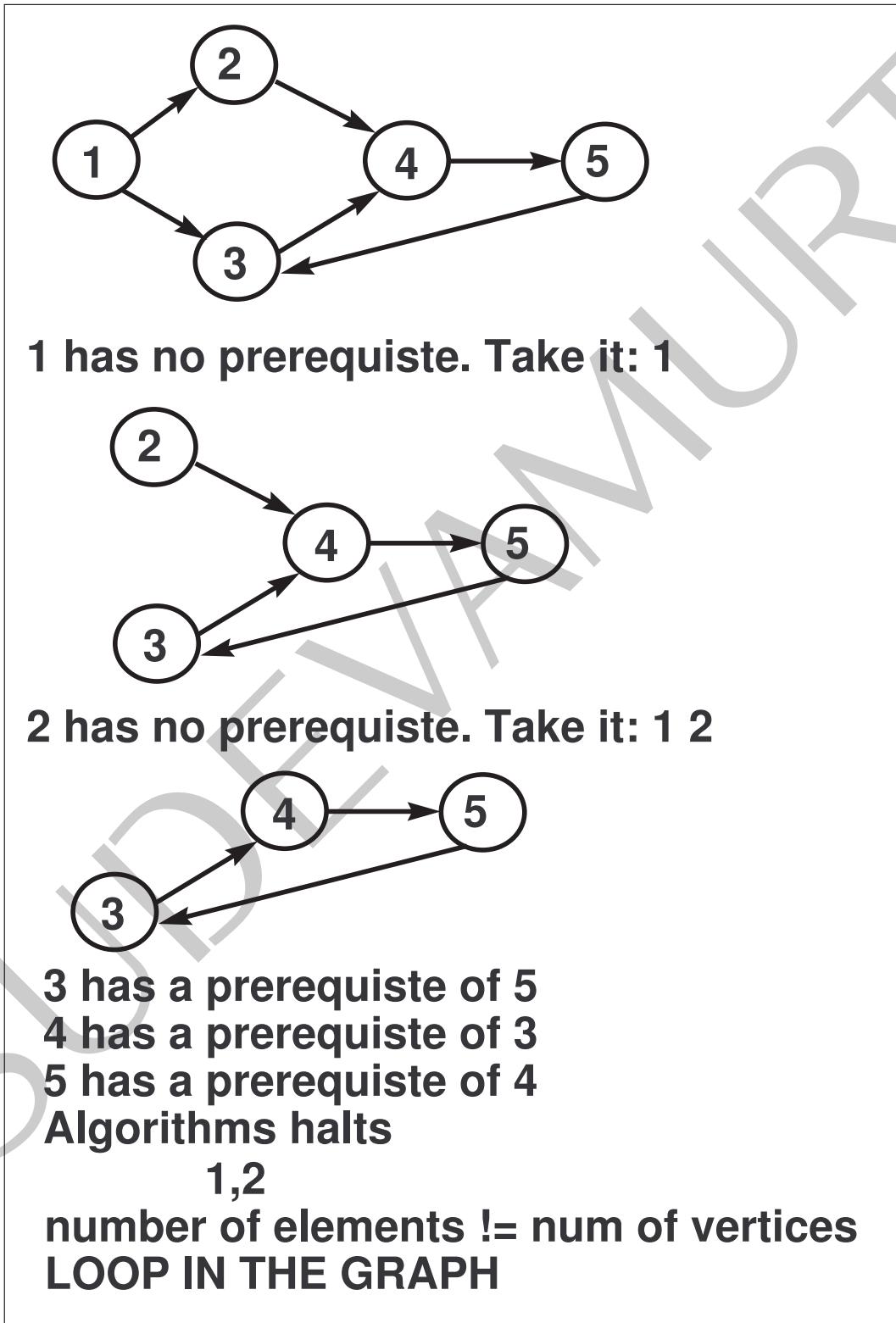


Figure 18.17: Topological sort on a graph that has cycle

18.5. TOPOLOGICAL SORT

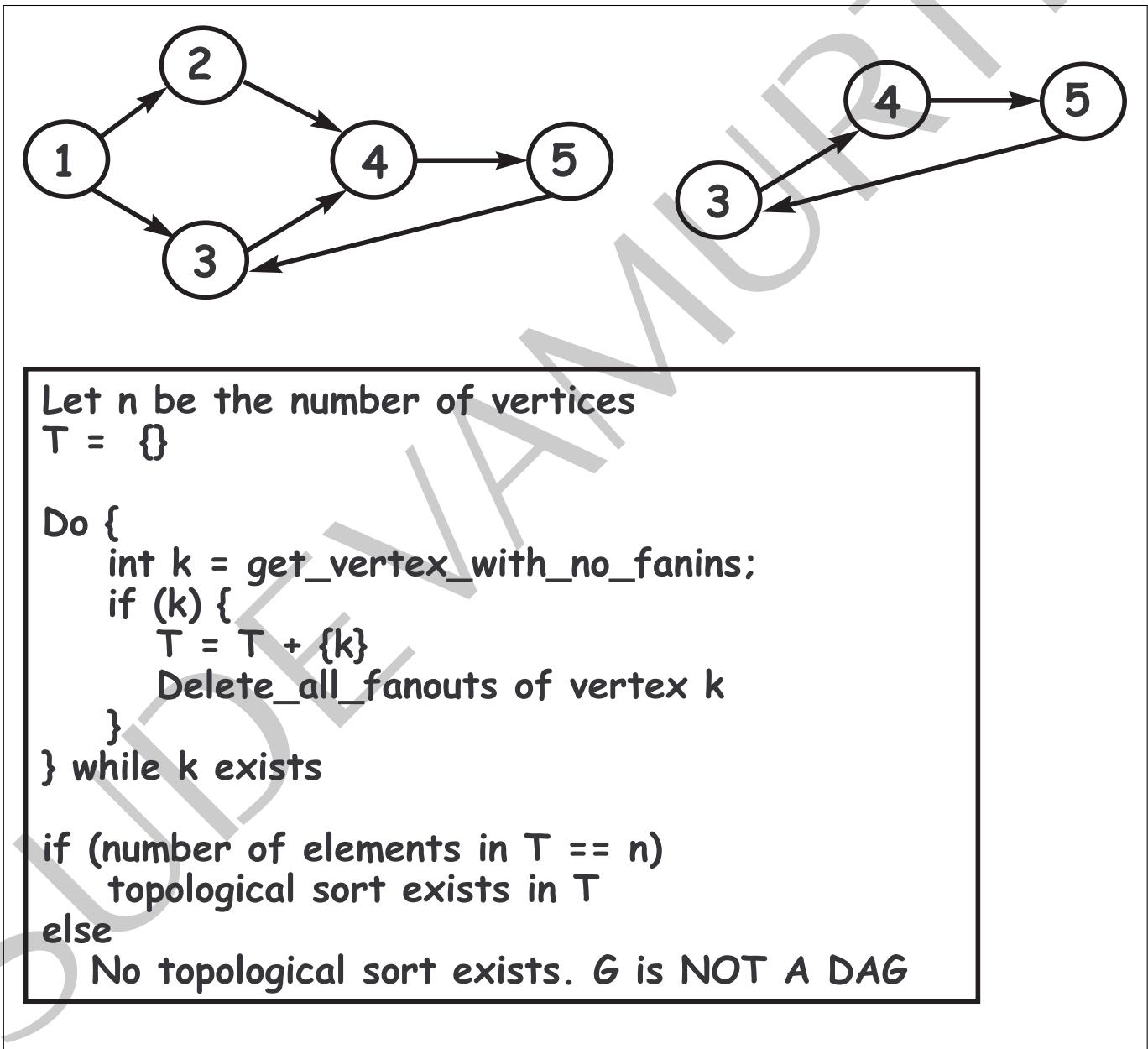
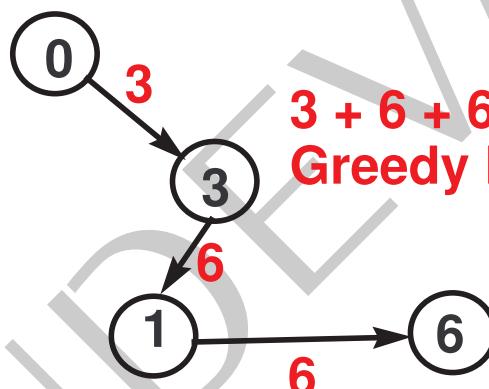
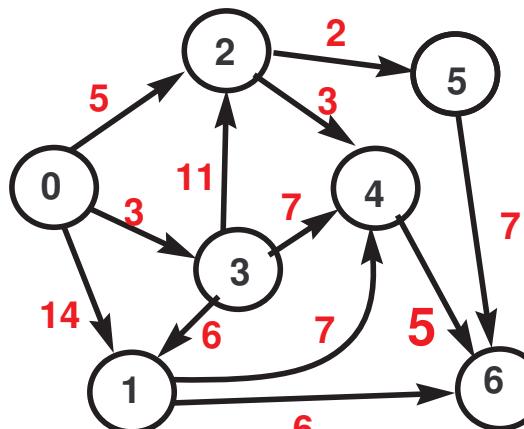


Figure 18.18: Topological sort algorithm

18.6 Dynamic programming

Find the minimum distance between 0 and 6



Greedy DOES NOT work

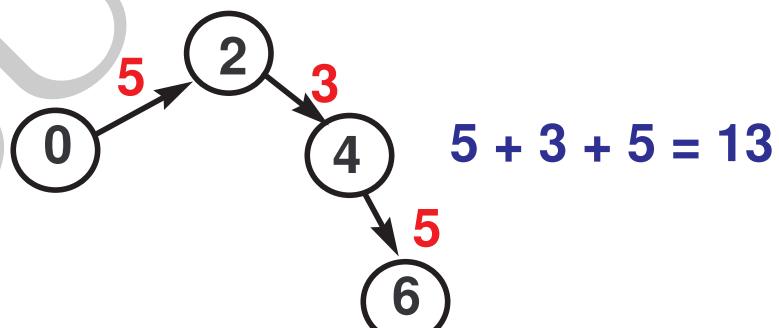


Figure 18.19: Greedy algorithm that does not work

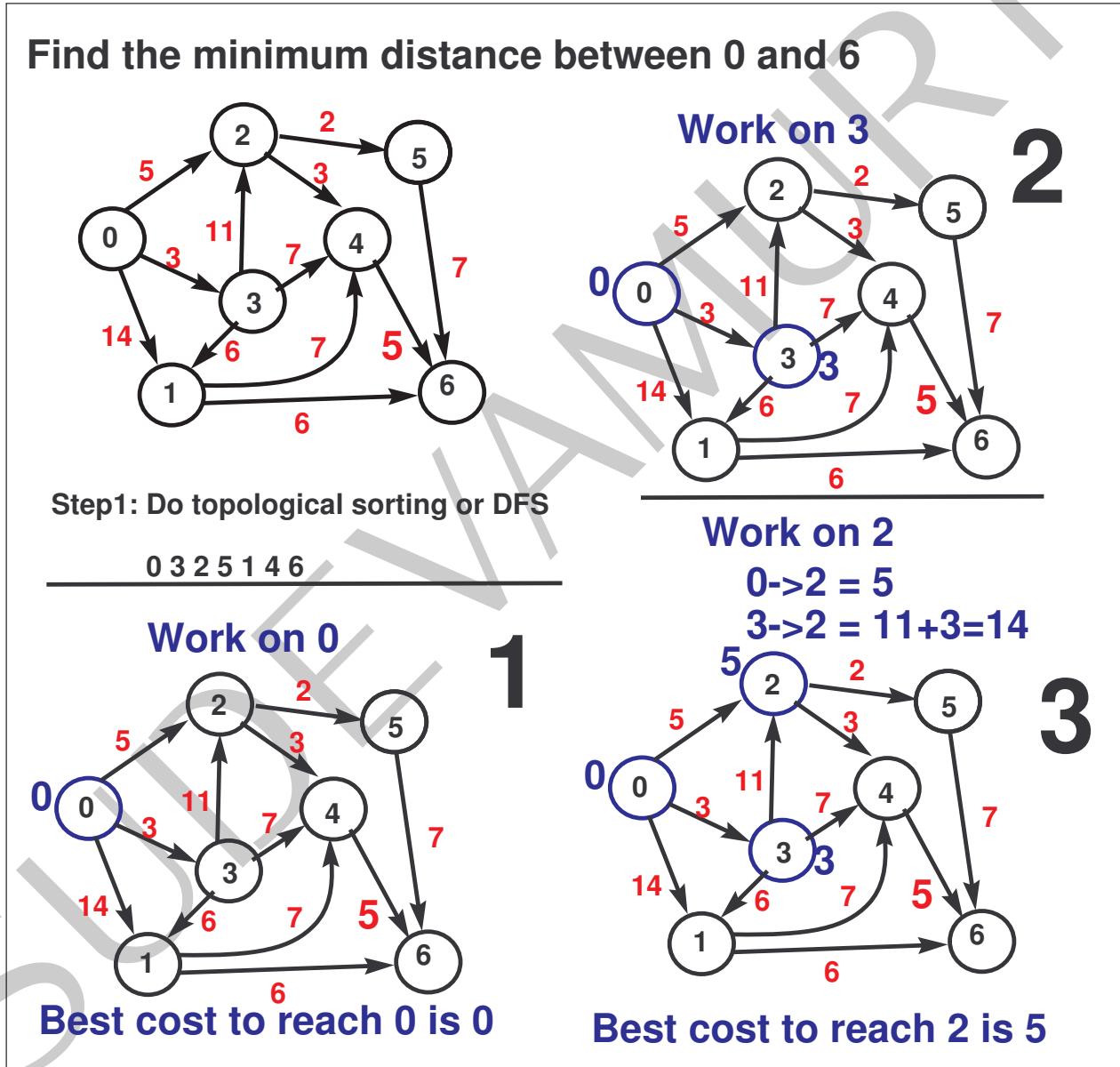


Figure 18.20: Dynamic programming

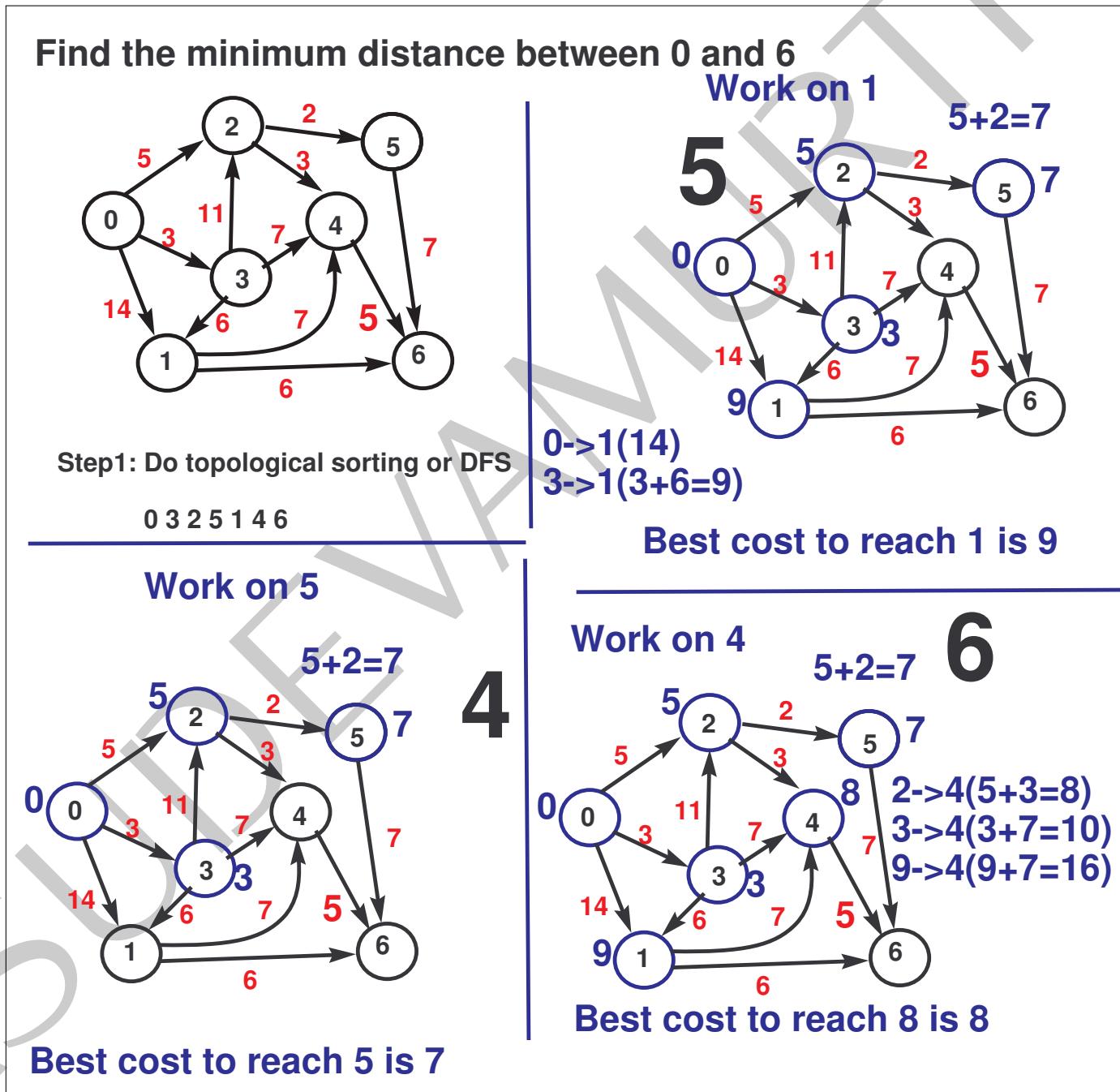
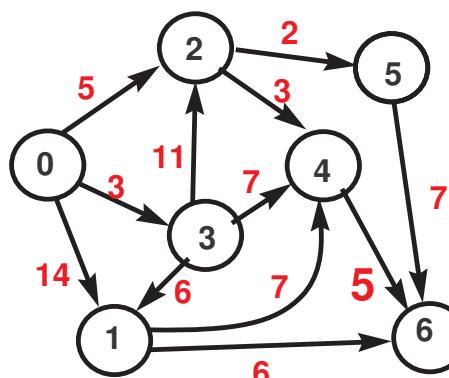


Figure 18.21: Dynamic programming (continued)

Find the minimum distance between 0 and 6



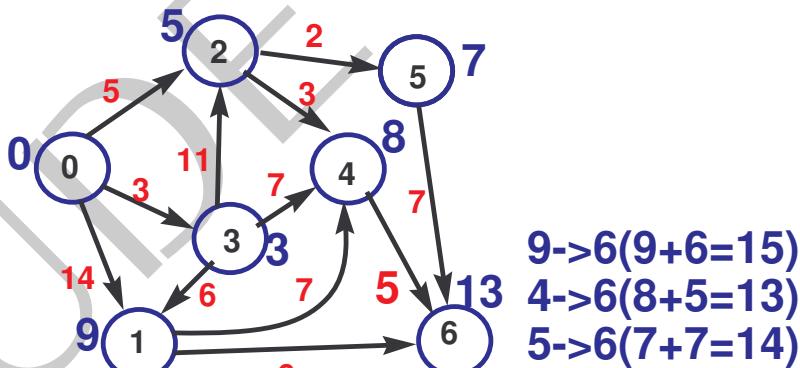
Step1: Do topological sorting or DFS

0 3 2 5 1 4 6

Work on 6

$$5+2=7$$

7



Best cost to reach 6 is 13

At this point, we also know how to reach any node with minimum cost

Figure 18.22: Dynamic programming (continued)

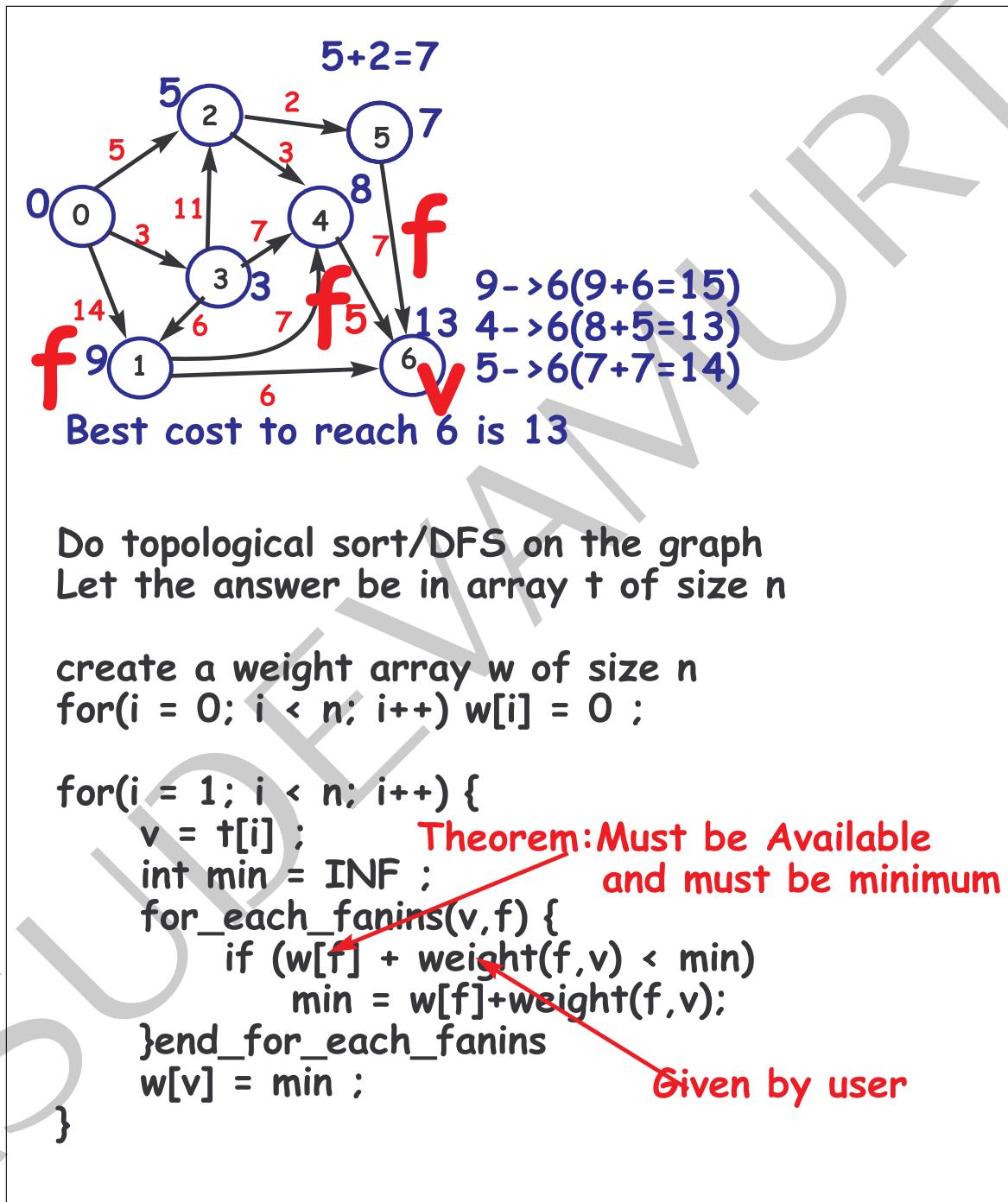


Figure 18.23: Dynamic programming algorithm

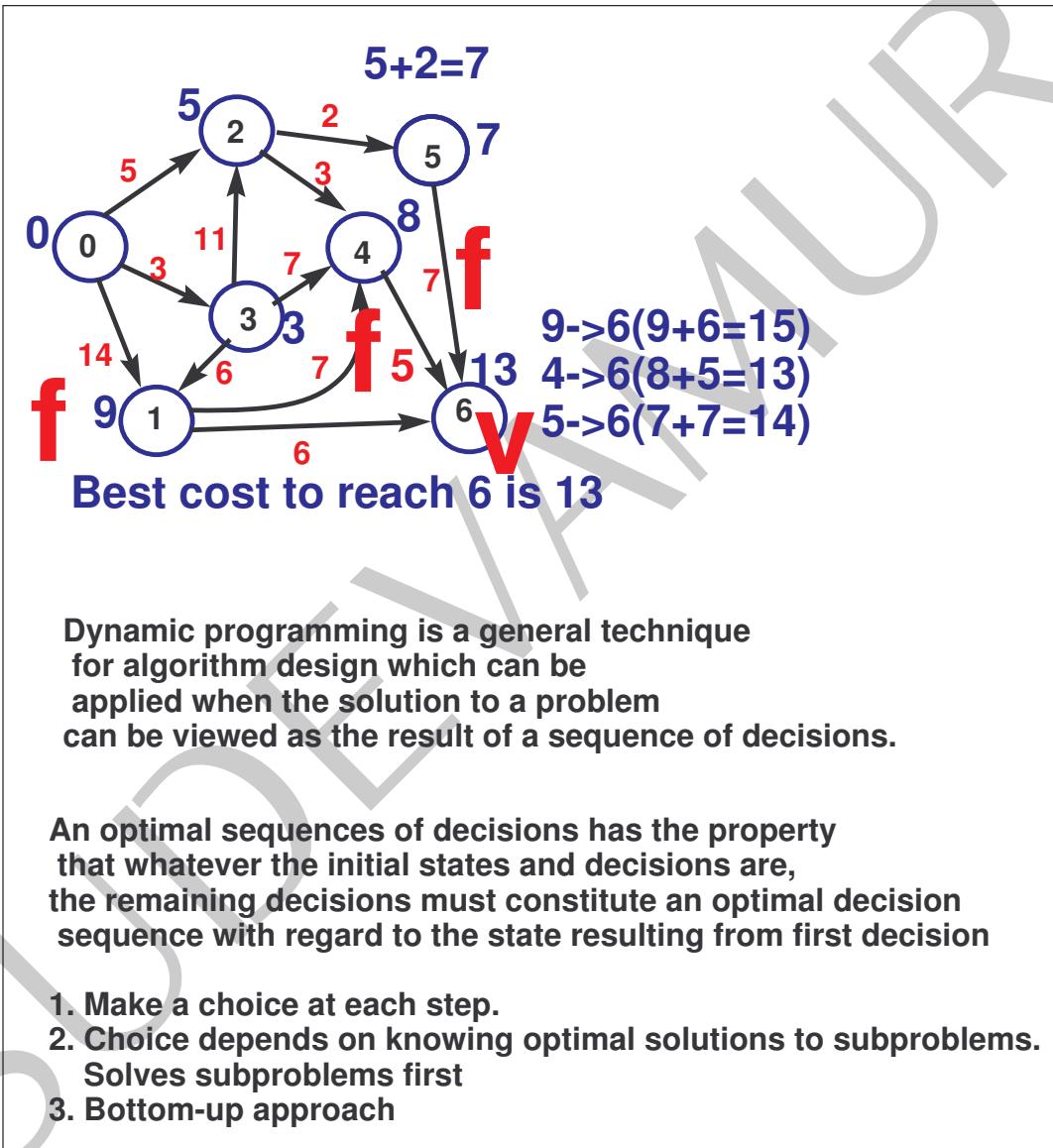


Figure 18.24: Principle of optimality

18.7 Dijkstra's algorithm

A Note on Two Problems in Connexion with Graphs

By

E. W. DIJKSTRA

We consider n points (nodes), some or all pairs of which are connected by a branch; the length of each branch is given. We restrict ourselves to the case where at least one path exists between any two nodes. We now consider two problems.

Problem 1. Construct the tree of minimum total length between the n nodes. (A tree is a graph with one and only one path between every two nodes.)

In the course of the construction that we present here, the branches are subdivided into three sets:

I. the branches definitely assigned to the tree under construction (they will form a subtree);

II. the branches from which the next branch to be added to set I, will be selected;

III. the remaining branches (rejected or not yet considered).

The nodes are subdivided into two sets:

A. the nodes connected by the branches of set I,

B. the remaining nodes (one and only one branch of set II will lead to each of these nodes).

We start the construction by choosing an arbitrary node as the only member of set A, and by placing all branches that end in this node in set II. To start with, set I is empty. From then onwards we perform the following two steps repeatedly.

Step 1. The shortest branch of set II is removed from this set and added to set I. As a result one node is transferred from set B to set A.

Step 2. Consider the branches leading from the node, that has just been transferred to set A, to the nodes that are still in set B. If the branch under consideration is longer than the corresponding branch in set II, it is rejected; if it is shorter, it replaces the corresponding branch in set II, and the latter is rejected.

We then return to step 1 and repeat the process until sets II and B are empty. The branches in set I form the tree required.

The solution given here is to be preferred to the solution given by J. B. KRUSKAL [1] and those given by H. LOBERMAN and A. WEINBERGER [2]. In their solutions all the — possibly $\frac{1}{2}n(n-1)$ — branches are first of all sorted according to length. Even if the length of the branches is a computable function of the node coordinates, their methods demand that data for all branches are stored simultaneously. Our method only requires the simultaneous storing of

the data for at most n branches, viz. the branches in sets I and II and the branch under consideration in step 2.

Problem 2. Find the path of minimum total length between two given nodes P and Q .

We use the fact that, if R is a node on the minimal path from P to Q , knowledge of the latter implies the knowledge of the minimal path from P to R . In the solution presented, the minimal paths from P to the other nodes are constructed in order of increasing length until Q is reached.

In the course of the solution the nodes are subdivided into three sets:

A. the nodes for which the path of minimum length from P is known; nodes will be added to this set in order of increasing minimum path length from node P ;

B. the nodes from which the next node to be added to set A will be selected; this set comprises all those nodes that are connected to at least one node of set A but do not yet belong to A themselves;

C. the remaining nodes.

The branches are also subdivided into three sets:

I. the branches occurring in the minimal paths from node P to the nodes in set A;

II. the branches from which the next branch to be placed in set I will be selected; one and only one branch of this set will lead to each node in set B;

III. the remaining branches (rejected or not yet considered).

To start with, all nodes are in set C and all branches are in set III. We now transfer node P to set A and from then onwards repeatedly perform the following steps.

Step 1. Consider all branches r connecting the node just transferred to set A with nodes R in sets B or C. If node R belongs to set B, we investigate whether the use of branch r gives rise to a shorter path from P to R than the known path that uses the corresponding branch in set II. If this is not so, branch r is rejected; if, however, use of branch r results in a shorter connexion between P and R than hitherto obtained, it replaces the corresponding branch in set II and the latter is rejected. If the node R belongs to set C, it is added to set B and branch r is added to set II.

Step 2. Every node in set B can be connected to node P in only one way if we restrict ourselves to branches from set I and one from set II. In this sense each node in set B has a distance from node P : the node with minimum distance from P is transferred from set B to set A, and the corresponding branch is transferred from set II to set I. We then return to step 1 and repeat the process until node Q is transferred to set A. Then the solution has been found.

Remark 1. The above process can also be applied in the case where the length of a branch depends on the direction in which it is traversed.

Remark 2. For each branch in sets I and II it is advisable to record its two nodes (in order of increasing distance from P), and the distance between P and that node of the branch that is furthest from P . For the branches of set I this

is the actual minimum distance, for the branches of set II it is only the minimum thus far obtained.

The solution given above is to be preferred to the solution by L. R. FORD [3] as described by C. BERGE [4], for, irrespective of the number of branches, we need not store the data for all branches simultaneously but only those for the branches in sets I and II, and this number is always less than n . Furthermore, the amount of work to be done seems to be considerably less.

References

- [1] KRUSKAL jr., J. B.: On the Shortest Spanning Subtree of a Graph and the Travelling Salesman Problem. Proc. Amer. Math. Soc. 7, 48–50 (1956).
- [2] LOBERMAN, H., and A. WEINBERGER: Formal Procedures for Connecting Terminals with a Minimum Total Wire Length. J. Ass. Comp. Mach. 4, 428–437 (1957).
- [3] FORD, L. R.: Network flow theory. Rand Corp. Paper, P-923, 1956.
- [4] BERGE, C.: Théorie des graphes et ses applications, pp. 68–69. Paris: Dunod 1958.

Mathematisch Centrum
2e Boerhaavestraat 49
Amsterdam-O

(Received June 11, 1959)

18.7. DIJKSTRA'S ALGORITHM

Edsger Wybe Dijkstra	
	
Born	May 11, 1930 Rotterdam, Netherlands
Died	August 6, 2002 (aged 72) Nuenen, Netherlands
Fields	Computer science
Institutions	Mathematisch Centrum Eindhoven University of Technology The University of Texas at Austin
Doctoral advisor	Adriaan van Wijngaarden
Doctoral students	Nico Habermann Martin Rem David Naumann Cornelis Hemerik Jan Tijmen Udding Johannes van de Snepscheut Antonetta van Gasteren
Known for	Dijkstra's algorithm Structured programming THE multiprogramming system Semaphore
Notable awards	Turing Award Association for Computing Machinery

Figure 18.25: Dijkstra

Dijkstra's Algorithm

Finds the shortest path from a vertex s to any other vertex w .

Graph G must be directed and all edge weights must be ≥ 0 .

Graph does not need to be a DAG.

Graph G can have cycles

Greedy Algorithm

USES THREE CONCEPTS:

1. BFS
2. Min Heap(Priority Queue) instead of Queue
3. Relaxation

Figure 18.26: Dijkstra's algorithm concepts

18.7.1 Concept of relaxation

18.7. DIJKSTRA'S ALGORITHM

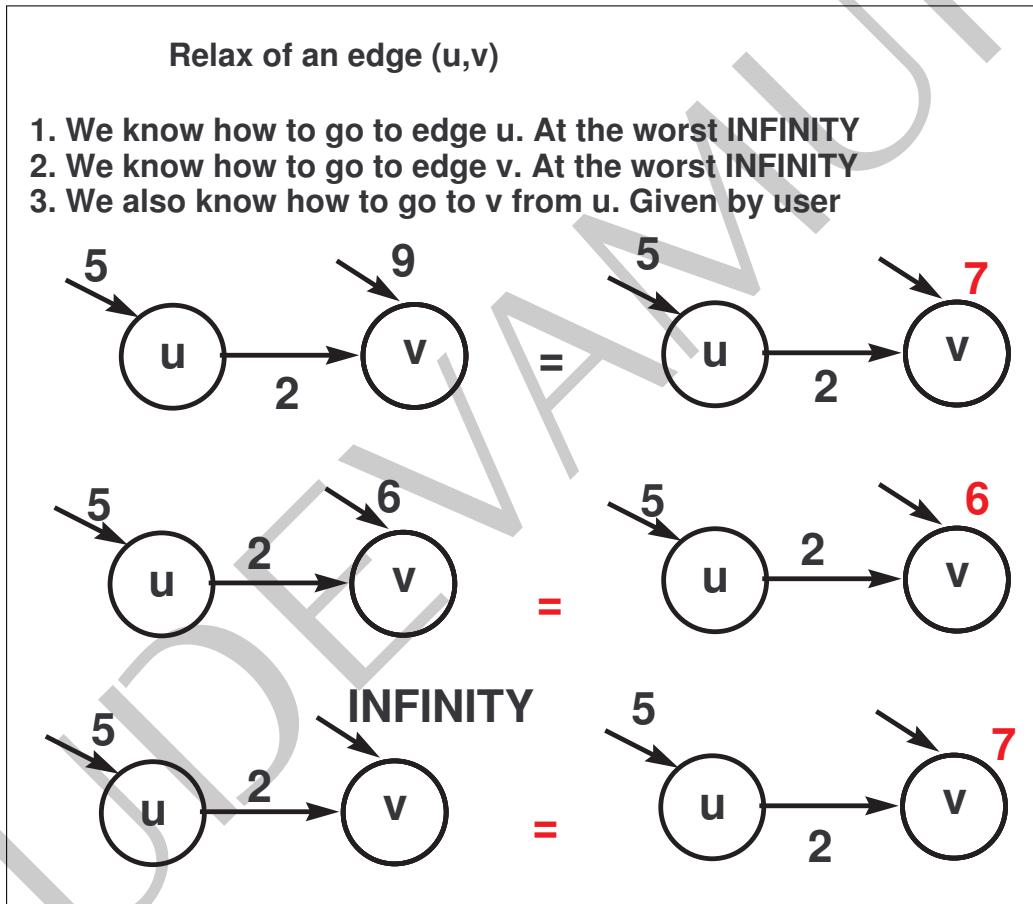
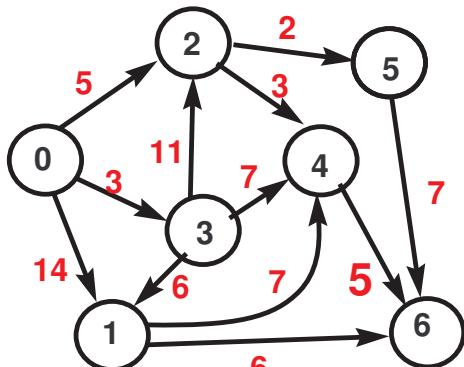
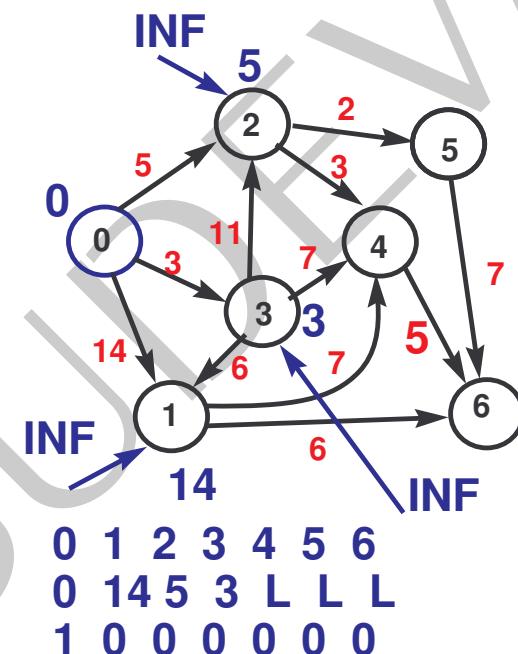


Figure 18.27: Relaxation of an edge

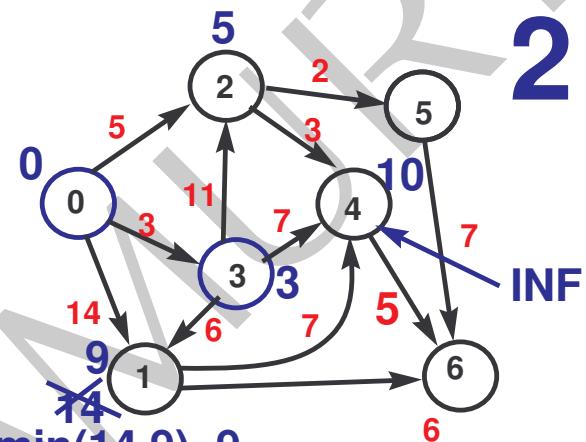
Find the minimum distance between 0 and 6



Work on vertex 0
1

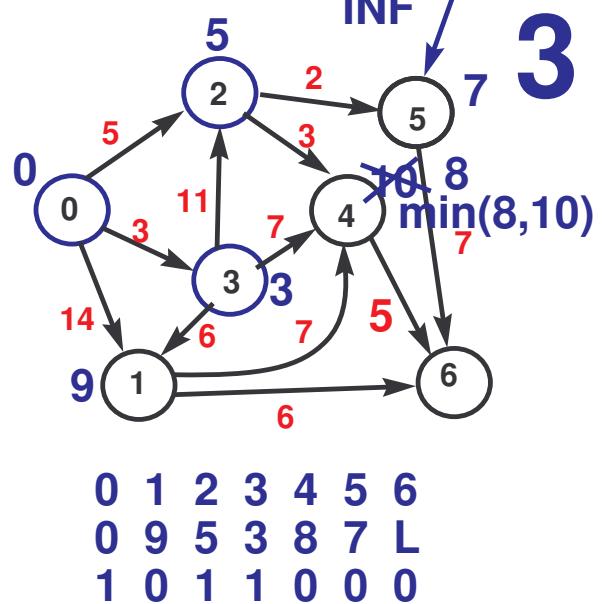


Work on vertex 3 since it has the min value 3
2



$\min(14, 9) = 9$

Work on vertex 2 since it has the min value 5
1



$\min(8, 10) = 8$

Figure 18.28: Dijkstra's algorithm in action

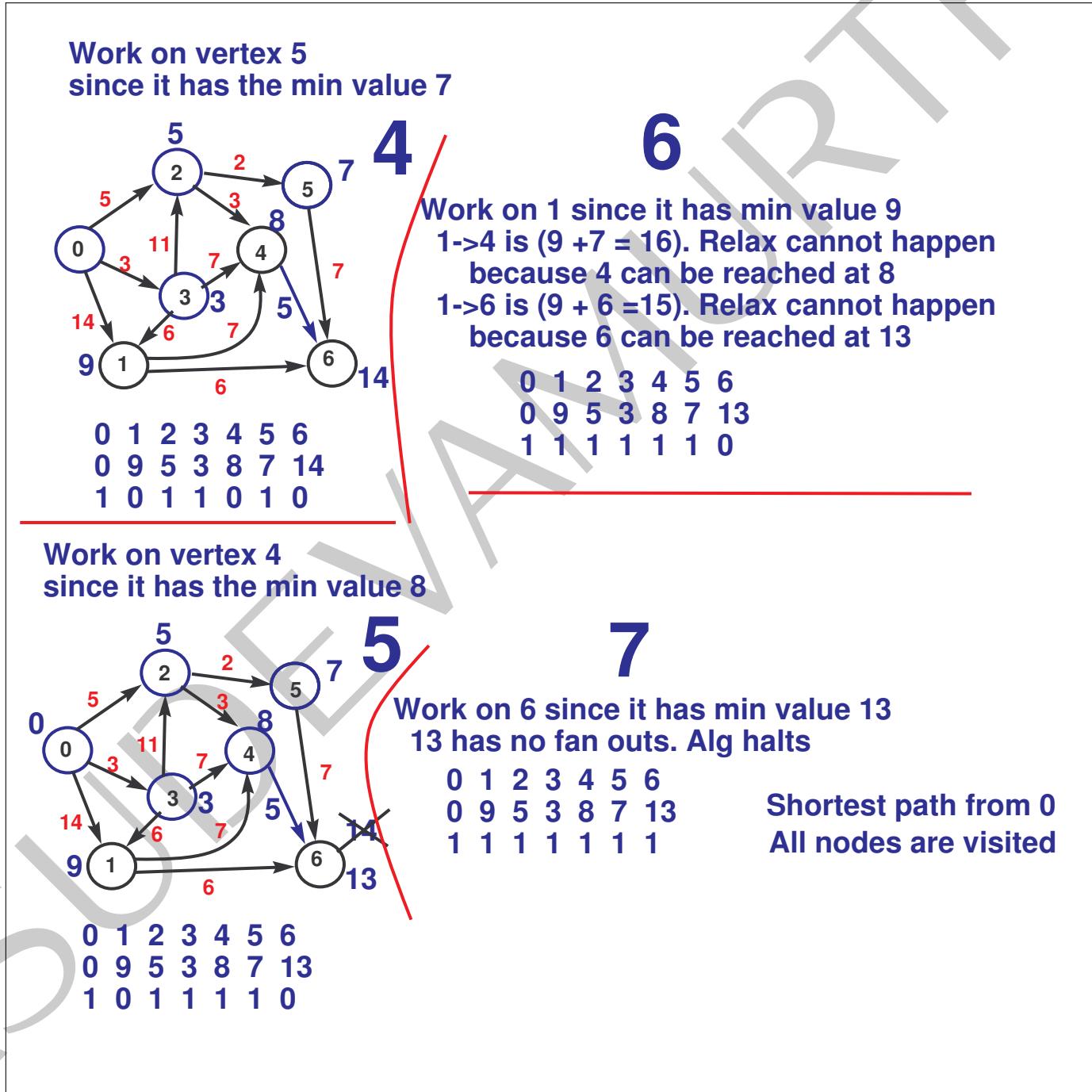


Figure 18.29: Dijkstra's algorithm in action, continued

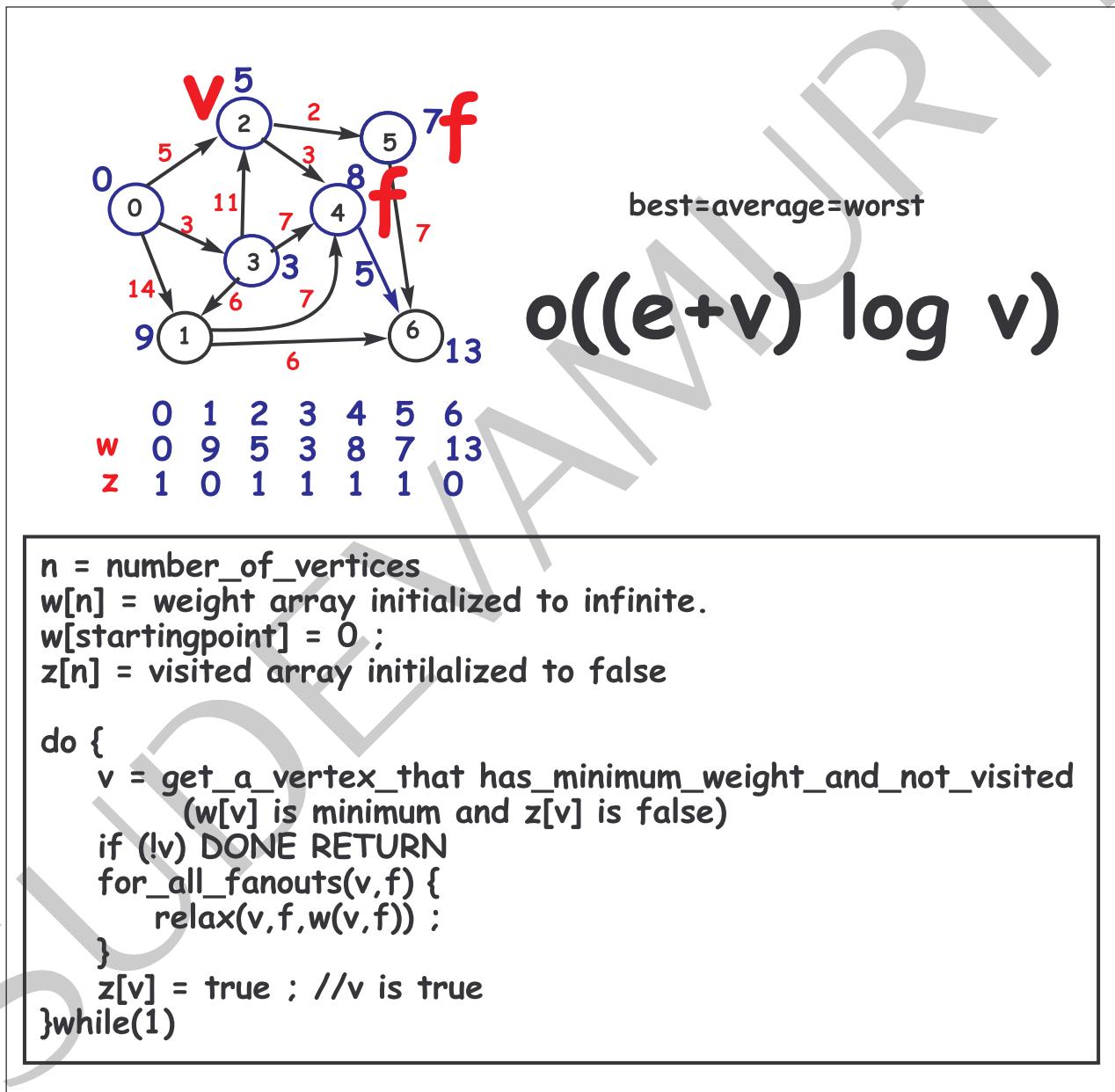


Figure 18.30: Dijkstra's algorithm

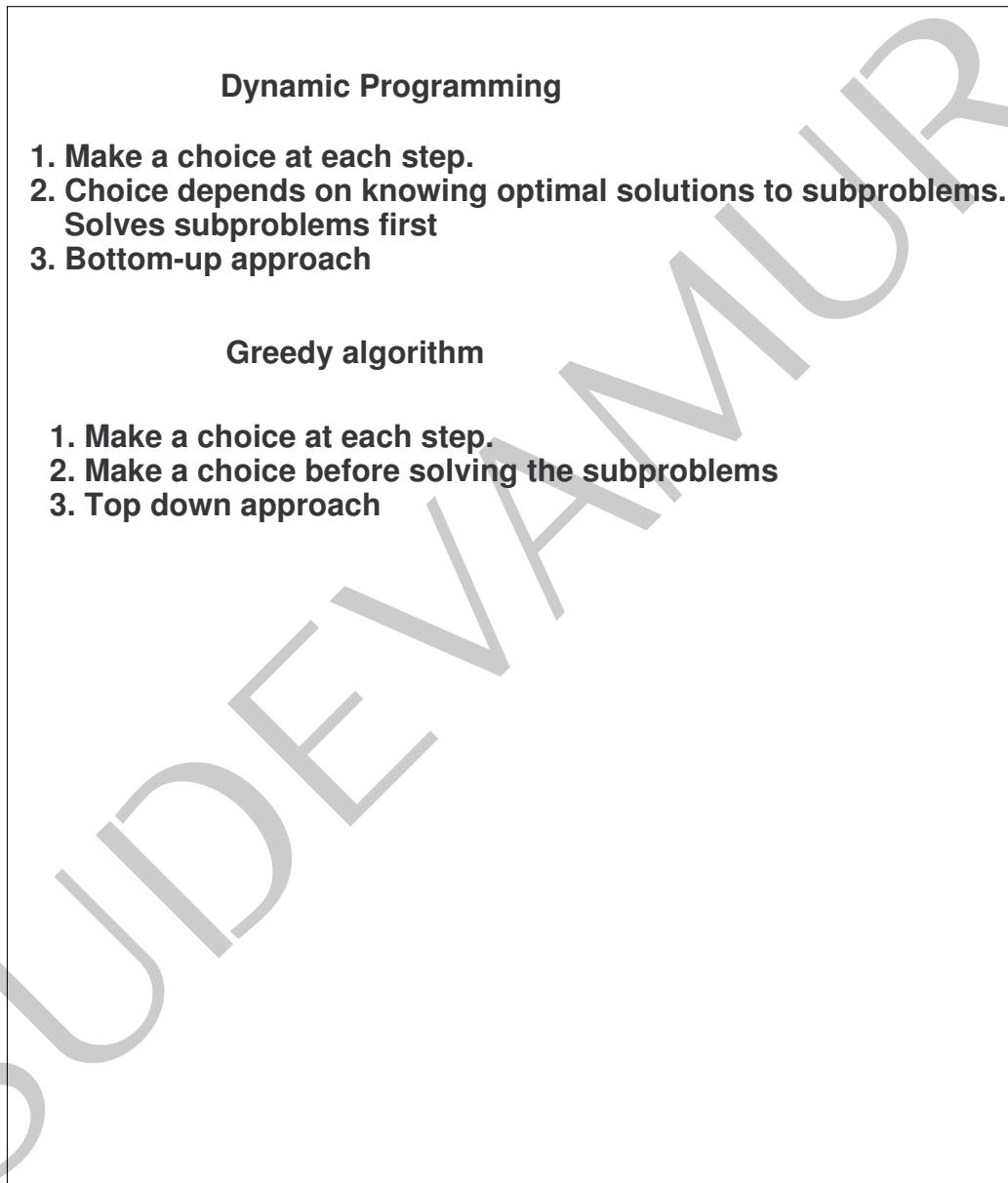
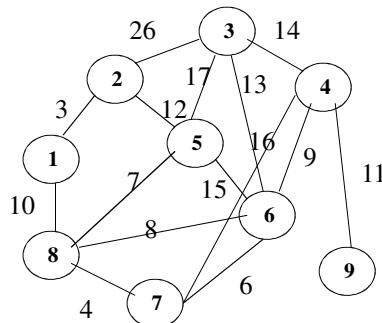


Figure 18.31: Dynamic algorithm versus Greedy algorithm

18.8 Minimal spanning trees

Minimal Spanning Tree

- Laying the cheap road. All vertices should be connected.



6/5/2011

Jagadeesh Vasudevamurthy

Minimal Spanning Tree

- MST is a tree that “spans” the graph
- Every node can be reached
- Minimal in the sense that the sum of the cost along the edges of the graph is minimal
- It must be a tree and cannot contain cycle.

6/5/2011

Jagadeesh Vasudevamurthy

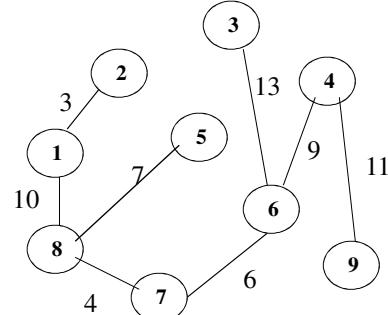
Kruskal Algorithm

1->2 3 Edge added
 7->8 4 Edge added
 6->7 6 Edge added
 5->8 7 Edge added
 6->8 8 Creates loop. Do not this edge
 4->6 9 Edge added
 1->8 10 Edge added
 4->9 11 Edge added
 2->5 12 Creates loop. Do not this edge
 3->6 13 Edge added
 3->4 14 Creates loop. Do not this edge
 5->6 15 Creates loop. Do not this edge
 4->7 16 Creates loop. Do not this edge
 2->3 26 Creates loop. Do not this edge

mst cost 63

6/5/2011

Jagadeesh Vasudevamurthy



Algorithm In action

1->2 3 Edge added
 7->8 4 Edge added
 6->7 6 Edge added
 5->8 7 Edge added
 6->8 8 Creates loop. Do not this edge
 4->6 9 Edge added
 1->8 10 Edge added
 4->9 11 Edge added
 2->5 12 Creates loop. Do not this edge
 3->6 13 Edge added
 3->4 14 Creates loop. Do not this edge
 5->6 15 Creates loop. Do not this edge
 4->7 16 Creates loop. Do not this edge
 2->3 26 Creates loop. Do not this edge

mst cost 63

6/5/2011

	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
1,2	1	1	3	4	5	6	7	8	9
7,8							7	7	
6,7						6	6	6	
5,8					5	5	5	5	
6,8						X		X	
4,6				4	4	4	4	4	
1,8	1	1	3	1	1	1	1	1	9
4,9	1	1	3	1	1	1	1	1	1
2,5		X			X				
3,6	3	3	3	3	3	3	3	3	3

Jagadeesh Vasudevamurthy

Algorithm

- N nodes – $N-1$ edges.
- Arrange edges in increasing order of weights.
- Add an edge e if e does not introduce a loop.
- Greedy Algorithm.
- Undirected Graph.

6/5/2011

Jagadeesh Vasudevamurthy

Prims Algorithm

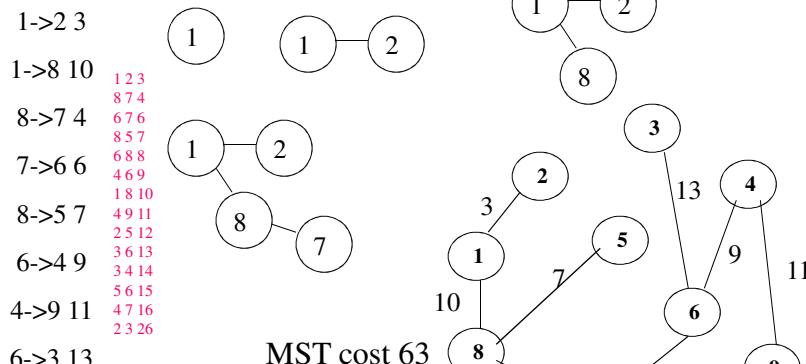
- Does not require to store edges in increasing order of weights.
- Does not require to be checked if a loop is introduced.
- $BEST = AVERAGE=WORST = O((v + e) * \log v)$

6/5/2011

Jagadeesh Vasudevamurthy

Prim's Algorithm in Action

1->2 3
 1->8 10
 8->7 4
 8->5 7
 7->6 6
 7->10 18
 8->5 9
 6->4 9
 4->9 11
 6->3 13



6/5/2011

Jagadeesh Vasudevamurthy

Algorithm

- $T = \{ \}$
- Start with a vertex v_0 . $T=\{v_0\}$
- Look at all the adjacent edges from the vertices of T and find an edge e such that, $e=\{m,n\}$, m in T and n not in T and e is minimum cost. Add n to T .
 $T = \{v_0, n\}$
- If number of vertices in T is $n-1$, mst is found. If e cannot be found, mst cannot be found.

6/5/2011

Jagadeesh Vasudevamurthy

18.9 Code using matrix representation

```
1 /*-
2 Copyright (c) 2010 Author: Jagadeesh VasudevaMurthy
3 Filename: sgraph.cpp
4 compile: g++ sgraph.cpp
5 -----
6
7 #include "util.h"
8
9 */
10 class sgraph
11 -----
12 class sgraph {
13 public:
14     sgraph(int n = 0);
15     ~sgraph() { _n = 0; _release(); }
16     sgraph(const sgraph& x);
17     sgraph& operator=(const sgraph& x);
18
19     static void build_graph(const char *file_name, sgraph& g);
20     friend ostream& operator<<(ostream& o, const sgraph& g);
21     void print_matrix(const char* title = NULL) const;
22
23     int num_vertices() const {return _n; }
24     bool has_a_vertex(int r, int c) const;
25     int weight_of_a_vertex(int r, int c) const {return _matrix[_index(r,c)]; }
26     void insert_a_value_to_a_vertex(int r, int c, int v) {_matrix[_index(r,c)] = v; }
27
28     sgraph transpose() const;
29     friend sgraph product(const sgraph& a, const sgraph& b);
30     void warshall(bool display = false) const;
31     void make_unconnected_path_to_infinity();
32     void floyd(bool display = false) const;
33     void dfs() const;
34
35
36 private:
37     int* _matrix; //Size is n * n
38     int _n; //Number of vertices
39     bool _weighted_graph;
40
41     void _copy(const sgraph& s);
42     void _release(){FREEVEC(_matrix);}
43     int _index(int r, int c) const { return ((r * _n) + c); }
44     void _init_matrix();
45     void _dfs_r(int v, int* c, int* n, int& nindex, bool& cycle) const;
46 };
47
48 */
49 all defines here
50 -----
51 #define INFINITY 999999
52 #define GREEN 1
53 #define BLUE 2
54 #define RED 3
```

```
56 #define foreach_vertex(g,v)    \
57   for (v = 0; v < g.num_vertices(); v++) { \
58 
59 #define end	foreach_vertex } \
60 
61 #define foreach_adjacent_vertex(g,v,e)    \
62   for (e = 0; e < g.num_vertices(); e++) { \ 
63     if (g.has_a_vertex(v,e))  \
64 
65 
66 #define end	foreach_adjacent_vertex } \
67 
68 
69 /*-----*/
70 Constructor \
71 -----*/ \
72 sgraph::sgraph(int n):_n(n),_matrix(NULL),_weighted_graph(false){ \
73   if (n) { \
74     _matrix = ALLOCVEC(int,(_n * _n)); \
75     _init_matrix(); \
76   } \
77 } \
78 
79 /*-----*/
80 _copy \
81 -----*/ \
82 void sgraph::_copy(const sgraph& from){ \
83   _n = from._n; \
84   _weighted_graph = from._weighted_graph; \
85   _matrix = ALLOCVEC(int,(_n * _n)); \
86   for (int r = 0; r < _n; r++) { \
87     for (int c = 0; c < _n; c++) { \
88       int v = from.weight_of_a_vertex(r,c); \
89       insert_a_value_to_a_vertex(r,c,v); \
90     } \
91   } \
92 } \
93 
94 /*-----*/
95 Copy Constructor \
96 -----*/ \
97 sgraph::sgraph(const sgraph& s){ \
98   _copy(s); \
99 } \
100 
101 /*-----*/
102 Equal Constructor \
103 -----*/ \
104 sgraph& sgraph::operator=(const sgraph& rhs){ \
105   if (this != &rhs) { \
106     _release(); \
107     _copy(rhs); \
108   } \
109   return *this; \
110 }
```

```
111
112 /*-
113 m[r,c] = 0 NO vertex
114 m[r,c] >=INFINITY NO vertex
115 */
116 bool sgraph::has_a_vertex(int r, int c) const{
117     int x = weight_of_a_vertex(r,c);
118     return ( (x == 0 || (x >= INFINITY)) ? false : true );
119 }
120
121 /*-
122 */
123 */
124 void sgraph::make_unconnected_path_to_infinity(){
125     for (int r = 0 ; r < _n ; r++) {
126         for (int c = 0; c < _n ; c++) {
127             int v = weight_of_a_vertex(r,c);
128             if (v == 0) {
129                 insert_a_value_to_a_vertex(r,c,INFINITY);
130             }
131         }
132     }
133 }
134
135 /*
136 Initialize the matrix to 0
137 */
138 void sgraph::_init_matrix(){
139     for (int r = 0 ; r < _n ; r++) {
140         for (int c = 0; c < _n ; c++) {
141             insert_a_value_to_a_vertex(r,c,0);
142         }
143     }
144 }
145
146
147 /*
148 Print a graph as dot file
149
150 ## Jagadeesh Vasudevanurthy #####
151 ## run: dot -Tps p395samanta.dot -o p395samanta.ps
152 ## See p395samanta.ps in GSview
153
154
155 digraph g {
156     label = "p395samanta"
157     0 -> 1
158     0 -> 2
159     1 -> 3
160     0 -> 4
161     1 -> 2
162     1 -> 4
163     2 -> 3
164     4 -> 3
165 }
```

```
166
167 -----*/
168 ostream& operator<<(ostream& o, const sgraph& g){
169     o << "## Jagadeesh Vasudevamurthy #####" << endl ;
170     o << "## run: dot -Tps graph.dot -o graph.ps" << endl ;
171     o << "## See graph.ps in GSview" << endl ;
172
173     o << "digraph g {" << endl ;
174     for (int r = 0; r < g._n; r++) {
175         for (int c = 0; c < g._n; c++) {
176             int v = g.weight_of_a_vertex(r,c) ;
177             if (v) {
178                 if (g._weighted_graph) {
179                     o << "    " << r << " -> " << c << "[label = " << v << "];" << endl ;
180                 }
181                 else {
182                     o << "    " << r << " -> " << c << endl ;
183                 }
184             }
185         }
186     }
187     o << "}" << endl ;
188     return o ;
189 }
190
191 /*-----*/
192 Print a graph as matrix
193 -----*/
194 void sgraph::print_matrix(const char *title) const {
195     if (title) {
196         cout << "***** " << title << " *****\n";
197     }
198     for (int r = 0; r < _n; r++) {
199         for (int c = 0; c < _n; c++) {
200             int v = weight_of_a_vertex(r,c) ;
201             if (v >= INFINITY) {
202                 cout << " L ";
203             }else {
204                 cout << v << " ";
205             }
206         }
207         cout << endl ;
208     }
209     cout << endl ;
210 }
211
212 /*-----*/
213 build graph
214 INPUT FILE FORMAT
215 0 1 0
216 0 2 0
217 1 3 0
218 0 4 0
219 1 2 0
220 1 4 0
```

```
221 2 3 0
222 4 2 0
223 4 3 0
224 -----
225 void sgraph::build_graph(const char *file_name, sgraph& g) {
226     ifstream in(file_name);
227     assert(in);
228     int from;
229     int to;
230     int weight;
231     int max = 0;
232
233     /* pass 1: get maximum */
234     while (!in.eof()) {
235         in >> from >> to >> weight;
236         if (weight) {
237             g._weighted_graph = true;
238         }
239         if (from > max) {
240             max = from;
241         }
242         if (to > max) {
243             max = to;
244         }
245     }
246     assert(max);
247     in.close();
248
249
250     g._n = max + 1; //because we start from 0.
251     g._matrix = ALLOCVEC(int,(g._n * g._n));
252     g._init_matrix();
253
254 {
255     /* pass 2: Fill the matrix*/
256     ifstream in(file_name);
257     assert(in);
258     while (!in.eof()) {
259         in >> from >> to >> weight;
260         if (!g._weighted_graph) {
261             weight = 1;
262         }
263         g.insert_a_value_to_a_vertex(from,to,weight);
264     }
265     in.close();
266 }
267 }
268
269 /*
270 Transpose of a matrix
271 -----
272 sgraph sgraph::transpose() const {
273     sgraph t(_n);
274     if (_n) {
275         for (int r = 0; r < _n ; r++) {
```

```
276     for (int c = 0; c < _n ; c++) {
277         int v = weight_of_a_vertex(r,c) ;
278         t.insert_a_value_to_a_vertex(c,r,v) ;
279     }
280 }
281 }
282 return t ;
283 }
284
285 /*-----*
286 product of a matrix
287
288 |(a b)| |(e f)| = |(ae + bg) (af + bh)|
289 |(c d)| |(g h)| = |(ce + dg) (cf + dh)|
290 -----*/
291 sgraph product(const sgraph& a, const sgraph& b) {
292     assert(a._n == b._n) ;
293     sgraph p(a._n) ;
294     if (a._n) {
295         for (int r = 0; r < a._n ; r++) {
296             for (int c = 0; c < b._n ; c++) {
297                 int s = 0 ;
298                 for (int k = 0; k < a._n; k++) {
299                     int x = a.weight_of_a_vertex(r,k) ;
300                     int y = b.weight_of_a_vertex(k,c) ;
301                     s = s + ( x * y) ;
302                 }
303                 //cout << r << "," << c << "=" << s << endl;
304                 p.insert_a_value_to_a_vertex(r,c,s) ;
305             }
306         }
307     }
308     return p ;
309 }
310
311 /*-----*
312 warshall algorithm
313 P[i,j] = P[i,j] or (P[i,k] and P[k,j])
314 -----*/
315 void sgraph::warshall(bool display) const {
316     int itt = 0 ;
317     sgraph w(_n) ;
318     if (_n) {
319         for (int k = 0; k < _n; k++) {
320             for (int i = 0; i < _n ; i++) {
321                 for (int j = 0; j < _n ; j++) {
322                     int pij_1 = weight_of_a_vertex(i,j) ;
323                     int pik_1 = weight_of_a_vertex(i,k) ;
324                     int pkj_1 = weight_of_a_vertex(k,j) ;
325                     bool pij = (pij_1 >= INFINITY || pij_1 == 0) ? false : true ;
326                     bool pik = (pik_1 >= INFINITY || pik_1 == 0) ? false : true ;
327                     bool pkj = (pkj_1 >= INFINITY || pkj_1 == 0) ? false : true ;
328                     bool s = ((pij) || (pik && pkj)) ;
329                     w.insert_a_value_to_a_vertex(i,j,s) ;
330                 }
331             }
332         }
333     }
334 }
```

```
331     }
332     if (display){
333         char t[100];
334         sprintf(t,"after %d iteration", ++itt);
335         w.print_matrix(t);
336     }
337 }
338 }
339 }
340
341 /*-----*
342 floyd algorithm
343 P[i,j] = P[i,j] or (P[i,k] and P[k,j])
344 -----*/
345 void sgraph::floyd(bool display) const {
346     int itt = 0;
347     sgraph w(*this); //we need to preserve all weights
348     w.make_unconnected_path_to_infinity();
349     if (_n) {
350         for (int k = 0; k < _n; k++) {
351             for (int i = 0; i < _n ; i++) {
352                 for (int j = 0; j < _n ; j++) {
353                     int pij = w.weight_of_a_vertex(i,j);
354                     int pik = w.weight_of_a_vertex(i,k);
355                     int pkj = w.weight_of_a_vertex(k,j);
356                     if ((pik + pkj) < pij) {
357                         w.insert_a_value_to_a_vertex(i,j,(pik+pkj));
358                     }
359                 }
360             }
361             if (display){
362                 char t[100];
363                 sprintf(t,"after %d iteration", ++itt);
364                 w.print_matrix(t);
365             }
366         }
367     }
368 }
369
370 /*-----*
371 c: color_array
372 n: node_array in dfs order
373 -----*/
374 void sgraph::_dfs_r(int v, int* c, int* n, int& nindex, bool& cycle) const {
375     if (cycle == false){
376         if (c[v] == GREEN){
377             //n[nindex++] = v; //APPLY
378             c[v] = BLUE;
379             int e;
380             foreach_adjacent_vertex((*this),v,e){
381                 _dfs_r(e,c,n,nindex,cycle);
382             }end_FOREACH_adjacent_vertex;
383             n[nindex++] = v; //APPLY
384             c[v] = RED;
385         }else if (c[v] == BLUE){
386             if (cycle == false){
387                 c[v] = GREEN;
388                 int e;
389                 foreach_adjacent_vertex((*this),v,e){
390                     _dfs_r(e,c,n,nindex,cycle);
391                 }end_FOREACH_adjacent_vertex;
392                 n[nindex++] = v; //APPLY
393                 c[v] = BLUE;
394             }else if (c[v] == RED){
395                 c[v] = GREEN;
396             }
397         }
398     }
399 }
```

```
386     cycle = true ;
387   }
388 }
389 }
390 */
391 /*-----*/
392 dfs
393 -----
394 void sgraph::dfs() const {
395   int* node_color_array = ALLOCVEC(int,_n) ;
396   int* dfs_nodes = ALLOCVEC(int,_n) ;
397   for (int i = 0; i < _n; i++) {
398     node_color_array[i] = GREEN ;
399   }
400   bool cycle = false ;
401   int dfs_nodes_index = 0 ;
402   _dfs_r(0,node_color_array,dfs_nodes,dfs_nodes_index,cycle) ;
403   if (cycle){
404     cout << "Graph has cycle " << endl ;
405   }else {
406     cout << "DFS ORDER is: " ;
407     for (int i = 0; i < dfs_nodes_index; i++) {
408       cout << dfs_nodes[i] ;
409       if (i < _n - 1){
410         cout << " -> " ;
411       }
412     }
413     cout << endl ;
414   }
415   FREEVEC(node_color_array) ;
416   FREEVEC(dfs_nodes) ;
417 }
418 */
419 /*-----*/
420 test_directed_graph
421 -----
422 void test_directed_graph(){
423   sgraph g ;
424   sgraph::build_graph("C:/jag/alg/course/code/c/data/p395samanta.dat",g) ;
425   cout << g ;
426   g.print_matrix("Graph g") ;
427
428   int v ;
429   foreach_vertex(g,v){
430     int e ;
431     int k = 0 ;
432     foreach_adjacent_vertex(g,v,e){
433       k++ ;
434     }end_FOREACH_adjacent_vertex;
435     cout << "outdegree of vertex " << v << " = " << k << endl ;
436   }end_FOREACH_vertex;
437
438   sgraph t = g.transpose() ;
439   t.print_matrix("Transpose of g = t") ;
440   sgraph gt = product(g,t);
```

```
441 gt.print_matrix("Product g.t - OUTDEGREE");
442 sgraph tg = product(t,g);
443 tg.print_matrix("Product t.g - INDEGREE");
444 g.warshall(true);
445 g.dfs();
446 }
447
448 /*-----
449 test_directed_weighted_graph_1
450 -----
451 void test_directed_weighted_graph_1(){
452 sgraph g;
453 sgraph::build_graph("C:/jag/alg/course/code/c/data/aho205.dat",g);
454 cout << g;
455 g.print_matrix("Graph g");
456
457 int v;
458 foreach_vertex(g,v){
459 int e;
460 int k = 0;
461 foreach_adjacent_vertex(g,v,e){
462 k++;
463 }end_FOREACH_ADJACENT_VERTEX;
464 cout << "outdegree of vertex " << v << " = " << k << endl;
465 }end_FOREACH_VERTEX;
466
467 sgraph t = g.transpose();
468 t.print_matrix("Transpose of g = t");
469 sgraph gt = product(g,t);
470 gt.print_matrix("Product g.t - OUTDEGREE");
471 sgraph tg = product(t,g);
472 tg.print_matrix("Product t.g - INDEGREE");
473 g.warshall(true);
474 g.floyd(true);
475 g.dfs();
476 }
477
478 /*-----
479 test_directed_weighted_graph_2
480 -----
481 void test_directed_weighted_graph_2(){
482 sgraph g;
483 sgraph::build_graph("C:/jag/alg/course/code/c/data/isreal84.dat",g);
484 cout << g;
485 g.print_matrix("Graph g");
486 g.dfs();
487 }
488
489
490 /*-----
491
492 -----
493 int main(){
494 test_directed_graph();
495 test_directed_weighted_graph_1();
```

```
496 test_directed_weighted_graph_2();  
497 return 0;  
498 }  
499
```

18.10 Code using graph representation

18.11 Problem set

Problem 18.11.1. Implement **dfs**, **bfs**, **floyd** and **warshall** algorithm on the graph data structure described in section 18.9. Test your code on atleast two big graphs. You must display your graph using **dot**.

Problem 18.11.2. Implement the algorithm described in figure 18.32 on the graph data structure described in section 18.9.

Path between two vertices

```
graph LR; 1((1)) --- 2((2)); 1 --- 3((3)); 2 --- 3; 2 --- 4((4)); 5((5)) --- 6((6)); 5 --- 7((7)); 6 --- 7;
```

```
bool is_path(int from, int to) {  
    if (from == to) return true ;  
    is_path(? ,to) ;  
    return false ; //where  
}
```

```
is_path(1,1) --> true  
is_path(1,3) --> true  
is_path(1,6) --> false
```

Write recursive routine and draw the recursion tree by hand

Figure 18.32: Finding path between two vertices