DATA STRUCTURES WITH C

(Common to CSE & ISE)

Subject Code: 10CS35 **I.A. Marks : 25**

Hours/Week : 04 **Exam Hours: 03**

Total Hours : 52 **Exam Marks: 100**

PART – A

UNIT - 1 **8 Hours**

BASIC CONCEPTS: Pointers and Dynamic Memory Allocation, Algorithm Specification, Data Abstraction, Performance Analysis, Performance Measurement

UNIT - 2 **6 Hours**

ARRAYS and STRUCTURES: Arrays, Dynamically Allocated Arrays, Structures and Unions, Polynomials, Sparse Matrices, Representation of Multidimensional Arrays

UNIT - 3 **6 Hours**

STACKS AND QUEUES: Stacks, Stacks Using Dynamic Arrays, Queues, Circular Queues Using Dynamic Arrays, Evaluation of Expressions, Multiple Stacks and Queues.

UNIT - 4 **6 Hours**

LINKED LISTS: Singly Linked lists and Chains, Representing Chains in C, Linked Stacks and Queues, Polynomials, Additional List operations, Sparse Matrices, Doubly Linked Lists

PART - B

UNIT - 5 **6 Hours**

TREES – 1: Introduction, Binary Trees, Binary Tree Traversals, Threaded Binary Trees, Heaps.

UNIT – 6 **6 Hours**

TREES – 2, GRAPHS: Binary Search Trees, Selection Trees, Forests, Representation of Disjoint Sets, Counting Binary Trees, The Graph Abstract Data Type.

UNIT - 7 **6 Hours**

PRIORITY QUEUES Single- and Double-Ended Priority Queues, Leftist Trees, Binomial Heaps, Fibonacci Heaps, Pairing Heaps.

UNIT - 8 **8 Hours**

EFFICIENT BINARY SEARCH TREES: Optimal Binary Search Trees, AVL Trees, Red-Black Trees, Splay Trees.

Text Book:

1. Horowitz, Sahni, Anderson-Freed: Fundamentals of Data Structures in C, 2nd Edition, Universities Press, 2007. (Chapters 1, 2.1 to 2.6, 3, 4, 5.1 to 5.3, 5.5 to 5.11, 6.1, 9.1 to 9.5, 10)

Reference Books:

1. Yedidyah, Augenstein, Tannenbaum: Data Structures Using C and C++, 2nd Edition, Pearson Education, 2003.

2. Debasis Samanta: Classic Data Structures, 2nd Edition, PHI, 2009.

3. Richard F. Gilberg and Behrouz A. Forouzan: Data Structures A Pseudocode Approach with C, Cengage Learning, 2005.

TABLE OF CONTENTS

UNIT - 1

BASIC CONCEPTS

1.1 Pointers and Dynamic Memory Allocation

1.2 Algorithm Specification

1.3 Data Abstraction

1.4 Performance Analysis

1.5 Performance Measurement

UNIT - 1

BASIC CONCEPTS

1.1 Pointers and Dynamic Memory Allocation

Pointers to data significantly improve performance for repetitive operations such as traversing strings, lookup tables, control tables and tree structures. In particular, it is often much cheaper in time and space to copy and dereference pointers than it is to copy and access the data to which the pointers point.Pointers are also used to hold the addresses of entry points for called subroutines in procedural programming and for run-time linking to dynamic link libraries (DLLs). In object-oriented programming, pointers to functions are used for binding methods, often using what are called virtual method tables.

Declaring a pointer variable is quite similar to declaring an normal variable all you have to do is to insert a star '*' operator before it.

General form of pointer declaration is -

type* name;

where type represent the type to which pointer thinks it is pointing to.

Pointers to machine defined as well as user-defined types can be made

Pointer Intialization: variable_type *pointer_name = 0;

or

variable_type *pointer_name = NULL;

char *pointer_name = "string value here";

1.2 Algorithm Specification

A pragmatic approach to algorithm specification and verification is presented. The language AL provides a level of abstraction between a mathematical specification notation and programming language, supporting compact but expressive algorithm description.

Proofs of correctness about algorithms written in AL can be done via an embedding of the semantics of the language in a proof system; implementations of algorithms can be done through translation to standard programming languages.

The proofs of correctness are more tractable than direct verification of programming language code; descriptions in AL are more easily related to executable programs than standard mathematical specifications. AL provides an independent, portable description which can be related to different proof systems and different programming languages.

Several interfaces have been explored and tools for fully automatic translation of AL specifications into the HOL logic and Standard ML executable code have been implemented. A substantial case study uses AL as the common specification language from which both the formal proofs of correctness and executable code have been produced.

1.3 Data Abstraction

Abstraction is the process by which data and programs are defined with a representation similar to its meaning (semantics), while hiding away the implementation details. Abstraction tries to reduce and factor out details so that the programmer can focus on a few concepts at a time. A system can have several *abstraction layers* whereby different meanings and amounts of detail are exposed to the programmer. For example, low-level abstraction layers expose details of the hardware where the program is run, while high-level layers deal with the businesslogic of the program.

1.4 Performance Analysis

Performance analysis involves gathering formal and informal data to help customers and sponsors define and achieve their goals. Performance analysis uncovers several perspectives on a problem or opportunity, determining any and all drivers towards or barriers to successful performance, and proposing a solution system based on what is discovered.

A lighter definition is:

Performance analysis is the front end of the front end. It's what we do to figure out what to do. Some synonyms are planning, scoping, auditing, and diagnostics.

What does a performance analyst do?

Here's a list of some of the things you may be doing as part of a performance analysis:

Interviewing a sponsor

Reading the annual report

Chatting at lunch with a group of customer service representatives

Reading the organization's policy on customer service, focusing particularly on the recognition and incentive aspects

Listening to audiotapes associates with customer service complaints

Leading a focus group with supervisors

Interviewing some randomly drawn representatives

Reviewing the call log

Reading an article in a professional journal on the subject of customer service performance improvement

Chatting at the supermarket with somebody who is a customer, who wants to tell you about her experience with customer service

We distinguish three basic steps in the performance analysis process:

data collection,

data transformation, and

data visualization.

Data collection is the process by which data about program performance are obtained from an executing program. Data are normally collected in a file, either during or after execution, although in some situations it may be presented to the user in real time.

1.5 Performance Measurement

‘When you can measure what you are speaking about and express it in numbers, you know something about it’.

‘You cannot manage what you cannot measure’.

These are two often-quoted statements that demonstrate why measurement is important. Yet it is surprising that organisations find the area of measurement so difficult to manage.

In the cycle of never-ending improvement, performance measurement plays an important role in:

• Identifying and tracking progress against organisational goals

• Identifying opportunities for improvement

• Comparing performance against both internal and external standards

Reviewing the performance of an organisation is also an important step when formulating the direction of the strategic activities. It is important to know where the strengths and weaknesses of the organisation lie, and as part of the 'Plan –Do – Check – Act' cycle, measurement plays a key role in quality and productivity improvement activities. The main reasons it is needed are:

• To ensure customer requirements have been met

• To be able to set sensible objectives and comply with them

• To provide standards for establishing comparisons

• To provide visibility and a "scoreboard" for people to monitor their own performance level

• To highlight quality problems and determine areas for priority attention

• To provide feedback for driving the improvement effort

It is also important to understand the impact of TQM on improvements in business performance, on sustaining current performance and reducing any possible decline in performance.

UNIT - 2

ARRAYS and STRUCTURES

2.1 Arrays

2.2 Dynamically Allocated Arrays

2.3 Structures and Unions

2.4 Polynomials

2.5 Sparse Matrices

2.6 Representation of Multidimensional Arrays

UNIT - 2

ARRAYS and STRUCTURES

2.1 Arrays

Definition :Array by definition is a variable that hold multiple elements which has the same data type.

Declaring Arrays :

We can declare an array by specify its data type, name and the number of elements the array holds between square brackets immediately following the array name. Here is the syntax:

```
1 x data_type array_name[size];
```

For example, to declare an integer array which contains 100 elements we can do as follows:

```
1 x int a[100];
```

There are some rules on array declaration. The data type can be any valid C data types including structure and union. The array name has to follow the rule of variable and the size of array has to be a positive constant integer.We can access array elements via indexes *array_name[index]*. Indexes of array starts from 0 not 1 so the highest elements of an array is *array_name[size-1]*

Initializing Arrays :

It is like a variable, an array can be initialized. To initialize an array, you provide initializing values which are enclosed within curly braces in the declaration and placed following an equals sign after the array name. Here is an example of initializing an integer array.

```
int list[5] = {2,1,3,7,8};
```

structure ARRAY(value, index)
declare CREATE( )array
RETRIEVE(array,index) value
STORE(array,index,value) array;
for all A array, i,j
index, x value let
RETRIEVE(CREATE,i) :: = error
RETRIEVE(STORE(A,i,x),j) :: =
if EQUAL(i,j) then x else RETRIEVE(A,j)
end
end ARRAY

To allocate a one-dimensional array of length *N* of some particular type, simply use malloc to allocate enough memory to hold *N* elements of the particular type, and then use the resulting pointer as if it were an array. For example, the following code snippet allocates a block of *N* ints, and then, using array notation, fills it with the values 0 through *N*-1:

```
int *A = malloc (sizeof (int) * N);

int i;
for (i = 0; i < N; i++)
A[i] = i;
```

This idea is very useful for dealing with strings, which in C are represented by arrays of chars, terminated with a '\0' character. These arrays are nearly always expressed as pointers in the declaration of functions, but accessed via C's array notation. For example, here is a function that implements strlen:

```
int strlen (char *s){
int i;
for (i = 0; s[i] != '\0'; i++)
return (i) }
```

2.2 Dynamically Allocated Arrays

array is a pointer-to-pointer-to-int: at the first level, it points to a block of pointers, one for each row. That first-level pointer is the first one we allocate; it has n rows elements, with each element big enough to hold a pointer-to-int, or int *. If we successfully allocate it, we then fill in the pointers (all n rows of them) with a pointer (also obtained from malloc) to n columns number of ints, the storage for that row of the array. If this isn't quite making sense, a picture should make everything clear:

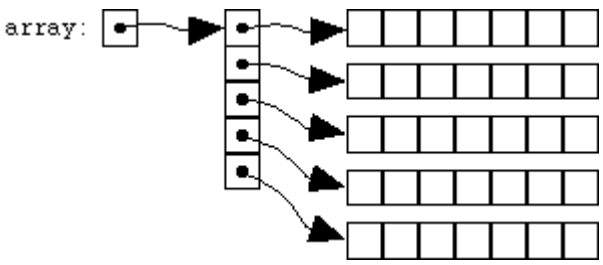


Fig1: representation of array

Once we've done this, we can (just as for the one-dimensional case) use array-like syntax to access our simulated multidimensional array. If we write

array[i][j]

The i'th pointer pointed to by array, and then for the j'th int pointed to by that inner pointer. (This is a pretty nice result: although some completely different machinery, involving two levels of pointer dereferencing, is going on behind the scenes, the simulated, dynamically-allocated two- dimensional

``array'' can still be accessed just as if it were an array of arrays, i.e. with the same pair of bracketed subscripts.).

2.3 Structures and Unions

Structure

A structure is a user-defined data type. You have the ability to define a new type of data considerably more complex than the types we have been using. A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. Structures are called “records” in some languages, notably Pascal. Structures help organize complicated data.

struct {field_list} variable_identifier;

struct struct_name

{

type1 fieldname1;

type2 fieldname2;

.

.

.

typeN fieldnameN;

};

struct struct_name variables;

The above format shown is not concrete and can vary, so different `

flavours of structure declaration is as shown.

struct

{

….

} variable_identifer;

Example

struct mob_equip;

```
{
long int IMEI;
char rel_date[10];
char model[10];
char brand[15];
};
```

Accessing a structure

A structure variable or a tag name of a structure can be used to access the members of a structure with the help of a special operator '.' –also called as member operator . In our previous example To access the idea of the IMEI of the mobile equipment in the structure mob_equip is done like this Since the structure variable can be treated as a normal variable All the IO functions for a normal variable holds good for the structure variable also with slight. The scanf statement to read the input to the IMEI is given below

scanf ("%d",&m1.IMEI);

Increment and decrement operation are same as the normal variables this includes postfix and prefix also. Member operator has more precedence than the increment or decrement. Say suppose in example quoted earlier we want count of student then

m1.count++; ++m1.count

Unions

Unions are very similar to structures, whatever discussed so far holds good for unions also then why dowe need unions? Size of unions depends on the size of its member of largest type or member with largest size, but this is not son in case of structures.

Example union abc1

```
{
int a;
float b;
char c;};
```

2.4 Polynomials

Polynomials appear in a wide variety of areas of mathematics and science. For example, they are used to form polynomial equations, which encode a wide range of problems, from elementary word problems to

complicated problems in the sciences; they are used to define polynomial functions, which appear in settings ranging from basic chemistry and physics to economics and social science; they are used in calculus and numerical analysis to approximate other functions. In advanced mathematics, polynomials are used to construct polynomial rings, a central concept in abstract algebra and algebraic geometry.

Polynomial comes form poly- (meaning "many") and -nomial (in this case meaning "term") ... so it says "many terms"

A polynomial can have:

constants (like 3, -20, or ½)

variables (like x and y)

exponents (like the 2 in y2) but only 0, 1, 2, 3, ... etc

That can be combined using:

+ addition,

- subtraction, and

× Multiplication

These are polynomials:

3x

x - 2

-6y2 - (7/9)x

3xyz + 3xy2z - 0.1xz - 200y + 0.5

512v5+ 99w5

1

2.5 Sparse Matrices

A sparse matrix is a matrix that allows special techniques to take advantage of the large number of zero elements. This definition helps to define "how many" zeros a matrix needs in order to be "sparse." The answer is that it depends on what the structure of the matrix is, and what you want to do with it. For example, a randomly generated sparse matrix with entries scattered randomly throughout the matrix is not sparse in the sense of Wilkinson.

Creating a sparse matrix:

If a matrix A is stored in ordinary (dense) format, then the command S = sparse(A) creates a copy of the matrix stored in sparse format. For example:

```
>> A = [0 0 1;1 0 2;0 -3 0]
A =
0 0 1
1 0 2
0 -3 0
>> S = sparse(A)
S =
(2,1) 1
(3,2) -3
(1,3) 1
(2,3) 2
```

2.6 Representation of Multidimensional Arrays

For a two-dimensional array, the element with indices *i*,*j* would have address $B + c \cdot i + d \cdot j$, where the coefficients *c* and *d* are the **row** and **column address increments**, respectively.

More generally, in a *k*-dimensional array, the address of an element with indices $i1, i2, \ldots, ik$ is

$B + c1 \cdot i1 + c2 \cdot i2 + \ldots + ck \cdot ik$

This formula requires only *k* multiplications and *k*−1 additions, for any array that can fit in memory. Moreover, if any coefficient is a fixed power of 2, the multiplication can be replaced by bit shifting.

The coefficients *ck* must be chosen so that every valid index tuple maps to the address of a distinct element. If the minimum legal value for every index is 0, then *B* is the address of the element whose indices are all zero. As in the one-dimensional case, the element indices may be changed by changing the base address *B*. Thus, if a two-dimensional array has rows and columns indexed from 1 to 10 and 1 to 20, respectively, then replacing *B* by $B + c1 - -3\ c1$ will cause them to be renumbered from 0 through 9 and 4 through 23, respectively. Taking advantage of this feature, some languages (like FORTRAN 77) specify that array indices begin at 1, as in mathematical tradition; while other languages (like Fortran 90, Pascal and Algol) let the user choose the minimum value for each index.

Compact layouts

There are two systematic compact layouts for a two-dimensional array. For example, consider the matrix

In the row-major order layout (adopted by C for statically declared arrays), the elements of each row are stored in consecutive positions:

In Column-major order (traditionally used by Fortran), the elements of each column are consecutive in memory:

For arrays with three or more indices, "row major order" puts in consecutive positions any two elements whose index tuples differ only by one in the last index. "Column major order" is analogous with respect to the first index. In systems which use processor cache or virtual memory, scanning an array is much faster if successive elements are stored in consecutive positions in memory, rather than sparsely scattered. Many algorithms that use multidimensional arrays will scan them in a predictable order. A programmer (or a sophisticated compiler) may use this information to choose between row- or column-major layout for each array. For example, when computing the product A•B of two matrices, it would be best to have A stored in row-major order, and B in column-major order.

The Representation of Multidimensional Arrays:

N-dimension, A[M0][M2]. . .[Mn-1]

Address of any entry A[i0][i1]...[in-1]

UNIT - 3

STACKS AND QUEUES

3.1 Stacks

3.2 Stacks Using Dynamic Arrays

3.3 Queues, Circular Queues Using Dynamic Arrays

3.4 Evaluation of Expressions

3.5 Multiple Stacks and Queues.

UNIT - 3

STACKS AND QUEUES

3.1 Stacks

A ***stack*** is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the ***top*** of the stack. A stack is a dynamic, constantly changing object as the definition of the stack provides for the insertion and deletion of items. It has single end of the stack as top of the stack, where both insertion and deletion of the elements takes place. The last element inserted into the stack is the first element deleted-***last in first out list (LIFO).*** After several insertions and deletions, it is possible to have the same frame again.

Primitive Operations

When an item is added to a stack, it is ***push***ed onto the stack. When an item is removed, it is ***pop***ped from the stack.

Given a stack *s*, and an item *i*, performing the operation *push(s,i)* adds an item *i* to the top of stack *s.*

push(s, H);

push(s, I);

push(s, J);

Operation *pop(s)* removes the top element. That is, if *i=pop(s)*, then the removed element is assigned to *i.*

pop(s);

Because of the push operation which adds elements to a stack, a stack is sometimes called a ***pushdown list.*** Conceptually, there is no upper limit on the number of items that may be kept in a stack. If a stack contains a single item and the stack is popped, the resulting stack contains no items and is called the ***empty stack.*** Push operation is applicable to any stack. Pop operation cannot be applied to the empty stack. If so, *underflow* happens. A Boolean operation *empty(s)*, returns TRUE if stack is empty. Otherwise FALSE, if stack is not empty.

Representing stacks in C

Before programming a problem solution that uses a stack, we must decide how to represent the stack in a programming language. It is an ordered collection of items. In C, we have ARRAY as an ordered collection of items. But a stack and an array are two different things. The number of elements in an array

is fixed. A stack is a dynamic object whose size is constantly changing. So, an array can be declared large enough for the maximum size of the stack. A stack in C is declared as a structure containing two objects:

• An array to hold the elements of the stack.

• An integer to indicate the position of the current stack top within the array.

```
#define STACKSIZE 100
struct stack {
int top;
int items[STACKSIZE];
};
```

The stack s may be declared by struct stack s;

The stack items may be int, float, char, etc. The empty stack contains no elements and can therefore be indicated by top= -1. To initialize a stack S to the empty state, we may initially execute

s.top= -1.

To determine stack empty condition,

```
if (s.top=-1)
stack empty;
else
stack is not empty;
```

The empty(s) may be considered as follows:

```
int empty(struct stack *ps)
{
if(ps->top== -1)
return(TRUE);
else
return(FALSE);
}
```

Implementing pop operation

If the stack is empty, print a warning message and halt execution. Remove the top element from the stack. Return this element to the calling program

```
int pop(struct stack *ps)
{
if(empty(ps)){
printf("%","stack underflow");
exit(1);
}
return(ps->items[ps->top--]);
}
```

3.2 Stacks Using Dynamic Arrays

For example:

```
Typedef struct
{
char *str;
} words;
main()
{
words x[100];
comesin.
}
```

For example here is the following array in which read individual words from a .txt file and save them word by word in the array:

Code:

```
char words[1000][15];
```

Here 1000 defines the number of words the array can save and each word may comprise of not more than 15 characters. Now the program should dynamically allocate the memory for the number of words it counts. For example, a .txt file may contain words greater that 1000. The program should count the number of words and allocate the memory accordingly. Since we cannot use a variable in place of [1000]

3.3 Queues

A queue is like a line of people waiting for a bank teller. The queue has a front and a rear.

When we talk of queues we talk about two distinct ends: the front and the rear. Additions to the queue take place at the rear. Deletions are made from the front. So, if a job is submitted for execution, it joins at the rear of the job queue. The job at the front of the queue is the next one to be executed

• New people must enter the queue at the rear. push, although it is usually called an enqueue operation.

• When an item is taken from the queue, it always comes from the front. **pop**, although it is usually called a **dequeue** operation.

Queue Operations

• Queue Overflow

• Insertion of the element into the queue

• Queue underflow

• Deletion of the element from the queue

• Display of the queue

Program for queue operations

```
struct Queue {
int que [size];
int front;
int rear;
}Q;
```

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 5
struct queue {
int que[size];
int front, rear;
} Q;
```

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 5
struct queue {
int que[size];
int front, rear;
} Q;
int Qfull ( ){
if (Q.rear >= size-1)
return 1;
else
return 0;
}
int Qempty( ){
if ((Q.front == -1)||(Q.front > Q.rear))
return 1;
else
return 0;
}
int insert (int item) {
if (Q.front == -1)
Q.front++;
Q.que[++Q.rear] = item;
return Q.rear;
}
Int delete () {
Int item;
```

```
Item = Q.que[Q.front];
Q.front++;
Return Q.front;
}
Void display () {
Int I;
For (i=Q.front;i<=Q.rear;i++)
Printf(" %d",Q.que[i]);
}
Void main (void) {
Int choice, item;
Q.front = -1; Q.Rear = -1;
do {
Printf("Enter your choice : 1:I, 2:D, 3:Display");
Scanf("%d", &choice);
Switch(choice){
Case 1: if(Qfull()) printf("Cannt Insert");
else scanf("%d",item); insert(item); break;
Case 2: if(Qempty()) printf("Underflow");
else delete(); break;
}
}
}
```

3.4 Circular Queues Using Dynamic Arrays

Circular Queue

• When an element moves past the end of a circular array, it wraps around to the beginning. A more efficient queue representation is obtained by regarding the array $Q(1:n)$ as circular. It now becomes more convenient to declare the array as $Q(0:n - 1)$. When rear = $n - 1$, the next element is entered at $Q(0)$ in case that spot is free. Using the same conventions as before, front will always point one position

counterclockwise from the first element in the queue. Again, *front* = *rear* if and only if the queue is empty. Initially we have *front* = *rear* = 1. Figure 3.4 illustrates some of the possible configurations for a circular queue containing the four elements *J*1-*J*4 with *n* > 4. The assumption of circularity changes the ADD and DELETE algorithms slightly. In order to add an element, it will be necessary to move *rear* one position clockwise, i.e.,

Queue Full Condition:

if(front == (rear+1)%size) Queue is Full

• Where do we insert:

rear = (rear + 1)%size; queue[rear]=item;

After deletion : front = (front+1)%size;

Example of a Circular Queue

• A Circular Q, the size of which is 5 has three elements 20, 40, and 60 where front is 0 and rear is 2. What are the values of after each of these operations:

Q = 20, 40, 60, - , - front–20[0], rear–60[2]

Insert item 50:

Q = 20, 40, 60, 50, - front-20[0], rear-50[3]

Insert item 10:

Q = 20, 40, 60, 50, 10 front-20[0], rear-10[4]

Q = 20, 40, 60, 50, 10 front-20[0], rear-10[4]

Insert 30

Rear = (rear + 1)%size = (4+1)%5 = 0, hence overflow.

Delete an item

delete 20, front = (front+1)%size = (0+1)%5=1

Delete an item

delete 40, front = (front+1)%size = (1+1)&5=2

Insert 30 at position 0

Rear = (rear + 1)%size = (4+1)%5 = 0

Similarly Insert 80 at position 1

3.5 Evaluation of Expressions

When pioneering computer scientists conceived the idea of higher level programming languages, they were faced with many technical hurdles. One of the biggest was the question of how to generate machine language instructions which would properly evaluate any arithmetic expression. A complex assignment statement such as *X A/B ** C + D* E - A * C* might have several meanings; and even if it were uniquely defined, say by a full use of parentheses, it still seemed a formidable task to generate a correct and reasonable instruction sequence. Fortunately the solution we have today is both elegant and simple. Moreover, it is so simple that this aspect of compiler writing is really one of the more minor issues. An expression is made up of operands, operators and delimiters. The expression above has five operands: *A,B,C,D,* and *E*. Though these are all one letter variables, operands can be any legal variable name or constant in our programming language. In any expression the values that variables take must be consistent with the operations performed on them. These operations are described by the operators. In most programming languages there are several kinds of operators which correspond to the different kinds of data a variable can hold. First, there are the basic arithmetic operators: plus, minus, times, divide, and exponentiation (+,-,*,/,**). Other arithmetic operators include unary plus, unary minus and **mod, ceil**, and **floor**. The latter three may sometimes be library subroutines rather than predefined operators. A second class are the relational operators: . These are usually defined to work for arithmetic operands, but they can just as easily work for character string data. ('CAT' is less than 'DOG' since it precedes 'DOG' in alphabetical order.) The result of an expression which contains relational operators is one of the two

constants: **true** or **false**. Such all expression is called Boolean, named after the mathematician George Boole, the father of symbolic logic.

The first problem with understanding the meaning of an expression is to decide in what order the operations are carried out. This means that every language must uniquely define such an order. For instance, if *A* = 4, *B* = *C* = 2, *D* = *E* = 3, then in eq. 3.1 we might want *X* to be assigned the value

4/(2 ** 2) + (3 * 3) - (4 * 2)

= (4/4) + 9 - 8

= 2.

Let us now consider an example. Suppose that we are asked to evaluate the following postfix expression:

6 2 3 + - 3 8 2 / + * 2 $ 3 +

Symb Opnd1 Opnd2 Value opndstk

6 6

2 6,2

3 6,2,3

+ 2 3 5 6,5

- 6 5 1 1

3 6 5 1 1,3

8 6 5 1 1,3,8

2 6 5 1 1,3,8,2

/ 8 2 4 1,3,4

8

+ 3 4 7 1,7

* 1 7 7 7

2 1 7 7 7,2

$ 7 2 49 49

3 7 2 49 49,3

+ 49 3 52 52

Program to evaluate postfix expression

Along with push, pop, empty operations, we have ***eval, isdigit*** and ***oper*** operations.

eval – the evaluation algorithm

```
double eval(char expr[])
{
int c, position;
double opnd1, opnd2, value;
struct stack opndstk;
opndstk.top=-1;
for (position=0 ;( c=expr [position])!='\0'; position++)
if (isdigit)
push (&opndstk, (double) (c-'0'));
else{
opnd2=pop (&opndstk);
```

9

```
opnd1=pop (&opndstk);
value=oper(c, opnd1,opnd2);
push (&opndstk. value);
}
return(pop(&opndstk));
}
```

isdigit – called by eval, to determine whether or not its argument is an operand

```
int isdigit(char symb)
{
return(symb>='0' && symb<='9');
}
```

oper – to implement the operation corresponding to an operator symbol

```
double oper(int symb, double op1, double op2)
{
switch (symb){
case '+' : return (op1+op2);
```

```
case '-' : return (op1-op2);
case '*' : return (op1*op2);
case '/' : return(op1/op2);
case '$' : return (pow (op1, op2);
default: printf ("%s","illegal operation");
exit(1);
}
}
```

Converting an expression from infix to postfix

Consider the given parentheses free infix expression:

A + B * C

Symb Postfix string opstk

1 A A

2 + A +

3 B AB +

4 * AB + *

5 C ABC + *

6 ABC * +

7 ABC * +

Consider the given parentheses infix expression:

(A+B)*C

Symb Postfix string Opstk

1 ((

2 A A (

3 + A (+

4 B AB (+

5) AB+

6 * AB+ *

7 C AB+C *

8 AB+C*

Program to convert an expression from infix to postfix

Along with pop, push, empty, popandtest, we also make use of additional functions such as, *isoperand, prcd, postfix.*

isoperand – returns TRUE if its argument is an operand and FALSE otherwise

prcd – accepts two operator symbols as arguments and returns TRUE if the first has precedence over the second when it appears to the left of the second in an infix string and FALSE otherwise

postfix – prints the postfix string

3.6 Multiple Stacks and Queues.

Up to now we have been concerned only with the representation of a single stack or a single queue in the memory of a computer. For these two cases we have seen efficient sequential data representations. What happens when a data representation is needed for several stacks and queues? Let us once again limit ourselves, to sequential mappings of these data objects into an array V(1:m). If we have only 2 stacks to represent. then the solution is simple. We can use V(1) for the bottom most element in stack 1 and V(m) for the corresponding element in stack 2. Stack 1 can grow towards V(m) and stack 2 towards V(1). It is therefore possible to utilize efficiently all the available space. Can we do the same when more than 2 stacks are to be represented? The answer is no, because a one dimensional array has only two fixed points V(1) and V(m) and each stack requires a fixed point for its bottommost element. When more than two stacks, say n, are to be represented sequentially, we can initially divide out the available memory V(1:m) into n segments and allocate one of these segments to each of the n stacks. This initial division of V(1:m) into segments may be done in proportion to expected sizes of the various stacks if the sizes are known. In the absence of such information, V(1:m) may be divided into equal segments. For each stack i we shall use B(i) to represent a position one less than the position in V for the bottommost element of that stack. T(i), 1 i n will point to the topmost element of stack i. We shall use the boundary condition B(i) = T(i) iff the i'th stack is empty. If we grow the i'th stack in lower memory indexes than the i + 1'st, then with roughly equal initial segments we have

B (i) = T (i) = m/n (i - 1), 1 i n ---- (3.2)

as the initial values of B(i) and T(i), (see figure 3.9). Stack i, 1 i n can grow from B(i) + 1 up to B(i + 1) before it catches up with the i + 1'st stack. It is convenient both for the discussion and the algorithms to define B(n + 1) = m. Using this scheme the add and delete algorithms become:

procedure ADD(i,X)

//add element X to the i'th stack, i n//

if T(i) = B(i + 1) then call STACK-FULL (i)

T(i) T(i) + 1

V(T(i)) X

//add X to the i'th stack//

end ADD

procedure DELETE(i,X)

//delete topmost element of stack *i*//

if *T*(*i*) = *B*(*i*) **then call** *STACK-EMPTY*(*i*)

X V*(*T*(i)) T*(*i*) *T*(*i*) - 1 **end** *DELETE*

The algorithms to add and delete appear to be a simple as in the case of only 1 or 2 stacks. This really is not the case since the STACK_FULL condition in algorithm ADD does not imply that all *m* locations of *V* are in use. In fact, there may be a lot of unused space between stacks *j* and *j* + 1 for 1 *j n* and *j i*. The procedure STACK_FULL (*i*) should therefore determine whether there is any free space in *V* and shift stacks around so as to make some of this free space available to the *i*'th stack.

Several strategies are possible for the design of algorithm STACK_FULL. We shall discuss one strategy in the text and look at some others in the exercises. The primary objective of algorithm STACK_FULL is to permit the adding of elements to stacks so long as there is some free space in *V*. One way to guarantee this is to design STACK_FULL along the following lines:

a) determine the least *j*, *i* < *j n* such that there is free space between stacks *j* and *j* + 1, i.e., *T*(*j*) < *B*(*j* + 1).

If there is such a j, then move stacks $i + 1$, $i + 2$, ...,j one position to the right (treating $V(1)$ as leftmost and $V(m)$ as rightmost), thereby creating a space between stacks i and $i + 1$.

b) if there is no j as in a), then look to the left of stack i. Find the largest j such that $1 \; j < i$ and there is space between stacks j and $j + 1$, i.e., $T(j) < B(j + 1)$. If there is such a j, then move stacks $j + 1$, $j + 2$, ...,i one space left creating a free space between stacks i and $i + 1$.

c) if there is no j satisfying either the conditions of a) or b), then all m spaces of V are utilized and there is no free space.

It should be clear that the worst case performance of this representation for the n stacks together with the above strategy for STACK_FULL would be rather poor. In fact, in the worst case $O(m)$ time may be needed for each insertion (see exercises). In the next chapter we shall see that if we do not limit ourselves to sequential mappings of data objects into arrays, then we can obtain a data representation for m stacks that has a much better worst case performance than the representation described here.

UNIT - 4

LINKED LISTS

4.1 Singly Linked lists and Chains

4.2 Representing Chains in C

4.3 Linked Stacks and Queues

4.4 Polynomials

4.5 Additional List operations

4.6 Sparse Matrices

4.7 Doubly Linked Lists

UNIT - 4

LINKED LISTS

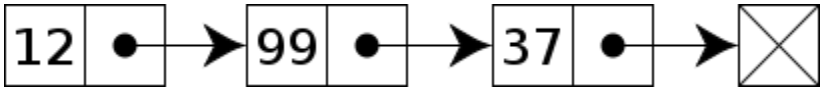
4.1 Singly Linked lists and Chains

Drawbacks of stacks and queues. During implementation, *overflow* occurs. No simple solution exists for more stacks and queues. In a sequential representation, the items of stack or queue are *implicitly* ordered by the sequential order of storage.

If the items of stack or queue are explicitly ordered, that is, each item contained within itself the address of the next item. Then a new data structure known as linear linked list. Each item in the list is called a node and contains two fields, an information field and a next address field. The information field holds the actual element on the list. The next address field contains the address of the next node in the list. Such an address, which is used to access a particular node, is known as a pointer.The null pointer is used to signal the end of a list. The list with no nodes – empty listor null list. The notations used in algorithms are:If p is a pointer to a node, node(p) refers to the node pointed to by p.

Singly linked list

Singly linked lists contain nodes which have a data field as well as a next field, which points to the next node in line of nodes.

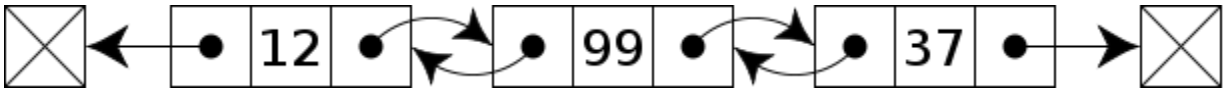


A singly linked list whose nodes contain two fields: an integer value and a link to the next node

Doubly linked list

Main article: Doubly linked list

In a doubly linked list, each node contains, besides the next-node link, a second link field pointing to the previous node in the sequence. The two links may be calledforward(s) and backwards, or next and prev(previous).

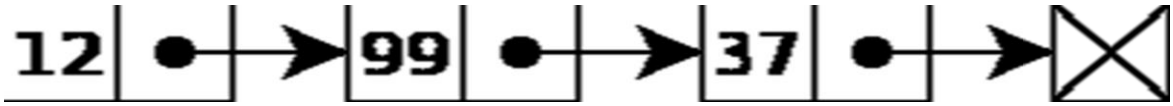


A doubly linked list whose nodes contain three fields: an integer value, the link forward to the next node, and the link backward to the previous node

A technique known as XOR-linking allows a doubly linked list to be implemented using a single link field in each node. However, this technique requires the ability to do bit operations on addresses, and therefore may not be available in some high-level language

information portion of the node that follows *node(p)* inthe list.

A linked list (or more clearly, "singly linked list") is a datastructure that consists of a sequence of nodes each of which contains a reference (i.e., a *link*) to the next node in the sequence.



A linked list whose nodes contain two fields: an integer value and a link to the next node

Linked lists are among the simplest and most common data structures. They can be used to implement several other common abstract data structures, including stacks, queues, associative arrays, and symbolic expressions, though it is not uncommon to implement the other data structures directly without using a list as the basis of implementation.

The principal benefit of a linked list over a conventional array is that the list elements can easily be added or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk. Linked lists allow insertion and removal of nodes at any point in the list, and can do so with a constant number of operations if the link previous to the link being added or removed is maintained during list traversal.

- There is a link or pointer from one element to the next.
- The last node has a NULL (or 0) pointer

An array and a sequential mapping is used to represent simple data structures in the previous chapters

•This representation has the property that successive nodes of the data object are stored a fixed distance apart

(1) If the element aijis stored at location Lij, then aij+1is at the location Lij+1

(2) If the i-thelement in a queue is at location Li, then the (i+1)-th element is at location Li+1% n for the circular representation

(3) If the topmost node of a stack is at location LT , then the node beneath it is at location LT-1, and so on

•When a sequential mapping is used for ordered lists, operations such as insertion and deletion of arbitrary elements become expensive.

In a linked representation–To access list elements in the correct order, with each element we store the address or location of the next element in the list–A linked list is comprised of nodes; each node has zero or more data fields and one or more link or pointer fields.

4.3 Linked Stacks and Queues

Pushing a Linked Stack

Error code Stack :: push(const Stack entry &item)

/* Post: Stack entry item is added to the top of the Stack; returns success or

returns a code of over_ow if dynamic memory is exhausted. */

```
{
Node *new top = new Node(item, top node);
if (new top == NULL) return over_ow;
top node = new top;
return success;
}
```

Popping a Linked Stack

```
Error code Stack :: pop( )
/* Post: The top of the Stack is removed. If the Stack is empty the method returns
under_ow; otherwise it returns success. */
{
Node *old top = top node;
if (top node == NULL) return under_ow;
top node = old top->next;
delete old top;
return success;
}
```

A queue is a particular kind of collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once an element is added, all elements that were added before have to be removed before the new element can be invoked. A queue is an example of a lineardatastructure.

Queues provide services, transport, and operations research where various entities such as data, objects,

persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a buffer.

```
#include<malloc.h>
#include<stdio.h> structnode{ intvalue; structnode*next;
};

voidInit(structnode*n){
n->next=NULL;
} voidEnqueue(structnode*root,intvalue){ structnode*j=(structnode*)malloc(sizeof(structnode)); j-
>value=value;
j->next=NULL;
structnode*temp ;
temp=root;
while(temp->next!=NULL)
{
temp=temp->next;
}
temp->next=j;
printf("Value Enqueued is : %d\n",value);

}
voidDequeue(structnode*root)
{
if(root->next==NULL)
{
printf("NoElementtoDequeue\n");
```

```
}
else
{ structnode*temp; temp=root->next;
root->next=temp->next; printf("ValueDequeuedis%d\n",temp->value); free(temp);
}
}
voidmain()
{structnodesample_queue; Init(&sample_queue); Enqueue(&sample_queue,10);
Enqueue(&sample_queue,50); Enqueue(&sample_queue,570); Enqueue(&sample_queue,5710);
Dequeue(&sample_queue); Dequeue(&sample_queue); Dequeue(&sample_queue);
}
```

4.4 Polynomials

A polynomial (from Greek *poly*, "many" and medieval Latin *binomium*, "binomial") is an expression of finite length constructed from variables (also known as indeterminates) and constants, using only the operations of addition, subtraction, multiplication, and non-negative integer exponents. For example, $x2 - 4x + 7$ is a polynomial, but $x2 - 4/x + 7x3/2$ is not, because its second term involves division by the variable x (4/x) and because its third term contains an exponent that is not a whole number (3/2). The term "polynomial" can also be used as an adjective, for quantities that can be expressed as a polynomial of some parameter, as in "polynomial time" which is used in computational complexity theory.

Polynomials appear in a wide variety of areas of mathematics and science. For example, they are used to form polynomial equations, which encode a wide range of problems, from elementary word problems to complicated problems in the sciences.

A polynomial is a mathematical expression involving a sum of powers in one or more variables multiplied by coefficients.

4.5 Additional List operations

It is often necessary and desirable to build a variety of routines for manipulating singly linked lists. Some that we have already seen are: 1) INIT which originally links together the *AV* list; 2) GETNODE and 3) RET

which get and return nodes to *AV*. Another useful operation is one which inverts a chain. This routine is especially interesting because it can be done "in place" if we make use of 3 pointers.

//a chain pointed at by *X* is inverted so that if *X* = (*a*1, ...,*am*)

then after execution *X* = (*am*, ...,*a*1)//

p X;*q* 0

while *p* 0 **do**

r q;*q p* //*r* follows *q; q* follows *p*//

p LINK(*p*) //*p* moves to next node//

LINK(*q*) *r* //link *q* to previous node//

end

X q

end *INVERT*

The reader should try this algorithm out on at least 3 examples: the empty list, and lists of length 1 and 2 to convince himself that he understands the mechanism. For a list of *m* 1 nodes, the **while** loop is executed *m* times and so the computing time is linear or *O*(*m*).

Another useful subroutine is one which concatenates two chains *X* and *Y*.

procedure *CONCATENATE*(*X, Y, Z*)

//*X* = (*a*1, ...,*am*), *Y* = (*b*1, ...,*bn*), *m*,*n* 0, produces a new chain

Z = (*a*1, ...,*am*,*b*1 , ...,*bn*)//

Z X

if *X* = 0 **then** [*Z Y*; **return**]

if *Y* = 0 **then return**

p X

while *LINK*(*p*) 0 **do** //find last node of *X*//

p LINK(*p*)

end

LINK(*p*) *Y* //link last node of X to Y//

end *CONCATENATE*

This algorithm is also linear in the length of the first list. From an aesthetic point of view it is nicer to write this procedure using the case statement in SPARKS. This would look like:

procedure *CONCATENATE*(*X, Y, Z*)

case

: *X = 0 :Z Y*

: *Y = 0 : Z X*

: **else** : *p X; Z X*

while *LINK*(*p*) 0 **do**

p LINK (*p*)

end

LINK(*p*) *Y*

end

end *CONCATENATE*

Suppose we want to insert a new node at the front of this list. We have to change the LINK field of the node containing *x*3. This requires that we move down the entire length of *A* until we find the last node. It is more convenient if the name of a circular list points to the last node rather than the first.

Now we can write procedures which insert a node at the front or at the rear of a circular list and take a fixed amount of time.

procedure *INSERT__FRONT*(*A, X*)

//insert the node pointed at by X to the front of the circular list

A, where A points to the last node//

if A = 0 **then** [*A X*

LINK (*X*) *A*]

else [*LINK*(*X*) *LINK* (*A*)

LINK(*A*) *X*]

end *INSERT--FRONT*

To insert *X* at the rear, one only needs to add the additional statement *A X* to the **else** clause of *INSERT_-_FRONT.*

As a last example of a simple procedure for circular lists, we write a function which determines the length of such a list.

procedure *LENGTH*(*A*)

//find the length of the circular list *A*//

i 0

if A 0 **then** [*ptr A*

repeat

i i + 1; *ptr* LINK(*ptr*)

until *ptr* = *A*]

return (*i*)

end *LENGTH*

4.6 Sparse Matrices

A sparse matrix is a matrix populated primarily with zeros (Stoer & Bulirsch 2002, p. 619). The term itself was coined by Harry M. Markowitz.

Conceptually, sparsity corresponds to systems which are loosely coupled. Consider a line of balls connected by springs from one to the next; this is a sparse system. By contrast, if the same line of balls had springs connecting each ball to all other balls, the system would be represented by a dense matrix. The concept of sparsity is useful in combinatorics and application areas such as network theory, which have a low density of significant data or connections.

A sparse matrix is a matrix that allows special techniques to take advantage of the large number of zero elements. This definition helps to define "how many" zeros a matrix needs in order to be "sparse." The answer is that it depends on what the structure of the matrix is, and what you want to do with it. For example, a randomly generated sparse matrix with entries scattered randomly throughout the matrix is not sparse in the sense of Wilkinson (for direct methods) since it takes .

Creating a sparse matrix

If a matrix A is stored in ordinary (dense) format, then the command S = sparse(A) creates a copy of the matrix stored in sparse format. For example:

>> A = [0 0 1;1 0 2;0 -3 0]

A =

0 0 1

1 0 2

0 -3 0

>> S = sparse(A)

S =

(2,1) 1

(3,2) -3

(1,3) 1

(2,3) 2

>> whos

Name Size Bytes Class

A 3x3 72 double array

S 3x3 64 sparse array

Grand total is 13 elements using 136 bytes

Unfortunately, this form of the sparse command is not particularly useful, since if A is large, it can be very time-consuming to first create it in dense format. The command S = sparse(m,n) creates an zero matrix in sparse format. Entries can then be added one-by-one:

>> A = sparse(3,2)

A =

All zero sparse: 3-by-2

>> A(1,2)=1;

>> A(3,1)=4;

>> A(3,2)=-1;

>> A

A =

(3,1) 4

(1,2) 1

(3,2) -1

4.7 Doubly Linked Lists

Although a circularly linked list has advantages over linear lists, it still has some drawbacks. One cannot traverse such a list backward. Double-linked lists require more space per node , and their elementary operations are more expensive; but they are often easier to manipulate because they allow sequential access to the list in both directions. In particular, one can insert or delete a node in a constant number of operations given only that node's address. (Compared with singly-linked lists, which require the *previous* node's address in order to correctly insert or delete.) Some algorithms require access in both directions.

On the other hand, they do not allow tail-sharing, and cannot be used as persistent data structures.

Operations on Doubly Linked Lists

One operation that can be performed on doubly linked list but not on ordinary linked list is to delete a given node. The following c routine deletes the node pointed by pfrom a doubly linked list and stores its contents in x. It is called by delete(p).

```
delete( p )
{
NODEPTR p, q, r;
int *px;
if ( p = = NULL )
{
printf(“ Void Deletion \n”);
return;
}
*px = p -> info;
q = p -> left;
r = p -> right;
q -> right = r;
r -> left = q;
freenode( p );
return;
}
```

A node can be inserted on the right or on the left of a given node. Let us consider insertion at right side of a given node. The routine insert right inserts a node with information field x to right of node(p) in a doubly linked list.

```
insertright( p, x) {
NODEPTR p, q, r;
int x;
if ( p = = NULL ) {
printf(“ Void Insertion \n”);
return;
}
q = getnode();
q -> info = x;
r = p -> right;
r -> left = q;
q -> right = r;
q -> left = p;
p -> left = q;
return;
}
```

UNIT - 5

TREES – 1

5.1 Introduction

5.2 Binary Trees

5.3 Binary Tree Traversals

5.4 Threaded Binary Trees

5.5 Heaps.

UNIT - 5

TREES – 1

5.1 Introduction

A *tree* is a finite set of one or more nodes such that: (i) there is a specially designated node called the *root*; (ii) the remaining nodes are partitioned into *n* 0 disjoint sets *T*1, ...,*Tn* where each of these sets is a tree. *T*1, ...,*Tn* are called the *subtrees* of the root. A tree structure means that the data is organized so that items of information are related by branches. One very common place where such a structure arises is in the investigation of genealogies.

AbstractDataType tree{

instances

A set of elements:

(1) empty or having a distinguished root element

(2) each non-root element having exactly one parent element operations

root()

degree()

child(k)

}

Some basic terminology for trees:

- Trees are formed from *nodes* and *edges*. Nodes are sometimes called *vertices*. Edges are sometimes called *branches*.
- Nodes may have a number of properties including *value* and *label*.
- Edges are used to relate nodes to each other. In a tree, this relation is called "parenthood."
- An edge {a,b} between nodes a and b establishes a as the *parent* of b. Also, b is called a *child*.

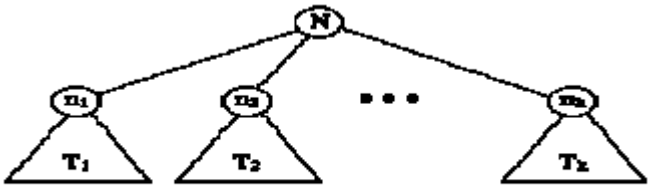
Although edges are usually drawn as simple lines, they are really directed from parent to child. In tree drawings, this is top-to-bottom.

Informal Definition: a *tree* is a collection of nodes, one of which is distinguished as "root," along with a relation ("parenthood") that is shown by edges.

Formal Definition: This definition is "recursive" in that it defines tree in terms of itself. The definition is also "constructive" in that it describes how to construct a tree.

1. A single node is a tree. It is "root."

2. Suppose N is a node and T1, T2, ..., Tk are trees with roots n1, n2, ...,nk, respectively. We can construct a new tree T by making N the parent of the nodes n1, n2, ..., nk. Then, N is the root of T and T1, T2, ..., Tk are subtrees.



The tree T, constructed using k subtrees

More terminology

- A node is either *internal* or it is a *leaf*.
- A *leaf* is a node that has no children.
- Every node in a tree (except root) has exactly one parent.
- The *degree of a node* is the number of children it has.
- The *degree of a tree* is the maximum degree of all of its nodes.

Paths and Levels

Definition: A *path* is a sequence of nodes n1, n2, ..., nk such that node ni is the parent of node ni+1 for all 1 <= i <= k.

Definition: The *length* of a path is the number of edges on the path (one less than the number of nodes).

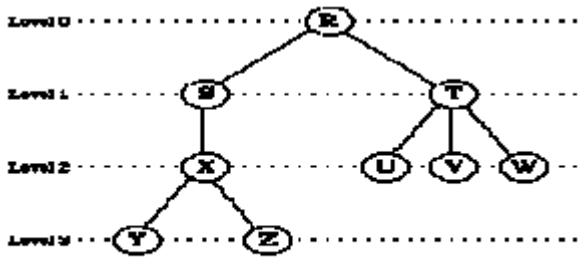
Definition: The *descendents* of a node are all the nodes that are on some path from the node to any leaf.

Definition: The *ancestors* of a node are all the nodes that are on the path from the node to the root.

Definition: The *depth* of a node is the length of the path from root to the node. The depth of a node is sometimes called its *level*.

Definition: The *height of a node* is the length of the longest path from the node to a leaf.

Definition: the *height of a tree* is the height of its root.



A general tree, showing node depths (levels)

In the example above:

- The nodes Y, Z, U, V, and W are leaf nodes.
- The nodes R, S, T, and X are internal nodes.
- The degree of node T is 3. The degree of node S is 1.
- The depth of node X is 2. The depth of node Z is 3.
- The height of node Z is zero. The height of node S is 2. The height of node R is 3.
- The height of the tree is the same as the height of its root R. Therefore the height of the tree is 3.
- The sequence of nodes R,S,X is a path.
- The sequence of nodes R,X,Y is not a path because the sequence does not satisfy the
- "parenthood" property (R is not the parent of X).

5.2 Binary Trees

Definition: A binary tree is a tree in which each node has degree of exactly 2 and the children of each node are distinguished as "left" and "right." Some of the children of a node may be empty.

Formal Definition: A binary tree is:

1. either empty, or

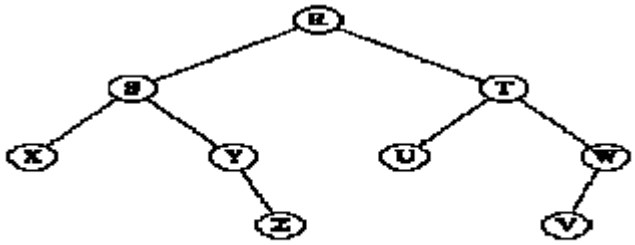
2. it is a node that has a left and a right subtree, each of which is a binary tree.

Definition: A *full binary tree* (FBT) is a binary tree in which each node has exactly 2 non-empty children or exactly two empty children, and all the leaves are on the same level. (Note that this definition differs from the text definition).

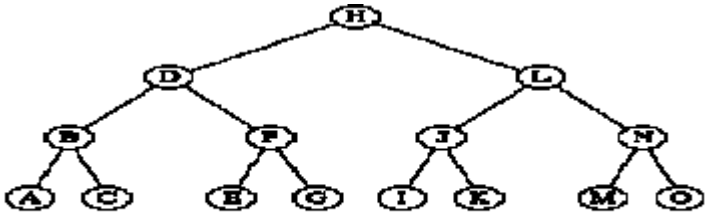
Definition: A *complete binary tree* (CBT) is a FBT except, perhaps, that the deepest level may not be completely filled. If not completely filled, it is filled from left-to-right.

A FBT is a CBT, but not vice-versa.

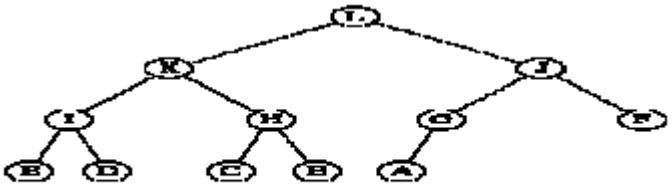
Examples of Binary Trees



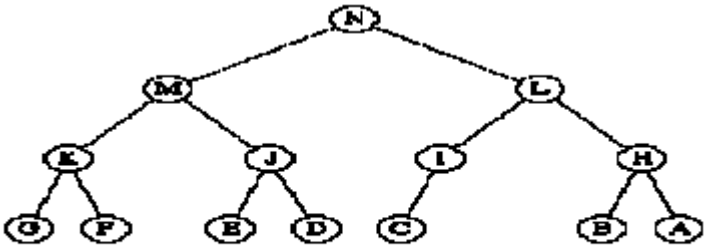
A Binary Tree. As usual, the empty children are not explicitly shown.



A Full Binary Tree. In a FBT, the number of nodes at level i is 2^i.



A Complete Binary Tree.



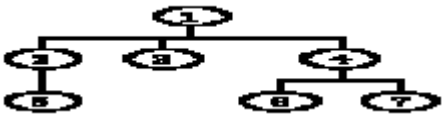
Not a Complete Binary Tree. The tree above is not a CBT because the deepest level is not filled from left-to-right.

5.3 Binary Tree Traversals

There are many operations that we often want to perform on trees. One notion that arises frequently is the idea of traversing a tree or visiting each node in the tree exactly once. A full traversal produces a linear order for the information in a tree. This linear order may be familiar and useful. When traversing a binary tree we

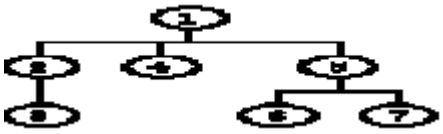
want to treat each node and its subtrees in the same fashion. If we let *L, D, R* stand for moving left, printing the data, and moving right when at a node then there are six possible combinations of traversal: *LDR, LRD, DLR, DRL, RDL,* and *RLD*. If we adopt the convention that we traverse left before right then only three traversals remain: *LDR, LRD* and *DLR*. To these we assign the names inorder, postorder and preorder because there is a natural correspondence between these traversals and producing the infix, postfix and prefix forms of an expression.

Level order



```
x := root()
if( x ) queue (x)
while( queue not empty ){
x := dequeue()
visit()
i=1; while( i <= degree() ){
queue( child(i) )
}
}
```

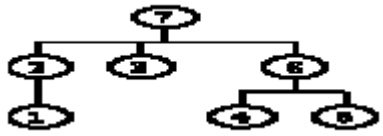
Preorder



```
procedure preorder(x){
visit(x)
i=1; while( i <= degree() ){
preorder( child(i) )
}
```

}

Postorder



procedure postorder(x){

i=1; while(i <= degree()){

postorder(child(i))

}

visit(x)

}

Inorder

Meaningful just for binary trees.

procedure inorder(x){

if(left_child_for(x)) { inorder(left_child(x)) }

visit(x)

if(right_child_for(x)) { inorder(right_child(x)) }

}

5.4 Threaded Binary Trees

If we look carefully at the linked representation of any binary tree, we notice that there are more null links than actual pointers. As we saw before, there are $n + 1$ null links and $2n$ total links. A clever way to make use of these null links has been devised by A. J. Perlis and C. Thornton. Their idea is to replace the null links by pointers, called threads, to other nodes in the tree. If the RCHILD(*P*) is normally equal to zero, we will replace it *by a pointer to the node which would be printed after P when traversing the tree in inorder*. A null LCHILD link at node *P* is replaced *by a pointer to the node which immediately precedes*

node P in inorder.

The tree *T* has 9 nodes and 10 null links which have been replaced by threads. If we traverse *T* in inorder the nodes will be visited in the order *H D I B E A F C G*. For example node *E* has a predecessor thread which points to B and a successor thread which points to *A*.

In the memory representation we must be able to distinguish between threads and normal pointers. This is done by adding two extra one bit fields LBIT and RBIT.

LBIT(*P*) =1 if LCHILD(*P*) is a normal pointer

LBIT(*P*) = 0 if LCHILD(*P*) is a thread

RBIT(*P*) = 1 if RCHILD(*P*) is a normal pointer

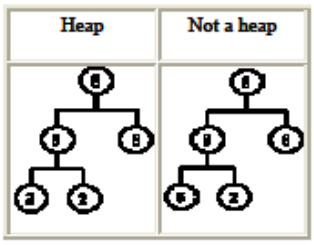
RBIT(*P*) = 0 if RCHILD(*P*) is a thread

5.5 Heaps.

A **heap** is a complete tree with an ordering-relation R holding between each node and its descendant.

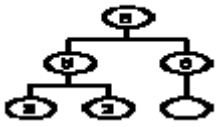
Examples for R: smaller-than, bigger-than

Assumption: In what follows, R is the relation 'bigger-than', and the trees have degree 2.

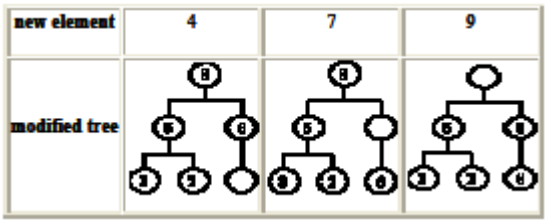


Adding an Element

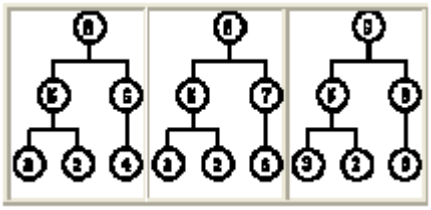
1. Add a node to the tree



2. Move the elements in the path from the root to the new node one position down, if they are smaller than the new element



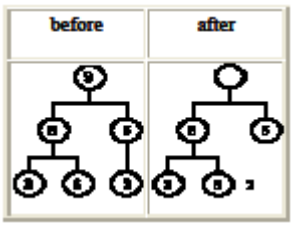
3. Insert the new element to the vacant node



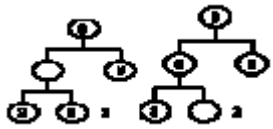
4. A complete tree of n nodes has depth log n , hence the time complexity is O(log n)

Deleting an Element

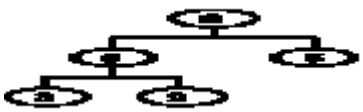
1. Delete the value from the root node, and delete the last node while saving its value.



2. As long as the saved value is smaller than a child of the vacant node, move up into the vacant node the largest value of the children.



3. Insert the saved value into the vacant node



4. The time complexity is O(log n)

Initialization:

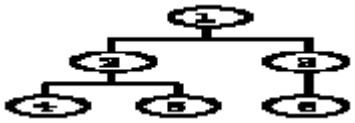
Brute Force

Given a sequence of n values e1, ..., en, repeatedly use the insertion module on the n given values.

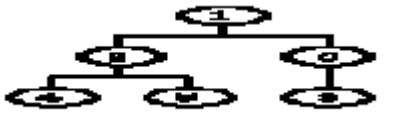
- Level h in a complete tree has at most 2h-1 = O(2n) elements
- Levels 1, ..., h - 1 have 20 + 21 + + 2h-2 = O(2h) elements
- Each element requires O(log n) time. Hence, brute force initialization requires O(n log n) time.

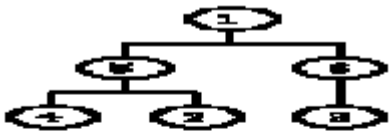
Efficient

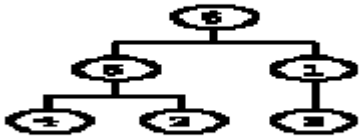
Insert the n elements e1, ..., en into a complete tree

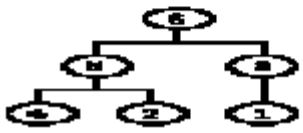


For each node, starting from the last one and ending at the root, reorganize into a heap the subtree whose root node is given. The reorganization is performed by interchanging the new element with the child of greater value, until the new element is greater than its children.









UNIT – 6

TREES – 2, GRAPHS

6.1 Binary Search Trees

6.2 Selection Trees

6.3 Forests, Representation of Disjoint Sets

6.4 Counting Binary Trees

6.5 The Graph Abstract Data Type.

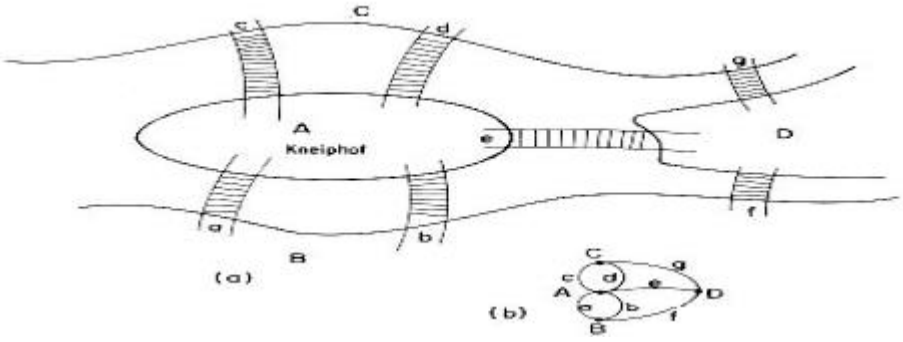
UNIT – 6

TREES – 2, GRAPHS

6.1 Binary Search Trees

Introduction

The first recorded evidence of the use of graphs dates back to 1736 when Euler used them to solve the now classical Koenigsberg bridge problem.Some of the applications of graphs are: analysis of electrical circuits, finding shortest routes, analysis of project planning, identification of chemical compounds, statistical mechanics, genetics, cybernetics, linguistics, social sciences, etc. Indeed, it might well be said that of all mathematical structures, graphs are the most widely used.



Definitions and Terminology

A graph, *G*, consists of two sets *V* and *E*. *V* is a finite non-empty set of *vertices. E* is a set of pairs of vertices, these pairs are called *edges*. *V*(*G*) and *E*(*G*) will represent the sets of vertices and edges of graph *G*.

We will also write *G* = (*V*,*E*) to represent a graph.

In an *undirected graph* the pair of vertices representing any edge is unordered . Thus, the pairs (*v*1, *v*2) and (*v*2, *v*1) represent the same edge.

In a *directed graph* each edge is represented by a directed pair (*v*1, *v*2). *v*1 is the *tail* and *v*2 the *head* of the edge. Therefore <*v*2, *v*1> and <*v*1, *v*2> represent two different edges. Figure 6.2 shows three graphs *G*1, *G*2 and *G*3.

The graphs *G*1 and *G*2 are undirected. *G*3 is a directed graph.

V (*G*1) = {1,2,3,4}; *E*(*G*1) = {(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)}

V (*G*2) = {1,2,3,4,5,6,7}; *E*(*G*2) = {(1,2),(1,3),(2,4),(2,5),(3,6),(3,7)}

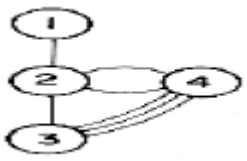
V (*G*3) = {1,2,3}; *E*(*G*3) = {<1,2>, <2,1>, <2,3>}.

Note that the edges of a directed graph are drawn with an arrow from the tail to the head. The graph *G*2 is also a tree while the graphs *G*1 and *G*3 are not. Trees can be defined as a special case of graphs, In addition, since *E*(*G*) is a set, a graph may not have multiple occurrences of the same edge. When this restriction is removed from a graph, the resulting data object is referred to as a multigraph. The data object of figure 6.3 is a multigraph which is not a graph.

The number of distinct unordered pairs (*vi*,*vj*) with *vi vj* in a graph with *n* vertices is *n*(*n* - 1)/2. This is the maximum number of edges in any *n* vertex undirected graph.

An *n* vertex undirected graph with exactly *n*(*n* - 1)/2 edges is said to be *complete. G*1 is the complete graph on 4 vertices while *G*2 and *G*3 are not complete graphs. In the case of a directed graph on *n* vertices the maximum number of edges is *n*(*n* - 1).

If (*v*1,*v*2) is an edge in *E*(*G*), then we shall say the vertices *v*1 and *v*2 are *adjacent* and that the edge (*v*1,*v*2) is *incident* on vertices *v*1 and *v*2. The vertices adjacent to vertex 2 in *G*2 are 4, 5 and 1. The edges incident on vertex 3 in *G*2 are (1,3), (3,6) and (3,7). If <*v*1,*v*2> is a directed edge, then vertex *v*1 will be said to be *adjacent to v*2 while *v*2 is *adjacent from v*1. The edge <*v*1,*v*2> is incident to *v*1 and *v*2. In *G*3 the edges incident to vertex 2 are <1,2>, <2,1> and <2,3>.



6.2 Selection Trees

A *subgraph* of *G* is a graph *G'* such that *V*(*G'*) *V*(*G*) and *E*(*G'*) *E*(*G*). Figure 6.4 shows some of the subgraphs of *G*1 and *G*3.

A *path* from vertex *vp* to vertex *vq* in graph *G* is a sequence of vertices *vp*,*vi*1,*vi*2, ...,*vin*,*vq* such that (*vp*,*vi*1),(*vi*1,*vi*2), ...,(*vin*,*vq*) are edges in *E*(*G*). If *G'* is directed then the path consists of <*vp*,*vi*1>,<*vi*,*vi*2>, ...,<*vin*,*vq*>, edges in *E*(*G'*).

A *simple path* is a path in which all vertices except possibly the first and last are distinct. A path such as (1,2) (2,4) (4,3) we write as 1,2,4,3. Paths 1,2,4,3 and 1,2,4,2 are both of length 3 in *G*1. The first is a simple path while the second is not. 1,2,3 is a simple directed path in *G*3. 1,2,3,2 is not a path in *G*3 as the edge <3,2> is not in *E*(*G*3).

A *cycle* is a simple path in which the first and last vertices are the same. 1,2,3,1 is a cycle in *G*1. 1,2,1 is a cycle in *G*3. For the case of directed graphs we normally add on the prefix "directed" to the terms cycle and path.

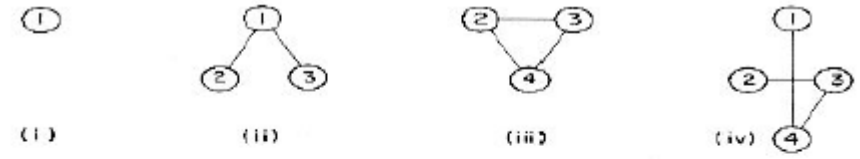
In an undirected graph, *G*, two vertices *v*1 and *v*2 are said to be *connected* if there is a path in *G* from *v*1 to *v*2 (since *G* is undirected, this means there must also be a path from *v*2 to *v*1). An undirected graph is said to be connected if for every pair of distinct vertices *vi*, *vi* in *V*(*G*) there is a path from *vi* to *vj* in *G*.

Graphs *G*1 and *G*2 are connected while *G*4 of figure 6.5 is not.

A *connected component* or simply a component of an undirected graph is a *maximal* connected subgraph.

*G*4 has two components *H*1 and *H*2. A *tree* is a connected acyclic (i.e., has no cycles) graph . A directed graph *G* is said to be *strongly connected* if for every pair of distinct vertices *vi*, vj in *V*(*G*) there is a directed path from *vi* to *vj* and also from *vj* to *vi*. The graph *G*3 is not strongly connected as there is no path from *v*3 to *v*2.

A *strongly connected component* is a maximal subgraph that is strongly connected. *G*3 has two strongly connected components.

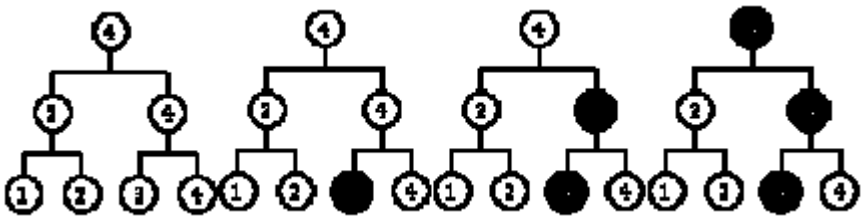


(a) Some of the subgraphs of G_1

A selection tree is a complete binary tree in which the leaf nodes hold a set of keys, and each internal node holds the "winner" key among its children.

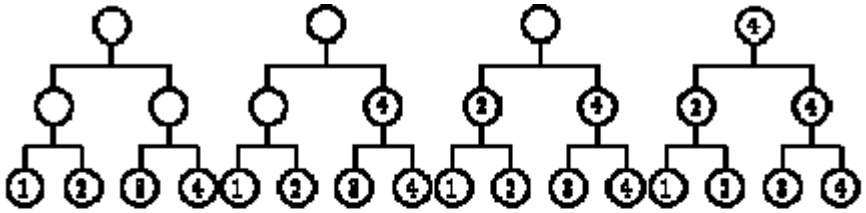
Modifying a Key

It takes O(log n) time to modify a selection tree in response to a change of a key in a leaf.



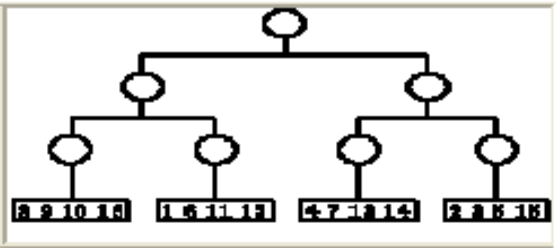
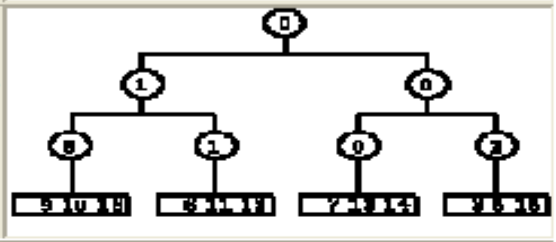
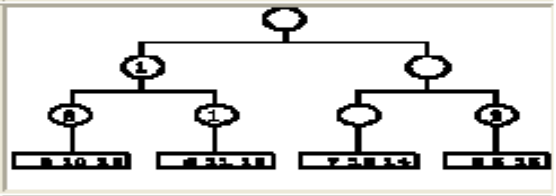
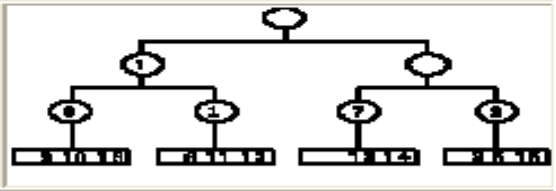
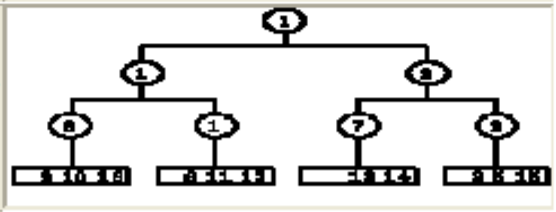
Initialization

The construction of a selection tree from scratch takes O(n) time by traversing it level-wise from bottom up.



Application: External Sort

Given a set of n values	16 9 10 8 6 11 12 1 4 7 14 13 2 15 5 3				n = 16
divide it into M chuncks,	16 9 10 8	6 11 12 1	4 7 14 13	2 15 5 3	M = 4
internally sort each chunk,	8 9 10 16	1 6 11 12	4 7 13 14	2 3 5 15	

construct complete binary tree of M leaves with the chunks attached to the leaves.	
Convert the tree into a selection tree with the keys being fed to the leaves from the chunks	
Remove the winner from the tree	
Feed to the empty leaf the next value from its corresponding chunk	
Adjust the selection tree to the change in the leaf	
Repeat the deletion sub process until all the values are consumed.	

The algorithm takes time to internally sort the elements of the chunks, O(M) to initialize the selection tree, and O(n log M) to perform the selection sort. For M « n the total time complexity is O(n log n).To reduce I/O operations, inputs from the chunks to the selection tree should go through buffers.

6.3 Forests

The default interdomain trust relationships are created by the system during domain controller creation.

The number of trust relationships that are required to connect *n* domains is *n* –1, whether the domains are linked in a single, contiguous parent-child hierarchy or they constitute two or more separate contiguous parent-child hierarchies.

When it is necessary for domains in the same organization to have different namespaces, create a separate tree for each namespace. In Windows 2000, the roots of trees are linked automatically by two-way, transitive trust relationships. Trees linked by trust relationships form a forest A single tree that is related to no other trees constitutes a forest of one tree.

The tree structures for the entire Windows 2000 forest are stored in Active Directory in the form of parent-child and tree-root relationships. These relationships are stored as trust account objects (class *trustedDomain*) in the System container within a specific domain directory partition. For each domain in a forest, information about its connection to a parent domain (or, in the case of a tree root, to another tree root domain) is added to the configuration data that is replicated to every domain in the forest. Therefore, every domain controller in the forest has knowledge of the tree structure for the entire forest, including knowledge of the links between trees.

6.4 Representation of Disjoint Sets

Set

In computer science, a **set** is an abstract data structure that can store certain values, without any particular order, and no repeated values. It is a computer implementation of the mathematical concept of a finite set.

Some set data structures are designed for **static sets** that do not change with time, and allow only query operations — such as checking whether a given value is in the set, or enumerating the values in some arbitrary order. Other variants, called **dynamic** or **mutable sets**, allow also the insertion and/or deletion of elements from the set.

A set can be implemented in many ways. For example, one can use a list, ignoring the order of the elements and taking care to avoid repeated values. Sets are often implemented using various flavors of trees, tries, hash tables, and more.

A set can be seen, and implemented, as a (partial) associative array, in which the value of each key-value pair has the unit type. In type theory, sets are generally identified with their indicator function: accordingly, a set of values of type may be denoted by or (Subtypes and subsets may be modeled by refinement types, and quotient sets may be replaced by setoids.)

Operations

Typical operations that may be provided by a static set structure *S* are

• element_of(*x*,*S*): checks whether the value *x* is in the set *S*.

• empty(*S*): checks whether the set *S* is empty.

• size(*S*): returns the number of elements in *S*.

• enumerate(*S*): yields the elements of *S* in some arbitrary order.

• pick(*S*): returns an arbitrary element of *S*.

• build(*x*1,*x*2,…,*xn*,): creates a set structure with values *x*1,*x*2,…,*xn*.

The enumerate operation may return a list of all the elements, or an iterator, a procedure object that returns one more value of *S* at each call.

Dynamic set structures typically add:

• create(*n*): creates a new set structure, initially empty but capable of holding up to *n* elements.

• add(*S*,*x*): adds the element *x* to *S*, if it is not there already.

• delete(*S*,*x*): removes the element *x* from *S*, if it is there.

• capacity(*S*): returns the maximum number of values that *S* can hold.

Some set structures may allow only some of these operations. The cost of each operation will depend on the implementation, and possibly also on the particular values stored in the set, and the order in which they are inserted. There are many other operations that can (in principle) be defined in terms of the above, such as:

• pop(*S*): returns an arbitrary element of *S*, deleting it from *S*.

• find(*S*, *P*): returns an element of *S* that satisfies a given predicate *P*.

• clear(*S*): delete all elements of *S*.

In particular, one may define the Boolean operations of set theory:

• union(*S*,*T*): returns the union of sets *S* and *T*.

• intersection(*S*,*T*): returns the intersection of sets *S* and *T*.

• difference(*S*,*T*): returns the difference of sets *S* and *T*.

• subset(*S*,*T*): a predicate that tests whether the set *S* is a subset of set *T*.

Other operations can be defined for sets with elements of a special type:

• sum(*S*): returns the sum of all elements of *S* (for some definition of "sum").

• nearest(*S*,*x*): returns the element of *S* that is closest in value to *x* (by some criterion).

In theory, many other abstract data structures can be viewed as set structures with additional operations and/or additional axioms imposed on the standard operations. For example, an abstract heap can be viewed as a set structure with a min(*S*) operation that returns the element of smallest value.

Implementations

Sets can be implemented using various data structures, which provide different time and space trade-offs for various operations. Some implementations are designed to improve the efficiency of very specialized operations, such as nearest or union. Implementations described as "general use" typically strive to optimize the element_of, add, and delete operation.

Sets are commonly implemented in the same way as associative arrays, namely, a self-balancing binary search tree for sorted sets (which has O(log n) for most operations), or a hash table for unsorted sets (which has O(1) average-case, but O(n) worst-case, for most operations). A sorted linear hash table may be used to provide deterministically ordered sets.

Other popular methods include arrays. In particular a subset of the integers 1..*n* can be implemented efficiently as an *n*-bit bit array, which also support very efficient union and intersection operations. A Bloom map implements a set probabilistically, using a very compact representation but risking a small chance of false positives on queries. The Boolean set operations can be implemented in terms of more elementary operations (pop, clear, and add), but specialized algorithms may yield lower asymptotic time bounds. If sets are implemented as sorted lists, for example, the naive algorithm for union(*S*,*T*) will take code proportional to the length *m* of *S* times the length *n* of *T*; whereas a variant of the list merging algorithm will do the job in time proportional to *m*+*n*. Moreover, there are specialized set data structures (such as the union-find data structure) that are optimized for one or more of these operations, at the expense of others.

6.5 Counting Binary Trees

Definition: A binary tree has a special vertex called its root. From this vertex at the top, the rest of the tree is drawn downward. Each vertex may have a left child and/or a right child.

Example. The number of binary trees with 1, 2, 3 vertices is:

Example. The number of binary trees with 4 vertices is:

Conjecture: The number of binary trees on n vertices is .

Proof: Every binary tree either:

! Has no vertices (x0) –or–

! Breaks down as one root vertex (x)

along with two binary trees beneath (B(x)2).

Therefore, the generating function for binary trees satisfies B(x) = 1 + xB(x)2. We conclude bn = 1

n+1#2n n $.

Another way: Find a recurrence for bn. Note:

b4 = b0b3 + b1b2 + b2b1 + b3b0.

In general, bn =%n−1 i=0 bibn−1−i .

Therefore, B(x) equals 1 + & n"1 '&n−1 i=0 bibn−1−I (xn = 1 + x & n"1 '&n−1 i=0

bibn−1−I (xn−1 = 1+x & k"0 '&k i=0 bibk−I (xk = 1+x '& k"0 bkxk ('& k"0 bkxk (= 1+xB(x)2.

6.6 The Graph Abstract Data Type.

A **graph** is an abstract data type that is meant to implement the graph and hypergraph concepts from mathematics.A graph data structure consists of a finite (and possibly mutable) set of ordered pairs, called **edges** or **arcs**, of certain entities called **nodes** or **vertices**. As in mathematics, an edge (*x*,*y*) is said to **point** or **go from** *x* **to** *y*. The nodes may be part of the graph structure, or may be external entities represented by integer indices or references. A graph data structure may also associate to each edge some **edge value**, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).

Algorithms

Graph algorithms are a significant field of interest within computer science. Typical higher-level operations associated with graphs are: finding a path between two nodes, like depth-first search and breadth-first search and finding the shortest path from one node to another, like Dijkstra's algorithm. A solution to finding the shortest path from each node to every other node also exists in the form of the Floyd–Warshall algorithm.A directed graph can be seen as a flow network, where each edge has a capacity and each edge receives a flow. The Ford–Fulkerson algorithm is used to find out the maximum flow from a source to a sink in a graph

Operations

The basic operations provided by a graph data structure *G* usually include:

- adjacent(*G*, *x*, *y*): tests whether there is an edge from node *x* to node *y*.
- neighbors(*G*, *x*): lists all nodes *y* such that there is an edge from *x* to *y*.
- add(*G*, *x*, *y*): adds to *G* the edge from *x* to *y*, if it is not there.

- delete(*G*, *x*, *y*): removes the edge from *x* to *y*, if it is there.
- get_node_value(*G*, *x*): returns the value associated with the node *x*.
- set_node_value(*G*, *x*, *a*): sets the value associated with the node *x* to *a*.

Structures that associate values to the edges usually also provide:

- get_edge_value(*G*, *x*, *y*): returns the value associated to the edge (*x*,*y*).
- set_edge_value(*G*, *x*, *y*, *v*): sets the value associated to the edge (*x*,*y*) to *v*.

UNIT - 7

PRIORITY QUEUES

7.1 Single- and Double-Ended Priority Queues

7.2 Leftist Trees

7.3 Binomial Heaps

7.4 Fibonacci Heaps

7.5 Pairing Heaps.

UNIT - 7

PRIORITY QUEUES

7.1 Single- and Double-Ended Priority Queues

Priority Queue:

Need for priority queue:

- In a multi user environment, the operating system scheduler must decide which of several processes to run only for a fixed period for time.
- For that we can use the algorithm of QUEUE, where Jobs are initially placed at the end of the queue.
- The scheduler will repeatedly take the first job on the queue, run it until either it finishes or its time limit is up, and placing it at the and of the queue if it doesn't finish.
- This strategy is generally not approximate, because very short jobs will soon to take a long time because of the wait involved to run.
- Generally, it is important that short jobs finish as fast as possible, so these jobs should have precedence over jobs that have already been running.
- Further more, some jobs that are not short are still very important and should also have precedence.
- This particular application seems to require a special kind of queue, known as aPRIORITY QUEUE.

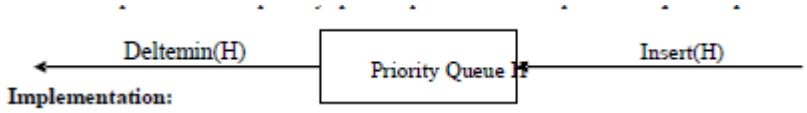
Priority Queue:

It is a collection of ordered elements that provides fast access to the minimum or maximum element.

Basic Operations performed by priority queue are:

1. Insert operation
2. Deletemin operation

- Insert operation is the equivalent of queue's ***Enqueue*** operation.
- Deletemin operation is the priority queue equivalent of the queue's ***Dequeue*** operation.



Implementation:

7.2 Leftist Trees

A (single-ended) priority queue is a data type supporting the following operations on an ordered set of values:

1) find the maximum value (FindMax);

2) delete the maximum value (DeleteMax);

3) add a new value x (Insert(x)).

Obviously, the priority queue can be redefined by substituting operations 1) and 2) with FindMin and DeleteMin, respectively. Several structures, some implicitly stored in an array and some using more complex data structures, have been presented for implementing this data type, including max heaps (or min-heaps)

Conceptually, a max-heap is a binary tree having the following properties:

a) heap-shape: all leaves lie on at most two adjacent levels, and the leaves on the last level occupy the leftmost positions; all other levels are complete.

b) max-ordering: the value stored at a node is greater than or equal to the values stored at its children. A max-heap of size n can be constructed in linear time and can be stored in an n-element array; hence it is referred to as an implicit data structure [g].

When a max-heap implements a priority queue, FindMax can be performed in constant time, while both DeleteMax and Insert(x) have logarithmic time. We shall consider a more powerful data type, the doubleended priority queue, which allows both FindMin and FindMax, as well as DeleteMin, DeleteMax, and Insert(x) operations. An important application of this data type is in external quicksort .

A traditional heap does not allow efficient implementation of all the above operations; for example, FindMin requires linear (instead of constant) time in a max-heap. One approach to overcoming this intrinsic limitation of heaps, is to place a max-heap “back-to-back” with a min-heap Definition

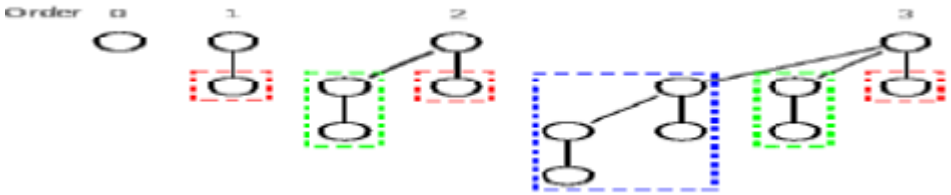
A **double-ended priority queue (DEPQ)** is a collection of zero or more elements. Each element has a priority or value. The operations performed on a double-ended priority queue are:

1. isEmpty() ... return true iff the DEPQ is empty
2. size() ... return the number of elements in the DEPQ
3. getMin() ... return element with minimum priority
4. getMax() ... return element with maximum priority
5. put(x) ... insert the element x into the DEPQ
6. removeMin() ... remove an element with minimum priority and return this element
7. removeMax() ... remove an element with maximum priority and return this element

7.3 Binomial Heaps

Binomial heap is a heap similar to a binary heap but also supports quickly merging two heaps. This is achieved by using a special tree structure. It is important as an implementation of the **mergeable heap** abstract data type (also called meldable heap), which is a priority queue supporting merge operation. A binomial heap is implemented as a collection of binomial trees (compare with a binary heap, which has a shape of a single binary tree). A **binomial tree** is defined recursively:

- A binomial tree of order 0 is a single node
- A binomial tree of order *k* has a root node whose children are roots of binomial trees of orders *k*−1, *k*−2, ..., 2, 1, 0 (in this order).



Binomial trees of order 0 to 3: Each tree has a root node with subtrees of all lower ordered binomial trees, which have been highlighted. For example, the order 3 binomial tree is connected to an order 2, 1, and 0(highlighted as blue, green and red respectively) binomial tree.

A binomial tree of order *k* has 2k nodes, height *k*.

Because of its unique structure, a binomial tree of order *k* can be constructed from two trees of order *k*−1trivially by attaching one of them as the leftmost child of root of the other one. This feature is central to the *merge* operation of a binomial heap, which is its major advantage over other conventional heaps

Structure of a binomial heap

A binomial heap is implemented as a set of binomial trees that satisfy the *binomial heap properties*:

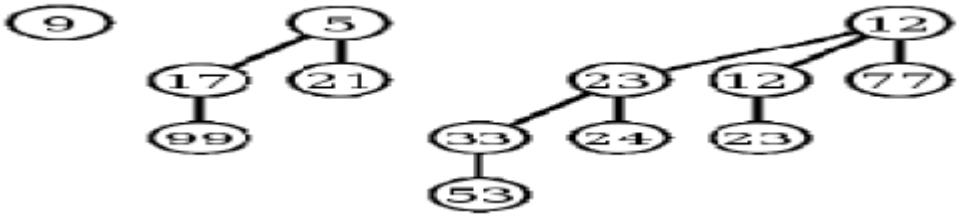
Each binomial tree in a heap obeys the *minimum-heap property*: the key of a node is greater than or equal to the key of its parent.

There can only be either *one* or *zero* binomial trees for each order, including zero order.

The first property ensures that the root of each binomial tree contains the smallest key in the tree, which applies to the entire heap.

The second property implies that a binomial heap with *n* nodes consists of at most log *n* + 1 binomial trees. In fact, the number and orders of these trees are uniquely determined by the number of nodes *n*: each

binomial tree corresponds to one digit in the binary representation of number *n*. For example number 13 is 1101 in binary, , and thus a binomial heap with 13 nodes will consist of three binomial trees of orders 3, 2, and 0 (see figure below).



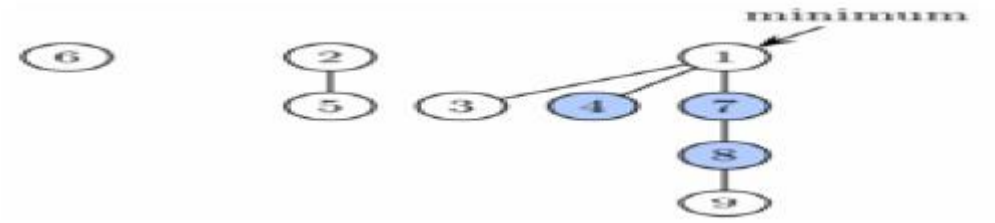
7.4 Fibonacci Heaps

A **Fibonacci heap** is a heap data structure consisting of a collection of trees. It has a better amortized running time than a binomial heap. Fibonacci heaps were developed by Michael L. Fredman and Robert E. Tarjan in 1984 and first published in a scientific journal in 1987. The name of Fibonacci heap comes from Fibonacci numbers which are used in the running time analysis.

Find-minimum is O(1) amortized time.Operations insert, decrease key, and merge (union) work in constant amortized time. Operations delete and delete minimum work in $O(\log n)$ amortized time. This means that starting from an empty data structure, any sequence of a operations from the first group and b operations from the second group would take $O(a + b \log n)$ time. In a binomial heap such a sequence of operations would take $O((a + b)\log (n))$ time. A Fibonacci heap is thus better than a binomial heap when b is asymptotically smaller than a.

Using Fibonacci heaps for priority queues improves the asymptotic running time of important algorithms, such as Dijkstra's algorithm for computing the shortest path between two nodes in a graph.

Structure



Fibonacci heap is a collection of trees satisfying the minimum-heap property, that is, the key of a child is always greater than or equal to the key of the parent. This implies that the minimum key is always at the root

of one of the trees. Compared with binomial heaps, the structure of a Fibonacci heap is more flexible. The trees do not have a prescribed shape and in the extreme case the heap can have every element in a separate tree. This flexibility allows some operations to be executed in a "lazy" manner, postponing the work for later operations. For example merging heaps is done simply by concatenating the two lists of trees, and operation *decrease key* sometimes cuts a node from its parent and forms a new tree. However at some point some order needs to be introduced to the heap to achieve the desired running time.

In particular, degrees of nodes (here degree means the number of children) are kept quite low: every node has degree at most $O(\log n)$ and the size of a subtree rooted in a node of degree k is at least $F_k + 2$, where F_k is the kth Fibonacci number. This is achieved by the rule that we can cut at most one child of each nonroot node. When a second child is cut, the node itself needs to be cut from its parent and becomes the root. As a result of a relaxed structure, some operations can take a long time while others are done very quickly. In the amortized running time analysis we pretend that very fast operations take a little bit longer than they actually do. This additional time is then later subtracted from the actual running time of slow operations. The amount of time saved for later use is measured at any given moment by a potential function. The potential of a Fibonacci heap is given by

Potential = $t + 2m$ where t is the number of trees in the Fibonacci heap, and m is the number of marked nodes. A node is marked if at least one of its children was cut since this node was made a child of another node (all roots are unmarked).

Thus, the root of each tree in a heap has one unit of time stored. This unit of time can be used later to link this tree with another tree at amortized time 0. Also, each marked node has two units of time stored. One can be used to cut the node from its parent. If this happens, the node becomes a root and the second unit of time will remain stored in it as in any other root.

Implementation of operations

To allow fast deletion and concatenation, the roots of all trees are linked using a circular, doubly linked list. The children of each node are also linked using such a list. For each node, we maintain its number of children and whether the node is marked. Moreover we maintain a pointer to the root containing the minimum key.

Operation **find minimum** is now trivial because we keep the pointer to the node containing it. It does not change the potential of the heap, therefore both actual and amortized cost is constant. As mentioned above,

merge is implemented simply by concatenating the lists of tree roots of the two heaps. This can be done in constant time and the potential does not change, leading again to constant amortized time. Operation **insert** works by creating a new heap with one element and doing merge. This takes constant time, and the potential increases by one, because the number of trees increases. The amortized cost is thus still constant.

7.5 Pairing Heaps.

Pairing heaps are a type of heap data structure with relatively simple implementation and excellent practical amortized performance. However, it has proven very difficult to determine the preciseasymptotic running time of pairing heaps.

Pairing heaps are heap ordered multiway trees. Describing the various heap operations is relatively simple (in the following we assume a min-heap):

- *find-min*: simply return the top element of the heap.
- *merge*: compare the two root elements, the smaller remains the root of the result, the larger element and its subtree is appended as a child of this root.
- *insert*: create a new heap for the inserted element and *merge* into the original heap.
- *decrease-key* (optional): remove the subtree rooted at the key to be decreased then *merge* it with the heap. *delete-min*: remove the root and *merge* its subtrees. Various strategies are employed.

The amortized time per *delete-min* is $O(\log n)$.The operations *find-min*, *merge*, and *insert* take $O(1)$ amortized time and *decrease-key* takes amortized time. Fredman proved that the amortized time per *decrease-key* is at least $\Omega(\log\log n)$. That is, they are less efficient than Fibonacci heaps, which perform *decrease-key* in $O(1)$ amortized time.

Implementation

A pairing heap is either an empty heap, or a pair consisting of a root element and a possibly empty list of pairing heaps. The heap ordering property requires that all the root elements of the subheaps in the list are not smaller that then root element of the heap. The following description assumes a purely functional heap that does not support the *decrease-key* operation.

type PairingHeap[Elem] = Empty | Heap(elem: Elem, subheaps: List[PairingHeap[Elem]])

Operations

find-min

The function *find-min* simply returns the root element of the heap:

function find-min(heap)

if heap == Empty

error

else

return heap.elem

merge:

Merging with an empty heap returns the other heap, otherwise a new heap is returned that has the minimum of the two root elements as its root element and just adds the heap with the larger root to the list of subheaps:

function merge(heap1, heap2)

if heap1 == Empty

return heap2

elsif heap2 == Empty

return heap1

elseif heap1.elem < heap2.elem

return Heap(heap1.elem, heap2 :: heap1.subheaps)

else

return Heap(heap2.elem, heap1 :: heap2.subheaps)

Insert:

The easiest way to insert an element into a heap is to merge the heap with a new heap containing just this element and an empty list of subheaps:

function insert(elem, heap)

return merge(Heap(elem, []), heap)

delete-min:

The only non-trivial fundamental operation is the deletion of the minimum element from the heap. The standard strategy first merges the subheaps in pairs (this is the step that gave this datastructure its name) from left to right and then merges the resulting list of heaps from right to left:

fuction delete-min(heap)

if heap == Empty

error

elsif length(heap.subheaps) == 0

return Empty

elsif length(heap.subheaps) == 1

return heap.subheaps[0]

else

return merge-pairs(heap.subheaps)

This uses the auxiliary function *merge-pairs*:

function merge-pairs(l)

if length(l) == 0

return Empty

elsif length(l) == 1

return l[0]

else

return merge(merge(l[0], l[1]), merge-pairs(l[2..]))

UNIT - 8

EFFICIENT BINARY SEARCH TREES

8.1 Optimal Binary Search Trees

8.2 AVL Trees

8.3 Red-Black Trees

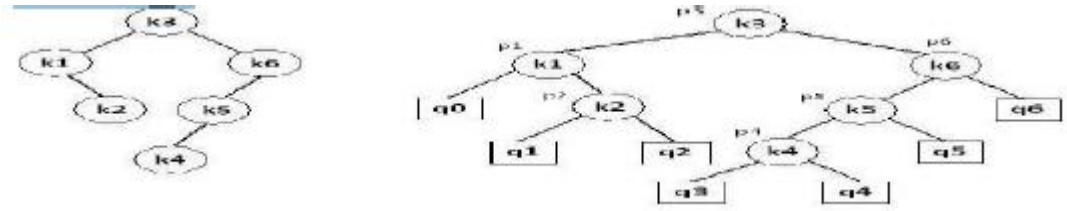
8.4 Splay Trees

UNIT - 8

EFFICIENT BINARY SEARCH TREES

8.1 Optimal Binary Search Trees

An optimal binary search tree is a binary search tree for which the nodes are arranged on levels such that the tree cost is minimum. For the purpose of a better presentation of optimal binary search trees, we will consider "extended binary search trees", which have the keys stored at their internal nodes. Suppose "n" keys k1, k2, … , k n are stored at the internal nodes of a binary search tree. It is assumed that the keys are given in sorted order, so that k1< k2 < … < kn. An extended binary search tree is obtained from the binary search tree by adding successor nodes to each of its terminal nodes as indicated in the following figure by squares:



In the extended tree: the squares represent terminal nodes. These terminal nodes represent unsuccessful searches of the tree for key values. The searches did not end successfully, that is, because they represent key values that are not actually stored in the tree; the round nodes represent internal nodes; these are the actual keys stored in the tree; assuming that the relative frequency with which each key value is accessed is known, weights can be assigned to each node of the extended tree (p1 … p6). They represent the relative frequencies of searches terminating at each node, that is, they mark the successful searches.

If the user searches a particular key in the tree, 2 cases can occur:

1 – The key is found, so the corresponding weight 'p' is incremented;

2 – The key is not found, so the corresponding 'q' value is incremented.

GENERALIZATION: the terminal node in the extended tree that is the left successor of k1 can be interpreted as representing all key values that are not stored and are less than k1. Similarly, the terminal node in the extended tree that is the right successor of kn, represents all key values not stored in the tree that are greater than kn. The terminal node that is successed between ki and ki-1 in an inorder traversal represents all key values not stored that lie between ki and ki - 1.

An obvious way to find an optimal binary search tree is to generate each possible binary search tree for the keys, calculate the weighted path length, and keep that tree with the smallest weighted path length.

This search through all possible solutions is not feasible, since the number of such trees grows exponentially with “n”.

An alternative would be a recursive algorithm. Consider the characteristics of any optimal tree. Of course it has a root and two subtrees. Both subtrees must themselves be optimal binary search trees with respect to their keys and weights. First, any subtree of any binary search tree must be a binary search tree.

Second, the subtrees must also be optimal. Since there are “n” possible keys as candidates for the root of the optimal tree, the recursive solution must try them all. For each candidate key as root, all keys less than

8.2 AVL Trees

8.3 Red-Black Trees

Properties

A binary search tree in which

The root is colored black

All the paths from the root to the leaves agree on the number of black nodes

No path from the root to a leaf may contain two consecutive nodes colored red

Empty subtrees of a node are treated as subtrees with roots of black color.

The relation $n > 2h/2 - 1$ implies the bound $h < 2 \log 2(n + 1)$.

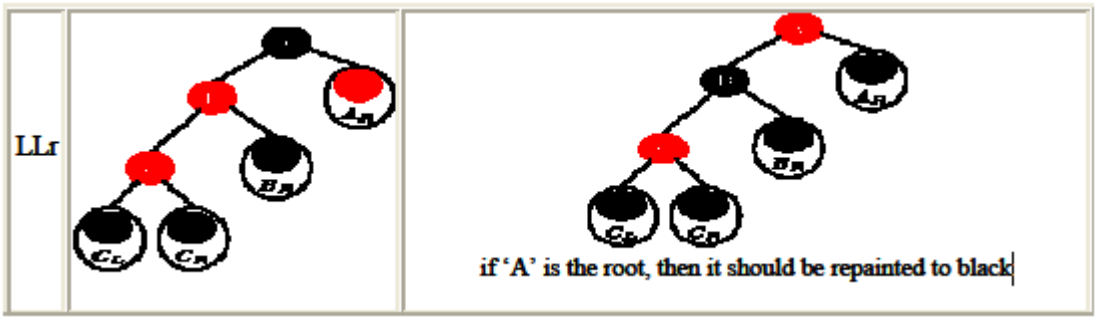
Insertions

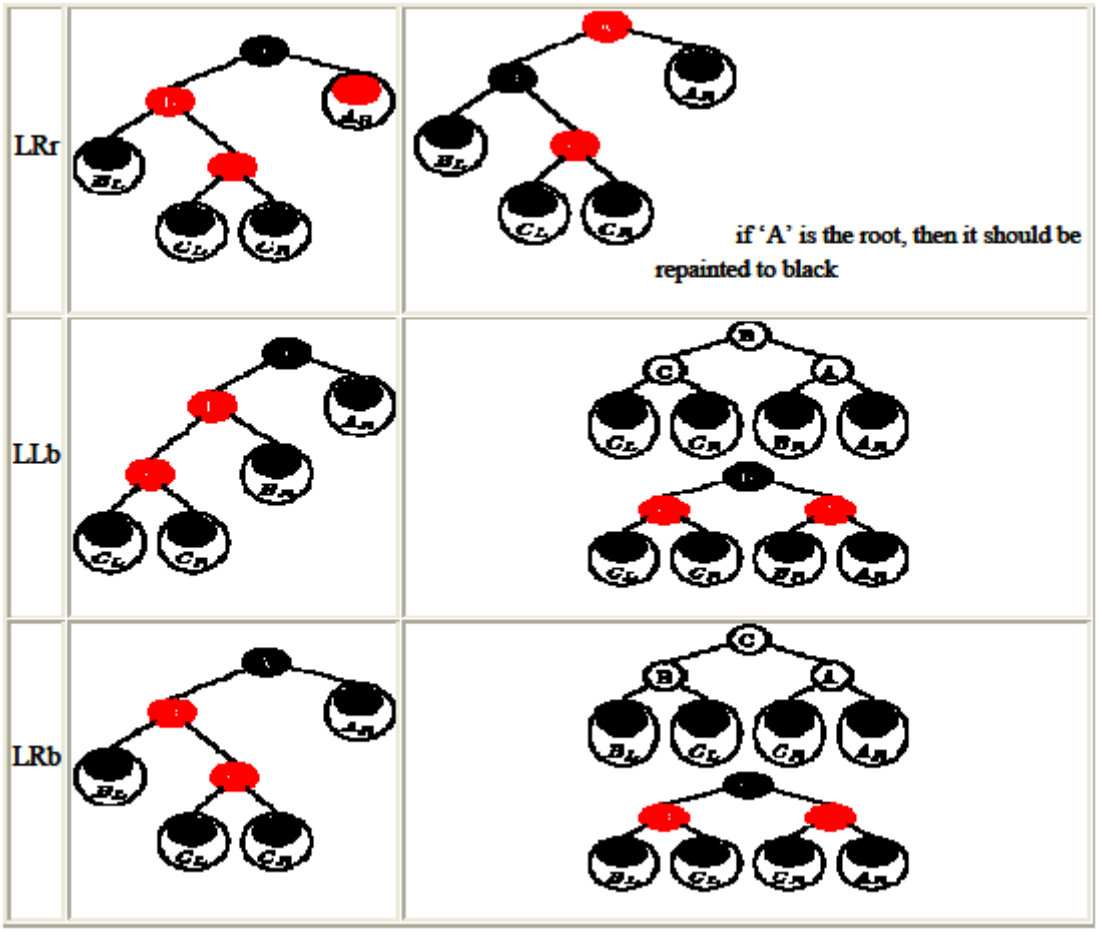
- Insert the new node the way it is done in binary search trees
- Color the node red
- If a discrepancy arises for the red-black tree, fix the tree according to the type of discrepancy.

A discrepancy can result from a parent and a child both having a red color. The type of discrepancy is determined by the location of the node with respect to its grand parent, and the color of the sibling of the parent.

Discrepancies in which the sibling is red, are fixed by changes in color. Discrepancies in which the siblings are black, are fixed through AVL-like rotations.

Changes in color may propagate the problem up toward the root. On the other hand, at most one rotation is sufficient for fixing a discrepancy.





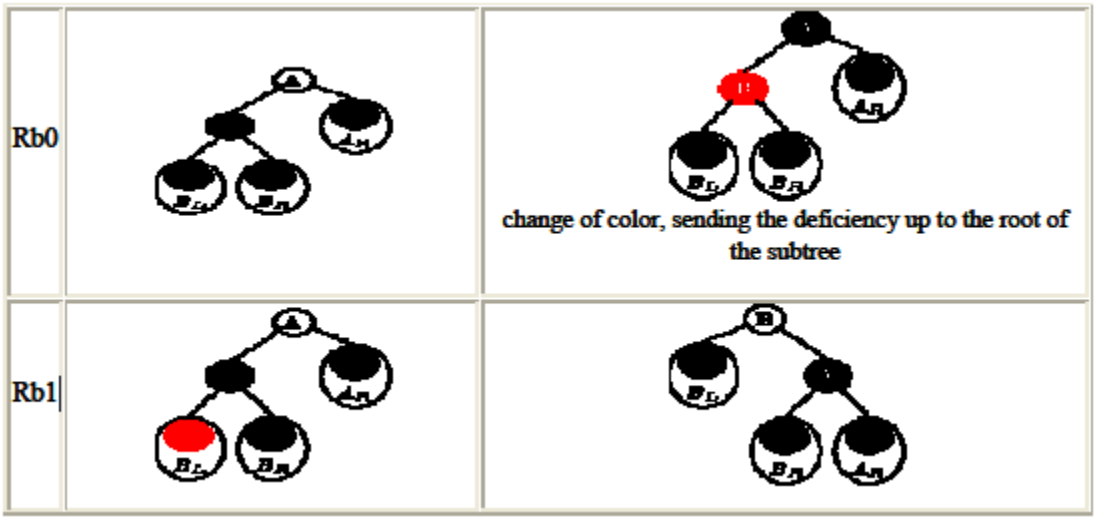
Deletions

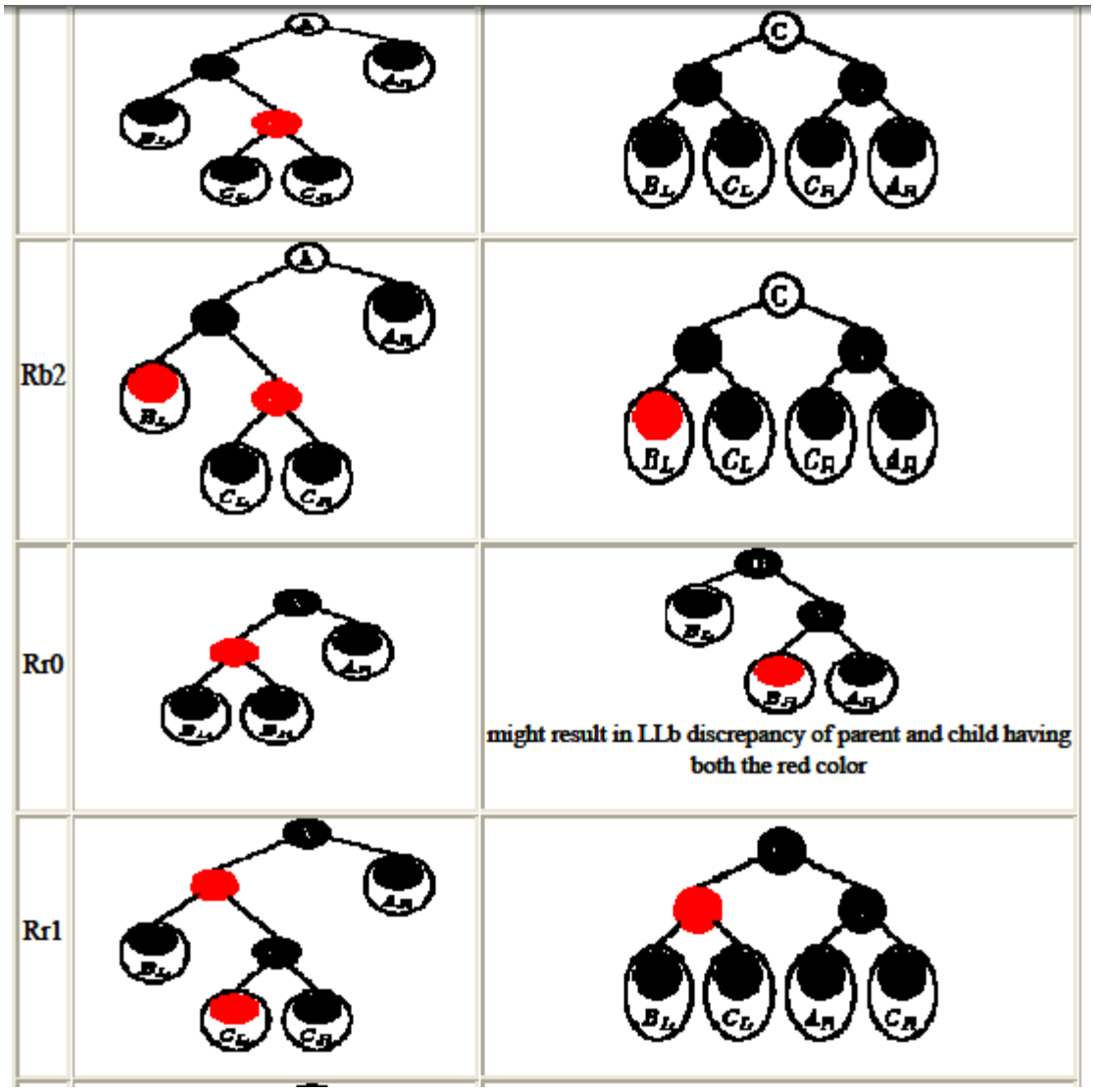
- Delete a key, and a node, the way it is done in binary search trees.

- A node to be deleted will have at most one child. If the deleted node is red, the tree is still a redblack
- tree. If the deleted node has a red child, repaint the child to black.
- If a discrepancy arises for the red-black tree, fix the tree according to the type of discrepancy.
- A discrepancy can result only from a loss of a black node.

Let A denote the lowest node with unbalanced subtrees. The type of discrepancy is determined by the location of the deleted node (**R**ight or **L**eft), the color of the sibling (**b**lack or **r**ed), the number of red children in the case of the black siblings, and and the number of grand-children in the case of red siblings.

In the case of discrepancies which result from the addition of nodes, the correction mechanism may propagate the color problem (i.e., parent and child painted red) up toward the root, and stopped on the way by a single rotation. Here, in the case of discrepancies which result from the deletion of nodes, the discrepancy of a missing black node may propagate toward the root, and stopped on the way by an application of an appropriate rotation.





8.4 Splay Trees

A **splay tree** is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log n) amortized time. For many sequences of non-random operations, splay trees perform better than

other search trees, even when the specific pattern of the sequence is unknown. The splay tree was invented by Daniel Dominic Sleator and Robert Endre Tarjan in 1985.

All normal operations on a binary search tree are combined with one basic operation, called *splaying*. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top. Alternatively, a top-down algorithm can combine the search and the tree reorganization into a single phase.

Splaying

When a node *x* is accessed, a splay operation is performed on *x* to move it to the root. To perform a splay operation we carry out a sequence of *splay steps*, each of which moves *x* closer to the root. By performing a splay operation on the node of interest after every access, the recently-accessed nodes are kept near the root and the tree remains roughly balanced, so that we achieve the desired amortized time bounds.

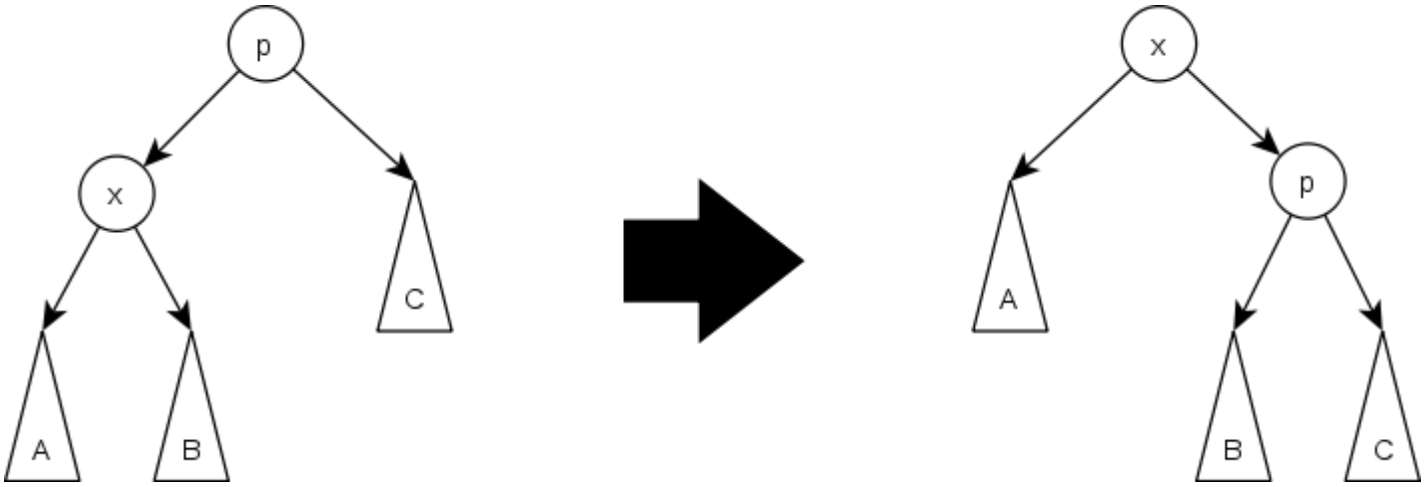
Each particular step depends on three factors:

- Whether *x* is the left or right child of its parent node, *p*,
- whether *p* is the root or not, and if not
- whether *p* is the left or right child of *its* parent, *g* (the *grandparent* of x).

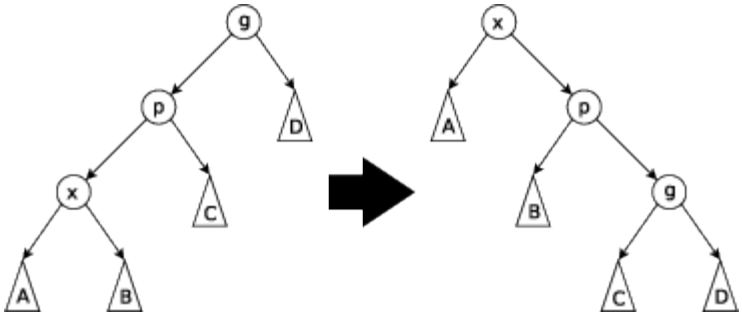
It is important to remember to set *gg* (the *great-grandparent* of x) to now point to x after any splay operation. If *gg* is null, then x obviously is now the root and must be updated as such.

There are three types of splay steps, each of which has a left- and right-handed case. For the sake of brevity, only one of these two is shown for each type. These three types are:

Zig Step: This step is done when *p* is the root. The tree is rotated on the edge between *x* and *p*. Zig steps exist to deal with the parity issue and will be done only as the last step in a splay operation and only when *x* has odd depth at the beginning of the operation.



Zig-zig Step: This step is done when *p* is not the root and *x* and *p* are either both right children or are both left children. The picture below shows the case where *x* and*p* are both left children. The tree is rotated on the edge joining *p* with *its* parent *g*, then rotated on the edge joining *x* with *p*. Note that zig-zig steps are the only thing that differentiate splay trees from the *rotate to root* method introduced by Allen and Munro[3] prior to the introduction of splay trees.



Zig-zag Step: This step is done when *p* is not the root and *x* is a right child and *p* is a left child or vice versa. The tree is rotated on the edge between *p* and x, and then rotated on the resulting edge between *x* and g.