Mount Kenya University

P.O. Box 342-01000 Thika

Email: info@mku.ac.ke

Web: www.mku.ac.ke

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE CODE: BIT22203

COURSE TITLE: DATA STRUCTURES AND ALGORITHMS

Contents

COURSE OUTLINE

BIT22203 DATA STRUCTURES AND ALGORITHMS

Purpose of the course : To present fundamental methodology and concepts of writing and applying algorithms to various data structures.

WEEK ONE & TWO

1. Data Structures and their applications

- One-dimensional array data structures
- Two-dimensional data structures
- Stack data structure
- Queue data structure
- Linked list data structure
- Tree data structure
- Graph data structure

WEEK THREE & FOUR

2. Operations on data structures

- Operations on Stack data structure
- Operations on Queue data structures
- Operations on Linked list data structure
- Operations on tree data structures

WEEK FIVE, SIX & SEVEN

3. Algorithms

- Algorithms for manipulating array data structures
- Algorithms for manipulating stack data structures
- Algorithms for manipulating queue data structures
- Algorithms for manipulating linked list
- Algorithm for manipulating trees
- Algorithms for manipulating postfix and infix expressions

WEEK EIGHT

4. Recursive functions

- Factorial functions
- Power functions
- Fibonacci functions

WEEK NINE

5. Postfix and infix expressions

WEEK TEN

6 .Traversal of the tree

- Pre-order traversal
- In-order traversal
- Post-order traversal

WEEK ELEVEN

7. Search

- Linear search
- Binary search

WEEK TWELVE, THIRTEEN & FOURTEEN

8. Sorting

- Bubble sort algorithm
- Selection sort algorithm
- Insertion sort algorithm
- Quick sort algorithm

Assessments

- Continuous Assessment Tests (CATs) (30%)
- End of semester examination (70%)

Total = 100%

Required text books

Salamis S, Data structures, Algorithms and application in Java, McGraw Hill

Banachowski I et al, Analysis of algorithms and Data structures, Addison wiley

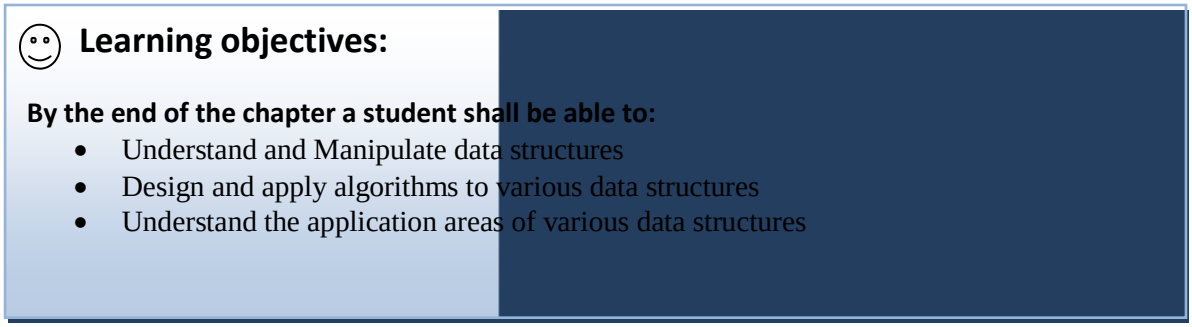
Text books for further reading

MKlaus W., Algorithms, data structures, programmes, New Delhi, MCGraw Hill

Compiled by : Joshua Agola

CHAPTER ONE

DATA STRUCTURES



1.1 ARRAY DATA STRUCTURES

What is an array?

Array is a very basic data structure provided by every programming language. Let's talk about an example scenario where we need to store ten employees' data in our C/C++ program including name, age and salary. One of the solutions is to declare ten different variables to store employee

name and ten more to store age and so on. Also you will need some sort of mechanism to get information about an employee, search employee records and sort them. To solve these types of problem C/C++ provide a mechanism called Arrays.

Definition

An array is simply a number of memory locations, each of which can store an item of data of the same data type and which are all referenced through the same variable name. Array may be defined abstractly as finite order set of homogeneous elements. So we can say that there are finite numbers of elements in an array and all the elements are of same data type. Also array elements are ordered i.e. we can access a specific array element by an index.

1.1.1 How to declare one-dimensional array?

The general form of declaring a simple (one dimensional) array is

array_type variable_name[array_size];

in your C/C++ program you can declare an array like

int Age[10];

Here array_type declares base type of array which is the type of each element in array. In our example array_type is int and its name is Age. Size of the array is defined by array_size i.e. 10. We can access array elements by index, and first item in array is at index 0. First element of array is called lower bound and its always 0. Highest element in array is called upper bound.

In C programming language upper and lower bounds cannot be changed during the execution of the program, so array length can be set only when the program in written.

Age 0	Age 1	Age 2	Age 3	Age 4	Age 5	Age 6	Age 7	Age 8	Age 9
30	32	54	32	26	29	23	43	34	5

Array has 10 elements

Note: One good practice is to declare array length as a constant identifier. This will minimise the required work to change the array size during program development.

Considering the array we declared above we can declare it like

```
#define NUM_EMPLOYEE 10
```

```
int Age[NUM_EMPLOYEE];
```

How to initialise an array?

Initialisation of array is very simple in c programming. There are two ways you can initialise arrays.

Declare and initialise array in one statement.

Declare and initialise array separately.

Look at the following C code which demonstrates the declaration and initialisation of an array.

```
int Age [5] = {30, 22, 33, 44, 25};
```

```
int  Age  [5];

Age  [0]=30;

Age  [1]=22;

Age  [2]=33;

Age  [3]=44;

Age [4]=25;
```

Array can also be initialised in a ways that array size is omitted, in such case compiler automatically allocates memory to array.

```
int Age [ ] = {30, 22, 33, 44, 25};
```

Let's write a simple program that uses arrays to print out number of employees having salary more than 3000.

Array in C Programming

```
#include <windows.h>

#include <stdio.h>

#include <stdlib.h>

#define NUM_EMPLOYEE 10
```

```
int main(int argc, char *argv[]){
	int Salary[NUM_EMPLOYEE], lCount=0,gCount=0,i=0;
	printf("Enter employee salary (Max 10)\n ");
	for (i=0; i<NUM_EMPLOYEE; i++){
	printf("\nEnter employee salary: %d - ",i+1);
	scanf("%d",&Salary[i]);
	}

	for(i=0; i<NUM_EMPLOYEE; i++){
	if(Salary[i]<3000)
		lCount++;
	else
		gCount++;
	}

	printf("\nThere are {%d} employee with salary more than 3000\n",gCount);
	printf("There are {%d} employee with salary less than 3000\n",lCount);
	printf("Press ENTER to continue...\n");
	getchar();
	return 0;
}
```

Array in C++ Programming

```
#include <cstdlib>
#include <iostream>
#define NUM_EMPLOYEE 10
using namespace std;
int main(int argc, char *argv[]){
	int Salary[NUM_EMPLOYEE], lCount=0,gCount=0,i=0;
	cout << "Enter employee salary (Max 10) " << endl;
	for (i=0; i<NUM_EMPLOYEE; i++){
	cout << "Enter employee salary: - " << i+1 << endl;
	cin >> Salary[i];
	}
	for(i=0; i<NUM_EMPLOYEE; i++){
	if(Salary[i]<3000)
		lCount++;
	else
		gCount++;
	}

	cout << "There are " << gCount << " employee with salary more than 3000" << endl
	<< "There are " << lCount << " employee with salary less than 3000" << endl;

	system("PAUSE");
	return EXIT_SUCCESS;
```

}

1.1.2 How to declare and initialize multi-dimensional arrays?

Often there is need to manipulate tabular data or matrices. For example if employee salary is increased by 20% and you are required to store both the salaries in your program. Then you will need to store this information into a two dimensional arrays. C/C++ gives you the ability to have arrays of any dimension.

Multi dimension arrays

Consider the example above, you have to store, previous salary, present salary and amount of increment. In that case you will need to store this information in three dimensional arrays.

First I will show you how to declare a two dimensional array and initialise it. Then write a complete program to use multidimensional arrays.

```
int Salary[10][2];
```

This defines an array containing 10 elements of type int. Each of these elements itself is an array of two integers. So to keep track of each element of this array is we have to use two indices. One is to keep track of row and other is to keep track of column.

Elements of multidimensional arrays

Here is a graphical view of multidimensional array that we use to store salary and increment on salary. First column stores the salary element of the array and second column stores increment on salary. We could add another column to store the new salary which adds the increment to the salary.

	Column 0 – Salary	Column 1 – Increment
Row 0		
Row 2		
Row 3		

Row 4		
Row 5		
Row 6		
Row 7		
Row 8		
Row 9		
Row 10		

Initialising multidimensional arrays

Multidimensional arrays can also be initialised in two ways just like one dimensional array. Two braces are used to surround the row element of arrays.

If you are initialising more than one dimension then you will have to use as many braces as the dimensions of the array are.

```
int Salary [5][2] = {
        {2300, 460},
        {3400, 680},
        {3200, 640},
        {1200, 240},
        {3450, 690}
};
int Salary [5][2] ={0};  //This will  initialise all the array elements to 0
int Salary [5][2];
Salary [0][0]=2300;
Salary [1][0]=3400;
```

Salary [2][0]=3200;

Salary [3][0]=1200;

Salary [4][0]=3450;

Salary [0][1]=460;

Salary [1][1]=680;

Salary [2][1]=640;

Salary [3][1]=240;

Salary [4][1]=690;

Here is a complete program written in both C and C++ to demonstrate the use of multidimensional arrays.

Demonstration of two dimension arrays

The code below demonstrates two dimension arrays. It uses the same example of employee salary to increment it by 20% and adds it to actual salary then print current salary, increment and new salary.

Two dimensional Array in C Programming

```
#include <windows.h>

#include <stdio.h>

#include <stdlib.h>

#define NUM_EMPLOYEE 10

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

  //initialise Salary of each employee

  int Salary[NUM_EMPLOYEE][2]={

          {2300,0},

          {3400,0},
```

```
        {3200,0},
        {1200,0},
        {3450,0},
        {3800,0},
        {3900,0},
        {2680,0},
        {3340,0},
        {3000,0}
 };
 int lCount=0,gCount=0,i=0;
 for(i=0; i<NUM_EMPLOYEE; i++){
        Salary[i][1] = ((Salary[i][0]*20)/100);
 }
 printf("Initial Salary + Increment = Total Salary\n");
 for (i=0; i<NUM_EMPLOYEE; i++){
        printf("%d\t\t%d\t\t%d\n",Salary[i][0],Salary[i][1],Salary[i][0]+Salary[i][1]);
 }
 printf("Press ENTER to continue...\n");
 getchar();
 return 0;
}
```

Two dimensional array in C++ Programming

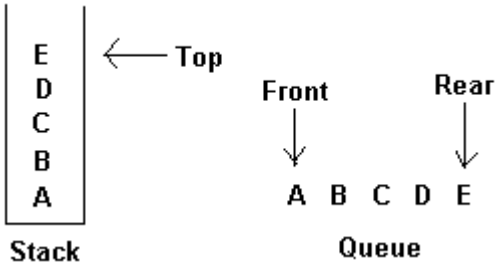
```
#include <cstdlib>
```

```
#include <iostream>

#define NUM_EMPLOYEE 10

using namespace std;

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

  //initialise Salary of each employee

  int Salary[NUM_EMPLOYEE][2]={

        {2300,0},

        {3400,0},

        {3200,0},

        {1200,0},

        {3450,0},
```

1.2 STACK DATA STRUCTURE

Two of the more common data objects found in computer algorithms are stacks and queues. Both of these objects are special cases of the more general data object, an ordered list.

A *stack* is an ordered list in which all insertions and deletions are made at one end, called the *top*. A *queue* is an ordered list in which all insertions take place at one end, the *rear*, while all deletions take place at the other end, the *front*. Given a stack S=(a[1],a[2],.....a[n]) then we say that a1 is the bottommost element and element a[i]) is on top of element a[i-1], 1<i<=n. When viewed as a queue with a[n] as the rear element one says that a[i+1] is behind a[i], 1<i<=n.



The restrictions on a stack imply that if the elements A,B,C,D,E are added to the stack, n that order, then th efirst element to be removed/deleted must be E. Equivalently we say that the last element to be inserted into the stack will be the first to be removed. For this reason stacks are sometimes referred to as Last In First Out (LIFO) lists.

Is a data structure that utilizes the concept of last in First Out (LIFO)

All the operations take place at the top

Areas where stack can be used include:

1. Plates in a cafeteria
2. Mathematical evaluations (postfix, infix, prefix)
3. Number conversion
4. Program execution

1.2.1 Operations of a stack

1. Dynamically initialize a stack (create)

By assigning a stack structure with
Top = 0

2. Test whether the stack is full

If the top = =max size of stack

3. Test whether the stack is empty

If the top = = 0

4. Pushing items into the stack

1) Increment top by 1
2) Insert the item

5. Poping items from the stack(Remove the item)

Decrease top by 1

1.2.2 The stack class

```
Class stack
{
Int top;
Int stackarray[50];
Public:
Stack( );
Int emptystack( );
Int fullstack( );
Void push(int item);
Void pop(int &item);
};
```

Example1

Using stack structure write a program for displaying numbers in the reverse order

Solution

```
#Include < iostream.h>

Const int maxsize = 20;

{

Int top;

Int stackarray[maxsize];

Public:

Int emptystack( );

Int fullstack( );

Void push(int item);

Void pop(int & item);

};

Stack : : stack( )

{

Top = 0;

}

Int stack : : emptystack( )

{

Return top = = 0;

}

Int stack : : fullstack( )

{

Return top = = maxsize;

}
```

```
Void stack : : push(int item)
{
Top + + ;
Stack away (top) = item;
}
Void stack : : Pop (int &item)
{
Item = stackarray[top]
top - -;
}
void main( )
{
Stack S;
Int i,n,x;
<out << “\n how many numbers”;
cin>>n;
cout << “\n Enter numbers in”;
For (i=1;i < = n; i++)
{ cin>>x;
s.push(x);
}
While (! S . emptystack( ))
```

{

S . pop (x);

cout <<x<<”\n”;

}

}

1.2.3 Application of the stack

Example: Converting Numbers from Base 10 to any other given base

Algorithm

1) Request for the number
2) Request for the base to convert to
3) While number >0

   i)Compute remainder
   ii)Push remainder into the stack
   iii)Compute next number (the quotient become the next number)
4) Display the content of stack((pop)

Example 2

Using stack, write a program for converting a number from base 10 to any other base (1-9)

Solution

include <iostream.h>

Const int max size = 50;

{

class stack

Int top;

Int stackarray(maxsize)

```
Public:
Stack ( );
Int emptystack( );
Int fullstack( );
Void push(int item);
};
{
Top = 0;
}
int stack : : emptystack ( )
{
Return top = = 0;
}
Int stack: : fullstack ( )
{return top = = max size;
}
Void stock : : push (int item)
{
Top ++;
Stakarray (top) = item;
}
Void stock : : pop (int pitem)
{
Item = stackrray (top);
Top = - ;
}
Void main( )
```

```
{
Stalk s ;
Int n ; // number
Int btest ; // base to convert to
Int remainder; // remainder
cout << "\n Enter base to convert to "'.
cin >>btest:
While (n>0)
{
Remainder= n % btest;
s.push (remainder);
n=n /btest;
}
While (! S.empty stack (j )
{
S.Pop (remainder);
Count << remainder << " ";
}
```

1.3 QUEUE DATA STRUCTURE

Queues are data structures that, like the stack, have restrictions on where you can add and remove elements. To understand a queue, think of a cafeteria line: the person at the front is served first, and people are added to the line at the back. Thus, the first person in line is served first, and the last person is served last. This can be abbreviated to **F**irst **I**n, **F**irst **O**ut (**FIFO**).

The cafeteria line is one type of queue. Queues are often used in programming networks, operating systems, and other situations in which many different processes must share resources such as CPU time.

Queue is a data structure that utililizes the concept of first-in-First Out (FIFO)

Inserting takes place at the rear and deleting takes place at the front

In a circular queue we need a counter that keep track of the the of element in a queue.

We need a generalised approach to compute the rear and the front position;

Rear =(rear +1)% maxsize,

Front = (front + 1) % maxsize

When inserting for the very first time, we need to adjust in position of front from zero to 1.

1.3.1 QUEUE Abstract Data Type

Const int maxsize =50;

Clas queue

```
{
Int  rear,front,count;
Int queuearry [maxisize];
Public:
Queue ( );
int fullqueue ( );
Void insertqueue (int item)
Void deletequeue (int item);
};
Queue : : Queue ( )
{
rear = 0;
front = 0;
count = 0;
}
In't Queue : : emptyqueue ( )
{
Return count count = =0;
}
Int Queue : : full quarter ( )
{
Return count = = maxsize;
}
Void Queue : : interqueue (int item)
{
If (count = = 0)
Front + +;
```

```
Rear = (rear + 1) % maxsize;
Queuearray [near ] = item;
Count + +;
}
Void Queue: : deletequeue (int titem)
{
Item = queuearray [front];
Front = (front + 1)% maxsize;
Count + - - ;
}
```

Example 1

Using Queue structure, write a program for displaying numbers in the same order of insetion

Solution

```
Include <io stream.h>
Void main ( )
{
Queue q ;
Int I,n,x;
Count<< "\n Enter the Number \n";
For (l=I ; i<=n ; l + + )
{
C;>>x ;
Q .insert queue (x) ;
}
Cout < < "\n show number \n";
```

```
While (q. ! emptyqueue ( ) )
{
Q . delete queue (x);
Count < < x < < "\n";
}
}
```

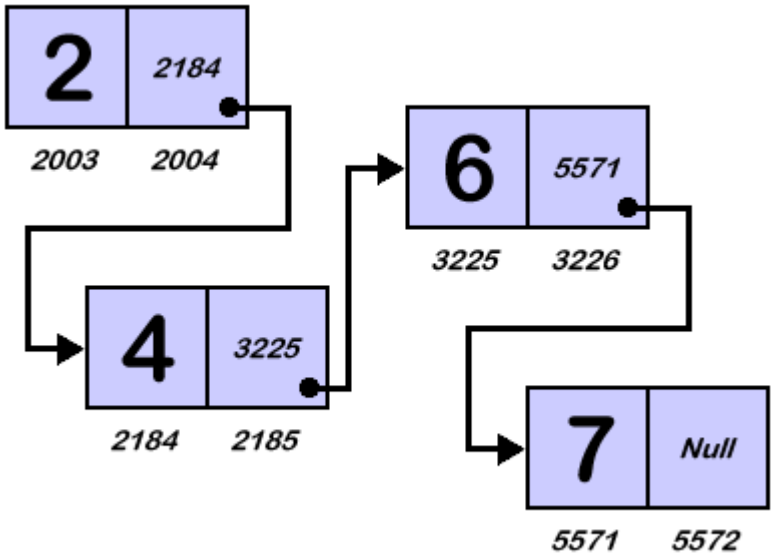
1.4 LINKED LIST DATA STRUCTURE

What is Linked List?

A linked list is a data structure that consists of a sequence of data records such that in each record there is a field that contains a reference (i.e., a link) to the next record in the sequence.

The linked list is a useful data structure that can dynamically grow according to data storage requirements. This is done by viewing data as consisting of a unit of data and a link to more units of data. Linked list is useful in the implementation of dynamic arrays, stacks, strings and sets. The link list is the basic ADT in some languages, for example, LISP.

Linked lists are most useful in environments with dynamic memory allocation. With dynamic memory allocation dynamic arrays can grow and shrink with less cost than in a static memory allocation environment. Linked lists are also useful to manage dynamic memory environments. Dramatically a linear linked list can be viewed as follows:



Each data element has an associated link to the next item in the list. The last item in the list has no link. The first element of the list is called the head , the last element is called the tail.

Linked lists can be implemented in most languages. Languages such as Lisp and Scheme have the data structure built in, along with operations to access the linked list. Procedural languages, such as C, or object-oriented languages, such as C++ and Java, typically rely on mutable references to create linked lists.

1.4.1 Pros and Cons of Linked Lists

Linked lists overcomes all the major limitations of arrays such as:

- The size of array is fixed. Though it can be deferred until the array is created at runtime, but after that it remains fixed.
- If you allocate a large space for arrays, the most of the space is really wasted. In case of small arrays declared, the code breaks in instances when more data elements are used than array size.
- Inserting new elements in the front is potentially expensive because existing elements need to be shifted over to make room

Linked lists have their own strengths and weakness, but they happen to be strong where arrays are weak. The array's features all follow from its strategy of allocating the memory for all its elements in one block of memory. Linked lists use an entirely different strategy. Linked lists allocate memory for each element separately and only when necessary. [*Linked List Basic* by By Nick Parlante]

A user does not have to declare the size of the linked list at the beginning of the program – you can add and remove objects from the list by merely unlinking them. In addition you can sort the members of a linked list – without actually moving data objects around – by changing the links.

The cost of flexibility in linked list is speed of access. You can't just reach in and grab the tenth element, for example, like you would in case of an array. Instead, you have to start at the beginning of the list and link ten times from one object to the next.

1.4.2 What is the good and bad thing about linked list?

The good thing about linked list is its runtime expandability. Unlike arrays, linked lists are much flexible in dynamically allocating space for data.

The bad thing about linked lists is its difficulty in accessing an object at random – you can't just reach and grab the element.

1.4.3 Types of linked lists

Linearly linked lists

In linearly linked lists the last node of a list contains a null reference to indicate it as tail or end of list. This type of lists are also called open linked list.

Circularly linked lists

In circular linked list the last node of a list contains a reference to the first node of the list. These lists are also called circular lists.

Singly Linked Lists

In singly-linked lists, each node contain a link to the next node.

Doubly linked lists

In doubly linked lists, each node contain a link to the next node and yet another to link to previous node. The two links may be called forwards and backwards or next and previous.

Multiply-linked Lists

In multiply-linked list, each node contains two or more link fields, each field being used to connect the same set of data records in a different order.

- A linked list is a structure consisting of nodes

- A node is a structure with two parts. One part stores the addresses of the next node.
- In a linked list they also have a special node referred to as the head. This node has only the information as to where the link structure starts.
- The last node in a linked list always point to NULL (Zero i.e. nothing)

1.4.4 How to insert a node at the beginning of a linked list

Steps

- Generate a new node
- Add information
- Establish position where to insert
- Le t the link of the node at the position to insert points to the position of the new node
- Let the link of the new node points to the position where the node at the insertion points was pointing.

1.4.5 How to insert a node at the end of a linked list

Steps

- Generate a new node
- Add information
- Let the last node parts to position of the new node.
- Let the link of the new node points to NULL

1.4.6 Deleting first node in a linked list

Steps

- Let pointer points to the first node (position of the head)Let the points to the link of the first node
- Delete the node (free space)

1.4.7 Deleting a node that is in between the chain

Steps

- Note in position of the node to delete
- Let the link of the previous node points to the link of the previous node points to the link of the node to be deleted
- Free space

1.4.8 Deleting last node in the chain

Steps

- Let the link of the previous node points to null
- Delete the node (free space)

1.4.9 Example: Using the linked list concept, write a program for manipulating a stack

```
#include <;ostream.h>
Class linkedstack
{
Private:
Structure linkedstacknode*link;
Linkedstacknode (int &item,Linked stackNode*head = NULL)
{
Data =item;
Link = head;
}
}
Linked stack Node* Top;
Public:
```

```
    Linked stack (  )
{
TOP = NULL;
}
Void push (int item) ;
Void pop (int litem);
In empty ( )
{ :
Return TOP = = NULL;
}
}
Void linkedstock : :  push (int item)
{
Top = new linked stacknode (item,top);
}
Void linkedstock =  = pop ( int titem)
{
Linked stack node* ptr;
Ptr = TOP;
;tem = TOP-> Data;
Top = Top - > link;
Delete ptr;
}
Linkedstack s;
Int x, n;
Count< < “\how  many Data”,
Cin >.n;
```

```
For (I =I; ;<=n;i+ +)
{ Cin>>x;
S.push (x);
}
Count << "\n show Data",
Whilev (! S.empty (j)
{
s. pop (x);
count << "\n" << x;
}
Return0;
}
```

Example: Using the link list concept, write a program for manipulating a queue structure.

```
Include < iostream .h>
Class linked Queue
{
Struct linkedQueuenode
{
Int Data;
linkedQueueNode*link;
LinkedQueue Node(int item, linkedQueue Node* mode modelink NULL)
{
Data = item;
Link = nodelinke;
}
```

```
};
LinkedQueuenode*rear,*front;
Public:
LinkedQueue
{
Front = NULL;
Rear = NULL;
}
Int emptyQueue ( )
{
Return front = = Null;
}
Void add Q(int item)
{
If(front = = NULL)
{
Rear = newlinkedQueuenode (item,rear);
From = rear;
}
Else
{rear –link=new linkeQueueNode(item,rear);
Front = rear;
}
Else
{
```

```
Rear->link =new linkdQueueNode(item,rear->link)
Rear = rear -> link;
}
}
Void delete(intlitem)
{
Linke QueueNode *ptr;
Ptr =front;
Item=front ->data;
Front = front ->link;
Delete ptr;
}
};
Main c )
{ Inti,x,n;
LinkedQueue Q;
Count> >\n how many data?”,
C”n > >n;
Count < < “\n enter Data”,
for( ; =l; ;<=n; ;+ +)
{
Cin > > x;
Q.add Q(x);
}
```

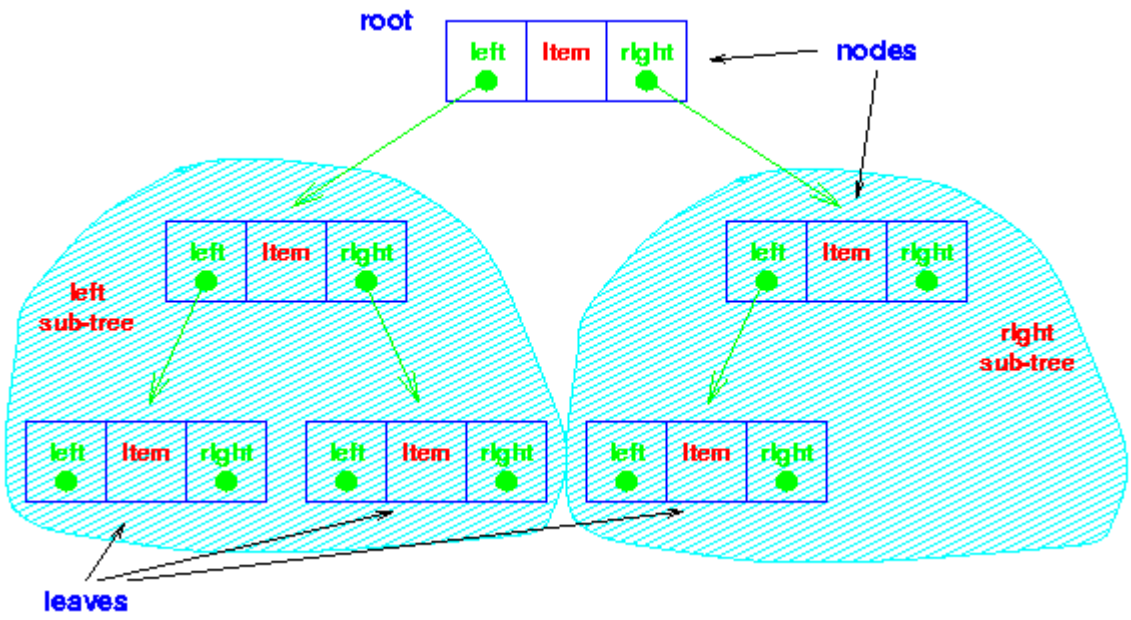
```
Count < < " inshow Data".

While (1 Q.emptyqueue (j)

{

Q. DeleteQ(x)

Count < < "\n" <<x;

}

Return O;

}
```

1.5 TREE DATA STRUCTURE

Binary Trees

The simplest form of tree is a **binary tree**. A binary tree consists of

a. a *node* (called the **root** node) and
b. left and right sub-trees.
   Both the sub-trees are themselves binary trees.



A binary tree

The nodes at the lowest levels of the tree (the ones with no sub-trees) are called **leaves**.

In an *ordered binary tree*,

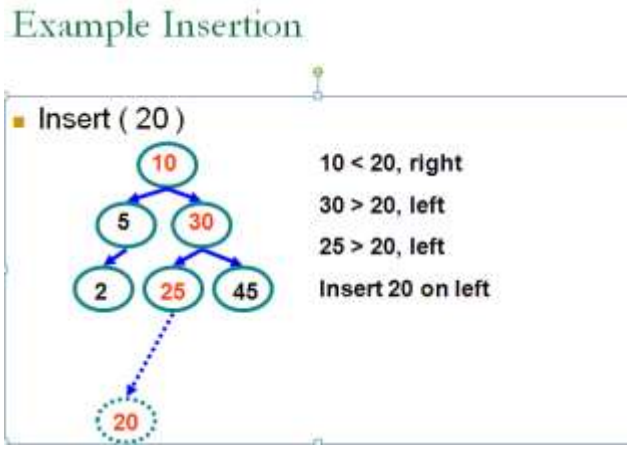
1. the keys of all the nodes in the left sub-tree are less than that of the root,
2. the keys of all the nodes in the right sub-tree are greater than that of the root,
3. the left and right sub-trees are themselves ordered binary trees.

1.5.1 Binary Search Tree Construction

- How to build & maintain binary trees?
  - Insertion
  - Deletion
- Maintain key property (invariant)
  - Smaller values in left subtree
  - Larger values in right subtree
- Binary Search Tree – Insertion

1.5.2 Algorithm

- Perform search for value X
- Search will end at node Y (if X not in tree)
- If X < Y, insert new leaf X as new left subtree for Y
- If X > Y, insert new leaf X as new right subtree for Y



Operator treated as the node/parent

Operand – the leaf

1.5.3 Binary Search Tree – Deletion

Algorithm

1. Perform search for value X
2. If X is a leaf, delete X
3. Else // must delete internal node

a) Replace with largest value Y on left subtree

OR smallest value Z on right subtree

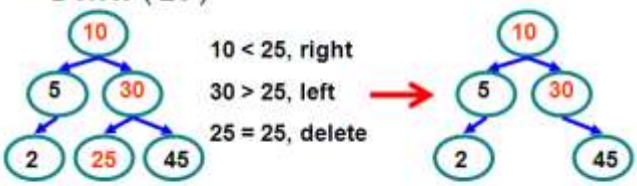
b) Delete replacement value (Y or Z) from subtree

Observation

- O(log(n)) operation for balanced tree
- Deletions may unbalance tree

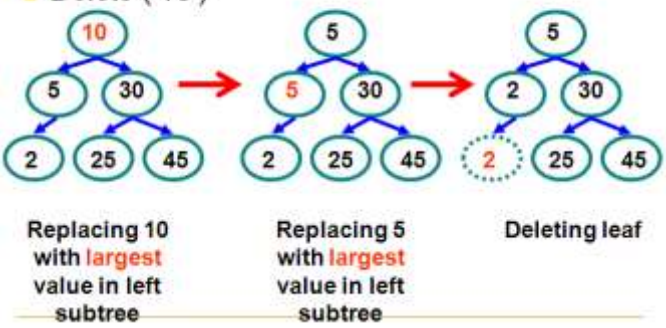
Example Deletion (Leaf)

- Delete (25)



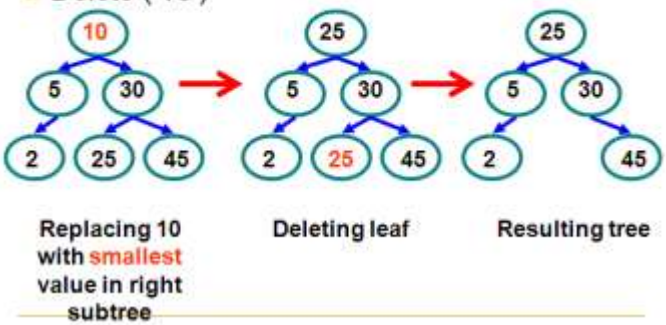
Example Deletion (Internal Node)

- Delete (10)



Example Deletion (Internal Node)

- Delete (10)

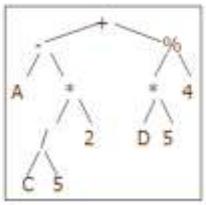


1.5.4 Traversal methods

Stepping through the items of a tree, by means of the connections between parents and children, is called **walking the tree**, and the action is a **walk** of the tree. Often, an operation might be performed when a pointer arrives at a particular node. A walk in which each parent node is traversed before its children is called a **pre-order walk**; a walk in which the children are traversed before their respective parents are traversed is called a **post-order walk**; a walk in which a node's left subtree, then the node itself, and then finally its right subtree are traversed is called an **in-order** traversal. (This last scenario, referring to exactly two subtrees, a left subtree and a right subtree, assumes specifically a binary tree.)

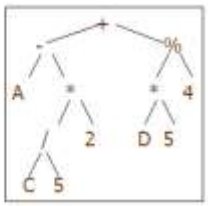
Parse Trees

- Expressions, programs, etc can be represented by tree structures
  - E.g. Arithmetic Expression Tree
  - A-(C/5 * 2) + (D*5 % 4)



Tree Traversal

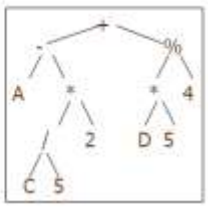
- Goal: visit every node of a tree
- in-order traversal




```
void Node::inOrder () {
   if (left != NULL)  {
      cout << "("; left->inOrder(); cout << ")";
   }
   cout << data << endl;
   if (right != NULL) right->inOrder()
}
```

Output: A – C / 5 * 2 + D * 5 % 4
To disambiguate: print brackets

Tree Traversal (contd.)



- **pre-order and post-order:**

```
void Node::preOrder () {
   cout << data << endl;
   if (left != NULL)   left->preOrder ();
   if (right != NULL) right->preOrder ();
}
```

Output: + - A * / C 5 2 % * D 5 4

```
void Node::postOrder () {
   if (left != NULL)   left->preOrder ();
   if (right != NULL) right->preOrder ();
   cout << data << endl;
}
```

Output: A C 5 / 2 * - D 5 * 4 % +

Pre-order traversal (pre-fix expression) NLR

Inorder – traversal (infix expression) LNR

Postorder – traversal (postfix expression) LRN

1.6 Graph Data Structures

E.g: Airline networks, road networks, electrical circuits

Consists of Nodes and Edges

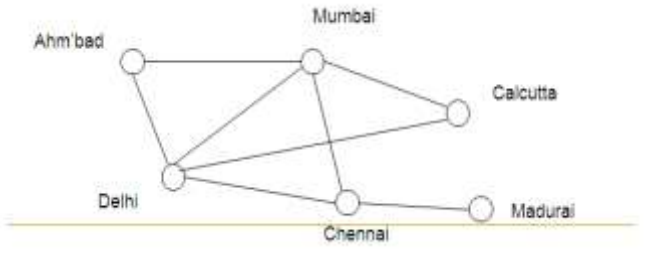
E.g. representation: class Node

a. Stores name

b. stores pointers to all adjacent nodes

i,e. edge == pointer

To store multiple pointers: use array or linked list



REVIEW QUESTIONS

1. Using C++, write a program that implements algorithm for inserting data elements into one dimensional array
2. Illustrate the concept of single linked list using C + + programming language.
3. Write a C++ program that implements the algorithms for pushing, popping and deleting data elements from the stack data structure
4. Illustrate how queue data structure is different from stack data structure
5. Construct a binary tree and apply the three traversal techniques on the following expression (A+B)*(C-D)
6. Discuss the concept of graph data structure

FURTHER READING

Salamis S, Data structures, Algorithms and application in Java, McGraw Hill

Banachowski I et al, Analysis of algorithms and Data structures, Addison wiley

CHAPTER TWO

INFIX AND POSTFIX EXPRESSION (REVERSE POLISH NOTATION)

Learning objectives:

By the end of the chapter a student shall be able to:

- **Understand and apply the algorithm in converting infix to postfix expression**

2.1 Infix Expression:

Any expression in the standard form like "2*3-4/5" is an Infix(Inorder) expression.

2.2 Postfix Expression:

The Postfix(Postorder) form of the above expression is "23*45/-".

2.3 Infix to Postfix Conversion:

In normal algebra we use the infix notation like a+b*c. The corresponding postfix notation is abc*+. The algorithm for the conversion is as follows :

- Scan the Infix string from left to right.
- Initialise an empty stack.
- If the scannned character is an operand, add it to the Postfix string. If the scanned character is an operator and if the stack is empty Push the character to stack.
- If the scanned character is an Operand and the stack is not empty, compare the precedence of the character with the element on top of the stack (topStack). If topStack has higher precedence over the scanned character Pop the stack else Push the scanned character to stack. Repeat this step as long as stack is not empty and topStack has precedence over the character.

- Repeat this step till all the characters are scanned.
- (After all characters are scanned, we have to add any character that the stack may have to the Postfix string.) If stack is not empty add topStack to Postfix string and Pop the stack. Repeat this step as long as stack is not empty.
- Return the Postfix string.

  Example:

  Let us see how the above algorithm will be imlemented using an example.

  Infix String : a+b*c-d

  Initially the Stack is empty and our Postfix string has no characters. Now, the first character scanned is 'a'. 'a' is added to the Postfix string. The next character scanned is '+'. It being an operator, it is pushed to the stack.

  - 

    Stack
  - 

    Postfix String

- Next character scanned is 'b' which will be placed in the Postfix string. Next character is '*' which is an operator. Now, the top element of the stack is '+' which has lower precedence than '*', so '*' will be pushed to the stack.

  - 

    Stack
  - 

    Postfix String

2.4 Evaluating Postfix Expression

Infix Expression:

Any expression in the standard form like "2*3-4/5" is an Infix(Inorder) expression.

Postfix Expression:

The Postfix(Postorder) form of the above expression is "23*45/-".

Postfix Evaluation:

In normal algebra we use the infix notation like a+b*c. The corresponding postfix notation is abc*+. The algorithm for the conversion is as follows :

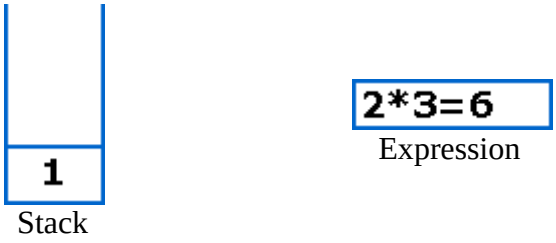
- Scan the Postfix string from left to right.
- Initialise an empty stack.
- If the scannned character is an operand, add it to the stack. If the scanned character is an operator, there will be atleast two operands in the stack.
- If the scanned character is an Operator, then we store the top most element of the stack(topStack) in a variable temp. Pop the stack. Now evaluate topStack(Operator)temp. Let the result of this operation be retVal. Pop the stack and Push retVal into the stack.
- Repeat this step till all the characters are scanned.
- After all characters are scanned, we will have only one element in the stack. Return topStack.
- Example :

  Let us see how the above algorithm will be imlemented using an example.

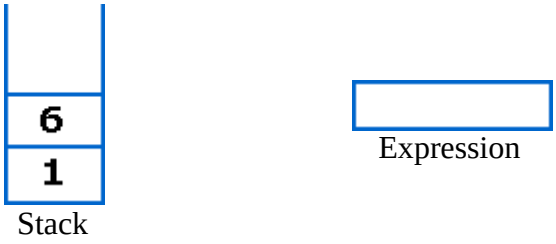
  Postfix String : 123*+4-

  Initially the Stack is empty. Now, the first three characters scanned are 1,2 and 3, which are operands. Thus they will be pushed into the stack in that order.

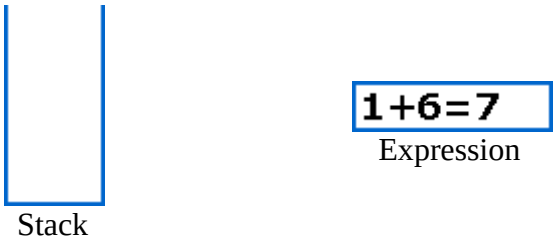
Next character scanned is "*", which is an operator. Thus, we pop the top two elements from the stack and perform the "*" operation with the two operands. The second operand will be the first element that is popped.



The value of the expression(2*3) that has been evaluated(6) is pushed into the stack.



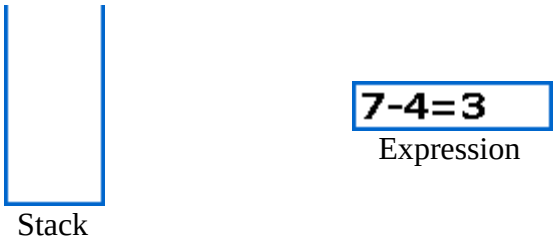
Next character scanned is "+", which is an operator. Thus, we pop the top two elements from the stack and perform the "+" operation with the two operands. The second operand will be the first element that is popped.



The value of the expression(1+6) that has been evaluated(7) is pushed into the stack.

Next character scanned is "4", which is added to the stack.

Next character scanned is "-", which is an operator. Thus, we pop the top two elements from the stack and perform the "-" operation with the two operands. The second operand will be the first element that is popped.



The value of the expression(7-4) that has been evaluated(3) is pushed into the stack.

Now, since all the characters are scanned, the remaining element in the stack (there will be only one element in the stack) will be returned.

End result :

- Postfix String : 123*+4-
- Result : 3

REVIEW QUESTIONS

1. Convert the expression a+b*c-d to Postfix expression
2. Evaluate the postfix string: 123*+4-

FURTHER READING

Salamis S, Data structures, Algorithms and application in Java, McGraw Hill

Banachowski I et al, Analysis of algorithms and Data structures, Addison wiley

CHAPTER THREE

RECURSIVE FUNCTIONS

Learning objectives:

By the end of the chapter a student shall be able to understand and write the algorithms of:

- **Factorial function**
- **Power function**
- **Fibonacci function**

These are functions that call themselves i.e self referencing functions

Some of the recursive functions include:

-factorial function

-fibonacci sequence

-power function

-Binary tree functions

3.1 Factorial functions

Example

4:=4x3x2x1 =24

4x3;

4x3x2;

4x3x2x1;

= 4(4-1)!

n: =n* (n-1) !

Example 1

Write a program for computing factorial of positive numbers using recursive functions.

Solu

```
#include <;ostrem h>
Void main ( )
{
Int x;
Int factorial (int n);
Cout<<"\n enter a positive number"'
Cin>>x.
Count<<' \in the answer  is "< < factorial (x)
}
Int factorial (int n)
{
If (n<1;
Else
Return 1;
Else return n* fractional (n-1)
}
```

3.2 Fibonacci Function

This is a sequence of the type

Fib(n) = 1 where n<=2

Fib(n) fib (n-1) + fib (n-2) where n>2

1,1,2,3,5,8,13,21,34,55,89,144

Fib (3) = fib (3-1) + fib (3-2)

=fib(2) + fib (1)

=1 + 1

2

Write a program for computing Fibonacci sequence

```
#include iostream.h>
Void main( )
{
Int I,n,
Int fib (in +n);
cout < < “In Enter the nth term of the sequence”;
cout>>n:
for (I = 1; i< = n; itt)
cout<<fib(i) << “\t”;
}
Int fib (int n)
{
```

```
If(n < =2)
Return  1;
Else
Return (fib(n-1) + fib(n-2));
}
```

3.3 Power function

Example

Write a program for computing “power” using recursive formation.

Solution

```
#include < iosream.n>
Void main ( )
{
Int x,n;
Int power(int x, int n);
Count<< “\n Enter a number”,
Cin >>x;
Count << \n enter power to use”;
Cin >.n;
Count<< “\n the result is” << power (x,n);
}
Int power (int x, int n)
{
If (n= =0)
Return I;
```

```
Else
Return * power (x,n-1)
}
Version 2
# include < iostream.h>
Class powers
{
Int  x,n;
Int  ans;
Public :
Void getpowervalues(int a, int b);
Void computerpower(int a,int b);
};
Void POWERS: : getpower values (int a, int b)
{
X =a;
N = b;
}
Void POWERS : : computerpower(int a, int b)
{
If (b = = 1)
ans =1;
Else
ans = a * computerpower(a, b-1);
Cout << “the answer is ”<< ans;
```

```
}
Void main ( )
{
POWERS P;
Int y,z;
Count<< " in enter power to raise to" << Z;
P.get power values (y,z)
p.computerpower (y,z)
}
```

Version 3

```
# include <iostream.h>
Class POWER: : getpowervalue  ( )
{
Count<< "\n  entera  number",
C:n>>x;
Count<< "\n  enterpower to raise",
Cin >>n
}
Void Powers: : computerpower( )
{
If (n = = 0)
   ans = 1;
Else
Ans =x * computepower(x, n-1);
Cout << " The answer is "<< ans;
```

```
}
Void main ( )
{
POWERS P;
p. getpowerva.ines( );
p.computerpower( );
}
```

REVIEW QUESTIONS

1. Write algorithms for computing:

- Power function
- Factorial function
- Fibonacci function

FURTHER READING

Salamis S, Data structures, Algorithms and application in Java, McGraw Hill

Banachowski I et al, Analysis of algorithms and Data structures, Addison wiley

CHAPTER FOUR

SEARCH AND SORT ALGORITHMS

Learning objectives:

By the end of the chapter a student shall be able to understand and apply the:

- **Linear search and binary search algorithms**
- **Bubble, insertion, selection and quick sort algorithms to data structures**

4.1 SEARCHING

Looking for data /information stored in a structure, array, file,etc

-we have two main search techniques namely:

4.1.1 Linear search

A linear search looks down a list, one item at a time, without jumping. In complexity terms this is an O(n) search - the time taken to search the list gets bigger at the same rate as the list does. Linear involves comparing data element to find

As an example, suppose you were looking for U in an A-Z list of letters (index 0-25; we're looking for the value at index 20).

A linear search would ask:

list[0] == 'U'? No.
list[1] == 'U'? No.
list[2] == 'U'? No.
list[3] == 'U'? No.
list[4] == 'U'? No.
list[5] == 'U'? No.
... list[20] == 'U'? Yes. Finished.

Linear- search algorithm:

- Create a structure
- Insert elements
- Initialize the counter to 1
- Indicate the search key
- Set a loop
- Start comparing with the data in the structure
- If found stop search else
- Increment counters by 1 and then go to (vi)

Example

Write a program for searching data stored in an array structure.

Solution

```
#include <iostream.h>
Void main ( )
{
Int I ;
Int list [50];
Int n;
Int x;
Count<< “in how many data elements?”.cin >>n;
Count<<” \n enter in data”,
For (I = 1; i< =n; i++)
{
```

Cin >> list[50];

}

Count << “\n Enter element to search”,

Cin>>x;i=1;

While (i< = n)

{

If(x = = list list [i])

{

Count <<”\n element is found at location”<< I;

Break;

}

Else

I ++;

}

If(i>n)

Count<< “\n element never found”;

}

Example2

C++ Source Code

```
#include <iostream.h>

int LinearSearch(int [], int, int);

int main()
{
const int NUMEL = 10;
int nums[NUMEL] = {5,10,22,32,45,67,73,98,99,101};
int item, location;


cout << "Enter the item you are searching for: ";
cin >> item;
```

```
location = LinearSearch(nums, NUMEL, item);

if (location > -1)
cout << "The item was found at index location " << location
<< endl;
else
cout << "The item was not found in the list\n";

return 0;
}

// this function returns the location of key in the list
// a -1 is returned if the value is not found
int LinearSearch(int list[], int size, int key)
{
int i;

for (i = 0; i < size; i++)
{
if (list[i] == key)
return i;
}

return -1;
}
```

Linear search takes along time to search.

4.1.2 Binary search

A binary search is when you start with the middle of a sorted list, and see whether that's greater than or less than the value you're looking for, which determines whether the value is in the first or second half of the list. Jump to the half way through the sublist, and compare again etc. This is pretty much how humans typically look up a word in a dictionary (although we use better heuristics, obviously - if you're looking for "cat" you don't start off at "M"). In complexity terms this is an O(log n) search - the number of search operations grows more slowly than the list does, because you're halving the "search space" with each operation.

Utilizes the concept of divide and conquer. It assume that the data element are sorted out. It addresses the limitation of linear search.

The binary search would ask:

Compare `list[12]` ('M') with 'U': Smaller, look further on. (Range=13-25)
Compare `list[19]` ('T') with 'U': Smaller, look further on. (Range=20-25)
Compare `list[22]` ('W') with 'U': Bigger, look earlier. (Range=20-21)
Compare `list[20]` ('U') with 'U': Found it! Finished.

The algorithm is as follows

3) Create a structure
4) Insert the elements
5) Indicate search key
6) Initialize top to 1
7) Initialize bottom to n (last position)
8) While (top< = bottom)
   i. comput mid position
   ii. compare search key with the element at the mid position; if found stop searching

   Else
   iii. if search key < element at mid position adjust bottom position, say mid-1

   else adjust top position say mid + 1

Example

4,9,11,13,16,20,28

Establish weather 20 is in the list

Step1

Top=1

Bottom = 7

Mid = (1+7)/2

Step 2

Top = mid +1

= 4+1

= 5

Bottom = 7

Mid = (7+5)/2

= 6

Example

Write a paragram for searching data elements using binary search technique.

Solution

```
# include < iostream.h>

Void main ( )

{

Int I,n,x;

Int list [50];

Int top,bottom,mind;

Count <<\"in how many data?"

Cin>>n;

Count<< "\n enter data;.

For (i=l; i< = n; I ++ )

Cin >> l1 st [i];

Count<<"\n enter element l search",

Cin>>x;

Top = 1;

Bottom = n;
```

```
While (top < = bottom)

{

Mind = (top + bottom)/2;

If (x = = list [mid])

{

Cout << "\n element found at location"<<mid;

Break;

}

Else

If (x<list [mid] )

Bottom = mid – 1;

Else

Top = mid + 1;

}

If (top >bottom)

Count << "\n  element never found";

}
```

Comparing the two:

- Binary search requires the input data to be sorted; linear search doesn't
- Binary search requires an *ordering* comparison; linear search only requires equality comparisons
- Binary search has complexity O(log n); linear search has complexity O(n) as discussed earlier
- Binary search requires random access to the data; linear search only requires sequential access (this can be very important - it means a linear search can *stream* data of arbitrary size)

4.2 SORTING ALGORITHMS

4.2.1 Bubble sort

Compare each element (except the last one) with its neighbor to the right

If they are out of order, swap them

This puts the largest element at the very end

The last element is now in the correct and final place

Compare each element (except the last two) with its neighbor to the right

If they are out of order, swap them

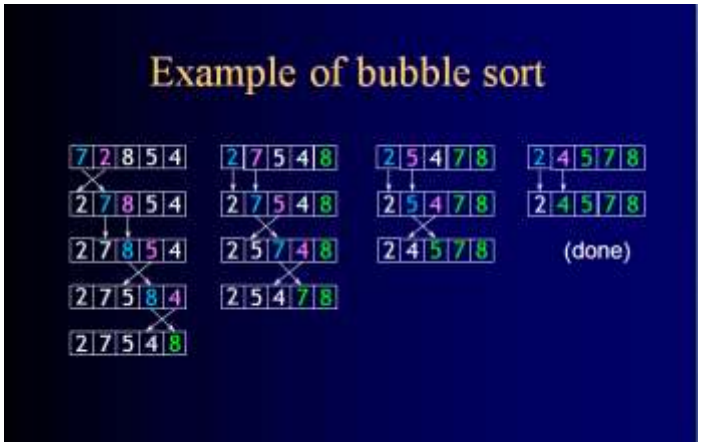
This puts the second largest element next to last

The last two elements are now in their correct and final places

Compare each element (except the last three) with its neighbor to the right

Continue as above until you have no unsorted elements on the left

Example of bubble sort



Code for bubble sort

```
public static void bubbleSort(int[] a) {
  int outer, inner;
  for (outer = a.length - 1; outer > 0; outer--) {  // counting down
    for (inner = 0; inner < outer; inner++) {        // bubbling up
      if (a[inner] > a[inner + 1]) {  // if out of order...
        int temp = a[inner];          // ...then swap
        a[inner] = a[inner + 1];
        a[inner + 1] = temp;
      }
    }
  }
}
```

Analysis of bubble sort

```
for (outer = a.length - 1; outer > 0; outer--) {
  for (inner = 0; inner < outer; inner++) {
    if (a[inner] > a[inner + 1]) {
      // code for swap omitted
    }
  }
}
```

Let n = a.length = size of the array

The outer loop is executed n-1 times (call it n, that's close enough)

Each time the outer loop is executed, the inner loop is executed

Inner loop executes n-1 times at first, linearly dropping to just once

On average, inner loop executes about n/2 times for each execution of the outer loop

In the inner loop, the comparison is always done (constant time), the swap might be done (also constant time)

Result is n * n/2 * k, that is, O(n2/2 + k) = O(n2)

Loop invariants

You run a loop in order to change things

Oddly enough, what is usually most important in understanding a loop is finding an invariant: that is, a condition that doesn't change

In bubble sort, we put the largest elements at the end, and once we put them there, we don't move them again

The variable outer starts at the last index in the array and decreases to 0

Our invariant is: Every element to the right of outer is in the correct place

That is, for all j > outer, if i < j, then a[i] <= a[j]

When this is combined with outer == 0, we know that all elements of the array are in the correct place

4.2.2 Selection sort

Given an array of length n,

Search elements 0 through n-1 and select the smallest

Swap it with the element in location 0

Search elements 1 through n-1 and select the smallest

Swap it with the element in location 1

Search elements 2 through n-1 and select the smallest

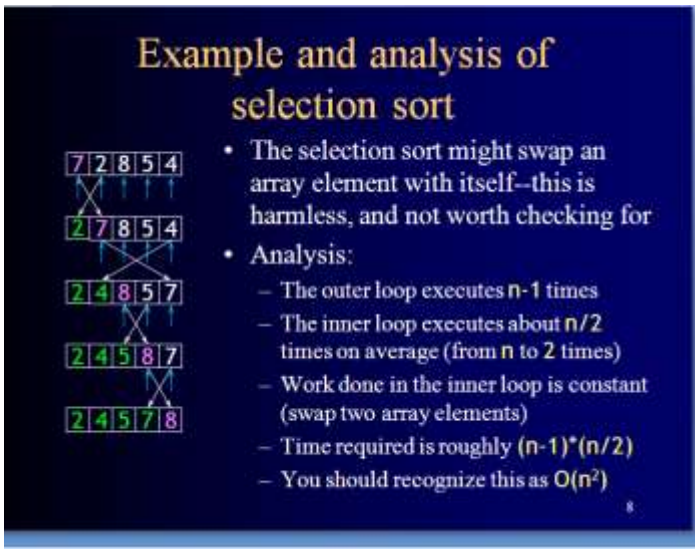
Swap it with the element in location 2

Search elements 3 through n-1 and select the smallest

Swap it with the element in location 3

Continue in this fashion until there's nothing left to search

Example and analysis of selection sort



The selection sort might swap an array element with itself--this is harmless, and not worth checking for

Analysis:

The outer loop executes n-1 times

The inner loop executes about n/2 times on average (from n to 2 times)

Work done in the inner loop is constant (swap two array elements)

Time required is roughly (n-1)*(n/2)

You should recognize this as O(n2)

Code for selection sort

```
public static void selectionSort(int[] a) {
   int outer, inner, min;
   for (outer = 0; outer < a.length - 1; outer++) { // outer counts down
      min = outer;
      for (inner = outer + 1; inner < a.length; inner++) {
         if (a[inner] < a[min]) {
            min = inner;
         }
```

```
            // Invariant: for all i, if outer <= i <= inner, then a[min] <= a[i]
        }

            // a[min] is least among a[outer]..a[a.length - 1]
        int temp = a[outer];
        a[outer] = a[min];
        a[min] = temp;
        // Invariant: for all i <= outer, if i < j then a[i] <= a[j]
    }
}
```

Invariants for selection sort

For the inner loop:

This loop searches through the array, incrementing inner from its initial value of outer+1 up to a.length-1

As the loop proceeds, min is set to the index of the smallest number found so far

Our invariant is:
for all i such that outer <= i <= inner, a[min] <= a[i]

For the outer (enclosing) loop:

The loop counts up from outer = 0

Each time through the loop, the minimum remaining value is put in a[outer]

Our invariant is:
for all i <= outer, if i < j then a[i] <= a[j]

4.2.3 Insertion sort

The outer loop of insertion sort is:

```
for (outer = 1; outer < a.length; outer++) {...}
```

The invariant is that all the elements to the left of outer are sorted with respect to one another

For all i < outer, j < outer, if i < j then a[i] <= a[j]

This does not mean they are all in their final correct place; the remaining array elements may need to be inserted

When we increase outer, a[outer-1] becomes to its left; we must keep the invariant true by inserting a[outer-1] into its proper place

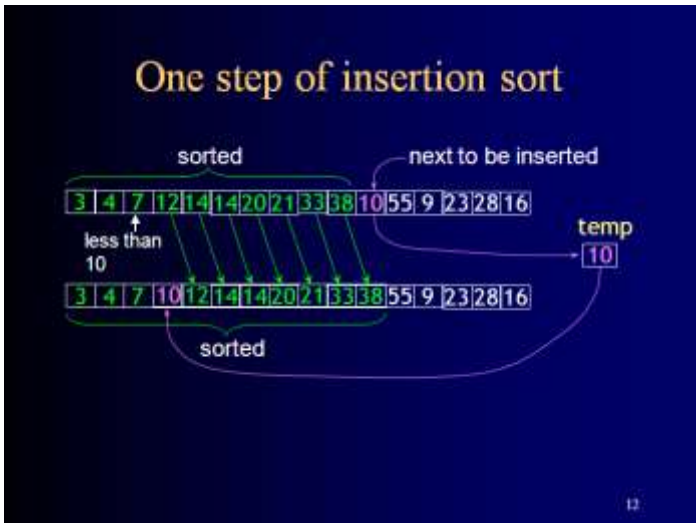
This means:

Finding the element's proper place

Making room for the inserted element (by shifting over other elements)

Inserting the element

One step of insertion sort



Analysis of insertion sort

We run once through the outer loop, inserting each of n elements; this is a factor of n

On average, there are n/2 elements already sorted

The inner loop looks at (and moves) half of these

This gives a second factor of n/4

Hence, the time required for an insertion sort of an array of n elements is proportional to n2/4

Discarding constants, we find that insertion sort is O(n2)

Summary

Bubble sort, selection sort, and insertion sort are all O(n2)

As we will see later, we can do much better than this with somewhat more complicated sorting algorithms

Within O(n2),

Bubble sort is very slow, and should probably never be used for anything

Selection sort is intermediate in speed

Insertion sort is usually the fastest of the three--in fact, for small arrays (say, 10 or 15 elements), insertion sort is faster than more complicated sorting algorithms

Selection sort and insertion sort are “good enough” for small arrays

REVIEW QUESTIONS

Differentiate the following types of concept:

1. Linear and binary search
2. Bubble and selection sort
3. Use the quick-sort algorithm to sort the data: 9,4,10, 3, 2,11

FURTHER READING

Salamis S, Data structures, Algorithms and application in Java, McGraw Hill

Banachowski I et al, Analysis of algorithms and Data structures, Addison wiley

MKlaus W., Algorithms, data structures, programmes, New Delhi, MCGraw Hill

SAMPLE EXAMS QUESTIONS

BIT 2204: Data structures and Algorithms

DATE: TIME: 2 HOURS

Answer question ONE and ANY other TWO

QUESTION ONE

a) Explain briefly the meaning of the following terms

i. data type
ii. Abstract data type (ADT)
iii. Pointers
iv. Data structure

(4 marks)

b) For each of the following situations, which of these ADT''s (1 through 4) would be most appropriate:

i. a queue,
ii. a stack,
iii. a list,
iv. none of these?

i. The customers at a Kenchicken's counter who take numbers to make their turn
ii. Integers that need to be sorted
iii. Arranging plates in the cafeteria
iv. People who are put on hold when they call Kenya Airways to make reservations
v. Converting infix to postfix expression

(5 marks)

c) Explain why a test for an empty stack must be carried out when performing stack operations. Write a procedure/ function for the function EMPTY of a stack identifier

(4 marks)

d) I) If you push the letters A, B, C and D in order onto a stack of characters and then POP them , in what order will they be deleted from the stack

(2 marks)

ii) Represent the following expression as binary tree and write prefix and postfix form of the expression

(A+B+C*D)-(A/B-CD+E)

(4 marks)

f) I) Define a Queue and explain why it is also reffered to as a FIFO

(2 marks)

ii) What is a priority Queue? Give an example

(3 marks)

g) State and define all the possible operations on a stack data structure

(6 marks)

QUESTION TWO

a) Describe how deletion of a node in between the linked list can be carried out illustrated your answer with a diagram

(5 marks)

b) Beginning with an empty binary search tree what binary search tree is formed when you insert the following values in the order

i. W,T,N,J,E,B,A
ii. A,B,W,J,N,T,E

(4 marks)

c) i)Explain the importance of a head node

(1 mark)

ii) State two advantages of linked list over arrays
(2 marks)

iii) Each element of a doubly linked structure has three fields. State the three fields illustrating your answer with a diagram
(2 marks)

iv) Describe the procedure of deleting an element at position P in a doubly linked list, illustrating your answer with a diagram
(4 marks)

v) State one advantage of circular list
(2 marks)

QUESTION THREE

a) Convert the following infix arithmetic expression into its equivalent reverse polish form

i. A+B*C
ii. (A+B)*C
iii. A/CB-(C+D)*(E-A)*C
iv. A/B-C+D*E+A+C

(4 marks)

b) Use stack to evaluate the postfix expression ABC+D*+E+. Show the status of the stack after each step of the algorithm. Assume the following values for the identifiers: A=8, B=5, C=3, D=9, E=4.

(4 marks)

c) i)Suppose that the vowels form a tree with “O” as the root and its children are “U”, “I”,”A”, left-to-right and “E” is the only child of “I”. Reconstruct this tree as a binary tree

(3 marks)

ii) Trace the bubble sort algorithm as it sort the following array into ascending order:

20 80 40 25 60 30

(2 marks)

d) Write an algorithm for converting Numbers from Base 10 to any other given base. Use an example program to implement the algorithm

(7 marks)

QUESTION FOUR

a) State the algorithm of fibonacci sequence. Use your algorithm to write a program for computing fibonacci sequence

(5 marks)

b) I) Briefly define the quicksort algorithm

(2 marks)

ii) Write the algorithm for the quicksort

(4 marks)

iii) Using quicksort technique sort the following data elements. Use diagrams to trace the algorithm

5 6 20 80 105 89 40 6 204 76 (9 marks)

QUESTION FIVE

a) I) Construct a binary search tree using the following data

50 70 25 90 30 55 25 15 25

(3 marks)

ii) Using the above information trace the algorithm for deleting node 30

(6 marks)

iii) Using the linked list concept, write a program for manipulating a Queue structure

(11 marks)