Ambo University Waliso Campus

Information Technology Department

Course Code: ITec2051

Course Title: Data structure and Algorithms

Target Group: 2rd year Information Technology Students

Chapter One

Data Structures and Algorithms Analysis

1. Introduction to Data Structures and Algorithms Analysis

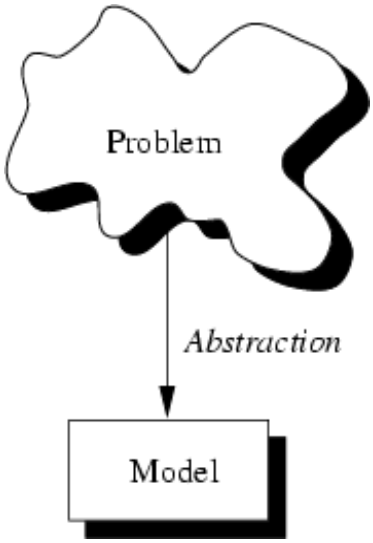
A program is written in order to solve a problem. A solution to a problem actually consists of two things:

- A way to organize the data
- Sequence of steps to solve the problem

The way data are organized in a computers memory is said to be Data Structure and the sequence of computational steps to solve a problem is said to be an algorithm. Therefore, a program is nothing but data structures plus algorithms.

1.1. Introduction to Data Structures

Given a problem, the first step to solve the problem is obtaining ones own abstract view, or *model*, of the problem. This process of modeling is called *abstraction.*



The model defines an abstract view to the problem. This implies that the model focuses only on problem related stuff and that a programmer tries to define the *properties* of the problem.

These properties include

- The *data* which are affected and
- The *operations* that are involved in the problem.

With abstraction you create a well-defined entity that can be properly handled. These entities define the *data structure* of the program.

An entity with the properties just described is called an *abstract data type* (ADT).

1.1.2. Abstraction

Abstraction is a process of classifying characteristics as relevant and irrelevant for the particular purpose at hand and ignoring the irrelevant ones.Applying abstraction correctly is the essence of successful programming.

1.2. Algorithms

An algorithm is a well-defined computational procedure that takes some value or a set of values as input and produces some value or a set of values as output. Data structures model the static part of the world. They are unchanging while the world is changing. In order to model the dynamic part of the world we need to work with algorithms. Algorithms are the dynamic part of a program's world model.

An algorithm transforms data structures from one state to another state in two ways:

- An algorithm may change the value held by a data structure
- An algorithm may change the data structure itself

The quality of a data structure is related to its ability to successfully model the characteristics of the world. Similarly, the quality of an algorithm is related to its ability to successfully simulate the changes in the world.

However, independent of any particular world model, the quality of data structure and algorithms is determined by their ability to work together well. Generally speaking, correct data structures lead to simple and efficient algorithms and correct algorithms lead to accurate and efficient data structures.

1.2.1. Properties of an algorithm

- **Finiteness**: Algorithm must complete after a finite number of steps.
- **Definiteness**: Each step must be clearly defined, having one and only one interpretation. At each point in computation, one should be able to tell exactly what happens next.
- **Sequence**: Each step must have a unique defined preceding and succeeding step. The first step (start step) and last step (halt step) must be clearly noted.
- **Feasibility**: It must be possible to perform each instruction.
- **Correctness**: It must compute correct answer for all possible legal inputs.
- **Language Independence**: It must not depend on any one programming language.
- **Completeness**: It must solve the problem completely.
- **Effectiveness**: It must be possible to perform each step exactly and in a finite amount of time.
- **Efficiency**: It must solve with the least amount of computational resources such as time and space.
- **Generality**: Algorithm should be valid on all possible inputs.
- **Input/Output**: There must be a specified number of input values, and one or more result values.

1.2.2. Algorithm Analysis Concepts

Algorithm analysis refers to the process of determining the amount of computing time and storage space required by different algorithms. In other words, it's a process of predicting the resource requirement of algorithms in a given environment.

In order to solve a problem, there are many possible algorithms. One has to be able to choose the best algorithm for the problem at hand using some scientific method. To classify some data

structures and algorithms as good, we need precise ways of analyzing them in terms of resource requirement. The main resources are:

- Running Time
- Memory Usage
- Communication Bandwidth

Running time is usually treated as the most important since computational time is the most precious resource in most problem domains.

There are two approaches to measure the efficiency of algorithms:

- Empirical: Programming competing algorithms and trying them on different instances.
- Theoretical: Determining the quantity of resources required mathematically (Execution time, memory space, etc.) needed by each algorithm.

However, it is difficult to use actual clock-time as a consistent measure of an algorithm's efficiency, because clock-time can vary based on many things. For example,

- Specific processor speed
- Current processor load
- Specific data for a particular run of the program
  - Input Size
  - Input Properties
- Operating Environment

Accordingly, we can analyze an algorithm according to the number of operations required, rather than according to an absolute amount of time involved. This can show how an algorithm's efficiency changes according to the size of the input.

1.3. Measures of Times

In order to determine the running time of an algorithm it is possible to define three functions $T_{best}(n)$, $T_{avg}(n)$ and $T_{worst}(n)$ as the best, the average and the worst case running time of the algorithm respectively.
Average Case (T_{avg}): The amount of time the algorithm takes on an "average" set of inputs.
Worst Case (T_{worst}): The amount of time the algorithm takes on the worst possible set of inputs.
Best Case (T_{best}): The amount of time the algorithm takes on the smallest possible set of inputs.

We are interested in the worst-case time, since it provides a bound for all input – this is called the "Big-Oh" estimate.

1.4. Asymptotic Analysis

Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

There are five notations used to describe a running time function. ***These are:***

- Big-Oh Notation (O)
- Big-Omega Notation (Ω)
- Theta Notation (Θ)
- Little-o Notation (o)
- Little-Omega Notation (ω)

1.4.1. The Big-Oh Notation

Big-Oh notation is a way of comparing algorithms and is used for computing the complexity of algorithms; i.e., the amount of time that it takes for computer program to run . It's only concerned with what happens for very a large value of n. Therefore only the largest term in the expression (function) is needed. For example, if the number of operations in an algorithm is $n^2 – n$, n is insignificant compared to n^2 for large values of n. Hence the n term is ignored. Of course, for small values of n, it may be important. However, Big-Oh is mainly concerned with large values of n.

Formal Definition: f (n)= O (g (n)) if there exist c, k ∈ $\mathscr{R}^+$ such that for all n≥ k, f (n) ≤ c.g (n).

Examples: The following points are facts that you can use for Big-Oh problems:

- 1<=n for all n>=1
- $n<=n^2$ for all n>=1
- 2^n <=n! for all n>=4
- $\log_2 n<=n$ for all n>=2
- $n<=n\log_2 n$ for all n>=2

1. f(n)=10n+5 and g(n)=n. Show that f(n) is O(g(n)).

To show that f(n) is O(g(n)) we must show that constants c and k such that

f(n) <=c.g(n) for all n>=k

Or 10n+5<=c.n for all n>=k

Try c=15. Then we need to show that 10n+5<=15n

Solving for n we get: 5<5n or 1<=n.

So f(n) =10n+5 <=15.g(n) for all n>=1.

(c=15,k=1).

2. f(n) = $3n^2$ +4n+1. Show that f(n)=O(n^2).

4n <=$4n^2$ for all n>=1 and 1<=n^2 for all n>=1

$3n^2$ +4n+1<=$3n^2$+$4n^2$+n^2 for all n>=1

<=$8n^2$ for all n>=1

So we have shown that f(n)<=$8n^2$ for all n>=1

Therefore, f (n) is O(n^2) (c=8,k=1)

Typical Orders

Here is a table of some typical cases. This uses logarithms to base 2, but these are simply proportional to logarithms in other base.

N	O(1)	O(log n)	O(n)	O(n log n)	O(n^2)	O(n^3)
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4,096
1024	1	10	1,024	10,240	1,048,576	1,073,741,824

Demonstrating that a function f(n) is big-O of a function g(n) requires that we find specific constants c and k for which the inequality holds (and show that the inequality does in fact hold).

Big-O expresses an *upper bound* on the growth rate of a function, for sufficiently large values of n.

An *upper bound* is the best algorithmic solution that has been found for a problem.

" What is the best that we know we can do?"

Exercise:

$f(n) = (3/2)n^2+(5/2)n-3$

Show that $f(n)= O(n^2)$

In simple words, f (n) =O(g(n)) means that the growth rate of f(n) is less than or equal to g(n).

1.4.1.1. Big-O Theorems

For all the following theorems, assume that f(n) is a function of n and that k is an arbitrary constant.

Theorem 1: k is O(1)

Theorem 2: A polynomial is O(the term containing the highest power of n).

Polynomial's growth rate is determined by the leading term

- If *f(n)* is a polynomial of degree *d*, then *f(n)* is $O(n^d)$

In general, f(n) is big-O of the dominant term of f(n).

Theorem 3: k*f(n) is O(f(n))

Constant factors may be ignored

E.g. $f(n) =7n^4+3n^2+5n+1000$ is $O(n^4)$

Theorem 4(Transitivity): If f(n) is O(g(n))and g(n) is O(h(n)), then f(n) is O(h(n)).

Theorem 5: For any base b, $\log_b(n)$ is O(logn).

All logarithms grow at the same rate

$\log_b n$ is $O(\log_d n)$ □*b, d > 1*

Theorem 6: Each of the following functions is big-O of its successors:

k

$\log_b n$

n

$n \log_b n$

n^2

n to higher powers

2^n

3^n

larger constants to the nth power

n!

n^n

$f(n) = 3n\log_b n + 4\log_b n + 2$ is $O(n\log_b n)$ and $)(n^2)$ and $O(2^n)$

1.4.1.2. Properties of the O Notation

Higher powers grow faster

- n^r is $O(n^s)$ if $0 <= r <= s$

Fastest growing term dominates a sum

- If f(n) is O(g(n)), then f(n) + g(n) is O(g)

E.g $5n^4 + 6n^3$ is $O(n^4)$

Exponential functions grow faster than powers, i.e. is $O(b^n)$ $\forall b > 1$ and $k >= 0$

E.g. n^{20} is $O(1.05^n)$

Logarithms grow more slowly than powers

- $\log_b n$ is $O(nk)$ $\forall$ $b > 1$ and $k >= 0$

E.g. $\log_2 n$ is $O(n^{0.5})$

1.4.2. Big-Omega Notation

Just as O-notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Formal Definition: A function f(n) is Ω(g (n)) if there exist constants c and k ∈ ℛ+ such that

f(n) >=c. g(n) for all n>=k.

f(n)= *Ω(g (n))* means that f(n) is greater than or equal to some constant multiple of g(n) for all values of n greater than or equal to some k.

Example*:* If $f(n) = n^{2,}$ then f(n)= *Ω(n)*

In simple terms, f(n)= *Ω(g (n))* means that the growth rate of f(n) is greater that or equal to g(n).

1.4.3. Theta Notation

A function f (n) belongs to the set of Θ (g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched between c1.g(n) and c2.g(n), for sufficiently large values of n.

Formal Definition: A function f (n) is Θ (g(n)) if it is both *O(g(n))* and *Ω (g(n))*. In other words, there exist constants c1, c2, and k >0 such that c1.g (n)<=f(n)<=c2. g(n) for all n >= k

If f(n)= Θ (g(n)), then g(n) is an asymptotically tight bound for f(n).

In simple terms, f(n)= Θ (g(n)) means that f(n) and g(n) have the same rate of growth.

Example:

1. If f(n)=2n+1, then f(n) = Θ (n)

2. $f(n) = 2n^2$ then, $f(n)=O(n^4)$, $f(n)=O(n^3)$, $f(n)=O(n^2)$

All these are technically correct, but the last expression is the best and tight one. Since $2n^2$ and n^2 have the same growth rate, it can be written as $f(n)= \Theta(n^2)$.

1.4.4. Little-o Notation

Big-Oh notation may or may not be asymptotically tight, for example: $2n^2 = O(n^2) = O(n^3)$

f(n)=o(g(n)) means for all c>0 there exists some k>0 such that f(n)<c.g(n) for all n>=k. Informally, f(n)=o(g(n)) means f(n) becomes insignificant relative to g(n) as n approaches infinity.

Example: f(n)=3n+4 is $o(n^2)$

In simple terms, f(n) has less growth rate compared to g(n).

g(n)= $2n^2$ g(n) =$o(n^3)$, $O(n^2)$, g(n) is not $o(n^2)$.

1.4.5. Little-Omega (ω notation)

Little-omega (ω) notation is to big-omega (Ω) notation as little-o notation is to Big-Oh notation. We use ω notation to denote a lower bound that is not asymptotically tight.

Formal Definition: f(n)= ω (g(n)) if there exists a constant no>0 such that 0<= c. g(n)<f(n) for all n>=k.

Example: $2n^2$=ω(n) but it's not ω (n^2).

1.5. Relational Properties of the Asymptotic Notations

Transitivity

- if f(n)=Θ(g(n)) and g(n)= Θ(h(n)) then f(n)=Θ(h(n)),
- if f(n)=O(g(n)) and g(n)= O(h(n)) then f(n)=O(h(n)),
- if f(n)=Ω(g(n)) and g(n)= Ω(h(n)) then f(n)=Ω (h(n)),
- if f(n)=o(g(n)) and g(n)= o(h(n)) then f(n)=o(h(n)), and
- if f(n)=ω (g(n)) and g(n)= ω(h(n)) then f(n)=ω (h(n)).

Symmetry

- f(n)=Θ(g(n)) if and only if g(n)=Θ(f(n))

Transpose symmetry

- f(n)=O(g(n)) if and only if g(n)=Ω(f(n)),
- f(n)=o(g(n)) if and only if g(n)=ω(f(n)).

Reflexivity

- $f(n)=\Theta(f(n))$,
- $f(n)=O(f(n))$,
- $f(n)=\Omega(f(n))$.

Chapter Two
Introduction to Sorting and Searching

Why do we study sorting and searching algorithms?

These algorithms are the most common and useful tasks operated by computer system. Computers spend a lot of time searching and sorting.

1. Simple Searching algorithms

Searching:- is a process of finding an element in a list of items or determining that the item is not in the list. To keep things simple, we shall deal with a list of numbers. A search method looks for a key, arrives by parameter. By convention, the method will return the index of the element corresponding to the key or, if unsuccessful, the value1.

There are two simple searching algorithms:

- Sequential Search, and
- Binary Search

Sequential Searching (Linear)

The most natural way of searching an item. Easy to understand and implement.

Algorithm:

- In a linear search, we start with top (beginning) of the list, and compare the element at top with the key.
- If we have a match, the search terminates and the index number is returned.
- If not, we go on the next element in the list.
- If we reach the end of the list without finding a match,we return_1.

Implementation:

Assume the size of the list is n.

```
#include <iostream>
int LinearSearch(int list[ ], int key);
using namespace std;
int main(){
int list[] = {5, 3, 7, 4, 6};
int k = 4;
int i = LinearSearch(list, k);
if(i==-1)
cout << "the search item is not found" << endl;
else
cout << "The value is found at index position " << i << endl;
return 0;}
int LinearSearch(int list[ ], int key){
int index=-1;
for(int i=0; i < n; i++){
if(list[i]==key){
index=i;
```

```
break;

}}

return index;}
```

Complexity Analysis:

Big-Oh of sequential searching: How many comparisons are made in the worst case ?

n O(n).

Binary Searching

It assumes the data is sorted it also uses divide and conquer strategy (approach).

Algorithm:

- In a binary search, we look for the key in the middle of the list. If we get a match, the search is over.
- If the key is greater than the element in the middle of the list, we make the top (upper) half the list to search.
- If the key is smaller, we make the bottom (lower) half the list to search.
- Repeat the above steps (I,II and III) until one element remains.
- If this element matches return the index of the element, else return -1 index. (-1 shows that the key is not in the list).

Implementation:

```
#include <iostream>
int BinarySearch(int list[ ], int key);
using namespace std;
int main(){
int list[] = {15, 23, 47, 54, 76};
int k = 54;
int i = BinarySearch(list, k);
if(i==-1)
cout << "the search item is not found" << endl;
else
cout << "The value is found at index position " << i << endl;
return 0;}
int BinarySearch(int list[ ], int key){
int found=0,index=0;
int top=n-1,bottom=0,middle;
do{
middle=(top + bottom)/2;
if(key==list[middle])
found=1;
```

```
else{
if(key < list[middle])
top=middle-1;
else bottom=middle+1;}
}while(found==0 && top>=bottom);
if(found==0)
index=-1;
else
index=middle;
return index;}
```

Complexity Analysis:

Example: find Big-Oh of Binary search algorithm in the worst case analysis.

Sorting: is a process of reordering a list of items in either increasing or decreasing order. Ordering a list of items is fundamental problem of computer science. Sorting is the most important operation performed by computers. Sorting is the first step in more complex algorithms.

Two basic properties of sorting algorithms:

In-place: It is possible to sort very large lists without the need to allocate additional working storage.

Stable: If two elements that are equal remain in the same relative position after sorting is completed.

Two classes of sorting algorithms:O(n2): Includes the bubble, insertion, and selection sorting algorithms. **O(nlog n):**Includes the heap, merge, and quick sorting algorithms.

Simple sorting algorithms include:

- Simple sorting
- Bubble Sorting
- Selection Sorting
- Insertion Sorting

Simple sort

Algorithm: In simple sort algorithm the first element is compared with the second, third and all subsequent elements. If any one of the other elements is less than the current first element then the first element is swapped with that element. Eventually, after the last element of the list is considered and swapped, then the first element has the smallest element in the list. The above steps are repeated with the second, third and all subsequent elements.

Implementation:

```
#include <iostream>
using namespace std;
void SimpleSort(int list[]);
int list[] = {5, 3, 7, 4, 6};
int main(){
cout << "The values before sorting are: \n";
for(int i = 0; i < 5; i++)
cout << list[i] << " ";
SimpleSort(list);
cout << endl;
cout << "The values after sorting are: \n";
for(int i = 0; i < 5; i++)
cout << list[i] << " ";
return 0;}
void SimpleSort(int list[]){
for(int i=0; i<=n-2;i++)
        for(int j=i+1; j<=n-1; j++)
        if(list[i] > list[j]){
int temp;
temp=list[i];
list[i]=list[j];
        list[j]=temp;27.}}
```

Analysis: O(?)

1st pass-----==> (n-1) comparisons

2nd pass----==> (n-2) comparisons

(n-1)th pass---==> 1 comparison

T(n)=1+2+3+4+-------+(n-2)+(n-1) = (n*(n-1))/2 =n2/2-n/2 =O(n2)

Complexity Analysis: Analysis involves number of comparisons and swaps.

How many comparisons? 1+2+3+…+(n-1)= O(n2)

How many swaps? 1+2+3+…+(n-1)= O(n2)

Example: Suppose we have 32 unsorted data.

a). How many comparisons are made by sequential search in the worst-case? ==> Number of comparisons =32.

b). How many comparisons are made by binary search in the worst-case? (Assuming simple sorting). ==> Number of comparisons = Number of comparisons for sorting + Number of comparisons for binary search = (n*(n-1))/2 + logn = 32/2(32-1)+ log 32 =16*31 + 5

c). How many comparisons are made by binary search in the worst-case if data is found to be already sorted? ==> Number of comparisons = log2 32 = 5.

Selection Sort

Algorithm

- The selection sort algorithm is in many ways similar to simple sort algorithms. The idea of algorithm is quite simple.
- Array is imaginary divided into two parts - sorted one and unsorted one.
- At the beginning, sorted part is empty, while unsorted one contains whole array.

- At every step, algorithm finds minimal element in the unsorted part and adds it to the end of the sorted one
- When unsorted part becomes empty, algorithm stops.
- Works by selecting the smallest unsorted item remaining in the list, and then swapping it with the item in the next position to be filled.
- Similar to the more efficient insertion sort.
- It yields a 60% performance improvement over the bubble sort.

Advantage: Simple and easy to implement.

Disadvantage: Inefficient for larger lists.

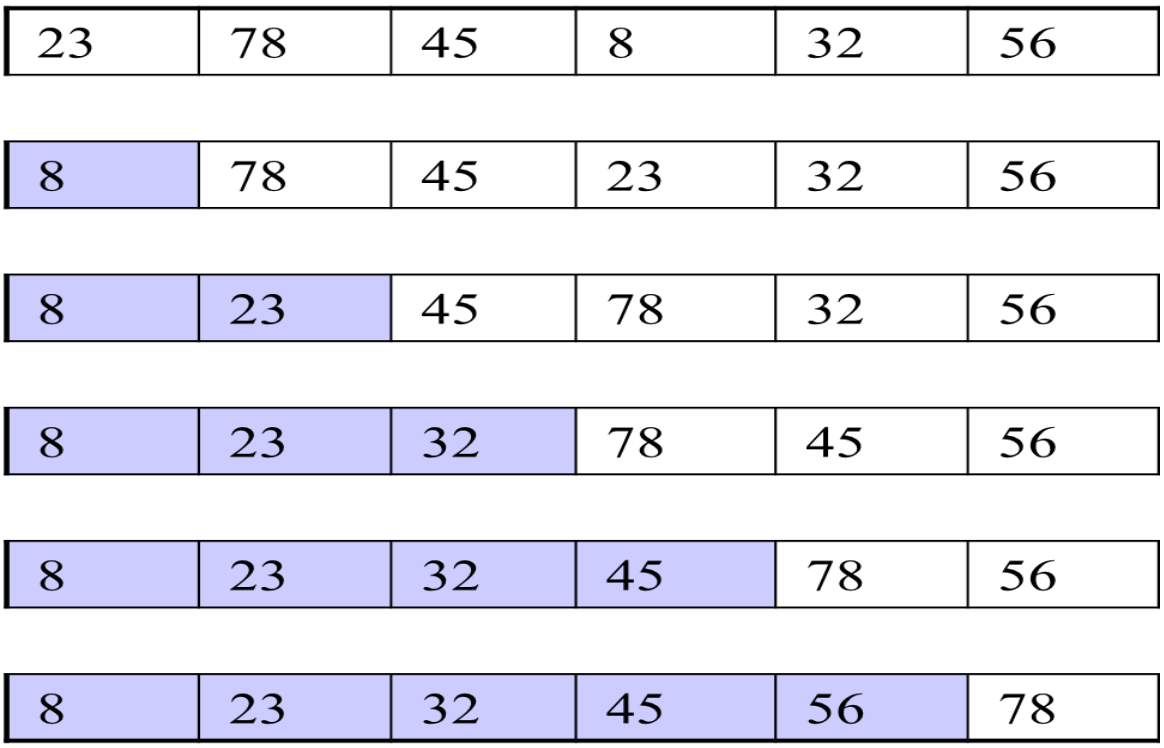
Implementation:

```
#include <iostream>
using namespace std;
void SimpleSort(int list[]);
int list[] = {5, 3, 7, 4, 6};
int main(){
cout << "The values before sorting are: \n";
for(int i = 0; i < 5; i++)
cout << list[i] << " ";
SimpleSort(list);
cout << endl;
cout << "The values after sorting are: \n";
for(int i = 0; i < 5; i++)
cout << list[i] << " ";
return 0;}
void SimpleSort(int list[]){
for(int i=0; i<=n-2;i++)
        for(int j=i+1; j<=n-1; j++)
        if(list[i] > list[j]){
int temp;
temp=list[i];
list[i]=list[j];
        list[j]=temp;}}
```

Complexity Analysis

Selection sort stops, when unsorted part becomes empty. As we know, on every step number of unsorted elements decreased by one. therefore, selection sort makes n-1 steps (n is number of elements in array) of outer loop, before stop. every step of outer loop requires finding minimum

in unsorted part. Summing up, (n - 1) + (n - 2) + ... + 1, results in O(n2) number of comparisons .number of swaps may vary from zero (in case of sorted array) to n-1 (in case array was sorted in reversed order), which results in O(n) number of swaps. overall algorithm complexity is O(n2).Fact, that selection sort requires n-1 number of swaps at most, makes it very efficient in situations, when write operation is significantly more expensive, than read operation.



Insertion Sort

Algorithm:

Insertion sort algorithm somewhat resembles Selection Sort and Bubble sort. Array is imaginary divided into two parts - sorted one and unsorted one. At the beginning, sorted part contains first element of the array and unsorted one contains the rest. At every step, algorithm takes first

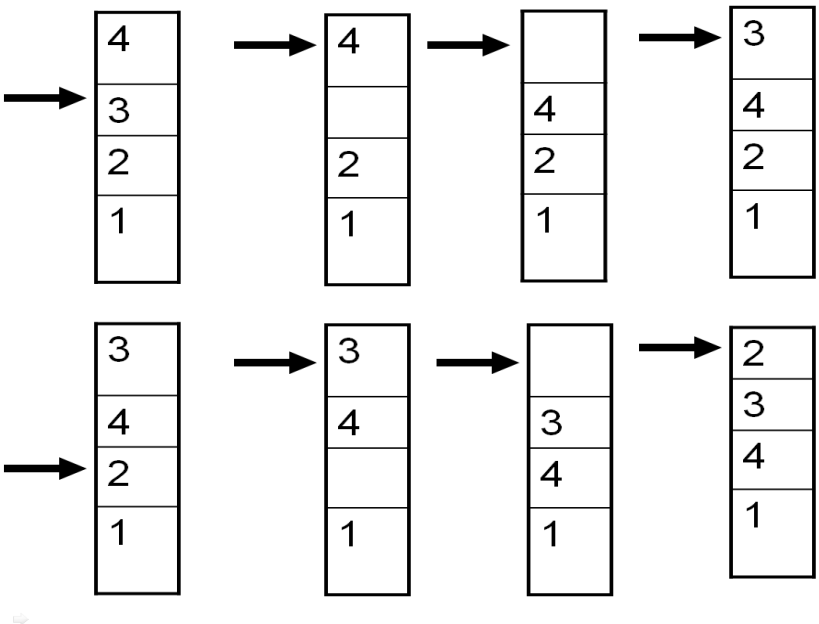
element in the unsorted part and inserts it to the right place of the sorted one. When unsorted part becomes empty, algorithm stops.

Using binary search

It is reasonable to use binary search algorithm to find a proper place for insertion. This variant of the insertion sort is called **binary insertion sort**. After position for insertion is found, algorithm shifts the part of the array and inserts the element. Insertion sort works by inserting item into its proper place in the list. Insertion sort is simply like playing cards: To sort the cards in your hand, you extract a card, shift the remaining cards and then insert the extracted card in the correct place. This process is repeated until all the cards are in the correct sequence. Is over twice as fast as the bubble sort and is just as easy to implement as the selection sort.

Advantage: Relatively simple and easy to implement.

Disadvantage: Inefficient for large lists



23	78	45	8	32	56
23	78	45	8	32	56
23	45	78	8	32	56
8	23	45	78	32	56
8	23	32	45	78	56
8	23	32	45	56	78

Implementation

```
#include <iostream>

using namespace std;

void InsertionSort(int list[]);

int list[] = {5, 3, 7, 4, 6};

int main(){

cout << "The values before sorting are: \n";

for(int i = 0; i < 5; i++)

cout << list[i] << " ";

InsertionSort(list);

cout << endl;

cout << "The values after sorting are: \n";

for(int i = 0; i < 5; i++)

cout << list[i] << " ";

return 0;}

void InsertionSort(int list[]){

for (int i = 1; i <= n-1; i++){
```

```
for(int j = i;j>=1; j--){

if(list[j-1] > list[j]){

int temp = list[j];

list[j] = list[j-1];

list[j-1] = temp;}

else break;

}}}
```

Complexity Analysis

The complexity of insertion sorting is O(n) at best case of an already sorted array andO(n2) at worst case, regardless of the method of insertion.Number of comparisons may vary depending on the insertion algorithm. O(n2) for shifting or swapping methods. O(nlogn) for binary insertion sort.

Chapter three

Abstract data type and link list

There are two broad types of data structure based on their memory allocation:

- Static Data Structures
- Dynamic Data Structure

Static Data Structures

Static Data Structures Are data structures that are defined & allocated before execution, thus the size cannot be changed during time of execution. Example: Array implementation of ADTs.

Dynamic Data Structure

Dynamic Data Structure Are data structure that can grow and shrink in size or permits discarding of unwanted memory during execution time.

Example: Linked list implementation of ADTs.

Structure: Structure is a collection of data items and the data items can be of different data type. The data item of structure is called **member of the structure**.

Declaration of structure

Structure is defined using the struct keyword.

```
struct name{
data type1 member 1;
data type2 member 2;
data type n member n;};
```

Example

```
struct student{
string name;
int age;
string Dept;};
```

The **struct** keyword creates a new user defined data type that is used to declare variable of an aggregated data type.

Accessing Members of Structure Variables

- The Dot operator (.): to access data members of structure variables.
- The Arrow operator (->): to access data members of pointer variables pointing to the structure.

Example:

struct student stud;

struct student *studptr;

cout<<stud.name;or cout<name;

Self-Referential Structures

Structures can hold pointers to instances of themselves.

Example:

struct student{

char name[20];

int age;

char Dept[20];

struct student *next;

};

Linked List

Linked List is self-referential structure. It is a collection of elements called **nodes**, each of which stores two types of fields. **Data items** and a **pointer** to next node in the case of singly linked list and **pointer** to previous node in the case of doubly linked list.

The data field: holds the actual elements on the list.

The pointer field: contains the address of the next and/or previous node in the list.

Adding a node to the list

Steps

Allocate a new node.

Set the node data values and make new node point to NULL.

Make old last node's next pointer point to the new node.

*Make the new last node's prev pointer point to the old last node. (This is only for Double Linked list).

Traversing through the list

To Move Forward: Set a pointer to point to the same thing as the start (head) pointer.

If the pointer points to NULL, display the message "list is empty" and stop.

Otherwise, move to the next node by making the pointer point to the same thing as the next pointer of the node it is currently indicating.

To Move Forward: (Double linked list)

Set a pointer to point to the same thing as the start pointer.

If the pointer points to NULL, display the message "list is empty" and stop.

Set a new pointer and assign the same value as start pointer and move forward until you find the node before the one we are considering at the moment.

To Move backward: (Double linked list)

Set a pointer to point to the same thing as the end (tail) pointer.

If the pointer points to NULL, display the message "list is empty" and stop.

Otherwise, move back to the previous node by making the pointer point to the same thing as the prev pointer of the node it is currently indicating.

Display the content of list

Steps:

- Set a temporary pointer to point to the same thing as the start pointer.
- If the pointer points to NULL, display the message "End of list" and stop.
- Otherwise, display the data values of the node pointed to by the start pointer.
- Make the temporary pointer point to the same thing as the next pointer of the node it is currently indicating.
- Jump back to step 2.

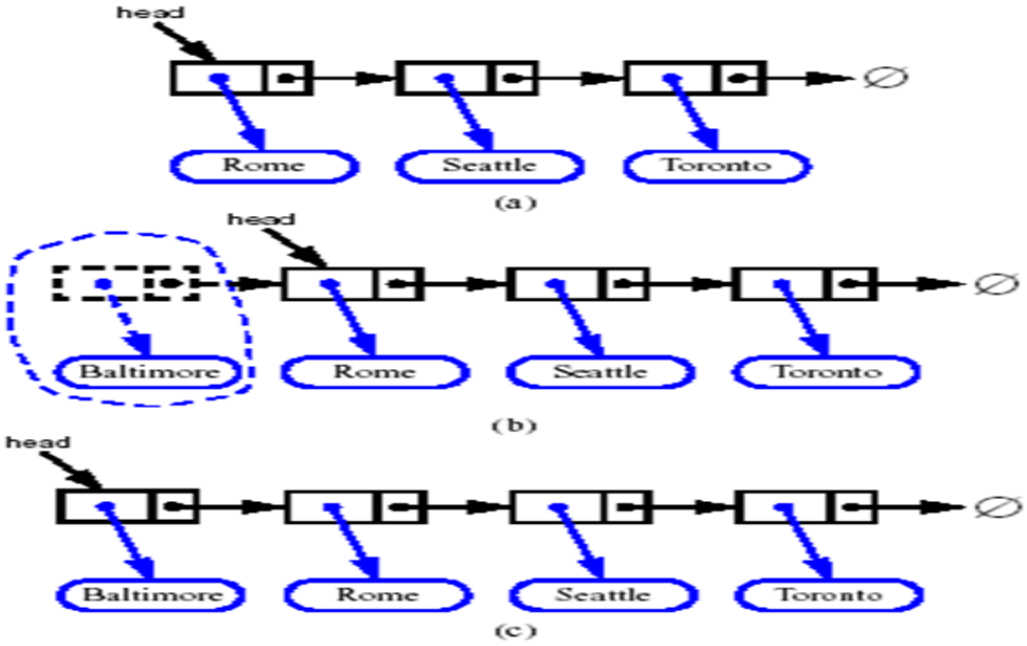
Insert at the front (beginning)

Allocate a new node.

Insert new element values.

Make the next pointer of the new node point to old head (start).

Update head (start) to point to the new node.



Inserting at the End

Steps

Allocate a new node.

Set the node data values and make the next pointer of the new node point to NULL.

Make old last node's next pointer point to the new node.

Update end to point to the new node.

Insertion in the middle

Steps:

Create a new Node

Set the node data Values

Break pointer connection

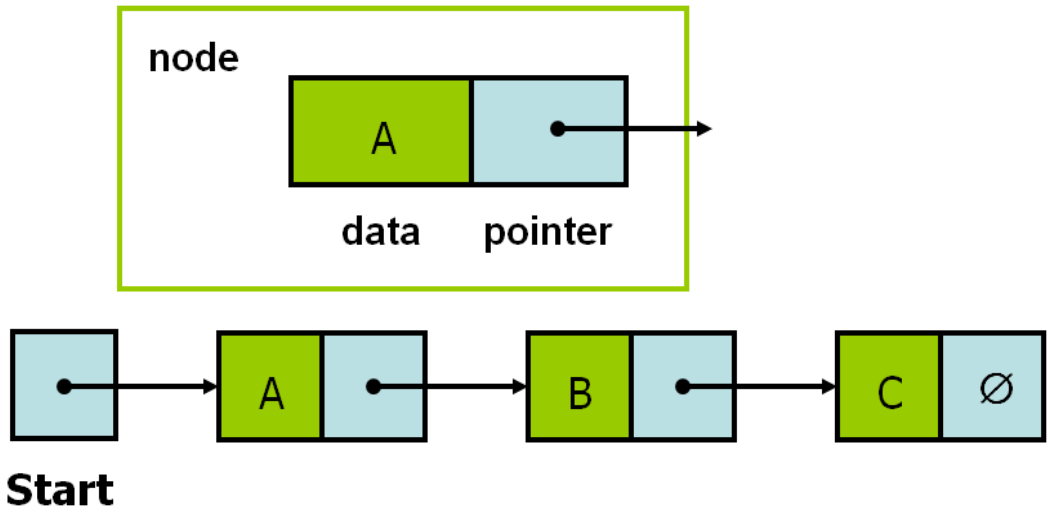
Re-connect the pointers

Types of Linked List

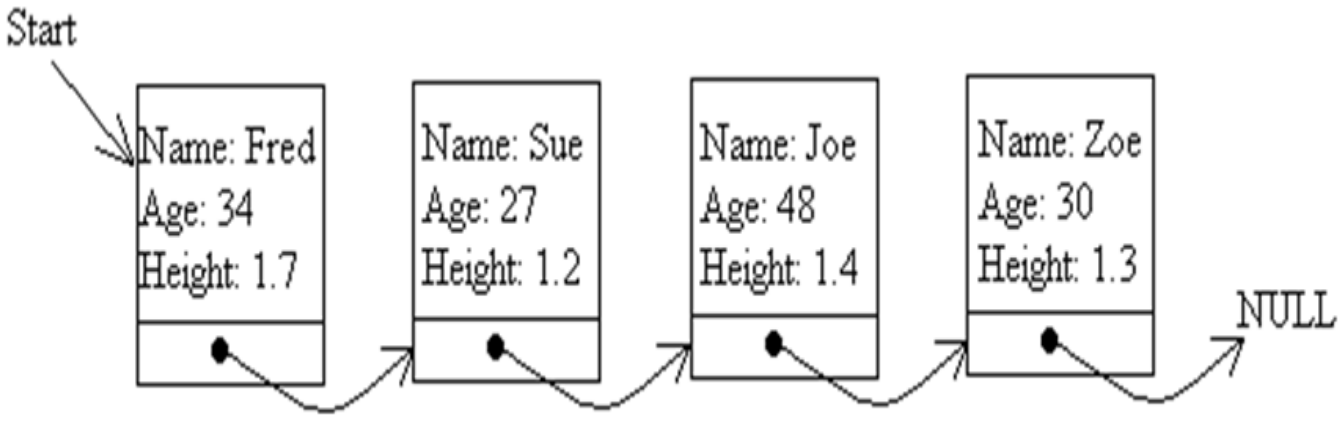
- **Single linked lists:**
- **Doubly linked lists**
- **Circular linked lists**

Singly Linked List

A singly linked list can be represented by a diagram like shown blow:



Start (Head): Special pointer that points to the first node of a linked list, so that we can keep track of the linked list. The last node should points to NULL to show that it is the last link in the chain (in the linked list).



According to the above example in the figure, it is the singly linked list which has four nodes in it, each with a link to the next node in the series (in the linked list).

C++ implementation of singly linked list:

```
struct node{

int data;

node *next;};

node *head = NULL;
```

Let us consider the above structure definition to perform the upcoming linked list operations.

Operations of singly Linked List

- Adding a node to the end of a singly linked list
- Adding a node to the left of a specific data in a singly linked list
- Adding a node to the right of a specific data in a singly linked list
- Deleting a node from the end of a singly linked list
- Deleting a node from the front of a singly linked list
- Deleting any node using the search data from a singly linked list
- Display the node from the singly linked list in a forward manner

Adding a node to the end of a singly linked list

```
void insert_end(int x){
        node *temp=new node;
        temp->data=x;
        temp->next=NULL;
        if(head==NULL)
        head = temp;
        else{
        node *temp2 = head;
        while(temp2->next!=NULL){
       temp2 = temp2->next;}
        temp2->next = temp;
        }}
```

Adding a node to the front of a singly linked list

```
void insert_front(int x){
node *temp=new node;
temp->data=x;
temp->next=NULL;
if(head==NULL)
head = temp;
```

```
else{
temp->next = head;
        head = temp;
        }}
```

Adding a node to the right of a specific value in a singly linked list

```
void insert_right_y(int x, int y){
        node *temp=new node;
        temp->data=x;
        temp->next=NULL;
        if(head==NULL)
        head = temp;
        else{
        node *temp2 = head;
        while(temp2->data!=y){
        temp2 = temp2->next;}
        temp->next = temp2->next;
        temp2->next = temp;
        }}
```

Adding a node to the left of a specific value in a singly linked list

```
        void insert_left_y(int x, int y){

        node *temp=new node;

        temp->data=x;

        temp->next=NULL;

        if(head==NULL)

        head = temp;

        else{

        node *temp2 = head;

        node *temp3;

        while(temp2->data!=y){

        temp3 = temp2;

        temp2 = temp2->next;}

        temp->next = temp3->next;
```

```
        temp3->next = temp;

        }}
```

Deleting a node from the front of a singly linked list

```
void delete_front(){
        node *temp;
        if(head==NULL)
        cout <<"No data inside\n";
        else{
        temp = head;
        head = head->next;
        delete temp;
        }}
```

Deleting a node from the end of a singly linked list

```
void delete_end(){
        node *temp, *temp3;
if(head==NULL)
        cout <<"No data inside\n";
        else {
        temp = head;
        while(temp->next!=NULL) {
        temp3 = temp;
        temp = temp->next;   }
        temp3->next = NULL;
        delete temp;
        }}
```

Deleting a node of specific data of a singly linked list

```
void delete_any(int x){

        node *temp, *temp3;

        if(head==NULL)

        cout <<"No data inside\n";

        if(head->data==x) {

        temp = head;

        head = head->next;
```

```
delete temp;}

else{

temp = head;

while(temp->data!=x){

temp3 = temp;

temp = temp->next;}

temp3->next = temp->next;

delete temp;}}
```

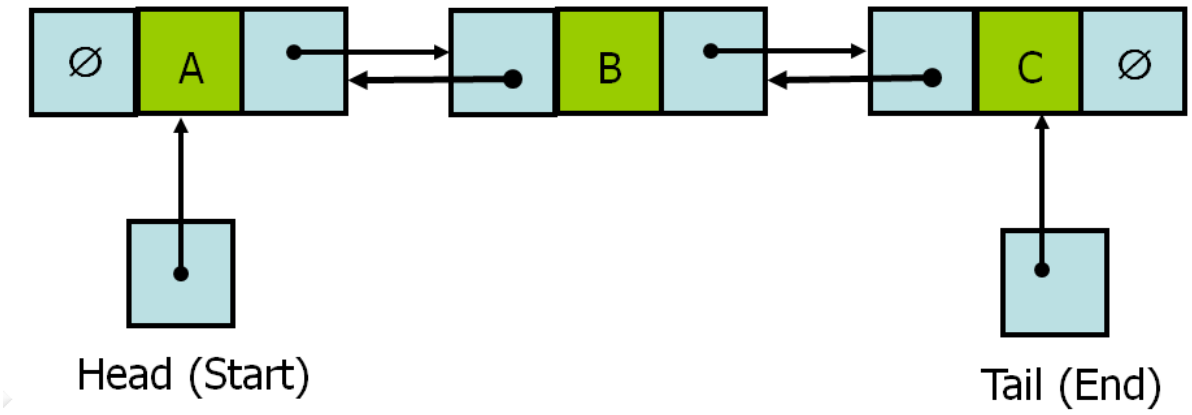
Display in a forward manner in a singly linked list

```
void display(){

node *temp;

if(head==NULL)

cout << "No data inside\n";

else{

temp = head;

while(temp!=NULL){

cout <<temp->data << endl;

temp = temp->next;

}}}
```

Doubly linked lists

Each node points not only to Successor node (Next node), but also to Predecessor node (Previous node).There are two NULL: at the first and last nodes in the linked list.

Advantage: given a node, it is easy to visit its predecessor (previous) node. It is convenient to traverse linked lists Forwards and Backwards.



Operations of Doubly Linked List

- Adding a node to the end of a doubly linked list
- Adding a node to the front of a doubly linked list
- Adding a node to the left of a specific data in a doubly linked list
- Adding a node to the right of a specific data in a doubly linked list
- Deleting a node from the end of a doubly linked list
- Deleting a node from the front of a doubly linked list
- Deleting any node using the search data from a doubly linked list
- Display the node from the doubly linked list in a forward manner
- Display the node from the doubly linked list in a backward manner.

```
struct node{
        int data;
        node *prev;
        node *next;};
        node *head = NULL, *tail = NULL;
```

Adding a node to the end of a doubly linked list

```
void insert_end(int x)     {
node* temp = new node;
        temp->data = x;
        temp->next = NULL;
        temp->prev = NULL;
```

```
if (head == NULL)

head = tail = temp;

else {

tail->next = temp;

temp->prev = tail;

tail = temp;

}}
```

Adding a node to the front of a doubly linked list

```
void insert_front(int x){

node* temp = new node;

 temp->data = x;

temp->next = NULL;

  temp->prev = NULL;

if (head == NULL)

  head = tail = temp;

else{

 temp->next = head;

head->prev = temp;

head = temp;

}}
```

Adding a node to the left of a specific data in a doubly linked list

```
void insert_left_y(int x, int y)
{
node* temp = new node;
temp->data = x;
temp->next = NULL;
temp->prev = NULL;
if (head == NULL)
head = tail = temp;
```

```
else
if(head->data==y)
temp3 = temp2;
temp2 = temp2->next;
}
temp->next = temp3->next;
temp3->next = temp;
temp->prev = temp3;
        temp2->prev = temp;
}}
```

Adding a node to the right of a specific data in a doubly linked list

```
        void insert_right_y(int x, int y){

        node* temp = new node;

        temp->data = x;

        temp->next = NULL;

       temp->prev = NULL;

        if (head == NULL)

        head = tail = temp;

        else

        if(head->data==y){

        if(head->next==NULL)

        tail = temp;

        temp->prev = head;

        temp->next = head->next;

        head->next->prev = temp;

        head->next = temp;}

        else    {

        node *temp2 = head;

        while(temp2->data!=y){
```

Deleting a node from the end of a doubly linked list

```
void delete_end(){
        node *temp;
        if(tail==NULL)
        cout <<"No data inside\n";
        else{
        temp = tail;
        tail = tail->prev;
       tail->next = NULL;
        delete temp;}}
```

Deleting a node from the front of a doubly linked list

```
void delete_front(){

        node *temp;

        if(head==NULL)

        cout <<"No data inside\n";

        else{

        temp = head;

        head = head->next;

        head->prev = NULL;

        delete temp;}}
```

Deleting any node using the search data from a doubly linked list

```
        void delete_any(int y){

        if(head==NULL)

        cout <<"No data inside\n";

        else{

        node *temp = head, *temp2;

        while(temp->data != y){

        temp2 = temp;

        temp = temp->next;}

        temp2->next = temp->next;
```

```
temp->next->prev = temp2;

delete temp;}}
```

Display the node from the doubly linked list in a forward manner

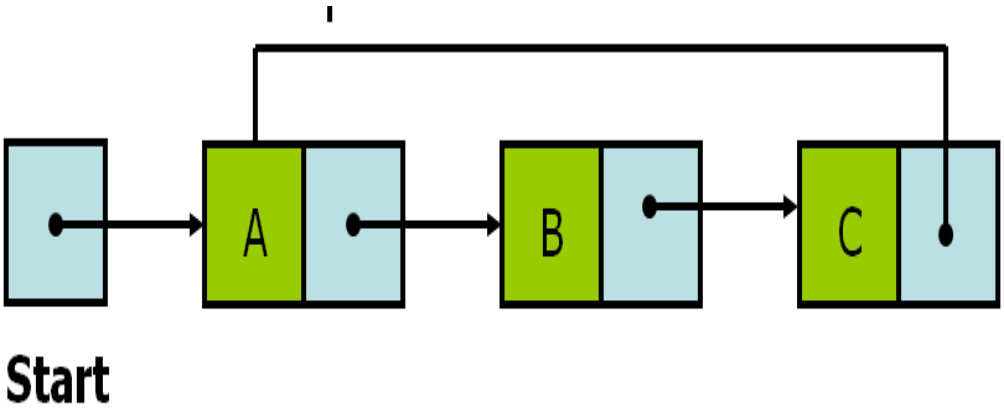
```
void display_ forward(){

node *temp;

if(head==NULL)

cout <<"No data inside\n";

else{

temp = head;

while(temp!=NULL){

cout << temp->data << endl;

temp = temp->next;

}}}
```

Display the node from the doubly linked list in a backward manner

```
void display_backward(){

node *temp;

if(tail==NULL)

cout <<"No data inside\n";

else{

temp = tail;

while(temp!=NULL){

cout << temp->data << endl;

temp = temp->prev;

}}}
```

Circular linked lists

Circular linked lists: The last node points to the first node of the list.



How do we know when we have finished traversing the list? (Hint: check if the pointer of the current node is equal to the Start (head) pointer).

Operations of Circular Singly Linked List

- Adding a node to the end of a Circular singly linked list
- Adding a node to the left of a specific data in a Circular singly linked list
- Deleting a node from the end of a Circular singly linked list
- Deleting a node from the front of a Circular singly linked list
- Display the node from the Circular singly linked list in a forward manner

```
struct node{

int data;

node *next;};

node *head = NULL;
```

Adding a node to the end of a Circular Singly linked list

```
void insert_end(int x){
        node *temp = new node;
        temp->data = x;
        temp->next = temp;
        if(head==NULL)
        head = temp;
        else{
```

```
node *temp2 = head;
while(temp2->next!=head){
temp2 = temp2->next;}
temp->next = head;
temp2->next = temp;
}}
```

Adding a node to the front of a Circular Singly linked list

```
void insert_front(int x){

node *temp = new node;

temp->data = x;

temp->next = temp;

if(head==NULL)

head = temp;

else{

node *temp2 = head;

while(temp2->next!=head){

temp2 = temp2->next;}

temp->next = head;

head = temp;

temp2->next = temp;

}}
```

Adding a node to the left of a specific data in a Circular Singly linked list

```
void insert_left_y(int x, int y){

node *temp=new node;

temp->data=x;

temp->next=temp;

if(head==NULL)

head = temp;
```

```
else
if(head->data==y){
node *temp2 = head;
while(temp2->next!=head){
temp2 = temp2->next;}
temp->next = head;
head = temp;
temp2->next = temp;}
else{
node *temp2 = head;
node *temp3;
while(temp2->data!=y){
temp3 = temp2;
temp2 = temp2->next;}
temp->next = temp3->next;
temp3->next = temp;}}
```

Adding a node to the right of a specific data in a Circular Singly linked list

```
void insert_right_y(int x, int y){
node *temp=new node;
temp->data=x;
temp->next=temp;
if(head==NULL)
head = temp;
else{
node *temp2 = head;
while(temp2->data!=y){
temp2 = temp2->next;}
temp->next = temp2->next;
temp2->next = temp;
}}
```

Deleting a node from the end of a Circular Singly linked list

```
void delete_end(){

        node *temp, *temp2;

        if(head==NULL)

        cout <<"No data inside\n";

        else{

        temp = head;

        while(temp->next!=head)   {

        temp2 = temp;

        temp = temp->next;}

        temp2->next = temp->next;

        delete temp;}}
```

Deleting a node from the front of a Circular Singly linked list

```
        void delete_front(){

        node *temp;

        if(head==NULL)

        cout <<"No data inside\n";

        else {

        temp = head;

        node *temp2 = head;

        while(temp2->next!=head){

        temp2 = temp2->next;}

        temp2->next = head->next;

        head = head->next;

        delete temp;

        }}
```

Deleting any node using the search data from a Circular Singly linked list

```
void delete_any(int x){
node *temp, *temp3;
if(head==NULL)
cout <<"No data inside\n";
else
if(head->data==x){
temp = head;
node *temp2 = head;
while(temp2->next!=head){
temp2 = temp2->next;}
temp2->next = head->next;
head = head->next;
delete temp;}
else{
temp = head;
while(temp->data!=x){
temp3 = temp;
temp = temp->next;}
temp3->next = temp->next;
delete temp;
}}
```

Display the node from the Circular Singly linked list in a forward manner

```
void display(){
node *temp;
if(head==NULL)
```

```
        cout <<"No data inside\n";
        else{
        temp = head;
        while(temp->next!=head){
cout << temp->data << endl;
        temp = temp->next;}
        cout << temp->data << endl;
        }}
```

Chapter 4

Stack and queue

4.1 stack

This chapters describes the different types of stack and queue data structure. The basic operation of stacks and queues is described, and then variations on these basic structures are introduced, such as deques and priority queues. Implementations are provided using C++ arrays or linked lists.

1. Stacks

In the last handout, we have seen how pointers can be used to create different types of list implementation. In this handout, we will look at how lists are used. We can divide lists into two basic categories, according to the operations that are required to operate on the data in the list. These categories are *stacks* and *queues*.

A stack is a list data structure that can only be accessed at one of its ends. In other words, if we want to add a new value into the stack, or remove a value from the stack, we can only do so from one end of the list. Consider Figure 1. To begin with, in Figure 1a, the stack is empty. Then we *push* the value 6 onto the stack. The term *push* is commonly used to refer to the operation of adding a value to a stack data structure. Next we push the value 11 onto the stack, so 11 becomes the 'top' element of the stack, and 6 moves down to second place. In Figure 1c, we *pop* an element from the stack. To *pop* an element from a stack simply means to remove the top element. In this case, the value 11 is popped. Next the values 8 and then 5 are pushed onto the stack, so in Figure 1f the top element is 5. Finally the top element, 5, is popped from the stack. Stacks are often referred to as a *LIFO* structure: *Last In/First Out*. You can think of stacks as similar to a pile of trays in a cafeteria. Trays are put on the top of

the pile and removed from the top. The last tray put on the pile is the first one removed. A tray can only be taken if there are trays in the pile, and a tray can be added to the pile only if there is enough room, i.e. if the pile is not too high.

A stack is defined in terms of the operations that we need to manipulate its elements. The operations are as follows:

- *clear()* – Remove all elements from the stack.
- *isEmpty()* – Check to see if there are elements on the stack.
- *push(el)* – Put the element *el* on the top of the stack.
- *pop()* – Take the topmost element from the stack.
- *topEl()* – Return the topmost element in the stack without removing it.

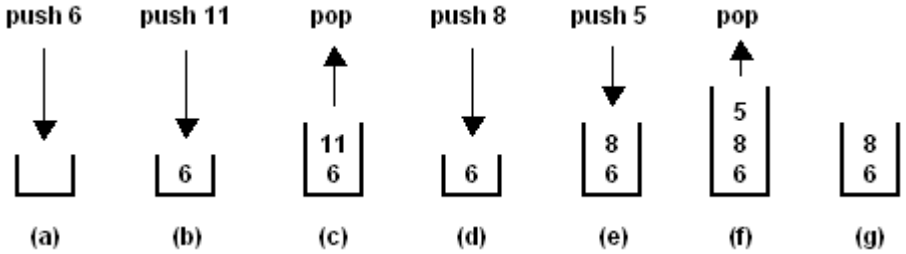


Figure 1 – The operation of a stack

When are stacks useful? One common application is in writing compilers. Whenever a function call is made, the code generated by the compiler must store the values of all local variables ready for when program execution returns from the function call. Within the called function, any number of nested function calls could be made, and for each of them the values of local variables (the *environment*) also needs to be stored. A stack is an ideal data structure for this task. Before each function call, the current environment is pushed onto a stack, and as each function finishes execution, the environment is popped from the stack. Consider the code below.

```
//*************** Function Calls.cpp  ***************
//     program to illustrate nested function calls

#include <iostream.h>

int f1 (int);
int f2 (int);

main () {
    int x = 3;
    cout << "f1(f2(3)) = " << f1(x) << endl;
}

int f1 (int a) {
    int p = a * 3;
```

```
    return f2(p + 2);
}

int f2 (int b) {
    int q = b * 2;
    return q - 1;
}
```

In the `main` function, the environment consists of the fact that the local variable `x` has the value 3. So when the function `f1` is called, this information is pushed onto the stack. Next, inside `f1`, the environment contains the information that `a` is equal to 3 and `p` is equal to 9, so when `f2` is called this information is pushed onto the stack. When execution of `f2` finishes, the topmost environment is popped from the stack, and when execution of `f1` finishes, the next environment is popped from the stack. In this way, the correct environment can be restored after each function call completes. This process is illustrated in Figure 2.

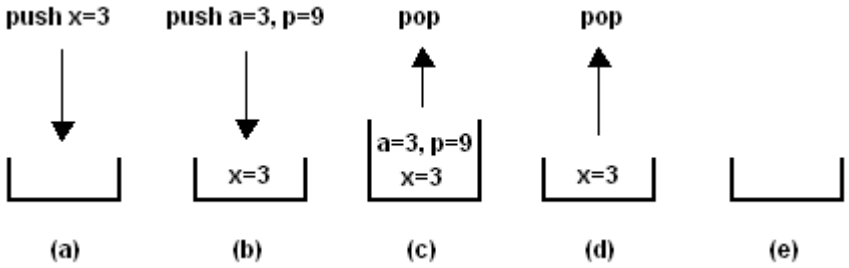


Figure 2 - The use of a stack in storing variable values in code execution

Now let us consider how to implement the stack data structure. One possible implementation is to use a C++ array. Figure 3 shows a generic stack class definition that uses templates to allow the programmer to create a stack of any size that can be used to store any type of object. This implementation provides all 5 of the basic stack operations described above. The data in the stack are stored in an array that is initially of size *capacity*. Once this maximum size is exceeded a new array of twice the size is allocated, and all of the data elements copied across to it.

What is the efficiency of this implementation? It is easy to see that popping an element from the stack is executed in constant time *O(1)*. Pushing an element onto the stack will also usually be executed in *O(1)*, but occasionally the array size needs to be increased. In this case copying across the existing data elements is more time-consuming, so in the worst case the push operation is executed in *O(n)*.

The array is not the only possibility for implementing a stack. The use of a dynamic data structure would improve the efficiency of the push operation, and would also eliminate wastage of space, since in the array implementation the capacity of the stack will often be significantly larger than the number of data elements stored. Figure 4 gives such an

implementation that uses the doubly linked list class we defined in Handout 2. In this implementation, the push and pop operations are both executed in *O(1)*.

2. Queues

A *queue* is also a type of list, but whereas with a stack the data elements are only added and removed from the same end of the list, with a queue the elements are always added to one end of the list, but removed from the other. You can think of a queue data structure as being like a queue of people waiting to use a payphone, or to be served at the bank. A queue is a *FIFO* structure: *First In/First Out*.

```
//******************  arrayStack.h  ******************
//       class for array implementation of stack

#ifndef ARRAY_STACK
#define ARRAY_STACK

template<class T, int capacity = 30>
class Stack {
public:
    Stack() {
        size = capacity;
        data = new T[size];
        n = 0;
    }
    void clear() {n = 0;}
    bool isEmpty() const {
        return (n == 0);
    }
    T& topEl() {
        if (n > 0) //only defined for non-empty stack
            return data[n - 1];
    }
    T pop() {
        if (n > 0) { //only defined for non-empty stack
            T el = data[n - 1];
            n--;
            return el;
        }
    }
    void push(const T& el) {
        if (n == size) {       // allocate new memory
            T *olddata = data; // and copy across data
            data = new T[size * 2];
            for (int i = 0; i < size; i++)
                data[i] = olddata[i];
            delete[] olddata;
            size *= 2;
        }
        data[n] = el; // add new element
        n++;
    }
private:
    T *data;
    int size, n;
};

#endif
```

Figure 3 – An array implementation of a stack data structure

```
//*********************  DLLStack.h  *******************
// class for doubly linked list implementation of stack

#include "DLList.h"

#ifndef DLL_STACK
#define DLL_STACK

template<class T>
class DLLStack {
public:
    DLLStack() {};
    void clear() {
        while (!data.isEmpty())
            data.deleteFromDLLTail();
    }
    bool isEmpty() {
        return data.isEmpty();
    }
    T topEl() {
        if (!data.isEmpty())       // only defined for
            return data.getTail(); // non-empty stack
   }
    T pop() {
        if (!data.isEmpty())
            return data.deleteFromDLLTail();
    }
    void push(const T& el) {
        data.addToDLLTail(el); // add new element
    }

private:
    DoublyLinkedList<T> data;
};

#endif
```

Figure 4 – A doubly-linked list implementation of a stack data structure

Queue operations are similar to stack operations. The following operations are needed to properly manage a queue:

- *clear()* – Remove all elements from the queue.
- *isEmpty()* – Check to see if there are elements in the queue.
- *enqueue(el)* – Put the element *el* at the end of the queue.
- *dequeue()* – Take the first element from the head of the queue.

- *firstEl()* – Return the first element in the queue without removing it.

A series of enqueue and dequeue operations is shown in Figure 5. This time, the new data elements are added to one end of the list (the right) and elements are removed from the other end (the left). For example, after enqueueing 6 and 11, the first dequeue operation (Figure 5c) removes the data element 6. If this were a stack data structure, then the last element to be entered (11) would have been removed. Similarly, at the dequeue in Figure 5f it is the 11 element that is removed, as this is at the head of the queue.

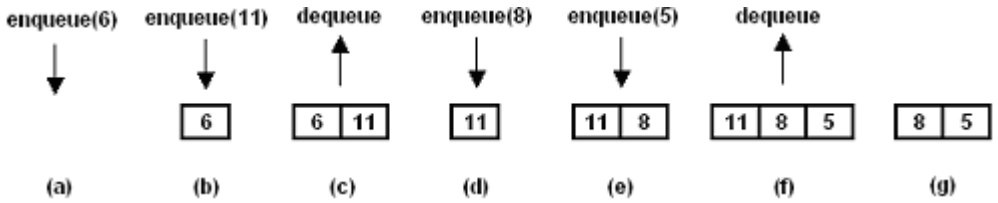


Figure 5 - The operation of a queue

A potential application of a queue data structure is for multitasking. Often an operating system will have a number of processes that all need to use the computer's CPU. A simple way of allocating CPU time is on a first-come-first-served basis – the first process to request CPU time is allowed to use it, and subsequent processes join a queue and are allowed to use the CPU when they reach the head of the queue.

Another common application of queues is in simulation. As an example, consider that the Commercial Bank of Ethiopia in Ambo have a number of clerks serving customers, with a single queue of customers waiting to be served. How many clerks should the bank employ? By observing the frequency of arrival of customers, and the amount of time taken to serve each customer, it is possible to write a simulation of the operation of the bank, using a queue data structure. This simulation will predict how many customers will be waiting in the queue on average, based on the number of clerks. By running the simulation for different numbers of clerks, the bank can decide on the optimal number of clerks required.

It is possible to implement a queue data structure using an array, in a similar way to the stack implementation described above. However, there are difficulties with such an implementation. Consider the sequence of enqueue and dequeue operations illustrated in Figure 6. Here we are using an array of fixed length 5 to implement the queue data structure. What happens when the final enqueue operation (Figure 6h) is requested? There is no space left to add the new data element at the end of the array, but there is free space at the beginning. These cells should not be wasted. Therefore we can add the new data to the beginning of the array. But now the beginning and end of the queue can potentially be anywhere in the array, so we need to remember where they are. Figure 7a shows the array after enqueueing the data element 2. The new element has been added to the beginning of the list but the *last* variable has been updated to indicate this fact. Sometimes it is easier to visualise such an array as a circular array, as in Figure 7b. Note that this is for visualisation

purposes only, it does not change the actual implementation.

We can now define the enqueue operation as follows:

- If the last element is in the last cell, and there are free cells at the beginning of the list, add the new value at the beginning of the list.
- If the last element is in any other position, then put the new value in the cell after the last value, if it is free.

An implementation of a queue using an array is given in Figure 8. Both the enqueue and dequeue operations are executed in constant time *O(1)*, but the queue is of limited length. Queues can also be implemented using dynamic data structures, which avoid the problems described above.

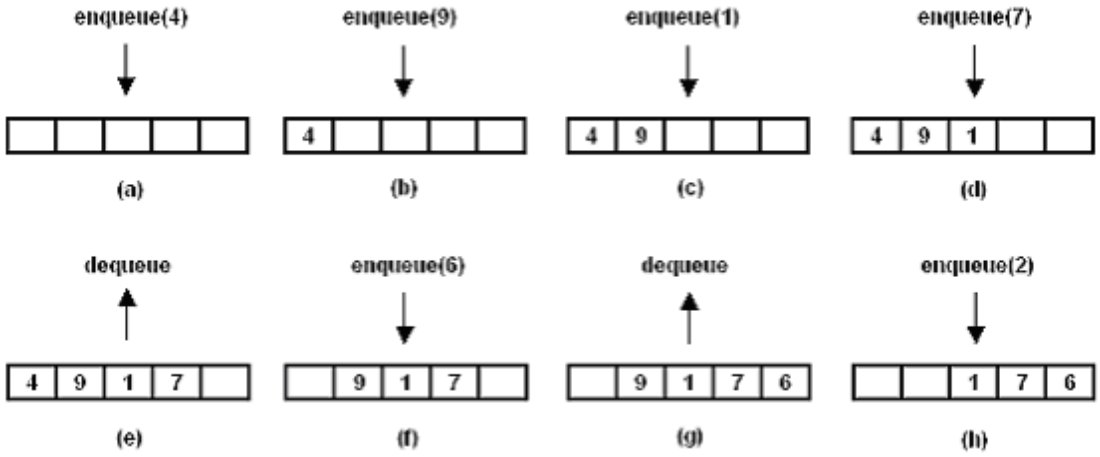


Figure 6 - The operation of a queue using a fixed-length array implementation

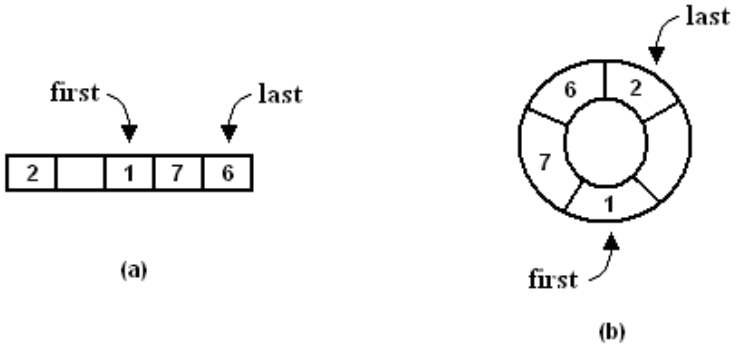


Figure 7 – Two different ways of visualising a fixed-length array implementation of a queue

```
//*************  genArrayQueue.h  *****************
//         queue implemented as an array

#ifndef ARRAY_QUEUE
#define ARRAY_QUEUE

template<class T, int size = 100>
class ArrayQueue {
public:
    ArrayQueue() {first = last = -1;}
    void enqueue(T);
    T dequeue();
    bool isFull() {
        return first == 0 && last == size-1 ||
               first == last + 1;
    }
    bool isEmpty() {return first == -1;}
private:
    int first, last;
    T storage[size];
};

template<class T, int size>
void ArrayQueue<T,size>::enqueue(T el) {
   if (!isFull())
        if (last == size-1 || last == -1) {
            storage[0] = el;
            last = 0;
            if (first == -1)
                first = 0;
        }
        else storage[++last] = el;
   else cout << "Full queue.\n";
}

template<class T, int size>
T ArrayQueue<T,size>::dequeue() {
    T tmp;
    tmp = storage[first];
    if (first == last)
         last = first = -1;
    else if (first == size-1)
         first = 0;
    else first++;
    return tmp;
}
```

```
#endif
```

Figure 8 – An array implementation of a queue data structure

3. Deques

A *deque* (pronounced like "deck") is a variation on the standard queue data structure. The word deque is an acronym derived from *double-ended queue*. Deques extend the basic queue data structure by allowing elements to be enqueued or dequeued from either end of the deque. The following operations are required to maintain a deque data structure:

- *clear()* – Remove all elements from the queue.
- *isEmpty()* – Check to see if there are elements in the queue.
- *enqueueHead(el)* – Put the element *el* at the head of the queue.
- *enqueueTail(el)* – Put the element *el* at the tail of the queue.
- *dequeueHead()* – Take the first element from the head of the queue.
- *dequeueTail()* – Take the last element from the tail of the queue.
- *head()* – Return the first element in the queue without removing it.
- *tail()* – Return the last element in the queue without removing it.

Figure 9 illustrates the operation of a deque. Note that we can think of a deque as a generalisation of the stack and queue data structures. If we only use the *enqueueHead()* and *dequeueHead()* operations then a deque behaves like a stack. If we only use the *enqueueHead()* and *dequeueTail()* operations then it behaves like a queue.

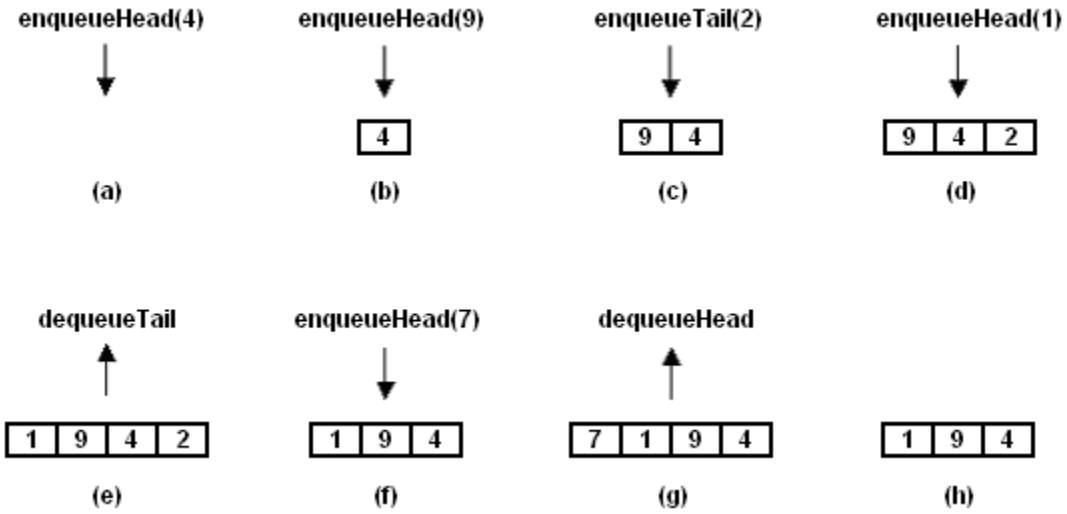


Figure 9 – The operation of a deque

4. Priority Queues

Often, the basic queue data structure is not appropriate. For example, consider the multitasking application described in Section 2 above. With the standard queue data structure

processes are executed by the CPU on a first-come-first-served basis. Sometimes this is what is required. However, occasionally a process will arrive at the end of the queue that is more urgent than the processes ahead of it. For example, this may be an error-handling process. In this case, we want the urgent process to jump the queue, and be executed before the existing processes in the queue. In situations like this, a variation on the queue, called a *priority queue*, is needed. In priority queues, elements arrive in an arbitrary order, but are enqueued with information about their priority. Elements are dequeued according to their priority and their current queue position.

Priority queues are usually implemented using dynamic data structures. A number of different possible implementations exist, but we will not cover them in this course.

5. The Standard Template Library

Although implementations of lists, stacks, queues and other data structures have been provided in these handouts, there is an existing C++ library of templated data structures, called the *Standard Template Library* (STL). The STL contains built-in classes for storing lists, stacks, queues, deques and priority queues. Many C++ textbooks contain details of the STL. For example, see "C++ Program Design" by Cohoon & Davidson, p486.

Summary of Key Points

The following points summarize the key concepts in this handout:

- A *stack* is a list data structure that can only be accessed at one end.
- Data elements are added to a stack using the *push* operation, and removed using the *pop* operation.
- A stack is a *LIFO (Last In/First Out)* data structure.
- Stacks can be implemented using either C++ arrays or linked lists.
- A *queue* is a list data structure in which data elements are added to one end of the list, and removed from the other.
- A queue is a *FIFO (First In/First Out)* data structure.
- Queues can be implemented using either C++ arrays or linked lists.
- A *deque (double-ended queue)* is a queue in which data elements can be added or removed from either end of the list.
- A *priority queue* is a queue in which data elements are removed according to their priority.
- The *Standard Template Library* in C++ contains templated classes for lists, stacks, queues, deques and priority queues.

Exercises

For the following exercises, all necessary source code can be found on the course intranet page. Solutions will also be made available after classes for this chapter.

1) Write a program that reads in a sequence of characters from the keyboard (up to the first newline character), and prints them to the screen in reverse order. Use the STL `stack` class in your implementation.

2) Write a program that checks if a word typed in at the keyboard is a *palindrome*. A word is a palindrome if it is the same when read in reverse, e.g. "madam", "gag", etc. Use the STL `stack` and `queue` classes in your implementation.

3) Write a C++ class that implements a queue data structure using a doubly linked list. The class should contain member functions to perform all the basic queue operations as listed in Section 2. You can reuse, and modify if necessary, the doubly linked list class given in Handout 2, or the code you developed in exercises 3 and 4 in Handout 2.

4) Write a C++ class that implements a deque data structure using a doubly linked list. The class should contain member functions to perform all the basic deque operations as listed in Section 3. You can reuse, and modify if necessary, the doubly linked list class given in Handout 2, or the code you developed in exercises 3 and 4 in Handout 2.

5) Write a C++ class that implements a priority queue data structure. The queue should store a sequence of character values, each of which has an associated integer priority value. The class should contain member functions to add a new value to the queue, to check if the queue is empty, to clear the queue, and to remove the highest priority element from the queue. If there are a number of elements with the same priority, then the one nearest the front of the queue should be removed first. Use doubly linked lists in your implementation.

There are two classifications of data structures:

Linear Data Structures: A data structure is said to be linear, if its elements form a sequence or linear list.

Example: Arrays, linked lists, Stacks, and Queues.

Non-linear Data Structures: A Data Structure is said to be non-linear, if its elements do not form a sequence.

Example: Trees and Graphs.

TREE

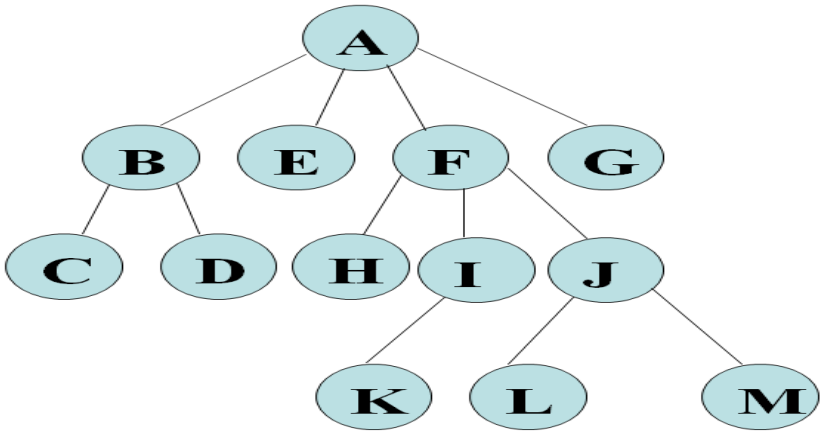
A tree is a set of nodes and edges that connect pairs of nodes.

Rooted tree has the following structure:

- One node distinguished as root.
- Every node C except the root is connected from exactly other node P.
- P is C's parent, and C is one of P's children.
- There is a unique path from the root to each node.
- The number of edges in a path is the length of the path.

Tree Terminologies

Consider the following tree. ABEFGCDHIJKLM.



Root: a node without a parent. ==> A

Internal node: a node with at least one child. ==>A, B, F, I, J

External (leaf) node: a node without a child. ==> C, D, E, H, K, L, M, G

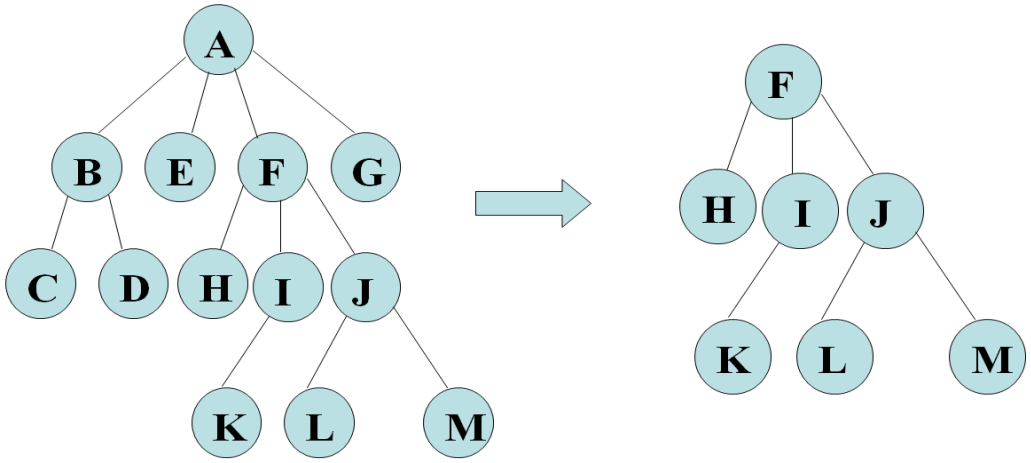
Ancestors of a node: parent, grandparent, grand-grandparent, etc of a node. Ancestors of K ==> A, F, I

Descendants of a node: children, grandchildren, grand-grandchildren etc of a node. Descendants of F ==> H, I, J, K, L, M

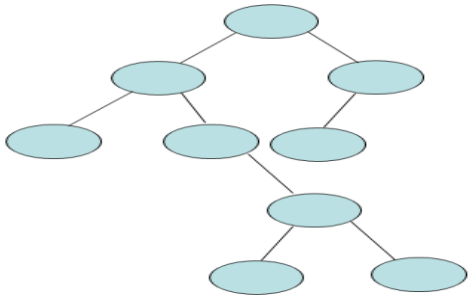
Depth of a node: number of ancestors or length of the path from the root to the node. Depth of H ==> 2

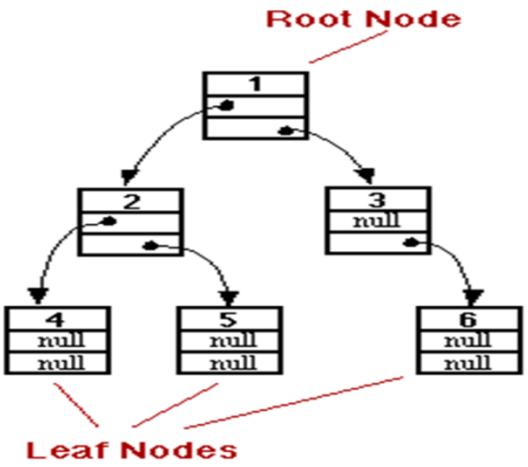
Height of a tree: depth of the deepest node. ==> 3

Subtree: a tree consisting of a node and its descendants.

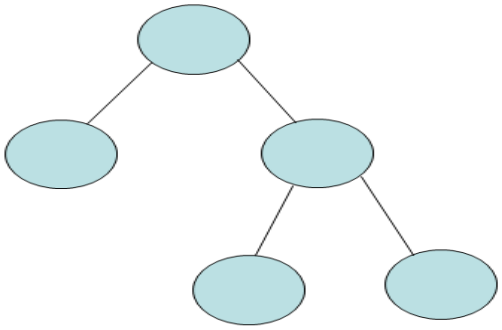


Binary tree: a tree in which each node has at most two children called left child and right child.

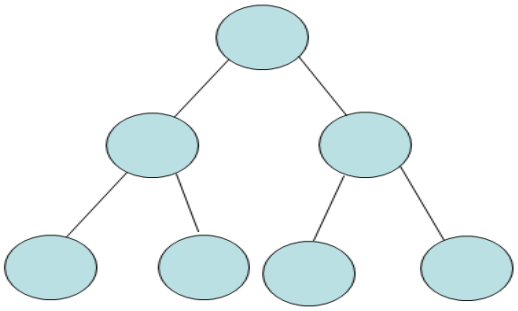




Full binary tree: a binary tree where each node has either 0 or 2 children.



Balanced binary tree: a binary tree where each node except the leaf nodes has left and right children and all the leaves are at the same level.

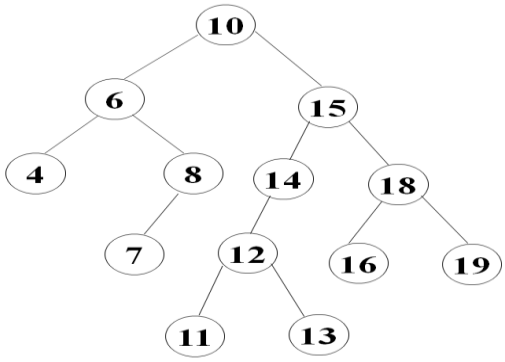


Complete binary tree: a binary tree in which the length from the root to any leaf node is either h or h-1. where h is the height of the tree. The deepest level should also be filled from left to right.

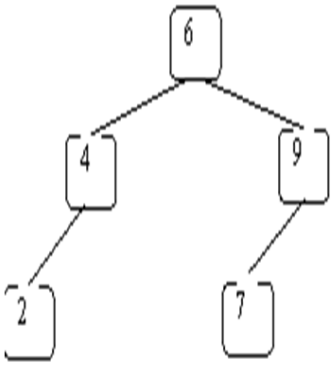
Binary search tree (ordered binary tree)

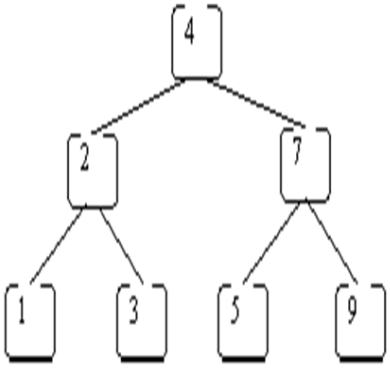
- A binary tree that may be empty, but if it is not empty it satisfies the following:
- Every node has a key and no two elements have the same key.
- The keys in the right subtree are larger than the key in the root.
- The keys in the left subtree are smaller than the key in the root.
- The left and the right subtrees are also binary search trees.

Examples of Binary Search Tree.

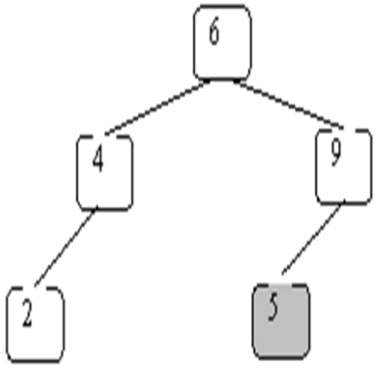


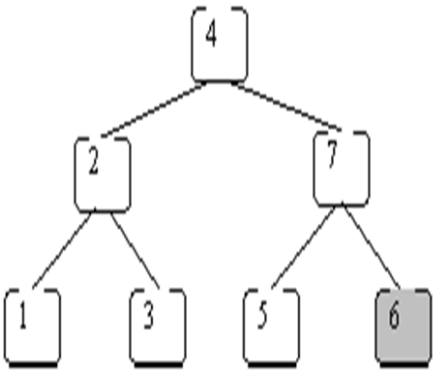
Here are some Binary Search Trees in which each node just stores an integer key:





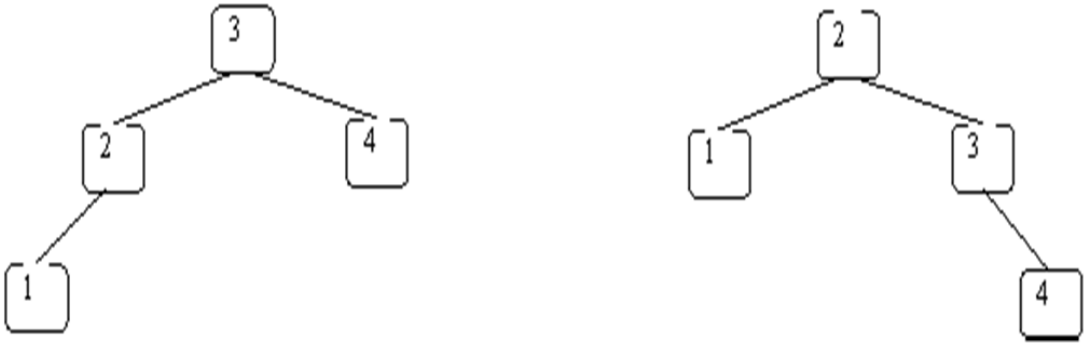
These are not Binary Search Trees:





Note that more than one Binary Search Tree can be used to store the same set of key values.

For example, both of the following are BSTs that store the same set of integer keys:



Operations on Binary Search Tree

The following operations can be implemented efficiently using a Binary Search Tree:

- Insert a key value.
- Determine whether a key value is in the tree.
- Remove a key value from the tree.
- Print all of the key values in sorted order.

Data Structure of a Binary Tree

Syntax:

```
struct DataModel {

Declaration of data fields DataModel * Left, *Right;

};

DataModel *RootDataModelPtr=NULL;
```

Example:

```
struct Node {

int Num;

Node * Left, *Right;

};
```

```
Node *RootNodePtr=NULL;
void InsertBST( ){
Node *InsNodePtr;
InsNodePtr = new Node;
cout << "Enter The Number" << endl;
cin>>InsNodePtr->Num;
InsNodePtr->Left=NULL;
InsNodePtr->Right=NULL;
if(RootNodePtr== NULL)
RootNodePtr=InsNodePtr;
else
{
Node *NP=RootNodePtr;
int Inserted=0;
while(Inserted ==0){
if(NP->Num > InsNodePtr->Num){
if(NP->Left == NULL){
NP->Left = InsNodePtr;
Inserted=1;}
else
NP = NP->Left;}
else {
if(NP->Right == NULL){
NP->Right = InsNodePtr;
Inserted=1;}
else
```

```
NP = NP->Right;

}}}}
```

Traversing (Visiting) in a BST

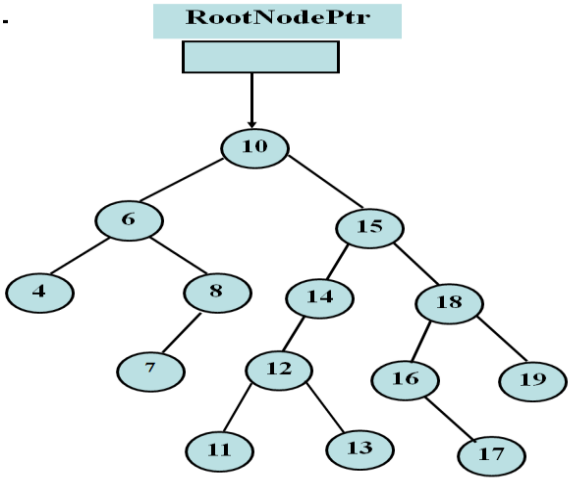
Binary search tree can be traversed in three ways.

Preorder traversal:- traversing binary tree in the order of parent, left and right.

Inorder traversal:- traversing binary tree in the order of left, parent and right.

Postorder traversal:- traversing binary tree in the order of left, right and parent.

Example:



Preorder traversal: 10, 6, 4, 8, 7, 15, 14, 12, 11, 13, 18, 16, 17, 19

Inorder traversal: 4, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 Used to display nodes in ascending order.

Postorder traversal: 4, 7, 8, 6, 11, 13, 12, 14, 17, 16, 19, 18, 15, 10

Application of binary tree traversal

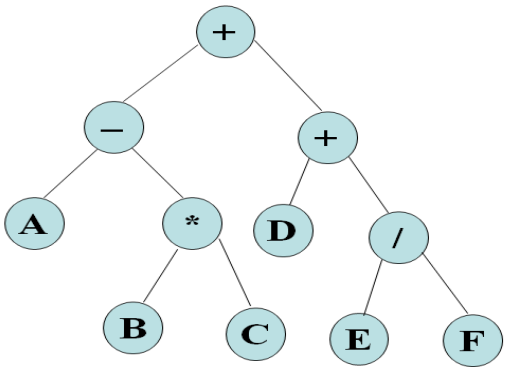
Store values on leaf nodes and operators on internal nodes:

Preorder traversal: - used to generate mathematical expression in prefix notation.

Inorder traversal: - used to generate mathematical expression in infix notation.

Postorder traversal: - used to generate mathematical expression in postfix notation.

Example:



- Preorder traversal: + – A * B C + D / E F ==> Prefix notation
- Inorder traversal: A – B * C + D + E / F ==> Infix notation
- Postorder traversal: A B C * – D E F / + + ==> Postfix notation

Preorder traversal

1. Process the value in the root (e.g. print the root value).

2. Traverse the left subtree with a preorder traversal.

3. Traverse the right subtree with a preorder traversal.

Inorder traversal :-prints the node values in ascending order:

1. Traverse the left subtree with an inorder traversal.

2. Process the value in the root (e.g. print the root value).

3. Traverse the right subtree with an inorder traversal.

Postorder traversal

1. Traverse the left subtree with a postorder traversal.

2. Traverse the right subtree with a postorder traversal.

3. Process the value in the root (e.g. print the root value).

Implementation of the Preorder traversals

```
void Preorder (Node *RootNodePtr){
if(RootNodePtr != NULL) {
cout << RootNodePtr->Num << endl; // or any operation on the node
Preorder(RootNodePtr->Left);
Preorder(RootNodePtr->Right);
}}
```

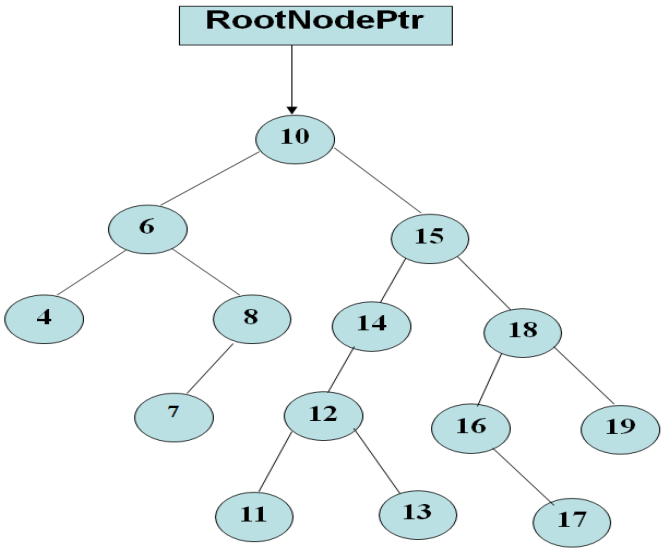
Implementation of the Inorder traversals

```
void Inorder (Node * RootNodePtr){
if(RootNodePtr != NULL){
Inorder(RootNodePtr->Left);
cout << RootNodePtr->Num << endl; // or any operation on the node
Inorder(RootNodePtr->Right);
}}
```

Implementation of the Postorder traversals

```
void Postorder(Node *RootNodePtr){
if(RootNodePtr != NULL){
Postorder(RootNodePtr->Left);
Postorder(RootNodePtr->Right);
cout << RootNodePtr->Num << endl; // or any operation on the node
}}
```

To search a node (whose Num value is X) in a binary search tree (whose root node is pointed by RootNodePtr). One of the three traversal methods can be used.

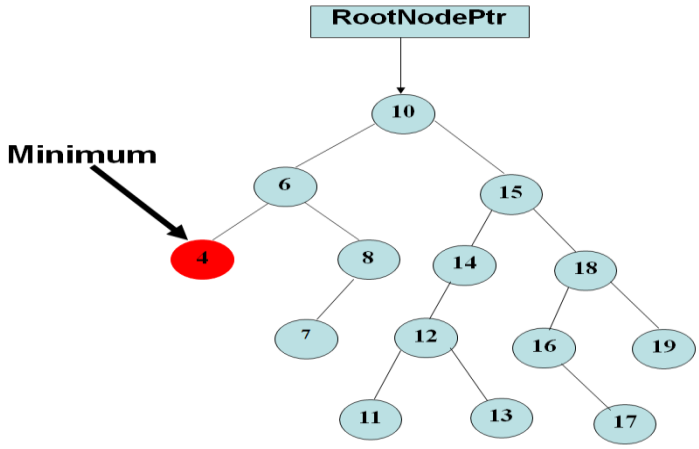


Implementation:

```
int SearchBST (
Node *RootNodePtr, int X){
if(RootNodePtr == NULL)
return 0; // 0 means (false, not found).
else
if(RootNodePtr ->Num == X)
return 1; // 1 means (true, found).
else if(RootNodePtr ->Num > X)
return(SearchBST(RootNodePtr ->Left, X));
else
return(SearchBST(RootNodePtr ->Right, X));
}
```

Finding Minimum value in a Binary Search Tree

We can get the minimum value from a Binary Search Tree, by locating the left most node in the tree. Then after locating the left most node, we display the value of that node.

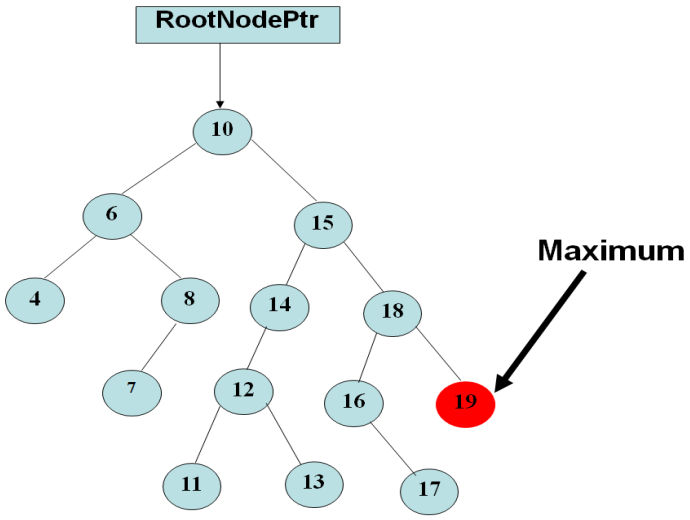


Implementation:

```
int findMin(Node *RootNodePtr){

if(RootNodePtr == NULL)

return -1;

else

if(RootNodePtr ->Left == NULL)

return RootNodePtr ->Num;

else

return findMin(RootNodePtr ->Left);

}
```

Finding Maximum value in a Binary Search Tree

We can get the maximum value from a Binary Search Tree, by locating the right most node in the tree. Then after locating the right most node, we display the value of that node.



Implementation:

```
int findMax(Node *RootNodePtr){

if(RootNodePtr == NULL)

return -1;

else

if(RootNodePtr ->Right == NULL)

return RootNodePtr ->Num;

else

return findMax(RootNodePtr ->Right);

}
```

Graph

A graph is a mathematical structure consisting of a set of vertices and a set of edges connecting the vertices.

Formally: G = (V,E), where V is a set and E is subset of V X V .

We can choose between two standard ways to represent a graph G = (V, E):

- As a collection of adjacency lists or
- As an adjacency matrix.

Either way applies to both directed and undirected graphs.

Because the adjacency-list representation provides a compact way to represent **sparse** graphs, those for which |E| is much less than $|V|^2$, it is usually the method of choice.

Most of the graph algorithms presented in this lesson assume that an input graph is represented in adjacency list form. We may prefer an adjacency-matrix representation, however, when the graph is **dense,** |E| is close to $|V|^2$ or when we need to be able to tell quickly if there is an edge connecting two given vertices.

For example, two of the all-pairs:

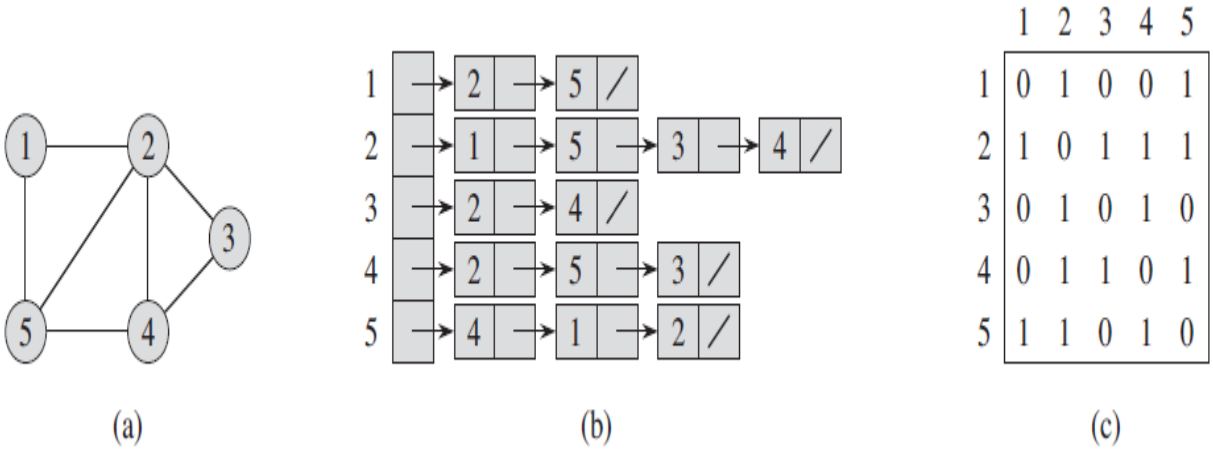


Figure 1Two representations of an undirected graph. (a) An undirected graph G with 5 vertices and 7 edges. (b) An adjacency-list representation of G. (c) The adjacency-matrix representation of G.

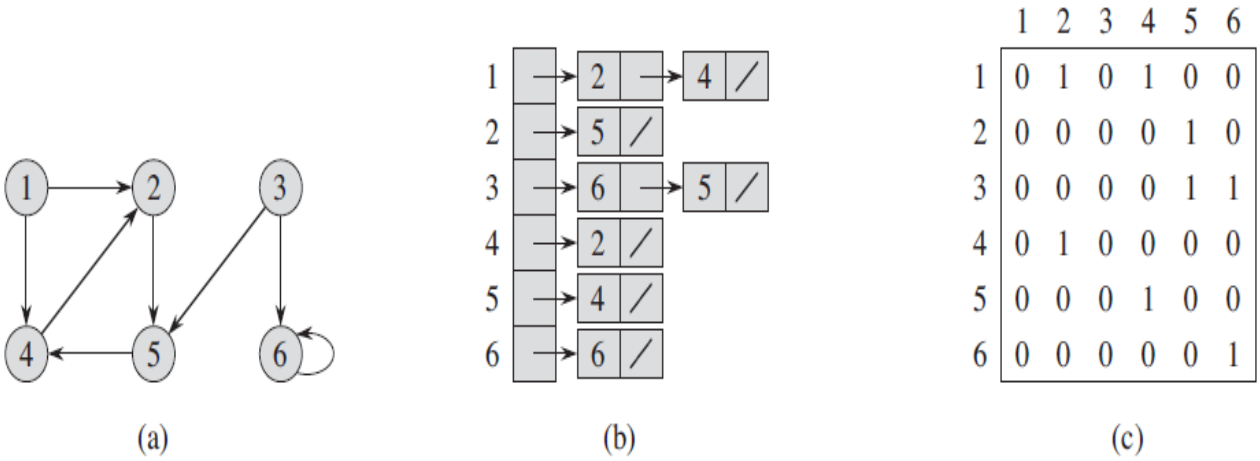


Figure 2 Two representations of a directed graph. (a) A directed graph G with 6 vertices and 8 edges. (b) An adjacency-list representation of G. (c) The adjacency-matrix representation of G.

Directed and Undirected Graph

G = (V,E) **undirected** if for all v,w € V : (v,w) € E < == > (w, v) € E. Otherwise directed.

A **directed** graph:

G = (V,E) with vertex set V = {0,1,2,3,4,5, 6} and edge set.

E = {(0, 2), (0, 4), (0, 5), (1, 0), (2, 1), (2, 5), (3, 1), (3, 6), (4, 0), (4, 5), (6, 3), (6, 5)}

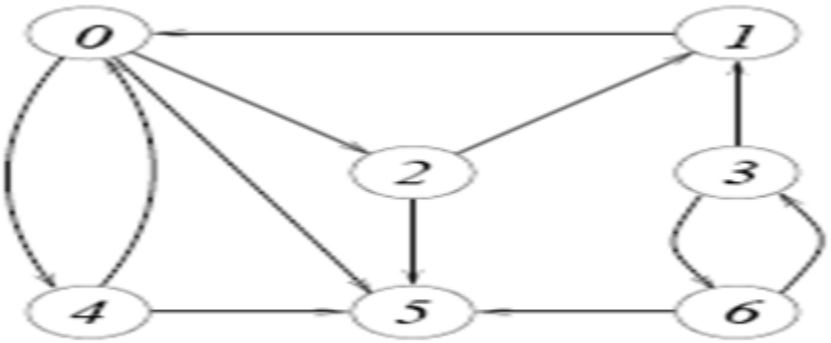


Figure 3. directed graph

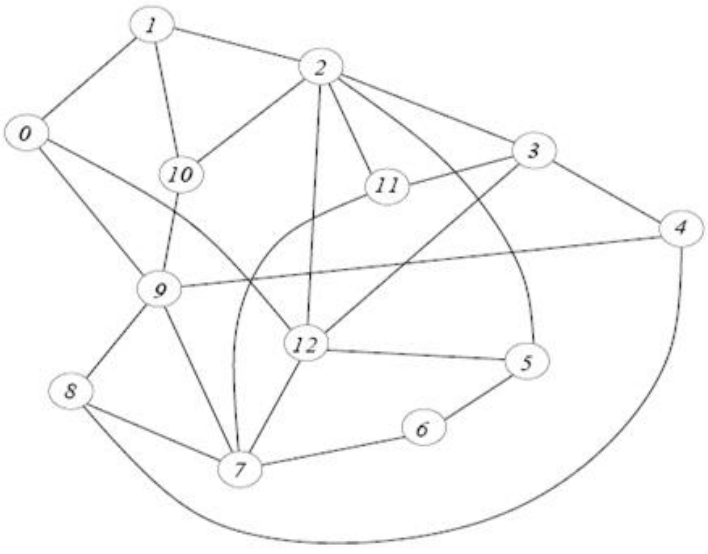


Figure 4. An undirected graph

Examples:

Computer Networks, Vertices represent computers and edges represent network connections (cables) between them.

For example:

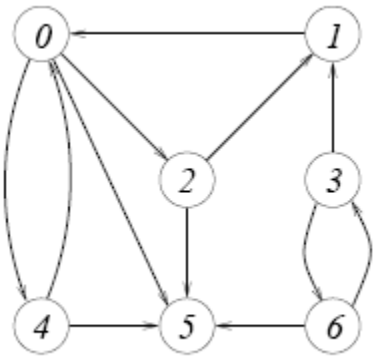
- we might be using a graph to represent a computer network (such as the Internet), and we might be interested in finding the fastest way to route a data packet between two computers.
- The World Wide Web. Vertices represent web pages, and edges represent hyperlinks.
- Flowcharts. Vertices represent boxes and edges represent arrows.

Let $G = (V, E)$ be a graph with n vertices. Vertices of G numbered $0, \ldots, n-1$.

The **adjacency matrix** of G is the $n \times n$ matrix $A = (a_{ij})_{0 \le i,j \le n-1}$ with

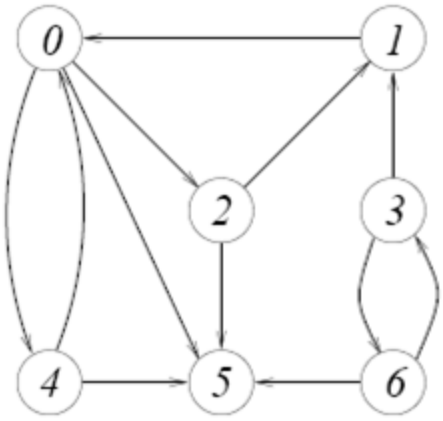
$$a_{ij} = \begin{cases} 1 & \text{if there is an edge from vertex } i \text{ to vertex } j \\ 0 & \text{otherwise.} \end{cases}$$

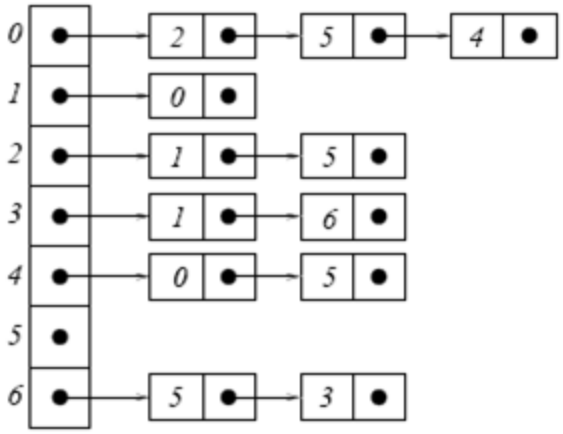
Example of adjacency matrix data structure



$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Examples of the adjacency list data structure Array with one entry for each vertex v, which is a list of all vertices adjacent to v.





CHAPTER FIVE
HASHING

This chapter gives an introduction to the subject of hashing. Common hash functions such as division, folding, mid-square function, extraction and radix transformation are discussed. In addition, a number of collision resolution techniques are described, such as open addressing, chaining and bucketing.

1. Hashing

All of the searching techniques we have seen so far operate by comparing the value being searched for with the values of a *key* value of each element. For example, when searching for an integer *val* in a binary search tree, we compare *val* with the integer (the *key*) stored at each node we visit. Such searching techniques vary in their complexity, but will always be more than *O(1).*

Hashing is an alternative way of storing data that aims to greatly improve the efficiency of search operations. With hashing, when adding a new data element, the key itself is used to directly determine the location to store the element. Therefore, when searching for a data element, instead of searching through a sequence of key values to find the location of the data we want, the key value itself can be used to directly determine the location in which the data is stored. This means that the search time is reduced from *O(n),* as in sequential search, or *O(log n),* as in binary search, to *O(1),* or constant complexity. Regardless of the number of elements stored, the search time is the same.

The question is, how can we determine the position to store a data element using only its key value? We need to find a function *h* that can transform a key value *K* (e.g. an integer, a string, etc.) into an index into a table used for storing data. The function *h* is called a *hash function*. If *h* transforms different keys into different indices it is called a *perfect hash function*. (A non-perfect hash function may transform two different key values into the same index.)

Consider the example of a compiler that needs to store the values of all program variables. The key in this case is the name of the variable, and the data to be stored is the variable's value. What hash function could we use? One possibility would be to add the ASCII codes of every letter in the variable name and use the resulting integer to index a table of values. But in this case the two variables abc and cba would have the same index. This problem is known as *collision* and will

be discussed later in this handout. The worth of a hash function depends to a certain extent on how well it avoids collisions.

2. Hash Functions

Clearly there are a large number of potential hash functions. In fact, if we wish to assign positions for n items in a table of size m, the number of potential hash functions is m^n, and the number of perfect hash functions is $\frac{m!}{(m-n)!}$. Most of these potential functions are not of practical use, so this section discusses a number of popular types of hash function.

2.1. Division

A hash function must guarantee that the value of the index that it returns is a valid index into the table used to store the data. In other words, it must be less than the size of the table. Therefore an obvious way to accomplish this is to perform a modulo (remainder) operation. If the key *K* is a number, and the size of the table is *Tsize*, the hash function is defined as *h(K) = K mod TSize*. Division hash functions perform best if the value of *TSize* is a prime number.

2.2. Folding

Folding hash functions work by dividing the key into a number of parts. For example, the key value 123456789 might be divided into three parts: 123, 456 and 789. Next these parts are combined together to produce the target address. There are two ways in which this can be done: *shift folding* and *boundary folding*.

In shift folding, the different parts of the key are left as they are, placed underneath one another, and processed in some way. For example, the parts 123, 456 and 789 can be added to give the result 1368. To produce the target address, this result can be divided modulo *TSize*.

In boundary folding, alternate parts of the key are left intact and reverse. In the example given above, 123 is left intact, 456 is reversed to give 654, and 789 is left intact. So this time the numbers 123, 654 and 789 are summed to give the result 1566. This result can be converted to the target address by using the modulo operation.

2.3. Mid-Square Function

In the mid-square method, the key is squared and the middle part of the result is used as the address. For example, if the key is 2864, then the square of 2864 is 8202496, so we use 024 as the address, which is the middle part of8202496. If the key is not a number, it can be pre-processed to convert it into one.

2.4. Extraction

In the extraction method, only a part of the key is used to generate the address. For the key 123456789, this method might use the first four digits (1234), or the last four (6789), or the first two and last two (1289). Extraction methods can be satisfactory so long as the omitted portion of the key is not significant in distinguishing the keys. For example, at Mekelle University many student ID numbers begin with the letters "RDG", so the first three letters can be omitted and the following numbers used to generate the key using one of the other hash function techniques.

2.5. Radix Transformation

If *TSize* is 100, and a division technique is used to generate the target address, then the keys 147 and 247 will produce the same address. Therefore this would not be a perfect hash function. The radix transformation technique attempts to avoid such collisions by changing the number base of the key before generating the address. For example, if we convert the keys 147_{10} and 247_{10} into base 9, we get 173_9 and 304_9. Therefore, after a modulo operation the addresses used would be 47 and 04. Note, however, that radix transformation does not completely avoid collisions: the two keys 147_{10} and 66_{10} are converted to 173_9 and 73_9, so they would both hash to the same address, 73.

3. Collision Resolution

If the hash function being used is not a perfect hash function (which is usually the case), then the problem of collisions will arise. Collisions occur when two keys hash to the same address. The chance of collisions occurring can be reduced by choosing the right hash function, or by increasing the size of the table, but it can never be completely eliminated. For this reason, any hashing system should adopt a *collision resolution* strategy. This section examines some common strategies.

3.6. Open Addressing

In open addressing, if a collision occurs, an alternative address within the table is found for the new data. If this address is also occupied, another alternative is tried. The

sequence of alternative addresses to try is known as the *probing sequence*. In general terms, if position *h(K)* is occupied, the probing sequence is

$$norm(h(K) + p(1)), norm(h(K) + p(2)), \ldots, norm(h(K) + p(i)), \ldots$$

where function *p* is the probing function and *norm* is a normalisation function that ensures the address generated is within an acceptable range, for example the modulo function.

The simplest method is *linear probing*. In this technique the probing sequence is simply a series of consecutive addresses; in other words the probing function *p(i) = i*. If one address is occupied, we try the next address in the table, then the next, and so on. If the last address is occupied, we start again at the beginning of the table. Linear probing has the advantage of simplicity, but it has the tendency to produce *clusters* of data within the table. For example, Figure 1 shows a sequence of insertions into a hash table using the following key/value pairs:

Key	Value
15	A
2	B
33	C
5	D
19	E
22	F
9	G
32	H

The first three insertions (A, B and C) do not result in collisions. However, when data D is inserted it hashes to the address 5, which is currently occupied by A, so it is placed in the next address. Similarly, when data F is inserted at address 2 it collides with B, so we try address 3 instead. Here it collides with C, so we have to place it at address 4. Data G also collides with E at address 9, so because 9 is the last address in the table we place it at address 1. Finally data H collides with 5 different elements before being successfully placed at address 7.

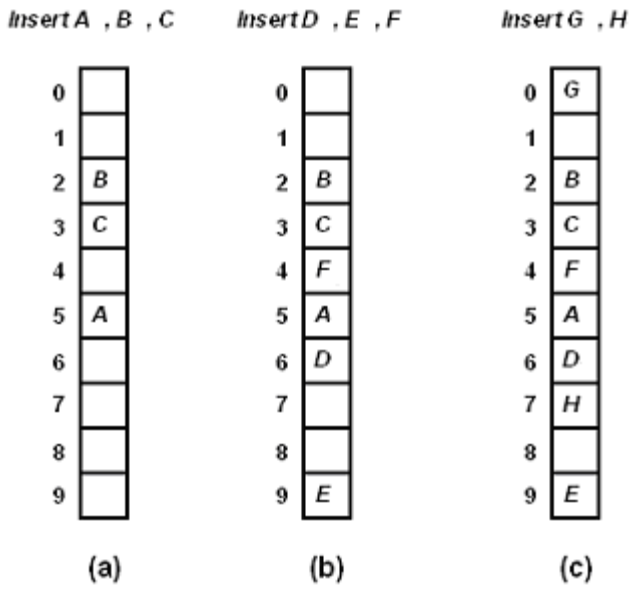


Figure 1 – Collision resolution using linear probing.

We can see in Figure 1 that there is a cluster of 6 elements (from addresses 2 to 7) stored next to each other. The problem with clusters is that the probability of a collision for a key is dependent on the address that it hashes to. Clustering can be avoided by using a more careful choice of probing function *p*. One possible choice is to use the sequence of addresses

$$h(K)+i^2, h(K)-i^2, \text{ for } i = 1, 2, \ldots, (Tsize - 1) / 2.$$

Including the original attempt to hash *K*, this formula results in the sequence *h(K), h(K) + 1, h(K) – 1, h(K) + 4, h(K) – 4*, etc. All of these addresses should be divided modulo *Tsize*. For example, for the H_2 data in Figure 1, we first try address 2, then address 3 (2 + 1), and then address 1 (2 – 1), where the data is successfully placed. This technique is known as *quadratic probing*. Quadratic probing results in fewer clusters than linear probing, but because the same probing sequence is used for every key, sometimes clusters can build up away from the original address. These clusters are known as *secondary clusters*.

Another possibility, which avoids the problem of secondary clusters, is to use a different probing sequence for each key. This can be achieved by using a random number generator seeded by a value that is dependent on the key. Remember that random number generators always require a seed value, and if the same seed is used the same sequence of 'random' numbers will be generated. So if, for example, the value of the key (if it is an integer), were to be used, each different key would generate a different sequence of probes, thus avoiding secondary clusters.

Another way to avoid secondary clusters is to use *double hashing*. Double hashing uses two different hashing functions: one to find the primary position of a key, and another for resolving conflicts. The idea is that if the primary hashing function, *h(K)*, hashes two

keys K_1 and K_2 to the same address, then the secondary hashing function, $h_p(K)$, will probably not. The probing sequence is therefore

$$h(K), h(K) + h_p(K), h(K) + 2 \cdot h_p(K), \ldots, h(K) + i \cdot h_p(K), \ldots$$

Experiments indicate that double hashing generally eliminates secondary clustering, but using a second hash function can be time-consuming.

3.7. Chaining

In *chaining*, each address in the table refers to a list, or *chain*, of data values. If a collision occurs the new data is simply added to end of the chain. Figure 2 shows an example of using chaining for collision resolution.

Provided that the lists do not become very long, chaining is an efficient technique. However, if there are many collisions the lists will become long and retrieval performance can be severely degraded. Performance can be improved by ordering the values in the list (so that an exhaustive search is not necessary for unsuccessful searches) or by using self-organising lists.

An alternative version of chaining is called *coalesced hashing*, or *coalesced chaining*. In this method, the link to the next value in the list actually points to another table address. If a collision occurs, then a technique such as linear probing is used to find an available address, and the data is placed there. In addition, a link is placed at the original address indicating where the next data element is stored. Figure 3 shows an example of this technique. When the keys D5 and F2 collide Figure 3b, linear probing is used to position the keys, but links from their original hashed addresses are maintained. Variations on coalesced hashing include always placing colliding keys at the end of the table, or storing colliding keys in a special reserved area known as the *cellar*. In both cases a link from the original hashed address will point to the new location. The advantage of coalesced hashing is that it avoids the need to make a sequential search through the table for the required data in the event of collisions.

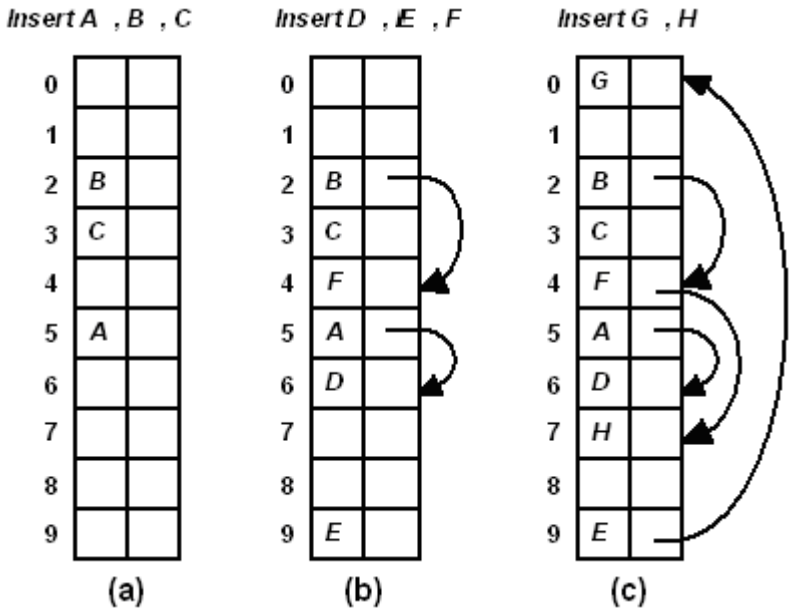


Figure 3 – Collision resolution using coalesced hashing.

3.8. Bucket Addressing

Bucket addressing is similar to chaining, except that the data are stored in a *bucket* at each table address. A bucket is a block of memory that can store a number of items, but not an unlimited number as in the case of chaining.

Bucketing reduces the chance of collisions, but does not totally avoid them. If the bucket becomes full, then an item hashed to it must be stored elsewhere. Therefore bucketing is commonly combined with an open addressing technique such as linear or quadratic probing. Figure 4 shows an example of bucketing that uses a bucket size of 3 elements at each address.

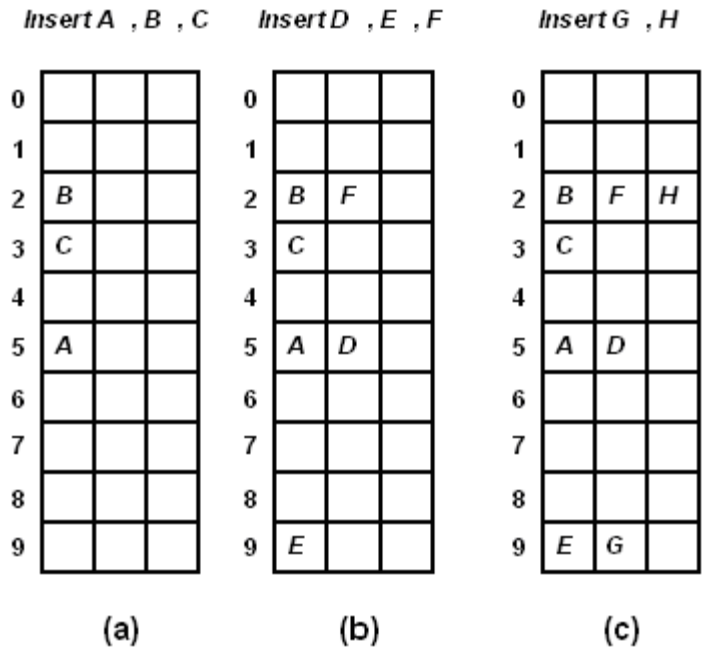


Figure 4 – Collision resolution using bucketing.

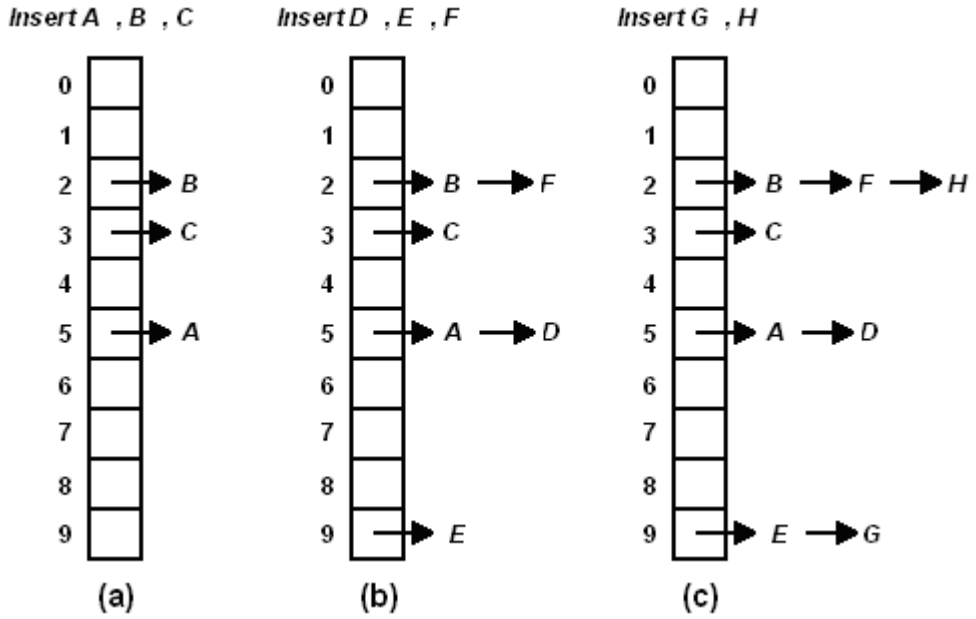


Figure 2 – Collision resolution using chaining.

Summary of Key Points

The following points summarize the key concepts in this handout:

- *Hashing* is a data storage technique that aims to improve the efficiency of search operations.
- Using hashing, a *hash function h(K)* is used to determine the address within a table at which a key *K* will be stored.
- A *perfect hash function* is one that will generate different addresses for different keys.
- If two keys hash to the same address a *collision* will occur.
- The simplest hash function is to use a modulo operation using the number of addresses in the table as the divisor.
- *Folding* hash functions work by dividing the key into a number of parts and then combining them to produce the target address.
- In *shift folding* the different parts of the key are left intact before being combined.
- In *boundary folding* alternate parts of the key are reversed before combination.
- In the *mid-square* hash function, the key is squared and the middle part of the result is used as the address.
- In the *extraction* method, only a part of the key is used to generate the address.
- In the *radix transformation* technique, the number base of the key is changed to try to avoid collisions.
- *Open addressing* attempts to resolve collisions by finding an alternative address at which to store collided keys.
- The *probing sequence* is the series of addresses tried by an open addressing scheme.
- *Linear probing* uses a probing sequence consisting of consecutive addresses in the table.
- *Quadratic probing* used a probing sequence of the form *h(K), h(K) + 1, h(K) – 1, h(K) + 4, h(K) – 4*, etc.
- A *cluster* is a set of keys that are stored in addresses in the same part of the table.
- A *primary cluster* occurs when many keys hash to the same (or similar) primary address.
- A *secondary cluster* occurs when many keys hash to the same (or similar) alternative address.
- Secondary clusters can be avoided by using a random number technique or by using *double hashing*.
- In double hashing a different hash function is used to generate the probing sequence.
- In *chaining*, each table address refers to a linked list of data elements.
- In *coalesced chaining*, or *coalesced hashing*, collided keys are stored in an alternative position in the table but a link from the original hashed address is maintained.
- In *bucket addressing*, each table address contains a *bucket* capable of storing multiple data elements.

Exercises

1) Write a C++ program to implement a simple division hashing scheme. The program should read in a sequence of *key-value* pairs from the keyboard – the *key* should be a positive integer and the *value* should be a string. Each *key-value* pair should be stored in a table of size 100. Use linear probing for collision resolution. After the user has finished entering *key-value* pairs (e.g. they could enter a negative key), they should be able to retrieve a sequence of values by entering their keys.

2) Update the program you wrote in (1) to make it use quadratic probing instead of linear probing.

References: C++ Programming: Program Design Including Data Structures, Fifth Edition