L11: Improving the Problem not the Code

Finding Max & Min of a number

Comparing two adjacent numbers and finding the maximum and minimum
e.g. 5 7 8 9 10 1 3
Normally T(n)= 2N-2
Comparison with an oracle, in this case a tournament theory
N-1 matches are necessary & sufficient to find a winner in a tournament of n players
Minimum has to be among those who has lost first match, so we can use losers of first comparison and try to minimize this set of numbers. The loser set will at least consist of N/2 numbers if we play pair wise, so in that case total will be N/2+(N/2-1)+(N/2-1)=3N/2-2
Instead of improving the code we went into the construction of the problem. It is to be solved by dividing the numbers into two sets when taking input of two numbers each.

Finding max & nextmax

Second largest must necessarily loose from the largest. So second largest must b found from those who were compared & lost from the largest, and the set of such number should be minimized.
The length of the path of the winner will give number of compared elements with the winner.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
15	15	9	8	15	9	4	8	7	15	1	6	9	4	3

Storing numbers from n to 2n-1
Time taken by maxmin on array of size n
let $n= 2^k$ for some k>0
T(n) = 2T(n/2) + 2
T(2) = 1
T(n) = 2{ 2T(n/4) + 2} + 2
$2^2T(n/2^2) + 2^2 + 2$
:
$= 2^{k-1} T(n/2^{k-1}) +$Sigma (i=1to k-1) 2^i
$= 2^{k-1}+$ Sigma (i=1to k-1) 2^i
$T(n)= 2^{k-1}+ (2^{(k-1)+1}-1)/2-1$
$= 2^{k-1}+ 2^K-1$
$= n/2 + n -1$
$= 3n/2 – 1$

Towers of Hanoi

At any time smaller disk should not be below the larger disk
Only disk 1 <a,b>
2 disk case <a,c>,<a,b><c,b>
3 disk case <a,c><a,b><c,a><b,c><a,c><a,b><c,a><c,b><a,b>
<a,b><a,c><b,c><a,b><c,a><c,b><a,b>
3(a,b,c) 2(a,c,b,) (1,a,b,c) (2,c,b,a)
Base: (n==1) L=<a,b>
Induction: (n>1)
l1=towers(n-1,a,c,b)
l2=towers(1,a,b,c)
l3=towers(n-1,c,b,a)

L12: Arrays and Link Lists

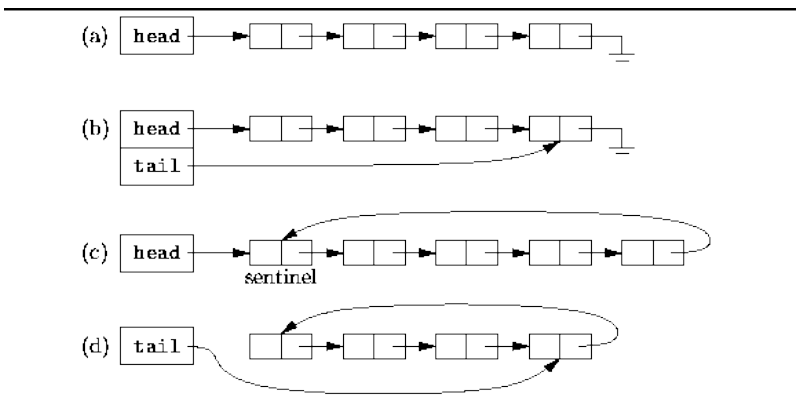
Arrays are stored in contiguous memory locations and contain similar data
An element can be accessed, inserted or removed by specifying its position (number of elements preceding it)

- Deletion: In operation removeAtPosition(r), we need to fill the hole left by the removed element by shifting backward the n - r - 1 elements V[r + 1], …, V[n - 1]. In the worst case (r = 0), this takes O(n) time
- Insertion: In operation insertAtposition(r, o), we need to make room for the new element by shifting forward the n - r elements V[r]… V[n - 1]. In the worst case (r = 0), this takes O(n) time

Accessing an array: O(1) time
Copying an array: O(n) time
Merging two arrays: O(m+n) time
Splitting an array: O(n) time

Linked List

A singly linked list is a concrete data structure consisting of a sequence of nodes
Each node stores element and link to the next node
There may be a header link and trailer link
Operations on a link list
Access (): Accessing an element from a list
Worst case: O (n)
Average Case: 1+2+3+……..n = n(n-1)/2 divided by n = n-1/2 = O(n)
Delete (): O(1) complexity, but every deletion operation will actually be preceded by accessing or reaching that position of the element, effectively taking O(n) time. Delete first will take O(1) time
Insert (): O(1) complexity, but inserting before or after an element will require accessing element effectively taking O(n) time. Inserting at first place will take O(1) time but insertion in array at first place will take O(n) .
Merging: O(1)Time complexity equal to the length of the smaller list
Size (): O(n) time
Intersection: $O(n^2)$
Union: $O(n^2)$



Array vs. Link List

- It is easier to delete in the link list
- it is easier to insert in the link list
- It is easier to access in the array
- Due to Address part there is wastage of memory in link list, it is not much if we look at the practical size of an individual record consisting of name, age, address etc
- We have to predefine the array so there can be wastage of memory

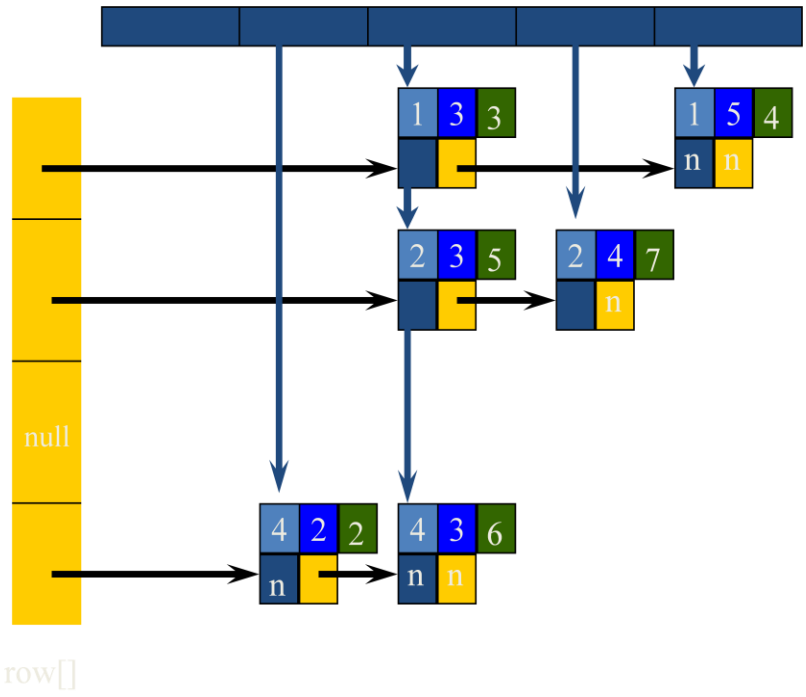
- Array Size has to be defined and is limited to the defined size while link list can grow to limit of memory
- Continuous allocation is required for array, while this is not the case with link list
- Arrays can be accessed backward and forward but we have to use special link lists like doubly linked list for backward access
- Arrays definition is part of the language construct but link list we have to create
- Merging two arrays is very difficult while merging link lists is easy
- There will be more cache misses in the Link list then in the arrays

Doubly Linked List

A doubly linked list provides a natural implementation of the List
Nodes implement Position and store:
Element link to the previous node link to the next node
Special trailer and header nodes

Circular Link List and circular doubly link list: Input output buffers use circular queue for buffering.
List: we specify the element by its position: example is the link list
Vector: we also have a notion of rank or (key, value pair)
Sequence: where both are combined and we can have the methods like rankatposition(), positionatrank()

0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0



❶❷

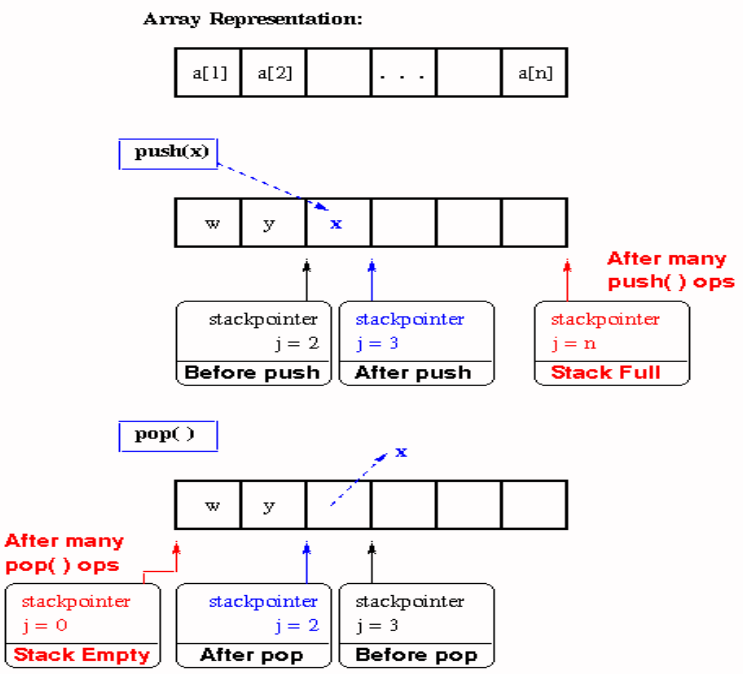
L13: Stack & Queue data structure

Stack

Last in First out, in and out only from one end, other end is closed
Direct applications
Page-visited history in a Web browser, Undo sequence in a text editor, Chain of method calls in the Java Virtual Machine, Function Calls, Recursion, in other data Structures
Creation of an stack: O(n)
push(object): inserts an element O(1)
pop(): removes and returns the last inserted element O(1)
top(): returns the last inserted element without removing it O(1)
size(): returns the number of elements stored O(1)
isEmpty(): indicates whether no elements are stored O(1)
isFull(): indicates whether array limit is over O(1)

Performance
Let ***n*** be the number of elements in the stack
The space used is ***O***(***n***) and Each operation runs in time ***O***(1)
The maximum size of the stack must be defined a priori and cannot be changed
Incremental strategy: increase the size by a constant ***c***
Doubling strategy: double the size
If we increment by C every time then the creation, copying, insertion cost will be as follows
Creation copying insertion (iteration wise)
c*i c(i-1) c
Total = 2ci
if there are n elements in the end then the total number of iteration will be n/c
So it comes out to be $2c(1+2+3+\ldots\ldots.n/c) = 2c(n/c)(n/c+1)/2 = n^2/c = O(n^2)$
If we double the array every time then the creation, copying, insertion cost will be as follows
Creation copying insertion (iteration wise)
2^i 2^{i-1} 2^{i-1}
Total cost in every iteration is 2^{i+1}
if there are n elements to be inserted then there will be lgn iterations so the total becomes
$(2^1+2^3+2^3+\ldots\ldots\ldots\ldots+2^{lgn+1}) = 4n-1 = O(n)$

axioms
pop(push(S,v))=S Top(push(S,v))=v

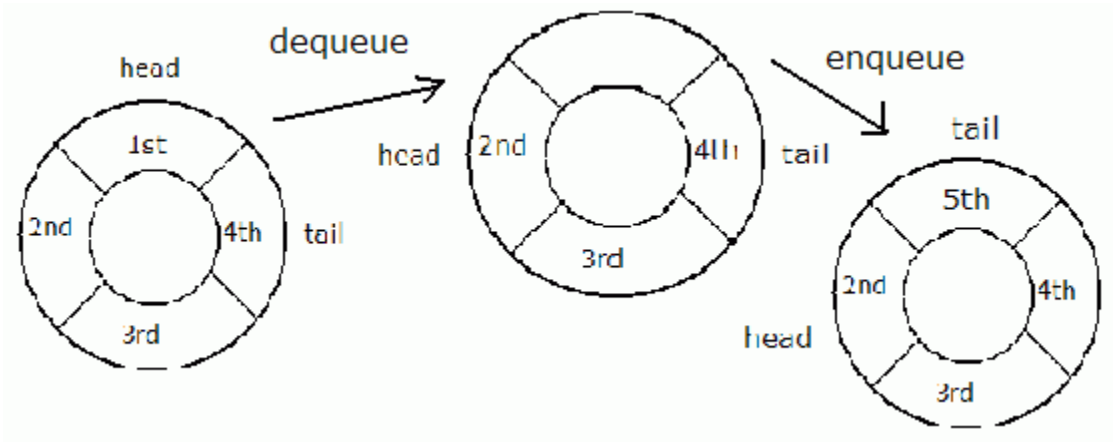


Queue

Insertions and deletions follow the first-in first-out scheme
Insertions are at the rear of the queue and removals are at the front of the queue
Two variables keep track of the front and rear
operations:

enqueue(Object o): inserts an element o at the end of the queue time is O(1)
dequeue(): removes and returns the element at the front of the queue O(1)
front(): returns the element at the front without removing it O(1)
size(): returns the number of elements stored O(1)
isEmpty(): returns a Boolean indicating whether no elements are stored O(1)
Direct applications
Waiting lists, bureaucracy
Access to shared resources (e.g., printer)
Multiprogramming
Tunnel
In other Data Structures
Axioms
Front(Enqueue(Enqueue(Q,w),v))=front(Enqueue(Q,w))
Dequeue(Enque(Enque(Q,w),v))=Enqueue(Dequeue(Enqueue(Q,w)),v)
In an enqueue operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one, Similar to what we did for an array-based stack

Discuss dequeue with its operational difference from normal queue
it should be implemented with two pointers F and R so that on insertion R<- R+1 if R=F then overflow if R=M then R<-1 and on deletion F<-F+1 if F=R then underflow if F=M then F<-1



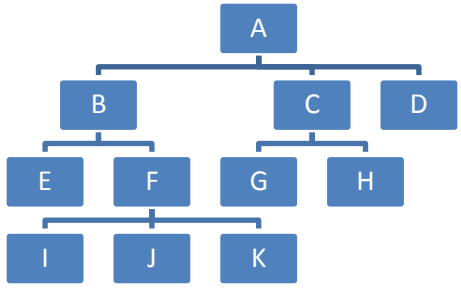
❶❸

L14: Trees & Binary Search Trees

In computer science, a tree is an abstract model of a hierarchical structure
A tree consists of nodes with a parent-child relation
Applications: Organization charts, File systems, Programming environments

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (or leaf node): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Descendant of a node: child, grandchild etc.
- Subtree: tree consisting of a node and its descendants
- Siblings: Children of the same parent

size()
isEmpty()
root()
parent(p)
children(p)
isInternal(p)
isExternal(p)
isRoot(p)
swapElements(p, q)
replaceElement(p, o)

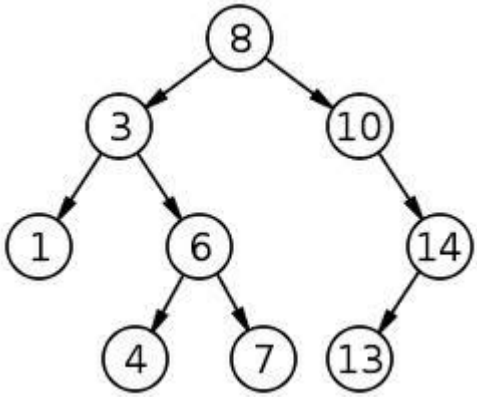


Binary Tree

is a tree in which each internal node has at most two children
Ordered Binary Tree
The children of a node are an ordered pair and are referred as left child and right child
Alternative recursive definition: a binary tree is either a tree consisting of a single node, or whose root has an ordered pair of children, each of which is a binary tree
Complete Binary tree which consists of each internal node having exactly two children.
Arithmetic expressions, decision processes, searching

n	number of nodes
e	number of external nodes
i	number of internal nodes
h	height

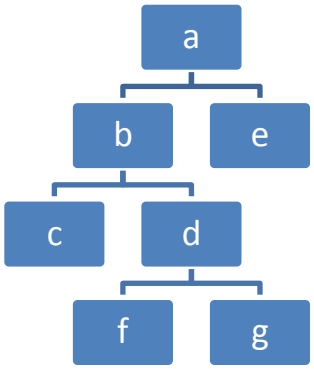
Properties:
$e = i + 1$
$n = 2e - 1$
$h \leq i$
$h \leq (n - 1)/2$
$e \leq 2^h$
$h \geq \log_2 e$
$h \geq \log_2 (n + 1) - 1$



14

L15: Tree traversals and operations

A traversal visits the nodes of a tree in a systematic manner
In a preorder traversal, a node is visited before its descendants
Application: print a structured document
In a postorder traversal, a node is visited after its descendants
Application: compute space used by files in a directory and its subdirectories
Specialization of an inorder traversal
Preorder abcdfge Postorder cfgbdea Inorder cbfdgae



Evaluating arithmetic operation is equivalent to a postorder traversal

Given pre and inorder we can find postorder

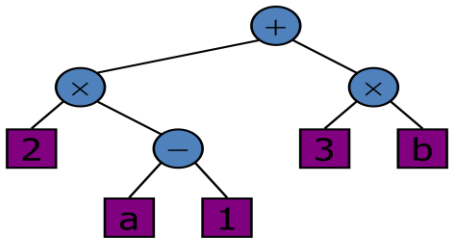
Preorder abcdfge Inorder cbfdgae
From preorder we know that a is the root, now find a in the inorder traversal
Now I know e is the right subtree
We are left with the subtree
bcdfg cbfdg
dfg fdg

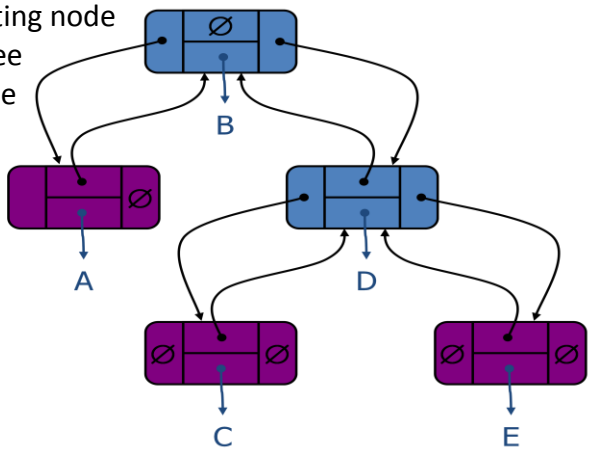
Given Pre & postorder

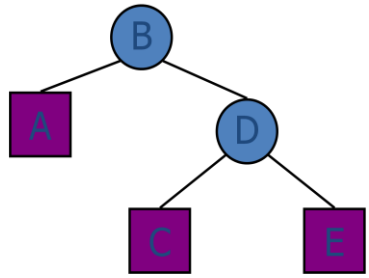
We cannot find inorder because there can be two trees with the same pre and post order
pre a b c
post c b c
If each internal node of the binary tree has at least two children then the tree can be determined from pre and post order traversals.
pre post
a b c d f g e c f g d b e a from this I know that a is the root then I know that e is the right child of a from post order and from Preorder I see that there is nothing after e so e is the leaf
b c d f g c f g d b
Now I see that b is the root of the left sub tree and d is the right children (from post order) and then from inorder I see that c is the only child of d which is the left child and so on.

Given post and inorder

c f g d b e a c b f d g a e
c f g d b c b f d g
f g d f d g
Specialization of an inorder traversal
print operand or operator when visiting node
print "(" before traversing left subtree
print ")" after traversing right subtree
((2 × (a - 1)) + (3 × b))





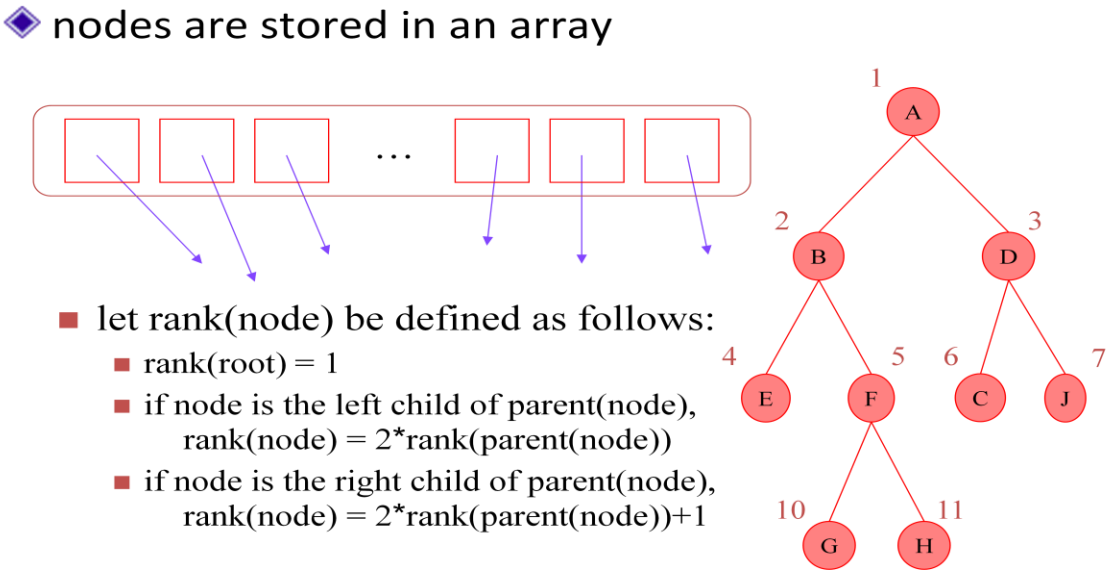


Linked Data structure for binary trees

A node is represented
by an object storing
Element

Parent node
Left child node
Right child node

Array based representation



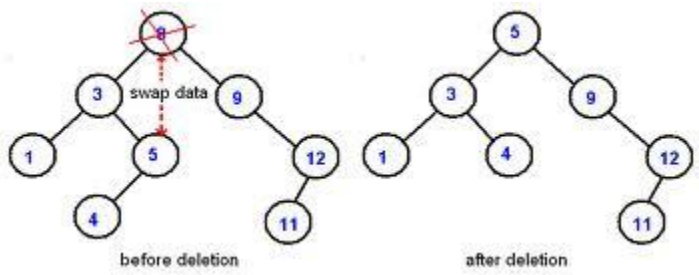
To find the smallest element in a tree we start form the top and keep going left unless it is null . For maximum keep going right unless it becomes null.

Successor

Given x, find the node with the smallest key greater then key[x]
we can distinguish two cases depending upon the right subtree of x
case 1: Right subtree of x is nonempty, then successor is leftmost node in the right subtree
case 2: The right subtree of x is empty, then the successor is the closest ancestor v of x such that x is descended from the left child of v.If there is no such ancestor then successor is undefined.

Insertion :

take an element whose left and right children are null and insert it into T
find place in T where z belongs (as if searching for z)
and add z
Runtime on a tree of height h is O(h)



Deletion

Case 1 :
1. if x has no children : just remove x
2. if x has exactly one child then to delete x simply make p[x] point to that child
3. if x has two children, then to delete it we have to find its predecessor(going left and finding the rightmost node) or successor y
Replace x with y (it will have at most one child) and delete y
Running time for delete in worst case is O(h)
Inorder traversal of a BST gives a sorted list and taken O(n) time.

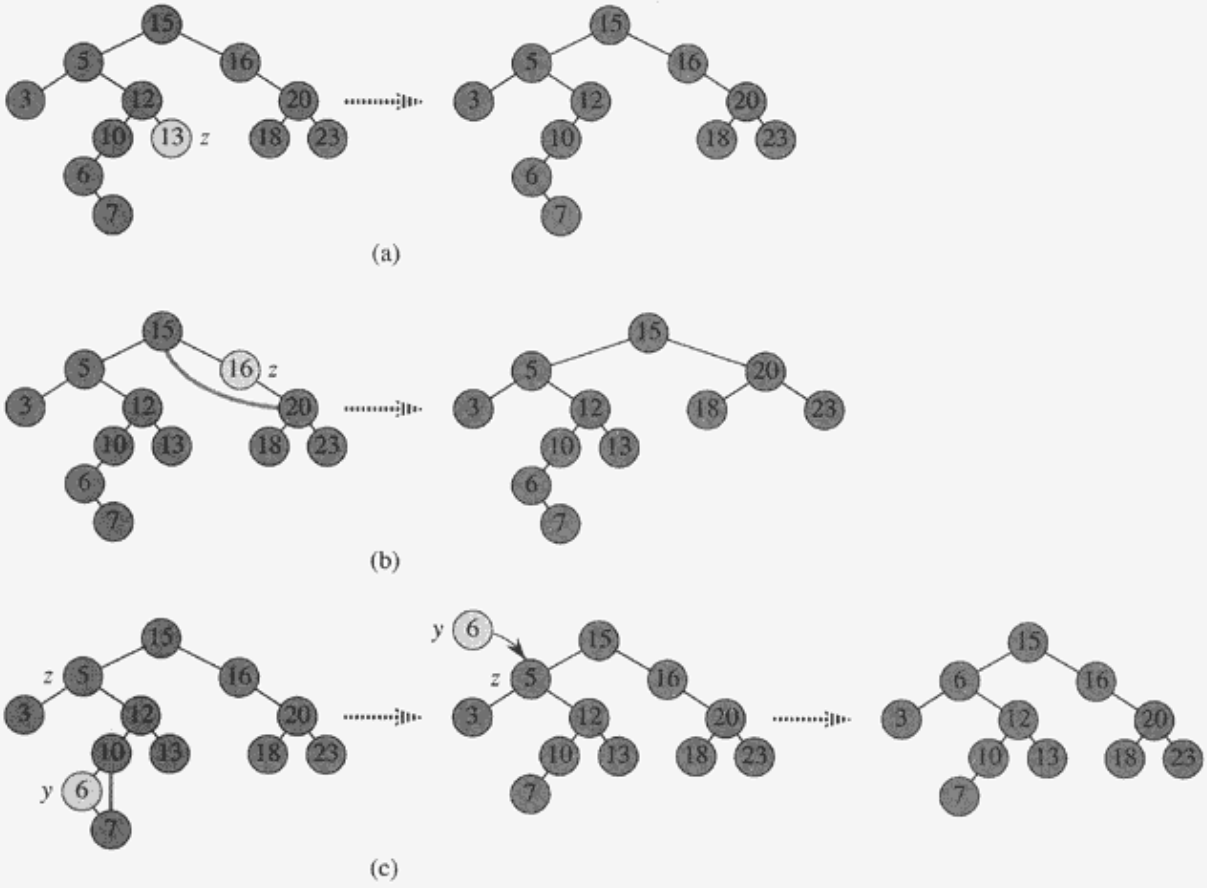


Figure 12.4 Deleting a node z from a binary search tree. Which node is actually removed depends on how many children z has; this node is shown lightly shaded. **(a)** If z has no children, we just remove it. **(b)** If z has only one child, we splice out z. **(c)** If z has two children, we splice out its successor y, which has at most one child, and then replace z's key and satellite data with y's key and satellite data.

15

L16: Solving Recurrences

Substitution Method

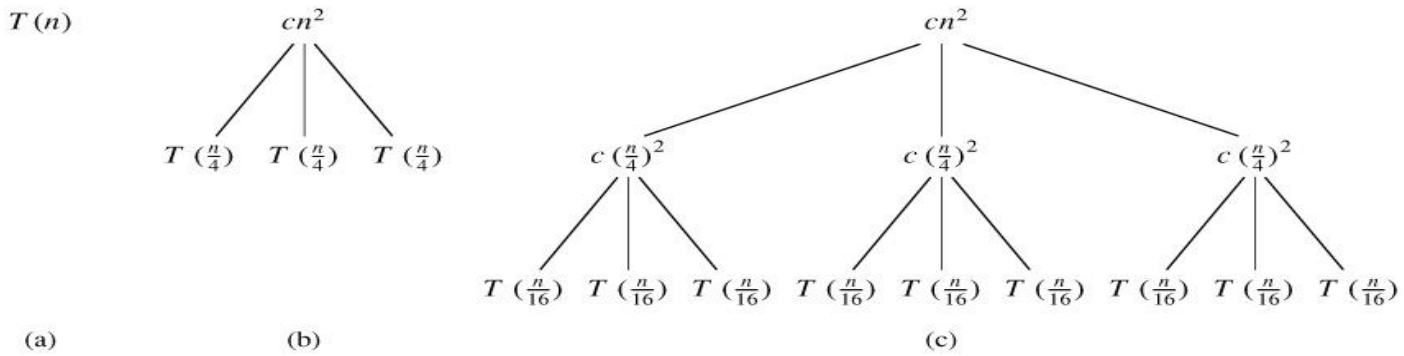
1. Guess the form of the solution
2. Verify by induction and show that the solution works for a set of constants

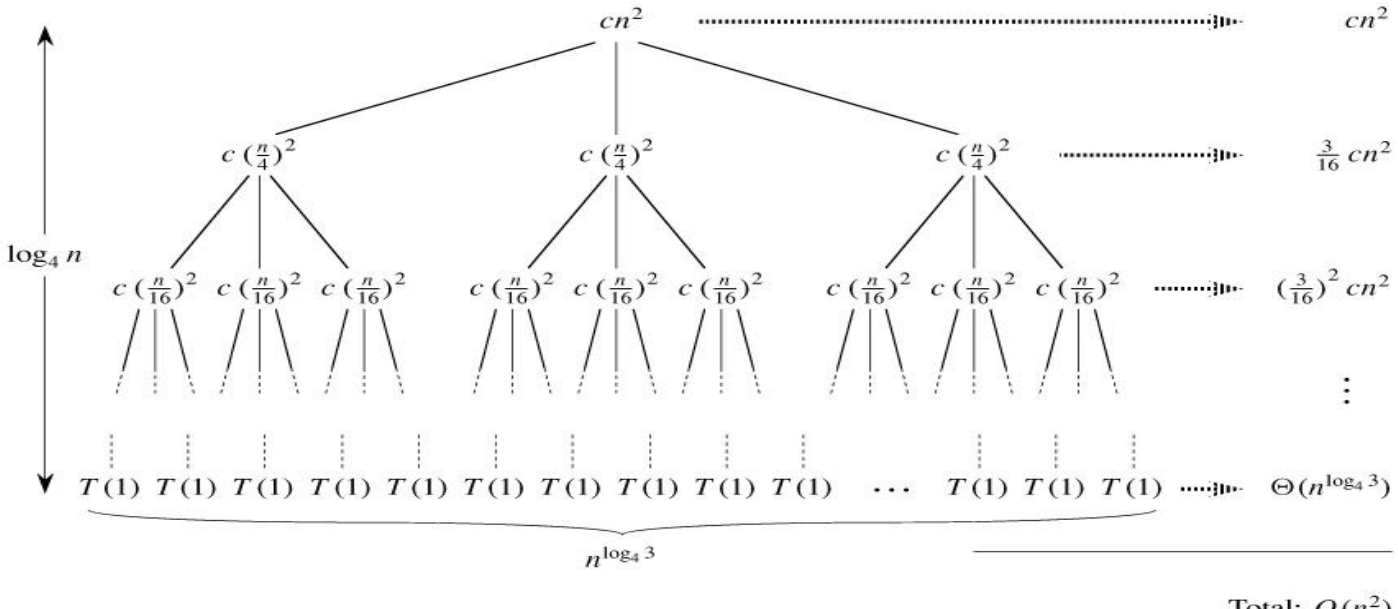
T(n)=2T(n/2) + n Which means that now the problem has been divided into two sub problems and the size of the sub problem is n/2

We guess that it works for T(n) = O(n lg n)
To prove that T(n) ≤ cn lg n for an appropriate c.

Assuming that the guess works we will have
T(n) = 2(c (n/2) lg (n/2))+n
= cn lg(n/2) +n
= cn lg n – cn lg 2 + n
= cn lg n – n(c-1)
≤ cnlgn
this is true for all c ≥ 1

Recursion Tree Method



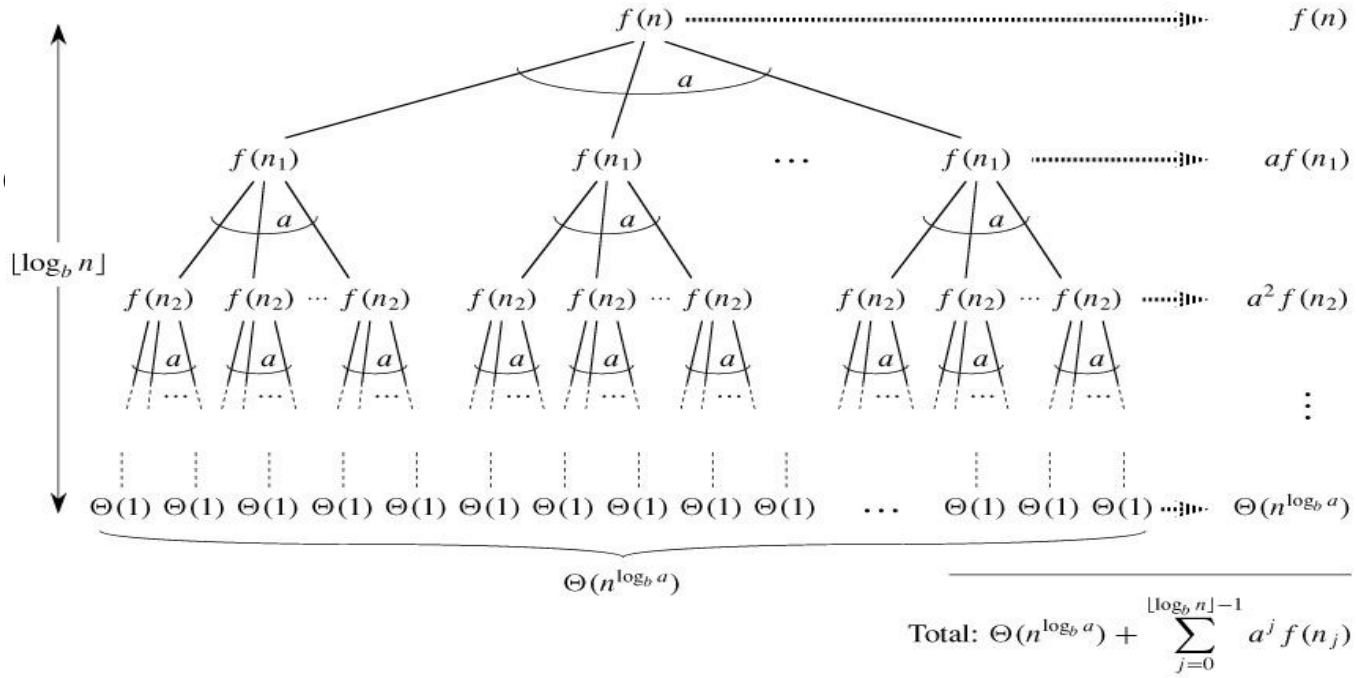


Total = $n^2 + (3/16)n^2 + (3/16)^2 n^2 + \ldots\ldots\ldots + 3^k/16^k + \ldots$

This is a geometric series
$\leq n^2 (1/1-(3/16))$
$\leq 2n^2$
$\leq \theta(n^2)$

Master Method

T(n)= aT(n/b) + f(n) : Where a ≥ 1 and b>1 are constants and f(n) is an asymptotically positive function
We are dividing a problem of size n into a sub problems, each of size n/b, where a and b are positive constants. a sub problems are solved recursively each in time T(n/b) where a and b are positive constants. The cost of dividing the problem and combining the results of the sub problems is described by the function f(n).
1. If $F(n) = O(n^{\log_b a - \varepsilon})$for some constant ε > 0, then $T(n) = \theta (n^{\log_b a})$
2. If , $f(n) = \theta (n^{\log_b a} \log^k n)$then . $T(n)= \theta(n^{\log_b a} \log^{k+1} n)$
3. If $f(n)=\Omega(n^{\log_b a+ \varepsilon})$ for some constant ε > 0, and if $a f (n/b) \leq cf (n)$ for some constant $c <$
1 and all sufficiently large n, then $T (n) = \Theta(f (n))$.
Three cases do not cover all the possibilities for $f (n)$. There is a gap between cases 1 and 2 when $f (n)$ is smaller than but not polynomially smaller. Similarly, there is a gap between cases 2 and 3 when $f (n)$ is larger than but not polynomially larger. If the function $f (n)$ falls into one of these gaps, or if the regularity condition in case 3 fails to hold, the master method cannot be used
T(n) = 4T(n/2) + n
$n^{\log_b a} = n^2$ which is polynomially larger then f(n) so we are in case 1 and answer is $\theta(n^2)$
$T(n) = 4T(n/2) + n^2$
$n^{\log_b a} = n^2$ which is same similar to f(n) so we are in case 2 and answer is $\theta(n^2 \log n)$
$T(n) = 4T(n/2) + n^3$
$n^{\log_b a} = n^2$ which is polynomially larger from f(n) so we are in case 3 and answer is $\theta(n^3)$
$T(n)=2T(n/2)+n\log n$ case 2 $O(n\log^2 n)$
$T(n) = 4T(n/2) + n^2 / \log n$ Master method does not apply
Binary Search $T(n) = 1.T(n/2) + O(1)= \theta(\log n)$
Finding Powers of a number $T(n)= T(n/2)+ \theta(1) = \theta(\log n)$



Case1: The Weight increases Geometrically from the root to the leaves. The leaves hold a constant fraction of the total weight. Last term is dominating. Case 2: for k=0 weight is approximately same on all levels. It is polynomial or slowly decreasing function. Case 3: The Weight decreases Geometrically from the root to the leaves. The root hold a constant fraction of the total weight. First term is dominating.

L17: Hashing

Dictionary data structure
Symbol table in the compiler having a record with key and satellite information
A hash function h maps keys of a given type to integers in a fixed interval [0,N-1]. The goal of a hash function is to uniformly disperse keys in the range [0,N-1]
Develop a structure that will allow user to insert/delete/find records in
Constant average time
Structure will be a table (relatively small) and Table completely contained in memory and implemented by an array. It capitalizes on ability to access any element of the array in constant time
It determines position of key in the array.
Multiplication method for creating hash functions:
Multiply key k by a constant A in the range 0<A<1 and extract the fractional part of kA. Then, we multiply this value by m and take the floor of the result.
h(k) = floor(m(kA mod 1)
Where kA mod 1 means the fractional part of kA
Value of m is not critical and we typically choose it to be a power of 2 ($m=2^p$ for some integer p.
Assuming computer has w bit word and k fits into a single word then A can be a fraction of the form $s/2^w$ where s is an integer in the range $0<s<2^w$. We first multiply k by w-bit integer $s=A.2^w$ The result is a 2w bit value $r_1 2^w + r_0$ where r1 is the high order word of the product and r_0 is the low order word of the product. Desired p-bit hash value consists of the p most significant bits of r_0

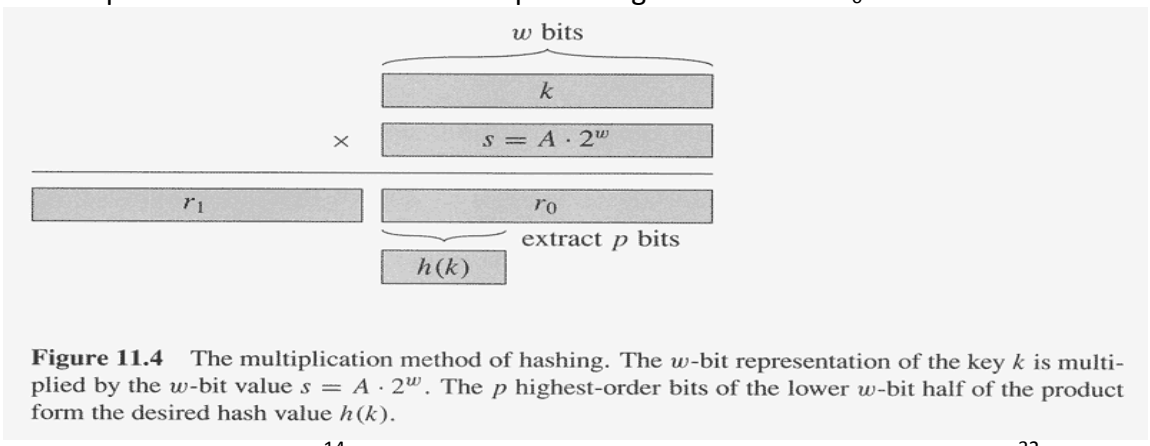


Figure 11.4 The multiplication method of hashing. The w-bit representation of the key k is multiplied by the w-bit value $s = A \cdot 2^w$. The p highest-order bits of the lower w-bit half of the product form the desired hash value $h(k)$.

k = 123456, p =14, m = 2^{14} = 16384, w = 32, s=2654435769 A = 2654435769/2^{32},
So, k*s = 327706022297664 = 76300*2^{32} + 17612864, which means r1 = 76300
and r0 = 17612864= 00000001000011001100000001000000.
The 14 most significant bits of r0 give h(k) =00000001000011 = 67.

Assume table (array) size is *N*
Function *f(x)* maps any key *x* to an int between 0 and *N*−1
For example, assume that *N*=15, that key *x* is a non-negative integer between 0 and MAX_INT, and hash function *f(x) = x* % 15 this is called division type hash function don't pick m with small divisor E.G. if d=2 and all keys are even odd slots are never used. IF $M=2^r$ then it does not depend on all bits. Good thing is take a prime not close to a power of 2 or 10
Let *f(x) = x* % 15. Then,

if *x* =	25	129	35	2501	47	36
f(x)=	10	9	5	11	2	6

Storing the keys in the array is straightforward:

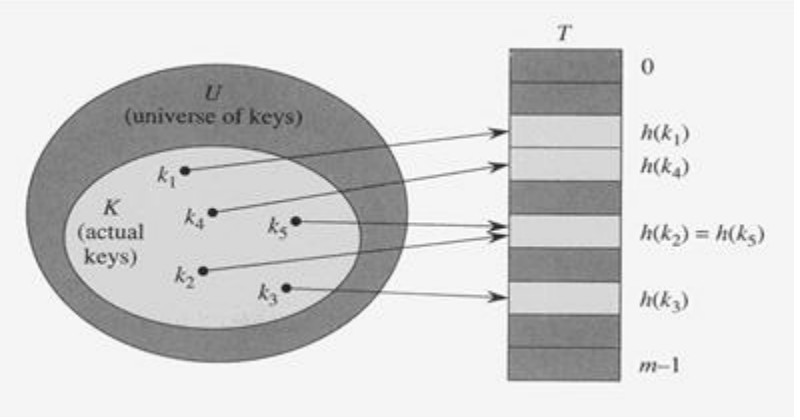
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
_	_	47	_	_	35	36	_	_	129	25	2501	_	_	

Thus, *delete* and *find* can be done in O(1), and also *insert*, except…
What happens when you try to insert: *x* = 65 ?

x = 65
f(x) = 5

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
_ _47 _ _ 35 36 _ _ 129 25 2501 _ _ 65(?)

when an insert maps to an already occupied slot it is called collision
Separate Chaining
Open Addressing (deletion is diffuclt)
Linear Probing
Quadratic Probing
Double Hashing (excellent method)



Let each array element be the head of a chain:
Where would you store: 29, 16, 14, 99, 127 ?

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
16 47 65 36 127 99 25 2501 14
↓ ↓ ↓
35 129 29

New keys go at the front of the relevant chain.
Parts of the array might never be used.
As chains get longer, search time increases to O(n) in the worst case.
Constructing new chain nodes is relatively expensive (still constant time, but the constant is high).
Is there a way to use the "unused" space in the array instead of using chains to make more space?
Average case: Any key is equally likely to be hashed independent of any other key filling the same slot.
Load Factor of a hash table with n keys and m slots alpha=n/m= average number of keys per slot
Expected unsuccessful search time = 1+ alpha=1+n/m
Expected search time = O(1)

Let key *x* be stored in element *f(x)=t* of the array

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
47 35 36 129 25 2501
65(?)

What do you do in case of a collision?
If the hash table is *not full*, attempt to store key in array elements (*t*+1)%*N,* (*t*+2)%*N*, (*t*+3)%*N …*
until you find an empty slot
If the hash table is *not full*, attempt to store key in array elements (*t*+1)%*N,* (*t*+2)%*N*, …
Where would you store: 65,29,16,14,99,127 ?

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
16 47 35 36 65 127 129 25 2501 29 99 14

Eliminates need for separate data structures (chains), and the cost of constructing nodes.
Leads to problem of clustering. Elements tend to cluster in dense intervals in the array.
Search efficiency problem remains.
Deletion becomes trickier….

Let key *x* be stored in element *f(x)=t* of the array

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
47 35 36 129 25 2501
65(?)

What do you do in case of a collision?
If the hash table is *not full*, attempt to store key in array elements (*t*+1^2)%*N,* (*t*+2^2)%*N*, (*t*+3^2)%*N …*
until you find an empty slot.
If the hash table is *not full*, attempt to store key in array elements (*t*+1^2)%*N,* (*t*+2^2)%*N …*
Where would you store: 65,29,16,14,99,127 ?

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
29 16 47 14 35 36 127 129 25 2501 99 65

↑
t
attempts

Tends to distribute keys better than linear probing
Alleviates problem of clustering
Runs the risk of an infinite loop on insertion, unless precautions are taken.
E.g., consider inserting the key 16 into a table of size 16, with positions 0, 1, 4 and 9 already occupied.
Therefore, table size should be prime
Use a hash function for the decrement value
Hash(key, i) = H_1(key) – (H_2(key) * i)
Now the decrement is a function of the key
The slots visited by the hash function will vary even if the initial slot was the same
Avoids clustering
Theoretically interesting, but in practice slower than quadratic probing, because of the need to evaluate a second hash function.
Hashing offers excellent performance for insertion and retrieval of data but Weakness of hashing: For any choice of hash functions, there exists a bad set of keys that all hash to the same slot. So it is easy for the adversary to make working of hash functions slow.
We can choose hash functions at random which is independent of the number.

Universal Hashing

Universal hashing consists of choosing a hash function randomly from some fixed set of hash functions independent of the keys. Therefore, universal hashing is (probabilistically) immune to bad distributions.

Let H be a finite collection of hash functions that each map a given universe U of keys into the range {0, 1, …, m-1}. H is said to be a universal collection if for each pair of distinct keys k, l $\in$ U, the number of hash fns. for which h(k) = h(l) is at most |H|/m, i.e., the chance of collision between k and l is no more than the chance 1/m of collision if h(k) and h(l) were independently and randomly chosen (as in simple uniform hashing).

A Universal Class of Hash Functions
Let prime p be large enough so that every possible key k is in the range $0 \le k \le p-1$.
Let Zp denote {0, 1, …, p-1} and Zp* denote {1, 2, …, p-1}.
Because the hash table size is smaller than that of the universe of keys, we have also m < p.
For any a $\in$ Zp* and b $\in$ Zp define the hash function ha,b :

$$h_{ab}(k) = ((ak+b) \bmod p) \bmod m.$$

E.g., if p = 17 and m = 6, we have $h_{3,4}(8) = 5$.
Ques: If p = 29 and m = 20, what is $h_{5,9}(17)$?
We shall show that the family of such hash functions $h_{a,b}$, i.e., the family

$$H_{p,m} = \{h_{a,b} : a \in Zp^* \text{ and } b \in Zp\}$$

is universal.

Theorem: The class $H_{p,m}$ of hash functions is universal.
Proof: Consider two distinct keys k and l from Z_p. For a given hash fn. $h_{a,b}$ let

$$r = h_{a,b}(k) = (ak + b) \bmod p$$
$$s = h_{a,b}(l) = (al + b) \bmod p$$

Now $r \neq s$. Why? Because, if r = s, then

$$ak + b = al + b \bmod p \Rightarrow ak = al \bmod p \Rightarrow k = l \bmod p,$$

contradicting that $k \neq l$. Therefore, k and l map to distinct values r and s mod p.
Moreover, each of the p(p-1) choices of the pair (a, b), with $a \neq 0$, yields a different resulting pair (r, s) with $r \neq s$. Why? Because, for a given r and s, we can solve the equations

$$ak + b = r \quad \text{and} \quad al + b = s$$

Uniquely to determine a and b (check this!). Therefore, different pairs (r,s) must give different pairs (a, b). Since, there are p(p-1) choices of pairs (r, s) with $r \neq s$, there is a one-to-one correspondence between pairs (a, b) with $a \neq 0$ and pairs (r, s) with $r \neq s$.

Therefore, if (a, b) is picked randomly from $Z_p^* \times Z_p$ the resulting pair (r, s) is equally likely to be any pair of distinct values *mod* p.

Now, given r, of the p-1 remaining possible values for s, the number of values such that $r \equiv s \pmod m$ is at most $\lceil p/m \rceil - 1 \leq (p-1)/m$

Therefore, the number of hash function $h_{a,b}$ in $H_{p,m}$ such that $h_{a,b}(k) = h_{a,b}(l)$ (which is exactly when $r \equiv s \pmod m$) is at most $p(p-1)/m = |H_{p,m}|/m$, proving that $H_{p,m}$ is indeed universal

Perfect Hashing

If the set of keys is static (e.g., a set of reserved words in a programming language), hashing can be used to obtain excellent worst-case performance.

A hashing technique is called perfect hashing if the worst-case time for a search is O(1).

A two-level scheme is used to implement perfect hashing with universal hashing used at each level.

The first level is same as for hashing with chaining: n keys are hashed into m = n slots using a hash fn. h from a universal collection.

At the next level though, instead of chaining keys that hash to the same slot j, we use a small secondary hash table Sj with an associated hash fn. hj. By choosing hj appropriately one can guarantee that there are no collisions at the secondary level and that the total space used for all the hash tables is O(n).

The first-level hash fn. h is chosen from the class Hp,m.

Those keys hashing into slot j are re-hashed into a secondary hash table Sj of size m_j , where $m_j = nj2$, the square of the number n_j of keys hashing into slot j, using a hash fn. h_j chosen from the class Hp,mj.

If we store n_j keys in a hash table of size $m_j = nj2$ using a hash function hj randomly chosen from H_p,m_j, then the probability of there being any collision is less than ½.

Therefore, repeatedly randomly choosing a hash function from H_p,m_j will soon yield an h_j that is collision-free.

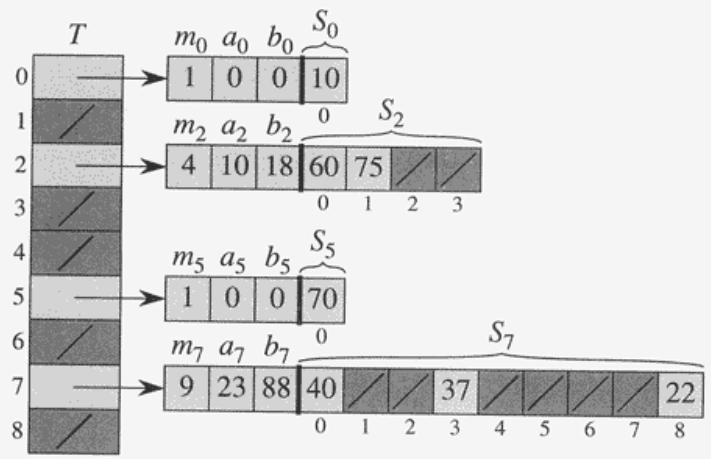


Figure 11.6 Using perfect hashing to store the set $K = \{10, 22, 37, 40, 60, 70, 75\}$. The outer hash function is $h(k) = ((ak + b) \bmod p) \bmod m$, where $a = 3$, $b = 42$, $p = 101$, and $m = 9$. For example, $h(75) = 2$, so key 75 hashes to slot 2 of table T. A secondary hash table S_j stores all keys hashing to slot j. The size of hash table S_j is m_j, and the associated hash function is $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$. Since $h_2(75) = 1$, key 75 is stored in slot 1 of secondary hash table S_2. There are no collisions in any of the secondary hash tables, and so searching takes constant time in the worst case.

L18: Sorting: Bubble, Selection, Insertion

Given N nos. , we have to rearrange them so that
$a1 \le a2 \le a3 \le a4 \le a5 \le a6 \le a7 \le a8 \ldots\ldots \le a_{n-1} \le a_n$
There are n! different permutations possible for this sequence, out of this only 1 is the right sequence

Why Sorting

Sorting solves togetherness problem
Matching items in two or more files
Searching by key values

Types of sorting

Sorting by comparison and by other methods like distribution
Sorting by Exchange-Interchanging pairs of elements that are out of order until no such pairs exist
Internal and external sorting
Stable sorting
In place sorting
Adaptive sorting which take into account the sortedness of the sequence
Whether it can be parallelized
Whether it is cache-oblivious

Inversions

How many numbers are in reverse order
Inversions of a permutation: if ai > aj and i<j the pair (ai,aj) is called the inversion of the permutation.
e.g. 3 1 4 2 has three inversion namely (3,1) (3,2) and (4,2). Each inversion is a pair of elements that are out of sort.

Inverse of a permutation

Is the permutation obtained by interchanging the two rows and then sorting the columns into increasing order of the new top row

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

Runs

In a given list represent sorted segments of the data in the said order
The sequence 3 5 7 | 1 6 8 9 | 4 | 2 has four runs

Tableaux

An array of integers with left justified rows is called tableaux such that entries in each row are in increasing order from left to right and entries in each column are increasing from top to bottom.

1	2	5	9	10	15
3	6	7	13		
4	8	12	14		
11					

Involution

An involution of a permutation is that is its own inverse. There are ten involution of 1,2,3,4

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

Counter example

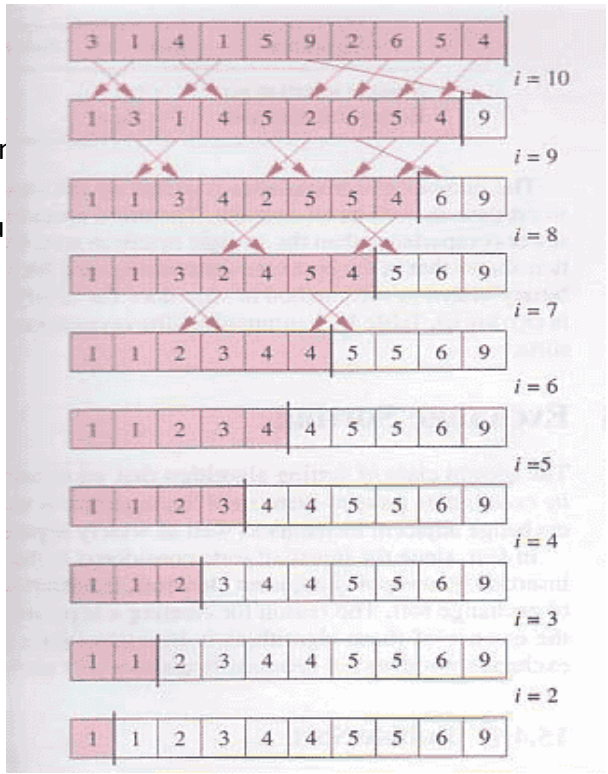
1 2 3 4
2 3 4 1

Also, ten tableaux can be formed from 1,2,3,4

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

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

Bubble sort



Bubble sort works by comparing adjacent elements of an array and exchanges them if they are not in order. After each iteration (pass) the largest element bubbles up to the last position of the array. In the next iteration the second largest element bubbles u to the second last position and so on

1. Begin
2. For i = 1 to n-1 do
2.1 For j = 1 to n-i do
2.2.1 If (a[j+1] < a[j]) then swap a[j] and a[j+1]
3. End

Improvements

Traversing in opposite direction in alternate passes
if two adjacent elements don't exchange for two Consecutive passes then we can fix their position
Sorting after implementing these improvements is also Called cocktail sort.
Problem areas: The elements are moving one at a time keeps on sorting even the sorted list
It is stable, in place and popular due to tiny code.
Worst case and average case complexity is $O(n^2)$

Selection Sort

To sort the given array a[1…n] in ascending order:

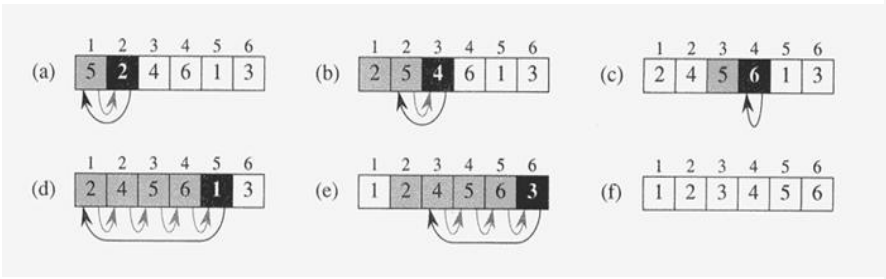
1. Begin
2. For i = 1 to n-1 do
2.1 set min = i
2.2 For j = i+1 to n do
2.2.1 If (a[j] < a[min]) then set min = j
2.3 If (i < min) then swap a[i] and a[min]
3. End

Complexity is $O(n^2)$ for average and worst case. It is also

stable and inplace sorting algorithm.
Min max can be applied
The number of swap operations is very less. It outperforms bubble sort.

Insertion Sort

Before examining record R_j, we assume that the preceding records R1 to Rj-1 have already been sorted and we insert Rj into its proper place among the previously sorted records




```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2      do key ← A[j]
3         ▷ Insert A[j] into the sorted sequence A[1 . . j − 1].
4         i ← j − 1
5         while i > 0 and A[i] > key
6             do A[i + 1] ← A[i]
7                i ← i − 1
8         A[i + 1] ← key
```

Two Way insertion

Binary Search with in sorted sequence

Complexity is $O(n^2)$ in worst and average case. It is also a stable and in place sorting. Inserting in between is very costly because we have to remove the remaining elements. It outperforms bubble and selection sort.

18

L19: Sorting: Shell, Merge, Count, Radix

Shell Sort

If we have a sorting algorithm which moves items only one position at a time, its average running time will be at best proportional to n^2. Since each record must travel an average of about n/3 positions during the sorting process. So if we want to make substantial improvements over straight insertion, we need some mechanism by which the records can take long leaps instead of short steps

One implementation of shell sort can be described as arranging the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort.

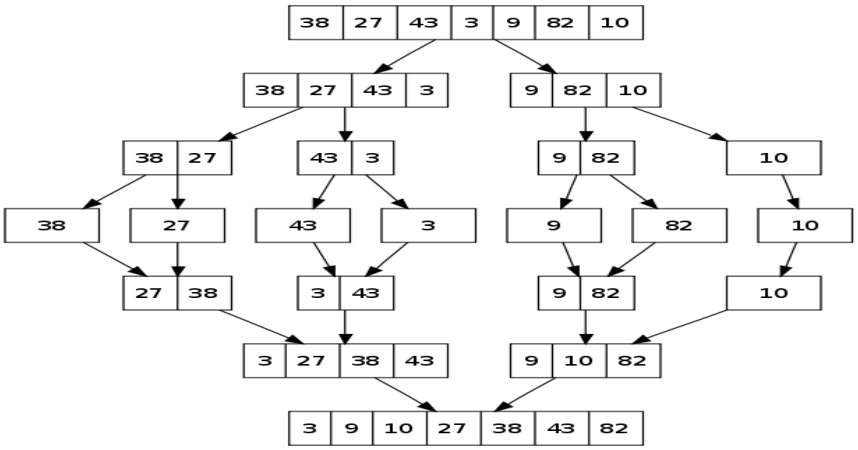
This does not allows the interaction between even and odd keys, so we can try 7-sort,5-sort,3-sort and 1-sort

Shell Sort performs better then Bubble, Selection or insertion sort. It is not a stable sort. It is in place. The Worst case complexity is $O(n\log^2 N)$ or $O(N^{3/2})$ or others depending upon the Gap sequence.

Original	32 95 16 82 24 66 35 19 75 54 40 43 93 68	
After 5-sort	32 35 16 68 24 40 43 19 75 54 66 95 93 82	6 swaps
After 3-sort	32 19 16 43 24 40 54 35 75 68 66 95 93 82	5 swaps
After 1-sort	16 19 24 32 35 40 43 54 66 68 75 82 93 95	15 swaps

Merge Sort

Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e., 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list. Its worst and average case running time is O(n log n). It is better than the previously discussed algorithms. It can be implemented in place or outofplace. It is a divide and conquer algorithm. it can be easily parallelized.



```
MERGE(A, p, q, r)
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
 4  for i ← 1 to n1
 5       do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7       do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13       do if L[i] ≤ R[j]
14            then A[k] ← L[i]
15                 i ← i + 1
16            else A[k] ← R[j]
17                 j ← j + 1
```

```
MERGE-SORT(A, p, r)
1  if p < r
2     then q ← ⌊(p + r)/2⌋
3          MERGE-SORT(A, p, q)
4          MERGE-SORT(A, q + 1, r)
5          MERGE(A, p, q, r)
```

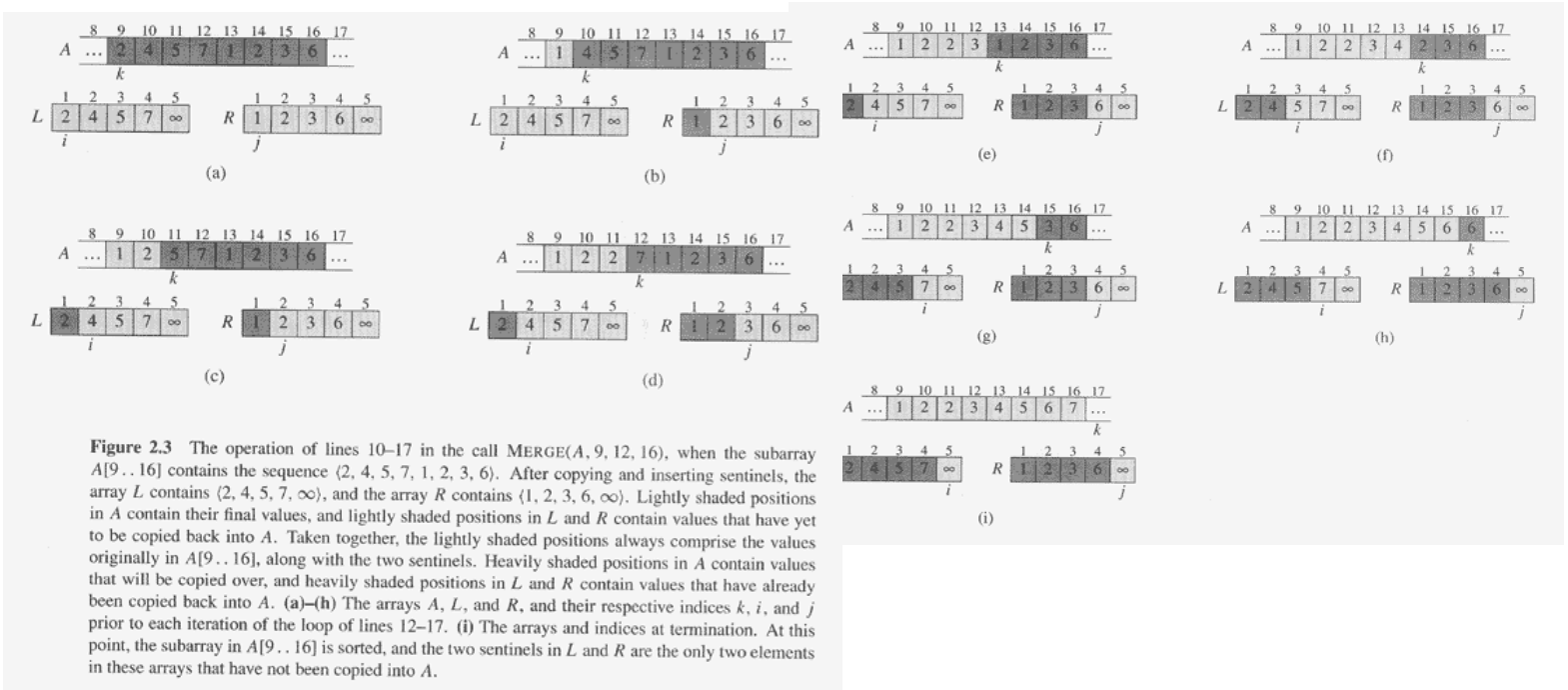


Figure 2.3 The operation of lines 10–17 in the call MERGE(A, 9, 12, 16), when the subarray A[9 . . 16] contains the sequence ⟨2, 4, 5, 7, 1, 2, 3, 6⟩. After copying and inserting sentinels, the array L contains ⟨2, 4, 5, 7, ∞⟩, and the array R contains ⟨1, 2, 3, 6, ∞⟩. Lightly shaded positions in A contain their final values, and lightly shaded positions in L and R contain values that have yet to be copied back into A. Taken together, the lightly shaded positions always comprise the values originally in A[9 . . 16], along with the two sentinels. Heavily shaded positions in A contain values that will be copied over, and heavily shaded positions in L and R contain values that have already been copied back into A. **(a)–(h)** The arrays A, L, and R, and their respective indices k, i, and j prior to each iteration of the loop of lines 12–17. **(i)** The arrays and indices at termination. At this point, the subarray in A[9 . . 16] is sorted, and the two sentinels in L and R are the only two elements in these arrays that have not been copied into A.

Counting Sort

for i←1 to k
do C[i] ←0
for j← 1 to n
C[A[j]] ←C[A[j]]+1
for i←2 to k
do
C[i] ←C[i] + C[i-1]
for j← n down to 1
do
B[C[A[j]]] ←A[j]
C[A[j]] ← C[A[j]]-1

A
numbers to be sorted

1 2 3 4 5
4 1 3 4 3

C
1 2 3 4
0 0 0 0
Frequency of occurrence of each number
1 0 2 2
Sum of values preceding a index
1 1 3 5

B
1 3 3 4 4

Complexity is O(k+n) if k=O(n) where k is the range of elements
It is stable and good for small range of numbers

Radix Sort
Sort by Least significant digit first using stable sort

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	455	720
355	839	657	839

For each round we use counting sort of O(k+n)
Suppose we have n integers each b bits ranging from 0 to 2^b-1
split into b/r digits each r bits long
Comlexity will be O(b/r(n+k) = O(b/r(n+2^r)
(b/r)*n wants r to be big for the overall term to be small
(b/r)*2^r wants r to be small
We wish n term to dominate 2^r
Choose r maximum subject to $n \geq 2^r$
r=lgn
so we get (bn/lgn)
if numbers are in the range 0…..2^b
then for numbers in the range 0…..n^d-1
then b=dlgn
and the complexity = O(dn) if d=O(1) then complexity=O(n)

Bucket Sort

```
BUCKET-SORT(A)
1  n ← length[A]
2  for i ← 1 to n
3       do insert A[i] into list B[⌊nA[i]⌋]
4  for i ← 0 to n − 1
5       do sort list B[i] with insertion sort
6  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

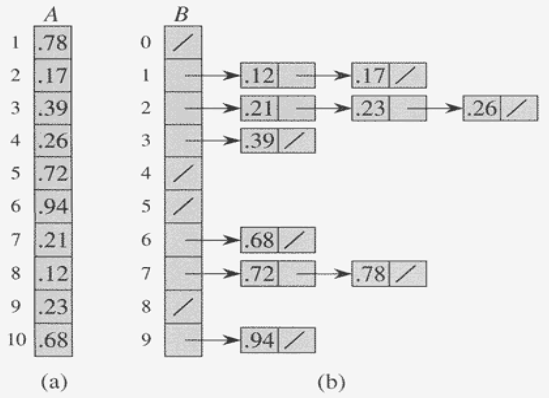


Figure 8.4 The operation of BUCKET-SORT. **(a)** The input array $A[1..10]$. **(b)** The array $B[0..9]$ of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

Bucket Sort average complexity is O(n) and worst case complexity is $O(n^2)$

Multilist Sort or Distribution Sort

Idea is to use each list for certain range of keys. The set of all possible values of the keys is partitioned into m parts, we provide additional storage of M list heads.

Suppose we have 16 keys used in our example and divided into 4 parts 0-249, 250-499,500-749,750-999.

	4 items entered	8 items entered		12 items entered		16 items entered
List 1	061,087	061,087,170		061,087,154,170		061,087,154,170
List 2		275		275,426		275,426
List3	503,512	503,512		503,509,512,653		503,509,512,612,653,677,703
List 4		897,908		897,908		765,897,908

L20: Graphs

A graph is a pair(V,E) where V is set of nodes called vertices E is a collection of pair of vertices called edges. vertices and edges are positions and store elements

Directed Edge: ordered pair of vertices(u,v) where first vertex u is the origin and second vertex v is the destination

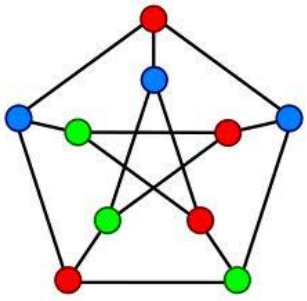
Undirected Edge:unordered pair of vertices for example distance between two cities

In Directed graph all the edges are directed and in undirected graph all the edges are undirected

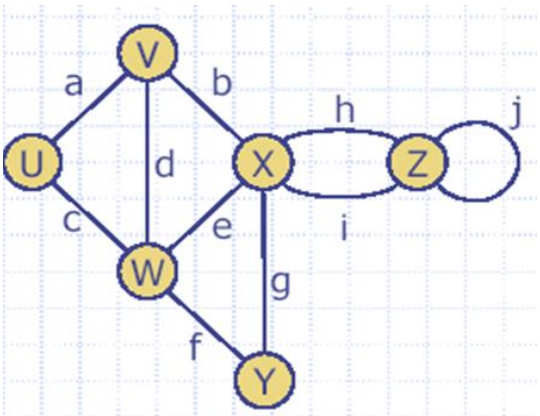
Electronic circuits, printed circuit boards, Integrated Circuits, Transportation network, Highway network, rail network, flight network, traffic network, electricity network, water network, Computer networks, LAN

Web , Databases- ER diagrams

End vertices of an edge U and V are the endpoints of a



Peterson Graph 3-regular



Edge incident on a vertex a, d,b are incident on v

adjacent vertices u and v are adjacent

Degree of a vertex : x has degree 5

If All vertices have the same degree it is called the regular graph

parallel edges: h and i are parallel edges

Self loop : j is a self loop

path-sequence of alternating vertices and edges

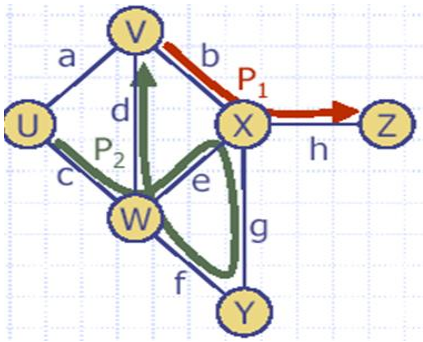
begins with a vertex and ends with a vertex

each edge is preceded and followed by its endpoints

Simple path- a path such that all edges and vertices are distinct

v,b,x,h,z is a simple path

u,c,w,e,x,g,y,f,w,d,v is not a simple path



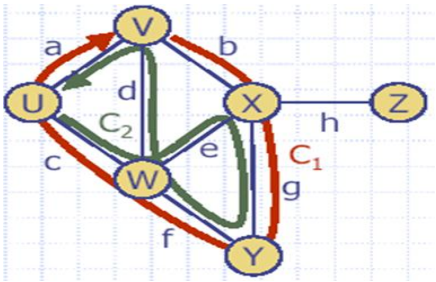
cycle- circular sequence of alternating vertices and edges

each edge is preceded and followed by its endpoints

Simple cycle : such that all its vertices and edges are distinct

v,b,x,g,y,f,w,c,u,a,v is a simple cycle

u,c,w,e,x,g,y,f,w,d,v,a, u is not a simple cycle



incidentedges(v)

endvertices(e)

isdirected(e)

origin(e)

destination(e)

opposite(v,e)

areadjacent(v,w)

insertvertex(o)

insertedge(v,w,o)

insertdirectededge(v,w,o)

removevertex(v)

removeedge(e)

numvertices()

numedges()

Sub Graph: A Graph consisting of Subset of edges and subset of vertices of another graph is a subgraph of that graph.

A connected graph without cycles is a tree

A Collection of trees is called a forest. m=no. of edges n= no. of vertices

fro a trees m=n-1 if m<n-1 Graph is not connected

Incidence matrix

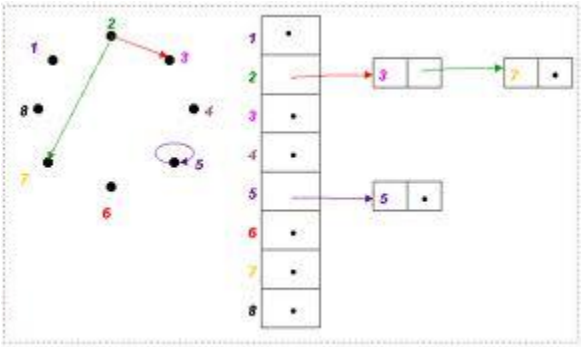
The graph is represented by a matrix of size |V| (number of vertices) by |E| (number of edges) where the entry [vertex, edge] contains the edge's endpoint data (simplest case: 1 - incident, 0 - not incident).

Adjacency matrix

This is an n by n matrix A, where n is the number of vertices in the graph. If there is an edge from a vertex x to a vertex y, then the element $a_{x,y}$ is 1 (or in general the number of xy edges), otherwise it is 0. Every element in matrix can be represented by a single bit. However in case of sparse graphs it can be a waste of space and time. In computing, this matrix makes it easy to find subgraphs, and to reverse a directed graph

Adjacency List

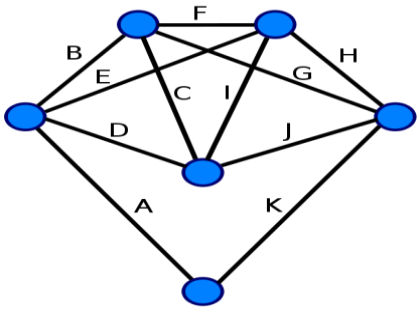
an array indexed by vertex numbers points to a singly-linked list of the neighbors of each vertex.

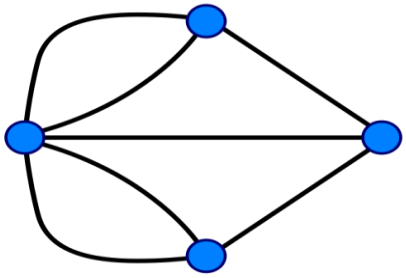


In a complete graph each pair of vertices is joined by an edge, that is, the graph contains all possible edges.

In a bipartite graph, the vertices can be divided into two sets, W and X, so that every edge has one vertex in each of the two sets.

A planar graph can be drawn in a plane with no crossing edges (i.e., embedded in a plane). Eulerian path is a path in a graph which visits each edge exactly once and returns to the starting vertex. A Graph has a Eulerian Path if all vertices have even degree.





Konigsberg

Hamiltonian path (or traceable path) is a path in an undirected graph which visits each vertex exactly once. Hamiltonian cycle (or Hamiltonian circuit) is a cycle in an undirected graph which visits each vertex exactly once and also returns to the starting vertex.

Independent set or stable set is a set of vertices in a graph, no two of which are adjacent. That is, it is a set *I* of vertices such that for every two vertices in *I*, there is no edge connecting the two.

A maximum independent set is a largest independent set for a given graph *G*.

Vertex cover of a graph is a set of vertices such that each edge of the graph is incident to at least one vertex of the set. A vertex cover of a graph *G* is a set *C* of vertices such that each edge of *G* is incident to at least one vertex in *C*. The set *C* is said to *cover* the edges of *G*. The following figure shows examples of vertex covers in two graphs (the set *C* is marked with red).

Minimum vertex cover is a vertex cover of smallest possible size. The vertex cover number τ is the size of a minimum vertex cover. The following figure shows examples of minimum vertex covers in two graphs.

Clique in an undirected graph $G = (V, E)$ is a subset of the vertex set $C \subseteq V$, such that for every two vertices in *C*, there exists an edge connecting the two. A maximum clique is a clique of the largest possible size in a given graph