Data Structures

:Introduction Dr. Sudhanshu

Gupta

What is Data structures

- Set of Domains, Set of functions, set of Axioms • An arrangement of data in a computer's memory or disk storage.
  - E.g. arrays, linked lists, queues, stacks, binary trees, and hash tables.
- Data contained in these data structures are manipulated as in searching and sorting. – Many algorithms apply directly to a specific data structures.
- Let the input and output be represented in a way that can be handled efficiently and effectively.

Examples

array

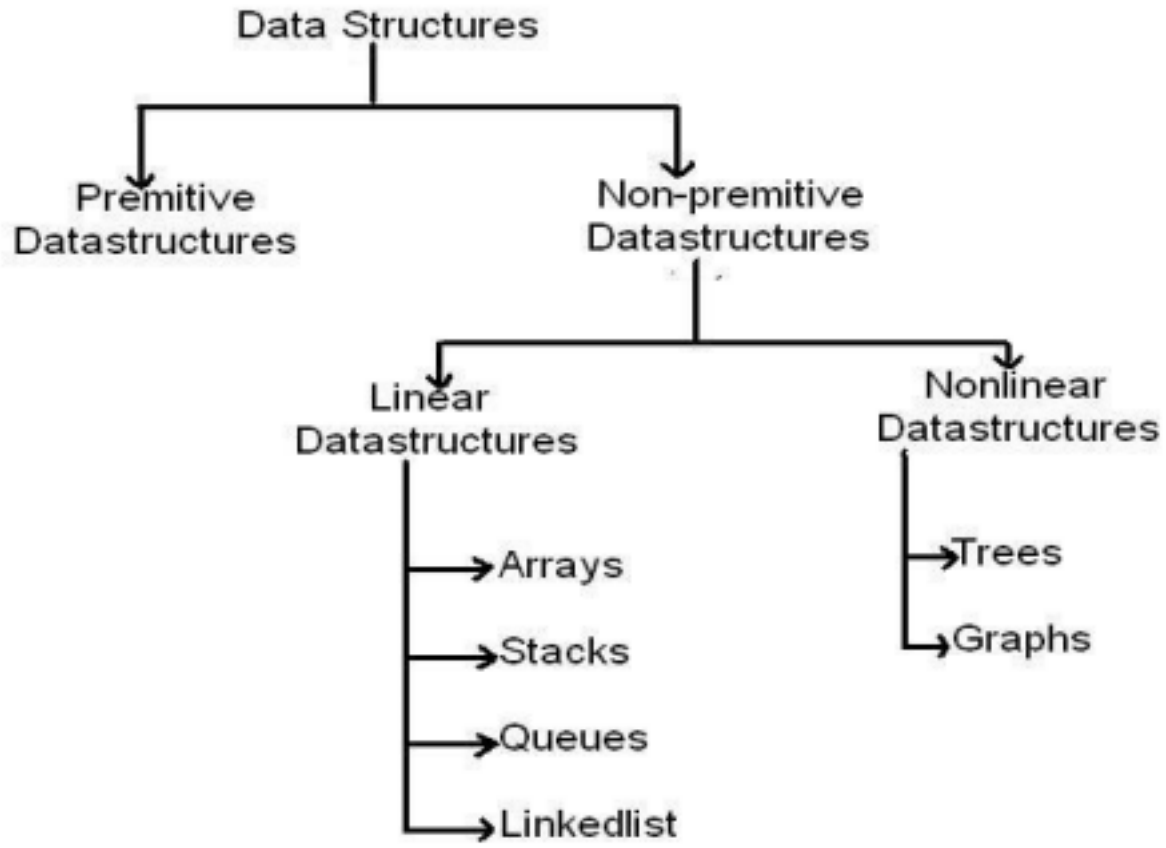
Linked list

queue

tree

stack

Types of Data Structures



Data Structures

- Two Types
  - Primitive data types
    - Directly supported by the machine. i.e. Any operation can be performed in these data items.
    - Integer, Float, Double, Character, Boolean
  - Non Primitive data types
    - Do not allow any specific instructions to be performed on the Data items directly.
    - Arrays, Structures, Unions, Class etc.
- Classification
  - Linear Data structures
    - involve arranging the elements in Linear fashion.
    - E.g. Stacks ,Queue ,Lists
  - Non-Linear Datastructures
    - involve representing the elements in Hierarchical order.
    - Eg:Trees, Graphs.

Data structure operations

- The data in the data structure is processed using various operations :
  - Traversing : To visit or process each data exactly once in the data structure
  - Searching : To search for a particular value in the data structure for the given key value.
  - Inserting : To add a new value to the data structure – Deleting : To remove a value from the data structure – Sorting : To arrange the values in the data structure in a particular order.
  - Merging : To join two same type of data structure values
- An algorithm manipulate data in some way – The way we choose to organize our data directly affects the efficiency of our algorithm

Algorithm

- A computational method for solving a problem. • An algorithm is a finite set of instructions which if followed accomplish a particular task.
- A sequence of steps that take us from the input to the output.
- In addition every algorithm must satisfy following criteria:
  - zero or more input
  - at least one output
  - Each instruction must be clear and unambiguous
  - Algorithm must end after finite number of steps – Every instruction must be feasible.

Algorithm

- An algorithm must be
  - Correct
    - It should provide a correct solution according to the specifications.
  - Finite
    - It should terminate.
  - General
    - It should work for every instance of a problem
  - Efficient
    - It should use few resources (such as time or memory).

Designing a solution to problem

- Data Abstraction
  - Solution that retains the general characteristics of "data and operations" model
  - No dependency on the type of data or method of implementation.
  - Code reusability.
- Encapsulation
  - Organization of data and the operations that can be performed on it are implemented should be hidden.
  - Data can be manipulated in a controlled way, only through an interface. The internal details are hidden.
  - Higher maintainability : the internal organization/implementation can be modified/improved without changing the interface.
  - Outside objects cannot interfere with the internal organization, inadvertently corrupting it.
- Information hiding
  - A solution typically consists of different modules that interact with one another. – Information hiding is the idea of concealing details from other modules that do not need to know those details.

Abstract Data Types (ADTs)

• A contract between the user of a data structure and its implementer. • It specifies:
– Type of data stored (e.g. Int, char, float, any structure)
– Operations allowed on it i.e. methods, with parameter and return types
– Error Conditions
• Benefits
– Encapsulation
– Division of labor
– Promotes code sharing
– Cheaper sub-contracts
– Facilitates unit-testing
• A Data Structure is a construct that implements a particular

Abstract Data Type Example : List

- List ADT : models a sequence of positions storing arbitrary objects/values
- Can be implemented in various ways:
  - Array or singly-linked or doubly-linked
- Accessor methods:
  - first(), last() , prev(p), next(p)
- Update methods:
  - replace(p, e)
  - insertBefore(p, e), insertAfter(p, e), insertFirst(e), insertLast(e)

– remove(p)
- Convenience methods:
  – isEmpty()

Abstract Data Type Example : indexed-List

- Data:
  – An indexed sequence of items of some type
  – lower index
  – upper index
- Operations:
  – Access item at position (index) i
  – Add an item at position i
  – Remove an item at position i
  – Change the bounds

- Find
- Update
- Retrieve
- Length

• Most operations refer to a certain position in the list.

Implementation of list data structure

• Contiguous memory
  - Use an array
  - Random access capability is good for retrieval if we use an index for element access
  - list ADT does not provide random access.
  - Need to shift elements every time we insert or delete
  - Need to reallocate whenever the array fills up.
• Linked memory

– Use a node structure to store the data and a pointer to the next node, to create a chain of nodes.

– Require more space but insert/delete do not require shifting. – However, deleting requires us to traverse the whole list in order to access the predecessor of the node to be deleted.

Books

• Data Structures, Algorithms and Applications in C++, Sartaj Sahni, Second Edition, University Press India Private Limited, 2005 • Introduction to the design and analysis of algorithms by Anany Levitin, 3rd Edition, Pearson Publishers, 2012

• Fundamental of Data Structures By Ellis Horowitz, Sartaj Sahni • Data Structures Using C and C++ By Y. Langsam, M. J.

Augenstein, A. M. Tenenbaum

- Data Structures and Algorithms By A. V. Aho, J. E. Hopcroft, J. D. Ullman
- Introductions to algorithms by Thomas H.Cormen, Leiserson, Rivest and Stein.
- Schaum's Outline Series, Theory and problems of Data Structures By Seymour Lipschutz

Data Structures : Algorithm and Analysis

Algorithmic Analysis

- Characterize the execution behavior of algorithms
  - independent of a particular platform, compiler, or language.
  - Abstract away the minor variations

– and describe the performance of algorithms in a more theoretical, processor independent fashion.

- Estimating the resources algorithm requires.
  - Time : How long will it take to execute?
    - Impossible to find exact value : Depends on implementation, compiler, architecture
    - So measure of time used is : number of steps/simple operations
  - Space
    - Amount of temporary storage required
    - don't count the input

Time Analysis

- Compute the running time of an algorithm.
- Compare the running times of algorithms that solve the same problem

- Time it takes to execute an algorithm usually depends on the size of the input:
  - Algorithm's time complexity is expressed as a function of the size of the input.
- Two different data sets of the same size may result in different running times
  - e.g. A sorting algorithm may run faster if the input array is already sorted.
- As the size of the input increases, one algorithm's running time may increase much faster than another's
  - The first algorithm will be preferred for small inputs but the second will be chosen when the input is expected to be large.

Time Analysis

- Discover how fast the running time of an algorithm increases as the size of the input increases.
  - This is called the order of growth of the algorithm
- The running time of an algorithm on an input of size n depends on the way the data is organized
  - Need to consider separate cases depending on whether the data is organized in a "favorable" way or not.
- Best case analysis
  - Given the algorithm and input of size n that makes it run fastest (compared to all other possible inputs of size n), what is the running time?
- Worst case analysis
  - Given the algorithm and input of size n that makes it run slowest (compared to all other possible inputs of size n), what is the running time?
- Average case analysis
  - Given the algorithm and a typical, average input of size n, running time?

Time Analysis of different

algorithms

- Iterative algorithms
  - Concentrate on the time it takes to execute the loops
- Recursive algorithms
  - Recursive function expressing the time and solve it.
- Different type of approaches
  - A brute force approach : linear search, worst case is (d x N)
  - divide and conquer, worst case is (c x log N)
  - Constants are unknown and largely irrelevant.

Algorithm Analysis

- As the size of the input increases : determine the rate of growth
  - Example 1:
    - In n^2+4n , n^2 grows much faster than n. So concentrate on the term n^2.
  - Example 2:
    - Both n^2+4n and n^2 are quadratic functions. They grow at approximately the same rate. considered them "equivalent".
- Find upper (and lower) bounds for the order

of growth of an algorithm.

– Big Oh, Big Omega, Big Theta

Algorithm Analysis :Big Oh

- The asymptotic execution time of the algorithm.
- The most common method for discussing the execution time of algorithms.
- Maximum time required by an algorithm for all input values.

- Describes the worst-case of an algorithm time complexity

Algorithm Analysis :Big Oh

- A function f(n) is O(g(n)) if there exist positive constants c, n0>0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n0$ – $c \cdot g(n)$ is an upper bound of f(n) for large n
  - There is a point n0 such that for all values of n that are past this point, f(n) is bounded by some multiple of g(n). – Thus if f(n) of the algorithm is $O(n^2)$ then, ignoring constants, at some point we can bound the running time by a quadratic function of the input size.
  - Given a linear algorithm, it is technically correct to say the running time is $O(n^2)$. O(n) is a more precise answer as to

the Big Oh bound of a linear algorithm.

Algorithm Analysis: Big Oh

- Examples:
  - f1(n) = $3n^2+2n+1$ is $O(n^2)$
  - f2(n) = 2n is O(n)
  - f3(n) = 3n+1 is also O(n)
    - f2(n) and f3(n) have the same order of growth.
  - $8n^2 + 10n * \log(n) + 100n + 10^{20} = O(n^2)$ – $3\log(n) + 2n^{1/2} = O(n^{1/2})$

– $2^{100} = O(1)$
– $T_{linearSearch}(n) = O(n)$
– $T_{binarySearch}(n) = O(log(n))$

Algorithm Analysis: Big Oh

- To show that f(n) is O(g(n)),
  – find appropriate values for the constants c, n0
- To show that f(n) is NOT O(g(n)), –

assume that you have found constants c, n0 for which $f(n) \leq c \cdot g(n)$ and

– try to reach a contradiction: show that f(n)

cannot possibly be less than c·g(n) if n>n0

Algorithmic Analysis : Big Omega

- Minimum time required by an algorithm for all input values.
- Describes the best-case of an algorithm time complexity
- A function f(n) is $\Omega(g(n))$ if there exist constants c, n0>0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n0$
  - c·g(n) is a lower bound of f(n) for large n

- Big Oh is similar to less than or equal, an upper bound. – Big Omega is similar to greater than or equal, a lower bound.

- Example:
  - $f(n) = n^2$ is $\Omega(n)$

Algorithm Analysis: Big Theta

- A function f(n) is $\Theta(g(n))$ if it is both $O(g(n))$ and $\Omega(g(n))$:
  - There exist constants c1,c2,n>0 such that $0 \leq c1\ g(n) \leq f(n) \leq c2g(n)$ for all $n \geq n0$
  - f(n) and g(n) have the same order of growth
  - Θ indicates a tight bound.

- Example:
  - f(n) = 2n+1 is Θ(n)
- *Big Theta* f(n) is θ(g(n)) if and only if f(n) is O(g(n))and f(n) is Ω(g(n)).
  - Big Theta is similar to equals.

Relative Rates of Growth

Analysis Type	al Expression	Relative Rates of Growth
Mathematic	f(r	
	f(n	

Big O $f(n) \leq g(n)$ Big Ω $f(n) \geq g(n)$

Big θ $f(n) = \theta(g(n))$ $f(n) = g(n)$

Algorithm Analysis

- f(n) is the actual growth rate of the algorithm.
- g(n) is the function that bounds the growth rate.
  - may be upper or lower bound
- f(n) may not equal g(n).
  - constants and lesser terms ignored because it is a *bounding function*
- Big O is more commonly used

- Additional precision offered by Big Theta - used by researchers

Assumptions in Big O Analysis

- Once found accessing the value of a primitive is constant time

```
x = y;
```

- Mathematical operations are constant time

```
x = y * 5 + z % 3;
```

- if statement: constant time if test and maximum time for each alternative are

constants

```
if( iMySuit ==DIAMONDS || iMySuit == HEARTS)
  return RED;
else
  return BLACK;
```


Loops

- Fixed-Size
  - for(i=0; i <= 10; i++)
  - { for(j = 0; j<10; j++)
  - { s = i+j; }
  - }
  - A constant number of iteration - considered to execute in constant time

don't depend on the size of some data set • Loops That Work on a Data Set

- public double minimum(double[] values,int n)

```
{ double minValue = values[0];
  for(int i = 1; i < n; i++)
    if(values[i] < minValue)
      minValue = values[i];
  return
minValue; }
```

- Number of executions of the loop depends on n.
- Actual number of executions is (n - 1).
- The run time is O(N).

Execution Times

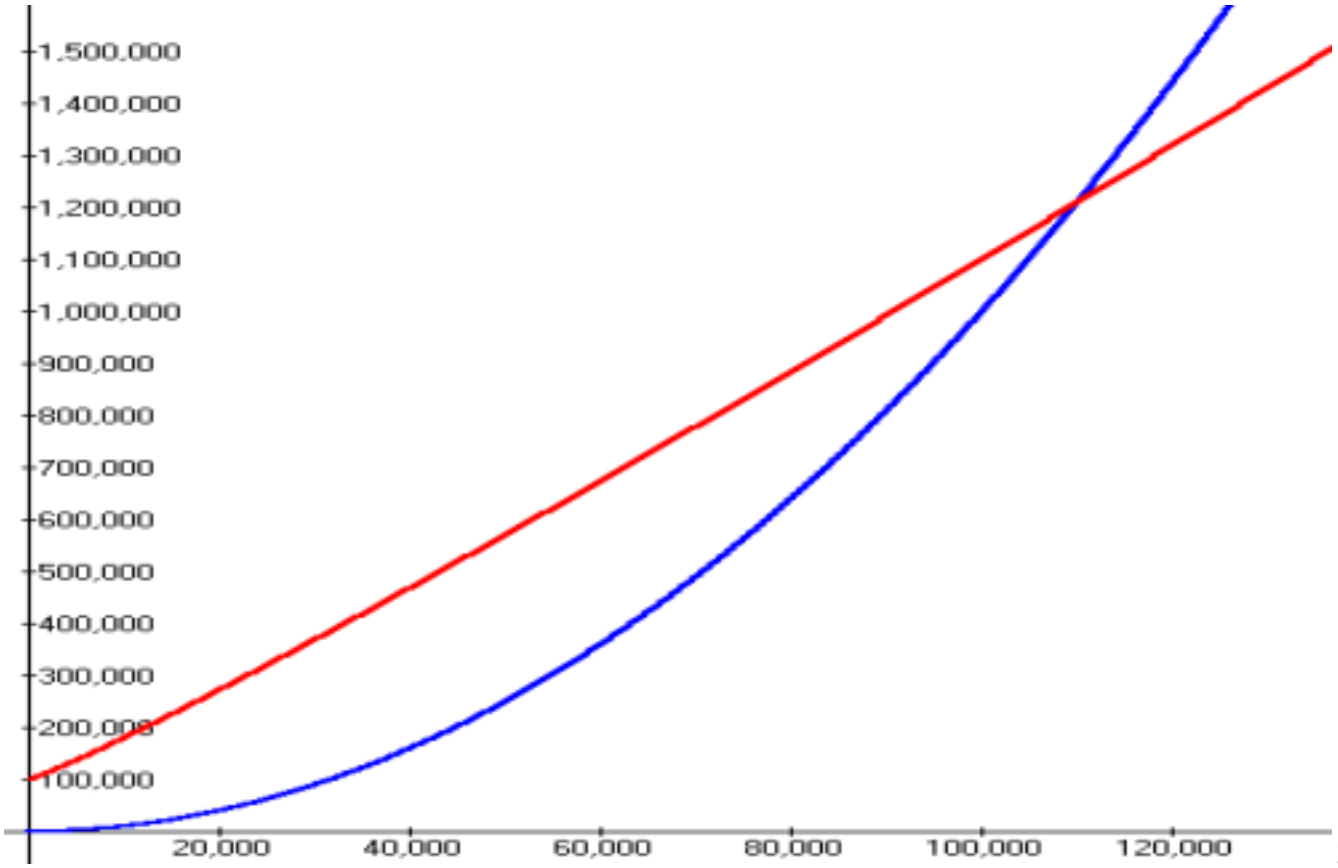
- When adding algorithmic complexities the larger value dominates.
  - $x^2/10000 + 2x \log_{10} x + 100000$
    - x^2term dominates even though it is divided by

10000?

- What if we separate the equation into (x^2/10000) and (2x log x + 100000)?

– For large values the x^2term dominates so the algorithm is O(N^2).

Execution Times



Ranking of Algorithmic Behaviors

Function Common Name

N! factorial

2^N Exponential

N^d, d > 3 Polynomial

N^3 Cubic

N^2 Quadratic

N N

N log N

N Linear

N Root - n

log N Logarithmic

1 Constant

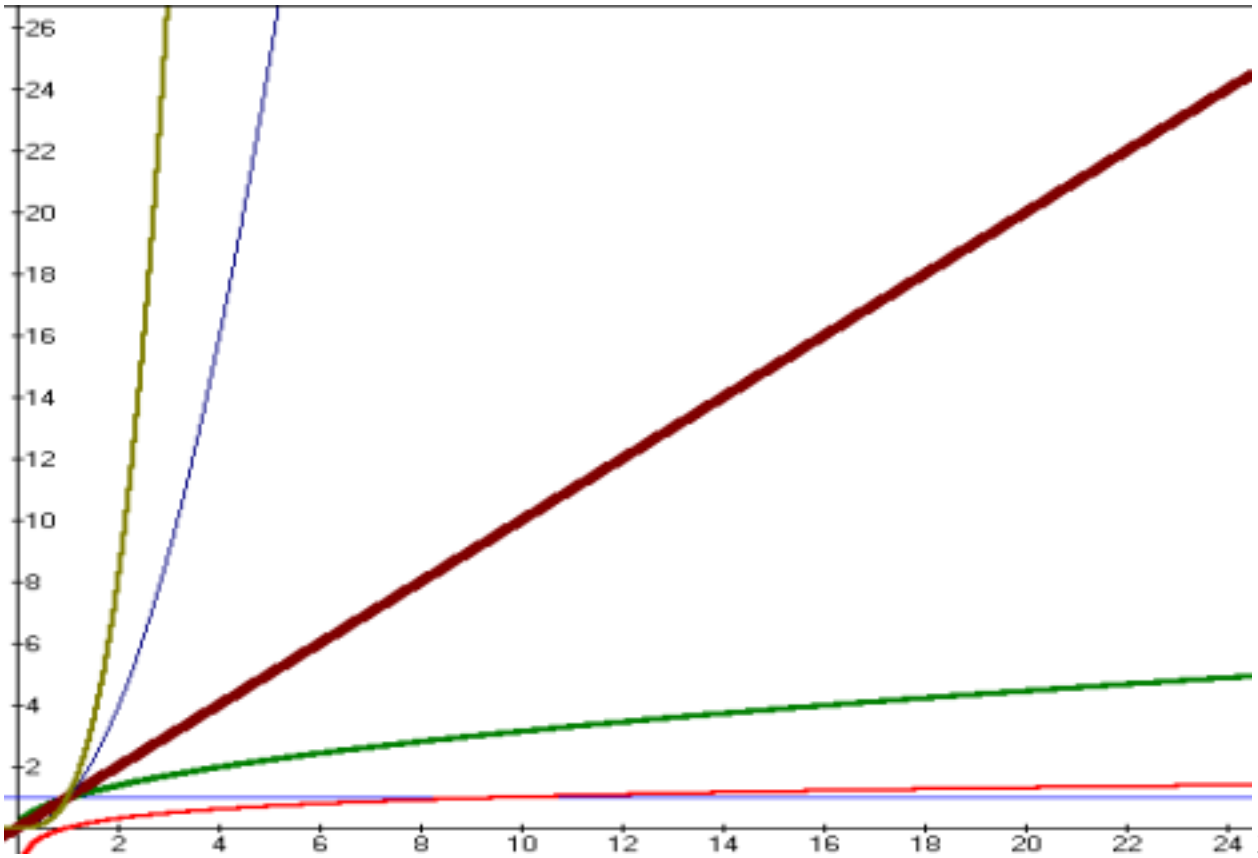
Running Times

- Assume N = 100,000 and processor speed is 1,000,000 operations per second

Function Running Time 2^N over 100 years N^3 31.7 years N^2 2.8 hours N N 31.6 seconds N log N 1.2 seconds N 0.1 seconds

N 3.2×10^{-4} seconds log N 1.2×10^{-5} seconds

Graphical Results



Recursive algorithms

- Recursive solution: solve a smaller version of the problem and combine the

smaller solutions.

- Example: to find the largest element in an array A[1..n]
  - If n is 1, the problem is easily solvable. Just return A[n]
  - If n>1, then find the largest element of A[1..n-1], compare it to A[n] and return the largest.
- Recursive solutions consist of:
  - Base case(s)
    - The problem is explicitly solved for certain values of the input (generally small values)
  - Recursive step
    - Divide the problem into one or more simpler or smaller parts.
    - Solve each part recursively
    - Combine the solutions of the parts into a solution to the problem
- Rules for successful recursion:
  - Handle the base case(s) first
  - Make sure that the recursive step is applied on a smaller version of the problem.
    - "smaller" means closer to the base case
    - But not too close, too fast. Make certain that the base case will be reached.

Recursive procedures

- Pros
  - Often intuitive, more elegant
  - Result in shorter programs
  - Sometimes, a recursive solution may result in a faster algorithm
  - Usually easier to prove correctness
- Cons:
  - More overhead due to function calls
  - More memory used at runtime
  - Sometimes, not as fast as an iterative solution to the problem • Any problem that can be solved recursively can be solved iteratively • Choose recursion when
  - you have a recursive data structure
  - the recursive solution is easier to understand/debug
- Do not choose recursion when
  - performance is an issue
- Examples where recursion is better
  - Towers of Hanoi, certain sorting algorithms

Time Analysis

- A recursive algorithm contains a call to itself.
- Computing the running time of a recursive algorithm involves solving a recurrence equation: one that describes the running time in a recursive manner.

Solving Recurrences

- Recursive algorithms we get a recurrence relation for time complexity.
- We get running time on an input of size n as a function of n and the running time on inputs of smaller sizes.
  - For example Time complexity of Merge Sort can be written as $T(n) = 2T(n/2) + cn$
  - similarly Binary Search, Tower of Hanoi, etc.
- Three ways for solving recurrences.
- 1) Substitution Method
- 2) Recurrence Tree Method
- 3) Master Method

Substitution Method

- Make a guess for the solution and then we use mathematical induction to prove the the guess is correct.
  - For example recurrence T(n) = 2T(n/2) + n
  - We guess the solution as T(n) = O(nLogn).
- To prove that T(n) <= cnLogn. We can assume that it is true for values smaller than n.
- T(n) = 2T(n/2) + n
- <= cn/2Log(n/2) + n
- = cnLogn - cnLog2 + n
- = cnLogn - cn + n
- <= cnLogn

Recurrence Tree Method

- Draw a recurrence tree and calculate the time taken by every level of tree.
- Finally, we sum the work done at all levels. • We start from the given recurrence and keep drawing till we find a pattern among levels. • The pattern is typically a arithmetic or geometric series.

Recurrence Tree Method

- $T(n) = T(n/4) + T(n/2) + cn^2$
- cn^2
- / \
- T(n/4) T(n/2)

- If we further break down the expression T(n/4) and T(n/2), we get following recursion tree.
- cn^2
- / \
- $c(n^2)/16$ $c(n^2)/4$
- / \ / \
- T(n/16) T(n/8) T(n/8) T(n/4)
- cn^2
- / \
- $c(n^2)/16$ $c(n^2)/4$
- / \ / \
- $c(n^2)/256$ $c(n^2)/64$ $c(n^2)/64$ $c(n^2)/16$
- / \ / \ / \ / \

Recurrence Tree Method

- To know the value of T(n), we need to calculate sum of tree nodes level by level.

- T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ... • The above series is geometrical progression with ratio 5/16.
- To get an upper bound, we can sum the infinite series. i.e $(n^2)/(1 - 5/16)$ which is $O(n^2)$

Master Method

- Direct way to get the solution.
- Works only for following type of recurrences or for recurrences that can be transformed to following type.
- T(n) = aT(n/b) + f(n) where a >= 1 and b > 1

- There are following three cases:
  - **1.** If $f(n) = \Theta(n^c)$ where $c < \text{Log}_b a$ then $T(n) = \Theta(n^{\text{Log}_b a})$
  - **2.** If $f(n) = \Theta(n^c)$ where $c = \text{Log}_b a$ then $T(n) = \Theta(n^c \text{Log } n)$
  - **3.**If $f(n) = \Theta(n^c)$ where $c > \text{Log}_b a$ then $T(n) = \Theta(f(n))$

Master Method

- Derived from recurrence tree method.
- If we draw recurrence tree of $T(n) = aT(n/b) + f(n)$, we can see that the work done at root is $f(n)$ and work done at all leaves is $\Theta(n^c)$ where c is $\text{Log}_b a$.
- And the height of recurrence tree is $\text{Log}_b n$ – In recurrence tree method, we calculate total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1).

- If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (Case 2).
- If work done at root is asymptotically more, then our result ~~becomes work done at root (Case 3).~~

Master Method Examples

- Some algorithms whose time complexity can be evaluated using Master Method
  Merge Sort: $T(n) = 2T(n/2) + \Theta(n)$.
  - It falls in case 2 as c is 1 and Logba] is also 1. So the solution is $\Theta(n Logn)$
- Binary Search: $T(n) = T(n/2) + \Theta(1)$.
  - It also falls in case 2 as c is 0 and Logba is also 0. So the solution is $\Theta(Logn)$
- 1) It is not necessary that a recurrence of the form $T(n) = aT(n/b) + f(n)$ can be solved using Master Theorem. The given three cases have some gaps between them.

– For example, the recurrence T(n) = 2T(n/2) + n/Logn cannot be solved using master method.

- 2) Case 2 can be extended for f(n) = Θ(ncLogkn)
  If f(n) = Θ(ncLogkn) for some constant k >= 0 and c = Logba, then
  T(n) = Θ(ncLogk+1n)

Data Structures :

Lists Sudhanshu Gupta

List ADT

- A sequence of elements together with these operations:
  - Initialize the list.
  - Determine whether the list is empty.
  - Determine whether the list is full. –

  Find the size of the list.
  - Insert an item anywhere in the list.
  - Delete an item anywhere in a list.
  - Go to a particular position in a list.

Implementation of list data structure

- Contiguous memory
  - Use an array
  - Random access capability is good for retrieval if we use an index for element access
  - list ADT does not provide random access.
  - Need to shift elements every time we insert or delete
  - Need to reallocate whenever the array fills up.
- Linked memory
  - Use a node structure to store the data and a pointer to the next node, to create a chain of nodes.
  - Require more space but insert/delete do not require shifting. – However, deleting requires us to traverse the whole list in order to access the predecessor of the node to be deleted.

Array is a variable: holds multiple values of same type ,stored in contiguous memory locations.

- Memory allocated in the **Stack** section.
- Single dimensional arrays
- Multidimensional arrays

Single-Dimensional Arrays

typename variablename[size]

- typename is any type
- variablename is any legal variable name
- size is a number the compiler can figure out
- For example

  int a[10];
- Defines an array of integers with subscripts ranging from 0 to 9
- There are 10*sizeof(int) bytes of memory reserved for this array.
- a[0]=10; x=a[2]; a[3]=a[2]; etc.
- scanf("%d",&a[3]);

Initialization of arrays can be done by a comma separated list at the time of definition

int array [4] = { 100, 200, 300, 400 };

This is equivalent to:

```
int array [4];
array[0] = 100;
array[1] = 200;
array[2] = 300;
array[3] = 400;
```

Compiler can figure out the array size for you:

```
int array[] = { 100, 200, 300, 400};
```

Example

```
#include <stdio.h>
int main() {
```

```
float expenses[12]={10.3, 9, 7.5, 4.3, 10.5, 7.5, 7.5, 8, 9.9, 10.2, 11.5,
   7.8};
int count,month;
float total;
for (month=0, total=0.0; month < 12; month++)
{
     total+=expenses[month];
}
for (count=0; count < 12; count++)
   printf ("Month %d = %.2f K$\n", count+1,
expenses[count]); printf("Total = %.2f K$, Average = %.2f
K$\n", total, total/12); return 0;
}
```

Multidimensional Arrays

- Definition has additional subscripts.
- int a [4] [3] ;

  defines a two dimensional array

a is an array of int[3];

Initializing Multidimensional Arrays

int a[4] [3] = { {1, 2, 3} , { 4, 5, 6} , {7, 8, 9} , {10, 11, 12} };

Also can be done by:

```
int a[4] [3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

is equivalent to

```
a[0][0] = 1;
a[0][1] = 2;
a[0][2] = 3;
a[1][0] = 4;
...
a[3][2] = 12;
```

Example

```
#include <stdio.h>
#include <stdlib.h>
int main () {
    int random1[8][8];
    int a, b;
    for (a = 0; a < 8; a++)
```

```
      for (b = 0; b < 8; b++)
                  random1[a][b] = rand()%2;
    for (a = 0; a < 8; a++)
    {
      for (b = 0; b < 8; b++)
          printf ("%c " , random1[a][b] ? 'x' :
      'o'); printf("\n");
    }
  return 0;
}
```

Singly Linked Lists

- A concrete data structure consisting of a sequence of nodes
  - Each node stores – Element

– link to the next node Start or head

next

Element Pointer to next node

∅
i.e.
Null

Linked List : Insert

- Step through list to find the element that comes before item and the one after. • Change the new item's next value to be the position of the next element.
- if the new item is not the first element – change the previous element's next value to be the position of the new item.
- if the new item is the first element, change the start of the list to equal the new item's position.
- increase count.

Singly linked list : creating node

```
struct node
{
    int data;
    struct node *next;
};

struct node * createNode(int num){
   struct node *newNode;
    newNode= (struct node *)malloc(sizeof(struct node));
    newNode->data=num;
    newNode->next = NULL;
    return newNode;
}
```

Singly linked list : Insert at front

```
Struct node * addFront( int num, struct node * head )
{
    struct node *newNode;
    newNode= createNode(num);
    if (head== NULL){
        head=newNode;
     }else{
        newNode->next=head;
        head=newNode;
    }
    return head;
}
```

Singly linked list : append

```
Struct node * append(int num, struct node *
head) {
    struct node * newNode,*ptr;
    newNode= createNode(num);
    ptr=head;
    if(head == NULL){
            head = newNode;
     }else{
       while(ptr->next != NULL)
            ptr=ptr->next;
       ptr->next =newNode;
     }
     return head;
```

}

Singly linked list : Insert

```
Struct node * addafter(int num, int loc, struct node *head)
{
    int i;
    struct node *newNode,*temp,*ptr;
    ptr=head;
    if (head == null){
        head = newNode
    }else{
      for(i=1;i<loc;i++) {
         temp=ptr;
         ptr=ptr->next;
       }
      newNode= createNode(num);
      newNode->next=temp->next;
      temp->next=newNode;
    } //end of if else
Return head;
}
```

Singly linked list : Delete

```
int delete(int num, struct node * head){
    struct node *p, *prev;
    p=head;
    while(p!=NULL) {
            if(p->data==num) {
                if(p==head) { head=p->next; free(p); }
                else { prev->next=p->next; free(p); }
             } else {
                 prev=p;
                 p= p->next;
             }
    }
    return 0;
}
```

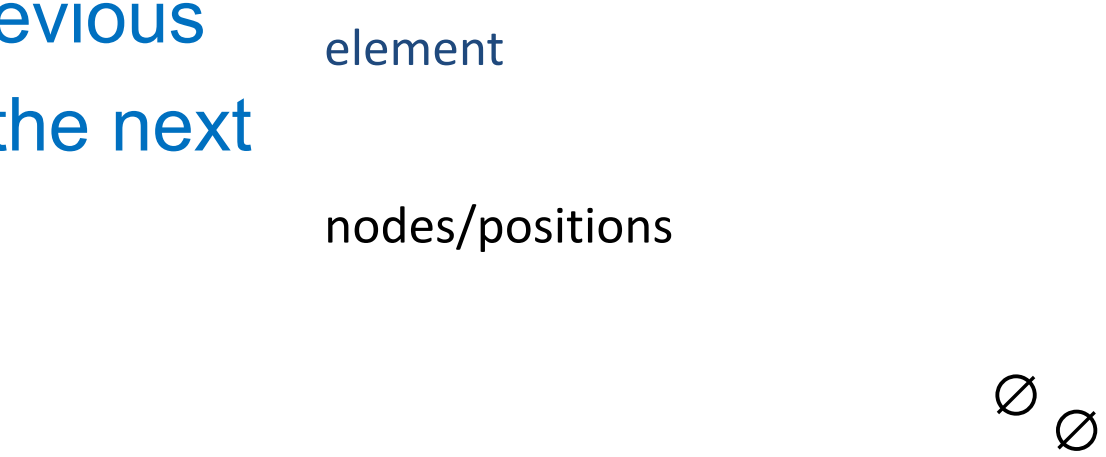
Singly linked list : count & display

```
int count(struct node *
head){ struct node *n=head;
  int c=0;
  while(n!=NULL) {
    n=n->next;
    c++;
  }
  return c;
}
```

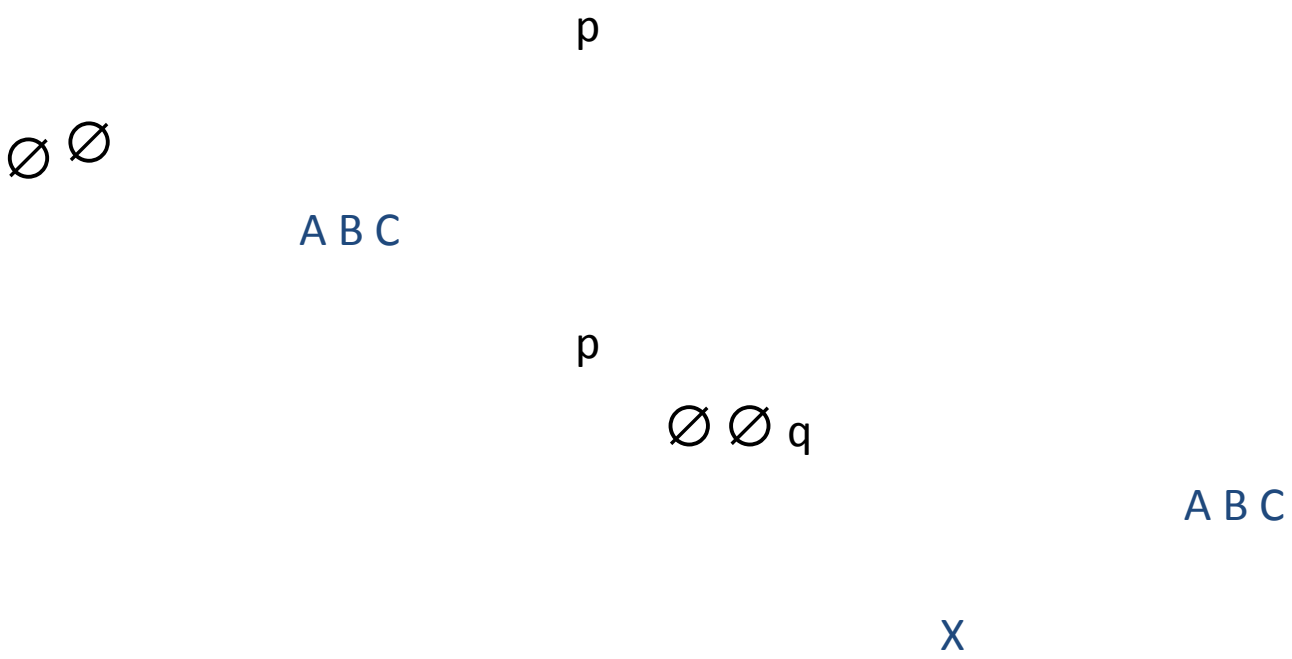
```
void display(struct node
 *head){
    struct node * r=head;
    if(r==NULL){ return; }
    while(r!=NULL) {
          printf("%d\n ",r->data);
          r=r->next;
    }
 }
```

Doubly Linked List

- A Nodes stores:
  - element
  - link to the previous node
  - link to the next node



Insertion



p q

∅ ∅A B X C

Deletion

p

A B C D

A B C

p

D

A B C


```
struct node
{
  int data;
  struct node *next;
  struct node *prev;
};
```

```
struct node * createNode()
{
    int data;
    struct node * temp;
    temp =(struct node *)malloc(sizeof(struct
    node)); temp->prev = temp->next =NULL;
    printf("\n Enter value to node : ");
    scanf("%d", & temp->data);
    return temp;
}
```

Insert at beginning

```
struct node * insertAtBegin(struct node * head){
   struct node * temp = createNode();
   if (head == NULL) {
       head = temp;
   } else{
        temp->next = head;
        head->prev = temp;
        head = temp;
    }
   return head;
}
```

To insert at end

```
Struct node * insertAtEnd(struct node * head) {
    struct node * temp = createNode();
    if (head == NULL) {
      head = temp;
     } else {
       p = head;
       while(p->next !=NULL)
              p =p->next;
        temp->prev = p;
        temp -> next= 0;
    }
```

```
    return head;
}
```


Insert at any position

```
struct node * insertAtAnyPosition(struct node * head){
    int data;
    struct node * temp = head;
    printf("\n Enter value to be inserted : "); scanf("%d", &data);
    temp = createNode();
   if (head == NULL) { head = temp; }
   else {
        p = Head;
        while ( p->data < data && p->next !=0) { p = p->next; }
         temp->prev = p->perv;
         p->prev ->next = temp;
         p->prev = temp;
         temp->next = p;
```

```
}
 return head;
}
```


Delete an element

```
void delete(struct node * head){
int data;
    struct node * p = head;
    printf("\n Enter value to be delete: "); scanf("%d", &data)
  if (head == NULL) {
     printf("\n Error : Empty list no elements to delete"); return;
  } else {
            while ( p->data != data && p->next !=NULL) { prev= p; p= p->next;}
            if (p->next == NULL) {printf(”element not found”);}
            else {
                  if (p==head) { p->next->prev= 0; head =p->next; }
                  else { p->next->prev = p->prev;
                         p-> prev ->next = p->next; }
                  free(p);
            }
```

```
    }
  }
}
```


Traversing

```
void traversing(struct node * head){
    p=head;
    if (p== NULL) {
        printf("List empty to display \n");
        return;
    }
    while (p != NULL) {
        printf(" %d ", p->data);
        p = p->next; }
    }
```

}

A Stack ADT

- A collection of homogeneous elements arranged in a sequence.
  - Last-In, First-Out (LIFO)
- and operations:
  - Initialize the stack.
  - Determine whether the stack is empty.
  - Determine whether the stack is full.
  - Push an item onto the top of the stack.
  - Pop an item off the top of the stack.

- Implementation
  - Contiguous memory : array
  - Linked memory: linked list

Array-based Stack

- A simple way of implementing the Stack ADT
- Add elements from left to right
- A variable keeps track of the index of the top element
- The array storing the stack elements may become full
  - A push operation is not possible

– Limitation of the array-based implementation

Stack ADT

```
#define size 15

struct stack {
 int top;
 int items [size]
} ;

void initialize(struct stack *) ;
void push (int , struct stack *);
```

```
void pop (int*, struct stack*);
int isEmpty(struct stack *);
int isFull(struct stack*);
```


Stack ADT

```
void initalize (struct stack* stPtr)
{
  stPtr->top=0;
}

int isEmpty(struct stack *stPtr)
{
    return stPtr->top ->==0;
}
```

```
int isFull(struct stack *stPtr)
{
  return stPtr->top >=Size ;
}
```

Stack: Push and pop

```
void push (int item, struct stack *stPtr){ if
  (stPtr->top >=Size ) {printf ("stack is full ") }
  else{ stPtr->top++;
          stPtr->items [stPtr->top]=item;}
}

  void Pop (int* item , struct stack *stPtr){ if
  (stPtr->top <=0){ printf("stack is empty");}
    else { *item =stPtr->items [stPtr->top];
```

```
                    stPtr->top--;}
}
```

Applications of Stacks

- Delimiter matching
- Undo sequence in a text editor
- Chain of method calls in the Java Virtual Machine
- Evaluation of expression
  - Infix to postfix
  - Postfix evaluation
- Recursive function calls
- Decimal to binary