Data Structures and Algorithms

Reference Books

- A. M. Tanenbaum, *"Data Structures using C"*, Pearson Education.
- A. V. Aho, J. E. Hopcroft, J. D. Ullman, *"Data Structures and Algorithm"*, Pearson.
- D. Samanta, *"Classic Data Structure"*, PHI.
- S. Lipscutz, *"Data Structures with C"*, TMH.
- R. Kruse, C.L. Tondo, B. Leung, S. Mogalla, *"Data Structures and Program Design in C"*, Pearson.
- D. E. Knuth, *"The Art of Computer Programming"*, Addison-Wesley
- S. Chattopadhyay, D.G. Dastidar, M. Chattopadhyay, *"Data Structures through C Language"*, BPB Publications.

Introduction

Algorithms

- A *finite* set of instructions executed in *sequence* in *finite time*
- Algorithm for finding GCD

step 1: read two positive integers x and y
step 2: divide x by y to get remainder r and quotient q
step 3: if r is zero go to step 7
step 4: assign y to x
step 5: assign r to y
step 6: go to step 2
step 7: **y is the required GCD**, print y
step 8: stop

Algorithms

- Properties of an algorithm
  - *Input*
  - *Output*
  - *Finiteness*
    - For all input data, the algorithm must terminate after a ***finite*** number of steps
  - *Definiteness*
    - Clear and *unambiguous* steps
  - *Effectiveness*
    - Steps must be very basic

Algorithms

- Expressing an algorithm
  - Natural language, flowchart, programming languages
- Designing an algorithm
  - Innovative exercise, no methodology to automatically generate algorithms
  - Use of new techniques and strategies for good algorithms
    - Depends heavily on the organization of data
- Analyzing an algorithm
  - Validation
  - Complexity evaluation
    - Both time and space

Abstract Data Type

- Data type defines a *set of values* and permitted *operations*
- Built-in data types are not enough for most applications
  - Create new data types in terms of structure
    - Operations applicable to structure variables **can not be specified**
  - Concept of abstract data type (ADT) introduced
    - A mathematical model with collection of operations defined on that model

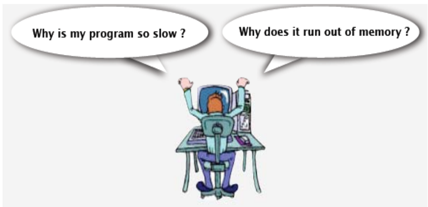
Abstract Data Type

- Define a new data type "SET"
- Operations on ADT "SET"
  - assign (SET A, SET B)
  - SET union (SET A, SET B)
  - SET Intersection (SET A, SET B)
  - int cardinality (SET A)
- ***No limit*** on the number of operations
- ***Implementation aspect*** is not considered

Data Structure

- A scheme to organize data
  - Stack, queue, tree, graph etc.
- Affects the performance of a program for different tasks
- Choice of data structure depends on:
  - ***Nature*** of data
  - ***Processes*** to be performed on the data

Analysis of Algorithms



Analysis of Algorithms

- Used to compare number of algorithms and choose the best one
- Typically, two quantitative metrics:
  - ***Space complexity***
    - Run time storage requirement
  - ***Time complexity***
    - Time required to complete execution
      - Usually depends on ***input size***, i.e., time complexity is a ***function of input size***

Analysis of Algorithms

- Types of analysis
  - Best Case
    - *Lower bound* on cost
    - Determined by *"easiest"* input
    - Provides a goal for all inputs
  - Worst Case
    - *Upper bound* on cost
    - Determined by most *"difficult"* input
    - Provides a guarantee for all inputs
  - Average Case
    - *Expected cost* for *random input*
    - Provides a way to predict performance

Analysis of Algorithms

- Time complexity
  - Very difficult to compute *exact time*
    - Several factors influence execution time which are outside the domain of programmers
      - Programs are translated to machine code
  - Define a function *f(n)* that gives an estimate of **volume of work done** by the algorithm on input size *n*

Analysis of Algorithms

- Big-Oh notation
  - $T(n)=O(f(n))$ if $T(n) \le c f(n)$

    for some constant, $c > 0$, and $n \ge n_0$

    f* is an upper bound for *T

    ie for sufficiently large ***n***
  - If $T(0)=0$, $T(1)=4$, and in general $T(n)=(n+1)^2$, then $T(n)=O(n^2)$
    - Let $n_0=1$, $c=4$, i.e., $\forall n \ge 1, (n+1)^2 \le 4n^2$

Analysis of Algorithms

- **Constant factors may be ignored**
  - $\forall\ k > 0,\ kn$ is $O(n)$
- **Higher powers grow faster**
  - n^r is $O(n^s)$ if $0 \le r \le s$

← **Fastest growing term dominates a sum**

e.g., $3n^3 + 2n^2$ is $O(n^3)$

c + cn + cnlogn is O(nlogn)

← **Polynomial's growth rate is determined by leading term**

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

Analysis of Algorithms

- *T* is *O(g)* is transitive
  - If T is O(g) and g is O(h) then T is O(h)
- Product of upper bounds is upper bound for the product
  - If f is O(g) and h is O(r) then fh is O(gr)
- Two additional notations
- *Ω(g(n))*
  → $T(n) \geq c\, g(n)$
  for some constant, c, and $n > n_0$
- θ(*g(n)*), **for some constants, c_1, c_2 and $n > n_0$**
  → $c_2 g(n) \geq T(n) \geq c_1 g(n)$

g(n) is a lower bound for T

Best and worst case complexities are same

Analysis of Algorithms

- Simple statement
  S=p+q
  - Time Complexity is O(1)
- Simple loops

```
for(i=0;i<n;i++) { s=p+q; }
```

  - Time complexity is n O(1) or O(n)
- Nested loops

```
for(i=0;i<n;i++)
    for(j=0;j<n;j++) { s=p+q; }
```

  - Time Complexity is n O(n) or $O(n^2)$

This part is *O(n)*

Analysis of Algorithms

- Loop index doesn't vary linearly

```
h = 1;
while ( h <= n ) {
      s;
      h = 2 * h;
      }
```

  - h takes values 1, 2, 4, … until it exceeds n
  - There are $1 + \log_2 n$ iterations
  - Complexity O(log n)

Analysis of Algorithms

- Loop index depends on outer loop index

```
for(j=0;j<n;j++)
   for(k=0;k<=j;k++){
     s;
    }
```

  - Inner loop executed
    - 1, 2, 3, …., n times

$$\sum_{i=1} i = \frac{n(n+1)}{2}$$

∴ Complexity $O(n^2)$

Analysis of Algorithms

- Common computing times are
  $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n)$
- *log n*
  - Logarithmic algorithm, cuts down the problem to smaller one
- *n*
  - Linear algorithm
- *n log n*
  - Breaks the large problem into sub-problems, solve sub-problems independently, combine the results
- *2^n*
  - Exponential running time, not suitable for practical use

The LIST ADT

LIST ADT

- Ordered sequence of data items called elements
  - $A_1, A_2, A_3, \ldots, A_N$ is a list of size N
- Size of an empty list is 0
- First element is A_1 called "head"
- Last element is A_N called "tail"

Operations ?

LIST ADT

- **Operations**
  - PrintList
  - Search
  - FindKth
  - Insert
  - Delete
  - Reverse
  - Sorting
  - MakeEmpty

LIST ADT

- **Example:**

  the elements of a list are

  34, 12, 52, 16, 12

  - Search (52) → 2
  - Insert (20, 3) → 34, 12, 52, 20, 16, 12
  - Delete (52) → 34, 12, 20, 16, 12
  - FindKth (3) → 16

LIST ADT

- You can see the difference between arrays and lists when you delete items.

Array Implementation of LIST

- Need to define a size for array
  - High overestimate (waste of space)
- Operations Running Times

PrintList, Search — O(N)

Insert, Delete — O(N) **(on avarage half needs to be moved)**

FindKth, Next, Previous — O(1)

Array Implementation of LIST

```
int search (int a[], int n, int val)
{
int i=0;
for (i=0; i<n;i++)
        if (a[i] = = val)
                break;          If a match is found come out of the for loop
        if (i = = n)
                return -1;      Val not found in a[ ]
        else
                return i;       Return the index of val in a[ ]
}
```

Best Case: **O(1)**

Average Case:

number of comparisons could be 1,2,..,n depending on the position of val.

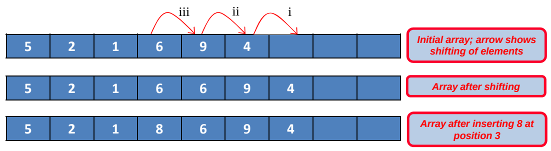
=(1+2+…+n)/n

=(n+1)/2

=**O(n)**

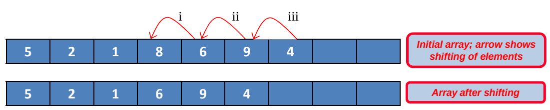
Worst Case: **O(n)**

Array Implementation of LIST



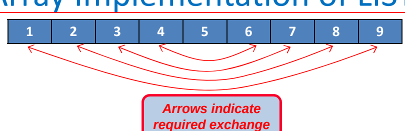

```
void insert (int a [], int n, int j, int val)
{
    int i;
    for ( i = n-1; i >= j; i-- )
          a[i+1] = a[i];      Shifting of elements to the right
    a[j] = val;
}
```


Array Implementation of LIST




```
Void delete (int a [], int n, int j)
{
    int i;
    for ( i = j+1; i < n; i++ )
          a[i-1] = a[i];      Shifting of elements to the left
}
```


Array Implementation of LIST




```
void reverse (int a [], int n)
{
    int i, temp;
    for ( i = 0; i < n/2 ; i++ )
          {
                temp = a [i];
                a [i] = a [n-1-i];
                a [n-1-i] = temp;
          }
}
```

Polynomials using Arrays

- Polynomial, P(x) of degree n is:

  $P(x) = a_0 + a_1x + a_2x^2 + \ldots\ldots + a_nx^n$
- Treat Polynomials as ADT
  - *Operations*
    - **Initialization**
    - **Copy**
    - **Add**
    - **Multiply**
    - **Evaluate**

Polynomials using Arrays

- Representation:

```
typedef struct poly
{
        float coeff [1000];
        int degree;
} poly
```

- Polynomial $1 + 4x^2 + 2x^8$ is stored as:

1	0	4	0	0	0	0	0	2

Wasteful representation, most of the elements are Zero

Polynomials using Arrays

- Store only **non-zero** terms a_ix^i
- Define the terms:

```
typedef struct term
{
        float coeff;
        int expo;
} term;
```

- Now, define the polynomial:

```
typedef struct poly
{
        term a[ 1000];
        int no_of_terms;
}poly;
```

Polynomials using Arrays

- Polynomial $1 + 4x^2 + 2x^8$ is stored as:

1,0	4,2	2,8	……

- Addition of polynomials $1 + 4x^2 + 2x^8$ and $3x + 5x^2 + 6x^7$

1,0	4,2	2,8	……

3,1	5,2	6,7	……

Polynomials using Arrays

- Addition of polynomials:

1,0	4,2	2,8	……

3,1	5,2	6,7	……

1,0								

1,0	4,2	2,8	……

3,1	5,2	6,7	……

1,0	3,1							

Polynomials using Arrays

- Addition of polynomials:

1,0	4,2	2,8	……

3,1	5,2	6,7	……

1,0	3,1	9,2						

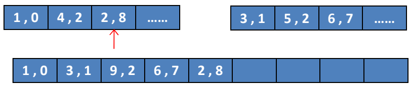
1,0	4,2	2,8	……

3,1	5,2	6,7	……

1,0	3,1	9,2	6,7					

Polynomials using Arrays

- Addition of polynomials:



Polynomials using Arrays

```
for ( i = 0, j = 0, k=0; ( i < P1→no_of_terms) && ( j < P2→no_of_terms); k++)
{
    if (P1→ a [i].expo == P2→ a[j].expo)
    {
        P3→a[k].coeff = P1→a[i].coeff + P2→a[j].coeff;
        P3→a[k].expo = P1→a[i].expo;
        i++; j++;
    }
    else if (P1→ a [i].expo < P2→ a[j].expo)
    {
        copy the term in P1 to P3;          i++;
    }
    else
    {
        copy the term in P2 to P3;          j++;
    }
}
```

For remaining terms in P1 and P2

```
if (i < P1→no_of_terms)
    for (l = i; l < P1→no_of_terms; l++, k++)
    {
        copy the terms in P1 to P3;
    }
else
    for (l = j; l < P2→no_of_terms; l++, k++)
    {
        copy the terms in P2 to P3;
    }
P3→no_of_terms = k;
```

Complexity: O(m+n), m and n are degree of the polynomials

Polynomials using Arrays

- Very large numbers can not be stored in variables of type “int” or “long”!!!!
  - 80 digit number is greater than the maximum value in “long”
- A number can be represented as a polynomial
  - e.g., **$12345 = 1x10^4 + 2x10^3 + 3x10^2 + 4x10^1 + 5x10^0$**
  - Equivalent to ***$P(x) = x^4 + 2x^3 + 3x^2 + 4x + 5$, for $x=10$***

Polynomials using Arrays

- An integer of *n* digits is a polynomial of degree *n-1*:

  $P(x)=\sum_{i=0,n-1} a_i x_i$ for $x=10$, $0 \le a_i \le 9$

- 2000020000000080000900000000001 can be represented as:

1,0	9,9	8,14	2,22	2,27

Bhaskar Sardar, Information Technology Department, Jadavpur University, India

Sparse Matrix

- Most of the elements are **zero** in a matrix

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 5 & 0 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

- Two dimensional array representation is **inefficient**
- Solution:
  - Store only ***non-zero elements***

Bhaskar Sardar, Information Technology Department, Jadavpur University, India

Sparse Matrix

- Treat a sparse matrix as a **ordered LIST** of non-zero elements
- Information regarding each element:
  - row, col, val
- Define the elements:

```
typedef struct element
{
        int row, col, val;
} element;
```

Bhaskar Sardar, Information Technology Department, Jadavpur University, India

Sparse Matrix

- Define the sparse matrix:

```
typedef struct sparsemat
{
        int no_of_nonzero_elements;
        int no_of_rows, no_of_cols;
        element a [100];
}sparsemat;
```

- Example matrix can be represented as follows:

0, 2, 1	1, 0, 2	1, 2, 5	2, 3, 3

Sparse Matrix

- Saving in Space
  - Space required to store m x n matrix of integers is
    m x n x (size of an integer)
  - Space required in ordered LIST
    3 x p x (size of an integer), p is size of array
  - LIST is *advantageous* if
    3 x p < m x n i.e., **p < m x n/3**

Sparse Matrix

- Finding number of non-zero elements in each column:

```
for ( i = 0; i < s→no_of_cols; i ++)
{
    count = 0;
    for (j = 0; j < s→no_of_nonzero_elements; j ++)
        if (s→a[j].col  == i)
            count ++;
    printf (“number of no-zero elements in column %d is %d”, i, count);
}
```

Number of comparisons = p x n
In worst case p = O(mn)
So, complexity is O (mn²)

However, a function using two dimensional representation takes O(mn) time !!!!!!

Sparse Matrix

- Use of programming trick helps to reduce the complexity
- Use an array column so that column [i] stores number of elements in *i*th column

```
for ( i = 0; i < s→no_of_nonzero_elements; i ++)
{
        j = s→a[i].col;
        col [j] = col [j] + 1;
}
```

Complexity is now O(mn)

Multi-dimensional Array

- Computer memory is one dimensional
  - Multi-dimensional arrays are represented as one dimensional array

  e.g., int a[5][10][4][10][5] may be stored as b[10000]

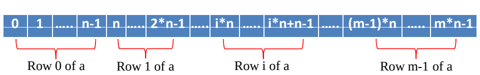
A difficult question:

Which element in "b" contains a particular element of "a"??

- Consider the following example:

Multi-dimensional Array

- Two dimensional array a[m][n] converted to one dimensional array b[mxn] as follows.



- This is **row-major** ordering
- Element a[i][j] is mapped to $(i \times n + j)^{th}$ element in array b

Multi-dimensional Array

- Three dimensional array a[2][3][2] looks as follows:

(0,0,0) (0,0,1) (0,1,0) (0,1,1) (0,2,0) (0,2,1)
(1,0,0) (1,0,1) (1,1,0) (1,1,1) (1,2,0) (1,2,1)

- To reach a[i][j][k], go to a[i][0][0]
  - Number of elements between a[0][0][0] and a[i][0][0] is **i x n x p**
- From a[i][0][0] go to a[i][j][0]
  - Number of elements **j x p**
- From a[i][j][0] go to a[i][j][k]
  - Number of elements **k**
- So, a[i][j][k] mapped to i x n x p + j x p + k

Multi-dimensional Array

- Consider the following m-dimensional array

a[u_0][u_1]....[u_{m-1}]

- The position of a[i_0][i_1]....[i_{m-1}] is

$$\sum_{j=0}^{m-2}\left(i_j * \prod_{i=j+1}^{m-1} u_i\right) + i_{m-1}$$

Linked Lists

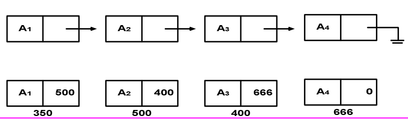
Limitations of Arrays

- Simple, Fast

  but
- Must specify size at construction time
  - Construct an array with space for *n*
    - *n* = twice your estimate of largest collection
  - Actual size is much less than n, wastage of space
  - Tomorrow you'll need *n*+1, overflow
- Shifting of elements during insertion and deletion
  - **More flexible system?**

Linked Lists

- Flexible space use
  - Dynamically allocate space for each element as needed
- Series of nodes
  - Each node of the list contains
    - the data item
    - a pointer to the next node
- Avoids the linear cost of insertion and deletion !



Linked Lists

- Define a node

```
typedef Struct node
{
     int val;
     struct node *next;
} node;
```

Recursive type definition

- Create the Header

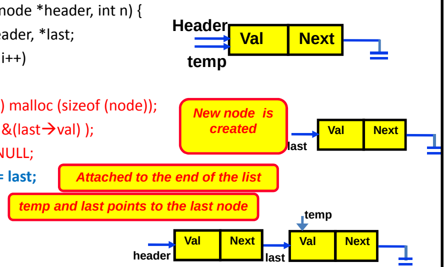
```
node* create_header( int item )
{
    Node* header = (node*) malloc( sizeof(node) );
    header->val = item;
    header->next = NULL;
    return header;
}
```

Header keeps track of the entire list; Carefully handle the header

Linked Lists

- Creation of list with n nodes:

```
void create_list (node *header, int n) {
node *temp = header, *last;
For ( i = 0;  i < n;  i++)
{
    last = (node* ) malloc (sizeof (node));
    scanf ( “%d”,  &(last→val) );
    last→next = NULL;
    temp→next = last;
    temp = last;
}
}
```



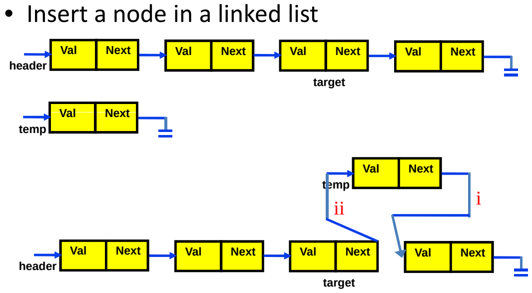
Linked Lists

- Printing content of a list is simple…

```
Void printlist (node *header)
{
        node *temp=header;
        while (temp != NULL)
        {
                printf (“%d”, temp→val);
                temp = temp → next;
        }
}
```


Linked Lists

- Insert a node in a linked list



Linked Lists

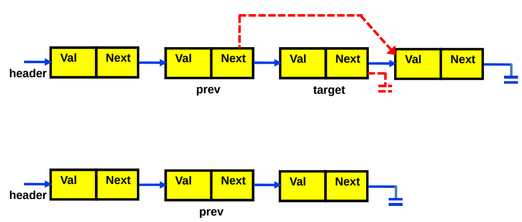
```
node* insert (node * header, int i, node *t) {
    int k;    node *temp = header;
    if ( i == 0)
    {
          t→next = temp;
          header = t;
          return header;
    }
for ( k = 1; ( k < i )  &&  ( temp != NULL );  k++)
          temp = temp→next;
If ( temp == NULL )          return header;
t→next  =  temp→next;
temp→next  =  t;
Return header;    }
```

Insert at the first position

Move to the target node in question

Linked Lists

- Delete a node from linked list



Linked Lists

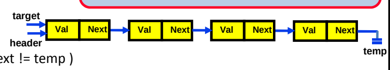
```
node* delete ( node * header, int k)            {
    int  i; node *temp = header, *target;
    If ( k == 0 )
    {
          header = header→next;    free ( temp );       return header;
    }
    target = temp→next;
    for ( i =  1; i < k && target != NULL; i ++)
    {
          temp = target;     target = target→next;
    }
    if ( target == NULL )           return header;
    temp→next = target→next;       target→next = NULL;
    free (target);        return header;                }
```

Linked Lists

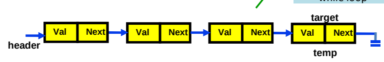
- Printing a linked list in reverse order

```
void printreverse ( node *header )      {
    node * target, *temp = NULL;
    if ( header == NULL)        return;
    while ( temp != header)
    {
        target = header;
        while ( target→next != temp )
            target = target→next;
        printf ( “%d”, target→val );
        temp = target;
    }
}
```

Node n is printed after visiting n nodes
Node n-1 is printed after visiting n-1 nodes
So, complexity=n(n+1)/2 = $O(n^2)$

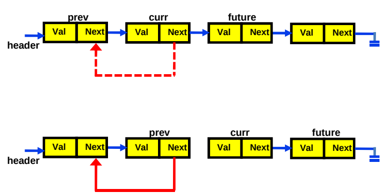


After one iteration of outer while loop



Linked Lists

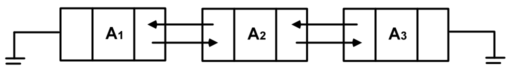
- Reverse a linked list



Linked Lists

```
node * reverse ( node * header)     {
        node *prev = NULL, *future, *curr = header;
        future = curr→next;
        while ( curr→next != NULL)
        {
                curr→next = prev;
                prev = curr;
                curr = future;
                future = future→next;
        }
        curr→next = prev;
        return ( curr );                }
```

Doubly Linked Lists



- Traversing list backwards
  - not easy with **regular lists**
- Insertion and deletion more pointer fixing
- Deletion is easier
  - **Previous node** is easy to find

Doubly Linked Lists

- Define a node:

```
typedef struct node
{
        int  val;
        struct node *prev;
        struct node *next;
} node;
```


Doubly Linked Lists

- Insert a node after target node:

```
Void insert ( node *header, node *new, node *target)
{
      new→next = target→next;
      new→prev = target;

            if ( target→next != NULL)
                  target→next→prev = new;
      target→next = new;
}
```

Set the pointers of node "new"

Inserting at the end

Doubly Linked Lists

- Delete a node pointed to by target:

```
node*  delete( node *header, node *target)          {
     if ( target != header )
               target→prev→next = target→next;
     else
     {
               header = target→next;
               header→prev = NULL;                  return header;
     }
     if ( target→next != NULL )
               target→next→prev = target→prev;
     free ( target );      return header;  }
```

Deleting the first node

Deleting the last node

Doubly Linked Lists

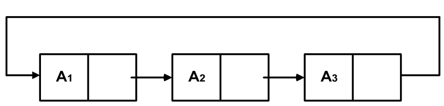
- Reverse a doubly linked list:

```
void reverse ( node *header)        {
    node *last = header, *start = header;
    while ( last→next != NULL)      last = last→next;
    while ( start != last )
         {
                  swap ( start→val , last→val);
                  start = start→next;
                  if ( start == last)   break;
                  last = last→prev;
         }
}
```

Move to the last node

Check for even number of nodes

Circular Linked Lists



- Last node points to the first node
- Traversing a circular linked list
  - Different than singly linked list
    - NULL pointer is missing
    - Save the starting pointer and traverse until the next field of a node becomes equal to the start node

Circular Linked Lists

- Identify a circular list by the pointer to the last node
  - Insertion at the **start or end** of a list takes O(1) time
  - Concatenating two lists also takes O(1) time

Josephus Problem

- ***"n"*** children arranged a in a circle
  - Children are numbered in clockwise fashion
- Choose a lucky number "m"
- Start counting from child 1 in clockwise fashion
  - The ***m^{th}*** child is eliminated
- Start the next round from the *child next to the eliminated child*
  - Continue until you are left with one child who is the **winner**

Josephus Problem

- Define a child:

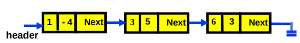
```
typedef struct child
{
        int position;
        struct child *nextchild;
} child;
```

Josephus Problem

```
int findwinner ( int n, int m)
{
    create a circular linked list with n children;
    while ( the list contain more than one child)
    {
        set a counter to zero;
        go to the next child and increment the counter as long as it is less than m;
        delete the current child;
    }
    get the position of the only child in the list;
    return the position;
}
```


Polynomials

- The polynomial $P(x) = 3x^6 + 5x^3 - 4x$ can be represented as:

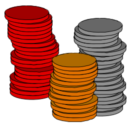



```
typedef struct poly
{
    int expo, coeff;
    struct poly * next;
}
```


Stacks

What is a Stack

- Special form of linear list
- Principle: Last In First Out
  - the last element inserted is the first one to be removed
- Like a plate stacker

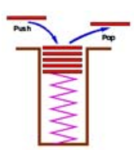


Stack Applications

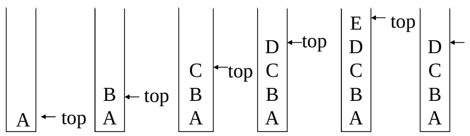
- Real life
  - Pile of books
  - Plate trays
- More applications related to computer science
  - Recursive function calls
  - Evaluating expressions

Stack Operations

- construct a stack (usually empty)
- check if it is empty
- Push: add an element to the top
- Top: retrieve the top element
- Pop: remove the top element



Last In First Out



Array Implementation of Stack

- Allocate an array of some size (pre-defined)
  - MAX_STACK_SIZE elements in stack
  - Bottom stack element stored at position 0
  - Last element in the stack is the *top*
- *Define the stack:*

```
typedef struct stack
{
        int a [MAX_STACK_SIZE];
        int top;
} stack;
```


Array Implementation of Stack

- Push Operation

```
void push ( stack *s, int item)
{
    if ( s→top == MAX_STACK_SIZE - 1 )   return;      Stack is full
    else
        {
            s→top = s→top + 1;      Increment the top
            s→a [s→top] = item;     Insert the element
        }
}
```

Array Implementation of Stack

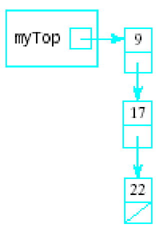
- Pop Operation

```
int pop ( stack *s, int *x )
{
        if ( s→top == -1)   return 0;        Stack is empty
        else
                {
                        *x = s→a[s→top];     Remove topmost element
                        s→top = s→top - 1;   Decrement the top
                        return 1;
                }
}
```


Linked List Implementation

- Define the nodes:

```
typedef struct node
{
        int item;
        struct node *next;
} node;
```



- Define the top:

```
typedef struct stack
{
        node *mytop;
} stack;
```


Linked List Implementation

- Push Operation

```
void push ( stack *s, int data )
{
     node *newnode = ( node *) malloc ( sizeof ( node ) );
     newnode→item = data;
     newnode→next = s→mytop;      New node inserted
     s→mytop = newnode;           New node becomes the top
}
```

Linked List Implementation

- Pop Operation

```
int pop ( stack *s, int *x )
{
    node *temp;
    if ( s→mytop == NULL )        Empty stack
    {            x = NULL;        return 0;        }

    *x = s→mytop→item;        Remove item from top
    temp = s→mytop;
    s→mytop = s→mytop→next;        Advance top to the next node
    free ( temp );     return 1;
}
```


Application of Stacks

Function Calls

Consider events when a function begins execution

- *Stack frame* is created
- Copy of stack frame pushed onto run-time stack
- Arguments copied into parameter spaces
- Control transferred to starting address of body of function

Function Calls

When function terminates

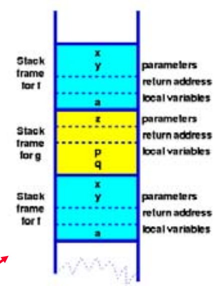
- Run-time stack popped
  - Removes stack frame of terminated function
  - exposes stack frame of previously executing function
- Stack frame used to restore environment of interrupted function
- Interrupted function resumes execution

Function Calls

```
function f( int x, int y) {
   int a;
   if ( term_cond ) return …;
   a = ….;
   return g( a );
   }

function g( int z ) {
   int p, q;
   p = …. ; q = …. ;
   return f(p,q);
   }
```

Context for execution of f



Evaluation of Arithmetic Expression

INFIX	POSTFIX	PREFIX
A + B	A B +	+ A B
A * B + C	A B * C +	+ * A B C
A * (B + C)	A B C + *	* A + B C
A - (B - (C - D))	A B C D- - -	-A-B-C D
A - B - C - D	A B-C-D-	- - -A B C D

- Most compilers convert an expression in ***infix*** notation to ***postfix*** notation
- Advantage:
  ✓ expressions can be written without parentheses

Evaluation of Arithmetic Expression

- Evaluating Postfix Expression
  - *"By hand" (Underlining technique)*:
    - Scan the expression from left to right to find an operator.
    - Locate ("underline") the last two preceding operands and combine them using this operator.
    - Repeat until the end of the expression is reached.

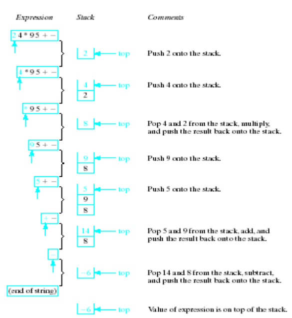
```
→2 3 4 + 5 6 - - *
→2 3 4 + 5 6 - - *
→2 7 5 6 - - *
→2 7 5 6 - - *
→2 7 -1 - *
→2 7 -1 - *
→2 8 *
→2 8 *
→16
```


Evaluation of Arithmetic Expression

- Evaluating Postfix Expression
  1. Initialize an empty stack
  2. Repeat the following until the end of the expression is encountered
     1. Get the next element from the expression
     2. Operand – push onto stack
        Operator – do the following
        1. **Pop 2 values from stack**
        2. **Apply operator to the two values**
        3. **Push resulting value back onto stack**
  3. When end of expression encountered, value of expression is the (only) number left in stack

Evaluation of Arithmetic Expression

- Evaluating Postfix Expression



Evaluation of Arithmetic Expression

- Infix to Postfix Conversion
  - *By hand:* "Fully parenthesize-move-erase" method:
    - Fully parenthesize the expression.
    - Replace each right parenthesis by the corresponding operator.
    - Erase all left parentheses.

```
A * B + C  → ((A * B) + C)      A * (B + C)  → (A * (B + C) )
           → ((A B * C +                     → (A (B C + *
           → A B * C +                       → A B C + *
```

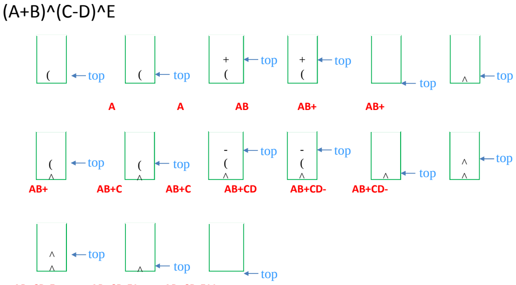

Evaluation of Arithmetic Expression

1. Initialize an empty stack of operators
2. While !end of expression
   a) Get next input "token" from infix expression
   b) If token is ...
      i. operand display it
      ii. operator
          if operator has higher priority than top of stack
          push token onto stack
          else
          pop and display top of stack
          repeat comparison of token with top of stack
      iii. "(" : push onto stack
      iv. ")" : pop and display stack elements until "(" occurs, do not display it
3. When end of infix reached, pop and display stack items until empty

Operator	In-Stack Priority	Input Priority
+, -	1	1
*, /	2	2
^	3	4
(	0	4

Evaluation of Arithmetic Expression

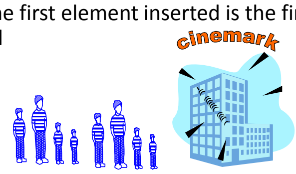
- (A+B)^(C-D)^E



Queues

What is a queue

- Stores a set of elements in a particular order
- Stack principle: FIRST IN FIRST OUT
- = FIFO
- It means: the first element inserted is the first one to be removed
- Example

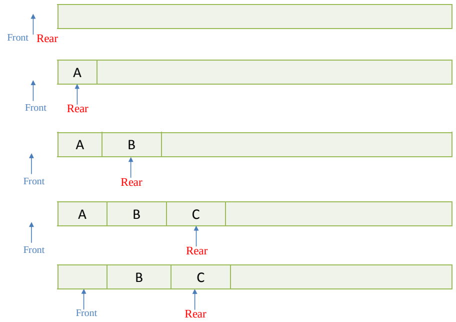


- The first one in line is the first one to be served

Queue Applications

- Real life examples
  - Waiting in line
  - Waiting on hold for tech support
- Applications related to Computer Science
  - Threads
  - Job scheduling (e.g. Round-Robin algorithm for CPU allocation)

First In First Out



Job Scheduling

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					queue is empty
-1	0	J1				Job 1 is added
-1	1	J1	J2			Job 2 is added
-1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

Queue Operations

- Create a queue
- Check if a queue is full or not
- Check if a queue is empty or not
- Add an element to a queue (enqueue)
- Remove an element from queue (dequeue)

Array Implementation of Queue

- As with the array-based stack implementation, the array is of fixed size
  - A queue of maximum N elements
- Slightly more complicated
  - Need to maintain track of both front and rear
- Define the queue:

```
typedef struct queue
{
        int a[MAX_SIZE];
        int rear, front;
} queue;
```


Array Implementation of Queue

- Enqueue Operation

```
int enqueue (queue *q, int data)
{
        if (q→rear == MAX_SIZE – 1)        Queue is full
                return 0;
        else
        {
                q→rear == q→rear + 1;      Increment the rear
                q→a[q→rear] = data;        Insert the element
                return 1;
        }
}
```

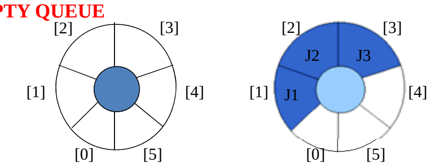

Array Implementation of Queue

- Dequeue Operation

```
int dequeue (queue *q, int *data)
{
        if (q→rear == q→front)
        {
                q→front = -1;        Queue is empty
                q→rear = -1;
                *data = NULL;
                return 0;
        }
        else
        {
                q→front = q→front + 1;      Increment the front
                *data = q→a[q→front];       Remove the element
                return 1;
        }
}
```

Circular Queue

EMPTY QUEUE



front = 0
rear = 0

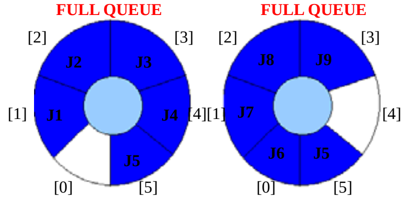
front = 0
rear = 3

Can be seen as a circular queue

Circular Queue

Leave one empty space when queue is full. Why?

FULL QUEUE **FULL QUEUE**



front =0
rear = 5

front =4
rear =3

How to test when queue is empty?
How to test when queue is full?

Circular Queue

- Enqueue Operation

```
void enqueue (queue *q, int data)
{
        q→rear = (q→rear +1) % MAX_SIZE;
        if (q→front == q→rear)
                return;
        q→a[q→rear] = item;
}
```

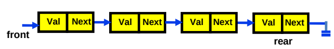
Circular Queue

- Dequeue Operation

```
void dequeue (queue *q, int *data)
{
        if (q→front  ==  q→rear)
                return;
        q→front  =  (q→front+1) % MAX_SIZE;
        *data  =  q→a[q→front];
}
```


Linked List Implementation

- Define the nodes




```
typedef struct node
{
        int item;
        struct node *next;
} node;
```

- Define the rear and front

```
typedef struct queue
{
        node *rear;
        node *front;
}
```


Linked List Implementation

- Enqueue Operation

```
void enqueue (queue *q, int data)
{
        node *temp = (node *) malloc (size of (queue));
        temp→item = data;                  New node created
        temp→next = NULL;
        if (q→front == NULL)               Queue is empty
        {
                q→front = temp;
                q→rear = temp;
                return;
        }
        q→rear→next = temp;                Insert at the rear
        q→rear = temp;                     New node becomes rear
}
```

Linked List Implementation

- Dequeue Operation

```
int dequeue (queue *q, int *data)
{
        node *temp;
        if (q→front == NULL)
        {
                data = NULL;
                return 0;
        }
        temp = q→front;
        *data = temp→item;
        q→front = q→front→next;
        if (q→rear == temp)
                q→rear = NULL;
        free (temp);
        return 1;
}
```

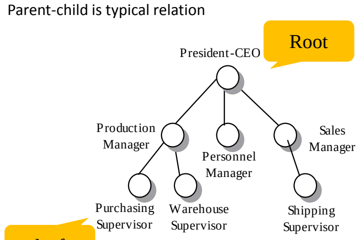
Queue is empty

Only node in the Queue

Trees

Introduction

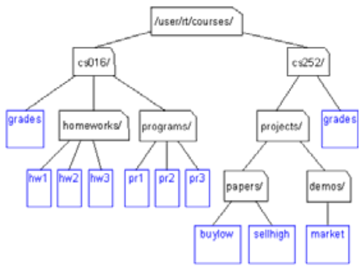
- Non-linear data structure
- Depicts **hierarchical** relationship between the nodes
  - Parent-child is typical relation



HIERARCHICAL TREE STRUCTURE

Another Example

- Unix / Windows file structure

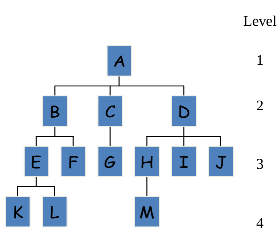


Definition

- A tree T is a finite set of one or more nodes such that:
  - There is a specially designated node called the **root**.
  - The remaining nodes are partitioned into n>=0 disjoint sets $T_1, \ldots, T_n$, where each of these sets is a tree.
  - We call $T_1, \ldots, T_n$ the **subtrees** of the root.

Terminology

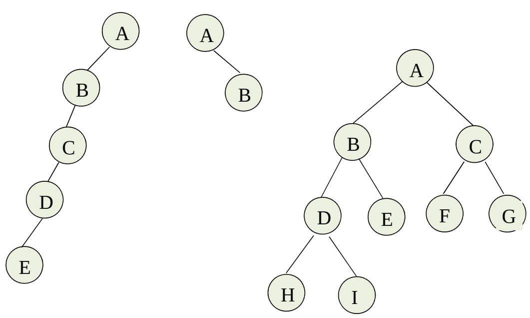
- **Degree** of a node: number of nodes connected of that node
  - The node with degree 1 is a leaf or terminal node.
- Children of the same parent are ***siblings***.
- **Ancestors** of a node: all the nodes along the path from the root to the node.
- **Level**: level of a node is one more than its parent. Root node has a level 1.
- **Height** of a tree: maximum level of any node



Binary Tree

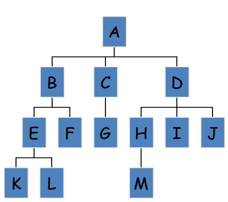
- A special class of trees: max number of child for each node is **2**.
- Recursive definition: A binary tree is a finite set of nodes that is either empty or consists of a root and two **disjoint binary trees** called *the left subtree* and *the right subtree*.
- Any tree can be transformed into binary tree.
  - by left child-right sibling representation
- Total number of binary trees possible with n nodes is $^{2n}C_n - {}^{2n}C_{n-1}$

Examples

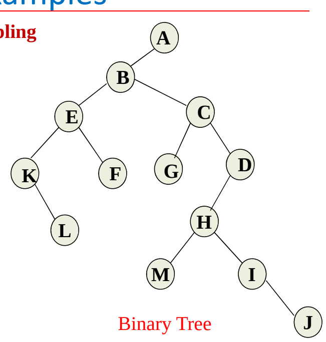


Examples

Left Child - Right Sibling



General Tree



Binary Tree

Properties

- The maximum number of nodes on level i of a binary tree is 2^{i-1}, i>=1.
- The maximum number of nodes in a binary tree of depth **h** is 2^h-1, h>=1.

Prove by induction

$$\sum_{i=1}^{h} 2^{i-1} = 2^h - 1$$

In other words
h=log(n+1)

Properties

For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 the number of nodes with 2 children, then $n_0=n_2+1$

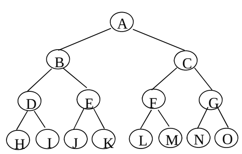
proof:

Let n and B are the total number of nodes & branches in T.
Let n_0, n_1, n_2 represent the nodes with no children, single child, and two children respectively.

$n= n_0+n_1+n_2$, $B+1=n$, $B=n_1+2n_2$ ==>
$n_1+2n_2+1= n$,
$n_1+2n_2+1= n_0+n_1+n_2$ ==> $n_0=n_2+1$

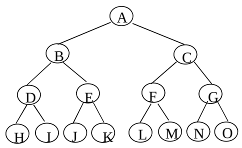
Full Binary Tree

- If all non-leaf nodes of a binary tress have **exactly two non-empty children** and **all leaf nodes are at the same level**.
- A full binary tree of depth *h* is a binary tree of depth *h* having 2^h **-1** nodes, *h*>=1.

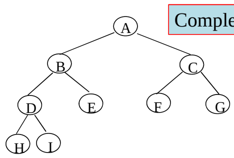


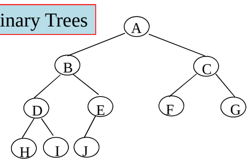
Complete Binary Tree

- A full binary tree or Full up to level h-1 and if any node at **level h-1 has one child**, that must be a **left child**.



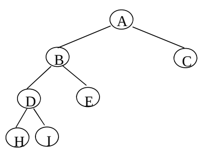
Complete Binary Trees

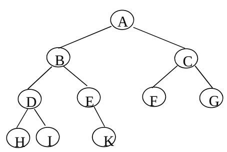




Complete Binary Tree

- A full binary tree or Full up to level h-1 and if any node at **level h-1 has one child**, that must be a **left child**.

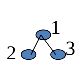


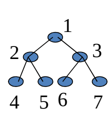


Non-Complete Binary Trees

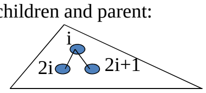
Binary Tree Representation

- If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1<=i<=n$, we have:
  - *parent*(i) is at $i/2$ if $i!=1$. If $i=1$, i is at the root and has no parent.
  - *leftChild*(i) is at $2i$ if $2i<=n$. If $2i>n$, then i has no left child.
  - *rightChild*(i) is at $2i+1$ if $2i+1<=n$. If $2i+1>n$, then i has no right child.

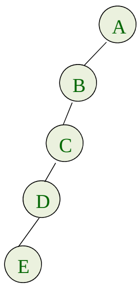




Relationships between labels of children and parent:

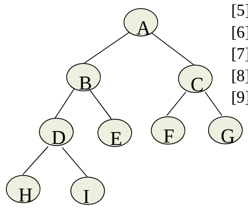


Binary Tree Representation



[1]	A
[2]	B
[3]	--
[4]	C
[5]	--
[6]	--
[7]	--
[8]	D
[9]	--
.	.
[16]	E

Wastage of space

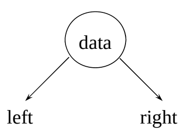


[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Linked Representation

```
typedef struct btnode
    {
        int data;
        btnode *lchild, *rchild;
    }btnode;
```

left	data	right



Creation of Binary Tree

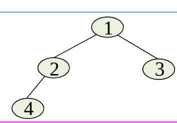
```
main ()
{
    btnode *Root=NULL;
    btnode *p, *q;

    p=(btnode*)malloc(sizeof(btnode));
    p→data=1;
    p→lchild=p→rchild=NULL;
    Root=p;         q=p;

    p=(btnode*)malloc(sizeof(btnode));
    p→data=2;
    p→lchild=p→rchild=NULL;
    q→lchild=p;         q=p;
```

```
p=(btnode*)malloc(sizeof(btnode));
p→data=4;
p→lchild=p→rchild=NULL;
q→lchild=p;
q=p;

q=Root;
p=(btnode*)malloc(sizeof(btnode));
p→data=3;
p→lchild=p→rchild=NULL;
q→rchild=p;
}
```



Binary Tree Traversal

- Traversal is the process of visiting every node once
- Let l, R, and r denotes moving left, visiting the node, and moving right.
- Six possible combinations of traversal
  - lRr, lrR, Rlr, Rrl, rRl, rlR
- Adopt convention that we traverse left before right, only 3 traversals remain
  - lRr, lrR, Rlr
  - inorder, postorder, preorder

Binary Tree Traversal

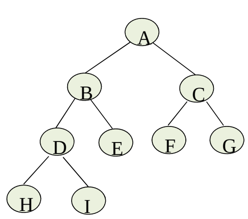
Inorder Traversal
1. Traverse left subtree
2. Visit the root
3. Traverse right subtree

Postorder Traversal
1. Traverse left subtree
2. Traverse right subtree
3. Visit the root

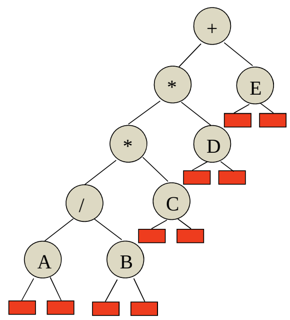
Preorder Traversal
1. Visit the root
2. Traverse left subtree
3. Traverse right subtree

Binary Tree Traversal

- Inorder:
  H D I B E A F C G
- Postorder:
  H I D E B F G C A
- Preorder:
  A B D H I E C F G



Binary Tree Traversal



Inorder
A / B * C * D + E
infix expression

postorder
A B / C * D * E +
postfix expression

preorder
+ * * / A B C D E
prefix expression

Binary Tree Traversal

```
void inorder(btnode *Root)
{
    if (Root) {
        inorder(Root->lchild);
        printf("%d",Root->data);
        inorder(Root->rchild);
    }
}
```


Binary Tree Traversal

```
void postorder(btnode *Root)
{
    if (Root) {
        postorder(Root->lchild);
        postorder(Root->rchild);
        printf("%d",Root->data);
    }
}
```

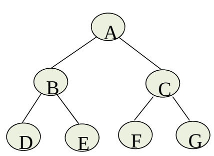
Binary Tree Traversal

```
void preorder(btnode *Root)
{
    if (Root) {
        printf("%d",Root->data);
        preorder(Root->lchild);
        preorder(Root->rchild);
    }
}
```


Reconstruction of Binary Tree

- It is impossible to reconstruct binary tree from inorder or preorder or postorder traversals alone.
- However, if inorder and preorder traversals are given, a unique binary tree can be reconstructed

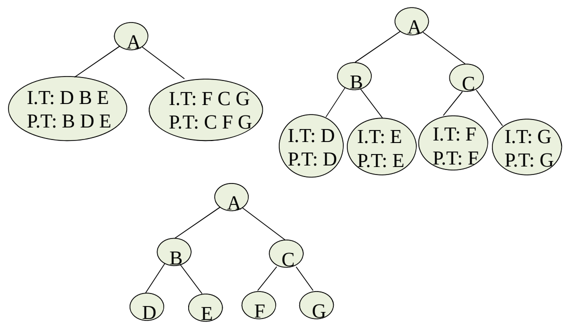
Reconstruction of Binary Tree



Inorder: D B E A F C G

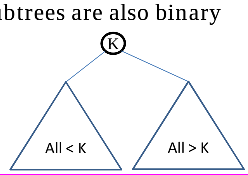
Preorder: A B D E C F G

Reconstruction of Binary Tree



Binary Search Trees

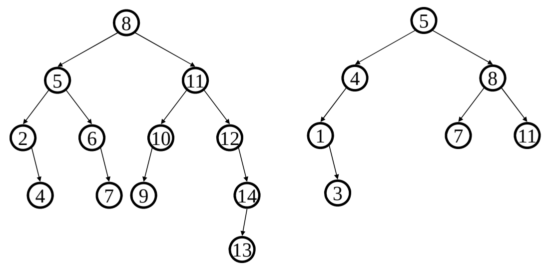
- Definition
  - The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
  - The left and right subtrees are also binary search trees.



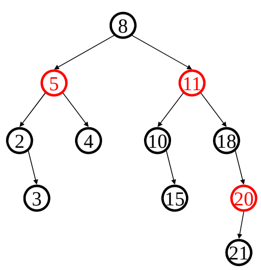
Binary Search Trees

- Binary Search Trees (BST) are a type of Binary Trees with a special organization of data.
- Leads to O(log n) complexity for searches, insertions and deletions in certain types of BST (balanced trees).
  - O(h) in general

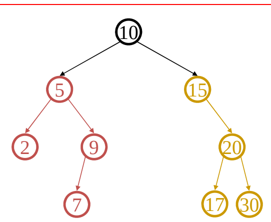
Examples and Counter Examples



Examples and Counter Examples



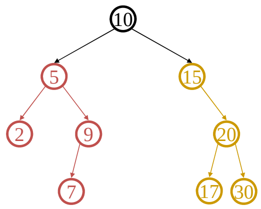
Inorder Traversal



2→5→7→9→10→15→17→20→30

What does this guarantee with a BST?

rRl Traversal



30 →20 →17 →15 →10→9→7→5→2

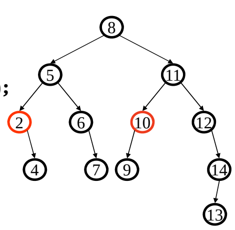
What does this guarantee with a BST?

BST Representation

```
typedef struct bstnode
{
    int data;
    bstnode *lchild, *rchild;
}bstnode;
```


Searching in a BST

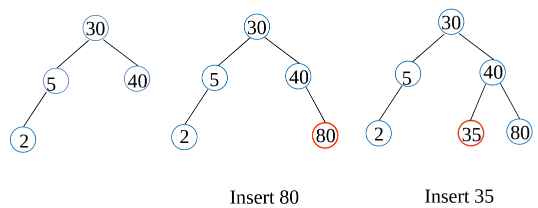
```
bstnode *search(int key, bstnode * root)
{
 if (root == NULL) return root;
 else if (key < root→data)
  return search(key, root→lchild);
 else if (key > root→data)
  return find(key, root→rchild);
 else
  return root;
}
```



Insertion into a BST

- based on comparisons of the new item and values of nodes in the BST
- **starting at the root** probe **down** the tree till you find a node whose left or right pointer is empty and is a logical place for the new value
- In other words, all inserts take place at a leaf or at a leaflike node – a node that has only one null subtree.

Insertion into a BST



Insertion into a BST

```
void insert(bstnode * newnode, bstnode * root)
{
    if (root→data > newnode→data)
    {
        if (root→lchild == NULL)
            root→lchild=newnode;
        else
            insert( newnode, root→lchild );
    }
    else
    {
        if (root→rchild == NULL)
            root→rchild=newnode;
        else
            insert( newnode, root→rchild );
    }
}
```

Insertion into a BST

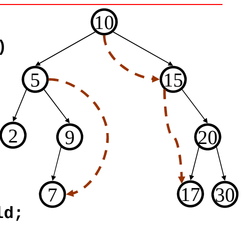
- The order of supplying the data determines where it is placed in the BST , which determines the shape of the BST
- Create BSTs from the same set of data presented each time in a different order:

a) 17 4 14 19 15 7 9 3 16 10

b) 9 10 17 4 3 7 14 16 15 19

c) 19 17 16 15 14 10 9 7 4 3 **can you guess this shape?**

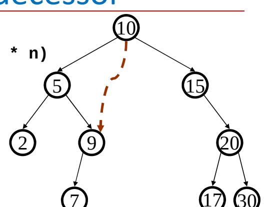
Inorder Successor




```
bstnode * successor(bstnode * n)
{
  bstnode *iosuccessor;
      if (n→rchild == NULL)
          iosuccessor=NULL;
      else
      {
             iosuccessor=n→rchild;
             while (iosuccessor→lchild != NULL)
                    iosuccessor=iosuccessor→lchild;
      }
return iosuccessor;
}
```

How many children can the inorder successor of a node have?

Inorder Predecessor




```
bstnode * predecessor(bstnode * n)
{
  bstnode *iopredecessor;
      if (n→lchild == NULL)
          iopredecessor=NULL;
      else
      {
             iopredecessor=n→lchild;
             while (iopredecessor→rchild != NULL)
             iopredecessor=iopredecessor→rchild;
      }
return iopredecessor;
}
```

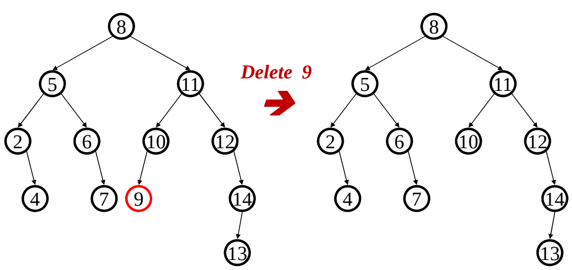
How many children can the inorder predecessor of a node have?

Delete a Node from a BST

- Following are the possible cases when we delete a node:
  - The node to be deleted has no children.
    - Set the respective pointer of its parent to NULL.
  - The node to be deleted has only a right subtree.
    - Attach respective pointer of node's parent to right subtree.
  - The node to be deleted has only a left subtree.
    - Attach respective pointer of node's parent to left subtree.
  - The node to be deleted has two subtrees.
    - Replace node's data with data in **inorder successor** (predecessor) and delete the **inorder successor** (predecessor).

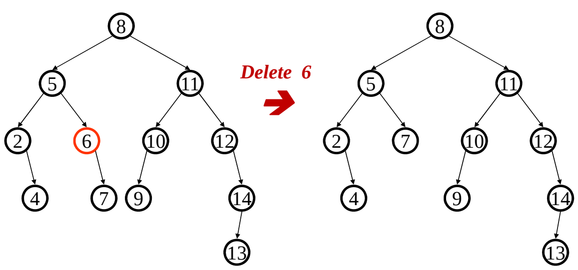
Delete a Node from a BST

Case 1: deleting a node with 2 EMPTY SUBTREES



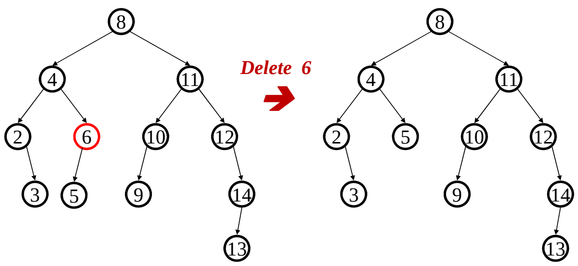
Delete a Node from a BST

Case 2: deleting a node with only RIGHT SUBTREES



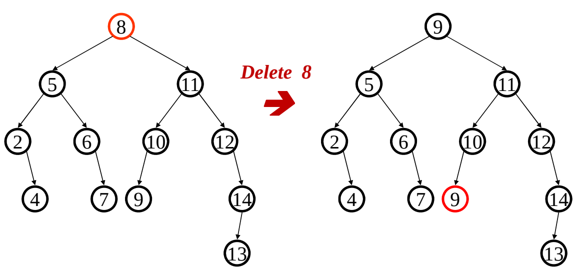
Delete a Node from a BST

<u>Case 3:</u> deleting a node with only LEFT SUBTREES



Delete a Node from a BST

<u>Case 4:</u> deleting a node with 2 NON-EMPTY SUBTREES



Delete a Node from a BST

```
bstnode * delete (bstnode * root, bstnode * n, bstnode * parent)
{
    bstnode *iosuccessor, *retval;
    if (n→lchild != NULL && n→rchild != NULL)
    {
        successor (n, &parent, &iosuccessor);
        n→data = iosuccessor→data;
        n=iosuccessor;
    }
    if (n→lchild == NULL)
    {
        if (parent != NULL)
            {
```

This function takes node to be deleted, its parent as argument. When the function terminates iosuccessor contains the inorder successor of n, parent becomes the parent of iosuccessor

Now delete the inorder successor

n does not have left subtree

Check for deleting root

Delete a Node from a BST

```
            if (parent→lchild == n)
                    parent→lchild=n→rchild;
            else
                    parent→rchild=n→rchild;
            retval=root;
        }
        else
            retval=n→rchild;
    }
    if (n→rchild == NULL)
        {
            if (parent != NULL)
```

Check to see whether n belongs to the left subtree of parent

Deleting root

Delete a Node from a BST

```
        {
            if (parent→lchild == n)
                    parent→lchild=n→lchild;
            else
                    parent→rchild=n→lchild;
            retval=root;
        }
        else
            retval=n→lchild;
    }
return retval;
}
```

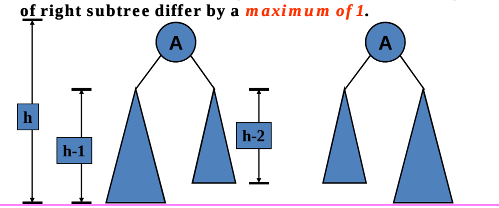
Deleting root

Analysis of BST Operations

- The complexity of operations **search**, **insert** and **delete** in BST is **O(h)** , where **h= O(log n)**, the height of BST.
  - **But**, the BST can take a linear shape and the operations will become **O (n)**

Height Balanced or AVL Trees

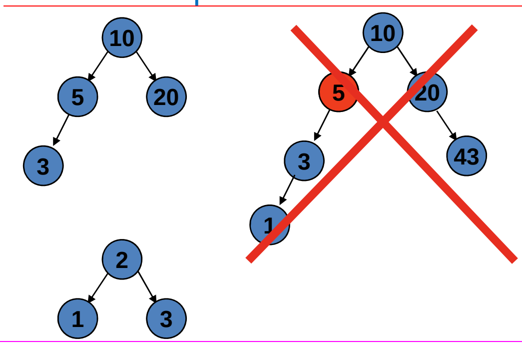
- **Unbalanced Binary Search Trees are bad. Worst case: operations take O(n).**
- **Height Balanced or AVL (Adelson-Velskii & Landi) trees maintain balance.**
  - **For each node in BST, height of left subtree and height of right subtree differ by a *maximum of 1*.**



Height Balanced or AVL Trees

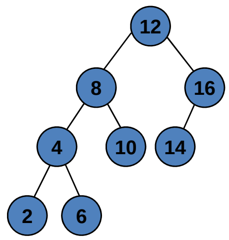
```
typedef struct avlnode
{
    int data;
    int balancefactor;
    bstnode *lchild, *rchild;
}avlnode;
```


Examples of AVL trees



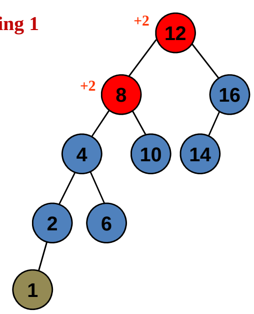
Insertion into an AVL Tree

Insert 1 in the following AVL tree



Insertion into an AVL Tree

After inserting 1

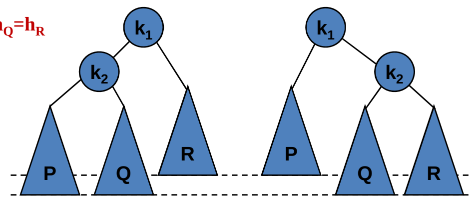


Insertion into an AVL Tree

- Insertion of a node into AVL tree may result in ***imbalance***.
- To ensure balance condition, after insertion of a new node, back up the path from the inserted node to root and calculate **balance factor** for each node.
  - **If the balance condition does not hold in a certain node, we do one of the following rotations:**
    - **Single rotation**
    - **Double rotation**

Insertion into an AVL Tree

$h_P=h_Q=h_R$



Possible cases of insertions that may result imbalance

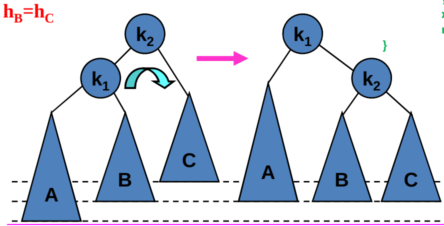
- An insertion into the subtree:
  - Case 1: insert into P **(outside)**
  - Case 2: insert into Q **(inside)**
- An insertion into the subtree:
  - Case 3: insert into Q **(inside)**
  - Case 4: insert into R **(outside)**

Insertion into an AVL Tree

Case 1: Single Rotation

```
avlnode* rotateright (avlnode *y)
{
        avlnode *x;
        x=y→lchild;
        y→lchild=x→rchild;
        x→rchild=y;
        return x;
}
```

$h_A=h_B+1$
$h_B=h_C$

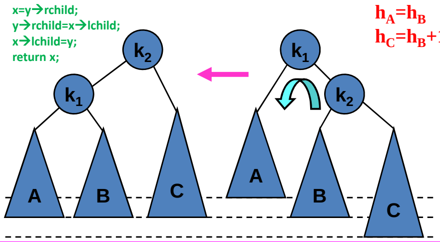


Insertion into an AVL Tree

Case 4: Single Rotation

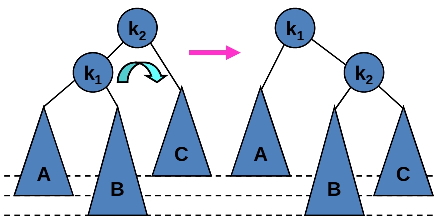
```
avlnode* rotateleft (avlnode *y)
{
        avlnode *x;
        x=y→rchild;
        y→rchild=x→lchild;
        x→lchild=y;
        return x;
}
```

$h_A=h_B$
$h_C=h_B+1$



Insertion into an AVL Tree

Case 2 & 3 (inside case): Single Rotation does not work

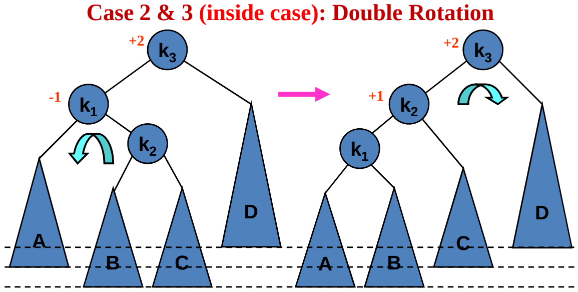


$h_B = h_A + 1$
$h_A = h_C$

Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Insertion into an AVL Tree

Case 2 & 3 (inside case): Double Rotation

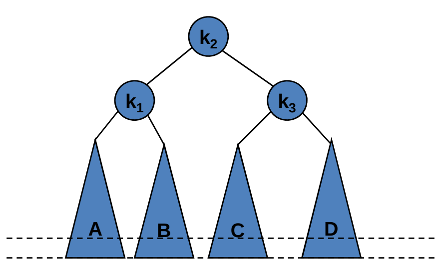


$h_A = h_B = h_C = h_D$

Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Insertion into an AVL Tree

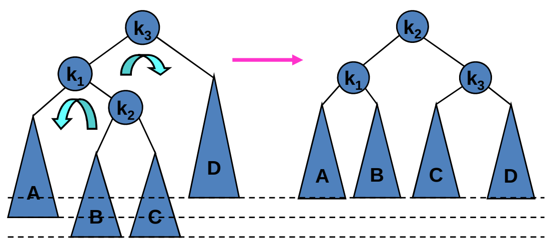
Case 2 & 3 (inside case): Double Rotation



Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Insertion into an AVL Tree

Double Rotation

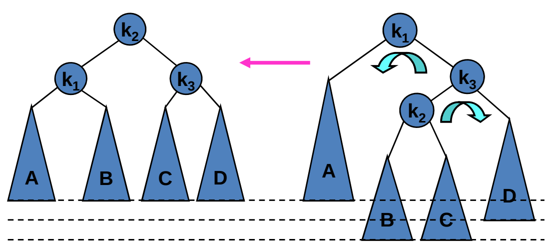


$h_A=h_B=h_C=h_D$

Bhaskar Sardar, Information Technology Department, Jadavpur University, India

Insertion into an AVL Tree

Double Rotation

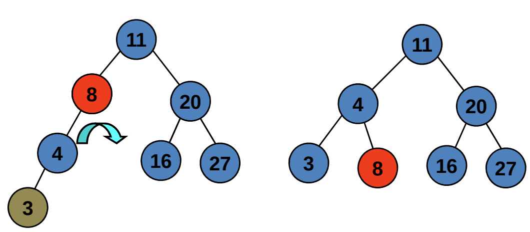


$h_A=h_B=h_C=h_D$

Bhaskar Sardar, Information Technology Department, Jadavpur University, India

Insertion into an AVL Tree

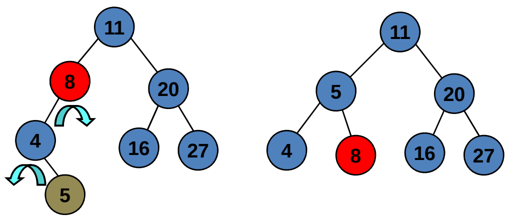
Insert 3 into the AVL tree



Bhaskar Sardar, Information Technology Department, Jadavpur University, India

Insertion into an AVL Tree

Insert 5 into the AVL tree



Delete a Node from AVL Tree

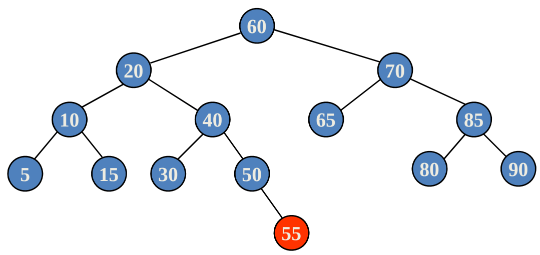
- Deleting a node from an AVL Tree is similar to that of deleting a node from a binary search tree. However, it may **unbalance** the tree.
- Starting from the deleted node, check all the nodes in the path up to the root for the first unbalance node.
  - Use appropriate **single or double rotation**.
  - May need to continue searching for unbalanced nodes all the way to the root.

Delete a Node from AVL Tree

- Deletion:
  - Case 1: if X is a **leaf**, delete X
  - Case 2: if X has **1 child**, use it to replace X
  - Case 3: if X has **2 children**, replace X with its **inorder predecessor** (and recursively delete it)
- Rebalancing

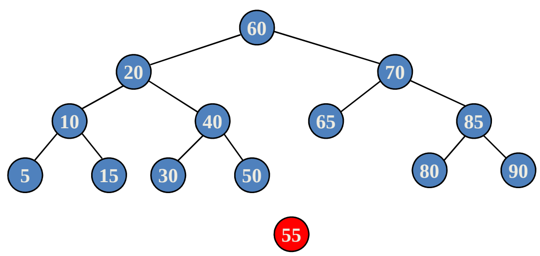
Delete a Node from AVL Tree

Delete 55



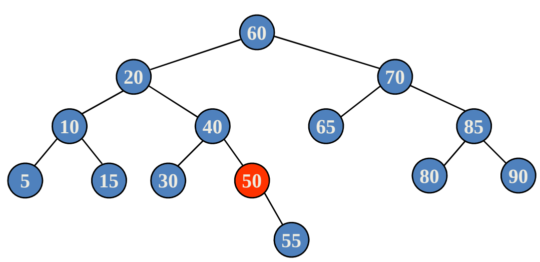
Delete a Node from AVL Tree

Delete 55



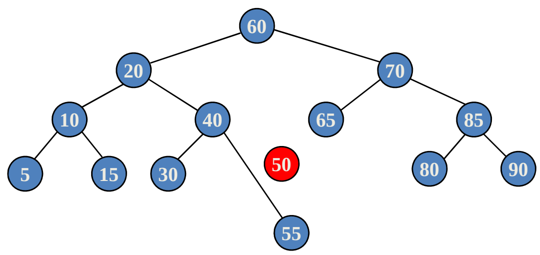
Delete a Node from AVL Tree

Delete 50



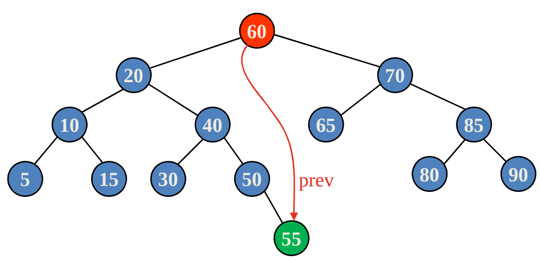
Delete a Node from AVL Tree

Delete 50



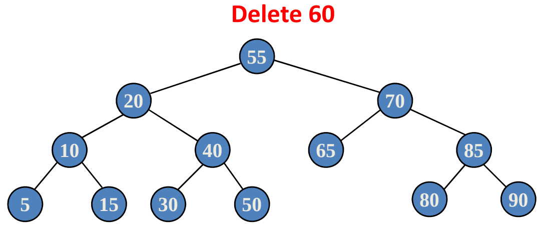
Delete a Node from AVL Tree

Delete 60



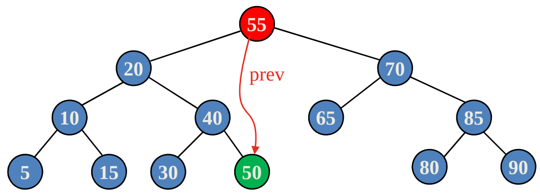
Delete a Node from AVL Tree

Delete 60



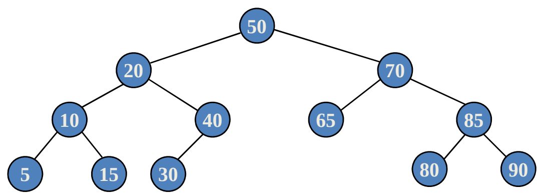
Delete a Node from AVL Tree

Delete 55



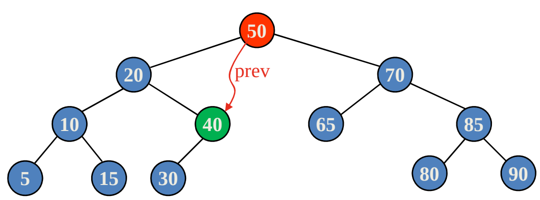
Delete a Node from AVL Tree

Delete 55



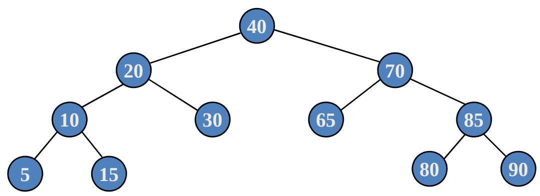
Delete a Node from AVL Tree

Delete 50



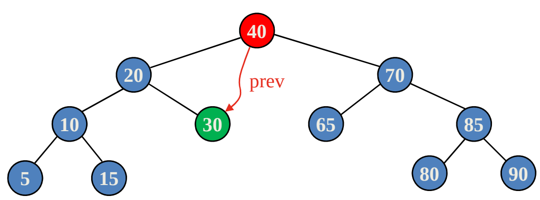
Delete a Node from AVL Tree

Delete 50



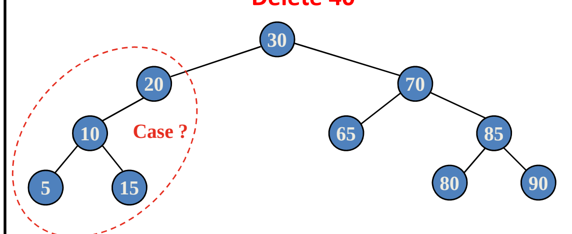
Delete a Node from AVL Tree

Delete 40



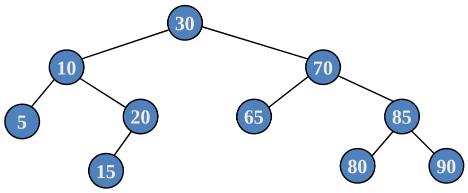
Delete a Node from AVL Tree

Delete 40



Delete a Node from AVL Tree

After Rebalancing



Any Doubt ?

- Please feel free to write to me:

bhaskargit@yahoo.co.in