

DATA STRUCTURES

UNIT-1

ALGORITHM

An algorithm is a step by step representation or a procedure for solving a problem.

or

It is a method of finding a right solution to a problem or to a different problem or to a different problem breaking into simple cases.

PROPERTIES OF AN ALGORITHM

Finiteness:

An algorithm should terminate at finite number of steps.

Definiteness:

Each step of an algorithm must be precisely stated.

Effectiveness:

It consists of basic instructions that are realizable.

This means that the instructions can be performed by using the given inputs in a finite amount of time.

Input:

An algorithm accepts zero or more inputs.

Output:

It produces at least one output.

PSEUDOCODE

It is a representation of algorithm in which instruction sequence can be given with the help of programming constructs.

or

Pseudo code, on the other hand, is not a programming language, but simply an informal way of describing a program.

Because it is not an actual programming language, pseudo code cannot be compiled into an executable program.

Therefore, pseudo code must be converted into a specific programming language if it is to become an usable application.

PSEUDOCODE CONVENTIONS

1. Algorithm is a procedure consisting of heading and body. The heading consists of a name of the procedure and parameter list. The syntax is

Algorithm name_of
_procedure(paramater1,parameter2,.....parameter n).

- 2.Using assignment operator:=an assignment statement can be given. For instance: variable:=expression;
3. Boolean operators, logical operators, relational operators can be used in pseudo code.
4. All different types of arrays can be used and array indices stored in [and] brackets.

PSEUDOCODE CONVENTIONS

5. The beginning and end of block should be indicated by { and } resp. the compound statements should be enclosed within { and } brackets.
6. The delimiters ; are used at the end of each statement.
7. Single line comments are written using // as beginning of comment.
8. The identifier should begin by letter only.
9. No need to write data types explicitly for identifiers.

PSEUDOCODE CONVENTIONS

10. The inputting and outputting can be done using read and write.
11. The conditional statements and the looping statements have the same syntax as in C language.

EXAMPLES

1) write an algorithm to count the sum of n numbers

Algorithm sum(1,n)

{

Result:=0;

for i:=1 to n do i:=i+1

Result:=result+i;

}

EXAMPLES

2) write an algorithm to check whether given number is even or odd

Algorithm events (val)

```
{  
    if (val%2==0) then  
        write("given no is even");  
    else  
        write("given no is odd");  
}
```

EXAMPLES

3) write an algorithm to find factorial of n number.

Algorithm fact(n)

```
{  
    if n:=1 then  
        return 1;  
    else  
        return n*fact(n-1);  
}
```

EXAMPLES

4) write an algorithm to perform multiplication of two matrices.

Algorithm Mul(A,B,n)

{

 for i:=1 to n do

 for j:=1 to n do

 c[i,j]:=0

 for k:=1 to n do

 c[i,j]:=c[i,j]+a[i,k]*b[k,j];

}

RECURSIVE ALGORITHM

A recursive routine is one whose design includes a call to itself.

Or

A function that calls itself is known as recursive function and this technique is known as recursion in C programming.

EXAMPLES

Factorial of a number

Algorithm factorial(a)

```
int a;
```

```
{
```

```
    int fact=1
```

```
        if(a>1)
```

```
            Fact = a* factorial(a-1);
```

```
            Return(fact);
```

```
}
```

DATA ABSTRACTION

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

PERFORMANCE EVALUATION

The total effectiveness of a computer system, including throughput , individual response time , and availability.

Performance evaluation can be loosely divided into 2 phases

- 1) A priori estimates which is known as performance analysis
- 1) A posterior testing which is known as performance measurement.

PERFORMANCE ANALYSIS

The efficiency of an algorithm can be decided by measuring the performance of an algorithm.

The performance of an algorithm by computing amount of time(time complexity)and storage requirement(space complexity).

TIME COMPLEXITY

This is the amount of computing time required by an algorithm to run to completion.

There are 2 types of computing time- compile time and run time.

The time complexity is generally computed using run time or execution time.

The time complexity is given in terms of frequency count.

Frequency count is basically a count denoting number of times of execution of statement.

TIME COMPLEXITY

The time complexity is computed using run time is calculated by input size and asymptotic notations .

Input size: the input size of any instance of a problem is defined as the number of words required to describe that instance of problem.

Asymptotic notations: This is the shorthand way to represent the time complexity.

Time complexity is given as fastest possible, slowest possible or average time.

Notations such as Ω , θ , O are the asymptotic notations.

BIG OH NOTATION

It represents the upper bound of algorithms running time.

The longest amount of time taken by the algorithm to complete is calculated by big oh (O).

Def: let $f(n)$ and $g(n)$ are two non-negative functions. And is there exists an integer n_0 and constant C such that $C > 0$ and for all integers $n > n_0$, $f(n) \leq C * g(n)$, then $f(n)$ is big oh of $g(n)$. It is also denoted as “ $f(n) = O(g(n))$ ”.

BIG OH NOTATION

Various meanings associated with big-oh are

$O(1)$ - constant computing time

$O(n)$ - linear

$O(n^2)$ -quadratic

$O(n^3)$ -cubic

$O(2^n)$ -exponential

$O(\log n)$ -logarithmic

OMEGA NOTATION

It represents the lower bound of algorithms running time.

It is the shortest amount of time taken by algorithm to complete.

$$F(n) > C * g(n).$$

This is denoted by $f(n) = \Omega g(n)$.

THETA NOTATION

It represents the running time between upper bound and lower bound.

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

It is denoted by $f(n) = \theta g(n)$.

SPACE COMPLEXITY

This is the amount of memory required by an algorithm to run.

There are two factors to compute space complexity.

1) constant

2)instance

SPACE COMPLEXITY

The space requirement $S(p)$ can be given as

$$S(p) = C + S_p$$

C is the constant

S_p is a space dependent upon instance characteristics.

DATA STRUCTURE

The data structure can be defined as the collection of elements and all the possible operations which are required for those set of elements.

Or

Data structure is a combination of a set of elements and corresponding set of operations.

The data structures can be implemented by building the suitable algorithms for them.

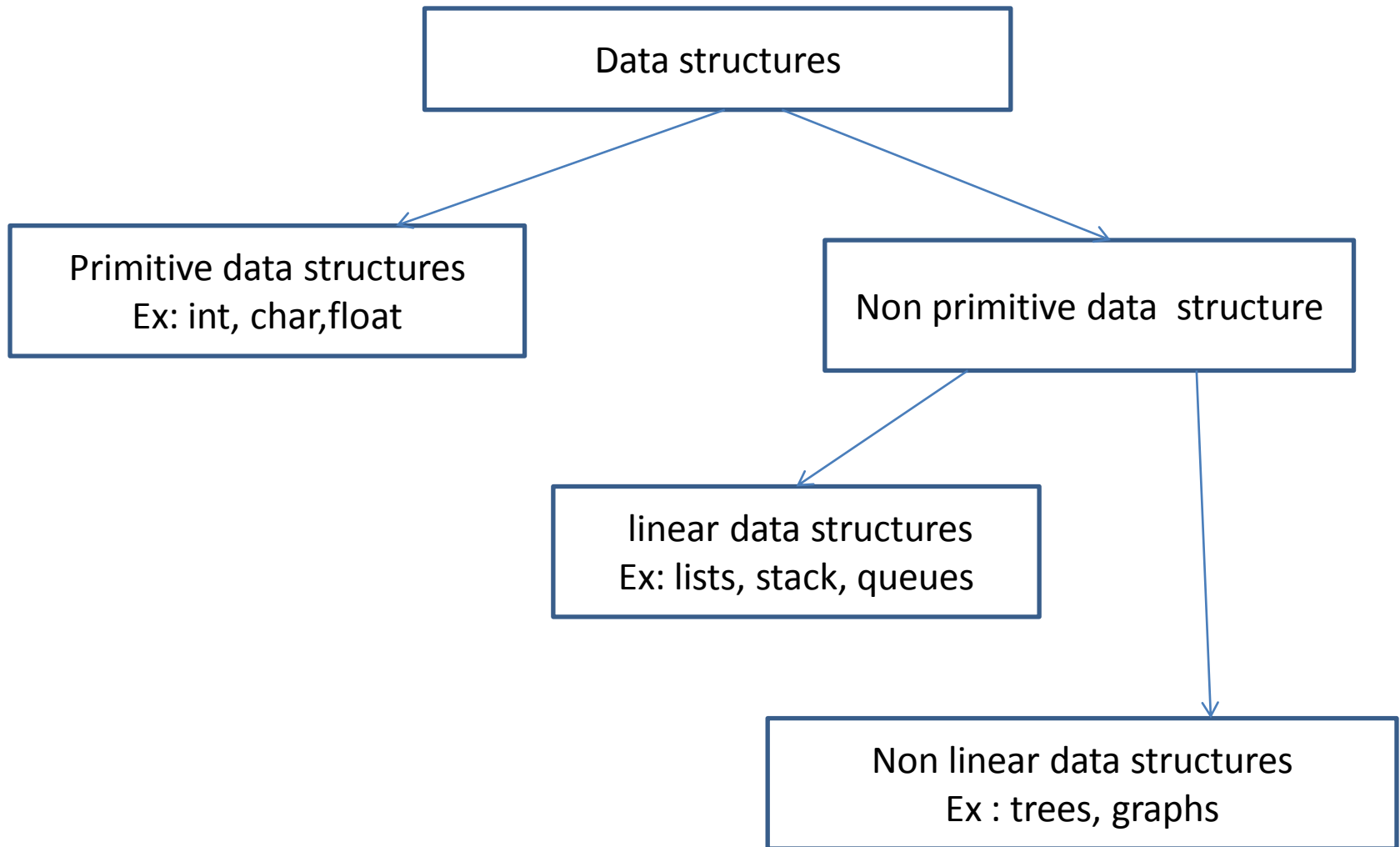
TYPES OF DATA STRUCTURES

The data structure can be divided into two basic types.

- 1) Preliminary data structures

- 1) Secondary data structures

TYPES OF DATA STRUCTURES



LIST

List is the collection of elements arranged in a sequential manner.

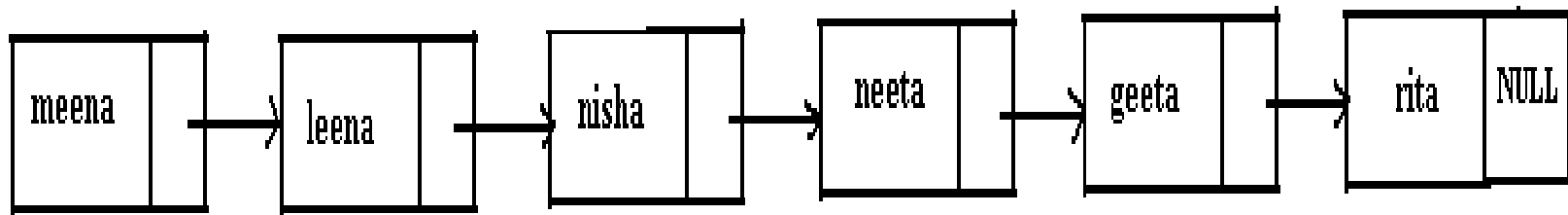
There are two representations

- 1) list of sequentially stored elements----using arrays
- 2) list of elements with associated pointers---using linked list.

LIST REPRESENTATION

meena	leena	nisha	neeta	geeta	rita
-------	-------	-------	-------	-------	------

list of sequentially stored elements using arrays



OPERATIONS ON AN ORDERED LIST

- 1)display of list.
- 2)search an element in the list.
- 3) insert an element into the list.
- 4) delete an element from the list.

SINGLY LINKED LIST

In the single linked list, a node is connected to the next node by a single link.

In this list a node contains two types of fields-
data:

which holds a list element

next(pointer):

which holds a link to the next node in the list.

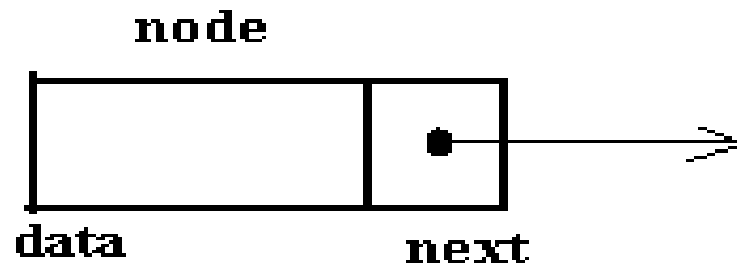
The head of the pointer is used to gain access to the list and the end of the list is denoted by a NULL pointer

STRUCTURE OF A SINGLE LINKED LIST

```
struct node
```

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

The list holds two members ,an integer type variable “data” which holds the elements and another member of type “node”, which h



SINGLE LINKED LIST OPERATIONS

Creating a linked list

Inserting in a linked list

Deleting a linked list

Searching an element in the linked list

Display the elements

Merging two linked list

Sorting a linked list

Reversing a list

CREATING A LINKED LIST

List can be created by using pointers and dynamic memory allocation function such as **malloc**.

The head pointer is used to create and access unnamed nodes.

CREATING A LINKED LIST

```
struct list
```

```
{
```

```
    int no;
```

```
    struct list *next;
```

```
};
```

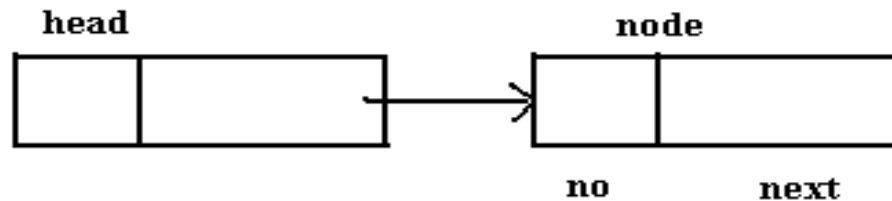
```
typedef struct list node;
```

```
node *head;
```

```
head=(node*) malloc (size of(node));
```


CREATING A LINKED LIST

The statement obtains memory to store a node and assigns its address to head which is a pointer variable.



To store values in the member fields :

```
head→no=10;  
head→next=NULL;
```

The second node can be added as:

```
head→next=(node*)malloc(sizeof(node));  
head→next→number=20;  
head→next→next=NULL;
```

INSERTING AN ELEMENT

Insertion is done in three ways:

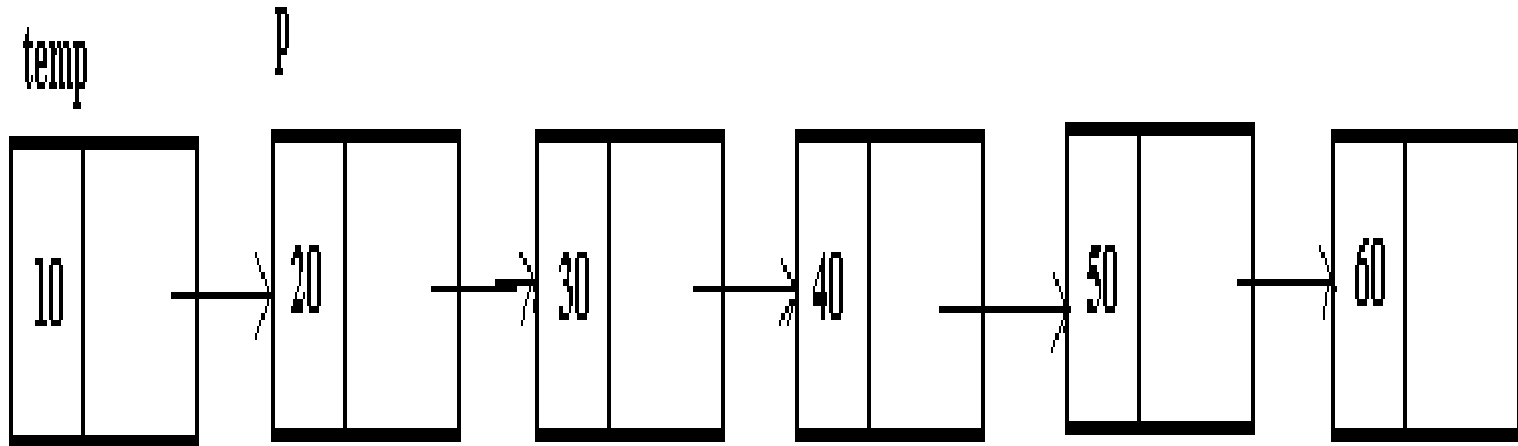
Insertion at the beginning of the list.

Insertion after any specified node.

Inserting node at the end of the list.

INSERTING AN ELEMENT

Function to insert a node at the beginning of the list:



INSERTING AN ELEMENT

Function to insert a node at the beginning of the list:

```
void add_beg(struct node **q, int no)
{
    struct node *temp;    /*add new node*/

    temp→data=no;

    temp→next=*q;t

    *q=temp;

}
```

here temp variable is take and space is allocated using “malloc” function.

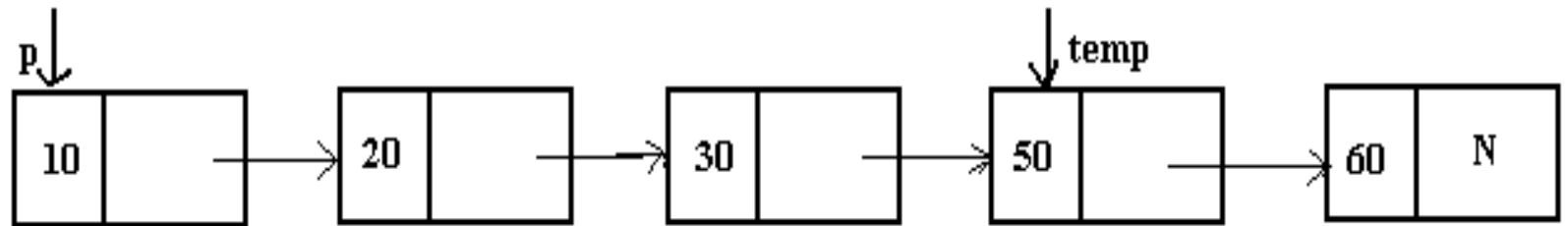
INSERTING AN ELEMENT

Insertion after any specified node:

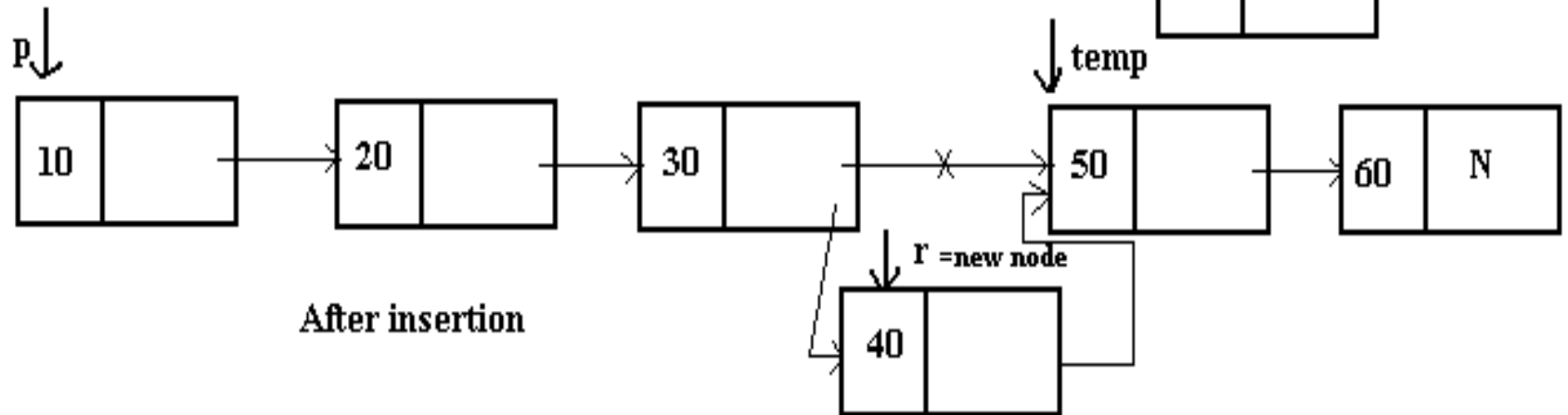
Inserting a node in the middle of the list,

if you consider to insert a node after the element then the process is as follows.

INSERTING AN ELEMENT



before insertion



After insertion

INSERTING AN ELEMENT

Function to insert a node at the middle of the list:

Void add_after(struct node *q, int loc, int no)

```
{
    struct node *temp, *r;

    int l;

    temp=q;/*skip to desire portion*/

    for(i=0;i<loc;i++)
    {
        temp=temp→next;
```

INSERTING AN ELEMENT

```
if(temp==NULL)
{
    printf("\n there are less than %d elements in list",loc);
    return;
}
/*insert new node*/
r=malloc(sizeof(struct node));

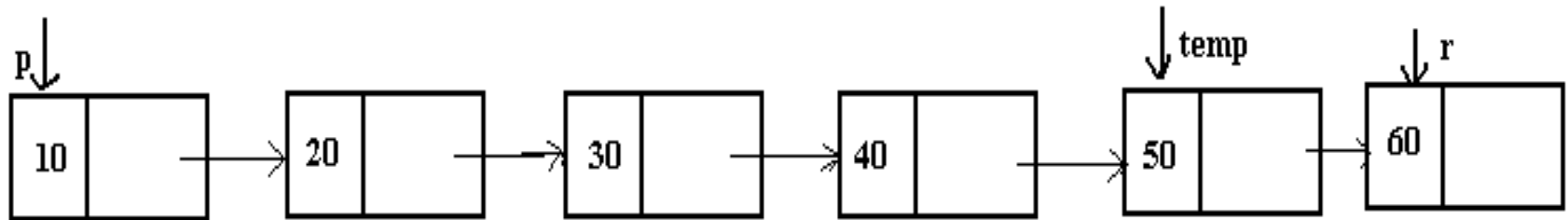
r→data=n0;

r→next=temp→next;

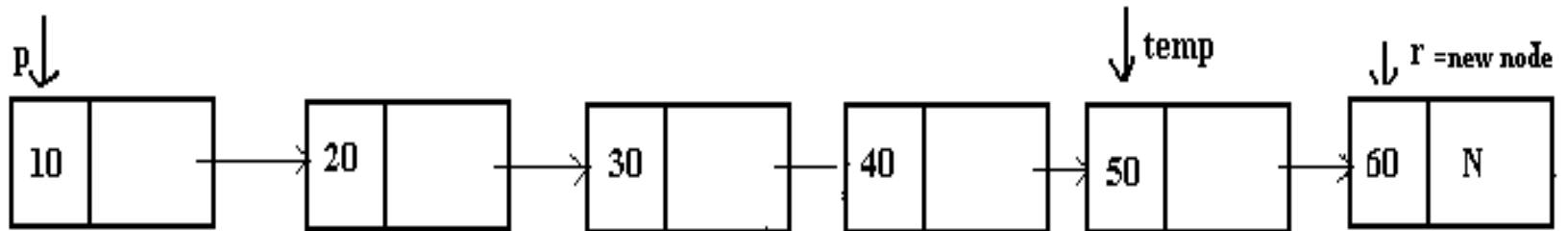
temp→next=r;
```


INSERTING AN ELEMENT

Inserting node at the end of the list:



before insertion



After insertion

INSERTING AN ELEMENT

Inserting node at the end of the list:

```
void create(struct node **q, int no)
{
    struct node *temp,*r;

    if(*q==NULL)    /*if the list is empty,create first node*/
    {
        temp=malloc(sizeof(struct node));

        temp→data=no;
```

INSERTING AN ELEMENT

```
temp→next=NULL;
```

```
*q=temp;
```

```
}
```

```
else
```

```
{
```

```
temp=*q;  /* go to last node*/
```

```
while(temp→next!=NULL)
```

INSERTING AN ELEMENT

```
temp=temp→next;
```

```
r=malloc(sizeof(struct node));
```

```
r→data=no;
```

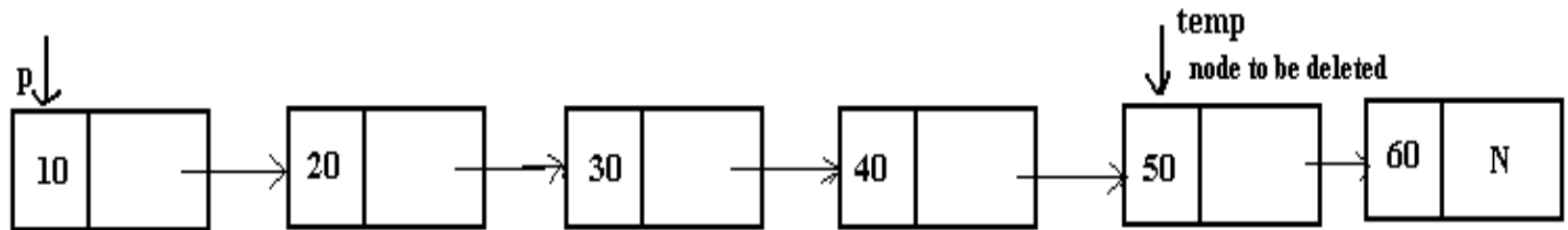
```
r→next=NULL;
```

```
temp→next=r;
```

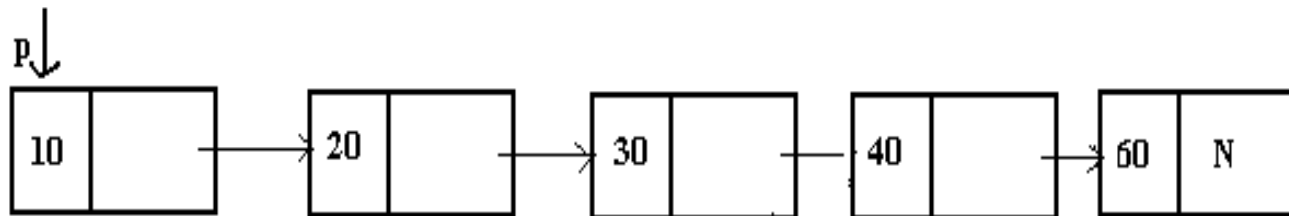
```
}
```

```
}
```

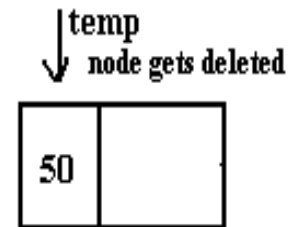
DELETING AN ELEMENT



before deletion



after deletion



DELETING AN ELEMENT

We traverse through the entire linked list to check each node whether it has to be deleted.

if we want to delete the first node in the list then we shift the structure type pointer variable to the next node and then delete the entire node.

if the node is a intermediate node then the various pointers the linked list before and after deletion should be taken care of

DISPLAYING THE CONTENTS OF THE LINKED LIST

Displays the elements of the linked list contained in the data part.

Function to display the contents of the linked list.

```
void display(struct node *start)
{
    printf("\n");
```

DISPLAYING THE CONTENTS OF THE LINKED LIST

```
/*traverse the entire list*/
```

```
while(start!=NULL)
```

```
{
```

```
printf(“%d”,start→data);
```

```
start=start→next;
```

```
}
```

```
}
```


OTHER OPERATIONS OF SINGLY LINKED LIST

Searching the linked list:

Searching means finding information in a given linked list.

Reversing a linked list:

The reversing of the linked list that last node becomes the first node and first becomes the last.

OTHER OPERATIONS OF SINGLY LINKED LIST

Sorting the list:

In sorting function the node containing the largest element is removed from the linked list and is appended to the new list in the ascending order. **CREATING A LINKED LIST**

Merging the two linked list:

Merging two list pointed by two pointers into a third list.

While merging be ensure that the elements common to the lists appear only once in the third list.

CIRCULAR LINKED LIST

A linked list in which last node points to the header node is called the circular linked list.

The list have neither a beginning nor an end.

In this list the last node contains a pointer back to the first node rather than the NULL pointer.

CIRCULAR LINKED LIST

The structure defined for circular linked list

```
struct node
```

```
{
```

```
    int data;
```

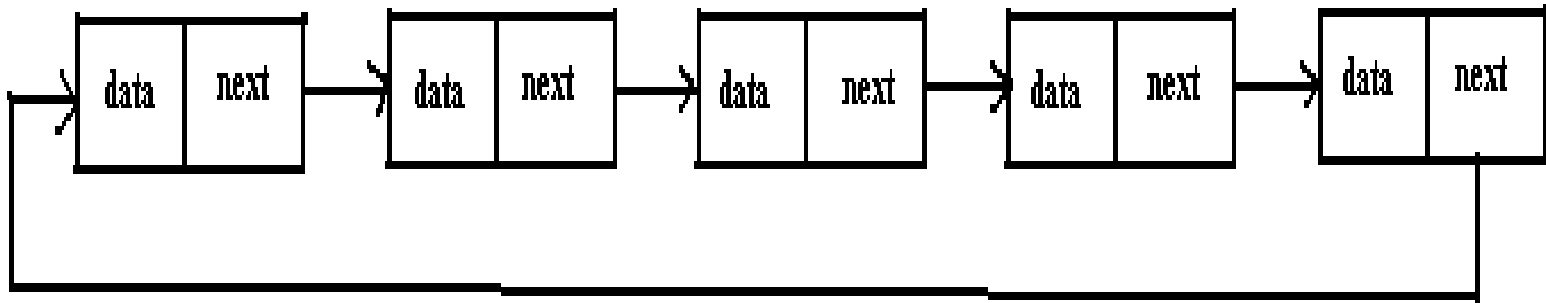
```
    struct node *next;
```

```
}
```

CIRCULAR LINKED LIST

A circular linked list is represented as follows:

A circular linked list can be used to represent a stack and a queue.



OPERATION OF CIRCULAR LINKED LIST

Adding elements in the circular linked list.

Deleting element from the circular list.

Displaying elements from the circular list.

ADDING ELEMENTS IN THE CIRCULAR LINKED LIST

Ciradd():

this function accepts three parameters:

receives the address of the pointer to the first node.

receives the address of the pointer to the last node.

holds the data items that need to add in the list.

DELETING ELEMENTS FROM THE CIRCULAR LINKED LIST

delcirq():

this function receives two parameters.

the pointer to the front .

the pointer to the rear .

DELETING ELEMENTS FROM THE CIRCULAR LINKED LIST

The condition is checked for the empty list.

If the list is not empty,

then it is checked whether the front and rear point to the same node or not.

If they point to the same node,

then the memory occupied by the node is released and front and rear are both assigned a NULL value.

DISPLAYING THE CIRCULAR LIST

Cirq_disp():

the function receives the pointer to the first node in the list as a parameter.

The q is also made to point to the first node in the list.

The entire list is traversed using q.

Another pointer p is set to NULL initially.

The circular list is traversed through a loop till the time it reach the first node again.

It reach first node again when q equals p.

DOUBLY LINKED LIST

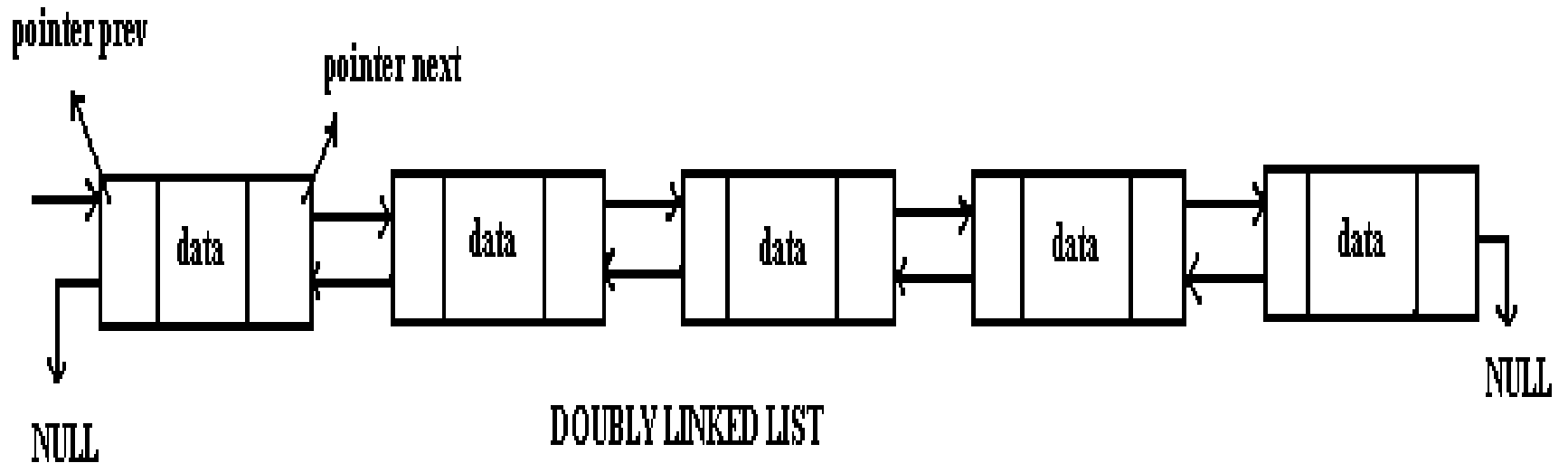
The doubly linked list uses double set of pointer's, one pointing to the next item and the other pointing to the preceding item.

It can traverse in two directions:

from the beginning of the list to the end
or

In the backward direction from the end of the list to the beginning.

DOUBLY LINKED LIST



DOUBLY LINKED LIST

Each node contains three parts:

An information field which contains the data.

A pointer field next which contains the location of the next node n the list.

A pointer field prev which contains the location of the preceding node in the list.

Structure to define DLL:

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

CREATING A DLL

To create DLL at the nodes to the existing list:

To create the list the function `d_create` can be used before creating the list the function checks if the list is empty.

Here the function accepts two parameters.

`s` of type `struct dnode **` which contains the address of the pointer to the first node of the list.

parameter `num` is an integer which is to be added in the list.

CREATING A DLL

To create DLL at the nodes to the existing list:

To create the list the function `d_create` can be used before creating the list the function checks if the list is empty.

Here the function accepts two parameters.

`s` of type `struct dnode **` which contains the address of the pointer to the first node of the list.

parameter `num` is an integer which is to be added in the list.

OPERATIONS OF DLL

Adding a node in the beginning of DLL:

To add the node at the beginning of the list the function `d_addatbeg()` is used .

This function takes two parameters:

s of type `dounode **` which contains the address of the pointer to the first node .

num is an integer to be added in the list.

OPERATIONS OF DLL

The allocation of memory for the new node is done whose address is stored in q.

The num is the data part of the node.

A NULL value is stored in the prev part of new node as this is the first node in the list.

OPERATIONS OF DLL

Function to add a node at the beginning of list.

```
Void d_addatbeg(struct dnode **s,int num)
{
    struct dnode *q;
    q=malloc(sizeof(struct dnode));
    q→prev=NULL;
    q→data=num;
    q→next=*s;
    (*s)→prev=q;
    *s=q;
}
```

OPERATIONS OF DLL

Adding a node in the middle of the list:

To add the node in the middle of the list we use the function `d_addafter()`.

The function accepts three parameters.

`q` points to the first node of the list.

`loc` specifies the node number after which new node must be inserted.

`num` which is to be added to the list.

To reach to the position where node is to be inserted, a loop is executed.

OPERATIONS OF DLL

Deleting a node from DLL:

This function deletes a node from the list if the data part matches a with num.

The function receives two parameters
the address of the pointer to the first node.
the number to be deleted.

To traverse the list ,a loop is run.

The data part of each node is compared with the num.

If the num value matches the data part, then the position of the node to be deleted is checked

OPERATIONS OF DLL

Display the contents of DLL.

to display the contents of the doubly linked list, we follow the same algorithm that had used in the singly linked list.

Here q points to the first node in the list and the entire list is traversed .

Function to display the DLL.

```
void d_disp(struct dnode *q)
{ printf("\n");
while(q!=NULL)
{ printf("%2d",q->data);
q=q->next;
}
}
```

ARRAYS

A collection of objects of the *same type* stored contiguously in memory under one name.

May be type of any kind of variable

May even be collection of arrays!

The elements of the array are stored in consecutive memory locations and are referenced by an index (subscript).

ARRAYS

To refer to an element, specify

- Array name

- Position number

Syntax:

`array_name[position number]`

ARRAYS

Array Declaration

When declaring arrays

- Name
- Type of data elements
- Number of elements

Syntax

Data_Type array_Name[Number_Of_Elements];

ARRAYS

Examples:

```
int c[ 10 ];
```

```
float myArray[ 3284 ];
```

Declaring multiple arrays of same type

- Format similar to regular variables
- Example:

```
int b[ 100 ], x[ 27 ];
```

ARRAYS

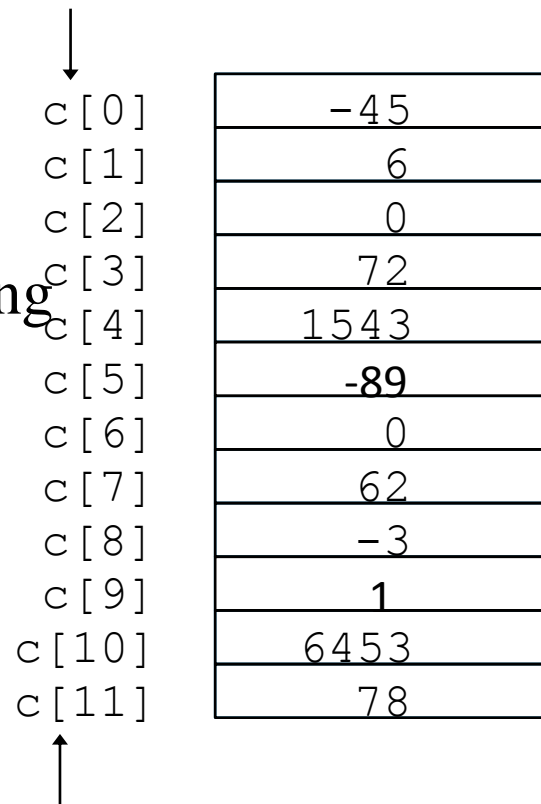
`int c[12]`

- An array of ten integers
- `c[0], c[1], ..., c[11]`

`double B[20]`

- An array of twenty long floating point numbers
- `B[0], B[1], ..., B[19]`

Name of array (Note that all elements of this array have the same name, `c`)



<code>c[0]</code>	-45
<code>c[1]</code>	6
<code>c[2]</code>	0
<code>c[3]</code>	72
<code>c[4]</code>	1543
<code>c[5]</code>	-89
<code>c[6]</code>	0
<code>c[7]</code>	62
<code>c[8]</code>	-3
<code>c[9]</code>	1
<code>c[10]</code>	6453
<code>c[11]</code>	78

Position number of the element within array `c`

ARRAYS

Arrays of structs, unions,
pointers, etc., are also allowed

Array indexes always

start at zero in *C*

ARRAYS

Two Dimensional Array

- Syntax

`Data_Type array_Name[Row_Elements][Column_Elements];`

- Example

`int D[10][20]`

- An array of ten rows, each of which is an array of twenty integers
- `D[0][0], D[0][1], ..., D[1][0], D[1][1], ..., D[9][19]`
- Not used so often as arrays of pointers

ARRAYS

Two Dimensional Array

- Multiple subscripted arrays as
 - Tables with rows and columns (m×n array)
 - Like matrices: specify row, then column

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Diagram illustrating the structure of a 2D array (matrix) with rows and columns. The array is represented as a table with 3 rows (Row 0, Row 1, Row 2) and 4 columns (Column 0, Column 1, Column 2, Column 3). Each element is accessed using the format `a[row][column]`.

Labels and arrows indicating the components of the array notation:

- Array name:** Points to the `a` in the notation.
- Row subscript:** Points to the first index (row number) in the notation.
- Column subscript:** Points to the second index (column number) in the notation.

ARRAYS

Multi Dimensional Arrays

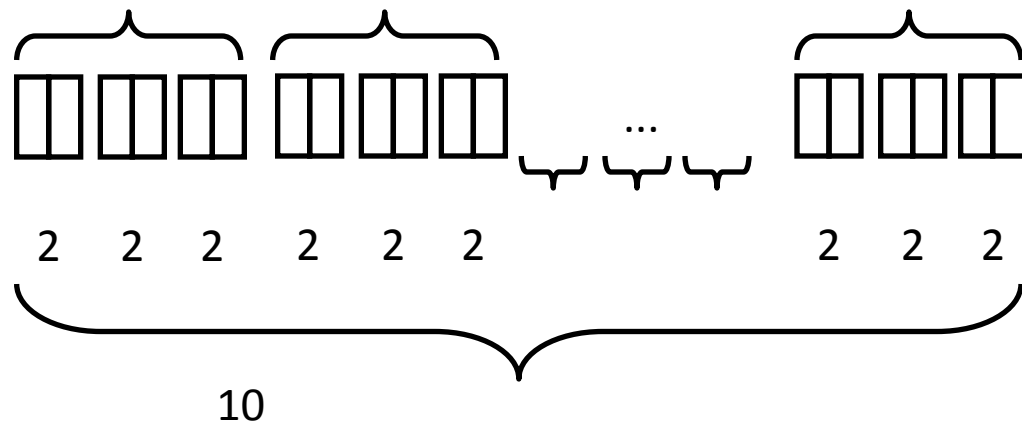
- Array declarations read right-to-left
- Syntax

`Data_Type array_Name[Size][Size][Size] ... Size];`

- Example

`int a[10][3][2];`

“an array₃ of ten arrays of three arrays of two elements”
in memory



SPARSE MATRIX

A sparse matrix is a matrix that allows special techniques to take advantage of the large number of zero elements.

Sparse matrix is very useful in engineering field, when solving the partial differentiation equations.

if there are maximum zeros then the matrix is known as sparse. matrix.

if there are few zeros then the matrix is dense matrix.

CATEGORIES OF SPARSE MATRIX

Sparse matrix has

N^2 sparse matrix and

Triangular sparse matrix

A matrix with zero entries that form a square or a bar is N^2 sparse matrix.

A matrix with zero entries in its diagonal either in the upper or lower side is known as triangular sparse matrix.

REPRESENTATION OF SPARSE MATRIX

Sparse matrix can be represented in

Tuple method

Array representation

Linked list representation

Only non zero elements are stored in any of the above representations.

TUPLE METHOD

consider a matrix

[15 0 0 21
22 11 0 0
0 19 35 16]

This is N^2 sparse matrix

Tuple matrix is

row	column	value
1	1	15
1	4	21
2	1	22
2	2	11
3	2	19
3	3	35
3	4	16

TUPLE METHOD

consider a matrix

[4 0 0 0
3 11 0 0
1 22 33 0
7 45 41 22]

This is triangular matrix

Tuple matrix is

row	column	value
1	1	4
2	1	3
2	2	11
3	1	1
3	2	22
3	3	33
4	1	7
4	2	45
4	3	41
4	4	22

ARRAY METHOD

consider a matrix

$$\begin{bmatrix} 15 & 0 & 0 & 21 \\ 22 & 11 & 0 & 0 \\ 0 & 19 & 35 & 15 \end{bmatrix}$$

This is triangular matrix

The elements are represented as follows

1,1,15	1,4,21	2,1,22	2,2,11	3,2,11	3,2,19	3,3,35	3,4,16
--------	--------	--------	--------	--------	--------	--------	--------

SPARSE MATRIX OPERATIONS USING ARRAYS

Addition of two sparse matrix:

The function `addmat()` carries addition

The function `display()` displays the result.

Multiplication of two sparse matrix:

SPARSE MATRIX OPERATIONS USING ARRAYS

This holds three functions

Sparseprod() stores the result.

Search_nonzero() checks whether a non zero element is present or not.

Searchinb() searches an element whose row number is equal to column number.

Transpose of a sparse matrix:

Transpose() is used to allocate memory to store the elements.

REPRESENTATION OF SPARSE MATRIX THROUGH LINKED LIST

The elements of sparse matrix consist of three integers.

Its row number

Its column number

Its value

The head node consist of three parts.

Row number indicates the row to which the “head” node is pointing to the component element.

The head also points to another head the node for the next row.

REPRESENTATION OF SPARSE MATRIX THROUGH LINKED LIST

The `create_list()` function stores the information in the form of linked list.

`Insert()` accepts a pointer to the special node .

`show_list()` reads and displays the data stored in the linked list.

REPRESENTATION OF SPARSE MATRIX THROUGH LINKED LIST

next row	row number	elements in this row
----------	------------	-------------------------

parts of head node for a row

REPRESENTATION OF SPARSE MATRIX THROUGH LINKED LIST

next element in this column	row number	column number	value	elements in this row
--------------------------------	------------	------------------	-------	-------------------------

structure for an element

REPRESENTATION OF SPARSE MATRIX THROUGH LINKED LIST

first row	number of row	no. of columns	first columns
-----------	---------------	----------------	---------------

node

UNIT-II

UNIT-2

UNIT-II

Stack ADT, definition, operations, array and linked implementations in C, applications-infix to postfix conversion, Postfix expression evaluation, recursion implementation, Queue ADT, definition and operations ,array and linked Implementations in C, Circular queues-Insertion and deletion operations, Deque (Double ended queue)ADT, array and linked implementations in C.

STACKS

A stack is a linear structures in which addition or deletion of elements takes place at the same end.

Or

The stack is an ordered list in which insertion and deletion is done at the same end.

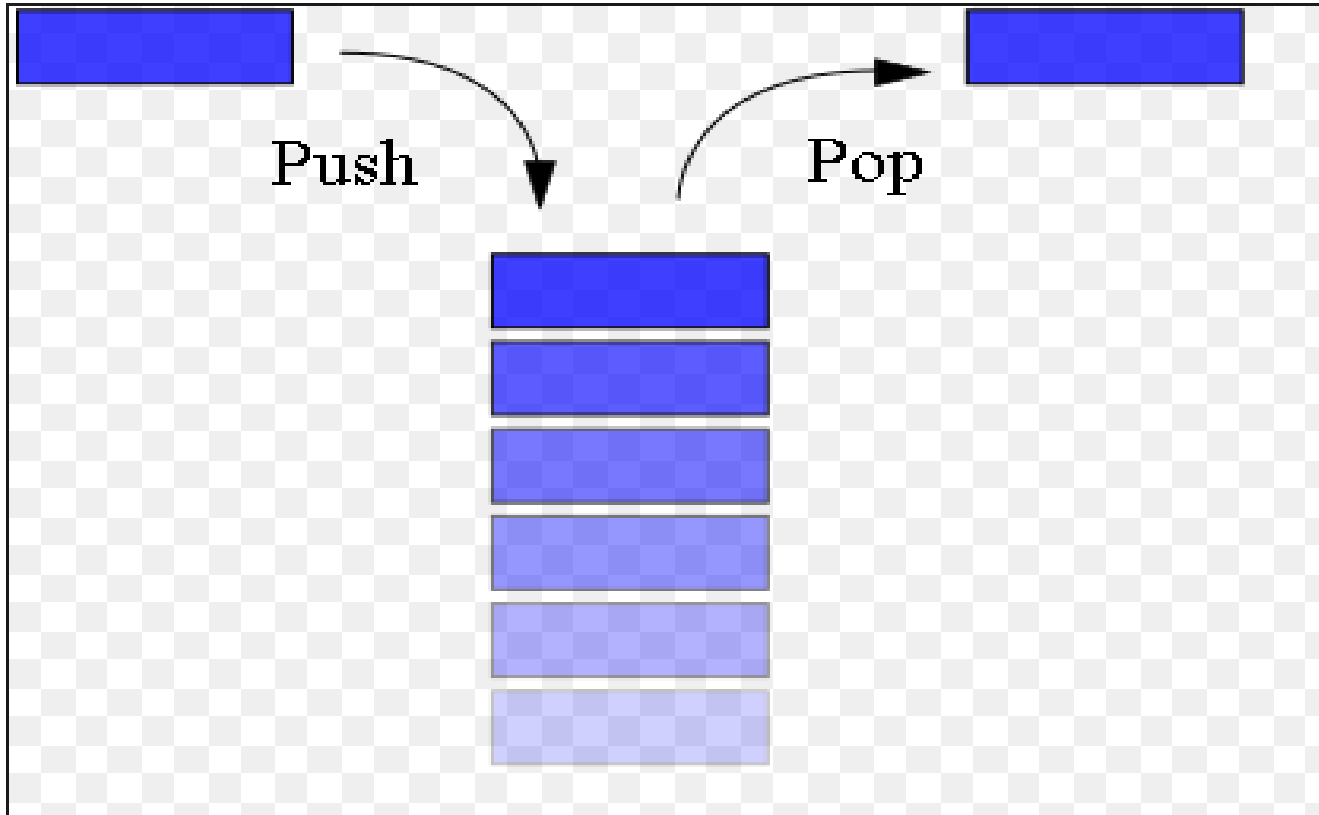
The end is called the top of stack.

Insertion and deletion cannot be done from the middle.

A technique of Last In First Out is followed.

Stack can be implemented by using both arrays and linked lists.

STACKS



STACK ADT

Stacks can also be defined as Abstract Data Types(ADT).

A stack of elements of any particular type is a finite sequence of elements of that type together with specific operations.

Therefore, stacks are called LIFO lists.

STACK OPERATIONS

The primitive operations on stack are

To create a stack.

To insert an element on to the stack.

To delete an element from the stack.

To check which element is at the top of the stack.

To check whether a stack is empty or not.

STACK OPERATIONS

If Stack is not full ,
then add a new node at one end of the stack
this operation is called PUSH.

If the stack is not empty
then delete the node at its top.
This operation is called POP.

PUSH and POP are functions of stack used to fulfill the stack operations.

TOP is the pointer locating the stack current position.

ARRAY IMPLEMENTATION IN C

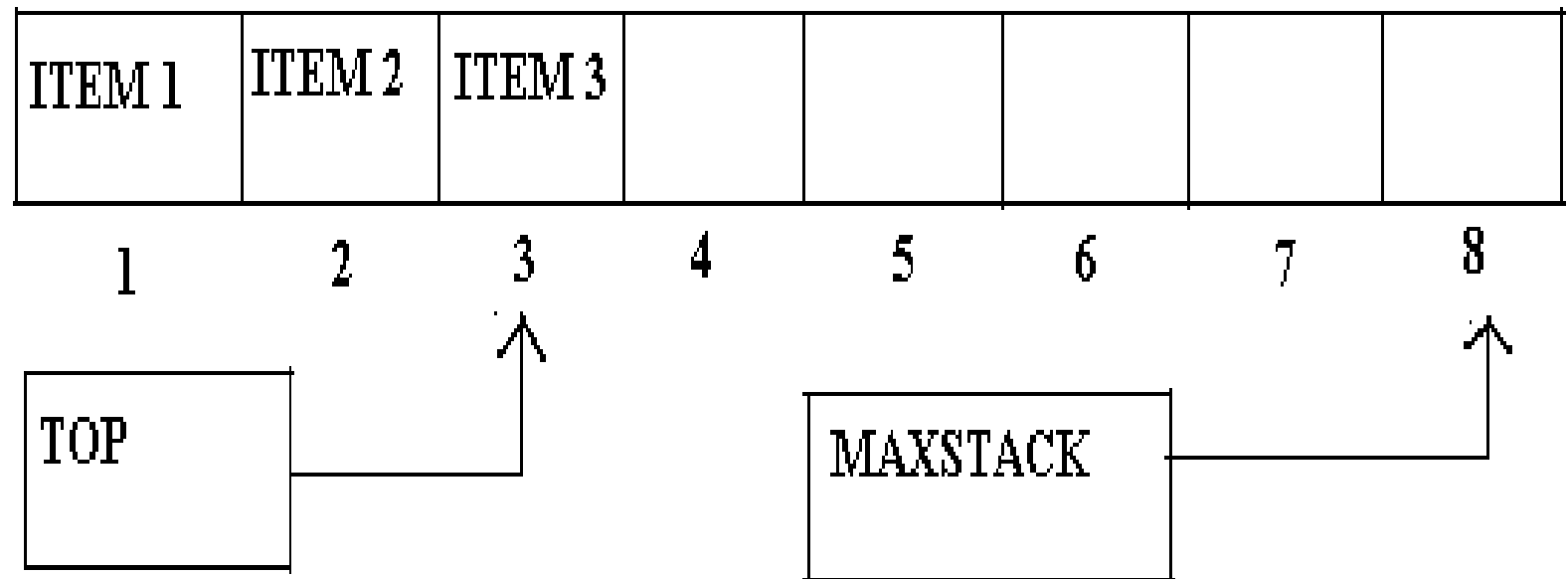
Stacks can be represented in the memory arrays by maintaining a linear array STACK and a pointer variable TOP which contains the location of top element.

The Variable MAXSTACK gives maximum number of elements held by the stack.

The TOP=NULL /0 will indicate that the stack is empty.

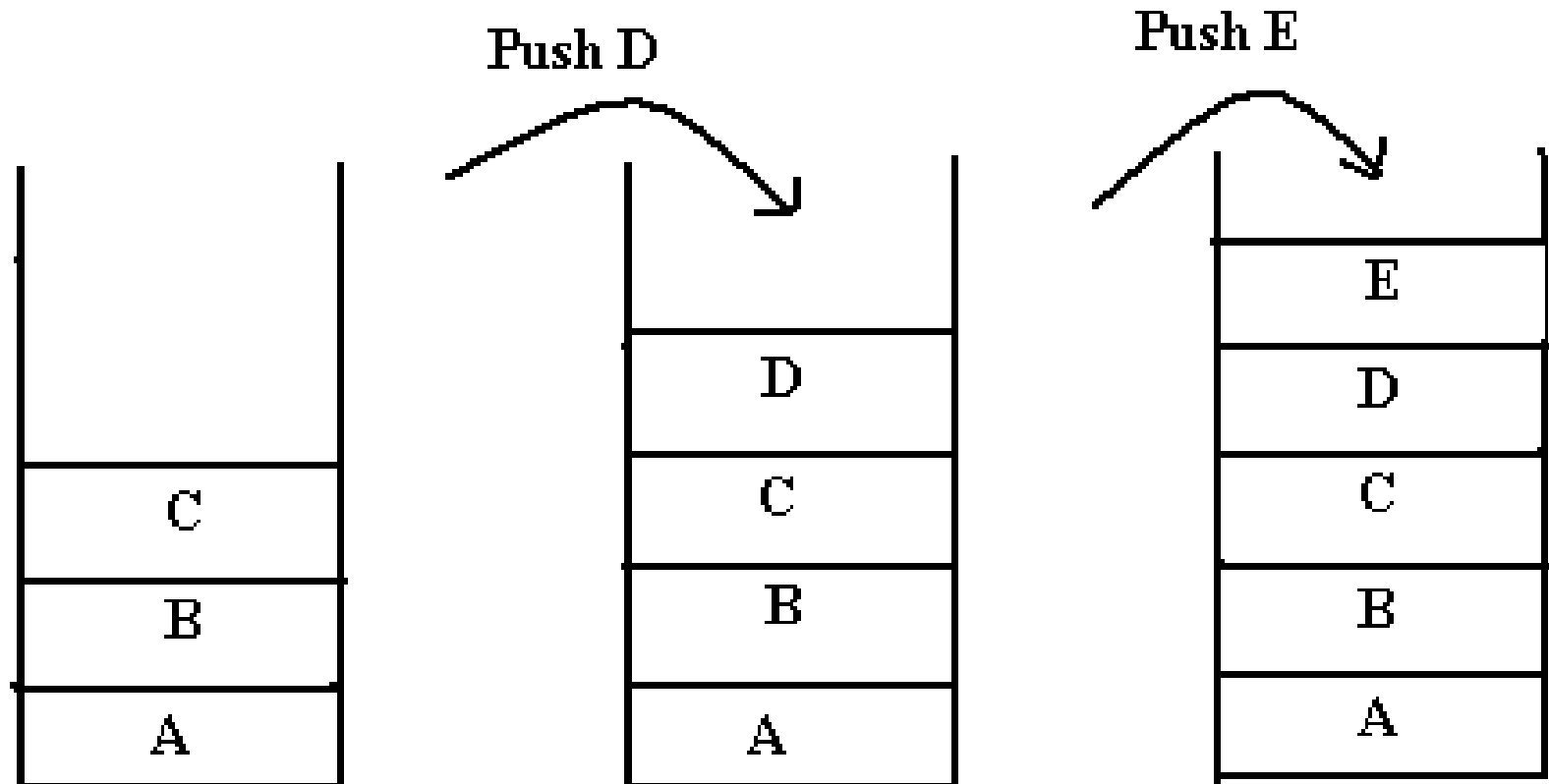
The operation of adding and removing an item in the stack can be implemented using the PUSH and POP functions.

Figure shows the array representation



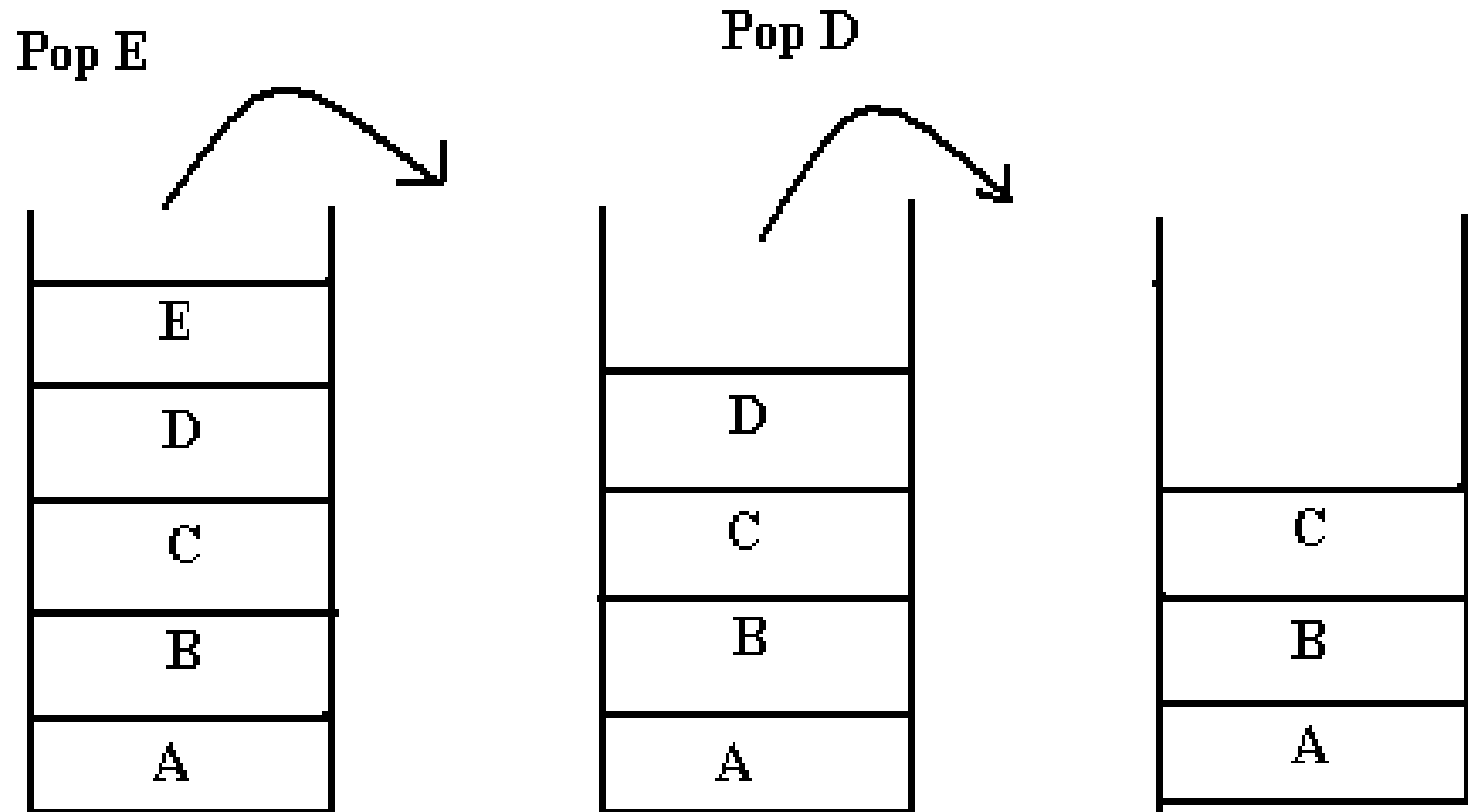
ARRAY REPRESENTATION OF A STACK

Pictorial depiction of pushing elements in stack



Pushing Elements in Stack

Pictorial depiction of popping elements in stack



Popping Elements from stack

DISADVANTAGE OF STACK USING ARRAYS

The array representation of stack suffers from the drawbacks of the array's size, that cannot be increased or decreased once it is declared .

The space is wasted, if not used , or, there is shortage of space if needed.

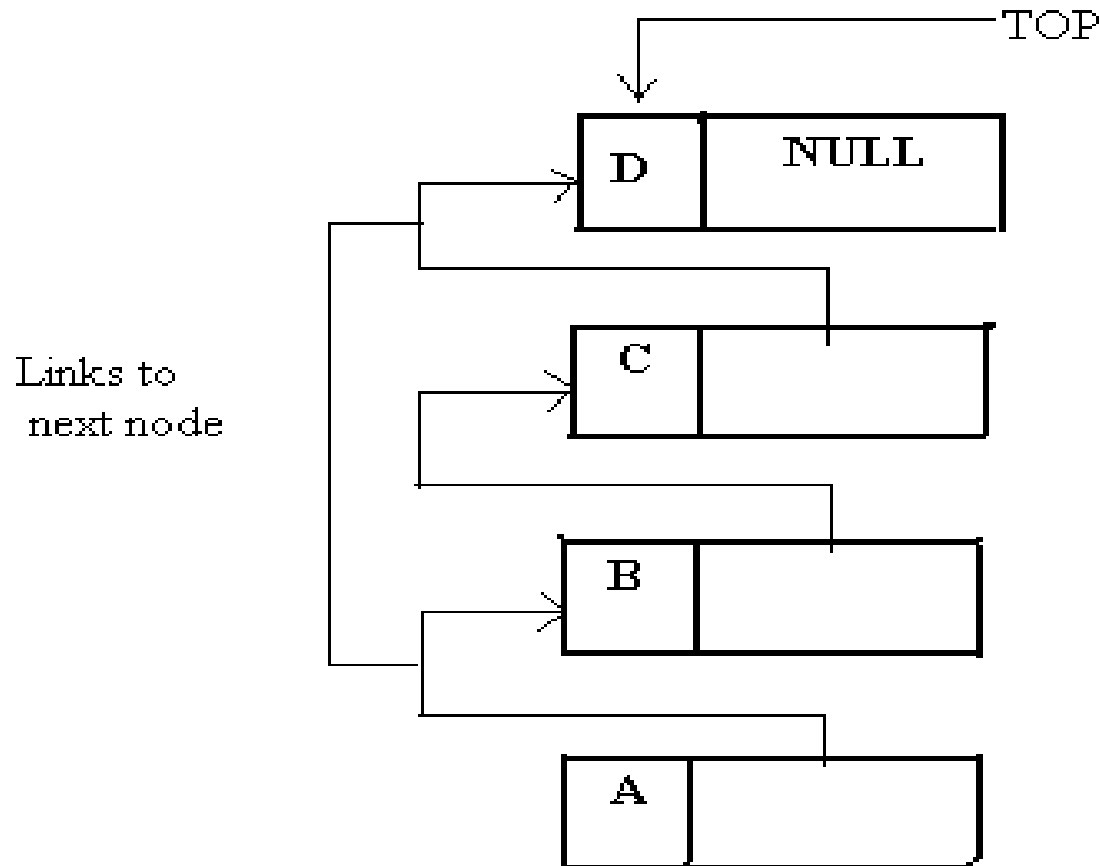
LINKED IMPLEMENTATION IN C

The stack can be implemented using linked lists.

The stack as linked list is represented as a single linked list.

Each node in the list contains data and a pointer to the next node.

Pictorial depiction of stack in linked list



APPLICATION OF STACKS

Reversing a list.

Conversion of Infix to Postfix Expression.

Evaluation of Postfix Expression.

Conversion of Infix to Prefix Expression.

Evaluation of Prefix Expression.

CONVERSION OF INFIX TO POSTFIX EXPRESSION

While evaluating an infix expression,
operations are executed according to the order as follows:

Brackets / Parentheses.

Exponentiation.

Multiplication / Division.

Addition / Subtraction.

the operators with the same priority(e.g. * and /) are
evaluated from left to right.

STEPS TO CONVERT INFIX TO POSTFIX EXPRESSION

Step 1: The actual evaluation is determined by inserting braces.

Step 2: Convert the expression in the innermost braces into postfix notation by putting the operator after the operands.

Step 3: Repeat the above step (2) until the entire expression is converted into postfix notation.

EXAMPLE OF INFIX TO POSTFIX CONVERSION

Infix	Postfix
$A + B$	$AB +$
$12 + 60 - 23$	$12 60 + 23 -$
$(A + B) * (C - D)$	$AB + CD - *$
$AB * C - D + E / F$	$ABC * D - EF / +$

RECURSION IMPLEMENTATION

If a procedure contains either a call statement to itself/to a second procedure that may eventually result in a call statement back to the original procedure. Then such a procedure is called as recursive procedure.

Recursion may be useful in developing algorithms for specific problems. The stack may be used to implement recursive procedures.

QUEUE

Queue is a linear list of elements in which deletion of an element can take place only at one end,

called the front

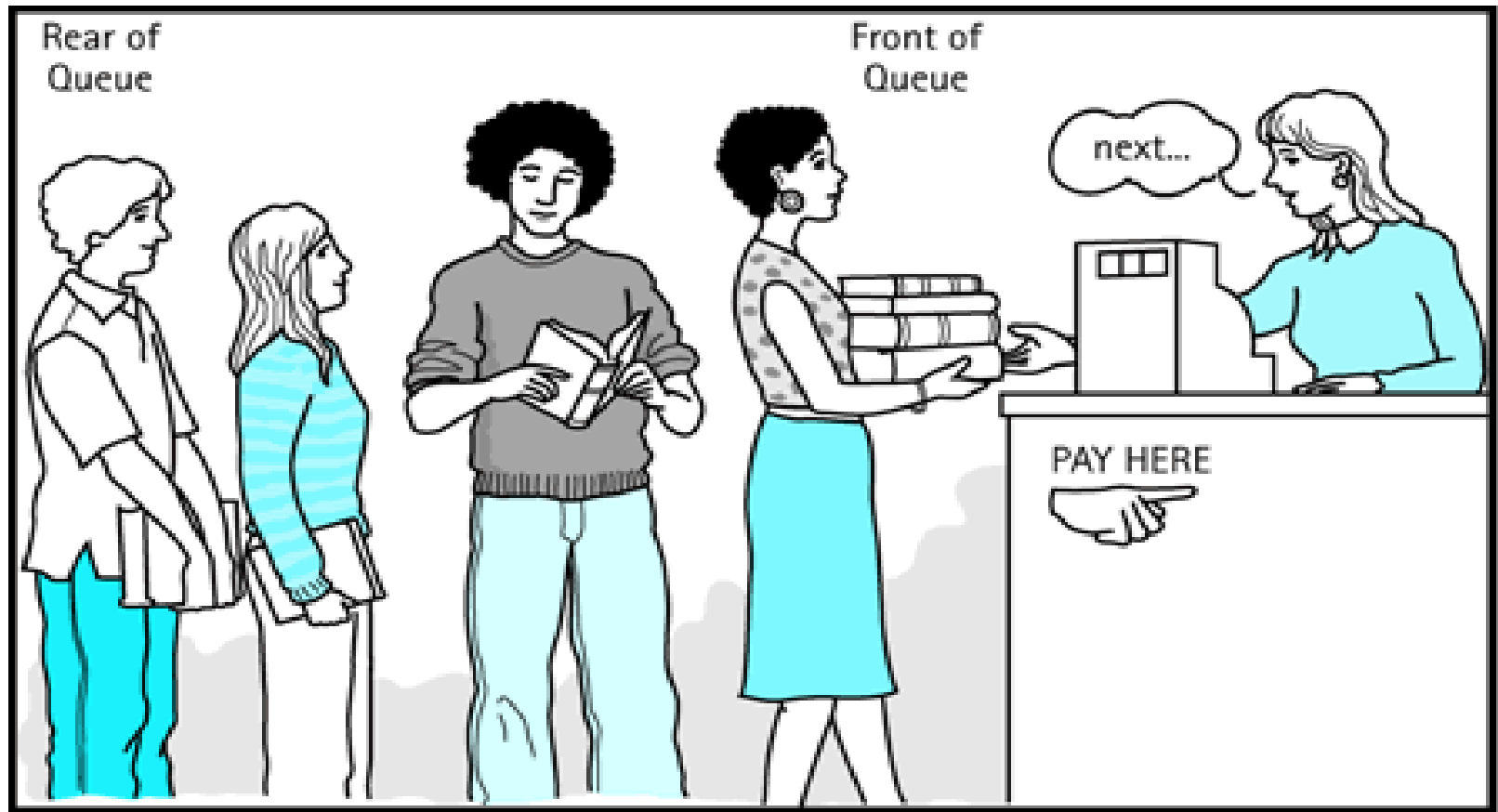
and insertion can take place only at the other end,

called the rear.

The first element in a queue will be the first one to be removed from the list.

Therefore, queues are called FIFO lists.

QUEUE



QUEUE ADT

The definition of an abstract data type clearly states that for a data structure to be abstract, it should have the two characteristics as follows.

There should be a particular way in which components are related to each other.

A statement of the operations that can be performed on element of the abstract data type should specified.

QUEUE OPERATIONS

Queue overflow.

Insertion of the element into the queue.

Queue underflow.

Deletion of the element from the queue.

Display of the queue.

ARRAY IMPLEMENTATION IN C

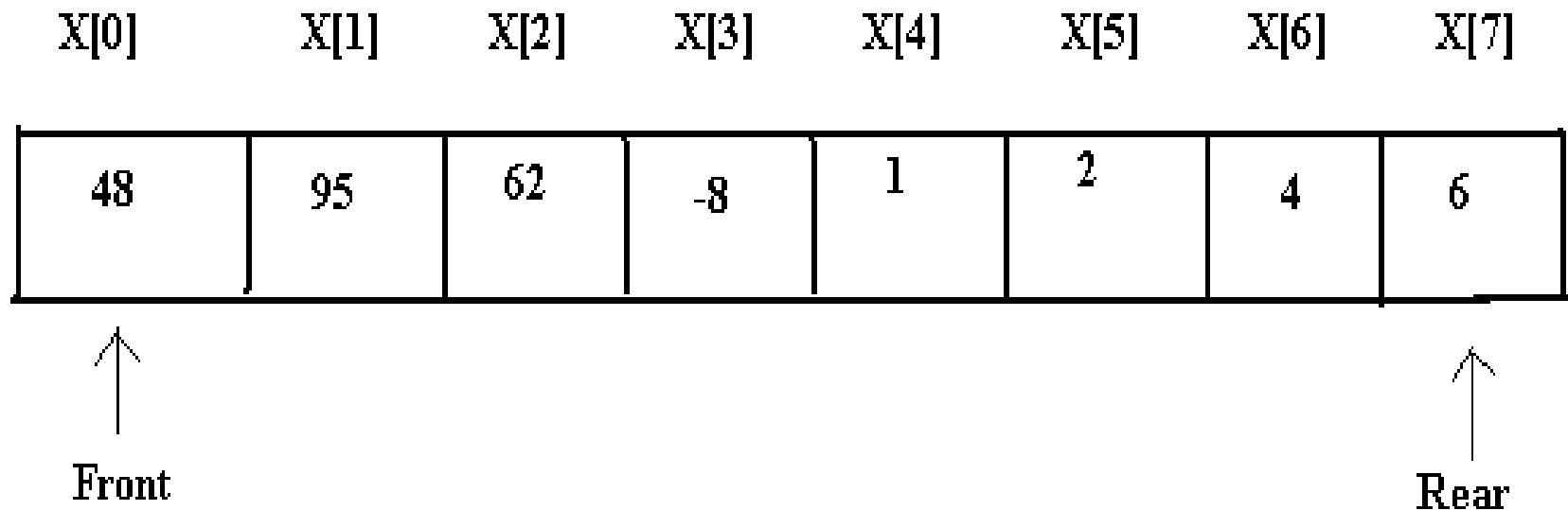
Array is a data structure that stores a fixed number of elements.

One of the major limitations of an array is that its size should be fixed prior to using it.

The size of the queue keeps on changing as the elements are either removed from the front end or added at the rear end.

The solution of this problem is to declare an array with a maximum size.

FIGURE TO REPRESENT A QUEUE USING ARRAY



INSERTION AND DELETION OPERATIONS IN QUEUE USING ARRAYS

We consider two variables front and rear which are declared to point to both the ends of the queue.

The array begins with index therefore , the maximum number of elements that can be stored can be consider as $MAX-1(n-1)$.

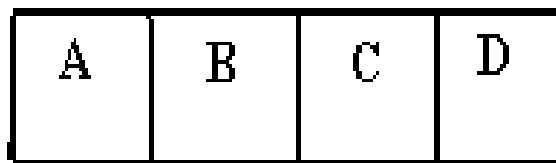
If the number of elements are already stored in the queue is reported to be full.

If the elements are added then the rear is incremented using the pointer and new item is stored in the array.

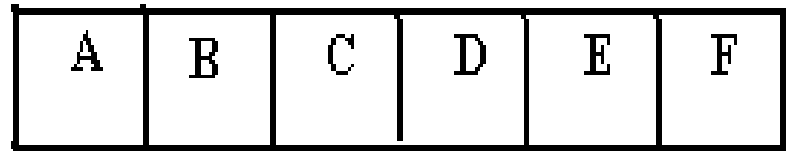
ADDING ELEMENTS IN A QUEUE

The front and rear variables are initially set to -1, which denotes that the queue is empty.

If the item being added is the first element then as the item is added, the queue front is set to 0 indicating that the queue is now full.



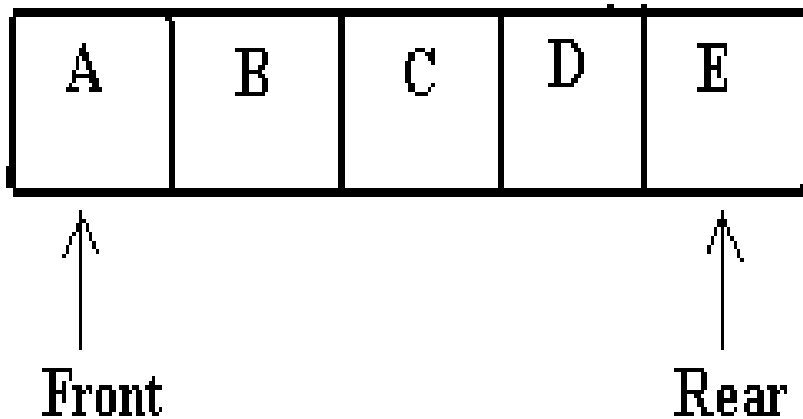
Before adding elements



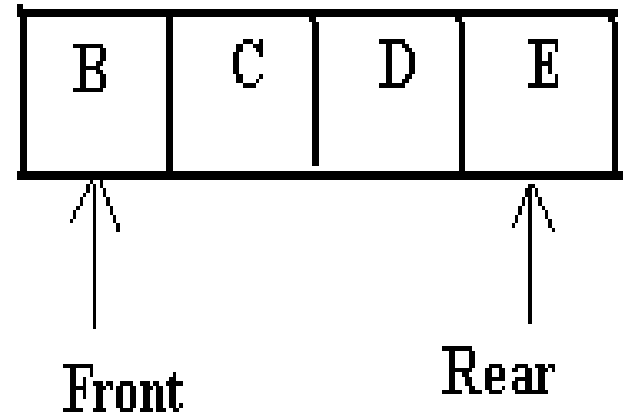
After adding elements

DELETING ELEMENTS IN A QUEUE

For deleting elements from the queue, the function first checks if there are any elements for deletion. If not, the queue is said to be empty otherwise an element is deleted.



Before deleting elements



After deleting elements

LINKED IMPLEMENTATION IN C

The linked list representation of a queue does not have any restrictions on the number of elements it can hold.

The elements are allocated dynamically , hence it can grow as long as there is sufficient memory available for dynamic allocation.

APPLICATION OF QUEUE

Job scheduling.

Categorizing data.

Random number generation.

TYPES OF QUEUES

Circular queue.

De queue (double ended queue).

Priority queue.

CIRCULAR QUEUE

Circular queues are implemented in circular form rather than in a straight line.

This form over come the problem of unutilized space in linear queue implemented as an array.

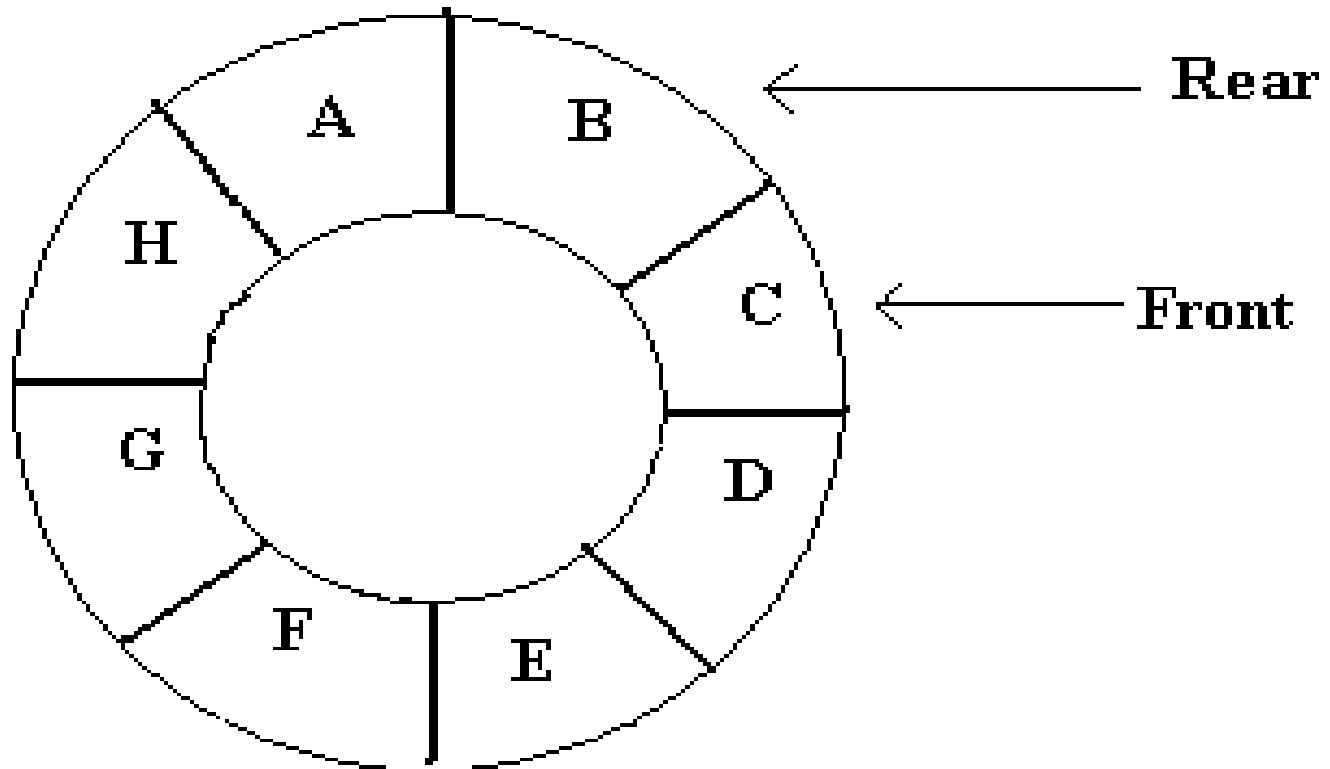
In the array implementation there is a possibility that the queue is reported full even though slots of the queue are empty.

CIRCULAR QUEUE

Suppose an array x of n elements is used to implement a circular queue. If we go on adding elements to the queue we may reach $x[n-1]$.

In a queue array if the elements reach the end then it reports the queue is full even some slots are empty but in circular queue ,it would not report as full until all the slots are occupied.

REPRESENTATION OF CIRCULAR QUEUE



Circular queue

ADDING ELEMENTS INTO CIRCULAR QUEUE

The conditions that are checked before inserting the elements :

If the front and rear are in adjacent locations(i.e. rear following front)the message 'Queue is full' is displayed.

If the value of front is -1 then it denotes that the queue is empty and that the element to be added would be the first element in the queue . The value of front and rear in such a case are set to 0 and new element gets placed at 0Th position.

ADDING ELEMENTS INTO CIRCULAR QUEUE

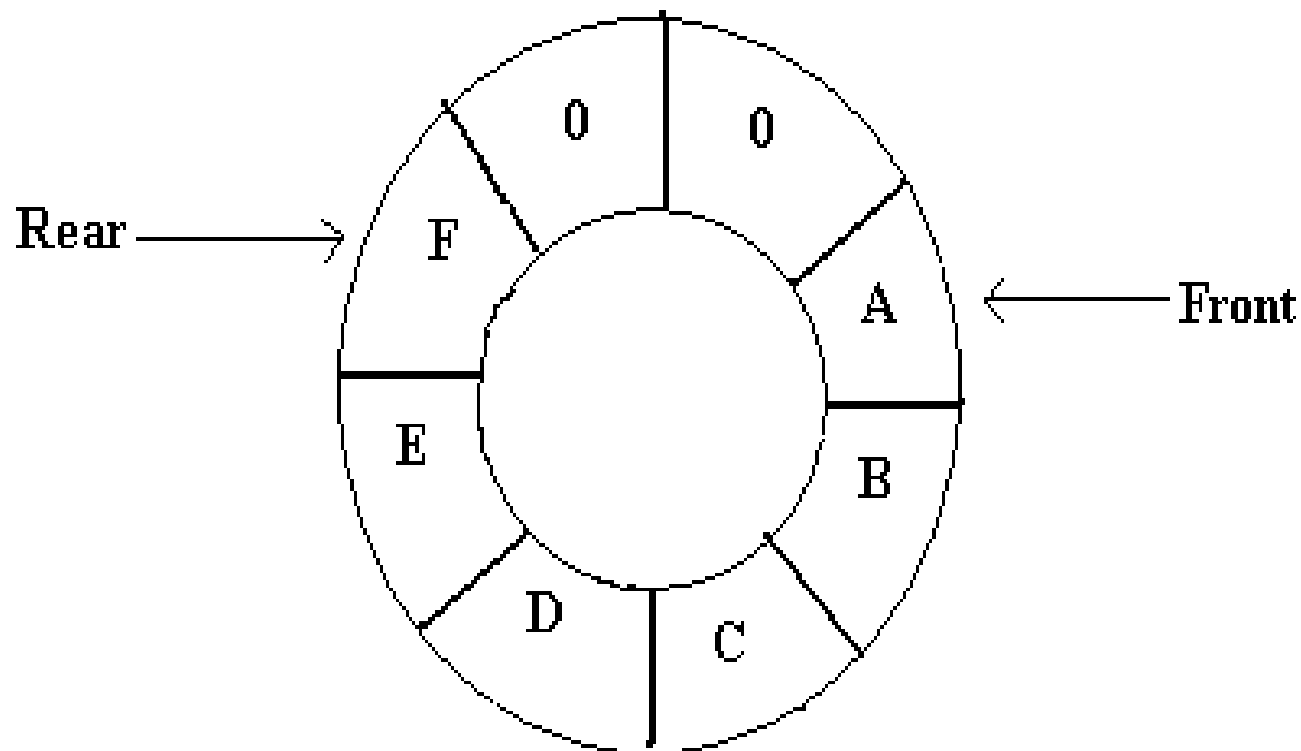
Some of the positions at the front end of the array might be empty .

This happens if we have deleted some elements from the queue , when the value of rear is MAX-1 and the value of front is greater than 0.

In such a case value of rear is set to 0 and the element to be added is added to this position.

The element is added at the rear position in case the value of front is either equal to or greater than 0 and the value of rear is less than MAX-1.

ADDING ELEMENTS IN CIRCULAR QUEUE



Circular queue after adding 6 elements

DELETING ELEMENTS INTO CIRCULAR QUEUE

The conditions that are checked before deleting the elements :

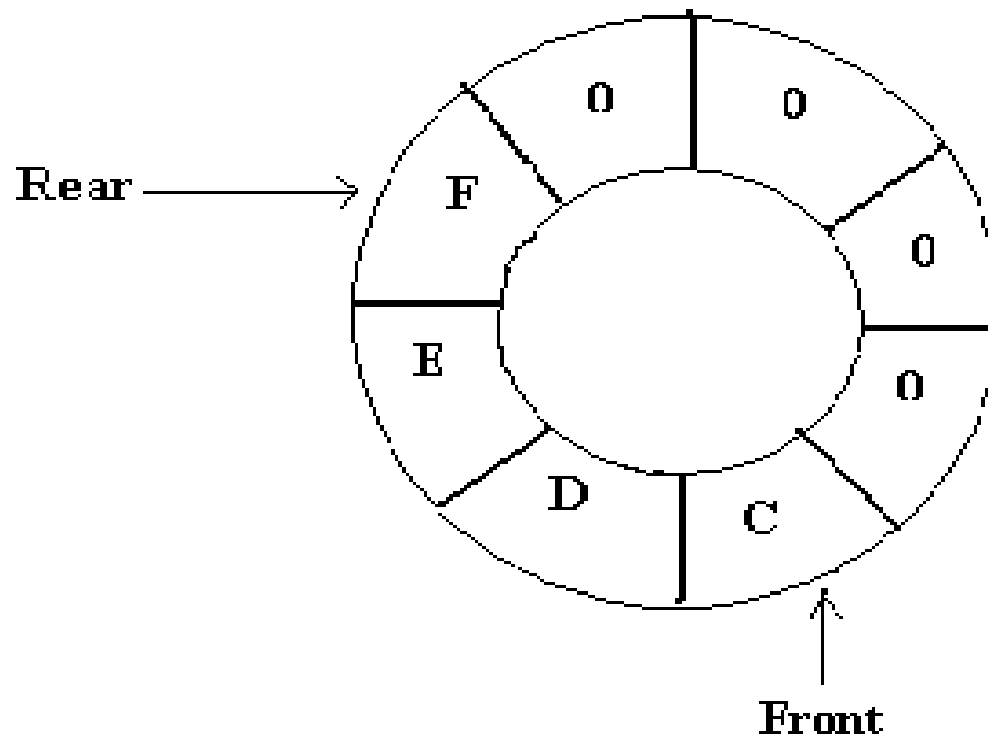
First it is checked whether the queue is empty or not . The elements at the front position will be deleted.

Now , it is checked if the value of front is equal to rear . If it is, then the element which will be deleted is the only element in the queue .

If the element is removed, the queue will be empty and front and rear are set to -1.

DELETING ELEMENTS IN CIRCULAR QUEUE

On Deleting an element from the queue the value of front is set to 0 if it is equal to MAX-1 otherwise front is simply incremented by 1.



Circular queue After deleting 2 elements

DOUBLE ENDED QUEUE

A deque is a linear list in which elements can be added or removed at either end but not in the middle.

There are two variations of a deque an input restricted deque and an output restricted deque which are intermediate between deque and a regular queue.

An input restricted deque is a deque which allows insertions at only one end of the list , but allows deletions at both ends of the list

DOUBLE ENDED QUEUE

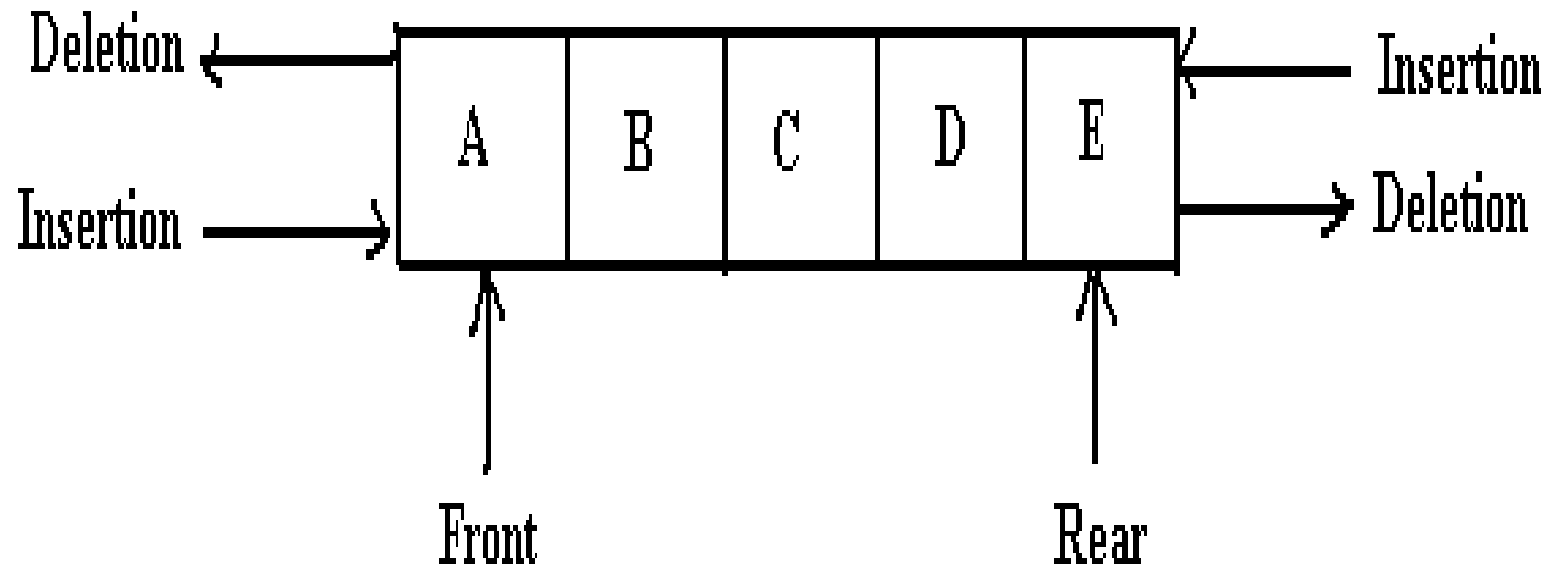
The output restricted deque is a deque which allows deletions at only one end of the list but allows insertions at both ends of the list.

The two possibilities that must consider while inserting /deleting elements into the queue are:

When an attempt is made to insert an element into a deque which is already full, an overflow occurs.

When an attempt is made to delete an element from a deque which is empty, underflow occurs.

REPRESENTATION OF DEQUE



Representation of a deque

UNIT - 3

Trees

Definition of Tree

- A tree 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 $n \geq 0$ disjoint sets T_1, \dots, T_n , where each of these sets is a tree.
- We call T_1, \dots, T_n the subtrees of the root.

Representation of Tree

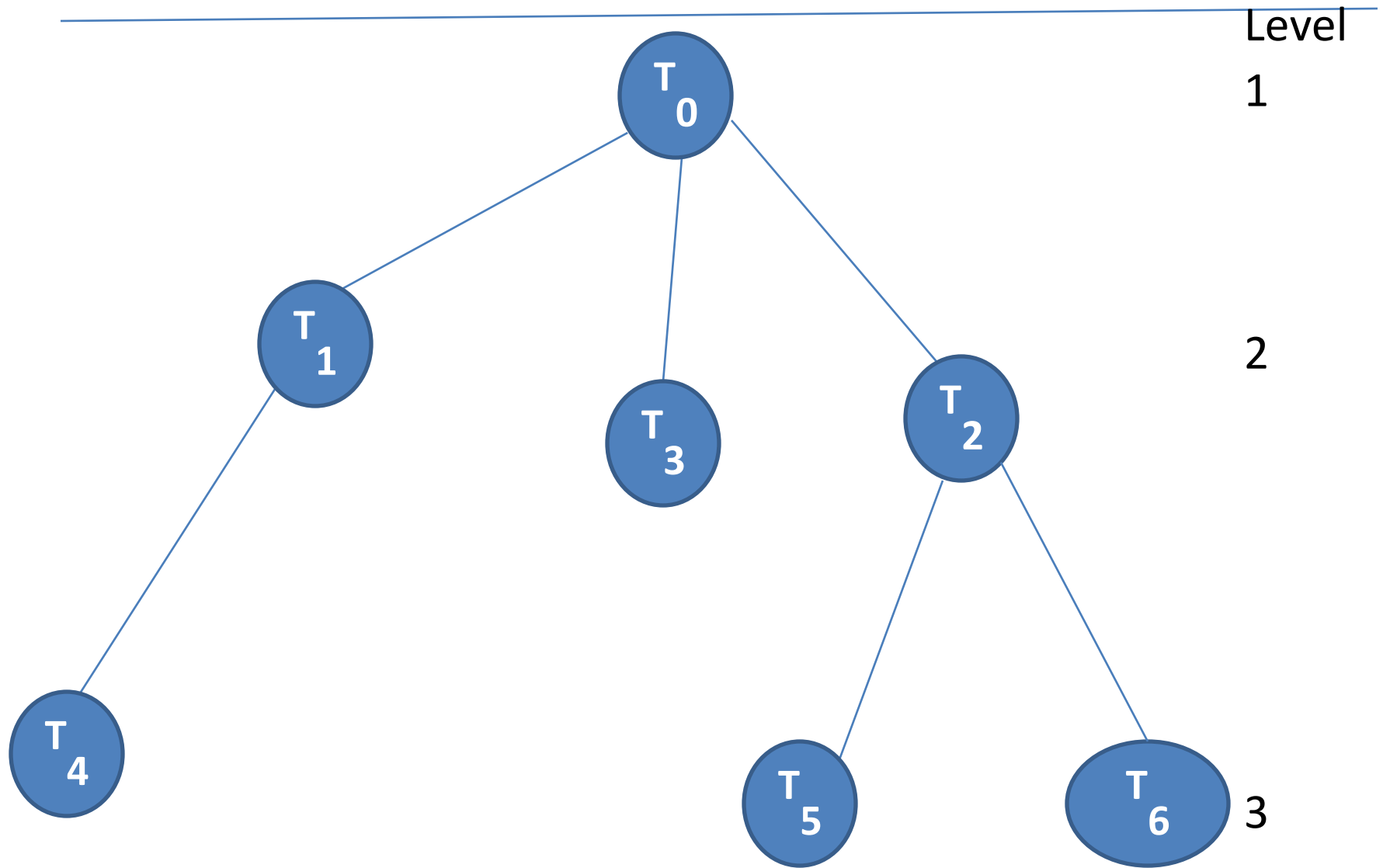


Fig.Tree 1

Terminology

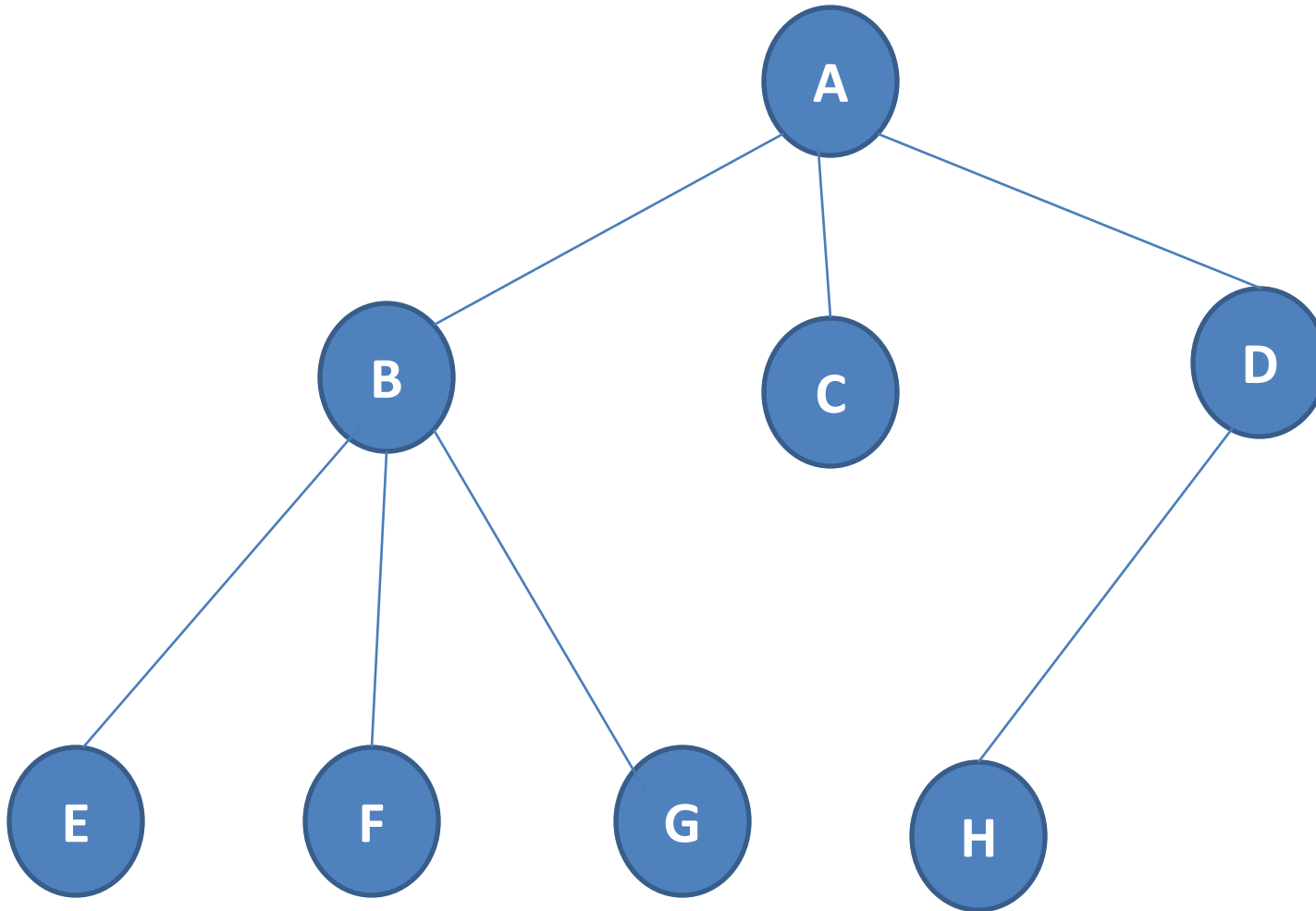


Fig.Tree 2

➤ **ROOT:**

This is the unique node in the tree to which further subtrees are attached. In the above fig node **A** is a root node.

➤ **Degree of the node:**

The total number of sub-trees attached to the node is called the degree of the node.

Node	degree
A	3
E	0

➤ **Leaves:**

These are terminal nodes of the tree. The nodes with degree 0 are always the leaf nodes. In above given tree **E, F, G, C** and **H** are the leaf nodes.

➤ **Internal nodes:**

The nodes other than the root node and the leaves are called the internal nodes. Here **B** and **D** are internal nodes.

Parent nodes:

The node which is having further sub-trees(branches)is called the parent node of those sub-trees. In the given example **node B** is parent node of **E,F and G** nodes.

Predecessor:

While displaying the tree ,if some particular node occurs previous to some other node then that node is called the predecessor of the other node.In above figure E is a predecessor of the node B.

successor:

The node which occurs next to some other node is a successor node.In above figure B is successor of F and G.

Level of the tree:

The root node is always considered at level 0,then its adjacent children are supposed to be at level 1 and so on.In above figure the node A is at level 0,the nodes B,C,D are at level 1,the nodes E,F,G,H are at level 2.

Height of the tree:

The maximum level is the height of the tree. Here height of the tree is 3. The height of the tree is also called depth of the tree.

Degree of tree:

The maximum degree of the node is called the degree of the tree.

The degree of a node is the number of subtrees of the node

– The degree of A is 3; the degree of C is 1.

- The node with degree 0 is a leaf or terminal node.
- A node that has subtrees is the *parent* of the roots of the subtrees.
- The roots of these subtrees are the *children* of the node.
- Children of the same parent are *siblings*.
- The *ancestors* of a node are all the nodes along the path from the root to the node.

Binary Trees

- 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
- The left subtree and the right subtree are distinguished.

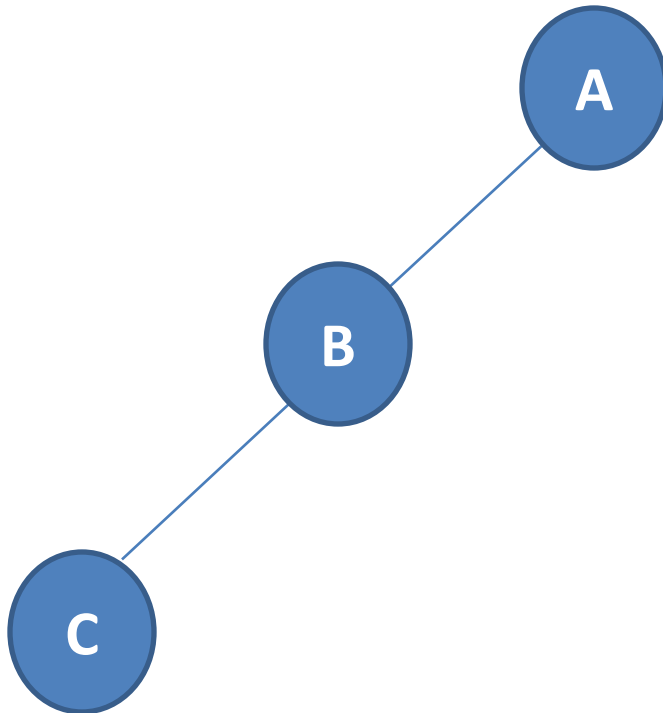
Types Of Binary Trees

There are three types of binary trees

- Left skewed binary tree
- Right skewed binary tree
- Complete binary tree

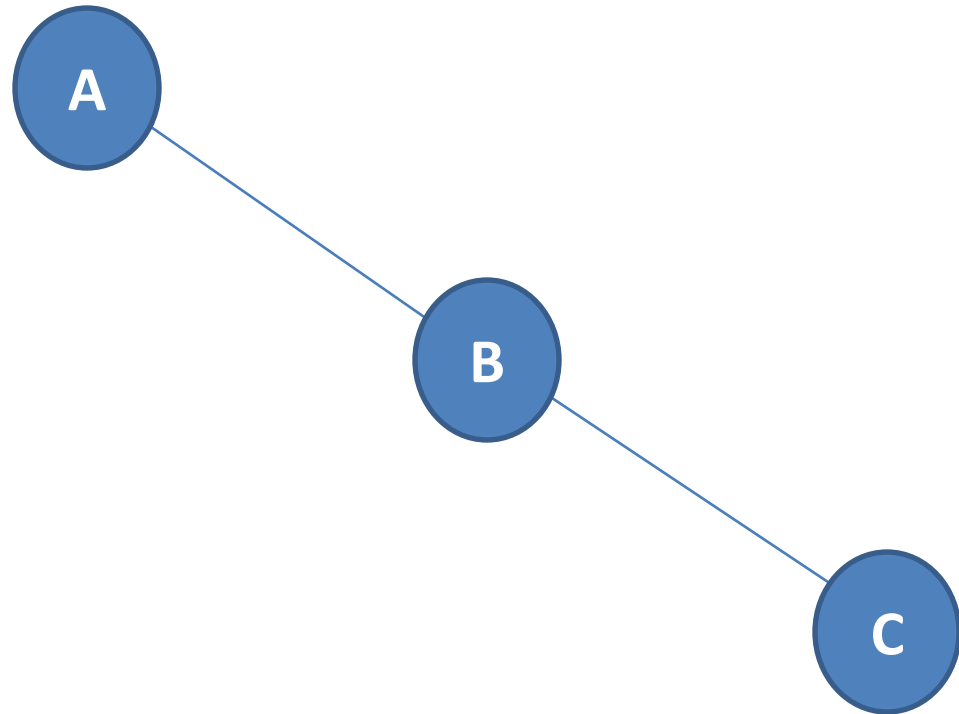
Left skewed binary tree

- If the right subtree is missing in every node of a tree we call it as left skewed tree.



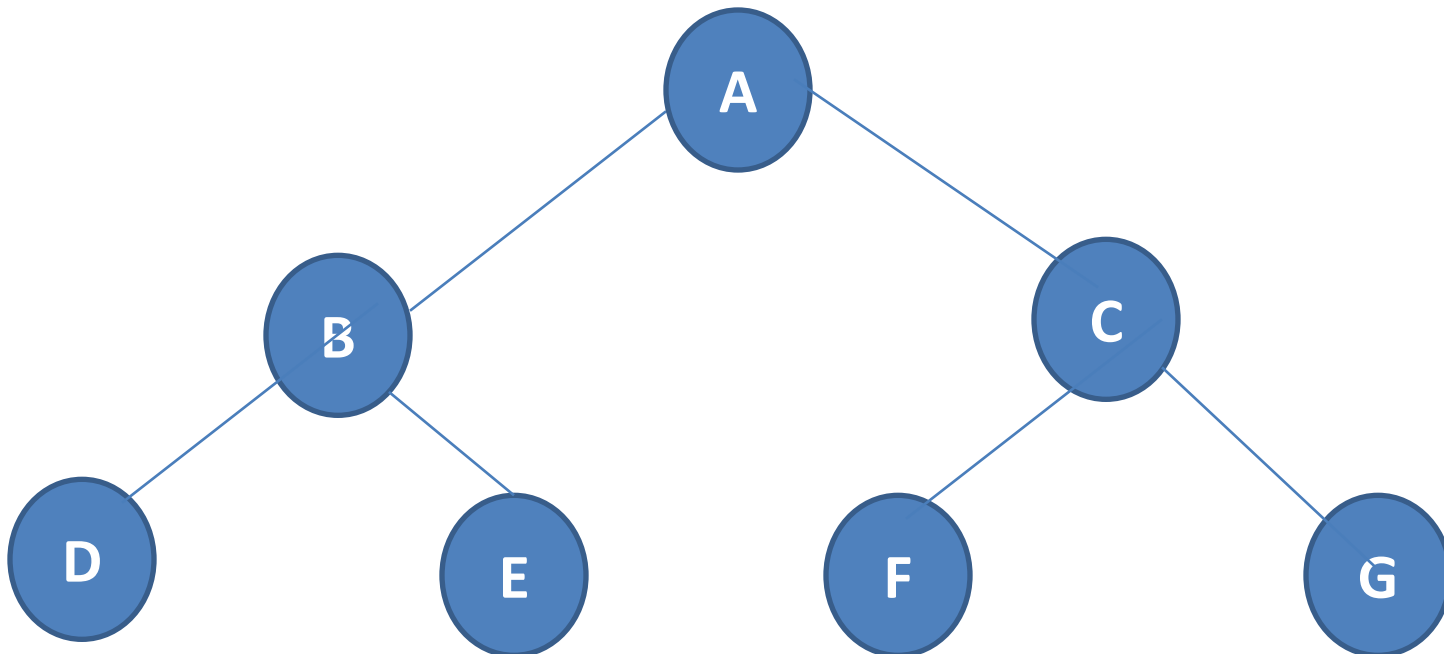
Right skewed binary tree

- If the left subtree is missing in every node of a tree we call it as right subtree.



Complete binary tree

- The tree in which degree of each node is at the most two is called a complete binary tree. In a complete binary tree there is exactly one node at level 0, two nodes at level 1 and four nodes at level 2 and so on. So we can say that a complete binary tree of depth d will contain exactly 2^l nodes at each level l , where l is from 0 to d .



Abstract Data Type Binary_Tree

structure *Binary_Tree*(abbreviated *BinTree*) is
objects: a finite set of nodes either empty or
consisting of a root node, left *Binary_Tree*,
and right *Binary_Tree*.

functions:

for all $bt, bt1, bt2 \in BinTree, item \in element$
Bintree Create() ::= creates an empty binary tree
Boolean IsEmpty(bt) ::= if ($bt == \text{empty binary tree}$) return *TRUE* else return *FALSE*

BinTree MakeBT(*bt1*, *item*, *bt2*) ::= return a binary tree

whose left subtree is *bt1*, whose right subtree is *bt2*,

and whose root node contains the data *item*

Bintree Lchild(*bt*) ::= if (IsEmpty(*bt*)) return error
else return the left subtree of *bt*

element Data(*bt*) ::= if (IsEmpty(*bt*)) return error
else return the data in the root node of *bt*

Bintree Rchild(*bt*) ::= if (IsEmpty(*bt*)) return error
else return the right subtree of *bt*

Maximum Number of Nodes in BT

- The maximum number of nodes on level i of a binary tree is 2^{i-1} , $i \geq 1$.
- The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.

Prove by induction.

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

Binary Tree Representation

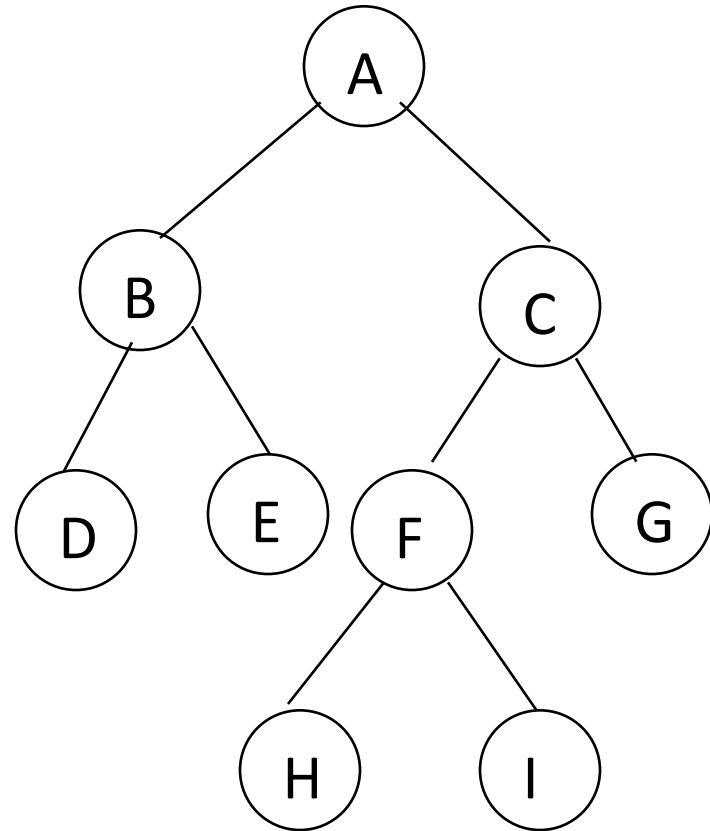
- Sequential (Arrays) representation
- Linked representation

Array Representation Of Binary Tree

This representation uses only a single linear array tree as follows:

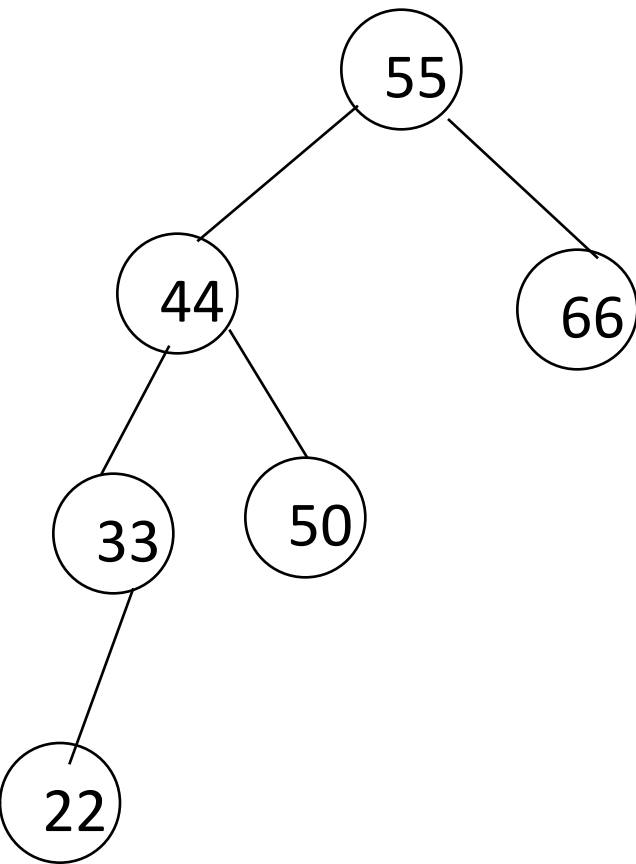
- i) The root of the tree is stored in $\text{tree}[0]$.
- ii) if a node occupies $\text{tree}[i]$, then its left child is stored in $\text{tree}[2*i+1]$, its right child is stored in $\text{tree}[2*i+2]$, and the parent is stored in $\text{tree}[(i-1)/2]$.

Sequential Representation



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

Sequential Representation



[0]	55
[1]	44
[2]	66
[3]	33
[4]	50
[5]	
[6]	
[7]	22
[8]	

Advantages of sequential representation

The only advantage with this type of representation is that the direct access to any node can be possible and finding the parent or left right children of any particular node is fast because of the random access.

Disadvantages of sequential representation

- The major disadvantage with this type of representation is wastage of memory.
- The maximum depth of the tree has to be fixed.
- The insertions and deletion of any node in the tree will be costlier as other nodes has to be adjusted at appropriate positions so that the meaning of binary tree can be preserved.

Linked Representation

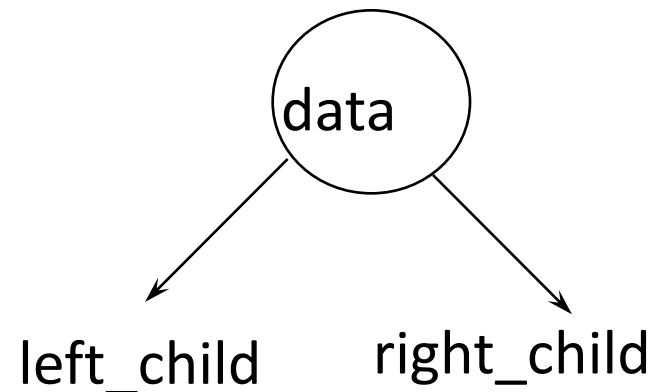
```
struct node
```

```
{
```

```
int data;
```

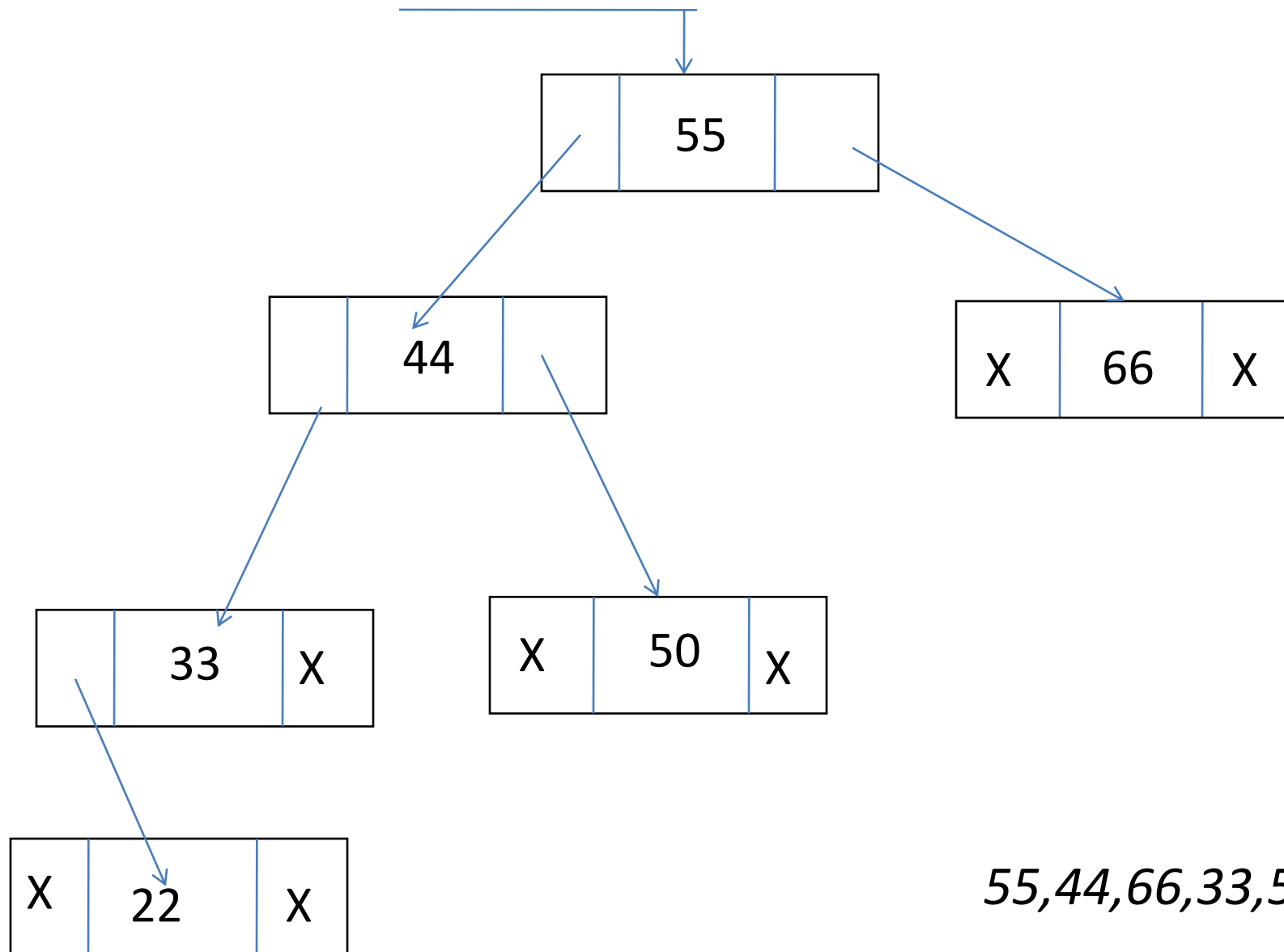
```
struct node * left_child, *right_child;
```

```
};
```



Linked Representation

root



55,44,66,33,50,22

Advantages of Linked representation

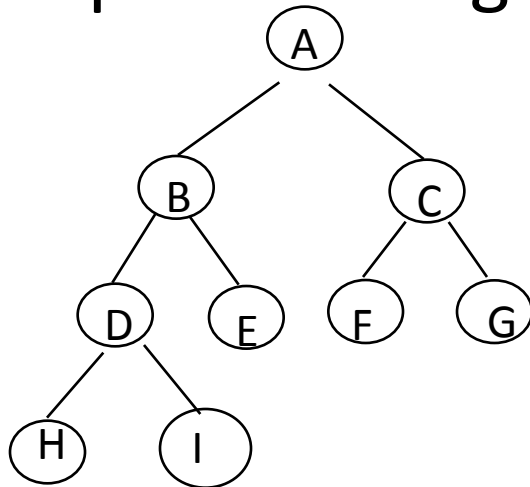
- This representation is superior to our representation as there is no wastage of memory.
- Insertions and deletions which are the most common operations can be done without moving the other nodes.

Disadvantages of linked representation

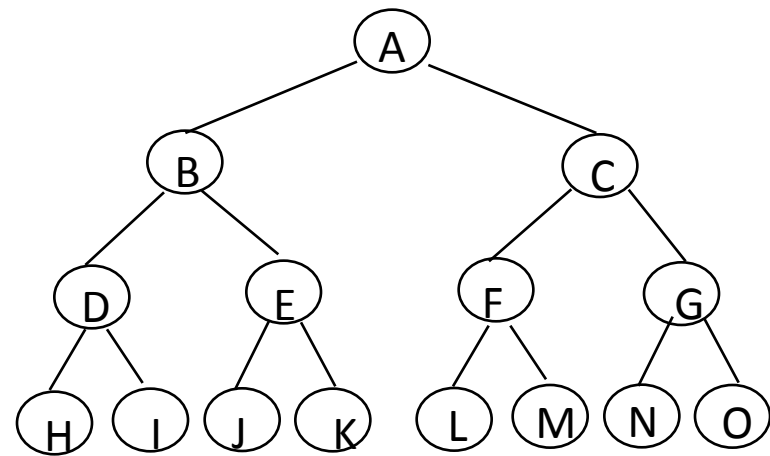
- This representation does not provide direct access to a node and special algorithms are required.
- This representation needs additional space in each node for storing the left and right subtrees.

Full BT VS Complete BT

- A binary tree with n nodes and depth k is complete *iff* its n nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k .
- A full binary tree of depth k is a binary tree of depth k having $2^{k+1} - 1$ nodes, $k \geq 0$.



Complete binary tree



Full binary tree of depth 4

Binary Tree Traversals

The process of going through a tree in such a way that each node is visited once is tree traversal. Several methods are used for tree traversal. The traversal in a binary tree involves three kinds of basic activities such as:

Visiting the root

Traverse left subtree

Traverse right subtree

We will use some notations to traverse a given binary tree as follows:

L means move to the Left child.

R means move to the Right child.

D means the root/parent node.

The only difference among the methods is the order in which these three operations are performed.

There are three standard ways of traversing a non empty binary tree namely :

Preorder

Inorder

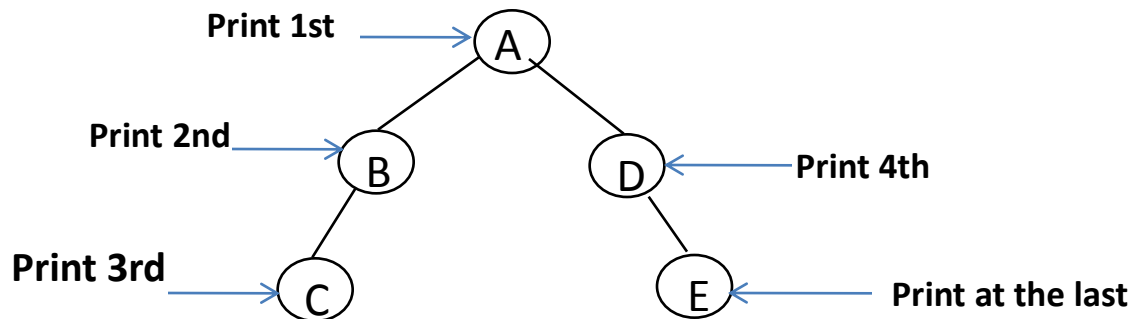
Postorder

Preorder(also known as depth-first order)

1.Visit the root(D)

2.Traverse the left subtree in preorder(L)

3.Traverse the right subtree in preorder(R)



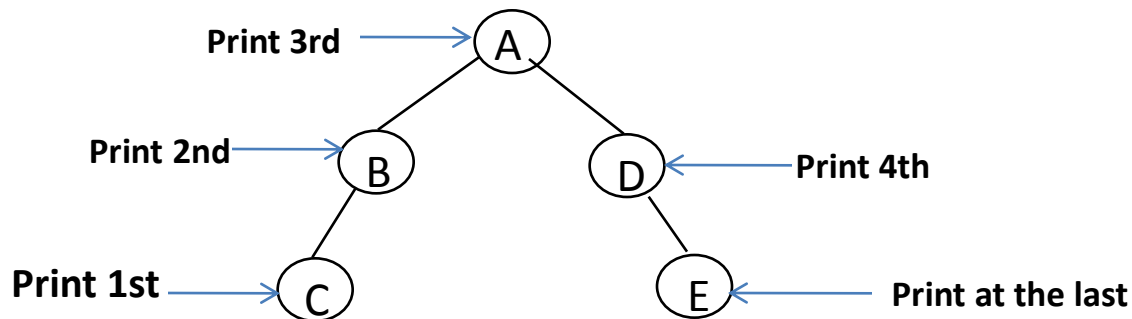
A-B-C-D-E is the preorder traversal of the above figure.

Inorder(also known as symmetric order)

1.Traverse the left subtree in Inorder(L)

2.Visit the root(D)

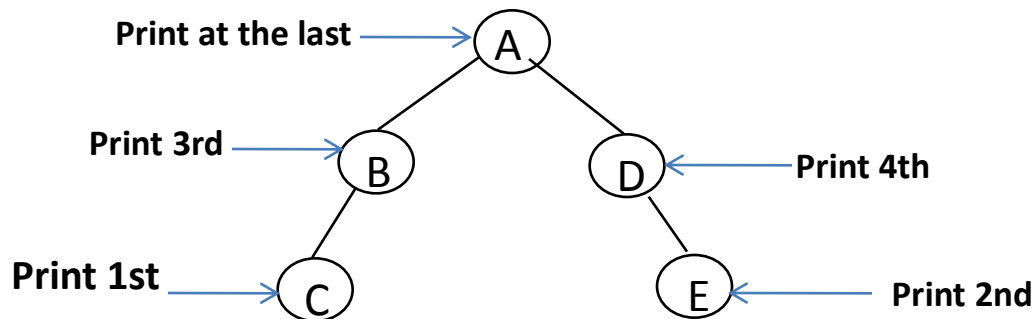
3.Traverse the right subtree in Inorder(R)



C-B-A-D-E is the Inorder traversal of the above figure.

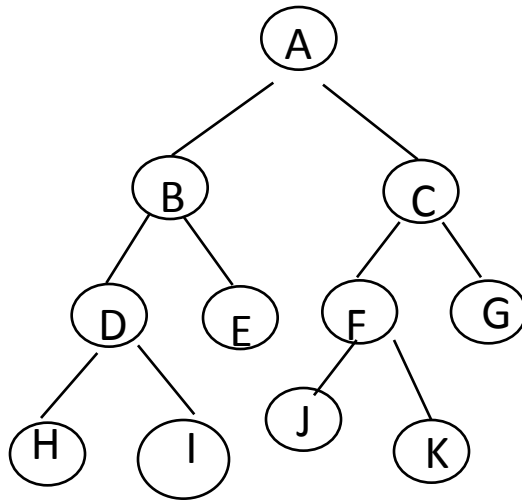
Postorder

1. Traverse the left subtree in postorder(L)
2. Traverse the right subtree in postorder(R)
3. Visit the root(D)



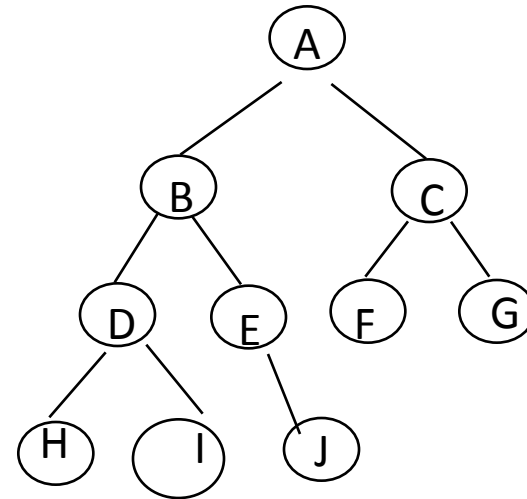
C-D-B-E-A is the postorder traversal of the above figure.

Binary tree traversals



FIG(a)

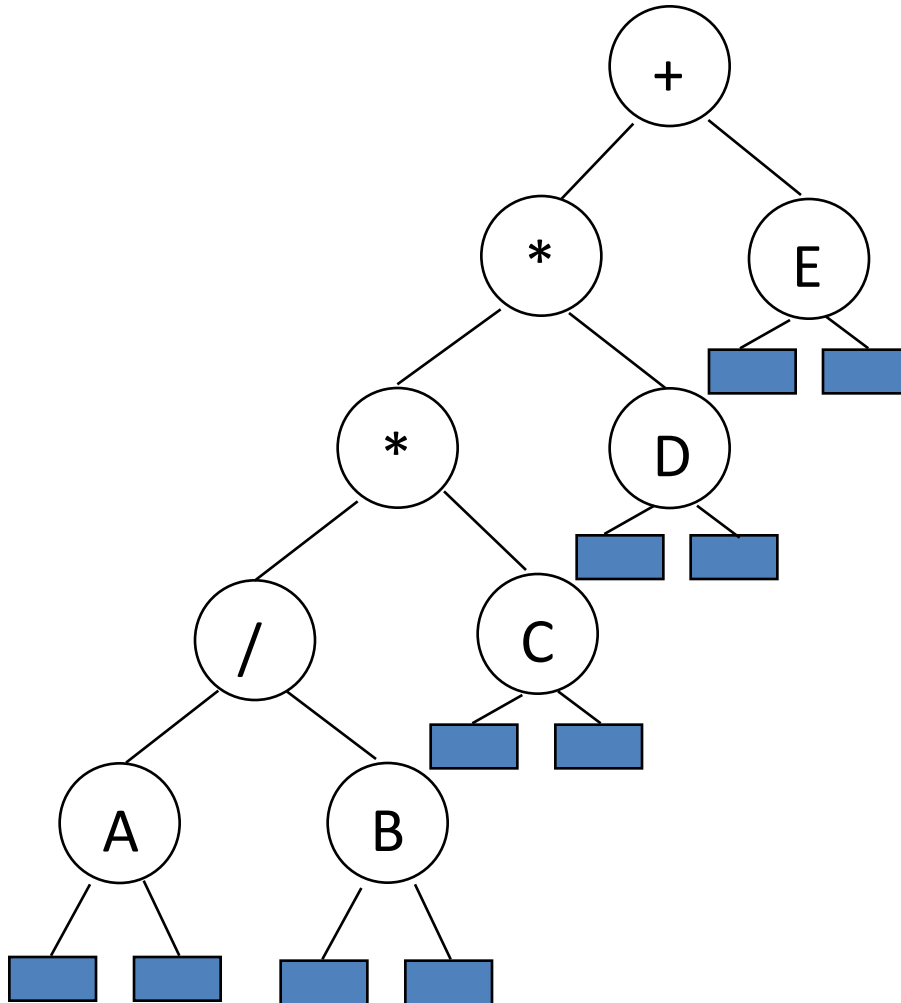
Preorder:ABDHIECFJKG
Inorder:HDIBEAJFKCG
Postorder:HIDEBJKFGCA



FIG(b)

preorder :ABDHIEJCFG
inorder: HDIBJEAFCG
postorder:HIDJEBFGCA

Arithmetic Expression Using BT



inorder traversal

$A / B * C * D + E$

infix expression

preorder traversal

$+ * * / A B C D E$

prefix expression

postorder traversal

$A B / C * D * E +$

postfix expression

level order traversal

$+ * E * D / C A B$

Inorder Traversal (recursive version)

```
void inorder(tree_pointer ptr)
/* inorder tree traversal */
{
    if (ptr) {
        inorder(ptr->left_child);
        printf("%d", ptr->data);
        indorder(ptr->right_child);
    }
}
```

$A / B * C * D + E$

Preorder Traversal (recursive version)

```
void preorder(tree_pointer ptr)
/* preorder tree traversal */
{
    if (ptr) {
        printf("%d", ptr->data);
        preorder(ptr->left_child);
        preorder(ptr->right_child);
    }
}
```

+ * * / A B C D E

Postorder Traversal (recursive version)

```
void postorder(tree_pointer ptr)
/* postorder tree traversal */
{
    if (ptr) {
        postorder(ptr->left_child);
        postorder(ptr->right_child);
        printf("%d", ptr->data);
    }
}
```

$A B / C * D * E +$

Iterative Inorder Traversal

(using stack)

```
void iter_inorder(tree_pointer node)
{
    int top= -1; /* initialize stack */
    tree_pointer stack[MAX_STACK_SIZE];
    for (;;) {
        for (; node; node=node->left_child)
            add(&top, node); /* add to stack */
        node= delete(&top);
                        /* delete from stack */
        if (!node) break; /* empty stack */
        printf("%D", node->data);
        node = node->right_child;
    }
}
```

$O(n)$

Trace Operations of Inorder Traversal

Call of inorder	Value in root	Action	Call of inorder	Value in root	Action
1	+		11	C	
2	*		12	NULL	
3	*		11	C	printf
4	/		13	NULL	
5	A		2	*	printf
6	NULL		14	D	
5	A	printf	15	NULL	
7	NULL		14	D	printf
4	/	printf	16	NULL	
8	B		1	+	printf
9	NULL		17	E	
8	B	printf	18	NULL	
10	NULL		17	E	printf
3	*	printf	19	NULL	

Level Order Traversal

(using queue)

```
void level_order(tree_pointer ptr)
/* level order tree traversal */
{
    int front = rear = 0;
    tree_pointer queue[MAX_QUEUE_SIZE];
    if (!ptr) return; /* empty queue */
    addq(front, &rear, ptr);
    for (;;) {
        ptr = deleteq(&front, rear);
```

```

if (ptr) {
    printf("%d", ptr->data);
    if (ptr->left_child)
        addq(front, &rear,
              ptr->left_child);
    if (ptr->right_child)
        addq(front, &rear,
              ptr->right_child);
}
else break;
}
}

```

$+ * E * D / C A B$

Copying Binary Trees

```
tree_pointer copy(tree_pointer original)
{
    tree_pointer temp;
    if (original) {
        temp=(tree_pointer) malloc(sizeof(node));
        if (IS_FULL(temp)) {
            fprintf(stderr, "the memory is full\n");
            exit(1);
        }
        temp->left_child=copy(original->left_child);
        temp->right_child=copy(original->right_child);
        temp->data=original->data;
        return temp;
    }
    return NULL;
}
```

postorder

Post-order-eval function

```
void post_order_eval(tree_pointer node)
{
/* modified post order traversal to evaluate a propositional
   calculus tree */
   if (node) {
      post_order_eval(node->left_child);
      post_order_eval(node->right_child);
      switch(node->data) {
      case not: node->value =
                !node->right_child->value;
                break;
```

```
case and:  node->value =
           node->right_child->value &&
           node->left_child->value;
           break;
case or:   node->value =
           node->right_child->value | |
           node->left_child->value;
           break;
case true: node->value = TRUE;
           break;
case false: node->value = FALSE;
}
}
}
```


Threaded Binary Trees

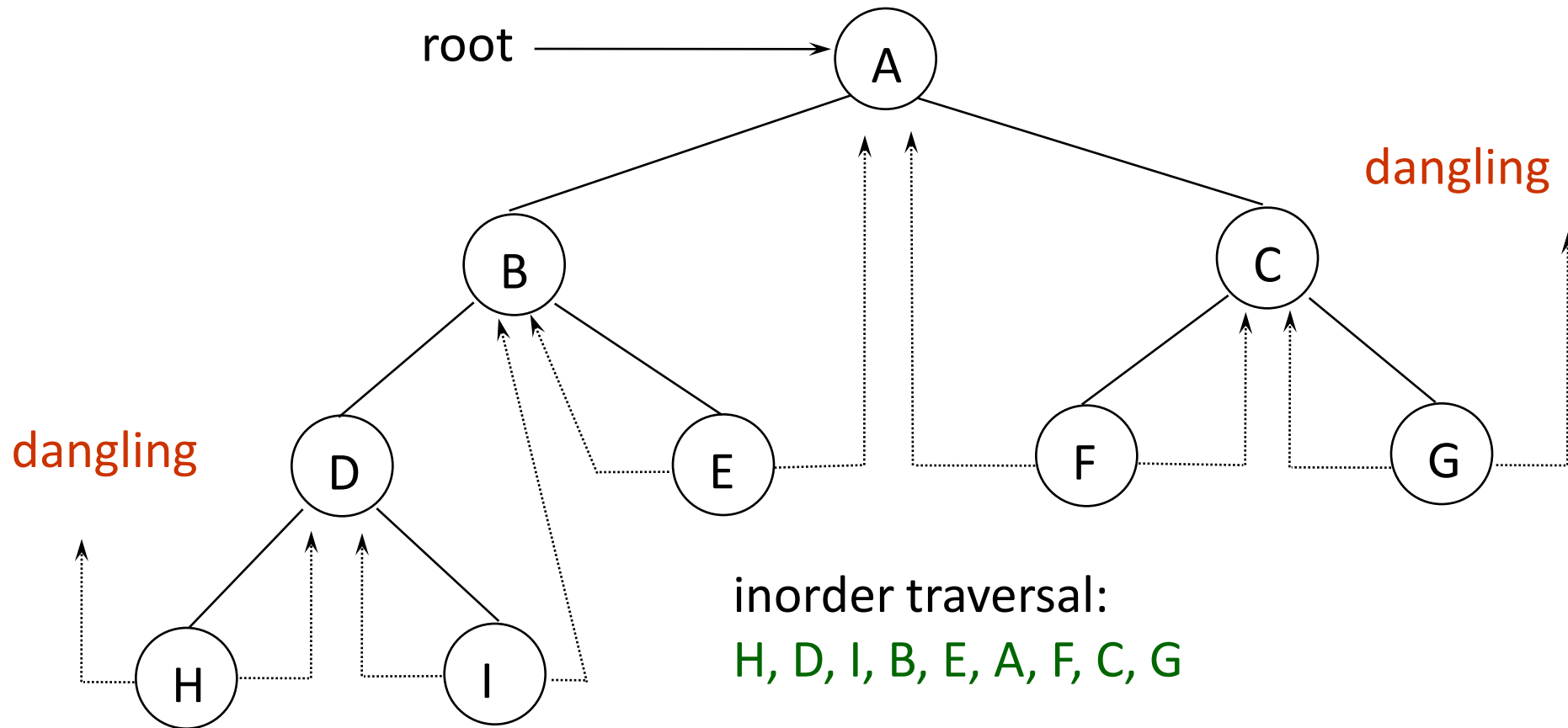
- Too many null pointers in current representation of binary trees
 - n: number of nodes
 - number of non-null links: $n-1$
 - total links: $2n$
 - null links: $2n-(n-1)=n+1$
- Replace these null pointers with some useful “threads”.

Threaded Binary Trees *(Continued)*

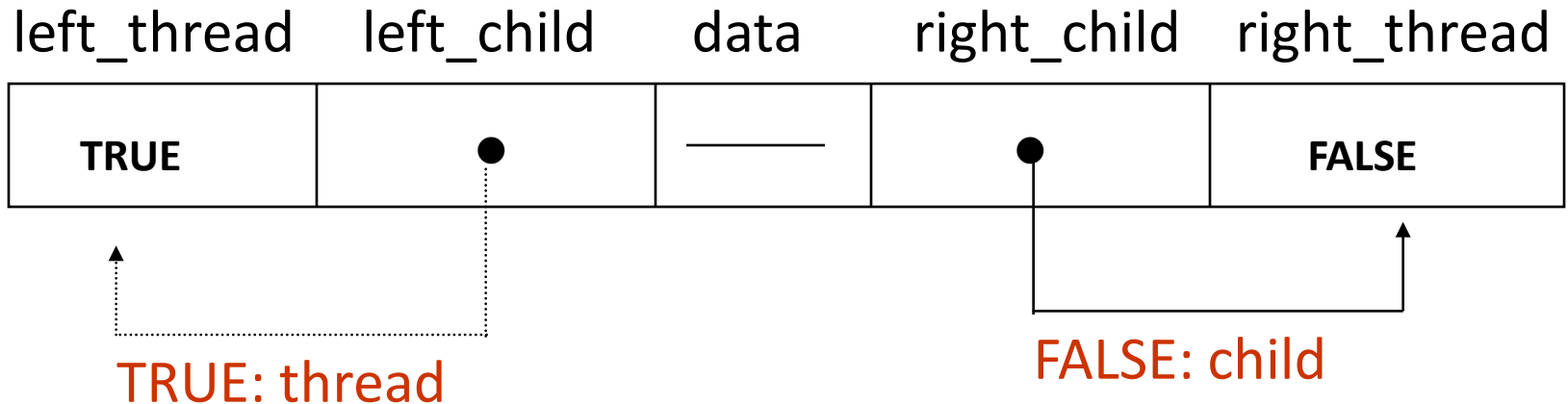
If `ptr->left_child` is null,
replace it with a pointer to the node that would be
visited *before* `ptr` in an *inorder traversal*

If `ptr->right_child` is null,
replace it with a pointer to the node that would be
visited *after* `ptr` in an *inorder traversal*

A Threaded Binary Tree

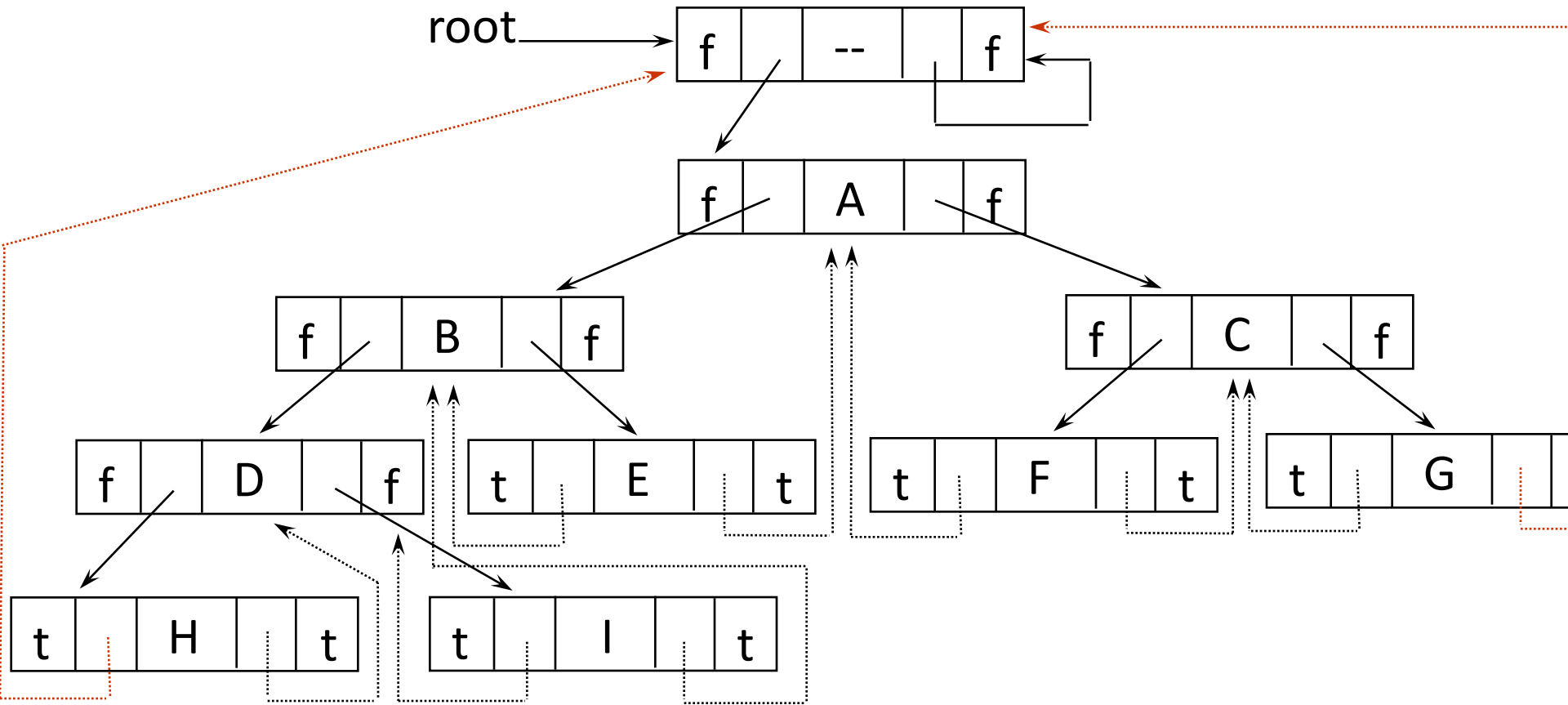


Data Structures for Threaded BT



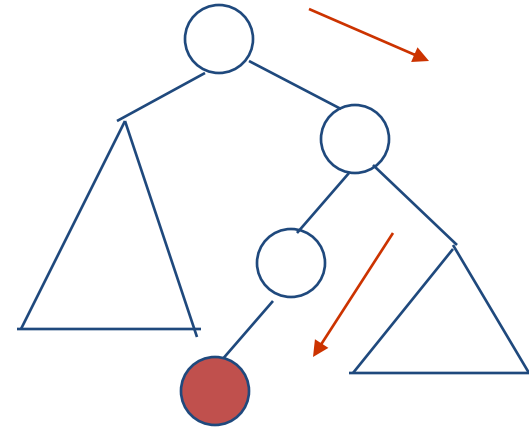
```
typedef struct threaded_tree *threaded_pointer;  
typedef struct threaded_tree {  
    short int left_thread;  
    threaded_pointer left_child;  
    char data;  
    threaded_pointer right_child;  
    short int right_thread; };
```

Memory Representation of A Threaded BT



Next Node in Threaded BT

```
threaded_pointer insucc(threaded_pointer
    tree)
{
    threaded_pointer temp;
    temp = tree->right_child;
    if (!tree->right_thread)
        while (!temp->left_thread)
            temp = temp->left_child;
    return temp;
}
```



Inorder Traversal of Threaded BT

```
void tinorder(threaded_pointer tree)
{
    /* traverse the threaded binary tree
       inorder */
    threaded_pointer temp = tree;
    for (;;) {
        temp = insucc(temp);
        if (temp==tree) break;
        printf("%3c", temp->data);
    }
}
```

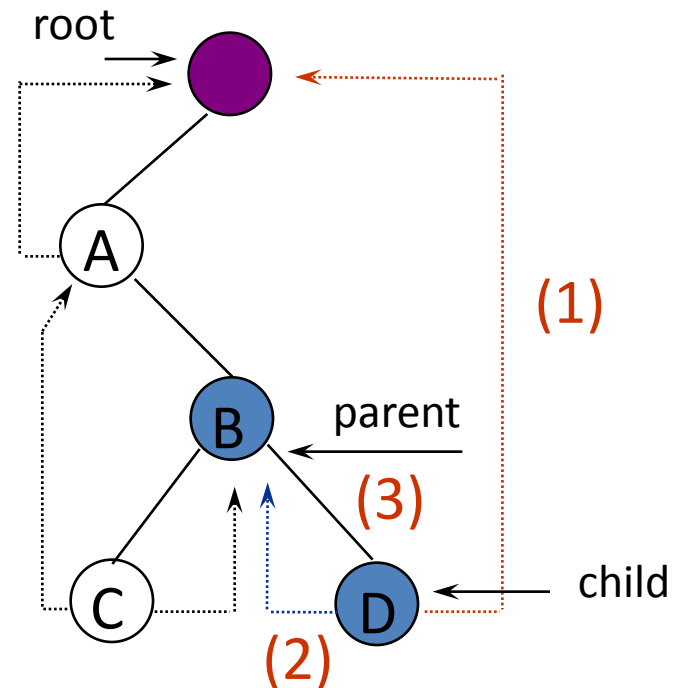
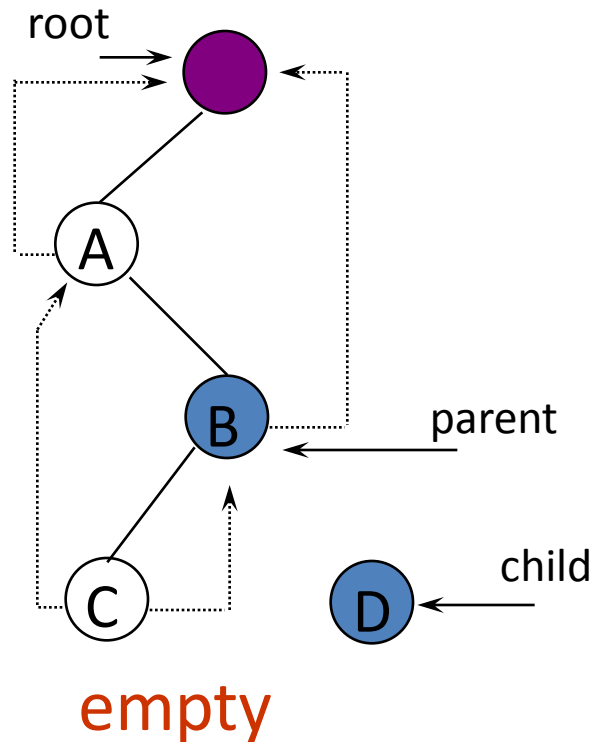
$O(n)$ (timecomplexity)

Inserting Nodes into Threaded BTs

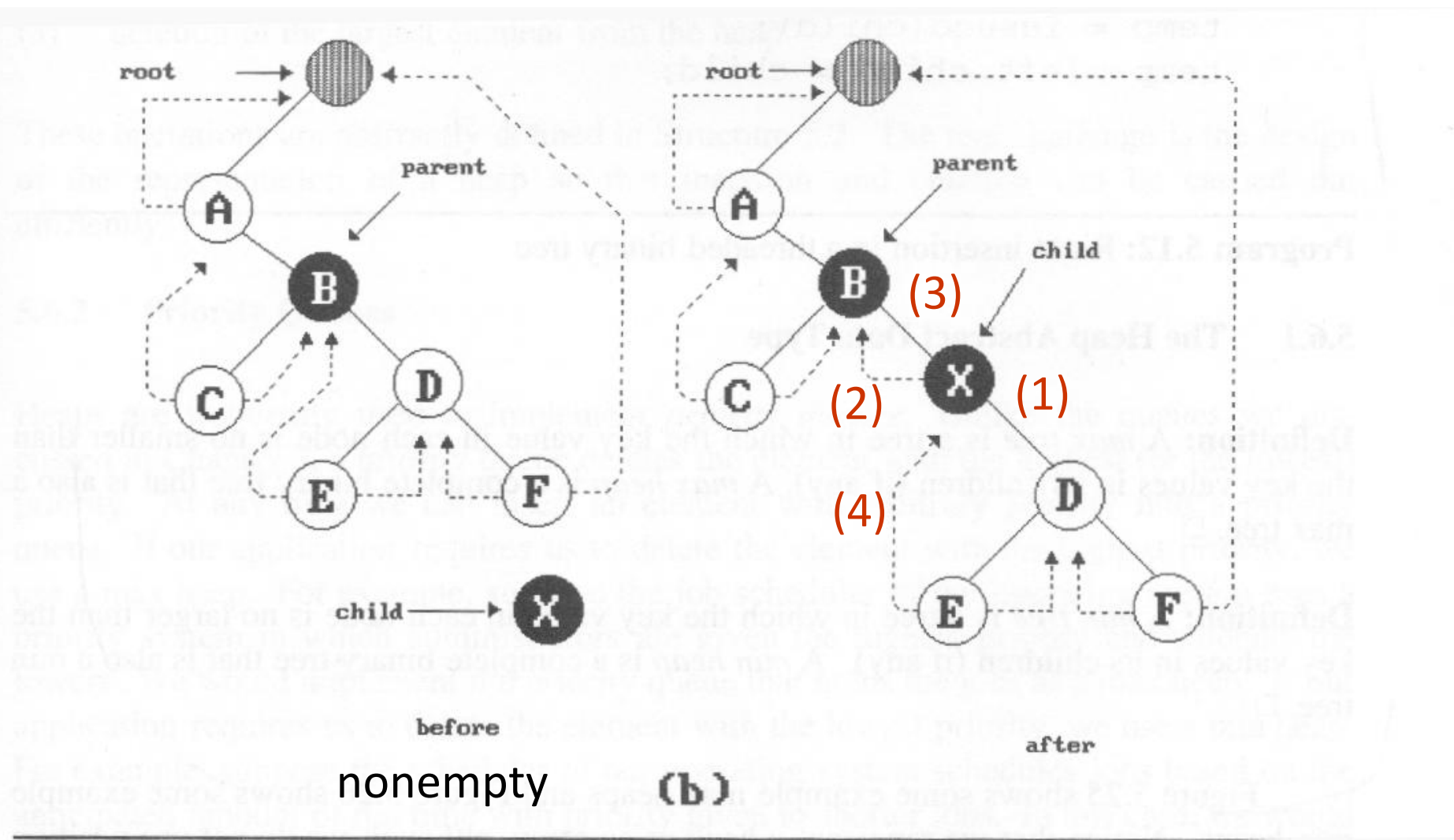
- Insert `child` as the right child of node `parent`
 - **change** `parent->right_thread` to **FALSE**
 - **set** `child->left_thread` **and** `child->right_thread` to **TRUE**
 - **set** `child->left_child` to **point to** `parent`
 - **set** `child->right_child` to `parent->right_child`
 - **change** `parent->right_child` to **point to** `child`

Examples

Insert a node D as a right child of B.



***Figure 5.24:** Insertion of child as a right child of parent in a threaded binary tree (p.217)



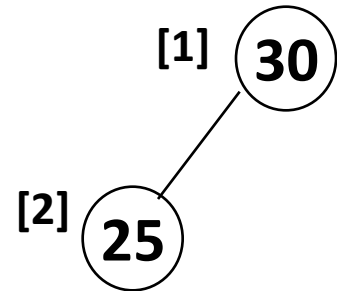
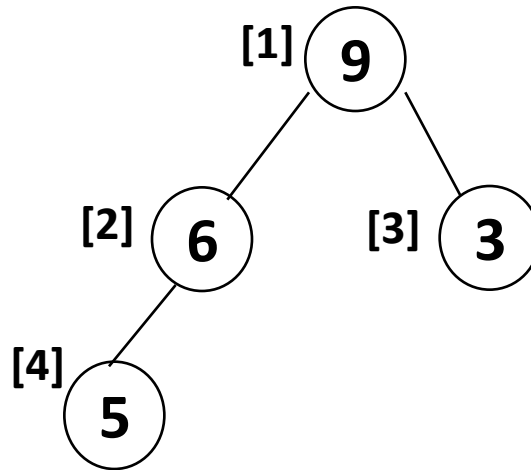
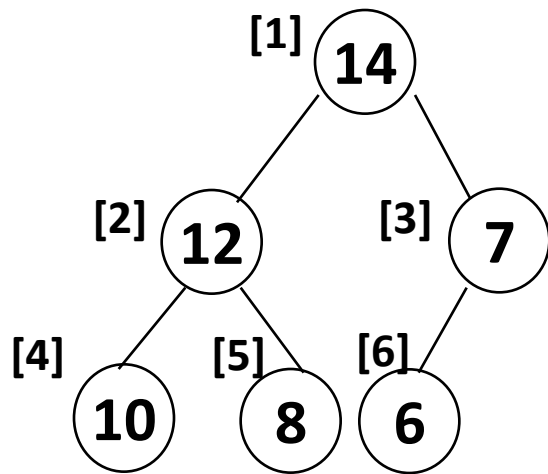
Right Insertion in Threaded BTs

```
void insert_right(threaded_pointer parent,
                  threaded_pointer child)
{
    threaded_pointer temp;
    (1) child->right_child = parent->right_child;
    child->right_thread = parent->right_thread;
    (2) child->left_child = parent;    case (a)
    child->left_thread = TRUE;
    (3) parent->right_child = child;
    parent->right_thread = FALSE;
    if (!child->right_thread) { case (b)
    (4) temp = insucc(child);
        temp->left_child = child;
    }
}
```

Heap

- A *max tree* is a tree in which the key value in each node is **no smaller than** the key values in its children. A *max heap* is a **complete binary tree** that is also a max tree.
- A *min tree* is a tree in which the key value in each node is **no larger than** the key values in its children. A *min heap* is a **complete binary tree** that is also a min tree.
- Operations on heaps
 - creation of an empty heap
 - insertion of a new element into the heap;
 - deletion of the largest element from the heap

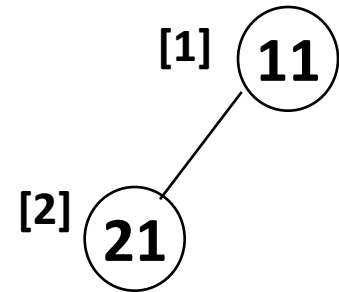
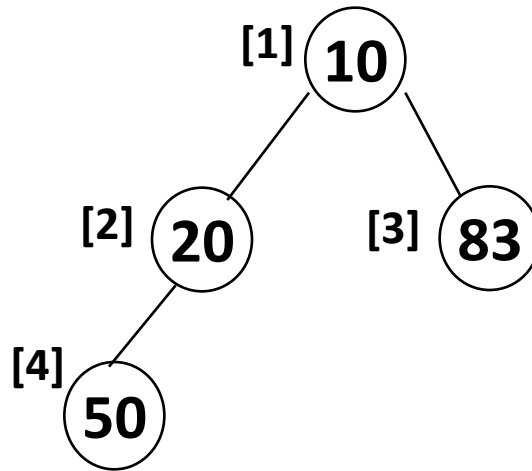
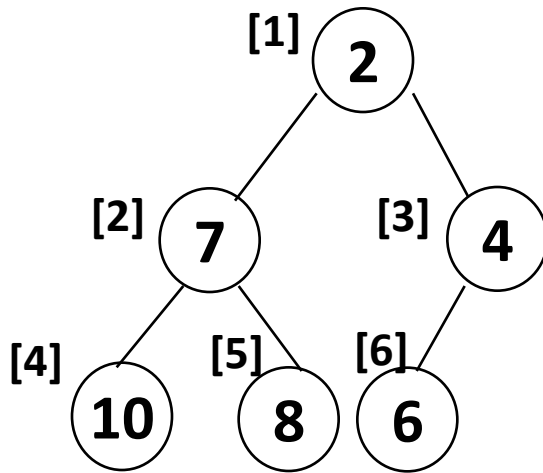
Sample max heaps



Property:

The root of **max heap** contains the **largest** .

Sample min heaps



Property:

The root of **min heap** contains the **smallest**.

ADT for Max Heap

structure MaxHeap

objects: a complete binary tree of $n > 0$ elements organized so that the value in each node is at least as large as those in its children

functions:

for all *heap* belong to *MaxHeap*, *item* belong to *Element*, *n*,
max_size belong to integer

MaxHeap Create(*max_size*)::= create an empty heap that can
hold a maximum of *max_size* elements

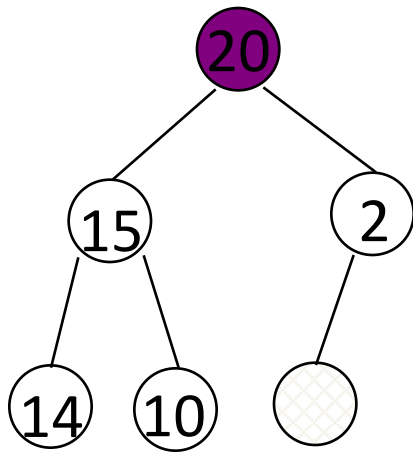
Boolean HeapFull(*heap*, *n*)::= if ($n == \text{max_size}$) return TRUE
else return FALSE

MaxHeap Insert(*heap*, *item*, *n*)::= if ($\neg \text{HeapFull}(\text{heap}, n)$) insert
item into heap and return the resulting heap
else return error

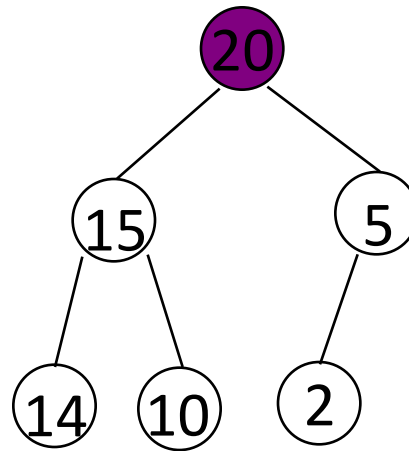
Boolean HeapEmpty(*heap*, *n*)::= if ($n > 0$) return FALSE
else return TRUE

Element Delete(*heap*, *n*)::= if ($\neg \text{HeapEmpty}(\text{heap}, n)$) return one
instance of the **largest** element in the heap
and remove it from the heap
else return error

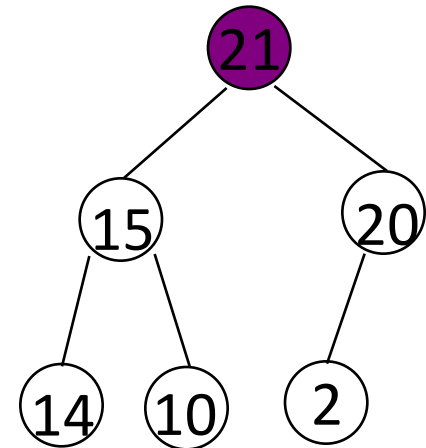
Example of Insertion to Max Heap



initial location of new node



insert 5 into heap



insert 21 into heap

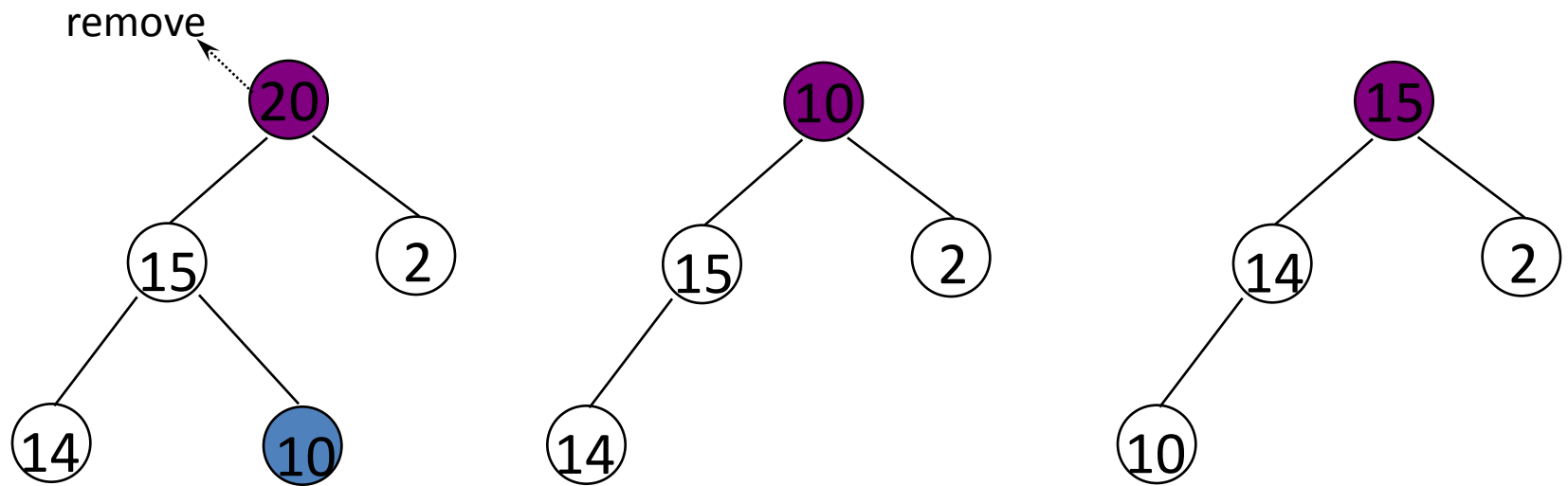
Insertion into a Max Heap

```
void insert_max_heap(element item, int *n)
{
    int i;
    if (HEAP_FULL(*n)) {
        fprintf(stderr, "the heap is full.\n");
        exit(1);
    }
    i = ++(*n);
    while ((i!=1)&&(item.key>heap[i/2].key)) {
        heap[i] = heap[i/2];
        i /= 2;
    }
    heap[i] = item;
}
```

$2^k - 1 = n \implies k = \lceil \log_2(n+1) \rceil$

$O(\log_2 n)$

Example of Deletion from Max Heap



Deletion from a Max Heap

```
element delete_max_heap(int *n)
{
    int parent, child;
    element item, temp;
    if (HEAP_EMPTY(*n)) {
        fprintf(stderr, "The heap is empty\n");
        exit(1);
    }
    /* save value of the element with the
       highest key */
    item = heap[1];
    /* use last element in heap to adjust heap
    temp = heap[(*n)--];
    parent = 1;
    child = 2;
```

```

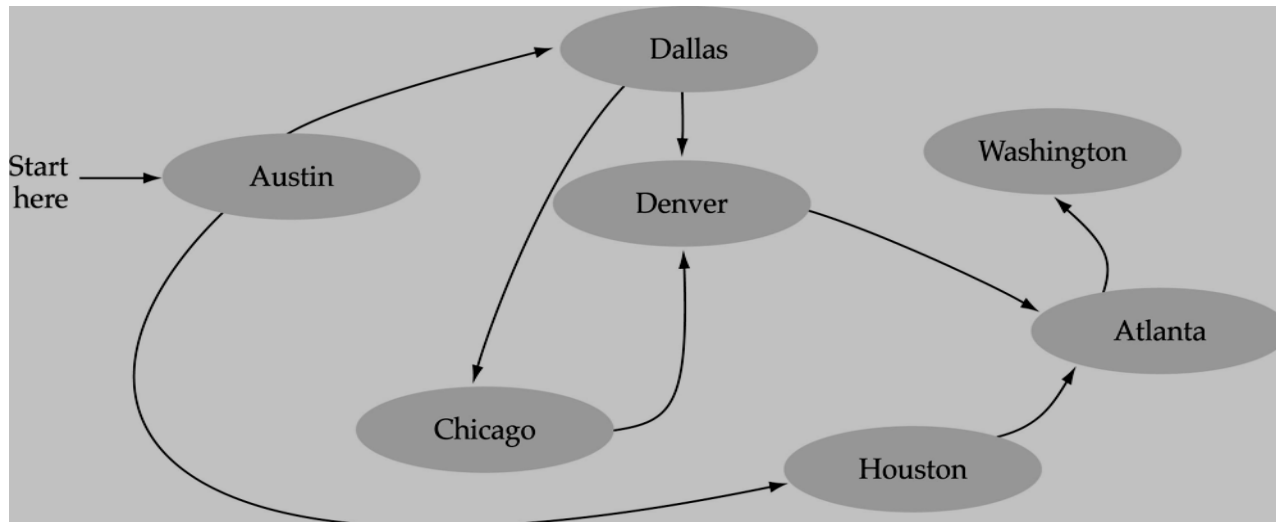
while (child <= *n) {
    /* find the larger child of the current
       parent */
    if ((child < *n) &&
        (heap[child].key < heap[child+1].key))
        child++;
    if (temp.key >= heap[child].key) break;
    /* move to the next lower level */
    heap[parent] = heap[child];
    child *= 2;
}
heap[parent] = temp;
return item;
}

```

Graphs

What is a graph?

- A data structure that consists of a set of nodes (*vertices*) and a set of edges that relate the nodes to each other
- The set of edges describes relationships among the vertices



Formal definition of graphs

- A graph G is defined as follows:

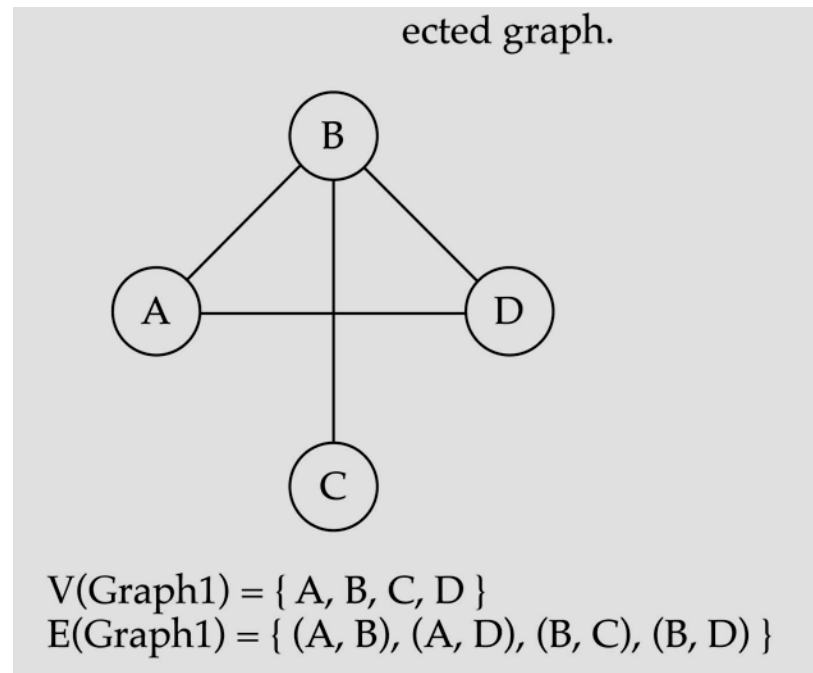
$$G=(V,E)$$

$V(G)$: a finite, nonempty set of vertices

$E(G)$: a set of edges (pairs of vertices)

Directed vs. undirected graphs

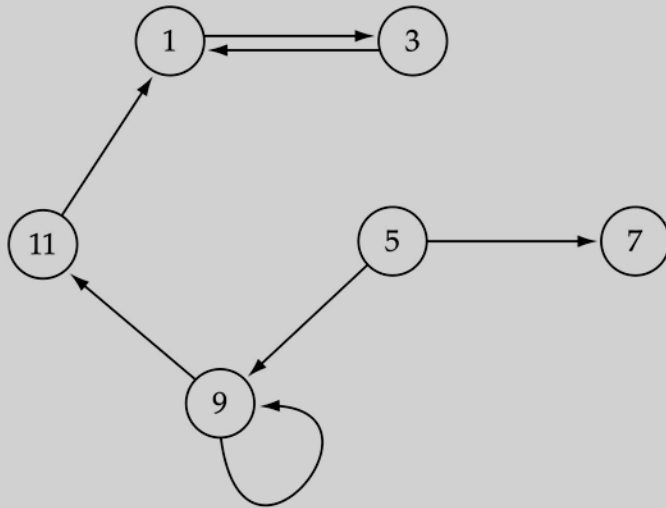
- When the edges in a graph have no direction, the graph is called *undirected*



Directed vs. undirected graphs (cont.)

- When the edges in a graph have a direction, the graph is called *directed* (or *digraph*)

(b) Graph2 is a directed graph.



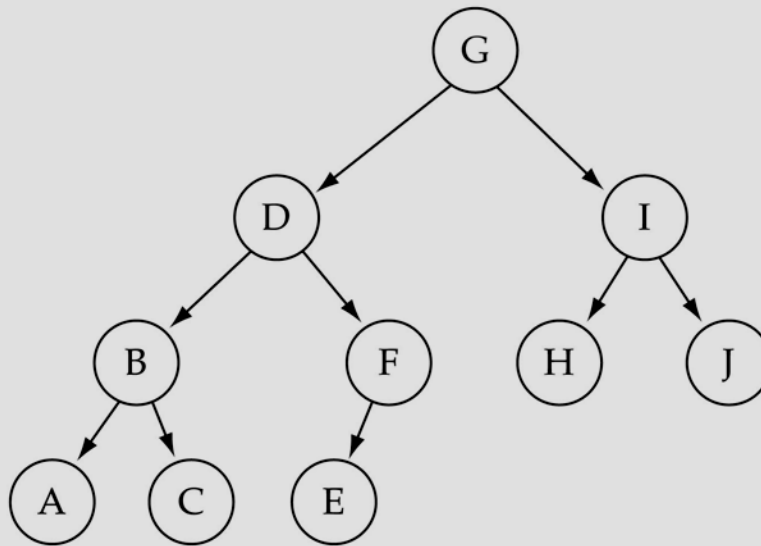
$V(\text{Graph2}) = \{ 1, 3, 5, 7, 9, 11 \}$

$E(\text{Graph2}) = \{(1,3) (3,1) (5,9) (9,11) (5,7) 1), (9, 9), (11, 1) \}$

Trees vs graphs

- Trees are special cases of graphs!!

(c) Graph3 is a directed graph.

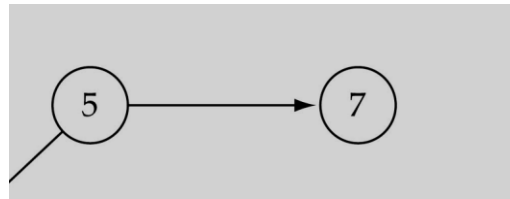


$V(\text{Graph3}) = \{ A, B, C, D, E, F, G, H, I, J \}$

$E(\text{Graph3}) = \{ (G, D), (G, I), (D, B), (D, F), (I, H), (I, J), (B, A), (B, C), (F, E) \}$

Graph terminology

- Adjacent nodes: two nodes are adjacent if they are connected by an edge



5 is adjacent to 7
7 is adjacent from 5

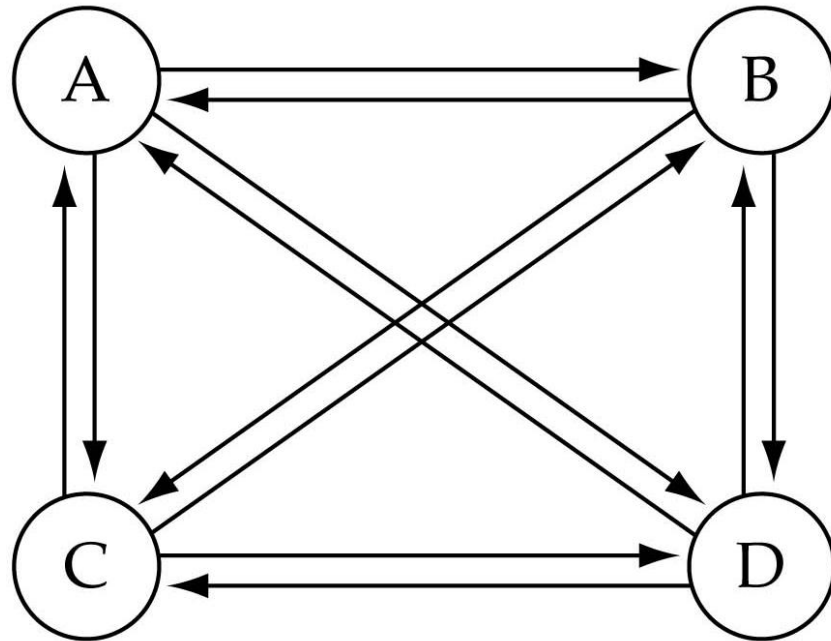
- Path: a sequence of vertices that connect two nodes in a graph
- Complete graph: a graph in which every vertex is directly connected to every other vertex

Graph terminology (cont.)

- What is the number of edges in a complete directed graph with N vertices?

$$N * (N-1)$$

$$O(N^2)$$



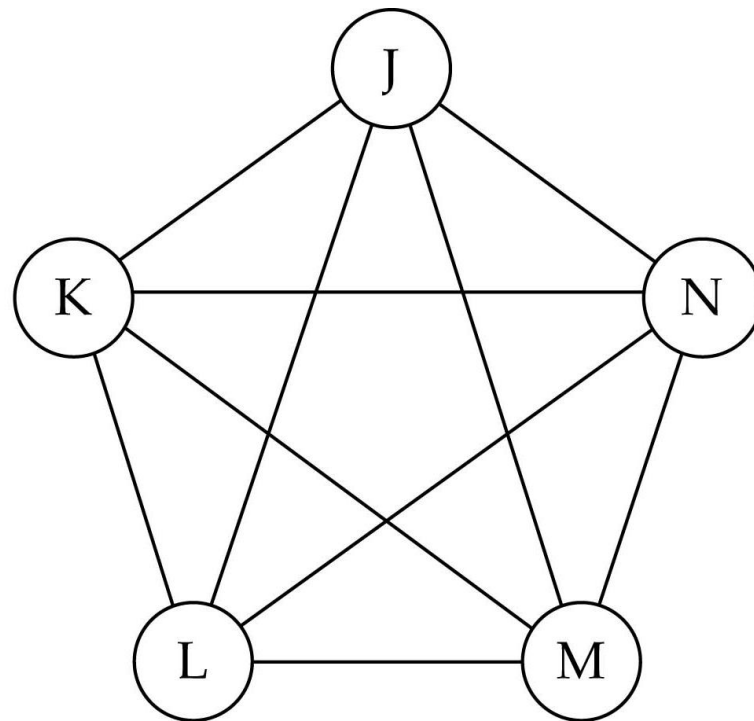
(a) Complete directed graph.

Graph terminology (cont.)

- What is the number of edges in a complete undirected graph with N vertices?

$$N * (N-1) / 2$$

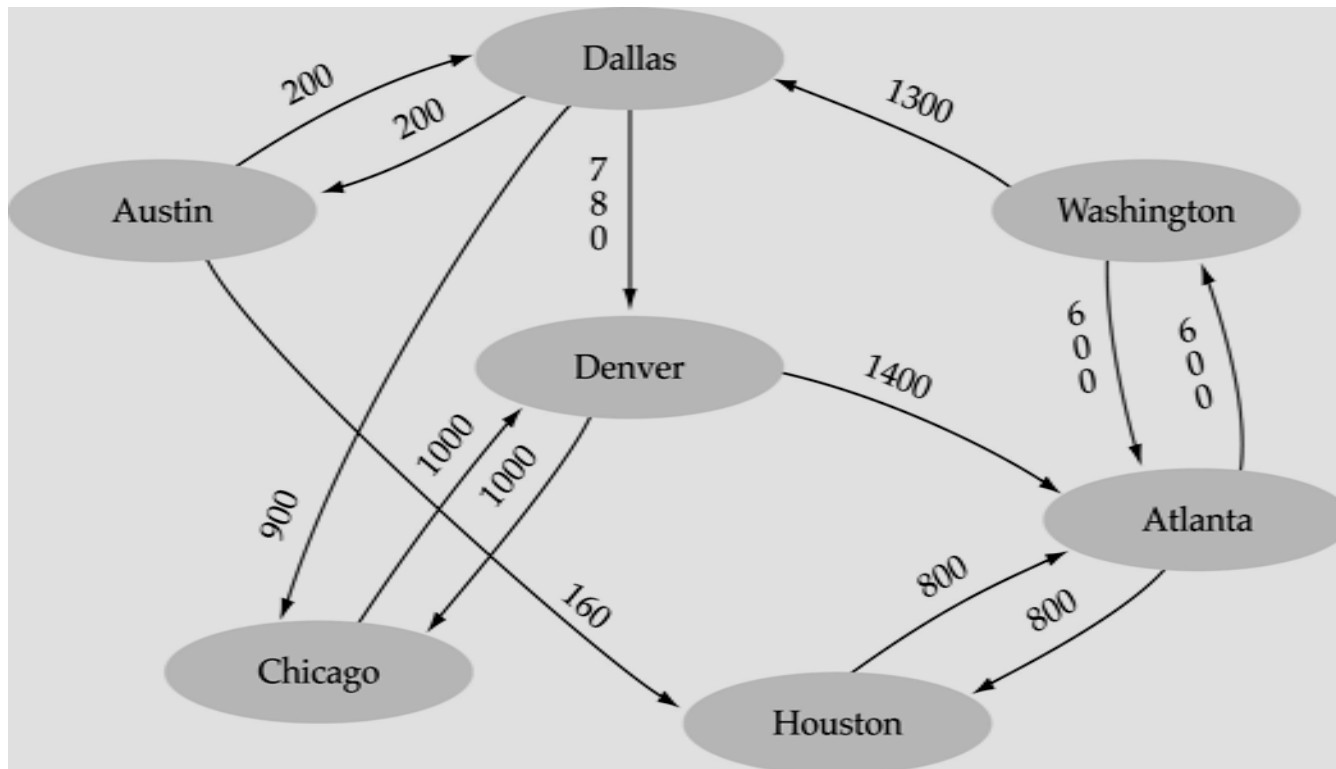
$$O(N^2)$$



(b) Complete undirected graph.

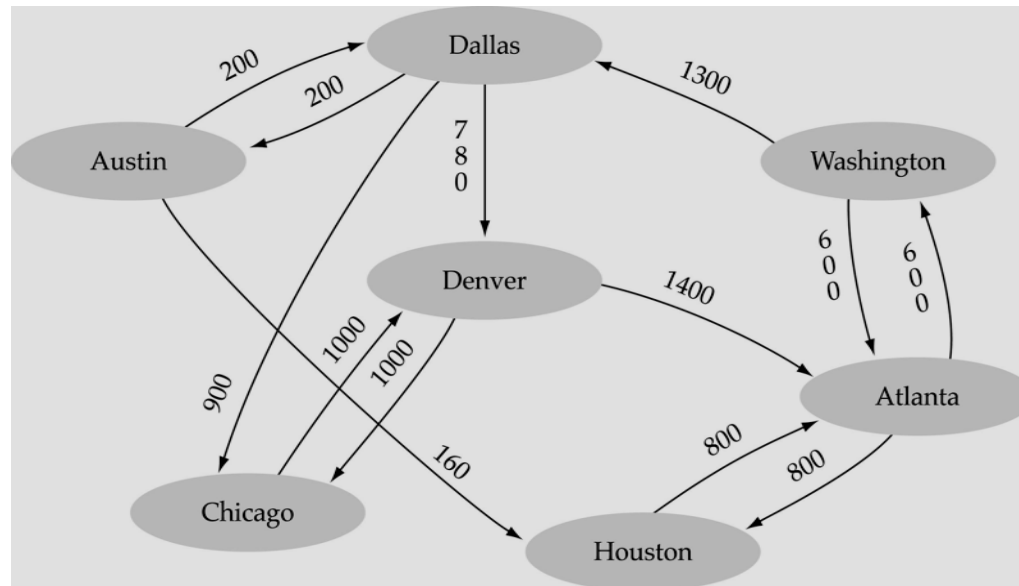
Graph terminology (cont.)

- Weighted graph: a graph in which each edge carries a value



Graph implementation

- Array-based implementation
 - A 1D array is used to represent the vertices
 - A 2D array (adjacency matrix) is used to represent the edges



graph

.numVertices 7

.vertices

[0]	"Atlanta"	"
[1]	"Austin"	"
[2]	"Chicago"	"
[3]	"Dallas"	"
[4]	"Denver"	"
[5]	"Houston"	"
[6]	"Washington"	"
[7]		
[8]		
[9]		

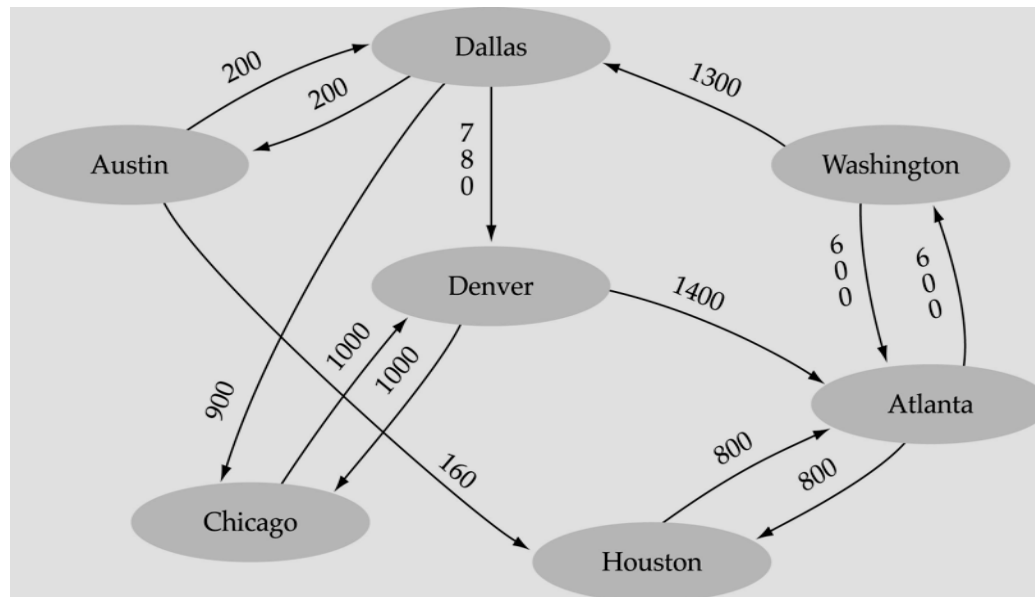
.edges

[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

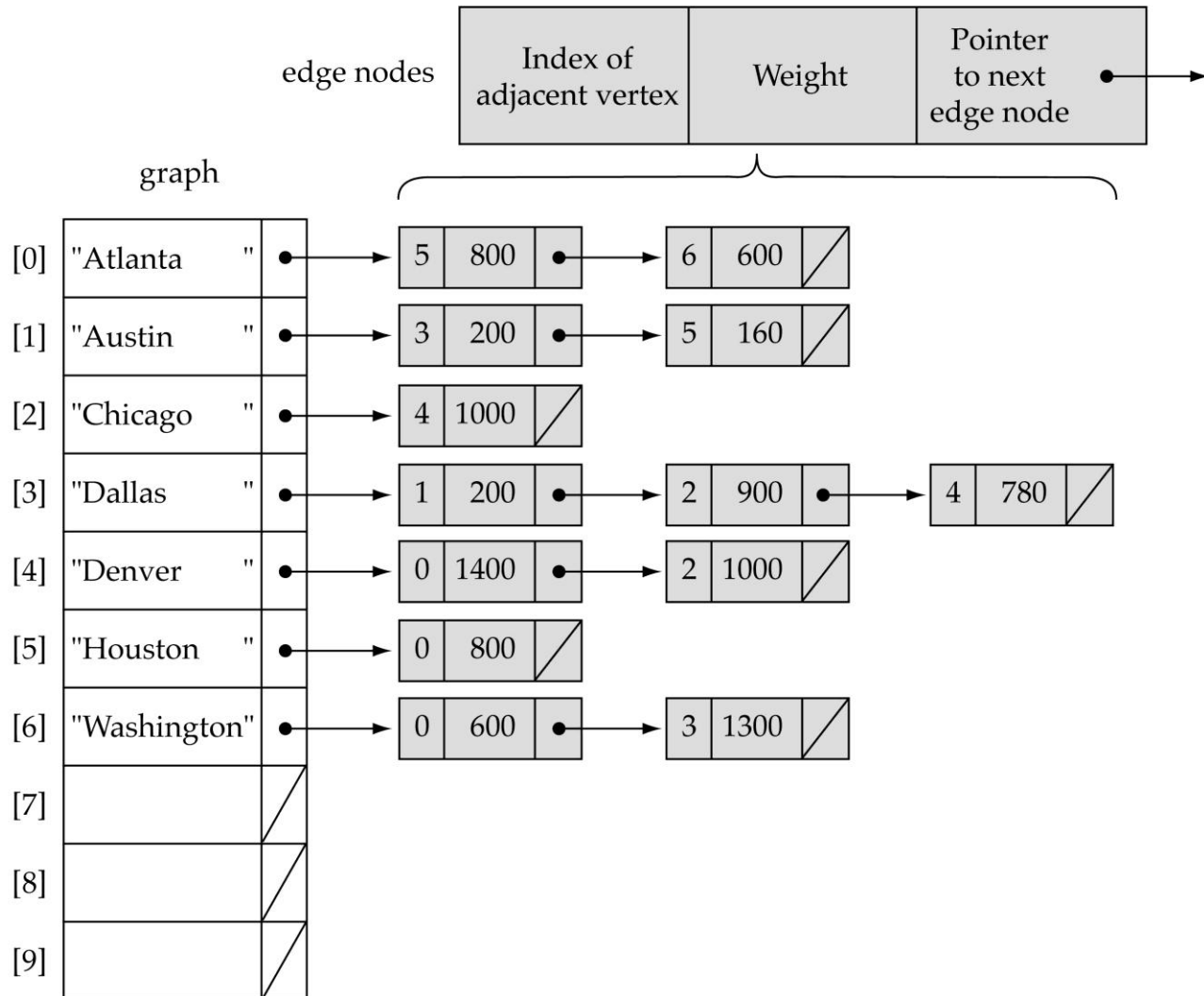
(Array positions marked '•' are undefined)

Graph implementation (cont.)

- Linked-list implementation
 - A 1D array is used to represent the vertices
 - A list is used for each vertex v which contains the vertices which are adjacent from v (adjacency list)



(a)



Adjacency matrix vs. adjacency list representation

- **Adjacency matrix**

- Good for dense graphs -- $|E| \sim O(|V|^2)$
- Memory requirements: $O(|V| + |E|) = O(|V|^2)$
- Connectivity between two vertices can be tested quickly

- **Adjacency list**

- Good for sparse graphs -- $|E| \sim O(|V|)$
- Memory requirements: $O(|V| + |E|) = O(|V|)$
- Vertices adjacent to another vertex can be found quickly

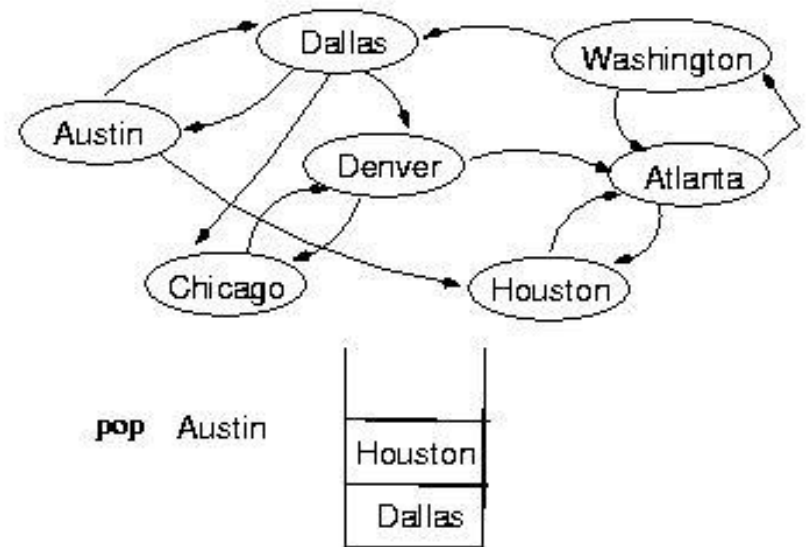
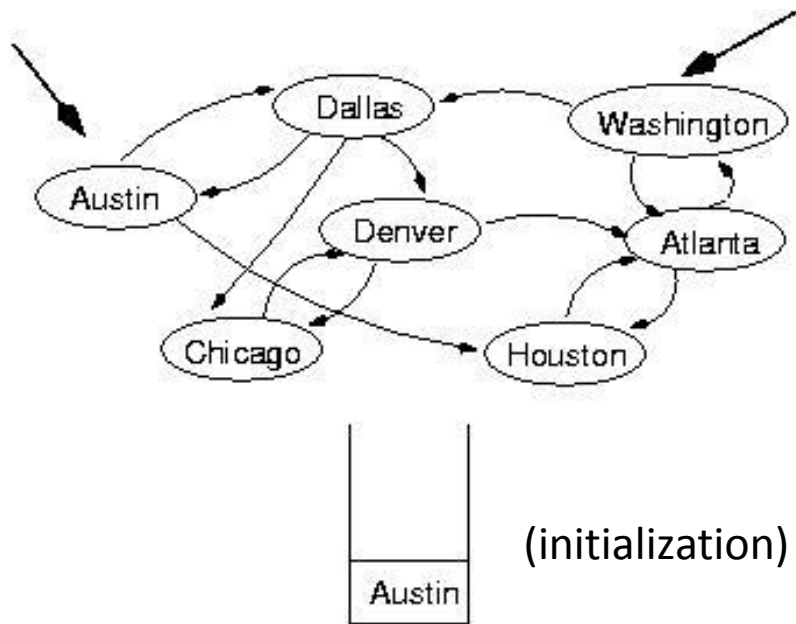
Depth-First-Search (DFS)

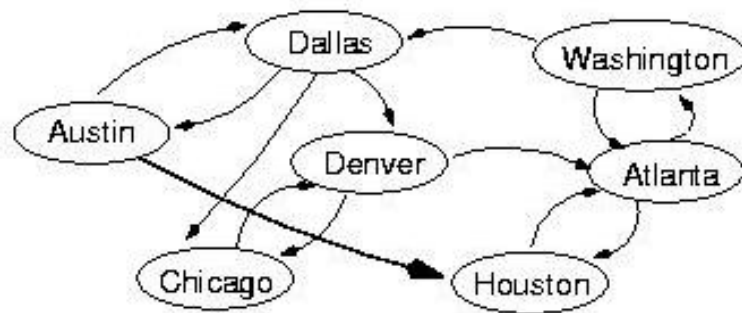
- What is the idea behind DFS?
 - Travel as far as you can down a path
 - Back up *as little as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)
- DFS can be implemented efficiently using a *stack*

Depth-First-Search (DFS) (*cont.*)

```
Set found to false
stack.Push(startVertex)
DO
    stack.Pop(vertex)
    IF vertex == endVertex
        Set found to true
    ELSE
        Push all adjacent vertices onto stack
WHILE !stack.IsEmpty() AND !found

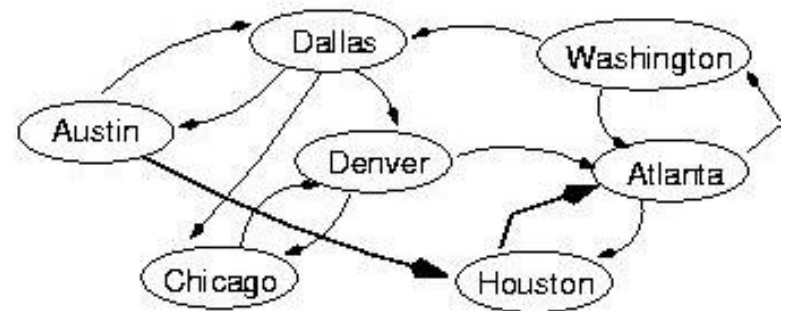
IF(!found)
    Write "Path does not exist"
```





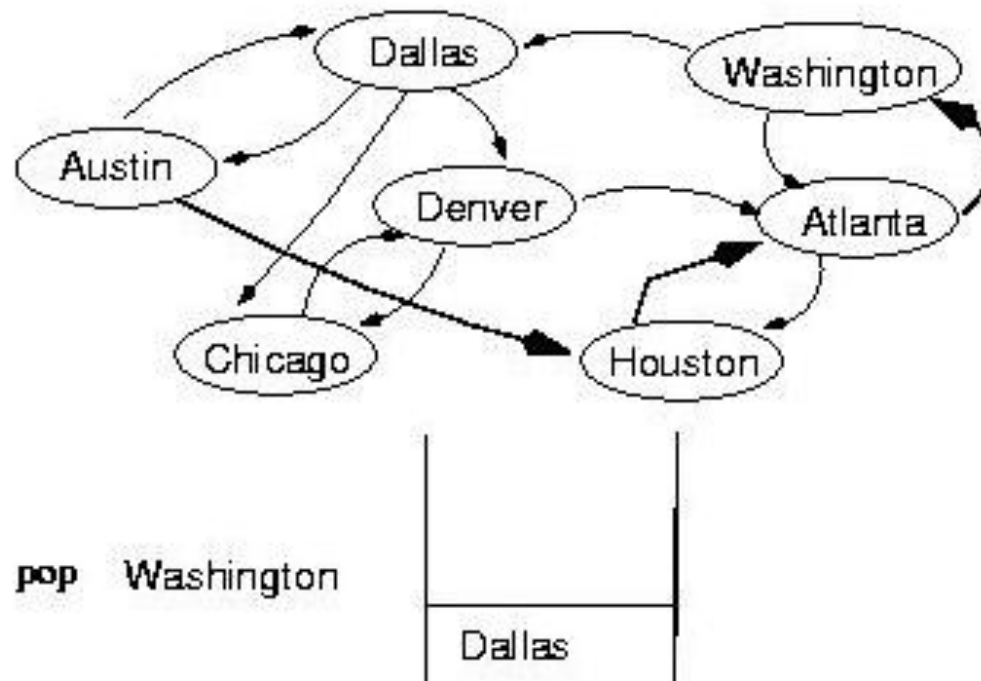
pop Houston

Atlanta
Dallas



pop Atlanta

Washington
Dallas




```
template <class ItemType>
void DepthFirstSearch(GraphType<VertexType> graph, VertexType
    startVertex, VertexType endVertex)
{
    StackType<VertexType> stack;
    QueType<VertexType> vertexQ;

    bool found = false;
    VertexType vertex;
    VertexType item;

    graph.ClearMarks();
    stack.Push(startVertex);
    do {
        stack.Pop(vertex);
        if(vertex == endVertex)
            found = true;
```

(continues)

```

else {
    if(!graph.IsMarked(vertex)) {
        graph.MarkVertex(vertex);
        graph.GetToVertices(vertex, vertexQ);

        while(!vertexQ.IsEmpty()) {
            vertexQ.Dequeue(item);
            if(!graph.IsMarked(item))
                stack.Push(item);
        }
    }
} while(!stack.IsEmpty() && !found);

if(!found)
    cout << "Path not found" << endl;
}

```

(continues)

```

template<class VertexType>
void GraphType<VertexType>::GetToVertices(VertexType vertex,
                                           QueType<VertexType>& adjvertexQ)
{
    int fromIndex;
    int toIndex;

    fromIndex = IndexIs(vertices, vertex);
    for(toIndex = 0; toIndex < numVertices; toIndex++)
        if(edges[fromIndex][toIndex] != NULL_EDGE)
            adjvertexQ.Enqueue(vertices[toIndex]);
}

```

Breadth-First-Searching (BFS)

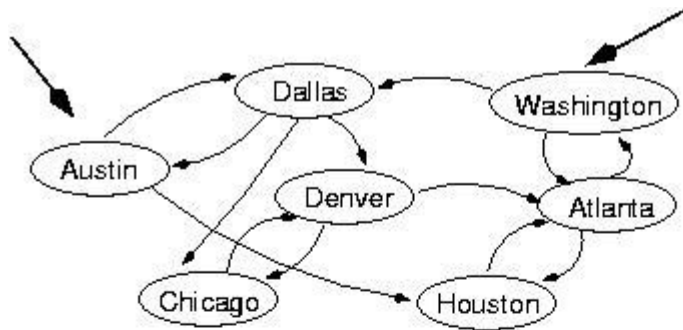
- What is the idea behind BFS?
 - Look at all possible paths at the same depth before you go at a deeper level
 - Back up *as far as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)

Breadth-First-Searching (BFS) (cont.)

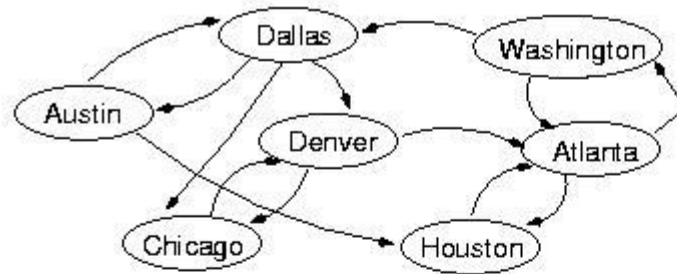
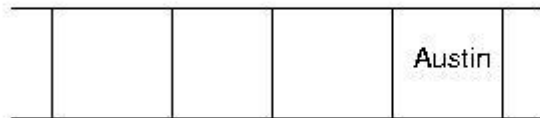
- BFS can be implemented efficiently using a *queue*

```
Set found to false
queue.Enqueue(startVertex)
DO
  queue.Dequeue(vertex)
  IF vertex == endVertex
    Set found to true
  ELSE
    Enqueue all adjacent vertices onto queue
WHILE !queue.IsEmpty() AND !found
```

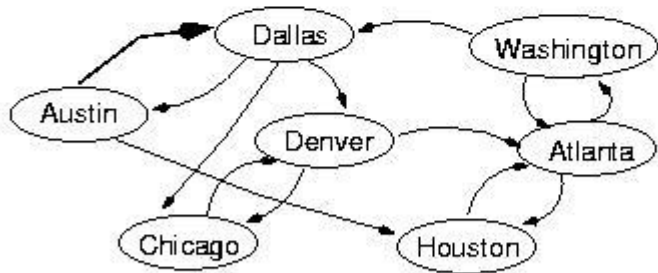
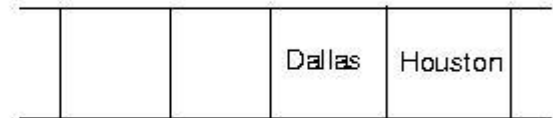
- Should we mark a vertex when it is enqueued or when it is dequeued ?



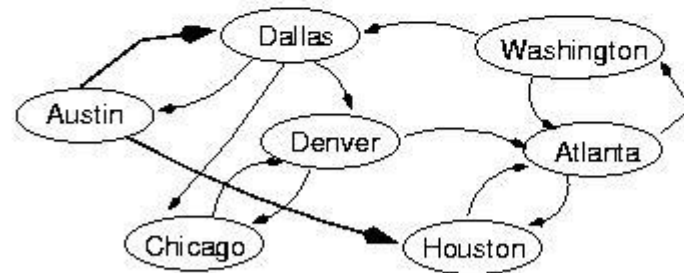
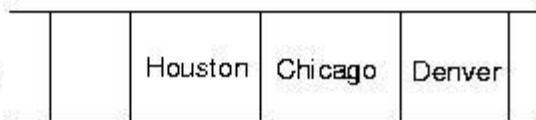
(initialization)



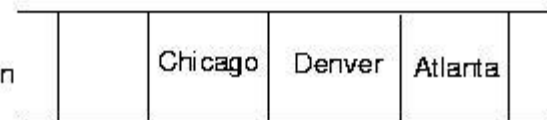
dequeue Austin

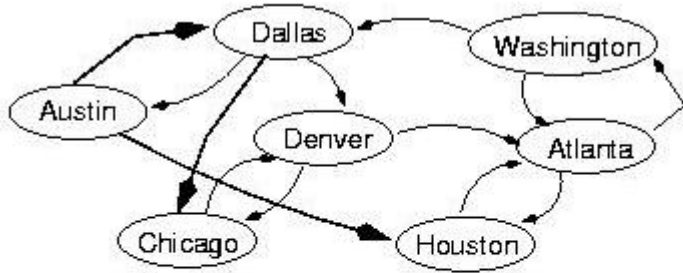


dequeue Dallas



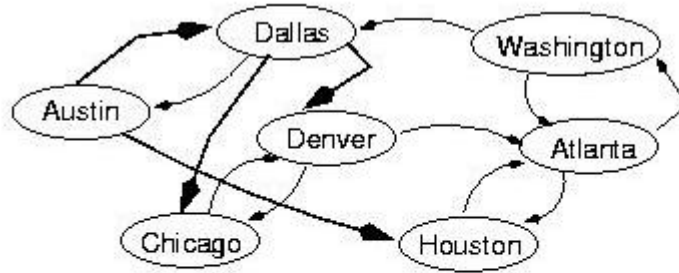
dequeue Houston





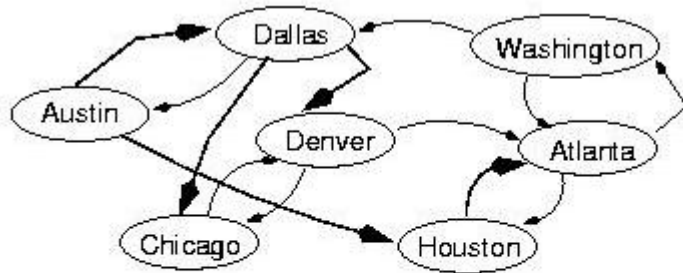
dequeue Chicago

		Denver	Atlanta	Denver
--	--	--------	---------	--------



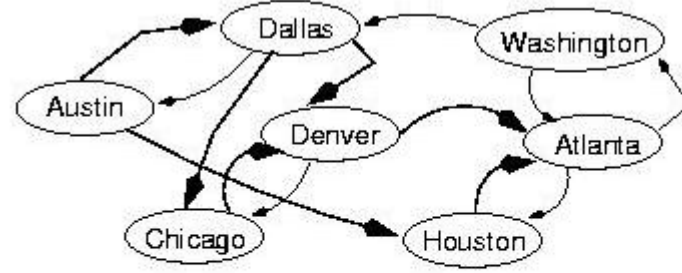
dequeue Denver

		Atlanta	Denver	Atlanta
--	--	---------	--------	---------



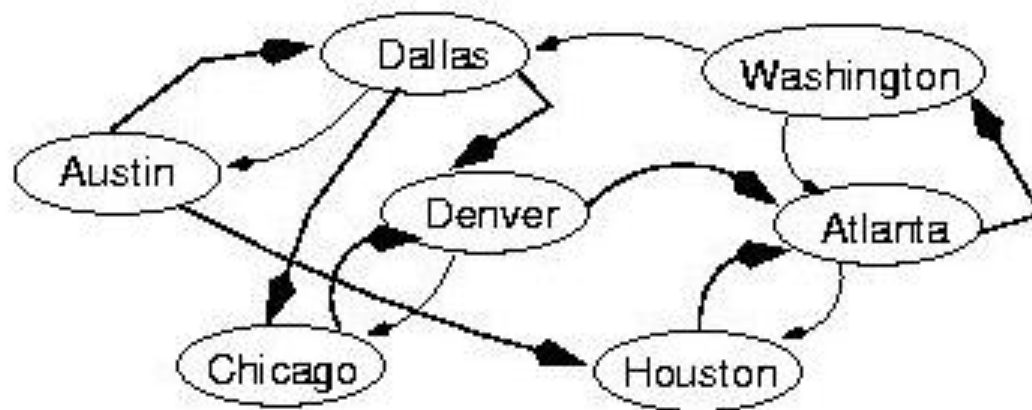
dequeue Atlanta

		Denver	Atlanta	Washington
--	--	--------	---------	------------



dequeue Denver,
next: Atlanta

		Washington	Washington
--	--	------------	------------



dequeue Washington			Washington	
---------------------------	--	--	------------	--


```

template<class VertexType>
void BreadthFirstSearch(GraphType<VertexType> graph,
    VertexType startVertex, VertexType endVertex);
{
    QueType<VertexType> queue;
    QueType<VertexType> vertexQ;//

    bool found = false;
    VertexType vertex;
    VertexType item;

    graph.ClearMarks();
    queue.Enqueue(startVertex);
    do {
        queue.Dequeue(vertex);
        if(vertex == endVertex)
            found = true;
    }

```

(continues)

```

else {
    if(!graph.IsMarked(vertex)) {
        graph.MarkVertex(vertex);
        graph.GetToVertices(vertex, vertexQ);

        while(!vertexQ.IsEmpty()) {
            vertexQ.Dequeue(item);
            if(!graph.IsMarked(item))
                queue.Enqueue(item);
        }
    }
}
} while (!queue.IsEmpty() && !found);

if(!found)
    cout << "Path not found" << endl;
}

```

Single-source shortest-path problem

- There are multiple paths from a source vertex to a destination vertex
- Shortest path: the path whose total weight (i.e., sum of edge weights) is minimum
- Examples:
 - Austin->Houston->Atlanta->Washington: 1560 miles
 - Austin->Dallas->Denver->Atlanta->Washington: 2980 miles

Single-source shortest-path problem (cont.)

- Common algorithms: *Dijkstra's* algorithm, *Bellman-Ford* algorithm
- BFS can be used to solve the shortest graph problem when the graph is weightless or all the weights are the same

(mark vertices before Enqueue)

UNIT- 4

UNIT-4

Sorting and Searching

"There's nothing in your head the sorting hat can't see. So try me on and I will tell you where you ought to be."

-The Sorting Hat, *Harry Potter and the Sorcerer's Stone*

Sorting and Searching

- Fundamental problems in computer science and programming
- Sorting done to make searching easier
- Multiple different algorithms to solve the same problem
 - How do we know which algorithm is "better"?
- Look at searching first
- Examples will use arrays of ints to illustrate algorithms

Searching

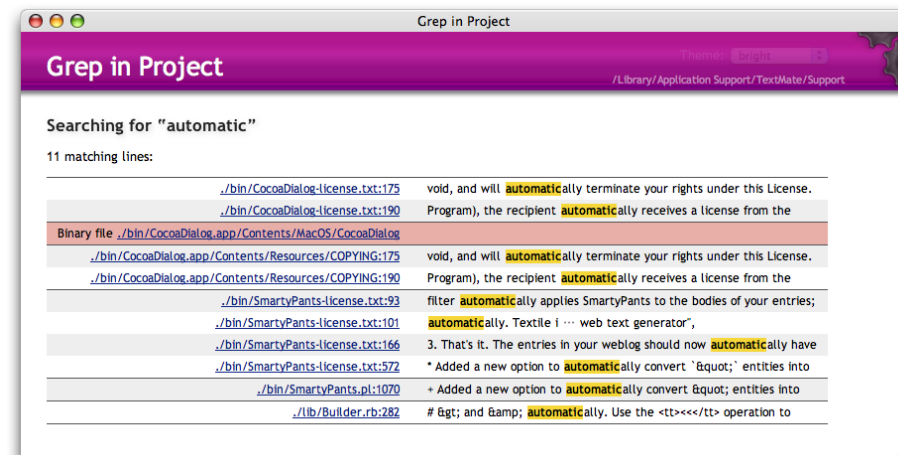


recursive backtracking|

Google Search

I'm Feeling Lucky

[Advanced Search](#)
[Preferences](#)
[Language Tools](#)



Searching

- Given a list of data find the location of a particular value or report that value is present
- linear search
 - intuitive approach
 - start at first item
 - is it the one I am looking for?
 - if not go to next item
 - repeat until found or all items checked
- If items not sorted or unsortable this approach is necessary



Linear Search

```
/* pre: list != null
   post: return the index of the first occurrence
        of target in list or -1 if target not present in
        list
*/
public int linearSearch(int[] list, int target) {
    for(int i = 0; i < list.length; i++)
        if( list[i] == target )
            return i;
    return -1;
}
```

Linear Search, Generic

```
/* pre: list != null, target != null
   post: return the index of the first occurrence
        of target in list or -1 if target not present in
        list
*/
public int linearSearch(Object[] list, Object target) {
    for(int i = 0; i < list.length; i++)
        if( list[i] != null && list[i].equals(target) )
            return i;
    return -1;
}
```

T(N)? Big O? Best case, worst case, average case?

Attendance Question 1

- What is the average case Big O of linear search in an array with N items, if an item is present?
- A. $O(N)$
 - B. $O(N^2)$
 - C. $O(1)$
 - D. $O(\log N)$
 - E. $O(N \log N)$

Searching in a Sorted List

- If items are sorted then we can *divide and conquer*
- dividing your work in half with each step
 - generally a good thing
- The Binary Search on List in Ascending order
 - Start at middle of list
 - is that the item?
 - If not is it less than or greater than the item?
 - less than, move to second half of list
 - greater than, move to first half of list
 - repeat until found or sub list size = 0

Binary Search

list



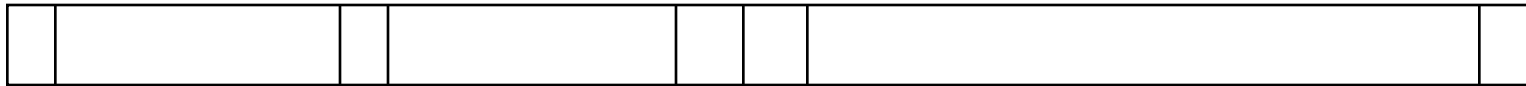
low item

middle item

high item

Is middle item what we are looking for? If not is it
more or less than the target item? (Assume lower)

list



low
item

middle
item

high

item
and so forth...

Binary Search in Action

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

2	3	5	7	11	13	17	19	23	29	31	37	41	43	47	53
---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----

```
public static int bsearch(int[] list, int target)
{
    int result = -1;
    int low = 0;
    int high = list.length - 1;
    int mid;
    while( result == -1 && low <= high )
    {
        mid = low + ((high - low) / 2);
        if( list[mid] == target )
            result = mid;
        else if( list[mid] < target )
            low = mid + 1;
        else
            high = mid - 1;
    }
    return result;
}

// mid = ( low + high ) / 2; // may overflow!!!
// or mid = (low + high) >>> 1; using bitwise op
```

Trace When Key == 3
Trace When Key == 30

Variables of Interest?

Attendance Question 2

What is the worst case Big O of binary search in an array with N items, if an item is present?

- A. $O(N)$
- B. $O(N^2)$
- C. $O(1)$
- D. $O(\log N)$
- E. $O(N \log N)$

Generic Binary Search

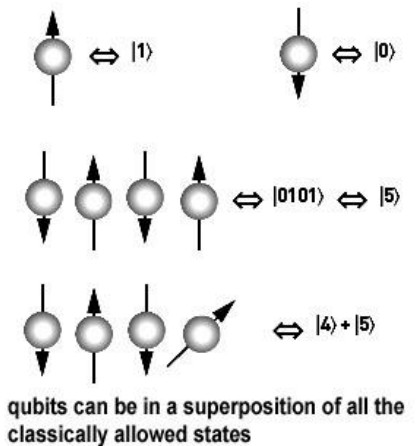
```
public static int bsearch(Comparable[] list, Comparable target)
{
    int result = -1;
    int low = 0;
    int high = list.length - 1;
    int mid;
    while( result == -1 && low <= high )
    {
        mid = low + ((high - low) / 2);
        if( target.equals(list[mid]) )
            result = mid;
        else if(target.compareTo(list[mid]) > 0)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return result;
}
```

Recursive Binary Search

```
public static int bsearch(int[] list, int target) {  
    return bsearch(list, target, 0, list.length - 1);  
}  
  
public static int bsearch(int[] list, int target,  
                           int first, int last) {  
    if( first <= last ) {  
        int mid = low + ((high - low) / 2);  
        if( list[mid] == target )  
            return mid;  
        else if( list[mid] > target )  
            return bsearch(list, target, first, mid - 1);  
        else  
            return bsearch(list, target, mid + 1, last);  
    }  
    return -1;  
}
```

Other Searching Algorithms

- Interpolation Search
 - more like what people really do
- Indexed Searching
- Binary Search Trees
- Hash Table Searching
- Grover's Algorithm (Waiting for quantum computers to be built)
- best-first
- A^*



As of 4/24/08

Women

1	1	2:19:36	Deena Kastor nee Drossin
2		2:21:16	Drossin (2)
3	2	2:21:21	Joan Benoit Samuelson
4		2:21:25	Kastor (3)
5		2:22:43a	Benoit (2)
6		2:24:52a	Benoit (3)
7		2:26:11	Benoit (4)
8	3	2:26:26a	Julie Brown
9	4	2:26:40a	Kim Jones

Sorting



Song Name	Time	Track #	Artist	Album
Letters from the Wasteland	4:29	1 of 10	The Wallflowers	Breach
When You're On Top	3:54	1 of 13	The Wallflowers	Red Letter Days
Hand Me Down	3:35	2 of 10	The Wallflowers	Breach
How Good It Can Get	4:11	2 of 13	The Wallflowers	Red Letter Days
Sleepwalker	3:31	3 of 10	The Wallflowers	Breach
Closer To You	3:17	3 of 13	The Wallflowers	Red Letter Days
I've Been Delivered	5:01	4 of 10	The Wallflowers	Breach
Everybody Out Of The Water	3:42	4 of 13	The Wallflowers	Red Letter Days
Witness	3:34	5 of 10	The Wallflowers	Breach
Three Ways	4:19	5 of 13	The Wallflowers	Red Letter Days
Some Flowers Bloom Dead	4:43	6 of 10	The Wallflowers	Breach
Too Late to Quit	3:54	6 of 13	The Wallflowers	Red Letter Days
Mourning Train	4:04	7 of 10	The Wallflowers	Breach
If You Never Got Sick	3:44	7 of 13	The Wallflowers	Red Letter Days
Up from Under	3:38	8 of 10	The Wallflowers	Breach
Health and Happiness	4:03	8 of 13	The Wallflowers	Red Letter Days
Murder 101	2:31	9 of 10	The Wallflowers	Breach
See You When I Get There	3:09	9 of 13	The Wallflowers	Red Letter Days
Birdcage	7:42	10 of 10	The Wallflowers	Breach
Feels Like Summer Again	3:48	10 of 13	The Wallflowers	Red Letter Days
Everything I Need	3:37	11 of 13	The Wallflowers	Red Letter Days
Here in Pleasantville	3:40	12 of 13	The Wallflowers	Red Letter Days
Empire in My Mind (Bonus Track)	3:31	13 of 13	The Wallflowers	Red Letter Days

Sorting Fun

Why Not Bubble Sort?



Sorting

- A fundamental application for computers
- Done to make finding data (searching) faster
- Many different algorithms for sorting
- One of the difficulties with sorting is working with a fixed size storage container (array)
 - if resize, that is expensive (slow)
- The "simple" sorts run in quadratic time $O(N^2)$
 - bubble sort
 - selection sort
 - insertion sort

Stable Sorting

- A property of sorts
- If a sort guarantees the relative order of equal items stays the same then it is a *stable sort*
- $[7_1, 6, 7_2, 5, 1, 2, 7_3, -5]$
 - subscripts added for clarity
- $[-5, 1, 2, 5, 6, 7_1, 7_2, 7_3]$
 - result of stable sort
- Real world example:
 - sort a table in [Wikipedia](#) by one criteria, then another
 - sort by country, then by major wins

Selection sort

- Algorithm
 - Search through the list and find the smallest element
 - swap the smallest element with the first element
 - repeat starting at second element and find the second smallest element

```
public static void selectionSort(int[] list)
{
    int min;
    int temp;
    for(int i = 0; i < list.length - 1; i++) {
        min = i;
        for(int j = i + 1; j < list.length; j++)
            if( list[j] < list[min] )
                min = j;
        temp = list[i];
        list[i] = list[min];
        list[min] = temp;
    }
}
```

Selection Sort in Practice

44 68 191 119 119 37 83 82 191 45 158 130 76 153 39 25

What is the $T(N)$, *actual* number of statements executed, of the selection sort code, given a list of N elements? What is the Big O?

Generic Selection Sort

```
public void selectionSort(Comparable[] list)
{
    int min; Comparable temp;
    for(int i = 0; i < list.length - 1; i++) {
        min = i;
        for(int j = i + 1; j < list.length; j++)
            if( list[min].compareTo(list[j]) > 0 )
                min = j;
        temp = list[i];
        list[i] = list[min];
        list[min] = temp;
    }
}
```

- ▶ Best case, worst case, average case Big O?

Attendance Question 3

Is selection sort always stable?

- A. Yes
- B. No

Insertion Sort

- Another of the $O(N^2)$ sorts
- The first item is sorted
- Compare the second item to the first
 - if smaller swap
- Third item, compare to item next to it
 - need to swap
 - after swap compare again
- And so forth...

Insertion Sort Code

```
public void insertionSort(int[] list)
{
    int temp, j;
    for(int i = 1; i < list.length; i++)
    {
        temp = list[i];
        j = i;
        while( j > 0 && temp < list[j - 1])
        {
            // swap elements
            list[j] = list[j - 1];
            list[j - 1] = temp;
            j--;
        }
    }
}
```

- Best case, worst case, average case Big O?

Attendance Question 4

- Is the version of insertion sort shown always stable?
- A. Yes
- B. No

Comparing Algorithms

- Which algorithm do you think will be faster given random data, selection sort or insertion sort?
- Why?

Sub Quadratic Sorting Algorithms

Sub Quadratic means having a Big O
better than $O(N^2)$

ShellSort

- Created by Donald Shell in 1959
- Wanted to stop moving data small distances (in the case of insertion sort and bubble sort) and stop making swaps that are not helpful (in the case of selection sort)
- Start with sub arrays created by looking at data that is far apart and then reduce the gap size



ShellSort in practice

46 2 83 41 102 5 17 71 64 49 18
Gap of five. Sort sub array with 46, 5, and 18

5 2 83 41 102 18 17 31 64 49 46

Gap still five. Sort sub array with 2 and 17

5 2 83 41 102 18 17 31 64 49 46

Gap still five. Sort sub array with 83 and 31

5 2 31 41 102 18 17 83 64 49 46

Gap still five Sort sub array with 41 and 64

5 2 31 41 102 18 17 83 64 49 46

Gap still five. Sort sub array with 102 and 49

5 2 31 41 49 18 17 83 64 102 46

Continued on next slide:

Completed Shellsort

5 2 31 41 49 18 17 83 64 102 46

Gap now 2: Sort sub array with 5 31 49 17 64 46

5 2 17 41 31 18 46 83 49 102 64

Gap still 2: Sort sub array with 2 41 18 83 102

5 2 17 18 31 41 46 83 49 102 64

Gap of 1 (Insertion sort)

2 5 17 18 31 41 46 49 64 83 102

Array sorted

Shell sort on Another Data Set

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		
44	68	191	119	119	37	83	82	191	45	158	130	76	153	39	25		

Initial gap = length / 2 = 16 / 2 = 8

initial sub arrays indices:

{0, 8}, {1, 9}, {2, 10}, {3, 11}, {4, 12}, {5, 13}, {6, 14}, {7, 15}

next gap = 8 / 2 = 4

{0, 4, 8, 12}, {1, 5, 9, 13}, {2, 6, 10, 14}, {3, 7, 11, 15}

next gap = 4 / 2 = 2

{0, 2, 4, 6, 8, 10, 12, 14}, {1, 3, 5, 7, 9, 11, 13, 15}

final gap = 2 / 2 = 1

ShellSort Code

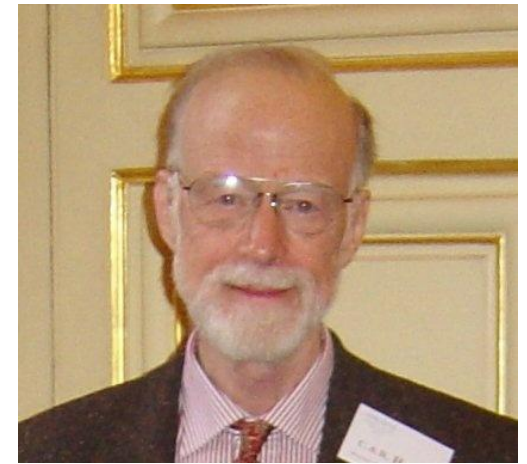
```
public static void shellsort(Comparable[] list)
{
    Comparable temp; boolean swap;
    for(int gap = list.length / 2; gap > 0; gap /= 2)
        for(int i = gap; i < list.length; i++)
        {
            Comparable tmp = list[i];
            int j = i;
            for( ; j >= gap &&
                tmp.compareTo( list[j - gap] ) < 0;
                j -= gap )
                list[ j ] = list[ j - gap ];
            list[ j ] = tmp;
        }
}
```

Comparison of Various Sorts

Num Items	Selection	Insertion	Shellsort	Quicksort
1000	16	5	0	0
2000	59	49	0	6
4000	271	175	6	5
8000	1056	686	11	0
16000	4203	2754	32	11
32000	16852	11039	37	45
64000	expected?	expected?	100	68
128000	expected?	expected?	257	158
256000	expected?	expected?	543	335
512000	expected?	expected?	1210	722
1024000	expected?	expected?	2522	1550

times in milliseconds

Quicksort



- Invented by C.A.R. (Tony) Hoare
 - A divide and conquer approach that uses recursion
1. If the list has 0 or 1 elements it is sorted
 2. otherwise, pick any element p in the list. This is called the pivot value
 3. Partition the list minus the pivot into two sub lists according to values less than or greater than the pivot. (equal values go to either)
 4. return the quicksort of the first list followed by the quicksort of the second list

Quicksort in Action

39 23 17 90 33 72 46 79 11 52 64 5 71

Pick middle element as pivot: 46

Partition list

23 17 5 33 39 11 46 79 72 52 64 90 71

quicksort the less than list

Pick middle element as pivot: 33

23 17 5 11 33 39

quicksort the less than list, pivot now 5

{ } 5 23 17 11

quicksort the less than list, base case

quicksort the greater than list

Pick middle element as pivot: 17

and so on....

Quicksort on Another Data Set

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		
44	68	191	119	119	37	83	82	191	45	158	130	76	153	39	25		

Big O of Quicksort?

```

public static void swapReferences( Object[] a, int index1, int index2 )
{
    Object tmp = a[index1];
    a[index1] = a[index2];
    a[index2] = tmp;
}

public void quicksort( Comparable[] list, int start, int stop )
{
    if(start >= stop)
        return; //base case list of 0 or 1 elements

    int pivotIndex = (start + stop) / 2;

    // Place pivot at start position
    swapReferences(list, pivotIndex, start);
    Comparable pivot = list[start];

    // Begin partitioning
    int i, j = start;

    // from first to j are elements less than or equal to pivot
    // from j to i are elements greater than pivot
    // elements beyond i have not been checked yet
    for(i = start + 1; i <= stop; i++ )
    {
        //is current element less than or equal to pivot
        if(list[i].compareTo(pivot) <= 0)
        {
            // if so move it to the less than or equal portion
            j++;
            swapReferences(list, i, j);
        }
    }

    //restore pivot to correct spot
    swapReferences(list, start, j);
    quicksort( list, start, j - 1 );    // Sort small elements
    quicksort( list, j + 1, stop );    // Sort large elements
}

```

Attendance Question 5

- What is the best case and worst case Big O of quicksort?

Best

Worst

- | | |
|------------------|---------------|
| A. $O(N \log N)$ | $O(N^2)$ |
| B. $O(N^2)$ | $O(N^2)$ |
| C. $O(N^2)$ | $O(N!)$ |
| D. $O(N \log N)$ | $O(N \log N)$ |
| E. $O(N)$ | $O(N \log N)$ |

Quicksort Caveats

- Average case Big O?
- Worst case Big O?
- Coding the partition step is usually the hardest part

Attendance Question 6

- You have 1,000,000 items that you will be searching. How many searches need to be performed before the data is changed to make sorting worthwhile?
- A. 10
 - B. 40
 - C. 1,000
 - D. 10,000
 - E. 500,000

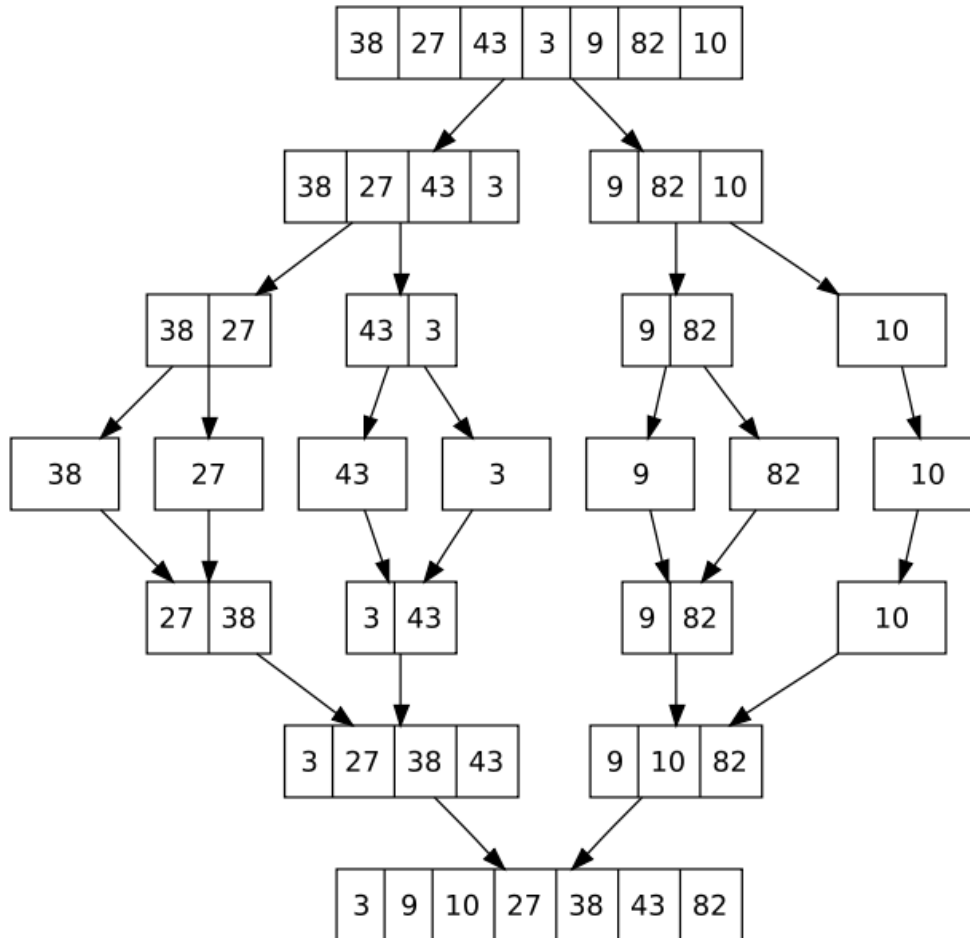
Merge Sort Algorithm

Don Knuth cites John von Neumann as the creator of this algorithm

1. If a list has 1 element or 0 elements it is sorted
2. If a list has more than 2 split into into 2 separate lists
3. Perform this algorithm on each of those smaller lists
4. Take the 2 sorted lists and merge them together



Merge Sort



When implementing one temporary array is used instead of multiple temporary arrays.

Why?

Merge Sort code

```
/**
 * perform a merge sort on the data in c
 * @param c c != null, all elements of c
 * are the same data type
 */
public static void mergeSort(Comparable[] c)
{
    Comparable[] temp = new Comparable[ c.length ];
    sort(c, temp, 0, c.length - 1);
}

private static void sort(Comparable[] list, Comparable[] temp,
                        int low, int high)
{
    if( low < high){
        int center = (low + high) / 2;
        sort(list, temp, low, center);
        sort(list, temp, center + 1, high);
        merge(list, temp, low, center + 1, high);
    }
}
```

Merge Sort Code

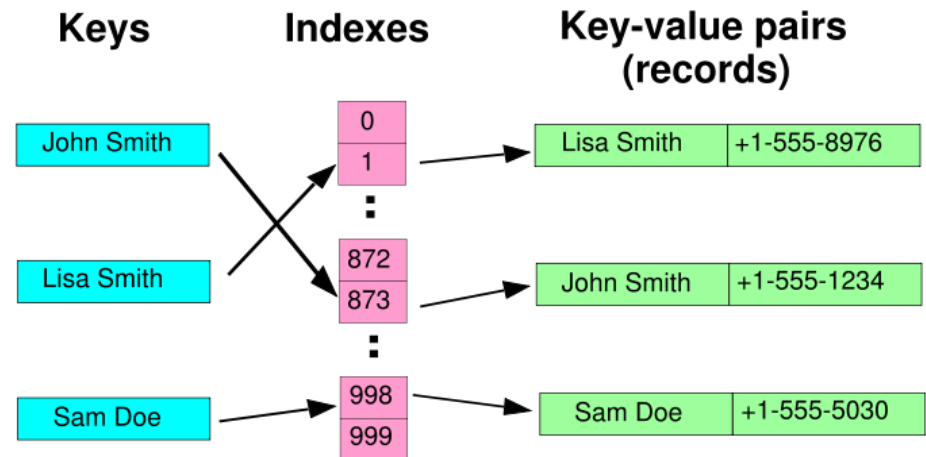
```
private static void merge( Comparable[] list, Comparable[] temp,
                          int leftPos, int rightPos, int rightEnd){
    int leftEnd = rightPos - 1;
    int tempPos = leftPos;
    int numElements = rightEnd - leftPos + 1;
    //main loop
    while( leftPos <= leftEnd && rightPos <= rightEnd){
        if( list[ leftPos ].compareTo(list[rightPos]) <= 0){
            temp[ tempPos ] = list[ leftPos ];
            leftPos++;
        }
        else{
            temp[ tempPos ] = list[ rightPos ];
            rightPos++;
        }
        tempPos++;
    }
    //copy rest of left half
    while( leftPos <= leftEnd){
        temp[ tempPos ] = list[ leftPos ];
        tempPos++;
        leftPos++;
    }
    //copy rest of right half
    while( rightPos <= rightEnd){
        temp[ tempPos ] = list[ rightPos ];
        tempPos++;
        rightPos++;
    }
    //Copy temp back into list
    for(int i = 0; i < numElements; i++, rightEnd--)
        list[ rightEnd ] = temp[ rightEnd ];
}
```

Final Comments

- Language libraries often have sorting algorithms in them
 - Java Arrays and Collections classes
 - C++ Standard Template Library
 - Python sort and sorted functions
- Hybrid sorts
 - when size of unsorted list or portion of array is small use insertion sort, otherwise use $O(N \log N)$ sort like Quicksort or Mergesort
- Many other sorting algorithms exist.

Hash tables

- Keys and values
- $O(1)$ lookup
- Hash function
 - Good v fast
- Clustering
- Databases

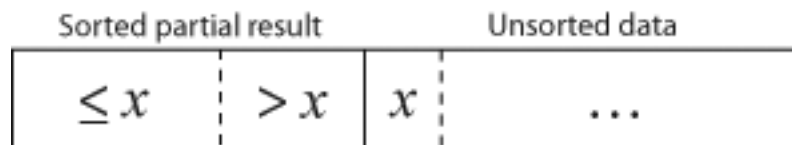


Selection sort :- (

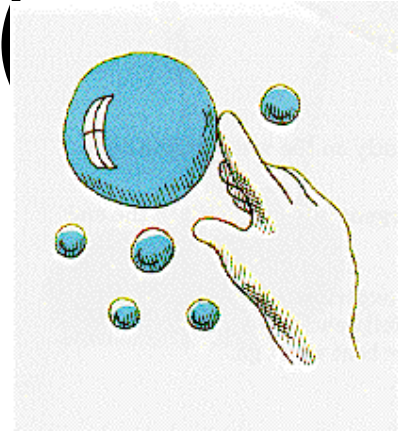
- $O(n^2)$
- Algorithm:
 - Find the minimum value
 - Swap with 1st position value
 - Repeat with 2nd position down

Insertion sort :-)

- $O(n^2)$
- $O(1)$ space
- Great with small number of elements (becomes relevant later)
- Algorithm:
 - Move element from unsorted to sorted list



Bubble sort :- (



- $O(n^2)$
- Algorithm:
 - Iterate through each n , and sort with $n+1$ element
- Maybe go $n-1$ steps every iteration?
- Great for big numbers, bad for small
- Totally useless?

Merge sort :-)

- $O(n \log n)$
- Requires $O(n)$ extra space
- Parallelizable
- Algorithm:
 - Break list into 2 sublists
 - Sort sublist
 - Merge



Quick sort :-)

- Average $O(n \log n)$, worst $O(n^2)$
- $O(n)$ extra space (can optimized for $O(\log n)$)
- Algorithm:
 - pick a pivot
 - put all $x < \text{pivot}$ in less, all $x > \text{pivot}$ in more
 - Concat and recurse through less, pivot, and more
- Advantages also based on caching, registry (single pivot comparison)
- Variations: use fat pivot

Linear search :- (

- $O(n)$
- Examines every item

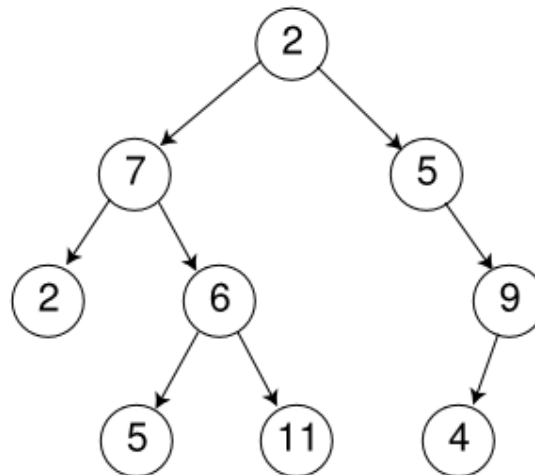
Binary search :-)

- Requires a sorted list
- $O(\log n)$
- Divide and conquer

Tre

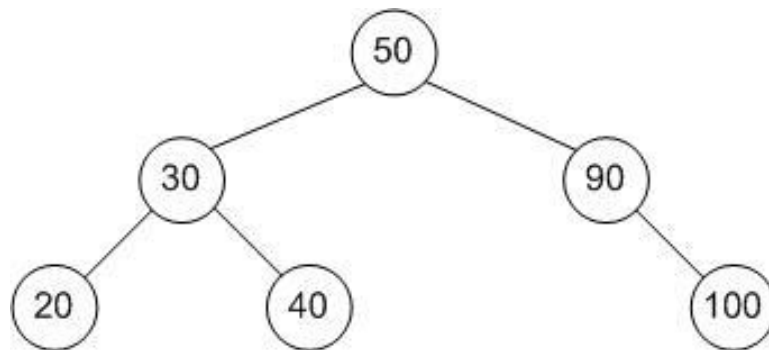


- Almost like linked lists!
- Traverse: Pre-order v. Post-order v. In-order
- Node, edge, sibling/parent/child, leaf

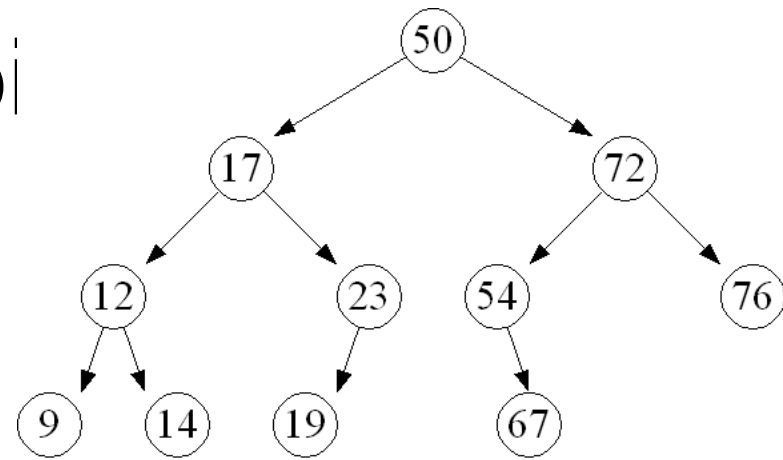


Binary trees

- 0, 1, or 2 children per node
- Binary Search Tree: a binary tree where $\text{node.left_child} < \text{node.value}$ and $\text{node.right_child} \geq \text{node.value}$

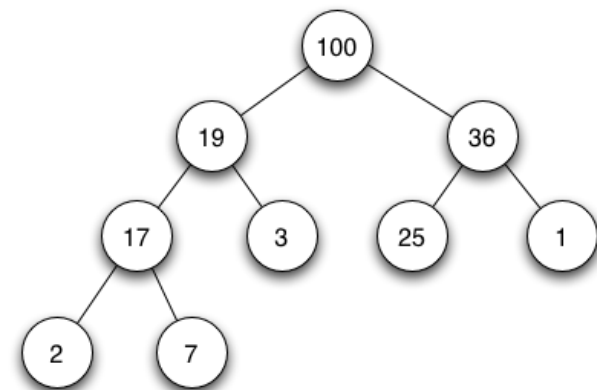


Balanced binary trees



- Minimizes the level of nodes
- Compared with “bad” binary tree?
- Advantages:
 - Lookup, insertion, removal: $O(\log n)$
- Disadvantages:
 - Overhead to maintain balance

Heaps (binary)



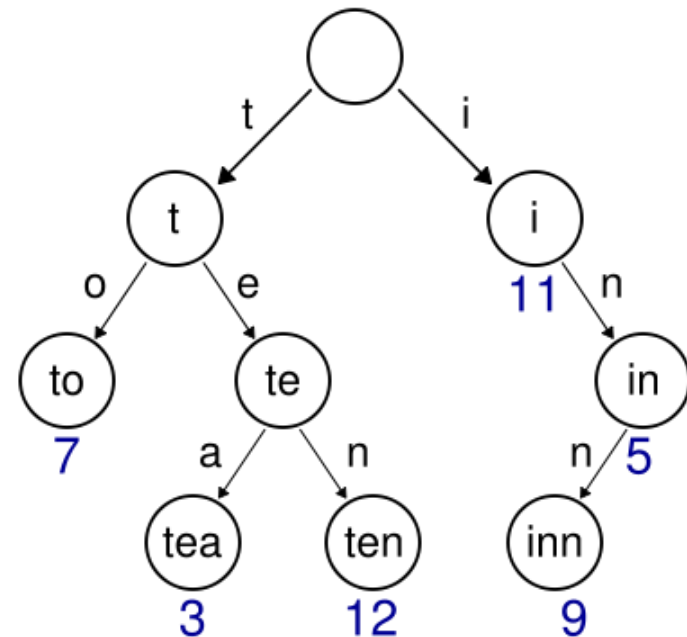
- Complete: all leafs are at n or $n-1$, left
- $\text{Node.value} \geq \text{child.value}$
- In binary min/max heap
 - Insert = $O(\log n)$.. add to bottom, bubble-up
 - deleteMax = $O(\log n)$.. Move last to root and bubble-down

Heapsort

- $O(n \log n)$
- Algorithm:
 - Build a heap
 - deleteMax (or Min) repeatedly
- $O(1)$ overhead

Why bother?

- Tries (say trees)
 - Position determines the key
 - Great for lots of short words
 - Prefix matching
- But..
 - Long strings..
 - Complex algorithms





- Minimax:

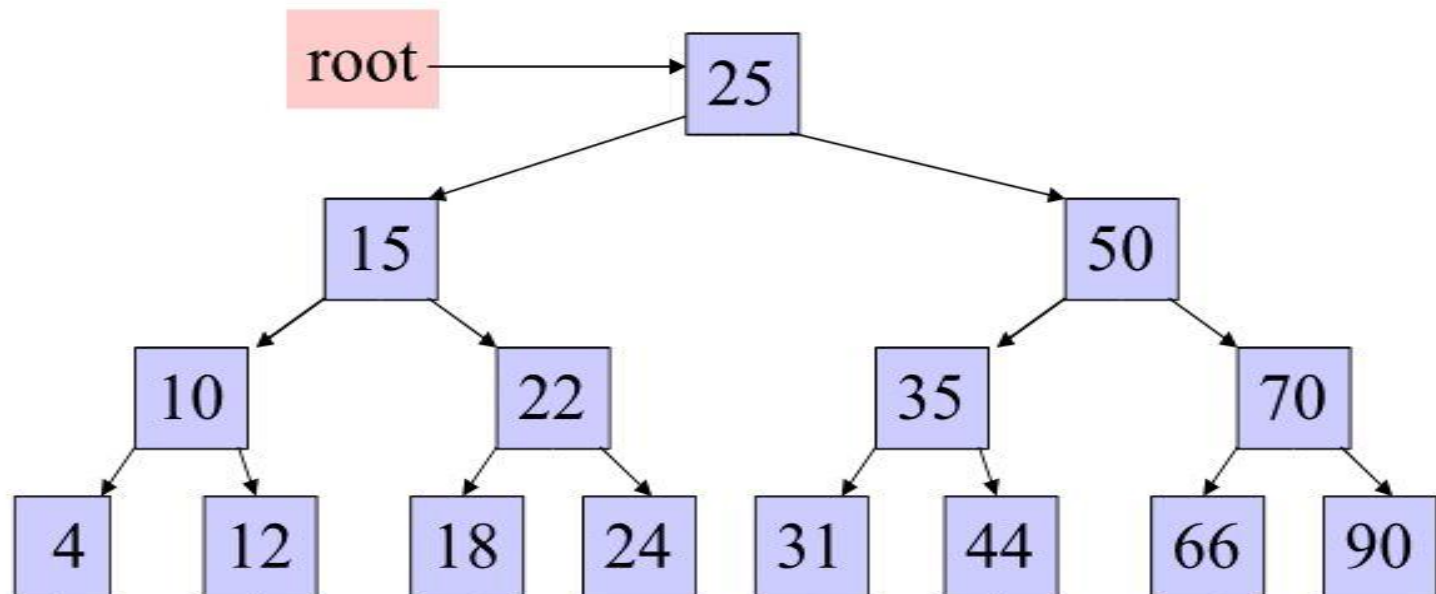
	B: B1	B: B2	B: B3
A: A1	+3	-2	+2
A: A2	-1	0	+4
A: A3	-4	-3	+1

- Alpha-beta pruning - pick a bag!
 - ordering

UNIT- 5

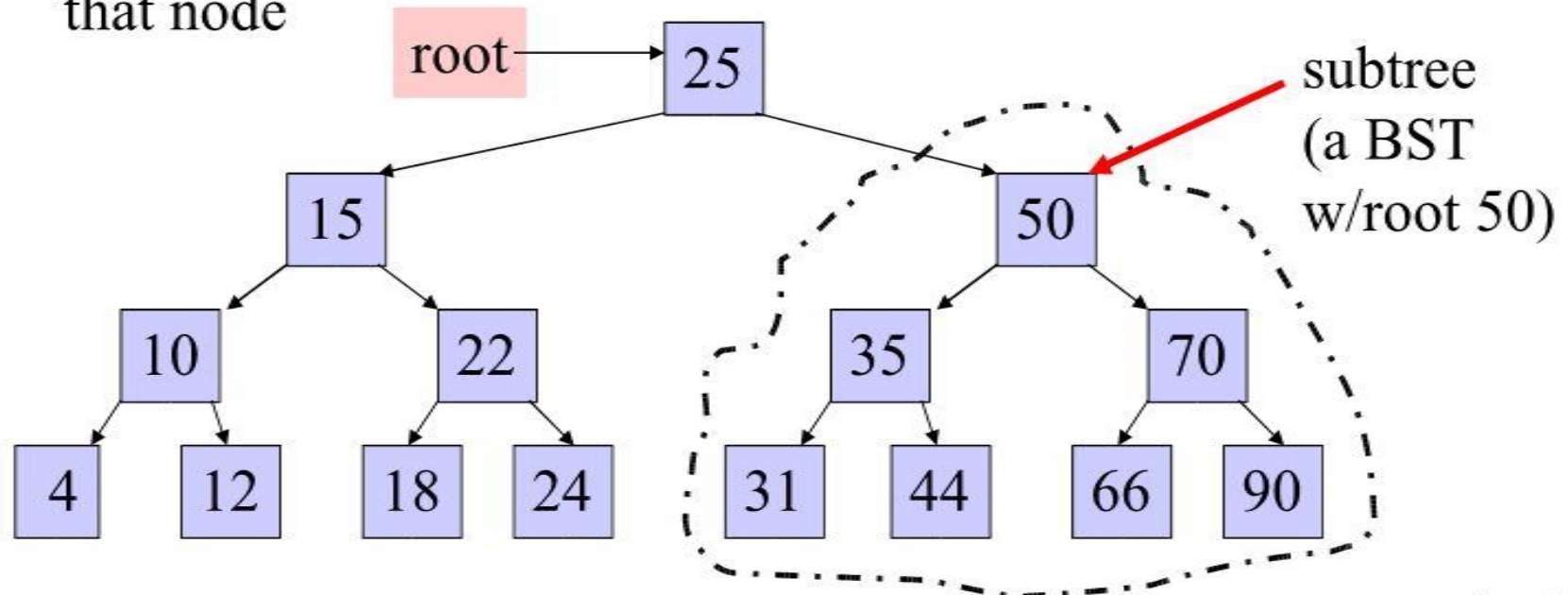
Binary Search Trees (BST)

1. Hierarchical data structure with a single pointer to root node
2. Each node has at most two child nodes (a left and a right child)
3. Nodes are organized by the Binary Search property:
 - Every node is ordered by some key data field(s)
 - For every node in the tree, its key is greater than its left child's key and less than its right child's key



Some BST Terminology

1. The Root node is the top node in the hierarchy
2. A Child node has exactly one Parent node, a Parent node has at most two child nodes, Sibling nodes share the same Parent node (ex. node 22 is a child of node 15)
3. A Leaf node has no child nodes, an Interior node has at least one child node (ex. 18 is a leaf node)
4. Every node in the BST is a Subtree of the BST rooted at that node



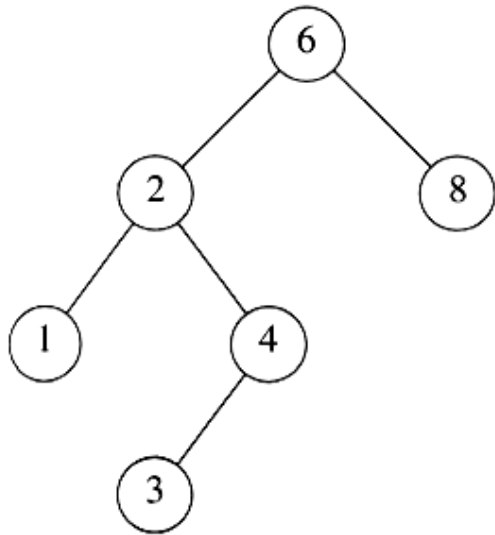
Comparision Between Binary Tree & Binary Search Tree

- * A binary search tree is a binary tree in which it has atmost two children, the key values in the left node is less than the root and the key values in the right node is greater than the root.**
- * It doesn't have any order.**

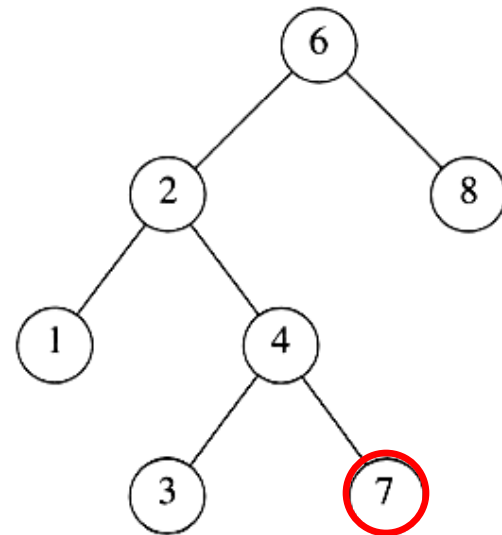
Note : * Every binary search tree is a binary tree.

- * All binary trees need not be a binary search tree**

Example of Binary Search Tree



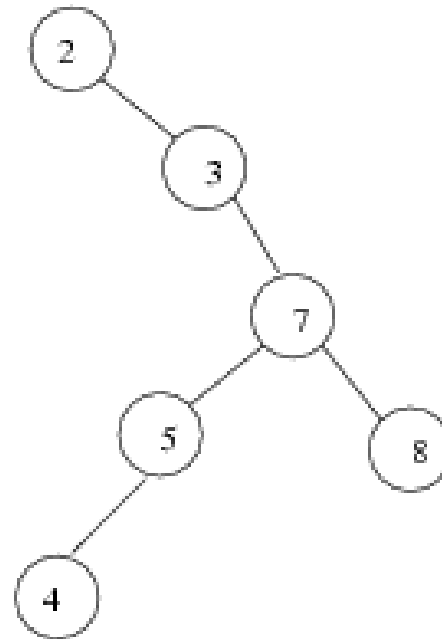
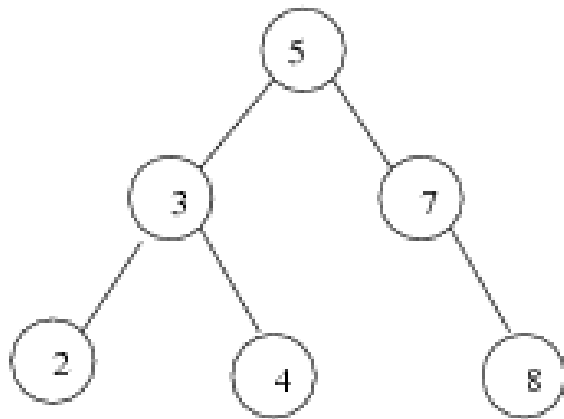
A binary search tree



Not a binary search tree

Binary Search Trees

The same set of keys may have different BSTs



DECLARATION ROUTINE FOR BINARY SEARCH TREE

Struct TreeNode

{

int Element ;

SearchTree Left;

SearchTree Right;

};

BST Operations

•The 3 basic BST operations are: search, insert, and delete; and develop algorithms for searches, insertion, and deletion.

- Searches**
- Insertion**
- Deletion**

Three BST search algorithms:

- Find the **smallest** node
- Find the **largest** node
- Find a **requested** node

Find : -

- **Check whether the root is NULL if so then return NULL.**
- **Otherwise, Check the value X with the root node value (i.e. T data)**
- **(1) If X is equal to T data, return T.**
- **(2) If X is less than T data, Traverse the left of T recursively.**
- **(3) If X is greater than T data, traverse the right of T recursively.**

ROUTINE FOR FIND OPERATION

Int Find (int X, SearchTree T)

```
{  
  If T == NULL)  
    Return NULL ;  
  If (X < T Element)  
    return Find (X, T →left);  
  else  
    If (X > T → Element)  
      return Find (X, T →Right);  
  else  
    return T; // returns the position of  
      the search element.
```

}

ALGORITHM

Search BST

Algorithm searchBST (root, targetKey)

Search a binary search tree for a given value.

Pre root is the root to a binary tree or
 targetKey is the key value requested

Return the node address if the value is found
 null if the node is not in the tree

1 if (empty tree)

 Not found

 1 return null

2 end if

3 if (targetKey < root)

 1 return searchBST (left subtree, targetKey)

4 else if (targetKey > root)

 1 return searchBST (right subtree, targetKey)

5 else

 Found target key

 1 return root

6 end if

end searchBST

Find Min :

- **This operation returns the position of the smallest element in the tree.**
- **To perform FindMin, start at the root and go left as long as there is a left child. The stopping point is the smallest element.**

RECURSIVE ROUTINE FOR FINDMIN

```
int FindMin (SearchTree T)
{
    if (T == NULL);
    return NULL ;
    else if (T →left == NULL)
    return T;
    else
    return FindMin (T → left);
```


Example : - Root T

(a) $T \neq \text{NULL}$ and $T \rightarrow \text{left} \neq \text{NULL}$,

(b) $T \neq \text{NULL}$ and $T \rightarrow \text{left} \neq \text{NULL}$,

Traverse left Traverse left until Min T

(c) Since Tleft is Null, return T as a minimum element.

NON - RECURSIVE ROUTINE FOR FINDMIN

```
int FindMin (SearchTree T)  
{  
if (T! = NULL)  
while (T →Left ! = NULL)  
T = T →Left ;  
return T;  
}
```

RECURSIVE ROUTINE FOR FINDMAX

```
int FindMax (SearchTree T)  
{  
if (T == NULL)  
return NULL ;  
else if (T →Right == NULL)  
return T;  
else FindMax (T →Right);  
}
```

Example :- Root T

(a) T! = NULL and T→Right!=NULL, (b) T! = NULL
and T→Right!=NULL,

Traverse Right Traverse Right
Max

(c) Since T Right is NULL, return T as a
Maximum element.

FindMax

- **FindMax routine return the position of largest elements in the tree.**
- **To perform a FindMax, start at the root and go right as long as there is a right child.**
- **The stopping point is the largest element.**

RECURSIVE ROUTINE FOR FINDMAX

```
int FindMax (SearchTree T)  
{  
if (T == NULL)  
return NULL ;  
else if (T →Right == NULL)  
return T;  
else FindMax (T →Right);  
}
```

Example :- Root T

(a) $T \neq \text{NULL}$ and $T \rightarrow \text{Right} \neq \text{NULL}$, (b) $T \neq \text{NULL}$ and $T \rightarrow \text{Right} \neq \text{NULL}$,

Traverse Right Traverse Right Max

(c) Since T Right is NULL, return T as a Maximum element.

NON - RECURSIVE ROUTINE FOR FINDMAX

```
int FindMax (SearchTree T)  
{  
if (T! = NULL)  
while (T Right ! = NULL)  
T = T →Right ;  
return T ;  
}
```

Find a Node into the BST

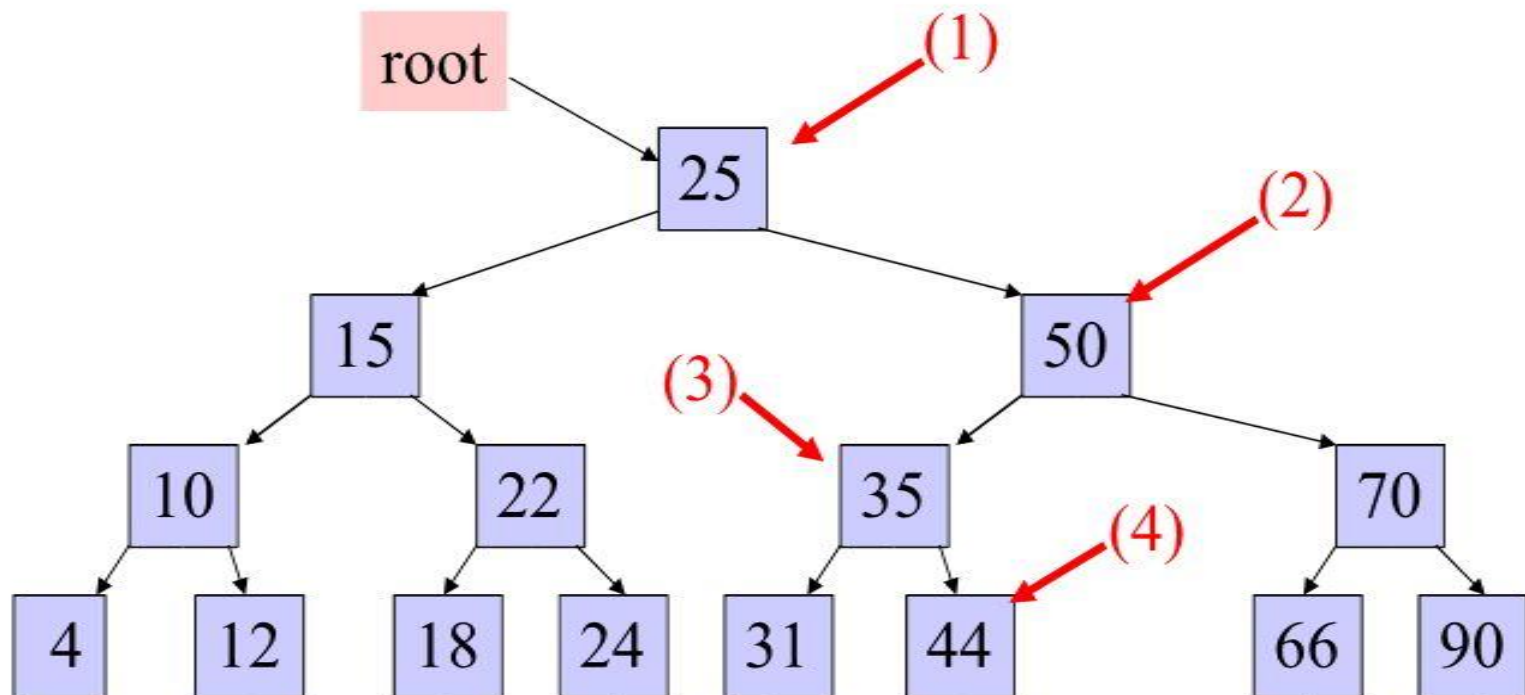
- Use the search key to direct a recursive binary

search for a matching node

1. Start at the root node as current node
2. If the search key's value matches the current node's key then found a match
3. If search key's value is greater than current node's
 1. If the current node has a right child, search right
 2. Else, no matching node in the tree
4. If search key is less than the current node's
 1. If the current node has a left child, search left
 2. Else, no matching node in the tree

Example: search for 45 in the tree:

1. start at the root, 45 is greater than 25, search in right subtree
2. 45 is less than 50, search in 50's left subtree
3. 45 is greater than 35, search in 35's right subtree
4. 45 is greater than 44, but 44 has no right subtree so 45 is not in the BST



ALGORITHM

Find Smallest Node in a BST

Algorithm findSmallestBST (root)

This algorithm finds the smallest node in a BST.

Pre root is a pointer to a nonempty BST or subtree

Return address of smallest node

1 if (left subtree empty)

 1 return (root)

2 end if

3 return findSmallestBST (left subtree)

end findSmallestBST

ALGORITHM

Find Largest Node in a BST

Algorithm findLargestBST (root)

This algorithm finds the largest node in a BST.

Pre root is a pointer to a nonempty BST or subtree

Return address of largest node returned

1 if (right subtree empty)

1 return (root)

2 end if

3 return findLargestBST (right subtree)

end findLargestBST

Make Empty :-

This operation is mainly for initialization when the programmer prefer to initialize the first element as a one - node tree.

ROUTINE TO MAKE AN EMPTY TREE :-

```
SearchTree MakeEmpty (SearchTree T)  
{  
if (T! = NULL)  
{  
MakeEmpty (T left);  
MakeEmpty (T Right);  
free (T);  
}  
return NULL ;  
}
```

Insert operation: -

To insert the element X into the tree,

- **Check with the root node T**
- **If it is less than the root,**
- **Traverse the left subtree recursively until it reaches the T left equals to NULL. Then X is placed in T left.**
- **If X is greater than the root.**
- **Traverse the right subtree recursively until it reaches the T right equals to NULL. Then x is placed in TRight.**

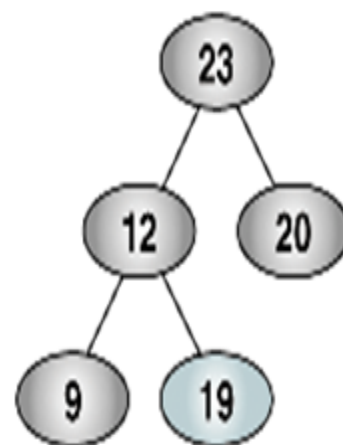
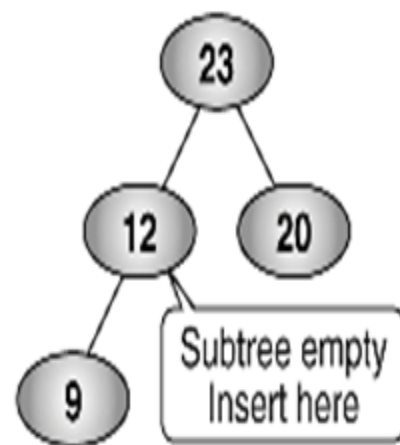
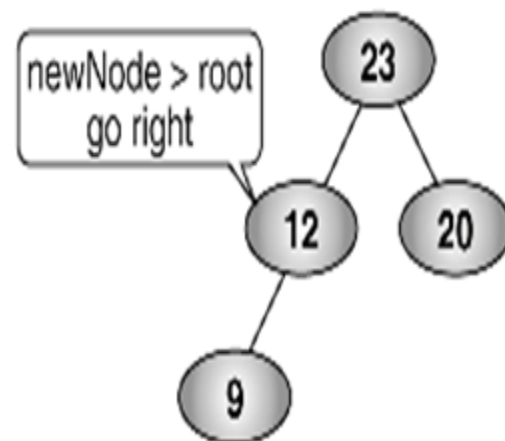
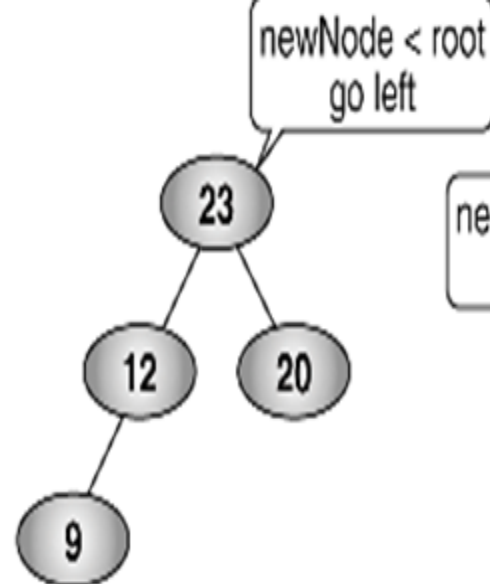
ROUTINE TO INSERT INTO A BINARY SEARCH TREE

SearchTree Insert (int X, searchTree T)

```
{  
if (T == NULL)  
{  
T = malloc (size of (Struct TreeNode));  
if(T != NULL)// First element is placed in  
the root.  
{  
T →Element = X;  
T→ left = NULL;  
T →Right = NULL;  
}} }
```

```
else  
if (X < T → Element)  
T left = Insert (X, T → left);  
else  
if (X > T → Element)  
T Right = Insert (X, T → Right);  
// Else X is in the tree already.  
return T;
```

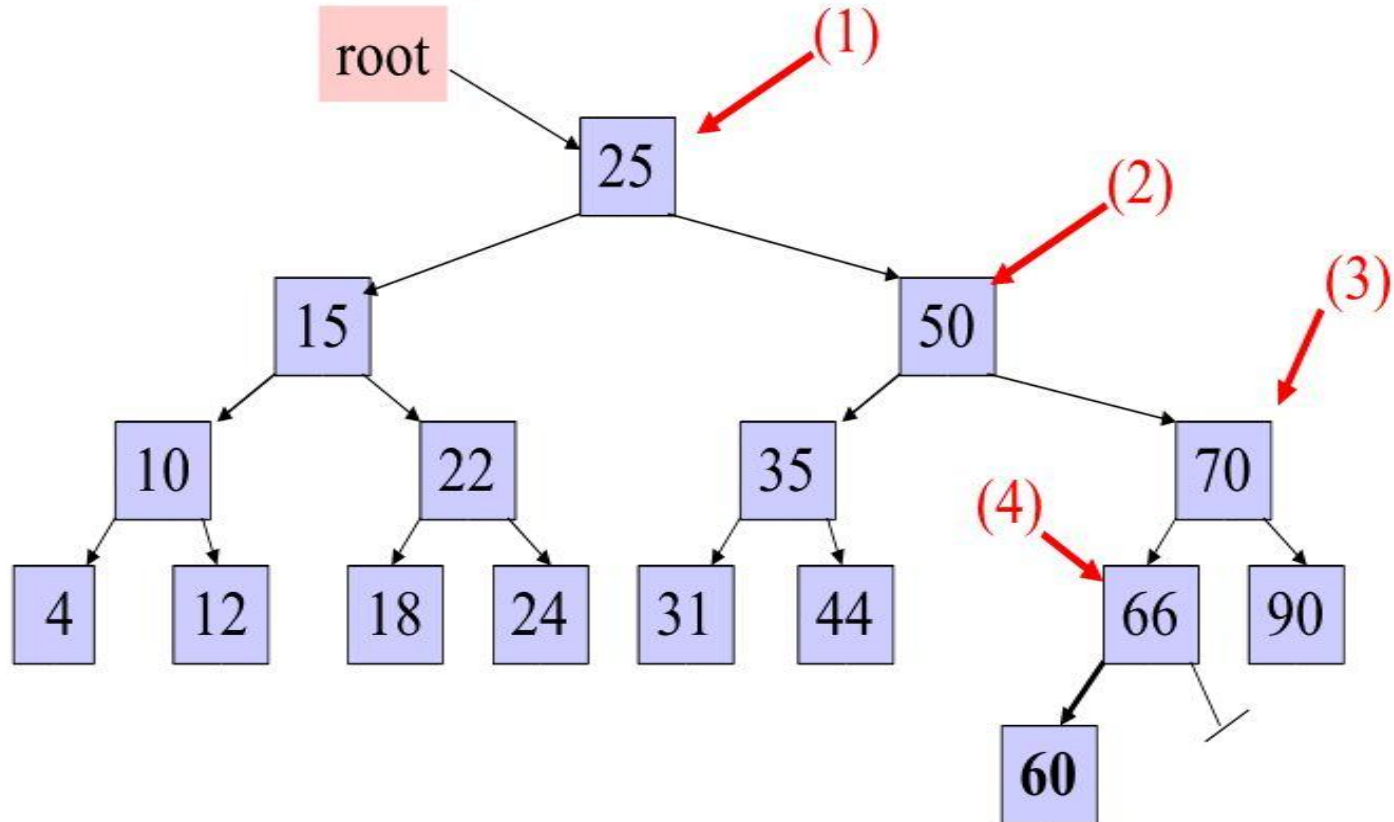
19
newNode

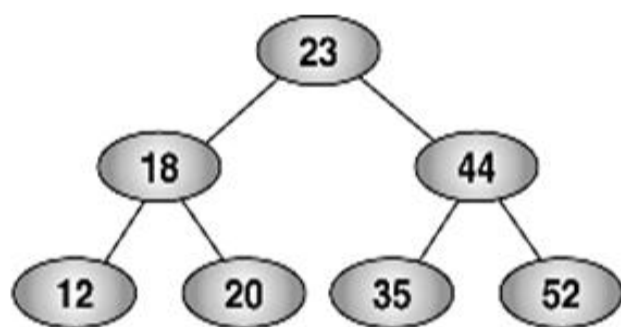


Trace of Recursive BST Insert

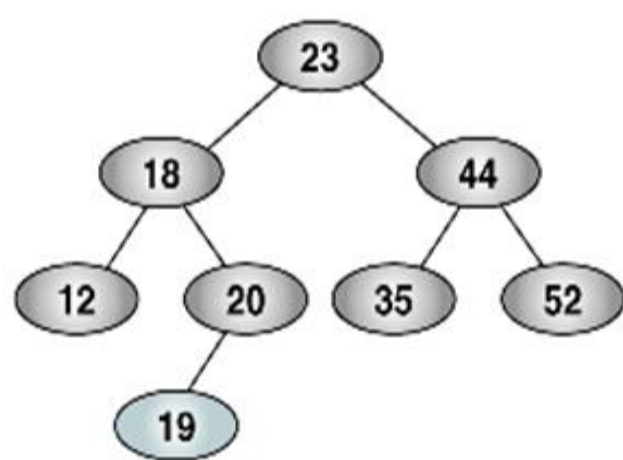
Example: insert 60 in the tree:

1. start at the root, 60 is greater than 25, search in right subtree
2. 60 is greater than 50, search in 50's right subtree
3. 60 is less than 70, search in 70's left subtree
4. 60 is less than 66, add 60 as 66's left child

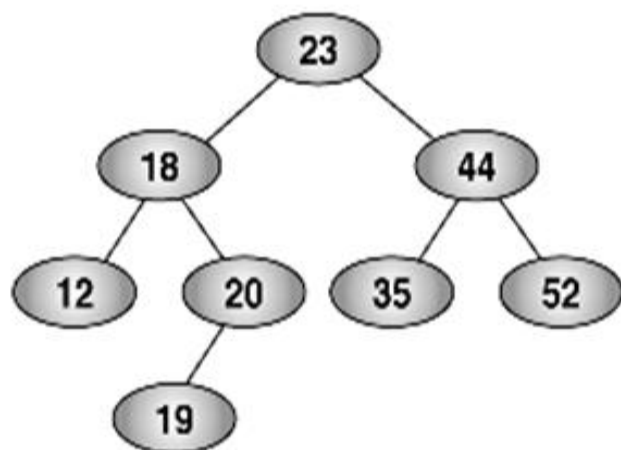




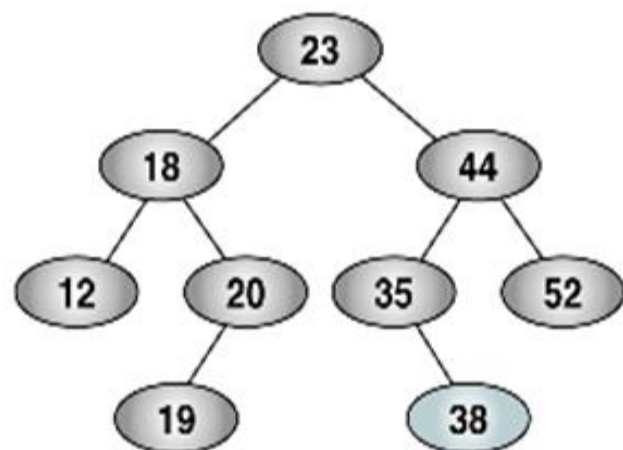
(a) Before inserting 19



(b) After inserting 19



(c) Before inserting 38



(d) After inserting 38

Add Node to BST

Algorithm addBST (root, newNode)

Insert node containing new data into BST using recursion.

Pre root is address of current node in a BST

 newNode is address of node containing data

Post newNode inserted into the tree

Return address of potential new tree root

1 if (empty tree)

1 set root to newNode

2 return newNode

2 end if

ALGORITHM

Add Node to BST (*continued*)

```
    Locate null subtree for insertion
3  if (newNode < root)
    1  return addBST (left subtree, newNode)
4  else
    1  return addBST (right subtree, newNode)
5  end if
end addBST
```

Delete operation:

- **Deletion operation is the complex operation in the Binary search tree. To delete an element, consider the following three possibilities.**

- **CASE 1 Node with no children (Leaf node)**

If the node is a leaf node, it can be deleted immediately.

- **CASE 2 : - Node with one child**

If the node has one child, it can be deleted by adjusting its parent pointer that points to its child node

- **Case 3 : Node with two children**

It is difficult to delete a node which has two children. The general strategy is to replace the data of the node to be deleted with its smallest data of the right subtree and recursively delete that node.

DELETION ROUTINE FOR BINARY SEARCH TREES

```
SearchTree Delete (int X, searchTree T)  
{  
int Tmpcell ;  
if (T == NULL)  
Error ("Element not found");  
else  
if (X < T →Element) // Traverse towards left  
T →Left = Delete (X, T Left);  
else  
if (X > T Element) // Traverse towards right  
T →Right = Delete (X, T →Right);  
// Found Element to be deleted
```

```

else // Two children
if (T→ Left && T→ Right)
{ // Replace with smallest data in right subtree
Tmpcell = FindMin (T→ Right);
T →Element = Tmpcell Element ;
T →Right = Delete (T →Element; T →Right);
}
else { // one or zero children
Tmpcell = T;
if (T →Left == NULL)
T = T→ Right;
else if (T→ Right == NULL)
T = T →Left ;
free (TmpCell);
}
return T; }

```

Delete node from BST

Algorithm deleteBST (root, dltKey)

This algorithm deletes a node from a BST.

Pre root is reference to node to be deleted
 dltKey is key of node to be deleted

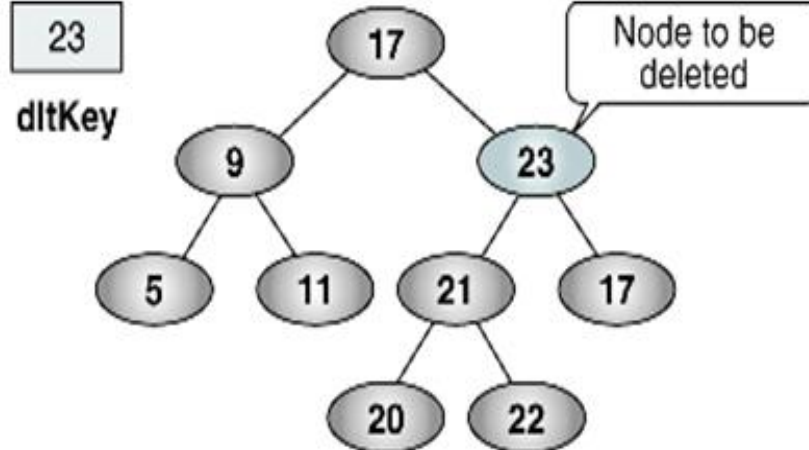
Post node deleted
 if dltKey not found, root unchanged

Return true if node deleted, false if not found

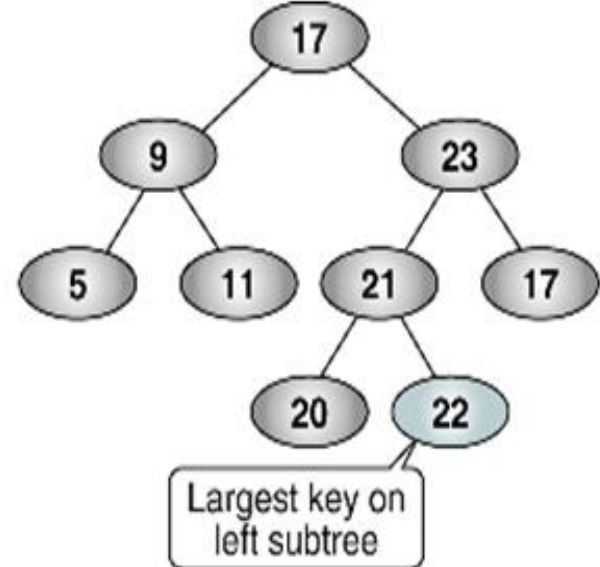
```
1 if (empty tree)
  1 return false
2 end if
3 if (dltKey < root)
  1 return deleteBST (left subtree, dltKey)
4 else if (dltKey > root)
  1 return deleteBST (right subtree, dltKey)
5 else
  Delete node found--test for leaf node
  1 If (no left subtree)
    1 make right subtree the root
    2 return true
```

Delete node from BST (continued)

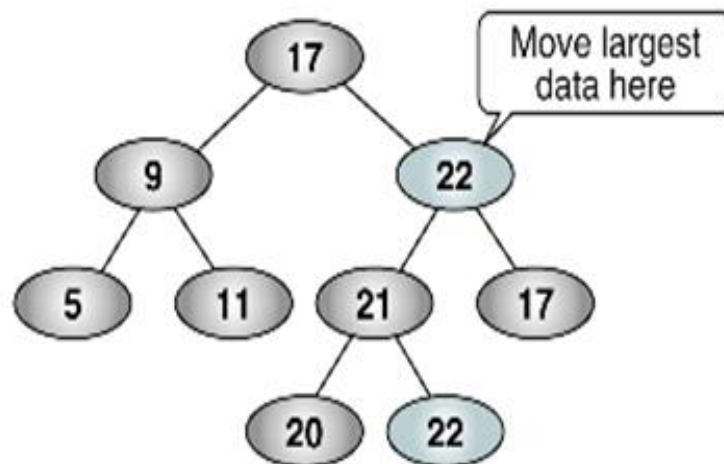
```
2  else if (no right subtree)
    1  make left subtree the root
    2  return true
3  else
    Node to be deleted not a leaf. Find largest node on
    left subtree.
    1  save root in deleteNode
    2  set largest to largestBST (left subtree)
    3  move data in largest to deleteNode
    4  return deleteBST (left subtree of deleteNode,
                        key of largest
4  end if
6 end if
end deleteBST
```

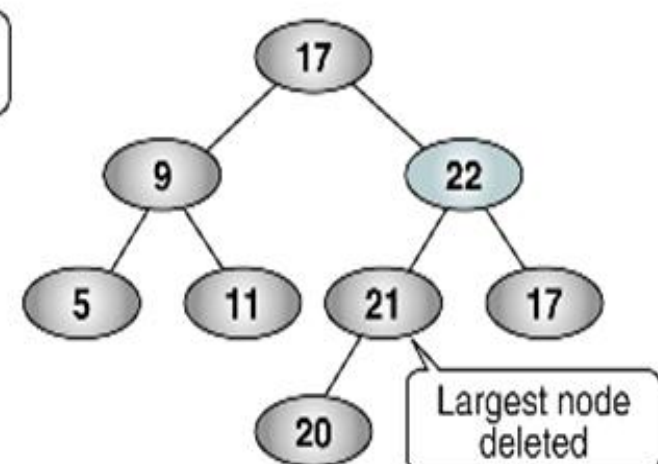
(a) Find dlitKey



(b) Find largest



(c) Move largest data



(d) Delete largest node

AVL Trees

These are self-adjusting, height-balanced binary search trees and are named after the inventors: Adelson-Velskii and Landis.

Definition:

The height of a binary tree is the maximum path length from the root to a leaf. A single-node binary tree has height 0, and an empty binary tree has height -1

- An **AVL tree** is a binary search tree in which every node is **height balanced**, that is, the difference in the heights of its two subtrees is at most 1.
- The **balance factor** of a node is the height of its right subtree minus the height of its left subtree. An equivalent definition, then, for an AVL tree is that it is a binary search tree in which each node has a balance factor of -1, 0, or +1.
- Note :balance factor of -1 means that the subtree is left-heavy, and
- a balance factor of +1 means that the subtree is right-heavy.

AVL Trees

These are self-adjusting, height-balanced binary search trees and are named after the inventors: Adelson-Velskii and Landis.

Definition:

The height of a binary tree is the maximum path length from the root to a leaf. A single-node AVLtree has height 0, and an empty AVL tree has height -1

AVL Tree

Definition

- Binary Search tree.
- If T is a nonempty binary Search tree with T_L and T_R as its left and right subtrees, then T is an AVL tree iff
 1. T_L and T_R are AVL trees, and
 2. $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R , respectively

AVL Tree

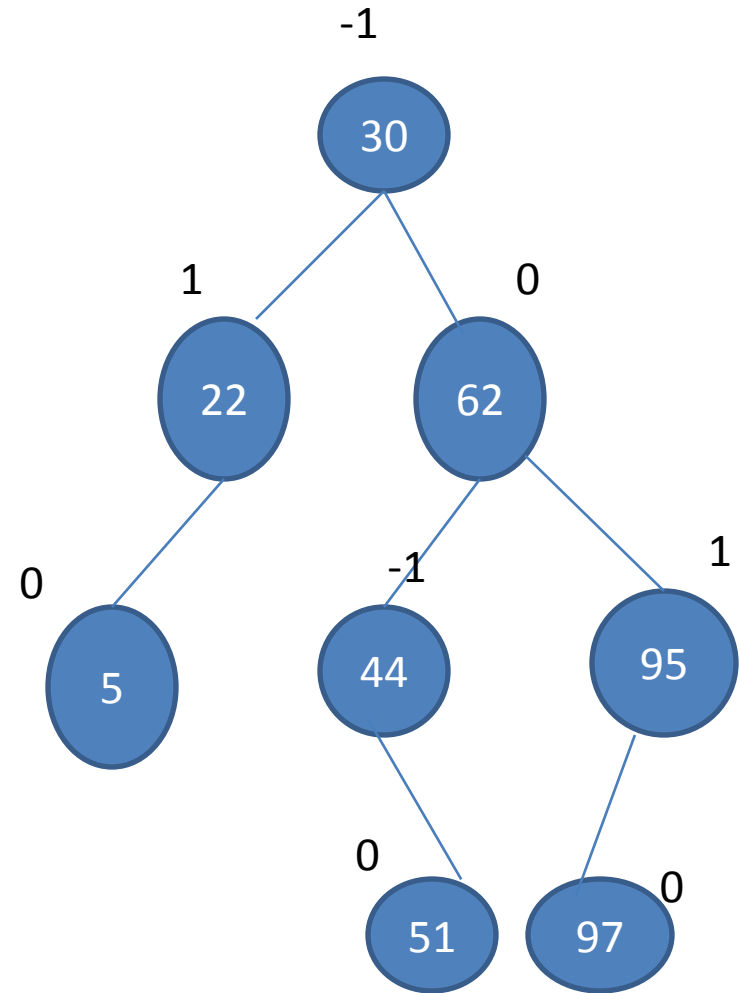
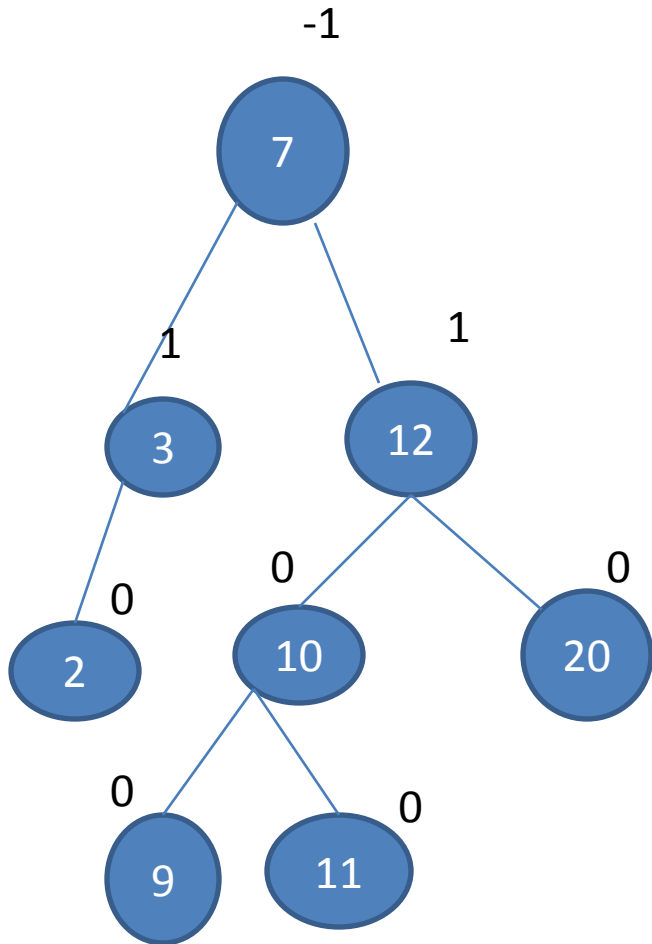
Definition

- Binary tree.
- If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is an AVL tree iff
 1. T_L and T_R are AVL trees, and
 2. $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R , respectively

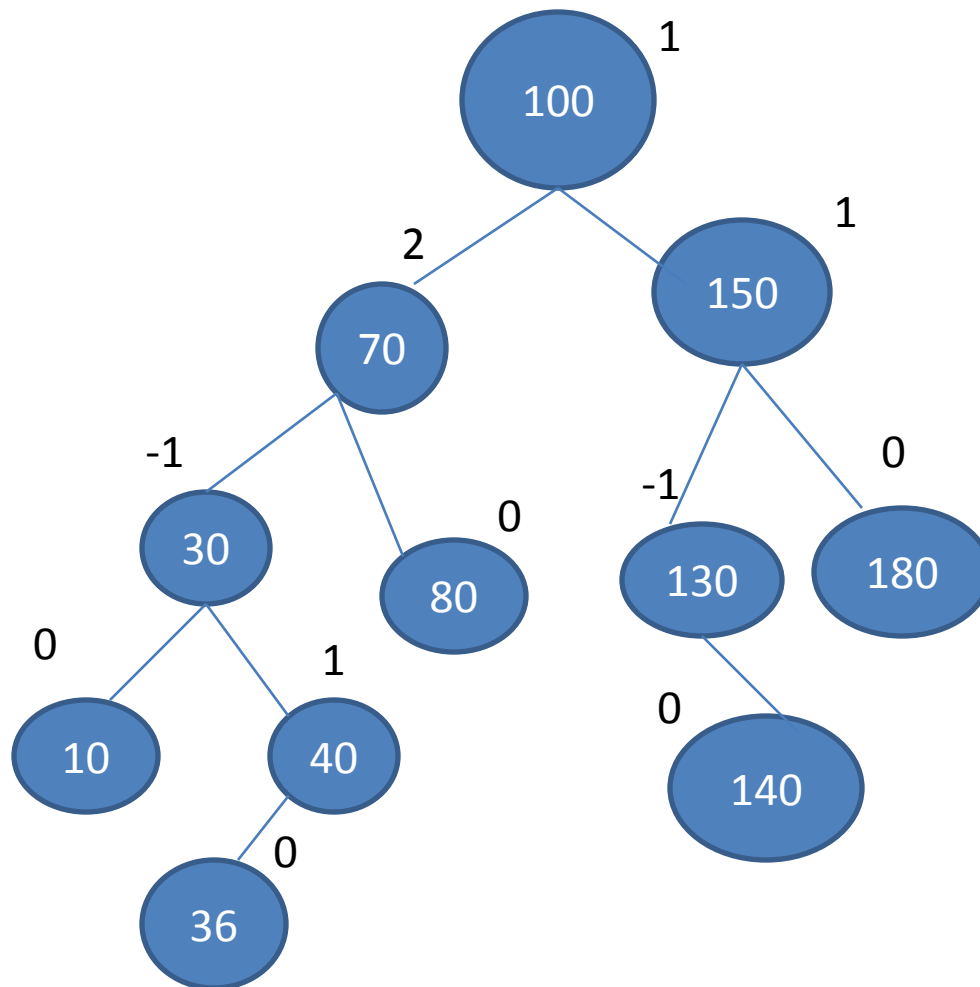
Balance Factor

- AVL trees are normally represented using the linked representation
- To facilitate insertion and deletion, a **balance factor (bf)** is associated with each node.
- The balance factor $bf(x)$ of a node x is defined as
 $height(x \rightarrow leftChild) - height(x \rightarrow rightChild)$
- Balance factor of each node in an AVL tree must be **-1, 0, or 1**

Eg with balance factors



Not an AVL TREE



AVL tree structure in C

For each node, the difference of height between left and right are no more than 1.

```
struct AVLnode_s {  
    Datatype element;  
    struct AVLnode *left;  
    struct AVLnode *right;  
};  
typedef struct AVLnode_s AVLnode;
```

Inserting into an AVL Search Trees

- If we insert an element into an AVL search tree, the result may not be an AVL tree
- That is, the tree may become **unbalanced**
- If the tree becomes unbalanced, we must adjust the tree to restore balance - this adjustment is called **rotation**.
- There are Four Models of rotations:
- There are four models about the operation of AVL Tree:
 1. **LL**: new node is in the **left subtree of the left subtree** of A
 2. **LR**: new node is in the **right subtree of the left subtree** of A
 3. **RR**: new node is in the **right subtree of the right subtree** of A
 4. **RL**: new node is in the **left subtree of the right subtree** of A

Rotation

Definition

- To switch *children* and *parents* among two or three adjacent nodes to **restore balance of a tree**.
- A rotation may change the depth of some nodes, but does not change their relative ordering.

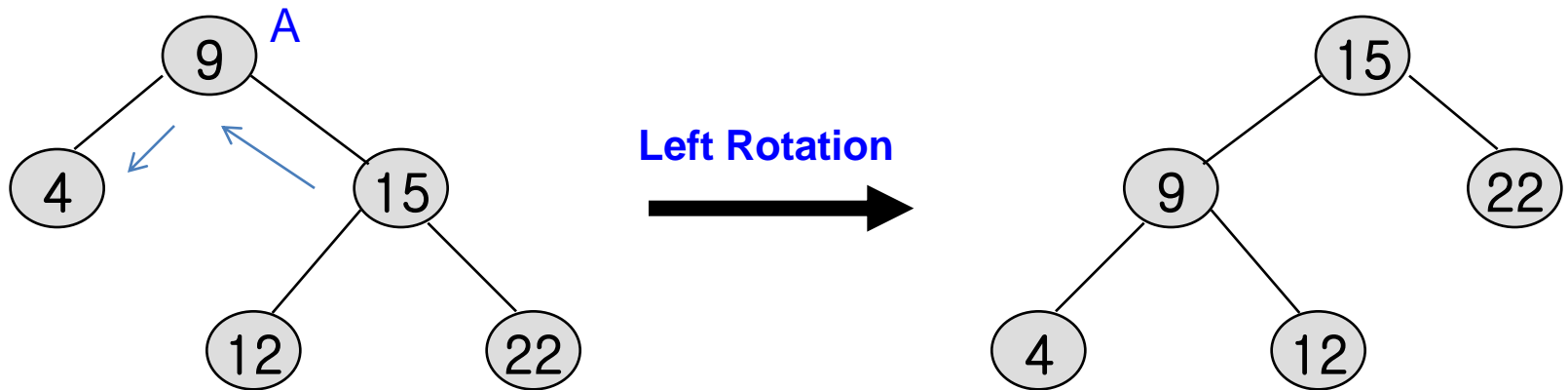
Single and Double Rotations

- **Single rotations**: the transformations done to correct LL and RR imbalances
- **Double rotations**: the transformations done to correct LR and RL imbalances
- The transformation **to correct LR imbalance** can be achieved by an **RR rotation followed by an LL rotation**
- The transformation **to correct RL imbalance** can be achieved by an **LL rotation followed by an RR rotation**

Left Rotation

Definition

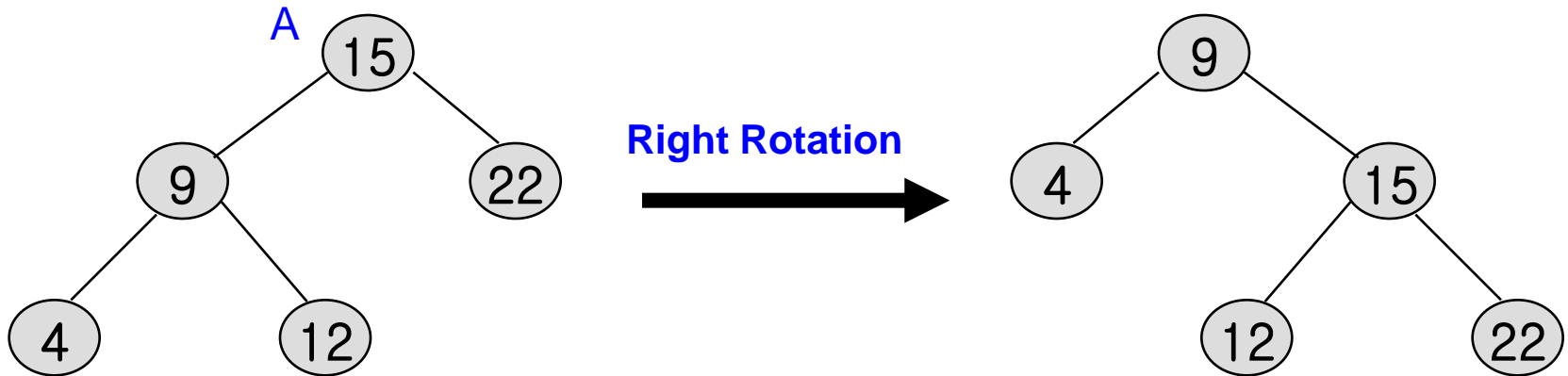
- In a **binary search tree**, pushing a node A down and to the left to balance the tree.
- A's right child replaces A, and the right child's left child becomes A's right child.



Right Rotation

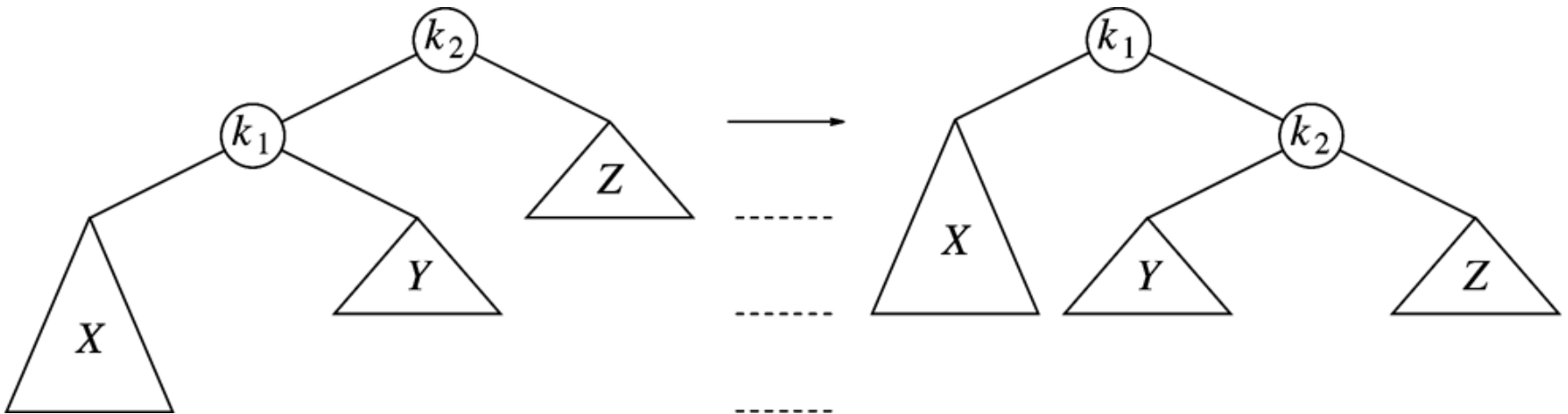
Definition

- In a **binary search tree**, pushing a node **A** down and to the **right** to **balance** the tree.
- A's left child replaces A, and the left child's right child becomes A's left child.



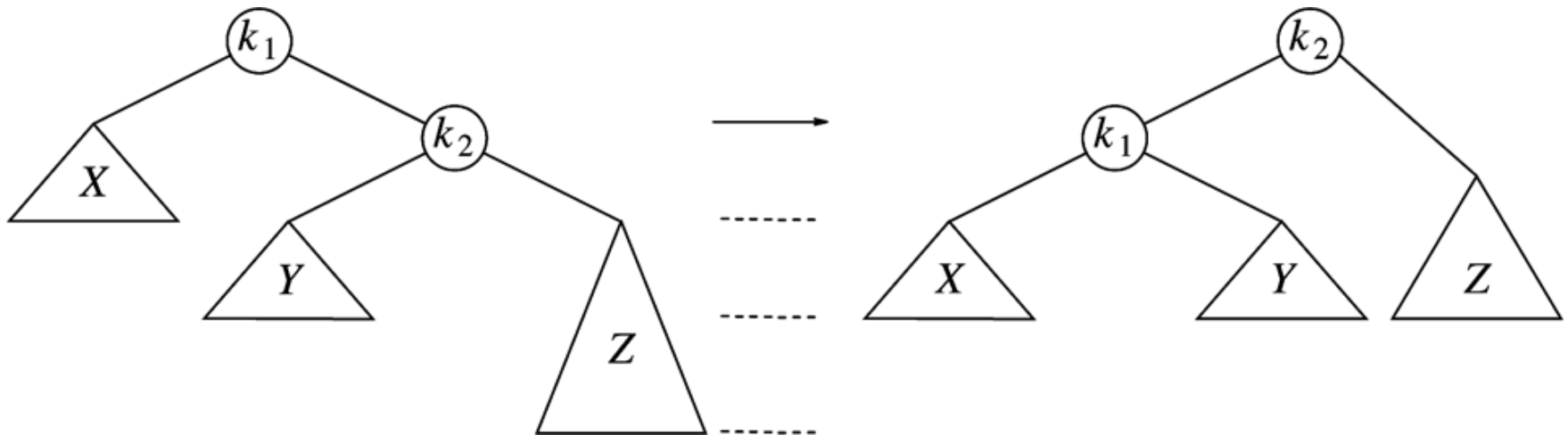
Single Rotation (LL)

- Let k_2 be the first node on the path up violating AVL balance property. Figure below is the only possible scenario that allows k_2 to satisfy the AVL property before the insertion but violate it afterwards. Subtree X has grown an extra level (2 levels deeper than Z now). Y cannot be at the same level as X (k_2 then out of balance before insertion) and Y cannot be at the same level as Z (then k_1 would be the first to violate).



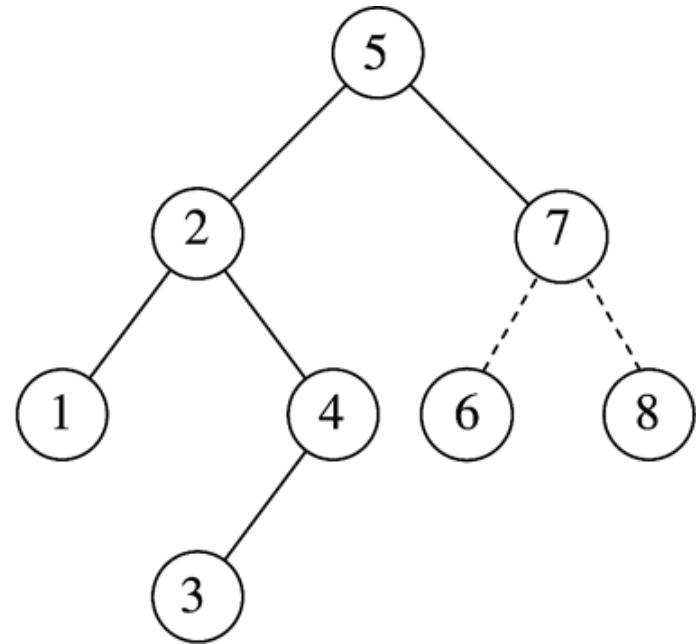
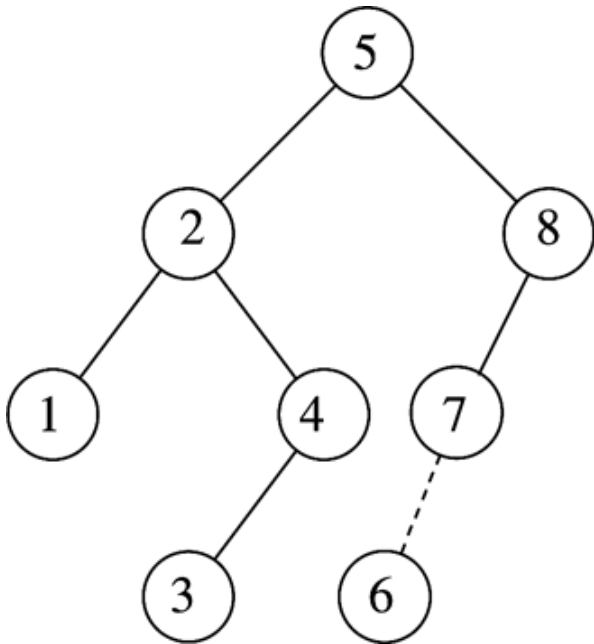
Single Rotation (RR)

- Note that in single rotation inorder traversal orders of the nodes are preserved.
- The new height of the subtree is exactly the same as before. Thus no further updating of the nodes on the path to the root is needed.



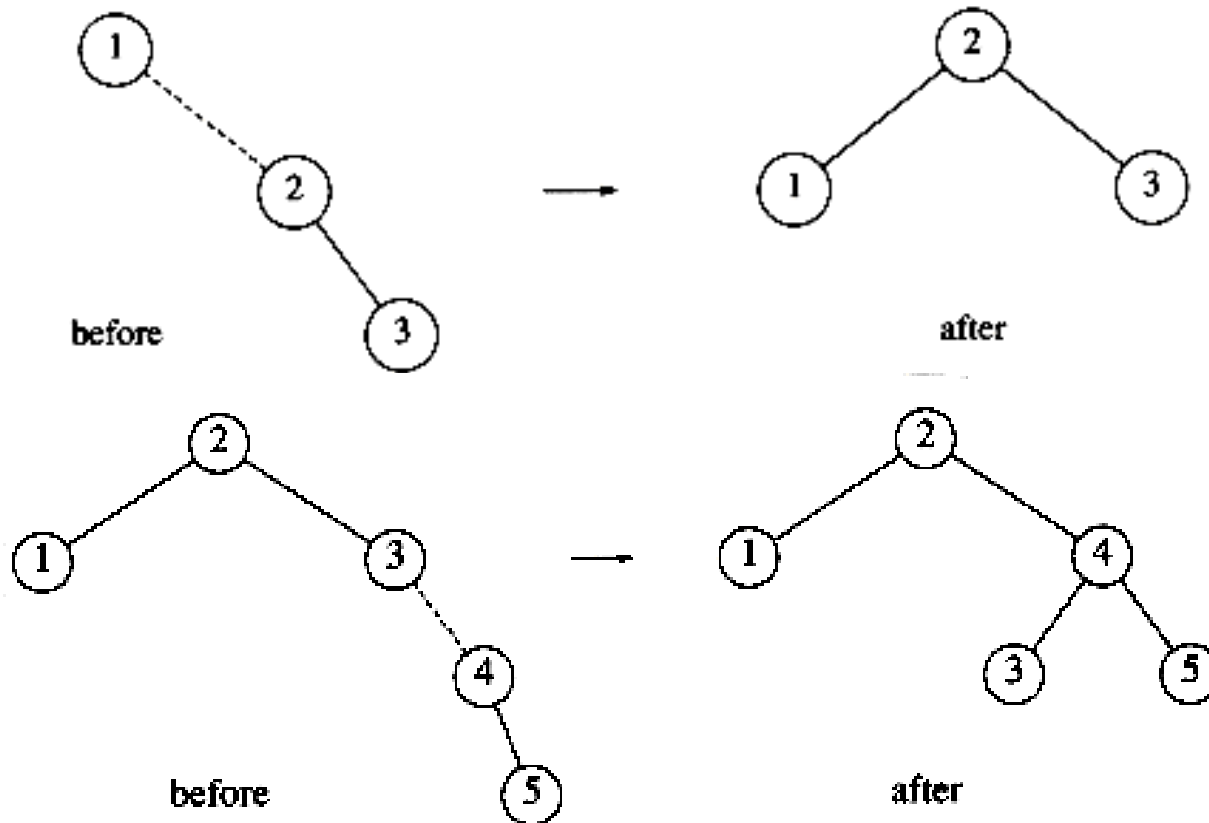
Single Rotation-Example I

- AVL property destroyed by insertion of 6, then fixed by a single rotation.
- BST node structure needs an additional field for height.



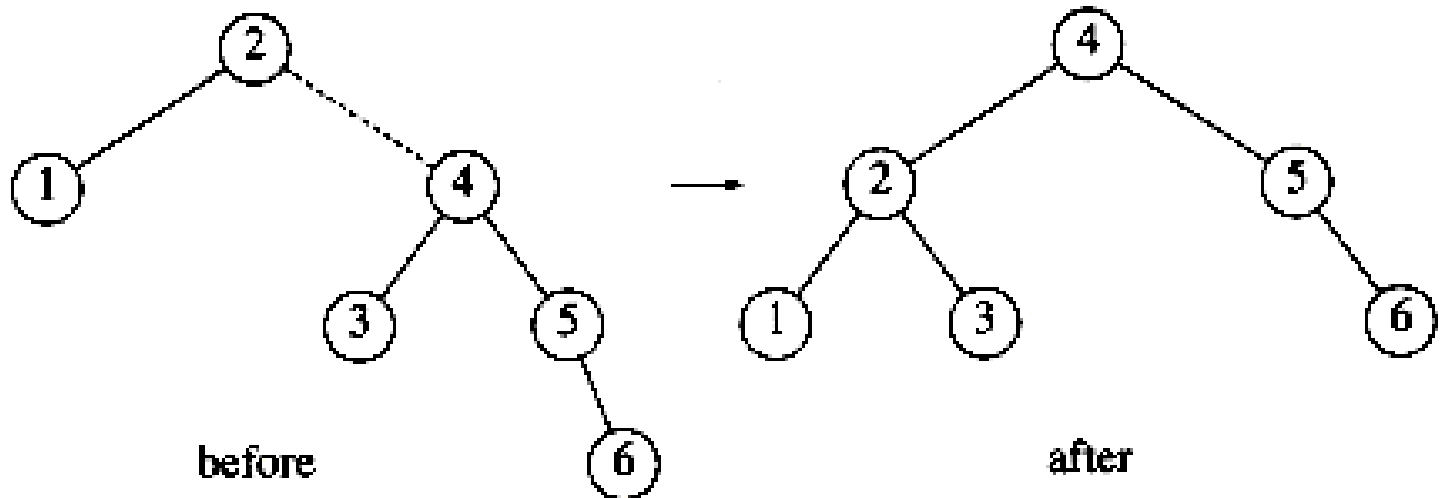
Single Rotation-Example II

- Start with an initially empty tree and insert items 1 through 7 sequentially. Dashed line joins the two nodes that are the subject of the rotation.

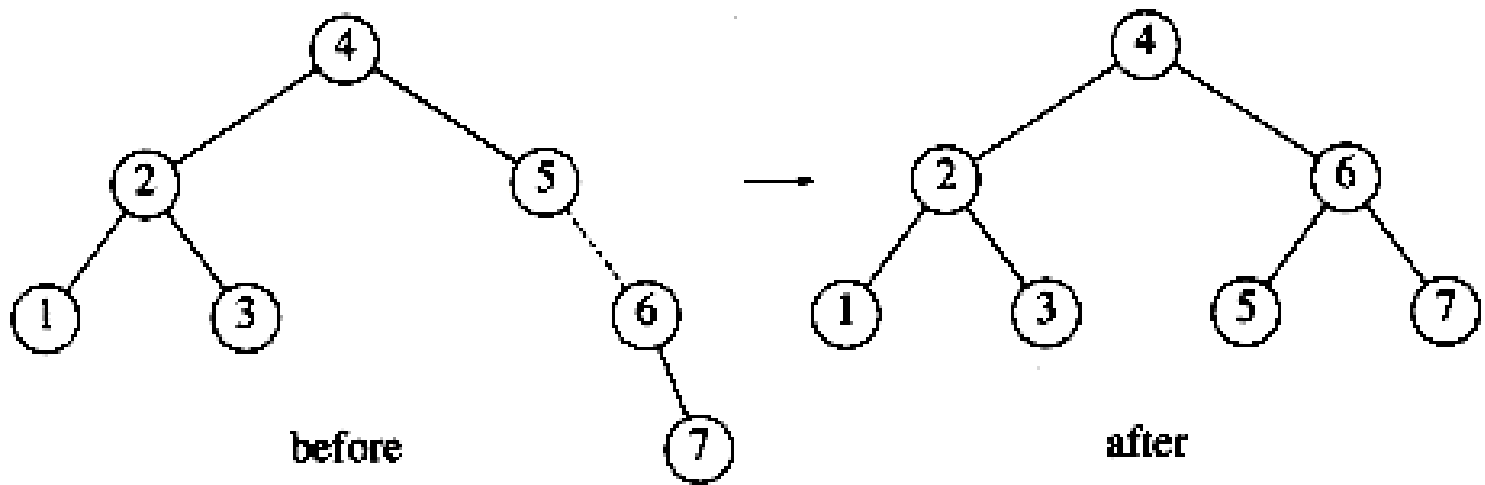


Single Rotation-Example III

Insert 6.
Balance
problem at th
root. So a
single rotator
is performed.

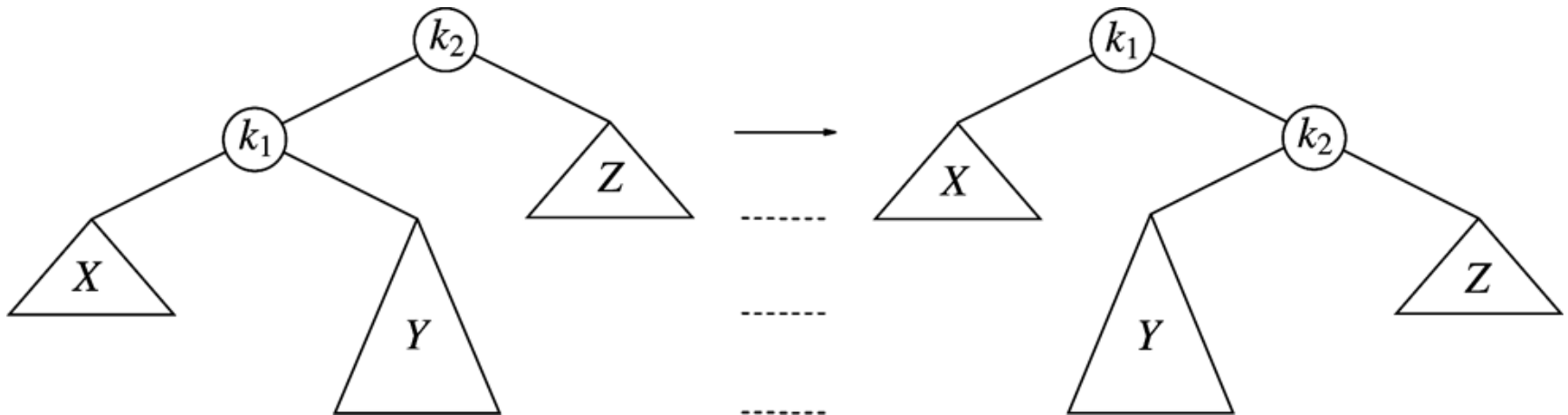


Finally, Insert
7 causing
another
rotation.



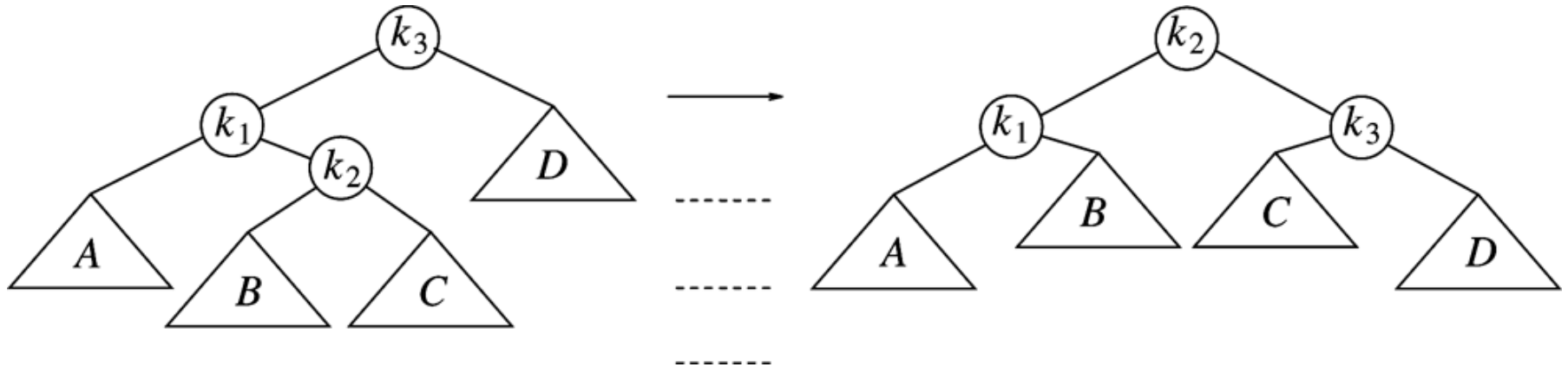
Double Rotation (LR, RL) - I

- The algorithm that works for cases 1 and 4 (LL, RR) does not work for cases 2 and 3 (LR, RL). The problem is that subtree Y is too deep, and a single rotation does not make it any less deep.
- The fact that subtree Y has had an item inserted into it guarantees that it is nonempty. Assume it has a root and two subtrees.



Double Rotation (LR) - II

Below are 4 subtrees connected by 3 nodes. Note that exactly one of tree B or C is 2 levels deeper than D (unless all empty). To rebalance, k_3 cannot be root and a rotation between k_1 and k_3 was shown not to work. So the only alternative is to place k_2 as the new root. This forces k_1 to be k_2 's left child and k_3 to be its right child. It also completely determines the locations of all 4 subtrees. AVL balance property is now satisfied. Old height of the tree is restored; so, all the balancing and height updating is complete.

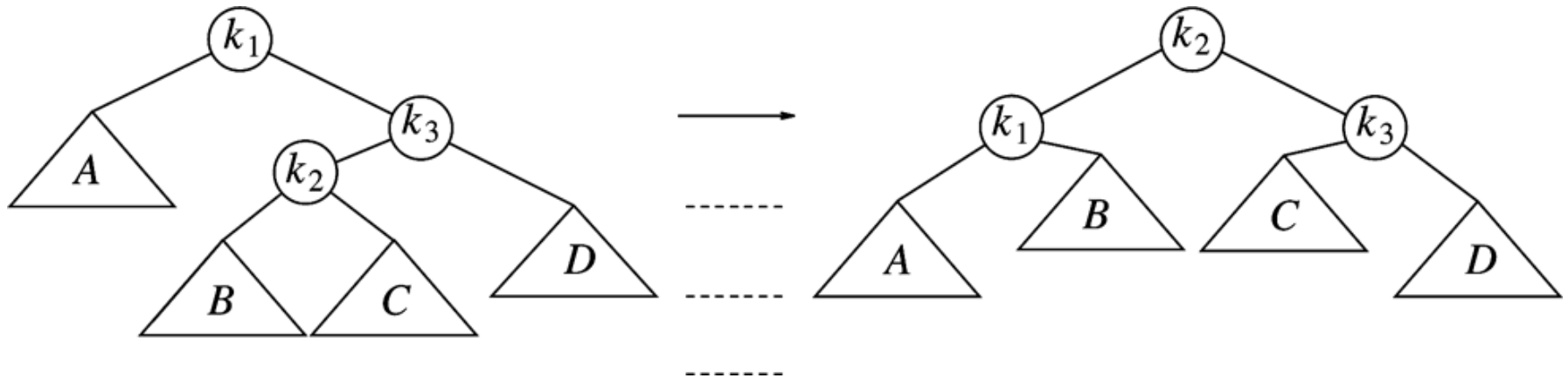


Double Rotation (RL) - III

In both cases (LR and RL), the effect is the same as rotating between α 's child and grandchild and then between α and its new child. Every double rotation can be modelled in terms of 2 single rotations. Inorder traversal orders are always preserved between k_1 , k_2 , and k_3 .

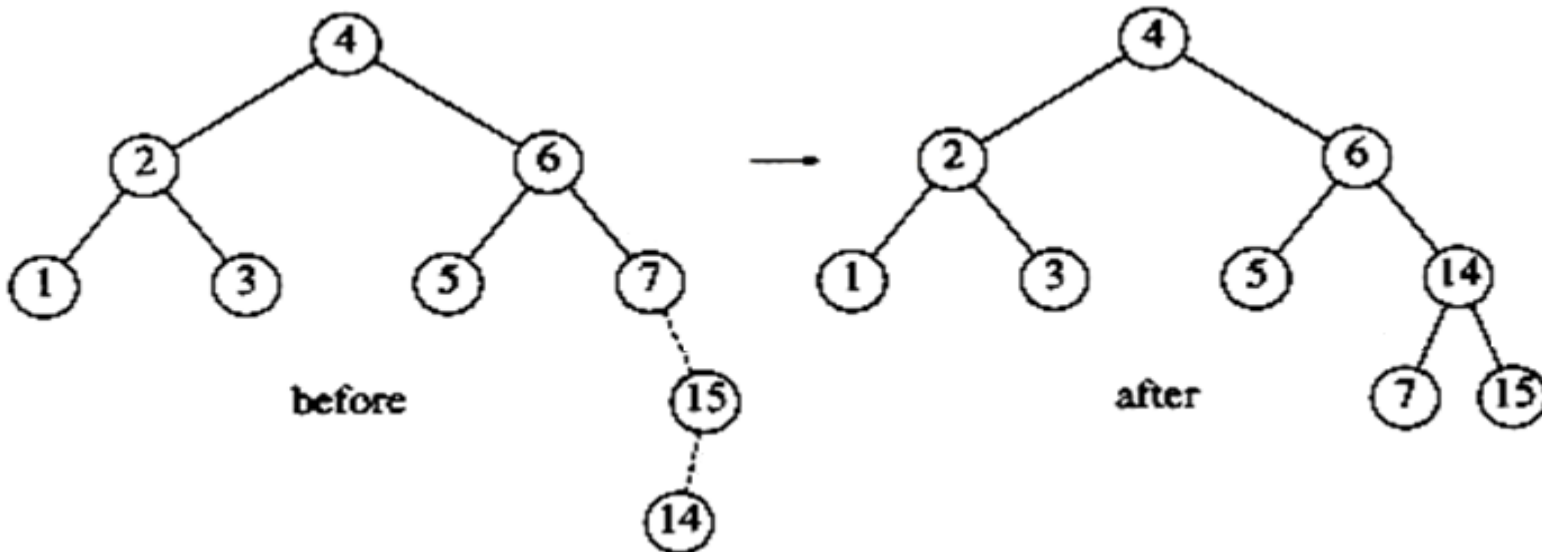
Double RL = Single LL ($\alpha \rightarrow \text{right}$) + Single RR (α)

Double LR = Single RR ($\alpha \rightarrow \text{left}$) + Single LL (α)



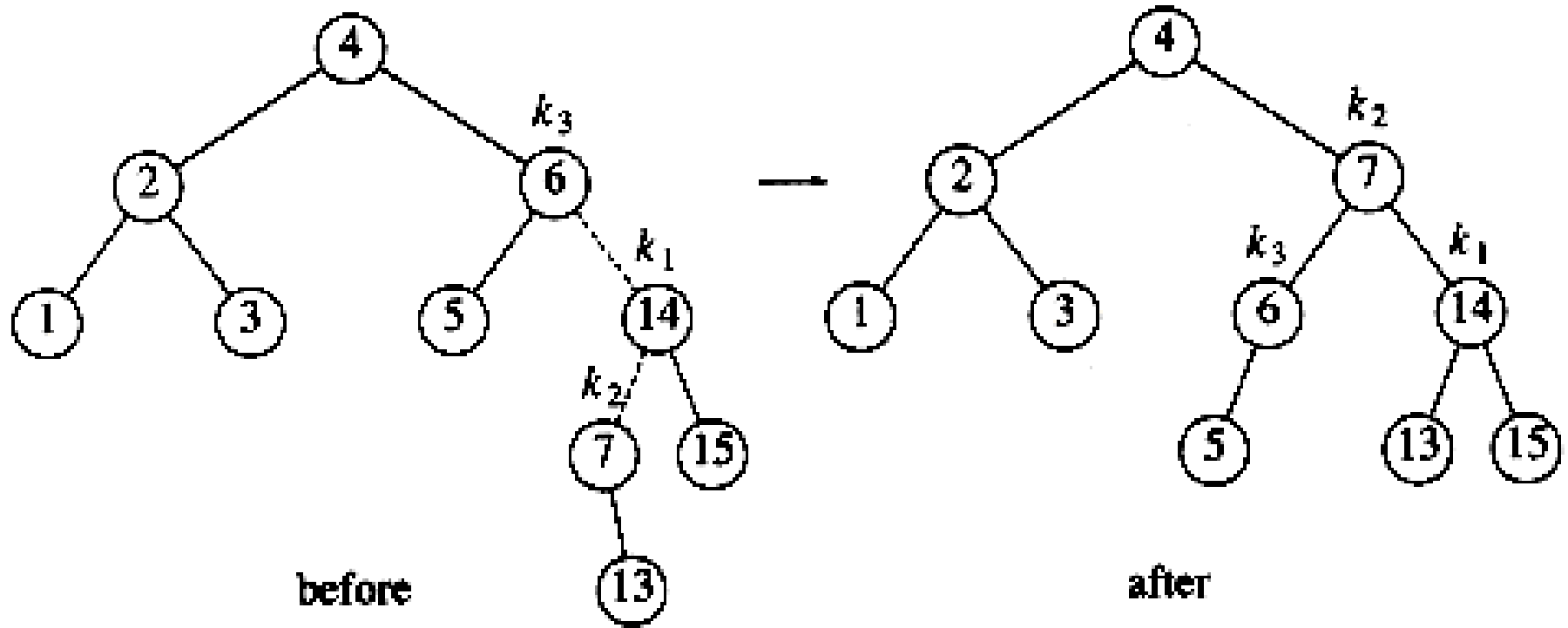
Double Rotation Example - I

- Continuing our example, suppose **keys 8 through 15** are inserted in reverse order. Inserting 15 is easy but inserting 14 causes a height imbalance at node 7. The double rotation is an RL type and involves 7, 15, and 14.



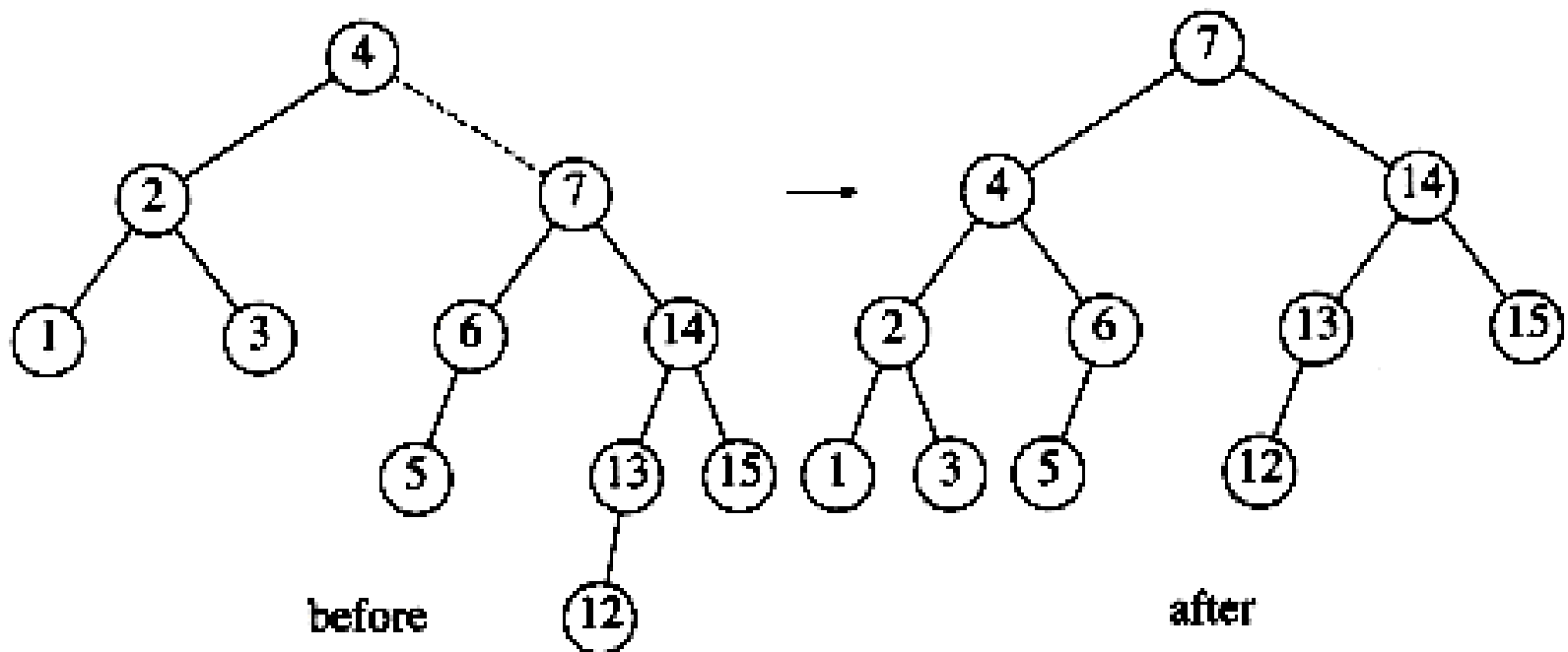
Double Rotation Example - II

- insert 13: double rotation is RL that will involve 6, 14, and 7 and will restore the tree.



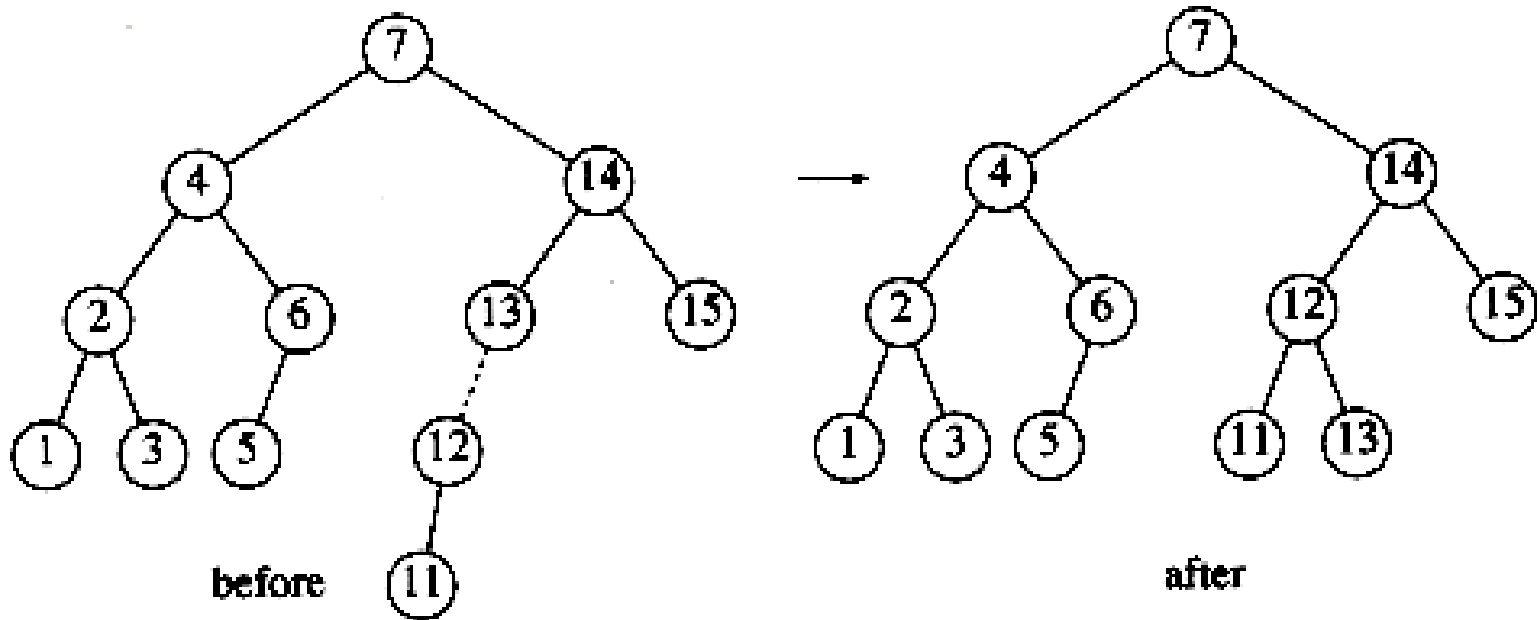
Double Rotation Example - III

- If 12 is now inserted, there is an imbalance at the root. Since 12 is not between 4 and 7, we know that the single rotation RR will work.



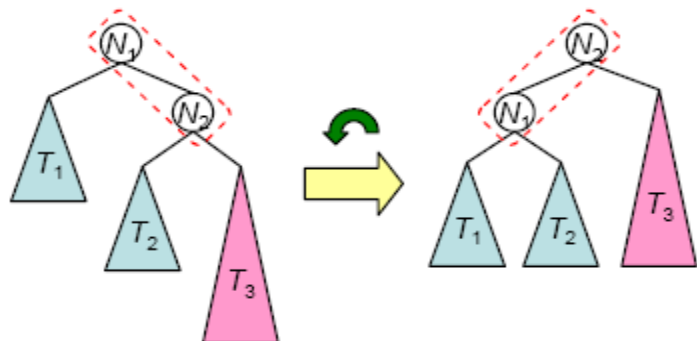
Double Rotation Example - IV

- Insert 11: single rotation LL; insert 10: single rotation LL; insert 9: single rotation LL; insert 8: without a rotation.



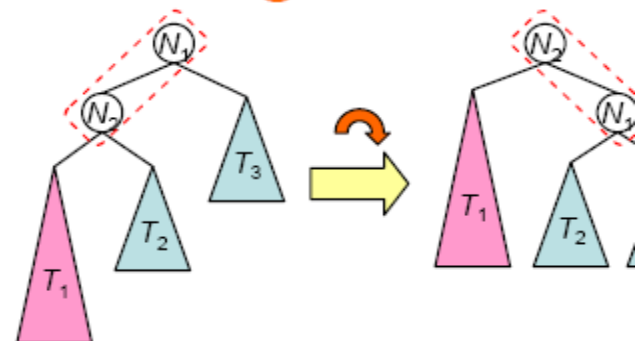
Case 1: insertion to *right* subtree of *right* child

Solution: *Left* rotation



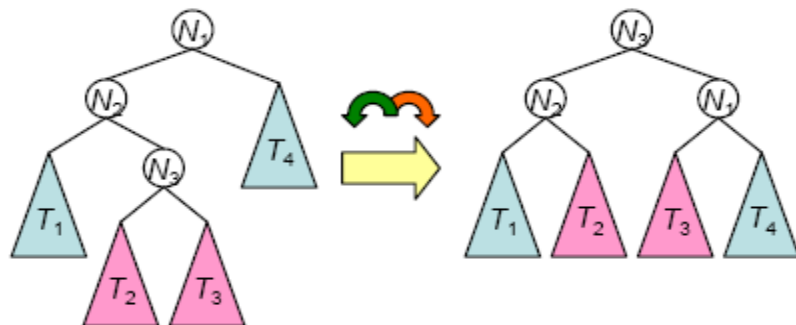
Case 2: insertion to *left* subtree of *left* child

Solution: *Right* rotation



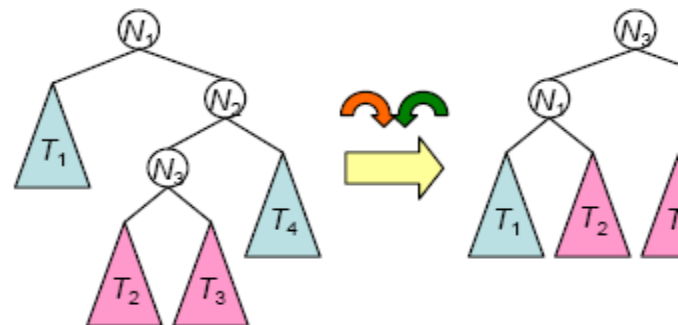
Case 3: insertion to *right* subtree of *left* child

Solution: *Left-right* rotation



Case 4: insertion to *left* subtree of *right* child

Solution: *Right-left* rotation

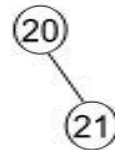
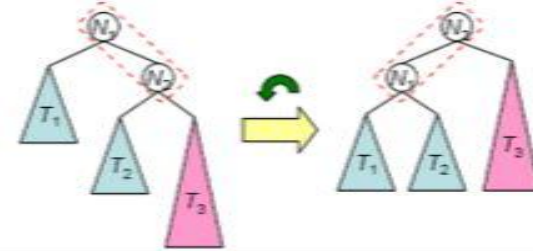


examples

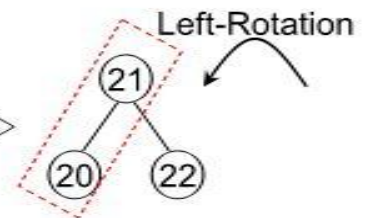
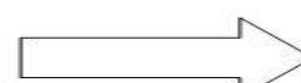
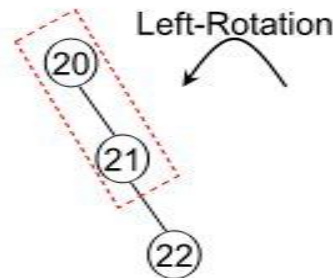
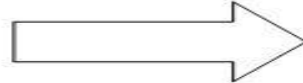
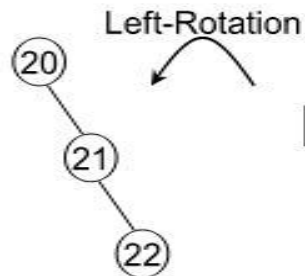
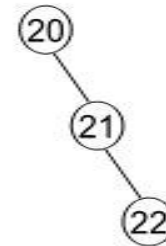
Left-Rotation

Case 1: insertion to *right*
subtree of *right* child

Solution: *Left* rotation



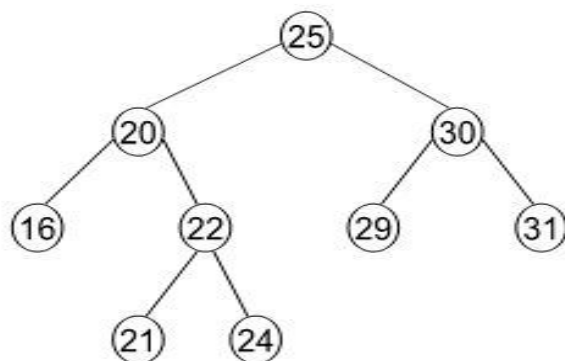
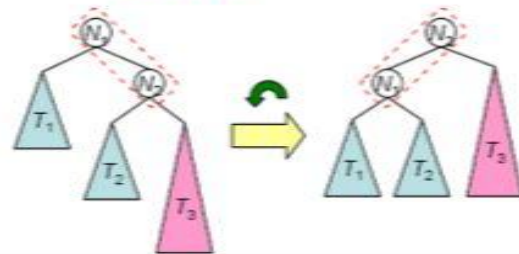
Add 22



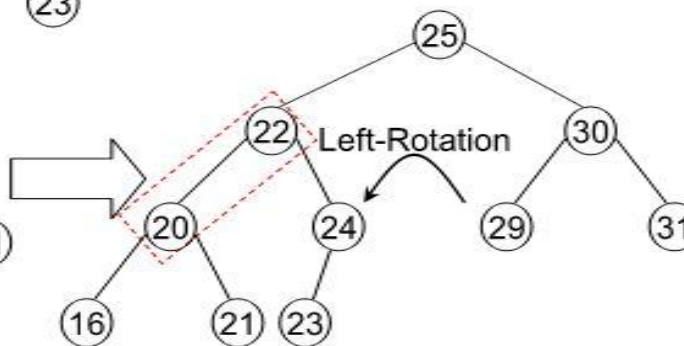
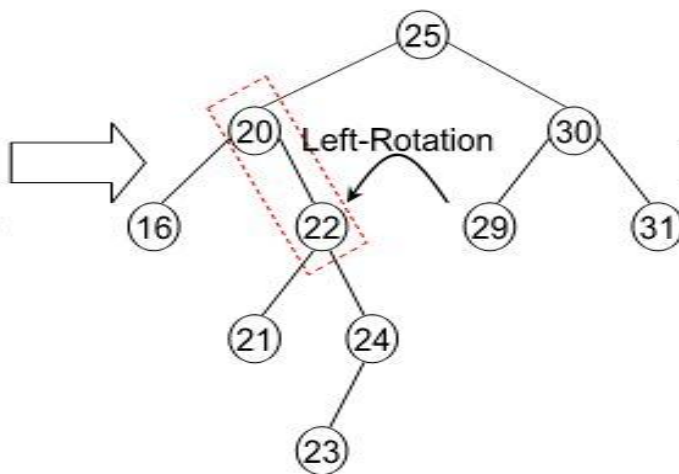
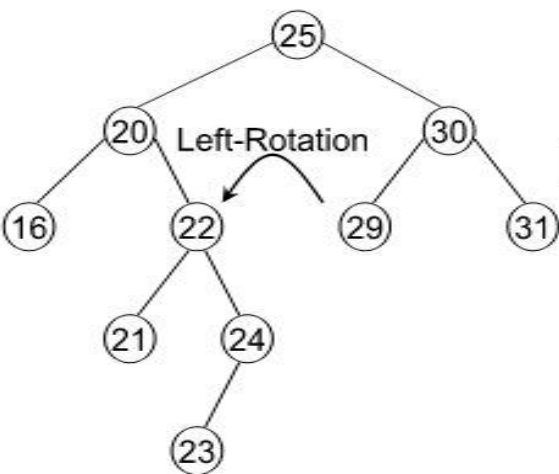
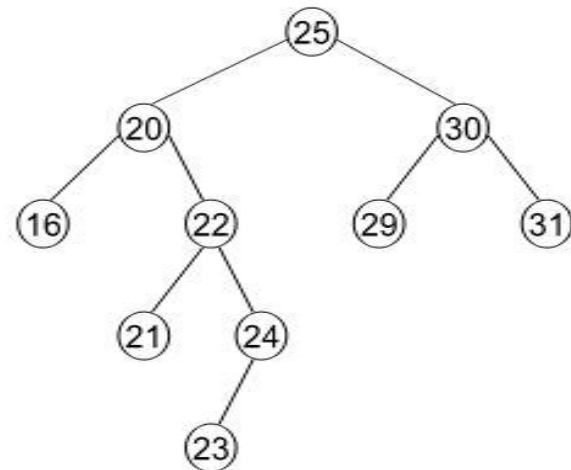
Left-Rotation

Case 1: insertion to *right* subtree of *right* child

Solution: *Left* rotation



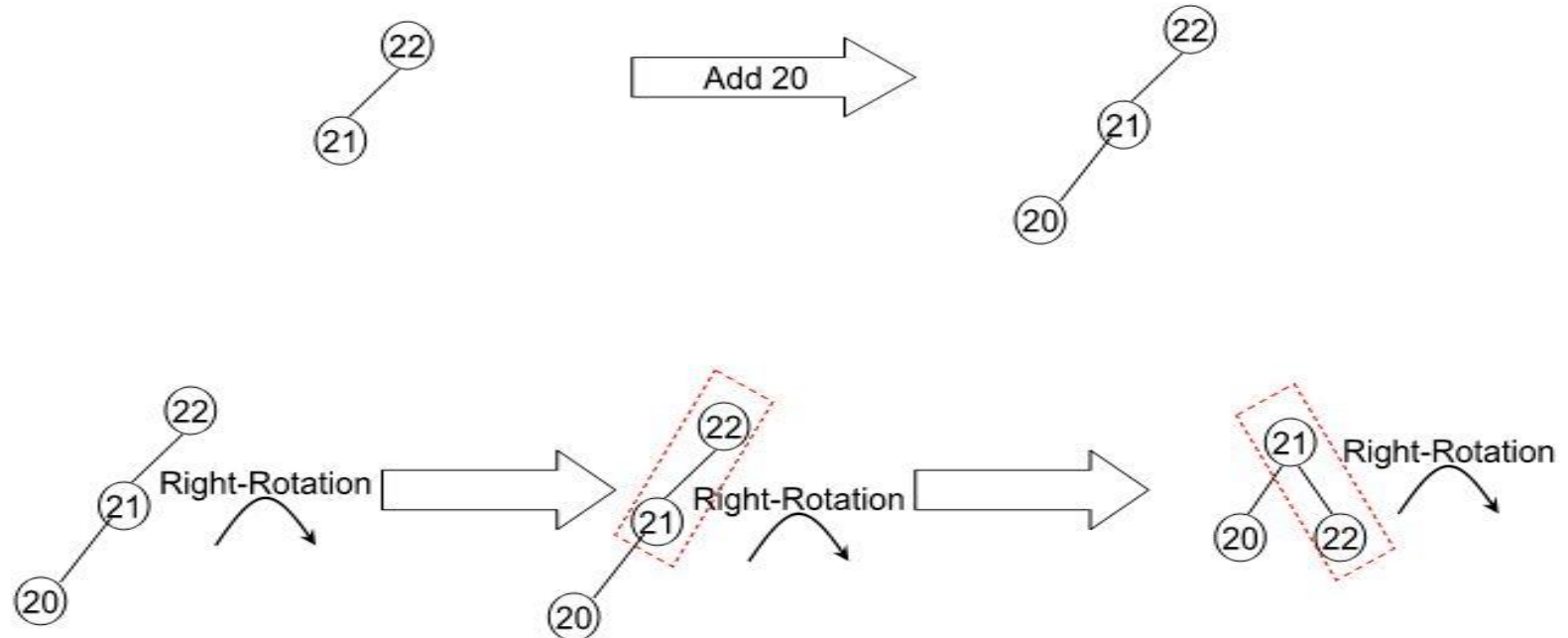
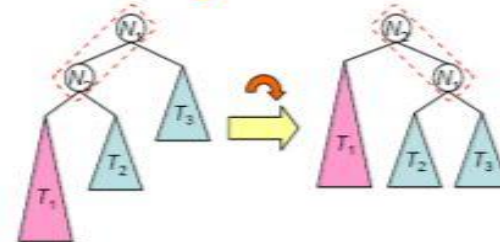
Add 23



Right-Rotation

Case 2: insertion to *left* subtree of *left* child

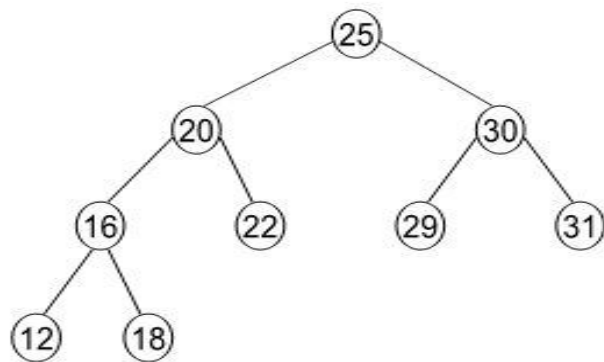
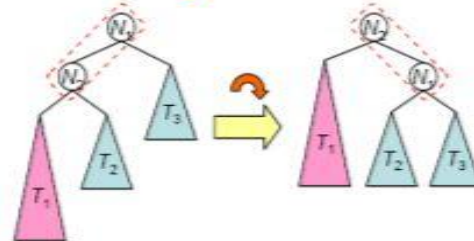
Solution: *Right* rotation



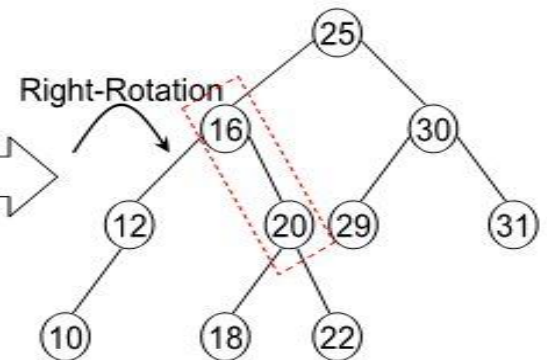
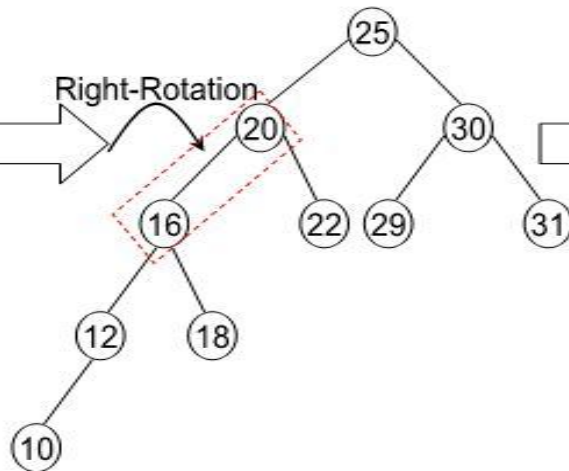
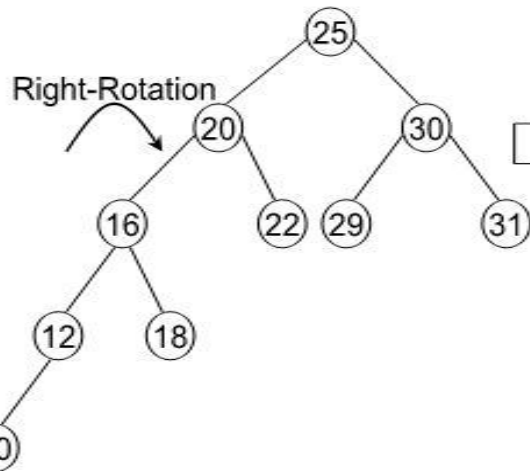
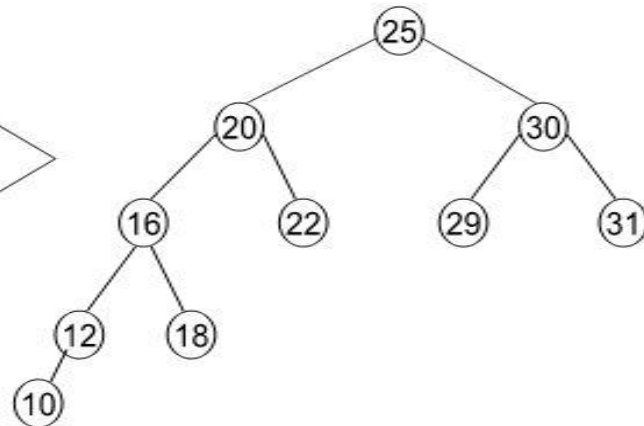
Right-Rotation

Case 2: insertion to *left* subtree of *left* child

Solution: *Right* rotation



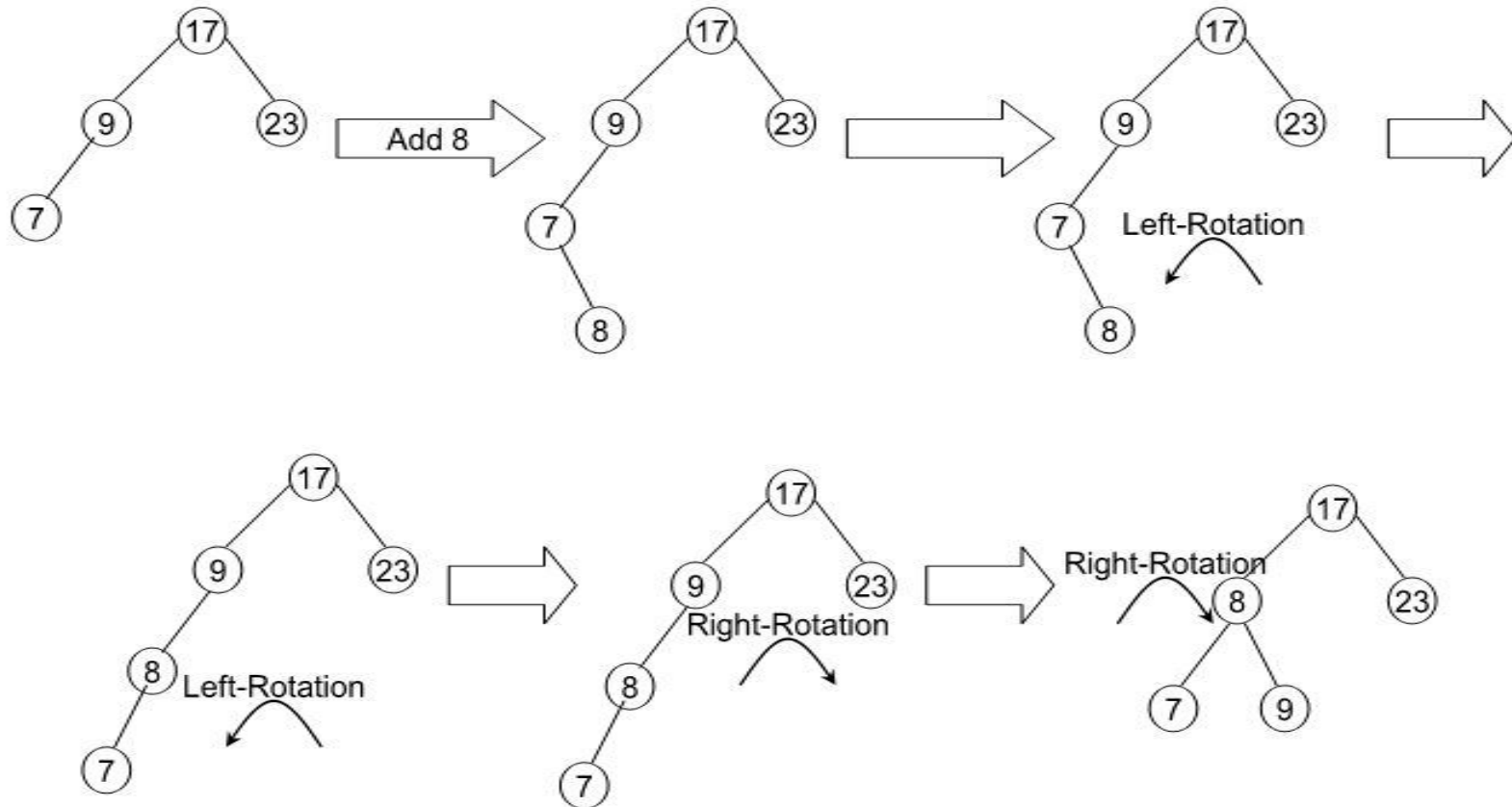
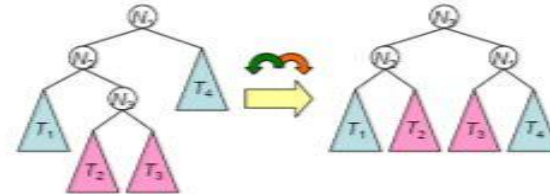
Add 10



Left-Right Rotation

Case 3: insertion to *right* subtree of *left* child

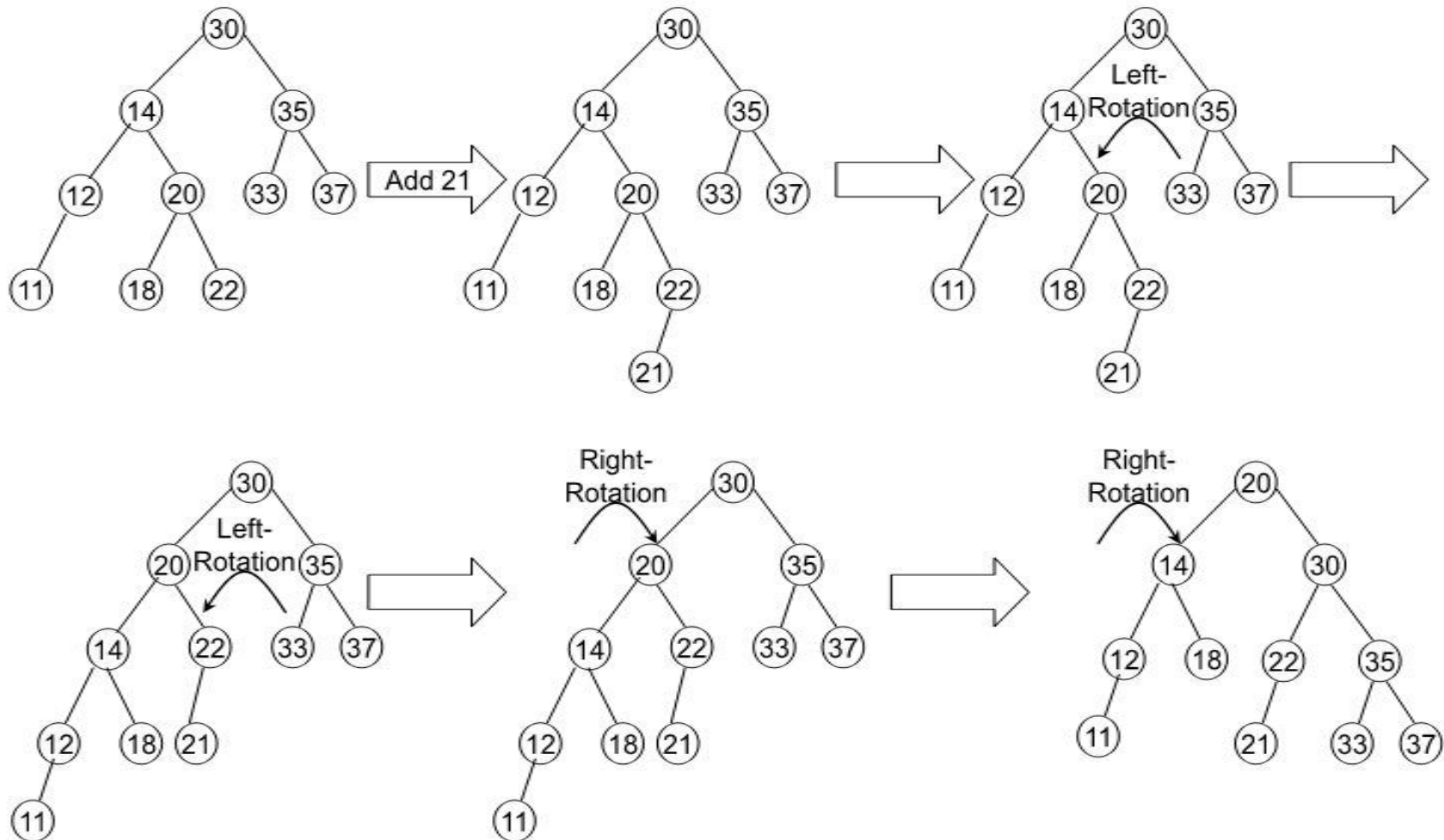
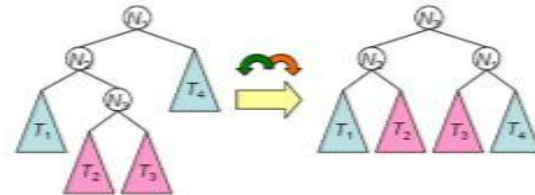
Solution: *Left-right* rotation



Left-Right Rotation

Case 3: insertion to *right* subtree of *left* child

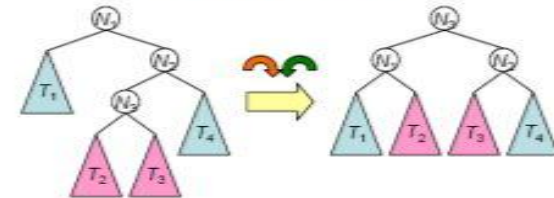
Solution: *Left-right* rotation



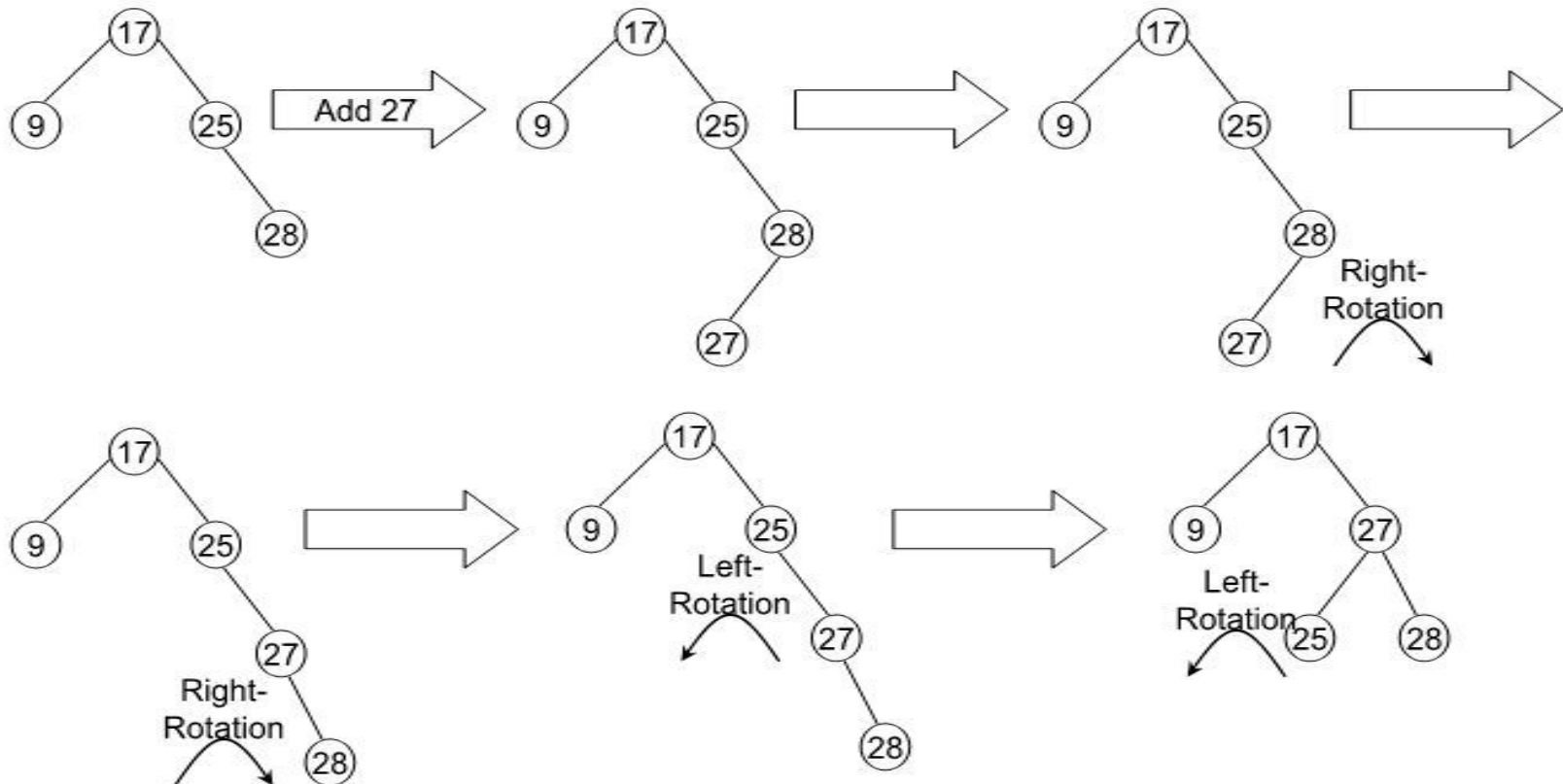
Right-Left Rotation

Case 4: insertion to *left*
subtree of *right* child

Solution: *Right-left* rotation



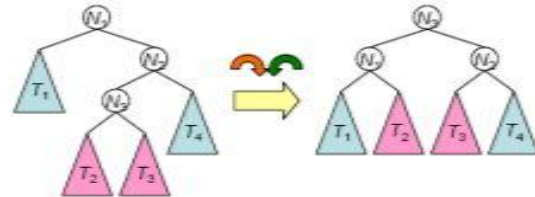
35



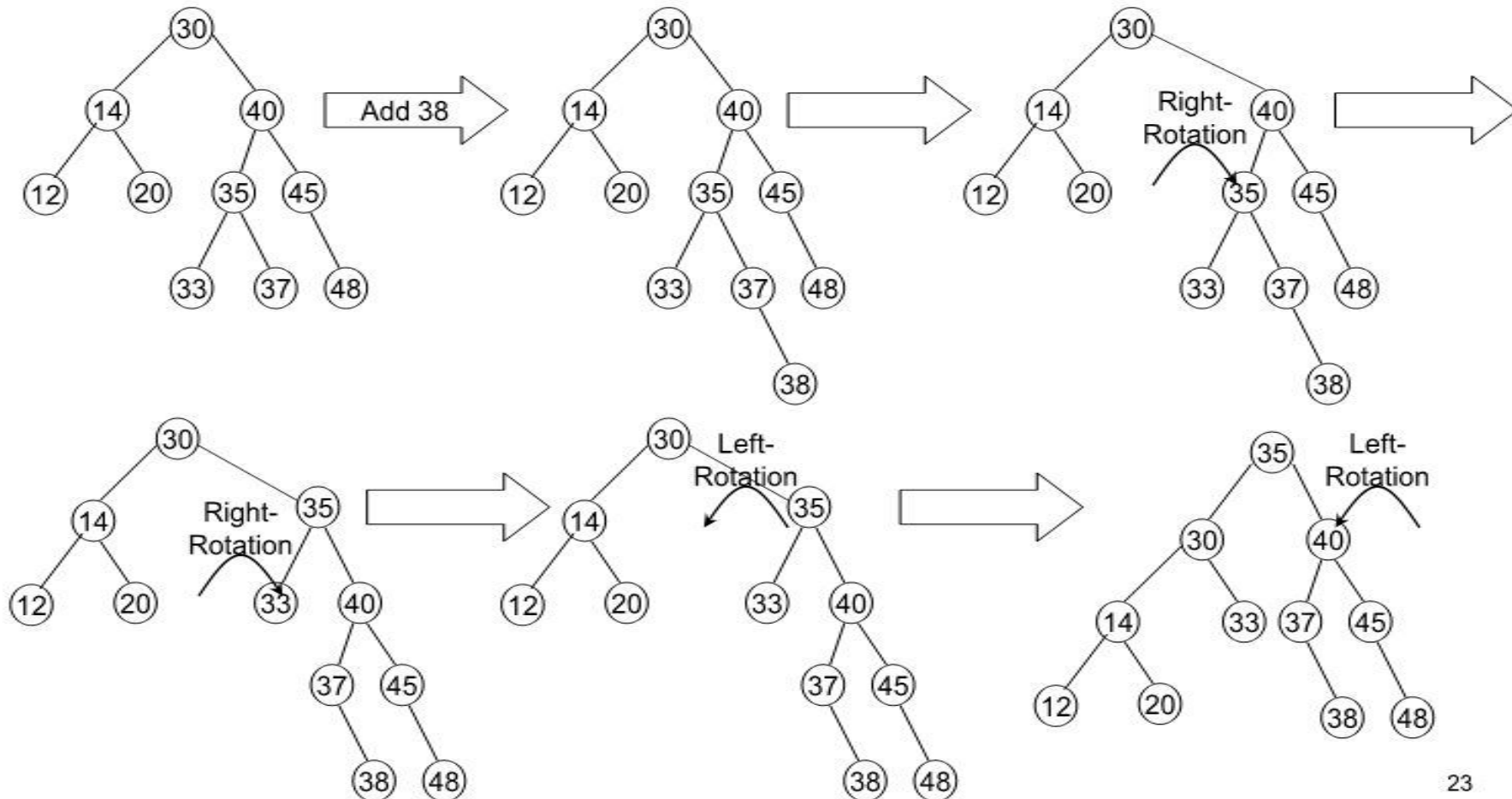
Right-Left Rotation

Case 4: insertion to *left* subtree of *right* child

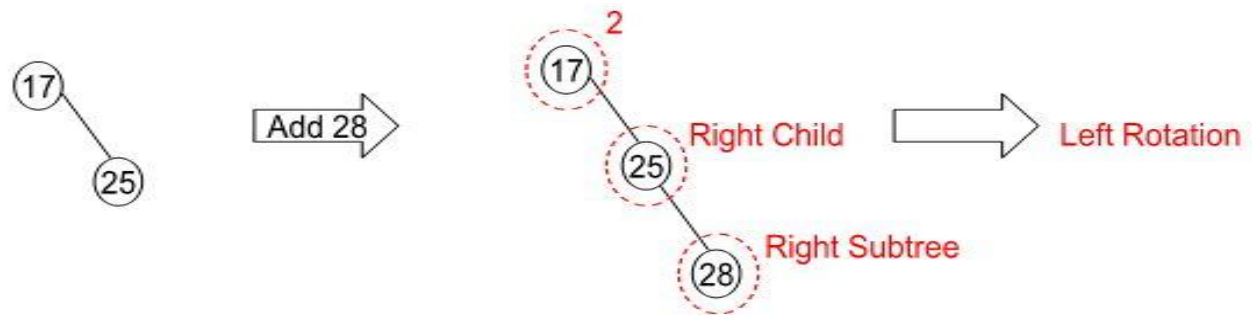
Solution: *Right-left* rotation



35

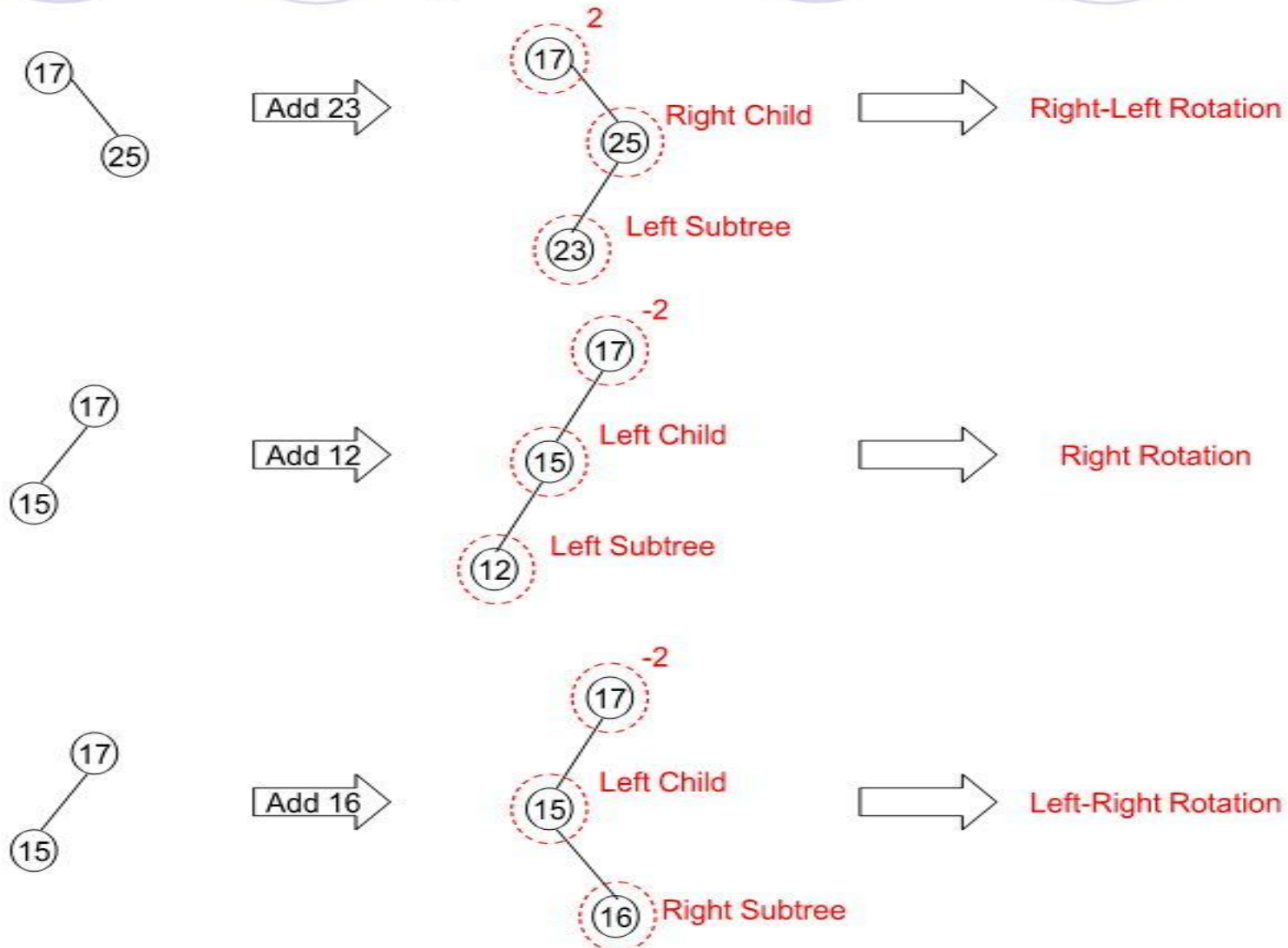


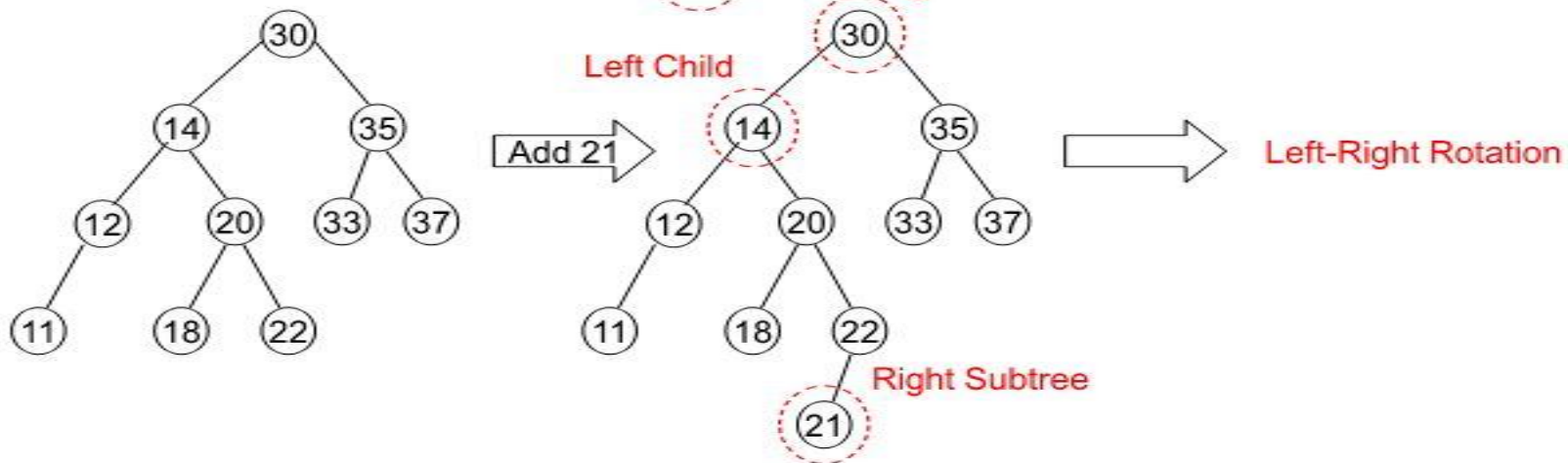
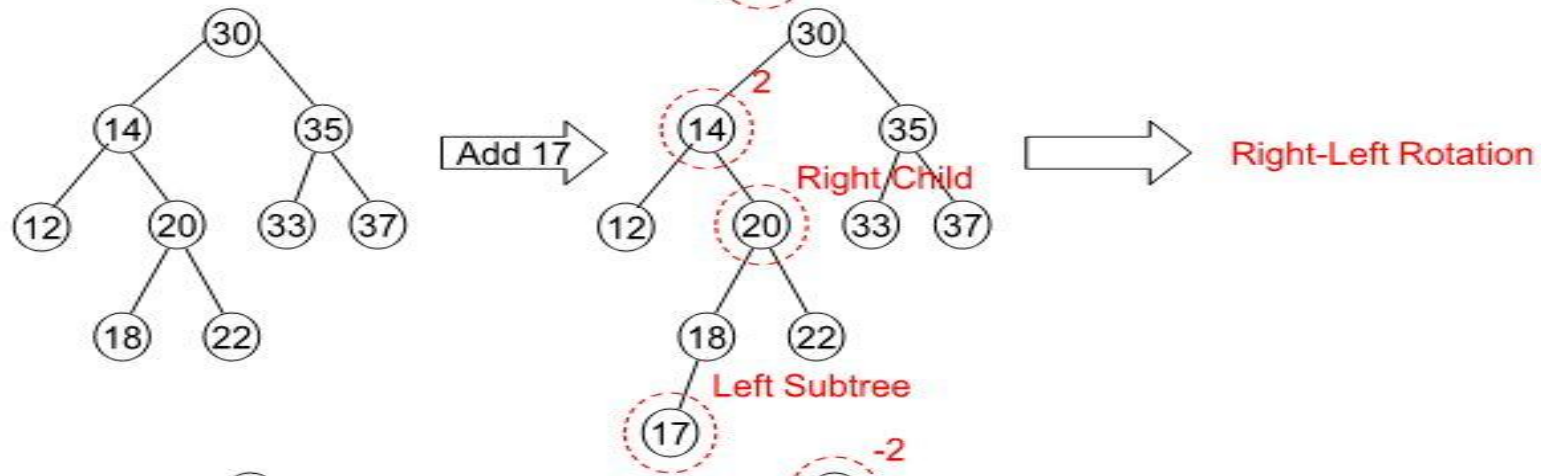
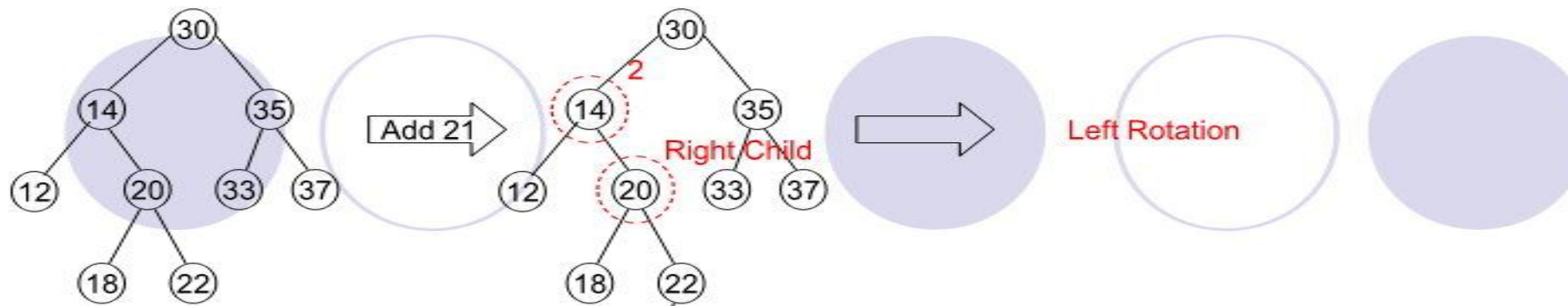
How to identify rotations?



- First find the node that cause the imbalance (balance factor)
- Then find the corresponding child of the imbalanced node (left node or right node)
- Finally find the corresponding subtree of that child (left or right)

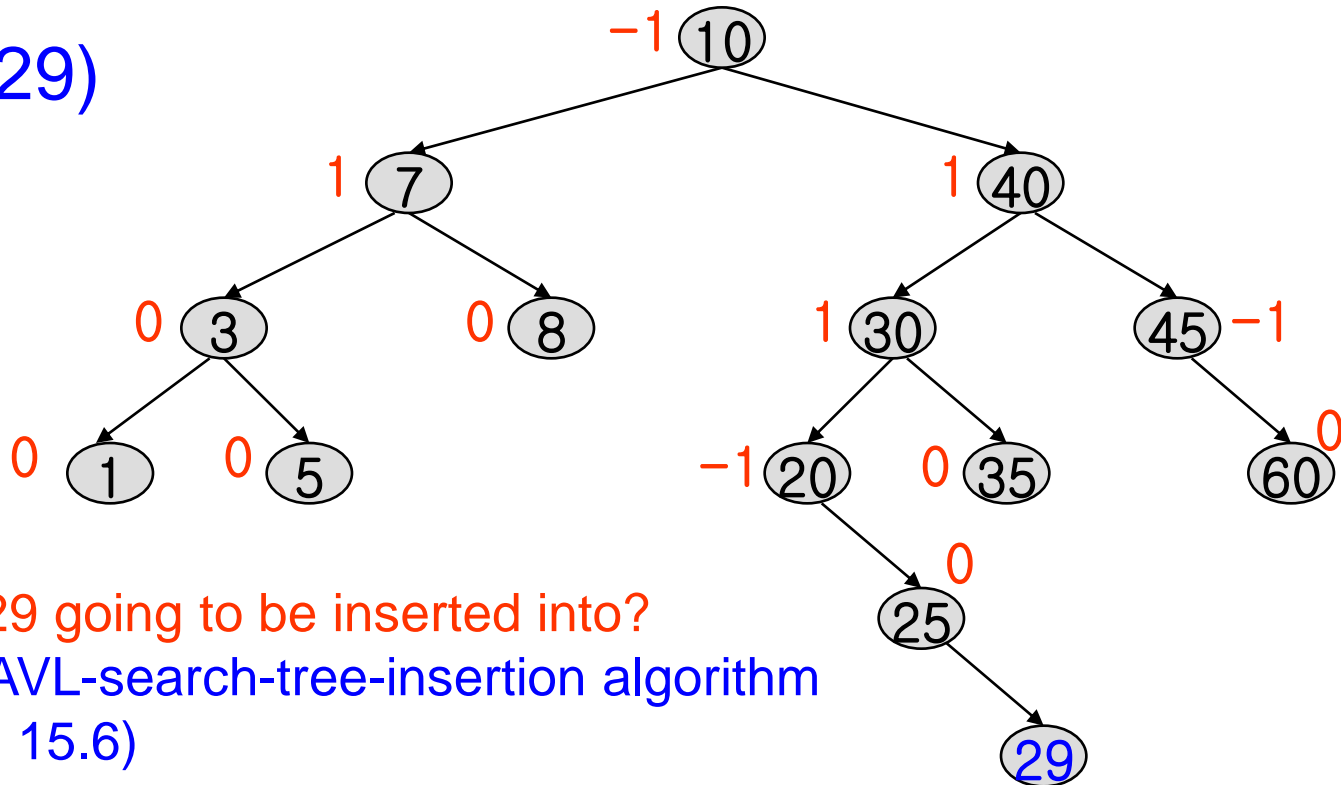
How to identify rotations?





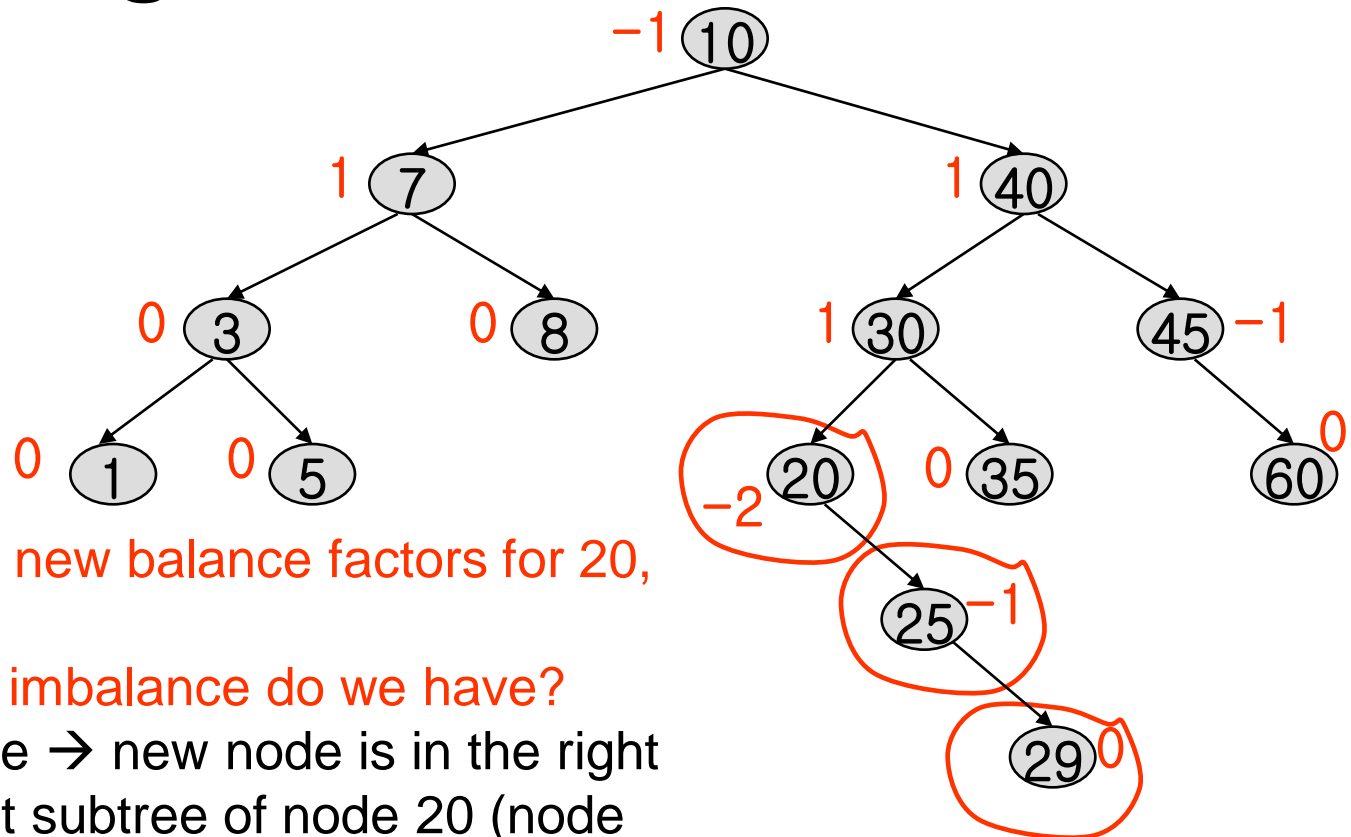
Inserting into an AVL Search Tree

Insert(29)



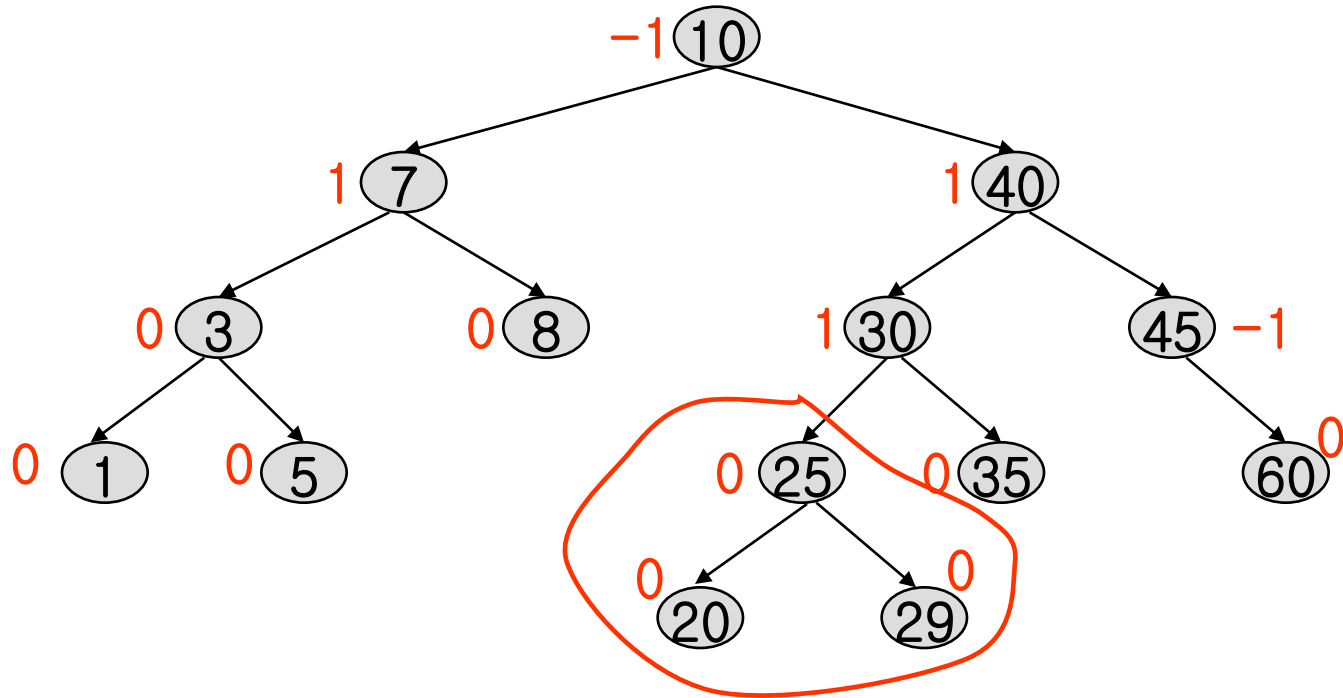
- Where is 29 going to be inserted into?
 - use the AVL-search-tree-insertion algorithm in Figure 15.6)
- After the insertion, is the tree still an AVL search tree? (i.e., still balanced?)

Inserting into an AVL Search Tree



- What are the new balance factors for 20, 25, 29?
- What type of imbalance do we have?
- RR imbalance → new node is in the right subtree of right subtree of node 20 (node with bf = -2) → what rotation do we need?
- What would the left subtree of 30 look like after RR rotation?

After RR Rotation



- After the RR rotation, is the resulting tree an AVL search tree?

Balancing an AVL tree after an insertion

- Begin at the node containing the item which was just inserted and move back along the access path toward the root. {
 - For each node determine its height and **check the balance condition**. {
 - If the tree is AVL balanced and no further nodes need be considered.
 - else If the node has become unbalanced, a rotation is needed to balance it.}
- } we proceed to the next node on the access path.

```
AVLnode *insert(Datatype x, AVLnode *t) {  
    if (t == NULL) {  
        /* CreateNewNode */  
    }  
    else if (x < t->element) {  
        t->left = insert(x, t->left);  
        /* DoLeft */  
    }  
    else if (x > t->element) {  
        t->right = insert(x, t->right);  
        /* DoRight */  
    }  
}
```

AVL tree

- **CreateNewNode**

```
t = malloc(sizeof(struct AVLnode);  
t->element = x;  
t->left = NULL;  
t->right = NULL;
```

AVL tree

- **DoLeft**

```
if (height(t->left) - height(t->right) == 2)
  if (x < t->left->element)
    t = singleRotateWithLeft(t); // LL
  else
    t = doubleRotateWithLeft(t); // LR
```

AVL tree

- **DoRight**

```
if (height(t->right) - height(t->left) == 2)
    if (x > t->right->element)
        t = singleRotateWithRight(t); // RR
    else
        t = doubleRotateWithRight(t); // RL
```

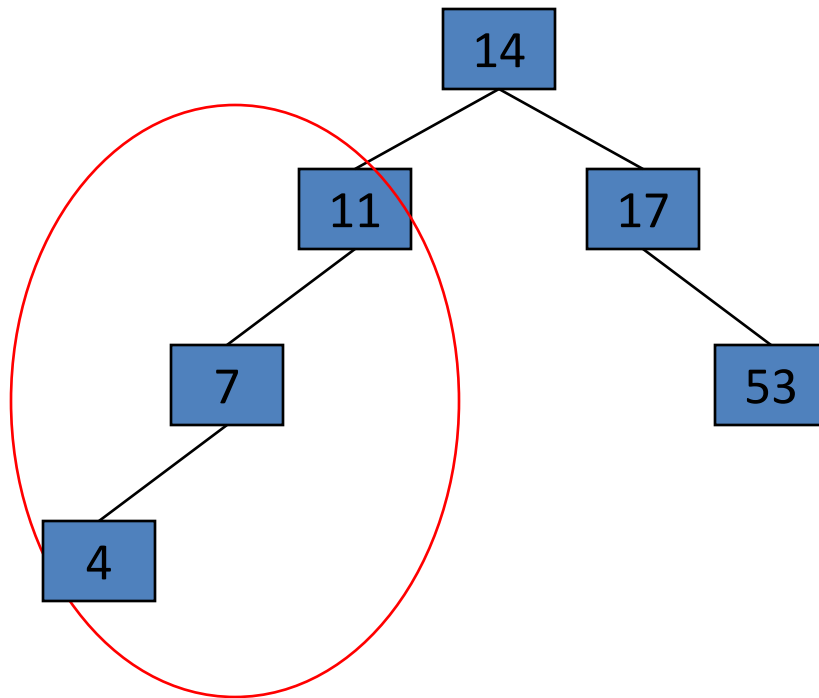
Deletion from an AVL Search Tree

Deletion procedure is more complex than insertion in 2 ways:

- **1) More number of cases for rebalancing may arise in deletion;**
- **2) In insertion there is only one rebalancing, but in deletion there can be as many rebalancing as the length of the path from the deleted node to the root.**

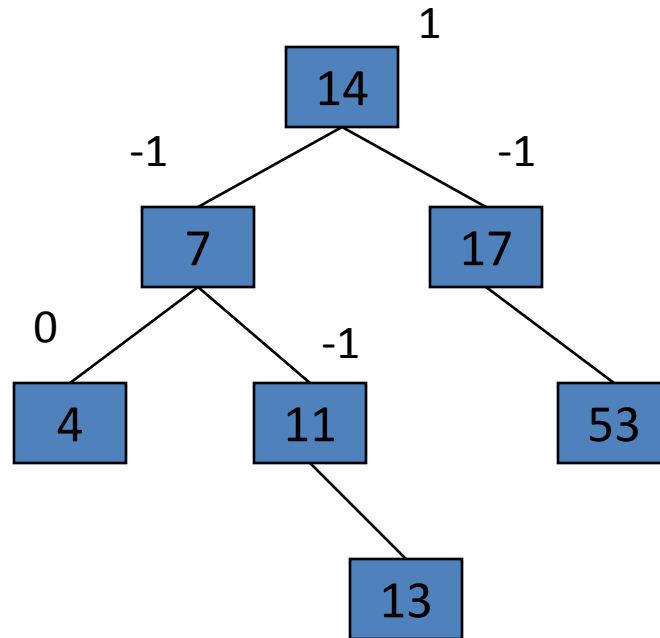
AVL Tree Example:

- Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree



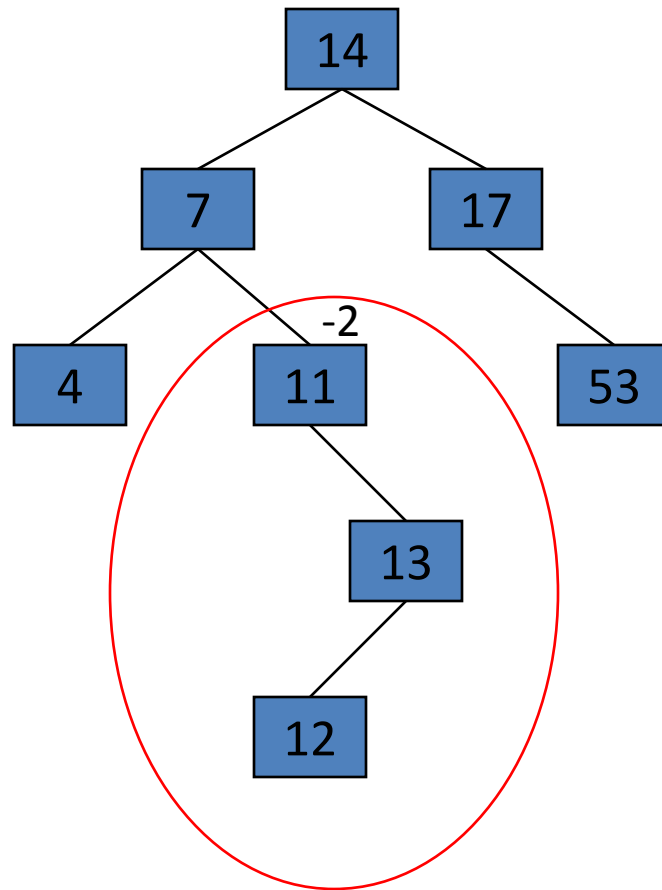
AVL Tree Example:

- Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree



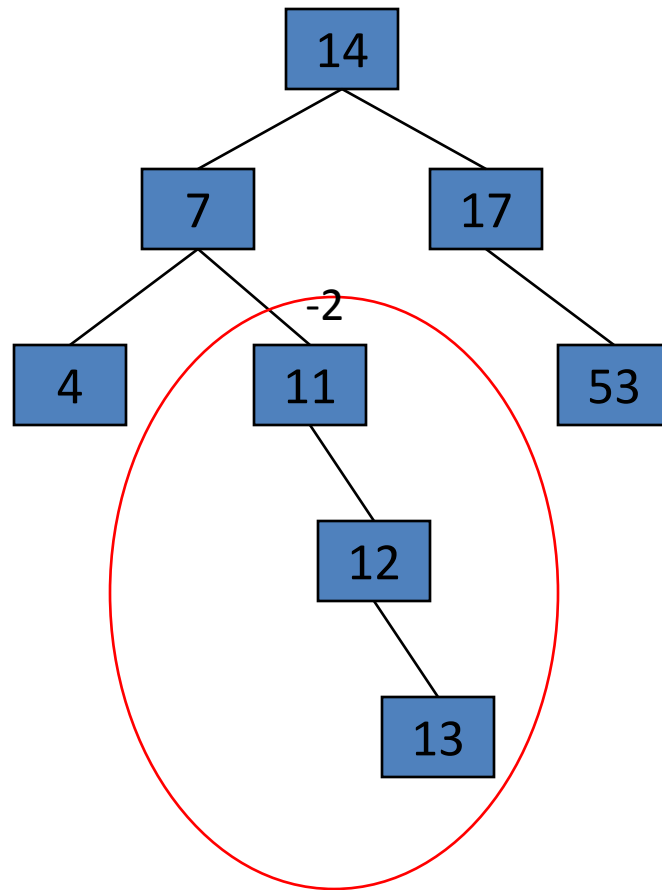
AVL Tree Example:

- Now insert 12



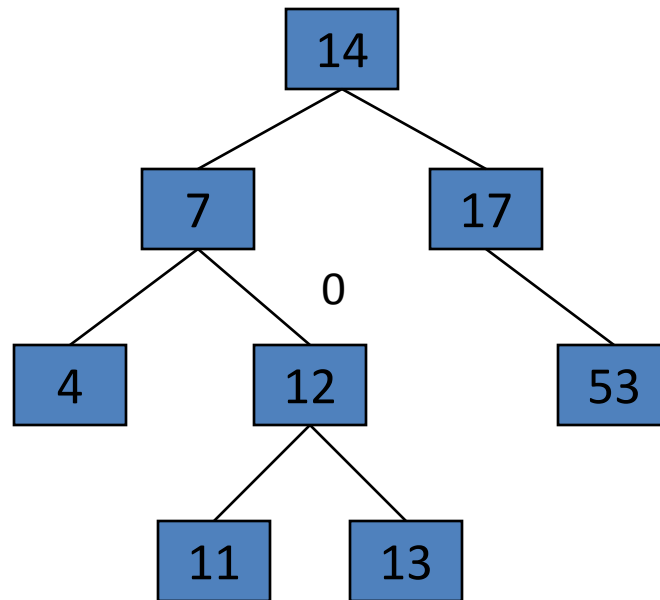
AVL Tree Example:

- Now insert 12



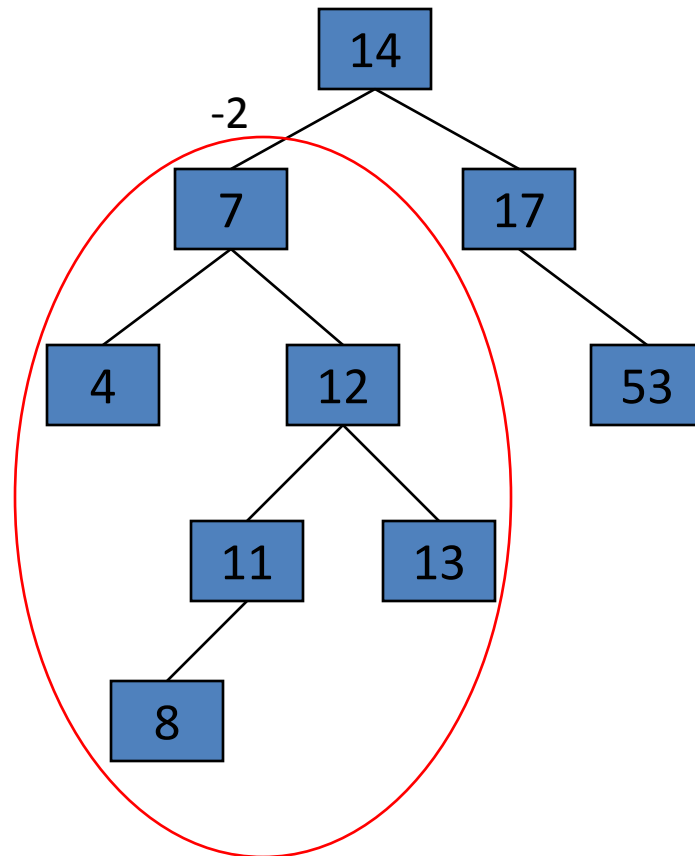
AVL Tree Example:

- Now the AVL tree is balanced.



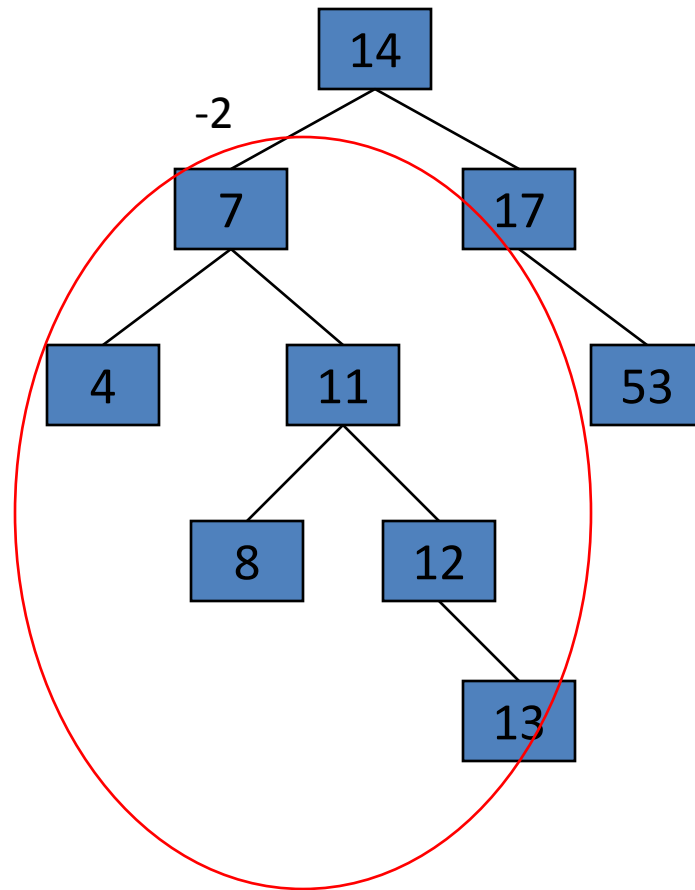
AVL Tree Example:

- Now insert 8



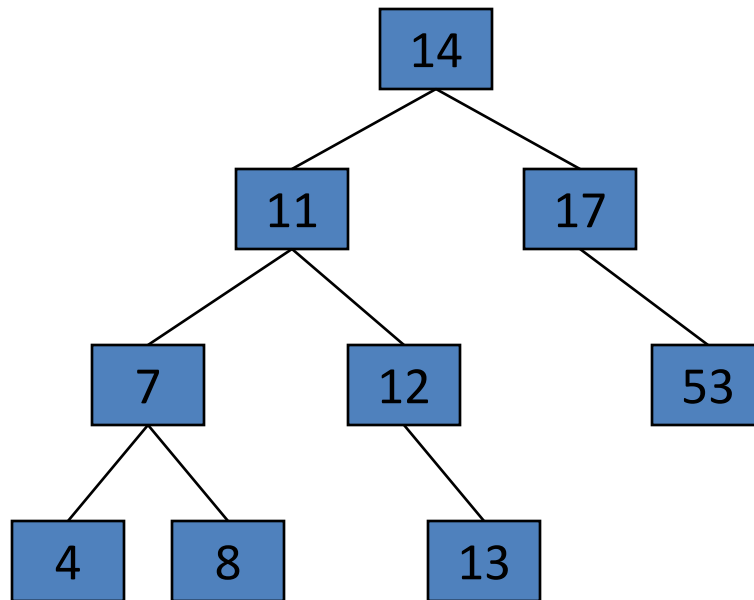
AVL Tree Example:

- Now insert 8



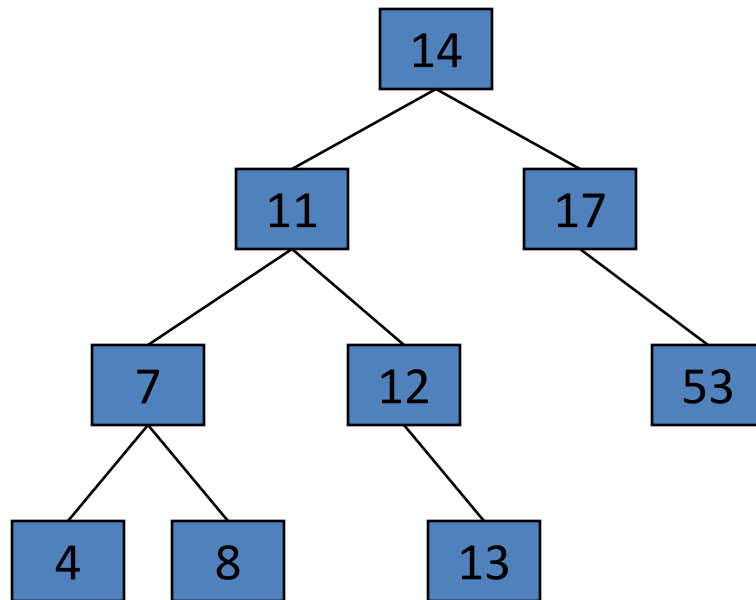
AVL Tree Example:

- Now the AVL tree is balanced.



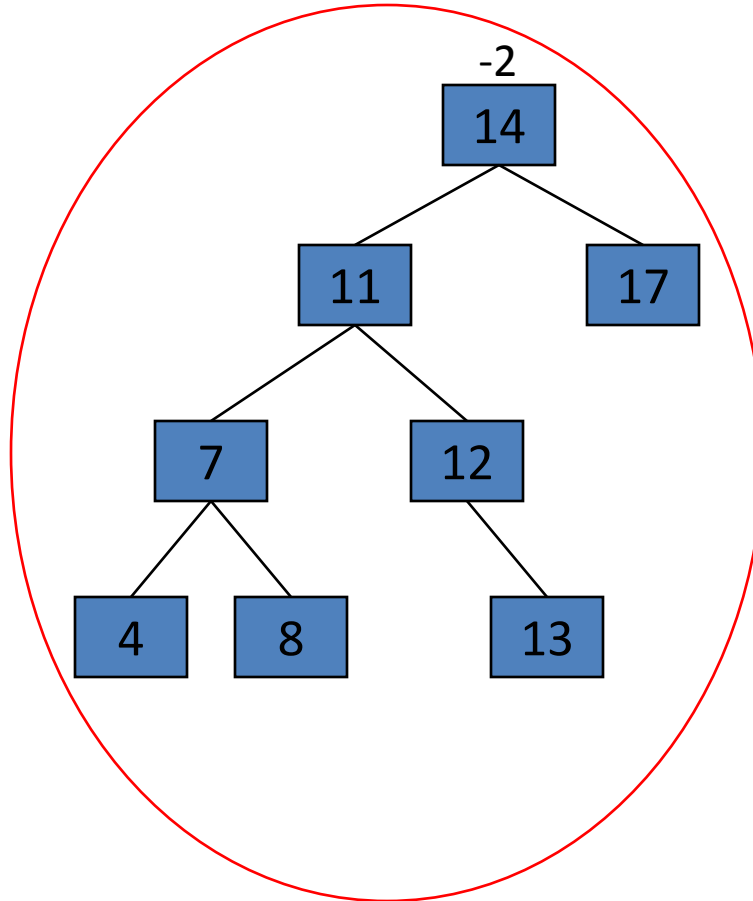
AVL Tree Example:

- Now remove 53



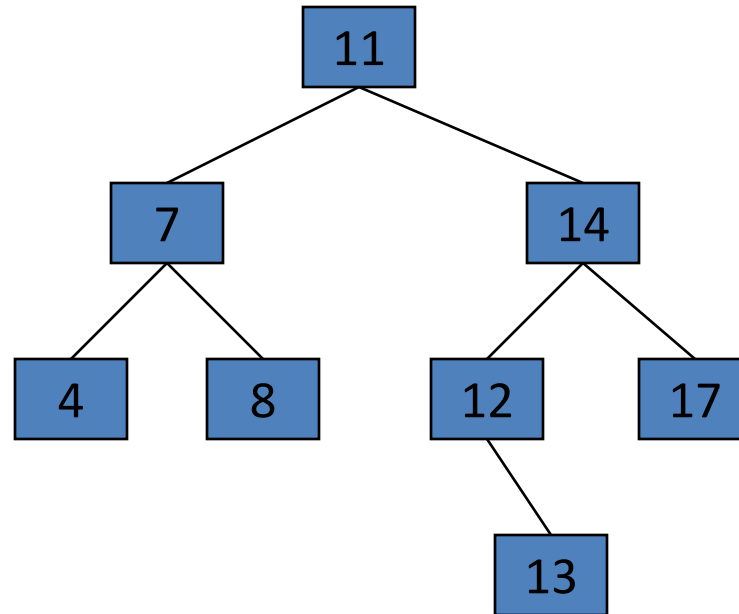
AVL Tree Example:

- Now remove 53, unbalanced



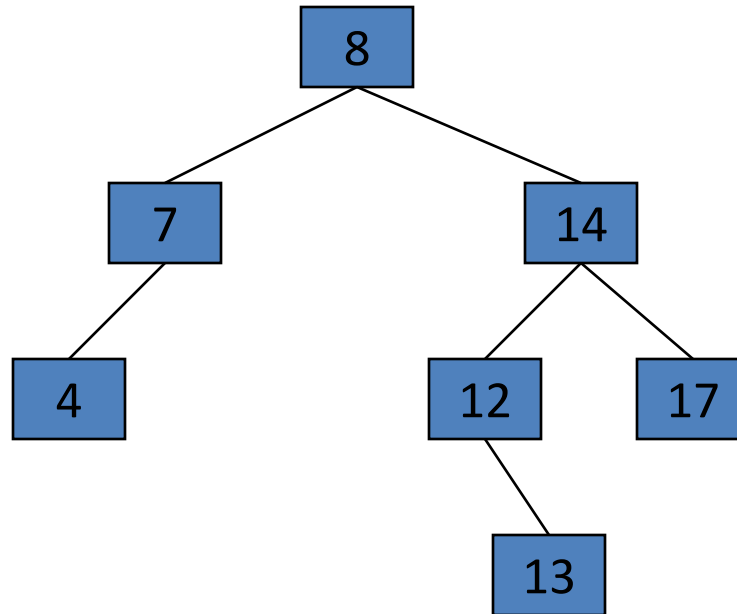
AVL Tree Example:

- **Balanced! Remove 11**



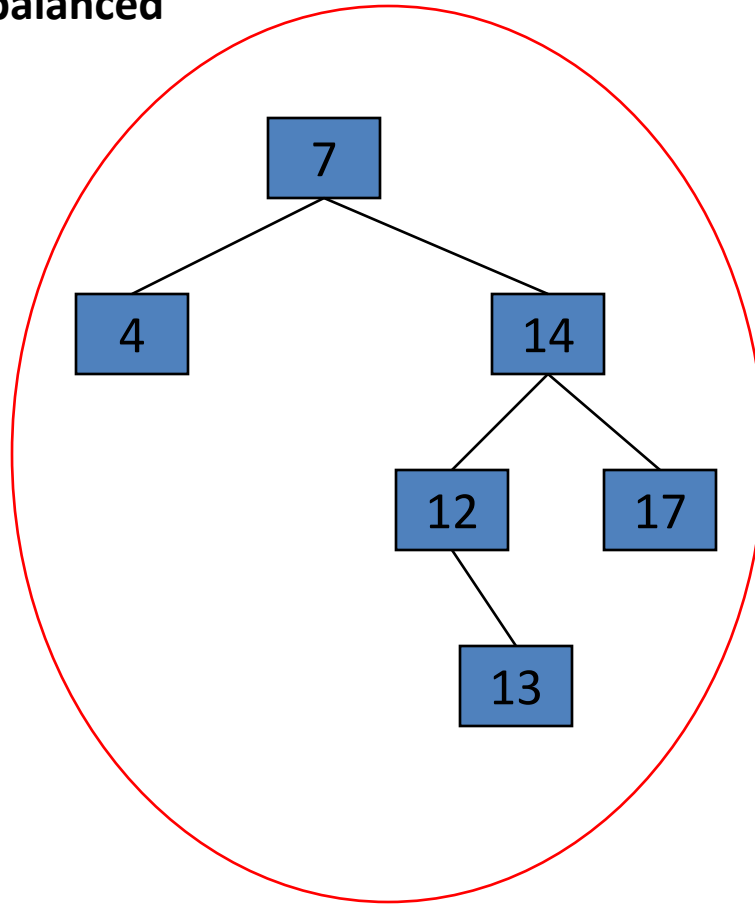
AVL Tree Example:

- Remove 11, replace it with the largest in its left branch



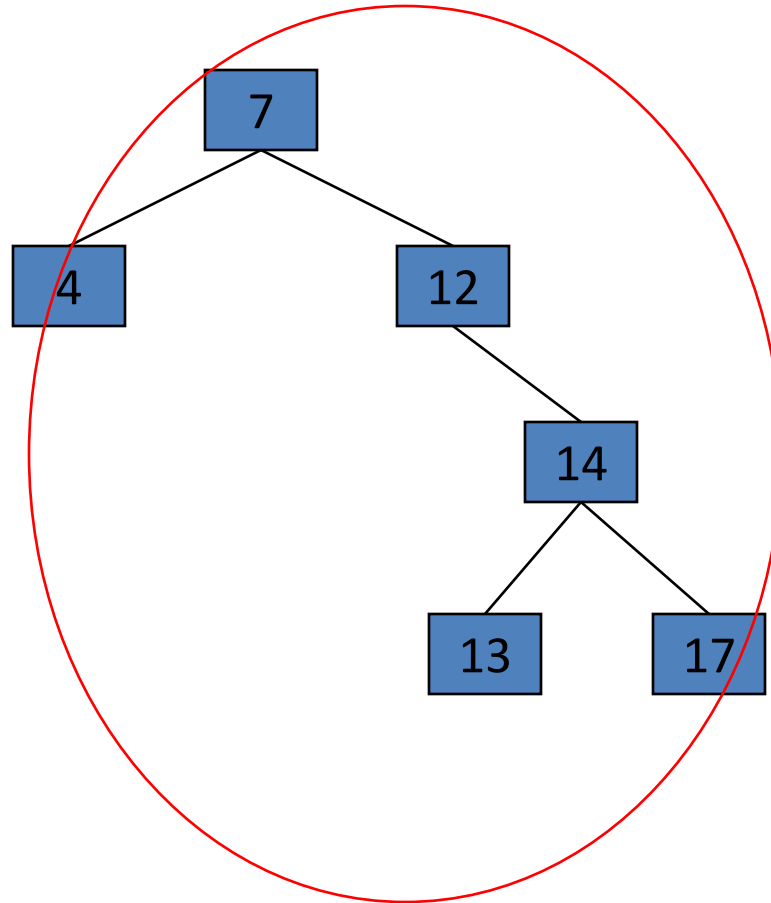
AVL Tree Example:

- Remove 8, unbalanced



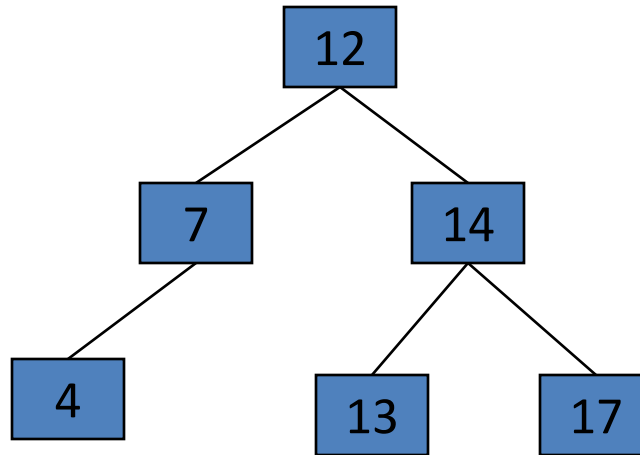
AVL Tree Example:

- Remove 8, unbalanced



AVL Tree Example:

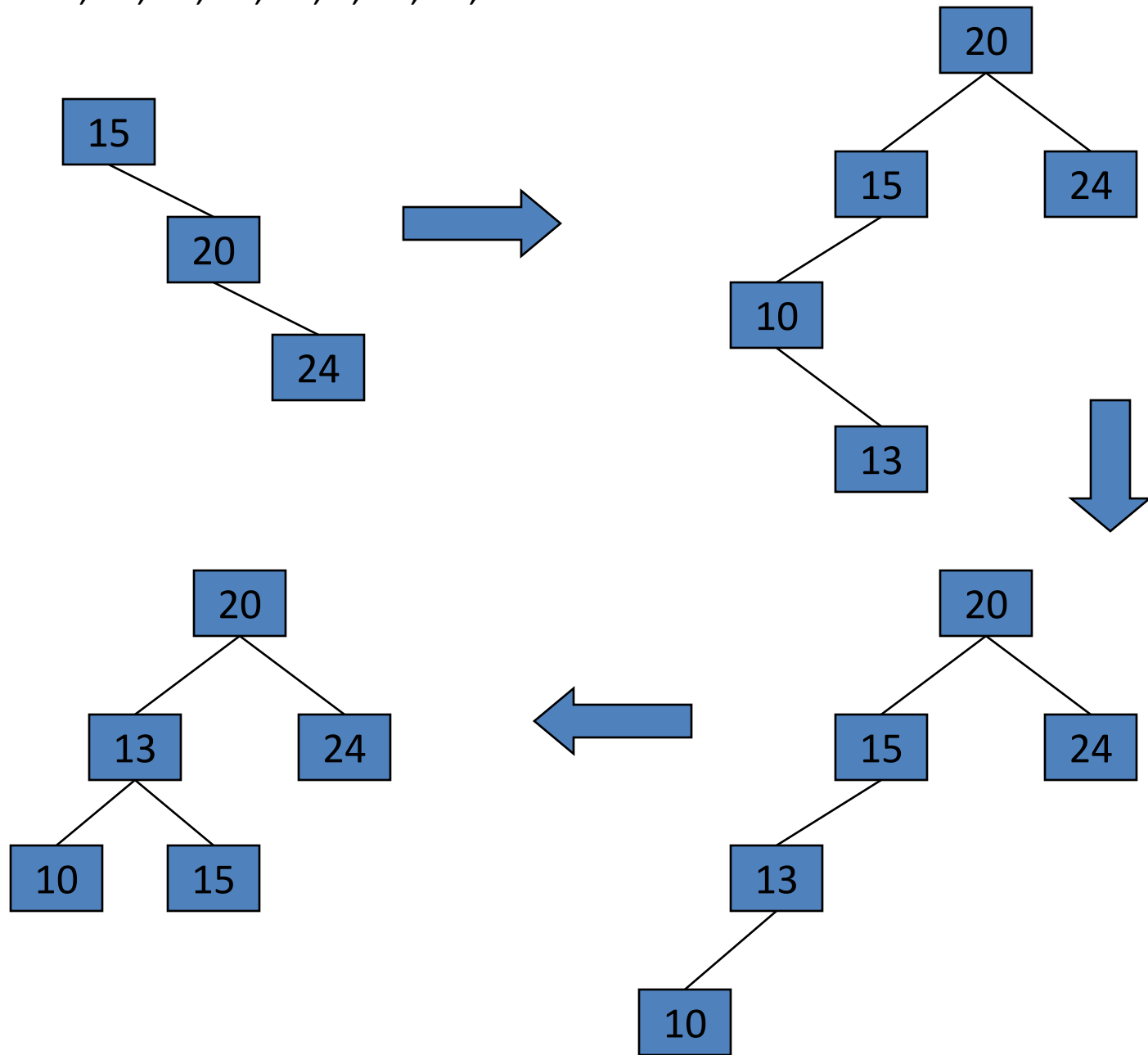
- **Balanced!!**



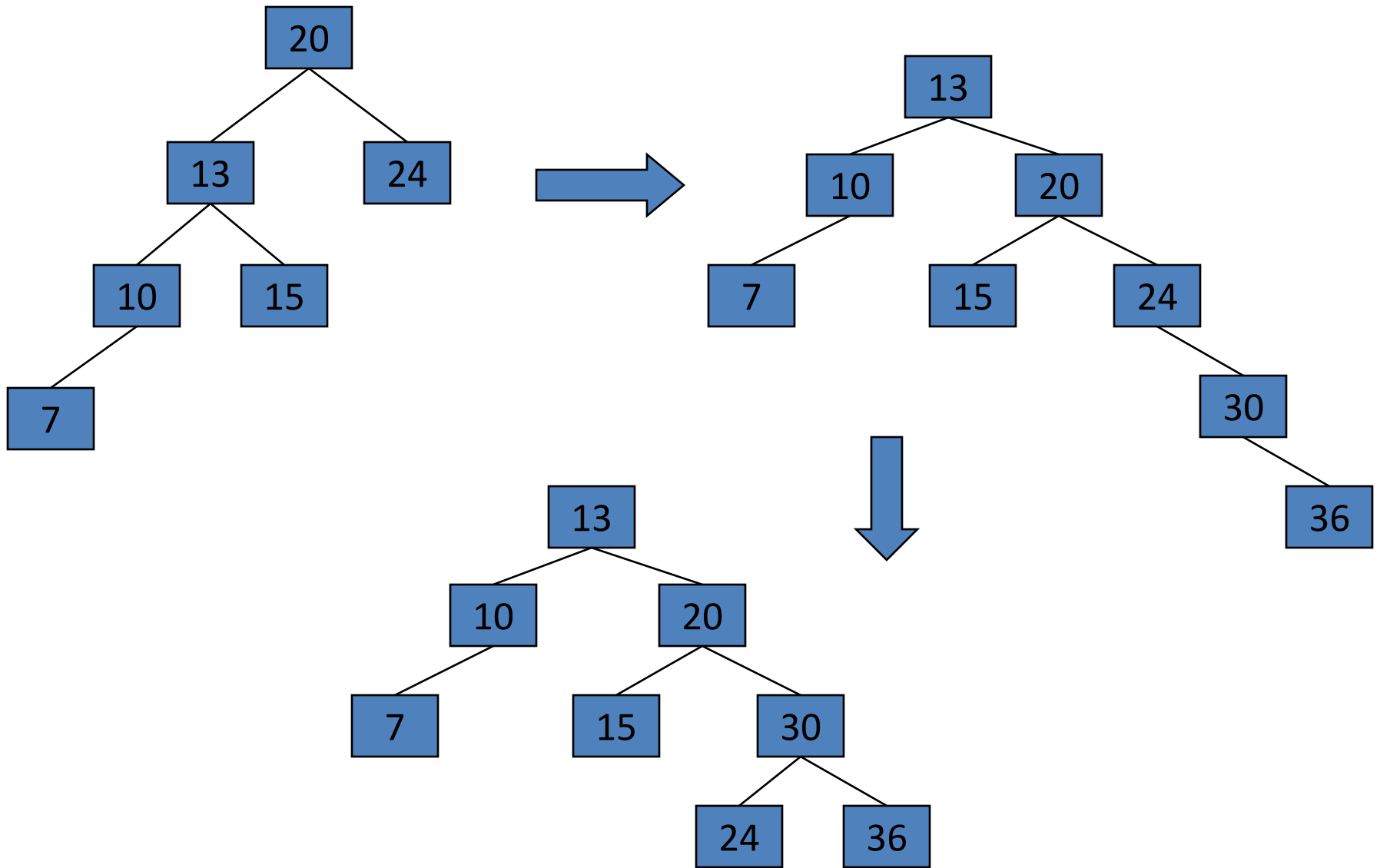
Exercise

- Build an AVL tree with the following values:
15, 20, 24, 10, 13, 7, 30, 36, 25

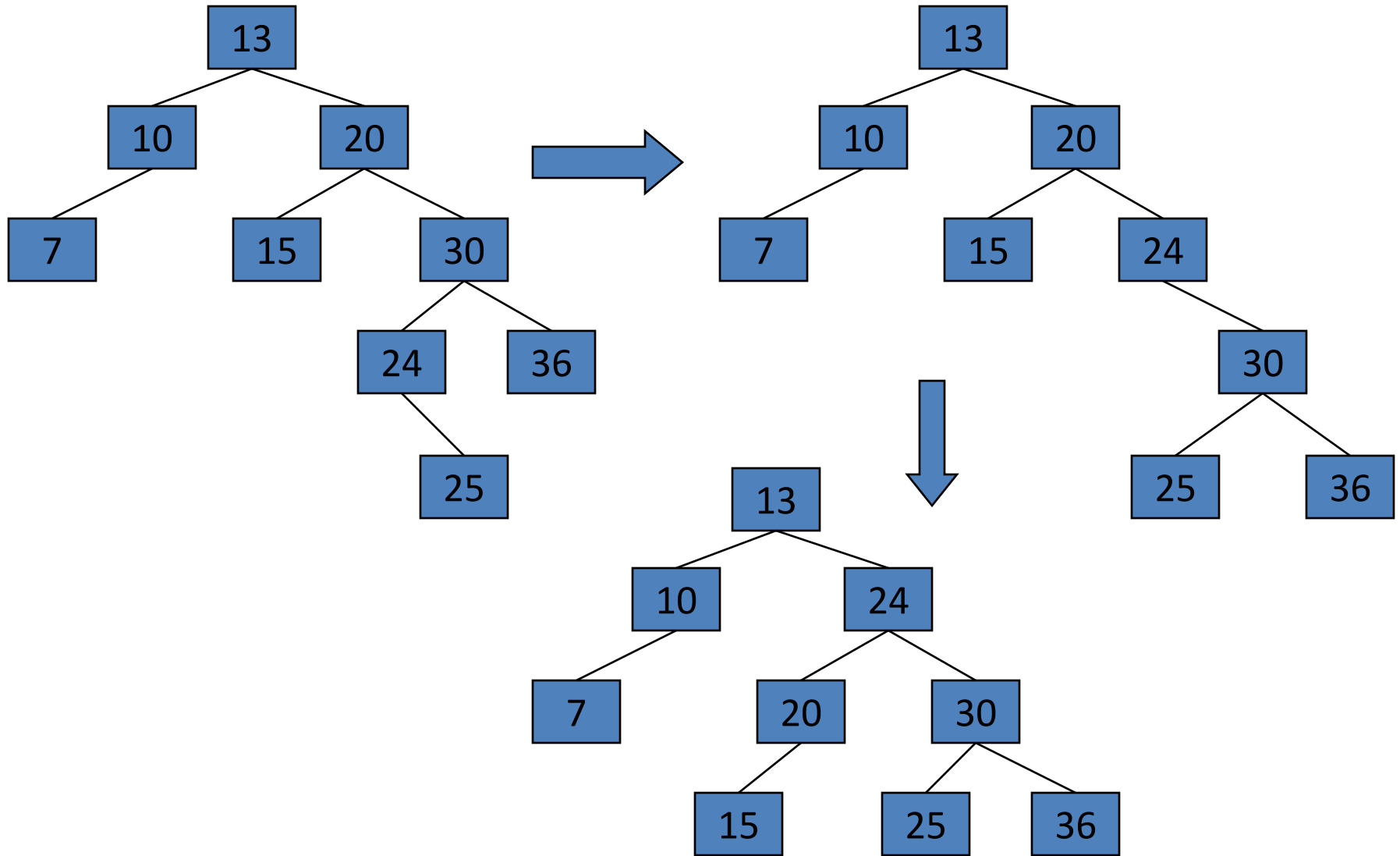
15, 20, 24, 10, 13, 7, 30, 36, 25



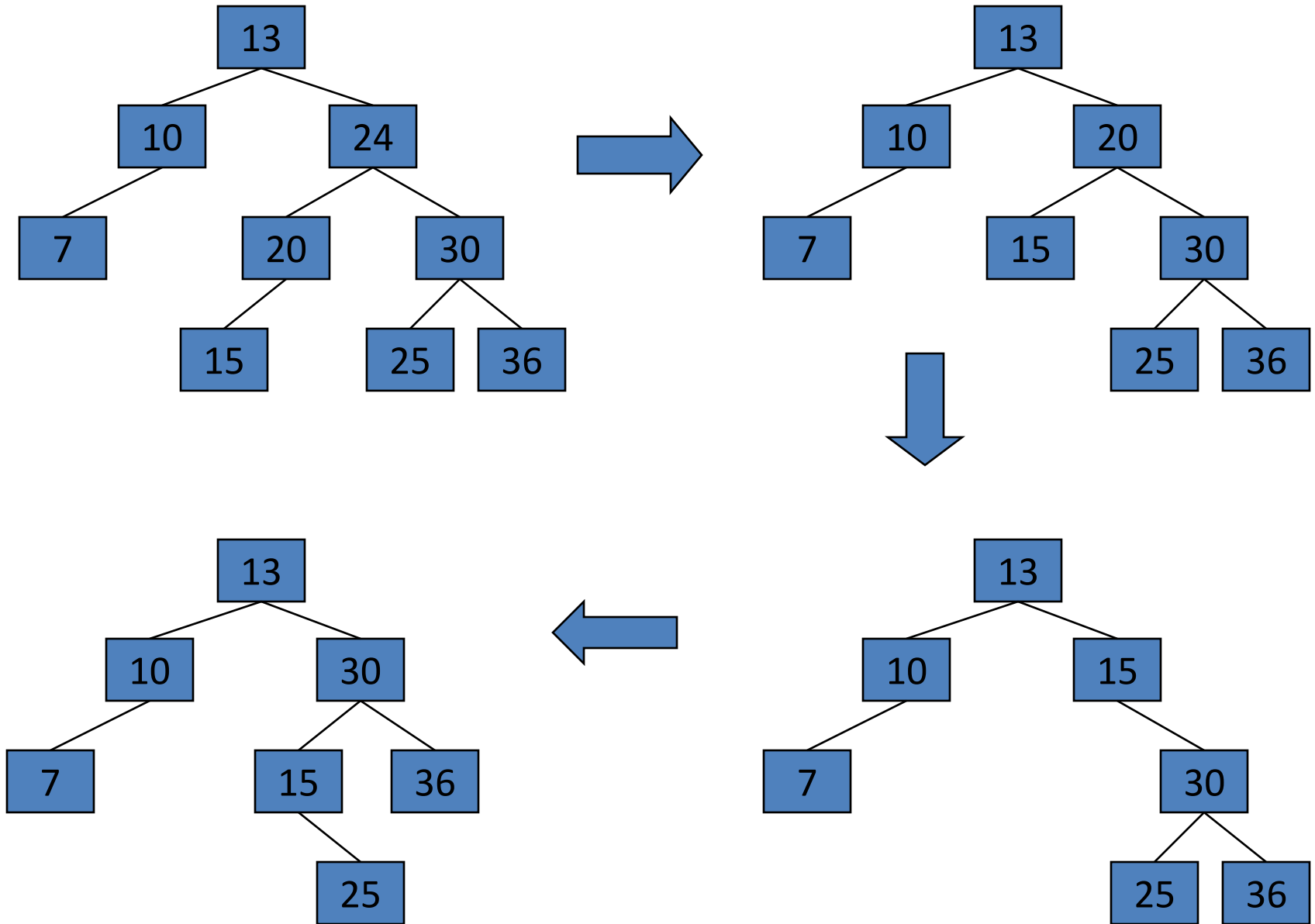
15, 20, 24, 10, 13, 7, 30, 36, 25



15, 20, 24, 10, 13, 7, 30, 36, 25



Remove 24 and 20 from the AVL tree.



B - TREE

B-tree of order n

- Every B-tree is of some "order n ", meaning nodes contain from n to $2n$ keys (so nodes are always at least half full of keys), and $n+1$ to $2n+1$ pointers, and n can be any number.
- Keys are kept in sorted order within each node. A corresponding list of pointers are effectively interspersed between keys to indicate where to search for a key if it isn't in the current node.

- **A B-tree of order n is a multi-way search tree with two properties:**
 - **1. All leaves are at the same level**
 - **2. The number of keys in any node lies between n and $2n$, with the possible exception of the root which may have fewer keys.**

Other definition

A B-tree of order m is a m -way tree that satisfies the following conditions.

- **Every node has $\leq m$ children.**
- **Every internal node (except the root) has $\leq m/2$ children.**
- **The root has ≥ 2 children.**
- **An internal node with k children contains $(k-1)$ ordered keys. The leftmost child contains keys less than or equal to the first key in the node. The second child contains keys greater than the first keys but less than or equal to the second key, and so on.**

B-tree of order n

- Every B-tree is of some "order n ", meaning nodes contain from n to $2n$ keys (so nodes are always at least half full of keys), and $n+1$ to $2n+1$ pointers, and n can be any number.
- Keys are kept in sorted order within each node. A corresponding list of pointers are effectively interspersed between keys to indicate where to search for a key if it isn't in the current node.

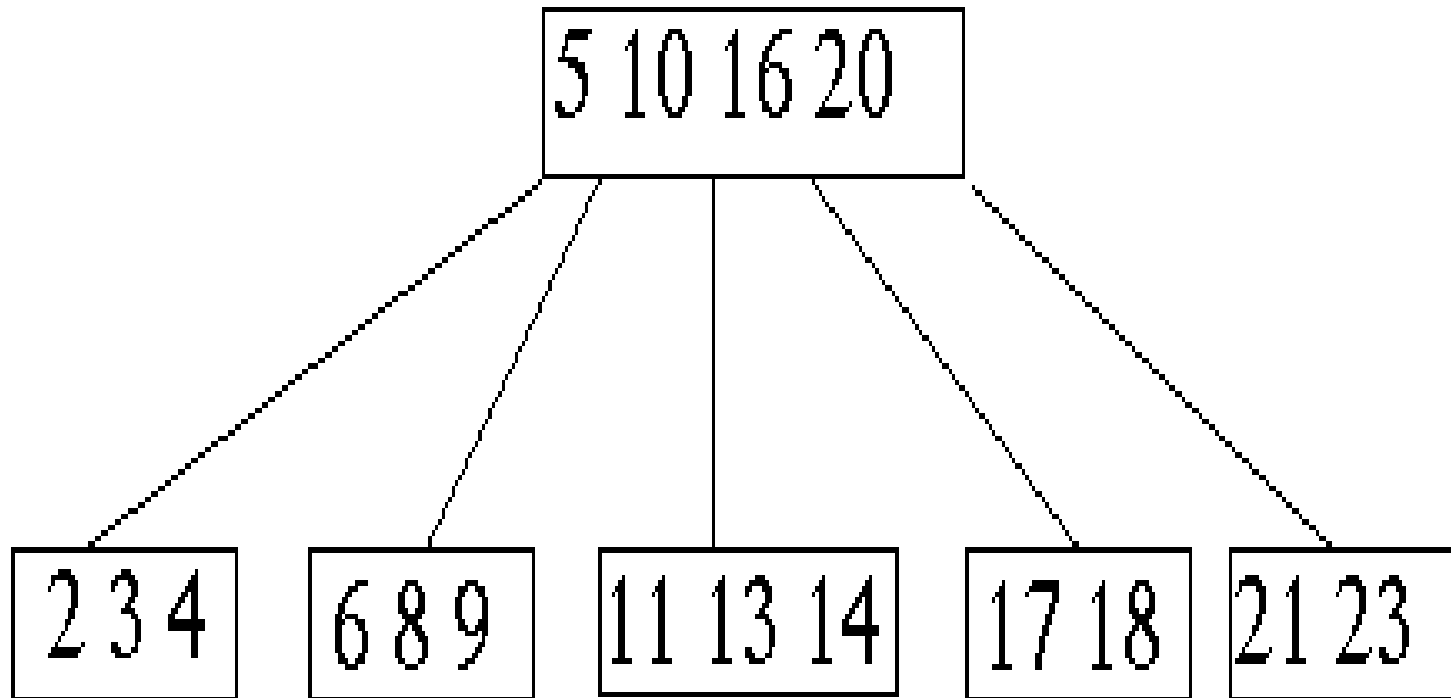
- **A B-tree of order n is a multi-way search tree with two properties:**
 - **1.All leaves are at the same level**
 - **2.The number of keys in any node lies between n and $2n$, with the possible exception of the root which may have fewer keys.**

Other definition

A B-tree of order m is a m -way tree that satisfies the following conditions.

- **Every node has $\leq m$ children.**
- **Every internal node (except the root) has $\leq m/2$ children.**
- **The root has ≥ 2 children.**
- **An internal node with k children contains $(k-1)$ ordered keys. The leftmost child contains keys less than or equal to the first key in the node. The second child contains keys greater than the first keys but less than or equal to the second key, and so on.**

A B-tree of order 2

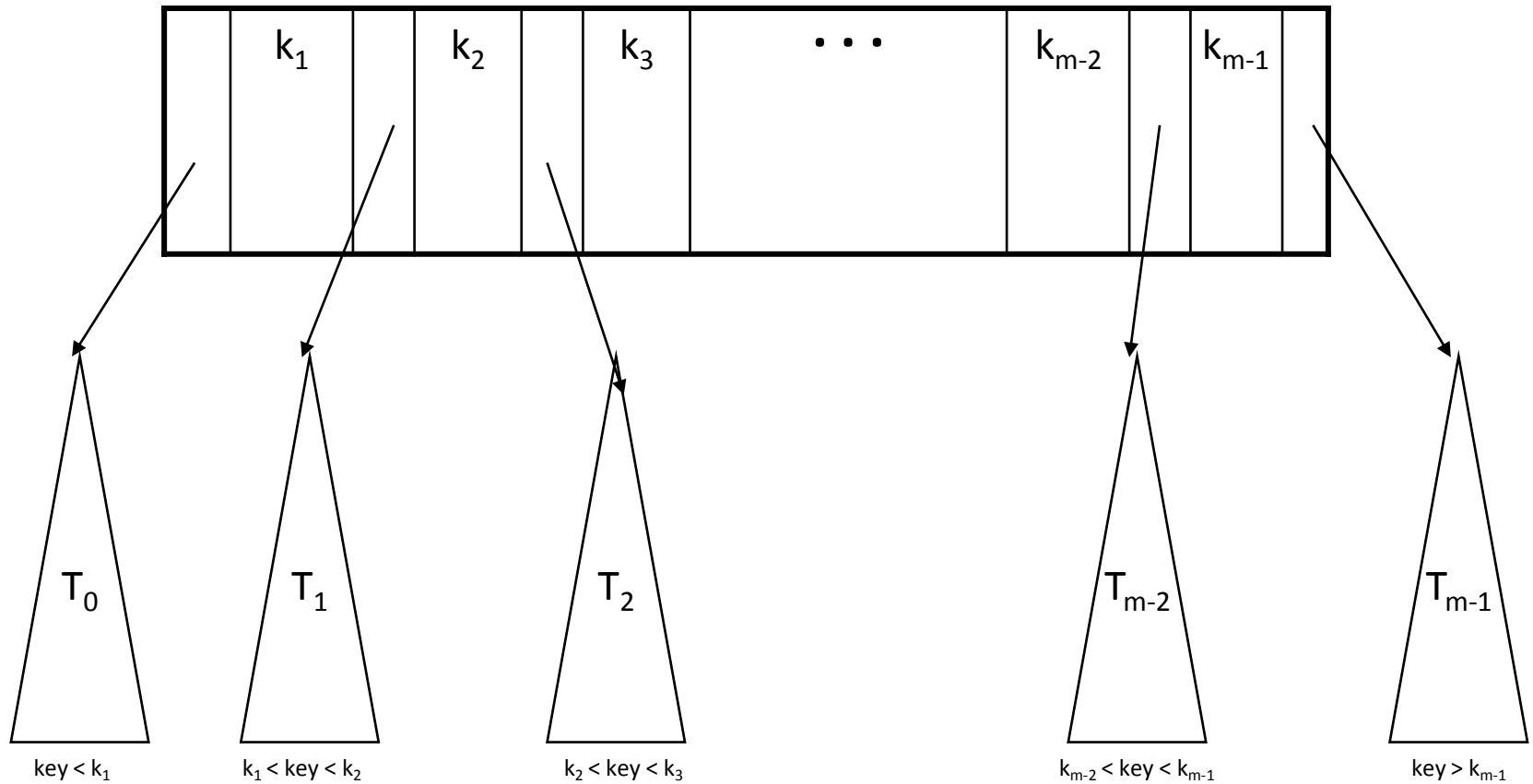


- A multi-way (or m-way) search tree of order m is a tree in which
 - Each node has at-most m subtrees, where the subtrees may be empty.
 - Each node consists of at least 1 and at most m-1 distinct keys
 - The keys in each node are sorted.
- The keys and subtrees of a non-leaf node are ordered as:

$T_0, k_1, T_1, k_2, T_2, \dots, k_{m-1}, T_{m-1}$ such that:

 - All keys in subtree T_0 are less than k_1 .
 - All keys in subtree T_i , $1 \leq i \leq m - 2$, are greater than k_i but less than k_{i+1} .
 - All keys in subtree T_{m-1} are greater than k_{m-1}

Multi-way tree



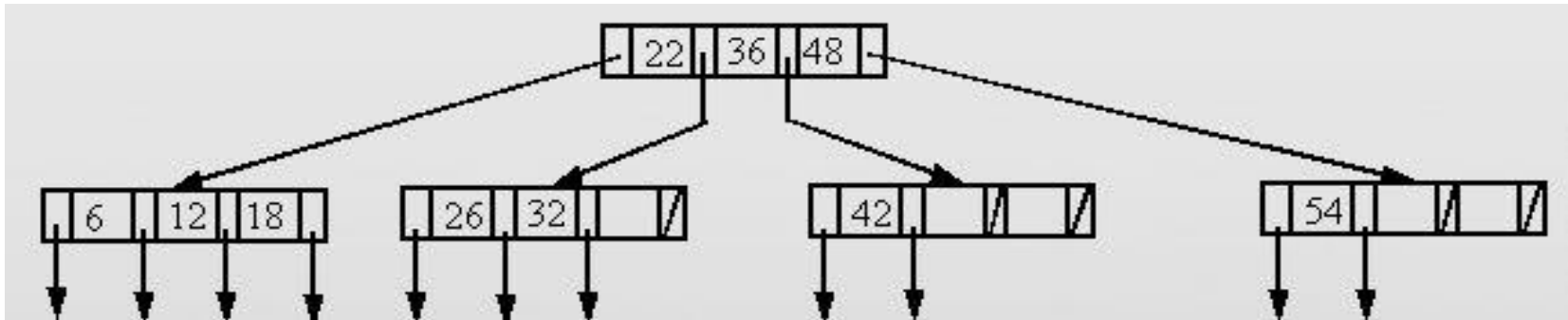
What is B-tree?

- **B-tree of order m (or branching factor m), where $m > 2$, is either an empty tree or a multiway search tree with the following properties:**
 - The root is either a leaf or it has at least two non-empty subtrees and at most m non-empty subtrees.
 - Each non-leaf node, other than the root, has at least $\lceil m/2 \rceil$ non-empty subtrees and at most m non-empty subtrees. (Note: $\lceil x \rceil$ is the lowest integer $> x$).
 - The number of keys in each non-leaf node is one less than the number of non-empty subtrees for that node.
 - All leaf nodes are at the same level; that is the tree is perfectly balanced

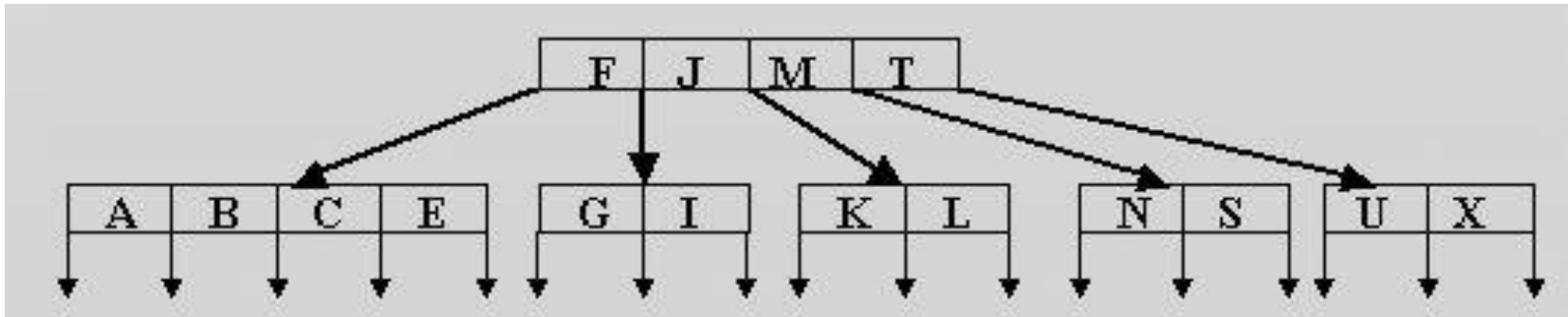
- For a non-empty B-tree of order m :

	Root node	Non-root node
Minimum number of keys	1	$\lceil m/2 \rceil - 1$
Minimum number of non-empty subtrees	2	$\lceil m/2 \rceil$
Maximum number of keys	$m - 1$	$m - 1$
Maximum number of non-empty subtrees	m	m

Example: A B-tree of order 4



Example: A B-tree of order 5



Note:

- The data references are not shown.
- The leaf references are to empty subtrees

Height of B-Trees

- For n greater than or equal to one, the height of an n -key b-tree T of height h with a minimum degree t greater than or equal to 2

$$h \leq \log_t \frac{n+1}{2}$$

Operations of B-Trees

- **B-Tree-Search(x, k)**

- The search operation on a b-tree is similar to a search on a binary tree. The *B-Tree-search* runs in time $O(\log_t n)$.

- **B-Tree-Create(T)**

- The *B-Tree-Create* operation creates an empty b-tree by allocating a new root node that has no keys and is a leaf node. Only the root node is permitted to have these properties; all other nodes must meet the criteria outlined previously. The *B-Tree-Create* operation runs in time $O(1)$.

Operations of B-Trees

- **B-Tree-Split-Child(x, i, y)**

—If a node becomes "too full," it is necessary to perform a split operation. The split operation moves the median key of node x into its parent y where x is the i th child of y . A new node, z , is allocated, and all keys in x right of the median key are moved to z . The keys left of the median key remain in the original node x . The new node, z , becomes the child immediately to the right of the median key that was moved to the parent y , and the original node, x , becomes the child immediately to the left of the median key that was moved into the parent. The B-Tree-Split-Child algorithm will run in time $O(t)$, T is constrain

Operations of B-Trees

- **B-Tree-Insert(T, k)**
- **B-Tree-Insert-Nonfull(x, k)**

To perform an insertion on a b-tree, the appropriate node for the key must be located using an algorithm similar to *B-Tree-Search*. Next, the key must be inserted into the node.

- **If the node is not full prior to the insertion, no special action is required; however, if the node is full, the node must be split to make room for the new key. Since splitting the node results in moving one key to the parent node, the parent node must not be full or another split operation is required. This process may repeat all the way up to the root and may require splitting the root node.**
- **This approach requires two passes. The first pass locates the node where the key should be inserted; the second pass performs any required splits on the ancestor nodes. runs in time $O(t \log_t n)$**

- **OVERFLOW CONDITION:**

A root-node or a non-root node of a B-tree of order m overflows if, after a key insertion, it contains m keys.

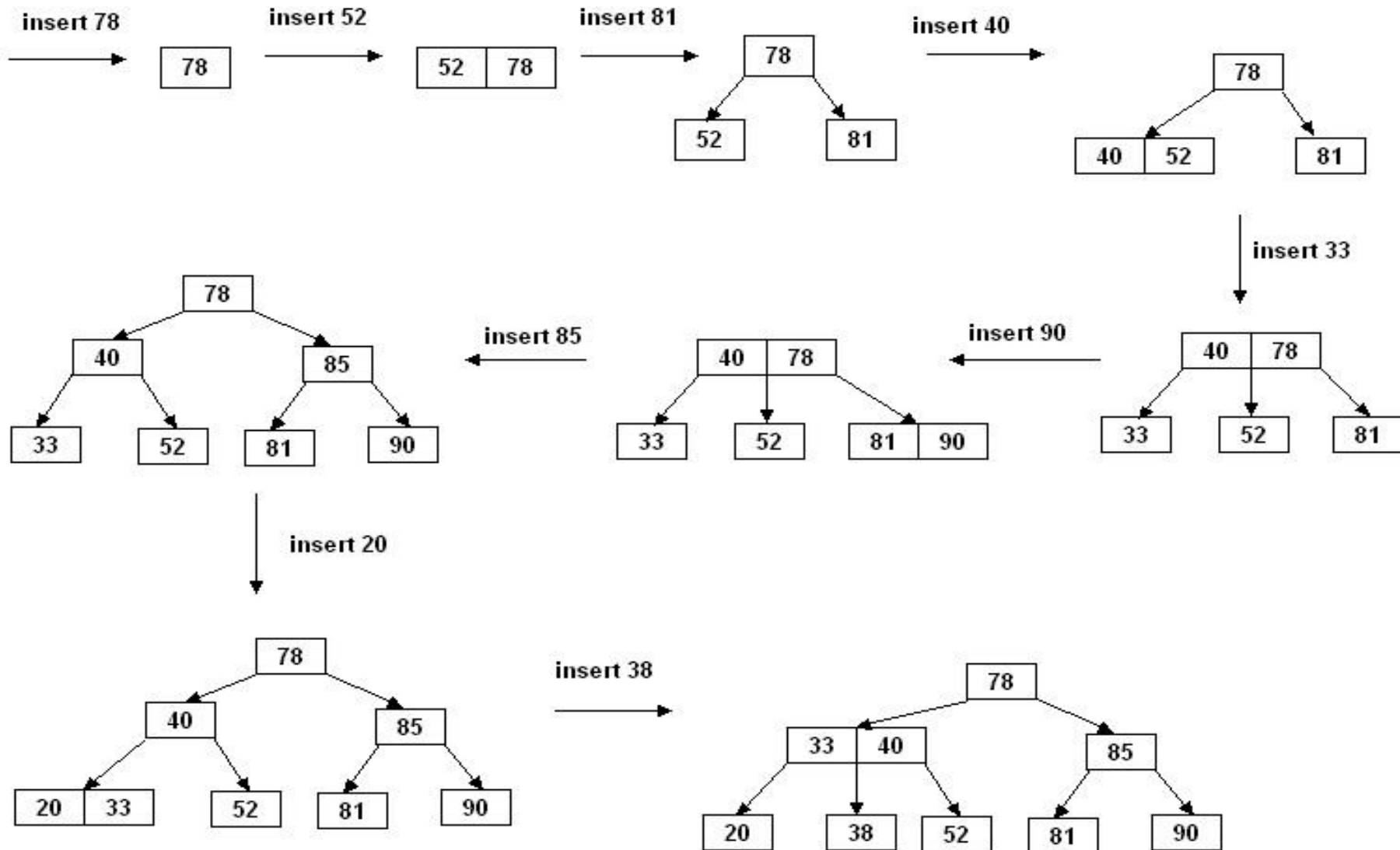
- **Insertion algorithm:**

If a node overflows, split it into two, propagate the "middle" key to the parent of the node. If the parent overflows the process propagates upward. If the node has no parent, create a new root node.

- **Note: Insertion of a key always starts at a leaf node.**

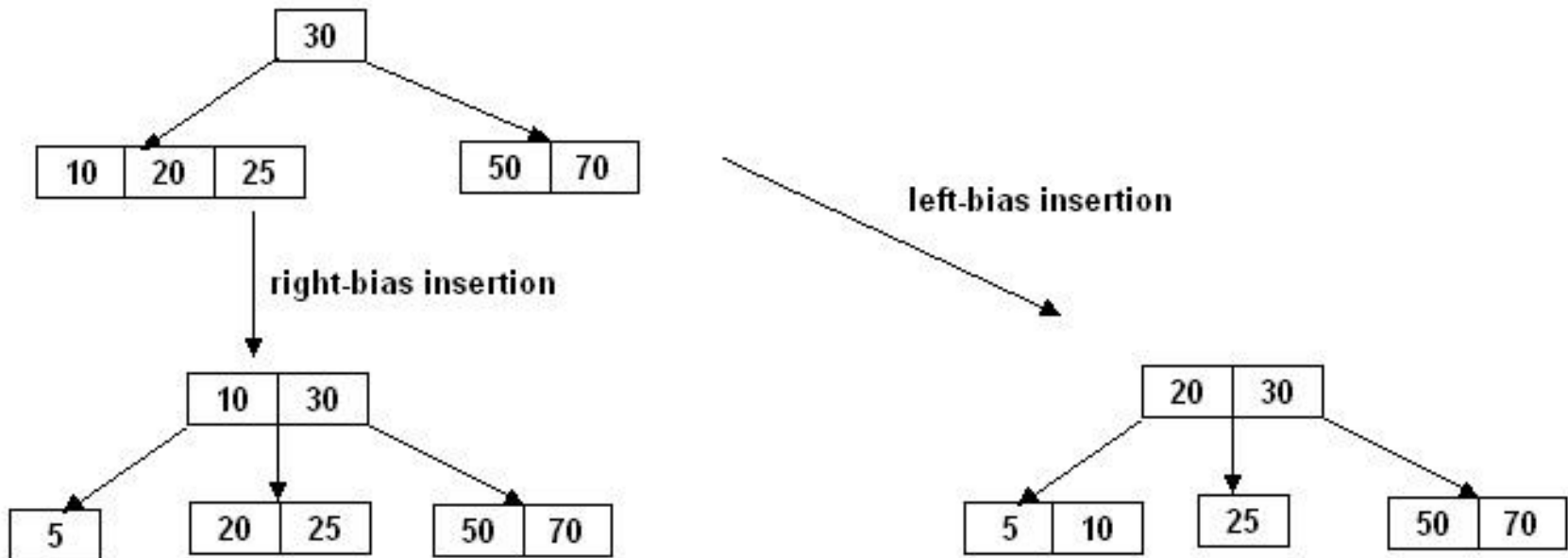
Insertion

- Insertion in a B-tree of odd order
- Example: Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in this order in an initially empty B-tree of order 3



Insertion in B-Trees

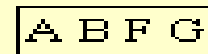
- Insertion in a B-tree of even order
- **right-bias**: The node is split such that its right subtree has more keys than the left subtree.
- **left-bias**: The node is split such that its left subtree has more keys than the right subtree.
- **Example**: Insert the key 5 in the following B-tree of order 4:



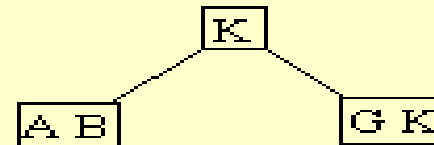
Insertion

- Insert the keys in the following order into a B-tree of order 5.
- A, G, F, B, K, D, H, M, J, E, S, I, R, X, C, L, N, T, U, P.

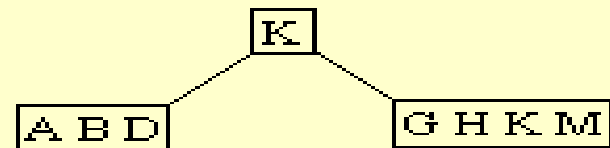
(1) Insert A, G, F and B.



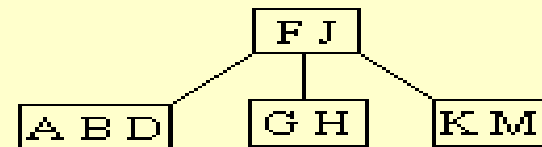
(2) Insert K.



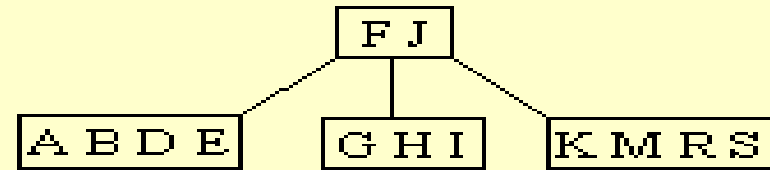
(3) Insert D, H and M.



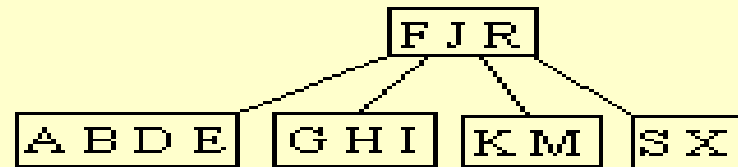
(4) Insert J.



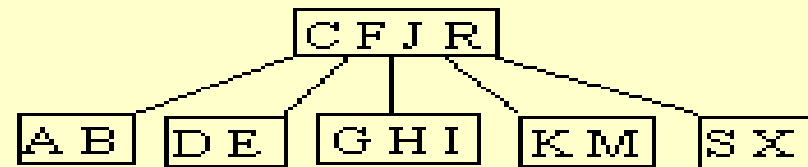
(5) Insert E, S, I and R.



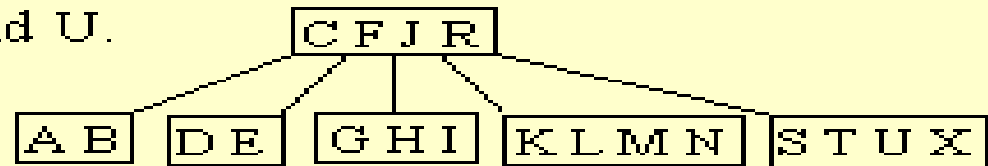
(6) Insert X.



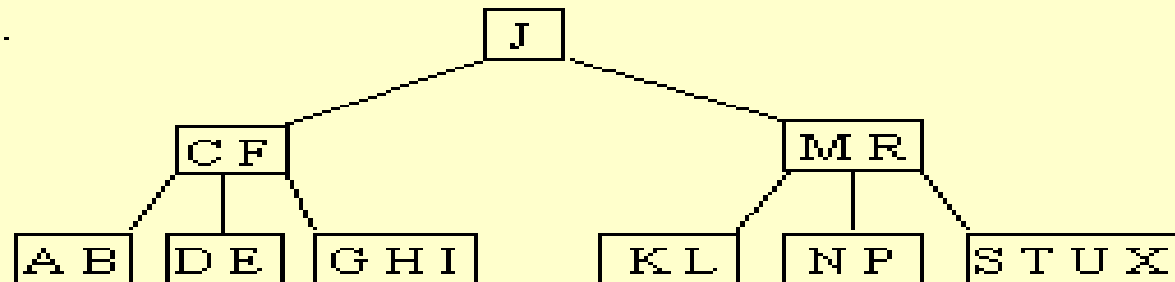
(7) Insert C.



(8) Insert L, N, T and U.



(9) Insert P.



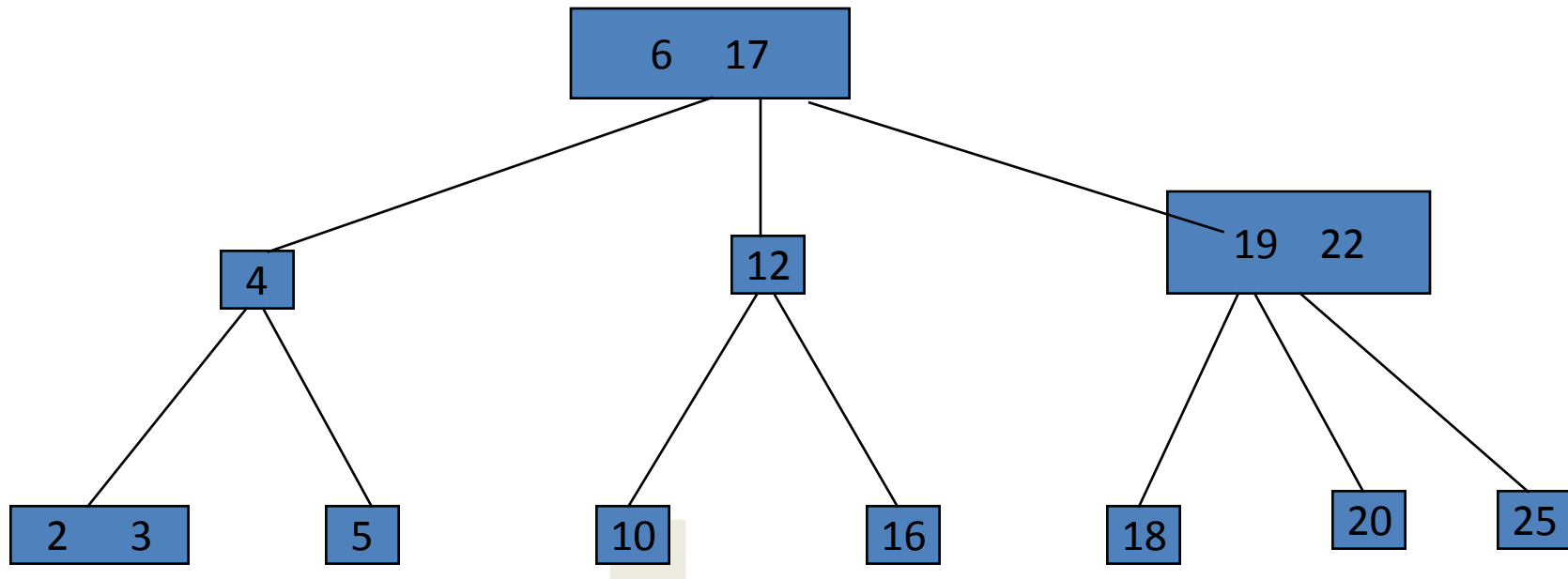
Searching

Searching for an Item in a B-Tree:

1. Make a local variable, i , equal to the first index such that $\text{data}[i] \geq \text{target}$. If there is no such index, then set i equal to data_count , indicating that none of the entries is greater than or equal to the target.
2. if (we found the target at $\text{data}[i]$)
 return true;
 else if (the root has no children)
 return false;
 else
 return $\text{subset}[i] \rightarrow \text{contains}(\text{target})$;

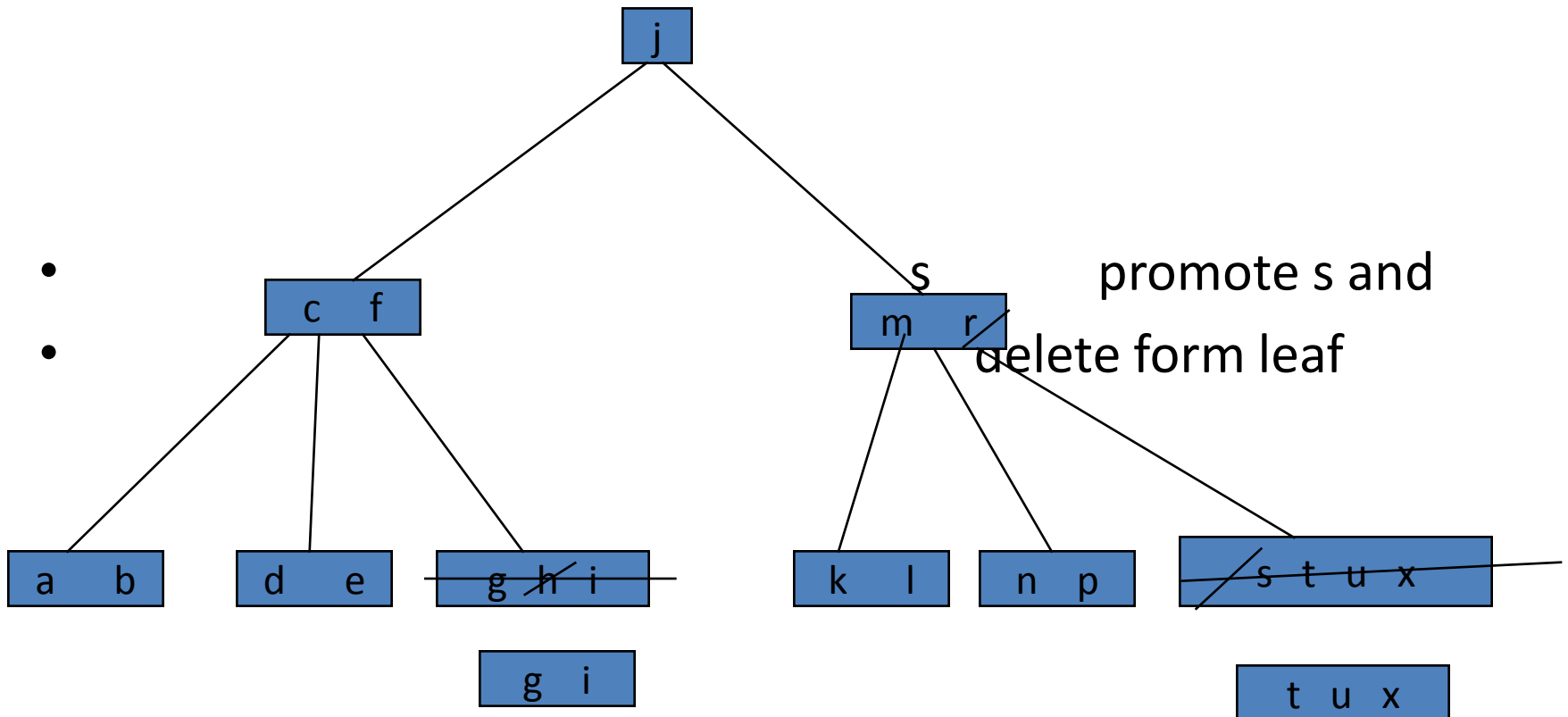
Searching (cont.)

- Example: target = 10



Deletion from a B-Tree

- 1. delete h, r :



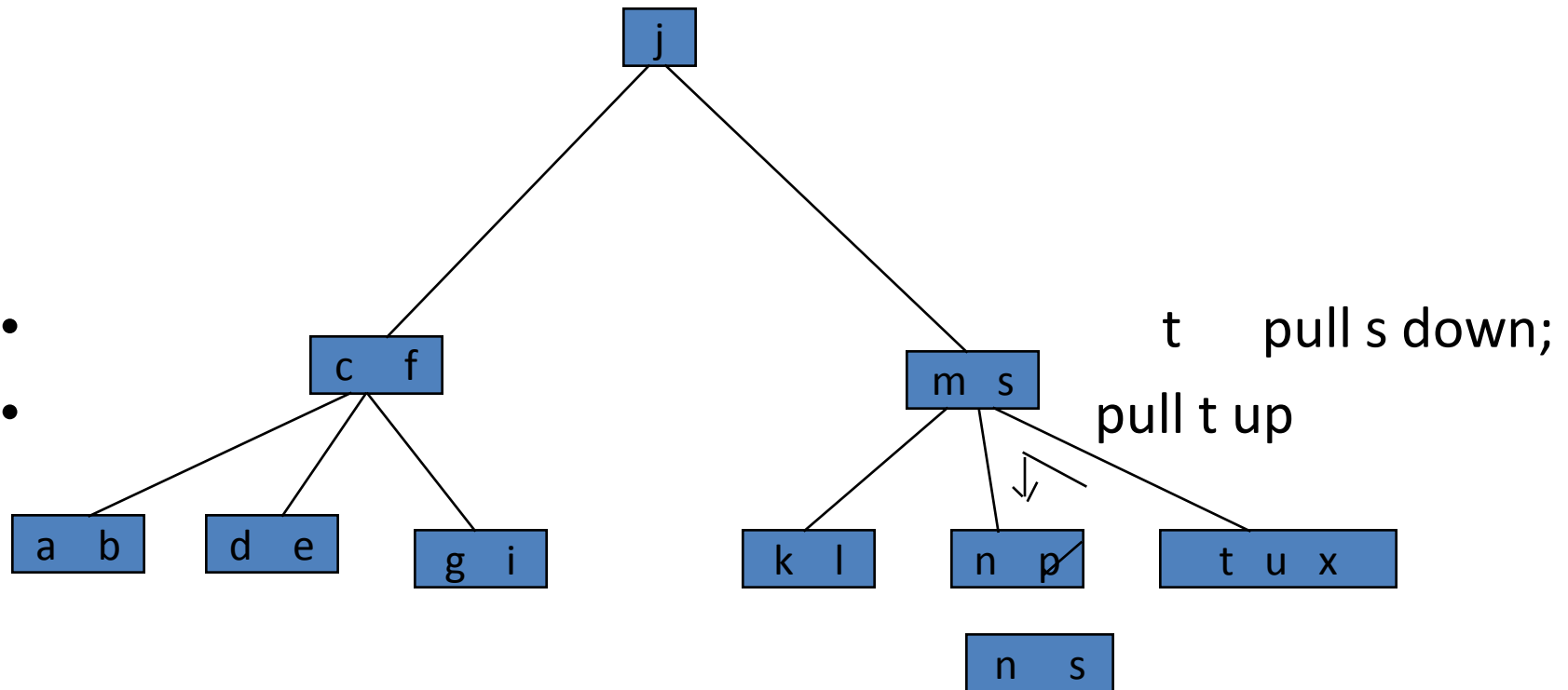
Deletion (cont.)

- 2. delete p :

-

-

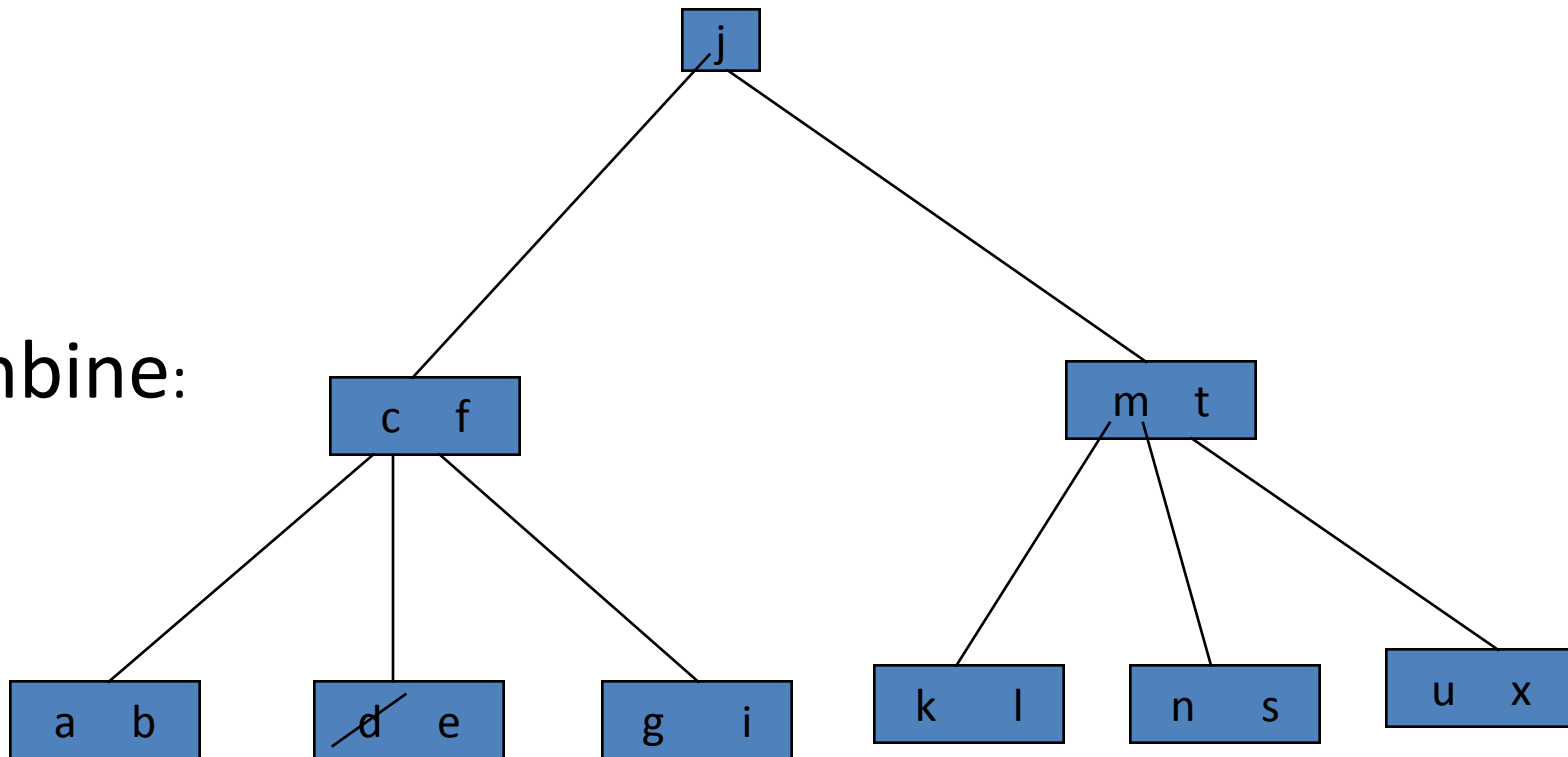
-



Deletion (cont.)

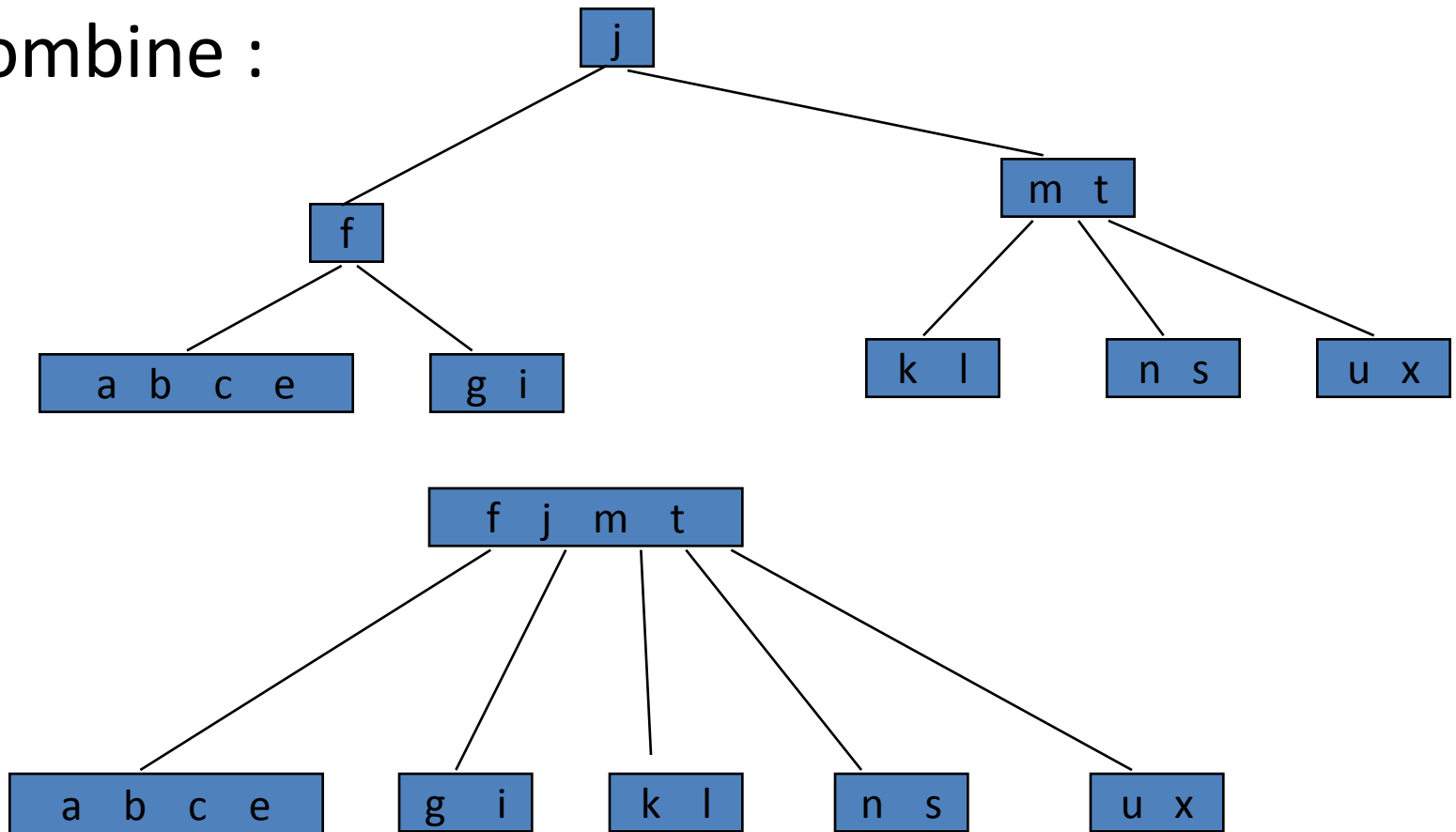
- 3. delete d:

- Combine:



Deletion (cont.)

- combine :



Deleting from a B-Tree

- To delete a key value x from a B-tree, first search to determine the leaf node that contains x .
- If removing x leaves that leaf node with fewer than the minimum number of keys, try to adopt a key from a neighboring node. If that's possible, then you're finished.

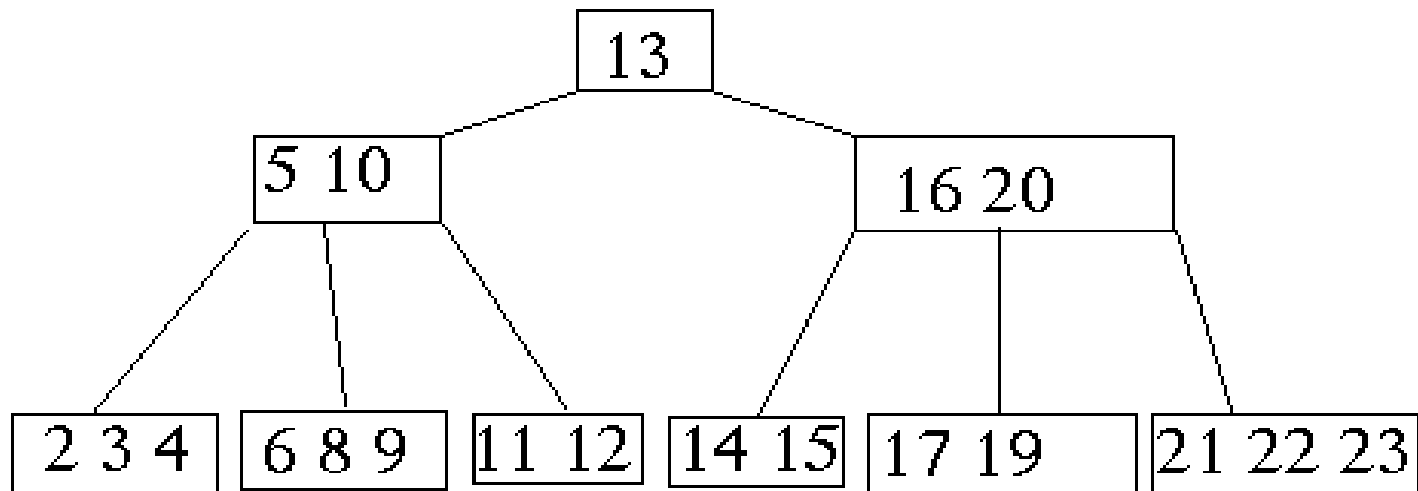
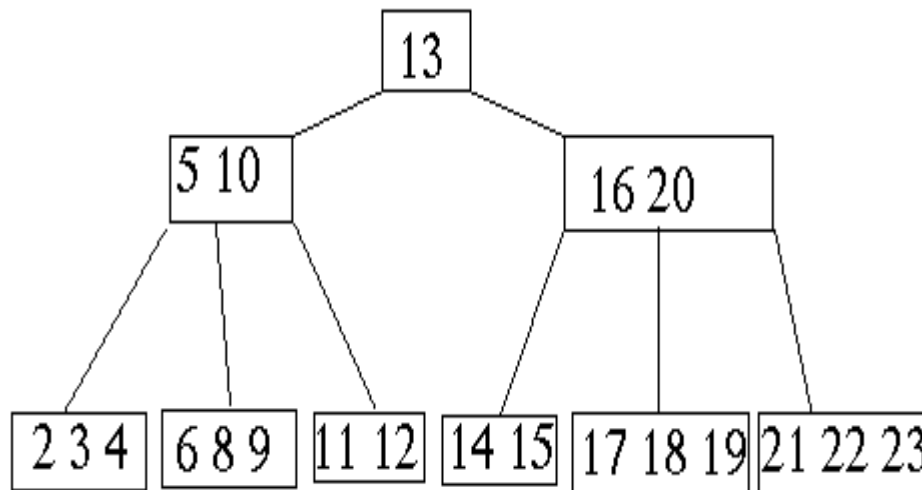
Deleting from a B-Tree (continued)

- If the neighboring node is already at its minimum, combine the leaf node with its neighboring node, resulting in one full leaf node.
- This will require restructuring the parent node since it has lost a child
- If the parent now has fewer than the minimum keys, adopt a key from one of its neighbors. If that's not possible, combine the parent with its neighbor.

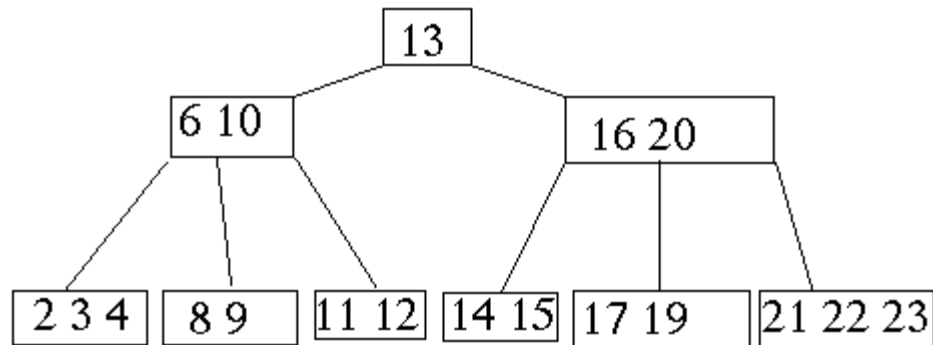
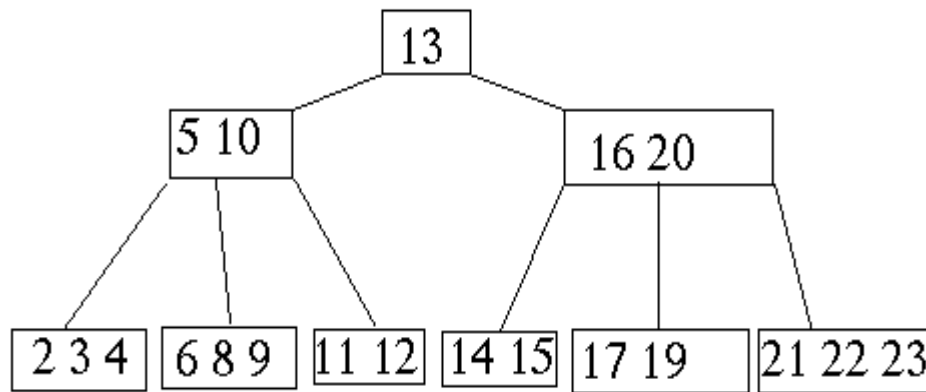
Deleting from a B-Tree (continued)

- This process may percolate all the way to the root.
- If the root is left with only one child, then remove the root node and make its child the new root.
- Both insertion and deletion are $O(h)$, where h is the height of the tree.

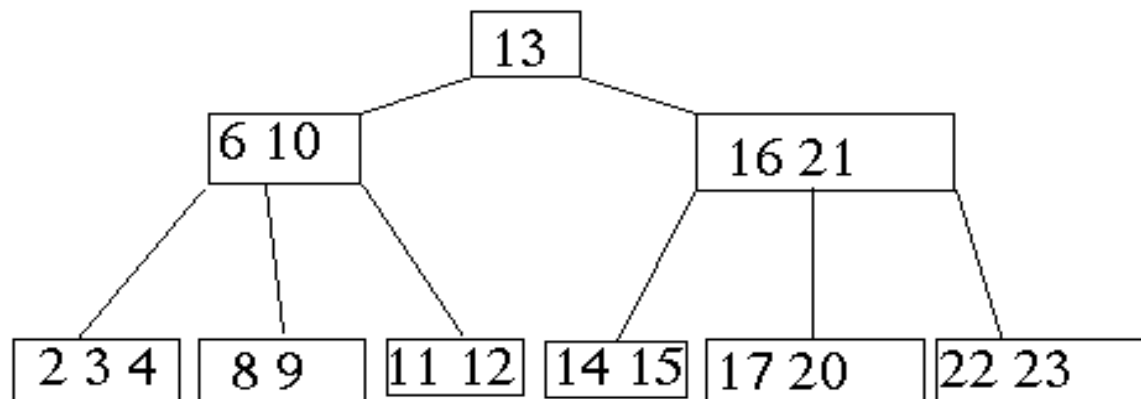
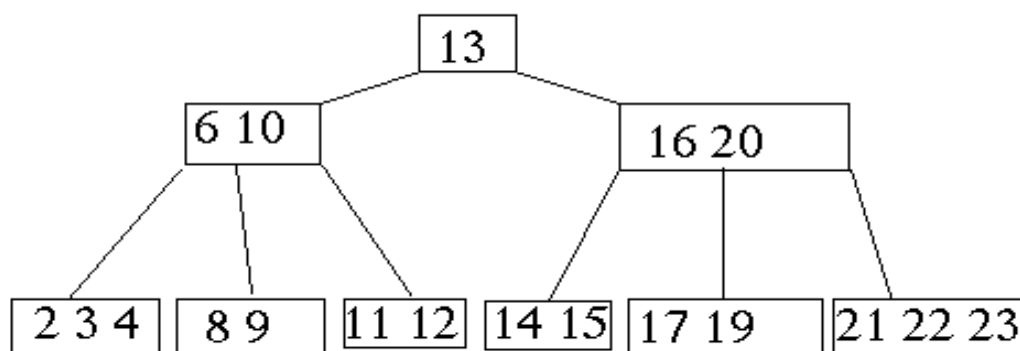
Delete 18



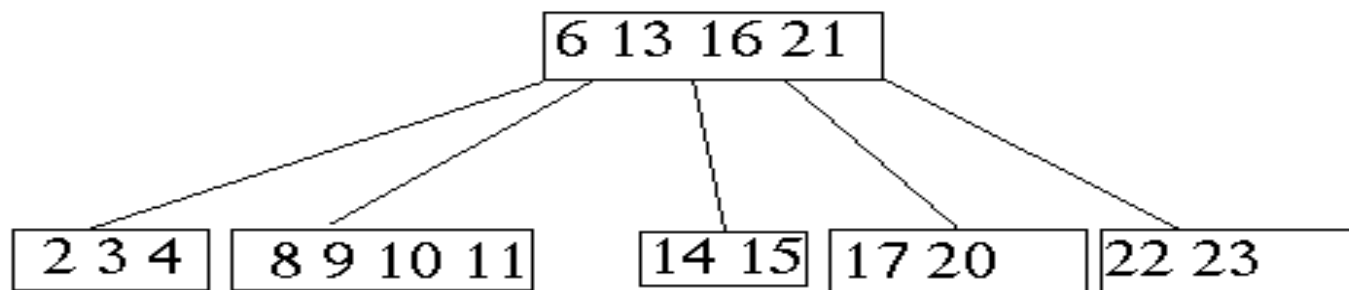
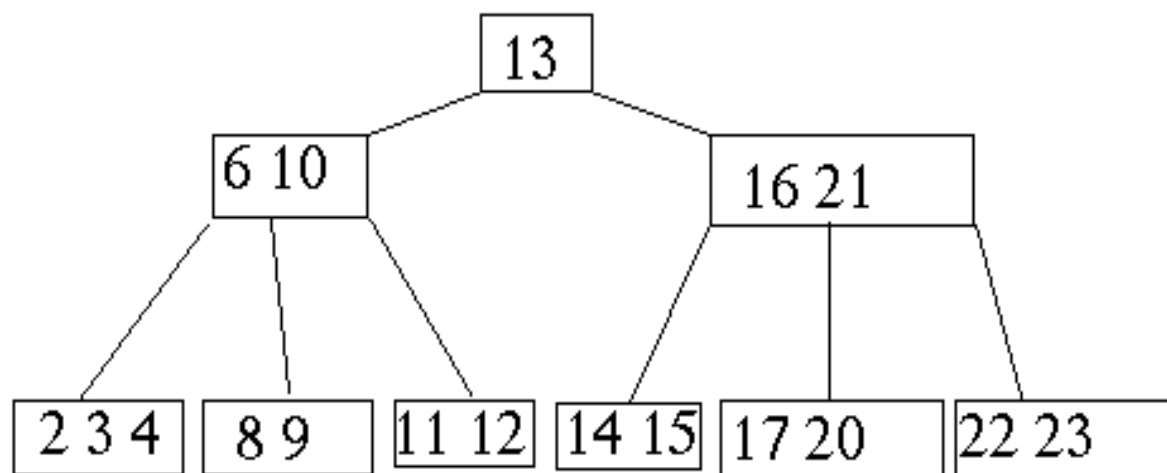
Delete 5



Delete 19



Delete 12



Deletion in B-Tree

- **B-Tree-Delete**

- UNDERFLOW CONDITION
- A non-root node of a B-tree of order m underflows if, after a key deletion, it contains $\lceil m / 2 \rceil - 2$ keys
- The root node does not underflow. If it contains only one key and this key is deleted, the tree becomes empty.

Deletion in B-Tree

- There are **five** deletion cases:

1. The leaf does not underflow.

2. The leaf underflows and the adjacent right sibling has at least $\lceil m / 2 \rceil$ keys.

perform a left key-rotation

3. The leaf underflows and the adjacent left sibling has at least $\lceil m / 2 \rceil$ keys.

perform a right key-rotation

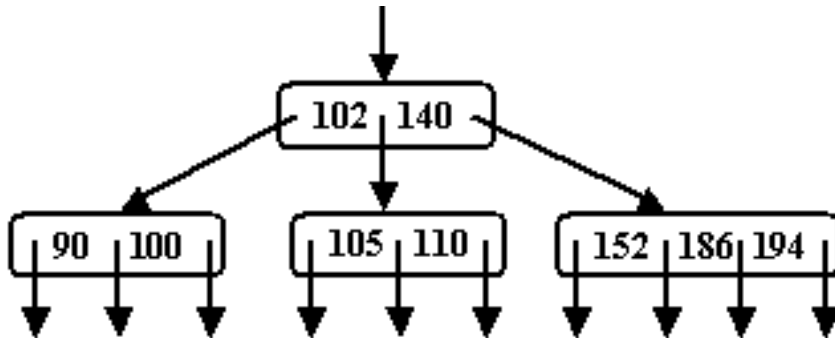
4. The leaf underflows and each of the adjacent right sibling and the adjacent left sibling has at least $\lceil m / 2 \rceil$ keys.

perform either a left or a right key-rotation & perform a merging

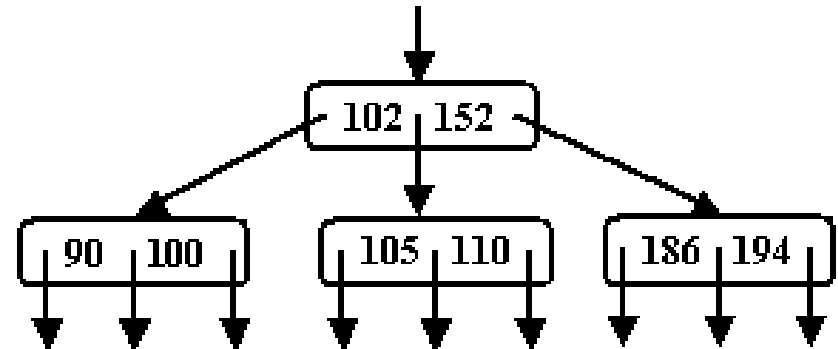
5. The leaf underflows and each adjacent sibling has $\lceil m / 2 \rceil - 1$ keys.

- Case1: The leaf does not underflow.
- Example : B-tree of order 4

Deletion in B-Tree



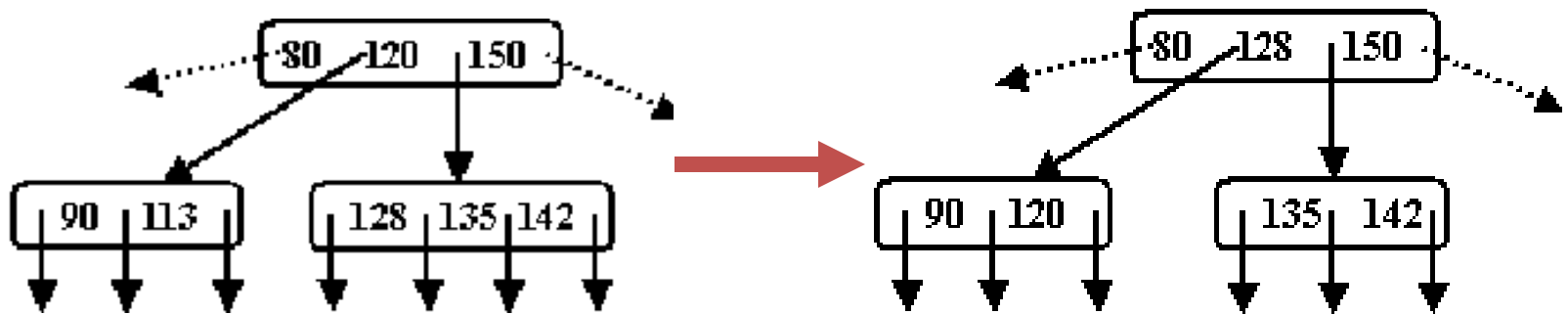
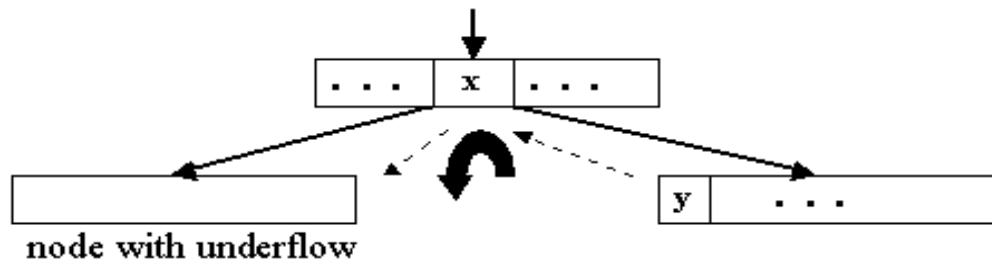
Delete 140



- Case2: The leaf underflows and the adjacent right sibling has at least $\lceil m/2 \rceil$ keys.

Deletion in B-Tree

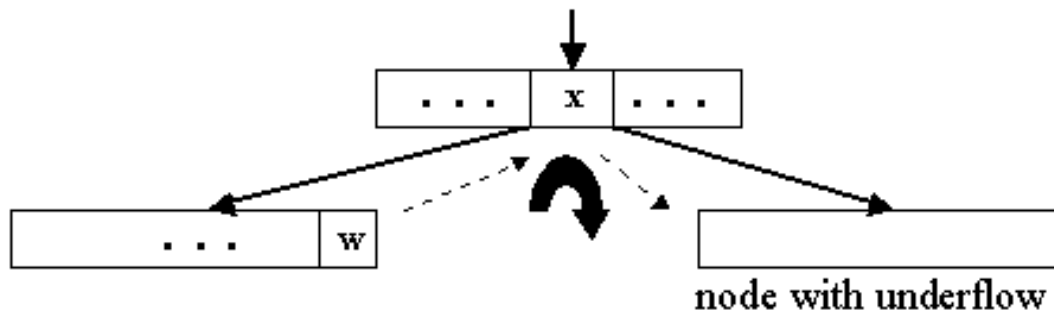
- Example : B-tree of order 5**



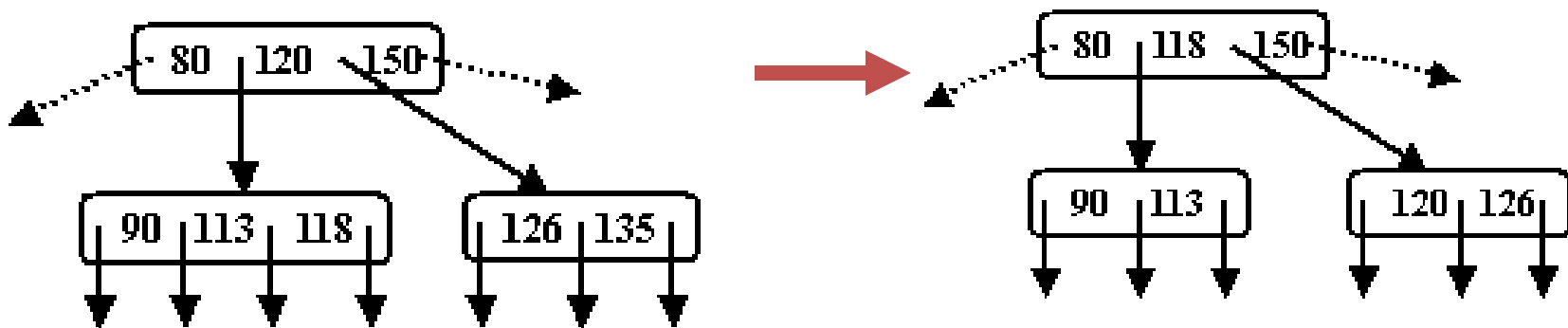
Delete 113

Deletion in B-Tree

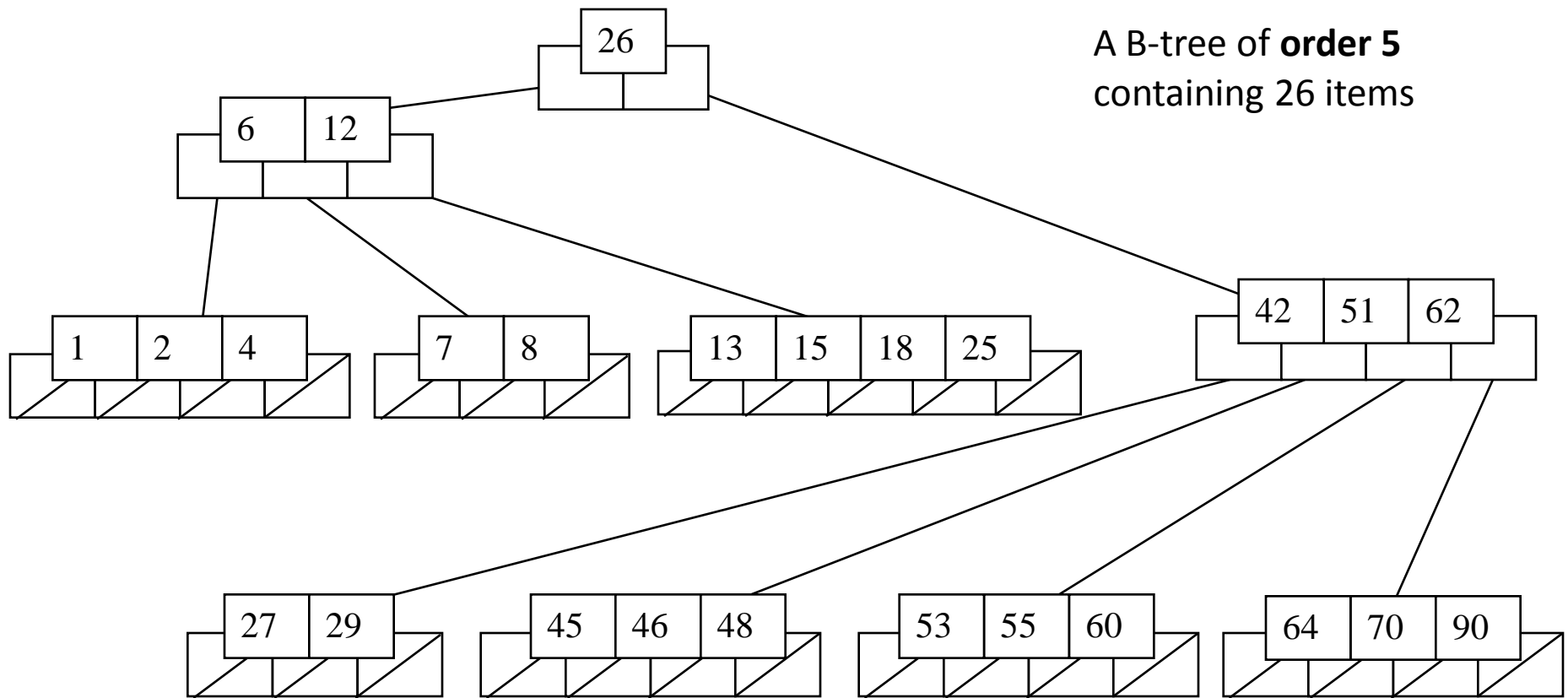
- Case 3: The leaf underflows and the adjacent left sibling has at least $\lceil m / 2 \rceil$ keys.
- **Example : B-tree of order 5**



Delete 135



An example B-Tree



Note that all the leaves are at the same level

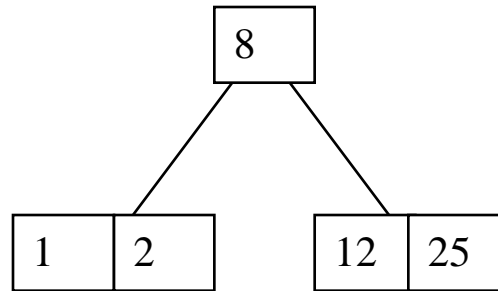
Constructing a B-tree

- Suppose we start with an empty B-tree and keys arrive in the following order: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45
- We want to construct a B-tree of order 5
- The first four items go into the root:

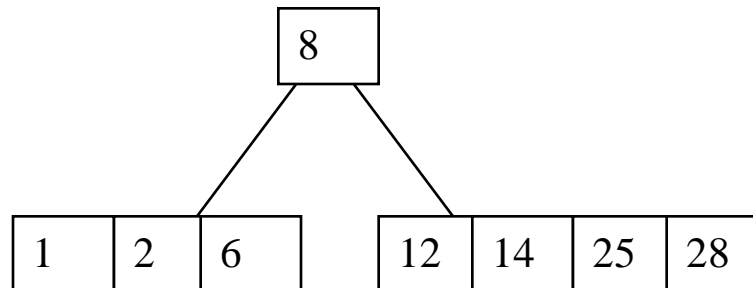
1	2	8	12
---	---	---	----

- To put the fifth item in the root would violate condition 5
- Therefore, when 25 arrives, pick the middle key to make a new root

Constructing a B-tree (contd.)

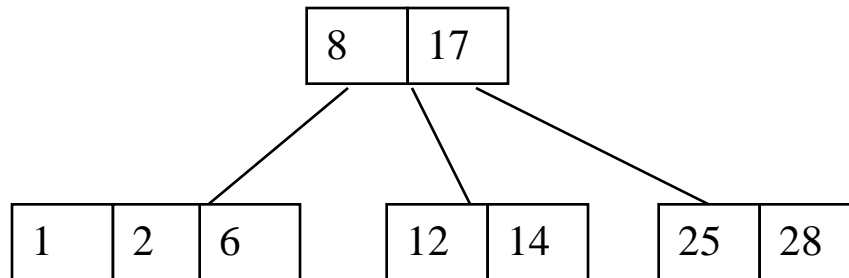


6, 14, 28 get added to the leaf nodes:

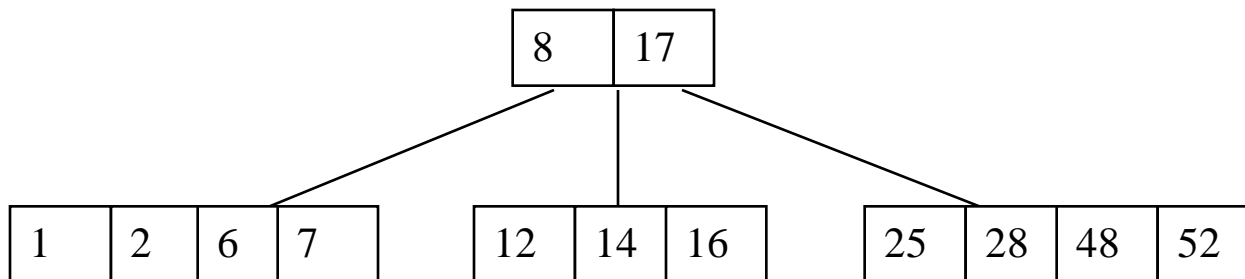


Constructing a B-tree (contd.)

Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf

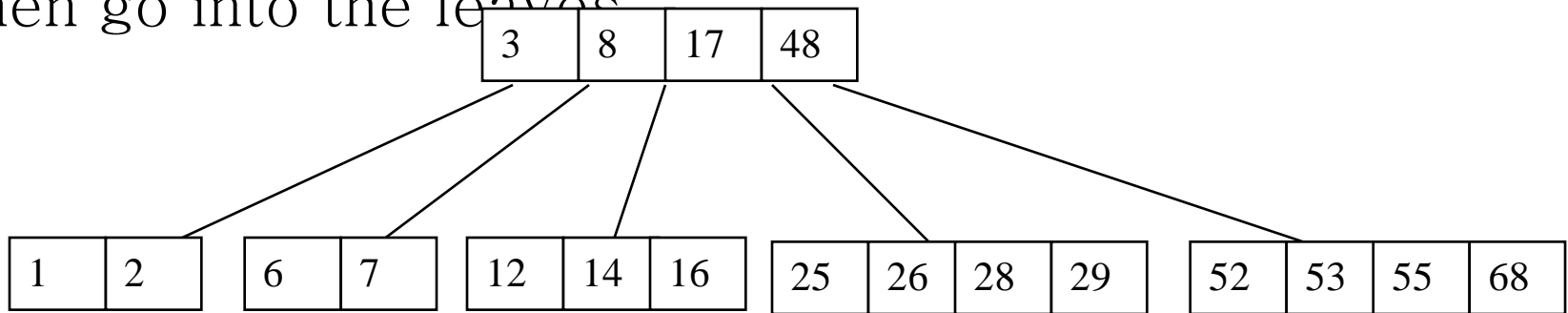


7, 52, 16, 48 get added to the leaf nodes



Constructing a B-tree (contd.)

Adding 68 causes us to split the right most leaf, promoting 48 to the root, and adding 3 causes us to split the left most leaf, promoting 3 to the root; 26, 29, 53, 55 then go into the leaves

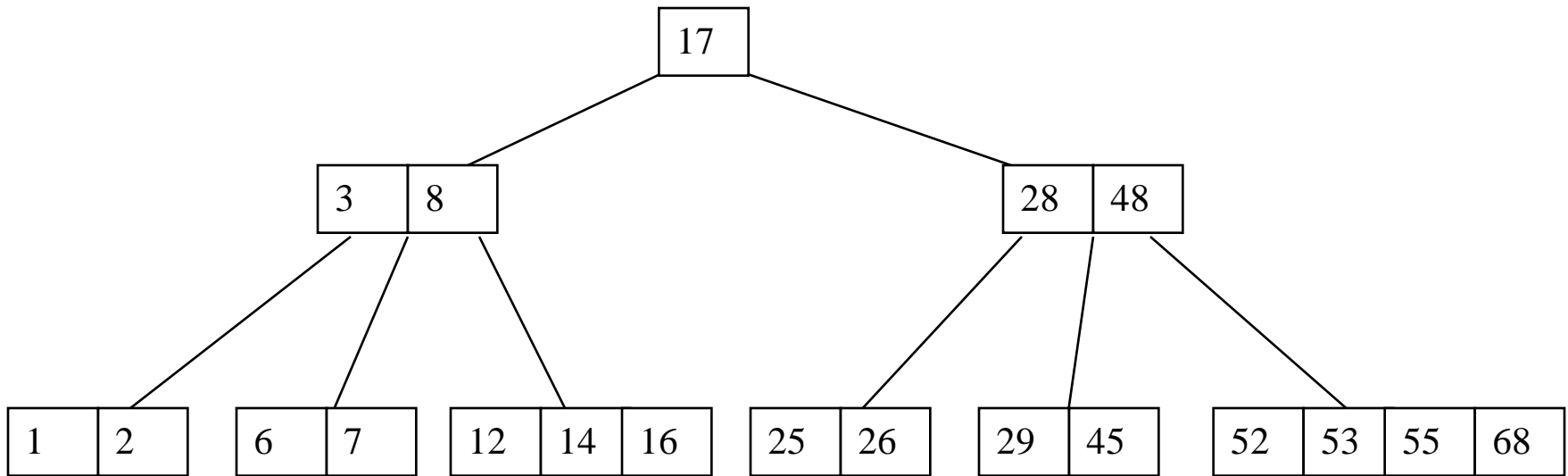


Adding 45 causes a split of

25	26	28	29
----	----	----	----

and promoting 28 to the root then causes the root to split

Constructing a B-tree (contd.)



Inserting into a B-Tree

- Attempt to insert the new key into a leaf
- If this would result in that leaf becoming too big, split the leaf into two, promoting the middle key to the leaf's parent
- If this would result in the parent becoming too big, split the parent into two, promoting the middle key
- This strategy might have to be repeated all the way to the top
- If necessary, the root is split in two and the middle key is promoted to a new root, making the tree one level higher

Exercise in Inserting a B-Tree

- Insert the following keys to a 5-way B-tree:
- 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56

Removal from a B-tree

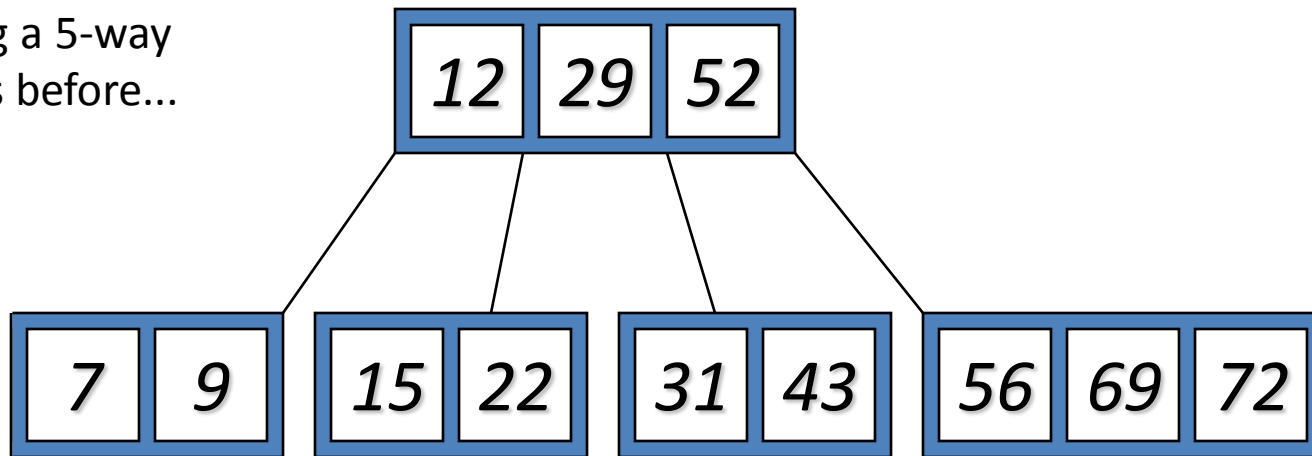
- During insertion, the key always goes *into* a *leaf*. For deletion we wish to remove *from* a leaf. There are three possible ways we can do this:
- 1 - If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted.
- 2 - If the key is *not* in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf -- in this case we can delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.

Removal from a B-tree (2)

- If (1) or (2) lead to a leaf node containing less than the minimum number of keys then we have to look at the siblings immediately adjacent to the leaf in question:
 - 3: if one of them has more than the min. number of keys then we can promote one of its keys to the parent and take the parent key into our lacking leaf
 - 4: if neither of them has more than the min. number of keys then the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key) and the new leaf will have the correct number of keys; if this step leave the parent with too few keys then we repeat the process up to the root itself, if required

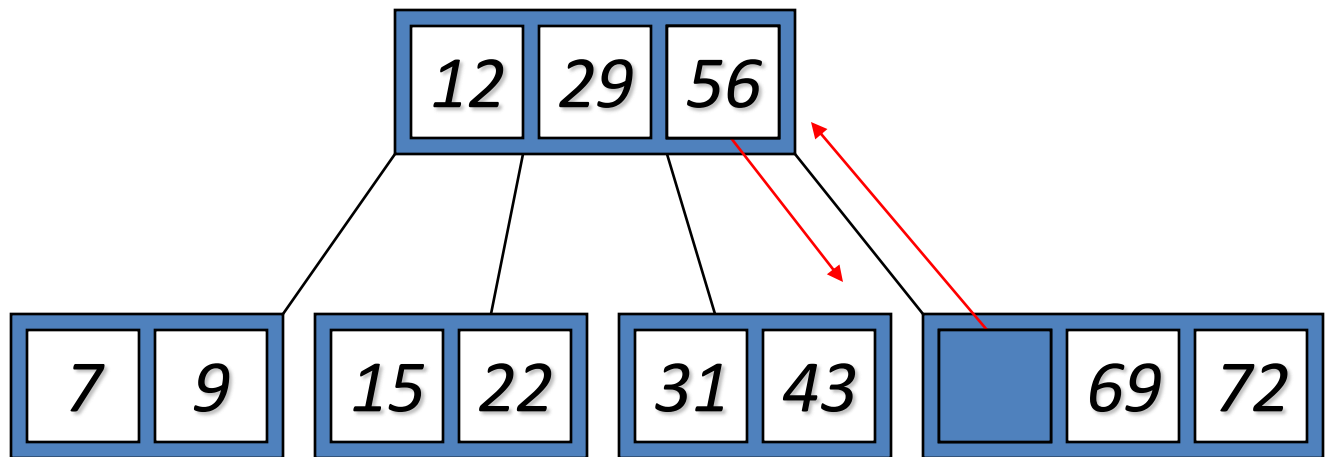
Type #1: Simple leaf deletion

Assuming a 5-way
B-Tree, as before...



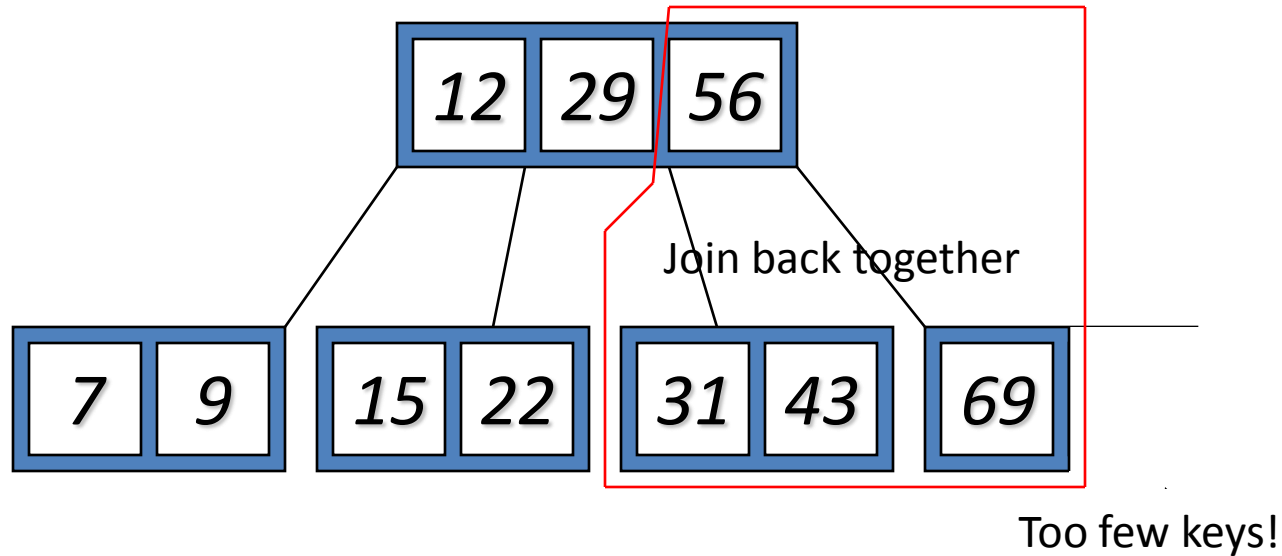
Delete 2: Since there are enough
keys in the node, just delete it

Type #2: Simple non-leaf deletion

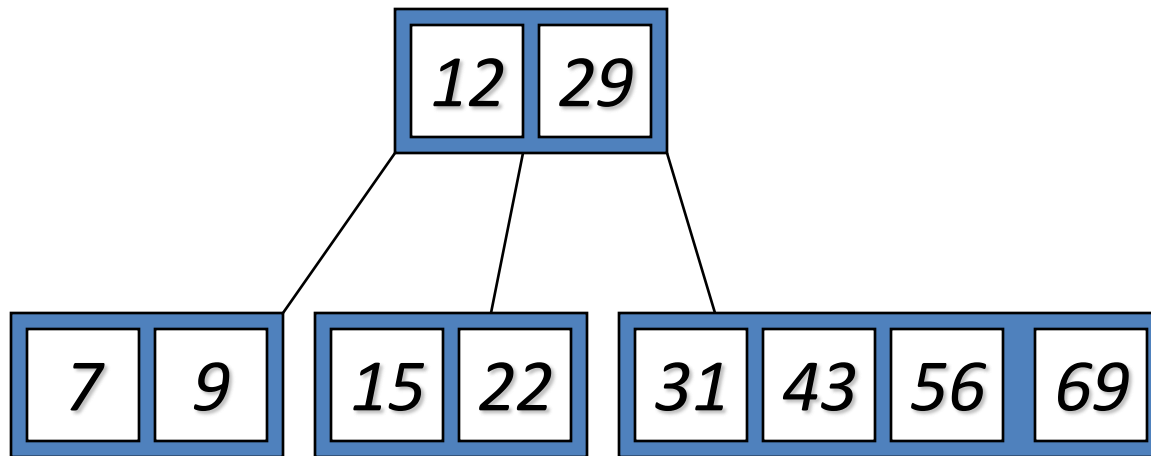


Borrow the predecessor
or (in this case) successor

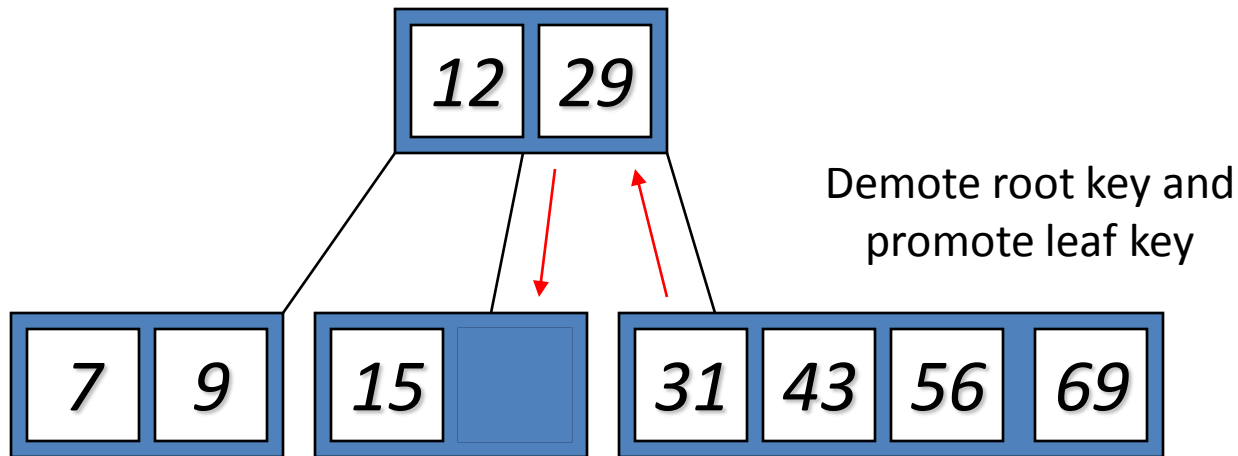
Type #4: Too few keys in node and its siblings



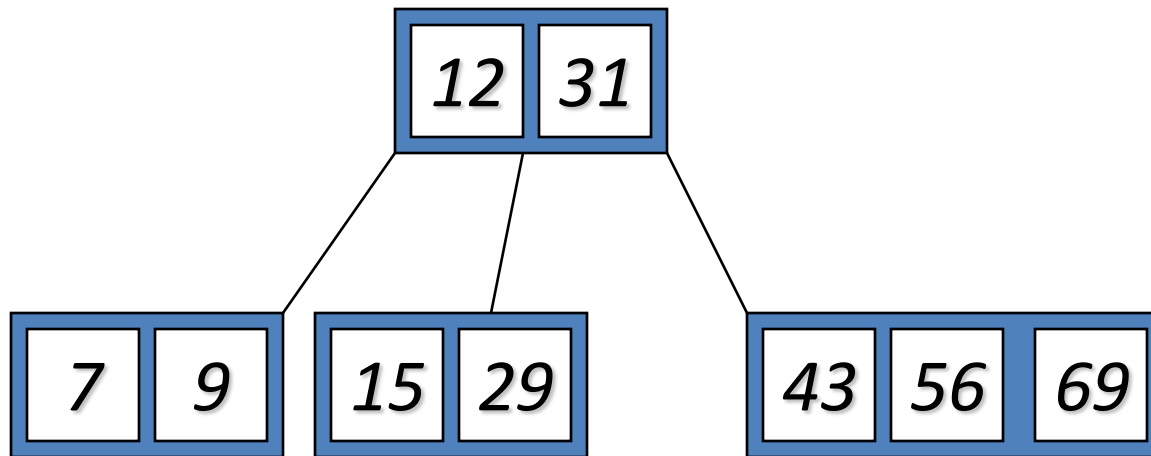
Type #4: Too few keys in node and its siblings



Type #3: Enough siblings



Type #3: Enough siblings

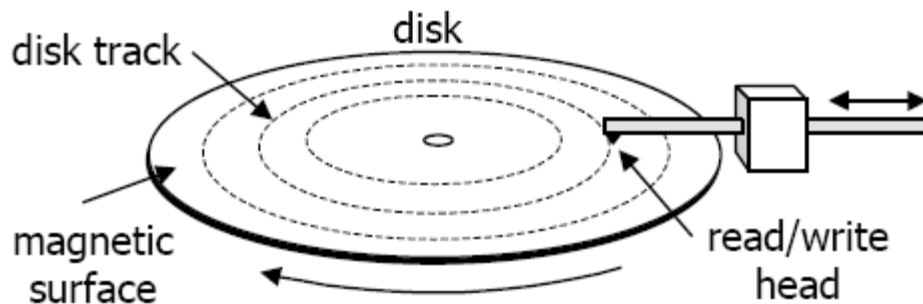


Summary

- The B-tree is a tree-like structure that helps us to organize data in an efficient way.
- The B-tree index is a technique used to minimize the disk I/Os needed for the purpose of locating a row with a given index key value.
- Because of its advantages, the B-tree and the B-tree index structure are widely used in databases nowadays.
- In addition to its use in databases, the B-tree is also used in file systems to allow quick random access to an arbitrary block in a particular file. The basic problem is turning the file block i address into a disk block.

MS/Dos - FAT (File allocation table)

- entry for each disk block
- entry identifies whether its block is used by a file
- which block (if any) is the next disk block of the same file
- allocation of each file is represented as a linked list in the table



Secondary Storages

Red-Black Trees

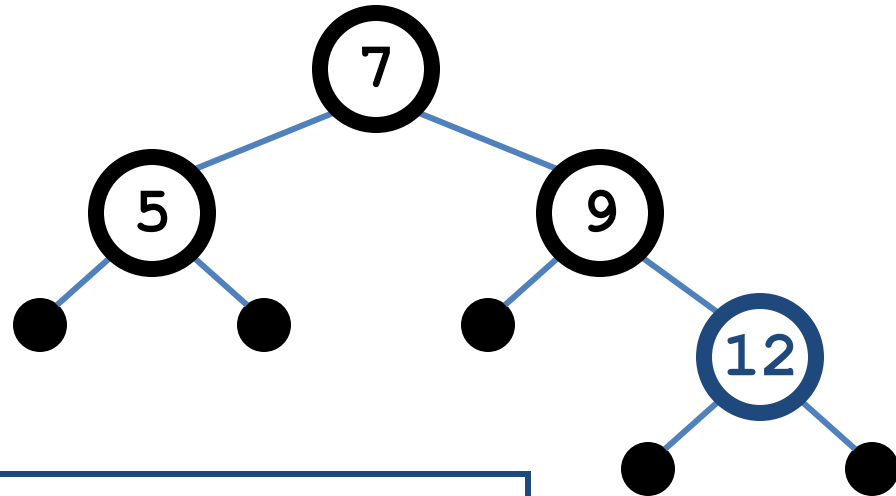
Red-Black Properties

The red-black properties:

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
Note: this means every “real” node has 2 children
3. If a node is red, both children are black
Note: can’t have 2 consecutive reds on a path
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: An Example

- Color this tree:



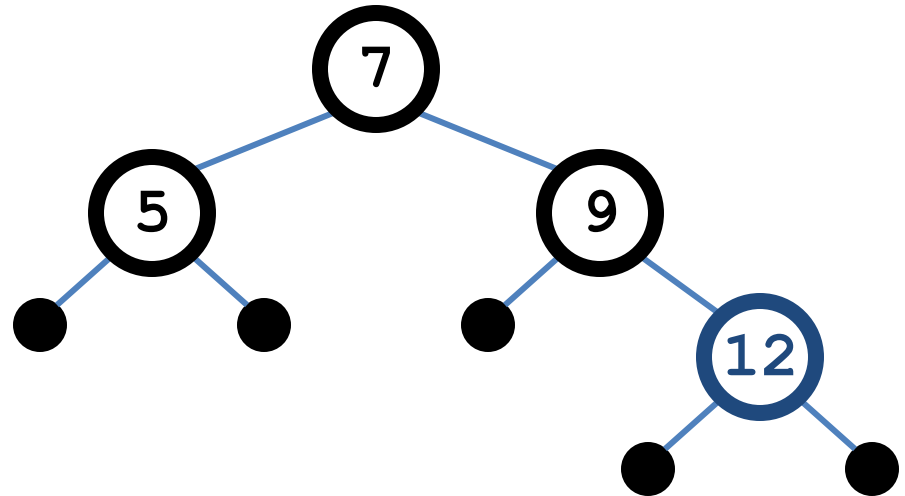
Red-black properties:

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees:

The Problem With Insertion

- Insert 8
 - Where does it go?

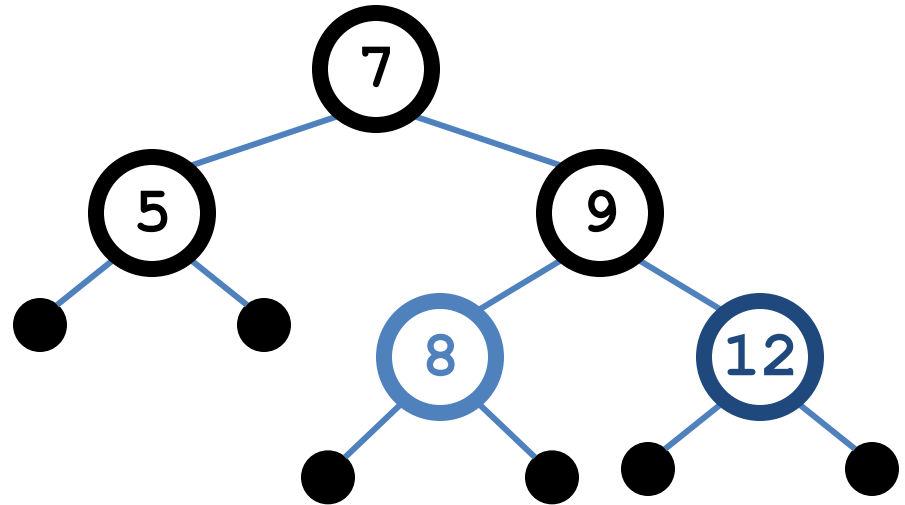


1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees:

The Problem With Insertion

- Insert 8
 - Where does it go?
 - What color should it be?

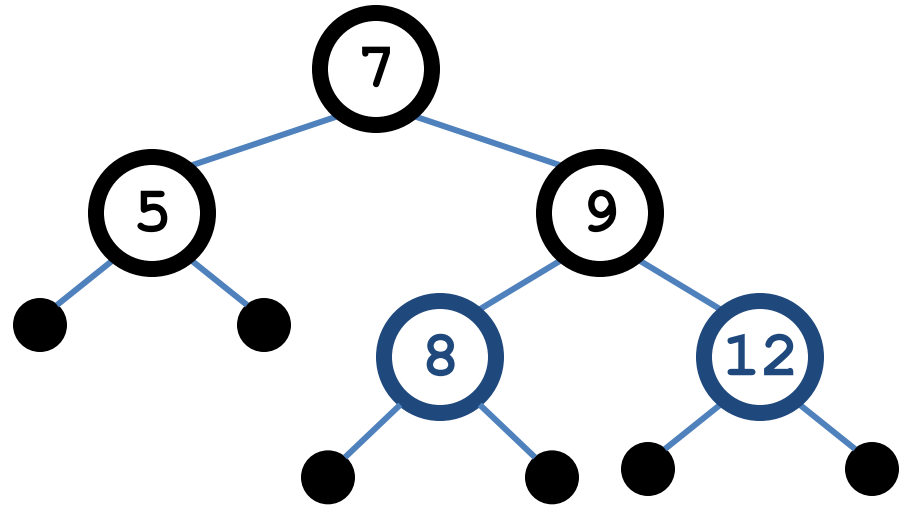


1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees:

The Problem With Insertion

- Insert 8
 - Where does it go?
 - What color should it be?

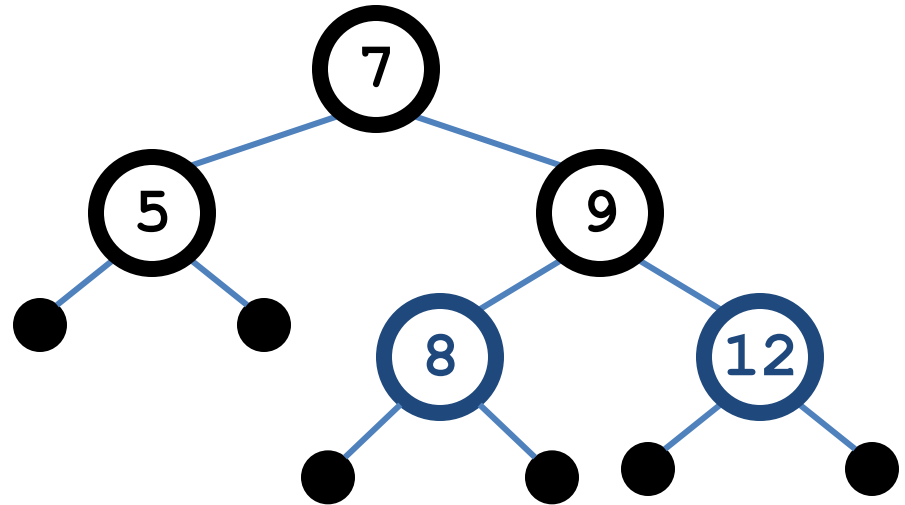


1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees:

The Problem With Insertion

- Insert 11
 - Where does it go?

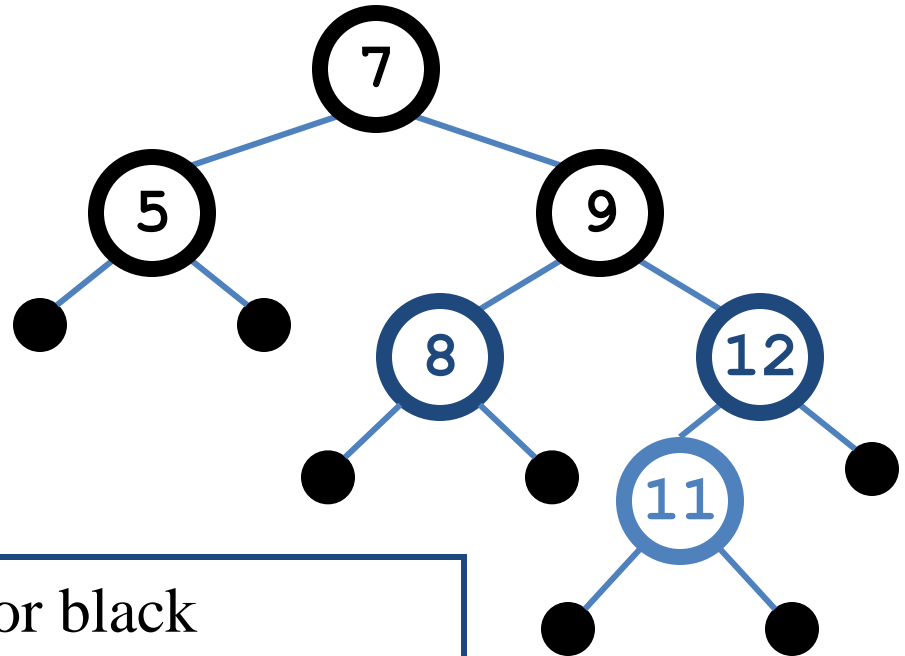


1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees:

The Problem With Insertion

- Insert 11
 - Where does it go?
 - What color?

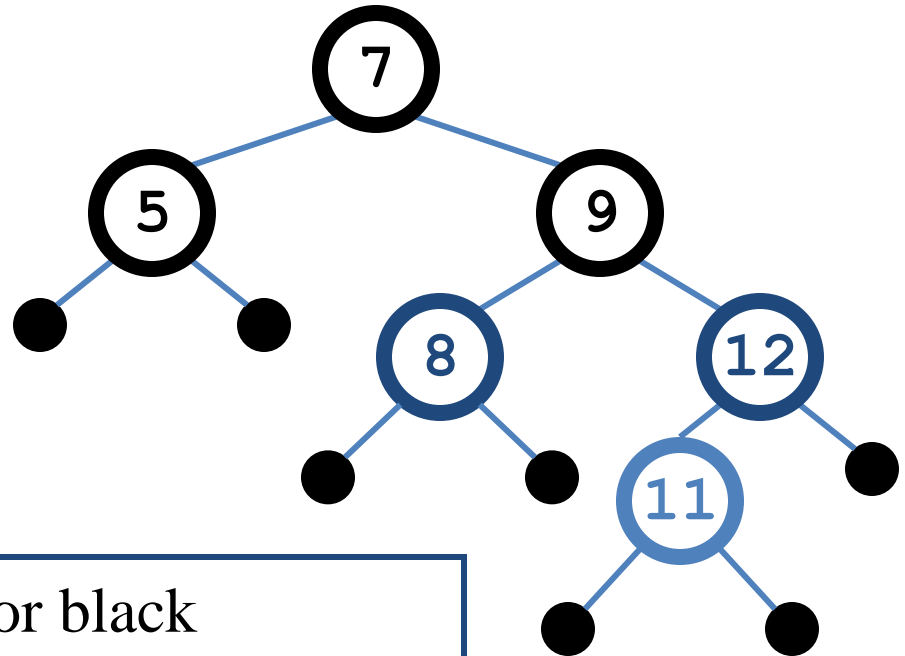


1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees:

The Problem With Insertion

- Insert 11
 - Where does it go?
 - What color?
 - Can't be red! (#3)

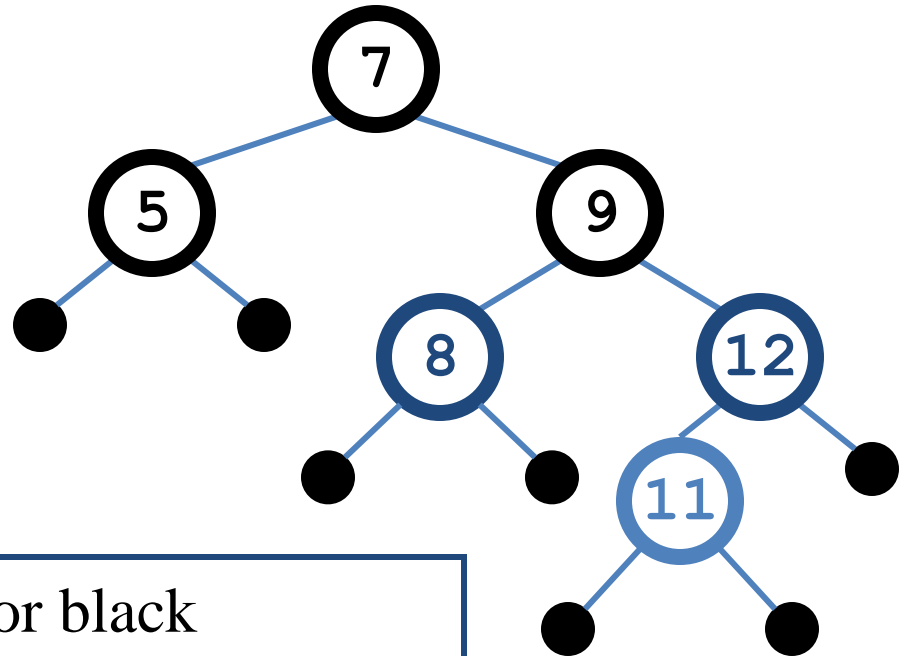


1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees:

The Problem With Insertion

- Insert 11
 - Where does it go?
 - What color?
 - Can't be red! (#3)
 - Can't be black! (#4)

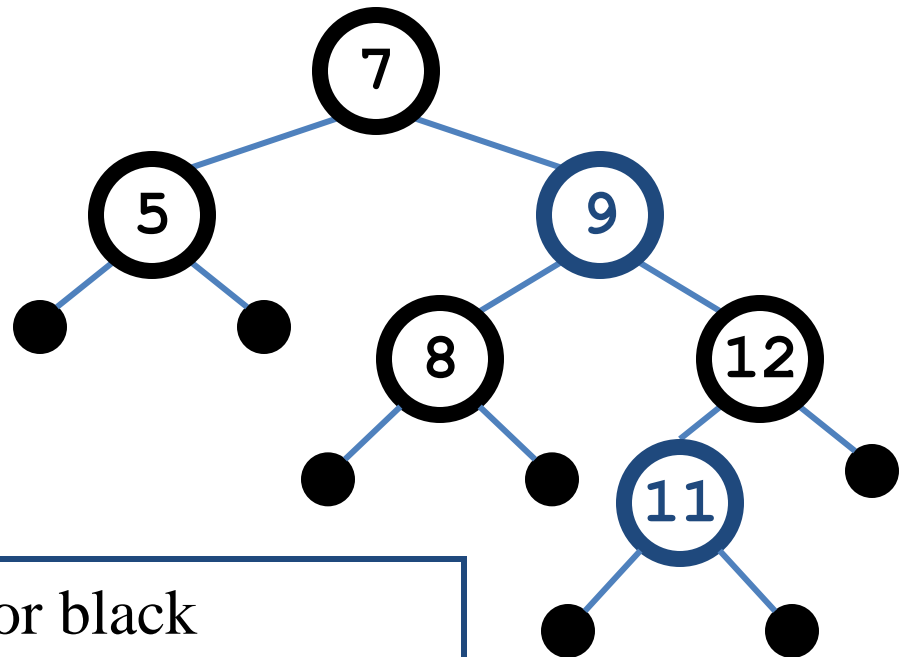


1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees:

The Problem With Insertion

- Insert 11
 - Where does it go?
 - What color?
 - Solution:
recolor the tree

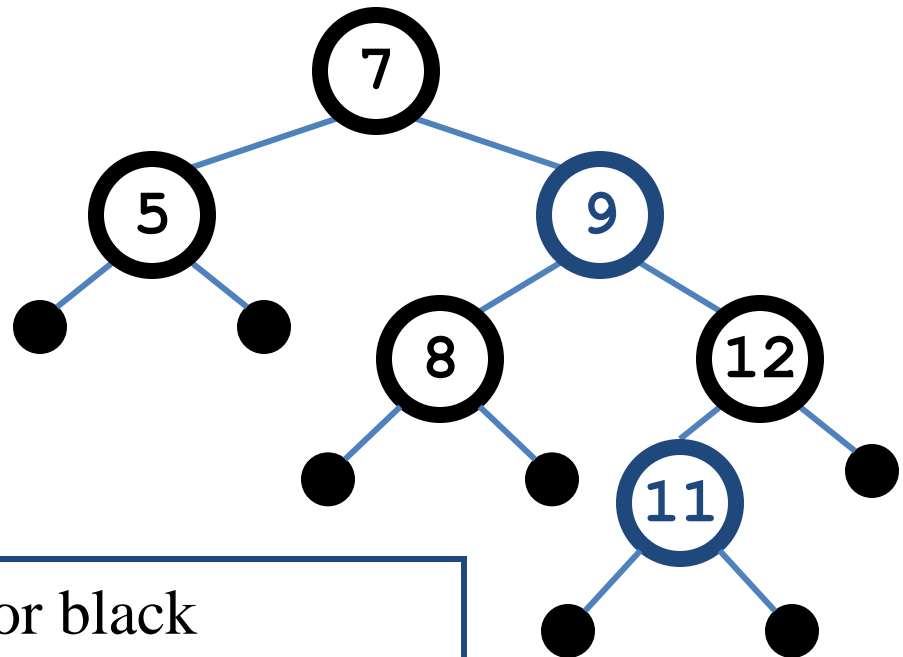


1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees:

The Problem With Insertion

- Insert 10
 - Where does it go?

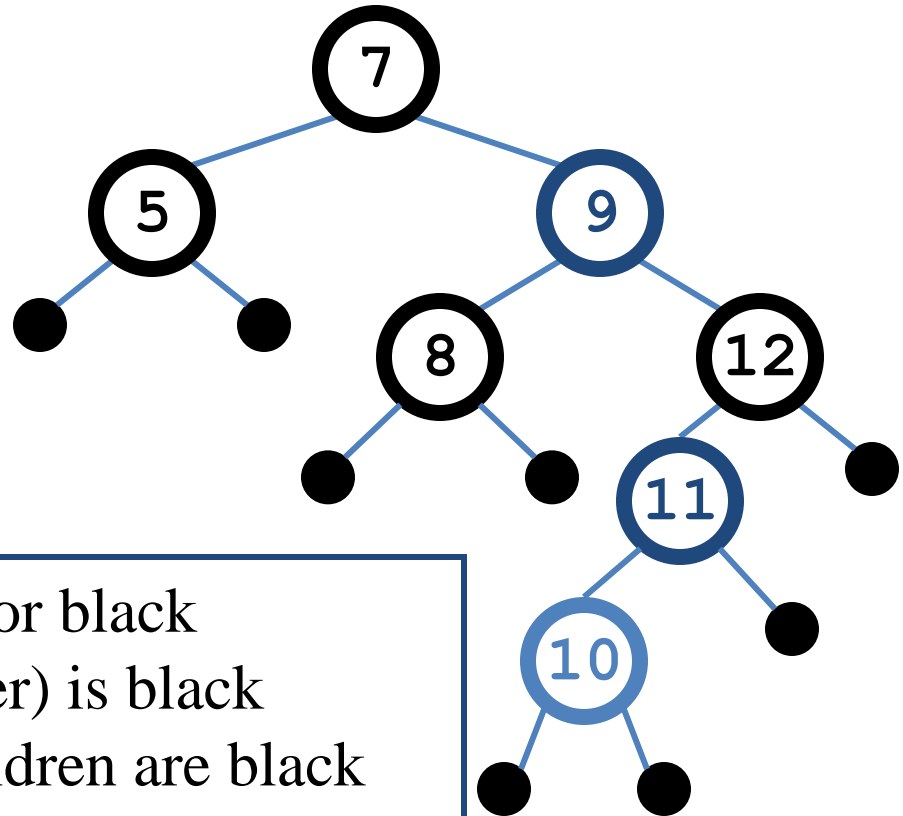


1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees:

The Problem With Insertion

- Insert 10
 - Where does it go?
 - What color?

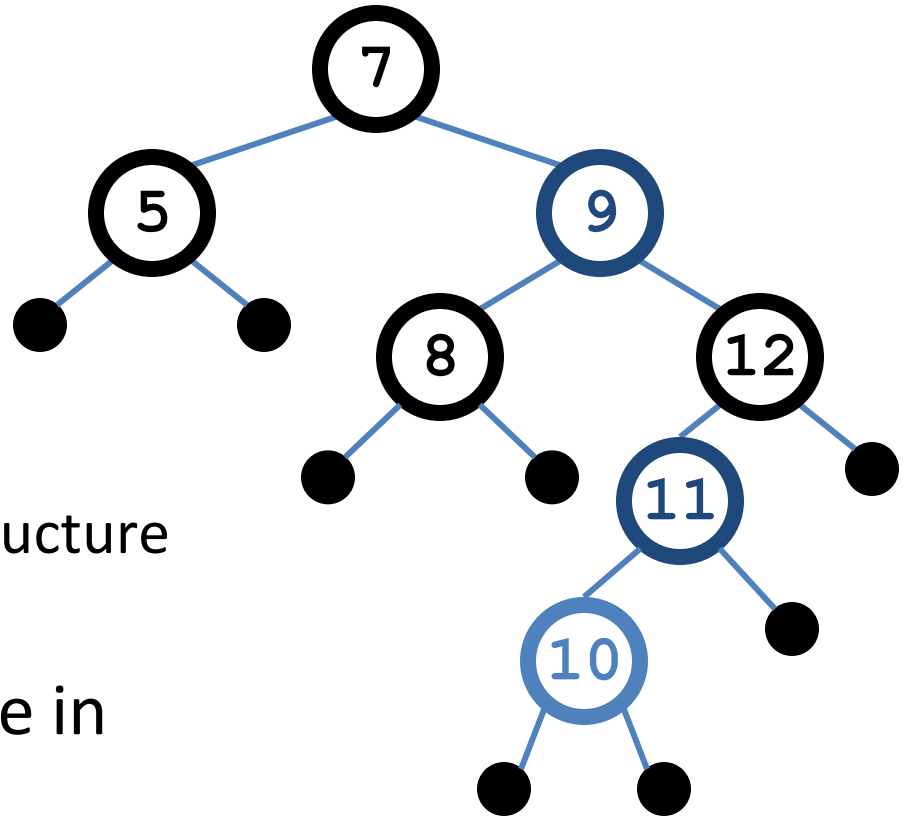


1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees:

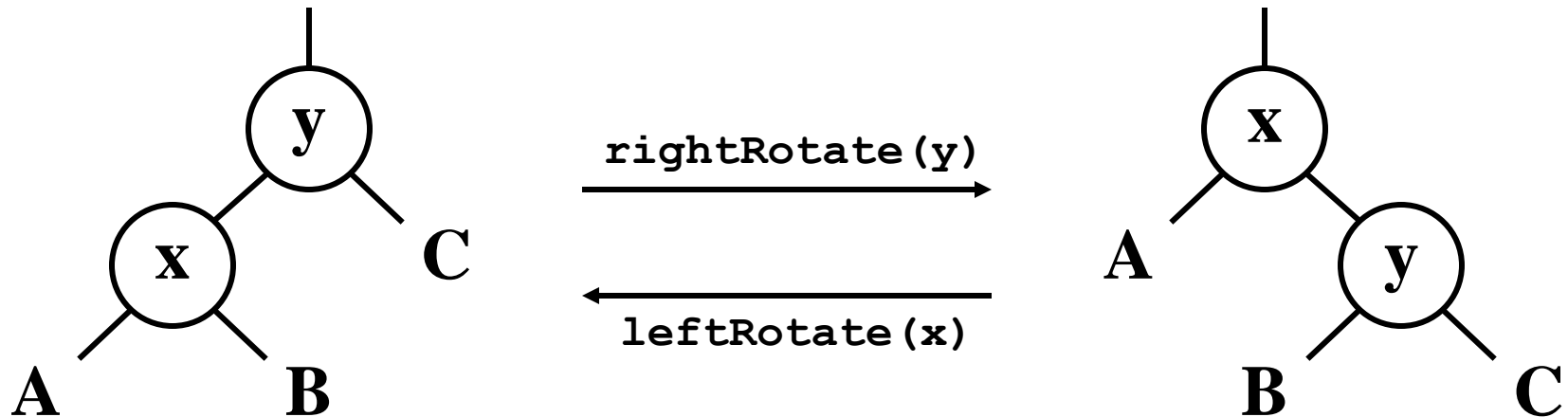
The Problem With Insertion

- Insert 10
 - Where does it go?
 - What color?
 - A: no color! Tree is too imbalanced
 - Must change tree structure to allow recoloring
 - Goal: restructure tree in $O(\lg n)$ time



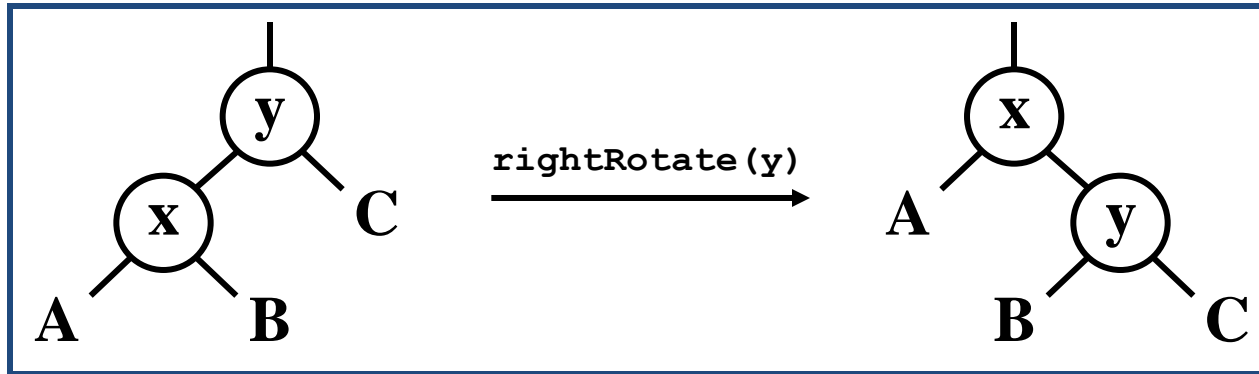
RB Trees: Rotation

- Our basic operation for changing tree structure is called **rotation**:



- Does rotation preserve inorder key ordering?
- What would the code for `rightRotate()` actually do?

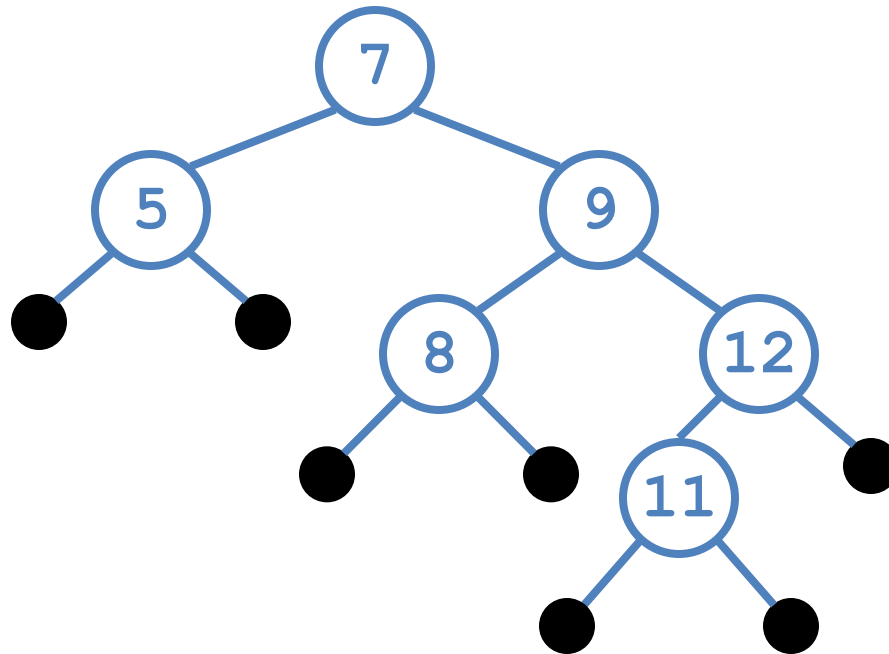
RB Trees: Rotation



- Answer: A lot of pointer manipulation
 - **x** keeps its left child
 - **y** keeps its right child
 - **x**'s right child becomes **y**'s left child
 - **x**'s and **y**'s parents change
- What is the running time?

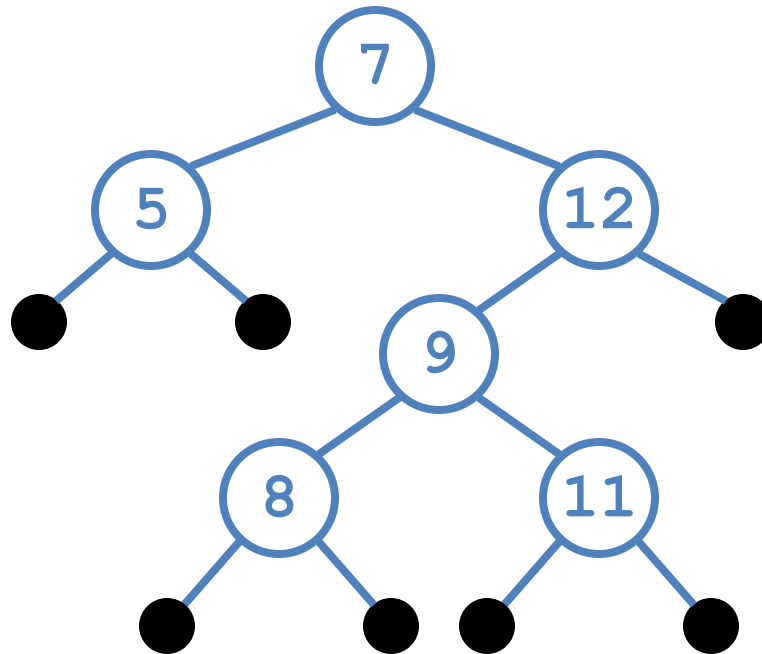
Rotation Example

- Rotate left about 9:

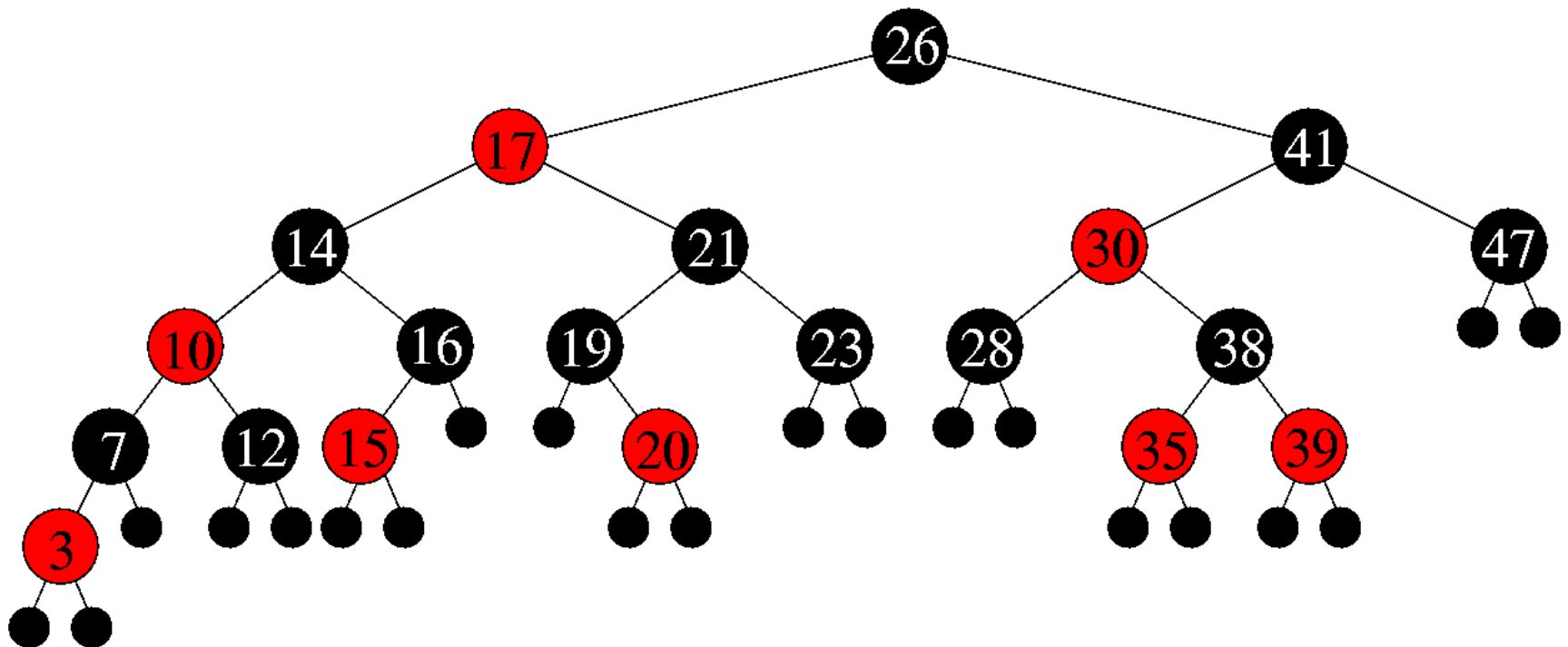


Rotation Example

- Rotate left about 9:



Example Red-Black Tree



Splay Trees

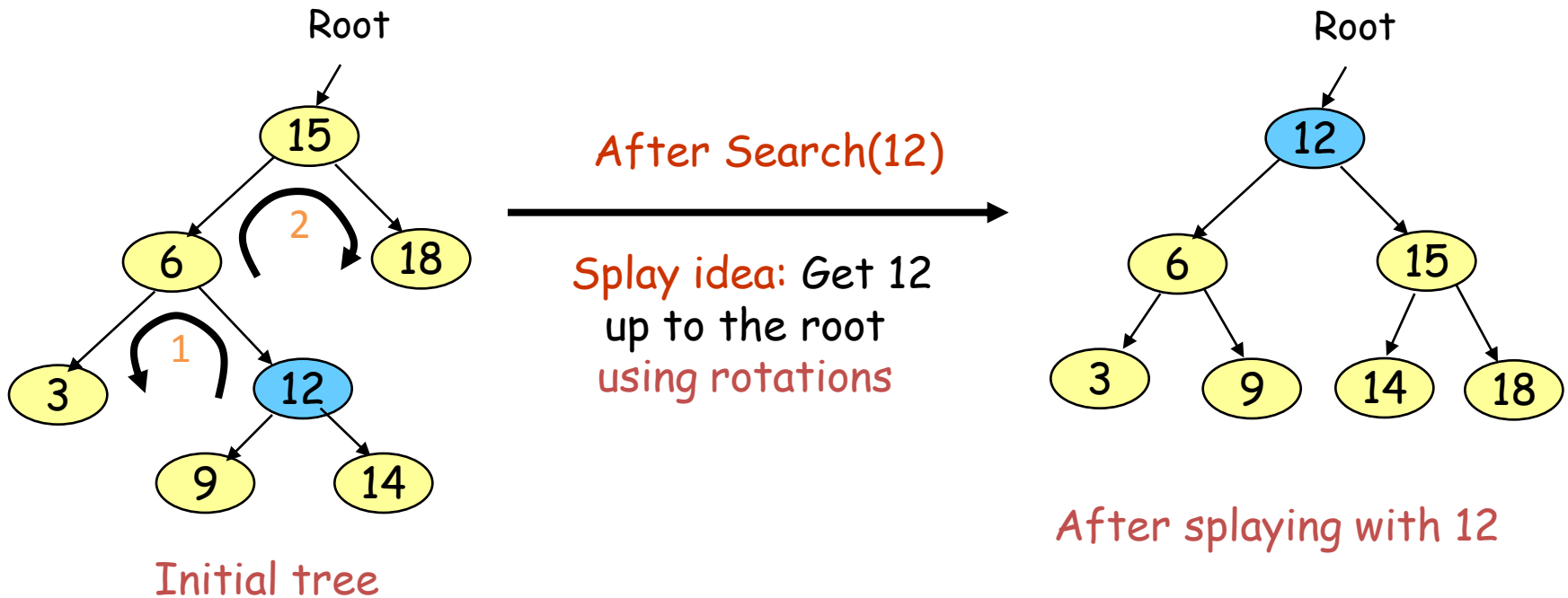
Splay trees are binary search trees (BSTs) that:

- Are not perfectly balanced all the time
- Allow **search and insertion** operations **to try to balance** the tree so that **future operations may run faster**

Based on the heuristic:

- If X is accessed once, it is likely to be accessed again.
- After node X is accessed, perform **“splaying”** operations to bring X up to the root of the tree.
- Do this in a way that leaves the tree more or less balanced as a whole.

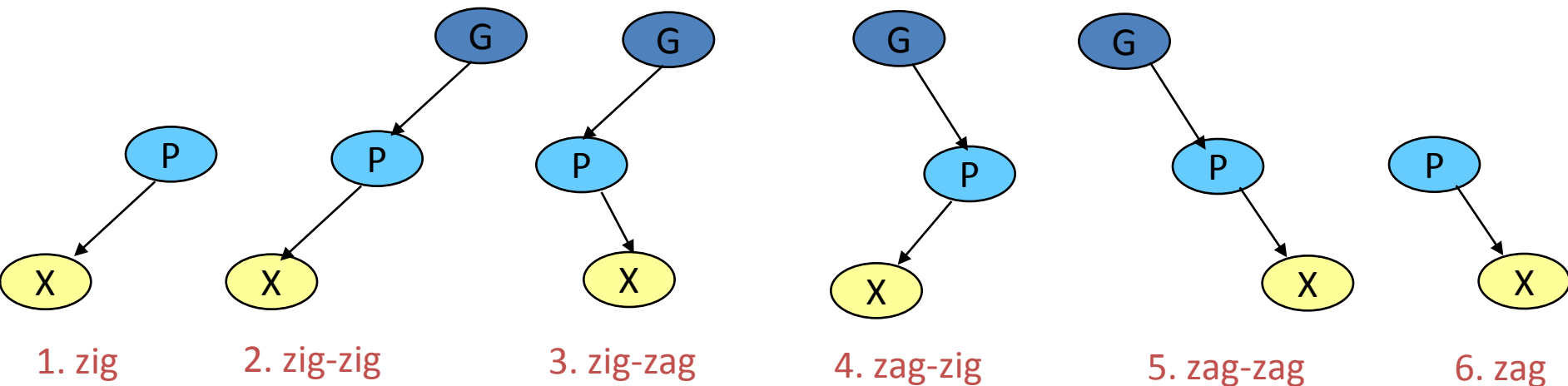
Example



- Not only splaying with 12 makes the tree **balanced**, subsequent accesses for 12 will take **$O(1)$** time.
- **Active (recently accessed)** nodes will move towards the root and **inactive** nodes will slowly move further from the root

Splay Tree Terminology

- Let X be a **non-root** node, i.e., has at least 1 ancestor.
- Let P be its **parent** node.
- Let G be its **grandparent** node (if it exists)
- Consider a path from G to X :
 - Each time we go **left**, we say that we “**zig**”
 - Each time we go **right**, we say that we “**zag**”
- There are 6 possible cases:

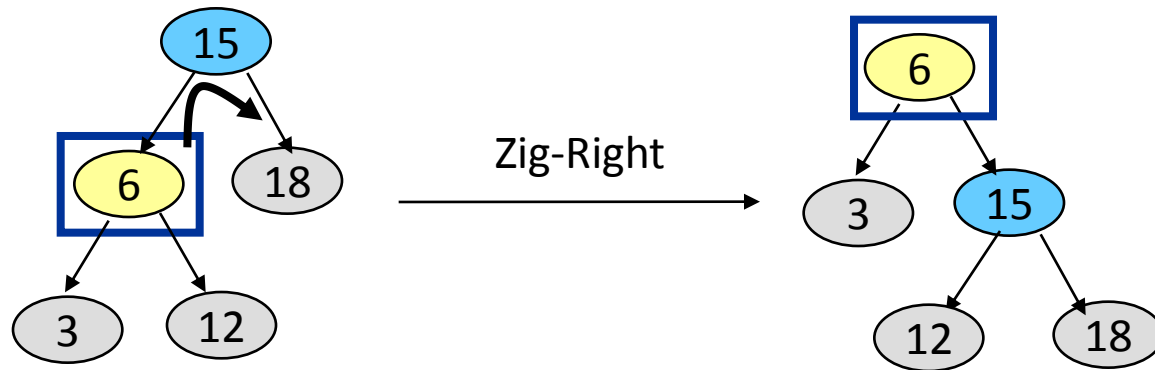


Splay Tree Operations

- When node X is accessed, apply one of **six** rotation operations:
 - **Single Rotations (X has a P but no G)**
 - zig, zag
 - **Double Rotations (X has both a P and a G)**
 - zig-zig, zig-zag
 - zag-zig, zag-zag

Splay Trees: Zig Operation

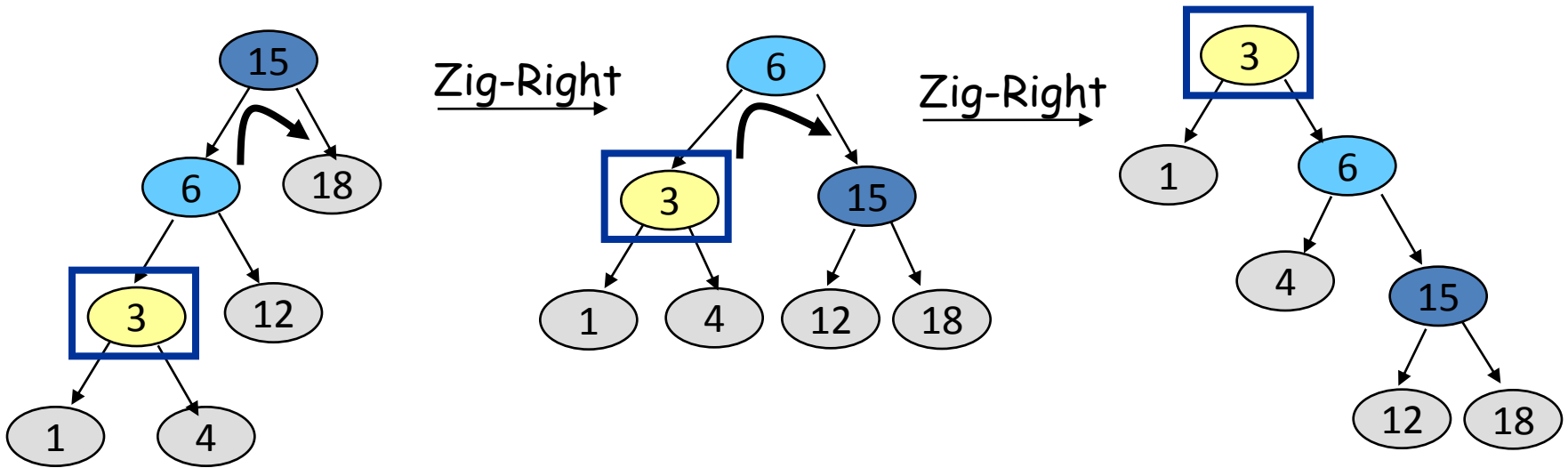
- “Zig” is just a **single rotation**, as in an AVL tree
- Suppose 6 was the node that was accessed (e.g. using Search)



- “Zig-Right” moves 6 to the root.
- Can access 6 faster next time: $O(1)$
- Notice that this is simply a **right rotation** in AVL tree terminology.

Splay Trees: Zig-Zig Operation

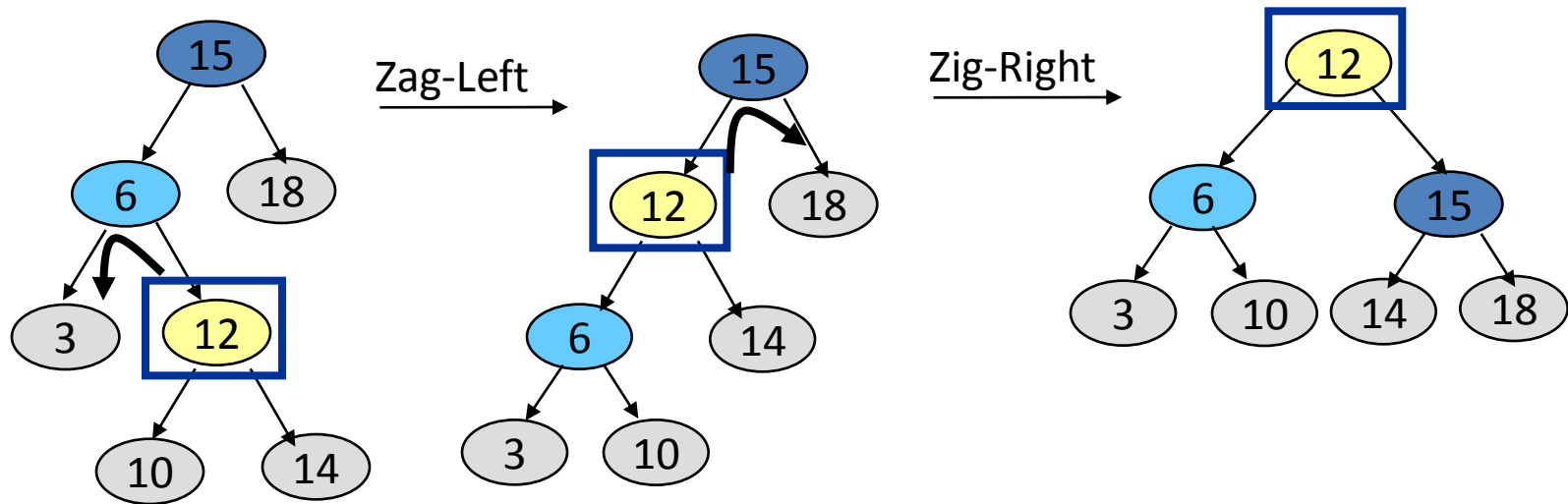
- “Zig-Zig” consists of **two single rotations of the same type**
- Suppose **3** was the node that was accessed (e.g., using Search)



- Due to “zig-zig” splaying, 3 has bubbled to the top!
- Note: Parent-Grandparent is rotated first.

Splay Trees: Zig-Zag Operation

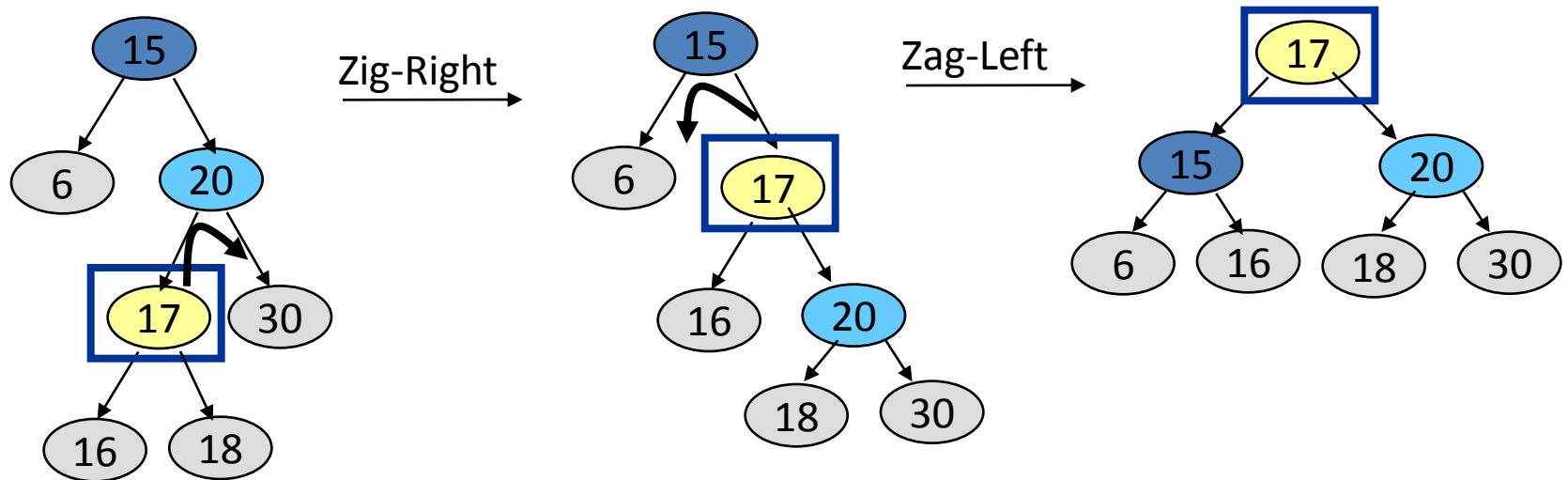
- “Zig-Zag” consists of **two rotations of the opposite type**
- Suppose **12** was the node that was accessed (e.g., using Search)



- Due to “zig-zag” splaying, 12 has bubbled to the top!
- Notice that this is simply an **LR imbalance correction** in AVL tree terminology (first a left rotation, then a right rotation)

Splay Trees: Zag-Zig Operation

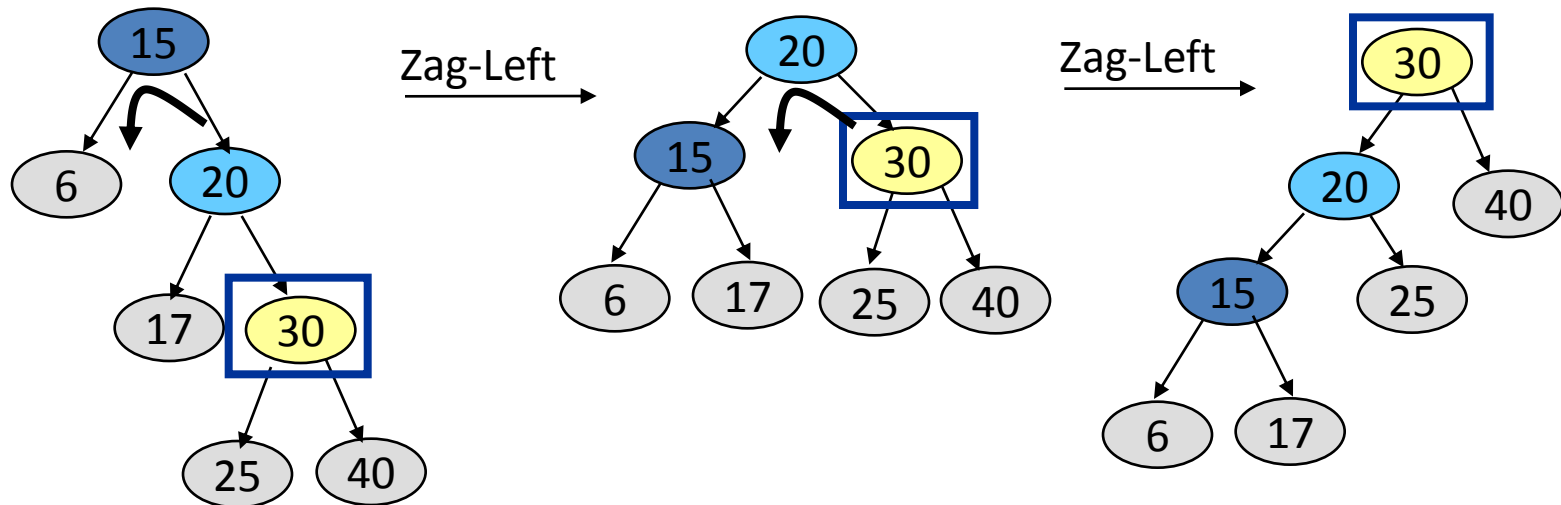
- “Zag-Zig” consists of **two rotations of the opposite type**
- Suppose **17** was the node that was accessed (e.g., using Search)



- Due to “zag-zig” splaying, 17 has bubbled to the top!
- Notice that this is simply an **RL imbalance correction** in AVL tree terminology (first a right rotation, then a left rotation)

Splay Trees: Zag-Zag Operation

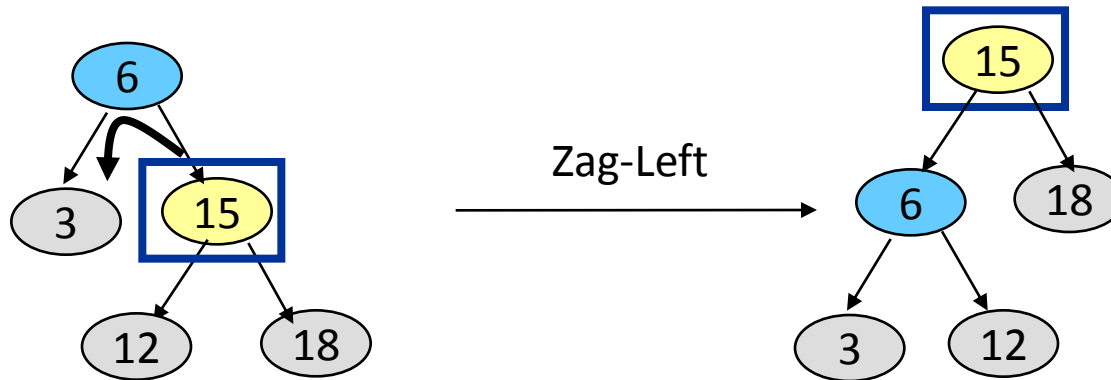
- “Zag-Zag” consists of **two single rotations of the same type**
- Suppose **30** was the node that was accessed (e.g., using Search)



- Due to “zag-zag” splaying, 30 has bubbled to the top!
- Note: **Parent-Grandparent** is rotated first.

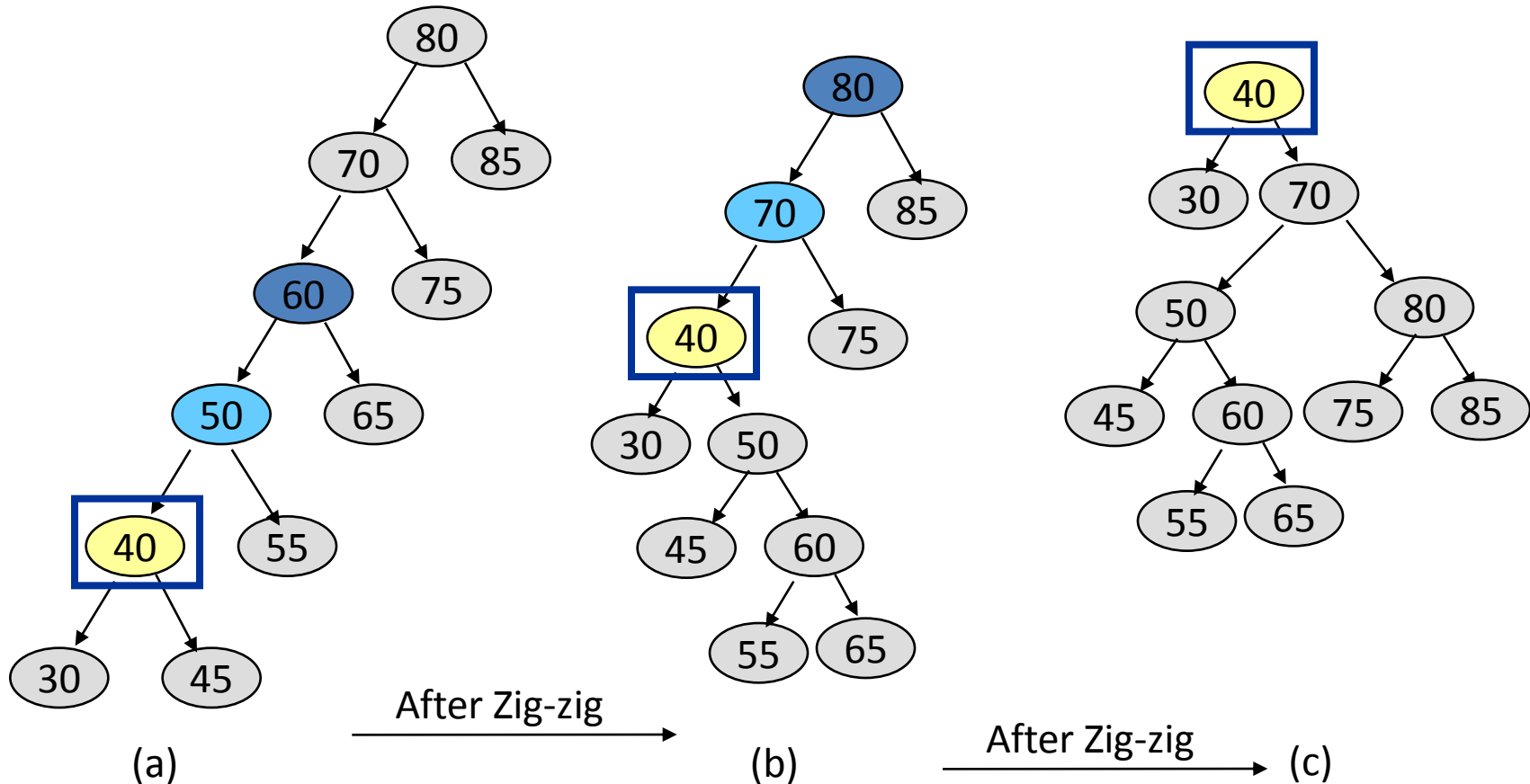
Splay Trees: Zag Operation

- “Zag” is just a **single rotation**, as in an AVL tree
- Suppose **15** was the node that was accessed (e.g., using Search)

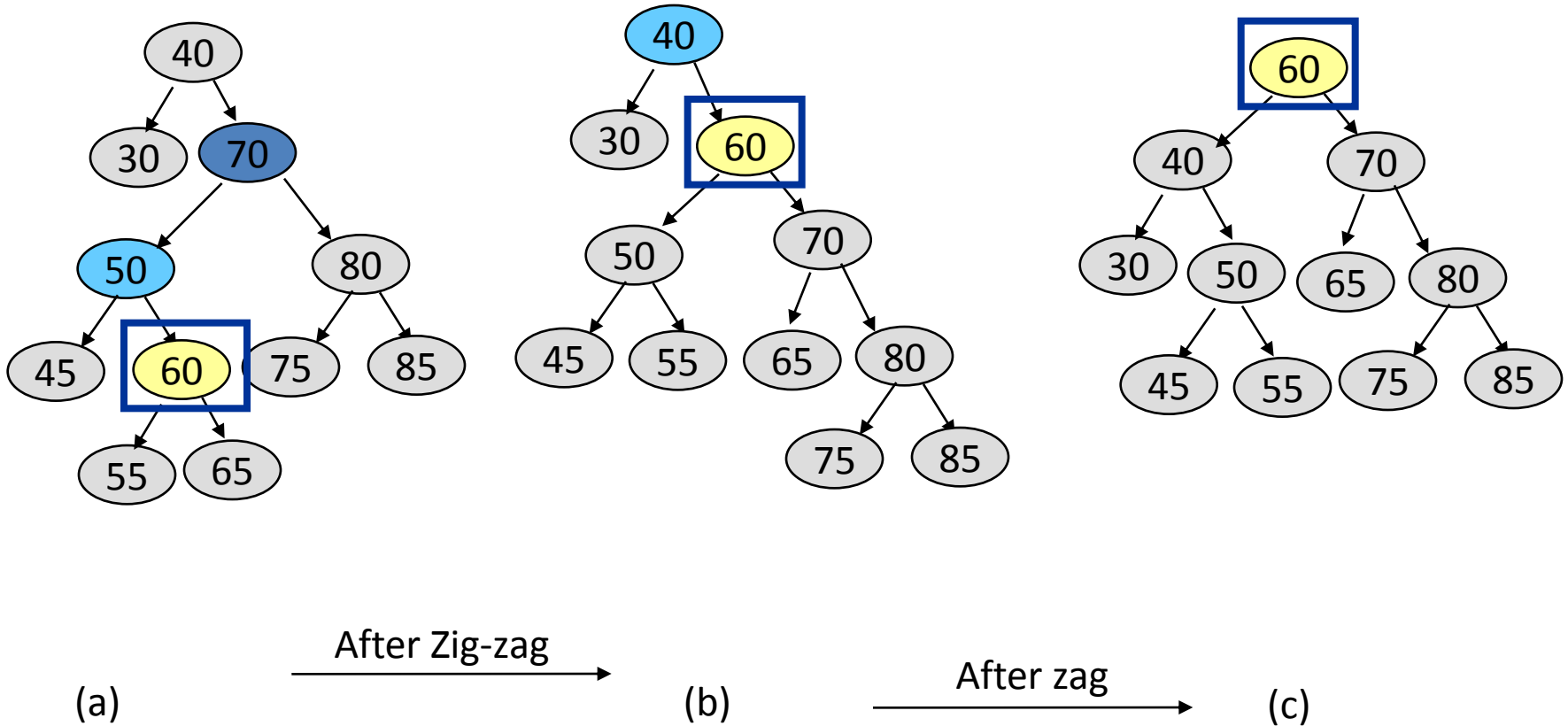


- “Zag-Left” moves 15 to the root.
- Can access 15 faster next time: $O(1)$
- Notice that this is simply a **left rotation** in AVL tree terminology

Splay Trees: Example – 40 is accessed



Splay Trees: Example – 60 is accessed



Splaying during other operations

- Splaying can be done not just after Search, but also after other operations such as Insert/Delete.
- **Insert X:** After inserting X at a leaf node (as in a regular BST), splay X up to the root
- **Delete X:** Do a Search on X and get X up to the root. Delete X at the root and move the largest item in its left sub-tree, i.e, its predecessor, to the root using splaying.
- **Note on Search X:** If X was not found, splay the leaf node that the Search ended up with to the root.

Summary of Splay Trees

- Examples suggest that splaying causes tree to get balanced.
- The actual analysis is rather advanced and is in Chapter 11. Such analysis is called “amortized analysis”
- **Result of Analysis:** Any sequence of M operations on a splay tree of size N takes $O(M \log N)$ time. So, the amortized running time for one operation is $O(\log N)$.
- This guarantees that even if the depths of some nodes get very large, you cannot get a long sequence of $O(N)$ searches because each search operation causes a rebalance.
- Without splaying, total time could be $O(MN)$.

Comparison of Search Trees

Tree	Worst Case			Expected		
	Search	Insert	Remove	Search	Insert	Remove
BST	n	n	n	log n	log n	log n
AVL tree	log n	log n	log n	log n	log n	log n
red-black tree	log n	log n	log n	log n	log n	log n
splay tree	n	n	n	log n	log n	log n
B-trees	log n	log n	log n	log n	log n	log n

Knuth-Morris-Pratt Algorithm

The problem of String Matching

Given a string 'S', the problem of string matching deals with finding whether a pattern 'p' occurs in 'S' and if 'p' does occur then returning position in 'S' where 'p' occurs.

.... a $O(mn)$ approach

One of the most obvious approach towards the string matching problem would be to compare the first element of the pattern to be searched 'p', with the first element of the string 'S' in which to locate 'p'. If the first element of 'p' matches the first element of 'S', compare the second element of 'p' with second element of 'S'. If match found proceed likewise until entire 'p' is found. If a mismatch is found at any position, shift 'p' one position to the right and repeat comparison beginning from first element of 'p'.

How does the $O(mn)$ approach work

Below is an illustration of how the previously described $O(mn)$ approach works.

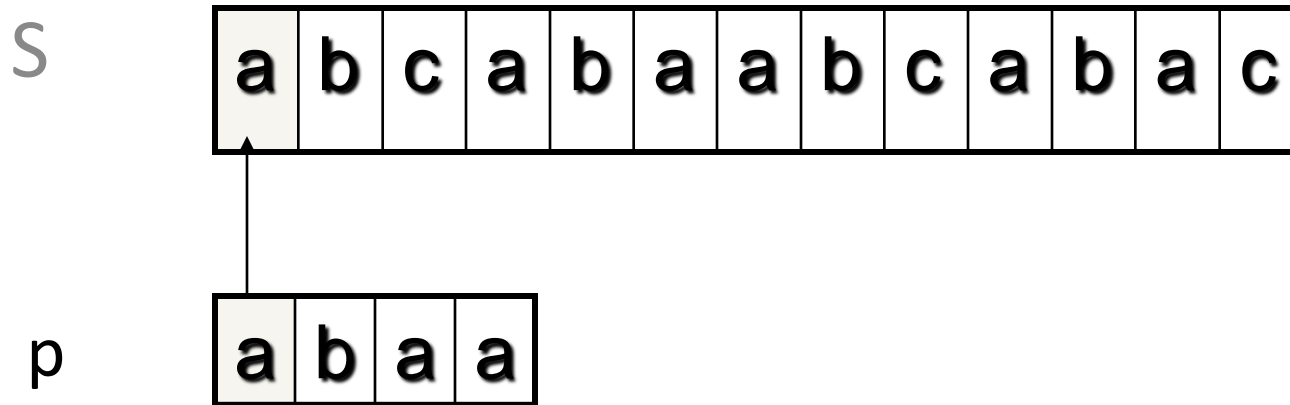
String S

a	b	c	a	b	a	a	b	c	a	b	a	c
---	---	---	---	---	---	---	---	---	---	---	---	---

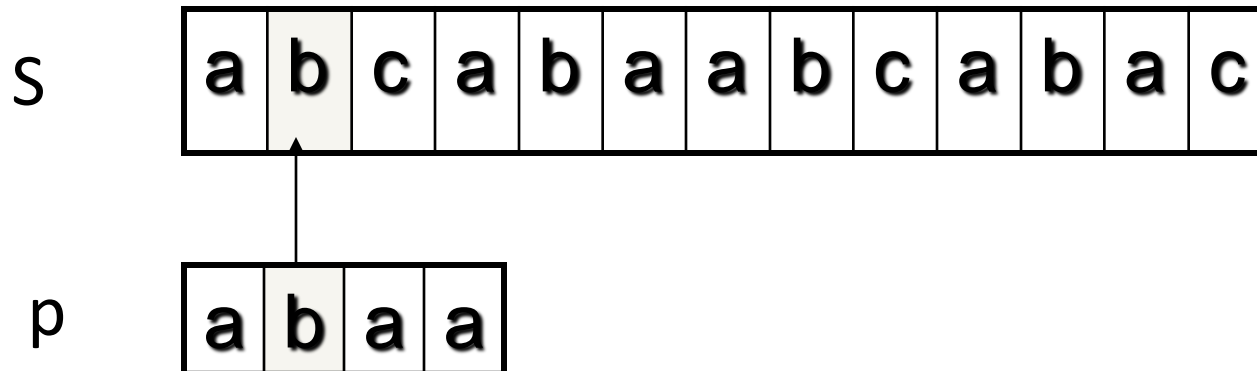
Pattern p

a	b	a	a
---	---	---	---

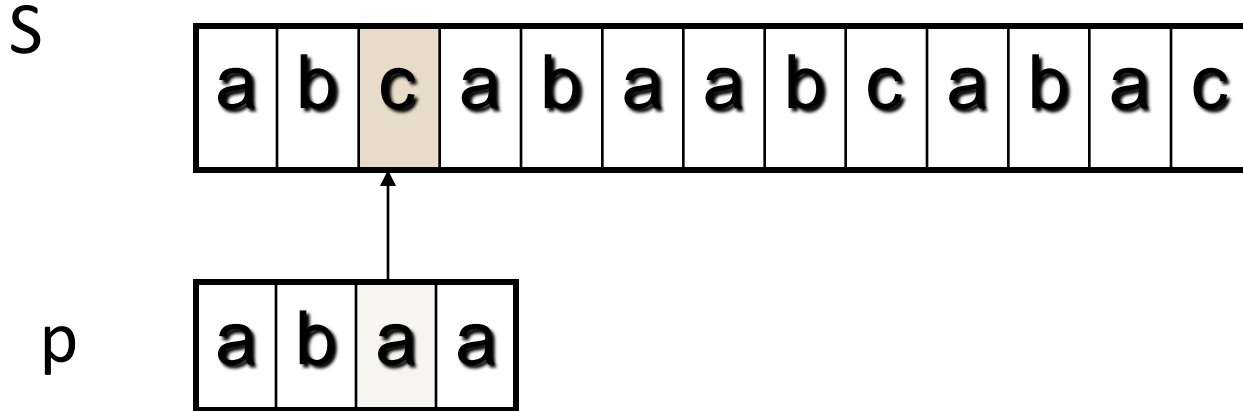
Step 1: compare $p[1]$ with $S[1]$



Step 2: compare $p[2]$ with $S[2]$

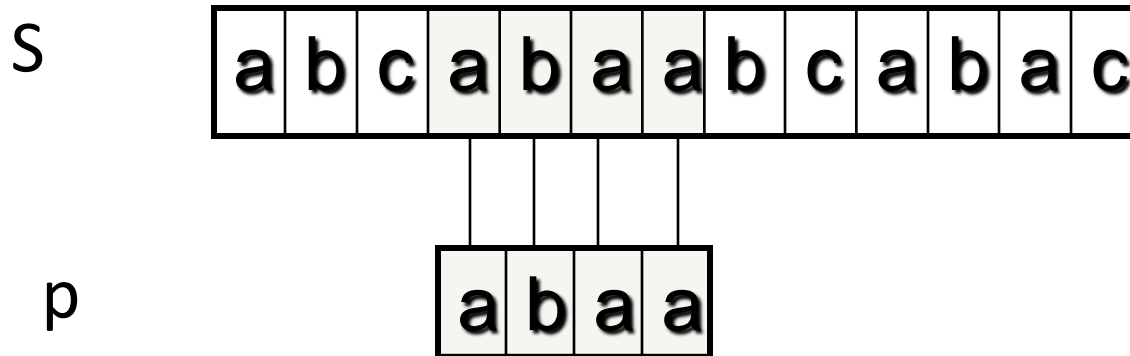


Step 3: compare $p[3]$ with $S[3]$



Mismatch occurs here..

Since mismatch is detected, shift 'p' one position to the right and repeat matching procedure.



Finally, a match would be found after shifting 'p' three times to the right side.

Drawbacks of this approach: if 'm' is the length of pattern 'p' and 'n' the length of string 'S', the matching time is of the order $O(mn)$. This is a certainly a very slow running algorithm.

What makes this approach so slow is the fact that elements of 'S' with which comparisons had been performed earlier are involved again and again in comparisons in some future iterations. For example: when mismatch is detected for the first time in comparison of $p[3]$ with $S[3]$, pattern 'p' would be moved one position to the right and matching procedure would resume from here. Here the first comparison that would take place would be between $p[0]='a'$ and $S[1]='b'$. It should be noted here that $S[1]='b'$ had been previously involved in a comparison in step 2. this is a repetitive use of $S[1]$ in another comparison.

It is these repetitive comparisons that lead to the runtime of $O(mn)$.

The Knuth-Morris-Pratt Algorithm

Knuth, Morris and Pratt proposed a linear time algorithm for the string matching problem.

A matching time of $O(n)$ is achieved by avoiding comparisons with elements of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs

Components of KMP algorithm

- The prefix function, Π

The prefix function, Π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. This information can be used to avoid useless shifts of the pattern 'p'. In other words, this enables avoiding backtracking on the string 'S'.

- The KMP Matcher

With string 'S', pattern 'p' and prefix function ' Π ' as inputs, finds the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrence is found.

The prefix function, Π

Following pseudocode computes the prefix function, Π :

Compute-Prefix-Function (p)

```
1  $m \leftarrow \text{length}[p]$            //'p' pattern to be matched
2  $\Pi[1] \leftarrow 0$ 
3  $k \leftarrow 0$ 
4   for  $q \leftarrow 2$  to  $m$ 
5       do while  $k > 0$  and  $p[k+1] \neq p[q]$ 
6           do  $k \leftarrow \Pi[k]$ 
7           If  $p[k+1] = p[q]$ 
8               then  $k \leftarrow k + 1$ 
9            $\Pi[q] \leftarrow k$ 
10  return  $\Pi$ 
```

Example: compute Π for the pattern 'p' below:

p

a	b	a	b	a	c	a
---	---	---	---	---	---	---

Initially: $m = \text{length}[p] = 7$

$\Pi[1] = 0$

$k = 0$

Step 1: $q = 2, k = 0$

$\Pi[2] = 0$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0					

Step 2: $q = 3, k = 0$,

$\Pi[3] = 1$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1				

Step 3: $q = 4, k = 1$

$\Pi[4] = 2$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
Π	0	0	1	2			

Step 4: $q = 5, k = 2$
 $\Pi[5] = 3$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3		

Step 5: $q = 6, k = 3$
 $\Pi[6] = 1$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	1	

Step 6: $q = 7, k = 1$
 $\Pi[7] = 1$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	1	1

After iterating 6 times, the prefix
 function computation is
 complete: \rightarrow

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
Π	0	0	1	2	3	1	1

The KMP Matcher

The KMP Matcher, with pattern 'p', string 'S' and prefix function ' Π ' as input, finds a match of p in S.

Following pseudocode computes the matching component of KMP algorithm:

KMP-Matcher(S,p)

```
1  $n \leftarrow \text{length}[S]$ 
2  $m \leftarrow \text{length}[p]$ 
3  $\Pi \leftarrow \text{Compute-Prefix-Function}(p)$ 
4  $q \leftarrow 0$  //number of characters matched
5 for  $i \leftarrow 1$  to  $n$  //scan S from left to right
6   do while  $q > 0$  and  $p[q+1] \neq S[i]$ 
7     do  $q \leftarrow \Pi[q]$  //next character does not match
8   if  $p[q+1] = S[i]$ 
9     then  $q \leftarrow q + 1$  //next character matches
10  if  $q = m$  //is all of p matched?
11    then print "Pattern occurs with shift"  $i - m$ 
12     $q \leftarrow \Pi[q]$  // look for the next match
```

Note: KMP finds every occurrence of a 'p' in 'S'. That is why KMP does not terminate in step 12, rather it searches remainder of 'S' for any more occurrences of 'p'.

Illustration: given a String 'S' and pattern 'p' as follows:

S

b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

p

a	b	a	b	a	c	a
---	---	---	---	---	---	---

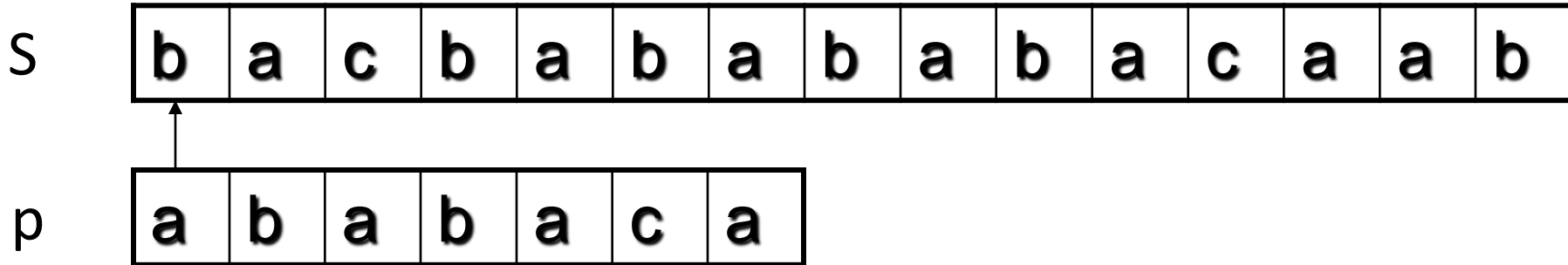
Let us execute the KMP algorithm to find whether 'p' occurs in 'S'.

For 'p' the prefix function, Π was computed previously and is as follows:

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	1	1

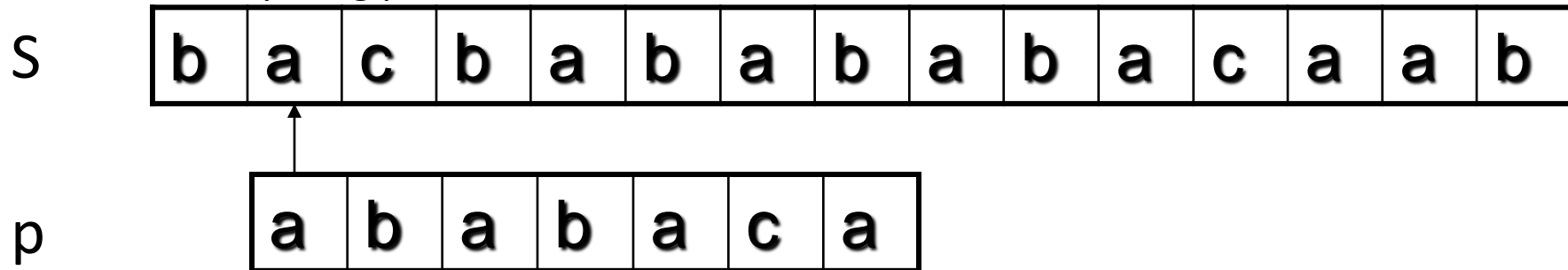
Initially: $n = \text{size of } S = 15;$
 $m = \text{size of } p = 7$

Step 1: $i = 1, q = 0$
comparing $p[1]$ with $S[1]$



$P[1]$ does not match with $S[1]$. 'p' will be shifted one position to the right.

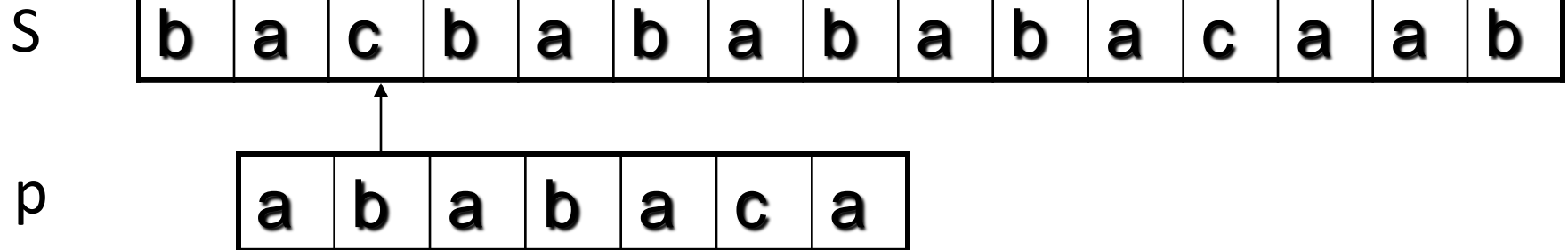
Step 2: $i = 2, q = 0$
comparing $p[1]$ with $S[2]$



$P[1]$ matches $S[2]$. Since there is a match, p is not shifted.

Step 3: $i = 3, q = 1$

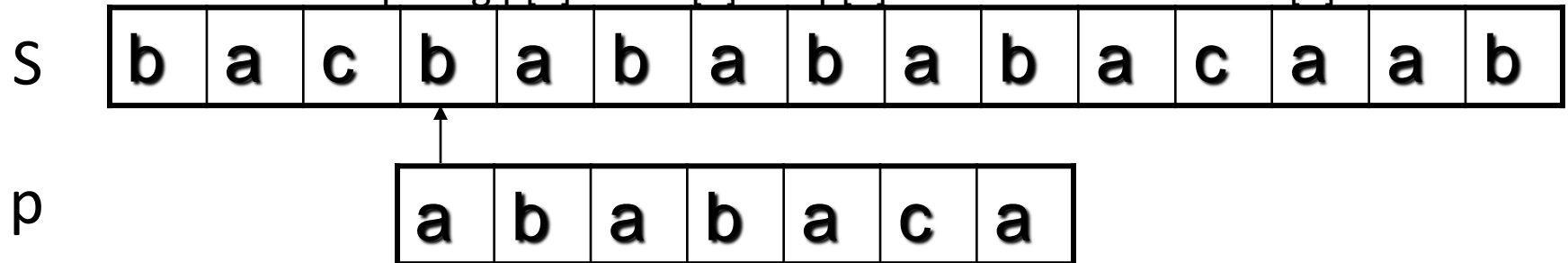
Comparing $p[2]$ with $S[3]$ $p[2]$ does not match with $S[3]$



Backtracking on p, comparing $p[1]$ and $S[3]$

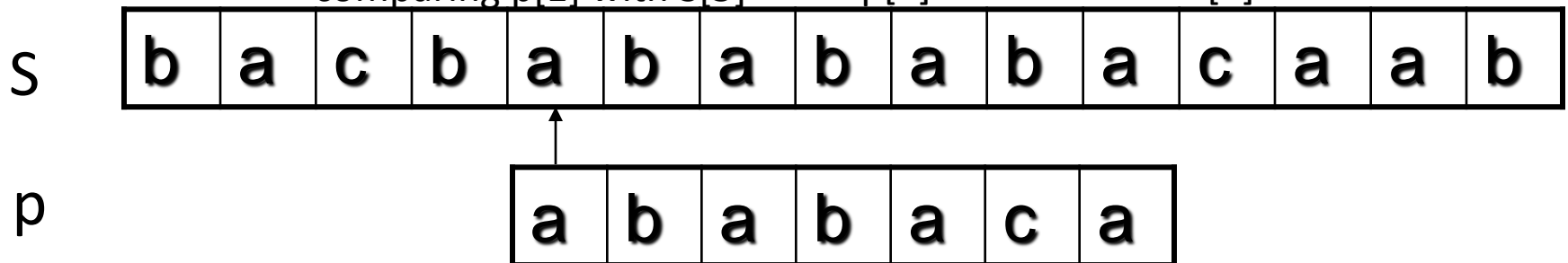
Step 4: $i = 4, q = 0$

comparing $p[1]$ with $S[4]$ $p[1]$ does not match with $S[4]$

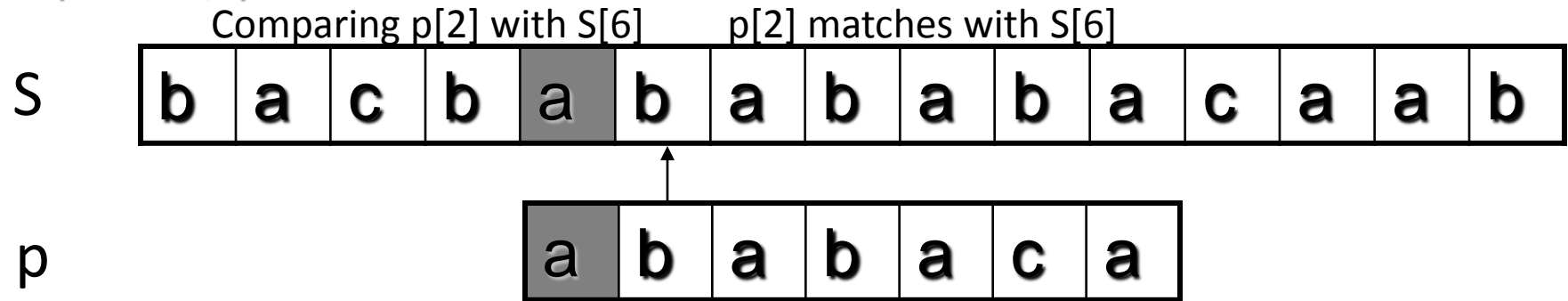


Step 5: $i = 5, q = 0$

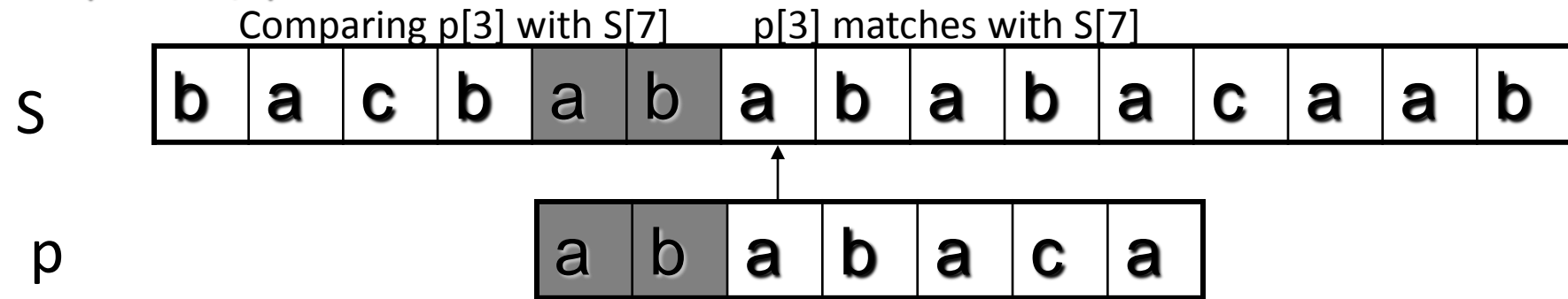
comparing $p[1]$ with $S[5]$ $p[1]$ matches with $S[5]$



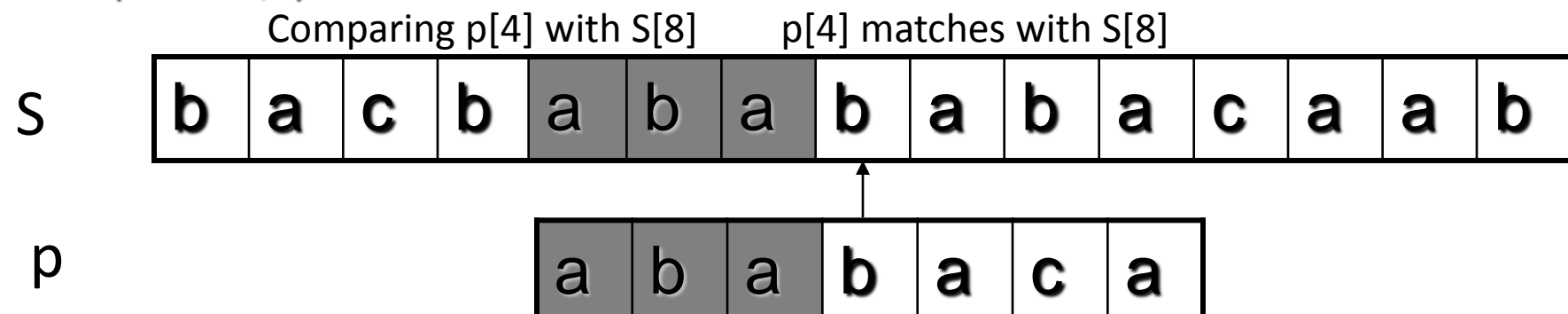
Step 6: $i = 6, q = 1$



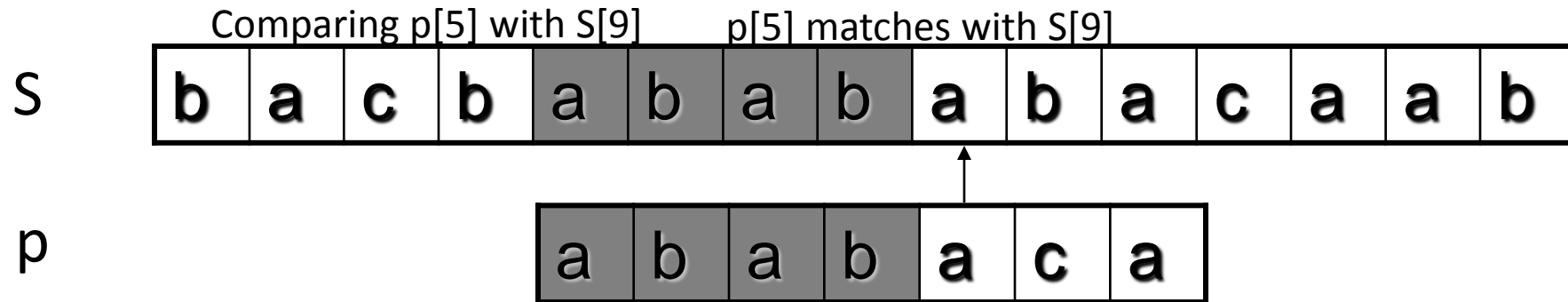
Step 7: $i = 7, q = 2$



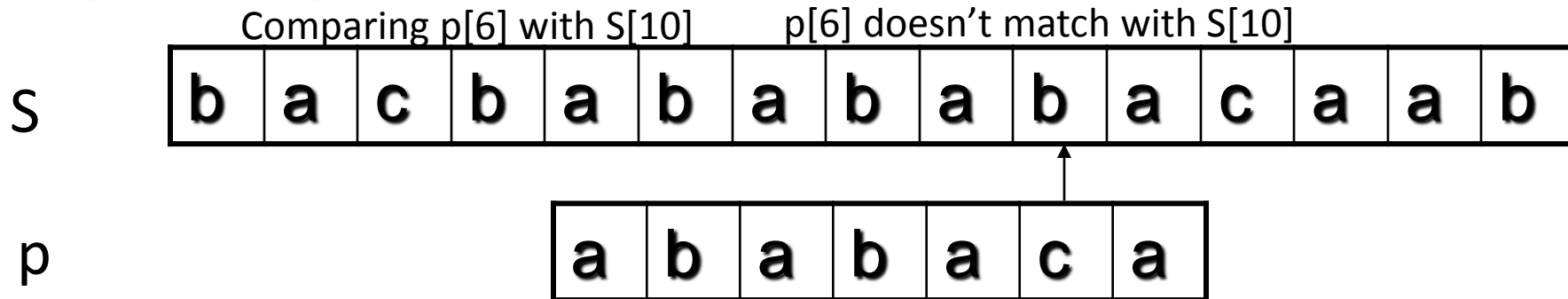
Step 8: $i = 8, q = 3$



Step 9: $i = 9, q = 4$

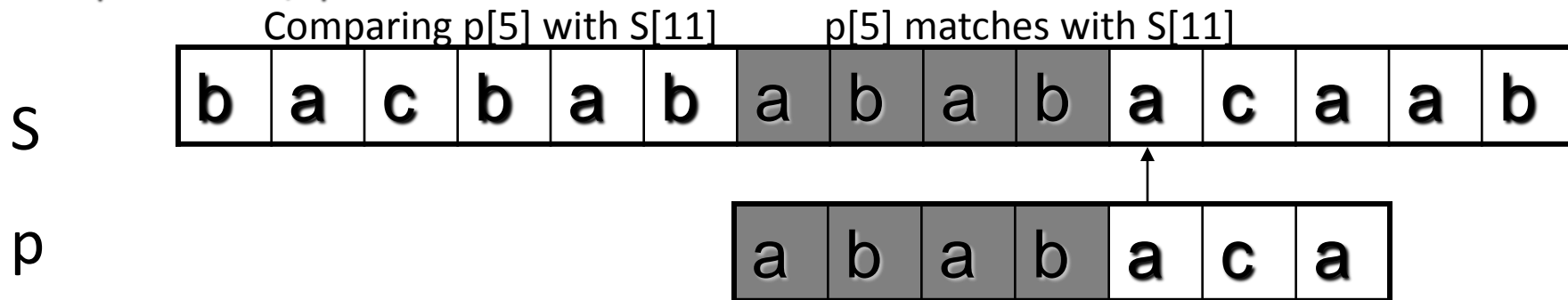


Step 10: $i = 10, q = 5$

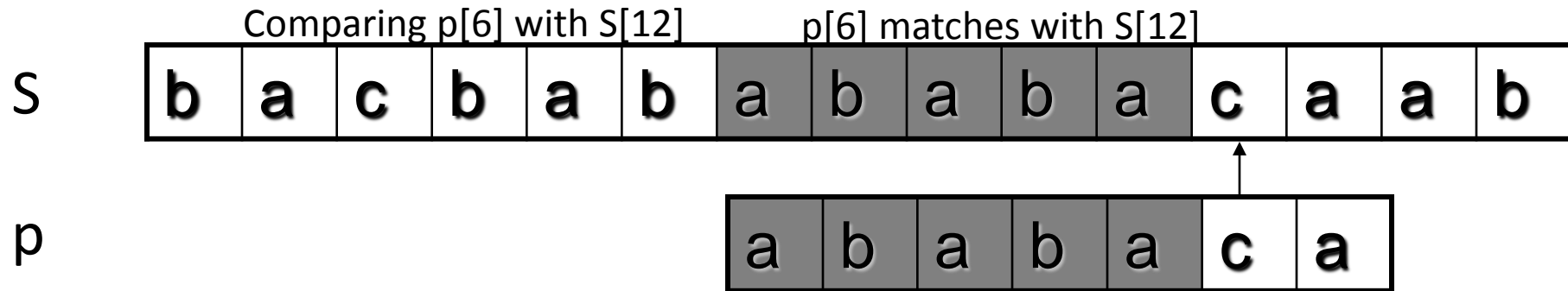


Backtracking on p, comparing $p[4]$ with $S[10]$ because after mismatch $q = \Pi[5] = 3$

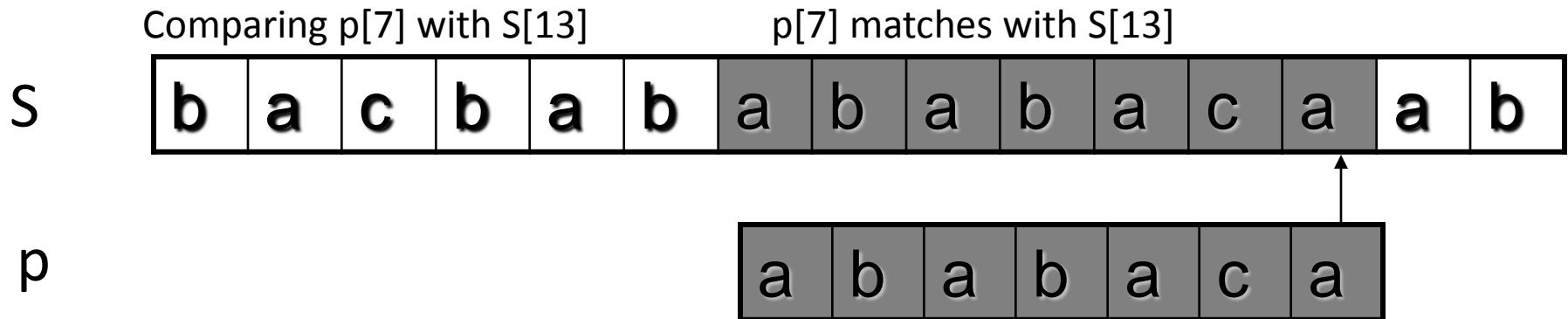
Step 11: $i = 11, q = 4$



Step 12: $i = 12, q = 5$



Step 13: $i = 13, q = 6$



Pattern 'p' has been found to completely occur in string 'S'. The total number of shifts that took place for the match to be found are: $i - m = 13 - 7 = 6$ shifts.

Running - time analysis

- Compute-Prefix-Function (Π)

```
1 m  $\leftarrow$  length[p]      //'p' pattern to be
   matched
2  $\Pi[1] \leftarrow 0$ 
3 k  $\leftarrow 0$ 
4   for q  $\leftarrow 2$  to m
5       do while k > 0 and p[k+1] != p[q]
6           do k  $\leftarrow \Pi[k]$ 
7           if p[k+1] = p[q]
8               then k  $\leftarrow$  k + 1
9            $\Pi[q] \leftarrow k$ 
10  return  $\Pi$ 
```

In the above pseudocode for computing the prefix function, the for loop from step 4 to step 10 runs 'm' times. Step 1 to step 3 take constant time. Hence the running time of compute prefix function is $\Theta(m)$.

- KMP Matcher

```
1 n  $\leftarrow$  length[S]
2 m  $\leftarrow$  length[p]
3  $\Pi \leftarrow$  Compute-Prefix-Function(p)
4 q  $\leftarrow 0$ 
5 for i  $\leftarrow 1$  to n
6   do while q > 0 and p[q+1] != S[i]
7       do q  $\leftarrow \Pi[q]$ 
8   if p[q+1] = S[i]
9       then q  $\leftarrow$  q + 1
10  if q = m
11      then print "Pattern occurs with shift" i
        - m
12      q  $\leftarrow \Pi[q]$ 
```

The for loop beginning in step 5 runs 'n' times, i.e., as long as the length of the string 'S'. Since step 1 to step 4 take constant time, the running time is dominated by this for loop. Thus running time of matching function is $\Theta(n)$.

Tries

- ***Trie*** is a special structure to represent sets of character strings.
- Can also be used to represent data types that are objects of any type e.g. strings of integers.
- The word “***trie***” is derived from the middle letters of the word “***retrieval***”.

Tries: Example

One way to implement a spelling checker is

- Read a text file.
- Break it into words(character strings separated by blanks and new lines).
- Find those words not in a standard dictionary of words.
- Words in the text but not in the dictionary are printed out as possible misspellings.

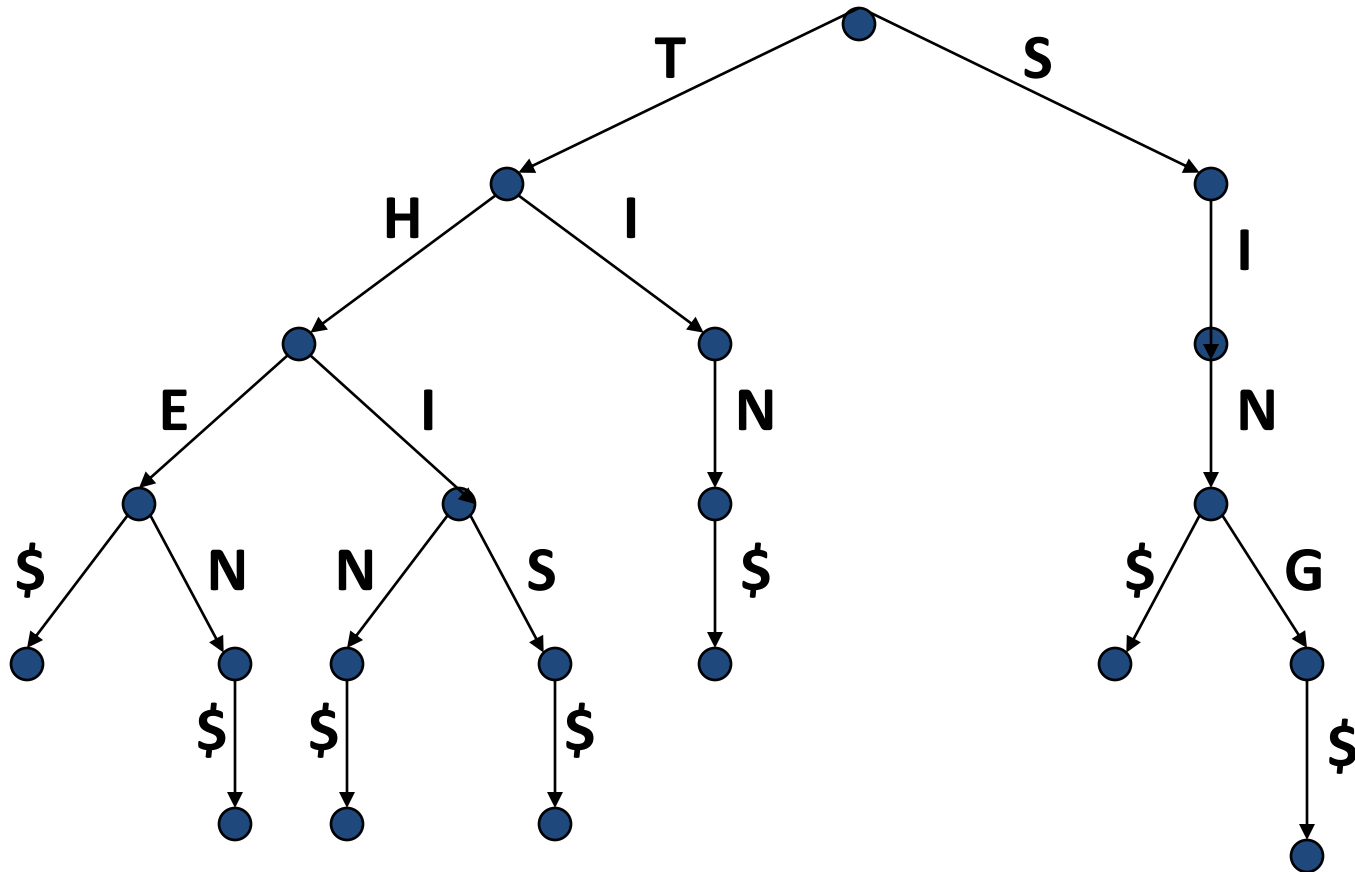
Tries: Example

It can be implemented by a set having operations of :

- INSERT
- DELETE
- MAKENULL
- PRINT

A Trie structure supports these set operations when the element of the set are words.

Tries: Example



Tries: Example

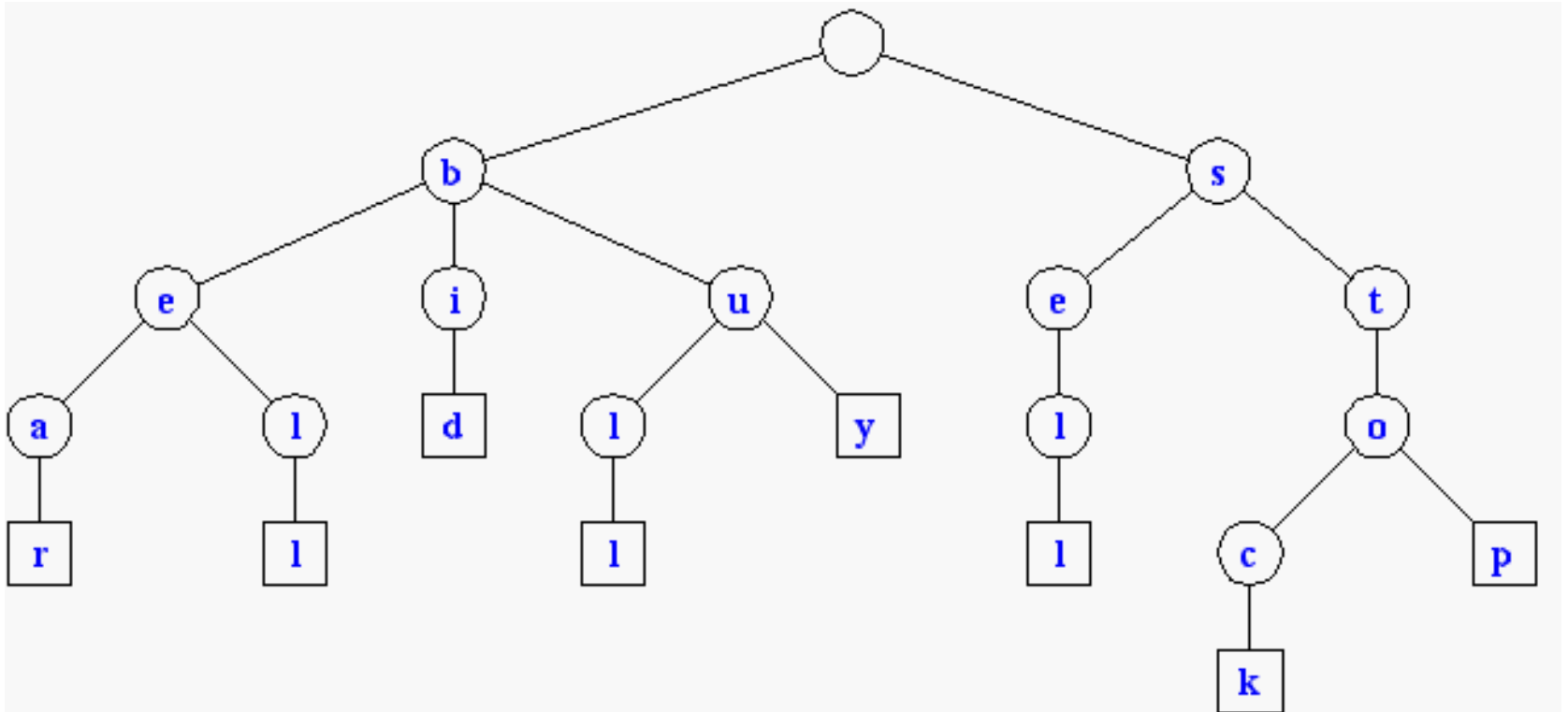
- Tries are appropriate when many words begin with the same sequence of letters.
- i.e; when the number of distinct prefixes among all words in the set is much less than the total length of all the words.
- Each path from the root to the leaf corresponds to one word in the represented set.
- Nodes of the trie correspond to the prefixes of words in the set.

Tries: Example

- The symbol \$ is added at the end of each word so that no prefix of a word can be a word itself.
- The Trie corresponds to the set {THE, THEN THIN, TIN, SIN, SING}
- Each node has at most 27 children, one for each letter and \$
- Most nodes will have many fewer than 27 children.
- A leaf reached by an edge labeled \$ cannot have any children.

Tries

- Standard Tries
- Compressed Tries
- Suffix Tries



Text Processing

- We have seen that preprocessing the pattern speeds up pattern matching queries
- After preprocessing the pattern in time proportional to the pattern length, the Boyer-Moore algorithm searches an arbitrary English text in (average) time **proportional to the text length**
- If the text is large, immutable and searched for often (e.g., works by Shakespeare), we may want to preprocess the text instead of the pattern in order to perform pattern matching queries in time **proportional to the pattern length**.
- Tradeoffs in text searching

	Preprocess Pattern	Preprocess Text	Space	Search Time
Brute Force			$O(1)$	$O(mn)$
Boyer Moore	$O(m+d)$		$O(d)$	$O(n)$ *
Suffix Trie		$O(n)$	$O(n)$	$O(m)$

n = text size

m = pattern size

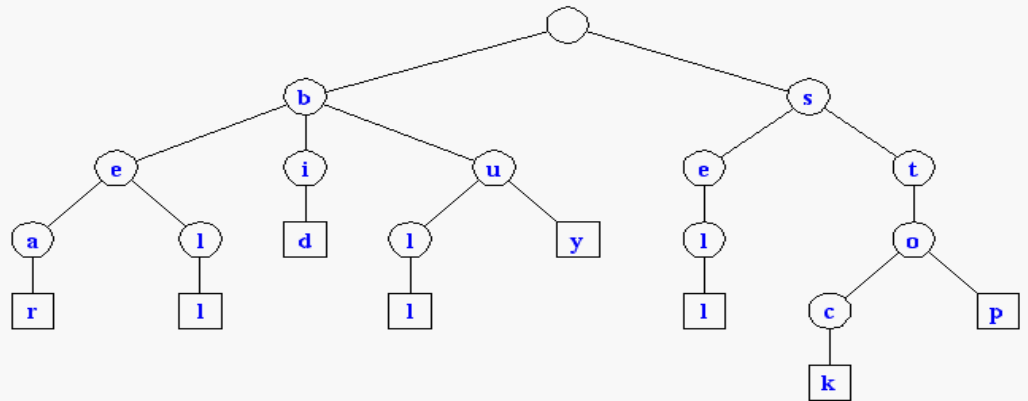
* on average

Standard Tries

- The *standard trie* for a set of strings S is an ordered tree such that:
 - each node but the root is labeled with a character
 - the children of a node are alphabetically ordered
 - the paths from the external nodes to the root yield the strings of S

- Example: standard trie for the set of strings

$S = \{ \text{bear, bell, bid, bull, buy, sell, stock, stop} \}$

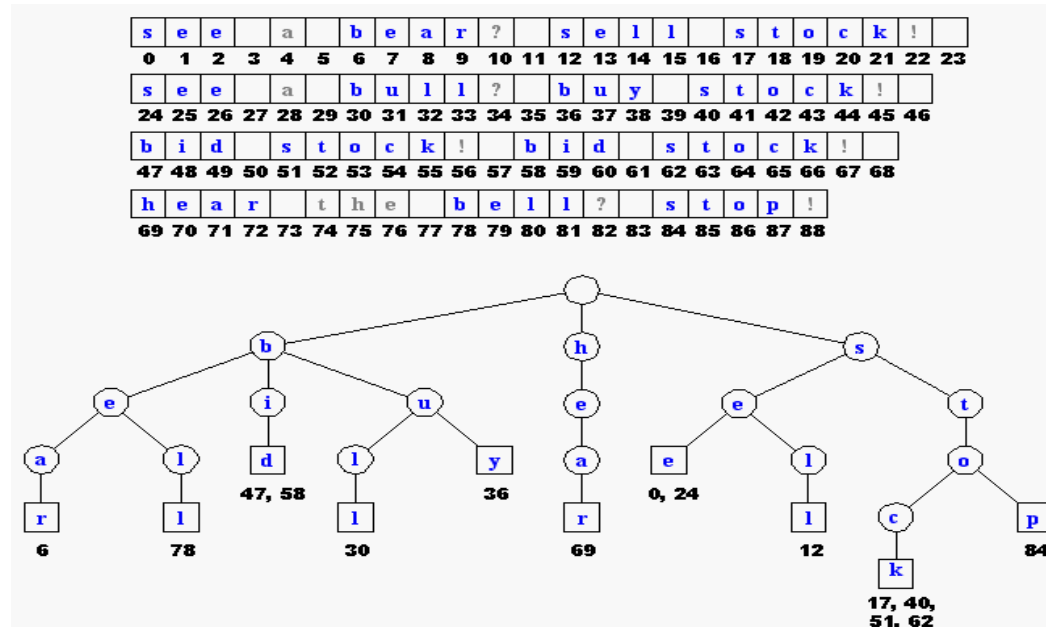


- A standard trie uses $O(n)$ space. Operations (*find, insert, remove*) take time $O(dm)$ each, where:

- n = total size of the strings in S ,
- m = size of the string parameter of the operation
- d = alphabet size,

Applications of Tries

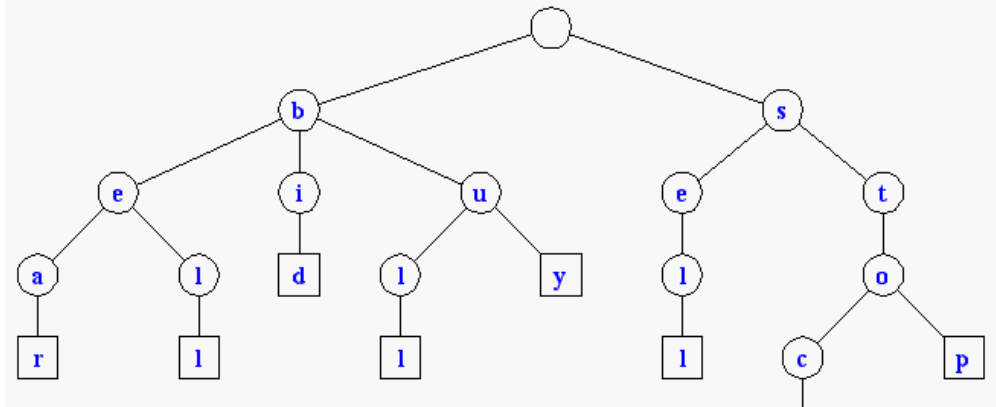
- A standard trie supports the following operations on a preprocessed text in time $O(m)$, where $m = |X|$
 - word matching**: find the first occurrence of word X in the text
 - prefix matching**: find the first occurrence of the longest prefix of word X in the text
- Each operation is performed by tracing a path in the trie starting at the root



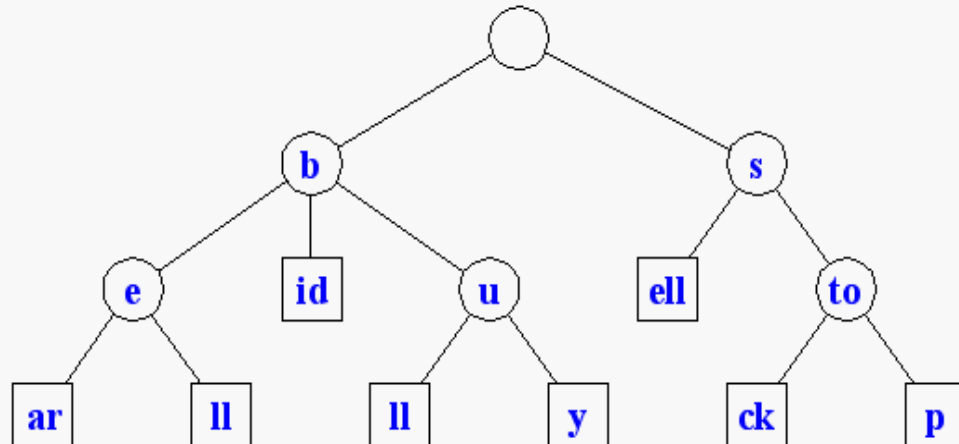
Compressed Tries

- Trie with nodes of degree at least 2
- Obtained from standard trie by compressing chains of *redundant nodes*

Standard Trie:

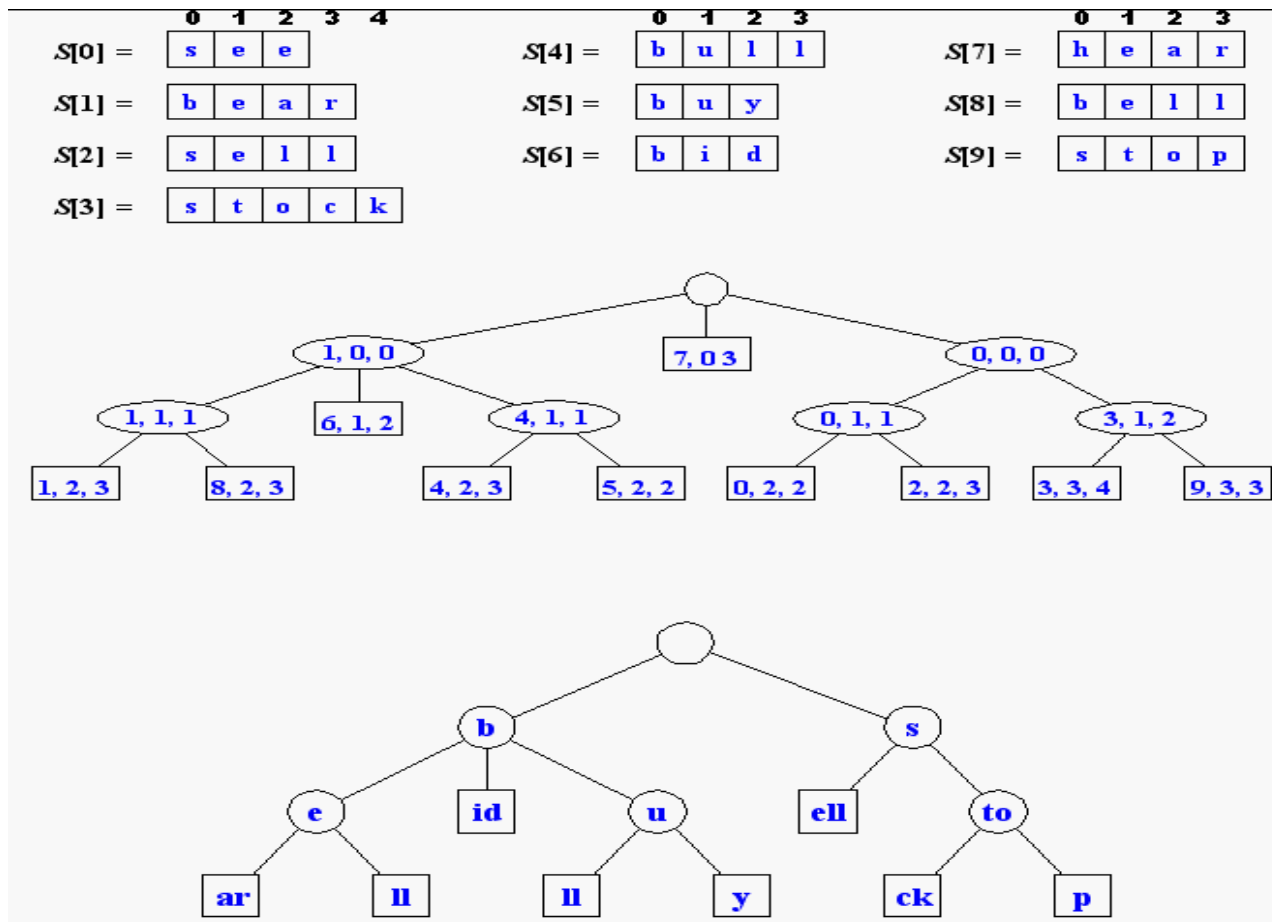


Compressed Trie:

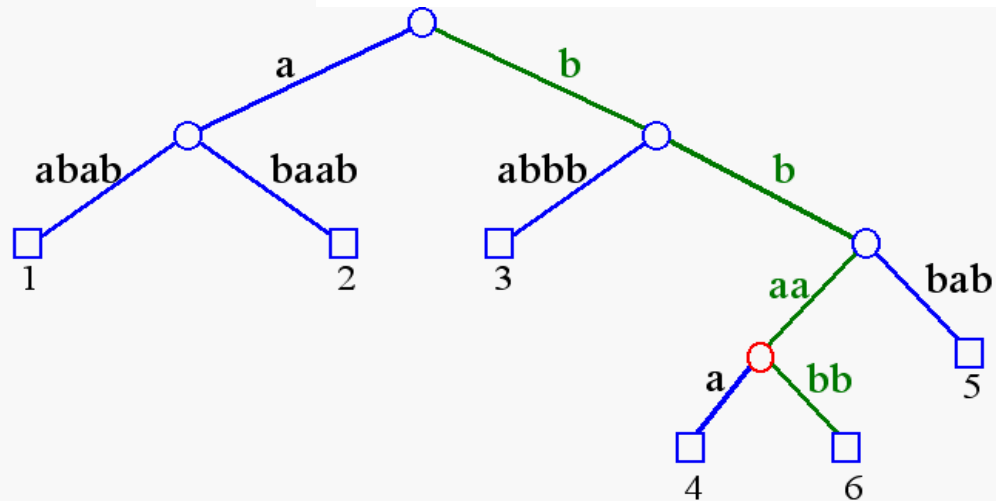
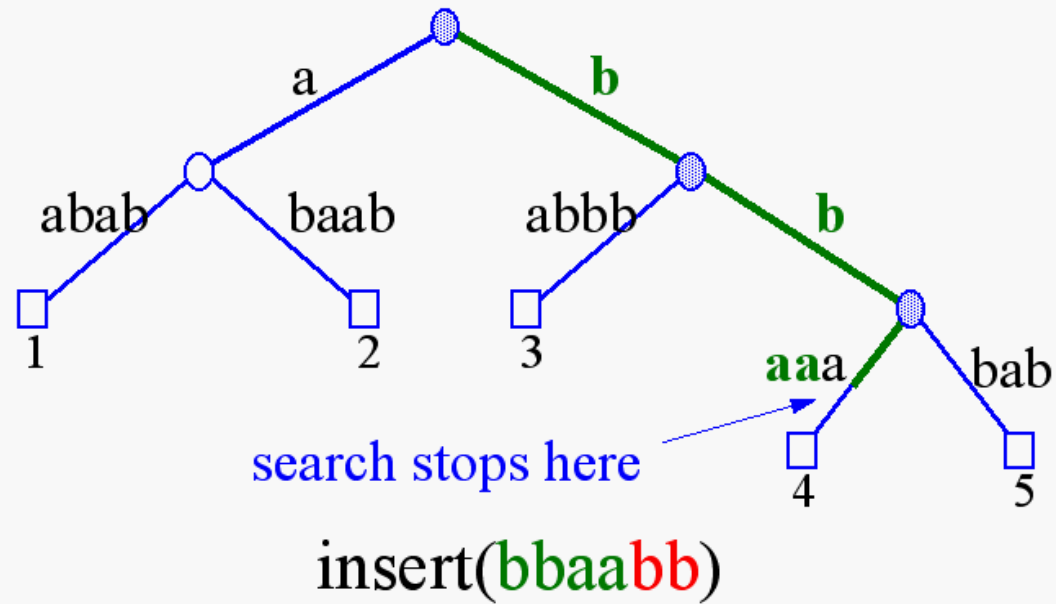


Compact Storage of Compressed Tries

- A compressed trie can be stored in space $O(s)$, where $s = |S|$, by using $O(1)$ space *index ranges* at the nodes



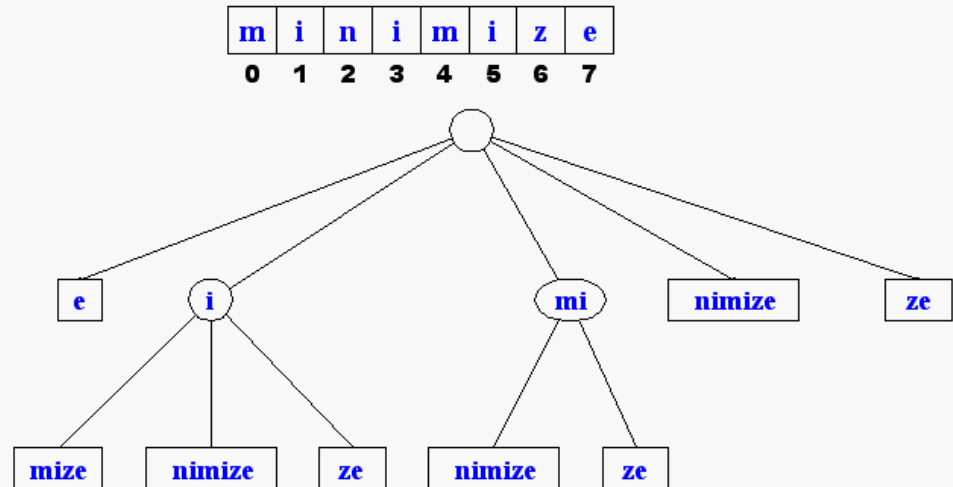
Insertion and Deletion into/from a Compressed Trie



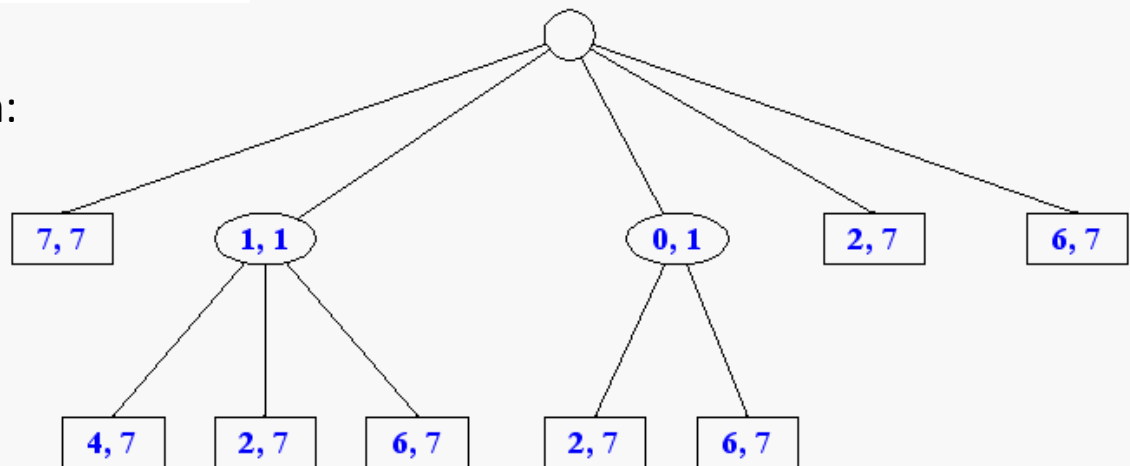
Suffix Tries

- A *suffix trie* is a compressed trie for all the suffixes of a text

Example:

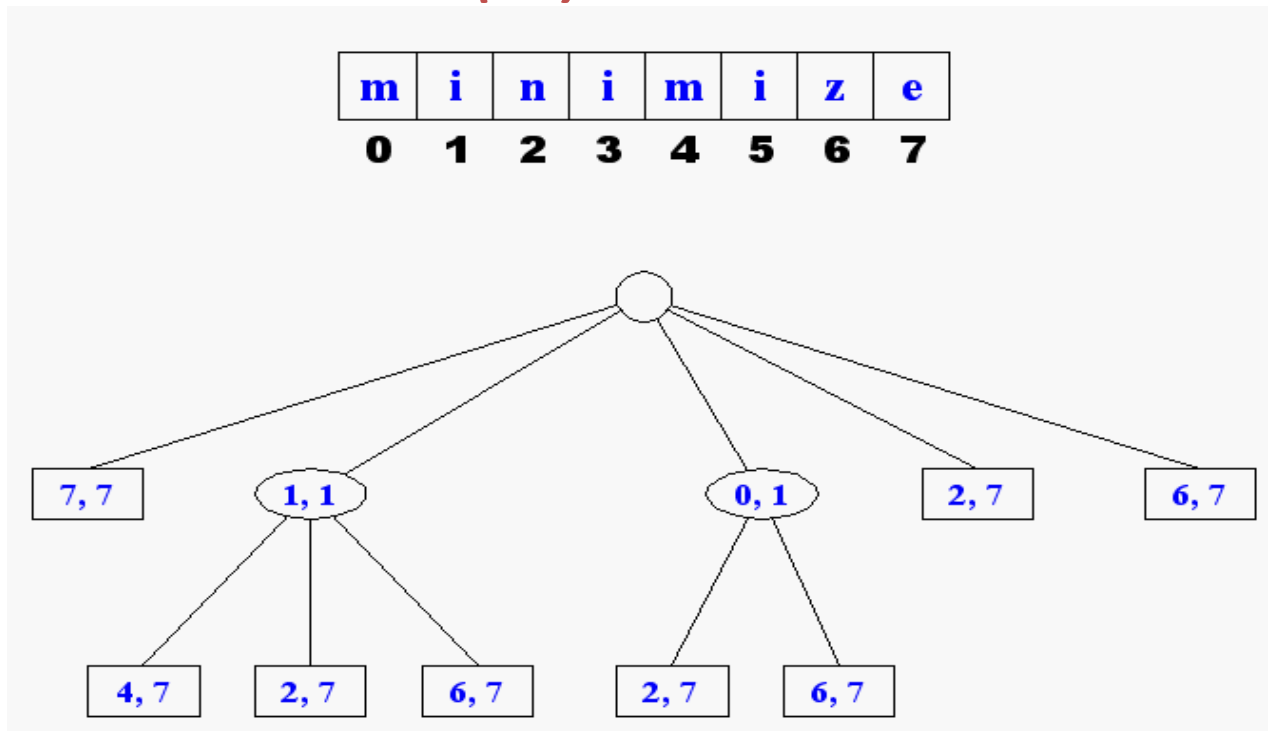


Compact representation:



Properties of Suffix Tries

- The *suffix trie* for a text X of size n from an alphabet of size d
 - stores all the $n(n-1)/2$ *suffixes* of X in $O(n)$ space
 - supports arbitrary *pattern matching* and prefix matching queries in $O(dm)$ time, where m is the length of the pattern
 - can be constructed in $O(dn)$ time



Tries and Web Search Engines

- The *index of a search engine* (collection of all searchable words) is stored into a compressed trie
- Each leaf of the trie is associated with a word and has a list of pages (URLs) containing that word, called *occurrence list*
- The trie is kept in internal memory
- The occurrence lists are kept in external memory and are ranked by relevance
- Boolean queries for sets of words (e.g., Java and coffee) correspond to set operations (e.g., intersection) on the occurrence lists
- Additional *information retrieval* techniques are used, such as
 - stopword elimination (e.g., ignore “the” “a” “is”)
 - stemming (e.g., identify “add” “adding” “added”)
 - link analysis (recognize authoritative pages)

Tries and Internet Routers

- Computers on the internet (hosts) are identified by a unique 32-bit IP (*internet protocol*) address, usually written in “dotted-quad-decimal” notation
- E.g., www.cs.brown.edu is 128.148.32.110
- Use nslookup on Unix to find out IP addresses
- An organization uses a subset of IP addresses with the same prefix, e.g., Brown uses 128.148.*.*, Yale uses 130.132.*.*
- Data is sent to a host by fragmenting it into packets. Each packet carries the IP address of its destination.
- The internet whose nodes are *routers*, and whose edges are communication links.
- A router forwards packets to its neighbors using IP *prefix matching* rules. E.g., a packet with IP prefix 128.148. should be forwarded to the Brown gateway router.
- Routers use tries on the alphabet 0,1 to do prefix matching.