DATA STRUCTURES AND PROBLEM SOLVING

Prepared by
Mr. Rajasekhar Nennuri
Assistant Professor
C.S.E Department

UNIT-1

BASIC CONCEPTS

DATA STRUCTURE

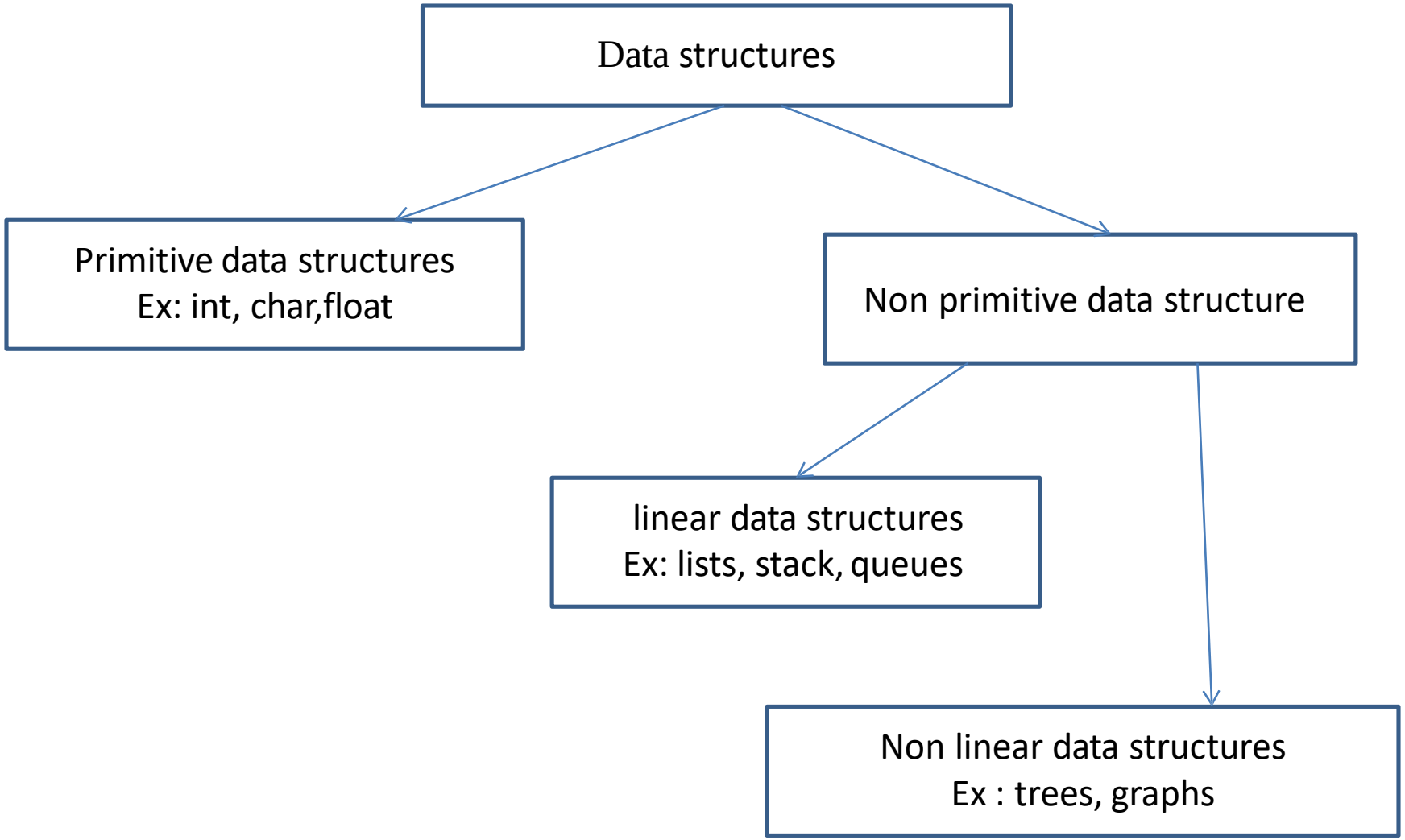
The data structure can be defined as the collection of elements and all the possible operations which are required for those set of elements.

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



OPERATIONS ON DATA STRUCTURES

1. Traversing- It is used to access each data item exactly once so that it can be processed.
2. Searching- It is used to find out the location of the data item if it exists in the given collection of data items.
3. Inserting- It is used to add a new data item in the given collection of data items.
4. Deleting- It is used to delete an existing data item from the given collection of data items.
5. Sorting- It is used to arrange the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.
6. Merging- It is used to combine the data items of two sorted files into single file in the sorted form.

ABSTRACT DATA TYPE

An abstract data type, sometimes abbreviated ADT, is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented. This means that we are concerned only with what the data is representing and not with how it will eventually be constructed.

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.

DIFFERENT APPROACHES TO DESIGN AN ALGORITHM

- Various design techniques exist:

  Classifying algorithms based on design ideas or commonality.

  General-problem solving strategies.

- Brute force
- Divide-and-conquer
- Decrease-and-conquer
- Transform-and-conquer
- Space-and-time tradeoffs
- Dynamic programming
- Greedy techniques

DIFFERENT APPROACHES TO DESIGN AN ALGORITHM

- Brute force

Selection sort, Brute-force string matching, Convex hull problem

Exhaustive search: Traveling salesman, Knapsack, and Assignment problems

- Divide-and-conquer

Master theorem, Mergesort, Quicksort, Quickhull

- Decrease-and-conquer

Insertion sort, Permutations (Minimal change approach, Johnson-Trotter algorithm)

Fake-coin problem (Ternary search), Computing a median, Topological sorting

.

DIFFERENT APPROACHES TO DESIGN AN ALGORITHM

- Transform-and-conquer

Gaussian elimination, Heaps and Heapsort, Problem reduction

- Space-and-time tradeoffs

String matching: Horspool's algorithm, Boyer-Moore algorithm

- Dynamic programming

Warshall's algorithm for transitive closure

Floyd's algorithms for all-pairs shortest paths

DIFFERENT APPROACHES TO DESIGN AN ALGORITHM

- Greedy techniques

MST problem: Prim's algorithm, Kruskal's algorithm (Sets and set operations)

Dijkstra's algorithm for single-source shortest path problem

Huffman tree and code

- More on algorithms.

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

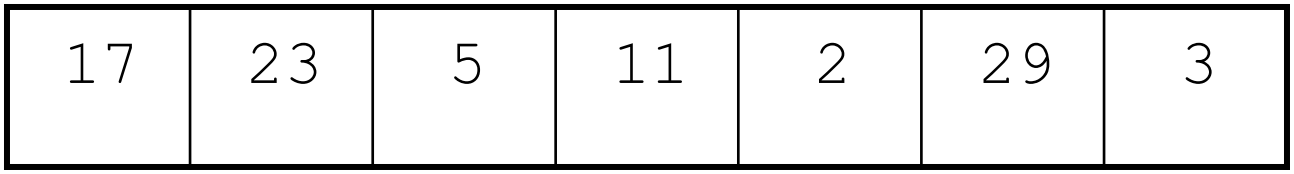
Factorial of a number

```
Algorithm factorial(a)
int a;
        {
          int fact=1
              if(a>1)
              Fact = a* factorial(a-1);
              Return(fact);
        }
```

SEARCHING TECHNIQUES

LINEAR SEARCH - EXAMPLE

- Array numlist contains:



- Searching for the the value 11, linear search examines 17, 23, 5, and 11
- Searching for the the value 7, linear search examines 17, 23, 5, 11, 2, 29, and 3

LINEAR SEARCH

PROS

- Easy to understand
- Array can be of any order

CONS

- Ineffiencient for an array of N elements

BINARY SEARCH

A **binary search** looks for an item in a list using a divide-and-conquer strategy.

BINARY SEARCH

Requires array elements to be in order

1. Divides the array into three sections:
   - middle element
   - elements on one side of the middle element
   - elements on the other side of the middle element
2. If the middle element is the correct value, done. Otherwise, go to step 1. using only the half of the array that may contain the correct value.
3. Continue steps 1. and 2. until either the value is found or there are no more elements to examine

BINARY SEARCH

$$mid = \frac{left + right}{2}$$

BINARY SEARCH

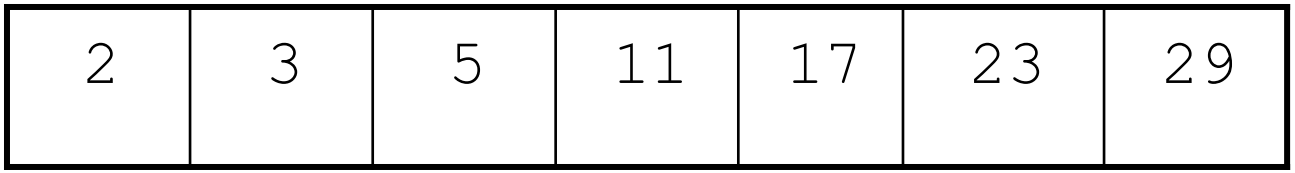
```
bool BinSearch(double list[ ], int n, double item, int&index)
{
        int left=0;
        int right=n-1;
        int mid;
        while(left<=right)
{
       mid=(left+right)/2;
```

BINARY SEARCH

```
if(item> list [mid]){ left=mid+1; }
        else if(item< list [mid]){right=mid-1;}
        else{
        item= list [mid];
        index=mid;
        return true; }
     }// while
   return false;
  }
```

BINARY SEARCH

- Array numlist2 contains:



- Searching for the the value 11, binary search examines 11 and stops
- Searching for the the value 7, binary search examines 11, 3, 5, and stops

BINARY SEARCH

- Benefits:
  - Much more efficient than linear search. For array of N elements, performs at most $\boldsymbol{log_2 N}$ comparisons

- Disadvantages:
  - Requires that array elements be sorted

FIBONACCI SEARCH

A possible improvement in binary search is not to use the middle element at each step, but to guess more precisely where the key being sought falls within the current interval of interest.

This improved version is called **fibonacci search**. Instead of splitting the array in the middle, this implementation splits the array corresponding to the **fibonacci numbers**, which are defined in the following manner.

$F_0 = 0$, $F_1 = 1$ $F_n = F_{n-1} + F_{n-2}$ for $n >= 2$.

FIBONACCI SEARCH

- Fibonacci search is used to search an element of a sorted array with the help of Fibonacci numbers. It studies the locations whose addresses have lower dispersion. Fibonacci number is subtracted from the index thereby reducing the size of the list.
- When the search element has non-uniform access memory storage, the Fibonacci search algorithm reduces the average time needed for accessing a storage location.
- Time complexity:-0(log (n))

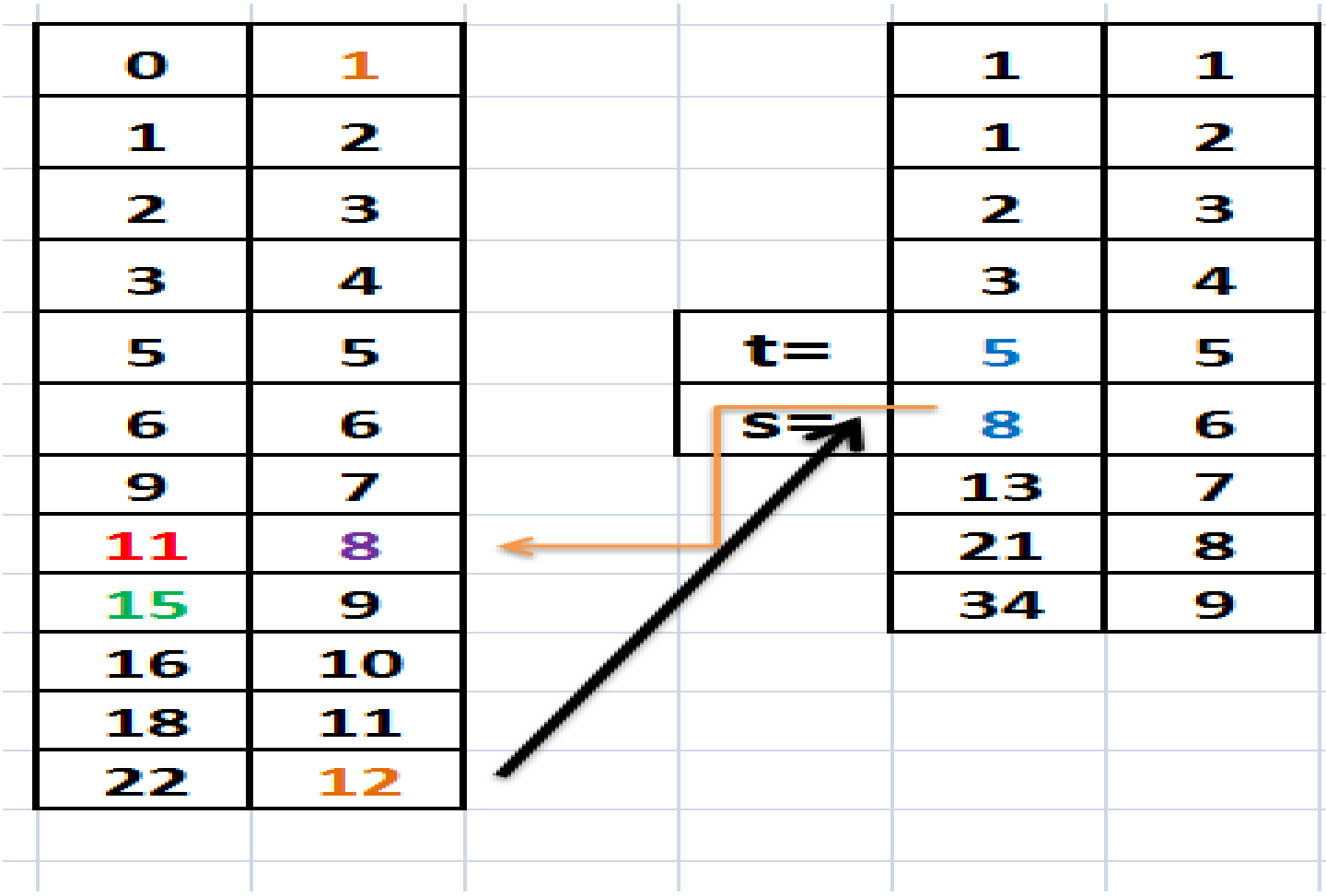
FIBONACCI SEARCH

- 0 1 2 3 5 6 9 11 15 16 18 22

Searching element

0	1
1	2
2	3
3	4
5	5
6	6
9	7
11	8
15	9
16	10
18	11
22	12

1	1
1	2
2	3
3	4
5	5
8	6
13	7
21	8
34	9


0 1
1 2
2 3
3 4
5 5
6 6
9 7
11 8
15 9
16 10
18 11
22 12
t=
s=
1 1
1 2
2 3
3 4
5 5
8 6
13 7
21 8
34 9

0	1
1	2
2	3
3	4
5	5
6	6
9	7
11	8
15	9
16	10
18	11
22	12

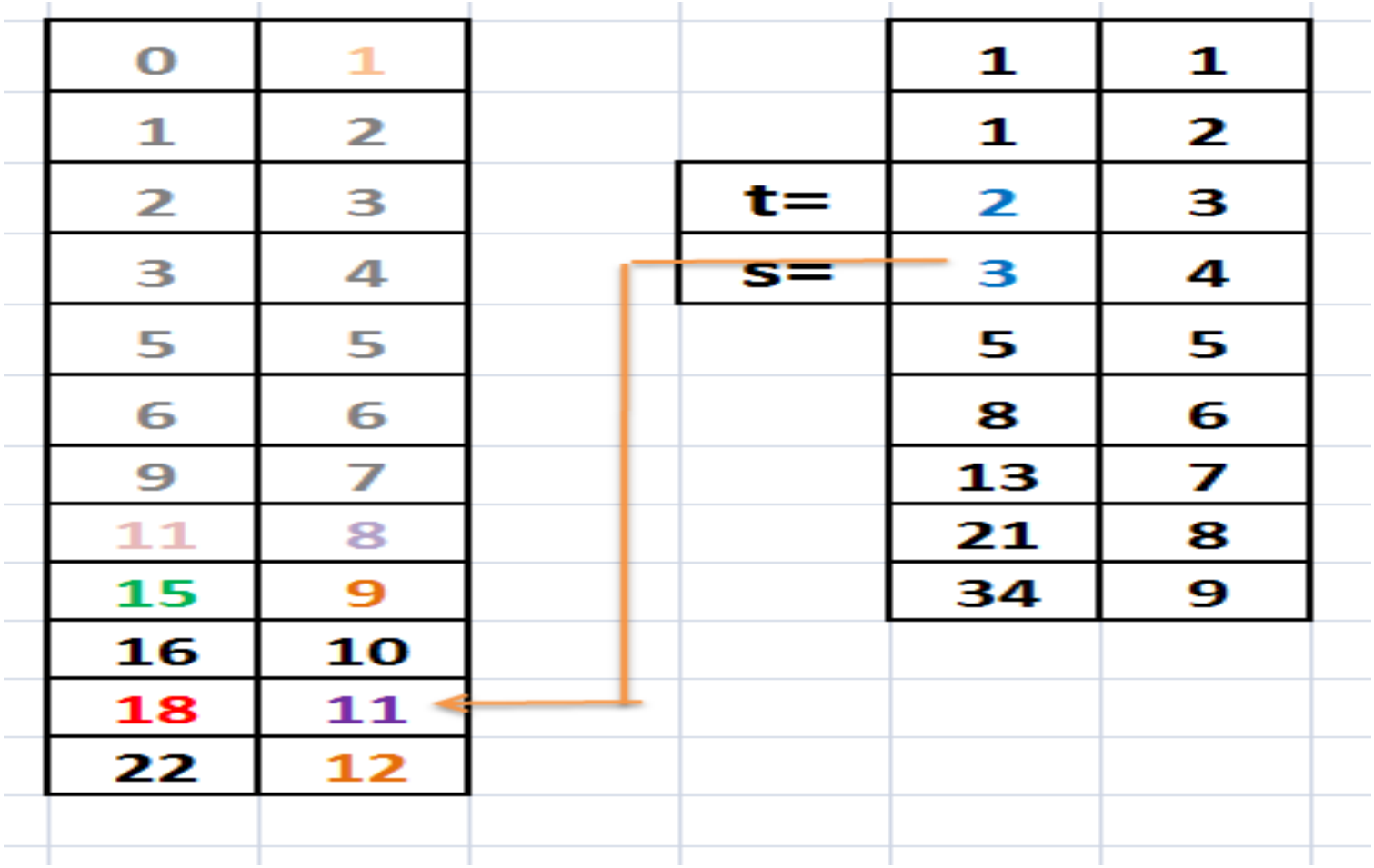
	1	1
	1	2
	2	3
	3	4
t=	5	5
s=	8	6
	13	7
	21	8
	34	9

11=15?

11<15

then shift right by s

RIGHT MEAN DOWN IN THIS



0	1		1	1
1	2		1	2
2	3	t=	2	3
3	4	s=	3	4
5	5		5	5
6	6		8	6
9	7		13	7
11	8		21	8
15	9		34	9
16	10			
18	11			
22	12			

0	1
1	2
2	3
3	4
5	5
6	6
9	7
11	8
15	9
16	10
18	11
22	12

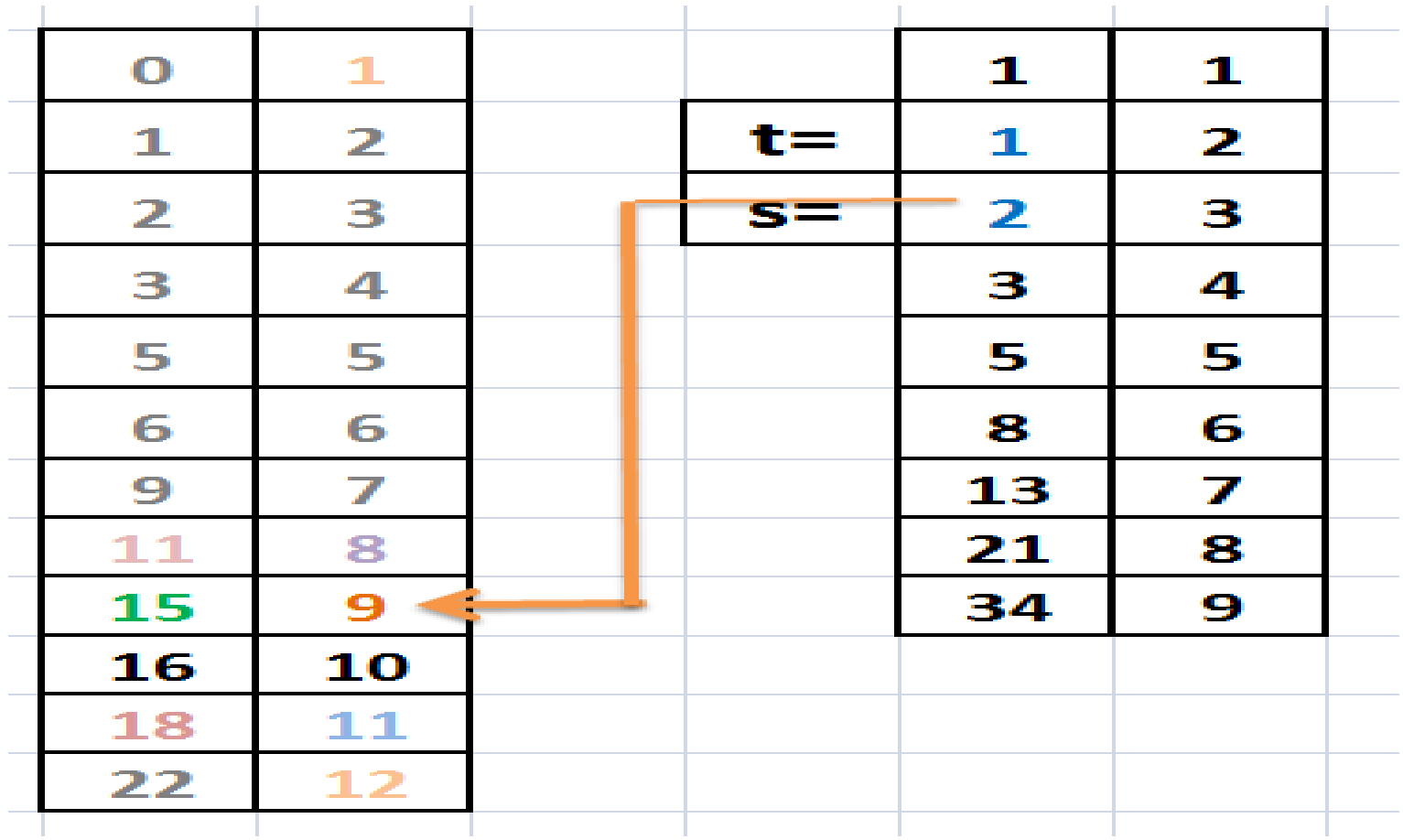
	1	1
	1	2
t=	2	3
s=	3	4
	5	5
	8	6
	13	7
	21	8
	34	9

18=15?

18>15

then shift left by s

LEFT MEAN UP IN THIS


0 1
1 2
2 3
3 4
5 5
6 6
9 7
11 8
15 9
16 10
18 11
22 12
t=
s=
1 1
1 2
2 3
3 4
5 5
8 6
13 7
21 8
34 9

0	1
1	2
2	3
3	4
5	5
6	6
9	7
11	8
15	9
16	10
18	11
22	12

t=	1	1
s=	1	2
	2	3
	3	4
	5	5
	8	6
	13	7
	21	8
	34	9

15=15?

15=15

then will be answer A(9)

SORTING TECHNIQUES

SORTING

- To arrange a set of items in sequence.
- It is estimated that 25~50% of all computing power is used for sorting activities.
- Possible reasons:
  - Many applications require sorting;
  - Many applications perform sorting when they don't have to;
  - Many applications use inefficient sorting algorithms.

SORTING

Sorting: an operation that segregates items into groups according to specified criterion.

A = { 3 1 6 2 1 3 4 5 9 0 }

A = { 0 1 1 2 3 3 4 5 6 9 }

SORTING

- Internal Sort
  - The data to be sorted is all stored in the computer's main memory.
- External Sort
  - Some of the data to be sorted might be stored in some external, slower, device.
- In Place Sort
  - The amount of extra space required to sort the data is constant with the input size.

SORTING

There are many, many different types of sorting algorithms, but the primary ones are:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Shell Sort
- Radix Sort
- Swap Sort
- Heap Sort

INSERTION SORT

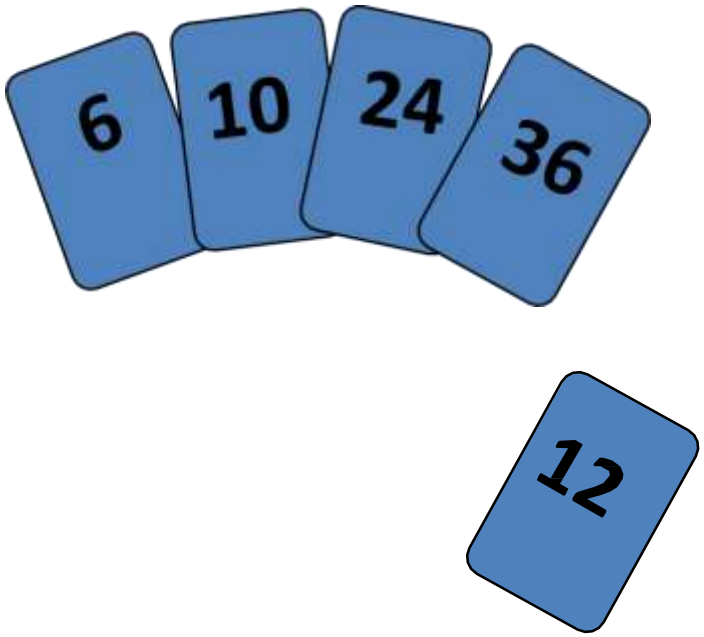
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand

INSERTION SORT

- compare it with each of the cards already in the hand, from right to left

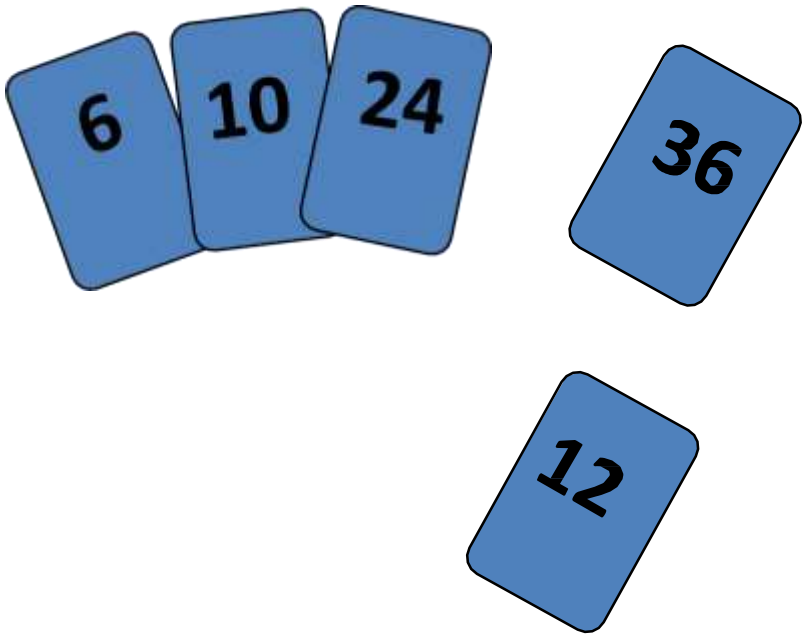
– The cards held in the left hand are sorted
  - these cards were originally the top cards of the pile on the table

INSERTION SORT

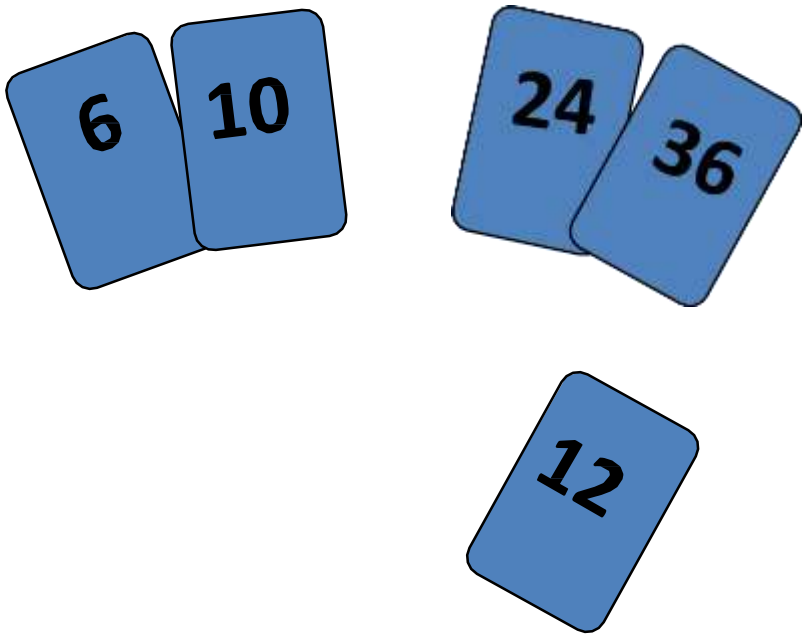


To insert 12, we need to make room for it by moving first 36 and then 24.

INSERTION SORT



INSERTION SORT

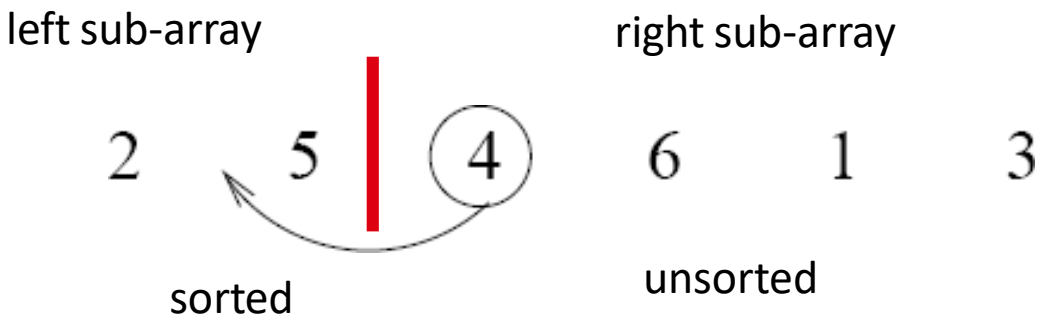


INSERTION SORT

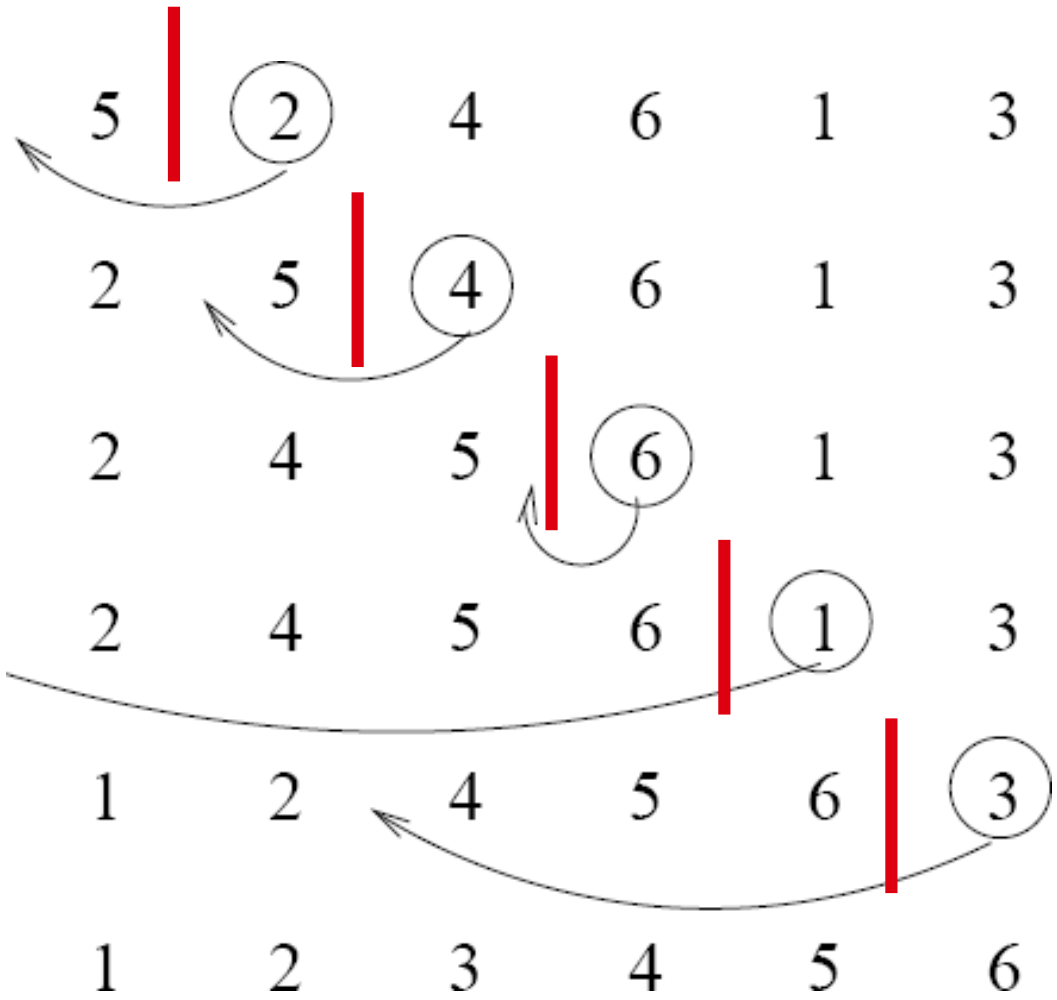
input array

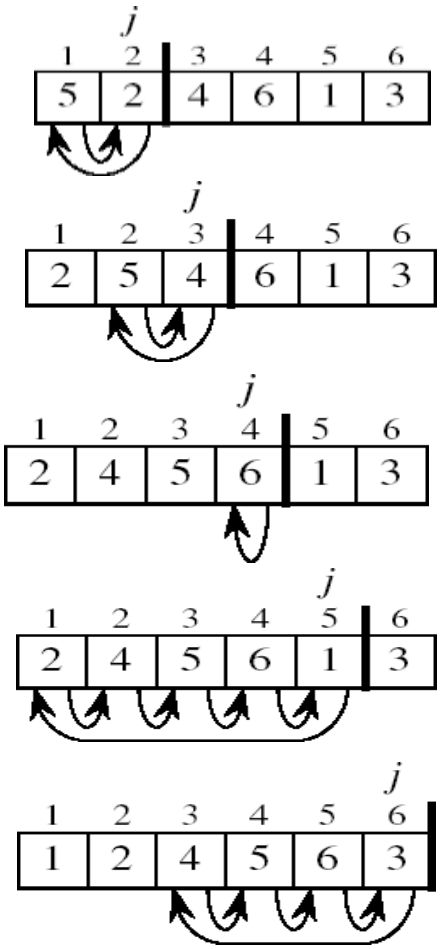
5 2 4 6 1 3

at each iteration, the array is divided in two sub-arrays:



INSERTION SORT





INSERTION SORT

- Running time analysis:
  - Worst case: $O(N^2)$
  - Best case: O(N)

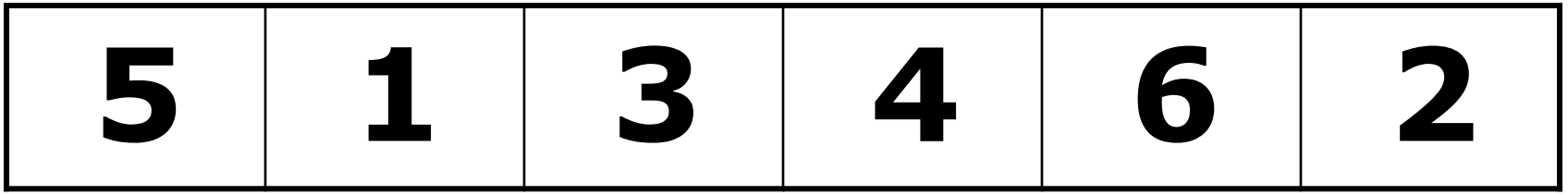
SELECTION SORT

1. We have two group of items:
   - sorted group, and
   - unsorted group
2. Initially, all items are in the unsorted group. The sorted group is empty.
   - We assume that items in the unsorted group unsorted.
   - We have to keep items in the sorted group sorted.

SELECTION SORT

1. Select the “best” (eg. smallest) item from the unsorted group, then put the “best” item at the end of the sorted group.
2. Repeat the process until the unsorted group becomes empty.

SELECTION SORT

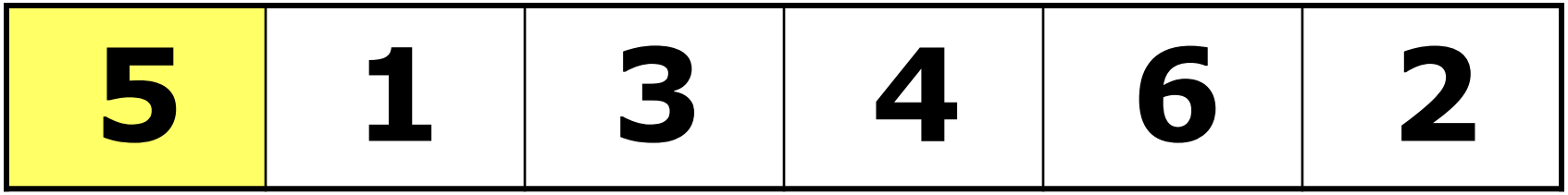


Comparison

Data Movement

Sorted

SELECTION SORT

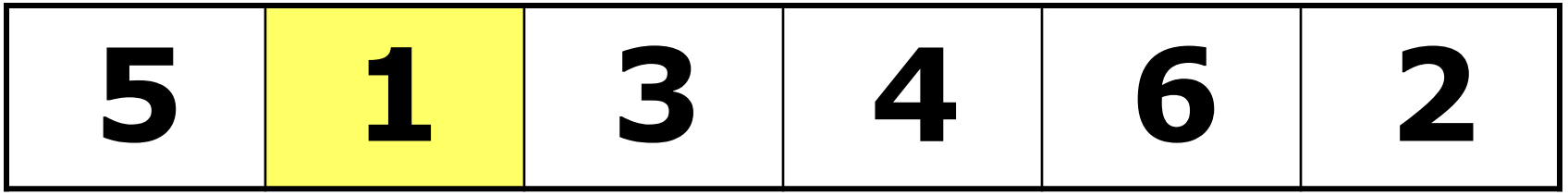


Comparison

Data Movement

Sorted

SELECTION SORT

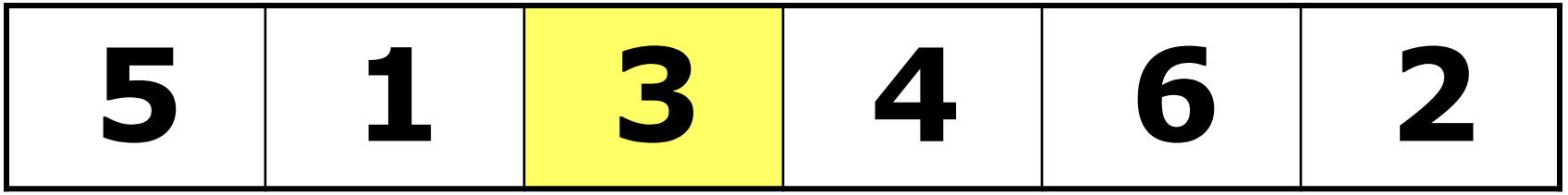


Comparison

Data Movement

Sorted

SELECTION SORT

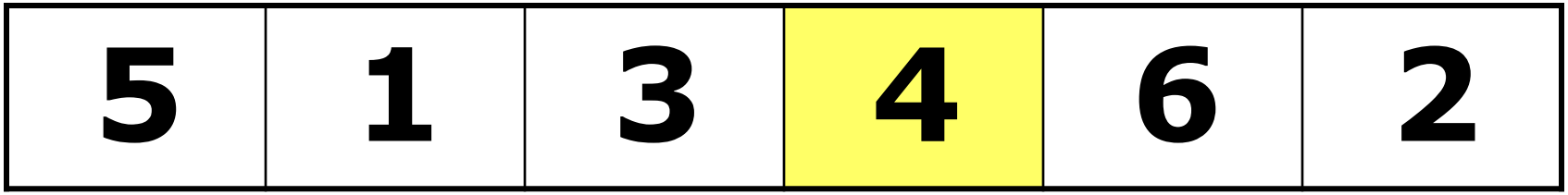


Comparison

Data Movement

Sorted

SELECTION SORT

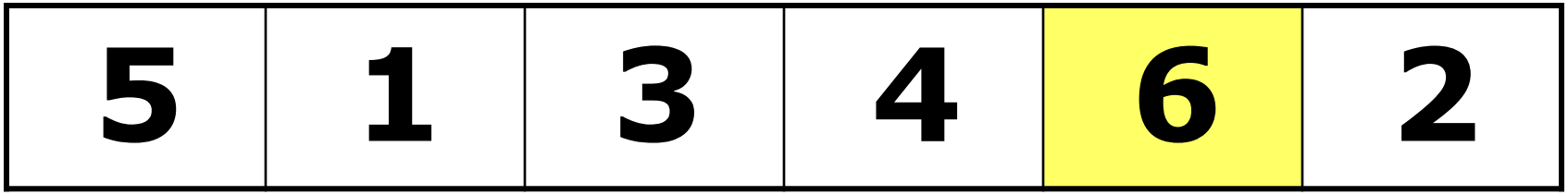


Comparison

Data Movement

Sorted

SELECTION SORT

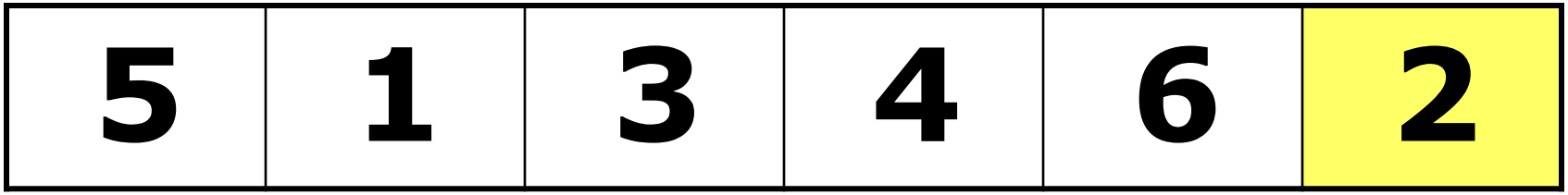


Comparison

Data Movement

Sorted

SELECTION SORT

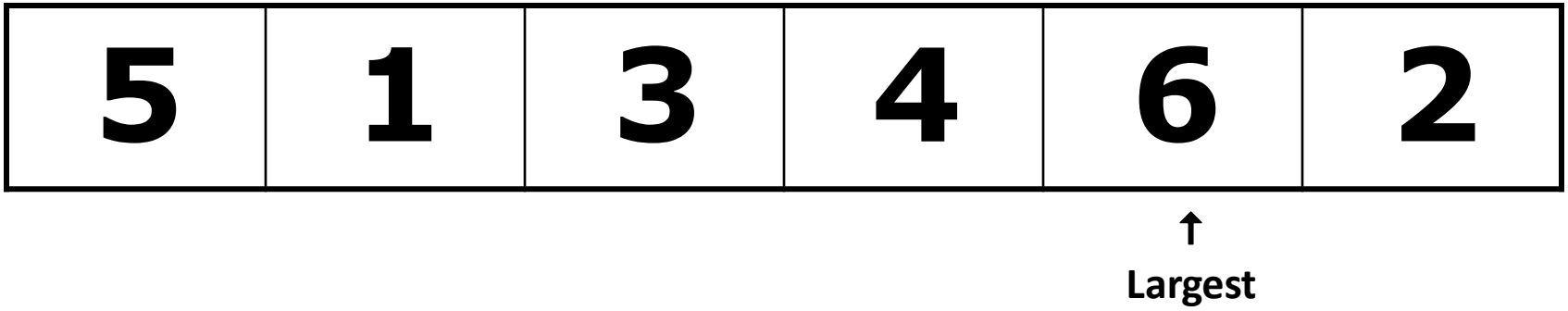


Comparison

Data Movement

Sorted

SELECTION SORT

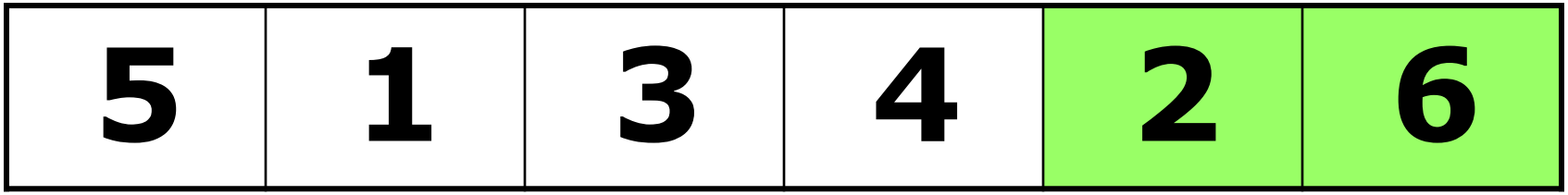


Comparison

Data Movement

Sorted

SELECTION SORT

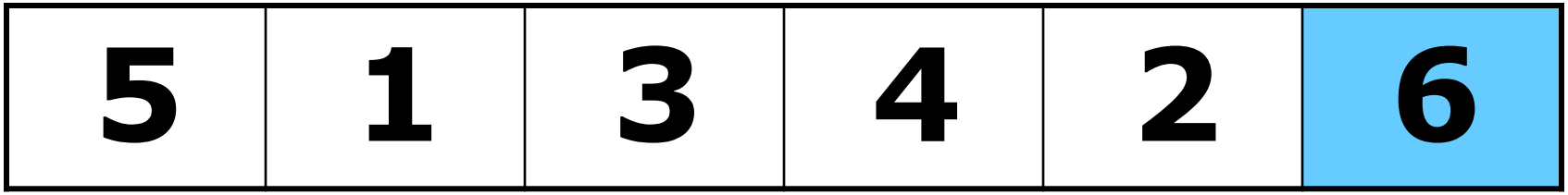


Comparison

Data Movement

Sorted

SELECTION SORT

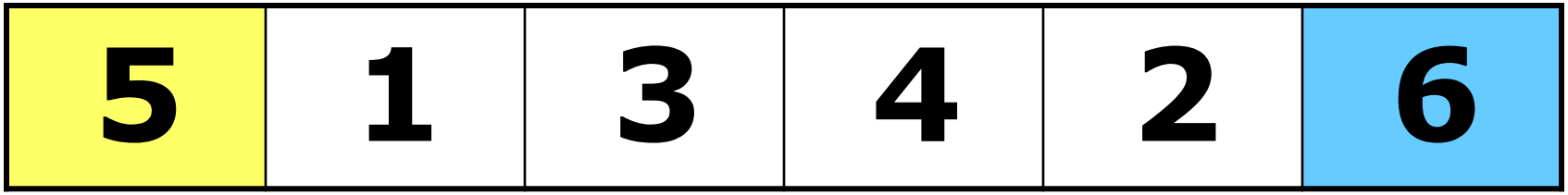


Comparison

Data Movement

Sorted

SELECTION SORT

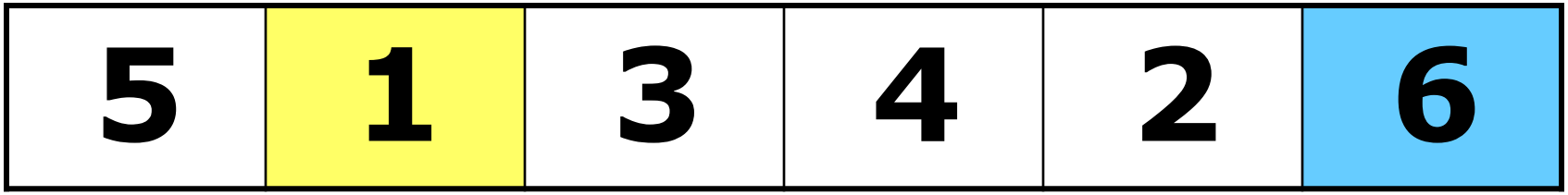


Comparison

Data Movement

Sorted

SELECTION SORT

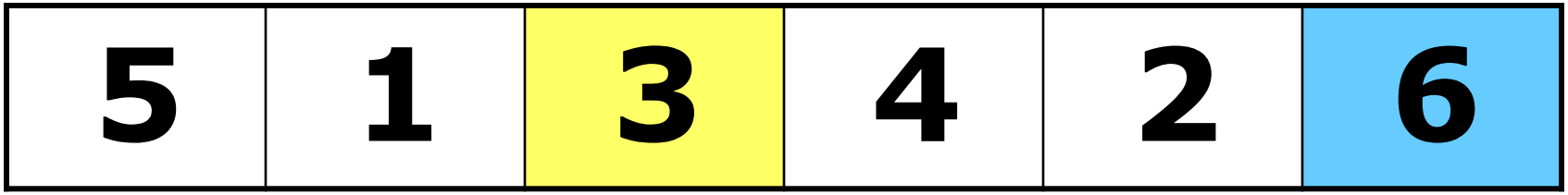


Comparison

Data Movement

Sorted

SELECTION SORT

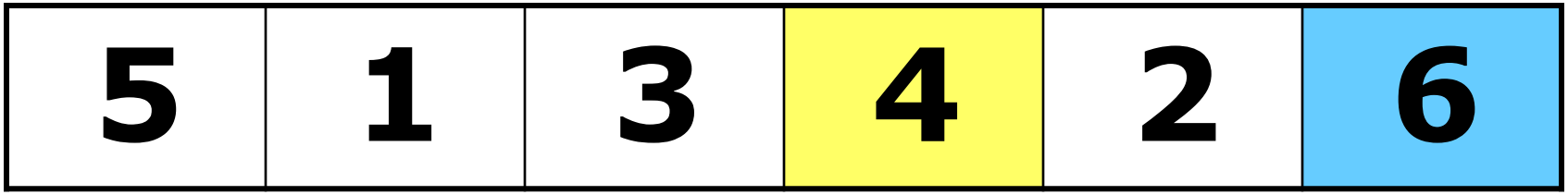


Comparison

Data Movement

Sorted

SELECTION SORT

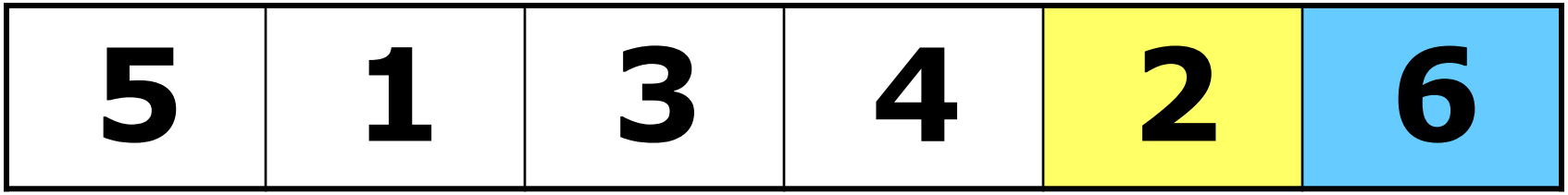


Comparison

Data Movement

Sorted

SELECTION SORT

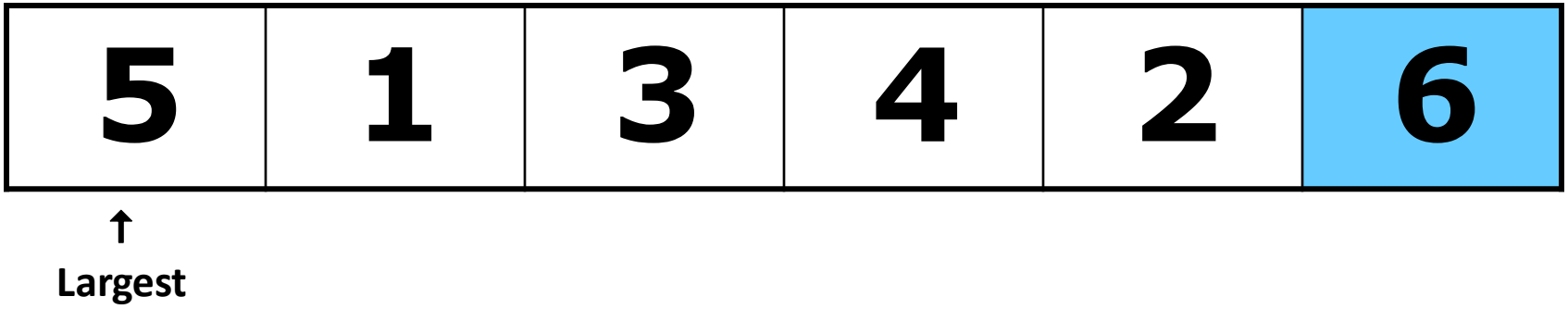


Comparison

Data Movement

Sorted

SELECTION SORT

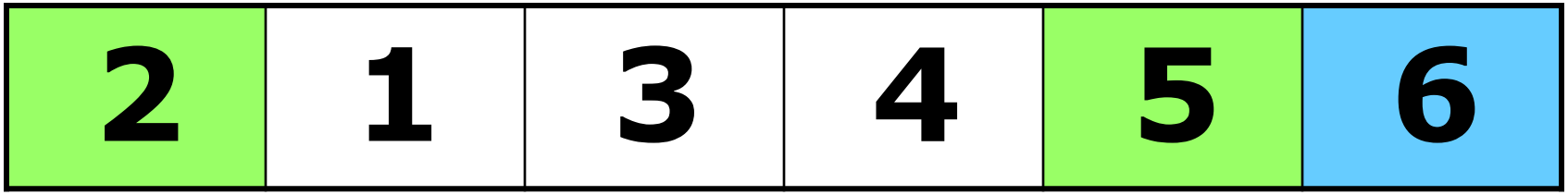


Comparison

Data Movement

Sorted

SELECTION SORT

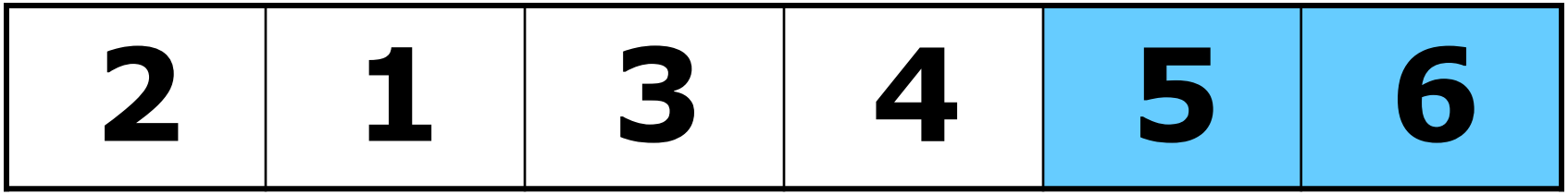


Comparison

Data Movement

Sorted

SELECTION SORT

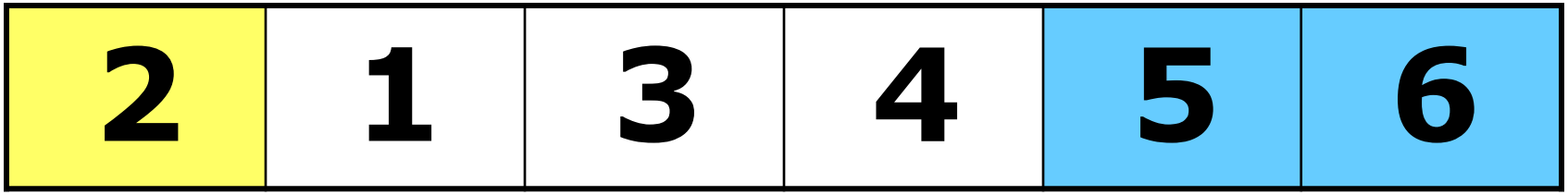


Comparison

Data Movement

Sorted

SELECTION SORT

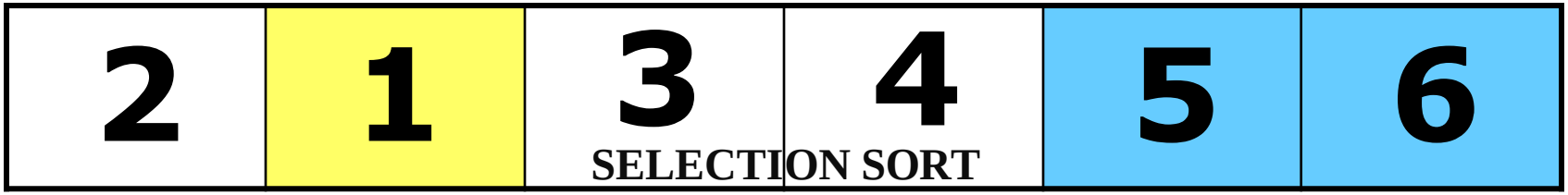


Comparison

Data Movement

Sorted

SELECTION SORT

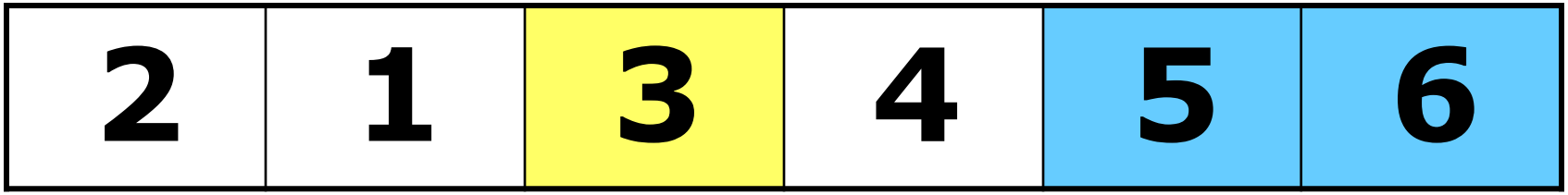


Comparison

Data Movement

Sorted

SELECTION SORT

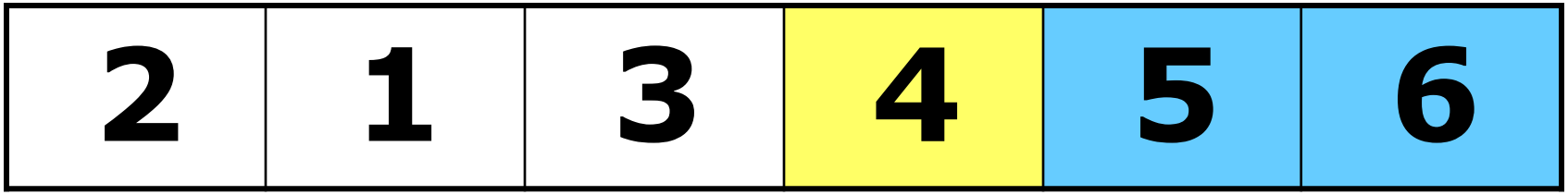


Comparison

Data Movement

Sorted

SELECTION SORT

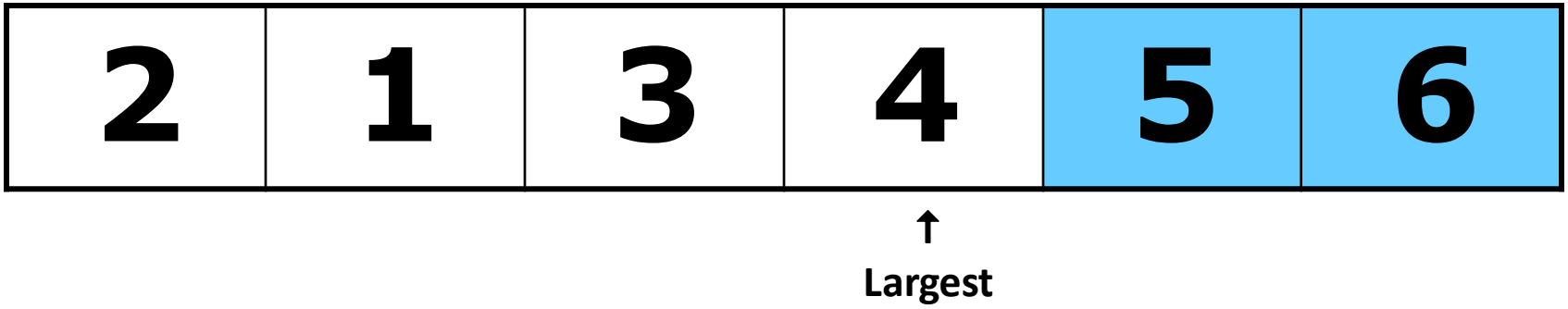


Comparison

Data Movement

Sorted

SELECTION SORT

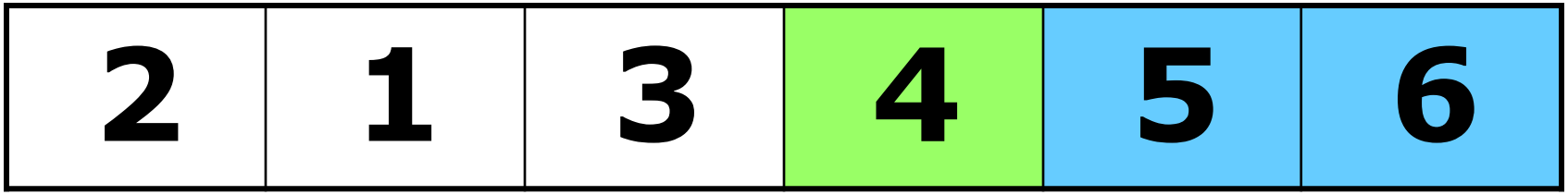


Comparison

Data Movement

Sorted

SELECTION SORT

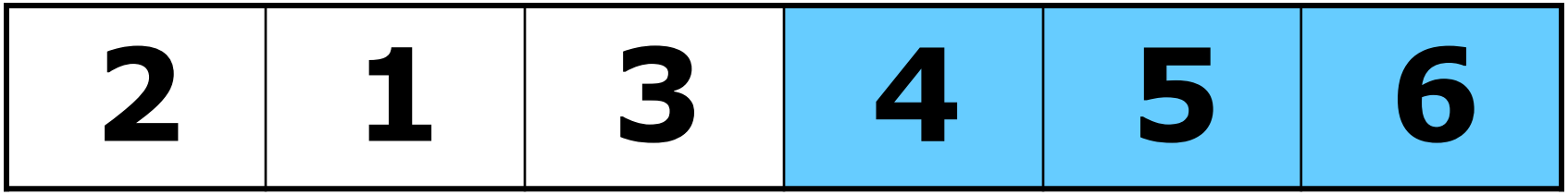


Comparison

Data Movement

Sorted

SELECTION SORT

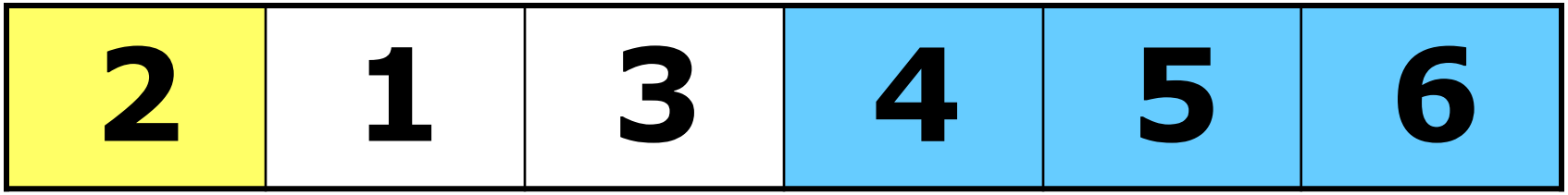


Comparison

Data Movement

Sorted

SELECTION SORT

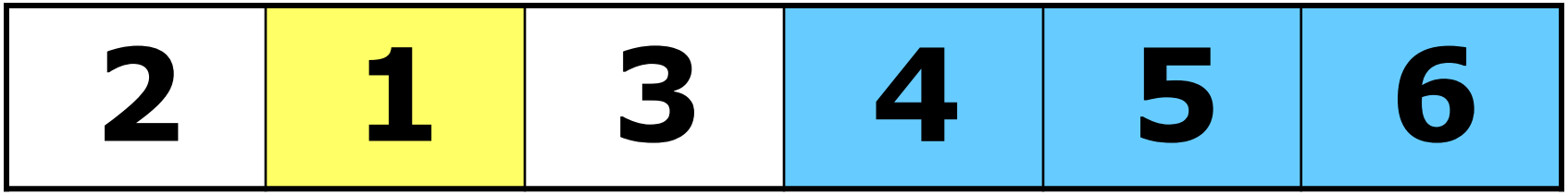


Comparison

Data Movement

Sorted

SELECTION SORT

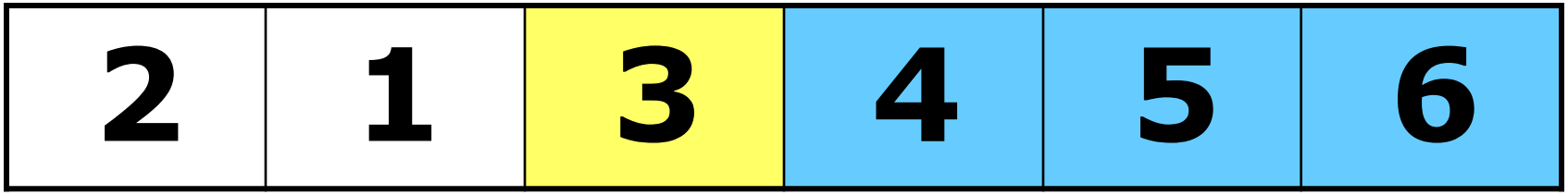


Comparison

Data Movement

Sorted

SELECTION SORT

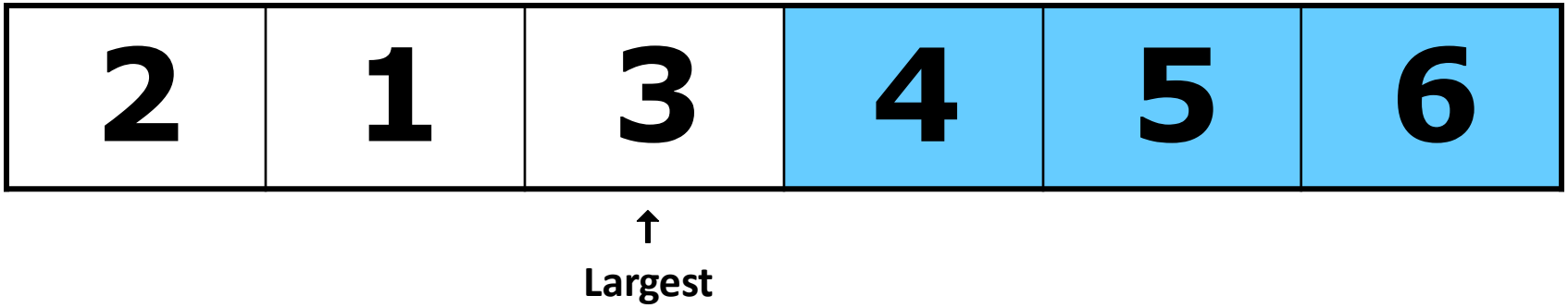


Comparison

Data Movement

Sorted

SELECTION SORT

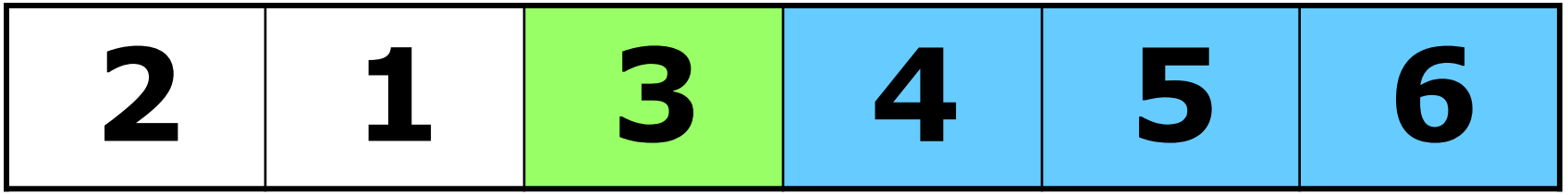


Comparison

Data Movement

Sorted

SELECTION SORT

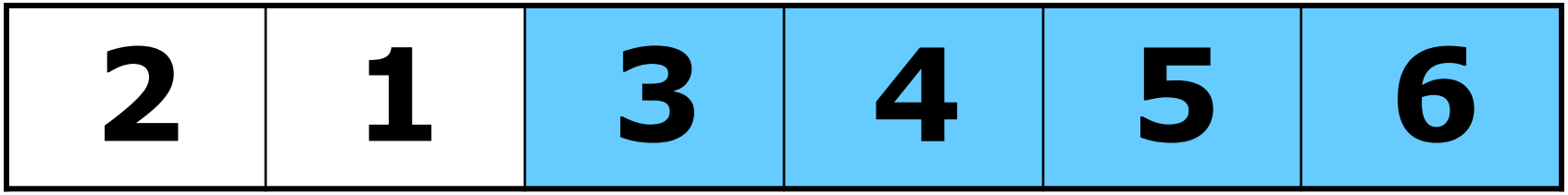


Comparison

Data Movement

Sorted

SELECTION SORT

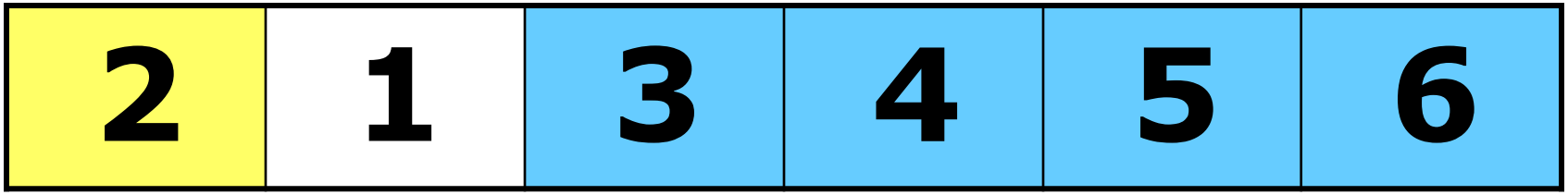


Comparison

Data Movement

Sorted

SELECTION SORT

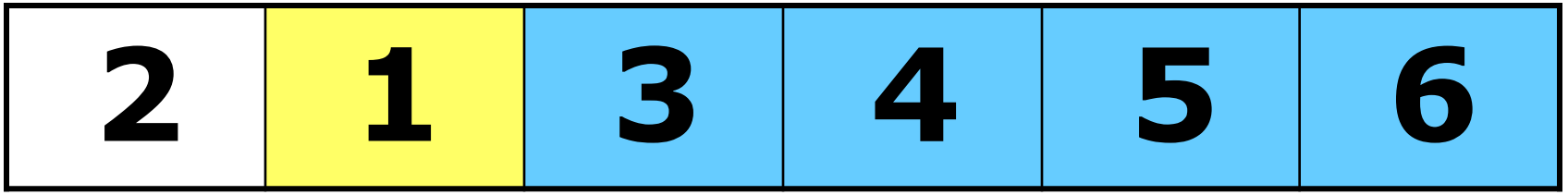


Comparison

Data Movement

Sorted

SELECTION SORT

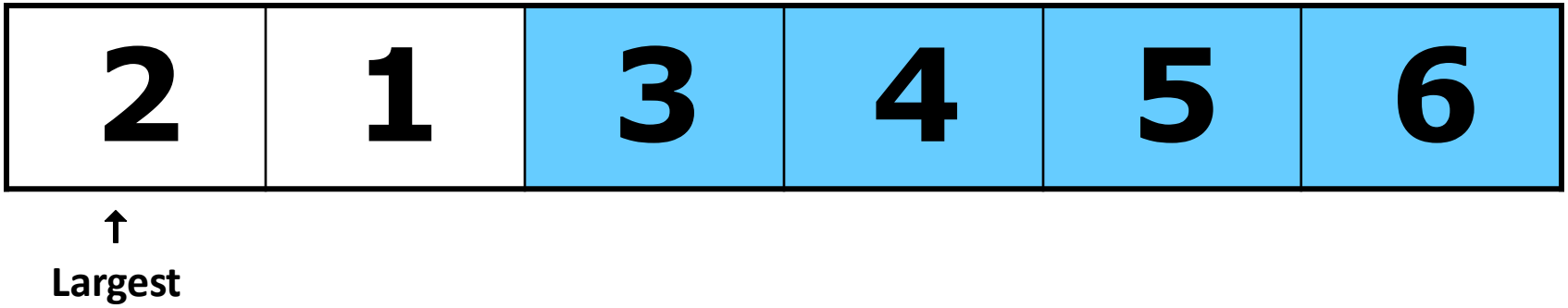


Comparison

Data Movement

Sorted

SELECTION SORT

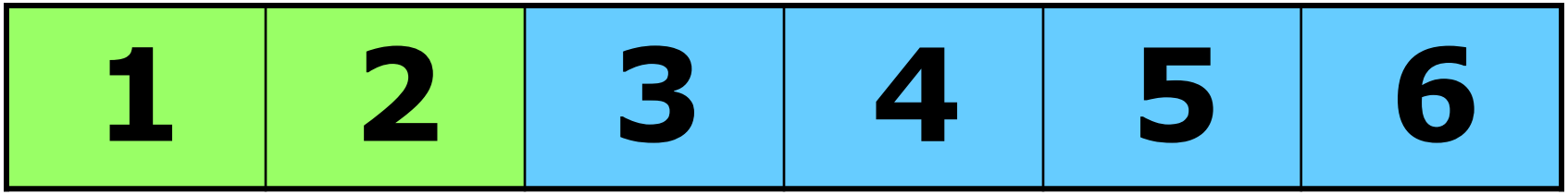


Comparison

Data Movement

Sorted

SELECTION SORT

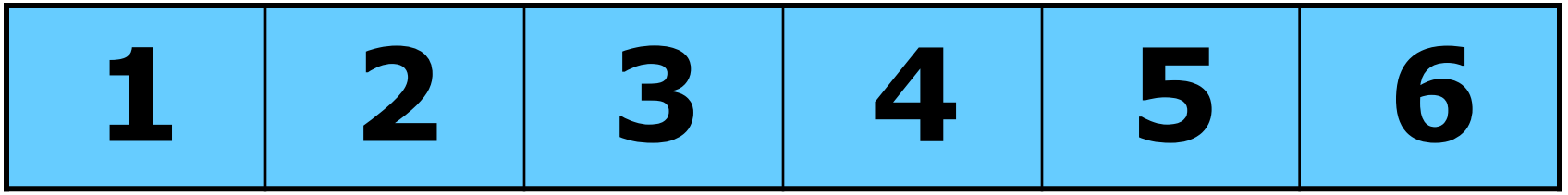


Comparison

Data Movement

Sorted

SELECTION SORT



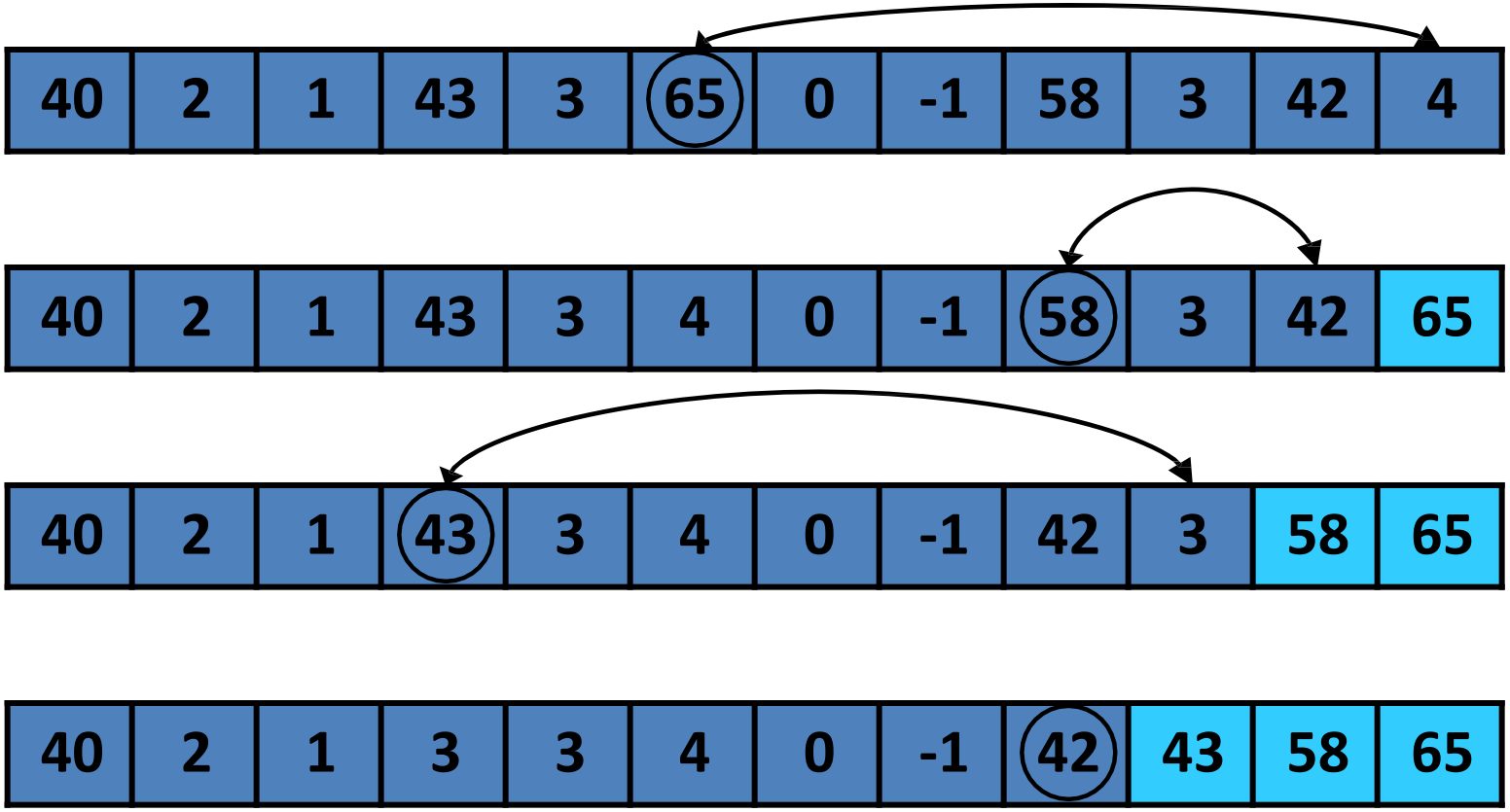
DONE!

Comparison

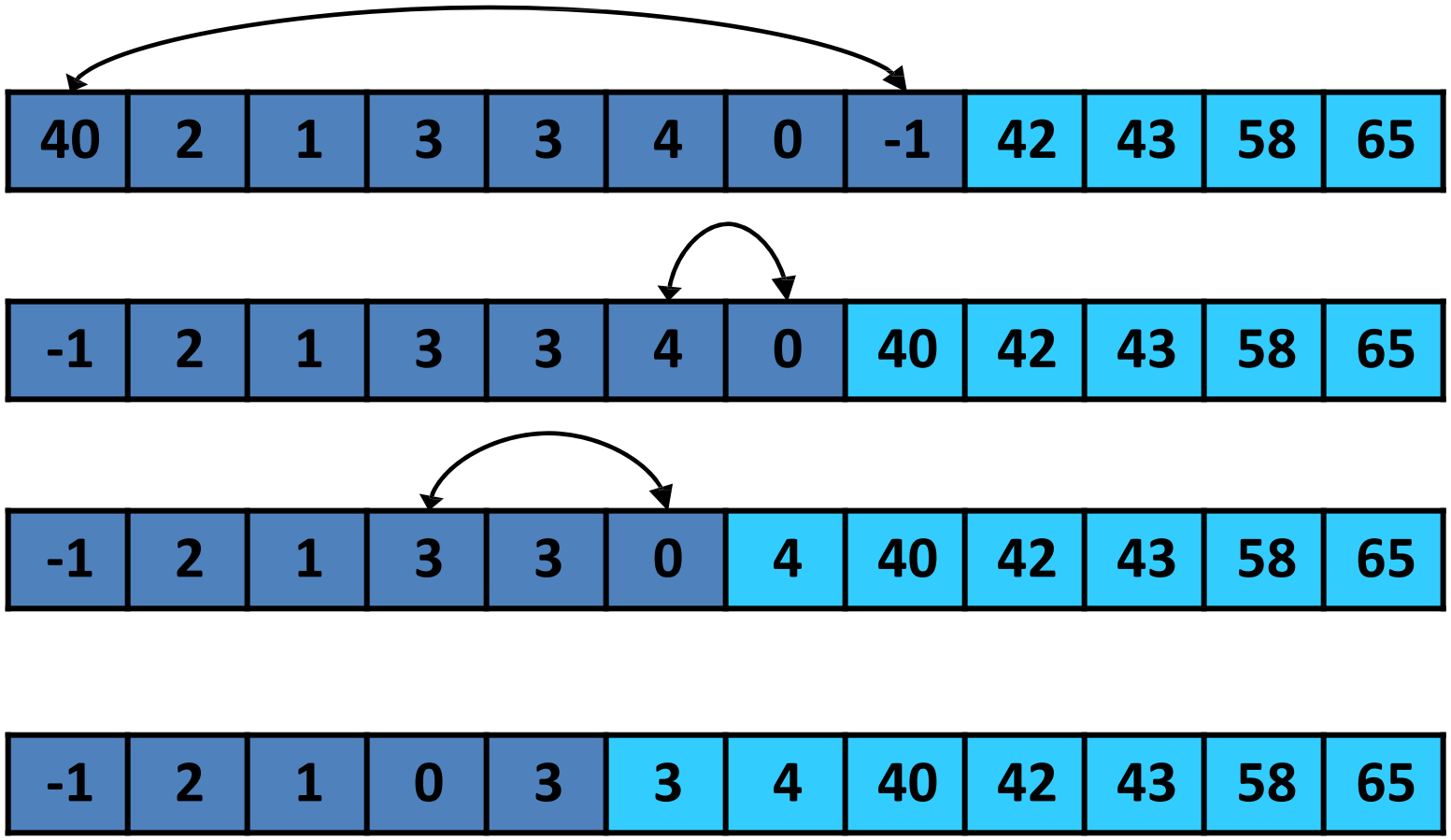
Data Movement

Sorted

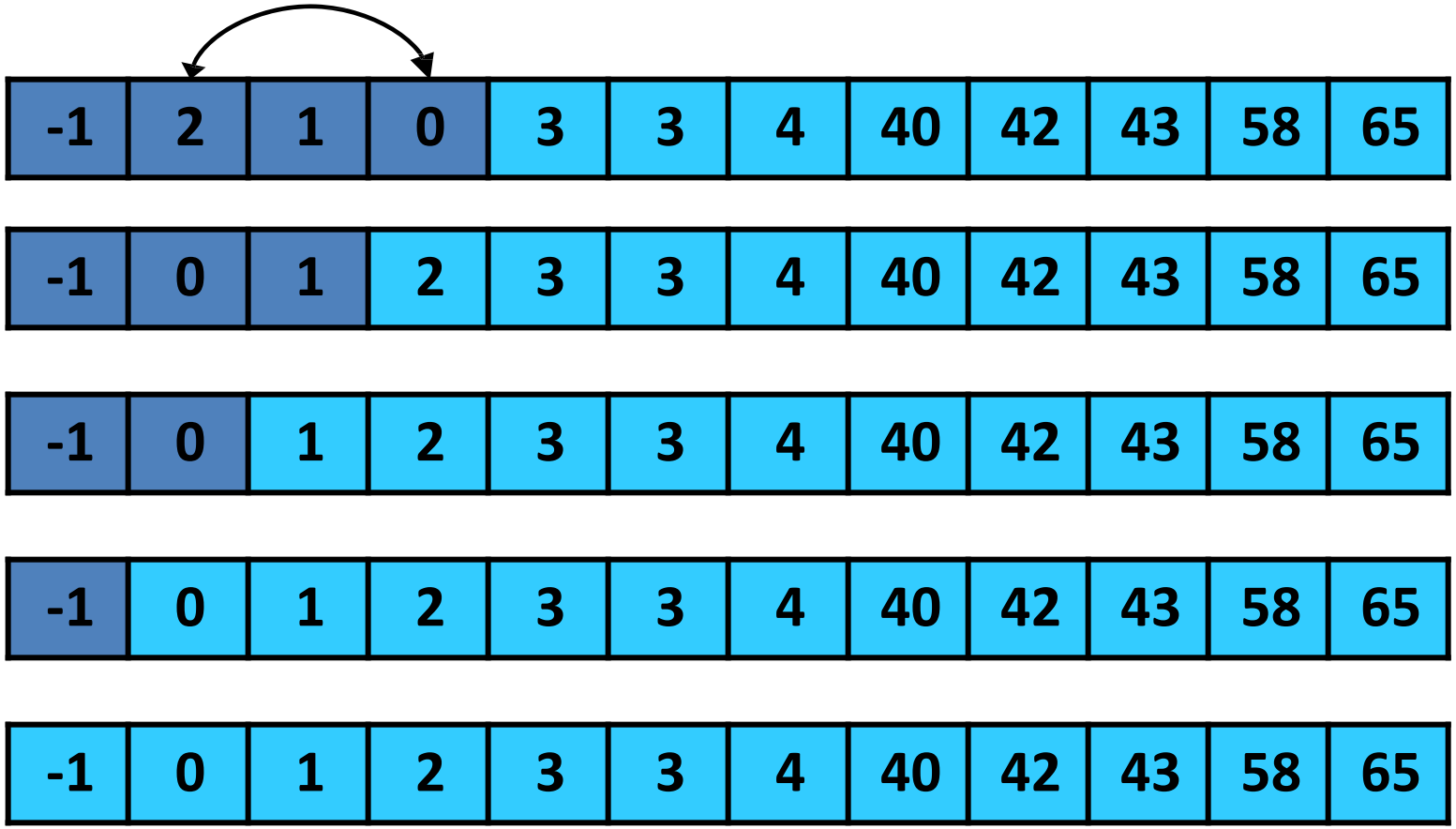
SELECTION SORT



SELECTION SORT



SELECTION SORT

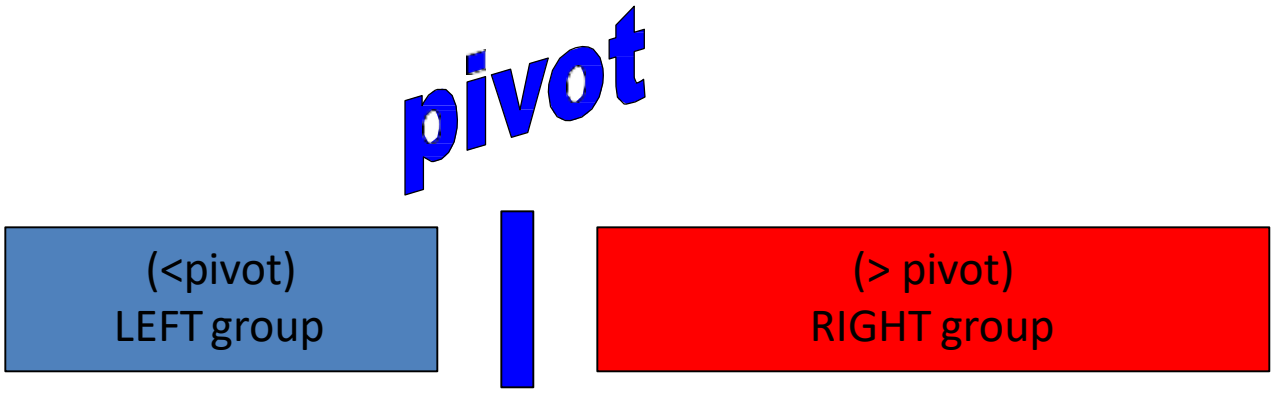


SELECTION SORT

- Running time:
  - Worst case: $O(N^2)$
  - Best case: $O(N^2)$

QUICKSORT

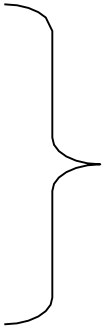
- Basic Concept: divide and conquer
- Select a pivot and split the data into two groups: (< pivot) and (> pivot):



- Recursively apply Quicksort to the subgroups

QUICKSORT

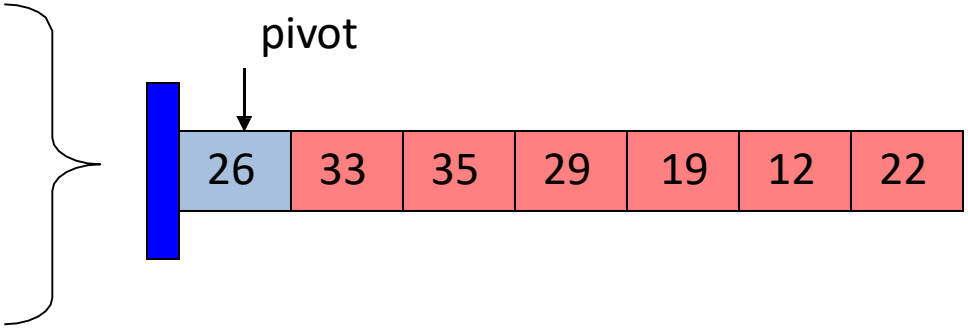
Start with all data
in an array, and
consider it unsorted



Unsorted Array

QUICKSORT

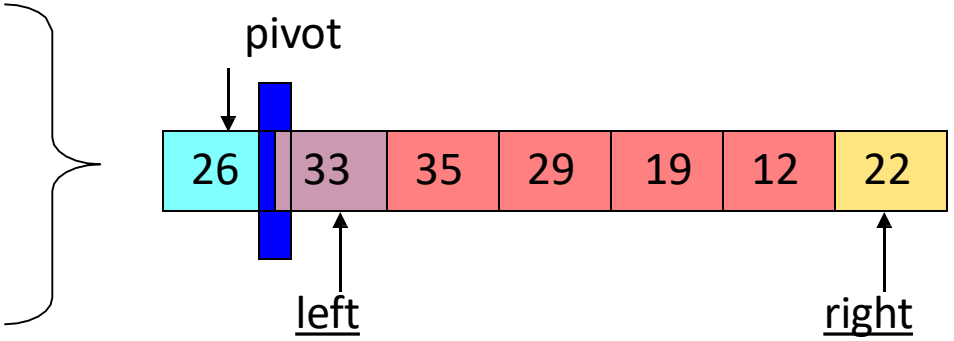
Step 1, select a pivot
(it is arbitrary)



We will select the first element, as presented in the original algorithm by C.A.R. Hoare in 1962.

QUICKSORT

Step 2, start process of dividing data into LEFT and RIGHT groups:



The LEFT group will have elements less than the pivot.

The RIGHT group will have elements greater that the pivot.

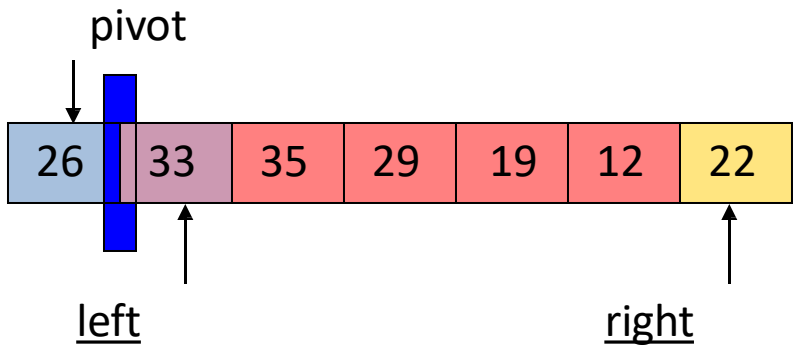
Use markers left and right

QUICKSORT

Step 3,
If left element belongs to LEFT group, then increment left index.

If right index element belongs to RIGHT, then decrement right.

Exchange when you find elements that belong to the other group.



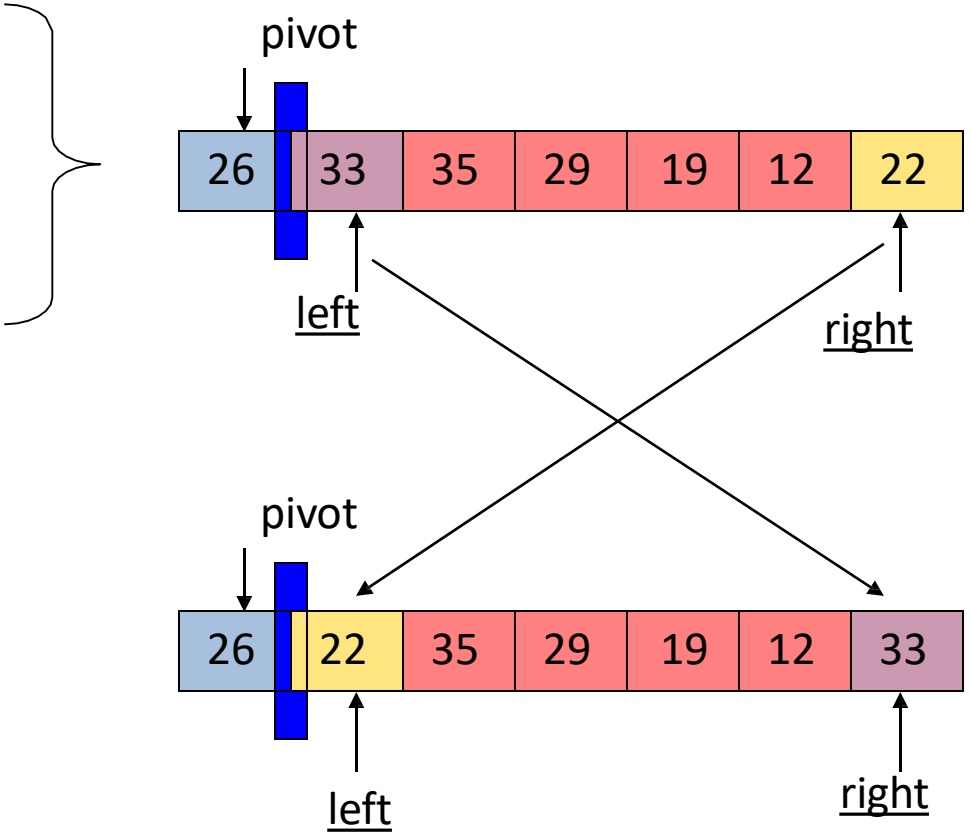
QUICKSORT

Step 4:

Element 33 belongs to RIGHT group.

Element 22 belongs to LEFT group.

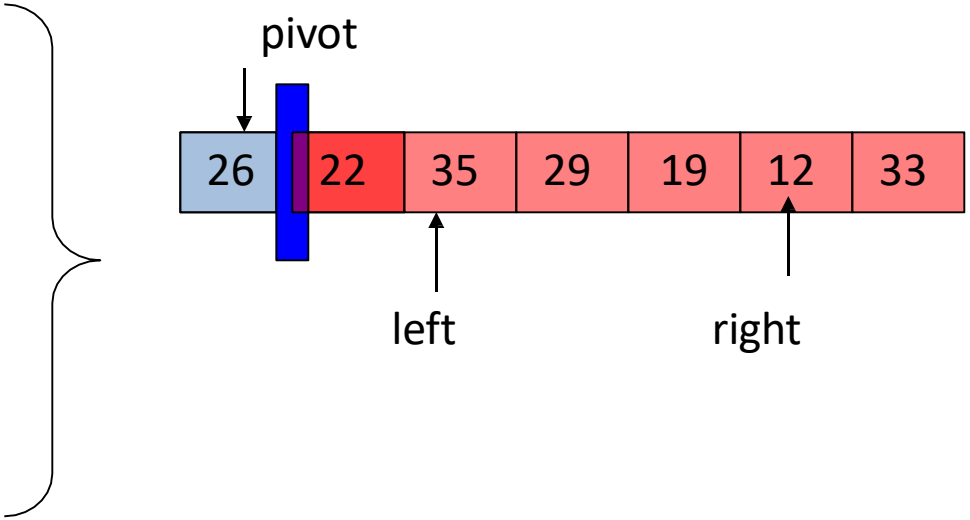
Exchange the two elements.



QUICKSORT

Step 5:

After the exchange,
increment left marker,
decrement right marker.



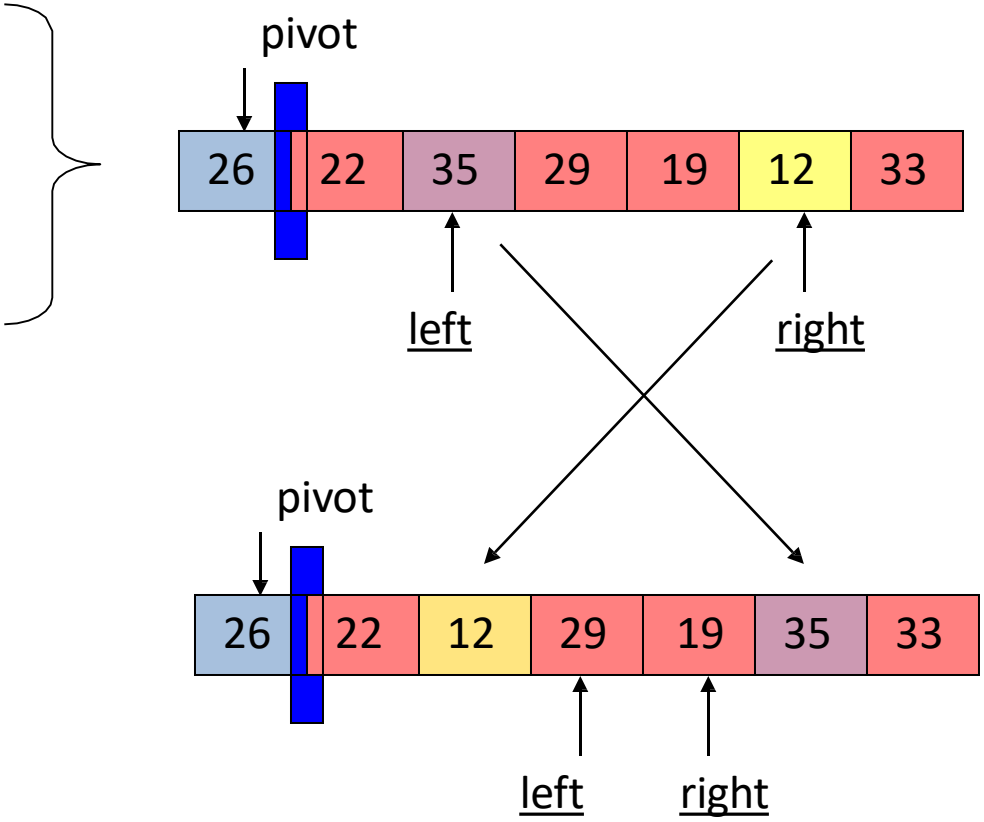
QUICKSORT

Step 6:

Element 35 belongs to RIGHT group.

Element 12 belongs to LEFT group.

Exchange, increment left, and decrement right.



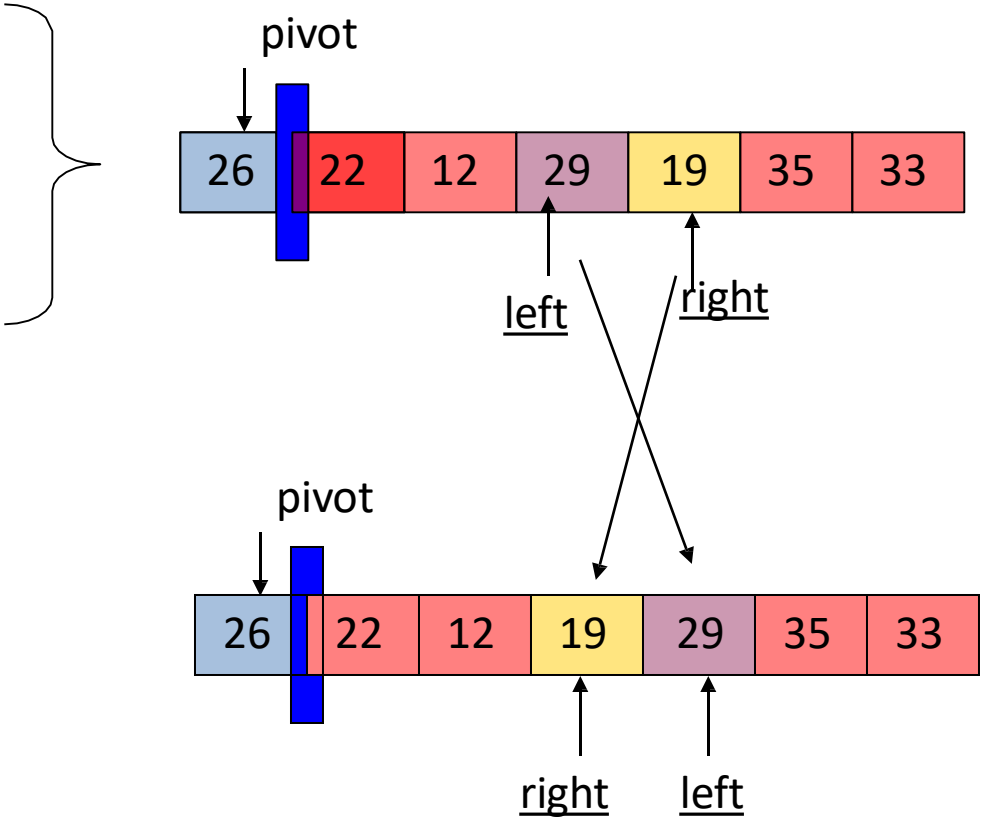
QUICKSORT

Step 7:

Element 29 belongs to RIGHT.

Element 19 belongs to LEFT.

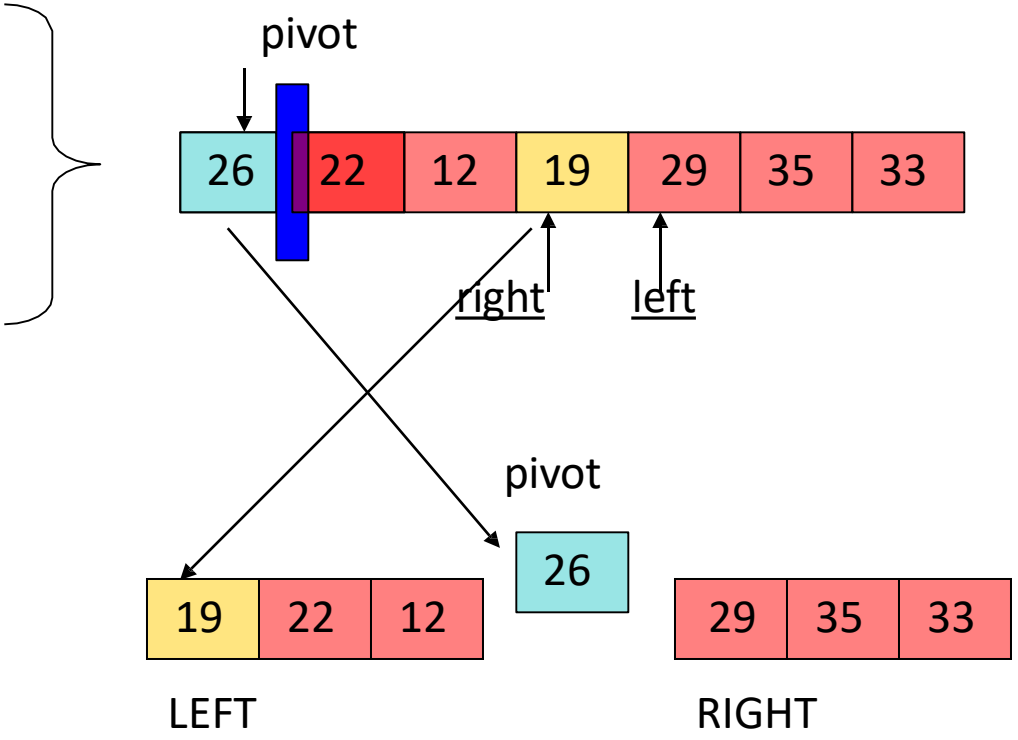
Exchange, increment left, decrement right.



QUICKSORT

Step 8:
When the left and right markers pass each other, we are done with the partition task.

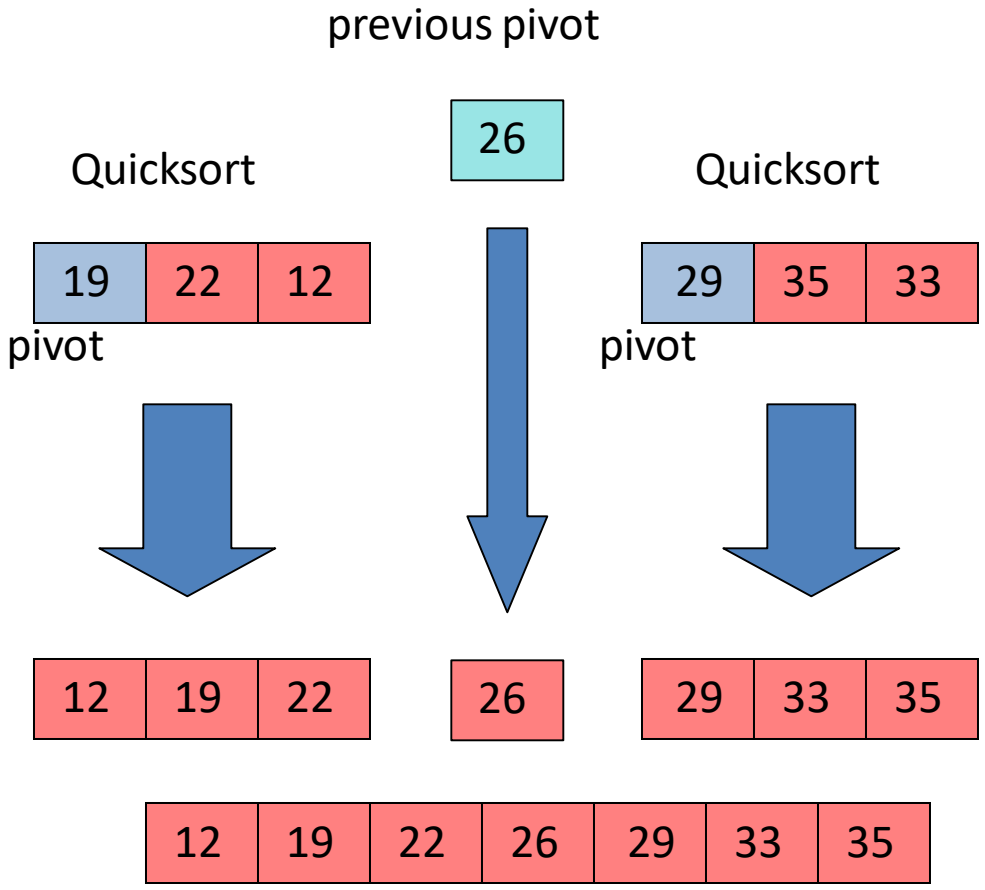
Swap the right with pivot.



QUICKSORT

Step 9:

Apply Quicksort to the LEFT and RIGHT groups, recursively.



Assemble parts when done

QUICKSORT

The partitioning of an array into two parts is O(n)

The number of recursive calls to Quicksort depends on how many times we can split the array into two groups.
On average this is O ($\log_2$ n)

The overall Quicksort efficiency is O(n) = n $\log_2$n

What is the worst-case efficiency?
Compare this to the worst case for the heapsort.

COMPARISON OF SORTING METHODS

		Time Complexity					
		Best	Worst	Avg.	Space	Stable	Comments
Comparison Sort	Bubble Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Yes	*For each pair of indices, swap the elements if they are out of order*
	Modified Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes	*At each Pass check if the Array is already sorted. Best Case-Array Already sorted*
	Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Yes	*Swap happens only when once in a Single pass*
	Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes	*Very small constant factor even if the complexity is O(n^2).* ***Best Case:*** *Array already sorted* ***Worst Case:*** *sorted in reverse order*
	Quick Sort	O(n.lg(n))	O(n^2)	O(n.lg(n))	O(1)	Yes	*Best Case: when pivot divide in 2 equal halves Worst Case: Array already sorted - 1/n-1 partition*
	Randomized Quick Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(1)	Yes	*Pivot chosen randomly*
	Merge Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(n)	Yes	*Best to sort linked-list (constant extra space). Best for very large number of elements which cannot fit in memory (External sorting)*
	Heap Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(1)	No	
Non-Comparison Sort	Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(n+2^k)	Yes	*k = Range of Numbers in the list*
	Radix Sort	O(n.k/s)	O(2^s.n.k/s)	O(n.k/s)	O(n)	No	
	Bucket Sort	O(n.k)	O(n^2.k)	O(n.k)	O(n.k)	Yes	

UNIT-2

LINEAR DATA STRUCTURES

Stacks: Primitive operations, implementation of stacks using Arrays, applications of stacks arithmetic expression conversion and evaluation.

Queues: Primitive operations; Implementation of queues using Array, applications of linear queue, circular queue and double ended queue (deque).

STACKS

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.

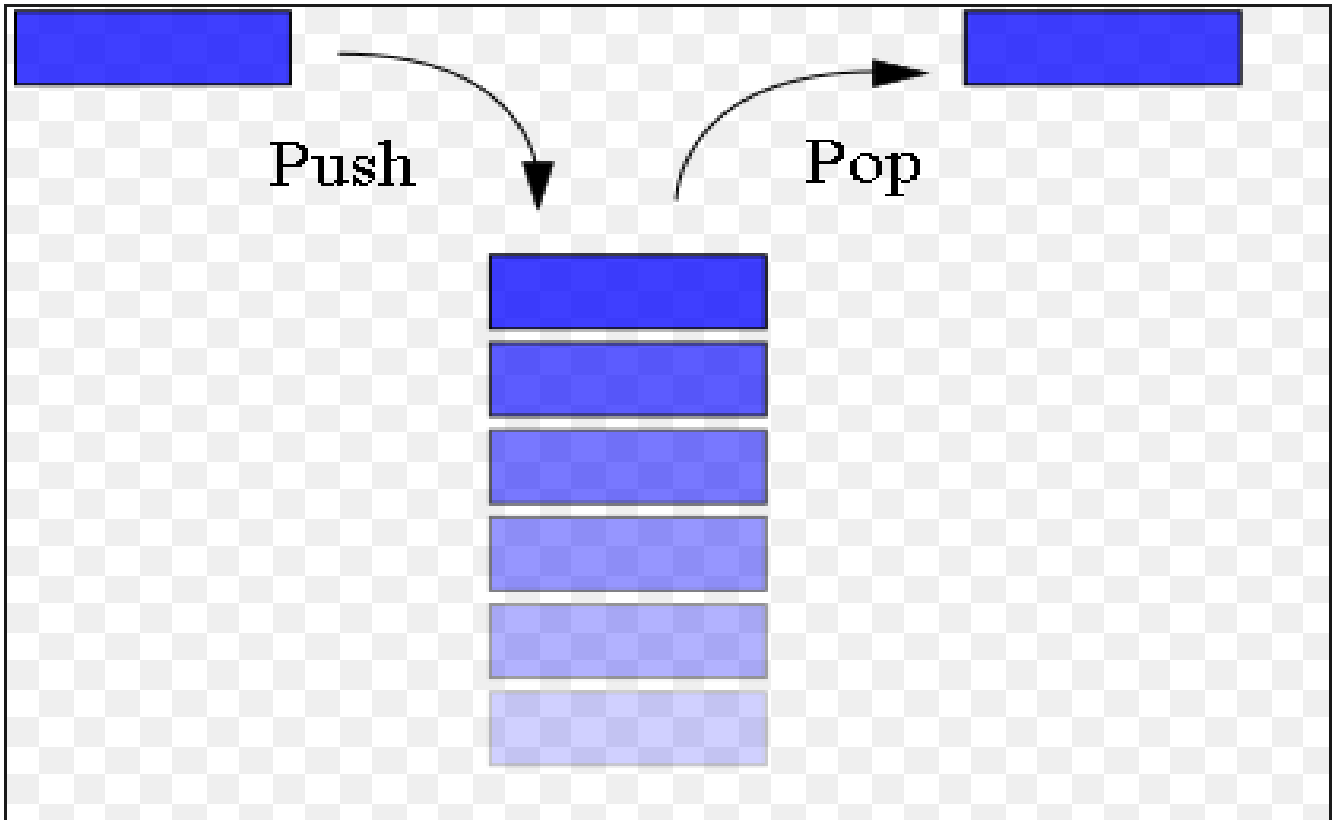
The end is called the top of stack.

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

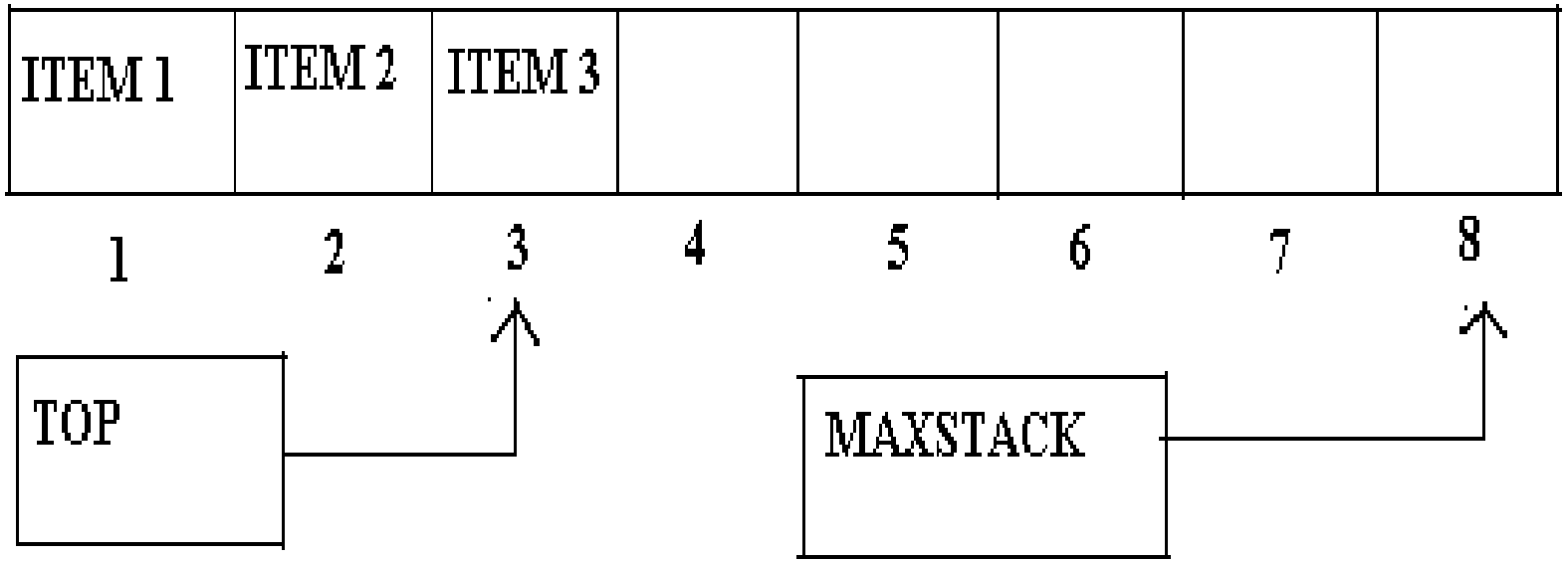
Stacks can be represented in the memory arrays by maintaining a linear array STACK and a pointer variable TOP which contains the location of top element.

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

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

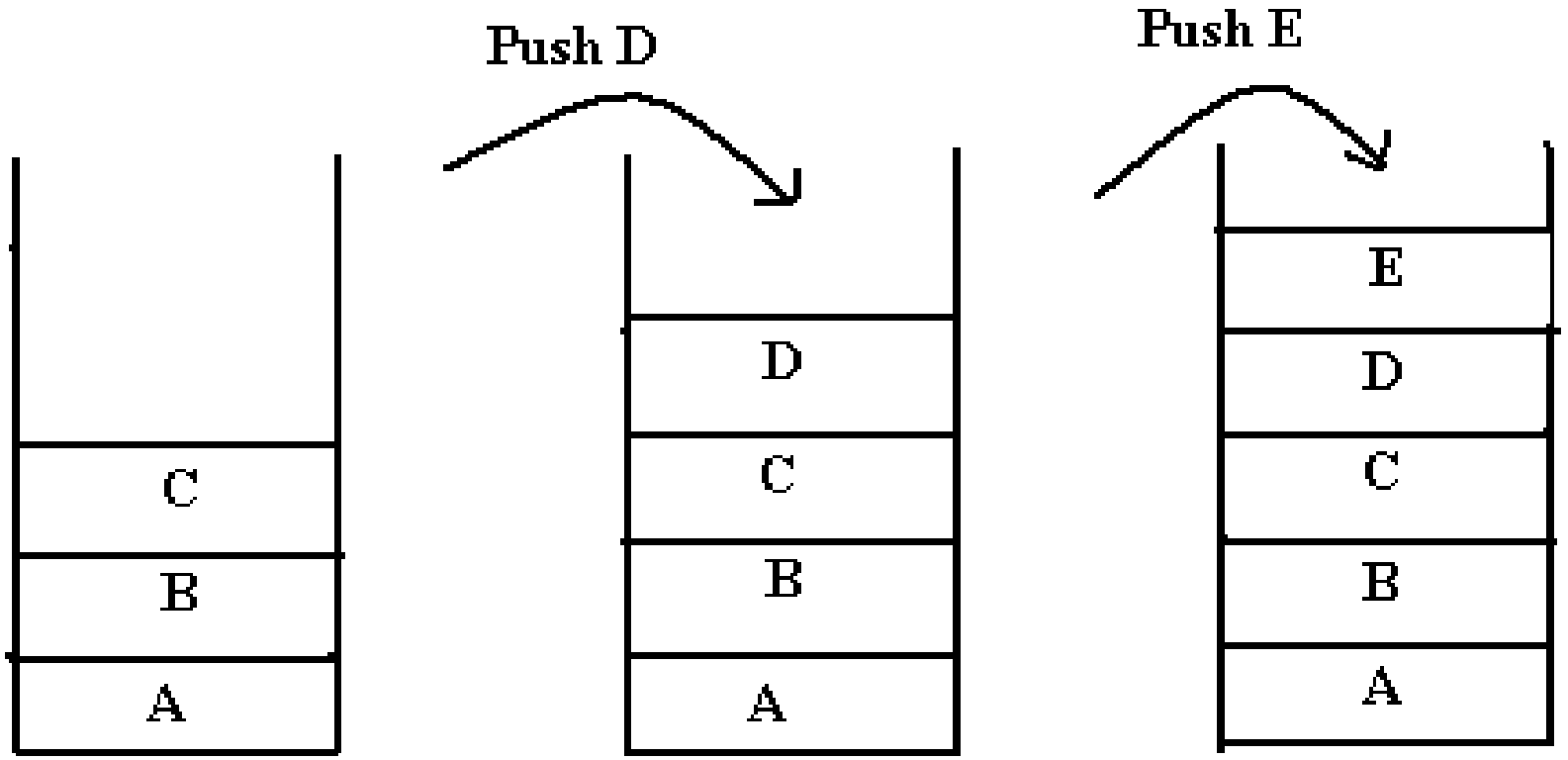
The operation of adding and removing an item in the stack can be implemented using the PUSH and POP functions.

STACK ARRAY REPRESENTATION



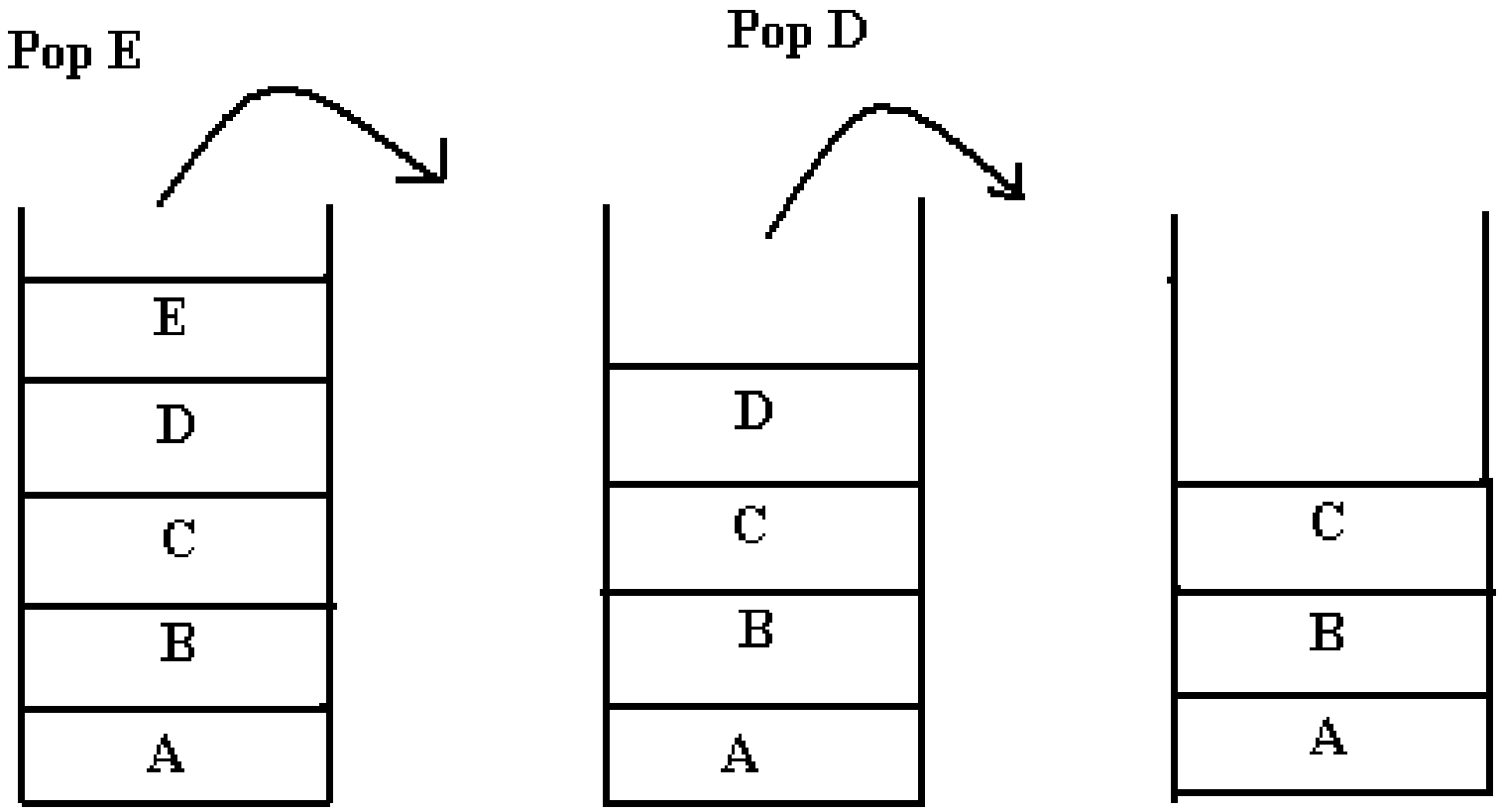
ARRAY REPRESENTATION OF A STACK

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.

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	A B +
12 + 60 – 23	12 60 + 23 –
(A + B)*(C – D)	A B + C D – *
A B * C – D + E/F	A B C*D – E F/+

RECURSION IMPLEMENTATION

If a procedure contains either a call statement to itself/to a second procedure that may eventually result in a cell 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

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

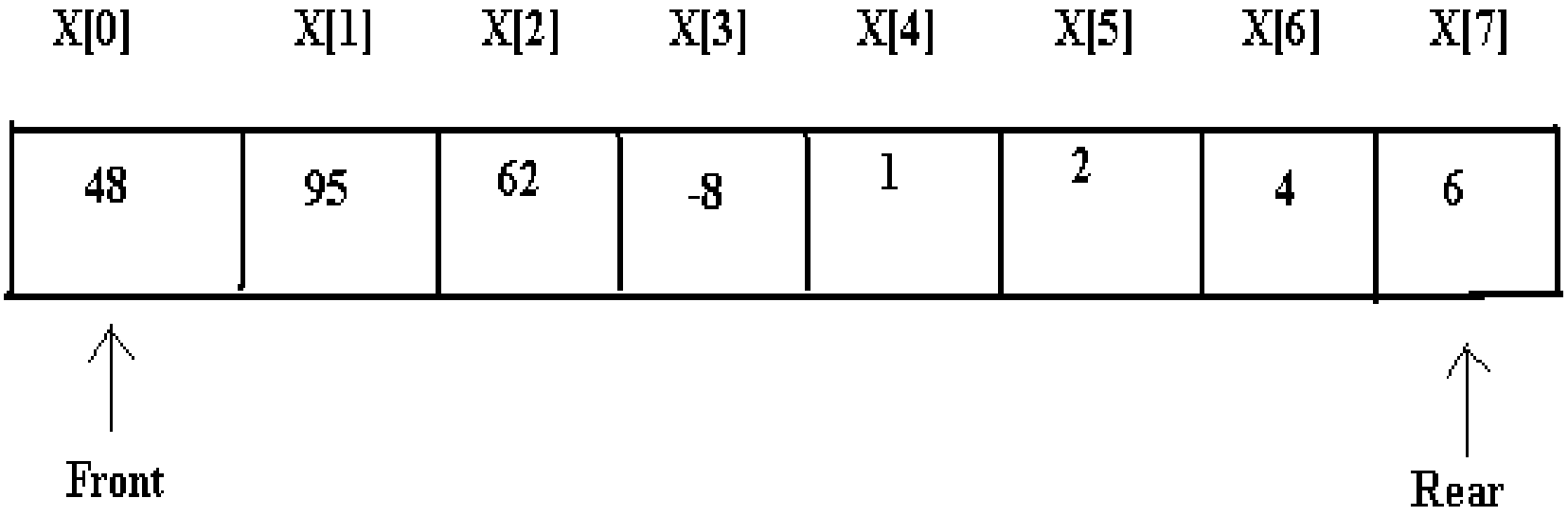
Array is a data structure that stores a fixed number of elements.

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.

QUEUE USING ARRAY



INSERTION AND DELETION 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).

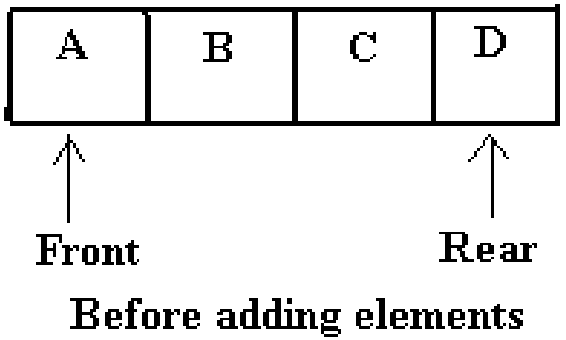
If the number of elements are already stored in the queue is reported to be full.

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

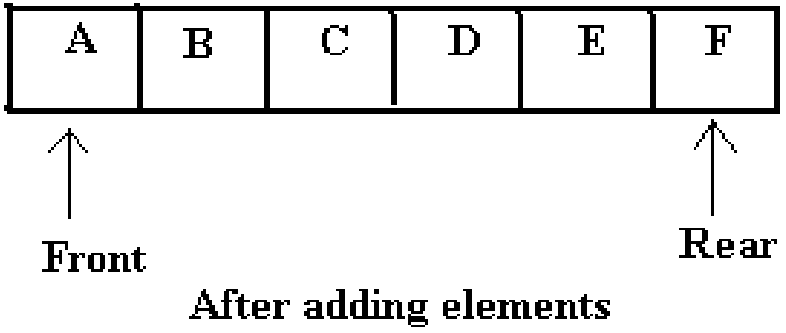
ADDING ELEMENTS IN AQUEUE

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.



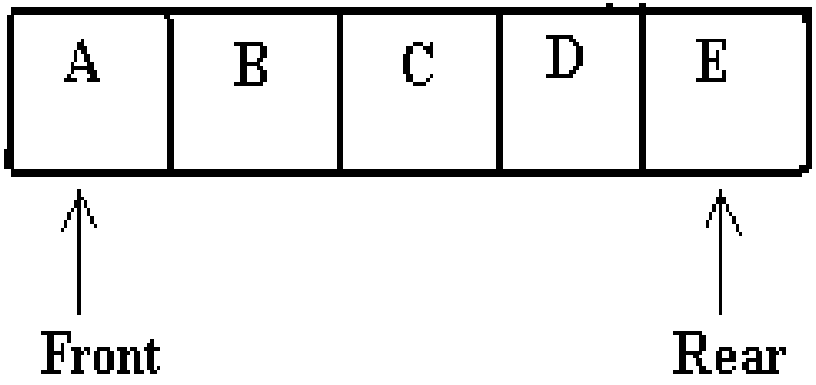
Before adding elements



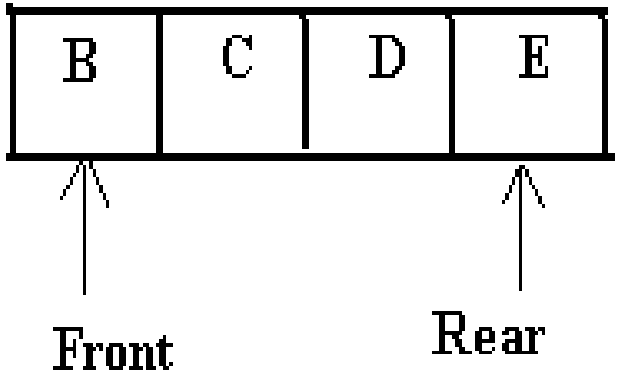
After adding elements

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

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.

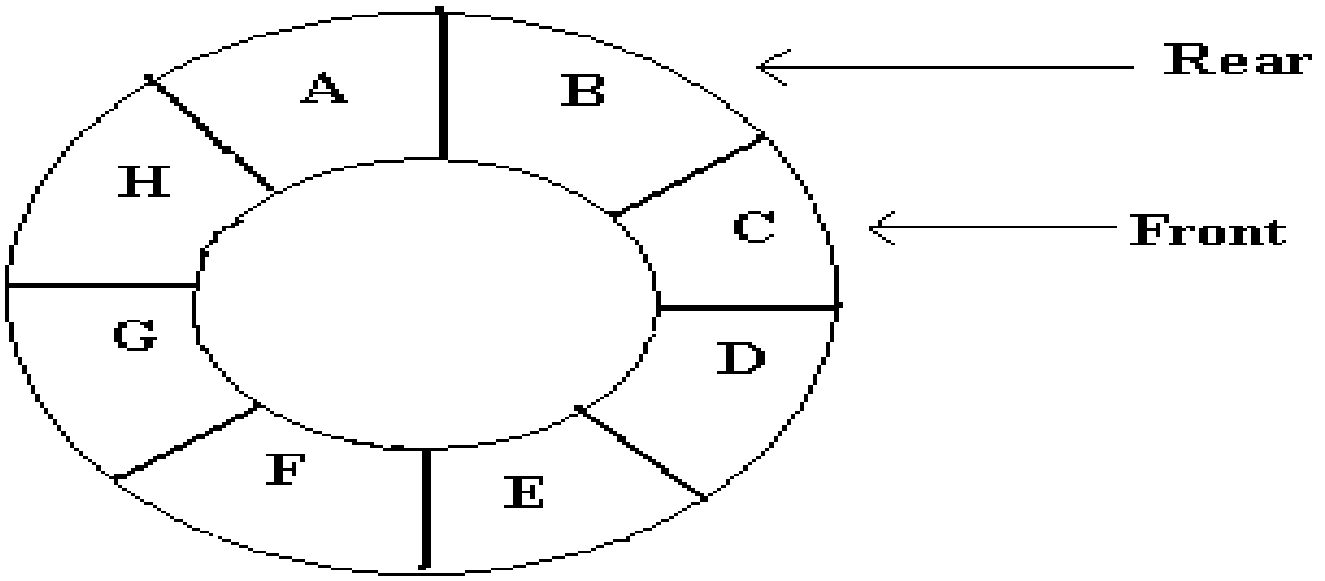
In the array implementation there is a possibility that the queue is reported full even though slots of the queue are empty.

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. rare 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 0^{Th} position.

ADDING ELEMENTS INTO CIRCULAR QUEUE

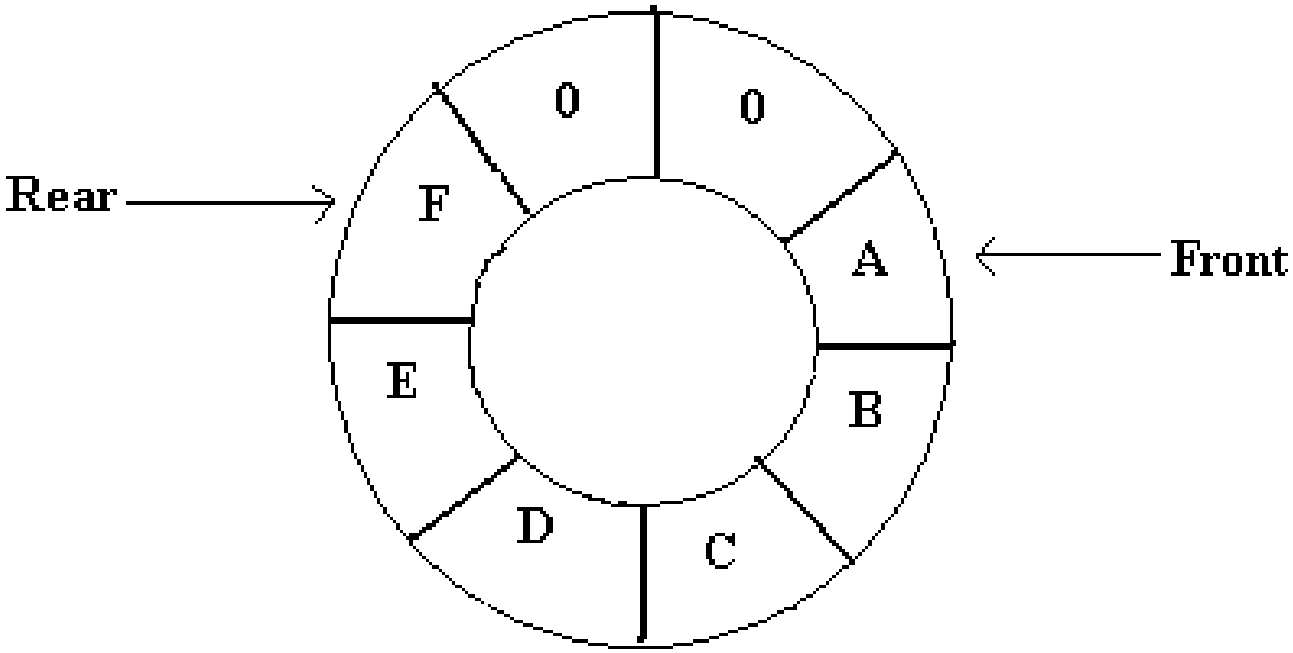
Some of the positions at the front end of the array might be empty .

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 :

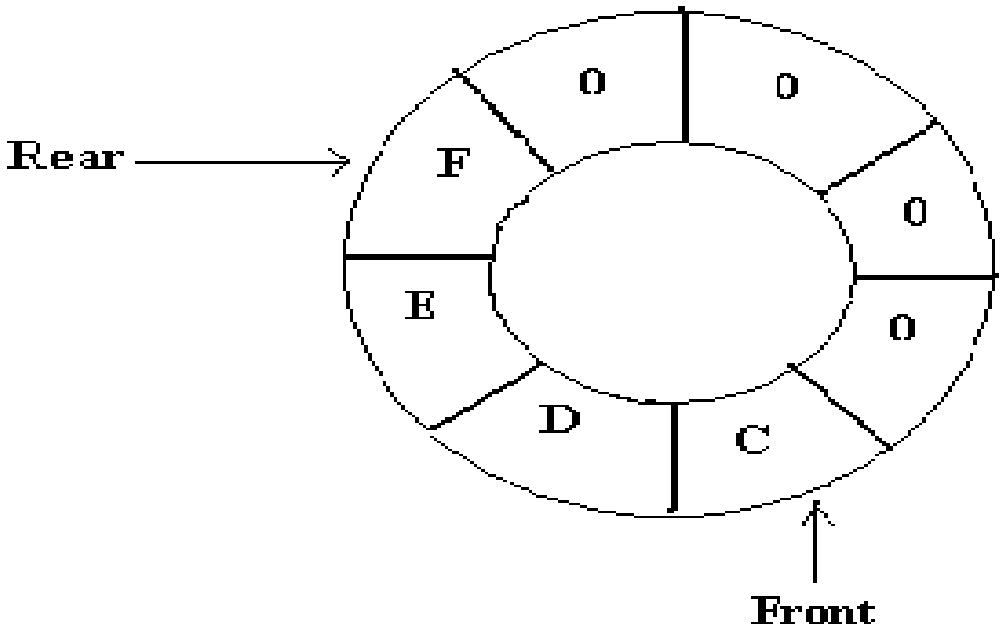
First it is checked whether the queue is empty or not . The elements at the front position will be deleted.

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 removes, 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

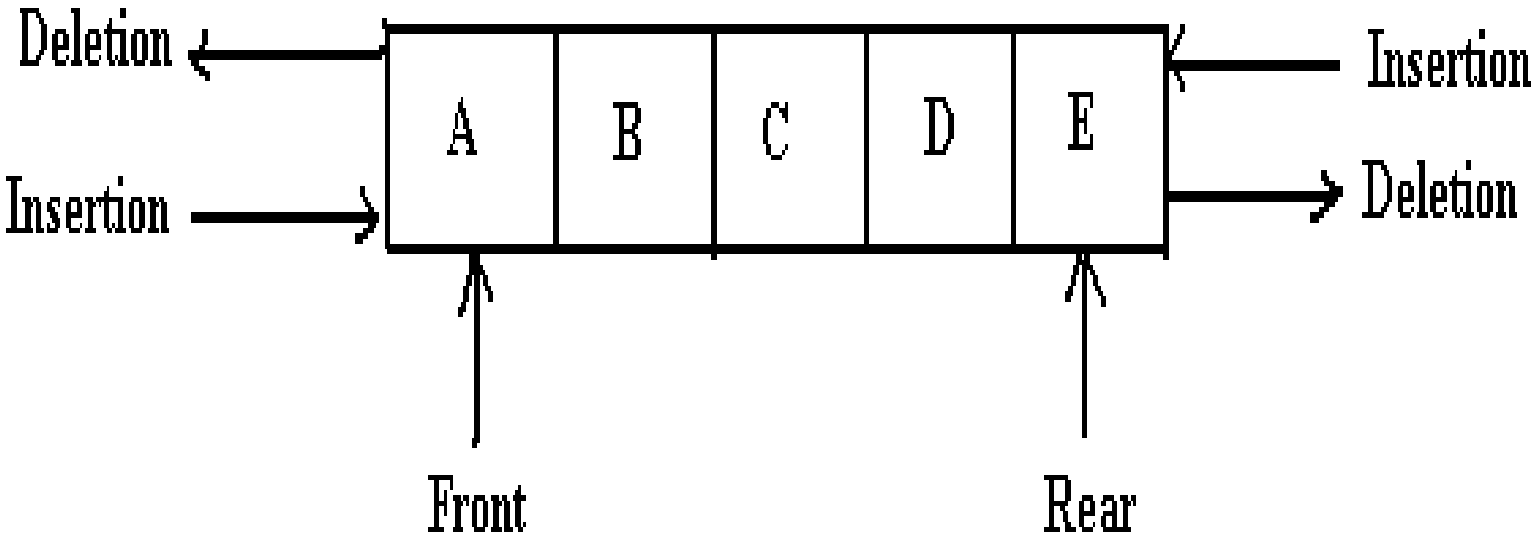
The output restricted deque is a deque which allows deletions at only one end of the list but allows insertions at both ends of the list.

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



Represention of a deque

UNIT-3

LINKED LISTS

Linked lists: Introduction, singly linked list, representation of a linked list in memory, operations on a single linked list.

Applications of linked lists: Polynomial representation and sparse matrix manipulation.

Types of linked lists: Circular linked lists, doubly linked lists; linked list representation and operations of Stack, linked list representation and operations of queue.

LINKED LISTS

LIST

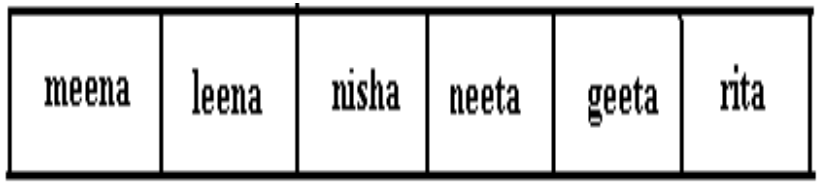
List is the collection of elements arranged in a sequential manner.

There are two representations

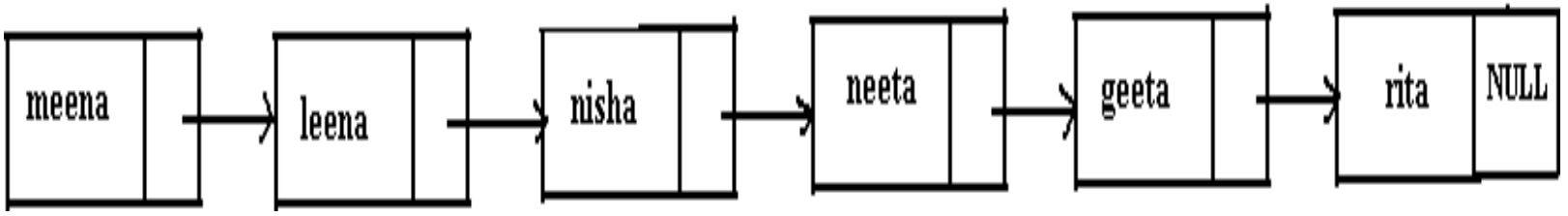
1) list of sequentially stored elements----using arrays

2) list of elements with associated pointers---using linked list.

LIST REPRESENTATION



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

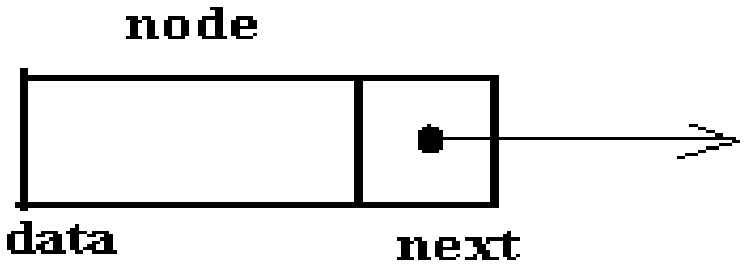
next(pointer): which holds a link to the next node in the list.

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 h 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**.

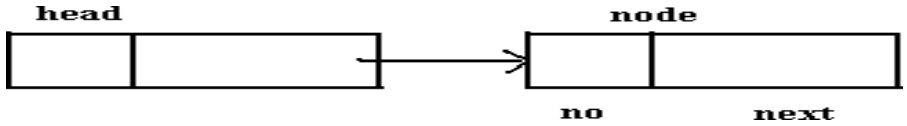
The head pointer is used to create and access unnamed nodes.

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(size_of(node));

head→next→number=20;

head→next→next=NULL;

INSERTING AN ELEMENT

Insertion is done in three ways:

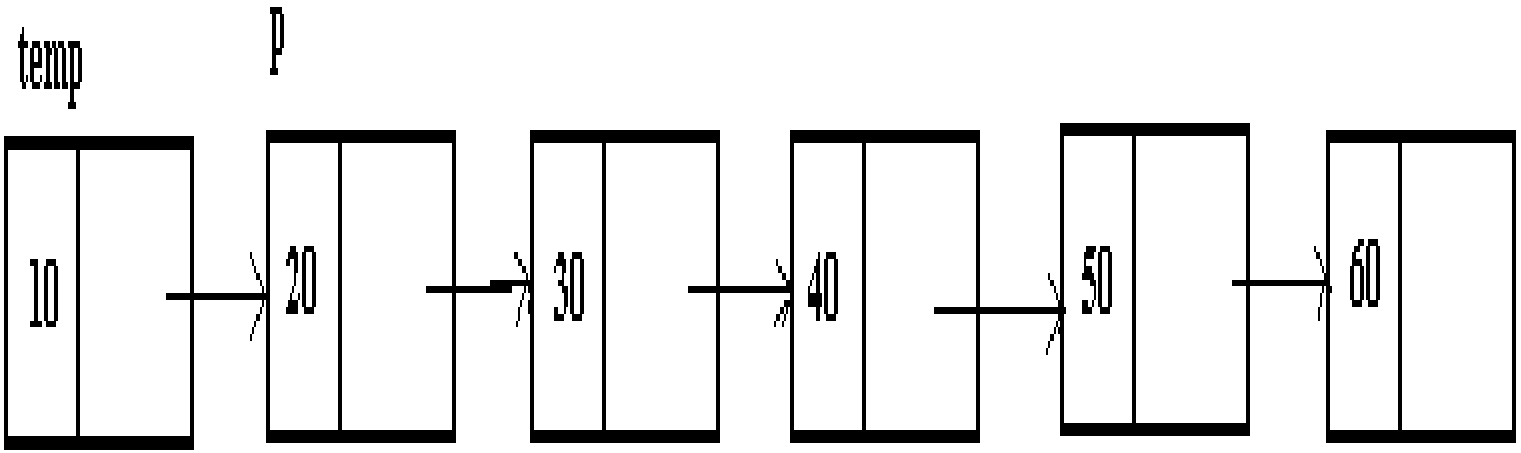
Insertion at the beginning of the list.

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

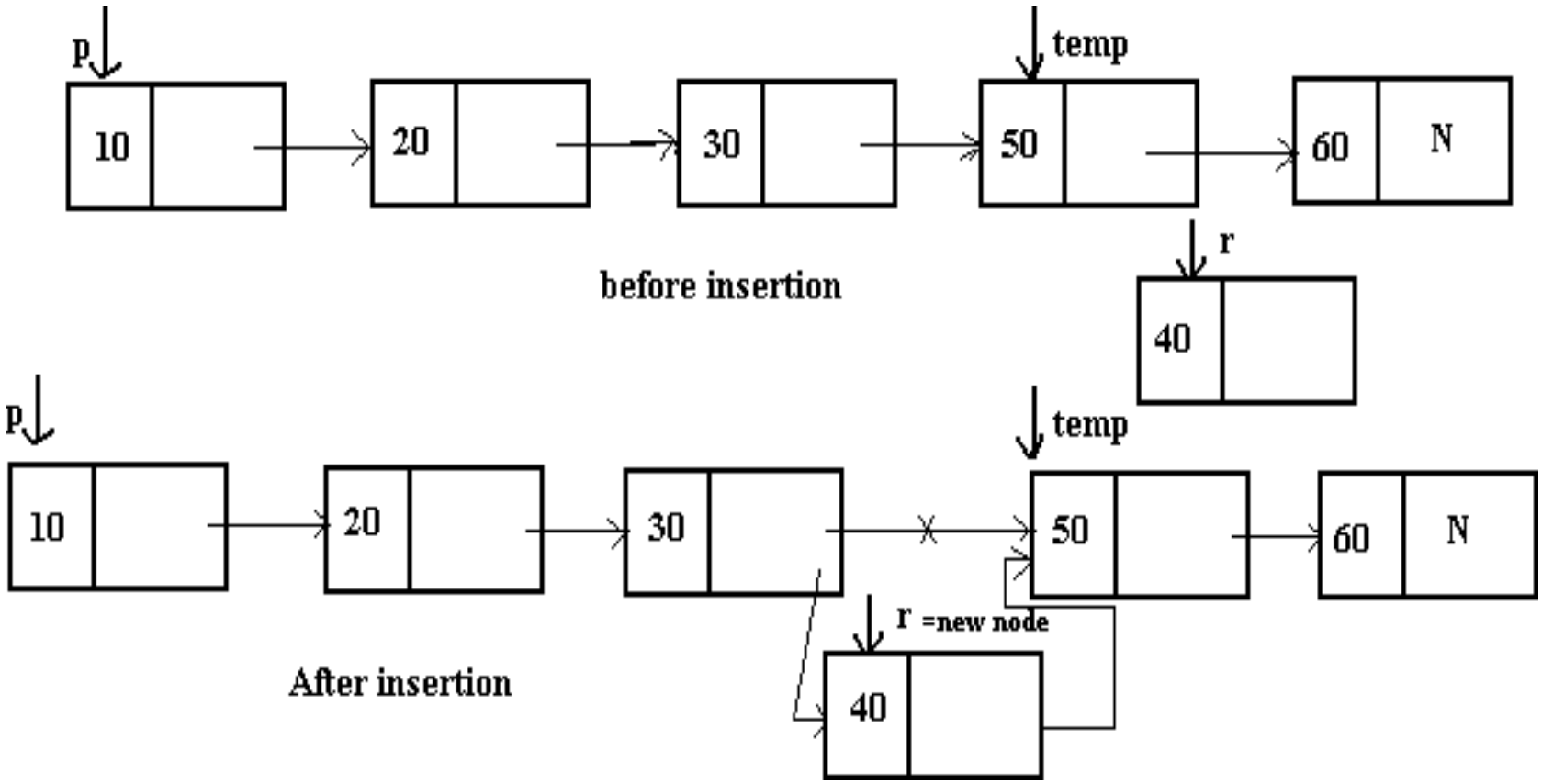
Insertion after any specified node:

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



INSERTING AN ELEMENT

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

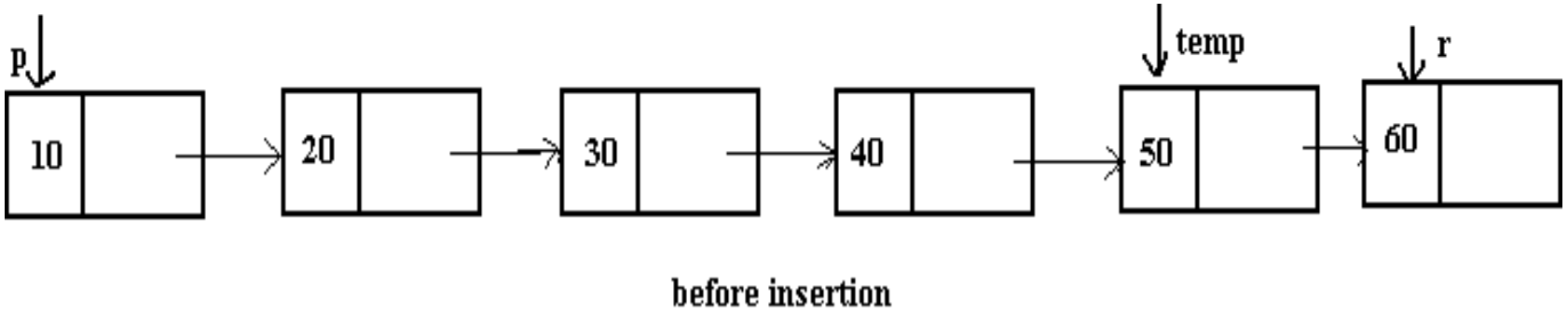
```
Void add_after(struct node *q, int loc, int no)
{
 struct node *temp, *r;
    int l;
    temp=q;/*skip to desire portion*/
    for(i=0;i<loc;i++)
          {
          temp=temp→next;
```

INSERTING AN ELEMENT

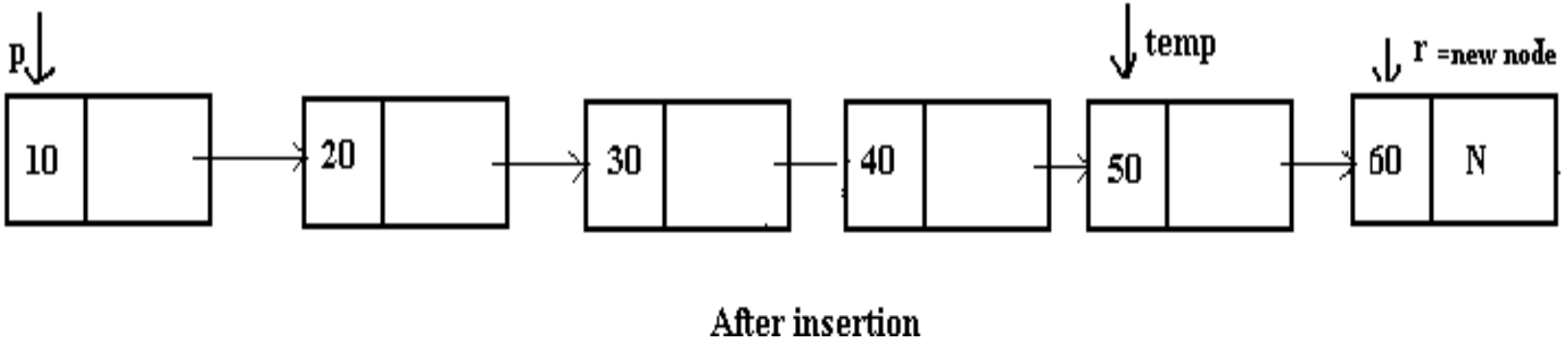
```
        if(temp==NULL)
        {
        printf("\n there are less than %d elements in list",loc);
         return;
        }
}                    ?/*insert new node*/
   r=malloc(sizeof(struct node));

   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;
```

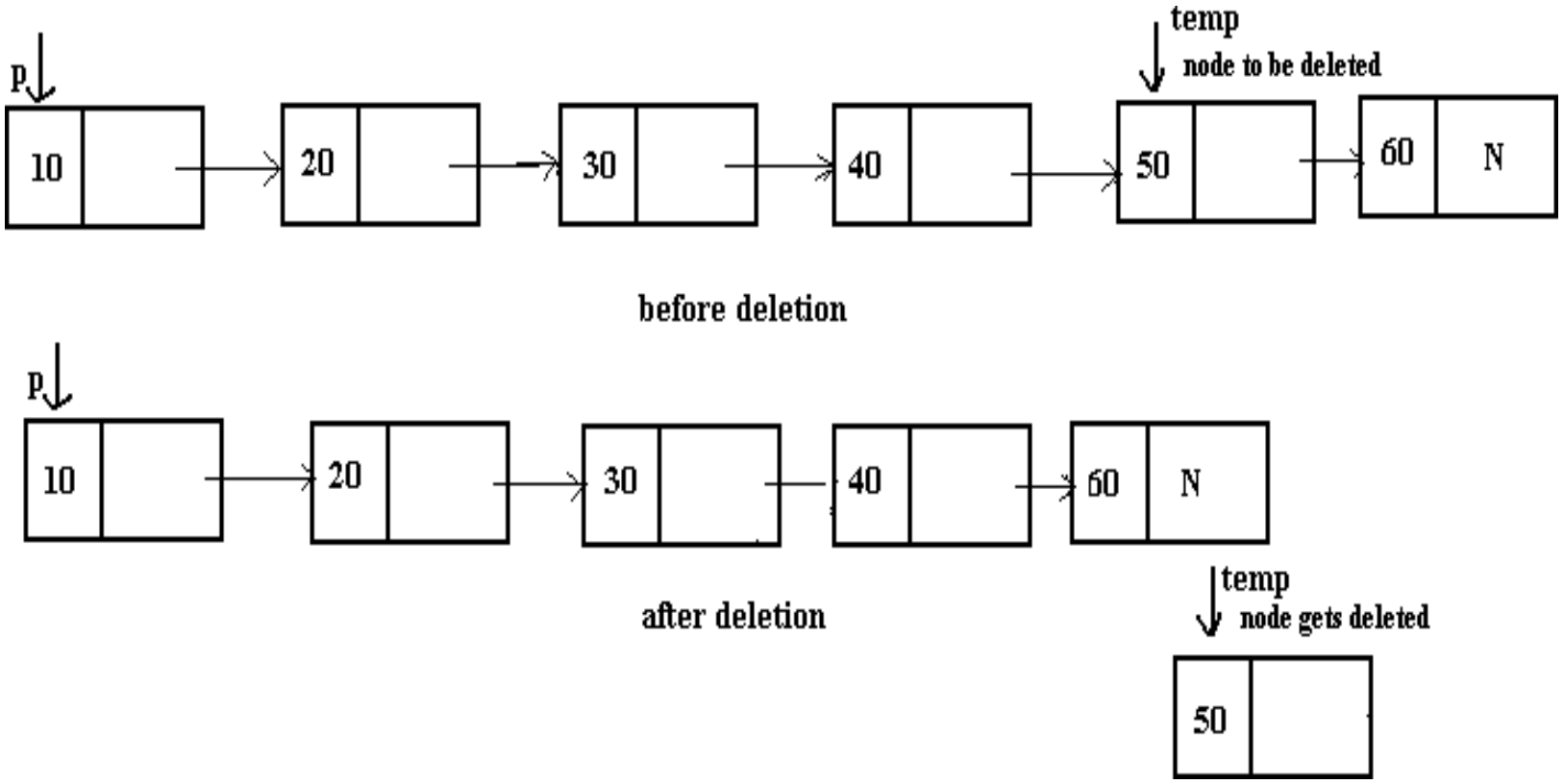
INSERTING AN ELEMENT

```
 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



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.

Merging the two linked list:

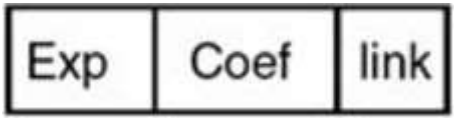
Merging two list pointed by two pointers into a third list.
While merging be ensure that the elements common to the lists appear only once in the third list.

APPLICATIONS OF LINKED LISTS

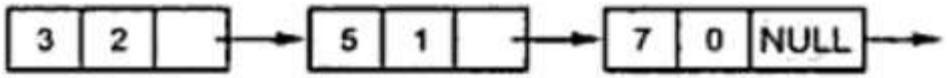
POLYNOMIAL REPRESENTATION

A Polynomial has mainly two fields. exponent and coefficient.

Node of a Polynomial:



For example 3x^2 + 5x + 7 will represent as follows.



POLYNOMIAL REPRESENTATION

In each node the exponent field will store the corresponding exponent and the coefficient field will store the corresponding coefficient. Link field points to the next item in the polynomial.

Addition and multiplication of polynomials is possible.

POLYNOMIAL REPRESENTATION

Addition of two Polynomials

Addition of two polynomials using linked list requires comparing the exponents, and wherever the exponents are found to be same, the coefficients are added up.

For terms with different exponents, the complete term is simply added to the result thereby making it a part of addition result.

POLYNOMIAL REPRESENTATION

Multiplication of two Polynomials

Multiplication of two polynomials however requires manipulation of each node such that the exponents are added up and the coefficients are multiplied.

After each term of first polynomial is operated upon with each term of the second polynomial, then the result has to be added up by comparing the exponents and adding the coefficients for similar exponents and including terms as such with dissimilar exponents in the result.

SPARSE MATRIX MANIPULATION

In computer programming, a matrix can be defined with a 2-dimensional array.

Any array with 'm' columns and 'n' rows represents a mXn matrix.

There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values.

Such matrix is known as sparse matrix.

SPARSE MATRIX MANIPULATION

When a sparse matrix is represented with 2-dimensional array, we waste lot of space to represent that matrix. For example, consider a matrix of size 100 X 100 containing only 10 non-zero elements. In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of matrix are filled with zero. That means, totally we allocate 100 X 100 X 2 = 20000 bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times.

SPARSE MATRIX MANIPULATION

A sparse matrix can be represented by using TWO representations, those are as follows...

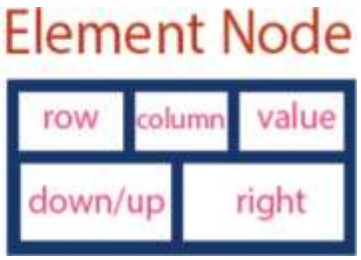
Triplet Representation

Linked Representation

SPARSE MATRIX MANIPULATION

Linked Representation

In linked representation, we use linked list data structure to represent a sparse matrix. In this linked list, we use two different nodes namely **header node** and **element node**. Header node consists of three fields and element node consists of five fields as shown in the image...



TYPES OF LINKED LISTS

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

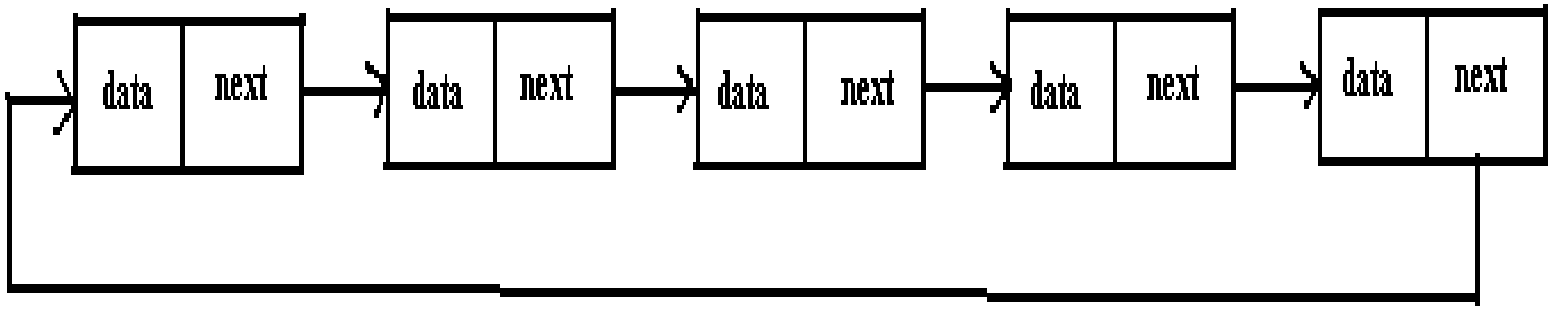
The structure defined for circular linked list

```
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.

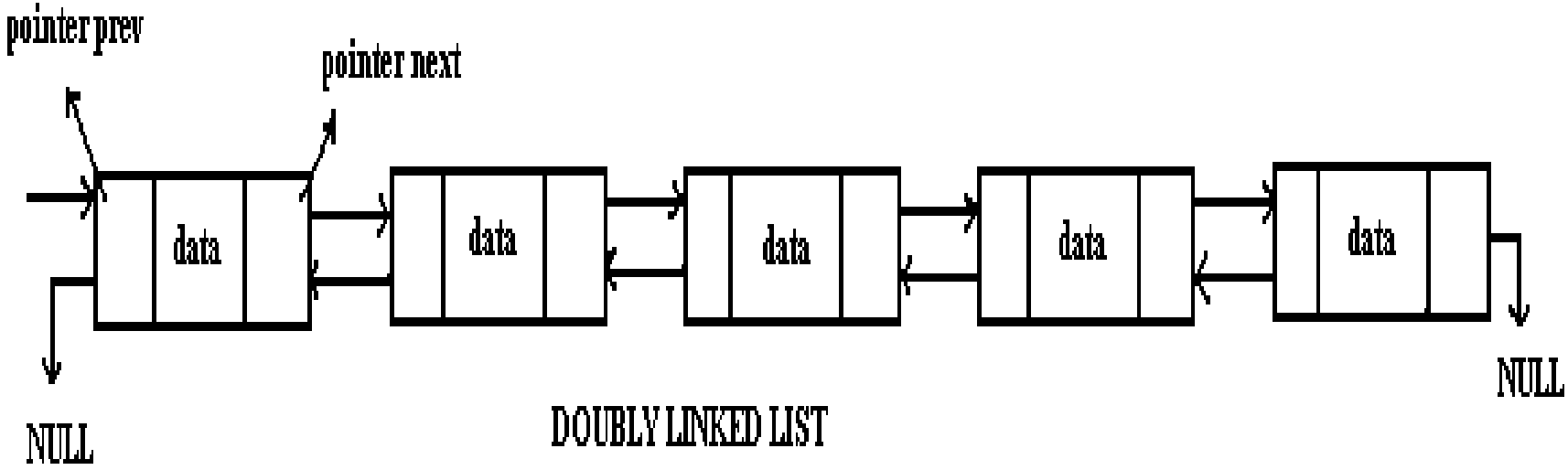
It can traverse in two directions:

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 a 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;
}
}
```

STACK USING LINKED LIST

The major problem with the stack implemented using array is, it works only for fixed number of data values.

A stack data structure can be implemented by using linked list data structure.

The stack implemented using linked list can work for unlimited number of values.

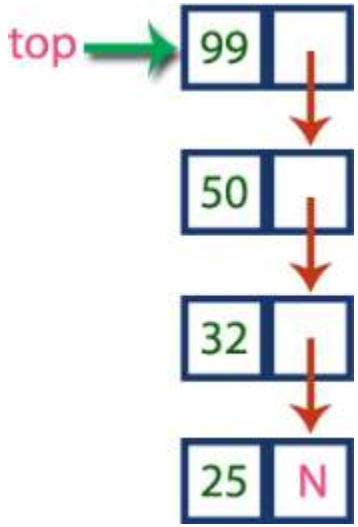
STACK USING LINKED LIST

In linked list implementation of a stack, every new element is inserted as '**top**' element.

Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its next node in the list.

The **next** field of the first element must be always **NULL**.

STACK USING LINKED LIST



STACK USING LINKED LIST

To implement stack using linked list, we need to set the following things before implementing actual operations.

Step 1: Include all the **header files** which are used in the program. And declare all the **user defined functions**.

Step 2: Define a '**Node**' structure with two members **data** and **next**.

Step 3: Define a **Node** pointer '**top**' and set it to **NULL**.

Step 4: Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

STACK USING LINKED LIST

Operations

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

Step 1: Create a **newNode** with given value.

Step 2: Check whether stack is **Empty** (**top** == **NULL**)

Step 3: If it is **Empty**, then set **newNode** → **next** = **NULL**.

Step 4: If it is **Not Empty**, then set **newNode** → **next** = **top**.

Step 5: Finally, set **top** = **newNode**.

STACK USING LINKED LIST

Operations

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

Step 1: Check whether **stack** is **Empty** (**top == NULL**).

Step 2: If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function

Step 3: If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.

Step 4: Then set '**top = top → next**'.

Step 5: Finally, delete '**temp**' (**free(temp)**).

STACK USING LINKED LIST

Operations

display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

Step 1: Check whether stack is **Empty** (**top** == **NULL**).

Step 2: If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.

Step 3: If it is **Not Empty**, then define a Node pointer '**temp**' and initialize with **top**.

Step 4: Display '**temp → data --->**' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack (**temp → next** != **NULL**).

Step 4: Finally! Display '**temp → data ---> NULL**'.

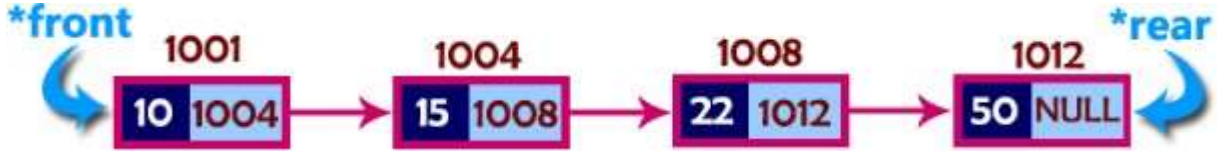
QUEUE USING LINKED LIST

A queue data structure can be implemented using linked list data structure. The queue which is implemented using linked list can work for unlimited number of values.

The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

QUEUE USING LINKED LIST



QUEUE USING LINKED LIST

To implement queue using linked list, we need to set the following things before implementing actual operations.

Step 1: Include all the **header files** which are used in the program. And declare all the **user defined functions**.

Step 2: Define a '**Node**' structure with two members **data** and **next**.

Step 3: Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.

Step 4: Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

QUEUE USING LINKED LIST

Operations

enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...

Step 1: Create a **newNode** with given value and set '**newNode → next**' to **NULL**.

Step 2: Check whether queue is **Empty** (**rear** == **NULL**)

Step 3: If it is **Empty** then,
set **front** = **newNode** and **rear** = **newNode**.

Step 4: If it is **Not Empty** then, set **rear →**
next = **newNode** and **rear** = **newNode**.

QUEUE USING LINKED LIST

Operations

deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue...

Step 1: Check whether **queue** is **Empty** (**front == NULL**).

Step 2: If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function

Step 3: If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.

Step 4: Then set '**front** = **front** → **next**' and delete '**temp**' (**free(temp)**).

QUEUE USING LINKED LIST

- Operations
- **display() - Displaying the elements of Queue**
- We can use the following steps to display the elements (nodes) of a queue...
- **Step 1:** Check whether queue is **Empty** (**front** == **NULL**).
- **Step 2:** If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.
- **Step 3:** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.
- **Step 4:** Display '**temp → data** --->' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp → next** != **NULL**).
- **Step 4:** Finally! Display '**temp → data** ---> **NULL**'.

UNIT-4

NON LINEAR DATA STRUCTURES

Trees: Basic concept, binary tree, binary tree representation, array and linked representations, binary tree traversal, binary search tree, tree variants, application of trees.

Graphs: Basic concept, graph terminology, graph implementation, graph traversals, Application of graphs, Priority Queue.

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>=0 disjoint sets
T1, ...,Tn, where each of these sets is a tree.

We call T1, ..., Tn the subtrees of the root.

REPRESENTATION OF TREE

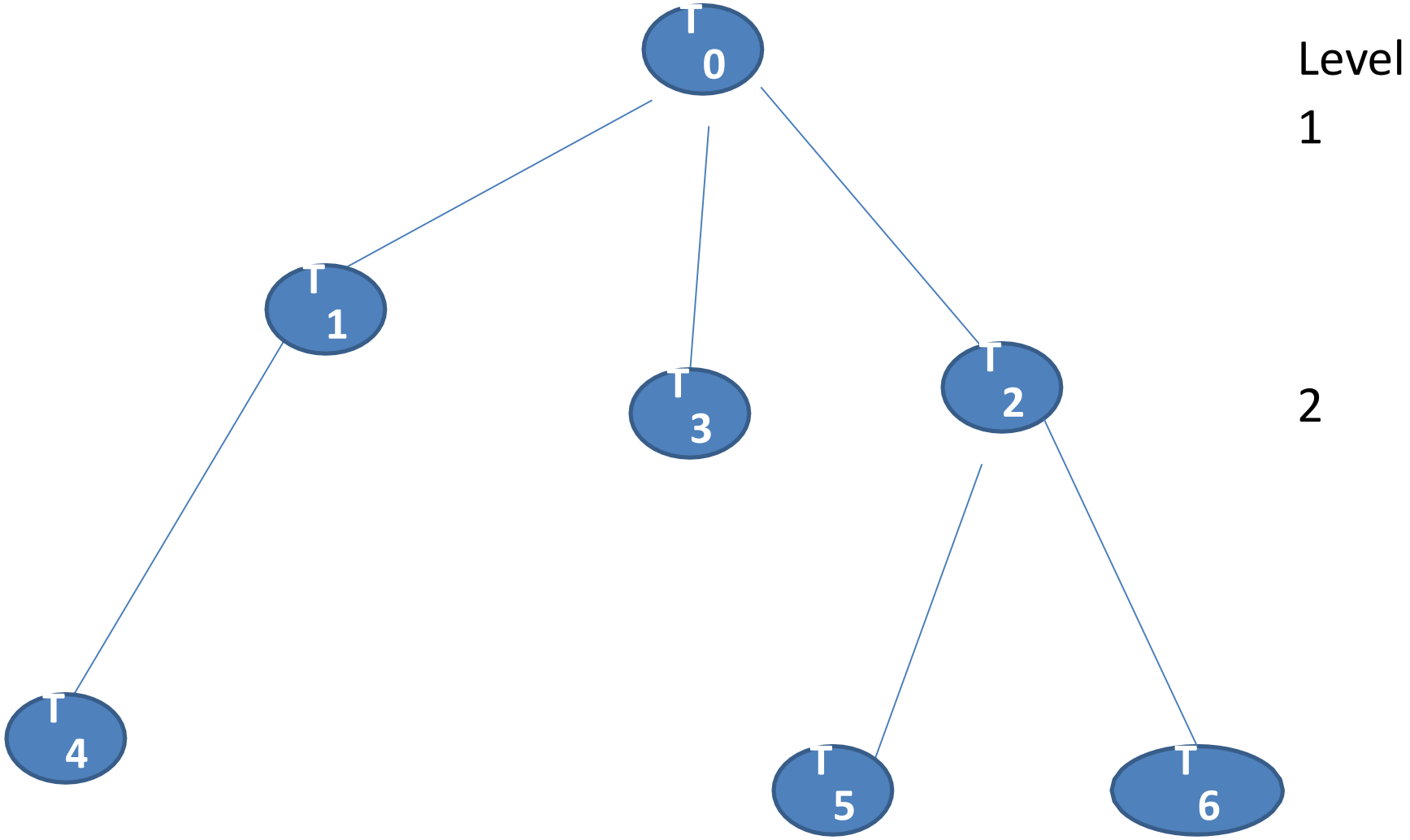


Fig.Tree 1

TERMINOLOGY

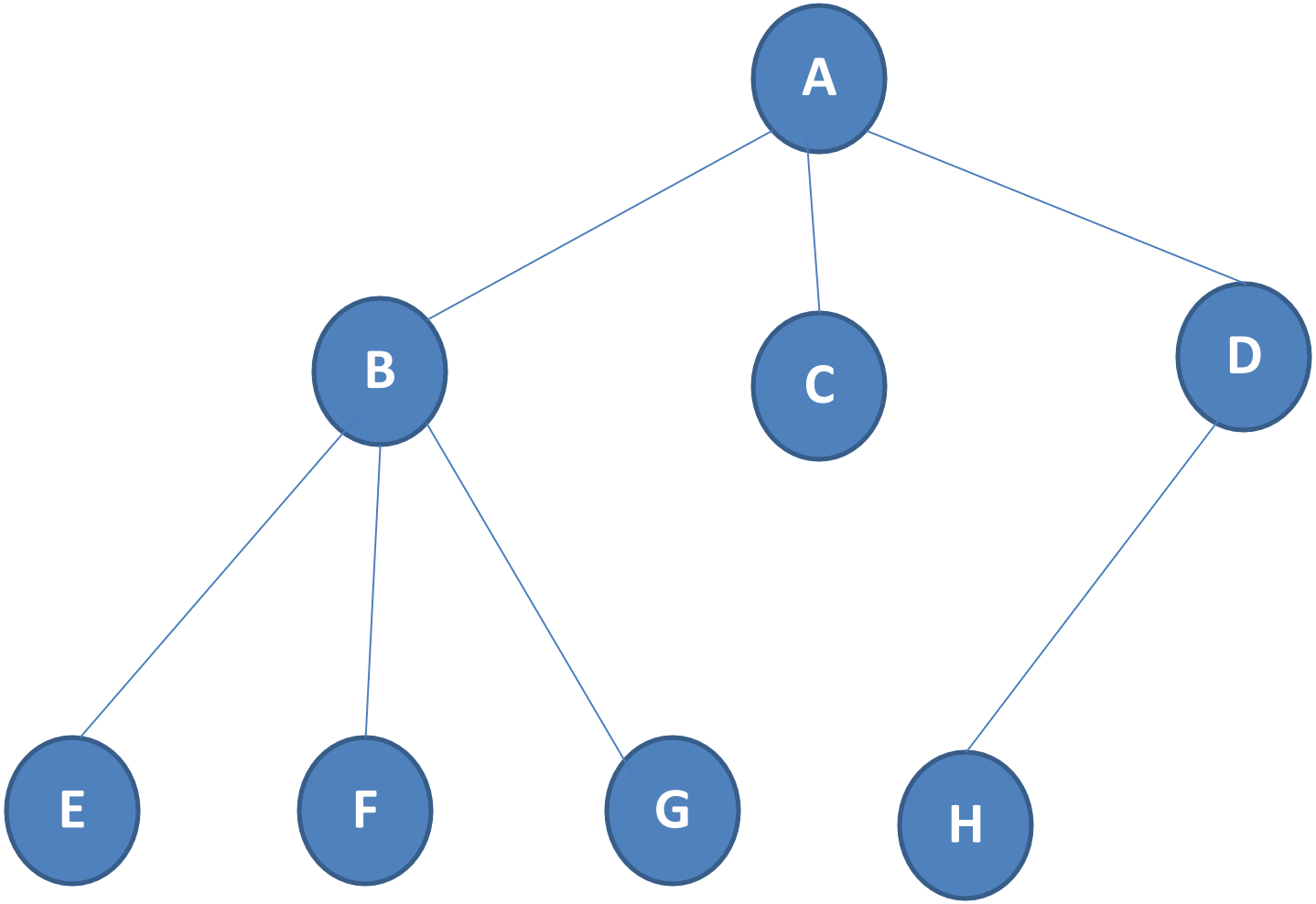


Fig.Tree 2

➢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

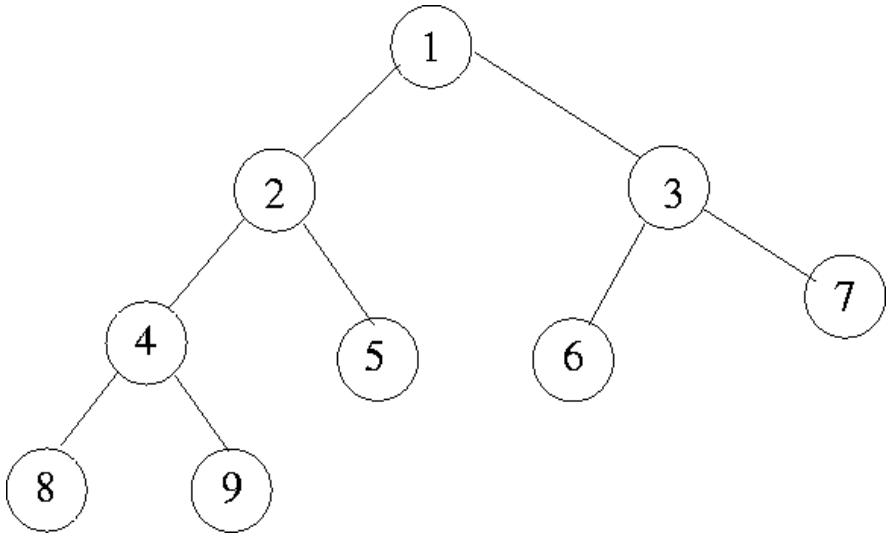
- The degree of A is 3; the degree of C is 1.

- 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.

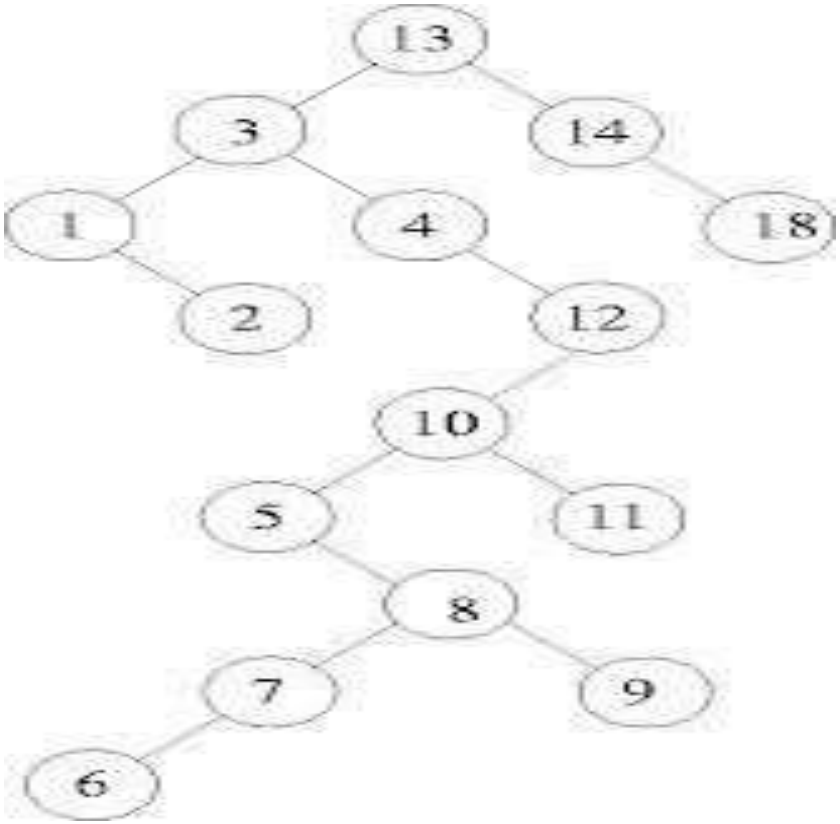
BINARY TREES

- 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.

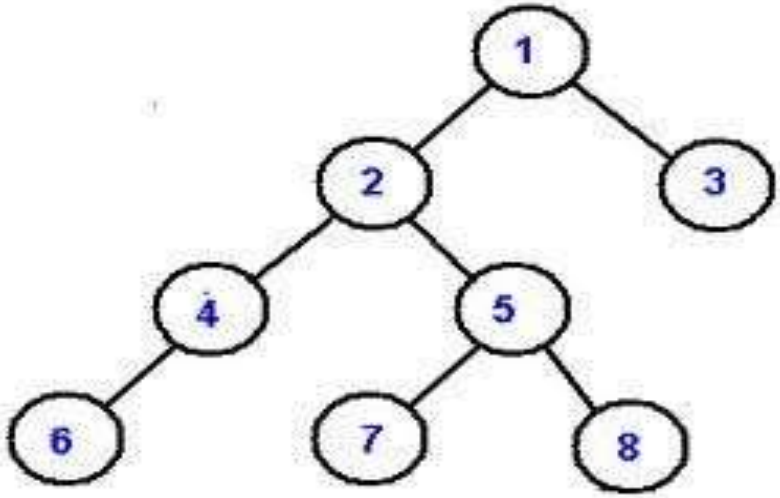
BINARY TREES



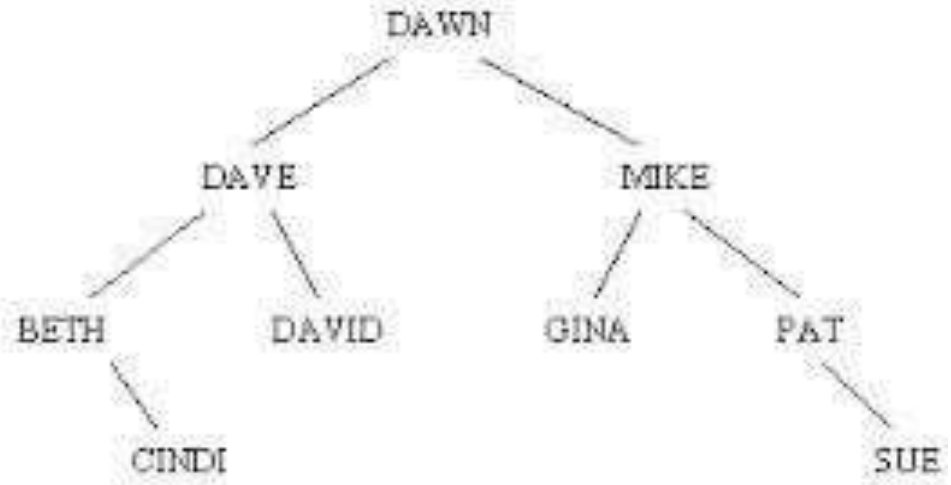
BINARY TREES



BINARY TREES



BINARY TREES



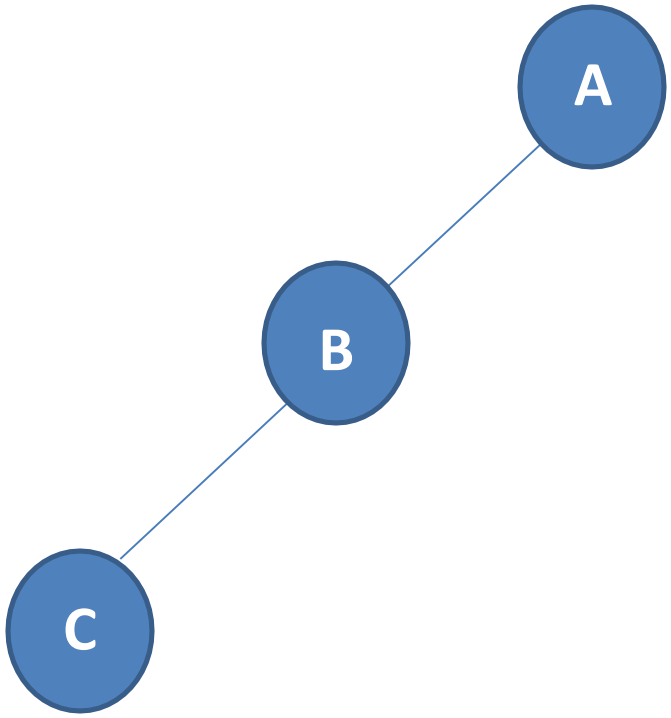
TYPES OF BINARY TREES

There are three types of binary trees

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

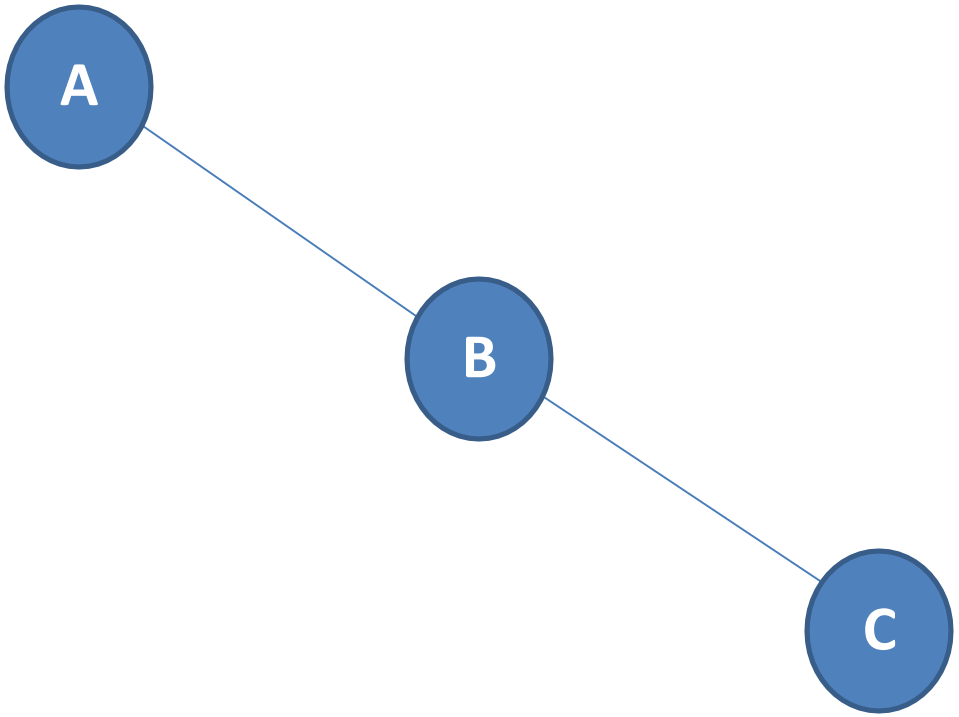
LEFT SKEWED BINARY TREE

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



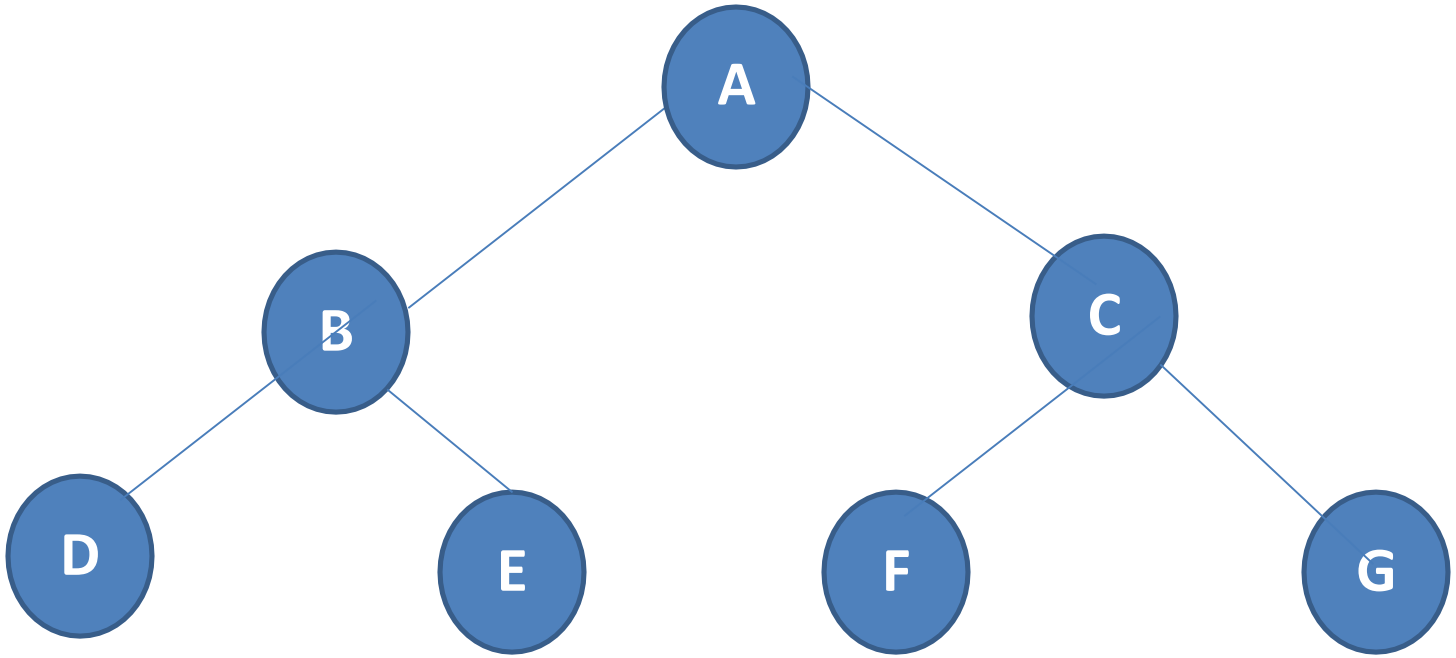
RIGHT SKEWED BINARY TREE

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

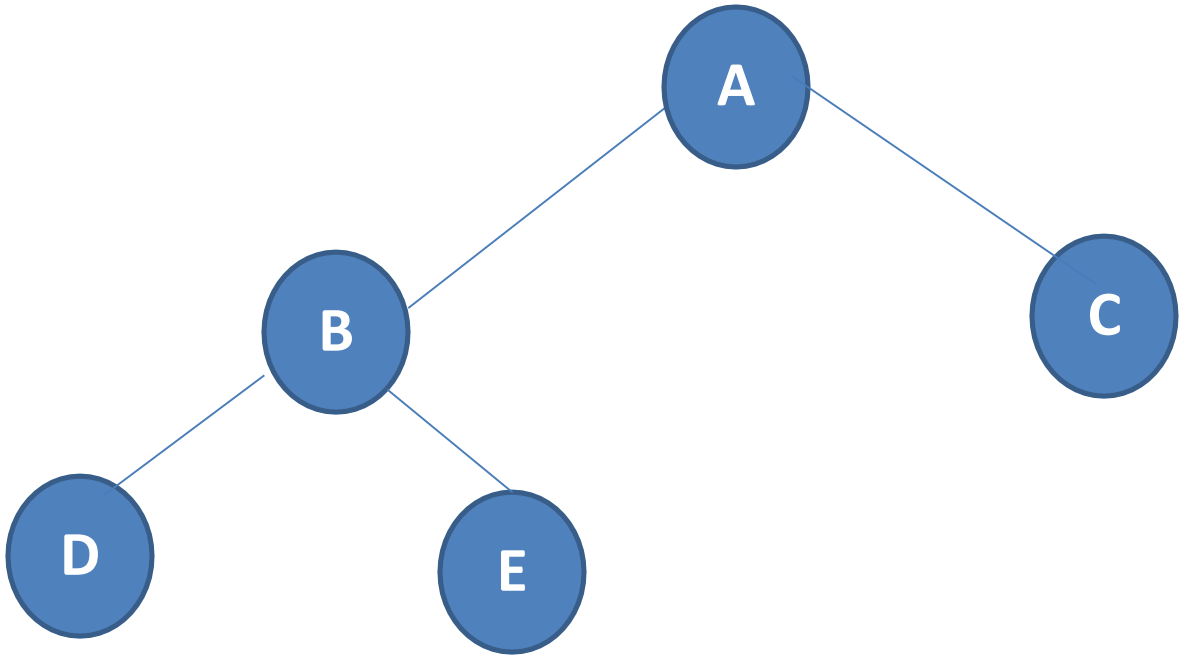


COMPLETE BINARY TREE

- 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,twonodes 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 contains exactly 2^l nodes at each level l,where l is from 0 to d.



FULL BINARY TREE



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*==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>=1.
- The maximum nubmer of nodes in a binary tree of depth k is 2^k-1, k>=1.

Prove by induction.

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

BINARY TREE REPRESENTATION

- Sequential(Arrays) representation
- Linked representation

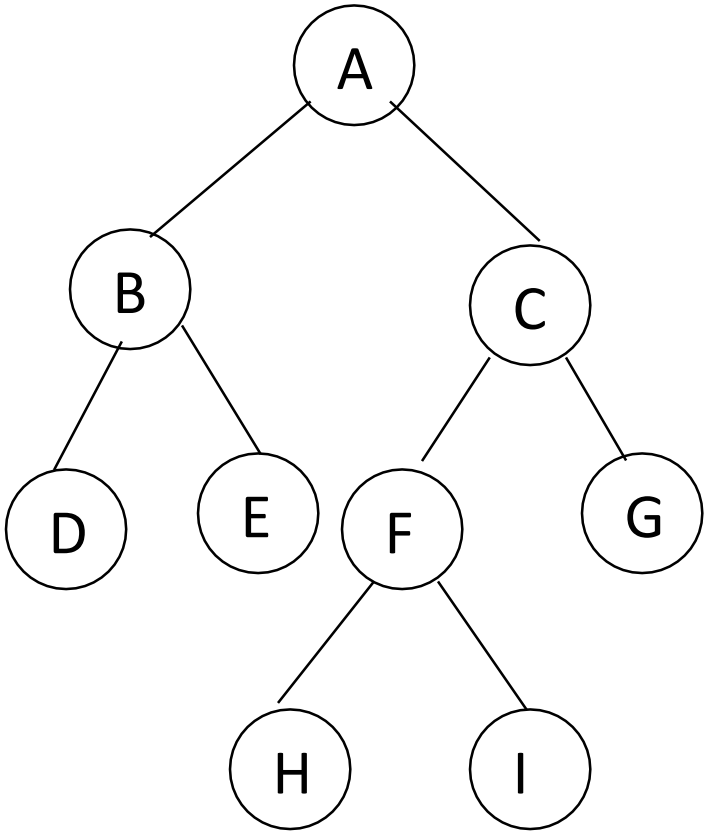
ARRAY REPRESENTATION OF BINARY TREE

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

i)The root of the tree is stored in tree[0].

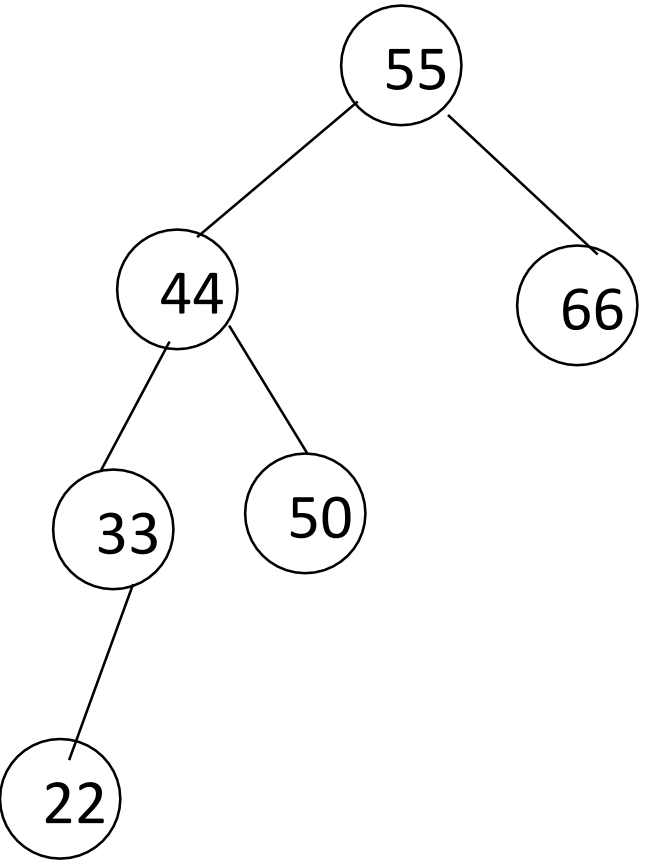
ii)if a node occupies tree[i],then its left child is stored in tree[2*i+1],its right child is stored in tree[2*i+2],and the parent is stored in tree[(i-1)/2].

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

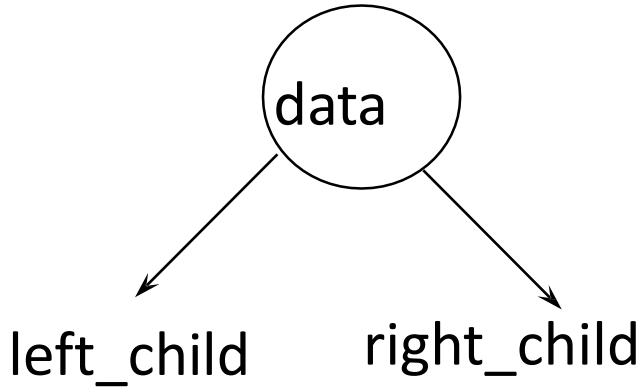
The only advantage with this type of representation is that the direct access to any node can be possible and finding the parent or left right children of any particular node is fast because of the random access.

DISADVANTAGES OF SEQUENTIALREPRESENTATION

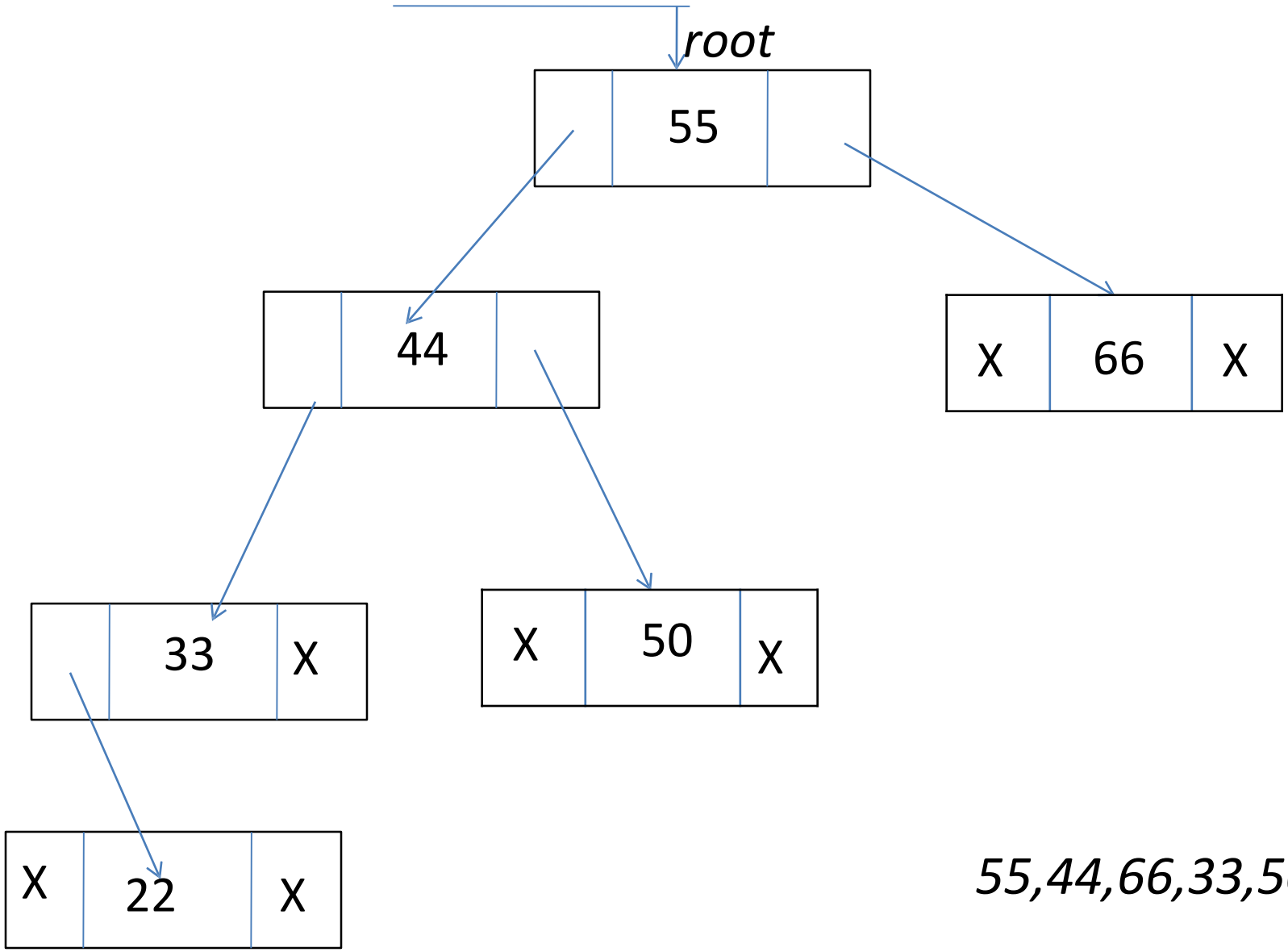
- The major disadvantage with this type of representation is wastage of memory.
- The maximum depth of the tree has to be fixed.
- The insertions and deletion of any node in the tree will be costlier as other nodes has to be adjusted at appropraite positions so that the meaning of binary tree can be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

LINKED REPRESENTATION

```
struct node
 {
 int data;
struct node * left_child, *right_child;
};
```



Linked Representation



55,44,66,33,50,22

ADVANTAGES OF LINKED REPRESENTATION

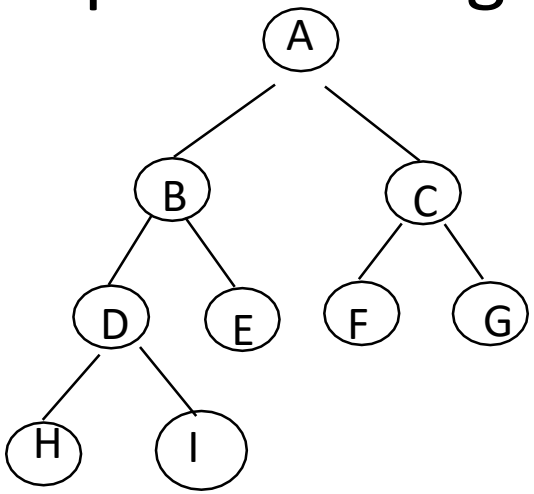
- This representation is superior to our representation as there is no wastage of memory.
- Insertions and deletions which are the most common operations can be done without moving the other nodes.

DISADVANTAGES OF LINKED REPRESENTATION

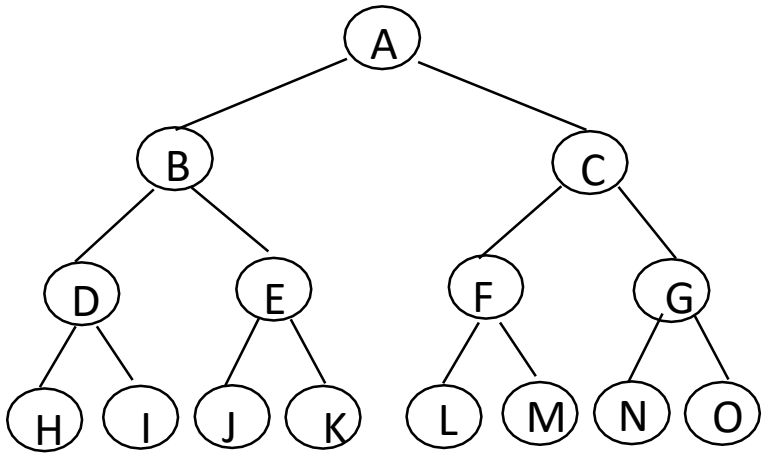
- This representation does not provide direct access to a node and special algorithms are required.
- This representation needs additional space in each node for storing the left and right sub-trees.

FULL BT VS COMPLETE BT

- A binary tree with n nodes and depth k is complete *iff* its n^{th} 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 $2^k - 1$ nodes, $k>=0$.



full binary tree



complete binary tree of depth 4

BINARY TREE TRAVERSALS

The process of going through a tree in such a way that each node is visted once is tree traversal.several method 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 :

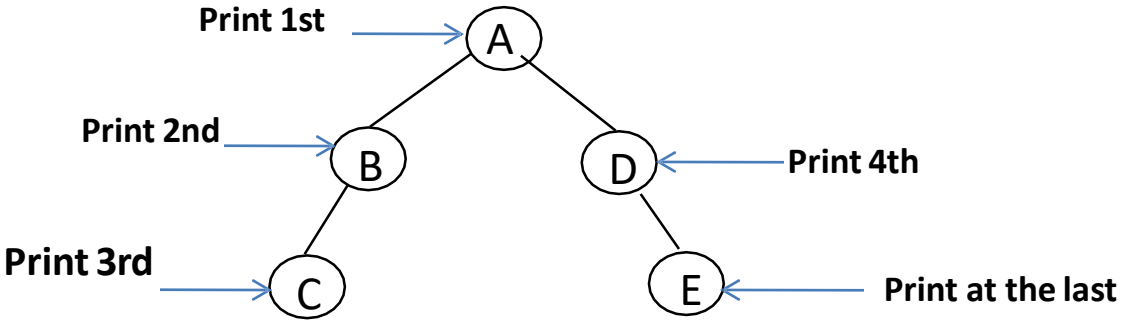
Preorder

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)



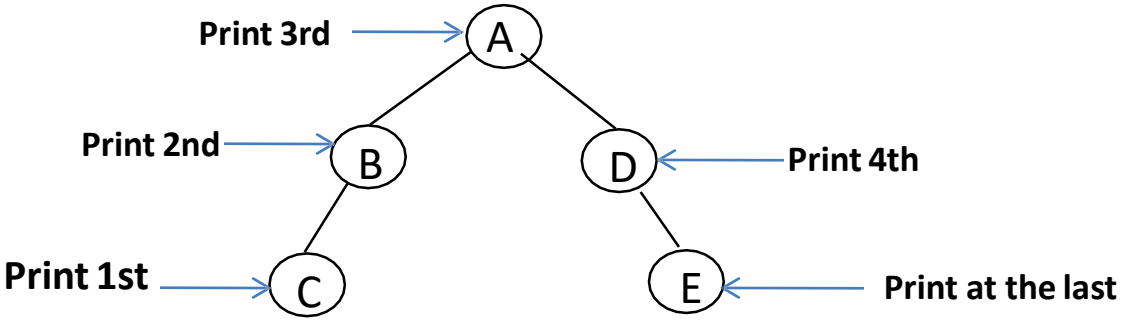
A-B-C-D-E is the preorder traversal of the above figure.

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)



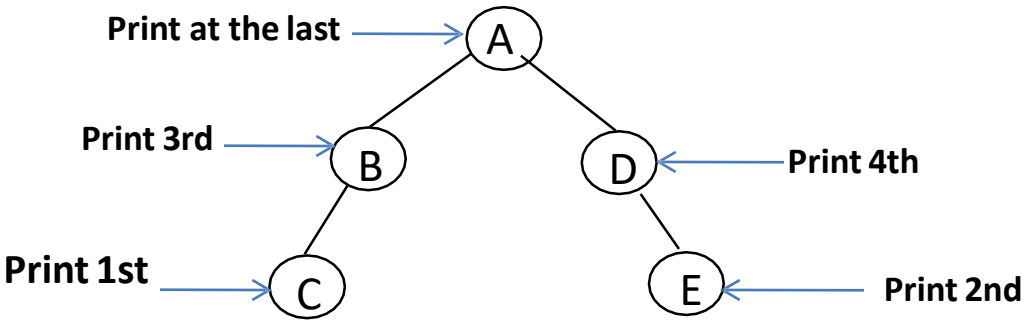
C-B-A-D-E is the Inorder traversal of the above figure.

Postorder

1.Traverse the left subtree in postorder(L)

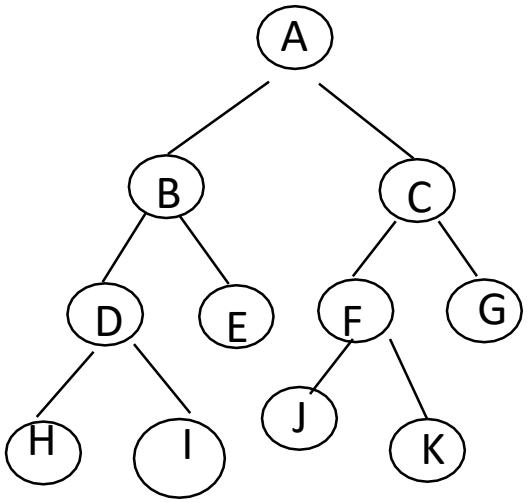
2.Traverse the right subtree in postorder(R)

3.Visit the root(D)

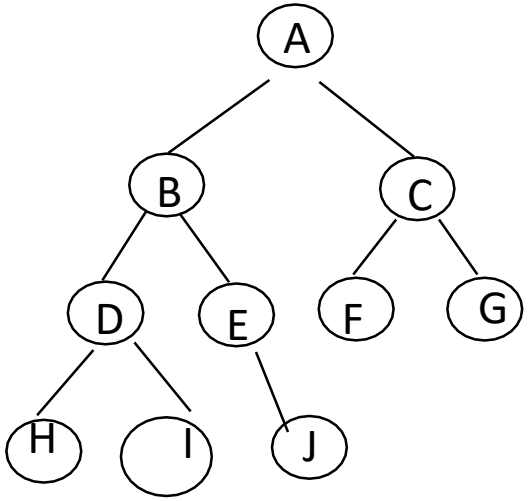


C-D-B-E-A is the postorder traversal of the above figure.

BINARY TREE TRAVERSALS



FIG(a)

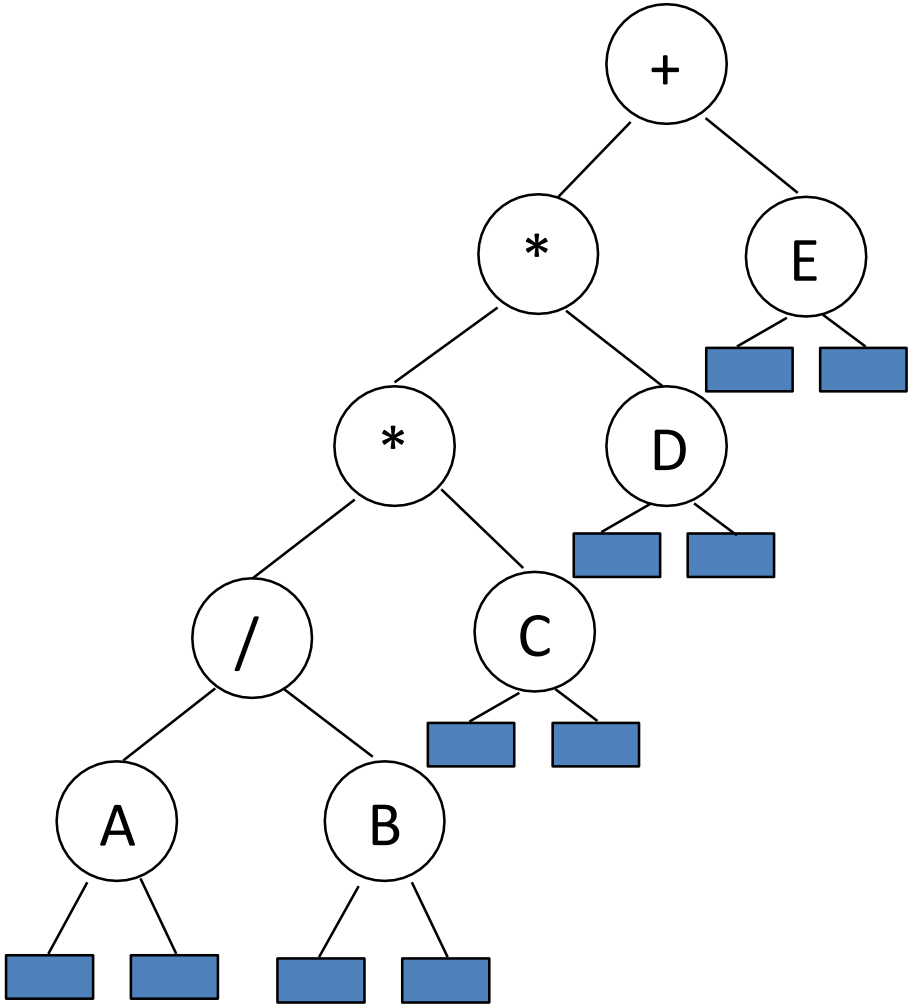


FIG(b)

Preorder:ABDHIECFJKG
Inorder:HDIBEAJFKCG
Postorder:HIDEBJKFGCA

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);
        predorder(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);
        postdorder(ptr->right_child);
        printf("%d", ptr->data);
    }
}
```

A B / C * D * E +

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	

THREADED BINARY TREES

- Two many null pointers in current representatio 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

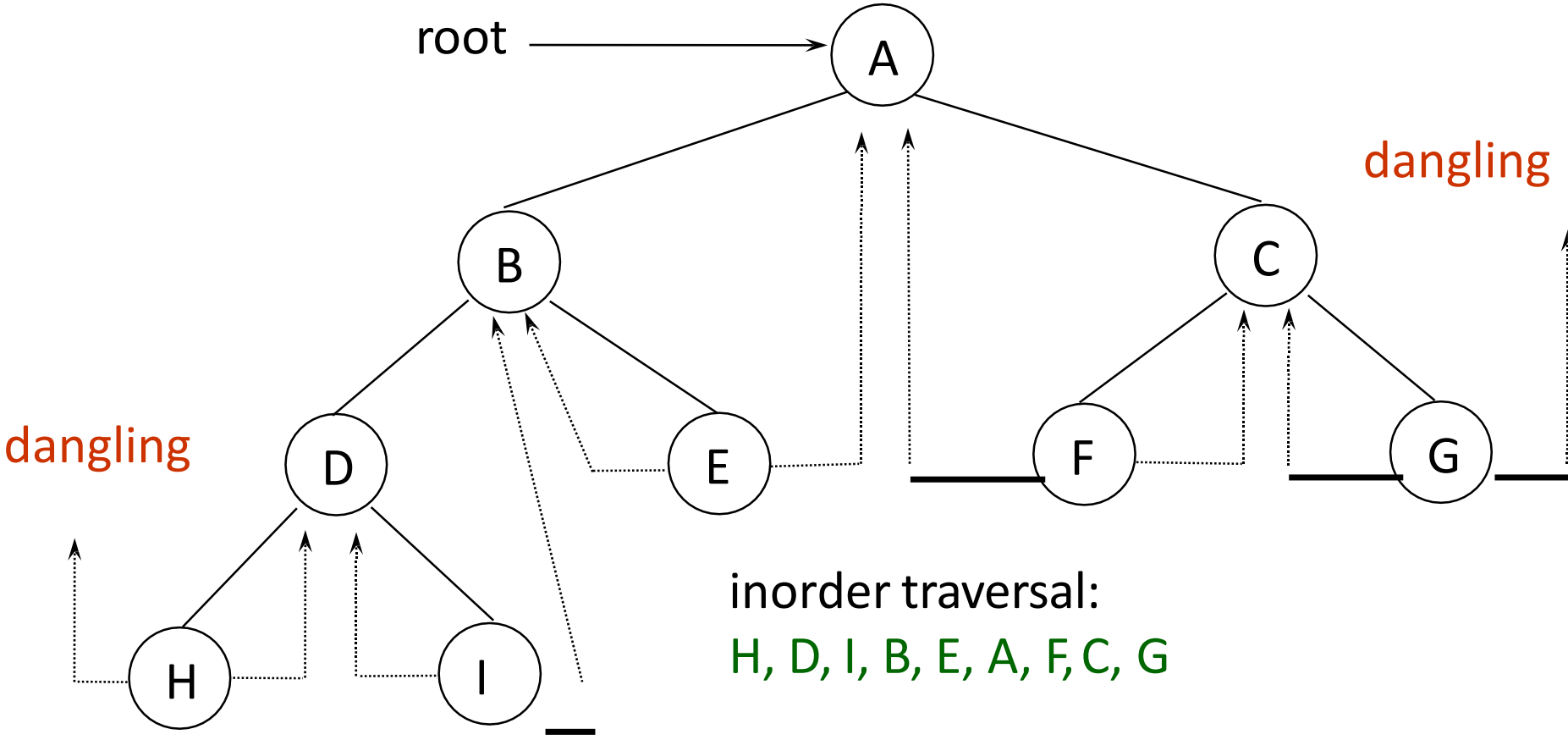
If `ptr->left_child` is null,

replace it with a pointer to the node that would be visited *before* `ptr` in an *inorder traversal*

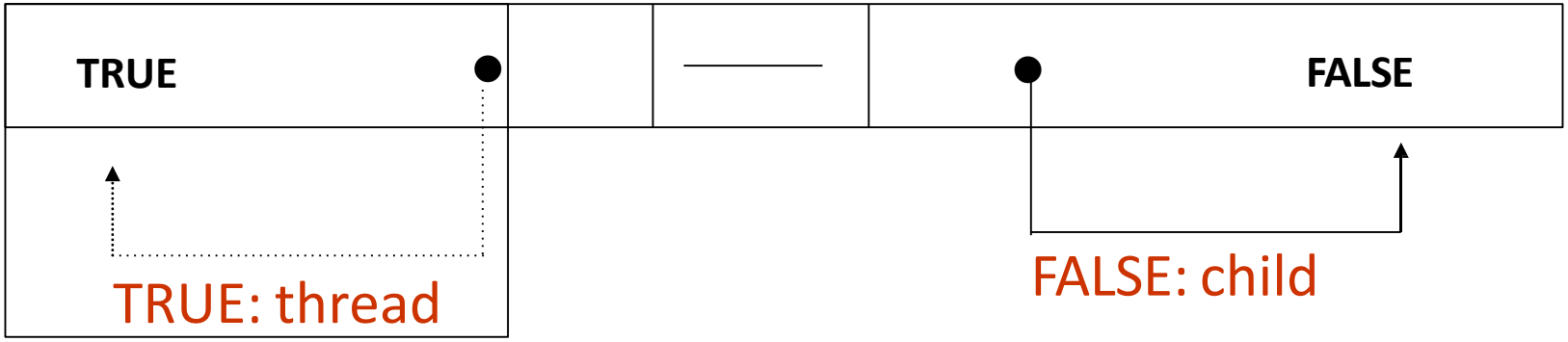
If `ptr->right_child` is null,

replace it with a pointer to the node that would be visited *after* `ptr` in an *inorder traversal*

A THREADED BINARY TREE



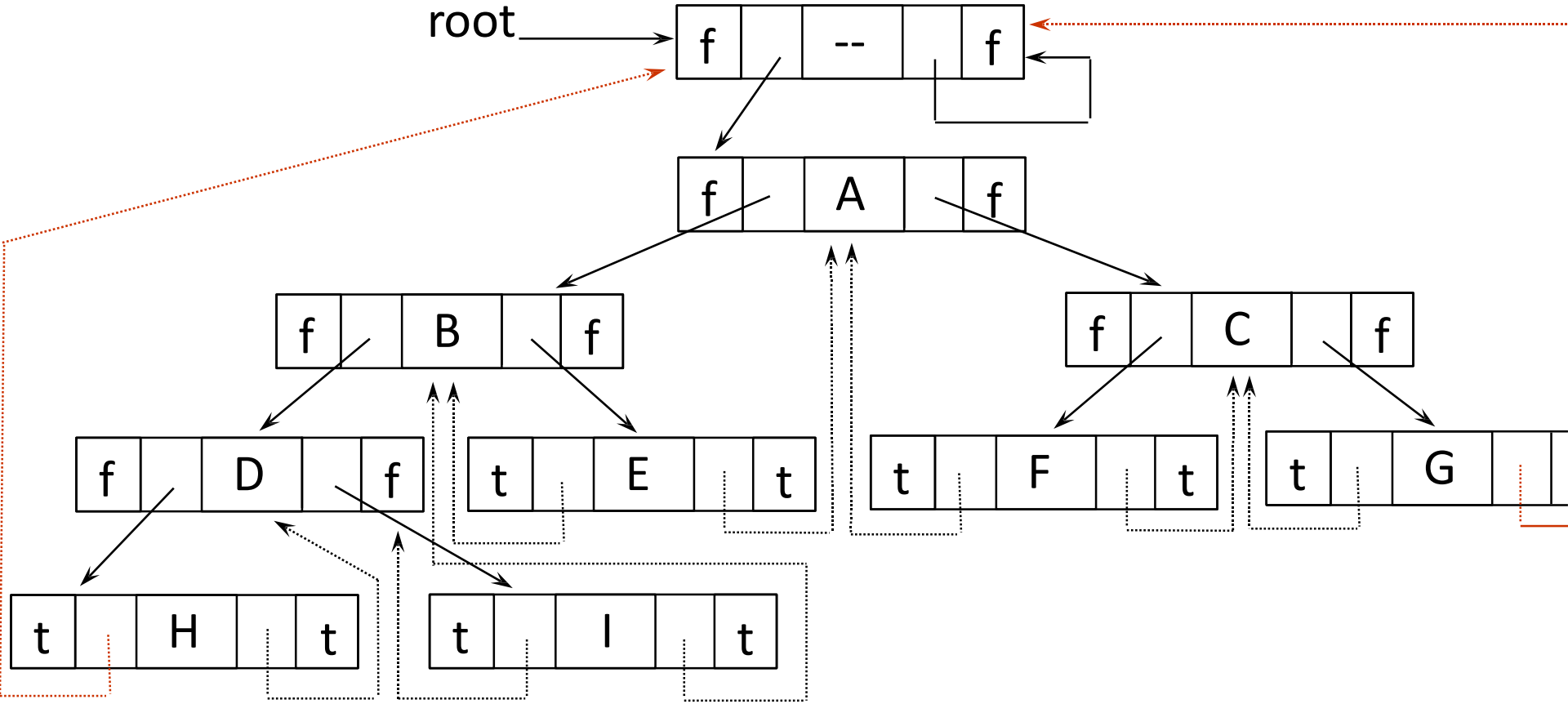
DATA STRUCTURES FOR THREADED BT



left_thread left_child data right_child typedef right_thread struct threaded_tree *threaded_pointer;

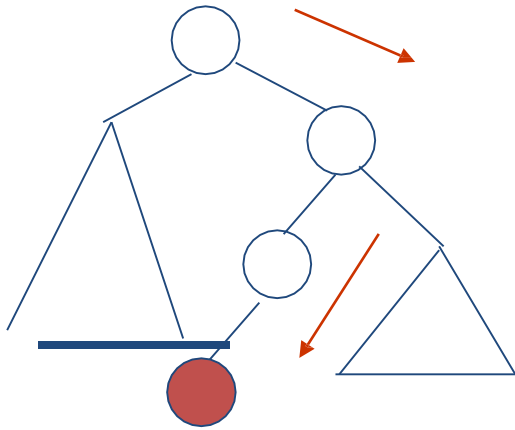
```
typedef struct threaded_tree {
    short int left_thread;
    threaded_pointer left_child;
    char data;
    threaded_pointer right_child;
    short int right_thread;  };
```

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);
     }
}
```

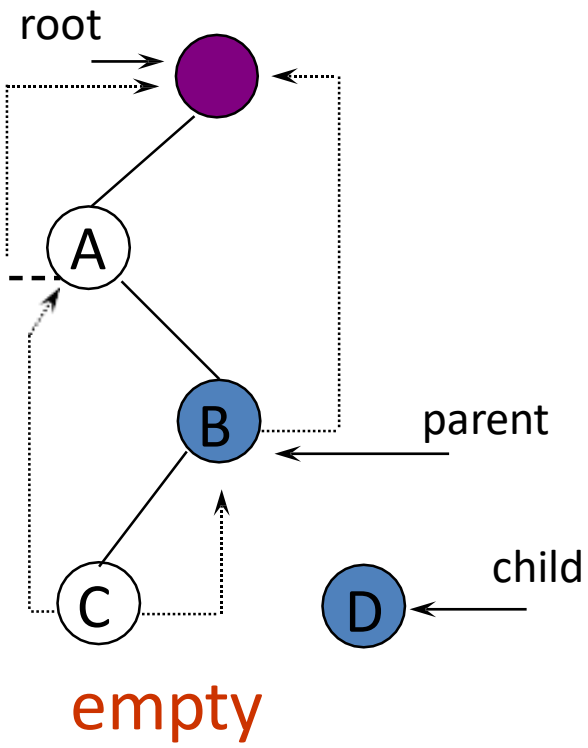
O(n)(timecomplexity)

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.



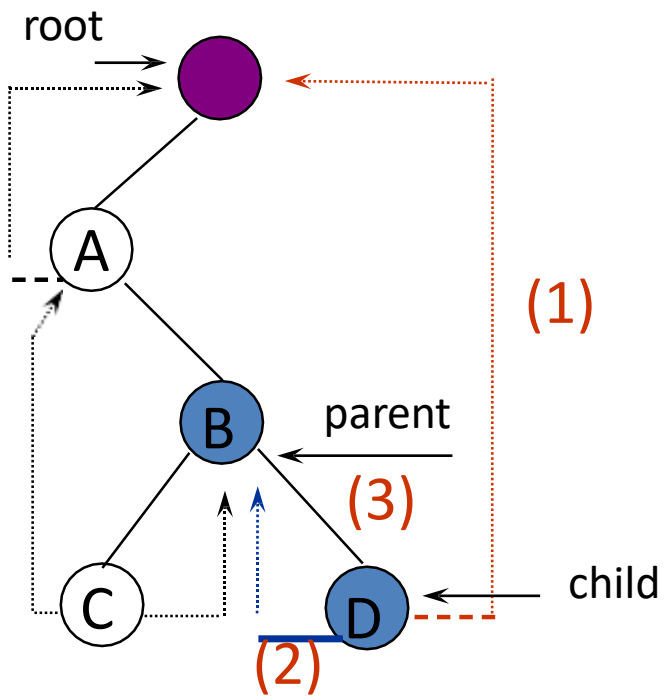
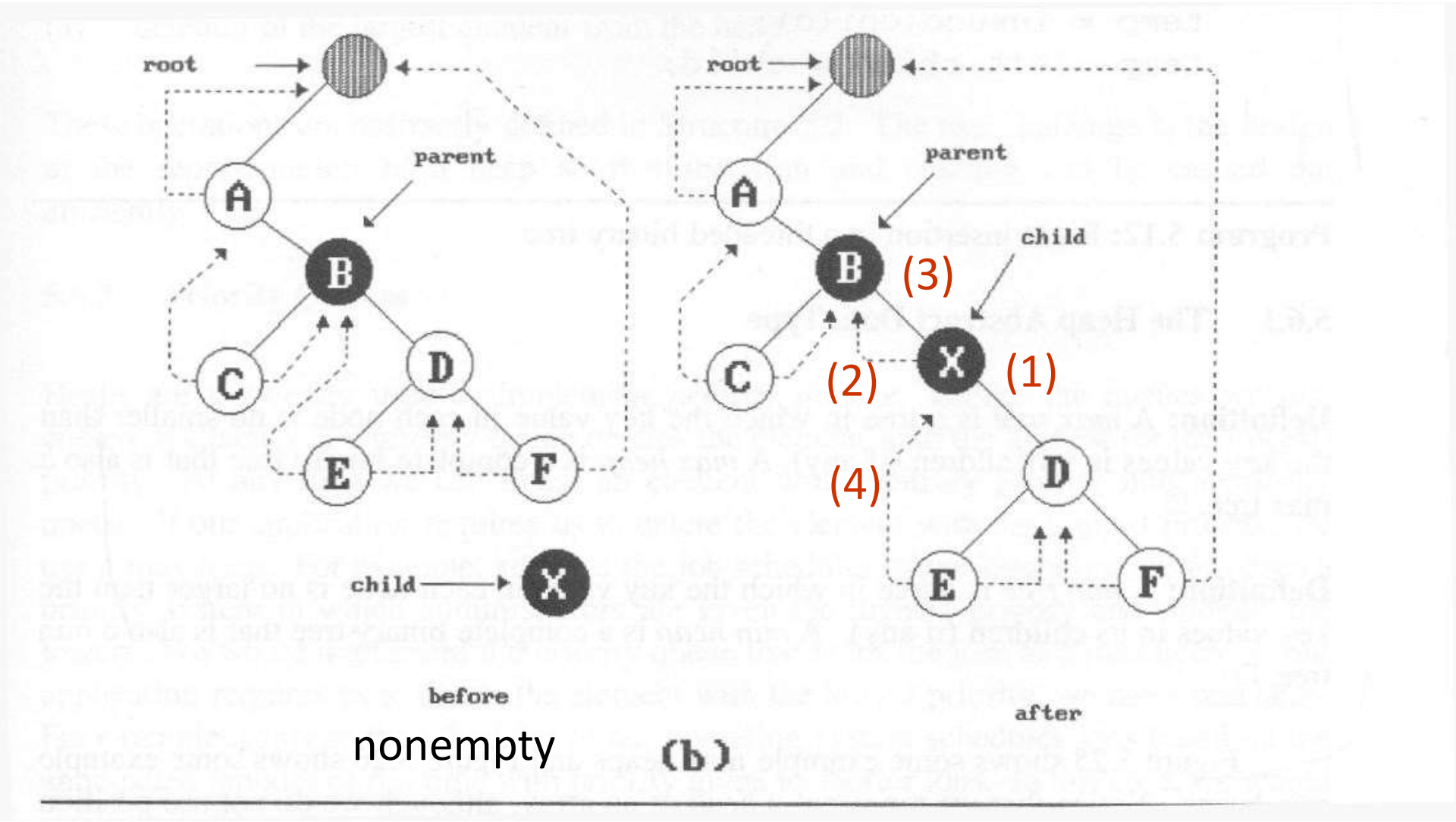


***Figure 5.24:** Insertion of child as a right child of parent in a threaded binary tree (p.217)



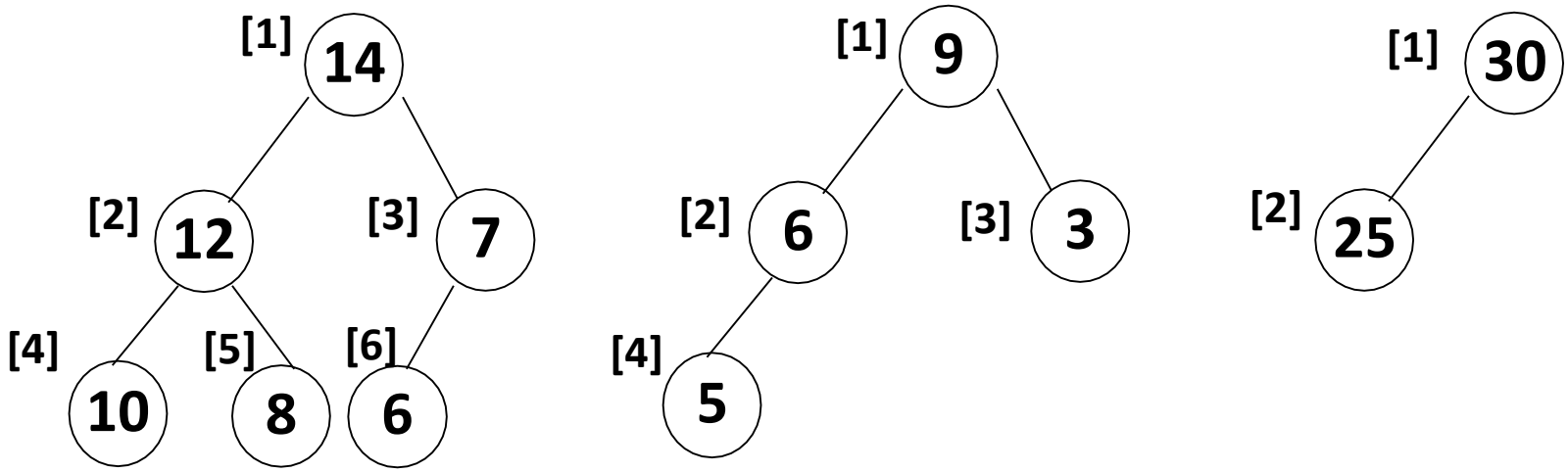
RIGHT INSERTION IN THREADED BTS

```
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

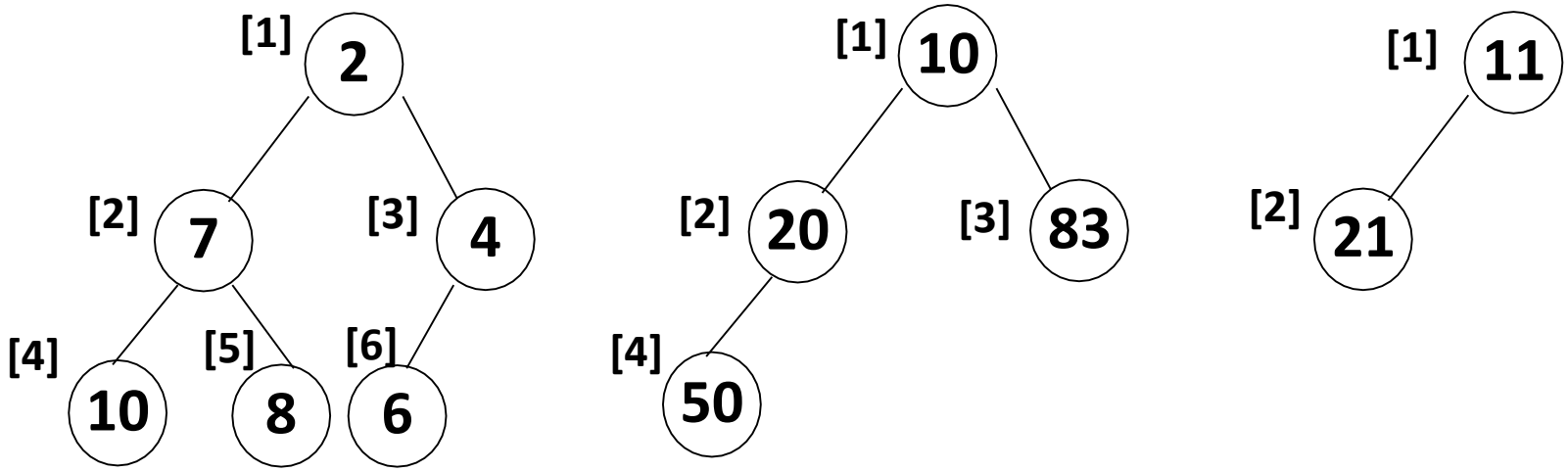
Sample max heaps



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

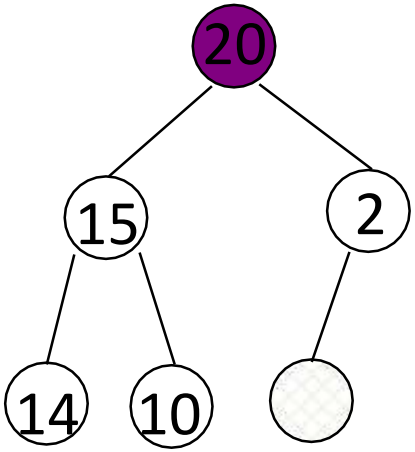
Boolean HeapFull(heap, n)::= if (n==max_size) return TRUE
else return FALSE

MaxHeap Insert(heap, item, n)::= if (!HeapFull(heap,n)) insert item into heap and return the resulting heap
else return error

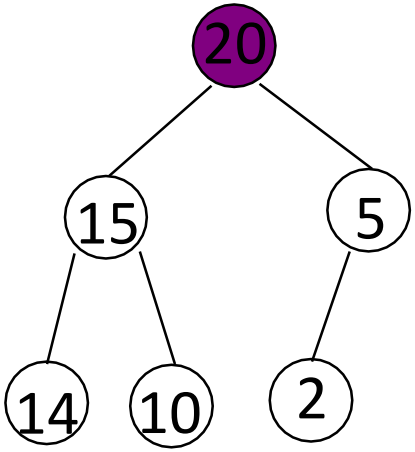
Boolean HeapEmpty(heap, n)::= if (n>0) return FALSE
else return TRUE

Element Delete(heap,n)::= if (!HeapEmpty(heap,n)) return one instance of the largest element in the heap and remove it from the heap
else return error

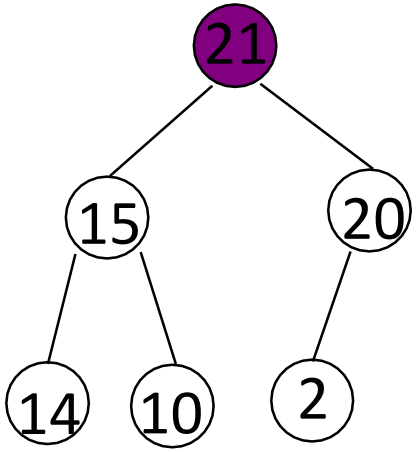
EXAMPLE OF INSERTION TO MAX HEAP



initial location of new node



insert 5 into heap



insert 21 into heap

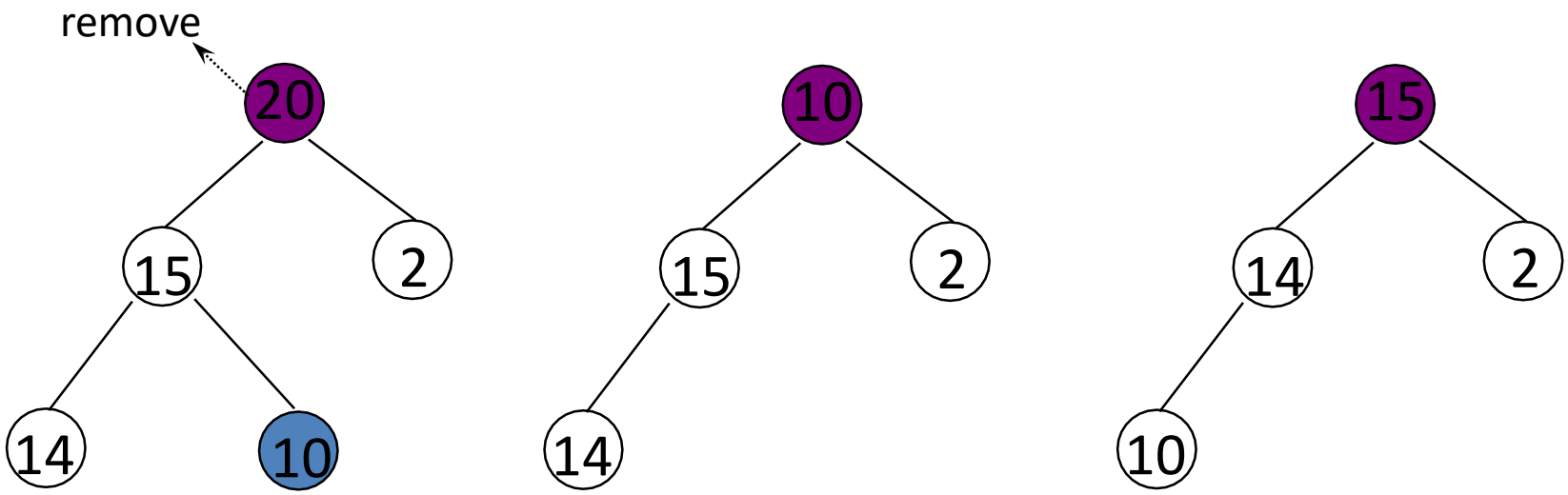
INSERTION INTO A MAX HEAP

```
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;
}
```

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

$O(\log_2 n)$

EXAMPLE OF DELETION FROM MAX HEAP



DELETION FROM A MAX HEAP

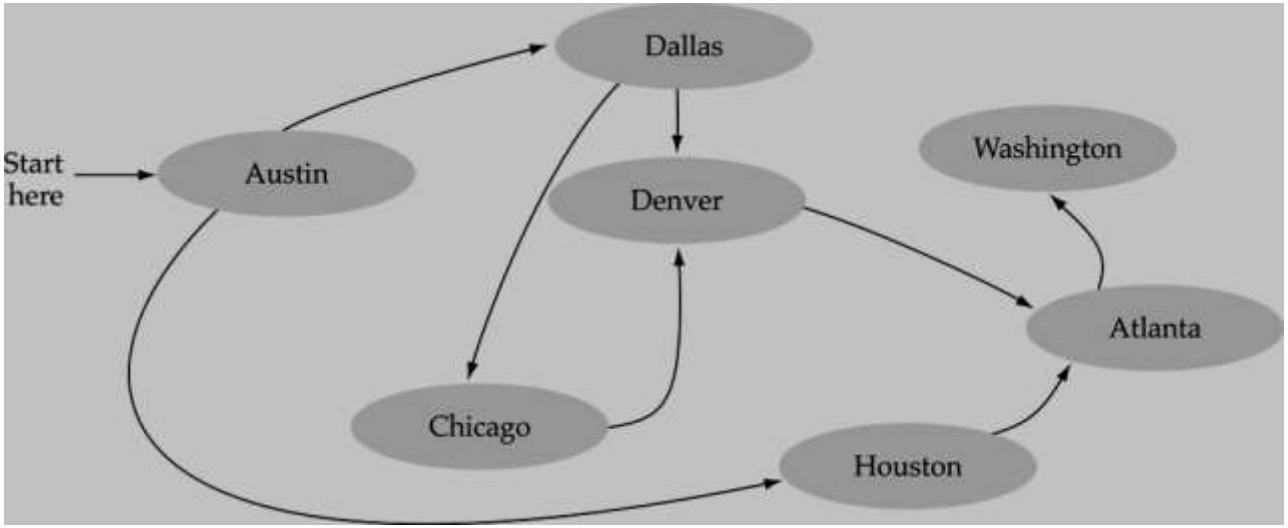
```
element delete_max_heap(int *n)
{
  int parent, child;
  element item, temp;
  if (HEAP_EMPTY(*n)) {
    fprintf(stderr, "The heap is empty\n");
    exit(1);
  }
  /* save value of the element with the
     highest key */
  item = heap[1];
  /* use last element in heap to adjust heap
  temp = heap[(*n)--];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

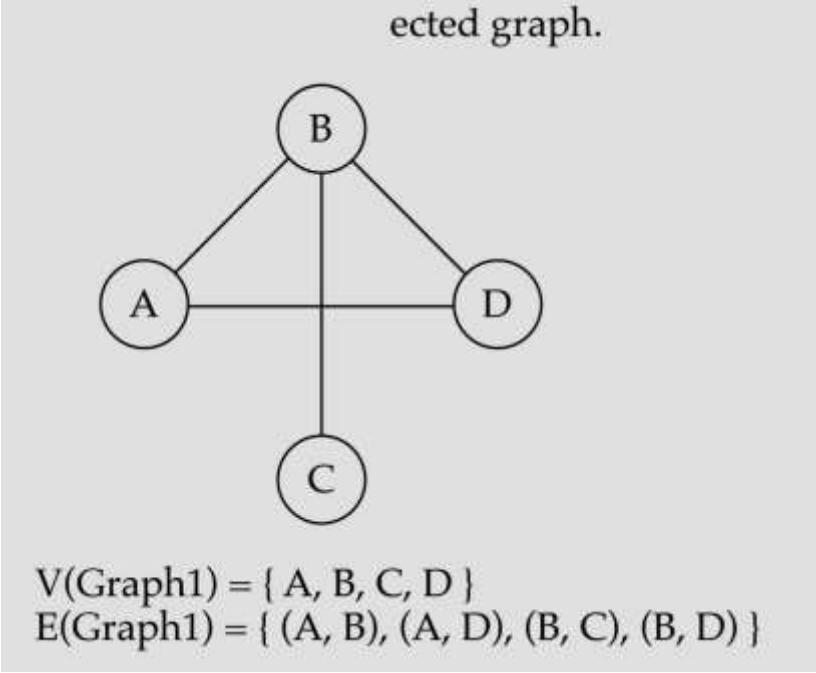
  G=(V,E)

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

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

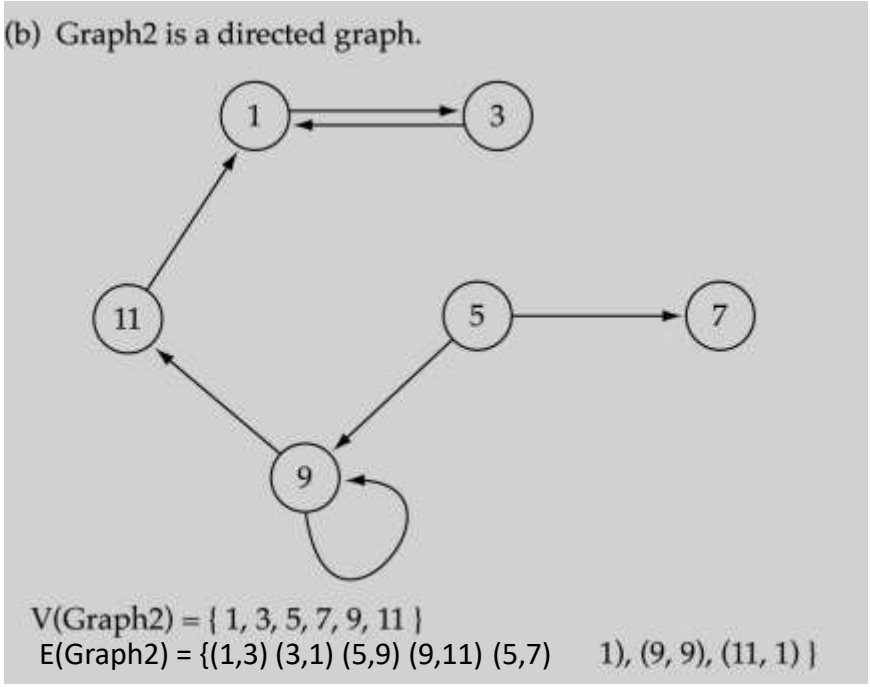
DIRECTED VS. UNDIRECTED GRAPHS

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



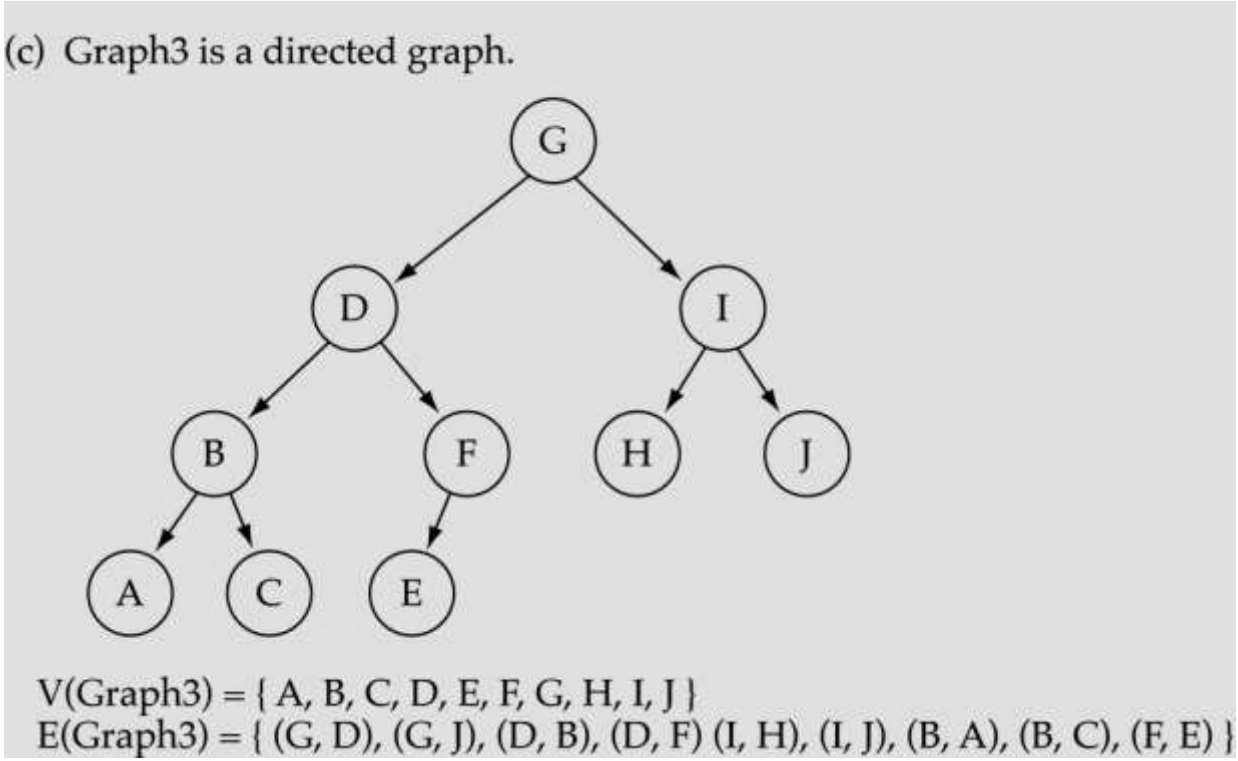
DIRECTED VS. UNDIRECTED GRAPHS

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



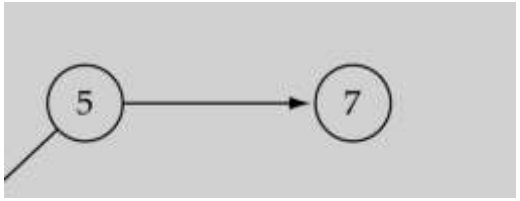
TREES VS GRAPHS

- Trees are special cases of graphs!!



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

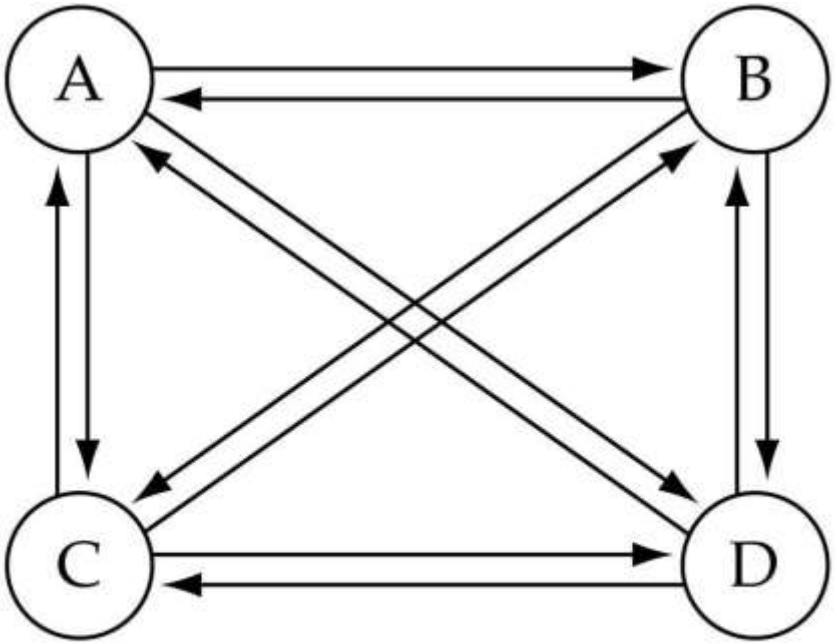
- Path: a sequence of vertices that connect two nodes in a graph
- Complete graph: a graph in which every vertex is directly connected to every other vertex

GRAPH TERMINOLOGY

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

*N * (N-1)*

$O(N^2)$



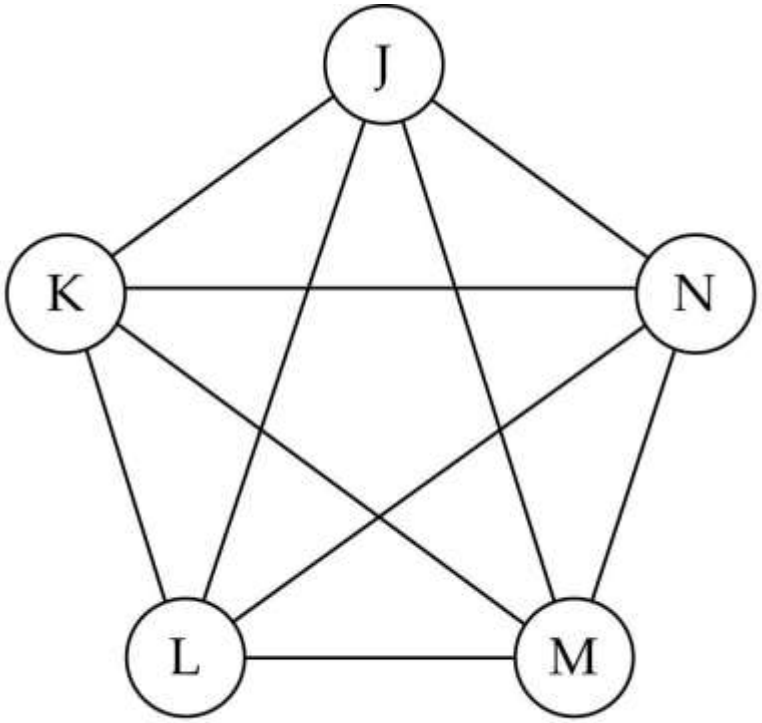
(a) Complete directed graph.

GRAPH TERMINOLOGY

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

*N * (N-1) / 2*

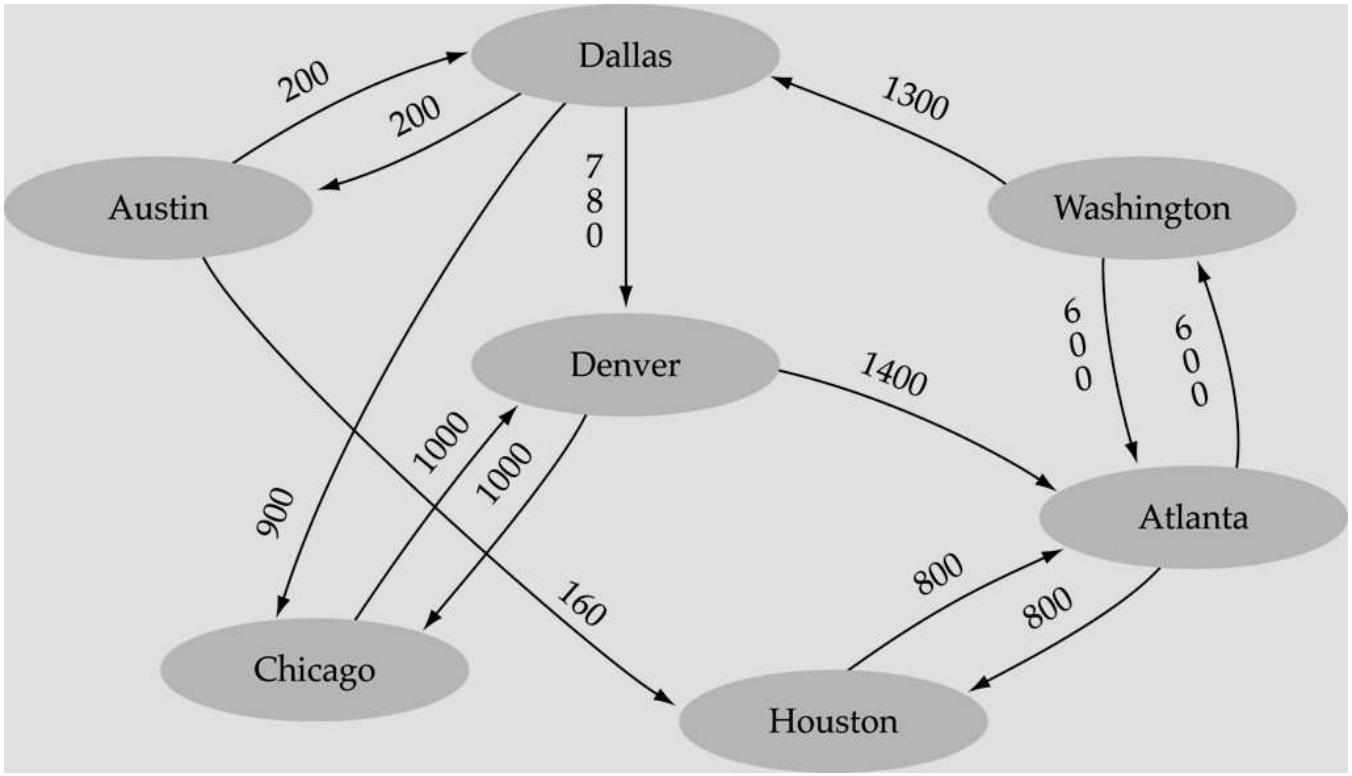
$O(N^2)$



(b) Complete undirected graph.

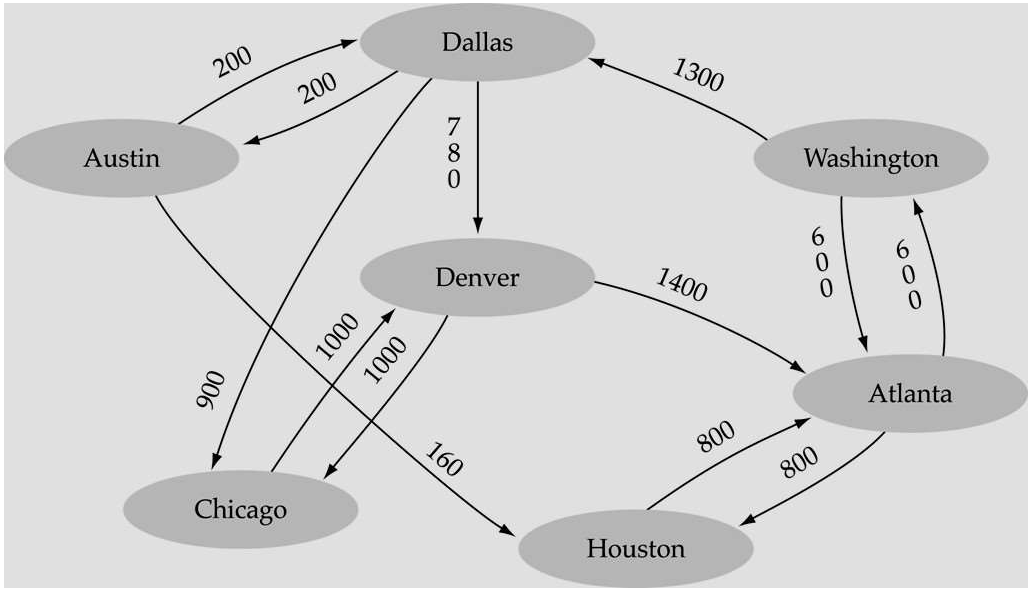
GRAPH TERMINOLOGY

- 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]	

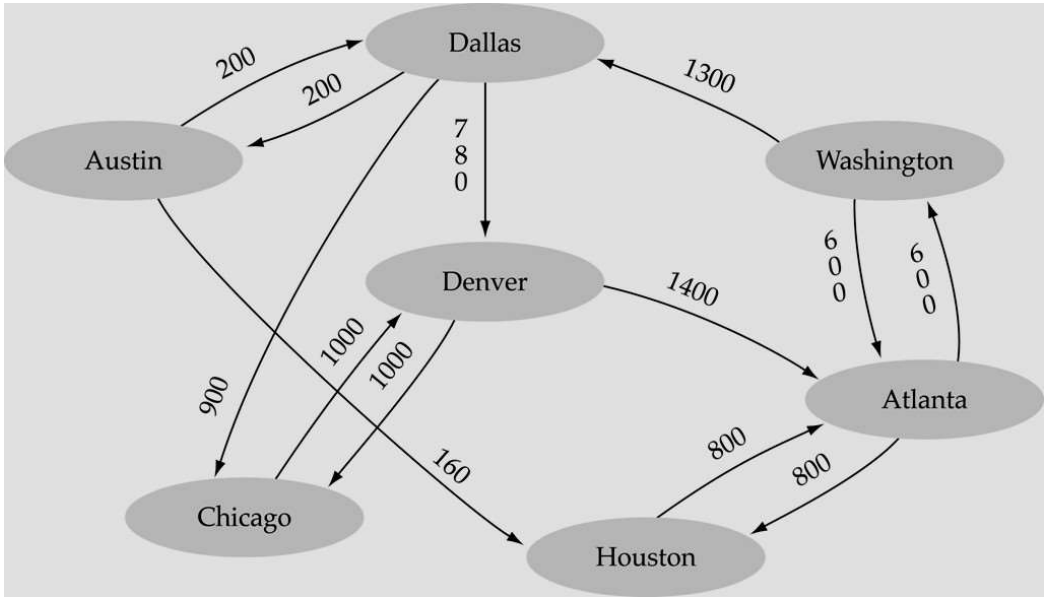
.edges

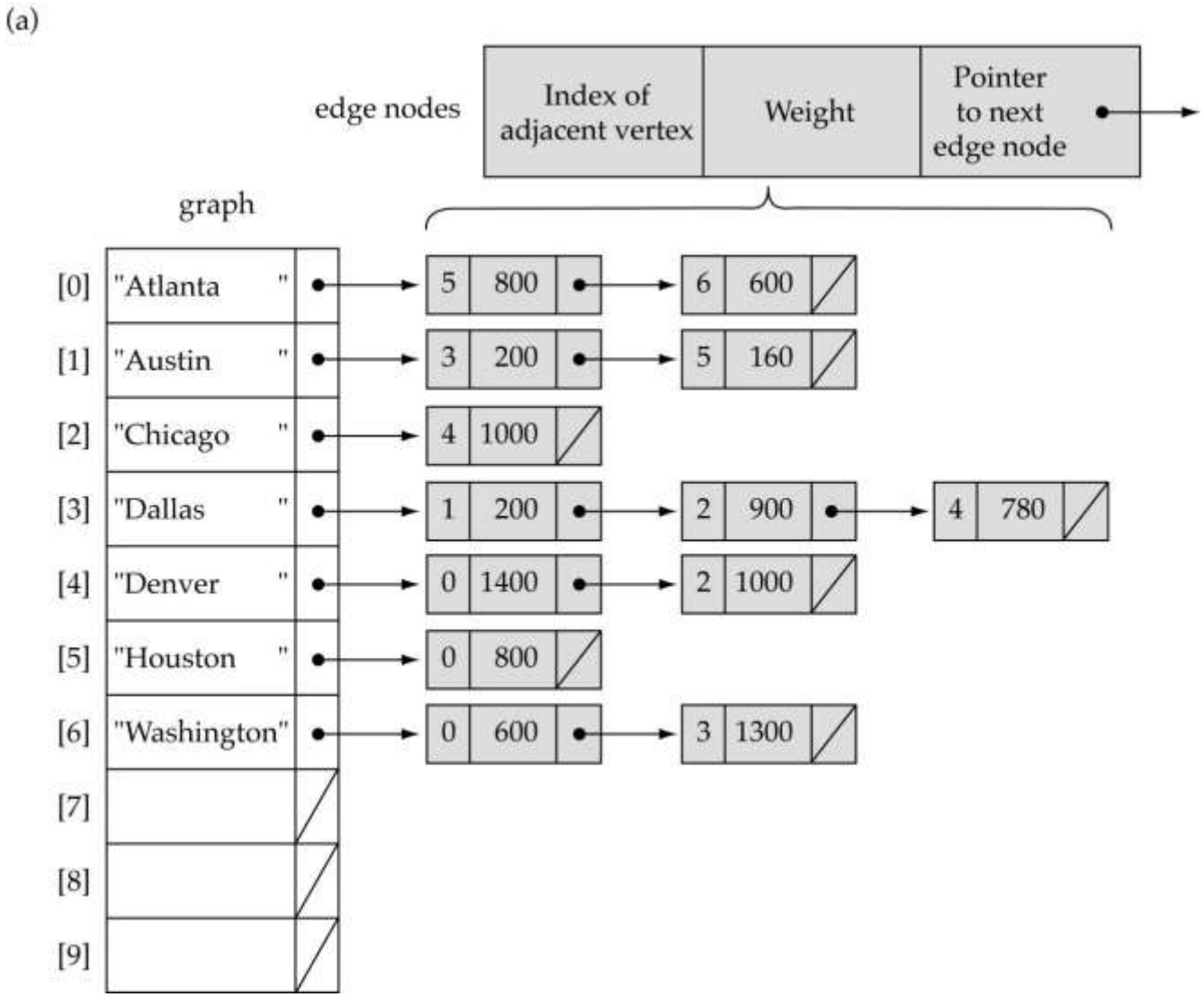
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•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

(Array positions marked '•' are undefined)

GRAPH IMPLEMENTATION

- 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)
edge nodes
Index of adjacent vertex
Weight
Pointer to next edge node
graph
[0] "Atlanta "
[1] "Austin "
[2] "Chicago "
[3] "Dallas "
[4] "Denver "
[5] "Houston "
[6] "Washington"
[7]
[8]
[9]
5 800 6 600
3 200 5 160
4 1000
1 200 2 900 4 780
0 1400 2 1000
0 800
0 600 3 1300

ADJACENCY MATRIX VS. ADJACENCY LIST REPRESENTATION

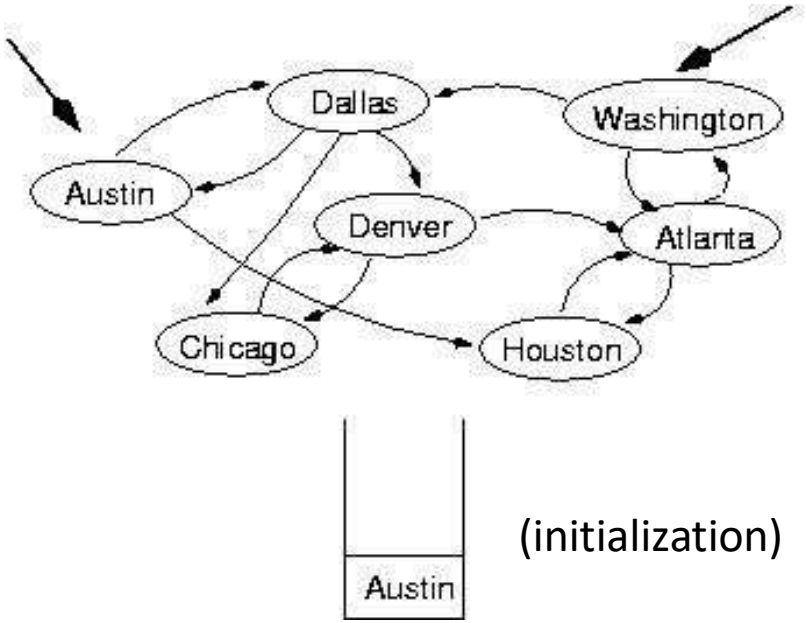
- **Adjacency matrix**
  - Good for dense graphs --$|E| \sim O(|V|^2)$
  - Memory requirements: $O(|V| + |E|) = O(|V|^2)$
  - Connectivity between two vertices can be tested quickly
- **Adjacency list**
  - Good for sparse graphs -- $|E| \sim O(|V|)$
  - Memory requirements: $O(|V| + |E|) = O(|V|)$
  - Vertices adjacent to another vertex can be found quickly

DEPTH-FIRST-SEARCH (DFS)

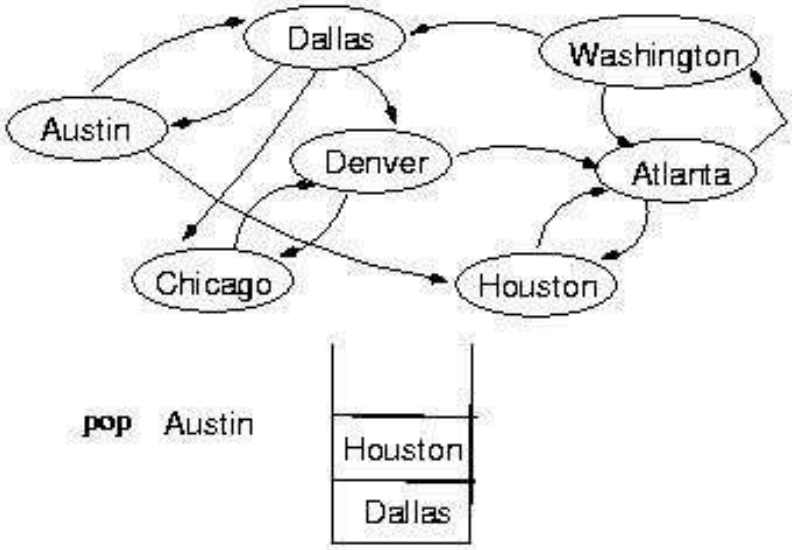
- What is the idea behind DFS?
  - Travel as far as you can down a path
  - Back up *as little as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)
- DFS can be implemented efficiently using a *stack*

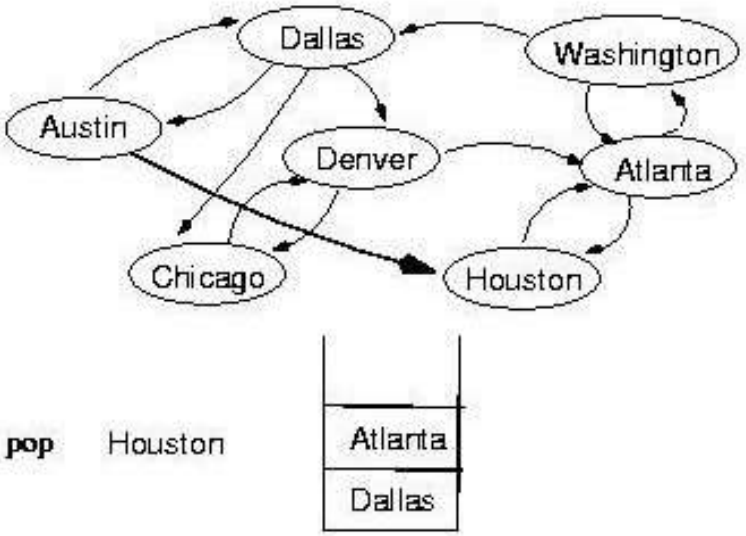
DEPTH-FIRST-SEARCH (DFS)

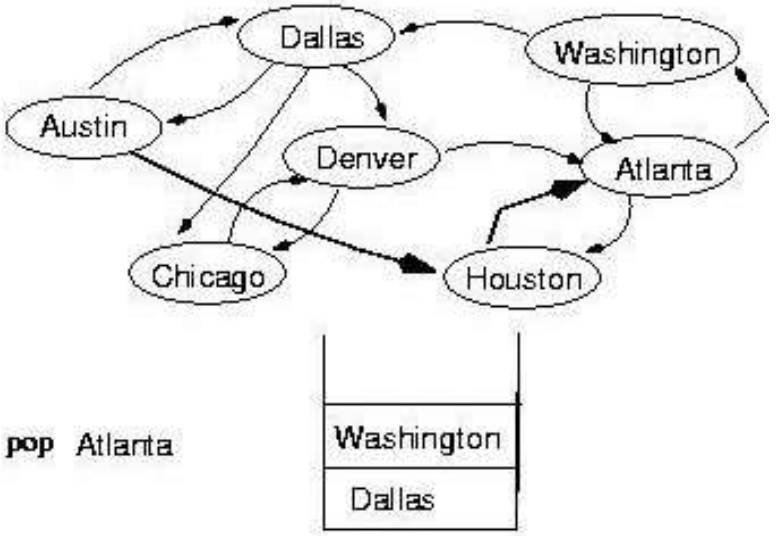
```
Set found to false
stack.Push(startVertex)
DO
  stack.Pop(vertex)
  IF vertex == endVertex
  Set found to true
  ELSE
    Push all adjacent vertices onto stack
WHILE !stack.IsEmpty() AND !found

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

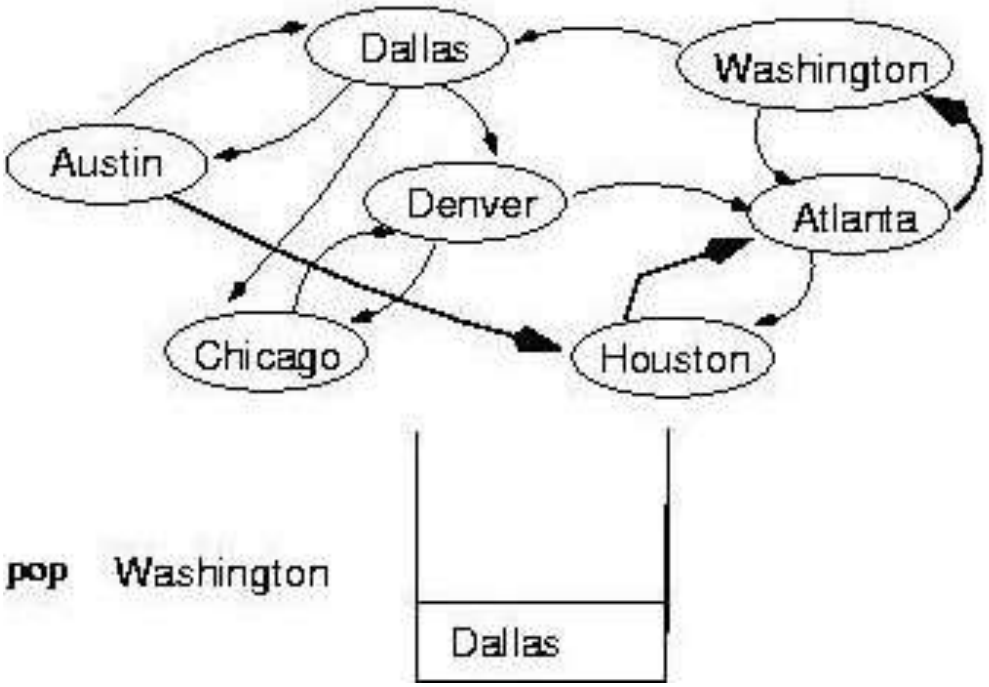


(initialization)




Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
pop Houston
Atlanta
Dallas


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
pop Atlanta
Washington
Dallas


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
pop 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,
                                  QueTye<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)

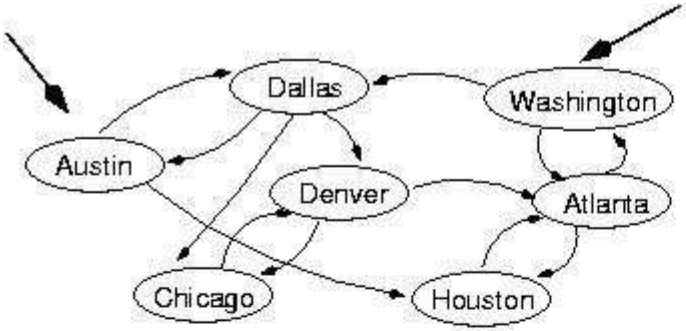
- What is the idea behind BFS?
  - Look at all possible paths at the same depth before you go at a deeper level
  - Back up *as far as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)

BREADTH-FIRST-SEARCHING (BFS)

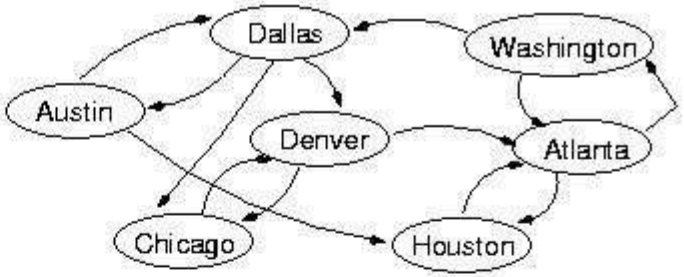
- BFS can be implemented efficiently using a *queue*

```
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
```

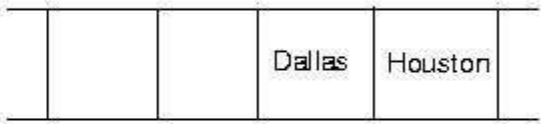
- Should we mark a vertex when it is enqueued or when it is dequeued ?

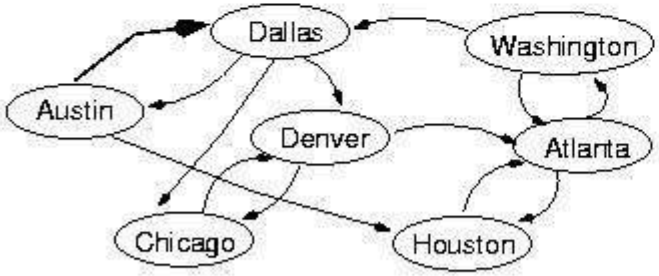


(initialization)

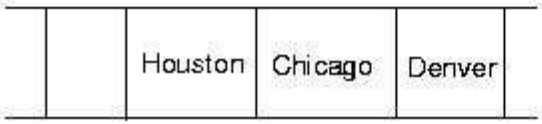


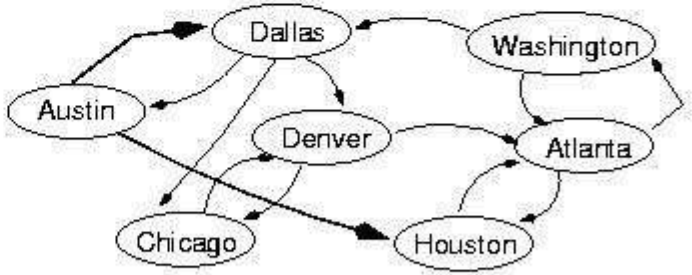
dequeue Austin



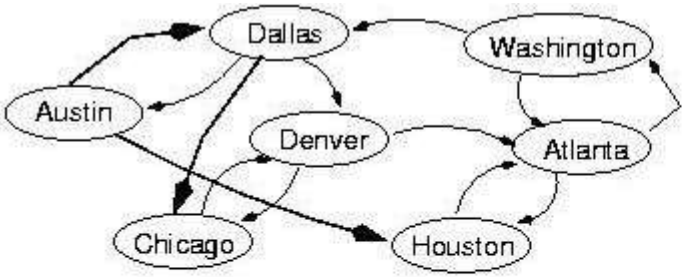


dequeue Dallas

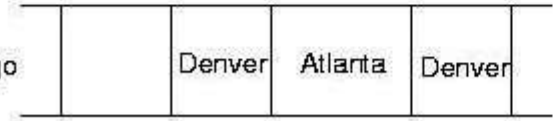


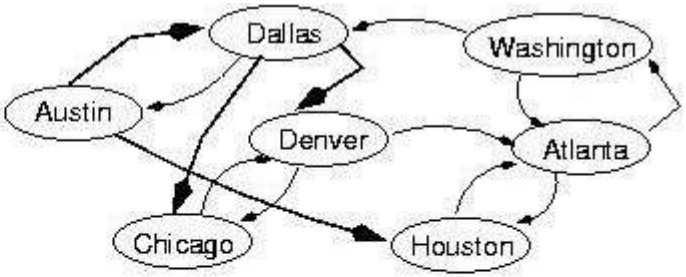


dequeue Houston

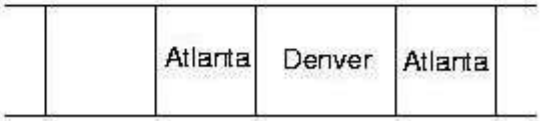


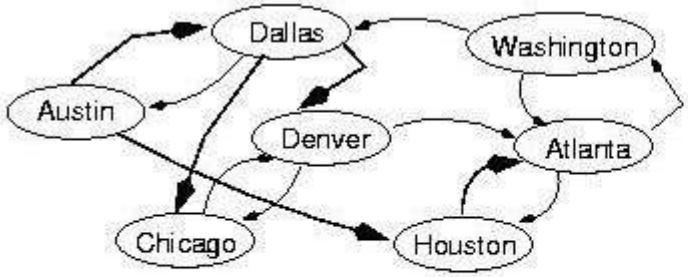
dequeue Chicago



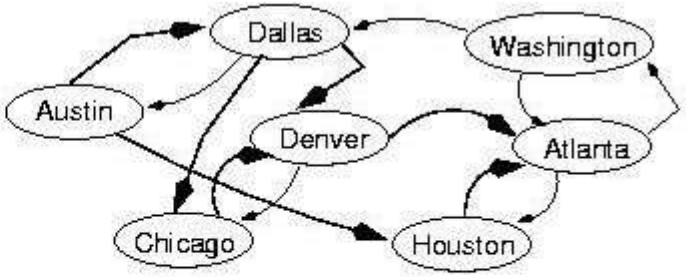


dequeue Denver

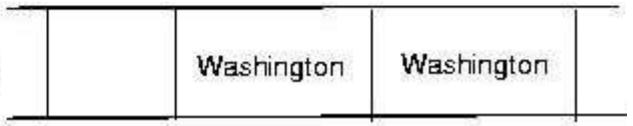


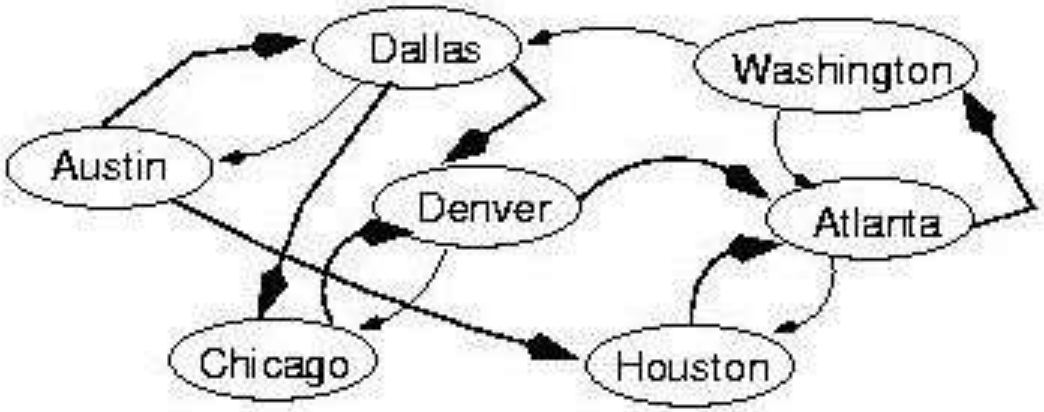


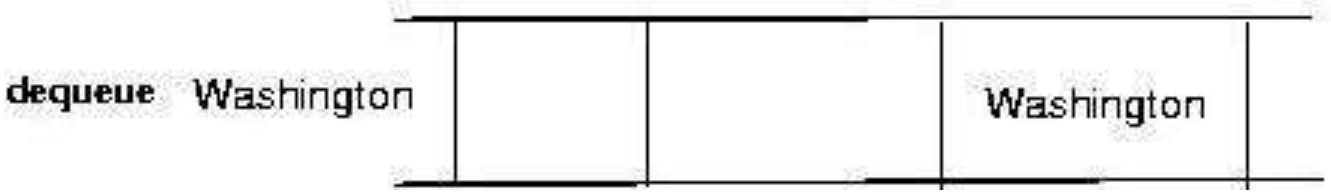
dequeue Atlanta



dequeue Denver,
next: Atlanta




Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston


dequeue Washington
Washington

```
template<class VertexType>
void BreadthFirtsSearch(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(!vertxQ.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

- 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-5

BINARY TREES AND HASHING

Binary search trees: Binary search trees, properties and operations.

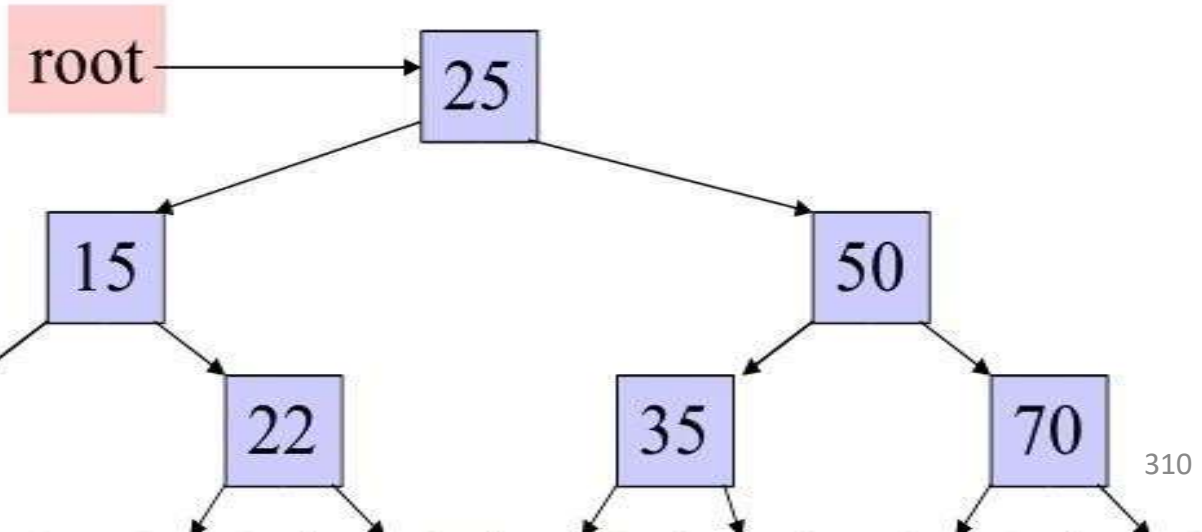
Balanced search trees: AVL trees

Introduction to M-Way search trees, B trees.

Hashing and collision: Introduction, hash tables, hash functions, collisions, applications of hashing.

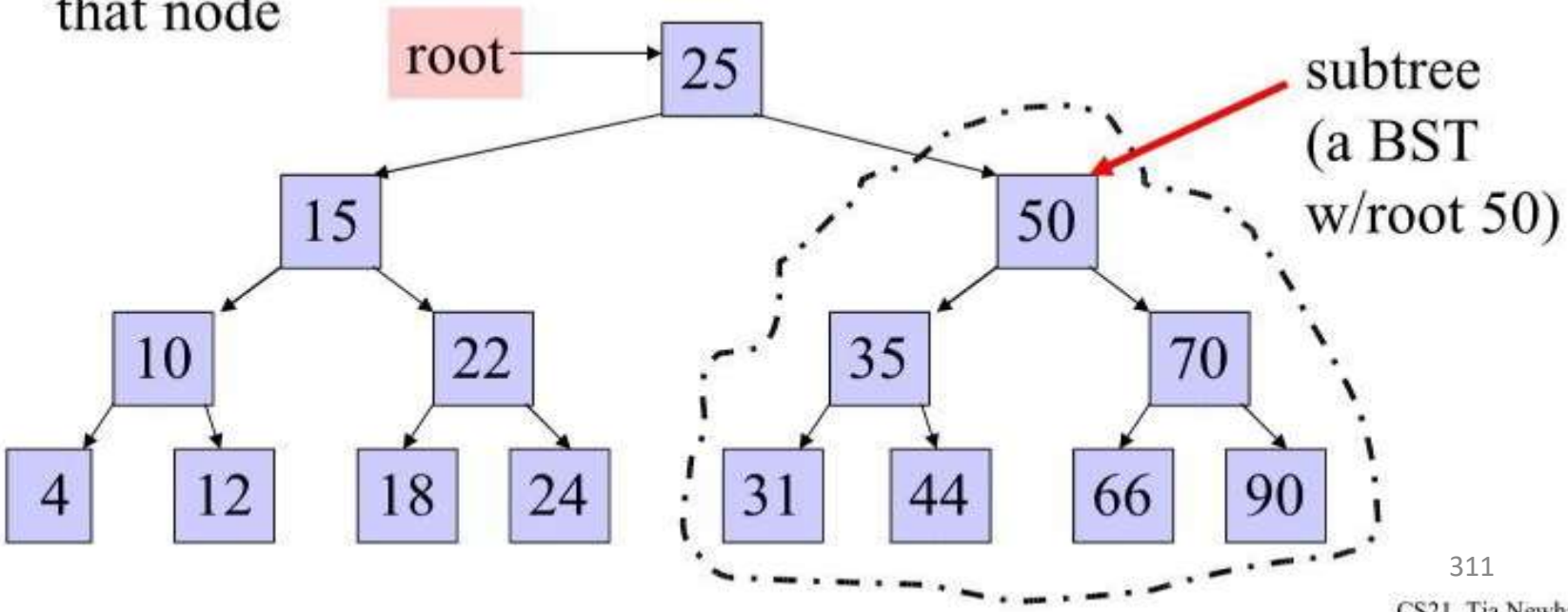
Binary Search Trees (BST)

1. Hierarchical data structure with a single pointer to root no
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

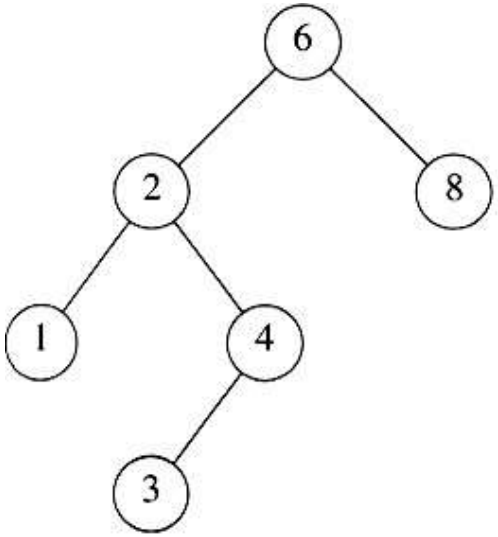
- **A binary search tree is a binary tree in which it has at most two children, the key values in the left node is less than the root and the key values in the right node is greater than the root.**

*** It doesn't have any order.**

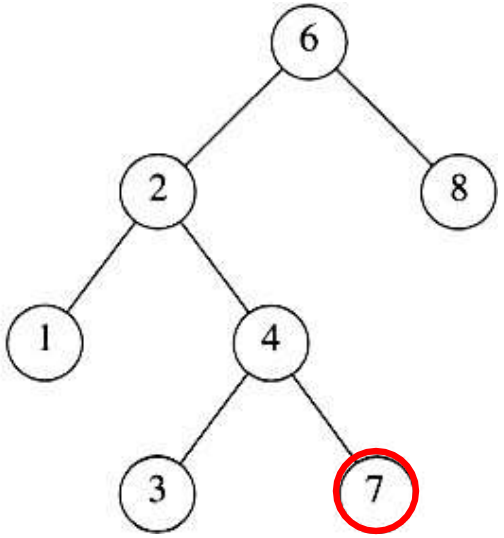
Note : * Every binary search tree is a binary tree.

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

EXAMPLE OF BINARY SEARCH TREE



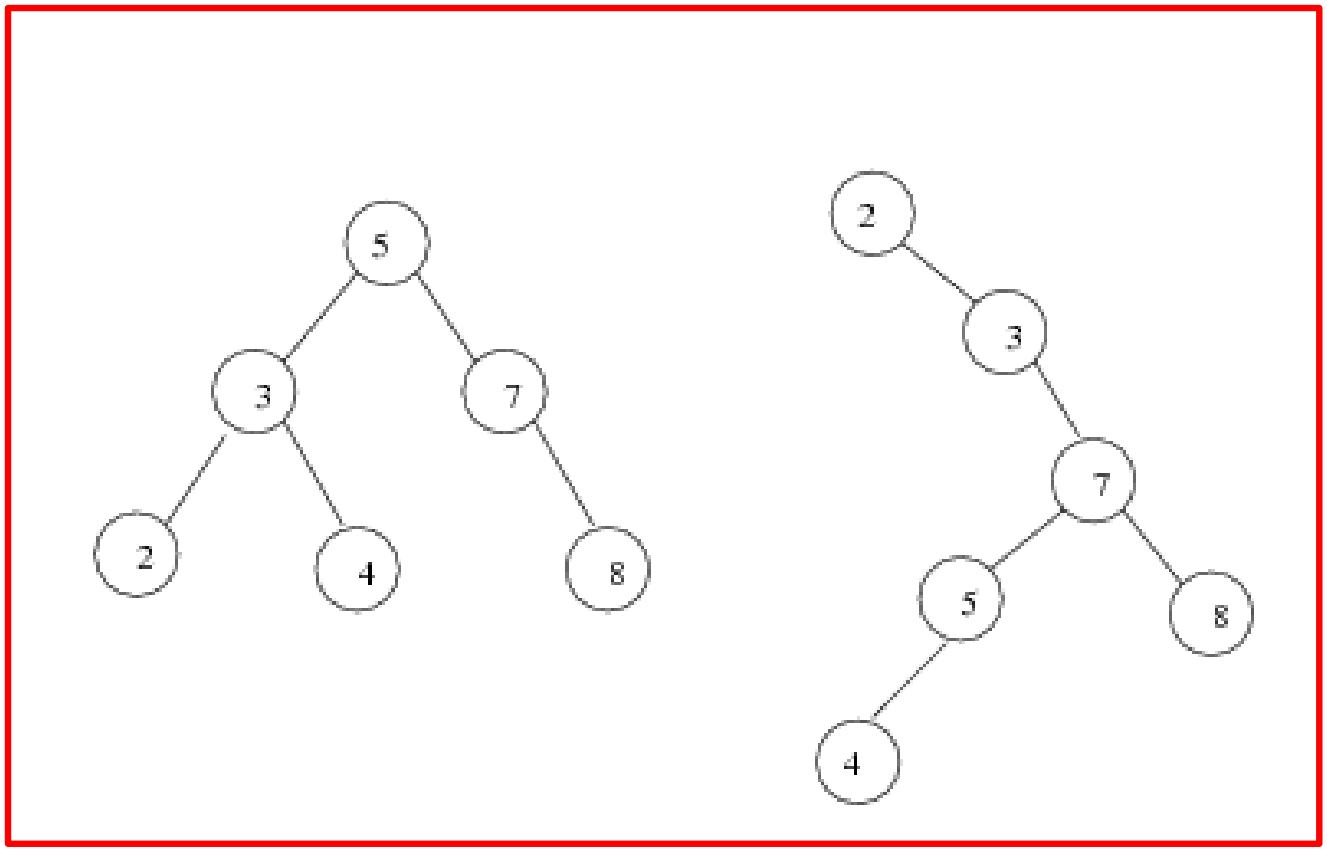
A binary search tree



Not a binary search tree

BINARY SEARCH TREES

The same set of keys may have different BSTs



BST OPERATIONS

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

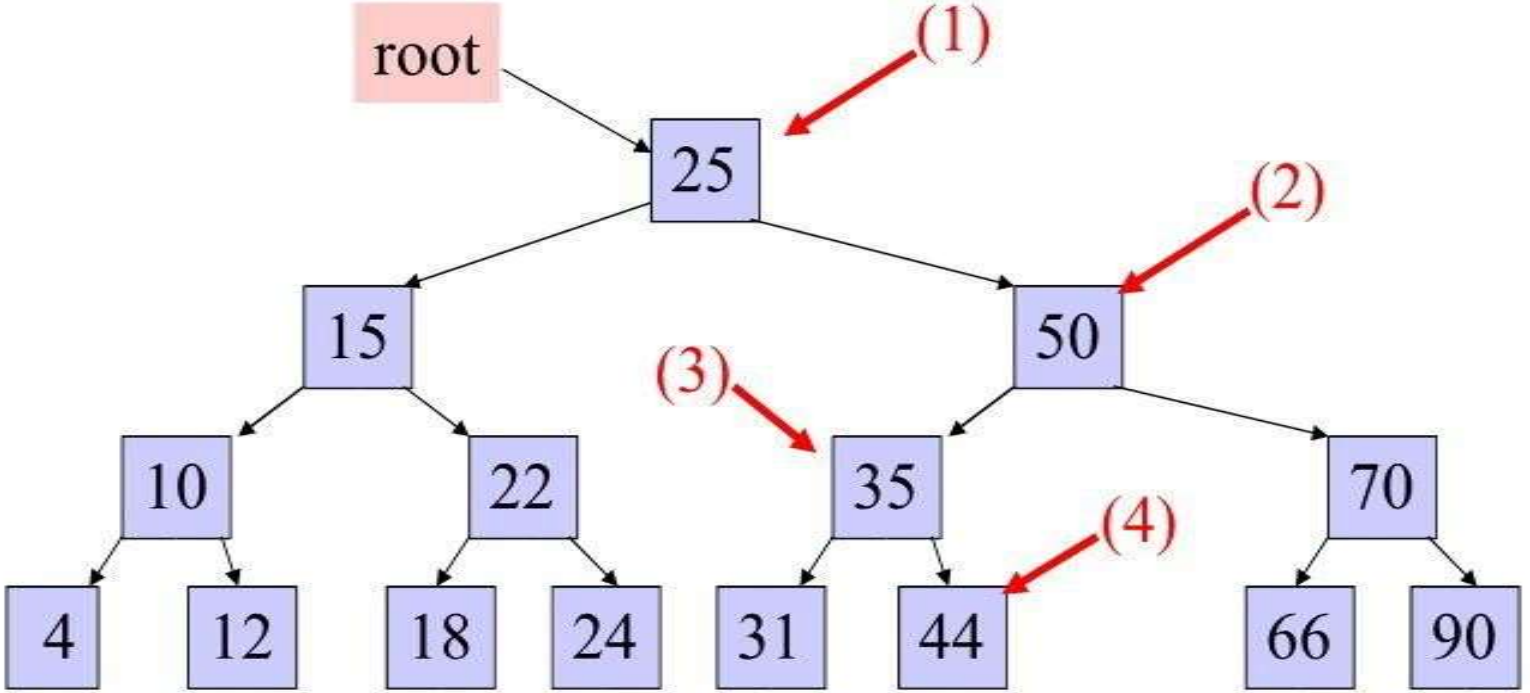
- **Searches**
- **Insertion**
- **Deletion**

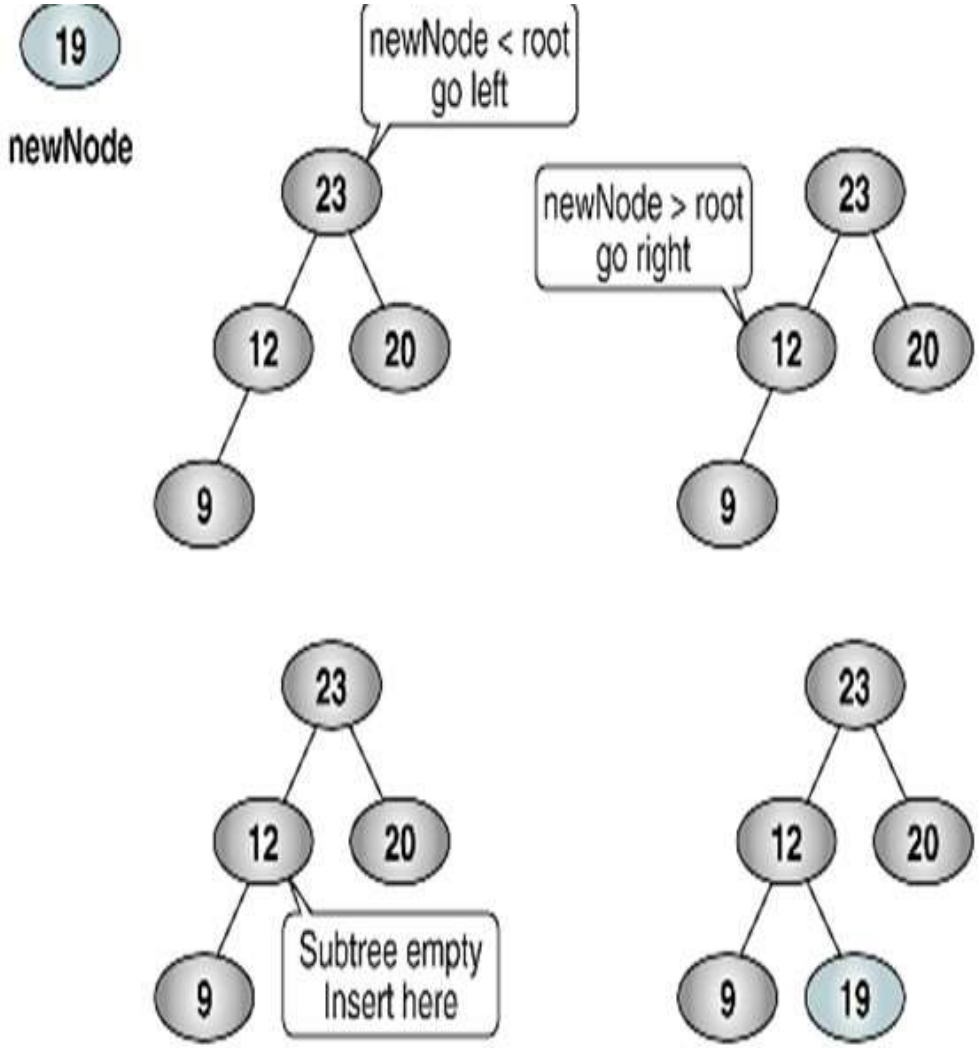
THREE BST SEARCH ALGORITHMS

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

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

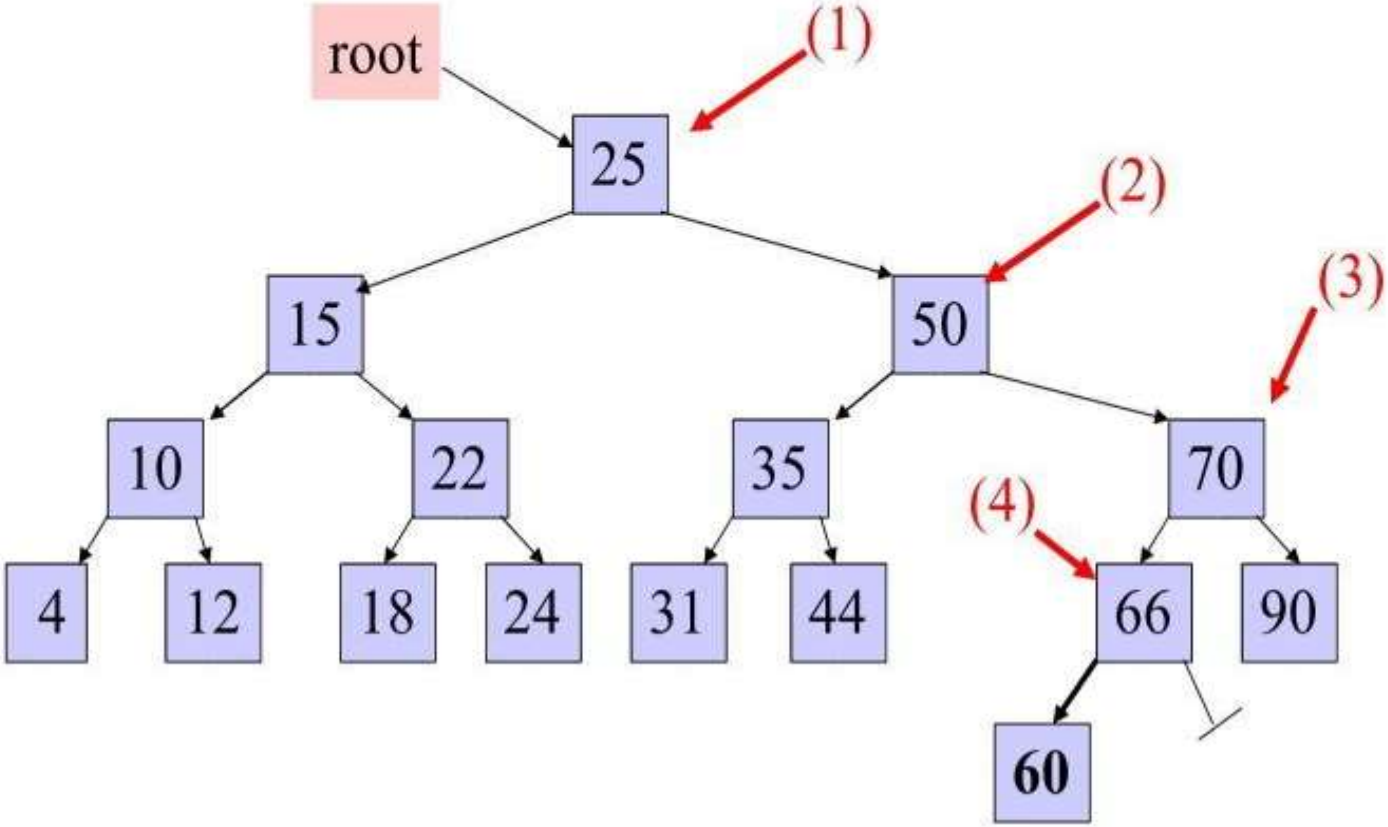


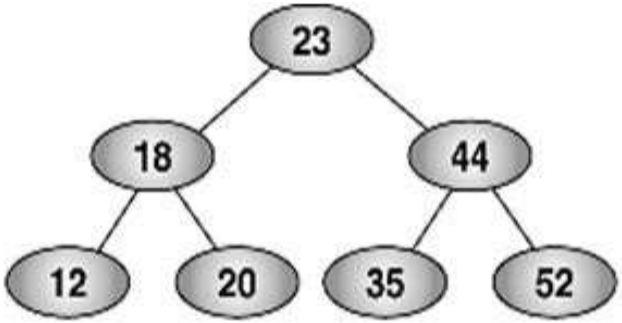


Trace of Recursive BST Insert

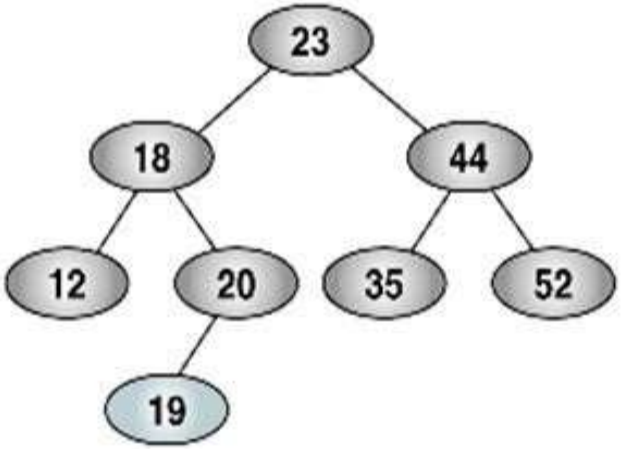
Example: insert 60 in the tree:

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

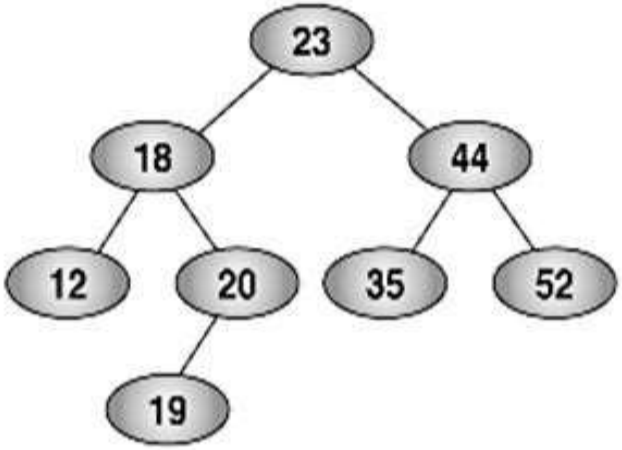




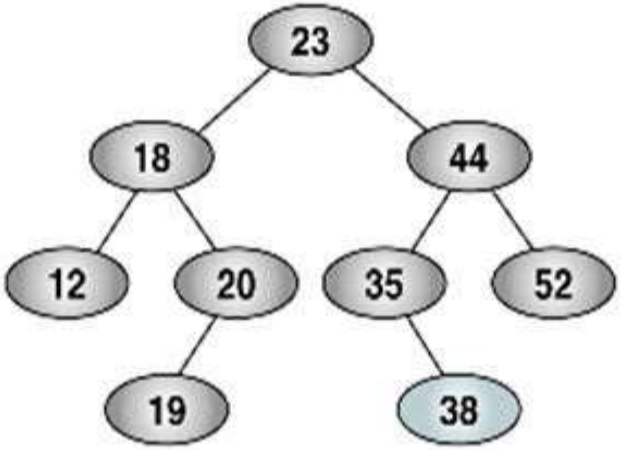
(a) Before inserting 19



(b) After inserting 19



(c) Before inserting 38



(d) After inserting 38

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.

Delete operation:

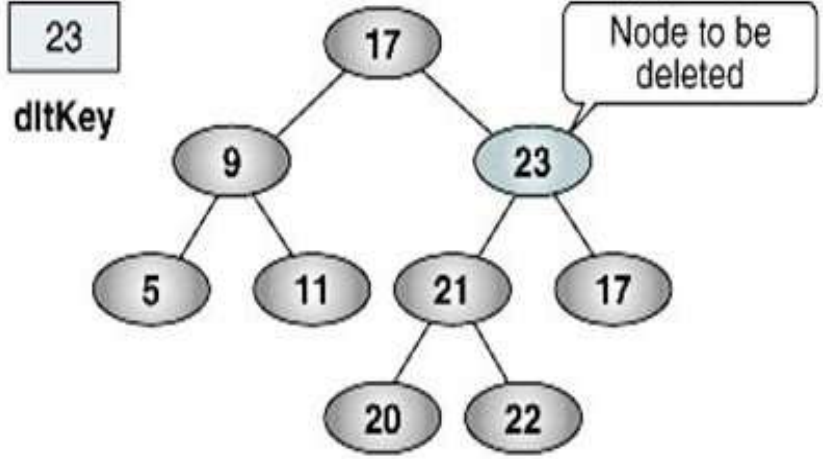
CASE 2 : - Node with one child

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

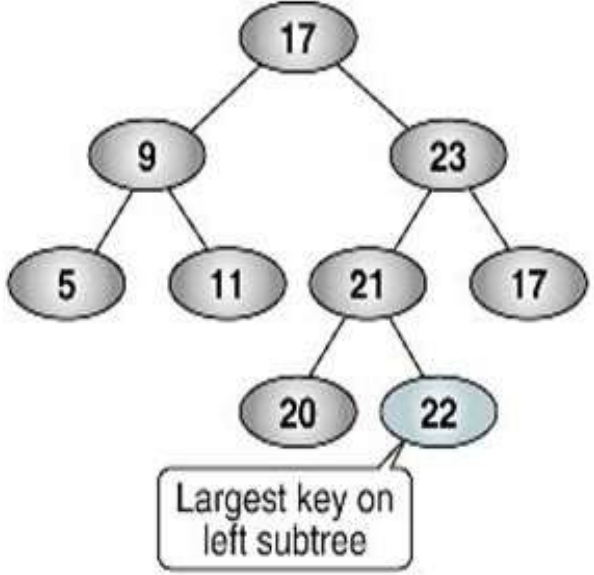
Delete operation:

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

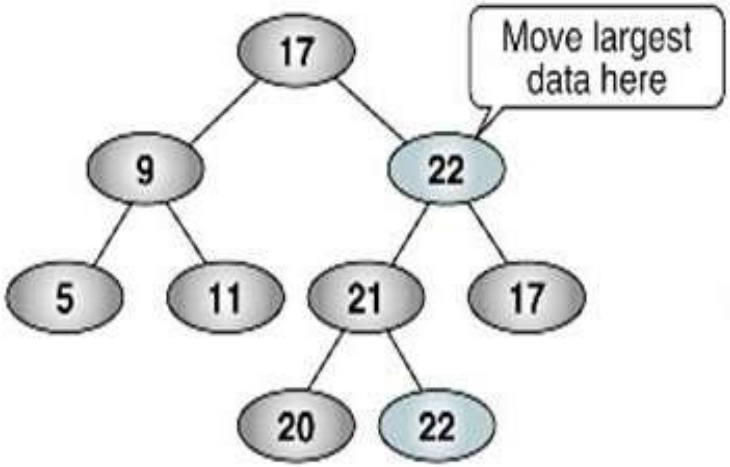
  It is difficult to delete a node which has two children. The general strategy is to replace the data of the node to be deleted with its smallest data of the right subtree and recursively delete that node.



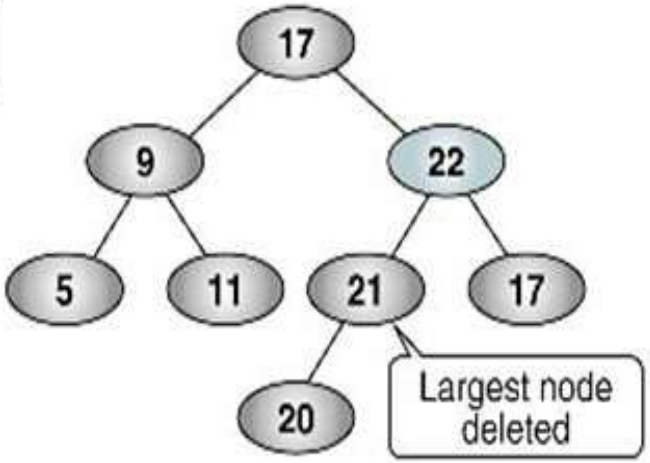
(a) Find dltKey



(b) Find largest



(c) Move largest data



(d) Delete largest node

Delete BST Test Cases

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

AVL TREES

- 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 TREE

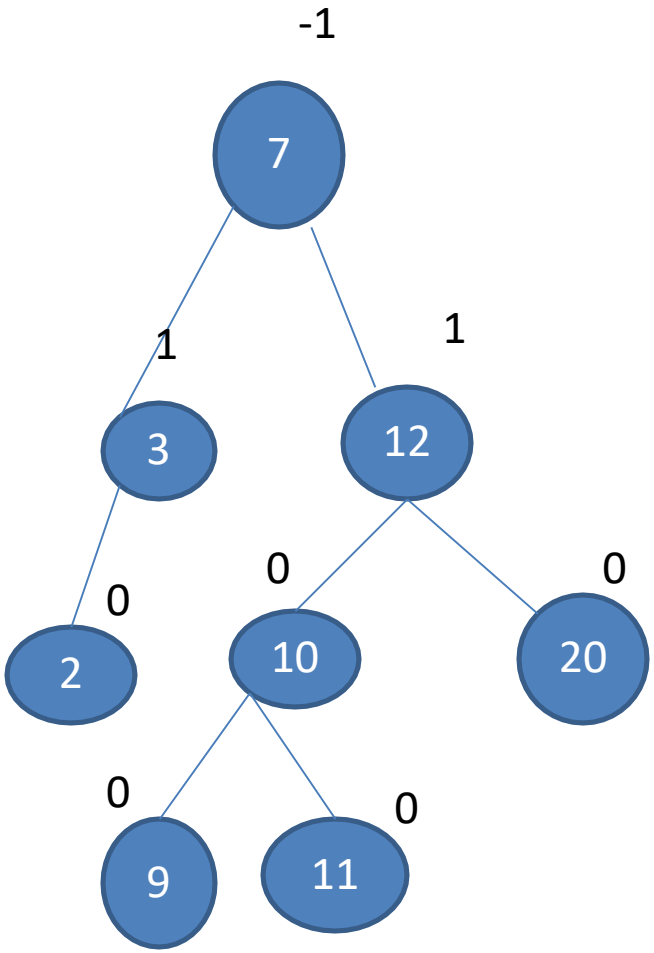
Definition

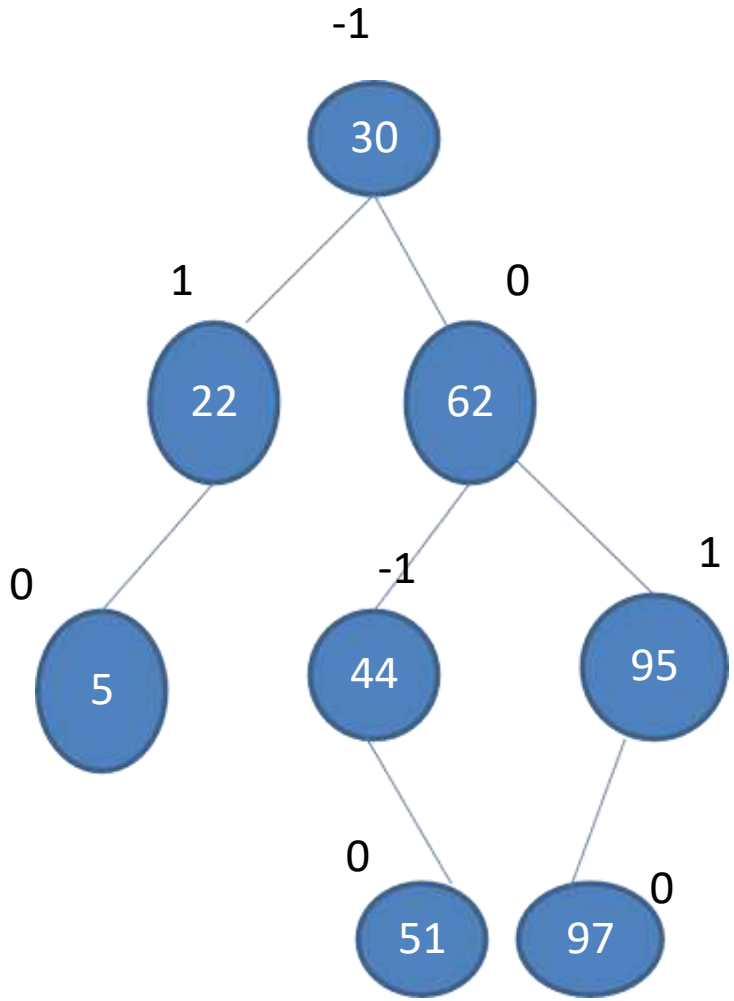
- Binary Search tree.
- If T is a nonempty binary Search tree with T_L and T_R as its left and right subtrees, then T is an AVL tree iff
  1. T_L and T_R are AVL trees, and
  2. $\mathbf{|h_L - h_R| \leq 1}$ where h_L and h_R are the heights of T_L and T_R, respectively

BALANCE FACTOR

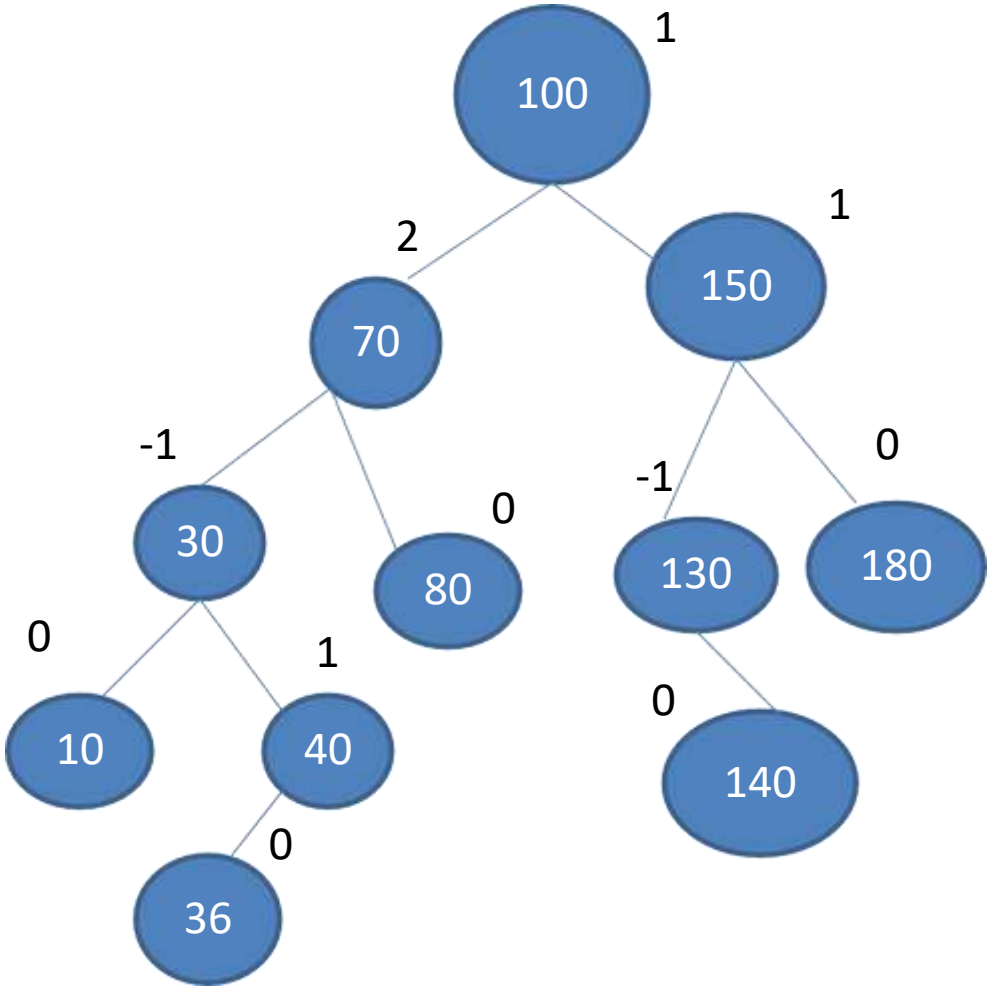
- AVL trees are normally represented using the linked representation
- To facilitate insertion and deletion, a balance factor (bf) is associated with each node.
- The balance factor bf(x) of a node x is defined as height(x→leftChild) – height(x→rightChild)
- Balance factor of each node in an AVL tree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

Eg with balance factors





Not an AVL TREE



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:

Inserting into an AVL Search Trees

- 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.

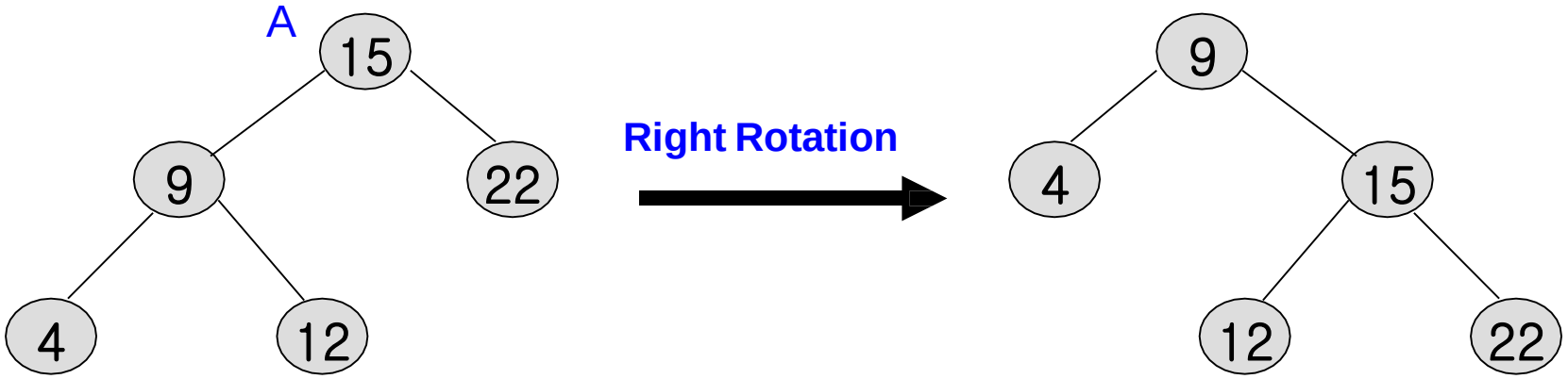
Single and Double Rotations

- 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 – Left Rotation

Definition

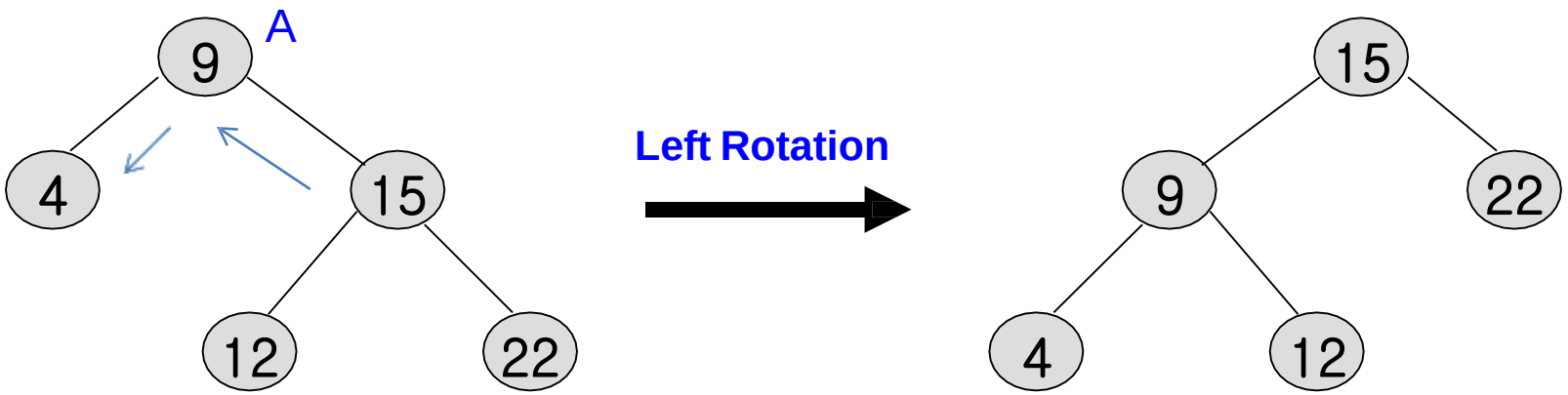
- **In a binary search tree, pushing a node A down and to the right to balance the tree.**
- **A's left child replaces A, and the left child's right child becomes A's left child.**



Right- Right Rotation

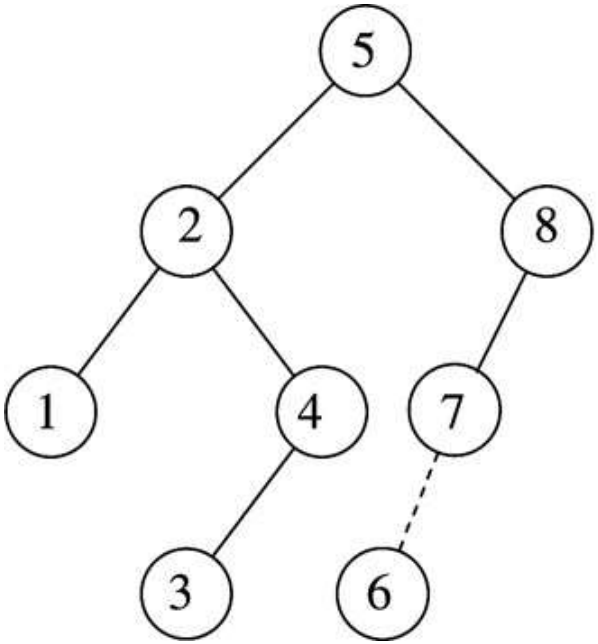
Definition

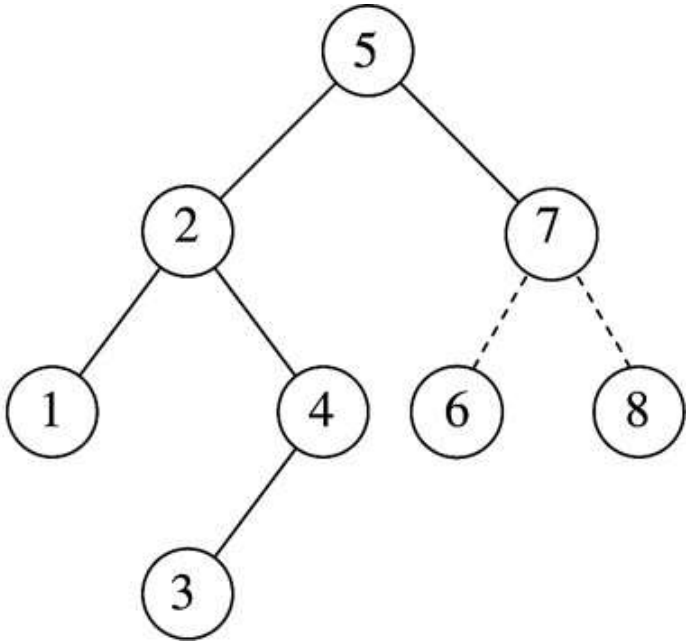
- **In a binary search tree, pushing a node A down and to the left to balance the tree.**
- **A's right child replaces A, and the right child's left child becomes A's right child.**



Single Rotation-Example I

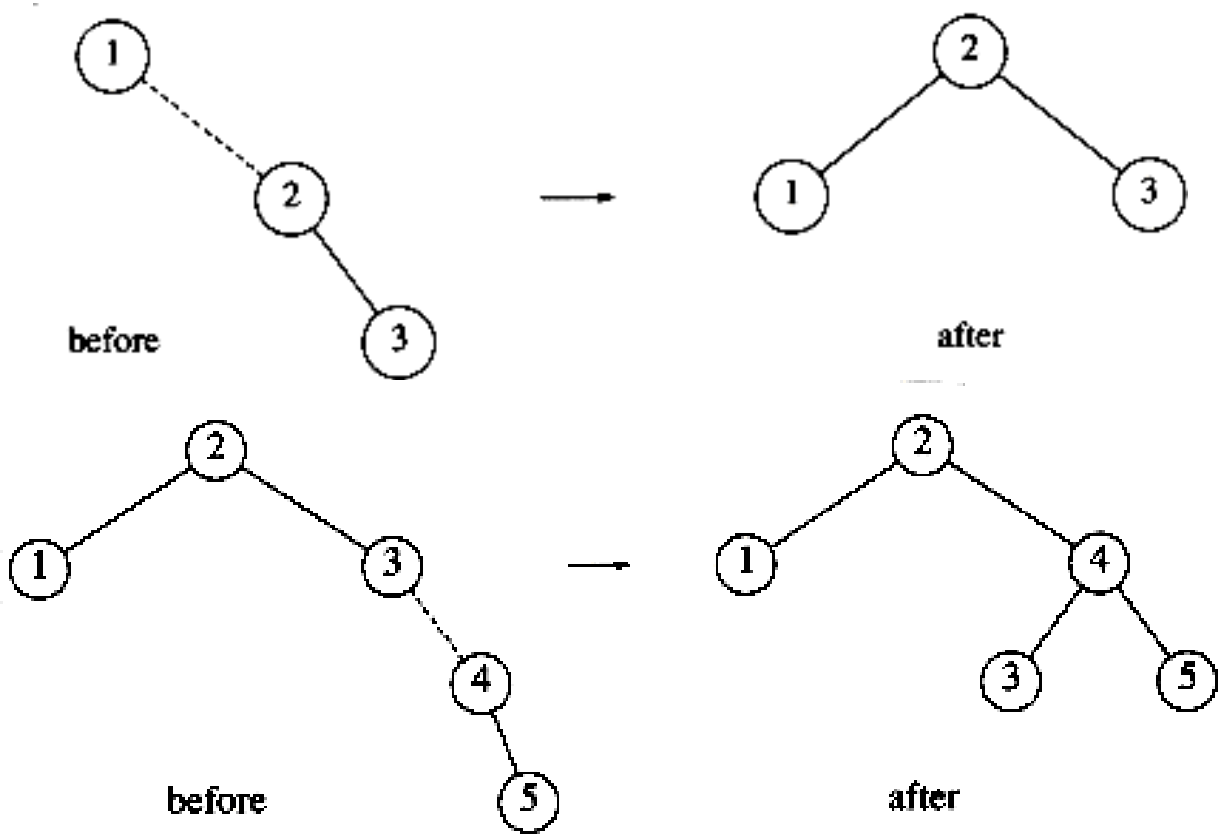
- **AVL property destroyed by insertion of 6, then fixed by a single rotation.**
- **BST node structure needs an additional field for height.**





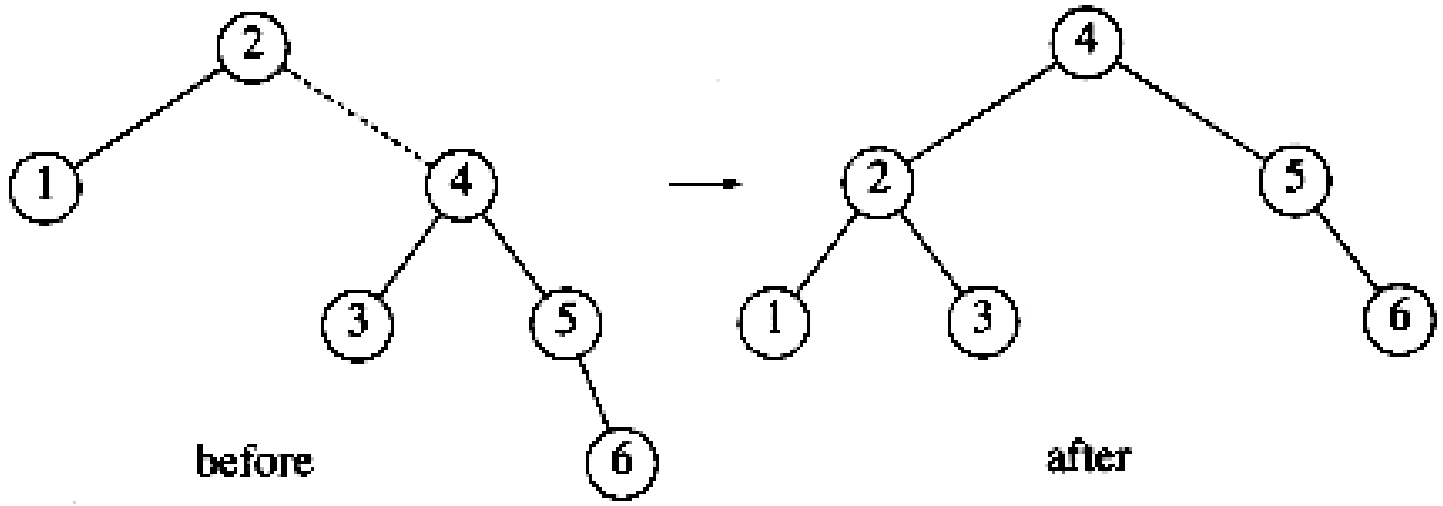
Single Rotation-Example II

- 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.

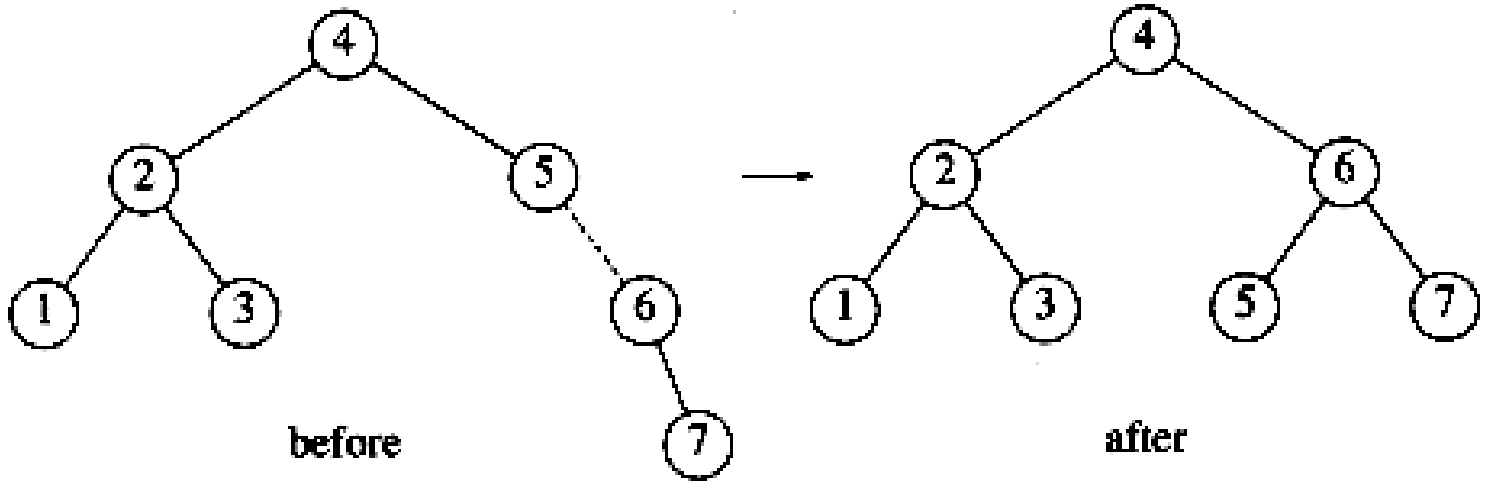


Single Rotation-Example III

Insert 6. Balance problem at the root. So a single rotation is performed.



Finally, Insert 7 causing another rotation.



Left -Right Rotation

Definition

- The left subtree's right child **will be** the changed root position.
- The left subtree's root **will be** the left child in the changed tree.
- The left subtrees's left child **will be** left child of the left subtree in the changed tree.
- The root node's left child's right child's left child **will be**
- the right child of the left child of the root in the changed tree
- The root node's left child's right child's right child **will be**

  the right child's left child in the changed tree.

Right -left Rotation

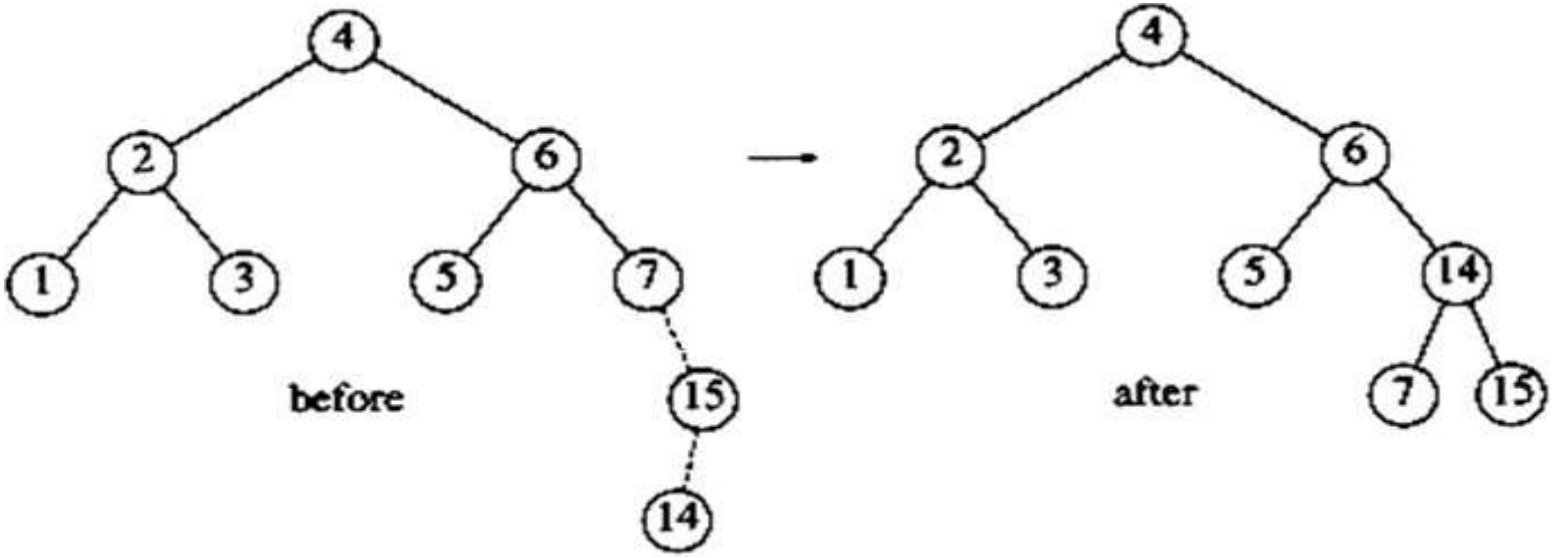
Definition

- The root node's right child 's left child **will be** the changed root position.
- The root node's right child **will be** the right child in the changed tree.
- The root nodes left child's left child **will be** left child of the changed tree.
- The root node's left child's right child's left child **will be**
- the right child of the left child of the root in the changed tree
- The root node's left child's right child's right child **will be**

  the right child's left child in the changed tree.

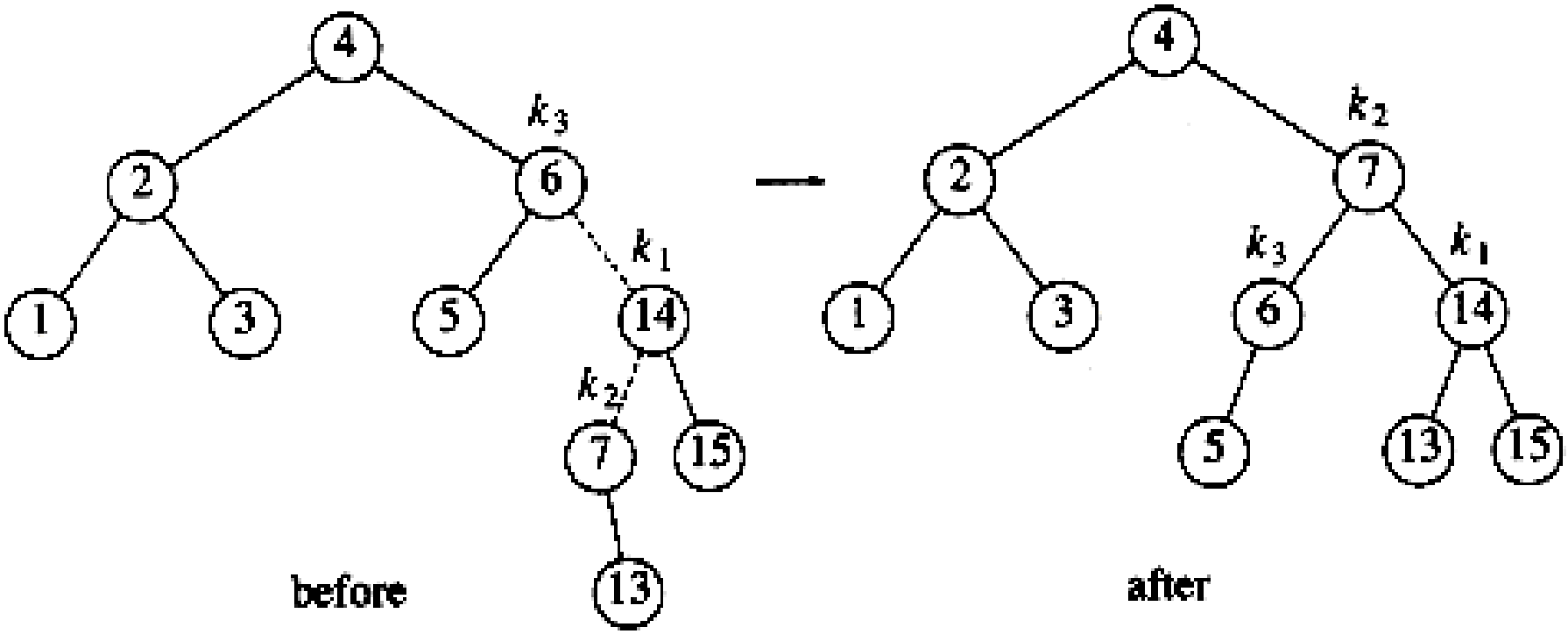
Double Rotation Example - I

- 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.



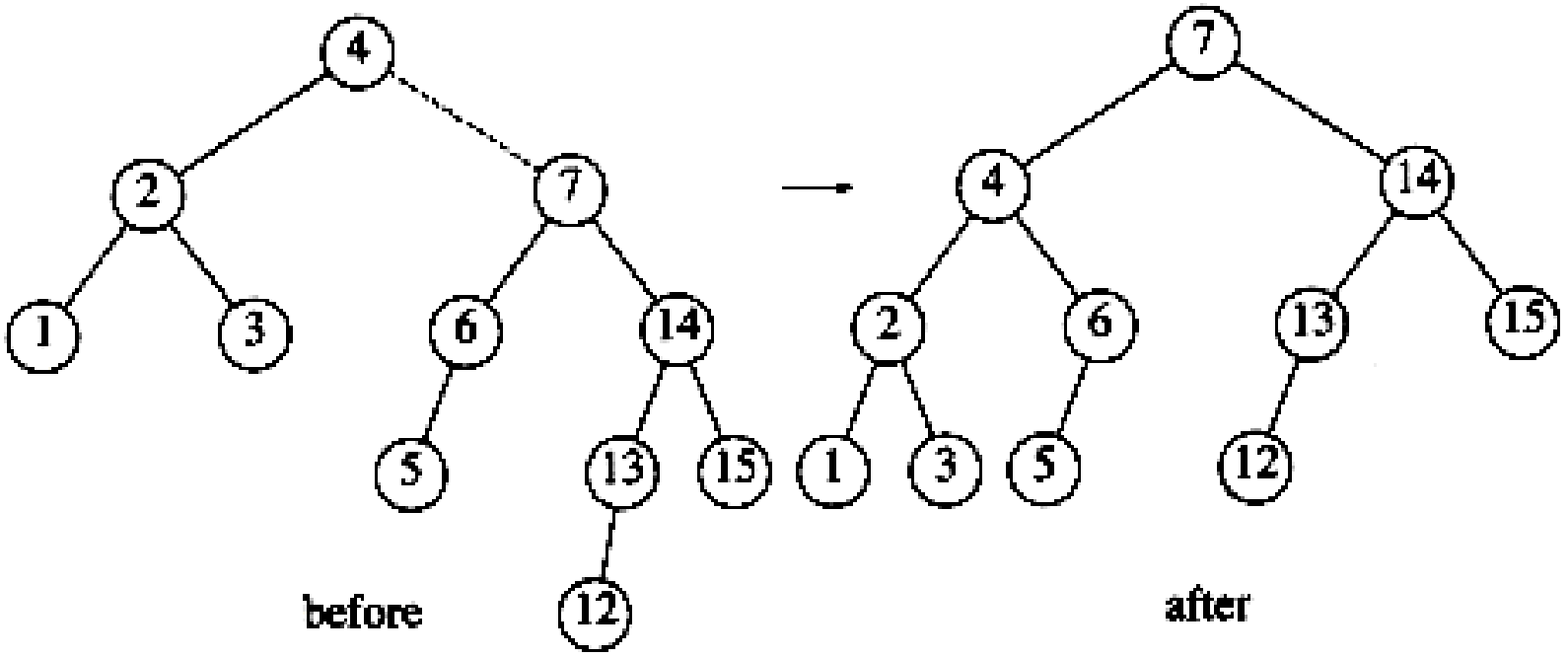
Double Rotation Example - II

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



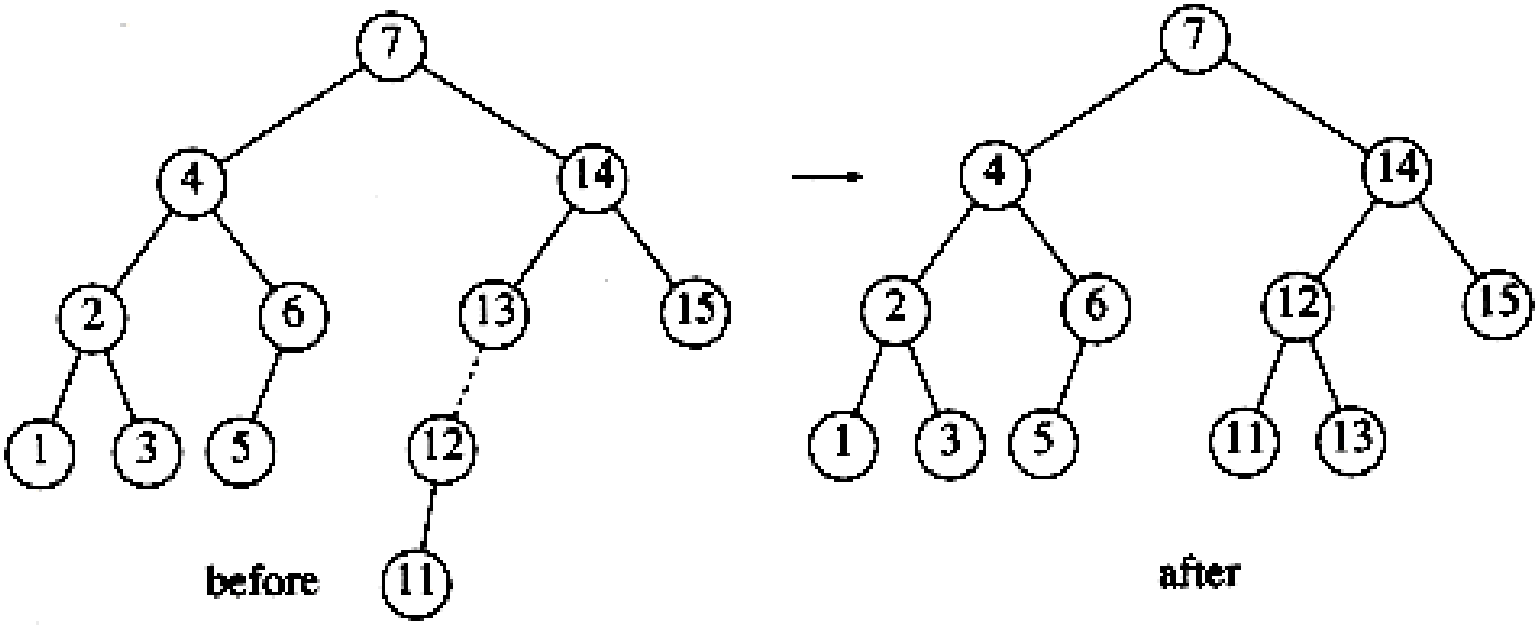
Double Rotation Example - III

- **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.**



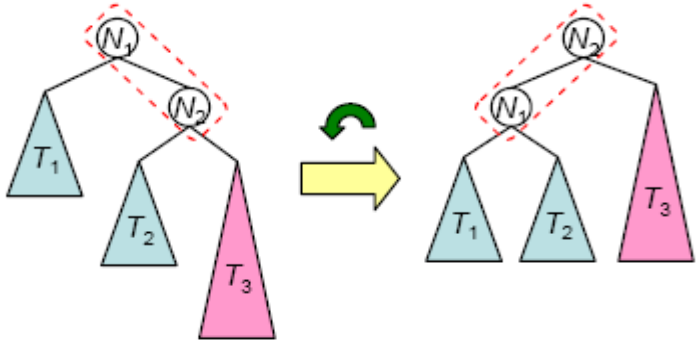
Double Rotation Example - IV

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



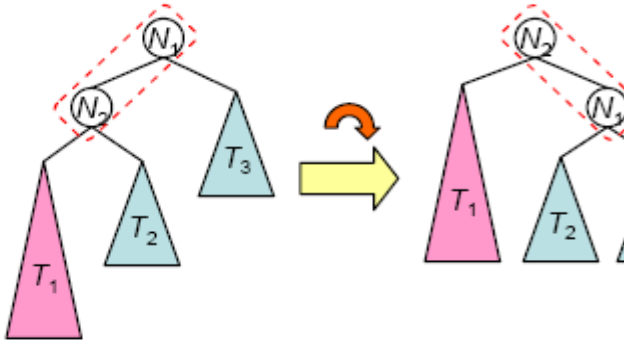
Case 1: insertion to *right* subtree of *right* child

Solution: *Left* rotation



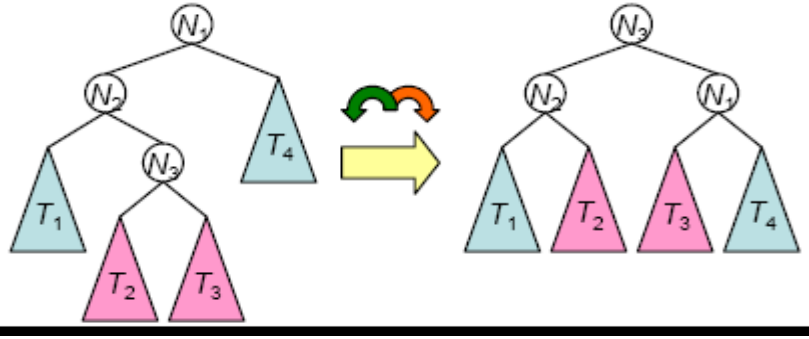
Case 2: insertion to *le* subtree of *left* child

Solution: *Right* rotation



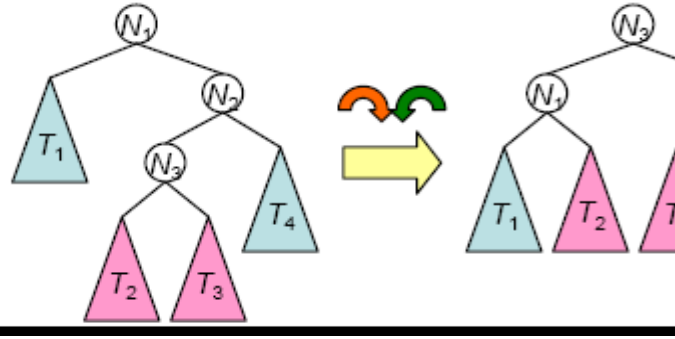
Case 3: insertion to *right* subtree of *left* child

Solution: *Left-right* rotation



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

Solution: *Right-left* rota

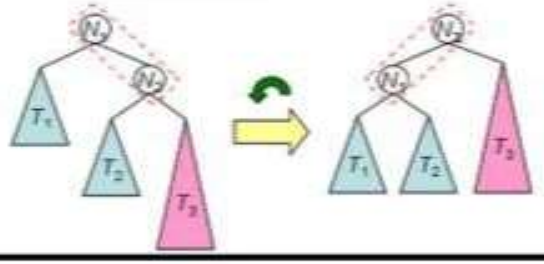


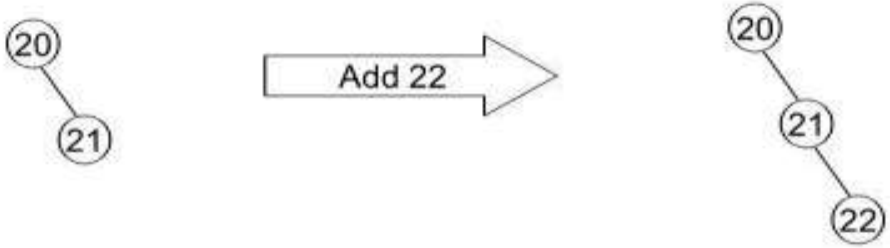
examples

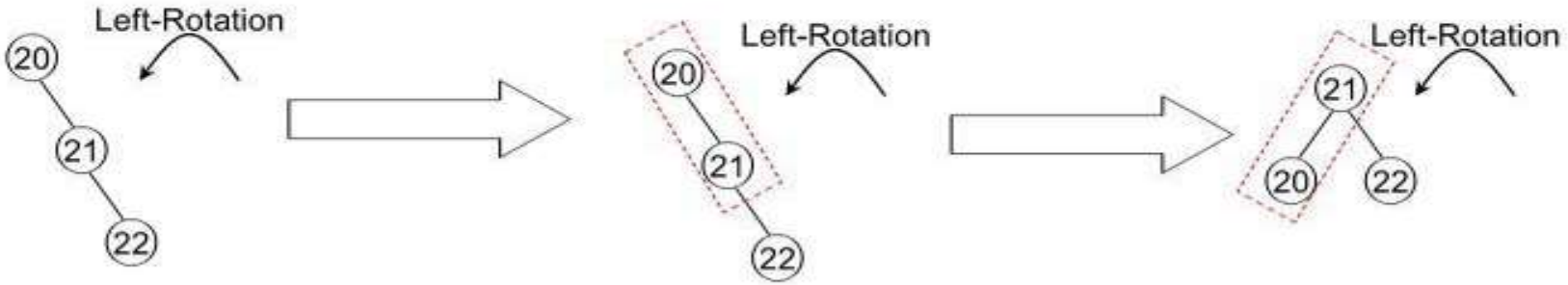
Left-Rotation

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

Solution: *Left* rotation



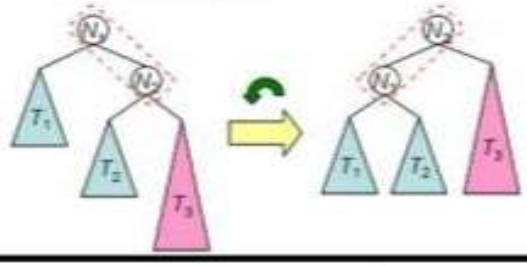


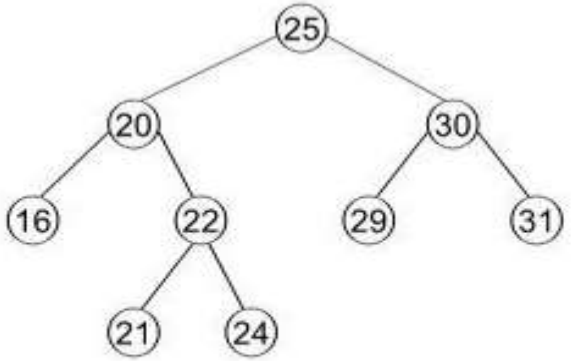


Left-Rotation

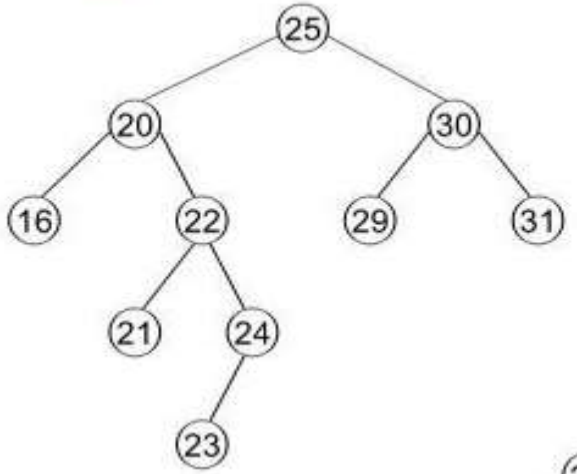
Case 1: insertion to *right* subtree of *right* child

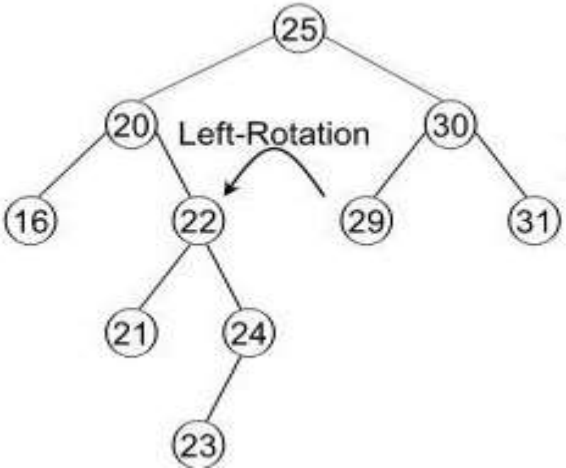
Solution: *Left* rotation

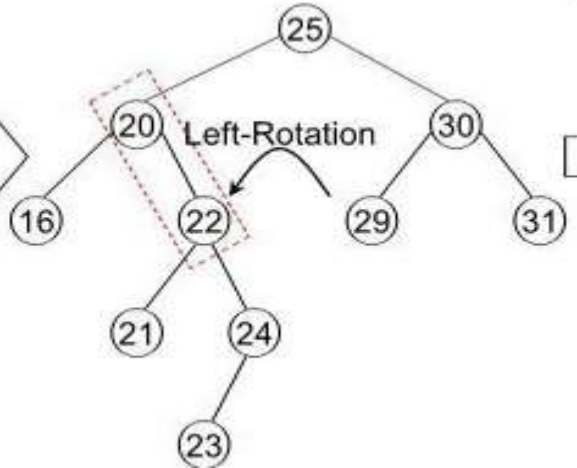


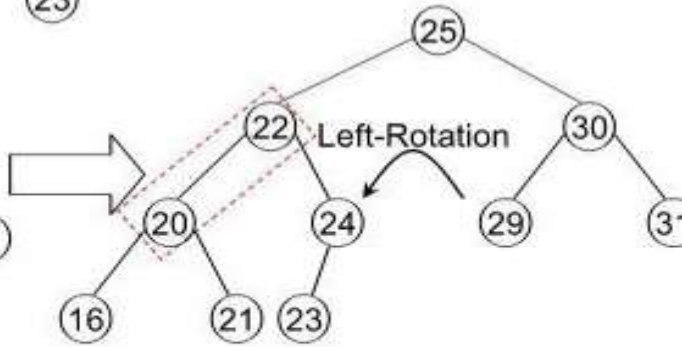


Add 23





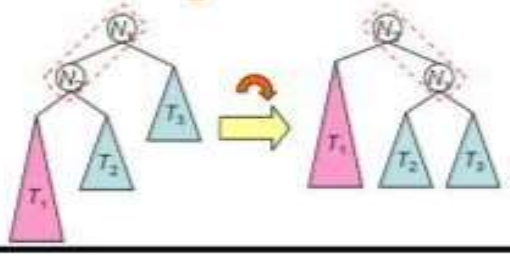


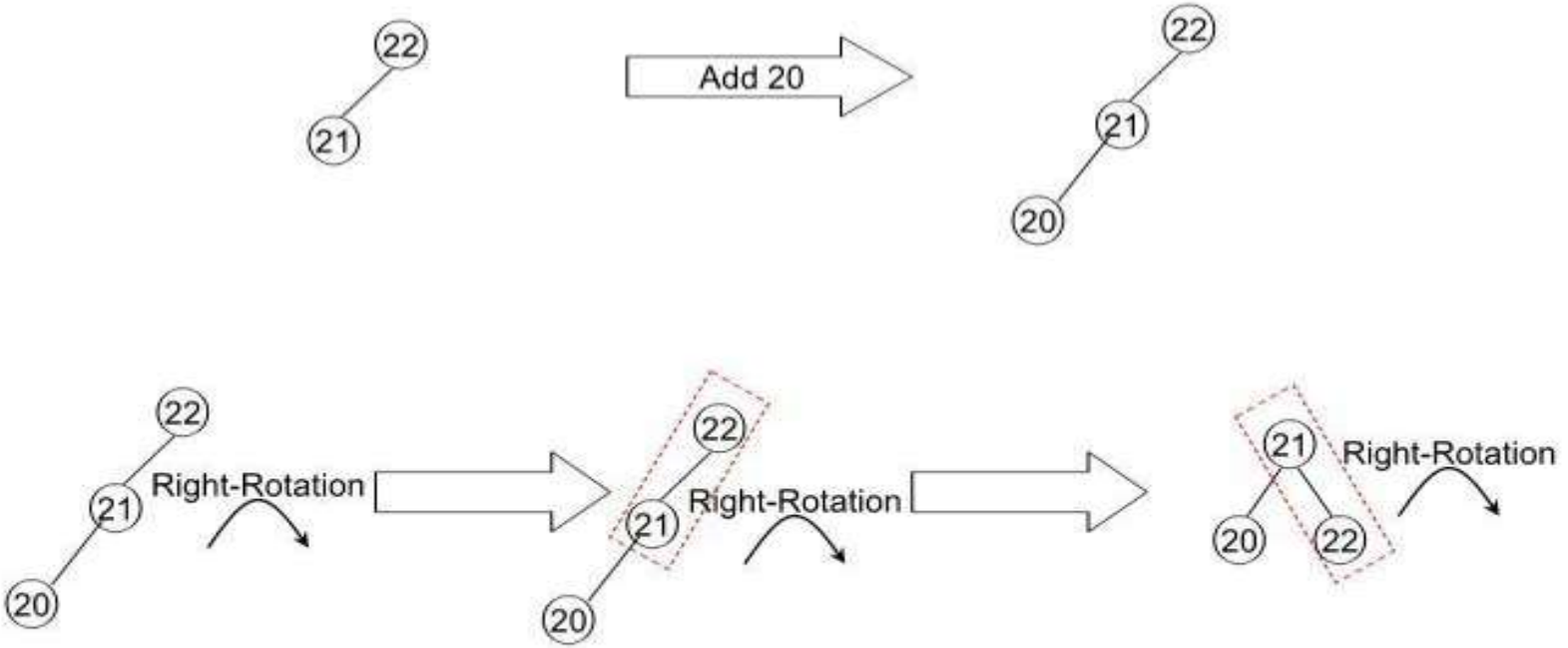


Right-Rotation

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

Solution: *Right* rotation

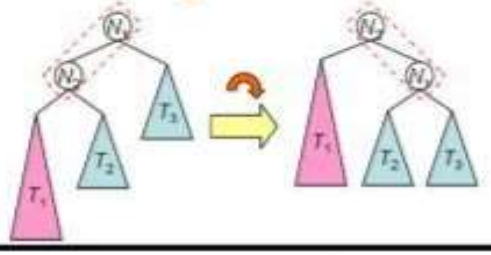


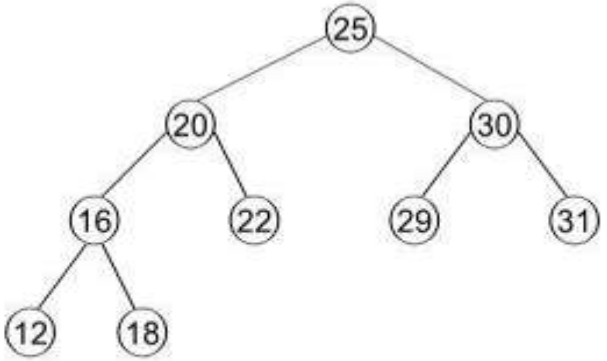


Right-Rotation

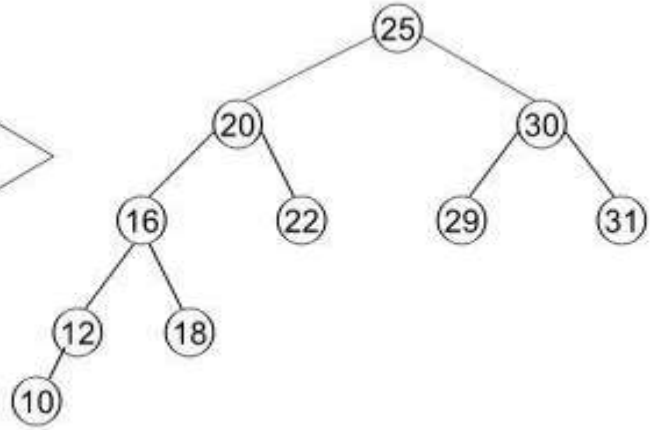
Case 2: insertion to *left* subtree of *left* child

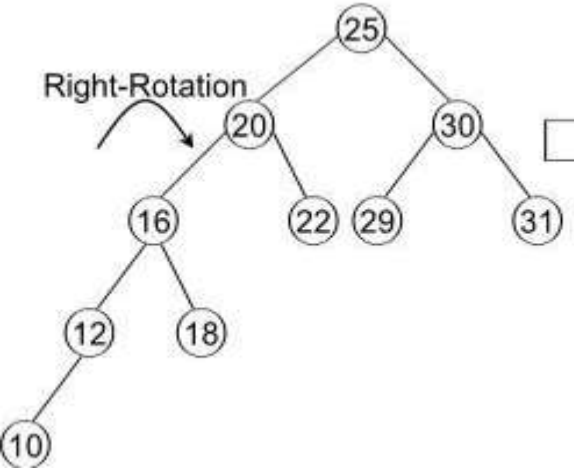
Solution: *Right* rotation

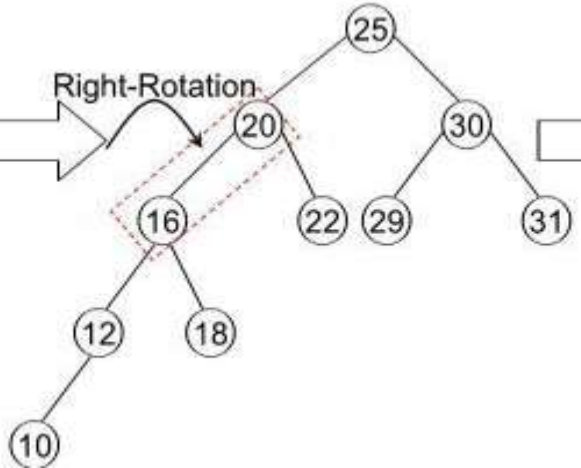


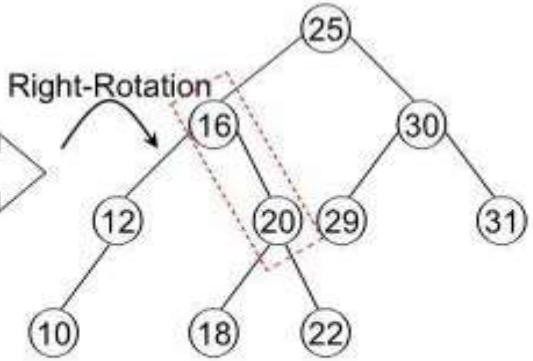


Add 10





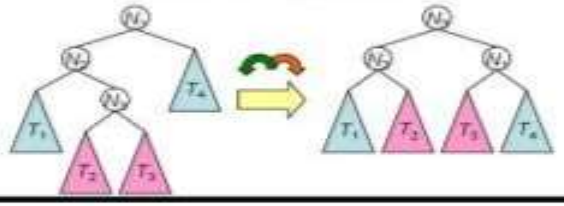


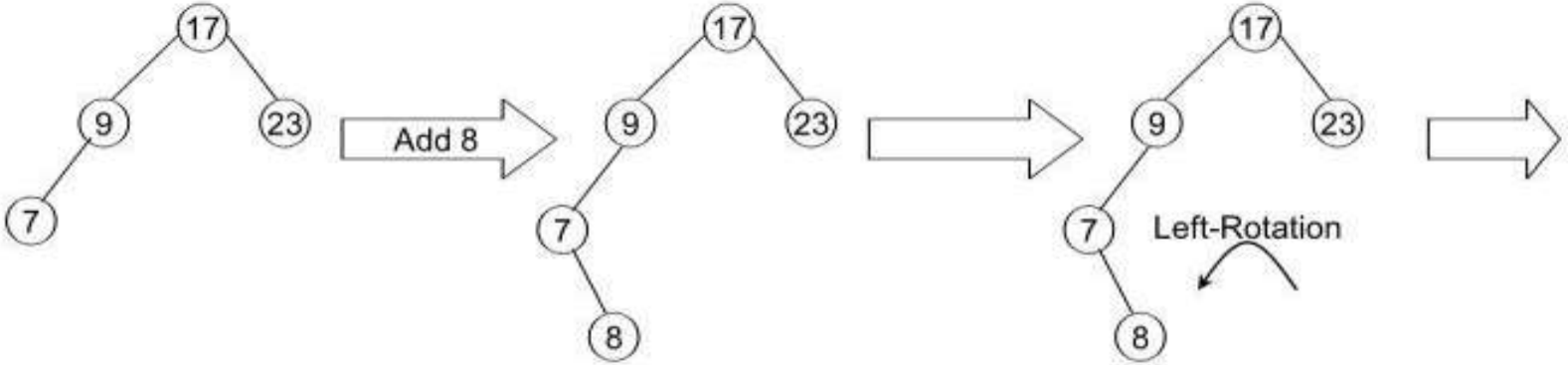


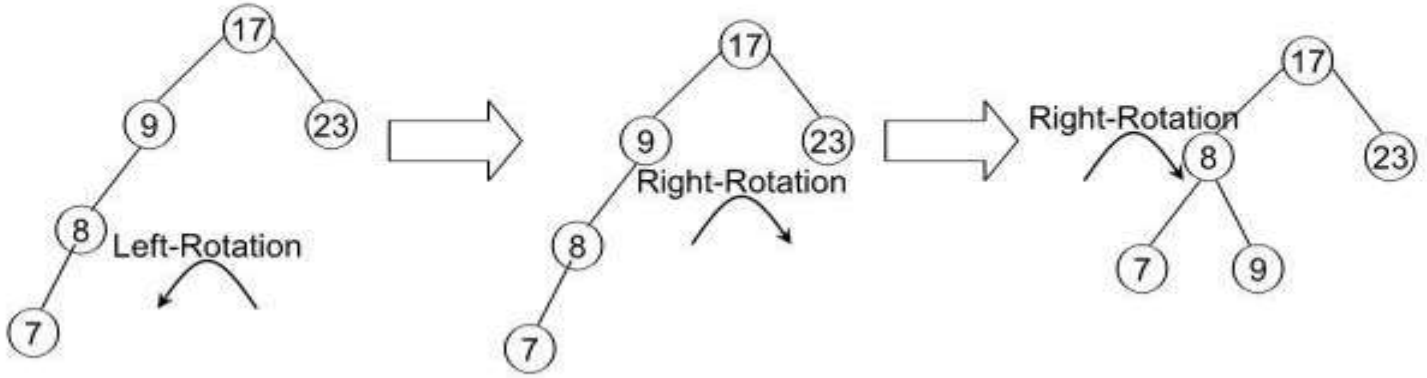
Left-Right Rotation

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

Solution: *Left-right* rotation



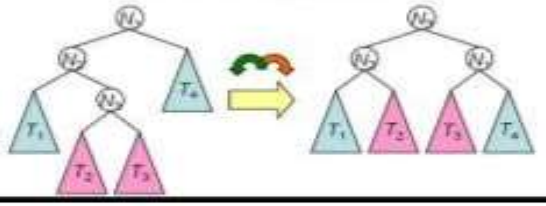


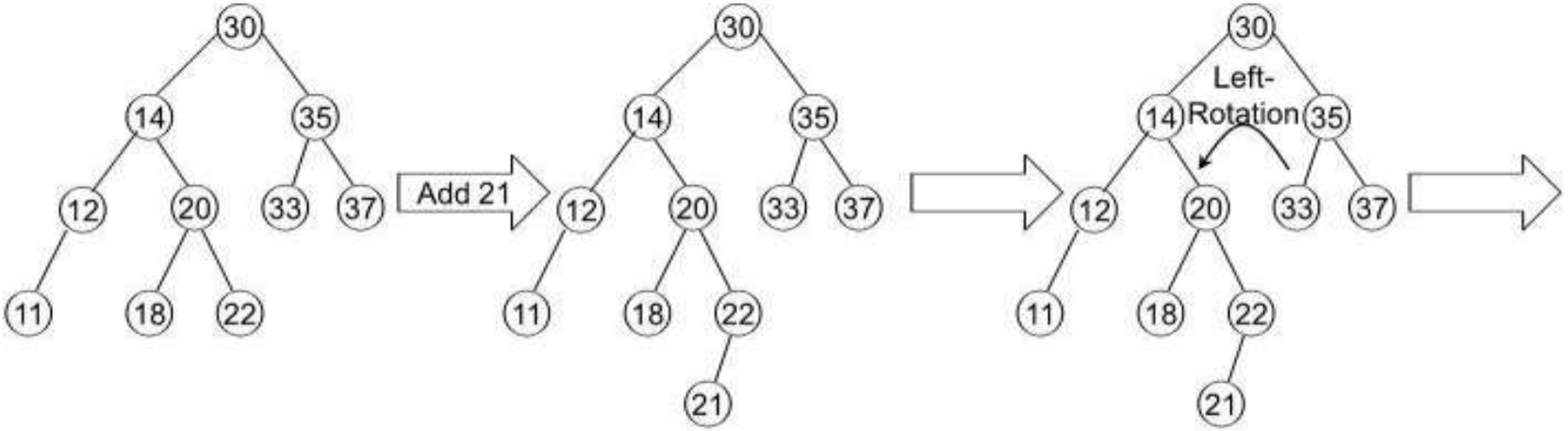


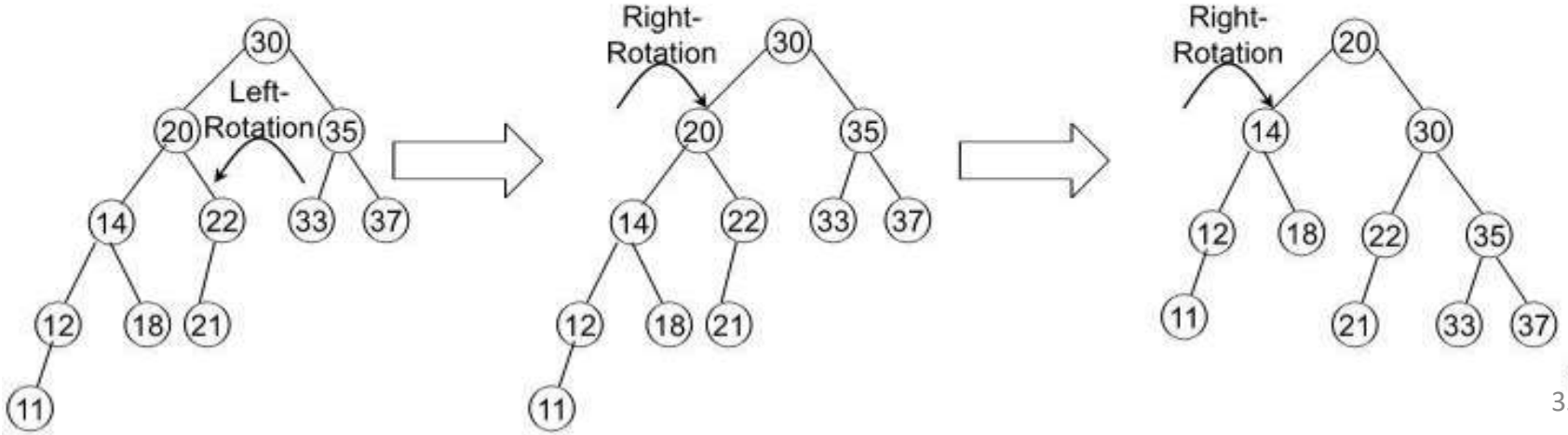
Left-Right Rotation

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

Solution: *Left-right* rotation



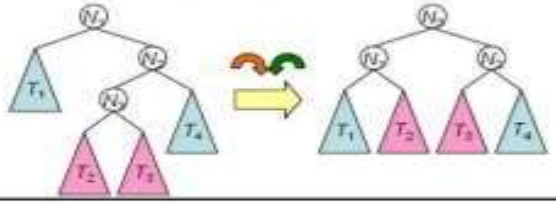




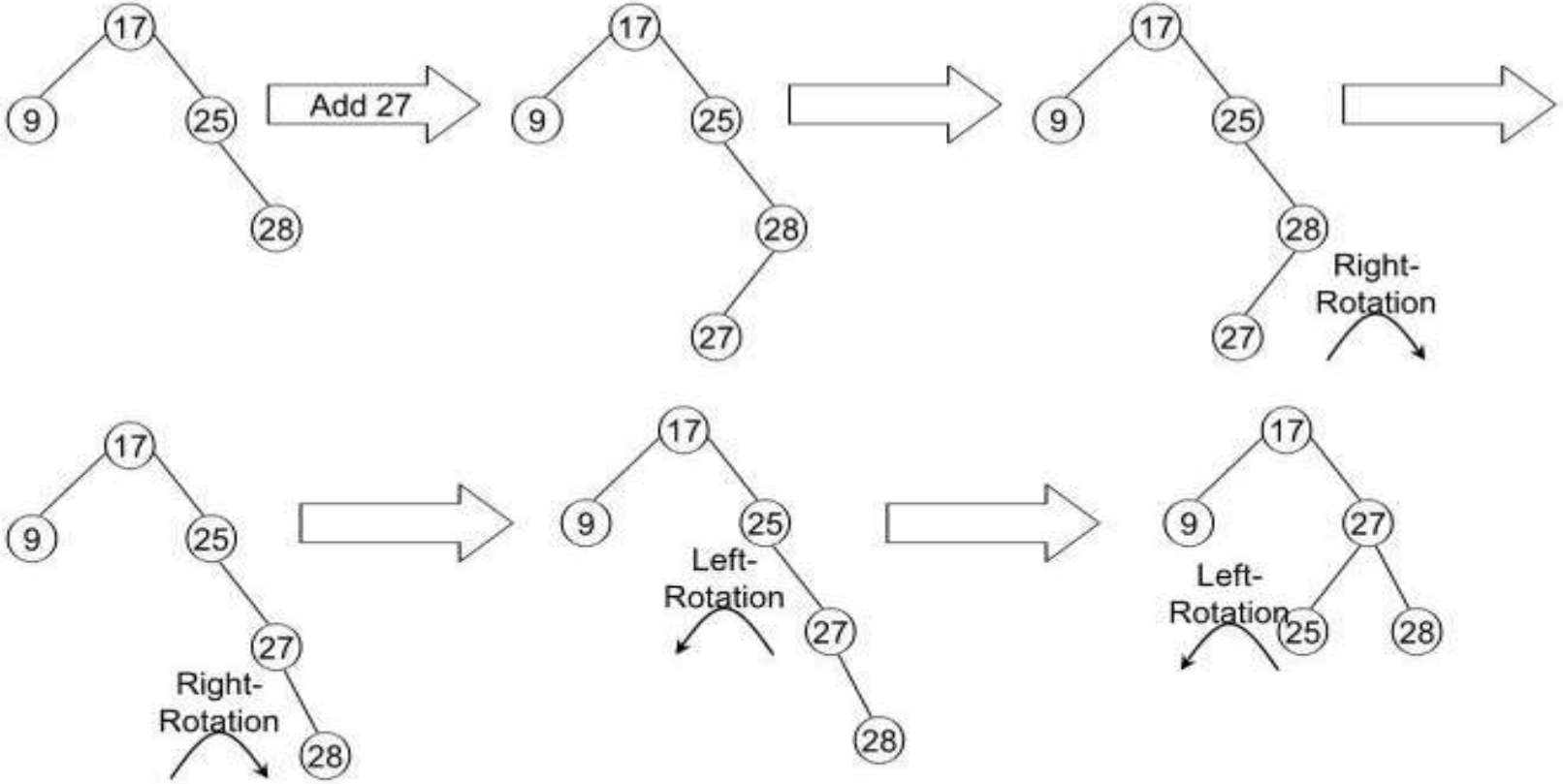
Right-Left Rotation

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

Solution: *Right-left* rotation

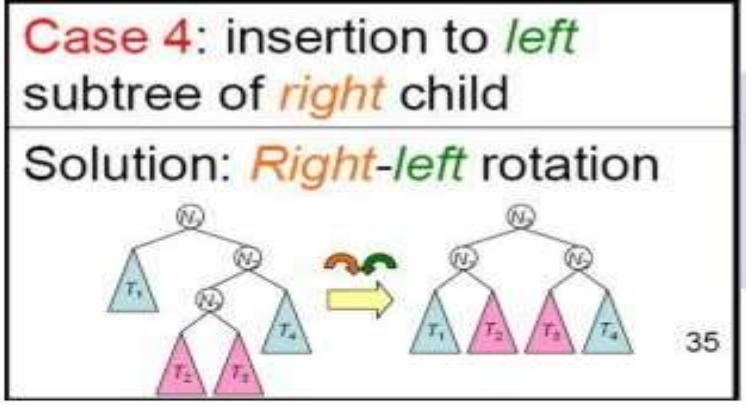


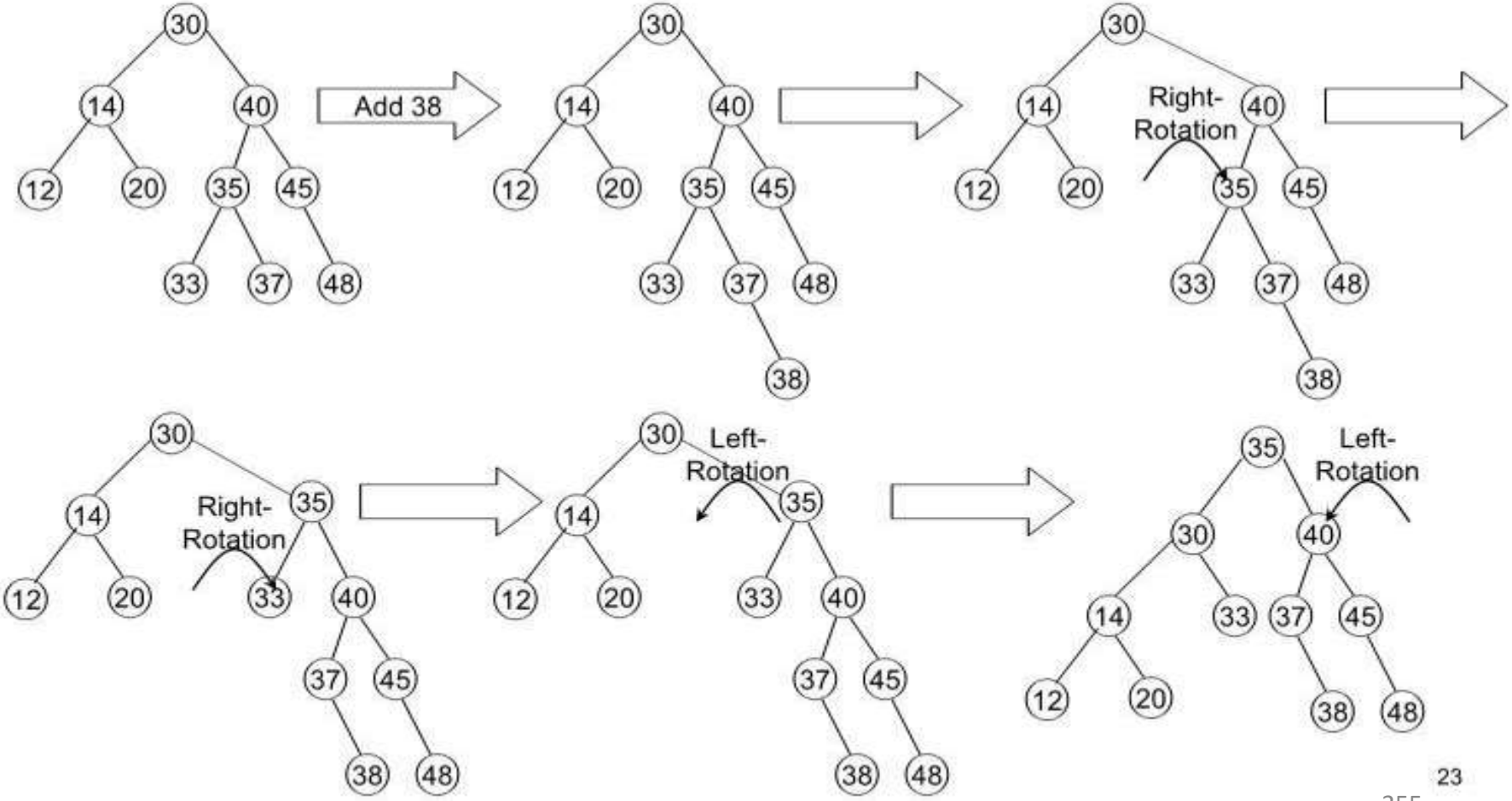
35



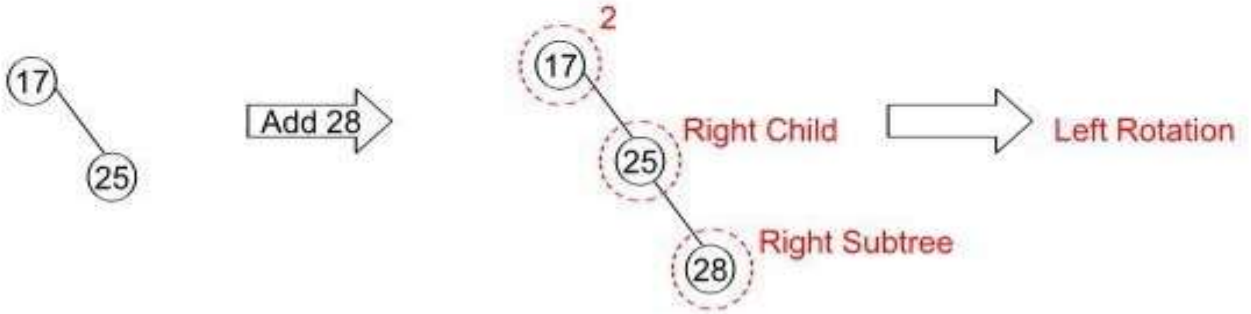
22

Right-Left Rotation



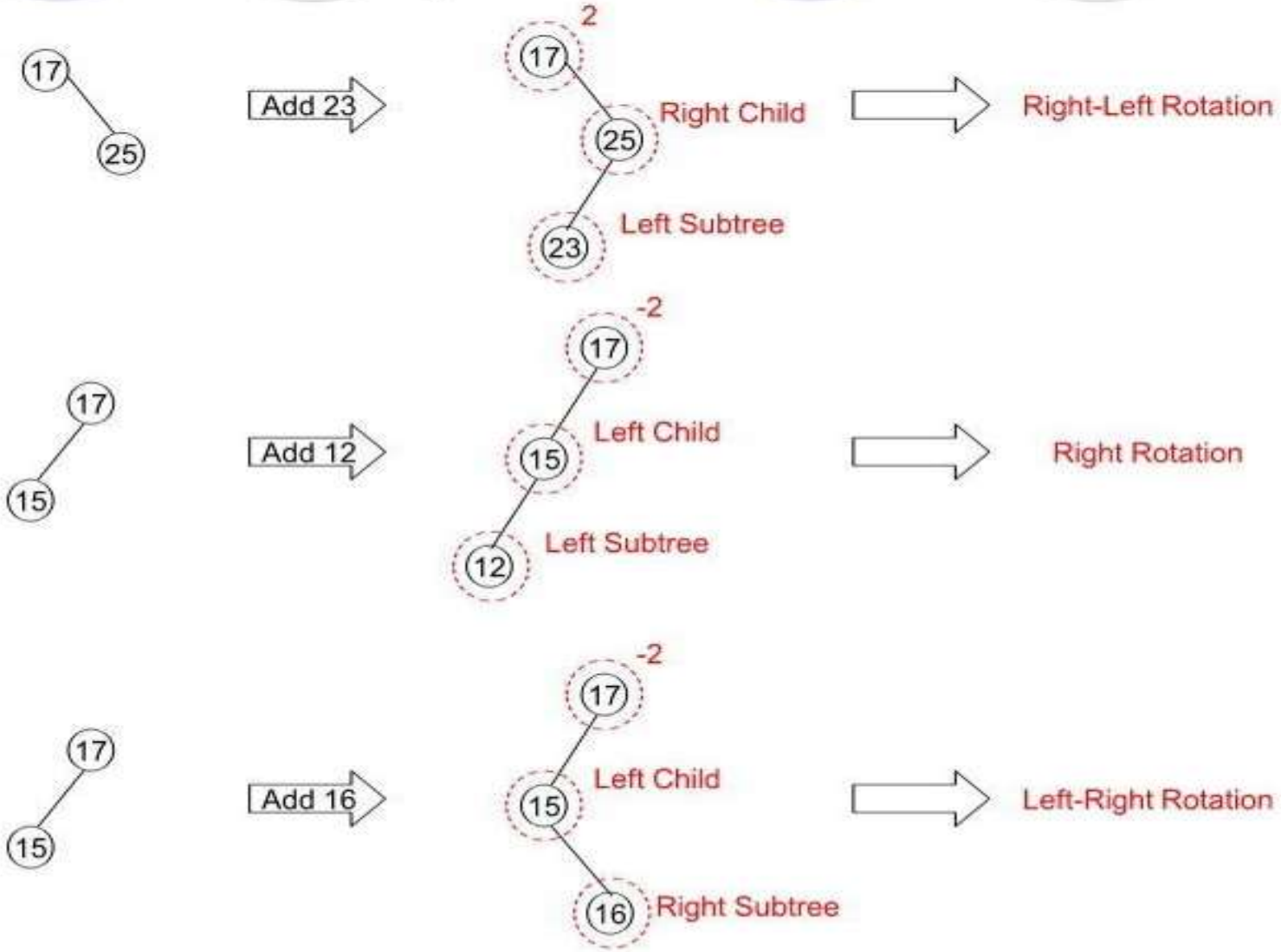


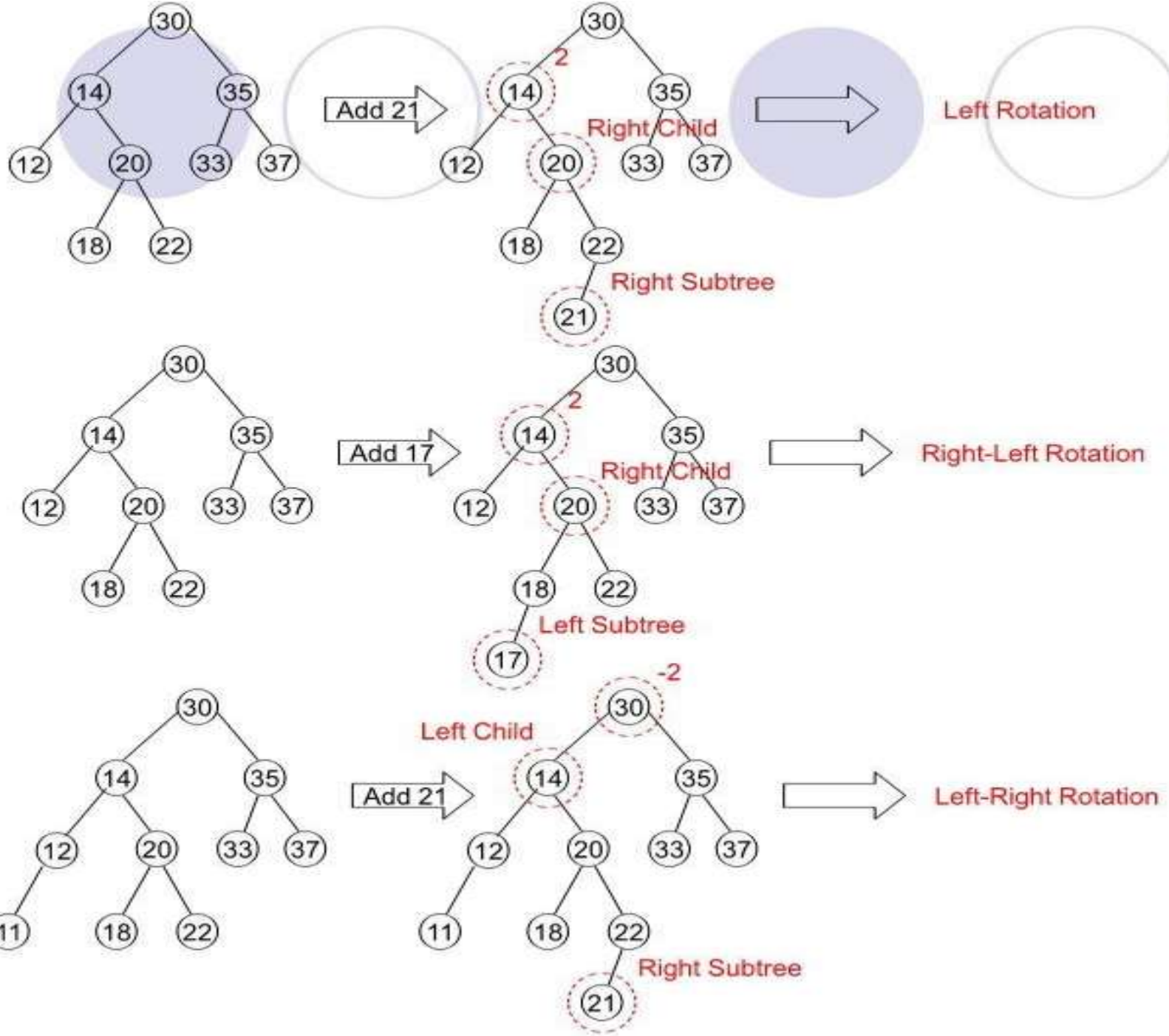
How to identify rotations?



- 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)

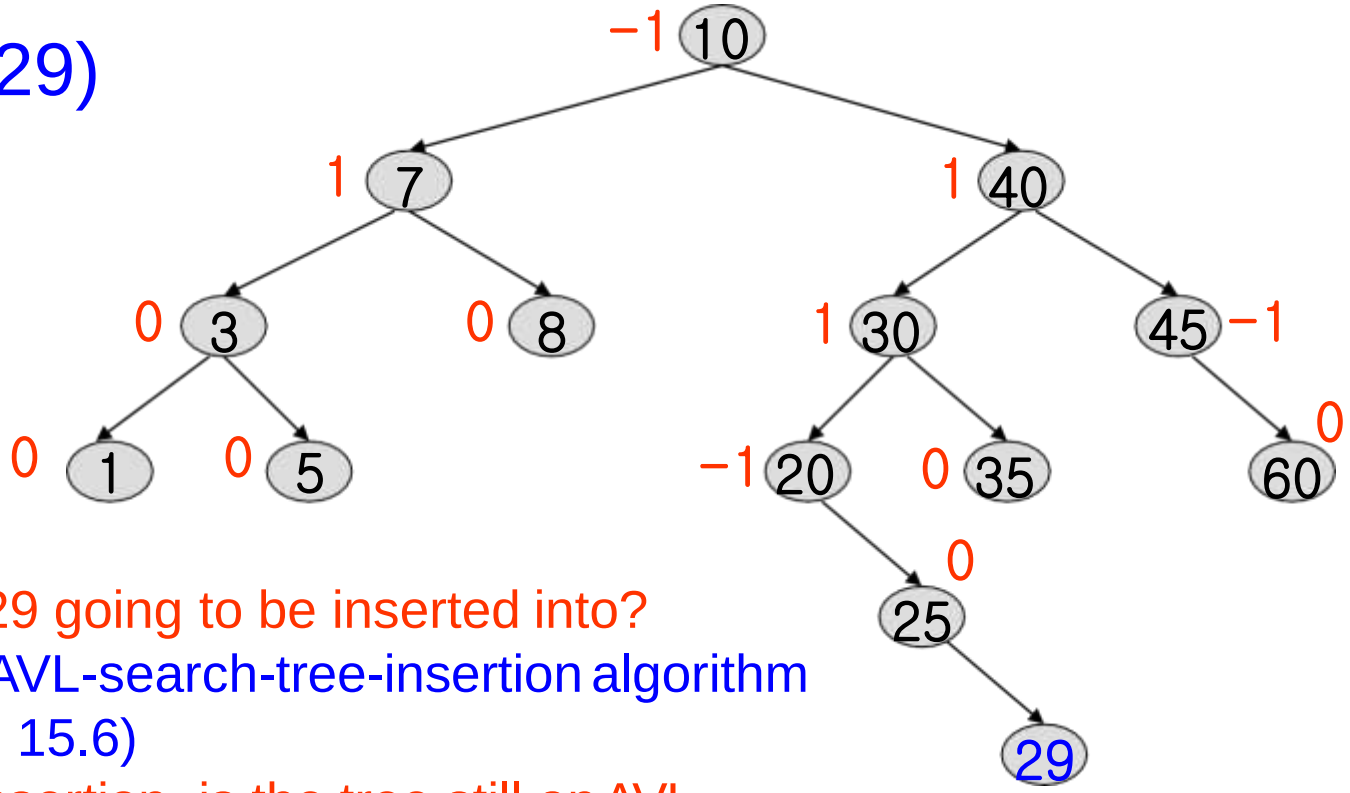
How to identify rotations?




30
14
35
12
20
33
37
18
22
Add 21
2
Right Child
21
Right Subtree
Left Rotation
Add 17
Left Subtree
17
Right-Left Rotation
-2
Left Child
11
Add 21
Right Subtree
Left-Right Rotation

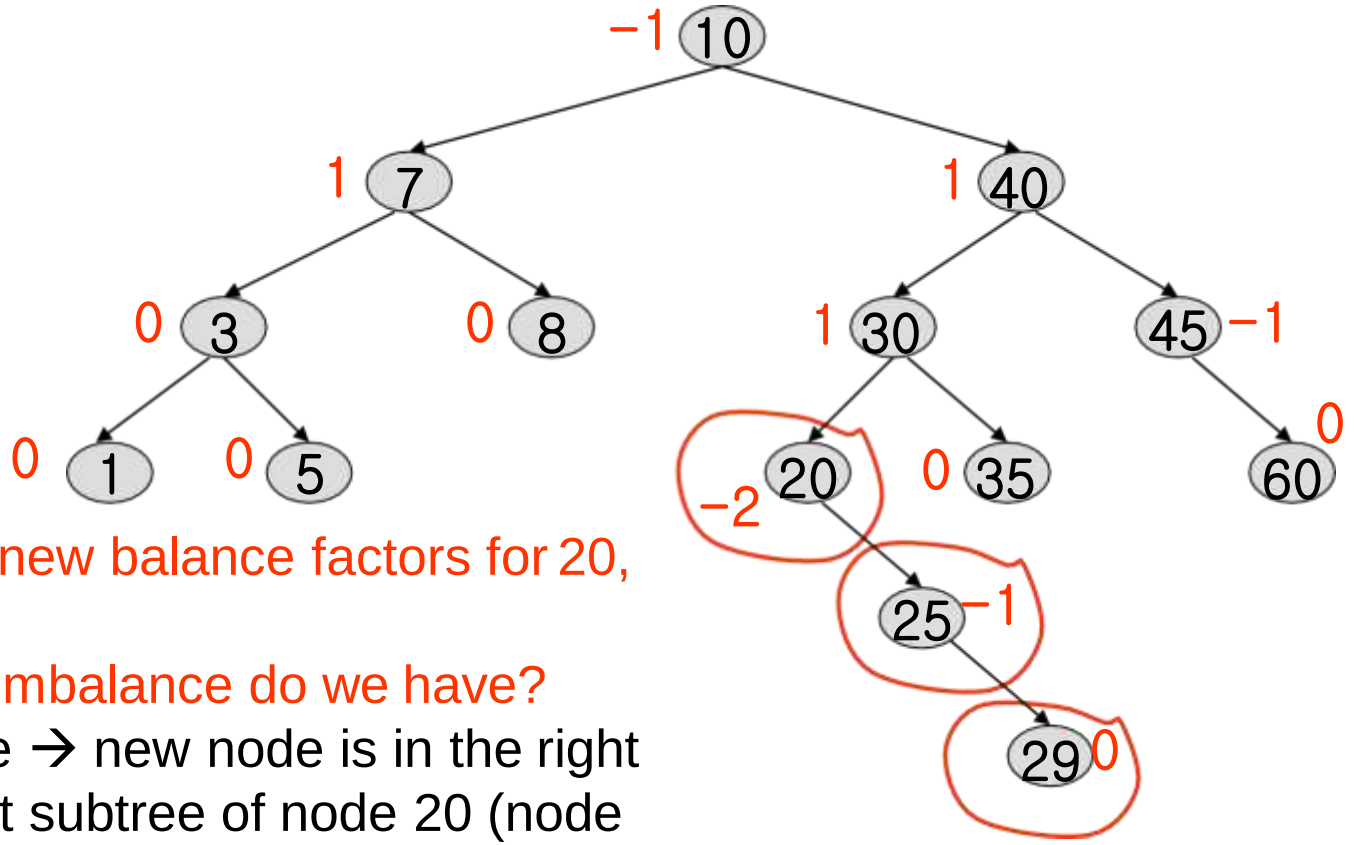
Inserting into an AVL Search Tree

Insert(29)



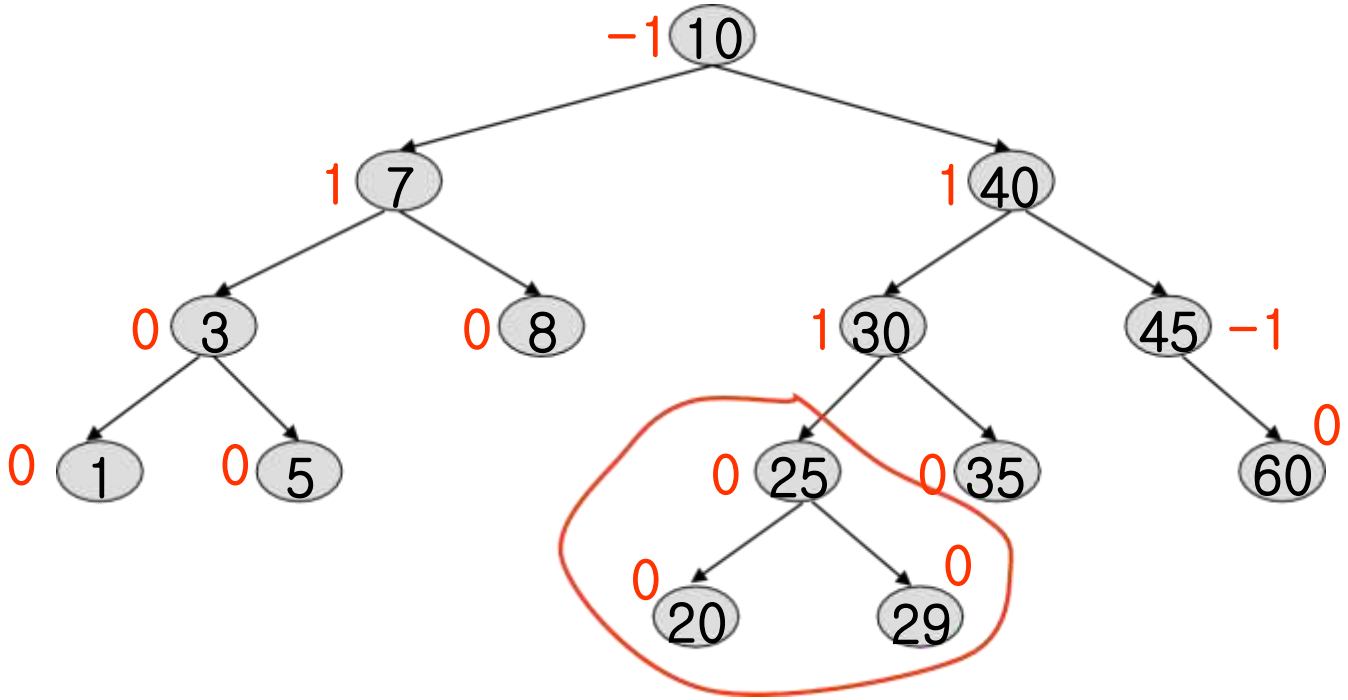
- Where is 29 going to be inserted into?
  - use the AVL-search-tree-insertion algorithm in Figure 15.6)
- After the insertion, is the tree still an AVL search tree? (i.e., still balanced?)

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?

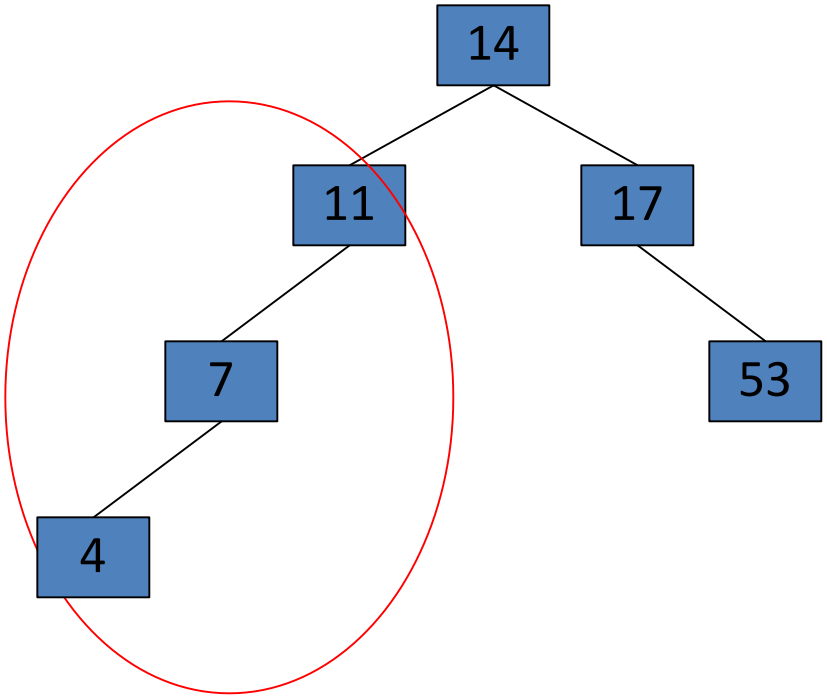
Deletion from an AVL Search Tree

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.**

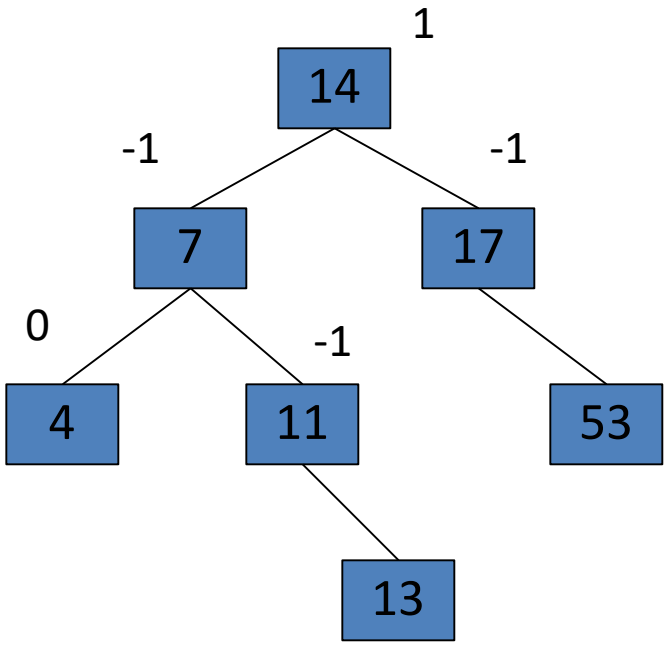
AVL Tree Example:

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



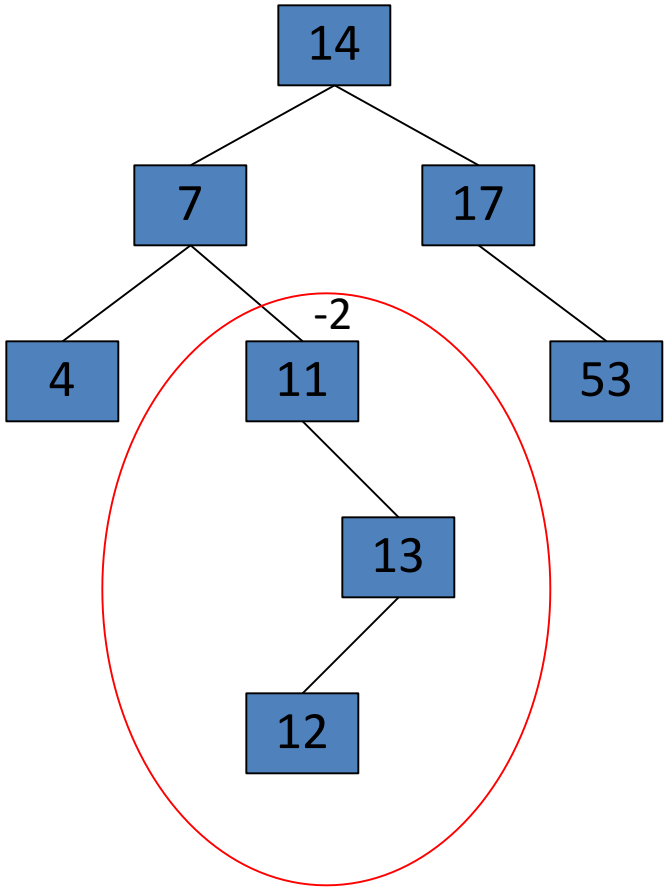
AVL Tree Example:

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



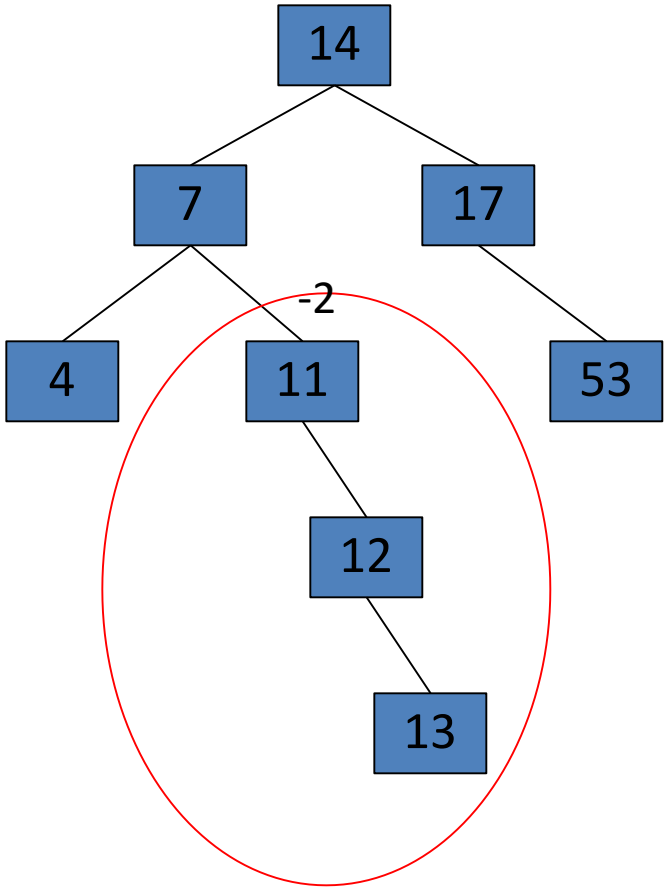
AVL Tree Example:

- **Now insert 12**



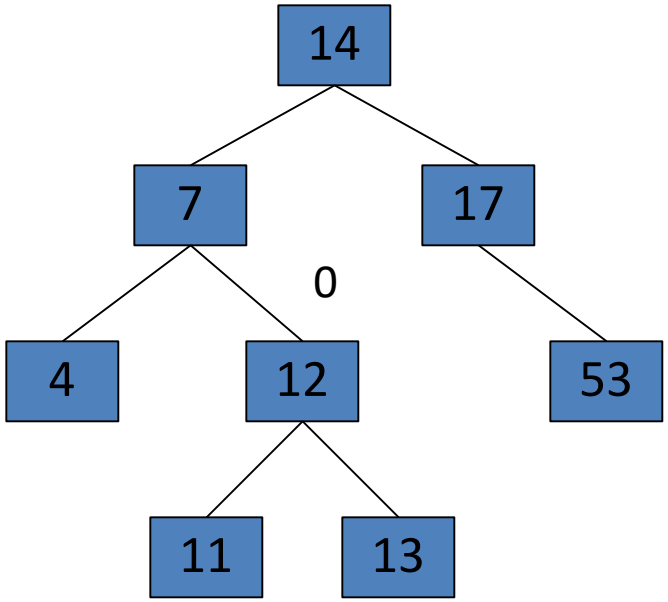
AVL Tree Example:

- **Now insert 12**



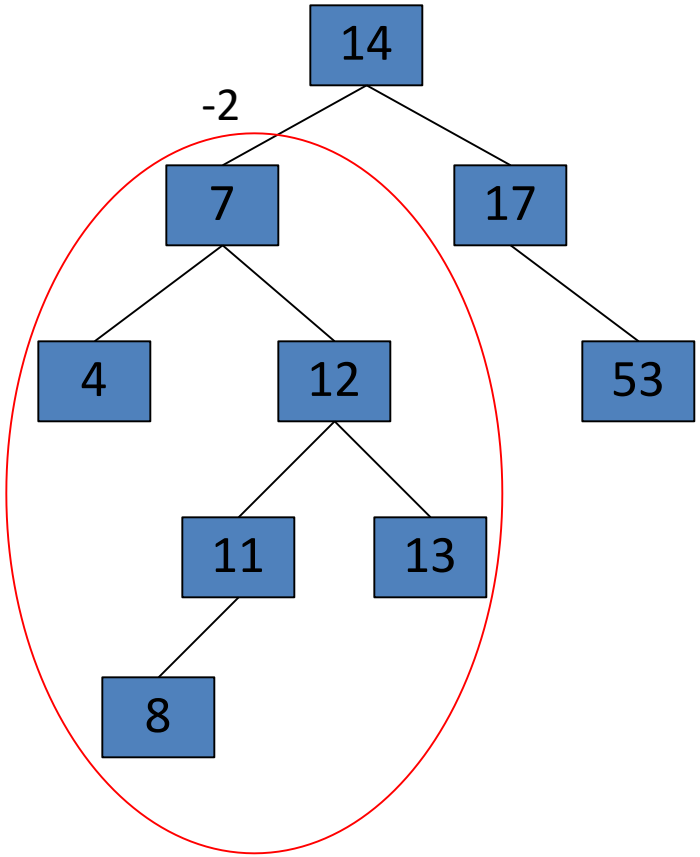
AVL Tree Example:

- **Now the AVL tree is balanced.**



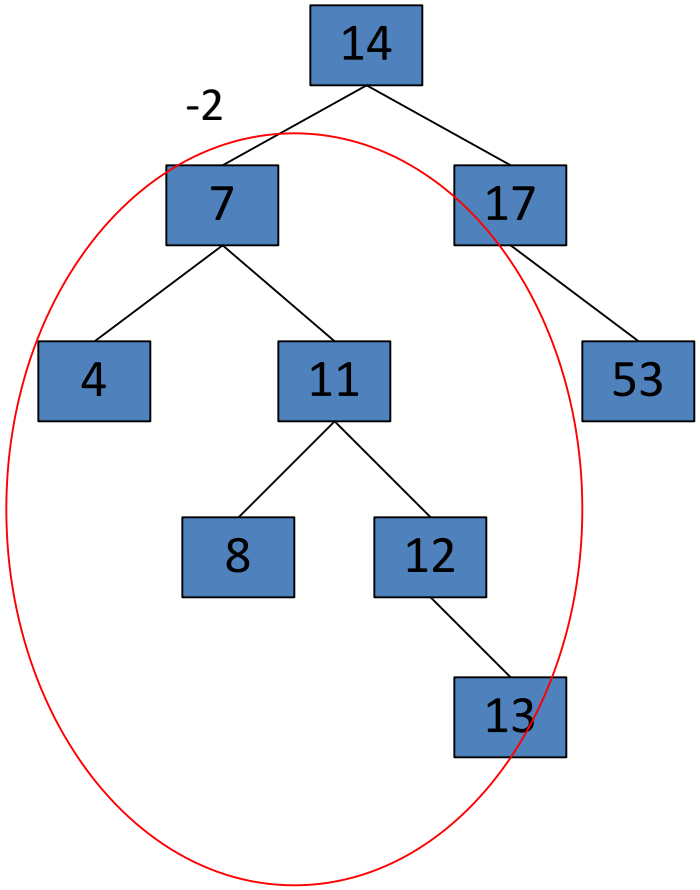
AVL Tree Example:

- **Now insert 8**



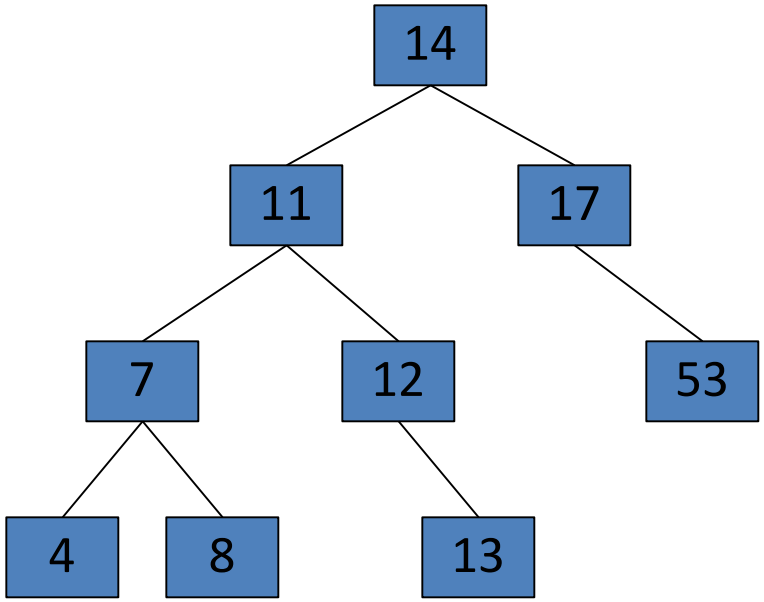
AVL Tree Example:

- **Now insert 8**



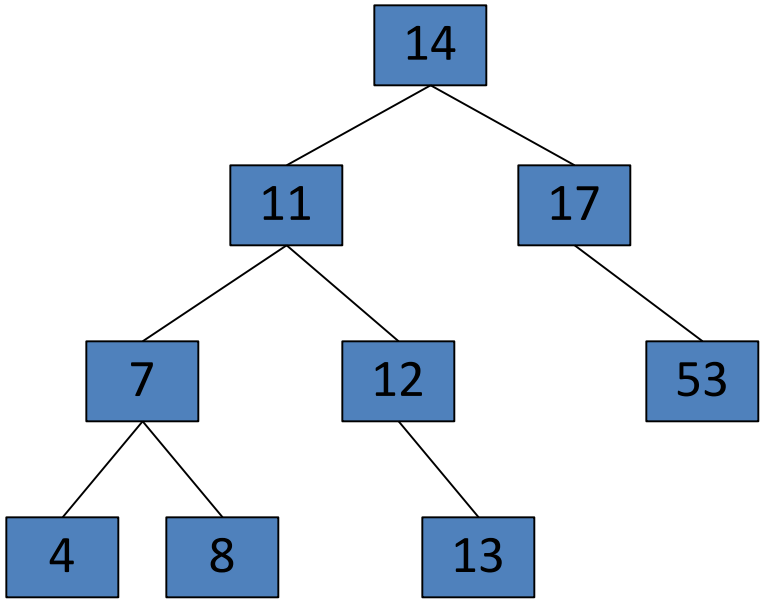
AVL Tree Example:

- **Now the AVL tree is balanced.**



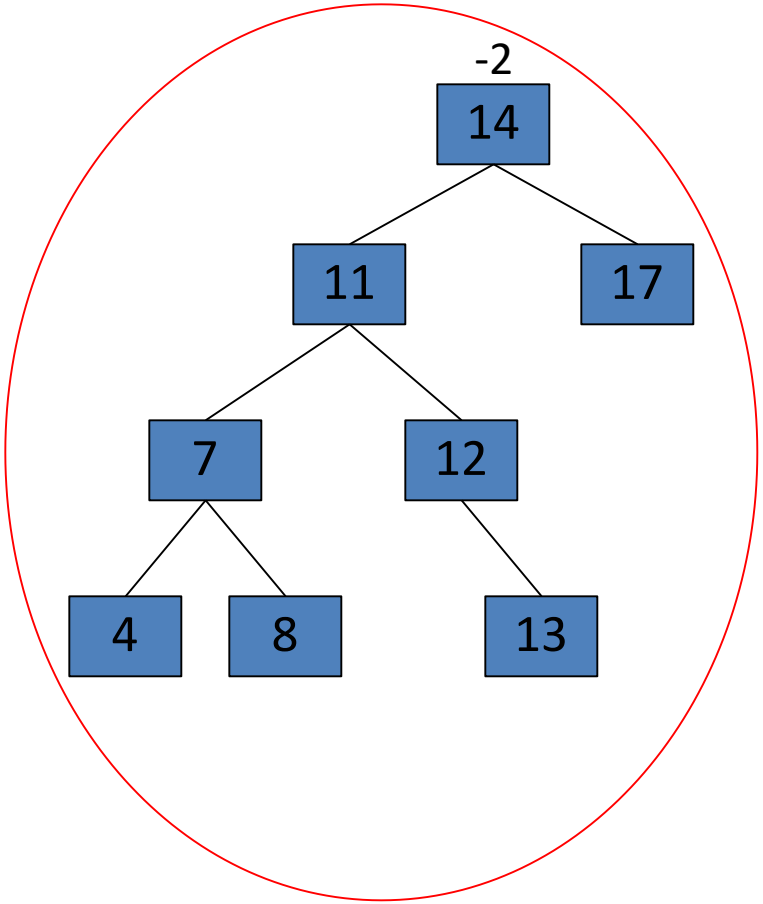
AVL Tree Example:

- **Now remove 53**



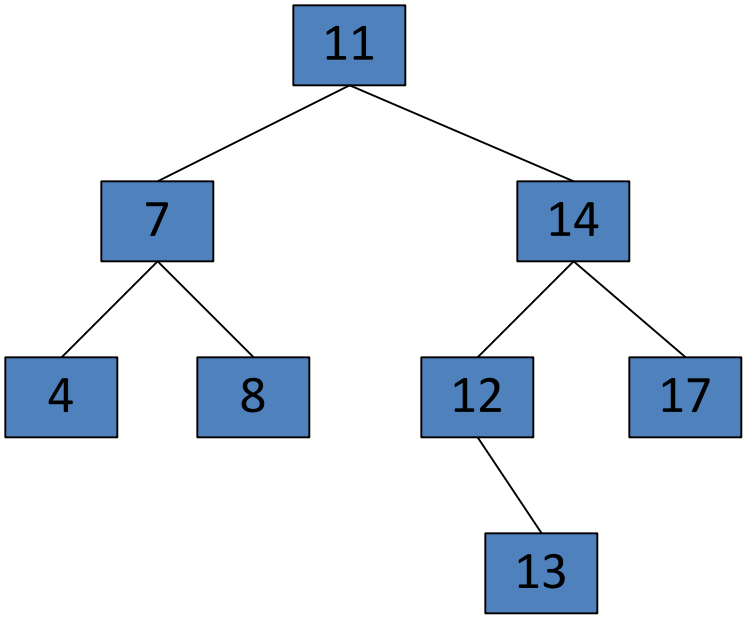
AVL Tree Example:

- **Now remove 53, unbalanced**



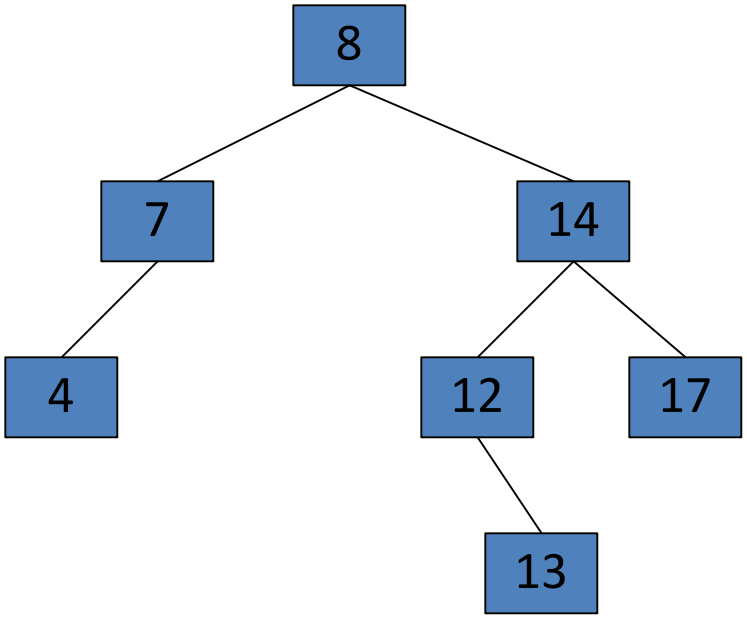
AVL Tree Example:

- **Balanced! Remove 11**



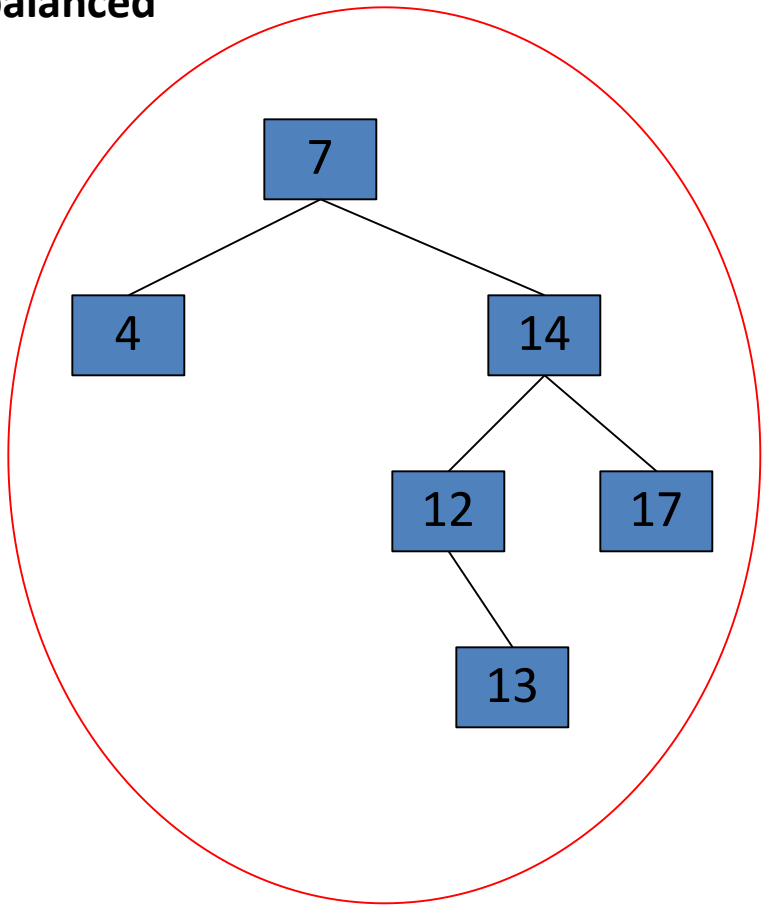
AVL Tree Example:

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



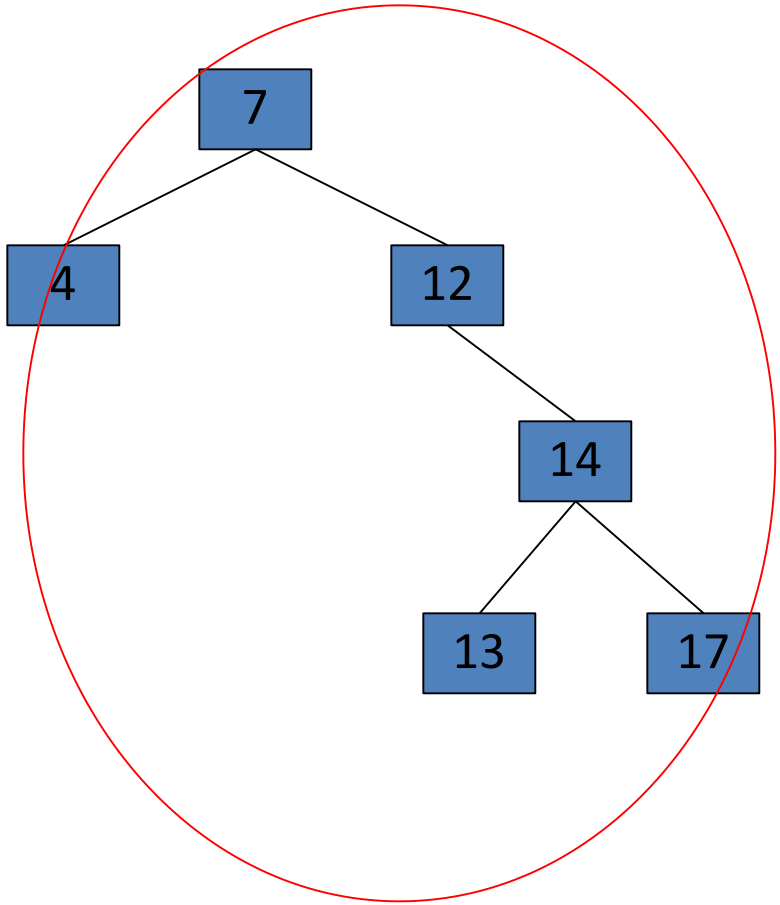
AVL Tree Example:

- **Remove 8, unbalanced**



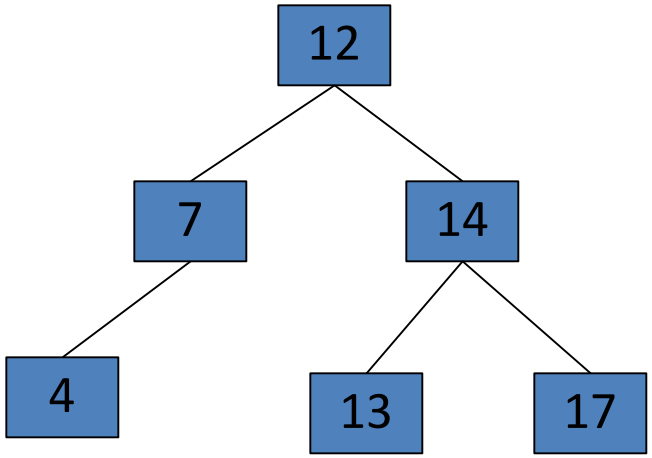
AVL Tree Example:

- **Remove 8, unbalanced**



AVL Tree Example:

- **Balanced!!**

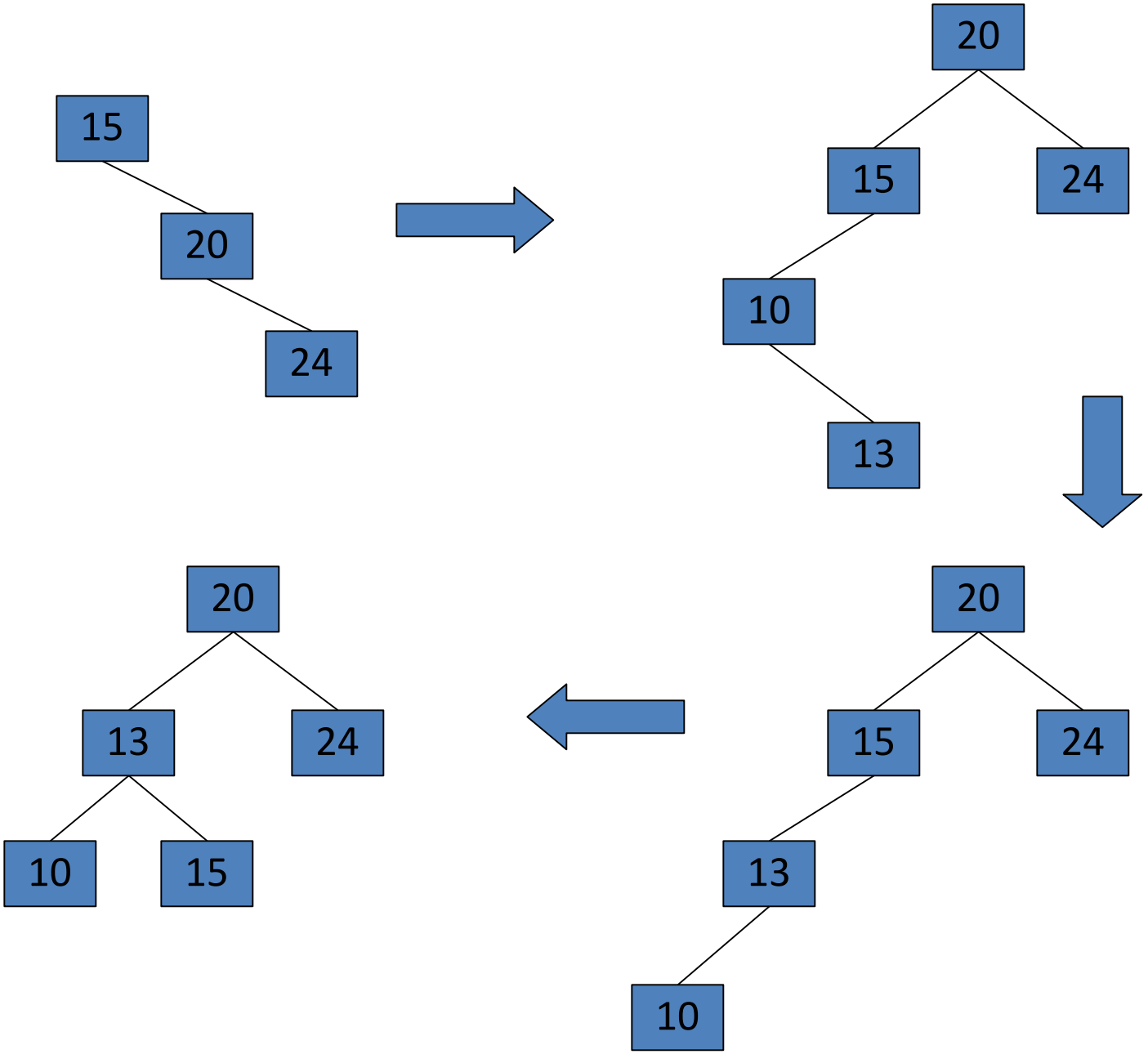


Exercise

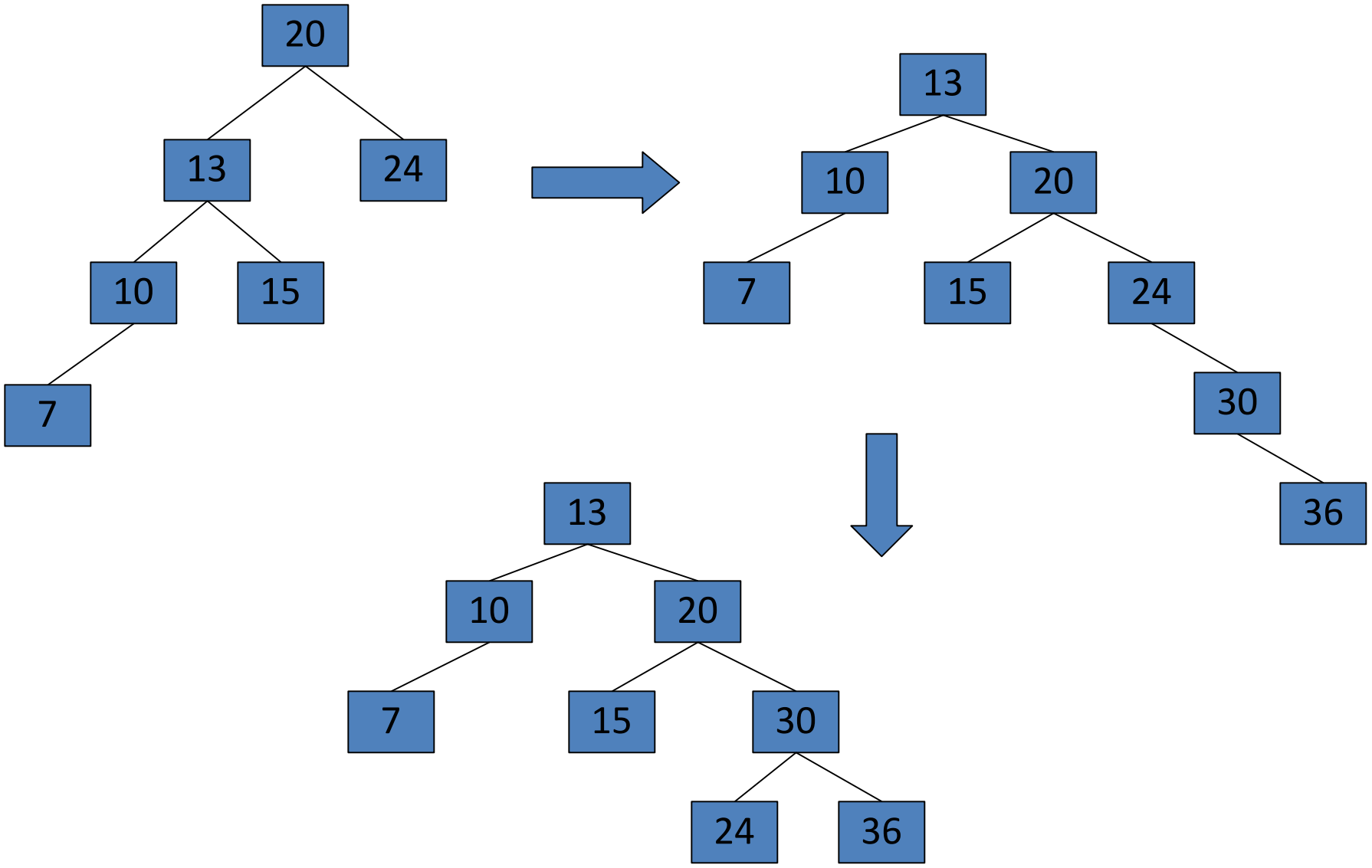
- Build an AVL tree with the following values:

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

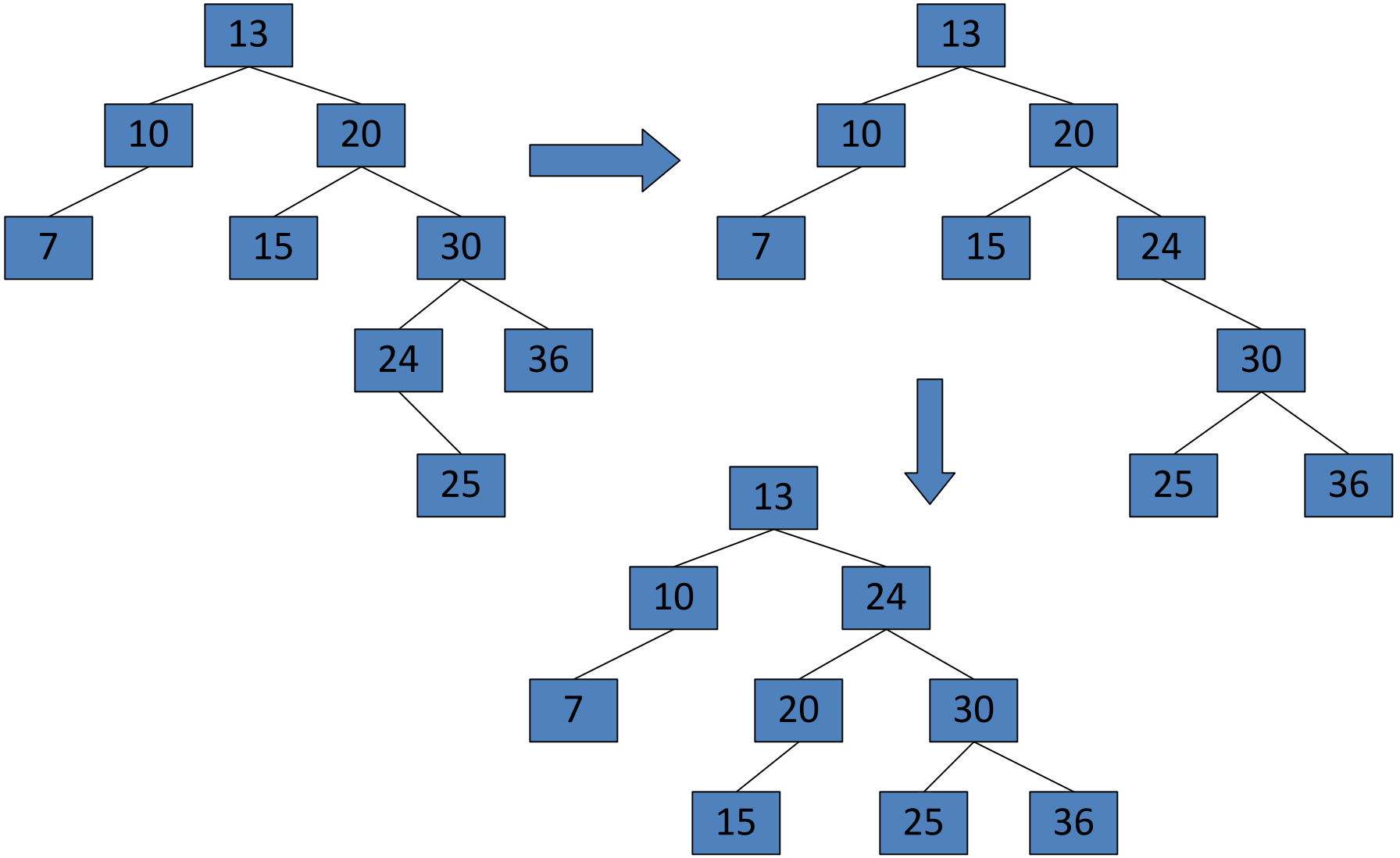
15, 20, 24, 10, 13, 7, 30, 36, 25



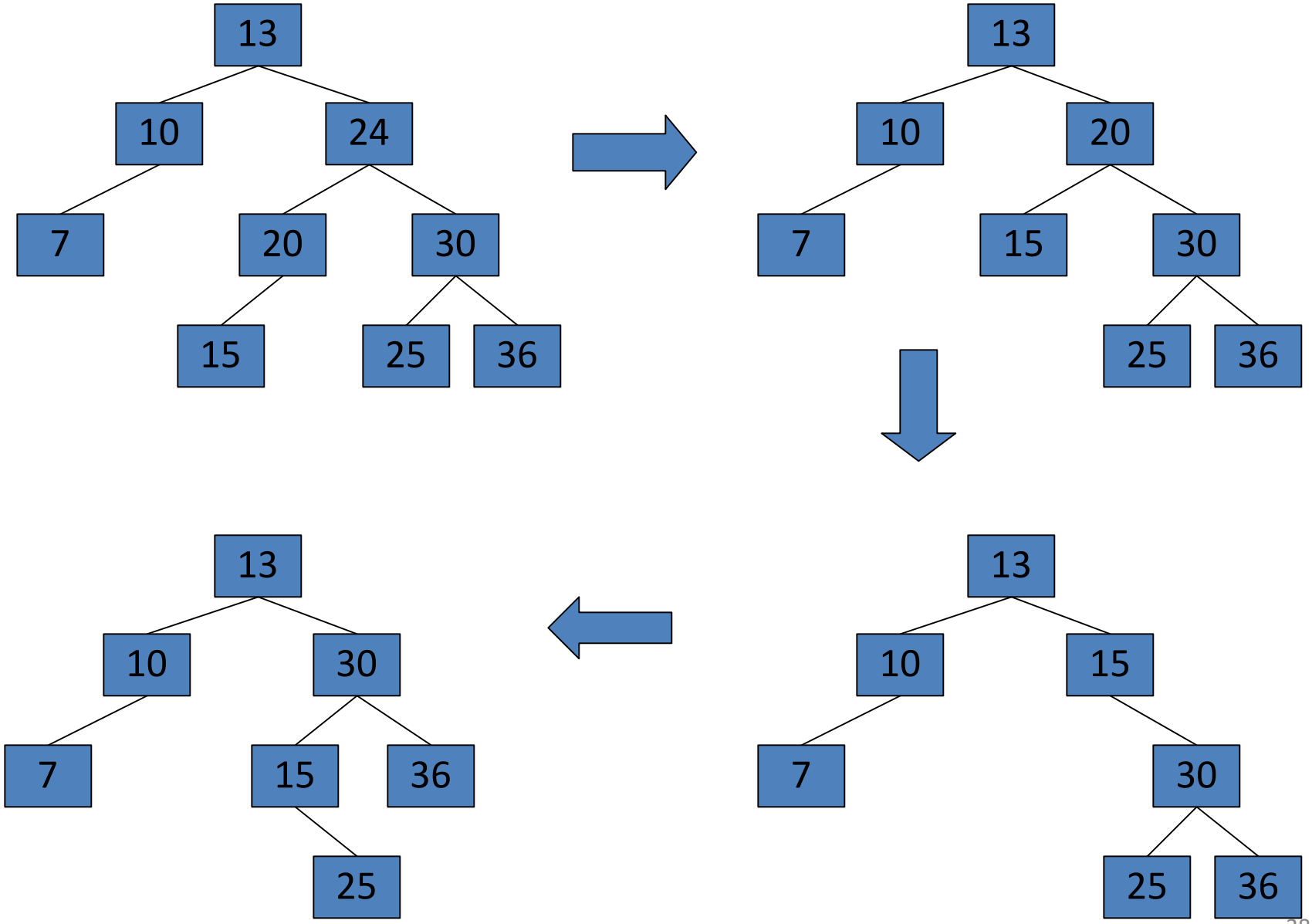
15, 20, 24, 10, 13, 7, 30, 36, 25



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



Remove 24 and 20 from the AVL tree.



TREES

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.**

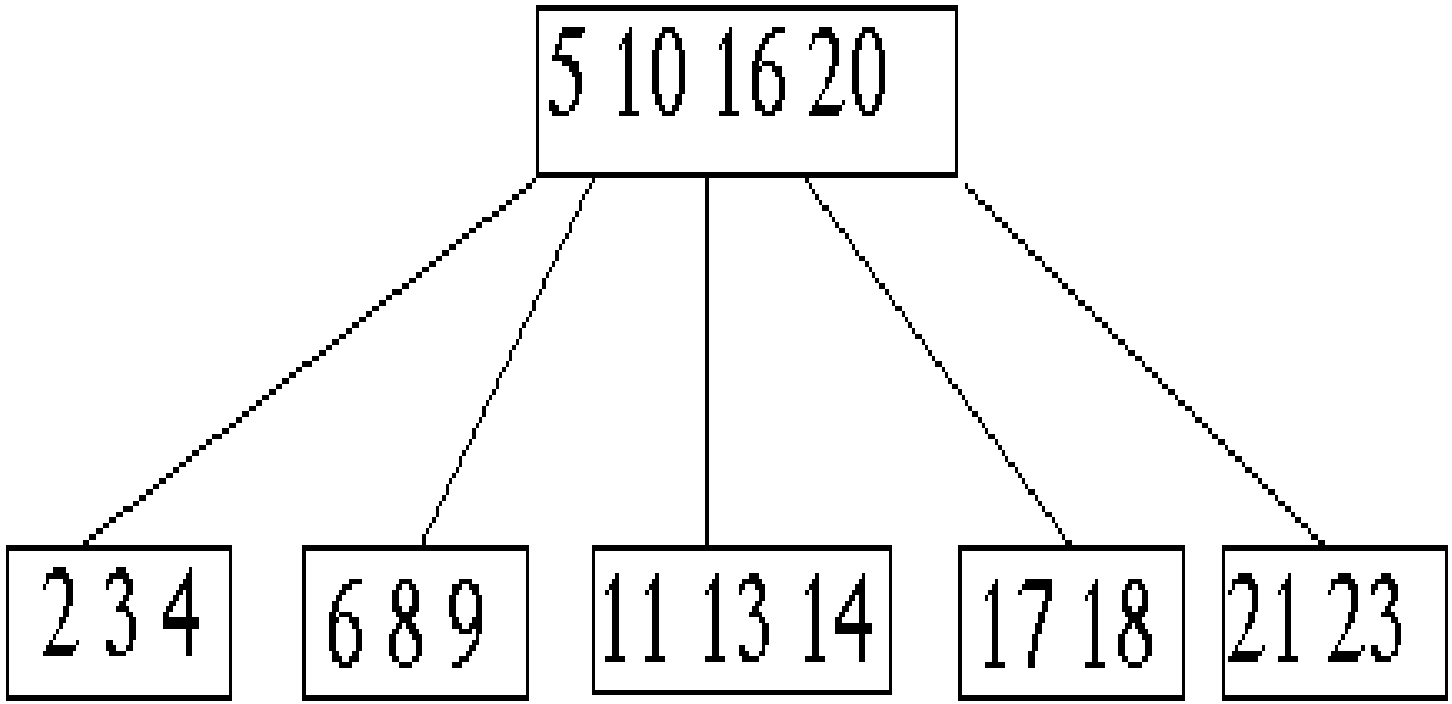
- **A B-tree of order n is a multi-way search tree with two properties:**
- **1.All leaves are at the same level**
- **2.The number of keys in any node lies between n and 2n, with the possible exception of the root which may have fewer keys.**

Other definition

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

- **Every node has ≤ m children.**
- **Every internal node (except the root) has ≤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.**

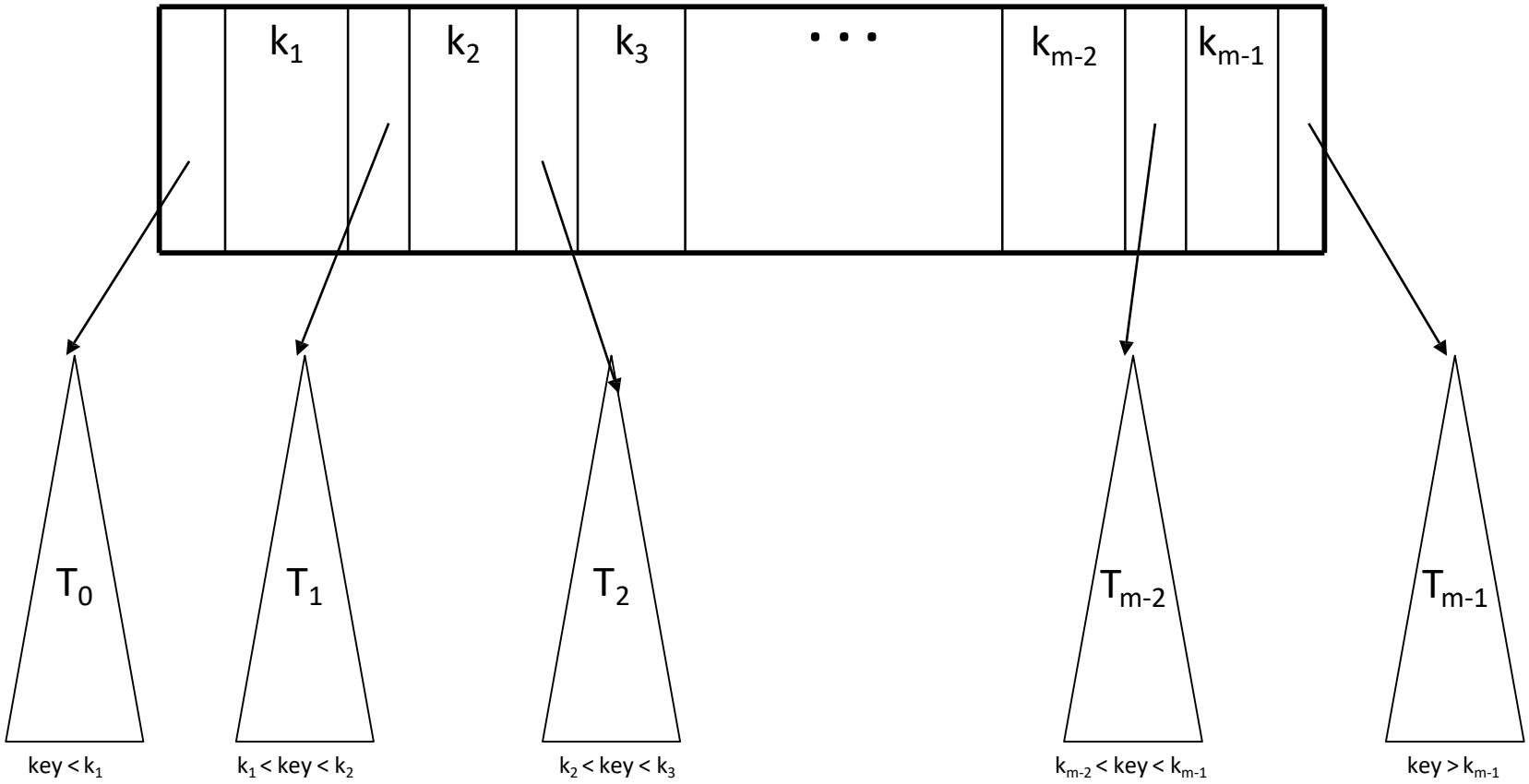
A B-tree of order 2



- **A multi-way (or m-way) search tree of order m is a tree in which**
  - **Each node has at-most m sub trees, where the sub trees 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 sub trees of a non-leaf node are ordered as: $T_0, k_1, T_1, k_2, T_2, \ldots, k_{m-1}, T_{m-1}$ such that:**
  - **All keys in sub tree T0 are less than k1.**
  - **All keys in sub tree T_i , $1 <= i <= m - 2$, are greater than k_i but less than k_{i+1}.**
  - **All keys in sub tree T_{m-1} are greater than k_{m-1}**

Multi-way tree



What is B-tree?

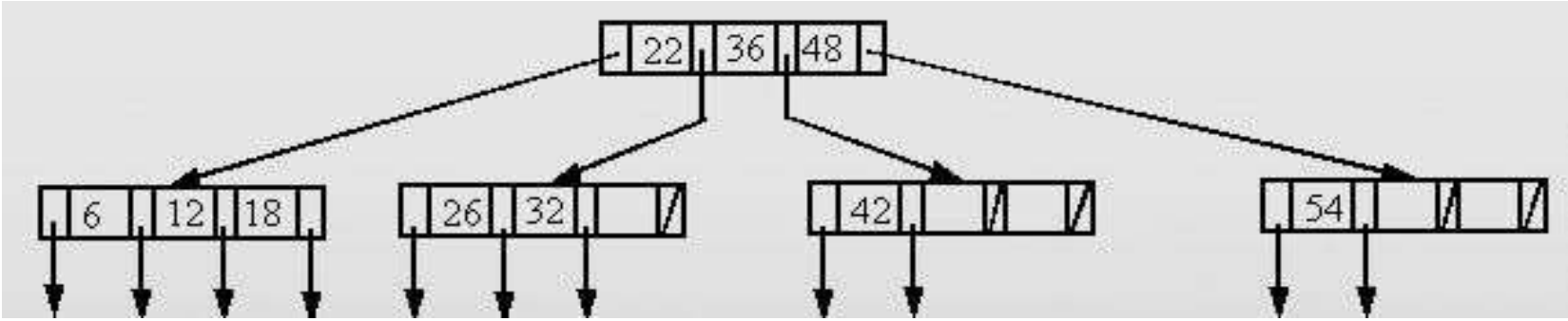
- **B-tree of order m (or branching factor m), where m > 2, is either an empty tree or a multiway search tree with the following properties:**
  - –The root is either a leaf or it has at least two non-empty subtrees and at most m non-empty subtrees.
  - –Each non-leaf node, other than the root, has at 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

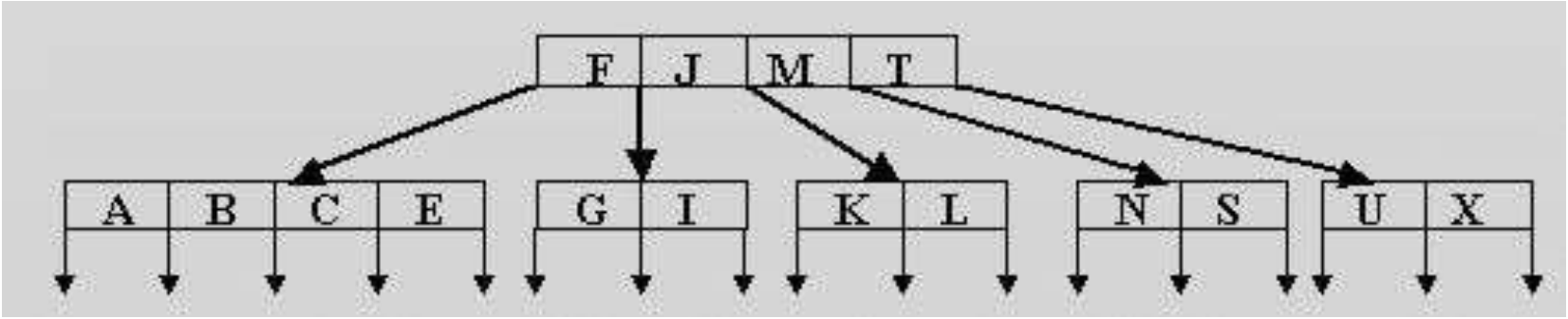
- For a non-empty B+ tree of order m:

What is a B-tree?

	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





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 is 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 ith 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 similiar 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:**

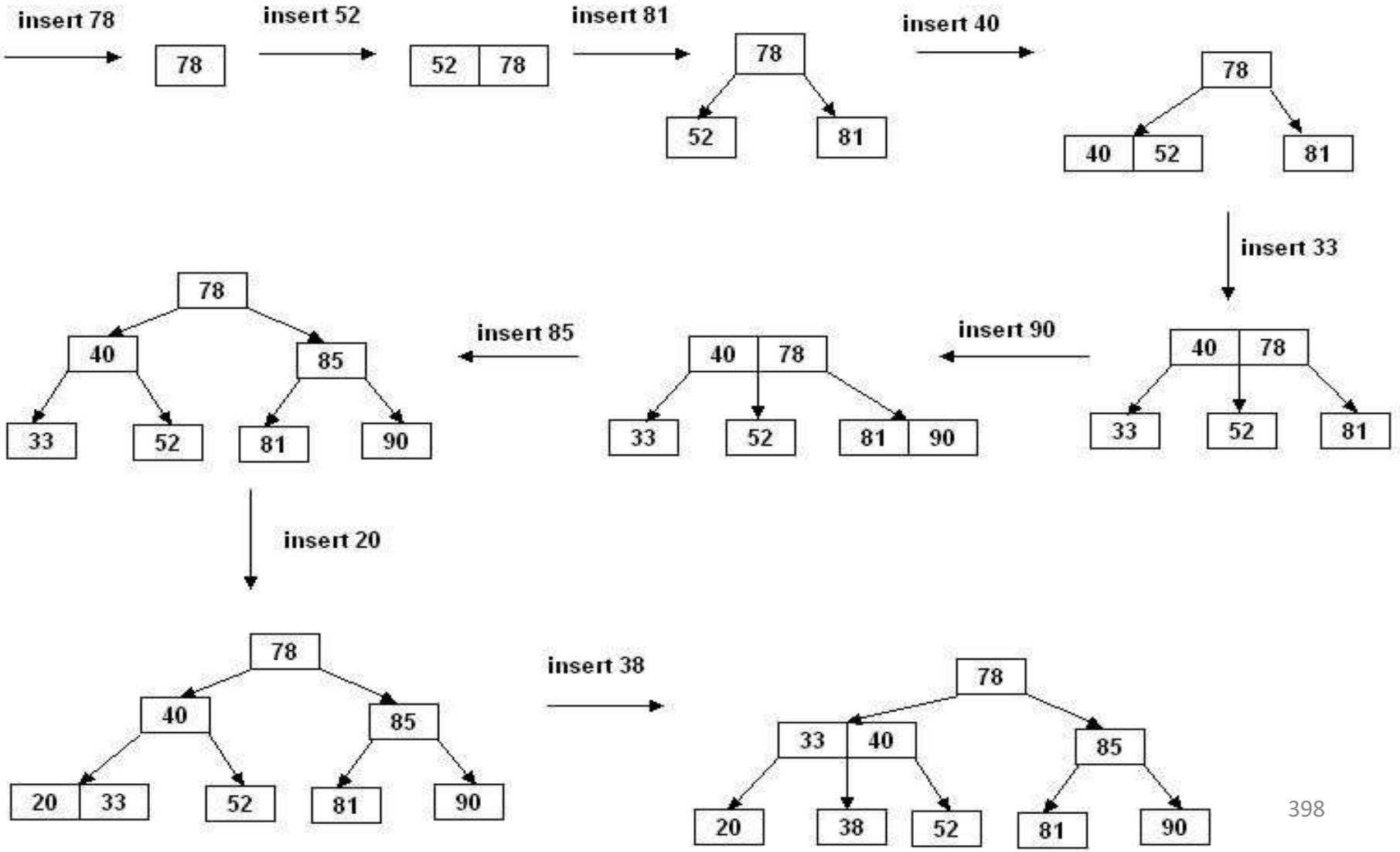
  A root-node or a non-root node of a B-tree of order m overflows if, after a key insertion, it contains m keys.

- **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.

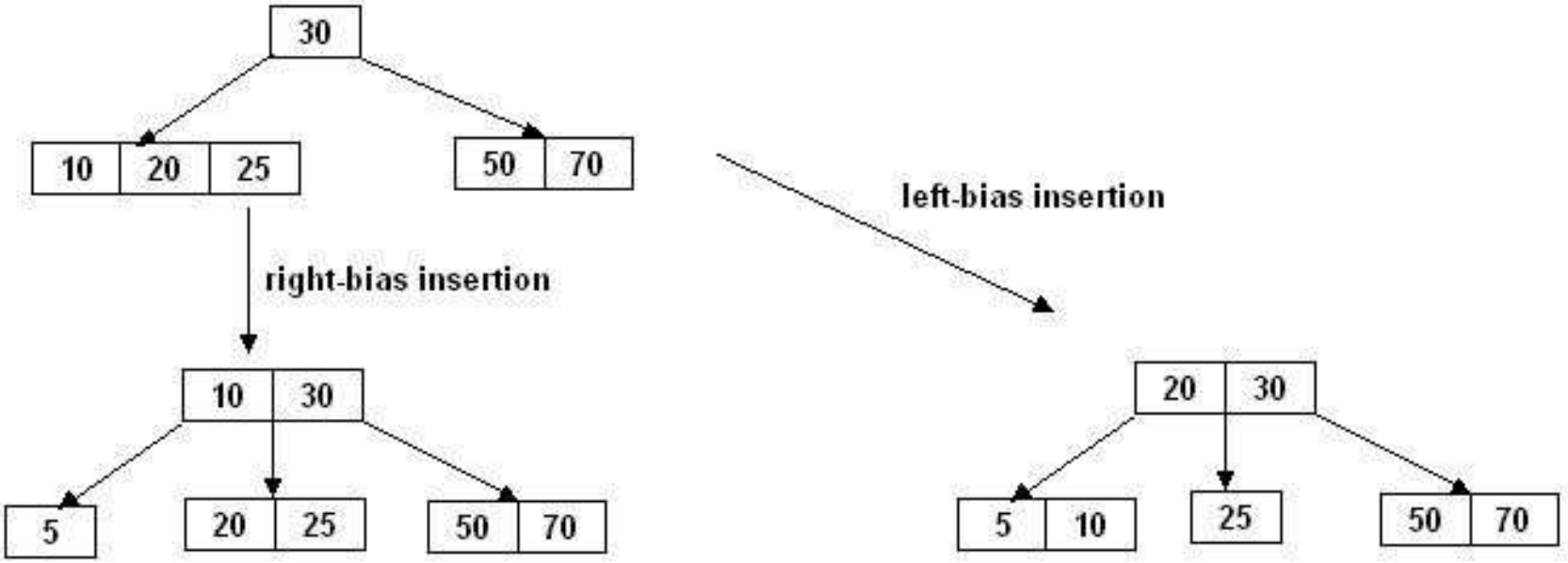
- **Note: Insertion of a key always starts at a leaf node.**

- **Insertion in a B-tree of <u>odd</u> 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

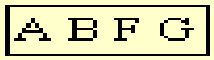
- **Insertion in a B-tree of <u>even</u> order**
- **right-bias: The node is split such that its right subtree has more keys than the left subtree.**
- **left-bias: The node is split such that its left subtree has more keys than the right subtree.**
- **Example: Insert the key 5 in the following B-tree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



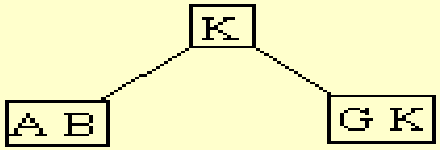
Insertion

- **Insert the keys in the folowing 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.**

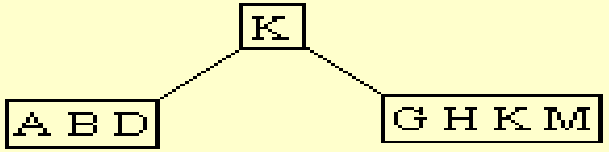
(1) Insert A, G, F and B.



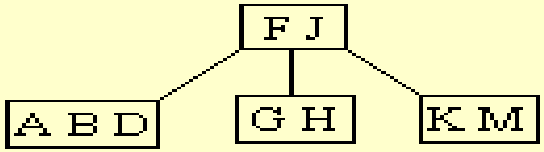
(2) Insert K .



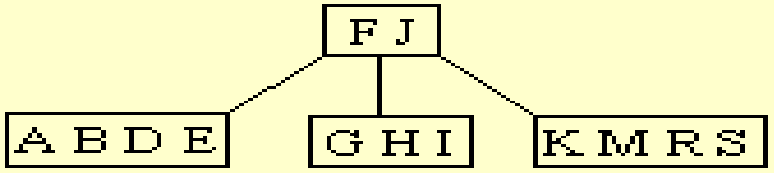
(3) Insert D, H and M.



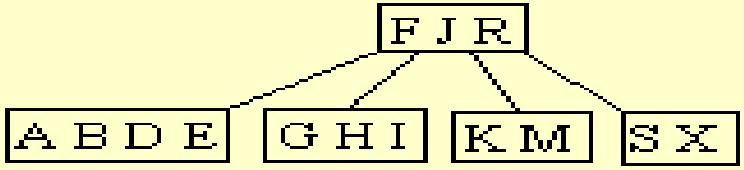
(4) Insert J .



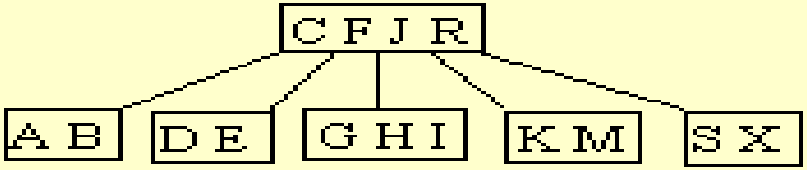
(5) Insert E, S, I and R.



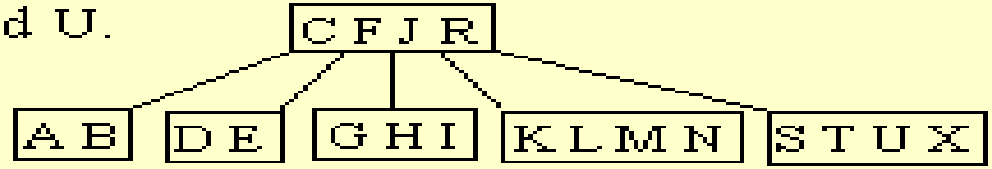
(6) Insert X.



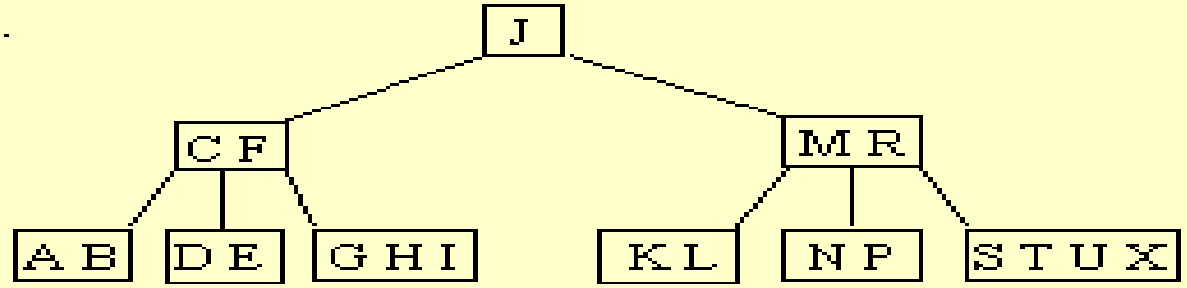
(7) Insert C.



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



(9) Insert P.



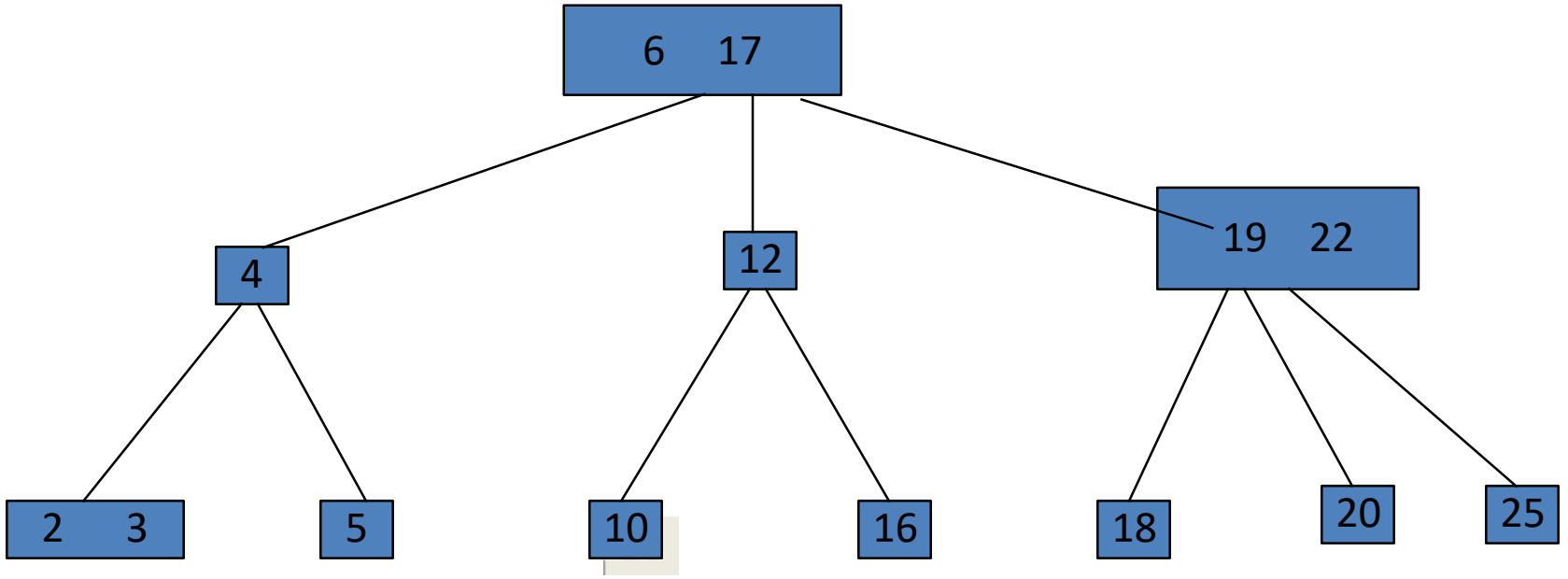
Searching

Searching for an Item in a B-Tree:

1. Make a local variable, i, equal to the first index such that data[i] >= 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
    data[i])  return true;
else if (the root has no children)
     return false;
else
     return subset[i]->contains (target);
```

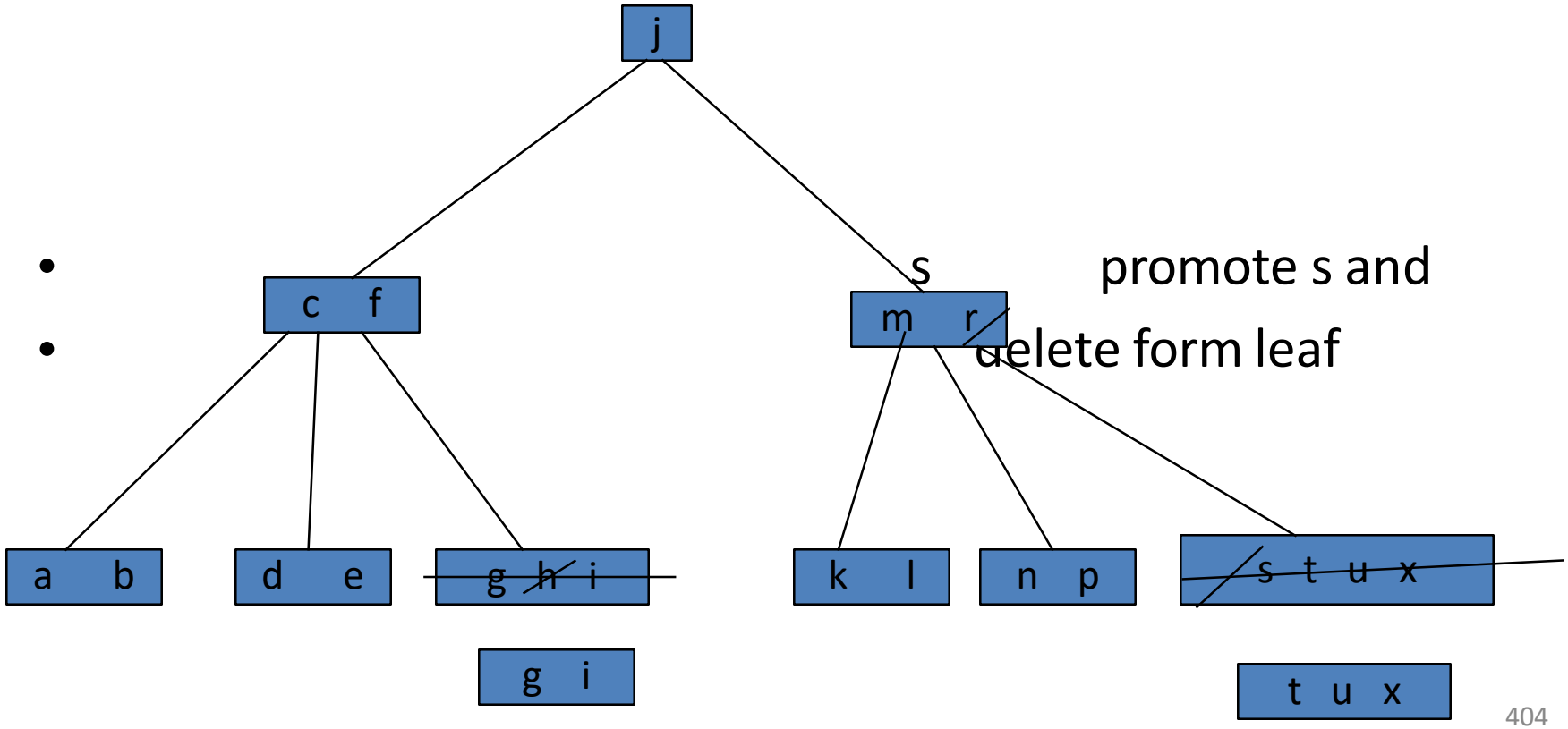
Searching (cont.)

- Example: target = 10



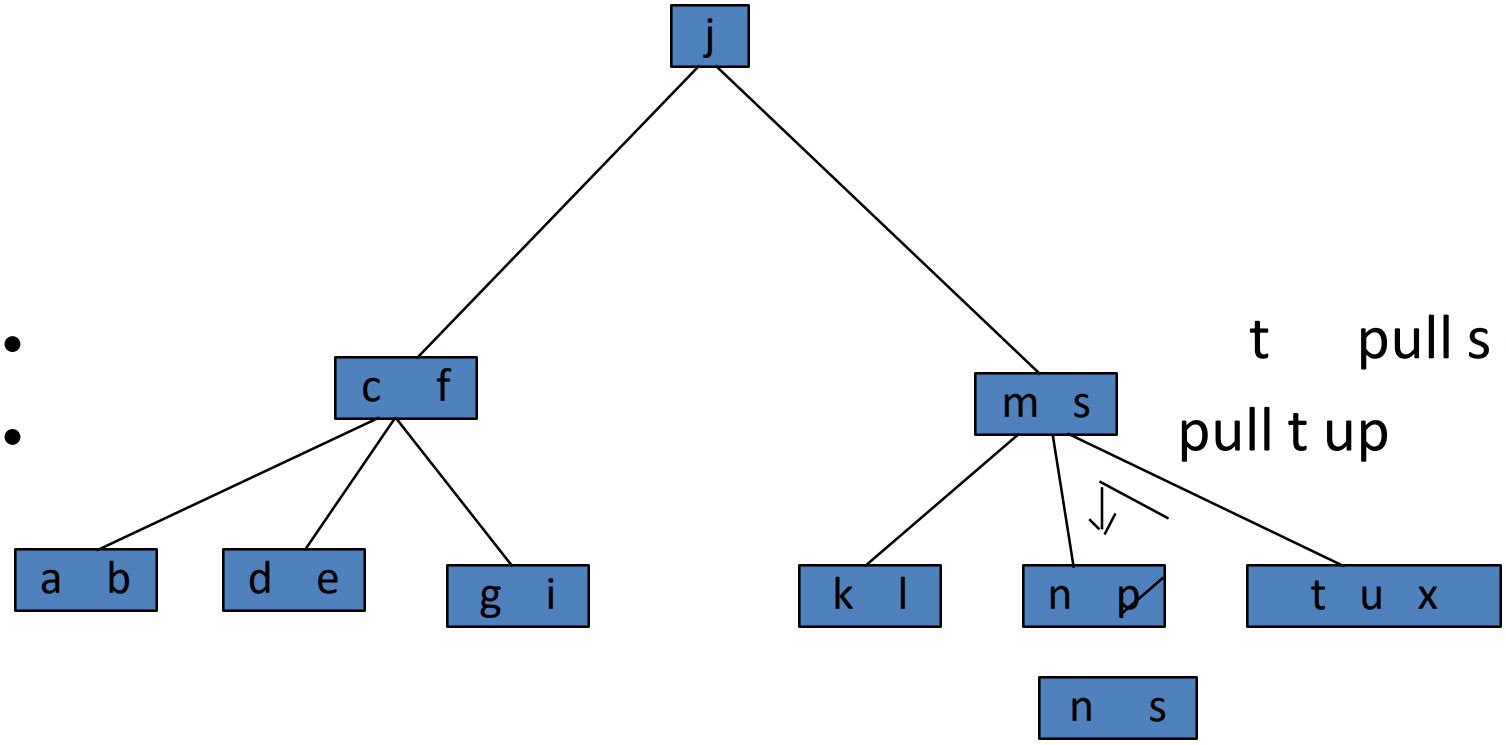
Deletion form a B-Tree

- 1. detete h, r :



Deletion (cont.)

- 2. delete p :
-
-
-



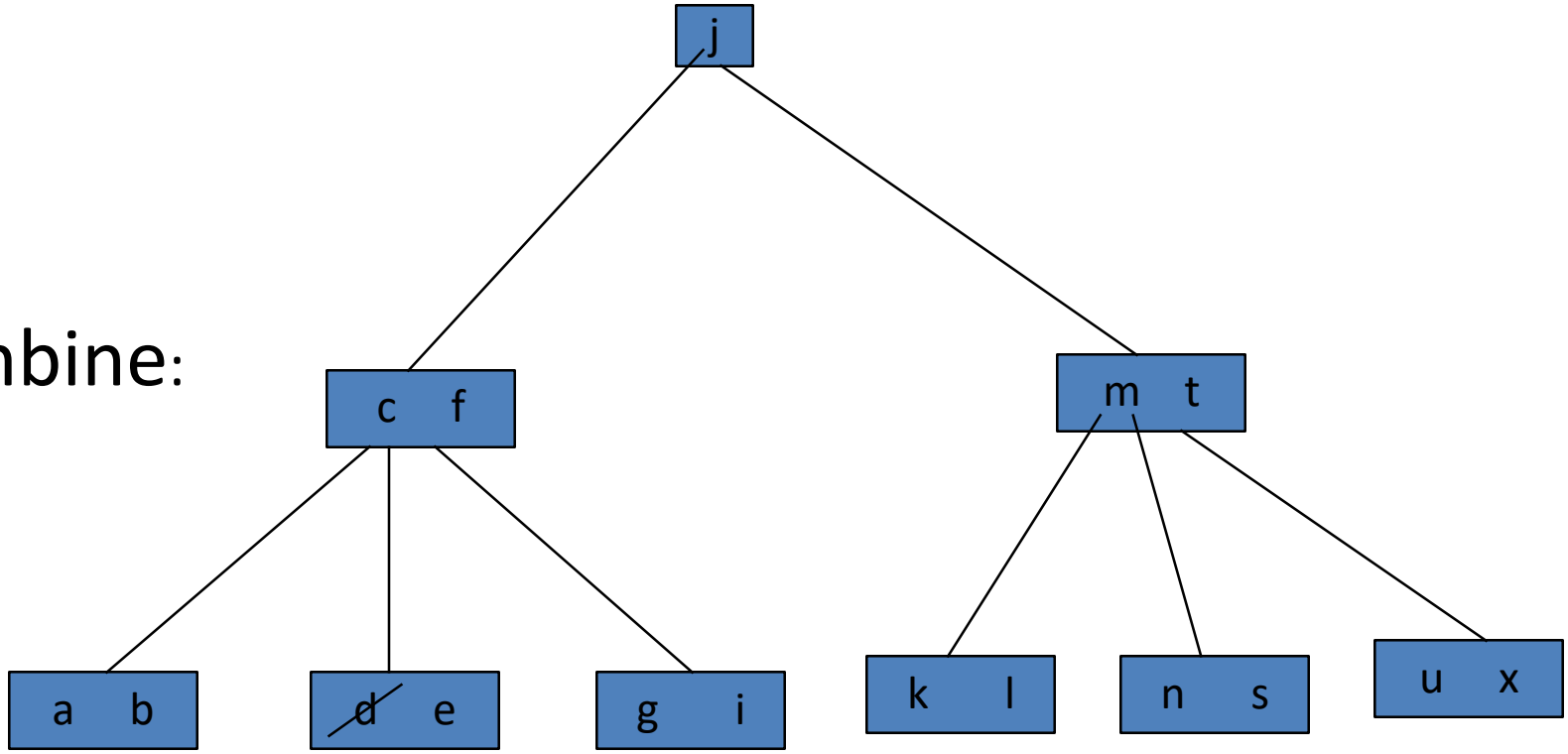
t pull s down;

pull t up

Deletion (cont.)

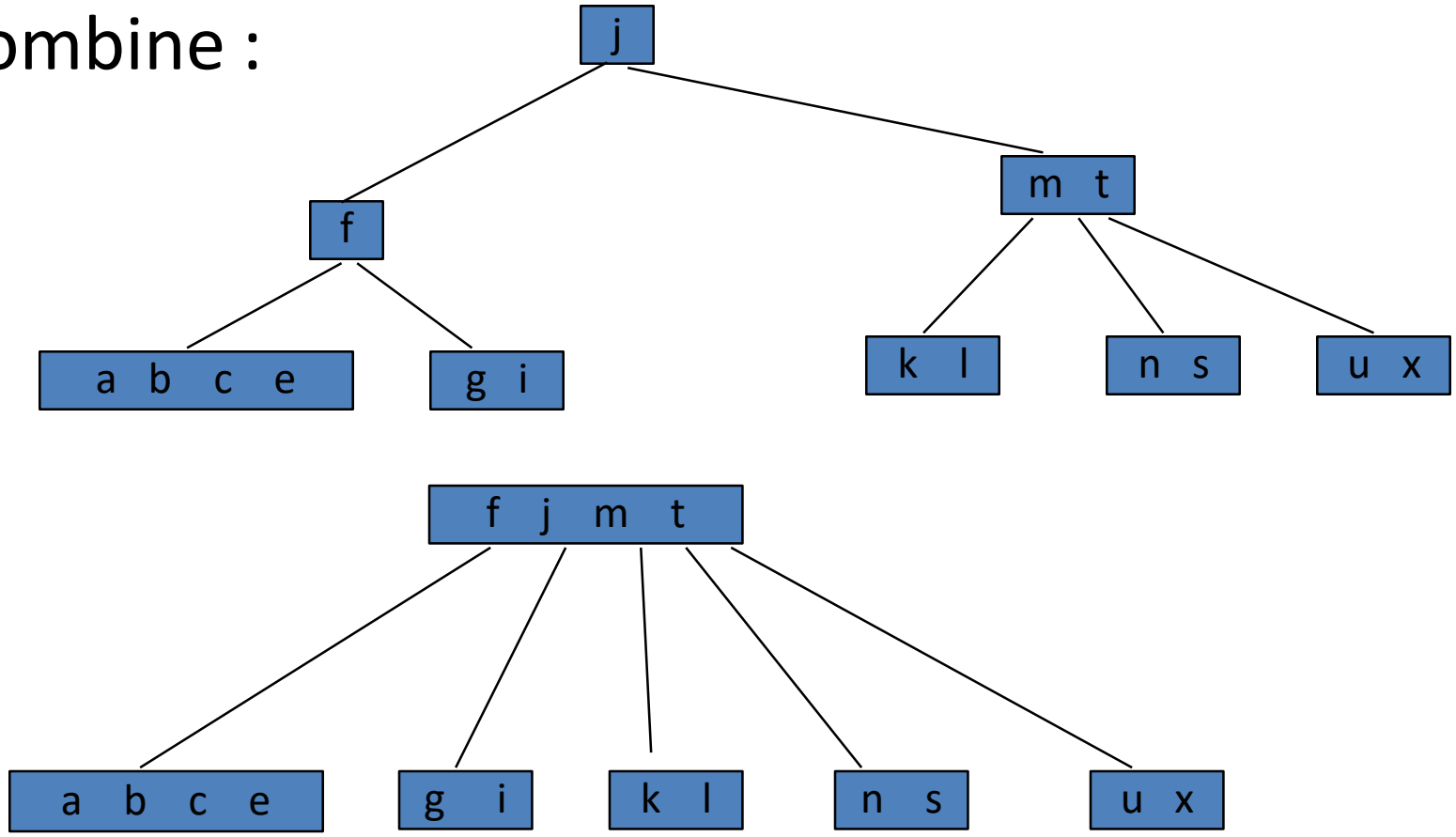
- 3. delete d:

- 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.

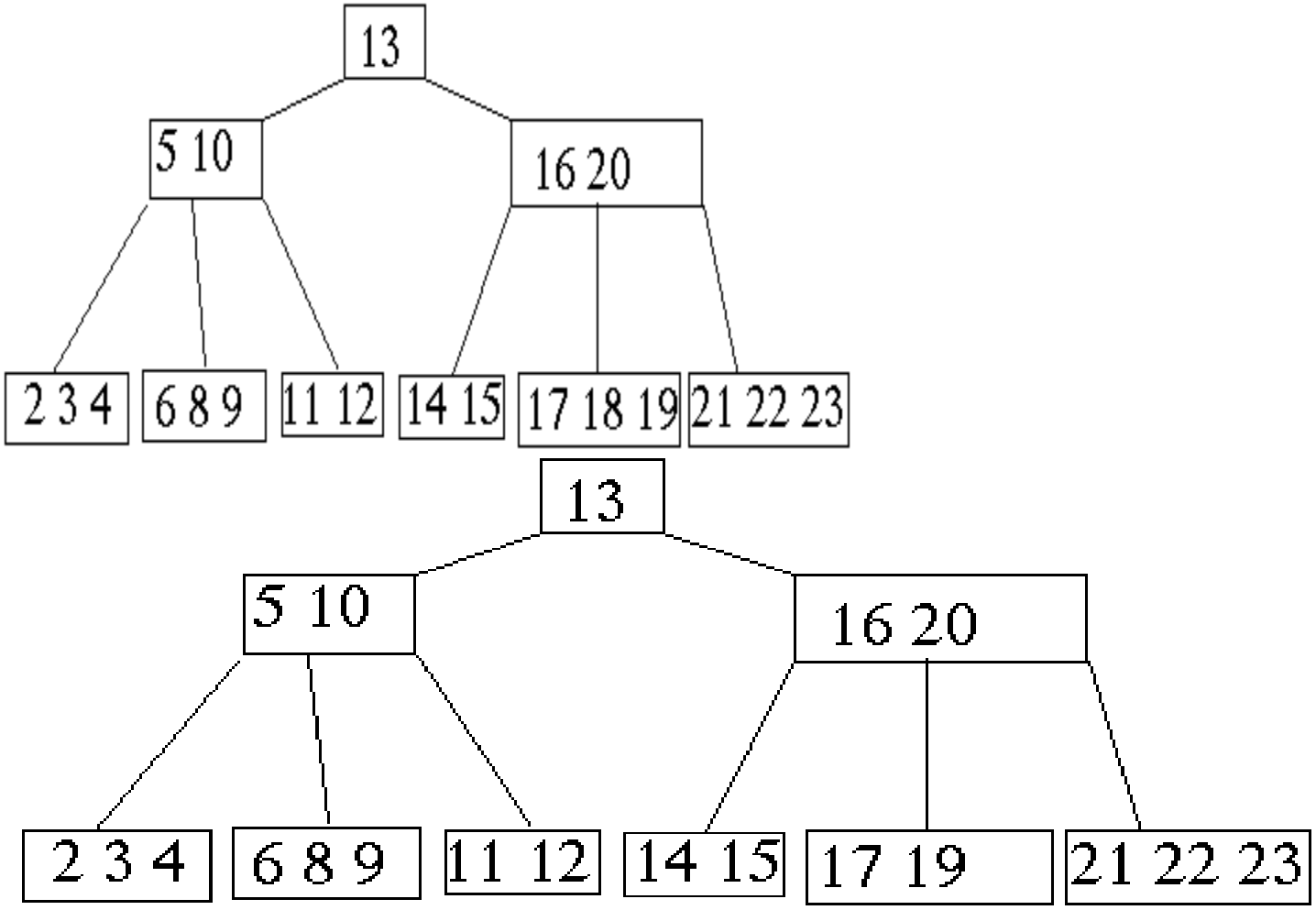
Deleting from a B-Tree (continued)

- 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.

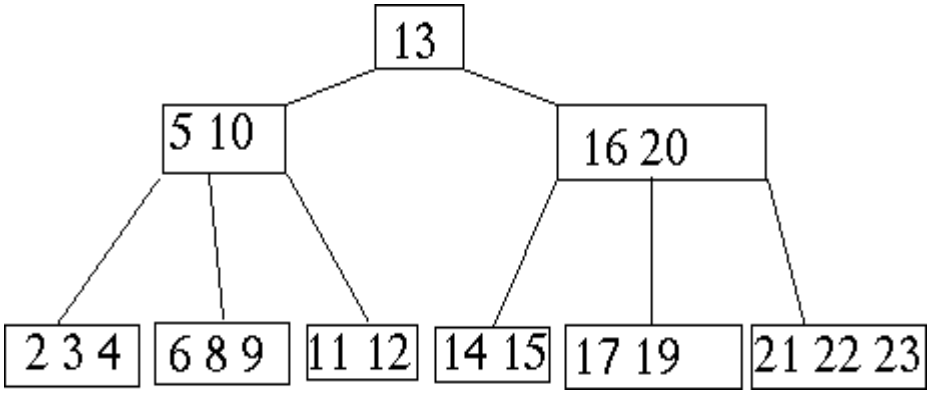
Deleting from a B-Tree (continued)

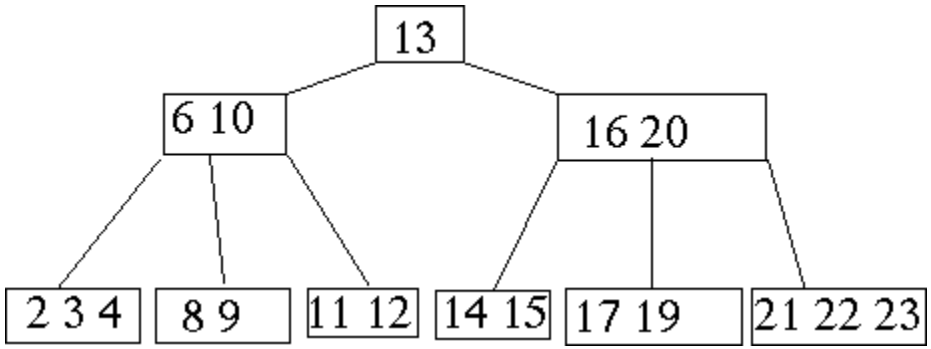
- This process may percolate all the way to the root.
- If the root is left with only one child, then remove the root node and make its child the new root.
- Both insertion and deletion are O(h), where h is the height of the tree.

Delete 18

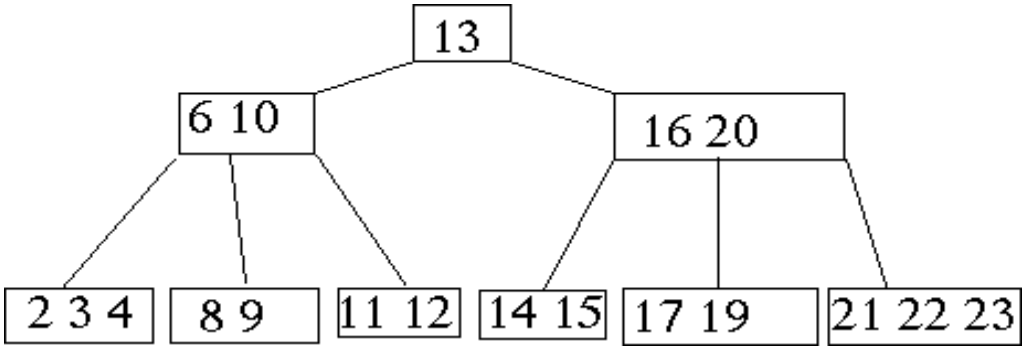

13
5 10
16 20
2 3 4
6 8 9
11 12
14 15
17 18 19
21 22 23
13
5 10
16 20
2 3 4
6 8 9
11 12
14 15
17 19
21 22 23

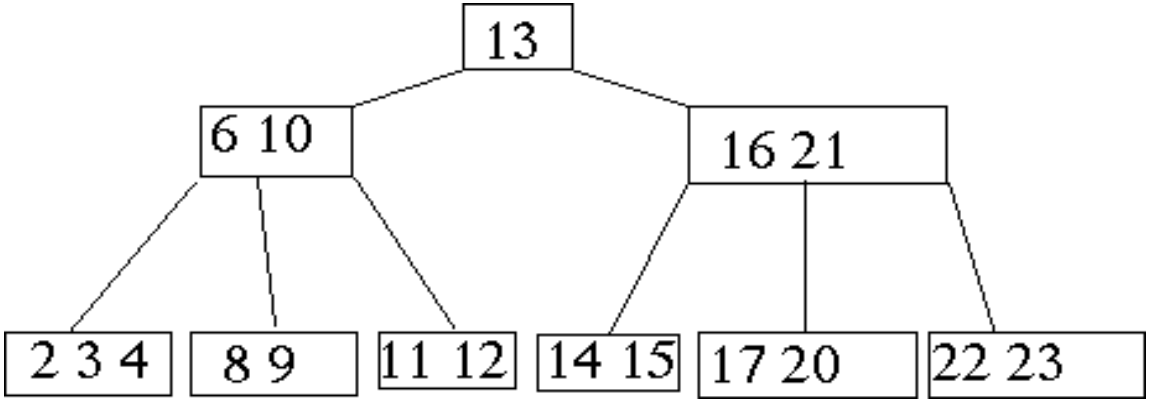
Delete 5


13
5 10
16 20
2 3 4
6 8 9
11 12
14 15
17 19
21 22 23

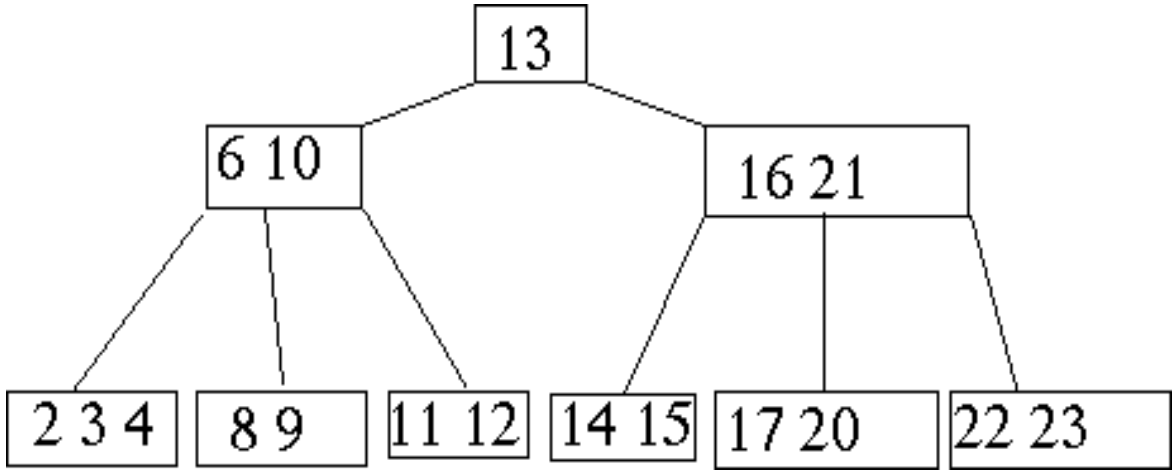

13
6 10
16 20
2 3 4
8 9
11 12
14 15
17 19
21 22 23

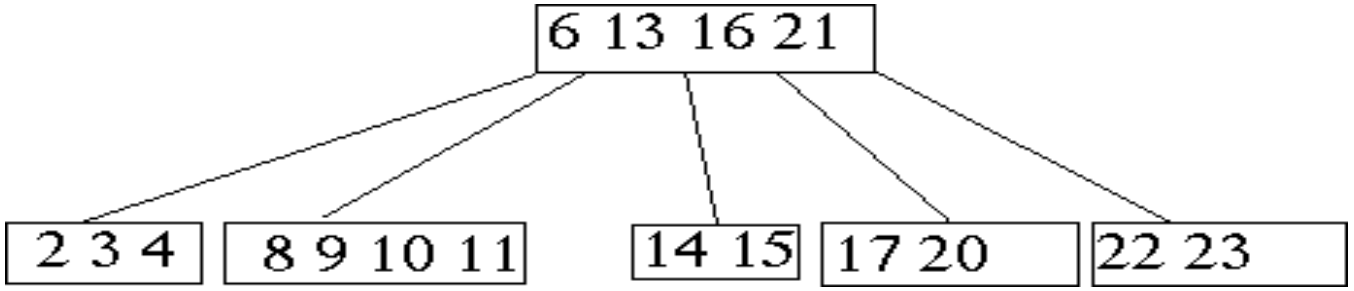
Delete 19


13
6 10
16 20
2 3 4
8 9
11 12
14 15
17 19
21 22 23


13
6 10
16 21
2 3 4
8 9
11 12
14 15
17 20
22 23

Delete 12


13
6 10
16 21
2 3 4
8 9
11 12
14 15
17 20
22 23


6 13 16 21
2 3 4
8 9 10 11
14 15
17 20
22 23

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 $\lceil \mathbf{m / 2} \rceil \mathbf{- 2}$ 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

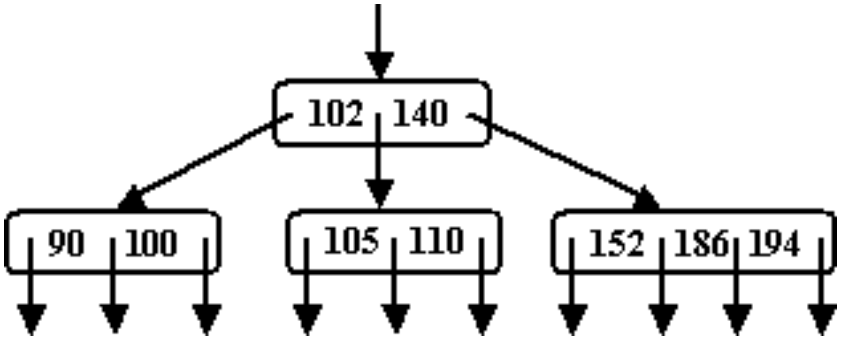
4.The leaf underflows and each of the adjacent right sibling and the adjacent left sibling has at least $\lceil m / 2 \rceil$ keys.

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

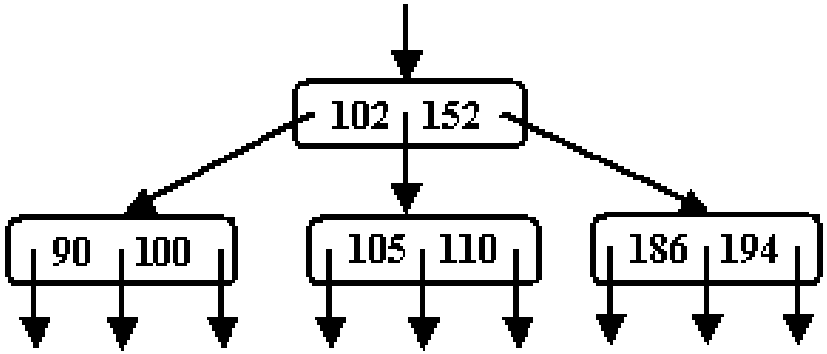
5. The leaf underflows and each adjacent sibling has $\lceil m / 2 \rceil - 1$ keys.

Deletion in B-Tree

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

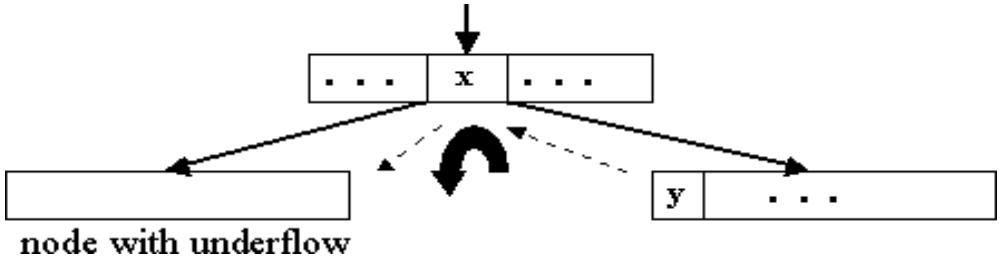


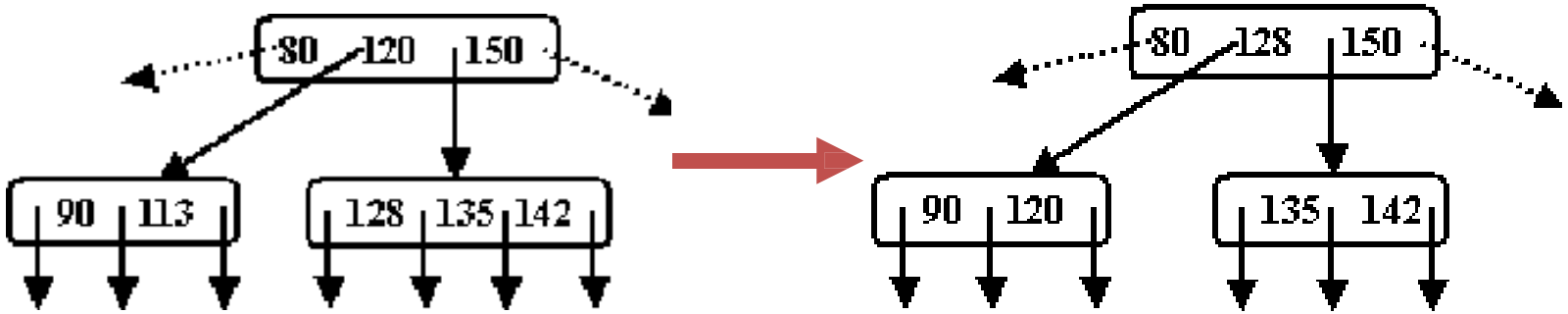
Delete 140



Deletion in B-Tree

- Case2: The leaf underflows and the adjacent right sibling has at least ⌈m/2⌉ keys.
- **Example : B-tree of order 5**

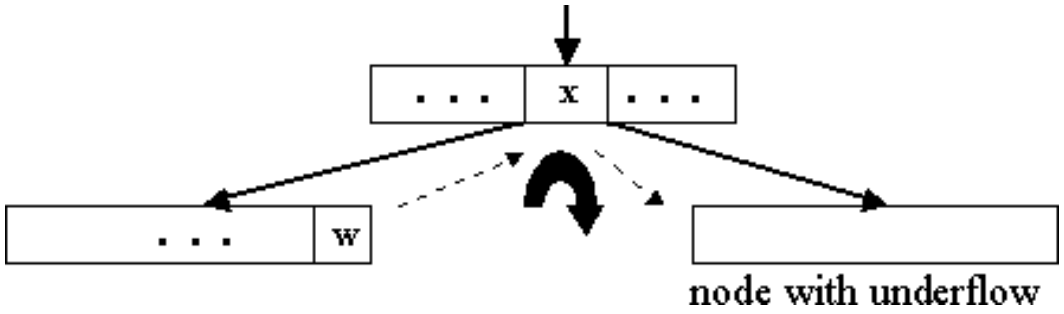




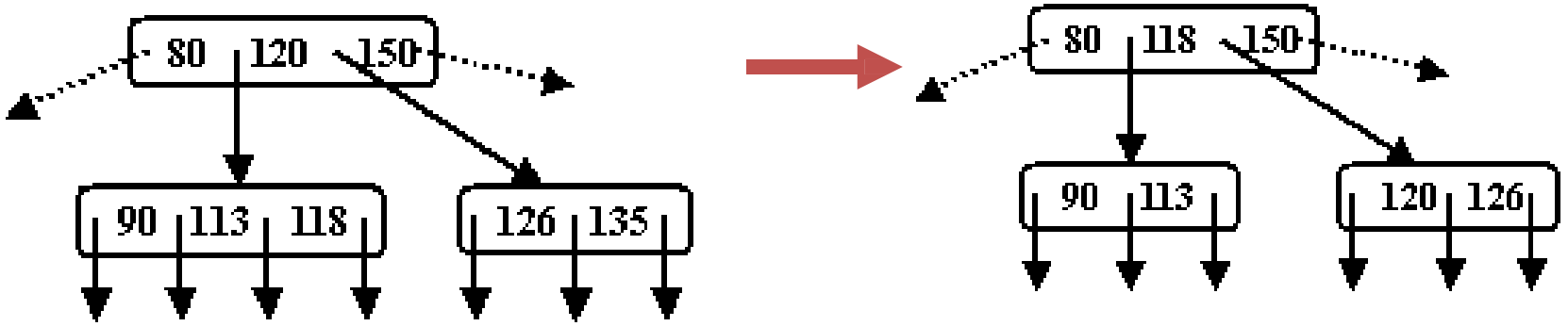
Delete 113

Deletion in B-Tree

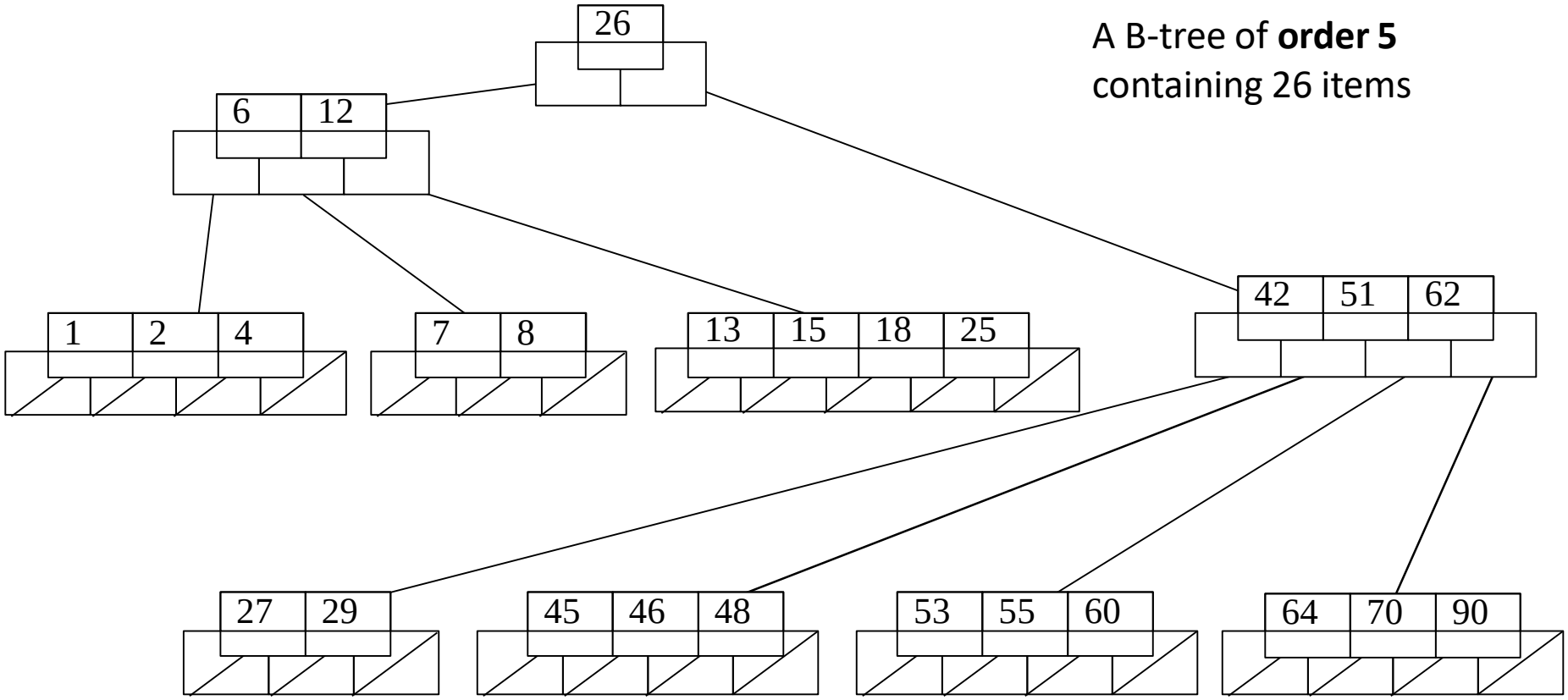
- Case 3: The leaf underflows and the adjacent left sibling has at least $\lceil m / 2 \rceil$ keys.
- **Example : B-tree of order 5**



Delete 135



An example B-Tree



A B-tree of **order 5** containing 26 items

Note that all the leaves are at the same level

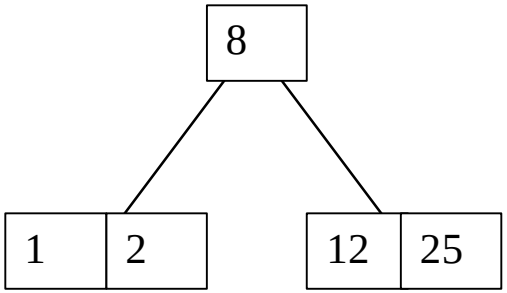
Constructing a B-tree

- 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:

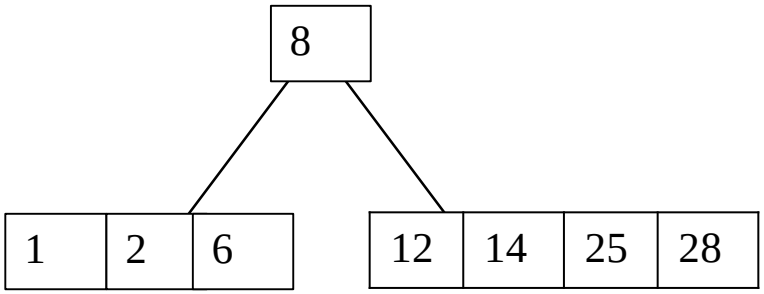
1	2	8	12

- 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.)

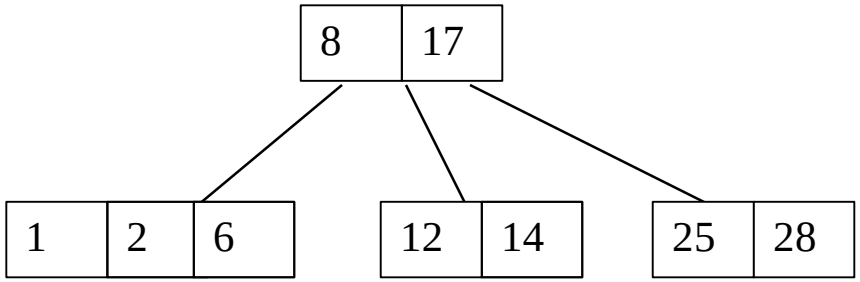


6, 14, 28 get added to the leaf nodes:

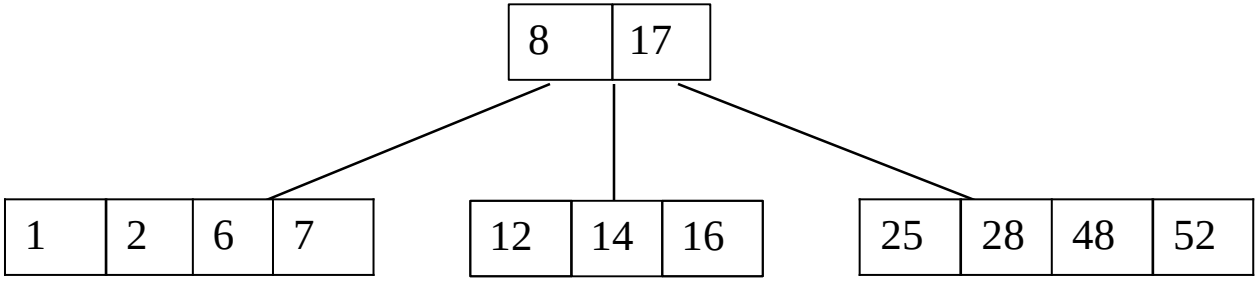


Constructing a B-tree (contd.)

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

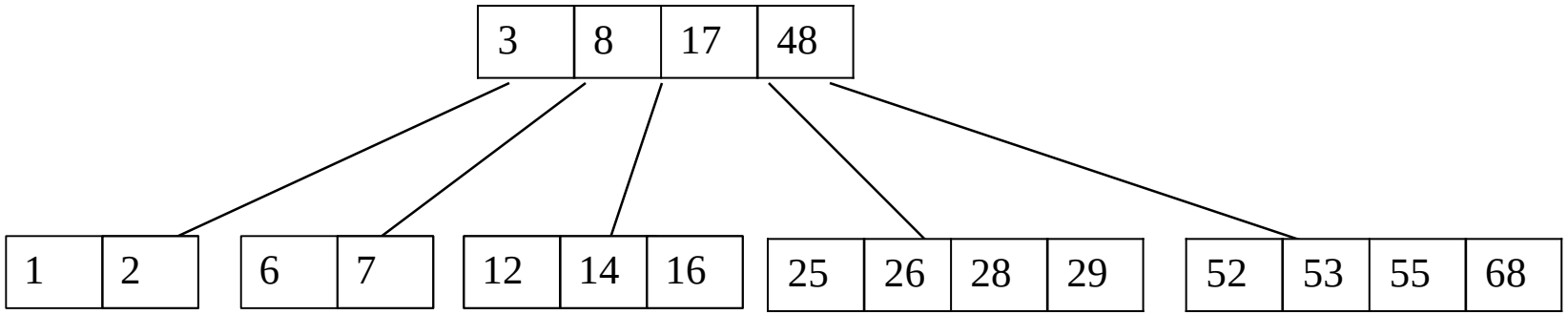


7, 52, 16, 48 get added to the leaf nodes

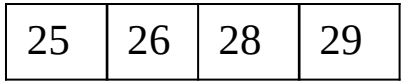


Constructing a B-tree (contd.)

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

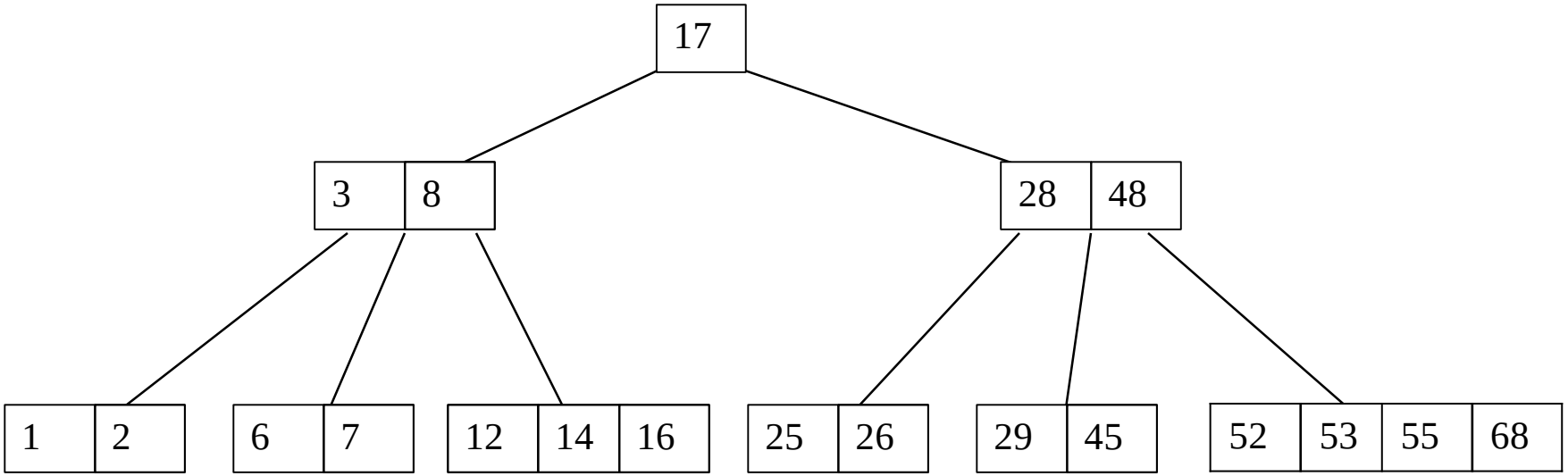


Adding 45 causes a split of



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

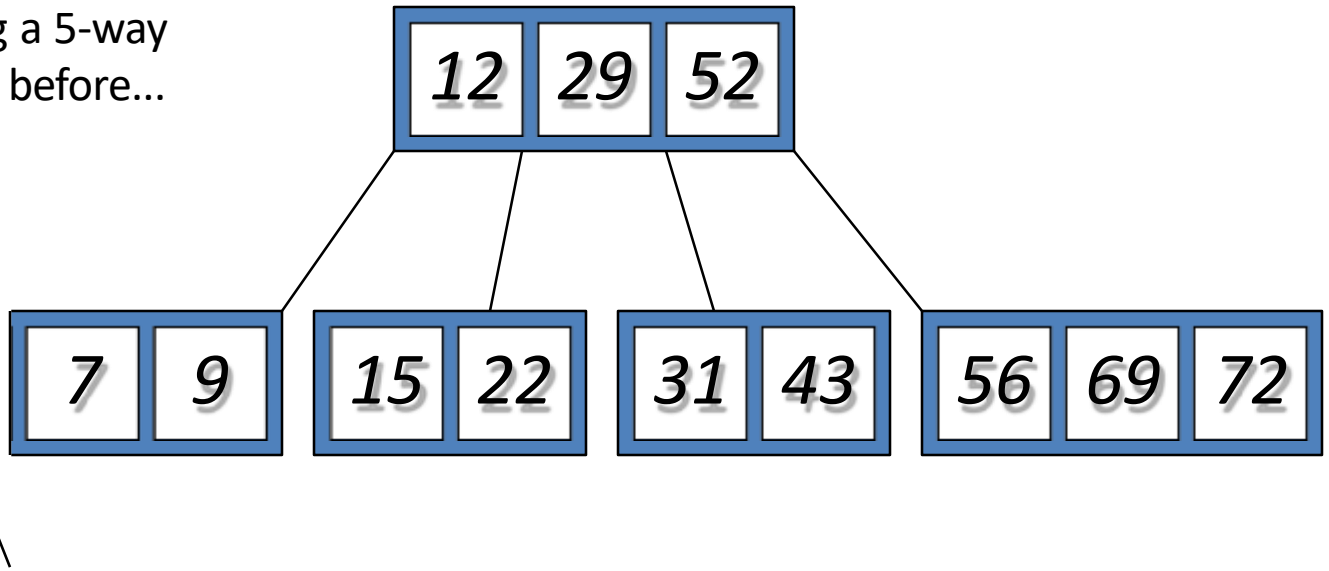
- During insertion, the key always goes *into* a *leaf*. For deletion we wish to remove *from* a leaf. There are three possible ways we can do this:
- 1 - If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted.
- 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

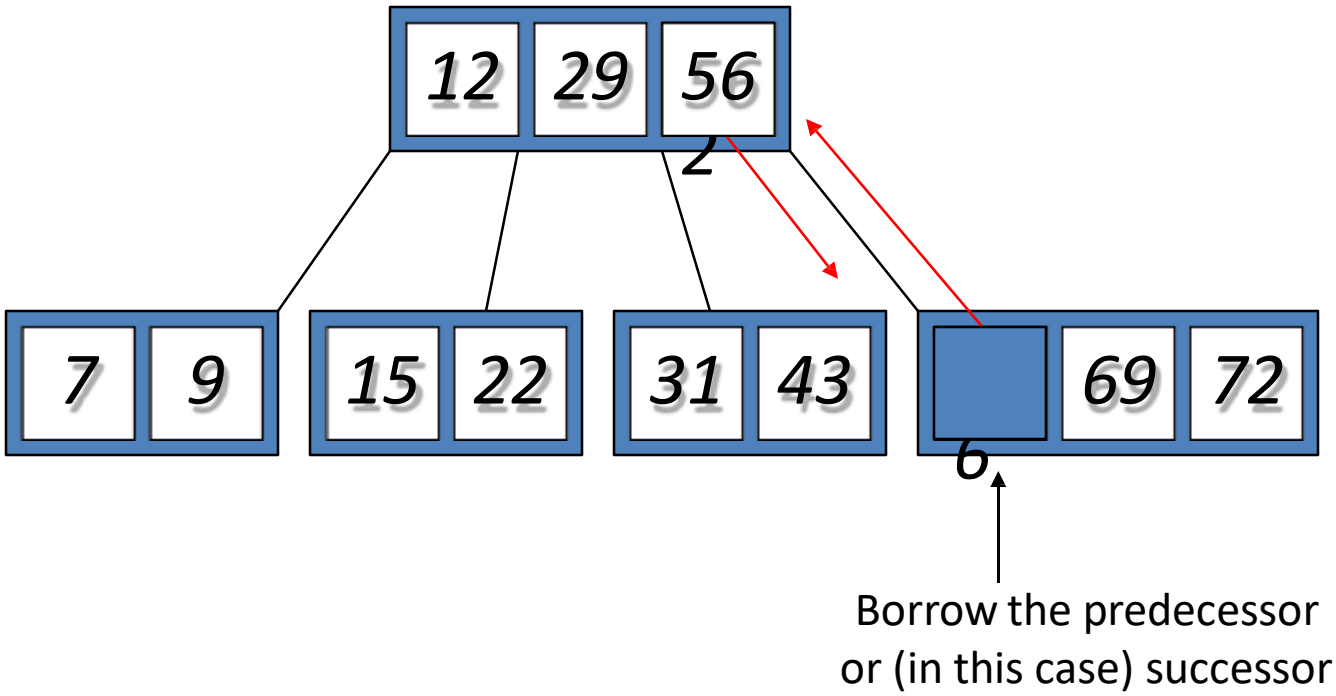
Type #1: Simple leaf deletion

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

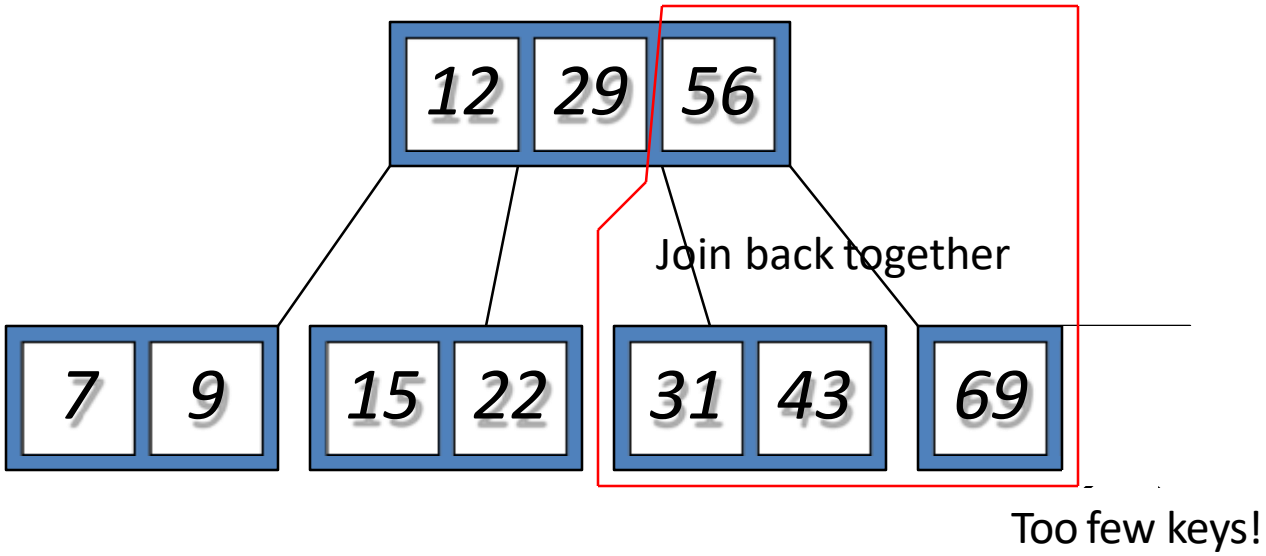


Delete 2: Since there are enough keys in the node, just delete it

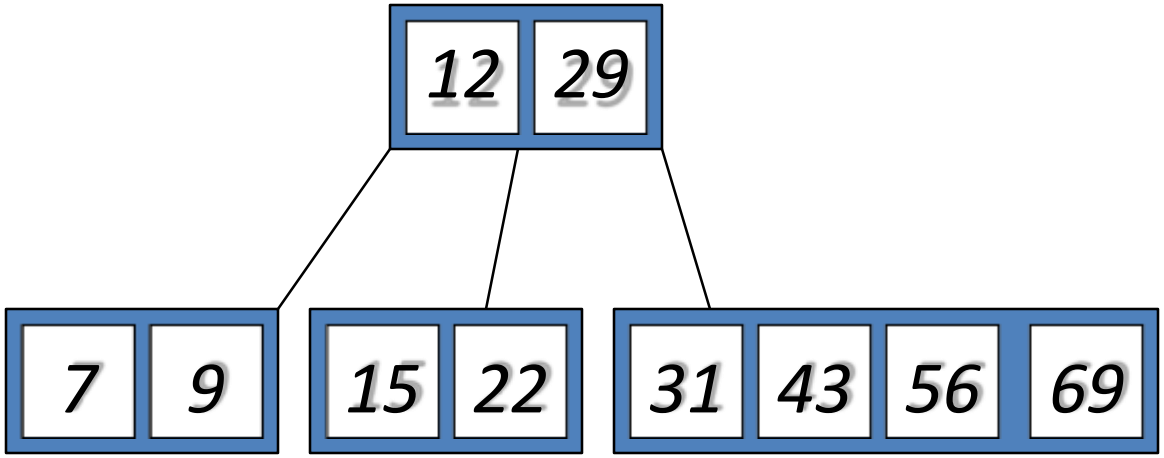
Type #2: Simple non-leaf deletion



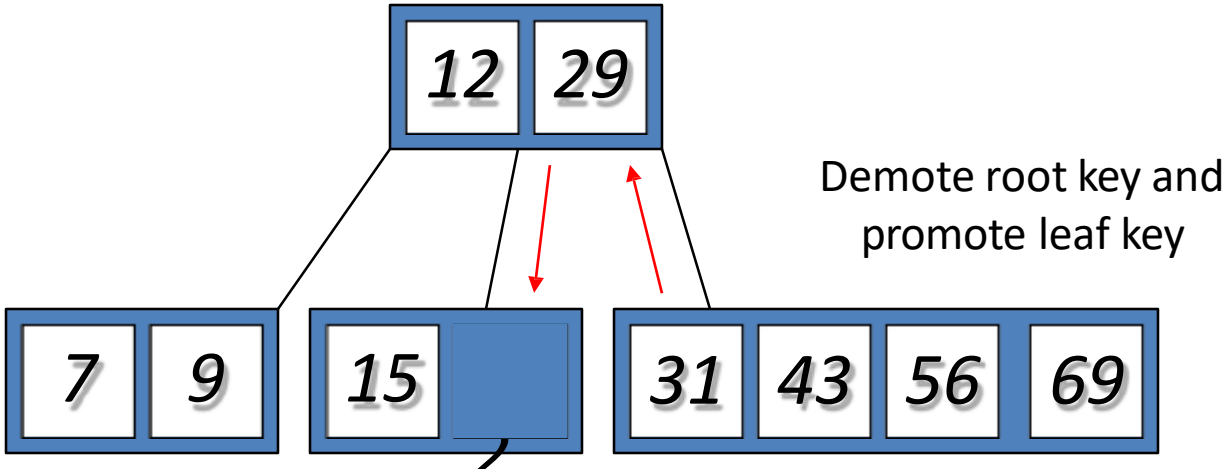
Type #4: Too few keys in node and its siblings



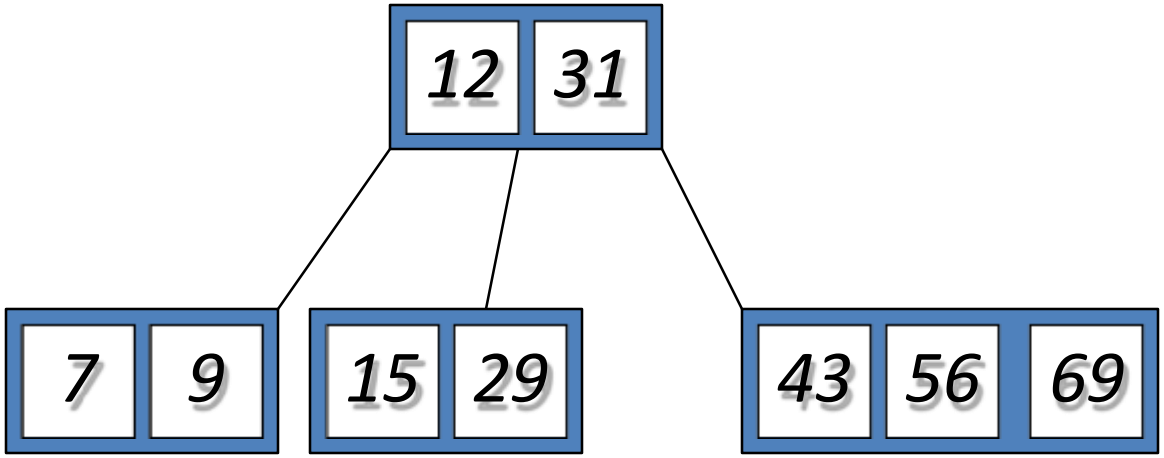
Type #4: Too few keys in node and its siblings



Type #3: Enough siblings



Type #3: Enough siblings

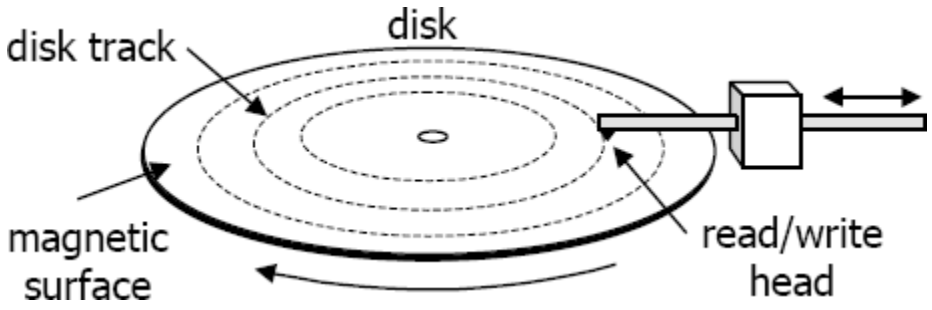


Summary

- 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.

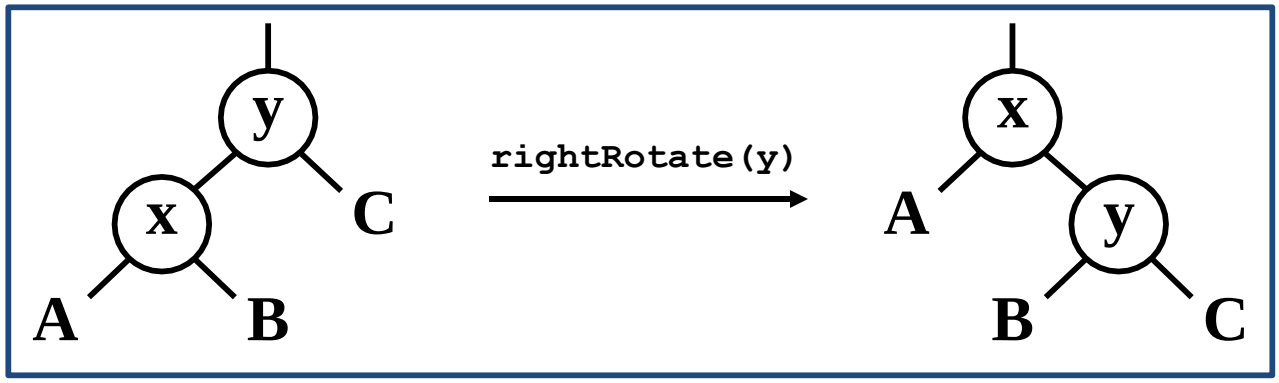
MS/Dos - FAT (File allocation table)

- 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



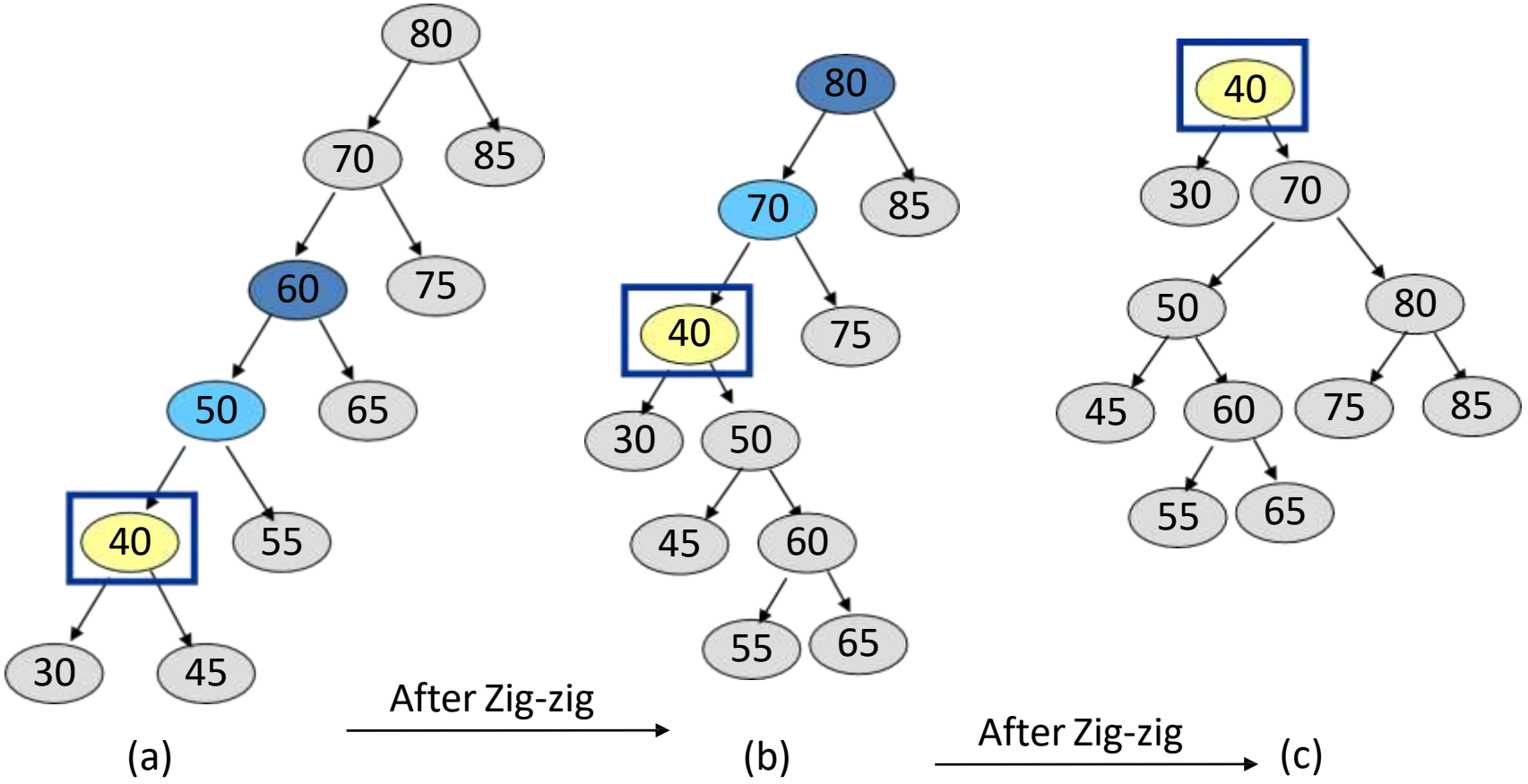
Secondary Storages

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?

Splay Trees: Example – 40 is accessed



Comparison of Search Trees

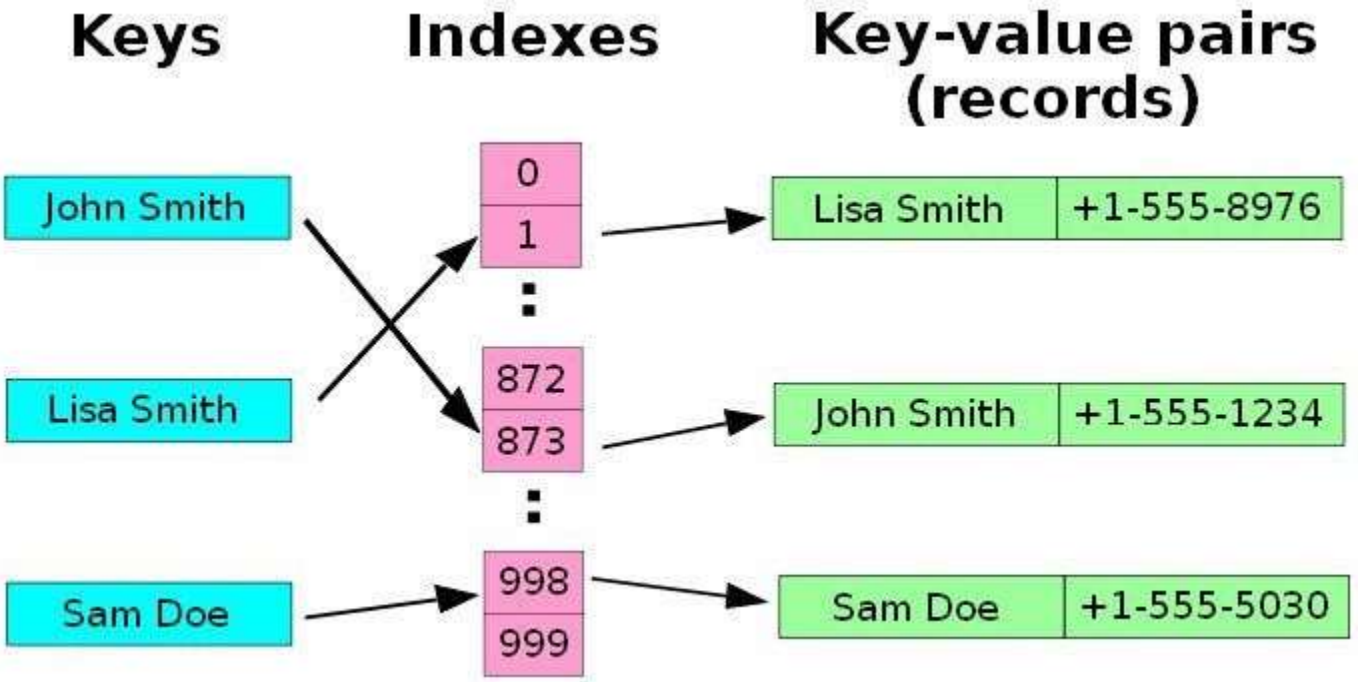
Tree	Worst Case			Expected		
	Search	Insert	Remove	Search	Insert	Remove
BST	n	n	n	log n	log n	log n
AVL tree	log n	log n	log n	log n	log n	log n
red-black tree	log n	log n	log n	log n	log n	log n
splay tree	n	n	n	log n	log n	log n
B-trees	log n	log n	log n	log n	log n	log n

Concept of Hashing

- In CS, a **hash table**, or a **hash map**, is a data structure that associates keys (names) with values (attributes).

  - Look-Up Table
  - Dictionary
  - Cache
  - Extended Array

Example


Keys
Indexes
Key-value pairs (records)
John Smith
Lisa Smith
Sam Doe
0
1
872
873
998
999
Lisa Smith
+1-555-8976
John Smith
+1-555-1234
Sam Doe
+1-555-5030

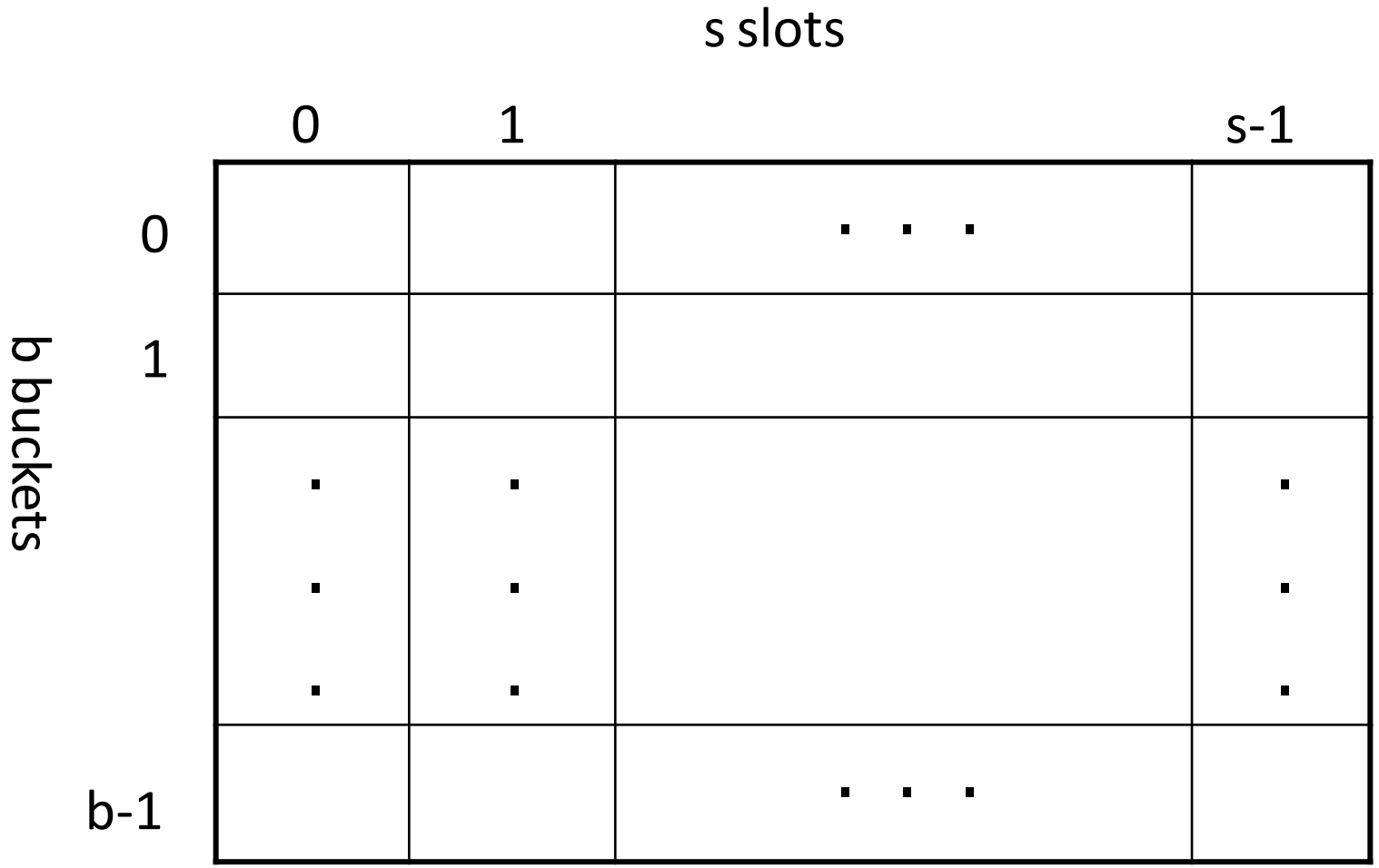
Search vs. Hashing

- Search tree methods: key comparisons
  - Time complexity: O(size) or O(log n)
- Hashing methods: hash functions
  - Expected time: O(1)
- Types
  - Static hashing (section 8.2)
  - Dynamic hashing (section 8.3)

Static Hashing

- Key-value pairs are stored in a fixed size table called a *hash table*.
  - A hash table is partitioned into many ***buckets***.
  - Each bucket has many ***slots***.
  - Each slot holds one record.
  - A hash function f(x) transforms the identifier (key) into an address in the hash table

Hash table



Data Structure for Hash Table

```
#define MAX_CHAR  10
#define TABLE_SIZE  13
typedef struct {
    char key[MAX_CHAR];
    /* other fields */
} element;
element hash_table[TABLE_SIZE];
```

Some Issues

- **Choice of hash function.**
  - ***Really tricky!***
  - To avoid collision (two different pairs are in the same the same bucket.)
  - Size (number of buckets) of hash table.
- **Overflow handling method.**
  - Overflow: there is no space in the bucket for the new pair.

Example (fig 8.1)

synonyms:
char, ceil,
clock, ctime

overflow

	Slot 0	Slot 1
0	acos	atan synonyms
1		
2	char	ceil synonyms
3	define	
4	exp	
5	float	floor
6		
…		
25		

Choice of Hash Function

- Requirements
  - easy to compute
  - minimal number of collisions
- If a hashing function groups key values together, this is called clustering of the keys.
- A good hashing function distributes the key values uniformly throughout the range.

Some hash functions

- Middle of square
  - H(x):= return middle digits of x^2
- Division
  - H(x):= return x % k
- Multiplicative:
  - H(x):= return the first few digits of the fractional part of x*k, where k is a fraction.

Some hash functions II

- Folding:
  - Partition the identifier x into several parts, and add the parts together to obtain the hash address
  - e.g. x=12320324111220; partition x into 123,203,241,112,20; then return the address 123+203+241+112+20=699
  - Shift folding vs. folding at the boundaries
- Digit analysis:
  - If all the keys have been known in advance, then we could delete the digits of keys having the most skewed distributions, and use the rest digits as hash address.

Overflow Handling

- An overflow occurs when the home bucket for a new pair (key, element) is full.
- We may handle overflows by:
  - Search the hash table in some systematic fashion for a bucket that is not full.
    - .

  - Linear probing (linear open addressing).
  - Quadratic probing.
  - Random probing.
- Eliminate overflows by permitting each bucket to keep a list of all pairs for which it is the home bucket.
  - Array linear list.
  - Chain

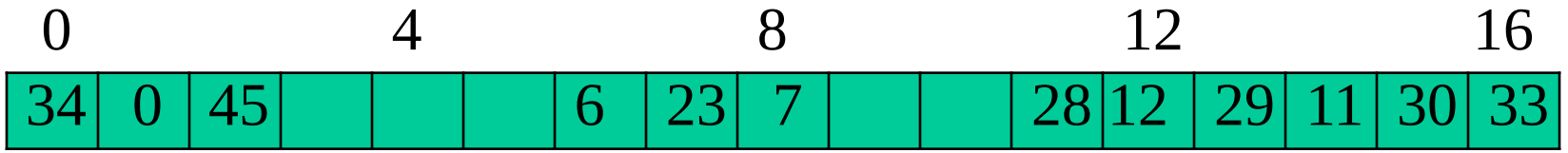
Linear probing (linear open addressing)

- **Open addressing** ensures that all elements are stored directly into the hash table, thus it attempts to resolve collisions using various methods.

- **Linear Probing** resolves collisions by placing the data into the next open slot in the table.

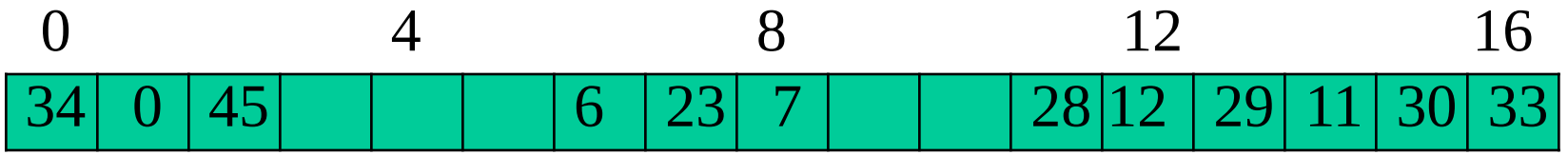
Linear Probing – Get And Insert

- divisor = b (number of buckets) = 17.
- Home bucket = key % 17.

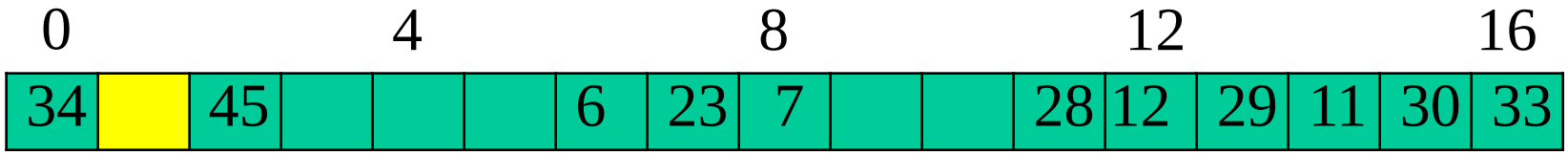


- Insert pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

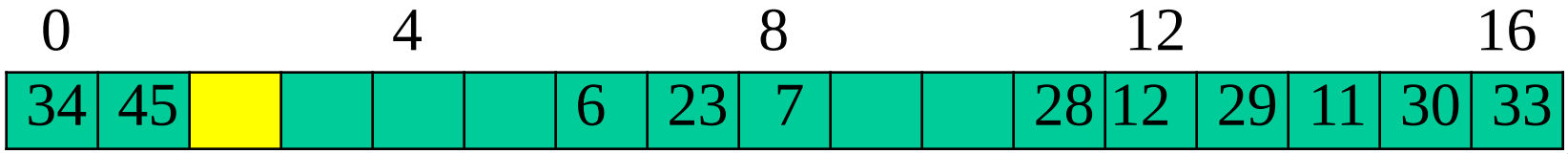
Linear Probing – Delete



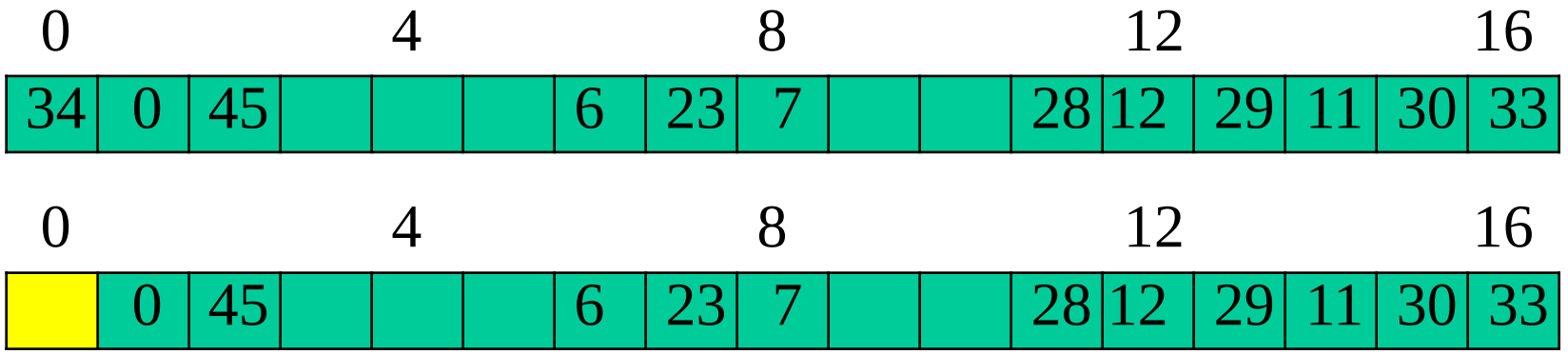
- Delete(0)



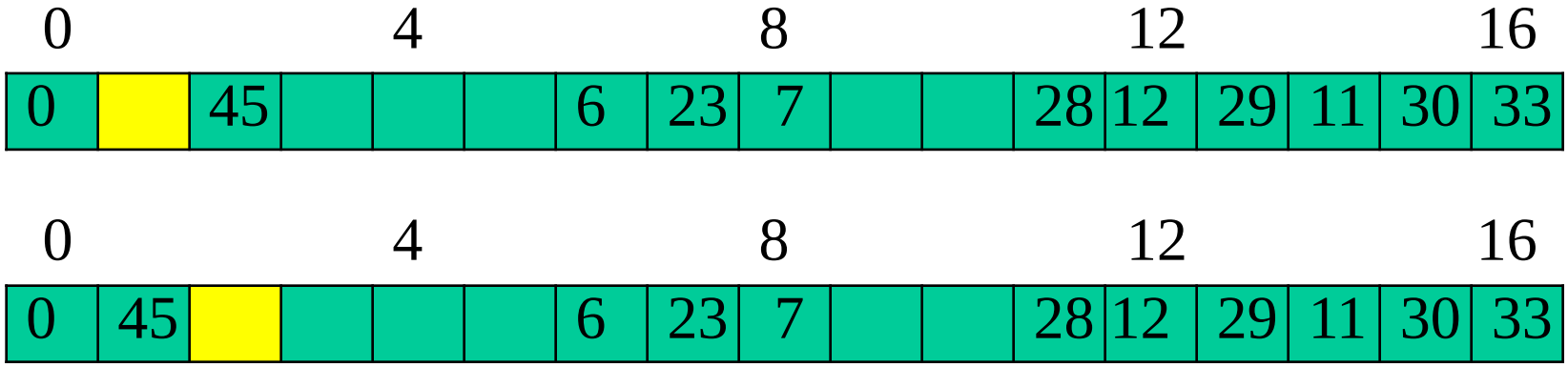
- Search cluster for pair (if any) to fill vacated bucket.



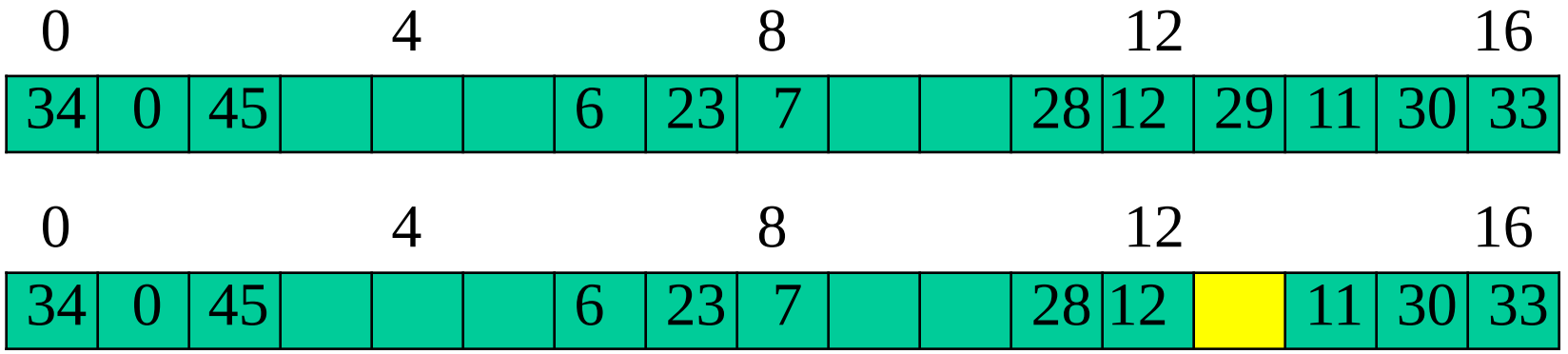
Linear Probing – Delete(34)



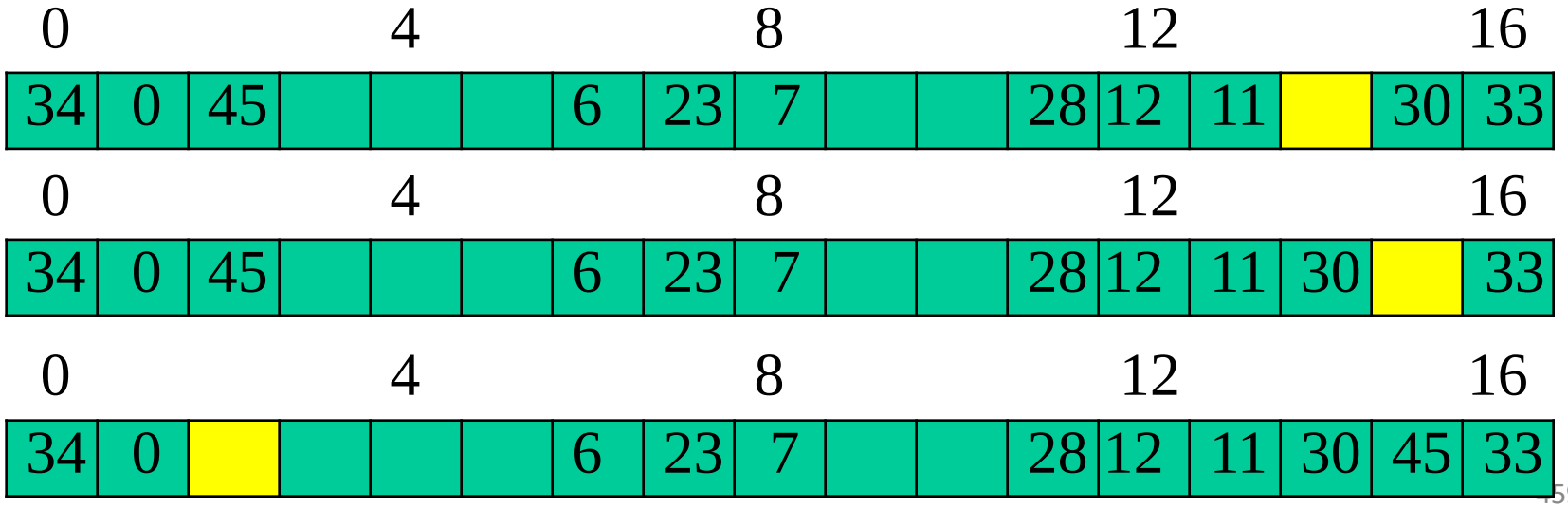
- Search cluster for pair (if any) to fill vacated bucket.



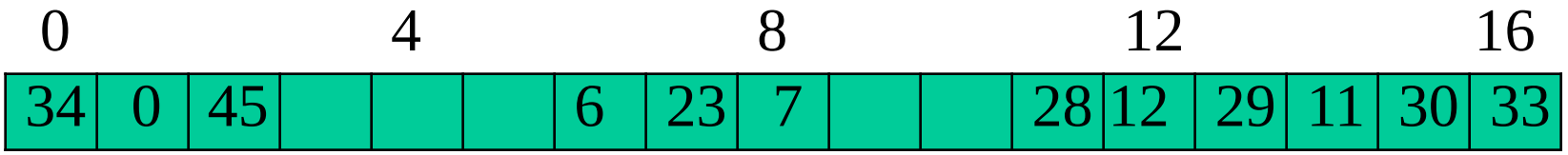
Linear Probing – Delete(29)



- Search cluster for pair (if any) to fill vacated bucket.



Performance Of Linear Probing



- Worst-case find/insert/erase time is $\Theta(n)$, where n is the number of pairs in the table.
- This happens when all pairs are in the same cluster.

Problem of Linear Probing

- Identifiers tend to cluster together
- Adjacent cluster tend to coalesce
- Increase the search time

Quadratic Probing

- Linear probing searches buckets $(H(x)+i^2)\%b$
- Quadratic probing uses a quadratic function of *i* as the increment
- Examine buckets $H(x)$, $(H(x)+i^2)\%b$, $(H(x)-i^2)\%b$, for $1<=i<=(b-1)/2$
- b is a prime number of the form 4j+3, j is an integer

Random Probing

- Random Probing works incorporating with random numbers.
  - H(x):= (H'(x) + S[i]) % b
  - S[i] is a table with size b-1
  - S[i] is a random permuation of integers [1,b-1].

Some Applications of Hash Tables

- **Database systems**: Specifically, those that require efficient random access. Generally, database systems try to optimize between two types of access methods: sequential and random. Hash tables are an important part of efficient random access because they provide a way to locate data in a constant amount of time.

Some Applications of Hash Tables

- **Symbol tables**: The tables used by compilers to maintain information about symbols from a program. Compilers access information about symbols frequently. Therefore, it is important that symbol tables be implemented very efficiently.

Some Applications of Hash Tables

- **Data dictionaries**: Data structures that support adding, deleting, and searching for data. Although the operations of a hash table and a data dictionary are similar, other data structures may be used to implement data dictionaries. Using a hash table is particularly efficient.

Some Applications of Hash Tables

- **Network processing algorithms**: Hash tables are fundamental components of several network processing algorithms and applications, including route lookup, packet classification, and network monitoring.

- **Browser Cashes**: Hash tables are used to implement browser cashes.

Problems for Which Hash Tables are not Suitable

1.Problems for which data ordering is required.

Because a hash table is an unordered data structure, certain operations are difficult and expensive. Range queries, proximity queries, selection, and sorted traversals are possible only if the keys are copied into a sorted data structure. There are hash table implementation that keep the keys in order, but they are far from efficient.

Problems for Which Hash Tables are not Suitable

- 2. Problems having **multidimensional data**.

- 3. **Prefix searching** especially if the keys are long and of variable-lengths.

Problems for Which Hash Tables are not Suitable

- 4. **Problems that have dynamic data**:
- Open-addressed hash tables are based on 1D-arrays, which are difficult to resize
- once they have been allocated. Unless you want to implement the table as a
- dynamic array and rehash all of the keys whenever the size changes. This is an
- incredibly expensive operation. An alternative is use a separate-chained hash tables or dynamic hashing.

Problems for Which Hash Tables are not Suitable

- 5. **Problems in which the data does not have unique keys.**
- Open-addressed hash tables cannot be used if the data does not have unique keys. An alternative is use separate-chained hash tables.