Data Structures

VIVEN

Embedded Academy

Why Data structures

- Data structures is concerned with the representation and manipulation of data.
- All programs manipulate data. So, all programs represent data in some way.
- Data manipulation requires an algorithm.

Types of Data Structures

- Arrays
- Linked Lists
- Stacks
- Queues
- Trees
- Graphs

VIVEN
Embedded Academy

Self-referential structures

- **It is a structure which contains one member which is a pointer of its own type.**

Example:

```
struct student
{
    char name[12];
    int rollno;
    float fees;
    struct student *next;
};
```

A pointer to a structure of the same type

Linked Lists

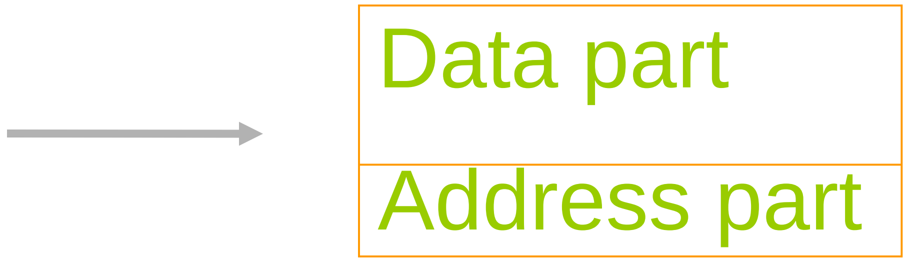
- Basic idea
  - a linked list is a chain of nodes
  - each node references the next node in chain
  - last node references NULL.

- Components in a linked list
  - Node – data to store
  - Head – first node in the list
  - New – the newly created node
  - Ptr – to navigate across the list

Node

- Node stores a piece of data
- Node stores a reference to another node

```
struct sample
{
  int data;
  Char name[10];
  struct list*next;
};
```



Note: here we have declared a node called
list. “node” is not a keyword

Example

```
#include <stdio.h>

struct list

{

  int data;

  struct list *next;

};
```

In this example:

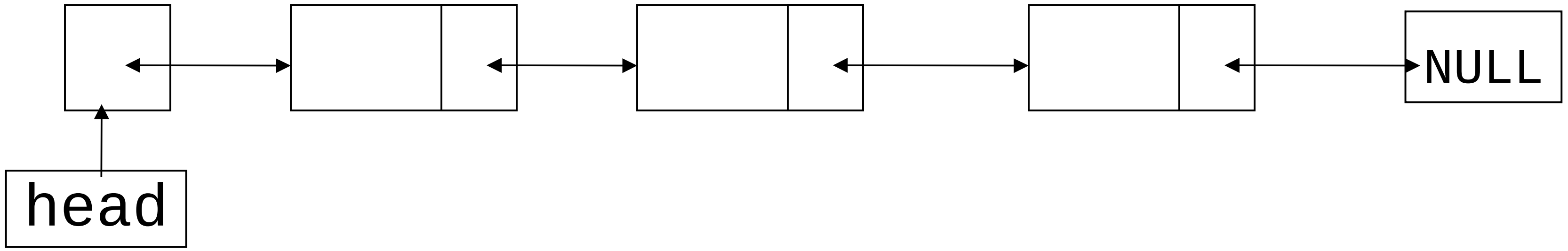
- *head: this pointer is used to store the base address of the list

- *p: this pointer is used to navigate through the list.

- *new: this is used to store the address of the newly allocated node;

Linked List ADT(Abstract Data Type)

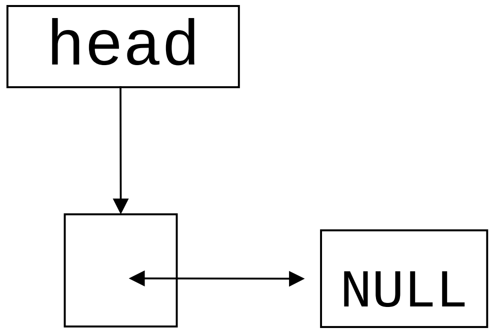
Linked list: set of data structures (nodes) that contain references to other data structures.



- Linked list contains 0 or more nodes
- Has a list head to point to first node
- Last node points to NULL

Empty List

- If a list currently contains 0 nodes, it is the <u>empty list.</u>
- In this case the list head points to `NULL`.



Declaring a Node

- Declare a node:

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

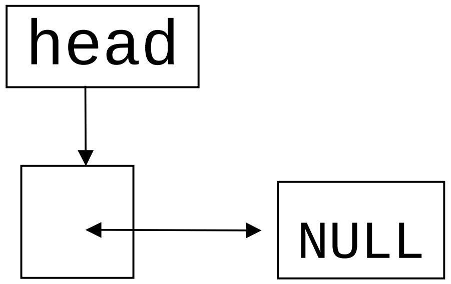
- No memory is allocated at this time

Defining a Linked List

- Define a pointer for the head of the list:

```
    ListNode *head = NULL;
```

- Head pointer initialized to `NULL` to indicate an empty list



Create a New Node

- Allocate memory for the new node:

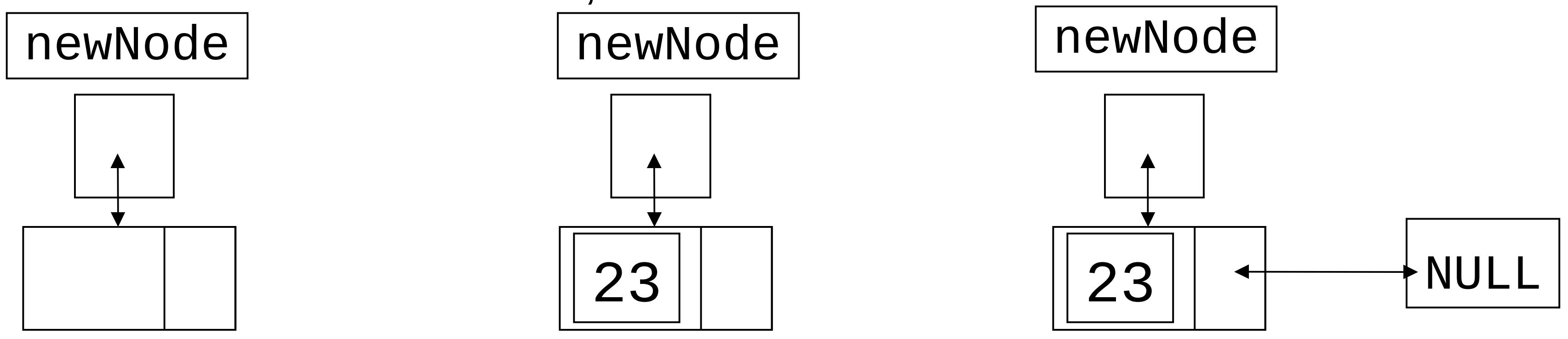
```
newNode = malloc(sizeof(struct ListNode));
```

- Initialize the contents of the node:

```
newNode->value = num;
```

- Set the pointer field to NULL:

```
newNode->next = NULL;
```



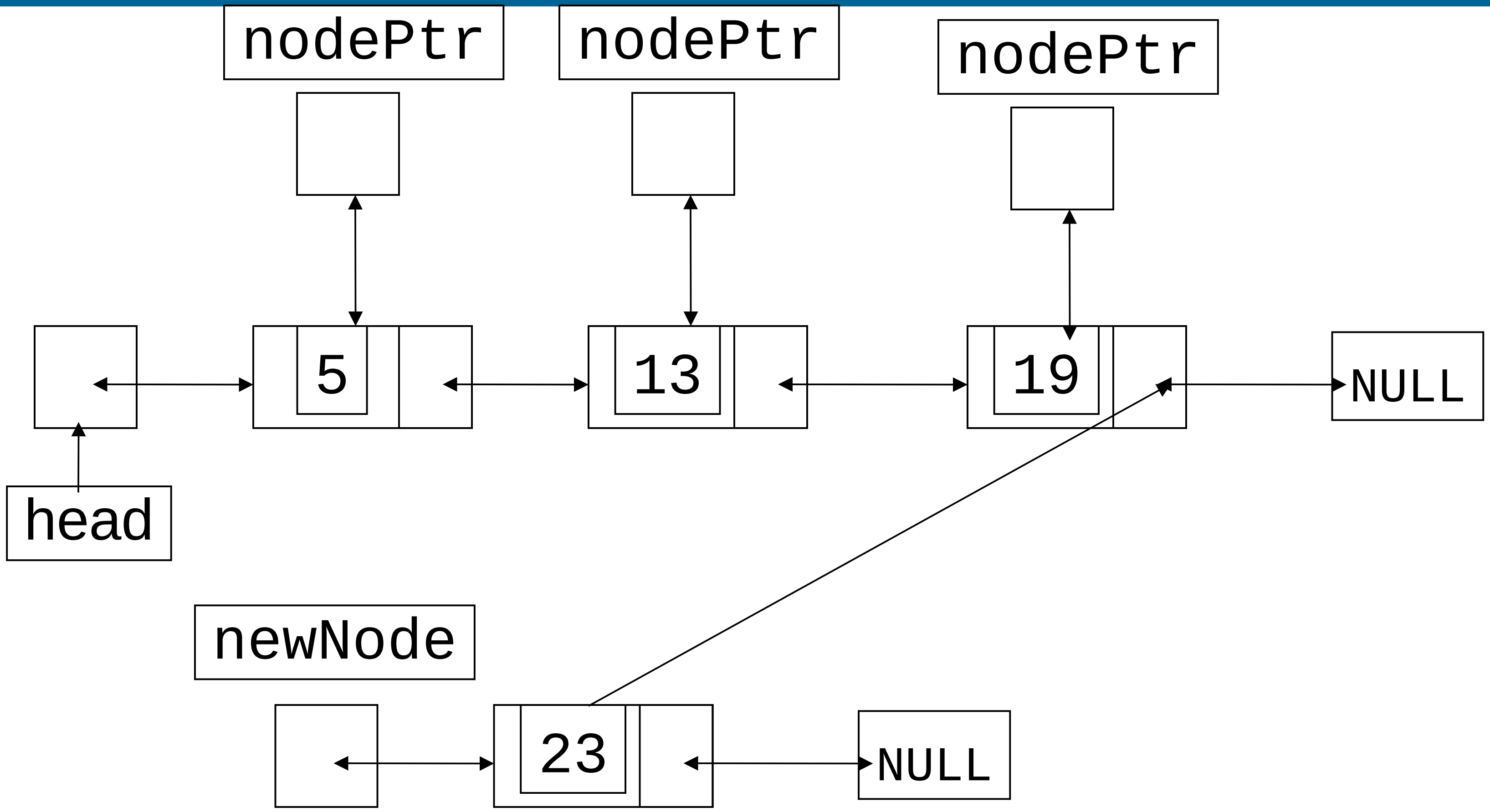
Linked List Operations

- Basic operations:
  - Add a node to head of the list.
  - append a node to the end of the list
  - traverse the linked list
  - insert a node within the list
  - delete a node
  - delete/destroy the list
  - Search (sorted and unsorted linked list)
  - Simple recursion

Appending a Node

- Add a node to the end of the list
- Basic process:
  - Create the new node (as already described)
  - Add node to the end of the list:
    - If list is empty, set head pointer to this node
    - Else,
      - traverse the list to the end
      - set pointer of last node to point to new node

Appending a Node

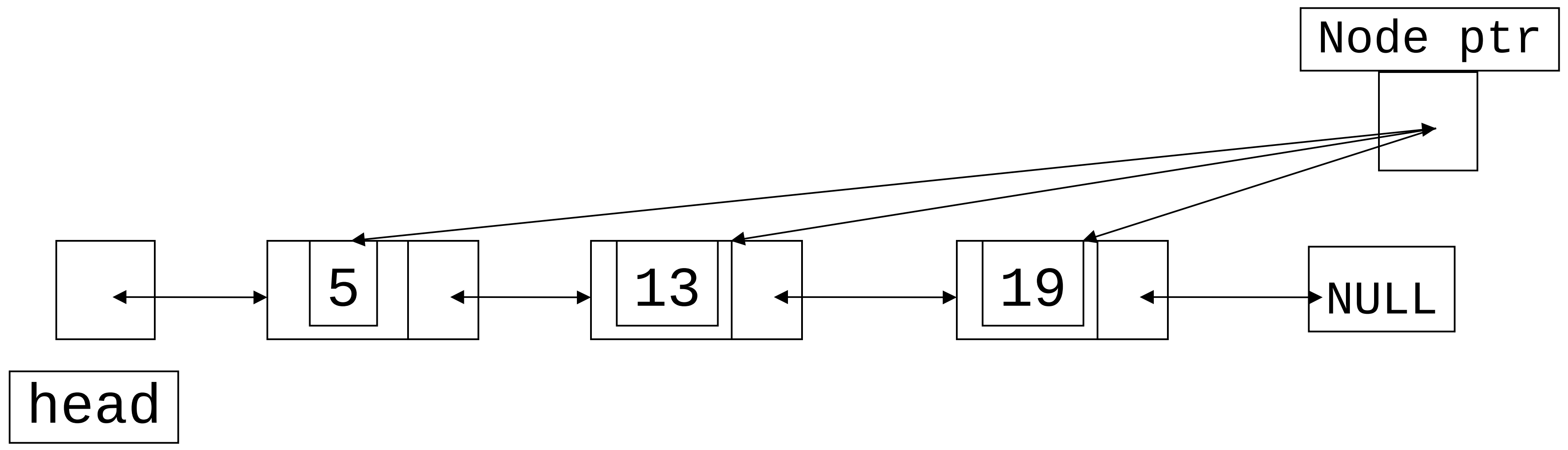


1. New node created, end of list located.
2. New node added to end of list

Traversing a Linked List

- Visit each node in a linked list: display contents, validate data, etc.
- Basic process:
  - set a pointer to the contents of the head pointer
  - while pointer is not `NULL`
    - process data
    - set pointer to the pointer field of the current node in the list
  - end while

Traversing a Linked List



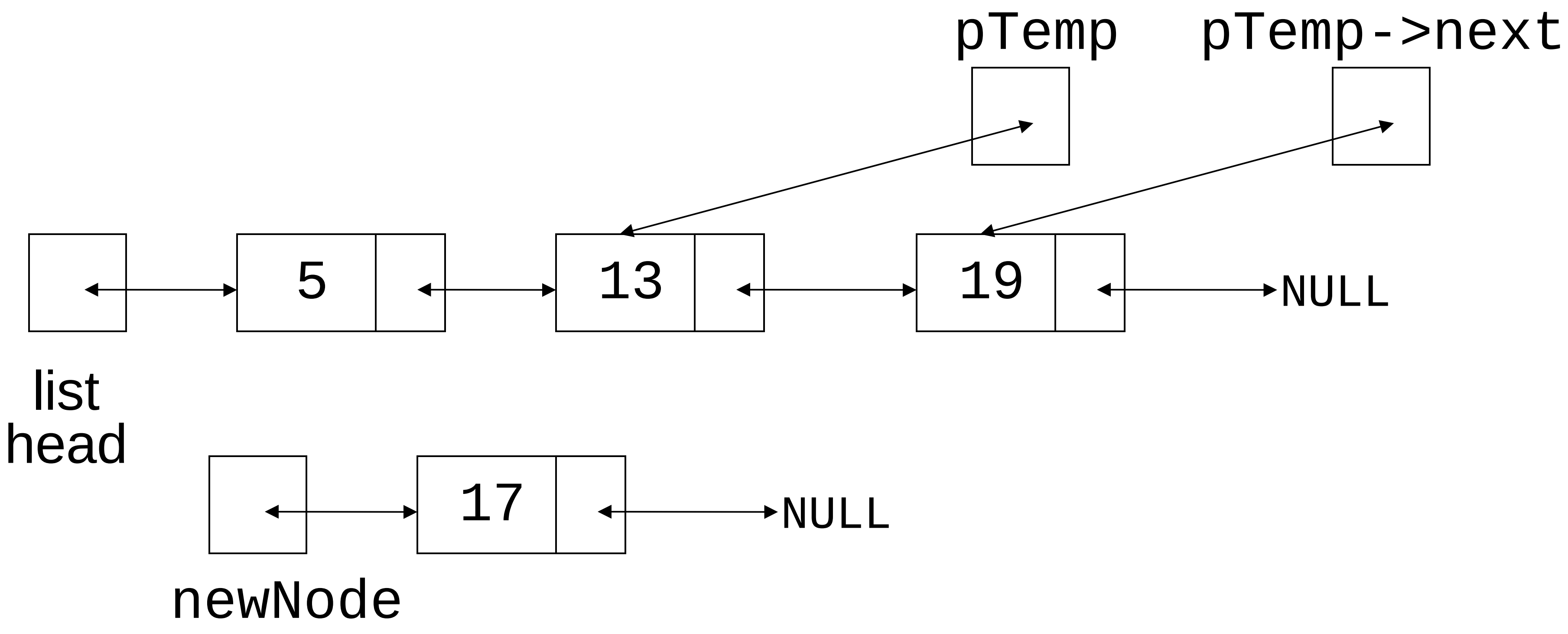
1. First nodePtr points to the node containing 5.
2. Then nodePtr points to the node containing 13.
3. Then nodePtr points to the node containing 19.
4. Then nodePtr points to NULL and the list traversal stops

Inserting a Node into a Linked list

- Maintain a linked list in order.
- Need to check two conditions.
  - Check whether the value pointed to by the head node is greater than the present value. If so add the new node to head of the linked list.
  - Use pTemp pointer to traverse the list and see whether the value pointed to by next node(with reference to the pTemp) is greater than the new nodes value & New node is inserted between.

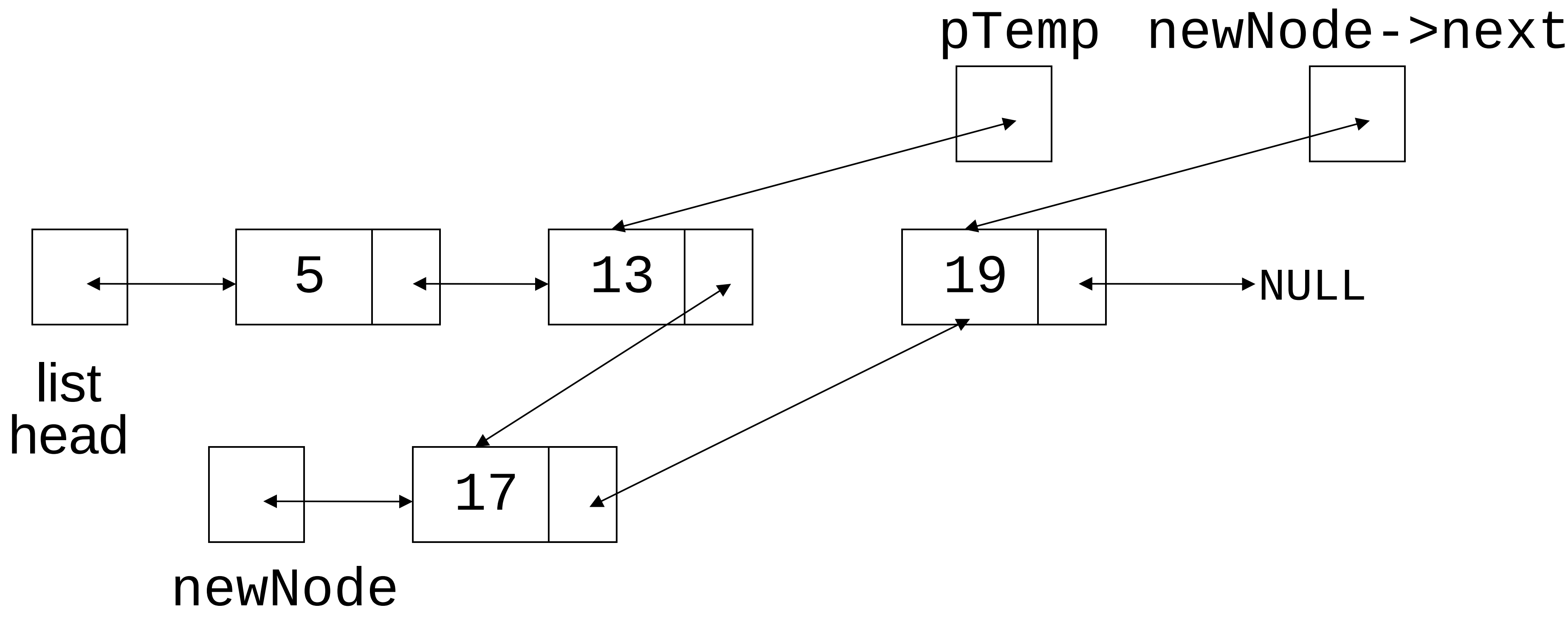
VIVEN
Embedded Academy

Inserting a Node into a Linked List



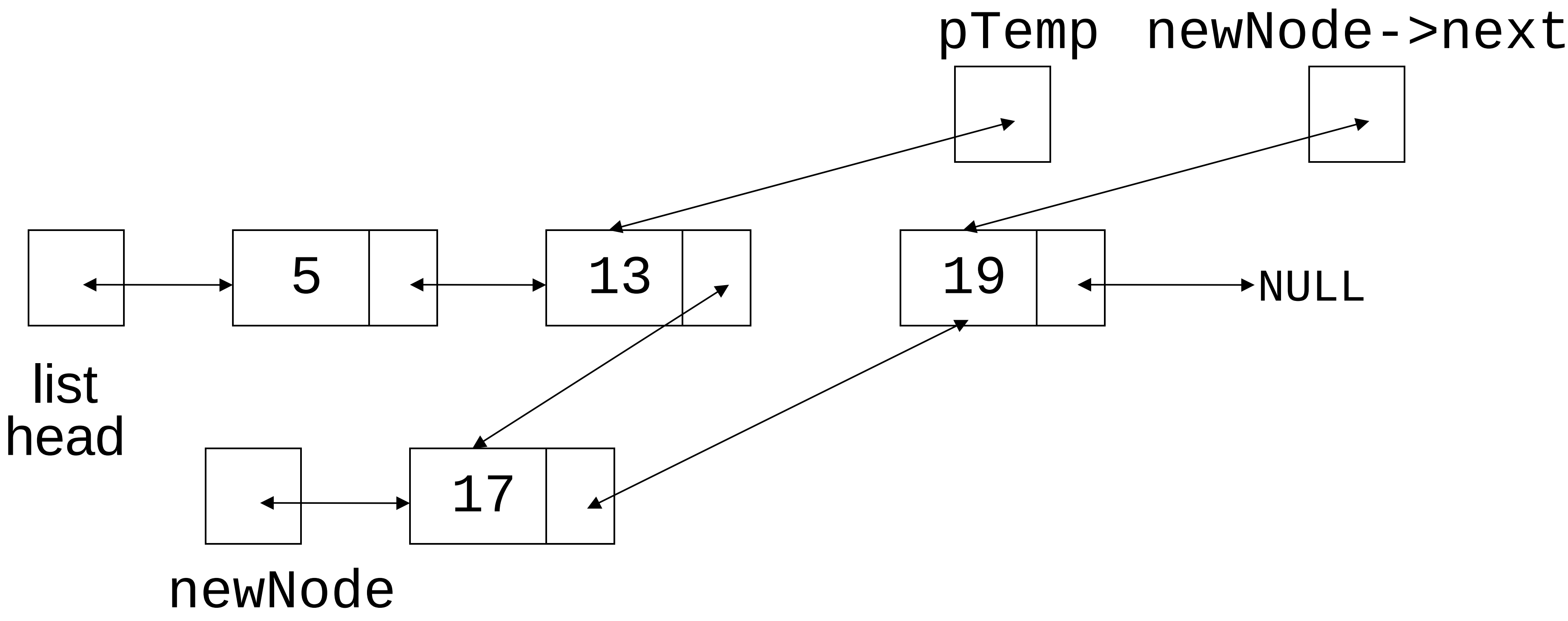
New node created, correct position located

Inserting a Node into a Linked List*



New node inserted in order in the linked list

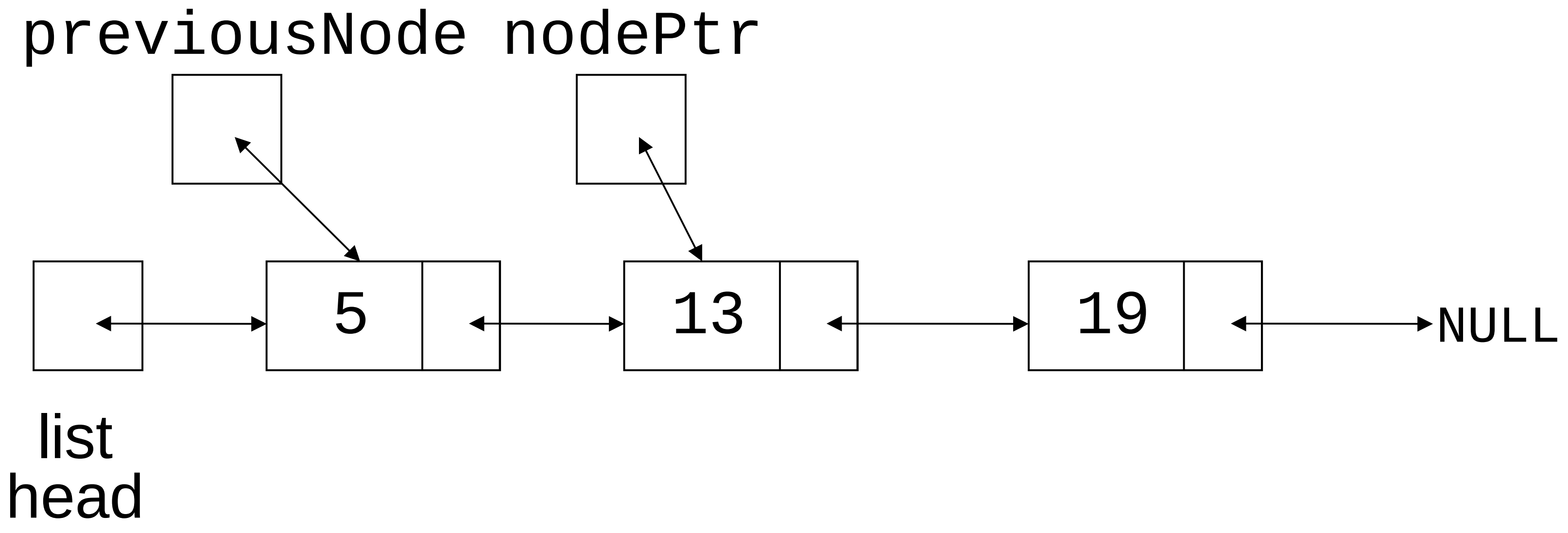
Inserting a Node into a Linked List*



New node inserted in order in the linked list

VIVEN Embedded Academy

Deleting a Node

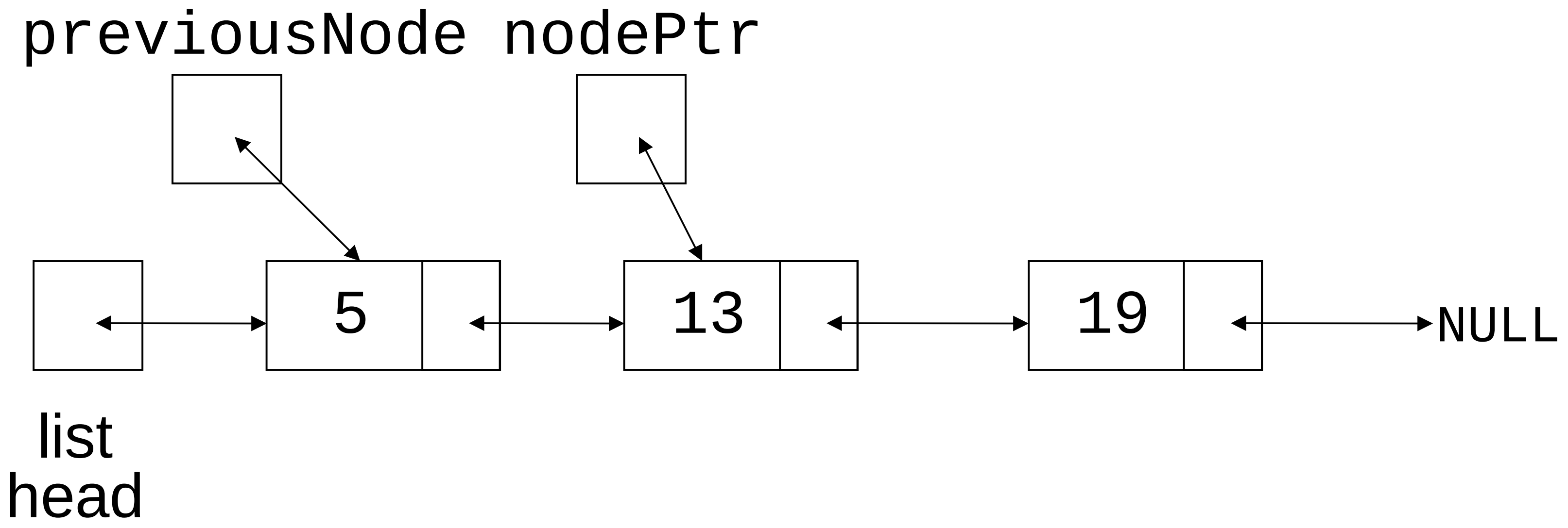


Locating the node containing 13

Destroying a Linked List

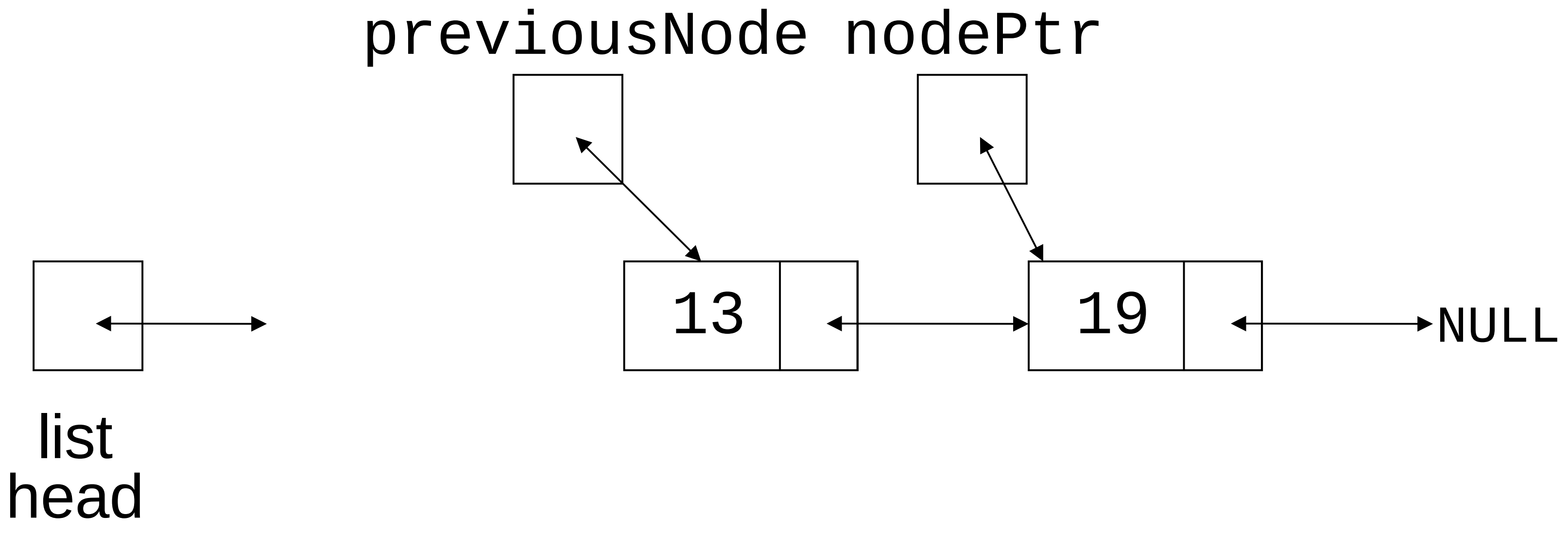
- Must remove all nodes used in the list
- To do this, use list traversal to visit each node
- For each node,
  - Unlink the node from the list
  - If the list uses dynamic memory, then free the node’s memory
- Set the list head to NULL
- Can be done iteratively and recursively

Destroying a Linked List



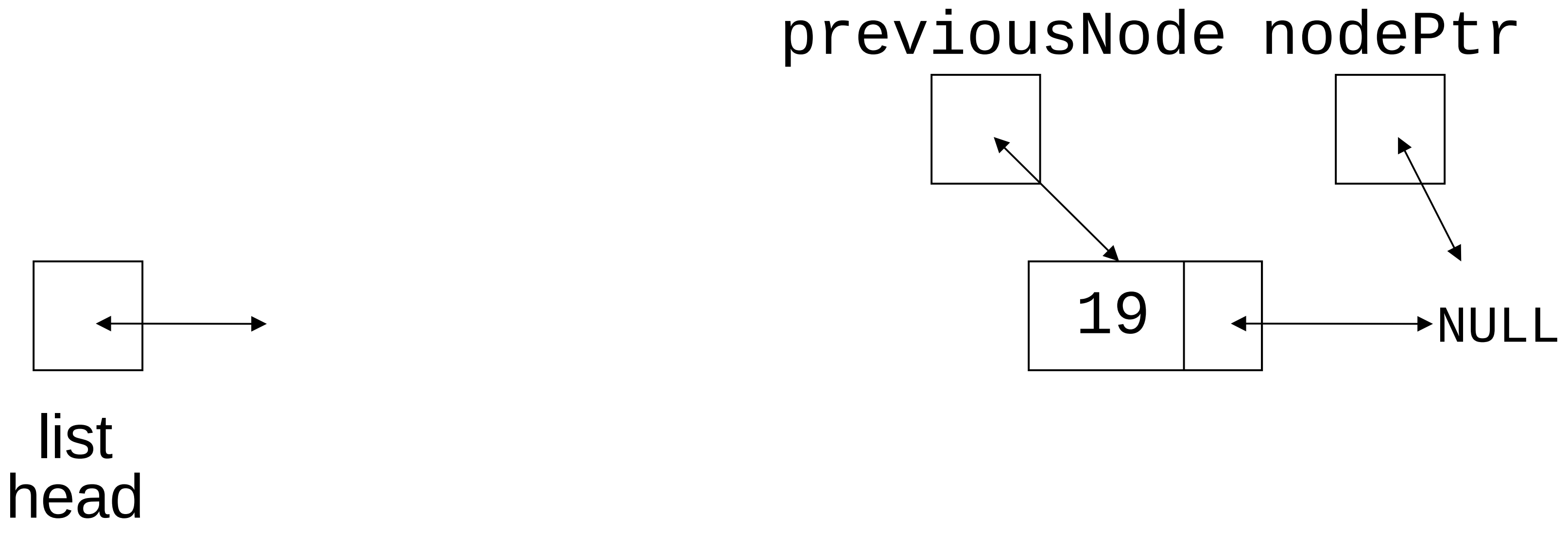
Set `previousNode` to point same address as head
Set `nodePtr` to point to `previousNode->next`

Destroying a Linked List



Destroy the node containing 13

Destroying a Linked List



Destroy the node containing 19

Searching a Linked List*

- An unsorted list must be searched to the end
- A sorted list is searched until one occurs:
  - End of the list is reached → return false
  - Search key is = to value in the node→ return true
  - Search key is > than value in node
- For efficiency, sort the linked list as you build it

Recursive Linked List Operations

- Recursive functions can be members of a linked list class
- Normally use indirect recursion
- Default base case is almost always the end of the list
- Some applications:
  - Traverse the list in reverse order
  - Compute the size of (number of nodes in) a list

Contents of a List Reverse order

- Algorithm:
  - pointer starts at head of list
  - If the pointer is NULL, return (base case)
  - If the pointer is not NULL, advance to next node
  - Upon returning from recursive call, display contents of current node

Creation of a Singly Linked List

VIVEN Embedded Academy

1. If the value of `head` (pointer used to point to the base address of the list.) is equal to `NULL`, it implies that it does not store the address of any node, which means that the list is empty. Now insert the first node. This involves two steps –

2. Creating a new node (let this node be called new1) i,e. allocating memory for the node and accept data into it.

3. Storing the address of the node created in head .
   head = new1;

Insertion in a List

Initialize, allocate memory and accept data into a node.
(called new1).

Make new1's next point to what head is pointing to, i.e., the first node in the list. This can be done by
new1 -> next = head;

Make a `head` point to `new1` (i.e., make it the first node). This is done by making
```
start = new1
```

Appending to the list

Initialize, allocate memory and accept data into a node. (called new1).

Initialize p to head.
p=head;

Locate the end of the list using a loop

```
while (p->next!=NULL)
p=p->next;
```

At the end of the loop p will be pointing to the last node.

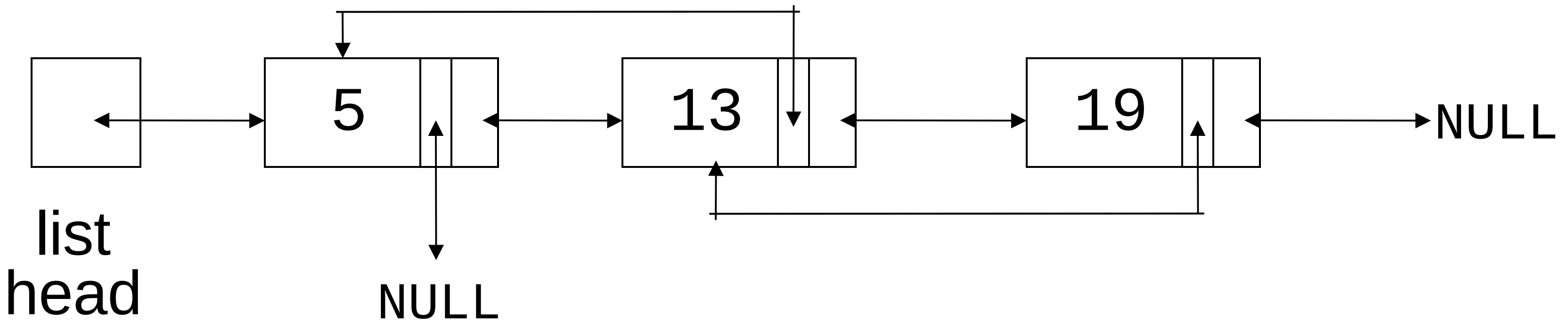
Add the new node to the end of the list by initializing p->next to the address of new1.
p->next=new;
new->next=NULL;

Types of Linked List

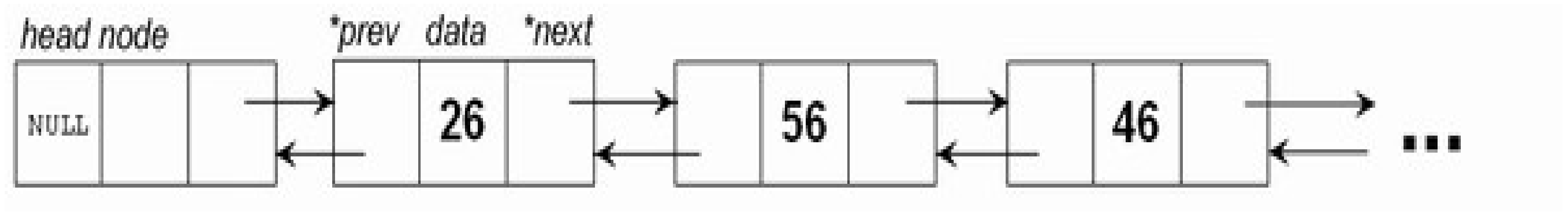
- Single
- Doubly
- Circular

Doubly Linked List

- doubly-linked list: each node contains two pointers: one to the next node in the list, one to the previous node in the list

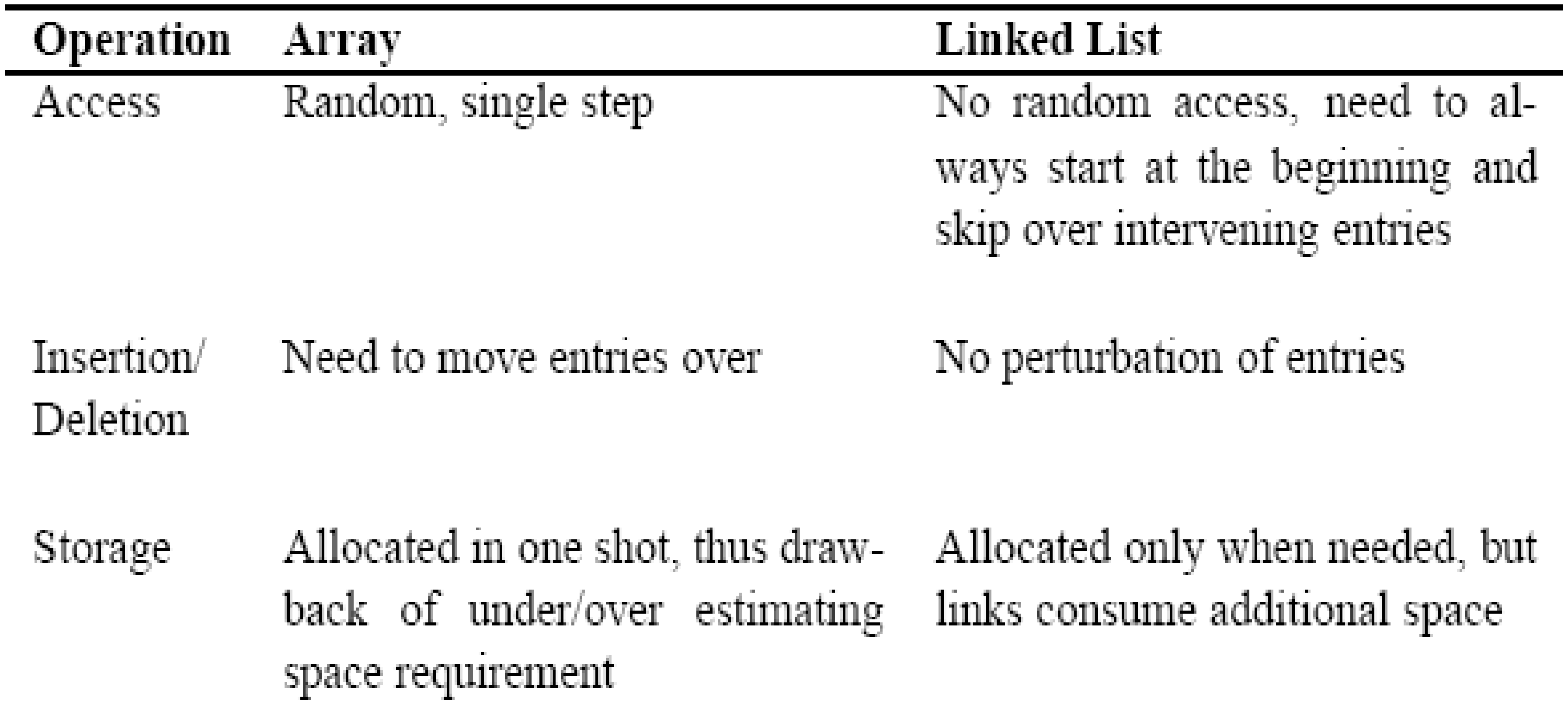


Continued…



```
struct student
 {
     char name[12];
     float fees;
     struct student *next;
     struct student *Prev;
 };
```

Arrays vs LL



Operation	Array	Linked List
Access	Random, single step	No random access, need to always start at the beginning and skip over intervening entries
Insertion/ Deletion	Need to move entries over	No perturbation of entries
Storage	Allocated in one shot, thus drawback of under/over estimating space requirement	Allocated only when needed, but links consume additional space

Arrays	Linked lists
Typically arrays are allocated statically, that is during compile time. So these are called static data structures.	Linked lists are typically created using dynamic allocation(that is created during run time). So these are called dynamic data structures.

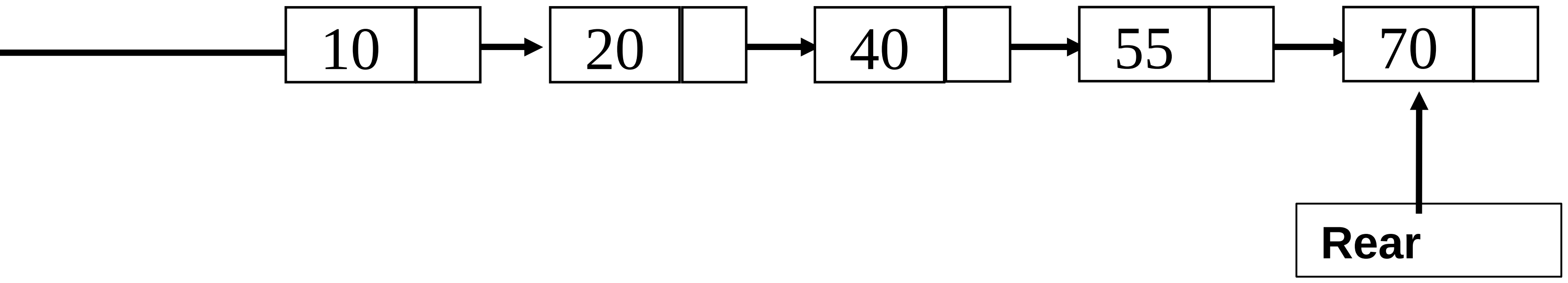
Double Linked list vs Single Linked lists

Double Linked List	Single Linked list
Can be traversed in both directions	Traversal in only one direction

Circular Linked List

Circular Linked Lists

- A Circular Linked List is a special type of Linked List
- It supports traversing from the end of the list to the beginning by making the last node point back to the head of the list
- A Rear pointer is often used instead of a Head pointer



Motivation

- Circular linked lists are usually sorted
- Circular linked lists are useful for playing video and sound files in “looping” mode
- They are also a stepping stone to implementing graphs, an important topic in comp171

Circular Linked List Definition

```
#include <iostream>
using namespace std;

struct Node{
  int data;
  Node* next;
};
typedef Node* NodePtr;
```

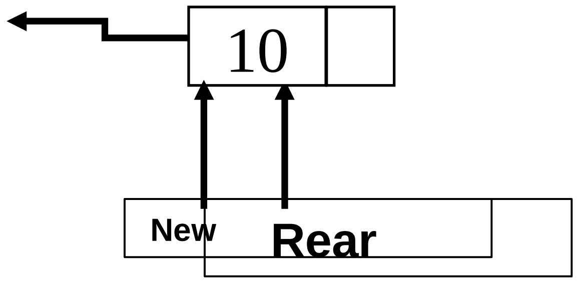
Circular Linked List Operations

```
insertNode(NodePtr& Rear, int item)
  //add new node to ordered circular linked list

deleteNode(NodePtr& Rear, int item)
  //remove a node from circular linked list

print(NodePtr Rear)
  //print the Circular Linked List once
```

Insert Node

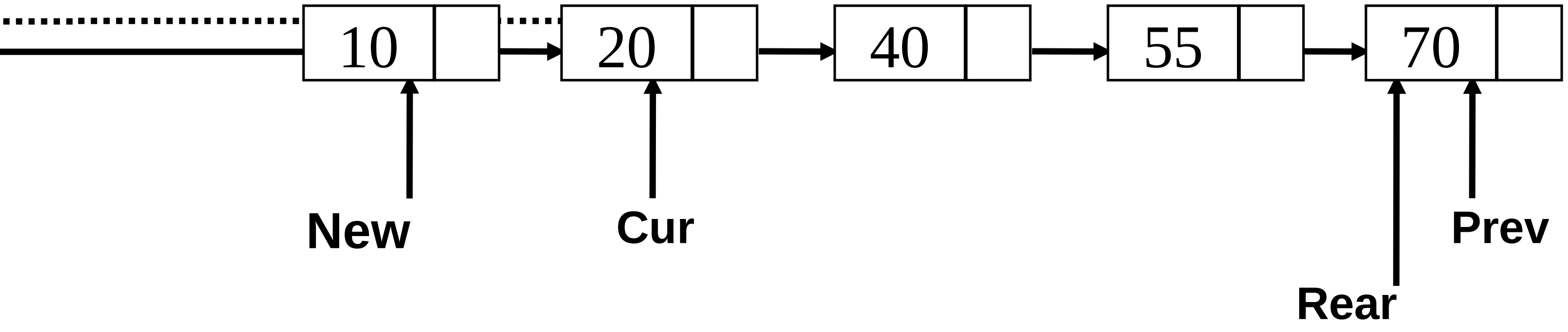
- Insert into an empty list

```
NotePtr New = new Node;
New->data = 10;

Rear = New;
Rear->next = Rear;
```



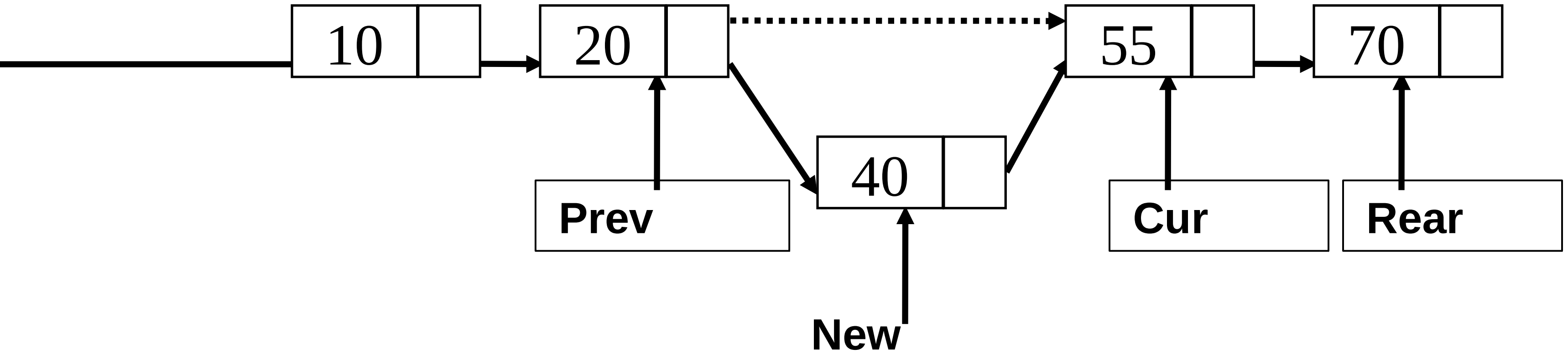
- Insert to head of a Circular Linked List

```
New->next = Cur;  // same as: New->next = Rear->next;
Prev->next = New; // same as: Rear->next = New;
```



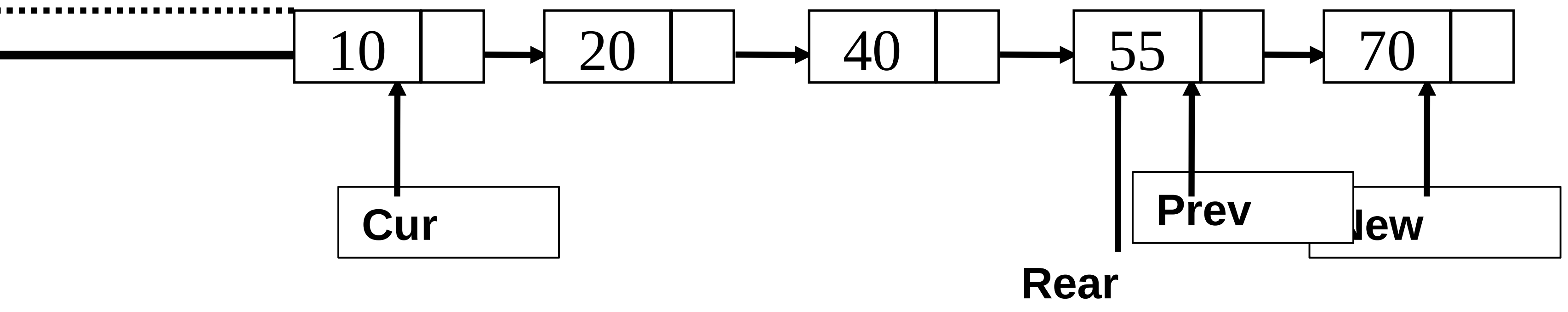
- Insert to middle of a Circular Linked List between `Pre` and `Cur`

```
New->next = Cur;

Prev->next = New;
```



- Insert to end of a Circular Linked List

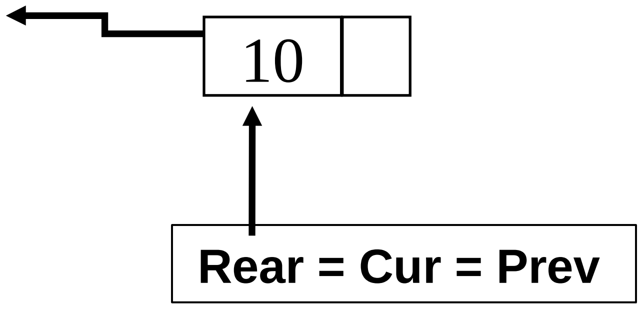
```
New->next = Cur;    // same as: New->next = Rear->next;
Prev->next = New;   // same as: Rear->next = New;
Rear = New;
```



```
void insertNode(NodePtr& Rear, int item){
    NodePtr  New, Cur, Prev;
    New = new Node;
    New->data = item;
    if(Rear == NULL){    // insert into empty list
     Rear = New;
     Rear->next = Rear;
     return;
    }
    Prev = Rear;
    Cur = Rear->next;
    do{                  // find Prev and Cur
     if(item <= Cur->data)
          break;
     Prev = Cur;
     Cur = Cur->next;
    }while(Cur != Rear->next);
    New->next = Cur;     // revise pointers
    Prev->next = New;
    if(item > Rear->data)     //revise Rear pointer if adding to end
     Rear = New;
}
```

- Delete a node from a single-node Circular Linked List

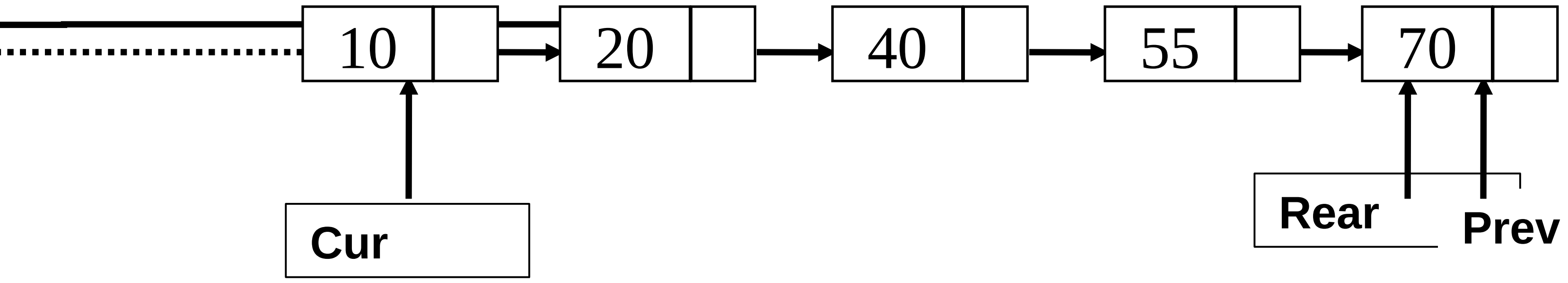
```
Rear = NULL;
delete Cur;
```



Delete Node

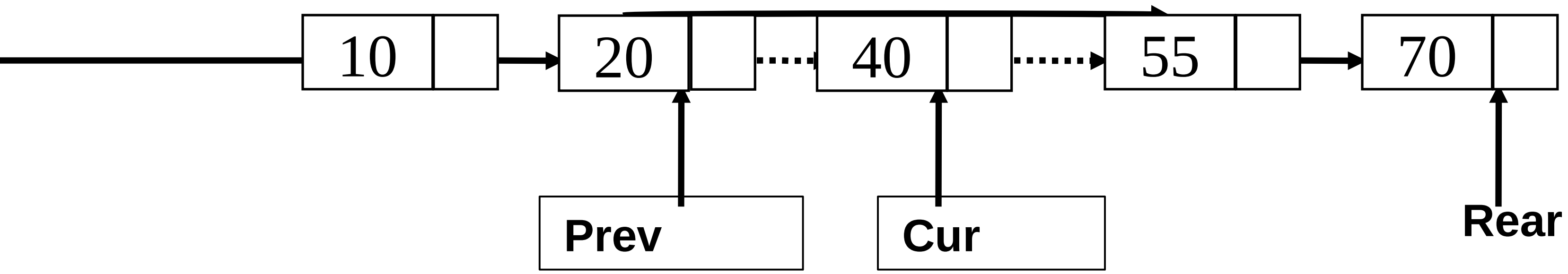
- Delete the head node from a Circular Linked List

```
Prev->next = Cur->next; // same as: Rear->next = Cur->next

delete Cur;
```



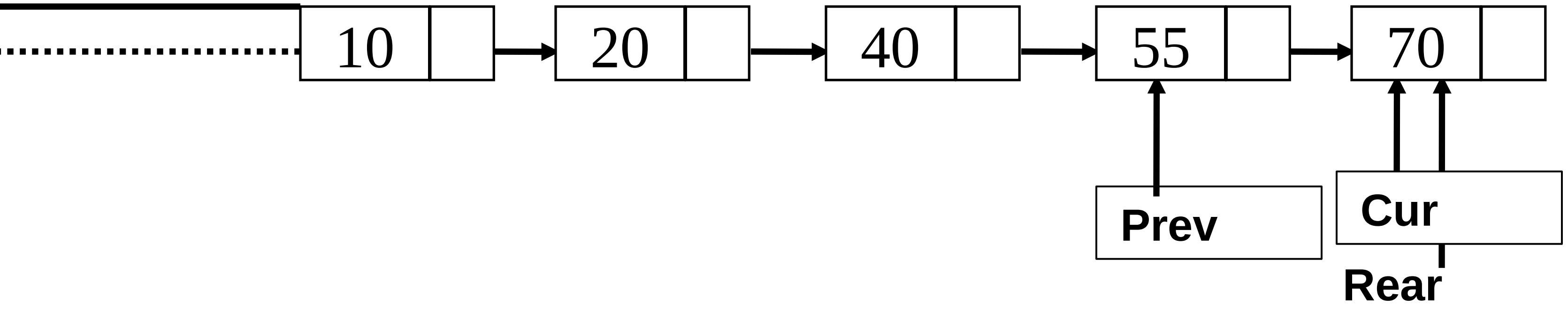
- Delete a middle node `Cur` from a Circular Linked List

```
Prev->next = Cur->next;

delete Cur;
```



- Delete the end node from a Circular Linked List

```
Prev->next = Cur->next;   // same as: Rear->next;

delete Cur;

Rear = Prev;
```



VIVEN Embedded Academy

TREES

Trees

- Linear access time of linked lists is prohibitive
    - Does there exist any simple data structure for which the running time of most operations (search, insert, delete) is O(log N)?

Trees

- A tree is a hierarchical collection of nodes
  - The collection can be empty
  - (recursive definition) If not empty, a tree consists of a distinguished node r (the *root*), and zero or more nonempty *subtrees* T_1, T_2, ..., T_k, each of whose roots are connected by a directed *edge* from r

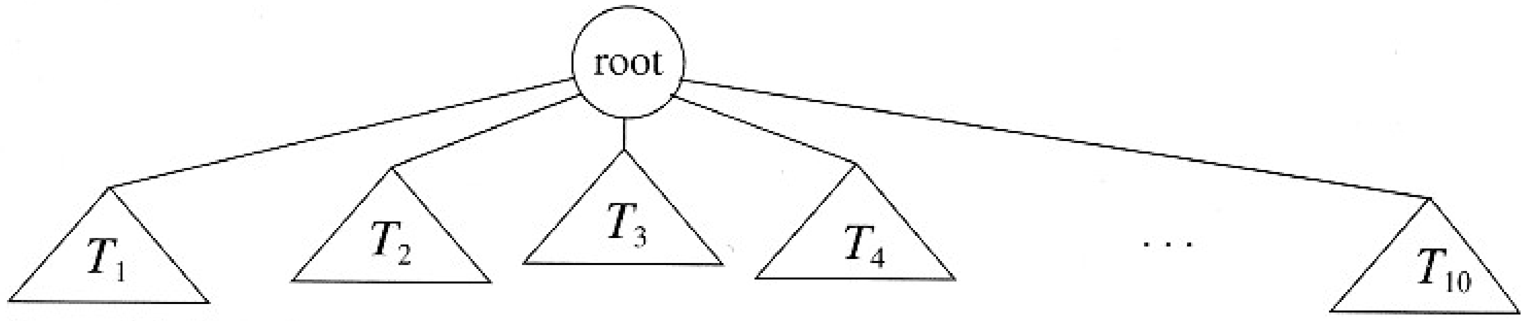


Figure 4.1 Generic tree

Some Terminologies

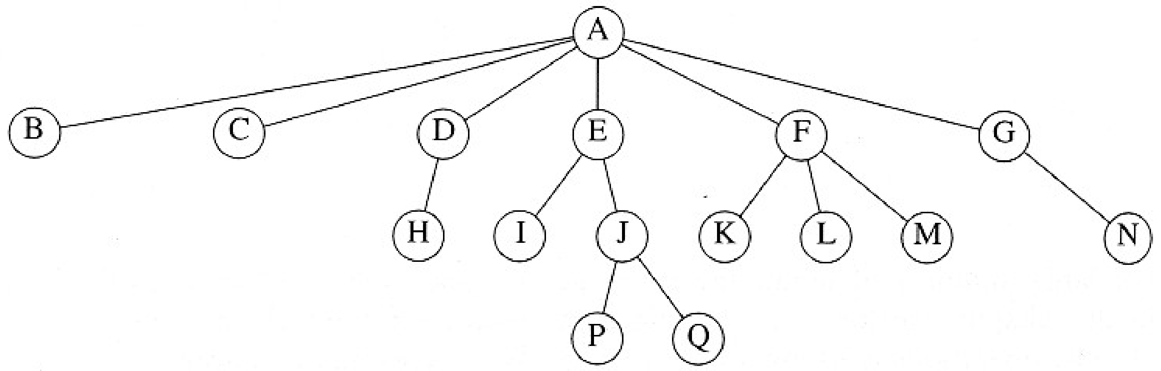


Figure 4.2 A tree

- ***Child*** **and** ***parent***
  - **Every node except the root has one parent**
  - **A node can have an arbitrary number of children**
- ***Leaves***
  - **Nodes with no children**
- ***Sibling***
  - **nodes with same parent**

Some Terminologies

- ***Path***
- ***Length***
    - **The length of the path is one less than the number of nodes in the path.**

Depth **of a node**

- **Number of nodes along the path from the root to the node.**
- **The depth of a tree is equal to the depth of the deepest leaf**

- ***Height*** **of a node**
    - **The maximum level in a tree determines its height. The height of a node in a tree is the length of the longest path from that node to a leaf**
    - **all leaves are at height 0**
    - **The height of a tree is equal to the height of the root**
- ***Ancestor*** **and** ***descendant***
    - ***Proper ancestor*** **and** ***proper descendant***

Example: UNIX Directory

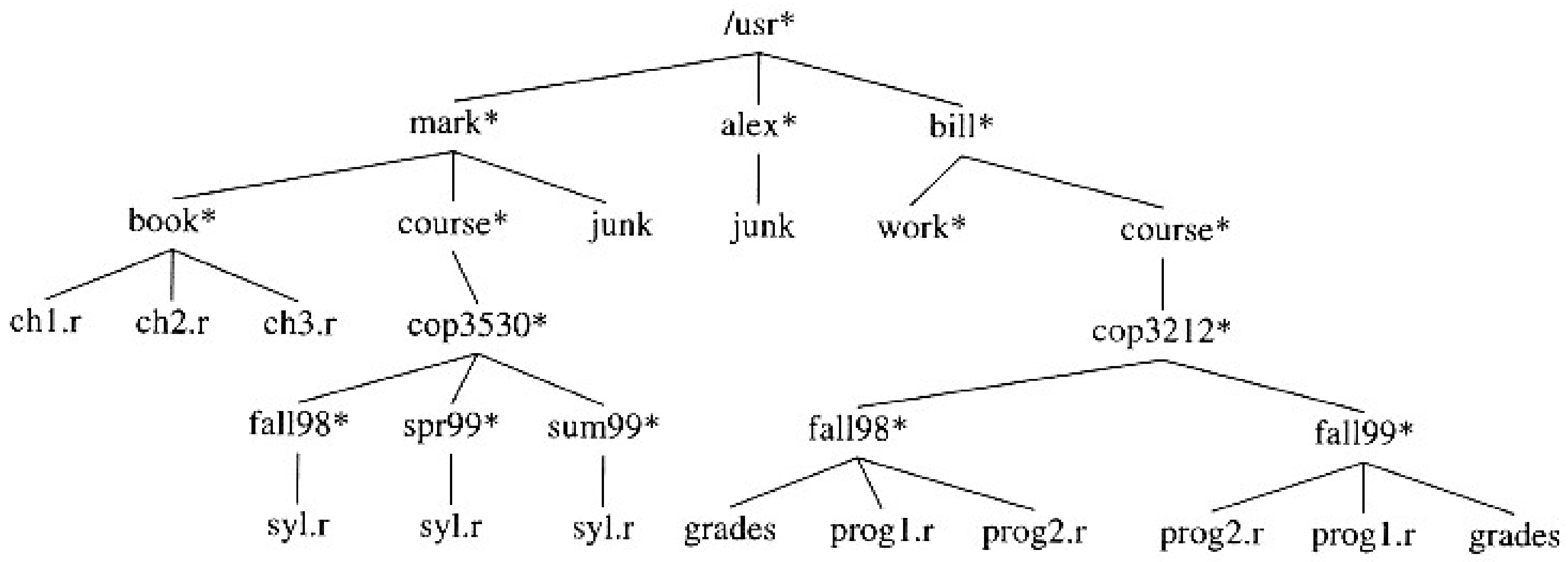
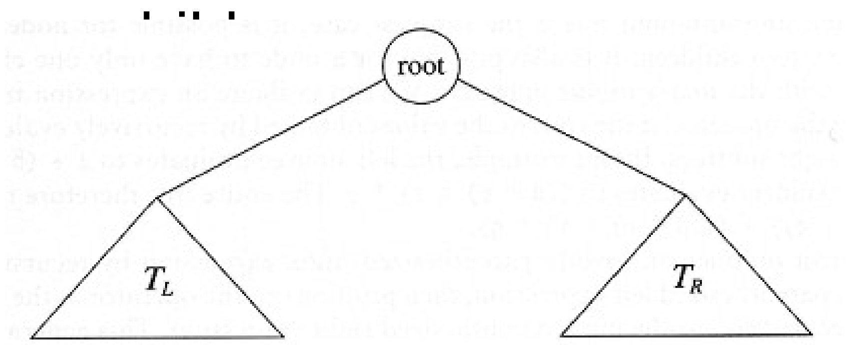
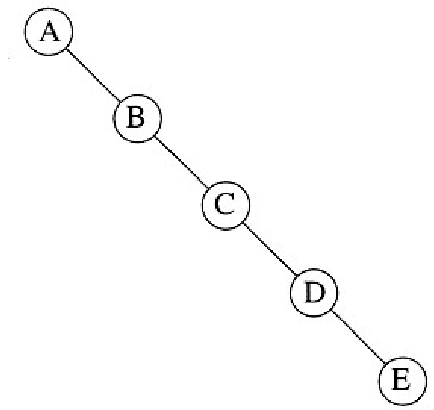


Figure 4.5 UNIX directory

Binary Trees

- A tree in which no node can have more than two





- The depth of an “average” binary tree is considerably smaller than N, eventhough in the worst case, the depth can be as large as N – 1.

Example: Expression Trees

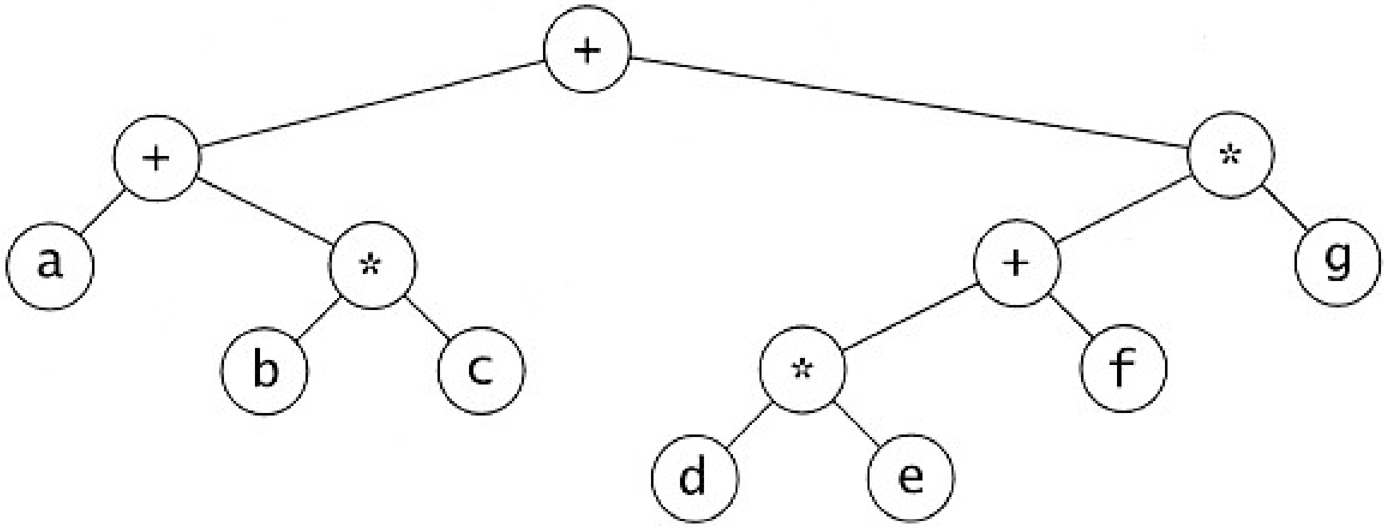


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- Leaves are operands (constants or variables)
- The other nodes (internal nodes) contain operators
- Will not be a binary tree if some operators are not binary

Tree traversal

- A tree traversal is a method of visiting every node in the tree.
  - Ex: Used to print out the data in a tree in a certain order
- There are three common ways to traverse a binary tree.
  - Pre-order traversal

Preorder traversal

VIVEN Embedded Academy

Pre-order traversal

- Each root node is visited before its left and right subtrees are traversed. Pre-order search is also called backtracking.
- The steps for traversing a binary tree in pre-order traversal are
  - Visit the root

Preorder

- Preorder traversal
  - node, left, right
  - prefix expression
    - ++a*bc*++*defg

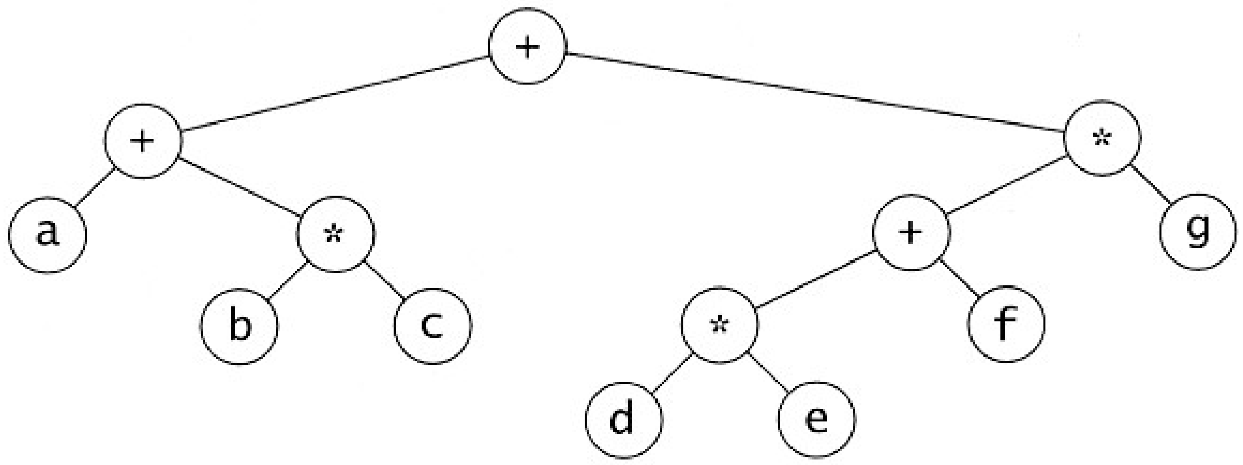


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Postorder traversal

Post-order traversal

- Each root node is visited after its left and right subtrees have been traversed.
- The steps for traversing a binary tree in post-order traversal are
  - Visit the left subtree, using postorder
  - Visit the right subtree, using postorder.
  - Visit the root

The algorith for postorder traversal is as follows.

```
Void postorder(node *root)

{

if(root!=NULL)

{

postorder(root->lchild);

postorder(root->rchild);

print root->data;

}

}
```

Inorder traversal

In-order traversal

- The root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins.
- The steps for traversing a binary tree in post-order traversal are
  - Visit the left subtree, using postorder
  - Visit the right subtree, using postorder.
  - Visit the root
- The algorith for inorder traversal is as follows.

```
    Void inorder(node *root)
    {
```

Preorder, Postorder and Inorder

- Postorder traversal
  - left, right, node
  - postfix expression
    - abc*+de*f+g*+
- Inorder traversal
  - left, node, right.
  - infix expression
    - a+b*c+d*e+f*g

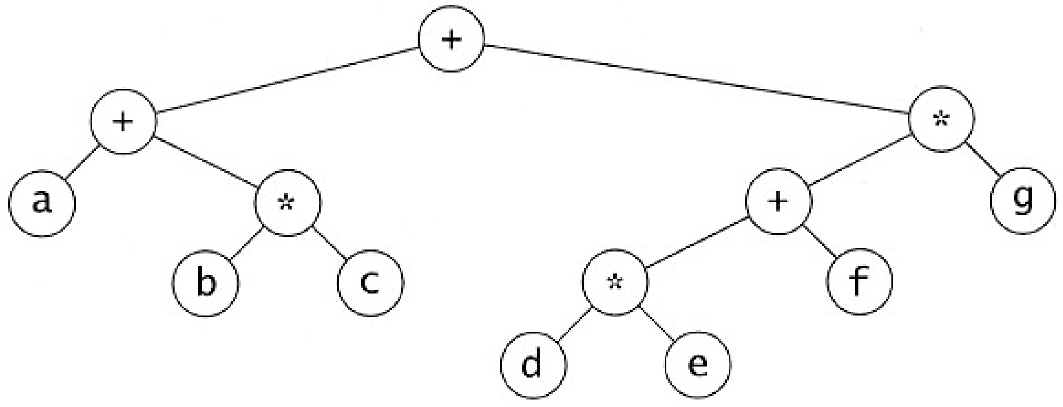


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

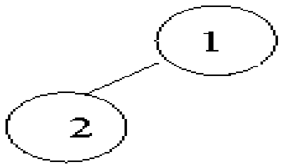
Preorder, Postorder and Inorder

```
Algorithm Preorder($x$)
Input: $x$ is the root of a subtree.
1.   if $x \neq$ NULL
2.       then output key($x$);
3.            Preorder(left($x$));
4.            Preorder(right($x$));
```

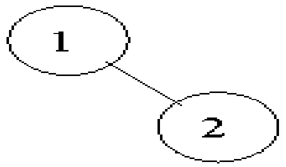
```
Algorithm Postorder($x$)
Input: $x$ is the root of a subtree.
1.   if $x \neq$ NULL
2.       then Postorder(left($x$));
3.            Postorder(right($x$));
4.            output key($x$);
```

```
Algorithm Inorder($x$)
Input: $x$ is the root of a subtree.
1.   if $x \neq$ NULL
2.       then Inorder(left($x$));
3.            output key($x$);
4.            Inorder(right($x$));
```

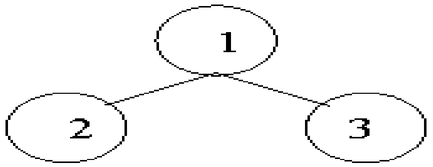
Binary Trees



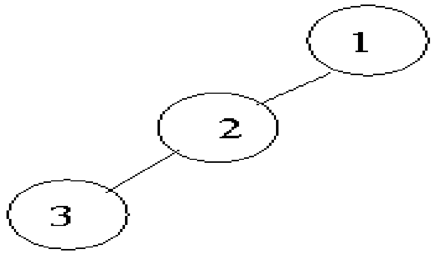
Pre: 12
Post: 21
In: 21



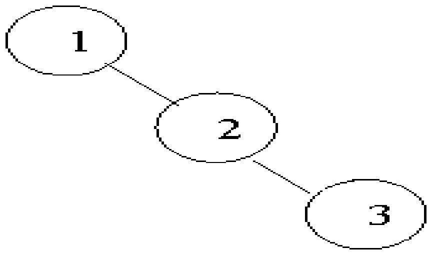
Pre: 12
Post: 21
In: 12



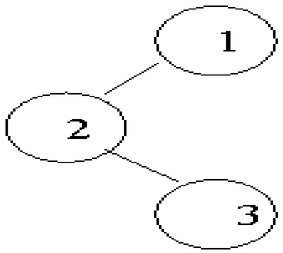
Pre: 123
Post: 231
In: 213



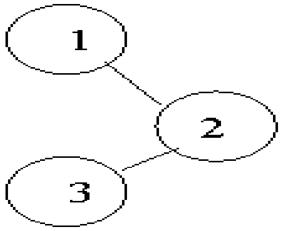
Pre: 123
Post: 321
In: 321



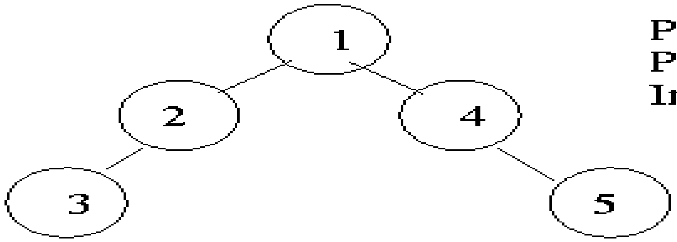
Pre: 123
Post: 321
In: 123



Pre: 123
Post: 321
In: 231



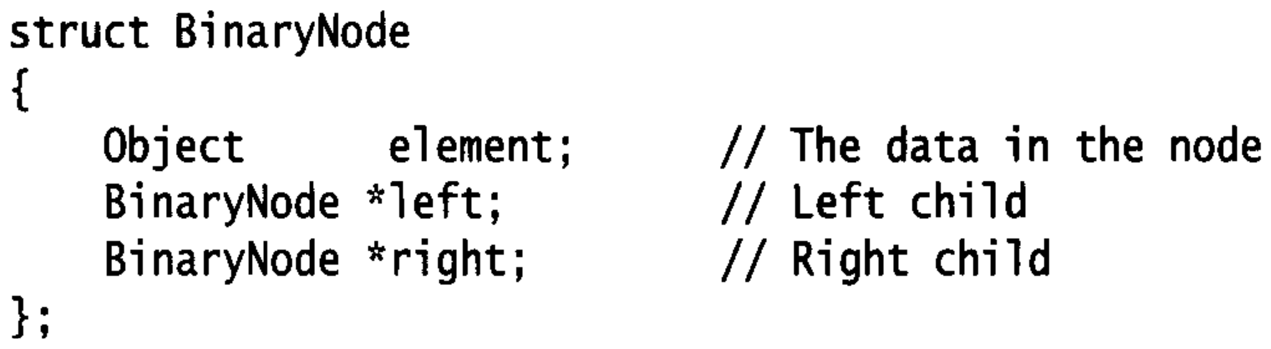
Pre: 123
Post: 321
In: 132



Pre: 12345
Post: 32541
In: 32145

Binary Trees

- Possible operations on the Binary Tree ADT
  - parent
  - left_child, right_child
  - sibling
  - root, etc
- Implementation
  - Because a binary tree has at most two children, we can keep direct pointers to them



```
struct BinaryNode
{
    Object      element;       // The data in the node
    BinaryNode *left;          // Left child
    BinaryNode *right;         // Right child
};
```

Implementation of a general tree

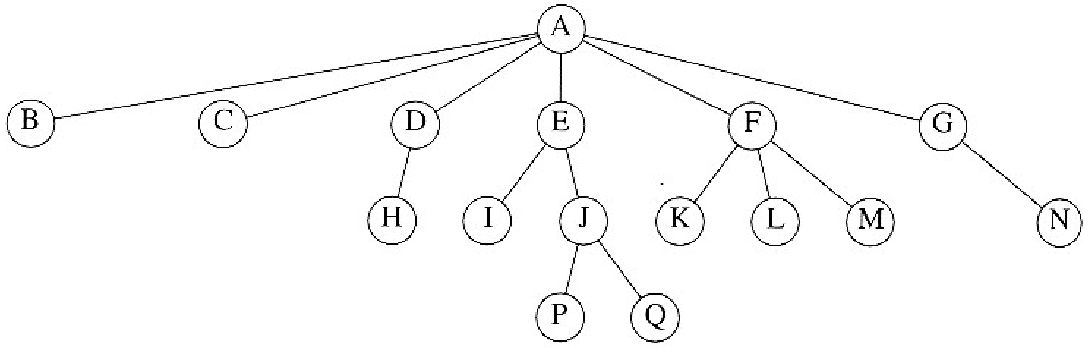


Figure 4.2 A tree

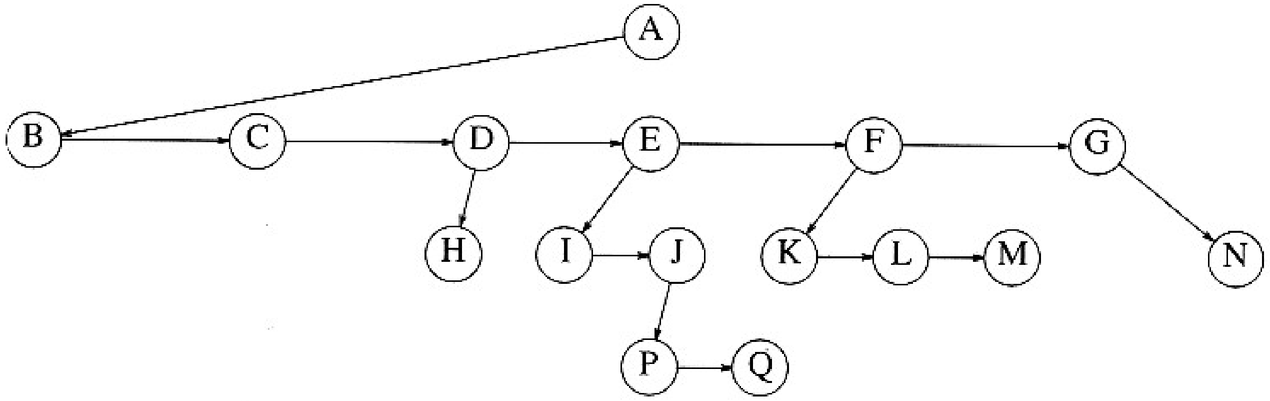


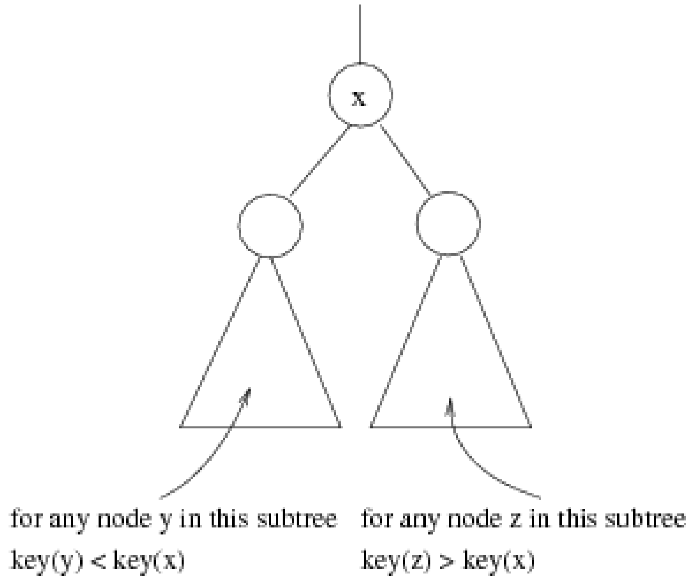
Figure 4.4 First child/next sibling representation of the tree shown in Figure 4.2

Binary Search Trees

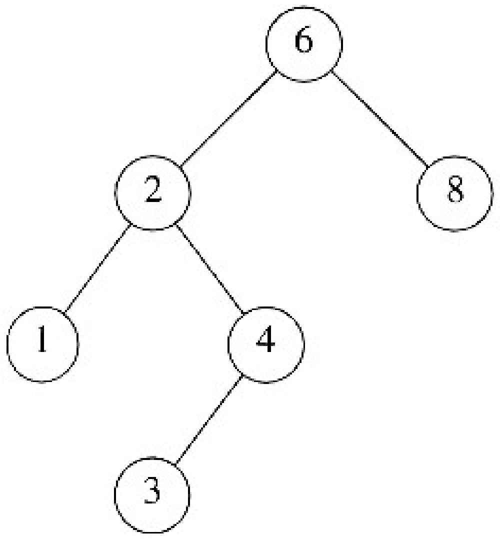
- Stores keys in the nodes in a way so that searching, insertion and deletion can be done efficiently.

Binary search tree property

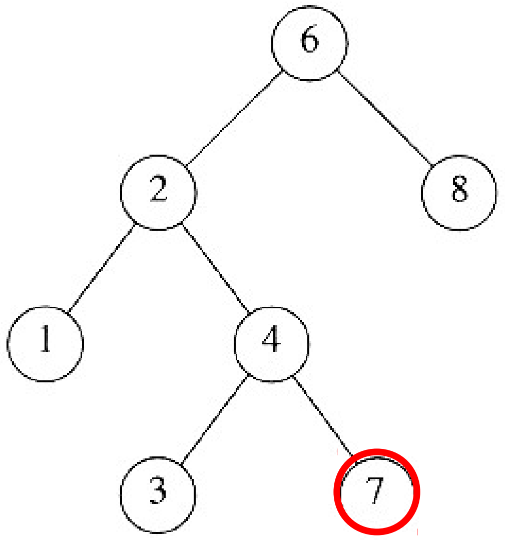
- For every node X, all the keys in its left subtree are smaller than the key value in X, and all the keys in its right subtree are larger than the key value in X.


x
for any node y in this subtree
key(y) < key(x)
for any node z in this subtree
key(z) > key(x)

Binary Search Trees



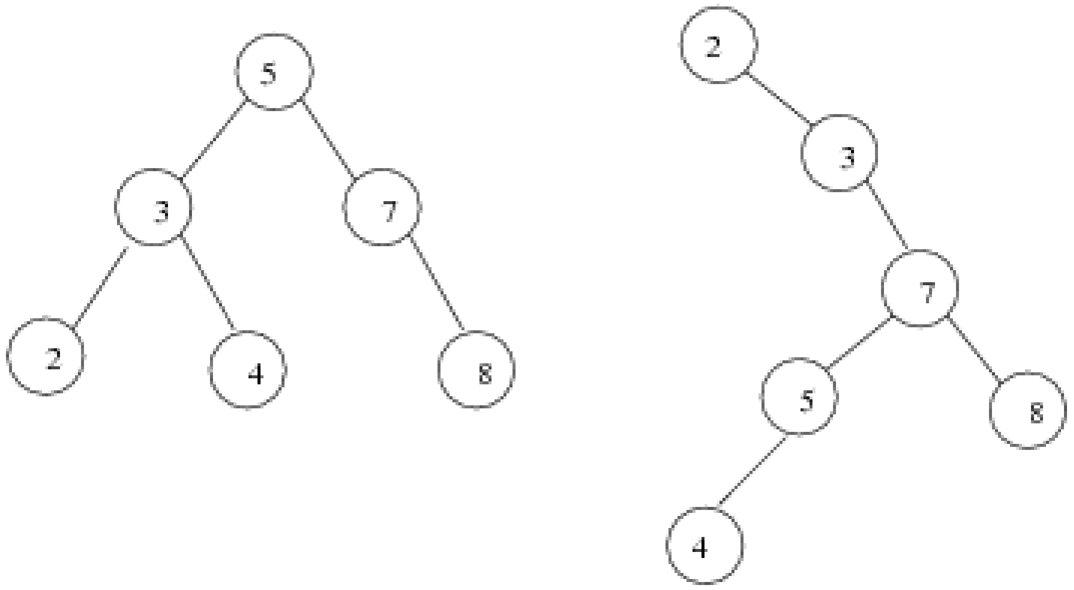
A binary search tree



Not a binary search tree

Binary search trees

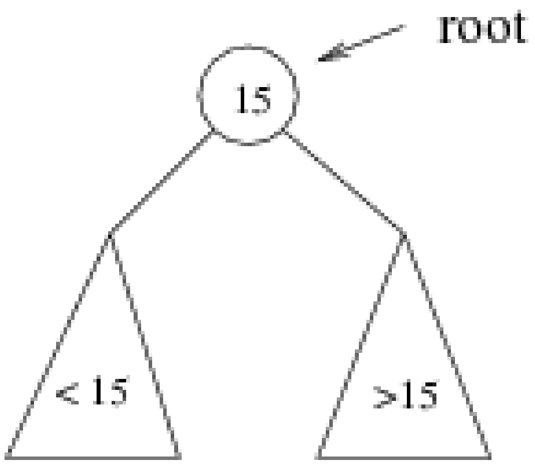
Two binary search trees representing the same set:



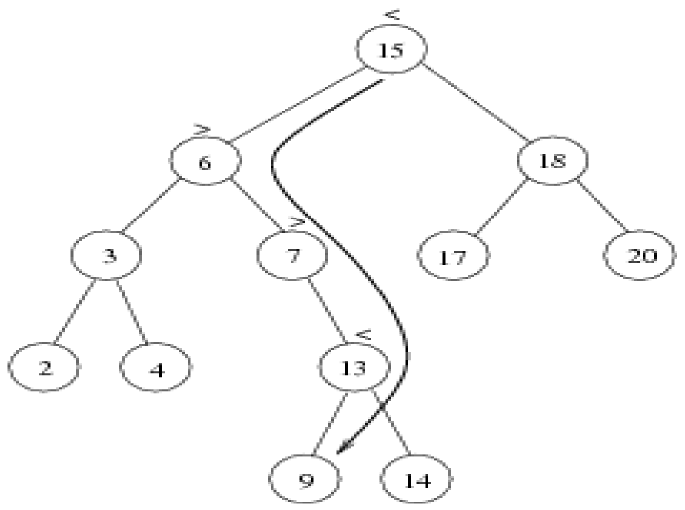
- Average depth of a node is O(log N); maximum depth of a node is O(N)

Searching BST

- If we are searching for 15, then we are done.
- If we are searching for a key < 15, then we should search in the left subtree.
- If we are searching for a key > 15, then we should search in the right subtree.



Example: Search for 9 ...

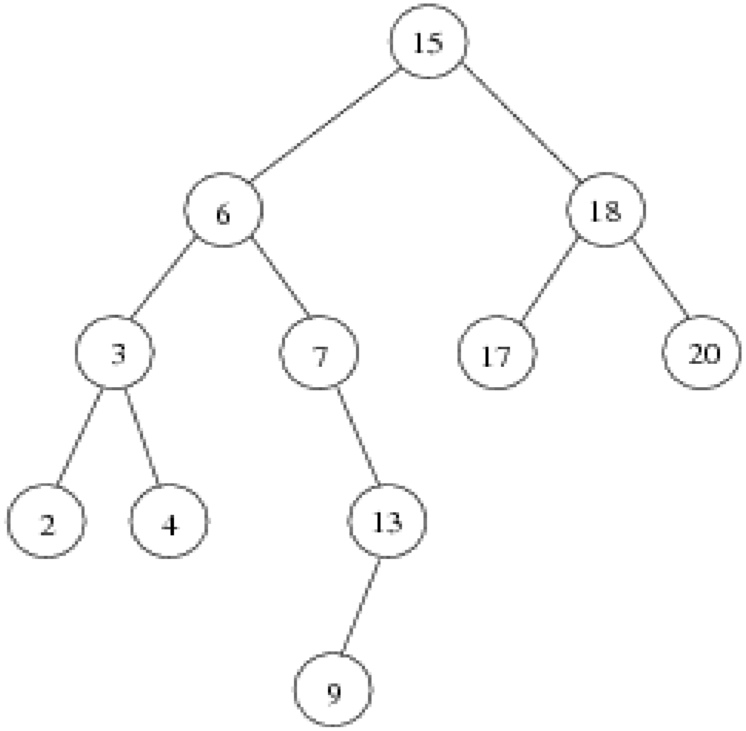


Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

Inorder traversal of BST

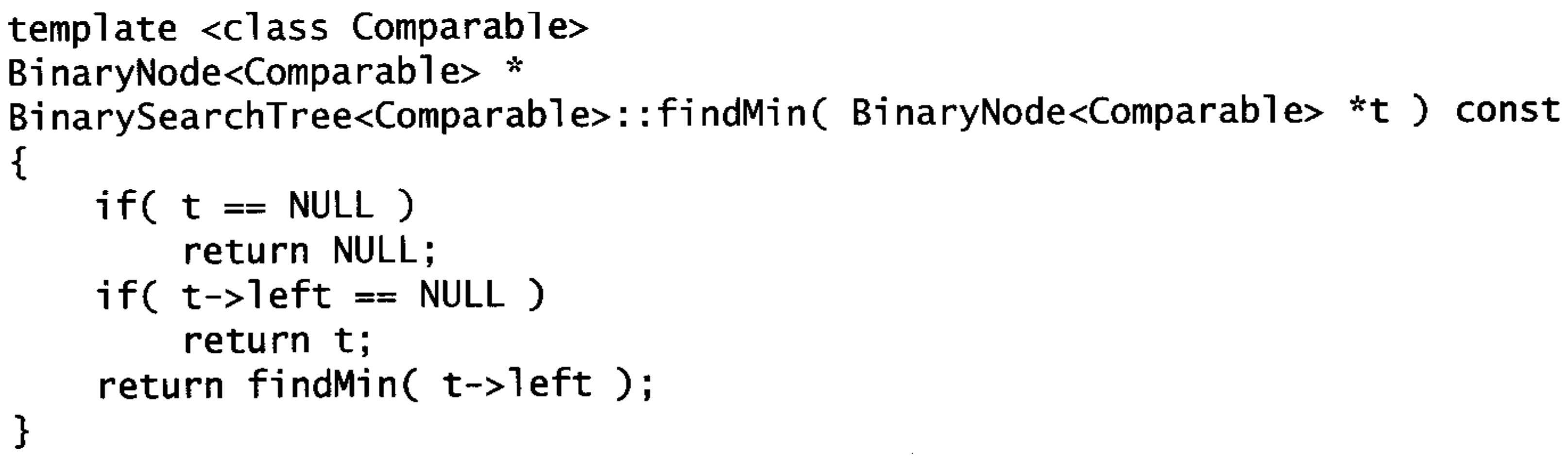
- Print out all the keys in sorted order



Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

findMin/ findMax

- Return the node containing the smallest element in the tree
- Start at the root and go left as long as there is a left child. The stopping point is the smallest element

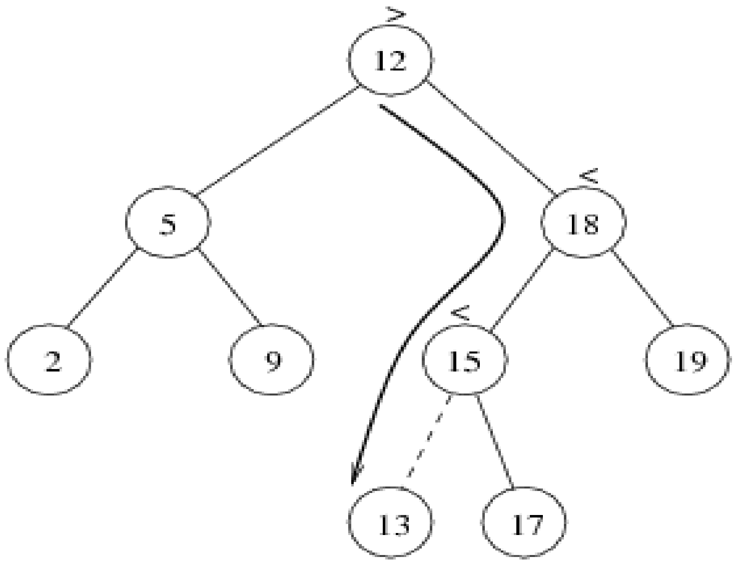


```
template <class Comparable>
BinaryNode<Comparable> *
BinarySearchTree<Comparable>::findMin( BinaryNode<Comparable> *t ) const
{
    if( t == NULL )
        return NULL;
    if( t->left == NULL )
        return t;
    return findMin( t->left );
}
```

- Similarly for findMax
- Time complexity = O(height of the tree)

insert

- Proceed down the tree as you would with a find
- If X is found, do nothing (or update something)
- Otherwise, insert X at the last spot on the path traversed



delete

- When we delete a node, we need to consider how we take care of the children of the deleted node.
  - This has to be done such that the property of the search tree is maintained.

delete

Three cases:

(1) the node is a leaf

- Delete it immediately

(2) the node has one child

- Adjust a pointer from the parent to bypass that node

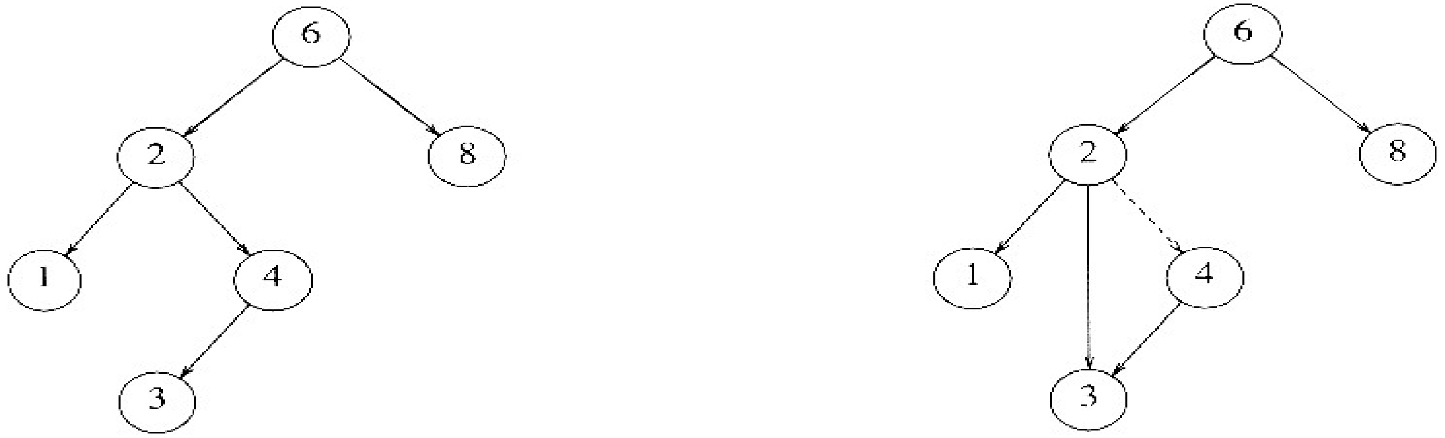


Figure 4.24 Deletion of a node (4) with one child, before and after

delete

(3) the node has 2 children

- replace the key of that node with the minimum element at the right subtree
- delete the minimum element
    - Has either no child or only right child because if it has a left child, that left child would be smaller and would have been chosen. So invoke case 1 or 2.

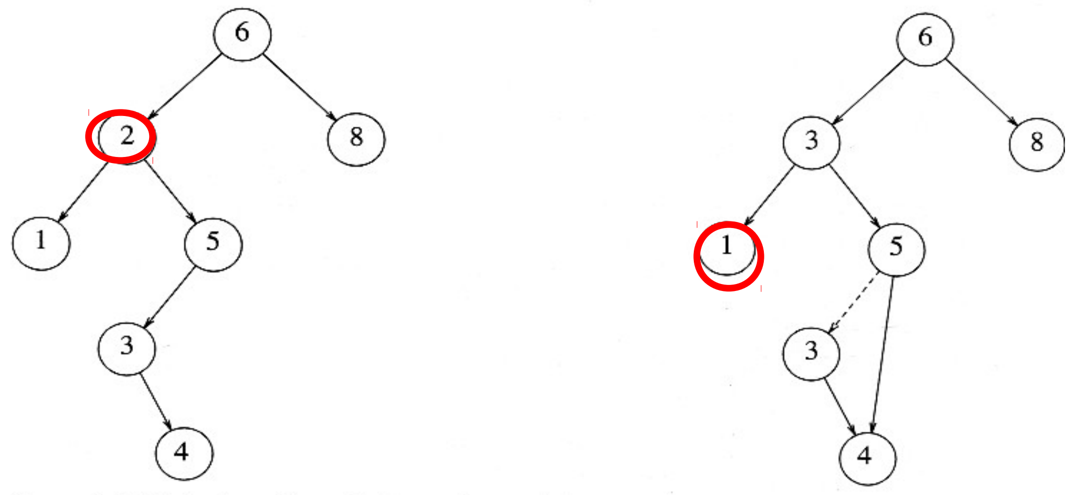


Figure 4.25 Deletion of a node (2) with two children, before and after

- Time complexity = O(height of the tree)

VIVEN Embedded Academy

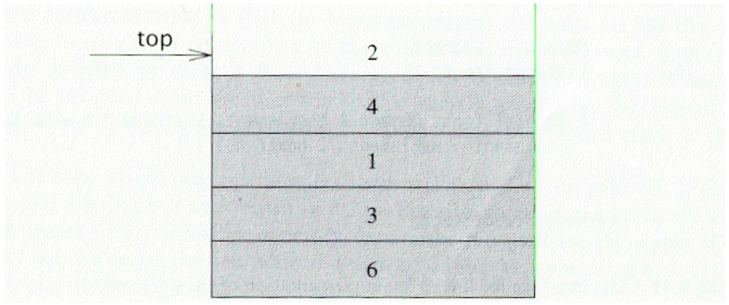
Stacks

Stack Overview

- Stack ADT
- Basic operations of stack
  - Pushing, popping etc.
- Implementations of stacks using
  - array
  - linked list

The Stack ADT

- A stack is a list with the restriction
  - that insertions and deletions can only be performed at the *top* of the list



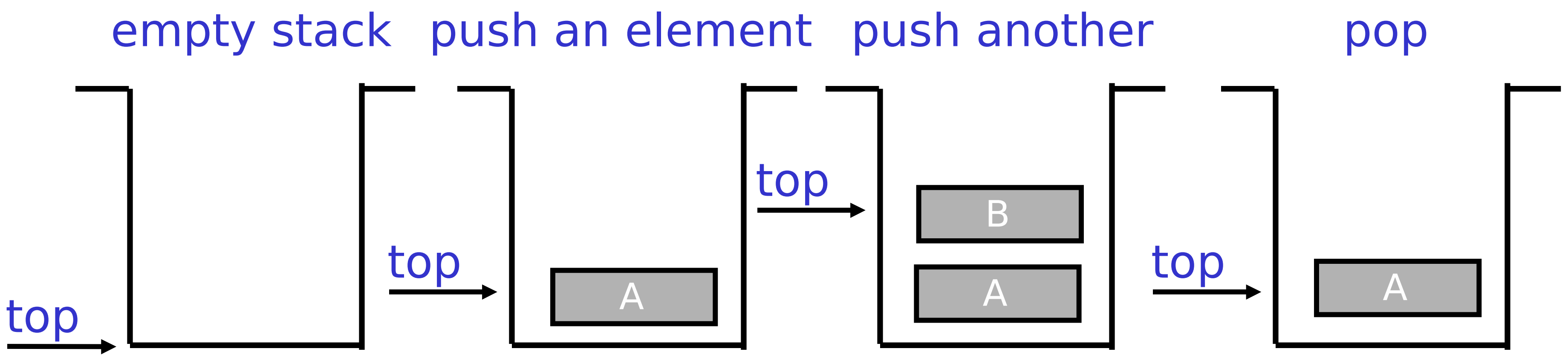
  - The other end is called bottom
- Fundamental operations:
  - Push: Equivalent to an insert
  - Pop: Deletes the most recently inserted element
  - Top: Examines the most recently inserted element

Stack ADT

- Stacks are less flexible but are more efficient and easy to implement
- Stacks are known as LIFO (Last In, First Out) lists.
    - The last element inserted will be the first to be retrieved

Push and Pop

- Primary operations: Push and Pop
- Push
  - Add an element to the top of the stack
- Pop
  - Remove the element at the top of the stack



Implementation of Stacks

- Any list implementation could be used to implement a stack
  - Arrays (static: the size of stack is given initially)
  - Linked lists (dynamic: never become full)
- We will explore implementations based on array and linked list
- Let’s see how to use an array to implement a stack first

Array Implementation

- Need to declare an array size ahead of time
- Associated with each stack is TopOfStack
    - for an empty stack, set TopOfStack to -1
- Push
    - (1) Increment TopOfStack by 1.
    - (2) Set Stack[TopOfStack] = X
- Pop
    - (1) Set return value to Stack[TopOfStack]
    - (2) Decrement TopOfStack by 1
- These operations are performed in very fast constant time

Stack class

- Attributes of `Stack`
  - **`maxTop`**: the max size of stack
  - **`top`**: the index of the top element of stack
  - **`values`**: point to an array which stores elements of stack
- Operations of `Stack`
  - **`IsEmpty`**: return true if stack is empty, return false otherwise
  - **`IsFull`**: return true if stack is full, return false otherwise
  - **`Top`**: return the element at the top of stack
  - **`Push`**: add an element to the top of stack
  - **`Pop`**: delete the element at the top of stack
  - **`DisplayStack`**: print all the data in the stack

Create Stack

- The constructor of `Stack`
  - Allocate a stack array of `size`. By default, `size = 10`.
  - When the stack is full, `top` will have its maximum value, i.e. `size – 1`.
  - Initially `top` is set to -1. It means the stack is empty.

```
Stack(int size /*= 10*/) {
   maxTop   =  size - 1;
   values   =  new double[size];
   top      =  -1;
}
```

Although the constructor dynamically allocates the stack array, the stack is still static. The size is fixed after the initialization.

Push Stack

- `void Push(const double x);`
  - Push an element onto the stack
  - If the stack is full, print the error information.
  - Note `top` always represents the index of the top element. After pushing an element, increment `top`.

Pop Stack

- `double Pop()`
  - Pop and return the element at the top of the stack
  - If the stack is empty, print the error information. (In this case, the return value is useless.)
  - Don’t forgot to decrement **`top`**

Stack vs Queue

Stacks	Queues
Last in first out(LIFO)	First in First Out(FIFO)

Stack Top

- `double Top()`
  - Return the top element of the stack
  - Unlike Pop, this function does not remove the top element

Balancing Symbols

VIVEN
Embedded Academy

- To check that every right brace, bracket, and parentheses must correspond to its left counterpart
  - e.g. [( )] is legal, but [( ]) is illegal
- Algorithm

(1) Make an empty stack.

(2) Read characters until end of file

i. If the character is an opening symbol, push it onto the stack

ii. If it is a closing symbol, then if the stack is empty, report an error

iii. Otherwise, pop the stack. If the symbol popped is not the corresponding opening symbol, then report an error

(3) At end of file, if the stack is not empty, report an error

Postfix Expressions

- Calculate 4.99 * 1.06 + 5.99 + 6.99 * 1.06
  - Need to know the precedence rules
- Postfix (reverse Polish) expression
  - 4.99 1.06 * 5.99 + 6.99 1.06 * +
- Use stack to evaluate postfix expressions
  - When a number is seen, it is pushed onto the stack
  - When an operator is seen, the operator is applied to the 2 numbers that are popped from the stack. The result is pushed onto the stack
- Example
  - evaluate 6 5 2 3 + 8 * + 3 + *
- The time to evaluate a postfix expression is O(N)
  - processing each element in the input consists of stack operations and thus takes constant time

VIVEN
Embedded Academy

QUEUES

Queue Overview

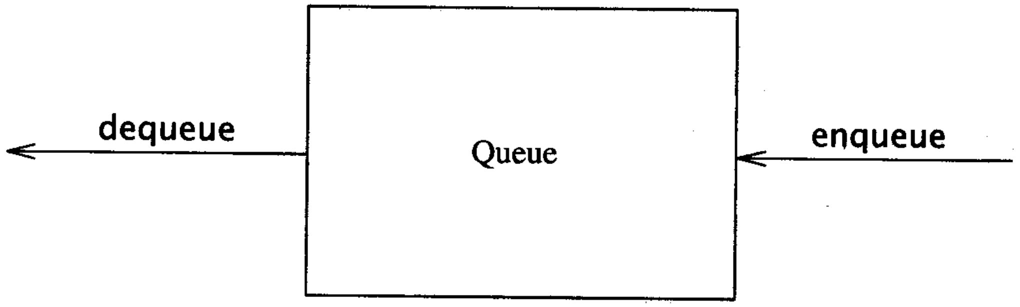
- Queue ADT
- Basic operations of queue
  - Enqueuing, dequeuing etc.
- Implementation of queue
  - Array
  - Linked list

Queue ADT

- Like a stack, a *queue* is also a list. However, with a queue, insertion is done at one end, while deletion is performed at the other end.
- Accessing the elements of queues follows a First In, First Out (FIFO) order.
  - Like customers standing in a check-out line in a store, the first customer in is the first customer served.

The Queue ADT

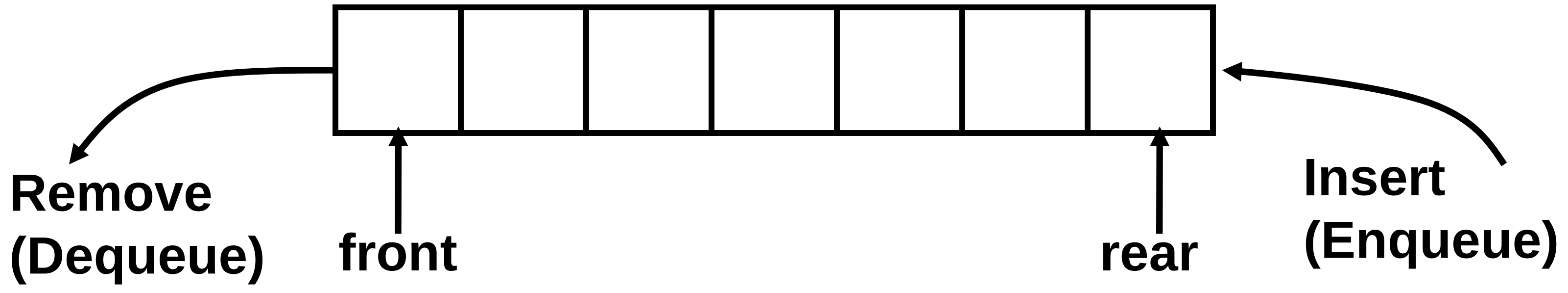
- Another form of restricted list
  - Insertion is done at one end, whereas deletion is performed at the other end
- Basic operations:
  - enqueue: insert an element at the rear of the list
  - dequeue: delete the element at the front of the list



- First-in First-out (FIFO) list

Enqueue and Dequeue

- Primary queue operations: Enqueue and Dequeue
- Like check-out lines in a store, a queue has a front and a rear.
- Enqueue
  - Insert an element at the rear of the queue
- Dequeue
  - Remove an element from the front of the queue

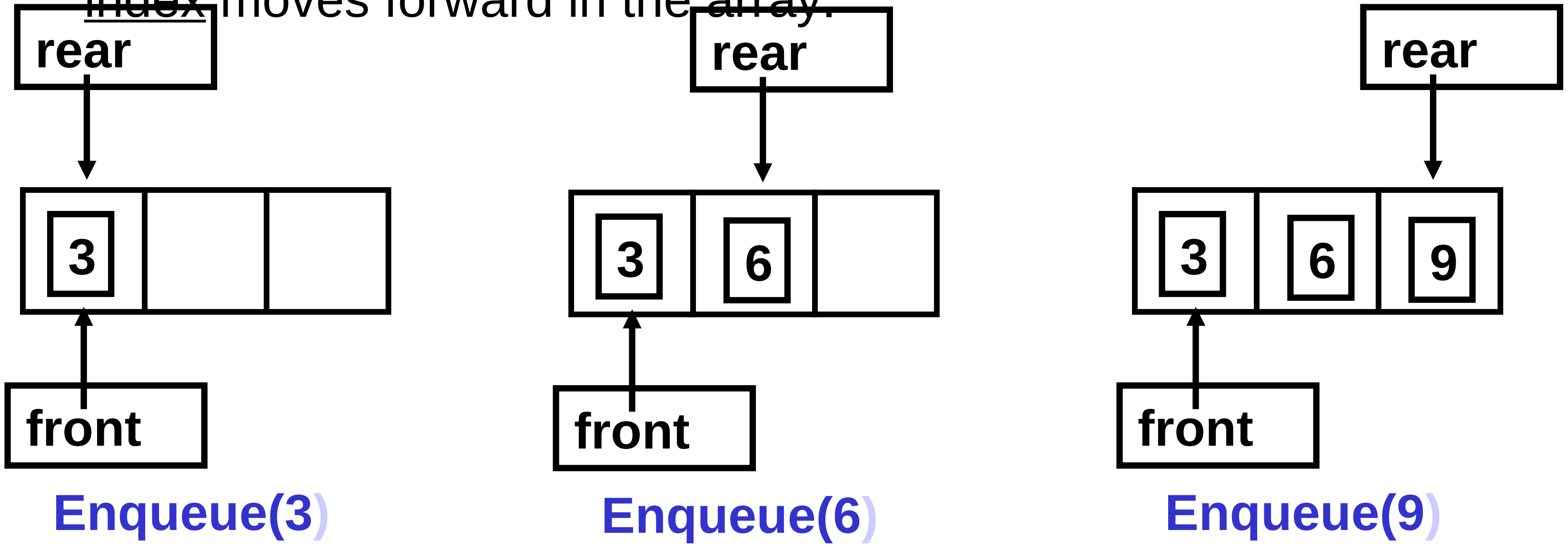


Implementation of Queue

- Just as stacks can be implemented as arrays or linked lists, so with queues.
- Dynamic queues have the same advantages over static queues as dynamic stacks have over static stacks

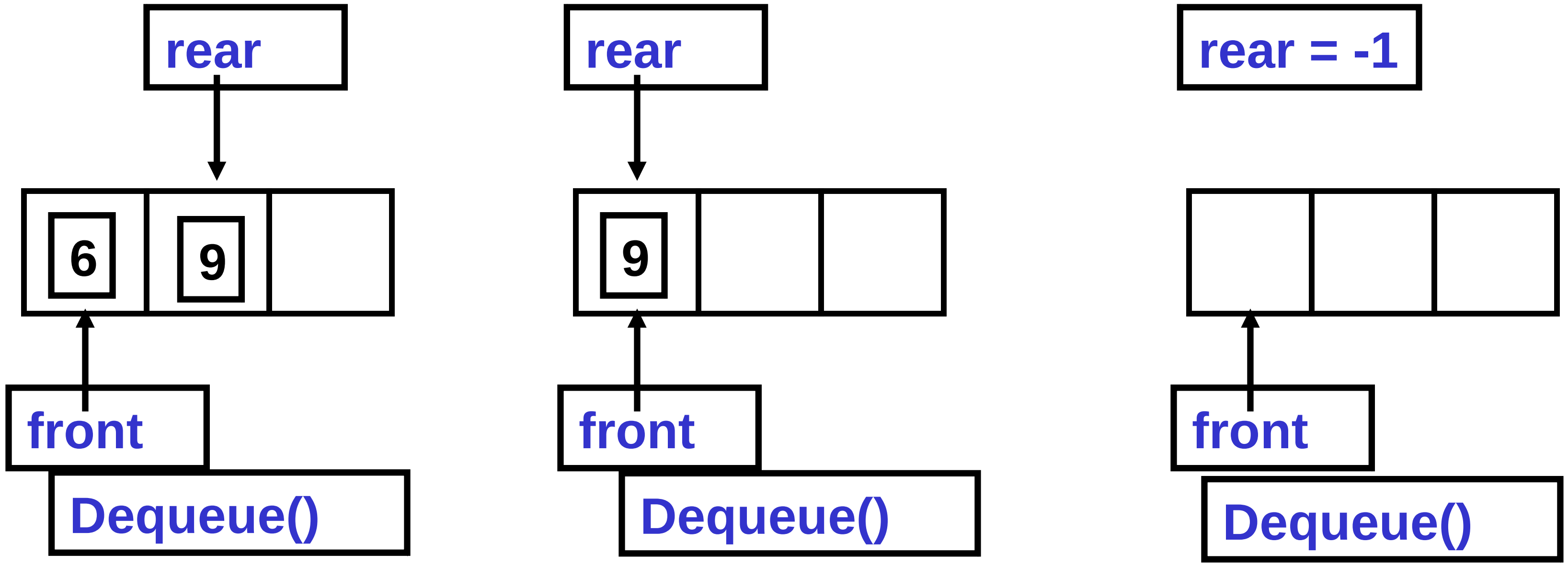
Queue Implementation of Array

- There are several different algorithms to implement Enqueue and Dequeue
- Naïve way
  - When enqueuing, the front index is always fixed and the rear index moves forward in the array.



Queue Implementation of Array

- Naïve way
  - When enqueuing, the front index is always fixed and the rear index moves forward in the array.
  - When dequeuing, the element at the front the queue is removed. Move all the elements after it by one position. (Inefficient!!!)



Queue Implementation of Array

- Better way
  - When an item is enqueued, make the <u>rear index</u> move forward.
  - When an item is dequeued, the <u>front index</u> moves by one element towards the back of the queue (thus removing the front item, so no copying to neighboring elements is needed).

```
                ────────→
(front) XXXXOOOOO   (rear)
        OXXXXOOOO   (after 1 dequeue, and 1 enqueue)
        OOXXXXXOO   (after another dequeue, and 2 enqueues)
        OOOOXXXXX   (after 2 more dequeues, and 2 enqueues)
```

The problem here is that the rear index cannot move beyond the last element in the array.

Implementation using Circular Array

- Using a circular array
- When an element moves past the end of a circular array, it wraps around to the beginning, e.g.
    - OOOOO7963 → 4OOOO7963 (after Enqueue(4))
    - After Enqueue(4), the <u>rear index</u> moves from 3 to 4.

Empty or Full?

- Empty queue
  - back = front - 1
- Full queue?
  - the same!
  - Reason: n values to represent n+1 states
- Solutions
  - Use a boolean variable to say explicitly whether the queue is empty or not
  - Make the array of size n+1 and only allow n elements to be stored
  - Use a counter of the number of elements in the queue

Queue Class

- Attributes of `Queue`
  - `front/rear`: front/rear index
  - `counter`: number of elements in the queue
  - `maxSize`: capacity of the queue
  - `values`: point to an array which stores elements of the queue
- Operations of `Queue`
  - return true if queue is empty, return false otherwise
  - return true if queue is full, return false otherwise
  - add an element to the rear of queue
  -

VIVEN
Embedded Academy

Search Algorithms

Introduction to Search Algorithms

- Search: locate an item in a list of information
- Two algorithms we will examine:
  - Linear search
  - Binary search

Linear Search

- Also called the sequential search
- Starting at the first element, this algorithm sequentially steps through an array examining each element until it locates the value it is searching for.

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`

```
17  23  5  11  2  29  3
```

Linear Search

- Algorithm:

```
set found to false; set position to –1; set index to 0
while index < number of elts. and found is false
    if list[index] is equal to search value
        found = true
        position = index
    end if
    add 1 to index
end while
return position
```

A Linear Search Function

```
int searchList(int list[], int numElems, int value)
{
   int index = 0;       // Used as a subscript to search array
   int position = -1;  // To record position of search value
   bool found = false; // Flag to indicate if value was found

   while (index < numElems && !found)
   {
      if (list[index] == value) // If the value is found
      {
         found = true; // Set the flag
         position = index; // Record the value's subscript
      }
      index++; // Go to the next element
   }
return position; // Return the position, or -1
}
```

Linear Search - Tradeoffs

VIVEN Embedded Academy

- Benefits:
  - Easy algorithm to understand
  - Array can be in any order
- Disadvantages:
  - Inefficient (slow): for array of N elements,

Binary Search

Requires array elements to be in order

- 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
- 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.
- Continue steps 1. and 2. until either the value is found or there are no more elements to examine

Binary Search - Example

- Array `numlist2` contains:

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

```
2   3   5   11   17   23   29
```

Binary Search

```
Set first index to 0.
Set last index to the last subscript in the array.
Set found to false.
Set position to -1.
While found is not true and first is less than or equal to last
    Set middle to the subscript half-way between array[first] and array[last].
    If array[middle] equals the desired value
        Set found to true.
        Set position to middle.
    Else If array[middle] is greater than the desired value
        Set last to middle - 1.
    Else
        Set first to middle + 1.
    End If.
End While.
Return position.
```

A Binary Search Function

```
int binarySearch(int array[], int numElems, int value)
{
   int first = 0,             // First array element
       last = numElems - 1, // Last array element
       middle,              // Mid point of search
       position = -1;       // Position of search value
   bool found = false;      // Flag

   while (!found && first <= last)
   {
      middle = (first + last) / 2; // Calculate mid point
      if (array[middle] == value)  // If value is found at mid
      {
         found = true;
         position = middle;
      }
      else if (array[middle] > value) // If value is in lower half
         last = middle - 1;
      else
         first = middle + 1; // If value is in upper half
   }
return position;
}
```

Binary Search - Tradeoffs

- Benefits:
  - Much more efficient than linear search. For array of N elements, performs at most $\log_2 N$ comparisons
- Disadvantages:
  - Requires that array elements be sorted

Sorting Algorithms

Introduction to Sorting Algorithms

- Sort: arrange values into an order:
  - Alphabetical
  - Ascending numeric
  - Descending numeric

- THREE algorithms considered here:
  - Bubble sort
  - Selection sort
  - Quick sort

VIVEN Embedded Academy

Bubble Sort

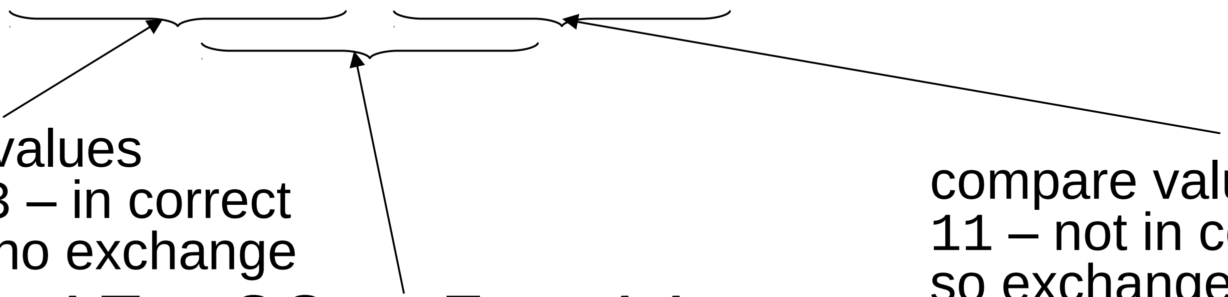
VIVEN Embedded Academy

Concept:

- Compare 1st two elements
  - If out of order, exchange them to put in order
- Move down one element, compare 2nd and 3rd elements, exchange if necessary. Continue until end of array.
- Pass through array again, exchanging as necessary
- Repeat until pass made with no exchanges

Example – First Pass

Array `numlist3` contains:



```
17   23   5   11
```

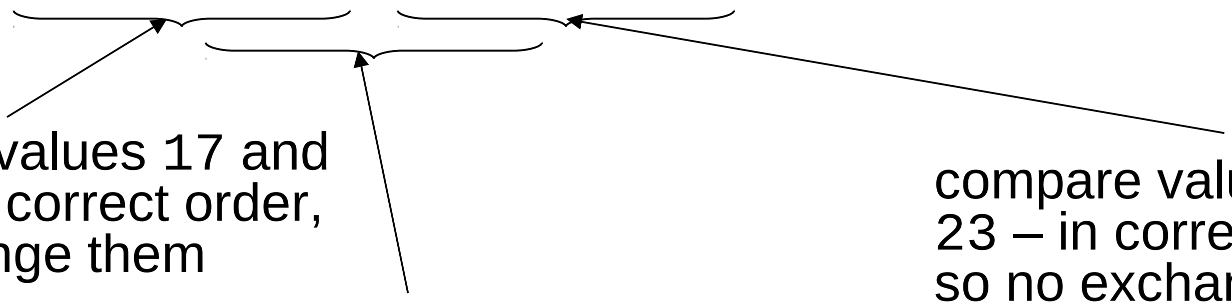
compare values
17 and 23 – in correct
order, so no exchange

compare values 23 and
5 – not in correct order,
so exchange them

compare values 23 and
11 – not in correct order,
so exchange them

Example – Second Pass

After first pass, array `numlist3` contains:



compare values 17 and 5 – not in correct order, so exchange them

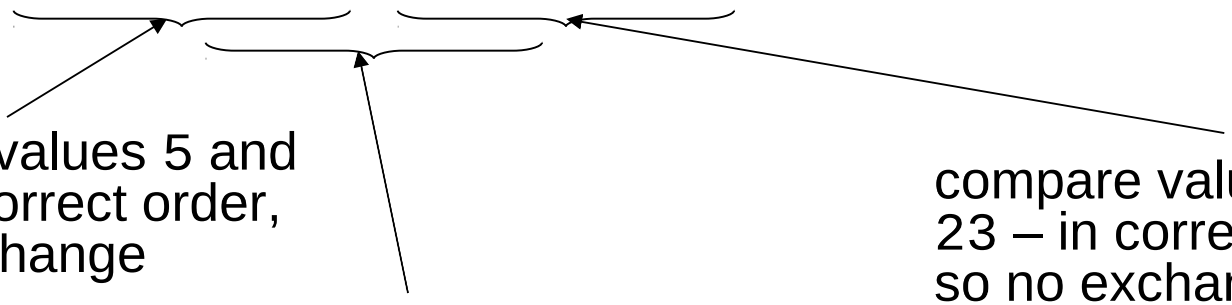
compare values 17 and 11 – not in correct order, so exchange them

compare values 17 and 23 – in correct order, so no exchange

```
17   5   11   23
```

Example – Third Pass

After second pass, array `numlist3` contains:



compare values 5 and 11 – in correct order, so no exchange

compare values 11 and 17 – in correct order, so no exchange

compare values 17 and 23 – in correct order, so no exchange

No exchanges, so array is in order

```
5   11   17   23
```

Bubble Sort - Tradeoffs

- Benefit:
  - Easy to understand and implement

- Disadvantage:
  - Inefficient: slow for large arrays

Selection Sort

- Concept for sort in ascending order:
  - Locate smallest element in array. Exchange it with element in position 0
  - Locate next smallest element in array. Exchange it with element in position 1.
  - Continue until all elements are arranged in order

Selection Sort - Example

Array `numlist` contains:

1. Smallest element is 2. Exchange 2 with element in 1st position in array:

```
11   2   29   3
```

Example (Continued)

- Next smallest element is 3. Exchange 3 with element in 2nd position in array:

3. Next smallest element is 11. Exchange 11 with element in 3rd position in array:

```
2    3    29   11
```

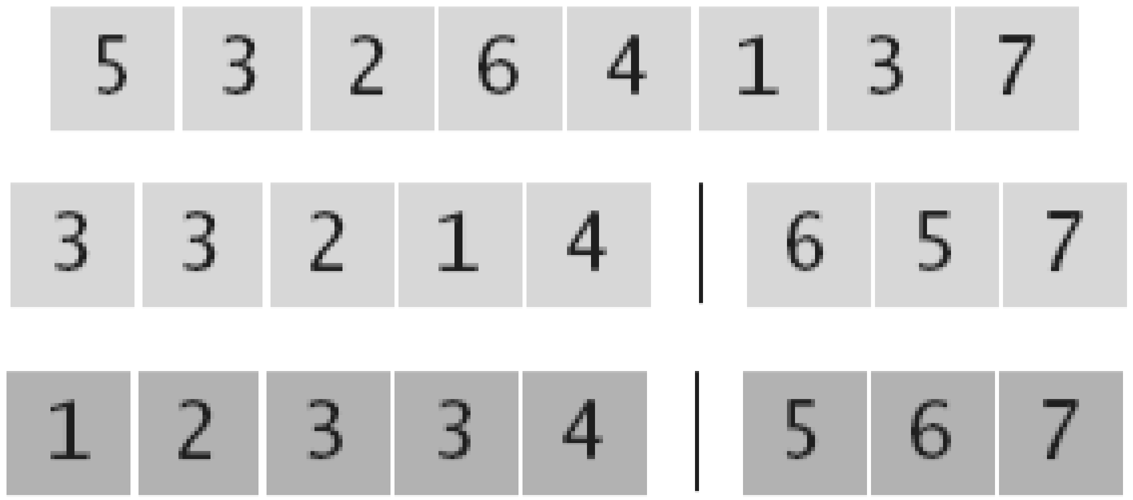
Selection Sort - Tradeoffs

- Benefit:
  - More efficient than Bubble Sort, since fewer exchanges
- Disadvantage:
  - May not be as easy as Bubble Sort to understand

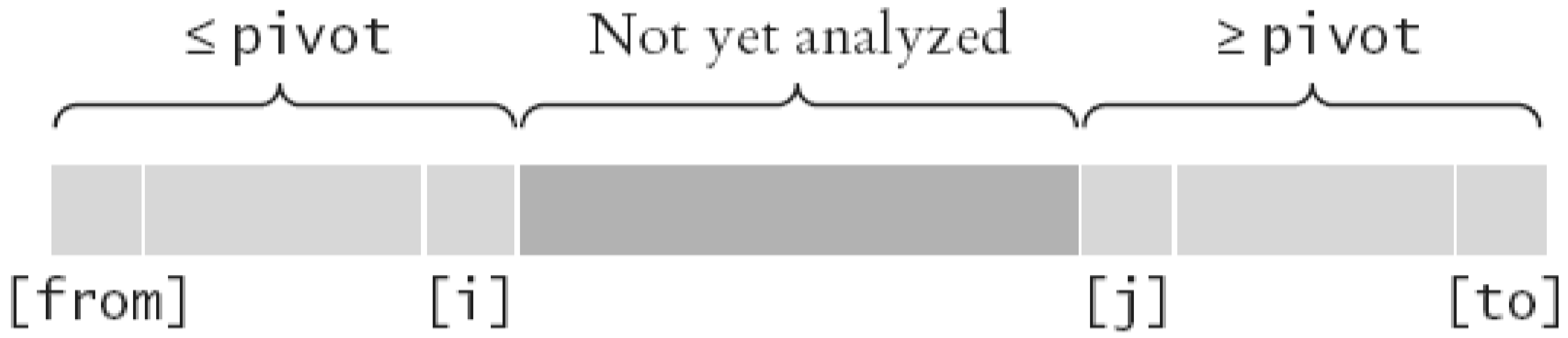
QUICK SORT

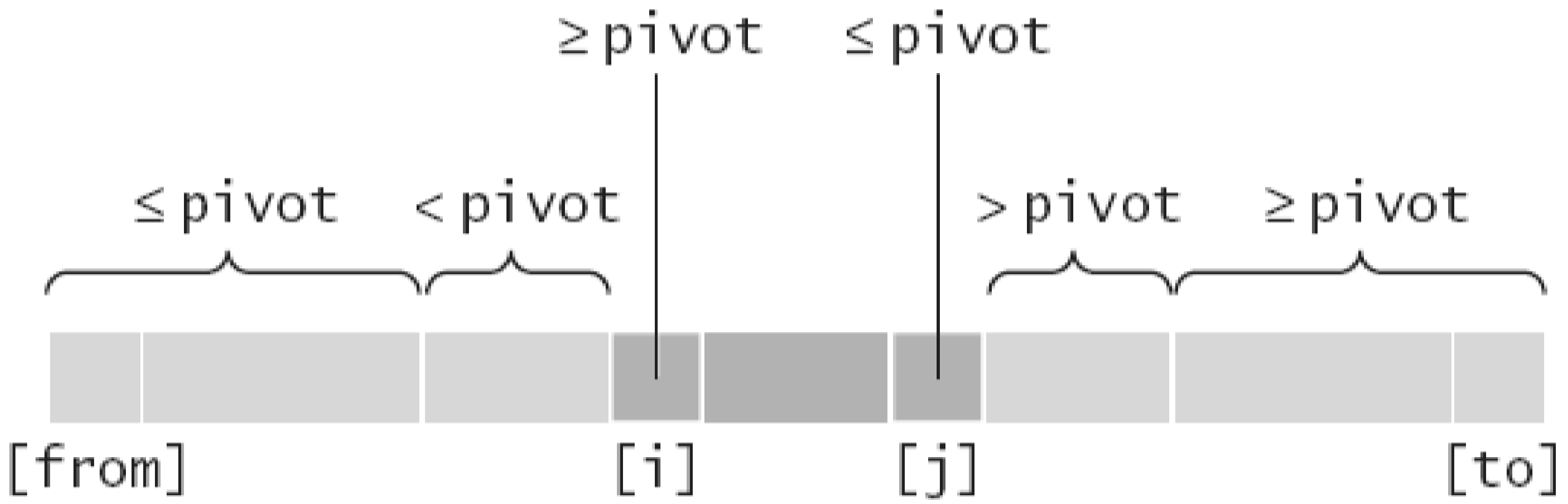
Divide and conquer

1. Partition the range
2. Sort each partition



The Quicksort Algorithm





≥ pivot
≤ pivot
≤ pivot
< pivot
> pivot
≥ pivot
[from]
[i]
[j]
[to]

Quick Sort

- Fast and reliable method for sorting data.
- Divide and conquer technique is used.
- Partition the array into two groups . The first group contains those elements less than some arbitrary **chosen value** taken from the set.
- Second group contains those elements greater than or equal to the **chosen value**.
- The chosen value is known as **pivot.**
- Once the array has been rearranged in this way with respect to pivot, the same partitioning procedure is **recursively** applied to each of the two subsets.

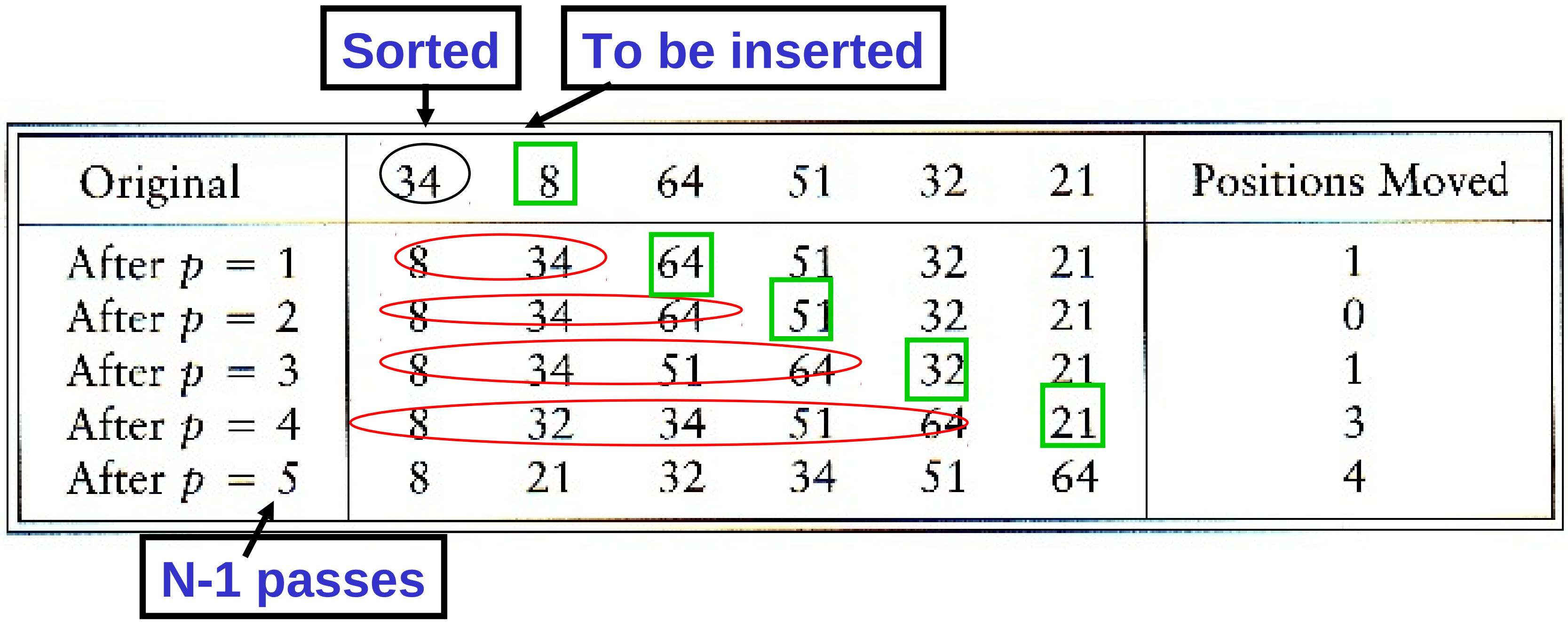
Quick Sort

- Select the first element as pivot element.
  - Move up pointer from left to right in search of larger element than pivot.
  - Move down pointer from right to left in search of an element smaller than pivot.
  - If such elements are found the elements are to be swapped.
- This process continues till the 'up' pointer crosses the 'down' pointer.
- If up pointer crosses the down pointer, the position for pivot is found and interchange pivot with the element at down position.

Insertion Sort

- One of the simplest sorting algorithms
- Consists of N-1 passes
- Algorithm
  - Initially p = 1
  - Let the first p elements be sorted.
  - Insert the (p+1)th element properly in the list so that now p+1 elements are sorted.
  - increment p and go to step (3)

Example


Sorted
To be inserted
N-1 passes

Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	8	34	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	0
After $p = 3$	8	34	51	64	32	21	1
After $p = 4$	8	32	34	51	64	21	3
After $p = 5$	8	21	32	34	51	64	4

- Pass 0: check 1st element, no change, 1st element sorted (usually omitted)
- Pass 1: check 2nd element, compare with 1st element, swap the position if needed, first 2 element sorted

- Pass 2: check 3rd element, compare with 1st and 2nd element, insert to correct position, first 3 elements sorted…
- Pass k: check (k+1)th element, compare with 1st to kth elements, insert to correct position, first k+1 elements sorted
- Pass N-1: check Nth element, compare with 1st to (N-1)th elements, insert to correct position, N elements sorted

Extended Example

To sort the following numbers in increasing order:

34 8 64 51 32 21

p = 1; tmp = 8;

34 > tmp, so second element is set to 34.

We have reached the front of the list. Thus, 1st position = tmp

After second pass: 8 34 64 51 32 21

(first 2 elements are sorted)

p = 2; tmp = 64;

34 < 64, so stop at 3rd position and set 3rd position = 64

After third pass: 8 34 64 51 32 21

(first 3 elements are sorted)

Extended Example

p = 3; tmp = 51;

51 < 64, so we have 8   34   64   64 32   21,

34 < 51, so stop at 2nd position, set 3rd position = tmp,

After fourth pass: 8   34   51 64 32   21

(first 4 elements are sorted)

p = 4; tmp = 32,

32 < 64, so 8   34   51 64 64   21,

32 < 51, so 8   34   51 51 64   21,

next 32 < 34, so 8   34   34, 51 64   21,

next 32 > 8, so stop at 1st position and set 2nd position = 32,

After fifth pass: 8   32   34   51   64   21

p = 5; tmp = 21, . . .

After sixth pass: 8   21 32 34   51   64

Tightness of the Bound

- The bound is tight $\Theta(N^2)$
- That is, there exists some input which actually uses $\Omega(N^2)$ time
- The worst input: reverse sorted list
  - When a[p] is inserted into the sorted a[0..p-1], we need to compare a[p] with all elements in a[0..p-1] and move each element one position to the right

    $\Rightarrow$ $\Omega(i)$ steps
  - the total number of steps is $\Omega(\Sigma i) = \Omega\ (N(N-1)/2) = \Omega(N^2)$

Best Case Analysis

- The best input: already sorted in required order
  - When inserting A[p] into the sorted A[0..p-1], only one comparison (compare A[p] with A[p-1]) needed
  - No data movement
  - For each iteration of the outer for-loop, the inner for-loop terminates after checking the loop condition once => O(N) time
- If input is *nearly sorted*, insertion sort runs fast

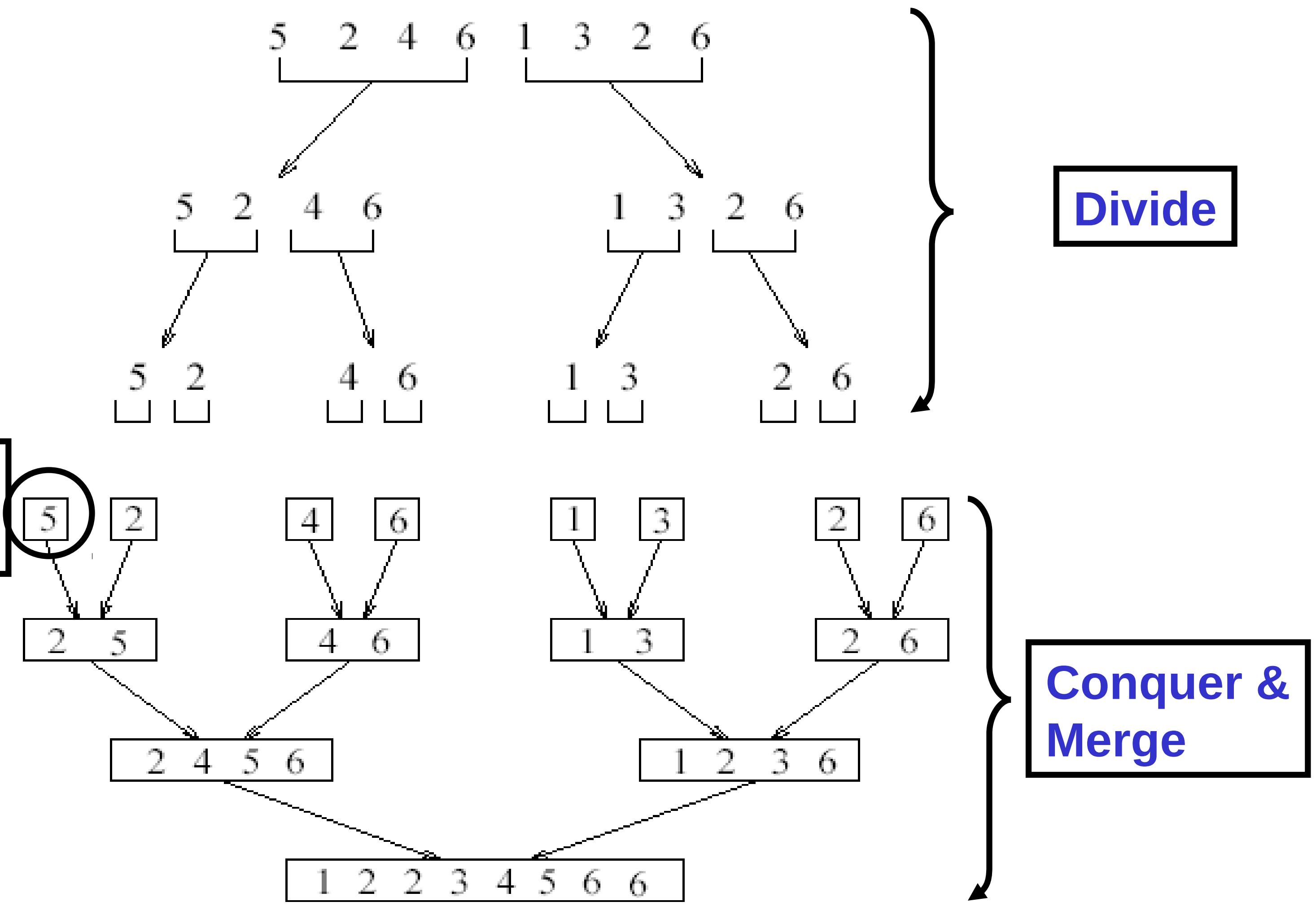
Merge Sort

- Merge sort is based on divide-and-conquer strategy
  - Divide the list into two smaller lists of about equal sizes
  - Sort each smaller list *recursively*
  - Merge the two sorted lists to get one sorted list
- Our Focus
  - How do we divide the list?
  - How do we merge the two sorted lists?
  - For each task, how much time needed?

Merge Sort

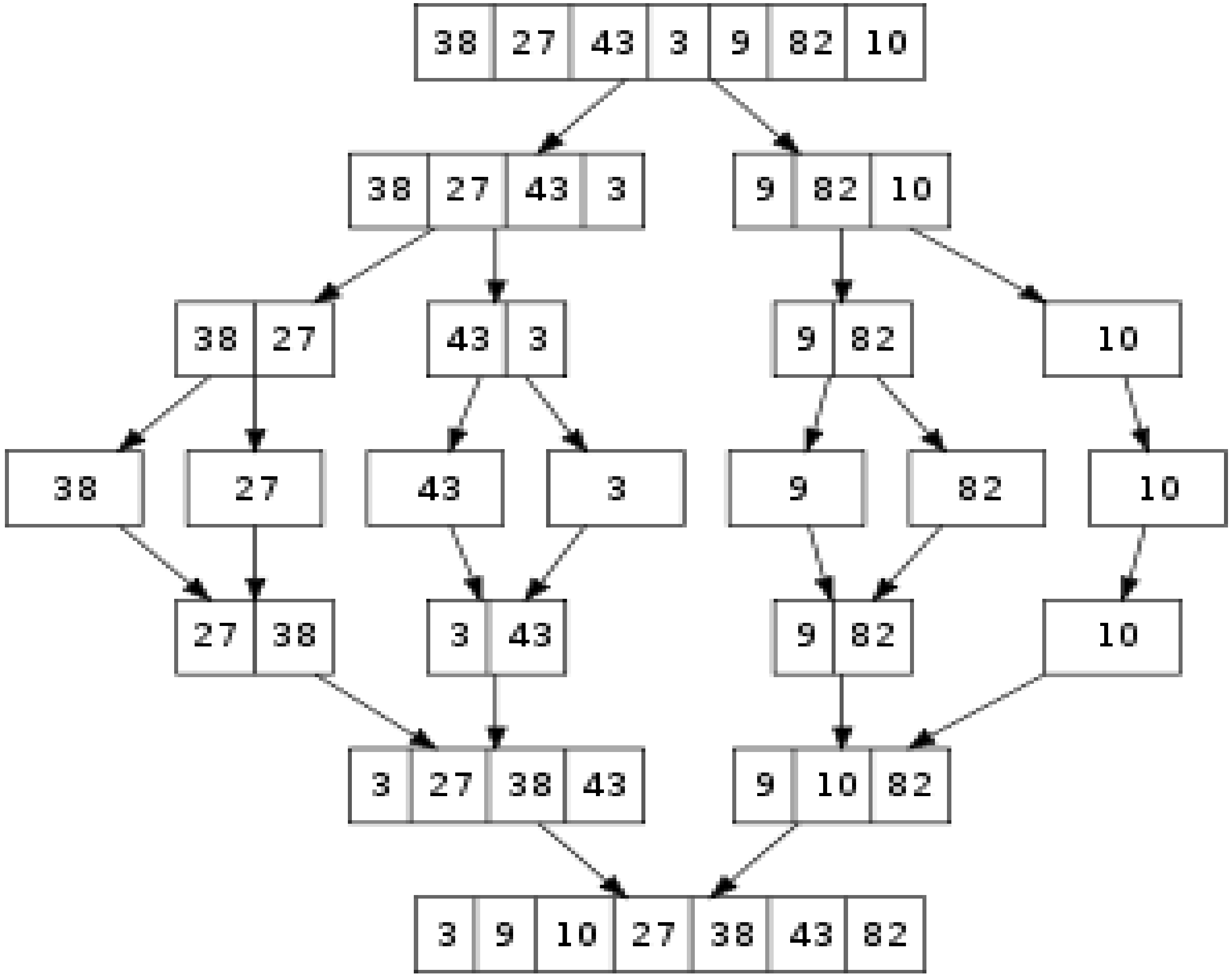
- Divide-and-conquer strategy
  - Recursively mergesort the first half and the second half
  - Merge the two sorted halves together

Merge Sort Example



Base case for recursion

Merge Sort Example



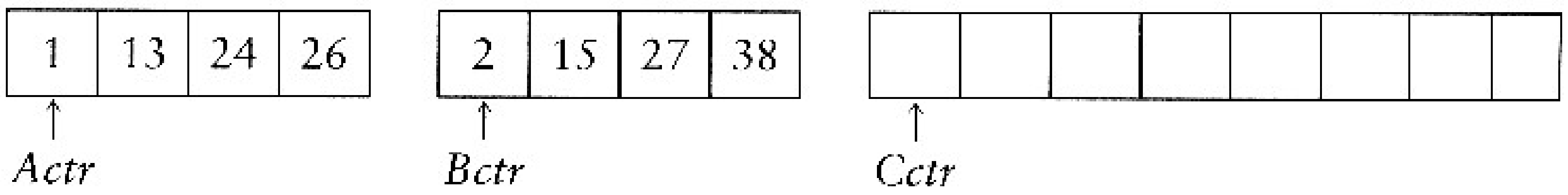
Dividing

- Cut from the middle
- If the input list is a linked list, dividing takes Θ(N) time
  - We scan the linked list, stop at the $\lfloor N/2 \rfloor$ th entry and cut the link
- If the input list is an array A[0..N-1]: dividing takes O(1) time
  - we can represent a sublist by two integers `left` and `right`: to divide `A[left..Right]`, we compute `center=(left+right)/2` and obtain `A[left..Center]` and `A[center+1..Right]`

Merging

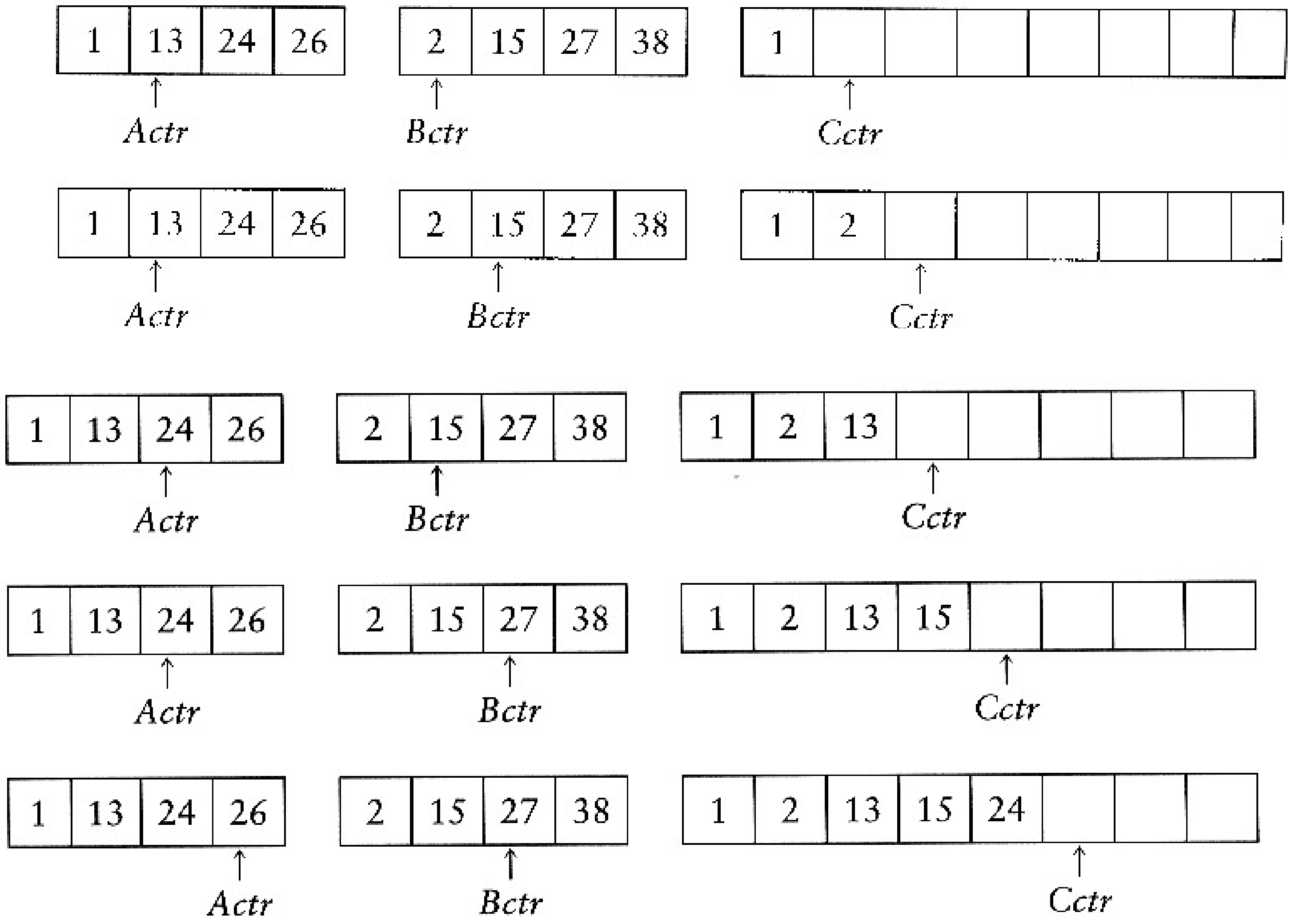
- Input: two sorted array A and B
- Output: an output sorted array C (of size equals to A+B)

- Three counters: Actr, Bctr, and Cctr
  - Step1: Initially set to the beginning of their respective arrays

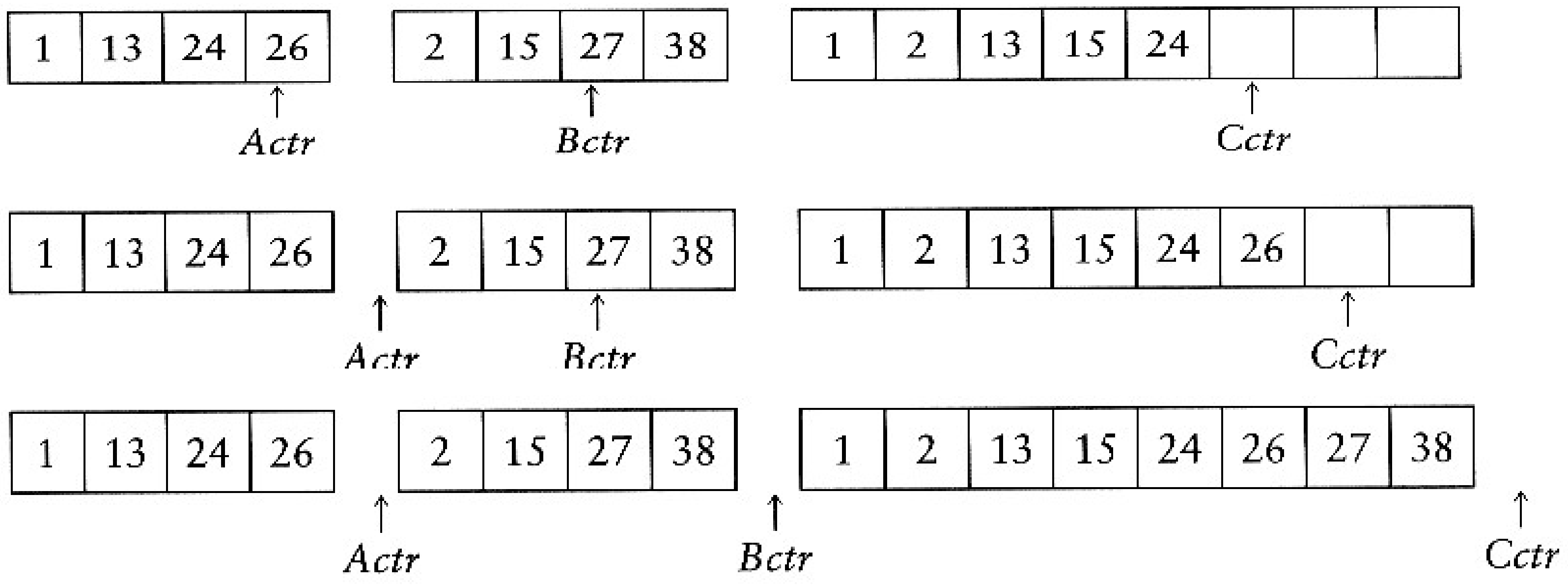


  - Step 2: The smaller of A[Actr] and B[Bctr] is copied to the next entry in C, and the appropriate counters are advanced
  - Step 3: When either input list is exhausted, the remainder of the other list is copied to C

Example: Merge



Example: Merge...



1. Running time analysis:
   - Clearly, `merge` takes O(m1 + m2) where m1 and m2 are the sizes of the two sublists.
2. Space requirement:
   - merging two sorted lists requires linear extra memory
   - additional work to copy to the temporary array and back

```
Algorithm merge(A, p, q, r)
Input: Subarrays A[p..l] and A[q..r] s.t. $p \le l = q - 1 < r$.
Output: A[p..r] is sorted.
(* T is a temporary array. *)
1.  $k = p$; $i = 0$; $l = q - 1$;
2.  while $p \le l$ and $q \le r$
3.      do if $A[p] \le A[q]$
4.              then $T[i] = A[p]$; $i = i + 1$; $p = p + 1$;
5.              else $T[i] = A[q]$; $i = i + 1$; $q = q + 1$;
6.  while $p \le l$
7.      do $T[i] = A[p]$; $i = i + 1$; $p = p + 1$;
8.  while $q \le r$
9.      do $T[i] = A[q]$; $i = i + 1$; $q = q + 1$;
10. for $i = k$ to $r$
11.     do $A[i] = T[i - k]$;
```

Step 1

Step 2

Step 3

Only one while loop will be actually executed

Analysis of Merge Sort

- Let T(N) denote the worst-case running time of mergesort to sort N numbers.
- Assume that N is a power of 2.

- Divide step: O(1) time
- Conquer step: 2T(N/2) time
- Combine (or merge) step: O(N) time

Recurrence equation:

$T(1) = 1$

$T(N) = 2T(N/2) + N$

Analysis: Solving Recurrence

$$T(N) = 2T\left(\frac{N}{2}\right) + N$$
$$= 2\left(2T\left(\frac{N}{4}\right) + \frac{N}{2}\right) + N$$
$$= 4T\left(\frac{N}{4}\right) + 2N$$
$$= 4\left(2T\left(\frac{N}{8}\right) + \frac{N}{4}\right) + 2N$$
$$= 8T\left(\frac{N}{8}\right) + 3N = \cdots$$
$$= 2^k T\left(\frac{N}{2^k}\right) + kN$$

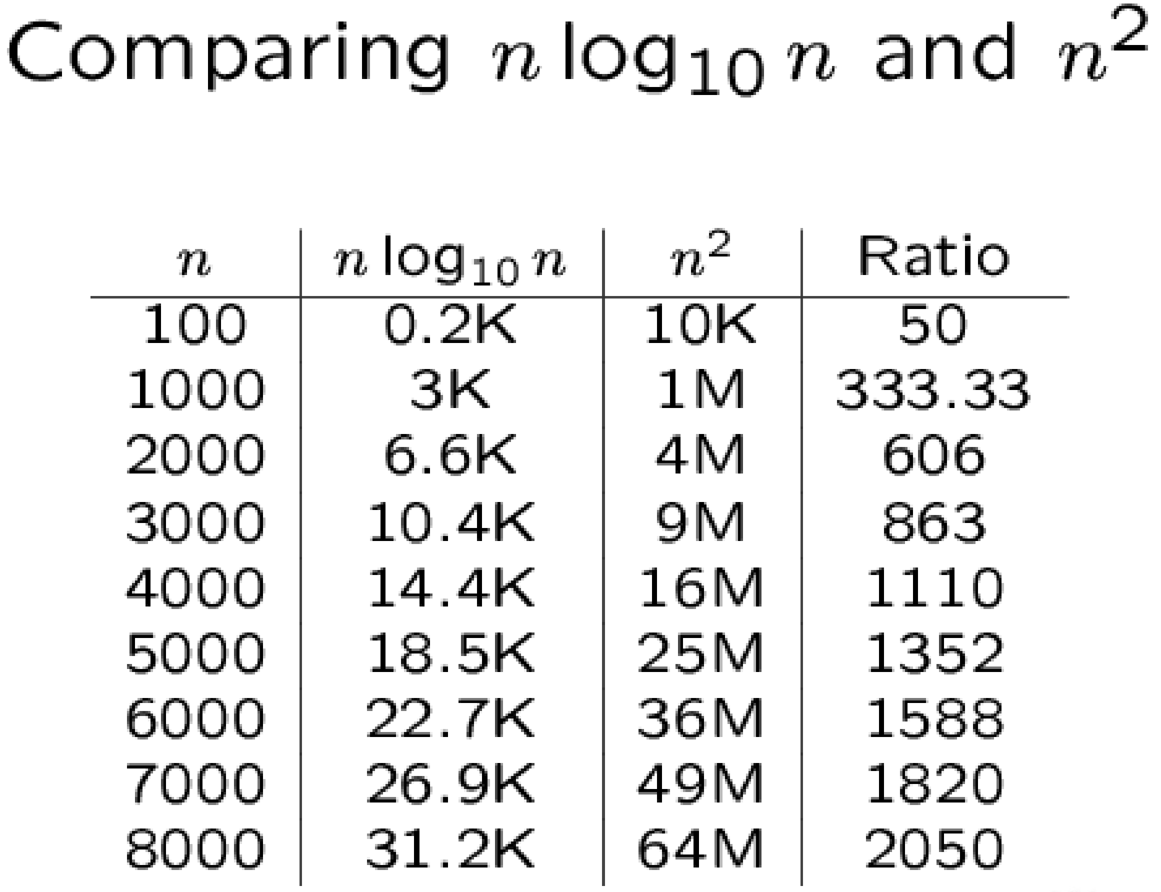
Since $N=2^k$, we have $k=\log_2 n$

$$T(N) = 2^k T\left(\frac{N}{2^k}\right) + kN$$
$$= N + N\log N$$
$$= O(N\log N)$$

Merge sort will take O(NlogN) time

VIVEN Embedded Academy

Comparison: $O(N^2)$ and $O(N\log N)$



Comparing $n \log_{10} n$ and n^2

n	$n \log_{10} n$	n^2	Ratio
100	0.2K	10K	50
1000	3K	1M	333.33
2000	6.6K	4M	606
3000	10.4K	9M	863
4000	14.4K	16M	1110
5000	18.5K	25M	1352
6000	22.7K	36M	1588
7000	26.9K	49M	1820
8000	31.2K	64M	2050