DATA STRUCTURES AND ALGORITHMS

Data Structures

Arrays

Description of various Data Structures : Arrays

- An array is defined as a set of finite number of homogeneous elements or same data items.
- It means an array can contain one type of data only, either all integer, all float-point number or all character.

Arrays

- Simply, declaration of array is as follows:

  int arr[10]

- Where int specifies the data type or type of elements arrays stores.

- “arr” is the name of array & the number specified inside the square brackets is the number of elements an array can store, this is also called sized or length of array.

Arrays

Following are some of the concepts to be remembered about arrays:

- The individual element of an array can be accessed by specifying name of the array, following by index or subscript inside square brackets.
- The first element of the array has index zero[0]. It means the first element and last element will be specified as:arr[0] & arr[9] Respectively.
- The elements of array will always be stored in the consecutive (continues) memory location.
- The number of elements that can be stored in an array, that is the size of array or its length is given by the following equation:

  (Upperbound-lowerbound)+1

Arrays (Contd.)

- **For the above array it would be**

  (9-0)+1=10,where 0 is the lower bound of array and 9 is the upper bound of array.

- Array can always be read or written through loop. If we read a one-dimensional array it require one loop for reading and other for writing the array.

Arrays

For example: Reading an array

```
For(i=0;i<=9;i++)
        scanf("%d",&arr[i]);
```

For example: Writing an array

```
For(i=0;i<=9;i++)
        printf("%d",arr[i]);
```

Arrays

- If we are reading or writing two-dimensional array it would require two loops. And similarly the array of a N dimension would required N loops.

- **Some common operation performed on array are:**
  - ✓ Creation of an array
  - ✓ Traversing an array

Arrays

- Insertion of new element
- Deletion of required element
- Modification of an element
- Merging of arrays

Linked Lists

Array Usage – A Perspective

Consider the following example:

```
#include<stdio.h>
main( )
 {
   int num_array[50],i;
   for( i=0; i < 50; i++ )
   {
    num_array[i]=0;
    scanf( "%d",&num_array[i] );
    fflush(stdin);
   }
 }
```

Array Usage – A Perspective

- When this program is compiled, the compiler estimates the amount of memory required for the variables, and also the instructions defined by you as part of the program.

- The compiler writes this information into the header of the executable file that it creates. When the executable is loaded into memory at runtime, the specified amount of memory is set aside.

- A part of the memory allocated to the program is in an area of memory called a runtime stack or the call stack. Once the size of the stack is fixed, it cannot be changed dynamically.

Array Usage – A Perspective

- Therefore, arrays present the classic problem of the programmer having allocated too few elements in the array at the time of writing the program, and then finding at run time that more values are required to be stored in the array than what had originally been defined.

- The other extreme is of the programmer allocating too many elements in the array and then finding at run time that not many values need to be stored in the array thereby resulting in wastage of precious memory.

Array Usage – A Perspective

- Moreover, array manipulation (in terms of insertion and deletion of elements from the array) is more complex and tedious, and a better alternative to all this is to go for dynamic data structures.

- Before venturing into dynamic variables, or dynamic data structures, it would be prudent at this juncture to differentiate between stack and heap variables, and their characteristics.

- Let us begin by understanding the concept of lifetime of variables.

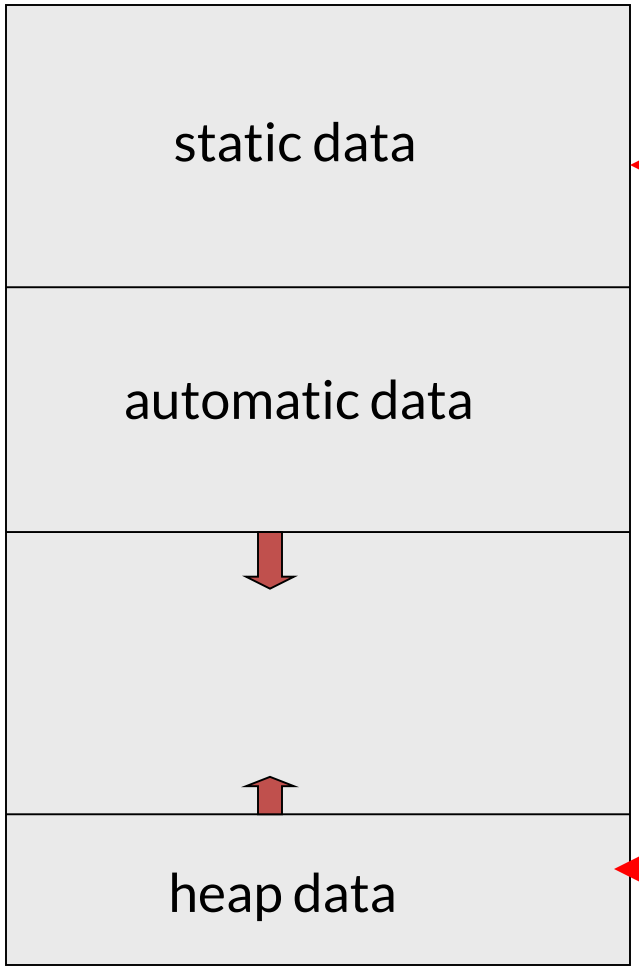
Lifetime Of A Variable

- is a run-time concept

- period of time during which a variable has memory space associated with it
  - ✓ begins when space is allocated
  - ✓ ends when space is de-allocated

- three categories of "lifetime"
  - ✓ static - start to end of program execution
  - ✓ automatic (stack) - start to end of declaring function's execution
  - ✓ heap (variable declared dynamic at runtime, and also de-allocated dynamically at runtime.

Data Memory Model



space for global variables

run-time stack - activation records added and removed as program runs (expands and shrinks in an orderly LIFO manner)

space for variables allocated at run-time (allocation and de-allocation requests occur in unpredictable order)

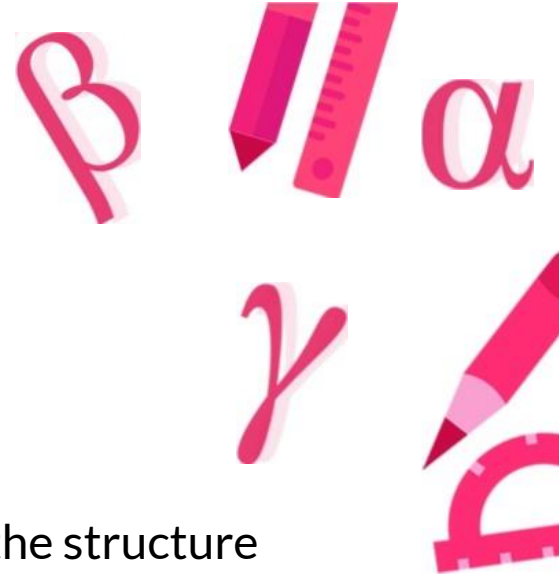
Heap Variables

- Space for heap variables is allocated from an area of runtime memory known as the heap or free store.
- Heap variables do not have an explicit name, and are accessed indirectly via a pointer.
- Memory space for heap variables is explicitly allocated at runtime using malloc( ).
- Space occupied by heap variables must be explicitly returned back to the heap to avoid memory leaks. This is done using the free( ) function.

Dynamic Data Structures

- Rather than pre-define array on the stack at compile time, the alternative should be to define a structure type, and dynamically declare at runtime as many instances of the structure variables as needed by the application.

- When the code containing the structure type is compiled, what the compiler sees is only a structure type declaration. It therefore, does not allocate memory for the structure type since no variable has been defined based on the structure type.



Dynamic Data Structures

- When the program begins execution, it will need to create variables of the structure type. Therefore, the language must support runtime declaration of variables.
- The problem with these variables is that they cannot be accommodated into the stack, as they were not declared at the time of compilation, and the stack would have been sized based on the stack variables already declared at compile-time.
- The C language provides the malloc() function which a program can use to declare variables dynamically.

The malloc( ) Function

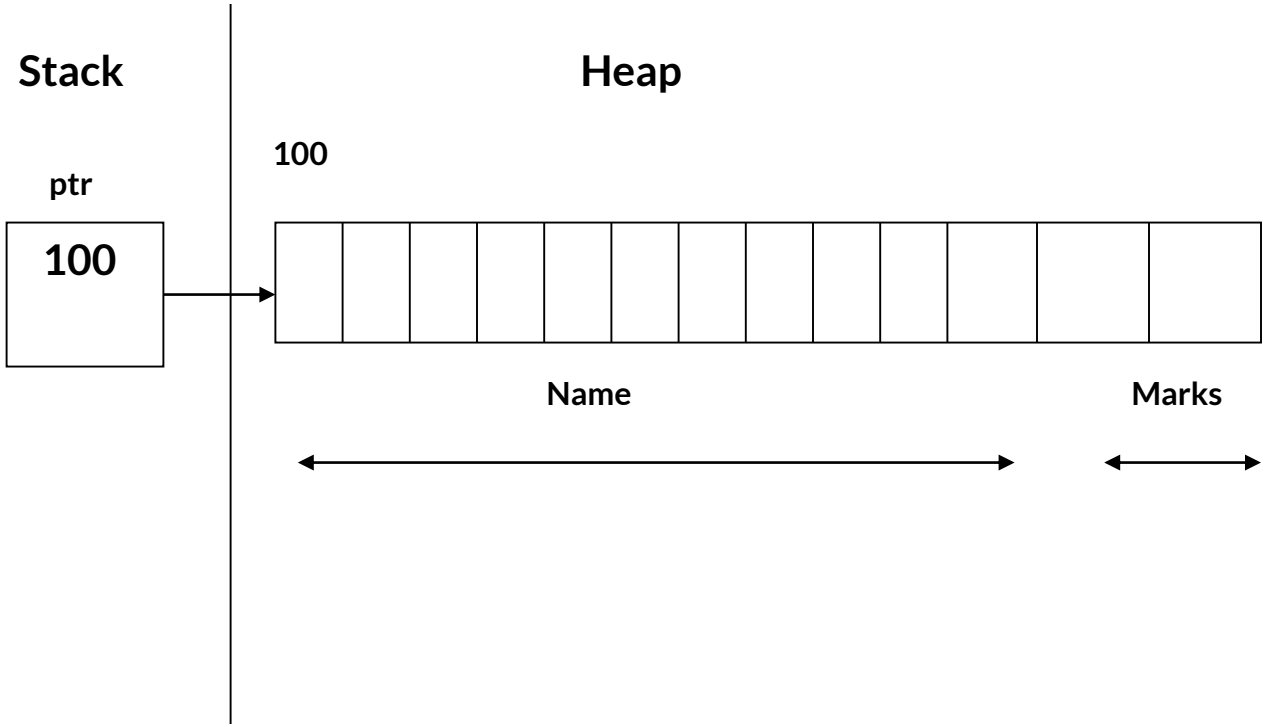
- The parameter to malloc( ) is an unsigned integer which represents the number of bytes that the programmer has requested malloc( ) to allocate on the heap.
- A more effective way of passing the number of bytes to malloc( ) would be to pass the structure type along with the sizeof( ) operator to malloc( ).
- The sizeof( ) operator can be used to determine the size of any data type in C, instead of manually determining the size and using that value. Therefore, the benefit of using sizeof( ) operator in any program makes it portable.

The malloc( ) Function

Consider the following example:

```
#include<stdio.h>
main()
 {
   struct marks_data
    {
     char name[11];
     int marks;
    };
    struct marks_data *ptr;
    /* declaration of a stack variable */
    ptr = (struct marks_data *)
    malloc(sizeof(struct marks_data));
    /* declaration of a block of memory on the heap and  the block in turn being referenced by ptr */
 }
```

The malloc( ) Function



The malloc( ) Function

- The malloc( ) function also returns the starting address to the marks_data structure variable on the heap.
- However, malloc( ) returns this not as a pointer to a structure variable of type marks_data , but as a void pointer.
- Therefore, the cast operator was used on the return value of malloc( ) to cast it as a pointer to a structure of type marks_data before being assigned to ptr, which has accordingly been defined to be a pointer to a structure of type marks_data.

Self-Referential Structures

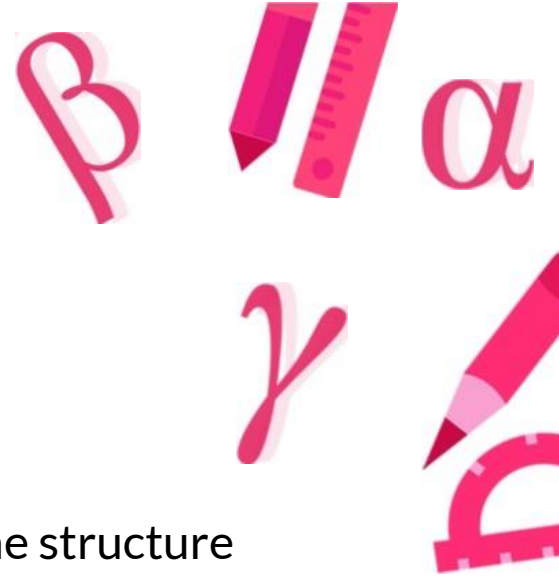
- Suppose, you have been given a task to store a list of marks. The size of the list is not known.
- If it were known, then it would have facilitated the creation of an array of the said number of elements and have the marks entered into it.
- Elements of an array are contiguously located, and therefore, array manipulation is easy using an integer variable as a subscript, or using pointer arithmetic.

Self-Referential Structures

- However, when runtime variables of a particular type are declared on the heap, let's say a structure type in which we are going to store the marks, **each variable of the structure type marks will be located at a different memory location on the heap, and not contiguously located.**

- **Therefore, these variables cannot be processed the way arrays are processed, i.e., using a subscript, or using pointer arithmetic.**

- **An answer to this is a self-referential structure.**



Self-Referential Structures

- A self-referential structure is so defined that one of the elements of the structure variable is able to reference another subsequent structure variable of the same type, wherever it may be located on the heap.

- In other words, each variable maintains a link to another variable of the same type, thus forming a non-contiguous, loosely linked data structure.

- This self-referential data structure is also called a linked list.

Declaring a Linked List

- Let us define a self-referential structure to store a list of marks the size of which may not be known.

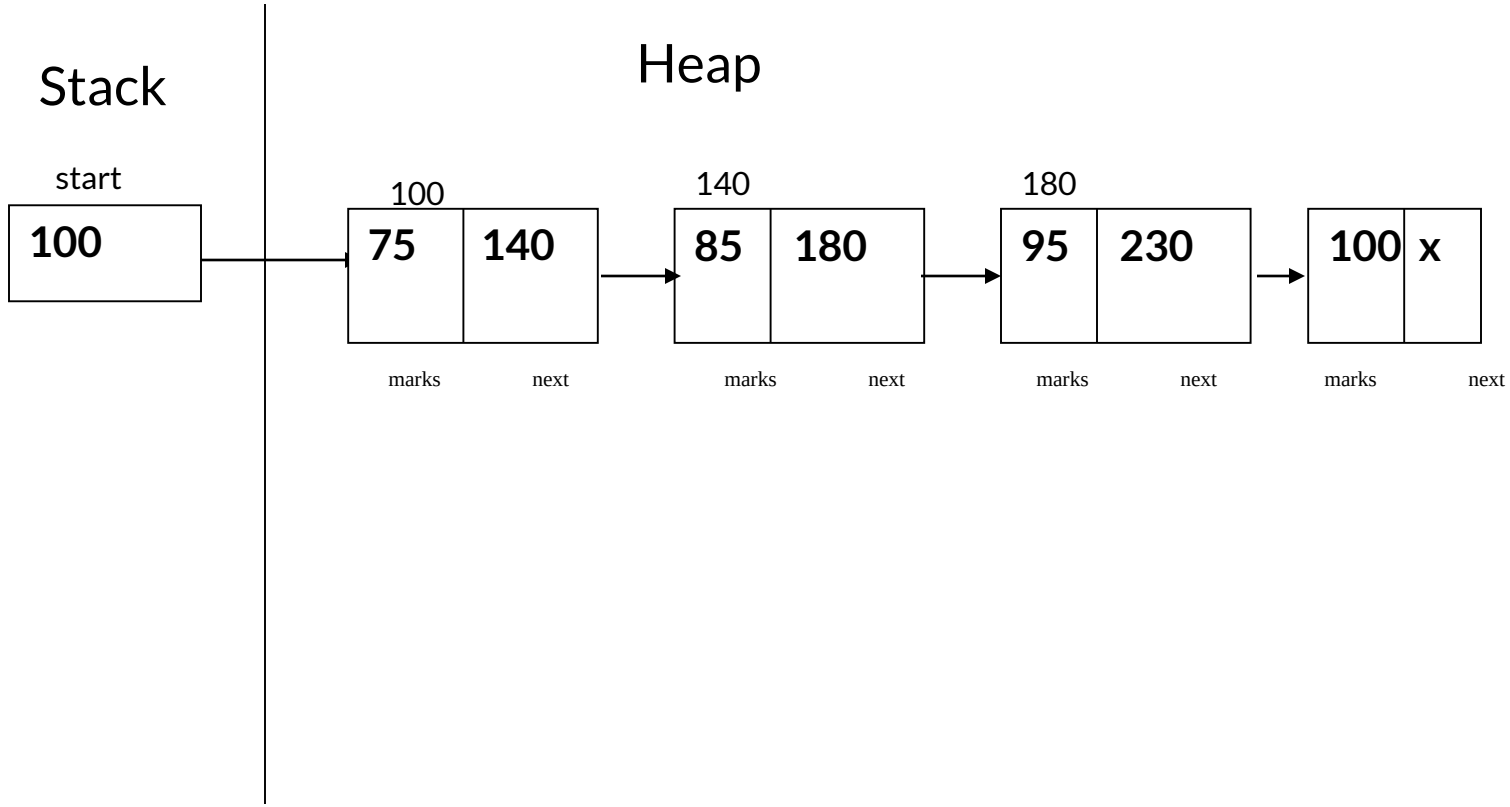
```
struct marks_list
{
   int marks;
   struct marks_list *next;
 };
```

- We have defined a structure of type marks_list. It consists of two elements, one integer element marks and the other element, a pointer next, which is a pointer to a structure of the same type, i.e., of type marks_list itself.

Declaring a Linked List

- Therefore, a part of the structure is referencing a structure type of itself, and hence the name **self-referential structure**.
- Such data structures are also popularly known as **linked lists**, since each structure variable contains a link to other structure variables of the same type.
- One can visualize a linked list as shown in the following slide:

Visualizing a Linked List



Creating a Sorted Linked List

```
#include<stdio.h>
#include<malloc.h>
struct marks_list *start, *prev;
/* variables declared outside main() are global in nature and can be accessed by other functions called from main() */
struct marks_list
{
 int marks;
 struct marks_list *next;
};
main()
 {
  struct marks_list * makenode();
  /* function  prototype declaration */
```

Creating a Sorted Linked List

```
struct marks_list *new;
start = NULL;
char menu = '0 ';
while (menu != '4')
 {
   printf( "Add Nodes    :\n");
   printf( "Delete Nodes  :\n");
   printf( "Traverse a list :\n");
   printf( "Exit          :\n");
   menu = getchar( );
```

Creating a Sorted Linked List

```
switch (menu)
{
  case '1' : addnode( );
             break;
  case '2' : deletenode( )
             break;
  case '3' : traverse( );
             break;
  case '4': exit( );
             break;
} /* end of switch */
  } /* end of main( ) */
```

Creating a Sorted Linked List

```
addnode( )
{
  char ch = 'y';
   while ( ch = = 'y' )
    {
     new = makenode();
     /* creation of a list is treated as a special case of insertion  */
     if ( start == NULL)
      {
       start = new;
      }
     else
      {
       insert();
       traverse( );
      }
```

Creating a Sorted Linked List

```
 printf("%s","Want to add more nodes\n");
 scanf( "%c", &ch );
 fflush( stdin );
} /* end of while  */
} /* end of addnode( )
```

Creating a Sorted Linked List

```
struct marks_list * makenode()
 {
  struct marks_list *new;
  new=(struct marks_list *)  malloc(sizeof(struct(marks_list));
  scanf("%d",&new->marks);
  new->next = NULL;
  return(new);
 }
```

Creating a Sorted Linked List

```
insert(struct marks_list *start)
{
 struct marks_list *ptr, *prev;
 for(ptr=start,prev=start;(ptr);prev=ptr,ptr=ptr->next)
 {
  if (new->marks < start->marks)
   {
    /* insertion at the beginning of a  list */
    new->next = start;
    start = new;
  }
```

Creating a Sorted Linked List

```
/* insertion in the middle of a list */
  if(new->marks  > ptr->marks)
   {
    continue;
   }
  else
   {
    prev->next = new;
    new->next = ptr;
   }
} /* end of for loop */
```

Creating a Sorted Linked List

```
/* insertion at the end of the list */
if (ptr == null)
 {
  prev->next = new;
  new->next = null;
 }
} /* end of insert */
```

Searching a Value in a Linked List

```
struct marks_list *search( int val)
{
 for( ptr = start; (ptr); ptr = ptr->next)
  {
    if (val = = ptr-> marks)
      return ptr;
  }
```

Deleting a Node From a Linked List

```
delete ( )
 {
  struct marks_list *ptr, *prev, *temp;
  int score;
   /* search the linked list for the value to be deleted */
   scanf("%d", &score);
  fflush(stdin);
  for (ptr = start, prev = start; (ptr); prev = ptr, ptr = ptr->next)
  {
   /* deletion of the first node in the list */
   if (score = = start-> marks)
   {
    temp =start;
    start = start-> next;
    free(temp);
   }
```

Deleting a Node From a Linked List

```
/* this code would hold true for deletion in the middle and at the end of a linked list */
 if (score = = ptr-> marks)
   {
    prev-> next = ptr-> next;
    free(ptr);
    }
  }/* end of for loop */
} /* end of delete */
```

Summary

In this session, you learnt to:

Describe the limitations of using a data structure such as an array

Define stack and heap variables

Describe the characteristics of stack and heap variables

State the need for dynamic memory allocation

Use the malloc() function to allocate memory

Define self-referential structures and their advantages

Write code to:

- Create a sorted linked list,
- Insert nodes into a sorted linked list
- Traverse a linked list
- Delete nodes from a linked list

Stacks

What is a Stack?

A stack is a data structure in which insertions and deletions can only be done at the top.

A common example of a stack, which permits the selection of only its end elements, is a stack of books.

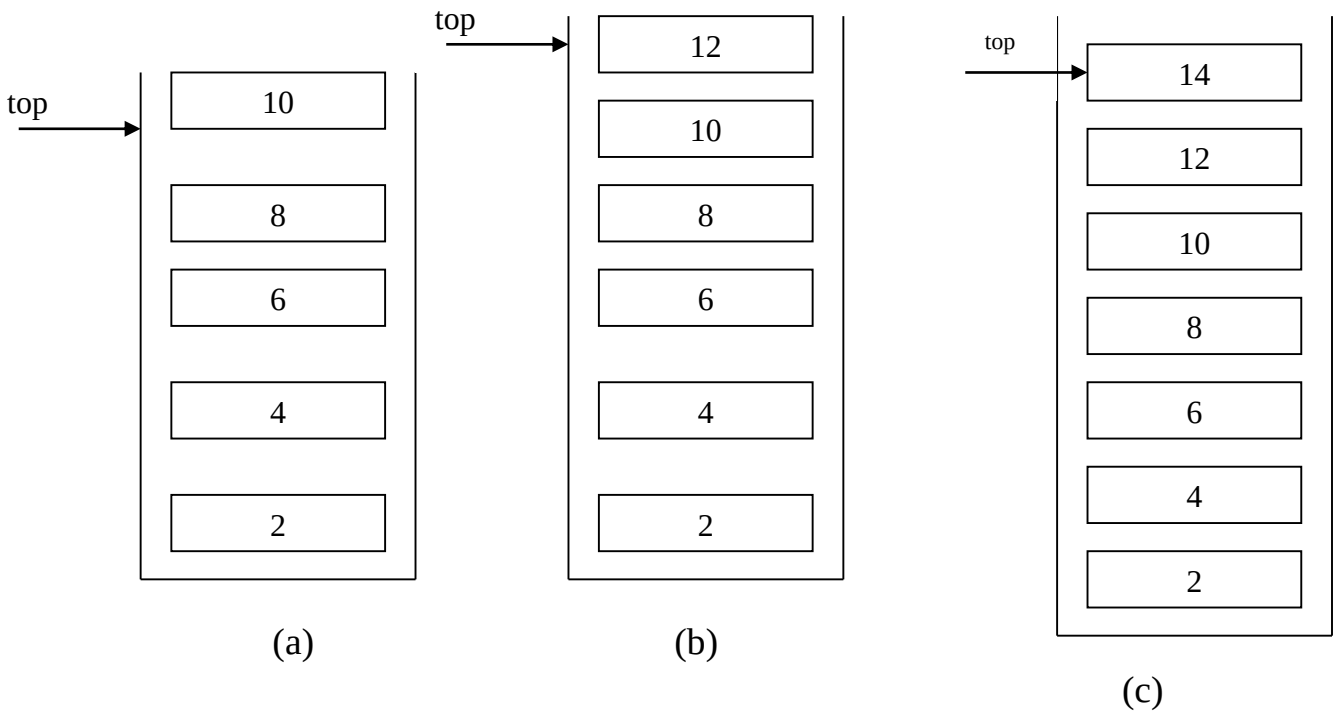
A person desiring a book can pick up only the book at the top, and if one wants to place a plate on the pile of plates, one has to place it on the top.

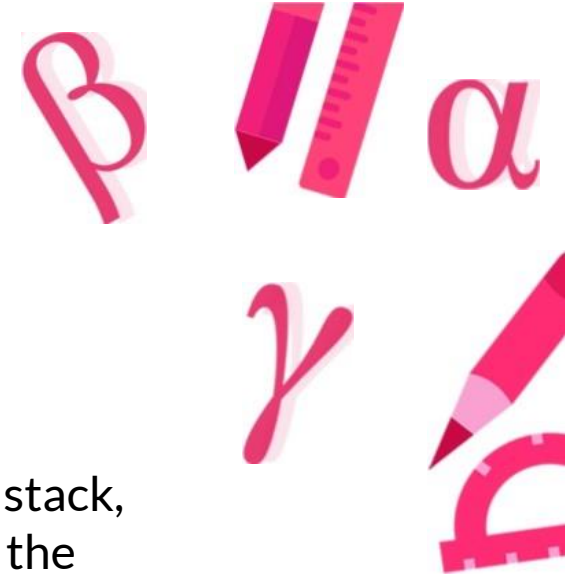
What is a Stack?

- You would have noticed that whenever a book is placed on top of the stack, the top of the stack moves upward to correspond to the new element (the stack grows).

- And, whenever a book is picked, or removed from the top of the stack, the top of the stack moves downward to correspond to the new highest element (the stack shrinks).

What is a Stack?





Characteristics of a Stack

- When you add an element to the stack, you say that you **push** it on the stack, and if you delete an element from a stack, you say that you **pop** it from the stack.

- Since the last item pushed into the stack is always the first item to be popped from the stack, a stack is also called as a **Last In, First Out or a LIFO structure**.

- Unlike an array that is static in terms of its size, **a stack is a dynamic data structure.**

Characteristics of a Stack

- Since the definition of the stack provides for the insertion and deletion of nodes into it, a stack can grow and shrink dynamically at runtime.

- An ideal implementation of a stack would be a special kind of linked list in which insertions and deletions can only be done at the top of the list.

Operations on Stacks

Some of the typical operations performed on stacks are:

- **create (s)** – to create s as an empty stack
- **push (s, i)** – to insert the element i on top of the stack **s**
- **pop (s)** – to remove the top element of the stack and to return the removed element as a function value.
- **top (s)** – to return the top element of stack(s)
- **empty(s)** – to check whether the stack is empty or not. It returns true if the stack is empty, and returns false otherwise.

Implementation of Stacks

- An array can be used to implement a stack.
- But since array size is defined at compile time, it cannot grow dynamically at runtime, and therefore, an attempt to insert an element into a array implementation of a stack that is already full causes a stack overflow.
- A stack, by definition, is a data structure that cannot be full since it can dynamically grow and shrink at runtime.

Implementation of Stacks

- An ideal implementation for a stack is a linked list that can dynamically grow and shrink at runtime.
- Since you are going to employ a variation of a linked list that functions as a stack, you need to employ an additional pointer (top) that always points to the first node in the stack, or the top of the stack.
- It is using top that a node will either be inserted at the beginning or top of the stack (push a node into the stack), or deleted from the top of the stack (popping a node at the top or beginning of the stack).

Code Implementing for a Stack

- The push( ) and the pop( ) operations on a stack are analogous to insert-first-node and delete-first-node operations on a linked list that functions as a stack.

```
struct stack
{  int info;
 struct stack *next;
};
/* pointer declaration to point to the top of the stack */
struct stack *top;
main( )
{
```

Code Implementing for a Stack

```
top = NULL;
char menu = '0 ';
while (menu != '3')
 {
   printf( "Add Nodes    :\n");
   printf( "Delete Nodes  :\n");
   printf( "Exit          :\n");
   menu = getchar( );
  switch (menu)
{
  case '1' : push( );
             break;
  case '2' : pop( )
             break;
```

Code Implementing for a Stack

```
case '3': exit( );
          break;
} /* end of switch */
  } /* end of main( ) */
```

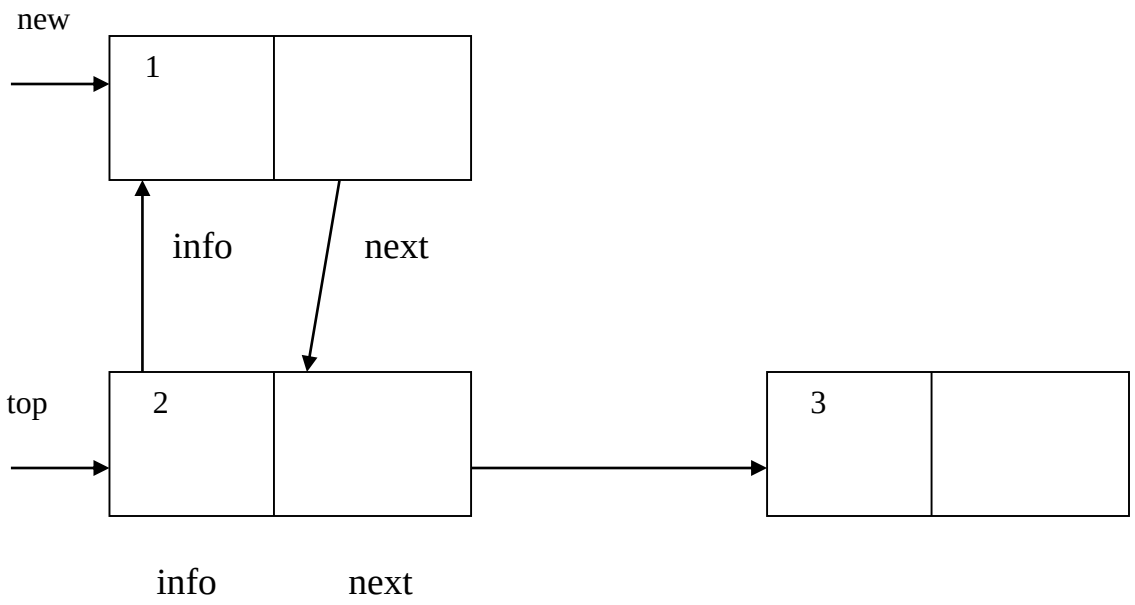
Implementing push( )

```
push( )
{
 struct stack * new;
 char ch;
 ch = 'y';
 while (ch = = 'y')
 {
   new = getnode( )
   /* checking for an empty stack */
   if ( top = = null)
   {
    top = new;
   }
```

Implementing push( )

```
  else
  {
    new->next = top;
    top = new;
  }
 printf("%s","Want to add more nodes\n");
 scanf( "%c", &ch );
 fflush( stdin );
} /* end of while  */
} /* end of push( )
```

A View of the Stack After Insertion



Creating a Node on a Stack

```
struct stack * makenode()
  {
   struct stack *new;
   new=(struct stack *)  malloc(sizeof(struct(stack));
   scanf("%d",&new->info);
   new->next = NULL;
   return(new);
  }
```

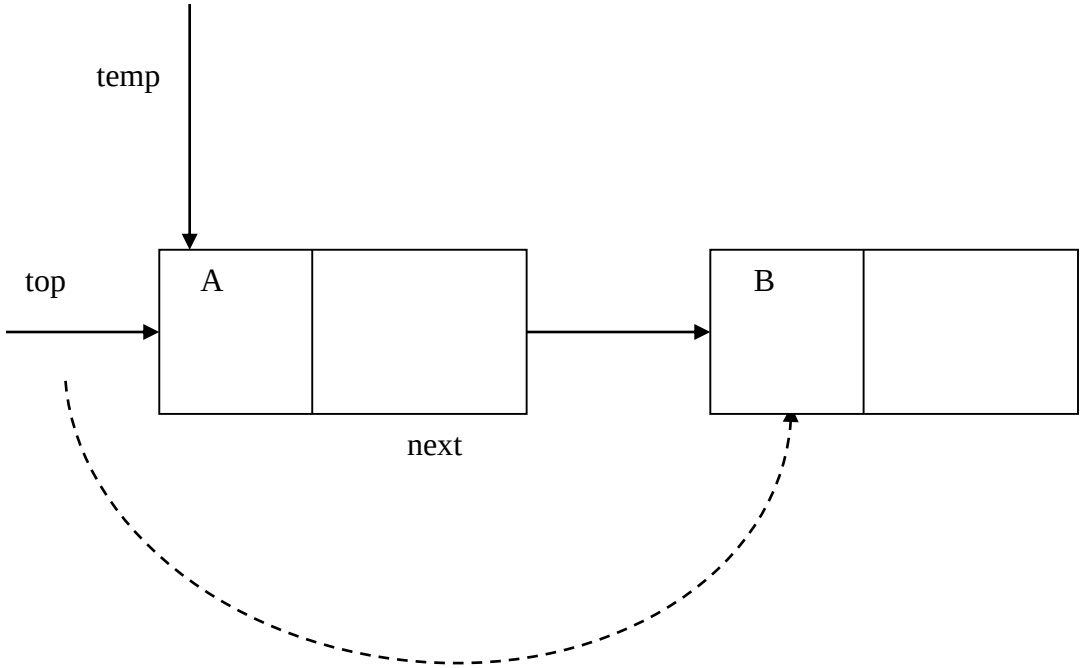
Implementing pop( )

```
pop( )
{
 struct stack * temp;
 int x;
 /* check for an empty stack */
 if (top = = null)
  {
    printf (“Cannot remove nodes from an empty stack */
    exit( );
   }
```

Implementing pop( )

```
else
{
   temp = top;
    x = top->info;
    top = top->next;
    free( temp);
    return x;
 }
```

A View of the Stack After Deletion



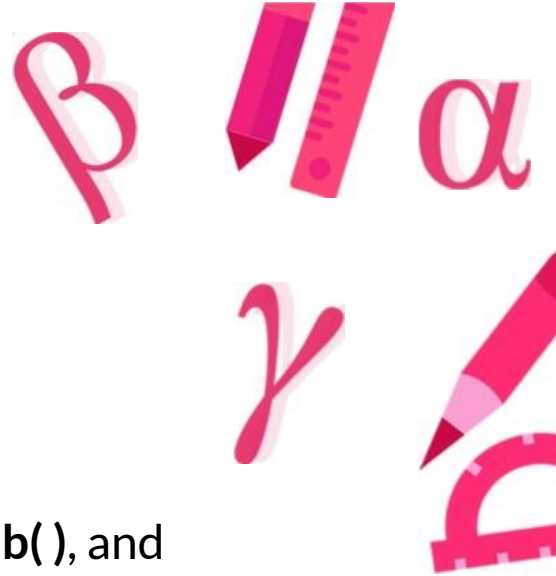
Applications of Stacks

- **As a stack is a LIFO structure, it is an appropriate data structure for applications in which information must be saved and later retrieved in reverse order.**

- Consider what happens within a computer when function main( ) calls another function.

- How does a program remember where to resume execution from after returning from a function call?

- From where does it pick up the values of the local variables in the function main( ) after returning from the subprogram?



Applications of Stacks

- As an example, let **main( )** call **a( )**. Function **a( )**, in turn, calls function **b( )**, and function **b( )** in turn invokes function **c( )**.
- **main( )** is the first one to execute, but it is the last one to finish, after **a( )** has finished and returned.
- **a( )** cannot finish its work until **b( )** has finished and returned. **b( )** cannot finish its work until **c( )** has finished and returned.

Applications of Stacks

- When **a( )** is called, its calling information is pushed on to the stack (calling information consists of the address of the return instruction in **main( )** after **a( )** was called, and the local variables and parameter declarations in **main( )**.

- When **b( )** is called from **a( )**, b( )'s calling information is pushed onto the stack (calling information consists of the address of the return instruction in **a( )** after **b( )** was called, and the local variables and parameter declarations in **a( )**).

Applications of Stacks

- Then, when **b( )** calls **c( )**, **c( )**'s calling information is pushed onto the stack (calling information consists of the address of the return instruction in **b( )** after **c( )** was called, and the local variables and parameter declarations in **b( )**).
- When **c( )** finishes execution, the information needed to return to **b( )** is retrieved by popping the stack.
- Then, when **b( )** finishes execution, its return address is popped from the stack to return to **a( )**

Applications of Stacks

- Finally, when **a( )** completes, the stack is again popped to get back to **main( )**.
- When **main( )** finishes, the stack becomes empty.
- Thus, a stack plays an important role in function calls.
- The same technique is used in recursion when a function invokes itself.

Queues

Defining a Queue

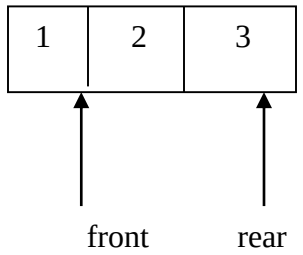
- A Queue is a data structure in which elements are added at one end (called the **rear**), and elements are removed from the other end (called the **front**).
- You come across a number of examples of a queue in real life situations.
- For example, consider a line of students at a fee counter. Whenever a student enters the queue, he stands at the end of the queue (analogous to the addition of nodes to the queue)

Defining a Queue

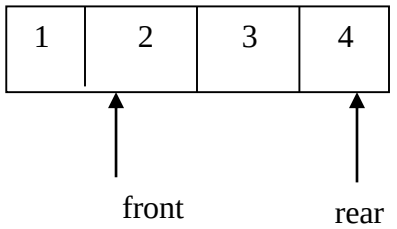
- Every time the student at the front of the queue deposits the fee, he leaves the queue (analogous to deleting nodes from a queue).
- The student who comes first in the queue is the one who leaves the queue first.

Therefore, a queue is commonly called a first-in-first-out or a FIFO data structure.

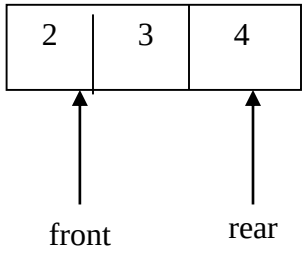
Queue Insertions and Deletions



(a)



(b)



(c)

Queue Operations

- To complete this definition of a queue, you must specify all the operations that it permits.
- The first step you must perform in working with any queue is to initialize the queue for further use. Other important operations are to add an element, and to delete an element from a queue.
- Adding an element is popularly known as ENQ and deleting an element is know as DEQ. The following slide lists operations typically performed on a queue.

Queue Operations

create(q) – which creates q as an empty queue

enq(i) – adds the element i to the rear of the queue and returns the new queue

deq(q) – removes the element at the front end of the queue (q) and returns the resulting queue as well as the removed element

empty (q) – it checks the queue (q) whether it is empty or not. It returns true if the queue is empty and returns false otherwise

front(q) – returns the front element of the queue without changing the queue

queuesize (q) – returns the number of entries in the queue

Implementing Queues

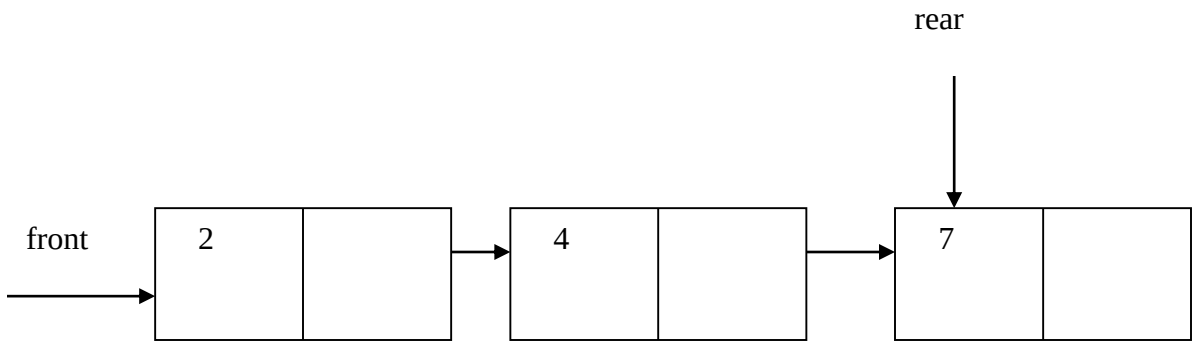
- Linked lists offer a flexible implementation of a queue since insertion and deletion of elements into a list are simple, and a linked list has the advantage of dynamically growing or shrinking at runtime.

- Having a list that has the functionality of a queue implies that insertions to the list can only be done at the rear of the list, and deletions to the list can only be done at front of the list.

Implementing Queues

- Queue functionality of a linked list can be achieved by having two pointers **front** and **rear**, pointing to the first element, and the last element of the queue respectively.

- The following figure gives a visual depiction of linked list implementation of a queue.



Queue Declaration & Operations

```
struct queue
 {
   int info;
   struct queue *next;
 };
struct queue *front, *rear;
```

An empty queue is represented by q->front = q->rear = null. Therefore, clearq( ) can be implemented as follows:

```
void clearq(struct queue * queue_pointer)
{
   queue_pointer->front = queue_pointer ->rear = null;
}
```

Queue Operations

You can determine whether a queue is empty or not by checking its front pointer. The front pointer of a queue can be passed as an argument to emptyq( ) to determine whether it is empty or not.

```
int emptyq (queue_pointer)
 {
   if (queue_pointer = = null)
    return (1);
   else
    return(0);
}
```

Insertion into a Queue

```
struct queue
 {  int info;
  struct queue *next;
};
/* pointer declarations to point to the front, and rear of the queue */
struct queue *front, *rear;
main( )
{
  front = NULL;
   rear = NULL;
```

Insertion into a Queue

```
char menu = '0 ';
while (menu != '3')
 {
   printf( "Add Nodes    :\n");
   printf( "Delete Nodes  :\n");
   printf( "Exit          :\n");
   menu = getchar( );
  switch (menu)
{
  case '1' : enq( );
             break;
  case '2' : deq( )
             break;
```

Insertion into a Queue

```
case '3': exit( );
          break;
} /* end of switch */
  } /* end of main( ) */
```

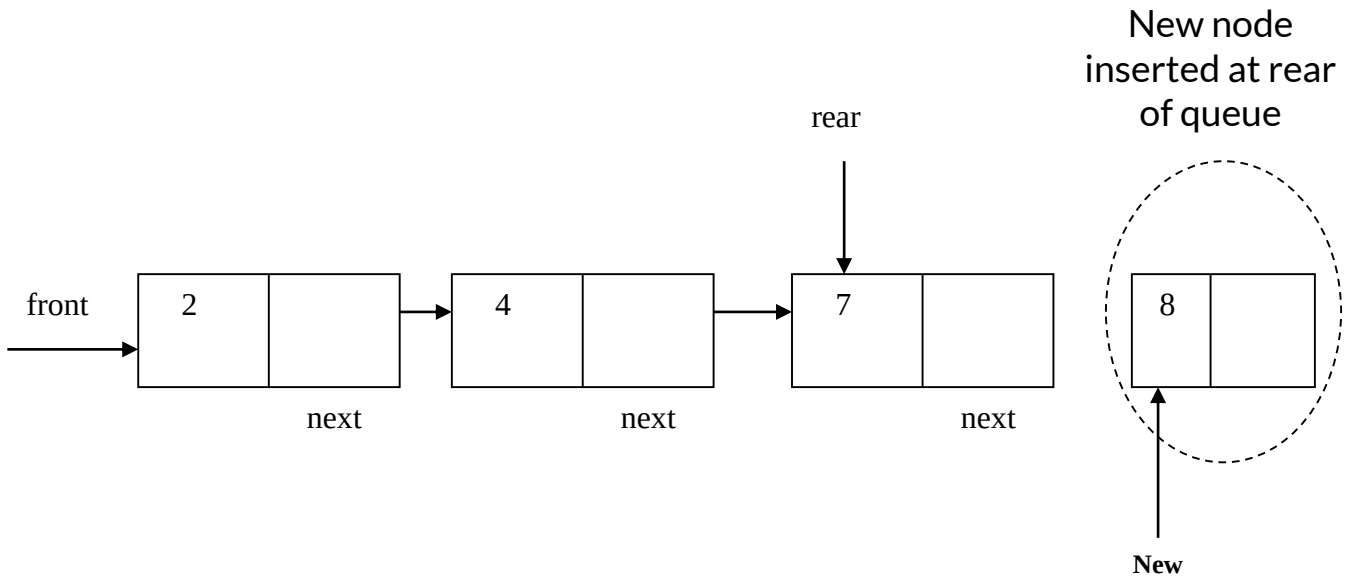
Insertion into a Queue

```
void enq( )
  {
    struct queue *new;
    new = getnode( );
   if(queue_pointer->front= =queue_pointer->rear = = null)
    {
      queue_pointer->front = new;
      queue_pointer->rear = new;
    }
    else
    {
      rear->next = new;
      rear = new;
    }
  }
```

Creating a Node on a Queue

```
struct queue * makenode()
 {
  struct queue *new;
  new=(struct queue *)  malloc(sizeof(struct(queue));
  scanf("%d",&new->info);
  new->next = NULL;
  return(new);
 }
```

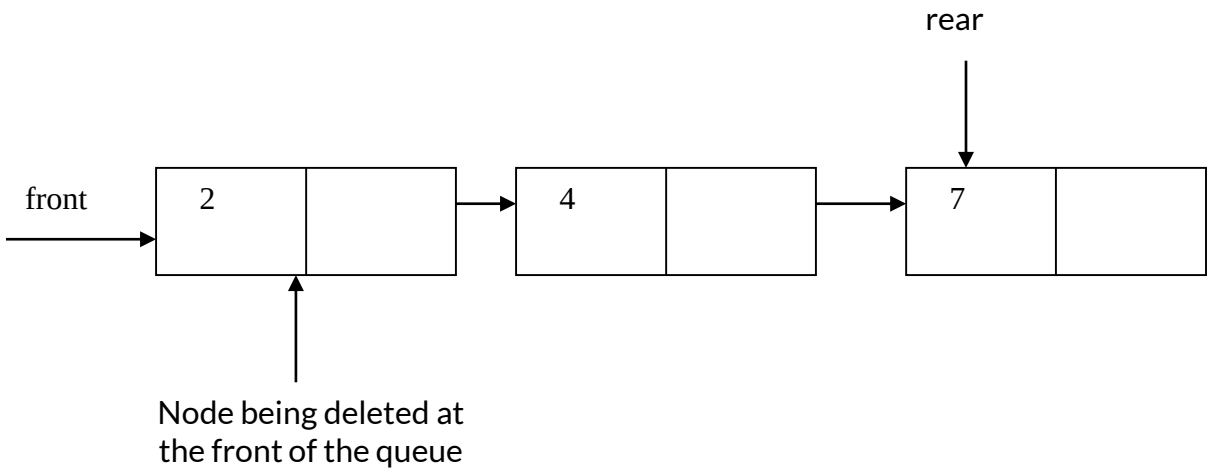
Insertion into a Queue



Deletion from a Queue

```
int deq( )
 {
   struct queue *temp;
   int x;
   if(queue_pointer->front= =queue_pointer->rear = = null)
    {
      printf( “Queue Underflow\n”);
      exit (1);
     }
    temp = front;
    x=temp->info;
    front = front->next;
    free(temp);
   if(front = = null) /* check for queue becoming empty after node deletion */
    rear = null;
   return(x);
 }
```

Deletion of a Node From a Queue



Applications of Queues

- Queues are also very useful in a time-sharing multi-user operating system where many users share the CPU simultaneously.

- Whenever a user requests the CPU to run a particular program, the operating system adds the request (by first of all converting the program into a process that is a running instance of the program, and assigning the process an ID).

This process ID is then added at the end of the queue of jobs waiting to be executed.

Applications of Queues

- Whenever the CPU is free, it executes the job that is at the front of the job queue.
- Similarly, there are queues for shared I/O devices. Each device maintains its own queue of requests.
- An example is a print queue on a network printer, which queues up print jobs issued by various users on the network.
- The first print request is the first one to be processed. New print requests are added at the end of the queue.

Doubly Linked Lists

Need For a Doubly Linked List

- The disadvantage with a singly linked list is that traversal is possible in only direction, i.e., from the beginning of the list till the end.
- If the value to be searched in a linked list is toward the end of the list, the search time would be higher in the case of a singly linked list.
- It would have been efficient had it been possible to search for a value in a linked list from the end of the list.

Properties of a Doubly Linked List

- This would be possible only if we have a doubly linked list.
- In a doubly linked list, each node has two pointers, one say, **next** pointing to the next node in the list, and another say, **prior** pointing to the previous node in the list.
- Therefore, traversing a doubly linked list in either direction is possible, **from the start to the end using next**, and **from the end of the list to the beginning of the list using prior**.

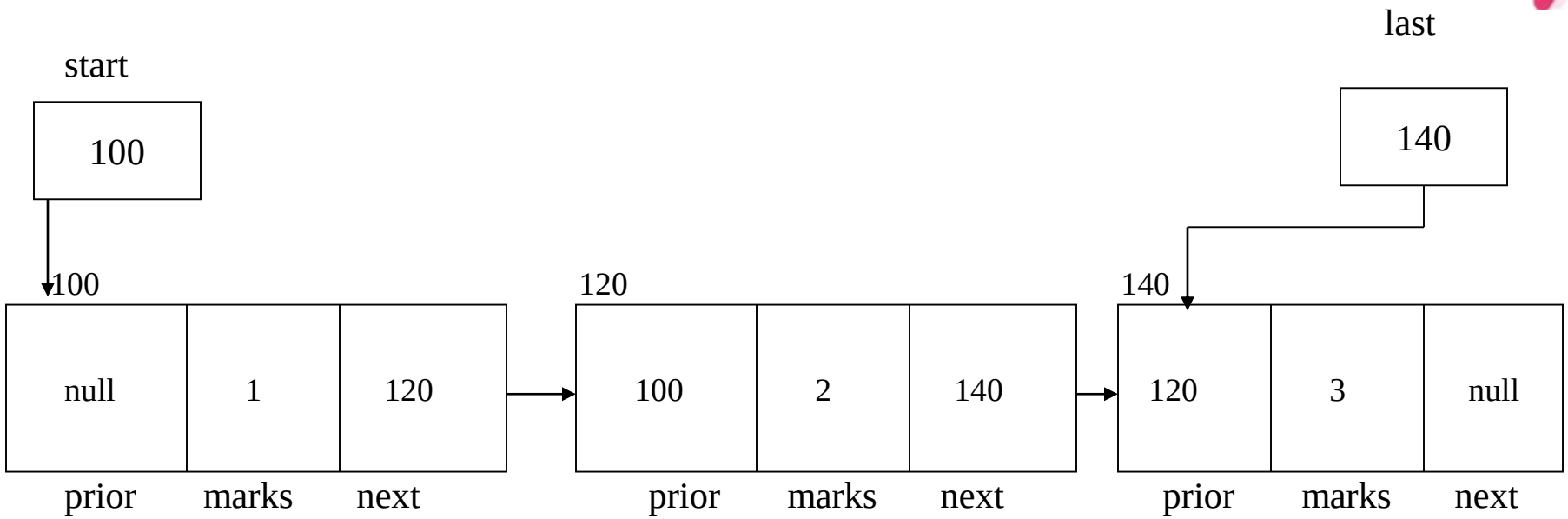
Properties of a Doubly Linked List

- In a doubly linked list, the **prior pointer of the first node will be null**, as there is no node before the first node.
- In a doubly linked list, the **next pointer of the last node will be null**, as there is no node after this list.
- Bidirectional traversal of a doubly linked list is useful for implementing **page up**, and **page down** functionality when using doubly linked lists to create editors.
- A doubly linked list would have two pointers, **start** and **last** to facilitate traversal from the beginning and end of the list respectively.

Declaration of a Doubly Linked List

```
struct marks_list
{
  struct double_list *prior;
  int info;
  struct marks_list *next;
}
```

Visualizing a Doubly Linked List



Creating a Sorted Doubly Linked List

```
#include<stdio.h>
#include<malloc.h>
struct marks_list *start, *last;
/* variables declared outside main() are global in nature and can be accessed by other functions called from main() */
struct marks_list
 {
  struct marks_list *prior;
  int marks;
  struct marks_list *next;
 };
main()
   {
    struct marks_list * makenode();
    /* function  prototype declaration */
```

Creating a Sorted Doubly Linked List

```
struct marks_list *new;
start = NULL;
char menu = '0 ';
while (menu != '5')
 {
   printf( "Add Nodes    :\n");
   printf( "Delete Nodes  :\n");
   printf( "Forward Traverse a list :\n");
   printf( "Reverse Traverse a list :\n");
   printf( "Exit            :\n");
   menu = getchar( );
```

Creating a Sorted Doubly Linked List

```
switch (menu)
 {
   case '1' : addnode( );
              break;
   case '2' : deletenode( )
              break;
   case '3' : forward_traverse( );
              break;
   case '4' : reverse_traverse( );
              break;
   case '5': exit( );
             break;
} /* end of switch */
  } /* end of main( ) */
```

Creating a Sorted Doubly Linked List

```
addnode( )
{
  char ch = 'y';
   while ( ch = = 'y' )
   {
    new = makenode();
    /* creation of a list is treated as a special case of insertion  */
    if ( start = = NULL)
    {
     start = new;
     start->prior =  null;
    }
   else
    {
     insert();
     traverse( );
    }
```

Creating a Sorted Doubly Linked List

```
 printf("%s","Want to add more nodes\n");
 scanf( "%c", &ch );
 fflush( stdin );
} /* end of while  */
} /* end of addnode( )
```

Creating a Sorted Doubly Linked List

```
struct marks_list * makenode()
  {
   struct marks_list *new;
   new=(struct marks_list *)  malloc(sizeof(struct(marks_list));
   scanf("%d",&new->marks);
   new->prior = NULL;
   new->next = NULL;
   return(new);
  }
```

Creating a Sorted Linked List

```
insert( )
{
 struct marks_list *ptr, *prev;
 for(ptr=start,prev=start;(ptr);prev=ptr,ptr=ptr->next)
 {
  if (new->marks < start->marks)
   {
    /* insertion at the beginning of a  list */
    new->next = start;
    new->prior = NULL;
    start->prior = new;
    last = start;
    start = new;
    }
```

Creating a Sorted Doubly Linked List

```
/* insertion in the middle of a list */
  if(new->marks  > ptr->marks)
   {
    continue;
   }
  else
   {
    prev->next = new;
    new->prior = prev;
    new->next = ptr;
    ptr->prior = new;
   }
} /* end of for loop */
```

Creating a Sorted Linked List

```
/* insertion at the end of the list */
if (ptr = = null)
 {
  prev->next = new;
  new->prior = prev;
  new->next = null;
  last = new;
 }
} /* end of insert */
```

Searching a Value in a Doubly Linked List

```
struct marks_list *search( int val)
{
 for( ptr = start; (ptr); ptr = ptr->next)
  {
    if (val = = ptr-> marks)
      return ptr;
  }

struct marks_list *search( int val)
{
 for( ptr = last; (ptr); ptr = ptr->prior)
  {
    if (val = = ptr-> marks)
      return ptr;
  }
```

Deleting a Node From a Doubly Linked List

```
delete ( )
 {
  struct marks_list *ptr, *prev, *temp;
  int score;
   /* search the linked list for the value to be deleted */
   scanf(“%d”, &score);
  fflush(stdin);
  for (ptr = start, prev = start; (ptr); prev = ptr, ptr = ptr->next)
  {
   /* deletion of the first node in the list */
   if (score = = start-> marks)
   {
    temp =start;
    start = start-> next;
    start->prior = null;
    free(temp);
   }
```

Deleting a Node From a Linked List

```
/* deletion in the middle of a linked list */
 if (score = = ptr-> marks)
   {
    prev-> next = ptr-> next;
    ptr->next->prior = ptr->prior;
    free(ptr);
   }
/* deletion at the end of the list */
 if (ptr->next = = null)
 {
  temp = ptr;
  prev->next = null;
  last = ptr->prior;
 }
}}
```

Traversal of a Doubly Linked List

```
/* forward traversal of a doubly linked list */
for( ptr = start; (ptr); ptr = ptr->next)
{
  printf("%d", ptr->marks);
}

/* reverse traversal of a doubly linked list */
for( ptr = last; (ptr); ptr = ptr->prior)
{
  printf("%d", ptr->marks);
}
```

Binary Trees

Eliminative or a Binary Search

The mode of accessing data in a linked list is linear.

Therefore, in the worst case scenario of the data in question being stored at the extremes of the list, it would involve starting with the first node, and traversing through all the nodes till one reaches the last node of the list to access the data.

Therefore, **search through a linked list is always linear**.

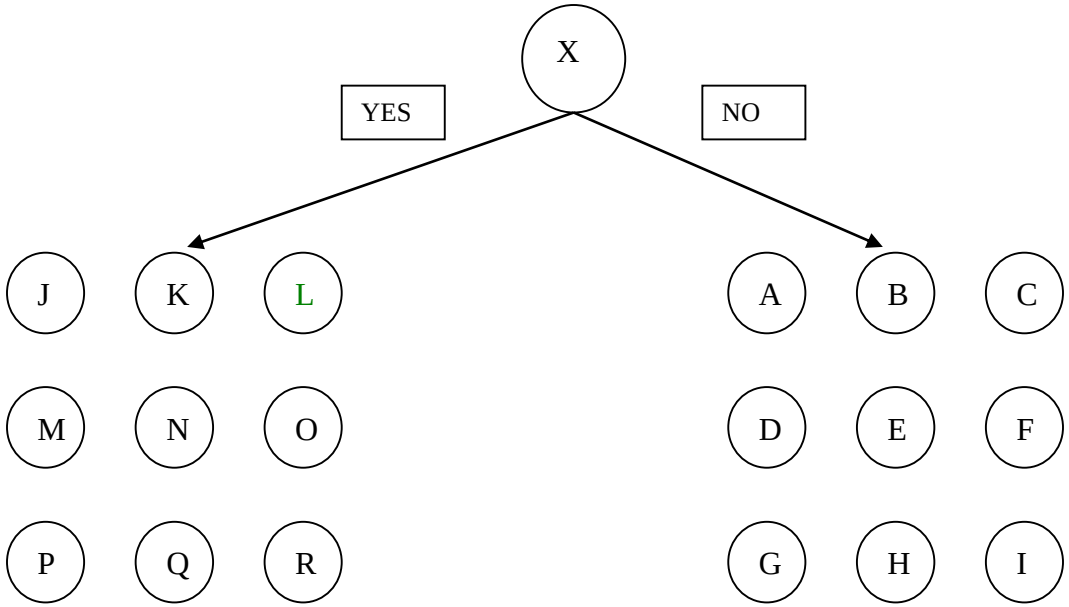
Eliminative or a Binary Search

- A linear search is fine if the nodes to be searched in a linked list are small in number.
- But the linear search becomes ineffective as the number of nodes in a linked list increase.
- The search time increases in direct proportion with the size of the linked list.
- It becomes imperative to have better searching mechanisms than a linear search.

Eliminative or a Binary Search

You will now be exposed to a game that will highlight a new mechanism of searching.

Coin Search in a Group



Eliminative or a Binary Search

- A coin is with one of the members in the audience divided into the two sections on the left and the right respectively as shown in the diagram.

- The challenge facing X, the protagonist, is to find the person with the coin in the least number of searches.

Employing the Linear Search

- X, familiar with a linear search, starts using it to search for the coin among the group.
- Let us assume the worst-case scenario of the coin being with R.
- If X was to start the search with A and progress linearly through B, C, ….M and finally to R, he would have taken a minimum of 18 searches (R being the 18th person searched in sequence to find the coin.

Employing the Linear Search

- As you can see, this kind of search is not very efficient especially when the number of elements to be searched is high.
- An eliminative search, also called a binary search, provides a far better searching mechanism.

Employing the Eliminative or The Binary Search

- Assume that X can pose intelligent questions to the audience to cut down on the number of searches.
- A valid question that he could pose is “Which side of the audience has the coin?”
- ‘A’ in the audience to the left of him says that his side of the audience does not have the coin.

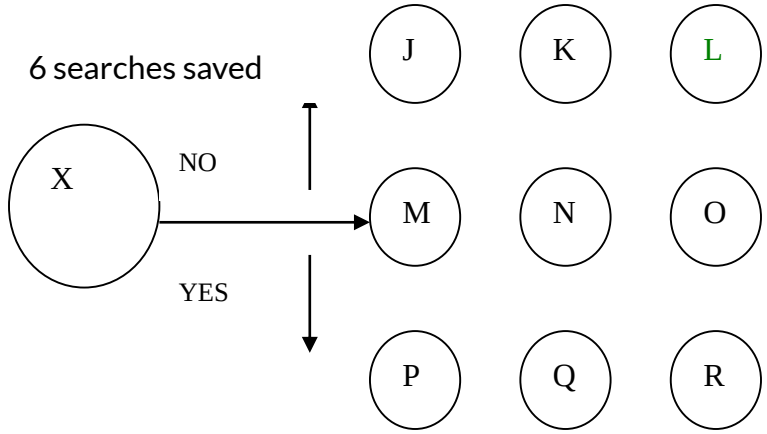
So X can completely do away with searching the audience to his left. That saves him 9 searches.

Employing the Eliminative or The Binary Search

- X has now got to search the audience to the right of him for searching out the coin.
- Here too, he can split the audience into half by standing adjacent to the middle row, and posing the same question that he asked earlier “Which side of the audience has the coin?”
- ‘M’ in the middle row replies that the coin is with the audience to the left of him.

Employing the Eliminative or The Binary Search

X has in the process now eliminated 6 more searches, that is, the middle row and the row to the left of the middle as shown in the diagram below.

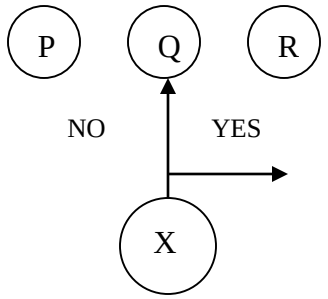


Employing the Eliminative or The Binary Search

- X is now left with row to the right of the middle row containing P, Q, and R that has to be searched.

- Here too, he can position himself right at the middle of the row adjacent to Q and pose the same question,

- “Which side of the audience has the coin?” ‘Q’ replies that the coin is to his left.



Eliminative or Binary Search

- That completes our eliminative or binary search.

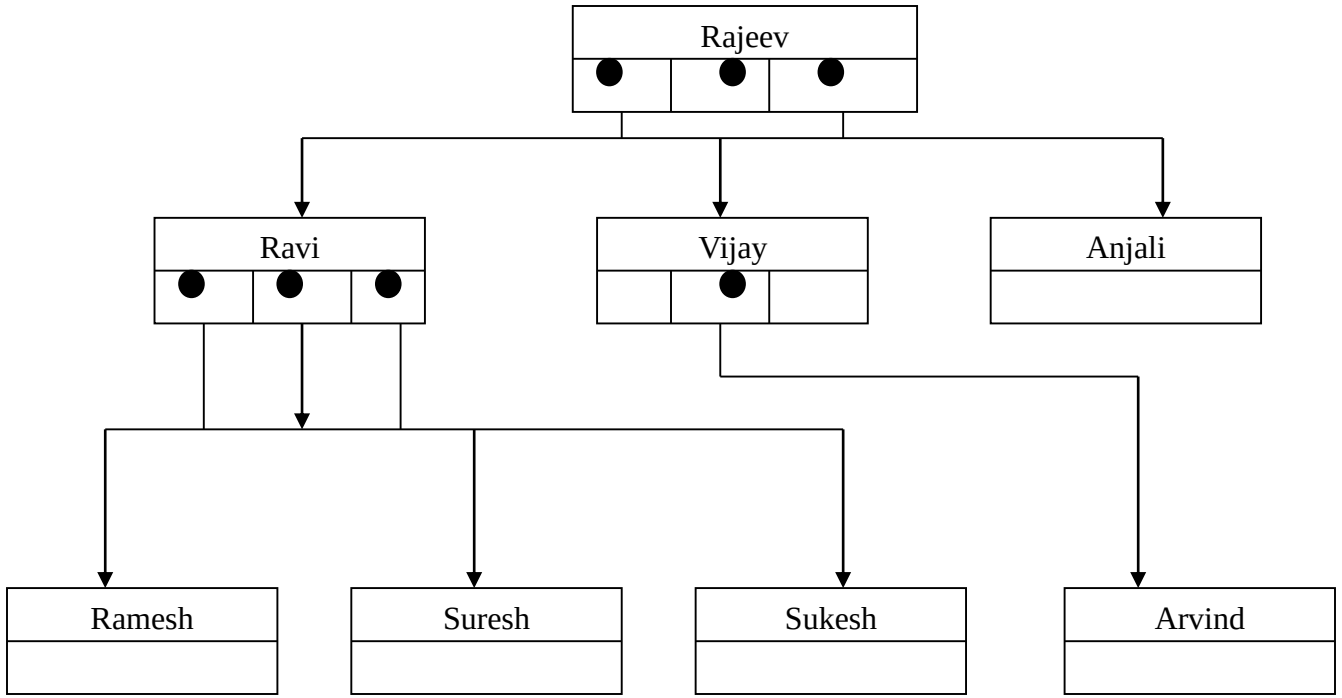
It is called a binary search because at each stage of the search, the search is cut by more than half.

- This kind of search forms the basis for the searching mechanism employed in a data structure wherein the data is represented in a hierarchal manner unlike the linear mechanism of storage employed in a linked list.

Trees

- Compared to linked lists that are linear data structures, **trees are non-linear data structures**.
- In a linked list, each node has a link which points to another node.
- In a tree structure, however, each node may point to several nodes, which may in turn point to several other nodes.
- Thus, a tree is a very flexible and a powerful data structure that can be used for a wide variety of applications.

Trees

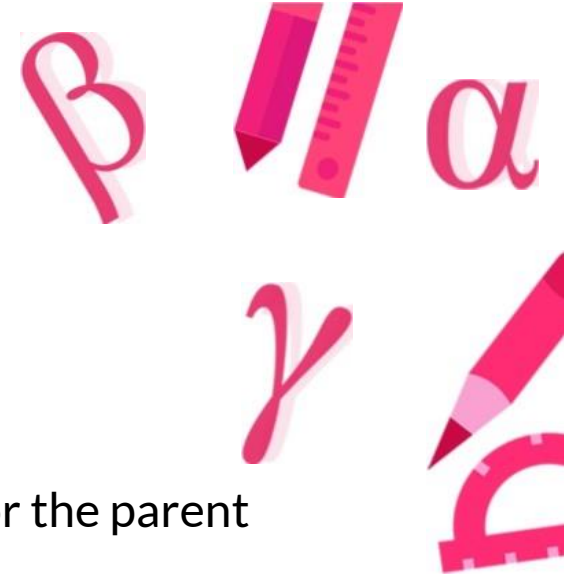


Trees

- A tree consists of a collection of nodes that are connected to each other.
- **A tree contains a unique first element known as the root, which is shown at the top of the tree structure.**
- A node which points to other nodes is said to be the parent of the nodes to which it is pointing, and the nodes that the parent node points to are called the children, or child nodes of the parent node.

Trees

- The root is the only node in the tree that does not have a parent.
- All other nodes in the tree have exactly one parent.
- There are nodes in the tree that do not have any children. Such nodes are called leaf nodes.
- Nodes are siblings if they have the same parent.



Trees

- A node is an ancestor of another node if it is the parent of that node, or the parent of some other ancestor of that node.
- The root is an ancestor of every other node in the tree.
- Similarly, we can define a node to be a descendant of another node if it is the child of the node, or the child of some other descendant of that node.
- You may note that all the nodes in the tree are descendants of the root node.

Tree

- An important feature of a tree is that there is a single unique path from the root to any particular node.
- The length of the longest path from the root to any node is known as the depth of the tree.
- The root is at level 0 and the level of any node in the tree is one more than the level of its parent.
- In a tree, any node can be considered to be a root of the tree formed by considering only the descendants of that node. Such a tree is called the subtree that itself is a tree.

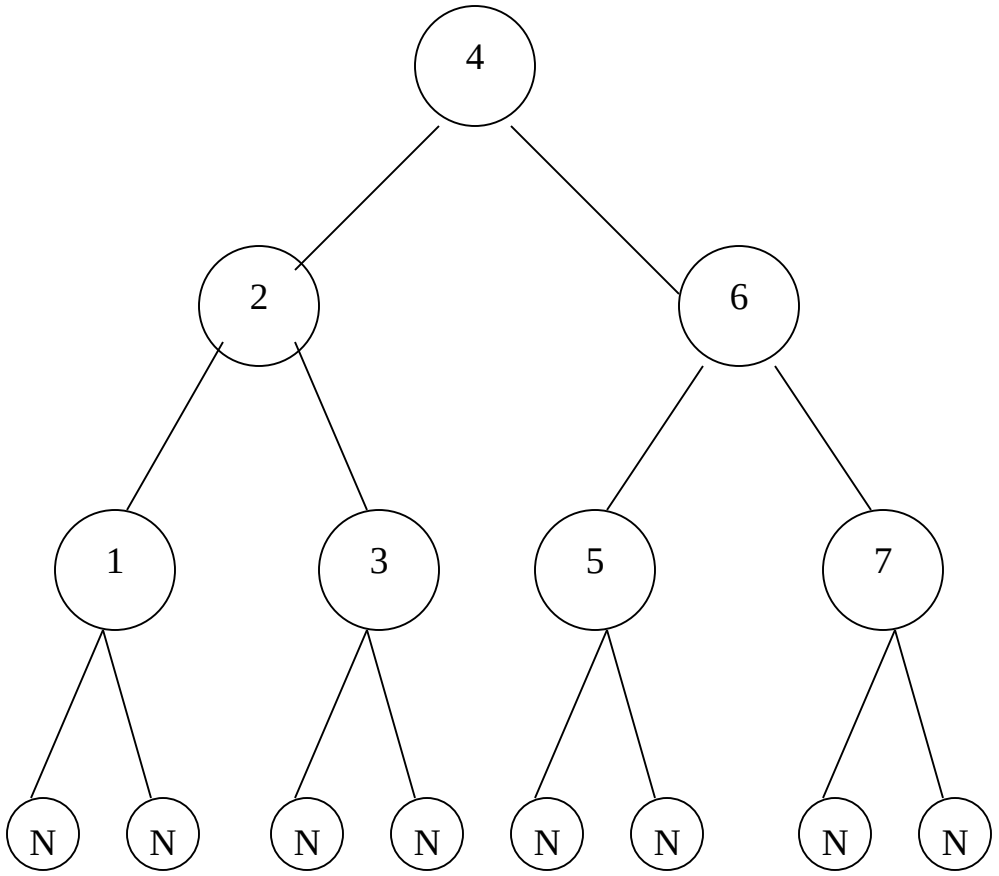
Binary Tree

- If you can introduce a restriction that each node can have a maximum of two children or two child nodes, then you can have a binary tree.
- You can give a formal definition of a binary tree as a tree which is either empty or consists of a root node together with two nodes, each of which in turn forms a subtree.
- You therefore have a left subtree and a right subtree under the root node.

Binary Search Tree

- A complete binary tree can be defined as one whose non-leaf nodes have non-empty left and right subtrees and all leaves are at the same level.
- This is also called as a balanced binary tree.
- If a binary tree has the property that all elements in the left subtree of a node n are less than the contents of n, and all elements in the right subtree are greater than the contents of n, such a tree is called a binary search tree.
- The following is an example of a balanced binary search tree.

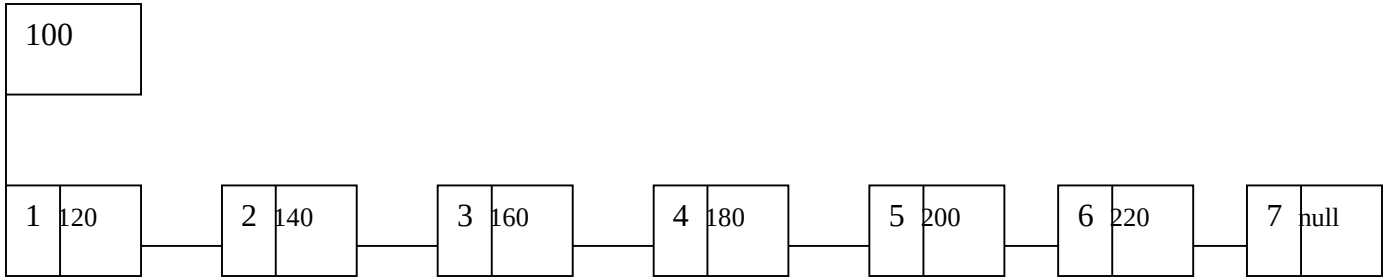
Balanced Binary Search Tree



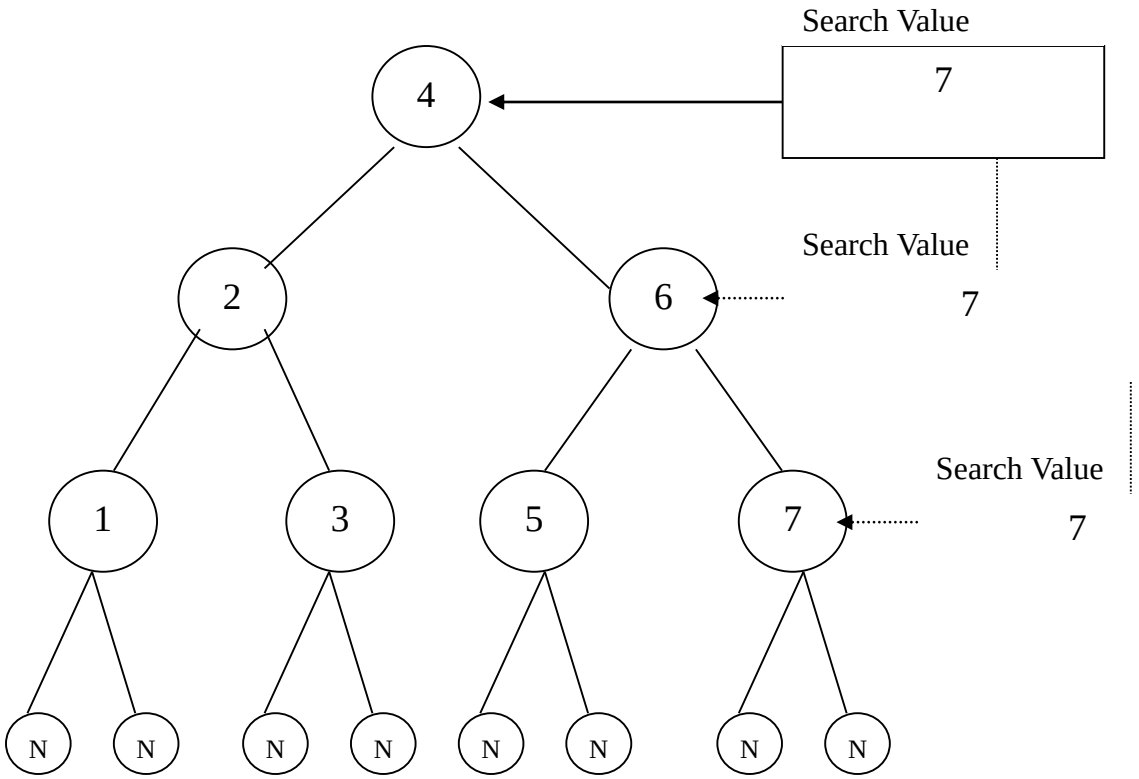
Binary Vs. Linear Search

- As the name suggests, balanced binary search trees are very useful for searching an element just as with a binary search.
- If we use linked lists for searching, we have to move through the list linearly, one node at a time.
- If we search an element in a binary search tree, we move to the left subtree for smaller values, and to the right subtree for larger values, every time reducing the search list by half approximately.

Linear Search in a Linked List



Binary Search in a Binary Search Tree



The Essence of a Binary Search

- To summarize, you have completely done away with searching with the entire left subtree of the root node and its descendant subtrees, in the process doing away with searching one-half of the binary search tree.
- Even while searching the right subtree of the root node and its descendant subtrees, we keep searching only one-half of the right subtree and its descendants.
- This is more because of the search value in particular, which is 7. The left subtree of the right subtree of the root could have been searched in case the value being searched for was say 5.

The Essence of a Binary Search

- Thus we can conclude that while searching for a value in a balanced binary search tree, the number of searches is cut by more than half (3 searches in a balanced binary search tree) compared to searching in a linked list (7 searches).

Thus a search that is hierarchical, eliminative and binary in nature is far efficient when compared to a linear search.

Data Structure Representation of a Binary Trees

- A tree node may be implemented through a structure declaration whose elements consist of a variable for holding the information and also consist of two pointers, one pointing to the left subtree and the other pointing to the right subtree.
- A binary tree can also be looked at as a special case of a doubly linked list that is traversed hierarchically.
- The following is the structure declaration for a tree node:

Data Structure Representation of a Binary Trees

```
struct btreenode
{
  int info;
  struct btreenode *left;
  struct btreenode *right;
};
```

Traversing a Binary Tree

- Traversing a binary tree entails visiting each node in the tree exactly once.
- Binary tree traversal is useful in many applications, especially those involving an indexed search.
- Nodes of a binary search tree are traversed hierarchically.
- The methods of traversing a binary search tree differ primarily in the order in which they visit the nodes.

Traversing a Binary Tree

- At a given node, there are three things to do in some order. They are:
  - ✓ To visit the node itself
  - ✓ To traverse its left subtree
  - ✓ To traverse its right subtree
- We can traverse the node before traversing either subtree.
- Or, we can traverse the node between the subtrees.
- Or, we can traverse the node after traversing both subtrees.

Traversing a Binary Tree

- If we designate the task of visiting the root as R', traversing the left subtree as L and traversing the right subtree as R, then the three modes of tree traversal discussed earlier would be represented as:
  - ✓ R'LR – Preorder
  - ✓ LRR' – Postorder
  - ✓ LR'R – Inorder

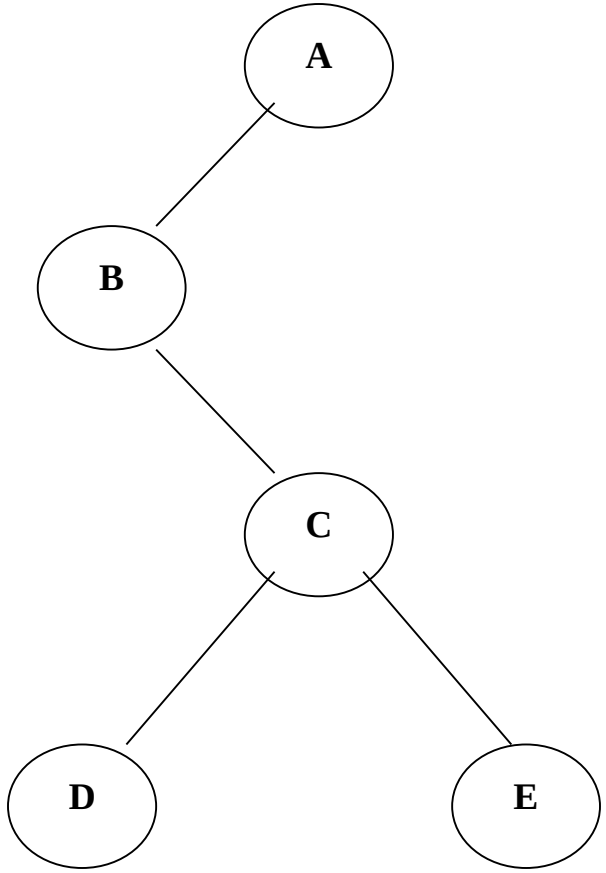
Traversing a Binary Tree

- The functions used to traverse a binary tree using these methods can be kept quite short if we understand the recursive nature of the binary tree.
- Recall that a binary tree is recursive in that each subtree is really a binary tree itself.
- Thus traversing a binary tree involves visiting the root node, and traversing its left and right subtrees.
- The only difference among the methods is the order in which these three operations are performed.

Traversing a Binary Tree

- Depending on the position at which the given node or the root is visited, the name is given.
- If the root is visited before traversing the subtree, it is called the preorder traversal.
- If the root is visited after traversing the subtrees, it is called postorder traversal.
- If the root is visited in between the subtrees, it is called the inorder traversal.

Traversing a Binary Tree



Preorder Traversal

- When we traverse the tree in preorder, the root node is visited first. So, the node containing A is traversed first.
- Next, we traverse the left subtree. This subtree must again be traversed using the preorder method.
- Therefore, we visit the root of the subtree containing B and then traverse its left subtree.
- The left subtree of B is empty, so its traversal does nothing. Next we traverse the right subtree that has root labeled C.

Preorder Traversal

- Then, we traverse the left and right subtrees of C getting D and E as a result.
- Now, we have traversed the left subtree of the root containing A completely, so we move to traverse the right subtree of A.
- The right subtree of A is empty, so its traversal does nothing. Thus the preorder traversal of the binary tree results in the values **ABCDE.**

Inorder Traversal

- For inorder traversal, we begin with the left subtree rooted at B of the root.
- Before we visit the root of the left subtree, we must visit its left subtree, which is empty.
- Hence the root of the left subtree rooted at B is visited first. Next, the right subtree of this node is traversed inorder.

Inorder Traversal

- Again, first its left subtree containing only one node D is visited, then its root C is visited, and finally the right subtree of C that contains only one node E is visited.
- After completing the left subtree of root A, we must visit the root A, and then traverse its right subtree, which is empty.
- Thus, the complete inorder traversal of the binary tree results in values **BDCEA.**

Postorder Traversal

- For postorder traversal, we must traverse both the left and the right subtrees of each node before visiting the node itself.
- Hence, we traverse the left subtree in postorder yielding values D, E, C and B.
- Then we traverse the empty right subtree of root A, and finally we visit the root which is always the last node to be visited in a postorder traversal.
- Thus, the complete postorder traversal of the tree results in **DECBA.**

Code - Preorder Traversal

```
void preorder (p)
struct btreenode *p;
 {
  /* Checking for an empty tree */
  if ( p != null)
   {
    /* print the value of the root node */
    printf("%d", p->info);
    /* traverse its left subtree */
    preorder(p->left);
    /* traverse its right subtree */
    preorder(p->right);
   }
 }
```

Code – Inorder Traversal

```
void inorder(p)
struct btreenode *p;
{
 /* checking for an empty tree */
 if (p != null)
  {
   /* traverse the left subtree inorder */
   inorder(p->left);
   /* print the value of the root node */
   printf("%d", p->info);
   /*traverse the right subtree inorder */
   inorder(p->right);
  }
}
```

Code – Postorder Traversal

```
void postorder(p)
struct btreenode *p;
{
 /* checking for an empty tree */
 if (p != null)
  {
   /* traverse the left subtree */
   postorder(p->left);
   /* traverse the right subtree */
   postorder(p->right);
   /* print the value of the root node */
   printf("%d", p->info);
  }
}
```

Accessing Values From a Binary Search Tree Using Inorder Traversal

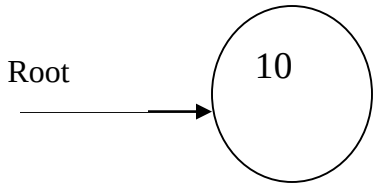
- You may note that when you traverse a binary search tree inorder, the keys will be in sorted order because all the keys in the left subtree are less than the key in the root, and all the keys in the right subtree are greater than that in the root.
- The same rule applies to all the subtrees until they have only one key.
- Therefore, given the entries, we can build them into a binary search tree and use inorder traversal to get them in sorted order.

Insertion into a Tree

- Another important operation is to create and maintain a binary search tree.
- While inserting any node, we have to take care the resulting tree satisfies the properties of a binary search tree.
- A new node will always be inserted at its proper position in the binary search tree as a leaf.
- Before writing a routine for inserting a node, consider how a binary tree may be created for the following input: 10, 15, 12, 7, 8, 18, 6, 20.

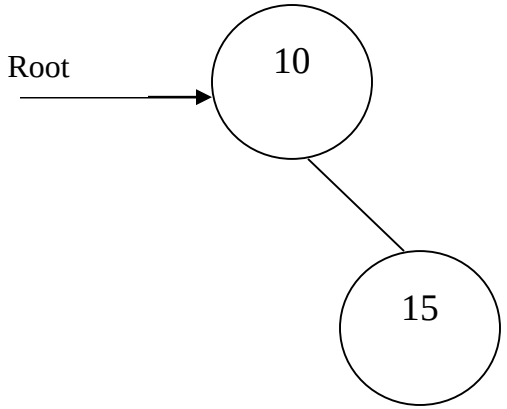
Insertion into a Tree

- First of all, you must initialize the tree.
- To create an empty tree, you must initialize the root to null. The first node will be inserted into the tree as a root node as shown in the following figure.



Insertion into a Tree

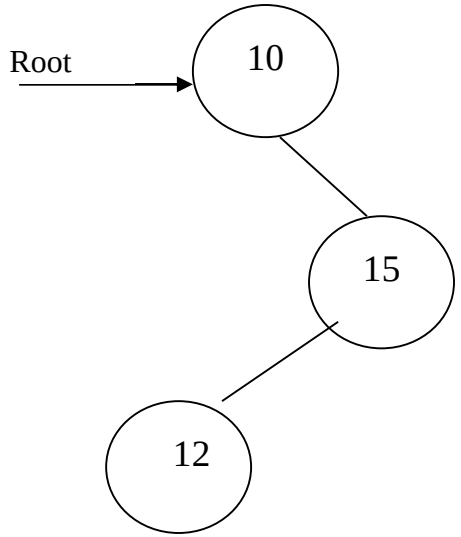
Since 15 is greater than 10, it must be inserted as the right child of the root as shown in the following figure.



Insertion into a Tree

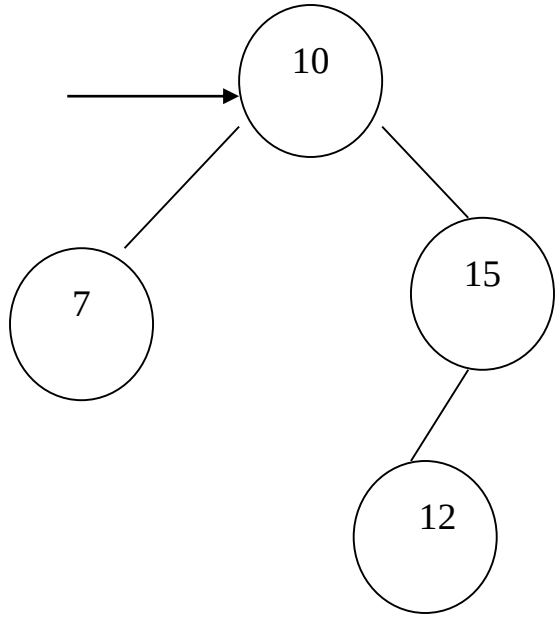
Now 12 is larger than the root; it must go to the right subtree of the root.

Further, since it is smaller than 15, it must be inserted as the left child of the root as shown below.



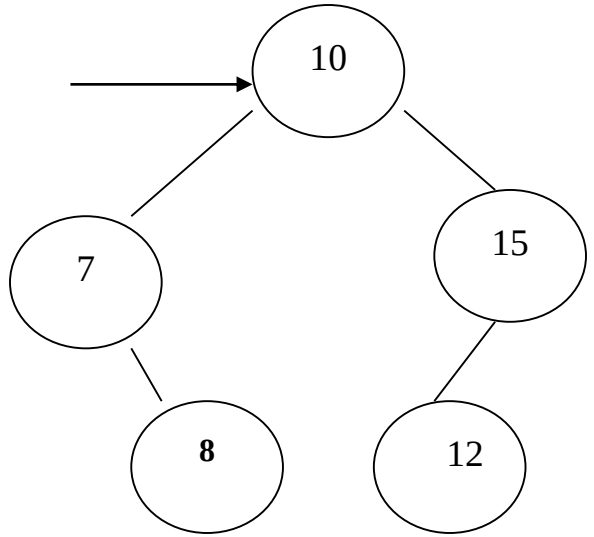
Insertion into a Tree

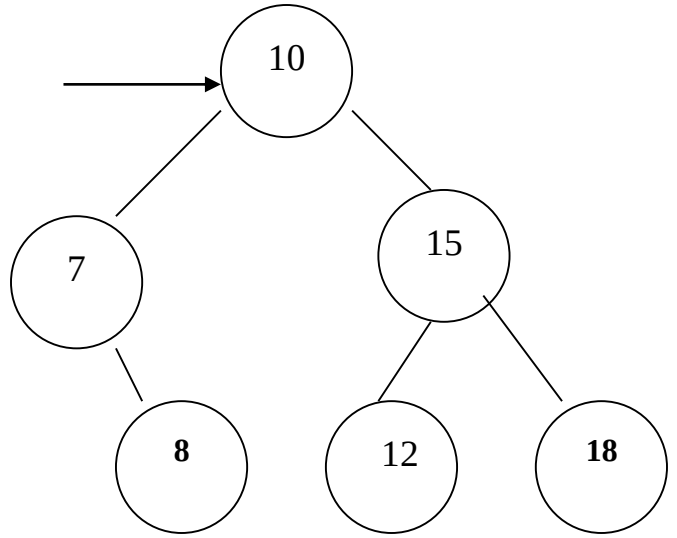
Next, 7 is smaller than the root. Therefore, it must be inserted as the left child of the root as shown in the following figure.



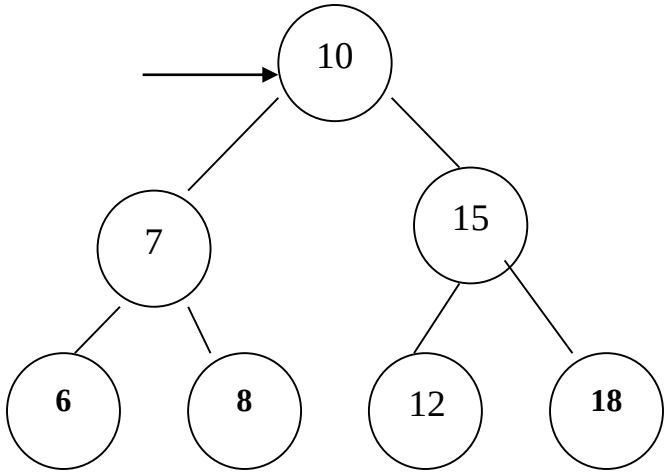
Insertion into a Tree

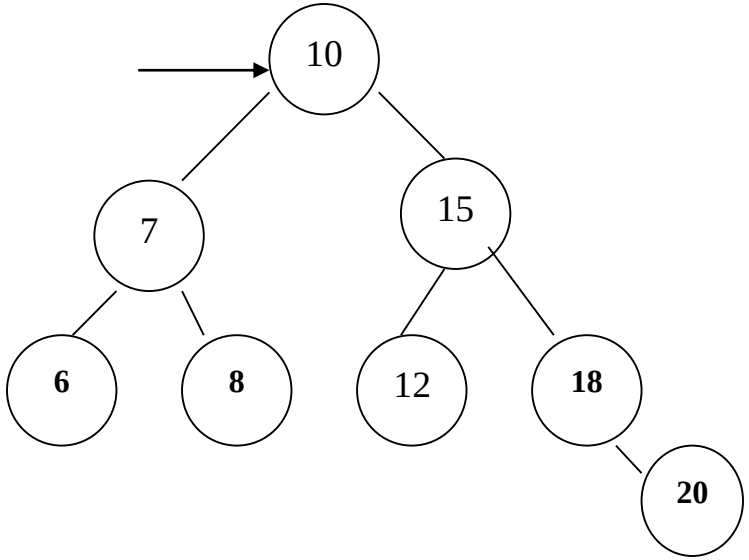
Similarly, 8, 18, 6 and 20 are inserted at the proper place as shown in the following figures.





Insertion into a Tree





Insertion into a Tree

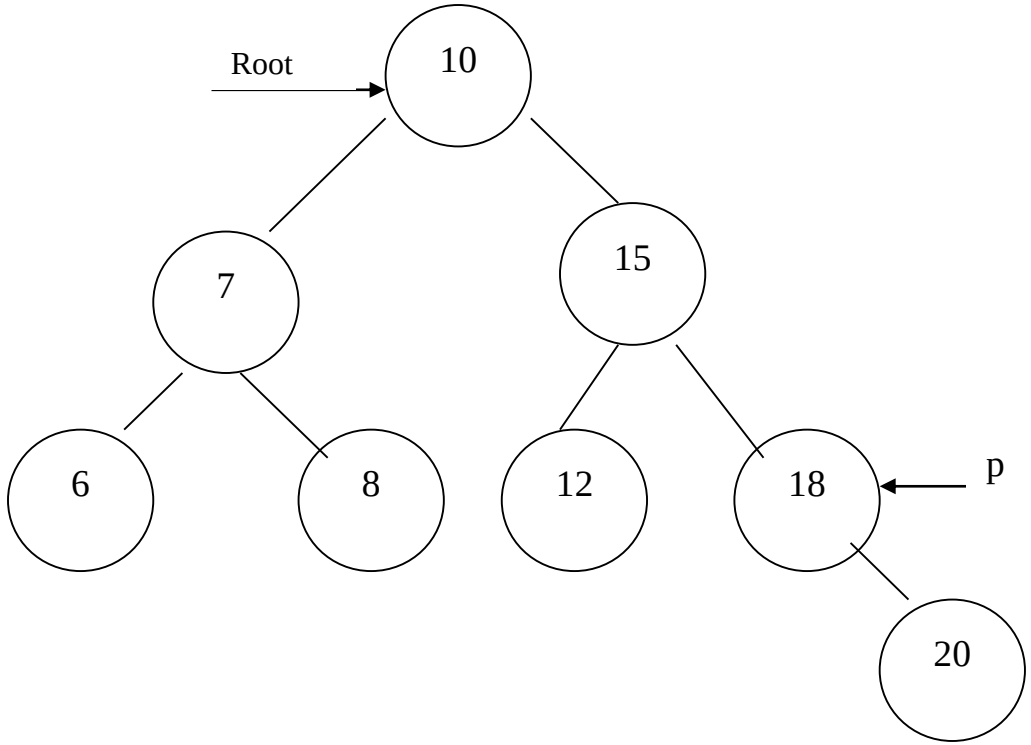
- This example clearly illustrates that given the root of a binary search tree and a value to be added to the tree, we must search for the proper place where the new value can be inserted.
- We must also create a node for the new value and finally, we have to adjust the left and right pointers to insert the new node.
- To find the insertion place for the new value, say 17, we initialize a temporary pointer p, which points to the root node.

Insertion into a Tree

- We can change the contents of p to either move left or right through the tree depending on the value to be inserted.

- When p becomes null, we know that we have found the insertion place as in the following figure.

Insertion into a Tree

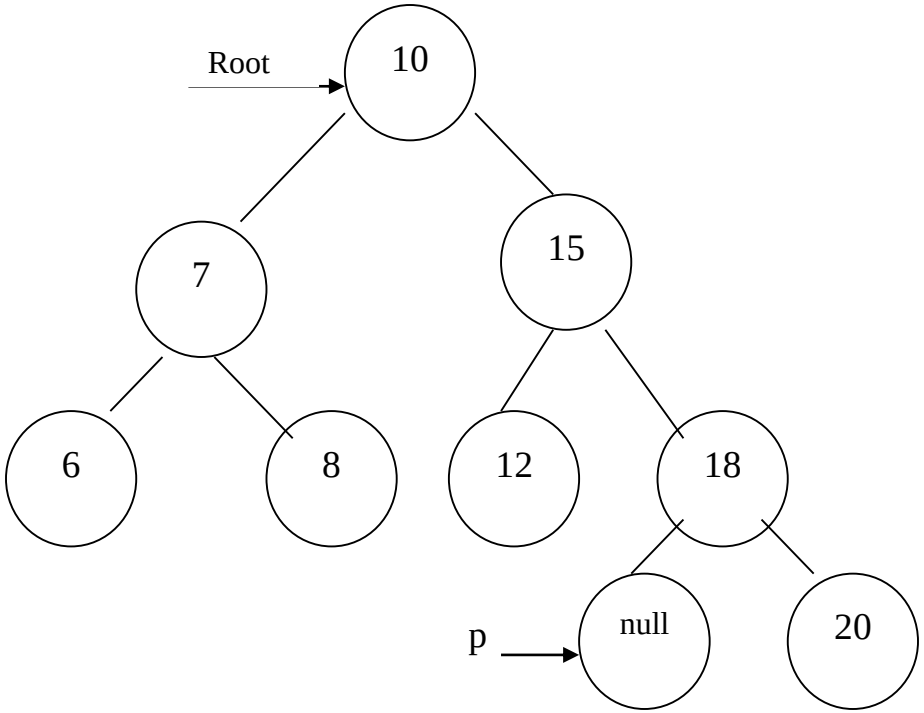


Insertion into a Tree

- But once p becomes null, it is not possible to link the new node at this position because there is no access to the node that p was pointing to (node with value 18) just before it became null.

- From the following figure, p becomes null when we have found that 17 will be inserted at the left of 18.

Insertion into a Tree



Insertion into a Tree

- You therefore need a way to climb back into the tree so that you can access the node containing 18, in order to make its left pointer point to the new node with the value 17.
- For this, you need a pointer that points to the node containing 18 when p becomes null.
- To achieve this, you need to have another pointer (trail) that must follow p as p moves through the tree.

Insertion into a Tree

- When p becomes null, this pointer will point to the leaf node (the node with value 18) to which you must link the new node (node with value 17).
- Once you know the insertion place, you must adjust the pointers of the new node.
- At this point, you only have a pointer to the leaf node to which the new node is to be linked.
- You must determine whether the insertion is to be done at the left subtree or the right subtree of the leaf node.

Insertion into a Tree

- To do that, you must compare the value to be inserted with the value in the leaf node.
- If the value in the leaf node is greater, we insert the new node as its left child; otherwise we insert the new node as its right child.

Creating a Tree – A Special Case of Insertion

- A special case of insertion that you need to watch out for arises when the tree in which you are inserting a node is an empty tree.
- You must treat it as a special case because when p equals null, the second pointer (trail) trailing p will also be null, and any reference to info of trail like trail->info will be illegal.
- You can check for an empty tree by determining if trail is equal to null. If that is so, we can initialize root to point to the new node.

Code Implementation For Insertion Into a Tree

- The C function for insertion into a binary tree takes two parameters; one is the pointer to the root node (root), and the other is the value to be inserted (x).

- You will implement this algorithm by allocating the nodes dynamically and by linking them using pointer variables. The following is the code implementation of the insert algorithm.

Code Implementation For Insertion Into a Tree

```
tree insert(s,x)
int x;
tree *s;
{
 tree *trail, *p, *q;
 q = (struct tree *) malloc (sizeof(tree));
 q->info = x;
 q->left = null;
 q->right = null;
 p = s;
 trail = null;
```

Code Implementation For Insertion Into a Tree

```
while (p != null)
{
 trail = p;
 if (x < p->info)
 {
  p = p->left;
 }
 else
 {
  p = p->right;
 }
}
```

Code Implementation For Insertion Into a Tree

```
/*insertion into an empty tree; a special case of insertion */
if (trail == null)
{
 s = q;
 return (s);
}
if(x < trail->info)
{
 trail->left = q;
}
else
{
 trail->right = q;
}
return (s);
}
```

Code Implementation For Insertion Into a Tree Using Recursion

- You have seen that to insert a node, you must compare x with root->info.
- If x is less than root->info, then x must be inserted into the left subtree.
- Otherwise, x must be inserted into the right subtree.
- This description suggests a recursive method where you compare the new value (x) with the one in the root and you use exactly the same insertion method either on the left subtree or on the right subtree.

Code Implementation For Insertion Into a Tree Using Recursion

The base case is inserting a node into an empty tree.

You can write a recursive routine (rinsert) to insert a node recursively as follows:

```
tree rinsert (s,x)
tree *s;
int x;
 {
  /* insertion into an empty tree; a special case
  of insertion */
  if (!s)
   {
   s=(struct tree*) malloc (sizeof(struct tree));
```

Code Implementation For Insertion Into a Tree Using Recursion

```
s->info = x;
  s->left = null;
  s->right = null;
  return (s);
 }
 if (x < s->info)
  s->left = rinsert(x, s->left);
 else
  if (x > s->info)
  s->right = rinsert(x, s->right);
return (s);
}
```

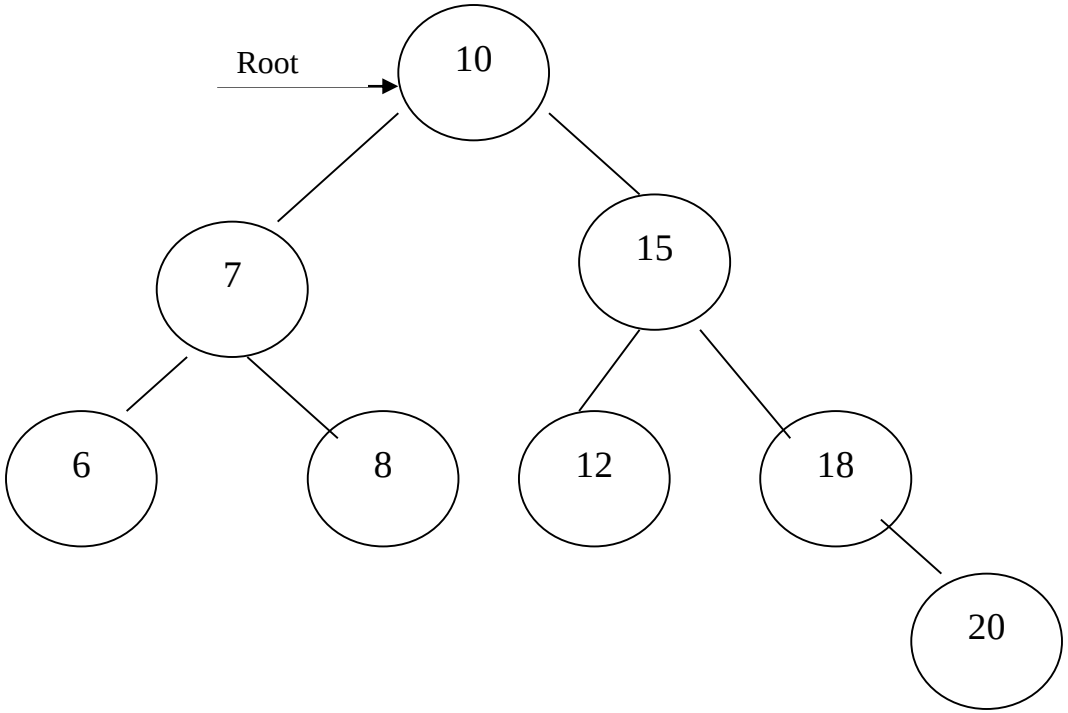
Circumstances When a Binary Tree Degenerates Into a Linked List

- The shape of a binary tree is determined by the order in which the nodes are inserted.

- Given the following input, their insertion into the tree in the same order would more or less produce a balanced binary search tree as shown below:

  Input values: 10, 15, 12, 7, 8, 18, 6, 20

Circumstances When a Binary Tree Degenerates Into a Linked List

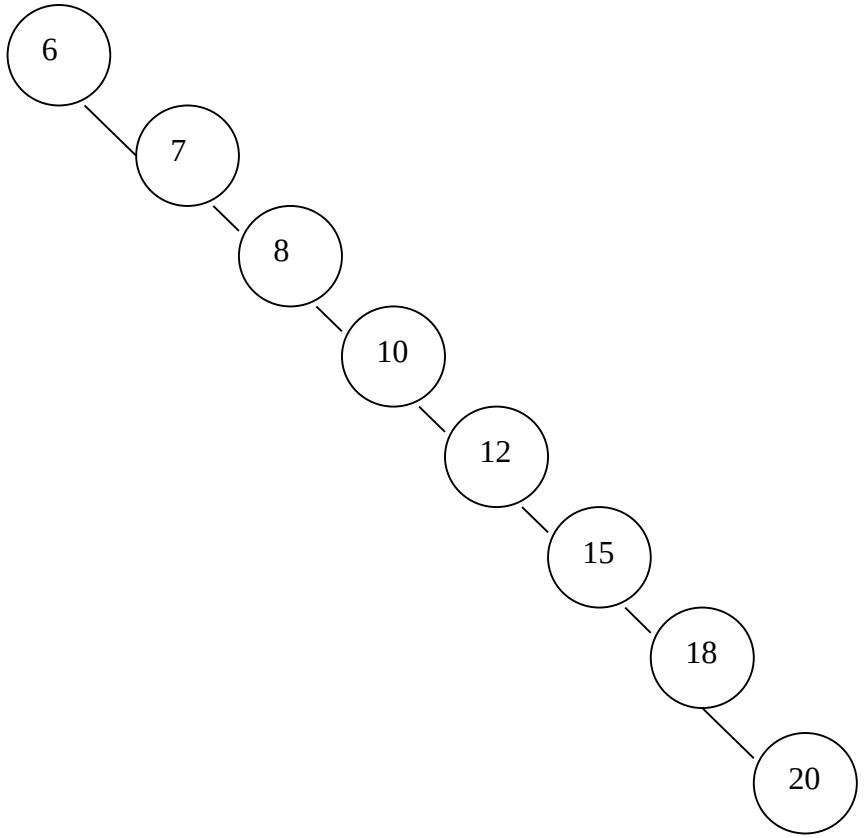


Circumstances When a Binary Tree Degenerates Into a Linked List

- If the same input is given in the sorted order as
- 6, 7, 8, 10, 12, 15, 18, 20, you will construct a lopsided tree with only right subtrees starting from the root.

- Such a tree will be conspicuous by the absence of its left subtree from the top.

Circumstances When a Binary Tree Degenerates Into a Linked List



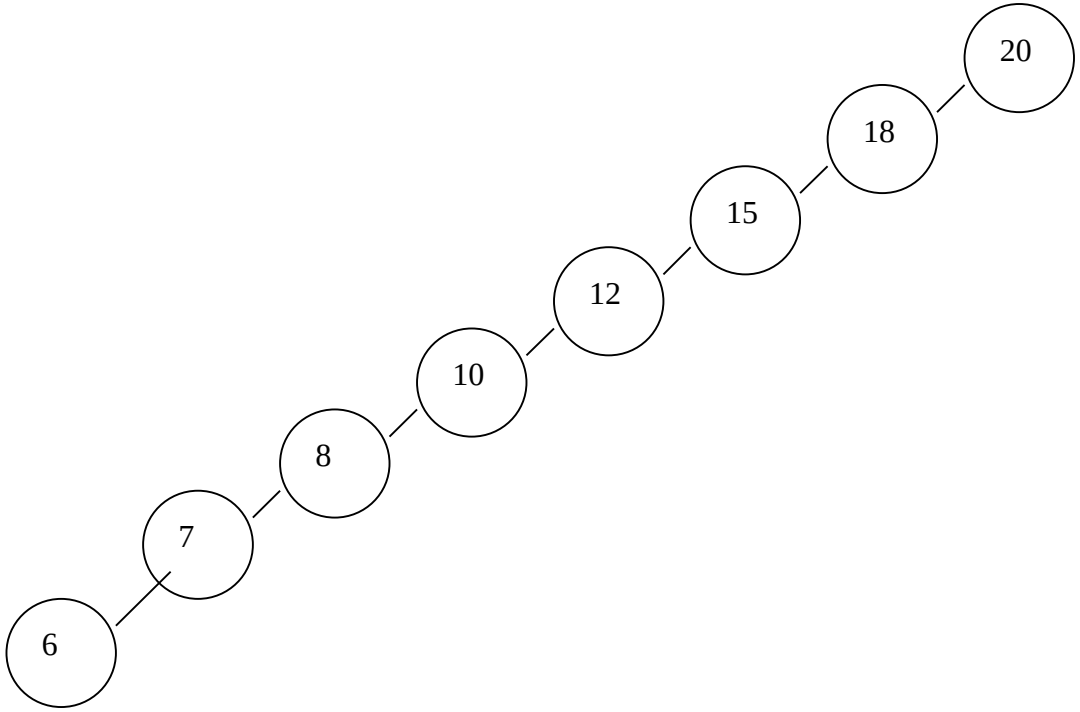
A Lopsided Binary Tree With Only Right Subtrees

Circumstances When a Binary Tree Degenerates Into a Linked List

- However if you reverse the input as
- 20, 18, 15, 12, 10, 8, 7, 6, and insert them into a tree in the same sequence, you will construct a lopsided tree with only the left subtrees starting from the root.

- Such a tree will be conspicuous by the absence of its right subtree from the top.

Circumstances When a Binary Tree Degenerates Into a Linked List



A Lopsided Binary Tree With Only Left Subtrees

Search The Tree

- To search a tree, you employ a traversal pointer p, and set it equal to the root of the tree.
- Then you compare the information field of p with the given value x. If the information is equal to x, you exit the routine and return the current value of p.
- If x is less than p->info, you search in the left subtree of p.
- Otherwise, you search in the right subtree of p by making p equal to p->right.

Search the Tree

You continue searching until you have found the desired value or reach the end of the tree. You can write the code implementation for a tree search as follows:

```
search (p,x)
int x;
struct tree *p;
 {
  p = root;
  while (p != null && p->info != x)
   {
    if (p->info > x)
     p = p->left;
    else
     p = p->right;
   }  return (p);
  }
```

Thank You!